Bioinformatics Toolbox™

Reference

MATLAB®

R2023a

MathWorks®

How to Contact MathWorks

Latest news:	`www.mathworks.com`
Sales and services:	`www.mathworks.com/sales_and_services`
User community:	`www.mathworks.com/matlabcentral`
Technical support:	`www.mathworks.com/support/contact_us`
Phone:	508-647-7000

The MathWorks, Inc.
1 Apple Hill Drive
Natick, MA 01760-2098

Bioinformatics Toolbox™ Reference

Revision History

May 2005	Online only	New for Version 2.1 (Release 14SP2+)
September 2005	Online only	Revised for Version 2.1.1 (Release 14SP3)
November 2005	Online only	Revised for Version 2.2 (Release 14SP3+)
March 2006	Online only	Revised for Version 2.2.1 (Release 2006a)
May 2006	Online only	Revised for Version 2.3 (Release 2006a+)
September 2006	Online only	Revised for Version 2.4 (Release 2006b)
March 2007	Online only	Revised for Version 2.5 (Release 2007a)
April 2007	Online only	Revised for Version 2.6 (Release 2007a+)
September 2007	Online only	Revised for Version 3.0 (Release 2007b)
March 2008	Online only	Revised for Version 3.1 (Release 2008a)
October 2008	Online only	Revised for Version 3.2 (Release 2008b)
March 2009	Online only	Revised for Version 3.3 (Release 2009a)
September 2009	Online only	Revised for Version 3.4 (Release 2009b)
March 2010	Online only	Revised for Version 3.5 (Release 2010a)
September 2010	Online only	Revised for Version 3.6 (Release 2010b)
April 2011	Online only	Revised for Version 3.7 (Release 2011a)
September 2011	Online only	Revised for Version 4.0 (Release 2011b)
March 2012	Online only	Revised for Version 4.1 (Release 2012a)
September 2012	Online only	Revised for Version 4.2 (Release 2012b)
March 2013	Online only	Revised for Version 4.3 (Release 2013a)
September 2013	Online only	Revised for Version 4.3.1 (Release 2013b)
March 2014	Online only	Revised for Version 4.4 (Release 2014a)
October 2014	Online only	Revised for Version 4.5 (Release 2014b)
March 2015	Online only	Revised for Version 4.5.1 (Release 2015a)
September 2015	Online only	Revised for Version 4.5.2 (Release 2015b)
March 2016	Online only	Revised for Version 4.6 (Release 2016a)
September 2016	Online only	Revised for Version 4.7 (Release 2016b)
March 2017	Online only	Revised for Version 4.8 (Release 2017a)
September 2017	Online only	Revised for Version 4.9 (Release 2017b)
March 2018	Online only	Revised for Version 4.10 (Release 2018a)
September 2018	Online only	Revised for Version 4.11 (Release 2018b)
March 2019	Online only	Revised for Version 4.12 (Release 2019a)
September 2019	Online only	Revised for Version 4.13 (Release 2019b)
March 2020	Online only	Revised for Version 4.14 (Release 2020a)
September 2020	Online only	Revised for Version 4.15 (Release 2020b)
March 2021	Online only	Revised for Version 4.15.1 (Release 2021a)
September 2021	Online only	Revised for Version 4.15.2 (Release 2021b)
March 2022	Online only	Revised for Version 4.16 (Release 2022a)
September 2022	Online only	Revised for Version 4.16.1 (Release 2022b)
March 2023	Online only	Revised for Version 4.17 (Release 2023a)

Contents

Functions and Apps

aa2int

Convert amino acid sequence from letter to integer representation

Syntax

```
SeqInt = aa2int(SeqChar)
SeqInt = aa2int(SeqChar,'Unknown',unknownAA)
```

Input Arguments

SeqChar	One of the following: • Character vector or string containing single-letter codes specifying an amino acid sequence. For valid letter codes, see the table Mapping Amino Acid Letter Codes to Integers. Unknown characters are mapped to `0`. Integers are arbitrarily assigned to IUB/IUPAC letters. • MATLAB® structure containing a `Sequence` field that contains an amino acid sequence, such as returned by `fastaread`, `getgenpept`, `genpeptread`, `getpdb`, or `pdbread`.
unknownAA	Number representing an unknown amino acid. Default is `0`.

Output Arguments

SeqInt	Amino acid sequence specified by a row vector of integers.

Description

`SeqInt = aa2int(SeqChar)` converts *SeqChar*, a character vector or string containing single-letter codes specifying an amino acid sequence, to *SeqInt*, a row vector of integers specifying the same amino acid sequence. For valid letter codes, see the table Mapping Amino Acid Letter Codes to Integers.

`SeqInt = aa2int(SeqChar,'Unknown',unknownAA)` specifies the number used to represent an unknown amino acid.

Mapping Amino Acid Letter Codes to Integers

Amino Acid	Code	Integer
Alanine	A	1
Arginine	R	2
Asparagine	N	3
Aspartic acid (Aspartate)	D	4
Cysteine	C	5
Glutamine	Q	6
Glutamic acid (Glutamate)	E	7
Glycine	G	8
Histidine	H	9
Isoleucine	I	10
Leucine	L	11
Lysine	K	12
Methionine	M	13
Phenylalanine	F	14
Proline	P	15
Serine	S	16
Threonine	T	17
Tryptophan	W	18
Tyrosine	Y	19
Valine	V	20
Asparagine or Aspartic acid (Aspartate)	B	21
Glutamine or Glutamic acid (Glutamate)	Z	22
Unknown amino acid (any amino acid)	X	23
Translation stop	*	24
Gap of indeterminate length	-	25
Unknown character (any character or symbol not in table)	?	0

Examples

Convert an amino acid sequence to integer representation

Create a random amino acid sequence.

```
seq = randseq(20,'alphabet','amino')
```

```
seq =
'TYNYMRQLVVDVVITNHYSV'
```

Convert the sequence from letter to integer representation.

```
seqInt = aa2int(seq)
```

```
seqInt = 1x20 uint8 row vector

   17   19    3   19   13    2    6   11   20   20    4   20   20   10   17    3    9   19   16
```

Version History
Introduced before R2006a

See Also
`aminolookup` | `int2aa` | `int2nt` | `nt2int`

addBlock

Package: `bioinfo.pipeline`

Add blocks to pipeline

Syntax

```
addBlock(pipeline,inBlocks)
addBlock(pipeline,inBlocks,blockNames)
addedBlocks = addBlock( ___ )
```

Description

`addBlock(pipeline,inBlocks)` adds a block or vector of blocks `inBlocks` to `pipeline`, a `bioinfo.pipeline.Pipeline` object.

`addBlock(pipeline,inBlocks,blockNames)` also specifies the corresponding block name or vector of block names `blockNames`.

`addedBlocks = addBlock( ___ )`, for any syntax, returns the added block or vector of blocks `outBlocks` as a `bioinfo.pipeline.Block` object or vector of such objects.

Examples

Add Blocks to Bioinformatics Pipeline

Import the Pipeline and block objects needed for the example.

```
import bioinfo.pipeline.Pipeline
import bioinfo.pipeline.blocks.*
```

Create a pipeline.

```
qcpipeline = Pipeline;
```

Select an input FASTQ file using a `FileChooser` block.

```
fastqfile = FileChooser(which("SRR005164_1_50.fastq"));
```

Add the `fastqfile` block to the pipeline.

```
addBlock(qcpipeline,fastqfile);
```

By default, if you do not provide the block name, the pipeline gives a unique and valid variable name for the block when the block is added to the pipeline.

```
qcpipeline.BlockNames
```

```
ans = 
"FileChooser_1"
```

Add the sequence filter (`SeqFilter`) block to the pipeline. Optionally, you can provide the block name.

```
sequenceFilter = addBlock(qcpipeline,SeqFilter,"SF");
qcpipeline.BlockNames
```

```
ans = 2×1 string
    "FileChooser_1"
    "SF"
```

Input Arguments

`pipeline` — Bioinformatics pipeline
`bioinfo.pipeline.Pipeline` object

Bioinformatics pipeline, specified as a `bioinfo.pipeline.Pipeline` object.

`inBlocks` — Input blocks
`bioinfo.pipeline.Block` object | vector of objects

Input blocks, specified as a `bioinfo.pipeline.Block` object or vector of such objects. Each block gets assigned a unique name unless you provide `blockNames` as the third input argument.

`blockNames` — Names of input blocks
character vector | string scalar | string vector | cell array of character vectors

Names of input blocks, specified as a character vector, string scalar, string vector, or cell array of character vectors.

If `inBlocks` is a single block, `blockNames` must be a character vector or string scalar. Otherwise, `blockNames` must be a string vector or cell array of character vectors with the same number of elements as `inBlocks`. The block names must be valid variable names and must be unique within a pipeline.

Data Types: `char` | `string` | `cell`

Output Arguments

`addedBlocks` — Added blocks
`bioinfo.pipeline.Block` object | vector of objects

Added blocks, returned as a `bioinfo.pipeline.Block` object or vector of block objects.

Version History

Introduced in R2023a

See Also

`bioinfo.pipeline.Block` | `bioinfo.pipeline.Pipeline` | `connect` | **Biopipeline Designer**

bioinfo.pipeline.Block

Block object for Bioinformatics pipeline

Description

The `bioinfo.pipeline.Block` object is used in a bioinformatics pipeline to perform a unit of work, such as aligning reads to a reference, that is necessary to achieve the final goal of an analysis pipeline.

Bioinformatics Toolbox provides built-in blocks to accomplish some bioinformatics-specific tasks. For instance, you can use the `SeqTrim` block to trim genomic reads and `Bowtie2` block to align reads to a reference genome.

In addition to built-in blocks, you can also convert any existing MATLAB function into a block by using a `UserFunction` block and use it in your pipeline.

You can implement most pipelines and analysis workflows by using a combination of `UserFunction` blocks and built-in blocks. However, if you are a developer or advanced user who needs to customize a block behavior beyond those of the built-in blocks and `UserFunction` blocks, you can create your own block object as a subclass of the `bioinfo.pipeline.Block` class. For details, see “Subclass Pipeline Block” on page 1-12.

Creation

To create one of the built-in blocks, use `bioinfo.pipeline.blocks.`*`BlockName`*, where *BlockName* is the name of the built-in block. For example, to create a `SamSort` block, enter `bioinfo.pipeline.blocks.SamSort`. Similarly, to create a `UserFunction` block, use `bioinfo.pipeline.blocks.UserFunction`.

Tip To see a list of built-in blocks at the MATLAB command line, enter

```
bioinfo.pipeline.blocks.
```

And then hit the **Tab** key.

Properties

`ErrorHandler` — Function to handle errors from the `run` method
empty `function_handle` array (default) | function handle

Function to handle errors from the `run` method of the block, specified as a function handle. It specifies the function to call if the run method encounters an error within a pipeline. In order for the pipeline to continue after a block fails, `ErrorHandler` must return a structure compatible with the output ports of the block. The error handling function is called with the following two input arguments:

- Structure with the following fields:

Field	Description
identifier	Identifier of the error that occurred
message	Text of the error message
index	Linear index indicating which block process failed in the parallel run. By default, the index is always 1 because there is only one run per block. For details on how block inputs can be split across different dimensions for multiple run calls, see “Bioinformatics Pipeline SplitDimension”.

- Input structure passed to the `run` method when it failed.

Data Types: `function_handle`

`Inputs` — Block input ports
structure

This property is read-only.

Block input ports, specified as a structure. The field names are the names of the input ports and the values are `bioinfo.pipeline.Input` objects describing the input port behavior. These input port names are the expected fields of the input struct passed to the `run` method of the block.

You can change or set the values of the input port structure as follows:
`blockObj.Inputs.`*`FieldName`*`.Value = someValue`.

Data Types: `struct`

`Outputs` — Block output ports
structure

This property is read-only.

Block output ports, specified as a structure. The field names are the names of the output ports and the values are `bioinfo.pipeline.Output` objects describing the output port behavior. These output port names are the expected fields of the output struct returned by the `run` method of the block.

Data Types: `struct`

Object Functions

compile	Perform block-specific additional checks and validations
copy	Copy array of handle objects
emptyInputs	Create input structure for use with run method
eval	Evaluate block object
run	Run block object

Examples

Create a Simple Pipeline to Plot Sequence Quality Data

Import the Pipeline and block objects needed for the example.

```
import bioinfo.pipeline.Pipeline
import bioinfo.pipeline.blocks.*
```

Create a pipeline.

```
qcpipeline = Pipeline;
```

Select an input FASTQ file using a `FileChooser` block.

```
fastqfile = FileChooser(which("SRR005164_1_50.fastq"));
```

Create a `SeqFilter` block.

```
sequencefilter = SeqFilter;
```

Define the filtering threshold value. Specifically, filter out sequences with a total of more than 10 low-quality bases, where a base is considered a low-quality base if its quality score is less than 20.

```
sequencefilter.Options.Threshold = [10 20];
```

Add the blocks to the pipeline.

```
addBlock(qcpipeline,[fastqfile,sequencefilter]);
```

Connect the output of the first block to the input of the second block. To do so, you need to first check the input and output port names of the corresponding blocks.

View the `Outputs` (port of the first block) and `Inputs` (port of the second block).

```
fastqfile.Outputs
```

```
ans = struct with fields:
    Files: [1×1 bioinfo.pipeline.Output]
```

```
sequencefilter.Inputs
```

```
ans = struct with fields:
    FASTQFiles: [1×1 bioinfo.pipeline.Input]
```

Connect the `Files` output port of the `fastqfile` block to the `FASTQFiles` port of `sequencefilter` block.

```
connect(qcpipeline,fastqfile,sequencefilter,["Files","FASTQFiles"]);
```

Next, create a `UserFunction` block that calls the `seqqcplot` function to plot the quality data of the filtered sequence data. In this case, `inputFile` is the required argument for the `seqqcplot` function. The required argument name can be anything as long as it is a valid variable name.

```
qcplot = UserFunction("seqqcplot",RequiredArguments="inputFile",OutputArguments="figureHandle");
```

Alternatively, you can also use dot notation to set up your `UserFunction` block.

```
qcplot = UserFunction;
qcplot.RequiredArguments = "inputFile";
qcplot.Function = "seqqcplot";
qcplot.OutputArguments = "figureHandle";
```

Add the block.

```
addBlock(qcpipeline,qcplot);
```

Check the port names of `sequencefilter` block and `qcplot` block.

```
sequencefilter.Outputs
```

```
ans = struct with fields:
    FilteredFASTQFiles: [1×1 bioinfo.pipeline.Output]
         NumFilteredIn: [1×1 bioinfo.pipeline.Output]
        NumFilteredOut: [1×1 bioinfo.pipeline.Output]
```

```
qcplot.Inputs
```

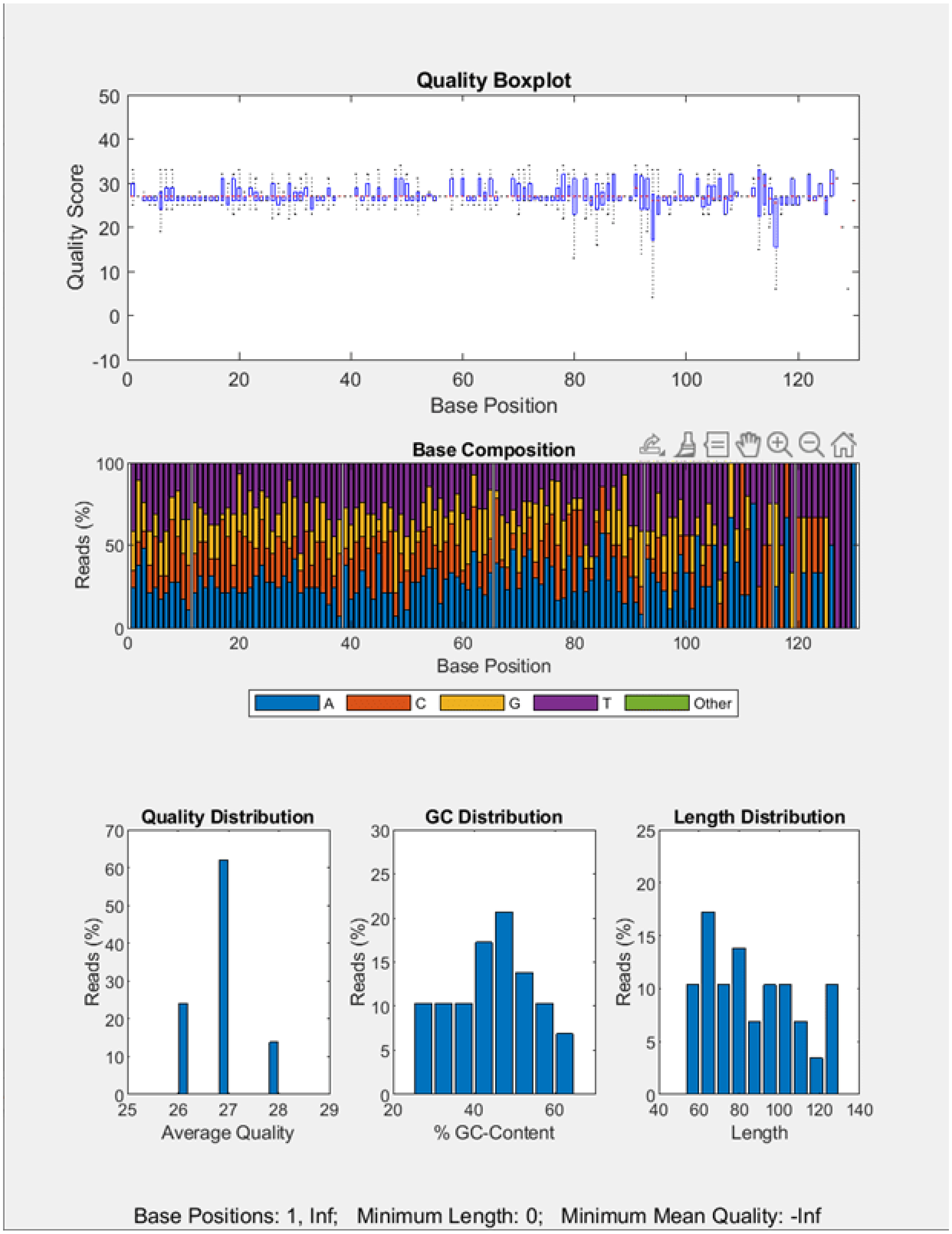
```
ans = struct with fields:
    inputFile: [1×1 bioinfo.pipeline.Input]
```

Connect the `FilteredFASTQFiles` port of the `sequencefilter` block to the `inputFile` port of the `qcplot` block.

```
connect(qcpipeline,sequencefilter,qcplot,["FilteredFASTQFiles","inputFile"]);
```

Run the pipeline to plot the sequence quality data.

```
run(qcpipeline);
```



Base Positions: 1, Inf; Minimum Length: 0; Minimum Mean Quality: -Inf

More About

Subclass Pipeline Block

For most pipelines, using a combination of built-in blocks and `UserFunction` blocks is sufficient and recommended. However, if a block requires more complexity, such as additional configuration options or methods, you can create a custom block by subclassing the `bioinfo.pipeline.Block` object to implement additional capabilities.

Subclasses of the `Block` object must:

- Define their `Inputs` and `Outputs` properties.
- Define the `eval` method.

For instance, the following lines of code defines a custom block object named `Seqlinkage` which is defined as a subclass of `bioinfo.pipeline.Block.` For illustration purposes, the `Seqlinkage` block uses the `seqlinkage` function, which constructs a phylogenetic tree from pairwise distances as the underlying function. The function accepts a matrix or vector of pairwise distances and a distance method to use.

```
classdef Seqlinkage < bioinfo.pipeline.Block
      % Define the block properties
      properties
          % Define the Method property that accepts two distance methods.
          % For details on these methods, enter the following command at
          % the command line: doc seqlinkage
          Method {mustBeMember(Method, ["average", "weighted"])} = "average";
      end

      methods
          % Define inputs and outputs.
          function block = Seqlinkage()
              import bioinfo.pipeline.Input
              import bioinfo.pipeline.Output

              % Define the Inputs and Outputs property of the object.

              % Name the first input port asd Distances and as a required
              % input that takes in a matrix or vector of pairwise distances.
              block.Inputs.Distances = Input('Required',true);

              % Name the second input port as Names and as an optional
              % input that takes in a list of names for nodes.
              block.Inputs.Names = Input('Required',false);

              % Name the output port Phytree.
              block.Outputs.Phytree = Output;
          end

          % Define custom evaluation method for the block.
          function outStruct = eval(obj, inStruct)
              % If the optional input (a list of node names) is passed in
              if isfield(inStruct,'Names')
                  % Call the seqlinkage function with three inputs and save
                  % the returned phytree object as output.
```

```
                    outStruct.Phytree = seqlinkage(inStruct.Distances,obj.Method,inStruct.Names);
                else
                    % Call seqlinkage with two inputs.
                    outStruct.Phytree = seqlinkage(inStruct.Distances,obj.Method);
                end
            end
        end
    end
```

You can save such a class definition to a separate MATLAB® program file. For this example, the above `Seqlinkage` class definition has already been saved and provided as `Seqlinkage.m`. Note that the definition file must be in the current working folder or MATLAB search path before you can use the block object in your pipeline.

Next, you can define a pipeline that uses the `Seqlinkage` block as follows to build a phylogenetic tree from pairwise distances.

```
import bioinfo.pipeline.Pipeline
import bioinfo.pipeline.blocks.*
P = Pipeline;
```

Create three blocks needed for the workflow.

```
fastaReadBlock  = UserFunction("fastaread",RequiredArguments="inputFile",OutputArguments="sequenc
seqpdistBlock   = UserFunction("seqpdist",RequiredArguments="inputSequences",OutputArguments="pai
seqlinkageBlock = Seqlinkage;
```

Set the input port of `fastaReadBlock` to a FASTA file containing amino acid sequences.

```
fastaReadBlock.Inputs.inputFile.Value = which("pf00002.fa");
```

Add the blocks to the pipeline.

```
addBlock(P,[fastaReadBlock,seqpdistBlock,seqlinkageBlock],["fastaread","seqpdist","seqlinkage"])
```

Connect the first two blocks.

```
connect(P,"fastaread","seqpdist",["sequences","inputSequences"]);
```

Connect `seqpdistBlock` to `seqlinkageBlock`. Specifically, connect the output port "`pairwiseDistances`" of `seqpdistBlock` to one of the input ports of the `seqlinkageBlock` "`Distances`", as defined in the `Seqlinkage.m`. Note that `Distances` is a required input port that must have its value set. One of the valid ways is to connect to another port. For other ways to set input ports, see “Satisfy Input Ports” on page 1-1648.

```
connect(P,"seqpdist","seqlinkage",["pairwiseDistances","Distances"]);
```

Connect the output port "sequences" of the `fastaread` block to the "`Names`" optional input port of `seqlinkageBlock` to label the leave nodes of the output phylogenetic tree.

```
connect(P,"fastaread","seqlinkage",["sequences","Names"]);
```

Optionally, you can also pass in a list of node names of as the value of the optional input port "`Names`" by setting the property `seqlinkageBlock.Inputs.Names.Value`.

Before you run the pipeline, ensure that the folder of the class definition file is on the MATLAB search path. The reason is because each pipeline block runs in its own folder, and the external class definition files or function files will not be detected by the pipeline unless they are on the path.

```
% Update the following addpath call to specify the path to your class definition file. For instan
addpath("C:\Examples\SubclassExample\");
```

Run the pipeline.

```
run(P);
```

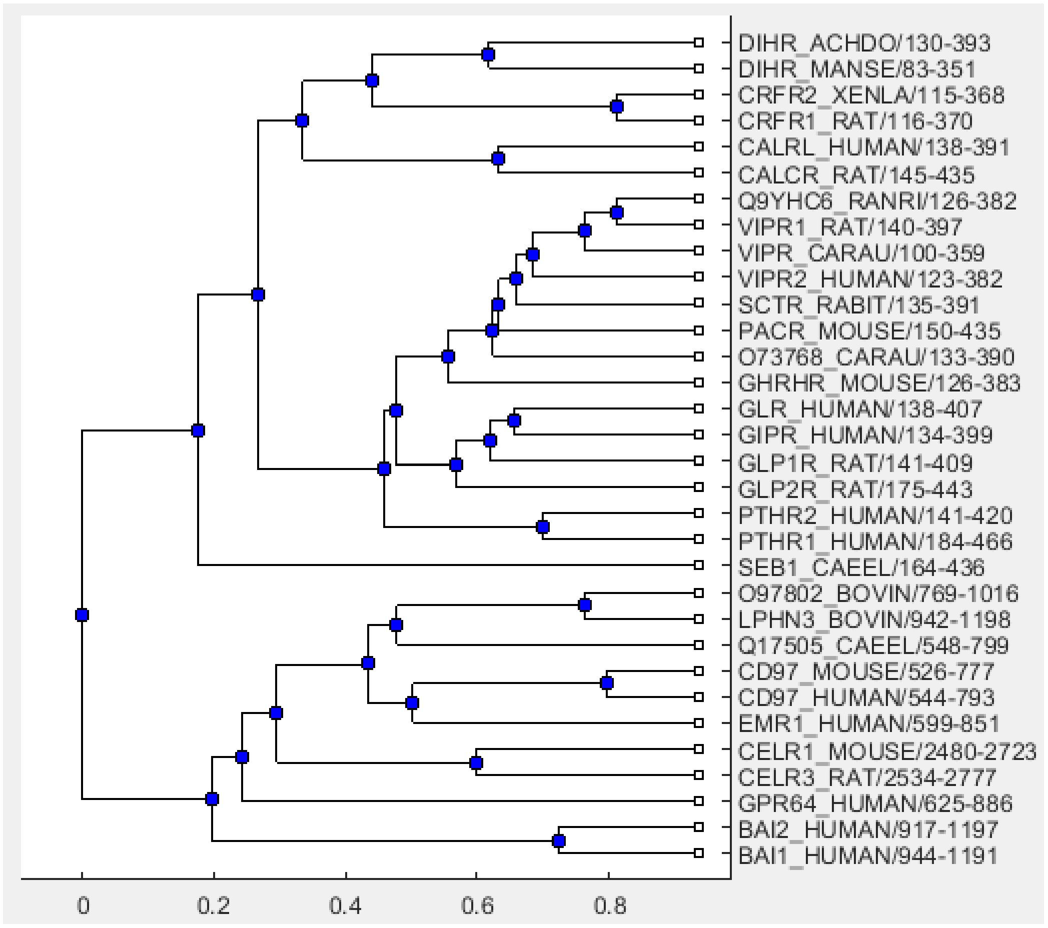
Get the result from `seqlinkageBlock`, which contains a phlogenetic tree object.

```
linkageResult = fetchResults(P,seqlinkageBlock)
```

```
linkageResult = struct with fields:
    Phytree: [1×1 phytree]
```

Use the following command to visualize the phylogenetic tree.

```
view(linkageResult.Phytree)
```



Version History

Introduced in R2023a

See Also

`bioinfo.pipeline.Pipeline` | `bioinfo.pipeline.Input` | `bioinfo.pipeline.Output`

bioinfo.pipeline.blocks.BamSort

Bioinformatics pipeline block to sort BAM files

Description

A `BamSort` block enables you to sort BAM files. The block sorts the alignment records by the reference sequence name first, and then by position within the reference.

Creation

Syntax

```
b = bioinfo.pipeline.blocks.BamSort
b = bioinfo.pipeline.blocks.BamSort(outFileName)
```

Description

`b = bioinfo.pipeline.blocks.BamSort` creates a `BamSort` block.

`b = bioinfo.pipeline.blocks.BamSort(outFileName)` also specifies the output file name.

Input Arguments

`outFileName` — Sorted file name
string | character vector

Sorted file name, specified as a string or character vector.

Data Types: `char` | `string`

Properties

`ErrorHandler` — Function to handle errors from `run` method
function handle

Function to handle errors from the `run` method of the block, specified as a function handle. It specifies the function to call if the run method encounters an error within a pipeline. In order for the pipeline to continue after a block fails, `ErrorHandler` must return a structure compatible with the output ports of the block. The error handling function is called with the following two input arguments:

- Structure with the following fields:

Field	Description
identifier	Identifier of the error that occurred
message	Text of the error message

Field	Description
index	Linear index indicating which block process failed in the parallel run. By default, the index is always 1 because there is only one run per block. For details on how block inputs can be split across different dimensions for multiple run calls, see “Bioinformatics Pipeline SplitDimension”.

- Input structure passed to the `run` method when it failed.

Data Types: `function_handle`

`Inputs` — Input ports
structure

This property is read-only.

Input ports of the block, specified as a structure. The field names of the structure are the names of the block input ports and the field values are `bioinfo.pipeline.Input` objects. These objects describe the input port behaviors. The input port names are the expected field names of the input structure that you pass in for the block `run` method.

The `BamSort` block `Inputs` structure has the field named `BAMFile`. It is a required input port and must be satisfied on page 1-1648 to run the block.

Data Types: `struct`

`Outputs` — Output ports
structure

This property is read-only.

Output ports of the block, specified as a structure. The field names of the structure are the names of the block output ports and the field values are `bioinfo.pipeline.Output` objects. These objects describe the output port behaviors. The output structure returned by the block `run` method has the field names that are the same as the output port names.

The `BamSort` block `Outputs` structure has the field named `SortedBAMFile`.

Data Types: `struct`

`OutFilename` — Sorted BAM file name
empty string array (default) | string | character vector

Sorted file name, specified as a string or character vector. The name must end with the extension `.bam`.

If `OutFilename` is empty (default), the output file has the same base name as the input BAM file with the extension `.sorted.bam`.

If you specify the file name, ensure the name has the `.bam` extension.

Data Types: `char` | `string`

Object Functions

compile	Perform block-specific additional checks and validations
copy	Copy array of handle objects
emptyInputs	Create input structure for use with run method
eval	Evaluate block object
run	Run block object

Examples

Sort BAM file Using Bioinformatics Pipeline

Import the Pipeline and blocks objects needed for the example.

```
import bioinfo.pipeline.Pipeline
import bioinfo.pipeline.blocks.*
```

Create a pipeline object.

```
P = Pipeline;
```

Create a `BamSort` block and `FileChooser` block that takes in a sample BAM file.

```
bs = BamSort;
fc = FileChooser(which("ex1.bam"));
```

Add the blocks.

```
addBlock(P,[bs,fc]);
```

Connect the `Files` output port of the `FileChooser` block to the `BAMFile` input port of `BamSort` block.

```
connect(P,fc,bs,["Files","BAMFile"]);
```

Run the pipeline.

```
run(P);
```

Get the block result, which is a sorted BAM file.

```
outBAM = fetchResults(P,bs)
```

```
outBAM = struct with fields:
    SortedBAMFile: [1×1 bioinfo.pipeline.datatypes.File]
```

Use `unwrap` to see the stored location of the sorted BAM file.

```
unwrap(outBAM.SortedBAMFile)
```

Version History

Introduced in R2023a

See Also

`bioinfo.pipeline.Block` | `bioinfo.pipeline.blocks.SamSort` | `bamsort`

bioinfo.pipeline.blocks.Bowtie2

Bioinformatics pipeline block to align sequencing reads to reference sequences

Description

A `Bowtie2` block enables you to map sequencing reads to reference sequences.

The block requires the Bowtie 2 Support Package for Bioinformatics Toolbox. If this support package is not installed, then a download link is provided. For details, see “Bioinformatics Toolbox Software Support Packages”.

Creation

Syntax

```
b = bioinfo.pipeline.blocks.Bowtie2
b = bioinfo.pipeline.blocks.Bowtie2(options)
b = bioinfo.pipeline.blocks.Bowtie2(OutFilename=fileName)
b = bioinfo.pipeline.blocks.Bowtie2(Name=Value)
```

Description

`b = bioinfo.pipeline.blocks.Bowtie2` creates a `Bowtie2` block.

`b = bioinfo.pipeline.blocks.Bowtie2(options)` also specifies additional alignment `options`.

`b = bioinfo.pipeline.blocks.Bowtie2(OutFilename=fileName)` also specifies the output file name.

`b = bioinfo.pipeline.blocks.Bowtie2(Name=Value)` specifies additional options as the property names and values of a `Bowtie2AlignOptions` object. This object is set as the value of the `Options` property of the block. For example, `bt2Block = bioinfo.pipeline.blocks.Bowtie2(Trim5=10)` sets the `Trim5` property of the object to trim 10 residues from the 5' end.

Input Arguments

`fileName` — Output file name
string | character vector

Output file name, specified as a string or character vector. The file extension must end with `.sam`. The block saves the mapping results to this file.

Data Types: `char` | `string`

`options` — Bowtie2 options
`Bowtie2AlignOptions` | string | character vector

Bowtie2 options, specified as a `Bowtie2AlignOptions` object, string, or character vector.

If you are specifying a string or character vector, it must be in the native `bowtie2` option syntax (prefixed by one or two dashes) [1].

Data Types: `char` | `string`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: `bt2Block = bioinfo.pipeline.blocks.Bowtie2(Trim3=6)` specifies to trim 6 residues from the 3' end.

Note The following list of arguments is a partial list. For the complete list, refer to the properties on page 1-439 of `Bowtie2AlignOptions` object.

`AllowDovetail` — Flag to allow dovetail configurations
`false` or 0 (default) | `true` or 1

Flag to allow dovetail configurations, specified as 1 (`true`) or 0 (`false`). This property specifies whether the alignment of one mate can extend past the beginning of the alignment of the other mate and be considered concordant.

This property applies to paired-end reads only.

Data Types: `double` | `logical`

`AmbiguousPenalty` — Penalty for positions with ambiguous characters
`1` (default) | nonnegative integer

Penalty for positions with ambiguous characters on the read sequence, reference sequence, or both, specified as a nonnegative integer.

Data Types: `double`

Properties

`ErrorHandler` — Function to handle errors from `run` method
function handle

Function to handle errors from the `run` method of the block, specified as a function handle. It specifies the function to call if the run method encounters an error within a pipeline. In order for the pipeline to continue after a block fails, `ErrorHandler` must return a structure compatible with the output ports of the block. The error handling function is called with the following two input arguments:

- Structure with the following fields:

Field	Description
identifier	Identifier of the error that occurred

Field	Description
message	Text of the error message
index	Linear index indicating which block process failed in the parallel run. By default, the index is always 1 because there is only one run per block. For details on how block inputs can be split across different dimensions for multiple run calls, see “Bioinformatics Pipeline SplitDimension”.

- Input structure passed to the `run` method when it failed.

Data Types: `function_handle`

`Inputs` — Input ports
structure

This property is read-only.

Input ports of the block, specified as a structure. The field names of the structure are the names of the block input ports and the field values are `bioinfo.pipeline.Input` objects. These objects describe the input port behaviors. The input port names are the expected field names of the input structure that you pass in for the block `run` method.

The `Bowtie2` block `Inputs` structure has the following fields:

- `IndexBaseName` — Base name of the reference index files. The index files are in the `BT2` or `BT21` format. For example, if you have `Dmel_chr4.1.bt2` and `Dmel_chr4.2.bt2` as your index files, specify `IndexBaseName` as `"Dmel_chr4"`. This input is a required input that must be satisfied on page 1-1648.
- `Reads1Files` — Names of FASTQ files for the first mate reads or single-end reads. For paired-end data, sequences in `Reads1Files` must correspond file-for-file and read-for-read to sequences in `Reads2Files`. This input is a required input that must be satisfied on page 1-1648.
- `Reads2Files` — Names of FASTQ files for the second mate reads for paired-end data. This input is an optional input.

The default value for each of these inputs is a `bioinfo.pipeline.datatypes.Unset` object, which means that the input value is not set yet.

Data Types: `struct`

`Outputs` — Output ports
structure

This property is read-only.

Output ports of the block, specified as a structure. The field names of the structure are the names of the block output ports and the field values are `bioinfo.pipeline.Output` objects. These objects describe the output port behaviors. The output structure returned by the block `run` method has the field names that are the same as the output port names.

The `Bowtie2` block `Outputs` structure has the field named `SAMFile`.

Data Types: `struct`

`Options` — Bowtie2 options
`Bowtie2AlignOptions` object (default)

Bowtie2 options, specified as a `Bowtie2AlignOptions` object. The default value is a default `Bowtie2AlignOptions` object.

`OutFilename` — Output file name
`"Aligned.sam"` (default) | string

Output file name, specified as a string. By default, the output file is named as `Aligned.sam`, which contains the mapping results.

Data Types: `string`

Object Functions

compile	Perform block-specific additional checks and validations
copy	Copy array of handle objects
emptyInputs	Create input structure for use with run method
eval	Evaluate block object
run	Run block object

Version History

Introduced in R2023a

References

[1] Langmead, Ben, and Steven L Salzberg. “Fast Gapped-Read Alignment with Bowtie 2.” Nature Methods 9, no. 4 (April 2012): 357–59. https://doi.org/10.1038/nmeth.1923.

See Also

`bioinfo.pipeline.Block` | `bioinfo.pipeline.blocks.Bowtie2Build`

bioinfo.pipeline.blocks.Bowtie2Build

Bioinformatics pipeline block to create Bowtie2 index from reference sequence

Description

A `Bowtie2Build` block enables you to create index files from a reference sequence.

The block requires the Bowtie 2 Support Package for Bioinformatics Toolbox. If this support package is not installed, then a download link is provided. For details, see “Bioinformatics Toolbox Software Support Packages”.

Creation

Syntax

```
b = bioinfo.pipeline.blocks.Bowtie2Build
b = bioinfo.pipeline.blocks.Bowtie2Build(options)
b = bioinfo.pipeline.blocks.Bowtie2Build(Name=Value)
```

Description

`b = bioinfo.pipeline.blocks.Bowtie2Build` creates a `Bowtie2Build` block.

`b = bioinfo.pipeline.blocks.Bowtie2Build(options)` also specifies additional `options`.

`b = bioinfo.pipeline.blocks.Bowtie2Build(Name=Value)` specifies additional options as the property names and values of a `Bowtie2BuildOptions` object. This object is set as the value of the `Options` property of the block. For example, `bt2buildBlock = bioinfo.pipeline.blocks.Bowtie2Build(ForceLargeIndex=true)` sets the `ForceLargeIndex` property of the object to force the creation of a large index even if the reference is less than 4 billion nucleotides long.

Input Arguments

`options` — Bowtie2Build options
`Bowtie2BuildOptions` | string | character vector

Bowtie2Build options, specified as a `Bowtie2BuildOptions` object, string, or character vector.

If you are specifying a string or character vector, it must be in the native `bowtie2` option syntax (prefixed by one or two dashes) [1].

Data Types: `char` | `string`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Note The following list of arguments is a partial list. For the complete list, refer to the properties on page 1-452 of `Bowtie2BuildOptions` object.

`BuildOnlyReference` — Boolean indicator to build only bitpacked reference
`false` (default) | `true`

Boolean indicator to build only the `3.bt2` and `4.bt2` files that correspond to the bitpacked version of reference sequences, specified as `true` or `false`.

Data Types: `logical`

`BuildNoReference` — Boolean indicator to omit building bitpacked reference
`false` (default) | `true`

Boolean indicator to omit building the `3.bt2` and `4.bt2` files that correspond to the bitpacked version of reference sequences, specified as `true` or `false`.

Data Types: `logical`

Properties

`ErrorHandler` — Function to handle errors from `run` method
function handle

Function to handle errors from the `run` method of the block, specified as a function handle. It specifies the function to call if the run method encounters an error within a pipeline. In order for the pipeline to continue after a block fails, `ErrorHandler` must return a structure compatible with the output ports of the block. The error handling function is called with the following two input arguments:

- Structure with the following fields:

Field	Description
identifier	Identifier of the error that occurred
message	Text of the error message
index	Linear index indicating which block process failed in the parallel run. By default, the index is always 1 because there is only one run per block. For details on how block inputs can be split across different dimensions for multiple run calls, see “Bioinformatics Pipeline SplitDimension”.

- Input structure passed to the `run` method when it failed.

Data Types: `function_handle`

`Inputs` — Input ports
structure

This property is read-only.

Input ports of the block, specified as a structure. The field names of the structure are the names of the block input ports and the field values are `bioinfo.pipeline.Input` objects. These objects describe the input port behaviors. The input port names are the expected field names of the input structure that you pass in for the block `run` method.

The `Bowtie2Build` block `Inputs` structure has the following fields:

- `ReferenceFASTAFiles` — Names of files with reference sequence information. This input is a required input that must be satisfied on page 1-1648.
- `IndexBaseName` — Base name of the reference index files. The index files are in the `BT2` or `BT21` format. For example, if you specify `"Dmel_chr4"` as `IndexBaseName`, the generated index files would be `Dmel_chr4.1.bt2`, `Dmel_chr4.2.bt2`, and so on. This input is optional, and by default, the block uses `"Bowtie2Index"` as the prefix.

The default value for each of these inputs is a `bioinfo.pipeline.datatypes.Unset` object, which means that the input value is not set yet.

Data Types: `struct`

`Outputs` — Output ports
structure

This property is read-only.

Output ports of the block, specified as a structure. The field names of the structure are the names of the block output ports and the field values are `bioinfo.pipeline.Output` objects. These objects describe the output port behaviors. The output structure returned by the block `run` method has the field names that are the same as the output port names.

The `Bowtie2Build` block `Outputs` structure has the field named `IndexBaseName`.

Data Types: `struct`

`Options` — Bowtie2Build options
`Bowtie2BuildOptions` object (default)

Bowtie2Build options, specified as a `Bowtie2BuildOptions` object. The default value is a default `Bowtie2BuildOptions` object.

Object Functions

compile	Perform block-specific additional checks and validations
copy	Copy array of handle objects
emptyInputs	Create input structure for use with run method
eval	Evaluate block object
run	Run block object

Version History

Introduced in R2023a

References

[1] Langmead, B., and S. Salzberg. "Fast gapped-read alignment with Bowtie 2." *Nature Methods*. 9, 2012, 357–359.

See Also

`bioinfo.pipeline.Block` | `bioinfo.pipeline.blocks.Bowtie2` | `bowtie2build`

bioinfo.pipeline.blocks.BwaIndex

Bioinformatics pipeline block to create BWA indices from reference sequences

Description

A `BwaIndex` block enables you to create BWA index files from reference sequences.

The block requires the BWA Support Package for Bioinformatics Toolbox. If the support package is not installed, then a download link is provided. For details, see “Bioinformatics Toolbox Software Support Packages”.

Creation

Syntax

```
b = bioinfo.pipeline.blocks.BwaIndex
b = bioinfo.pipeline.blocks.BwaIndex(options)
b = bioinfo.pipeline.blocks.BwaIndex(Name=Value)
```

Description

`b = bioinfo.pipeline.blocks.BwaIndex` creates a `BwaIndex` block.

`b = bioinfo.pipeline.blocks.BwaIndex(options)` also specifies additional `options`.

`b = bioinfo.pipeline.blocks.BwaIndex(Name=Value)` specifies additional options as the property names and values of a `BWAIndexOptions` object. This object is set as the value of the `Options` property of the block.

Input Arguments

`options` — BwaIndex options
`BwaIndexOptions` | string | character vector

BwaIndex options, specified as a `BWAIndexOptions` object, string, or character vector.

If you are specifying a string or character vector, it must be in the `bwa` native syntax (prefixed by a dash) [1][2] .

Data Types: `char` | `string`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Note The following list of arguments is a partial list. For the complete list, refer to the properties on page 1-468 of `BwaIndexOptions` object.

`Algorithm` — Algorithm to construct BWT index
empty string array (default) | `'is'` | `'bwtsw'`

Algorithm to construct the BWT (Burrows-Wheeler transform) index, specified as a character vector or string. Options are:

- `'is'` — Linear-time algorithm. The memory requirement for using this option is 5.37 times the size of the database. You cannot use this option if your database is larger than 2 GB.
- `'bwtsw'` — BWT-SW algorithm.

The default algorithm is chosen automatically based on the size of the reference genome.

Data Types: `char` | `string`

`BlockSize` — Number of bases processed per batch
`1e7` (default) | positive scalar

Number of bases processed per batch in the `bwtsw` algorithm, specified as a positive scalar.

Data Types: `double`

Properties

`ErrorHandler` — Function to handle errors from `run` method
function handle

Function to handle errors from the `run` method of the block, specified as a function handle. It specifies the function to call if the run method encounters an error within a pipeline. In order for the pipeline to continue after a block fails, `ErrorHandler` must return a structure compatible with the output ports of the block. The error handling function is called with the following two input arguments:

- Structure with the following fields:

Field	Description
identifier	Identifier of the error that occurred
message	Text of the error message
index	Linear index indicating which block process failed in the parallel run. By default, the index is always 1 because there is only one run per block. For details on how block inputs can be split across different dimensions for multiple run calls, see “Bioinformatics Pipeline SplitDimension”.

- Input structure passed to the `run` method when it failed.

Data Types: `function_handle`

`Inputs` — Input ports
structure

This property is read-only.

Input ports of the block, specified as a structure. The field names of the structure are the names of the block input ports and the field values are `bioinfo.pipeline.Input` objects. These objects describe the input port behaviors. The input port names are the expected field names of the input structure that you pass in for the block `run` method.

The `BwaIndex` block `Inputs` structure has the following field:

- `ReferenceFASTAFile` — Name of the FASTA-formatted reference sequence file. This input is a required input that must be satisfied on page 1-1648. The default value is a `bioinfo.pipeline.datatypes.Unset` object, which means that the input value is not set yet.

Data Types: `struct`

`Outputs` — Output ports
structure

This property is read-only.

Output ports of the block, specified as a structure. The field names of the structure are the names of the block output ports and the field values are `bioinfo.pipeline.Output` objects. These objects describe the output port behaviors. The output structure returned by the block `run` method has the field names that are the same as the output port names.

The `BwaIndex` block `Outputs` structure has the field named `IndexBaseName` and writes the index files to the same location where the reference file (`ReferenceFASTAFile`) is.

Data Types: `struct`

`Options` — BwaIndex options
`BwaIndexOptions` object (default)

`BwaIndex` options, specified as a `BWAIndexOptions` object. The default value is a default `BwaIndexOptions` object.

Object Functions

compile	Perform block-specific additional checks and validations
copy	Copy array of handle objects
emptyInputs	Create input structure for use with run method
eval	Evaluate block object
run	Run block object

Examples

Create BWA index files

Create a set of BWA index files for chromosome 4 of the Drosophila genome.

```
import bioinfo.pipeline.blocks.*
import bioinfo.pipeline.Pipeline

FC = FileChooser(which("Dmel_chr4.fa"));
BI = BwaIndex;
```

```
P = Pipeline;
addBlock(P,[FC BI]);
connect(P,FC,BI,["Files","ReferenceFASTAFile"]);

% Set the custom location to save the index files.
mkdir C:\BWAIndexFiles\
BI.Options.Prefix = "C:\BWAIndexFiles\BWAIndex_Dmel_chr4";

run(P);
R = results(P,BI)
```

```
R = 

  struct with fields:

    IndexBaseName: "C:\BWAIndexFiles\BWAIndex_Dmel_chr4"
```

The generated index files are now saved under the folder that you have specified.

Version History

Introduced in R2023a

References

[1] Li, Heng, and Richard Durbin. “Fast and Accurate Short Read Alignment with Burrows-Wheeler Transform.” *Bioinformatics* 25, no. 14 (July 15, 2009): 1754–60. https://doi.org/10.1093/bioinformatics/btp324.

[2] Li, Heng, and Richard Durbin. “Fast and Accurate Long-Read Alignment with Burrows–Wheeler Transform.” *Bioinformatics* 26, no. 5 (March 1, 2010): 589–95. https://doi.org/10.1093/bioinformatics/btp698.

See Also

`bioinfo.pipeline.Block` | `bioinfo.pipeline.blocks.BwaMEM` | `bwaindex`

bioinfo.pipeline.blocks.BwaMEM

Bioinformatics pipeline block to map sequence reads to reference genome

Description

A `BwaMEM` block enables you to map sequencing reads to a reference genome.

The block requires the BWA Support Package for Bioinformatics Toolbox. If the support package is not installed, then a download link is provided. For details, see “Bioinformatics Toolbox Software Support Packages”.

Creation

Syntax

```
b = bioinfo.pipeline.blocks.BwaMEM
b = bioinfo.pipeline.blocks.BwaMEM(options)
b = bioinfo.pipeline.blocks.BwaMEM(OutFilename=fileName)
b = bioinfo.pipeline.blocks.BwaMEM(Name=Value)
```

Description

`b = bioinfo.pipeline.blocks.BwaMEM` creates a `BwaMEM` block.

`b = bioinfo.pipeline.blocks.BwaMEM(options)` also specifies additional `options`.

`b = bioinfo.pipeline.blocks.BwaMEM(OutFilename=fileName)` also specifies the output file name.

`b = bioinfo.pipeline.blocks.BwaMEM(Name=Value)` specifies additional options as the property names and values of a `BWAMEMOptions` object. This object is set as the value of the `Options` property of the block.

Input Arguments

`fileName` — Output file name
string | character vector

Output file name, specified as a string or character vector. The block saves the mapping results to this file.

Data Types: `char` | `string`

`options` — BwaMEM options
`BwaMEMOptions` | string | character vector

`BwaMEM` options, specified as a `BWAMEMOptions` object, string, or character vector.

If you are specifying a string or character vector, it must be in the `bwa` native syntax (prefixed by a dash) [1][2] .

Data Types: `char` | `string`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Note The following list of arguments is a partial list. For the complete list, refer to the properties on page 1-482 of `BwaMEMOptions` object.

`AlternativeHitsThreshold` — Threshold for determining which hits receive XA tag in output SAM file
`[5 200]` (default) | nonnegative integer | two-element numeric vector

Threshold for determining which hits receive an XA tag in the output SAM file, specified as a nonnegative integer *n* or two-element numeric vector `[n m]`, where *n* and *m* must be nonnegative integers.

If a read has less than *n* hits with a score greater than 80% of the best score for that read, all hits receive an XA tag in the output SAM file.

When you also specify *m*, the software returns up to *m* hits if the hit list contains a hit to an ALT contig.

Data Types: `double`

`AppendReadCommentsToSAM` — Flag to append FASTA or FASTQ comments to output SAM file
`false` (default) | `true`

Flag to append FASTA or FASTQ comments to the output SAM file, specified as `true` or `false`. The comments appear as text after a space in the file header.

Data Types: `logical`

Properties

`ErrorHandler` — Function to handle errors from `run` method
function handle

Function to handle errors from the `run` method of the block, specified as a function handle. It specifies the function to call if the run method encounters an error within a pipeline. In order for the pipeline to continue after a block fails, `ErrorHandler` must return a structure compatible with the output ports of the block. The error handling function is called with the following two input arguments:

- Structure with the following fields:

Field	Description
identifier	Identifier of the error that occurred

Field	Description
message	Text of the error message
index	Linear index indicating which block process failed in the parallel run. By default, the index is always 1 because there is only one run per block. For details on how block inputs can be split across different dimensions for multiple run calls, see “Bioinformatics Pipeline SplitDimension”.

- Input structure passed to the `run` method when it failed.

Data Types: `function_handle`

`Inputs` — Input ports
structure

This property is read-only.

Input ports of the block, specified as a structure. The field names of the structure are the names of the block input ports and the field values are `bioinfo.pipeline.Input` objects. These objects describe the input port behaviors. The input port names are the expected field names of the input structure that you pass in for the block `run` method.

The `BwaMEM` block `Inputs` structure has the following fields:

- `IndexBaseName` — Base name of the reference index files. The index files are in the AMB, ANN, BWT, PAC, and SA file formats. For example, the base name of an index file `Dmel_chr4.bwt` is `"Dmel_chr4"`. This input is a required input that must be satisfied on page 1-1648.
- `Reads1File` — Name of FASTQ file for the first mate reads or single-end reads. For paired-end data, sequences in `Reads1File` must correspond read-for-read to sequences in `Reads2File`. This input is a required input that must be satisfied on page 1-1648.
- `Reads2File` — Name of FASTQ file for the second mate reads for paired-end data. This input is an optional input.

The default value for each of these inputs is a `bioinfo.pipeline.datatypes.Unset` object, which means that the input value is not set yet.

Data Types: `struct`

`Outputs` — Output ports
structure

This property is read-only.

Output ports of the block, specified as a structure. The field names of the structure are the names of the block output ports and the field values are `bioinfo.pipeline.Output` objects. These objects describe the output port behaviors. The output structure returned by the block `run` method has the field names that are the same as the output port names.

The `BwaMEM` block `Outputs` structure has the field named `SAMFile`.

Tip To see the actual location of the output file, first get the results of the block. Then use the `unwrap` method as shown in this example on page 1-35.

Data Types: `struct`

`Options` — BwaMEM options
`BwaMEMOptions` object (default)

`BwaMEM` options, specified as a `BWAMEMOptions` object. The default value is a default `BwaMEMOptions` object.

`OutFilename` — Output file name
`"Aligned.sam"` (default) | string

Output file name, specified as a string. By default, the output file is named as `Aligned.sam`, which contains the mapping results.

Data Types: `string`

Object Functions

compile	Perform block-specific additional checks and validations
copy	Copy array of handle objects
emptyInputs	Create input structure for use with run method
eval	Evaluate block object
run	Run block object

Examples

Map Reads to Reference Using BwaMEM

Map reads to the Drosophila chromosome 4 sequence using the `BwaMEM` block.

```
import bioinfo.pipeline.blocks.*
import bioinfo.pipeline.Pipeline

FC1 = FileChooser(which("Dmel_chr4.fa"));
FC2 = FileChooser(which("SRR6008575_10k_1.fq"));
BI = BwaIndex;
BM = BwaMEM;

P = Pipeline;
addBlock(P,[FC1,FC2,BI,BM]);
connect(P,FC1,BI,["Files","ReferenceFASTAFile"]);
connect(P,BI,BM,["IndexBaseName", "IndexBaseName"]);
connect(P,FC2,BM,["Files", "Reads1File"]);

run(P);
results(P,BM)
```

```
ans = 

  struct with fields:

    SAMFile: [1×1 bioinfo.pipeline.datatypes.File]
```

Call `unwrap` on `SAMFile` to see the location of the output file.

```
unwrap(R.FilteredFASTQFiles)
```

```
ans = 

    "C:\PipelineResults\BwaMEM_1\1\Aligned.sam"
```

Version History

Introduced in R2023a

References

[1] Li, Heng, and Richard Durbin. “Fast and Accurate Short Read Alignment with Burrows-Wheeler Transform.” *Bioinformatics* 25, no. 14 (July 15, 2009): 1754–60. https://doi.org/10.1093/bioinformatics/btp324.

[2] Li, Heng, and Richard Durbin. “Fast and Accurate Long-Read Alignment with Burrows–Wheeler Transform.” *Bioinformatics* 26, no. 5 (March 1, 2010): 589–95. https://doi.org/10.1093/bioinformatics/btp698.

See Also

`bioinfo.pipeline.Block` | `bioinfo.pipeline.blocks.BwaIndex` | `bwamem`

bioinfo.pipeline.blocks.CuffCompare

Bioinformatics pipeline block to compare assembled transcripts

Description

A `CuffCompare` block enables you to compare assembled transcripts across multiple experiments.

The block requires the Cufflinks Support Package for the Bioinformatics Toolbox. If the support package is not installed, then a download link is provided. For details, see “Bioinformatics Toolbox Software Support Packages”.

Creation

Syntax

```
b = bioinfo.pipeline.blocks.CuffCompare
b = bioinfo.pipeline.blocks.CuffCompare(options)
b = bioinfo.pipeline.blocks.CuffCompare(Name=Value)
```

Description

`b = bioinfo.pipeline.blocks.CuffCompare` creates a `CuffCompare` block.

`b = bioinfo.pipeline.blocks.CuffCompare(options)` also specifies additional `options`.

`b = bioinfo.pipeline.blocks.CuffCompare(Name=Value)` specifies additional options as the property names and values of a `CuffCompareOptions` object. This object is set as the value of the `Options` property of the block.

Input Arguments

`options` — CuffCompare options
`CuffCompareOptions` | string | character vector

CuffCompare options, specified as a `CuffCompareOptions` object, string, or character vector.

If you are specifying a string or character vector, it must be in the `cuffcompare` native syntax (prefixed by one or two dashes) [1].

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Note The following list of arguments is a partial list. For the complete list, refer to the properties on page 1-615 of `CuffCompareOptions` object.

`ConsensusPrefix` — Prefix for consensus transcript names
`"TCONS"` (default) | string | character vector

Prefix for consensus transcript names in the output `combined.gtf` file, specified as a string or character vector. This option must be a string or character vector with a non-zero length.

Example: `"consensusTs"`

Data Types: `char` | `string`

`DiscardIntronRedundant` — Flag to ignore intron-redundant transfrags
`false` (default) | `true`

Flag to ignore intron-redundant transfrags if they have the same 5' end but different 3' ends, specified as `true` or `false`.

Example: `true`

Data Types: `logical`

Properties

`ErrorHandler` — Function to handle errors from `run` method
function handle

Function to handle errors from the `run` method of the block, specified as a function handle. It specifies the function to call if the run method encounters an error within a pipeline. In order for the pipeline to continue after a block fails, `ErrorHandler` must return a structure compatible with the output ports of the block. The error handling function is called with the following two input arguments:

- Structure with the following fields:

Field	Description
identifier	Identifier of the error that occurred
message	Text of the error message
index	Linear index indicating which block process failed in the parallel run. By default, the index is always 1 because there is only one run per block. For details on how block inputs can be split across different dimensions for multiple run calls, see “Bioinformatics Pipeline SplitDimension”.

- Input structure passed to the `run` method when it failed.

Data Types: `function_handle`

`Inputs` — Input ports
structure

This property is read-only.

Input ports of the block, specified as a structure. The field names of the structure are the names of the block input ports and the field values are `bioinfo.pipeline.Input` objects. These objects describe the input port behaviors. The input port names are the expected field names of the input structure that you pass in for the block `run` method.

The `CuffCompare` block `Inputs` structure has the following field:

- `GenomicAnnotationFiles` — Names of GTF files. Each GTF file corresponds to a sample produced by the `Cufflinks` block. This input is a required input that must be satisfied on page 1-1648. The default value is a `bioinfo.pipeline.datatypes.Unset` object, which means that the input value is not set yet.

Data Types: `struct`

`Outputs` — Output ports
structure

This property is read-only.

Output ports of the block, specified as a structure. The field names of the structure are the names of the block output ports and the field values are `bioinfo.pipeline.Output` objects. These objects describe the output port behaviors. The output structure returned by the block `run` method has the field names that are the same as the output port names.

The `CuffCompare` block `Outputs` structure has the following fields:

- `StatsFile` — Name of the text file containing statistics related to the accuracy of the transcripts in each sample.
- `CombinedGTFFile` — Name of the file containing the union of all transfrags in each sample.
- `LociFile` — Name of file with all processed loci across all transcripts.
- `TrackingFile` — Name of the file containing transcripts with identical coordinates, introns, and strands.

Tip To see the actual location of these files, first get the results of the block. Then use the `unwrap` method as shown in this example on page 1-39.

Data Types: `struct`

`Options` — `CuffCompare` options
`CuffCompareOptions` object (default)

`CuffCompare` options, specified as a `CuffCompareOptions` object. The default value is a default `CuffCompareOptions` object.

Object Functions

compile	Perform block-specific additional checks and validations
copy	Copy array of handle objects
emptyInputs	Create input structure for use with run method
eval	Evaluate block object
run	Run block object

Examples

Use CuffCompare Block to Compare Assembled Transcript

Compare assembled transcripts using the provided GTF files which were pregenerated by `cufflinks`.

```
import bioinfo.pipeline.blocks.*
import bioinfo.pipeline.Pipeline

gtfFiles = {which("Myco_1_1.transcripts.gtf"),which("Myco_2_1.transcripts.gtf")};
FC = FileChooser(gtfFiles);
CC = CuffCompare;

P = Pipeline;
addBlock(P,[FC,CC]);
connect(P,FC,CC,["Files","GenomicAnnotationFiles"]);

run(P);
R = results(P,CC)
```

```
R = 

  struct with fields:

          StatsFile: [1×1 bioinfo.pipeline.datatypes.File]
    CombinedGTFFile: [1×1 bioinfo.pipeline.datatypes.File]
          LociFile: [1×1 bioinfo.pipeline.datatypes.File]
      TrackingFile: [1×1 bioinfo.pipeline.datatypes.File]
```

Call `unwrap` on each field of the result structure `R` to see the location of each output file. For example, to see the location of `StatsFile`, enter the following.

```
unwrap(R.StatsFile)
```

```
ans = 

    "C:\PipelineResults\CuffCompare_1\1\cuffcmp.stats"
```

Version History

Introduced in R2023a

References

[1] Trapnell, Cole, Brian A Williams, Geo Pertea, Ali Mortazavi, Gordon Kwan, Marijke J van Baren, Steven L Salzberg, Barbara J Wold, and Lior Pachter. "Transcript Assembly and Quantification by RNA-Seq Reveals Unannotated Transcripts and Isoform Switching during Cell Differentiation." *Nature Biotechnology* 28, no. 5 (May 2010): 511–15.

See Also

`bioinfo.pipeline.Block` | `cuffcompare` | `bioinfo.pipeline.blocks.Cufflinks`

bioinfo.pipeline.blocks.CuffDiff

Bioinformatics pipeline block to identify significant changes in transcript expression

Description

A `CuffDiff` block enables you to identify significant changes in transcript expression between the samples.

The block requires the Cufflinks Support Package for the Bioinformatics Toolbox. If the support package is not installed, then a download link is provided. For details, see “Bioinformatics Toolbox Software Support Packages”.

Creation

Syntax

```
b = bioinfo.pipeline.blocks.CuffDiff
b = bioinfo.pipeline.blocks.CuffDiff(options)
b = bioinfo.pipeline.blocks.CuffDiff(Name=Value)
```

Description

`b = bioinfo.pipeline.blocks.CuffDiff` creates a `CuffDiff` block.

`b = bioinfo.pipeline.blocks.CuffDiff(options)` also specifies additional `options`.

`b = bioinfo.pipeline.blocks.CuffDiff(Name=Value)` specifies additional options as the property names and values of a `CuffDiffOptions` object. This object is set as the value of the `Options` property of the block.

Input Arguments

`options` — CuffDiff options
`CuffDiffOptions` | string | character vector

CuffDiff options, specified as a `CuffDiffOptions` object, string, or character vector.

If you are specifying a string or character vector, it must be in the `CuffDiff` native syntax (prefixed by one or two dashes) [1].

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Note The following list of arguments is a partial list. For the complete list, refer to the properties on page 1-635 of `CuffDiffOptions` object.

`ConditionLabels` — Sample labels
string | string vector | character vector | cell array of character vector

Sample labels, specified as a string, string vector, character vector, or cell array of character vectors. The number of labels must equal the number of samples or the value must be empty `[]`.

Example: `["Control","Mutant1","Mutant2"]`

Data Types: `string` | `char` | `cell`

`ContrastFile` — Contrast file name
string | character vector

Contrast file name, specified as a string or character vector. The file must be a two-column tab-delimited text file, where each line indicates two conditions to compare using `cuffdiff`. The condition labels in the file must match either the labels specified for `ConditionLabels` or the sample names. The file must have a single header line as the first line, followed by one line for each contrast. An example of the contrast file format follows.

condition_A	condition_B
Control	Mutant1
Control	Mutant2

If you do not provide this file, `cuffdiff` compares every pair of input conditions, which can impact performance.

Example: `"contrast.txt"`

Data Types: `char` | `string`

Properties

`ErrorHandler` — Function to handle errors from `run` method
function handle

Function to handle errors from the `run` method of the block, specified as a function handle. It specifies the function to call if the run method encounters an error within a pipeline. In order for the pipeline to continue after a block fails, `ErrorHandler` must return a structure compatible with the output ports of the block. The error handling function is called with the following two input arguments:

- Structure with the following fields:

Field	Description
identifier	Identifier of the error that occurred
message	Text of the error message

Field	Description
index	Linear index indicating which block process failed in the parallel run. By default, the index is always 1 because there is only one run per block. For details on how block inputs can be split across different dimensions for multiple run calls, see “Bioinformatics Pipeline SplitDimension”.

- Input structure passed to the `run` method when it failed.

Data Types: `function_handle`

`Inputs` — Input ports
structure

This property is read-only.

Input ports of the block, specified as a structure. The field names of the structure are the names of the block input ports and the field values are `bioinfo.pipeline.Input` objects. These objects describe the input port behaviors. The input port names are the expected field names of the input structure that you pass in for the block `run` method.

The `CuffDiff` block `Inputs` structure has the following fields:

- `GenomicAnnotationFile` — Name of the transcript annotation file. The file can be a GTF or GFF file produced by `Cufflinks`, `CuffCompare`, or another source of GTF annotations. This input is a required input that must be satisfied on page 1-1648.
- `GenomicAlignmentFiles` — Names of SAM, BAM, or CXB files containing alignment records for each sample. This input is a required input that must be satisfied on page 1-1648.

The default value for each input field is a `bioinfo.pipeline.datatypes.Unset` object, which means that the input value is not set yet.

Data Types: `struct`

`Outputs` — Output ports
structure

This property is read-only.

Output ports of the block, specified as a structure. The field names of the structure are the names of the block output ports and the field values are `bioinfo.pipeline.Output` objects. These objects describe the output port behaviors. The output structure returned by the block `run` method has the field names that are the same as the output port names.

The `CuffDiff` block `Outputs` structure has the following fields:

- `IsoformDiffFile` — Name of a file containing transcript-level differential expression results.
- `GeneDiffFile` — Name of a file containing gene-level differential expression results.
- `TSSDiffFile` — Name of a file containing primary transcript differential expression results.
- `CDSExpDiffFile` — Name of a file containing coding sequence differential expression results.
- `SplicingDiffFile` — Name of a file containing differential splicing results for isoforms.

- `CDSDiffFile` — Name of a file containing differential coding sequence output.
- `PromotersDiffFile` — Name of a file containing information on differential promoter use that exists between samples.

Tip To see the actual location of these files, first get the results of the block. Then use the `unwrap` method as shown in this example on page 1-44.

Data Types: `struct`

`Options` — CuffDiff options
`CuffDiffOptions` object (default)

CuffDiff options, specified as a `CuffDiffOptions` object. The default value is a default `CuffDiffOptions` object.

Object Functions

compile	Perform block-specific additional checks and validations
copy	Copy array of handle objects
emptyInputs	Create input structure for use with run method
eval	Evaluate block object
run	Run block object

Examples

Use `CuffDiff` Block to Perform Differential Expression

Perform differential expression transcripts using the provided SAM files which contain aligned reads from *Mycoplasma pneumoniae* from two samples.

```
import bioinfo.pipeline.blocks.*
import bioinfo.pipeline.Pipeline

FC1 = FileChooser(which("gyrAB.gtf"));
samFiles = {which("Myco_1_1.sam"),which("Myco_1_2.sam")};
FC2 = FileChooser(samFiles);
CD = CuffDiff;

P = Pipeline;
addBlock(P,[FC1,FC2,CD]);
connect(P,FC1,CD,["Files","GenomicAnnotationFile"]);
connect(P,FC2,CD,["Files","GenomicAlignmentFiles"]);

run(P);
R = results(P,CD)
```

```
R = 

  struct with fields:

      IsoformDiffFile: [1×1 bioinfo.pipeline.datatypes.File]
         GeneDiffFile: [1×1 bioinfo.pipeline.datatypes.File]
          TSSDiffFile: [1×1 bioinfo.pipeline.datatypes.File]
```

```
       CDSExpDiffFile: [1×1 bioinfo.pipeline.datatypes.File]
     SplicingDiffFile: [1×1 bioinfo.pipeline.datatypes.File]
          CDSDiffFile: [1×1 bioinfo.pipeline.datatypes.File]
    PromotersDiffFile: [1×1 bioinfo.pipeline.datatypes.File]
```

Call `unwrap` on each field of the result structure `R` to see the location of each output file. For example, to see the location of `IsoformDiffFile`, enter the following.

```
unwrap(R.IsoformDiffFile)

ans = 

    "C:\PipelineResults\CuffDiff_1\1\isoform_exp.diff"
```

Version History

Introduced in R2023a

References

[1] Trapnell, Cole, Brian A Williams, Geo Pertea, Ali Mortazavi, Gordon Kwan, Marijke J van Baren, Steven L Salzberg, Barbara J Wold, and Lior Pachter. “Transcript Assembly and Quantification by RNA-Seq Reveals Unannotated Transcripts and Isoform Switching during Cell Differentiation.” *Nature Biotechnology* 28, no. 5 (May 2010): 511–15.

See Also

`bioinfo.pipeline.Block` | `cuffdiff` | `bioinfo.pipeline.blocks.Cufflinks`

bioinfo.pipeline.blocks.Cufflinks

Bioinformatics pipeline block to assemble transcriptome from aligned reads

Description

A `Cufflinks` block enables you to assemble a transcriptome from aligned reads.

The block requires the Cufflinks Support Package for the Bioinformatics Toolbox. If the support package is not installed, then a download link is provided. For details, see “Bioinformatics Toolbox Software Support Packages”.

Creation

Syntax

```
b = bioinfo.pipeline.blocks.Cufflinks
b = bioinfo.pipeline.blocks.Cufflinks(options)
b = bioinfo.pipeline.blocks.Cufflinks(Name=Value)
```

Description

`b = bioinfo.pipeline.blocks.Cufflinks` creates a `Cufflinks` block.

`b = bioinfo.pipeline.blocks.Cufflinks(options)` also specifies additional `options`.

`b = bioinfo.pipeline.blocks.Cufflinks(Name=Value)` specifies additional options as the property names and values of a `CufflinksOptions` object. This object is set as the value of the `Options` property of the block.

Input Arguments

`options` — Cufflinks options
`CufflinksOptions` | string | character vector

Cufflinks options, specified as a `CufflinksOptions` object, string, or character vector.

If you are specifying a string or character vector, it must be in the `Cufflinks` native syntax (prefixed by one or two dashes) [1].

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Note The following list of arguments is a partial list. For the complete list, refer to the properties on page 1-680 of `CufflinksOptions` object.

`EffectiveLengthCorrection` — Flag to normalize fragment counts
`true` (default) | `false`

Flag to normalize fragment counts to fragments per kilobase per million mapped reads (FPKM), specified as `true` or `false`.

Example: `false`

Data Types: `logical`

`ExtraCommand` — Additional commands
`""` (default) | character vector | string

Additional commands, specified as a character vector or string.

The commands must be in the native syntax (prefixed by one or two dashes). Use this option to apply undocumented flags and flags without corresponding MATLAB properties.

When the software converts the original flags to MATLAB properties, it stores any unrecognized flags in this property.

Example: `'--library-type fr-secondstrand'`

Data Types: `char` | `string`

Properties

`ErrorHandler` — Function to handle errors from `run` method
function handle

Function to handle errors from the `run` method of the block, specified as a function handle. It specifies the function to call if the run method encounters an error within a pipeline. In order for the pipeline to continue after a block fails, `ErrorHandler` must return a structure compatible with the output ports of the block. The error handling function is called with the following two input arguments:

- Structure with the following fields:

Field	Description
identifier	Identifier of the error that occurred
message	Text of the error message
index	Linear index indicating which block process failed in the parallel run. By default, the index is always 1 because there is only one run per block. For details on how block inputs can be split across different dimensions for multiple run calls, see “Bioinformatics Pipeline SplitDimension”.

- Input structure passed to the `run` method when it failed.

Data Types: `function_handle`

`Inputs` — Input ports
structure

This property is read-only.

Input ports of the block, specified as a structure. The field names of the structure are the names of the block input ports and the field values are `bioinfo.pipeline.Input` objects. These objects describe the input port behaviors. The input port names are the expected field names of the input structure that you pass in for the block `run` method.

The `Cufflinks` block `Inputs` structure has the following field:

- `GenomicAlignmentFiles` — Names of SAM or BAM files containing alignment records for each sample. This input is a required input that must be satisfied on page 1-1648. The default value is a `bioinfo.pipeline.datatypes.Unset` object, which means that the input value is not set yet.

Data Types: `struct`

`Outputs` — Output ports
structure

This property is read-only.

Output ports of the block, specified as a structure. The field names of the structure are the names of the block output ports and the field values are `bioinfo.pipeline.Output` objects. These objects describe the output port behaviors. The output structure returned by the block `run` method has the field names that are the same as the output port names.

The `Cufflinks` block `Outputs` structure has the following fields:

- `TranscriptsGTFFile` — Transcript file name.
- `IsoformsFPKMFile` — Estimated isoform-level expression file name.
- `GenesFPKMFile` — Estimated gene-level expression file name.
- `SkippedTranscriptsGTFFile` — Name of the file containing skipped transcripts when processing a locus.

Tip To see the actual location of these files, first get the results of the block. Then use the `unwrap` method as shown in this example on page 1-49.

Data Types: `struct`

`Options` — Cufflinks options
`CufflinksOptions` object (default)

Cufflinks options, specified as a `CufflinksOptions` object. The default value is a default `CufflinksOptions` object.

Object Functions

compile	Perform block-specific additional checks and validations
copy	Copy array of handle objects
emptyInputs	Create input structure for use with run method
eval	Evaluate block object
run	Run block object

Examples

Use Cufflinks Block to Assemble Transcriptome

Assemble a transcriptome using the provided SAM files which contain aligned reads from *Mycoplasma pneumoniae* from two samples.

```
import bioinfo.pipeline.blocks.*
import bioinfo.pipeline.Pipeline

samFiles = {which("Myco_1_1.sam"),which("Myco_1_2.sam")};
FC = FileChooser(samFiles);
CL = Cufflinks;

P = Pipeline;
addBlock(P,[FC,CL]);
connect(P,FC,CL,["Files","GenomicAlignmentFiles"]);

run(P);
R = results(P,CL)
```

```
R = 

  struct with fields:

           TranscriptsGTFFile: [1×2 bioinfo.pipeline.datatypes.File]
             IsoformsFPKMFile: [1×2 bioinfo.pipeline.datatypes.File]
                GenesFPKMFile: [1×2 bioinfo.pipeline.datatypes.File]
    SkippedTranscriptsGTFFile: [1×2 bioinfo.pipeline.datatypes.File]
```

Call `unwrap` on each field of the result structure `R` to see the location of each output file. For example, to see the location of `TranscriptsGTFFile`, enter the following.

```
unwrap(R.TranscriptsGTFFile)'

ans = 

  2×1 string array

    "C:\PipelineResults\Cufflinks_1\1\Myco_1_1.transcripts.gtf"
    "C:\PipelineResults\Cufflinks_1\1\Myco_1_2.transcripts.gtf"
```

Version History

Introduced in R2023a

References

[1] Trapnell, Cole, Brian A Williams, Geo Pertea, Ali Mortazavi, Gordon Kwan, Marijke J van Baren, Steven L Salzberg, Barbara J Wold, and Lior Pachter. “Transcript Assembly and Quantification by RNA-Seq Reveals Unannotated Transcripts and Isoform Switching during Cell Differentiation.” *Nature Biotechnology* 28, no. 5 (May 2010): 511–15.

See Also

bioinfo.pipeline.Block | bioinfo.pipeline.blocks.CuffCompare | bioinfo.pipeline.blocks.CuffDiff | bioinfo.pipeline.blocks.CuffMerge | bioinfo.pipeline.blocks.CuffNorm | bioinfo.pipeline.blocks.CuffQuant | cufflinks

bioinfo.pipeline.blocks.CuffMerge

Bioinformatics pipeline block to merge transcript assemblies

Description

A `CuffMerge` block enables you to merge assembled transcripts from two or more GTF files.

The block requires the Cufflinks Support Package for the Bioinformatics Toolbox. If the support package is not installed, then a download link is provided. For details, see “Bioinformatics Toolbox Software Support Packages”.

Creation

Syntax

```
b = bioinfo.pipeline.blocks.CuffMerge
b = bioinfo.pipeline.blocks.CuffMerge(options)
b = bioinfo.pipeline.blocks.CuffMerge(Name=Value)
```

Description

`b = bioinfo.pipeline.blocks.CuffMerge` creates a `CuffMerge` block.

`b = bioinfo.pipeline.blocks.CuffMerge(options)` also specifies additional `options`.

`b = bioinfo.pipeline.blocks.CuffMerge(Name=Value)` specifies additional options as the property names and values of a `CuffMergeOptions` object. This object is set as the value of the `Options` property of the block.

Input Arguments

`options` — CuffMerge options
`CuffMergeOptions` | string | character vector

CuffMerge options, specified as a `CuffMergeOptions` object, string, or character vector.

If you are specifying a string or character vector, it must be in the `CuffMerge` native syntax (prefixed by one or two dashes) [1].

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Note The following list of arguments is a partial list. For the complete list, refer to the properties on page 1-698 of `CuffMergeOptions` object.

`ExtraCommand` — Additional commands
`""` (default) | string | character vector

The commands must be in the native syntax (prefixed by one or two dashes). Use this option to apply undocumented flags and flags without corresponding MATLAB properties.

When the software converts the original flags to MATLAB properties, it stores any unrecognized flags in this property.

Example: `'--library-type fr-secondstrand'`

Data Types: `char` | `string`

`IncludeAll` — Flag to use all object properties
`false` (default) | `true`

Flag to include all the object properties with the corresponding default values when converting to the original options syntax, specified as `true` or `false`. You can convert the properties to the original syntax prefixed by one or two dashes (such as `'-d 100 -e 80'`) by using `getCommand`. The default value `false` means that when you call `getCommand(optionsObject)`, it converts only the specified properties. If the value is `true`, `getCommand` converts all available properties, with default values for unspecified properties, to the original syntax.

Note If you set `IncludeAll` to `true`, the software translates all available properties, with default values for unspecified properties. The only exception is that when the default value of a property is `NaN`, `Inf`, `[]`, `''`, or `""`, then the software does not translate the corresponding property.

Example: `true`

Data Types: `logical`

Properties

`ErrorHandler` — Function to handle errors from `run` method
function handle

Function to handle errors from the `run` method of the block, specified as a function handle. It specifies the function to call if the run method encounters an error within a pipeline. In order for the pipeline to continue after a block fails, `ErrorHandler` must return a structure compatible with the output ports of the block. The error handling function is called with the following two input arguments:

- Structure with the following fields:

Field	Description
identifier	Identifier of the error that occurred
message	Text of the error message

Field	Description
index	Linear index indicating which block process failed in the parallel run. By default, the index is always 1 because there is only one run per block. For details on how block inputs can be split across different dimensions for multiple run calls, see “Bioinformatics Pipeline SplitDimension”.

- Input structure passed to the `run` method when it failed.

Data Types: `function_handle`

`Inputs` — Input ports
structure

This property is read-only.

Input ports of the block, specified as a structure. The field names of the structure are the names of the block input ports and the field values are `bioinfo.pipeline.Input` objects. These objects describe the input port behaviors. The input port names are the expected field names of the input structure that you pass in for the block `run` method.

The `CuffMerge` block `Inputs` structure has the following field:

- `GenomicAlignmentFiles` — Names of SAM or BAM files containing alignment records for each sample. This input is a required input that must be satisfied on page 1-1648. The default value is a `bioinfo.pipeline.datatypes.Unset` object, which means that the input value is not set yet.

Data Types: `struct`

`Outputs` — Output ports
structure

This property is read-only.

Output ports of the block, specified as a structure. The field names of the structure are the names of the block output ports and the field values are `bioinfo.pipeline.Output` objects. These objects describe the output port behaviors. The output structure returned by the block `run` method has the field names that are the same as the output port names.

The `CuffMerge` block `Outputs` structure has the following field:

- `MergedGTFFile` — Name of the output GTF file containing the merged transcriptome.

Tip To see the actual location of the output file, first get the results of the block. Then use the `unwrap` method as shown in this example on page 1-54.

Data Types: `struct`

`Options` — CuffMerge options
`CuffMergeOptions` object (default)

CuffMerge options, specified as a `CuffMergeOptions` object. The default value is a default `CuffMergeOptions` object.

Object Functions

compile	Perform block-specific additional checks and validations
copy	Copy array of handle objects
emptyInputs	Create input structure for use with run method
eval	Evaluate block object
run	Run block object

Examples

Use `CuffMerge` Block to Merge Assembled Transcripts

Merge GTF files that contain assembled isoforms.

```
import bioinfo.pipeline.blocks.*
import bioinfo.pipeline.Pipeline

gtfFiles = {which("Myco_1_1.transcripts.gtf"), which("Myco_1_2.transcripts.gtf")};
FC = FileChooser(gtfFiles);
CM = CuffMerge;

P = Pipeline;
addBlock(P,[FC,CM]);
connect(P,FC,CM,["Files","GenomicAnnotationFiles"]);

run(P);
R = results(P,CM)
```

```
R = 

  struct with fields:

    MergedGTFFile: [1×1 bioinfo.pipeline.datatypes.File]
```

Call `unwrap` on the field of the result structure `R`.

```
unwrap(R.MergedGTFFile)
```

```
ans = 

    "C:\PipelineResults\CuffMerge_1\1\merged_asm\merged.gtf"
```

Version History

Introduced in R2023a

References

[1] Trapnell, Cole, Brian A Williams, Geo Pertea, Ali Mortazavi, Gordon Kwan, Marijke J van Baren, Steven L Salzberg, Barbara J Wold, and Lior Pachter. "Transcript Assembly and Quantification

by RNA-Seq Reveals Unannotated Transcripts and Isoform Switching during Cell Differentiation." *Nature Biotechnology* 28, no. 5 (May 2010): 511–15.

See Also
`bioinfo.pipeline.Block` | `bioinfo.pipeline.blocks.Cufflinks` | `cuffmerge`

bioinfo.pipeline.blocks.CuffNorm

Bioinformatics pipeline block to normalize transcript expression levels

Description

A `CuffNorm` block enables you to generate expression tables which contain normalized expression level for each isoform, gene, transcript start site, and coding sequence based on library size.

The block requires the Cufflinks Support Package for the Bioinformatics Toolbox. If the support package is not installed, then a download link is provided. For details, see “Bioinformatics Toolbox Software Support Packages”.

Creation

Syntax

```
b = bioinfo.pipeline.blocks.CuffNorm
b = bioinfo.pipeline.blocks.CuffNorm(options)
b = bioinfo.pipeline.blocks.CuffNorm(Name=Value)
```

Description

`b = bioinfo.pipeline.blocks.CuffNorm` creates a `CuffNorm` block.

`b = bioinfo.pipeline.blocks.CuffNorm(options)` also specifies additional `options`.

`b = bioinfo.pipeline.blocks.CuffNorm(Name=Value)` specifies additional options as the property names and values of a `CuffNormOptions` object. This object is set as the value of the `Options` property of the block.

Input Arguments

`options` — CuffNorm options
`CuffNormOptions` | string | character vector

CuffNorm options, specified as a `CuffNormOptions` object, string, or character vector.

If you are specifying a string or character vector, it must be in the `CuffNorm` native syntax (prefixed by one or two dashes) [1].

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Note The following list of arguments is a partial list. For the complete list, refer to the properties on page 1-711 of `CuffNormOptions` object.

`ExtraCommand` — Additional commands
`""` (default) | string | character vector

The commands must be in the native syntax (prefixed by one or two dashes). Use this option to apply undocumented flags and flags without corresponding MATLAB properties.

When the software converts the original flags to MATLAB properties, it stores any unrecognized flags in this property.

Example: `'--library-type fr-secondstrand'`

Data Types: `char` | `string`

`IncludeAll` — Flag to use all object properties
`false` (default) | `true`

Flag to include all the object properties with the corresponding default values when converting to the original options syntax, specified as `true` or `false`. You can convert the properties to the original syntax prefixed by one or two dashes (such as `'-d 100 -e 80'`) by using `getCommand`. The default value `false` means that when you call `getCommand(optionsObject)`, it converts only the specified properties. If the value is `true`, `getCommand` converts all available properties, with default values for unspecified properties, to the original syntax.

Note If you set `IncludeAll` to `true`, the software translates all available properties, with default values for unspecified properties. The only exception is that when the default value of a property is `NaN`, `Inf`, `[]`, `''`, or `""`, then the software does not translate the corresponding property.

Example: `true`

Data Types: `logical`

Properties

`ErrorHandler` — Function to handle errors from `run` method
function handle

Function to handle errors from the `run` method of the block, specified as a function handle. It specifies the function to call if the run method encounters an error within a pipeline. In order for the pipeline to continue after a block fails, `ErrorHandler` must return a structure compatible with the output ports of the block. The error handling function is called with the following two input arguments:

- Structure with the following fields:

Field	Description
identifier	Identifier of the error that occurred
message	Text of the error message

Field	Description
index	Linear index indicating which block process failed in the parallel run. By default, the index is always 1 because there is only one run per block. For details on how block inputs can be split across different dimensions for multiple run calls, see “Bioinformatics Pipeline SplitDimension”.

- Input structure passed to the `run` method when it failed.

Data Types: `function_handle`

`Inputs` — Input ports
structure

This property is read-only.

Input ports of the block, specified as a structure. The field names of the structure are the names of the block input ports and the field values are `bioinfo.pipeline.Input` objects. These objects describe the input port behaviors. The input port names are the expected field names of the input structure that you pass in for the block `run` method.

The `CuffNorm` block `Inputs` structure has the following fields:

- `GenomicAnnotationFile` — Name of the transcript annotation file. The file can be a GTF or GFF file produced by `Cufflinks`, `CuffCompare`, or another source of GTF annotations. This input is a required input that must be satisfied on page 1-1648.
- `GenomicAlignmentFiles` — Names of SAM, BAM, or CXB files containing alignment records for each sample. This input is a required input that must be satisfied on page 1-1648.

The default value for each input field is a `bioinfo.pipeline.datatypes.Unset` object, which means that the input value is not set yet.

Data Types: `struct`

`Outputs` — Output ports
structure

This property is read-only.

Output ports of the block, specified as a structure. The field names of the structure are the names of the block output ports and the field values are `bioinfo.pipeline.Output` objects. These objects describe the output port behaviors. The output structure returned by the block `run` method has the field names that are the same as the output port names.

The `CuffNorm` block `Outputs` structure has the following fields:

- `IsoformFPKMFile` — Name of a file containing the normalized expression level for each isoform.
- `GeneFPKMFile` — Name of a file containing the normalized expression level for each gene.
- `TSSFPKMFile` — Name of a file containing the normalized expression level for each transcript start site (TSS).
- `CDSFPKMFile` — Name of a file containing the normalized expression level for each coding sequence.

Tip To see the actual location of the output file, first get the results of the block. Then use the `unwrap` method as shown in this example on page 1-59.

Data Types: `struct`

`Options` — CuffNorm options
`CuffNormOptions` object (default)

CuffNorm options, specified as a `CuffNormOptions` object. The default value is a default `CuffNormOptions` object.

Object Functions

compile	Perform block-specific additional checks and validations
copy	Copy array of handle objects
emptyInputs	Create input structure for use with run method
eval	Evaluate block object
run	Run block object

Examples

Use `CuffNorm` Block to Normalize Transcript Expressions

Generate normalized expression tables using `CuffNorm`.

```
import bioinfo.pipeline.blocks.*
import bioinfo.pipeline.Pipeline

FC1 = FileChooser(which("gyrAB.gtf"));
samFiles = {which("Myco_1_1.sam"),which("Myco_1_2.sam")};
FC2 = FileChooser(samFiles);
CN = CuffNorm;

P = Pipeline;
addBlock(P, [FC1,FC2,CN]);
connect(P, FC1,CN,["Files","GenomicAnnotationFile"]);
connect(P,FC2,CN,["Files","GenomicAlignmentFiles"]);

run(P);
R = results(P,CN)
```

```
R = 

  struct with fields:

    IsoformFPKMFile: [1×1 bioinfo.pipeline.datatypes.File]
       GeneFPKMFile: [1×1 bioinfo.pipeline.datatypes.File]
        TSSFPKMFile: [1×1 bioinfo.pipeline.datatypes.File]
        CDSFPKMFile: [1×1 bioinfo.pipeline.datatypes.File]
```

Call `unwrap` on each field of the result structure `R` to see the location of each output file. For example, to see the location of `IsoformFPKMFile`, enter the following.

```
unwrap(R.IsoformFPKMFile)
```

```
ans = 

    "C:\PipelineResults\CuffNorm_1\1\isoforms.fpkm_table"
```

Version History

Introduced in R2023a

References

[1] Trapnell, Cole, Brian A Williams, Geo Pertea, Ali Mortazavi, Gordon Kwan, Marijke J van Baren, Steven L Salzberg, Barbara J Wold, and Lior Pachter. “Transcript Assembly and Quantification by RNA-Seq Reveals Unannotated Transcripts and Isoform Switching during Cell Differentiation.” *Nature Biotechnology* 28, no. 5 (May 2010): 511–15.

See Also

`bioinfo.pipeline.Block` | `bioinfo.pipeline.blocks.Cufflinks` | `cuffnorm`

bioinfo.pipeline.blocks.CuffQuant

Bioinformatics pipeline block to quantify gene and transcript expression profiles

Description

A `CuffQuant` block enables you to quantify gene and transcript expression profiles by generating abundance estimates for the samples.

The block requires the Cufflinks Support Package for the Bioinformatics Toolbox. If the support package is not installed, then a download link is provided. For details, see "Bioinformatics Toolbox Software Support Packages".

Creation

Syntax

```
b = bioinfo.pipeline.blocks.CuffQuant
b = bioinfo.pipeline.blocks.CuffQuant(options)
b = bioinfo.pipeline.blocks.CuffQuant(Name=Value)
```

Description

`b = bioinfo.pipeline.blocks.CuffQuant` creates a `CuffQuant` block.

`b = bioinfo.pipeline.blocks.CuffQuant(options)` also specifies additional `options`.

`b = bioinfo.pipeline.blocks.CuffQuant(Name=Value)` specifies additional options as the property names and values of a `CuffQuantOptions` object. This object is set as the value of the `Options` property of the block.

Input Arguments

`options` — CuffQuant options
`CuffQuantOptions` | string | character vector

CuffQuant options, specified as a `CuffQuantOptions` object, string, or character vector.

If you are specifying a string or character vector, it must be in the `CuffQuant` native syntax (prefixed by one or two dashes) [1].

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Note The following list of arguments is a partial list. For the complete list, refer to the properties on page 1-724 of `CuffQuantOptions` object.

`EffectiveLengthCorrection` — Flag to normalize fragment counts
`true` (default) | false

Flag to normalize fragment counts to fragments per kilobase per million mapped reads (FPKM), specified as `true` or `false`.

Example: `false`

Data Types: `logical`

`ExtraCommand` — Additional commands
`""` (default) | string | character vector

The commands must be in the native syntax (prefixed by one or two dashes). Use this option to apply undocumented flags and flags without corresponding MATLAB properties.

When the software converts the original flags to MATLAB properties, it stores any unrecognized flags in this property.

Example: `'--library-type fr-secondstrand'`

Data Types: `char` | `string`

Properties

`ErrorHandler` — Function to handle errors from `run` method
function handle

Function to handle errors from the `run` method of the block, specified as a function handle. It specifies the function to call if the run method encounters an error within a pipeline. In order for the pipeline to continue after a block fails, `ErrorHandler` must return a structure compatible with the output ports of the block. The error handling function is called with the following two input arguments:

- Structure with the following fields:

Field	Description
identifier	Identifier of the error that occurred
message	Text of the error message
index	Linear index indicating which block process failed in the parallel run. By default, the index is always 1 because there is only one run per block. For details on how block inputs can be split across different dimensions for multiple run calls, see “Bioinformatics Pipeline SplitDimension”.

- Input structure passed to the `run` method when it failed.

Data Types: `function_handle`

`Inputs` — Input ports
structure

This property is read-only.

Input ports of the block, specified as a structure. The field names of the structure are the names of the block input ports and the field values are `bioinfo.pipeline.Input` objects. These objects describe the input port behaviors. The input port names are the expected field names of the input structure that you pass in for the block `run` method.

The `CuffQuant` block `Inputs` structure has the following fields:

- `GenomicAnnotationFile` — Name of the transcript annotation file. The file can be a GTF or GFF file produced by `Cufflinks`, `CuffCompare`, or another source of GTF annotations. This input is a required input that must be satisfied on page 1-1648.
- `GenomicAlignmentFiles` — Names of SAM or BAM files containing alignment records for each sample. This input is a required input that must be satisfied on page 1-1648.

The default value for each input field is a `bioinfo.pipeline.datatypes.Unset` object, which means that the input value is not set yet.

Data Types: `struct`

`Outputs` — Output ports
structure

This property is read-only.

Output ports of the block, specified as a structure. The field names of the structure are the names of the block output ports and the field values are `bioinfo.pipeline.Output` objects. These objects describe the output port behaviors. The output structure returned by the block `run` method has the field names that are the same as the output port names.

The `CuffQuant` block `Outputs` structure has the following field:

- `CXBFile` — Name of the abundances file.

Tip To see the actual location of the output file, first get the results of the block. Then use the `unwrap` method as shown in this example on page 1-63.

Data Types: `struct`

`Options` — CuffQuant options
`CuffQuantOptions` object (default)

CuffQuant options, specified as a `CuffQuantOptions` object. The default value is a default `CuffQuantOptions` object.

Object Functions

compile	Perform block-specific additional checks and validations
copy	Copy array of handle objects
emptyInputs	Create input structure for use with run method
eval	Evaluate block object
run	Run block object

Examples

Use CuffQuant Block to Quantify Transcript Expressions

Calculate abundances (expression levels) from aligned reads for samples using CuffQuant.

```
import bioinfo.pipeline.blocks.*
import bioinfo.pipeline.Pipeline

FC1 = FileChooser(which("gyrAB.gtf"));
samFiles = {which("Myco_1_1.sam"),which("Myco_1_2.sam")};
FC2 = FileChooser(samFiles);
CQ = CuffQuant;

P = Pipeline;
addBlock(P,[FC1,FC2,CQ]);
connect(P,FC1,CQ,["Files","GenomicAnnotationFile"]);
connect(P,FC2,CQ,["Files","GenomicAlignmentFiles"]);

run(P);
R = results(P,CQ)
```

```
R = 

  struct with fields:

    CXBFile: [1×1 bioinfo.pipeline.datatypes.File]
```

Call unwrap on the field of the result structure R.

```
unwrap(R.CXBFile)
```

```
ans = 

    "C:\PipelineResults\CuffQuant_1\1\abundances.cxb"
```

Version History

Introduced in R2023a

References

[1] Trapnell, Cole, Brian A Williams, Geo Pertea, Ali Mortazavi, Gordon Kwan, Marijke J van Baren, Steven L Salzberg, Barbara J Wold, and Lior Pachter. "Transcript Assembly and Quantification by RNA-Seq Reveals Unannotated Transcripts and Isoform Switching during Cell Differentiation." *Nature Biotechnology* 28, no. 5 (May 2010): 511–15.

See Also

bioinfo.pipeline.Block | bioinfo.pipeline.blocks.Cufflinks | cuffquant

bioinfo.pipeline.blocks.FeatureCount

Bioinformatics pipeline block to count reads mapped to genomic features

Description

A `FeatureCount` block enables you to compute the number of reads mapped to genomic features.

Creation

Syntax

```
b = bioinfo.pipeline.blocks.FeatureCount
b = bioinfo.pipeline.blocks.FeatureCount(options)
b = bioinfo.pipeline.blocks.FeatureCount(Name=Value)
```

Description

`b = bioinfo.pipeline.blocks.FeatureCount` creates a `FeatureCount` block.

`b = bioinfo.pipeline.blocks.FeatureCount(options)` also specifies additional `options`.

`b = bioinfo.pipeline.blocks.FeatureCount(Name=Value)` specifies additional options as the property names and values of a `FeatureCountOptions` object. This object is set as the value of the `Options` property of the block.

Input Arguments

`options` — FeatureCount options
`bioinfo.pipeline.options.FeatureCountOptions`

FeatureCount options, specified as a `FeatureCountOptions` object.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Note The following list of arguments is a partial list. For the complete list, refer to the properties on page 1-108 of `FeatureCountOptions` object.

`Feature` — Feature type
`'exon'` (default) | character vector | string

Feature type, specified as a character vector or string. This is used to decide what feature to consider from the GTF file. Default is `'exon'`.

`Metafeature` — Attribute type
`'gene_id'` (default) | character vector | string

Attribute type, specified as a character vector or string. This is used to decide what attribute to consider from the GTF file for grouping features into metafeatures and summarizing the read count.

Properties

`ErrorHandler` — Function to handle errors from `run` method
function handle

Function to handle errors from the `run` method of the block, specified as a function handle. It specifies the function to call if the run method encounters an error within a pipeline. In order for the pipeline to continue after a block fails, `ErrorHandler` must return a structure compatible with the output ports of the block. The error handling function is called with the following two input arguments:

- Structure with the following fields:

Field	Description
identifier	Identifier of the error that occurred
message	Text of the error message
index	Linear index indicating which block process failed in the parallel run. By default, the index is always 1 because there is only one run per block. For details on how block inputs can be split across different dimensions for multiple run calls, see “Bioinformatics Pipeline SplitDimension”.

- Input structure passed to the `run` method when it failed.

Data Types: `function_handle`

`Inputs` — Input ports
structure

This property is read-only.

Input ports of the block, specified as a structure. The field names of the structure are the names of the block input ports and the field values are `bioinfo.pipeline.Input` objects. These objects describe the input port behaviors. The input port names are the expected field names of the input structure that you pass in for the block `run` method.

The `FeatureCount` block `Inputs` structure has the following fields:

- `GTFFile` — GTF-formatted file name. This input is a required input that must be satisfied on page 1-1648.
- `GenomicAlignmentFiles` — Names of BAM- or SAM-formatted files. This input is a required input that must be satisfied on page 1-1648.

The default value for each input field is a `bioinfo.pipeline.datatypes.Unset` object, which means that the input value is not set yet.

Data Types: `struct`

`Outputs` — Output ports
structure

This property is read-only.

Output ports of the block, specified as a structure. The field names of the structure are the names of the block output ports and the field values are `bioinfo.pipeline.Output` objects. These objects describe the output port behaviors. The output structure returned by the block `run` method has the field names that are the same as the output port names.

The `FeatureCount` block `Outputs` structure has the following fields:

- `CountsTable` — Results containing sequence reads mapped to genomic features, returned as a table.
- `SummaryTable` — Summary of assigned and unassigned alignment entries, returned as a table.

Data Types: `struct`

`Options` — FeatureCount options
`bioinfo.pipeline.options.FeatureCountOptions` object (default)

FeatureCount options, specified as a `FeatureCountOptions` object. The default is a `FeatureCountOptions` object with default property values.

Object Functions

compile	Perform block-specific additional checks and validations
copy	Copy array of handle objects
emptyInputs	Create input structure for use with run method
eval	Evaluate block object
run	Run block object

Examples

Count Reads Using FeatureCount Block

Use a `FeatureCount` block to count reads that are mapped to exons and summarize the total number of reads at the gene level.

```
import bioinfo.pipeline.blocks.*
import bioinfo.pipeline.Pipeline

FC1 = FileChooser(which("Dmel_BDGP5_nohc.gtf"));
FC2 = FileChooser(which("rnaseq_sample1.sam"));
F = FeatureCount;

P = Pipeline;
addBlock(P,[FC1,FC2,F]);
connect(P,FC1,F,["Files","GTFFile"]);
connect(P,FC2,F,["Files","GenomicAlignmentFiles"]);

run(P);
```

```
Processing GTF file C:\Program Files\MATLAB\R2023a\toolbox\bioinfo\bioinfodata\Dmel_BDGP5_nohc.g
Processing SAM file C:\Program Files\MATLAB\R2023a\toolbox\bioinfo\bioinfodata\rnaseq_sample1.sa
Processing reference chr2L ...
Processing reference chr2R ...
Processing reference chr3L ...
Processing reference chr3R ...
Processing reference chr4 ...
Processing reference chrX ...
Done.
```

Get the block results.

```
R = results(P);
head(R.CountsTable)
```

```
          ID            Reference    rnaseq_sample1
    _______________    _________    ______________

    {'FBgn0002121'}    {'chr2L'}          9
    {'FBgn0067779'}    {'chr2L'}          2
    {'FBgn0005278'}    {'chr2L'}          4
    {'FBgn0031220'}    {'chr2L'}          4
    {'FBgn0025683'}    {'chr2L'}         13
    {'FBgn0053635'}    {'chr2L'}          2
    {'FBgn0016977'}    {'chr2L'}         22
    {'FBgn0086902'}    {'chr2L'}         27
```

```
R.SummaryTable
```

```
ans =

  9×1 table

                                  rnaseq_sample1
                                  ______________

    TotalEntries                      33354
    Assigned                          16399
    Unassigned_ambiguous                167
    Unassigned_filtered                   0
    Unassigned_lowMappingQuality          0
    Unassigned_multiMapped                0
    Unassigned_noFeature              16788
    Unassigned_supplementary              0
    Unassigned_unmapped                   0
```

Version History

Introduced in R2023a

See Also

bioinfo.pipeline.Block | FeatureCountOptions | bioinfo.pipeline.blocks.SeqFilter | bioinfo.pipeline.blocks.SeqSplit | bioinfo.pipeline.blocks.FileChooser

bioinfo.pipeline.blocks.FileChooser

Bioinformatics pipeline block to select files or URLs

Description

A `FileChooser` block enables you to select files or download files from URLs.

Creation

Syntax

```
fcBlock = bioinfo.pipeline.blocks.FileChooser
fcBlock = bioinfo.pipeline.blocks.FileChooser(fileNames)
```

Description

`fcBlock = bioinfo.pipeline.blocks.FileChooser` creates a `FileChooser` block.

`fcBlock = bioinfo.pipeline.blocks.FileChooser(fileNames)` also sets the `Files` property of the block to `fileNames`.

Input Arguments

`fileNames` — Names of files or URLs
string | character vector | ...

Names of files or URLs, specified as a string, character vector, string vector, or cell array of character vectors. You can include a full or relative file path. The block does not use the MATLAB path.

Data Types: `char` | `string` | `cell`

Properties

`ErrorHandler` — Function to handle errors from `run` method
function handle

Function to handle errors from the `run` method of the block, specified as a function handle. It specifies the function to call if the run method encounters an error within a pipeline. In order for the pipeline to continue after a block fails, `ErrorHandler` must return a structure compatible with the output ports of the block. The error handling function is called with the following two input arguments:

- Structure with the following fields:

Field	Description
identifier	Identifier of the error that occurred

Field	Description
message	Text of the error message
index	Linear index indicating which block process failed in the parallel run. By default, the index is always 1 because there is only one run per block. For details on how block inputs can be split across different dimensions for multiple run calls, see “Bioinformatics Pipeline SplitDimension”.

- Input structure passed to the `run` method when it failed.

Data Types: `function_handle`

`Files` — File names or URLs
empty string array (default) | string | character vector | ...

Names of files or URLs, specified as a string, character vector, string vector, or cell array of character vectors.

`Files` is always appended after `PathRoot` to determine the file or URL destinations. Files can include `file`, `http`, `https` as a scheme if `PathRoot` is empty.

Data Types: `char` | `string` | `cell`

`Inputs` — Input ports
structure

This property is read-only.

Input ports of the block, specified as a structure. The field names of the structure are the names of the block input ports and the field values are `bioinfo.pipeline.Input` objects. These objects describe the input port behaviors. The input port names are the expected field names of the input structure that you pass in for the block `run` method.

Data Types: `struct`

`Outputs` — Output ports
structure

This property is read-only.

Output ports of the block, specified as a structure. The field names of the structure are the names of the block output ports and the field values are `bioinfo.pipeline.Output` objects. These objects describe the output port behaviors. The output structure returned by the block `run` method has the field names that are the same as the output port names.

The `FileChooser` block `Outputs` structure has the following field:

- `Files` — Output file names.

  .

Tip To see the actual location of these files, first get the results of the block. Then use the `unwrap` method as shown in this example on page 1-75.

Data Types: `struct`

`Options` — Parameters for obtaining data from web server
`weboptions` object (default)

Parameters for obtaining data from a web server, specified as a `weboptions` object.

This property is used only when the scheme is `http` or `https`, or when `Files` contains a URL. The default value is a `weboptions` object with default property values.

`PathRoot` — Root path for `Files`
empty string array (default) | string | character vector

Root path for all files in the `Files` property, specified as a string or character vector.

`PathRoot` is always prefixed to `Files` to determine the file or URL destinations. `PathRoot` can include `file`, `http`, `https` as a scheme if `PathRoot` is empty.

Data Types: `char` | `string`

Object Functions

compile	Perform block-specific additional checks and validations
copy	Copy array of handle objects
emptyInputs	Create input structure for use with run method
eval	Evaluate block object
run	Run block object

Examples

Create a Simple Pipeline to Plot Sequence Quality Data

Import the Pipeline and block objects needed for the example.

```
import bioinfo.pipeline.Pipeline
import bioinfo.pipeline.blocks.*
```

Create a pipeline.

```
qcpipeline = Pipeline;
```

Select an input FASTQ file using a `FileChooser` block.

```
fastqfile = FileChooser(which("SRR005164_1_50.fastq"));
```

Create a `SeqFilter` block.

```
sequencefilter = SeqFilter;
```

Define the filtering threshold value. Specifically, filter out sequences with a total of more than 10 low-quality bases, where a base is considered a low-quality base if its quality score is less than 20.

```
sequencefilter.Options.Threshold = [10 20];
```

Add the blocks to the pipeline.

```
addBlock(qcpipeline,[fastqfile,sequencefilter]);
```

Connect the output of the first block to the input of the second block. To do so, you need to first check the input and output port names of the corresponding blocks.

View the `Outputs` (port of the first block) and `Inputs` (port of the second block).

```
fastqfile.Outputs
```

```
ans = struct with fields:
    Files: [1×1 bioinfo.pipeline.Output]
```

```
sequencefilter.Inputs
```

```
ans = struct with fields:
    FASTQFiles: [1×1 bioinfo.pipeline.Input]
```

Connect the `Files` output port of the `fastqfile` block to the `FASTQFiles` port of `sequencefilter` block.

```
connect(qcpipeline,fastqfile,sequencefilter,["Files","FASTQFiles"]);
```

Next, create a `UserFunction` block that calls the `seqqcplot` function to plot the quality data of the filtered sequence data. In this case, `inputFile` is the required argument for the `seqqcplot` function. The required argument name can be anything as long as it is a valid variable name.

```
qcplot = UserFunction("seqqcplot",RequiredArguments="inputFile",OutputArguments="figureHandle");
```

Alternatively, you can also use dot notation to set up your `UserFunction` block.

```
qcplot = UserFunction;
qcplot.RequiredArguments = "inputFile";
qcplot.Function = "seqqcplot";
qcplot.OutputArguments = "figureHandle";
```

Add the block.

```
addBlock(qcpipeline,qcplot);
```

Check the port names of `sequencefilter` block and `qcplot` block.

```
sequencefilter.Outputs
```

```
ans = struct with fields:
    FilteredFASTQFiles: [1×1 bioinfo.pipeline.Output]
         NumFilteredIn: [1×1 bioinfo.pipeline.Output]
        NumFilteredOut: [1×1 bioinfo.pipeline.Output]
```

```
qcplot.Inputs
```

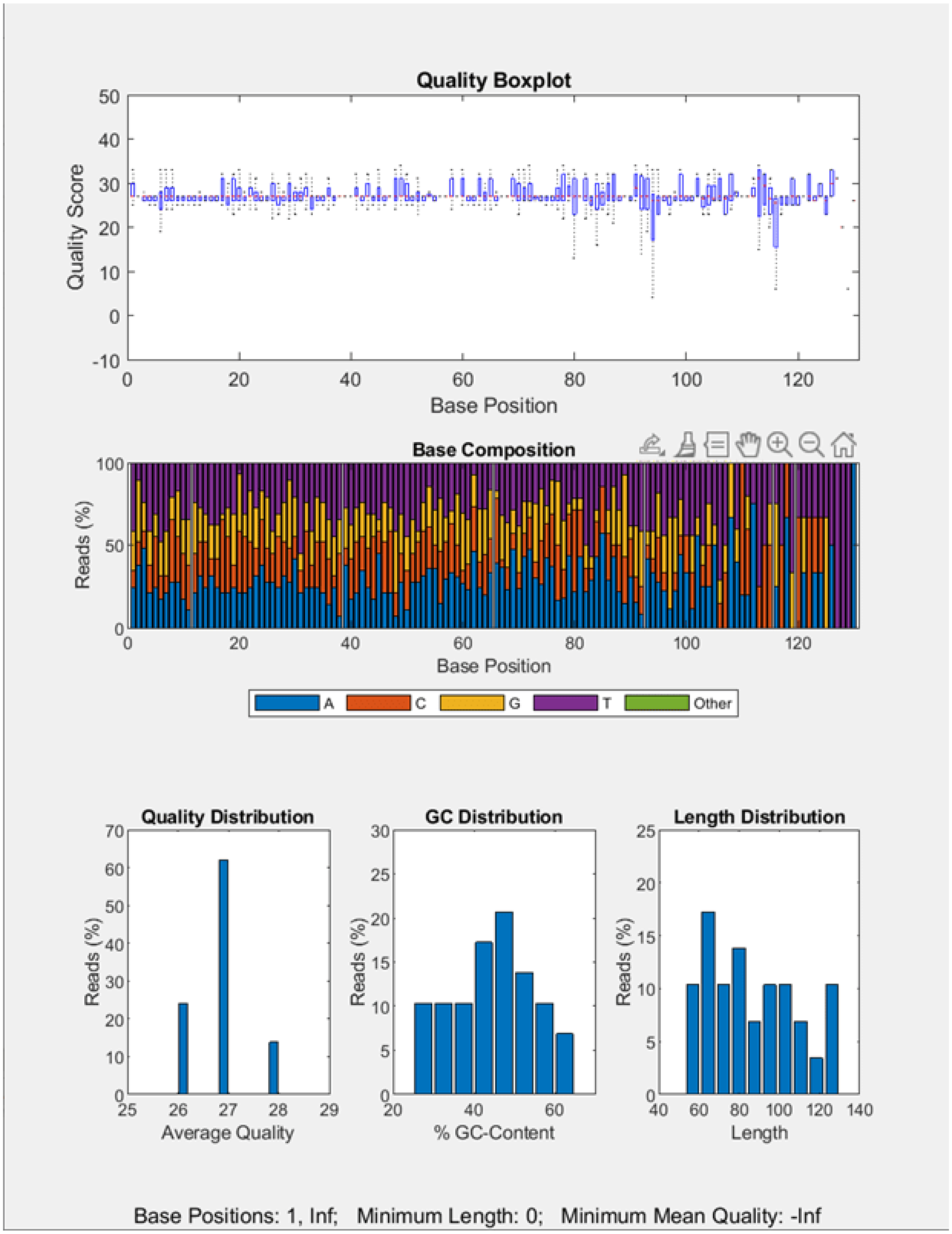
```
ans = struct with fields:
    inputFile: [1×1 bioinfo.pipeline.Input]
```

Connect the `FilteredFASTQFiles` port of the `sequencefilter` block to the `inputFile` port of the `qcplot` block.

```
connect(qcpipeline,sequencefilter,qcplot,["FilteredFASTQFiles","inputFile"]);
```

Run the pipeline to plot the sequence quality data.

```
run(qcpipeline);
```



Base Positions: 1, Inf; Minimum Length: 0; Minimum Mean Quality: -Inf

Select Input File Using FileChooser Block

Use a `FileChooser` block to select an input file provided with the toolbox.

```
import bioinfo.pipeline.blocks.FileChooser
import bioinfo.pipeline.Pipeline

FC = FileChooser(which("SRR6008575_10k_1.fq"));

P = Pipeline;
addBlock(P, FC);

run(P);
R = results(P, FC)
```

```
R = 

  struct with fields:

    Files: [1×1 bioinfo.pipeline.datatypes.File]
```

Call `unwrap` on `Files` to see the location of the file.

```
unwrap(R.Files)
```

```
ans = 

    "C:\Program Files\MATLAB\R2023a\toolbox\bioinfo\bioinfodata\SRR6008575_10k_1.fq"
```

Version History
Introduced in R2023a

See Also
`bioinfo.pipeline.Block`

bioinfo.pipeline.blocks.SamSort

Bioinformatics pipeline block to sort SAM files

Description

A `SamSort` block enables you to sort alignment records from SAM files by the reference sequence name first, and then by position within the reference.

Creation

Syntax

```
fcBlock = bioinfo.pipeline.blocks.SamSort
fcBlock = bioinfo.pipeline.blocks.SamSort(outFileName)
```

Description

`fcBlock = bioinfo.pipeline.blocks.SamSort` creates a `SamSort` block.

`fcBlock = bioinfo.pipeline.blocks.SamSort(outFileName)` specifies the output file name.

Input Arguments

`outFileName` — Name of output file
string | character vector

Name of the output file, specified as a string or character vector. The file extension must be `.sam`.

Data Types: `char` | `string`

Properties

`ErrorHandler` — Function to handle errors from `run` method
function handle

Function to handle errors from the `run` method of the block, specified as a function handle. It specifies the function to call if the run method encounters an error within a pipeline. In order for the pipeline to continue after a block fails, `ErrorHandler` must return a structure compatible with the output ports of the block. The error handling function is called with the following two input arguments:

- Structure with the following fields:

Field	Description
identifier	Identifier of the error that occurred
message	Text of the error message

Field	Description
index	Linear index indicating which block process failed in the parallel run. By default, the index is always 1 because there is only one run per block. For details on how block inputs can be split across different dimensions for multiple run calls, see “Bioinformatics Pipeline SplitDimension”.

- Input structure passed to the `run` method when it failed.

Data Types: `function_handle`

`Inputs` — Input ports
structure

This property is read-only.

Input ports of the block, specified as a structure. The field names of the structure are the names of the block input ports and the field values are `bioinfo.pipeline.Input` objects. These objects describe the input port behaviors. The input port names are the expected field names of the input structure that you pass in for the block `run` method.

The `SamSort` block `Inputs` structure has the following field:

- `SAMFile` — Name of a SAM file. This input is a required input that must be satisfied on page 1-1648. The default value is a `bioinfo.pipeline.datatypes.Unset` object, which means that the input value is not set yet.

Data Types: `struct`

`Outputs` — Output ports
structure

This property is read-only.

Output ports of the block, specified as a structure. The field names of the structure are the names of the block output ports and the field values are `bioinfo.pipeline.Output` objects. These objects describe the output port behaviors. The output structure returned by the block `run` method has the field names that are the same as the output port names.

The `SamSort` block `Outputs` structure has the following field:

- `SortedSAMFile` — Output file name. By default, it has the same base name as the input SAM file but with the extension `.sorted.sam`.

  .

  Tip To see the actual location of these files, first get the results of the block. Then use the `unwrap` method as shown in this example on page 1-78.

Data Types: `struct`

`OutFilename` — Output file name
string

Output file name, specified as a string. If it is empty, the output file will have the same base name as the input file with the extension `.sorted.sam`.

Data Types: `string`

Object Functions

compile	Perform block-specific additional checks and validations
copy	Copy array of handle objects
emptyInputs	Create input structure for use with run method
eval	Evaluate block object
run	Run block object

Examples

Sort SAM File Using SamSort Block

Sort a sample SAM file using a `SamSort` block.

```
import bioinfo.pipeline.blocks.*
import bioinfo.pipeline.Pipeline

FC = FileChooser(which("Myco_1_1.sam"));
SS = SamSort;
P = Pipeline;

addBlock(P,[FC,SS]);
connect(P,FC,SS,["Files","SAMFile"]);

run(P);
R = results(P,SS)
```

```
R = 

  struct with fields:

    SortedSAMFile: [1×1 bioinfo.pipeline.datatypes.File]
```

Call `unwrap` on `SortedSAMFile` to see the location of the output file.

```
unwrap(R.SortedSAMFile)
```

```
ans = 

    "C:\PipelineResults\SamSort_1\1\Myco_1_1.sorted.sam"
```

Version History

Introduced in R2023a

See Also

`bioinfo.pipeline.Block` | `bioinfo.pipeline.blocks.BamSort` | `samsort`

bioinfo.pipeline.blocks.SeqFilter

Bioinformatics pipeline block to filter sequences

Description

A `SeqFilter` block enables you to filter sequences based on a specified criterion.

Creation

Syntax

```
b = bioinfo.pipeline.blocks.SeqFilter
b = bioinfo.pipeline.blocks.SeqFilter(options)
b = bioinfo.pipeline.blocks.SeqFilter(Name=Value)
```

Description

`b = bioinfo.pipeline.blocks.SeqFilter` creates a `SeqFilter` block.

`b = bioinfo.pipeline.blocks.SeqFilter(options)` also specifies additional `options`.

`b = bioinfo.pipeline.blocks.SeqFilter(Name=Value)` specifies additional options as the property names and values of a `SeqFilterOptions` object. This object is set as the value of the `Options` property of the block.

Note The block always overwrites existing output files, unlike the `seqfilter` function.

Input Arguments

`options` — SeqFilter options
`bioinfo.pipeline.options.SeqFilterOptions`

SeqFilter options, specified as a `SeqFilterOptions` object.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Note The following list of arguments is a partial list. For the complete list, refer to the properties on page 1-114 of `SeqFilterOptions` object.

`Method` — Criterion to filter sequences
`'MaxNumberLowQualityBases'` (default) | `'MaxPercentLowQualityBases'` | `'MeanQuality'` | `'MinLength'`

Criterion to filter sequences, specified as one of the following options. Specify only one filtering criterion per function call.

- `'MaxNumberLowQualityBases'`– applies a maximum threshold on the number of low-quality bases allowed.
- `'MaxPercentLowQualityBases'`– applies a maximum threshold on the percentage of low-quality bases allowed.
- `'MeanQuality'`– applies a minimum threshold on the average base quality across each sequence.
- `'MinLength'`– applies a minimum threshold on the sequence length.

Use this name-value pair argument together with `'Threshold'` to specify the appropriate threshold value. Depending on the filtering criterion, the corresponding value for `'Threshold'` can be a scalar or two-element vector. See the `'Threshold'` option for the default values. If you do not specify `'Threshold'`, then the function uses the default threshold value of the specified method. For each filtering criterion, the function uses the base quality encoding format specified by the `'Encoding'` name-value pair argument.

`Threshold` — Threshold value for filtering criterion
scalar | vector

Threshold value for the filtering criterion, specified as a scalar or vector. Use this name-value pair to define the threshold value for the filtering criterion specified by `'Method'`.

Depending on the filtering criterion, the corresponding value for `'Threshold'` can be a scalar or two-element vector. If you do not specify `'Threshold'`, then the function uses the default threshold value of the corresponding method. For each filtering criterion, the function uses the encoding format of the base quality specified by the `'Encoding'` name-value pair argument.

`'Method'`	`'Threshold'`	Default `'Threshold'` value
`'MaxNumberLowQualityBases'`	Two-element vector `[V1 V2]`. *V1* is a nonnegative integer that specifies the maximum number of low-quality bases allowed. *V2* specifies the minimum base quality. Any base with quality less than *V2* is considered a low-quality base. Any sequence containing a number of low-quality bases greater than *V1* is filtered out and not saved in the output file.	`[0 10]`
`'MaxPercentLowQualityBases'`	Two-element vector `[V1 V2]`. *V1* is a scalar between 0 and 100 that specifies the maximum percentage of low-quality bases allowed. *V2* specifies the minimum base quality. Any base with quality less than *V2* is considered a low-quality base. Any sequence containing a percentage of low-quality bases greater than *V1* is filtered out and not saved in the output file.	`[0 10]`

'Method'	'Threshold'	Default 'Threshold' value
'MeanQuality'	Positive scalar that specifies the minimum threshold on the average base quality across each sequence. Any sequence with average base quality less than this value is filtered out.	0
'MinLength'	Nonnegative integer that specifies the minimum threshold on the sequence length allowed. Any sequence with length less than this value is filtered out.	1

Properties

`ErrorHandler` — Function to handle errors from `run` method

function handle

Function to handle errors from the `run` method of the block, specified as a function handle. It specifies the function to call if the run method encounters an error within a pipeline. In order for the pipeline to continue after a block fails, `ErrorHandler` must return a structure compatible with the output ports of the block. The error handling function is called with the following two input arguments:

- Structure with the following fields:

Field	Description
identifier	Identifier of the error that occurred
message	Text of the error message
index	Linear index indicating which block process failed in the parallel run. By default, the index is always 1 because there is only one run per block. For details on how block inputs can be split across different dimensions for multiple run calls, see “Bioinformatics Pipeline SplitDimension”.

- Input structure passed to the `run` method when it failed.

Data Types: `function_handle`

`Inputs` — Input ports

structure

This property is read-only.

Input ports of the block, specified as a structure. The field names of the structure are the names of the block input ports and the field values are `bioinfo.pipeline.Input` objects. These objects describe the input port behaviors. The input port names are the expected field names of the input structure that you pass in for the block `run` method.

The `SeqFilter` block `Inputs` structure has the following field:

- `FASTQFiles` — Names of FASTQ-formatted files with sequence and quality information. This input is a required input that must be satisfied on page 1-1648. The default value is a `bioinfo.pipeline.datatypes.Unset` object, which means that the input value is not set yet.

Data Types: `struct`

`Outputs` — Output ports
structure

This property is read-only.

Output ports of the block, specified as a structure. The field names of the structure are the names of the block output ports and the field values are `bioinfo.pipeline.Output` objects. These objects describe the output port behaviors. The output structure returned by the block `run` method has the field names that are the same as the output port names.

The `SeqFilter` block `Outputs` structure has the following fields:

- `FilteredFASTQFiles` — Output file names. By default, the name of each output file consists of the input file name followed by the output suffix (`'_filtered'`).

  Tip To see the actual location of these files, first get the results of the block. Then use the `unwrap` method as shown in this example on page 1-82.
- `NumFilteredIn` — Number of sequences selected from each input file, returned as a scalar or an *n*-by-1 vector where *n* is the number of input files. If there are multiple input files, the order in `NumFilteredIn` corresponds to the order of the input files.
- `NumFilteredOut` — Number of sequences excluded from each input file, returned as a scalar or an *n*-by-1 vector where *n* is the number of input files. If there are multiple input files, the order in `NumFilteredOut` corresponds to the order of the input files.

Data Types: `struct`

`Options` — `SeqFilter` options
`bioinfo.pipeline.options.SeqFilterOptions` object (default)

`SeqFilter` options, specified as a `SeqFilterOptions` object. The default value is a default `SeqFilterOptions` object.

Object Functions

compile	Perform block-specific additional checks and validations
copy	Copy array of handle objects
emptyInputs	Create input structure for use with run method
eval	Evaluate block object
run	Run block object

Examples

Filter Out Low Quality Sequences Using `SeqFilter` Block

Use a `SeqFilter` block to filter out sequences with low-quality bases, where a base is considered low-quality if its quality score is less than 15 (default).

```
import bioinfo.pipeline.blocks.*
import bioinfo.pipeline.Pipeline

FC = FileChooser(which("SRR005164_1_50.fastq"));
SF = SeqFilter;

P = Pipeline;
addBlock(P,[FC,SF]);
connect(P,FC,SF,["Files","FASTQFiles"]);

run(P);
R = results(P,SF)

R = 

  struct with fields:

    FilteredFASTQFiles: [1×1 bioinfo.pipeline.datatypes.File]
         NumFilteredIn: 3
        NumFilteredOut: 47
```

Call `unwrap` on `FilteredFASTQFiles` to see the location of the output file.

```
unwrap(R.FilteredFASTQFiles)

ans = 

    "C:\PipelineResults\SeqFilter_1\1\SRR005164_1_50_filtered.fastq"
```

Create a Simple Pipeline to Plot Sequence Quality Data

Import the Pipeline and block objects needed for the example.

```
import bioinfo.pipeline.Pipeline
import bioinfo.pipeline.blocks.*
```

Create a pipeline.

```
qcpipeline = Pipeline;
```

Select an input FASTQ file using a `FileChooser` block.

```
fastqfile = FileChooser(which("SRR005164_1_50.fastq"));
```

Create a `SeqFilter` block.

```
sequencefilter = SeqFilter;
```

Define the filtering threshold value. Specifically, filter out sequences with a total of more than 10 low-quality bases, where a base is considered a low-quality base if its quality score is less than 20.

```
sequencefilter.Options.Threshold = [10 20];
```

Add the blocks to the pipeline.

```
addBlock(qcpipeline,[fastqfile,sequencefilter]);
```

Connect the output of the first block to the input of the second block. To do so, you need to first check the input and output port names of the corresponding blocks.

View the `Outputs` (port of the first block) and `Inputs` (port of the second block).

```
fastqfile.Outputs
```

```
ans = struct with fields:
    Files: [1×1 bioinfo.pipeline.Output]
```

```
sequencefilter.Inputs
```

```
ans = struct with fields:
    FASTQFiles: [1×1 bioinfo.pipeline.Input]
```

Connect the `Files` output port of the `fastqfile` block to the `FASTQFiles` port of `sequencefilter` block.

```
connect(qcpipeline,fastqfile,sequencefilter,["Files","FASTQFiles"]);
```

Next, create a `UserFunction` block that calls the `seqqcplot` function to plot the quality data of the filtered sequence data. In this case, `inputFile` is the required argument for the `seqqcplot` function. The required argument name can be anything as long as it is a valid variable name.

```
qcplot = UserFunction("seqqcplot",RequiredArguments="inputFile",OutputArguments="figureHandle");
```

Alternatively, you can also use dot notation to set up your `UserFunction` block.

```
qcplot = UserFunction;
qcplot.RequiredArguments = "inputFile";
qcplot.Function = "seqqcplot";
qcplot.OutputArguments = "figureHandle";
```

Add the block.

```
addBlock(qcpipeline,qcplot);
```

Check the port names of `sequencefilter` block and `qcplot` block.

```
sequencefilter.Outputs
```

```
ans = struct with fields:
    FilteredFASTQFiles: [1×1 bioinfo.pipeline.Output]
         NumFilteredIn: [1×1 bioinfo.pipeline.Output]
        NumFilteredOut: [1×1 bioinfo.pipeline.Output]
```

```
qcplot.Inputs
```

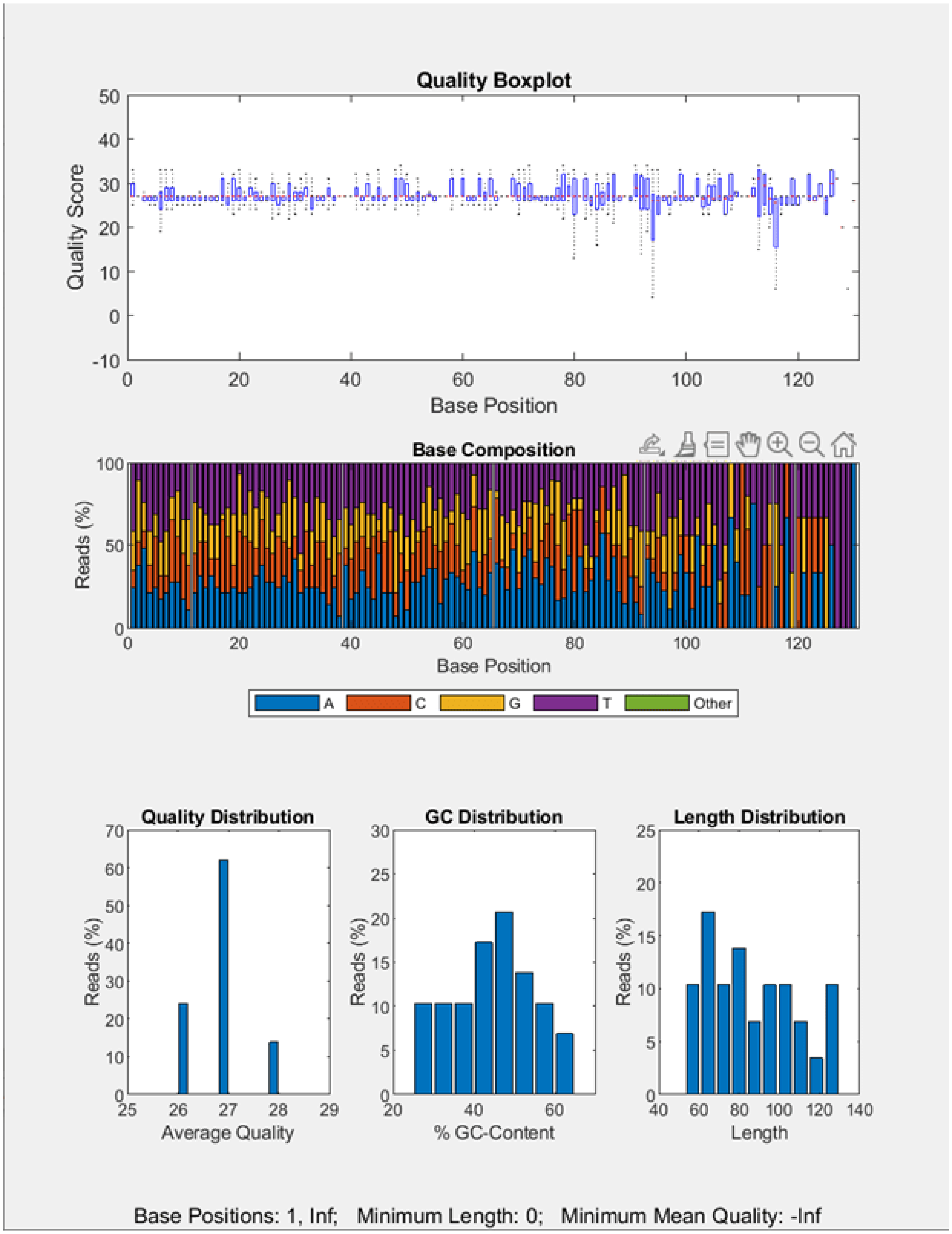
```
ans = struct with fields:
    inputFile: [1×1 bioinfo.pipeline.Input]
```

Connect the `FilteredFASTQFiles` port of the `sequencefilter` block to the `inputFile` port of the `qcplot` block.

```
connect(qcpipeline,sequencefilter,qcplot,["FilteredFASTQFiles","inputFile"]);
```

Run the pipeline to plot the sequence quality data.

```
run(qcpipeline);
```



Base Positions: 1, Inf;   Minimum Length: 0;   Minimum Mean Quality: -Inf

Version History

Introduced in R2023a

See Also

`bioinfo.pipeline.Block` | `SeqFilterOptions` | `bioinfo.pipeline.blocks.SeqSplit` | `bioinfo.pipeline.blocks.SeqTrim` | `bioinfo.pipeline.blocks.FileChooser` | `seqfilter`

bioinfo.pipeline.blocks.SeqSplit

Bioinformatics pipeline block to split sequences into separate files

Description

A `SeqSplit` block enables you to split sequences according to the provided barcodes and save the sequences in separate files.

Creation

Syntax

```
b = bioinfo.pipeline.blocks.SeqSplit
b = bioinfo.pipeline.blocks.SeqSplit(options)
b = bioinfo.pipeline.blocks.SeqSplit(Name=Value)
```

Description

`b = bioinfo.pipeline.blocks.SeqSplit` creates a `SeqSplit` block.

`b = bioinfo.pipeline.blocks.SeqSplit(options)` also specifies additional `options`.

`b = bioinfo.pipeline.blocks.SeqSplit(Name=Value)` specifies additional options as the property names and values of a `SeqSplitOptions` object. This object is set as the value of the `Options` property of the block.

Input Arguments

`options` — SeqSplit options
`bioinfo.pipeline.options.SeqSplitOptions`

SeqSplit options, specified as a `SeqSplitOptions` object. The default is a default `SeqSplitOptions` object.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Note The following list of arguments is a partial list. For the complete list, refer to the properties on page 1-117 of `SeqSplitOptions` object.

`MaxMismatches` — Maximum number of mismatches allowed during barcode matching
`0` (default) | nonnegative integer

Maximum number of mismatches allowed during barcode matching, specified as a nonnegative integer. The default is 0, that is, no mismatches are allowed.

`BarcodeFormat` — Type of barcode to match
5 (default) | 3

Type of barcode to match, specified as `3` or `5`. A value of `5` corresponds to the barcode located at the `5'` end of each sequence, and `3` corresponds to the `3'` end.

Example:

Properties

`ErrorHandler` — Function to handle errors from `run` method
function handle

Function to handle errors from the `run` method of the block, specified as a function handle. It specifies the function to call if the run method encounters an error within a pipeline. In order for the pipeline to continue after a block fails, `ErrorHandler` must return a structure compatible with the output ports of the block. The error handling function is called with the following two input arguments:

- Structure with the following fields:

Field	Description
identifier	Identifier of the error that occurred
message	Text of the error message
index	Linear index indicating which block process failed in the parallel run. By default, the index is always 1 because there is only one run per block. For details on how block inputs can be split across different dimensions for multiple run calls, see “Bioinformatics Pipeline SplitDimension”.

- Input structure passed to the `run` method when it failed.

Data Types: `function_handle`

`Inputs` — Input ports
structure

This property is read-only.

Input ports of the block, specified as a structure. The field names of the structure are the names of the block input ports and the field values are `bioinfo.pipeline.Input` objects. These objects describe the input port behaviors. The input port names are the expected field names of the input structure that you pass in for the block `run` method.

The `SeqSplit` block `Inputs` structure has the following fields:

- `FASTQFiles` — Names of FASTQ-formatted files with sequence and quality information. This input is a required input that must be satisfied on page 1-1648.

- `BarcodeFile` — Name of barcode file with barcode information. This input is a required input that must be satisfied on page 1-1648.

The default value for each input field is a `bioinfo.pipeline.datatypes.Unset` object, which means that the input value is not set yet.

Data Types: `struct`

`Outputs` — Output ports
structure

This property is read-only.

Output ports of the block, specified as a structure. The field names of the structure are the names of the block output ports and the field values are `bioinfo.pipeline.Output` objects. These objects describe the output port behaviors. The output structure returned by the block `run` method has the field names that are the same as the output port names.

The `SeqSplit` block `Outputs` structure has the following fields:

- `SplitFASTQFiles` — Output file names. By default, the name of each output file consists of the input file name followed by the output suffix (`'_split'`) and the barcode identifier.

  Tip To see the actual location of these files, first get the results of the block. Then use the `unwrap` method as shown in this example on page 1-90.

- `NumSplit` — Numbers of sequences saved in each output file, returned as a scalar or an *n*-by-1 vector, where *n* is the number of output files. If there are multiple output files, the order within *n* corresponds to the order of the output files.

Data Types: `struct`

`Options` — SeqSplit options
`bioinfo.pipeline.options.SeqSplitOptions` object (default)

SeqSplit options, specified as a `SeqSplitOptions` object. The default value is a default `SeqSplitOptions` object.

Object Functions

compile	Perform block-specific additional checks and validations
copy	Copy array of handle objects
emptyInputs	Create input structure for use with run method
eval	Evaluate block object
run	Run block object

Examples

Split Sequences Based on Barcodes

Use a `SeqSplit` block to split sequences into separate files based on barcodes.

```
import bioinfo.pipeline.blocks.*
import bioinfo.pipeline.Pipeline
```

```
%%
% Create a tab-delimited file with barcode info.
barcodeInfo = {'ID1','AAAAC';'ID2', 'AGATT';'ID3', 'GACTT'};
writetable(cell2table(barcodeInfo), 'barcodeExample.txt', ...
           'Delimiter','\t','WriteVariableNames',false);
%%
%  Create and add blocks to a pipeline.
FC1 = FileChooser(which("SRR005164_1_50.fastq"));
FC2 = FileChooser(which("barcodeExample.txt"));
SS = SeqSplit;
P = Pipeline;
addBlock(P,[FC1,FC2,SS]);
connect(P,FC1,SS,["Files","FASTQFiles"]);
connect(P,FC2,SS,["Files","BarcodeFile"]);

%%
% Run the pipeline and get the results.
run(P);
R = results(P,SS)
```

```
R = 

  struct with fields:

    SplitFASTQFiles: [3×1 bioinfo.pipeline.datatypes.File]
          NumSplit: [3×1 double]
```

Call `unwrap` on `SplitFASTQFiles` to see the location of the generated files.

```
unwrap(R.SplitFASTQFiles)
```

```
ans = 

  3×1 string array

    "C:\PipelineResults\SeqSplit_1\1\SRR005164_1_50_split_ID1.fastq"
    "C:\PipelineResults\SeqSplit_1\1\SRR005164_1_50_split_ID2.fastq"
    "C:\PipelineResults\SeqSplit_1\1\SRR005164_1_50_split_ID3.fastq"
```

Version History

Introduced in R2023a

See Also

`bioinfo.pipeline.Block` | `SeqSplitOptions` | `bioinfo.pipeline.blocks.SeqFilter` | `bioinfo.pipeline.blocks.SeqTrim` | `bioinfo.pipeline.blocks.FileChooser` | `seqsplit`

bioinfo.pipeline.blocks.SeqTrim

Bioinformatics pipeline block to trim sequences

Description

A `SeqTrim` block enables you to trim sequences based on a specified criterion.

Creation

Syntax

```
b = bioinfo.pipeline.blocks.SeqTrim
b = bioinfo.pipeline.blocks.SeqTrim(options)
b = bioinfo.pipeline.blocks.SeqTrim(Name=Value)
```

Description

`b = bioinfo.pipeline.blocks.SeqTrim` creates a `SeqTrim` block.

`b = bioinfo.pipeline.blocks.SeqTrim(options)` also specifies additional `options`.

`b = bioinfo.pipeline.blocks.SeqTrim(Name=Value)` specifies additional options as the property names and values of a `SeqTrimOptions` object. This object is set as the value of the `Options` property of the block.

Note The block always overwrites existing output files, unlike the `seqtrim` function.

Input Arguments

`options` — SeqTrim options
`bioinfo.pipeline.options.SeqTrimOptions`

SeqTrim options, specified as a `SeqTrimOptions` object.

Data Types: `char` | `string`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Note The following list of arguments is a partial list. For the complete list, refer to the properties on page 1-119 of `SeqTrimOptions` object.

Encoding — Base quality encoding format
'Illumina18' (default) | 'Sanger' | 'Solexa' | 'Illumina13' | 'Illumina15'

Base quality encoding format, specified as a character vector or string.

Method — Criterion to trim sequences
'MaxNumberLowQualityBases' (default) | 'MaxPercentLowQualityBases' | 'MeanQuality' | 'BasePositions' | 'Termini'

Criterion to trim sequences, specified as one of the following options. Specify only one trimming criterion per function call.

- 'MaxNumberLowQualityBases'– applies a maximum threshold on the number of low-quality bases allowed before trimming a sequence starting at the 5' end.
- 'MaxPercentLowQualityBases'– applies a maximum threshold on the percentage of low-quality bases allowed before trimming a sequence starting at the 5' end.
- 'MeanQuality'– applies a minimum threshold on the running average base quality allowed before trimming a sequence starting at the 5' end.
- 'BasePositions'– trims each sequence according to the base positions (first base and last base) starting at the 5' end.
- 'Termini'– trims each sequence from either the 5' or 3' end or from both ends.

Use this name-value pair argument together with 'Threshold' to specify the appropriate threshold value. Depending on the trimming criterion, the corresponding value for 'Threshold' varies. See the 'Threshold' option for the default values.

Note Sequences resulting in empty sequences after trimming are saved in the output files as empty sequences. To remove empty sequences from files, use the seqfilter function with the 'MinLength' option set to the value of 1.

Properties

ErrorHandler — Function to handle errors from run method
function handle

Function to handle errors from the run method of the block, specified as a function handle. It specifies the function to call if the run method encounters an error within a pipeline. In order for the pipeline to continue after a block fails, ErrorHandler must return a structure compatible with the output ports of the block. The error handling function is called with the following two input arguments:

- Structure with the following fields:

Field	Description
identifier	Identifier of the error that occurred
message	Text of the error message

Field	Description
index	Linear index indicating which block process failed in the parallel run. By default, the index is always 1 because there is only one run per block. For details on how block inputs can be split across different dimensions for multiple run calls, see “Bioinformatics Pipeline SplitDimension”.

- Input structure passed to the `run` method when it failed.

Data Types: `function_handle`

`Inputs` — Input ports
structure

This property is read-only.

Input ports of the block, specified as a structure. The field names of the structure are the names of the block input ports and the field values are `bioinfo.pipeline.Input` objects. These objects describe the input port behaviors. The input port names are the expected field names of the input structure that you pass in for the block `run` method.

The `SeqTrim` block `Inputs` structure has the following field:

- `FASTQFiles` — Names of FASTQ-formatted files with sequence and quality information. This input is a required input that must be satisfied on page 1-1648. The default value is a `bioinfo.pipeline.datatypes.Unset` object, which means that the input value is not set yet.

Data Types: `struct`

`Outputs` — Output ports
structure

This property is read-only.

Output ports of the block, specified as a structure. The field names of the structure are the names of the block output ports and the field values are `bioinfo.pipeline.Output` objects. These objects describe the output port behaviors. The output structure returned by the block `run` method has the field names that are the same as the output port names.

The `SeqTrim` block `Outputs` structure has the following fields:

- `TrimmedFASTQFiles` — Output file names. By default, the name of each output file consists of the input file name followed by the output suffix (`'_trimmed'`).

  Tip To see the actual location of these files, first get the results of the block. Then use the `unwrap` method as shown in this example on page 1-95.

- `NumTrimmed` — Number of sequences trimmed from each input file, returned as a scalar or an *n*-by-1 vector where *n* is the number of input files. If there are multiple input files, the order in `NumTrimmed` corresponds to the order of the input files.
- `NumUntrimmed` — Number of sequences untrimmed from each input file, returned as a scalar or an *n*-by-1 vector where *n* is the number of input files. If there are multiple input files, the order in `NumUntrimmed` corresponds to the order of the input files.

Data Types: `struct`

`Options` — `SeqTrim` options
`bioinfo.pipeline.options.SeqTrimOptions` object (default)

`SeqTrim` options, specified as a `SeqTrimOptions` object. The default value is a default `SeqTrimOptions` object.

Object Functions

compile	Perform block-specific additional checks and validations
copy	Copy array of handle objects
emptyInputs	Create input structure for use with run method
eval	Evaluate block object
run	Run block object

Examples

Trim Sequences Using `SeqTrim` Block

Use a `SeqTrim` block to start scanning the sequence at the 5' end and trim at the first base with a low quality score of 10 (default).

```
import bioinfo.pipeline.blocks.*
import bioinfo.pipeline.Pipeline

FC = FileChooser(which("SRR6008575_10k_1.fq"));
ST = SeqTrim;

P = Pipeline;
addBlock(P,[FC,ST]);
connect(P,FC,ST,["Files","FASTQFiles"]);

run(P);
R = results(P,ST)
```

```
R = 

  struct with fields:

    TrimmedFASTQFiles: [1×1 bioinfo.pipeline.datatypes.File]
           NumTrimmed: 1495
         NumUntrimmed: 8505
```

Call `unwrap` on `TrimmedFASTQFiles` to see the location of the output file.

```
unwrap(R.TrimmedFASTQFiles)
```

```
ans = 

    "C:\PipelineResults\SeqTrim_1\1\SRR6008575_10k_1_trimmed.fastq"
```

Version History

Introduced in R2023a

See Also

`bioinfo.pipeline.Block` | `SeqTrimOptions` | `bioinfo.pipeline.blocks.SeqFilter` | `bioinfo.pipeline.blocks.SeqSplit` | `bioinfo.pipeline.blocks.FileChooser` | `seqtrim`

bioinfo.pipeline.blocks.UserFunction

Bioinformatics pipeline block to call custom function

Description

A `UserFunction` block enables you to use any existing or custom function as a block in your pipeline, similar to any other built-in blocks.

Creation

Syntax

```
ufBlock = bioinfo.pipeline.blocks.UserFunction
ufBlock = bioinfo.pipeline.blocks.UserFunction(fcn)
ufBlock = bioinfo.pipeline.blocks.UserFunction(fcn,Name=Value)
```

Description

`ufBlock = bioinfo.pipeline.blocks.UserFunction` creates a `UserFunction` block.

`ufBlock = bioinfo.pipeline.blocks.UserFunction(fcn)` creates a `UserFunction` block from a custom function `fcn`, which can be a function handle, name of an existing or custom function, or function signature string.

`ufBlock = bioinfo.pipeline.blocks.UserFunction(fcn,Name=Value)` sets some of the block properties using one or more name-value arguments. `fcn` must be a function handle or name of a function.

Input Arguments

`fcn` — Custom function
function handle | string | character vector

Custom function, specified as a function handle, string or character vector representing the name of a function.

Data Types: `char` | `string` | `function_handle`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: `blosumBlock = UserFunction(@blosum62, OutputArguments="Matrix")` specifies to create a `UserFunction` block for the `blosum62` function using the string `"Matrix"` as the name of the block output port.

`RequiredArguments` — Names of required positional input arguments for custom function
string | character vector | ...

Names of the required positional input arguments for the custom function `fcn`, specified as a string, character vector, string vector, or cell array of character vectors. The order of arguments specified in this property is the same order used to call the underlying function `fcn`. The specified names are used as the names of required input ports of the block.

The corresponding input ports of the `UserFunction` block have the `Required` property set to true to indicate that these ports are required and must be satisfied on page 1-1648.

`OutputArguments` — Names of the outputs returned by custom function
string | character vector | ...

Names of the output arguments returned by the custom function `fcn`, specified as a string, character vector, string vector, or cell array of character vectors.

`NameValueArguments` — Names of optional name-value arguments for custom function
string | character vector | ...

Names of the optional name-value arguments for the custom function `fcn`, specified as a string, character vector, string vector, or cell array of character vectors.

The corresponding input ports of the `UserFunction` block have the `Required` property set to false to indicate that these ports are optional.

Properties

`ErrorHandler` — Function to handle errors from `run` method
function handle

Function to handle errors from the `run` method of the block, specified as a function handle. It specifies the function to call if the run method encounters an error within a pipeline. In order for the pipeline to continue after a block fails, `ErrorHandler` must return a structure compatible with the output ports of the block. The error handling function is called with the following two input arguments:

- Structure with the following fields:

Field	Description
identifier	Identifier of the error that occurred
message	Text of the error message
index	Linear index indicating which block process failed in the parallel run. By default, the index is always 1 because there is only one run per block. For details on how block inputs can be split across different dimensions for multiple run calls, see “Bioinformatics Pipeline SplitDimension”.

- Input structure passed to the `run` method when it failed.

Data Types: `function_handle`

`Function` — Function to evaluate
function handle | string scalar | character vector

Function to evaluate when you run the block, specified as a function handle, string scalar, or character vector that represents the name of any custom function.

When you call the `run` method with an input structure, it converts the input structure to positional and name-value arguments as determined by the `Signature` property and runs the specified function with those converted inputs. For examples, see "Create UserFunction Blocks For MATLAB Functions" on page 1-100.

Data Types: `char` | `string` | `function_handle`

`Inputs` — Input ports
structure

This property is read-only.

Input ports of the block, specified as a structure. The field names of the structure are the names of the block input ports and the field values are `bioinfo.pipeline.Input` objects. These objects describe the input port behaviors. The input port names are the expected field names of the input structure that you pass in for the block `run` method.

Data Types: `struct`

`NameValueArguments` — Name-value arguments for block `Function`
string | character vector | ...

Name-value arguments for the block `Function`, specified as a string, character vector, string vector, or cell array of character vectors. If you provide multiple name-value arguments, the `UseFunction` block sorts and stores them alphabetically.

Data Types: `char` | `string` | `cell`

`OutputArguments` — Names of output arguments of block `Function`
string | character vector | ...

Names of the output arguments of the block `Function`, returned as a string, character vector, string vector, or cell array of character vectors. The order of these names determines the order of outputs returned by the block.

The specified names are used as the names of required output ports of the block. Changing this property for an existing `UserFunction` block renames the block output ports and resets the order and number of output ports to match the new value.

Data Types: `char` | `string` | `cell`

`Outputs` — Output ports
structure

This property is read-only.

Output ports of the block, specified as a structure. The field names of the structure are the names of the block output ports and the field values are `bioinfo.pipeline.Output` objects. These objects describe the output port behaviors. The output structure returned by the block `run` method has the field names that are the same as the output port names.

Data Types: `struct`

`RequiredArguments` — Names of required positional arguments to block `Function`
string | character vector | ...

Names of the required positional input arguments to the block `Function`, specified as a string, character vector, string vector, or cell array of character vectors. The order of these names determines the order that the inputs are passed to the block `Function` when you call the block `run` method.

The specified names are used as the names of required input ports of the block. Changing this property for an existing `UserFunction` block renames the block input ports and resets the order and number of input ports to match the new value.

Data Types: `char` | `string` | `cell`

`Signature` — Block `Function` signature
string | character vector

Block `Function` signature, specified as a string or character vector. The `Signature` property defines how the underlying custom function is called when you run the block. In other words, the signature is typically similar to what you would enter at the MATLAB command line to run such a function. For example, the `Signature` to run the `aa2int` function with one input and one output argument would be: `"numbers = aa2int(Seq)"`, where *numbers* is an output variable and *Seq* is an input variable. For examples, see “Create UserFunction Blocks For MATLAB Functions” on page 1-100

If you specify the `Signature` property, other related properties, namely, `Function`, `RequiredArguments`, `NameValueArguments`, and `OutputArguments`, are automatically derived and set. Ensure that the signature you specify is a valid MATLAB expression containing one function call.

Data Types: `char` | `string`

Object Functions

compile	Perform block-specific additional checks and validations
copy	Copy array of handle objects
emptyInputs	Create input structure for use with run method
eval	Evaluate block object
run	Run block object

Examples

Create `UserFunction` Blocks For MATLAB Functions

You can create a `UserFunction` block for any existing or custom MATLAB function.

Create `UserFunction` for `size` Function

Create a `UserFunction` block for the `size` function with a single input and output.

```
ufSize = bioinfo.pipeline.blocks.UserFunction;
ufSize.Function = "size";
ufSize.RequiredArguments = "A";
ufSize.OutputArguments = "sz"
```

```
ufSize = 
  UserFunction with properties:

             Signature: "sz = size(A)"
     RequiredArguments: "A"
    NameValueArguments: [0×0 string]
       OutputArguments: "sz"
              Function: @size
                Inputs: [1×1 struct]
               Outputs: [1×1 struct]
          ErrorHandler: []
```

The `UserFunction` block is created. Next, create an input structure with the field name matching the input port name "`A`".

```
inStruct = struct("A",ones(2,3));
```

Run the block using the input structure. The block result is returned as a structure with the field named "`sz`", which matches the output port on the block.

```
sizeResult = run(ufSize,inStruct)
```

```
sizeResult = struct with fields:
    sz: [2 3]
```

Create `UserFunction` for `align2cigar` Function

Create a `UserFunction` block for the `align2cigar` function with two inputs and two outputs.

```
ufalign2cigar = bioinfo.pipeline.blocks.UserFunction;
ufalign2cigar.Function          = "align2cigar";
ufalign2cigar.RequiredArguments = ["alignment","ref"];
ufalign2cigar.OutputArguments   = ["cigars","starts"]
```

```
ufalign2cigar = 
  UserFunction with properties:

             Signature: "[cigars, starts] = align2cigar(alignment, ref)"
     RequiredArguments: [2×1 string]
    NameValueArguments: [0×1 string]
       OutputArguments: [2×1 string]
              Function: @align2cigar
                Inputs: [1×1 struct]
               Outputs: [1×1 struct]
          ErrorHandler: []
```

The `UserFunction` block is created with two input ports and two output ports, which are named after the inputs (`alignment` and `ref`) and outputs (`cigars` and `starts`) that you specified.

```
ufalign2cigar.RequiredArguments
```

```
ans = 2×1 string
    "alignment"
    "ref"
```

```
ufalign2cigar.OutputArguments
```

```
ans = 2×1 string
    "cigars"
    "starts"
```

Use `emptyInputs` to create an input structure with the fields automatically named after the block input ports.

```
inStruct = emptyInputs(ufalign2cigar)
```

```
inStruct = struct with fields:
    alignment: []
          ref: []
```

Set the values of the structure fields.

```
inStruct.alignment = ['ACG-ATGC'; 'ACGT-TGC'; '  GTAT-C'];
inStruct.ref       = 'ACGTATGC';
```

Run the block with the input structure. The block results are returned as a structure with the fields `cigars` and `starts`.

```
a2cResults = run(ufalign2cigar,inStruct)
```

```
a2cResults = struct with fields:
    cigars: {'3=1D4='  '4=1D3='  '4=1D1='}
    starts: [1 1 3]
```

Create for `samread`

Create a `UserFunction` block for the `samread` function that takes in multiple name-value arguments.

```
ufsamread = bioinfo.pipeline.blocks.UserFunction;
ufsamread.Function = "samread";
ufsamread.RequiredArguments  = "File";
ufsamread.OutputArguments    = ["samData","headerData"];
ufsamread.NameValueArguments = ["blockread","tags"]
```

```
ufsamread = 
  UserFunction with properties:

             Signature: "[samData, headerData] = samread(File, 'blockread', blockreadValue, 'tags
     RequiredArguments: "File"
    NameValueArguments: [2×1 string]
       OutputArguments: [2×1 string]
              Function: @samread
                Inputs: [1×1 struct]
               Outputs: [1×1 struct]
          ErrorHandler: []
```

Use `emptyInputs` with `IncludeOptional=true` so that the structure has the fields for the required input (`File`) and optional name-value arguments (`blockread` and `tags`).

```
inStruct = emptyInputs(ufsamread,IncludeOptional=true)
```

```
inStruct = struct with fields:
         File: []
    blockread: []
         tags: []
```

Set the input values. For the `File` input, use the provided SAM file. Read a block of sequence entries from 5 to 10 and exclude the tags.

```
inStruct.File = which("ex1.sam");
inStruct.blockread = [5 10];
inStruct.tags = false;
```

Run the block. The results are returned as a structure. `samData` field contains sequence alignment and mapping information from the SAM file. `headerData` contains the header information about the SAM file.

```
results = run(ufsamread,inStruct)
```

```
results = struct with fields:
       samData: [6×1 struct]
    headerData: [1×1 struct]
```

```
results.samData(1)
```

```
ans = struct with fields:
            QueryName: 'EAS56_59:8:38:671:758'
                 Flag: 137
        ReferenceName: 'seq1'
             Position: 9
       MappingQuality: 99
          CigarString: '35M'
    MateReferenceName: '*'
         MatePosition: 0
           InsertSize: 0
             Sequence: 'GCTCATTGTAAATGTGTGGTTTAACTCGTCCATGG'
              Quality: '<<<<<<<<<<<<<<<;<;7<<<<<<<<7<<;:<5%'
```

```
results.headerData.SequenceDictionary
```

```
ans = struct with fields:
        SequenceName: 'seq1'
    GenomeAssemblyID: 'HG18'
      SequenceLength: 62435964
```

Create a Simple Pipeline to Plot Sequence Quality Data

Import the Pipeline and block objects needed for the example.

```
import bioinfo.pipeline.Pipeline
import bioinfo.pipeline.blocks.*
```

Create a pipeline.

```
qcpipeline = Pipeline;
```

Select an input FASTQ file using a `FileChooser` block.

```
fastqfile = FileChooser(which("SRR005164_1_50.fastq"));
```

Create a `SeqFilter` block.

```
sequencefilter = SeqFilter;
```

Define the filtering threshold value. Specifically, filter out sequences with a total of more than 10 low-quality bases, where a base is considered a low-quality base if its quality score is less than 20.

```
sequencefilter.Options.Threshold = [10 20];
```

Add the blocks to the pipeline.

```
addBlock(qcpipeline,[fastqfile,sequencefilter]);
```

Connect the output of the first block to the input of the second block. To do so, you need to first check the input and output port names of the corresponding blocks.

View the `Outputs` (port of the first block) and `Inputs` (port of the second block).

```
fastqfile.Outputs
```

```
ans = struct with fields:
    Files: [1×1 bioinfo.pipeline.Output]
```

```
sequencefilter.Inputs
```

```
ans = struct with fields:
    FASTQFiles: [1×1 bioinfo.pipeline.Input]
```

Connect the `Files` output port of the `fastqfile` block to the `FASTQFiles` port of `sequencefilter` block.

```
connect(qcpipeline,fastqfile,sequencefilter,["Files","FASTQFiles"]);
```

Next, create a `UserFunction` block that calls the `seqqcplot` function to plot the quality data of the filtered sequence data. In this case, `inputFile` is the required argument for the `seqqcplot` function. The required argument name can be anything as long as it is a valid variable name.

```
qcplot = UserFunction("seqqcplot",RequiredArguments="inputFile",OutputArguments="figureHandle");
```

Alternatively, you can also use dot notation to set up your `UserFunction` block.

```
qcplot = UserFunction;
qcplot.RequiredArguments = "inputFile";
qcplot.Function = "seqqcplot";
qcplot.OutputArguments = "figureHandle";
```

Add the block.

```
addBlock(qcpipeline,qcplot);
```

Check the port names of `sequencefilter` block and `qcplot` block.

```
sequencefilter.Outputs
```

```
ans = struct with fields:
    FilteredFASTQFiles: [1×1 bioinfo.pipeline.Output]
         NumFilteredIn: [1×1 bioinfo.pipeline.Output]
        NumFilteredOut: [1×1 bioinfo.pipeline.Output]
```

```
qcplot.Inputs
```

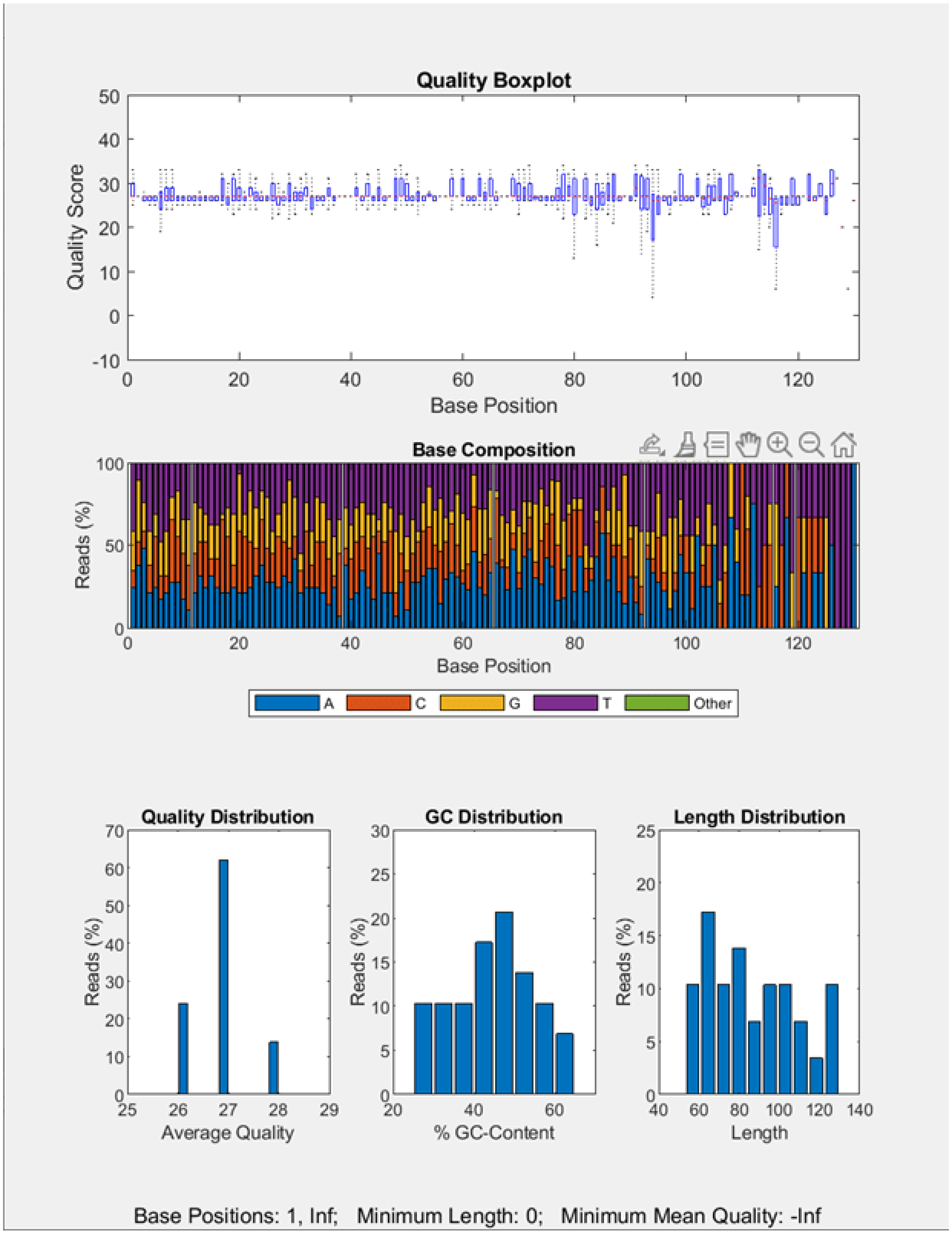
```
ans = struct with fields:
    inputFile: [1×1 bioinfo.pipeline.Input]
```

Connect the `FilteredFASTQFiles` port of the `sequencefilter` block to the `inputFile` port of the `qcplot` block.

```
connect(qcpipeline,sequencefilter,qcplot,["FilteredFASTQFiles","inputFile"]);
```

Run the pipeline to plot the sequence quality data.

```
run(qcpipeline);
```



Base Positions: 1, Inf; Minimum Length: 0; Minimum Mean Quality: -Inf

Version History

Introduced in R2023a

See Also

`bioinfo.pipeline.Block`

FeatureCountOptions

Contain options to compute number of reads mapped to genomic features

Description

A `FeatureCountOptions` object contains options to compute the number of reads mapped to genomic features. This object is used as the value of `Options` property of the `bioinfo.pipeline.blocks.FeatureCount` block.

Creation

Syntax

```
optionsObj = bioinfo.pipeline.options.FeatureCountOptions
optionsObj = bioinfo.pipeline.options.FeatureCountOptions(Name=Value)
```

Description

`optionsObj = bioinfo.pipeline.options.FeatureCountOptions` creates a `FeatureCountOptions` object with default property values.

`optionsObj = bioinfo.pipeline.options.FeatureCountOptions(Name=Value)` sets properties on page 1-108 using one or more name-value arguments. `Name` is the property name and `Value` is the property value. For example, `ssOpt = bioinfo.pipeline.options.FeatureCountOptions(CountFragments=true)` specifies to count reads as pairs of mates.

Properties

`Feature` — Feature type
`'exon'` (default) | character vector | string

Feature type, specified as a character vector or string. This is used to decide what feature to consider from the GTF file. Default is `'exon'`.

`Metafeature` — Attribute type
`'gene_id'` (default) | character vector | string

Attribute type, specified as a character vector or string. This is used to decide what attribute to consider from the GTF file for grouping features into metafeatures and summarizing the read count.

`Summarization` — Boolean variable indicating whether to summarize at the metafeature level
`true` (default) | `false`

Boolean variable indicating whether to summarize at the metafeature level, specified as `true` or `false`.

Default is `true`, meaning the function groups features into metafeatures and reports the read counts for metafeatures.

`Alias` — Name of file containing aliases of reference names
character vector | string

Name of file containing aliases of reference names, specified as a character vector or string. The file must be a tab-delimited file where the first column corresponds to the reference names used in the GTF file, and the second column corresponds to the reference names used in the input file(s). The names are case-sensitive. It is necessary to include only the reference names that are different in the GTF file and the input file. The file must contain only one alias term for any reference listed in the input file. By default, the reference names in the GTF file and those in the input files are assumed to be the same.

`CountFragments` — Boolean variable indicating whether to count reads as pairs of mates
`false` (default) | `true`

Boolean variable indicating whether to count reads as fragments, specified as `true` or `false`. Paired-end reads must have the same ID for the field `QNAME` in the input file, and the mutual order of mates is inferred by the appropriate bit in the `FLAG` field within the input file. Reads that have no valid mate either because the mate is unmapped or filtered out by input criteria are still counted if they satisfy the overlapping criteria.

Default is `false`, that is, the reads are counted as single-end reads, and their pairing information is ignored.

`StrandSpecificity` — Strand specificity of sequencing protocol
`'unstranded'` (default) | `'stranded'` | `'reverse'`

Strand specificity of the sequencing protocol, specified as `'unstranded'` (default), `'stranded'`, or `'reverse'`.

- If `'unstranded'`, the strand of the reads (or fragments) is ignored.
- If `'stranded'`, the strand of the reads (or fragments) is considered, and only those having the same strand as the feature they overlap are counted.
- If `'reverse'`, the opposite direction of the strand of the reads (or fragments) is considered, and only those having the opposite strand as the feature they overlap are counted.

When counting fragments (paired-end reads), the strand of the first mate is considered as the strand of the whole fragment. The mutual order of mates (first or second) is inferred from the appropriate bit in the `FLAG` field of the input file.

`MinOverlap` — Minimum number of overlapped bases required
`1` (default) | positive integer

Minimum number of overlapped bases required to assign a read to a feature, specified as a positive integer. When counting fragments, the sum of the overlaps from each end is used as the minimum number of overlapped bases.

`MinMappingQuality` — Minimum mapping quality for a given read
`0` (default) | non-negative integer

Minimum mapping quality for a given read to be considered for counting, specified as a non-negative integer. This corresponds to the `MAPQ` field in the input file. If counting fragments, at least one of the read mates must satisfy this criterion in order to be considered for counting.

`CountMultiOverlap` — Boolean variable indicating whether to count reads overlapping multiple features
`false` (default) | `true`

Boolean variable indicating whether to count reads overlapping multiple features, specified as `true` or `false` (default).

If `true`, a read (or fragment) overlapping multiple features is counted multiple times. During summarization at the metafeature level, a read (or fragment) is counted only once if it overlaps with multiple features belonging to the same metafeature as long as it does not overlap with other metafeaures.

`CountMultiMapped` — Counting option for reads having multiple mapping locations in the input file
`'primary'` (default) | `'none'` | `'all'`

Counting option for reads having multiple mapping locations in the input file, specified as `'primary'` (default), `'none'`, or `'all'`.

- If `'primary'`, only the primary alignment of a multi-mapped read is considered. The appropriate bit in the input file is used to identify primary alignments.
- If `'none'`, all alignments of a multi-mapped read are ignored. The NH tag is used to identify multi-mapped reads.
- If `'all'`, all alignments of a multi-mapped read are considered and counted multiple times.

`BothEndsMapped` — Boolean variable indicating whether a fragment must have both mates mapped
`false` (default) | `true`

Boolean variable indicating whether a fragment must have both mates mapped, specified as `true` or `false`. Mate mapping information is retrieved from the `FLAG` field in the input file. Default is `false`.

`ProperlyPaired` — Boolean variable indicating whether a fragment must be properly paired
`false` (default) | `true`

Boolean variable indicating whether a fragment must be properly paired, specified as `true` or `false`. Mate pairing information is retrieved from the `FLAG` field in the input file. Default is `false`.

`ShowZeroCounts` — Boolean variable indicating whether to report features or metafeatures with zero count
`false` (default) | `true`

Boolean variable indicating whether to report features or metafeatures with zero count for every input file in the output table, specified as `true` or `false`.

Default is `false`, that is, only rows with non-zero counts and columns with non-zero counts are included in the output table.

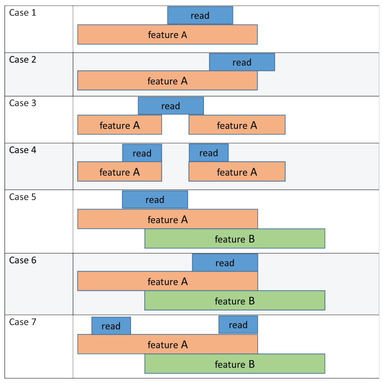
`OverlapMethod` — Method to use when assigning a given read to metafeature
`'partial'` (default) | `'full'` | `'max'` | `'hits'`

Method to use when assigning a given read to metafeature, specified as `'partial'`, `'full'`, `'max'`, or `'hits'`. If `'Summarization'` is set to `false`, then the reads are assigned to features, instead of metafeatures, based on the specified method.

In the following table, *R* refers to a read or fragment, and *M* refers to a metafeature.

Method	Description
`'partial'`	*R* is assigned to *M* if *R* overlaps (even partially) only with *M*. Otherwise *R* is considered ambiguous.
`'full'`	*R* is assigned to *M* if *R* is completely mapped only within *M*, that is, fully overlapping only M. Otherwise *R* is considered ambiguous
`'max'`	*R* is assigned to *M* if *R* satisfies the overlapping criteria only with M, or if *R* satisfies the overlapping criteria with several metafeatures but overlaps fully only with M.
`'hits'`	*R* is assigned to *M* if *R* overlaps even partially only *M*, or if *M* is the only metafeature with the highest number of features hit by *R*; otherwise *R* is considered ambiguous.

The following schematic diagram and table illustrate the outcome of these methods in conjunction with the `'CountMultiOverlap'` name-value pair argument. In the figure, the read refers to a short-read sequence from an input file, and feature A and feature B refers to features listed in a GTF file.



Each method column lists the feature that the read is assigned to based on the corresponding method. The `'CountMultiOverlap'` column indicates whether this name-value pair is set to `true` or `false` and if it has any effect in the outcome of each method.

	`'CountMultiOverlap'`	`'partial'`	`'full'`	`'max'`	`'hits'`
Case 1	No effect since the read maps only to one feature (feature A).	feature A	feature A	feature A	feature A

	`'CountMultiOverlap'`	`'partial'`	`'full'`	`'max'`	`'hits'`
Case 2	No effect since the read maps only to one feature (feature A).	feature A	no feature	feature A	feature A
Case 3	No effect since the read maps only to one feature (feature A).	feature A	no feature	feature A	feature A
Case 4	No effect since the read maps only to one feature (feature A).	feature A	feature A	feature A	feature A
Case 5	`false`	ambiguous	feature A	feature A	ambiguous
	`true`	feature A, feature B	feature A	feature A	feature A, feature B
Case 6	`false`	ambiguous	ambiguous	ambiguous	ambiguous
	`true`	feature A, feature B	feature A, feature B	feature A, feature B	feature A, feature B
Case 7	`false`	Ambiguous	feature A	feature A	feature A
	`true`	feature A, feature B	feature A	feature A	feature A

no feature means that the read is not assigned to any feature. If you have specified the second output table `S`, its `Unassigned_noFeature` row is incremented by one for such occurrence. *ambiguous* means that the read is not assigned to any feature since it satisfies the overlapping criteria for multiple features, and the `Unassigned_ambiguous` row is incremented by one for such occurrence.

`UseParallel` — Boolean variable indicating whether to compute in parallel
`false` (default) | `true`

Boolean variable indicating whether to compute in parallel, specified as `true` or `false`.

In order to execute the computation in parallel, you must have Parallel Computing Toolbox™. If a MATLAB parallel pool does not exist, one is automatically created when the auto-creation option is enabled in your parallel preferences. Otherwise, computation runs in the serial mode.

Default is `false`, that is, serial mode.

`Verbose` — Boolean variable indicating whether to display the progress of computation
`true` (default) | `false`

Boolean variable indicating whether to display the progress of computation, specified as `true` or `false`.

Version History

Introduced in R2023a

See Also

`bioinfo.pipeline.blocks.FeatureCount` | `featurecount` | `bioinfo.pipeline.blocks.SeqSplit` | `bioinfo.pipeline.blocks.SeqFilter` | `bioinfo.pipeline.blocks.SeqTrim`

SeqFilterOptions

Contain options to filter sequences

Description

A `SeqFilterOptions` object contains options to filter sequences based on a specified criterion. This object is used as the value of `Options` property of the `bioinfo.pipeline.blocks.SeqFilter` block.

Creation

Syntax

```
optionsObj = bioinfo.pipeline.options.SeqFilterOptions
optionsObj = bioinfo.pipeline.options.SeqFilterOptions(Name=Value)
```

Description

`optionsObj = bioinfo.pipeline.options.SeqFilterOptions` creates a `SeqFilterOptions` object with default property values.

`optionsObj = bioinfo.pipeline.options.SeqFilterOptions(Name=Value)` sets properties on page 1-114 using one or more name-value arguments. `Name` is the property name and `Value` is the property value. For example, `optionsObj = bioinfo.pipeline.options.SeqFilterOptions(Threshold=[5 15])` specifies to filter out sequences with a total of more than five low-quality bases, where a base is low quality if its score is less than 15.

Properties

`Method` — Criterion to filter sequences
`'MaxNumberLowQualityBases'` (default) | `'MaxPercentLowQualityBases'` | `'MeanQuality'` | `'MinLength'`

Criterion to filter sequences, specified as one of the following options. Specify only one filtering criterion per function call.

- `'MaxNumberLowQualityBases'`– applies a maximum threshold on the number of low-quality bases allowed.
- `'MaxPercentLowQualityBases'`– applies a maximum threshold on the percentage of low-quality bases allowed.
- `'MeanQuality'`– applies a minimum threshold on the average base quality across each sequence.
- `'MinLength'`– applies a minimum threshold on the sequence length.

Use this name-value pair argument together with `'Threshold'` to specify the appropriate threshold value. Depending on the filtering criterion, the corresponding value for `'Threshold'` can be a scalar or two-element vector. See the `'Threshold'` option for the default values. If you do not specify `'Threshold'`, then the function uses the default threshold value of the specified method. For each filtering criterion, the function uses the base quality encoding format specified by the `'Encoding'` name-value pair argument.

`Threshold` — Threshold value for filtering criterion
scalar | vector

Threshold value for the filtering criterion, specified as a scalar or vector. Use this name-value pair to define the threshold value for the filtering criterion specified by `'Method'`.

Depending on the filtering criterion, the corresponding value for `'Threshold'` can be a scalar or two-element vector. If you do not specify `'Threshold'`, then the function uses the default threshold value of the corresponding method. For each filtering criterion, the function uses the encoding format of the base quality specified by the `'Encoding'` name-value pair argument.

`'Method'`	`'Threshold'`	Default `'Threshold'` value
`'MaxNumberLowQualityBases'`	Two-element vector `[`*V1 V2*`]`. *V1* is a nonnegative integer that specifies the maximum number of low-quality bases allowed. *V2* specifies the minimum base quality. Any base with quality less than *V2* is considered a low-quality base. Any sequence containing a number of low-quality bases greater than *V1* is filtered out and not saved in the output file.	`[0 10]`
`'MaxPercentLowQualityBases'`	Two-element vector `[`*V1 V2*`]`. *V1* is a scalar between 0 and 100 that specifies the maximum percentage of low-quality bases allowed. *V2* specifies the minimum base quality. Any base with quality less than *V2* is considered a low-quality base. Any sequence containing a percentage of low-quality bases greater than *V1* is filtered out and not saved in the output file.	`[0 10]`
`'MeanQuality'`	Positive scalar that specifies the minimum threshold on the average base quality across each sequence. Any sequence with average base quality less than this value is filtered out.	`0`
`'MinLength'`	Nonnegative integer that specifies the minimum threshold on the sequence length allowed. Any sequence with length less than this value is filtered out.	`1`

`WindowSize` — Size of sliding window to apply filtering criterion to sequence
`Inf` (default) | positive integer

Size of the sliding window to apply the filtering criterion to a sequence, specified as a positive integer. The size of the window corresponds to the number of bases that the function uses at one time to apply the criterion. If any window fails the criterion, the whole sequence is discarded.

The default is `Inf`, that is, the filtering criterion is applied to the whole sequence.

`Encoding` — Base quality encoding format
`'Illumina18'` (default) | `'Sanger'` | `'Solexa'` | `'Illumina13'` | `'Illumina15'`

Base quality encoding format, specified as a character vector or string.

`OutputSuffix` — Suffix to use in output file name
`'_filtered'` (default) | character vector | string

Suffix to use in the output file name, specified as a character vector or string. It is inserted after the input file name and before the file extension. The default is `'_filtered'`.

`PairedFiles` — Whether to consider input files as pairs for paired-end sequence data
`false` (default) | `true`

Whether to consider the input files as pairs for paired-end sequence data, specified as `true` or `false`.

If `true`, the input files are read as pairs, and the sequence data is maintained in sync between the files. That is, if a sequence is filtered out in the first file, the corresponding sequence in the paired file is also filtered out.

`WriteSingleton` — Whether to save singleton sequences in a separate output file
`false` (default) | `true`

Whether to save singleton sequences in a separate output file, specified as `true` or `false`. To set this to `true`, the `'PairedFiles'` option must also be set to `true`.

A singleton sequence is the sequence that pass the filtering criterion but its corresponding sequence in the paired file does not. If `true`, singleton sequences are saved in a separate file with the suffix `'_singleton'`. The default is `false`, meaning that, only sequences that pass the filtering criterion in both input files of a given pair are saved in the output files.

Version History
Introduced in R2023a

See Also
`bioinfo.pipeline.blocks.SeqSplit` | `bioinfo.pipeline.blocks.SeqFilter` | `bioinfo.pipeline.blocks.SeqTrim` | `seqfilter`

SeqSplitOptions

Contain options to split sequences based on barcodes

Description

A `SeqSplitOptions` object contains options to split sequences into multiple files based on barcodes.This object is used as the value of `Options` property of the `bioinfo.pipeline.blocks.SeqSplit` block.

Creation

Syntax

```
optionsObj = bioinfo.pipeline.options.SeqSplitOptions
optionsObj = bioinfo.pipeline.options.SeqSplitOptions(Name=Value)
```

Description

`optionsObj = bioinfo.pipeline.options.SeqSplitOptions` creates a `SeqSplitOptions` object with default property values.

`optionsObj = bioinfo.pipeline.options.SeqSplitOptions(Name=Value)` sets properties on page 1-117 using one or more name-value arguments. `Name` is the property name and `Value` is the property value. For example, `optionsObj = bioinfo.pipeline.options.SeqSplitOptions(MaxMismatches=2)` specifies to allow up to two mismatches during barcode matching.

Properties

`MaxMismatches` — Maximum number of mismatches allowed during barcode matching
`0` (default) | nonnegative integer

Maximum number of mismatches allowed during barcode matching, specified as a nonnegative integer. The default is 0, that is, no mismatches are allowed.

`BarcodeFormat` — Type of barcode to match
`5` (default) | `3`

Type of barcode to match, specified as `3` or `5`. A value of `5` corresponds to the barcode located at the `5'` end of each sequence, and `3` corresponds to the `3'` end.

Example:

`RemoveBarcode` — Whether to remove the barcode
`true` (default) | `false`

Whether to remove the barcode and corresponding quality information from the matched sequences, specified as `true` or `false`. The default is `true`.

`WriteUnmatched` — Whether to save unmatched sequences
`false` (default) | `true`

Whether to save unmatched sequences and corresponding quality information in a separate output file, specified as `true` or `false`. The output file name has the suffix `'_unmatched'` instead of the barcode ID.

`OutputSuffix` — Suffix to use in output file name
`'_split'` (default) | character vector | string

Suffix to use in the output file name, specified as a character vector or string. It is inserted after the input file name and before the barcode ID. The default is `'_split'`.

Version History

Introduced in R2023a

See Also

`bioinfo.pipeline.blocks.SeqSplit` | `bioinfo.pipeline.blocks.SeqFilter` | `bioinfo.pipeline.blocks.SeqTrim` | `seqsplit`

SeqTrimOptions

Contain options to trim sequences based on specified criterion

Description

A `SeqTrimOptions` object contains options to trim sequences based on a specified criterion. This object is used as the value of `Options` property of the `bioinfo.pipeline.blocks.SeqTrim` block.

Creation

Syntax

```
optionsObj = bioinfo.pipeline.options.SeqTrimOptions
optionsObj = bioinfo.pipeline.options.SeqTrimOptions(Name=Value)
```

Description

`optionsObj = bioinfo.pipeline.options.SeqTrimOptions` creates a `SeqTrimOptions` object with default property values.

`optionsObj = bioinfo.pipeline.options.SeqTrimOptions(Name=Value)` sets properties on page 1-119 using one or more name-value arguments. `Name` is the property name and `Value` is the property value. For example, `ssOpt = bioinfo.pipeline.options.SeqTrimOptions(Threshold=[3 20])` specifies to trim each sequence when the number of bases with quality below 20 is greater than 3

Properties

`Encoding` — Base quality encoding format
`'Illumina18'` (default) | `'Sanger'` | `'Solexa'` | `'Illumina13'` | `'Illumina15'`

Base quality encoding format, specified as a character vector or string.

`Method` — Criterion to trim sequences
`'MaxNumberLowQualityBases'` (default) | `'MaxPercentLowQualityBases'` | `'MeanQuality'` | `'BasePositions'` | `'Termini'`

Criterion to trim sequences, specified as one of the following options. Specify only one trimming criterion per function call.

- `'MaxNumberLowQualityBases'`– applies a maximum threshold on the number of low-quality bases allowed before trimming a sequence starting at the `5'` end.
- `'MaxPercentLowQualityBases'`– applies a maximum threshold on the percentage of low-quality bases allowed before trimming a sequence starting at the `5'` end.
- `'MeanQuality'`– applies a minimum threshold on the running average base quality allowed before trimming a sequence starting at the `5'` end.

- `'BasePositions'`– trims each sequence according to the base positions (first base and last base) starting at the `5'` end.
- `'Termini'`– trims each sequence from either the `5'` or `3'` end or from both ends.

Use this name-value pair argument together with `'Threshold'` to specify the appropriate threshold value. Depending on the trimming criterion, the corresponding value for `'Threshold'` varies. See the `'Threshold'` option for the default values.

Note Sequences resulting in empty sequences after trimming are saved in the output files as empty sequences. To remove empty sequences from files, use the `seqfilter` function with the `'MinLength'` option set to the value of `1`.

`OutputSuffix` — Suffix to use in output file name

`'_trimmed'` (default) | character vector | string

Suffix to use in the output file name, specified as a character vector or string. It is inserted after the input file name and before the file extension. The default is `'_trimmed'`.

`Threshold` — Threshold value for trimming criterion

scalar | vector

Threshold value for the trimming criterion, specified as a scalar or vector. Use this name-value pair to define the threshold value for the trimming criterion specified by `'Method'`.

Depending on the trimming criterion, the corresponding value for `'Threshold'` can be a scalar or two-element vector. If you do not specify `'Threshold'`, then the function uses the default threshold value of the corresponding method. For each trimming criterion, the function uses the encoding format of the base quality specified by the `'Encoding'` name-value pair argument.

`'Method'`	`'Threshold'`	Default `'Threshold'` value
`'MaxNumberLowQualityBases'`	Two-element vector `[`*V1 V2*`]`. *V1* is a nonnegative integer that specifies the maximum number of low-quality bases allowed before trimming. *V2* specifies the minimum base quality. Any base with quality less than *V2* is considered a low-quality base.	`[0 10]`
`'MaxPercentLowQualityBases'`	Two-element vector `[`*V1 V2*`]`. *V1* is a scalar between 0 and 100 that specifies the maximum percentage of low quality bases allowed before trimming. *V2* specifies the minimum base quality. Any base with quality less than *V2* is considered a low-quality base.	`[0 10]`
`'MeanQuality'`	Positive scalar that specifies the minimum threshold on the running average base quality allowed before trimming a sequence starting at the `5'` end.	`0`

'Method'	'Threshold'	Default 'Threshold' value
'BasePositions'	Two-element vector [*V1 V2*], where *V1* and *V2* are positive integers specifying the base positions to start trimming at the 5' end and 3' end, respectively. To trim only the 5' end of each sequence before position *V1*, use [*V1* Inf]. To trim only the 3' end of each sequence after position *V2*, use [1 *V2*].	[1 Inf], that is, each sequence is left untrimmed.
'Termini'	Two-element vector [*V1 V2*], where V1 and V2 are nonnegative integers specifying the number of bases to trim at the 5' end and the 3' end, respectively. To trim *V1* bases at the 5' end only, use [*V1* 0]. To trim *V2* bases at the 3' end only, use [0 *V2*].	[0 0], that is, each sequence is left untrimmed.

WindowSize — Size of sliding window to apply filtering criterion to sequence
Inf (default) | positive integer

Size of the sliding window to apply the trimming criterion to a sequence, specified as a positive integer. The size of the window corresponds to the number of bases that the function uses at one time to apply the criterion. Any given sequence is trimmed before the first base of the window that violates the given criterion.

The sliding window can be applied to the following methods:

- 'MaxNumberLowQualityBases',
- 'MaxPercentLowQualityBases', and
- 'MeanQuality'.

Note Sequences shorter than the size of the window are saved in the output file as empty sequences. To remove empty sequences from files, use the seqfilter function with the 'MinLength' option set to the value of 1.

Version History

Introduced in R2023a

See Also

`bioinfo.pipeline.blocks.SeqSplit` | `bioinfo.pipeline.blocks.SeqFilter` | `bioinfo.pipeline.blocks.SeqTrim` | `seqtrim`

blockName

Package: `bioinfo.pipeline`

Return the names of specified blocks in pipeline

Syntax

```
names = blockName(pipeline,blocks)
```

Description

`names = blockName(pipeline,blocks)` returns the `names` of the `blocks` in the `pipeline`.

Examples

Get Block Names from Bioinformatics Pipeline

Import the pipeline and block objects needed for the example.

```
import bioinfo.pipeline.Pipeline
import bioinfo.pipeline.blocks.*
```

Create a pipeline.

```
P = Pipeline;
```

Create a FileChooser block.

```
fcBlock = FileChooser(which("ex1.sam"));
```

Create a SamSort block.

```
ssBlock = SamSort;
```

Add blocks to the pipeline.

```
addBlock(P,[fcBlock,ssBlock]);
```

Retrieve the block names.

```
bNames = blockName(P,[fcBlock,ssBlock])
```

```
bNames = 1×2 string
    "FileChooser_1"    "SamSort_1"
```

Input Arguments

pipeline — Bioinformatics pipeline
`bioinfo.pipeline.Pipeline` object

Bioinformatics pipeline, specified as a `bioinfo.pipeline.Pipeline` object.

`blocks` — Blocks in pipeline
`bioinfo.pipeline.Block` object | vector of objects

Blocks in the pipeline, specified as a `bioinfo.pipeline.Block` object or vector of such objects.

Output Arguments

`names` — Block names
string scalar | string array

Block names, returned as a string scalar or string array. If `blocks` is an array, `names(i)` is the name of the *i*th block `blocks(i)`.

Version History

Introduced in R2023a

See Also

`bioinfo.pipeline.Block` | `bioinfo.pipeline.Pipeline` | **Biopipeline Designer**

cancel

Package: `bioinfo.pipeline`

Cancel blocks in pipeline that are running in parallel

Syntax

`cancel(pipeline)`

Description

`cancel(pipeline)` stops all blocks in the pipeline that are currently running in a parallel environment. The function also prevents new blocks from being queued. The function has no effect on the blocks that have been completed prior to calling the function.

Examples

Cancel Bioinformatics Pipeline Blocks Running in Parallel

Import the pipeline and block objects needed for the example.

```
import bioinfo.pipeline.Pipeline
import bioinfo.pipeline.blocks.*
```

Create a pipeline.

```
P = Pipeline;
```

Add some pause blocks for illustration purposes. Each block pauses for 1 and 10 seconds, respectively.

```
PB1 = UserFunction(@() pause(1));
PB2 = UserFunction(@() pause(10));
addBlock(P,[PB1,PB2]);
```

Run the pipeline in parallel.

```
run(P,UseParallel=true);
```

Let the pipeline run for a few seconds in parallel. Then cancel the pipeline while it is still running.

```
cancel(P);
```

Check the process table that contains the run status of each block. Set `Expanded` to true to expand the process table variable values which are in cell arrays.

```
t = processTable(P,Expanded=true)
```

```
t=2×5 table
          Block           Status            RunStart                      RunEnd                  RunErro
    ________________     ______     ________________________     ________________________     ____________
    "UserFunction_1"    Error     07-Dec-2022 15:04:16    07-Dec-2022 15:04:16    {1×1 ParallelEx
    "UserFunction_2"    Error     07-Dec-2022 15:04:16    07-Dec-2022 15:04:16    {1×1 ParallelEx
```

You can extract more information from the process table. For example, check the run status of the first block.

```
PB1Info = t(1,:);
PB1Info.Status
```

```
ans = 
  RunStatus enumeration

    Error
```

Check any error message associated with the block.

```
PB1Info.RunErrors{:}
```

```
ans = 
  ParallelException with properties:

     identifier: 'parallel:fevalqueue:ExecutionCancelled'
        message: 'Execution of the future was cancelled.'
          cause: {}
    remotecause: {}
          stack: [0×1 struct]
     Correction: []
```

Input Arguments

pipeline — Bioinformatics pipeline
`bioinfo.pipeline.Pipeline` object

Bioinformatics pipeline, specified as a `bioinfo.pipeline.Pipeline` object.

Version History

Introduced in R2023a

See Also

`bioinfo.pipeline.Pipeline` | **Biopipeline Designer**

compile

Package: `bioinfo.pipeline`

Perform block-specific additional checks and validations

Syntax

`compile(blockObj)`

Description

`compile(blockObj)` performs additional checks and validations specific to a block `blockObj`. The `compile` method of a block is called when you run the parent pipeline.

You can only create and define a compile method if you have created as a subclass of `bioinfo.pipeline.Block`. Then you can implement your own block-specific checks, such as validating relationships between block properties within the `compile` method. You cannot edit or add a compile method for the built-in blocks, which includes the `UserFunction` block.

Input Arguments

`blockObj` — Block object
`bioinfo.pipeline.Block` object | ...

Block object, specified as a scalar `bioinfo.pipeline.Block` object.

Version History

Introduced in R2023a

See Also

`compile` | `bioinfo.pipeline.Block` | `bioinfo.pipeline.Pipeline`

compile

Package: `bioinfo.pipeline`

Verify pipeline structure and check for warnings and errors

Syntax

```
compile(pipeline)
compile(pipeline,inputStruct)
```

Description

`compile(pipeline)` checks if the `pipeline` is ready to run. The function also performs block-specific checks by running the `compile` methods of individual blocks in the pipeline and also checks that all required input ports are satisfied on page 1-130.

`compile(pipeline,inputStruct)` compiles the pipeline using a structure `inputStruct` as an input.

Examples

Compile Bioinformatics Pipeline

Checks for errors and warnings before running a pipeline.

Import the pipeline and block objects needed for the example.

```
import bioinfo.pipeline.Pipeline
import bioinfo.pipeline.blocks.*
```

Create a pipeline.

```
P = Pipeline;
```

Add some blocks to the pipeline.

```
FCB = FileChooser(which("ex1.sam"));
SSB = SamSort;
addBlock(P,[FCB, SSB]);
```

Compile the pipeline.

```
compile(P)
```

```
Error using bioinfo.pipeline.Pipeline/compile
Pipeline compilation failed with the following reason.

Caused by:
    Block 'SamSort_1' has one or more required ports that are not connected or passed in as the
    pipeline inputs.
```

The error is due to the blocks being not connected. Connect the blocks and recompile to ensure that there is no more error, and the pipeline is ready to run.

```
connect(P,FCB,SSB,["Files","SAMFile"]);
compile(P)
```

Run Bioinformatics Pipeline Using Input Structure

Import the Pipeline and block objects needed for the example.

```
import bioinfo.pipeline.Pipeline
import bioinfo.pipeline.blocks.*
```

Create a pipeline.

```
P = Pipeline;
```

Create a `Bowtie2Build` block to build index files for the reference genome.

```
bowtie2build = Bowtie2Build;
```

Create a `Bowtie2` block to map the read sequences to the reference sequence.

```
bowtie2 = Bowtie2;
```

Add the blocks to the pipeline.

```
addBlock(P,[bowtie2build,bowtie2],["bowtie2build","bowtie2"]);
```

Get the list of names of all the required input ports from every block in the pipeline that are needed to be set or connected. `IndexBaseName` is an input port of both `bowtie2build` and `bowtie2` block. `Reads1File` is the input port of the `bowtie2` block and `ReferenceFASTAFile` is the input of `bowtie2build` block.

```
portnames = inputNames(P)
```

```
portnames = 1×3 string
    "IndexBaseName"    "Reads1Files"    "ReferenceFASTAFiles"
```

Some blocks have optional input ports. To see the names of these ports, set `IncludeOptional=true`. For instance, the `Bowtie2` block has an optional input port (`Reads2Files`) that accept files for the second mate reads when you have paired-end read data.

```
allportnames = inputNames(P,IncludeOptional=true)
```

```
allportnames = 1×4 string
    "IndexBaseName"    "Reads1Files"    "Reads2Files"    "ReferenceFASTAFiles"
```

Create an input structure to set the input port values of the `bowtie2` and `bowtie2build` blocks. Specifically, set `IndexBaseName` to `"Dmel_chr4"` which is the base name for the reference index files for the Drosophila genome. Set `Reads1Files` to `"SRR6008575_10k_1.fq"` and `Reads2Files` to `"SRR6008575_10k_2.fq"`. Set `ReferenceFASTAFile` to `"Dmel_chr4.fa"`. These read files are already provided with the toolbox.

```
inputStruct.IndexBaseName = "Dmel_chr4";
inputStruct.Reads1Files   = "SRR6008575_10k_1.fq";
inputStruct.Reads2Files   = "SRR6008575_10k_2.fq";
inputStruct.ReferenceFASTAFiles = "Dmel_chr4.fa";
```

Optionally, you can compile and check if the input structure is set up correctly. Note that this compilation also happens automatically when you run the pipeline.

```
compile(P,inputStruct);
```

Run the pipeline using the structure as an input.

```
run(P,inputStruct);
```

Get the `bowtie2` block result after the pipeline finishes running.

```
wait(P);
mappedFile = results(P,bowtie2)
```

```
mappedFile = struct with fields:
    SAMFile: [1×1 bioinfo.pipeline.datatypes.File]
```

The `Bowtie2` block generates a SAM file that contains the mapped results. To see the location of the file, use `unwrap`.

```
unwrap(mappedFile.SAMFile)
```

Input Arguments

pipeline — Bioinformatics pipeline
`bioinfo.pipeline.Pipeline` object

Bioinformatics pipeline, specified as a `bioinfo.pipeline.Pipeline` object.

`inputStruct` — Input structure to satisfy input ports
structure

Input structure to satisfy on page 1-1648 unconnected input ports, specified as a structure.

The field names of `inputStruct` must match the names of unconnected ports in the pipeline.

Tip Use `inputNames` to get the list of names for all unconnected input ports and use them as field names in `inputStruct`.

Data Types: `struct`

More About

Satisfy Input Ports

All required input ports of every block in a pipeline must be satisfied before you can run the pipeline.

To satisfy an input port, you must do one of the following:

- Connect to another port.
- Set the value of the input port, that is, `myBlock.Inputs.PropertyName.Value`. For example, consider a `BamSort` block. To specify the name of a BAM file as the block input value, set the value as `bamsortBlock.Inputs.BAMFile.Value = "ex1.bam"`.
- Pass in an input structure by calling `run(`*`pipeline`*`,`*`inputStruct`*`)`, where *inputStruct* has the field name equivalent to the input port name and the field value as the input port value.

Version History

Introduced in R2023a

See Also

`compile` | `bioinfo.pipeline.Pipeline` | **Biopipeline Designer**

connect

Package: `bioinfo.pipeline`

Connect two blocks in pipeline

Syntax

```
connect(pipeline,sourceBlock,targetBlock,portsToConnect)
mappedPorts = connect(pipeline,sourceBlock,targetBlock,portsToConnect)
```

Description

`connect(pipeline,sourceBlock,targetBlock,portsToConnect)` connects an output port of the `sourceBlock` to an input port of the `targetBlock`. `portsToConnect` specifies the output and input ports.

`mappedPorts = connect(pipeline,sourceBlock,targetBlock,portsToConnect)` also returns the complete list of connections `mappedPorts` between the source block and target block after connecting two blocks as specified by `portsToConnect`.

Examples

Connect Blocks in Bioinformatics Pipeline

Import the Pipeline and block objects needed for the example.

```
import bioinfo.pipeline.Pipeline
import bioinfo.pipeline.blocks.*
```

Create a pipeline.

```
qcpipeline = Pipeline;
```

Select an input FASTQ file using a `FileChooser` block.

```
fastqfile = FileChooser(which("SRR005164_1_50.fastq"));
```

Create a `SeqFilter` block.

```
sequencefilter = SeqFilter;
```

Add blocks to the pipeline. Optionally, you can specify the block names.

```
addBlock(qcpipeline,[fastqfile,sequencefilter],["FF","SF"]);
```

Connect the output of the first block to the input of the second block. To do so, you need to first check the input and output port names of the corresponding blocks.

View the Outputs (port of the first block) and Inputs (port of the second block).

```
fastqfile.Outputs
```

```
ans = struct with fields:
    Files: [1×1 bioinfo.pipeline.Output]
```

```
sequencefilter.Inputs
```

```
ans = struct with fields:
    FASTQFiles: [1×1 bioinfo.pipeline.Input]
```

Connect the `Files` output port of the `fastqfile` block to the `FASTQFiles` port of `sequencefilter` block. The output is a 1-by-2 string array, indicating the names of two ports that are now connected. You can use either the block objects themselves or the block names that you have defined previously. The next command uses the block names to identify and connect these two blocks.

```
connectedports = connect(qcpipeline,"FF","SF",["Files","FASTQFiles"])
```

```
connectedports = 1×2 string
    "Files"    "FASTQFiles"
```

You can also query the names of connected ports using `portMap`.

```
portnames = portMap(qcpipeline,"FF","SF")
```

```
portnames = 1×2 string
    "Files"    "FASTQFiles"
```

Input Arguments

`pipeline` — Bioinformatics pipeline
`bioinfo.pipeline.Pipeline` object

Bioinformatics pipeline, specified as a `bioinfo.pipeline.Pipeline` object.

`sourceBlock` — Block in pipeline
`bioinfo.pipeline.Block` object | character vector | string scalar

Block in the pipeline to connect from, specified as a `bioinfo.pipeline.Block` object or character vector or string scalar that represents the block name.

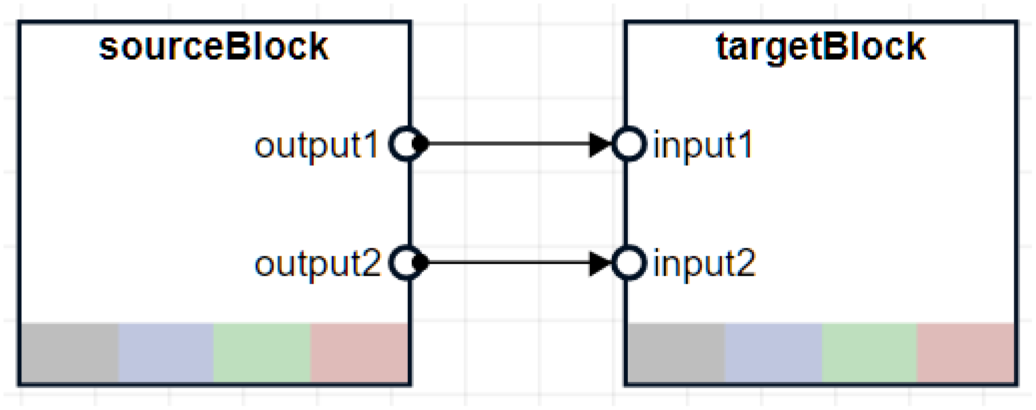
`targetBlock` — Block in pipeline
`bioinfo.pipeline.Block` object | character vector | string scalar

Block in the pipeline to connect to, specified as a `bioinfo.pipeline.Block` object or character vector or string scalar that represents the block name.

`portsToConnect` — Output port name and input port name
N-by-2 string array

Output port name of the source block and an input port name of the target block to connect, specified an *N*-by-2 string array, where *N* is the number of connections between two blocks.

For example, if you are connecting the source block that has two output ports, "`output1`" and "`output2`", to the target block that has two input ports, "`input1`" and "`input2`", specify `portsToConnect` as a 2-by-2 string array: `["output1","input1";"output2","input2"]`. The next figure is a graphical representation of such connections.



Tip To get the names of the input or output ports of a block, check the `Inputs` and `Outputs` properties of the block. Each property is returned as a structure, where each field name represents the corresponding port name that you can use to connect the blocks.

Data Types: `string`

Output Arguments

`mappedPorts` — Connections between source and target blocks
N-by-2 string array

Connections between source and target blocks, returned as an *N*-by-2 string array. *N* is the total number of connections between two blocks.

Version History

Introduced in R2023a

See Also

`bioinfo.pipeline.Block` | `bioinfo.pipeline.Pipeline` | `addBlock` | `portMap` | **Biopipeline Designer**

bioinfo.pipeline.datatypes.File

File object for bioinformatics pipeline

Description

A `bioinfo.pipeline.datatypes.File` object represents a file or an array of files that you can use in a bioinformatics pipeline. This object ensures that the file paths can be shared between blocks in a pipeline.

Tip For most workflows, use a `FileChooser` block to select input files for your pipeline. You might create this `File` object by itself when you are implementing your own block subclasses and want to share the files between blocks in your pipeline.

Creation

Syntax

`fileObj = bioinfo.pipeline.blocks.File(fileName)`

Description

`fileObj = bioinfo.pipeline.blocks.File(fileName)` creates a `File` object that represents the specified file `fileName`.

Input Arguments

`fileName` — File name
string | character vector | ...

File name, specified as a string, character vector, string array, or cell array of character vectors representing multiple file names.

Specify a file name or path and file name if the file is not in the current directory. This object does not use the MATLAB search path.

Data Types: `char` | `string` | `cell`

Object Functions

unwrap	Display full file path and name

Examples

Create File Object for Bioinformatics Pipeline

Create a `File` object.

```
file1 = which("SRR6008575_10k_1.fq");
fObj = bioinfo.pipeline.datatypes.File(file1)
```

```
fObj = 
  File with no properties.
```

Use `unwrap` to display the full file path.

```
unwrap(fObj)
```

Version History

Introduced in R2023a

See Also

`bioinfo.pipeline.Block` | `bioinfo.pipeline.datatypes.Unset` | `bioinfo.pipeline.datatypes.Incomplete`

unwrap

Package: bioinfo.pipeline.datatypes

Display full file path and name

Syntax

```
fullFileName = unwrap(fileObj)
```

Description

`fullFileName = unwrap(fileObj)` returns the full path and name of the file represented by the `bioinfo.pipeline.datatypes.File` object `fileObj`.

Examples

Create File Object for Bioinformatics Pipeline

Create a `File` object.

```
file1 = which("SRR6008575_10k_1.fq");
fObj = bioinfo.pipeline.datatypes.File(file1)
```

```
fObj = 
  File with no properties.
```

Use `unwrap` to display the full file path.

```
unwrap(fObj)
```

Input Arguments

`fileObj` — File object
`bioinfo.pipeline.datatypes.File` | array

File object, specified as a `bioinfo.pipeline.datatypes.File` object or array of such objects.

Output Arguments

`fullFileName` — Full file path and name
string | string array

Full file path and name, returned as a string or string array. If you specify an array of `File` objects as input, `fullFileName` will be a string array of the same size.

Version History
Introduced in R2023a

See Also
`bioinfo.pipeline.datatypes.File`

bioinfo.pipeline.datatypes.Incomplete

Incomplete pipeline result object

Description

A `bioinfo.pipeline.datatypes.Incomplete` object is the value returned by the `results` method when the block result is not yet computed.

When you query an incomplete result of a block at the command line, the value is displayed as follows.

```
results(pipeline,fileChooserBlock)
```

```
ans = 

  Incomplete pipeline result.
```

Creation

Use `bioinfo.pipeline.datatypes.Incomplete` to create the object.

Examples

Check Bioinformatics Pipeline Run Status and Results

Import the pipeline and block objects needed for the example.

```
import bioinfo.pipeline.Pipeline
import bioinfo.pipeline.blocks.*
```

Create a pipeline.

```
P = Pipeline;
```

Create a `FileChooser` block that takes in a SAM file as an input.

```
samFile = FileChooser(which("Myco_1_1.sam"));
```

Create a `SeqTrim` block.

```
trimsequences = SeqTrim;
```

Add blocks to the pipeline.

```
addBlock(P,[samFile,trimsequences]);
```

Check the names of the output port and input port of the blocks to connect.

```
samFile.Outputs
```

```
ans = struct with fields:
    Files: [1×1 bioinfo.pipeline.Output]
```

```
trimsequences.Inputs
```

```
ans = struct with fields:
    FASTQFiles: [1×1 bioinfo.pipeline.Input]
```

Connect the ports.

```
connect(P,samFile,trimsequences,["Files","FASTQFiles"]);
```

The block returns an incomplete result if the block has not yet run.

```
results(P,trimsequences)
```

```
ans = 
  Incomplete pipeline result.
```

Run the pipeline and check the run status of each block in the process table. The `samFile` block had no error, but the `SeqTrim` block generated an error as indicated by the `Status` column. The `SeqTrim` block generated an error because it was expecting a FASTQ file as an input but received a SAM file instead.

```
run(P);
t = processTable(P,Expanded=true)
```

```
t=2×5 table
         Block           Status            RunStart                 RunEnd                RunErrors
    _______________    _________    ____________________    ____________________    _____________

    "FileChooser_1"    Completed    22-Jan-2023 17:32:44    22-Jan-2023 17:32:44    {0×0 MExcept
    "SeqTrim_1"        Error        22-Jan-2023 17:32:45    22-Jan-2023 17:32:45    {1×1 MExcept
```

You can see the printed error message at the MATLAB command line. You can also enter the following commands to see the message.

```
seqTrimInfo = t(2,:);
seqTrimInfo.RunErrors{:}
```

Version History

Introduced in R2023a

See Also

`results` | `bioinfo.pipeline.datatypes.Unset` | `bioinfo.pipeline.datatypes.File` | `bioinfo.pipeline.Pipeline`

bioinfo.pipeline.datatypes.Unset

Unset input port value

Description

A `bioinfo.pipeline.datatypes.Unset` object is used as a placeholder value used by the blocks to indicate that an input port value is not set.

When you query the input port value of a block at the command line, the value is displayed as follows.

```
st.Inputs.FASTQFiles.Value
```

```
ans = 

  Unset input port value.
```

Creation

Use `bioinfo.pipeline.datatypes.Unset` to create the object.

Version History

Introduced in R2023a

See Also

`bioinfo.pipeline.Input` | `bioinfo.pipeline.datatypes.Incomplete` | `bioinfo.pipeline.datatypes.File` | `bioinfo.pipeline.Pipeline`

deleteResults

Package: `bioinfo.pipeline`

Delete block results from pipeline

Syntax

```
deleteResults(pipeline)
deleteResults(pipeline,blocks)
deleteResults(___,IncludeFiles=tf)
```

Description

`deleteResults(pipeline)` deletes all computed results and metadata of `pipeline`, a `bioinfo.pipeline.Pipeline` object. By default, the function does not delete the generated directories and output files. Set `IncludeFiles` name-value argument to true to delete the folders and files also.

`deleteResults(pipeline,blocks)` deletes the results of the specified `blocks` only.

`deleteResults(___,IncludeFiles=tf)`, for any syntax, specifies whether to also delete the generated directories and files of the blocks in the pipeline.

Examples

Delete Pipeline Results

Import the pipeline and block objects needed for the example.

```
import bioinfo.pipeline.Pipeline
import bioinfo.pipeline.blocks.*
```

Create a pipeline.

```
P = Pipeline;
```

Create `FileChooser` and `SamSort` blocks.

```
FCB = FileChooser(which("ex1.sam"));
SSB = SamSort;
```

Add blocks to the pipeline. You can pass in the names of the blocks ("myFileChooser" and "mySamSorter"). And connect them.

```
addBlock(P,[FCB, SSB],["myFileChooser","mySamSorter"]);
connect(P, FCB, SSB, ["Files","SAMFile"]);
```

Run the pipeline.

```
run(P,ResultsDirectory="C:\Examples\")
```

Check the results. During the pipeline run, the pipeline created a folder for each block with the same name as the block name. In this case, the pipeline created `myFileChooser` and `mySamSorter` folders in the above results directory `C:\Examples`. Each block results folder can contain 1 to N number of folders determined by the number of times a block processes in a given run. In this case, each block is processed just one time and the block results folder contains a subfolder named "1" that has the block results.

Set `Expanded` to true to expand the table variable values which are cell arrays.

```
tbl = processTable(P,Expanded=true)
```

```
tbl=2×5 table
         Block           Status            RunStart                  RunEnd                RunErrors
    _______________    _________    ____________________    ____________________    _____________

    "myFileChooser"    Completed    27-Jan-2023 16:28:27    27-Jan-2023 16:28:27    {0×0 MExcept
    "mySamSorter"      Completed    27-Jan-2023 16:28:27    27-Jan-2023 16:28:27    {0×0 MExcept
```

Check the result of the SamSort block. It indicates the output of the block is a sorted SAM file.

```
r = results(P,SSB)
```

```
r = struct with fields:
    SortedSAMFile: [1×1 bioinfo.pipeline.datatypes.File]
```

Check the full file path of the sorted file.

```
unwrap(r.SortedSAMFile)
```

```
ans =
"C:\Examples\mySamSorter\1\ex1.sorted.sam"
```

Delete the SamSort block result from the pipeline. After deletion, the SAM file output value is now incomplete.

```
deleteResults(P,SSB)
r = results(P,SSB)
```

```
r =
  Incomplete pipeline result.
```

However, the generated directory (*mySamSorter*) and actual SAM file still exists in that directory. To delete them also from your computer completely, use `IncludeFiles` name-value argument.

```
deleteResults(P,SSB,IncludeFiles=true)
```

Input Arguments

pipeline — Bioinformatics pipeline
`bioinfo.pipeline.Pipeline` object

Bioinformatics pipeline, specified as a `bioinfo.pipeline.Pipeline` object.

`blocks` — Input blocks

`bioinfo.pipeline.Block` object | vector of objects | character vector | string scalar | ...

Input blocks, specified as a `bioinfo.pipeline.Block` object, vector of block objects, character vector, string scalar, string vector, or cell array of character vectors representing block names.

`tf` — Flag to delete generated directories and files

`false` or 0 (default) | `true` or 1

Flag to delete generated block results folders and contents inside those folders, specified as a numeric or logical 1 (`true`) or 0 (`false`).

Data Types: `logical`

Version History

Introduced in R2023a

See Also

`results` | `fetchResults` | `bioinfo.pipeline.Block` | `bioinfo.pipeline.Pipeline` | **Biopipeline Designer**

disconnect

Package: `bioinfo.pipeline`

Remove connection between ports in a pipeline

Syntax

```
disconnect(pipeline,sourceBlock,targetBlock)
portMaps = disconnect(pipeline,sourceBlock,targetBlock,portsToDisconnect)
```

Description

`disconnect(pipeline,sourceBlock,targetBlock)` removes all connections between `sourceBlock` and `targetBlock` in the `pipeline`.

`portMaps = disconnect(pipeline,sourceBlock,targetBlock,portsToDisconnect)` removes the specified connections, `portsToDisconnect`, between `sourceBlock` and `targetBlock`. `portMaps` is the list of remaining connections between the blocks after disconnecting the specified ports.

Examples

Disconnect Blocks in Bioinformatics Pipeline

Import the pipeline and block objects needed for the example.

```
import bioinfo.pipeline.Pipeline
import bioinfo.pipeline.blocks.*
```

Create a pipeline.

```
P = Pipeline;
```

Create two `FileChooser` blocks and a `Bowtie2` block. The files represent pair-end read data for the Drosophila genome. These read files are already provided with the toolbox.

```
reads1 = FileChooser(which("SRR6008575_10k_1.fq"));
reads2 = FileChooser(which("SRR6008575_10k_2.fq"));
bowtie2Block = Bowtie2;
```

Add the blocks. Define the block names as "reads1","reads2", and "bowtie2" and add them in the same call.

```
addBlock(P,[reads1,reads2,bowtie2Block],["reads1","reads2","bowtie2"]);
```

Check the names of the output and input ports of the blocks to connect.

```
reads1.Outputs
```

```
ans = struct with fields:
    Files: [1×1 bioinfo.pipeline.Output]
```

```
reads2.Outputs
```

```
ans = struct with fields:
    Files: [1×1 bioinfo.pipeline.Output]
```

```
bowtie2Block.Inputs
```

```
ans = struct with fields:
    IndexBaseName: [1×1 bioinfo.pipeline.Input]
      Reads1Files: [1×1 bioinfo.pipeline.Input]
      Reads2Files: [1×1 bioinfo.pipeline.Input]
```

The `Bowtie2` block has two required and one optional inputs. Set the value of the first required input `IndexBaseName` to `"Dmel_chr4"`, which is the base name for the reference index files, which are provided with the toolbox, for the Drosophila genome.

```
bowtie2Block.Inputs.IndexBaseName.Value = "Dmel_chr4";
```

Because you are providing the pair-end (paired) read data, connect `Reads1Files` to the `Files` output of the `reads1` block, which contains the first mate reads "SRR6008575_10k_1.fq" and connect `Reads2Files` to the `reads2` block that contains the second mate reads "SRR6008575_10k_2.fq". The `Reads2Files` input is optional and is not needed if you have the single-end (unpaired) read data.

```
connect(P,reads1,bowtie2Block,["Files","Reads1Files"]);
connect(P,reads2,bowtie2Block,["Files","Reads2Files"]);
```

Run the pipeline.

```
run(P);
```

The Bowtie2 block generates a SAM file as the output.

```
mappedReads = results(P,bowtie2Block)
```

```
mappedReads = struct with fields:
    SAMFile: [1×1 bioinfo.pipeline.datatypes.File]
```

Suppose you want to provide a single end read data to the `bowtie2Block`. You can update the `reads1` block to read in such a single end read file and disconnect the `Reads2File` optional input of bowtie2Block. The `disconnect` funciton returns an empty string array, which means that there are no more connection between two blocks.

```
reads1.Files = which("SRR005164_1_50.fastq");
disconnect(P,reads2,bowtie2Block,["Files","Reads2Files"])
```

```
ans = 

  0×2 empty string array
```

Alternatively, you can use portMap to check if there are any connections between two blocks.

```
portMap(P,reads2,bowtie2Block)
```

```
ans = 

  0×2 empty string array
```

You can then run the pipeline again.

```
run(P);
mappedReads = results(P,bowtie2Block)

mappedReads = struct with fields:
    SAMFile: [1×1 bioinfo.pipeline.datatypes.File]
```

Input Arguments

`pipeline` — Bioinformatics pipeline
`bioinfo.pipeline.Pipeline` object

Bioinformatics pipeline, specified as a `bioinfo.pipeline.Pipeline` object.

`sourceBlock` — Source block
`bioinfo.pipeline.Block` object | character vector | string scalar

Source block, specified as a `bioinfo.pipeline.Block` object or character vector or string scalar representing a block name.

`targetBlock` — Target block
`bioinfo.pipeline.Block` object | character vector | string scalar

Target block, specified as a `bioinfo.pipeline.Block` object or character vector or string scalar representing a block name.

Output Arguments

`portsToDisconnect` — Remaining connections between source and target blocks
N-by-2 string array

Remaining connections between the source and target blocks, returned as an *N*-by-2 string array, where *N* is the total number of connections. The string array lists the names of output ports (of the source blocks) and input ports (of the target blocks) that are still connected.

Version History

Introduced in R2023a

See Also

`connect` | `portMap` | `bioinfo.pipeline.Block` | `bioinfo.pipeline.Pipeline` | **Biopipeline Designer**

emptyInputs

Package: `bioinfo.pipeline`

Create input structure for use with `run` method

Syntax

```
S = emptyInputs(blockObj)
S = emptyInputs(blockObj,IncludeOptional=tf)
```

Description

`S = emptyInputs(blockObj)` returns a scalar structure that you can then modify and use as an input for the block `run` method.

`S = emptyInputs(blockObj,IncludeOptional=tf)` specifies to also include the optional input ports as fields of `S`.

Examples

Create Input Structure from Bioinformatics Pipeline Block

Create a `Bowtie2` block to map the read sequences to the reference sequence.

```
bowtie2block = bioinfo.pipeline.blocks.Bowtie2;
```

Use the `emptyInputs` method of the block to construct an input structure with field names prepopulated to match the names of required input port names of the block.

```
inStruct = emptyInputs(bowtie2block)
```

```
inStruct = struct with fields:
    IndexBaseName: []
      Reads1Files: []
```

Some of the built-in blocks have optional input ports. To see the names of these ports, set `IncludeOptional=true`. For instance, the `Bowtie2` block has an optional input port (`Reads2Files`) that accepts files for the second mate reads when you have paired-end read data.

```
inStructAll = emptyInputs(bowtie2block,IncludeOptional=true)
```

```
inStructAll = struct with fields:
    IndexBaseName: []
      Reads1Files: []
      Reads2Files: []
```

Set the field values of the prepopulated structure. Specifically, set `IndexBaseName` to `"Dmel_chr4"`, which is the base name for the reference index files for the Drosophila genome. Set `Reads1Files` to

"SRR6008575_10k_1.fq" and Reads2Files to "SRR6008575_10k_2.fq". These files are already provided with the toolbox.

```
inStructAll.IndexBaseName = "Dmel_chr4";
inStructAll.Reads1Files   = "SRR6008575_10k_1.fq";
inStructAll.Reads2Files   = "SRR6008575_10k_2.fq";
```

Run the block using the structure as an input. The mapped results are saved as Aligned.sam in the current working directory.

```
run(bowtie2block,inStructAll);
```

Input Arguments

blockObj — Block object
bioinfo.pipeline.Block object | ...

Block object, specified as a scalar bioinfo.pipeline.Block object.

tf — Flag to include optional input port names
false or 0 (default) | true or 1

Flag to include optional input port names, specified as a numeric or logical 1 (true) or 0 (false). An optional inport port is an input port (bioinfo.pipeline.Input) with its Required property set to false.

Output Arguments

S — Prepared input structure for run method
structure

Prepared input structure for the block run method, returned as a structure. The field names of the structure are the names of the required input ports of the block blockObj. The value of each field is set as an empty array [].

If you set IncludeOptional=true, the structure also contains the fields for optional input ports of the block.

Version History

Introduced in R2023a

See Also

compile | bioinfo.pipeline.Block | bioinfo.pipeline.Pipeline

eval

Package: `bioinfo.pipeline`

Evaluate block object

Syntax

```
outStruct = eval(blockObj,inStruct)
```

Description

`outStruct = eval(blockObj,inStruct)` evaluates a block object `blockObj` using the input structure `inStruct` and returns the block outputs in the output structure `outStruct`.

Note

- It is recommended that you use the `run` method of a block instead of `eval` because the `run` method performs additional error checks and ensures that the block inputs and outputs are satisfied on page 1-130 and that the `eval` method accepts and returns a scalar structure, which is a requirement to run the block as part of a pipeline.
- For all block objects, the `run` method of a block calls the `eval` method. `run` is a sealed method that performs some validation on the inputs and outputs for `eval`. The `eval` method is a customized evaluation method of a block with additional validation on the inputs and outputs and is sealed. To implement a custom block behavior, create a block subclass and write your own `eval` function as described in this example on page 1-150.
- Do not call `eval` directly unless you are prototyping your own block subclass and implementing your block tasks in the `eval` method.
- When you implement your custom `eval` method for a block, ensure that the method does not modify any properties of the block so that the pipeline can skip execution of the block when the inputs are the same as a previous run.

Examples

Subclass Pipeline Block

For most pipelines, using a combination of built-in blocks and `UserFunction` blocks is sufficient and recommended. However, if a block requires more complexity, such as additional configuration options or methods, you can create a custom block by subclassing the `bioinfo.pipeline.Block` object to implement additional capabilities.

Subclasses of the `Block` object must:

- Define their `Inputs` and `Outputs` properties.
- Define the `eval` method.

For instance, the following lines of code defines a custom block object named `Seqlinkage` which is defined as a subclass of `bioinfo.pipeline.Block.` For illustration purposes, the `Seqlinkage` block uses the `seqlinkage` function, which constructs a phylogenetic tree from pairwise distances as the underlying function. The function accepts a matrix or vector of pairwise distances and a distance method to use.

```
classdef Seqlinkage < bioinfo.pipeline.Block
      % Define the block properties
      properties
          % Define the Method property that accepts two distance methods.
          % For details on these methods, enter the following command at
          % the command line: doc seqlinkage
          Method {mustBeMember(Method, ["average", "weighted"])} = "average";
      end

      methods
          % Define inputs and outputs.
          function block = Seqlinkage()
              import bioinfo.pipeline.Input
              import bioinfo.pipeline.Output

              % Define the Inputs and Outputs property of the object.

              % Name the first input port asd Distances and as a required
              % input that takes in a matrix or vector of pairwise distances.
              block.Inputs.Distances = Input('Required',true);

              % Name the second input port as Names and as an optional
              % input that takes in a list of names for nodes.
              block.Inputs.Names = Input('Required',false);

              % Name the output port Phytree.
              block.Outputs.Phytree = Output;
          end

          % Define custom evaluation method for the block.
          function outStruct = eval(obj, inStruct)
              % If the optional input (a list of node names) is passed in
              if isfield(inStruct,'Names')
                  % Call the seqlinkage function with three inputs and save
                  % the returned phytree object as output.
                  outStruct.Phytree = seqlinkage(inStruct.Distances,obj.Method,inStruct.Names);
              else
                  % Call seqlinkage with two inputs.
                  outStruct.Phytree = seqlinkage(inStruct.Distances,obj.Method);
              end
          end
      end
  end
```

You can save such a class definition to a separate MATLAB® program file. For this example, the above `Seqlinkage` class definition has already been saved and provided as `Seqlinkage.m`. Note that the definition file must be in the current working folder or MATLAB search path before you can use the block object in your pipeline.

Next, you can define a pipeline that uses the `Seqlinkage` block as follows to build a phylogenetic tree from pairwise distances.

```
import bioinfo.pipeline.Pipeline
import bioinfo.pipeline.blocks.*
P = Pipeline;
```

Create three blocks needed for the workflow.

```
fastaReadBlock  = UserFunction("fastaread",RequiredArguments="inputFile",OutputArguments="sequen
seqpdistBlock   = UserFunction("seqpdist",RequiredArguments="inputSequences",OutputArguments="pa
seqlinkageBlock = Seqlinkage;
```

Set the input port of `fastaReadBlock` to a FASTA file containing amino acid sequences.

```
fastaReadBlock.Inputs.inputFile.Value = which("pf00002.fa");
```

Add the blocks to the pipeline.

```
addBlock(P,[fastaReadBlock,seqpdistBlock,seqlinkageBlock],["fastaread","seqpdist","seqlinkage"])
```

Connect the first two blocks.

```
connect(P,"fastaread","seqpdist",["sequences","inputSequences"]);
```

Connect `seqpdistBlock` to `seqlinkageBlock`. Specifically, connect the output port "`pairwiseDistances`" of `seqpdistBlock` to one of the input ports of the `seqlinkageBlock` "`Distances`", as defined in the `Seqlinkage.m`. Note that `Distances` is a required input port that must have its value set. One of the valid ways is to connect to another port. For other ways to set input ports, see “Satisfy Input Ports” on page 1-1648.

```
connect(P,"seqpdist","seqlinkage",["pairwiseDistances","Distances"]);
```

Connect the output port "sequences" of the `fastaread` block to the "`Names`" optional input port of `seqlinkageBlock` to label the leave nodes of the output phylogenetic tree.

```
connect(P,"fastaread","seqlinkage",["sequences","Names"]);
```

Optionally, you can also pass in a list of node names of as the value of the optional input port "`Names`" by setting the property `seqlinkageBlock.Inputs.Names.Value`.

Before you run the pipeline, ensure that the folder of the class definition file is on the MATLAB search path. The reason is because each pipeline block runs in its own folder, and the external class definition files or function files will not be detected by the pipeline unless they are on the path.

```
% Update the following addpath call to specify the path to your class definition file. For insta
addpath("C:\Examples\SubclassExample\");
```

Run the pipeline.

```
run(P);
```

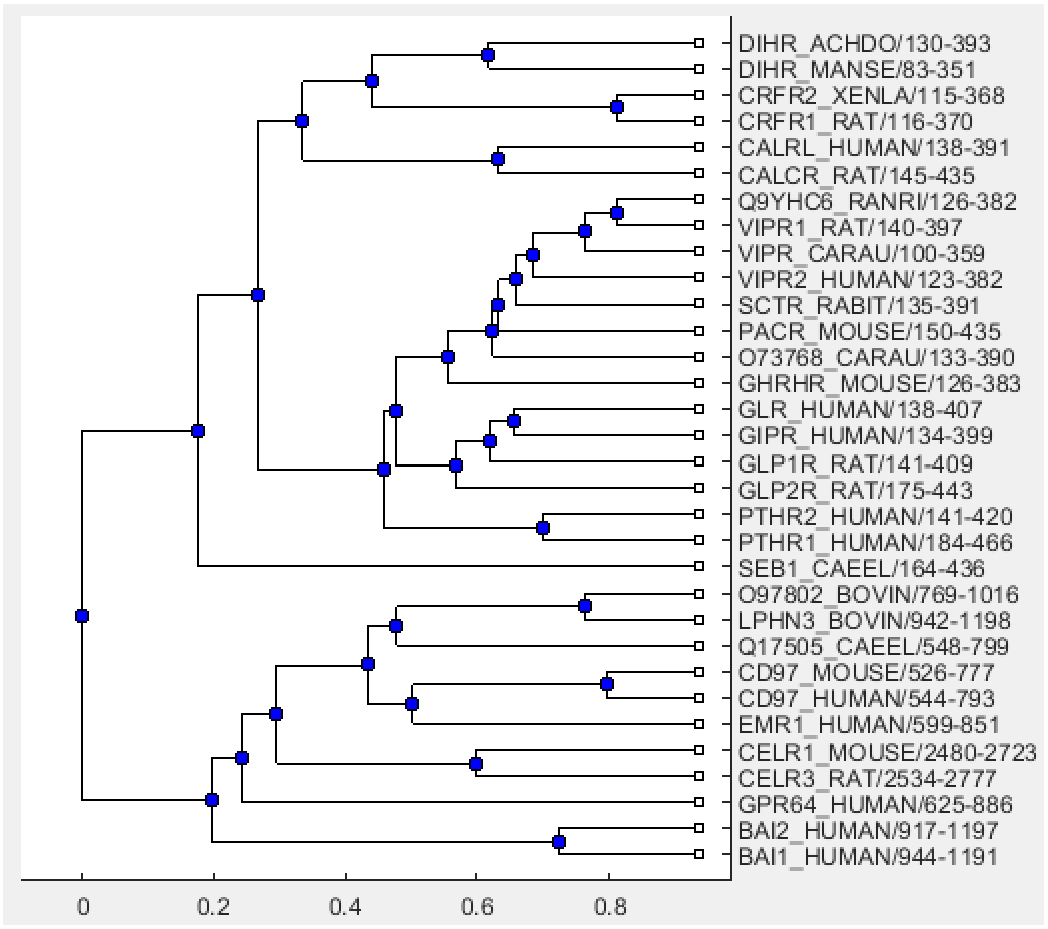
Get the result from `seqlinkageBlock`, which contains a phlogenetic tree object.

```
linkageResult = fetchResults(P,seqlinkageBlock)
```

```
linkageResult = struct with fields:
    Phytree: [1×1 phytree]
```

Use the following command to visualize the phylogenetic tree.

```
view(linkageResult.Phytree)
```



Input Arguments

`blockObj` — Block object
`bioinfo.pipeline.Block` object | ...

Block object, specified as a scalar `bioinfo.pipeline.Block` object.

`inStruct` — Block input
structure

Block input, specified as a structure. The field names of the structure must be the names of the input ports of the block.

Output Arguments

`outStruct` — Block output
structure

Block output, returned as a structure. The field names of the structure are the names of the output ports of the block, and the field values are the output values of the block.

Version History

Introduced in R2023a

See Also

`compile` | `bioinfo.pipeline.Block` | `bioinfo.pipeline.Pipeline`

getStop

Class: `BioMap`

Compute stop positions of aligned read sequences from `BioMap` object

Syntax

```
Stop = getStop(BioObj)
Stop = getStop(BioObj, Subset)
```

Description

`*Stop* = getStop(*BioObj*)` returns *`Stop`*, a vector of integers specifying the stop position of aligned read sequences with respect to the position numbers in the reference sequence from a `BioMap` object.

`*Stop* = getStop(*BioObj*, *Subset*)` returns a stop position for only read sequences specified by *`Subset`*.

Input Arguments

`BioObj`

Object of the `BioMap` class.

Default:

`Subset`

One of the following to specify a subset of the elements in *`BioObj`*:

- Vector of positive integers
- Logical vector
- Cell array of character vectors containing valid sequence headers

Note If you use a cell array of headers to specify *`Subset`*, be aware that a repeated header specifies all elements with that header.

Default:

Output Arguments

`Stop`

Vector of integers specifying the stop position of aligned read sequences with respect to the position numbers in the reference sequence. *`Stop`* includes the stop positions for only read sequences specified by *`Subset`*.

Examples

Construct a `BioMap` object, and then compute the stop position for different sequences in the object:

```
% Construct a BioMap object from a SAM file
BMObj1 = BioMap('ex1.sam');
% Compute the stop position of the second sequence in the object
Stop_2 = getStop(BMObj1, 2)

Stop_2 =

          37

% Compute the stop positions of the first and third sequences in
% the object
Stop_1_3 = getStop(BMObj1, [1 3])

Stop_1_3 =

          36
          39

% Compute the stop positions of all sequences in the object
Stop_All = getStop(BMObj1);
```

See Also

`BioMap` | `getStart`

Topics

"Manage Sequence Read Data in Objects"

External Websites

Sequence Read Archive
SAM format specification

fetchResults

Package: `bioinfo.pipeline`

Wait for parallel-running block to finish and return its results

Syntax

```
blockResults = fetchResults(pipeline,block)
```

Description

`blockResults = fetchResults(pipeline,block)` returns the results of `block` in the `pipeline`. This function is equivalent to the `results` function with the exception that if you are running the pipeline in parallel, `fetchResults` waits for the block to complete before returning its results.

Input Arguments

`pipeline` — Bioinformatics pipeline
`bioinfo.pipeline.Pipeline` object

Bioinformatics pipeline, specified as a `bioinfo.pipeline.Pipeline` object.

`block` — Block in pipeline
`bioinfo.pipeline.Block` object | character vector | string scalar

Block in the pipeline, specified as a scalar `bioinfo.pipeline.Block` object, character vector, or string scalar as the block name. To get the list of block names, enter `pipeline.“BlockNames” on page 1-0` at the command line.

Output Arguments

`blockResults` — Block results
structure | `bioinfo.pipeline.datatypes.Incomplete`

Block results, returned as a structure or `bioinfo.pipeline.datatypes.Incomplete` object. The `Incomplete` object is returned if the block results are not yet available because the block has not finished running or will not run due to errors.

If it is a structure, the field names are the output port names of the block, and the field values are the output values.

Data Types: `struct`

Version History

Introduced in R2023a

See Also

`deleteResults` | `results` | `bioinfo.pipeline.Block` | `bioinfo.pipeline.Pipeline` | **Biopipeline Designer**

findBlock

Package: `bioinfo.pipeline`

Get block objects from bioinformatics pipeline

Syntax

```
blockObjects = findBlock(pipeline)
blockObjects = findBlock(pipeline,blockIdentifiers)
```

Description

`blockObjects = findBlock(pipeline)` returns all blocks in the pipeline as the array of block objects `blockObjs`.

`blockObjects = findBlock(pipeline,blockIdentifiers)` returns only the blocks specified by `blockIdentifiers`.

Examples

Find Blocks from Bioinformatics Pipeline

Import the pipeline and block objects needed for the example.

```
import bioinfo.pipeline.Pipeline
import bioinfo.pipeline.blocks.*
```

Create a pipeline.

```
P = Pipeline;
```

Create the FileChooser and SamSort blocks.

```
FCB = FileChooser(which("ex1.sam"));
SSB = SamSort;
```

Add blocks to the pipeline and connect them.

```
addBlock(P, [FCB, SSB],["fcb","ssb"]);
connect(P, FCB, SSB, ["Files", "SAMFile"]);
```

Get all the block objects in the pipeline.

```
allBlocks = findBlock(P)
```

```
allBlocks=2×1 object
  2×1 heterogeneous Block (FileChooser, SamSort) array with properties:

    Inputs
    Outputs
    ErrorHandler
```

Find blocks by their names.

```
b1 = findBlock(P,"ssb")
```

```
b1 = 
  SamSort with properties:

     OutFilename: [0×0 string]
          Inputs: [1×1 struct]
         Outputs: [1×1 struct]
    ErrorHandler: []
```

```
b1_2 = findBlock(P,["fcb","ssb"])
```

```
b1_2=2×1 object
  2×1 heterogeneous Block (FileChooser, SamSort) array with properties:

    Inputs
    Outputs
    ErrorHandler
```

Input Arguments

`pipeline` — Bioinformatics pipeline
`bioinfo.pipeline.Pipeline` object

Bioinformatics pipeline, specified as a `bioinfo.pipeline.Pipeline` object.

`blockIdentifiers` — Block identifiers
`bioinfo.pipeline.Block` object | vector of block objects | character vector | string scalar | ...

Block identifiers, specified as a scalar `bioinfo.pipeline.Block` object or vector of block objects. You can also specify a character vector, string scalar, string vector, or cell array of character vectors, representing block names.

Output Arguments

`blockObjects` — Block objects
`bioinfo.pipeline.Block` object | vector of block objects

Block objects, returned as a `bioinfo.pipeline.Block` object or a vector of block objects.

Version History

Introduced in R2023a

See Also

`blockName` | `bioinfo.pipeline.Block` | `bioinfo.pipeline.Pipeline` | **Biopipeline Designer**

filterByFlag

Class: `BioMap`

Filter sequence reads by SAM flag

Syntax

```
Indices = filterByFlag(BioObj, FlagName, FlagValue)
Indices = filterByFlag(BioObj, Subset, FlagName, FlagValue)
Indices = filterByFlag(..., FlagName1, FlagValue1, FlagName2,
FlagValue2, ...)
```

Description

`Indices = filterByFlag(BioObj, FlagName, FlagValue)` returns *`Indices`*, a vector of logical indices, indicating the read sequences in *`BioObj`*, a `BioMap` object, with *`FlagName`* set to *`FlagValue`*.

`Indices = filterByFlag(BioObj, Subset, FlagName, FlagValue)` returns *`Indices`*, a vector of logical indices, indicating the read sequences that meet the specified criteria from a subset of entries in a `BioMap` object.

`Indices = filterByFlag(..., FlagName1, FlagValue1, FlagName2, FlagValue2, ...)` applies multiple flag filters in a single statement.

Input Arguments

`BioObj`

Object of the `BioMap` class.

Default:

`Subset`

Either of the following to specify a subset of the elements in *`BioObj`*:

- Vector of positive integers
- Logical vector

Default:

`FlagName`

Character vector or string specifying one of the following flags to filter by:

- `'pairedInSeq'` — The read is paired in sequencing, regardless if it is mapped as a pair.
- `'pairedInMap'` — The read is mapped in a proper pair.
- `'unmappedQuery'` — The read is unmapped.

- `'unmappedMate'` — The mate is unmapped.
- `'strandQuery'` — Strand direction of the read (`0` = forward, `1` = reverse).
- `'strandMate'` — Strand direction of the mate (`0` = forward, `1` = reverse).
- `'readIsFirst'` — The read is first in a pair.
- `'readIsSecond'` — The read is second in a pair.
- `'alnNotPrimary'` — The read's alignment is not primary.
- `'failedQualCheck'` — The read fails platform or vendor quality checks.
- `'duplicate'` — The read is a PCR or optical duplicate.

Default:

`FlagValue`

Logical value indicating the status of a flag. A `0` indicates `false` or forward, and a `1` indicates `true` or reverse.

Default:

Output Arguments

`Indices`

Vector of logical indices, indicating the read sequences in *`BioObj`* with *`FlagName`* set to *`FlagValue`*.

Examples

Construct a `BioMap` object, and then determine the read sequences that are both mapped in a proper pair and first in a pair:

```
% Construct a BioMap object from a SAM file
BMObj1 = BioMap('ex1.sam');
% Filter the elements using 'pairedInMap' and 'readIsFirst' flags
Indices = filterByFlag(BMObj1, 'pairedInMap', true,...
                       'readIsFirst', true);
% Return the headers of the filtered elements
filtered_Headers = BMObj1.Header(Indices);
```

See Also
`BioMap`

Topics
“Manage Sequence Read Data in Objects”

External Websites
Sequence Read Archive
SAM format specification

getAlignment

Class: `BioMap`

Construct alignment represented in `BioMap` object

Syntax

```
Alignment = getAlignment(BioObj, StartPos, EndPos)
Alignment = getAlignment(BioObj, StartPos, EndPos, R)
Alignment = getAlignment(..., 'ParameterName', ParameterValue)
[Alignment, Indices] = getAlignment(...)
```

Description

`*Alignment* = getAlignment(*BioObj*, *StartPos*, *EndPos*)` returns *`Alignment`*, a character array containing the aligned read sequences from *`BioObj`*, a `BioMap` object. The read sequences must align within a specific region of the reference sequence, which is defined by *`StartPos`* and *`EndPos`*, two positive integers such that *`StartPos`* is less than *`EndPos`*, and both are smaller than the length of the reference sequence.

`*Alignment* = getAlignment(*BioObj*, *StartPos*, *EndPos*, *R*)` selects the reference where `getAlignment` reconstructs the alignment.

`*Alignment* = getAlignment(..., '*ParameterName*', *ParameterValue*)` accepts one or more comma-separated parameter name/value pairs. Specify *`ParameterName`* inside single quotes.

`[*Alignment*, *Indices*] = getAlignment(...)` returns *`Indices`*, a vector of indices specifying the read sequences that align within a specific region of the reference sequence.

Input Arguments

`BioObj`

Object of the `BioMap` class.

Default:

`StartPos`

Positive integer that defines the start of a region of the reference sequence. *`StartPos`* must be less than *`EndPos`*, and smaller than the total length of the reference sequence.

Default:

`EndPos`

Positive integer that defines the end of a region of the reference sequence. *`EndPos`* must be greater than *`StartPos`*, and smaller than the total length of the reference sequence.

Default:

R

Positive integer indexing the `SequenceDictionary` property of *`BioObj`*, or a character vector or string specifying the actual name of the reference.

Parameter Name/Value Pairs

`OffsetPad`

Specifies if padding blanks are added at the beginning of each aligned sequence to represent the offset of the start position of each aligned sequence with respect to the reference. Choices are `true` or `false` (default).

Default:

Output Arguments

`Alignment`

Character array containing the aligned read sequences from *`BioObj`* that align within a specific region of the reference sequence. Each row of the character array contains one aligned sequence, that is, the sequence positions that fall within the specified region of the reference sequence. Each aligned sequence can include gaps.

`Indices`

Vector of indices specifying the read sequences from *`BioObj`* that align within a specific region of the reference sequence.

Examples

Construct a `BioMap` object, and then reconstruct the alignment between positions 10 and 25 of the reference sequence:

```
% Construct a BioMap object from a SAM file
BMObj1 = BioMap('ex1.sam');
% Construct the alignment between positions 10 and 25 of the
% reference sequence.
Alignment = getAlignment(BMObj1, 10, 25)

Alignment =

CTCATTGTAAATGTGT
CTCATTGTAAATGTGT
CTCATTGTAAATGTGT
CTCATTGTAATTTTTT
CTCATTGTAAATGTGT
   ATTGTAAATGTGT
   ATTGTAAATGTGT
     TGTAAATGTGT
        AAATGTGT
            GTGT
            GTGT
              GT
```

Algorithms

`getAlignment` assumes the reference sequence has no gaps. Therefore, positions in reads corresponding to insertions (I) and padding (P) do not appear in the alignment.

Because soft clipped positions (S) are not associated with positions that align to the reference sequence, they do not appear in the alignment.

A skipped position (N) appears as a `.` (period) in the alignment.

Hard clipped positions (H) do not appear in the sequences or the alignment.

See Also

`BioMap` | `getBaseCoverage` | `getCompactAlignment` | `align2cigar` | `cigar2align`

Topics
“Manage Sequence Read Data in Objects”

External Websites
Sequence Read Archive
SAM format specification

getBaseCoverage

Class: `BioMap`

Return base-by-base alignment coverage of reference sequence in `BioMap` object

Syntax

```
Cov = getBaseCoverage(BioObj, StartPos, EndPos)
Cov = getBaseCoverage(BioObj, StartPos, EndPos, R)
Cov = getBaseCoverage(..., Name,Value)
[Cov, BinStart] = getBaseCoverage(...)
```

Description

`Cov = getBaseCoverage(BioObj, StartPos, EndPos)` returns *`Cov`*, a row vector of nonnegative integers. This vector indicates the base-by-base alignment coverage of a range or set of ranges in the reference sequence in *`BioObj`*, a `BioMap` object. The range or set of ranges are defined by *`StartPos`* and *`EndPos`*. *`StartPos`* and *`EndPos`* can be two nonnegative integers such that *`StartPos`* is less than *`EndPos`*, and both integers are smaller than the length of the reference sequence. *`StartPos`* and *`EndPos`* can also be two column vectors representing a set of ranges (overlapping or segmented). When *`StartPos`* and *`EndPos`* specify a segmented range, *`Cov`* contains `NaN` values for base positions between segments.

`Cov = getBaseCoverage(BioObj, StartPos, EndPos, R)` selects the reference where `getBaseCoverage` calculates the coverage.

`Cov = getBaseCoverage(..., Name,Value)` returns alignment coverage information with additional options specified by one or more `Name,Value` pair arguments.

`[Cov, BinStart] = getBaseCoverage(...)` returns *`BinStart`*, a row vector of positive integers specifying the start position of each bin (when binning occurs).

Input Arguments

`BioObj`

Object of the `BioMap` class.

Default:

`StartPos`

Either of the following:

- Nonnegative integer that defines the start of a range in the reference sequence. *`StartPos`* must be less than *`EndPos`* and smaller than the total length of the reference sequence.
- Column vector of nonnegative integers, each defining the start of a range in the reference sequence.

Default:

`EndPos`

Either of the following:

- Nonnegative integer that defines the end of a range in the reference sequence. *`EndPos`* must be greater than *`StartPos`* and smaller than the total length of the reference sequence.
- Column vector of nonnegative integers, each defining the end of a range in the reference sequence.

`R`

Positive integer indexing the `SequenceDictionary` property of *`BioObj`*, or a character vector or string specifying the actual name of the reference.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

`binWidth`

Positive integer specifying the bin width, in number of base pairs (bp). Bins are centered within `min(`*`StartPos`*`)` and `max(`*`EndPos`*`)`. Thus, the first and last bins span approximately equally outside the range from `min(`*`StartPos`*`)` to `max (`*`EndPos`*`)`.

Note You cannot specify both `binWidth` and `numberOfBins`.

Default:

`numberOfBins`

Positive integer specifying the number of equal-width bins to use to span the requested region. Bins are centered within `min(`*`StartPos`*`)` and `max(`*`EndPos`*`)`. Thus, the first and last bins span approximately equally outside the range from `min(`*`StartPos`*`)` to `max (`*`EndPos`*`)`.

Note You cannot specify both `binWidth` and `numberOfBins`.

Default:

`binType`

Character vector or string specifying the binning algorithm. Choices are:

- `'max'` — From the bin, `getBaseCoverage` selects the base position with the most reads aligned to it, then uses its alignment coverage value for the bin.
- `'min'` — From the bin, `getBaseCoverage` selects the base position with the least reads aligned to it, then uses its alignment coverage value for the bin.

- `'mean'` — Uses the average alignment coverage, computed from all base positions within the bin.

Default: `'max'`

`complementRanges`

Specifies whether to return the alignment coverage for the base positions between segments, instead of within segments. If `true`, the length of *`Cov`* is `numel(min(`*`StartPos`*`):max(`*`EndPos`*`))`, and *`Cov`* contains `NaN` values for base positions within segments.

Default: `false`

`Spliced`

Logical specifying whether short reads are spliced during mapping (as in mRNA-to-genome mapping). N symbols in the `Signature` property of the object are not counted.

Default: `false`

Output Arguments

`Cov`

Row vector of nonnegative integers. This vector specifies the number of read sequences that align with each base position or bin in the requested regions. A set of ranges can be overlapping or segmented. For a range, the length of *`Cov`* is `numel(`*`StartPos`*`:`*`EndPos`*`)`. For a segmented range, the length of *`Cov`* is `numel(min(`*`StartPos`*`):max(`*`EndPos`*`))`. *`Cov`* contains `NaN` values for base positions between segments. When binning occurs, the number of elements in *`Cov`* equals the number of bins.

`BinStart`

Row vector of positive integers specifying the start position of each bin. *`BinStart`* is the same length as *`Cov`*. If no binning occurs, then *`BinStart`* equals `min(`*`StartPos`*`):max(`*`EndPos`*`)`.

Examples

Construct a `BioMap` object, and then return the alignment coverage of each of the first 12 base positions of the reference sequence:

```
% Construct a BioMap object from a SAM file
BMObj1 = BioMap('ex1.sam');
% Return the number of reads that align to each of
% the first 12 base positions of the reference sequence
cov = getBaseCoverage(BMObj1, 1, 12)

cov =

     1     1     2     2     3     4     4     4     5     5     5     5
```

Construct a `BioMap` object, and then return the alignment coverage of the range between 1 and 1000, on a bin-by-bin basis, using bins with a width of 100 bp:

```
% Construct a BioMap object from a SAM file
BMObj1 = BioMap('ex1.sam');
```

```
% Return the number of reads that align to each 100-bp bin
% in the 1:1000 range of the reference sequence. Also return the
% start position of each bin
[cov, bin_starts] = getBaseCoverage(BMObj1, 1, 1000, 'binWidth', 100)

cov =

    17    20    41    44    45    48    48    45    46    42


bin_starts =

     1   101   201   301   401   501   601   701   801   901
```

See Also

`BioMap` | `getCounts` | `getIndex` | `getAlignment` | `getCompactAlignment` | `align2cigar` | `cigar2align`

Topics

"Manage Sequence Read Data in Objects"

External Websites

Sequence Read Archive
SAM format specification

getCompactAlignment

Class: `BioMap`

Construct compact alignment represented in `BioMap` object

Syntax

```
CompAlignment = getCompactAlignment(BioObj, StartPos, EndPos)
CompAlignment = getCompactAlignment(BioObj, StartPos, EndPos, R)
CompAlignment = getCompactAlignment(..., 'ParameterName', ParameterValue)
[CompAlignment, Indices] = getCompactAlignment(...)
[CompAlignment, Indices, Rows] = getCompactAlignment(...)
```

Description

`CompAlignment = getCompactAlignment(BioObj, StartPos, EndPos)` returns *`CompAlignment`*, a character array containing the aligned read sequences from *`BioObj`*, a `BioMap` object, in a compact format. The read sequences must align within a specific region of the reference sequence, which is defined by *`StartPos`* and *`EndPos`*, two positive integers such that *`StartPos`* is less than *`EndPos`*, and both are smaller than the length of the reference sequence.

`CompAlignment = getCompactAlignment(BioObj, StartPos, EndPos, R)` selects the reference where `getCompactAlignment` reconstructs the alignment.

`CompAlignment = getCompactAlignment(..., 'ParameterName', ParameterValue)` accepts one or more comma-separated parameter name/value pairs. Specify *`ParameterName`* inside single quotes.

`[CompAlignment, Indices] = getCompactAlignment(...)` returns *`Indices`*, a vector of indices specifying the read sequences that align within a specific region of the reference sequence.

`[CompAlignment, Indices, Rows] = getCompactAlignment(...)` returns *`Rows`*, a vector of positive numbers specifying the row in *`CompAlignment`* where each read sequence is best displayed.

Input Arguments

`BioObj`

Object of the `BioMap` class.

Default:

`StartPos`

Positive integer that defines the start of a region of the reference sequence. *`StartPos`* must be less than *`EndPos`*, and smaller than the total length of the reference sequence.

Default:

`EndPos`

Positive integer that defines the end of a region of the reference sequence. *`EndPos`* must be greater than *`StartPos`*, and smaller than the total length of the reference sequence.

Default:

`R`

Positive integer indexing the `SequenceDictionary` property of *`BioObj`*, or a character vector or string specifying the actual name of the reference.

Parameter Name/Value Pairs

`Full`

Specifies whether or not to include only the read sequences that fully align with the defined region of the reference sequence, that is, they are completely contained within the region, and do not extend beyond the region. Choices are `true` or `false` (default).

Default: `false`

`TrimAlignment`

Specifies whether or not to trim empty leading and trailing columns from the alignment. Choices are `true` or `false`. Default is `false`, which does not trim the alignment, but includes any empty leading or trailing columns, and returns an alignment always of length *`EndPos`* – *`StartPos`* + `1`.

Default: `false`

Output Arguments

`CompAlignment`

Character array containing the aligned read sequences from *`BioObj`* that align within the requested region. The character array represents a compact alignment, that is each row of the character array contains one or more aligned sequences, such that the number of rows in the character array is minimized. Each aligned sequence includes only the sequence positions that fall within the requested region, and each aligned sequence can include gaps.

`Indices`

Vector of indices specifying the read sequences from *`BioObj`* that align within the requested region.

`Rows`

Vector of positive numbers specifying the row in *`CompAlignment`* where each read sequence is best displayed.

Examples

Construct a `BioMap` object, and then construct the compact alignment between positions 30 and 59 of the reference sequence:

```
% Construct a BioMap object from a SAM file
BMObj1 = BioMap('ex1.sam');
% Construct the compact alignment between positions 30 and 59 of
% the reference sequence, and return the indices of the reads in the
% compact alignment, as well as the row each read is in.
[CompAlignment, Ind, Row] = getCompactAlignment(BMObj1, 30, 59)

CompAlignment =

TAACTCG      GCCCAGCATTAGGGAGC
TAACTCGT           CATTAGGGAGC
TAACTCGTCC          ATTAGGGAGC
TAACTCTTCTCT         TTAGGGAGC
TAACTCGTCCATGG        TAGGGAGC
TAACTCGTCCCTGGCCCA           C
TAACTCGTCCATGGCCCAG
TAACTCGTCCATTGCCCAGC
TAACTCGTCCATGGCCCAGCATT
TAACTCGTCCATGGCCCAGCATTTGGG
TAACTCGTCCATGGCCCAGCATTAGGG
TAACTCGTCCATGGCCCAGCATTAGGGAGC
TAACTCGTCCATGGCCCAGCATTAGGGATC
TAACTCGTCCATGGCCCAGCATTAGGGAGC
 AACTCGTCCATGGCCCAGCATTAGGGAGC
      GTACATGGCCCAGCATTAGGGAGC
       TCCATGGCCCAGCATTAGGGCGC


Ind =

     1
     2
     3
     4
     5
     6
     7
     8
     9
    10
    11
    12
    13
    14
    15
    16
    17
    18
    19
    20
    21
    22
    23


Row =

     1
     2
     3
     4
     5
     6
     7
     8
     9
    10
    11
    12
    13
    14
    15
    16
    17
     1
     2
     3
     4
     5
     6
```

Algorithms

`getCompactAlignment` assumes the reference sequence has no gaps. Therefore, positions in reads corresponding to insertions (I) and padding (P) do not appear in the alignment.

Because soft clipped positions (S) are not associated with positions that align to the reference sequence, they do not appear in the alignment.

A skipped position (N) appears as a `-` (hyphen) in the alignment.

Hard clipped positions (H) do not appear in the sequences or the alignment.

See Also

`BioMap` | `getBaseCoverage` | `getAlignment` | `align2cigar` | `cigar2align`

Topics
“Manage Sequence Read Data in Objects”

External Websites
Sequence Read Archive
SAM format specification

getCounts

Class: `BioMap`

Return count of read sequences aligned to reference sequence in `BioMap` object

Syntax

```
Count = getCounts(BioObj, StartPos, EndPos)
GroupCount = getCounts(BioObj, StartPos, EndPos, Groups)
GroupCount = getCounts(BioObj, StartPos, EndPos, Groups, R)
___ = getCounts(___, Name,Value)
```

Description

`Count = getCounts(BioObj, StartPos, EndPos)` returns *`Count`*, a nonnegative integer specifying the number of read sequences in *`BioObj`*, a `BioMap` object, that align to a specific range or set of ranges in the reference sequence. The range or set of ranges are defined by *`StartPos`* and *`EndPos`*. *`StartPos`* and *`EndPos`* can be two nonnegative integers such that *`StartPos`* is less than *`EndPos`*, and both integers are smaller than the length of the reference sequence. *`StartPos`* and *`EndPos`* can also be two column vectors representing a set of ranges (overlapping or segmented).

By default, `getCounts` counts each read only once. Therefore, if a read spans multiple ranges, that read instance is counted only once. When *`StartPos`* and *`EndPos`* specify overlapping ranges, the overlapping ranges are considered as one range.

`GroupCount = getCounts(BioObj, StartPos, EndPos, Groups)` specifies *`Groups`*, a vector of integers or cell array of character vectors or string vector, indicating groups that segmented ranges belong to. The segmented ranges are treated independently.

`GroupCount = getCounts(BioObj, StartPos, EndPos, Groups, R)` specifies a reference for each of the segmented ranges defined by *`StartPos`*, *`EndPos`*, and *`Groups`*.

`___ = getCounts(___, Name,Value)` uses additional options specified by one or more `Name,Value` pair arguments.

Input Arguments

`BioObj`

Object of the `BioMap` class.

Default:

`StartPos`

Either of the following:

- Nonnegative integer that defines the start of a range in the reference sequence. *`StartPos`* must be less than *`EndPos`*, and smaller than the total length of the reference sequence.

- Column vector of nonnegative integers, each defining the start of a range in the reference sequence.

Default:

`EndPos`

Either of the following:

- Nonnegative integer that defines the end of a range in the reference sequence. *`EndPos`* must be greater than *`StartPos`*, and smaller than the total length of the reference sequence.
- Column vector of nonnegative integers, each defining the end of a range in the reference sequence.

Default:

`Groups`

Row vector of integers, cell array of character vectors, or string vector of the same size as *`StartPos`* and *`EndPos`*. This vector indicates the group to which each range belongs.

`R`

Vector of positive integers indexing the `SequenceDictionary` property of *`BioObj`*, or a cell array of character vectors or string vector of the reference names. *`R`* must be scalar or must have the same number of elements as *`Groups`*.

For a given value of *`Groups`*, all the corresponding elements in *`R`* must be the same.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

`Independent`

Logical that specifies whether to treat the ranges defined by *`StartPos`* and *`EndPos`* independently. If `true`, *`Count`* is a column vector containing the same number of elements as *`StartPos`* and *`EndPos`*. In this case, a read that spans multiple ranges, is counted once in each range.

Note This name-value pair argument is ignored when using the *`Groups`* input argument, because `getCounts` assumes that each group of ranges is independent.

Default: `false`

`Overlap`

Specifies the minimum number of base positions that a read must overlap in a range or set of ranges, to be counted. This value can be any of the following:

- Positive integer
- `'full'` — A read must be fully contained in a range or set of ranges to be counted.
- `'start'` — A read's start position must lie within a range or set of ranges to be counted.

Default: `1`

`Spliced`

Logical specifying whether short reads are spliced during mapping (as in mRNA-to-genome mapping). N symbols in the `Signature` property of the object are not counted.

Default: `false`

`Method`

Character vector or string specifying the method to measure the abundance of reads. Choices are:

- `'raw'` — Raw counts
- `'rpkm'` — Counts of reads per kilobase pairs per million aligned reads
- `'mean'` — Average coverage depth computed base-by-base
- `'max'` — Maximum coverage depth computed base-by-base
- `'min'` — Minimum coverage depth computed base-by-base
- `'sum'` — Sum of all aligned bases in all the reads

Default: `'raw'`

Output Arguments

`Count`

Either of the following:

- When `Independent` is `false`, this value is a nonnegative integer. The integer specifies the number of reads that align to a range or set of ranges (overlapping or segmented) of the reference sequence in *`BioObj`*, a `BioMap` object. Each read is counted only once, even if the read spans multiple ranges.
- When `Independent` is `true`, this value is a vector of nonnegative integers. This vector indicates the number of reads that align to the independent ranges specified by *`StartPos`* and *`EndPos`*. This vector contains the same number of elements as *`StartPos`* and *`EndPos`*.

`GroupCount`

Either of the following:

- If no reference or a single reference is specified, this value is a vector containing the number of reads for each unique group in *`Groups`*. The order of elements in *`GroupsCount`* corresponds to the ascending order of unique elements in *`Groups`*.
- If multiple references are specified, *`GroupCount`* is a cell array, where the ith element contains the number of reads for each unique group in the ith reference. The order of elements in *`GroupsCount`* corresponds to the ascending order of unique elements in *`R`*.

Examples

Compute the number of reads mapped to regions of reference sequence

Create a BioMap object.

```
obj = BioMap('ex1.sam');
```

Return the number of reads that cover at least one base of the segmented range 1:50 and 71:100. By default, the ranges are not treated independently, that is, a read is counted once even if it maps to both segmented ranges.

```
counts_1 = getCounts(obj,[1;71],[50;100])
```

```
counts_1 = 37
```

Compute the number of reads, treating the segmented ranges [1:50] and [71:100] independently. Observe that `sum(counts_2)` is greater than `counts_1` because there are four reads that span over the two segments and are counted twice in the second case.

```
counts_2 = getCounts(obj,[1;71],[50;100], 'Independent', true)
```

```
counts_2 = 2×1

    20
    21
```

Compute the number of reads that align to the segmented range 30:60 (associated with group 1) and the segmented range [1:10 50:60] (associated with group 2).

```
counts_3 = getCounts(obj,[1;30;50],[10;60;60],[2 1 2])
```

```
counts_3 = 2×1

    25
    22
```

Return the total number of reads aligned to the reference sequence.

```
getCounts(obj, min(getStart(obj)), max(getStop(obj)))
```

```
ans = 1482
```

See Also

`BioMap` | `getIndex` | `getBaseCoverage` | `getAlignment` | `getCompactAlignment` | `align2cigar` | `cigar2align` | `featurecount`

Topics
"Manage Sequence Read Data in Objects"

External Websites
Sequence Read Archive

SAM format specification

getCoverage

Class: `BioMap`

Compute read coverage in `BioMap` object

Note `getCoverage` has been removed. Use `getBaseCoverage`, `getCounts`, or `getIndex` instead.

Syntax

```
Cov = getCoverage(BioObj, StartPos, EndPos)
[Cov, Indices] = getCoverage(BioObj, StartPos, EndPos)
[Cov, Indices, Seqs] = getCoverage(BioObj, StartPos, EndPos)
... = getCoverage(BioObj, StartPos, EndPos, 'ParameterName', ParameterValue)
```

Description

`Cov = getCoverage(BioObj, StartPos, EndPos)` returns *`Cov`*, a nonnegative integer indicating the number of read sequences that cover (align within) a specific region of the reference sequence in *`BioObj`*, a `BioMap` object. The specific region of the reference sequence is defined by *`StartPos`* and *`EndPos`*. *`StartPos`* and *`EndPos`* can be two nonnegative integers such that *`StartPos`* is less than *`EndPos`*, and both are smaller than the length of the reference sequence. *`StartPos`* and *`EndPos`* can also be two column vectors representing a collection of regions of the reference sequence. In this case, *`Cov`* is a column vector of nonnegative integers indicating the number of read sequences that cover each region.

`[Cov, Indices] = getCoverage(BioObj, StartPos, EndPos)` also returns *`Indices`*, a vector of indices specifying the read sequences that align within a specific region of the reference sequence.

`[Cov, Indices, Seqs] = getCoverage(BioObj, StartPos, EndPos)` also returns *`Seqs`*, a cell array of strings containing the read sequences that align within a specific region of the reference sequence.

`... = getCoverage(BioObj, StartPos, EndPos, 'ParameterName', ParameterValue)` accepts one or more comma-separated parameter name/value pairs. Specify *`ParameterName`* inside single quotes.

Input Arguments

`BioObj`

Object of the `BioMap` class.

Default:

`StartPos`

Either of the following:

- Nonnegative integer that defines the start of a region of the reference sequence. *`StartPos`* must be less than *`EndPos`*, and smaller than the total length of the reference sequence.
- Column vector of nonnegative integers, each defining the start of a region of the reference sequence.

Default:

`EndPos`

Either of the following:

- Nonnegative integer that defines the end of a region of the reference sequence. *`EndPos`* must be greater than *`StartPos`*, and smaller than the total length of the reference sequence.
- Column vector of nonnegative integers, each defining the end of a region of the reference sequence.

Default:

Parameter Name/Value Pairs

`Base`

Specifies if the output *`Cov`* is computed base-by-base, that is determining the number of nongap symbols that align with each position in the specified region of the reference sequence. If `true`, *`Cov`* is a vector of positive integers corresponding to the base positions in the specified region of the reference sequence.

Default: `false`

`Full`

Specifies to include only the read sequences that fully align with the defined region of the reference sequence, that is, they are completely contained within the region, and do not extend beyond the region.

Default: `false`

Output Arguments

`Cov`

Either of the following:

- Nonnegative integer indicating the number of read sequences that cover (align within) a specific region of the reference sequence in *`BioObj`*.
- Column vector of nonnegative integers indicating the number of read sequences that cover each region specified by *`StartPos`* and *`EndPos`*, when they are both column vectors. In this case, *`Cov`* is the same length as *`StartPos`* and *`EndPos`*.

`Indices`

Vector of indices specifying the read sequences from *`BioObj`* that align within a specific region of the reference sequence.

`Seqs`

Cell array of strings containing the read sequences from *`BioObj`* that align within a specific region of the reference sequence. Each string is a sequence read without alignment information.

Examples

Construct a `BioMap` object, and then retrieve the coverage of the first 50 positions of the reference sequence:

```
% Construct a BioMap object from a SAM file
BMObj1 = BioMap('ex1.sam');
% Retrieve the number of sequences that cover the first 50
% positions of the reference sequence
cov = getCoverage(BMObj1, 1, 50)

cov =

    20
```

Construct a `BioMap` object, and then retrieve the starting positions for the read sequences that cover the first 50 positions of the reference sequence:

```
% Construct a BioMap object from a SAM file
BMObj1 = BioMap('ex1.sam');
% Retrieve the number of sequences that cover the first 50
% positions of the reference sequence
% Also retrieve the indices of these sequences
[cov, idx] = getCoverage(BMObj1, 1, 50);
% Use the indices for these sequences to determine their start
% positions
startPositions = getStart(BMObj1, idx);
```

Construct a `BioMap` object, and then retrieve the coverage of the first 50 positions of the reference sequence, considering only read sequences that align fully within the region:

```
% Construct a BioMap object from a SAM file
BMObj1 = BioMap('ex1.sam');
% Retrieve the number of sequences that cover the first 50
% positions of the reference sequence
% Consider only read sequences that align fully within the region
fullCov = getCoverage(BMObj1, 1, 50, 'full', true)

fullCov =

    8
```

Construct a `BioMap` object, and then retrieve the coverage for the first 10 positions of the reference sequence, on a base-by-base basis:

```
% Construct a BioMap object from a SAM file
BMObj1 = BioMap('ex1.sam');
% Retrieve the number of sequences that cover each base position of
% the first 10 positions of the reference sequence
baseCov = getCoverage(BMObj1, 1, 10, 'base', true)

baseCov =
     1
     1
     2
     2
     3
     4
     4
     4
     5
     5
```

Tips

Use the *`Indices`* output from the `getCoverage` method as input to other `BioMap` methods. Doing so lets you determine other information about the read sequences in the coverage region, such as header, start position, mapping quality, etc.

See Also

`BioMap` | `getAlignment` | `getCompactAlignment` | `align2cigar` | `cigar2align`

Topics

"Manage Sequence Read Data in Objects"

External Websites

Sequence Read Archive
SAM format specification

getFlag

Class: `BioMap`

Retrieve read sequence flags from `BioMap` object

Syntax

```
Flag = getFlag(BioObj)
Flag = getFlag(BioObj, Subset)
```

Description

`Flag = getFlag(BioObj)` returns *`Flag`*, a vector of nonnegative integers indicating the bit-wise information that specifies the status of the 11 flags described by the SAM format specification. Each integer corresponds to one read sequence from a `BioMap` object.

`Flag = getFlag(BioObj, Subset)` returns flag integers for only object elements specified by *`Subset`*.

Input Arguments

`BioObj`

Object of the `BioMap` class.

Default:

`Subset`

One of the following to specify a subset of the elements in *`BioObj`*:

- Vector of positive integers
- Logical vector
- Cell array of character vectors or string vector containing valid sequence headers

Note If you use a cell array of headers to specify *`Subset`*, be aware that a repeated header specifies all elements with that header.

Default:

Output Arguments

`Flag`

Vector of nonnegative integers. Each integer corresponds to one read sequence and indicates the bit-wise information that specifies the status of the 11 flags described by the SAM format specification. These flags describe different sequencing and alignment aspects of a read sequence. *`Flag`* includes flag integers for only read sequences specified by *`Subset`*.

Examples

Construct a `BioMap` object, and then retrieve the SAM flag values for different elements in the object:

```
% Construct a BioMap object from a SAM file
BMObj1 = BioMap('ex1.sam');
% Retrieve integer specifying bit-wise information for 11
% SAM flags of the second element
flagValue = getFlag(BMObj1, 2)

flagValue =

    73

% Retrieve integers specifying bit-wise information for 11
% SAM flags of the first and third elements
flagValues = getFlag(BMObj1, [1 3])

flagValues =

    73
   137

% Retrieve integers specifying bit-wise information for 11
% SAM flags of all elements
allFlagValues = getFlag(BMObj1);

% Determine the status of the fourth flag (mate is unmapped)
% for the second element, which has a flag value of 73
bitget(73, 4)

ans =

     1
```

Tips

After using the `getFlag` method to return the integer specifying the bit-wise information for the SAM flags, use the `bitget` function to determine the status of a specific SAM flag. For more information, see “Examples” on page 1-184.

Alternatives

An alternative to using the `getFlag` method is to use dot indexing with the `Flag` property:

```
BioObj.Flag(Indices)
```

In the previous syntax, *`Indices`* is a vector of positive integers or a logical vector. *`Indices`* cannot be a cell array of character vectors or string vector containing sequence headers.

See Also

`BioMap` | `setFlag` | `bitget`

Topics

“Manage Sequence Read Data in Objects”

External Websites

Sequence Read Archive

SAM format specification

getIndex

Class: `BioMap`

Return indices of read sequences aligned to reference sequence in `BioMap` object

Syntax

```
Indices = getIndex(BioObj, StartPos, EndPos)
Indices = getIndex(BioObj, StartPos, EndPos, R)
Indices = getIndex(..., Name,Value)
```

Description

`Indices = getIndex(BioObj, StartPos, EndPos)` returns *`Indices`*, a column vector of indices specifying the read sequences that align to a range or set of ranges in the reference sequence in *`BioObj`*, a `BioMap` object. The range or set of ranges are defined by *`StartPos`* and *`EndPos`*. *`StartPos`* and *`EndPos`* can be two nonnegative integers such that *`StartPos`* is less than *`EndPos`*, and both integers are smaller than the length of the reference sequence. *`StartPos`* and *`EndPos`* can also be two column vectors representing a set of ranges (overlapping or segmented).

`getIndex` includes each read only once. Therefore, if a read spans multiple ranges, the index for that read appears only once.

`Indices = getIndex(BioObj, StartPos, EndPos, R)` selects the reference associated with the range specified by *`StartPos`* and *`EndPos`*.

`Indices = getIndex(..., Name,Value)` returns indices with additional options specified by one or more `Name,Value` pair arguments.

Input Arguments

`BioObj`

Object of the `BioMap` class.

Default:

`StartPos`

Either of the following:

- Nonnegative integer that defines the start of a range in the reference sequence. *`StartPos`* must be less than *`EndPos`*, and smaller than the total length of the reference sequence.
- Column vector of nonnegative integers, each defining the start of a range in the reference sequence.

Default:

`EndPos`

Either of the following:

- Nonnegative integer that defines the end of a range in the reference sequence. *`EndPos`* must be greater than *`StartPos`*, and smaller than the total length of the reference sequence.
- Column vector of nonnegative integers, each defining the end of a range in the reference sequence.

Default:

`R`

Positive integer indexing the `SequenceDictionary` property of *`BioObj`*, or a character vector or string specifying the actual name of the reference.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

`Overlap`

Specifies the minimum number of base positions that a read must overlap in a range or set of ranges, to be included. This value can be any of the following:

- Positive integer
- `'full'` — A read must be fully contained in a range or set of ranges to be counted.
- `'start'` — A read's start position must lie within a range or set of ranges to be counted.

Default: `1`

`Depth`

Specifies to decimate the output indices. The coverage depth at any base position is less than or equal to `Depth`, a positive integer.

Default: `Inf`

`Spliced`

Logical specifying whether short reads are spliced during mapping (as in mRNA-to-genome mapping). N symbols in the `Signature` property of the object are not counted.

Default: `false`

Output Arguments

`Indices`

Column vector of indices specifying the reads that align to a range or set of ranges in the specified reference sequence in *`BioObj`*, a `BioMap` object.

Examples

Construct a `BioMap` object, and then use the indices of the reads to retrieve the start and stop positions for the reads that are fully contained in the first 50 positions of the reference sequence:

```
% Construct a BioMap object from a SAM file
BMObj1 = BioMap('ex1.sam');
% Return the indices of reads that are fully contained in the
% first 50 positions of the reference sequence
indices = getIndex(BMObj1, 1, 50, 'overlap', 'full');
% Use these indices to return the start and stop positions of
% the reads
starts = getStart(BMObj1, indices)
stops = getStop(BMObj1, indices)

starts =

           1
           3
           5
           6
           9
          13
          13
          15

stops =

          36
          37
          39
          41
          43
          47
          48
          49
```

Construct a `BioMap` object, and then use the indices of the reads to retrieve the sequences for the reads whose alignments overlap a segmented range by at least one base pair:

```
% Construct a BioMap object from a SAM file
BMObj1 = BioMap('ex1.sam');
% Return the indices of the reads that overlap the
% segmented range 98:100 and 198:200, by at least 1 base pair
indices = getIndex(BMObj1, [98;198], [100;200], 'overlap', 1);
% Use these indices to return the sequences of the reads
sequences = getSequence(BMObj1, indices);
```

Tips

Use the *`Indices`* output from the `getIndex` method as input to other `BioMap` methods. Doing so lets you retrieve other information about the reads in the range, such as header, start position, mapping quality, sequences, etc.

See Also

`BioMap` | `getStart` | `getStop` | `getSequence` | `getCounts` | `getBaseCoverage` | `getAlignment` | `getCompactAlignment` | `align2cigar` | `cigar2align`

Topics

"Manage Sequence Read Data in Objects"

External Websites

Sequence Read Archive
SAM format specification

getInfo

Class: `BioMap`

Retrieve information for single element of `BioMap` object

Syntax

`*Info* = getInfo(*BioObj*, *Element*)`

Description

`*Info* = getInfo(*BioObj*, *Element*)` returns *`Info`*, a tab-delimited character vector containing information about a single element in *`BioObj`*, a `BioMap` object.

Input Arguments

`BioObj`

Object of the `BioMap` class.

Default:

`Element`

One of the following to specify one element in *`BioObj`*:

- Scalar specifying an element index
- Logical vector
- Character vector containing a valid sequence header

Default:

Output Arguments

`Info`

Tab-delimited character vector containing information about a single element in *`BioObj`*, a `BioMap` object. The character vector contains the information from the following properties in order:

- `Header`
- `Flag`
- `Start`
- `MappingQuality`
- `Signature`
- `Sequence`
- `Quality`

Examples

Construct a `BioMap` object, and then retrieve information for the second element in the object:

```
% Construct a BioMap object from a SAM file
BMObj1 = BioMap('ex1.sam');
% Retrieve information for the second element in the object
element2Info = getInfo(BMObj1, 2)
```

```
element2Info =

EAS54_65:7:152:368:113    73    3    99    35M
CTAGTGGCTCATTGTAAATGTGTGGTTTAACTCGT
<<<<<<<<<<0<<<<655<<7<<<:9<<3/:<6):
```

See Also

`BioMap`

Topics

"Manage Sequence Read Data in Objects"

External Websites

Sequence Read Archive

SAM format specification

getMappingQuality

Class: `BioMap`

Retrieve sequence mapping quality scores from `BioMap` object

Syntax

```
MappingQuality = getMappingQuality(BioObj)
MappingQuality = getMappingQuality(BioObj, Subset)
```

Description

`MappingQuality = getMappingQuality(BioObj)` returns *`MappingQuality`*, a vector of integers specifying mapping quality scores for each read sequence in *`BioObj`*, a `BioMap` object.

`MappingQuality = getMappingQuality(BioObj, Subset)` returns mapping quality scores for only object elements specified by *`Subset`*.

Input Arguments

`BioObj`

Object of the `BioMap` class.

Default:

`Subset`

One of the following to specify a subset of the elements in *`BioObj`*:

- Vector of positive integers
- Logical vector
- Cell array of character vectors or string vector containing valid sequence headers

Note If you use a cell array of headers to specify *`Subset`*, be aware that a repeated header specifies all elements with that header.

Default:

Output Arguments

`MappingQuality`

`MappingQuality` property of a subset of elements in *`BioObj`*. *`MappingQuality`* is a vector of integers specifying the mapping quality scores for read sequences specified by *`Subset`*.

Examples

Construct a `BioMap` object, and then retrieve the mapping quality scores for different elements in the object:

```
% Construct a BioMap object from a SAM file
BMObj1 = BioMap('ex1.sam');
% Retrieve the mapping quality property of the second element in
% the object
MQ_2 = getMappingQuality(BMObj1, 2)

MQ_2 =

   99

% Retrieve the mapping quality properties of the first and third
% elements in the object
MQ_1_3 = getMappingQuality(BMObj1, [1 3])

MQ_1_3 =

   99
   99

% Retrieve the mapping quality properties of all elements in the
% object
MQ_All = getMappingQuality(BMObj1);
```

Alternatives

An alternative to using the `getMappingQuality` method is to use dot indexing with the `MappingQuality` property:

```
BioObj.MappingQuality(Indices)
```

In the previous syntax, *`Indices`* is a vector of positive integers or a logical vector. *`Indices`* cannot be a cell array of character vectors or string vector containing sequence headers.

See Also

`BioMap` | `setMappingQuality`

Topics
"Manage Sequence Read Data in Objects"

External Websites
Sequence Read Archive
SAM format specification

getMatePosition

Retrieve mate positions of read sequences from `BioMap` object

Syntax

```
MatePos = getMatePosition(BioObj)
MatePos = getMatePosition(BioObj,Subset)
```

Description

`MatePos = getMatePosition(BioObj)` returns the mate positions of read sequences with respect to the position numbers in the reference sequence from `BioObj`.

`MatePos = getMatePosition(BioObj,Subset)` returns mate positions for only read sequences specified by `Subset`.

Examples

Get Mate Positions from BioMap Object

Construct a `BioMap` object from a `SAM` file and determine the header for the 17th element.

```
BioObj = BioMap('ex1.sam');
hdr = BioObj.Header(17)
```

```
hdr = 1x1 cell array
    {'EAS114_32:5:78:583:499'}
```

Retrieve the `MatePosition` property of the 17th element in the object using the header.

```
MatePos_17 = getMatePosition(BioObj,hdr)
```

```
MatePos_17 = 2x1 uint32 column vector

   229
    37
```

Notice the previous example returned two mate positions. This is because the header EAS114_32:5:78:583:499 is a repeated header in the `BMObj1` object. The `getMatePosition` method returns mate positions for all elements in the object with that header.

Retrieve the `MatePosition` properties of the 37th and 47th elements in the object.

```
MatePos_37_47 = getMatePosition(BioObj, [37 47])
```

```
MatePos_37_47 = 2x1 uint32 column vector

    95
   283
```

Retrieve the `MatePosition` properties of all elements in the object. Examine the size of the returned mate positions.

```
MatePos_All = getMatePosition(BioObj);
size(MatePos_All)
```

```
ans = 1×2

        1501           1
```

Input Arguments

`BioObj` — Object for read sequences
`BioMap` object

Object for read sequences, specified as a `BioMap` object. Construct `BioObj` using `BioMap`.

`Subset` — Subset of elements in `BioObj`
vector of positive integers | logical vector | cell array of character vectors containing valid sequence headers | string vector containing valid sequence headers

Subset of the elements in `BioObj`, specified as one of the following:

- Vector of positive integers
- Logical vector
- Cell array of character vectors containing valid sequence headers
- String vector containing valid sequence headers

Note If you use a cell array of headers to specify `Subset`, a repeated header specifies all elements with that header.

Example: `[5 26]`

Data Types: `single` | `double` | `logical` | `char` | `string` | `cell`

Output Arguments

`MatePos` — Mate positions of read sequences with respect to the position numbers in the reference sequence
vector of nonnegative integers

Mate positions of read sequences with respect to the position numbers in the reference sequence, returned as a vector of nonnegative integers. `MatePos` gives the `MatePosition` property of all or a subset of elements in `BioObj`.

Not all values in the `MatePosition` vector represent valid mate positions. For example, mates that map to a different reference sequence or mates that do not map. To determine if a mate position is valid, use the `filterByFlag` method with the `'pairedInMap'` name-value argument.

Alternative Functionality

An alternative to using `getMatePosition` is to use dot indexing with the `MatePosition` property:

```
BioObj.MatePosition(Indices)
```

In this syntax, `Indices` is a vector of positive integers or a logical vector. `Indices` cannot be a cell array of character vectors or string vector containing sequence headers.

Version History

Introduced in R2010b

See Also

`BioMap` | `setMatePosition` | `filterByFlag`

Topics

"Manage Sequence Read Data in Objects"

External Websites

Sequence Read Archive
SAM format specification

getReference

Class: `BioMap`

Retrieve reference sequence from `BioMap` object

Syntax

`*Ref* = getReference(*BioObj*)`

Description

`*Ref* = getReference(*BioObj*)` returns the name of the reference sequence from a `BioMap` object. This is the `Reference` property of the object.

Input Arguments

`BioObj`

Object of the `BioRead` or `BioMap` class.

Default:

Output Arguments

`Ref`

`Reference` property of *`BioObj`*, the `BioMap` object. It is a character vector or string vector specifying the name of the reference sequence.

Examples

Construct a `BioMap` object, and then retrieve the reference sequence from the object:

```
% Construct a BioMap object from a SAM file
BMObj1 = BioMap('ex1.sam');
% Retrieve the reference sequence from the object
refSeq = getReference(BMObj1)

refSeq =

seq1
```

Alternatives

An alternative to using the `getReference` method is to use dot indexing with the `Reference` property:

```
BioObj.Reference
```

See Also

`BioMap` | `setReference`

Topics

"Manage Sequence Read Data in Objects"

External Websites

Sequence Read Archive

SAM format specification

getSignature

Class: `BioMap`

Retrieve signature (alignment information) from `BioMap` object

Syntax

```
Signature = getSignature(BioObj)
Signature = getSignature(BioObj, Subset)
```

Description

`Signature` `= getSignature(`*`BioObj`*`)` returns *`Signature`*, a cell array of CIGAR-formatted strings, each representing how a read sequence in a `BioMap` object aligns to the reference sequence.

`Signature` `= getSignature(`*`BioObj`*`,` *`Subset`*`)` returns signature strings for only object elements specified by *`Subset`*.

Input Arguments

`BioObj`

Object of the `BioMap` class.

Default:

`Subset`

One of the following to specify a subset of the elements in *`BioObj`*:

- Vector of positive integers
- Logical vector
- Cell array of character vectors or string vector containing valid sequence headers

Note If you use a cell array of headers to specify *`Subset`*, be aware that a repeated header specifies all elements with that header.

Default:

Output Arguments

`Signature`

`Signature` property of a subset of elements in *`BioObj`*. *`Signature`* is a cell array of CIGAR-formatted strings, each representing how read sequences, specified by *`Subset`*, align to the reference sequence.

Examples

Construct a `BioMap` object, and then retrieve the signatures for different elements in the object:

```
% Construct a BioMap object from a SAM file
BMObj1 = BioMap('ex1.sam');
% Retrieve the signature property of the second element in
% the object
Sig_2 = getSignature(BMObj1, 2)

Sig_2 =

    '35M'

% Retrieve the signature properties of the first and third
% elements in the object
Sig_1_3 = getSignature(BMObj1, [1 3])

Sig_1_3 =

    '36M'
    '35M'

% Retrieve the signature properties of all elements in the object
Sig_All = getSignature(BMObj1);
```

Alternatives

An alternative to using the `getSignature` method is to use dot indexing with the `Signature` property:

`BioObj``.Sgnature(`*`Indices`*`)`

In the previous syntax, *`Indices`* is a vector of positive integers or a logical vector. *`Indices`* cannot be a cell array of character vectors or string vector containing sequence headers.

See Also

`BioMap` | `setSignature` | `getAlignment`

Topics

"Manage Sequence Read Data in Objects"

External Websites

Sequence Read Archive
SAM format specification

getStart

Class: `BioMap`

Retrieve start positions of aligned read sequences from `BioMap` object

Syntax

```
Start = getStart(BioObj)
Start = getStart(BioObj, Subset)
```

Description

`Start = getStart(BioObj)` returns `Start`, a vector of integers specifying the start position of aligned read sequences with respect to the position numbers in the reference sequence from a `BioMap` object.

`Start = getStart(BioObj, Subset)` returns a start position for only read sequences specified by *`Subset`*.

Input Arguments

`BioObj`

Object of the `BioMap` class.

Default:

`Subset`

One of the following to specify a subset of the elements in *`BioObj`*:

- Vector of positive integers
- Logical vector
- Cell array of character vectors or string vector containing valid sequence headers

Note If you use a cell array of headers to specify *`Subset`*, be aware that a repeated header specifies all elements with that header.

Default:

Output Arguments

`Start`

`Start` property of a subset of elements in *`BioObj`*. It is a vector of integers specifying the start position of aligned read sequences with respect to the position numbers in the reference sequence. It includes the start positions for only read sequences specified by *`Subset`*.

Examples

Construct a `BioMap` object, and then retrieve the start position for different sequences in the object:

```
% Construct a BioMap object from a SAM file
BMObj1 = BioMap('ex1.sam');
% Retrieve the start property of the second element in the object
Start_2 = getStart(BMObj1, 2)

Start_2 =

           3

% Retrieve the start properties of the first and third elements in
% the object
Start_1_3 = getStart(BMObj1, [1 3])

Start_1_3 =

           1
           5

% Retrieve the start properties of all elements in the object
Start_All = getStart(BMObj1);
```

Alternatives

An alternative to using the `getStart` method is to use dot indexing with the `Start` property:

```
BioObj.Start(Indices)
```

In the previous syntax, *`Indices`* is a vector of positive integers or a logical vector. *`Indices`* cannot be a cell array of character vectors or string vector containing sequence headers.

See Also

`BioMap` | `setStart` | `getStop`

Topics

“Manage Sequence Read Data in Objects”

External Websites

Sequence Read Archive

SAM format specification

getSummary

Class: `BioMap`

Print summary of `BioMap` object

Syntax

```
getSummary(BioObj)
ds = getSummary(BioObj)
```

Description

`getSummary(BioObj)` prints a summary of a `BioMap` object. The summary includes the names of references, the number of sequences mapped to each reference, and the genomic range that the sequences cover in each reference.

`ds = getSummary(BioObj)` returns the summary information in a `dataset` array.

Input Arguments

`BioObj`

Object of the `BioMap` class.

Output Arguments

`ds`

`dataset` array containing the summary of the `BioMap` object, `BioObj`. The dataset array has an observation (row) for each reference in `BioObj`, and two variables (columns): the number of sequences mapped to each reference and the genomic range that the sequences cover in each reference.

`getSummary` stores additional metadata for the `BioMap` object in the `UserData` property of `ds`, which you can access using `ds.Properties.UserData`.

Examples

Construct a `BioMap` object, and then display a summary of the object:

```
% Construct a BioMap object from a SAM file
BMObj2 = BioMap('ex2.sam');
getSummary(BMObj2)

BioMap summary:
                                  Name: ''
                        Container_Type: 'Data is file indexed.'
             Total_Number_of_Sequences: 3307
    Number_of_References_in_Dictionary: 2
```

```
            Number_of_Sequences    Genomic_Range
    seq1    1501                   1  1569
    seq2    1806                   1  1567
```

See Also
`BioMap`

Topics
"Manage Sequence Read Data in Objects"

External Websites
Sequence Read Archive
SAM format specification

bioinfo.pipeline.Input

Input port object for bioinformatics pipeline block

Description

Each input port of a `bioinfo.pipeline.Block` object is a `bioinfo.pipeline.Input` object.

Creation

Create the object using `bioinfo.pipeline.Input`.

Properties

`SplitDimension` — Instruction on how to split block inputs
`[]` (default) | vector of positive integers | `"all"`

Instruction on how to split the block inputs across multiple runs of block in a pipeline, specified as a vector of positive integers or `"all"`.

Some of the blocks in a bioinformatics pipeline operate on their input data arrays as one single input while other blocks can operate on individual elements or slices of the input data array independently. The `SplitDimension` property of a block input controls how to split the block input data (or input array) across multiple runs of the same block in a pipeline. By default, the block input data are passed unchanged (that is, there is no dimensional splitting of the input data) to the `run` method of the block, which means that the block runs once for all of the input data array.

Specify a vector of integers to indicate which dimensions of the input array to split and pass to the block `run` method. By splitting the input array, you are specifying how many times you want to run the same block with different inputs. Use `"all"` to pass all elements of the input value to the `run` method of the block independently. If there are *n* elements, the block runs *n* times independently. For example, you can use a `Bowtie2` block to align three input files to a single SAM file, or use `"all"` to let `Bowtie2` run three times, aligning each input file to a distinct SAM file.

When a block has a single input with split dimensions, the input value is split in the corresponding dimensions (such as row-dimension or column-dimension) before being passed to the `run` method of the block. The total number of times the block runs within a pipeline is the product of the sizes of the input value in the split dimensions.

For details, see “Bioinformatics Pipeline SplitDimension”.

Data Types: `double` | `char` | `string`

`Required` — Flag to indicate input port is required
`true` or 1 | `false` or 0

This property is read-only.

Flag to indicate if the input port is required for the block to run, specified as a numeric or logical 1 (`true`) or 0 (`false`).

A required input port (`Required=true`) must be satisfied on page 1-1648. Otherwise, the pipeline fails to compile and does not run.

You can set the value as true or false when you define a block subclass. For details, see “Subclass Pipeline Block” on page 1-12.

Data Types: `double` | `logical`

Value — Input port value
`bioinfo.pipeline.datatypes.Unset` (default)

Input port value. By default, the value is set as a `bioinfo.pipeline.datatypes.Unset` object which means that no value is provided, and the input value comes from a connected upstream block or input structure passed to the `run` call.

If an input port with a set value is also connected to an output port of another block, the value coming from the connected block is used instead of the set value.

Examples

Split Input SAM files and Assemble Transcriptomes Using Bioinformatics Pipeline

Import the pipeline and block objects needed for the example.

```
import bioinfo.pipeline.Pipeline
import bioinfo.pipeline.blocks.*
```

Create a pipeline.

```
P = Pipeline
```

```
P = 
  Pipeline with properties:

        Blocks: [0×1 bioinfo.pipeline.Block]
    BlockNames: [0×1 string]
```

Use a `FileChooser` block to select the provided SAM files. The files contain aligned reads for *Mycoplasma pneumoniae* from two samples.

```
fileChooserBlock = FileChooser([which("Myco_1_1.sam"); which("Myco_1_2.sam")]);
```

Create a `Cufflinks` block.

```
cufflinksBlock = Cufflinks;
```

Add the blocks to the pipeline.

```
addBlock(P,[fileChooserBlock,cufflinksBlock]);
```

Connect the blocks.

```
connect(P,fileChooserBlock,cufflinksBlock,["Files","GenomicAlignmentFiles"]);
```

Set `SplitDimension` to `1` for the `GenomicAlignmentFiles` input port. The value of 1 corresponds to the row dimension of the input, which means that the `Cufflinks` block will run on each individual SAM files (`Myco_1_1.sam` and `Myco_1_1.sam`).

```
cufflinksBlock.Inputs.GenomicAlignmentFiles.SplitDimension = 1;
```

Run the pipeline. The pipeline runs `Cufflinks` block two times independently and generates a set of four files for each SAM file.

```
run(P);
```

Get the block results.

```
cufflinksResults = results(P,cufflinksBlock)
```

```
cufflinksResults = struct with fields:
           TranscriptsGTFFile: [2×1 bioinfo.pipeline.datatypes.File]
             IsoformsFPKMFile: [2×1 bioinfo.pipeline.datatypes.File]
                GenesFPKMFile: [2×1 bioinfo.pipeline.datatypes.File]
    SkippedTranscriptsGTFFile: [2×1 bioinfo.pipeline.datatypes.File]
```

Use the process table to check the total number of runs for each block. `Cufflinks` ran two times independently.

```
t = processTable(P,Expanded=true)
```

```
t=3×5 table
         Block          Status           RunStart                  RunEnd               RunErrors
    _______________    _________    ____________________    ____________________    ____________

    "FileChooser_1"    Completed    27-Jan-2023 14:54:26    27-Jan-2023 14:54:26    {0×0 MExcepti
    "Cufflinks_1"      Completed    27-Jan-2023 14:54:26    27-Jan-2023 14:54:28    {0×0 MExcepti
    "Cufflinks_1"      Completed    27-Jan-2023 14:54:28    27-Jan-2023 14:54:30    {0×0 MExcepti
```

Set `SplitDimension` to empty `[]` (which is the default). In this case, the pipeline does split the input files and runs `Cufflinks` just once for both SAM files, processing each SAM file one after another.

```
cufflinksBlock.Inputs.GenomicAlignmentFiles.SplitDimension = [];
deleteResults(P,IncludeFiles=true);
run(P);
cufflinksResults = results(P,cufflinksBlock)
```

```
cufflinksResults = struct with fields:
           TranscriptsGTFFile: [2×1 bioinfo.pipeline.datatypes.File]
             IsoformsFPKMFile: [2×1 bioinfo.pipeline.datatypes.File]
                GenesFPKMFile: [2×1 bioinfo.pipeline.datatypes.File]
    SkippedTranscriptsGTFFile: [2×1 bioinfo.pipeline.datatypes.File]
```

Check the process table, which confirms that `Cufflinks` ran just once.

```
t2 = processTable(P,Expanded=true)
```

```
t2=2×5 table
         Block          Status           RunStart                  RunEnd               RunErrors
    _______________    _________    ____________________    ____________________    ____________
    "FileChooser_1"    Completed    27-Jan-2023 14:54:30    27-Jan-2023 14:54:30    {0×0 MExcept
    "Cufflinks_1"      Completed    27-Jan-2023 14:54:30    27-Jan-2023 14:54:33    {0×0 MExcept
```

Tip: you can speed up the pipeline run by setting `UseParallel=true` if you have Parallel Computing Toolbox™. The pipeline can schedule independent executions of blocks on parallel pool workers.

```
run(P,UseParallel=true)
```

Version History

Introduced in R2023a

See Also

`bioinfo.pipeline.Output` | `bioinfo.pipeline.Block` | `bioinfo.pipeline.datatypes.Unset` | `run`

inputNames

Package: `bioinfo.pipeline`

Get names of unconnected block inputs from pipeline

Syntax

```
names = inputNames(pipeline)
names = inputNames(pipeline,IncludeOptional=tf)
```

Description

`names = inputNames(pipeline)` returns the input port names for all the blocks in the pipeline that are not connected.

`names = inputNames(pipeline,IncludeOptional=tf)` specifies whether to include the names of optional input ports.

Examples

Connect Blocks in Bioinformatics Pipeline

Import the Pipeline and block objects needed for the example.

```
import bioinfo.pipeline.Pipeline
import bioinfo.pipeline.blocks.*
```

Create a pipeline.

```
qcpipeline = Pipeline;
```

Select an input FASTQ file using a `FileChooser` block.

```
fastqfile = FileChooser(which("SRR005164_1_50.fastq"));
```

Create a `SeqFilter` block.

```
sequencefilter = SeqFilter;
```

Add blocks to the pipeline. Optionally, you can specify the block names.

```
addBlock(qcpipeline,[fastqfile,sequencefilter],["FF","SF"]);
```

Connect the output of the first block to the input of the second block. To do so, you need to first check the input and output port names of the corresponding blocks.

View the Outputs (port of the first block) and Inputs (port of the second block).

```
fastqfile.Outputs
```

```
ans = struct with fields:
    Files: [1×1 bioinfo.pipeline.Output]
```

```
sequencefilter.Inputs
```

```
ans = struct with fields:
    FASTQFiles: [1×1 bioinfo.pipeline.Input]
```

Connect the `Files` output port of the `fastqfile` block to the `FASTQFiles` port of `sequencefilter` block. The output is a 1-by-2 string array, indicating the names of two ports that are now connected. You can use either the block objects themselves or the block names that you have defined previously. The next command uses the block names to identify and connect these two blocks.

```
connectedports = connect(qcpipeline,"FF","SF",["Files","FASTQFiles"])
```

```
connectedports = 1×2 string
    "Files"    "FASTQFiles"
```

You can also query the names of connected ports using `portMap`.

```
portnames = portMap(qcpipeline,"FF","SF")
```

```
portnames = 1×2 string
    "Files"    "FASTQFiles"
```

Run Bioinformatics Pipeline Using Input Structure

Import the Pipeline and block objects needed for the example.

```
import bioinfo.pipeline.Pipeline
import bioinfo.pipeline.blocks.*
```

Create a pipeline.

```
P = Pipeline;
```

Create a `Bowtie2Build` block to build index files for the reference genome.

```
bowtie2build = Bowtie2Build;
```

Create a `Bowtie2` block to map the read sequences to the reference sequence.

```
bowtie2 = Bowtie2;
```

Add the blocks to the pipeline.

```
addBlock(P,[bowtie2build,bowtie2],["bowtie2build","bowtie2"]);
```

Get the list of names of all the required input ports from every block in the pipeline that are needed to be set or connected. `IndexBaseName` is an input port of both `bowtie2build` and `bowtie2` block. `Reads1File` is the input port of the `bowtie2` block and `ReferenceFASTAFile` is the input of `bowtie2build` block.

```
portnames = inputNames(P)
```

```
portnames = 1×3 string
    "IndexBaseName"    "Reads1Files"    "ReferenceFASTAFiles"
```

Some blocks have optional input ports. To see the names of these ports, set `IncludeOptional=true`. For instance, the `Bowtie2` block has an optional input port (`Reads2Files`) that accept files for the second mate reads when you have paired-end read data.

```
allportnames = inputNames(P,IncludeOptional=true)
```

```
allportnames = 1×4 string
    "IndexBaseName"    "Reads1Files"    "Reads2Files"    "ReferenceFASTAFiles"
```

Create an input structure to set the input port values of the `bowtie2` and `bowtie2build` blocks. Specifically, set `IndexBaseName` to `"Dmel_chr4"` which is the base name for the reference index files for the Drosophila genome. Set `Reads1Files` to `"SRR6008575_10k_1.fq"` and `Reads2Files` to `"SRR6008575_10k_2.fq"`. Set `ReferenceFASTAFile` to "`Dmel_chr4.fa`". These read files are already provided with the toolbox.

```
inputStruct.IndexBaseName = "Dmel_chr4";
inputStruct.Reads1Files   = "SRR6008575_10k_1.fq";
inputStruct.Reads2Files   = "SRR6008575_10k_2.fq";
inputStruct.ReferenceFASTAFiles = "Dmel_chr4.fa";
```

Optionally, you can compile and check if the input structure is set up correctly. Note that this compilation also happens automatically when you run the pipeline.

```
compile(P,inputStruct);
```

Run the pipeline using the structure as an input.

```
run(P,inputStruct);
```

Get the `bowtie2` block result after the pipeline finishes running.

```
wait(P);
mappedFile = results(P,bowtie2)
```

```
mappedFile = struct with fields:
    SAMFile: [1×1 bioinfo.pipeline.datatypes.File]
```

The `Bowtie2` block generates a SAM file that contains the mapped results. To see the location of the file, use `unwrap`.

```
unwrap(mappedFile.SAMFile)
```

Input Arguments

`pipeline` — Bioinformatics pipeline
`bioinfo.pipeline.Pipeline` object

Bioinformatics pipeline, specified as a `bioinfo.pipeline.Pipeline` object.

`tf` — Flag to include optional input port names
`false` or 0 (default) | `true` or 1

Flag to include optional input port names, specified as a numeric or logical 1 (`true`) or 0 (`false`). An optional inport port is an input port (`bioinfo.pipeline.Input`) with its `Required` property set to `false`.

Output Arguments

`names` — Names of unconnected block inputs
string scalar | string vector

Names of unconnected block inputs, returned as a string scalar or string vector. You can use these names as fields of the input structure in the run call `run(pipeline,inputStruct)`.

Version History

Introduced in R2023a

See Also

`bioinfo.pipeline.Pipeline` | **Biopipeline Designer**

bioinfo.pipeline.Output

Output port object for bioinformatics pipeline block

Description

Each output port of a `bioinfo.pipeline.Block` object is a `bioinfo.pipeline.Output` object.

Creation

Create the object using `bioinfo.pipeline.Output`.

Properties

`UniformOutput` — Flag to merge parallel outputs
`true` or 1 (default) | `false` or 0

Flag to merge parallel outputs with cell arrays, specified as a numeric or logical 1 (`true`) or 0 (`false`).

This property determines the merge strategy of the block output results after the block runs multiple times independently generating various results from each run according to the SplitDimension property.

If the value is true, the results from each block run are merged according to the output data type and concatenated along the specified split dimensions (similar to the `UniformOutput` name-value argument of `cellfun` or `arrayfun`). Additionally, the individual outputs from each run must be scalar unless the block only runs once. If there is only one block run, nonscalar outputs are allowed with `UniformOutput=true`.

If the value is false, the results from each run are wrapped in a cell array before concatenating.

Data Types: `double` | `logical`

`Required` — Flag to indicate output port is required
`true` or 1 (default) | `false` or 0

This property is read-only.

Flag to indicate if the output port is required for the block, specified as a numeric or logical 1 (`true`) or 0 (`false`).

A required output must be a field of a structure returned by the `run` method of a block. Otherwise, the block fails to run.

You can set the value as true or false when you define a block subclass. For details, see “Subclass Pipeline Block” on page 1-12.

Data Types: `double` | `logical`

Version History

Introduced in R2023a

See Also

`bioinfo.pipeline.Input` | `bioinfo.pipeline.Block`

bioinfo.pipeline.Pipeline

Pipeline object to build and run end-to-end bioinformatics analyses and workflows

Description

The `bioinfo.pipeline.Pipeline` object lets you construct and execute bioinformatics pipelines and workflows to analyze genomic data.

Creation

Create the object using `bioinfo.pipeline.Pipeline`.

Properties

`Blocks` — Blocks in pipeline
vector

Blocks in the pipeline, specified as a vector of block objects.

`BlockNames` — Names of blocks in the pipeline
string vector

Names of the blocks in the pipeline, specified as a string vector.

Object Functions

addBlock	Add blocks to pipeline
blockName	Return the names of specified blocks in pipeline
cancel	Cancel blocks in pipeline that are running in parallel
compile	Verify pipeline structure and check for warnings and errors
connect	Connect two blocks in pipeline
copy	Copy array of handle objects
deleteResults	Delete block results from pipeline
disconnect	Remove connection between ports in a pipeline
fetchResults	Wait for parallel-running block to finish and return its results
findBlock	Get block objects from bioinformatics pipeline
inputNames	Get names of unconnected block inputs from pipeline
portMap	Show connected ports between two blocks
processTable	Return information about all processes in pipeline
removeBlock	Remove blocks from pipeline
renameBlock	Rename block in pipeline
results	Get bioinformatics pipeline results
run	Run pipeline
wait	Wait for running blocks to complete

Examples

Create a Simple Pipeline to Plot Sequence Quality Data

Import the Pipeline and block objects needed for the example.

```
import bioinfo.pipeline.Pipeline
import bioinfo.pipeline.blocks.*
```

Create a pipeline.

```
qcpipeline = Pipeline;
```

Select an input FASTQ file using a `FileChooser` block.

```
fastqfile = FileChooser(which("SRR005164_1_50.fastq"));
```

Create a `SeqFilter` block.

```
sequencefilter = SeqFilter;
```

Define the filtering threshold value. Specifically, filter out sequences with a total of more than 10 low-quality bases, where a base is considered a low-quality base if its quality score is less than 20.

```
sequencefilter.Options.Threshold = [10 20];
```

Add the blocks to the pipeline.

```
addBlock(qcpipeline,[fastqfile,sequencefilter]);
```

Connect the output of the first block to the input of the second block. To do so, you need to first check the input and output port names of the corresponding blocks.

View the `Outputs` (port of the first block) and `Inputs` (port of the second block).

```
fastqfile.Outputs
```

```
ans = struct with fields:
    Files: [1×1 bioinfo.pipeline.Output]
```

```
sequencefilter.Inputs
```

```
ans = struct with fields:
    FASTQFiles: [1×1 bioinfo.pipeline.Input]
```

Connect the `Files` output port of the `fastqfile` block to the `FASTQFiles` port of `sequencefilter` block.

```
connect(qcpipeline,fastqfile,sequencefilter,["Files","FASTQFiles"]);
```

Next, create a `UserFunction` block that calls the `seqqcplot` function to plot the quality data of the filtered sequence data. In this case, `inputFile` is the required argument for the `seqqcplot` function. The required argument name can be anything as long as it is a valid variable name.

```
qcplot = UserFunction("seqqcplot",RequiredArguments="inputFile",OutputArguments="figureHandle");
```

Alternatively, you can also use dot notation to set up your `UserFunction` block.

```
qcplot = UserFunction;
qcplot.RequiredArguments = "inputFile";
qcplot.Function = "seqqcplot";
qcplot.OutputArguments = "figureHandle";
```

Add the block.

```
addBlock(qcpipeline,qcplot);
```

Check the port names of `sequencefilter` block and `qcplot` block.

```
sequencefilter.Outputs
```

```
ans = struct with fields:
    FilteredFASTQFiles: [1×1 bioinfo.pipeline.Output]
         NumFilteredIn: [1×1 bioinfo.pipeline.Output]
        NumFilteredOut: [1×1 bioinfo.pipeline.Output]
```

```
qcplot.Inputs
```

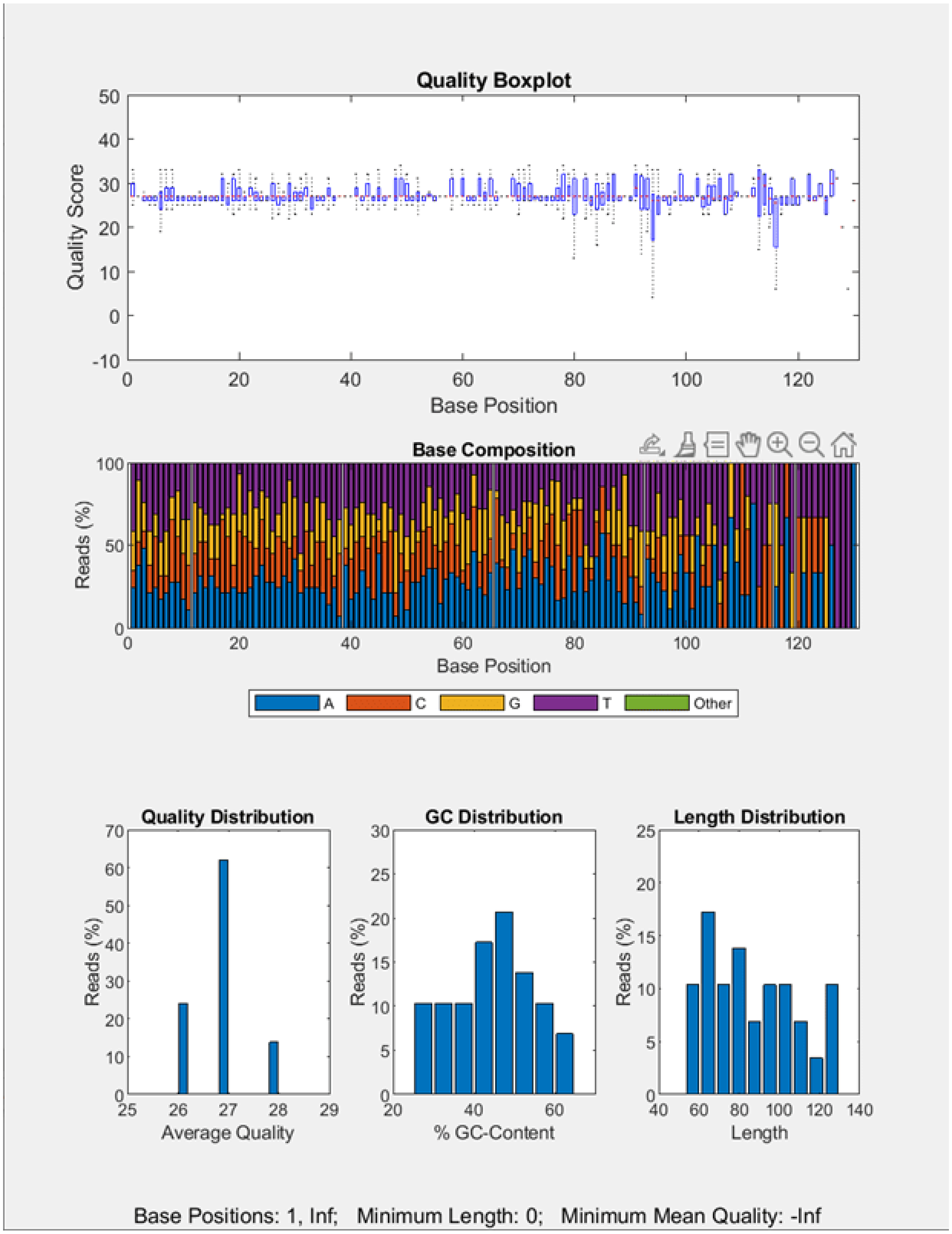
```
ans = struct with fields:
    inputFile: [1×1 bioinfo.pipeline.Input]
```

Connect the `FilteredFASTQFiles` port of the `sequencefilter` block to the `inputFile` port of the `qcplot` block.

```
connect(qcpipeline,sequencefilter,qcplot,["FilteredFASTQFiles","inputFile"]);
```

Run the pipeline to plot the sequence quality data.

```
run(qcpipeline);
```


Quality Boxplot
Quality Score
Base Position
Base Composition
Reads (%)
Base Position
A
C
G
T
Other
Quality Distribution
Reads (%)
Average Quality
GC Distribution
Reads (%)
% GC-Content
Length Distribution
Reads (%)
Length
Base Positions: 1, Inf; Minimum Length: 0; Minimum Mean Quality: -Inf

Version History

Introduced in R2023a

See Also

Biopipeline Designer | `bioinfo.pipeline.Block`

portMap

Package: `bioinfo.pipeline`

Show connected ports between two blocks

Syntax

```
portnames = portMap(pipeline,sourceBlock,targetBlock)
```

Description

`portnames = portMap(pipeline,sourceBlock,targetBlock)` returns the names of connected ports between `sourceBlock` and `targetBlock` in the `pipeline`.

Examples

Connect Blocks in Bioinformatics Pipeline

Import the Pipeline and block objects needed for the example.

```
import bioinfo.pipeline.Pipeline
import bioinfo.pipeline.blocks.*
```

Create a pipeline.

```
qcpipeline = Pipeline;
```

Select an input FASTQ file using a `FileChooser` block.

```
fastqfile = FileChooser(which("SRR005164_1_50.fastq"));
```

Create a `SeqFilter` block.

```
sequencefilter = SeqFilter;
```

Add blocks to the pipeline. Optionally, you can specify the block names.

```
addBlock(qcpipeline,[fastqfile,sequencefilter],["FF","SF"]);
```

Connect the output of the first block to the input of the second block. To do so, you need to first check the input and output port names of the corresponding blocks.

View the Outputs (port of the first block) and Inputs (port of the second block).

```
fastqfile.Outputs
```

```
ans = struct with fields:
    Files: [1×1 bioinfo.pipeline.Output]
```

```
sequencefilter.Inputs
```

```
ans = struct with fields:
    FASTQFiles: [1×1 bioinfo.pipeline.Input]
```

Connect the `Files` output port of the `fastqfile` block to the `FASTQFiles` port of `sequencefilter` block. The output is a 1-by-2 string array, indicating the names of two ports that are now connected. You can use either the block objects themselves or the block names that you have defined previously. The next command uses the block names to identify and connect these two blocks.

```
connectedports = connect(qcpipeline,"FF","SF",["Files","FASTQFiles"])
```

```
connectedports = 1×2 string
    "Files"    "FASTQFiles"
```

You can also query the names of connected ports using `portMap`.

```
portnames = portMap(qcpipeline,"FF","SF")
```

```
portnames = 1×2 string
    "Files"    "FASTQFiles"
```

Input Arguments

`pipeline` — Bioinformatics pipeline
`bioinfo.pipeline.Pipeline` object

Bioinformatics pipeline, specified as a `bioinfo.pipeline.Pipeline` object.

`sourceBlock` — Block in pipeline
`bioinfo.pipeline.Block` object | character vector | string scalar

Block in the pipeline, specified as a scalar `bioinfo.pipeline.Block` object or a character vector or string scalar that represents a block name.

`targetBlock` — Block in pipeline
`bioinfo.pipeline.Block` object | character vector | string scalar

Block in the pipeline, specified as a `bioinfo.pipeline.Block` object or a character vector or string scalar that represents a block name.

Output Arguments

`portnames` — Connected port names
string array

Connected port names, specified as an *N*-by-2 string array, where *N* is the number of connections. Each row of `portnames` is a connection. For instance, for the *n*th connection, `portnames(n,1)` is the name of an output port of a source block and `portnames(n,2)` is the name of the input port of a target block.

Version History

Introduced in R2023a

See Also

`connect` | `disconnect` | `bioinfo.pipeline.Block` | `bioinfo.pipeline.Pipeline` | **Biopipeline Designer**

removeBlock

Package: `bioinfo.pipeline`

Remove blocks from pipeline

Syntax

`removeBlock(pipeline,blocks)`

Description

`removeBlock(pipeline,blocks)` removes the specified `blocks` from the `pipeline`. Any existing connections to or from any of the removed ports are also removed.

Examples

Remove Blocks from Bioinformatics Pipeline

Import the Pipeline and block objects needed for the example.

```
import bioinfo.pipeline.Pipeline
import bioinfo.pipeline.blocks.*
```

Create a pipeline.

```
p = Pipeline;
```

Create some blocks.

```
FCB1 = FileChooser(which("ex1.bam"));
FCB2 = FileChooser(which("ex1.sam"));
FCB3 = FileChooser(which("Myco_1_1.sam"));
```

Add blocks to the pipeline.

```
addBlock(p,[FCB1,FCB2,FCB3], ["F1","F2","F3"]);
```

Remove blocks from the pipeline.

```
removeBlock(p,"F2");
p.Blocks
```

```
ans=2×1 object
  2×1 FileChooser array with properties:

    PathRoot
    Files
    Options
    Inputs
    Outputs
    ErrorHandler
```

```
removeBlock(p,[FCB1,FCB3])
p.Blocks

ans = 

  0×1 Block array with properties:

    Inputs
    Outputs
    ErrorHandler
```

Input Arguments

pipeline — Bioinformatics pipeline
bioinfo.pipeline.Pipeline object

Bioinformatics pipeline, specified as a `bioinfo.pipeline.Pipeline` object.

blocks — Pipeline blocks
bioinfo.pipeline.Block object | vector of objects | character vector | string scalar | ...

Pipeline blocks, specified as a `bioinfo.pipeline.Block` object or vector of block objects. You can also specify character vector, string scalar, string vector, or cell array of character vectors representing block names.

Version History

Introduced in R2023a

See Also

`addBlock` | `findBlock` | `renameBlock` | `bioinfo.pipeline.Block` | `bioinfo.pipeline.Pipeline` | **Biopipeline Designer**

renameBlock

Package: `bioinfo.pipeline`

Rename block in pipeline

Syntax

```
renameBlock(pipeline,block,name)
```

Description

`renameBlock(pipeline,block,name)` sets the name of the `block` to the specified `name`.

Examples

Rename Blocks in Bioinformatics Pipeline

Import the Pipeline and block objects needed for the example.

```
import bioinfo.pipeline.Pipeline
import bioinfo.pipeline.blocks.*
```

Create a pipeline.

```
p = Pipeline;
```

Create a FileChooser block.

```
FCB = bioinfo.pipeline.blocks.FileChooser(which("ex1.bam"));
```

Add the block to the pipeline and set the block name to "File".

```
addBlock(p,FCB,"File");
```

Rename the block. Use the existing block name to reference the block to be renamed.

```
renameBlock(p,"File","BamFile");
p.BlockNames
```

```
ans =
"BamFile"
```

Alternatively, you can use the block object itself.

```
renameBlock(p,FCB,"Ex1Bam");
p.BlockNames
```

```
ans =
"Ex1Bam"
```

Input Arguments

`pipeline` — Bioinformatics pipeline
`bioinfo.pipeline.Pipeline` object

Bioinformatics pipeline, specified as a `bioinfo.pipeline.Pipeline` object.

`block` — Pipeline block
`bioinfo.pipeline.Block` object | character vector | string scalar

Pipeline block, specified as a `bioinfo.pipeline.Block` object, character vector, or string scalar, representing the current block name.

`name` — New block name
character vector | string scalar

New block name, specified as a character vector or string scalar. The name must be a valid variable name and must be unique within a pipeline.

Version History

Introduced in R2023a

See Also

`removeBlock` | `addBlock` | `findBlock` | `bioinfo.pipeline.Block` | `bioinfo.pipeline.Pipeline` | **Biopipeline Designer**

results

Package: `bioinfo.pipeline`

Get bioinformatics pipeline results

Syntax

```
blockResults = results(pipeline,block)
```

Description

`blockResults = results(pipeline,block)` returns the results of `block` in the `pipeline`.

Tip Use `fetchResults` instead of `results` if you are running in parallel. `fetchResults` waits for the block to complete before returning its results.

Examples

Get Block Results from Bioinformatics Pipeline

Import the pipeline and block objects needed for the example.

```
import bioinfo.pipeline.Pipeline
import bioinfo.pipeline.blocks.*
```

Create a pipeline.

```
P = Pipeline;
```

Create the FileChooser and SamSort blocks.

```
FCB = FileChooser(which("ex1.sam"));
SSB = SamSort;
```

Add blocks to the pipeline and connect them.

```
addBlock(P,[FCB,SSB]);
connect(P,FCB,SSB,["Files","SAMFile"]);
```

Run the pipeline.

```
run(P);
```

The outputs of the FileChooser and SamSort blocks are files, that are saved to your file system.

```
fcbResults = results(P,FCB)
```

```
fcbResults = struct with fields:
    Files: [1×1 bioinfo.pipeline.datatypes.File]
```

```
ssbResults = results(P,SSB)

ssbResults = struct with fields:
    SortedSAMFile: [1×1 bioinfo.pipeline.datatypes.File]
```

Tip: Use the `unwrap` method to see the location of the output file. For example, `unwrap(ssbResults.SortedSAMFiles)` shows the location of the sorted SAM file.

Input Arguments

`pipeline` — Bioinformatics pipeline
`bioinfo.pipeline.Pipeline` object

Bioinformatics pipeline, specified as a `bioinfo.pipeline.Pipeline` object.

`block` — Block in pipeline
`bioinfo.pipeline.Block` object | character vector | string scalar

Block in the pipeline, specified as a scalar `bioinfo.pipeline.Block` object, character vector, or string scalar as the block name. To get the list of block names, enter `pipeline.“BlockNames” on page 1-0` at the command line.

Output Arguments

`blockResults` — Block results
structure | `bioinfo.pipeline.datatypes.Incomplete`

Block results, returned as a structure or `bioinfo.pipeline.datatypes.Incomplete` object. The `Incomplete` object is returned if the block results are not yet computed or available.

If it is a structure, the field names are the output port names of the block, and the field values are the output values. If you have not run the pipeline or the results are not available yet, each output value has the default value of `Incomplete`, which is a `bioinfo.pipeline.Incomplete` object.

Data Types: `struct`

Version History

Introduced in R2023a

See Also

`fetchResults` | `run` | `bioinfo.pipeline.Block` | `bioinfo.pipeline.Pipeline` | **Biopipeline Designer**

run

Package: `bioinfo.pipeline`

Run block object

Syntax

```
outStruct = run(blockObj,inStruct)
```

Description

`outStruct = run(blockObj,inStruct)` runs a block object `blockObj` using an input structure `inStruct` and returns the block outputs in an output structure `outStruct`.

Tip For most cases, use the `run` method of a block instead of `eval` because the `run` method performs additional error checks and ensures that the block inputs and outputs are satisfied on page 1-130 and the `eval` method accepts and returns a scalar structure, which is a requirement to run the block as part of a pipeline. In addition, when you run a pipeline, it calls the `run` method of each block.

Examples

Run Bioinformatics Block

Create a SamSort block.

```
SSBlock = bioinfo.pipeline.blocks.SamSort;
```

The `emptyInputs` method of the block creates an input structure with the field name equivalent to the name of the input port of the block. The `run` method of a block requires such a structure.

```
inputStruct = emptyInputs(SSBlock)
```

```
inputStruct = struct with fields:
    SAMFile: []
```

Set the value of the `SAMFile` field to a sample SAM file.

```
inputStruct.SAMFile = which("ex1.sam");
```

Call the `run` method of the block with the input structure. The output structure contains the field name that is the same as the output port name of the block and its value is the block output (which is a sorted SAM file).

```
outStruct = run(SSBlock,inputStruct)
```

```
outStruct = struct with fields:
    SortedSAMFile: [1×1 bioinfo.pipeline.datatypes.File]
```

Use `unwrap` to check the location of the generated sorted SAM file.

```
unwrap(outStruct.SortedSAMFile)
```

Input Arguments

`blockObj` — Block object
`bioinfo.pipeline.Block` object | ...

Block object, specified as a scalar `bioinfo.pipeline.Block` object.

`inStruct` — Block input
structure

Block input, specified as a structure. The field names of the structure must be the names of the input ports of the block.

Output Arguments

`outStruct` — Block output
structure

Block output, returned as a structure. The field names of the structure are the names of the output ports of the block, and the field values are the output values of the block.

Version History

Introduced in R2023a

See Also

`run` | `eval` | `compile` | `bioinfo.pipeline.Block` | `bioinfo.pipeline.Pipeline`

setFlag

Class: `BioMap`

Set read sequence flags for `BioMap` object

Syntax

```
NewObj = setFlag(BioObj, FlagValues)
NewObj = setFlag(BioObj, FlagValues, Subset)
```

Description

`NewObj = setFlag(BioObj, FlagValues)` returns `NewObj`, a new `BioMap` object, constructed from `BioObj`, an existing `BioMap` object, with the `Flag` property set to `FlagValues`, a vector of nonnegative integers indicating the bit-wise information that specifies the status of each of the 11 flags described by the SAM format specification.

`NewObj = setFlag(BioObj, FlagValues, Subset)` sets the `Flag` property of the elements specified by `Subset` to `FlagValues`.

Input Arguments

`BioObj` — Object of BioMap class
BioMap object

Object of the BioMap class, specified as a `BioMap` object

Note If *`BioObj`* was constructed from a `BioIndexedFile` object, you cannot set its `Flag` property.

`FlagValues` — SAM flag values
vector of nonnegative integers

SAM flag values, specified as vector of nonnegative integers. Each integer corresponds to one read sequence and indicates the bit-wise information that specifies the status of each of the 11 flags described by the SAM format specification. These flags describe different sequencing and alignment aspects of a read sequence.

`Subset` — Subset of elements in BioObj
vector of positive integers | logical vector | cell array of character vectors

Subset of elements in `BioObj`, specified as one of the following:

- vector of positive integers
- logical vector
- cell array of character vectors containing valid sequence headers

Note A one-to-one relationship must exist between the number and order of elements in `FlagValues` and `Subset`. If you use a cell array of headers to specify `Subset`, be aware that a repeated header specifies all elements with that header.

Output Arguments

`NewObj` — Object of BioMap class
BioMap object

Object of the BioMap class, returned as a `BioMap` object.

Examples

Update SAM flag values of BioMap object

Create a BioMap object from a SAM file. Set `'InMemory'` to true to allow modifying the object properties.

```
bm = BioMap('ex2.sam','InMemory',true);
```

Check the SAM flag value of the 5th read sequence.

```
bm.Flag(5)
```

```
ans = uint16
    137
```

Update the flag value.

```
bm2 = setFlag(bm,75,5);
bm2.Flag(5)
```

```
ans = uint16
    75
```

Update the flag values of the first and third elements.

```
bm3 = setFlag(bm,[0 0],[1 3]);
```

```
bm3.Flag(1)
```

```
ans = uint16
    0
```

```
bm3.Flag(3)
```

```
ans = uint16
    0
```

Alternatives

An alternative to using the `setFlag` method to update an existing object is to use dot indexing with the `Flag` property:

```
BioObj.Flag(Indices) = NewFlag
```

In the previous syntax, *`Indices`* is a vector of positive integers or a logical vector. *`Indices`* cannot be a cell array of character vectors containing sequence headers. *`NewFlag`* is a vector of nonnegative integers indicating the bit-wise information that specifies the status of each of the 11 flags described by the SAM format specification. Each integer corresponds to one read sequence in a `BioMap` object. *`Indices`* and *`NewFlag`* must have the same number and order of elements.

See Also

`BioMap` | `getFlag`

Topics

"Manage Sequence Read Data in Objects"

External Websites

Sequence Read Archive
SAM format specification

setMappingQuality

Class: `BioMap`

Set sequence mapping quality scores for `BioMap` object

Syntax

```
NewObj = setMappingQuality(BioObj, MappingQuality)
NewObj = setMappingQuality(BioObj, MappingQuality, Subset)
```

Description

`NewObj = setMappingQuality(BioObj, MappingQuality)` returns *`NewObj`*, a new `BioMap` object, constructed from *`BioObj`*, an existing `BioMap` object, with the `MappingQuality` property set to *`MappingQuality`*, a vector of integers specifying the mapping quality scores for read sequences.

`NewObj = setMappingQuality(BioObj, MappingQuality, Subset)` returns *`NewObj`*, a new `BioMap` object, constructed from *`BioObj`*, an existing `BioMap` object, with the `MappingQuality` property of a subset of the elements set to *`MappingQuality`*, a vector of integers specifying the mapping quality scores for read sequences. It sets the mapping quality scores for only the object elements specified by *`Subset`*.

Input Arguments

`BioObj`

Object of the `BioMap` class.

Note If *`BioObj`* was constructed from a `BioIndexedFile` object, you cannot set its `MappingQuality` property.

Default:

`MappingQuality`

Vector of integers specifying the mapping quality scores for read sequences.

Default:

`Subset`

One of the following to specify a subset of the elements in *`BioObj`*:

- Vector of positive integers
- Logical vector
- Cell array of character vectors containing valid sequence headers

---

Note A one-to-one relationship must exist between the number and order of elements in *`MappingQuality`* and *`Subset`*. If you use a cell array of headers to specify *`Subset`*, be aware that a repeated header specifies all elements with that header.

Default:

Output Arguments

`NewObj`

Object of the `BioMap` class.

Examples

Construct a `BioMap` object, and then set a subset of the mapping quality scores:

```
% Construct a BioMap object from a SAM file
BMObj1 = BioMap('ex1.sam');
% Set the Mapping Quality property of the second element to a new
% value
BMObj1 = setMappingQuality(BMObj1, 74, 2);
```

Tips

To update mapping quality scores in an existing `BioMap` object, use the same object as the input *`BioObj`* and the output *`NewObj`*.

Alternatives

An alternative to using the `setMappingQuality` method to update an existing object is to use dot indexing with the `MappingQuality` property:

```
BioObj.MappingQuality(Indices) = NewMappingQuality
```

In the previous syntax, *`Indices`* is a vector of positive integers or a logical vector. *`Indices`* cannot be a cell array of character vectors containing sequence headers. *`NewMappingQuality`* is a vector of integers specifying the mapping quality scores for read sequences. *`Indices`* and *`NewQuality`* must have the same number and order of elements.

See Also

`BioMap` | `getMappingQuality`

Topics
"Manage Sequence Read Data in Objects"

External Websites
Sequence Read Archive
SAM format specification

setMatePosition

Set mate positions of read sequences in `BioMap` object

Syntax

```
NewObj = setMatePosition(BioObj,MatePos)
NewObj = setMatePosition(BioObj,MatePos,Subset)
```

Description

`NewObj = setMatePosition(BioObj,MatePos)` returns a new `BioMap` object constructed from `BioObj` with the `MatePosition` property set to `MatePos`.

To update mate positions in an existing `BioMap` object, use the same object as the input `BioObj` and the output `NewObj`. For example,

```
BioObj = setMatePosition(BioObj,MatePos)
```

Note To set the mate position, when constructing `BioObj` you must set the `InMemory` name-value argument to `true`. See "Manage Sequence Read Data in Objects".

`NewObj = setMatePosition(BioObj,MatePos,Subset)` sets the mate positions for only the object elements specified by `Subset`.

Examples

Set Mate Position

Construct a `BioMap` object, and then set a subset of the sequence mate position values. Construct a `BioMap` object from a `SAM` file setting the `InMemory` name-value argument to `true`, and determine the header for the second element.

```
BioObj = BioMap('ex1.sam',InMemory=true);
hdr = BioObj.Header(2)
```

```
hdr = 1x1 cell array
    {'EAS54_65:7:152:368:113'}
```

Set the `MatePosition` property of the second element to a new value of 5.

```
NewObj = setMatePosition(BioObj,5,hdr)
```

```
NewObj = 
  BioMap with properties:

    SequenceDictionary: {'seq1'}
             Reference: {1501x1 cell}
             Signature: {1501x1 cell}
             Start: [1501x1 uint32]
    MappingQuality: [1501x1 uint8]
              Flag: [1501x1 uint16]
      MatePosition: [1501x1 uint32]
           Quality: {1501x1 cell}
          Sequence: {1501x1 cell}
            Header: {1501x1 cell}
             NSeqs: 1501
              Name: ''
```

Set the `MatePosition` properties of the first and third elements in the new object to 6 and 7 respectively.

```
NewObj = setMatePosition(NewObj, [6 7], [1 3]);
```

Set the `MatePosition` property of all elements in the new object to zero.

```
y = zeros(1,NewObj.NSeqs);
NewObj = setMatePosition(NewObj,y);
```

Input Arguments

`BioObj` — Object for read sequences
`BioMap` object

Object for read sequences, specified as a `BioMap` object. Construct `BioObj` using `BioMap`.

Note If `BioObj` was constructed from a `BioIndexedFile` object, you cannot set its `MatePosition` property.

`MatePos` — Mate positions of the read sequences with respect to the position numbers in the reference sequence
vector of nonnegative integers

Mate positions of the read sequences with respect to the position numbers in the reference sequence, specified as a vector of nonnegative integers.

Example: `[3 12]`

Data Types: `single` | `double`

`Subset` — Subset of elements in `BioObj`
vector of positive integers | logical vector | cell array of character vectors containing valid sequence headers | string vector containing valid sequence headers

Subset of the elements in `BioObj`, specified as one of the following:

- Vector of positive integers
- Logical vector
- Cell array of character vectors containing valid sequence headers
- String vector containing valid sequence headers

Note A one-to-one relationship must exist between the number and order of elements in `MatePos` and `Subset`. If you use a cell array of headers to specify `Subset`, a repeated header specifies all elements with that header.

Example: `[5 26]`

Data Types: `single` | `double` | `logical` | `char` | `string` | `cell`

Alternative Functionality

An alternative to using `setMatePosition` is to use dot indexing with the `MatePosition` property:

```
BioObj.MatePosition(Indices) = NewMatePos
```

In this syntax, `Indices` is a vector of positive integers or a logical vector. `Indices` cannot be a cell array of character vectors containing sequence headers. `NewMatePos` is a vector of integers specifying the mate positions of the read sequences with respect to the position numbers in the reference sequence. `Indices` and `NewMatePos` must have the same number and order of elements.

Version History

Introduced in R2010b

See Also

`BioMap` | `getMatePosition`

Topics

"Manage Sequence Read Data in Objects"

External Websites

Sequence Read Archive
SAM format specification

setReference

Class: `BioMap`

Set name of reference sequence for `BioMap` object

Syntax

```
newObj = setReference(Obj,RefSeqName)
newObj = setReference(Obj,RefSeqName,Indices)
```

Description

`newObj = setReference(Obj,RefSeqName)` returns a new `BioMap` object `newObj`, constructed from an existing object `Obj`, with the `Reference` property set to `RefSeqName`, a character vector or string specifying the name of the reference sequence.

`newObj = setReference(Obj,RefSeqName,Indices)` sets the `Reference` property of the elements indexed by `Indices` to `RefSeqName`.

Input Arguments

`Obj` — Object of BioMap class
BioMap object

Object of the BioMap class, specified as a `BioMap` object.

`RefSeqName` — Name of reference sequence
character vector | string | string vector | cell array of character vectors

Name of reference sequence, specified as a character vector, string, string vector, or cell array of character vectors.

`Indices` — Indices to elements of the input object
positive integer | vector of positive integers | logical array | character vector | string | string vector | cell array of character vectors

Indices to the elements of the input object, specified as a positive integer, vector of positive integers, logical array, character vector, string, string vector, or cell array of character vectors. For character vectors or strings, they must correspond to the valid `Header` values of the input object `Obj`.

Output Arguments

`newObj` — Object of BioMap class
BioMap object

Object of the BioMap class, returned as a `BioMap` object.

Examples

Change reference sequence of BioMap object

Create a BioMap object from a SAM file. Set `'InMemory'` to true to allow modifying the object properties.

```
bm = BioMap('ex2.sam','InMemory',true);
```

Check the reference sequence of the 5th read sequence.

```
bm.Reference(5)
```

```
ans = 1x1 cell array
    {'seq1'}
```

Change the reference sequence to another one in the sequence dictionary.

```
bm2 = setReference(bm,{'seq2'},5);
bm2.Reference(5)
```

```
ans = 1x1 cell array
    {'seq2'}
```

See Also

`BioMap` | `getReference`

Topics

"Manage Sequence Read Data in Objects"

External Websites

Sequence Read Archive
SAM format specification

setSignature

Class: `BioMap`

Set signature (alignment information) for `BioMap` object

Syntax

```
NewObj = setSignature(BioObj, Signature)
NewObj = setSignature(BioObj, Signature, Subset)
```

Description

`NewObj = setSignature(BioObj, Signature)` returns *`NewObj`*, a new `BioMap` object, constructed from *`BioObj`*, an existing `BioMap` object, with the `Signature` property set to *`Signature`*, a cell array of CIGAR-formatted character vectors, each representing how a read sequence aligns to the reference sequence.

`NewObj = setSignature(BioObj, Signature, Subset)` returns *`NewObj`*, a new `BioMap` object, constructed from *`BioObj`*, an existing `BioMap` object, with the `Signature` property of a subset of the elements set to *`Signature`*, a cell array of CIGAR-formatted character vectors, each representing how read sequences, specified by *`Subset`*, align to the reference sequence. It sets the signature for only the object elements specified by *`Subset`*.

Input Arguments

`BioObj`

Object of the `BioMap` class.

Note If *`BioObj`* was constructed from a `BioIndexedFile` object, you cannot set its `Signature` property.

Default:

`Signature`

Cell array of CIGAR-formatted character vectors, each representing how a read sequence aligns to the reference sequence. Signature can be empty.

Default:

`Subset`

One of the following to specify a subset of the elements in *`BioObj`*:

- Vector of positive integers
- Logical vector

- Cell array of character vectors containing valid sequence headers

---

Note A one-to-one relationship must exist between the number and order of elements in *`Signature`* and *`Subset`*. If you use a cell array of headers to specify *`Subset`*, be aware that a repeated header specifies all elements with that header.

Default:

Output Arguments

`NewObj`

Object of the `BioMap` class.

Examples

Construct a `BioMap` object, and then set a subset of the signatures:

```
% Construct a BioMap object from a SAM file
BMObj1 = BioMap('ex1.sam');
% Set the Signature property of the second element to a new value
BMObj1 = setSignature(BMObj1,  {'36M'}, 2);
```

Tips

- To update signatures in an existing `BioMap` object, use the same object as the input *`BioObj`* and the output *`NewObj`*.
- If you modify sequences or start positions in an object, you may need to use the `setSignature` method to modify the `Signature` property of modified sequences accordingly.

Alternatives

An alternative to using the `setSignature` method to update an existing object is to use dot indexing with the `Signature` property:

```
BioObj.Signature(Indices) = NewSignature
```

In the previous syntax, *`Indices`* is a vector of positive integers or a logical vector. *`Indices`* cannot be a cell array of character vectors containing sequence headers. *`NewSignature`* is a character vector or a cell array of CIGAR-formatted character vectors, each representing how a read sequence aligns to the reference sequence. Signature can be empty. *`Indices`* and *`NewSignature`* must have the same number and order of elements.

See Also

`BioMap` | `getSignature` | `setSequence` | `setStart` | `getAlignment`

Topics

"Manage Sequence Read Data in Objects"

External Websites
Sequence Read Archive
SAM format specification

setStart

Class: `BioMap`

Set start positions of aligned read sequences in `BioMap` object

Syntax

```
NewObj = setStart(BioObj, Start)
NewObj = setStart(BioObj, Start, Subset)
```

Description

`NewObj = setStart(BioObj, Start)` returns *`NewObj`*, a new `BioMap` object, constructed from *`BioObj`*, an existing `BioMap` object, with the `Start` property set to *`Start`*, a vector of positive integers specifying the start positions of the aligned read sequences with respect to the position numbers in the reference sequence. Modifying the `Start` property shifts the aligned sequences.

`NewObj = setStart(BioObj, Start, Subset)` returns *`NewObj`*, a new `BioMap` object, constructed from *`BioObj`*, an existing `BioMap` object, with the `Start` property of a subset of the elements set to *`Start`*, a vector of positive integers specifying the start positions of the aligned read sequences with respect to the position numbers in the reference sequence. It sets the start positions for only the object elements specified by *`Subset`*.

Input Arguments

`BioObj`

Object of the `BioMap` class.

Note If *`BioObj`* was constructed from a `BioIndexedFile` object, you cannot set its `Start` property.

Default:

`Start`

Vector of positive integers specifying the start positions of the aligned read sequences with respect to the position numbers in the reference sequence.

Default:

`Subset`

One of the following to specify a subset of the elements in *`BioObj`*:

- Vector of positive integers
- Logical vector
- Cell array of character vectors containing valid sequence headers

Note A one-to-one relationship must exist between the number and order of elements in *`Start`* and *`Subset`*. If you use a cell array of headers to specify *`Subset`*, be aware that a repeated header specifies all elements with that header.

Default:

Output Arguments

`NewObj`

Object of the `BioMap` class.

Examples

Construct a `BioMap` object, and then set a subset of the sequence start values:

```
% Construct a BioMap object from a SAM file
BMObj1 = BioMap('ex1.sam');
% Set the Start property of the second element to a new value
BMObj1 = setStart(BMObj1, 5, 2);
```

Tips

- To update start positions in an existing `BioMap` object, use the same object as the input *`BioObj`* and the output *`NewObj`*.
- If you modify sequences or signatures in an object, you may need to use the `setStart` method to modify the `Start` property to shift the alignment of modified sequences accordingly.

Alternatives

An alternative to using the `setStart` method to update an existing object is to use dot indexing with the `Start` property:

```
BioObj.Start(Indices) = NewStart
```

In the previous syntax, *`Indices`* is a vector of positive integers or a logical vector. *`Indices`* cannot be a cell array of character vectors containing sequence headers. *`NewStart`* is a vector of integers specifying the start positions of the aligned read sequences with respect to the position numbers in the reference sequence. *`Indices`* and *`NewStart`* must have the same number and order of elements.

See Also

`BioMap` | `getStart` | `setSequence` | `setSignature`

Topics
"Manage Sequence Read Data in Objects"

External Websites
Sequence Read Archive
SAM format specification

BioMap class

Superclasses: `BioRead`

Contain sequence, quality, alignment, and mapping data

Description

The `BioMap` class contains data from short-read sequences, including sequence headers, read sequences, quality scores for the sequences, and data about how each sequence aligns to a given reference. This data is typically obtained from a high-throughput sequencing instrument.

Construct a `BioMap` object from short-read sequence data. Each element in the object has a sequence, header, quality score, and alignment/mapping information associated with it. Use the object properties and methods to explore, access, filter, and manipulate all or a subset of the data, before analyzing or viewing the data.

Construction

`BioMapobj = BioMap` constructs `BioMapobj`, which is an empty `BioMap` object.

`BioMapobj = BioMap(File)` constructs `BioMapobj`, a `BioMap` object, from `File`, a SAM- or BAM-formatted file whose reads are ordered by start position in the reference sequence. The data remains in the source file, and the `BioMap` object accesses it using one or two auxiliary index files. For a SAM-formatted file, MATLAB uses or creates one index file that must have the same name as the source file, but with an `.idx` extension. For a BAM-formatted file, MATLAB uses or creates two index files that must have the same name as the source file, but with `*.bai` and `*.linearindex` extensions. If the index files are not found in the same folder as the source file, the `BioMap` constructor function creates the index files in that folder.

When you pass in an unordered BAM-formatted file, the constructor automatically orders the file and writes the data to an ordered file using the same base name and extension with an added character vector “.ordered” before the extension. The new file is indexed and used to instantiate the new `BioMap` object.

Note Because the data remains in the source file and is accessed using the index files:

- Do not delete the source file (SAM or BAM).
- Do not delete the index files (*`.idx`,*`.bai`, or *`.linearindex`).
- You cannot modify `BioMapobj` properties.

Tip To determine the number of reference sequences included in your source file, use the `saminfo` or `baminfo` function. Use SAMtools to check if the reads in your source file are ordered by position in the reference sequence, and also to reorder them, if needed.

`BioMapobj = BioMap(Struct)` constructs `BioMapobj`, a `BioMap` object, from `Struct`, a MATLAB structure containing sequence and alignment information, such as returned by the `samread` or bamread function. The data from Struct remains in memory, which lets you modify the BioMapobj properties.

BioMapobj = BioMap(___ ,'Name',Value) constructs the BioMap object using any of previous input arguments and additional options, specified as name-value pair arguments as follows.

BioMapobj = BioMap(___ ,'SelectReference',SelectRefValue) selects one or more references when the source data contains sequences mapped to more than one reference. By default, the constructor includes all of the references in the header dictionary of the source file. When the header dictionary is not available, the constructor defaults to including all reference names found in the source data. SelectRefValue is a character vector, string, string vector, or cell array of character vectors. By using this option, you can prevent the BioMap constructor from creating auxiliary index files for references that you will not use in your analysis. If any reads mapped to selected references are paired and BioMapobj is written to a file, the reference sequences of the mates are also included in the file header.

BioMapobj = BioMap(File,'InMemory',InMemoryValue) specifies whether to place the data in memory or leave the data in the source file. Leaving the data in the source file and accessing via an index file is more memory efficient, but does not let you modify properties of BioMapobj. Choices are true or false (default). If the first input argument is not a file name, then this name-value pair argument is ignored, and the data is automatically placed in memory.

Tip Set the 'InMemory' name-value pair argument to true if you want to modify the properties of BioMapobj.

BioMapobj = BioMap(___ ,'IndexDir',IndexDirValue) specifies the path to the folder where the index files (*.idx,*.bai, or *.linearindex) either exist or will be created.

Tip Use the 'IndexDir' name-value pair argument if you do not have write access to the folder where the source file is located.

BioMapobj = BioMap(___ ,'Sequence',SequenceValue) constructs BioMapobj, a BioMap object, from SequenceValue that contains he letter representations of nucleotide sequences. This name-value pair works only if the data is read into memory.

BioMapobj = BioMap(___ ,'Header',HeaderValue) constructs BioMapobj, a BioMap object, from HeaderValue that contains header text for nucleotide sequences. This name-value pair works only if the data is read into memory.

BioMapobj = BioMap(___ ,'Quality',QualityValue) constructs BioMapobj, a BioMap object, from QualityValue that contains the ASCII representation of per-base quality scores for nucleotide sequences. This name-value pair works only if the data is read into memory.

BioMapobj = BioMap(___ ,'Reference',ReferenceValue) constructs BioMapobj, a BioMap object, and sets the Reference property to ReferenceValue that contains the names of the reference sequences. This name-value pair works only if the data is read into memory.

BioMapobj = BioMap(___ ,'Signature',SignatureValue) constructs BioMapobj, a BioMap object, from SignatureValue that contains information describing the alignment of each read sequence with the reference sequence. This name-value pair works only if the data is read into memory.

`BioMapobj = BioMap( ___ ,'Start',StartValue)` constructs `BioMapobj`, a `BioMap` object, from `StartValue`, a vector of positive integers specifying the position in the reference sequence where the alignment of each read sequence starts. This name-value pair works only if the data is read into memory.

`BioMapobj = BioMap( ___ ,'Flag',FlagValue)` constructs `BioMapobj`, a `BioMap` object, from `FlagValue`, a vector of positive integers indicating the bit-wise information for the status of the 11 flags specified by the SAM format specification. These flags describe different sequencing and alignment aspects of the read sequences. This name-value pair works only if the data is read into memory.

`BioMapobj = BioMap( ___ ,'MappingQuality',MappingQualityValue)` constructs *`BioMapobj`*, a `BioMap` object, from `MappingQualityValue`, a vector of positive integers specifying the mapping quality for each read sequence. This name-value pair works only if the data is read into memory.

`BioMapobj = BioMap( ___ ,'MatePosition',MatePositionValue)` constructs `BioMapobj`, a `BioMap` object, from `MatePositionValue`, a vector of nonnegative integers specifying the mate position for each read sequence. This name-value pair works only if the data is read into memory.

Input Arguments

`File`

Character vector or string specifying a SAM- or BAM-formatted file that contains only one reference sequence and whose reads are ordered by start position in the reference sequence.

`Struct`

MATLAB structure containing sequence and alignment information, such as returned by the `samread` or `bamread` function. The structure must have a one-based start position.

Default:

`SelectRefValue`

Character vector, string, string vector, or cell array of character vectors specifying the name of the reference sequences in `File` or `Struct`. Use `saminfo` or `baminfo` to see a complete list of reference sequences in `File`.

`InMemoryValue`

Logical specifying whether to place the data in memory or leave the data in the source file. Leaving the data in the source file and accessing it via an index file is more memory efficient, but does not let you modify properties of the `BioMap` object. If the first input argument is not a file name, then this name-value pair argument is ignored, and the data is automatically placed in memory.

Default: `false`

`IndexDirValue`

Character vector or string specifying the path to the folder where the index file either exists or will be created.

Default: Folder where `File` is located

`SequenceValue`

String vector or cell array of character vectors containing the letter representations of nucleotide sequences. This information populates the `BioMap` object's `Sequence` property. The `samread` and `bamread` functions return this information in the `Sequence` field of the output structure.

`QualityValue`

String vector or cell array of character vectors containing the ASCII representation of per-base quality scores for nucleotide sequences. This information populates the `BioMap` object's `Quality` property. The `samread` and `bamread` functions return this information in the `Quality` field of the output structure.

`HeaderValue`

String vector or cell array of character vectors containing header text for nucleotide sequences. This information populates the `BioMap` object's `Header` property. The `samread` and `bamread` functions return this information in the `QueryName` field of the return structure.

`NameValue`

Character vector or string describing the `BioMap` object. This information populates the object's `Name` property.

Default: `' '`, an empty character vector

`ReferenceValue`

String vector or cell array of character vectors containing the names of the reference sequences. This information populates the object's `Reference` property. The `samread` function returns this information in the `ReferenceName` field of the `SAMStruct` output argument. The `bamread` function returns this information in the `Reference` field of the `HeaderStruct` output structure.

Default:

`SignatureValue`

String vector or cell array of character vectors containing information describing the alignment of each read sequence with the reference sequence. The `samread` and `bamread` functions return this information in the `CigarString` field of the return structure. This information populates the object's `Signature` property.

Default:

`StartValue`

Vector of positive integers specifying the position in the reference sequence where the alignment of each read sequence starts. This information populates the object's `Start` property. The `samread` and `bamread` functions return this information in the `Position` field of the output structure.

Default:

`FlagValue`

Vector of positive integers indicating the bit-wise information for the status of the 11 flags specified by the SAM format specification. These flags describe different sequencing and alignment aspects of the read sequences. This information populates the object's `Flag` property. The `samread` and `bamread` functions return this information in the `Flag` field of the output structure.

Default:

`MappingQualityValue`

Vector of positive integers specifying the mapping quality for each read sequence. This information populates the object's `MappingQuality` property. The `samread` and `bamread` functions return this information in the `MappingQuality` field of the output structure.

Default:

`MatePositionValue`

Vector of nonnegative integers specifying the mate position for each read sequence. This information populates the object's `MatePosition` property. The `samread` and `bamread` functions return this information in the `MatePosition` field of the output structure.

Default:

Properties

`Flag`

Flags associated with all read sequences represented in the `BioMap` object.

Vector of positive integers such that there is an integer for each read sequence in the object. Each integer indicates the bit-wise information that specifies the status of the 11 flags described by the SAM format specification. These flags describe different sequencing and alignment aspects of a read sequence. A one-to-one relationship exists between the number and order of elements in `Flag` and `Sequence`, unless `Flag` is an empty vector.

`Header`

Headers associated with all read sequences represented in the `BioMap` object.

Cell array of character vectors, such that there is a header for each read sequence in the object. Headers can be empty. A one-to-one relationship exists between the number and order of elements in `Header` and `Sequence`, unless `Header` is an empty cell array.

`MatePosition`

Positions of the mates for all read sequences represented in the `BioMap` object.

Vector of nonnegative integers such that there is an integer for each read sequence in the object. Each integer indicates the position of the corresponding mate sequence, relative to the reference sequence. A one-to-one relationship exists between the number and order of elements in `MatePosition` and `Sequence`, unless `MatePosition` is an empty vector.

Not all values in the `MatePosition` vector represent valid mate positions, for example, mates that map to a different reference sequence or mates that do not map. To determine if a mate position is valid, use the `filterByFlag` method with the `'pairedInMap'` flag.

`MappingQuality`

Mapping quality scores associated with all read sequences represented in the `BioMap` object.

Vector of integers, such that there is a mapping quality score for each read sequence in the object. A one-to-one relationship exists between the number and order of elements in `MappingQuality` and `Sequence`, unless `MappingQuality` is an empty vector.

`Name`

Description of the `BioMap` object.

Character vector describing the `BioMap` object.

Default: `' '`, an empty character vector

`NSeqs`

Number of sequences in the `BioMap` object.

This information is read-only.

`Quality`

Per-base quality scores associated with all read sequences represented in the `BioMap` object.

Cell array of character vectors, such that there is a quality for each read sequence in the object. Each quality is an ASCII representation of per-base quality scores for a read sequence. Quality can be an empty character vector. A one-to-one relationship exists between the number and order of elements in `Quality` and `Sequence`, unless `Quality` is an empty cell array.

`Reference`

Reference sequences in the `BioMap` object.

`BioMapobj.NSeqs`-by-1 cell array of character vectors specifying the names of the reference sequences.

The reference sequences are the sequences against which the read sequences are aligned.

`Sequence`

Read sequences in the `BioMap` object.

Cell array of character vectors containing the letter representations of the read sequences.

`SequenceDictionary`

Cell array of character vectors that catalogs the names of the references available in the `BioMap` object.

This information is read-only.

`Signature`

Alignment information associated with all read sequences represented in the `BioMap` object.

Cell array of CIGAR–formatted character vectors, such that there is alignment information for each read sequence in the object. Each character vector represents how a read sequence aligns to the reference sequence. Signatures can be empty character vectors. A one-to-one relationship exists between the number and order of elements in `Signature` and `Sequence`, unless `Signature` is an empty cell array.

`Start`

Start positions of all aligned read sequences represented in the `BioMap` object.

Vector of integers, such that there is a start position for each read sequence in the object. Each integer specifies the start position of the aligned read sequence with respect to the position numbers in the reference sequence. A one-to-one relationship exists between the number and order of elements in `Start` and `Sequence`, unless `Start` is an empty vector.

Methods

getStop	Compute stop positions of aligned read sequences from `BioMap` object
filterByFlag	Filter sequence reads by SAM flag
getAlignment	Construct alignment represented in `BioMap` object
getBaseCoverage	Return base-by-base alignment coverage of reference sequence in `BioMap` object
getCompactAlignment	Construct compact alignment represented in `BioMap` object
getCounts	Return count of read sequences aligned to reference sequence in `BioMap` object
getFlag	Retrieve read sequence flags from `BioMap` object
getIndex	Return indices of read sequences aligned to reference sequence in `BioMap` object
getInfo	Retrieve information for single element of `BioMap` object
getMappingQuality	Retrieve sequence mapping quality scores from `BioMap` object
getReference	Retrieve reference sequence from `BioMap` object
getSignature	Retrieve signature (alignment information) from `BioMap` object
getStart	Retrieve start positions of aligned read sequences from `BioMap` object
getSummary	Print summary of `BioMap` object
setFlag	Set read sequence flags for `BioMap` object
setMappingQuality	Set sequence mapping quality scores for `BioMap` object
setReference	Set name of reference sequence for `BioMap` object
setSignature	Set signature (alignment information) for `BioMap` object
setStart	Set start positions of aligned read sequences in `BioMap` object

Inherited Methods

combine	Combine two objects
get	Retrieve property of object
getHeader	Retrieve sequence headers from object
getQuality	Retrieve sequence quality information from object
getSequence	Retrieve sequences from object
getSubsequence	Retrieve partial sequences from object
getSubset	Retrieve subset of elements from object
set	Set property of object
setHeader	Update header information of reads
setQuality	Update quality information
setSequence	Update read sequences
setSubsequence	Update partial sequences
setSubset	Update elements of object
write	Write contents of BioRead or BioMap object to file

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects in the MATLAB Programming Fundamentals documentation.

Indexing

`BioMap` objects support dot . indexing to extract, assign, and delete data.

Examples

Construct a BioMap object

This example shows how to construct a BioMap object from a SAM file and from a structure.

Construct a BioMap object from a SAM-formatted file that is provided with Bioinformatics Toolbox™ and set the Name property.

```
BMObj1 = BioMap('ex1.sam', 'Name', 'MyObject')
```

```
BMObj1 = 
  BioMap with properties:

    SequenceDictionary: 'seq1'
             Reference: [1501x1 File indexed property]
             Signature: [1501x1 File indexed property]
                 Start: [1501x1 File indexed property]
        MappingQuality: [1501x1 File indexed property]
                  Flag: [1501x1 File indexed property]
          MatePosition: [1501x1 File indexed property]
```

```
               Quality: [1501x1 File indexed property]
              Sequence: [1501x1 File indexed property]
                Header: [1501x1 File indexed property]
                 NSeqs: 1501
                  Name: 'MyObject'
```

Construct a structure containing information from a SAM file.

```
SAMStruct = samread('ex1.sam');
```

Construct a BioMap object from this structure.

```
BMObj2 = BioMap(SAMStruct)
```

```
BMObj2 =
  BioMap with properties:

    SequenceDictionary: {'seq1'}
             Reference: {1501x1 cell}
             Signature: {1501x1 cell}
                 Start: [1501x1 uint32]
        MappingQuality: [1501x1 uint8]
                  Flag: [1501x1 uint16]
          MatePosition: [1501x1 uint32]
               Quality: {1501x1 cell}
              Sequence: {1501x1 cell}
                Header: {1501x1 cell}
                 NSeqs: 1501
                  Name: ''
```

See Also
`BioIndexedFile` | `BioRead` | `saminfo` | `samread` | `baminfo` | `bamread` | `align2cigar` | `cigar2align`

Topics
“Manage Sequence Read Data in Objects”

External Websites
Sequence Read Archive
SAM format specification

aa2nt

Convert amino acid sequence to nucleotide sequence

Syntax

```
ntSeq = aa2nt(aaSeq)
ntSeq = aa2nt(aaSeq,Name=Value)
```

Description

`ntSeq = aa2nt(aaSeq)` converts an amino acid sequence `aaSeq` to a nucleotide sequence `ntSeq` using the standard genetic code.

`ntSeq = aa2nt(aaSeq,Name=Value)` uses additional options specified by one or more name-value arguments. For example, `ntSeq = aa2nt(aaSeq,GeneticCode=2)` uses the vertebrate mitochondrial genetic code.

Examples

Convert an amino acid sequence to a nucleotide sequence

Create an amino acid sequence.

```
rng('default') % For reproducibility
seq = randseq(20,Alphabet="amino")
```

```
seq =
'TYNYMRQLVVDVVITNHYSV'
```

Convert it to a nucleotide sequence using the standard genetic code.

```
aa2nt(seq)
```

```
ans =
'ACATATAACTACATGAGACAGCTTGTAGTTGACGTTGTCATTACTAACCACTATAGCGTT'
```

Convert it using the vertebrate mitochondrial genetic code.

```
aa2nt(seq,GeneticCode=2)
```

```
ans =
'ACCTATAACTACATACGCCAACTCGTAGTGGATGTAGTAATTACTAATCACTATTCGGTT'
```

Convert using the Echinoderm Mitochondrial genetic code and the RNA alphabet.

```
aa2nt(seq,GeneticCode="ec",Alphabet="RNA")
```

```
ans =
'ACGUAUAACUACAUGCGGCAGUUAGUUGUCGACGUCGUGAUUACGAACCAUUAUAGUGUC'
```

Input Arguments

aaSeq — Amino acid sequence
character vector | string | row vector of positive integers | structure

Amino acid sequence, specified as one of the following.

- Character vector or string of single-letter codes specifying an amino acid sequence. For valid letter codes, see the table Mapping Amino Acid Letter Codes to Integers. Unknown characters are mapped to `0`.
- Row vector of integers specifying an amino acid sequence. For valid integers, see the table Mapping Amino Acid Integers to Letter Codes.
- MATLAB structure containing a `Sequence` field that contains an amino acid sequence, such as returned by `fastaread`, `getgenpept`, `genpeptread`, `getpdb`, or `pdbread`.

In general, the mapping from an amino acid to a nucleotide codon is not a one-to-one mapping. For amino acids with multiple possible nucleotide codons, this function randomly selects a codon corresponding to that particular amino acid. For the ambiguous characters `B` and `Z`, one of the amino acids corresponding to the letter is selected randomly, and then a codon sequence is selected randomly. For the ambiguous character `X`, a codon sequence is selected randomly from all possibilities.

Example: `"TYNYMRQLVV"`

Data Types: `char` | `string`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: `ntseq = aa2nt(seq,GeneticCode=2,Alphabet="RNA")`

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `ntseq = aa2nt(seq,'GeneticCode',"ec",'Alphabet',"RNA")`

`GeneticCode` — Genetic code number or name
`1` or `"Standard"` (default) | positive integer | character vector | string scalar

Genetic code number or name, specified as a positive integer, character vector, or string scalar. The following table has the list of genetic codes and their corresponding code names.

Genetic Codes

Code Number	Code Name
1	`Standard`
2	`Vertebrate Mitochondrial`
3	`Yeast Mitochondrial`
4	`Mold`, `Protozoan`, `Coelenterate Mitochondrial`, and `Mycoplasma/ Spiroplasma`
5	`Invertebrate Mitochondrial`
6	`Ciliate`, `Dasycladacean`, and `Hexamita Nuclear`
9	`Echinoderm Mitochondrial`
10	`Euplotid Nuclear`
11	`Bacterial` and `Plant Plastid`
12	`Alternative Yeast Nuclear`
13	`Ascidian Mitochondrial`
14	`Flatworm Mitochondrial`
15	`Blepharisma Nuclear`
16	`Chlorophycean Mitochondrial`
21	`Trematode Mitochondrial`
22	`Scenedesmus Obliquus Mitochondrial`
23	`Thraustochytrium Mitochondrial`

Tip If you use a code name, you can truncate the name to the first two letters of the name.

The amino acid to nucleotide codon mapping for the standard genetic code is shown next.

Amino Acid Name	Amino Acid Code	Nucleotide Codon
Alanine	`A`	`GCT GCC GCA GCG`
Arginine	`R`	`CGT CGC CGA CGG AGA AGG`
Asparagine	`N`	`AAT AAC`
Aspartic acid (Aspartate)	`D`	`GAT GAC`
Cysteine	`C`	`TGT TGC`
Glutamine	`Q`	`CAA CAG`
Glutamic acid (Glutamate)	`E`	`GAA GAG`
Glycine	`G`	`GGT GGC GGA GGG`
Histidine	`H`	`CAT CAC`
Isoleucine	`I`	`ATT ATC ATA`

Amino Acid Name	Amino Acid Code	Nucleotide Codon
Leucine	`L`	`TTA TTG† CTT CTC CTA CTG†` † indicates alternative start codon for the Standard Genetic Code as defined here. If you are using `nt2aa`, alternative start codons are converted to methionine (M) by default when one of these codons are the first codon of a sequence. To change this default behavior, set the `AlternativeStartCodons` name-value argument of `nt2aa` to `false`.
Lysine	`K`	`AAA AAG`
Methionine	`M`	`ATG`
Phenylalanine	`F`	`TTT TTC`
Proline	`P`	`CCT CCC CCA CCG`
Serine	`S`	`TCT TCC TCA TCG AGT AGC`
Threonine	`T`	`ACT ACC ACA ACG`
Tryptophan	`W`	`TGG`
Tyrosine	`Y`	`TAT TAC`
Valine	`V`	`GTT GTC GTA GTG`
Asparagine or Aspartic acid (Aspartate)	`B`	Random codon from `D` and `N`
Glutamine or Glutamic acid (Glutamate)	`Z`	Random codon from `E` and `Q`
Unknown amino acid (any amino acid)	`X`	Random codon
Translation stop	`*`	`TAA TAG TGA`
Gap of indeterminate length	`-`	`---`
Unknown character (any character or symbol not in table)	`?`	`???`

Data Types: `double` | `char` | `string`

`Alphabet` — Nucleotide alphabet
`"DNA"` (default) | `"RNA"`

Nucleotide alphabet, specified as `"DNA"` or `"RNA"`. If `"DNA"`, the function uses `A`, `C`, `G`, and `T`. If `"RNA"`, the function uses `A`, `C`, `G`, and `U`.

Data Types: `char` | `string`

Output Arguments

`ntSeq` — Nucleotide sequence
character vector

Nucleotide sequence, returned as a character vector.

Version History

Introduced before R2006a

See Also

`aminolookup` | `baselookup` | `geneticcode` | `nt2aa` | `revgeneticcode` | `seqviewer` | `rand` | `rng`

aacount

Count amino acids in sequence

Syntax

```
AAStruct = aacount(SeqAA)
AAStruct = aacount(SeqAA, ...'Ambiguous', AmbiguousValue, ...)
AAStruct = aacount(SeqAA, ...'Gaps', GapsValue, ...)
AAStruct = aacount(SeqAA, ...'Chart', ChartValue, ...)
```

Input Arguments

SeqAA	One of the following: • Character vector or string containing single-letter codes specifying an amino acid sequence. For valid letter codes, see the table Mapping Amino Acid Letter Codes to Integers. Unknown characters are mapped to `0`. • Row vector of integers specifying an amino acid sequence. For valid integers, see the table Mapping Amino Acid Integers to Letter Codes. • MATLAB structure containing a `Sequence` field that contains an amino acid sequence, such as returned by `fastaread`, `getgenpept`, `genpeptread`, `getpdb`, or `pdbread`. Examples: `'ARN'` or `[1 2 3]`
AmbiguousValue	Character vector or string specifying how to treat ambiguous amino acid characters (`B`, `Z`, or `X`). Choices are: • `'ignore'` (default) — Skips ambiguous characters • `'bundle'` — Counts ambiguous characters and reports the total count in the `Ambiguous` field. • `'prorate'` — Counts ambiguous characters and distributes them proportionately in the appropriate fields. For example, the counts for the character `B` are distributed evenly between the `D` and `N` fields. • `'individual'` — Counts ambiguous characters and reports them in individual fields. • `'warn'` — Skips ambiguous characters symbols and displays a warning.
GapsValue	Specifies whether gaps, indicated by a hyphen (`-`), are counted or ignored. Choices are `true` or `false` (default).
ChartValue	Character vector or string specifying a chart type. Choices are `'pie'` or `'bar'`.

Output Arguments

AAStruct	1-by-1 MATLAB structure containing fields for the standard 20 amino acids (A, R, N, D, C, Q, E, G, H, I, L, K, M, F, P, S, T, W, Y, and V).

Description

`AAStruct = aacount(SeqAA)` counts the number of each type of amino acid in *SeqAA*, an amino acid sequence, and returns the counts in *AAStruct*, a 1-by-1 MATLAB structure containing fields for the standard 20 amino acids (A, R, N, D, C, Q, E, G, H, I, L, K, M, F, P, S, T, W, Y, and V).

- Ambiguous amino acid characters (B, Z, or X), gaps, indicated by a hyphen (-), and end terminators (*) are ignored by default.
- Unrecognized characters are ignored and cause the following warning message.

```
Warning: Unknown symbols appear in the sequence. These will be ignored.
```

`AAStruct = aacount(SeqAA, ...'PropertyName', PropertyValue, ...)` calls `aacount` with optional properties that use property name/property value pairs. You can specify one or more properties in any order. Each *PropertyName* must be enclosed in single quotation marks and is case insensitive. These property name/property value pairs are as follows:

`AAStruct = aacount(SeqAA, ...'Ambiguous', AmbiguousValue, ...)` specifies how to treat ambiguous amino acid characters (B, Z, or X). Choices are:

- `'ignore'` (default)
- `'bundle'`
- `'prorate'`
- `'individual'`
- `'warn'`

`AAStruct = aacount(SeqAA, ...'Gaps', GapsValue, ...)` specifies whether gaps, indicated by a hyphen (-), are counted or ignored. Choices are `true` or `false` (default).

`AAStruct = aacount(SeqAA, ...'Chart', ChartValue, ...)` creates a chart showing the relative proportions of the amino acids. *ChartValue* can be `'pie'` or `'bar'`.

Examples

Count amino acids in a sequence

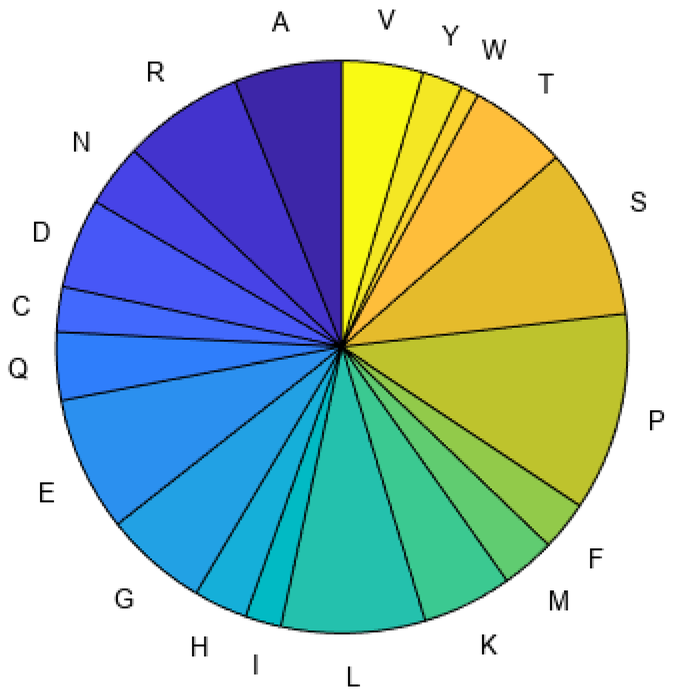
Use the `fastaread` function to load the sequence of the human p53 tumor protein.

```
p53 = fastaread('p53aa.txt')
```

```
p53 = struct with fields:
      Header: 'gi|8400738|ref|NP_000537.2| tumor protein p53 [Homo sapiens]'
    Sequence: 'MEEPQSDPSVEPPLSQETFSDLWKLLPENNVLSPLPSQAMDDLMLSPDDIEQWFTEDPGPDEAPRMPEAAPRVAPAPAAPT
```

Count the amino acids in the sequence, and display the results in a pie chart.

```
count = aacount(p53,'Chart','pie');
```



Version History

Introduced before R2006a

See Also

aminolookup | atomiccomp | basecount | codoncount | dimercount | isoelectric | molweight | proteinplot | proteinpropplot | seqviewer

abstract

Class: `bioma.ExpressionSet`
Package: `bioma`

Retrieve or set abstract describing experiment in ExpressionSet object

Syntax

```
Abstract = abstract(ESObj)
NewESObj = abstract(ESObj, NewAbstract)
```

Description

`Abstract = abstract(ESObj)` returns a character vector containing the abstract information describing the experiment from a MIAME object in an ExpressionSet object.

`NewESObj = abstract(ESObj, NewAbstract)` replaces the abstract information in the MIAME object in *`ESObj`*, an ExpressionSet object, with *`NewAbstract`*, a character vector containing new abstract information, and returns *`NewESObj`*, a new ExpressionSet object.

Input Arguments

`ESObj`

Object of the `bioma.ExpressionSet` class.

Default:

`NewAbstract`

Character vector containing new abstract information.

Default:

Output Arguments

`Abstract`

Character vector containing the abstract information describing the experiment from a MIAME object in an ExpressionSet object.

`NewESObj`

Object of the `bioma.ExpressionSet` class, returned after replacing the abstract information.

Examples

Construct an ExpressionSet object, `ESObj`, as described in the "Examples" on page 1-0 section of the `bioma.ExpressionSet` class reference page. Retrieve the abstract information stored in the MIAME object stored in the ExpressionSet object:

```
% Retrieve abstract text from the MIAME object
Abstract = abstract(ESObj)
```

See Also

`bioma.ExpressionSet` | `bioma.data.MIAME`

Topics

"Managing Gene Expression Data in Objects"

addTitle

Add title to heatmap or clustergram

Syntax

```
addTitle(hm_cg_object,title)
addTitle(hm_cg_object,title,Name,Value)
textObj = addTitle( ___ )
```

Description

`addTitle(hm_cg_object,title)` adds a title to the heatmap or clustergram.

`addTitle(hm_cg_object,title,Name,Value)` specifies the title text object properties using name-value pair arguments. For example, `addTitle(hm_cg_object,'Gene Expression Data','Color','red','FontSize',12)` displays the title in red 12-point font. You can specify multiple name-value pairs. Enclose each property name in quotes.

`textObj = addTitle( ___ )` returns a text object `textObj` used as the title of the heatmap or clustergram using any of the input argument combinations in the previous syntaxes.

Examples

Add Custom Title and Labels to Heatmap

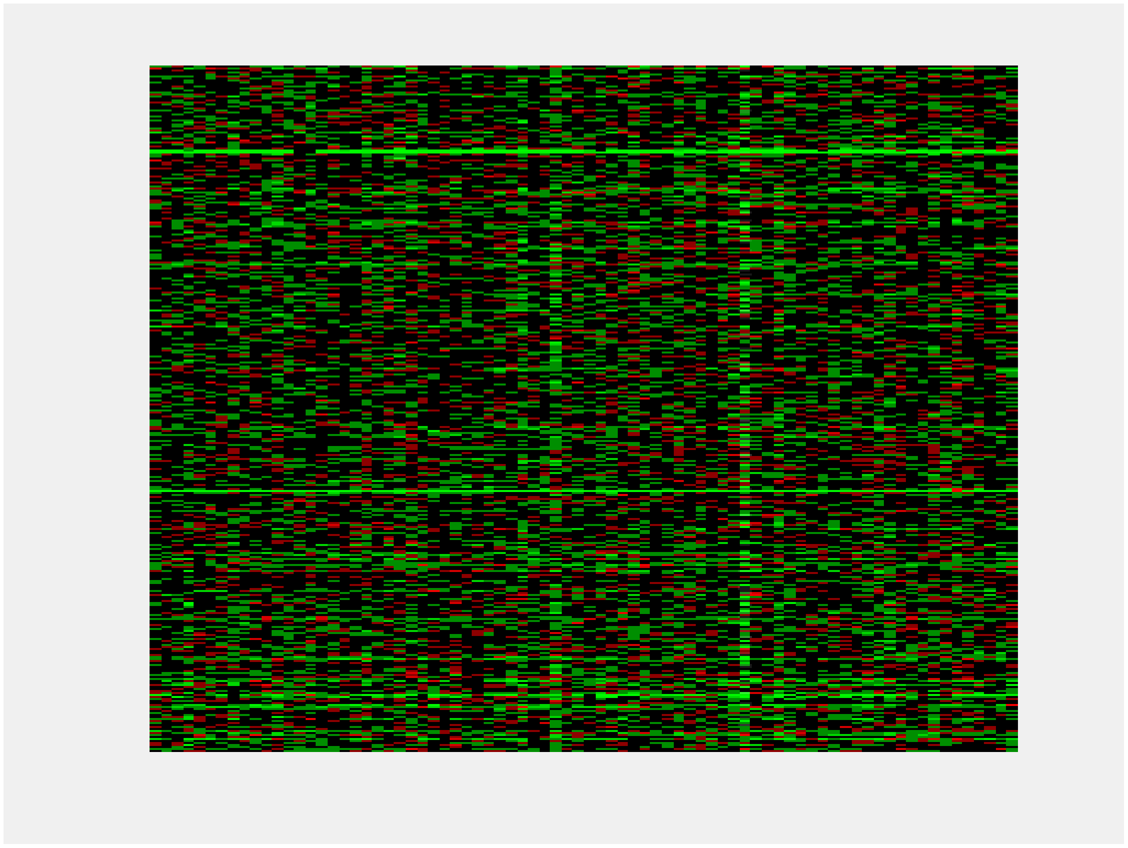
Load a sample of gene expression data.

```
load bc_train_filtered
```

Display a heatmap of the gene expression values for 4918 genes from 78 samples.

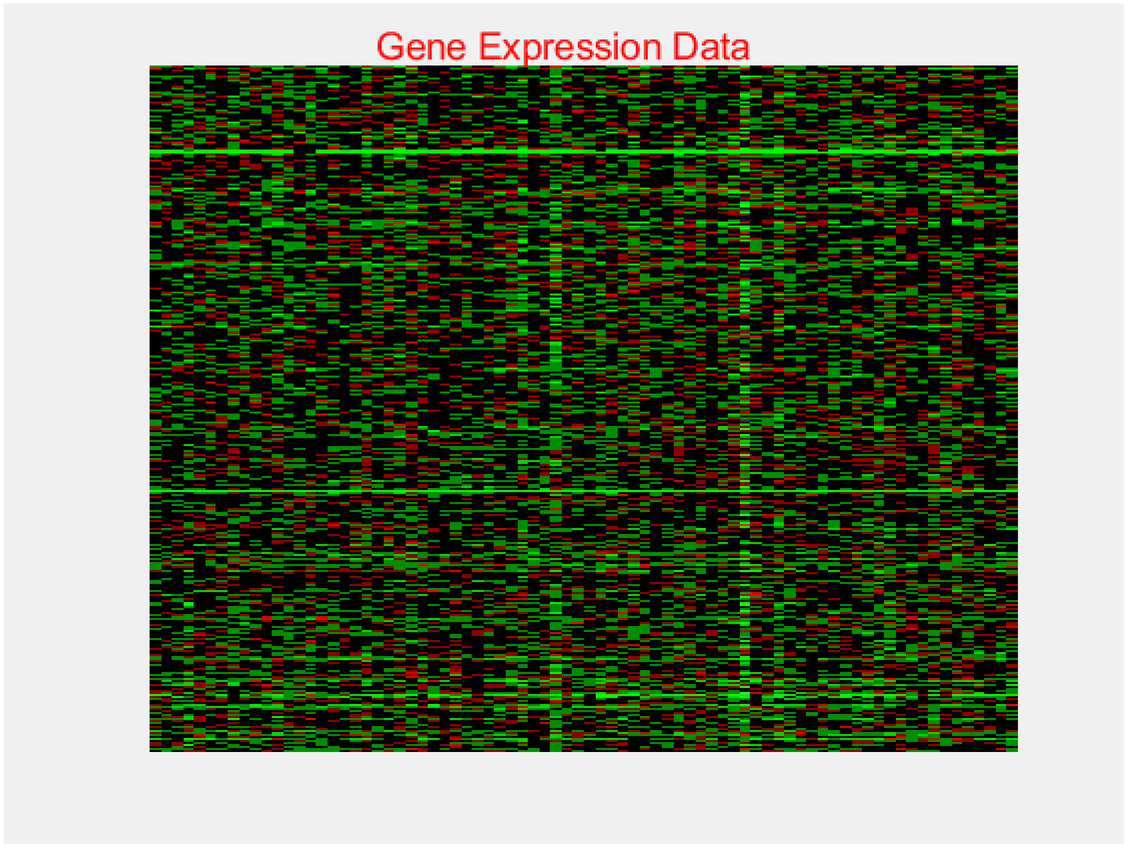
```
hmo = HeatMap(bcTrainData.Log10Ratio);
```

```
             Standardize: '[column | row | {none}]'
               Symmetric: '[true | false].'
            DisplayRange: 'Scalar.'
                Colormap: []
               ImputeFun: 'string -or- function handle -or- cell array'
            ColumnLabels: 'Cell array of strings, or an empty cell array'
               RowLabels: 'Cell array of strings, or an empty cell array'
      ColumnLabelsRotate: []
         RowLabelsRotate: []
                Annotate: '[on | {off}]'
          AnnotPrecision: []
              AnnotColor: []
       ColumnLabelsColor: 'A structure array.'
          RowLabelsColor: 'A structure array.'
       LabelsWithMarkers: '[true | false].'
    ColumnLabelsLocation: '[ top | {bottom} ]'
       RowLabelsLocation: '[ {left} | right ]'
```



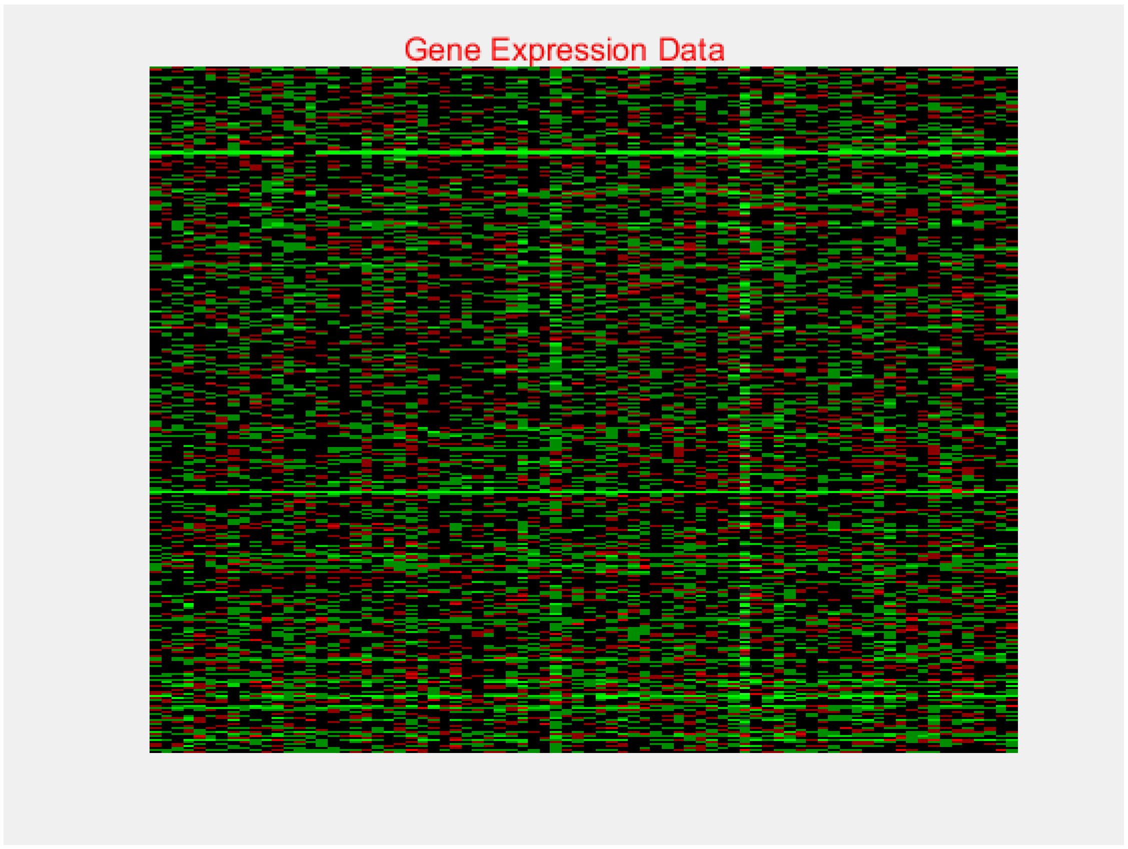
Add a title to the heatmap in red.

```
title = addTitle(hmo,'Gene Expression Data','Color','red');
```



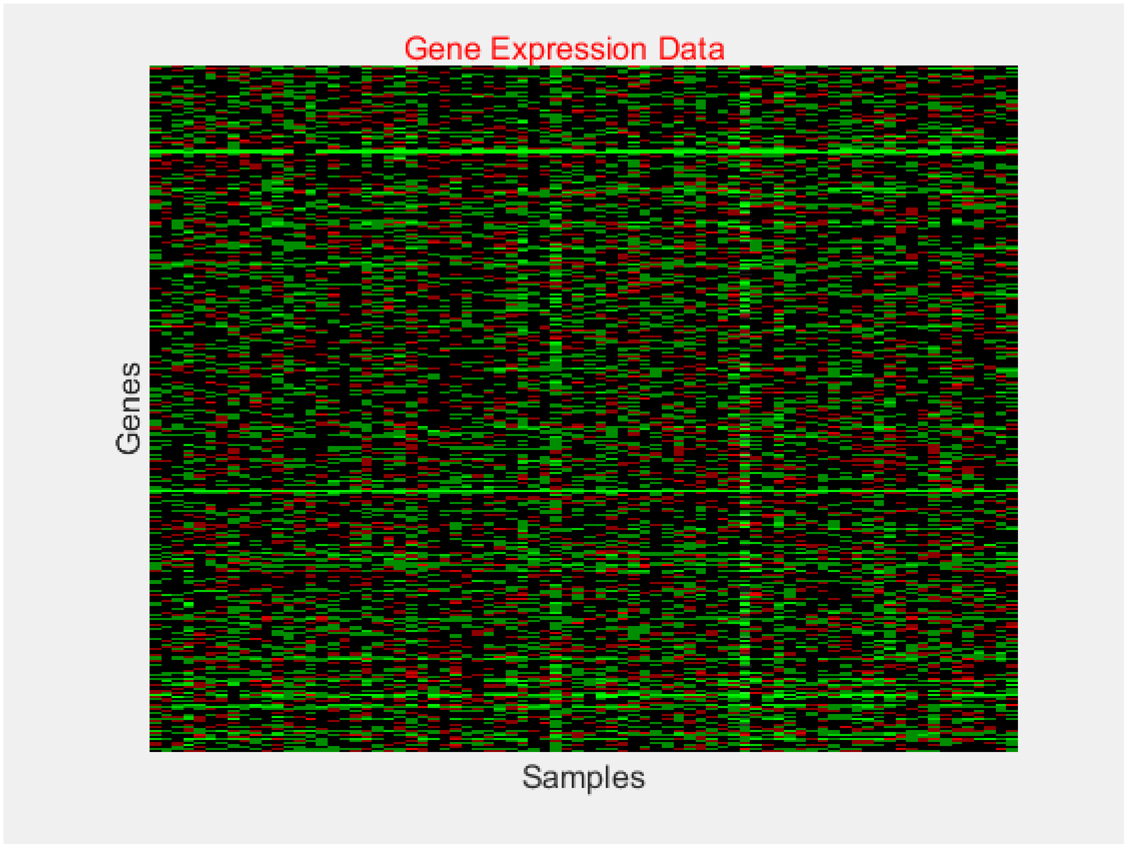
Change the title font size.

```
title.FontSize = 12;
```



Add labels to the x-axis and y-axis.

```
addXLabel(hmo,'Samples','FontSize',12);
addYLabel(hmo,'Genes','FontSize',12);
```



Input Arguments

`hm_cg_object` — Heatmap or clustergram object
`HeatMap` object | `clustergram` object

Heatmap or clustergram object, specified as a `HeatMap` object or `clustergram` object.

`title` — Title of heatmap or clustergram
character vector | string

Title of the heatmap or clustergram, specified as a character vector or string.

Example: `'Inverse of an Upper Hessenberg Matrix'`

Data Types: `char` | `string`

Output Arguments

`textObj` — Heatmap or clustergram title
`Text` object

Heatmap or clustergram title, returned as a `Text` object.

Version History

Introduced in R2009b

See Also

Text Properties | `HeatMap` | `clustergram`

addXLabel

Label *x*-axis of heatmap or clustergram

Syntax

```
addXLabel(hm_cg_object,label)
addXLabel(hm_cg_object,label,Name,Value)
textObj = addXLabel( ___ )
```

Description

`addXLabel(hm_cg_object,label)` adds a label below the *x*-axis of the heatmap or clustergram.

`addXLabel(hm_cg_object,label,Name,Value)` specifies the label text object properties using name-value pair arguments. For example, `addXLabel(hmObj,'Samples','Color','red','FontSize',12)` displays the label in red 12-point font. You can specify multiple name-value pairs. Enclose each property name in quotes.

`textObj = addXLabel( ___ )` returns a text object `textObj` used as the label of the heatmap or clustergram using any of the input argument combinations from the previous syntaxes.

Examples

Add Custom Title and Labels to Heatmap

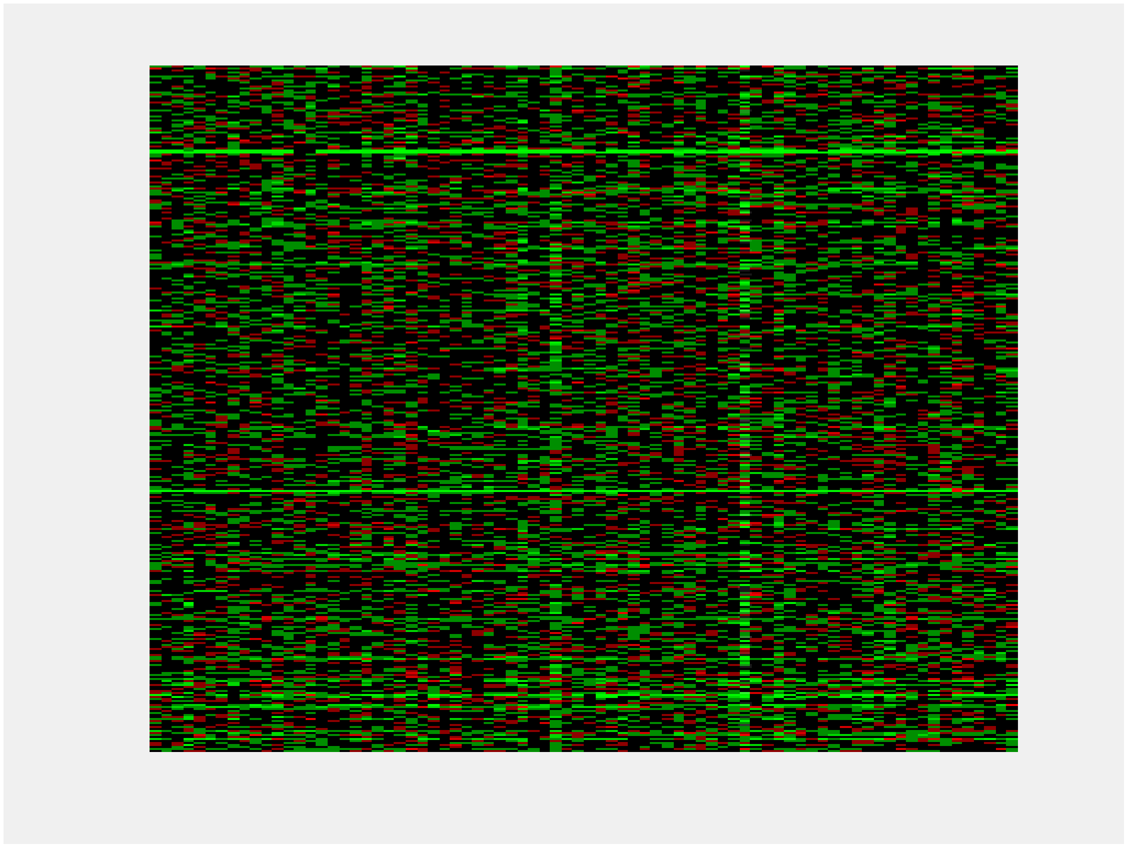
Load a sample of gene expression data.

```
load bc_train_filtered
```

Display a heatmap of the gene expression values for 4918 genes from 78 samples.

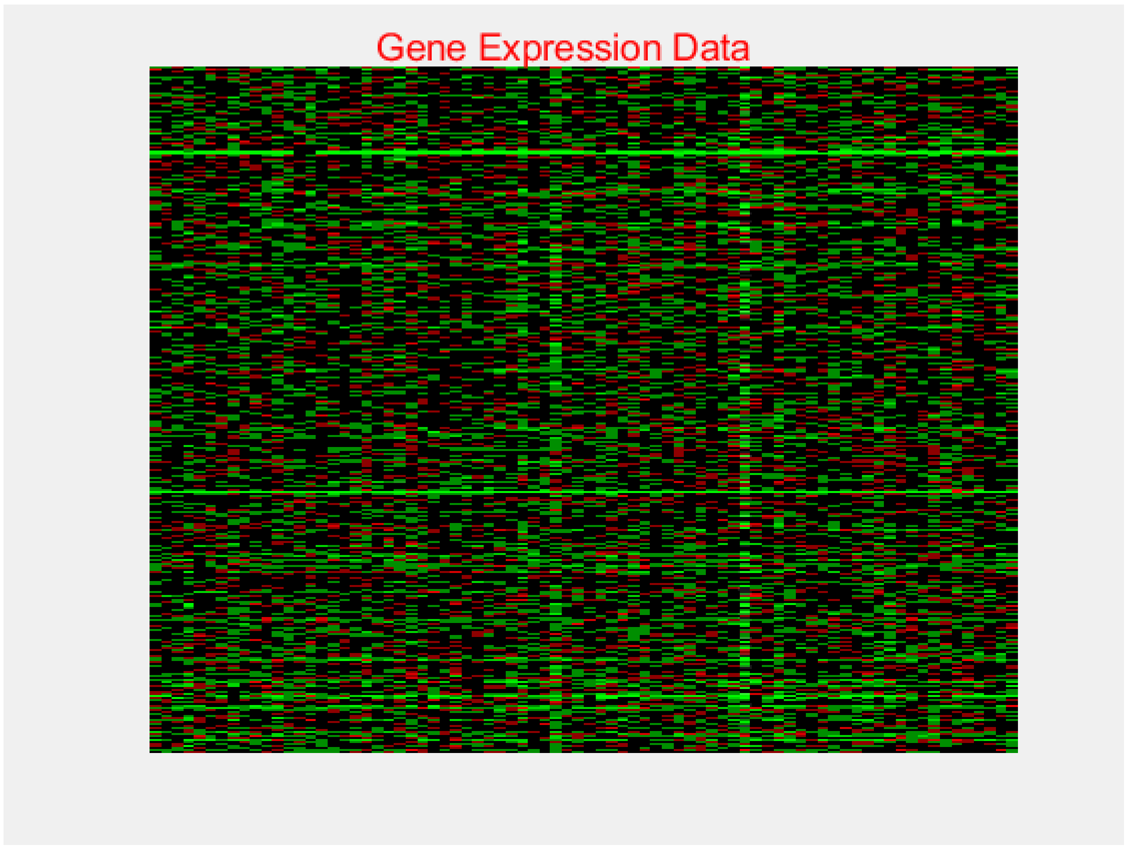
```
hmo = HeatMap(bcTrainData.Log10Ratio);
```

```
             Standardize: '[column | row | {none}]'
               Symmetric: '[true | false].'
            DisplayRange: 'Scalar.'
                Colormap: []
               ImputeFun: 'string -or- function handle -or- cell array'
            ColumnLabels: 'Cell array of strings, or an empty cell array'
               RowLabels: 'Cell array of strings, or an empty cell array'
      ColumnLabelsRotate: []
         RowLabelsRotate: []
                Annotate: '[on | {off}]'
          AnnotPrecision: []
              AnnotColor: []
       ColumnLabelsColor: 'A structure array.'
          RowLabelsColor: 'A structure array.'
       LabelsWithMarkers: '[true | false].'
    ColumnLabelsLocation: '[ top | {bottom} ]'
       RowLabelsLocation: '[ {left} | right ]'
```



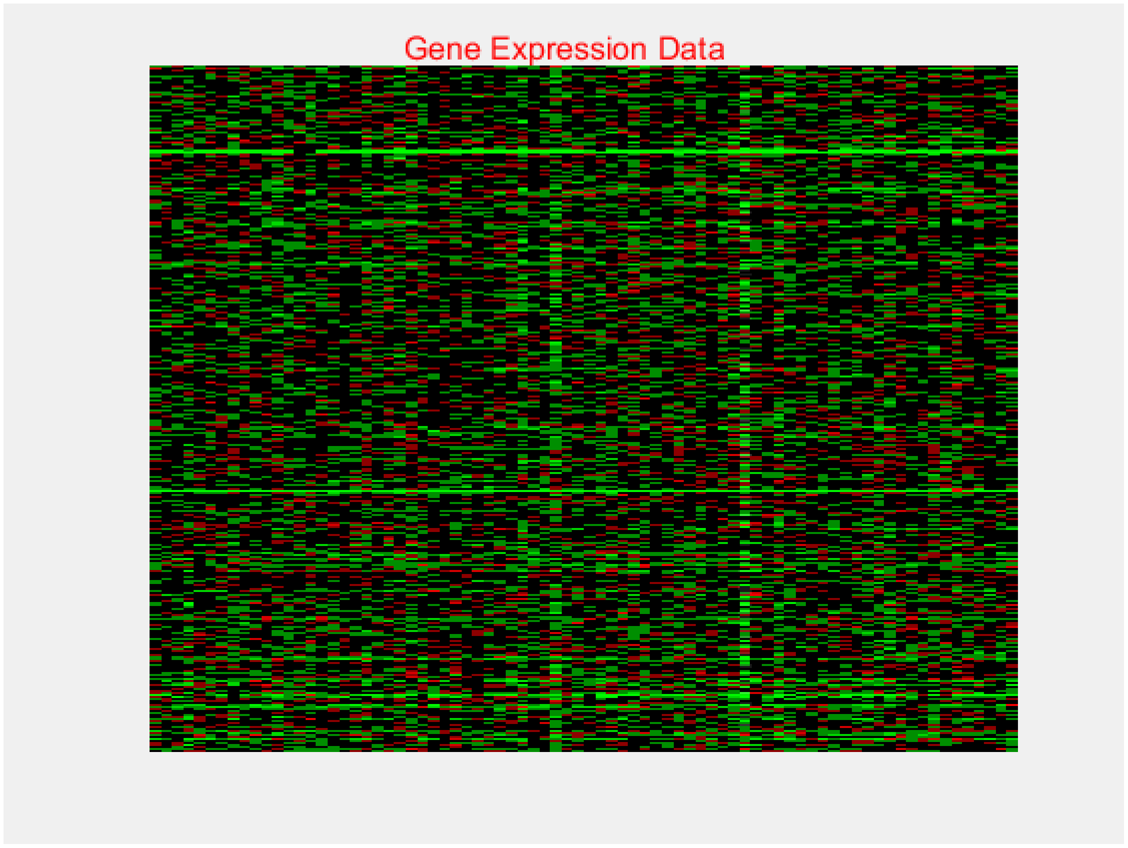
Add a title to the heatmap in red.

```
title = addTitle(hmo,'Gene Expression Data','Color','red');
```



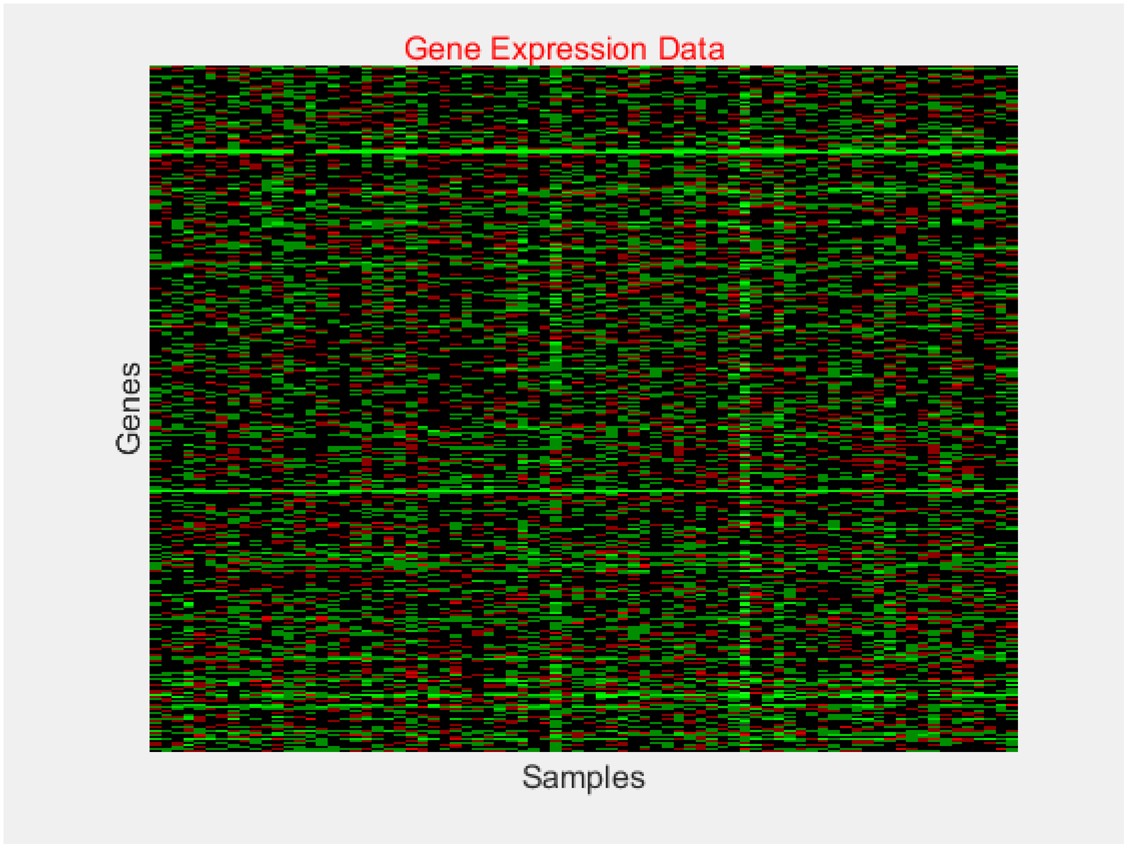
Change the title font size.

```
title.FontSize = 12;
```



Add labels to the x-axis and y-axis.

```
addXLabel(hmo,'Samples','FontSize',12);
addYLabel(hmo,'Genes','FontSize',12);
```



Input Arguments

`hm_cg_object` — Heatmap or clustergram object
`HeatMap` object | `clustergram` object

Heatmap or clustergram object, specified as a `HeatMap` object or `clustergram` object.

`label` — *x*-axis label
character vector | string

x-axis label, specified as a character vector or string.

Example: `'Samples'`

Data Types: `char` | `string`

Output Arguments

`textObj` — *x*-axis label
`Text` object

x-axis label, returned as a `Text` object.

Version History

Introduced in R2009b

See Also

Text Properties | `HeatMap` | `clustergram`

addYLabel

Label *y*-axis of heatmap or clustergram

Syntax

```
addYLabel(hm_cg_object,label)
addYLabel(hm_cg_object,label,Name,Value)
textObj = addYLabel( ___ )
```

Description

`addYLabel(hm_cg_object,label)` adds a label below the *y*-axis of the heatmap or clustergram.

`addYLabel(hm_cg_object,label,Name,Value)` specifies the label text object properties using name-value pair arguments. For example, `addYLabel(hmObj,'Samples','Color','red','FontSize',12)` displays the label in red 12-point font. You can specify multiple name-value pairs. Enclose each property name in quotes.

`textObj = addYLabel( ___ )` returns a text object `textObj` used as the label of the heatmap or clustergram using any of the input argument combinations from the previous syntaxes.

Examples

Add Custom Title and Labels to Heatmap

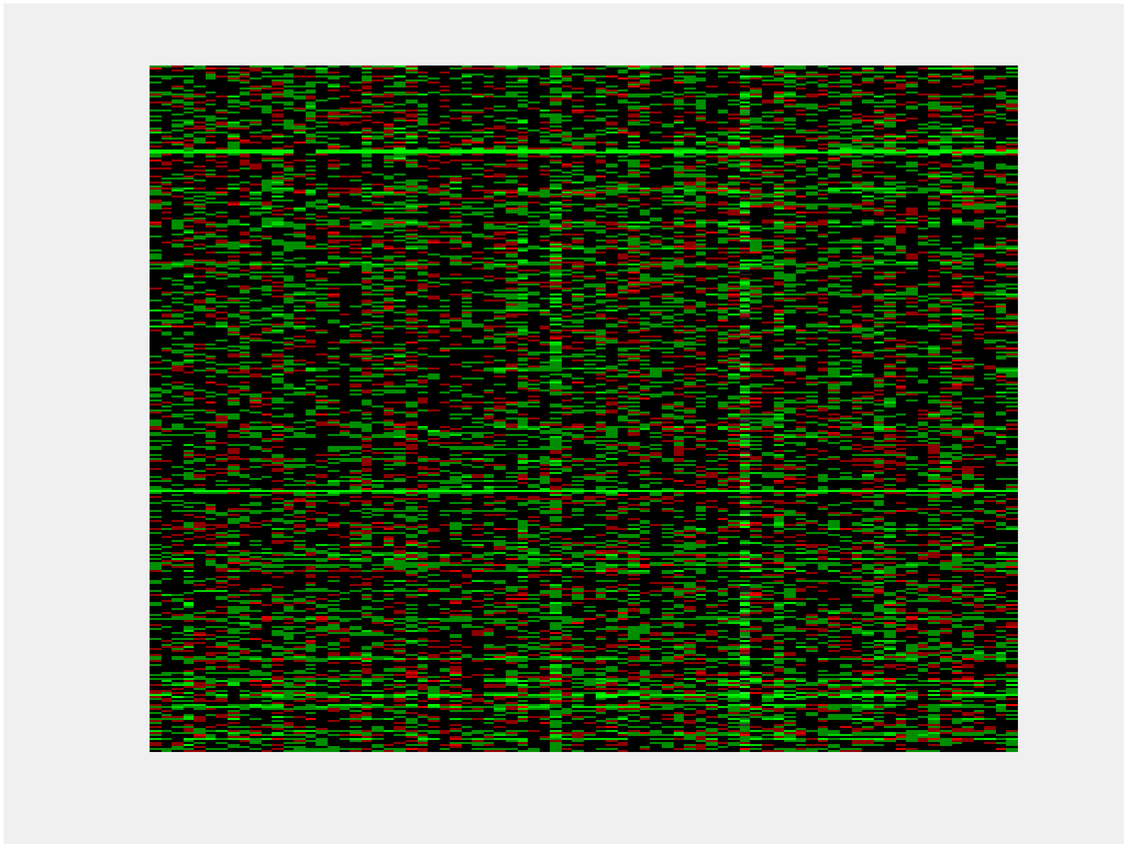
Load a sample of gene expression data.

```
load bc_train_filtered
```

Display a heatmap of the gene expression values for 4918 genes from 78 samples.

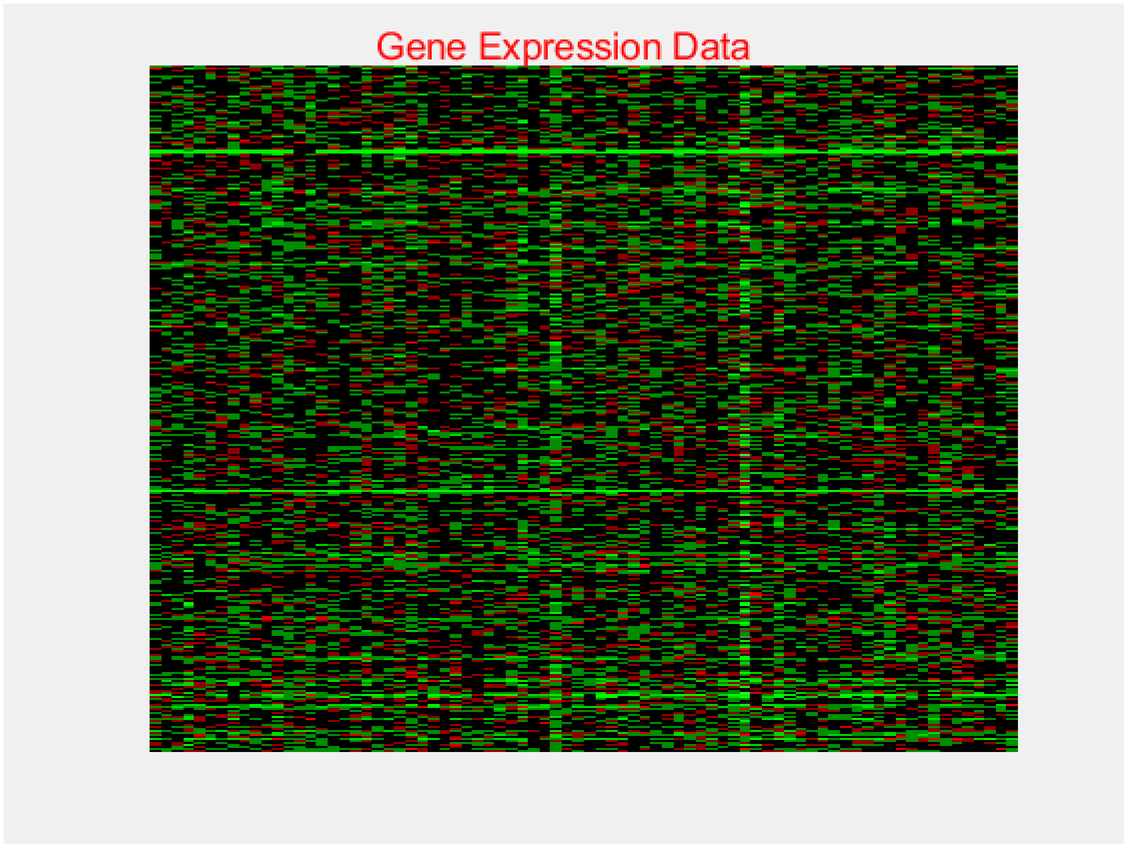
```
hmo = HeatMap(bcTrainData.Log10Ratio);
```

```
             Standardize: '[column | row | {none}]'
               Symmetric: '[true | false].'
            DisplayRange: 'Scalar.'
                Colormap: []
               ImputeFun: 'string -or- function handle -or- cell array'
            ColumnLabels: 'Cell array of strings, or an empty cell array'
               RowLabels: 'Cell array of strings, or an empty cell array'
      ColumnLabelsRotate: []
         RowLabelsRotate: []
                Annotate: '[on | {off}]'
          AnnotPrecision: []
              AnnotColor: []
       ColumnLabelsColor: 'A structure array.'
          RowLabelsColor: 'A structure array.'
       LabelsWithMarkers: '[true | false].'
    ColumnLabelsLocation: '[ top | {bottom} ]'
       RowLabelsLocation: '[ {left} | right ]'
```



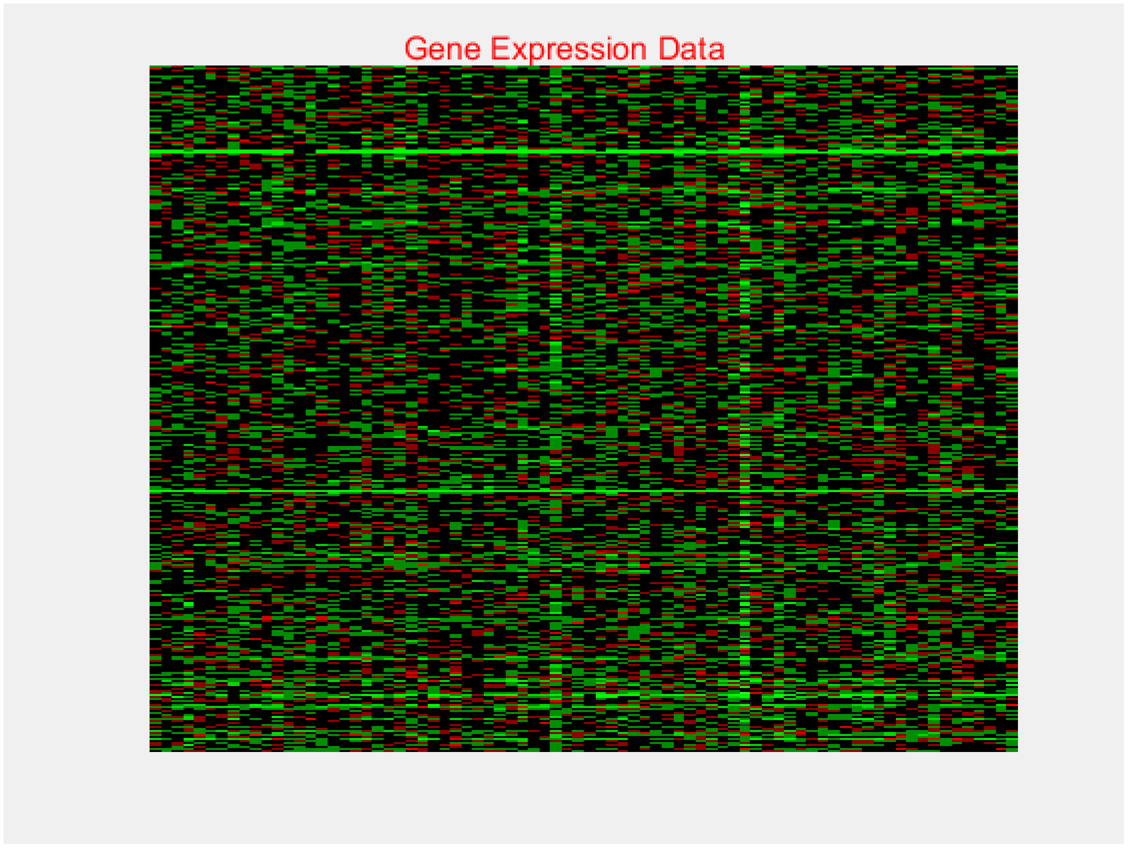
Add a title to the heatmap in red.

```
title = addTitle(hmo,'Gene Expression Data','Color','red');
```



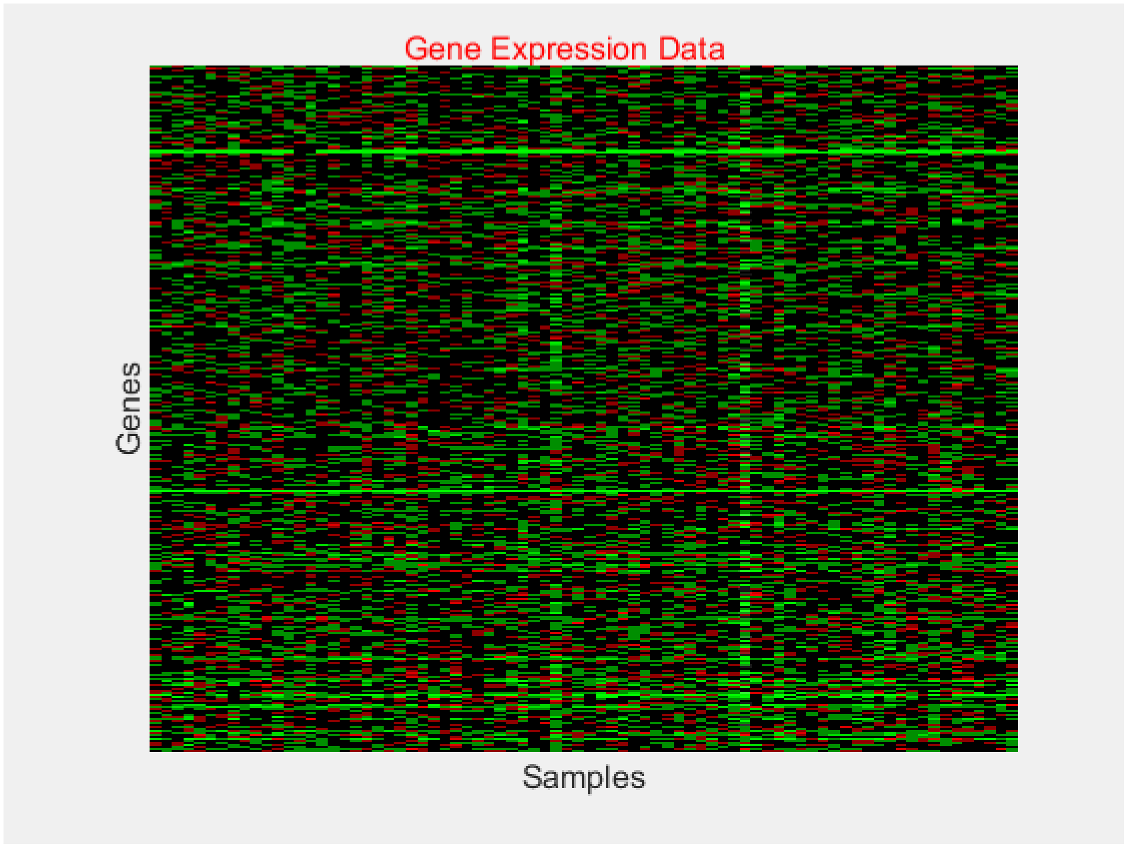
Change the title font size.

```
title.FontSize = 12;
```



Add labels to the x-axis and y-axis.

```
addXLabel(hmo,'Samples','FontSize',12);
addYLabel(hmo,'Genes','FontSize',12);
```



Input Arguments

`hm_cg_object` — Heatmap or clustergram object
`HeatMap` object | `clustergram` object

Heatmap or clustergram object, specified as a `HeatMap` object or `clustergram` object.

`label` — *y*-axis label
character vector | string

y-axis label, specified as a character vector or string.

Example: `'Genes'`

Data Types: `char` | `string`

Output Arguments

`textObj` — *y*-axis label
`Text` object

y-axis label, returned as a `Text` object.

Version History

Introduced in R2009b

See Also

Text Properties | `HeatMap` | `clustergram`

affygcrma

Perform GC Robust Multi-array Average (GCRMA) procedure on Affymetrix microarray probe-level data

Syntax

```
Expression = affygcrma(CELFiles, CDFFile, SeqFile)
Expression = affygcrma(ProbeStructure, Seq)

Expression = affygcrma(CELFiles, CDFFile, SeqFile, ...'CELPath',
CELPathValue, ...)
Expression = affygcrma(CELFiles, CDFFile, SeqFile, ...'CDFPath',
CDFPathValue, ...)
Expression = affygcrma(CELFiles, CDFFile, SeqFile, ...'SeqPath',
SeqPathValue, ...)
Expression = affygcrma(..., 'ChipIndex', ChipIndexValue, ...)
Expression = affygcrma(..., 'OpticalCorr', OpticalCorrValue, ...)
Expression = affygcrma(..., 'CorrConst', CorrConstValue, ...)
Expression = affygcrma(..., 'Method', MethodValue, ...)
Expression = affygcrma(..., 'TuningParam', TuningParamValue, ...)
Expression = affygcrma(..., 'GSBCorr', GSBCorrValue, ...)
Expression = affygcrma(..., 'Median', MedianValue, ...)
Expression = affygcrma(..., 'Output', OutputValue, ...)
Expression = affygcrma(..., 'Showplot', ShowplotValue, ...)
Expression = affygcrma(..., 'Verbose', VerboseValue, ...)
```

Input Arguments

CELFiles	Any of the following: • Character vector or string specifying a single CEL file name. • `'*'`, which reads all CEL files in the current folder. • `' '`, which opens the Select CEL Files dialog box from which you select the CEL files. From this dialog box, you can press and hold **Ctrl** or **Shift** while clicking to select multiple CEL files. • Cell array of character vectors or string vector containing CEL file names.
CDFFile	Either of the following: • Character vector or string specifying a CDF file name. • `' '`, which opens the Select CDF File dialog box from which you select the CDF file.

`SeqFile`	Either of the following: • Character vector or string specifying a file name of a sequence file (tab-separated or FASTA) that contains the following information for a specific type of Affymetrix® GeneChip® array: • Probe set IDs • Probe *x*-coordinates • Probe *y*-coordinates • Probe sequences in each probe set • Affymetrix GeneChip array type (FASTA file only) The sequence file (tab-separated or FASTA) must be on the MATLAB search path or in the Current Folder (unless you use the `SeqPath` property). In a tab-separated file, each row represents a probe; in a FASTA file, each header represents a probe. • An *N*-by-25 matrix of sequence information, such as returned by `affyprobeseqread`.
`Seq`	An *N*-by-25 matrix of sequence information, such as returned by `affyprobeseqread`.
`ProbeStructure`	MATLAB structure containing information from the CEL files, including probe intensities, probe indices, and probe set IDs, returned by the `celintensityread` function.
`CELPathValue`	Character vector or string specifying the path and folder where the files specified in *`CELFiles`* are stored.
`CDFPathValue`	Character vector or string specifying the path and folder where the file specified in *`CDFFile`* is stored.
`SeqPathValue`	Character vector or string specifying a folder or path and folder where *`SeqFile`* is stored.
`ChipIndexValue`	Positive integer specifying a chip. This chip's sequence information and mismatch probe intensity data is used to compute probe affinities. Default is `1`.
`OpticalCorrValue`	Controls the use of optical background correction on the input probe intensity values. Choices are `true` (default) or `false`.
`CorrConstValue`	Value that specifies the correlation constant, rho, for log background intensity for each PM/MM probe pair. Choices are any value `≥ 0` and `≤ 1`. Default is `0.7`.
`MethodValue`	Character vector or string that specifies the method to estimate the signal. Choices are `'MLE'`, a faster, ad hoc Maximum Likelihood Estimate method, or `'EB'`, a slower, more formal, empirical Bayes method. Default is `'MLE'`.

TuningParamValue	Value that specifies the tuning parameter used by the estimate method. This tuning parameter sets the lower bound of signal values with positive probability. Choices are a positive value. Default is `5` (MLE) or `0.5` (EB). **Tip** For information on determining a setting for this parameter, see Wu et al., 2004 on page 1-860.
GSBCorrValue	Specifies whether to perform gene-specific binding (GSB) correction using probe affinity data. Choices are `true` (default) or `false`. If there is no probe affinity information, this property is ignored.
MedianValue	Specifies the use of the median of the ranked values instead of the mean for normalization. Choices are `true` or `false` (default).
OutputValue	Specifies the scale of the returned gene expression values. Choices are: • `'log'` • `'log2'` • `'log10'` • `'linear'` • `@`*`functionname`* In the last instance, the data is transformed as defined by the function *`functionname`*. Default is `'log2'`.
ShowplotValue	Controls the display of a plot showing the $\log_2$ of mismatch (MM) probe intensity values from a specified chip (CEL file), versus that chip's MM probe affinities. The plot also shows the LOWESS fit for computing NSB data of the specified chip. Choices are `true`, `false`, or *`I`*, an integer specifying a chip. If set to `true`, the first chip is plotted. Default is: • `false` — When return values are specified. • `true` — When return values are not specified.
VerboseValue	Controls the display of the status of the reading of files and GCRMA processing. Choices are `true` (default) or `false`.

Output Arguments

Expression	DataMatrix object on page 1-734 containing the $\log_2$ gene expression values that have been background adjusted, normalized, and summarized using the GC Robust Multi-array Average (GCRMA) procedure. Each row in *`Expression`* corresponds to a gene (probe set), and each column corresponds to an Affymetrix CEL file.

Description

Expression = affygcrma(*CELFiles*, *CDFFile*, *SeqFile*) reads the specified Affymetrix CEL files, the associated CDF library file (created from Affymetrix GeneChip arrays for expression or genotyping assays), and the associated sequence file or matrix. It then processes the probe intensity values using GCRMA background adjustment, quantile normalization, and median-polish summarization procedures, then returns *Expression*, a DataMatrix object on page 1-734 containing the $\log_2$ based gene expression values in a matrix, the probe set IDs as row names, and the CEL file names as column names. Note that each row in *Expression* corresponds to a gene (probe set), and each column corresponds to an Affymetrix CEL file. (Each CEL file is generated from a separate chip. All chips should be of the same type.)

CELFiles is a character vector, string, string vector, or cell array of character vectors containing CEL file names. *CDFFile* is a character vector or string specifying a CDF file name. If you set *CELFiles* to '*', then it reads all CEL files in the current folder. If you set *CELFiles* or *CDFFile* to ' ', then it opens the Select Files dialog box from which you select the CEL files or CDF file. From this dialog box, you can press and hold **Ctrl** or **Shift** while clicking to select multiple CEL files. *SeqFile* is a file or matrix containing sequence information for probes on a specific type of Affymetrix GeneChip array.

Note For details on the reading of files and GCRMA processing, see celintensityread, affyprobeseqread, affyprobeaffinities, gcrma, gcrmabackadj, quantilenorm, and rmasummary.

Expression = affygcrma(*ProbeStructure*, *Seq*) uses GCRMA background adjustment, quantile normalization, and median-polish summarization procedures to process the probe intensity values in *ProbeStructure*. *ProbeStructure* is a MATLAB structure containing information from the CEL files, including probe intensities, probe indices, and probe set IDs, returned by the celintensityread function. *Seq* is a matrix containing sequence information for probes on a specific type of Affymetrix GeneChip array.

Expression = affygcrma(..., '*PropertyName*', *PropertyValue*, ...) calls affygcrma with optional properties that use property name/property value pairs. You can specify one or more properties in any order. Each *PropertyName* must be enclosed in single quotation marks and is case insensitive. These property name/property value pairs are as follows:

Expression = affygcrma(*CELFiles*, *CDFFile*, *SeqFile*, ...'CELPath', *CELPathValue*, ...) specifies a path and folder where the files specified by *CELFiles* are stored.

Expression = affygcrma(*CELFiles*, *CDFFile*, *SeqFile*, ...'CDFPath', *CDFPathValue*, ...) specifies a path and folder where the file specified by *CDFFile* is stored.

Expression = affygcrma(*CELFiles*, *CDFFile*, *SeqFile*, ...'SeqPath', *SeqPathValue*, ...) specifies a path and folder where the file specified by *SeqFile* is stored.

Expression = affygcrma(..., 'ChipIndex', *ChipIndexValue*, ...) computes probe affinities from MM probe intensity data using sequence information and mismatch probe intensity values from the chip specified by *ChipIndexValue*. Default *ChipIndexValue* is 1.

Expression = affygcrma(..., 'OpticalCorr', *OpticalCorrValue*, ...) controls the use of optical background correction on the input probe intensity values. Choices are true (default) or false.

Expression = affygcrma(..., 'CorrConst', *CorrConstValue*, ...) specifies the correlation constant, rho, for background intensity for each PM/MM probe pair. Choices are any value ≥ 0 and ≤ 1. Default is 0.7.

Expression = affygcrma(..., 'Method', *MethodValue*, ...) specifies the method to estimate the signal. Choices are 'MLE', a faster, ad hoc Maximum Likelihood Estimate method, or 'EB', a slower, more formal, empirical Bayes method. Default is 'MLE'.

Expression = affygcrma(..., 'TuningParam', *TuningParamValue*, ...) specifies the tuning parameter used by the estimate method. This tuning parameter sets the lower bound of signal values with positive probability. Choices are a positive value. Default is 5 (MLE) or 0.5 (EB).

Tip For information on determining a setting for this parameter, see Wu et al., 2004 on page 1-860.

Expression = affygcrma(..., 'GSBCorr', *GSBCorrValue*, ...) specifies whether to perform gene-specific binding (GSB) correction using probe affinity data. Choices are true (default) or false. If there is no probe affinity information, this property is ignored.

Expression = affygcrma(..., 'Median', *MedianValue*, ...) specifies the use of the median of the ranked values instead of the mean for normalization. Choices are true or false (default).

Expression = affygcrma(..., 'Output', *OutputValue*, ...) specifies the scale of the returned gene expression values. *OutputValue* can be:

- 'log'
- 'log2'
- 'log10'
- 'linear'
- @*functionname*

In the last instance, the data is transformed as defined by the function *functionname*. Default is 'log2'.

Expression = affygcrma(..., 'Showplot', *ShowplotValue*, ...) controls the display of a plot showing the $\log_2$ of mismatch (MM) probe intensity values from a specified chip (CEL file), versus that chip's MM probe affinities. The plot also shows the LOWESS fit for computing NSB data of the specified chip. Choices are true, false, or *I*, an integer specifying a chip. If set to true, the first chip is plotted. Default is:

- false — When return values are specified.
- true — When return values are not specified.

Expression = affygcrma(..., 'Verbose', *VerboseValue*, ...) controls the display of the status of the reading of files and GCRMA processing. Choices are true (default) or false.

Examples

The following example assumes that you have the `HG_U95Av2.CDF` library file stored at `D:\Affymetrix\LibFiles\HGGenome`, and that your current folder points to a location containing CEL files and a sequence file associated with this CDF library file. In this example, the `affygcrma` function reads all the CEL files and the sequence file in the current folder and a CDF file in a specified folder. It also performs GCRMA background adjustment, quantile normalization, and summarization procedures on the PM probe intensity values, and returns a DataMatrix object, containing the metadata and processed data.

```
Expression = affygcrma('*', 'HG_U95Av2.CDF','HG-U95Av2_probe_tab',...
                       'CDFPath', 'D:\Affymetrix\LibFiles\HGGenome');
```

Version History

Introduced in R2008b

References

[1] Naef, F., and Magnasco, M.O. (2003). Solving the Riddle of the Bright Mismatches: Labeling and Effective Binding in Oligonucleotide Arrays. Physical Review E *68*, 011906.

[2] Wu, Z., Irizarry, R.A., Gentleman, R., Murillo, F.M., and Spencer, F. (2004). A Model Based Background Adjustment for Oligonucleotide Expression Arrays. Journal of the American Statistical Association *99(468)*, 909–917.

[3] Wu, Z., and Irizarry, R.A. (2005). Stochastic Models Inspired by Hybridization Theory for Short Oligonucleotide Arrays. Proceedings of RECOMB 2004. J Comput Biol. *12(6)*, 882–93.

[4] Wu, Z., and Irizarry, R.A. (2005). A Statistical Framework for the Analysis of Microarray Probe-Level Data. Johns Hopkins University, Biostatistics Working Papers 73.

[5] Wu, Z., and Irizarry, R.A. (2003). A Model Based Background Adjustment for Oligonucleotide Expression Arrays. RSS Workshop on Gene Expression, Wye, England, `https://biosun01.biostat.jhsph.edu/%7Eririzarr/Talks/gctalk.pdf`.

[6] Speed, T. (2006). Background models and GCRMA. Lecture 10, Statistics 246, University of California Berkeley.

[7] Abd Rabbo, N.A., and Barakat, H.M. (1979). Estimation Problems in Bivariate Lognormal Distribution. Indian J. Pure Appl. Math *10(7)*, 815–825.

[8] Best, C.J.M., Gillespie, J.W., Yi, Y., Chandramouli, G.V.R., Perlmutter, M.A., Gathright, Y., Erickson, H.S., Georgevich, L., Tangrea, M.A., Duray, P.H., Gonzalez, S., Velasco, A., Linehan, W.M., Matusik, R.J., Price, D.K., Figg, W.D., Emmert-Buck, M.R., and Chuaqui, R.F. (2005). Molecular alterations in primary prostate cancer after androgen ablation therapy. Clinical Cancer Research *11*, 6823–6834.

[9] Irizarry, R.A., Hobbs, B., Collin, F., Beazer-Barclay, Y.D., Antonellis, K.J., Scherf, U., Speed, T.P. (2003). Exploration, Normalization, and Summaries of High Density Oligonucleotide Array Probe Level Data. Biostatistics. *4*, 249–264.

[10] Mosteller, F., and Tukey, J. (1977). Data Analysis and Regression (Reading, Massachusetts: Addison-Wesley Publishing Company), pp. 165–202.

See Also

`affyprobeaffinities` | `affyprobeseqread` | `affyrma` | `celintensityread` | `gcrma` | `gcrmabackadj` | `mafdr` | `mattest` | `quantilenorm` | `rmasummary`

affyinvarsetnorm

Perform rank invariant set normalization on probe intensities from multiple Affymetrix CEL or DAT files

Syntax

```
NormData = affyinvarsetnorm(Data)
[NormData, MedStructure] = affyinvarsetnorm(Data)

... affyinvarsetnorm(..., 'Baseline', BaselineValue, ...)
... affyinvarsetnorm(..., 'Thresholds', ThresholdsValue, ...)
... affyinvarsetnorm(..., 'StopPercentile', StopPercentileValue, ...)
... affyinvarsetnorm(..., 'RayPercentile', RayPercentileValue, ...)
... affyinvarsetnorm(..., 'Method', MethodValue, ...)
... affyinvarsetnorm(..., 'Showplot', ShowplotValue, ...)
```

Arguments

Data	Matrix of intensity values where each row corresponds to a perfect match (PM) probe and each column corresponds to an Affymetrix CEL or DAT file. (Each CEL or DAT file is generated from a separate chip. All chips should be of the same type.)
MedStructure	Structure of each column's intensity median before and after normalization, and the index of the column chosen as the baseline.
BaselineValue	Property to control the selection of the column index *N* from `Data` to be used as the baseline column. Default is the column index whose median intensity is the median of all the columns.
ThresholdsValue	Property to set the thresholds for the lowest average rank and the highest average rank, which are used to determine the invariant set. The rank invariant set is a set of data points whose proportional rank difference is smaller than a given threshold. The threshold for each data point is determined by interpolating between the threshold for the lowest average rank and the threshold for the highest average rank. Select these two thresholds empirically to limit the spread of the invariant set, but allow enough data points to determine the normalization relationship. *ThresholdsValue* is a 1-by-2 vector [*LT, HT*] where *LT* is the threshold for the lowest average rank and *HT* is threshold for the highest average rank. Values must be between `0` and `1`. Default is [`0.05, 0.005`].

StopPercentileValue	Property to stop the iteration process when the number of data points in the invariant set reaches *N* percent of the total number of data points. Default is `1`. **Note** If you do not use this property, the iteration process continues until no more data points are eliminated.
RayPercentileValue	Property to select the *N* percentage of the highest ranked invariant set of data points to fit a straight line through, while the remaining data points are fitted to a running median curve. The final running median curve is a piecewise linear curve. Default is `1.5`.
MethodValue	Property to select the smoothing method used to normalize the data. Enter `'lowess'` or `'runmedian'`. Default is `'lowess'`.
ShowplotValue	Property to control the plotting of two pairs of scatter plots (before and after normalization). The first pair plots baseline data versus data from a specified column (chip) from the matrix *Data*. The second is a pair of M-A scatter plots, which plots M (ratio between baseline and sample) versus A (the average of the baseline and sample). Enter either `'all'` (plot a pair of scatter plots for each column or chip) or specify a subset of columns (chips) by entering the column number(s) or a range of numbers.

Description

NormData = `affyinvarsetnorm(`*Data*`)` normalizes the values in each column (chip) of probe intensities in *Data* to a baseline reference, using the invariant set method. *NormData* is a matrix of normalized probe intensities from *Data*.

Specifically, `affyinvarsetnorm`:

- Selects a baseline index, typically the column whose median intensity is the median of all the columns.
- For each column, determines the proportional rank difference (*prd*) for each pair of ranks, *RankX* and *RankY*, from the sample column and the baseline reference.

  prd = abs(*RankX* - *RankY*)
- For each column, determines the invariant set of data points by selecting data points whose proportional rank differences (*prd*) are below *threshold*, which is a predetermined threshold for a given data point (defined by the *ThresholdsValue* property). It repeats the process until either no more data points are eliminated, or a predetermined percentage of data points is reached.

  The invariant set is data points with a *prd* < *threshold*.
- For each column, uses the invariant set of data points to calculate the lowess or running median smoothing curve, which is used to normalize the data in that column.

[*NormData*, *MedStructure*] = `affyinvarsetnorm(`*Data*`)` also returns a structure of the index of the column chosen as the baseline and each column's intensity median before and after normalization.

Note If *`Data`* contains NaN values, then *`NormData`* will also contain NaN values at the corresponding positions.

`... affyinvarsetnorm(..., 'PropertyName', PropertyValue, ...)` calls `affyinvarsetnorm` with optional properties that use property name/property value pairs. You can specify one or more properties in any order. Each *`PropertyName`* must be enclosed in single quotation marks and is case insensitive. These property name/property value pairs are as follows:

`... affyinvarsetnorm(..., 'Baseline', BaselineValue, ...)` lets you select the column index *`N`* from *`Data`* to be the baseline column. Default is the index of the column whose median intensity is the median of all the columns.

`... affyinvarsetnorm(..., 'Thresholds', ThresholdsValue, ...)` sets the thresholds for the lowest average rank and the highest average rank, which are used to determine the invariant set. The rank invariant set is a set of data points whose proportional rank difference is smaller than a given threshold. The threshold for each data point is determined by interpolating between the threshold for the lowest average rank and the threshold for the highest average rank. Select these two thresholds empirically to limit the spread of the invariant set, but allow enough data points to determine the normalization relationship.

`ThresholdsValue` is a 1-by-2 vector [*`LT, HT`*], where *`LT`* is the threshold for the lowest average rank and *`HT`* is threshold for the highest average rank. Values must be between `0` and `1`. Default is `[0.05, 0.005]`.

`... affyinvarsetnorm(..., 'StopPercentile', StopPercentileValue, ...)` stops the iteration process when the number of data points in the invariant set reaches *`N`* percent of the total number of data points. Default is `1`.

Note If you do not use this property, the iteration process continues until no more data points are eliminated.

`... affyinvarsetnorm(..., 'RayPercentile', RayPercentileValue, ...)` selects the *`N`* percentage of the highest ranked invariant set of data points to fit a straight line through, while the remaining data points are fitted to a running median curve. The final running median curve is a piecewise linear curve. Default is `1.5`.

`... affyinvarsetnorm(..., 'Method', MethodValue, ...)` selects the smoothing method for normalizing the data. When *`MethodValue`* is `'lowess'`, `affyinvarsetnorm` uses the lowess method. When *`MethodValue`* is `'runmedian'`, `affyinvarsetnorm` uses the running median method. Default is `'lowess'`.

`... affyinvarsetnorm(..., 'Showplot', ShowplotValue, ...)` plots two pairs of scatter plots (before and after normalization). The first pair plots baseline data versus data from a specified column (chip) from the matrix *`Data`*. The second is a pair of M-A scatter plots, which plots M (ratio between baseline and sample) versus A (the average of the baseline and sample). When *`ShowplotValue`* is `'all'`, `affyinvarsetnorm` plots a pair of scatter plots for each column or chip. When *`ShowplotValue`* is a number(s) or range of numbers, `affyinvarsetnorm` plots a pair of scatter plots for the indicated column numbers (chips).

Examples

Normalize Affymetrix data

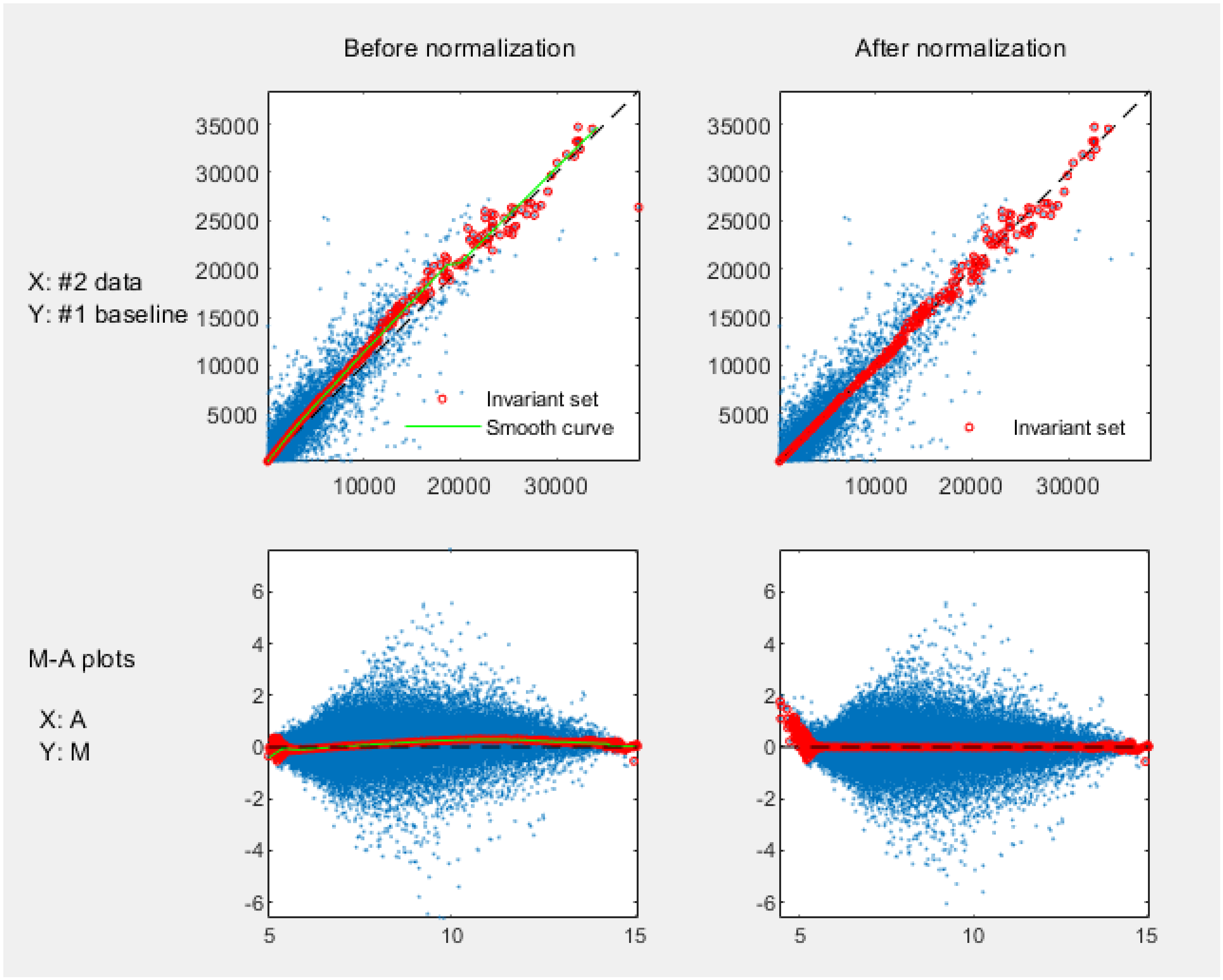
This example shows how to normalize affymetrix data. The `prostatecancerrawdata.mat` file used in the example contains data from Best et al., 2005.

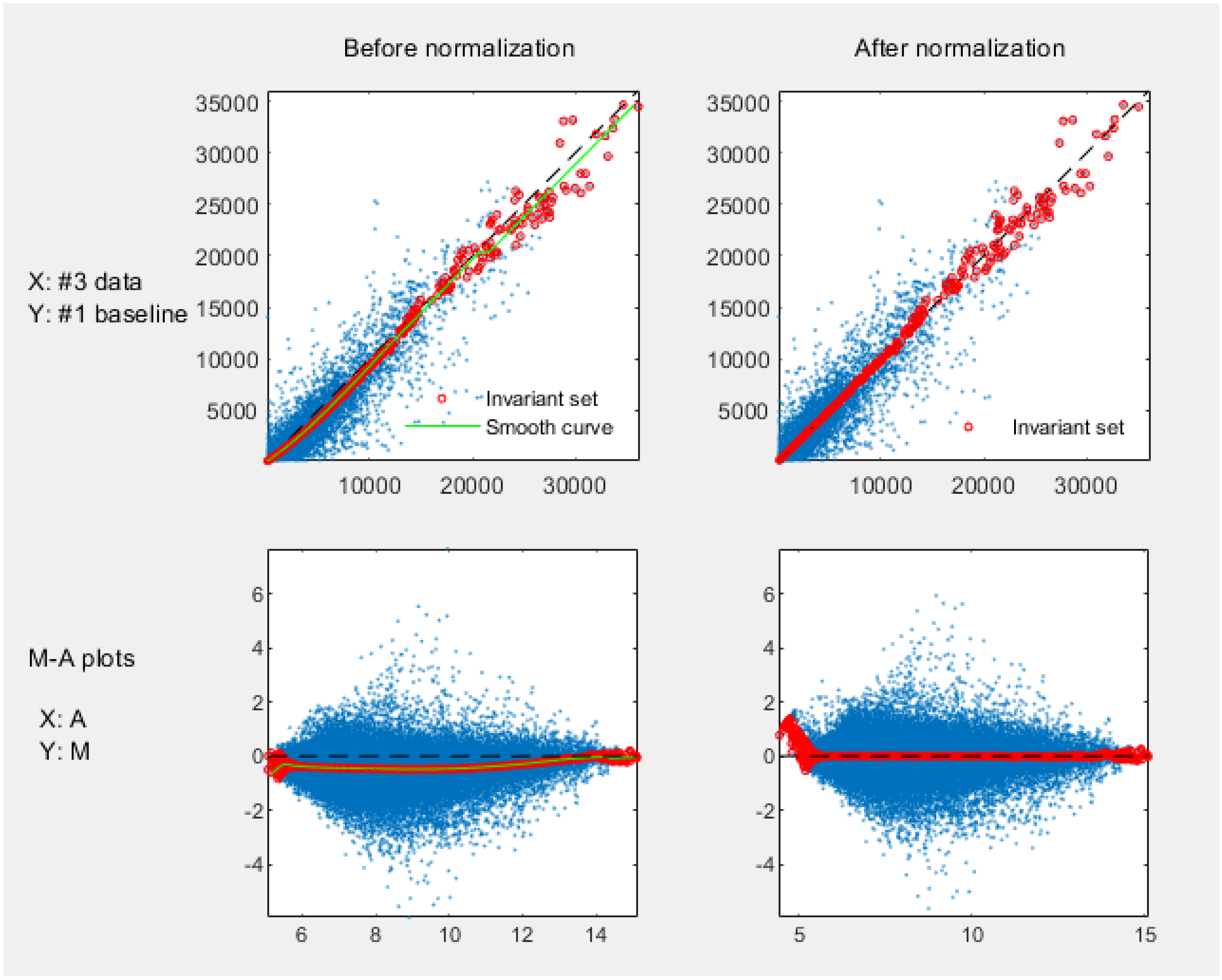
Load a MAT-file, included with the Bioinformatics Toolbox™ software, which contains Affymetrix data variables, including `pmMatrix` , a matrix of PM probe intensity values from multiple CEL files.

```
load prostatecancerrawdata
```

Normalize the data in pmMatrix and plot data from columns (chips) 2 and 3. Column 1 is the baseline.

```
NormMatrix = affyinvarsetnorm(pmMatrix, 'Showplot',[2 3]);
```





Version History

Introduced in R2006a

References

[1] Li, C., and Wong, W.H. (2001). Model-based analysis of oligonucleotide arrays: model validation, design issues and standard error application. Genome Biology *2(8)*: research0032.1-0032.11.

[2] https://sites.google.com/site/dchipsoft/

[3] Best, C.J.M., Gillespie, J.W., Yi, Y., Chandramouli, G.V.R., Perlmutter, M.A., Gathright, Y., Erickson, H.S., Georgevich, L., Tangrea, M.A., Duray, P.H., Gonzalez, S., Velasco, A., Linehan, W.M., Matusik, R.J., Price, D.K., Figg, W.D., Emmert-Buck, M.R., and Chuaqui, R.F. (2005). Molecular alterations in primary prostate cancer after androgen ablation therapy. Clinical Cancer Research *11*, 6823–6834.

See Also

affyread | celintensityread | mainvarsetnorm | malowess | manorm | quantilenorm | rmabackadj | rmasummary

affyprobeaffinities

Compute Affymetrix probe affinities from their sequences and MM probe intensities

Syntax

```
[AffinPM, AffinMM] = affyprobeaffinities(SequenceMatrix, MMIntensity)
[AffinPM, AffinMM, BaseProf] = affyprobeaffinities(SequenceMatrix,
MMIntensity)
[AffinPM, AffinMM, BaseProf, Stats] = affyprobeaffinities(SequenceMatrix,
MMIntensity)

... = affyprobeaffinities(SequenceMatrix, MMIntensity, ...'ProbeIndices',
ProbeIndicesValue, ...)
... = affyprobeaffinities(SequenceMatrix, MMIntensity, ...'Showplot',
ShowplotValue, ...)
```

Input Arguments

SequenceMatrix	An *N*-by-25 matrix of sequence information for the perfect match (PM) probes on an Affymetrix GeneChip array, where *N* is the number of probes on the array. Each row corresponds to a probe, and each column corresponds to one of the 25 sequence positions. Nucleotides in the sequences are represented by one of the following integers: • `0` — None • `1` — A • `2` — C • `3` — G • `4` — T **Tip** You can use the `affyprobeseqread` function to generate this matrix. If you have this sequence information in letter representation, you can convert it to integer representation using the `nt2int` function.
MMIntensity	Column vector containing mismatch (MM) probe intensities from a CEL file, generated from a single Affymetrix GeneChip array. Each row corresponds to a probe. **Tip** You can extract this column vector from the `MMIntensities` matrix returned by the `celintensityread` function.

ProbeIndicesValue	Column vector containing probe indexing information. Probes within a probe set are numbered 0 through *N* - 1, where *N* is the number of probes in the probe set. **Tip** You can use the `affyprobeseqread` function to generate this column vector.
ShowplotValue	Controls the display of a plot showing the affinity values of each of the four bases (A, C, G, and T) for each of the 25 sequence positions, for all probes on the Affymetrix GeneChip array. Choices are `true` or `false` (default).

Output Arguments

AffinPM	Column vector of PM probe affinities, computed from their probe sequences and MM probe intensities.
AffinMM	Column vector of MM probe affinities, computed from their probe sequences and MM probe intensities.
BaseProf	4-by-4 matrix containing the four parameters for a polynomial of degree 3, for each base, A, C, G, and T. Each row corresponds to a base, and each column corresponds to a parameter. These values are estimated from the probe sequences and intensities, and represent all probes on an Affymetrix GeneChip array.
Stats	Row vector containing four statistics in the following order: • R-square statistic • F statistic • p-value • Error variance

Description

[*AffinPM*, *AffinMM*] = `affyprobeaffinities`(*SequenceMatrix*, *MMIntensity*) returns a column vector of PM probe affinities and a column vector of MM probe affinities, computed from their probe sequences and MM probe intensities. Each row in *AffinPM* and *AffinMM* corresponds to a probe. NaN is returned for probes with no sequence information. Each probe affinity is the sum of position-dependent base affinities. For a given base type, the positional effect is modeled as a polynomial of degree 3.

[*AffinPM*, *AffinMM*, *BaseProf*] = `affyprobeaffinities`(*SequenceMatrix*, *MMIntensity*) also estimates affinity coefficients using multiple linear regression. It returns *BaseProf*, a 4-by-4 matrix containing the four parameters for a polynomial of degree 3, for each base, A, C, G, and T. Each row corresponds to a base, and each column corresponds to a parameter. These values are estimated from the probe sequences and intensities, and represent all probes on an Affymetrix GeneChip array.

[*AffinPM*, *AffinMM*, *BaseProf*, *Stats*] = `affyprobeaffinities`(*SequenceMatrix*, *MMIntensity*) also returns *Stats*, a row vector containing four statistics in the following order:

- R-square statistic
- F statistic
- p-value
- Error variance

`... = affyprobeaffinities(`*`SequenceMatrix`*`, `*`MMIntensity`*`, ...'`*`PropertyName`*`',` *`PropertyValue`*`, ...)` calls `affyprobeaffinities` with optional properties that use property name/property value pairs. You can specify one or more properties in any order. Each *`PropertyName`* must be enclosed in single quotation marks and is case insensitive. These property name/property value pairs are as follows:

`... = affyprobeaffinities(`*`SequenceMatrix`*`, `*`MMIntensity`*`, ...'ProbeIndices',` *`ProbeIndicesValue`*`, ...)` uses probe indices to normalize the probe intensities with the median of their probe set intensities.

Tip Use of the `ProbeIndices` property is recommended only if your *`MMIntensity`* data are not from a nonspecific binding experiment.

`... = affyprobeaffinities(`*`SequenceMatrix`*`, `*`MMIntensity`*`, ...'Showplot',` *`ShowplotValue`*`, ...)` controls the display of a plot of the probe affinity base profile. Choices are `true` or `false` (default).

Examples

Calculate Affymetrix probe affinities

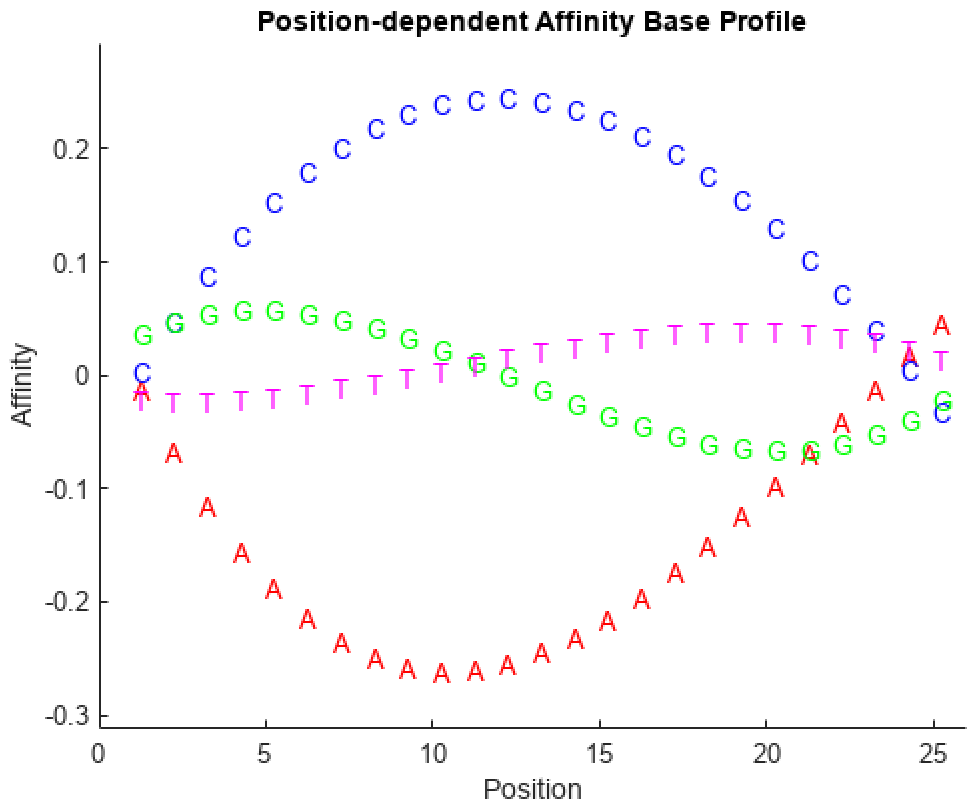
This example shows how to calculate Affymetrix PM and MM probe affinities from their sequences and MM probe intensities.

Load the MAT-file, included with the Bioinformatics Toolbox™ software, that contains Affymetrix data from a prostate cancer study. The variables in the MAT-file include `seqMatrix` , a matrix containing sequence information for PM probes, `mmMatrix` , a matrix containing MM probe intensity values, and `probeIndices` , a column vector containing probe indexing information.

```
load prostatecancerrawdata
```

Compute the Affymetrix PM and MM probe affinities from their sequences and MM probe intensities, and also plot the affinity values of each of the four bases (A, C, G, and T) for each of the 25 sequence positions, for all probes on the Affymetrix GeneChip array.

```
[apm, amm] = affyprobeaffinities(seqMatrix, mmMatrix(:,1),...
             'ProbeIndices', probeIndices, 'showplot', true);
```



The prostatecancerrawdata.mat file used in this example contains data from Best et al., 2005.

Version History

Introduced in R2007a

References

[1] Naef, F., and Magnasco, M.O. (2003). Solving the Riddle of the Bright Mismatches: Labeling and Effective Binding in Oligonucleotide Arrays. Physical Review E *68*, 011906.

[2] Wu, Z., Irizarry, R.A., Gentleman, R., Murillo, F.M. and Spencer, F. (2004). A Model Based Background Adjustment for Oligonucleotide Expression Arrays. Journal of the American Statistical Association *99(468)*, 909–917.

[3] Best, C.J.M., Gillespie, J.W., Yi, Y., Chandramouli, G.V.R., Perlmutter, M.A., Gathright, Y., Erickson, H.S., Georgevich, L., Tangrea, M.A., Duray, P.H., Gonzalez, S., Velasco, A., Linehan, W.M., Matusik, R.J., Price, D.K., Figg, W.D., Emmert-Buck, M.R., and Chuaqui, R.F. (2005). Molecular alterations in primary prostate cancer after androgen ablation therapy. Clinical Cancer Research *11*, 6823–6834.

See Also

`affygcrma` | `affyprobeseqread` | `affyread` | `celintensityread` | `probelibraryinfo`

affyprobeseqread

Read data file containing probe sequence information for Affymetrix GeneChip array

Syntax

```
Struct = affyprobeseqread(SeqFile, CDFFile)

Struct = affyprobeseqread(SeqFile, CDFFile, ...'SeqPath', SeqPathValue, ...)
Struct = affyprobeseqread(SeqFile, CDFFile, ...'CDFPath', CDFPathValue, ...)
Struct = affyprobeseqread(SeqFile, CDFFile, ...'SeqOnly', SeqOnlyValue, ...)
```

Input Arguments

SeqFile	Character vector or string specifying a file name of a sequence file (tab-separated or FASTA) that contains the following information for a specific type of Affymetrix GeneChip array: • Probe set IDs • Probe *x*-coordinates • Probe *y*-coordinates • Probe sequences in each probe set • Affymetrix GeneChip array type (FASTA file only) The sequence file (tab-separated or FASTA) must be on the MATLAB search path or in the Current Folder (unless you use the `SeqPath` property). In a tab-separated file, each row represents a probe; in a FASTA file, each header represents a probe.
CDFFile	Either of the following: • Character vector or string specifying a file name of an Affymetrix CDF library file, which contains information that specifies which probe set each probe belongs to on a specific type of Affymetrix GeneChip array. The CDF library file must be on the MATLAB search path or in the MATLAB Current Folder (unless you use the `CDFPath` property). • CDF structure, such as returned by the `affyread` function, which contains information that specifies which probe set each probe belongs to on a specific type of Affymetrix GeneChip array. **Caution** Make sure that *SeqFile* and *CDFFile* contain information for the same type of Affymetrix GeneChip array.
SeqPathValue	Character vector or string specifying a folder or path and folder where *SeqFile* is stored.
CDFPathValue	Character vector or string specifying a folder or path and folder where *CDFFile* is stored.

`SeqOnlyValue`	Controls the return of a structure, *`Struct`*, with only one field, `SequenceMatrix`. Choices are `true` or `false` (default).

Output Arguments

`Struct`	MATLAB structure containing the following fields: • `ProbeSetIDs` • `ProbeIndices` • `SequenceMatrix`

Description

`Struct` `= affyprobeseqread(`*`SeqFile`*`, `*`CDFFile`*`)` reads the data from files *`SeqFile`* and *`CDFFile`*, and stores the data in the MATLAB structure *`Struct`*, which contains the following fields.

Field	Description
`ProbeSetIDs`	Cell array containing the probe set IDs from the Affymetrix CDF library file.
`ProbeIndices`	Column vector containing probe indexing information. Probes within a probe set are numbered 0 through *N* - 1, where *N* is the number of probes in the probe set.
`SequenceMatrix`	An *N*-by-25 matrix of sequence information for the perfect match (PM) probes on the Affymetrix GeneChip array, where *N* is the number of probes on the array. Each row corresponds to a probe, and each column corresponds to one of the 25 sequence positions. Nucleotides in the sequences are represented by one of the following integers: • `0` — None • `1` — A • `2` — C • `3` — G • `4` — T **Note** Probes without sequence information are represented in `SequenceMatrix` as a row containing all `0`s. **Tip** You can use the `int2nt` function to convert the nucleotide sequences in `SequenceMatrix` to letter representation.

`Struct` `= affyprobeseqread(`*`SeqFile`*`, `*`CDFFile`*`, ...'`*`PropertyName`*`', `*`PropertyValue`*`, ...)` calls `affyprobeseqread` with optional properties that use property name/property value pairs. You can specify one or more properties in any order. Each *`PropertyName`* must be enclosed in single quotation marks and is case insensitive. These property name/property value pairs are as follows:

`Struct` `= affyprobeseqread(`*`SeqFile`*`,` *`CDFFile`*`, ...'SeqPath',` *`SeqPathValue`*`, ...)` lets you specify a path and folder where *`SeqFile`* is stored.

`Struct` `= affyprobeseqread(`*`SeqFile`*`,` *`CDFFile`*`, ...'CDFPath',` *`CDFPathValue`*`, ...)` lets you specify a path and folder where *`CDFFile`* is stored.

`Struct` `= affyprobeseqread(`*`SeqFile`*`,` *`CDFFile`*`, ...'SeqOnly',` *`SeqOnlyValue`*`, ...)` controls the return of a structure, *`Struct`*, with only one field, `SequenceMatrix`. Choices are `true` or `false` (default).

Examples

1 Read the data from a FASTA file and associated CDF library file, assuming both are located on the MATLAB search path or in the Current Folder.

```
S1 = affyprobeseqread('HG-U95A_probe_fasta', 'HG_U95A.CDF');
```

2 Read the data from a tab-separated file and associated CDF structure, assuming the tab-separated file is located in the specified folder and the CDF structure is in your MATLAB Workspace.

```
S2 = affyprobeseqread('HG-U95A_probe_tab',hgu95aCDFStruct,...
     'seqpath','C:\Affymetrix\SequenceFiles\HGGenome');
```

3 Access the nucleotide sequences of the first probe set (rows 1 through 20) in the `SequenceMatrix` field of the `S2` structure.

```
seq = int2nt(S2.SequenceMatrix(1:20,:))
```

Version History

Introduced in R2007a

See Also

`affygcrma` | `affyinvarsetnorm` | `affyread` | `celintensityread` | `int2nt` | `probelibraryinfo` | `probesetlookup` | `probesetplot` | `probesetvalues`

affyread

Read microarray data from Affymetrix GeneChip file

Syntax

```
AffyStruct = affyread(File)
AffyStruct = affyread(File, LibraryPath)
```

Description

`AffyStruct = affyread(File)` reads an Affymetrix file and creates a MATLAB structure. The `affyread` function can read Affymetrix EXP, DAT, CEL, CLF, BGP, CDF, and GIN files associated with Affymetrix GeneChip arrays for expression, genotyping (SNP), or resequencing assays. It can read Affymetrix CHP files associated with Affymetrix GeneChip arrays for expression assays only.

`AffyStruct = affyread(File, LibraryPath)` specifies the path and folder of a CDF or GIN library file.

Input Arguments

`File`

Character vector or string specifying a file name or a path and file name of one of the following Affymetrix file types associated with Affymetrix GeneChip arrays for expression, genotyping (SNP), or resequencing assays. However, if the file name is for a CHP file, it must be associated with an Affymetrix GeneChip array for an expression assay.

- **EXP** — Data file containing information about experimental conditions and protocols.
- **DAT** — Data file containing raw image data (pixel intensity values).
- **CEL** — Data file containing information about the intensity values of the individual probes.
- **CHP** — Data file containing summary information of the probe sets, including intensity values.
- **CLF** — Cell layout file that maps probe IDs to a location (*x*- and *y*-coordinates) in the CEL file.
- **BGP** — Background probe file that lists the probes to use for background correction.
- **CDF** — Library file containing information about which probes belong to which probe set.
- **GIN** — Library file containing information about the probe sets, such as the gene name associated with the probe set.

If you specify only a file name, put that file on the MATLAB search path or in the current folder. If you specify only a file name of a CDF or GIN library file, you can specify the path and folder in the *`LibraryPath`* input argument.

`LibraryPath`

Character vector or string specifying the path and folder of a:

- CDF library file associated with *`File`* when *`File`* is a CHP file

- CDF library file when *`File`* is a CDF file
- GIN library file when *`File`* is a GIN file

Note If you do not specify *`LibraryPath`* when reading a CHP file, `affyread` looks in the current folder for the CDF file. If it does not find the CDF file, it still reads the CHP file. However, it omits the probe set names and types from the return value, *`AffyStruct`*.

Output Arguments

`AffyStruct`

MATLAB structure containing information from an Affymetrix data or library file, for expression, genotyping (SNP), or resequencing assay types.

The following tables describe the fields in *`AffyStruct`* for the different Affymetrix file types.

EXP, DAT, CEL, CHP, CLF, BGP, CDF, and GIN Files

Field	Description
`Name`	File name.
`DataPath`	Path and folder of the file.
`LibPath`	Path and folder of the CDF and GIN library files associated with the file you are reading.
`FullPathName`	Path and folder of the file.
`ChipType`	Name of the Affymetrix GeneChip array (for example, DrosGenome1 or HG-Focus).
`Date` or `CreateDate`	File creation date.

EXP File

Field	Description

`ChipLot`
`Operator`
`SampleType`
`SampleDesc`
`Project`
`Comments`
`Reagents`
`ReagentLot`
`Protocol`
`Station`
`Module`
`HybridizeDate`
`ScanPixelSize`
`ScanFilter`
`ScanDate`
`ScannerID`
`NumberOfScans`
`ScannerType`
`NumProtocolSteps`
`ProtocolSteps`
	Information about experimental conditions and protocols captured by the Affymetrix software.

DAT File

Field	Description
NumPixelsPerRow	Number of pixels per row in the image created from the GeneChip array (number of columns).
NumRows	Number of rows in the image created from the GeneChip array.
MinData	Minimum intensity value in the image created from the GeneChip array.
MaxData	Maximum intensity value in the image created from the GeneChip array.
PixelSize	Size of one pixel in the image created from the GeneChip array.
CellMargin	Size of gaps between cells in the image created from the GeneChip array.
ScanSpeed	Speed of the scanner used to create the image.
ScanDate	Date the scan was performed.
ScannerID	Name of the scanning device used.
UpperLeftX UpperLeftY UpperRightX UpperRightY LowerLeftX LowerLeftY LowerRightX LowerRightY	Pixel coordinates of the scanned image.
ServerName	Not used.
Image	A `NumRows`-by-`NumPixelsPerRow` image of the scanned GeneChip array.

CEL File

Field	Description
`FileVersion`	Version of the CEL file format.
`Algorithm`	Algorithm used in the image-processing step that converts from DAT format to CEL format.
`AlgParams`	Character vector containing parameters used by the algorithm in the image-processing step.
`NumAlgParams`	Number of parameters in `AlgParams`.
`CellMargin`	Size of gaps between cells in the image created from the GeneChip array, used for computing the intensity values of the cells.
`Rows`	Number of rows of probes.
`Cols`	Number of columns of probes.
`NumMasked`	Number of masked probes, which are not used in subsequent processing.
`NumOutliers`	Number of cells identified as outliers (extremely high or extremely low intensity) by the image-processing step.
`NumProbes`	Number of probes (`Rows` * `Cols`) on the GeneChip array.

`UpperLeftX`
`UpperLeftY`
`UpperRightX`
`UpperRightY`
`LowerLeftX`
`LowerLeftY`
`LowerRightX`
`LowerRightY`
	Pixel coordinates of the scanned image.

`ProbeColumnNames`	Cell array containing the eight column names in the `Probes` field: • `PosX` — *x*-coordinate of the cell • `PosY` — *y*-coordinate of the cell • `Intensity` — Intensity value of the cell • `StdDev` — Standard deviation of intensity value • `Pixels` — Number of pixels in the cell • `Outlier` — True/false flag indicating if the cell was marked as an outlier • `Masked` — True/false flag indicating if the cell was masked • `ProbeType` — Integer indicating the probe type (for example, `1` = expression)
`Probes`	`NumProbes`-by-8 array of information about the individual probes, including intensity values. The `ProbeColumnNames` field contains the column names of this array.

CHP File

Field	Description
AssayType	Type of assay associated with the GeneChip array (for example, Expression, Genotyping, or Resequencing).
CellFile	File name of the CEL file from which the CHP file was created.
Algorithm	Algorithm used to convert from CEL format to CHP format.
AlgVersion	Version of the algorithm used to create the CHP file.
NumAlgParams	Number of parameters in AlgParams.
AlgParams	Character vector containing parameters used in steps required to create the CHP file (for example, background correction).
NumChipSummary	Number of entries in ChipSummary.
ChipSummary	Summary information for the GeneChip array, including background average, standard deviation, max, and min.
BackgroundZones	Structure containing information about the zones used in the background adjustment step.
Rows	Number of rows of probes.
Cols	Number of columns of probes.
NumProbeSets	Number of probe sets on the GeneChip array.
NumQCProbeSets	Number of QC probe sets on the GeneChip array.

Field	Description
ProbeSets (Expression GeneChip array)	NumProbeSets-by-1 structure array containing information for each expression probe set, including the following fields: • Name — Name of the probe set. • ProbeSetType — Type of the probe set. • CompDataExists — True/false flag indicating if the probe set has additional computed information. • NumPairs — Number of probe pairs in the probe set. • NumPairsUsed — Number of probe pairs in the probe set used for calculating the probe set signal (not masked). • Signal — Summary intensity value for the probe set. • Detection — Indicator of statistically significant difference between the intensity value of the PM probes and the intensity value of the MM probes in a single probe set (Present, Absent, or Marginal). • DetectionPValue — P-value for the Detection indicator. • CommonPairs — When CompDataExists is true, contains the number of common pairs between the experiment and the baseline after the removal of outliers and masked probes. • SignalLogRatio — When CompDataExists is true, contains the change in signal between the experiment and baseline. • SignalLogRatioLow — When CompDataExists is true, contains the lowest ratios of probes between the experiment and the baseline. • SignalLogRatioHigh — When CompDataExists is true, contains the highest ratios of probes between the experiment and the baseline. • Change — When CompDataExists is true, describes how the probe changes versus a baseline experiment. Choices are Increase, Marginal Increase, No Change, Decrease, or Marginal Decrease. • ChangePValue — When CompDataExists is true, contains the p-value associated with Change.

Field	Description
`ProbeSets` (Genotyping GeneChip array)	`NumProbeSets`-by-1 structure array containing information for each genotyping probe set, including the following fields: • `Name` — Name of the probe set. • `AlleleCall` — Allele that is present for the probe set. Possibilities are `AA` (homozygous for the major allele), `AB` (heterozygous for the major and minor allele), `BB` (homozygous for the minor allele), or `NoCall` (unable to determine allele). • `Confidence` — Measure of the accuracy of the allele call. • `RAS1` — Relative Allele Signal 1 for the SNP site, which is calculated using sense probes. • `RAS2`— Relative Allele Signal 2 for the SNP site, which is calculated using antisense probes. • `PValueAA` — p-value for an `AA` call. • `PValueAB` — p-value for an `AB` call. • `PValueBB` — p-value for a `BB` call. • `PValueNoCall` — p-value for a `NoCall` call.
`ProbeSets` (Resequencing GeneChip array)	`NumProbeSets`-by-1 structure array containing information for each resequencing probe set, including the following fields: • `CalledBases` — 1-by-`NumProbeSets` character vector containing the bases called by the resequencing algorithm. Possible values are `a`, `c`, `g`, `t`, and `n`. • `Scores` — 1-by-`NumProbeSets` array containing the score associated with each base call.

CLF File

Field	Description
`LibSetName`	Name of a collection of related library files for a given chip. There is only one `LibSetName` for a CLF file. For example, PGF and CLF files intended for use together must have the same `LibSetName`.
`LibSetVersion`	Version of a collection of related library files for a given chip. There is only one `LibSetVersion` for a CLF file. For example, PGF and CLF files intended for use together must have the same `LibSetVersion`.
`GUID`	Unique identifier for the CLF file.
`CLFFormatVersion`	Version of the CLF file format.
`Rows`	Number of rows in the CEL file. **Note** The CLF file is 1 base, which means the first row and column are designated 1,1, not 0,0.
`Cols`	Number of columns in the CEL file. **Note** The CLF file is 1 base, which means the first row and column are designated 1,1, not 0,0.
`StartID`	Starting number for the numbering of elements in the CLF file. **Tip** This information is useful when numbering does not start with 1.
`EndID`	Ending number for the numbering of elements in the CLF file. **Tip** This information is useful when numbering does not start with 1 and/or there are gaps in the numbering.
`Order`	Order in which the probe IDs are numbered in the CEL file, either `'row_major'` or `'col_major'`.
`DataColNames`	Names of the columns in the CEL file that contain data.
`Data`	If the numbering of elements in the CLF file is sequential, this field contains a function handle that calculates the *x*- and *y*- coordinates of each element in the file from the probe ID. If the numbering of elements in the CLF file is not sequential, this field contains a matrix indicating the number value of each element in the file.

BGP File

Field	Description
`LibSetName`	Name of a collection of related library files for a given chip. There is only one `LibSetName` for a BGP file.
`LibSetVersion`	Version of a collection of related library files for a given chip. There is only one `LibSetVersion` for a BGP file.
`GUID`	Unique identifier for a BGP file.
`ExecGUID`	Information about the algorithm used to generate the BGP file.
`ExecVersion`	
`Cmd`	
`Data`	Structure containing the following fields: • `probe_id` — ID of the probe to use for background correction. • `probeset_id` — ID of the probe set in the PGF file to which the probe belongs. • `type` — Classification information for the probe. • `gc_count` — Combined number of G and C bases in the probe. • `probe_length`— Length of the probe in base pairs. • `interrogation_position` — Interrogation position of the probe. It is typically 13 for 25-mer PM/MM probes. • `probe_sequence` — Sequence of the probe on the array, going in the direction from array surface to solution. For most standard Affymetrix arrays, this direction is from 3' to 5'. For example, for a sense target (st) probe (see the `probe_type` field), complement the sequence in this field before looking for matches to transcript sequences. For an antisense target (at), reverse this sequence. • `atom_id` — ID of the atom to which the probe belongs. • `x` — Column coordinate of the probe in the CEL file. • `y` — Row coordinate of the probe in the CEL file. • `probeset_type` — Classification information for the probe set, such as control, affx, or spike. This type information can include multiple classifications and can also be nested. • `probe_type` — Classification information for the probe, such as pm (perfect match), mm (mismatch), st (sense target), or at (antisense target). This type information can include multiple classifications and can also be nested.

CDF File

Field	Description
`Rows`	Number of rows of probes.
`Cols`	Number of columns of probes.
`NumProbeSets`	Number of probe sets on the GeneChip array.
`NumQCProbeSets`	Number of QC probe sets on the GeneChip array.
`ProbeSetColumnNames`	Cell array containing the six column names in the `ProbePairs` field in the `ProbeSets` array: • `GroupNumber` — Number identifying the group to which the probe pair belongs. For expression arrays, this value is always `1`. For genotyping arrays, this value is typically `1` (allele A, sense), `2` (allele B, sense), `3` (allele A, antisense), or `4` (allele B, antisense). • `Direction` — Number identifying the direction of the probe pair. `1` = sense and `2` = antisense. • `PMPosX` — *x*-coordinate of the perfect match probe. • `PMPosY` — *y*-coordinate of the perfect match probe. • `MMPosX` — *x*-coordinate of the mismatch probe. • `MMPosY` — *y*-coordinate of the mismatch probe.
`ProbeSets`	`NumProbeSets`-by-1 structure array containing information for each probe set, including the following fields: • `Name` — Name of the probe set. • `ProbeSetType` — Type of the probe set. • `CompDataExists` — True/false flag indicating if the probe set has additional computed information. • `NumPairs` — Number of probe pairs in the probe set. • `NumQCProbes` — Number of QC probes in the probe set. • `QCType` — Type of QC probes. • `GroupNames` — Name of the group to which the probe set belongs. For expression arrays, this field contains the name of the probe set. For genotyping arrays, this field contains the name of the alleles, for example `{'A' 'C' 'A' 'C'}`. • `ProbePairs` — `NumPairs`-by-6 array of information about the probe pairs. The column names of this array are contained in the `ProbeSetColumnNames` field.

GIN File

Field	Description
`Version`	GIN file format version.
`ProbeSetName`	Probe set ID/name.
`ID`	Identifier for the probe set (gene ID).
`Description`	Description of the probe set.
`SourceNames`	Source or sources of the probe sets.
`SourceURL`	Source URL or URLs for the probe sets.
`SourceID`	Vector of numbers specifying which `SourceNames` or `SourceURL` each probe set is associated with.

Examples

Visualize Microarray Data

This example shows how to read and visualize microarray data from Affymetrix® GeneChip® file.

You need some sample data files from here. This example uses the sample data from the *E. coli* Antisense Genome Array. Extract the data files from the DTT archive using the Data Transfer Tool.

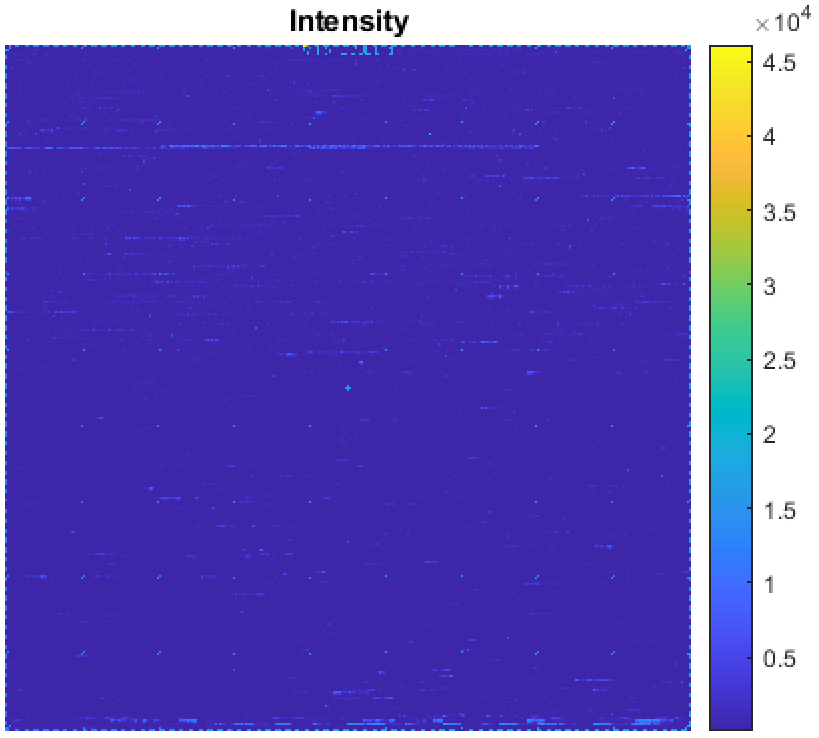
You also need to download the corresponding library files for the sample. For this example, Ecoli_ASv2.CDF and Ecoli_ASv2.GIN are used as for the *E. coli* Antisense Genome Array. You may already have these files if you have any Affymetrix GeneChip software installed on your machine. If not, get the library files by downloading and unzipping the *E. coli* Antisense Genome Array zip file.

Read the contents of a CEL file into a MATLAB structure.

```
celStruct = affyread('Ecoli-antisense-121502.CEL');
```

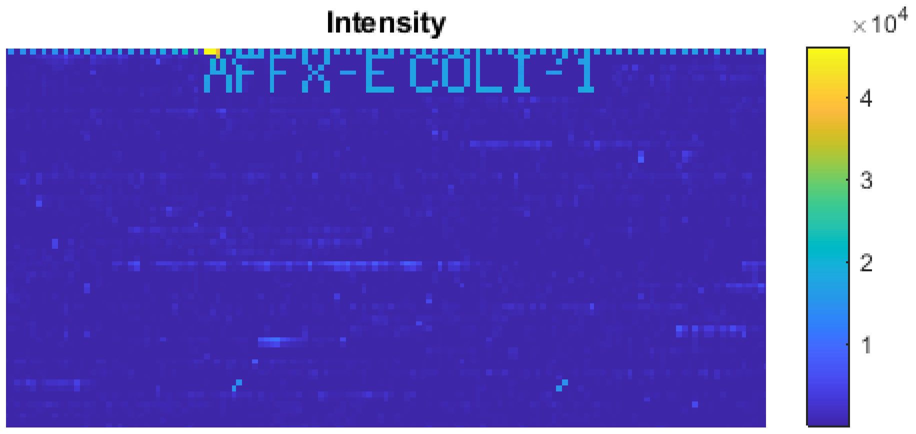
Display a spatial plot of the probe intensities.

```
maimage(celStruct, 'Intensity')
```



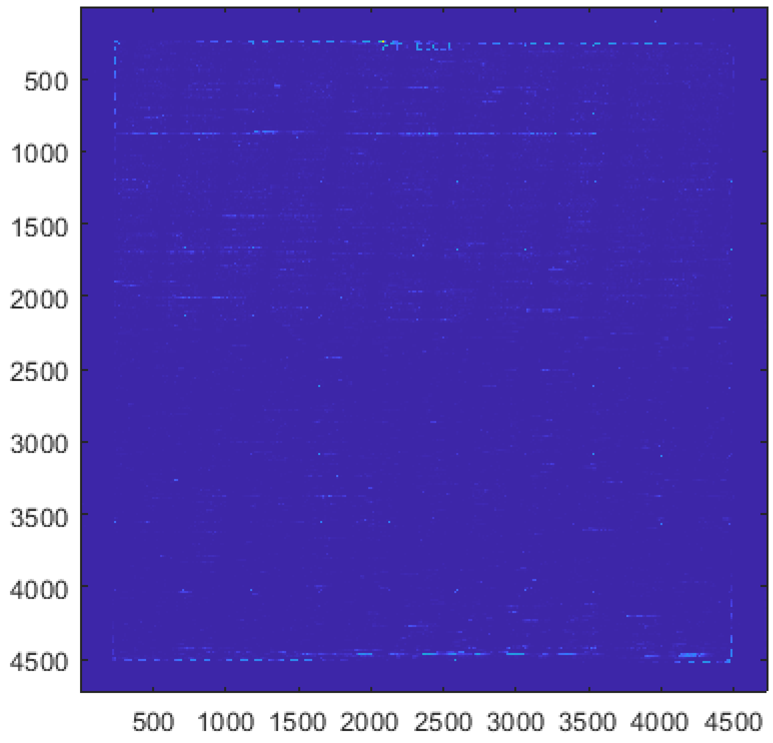
Zoom in on a specific region of the plot.

```
axis([200 340 0 70])
```



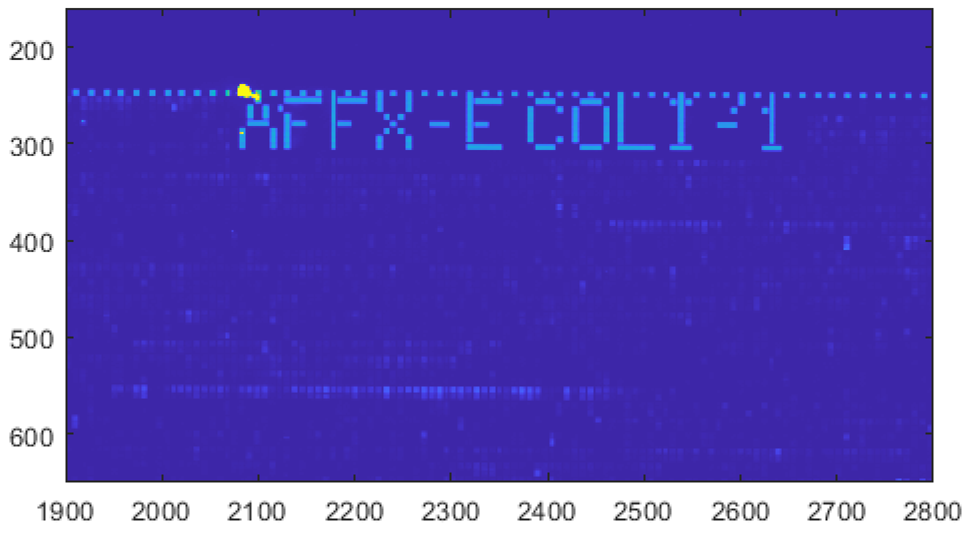
Read the contents of a DAT file into a MATLAB structure. Display the raw image data, and then use the `axis image` command to set the correct aspect ratio.

```
datStruct = affyread('Ecoli-antisense-121502.dat');
imagesc(datStruct.Image)
axis image
```



Zoom in on a specific region of the plot.

```
axis([1900 2800 160 650])
```



Read the contents of a CHP file into a MATLAB structure, specifying the location of the associated CDF library file. Then extract information for probe set 3315278.

```
chpStruct = affyread('Ecoli-antisense-121502.chp','C:\LibFiles\');
geneName = probesetlookup(chpStruct,'3315278')
```

```
geneName = 

  struct with fields:

      Identifier: '3315278'
     ProbeSetName: 'argG_b3172_at'
         CDFIndex: 5213
         GINIndex: 3074
      Description: '/start=3316278 /end=3317621 /direction=+ /description=argininosuccinate synthe
           Source: 'NCBI EColi Genome'
        SourceURL: 'http://www.ncbi.nlm.nih.gov/cgi-bin/Entrez/altvik?gi=115&db=g&from=3315278'
```

Version History

Introduced before R2006a

See Also

`affyrma` | `affygcrma` | `affysnpannotread` | `affysnpintensitysplit` | `agferead` | `celintensityread` | `geoseriesread` | `gprread` | `ilmnbsread` | `probelibraryinfo` | `probesetlookup` | `probesetplot` | `probesetvalues` | `sptread`

Topics
“Working with Affymetrix Data”
“Preprocessing Affymetrix Microarray Data at the Probe Level”
“Analyzing Affymetrix SNP Arrays for DNA Copy Number Variants”

affyrma

Perform Robust Multi-array Average (RMA) procedure on Affymetrix microarray probe-level data

Syntax

```
Expression = affyrma(CELFiles, CDFFile)
Expression = affyrma(ProbeStructure)

Expression = affyrma(CELFiles, CDFFile, ...'CELPath', CELPathValue, ...)
Expression = affyrma(CELFiles, CDFFile, ...'CDFPath', CDFPathValue, ...)
Expression = affyrma(..., 'Method', MethodValue, ...)
Expression = affyrma(..., 'Truncate', TruncateValue, ...)
Expression = affyrma(..., 'Median', MedianValue, ...)
Expression = affyrma(..., 'Output', OutputValue, ...)
Expression = affyrma(..., 'Showplot', ShowplotValue, ...)
Expression = affyrma(..., 'Verbose', VerboseValue, ...)
```

Input Arguments

`CELFiles`	Any of the following: • Character vector or string specifying a single CEL file name. • `'*'`, which reads all CEL files in the current folder. • `' '`, which opens the Select CEL Files dialog box from which you select the CEL files. From this dialog box, you can press and hold **Ctrl** or **Shift** while clicking to select multiple CEL files. • Cell array of character vectors or string vector containing CEL file names.
`CDFFile`	Either of the following: • Character vector or string specifying a CDF file name. • `' '`, which opens the Select CDF File dialog box from which you select the CDF file.
`ProbeStructure`	MATLAB structure containing information from the CEL files, including probe intensities, probe indices, and probe set IDs, returned by the `celintensityread` function.
`CELPathValue`	Character vector or string specifying the path and folder where the files specified in *`CELFiles`* are stored.
`CDFPathValue`	Character vector or string specifying the path and folder where the file specified in *`CDFFile`* is stored.
`MethodValue`	Specifies the estimation method for the background adjustment model parameters. Choices are `'RMA'` (to use estimation method described by Bolstad, 2005) or `'MLE'` (to estimate the parameters using maximum likelihood). Default is `'RMA'`.

TruncateValue	Specifies the background noise model. Choices are `true` (use a truncated Gaussian distribution) or `false` (use a nontruncated Gaussian distribution). Default is `true`.
MedianValue	Specifies the use of the median of the ranked values instead of the mean for normalization. Choices are `true` or `false` (default).
OutputValue	Specifies the scale of the returned gene expression values. Choices are: • `'log'` • `'log2'` • `'log10'` • `'linear'` • `@`*`functionname`* In the last instance, the data is transformed as defined by the function *`functionname`*. Default is `'log2'`.
ShowplotValue	Controls the plotting of a histogram showing the distribution of PM probe intensity values (blue) and the convoluted probability distribution function (red), with estimated parameters mu, sigma and alpha. Enter either `'all'` (plot a histogram for each column or chip) or specify a subset of columns (chips) by entering the column number, list of numbers, or range of numbers. For example: • `(..., 'Showplot', 3, ...)` plots the intensity values in column 3. • `(..., 'Showplot', [3,5,7], ...)` plots the intensity values in columns 3, 5, and 7. • `(..., 'Showplot', 3:9, ...)` plots the intensity values in columns 3 to 9.
VerboseValue	Controls the display of the status of the reading of files and RMA processing. Choices are `true` (default) or `false`.

Output Arguments

Expression	DataMatrix object on page 1-734 containing the $\log_2$ based gene expression values that have been background adjusted, normalized, and summarized using the Robust Multi-array Average (RMA) procedure. Each row in *`Expression`* corresponds to a gene (probe set), and each column corresponds to an Affymetrix CEL file.

Description

`Expression` `= affyrma(`*`CELFiles`*`,` *`CDFFile`*`)` reads the specified Affymetrix CEL files and the associated CDF library file (created from Affymetrix GeneChip arrays for expression or genotyping assays), processes the probe intensity values using RMA background adjustment, quantile normalization, and summarization procedures, then returns *`Expression`*, a DataMatrix object on

page 1-734 containing the $\log_2$ based gene expression values in a matrix, the probe set IDs as row names, and the CEL file names as column names. Note that each row in `Expression` corresponds to a gene (probe set), and each column corresponds to an Affymetrix CEL file. (Each CEL file is generated from a separate chip. All chips should be of the same type.)

`CELFiles` is a character vector, string, string vector, or cell array of character vectors containing CEL file names. *`CDFFile`* is a character vector or string specifying a CDF file name. If you set *`CELFiles`* to `'*'`, then it reads all CEL files in the current folder. If you set *`CELFiles`* to `' '`, then it opens the Select CEL Files dialog box from which you select the CEL files.

Note For details on the reading of files and RMA processing, see `celintensityread`, `rmabackadj`, `quantilenorm`, and `rmasummary`.

`Expression` `= affyrma(`*`ProbeStructure`*`)` uses RMA background adjustment, quantile normalization, and summarization procedures to process the probe intensity values in *`ProbeStructure`*, a MATLAB structure containing information from the CEL files, including probe intensities, probe indices, and probe set IDs, returned by the `celintensityread` function, and returns *`Expression`*.

`Expression` `= affyrma(..., '`*`PropertyName`*`',` *`PropertyValue`*`, ...)` calls `affyrma` with optional properties that use property name/property value pairs. You can specify one or more properties in any order. Each *`PropertyName`* must be enclosed in single quotation marks and is case insensitive. These property name/property value pairs are as follows:

`Expression` `= affyrma(`*`CELFiles`*`,` *`CDFFile`*`, ...'CELPath',` *`CELPathValue`*`, ...)` specifies a path and folder where the files specified by *`CELFiles`* are stored.

`Expression` `= affyrma(`*`CELFiles`*`,` *`CDFFile`*`, ...'CDFPath',` *`CDFPathValue`*`, ...)` specifies a path and folder where the file specified by *`CDFFile`* is stored.

`Expression` `= affyrma(..., 'Method',` *`MethodValue`*`, ...)` specifies the estimation method for the background adjustment model parameters. When *`MethodValue`* is `'RMA'`, `affyrma` implements the estimation method described by Bolstad, 2005. When *`MethodValue`* is `'MLE'`, `affyrma` estimates the parameters using maximum likelihood. Default is `'RMA'`.

`Expression` `= affyrma(..., 'Truncate',` *`TruncateValue`*`, ...)` specifies the background noise model used. When *`TruncateValue`* is `false`, `affyrma` uses nontruncated Gaussian as the background noise model. Default is `true`.

`Expression` `= affyrma(..., 'Median',` *`MedianValue`*`, ...)` specifies the use of the median of the ranked values instead of the mean for normalization. Choices are `true` or `false` (default).

`Expression` `= affyrma(..., 'Output',` *`OutputValue`*`, ...)` specifies the scale of the returned gene expression values. *`OutputValue`* can be:

- `'log'`
- `'log2'`
- `'log10'`
- `'linear'`
- `@`*`functionname`*

In the last instance, the data is transformed as defined by the function *`functionname`*. Default is `'log2'`.

`Expression` `= affyrma(..., 'Showplot',` *`ShowplotValue`*`, ...)` lets you plot a histogram showing the distribution of PM probe intensity values (blue) and the convoluted probability distribution function (red), with estimated parameters mu, sigma and alpha. When *`ShowplotValue`* is `'all'`, `rmabackadj` plots a histogram for each column or chip. When *`ShowplotValue`* is a number, list of numbers, or range of numbers, `rmabackadj` plots a histogram for the indicated column number (chip).

For example:

- `(..., 'Showplot', 3,...)` plots the intensity values in column 3.
- `(..., 'Showplot', [3,5,7],...)` plots the intensity values in columns 3, 5, and 7.
- `(..., 'Showplot', 3:9,...)` plots the intensity values in columns 3 to 9.

`Expression` `= affyrma(..., 'Verbose',` *`VerboseValue`*`, ...)` controls the display of the status of the reading of files and RMA processing. Choices are `true` (default) or `false`.

Examples

The following example assumes that you have the `HG_U95Av2.CDF` library file stored at `D:\Affymetrix\LibFiles\HGGenome`, and that your current folder points to a location containing CEL files associated with this CDF library file. In this example, the `affyrma` function reads all the CEL files in the current folder and a CDF file in a specified folder. It also performs RMA background adjustment, quantile normalization, and summarization procedures on the PM probe intensity values, and returns a DataMatrix object, containing the metadata and processed data.

```
Expression = affyrma('*', 'HG_U95Av2.CDF',...
                          'CDFPath', 'D:\Affymetrix\LibFiles\HGGenome');
```

Version History

Introduced in R2008b

References

[1] Irizarry, R.A., Hobbs, B., Collin, F., Beazer-Barclay, Y.D., Antonellis, K.J., Scherf, U., Speed, T.P. (2003). Exploration, Normalization, and Summaries of High Density Oligonucleotide Array Probe Level Data. Biostatistics. *4*, 249–264.

[2] Mosteller, F., and Tukey, J. (1977). Data Analysis and Regression (Reading, Massachusetts: Addison-Wesley Publishing Company), pp. 165–202.

[3] Best, C.J.M., Gillespie, J.W., Yi, Y., Chandramouli, G.V.R., Perlmutter, M.A., Gathright, Y., Erickson, H.S., Georgevich, L., Tangrea, M.A., Duray, P.H., Gonzalez, S., Velasco, A., Linehan, W.M., Matusik, R.J., Price, D.K., Figg, W.D., Emmert-Buck, M.R., and Chuaqui, R.F. (2005). Molecular alterations in primary prostate cancer after androgen ablation therapy. Clinical Cancer Research *11*, 6823–6834.

[4] Bolstad, B. (2005). “affy: Built-in Processing Methods” `https://www.bioconductor.org/packages/2.1/bioc/vignettes/affy/ inst/doc/builtinMethods.pdf`

See Also

affygcrma | celintensityread | gcrma | mafdr | mattest | quantilenorm | rmabackadj | rmasummary

affysnpannotread

Read Affymetrix Mapping DNA array data from CSV-format annotation file

Syntax

```
AnnotStruct = affysnpannotread(File, PID)
AnnotStruct = affysnpannotread(File, PID, 'LookUpField', LookUpFieldValue)
```

Input Arguments

File	Character vector or string specifying a file name or a path and file name of an Affymetrix CSV annotation file for a Mapping 10K array set, Mapping 100K array set, or Mapping 500K array set. If you specify only a file name, that file must be on the MATLAB search path or in the current folder.
PID	Character vector, string, string vector, or cell array of character vectors specifying one or more probe set IDs on an Affymetrix mapping array.
LookUpFieldValue	Character vector, string, string vector, or cell array of character vectors specifying one or more column headers in an Affymetrix CSV annotation file. Default are the fields shown in the following table.

Output Arguments

AnnotStruct	MATLAB structure containing information for one or more probe sets from `File`, an Affymetrix CSV annotation file. *AnnotStruct* contains a subset of the fields in `File`. The fields are described in the table below.

Description

AnnotStruct = `affysnpannotread(`*File*`,` *PID*`)` reads *File*, an Affymetrix CSV annotation file for a Mapping 10K array set, Mapping 100K array set, or Mapping 500K array set, and returns *AnnotStruct*, a MATLAB structure containing annotation information for one or more probe sets specified by *PID*, a character vector, string, string vector, or cell array of character vectors specifying one or more probe set IDs. *AnnotStruct* contains a subset of the fields in *File*. The fields are described in the following table.

Structure Created from an Affymetrix CSV Annotation File

Field	Description
`ProbeSetIDs`	Cell array containing the unique probe set IDs specified by the *`PID`* input.
`Chromosome`	Cell array containing the chromosome number on which each probe set is located.
`ChromPosition`	Cell array containing the SNP genomic position on the chromosome for each probe set.
`Cytoband`	Cell array containing the cytogenetic banding region of the chromosome on which each probe set is located.
`Sequence`	Cell array containing the sequence of each probe set.
`AlleleA`	Cell array containing the base that is allele A for each probe set.
`AlleleB`	Cell array containing the base that is allele B for each probe set.
`Accession`	Cell array containing the GenBank® accession number for each probe set.
`FragmentLength`	Cell array containing the length of each probe set.

`AnnotStruct` `= affysnpannotread(`*`File`*`,` *`PID`*`, 'LookUpField',` *`LookUpFieldValue`*`)` returns annotation information from only the field (column) specified by *`LookUpFieldValue`*, a character vector, string, string vector, or cell array of character vectors specifying one or more column headers in an Affymetrix CSV annotation file. Default are the fields shown in the previous table.

Note You can download Affymetrix CSV annotation files such as `Mapping50K_Xba240.na25.annot.csv` from:

```
https://www.affymetrix.com/support/technical/annotationfilesmain.affx
```

Examples

The following example assumes that you have the `Mapping50K_Xba240.CDF` file stored at `C:\AffyLibFiles\`, and that your current folder points to a location containing the `Mapping50K_Xba240.na25.annot.csv` annotation file.

1. Use the `affyread` function to create a structure containing information from the `Mapping50K_Xba240.CDF` library file.

   ```
   cdf = affyread('C:\AffyLibFiles\Mapping50K_Xba240.CDF');
   ```

2. Create a variable containing a cell array of the names of the probe sets, which are stored in the `Name` field of the `ProbeSets` field of the `cdf` structure.

   ```
   probesetIDs = {cdf.ProbeSets.Name}';
   ```

3. Return a structure containing annotation information for all the probe sets in the `Mapping50K_Xba240.na25.annot.csv` annotation file.

   ```
   snpInfo = affysnpannotread('Mapping50K_Xba240.na25.annot.csv',probesetIDs)

   snpInfo =
   ```

```
       ProbeSetIDs: {59024x1 cell}
        Chromosome: [59024x1 int8]
     ChromPosition: [59024x1 double]
          Cytoband: {59024x1 cell}
          Sequence: {59024x1 cell}
           AlleleA: {59024x1 cell}
           AlleleB: {59024x1 cell}
         Accession: {59024x1 cell}
    FragmentLength: [59024x1 double]
```

Version History

Introduced in R2008b

See Also

`affysnpintensitysplit` | `affyread`

affysnpintensitysplit

Split Affymetrix SNP probe intensity information for alleles A and B

Syntax

```
ProbeStructSplit = affysnpintensitysplit(ProbeStruct)
ProbeStructSplit = affysnpintensitysplit(ProbeStruct, 'Controls',
ControlsValue)
```

Input Arguments

ProbeStruct	MATLAB structure containing probe intensity information from an Affymetrix Mapping DNA array, such as returned by `celintensityread`.
ControlsValue	Controls the inclusion of control probes in *ProbeStructSplit*. Choices are `true` or `false` (default).

Output Arguments

ProbeStructSplit	MATLAB structure containing probe intensity information from an Affymetrix Mapping DNA array, split into information for alleles A and B.

Description

ProbeStructSplit = `affysnpintensitysplit(`*ProbeStruct*`)` splits *ProbeStruct*, a structure containing probe intensity information from an Affymetrix Mapping DNA array, into *ProbeStructSplit*, a structure containing probe intensity information from an Affymetrix Mapping DNA array, split into information for alleles A and B.

ProbeStructSplit contains the following fields.

Field	Description
`CDFName`	File name of the Affymetrix CDF library file.
`CELNames`	Cell array of names of the Affymetrix CEL files.
`NumChips`	Number of CEL files read into the input structure.
`NumProbeSets`	Number of probe sets in each CEL file.
`NumProbes`	Maximum number of probes for just one allele in each CEL file. **Note** If the number of probes for allele A is not the same as for allele B, the larger number is used.
`ProbeSetIDs`	Cell array of the probe set IDs from the Affymetrix CDF library file.

Field	Description
`ProbeIndices`	Column vector containing probe indexing information for just one allele in each cell file. Probes within a probe set are numbered `0` through `N - 1`, where `N` is the number of probes for one allele in the probe set. **Note** `ProbeIndices` has the same number of elements as `NumProbes`.
`PMAIntensities`	Matrix containing perfect match (PM) probe intensity values for allele A. Each row corresponds to an allele A probe, and each column corresponds to a CEL file. The rows are ordered the same way as in `ProbeIndices`, and the columns are ordered the same way as in the *`CELFiles`* input argument to the `celintensityread` function.
`PMBIntensities`	Matrix containing perfect match (PM) probe intensity values for allele B. Each row corresponds to an allele B probe, and each column corresponds to a CEL file. The rows are ordered the same way as in `ProbeIndices`, and the columns are ordered the same way as in the *`CELFiles`* input argument to the `celintensityread` function.
`MMAIntensities` (optional)	Matrix containing mismatch (MM) probe intensity values for allele A. Each row corresponds to an allele A probe, and each column corresponds to a CEL file. The rows are ordered the same way as in `ProbeIndices`, and the columns are ordered the same way as in the *`CELFiles`* input argument to the `celintensityread` function.
`MMBIntensities` (optional)	Matrix containing mismatch (MM) probe intensity values for allele B. Each row corresponds to an allele B probe, and each column corresponds to a CEL file. The rows are ordered the same way as in `ProbeIndices`, and the columns are ordered the same way as in the *`CELFiles`* input argument to the `celintensityread` function.

`ProbeStructSplit` `= affysnpintensitysplit(`*`ProbeStruct`*`, 'Controls',` *`ControlsValue`*`)` controls the return of control probe intensities. Choices are `true` or `false` (default).

Note Control probes sometimes contain information for only one allele. In this case, the value for the corresponding allele (A or B) that is not present is set to NaN.

Examples

The following example assumes that your current folder points to a location containing the `Mapping50K_Hind240.CDF` library file and 18 CEL files associated with this CDF library file. These files are associated with an Affymetrix Mapping DNA array.

1 Use the `celintensityread` function to read the `Mapping50K_Hind240.CDF` library file and 18 CEL files associated with it into a MATLAB structure.

```
ps = celintensityread('*','Mapping50K_Hind240.CDF')
ps =

          CDFName: 'Mapping50K_Hind240.CDF'
         CELNames: {18x1 cell}
         NumChips: 18
     NumProbeSets: 57299
        NumProbes: 1145780
      ProbeSetIDs: {57299x1 cell}
     ProbeIndices: [1145780x1 uint8]
     GroupNumbers: [1145780x1 uint8]
    PMIntensities: [1145780x18 single]
```

2 Extract the PM probe intensities for allele A and allele B into another MATLAB structure, without including intensity information for the control probes.

```
ps_split = affysnpintensitysplit(ps)

ps_split =

          CDFName: 'Mapping50K_Hind240.CDF'
         CELNames: {18x1 cell}
         NumChips: 18
     NumProbeSets: 57275
        NumProbes: 572750
      ProbeSetIDs: {57275x1 cell}
     ProbeIndices: [572750x1 uint8]
   PMAIntensities: [572750x18 single]
   PMBIntensities: [572750x18 single]
```

Version History
Introduced in R2008b

See Also
`affysnpannotread` | `affyread` | `celintensityread`

affysnpquartets

Create table of SNP probe quartet results for Affymetrix probe set

Syntax

SNPQStruct = affysnpquartets(*CELStruct*, *CDFStruct*, *PS*)

Input Arguments

CELStruct	Structure created by the affyread function from an Affymetrix CEL file, which contains information about the intensity values of the individual probes.
CDFStruct	Structure created by the affyread function from an Affymetrix CDF library file associated with the CEL file. The CDF library file contains information about which probes belong to which probe set.
PS	Probe set index or the probe set ID/name.

Output Arguments

SNPQStruct	Structure containing probe quartet results for a specific SNP probe set from the data in a CEL file and associated CDF library file.

Description

SNPQStruct = affysnpquartets(*CELStruct*, *CDFStruct*, *PS*) creates *SNPQStruct*, a structure containing probe quartet results for a specific SNP probe set, specified by *PS*, from the probe-level data in a CEL file and associated CDF library file. *CELStruct* is a structure created by the affyread function from an Affymetrix CEL file. *PS* is a probe set index or probe set ID/name from *CDFStruct*, a structure created by the affyread function from an Affymetrix CDF library file associated with the CEL file. *SNPQStruct* is a structure containing the following fields.

Field	Description
'ProbeSet'	Identifier for the probe set.
'AlleleA'	Character vector specifying the base that is allele A for the probe set.
'AlleleB'	Character vector specifying the base that is allele B for the probe set.
'Quartet'	Structure array containing intensity values for PM (perfect match) and MM (mismatch) probe pairs, including the sense and antisense probes for alleles A and B. Each structure in the array corresponds to a probe pair in the probe set.

Examples

The following example uses the NA06985_Hind_B5_3005533.CEL file. You can download the sample CEL files from here.

The `NA06985_Hind_B5_3005533.CEL` file is included in the `100K_trios.hind.1.zip` file.

The following example uses the CDF library file for the Mapping 50K Hind 240 array. The library files can be downloaded from here.

The following example assumes that the `NA06985_Hind_B5_3005533.CEL` file is stored on the MATLAB search path or in the current folder. It also assumes that the associated CDF library file, `Mapping50K_Hind240.cdf`, is stored at `D:\Affymetrix\LibFiles\`.

1 Read the contents of a CEL file into a MATLAB structure.

```
celStruct = affyread('NA06985_Hind_B5_3005533.CEL');
```

2 Read the contents of a CDF file into a MATLAB structure.

```
cdfStruct = affyread('D:\Affymetrix\LibFiles\Mapping50K_Hind240.cdf');
```

3 Create a structure containing SNP probe quartet results for the `SNP_A-1684395` probe set.

```
SNPQStruct = affysnpquartets(celStruct,cdfStruct,'SNP_A-1684395')

SNPQStruct = 

    ProbeSet: 'SNP_A-1684395'
     AlleleA: 'A'
     AlleleB: 'G'
     Quartet: [1x5 struct]
```

4 View the intensity values of the first probe pair in the probe set.

```
SNPQStruct.Quartet(1)

ans = 

        A_Sense_PM: 5013
        B_Sense_PM: 1290
        A_Sense_MM: 1485
        B_Sense_MM: 686
    A_Antisense_PM: 3746
    B_Antisense_PM: 1406
    A_Antisense_MM: 1527
    B_Antisense_MM: 958
```

Version History

Introduced in R2008a

See Also

`affyread` | `probesetvalues`

agferead

Read Agilent Feature Extraction Software file

Syntax

AGFEData = agferead(*File*)

Arguments

File	Microarray data file generated with the Agilent® Feature Extraction Software.

Description

AGFEData = agferead(*File*) reads files generated with the Feature Extraction Software from Agilent microarray scanners and creates a structure (*AGFEData*) containing the following fields:

- Header
- Stats
- Columns
- Rows
- Names
- IDs
- Data
- ColumnNames
- TextData
- TextColumnNames

The Feature Extraction Software takes an image from an Agilent microarray scanner and generates raw intensity data for each spot on the plate.

Examples

1. Read in a sample Agilent Feature Extraction Software file. Note that the file `fe_sample.txt` is not provided with the Bioinformatics Toolbox software.

   ```
   agfeStruct = agferead('fe_sample.txt')
   ```
2. Plot the median foreground.

   ```
   maimage(agfeStruct,'gMedianSignal');
   maboxplot(agfeStruct,'gMedianSignal');
   ```

Version History

Introduced before R2006a

See Also

`affyread` | `celintensityread` | `galread` | `geoseriesread` | `geosoftread` | `gprread` | `ilmnbsread` | `imageneread` | `magetfield` | `sptread`

align2cigar

Convert aligned sequences to corresponding signatures in CIGAR format

Syntax

```
Cigars = align2cigar(Alignment,Ref)
[Cigars,Starts] = align2cigar(Alignment,Ref)
```

Description

`Cigars = align2cigar(Alignment,Ref)` converts aligned sequences represented in `Alignment` into `Cigars` using the reference sequence specified by `Ref`.

`[Cigars,Starts] = align2cigar(Alignment,Ref)` also returns `Starts`, a vector of integers indicating the start position of each aligned sequence with respect to the ungapped reference sequence.

Examples

Convert aligned sequences to CIGAR strings

This example shows how to convert aligned strings to CIGAR strings

Create a cell array of aligned strings, create a string specifying a reference sequence, and then convert the alignment to CIGAR strings:

```
aln = ['ACG-ATGC'; 'ACGT-TGC'; '  GTAT-C']
```

```
aln = 3x8 char array
    'ACG-ATGC'
    'ACGT-TGC'
    '  GTAT-C'
```

```
ref =  'ACGTATGC';
[cigar, start] = align2cigar(aln, ref)
```

```
cigar = 1x3 cell
    {'3=1D4='}    {'4=1D3='}    {'4=1D1='}


start = 1×3

     1     1     3
```

Input Arguments

`Alignment` — Aligned sequence
cell array of aligned character vectors | string vector | character array

Aligned sequence, specified as a cell array of aligned character vectors, a string vector, or a character array. Soft clippings are assumed to be represented by lowercase letters in the aligned sequences. Skipped positions are assumed to be represented by a period `.` in the aligned sequences.

Data Types: `char` | `string` | `cell`

`Ref` — Aligned reference sequence
character vector | string

Aligned reference sequence, specified as a character vector or string. The length of `Ref` must equal the number of columns in `Alignment`.

Data Types: `char` | `string`

Output Arguments

`Cigars` — Converted sequence in CIGAR format
cell array of character vectors

Converted sequence in CIGAR format, returned as a cell array of character vectors. Each entry in `Cigars` corresponds to one entry in `Alignment`.

`Starts` — Start position of each aligned sequence
integer vector

Start position of each aligned sequence, returned as an integer vector. The start positions are with respect to the ungapped reference sequence specified by `Ref`.

Version History

Introduced in R2010b

See Also

`cigar2align` | `multialign` | `getBaseCoverage` | `getCompactAlignment` | `getAlignment` | `BioMap`

Topics
"Manage Sequence Read Data in Objects"

External Websites
Sequence Read Archive
SAM format specification

allshortestpaths (biograph)

(Removed) Find all shortest paths in biograph object

Note `allshortestpaths` has been removed. Use `distances` instead.

Syntax

```
[dist] = allshortestpaths(BGObj)

[dist] = allshortestpaths(BGObj, ...'Directed', DirectedValue, ...)
[dist] = allshortestpaths(BGObj, ...'Weights', WeightsValue, ...)
```

Arguments

`BGObj`	Biograph object created by `biograph` (object constructor).
`DirectedValue`	Property that indicates whether the graph is directed or undirected. Enter `false` for an undirected graph. This results in the upper triangle of the adjacency matrix being ignored. Default is `true`.
`WeightsValue`	Column vector that specifies custom weights for the edges in the N-by-N adjacency matrix extracted from a biograph object, *`BGObj`*. It must have one entry for every nonzero value (edge) in the matrix. The order of the custom weights in the vector must match the order of the nonzero values in the matrix when it is traversed column-wise. This property lets you use zero-valued weights. By default, `allshortestpaths` gets weight information from the nonzero entries in the matrix.

Description

Tip For introductory information on graph theory functions, see “Graph Theory Functions”.

`[`*`dist`*`] = allshortestpaths(`*`BGObj`*`)` finds the shortest paths between every pair of nodes in a graph represented by an N-by-N adjacency matrix extracted from a biograph object, *`BGObj`*, using Johnson's algorithm. Nonzero entries in the matrix represent the weights of the edges.

Output *`dist`* is an N-by-N matrix where *`dist`*`(S,T)` is the distance of the shortest path from source node `S` to target node `T`. Elements in the diagonal of this matrix are always `0`, indicating the source node and target node are the same. A `0` not in the diagonal indicates that the distance between the source node and target node is `0`. An `Inf` indicates there is no path between the source node and the target node.

Johnson's algorithm has a time complexity of `O(N*log(N)+N*E)`, where `N` and `E` are the number of nodes and edges respectively.

`[...] = allshortestpaths (`*`BGObj`*`, '`*`PropertyName`*`', `*`PropertyValue`*`, ...)` calls `allshortestpaths` with optional properties that use property name/property value pairs. You can specify one or more properties in any order. Each *`PropertyName`* must be enclosed in single quotes and is case insensitive. These property name/property value pairs are as follows:

`[`*`dist`*`] = allshortestpaths(`*`BGObj`*`, ...'Directed',` *`DirectedValue`*`, ...)` indicates whether the graph is directed or undirected. Set *`DirectedValue`* to `false` for an undirected graph. This results in the upper triangle of the adjacency matrix being ignored. Default is `true`.

`[`*`dist`*`] = allshortestpaths(`*`BGObj`*`, ...'Weights',` *`WeightsValue`*`, ...)` lets you specify custom weights for the edges. *`WeightsValue`* is a column vector having one entry for every nonzero value (edge) in the N-by-N adjacency matrix extracted from a biograph object, *`BGObj`*. The order of the custom weights in the vector must match the order of the nonzero values in the N-by-N adjacency matrix when it is traversed column-wise. This property lets you use zero-valued weights. By default, `allshortestpaths` gets weight information from the nonzero entries in the N-by-N adjacency matrix.

Version History

Introduced in R2006b

R2022b: Removed
Errors starting in R2022b

The function has been removed. Use `distances` instead.

R2022a: Warns
Warns starting in R2022a

The function issues a warning that it will be removed in a future release.

R2021b: To be removed
Not recommended starting in R2021b

The function runs without warning, but it will be removed in a future release.

References

[1] Johnson, D.B. (1977). Efficient algorithms for shortest paths in sparse networks. Journal of the ACM *24(1)*, 1-13.

[2] Siek, J.G., Lee, L-Q, and Lumsdaine, A. (2002). The Boost Graph Library User Guide and Reference Manual, (Upper Saddle River, NJ:Pearson Education).

See Also

`distances`

aminolookup

Find amino acid codes, integers, abbreviations, names, and codons

Syntax

```
aminolookup
aminolookup(SeqAA)
aminolookup('Code', CodeValue)
aminolookup('Integer', IntegerValue)
aminolookup('Abbreviation', AbbreviationValue)
aminolookup('Name', NameValue)
```

Arguments

SeqAA	Character vector or string containing single-letter codes or three-letter abbreviations representing an amino acid sequence. For valid codes and abbreviations, see the table Amino Acid Lookup.
CodeValue	Character vector or string specifying a single-letter code representing an amino acid. For valid single-letter codes, see the table Amino Acid Lookup.
IntegerValue	Single integer representing an amino acid. For valid integers, see the table Amino Acid Lookup.
AbbreviationValue	Character vector or string specifying a three-letter abbreviation representing an amino acid. For valid three-letter abbreviations, see the table Amino Acid Lookup.
NameValue	Character vector or string specifying an amino acid name. For valid amino acid names, see the table Amino Acid Lookup.

Description

`aminolookup` displays a table of amino acid codes, integers, abbreviations, names, and codons.

Amino Acid Lookup

Code	Integer	Abbreviation	Amino Acid Name	Codons
A	1	Ala	Alanine	GCU GCC GCA GCG
R	2	Arg	Arginine	CGU CGC CGA CGG AGA AGG
N	3	Asn	Asparagine	AAU AAC
D	4	Asp	Aspartic acid (Aspartate)	GAU GAC
C	5	Cys	Cysteine	UGU UGC
Q	6	Gln	Glutamine	CAA CAG
E	7	Glu	Glutamic acid (Glutamate)	GAA GAG
G	8	Gly	Glycine	GGU GGC GGA GGG
H	9	His	Histidine	CAU CAC
I	10	Ile	Isoleucine	AUU AUC AUA
L	11	Leu	Leucine	UUA UUG CUU CUC CUA CUG
K	12	Lys	Lysine	AAA AAG
M	13	Met	Methionine	AUG
F	14	Phe	Phenylalanine	UUU UUC
P	15	Pro	Proline	CCU CCC CCA CCG
S	16	Ser	Serine	UCU UCC UCA UCG AGU AGC
T	17	Thr	Threonine	ACU ACC ACA ACG
W	18	Trp	Tryptophan	UGG
Y	19	Tyr	Tyrosine	UAU UAC
V	20	Val	Valine	GUU GUC GUA GUG
B	21	Asx	Asparagine or Aspartic acid (Aspartate)	AAU AAC GAU GAC
Z	22	Glx	Glutamine or Glutamic acid (Glutamate)	CAA CAG GAA GAG
X	23	Xaa	Any amino acid	All codons
*	24	END	Termination codon (translation stop)	UAA UAG UGA
-	25	GAP	Gap of unknown length	NA

`aminolookup(`*`SeqAA`*`)` converts between single-letter codes and three-letter abbreviations for an amino acid sequence. If the input is a character vector or string of single-letter codes, then the output is a character vector of three-letter abbreviations. If the input is a character vector or string of three-letter abbreviations, then the output is a character vector of the corresponding single-letter codes.

If you enter one of the ambiguous single-letter codes `B`, `Z`, or `X`, this function displays the corresponding abbreviation for the ambiguous amino acid character.

```
aminolookup('abc')

ans =

AlaAsxCys
```

`aminolookup('Code', *CodeValue*)` displays the corresponding amino acid three-letter abbreviation and name.

`aminolookup('Integer', *IntegerValue*)` displays the corresponding amino acid single-letter code, three-letter abbreviation, and name.

`aminolookup('Abbreviation', *AbbreviationValue*)` displays the corresponding amino acid single-letter code and name.

`aminolookup('Name', *NameValue*)` displays the corresponding amino acid single-letter code and three-letter abbreviation.

Examples

Convert an amino acid sequence to single-letter or three-letter abbreviations

Convert an amino acid sequence in single-letter codes to the corresponding three-letter abbreviations.

```
aminolookup('MWKQAEDIRDIYDF')
```

```
ans =
'MetTrpLysGlnAlaGluAspIleArgAspIleTyrAspPhe'
```

Convert an amino acid sequence in three-letter abbreviations to the corresponding single-letter codes.

```
aminolookup('MetTrpLysGlnAlaGluAspIleArgAspIleTyrAspPhe')
```

```
ans =
'MWKQAEDIRDIYDF'
```

Display the three-letter abbreviation and name for the amino acid corresponding to the single-letter code R.

```
aminolookup('Code', 'R')
```

```
ans =
    'Arg    Arginine
     '
```

Display the single-letter code, three-letter abbreviation, and name for the amino acid corresponding to the integer 1.

```
aminolookup('Integer', 1)
```

```
ans =
    'A    Ala    Alanine
     '
```

Display the single-letter code and name for the amino acid corresponding to the three-letter abbreviation asn.

```
aminolookup('Abbreviation', 'asn')
```

```
ans =
    'N    Asparagine
     '
```

Display the single-letter code and three-letter abbreviation for the amino acid proline.

```
aminolookup('Name','proline')
```

```
ans =
    'P    Pro
     '
```

Version History

Introduced before R2006a

See Also

aa2int | aa2nt | aacount | geneticcode | int2aa | isotopicdist | nt2aa | revgeneticcode

atomiccomp

Calculate atomic composition of protein

Syntax

```
NumberAtoms = atomiccomp(SeqAA)
```

Arguments

SeqAA	Amino acid sequence. Enter a character vector or vector of integers from the table Mapping Amino Acid Letter Codes to Integers. You can also enter a structure with the field `Sequence`.

Description

`NumberAtoms` `= atomiccomp(`*`SeqAA`*`)` counts the type and number of atoms in an amino acid sequence (*`SeqAA`*) and returns the counts in a `1`-by-`1` structure (*`NumberAtoms`*) with fields `C`, `H`, `N`, `O`, and `S`.

Examples

Calculate the atomic composition of a protein

Retrieve an amino acid sequence from the NCBI GenPept database.

```
rhodopsin = getgenpept('NP_000530')
```

```
rhodopsin = 

  struct with fields:

                LocusName: 'NP_000530'
      LocusSequenceLength: '348'
     LocusNumberofStrands: ''
            LocusTopology: 'linear'
        LocusMoleculeType: ''
     LocusGenBankDivision: 'PRI'
    LocusModificationDate: '07-AUG-2015'
               Definition: 'rhodopsin [Homo sapiens].'
                Accession: 'NP_000530'
                  Version: 'NP_000530.1'
                       GI: '4506527'
                  Project: []
                   DBLink: ' DBSOURCE    REFSEQ: accession NM_000539.3'
                 Keywords: 'RefSeq.'
                  Segment: []
                   Source: 'Homo sapiens  human '
           SourceOrganism: [4×65 char]
```

```
          Reference: {1×18 cell}
            Comment: [26×67 char]
           Features: [219×74 char]
                CDS: [1×1 struct]
           Sequence: 'mngtegpnfyvpfsnatgvvrspfeypqyylaepwqfsmlaaymfl...'
          SearchURL: 'http://www.ncbi.nlm.nih.gov/entrez/viewer.fcgi...'
        RetrieveURL: 'http://eutils.ncbi.nlm.nih.gov/entrez/eutils/e...'
```

Count the atoms in the sequence.

```
rhodopsinAC = atomiccomp(rhodopsin)
```

```
rhodopsinAC = 

  struct with fields:

    C: 1814
    H: 2725
    N: 423
    O: 477
    S: 25
```

Retrieve the number of carbon atoms.

```
rhodopsinAC.C
```

```
ans =

        1814
```

Version History

Introduced before R2006a

See Also

aacount | molweight | proteinplot

baminfo

Return information about BAM file

Syntax

```
InfoStruct = baminfo(File)
InfoStruct = baminfo(File,Name,Value)
```

Description

`InfoStruct = baminfo(File)` returns a MATLAB structure containing summary information about a BAM-formatted file.

Use `baminfo` to investigate the size and content of a BAM-formatted file, including reference sequence names, before using the `bamread` function to read the file contents into a MATLAB structure.

`InfoStruct = baminfo(File,Name,Value)` specifies additional options using one or more name-value arguments. For example, to return the number of alignment records, `InfoStruct = baminfo(File,NumOfReads=true)`.

Examples

Retrieve information about a BAM file

This example shows how to retrieve information about the ex1.bam file included with the Bioinformatics Toolbox™.

```
info = baminfo('ex1.bam','ScanDictionary',true,'numofreads',true)
```

```
info = struct with fields:
                  Filename: 'ex1.bam'
                  FilePath: 'B:\matlab\toolbox\bioinfo\bioinfodata'
                  FileSize: 126692
               FileModDate: '07-May-2010 16:12:05'
                    Header: [1x1 struct]
                 ReadGroup: [1x2 struct]
        SequenceDictionary: [1x2 struct]
                  NumReads: 3307
         ScannedDictionary: {2x1 cell}
    ScannedDictionaryCount: [2x1 uint64]
```

List the number of references found in the BAM file.

```
numel(info.ScannedDictionary)
```

```
ans = 2
```

Alternatively, you can use the available header information from a BAM file to find out the number of references, thus avoiding the whole traversal of the source file.

```
info = baminfo('ex1.bam');
NRefs = numel(info.SequenceDictionary)
```

```
NRefs = 2
```

Input Arguments

File — Path to BAM-formatted file
string | character vector

Path to BAM-formatted file, specified as a character vector or string specifying a file name or path and file name of a file. If you specify only a file name, that file must be on the MATLAB search path or in the current folder.

Example: "C:\Documents\bamfile.bam"

Data Types: char | string

Name-Value Pair Arguments

Specify optional pairs of arguments as Name1=Value1,...,NameN=ValueN, where Name is the argument name and Value is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose Name *in quotes.*

Names are case-insensitive. For example, you can use "numofreads" instead of "NumOfReads".

Example: InfoStruct = baminfo("ex1.bam",ScanDictionary=true,NumOfReads=true)

NumOfReads — Indication to determine the number of alignment records in the file
false (default) | true

Indication to determine the number of alignment records in the file, specified as false or true. If true, the NumReads field of InfoStruct in contains this information.

Example: true

Data Types: logical

ScanDictionary — Indication to determine the number of alignment records in the file
false (default) | true

Indication to determine the reference names and the number of reads aligned to each reference.File, specified as false (do not determine) or true. If true, the ScannedDictionary and ScannedDictionaryCount fields of InfoStruct contain this information.

Example: true

Data Types: logical

Output Arguments

InfoStruct — Summary information about a BAM-formatted file
structure

Summary information about a BAM-formatted file, returned as a MATLAB structure. The structure contains these fields.

Field	Description
`Filename`	Name of the BAM-formatted file.
`FilePath`	Path to the file.
`FileSize`	Size of the file in bytes.
`FileModDate`	Modification date of the file.
`Header**`	Structure containing the file format version, sort order, and group order.
`ReadGroup**`	Structure containing the: • Read group identifier • Sample • Library • Description • Platform unit • Predicted median insert size • Sequencing center • Date • Platform
`SequenceDictionary**`	Structure containing the: • Sequence name • Sequence length • Genome assembly identifier • MD5 checksum of sequence • URI of sequence • Species
`Program**`	Structure containing the: • Program name • Version • Command line
`NumReads`	Number of reference sequences in the BAM-formatted file.
`ScannedDictionary*`	Cell array of character vectors specifying the names of the reference sequences in the BAM-formatted file.
`ScannedDictionaryCount*`	Cell array specifying the number of reads aligned to each reference sequence.

`*` — The `ScannedDictionary` and `ScannedDictionaryCount` fields are empty if you do not set the `ScanDictionary` name-value pair argument to `true`.

** — These structures and their fields appear in the output structure only if they are in the BAM file. The information in these structures depends on the information in the BAM file.

Version History

Introduced in R2010b

See Also

`bamread`

Topics

"Manage Sequence Read Data in Objects"

External Websites

Sequence Read Archive
SAM format specification

bamread

Read data from BAM file

Syntax

```
BAMStruct = bamread(File,RefSeq,Range)
BAMStruct = bamread(File,nomap)
[BAMStruct,HeaderStruct] = bamread(File,RefSeq,Range)
___ = bamread(File,RefSeq,Range,Name,Value)
```

Description

`BAMStruct = bamread(File,RefSeq,Range)` reads the alignment records in `File`, a BAM-formatted file, that align to `RefSeq`, a reference sequence, in the range specified by `Range`. The `BAMStruct` output contains the alignment data.

`BAMStruct = bamread(File,nomap)` returns reads that are not mapped to any reference.

`[BAMStruct,HeaderStruct] = bamread(File,RefSeq,Range)` also returns the header information in `HeaderStruct`

`___ = bamread(File,RefSeq,Range,Name,Value)`, for any output arguments, specifies additional options using one or more name-value arguments. For example, to save the alignment records to a file named `'alignmentrecords.sam'` in the current folder, `BAMStruct = bamread(File,RefSeq,Range,ToFile="alignmentrecords.sam")`.

Examples

Retrieve alignment records that align to reference sequences

Read multiple alignment records from the ex1.bam file that align to two different reference sequences.

```
data1 = bamread('ex1.bam', 'seq1', [100 200])
```

```
data1=59×1 struct array with fields:
    QueryName
    Flag
    Position
    MappingQuality
    CigarString
    MatePosition
    InsertSize
    Sequence
    Quality
    Tags
    ReferenceIndex
    MateReferenceIndex
```

```
data2 = bamread('ex1.bam', 'seq2', [100 200])
```

```
data2=79×1 struct array with fields:
    QueryName
    Flag
    Position
    MappingQuality
    CigarString
    MatePosition
    InsertSize
    Sequence
    Quality
    Tags
    ReferenceIndex
    MateReferenceIndex
```

Read alignments from the ex1.bam file that are fully contained in the 100 to 200 bp range of the seq1 reference sequence.

```
data3 = bamread('ex1.bam', 'seq1', [100 200], 'full', true)
```

```
data3=30×1 struct array with fields:
    QueryName
    Flag
    Position
    MappingQuality
    CigarString
    MatePosition
    InsertSize
    Sequence
    Quality
    Tags
    ReferenceIndex
    MateReferenceIndex
```

Read alignments from the ex1.bam file that align to the 100 to 300 bp range of the seq1 reference sequence. Read the same alignments using zero-based indexing. Compare the position of the 27th record in the two outputs.

```
data_one = bamread('ex1.bam','seq1', [100 300]);
data_zero = bamread('ex1.bam','seq1', [100 300], 'zerobased', true);
data_one(27).Position
```

```
ans = uint32
    135
```

```
data_zero(27).Position
```

```
ans = uint32
    134
```

Input Arguments

File — Path to a BAM-formatted file
string | character vector

Path to a BAM-formatted file, specified as a string or character vector. If the file is on the MATLAB search path or is in the current folder, `File` can be the file name alone without the path information.

Note `bamread` requires the BAM file to be ordered, except when returning reads that are not mapped to any reference.

Example: `"C:\Documents\myfile.bam"`

Data Types: `char` | `string`

`RefSeq` — Reference sequence in the BAM file
name of reference sequence | index of reference sequence

Reference sequence in the BAM file, specified as one of the following:

- Name of the reference sequence, specified as a string or character vector.
- Index of the reference sequence, specified as a positive integer. This number is also the index of the reference sequence in the `Reference` field of the `InfoStruct` structure returned by `baminfo`.

Data Types: `single` | `double` | `char` | `string`

Range — Range of references
name of a reference sequence in the BAM file | positive integer specifying the index of a reference sequence in the BAM file

Range of references, specified as one of the following:

- Character vector or string specifying the name of a reference sequence in the BAM file.
- Positive integer specifying the index of a reference sequence in the BAM file. This number is also the index of the reference sequence in the `Reference` field of the `InfoStruct` structure returned by `baminfo`.

Data Types: `double` | `char` | `string`

Range — Range positions in the reference sequence
two-element positive vector

Range positions in the reference sequence `RefSeq`, specified as a two-element vector of positive numbers with the second element being greater than or equal to the first.

Example: `[17,22]`

Data Types: `single` | `double`

nomap — Indication of no mapping
`0` | `'Unmapped'`

Indication that the returned information is not mapped to any reference, specified as `0` or `'Unmapped'`.

Example: `'Unmapped'`

Data Types: `single` | `double` | `char` | `string`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: To save the alignment records to a file named `'alignmentrecords.sam'` in the current folder, `BAMStruct = bamread(File,RefSeq,Range,ToFile="alignmentrecords.sam")`

`Full` — Indication to return only alignment records that are fully contained within the range specified by `Range`
`false` (default) | `true`

Indication to return only alignment records that are fully contained within the range specified by `Range`, specified as `false` or `true`.

Example: `true`

Data Types: `logical`

`Tags` — Indication to read optional tags
`true` (default) | `false`

Indication to read optional tags in addition to the first 11 fields for each alignment in the BAM-formatted file.

Example: `false`

Data Types: `logical`

`ToFile` — Path to file for saving alignment records
string | character vector

Path to a (new) file for saving alignment records in the specified range of a specific reference sequence, specified as a string or character vector. If you specify only a file name, the file is saved to the current folder. The resulting file is SAM-formatted.

The SAM-formatted file is always one-based, even if you set the `ZeroBased` name-value pair argument to `true`. You can use the SAM-formatted file as input when creating a `BioMap` object.

Example: `"C:\Documents\alignment.sam"`

Data Types: `char` | `string`

`ZeroBased` — Indication to use zero-based indexing when reading a file
`false` (default) | `true`

Indication to use zero-based indexing when reading a file, specified as `false` or `true`. The `ZeroBased` argument controls the return of zero-based or one-based positions in the `Position` and `MatePosition` fields in `BAMStruct`.

`ZeroBased` does not affect the `Range` input argument or the SAM file created when using the `ToFile` name-value pair argument. SAM files are always one-based.

Caution If you plan to use the `BAMStruct` output argument to construct a `BioMap` object, make sure the `ZeroBased` name-value pair argument is `false`.

Example: `true`

Data Types: `logical`

Output Arguments

`BAMStruct` — Alignment and mapping information

struct

Alignment and mapping information from a BAM-formatted file, returned as an *N*-by-1 array of structures, where *N* is the number of alignment records stored in the specified range. Each structure contains the following fields.

Field	Description
`QueryName`	Name of the read sequence (if unpaired) or the name of sequence pair (if paired).
`Flag`	Integer indicating the bit-wise information that specifies the status of each of 11 flags described by the SAM format specification. **Tip** You can use the `bitget` function to determine the status of a specific SAM flag.
`ReferenceIndex`	Index of the reference sequence. **Tip** To convert this index to a reference name, see the `Reference` field in the `HeaderStruct` output argument
`Position`	Position of the forward reference sequence where the leftmost base of the alignment of the read sequence starts. This position is zero-based or one-based, depending on the `ZeroBased` name-value pair argument.
`MappingQuality`	Integer specifying the mapping quality score for the read sequence.
`CigarString`	CIGAR-formatted string representing how the read sequence aligns with the reference sequence.
`MateReferenceIndex`	Index of the reference sequence associated with the mate. If there is no mate, then this value is `0`.
`MatePosition`	Position of the forward reference sequence where the leftmost base of the alignment of the mate of the read sequence starts. This position is zero-based or one-based, depending on the `ZeroBased` name-value pair argument.

Field	Description
`InsertSize`	The number of base positions between the read sequence and its mate, when both are mapped to the same reference sequence. Otherwise, this value is `0`.
`Sequence`	Character vector containing the letter representations of the read sequence. It is the reverse complement if the read sequence aligns to the reverse strand of the reference sequence.
`Quality`	Character vector containing the ASCII representation of the per-base quality score for the read sequence. The quality score is reversed if the read sequence aligns to the reverse strand of the reference sequence.
`Tags`	List of applicable SAM tags and their values.

`HeaderStruct` — Header information

structure

Header information for the BAM-formatted file, returned as a structure. The structure contains the following fields.

Field	Description
`NRefs`	Number of reference sequences in the BAM-formatted file.
`Reference`	1-by-`NRefs` array of structures containing these fields: • `Name` — Name of the reference sequence. • `Length` — Length of the reference sequence.
`Header*`	Structure containing the file format version, sort order, and group order.
`SequenceDictionary*`	Structure containing the: • Sequence name • Sequence length • Genome assembly identifier • MD5 checksum of sequence • URI of sequence • Species

Field	Description
`ReadGroup*`	Structure containing the: • Read group identifier • Sample • Library • Description • Platform unit • Predicted median insert size • Sequencing center • Date • Platform
`Program*`	Structure containing the: • Program name • Version • Command line

Tips

- Use the `baminfo` function to investigate the size and content, including reference sequence names, of a BAM-formatted file before using the `bamread` function to read the file contents into a MATLAB array of structures.
- If your BAM-formatted file is too large to read using available memory, try either of the following:
  - Use a smaller range.
  - Use `bamread` without specifying outputs, but using the `ToFile` Name,Value argument to create a SAM-formatted file. You can then use `samread` with the `BlockRead` Name,Value argument to read the SAM-formatted file. Or you can pass the SAM-formatted file to the `BioIndexedFile` constructor function to construct a `BioIndexedFile` object, which you can use to create a `BioMap` object.
- Use the `BAMStruct` output argument to construct a `BioMap` object, which lets you explore, access, filter, and manipulate all or a subset of the data, before doing subsequent analyses or viewing the data.

Version History

Introduced in R2010b

References

[1] Li, H., Handsaker, B., Wysoker, A., Fennell, T., Ruan, J., Homer, N., Marth, G., Goncalo, A., and Durbin, R. (2009). The Sequence Alignment/Map format and SAMtools. Bioinformatics *25*, *16*, 2078–2079.

See Also

`baminfo` | `samread` | `saminfo` | `soapread` | `fastqwrite` | `fastqinfo` | `fastainfo` | `fastaread` | `fastawrite` | `sffinfo` | `sffread` | `fastqread` | `BioIndexedFile` | `BioMap`

Topics

"Manage Sequence Read Data in Objects"
"Work with Next-Generation Sequencing Data"

External Websites

Sequence Read Archive
SAM format specification

bamsort

Sort BAM files

Syntax

```
sortedFile = bamsort(inFile)
bamsort(inFile,outFile)
```

Description

`sortedFile = bamsort(inFile)` sorts a BAM file `inFile` and returns the name of the sorted BAM file `sortedFile`. The function sorts the alignment records by the reference sequence name first, and then by position within the reference.

`bamsort(inFile,outFile)` sorts `inFile` and produces a sorted BAM file named `outFile`.

Examples

Sort BAM File

Sort a sample BAM file. The sorted file has the same base name as the input file, but with the extension `".sorted.bam"`. By default, the sorted file is saved in the current directory.

```
sortedFile = bamsort("ex1.bam")
```

```
sortedFile = 
"ex1.sorted.bam"
```

You can change the name of output file by specifying it as the second input.

```
bamsort("ex1.bam","sortedEx1.bam")
```

```
ans = 
"sortedEx1.bam"
```

You can also save the output file to an existing directory by providing the file path information.

```
mkdir("./OutputEx1BAM");
bamsort("ex1.bam","./OutputEx1BAM/sortedEx1.bam")
```

```
ans = 
"./OutputEx1BAM/sortedEx1.bam"
```

Input Arguments

inFile — Name of input BAM file to sort
character vector | string

Name of the input BAM file to sort, specified as a string or character vector. You can specify a file name or a path and file name. The file name must have the extension `.bam`.

Example: `"./InputData/ex1.bam"`

Data Types: `char` | `string`

`outFile` — Name of output BAM file
character vector | string

Name of the output BAM file, specified as a string or character vector. You can specify a file name or a path and file name. The file name must have the extension `.bam`. The file is saved in the current directory by default unless you specify the path information. If you specify the file path, the listed directories must exist before you run the function.

Example: `"./OutputData/ex1Sorted.bam"`

Data Types: `char` | `string`

Output Arguments

`sortedFile` — Name of output BAM file
string

Name of the output BAM file, returned as a string. `sortedFile` has the same base name as `inFile` but with the extension `.sorted.bam`. The file is saved in the current directory by default.

Version History

Introduced in R2019b

See Also

`samsort` | `bamread` | `samread` | `BioMap` | `baminfo`

Topics
"Data Import"
"Manage Sequence Read Data in Objects"

External Websites
Sequence Read Archive
SAM format specification

basecount

Count nucleotides in sequence

Syntax

```
NTStruct = basecount(SeqNT)
NTStruct = basecount(SeqNT, ...'Ambiguous', AmbiguousValue, ...)
NTStruct = basecount(SeqNT, ...'Gaps', GapsValue, ...)
NTStruct = basecount(SeqNT, ...'Chart', ChartValue, ...)
```

Input Arguments

SeqNT	One of the following: • Character vector or string containing codes specifying a nucleotide sequence. For valid letter codes, see the table Mapping Nucleotide Letter Codes to Integers • Row vector of integers specifying a nucleotide sequence. For valid integers, see the table Mapping Nucleotide Integers to Letter Codes • MATLAB structure containing a `Sequence` field that contains a nucleotide sequence, such as returned by `fastaread`, `fastqread`, `emblread`, `getembl`, `genbankread`, or `getgenbank`.
AmbiguousValue	Character vector or string specifying how to treat ambiguous nucleotide characters (`R`, `Y`, `K`, `M`, `S`, `W`, `B`, `D`, `H`, `V`, or `N`). Choices are: • `'ignore'` (default) — Skips ambiguous characters • `'bundle'` — Counts ambiguous characters and reports the total count in the `Ambiguous` field. • `'prorate'` — Counts ambiguous characters and distributes them proportionately in the appropriate fields. For example, the counts for the character `R` are distributed evenly between the `A` and `G` fields. • `'individual'` — Counts ambiguous characters and reports them in individual fields. • `'warn'` — Skips ambiguous characters and displays a warning.
GapsValue	Specifies whether gaps, indicated by a hyphen (`-`), are counted or ignored. Choices are `true` or `false` (default).
ChartValue	Character vector or string specifying a chart type. Choices are `'pie'` or `'bar'`.

Output Arguments

NTStruct	1-by-1 MATLAB structure containing the fields `A`, `C`, `G`, and `T`.

Description

`NTStruct = basecount(SeqNT)` counts the number of each type of base in `SeqNT`, a nucleotide sequence, and returns the counts in `NTStruct`, a 1-by-1 MATLAB structure containing the fields `A`, `C`, `G`, and `T`.

- The character `U` is added to the `T` field.
- Ambiguous nucleotide characters (`R`, `Y`, `K`, `M`, `S`, `W`, `B`, `D`, `H`, `V`, or `N`), and gaps, indicated by a hyphen (`-`), are ignored by default.
- Unrecognized characters are ignored and cause the following warning message.

  ```
  Warning: Unknown symbols appear in the sequence. These will be ignored.
  ```

`NTStruct = basecount(SeqNT, ...'PropertyName', PropertyValue, ...)` calls `basecount` with optional properties that use property name/property value pairs. You can specify one or more properties in any order. Each *`PropertyName`* must be enclosed in single quotation marks and is case insensitive. These property name/property value pairs are as follows:

`NTStruct = basecount(SeqNT, ...'Ambiguous', AmbiguousValue, ...)` specifies how to treat ambiguous nucleotide characters (`R`, `Y`, `K`, `M`, `S`, `W`, `B`, `D`, `H`, `V`, or `N`). Choices are:

- `'ignore'` (default)
- `'bundle'`
- `'prorate'`
- `'individual'`
- `'warn'`

`NTStruct = basecount(SeqNT, ...'Gaps', GapsValue, ...)` specifies whether gaps, indicated by a hyphen (`-`), are counted or ignored. Choices are `true` or `false` (default).

`NTStruct = basecount(SeqNT, ...'Chart', ChartValue, ...)` creates a chart showing the relative proportions of the nucleotides. *`ChartValue`* can be `'pie'` or `'bar'`.

Examples

Count nucleotides in a sequence

Count the bases in a DNA sequence and return the results in a structure.

```
bases = basecount('TAGCTGGCCAAGCGAGCTTG')
```

```
bases = struct with fields:
    A: 4
    C: 5
    G: 7
    T: 4
```

Get the count for adenosine (A) bases.

```
bases.A
```

```
ans = 4
```

Count the bases in a DNA sequence containing ambiguous characters (R, Y, K, M, S, W, B, D, H, V, or N), listing each of them in a separate field.

```
basecount('ABCDGGCCAAGCGAGCTTG','Ambiguous','individual')
```

```
ans = struct with fields:
    A: 4
    C: 5
    G: 6
    T: 2
    R: 0
    Y: 0
    K: 0
    M: 0
    S: 0
    W: 0
    B: 1
    D: 1
    H: 0
    V: 0
    N: 0
```

Version History

Introduced before R2006a

See Also

aacount | baselookup | codoncount | cpgisland | dimercount | nmercount | ntdensity | seqviewer

baselookup

Find nucleotide codes, integers, names, and complements

Syntax

```
baselookup
baselookup('Complement', SeqNT)
baselookup('Code', CodeValue)
baselookup('Integer', IntegerValue)
baselookup('Name', NameValue)
```

Arguments

SeqNT	Nucleotide sequence(s) represented by one of the following: • Character vector or string containing single-letter codes from the table Nucleotide Lookup • Cell array of sequences • Two-dimensional character array of sequences **Note** If the input is multiple sequences, the complement for each sequence is determined independently.
CodeValue	Nucleotide letter code represented by one of the following: • Character vector or string specifying a single-letter code representing a nucleotide. For valid single-letter codes, see the table Nucleotide Lookup. • Cell array of letter codes. • Two-dimensional character array of letter codes.
IntegerValue	Single integer representing a nucleotide. For valid integers, see the table Nucleotide Lookup.
NameValue	Nucleotide name represented by one of the following: • Character vector or string specifying a nucleotide name. For valid nucleotide names, see the table Nucleotide Lookup. • Cell array of names. • Two-dimensional character array of names.

Description

`baselookup` displays a table of nucleotide codes, integers, names, and complements.

Nucleotide Lookup

Code	Integer	Nucleotide Name	Meaning	Complement
A	1	Adenine	A	T
C	2	Cytosine	C	G
G	3	Guanine	G	C
T	4	Thymine	T	A
U	4	Uracil	U	A
R	5	Purine	A or G	Y
Y	6	Pyrimidine	C or T	R
K	7	Keto	G or T	M
M	8	Amino	A or C	K
S	9	Strong interaction (3 H bonds)	C or G	S
W	10	Weak interaction (2 H bonds)	A or T	W
B	11	Not A	C or G or T	V
D	12	Not C	A or G or T	H
H	13	Not G	A or C or T	D
V	14	Not T or U	A or C or G	B
N, X	15	Any nucleotide	A or C or G or T or U	N
-	16	Gap of indeterminate length	Gap	-

`baselookup('Complement', SeqNT)` displays the complementary nucleotide sequence.

`baselookup('Code', CodeValue)` displays the corresponding meaning and nucleotide name. For ambiguous nucleotide codes (R, Y, K, M, S, W, B, D, H, V, N, and X), the nucleotide name is a descriptive name.

`baselookup('Integer', IntegerValue)` displays the corresponding letter code, meaning, and nucleotide name.

`baselookup('Name', NameValue)` displays the corresponding letter code, meaning, and nucleotide name or descriptive name.

Examples

Convert a nucleotide sequence to its complementary sequence

```
baselookup('Complement', 'TAGCTGRCCAAGGCCAAGCGAGCTTN')
```

```
ans =
    'ATCGACYGGTTCCGGTTCGCTCGAAN
     '
```

Display the meaning and nucleotide name or descriptive name for the nucleotide codes G and Y.

```
baselookup('Code', 'G')
```

```
ans =
    'G    Guanine
     '
```

```
baselookup('Code', 'Y')
```

```
ans =
    'T|C    pYrimidine
     '
```

Display the nucleotide letter code, meaning, and nucleotide name or descriptive name for the integers 1 and 7.

```
baselookup('Integer', 1)
```

```
ans =
    'A    A - Adenine
     '
```

```
baselookup('Integer', 7)
```

```
ans =
    'K    G|T - Keto
     '
```

Display the corresponding nucleotide letter code, meaning, and name for cytosine and purine.

```
baselookup('Name','cytosine')
```

```
ans =
    'C    C - Cytosine
     '
```

```
baselookup('Name','purine')
```

```
ans =
    'R    G|A - puRine
     '
```

Version History

Introduced before R2006a

See Also

aa2nt | basecount | codoncount | dimercount | geneticcode | int2nt | nt2aa | nt2int | revgeneticcode | seqviewer

biograph object

(Removed) Data structure containing generic interconnected data used to implement directed graph

Note and its methods have been removed. Use `graph` or `digraph` instead. For details, see “Version History”.

Description

A biograph object is a data structure containing generic interconnected data used to implement a directed graph. Nodes represent proteins, genes, or any other biological entity, and edges represent interactions, dependences, or any other relationship between the nodes. A biograph object also stores information, such as color properties and text label characteristics, used to create a 2-D visualization of the graph.

You create a biograph object using the object constructor function `biograph`. You can view a graphical representation of a biograph object using the `view` method.

Method Summary

Following are methods of a biograph object:

allshortestpaths (biograph)	(Removed) Find all shortest paths in biograph object
conncomp (biograph)	(Removed) Find strongly or weakly connected components in biograph object
dolayout (biograph)	(Removed) Calculate node positions and edge trajectories
get (biograph)	(To be removed) Retrieve information about biograph object
getancestors (biograph)	(Removed) Find ancestors of a node in biograph object
getdescendants (biograph)	(Removed) Find descendants of a node in biograph object
getedgesbynodeid (biograph)	(Removed) Get handles to edges in biograph object
getmatrix (biograph)	(Removed) Get connection matrix from biograph object
getweightmatrix (biograph)	(Removed) Get connection matrix with weights from biograph object
getnodesbyid (biograph)	(Removed) Get handles to nodes
getrelatives (biograph)	(Removed) Find relatives of a node in biograph object
isdag (biograph)	(Removed) Test for cycles in biograph object
isomorphism (biograph)	(Removed) Find isomorphism between two biograph objects
isspantree (biograph)	(Removed) Determine if tree created from biograph object is spanning tree
maxflow (biograph)	(Removed) Calculate maximum flow in biograph object
minspantree (biograph)	(Removed) Find minimal spanning tree in biograph object
set (biograph)	(To be removed) Set property of biograph object
shortestpath (biograph)	(Removed) Solve shortest path problem in biograph object
topoorder (biograph)	(Removed) Perform topological sort of directed acyclic graph extracted from biograph object
traverse (biograph)	(Removed) Traverse biograph object by following adjacent nodes
view (biograph)	(Removed) Draw figure from biograph object

Following are methods of a node object:

getancestors (biograph)	(Removed) Find ancestors of a node in biograph object
getdescendants (biograph)	(Removed) Find descendants of a node in biograph object
getrelatives (biograph)	(Removed) Find relatives of a node in biograph object

Property Summary

A biograph object contains two kinds of objects, node objects and edge objects, that have their own properties. For a list of the properties of node objects and edge objects, see the following tables.

Properties of a Biograph Object

Property	Description
`ID`	Character vector to identify the biograph object. Default is `''`.
`Label`	Character vector to label the biograph object. Default is `''`.
`Description`	Character vector that describes the biograph object. Default is `''`.
`LayoutType`	Character vector that specifies the algorithm for the layout engine. Choices are: • `'hierarchical'` (default) — Uses a topological order of the graph to assign levels, and then arranges the nodes from top to bottom, while minimizing crossing edges. • `'radial'` — Uses a topological order of the graph to assign levels, and then arranges the nodes from inside to outside of the circle, while minimizing crossing edges. • `'equilibrium'` — Calculates layout by minimizing the energy in a dynamic spring system.
`EdgeType`	Character vector that specifies how edges display. Choices are: • `'straight'` • `'curved'` (default) • `'segmented'` **Note** Curved or segmented edges occur only when necessary to avoid obstruction by nodes. Biograph objects with `LayoutType` equal to `'equilibrium'` or `'radial'` cannot produce curved or segmented edges.
`Scale`	Positive number that post-scales the node coordinates. Default is `1`.
`LayoutScale`	Positive number that scales the size of the nodes before calling the layout engine. Default is `1`.
`EdgeTextColor`	Three-element numeric vector of RGB values. Default is `[0, 0, 0]`, which defines black.
`EdgeFontSize`	Positive number that sets the size of the edge font in points. Default is `8`.
`ShowArrows`	Controls the display of arrows with the edges. Choices are `'on'` (default) or `'off'`.
`ArrowSize`	Positive number that sets the size of the arrows in points. Default is `8`.
`ShowWeights`	Controls the display of text indicating the weight of the edges. Choices are `'on'` or `'off'` (default).

Property	Description
`ShowTextInNodes`	Character vector that specifies the node property used to label nodes when you display a biograph object using the `view` method. Choices are: • `'Label'` — Uses the `Label` property of the node object (default). • `'ID'` — Uses the `ID` property of the node object. • `'None'`
`NodeAutoSize`	Controls precalculating the node size before calling the layout engine. Choices are `'on'` (default) or `'off'`. **Note** Set it to `off` if you want to apply different node sizes by changing the `Size` property.
`NodeCallback`	User-defined callback for all nodes. Enter the name of a function, a function handle, or a cell array with multiple function handles. After using the `view` function to display the biograph object in the Biograph Viewer, you can double-click a node to activate the first callback, or right-click and select a callback to activate. Default is the anonymous function, `@(node) inspect(node)`, which displays the Property Inspector dialog box.
`EdgeCallback`	User-defined callback for all edges. Enter the name of a function, a function handle, or a cell array with multiple function handles. After using the `view` function to display the biograph object in the Biograph Viewer, you can right-click and select a callback to activate. Default is the anonymous function, `@(edge) inspect(edge)`, which displays the Property Inspector dialog box.
`CustomNodeDrawFcn`	Function handle to a customized function to draw nodes. Default is `[]`.
`Nodes`	Read-only column vector with handles to node objects of a biograph object. The size of the vector is the number of nodes. For properties of node objects, see Properties of a Node Object.
`Edges`	Read-only column vector with handles to edge objects of a biograph object. The size of the vector is the number of edges. For properties of edge objects, see Properties of an Edge Object.

Properties of a Node Object

Property	Description
`ID`	Character vector defined when the biograph object is created, either by the *`NodeIDs`* input argument or internally by the `biograph` constructor function. You can modify this property using the `set` method, but each node object's `ID` must be unique.
`Label`	Character vector for labeling a node when you display a biograph object using the `view` method. Default is `''`.
`Description`	Character vector that describes the node. Default is `''`.
`Position`	Two-element numeric vector of *x*- and *y*-coordinates, for example, `[150, 150]`. If you do not specify this property, default is initially `[]`, then when the layout algorithms are executed, it becomes a two-element numeric vector of *x*- and *y*-coordinates computed by the layout engine.
`Shape`	Character vector that specifies the shape of the nodes. Choices are: • `'box'`(default) • `'ellipse'` • `'circle'` • `'rect'` or `'rectangle'` • `'diamond'` • `'trapezium'` • `'invtrapezium'` • `'house'` • `'invhouse'` • `'parallelogram'`
`Size`	Two-element numeric vector calculated before calling the layout engine using the actual font size and shape of the node. Default is `[10, 10]`.
`Color`	Three-element numeric vector of RGB values that specifies the fill color of the node. Default is `[1, 1, 0.7]`, which defines yellow.
`LineWidth`	Positive number. Default is `1`.
`LineColor`	Three-element numeric vector of RGB values that specifies the outline color of the node. Default is `[0.3, 0.3, 1]`, which defines blue.
`FontSize`	Positive number that sets the size of the node font in points. Default is `8`.
`TextColor`	Three-element numeric vector of RGB values that specifies the color of the node labels. Default is `[0, 0, 0]`, which defines black.
`UserData`	Miscellaneous, user-defined data that you want to associate with the node. The node does not use this property, but you can access and specify it using the `get` and `set` functions. Default is `[]`.

Properties of an Edge Object

Property	Description
`ID`	Character vector automatically generated from the node `ID`s when the biograph object is created by the `biograph` constructor function. You can modify this property using the `set` method, but each edge object's `ID` must be unique.
`Label`	Character vector for labeling an edge. Default is `''`.
`Description`	Character vector that describes the edge. Default is `''`.
`Weight`	Value that represents the weight (cost, distance, length, or capacity) associated with the edge. Default is `1`.
`LineWidth`	Positive number. Default is `0.5`.
`LineColor`	Three-element numeric vector of RGB values that specifies the color of the edge. Default is `[0.5, 0.5, 0.5]`, which defines gray.
`UserData`	Miscellaneous, user-defined data that you want to associate with the edge. The edge does not use this property, but you can access and specify it using the `get` and `set` functions. Default is `[]`.

Version History

Introduced in R2006b

R2022b: Removed

Errors starting in R2022b

The `biograph object` and its methods have been removed. See the following table for alternative functions from `graph` and `digraph` objects.

Methods of biograph	Alternative Functions from graph and digraph Objects
`allshortestpaths (biograph)`	`distances`
`conncomp (biograph)`	`conncomp`
`dolayout (biograph)`	Not applicable
`getancestors (biograph)`	`predecessors`. Note that the function does not let you specify `NumGenerations`. You can use `distances` to find the distances from all nodes to the target node after setting the edge weight to 1, and filter the nodes by distance to the appropriate depth.
`getdescendants (biograph)`	`successors`. If you need to specify `NumGenerations`, see the workaround listed above for `predecessors`.
`getedgesbynodeid (biograph)`	`findedge`
`getmatrix (biograph)`	`adjacency`
`getnodesbyid (biograph)`	`findnode`
`getrelatives (biograph)`	`neighbors`

Methods of biograph	Alternative Functions from graph and digraph Objects
`getweightmatrix (biograph)`	`adjacency`
`isdag (biograph)`	`isdag`
`isomorphism (biograph)`	`isomorphism`
`isspantree (biograph)`	A graph is a spanning tree if and only if all nodes are reachable from an arbitrary start node, and `|E| == |N| - 1`, where *E* is the number of edges and *N* is the number of nodes. You can use either `bfsearch` or `dfsearch` to check such conditions are true for a given graph.
`maxflow (biograph)`	`maxflow`
`minspantree (biograph)`	`minspantree`
`shortestpath (biograph)`	`shortestpath`
`topoorder (biograph)`	`toposort`
`traverse (biograph)`	`bfsearch` or `dfsearch`
`view (biograph)`	`plot`. The name-value arguments of the `plot` function can be used to control the properties of the graph. The arguments can be used as replacements for many of the `biograph` object properties to change the appearance of the graph.

R2022a: Warns
Warns starting in R2022a

`biograph object` issues a warning that it will be removed in a future release.

R2021b: To be removed
Not recommended starting in R2021b

`biograph object` runs without warning, but it will be removed in a future release.

See Also
`graph` | `digraph`

biograph

(Removed) Create biograph object

Note has been removed. Use `graph` or `digraph` instead. For details, see “Version History”.

Syntax

```
BGobj = biograph(CMatrix)
BGobj = biograph(CMatrix, NodeIDs)

BGobj = biograph(CMatrix, NodeIDs, ...'ID', IDValue, ...)
BGobj = biograph(CMatrix, NodeIDs, ...'Label', LabelValue, ...)
BGobj = biograph(CMatrix, NodeIDs, ...'Description', DescriptionValue, ...)
BGobj = biograph(CMatrix, NodeIDs, ...'LayoutType', LayoutTypeValue, ...)
BGobj = biograph(CMatrix, NodeIDs, ...'EdgeType', EdgeTypeValue, ...)
BGobj = biograph(CMatrix, NodeIDs, ...'Scale', ScaleValue, ...)
BGobj = biograph(CMatrix, NodeIDs, ...'LayoutScale', LayoutScaleValue, ...)
BGobj = biograph(CMatrix, NodeIDs, ...'EdgeTextColor',
EdgeTextColorValue, ...)
BGobj = biograph(CMatrix, NodeIDs, ...'EdgeFontSize', EdgeFontSizeValue, ...)
BGobj = biograph(CMatrix, NodeIDs, ...'ShowArrows', ShowArrowsValue, ...)
BGobj = biograph(CMatrix, NodeIDs, ...'ArrowSize', ArrowSizeValue, ...)
BGobj = biograph(CMatrix, NodeIDs, ...'ShowWeights', ShowWeightsValue, ...)
BGobj = biograph(CMatrix, NodeIDs, ...'ShowTextInNodes',
ShowTextInNodesValue, ...)
BGobj = biograph(CMatrix, NodeIDs, ...'NodeAutoSize', NodeAutoSizeValue, ...)
BGobj = biograph(CMatrix, NodeIDs, ...'NodeCallback', NodeCallbackValue, ...)
BGobj = biograph(CMatrix, NodeIDs, ...'EdgeCallback', EdgeCallbackValue, ...)
BGobj = biograph(CMatrix, NodeIDs, ...'CustomNodeDrawFcn',
CustomNodeDrawFcnValue, ...)
```

Arguments

CMatrix	Full or sparse square matrix that acts as a connection matrix. That is, a value of `1` indicates a connection between nodes while a `0` indicates no connection. The number of rows/columns is equal to the number of nodes.

`NodeIDs`	Node labels. Enter any of the following: • Cell array of character vectors or string vector with the number of character vectors (or strings) equal to the number of rows or columns in the connection matrix *`CMatrix`*. Each character vector (or string) must be unique. • Character array with the number of rows equal to the number of nodes. Each row in the array must be unique. • Character vector or string with the number of characters equal to the number of nodes. Each character must be unique. Default values are the row or column numbers. **Note** You must specify *`NodeIDs`* if you want to specify property name/value pairs. Set *`NodeIDs`* to `[]` to use the default values of the row/column numbers.
`IDValue`	Character vector or string to identify the biograph object. Default is `''`.
`LabelValue`	Character vector or string to label the biograph object. Default is `''`.
`DescriptionValue`	Character vector or string that describes the biograph object. Default is `''`.
`LayoutTypeValue`	Character vector or string that specifies the algorithm for the layout engine. Choices are: • `'hierarchical'` (default) — Uses a topological order of the graph to assign levels, and then arranges the nodes from top to bottom, while minimizing crossing edges. • `'radial'` — Uses a topological order of the graph to assign levels, and then arranges the nodes from inside to outside of the circle, while minimizing crossing edges. • `'equilibrium'` — Calculates layout by minimizing the energy in a dynamic spring system.
`EdgeTypeValue`	Character vector or string that specifies how edges display. Choices are: • `'straight'` • `'curved'` (default) • `'segmented'` **Note** Curved or segmented edges occur only when necessary to avoid obstruction by nodes. Biograph objects with `LayoutType` equal to `'equilibrium'` or `'radial'` cannot produce curved or segmented edges.

ScaleValue	Positive number that post-scales the node coordinates. Default is `1`.
LayoutScaleValue	Positive number that scales the size of the nodes before calling the layout engine. Default is `1`.
EdgeTextColorValue	Three-element numeric vector of RGB values. Default is `[0, 0, 0]`, which defines black.
EdgeFontSizeValue	Positive number that sets the size of the edge font in points. Default is `8`.
ShowArrowsValue	Controls the display of arrows for the edges. Choices are `'on'` (default) or `'off'`.
ArrowSizeValue	Positive number that sets the size of the arrows in points. Default is `8`.
ShowWeightsValue	Controls the display of text indicating the weight of the edges. Choices are `'on'` or `'off'` (default).
ShowTextInNodesValue	Character vector or string that specifies the node property used to label nodes when you display a biograph object using the `view` method. Choices are: • `'Label'` — Uses the `Label` property of the node object (default). • `'ID'` — Uses the `ID` property of the node object. • `'None'`
NodeAutoSizeValue	Controls precalculating the node size before calling the layout engine. Choices are `'on'` (default) or `'off'`. **Note** Set it to `off` if you want to apply different node sizes by changing the `Size` property.
NodeCallbackValue	User callback for all nodes. Enter the name of a function, a function handle, or a cell array with multiple function handles. After using the `view` function to display the biograph in the Biograph Viewer, you can double-click a node to activate the first callback, or right-click and select a callback to activate. Default is `@(node) inspect(node)`, which displays the Property Inspector dialog box.
EdgeCallbackValue	User callback for all edges. Enter the name of a function, a function handle, or a cell array with multiple function handles. After using the `view` function to display the biograph object in the Biograph Viewer, you can right-click and select a callback to activate. Default is the anonymous function, `@(edge) inspect(edge)`, which displays the Property Inspector dialog box.
CustomNodeDrawFcnValue	Function handle to a customized function to draw nodes. Default is `[]`.

Description

BGobj = biograph(*CMatrix*) creates a biograph object, *BGobj*, using a connection matrix, *CMatrix*. All nondiagonal and positive entries in the connection matrix, *CMatrix*, indicate connected nodes, rows represent the source nodes, and columns represent the sink nodes.

BGobj = biograph(*CMatrix*, *NodeIDs*) specifies the node labels. *NodeIDs* can be:

- Cell array of character vectors or string vector with the number of character vectors (or strings) equal to the number of rows or columns in the connection matrix *CMatrix*. Each character vector or string must be unique.
- Character array with the number of rows equal to the number of nodes. Each row in the array must be unique.
- Character vector or string with the number of characters equal to the number of nodes. Each character must be unique.

Default values are the row or column numbers.

Note If you want to specify property name/value pairs, you must specify *NodeIDs*. Set *NodeIDs* to [] to use the default values of the row/column numbers.

BGobj = biograph(..., '*PropertyName*', *PropertyValue*, ...) calls biograph with optional properties that use property name/property value pairs. You can specify one or more properties in any order. Each *PropertyName* must be enclosed in single quotation marks and is case insensitive. These property name/property value pairs are as follows:

BGobj = biograph(*CMatrix*, *NodeIDs*, ...'ID', *IDValue*, ...) specifies an ID for the biograph object. Default is ''.

BGobj = biograph(*CMatrix*, *NodeIDs*, ...'Label', *LabelValue*, ...) specifies a label for the biograph object. Default is ''.

BGobj = biograph(*CMatrix*, *NodeIDs*, ...'Description', *DescriptionValue*, ...) specifies a description of the biograph object. Default is ''.

BGobj = biograph(*CMatrix*, *NodeIDs*, ...'LayoutType', *LayoutTypeValue*, ...) specifies the algorithm for the layout engine.

BGobj = biograph(*CMatrix*, *NodeIDs*, ...'EdgeType', *EdgeTypeValue*, ...) specifies how edges display.

BGobj = biograph(*CMatrix*, *NodeIDs*, ...'Scale', *ScaleValue*, ...) post-scales the node coordinates. Default is 1.

BGobj = biograph(*CMatrix*, *NodeIDs*, ...'LayoutScale', *LayoutScaleValue*, ...) scales the size of the nodes before calling the layout engine. Default is 1.

BGobj = biograph(*CMatrix*, *NodeIDs*, ...'EdgeTextColor', *EdgeTextColorValue*, ...) specifies a three-element numeric vector of RGB values. Default is [0, 0, 0], which defines black.

BGobj = biograph(*CMatrix*, *NodeIDs*, ...'EdgeFontSize', *EdgeFontSizeValue*, ...) sets the size of the edge font in points. Default is 8.

BGobj = biograph(*CMatrix*, *NodeIDs*, ...'ShowArrows', *ShowArrowsValue*, ...) controls the display of arrows for the edges. Choices are 'on' (default) or 'off'.

BGobj = biograph(*CMatrix*, *NodeIDs*, ...'ArrowSize', *ArrowSizeValue*, ...) sets the size of the arrows in points. Default is 8.

BGobj = biograph(*CMatrix*, *NodeIDs*, ...'ShowWeights', *ShowWeightsValue*, ...) controls the display of text indicating the weight of the edges. Choices are 'on' (default) or 'off'.

BGobj = biograph(*CMatrix*, *NodeIDs*, ...'ShowTextInNodes', *ShowTextInNodesValue*, ...) specifies the node property used to label nodes when you display a biograph object using the view method.

BGobj = biograph(*CMatrix*, *NodeIDs*, ...'NodeAutoSize', *NodeAutoSizeValue*, ...) controls precalculating the node size before calling the layout engine. Choices are 'on' (default) or 'off'.

BGobj = biograph(*CMatrix*, *NodeIDs*, ...'NodeCallback', *NodeCallbackValue*, ...) specifies user callback for all nodes.

BGobj = biograph(*CMatrix*, *NodeIDs*, ...'EdgeCallback', *EdgeCallbackValue*, ...) specifies user callback for all edges.

BGobj = biograph(*CMatrix*, *NodeIDs*, ...'CustomNodeDrawFcn', *CustomNodeDrawFcnValue*, ...) specifies function handle to customized function to draw nodes. Default is [].

Version History

Introduced in R2006a

R2022b: biograph has been removed

Errors starting in R2022b

biograph has been removed. See the following table for alternative functions from graph and digraph objects.

Methods of biograph	Alternative Functions from graph and digraph Objects
allshortestpaths (biograph)	distances
conncomp (biograph)	conncomp
dolayout (biograph)	Not applicable
getancestors (biograph)	predecessors. Note that the function does not let you specify NumGenerations. You can use distances to find the distances from all nodes to the target node after setting the edge weight to 1, and filter the nodes by distance to the appropriate depth.
getdescendants (biograph)	successors. If you need to specify NumGenerations, see the workaround listed above for predecessors.

Methods of biograph	Alternative Functions from graph and digraph Objects
`getedgesbynodeid (biograph)`	`findedge`
`getmatrix (biograph)`	`adjacency`
`getnodesbyid (biograph)`	`findnode`
`getrelatives (biograph)`	`neighbors`
`getweightmatrix (biograph)`	`adjacency`
`isdag (biograph)`	`isdag`
`isomorphism (biograph)`	`isomorphism`
`isspantree (biograph)`	A graph is a spanning tree if and only if all nodes are reachable from an arbitrary start node, and `|E| == |N| - 1`, where *E* is the number of edges and *N* is the number of nodes. You can use either `bfsearch` or `dfsearch` to check such conditions are true for a given graph.
`maxflow (biograph)`	`maxflow`
`minspantree (biograph)`	`minspantree`
`shortestpath (biograph)`	`shortestpath`
`topoorder (biograph)`	`toposort`
`traverse (biograph)`	`bfsearch` or `dfsearch`
`view (biograph)`	`plot`. The name-value arguments of the `plot` function can be used to control the properties of the graph. The arguments can be used as replacements for many of the `biograph` object properties to change the appearance of the graph.

R2022a: Warns
Warns starting in R2022a

`biograph` issues a warning that it will be removed in a future release.

R2021b: To be removed
Not recommended starting in R2021b

`biograph` runs without warning, but it will be removed in a future release.

See Also
`graph` | `digraph`

BioIndexedFile class

Allow quick and efficient access to large text file with nonuniform-size entries

Description

The `BioIndexedFile` class allows access to text files with nonuniform-size entries, such as sequences, annotations, and cross-references to data sets. It lets you quickly and efficiently access this data without loading the source file into memory.

This class lets you access individual entries or a subset of entries when the source file is too big to fit into memory. You can access entries using indices or keys. You can read and parse one or more entries using provided interpreters or a custom interpreter function.

Construction

`BioIFobj = BioIndexedFile(Format,SourceFile)` returns a `BioIndexedFile` object `BioIFobj` that indexes the contents of `SourceFile` following the parsing rules defined by `Format`, where `SourceFile` and `Format` specify the names of a text file and a file format, respectively. It also constructs an auxiliary index file to store information that allows efficient, direct access to `SourceFile`. The index file by default is stored in the same location as the source file and has the same name as the source file, but with an IDX extension. The `BioIndexedFile` constructor uses the index file to construct subsequent objects from `SourceFile`, which saves time.

`BioIFobj = BioIndexedFile(Format,SourceFile,IndexDir)` returns a `BioIndexedFile` object `BioIFobj` by specifying the relative or absolute path to a folder to use when searching for or saving the index file.

`BioIFobj = BioIndexedFile(Format,SourceFile,IndexFile)` returns a `BioIndexedFile` object `BioIFobj` by specifying a file name, optionally including a relative or absolute path, to use when searching for or saving the index file.

`BioIFobj = BioIndexedFile( ___ ,Name,Value)` returns a `BioIndexedFile` object `BioIFobj` by using any input arguments from the previous syntaxes and additional options, specified as one or more `Name,Value` pair arguments.

Input Arguments

`Format`

Character vector or string specifying a file format. Choices are:

- `'SAM'` — SAM-formatted file
- `'FASTQ'` — FASTQ-formatted file
- `'FASTA'` — FASTA-formatted file
- `'TABLE'` — Tab-delimited table with multiple columns. Keys can be in any column. Rows with the same key are considered separate entries.

- `'MRTAB'` — Tab-delimited table with multiple columns. Keys can be in any column. Contiguous rows with the same key are considered a single entry. Noncontiguous rows with the same key are considered separate entries.
- `'FLAT'` — Flat file with concatenated entries separated by a character vector, typically `'//'`. Within an entry, the key is separated from the rest of the entry by a white space.

Note For all file formats, the file contents must only use ASCII text characters. Non-ASCII characters may not be properly indexed.

Default:

`SourceFile`

Character vector or string specifying the name of a text file. It can include a relative or absolute path.

`IndexDir`

Character vector or string specifying the relative or absolute path to a folder to use when searching for or saving the index file.

`IndexFile`

Character vector or string specifying a file name, optionally including a relative or absolute path, to use when searching for or saving the index file.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

`IndexedByKeys`

Specifies if you can access the object `BioIFobj` using keys. Choices are `true` or `false`.

Tip Set the value to `false` if you do not need to access entries in the object using keys. Doing so saves time and space when creating the object.

Default: `true`

`MemoryMappedIndex`

Specifies whether the constructor stores the indices in the auxiliary index file and accesses them via memory maps (`true`) or loads the indices into memory at construction time (`false`).

Tip If memory is not an issue and you want to maximize performance when accessing entries in the object, set the value to `false`.

Default: `true`

`Interpreter`

Handle to a function that the `read` method uses when parsing entries from the source file. The interpreter function must accept a character vector of one or more concatenated entries and return a structure or an array of structures containing the interpreted data.

When `Format` is a general-purpose format such as `'TABLE'`, `'MRTAB'`, or `'FLAT'`, then the default is `[]`, which means the function is an anonymous function in which the output is equivalent to the input.

When `Format` is an application-specific format such as `'SAM'`, `'FASTQ'`, or `'FASTA'`, then the default is a function handle appropriate for that file type and typically does not require you to change it.

Default:

`Verbose`

Controls the display of the status of the object construction. Choices are `true` or `false`.

Default: `true`

Note The following name-value pair arguments apply only when both of the following are true:

- There is no pre-existing index file associated with your source file.
- Your source file has a general-purpose format such as `'TABLE'`, `'MRTAB'`, or `'FLAT'`.

For source files with application-specific formats, the following name-value pairs are pre-defined and you cannot change them.

`KeyColumn`

Positive integer specifying the column in the `'TABLE'` or `'MRTAB'` file that contains the keys.

Default: `1`

`KeyToken`

Character vector or string that occurs in each entry before the key, for `'FLAT'` files that contain keys. If the value is `' '`, it indicates the key is the first character vector (or string) in each entry and is delimited by blank spaces.

Default: `' '`

`HeaderPrefix`

Character vector or string specifying a prefix that denotes header lines in the source file so the constructor ignores them when creating the object. If the value is `[]`, it means the constructor does not check for header lines in the source file.

Default: `[]`

`CommentPrefix`

Character vector or string specifying a prefix that denotes comment lines in the source file so the constructor ignores them when creating the object. If the value is `[]`, it means the constructor does not check for comment lines in the source file.

Default: `[]`

`ContiguousEntries`

Specifies whether entries are on contiguous lines, which means they are not separated by empty lines or comment lines, in the source file or not. Choices are `true` or `false`.

Tip Set the value to `true` when entries are not separated by empty lines or comment lines. Doing so saves time and space when creating the object.

Default: `false`

`TableDelimiter`

Character vector or string specifying a delimiter symbol to use as a column separator for `SourceFile` when `Format` is `'TABLE'` or `'MRTAB'`. Choices are `'\t'` (horizontal tab), `' '` (blank space), or `','`, (comma).

Default: `'\t'`

`EntryDelimiter`

Character vector or string specifying a delimiter symbol to use as an entry separator for `SourceFile` when `Format` is `'FLAT'`.

Default: `'//'`

Properties

`FileFormat`

File format of the source file

This information is read only. Possible values are:

- `'SAM'` — SAM-formatted file
- `'FASTQ'` — FASTQ-formatted file
- `'FASTA'` — FASTA-formatted file
- `'TABLE'` — Tab-delimited table with multiple columns. Keys can be in any column. Rows with the same key are considered separate entries.
- `'MRTAB'` — Tab-delimited table with multiple columns. Keys can be in any column. Contiguous rows with the same key are considered a single entry. Noncontiguous rows with the same key are considered separate entries.
- `'FLAT'` — Flat file with concatenated entries separated by a character vector, typically `'//'`. Within an entry, the key is separated from the rest of the entry by a white space.

`IndexedByKeys`

Whether or not the entries in the source file can be indexed by an alphanumeric key.

This information is read only.

`IndexFile`

Path and file name of the auxiliary index file.

This information is read only. Use this property to confirm the name and location of the index file associated with the object.

`InputFile`

Path and file name of the source file.

This information is read only. Use this property to confirm the name and location of the source file from which the object was constructed.

`Interpreter`

Handle to a function used by the `read` method to parse entries in the source file.

This interpreter function must accept a character vector of one or more concatenated entries and return a structure or an array of structures containing the interpreted data. Set this property when your source file has a `'TABLE'`, `'MRTAB'`, or `'FLAT'` format. When your source file is an application-specific format such as `'SAM'`, `'FASTQ'`, or `'FASTA'`, then the default is a function handle appropriate for that file type and typically does not require you to change it.

`MemoryMappedIndex`

Whether the indices to the source file are stored in a memory-mapped file or in memory.

`NumEntries`

Number of entries indexed by the object.

This information is read only.

Methods

getDictionary	Retrieve reference sequence names from SAM-formatted source file associated with BioIndexedFile object
getEntryByIndex	Retrieve entries from source file associated with BioIndexedFile object using numeric index
getEntryByKey	Retrieve entries from source file associated with BioIndexedFile object using alphanumeric key
getIndexByKey	Retrieve indices from source file associated with BioIndexedFile object using alphanumeric key
getKeys	Retrieve alphanumeric keys from source file associated with BioIndexedFile object
getSubset	Create object containing subset of elements from BioIndexedFile object
read	Read one or more entries from source file associated with BioIndexedFile object

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects in the MATLAB Programming Fundamentals documentation.

Examples

Construct a BioIndexedFile object and access its gene ontology (GO) terms

This example shows how to construct a BioIndexedFile object and access its gene ontology (GO) terms.

Create a variable containing full absolute path of source file.

```
sourcefile = which('yeastgenes.sgd');
```

Copy the file to the current working directory.

```
copyfile(sourcefile,'yeastgenes_copy.sgd');
```

Construct a BioIndexedFile object from the source file that is a tab-delimited file, considering contiguous rows with the same key as a single entry. Indicate that keys are located in column 3 and that header lines are prefaced with '!'.

```
gene2goObj = BioIndexedFile('mrtab','yeastgenes_copy.sgd','KeyColumn',3,'HeaderPrefix','!');
```

```
Source File: yeastgenes_copy.sgd
   Path: C:\TEMP\Bdoc23a_2213998_3568\ib570499\25\tp1a9bae79\bioinfo-ex58973989
   Size: 21455392 bytes
   Date: 15-Mar-2018 17:45:16
Creating new index file ...
Indexer found 36266 entries after parsing 111912 text lines.
Index File: yeastgenes_copy.sgd.idx
   Path: C:\TEMP\Bdoc23a_2213998_3568\ib570499\25\tp1a9bae79\bioinfo-ex58973989
   Size: 494723 bytes
   Date: 03-Mar-2023 08:43:08
```

```
Mapping object to yeastgenes_copy.sgd.idx ...
Done.
```

Return the GO term from all entries that are associated with the gene YAT2. Access entries that have a key of YAT2.

```
YAT2_entries = getEntryByKey(gene2goObj,'YAT2');
```

Adjust object interpreter to return only the column containing the GO term.

```
gene2goObj.Interpreter = @(x) regexp(x,'GO:\d+','match');
```

Parse the entries with a key of YAT2 and return all GO terms from those entries.

```
GO_YAT2_entries = read(gene2goObj, 'YAT2')
```

```
GO_YAT2_entries = 1x14 cell
    {'GO:0004092'}    {'GO:0006066'}    {'GO:0006066'}    {'GO:0009437'}    {'GO:0005829'}    {'G
```

See Also

memmapfile | fastaread | fastqread | samread | genbankread

Topics

"Work with Next-Generation Sequencing Data"

bioma.data.ExptData class

Package: `bioma.data`

Contain data values from microarray experiment

Description

The ExptData class is designed to contain data values, such as gene expression values, from a microarray experiment. It stores the data values in one or more DataMatrix objects on page 1-734, each having the same row names (feature names) and column names (sample names). It provides a convenient way to store related experiment data in a single data structure (object). It also lets you manage and subset the data.

The ExptData class includes properties and methods that let you access, retrieve, and change data values from a microarray experiment. These properties and methods are useful to view and analyze the data.

Construction

`*EDobj* = bioma.data.ExptData(*Data1*, *Data2*, ...)` creates an ExptData object, from one or more matrices of data. Each matrix can be a logical matrix, a numeric matrix, or a DataMatrix object on page 1-734.

`*EDobj* = bioma.data.ExptData(..., {*DMobj1*, *Name1*}, {*DMobj2*, *Name2*}, ...)` specifies an element name for each DataMatrix object. *Name#* is a character vector or string specifying a unique name. Default names are `Elmt1`, `Elmt2`, etc.

`*EDobj* = bioma.data.ExptData({*Data1*, *Data2*, ...})` creates an ExptData object, from a cell array of matrices of data. Each matrix can be a logical matrix, a numeric matrix, or a DataMatrix object on page 1-734.

`*EDobj* = bioma.data.ExptData(..., '*PropertyName*', *PropertyValue*)` constructs the object using options, specified as property name/property value pairs.

`*EDobj* = bioma.data.ExptData(..., 'ElementNames', *ElementNamesValue*)` specifies element names for the matrix inputs. *ElementNamesValue* is a cell array of character vectors or string vector. Default names are `Elmt1`, `Elmt2`, etc.

`*EDobj* = bioma.data.ExptData(..., 'FeatureNames', *FeatureNamesValue*)` specifies feature names (row names) for the ExptData object. .

`*EDobj* = bioma.data.ExptData(..., 'SampleNames', *SampleNamesValue*)` specifies sample names (column names) for the ExptData object.

Input Arguments

`Data#`

Matrix of experimental data values specified by any of the following:

- Logical matrix
- Numeric matrix
- DataMatrix object on page 1-734

All inputs must have the same dimensions. All DataMatrix objects must also have the same row names and columns names. If you provide logical or numeric matrices, `bioma.data.ExptData` converts them to DataMatrix objects with either default row and column names, or the row and column names of DataMatrix inputs, if provided.

The rows must correspond to features and the columns must correspond to samples.

Default:

`DMobj#`

Variable name of a DataMatrix object in the MATLAB Workspace.

Default:

`Name#`

Character vector or string specifying an element name for the corresponding DataMatrix object

`ElementNamesValue`

Cell array of character vectors or string vector that specifies unique element names for the matrix inputs. The number of elements in *`ElementNamesValue`* must equal the number input matrices.

Default: {`Elmt1`, `Elmt2`, ...}

`FeatureNamesValue`

Feature names (row names) for the ExptData object, specified by one of the following:

- Cell array of character vectors
- Character array
- String vector
- Numeric or logical vector
- Character vector or string, which is used as a prefix for the feature names, with feature numbers appended to the prefix
- Logical `true` or `false` (default). If `true`, `bioma.data.ExptData` assigns unique feature names using the format `Feature1`, `Feature2`, etc.

If you use a cell array of character vectors, character array, string vector, numeric or logical vector, then the number of elements must be equal in number to the number of rows in *`Data1`*.

`SampleNamesValue`

Sample names (column names) for the ExptData object, specified by one of the following:

- Cell array of character vectors
- Character array

- String vector
- Numeric or logical vector
- Character vector or string, which is used as a prefix for the sample names, with sample numbers appended to the prefix
- Logical `true` or `false` (default). If `true`, `bioma.data.ExptData` assigns unique sample names using the format `Sample1`, `Sample2`, etc.

If you use a cell array of character vectors, character array, string vector, numeric or logical vector, then the number of elements must be equal in number to the number of columns in *`Data1`*. If the ExptData object is part of an ExpressionSet object that contains a MetaData object, the sample names (column names) in the ExptData object must match the sample names (row names) in a MetaData object.

Default:

Properties

`ElementClass`

Class type of the DataMatrix objects in the experiment

Cell array of character vectors specifying the class type of each DataMatrix object in the ExptData object. Possible values are MATLAB classes, such as `single`, `double`, and `logical`. This information is read-only.

Attributes:

`SetAccess`	`private`

`Name`

Name of the ExptData object.

Character vector specifying the name of the ExptData object. Default is `[]`.

`NElements`

Number of elements in the experiment

Positive integer specifying the number of elements (DataMatrix objects) in the experiment data. This value is equivalent to the number of DataMatrix objects in the ExptData object. This information is read-only.

Attributes:

`SetAccess`	`private`

`NFeatures`

Number of features in the experiment

Positive integer specifying the number of features in the experiment. This value is equivalent to the number of rows in each DataMatrix object in the ExptData object. This information is read-only.

Attributes:

`SetAccess`	`private`

`NSamples`

Number of samples in the experiment

Positive integer specifying the number of samples in the experiment. This value is equivalent to the number of columns in each DataMatrix object in the ExptData object. This information is read-only.

Attributes:

`SetAccess`	`private`

Methods

combine	Combine two ExptData objects
dmNames	Retrieve or set `Name` properties of DataMatrix objects in ExptData object
elementData	Retrieve or set data element (DataMatrix object) in ExptData object
elementNames	Retrieve or set element names of DataMatrix objects in ExptData object
featureNames	Retrieve or set feature names in ExptData object
isempty	Determine whether ExptData object is empty
sampleNames	Retrieve or set sample names in ExptData object
size	Return size of ExptData object

Instance Hierarchy

An ExpressionSet object contains an ExptData object. An ExptData object contains one or more DataMatrix objects.

Attributes

To learn about attributes of classes, see Class Attributes.

Copy Semantics

Value. To learn how this affects your use of the class, see Copying Objects.

Indexing

ExptData objects support 1-D parenthesis ( ) indexing to extract, assign, and delete data.

ExptData objects do not support:

- Dot . indexing
- Curly brace { } indexing

Examples

Construct an ExptData object

This example shows how to construct an ExptData object containing one DataMatrix object.

Import bioma.data package to make constructor functions available.

```
import bioma.data.*
```

Create a DataMatrix object from .txt file containing expression values from microarray experiment.

```
dmObj = DataMatrix('File', 'mouseExprsData.txt');
```

Construct an ExptData object from the DataMatrix object.

```
EDObj = ExptData(dmObj)
```

```
EDObj =
Experiment Data:
  500 features,  26 samples
  1 elements
  Element names: Elmt1
```

References

[1] Hovatta, I., Tennant, R S., Helton, R., et al. (2005). Glyoxalase 1 and glutathione reductase 1 regulate anxiety in mice. Nature *438*, 662–666.

See Also

bioma.data.MIAME | bioma.ExpressionSet | bioma.data.MetaData | getgeodata

Topics
“Working with Objects for Microarray Experiment Data”
“Analyzing Illumina Bead Summary Gene Expression Data”
Class Attributes
Property Attributes
“Representing Experiment Information in a MIAME Object”

bioma.data.MetaData class

Package: `bioma.data`

Contain metadata from microarray experiment

Description

The MetaData class is designed to contain metadata (variable values and descriptions) from a microarray experiment. It provides a convenient way to store related metadata in a single data structure (object). It also lets you manage and subset the data.

The metadata is a collection of variable names, for example related to samples or microarray features, along with descriptions and values for the variables. A MetaData object stores the metadata in two dataset arrays.

- **Values dataset array** — A dataset array containing the measured value of each variable per sample or feature. In this dataset array, the columns correspond to variables and rows correspond to either samples or features. The number and names of the columns in this dataset array must match the number and names of the rows in the Descriptions dataset array. If this dataset array contains *sample* metadata, then the number and names of the rows (samples) must match the number and names of the columns in the DataMatrix objects in the same ExpressionSet object. If this dataset array contains *feature* metadata, then the number and names of the rows (features) must match the number and names of the rows in the DataMatrix objects in the same ExpressionSet object.
- **Descriptions dataset array** — A dataset array containing a list of the variable names and their descriptions. In this dataset array, each row corresponds to a variable. The row names are the variable names, and a column, named `VariableDescription`, contains a description of the variable. The number and names of the rows in the Descriptions dataset array must match the number and names of the columns in the Values dataset array.

The MetaData class includes properties and methods that let you access, retrieve, and change metadata variables, and their values and descriptions. These properties and methods are useful to view and analyze the metadata.

Construction

`*MDobj* = bioma.data.MetaData(*VarValues*)` creates a MetaData object from one dataset array whose rows correspond to sample (observation) names and whose columns correspond to variables. The dataset array contains the measured value of each variable per sample.

`*MDobj* = bioma.data.MetaData(*VarValues*, *VarDescriptions*)` creates a MetaData object from two dataset arrays. `*VarDescriptions*` is a dataset array whose rows correspond to variables. The row names are the variable names, and another column, named `VariableDescription`, contains a description of each variable.

`*MDobj* = bioma.data.MetaData(*VarValues*, *VarDesc*)` creates a MetaData object from a dataset array and `*VarDesc*` a cell array of character vectors containing descriptions of the variables.

`*MDobj* = bioma.data.MetaData(..., '*PropertyName*', *PropertyValue*)` constructs the object using options, specified as property name/property value pairs.

MDobj = bioma.data.MetaData('File', *FileValue*) creates a MetaData object from a text file containing a table of metadata. The table row labels must be sample names, and its column headers must be variable names.

MDobj = bioma.data.MetaData('File', *FileValue*, ...'Path', *PathValue*) specifies a folder or path and folder where *FileValue* is stored.

MDobj = bioma.data.MetaData('File', *FileValue*, ...'Delimiter', *DelimiterValue*) specifies a delimiter symbol to use as a column separator for *FileValue*. Default is '\t'.

MDobj = bioma.data.MetaData('File', *FileValue*, ...'RowNames', *RowNamesValue*) specifies the row names (sample names) for the MetaData object. Default is the information in the first column of the table.

MDobj = bioma.data.MetaData('File', *FileValue*, ...'ColumnNames', *ColumnNamesValue*) specifies the columns of data to read from the table. *ColumnNamesValue* is a cell array of character vectors specifying the column header names. Default is to read all columns of data from the table, assuming the first row contains column headers.

MDobj = bioma.data.MetaData('File', *FileValue*, ...'VarDescChar', *VarDescCharValue*) specifies that lines in the table prefixed by *VarDescCharValue* to be read as descriptions and used to create the *VarDescriptions* dataset array. By default, bioma.data.MetaData does not read variable description information, and does not create a Descriptions dataset array. These prefixed lines must appear at the top of the file, before the table of metadata values.

MDobj = bioma.data.MetaData(...'Name', *NameValue*) specifies a name for the MetaData object.

MDobj = bioma.data.MetaData('File', *FileValue*, ...'Description', *DescriptionValue*) specifies a description for the MetaData object.

MDobj = bioma.data.MetaData('File', *FileValue*, ...'SampleNames', *SampleNamesValue*) specifies sample names (row names) for the MetaData object.

MDobj = bioma.data.MetaData('File', *FileValue*, ...'VariableNames', *VariableNamesValue*) specifies variable names (column names) for the MetaData object.

Input Arguments

VarValues

Dataset array whose rows correspond to sample (observation) names and whose columns correspond to variables. The dataset array contains the measured value of each variable per sample or feature.

The number and names of the columns in the *VarValues* dataset array must match the number and names of the rows in the *VarDescriptions* dataset array. If *VarValues* contains *sample* metadata, then the number and names of the rows (samples) must match the number and names of the columns in the DataMatrix objects in the same ExpressionSet object. If *VarValues* contains *feature* metadata, then the number and names of the rows (features) must match the number and names of the rows in the DataMatrix objects in the same ExpressionSet object.

`VarDescriptions`

Dataset array whose rows correspond to variables. The row names are the variable names, and a column, named `VariableDescription`, contains a description of the variable. The number and names of the rows in the *`VarDescriptions`* dataset array must match the number and names of the columns in the *`VarValues`* dataset array.

`VarDesc`

Cell array of character vectors containing descriptions of the variables. The number of elements in *`VarDesc`* must equal the number of columns (variable names) in *`VarValues`*.

`FileValue`

Character vector specifying a text file containing a table of metadata. The table row labels must be sample or feature names, and its column headers must be variable names. The text file must be on the MATLAB search path or in the Current Folder (unless you use the `Path` property).

Default:

`PathValue`

Character vector specifying a folder or path and folder where *`FileValue`* is stored.

`DelimiterValue`

Character vector specifying a delimiter symbol to use as a column separator for *`FileValue`*. Typical choices are:

- `' '`
- `'\t'` (default)
- `','`
- `';'`
- `'|'`

`RowNamesValue`

Row names (sample or feature names) for the MetaData object, specified by one of the following:

- Cell array of character vectors
- Single number indicating the column of the table containing the row names
- Character vector indicating the column header of the table containing the row names

If you specify `[]` for *`RowNamesValue`*, then `bioma.data.MetaData` provides numbered row names, starting with `1`.

Default: `1`, which specifies the information in the first column of the table

`ColumnNamesValue`

Cell array of character vectors specifying the column header names to indicate which columns of data to read from the table. Default is to read all columns of data from the table, assuming the first row contains column headers. If the table does not have column headers, specify `[]` for

ColumnNamesValue to read all columns of data and provide numbered column names, starting with `1`.

VarDescCharValue

Character vector specifying a character to prefix lines in the table that are to be read as descriptions and used to create the *VarDescriptions* dataset array. By default, `bioma.data.MetaData` does not read variable description information, and does not create a *VarDescriptions* dataset array. These prefixed lines must appear at the top of the file, before the table of metadata values.

NameValue

Character vector specifying a name for the MetaData object.

DescriptionValue

Character vector specifying a description for the MetaData object.

SampleNamesValue

Cell array of character vectors specifying sample names for the MetaData object. The number of elements in the cell array must equal the number of samples in the MetaData object. This input overwrites sample names from the input file. Default are the sample names (row names) from the input file.

VariableNamesValue

Cell array of character vectors specifying variable names for the MetaData object. The number of elements in the cell array must equal the number of variables in the MetaData object. This input overwrites variable names from the input file. Default are the variable names (column names) from the input file.

Properties

Description

Description of the MetaData object.

Character vector specifying a description of the MetaData object. Default is `[]`.

DimensionLabels

Row and column labels for the MetaData object.

Two-element cell array containing character vectors specifying labels of the rows and columns respectively in the MetaData object. Default is `{'Samples', 'Variables'}`.

Name

Name of the MetaData object.

Character vector specifying the name of the MetaData object. Default is `[]`.

NSamples

Number of samples (observations) in the experiment

Positive integer specifying the number of samples in the experiment. This value is equivalent to the number of rows in the *`VarValues`* dataset array. This information is read-only

Attributes:

`SetAccess`	`private`

`NVariables`

Number of variables in the experiment

Positive integer specifying the number of variables in the experiment. This value is equivalent to the number of columns in the *`VarValues`* dataset array. This information is read-only

Attributes:

`SetAccess`	`private`

Methods

combine	Combine two MetaData objects
isempty	Determine whether MetaData object is empty
sampleNames	Retrieve or set sample names in MetaData object
size	Return size of MetaData object
variableDesc	Retrieve or set variable descriptions for samples in MetaData object
variableNames	Retrieve or set variable names for samples in MetaData object
variableValues	Retrieve or set variable values for samples in MetaData object
varValuesTable	Create 2-D graphic table GUI of variable values in MetaData object

Instance Hierarchy

An ExpressionSet object contains two MetaData objects, one for sample information and one for microarray feature information. A MetaData object contains two dataset arrays. One dataset array contains the measured value of each variable per sample or feature. The other dataset array contains a list of the variable names and their descriptions.

Attributes

To learn about attributes of classes, see Class Attributes.

Copy Semantics

Value. To learn how this affects your use of the class, see Copying Objects.

Indexing

MetaData objects support 2-D parenthesis ( ) indexing and dot . indexing to extract, assign, and delete data.

MetaData objects do not support:

- Curly brace { } indexing
- Linear indexing

Examples

Construct a MetaData object containing sample variable information from a text file:

```
% Import bioma.data package to make constructor function
% available
import bioma.data.*
% Construct MetaData object from .txt file
MDObj2 = MetaData('File', 'mouseSampleData.txt', 'VarDescChar', '#');
% Display information about the MetaData object
MDObj2
% Supply a description for the MetaData object
MDObj2.Description = 'This MetaData Object contains sample variable info.'
```

See Also

`bioma.data.MIAME` | `bioma.ExpressionSet` | `bioma.data.ExptData` | `getgeodata`

Topics

"Working with Objects for Microarray Experiment Data"
"Analyzing Illumina Bead Summary Gene Expression Data"
Class Attributes
Property Attributes
"Representing Experiment Information in a MIAME Object"

bioma.data.MIAME class

Package: `bioma.data`

Contain experiment information from microarray gene expression experiment

Description

The MIAME class is designed to contain information about experimental methods and conditions from a microarray gene expression experiment. It loosely follows the Minimum Information About a Microarray Experiment (MIAME) specification. It can include information about:

- Experiment design
- Microarrays used in the experiment
- Samples used
- Sample preparation and labeling
- Hybridization procedures and parameters
- Normalization controls
- Preprocessing information
- Data processing specifications

It provides a convenient way to store related information about a microarray experiment in a single data structure (object).

The MIAME class includes properties and methods that let you access, retrieve, and change experiment information related to a microarray experiment. These properties and methods are useful to view and analyze the information.

Construction

`*MIAMEobj* = bioma.data.MIAME()` creates an empty MIAME object for storing experiment information from a microarray gene expression experiment.

`*MIAMEobj* = bioma.data.MIAME(*GeoSeriesStruct*)` creates a MIAME object from a structure containing Gene Expression Omnibus (GEO) Series data.

`*MIAMEobj* = bioma.data.MIAME(..., '*PropertyName*', *PropertyValue*)` constructs the object using options, specified as property name/property value pairs.

`*MIAMEobj* = bioma.data.MIAME(...,'Investigator', *InvestigatorValue*)` specifies the name of the experiment investigator.

`*MIAMEobj* = bioma.data.MIAME(...,'Lab', *LabValue*)` specifies the laboratory that conducted the experiment.

`*MIAMEobj* = bioma.data.MIAME(...,'Contact', *ContactValue*)` specifies the contact information for the experiment investigator or laboratory.

`*MIAMEobj* = bioma.data.MIAME(...,'URL', *URLValue*)` specifies the experiment URL.

Input Arguments

`GeoSeriesStruct`

Gene Expression Omnibus (GEO) Series data specified by either:

- MATLAB structure returned by the `getgeodata` function
- `Structure.Header.Series` substructure returned by the `getgeodata` function

Default:

`InvestigatorValue`

Character vector specifying the name of the experiment investigator.

Default:

`LabValue`

Character vector specifying the laboratory that conducted the experiment.

Default:

`ContactValue`

Character vector specifying the contact information for the experiment investigator or laboratory

Default:

`URLValue`

Character vector specifying the experiment URL.

Default:

Properties

`Abstract`

Abstract describing the experiment

Character vector containing an abstract describing the experiment.

`Arrays`

Information about the microarray chips used in the experiment

Cell array containing information about the microarray chips used in the experiment. Information can include array name, array platform, number of features on the array, and so on.

`Contact`

Contact information for the experiment investigator or laboratory

Character array containing contact information for the experiment investigator or laboratory.

ExptDesign

Brief description of the experiment design

Character array containing description of the experiment design.

Hybridization

Information about the experiment hybridization

Cell array containing information about the hybridization protocol used in the experiment. Information can include hybridization time, concentration, volume, temperature, and so on.

Investigator

Name of the experiment investigator

Character array containing the name of the experiment investigator.

Laboratory

Name of the laboratory where the experiment was conducted

Character array containing the name of laboratory.

Other

Other information about the experiment

Cell array containing other information about the experiment, not covered by the other properties.

Preprocessing

Information about the experiment preprocessing steps

Cell array containing information about the preprocessing steps used on the data from the experiment.

PubMedID

PubMed identifiers for relevant publications.

Character array containing PubMed identifiers for papers relevant to the data set used in the experiment.

QualityControl

Information about the experiment quality controls

Cell array containing information about the experiment quality control steps. Information can include replicates, dye swap, and so on.

Samples

Information about samples used in the experiment

Cell array containing information about the samples used in the experiment. Information can include sample source, sample organism, treatment type, compound, labeling protocol, external control, and so on.

`Title`

Experiment title

Character array containing a single sentence experiment title.

`URL`

URL for the experiment

Character array containing URL for the experiment.

Methods

combine	Combine two MIAME objects
isempty	Determine whether MIAME object is empty

Instance Hierarchy

An ExpressionSet object contains a MIAME object.

Attributes

To learn about attributes of classes, see Class Attributes.

Copy Semantics

Value. To learn how this affects your use of the class, see Copying Objects.

Examples

Construct a MIAME object

Create a MATLAB structure containing Gene Expression Omnibus (GEO) series data.

```
geoStruct = getgeodata('GSE4616');
```

Import bioma.data package to make the constructor function available.

```
import bioma.data.*
```

Construct MIAME object from the structure.

```
MIAMEObj1 = MIAME(geoStruct)
```

```
MIAMEObj1 =
Experiment Description:
  Author name: Mika,,Silvennoinen
Riikka,,KivelÃ¤
Maarit,,Lehti
Anna-Maria,,Touvras
Jyrki,,Komulainen
Veikko,,Vihko
Heikki,,Kainulainen
  Laboratory: LIKES - Research Center
  Contact information: Mika,,Silvennoinen
  URL: 
  PubMedIDs: 17003243
  Abstract: A 90 word abstract is available. Use the Abstract property.
  Experiment Design: A 234 word summary is available. Use the ExptDesign property.
  Other notes: 
    [1x84 char]
```

Supply a URL for the MIAME object.

```
MIAMEObj1.URL = 'www.nonexistinglab.com'
```

```
MIAMEObj1 = 

Experiment Description:
  Author name: Mika,,Silvennoinen
Riikka,,KivelÃ¤
Maarit,,Lehti
Anna-Maria,,Touvras
Jyrki,,Komulainen
Veikko,,Vihko
Heikki,,Kainulainen
  Laboratory: LIKES - Research Center
  Contact information: Mika,,Silvennoinen
  URL: www.nonexistinglab.com
  PubMedIDs: 17003243
  Abstract: A 90 word abstract is available. Use the Abstract property.
  Experiment Design: A 234 word summary is available. Use the ExptDesign property.
  Other notes: 
    [1x84 char]
```

Alternatively you can construct a MIAME object using customized properties.

```
MIAMEObj2 = MIAME('investigator', 'Jane Researcher',...
                  'lab', 'One Bioinformatics Laboratory',...
                  'contact', 'jresearcher@lab.not.exist',...
                  'url', 'www.lab.not.exist',...
                  'title', 'Normal vs. Diseased Experiment',...
                  'abstract', 'Example of using expression data',...
                  'other', {'Notes:Created from a text file.'})
```

```
MIAMEObj2 = 

Experiment Description:
  Author name: Jane Researcher
  Laboratory: One Bioinformatics Laboratory
  Contact information: jresearcher@lab.not.exist
  URL: www.lab.not.exist
  PubMedIDs: 
  Abstract: A 4 word abstract is available. Use the Abstract property.
  No experiment design summary available.
  Other notes:
    'Notes:Created from a text file.'
```

See Also

`bioma.ExpressionSet` | `bioma.data.ExptData` | `bioma.data.MetaData` | `getgeodata`

Topics

“Working with Objects for Microarray Experiment Data”
“Analyzing Illumina Bead Summary Gene Expression Data”
Class Attributes
Property Attributes
“Representing Experiment Information in a MIAME Object”

bioma.ExpressionSet class

Package: `bioma`

Contain data from microarray gene expression experiment

Description

The ExpressionSet class is designed to contain data from a microarray gene expression experiment, including expression values, sample and feature metadata, and information about experimental methods and conditions. It provides a convenient way to store related information about a microarray gene expression experiment in a single data structure (object). It also lets you manage and subset the data.

The ExpressionSet class includes properties and methods that let you access, retrieve, and change data, metadata, and other information about the microarray gene expression experiment. These properties and methods are useful for viewing and analyzing the data.

Construction

`*ExprSetobj* = bioma.ExpressionSet(*Data*)` creates an ExpressionSet object, from `*Data*`, a numeric matrix, a DataMatrix object on page 1-734, or an ExptData on page 1-386 object, which contains one or more DataMatrix objects with the same dimensions, row names and column names.

`*ExprSetobj* = bioma.ExpressionSet(*Data*, {*DMobj1*, *Name1*}, {*DMobj2*, *Name2*}, ...)` creates an ExpressionSet object, from `*Data*`, and additional DataMatrix objects with specified element names. All DataMatrix objects must have the same dimensions, row names, and column names.

`*ExprSetobj* = bioma.ExpressionSet(..., '*PropertyName*', *PropertyValue*)` constructs the object using options, specified as property name/property value pairs.

`*ExprSetobj* = bioma.ExpressionSet(..., 'SData', *SDataValue*)` includes a MetaData on page 1-391 object containing sample metadata in the ExpressionSet object.

`*ExprSetobj* = bioma.ExpressionSet(..., 'FData', *FDataValue*)` includes a MetaData on page 1-391 object containing microarray feature metadata in the ExpressionSet object.

`*ExprSetobj* = bioma.ExpressionSet(..., 'EInfo', *EInfoValue*)` includes a MIAME on page 1-397 object, which contains experiment information, in the ExpressionSet object.

Input Arguments

`Data`

Any of the following:

- Numeric matrix
- DataMatrix object on page 1-734
- ExptData on page 1-386 object, which contains one or more DataMatrix objects having the same dimensions

If you provide a DataMatrix object, `bioma.ExpressionSet` creates an ExptData object from it and names the DataMatrix object `Expressions`. If you provide an ExptData object, `bioma.ExpressionSet` renames the first DataMatrix object in the ExptData object to `Expressions`, unless another DataMatrix object in the ExptData object is already named `Expressions`.

`DMobj#`

Variable name of a DataMatrix object. Each DataMatrix object must have the same dimensions as *`Data`*.

`Name#`

Character vector or string specifying an element name for the corresponding DataMatrix object. Each DataMatrix object in an ExpressionSet object has an element name. At least one DataMatrix object in an ExpressionSet object has an element name of `Expressions`. By default, it is the first DataMatrix object.

`SDataValue`

Variable name of a MetaData on page 1-391 object containing sample metadata for the experiment. The variable name must exist in the MATLAB Workspace.

`FDataValue`

Variable name of a MetaData on page 1-391 object containing microarray feature metadata for the experiment. The variable name must exist in the MATLAB Workspace.

`EInfoValue`

Variable name of a MIAME on page 1-397 object, which contains information about the experiment methods and conditions. The variable name must exist in the MATLAB Workspace.

Properties

`NElements`

Number of elements in the experiment

Positive integer specifying the number of elements (DataMatrix objects) in the experiment data. This value is equivalent to the number of DataMatrix objects in the ExperimentSet object. This information is read-only.

Attributes:

`SetAccess`	`private`

`NFeatures`

Number of features in the experiment

Positive integer specifying the number of features in the experiment. This value is equivalent to the number of rows in each DataMatrix object in the ExperimentSet object. This information is read-only.

Attributes:

`SetAccess`	`private`

`NSamples`

Number of samples in the experiment

Positive integer specifying the number of samples in the experiment. This value is equivalent to the number of columns in each DataMatrix object in the ExperimentSet object. This information is read-only.

Attributes:

`SetAccess`	`private`

Methods

abstract	Retrieve or set abstract describing experiment in ExpressionSet object
elementData	Retrieve or set data element (DataMatrix object) in ExpressionSet object
elementNames	Retrieve or set element names of DataMatrix objects in ExpressionSet object
expressions	Retrieve or set `Expressions` DataMatrix object from ExpressionSet object
exprWrite	Write expression values in ExpressionSet object to text file
exptData	Retrieve or set experiment data in ExpressionSet object
exptInfo	Retrieve or set experiment information in ExpressionSet object
featureData	Retrieve or set feature metadata in ExpressionSet object
featureNames	Retrieve or set feature names in ExpressionSet object
featureVarDesc	Retrieve or set feature variable descriptions in ExpressionSet object
featureVarNames	Retrieve or set feature variable names in ExpressionSet object
featureVarValues	Retrieve or set feature variable data values in ExpressionSet object
pubMedID	Retrieve or set PubMed IDs in ExpressionSet object
sampleData	Retrieve or set sample metadata in ExpressionSet object
sampleNames	Retrieve or set sample names in ExpressionSet object
sampleVarDesc	Retrieve or set sample variable descriptions in ExpressionSet object
sampleVarNames	Retrieve or set sample variable names in ExpressionSet object
sampleVarValues	Retrieve or set sample variable values in ExpressionSet object
size	Return size of ExpressionSet object

Instance Hierarchy

An ExpressionSet object contains an ExptData object, two MetaData objects, and a MIAME object. These objects can be empty.

Attributes

To learn about attributes of classes, see Class Attributes.

Copy Semantics

Value. To learn how this affects your use of the class, see Copying Objects.

Indexing

ExpressionSet objects support 2-D parenthesis ( ) indexing to extract, assign, and delete data.

ExpressionSet objects do not support:

- Dot . indexing
- Curly brace { } indexing
- Linear indexing

Examples

Construct an ExpressionSet Object

This example shows how to construct an ExpressionSet object. The `mouseExprsData.txt` file used in this example contains data from Hovatta et al., 2005.

Import bioma.data package to make the constructor function available.

```
import bioma.data.*
```

Create a DataMatrix object from .txt file containing expression values from microarray experiment.

```
dmObj = DataMatrix('File', 'mouseExprsData.txt');
```

Construct an ExptData object.

```
EDObj = ExptData(dmObj)
```

```
EDObj = 

Experiment Data:
  500 features,  26 samples
  1 elements
  Element names: Elmt1
```

Construct a MetaData object from .txt file.

```
MDObj2 = MetaData('File', 'mouseSampleData.txt', 'VarDescChar', '#')
```

```
MDObj2 = 

Sample Names:
    A, B, ...,Z (26 total)
Variable Names and Meta Information:
              VariableDescription
    Gender    ' Gender of the mouse in study'
    Age       ' The number of weeks since mouse birth'
    Type      ' Genetic characters'
    Strain    ' The mouse strain'
    Source    ' The tissue source for RNA collection'
```

Create a MATLAB structure containing GEO Series data.

```
geoStruct = getgeodata('GSE4616');
```

Construct a MIAME object.

```
MIAMEObj = MIAME(geoStruct)
```

```
MIAMEObj =

Experiment Description:
  Author name: Mika,,Silvennoinen
Riikka,,KivelÃ¤
Maarit,,Lehti
Anna-Maria,,Touvras
Jyrki,,Komulainen
Veikko,,Vihko
Heikki,,Kainulainen
  Laboratory: LIKES - Research Center
  Contact information: Mika,,Silvennoinen
  URL:
  PubMedIDs: 17003243
  Abstract: A 90 word abstract is available. Use the Abstract property.
  Experiment Design: A 234 word summary is available. Use the ExptDesign property.
  Other notes:
    [1x84 char]
```

Import bioma package to make constructor function available.

```
import bioma.*
```

Construct an ExpressionSet object.

```
ESObj = ExpressionSet(EDObj, 'SData', MDObj2, 'EInfo', MIAMEObj)
```

```
ESObj =

ExpressionSet
Experiment Data: 500 features, 26 samples
  Element names: Expressions
Sample Data:
    Sample names:     A, B, ...,Z (26 total)
    Sample variable names and meta information:
        Gender:  Gender of the mouse in study
        Age:  The number of weeks since mouse birth
        Type:  Genetic characters
        Strain:  The mouse strain
        Source:  The tissue source for RNA collection
Feature Data: none
Experiment Information: use 'exptInfo(obj)'
```

References

[1] Hovatta, I., Tennant, R S., Helton, R., et al. (2005). Glyoxalase 1 and glutathione reductase 1 regulate anxiety in mice. Nature *438*, 662–666.

See Also

`bioma.data.MIAME` | `bioma.data.ExptData` | `bioma.data.MetaData` | `getgeodata`

Topics

"Working with Objects for Microarray Experiment Data"
"Analyzing Illumina Bead Summary Gene Expression Data"
Class Attributes
Property Attributes
"Representing Experiment Information in a MIAME Object"

BioRead

Contain sequence reads and their quality data

Description

The `BioRead` object contains sequencing read data, including sequence headers, nucleotide sequences, and quality scores.

Create a BioRead object from NGS (next-generation sequencing) data stored in an FASTQ- or SAM-formatted file. Each element in the object has a sequence, header, and quality score associated with it. Use the object properties and functions to explore, access, filter, and manipulate all the data or a subset of the data. If you have data with reads that are already mapped to a reference sequence, and you need to access alignment records, use `BioMap` instead.

Creation

Syntax

```
bioreadObj = BioRead
bioreadObj = BioRead(File)
bioreadObj = BioRead(S)
bioreadObj = BioRead(Seqs)
bioreadObj = BioRead(Seqs,Quals)
bioreadObj = BioRead(Seqs,Quals,Headers)
bioreadObj = BioRead( ___ ,Name,Value)
```

Description

`bioreadObj = BioRead` creates an empty `BioRead` object `bioreadObj`.

`bioreadObj = BioRead(File)` creates a `BioRead` object from `File`, an FASTQ- or SAM-formatted file. The data remains in the source file after the object is created, and you have access to data through the object properties but cannot modify the properties, except the `Name` property.

`bioreadObj = BioRead(S)` creates a `BioRead` object from `S`, a MATLAB structure, containing the fields `Header`, `Sequence`, and `Quality`. The data from `S` remains in memory, and you can modify the properties of the object.

`bioreadObj = BioRead(Seqs)` creates a `BioRead` object from `Seqs`, a cell array of character vectors or string vector containing nucleotide sequences.

`bioreadObj = BioRead(Seqs,Quals)` creates a `BioRead` object from `Seqs` and sets the `Quality` property of the object to `Quals`, a cell array of character vectors or string vector containing the ASCII representation of per-base quality scores for each read.

`bioreadObj = BioRead(Seqs,Quals,Headers)` also sets the `Header` property of the object to `Headers`, a cell array of character vectors or string vector containing the header text for each read.

`bioreadObj = BioRead( ___ ,Name,Value)` specifies options using one or more name-value pair arguments in addition to input arguments in previous syntaxes. For instance, `br = BioRead('SRR005164_1_50.fastq','InMemory',true)` specifies to load the data in memory instead of leaving it in the source file.

Input Arguments

`File` — Name of FASTQ- or SAM-formatted file

character vector | string

Name of FASTQ- or SAM-formatted file, specified as a character vector or string.

The `BioRead` object accesses data using an auxiliary index file. The index file must have the same name as the source file, but with an .idx extension. If the index file is not in the same folder as the source file, the `BioRead` function creates the index file in that folder.

Note Because the data remains in the source file, do not delete the source file and auxiliary index file.

Example: `'ex1.sam'`

Data Types: `char`

`S` — Sequence information

structure

Sequence information, specified as a structure. `S` must contain the fields `Header`, `Sequence`, and `Quality`. For instance, the `fastqread` and `samread` functions return such a structure.

Example: `S`

Data Types: `struct`

`Seqs` — Nucleotide sequences

cell array of character vectors | string vector

Nucleotide sequences, specified as a cell array of character vectors or string vector.

Data Types: `cell`

`Quals` — Sequence quality information

cell array of character vectors | string vector

Sequence quality information, specified as a cell array of character vectors.

Data Types: `cell`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `br = BioRead('SRR005164_1_50.fastq','InMemory',true)` specifies to load the data in memory instead of leaving it in the source file.

`InMemory` — Boolean indicator to keep data in memory
`false` (default) | `true`

Boolean indicator to keep data in memory, specified as the comma-separated pair consisting of `'InMemory'` and `true` or `false`.

When you create a `BioRead` object from a file, the object does not load the data in memory, but leaves it in the source file and accesses it using an index file to make the process more memory efficient. You cannot modify the object properties if you do not load the data in memory.

If the first input is not a file, this name-value pair argument is ignored, and the data is automatically placed in memory.

Example: `'InMemory',true`

Data Types: `logical`

`IndexDir` — Path to index file folder
character vector | string

Path to the index file folder where the index file exists or is created, specified as the comma-separated pair consisting of `'IndexDir'` and a character vector or string.

Example: `'IndexDir','C:\data\'`

Data Types: `char`

Properties

`Header` — Header information of reads
cell array of character vectors

Header information of reads, specified as a cell array of character vectors. Each character vector represents the header text for each read. There is a one-to-one relationship between the number and order of character vectors (elements) in the `Header` and `Sequence` properties, unless `Header` is an empty cell array.

Data Types: `cell`

`Name` — Object name
character vector | string

Object name, specified as a character vector or string.

Example: `'seqdata'`

Data Types: `char`

`NSeqs` — Number of reads
positive integer

Number of reads in the object, specified as a positive integer.

Example: `20000`

Data Types: `double`

`Quality` — Per-base quality scores for all reads
cell array of character vectors

Per-base quality scores for all reads, specified as a cell array of character vectors. Each element is an ASCII representation of per-base quality scores for each read. A one-to-one relationship exists between the number and order of elements in `Quality` and `Sequence`, unless `Quality` is an empty cell array.

Example: `{'<<:<<<','<<<7<:'}`

Data Types: `cell`

Sequence — Nucleotide sequences
cell array of character vectors

Nucleotide sequences (reads), specified as a cell array of character vectors.

Example: `{'TATCTG','ATCTAC'}`

Data Types: `cell`

Object Functions

combine	Combine two objects
get	Retrieve property of object
getHeader	Retrieve sequence headers from object
getQuality	Retrieve sequence quality information from object
getSequence	Retrieve sequences from object
getSubsequence	Retrieve partial sequences from object
getSubset	Retrieve subset of elements from object
set	Set property of object
setHeader	Update header information of reads
setQuality	Update quality information
setSequence	Update read sequences
setSubsequence	Update partial sequences
setSubset	Update elements of object
write	Write contents of BioRead or BioMap object to file

Examples

Create BioRead Object from NGS Data

Create a BioRead object from sequencing read data saved in a FASTQ-formatted file.

```
br = BioRead('SRR005164_1_50.fastq')
```

```
br = 
  BioRead with properties:

     Quality: [50x1 File indexed property]
    Sequence: [50x1 File indexed property]
      Header: [50x1 File indexed property]
       NSeqs: 50
        Name: ''
```

By default, when creating a BioRead object from a file, the function also creates an index file if one does not already exist. This example uses an existing index file created and saved in:

```
fullfile(matlabroot,'toolbox','bioinfo','bioinfodata','SRR005164_1_50.fastq.idx')
```

The data remains in the source file, and the object accesses the data using the index file, making the process more memory efficient. But you cannot edit the object properties, except the `Name` property.

To edit the properties, set `'InMemory'` to `true` .

```
brEdit = BioRead('SRR005164_1_50.fastq','InMemory',true);
brEdit.Header(1) = {'SR1'};
```

```
brEdit.Header(1)
```

```
ans = 1x1 cell array
    {'SR1'}
```

If you create the object from a MATLAB structure or cell array of nucleotide sequences, the sequence data is always saved in memory by default, and the `InMemory` option is ignored.

For instance, generate MATLAB variables containing synthetic sequences and quality scores.

```
seqs = {randseq(10);randseq(15);randseq(20)};
quals = {repmat('!',1,10); repmat('%',1,15);repmat('&',1,20)};
headers = {'H1';'H2';'H3'};
```

Create a structure using these variables.

```
structData = struct('Header',headers,'Sequence',seqs,'Quality',quals);
```

Create a BioRead object from the structure.

```
brStruct = BioRead(structData);
```

You can edit the properties of the object because the data remains in memory.

```
brStruct.Header(1) = {'H1.1'};
brStruct.Header(1)
```

```
ans = 1x1 cell array
    {'H1.1'}
```

Version History

Introduced in R2010a

See Also

`BioIndexedFile` | `BioMap` | `fastqinfo` | `fastqread` | `saminfo` | `samread` | `seqqcplot`

Topics

"Manage Sequence Read Data in Objects"

External Websites

Sequence Read Archive
SAM format specification

blastncbi

Create remote NCBI BLAST report request ID or link to NCBI BLAST report

Syntax

```
blastncbi(Seq,Program)
RID = blastncbi(Seq,Program)
[RID,RTOE] = blastncbi(Seq,Program)
___ = blastncbi(___,Name,Value)
```

Description

`blastncbi(Seq,Program)` sends a BLAST request to NCBI against `Seq`, a nucleotide or amino acid sequence, using `Program`, a specified BLAST program. Then it returns a link to the NCBI BLAST report. For help in selecting an appropriate BLAST program, visit https://blast.ncbi.nlm.nih.gov/producttable.shtml.

`RID = blastncbi(Seq,Program)` returns `RID`, the Request ID for the report.

`[RID,RTOE] = blastncbi(Seq,Program)` returns both `RID`, the Request ID for the NCBI BLAST report, and `RTOE`, the Request Time Of Execution, which is an estimated time needed for the search to finish.

`___ = blastncbi(___,Name,Value)` uses additional options specified by one or more name-value pair arguments, and any of the arguments in the previous syntaxes.

Examples

Perform BLAST search

Perform a BLAST search on a protein sequence and save the results to an XML file.

Get a sequence from the Protein Data Bank and create a MATLAB structure.

```
S = getpdb('1CIV');
```

Use the structure as input for the BLAST search with a significance threshold of `1e-10`. The first output is the request ID, and the second output is the estimated time (in minutes) until the search is completed.

```
[RID1,ROTE] = blastncbi(S,'blastp','expect',1e-10);
```

Get the search results from the report. You can save the XML-formatted report to a file for an offline access. Use ROTE as the wait time to retrieve the results.

```
report1 = getblast(RID1,'WaitTime',ROTE,'ToFile','1CIV_report.xml')
```

```
Blast results are not available yet. Please wait ...

report1 =
  struct with fields:

                RID: 'R49TJMCF014'
          Algorithm: 'BLASTP 2.6.1+'
           Database: 'nr'
            QueryID: 'Query_224139'
    QueryDefinition: 'unnamed protein product'
               Hits: [1×100 struct]
         Parameters: [1×1 struct]
         Statistics: [1×1 struct]
```

Use `blastread` to read BLAST data from the XML-formatted BLAST report file.

```
blastdata = blastread('1CIV_report.xml')
```

```
blastdata = 

  struct with fields:

                RID: ''
          Algorithm: 'BLASTP 2.6.1+'
           Database: 'nr'
            QueryID: 'Query_224139'
    QueryDefinition: 'unnamed protein product'
               Hits: [1×100 struct]
         Parameters: [1×1 struct]
         Statistics: [1×1 struct]
```

Alternatively, run the BLAST search with an NCBI accession number.

```
RID2 = blastncbi('AAA59174','blastp','expect',1e-10)
```

```
RID2 = 

    'R49WAPMH014'
```

Get the search results from the report.

```
report2 = getblast(RID2)
```

```
Blast results are not available yet. Please wait ...

report2 = 

  struct with fields:

                RID: 'R49WAPMH014'
          Algorithm: 'BLASTP 2.6.1+'
           Database: 'nr'
            QueryID: 'AAA59174.1'
    QueryDefinition: 'insulin receptor precursor [Homo sapiens]'
               Hits: [1×100 struct]
         Parameters: [1×1 struct]
```

```
        Statistics: [1×1 struct]
```

Input Arguments

Seq — Nucleotide or amino acid sequence
character vector | string | MATLAB structure

Nucleotide or amino acid sequence, specified as a character vector, string, or MATLAB structure containing a `Sequence` field.

If `Seq` is a character vector or string, the available options are:

- GenBank, GenPept, or RefSeq accession number
- Name of a FASTA file
- URL pointing to a sequence file

`Program` — BLAST program
character vector | string

BLAST program, specified as one of the following:

- `'blastn'` — Search nucleotide query versus nucleotide database.
- `'blastp'` — Search protein query versus protein database.
- `'blastx'` — Search (translated) nucleotide query versus protein database.
- `'megablast'` — Search for highly similar nucleotide sequences.
- `'tblastn'` — Search protein query versus translated nucleotide database.
- `'tblastx'` — Search (translated) nucleotide query versus (translated) nucleotide database.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `'Matrix','PAM70','Expect',1e-10` uses the `PAM70` substitution matrix with the significance threshold for matches set to 1e-10.

`Database` — Database to search
`'nr'` (default) | character vector | string

Database to search, specified as the comma-separated pair consisting of `'Database'` and a character vector or string.

For nucleotide databases, valid choices are:

- `'nr'` (default)
- `'refseq_rna'`
- `'refseq_genomic'`

- `'est'`
- `'est_human'`
- `'est_mouse'`
- `'est_others'`
- `'gss'`
- `'htgs'`
- `'pat'`
- `'pdb'`
- `'alu'`
- `'dbsts'`
- `'chromosome'`

For protein databases, valid choices are:

- `'nr'` (default)
- `'refseq_protein'`
- `'swissprot'`
- `'pat'`
- `'pdb'`
- `'env_nr'`

Note Available databases may change. Check the NCBI website for more information.

For help in selecting an appropriate database, visit

`https://blast.ncbi.nlm.nih.gov/producttable.shtml`

.

`MaxNumberSequences` — Maximum number of hits to return
100 (default) | positive integer

Maximum number of hits to return, specified as the comma-separated pair consisting of `'MaxNumberSequences'` and a positive integer. The actual search results may have fewer hits than what you specify, depending on the query, database, expectation value, and other parameters. The default value is `100`.

`Filter` — Filter applied to query sequence
character vector | string

Filter applied to the query sequence, specified as the comma-separated pair consisting of `'Filter'` and one of the following:

- `'L'` — Mask regions of low compositional complexity.
- `'R'` — Mask human repeat elements (valid for `blastn` and `megablast` only).
- `'m'` — Mask the query while producing blast seeds, but not during extension.

- `'none'` — No mask is applied.
- `'l'` — Mask any letter that is lowercase in the query.

You can specify multiple valid letters in a single character vector or string to apply multiple filters at once. For example, `'Lm'` applies both the low compositional complexity filter and the mask.

Choices vary depending on the selected `Program`. For more information, see the table Choices for Optional Properties by BLAST Program.

`Expect` — Statistical significance threshold for matches
`10` (default) | positive real number

Statistical significance threshold for matches against database sequences, specified as the comma-separated pair consisting of `'Expect'` and a positive real number. The default is `10`.

You can learn more about the statistics of local sequence comparison at https://blast.ncbi.nlm.nih.gov/tutorial/Altschul-1.html#head2.

`Word` — Word length for query sequence
positive integer

Word length for the query sequence, specified as the comma-separated pair consisting of `'Word'` and a positive integer.

Choices for a protein query search are:

- `2`
- `3` (default)

Choices for a nucleotide query search are:

- `7`
- `11` (default)
- `15`

Choices when `Program` is set to `'megablast'` are:

- `16`
- `20`
- `24`
- `28` (default)
- `32`
- `48`
- `64`
- 128

`Matrix` — Substitution matrix for amino acid sequences
`'BLOSUM62'` (default) | character vector | string

Substitution matrix for amino acid sequences, specified as the comma-separated pair consisting of `'Matrix'` and a character vector or string. The matrix assigns the score for a possible alignment of any two amino acid residues. Choices are:

- `'PAM30'`
- `'PAM70'`
- `'BLOSUM45'`
- `'BLOSUM62'` (default)
- `'BLOSUM80'`

`MatchScores` — Matching and mismatching scores in nucleotide alignment
two-element numeric vector

Matching and mismatching scores in a nucleotide alignment, specified as the comma-separated pair consisting of `'MatchScores'` and a two-element numeric vector `[R Q]`. The first element `R` is the match score, and the second element `Q` is the mismatch score. This option is for `blastn` and `megablast` only.

To ensure accurate evaluation of the alignment significance, only a limited set of combinations are supported. See the table “BLAST Optional Properties” on page 1-420 for all the supported values. The default value for `megablast` is `[1 -2]`, and the default value for `blastn` is `[1 -3]`.

`GapCosts` — Penalties for opening and extending gap
two-element numeric vector

Penalties for opening and extending a gap, specified as the comma-separated pair consisting of `'GapCosts'` and a two-element numeric vector. The vector contains two integers: the first is the penalty for opening a gap, and the second is the penalty for extending a gap.

Valid gap costs for `blastp`, `blastx`, `tblastn`, and `tblastx` vary according to the protein substitution matrix. For details, see GapCosts for blastp, blastx, tblastn, and tblastx.

Valid gap costs for `blastn` and `megablast` vary according to `MatchScores` (`[R Q]`). For details, see GapCosts for blastn and megablast.

`CompositionAdjustment` — Compositional adjustment type to compensate for amino acid compositions
`'none'` (default) | `'cbs'` | `'ccsm'` | `'ucsm'`

Compositional adjustment type to compensate for the amino acid compositions of the sequences being compared, specified as the comma-separated pair consisting of `'CompositionAdjustment'` and one of the following values:

- `'none'`— No adjustment is applied (default).
- `'cbs'`— Composition-based statistics approach is used for score adjustments.
- `'ccsm'`— Conditional compositional score matrix is used for score adjustments.
- `'ucsm'`— Universal compositional score matrix is used for score adjustments.

This option is for `blastp`, `blastx`, and `tblastn` only. The resulting scaled scores yield more accurate E-values than the standard, unscaled scores. For details, see Compositional adjustments.

`Entrez` — Entrez query syntax to search a subset of selected database
character vector | string

Entrez query syntax to search a subset of the selected database, specified as the comma-separated pair consisting of `'Entrez'` and a character vector or string. Use this option to limit searches based on molecule types, sequence lengths, organisms, and so on. For more information on limiting searches, see https://blast.ncbi.nlm.nih.gov/blastcgihelp.shtml#entrez_query.

`Adv` — Advanced options
character vector | string

Advanced options, specified as the comma-separated pair consisting of `'Adv'` and a character vector or string. For instance, to specify the reward and penalty values for nucleotide matches and mismatches, use `'-r 1 -q -3'`. For more information, see https://www.ncbi.nlm.nih.gov/blast/Doc/urlapi.html.

`TimeOut` — Connection timeout
5 (default) | positive scalar

Connection timeout (in seconds) to submit the BLAST query, specified as a positive scalar. For details, see here.

Data Types: `double`

Output Arguments

`RID` — Request ID for NCBI BLAST report
character vector

Request ID for the NCBI BLAST report, returned as a character vector.

`RTOE` — Request time of execution
integer

Request time of execution, returned as an integer. This is an estimated time in minutes until the search is completed.

Tip If you use the `getblast` function to retrieve the BLAST report, use this time estimate as the `'WaitTime'` option.

More About

BLAST Optional Properties

Choices for Optional Properties by BLAST Program

When BLAST program is...	Then choices for the following options are...					
	Database	Filter	Word	Matrix	MatchScores [R Q]	GapCosts
'blastn'	'nr' (default) 'refseq_rna' 'refseq_genomic' 'est' 'est_human' 'est_mouse' 'est_others' 'gss'	'L' 'm' (default) 'R' 'm' 'l' 'none'	7 11 (default) 15	—	[1 -3] (default) [1 -4] [1 -2] [1 -1] [2 -3] [4 -5]	See GapCosts for blastn and megablast.

When BLAST program is...	Then choices for the following options are...					
	Database	Filter	Word	Matrix	MatchScores [R Q]	GapCosts
'megablast'	'htgs' 'pat' 'pdb' 'alu' 'dbsts' 'chromosome'		16 20 24 28 (default) 32 48 64 128		[1 -3] [1 -4] [1 -2] (default) [1 -1] [2 -3] [4 -5]	
'tblastn'		'L' '(default)' 'm' 'l'	2 3 (default)	'PAM30' 'PAM70' 'BLOSUM45' 'BLOSUM62'	-	See GapCosts for blastp,

When BLAST program is...	'tblastx'	'blastp'	'blastx'
Then choices for the following options are...			
Database		'nr' (default) 'refseq_protein' 'swissprot' 'pat' 'pdb' 'env_nr'	
Filter	'none'	'L' 'm' 'l' 'none' (default)	'L' (default) 'm' 'l' 'none'
Word			
Matrix	(default) 'BLOSUM80'		
MatchScores [R Q]			
GapCosts	blastx, tblastn, and tblastx.		

GapCosts for `blastp`, `blastx`, `tblastn`, and `tblastx`

Substitution Matrix	Valid `'GapCosts'` Values

`'PAM30'`	`[7 2]`
	`[6 2]`
	`[5 2]`
	`[10 1]`
	`[9 1]` (default)
	`[8 1]`

`'PAM70'`
`'BLOSUM80'`
	`[8 2]`
	`[7 2]`
	`[6 2]`
	`[11 1]`
	`[10 1]` (default)
	`[9 1]`

`'BLOSUM45'`	`[13 3]`
	`[12 3]`
	`[11 3]`
	`[10 3]`
	`[15 2]` (default)
	`[14 2]`
	`[13 2]`
	`[12 2]`
	`[19 1]`
	`[18 1]`
	`[17 1]`
	`[16 1]`

`'BLOSUM62'`	`[9 2]`
	`[8 2]`
	`[7 2]`
	`[12 1]`
	`[11 1]` (default)
	`[10 1]`

GapCosts for `blastn` and `megablast`

MatchScores [R Q]	Valid `'GapCosts'` Values

`[1 -4]`	`[5 2]` (default)
	`[1 2]`
	`[0 2]`
	`[2 1]`
	`[1 1]`

`[1 -3]`	`[5 2]`(default)
	`[2 2]`
	`[1 2]`
	`[0 2]`
	`[2 1]`
	`[1 1]`

`[1 -2]`	`[5 2]`(default)
	`[2 2]`
	`[1 2]`
	`[0 2]`
	`[3 1]`
	`[2 1]`
	`[1 1]`

`[1 -1]`	`[5 2]`(default)
	`[3 2]`
	`[2 2]`
	`[1 2]`
	`[0 2]`
	`[4 1]`
	`[3 1]`
	`[2 1]`

`[2 -3]`	`[5 2]`(default)
	`[4 4]`
	`[2 4]`
	`[0 4]`
	`[3 3]`
	`[6 2]`
	`[4 2]`
	`[2 2]`

`[4 -5]`	`[5 2]`(default)
	`[6 5]`
	`[5 5]`
	`[4 5]`
	`[3 5]`

Version History

Introduced before R2006a

R2017b: `'psiblast'` BLAST program has been removed

Errors starting in R2017b

The BLAST program `'psiblast'` has been removed from one of supported programs.

R2017b: `'Inclusion'` option has been removed
Errors starting in R2017b

The `'Inclusion'` name-value pair has been removed since it only applies to the `psiblast` program which has been also removed.

R2017b: `'Descriptions'` option has been removed
Errors starting in R2017b

The `'Descriptions'` name-value pair has been removed. Use `'MaxNumberSequences'` instead to specify the maximum number of hits to return.

R2017b: `'Alignments'` option has been removed
Errors starting in R2017b

The `'Alignments'` name-value pair has been removed. Use `'MaxNumberSequences'` instead to specify the maximum number of hits to return.

R2017b: `'GapOpen'` option has been removed
Errors starting in R2017b

The `'GapOpen'` name-value pair has been removed. Use `'GapCosts'` instead.

R2017b: `'ExtendGap'` option has been removed
Errors starting in R2017b

The `'ExtendGap'` name-value pair has been removed. Use `'GapCosts'` instead.

R2017b: `'Pct'` option has been removed
Errors starting in R2017b

The `'Pct'` name-value pair has been removed.

References

[1] Altschul, S.F., W. Gish, W. Miller, E.W. Myers, and D.J. Lipman (1990). "Basic local alignment search tool." J. Mol. Biol. *215*, 403–410.

[2] Altschul, S.F., T.L. Madden, A.A. Schäffer, J. Zhang, Z. Zhang, W. Miller, and D. J. Lipman (1997). "Gapped BLAST and PSI-BLAST: a new generation of protein database search programs." Nucleic Acids Res. *25*, 3389–3402.

See Also
`blastread` | `getblast`

External Websites
BLAST Help
Entrez Help

blastread

Read data from NCBI BLAST report file

Syntax

```
blastdata = blastread(blastreport)
```

Description

`blastdata = blastread(blastreport)` reads the NCBI BLAST report data from an XML-formatted file, `blastreport`, and returns `blastdata`, a structure containing the corresponding BLAST data.

Examples

Perform BLAST search

Perform a BLAST search on a protein sequence and save the results to an XML file.

Get a sequence from the Protein Data Bank and create a MATLAB structure.

```
S = getpdb('1CIV');
```

Use the structure as input for the BLAST search with a significance threshold of `1e-10`. The first output is the request ID, and the second output is the estimated time (in minutes) until the search is completed.

```
[RID1,ROTE] = blastncbi(S,'blastp','expect',1e-10);
```

Get the search results from the report. You can save the XML-formatted report to a file for an offline access. Use ROTE as the wait time to retrieve the results.

```
report1 = getblast(RID1,'WaitTime',ROTE,'ToFile','1CIV_report.xml')
```

```
Blast results are not available yet. Please wait ...

report1 = 

  struct with fields:

                RID: 'R49TJMCF014'
          Algorithm: 'BLASTP 2.6.1+'
           Database: 'nr'
            QueryID: 'Query_224139'
    QueryDefinition: 'unnamed protein product'
               Hits: [1×100 struct]
         Parameters: [1×1 struct]
         Statistics: [1×1 struct]
```

Use `blastread` to read BLAST data from the XML-formatted BLAST report file.

```
blastdata = blastread('1CIV_report.xml')
```

```
blastdata = 

  struct with fields:

                  RID: ''
            Algorithm: 'BLASTP 2.6.1+'
             Database: 'nr'
              QueryID: 'Query_224139'
      QueryDefinition: 'unnamed protein product'
                 Hits: [1×100 struct]
           Parameters: [1×1 struct]
           Statistics: [1×1 struct]
```

Alternatively, run the BLAST search with an NCBI accession number.

```
RID2 = blastncbi('AAA59174','blastp','expect',1e-10)
```

```
RID2 =

    'R49WAPMH014'
```

Get the search results from the report.

```
report2 = getblast(RID2)
```

```
Blast results are not available yet. Please wait ...

report2 = 

  struct with fields:

                  RID: 'R49WAPMH014'
            Algorithm: 'BLASTP 2.6.1+'
             Database: 'nr'
              QueryID: 'AAA59174.1'
      QueryDefinition: 'insulin receptor precursor [Homo sapiens]'
                 Hits: [1×100 struct]
           Parameters: [1×1 struct]
           Statistics: [1×1 struct]
```

Input Arguments

blastreport — Name of BLAST report file

character vector | string

Name of an XML-formatted BLAST report file, specified as a character vector or string.

Example: `'blastreport.xml'`

Output Arguments

`blastdata` — BLAST report data
structure

BLAST report data, returned as a structure that contains the following fields:

Field	Description
`RID`	Request ID for retrieving results from a specific NCBI BLAST search
`Algorithm`	NCBI algorithm used to perform the BLAST search
`Database`	All databases searched
`QueryID`	Identifier of the query sequence
`QueryDefinition`	Definition of the query sequence
`Hits`	Structure containing information on the hit sequences, such as IDs, accession numbers, lengths, and HSPs (high-scoring segment pairs)
`Parameters`	Structure containing information on the input parameters used to perform the search
`Statistics`	Summary of statistical details about the performed search, such as lambda, kappa, and entropy values

More About

Hits

This table lists each field of `blastdata.Hits`.

Field	Description
`ID`	ID of the subject sequence that matched the query sequence
`Definition`	Description of the subject sequence
`Accession`	Accession of the subject sequence
`Length`	Length of the subject sequence
`Hsps`	Structure containing Information on the high-scoring segment pairs (HSPs)

Hits.Hsps

This table summarizes the fields of `Hits.Hsps`.

Field	Description
`Score`	Pairwise alignment score for a high-scoring segment pair between the query sequence and a subject sequence.
`BitScore`	Bit score for a high-scoring segment pair.

Field	Description
`Expect`	Expectation value for a high-scoring segment pair.
`Identities`	Number of identical or similar residues for a high-scoring segment pair between the query sequence and a subject sequence.
`Positives`	Number of identical or similar residues for a high-scoring sequence pair between the query sequence and a subject amino acid sequence. This field applies only to translated nucleotide or amino acid query sequences and databases.
`Gaps`	Nonaligned residues for a high-scoring segment pair.
`AlignmentLength`	Length of the alignment for a high-scoring segment pair.
`QueryIndices`	Indices of the query sequence residue positions for a high-scoring segment pair.
`SubjectIndices`	Indices of the subject sequence residue positions for a high-scoring segment pair.
`Frame`	Reading frame of the translated nucleotide sequence for a high-scoring segment pair.
`Alignment`	3-by-*N* character array showing the alignment for a high-scoring sequence pair between the query sequence and a subject sequence. The first row is the query sequence, the second row is the alignment, and the third row is the subject sequence.

Version History

Introduced before R2006a

See Also

`blastncbi` | `getblast`

blosum

Return BLOSUM scoring matrix

Syntax

```
Matrix = blosum(Identity)
[Matrix, MatrixInfo] = blosum(Identity)

... = blosum(Identity, ...'Extended', ExtendedValue, ...)
... = blosum(Identity, ...'Order', OrderValue, ...)
```

Input Arguments

`Identity`	Scalar specifying a percent identity level. Choices are: • Values from `30` to `90` in increments of `5` • `62` • `100`
`ExtendedValue`	Controls the listing of extended amino acid codes. Choices are `true` (default) or `false`.
`OrderValue`	Character vector or string containing legal amino acid characters that specifies the order amino acids are listed in the matrix. The length of the character vector or string must be `20` or `24`.

Output Arguments

`Matrix`	BLOSUM (Blocks Substitution Matrix) scoring matrix with a specified percent identity.
`MatrixInfo`	Structure of information about *`Matrix`* containing the following fields: • `Name` • `Scale` • `Entropy` • `ExpectedScore` • `HighestScore` • `LowestScore` • `Order`

Description

`Matrix` `= blosum(`*`Identity`*`)` returns a BLOSUM (Blocks Substitution Matrix) scoring matrix with a specified percent identity. The default ordering of the output includes the extended characters `B`, `Z`, `X`, and `*`.

```
A R N D C Q E G H I L K M F P S T W Y V B Z X *
```

`[`*`Matrix`*`, `*`MatrixInfo`*`] = blosum(`*`Identity`*`)` returns *`MatrixInfo`*, a structure of information about *`Matrix`*, a BLOSUM matrix. *`MatrixInfo`* contains the following fields:

- `Name`
- `Scale`
- `Entropy`
- `ExpectedScore`
- `HighestScore`
- `LowestScore`
- `Order`

`... = blosum(`*`Identity`*`, ...'`*`PropertyName`*`', `*`PropertyValue`*`, ...)` calls `blosum` with optional properties that use property name/property value pairs. You can specify one or more properties in any order. Each *`PropertyName`* must be enclosed in single quotation marks and is case insensitive. These property name/property value pairs are as follows:

`... = blosum(`*`Identity`*`, ...'Extended', `*`ExtendedValue`*`, ...)` controls the listing of extended amino acid codes. Choices are `true` (default) or `false`. If *`ExtendedValue`* is `false`, returns the scoring matrix for the standard `20` amino acids. Ordering of the output when *`ExtendedValue`* is `false` is

```
A R N D C Q E G H I L K M F P S T W Y V
```

`... = blosum(`*`Identity`*`, ...'Order', `*`OrderValue`*`, ...)` returns a BLOSUM matrix ordered by *`OrderValue`*, a character vector or string containing legal amino acid characters that specifies the order amino acids are listed in the matrix. The length of the character vector or string must be `20` or `24`.

Examples

Retrieve BLOSUM scoring matrix

Return a BLOSUM matrix with a percent identity level of 50.

```
B50 = blosum(50)
```

```
B50 = 24×24

     5    -2    -1    -2    -1    -1    -1     0    -2    -1    -2    -1    -1    -3    -1     1
    -2     7    -1    -2    -4     1     0    -3     0    -4    -3     3    -2    -3    -3    -1
    -1    -1     7     2    -2     0     0     0     1    -3    -4     0    -2    -4    -2     1
    -2    -2     2     8    -4     0     2    -1    -1    -4    -4    -1    -4    -5    -1     0
    -1    -4    -2    -4    13    -3    -3    -3    -3    -2    -2    -3    -2    -2    -4    -1
    -1     1     0     0    -3     7     2    -2     1    -3    -2     2     0    -4    -1     0
    -1     0     0     2    -3     2     6    -3     0    -4    -3     1    -2    -3    -1    -1
     0    -3     0    -1    -3    -2    -3     8    -2    -4    -4    -2    -3    -4    -2     0
    -2     0     1    -1    -3     1     0    -2    10    -4    -3     0    -1    -1    -2    -1
    -1    -4    -3    -4    -2    -3    -4    -4    -4     5     2    -3     2     0    -3    -3
      ⋮
```

Return a BLOSUM matrix with the amino acids in a specific order.

```
B75 = blosum(75,'Order','CSTPAGNDEQHRKMILVFYW')
```

```
B75 = 20×20

     9    -1    -1    -4    -1    -3    -3    -4    -5    -3    -4    -4    -4    -2    -1    -2
    -1     5     1    -1     1    -1     0    -1     0     0    -1    -1     0    -2    -3    -3
    -1     1     5    -1     0    -2     0    -1    -1    -1    -2    -1    -1    -1    -1    -2
    -4    -1    -1     8    -1    -3    -3    -2    -1    -2    -2    -2    -1    -3    -3    -3
    -1     1     0    -1     4     0    -2    -2    -1    -1    -2    -2    -1    -1    -2    -2
    -3    -1    -2    -3     0     6    -1    -2    -3    -2    -2    -3    -2    -3    -5    -4
    -3     0     0    -3    -2    -1     6     1    -1     0     0    -1     0    -3    -4    -4
    -4    -1    -1    -2    -2    -2     1     6     1    -1    -1    -2    -1    -4    -4    -4
    -5     0    -1    -1    -1    -3    -1     1     5     2     0     0     1    -2    -4    -4
    -3     0    -1    -2    -1    -2     0    -1     2     6     1     1     1     0    -3    -3
      ⋮
```

Version History

Introduced before R2006a

See Also

dayhoff | gonnet | localalign | nuc44 | nwalign | pam | swalign

bowtie

(Removed) Map short reads to reference sequence using Burrows-Wheeler transform

Note `bowtie` has been removed. Use `bowtie2` instead.

Syntax

```
bowtie(indexBaseName,reads,outputFileName)
bowtie(indexBaseName,reads,outputFileName,Name,Value)
```

Description

`bowtie(indexBaseName,reads,outputFileName)` aligns the reads specified in `reads` to the indexed reference specified by `indexBaseName`, and writes the results to the BAM-formatted file `outputFileName`.

Note `bowtie` runs on Mac and UNIX® platforms only.

`bowtie(indexBaseName,reads,outputFileName,Name,Value)` aligns reads using additional options specified by one or more name-value pair arguments.

Examples

Align Short Reads

Download the *E. coli* genome from NCBI.

```
getgenbank('NC_008253','tofile','NC_008253.fna','SequenceOnly',true)
```

Built a Bowtie index with the base name `ECOLI`.

```
bowtiebuild('NC_008253.fna','ECOLI')
```

Find the path to the example FASTQ file `ecoli100.fq`, which has *E. Coli* short reads.

```
fastqfile = which('ecoli100.fq')
```

Align the short reads in `ecoli100.fq` to the built index with base name `ECOLI`.

```
bowtie('ECOLI',fastqfile,'ecoli100.bam')
```

Access the mapped reads using `BioMap`.

```
bm = BioMap('ecoli100.bam')
```

```
bm = 

BioMap with properties:
    SequenceDictionary: {'gi|110640213|ref|NC_008253.1|'}
             Reference: [73x1 File indexed property]
             Signature: [73x1 File indexed property]
                 Start: [73x1 File indexed property]
        MappingQuality: [73x1 File indexed property]
                  Flag: [73x1 File indexed property]
          MatePosition: [73x1 File indexed property]
               Quality: [73x1 File indexed property]
              Sequence: [73x1 File indexed property]
                Header: [73x1 File indexed property]
                 NSeqs: 73
                  Name: ''
```

Input Arguments

`indexBaseName` — Name of indexed reference file
character vector | string

Name of indexed reference file for short read alignment, specified as a character vector or string containing the path and base name of the Bowtie index file.

`reads` — Short reads to align
character vector | string | string vector | cell array of character vectors

Short reads to align to the indexed reference, specified as a character vector, string, string vector, or cell array of character vectors indicating one or more FASTQ formatted files with the input reads.

`outputFileName` — Name for output file
character vector | string

Name for output file containing the results of the short read alignment, specified as a character vector or string. By default, the output file is BAM-formatted, and `bowtie` automatically adds the `.bam` extension if it is missing from the file name.

To specify a SAM-formatted output file, use the name-value pair argument `BamFileOutput,false`. In this case, `bowtie` automatically adds the `.sam` extension if it is missing from the file name.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `'BamFileOutput',false,'Paired',true` specifies the output file is SAM-formatted, and `bowtie` performs pair-read alignment.

`BamFileOutput` — Indicator for output file format
`true` (default) | `false`

Indicator for the output file format, specified as the comma-separated pair consisting of `'BamFileOutput'` and either `true` or `false`.

- If `true` (the default), then the output file is BAM-formatted, with a `.bam` extension.
- If `false`, then the output file is SAM-formatted, with a `.sam` extension.

`bowtie` automatically adds the corresponding file extension if it is missing from the input argument `outputFileName`.

Example: `'BamFileOutput',false`

Data Types: `logical`

`Paired` — Indicator for paired-read alignment performance
`false` (default) | `true`

Indicator for paired-read alignment performance, specified as the comma-separated pair consisting of `'Paired'` and either `true` or `false` (the default). If `false`, then `bowtie` performs paired-read alignment using the odd elements in `reads` as the upstream mates and the even elements in `reads` as the downstream mates.

Example: `'Paired',true`

Data Types: `logical`

`BowtieOptions` — Additional `bowtie` options
character vector | string

Additional `bowtie` options, specified as a character vector or string for any valid `bowtie` options. Type `bowtie('--help')` for available options.

Example: `'BowtieOptions','-k 5 -m 4'`

Tips

- More information on the Bowtie algorithm (Version 0.12.7) can be found at http://bowtie-bio.sourceforge.net/index.shtml.
- Some prebuilt index files for model organisms can be downloaded directly from the Bowtie repository.

Version History

Introduced in R2012b

R2023a: Removed
Errors starting in R2023a

`bowtie` has been removed. Use `bowtie2` instead.

R2022b: Warns
Warns starting in R2022b

`bowtie` issues a warning that it will be removed in a future release.

R2022a: To be removed
Not recommended starting in R2022a

`bowtie` runs without warning, but it will be removed in a future release.

See Also

`bowtie2` | `baminfo` | `BioMap` | `bowtiebuild` | `fastainfo` | `fastqinfo` | `samread` | `saminfo`

bowtie2

Map sequence reads to reference sequence

Syntax

```
bowtie2(indexBaseName,reads1,reads2,outputFileName)
bowtie2( ___ ,alignOptions)
flag = bowtie2( ___ )
```

Description

`bowtie2(indexBaseName,reads1,reads2,outputFileName)` maps the sequencing reads from `reads1` and `reads2` against the reference sequence and writes the results to the output file `outputFileName`. The input `indexBaseName` represents the base name (prefix) of the reference index files.

`bowtie2` requires the Bowtie 2 Support Package for Bioinformatics Toolbox. If this support package is not installed, then the function provides a download link. For details, see “Bioinformatics Toolbox Software Support Packages”.

`bowtie2( ___ ,alignOptions)` uses the additional options specified by `alignOptions`. Specify these options after all other input arguments.

`flag = bowtie2( ___ )` returns an exit `flag` of the function using any of the input arguments in the previous syntaxes.

Examples

Align Reads to Reference Sequence Using Bowtie 2

Build a set of index files for the Drosophila genome. An error message appears if you do not have the Bowtie 2 Support Package for Bioinformatics Toolbox installed when you run the function. Click the provided link to download the package from the Add-on menu.

For this example, the reference sequence `Dmel_chr4.fa` is already provided with the toolbox.

```
status = bowtie2build('Dmel_chr4.fa', 'Dmel_chr4_index');
```

If the index build is successful, the function returns `0` and creates the index files (`*.bt2`) in the current folder. The files have the prefix `'Dmel_chr4_index'`.

Sometimes the index files exist, and you want to know the reference sequence used to build the index. In this case, use the `bowtie2inspect` function to get more information about the reference.

```
bowtie2inspect('Dmel_chr4', 'Dmel_chr4_retrieved.fa');
```

By default, the output file `Dmel_chr4_retrieved.fa` contains the sequence of the reference. You can also get a summary information about the reference name and lengths instead of the actual sequence. For details on the available options, see `Bowtie2InspectOptions`.

Once the index is ready, map the read sequences to the reference using the `bowtie2` function. The paired-end read files (`SRR6008575_10k_1.fq` and `SRR6008575_10k_2.fq`) are already provided with the toolbox.

```
bowtie2('Dmel_chr4','SRR6008575_10k_1.fq','SRR6008575_10k_2.fq','SRR6008575_10k_chr4.sam');
```

The output is a SAM-formatted file that contains the mapping results.

You can specify different alignment options by passing in a Bowtie 2 syntax string or using a `Bowtie2AlignOptions` object.

Suppose you want to trim some residues from the `3'` end before aligning. First, create a `Bowtie2AlignOptions` object.

```
alignOpt = Bowtie2AlignOptions;
```

Trim four residues from the `3'` end before aligning.

```
alignOpt.Trim3 = 4;
```

Map reads to the reference using the specified alignment option.

```
flag = bowtie2('Dmel_chr4','SRR6008575_10k_1.fq','SRR6008575_10k_2.fq','SRR6008575_10k_chr4_trimm
```

Input Arguments

`indexBaseName` — Base name of reference index files
character vector | string

Base name (prefix) of the reference index files, specified as a character vector or string. The index files are in the `BT2` or `BT21` format.

Example: `'Dmel_chr4'`

Data Types: `char` | `string`

`reads1` — Names of files with first mate reads
string array | cell array of character vectors

Names of files with the first mate reads or single-end reads, specified as a string array or cell array of character vectors.

For paired-end data, sequences in `reads1` must correspond file-for-file and read-for-read to sequences in `reads2`.

Example: `'SRR6008575_10k_1.fq'`

Data Types: `char` | `string`

`reads2` — Names of files with second mate reads
string array | cell array of character vectors

Names of files with the second mate reads, specified as a string array or cell array of character vectors.

Specify `reads2` as an empty character vector or string (`''` or `""`) if the data consists of single-end reads only.

Example: `'SRR6008575_10k_2.fq'`

Data Types: `char` | `string`

`outputFileName` — Output file name
character vector | string

Output file name, specified as a character vector or string. This file contains the mapping results.

Example: `'SRR6008575_10k_chr4.sam'`

Data Types: `char` | `string`

`alignOptions` — Alignment options
character vector | string | `Bowtie2AlignOptions` object

Alignment options, specified as a character vector, string, or `Bowtie2AlignOptions` object. The character vector or string must be in the Bowtie 2 option syntax (prefixed by one or two dashes) [1].

For a `Bowtie2AlignOptions` object, only the modified properties are used to run the function.

Example: `'--trim5 10 -s 5'`

Output Arguments

`flag` — Exit status
integer

Exit status of the function, returned as an integer. `flag` is `0` if the function runs without errors or warning. Otherwise, it is nonzero.

Version History

Introduced in R2018a

References

[1] Langmead, Ben, and Steven L Salzberg. “Fast Gapped-Read Alignment with Bowtie 2.” *Nature Methods* 9, no. 4 (April 2012): 357–59. https://doi.org/10.1038/nmeth.1923.

See Also

`bowtie2inspect` | `bowtie2build` | `Bowtie2AlignOptions` | `Bowtie2BuildOptions` | `Bowtie2InspectOptions` | `featurecount` | `cufflinks`

Topics
“Count Features from NGS Reads”
“Bioinformatics Toolbox Software Support Packages”

External Websites
Bowtie 2 manual

Bowtie2AlignOptions

Options to map reads to reference sequence

Description

A `Bowtie2AlignOptions` object contains options to run the `bowtie2` function, which aligns reads to a reference sequence.

Creation

Syntax

```
alignOptions = Bowtie2AlignOptions
alignOptions = Bowtie2AlignOptions(Name,Value)
alignOptions = Bowtie2AlignOptions(S)
```

Description

`alignOptions = Bowtie2AlignOptions` creates a `Bowtie2AlignOptions` object with default property values.

`Bowtie2AlignOptions` requires the Bowtie 2 Support Package for Bioinformatics Toolbox. If this support package is not installed, then the function provides a download link. For details, see "Bioinformatics Toolbox Software Support Packages".

`alignOptions = Bowtie2AlignOptions(Name,Value)` sets properties on page 1-439 using one or more name-value pair arguments. Enclose each property name in quotes. For example, `alignOptions = Bowtie2AlignOptions('Trim5',10)` specifies to trim 10 residues from the 5' end.

`alignOptions = Bowtie2AlignOptions(S)` specifies optional parameters in a character vector `S`.

Input Arguments

S — Alignment parameters
character vector

Alignment parameters, specified as a character vector. `S` must be in the Bowtie 2 option syntax (prefixed by one or two dashes) [1].

Properties

`AllowDovetail` — Flag to allow dovetail configurations
`false` (default) | `true`

Flag to allow dovetail configurations, specified as `true` or `false`. This property specifies whether the alignment of one mate can extend past the beginning of the alignment of the other mate and be considered concordant.

This property applies to paired-end reads only.

Example: `'AllowDovetail',true`

Data Types: `logical`

`AmbiguousPenalty` — Penalty for positions with ambiguous characters
`1` (default) | nonnegative integer

Penalty for positions with ambiguous characters on the read sequence, reference sequence, or both, specified as a nonnegative integer.

Example: `'AmbiguousPenalty',2`

Data Types: `double`

`Encoding` — Encoding format of base quality
`'Phred33'` (default) | `'Phred64'` | `'Solexa'`

Encoding format of the base quality in the input files, specified as one of the following: `'Phred33'`, `'Phred64'`, or `'Solexa'`.

Example: `'Encoding','Phred64'`

Data Types: `char` | `string`

`ExcludeContain` — Flag to allow one mate alignment to contain other mate
`false` (default) | `true`

Flag to allow one mate alignment to contain the alignment of the other mate and to be considered concordant, specified as `true` or `false`.

This property applies to paired-end reads only.

Example: `'ExcludeContain',true`

Data Types: `logical`

`ExcludeDiscordant` — Flag to include discordant alignments
`false` (default) | `true`

Flag to include discordant alignments, specified as `true` or `false`. A discordant alignment is an alignment where both mates align uniquely, but not in a way that satisfies the paired-end constraints.

Example: `'ExcludeDiscordant',true`

Data Types: `logical`

`ExcludeMixed` — Flag to exclude mixed alignments
`false` (default) | `true`

Flag to exclude mixed alignments, specified as `true` or `false`. A mixed alignment consists of mate reads that are not concordant or discordant, but align individually.

This property applies to paired-end reads only.

Example: `'ExcludeMixed',true`

Data Types: `logical`

`ExcludeOverlap` — Flag to allow mate alignment overlap
`false` (default) | `true`

Flag to allow the alignment of one mate to overlap with the alignment of the other mate and to be considered concordant, specified as `true` or `false`.

Example: `'ExcludeOverlap',true`

Data Types: `logical`

`ExcludeUnaligned` — Flag to exclude reads that failed to align
`false` (default) | `true`

Flag to exclude reads that failed to align, specified as `true` or `false`.

Example: `'ExcludeUnaligned',true`

Data Types: `logical`

`ExtraBowtie2Command` — Additional options not included in object properties
`''` (default) | character vector

Additional options not included in the object properties, specified as a character vector. The character vector must be in the Bowtie 2 option syntax (prefixed by one or two dashes). The default value is an empty character vector `''`.

Example: `'ExtraBowtie2Command','--version'`

Data Types: `char` | `string`

`IgnoreQuality` — Flag to ignore read position quality
`false` (default) | `true`

Flag to ignore the actual read position quality when a mismatch occurs, specified as `true` or `false`. Setting this property to `true` allows the quality value at that mismatched position to be the highest possible, regardless of the actual value.

Example: `'IgnoreQuality',true`

Data Types: `logical`

`MatchBonus` — Reward added to alignment score
2 (default) | nonnegative integer

Reward added to the alignment score when a position in the read matches a position in the reference, specified as a nonnegative integer.

Example: `'MatchBonus',5`

Data Types: `double`

`MaxAmbiguousFunction` — Function governing maximum number of ambiguous characters
`'L,0,0.15'` (default) | character vector | string

Function governing the maximum number of ambiguous characters allowed in a read, specified as a character vector or string.

The function has the format `'f,B,A'`, where *f* is a function type, *B* is a constant term, and *A* is a coefficient. Available function types are:

- `'C'`– Constant
- `'L'`– Linear
- `'S'`– Square root
- `'G'`– Natural log

The resulting function is `H(x) = B + A * f(x)`, where *x* is the read length.

The default function is `'L,0,0.15'`, that is, `H(x) = 0 + 0.15 * x`.

Example: `'MaxAmbiguousFunction','L,-0.4,-0.6'`

Data Types: `char` | `string`

`MemoryMappedIndex` — Flag to use memory mapping when loading index
`false` (default) | `true`

Flag to use memory mapping (instead of file I/O) when loading the index, specified as `true` or `false`. Memory mapping allows many concurrent processes to share the memory image of the index, resulting in a more efficient parallelization of the task.

Example: `'MemoryMappedIndex',true`

Data Types: `logical`

`MinScoreFunction` — Function governing minimum score threshold of alignment
character vector | string

Function governing the minimum score threshold of an alignment, specified as a character vector or string.

The function has the format `'f,B,A'`, where *f* is a function type, *B* is a constant term, and *A* is a coefficient. Available function types are:

- `'C'`– Constant
- `'L'`– Linear
- `'S'`– Square root
- `'G'`– Natural log

The resulting function is `H(x) = B + A * f(x)`, where *x* is the read length.

For the `'EndToEnd'` alignment mode, the default function is `'L,-0.6,-0.6'`. For the `'Local'` mode, the default function is `'G,20,8'`.

Example: `'MinScoreFunction','L,-0.4,-0.6'`

Data Types: `char` | `string`

`MismatchPenalty` — Maximum and minimum values to compute mismatch penalty
`[6 2]` (default) | two-element vector

Maximum and minimum values to compute the mismatch penalty during alignment, specified as a two-element vector. The first element is the maximum value and the second element is the minimum value.

A number less than or equal to the maximum value, and greater than or equal to the minimum value is subtracted from the alignment score for each position where a read character aligns to a reference character, the characters do not match, and neither is an `N` character.

Example: `'MismatchPenalty',[5 3]`

Data Types: `double`

`Mode` — Alignment mode
`'EndToEnd'` (default) | `'Local'`

Alignment mode, specified as `'EndToEnd'` or `'Local'`.

In the `'Local'` mode, only part of the read must align to the reference, and some residues can be omitted (soft-clipped) to achieve the best alignment score. In the `'EndToEnd'` mode, the entire read must align without any soft-clipping.

Example: `'Mode','Local'`

Data Types: `char` | `string`

`Nondeterministic` — Flag to reinitialize pseudo-random generator
`false` (default) | `true`

Flag to reinitialize the pseudo-random generator for each read using the current time, specified as `true` or `false`. If `true`, the alignments reported for two identical reads can be different. The default value is `false`, that is, the pseudo-random generator is reinitialized using a seed derived from read information and the seed number.

Example: `'Nondeterministic',true`

Data Types: `logical`

`NoGapPositions` — Number of positions where gaps are not allowed
`4` (default) | nonnegative integer

Number of positions at the beginning or end of each read where gaps are not allowed, specified as a nonnegative integer.

Example: `'NoGapPositions',5`

Data Types: `double`

`NumAlignments` — Maximum number of valid alignments to report
`'Best'` (default) | `'All'` | positive integer

Maximum number of valid alignments to report before terminating the search, specified as a positive integer, `'Best'`, or `'All'`. If you specify a positive integer *N*, the function searches for up to *N* distinct, valid alignments for each read. `'Best'` reports the best alignment for each read. `'All'` reports all the valid alignments for each read sorted by alignment scores.

The alignment score for a paired-end alignment equals the sum of the alignment scores of individual mates.

Example: `'NumAlignments','All'`

Data Types: `double` | `char` | `string`

`NumReseedings` — Maximum number of reseeding attempts
`2` (default) | nonnegative integer

Maximum number of reseeding attempts with repetitive seeds, specified as a nonnegative integer. During reseeding, the function chooses a new set of reads at different offsets to find more alignments.

Example: `'NumReseedings',5`

Data Types: `double`

`NumSeedExtensions` — Maximum number of consecutive seed extension attempts
`15` (default) | nonnegative integer

Maximum number of consecutive seed extension attempts before getting a new seed, specified as a nonnegative integer. A seed extension fails if it does not yield an alignment with the best (or second-best) score.

Example: `'NumSeedExtensions',10`

Data Types: `double`

`NumSeedMismatches` — Number of allowed mismatches in seed alignment
`0` (default) | `1`

Number of allowed mismatches in a seed alignment during the multiseed alignment, specified as `0` or `1`.

Example: `'NumSeedMismatches',1`

Data Types: `double`

`NumThreads` — Number of parallel threads to perform alignment
`1` (default) | positive integer

Number of parallel threads to perform the alignment, specified as a positive integer. Threads run on separate processors or cores. Increasing the number of threads provides a significant increase in speed (close to linear) but also increases the memory footprint.

Example: `'NumThreads',4`

Data Types: `double`

`Offrate` — Offrate to use when reading index
`NaN` (default) | positive integer

Offrate to use when reading the index to reduce the memory footprint, specified as a positive integer. The offrate must be greater than the offrate used to build the index.

Example: `'Offrate',20`

Data Types: `double`

`PadPositions` — Position in reference sequence where alignment begins
`15` (default) | nonnegative integer

Position in the reference sequence where the alignment for each sequence begins, specified as a nonnegative integer.

Example: `'PadPositions',10`

Data Types: `double`

`ReadGapCosts` — Gap costs for opening and extending gap
`[5 3]` (default) | two-element vector of nonnegative integers

Gap costs for opening and extending a gap on the read, specified as a two-element vector of nonnegative integers. The first element is the cost of opening a gap, and the second element is the cost of extending a gap. Given the cost vector `[GO GE]`, a read gap of length *N* is assigned a penalty of `GO + N * GE`.

Example: `'ReadGapCosts',[4 2]`

Data Types: `double`

`ReadGroupID` — Read group ID to add on `@RG` header line
`''` (default) | character vector | string

Read group ID to add on the `@RG` header line in the output SAM report, specified as a character vector or string. If you specify any read group ID, the function prints the `@RG` header line with the tag `ID:` followed by the specified group ID.

Example: `'ReadGroupID','ID1'`

Data Types: `char` | `string`

`ReadGroup` — Read group information to add as field on `@RG` header line
`''` (default) | character vector | string

Read group information to add as a field on the `@RG` header line in the output SAM report, specified as a character vector or string. This property applies only if you specify `'ReadGroupID'`.

Example: `'ReadGroup','Control'`

Data Types: `char` | `string`

`RefGapCosts` — Gap costs for opening and extending gap
`[5 3]` (default) | two-element vector of nonnegative integers

Gap costs for opening and extending a gap on the reference, specified as a two-element vector of nonnegative integers. The first element is the cost of opening a gap, and the second element is the cost of extending a gap. Given the cost vector `[GO GE]`, a reference gap of length *N* is assigned a penalty of `GO + N * GE`.

Example: `'RefGapCosts',[4 2]`

Data Types: `double`

`Reorder` — Flag to reorder SAM records
`false` (default) | `true`

Flag to reorder SAM records to maintain the same order as in the input files, specified as `true` or `false`. This property applies only when the number of parallel threads is greater than one. When you use one thread, the order of the records in the output is the same as the order of the input.

Example: `'Reorder',true`

Data Types: `logical`

`Seed` — Number to set seed in pseudo-random number generator
`0` (default) | nonnegative integer

Number to set the seed in the pseudo-random number generator, specified as a nonnegative integer.

Example: `'Seed',3`

Data Types: `double`

`SeedIntervalFunction` — Function governing distance between seed substrings
character vector | string

Function governing the distance between seed substrings during the multiseed alignment, specified as a character vector or string.

The function has the format `'f,B,A'`, where *f* is a function type, *B* is a constant term, and *A* is a coefficient. Available function types are:

- `'C'`– Constant
- `'L'`– Linear
- `'S'`– Square root
- `'G'`– Natural log

The resulting function is `H(x) = B + A * f(x)`, where *x* is the read length.

For the `'EndToEnd'` alignment mode, the default function is `'S,1,1.15'`. For the `'Local'` mode, the default function is `'S,1,0.75'`.

Example: `'SeedIntervalFunction','S,2,2.15'`

Data Types: `char` | `string`

`SeedLength` — Seed substring length to align during multiseed alignment
`20` (default) | positive integer

Seed substring length to align during the multiseed alignment, specified as a positive integer.

Example: `'SeedLength',25`

Data Types: `double`

`Skip` — Number of reads to ignore
`0` (default) | nonnegative integer

Number of reads to ignore from the beginning of the input files, specified as a nonnegative integer.

Example: `'Skip',5`

Data Types: `double`

`Trim3` — Number of residues to trim from 3' end
`0` (default) | nonnegative integer

Number of residues to trim from the 3' end of each read before aligning, specified as a nonnegative integer.

Example: `'Trim3',5`

Data Types: `double`

`Trim5` — Number of residues to trim from 5' end
`0` (default) | nonnegative integer

Number of residues to trim from the 5' end of each read before aligning, specified as a nonnegative integer.

Example: `'Trim5',5`

Data Types: `double`

UpTo — Number of reads to consider from beginning of input files
`Inf` (default) | positive integer

Number of reads to consider from the beginning of input files, specified as a positive integer. The default value is `Inf`, that is, all reads are considered.

Example: `'UpTo',1000`

Data Types: `double`

Object Functions

getBowtie2Command	Translate object properties to Bowtie 2 options
getBowtie2Table	Retrieve table with object properties and equivalent Bowtie 2 options
preset	Set combination of alignment options
run	Map sequence reads to reference sequence using Bowtie 2

Examples

Align Reads to Reference Sequence Using Bowtie 2

Build a set of index files for the Drosophila genome. An error message appears if you do not have the Bowtie 2 Support Package for Bioinformatics Toolbox installed when you run the function. Click the provided link to download the package from the Add-on menu.

For this example, the reference sequence `Dmel_chr4.fa` is already provided with the toolbox.

```
status = bowtie2build('Dmel_chr4.fa', 'Dmel_chr4_index');
```

If the index build is successful, the function returns `0` and creates the index files (`*.bt2`) in the current folder. The files have the prefix `'Dmel_chr4_index'`.

Sometimes the index files exist, and you want to know the reference sequence used to build the index. In this case, use the `bowtie2inspect` function to get more information about the reference.

```
bowtie2inspect('Dmel_chr4', 'Dmel_chr4_retrieved.fa');
```

By default, the output file `Dmel_chr4_retrieved.fa` contains the sequence of the reference. You can also get a summary information about the reference name and lengths instead of the actual sequence. For details on the available options, see `Bowtie2InspectOptions`.

Once the index is ready, map the read sequences to the reference using the `bowtie2` function. The paired-end read files (`SRR6008575_10k_1.fq` and `SRR6008575_10k_2.fq`) are already provided with the toolbox.

```
bowtie2('Dmel_chr4','SRR6008575_10k_1.fq','SRR6008575_10k_2.fq','SRR6008575_10k_chr4.sam');
```

The output is a SAM-formatted file that contains the mapping results.

You can specify different alignment options by passing in a Bowtie 2 syntax string or using a `Bowtie2AlignOptions` object.

Suppose you want to trim some residues from the 3' end before aligning. First, create a Bowtie2AlignOptions object.

```
alignOpt = Bowtie2AlignOptions;
```

Trim four residues from the 3' end before aligning.

```
alignOpt.Trim3 = 4;
```

Map reads to the reference using the specified alignment option.

```
flag = bowtie2('Dmel_chr4','SRR6008575_10k_1.fq','SRR6008575_10k_2.fq','SRR6008575_10k_chr4_trimm
```

Version History

Introduced in R2018a

References

[1] Langmead, B., and S. Salzberg. "Fast gapped-read alignment with Bowtie 2." *Nature Methods*. 9, 2012, 357–359.

See Also

bowtie2 | bowtie2inspect | bowtie2build | Bowtie2BuildOptions | Bowtie2InspectOptions

External Websites

Bowtie 2 manual

bowtie2build

Create Bowtie 2 index files from reference sequences

Syntax

```
bowtie2build(referenceFileNames,indexBaseName)
bowtie2build( ___ ,buildOptions)
flag = bowtie2build( ___ )
```

Description

`bowtie2build(referenceFileNames,indexBaseName)` builds Bowtie2 index files from the reference sequence information saved in the FASTA files specified by `referenceFileNames`.

`bowtie2build` requires the Bowtie 2 Support Package for Bioinformatics Toolbox. If this support package is not installed, then the function provides a download link. For details, see “Bioinformatics Toolbox Software Support Packages”.

`bowtie2build( ___ ,buildOptions)` uses the additional options specified by `buildOptions`. Specify these options after all other input arguments.

`flag = bowtie2build( ___ )` returns an exit `flag` of the function using any of the input arguments in the previous syntaxes.

Examples

Build Bowtie2 Index Files for Reference Sequence

Build a set of index files for the Drosophila genome. An error message appears if you do not have the Bowtie 2 Support Package for Bioinformatics Toolbox installed when you run the function. Click the provided link to download the package from the Add-on menu.

For this example, the reference sequence `Dmel_chr4.fa` is already provided with the toolbox.

```
status = bowtie2build('Dmel_chr4.fa', 'Dmel_chr4_index');
```

If the index build is successful, the function returns `0` and creates the index files (`*.bt2`) in the current folder. The files have the prefix `'Dmel_chr4_index'`.

You can specify different options by using a `Bowtie2BuildOptions` object or by passing in a Bowtie 2 syntax string. For instance, you can specify whether to force the creation of a large index even if the reference is less than four billion nucleotides long as follows.

```
buildOpt = Bowtie2BuildOptions;
```

Set the `ForceLargeIndex` option to true.

```
buildOpt.ForceLargeIndex = true;
```

Build the index files using the specified option.

```
bowtie2build('Dmel_chr4.fa', 'Dmel_chr4_index_large',buildOpt);
```

Alternatively, you can pass in a Bowtie 2 syntax string.

```
flag = bowtie2build('Dmel_chr4.fa', 'Dmel_chr4_index_large2','--large-index');
```

Input Arguments

`referenceFileNames` — Names of files with reference sequence information

string | character vector | string array | cell array of character vectors

Names of files with reference sequence information, specified as a string, character vector, string array, or cell array of character vectors.

Example: `'Dmel_chr4.fa'`

Data Types: `char` | `string` | `cell`

`indexBaseName` — Base name of reference index files

character vector | string

Base name (prefix) of the reference index files, specified as a character vector or string. The index files are in the `BT2` or `BT21` format.

Example: `'Dmel_chr4'`

Data Types: `char` | `string`

`buildOptions` — Options to build index files

character vector | string | `Bowtie2BuildOptions` object

Options to build index files, specified as a character vector, string, or `Bowtie2BuildOptions` object. The character vector or string must be in the Bowtie 2 option syntax (prefixed by one or two dashes) [1].

For a `Bowtie2BuildOptions` object, only the modified properties are used to run the function.

Example: `'--trim5 10 -s 5'`

Output Arguments

`flag` — Exit status

integer

Exit status of the function, returned as an integer. `flag` is `0` if the function runs without errors or warning. Otherwise, it is nonzero.

Version History

Introduced in R2018a

References

[1] Langmead, B., and S. Salzberg. "Fast gapped-read alignment with Bowtie 2." *Nature Methods*. 9, 2012, 357–359.

See Also

`bowtie2` | `bowtie2inspect` | `Bowtie2AlignOptions` | `Bowtie2BuildOptions` | `Bowtie2InspectOptions` | `featurecount` | `cufflinks`

Topics

"Count Features from NGS Reads"

External Websites

Bowtie 2 manual

Bowtie2BuildOptions

Contain options to create Bowtie 2 index files from reference sequences

Description

A `Bowtie2BuildOptions` object contains options to run the `bowtie2build` function that builds Bowtie 2 index files from reference sequences.

Creation

Syntax

```
buildOptions = Bowtie2BuildOptions
buildOptions = Bowtie2BuildOptions(Name,Value)
buildOptions = Bowtie2BuildOptions(S)
```

Description

`buildOptions = Bowtie2BuildOptions` creates a `Bowtie2BuildOptions` object with default property values.

`Bowtie2BuildOptions` requires the Bowtie 2 Support Package for Bioinformatics Toolbox. If this support package is not installed, then the function provides a download link. For details, see "Bioinformatics Toolbox Software Support Packages".

`buildOptions = Bowtie2BuildOptions(Name,Value)` sets properties on page 1-452 using one or more name-value pairs. Enclose each property name in quotes. For example, `buildOptions = Bowtie2BuildOptions('ForceLargeIndex',true)` specifies to force the creation of a large index even if the reference is less than 4 billion nucleotides long.

`buildOptions = Bowtie2BuildOptions(S)` specifies optional parameters in a character vector or string `S`.

Input Arguments

`S` — Parameters to build index files
character vector

Parameters to build the index files, specified as a character vector. `S` must be in the Bowtie 2 option syntax (prefixed by one or two dashes) [1].

Data Types: `char` | `string`

Properties

`BuildOnlyReference` — Boolean indicator to build only bitpacked reference
`false` (default) | `true`

Boolean indicator to build only the `3.bt2` and `4.bt2` files that correspond to the bitpacked version of reference sequences, specified as `true` or `false`.

Example: `'BuildOnlyReference',true`

Data Types: `logical`

`BuildNoReference` — Boolean indicator to omit building bitpacked reference
`false` (default) | `true`

Boolean indicator to omit building the `3.bt2` and `4.bt2` files that correspond to the bitpacked version of reference sequences, specified as `true` or `false`.

Example: `'BuildNoReference',true`

Data Types: `logical`

`ExtraBowtie2Command` — Additional options not included in object properties
`''` (default) | character vector

Additional options not included in the object properties, specified as a character vector. The character vector must be in the Bowtie 2 option syntax (prefixed by one or two dashes). The default value is an empty character vector `''`.

Example: `'ExtraBowtie2Command','--version'`

Data Types: `char` | `string`

`ForceLargeIndex` — Boolean indicator to force building large index
`false` (default) | `true`

Boolean indicator to force building a large index even if the reference is less than four billion nucleotides long, specified as `true` or `false`.

Example: `'ForceLargeIndex',true`

Data Types: `logical`

`NumThreads` — Number of parallel threads to build index files
`1` (default) | positive integer

Number of parallel threads to build the index files, specified as a positive integer. Threads are run on separate processors or cores. Increasing the number of threads provides significant speed-up (close to linear) but also increases the memory footprint.

Example: `'NumThreads',4`

Data Types: `double`

`Offrate` — Number of Burrows-Wheeler rows to mark when building index
`5` (default) | positive integer

Number of Burrows-Wheeler rows to mark when building the index files, specified as a positive integer. To map the alignment back to positions on the reference sequences, the function uses this number to mark some of the rows in the Burrows-Wheeler algorithm with their corresponding location on the genome. The function marks every 2^n, where n is the offrate.

Increasing the number of marked rows makes the reference position lookups faster, but requires more memory.

Example: `'Offrate',6`

Data Types: `double`

Seed — Number to set seed in pseudo-random number generator
`0` (default) | nonnegative integer

Number to set the seed in the pseudo-random number generator, specified as a nonnegative integer.

Example: `'Seed',3`

Data Types: `double`

Object Functions

getBowtie2Command	Translate object properties to Bowtie 2 options
getBowtie2Table	Retrieve table with object properties and equivalent Bowtie 2 options
run	Build Bowtie 2 index files

Examples

Build Bowtie 2 Index Files for Reference Sequence

Build a set of index files for the Drosophila genome. An error message appears if you do not have the Bowtie 2 Support Package for Bioinformatics Toolbox installed when you run the function. Click the provided link to download the package from the Add-on menu.

For this example, the reference sequence `Dmel_chr4.fa` is already provided with the toolbox.

```
status = bowtie2build('Dmel_chr4.fa', 'Dmel_chr4_index');
```

If the index build is successful, the function returns `0` and creates the index files (`*.bt2`) in the current folder. The files have the prefix `'Dmel_chr4_index'`.

You can specify different options by using a `Bowtie2BuildOptions` object or by passing in a Bowtie 2 syntax string. For instance, you can specify whether to force the creation of a large index even if the reference is less than four billion nucleotides long as follows.

```
buildOpt = Bowtie2BuildOptions;
```

Set the `ForceLargeIndex` option to true.

```
buildOpt.ForceLargeIndex = true;
```

Build the index files using the specified option.

```
bowtie2build('Dmel_chr4.fa', 'Dmel_chr4_index_large',buildOpt);
```

Alternatively, you can pass in a Bowtie 2 syntax string.

```
flag = bowtie2build('Dmel_chr4.fa', 'Dmel_chr4_index_large2','--large-index');
```

Version History

Introduced in R2018a

References

[1] Langmead, B., and S. Salzberg. "Fast gapped-read alignment with Bowtie 2." *Nature Methods*. 9, 2012, 357–359.

See Also

`bowtie2` | `bowtie2inspect` | `bowtie2build` | `Bowtie2AlignOptions` | `Bowtie2InspectOptions`

External Websites

Bowtie 2 manual

bowtie2inspect

Inspect Bowtie 2 index files

Syntax

```
bowtie2inspect(indexBaseName,outputFileName)
bowtie2inspect( ___ ,inspectOptions)
flag = bowtie2inspect( ___ )
```

Description

`bowtie2inspect(indexBaseName,outputFileName)` inspects Bowtie2 index files with the prefix `indexBaseName`, checks the original reference sequences used to build the index, and saves the reference sequences in an output file `outputFileName`.

`bowtie2inspect` requires the Bowtie 2 Support Package for Bioinformatics Toolbox. If this support package is not installed, then the function provides a download link. For details, see “Bioinformatics Toolbox Software Support Packages”.

`bowtie2inspect( ___ ,inspectOptions)` uses the additional options specified by `inspectOptions`. Specify the options after all other input arguments.

`flag = bowtie2inspect( ___ )` returns an exit `flag` of the function using any of the input arguments in the previous syntaxes.

Examples

Inspect Bowtie2 Index and Retrieve Reference Sequence Information

Get information about the reference sequence used to build the corresponding index files by using `bowtie2inspect`. An error message appears if you do not have the Bowtie 2 Support Package for Bioinformatics Toolbox installed when you run the function. Click the provided link to download the package from the Add-on menu.

```
bowtie2inspect('Dmel_chr4', 'Dmel_chr4_retrieved.fa');
```

By default, the output file `Dmel_chr4_retrieved.fa` contains the actual sequence of the reference used to build the index.

You can also get summary information about the reference name and lengths instead of the actual sequence.

Create an options object.

```
inspectOpt = Bowtie2InspectOptions;
```

Set the `Summary` property to `true`.

```
inspectOpt.Summary = true;
```

Run the function again using the specified option.

```
flag = bowtie2inspect('Dmel_chr4', 'Dmel_chr4_summary.fa',inspectOpt);
```

If the index inspection is successful, the function returns `0`, and the output file now contains summary information of the reference sequence.

Alternatively, you can pass in a Bowtie 2 syntax string instead of using the option object.

```
flag = bowtie2inspect('Dmel_chr4', 'Dmel_chr4_summary2.fa','-s');
```

Input Arguments

`indexBaseName` — Base name of reference index files

character vector | string

Base name (prefix) of the reference index files, specified as a character vector or string. The index files are in the `BT2` or `BT21` format.

Example: `'Dmel_chr4'`

Data Types: `char` | `string`

`outputFileName` — Name of output file

string | character vector

Name of an output file, specified as a string or character vector. By default, the output file contains the reference sequences that are used to build the index files.

Example: `'refSeq.fa'`

Data Types: `char` | `string`

`inspectOptions` — Options to inspect index files

character vector | string | `Bowtie2InspectOptions` object

Options to inspect index files, specified as a character vector, string, or `Bowtie2InspectOptions` object. The character vector or string must be in the Bowtie 2 option syntax (prefixed by one or two dashes) [1].

For a `Bowtie2InspectOptions` object, only the modified properties are used to run the function.

Example: `'--trim5 10 -s 5'`

Output Arguments

`flag` — Exit status

integer

Exit status of the function, returned as an integer. `flag` is `0` if the function runs without errors or warning. Otherwise, it is nonzero.

Version History

Introduced in R2018a

References

[1] Langmead, B., and S. Salzberg. "Fast gapped-read alignment with Bowtie 2." *Nature Methods*. 9, 2012, 357–359.

See Also

`bowtie2` | `bowtie2build` | `Bowtie2AlignOptions` | `Bowtie2BuildOptions` | `Bowtie2InspectOptions`

External Websites

Bowtie 2 manual

Bowtie2InspectOptions

Contain options to inspect Bowtie 2 index files

Description

A `Bowtie2InspectOptions` object contains options to run the `bowtie2inspect` function that inspects Bowtie 2 index files.

Creation

Syntax

```
inspectOptions = Bowtie2InspectOptions
inspectOptions = Bowtie2InspectOptions(Name,Value)
inspectOptions = Bowtie2InspectOptions(S)
```

Description

`inspectOptions = Bowtie2InspectOptions` creates a `Bowtie2InspectOptions` object with default property values.

`Bowtie2InspectOptions` requires the Bowtie 2 Support Package for Bioinformatics Toolbox. If this support package is not installed, then the function provides a download link. For details, see "Bioinformatics Toolbox Software Support Packages".

`inspectOptions = Bowtie2InspectOptions(Name,Value)` sets properties on page 1-459 using one or more name-value pairs. Enclose each property name in quotes. For instance, `inspectOptions = Bowtie2InspectOptions('Summary',true)` specifies to return a summary of the index content instead of reference sequences.

`inspectOptions = Bowtie2InspectOptions(S)` specifies optional parameters in a character vector or string `S`.

Input Arguments

S — Parameters to inspect index files
character vector

Parameters to inspect the index files, specified as a character vector. `S` must be in the Bowtie 2 option syntax (prefixed by one or two dashes) [1].

Data Types: `char` | `string`

Properties

`ExtraBowtie2Command` — Additional options not included in object properties
`''` (default) | character vector

Additional options not included in the object properties, specified as a character vector. The character vector must be in the Bowtie 2 option syntax (prefixed by one or two dashes). The default value is an empty character vector `''`.

Example: `'ExtraBowtie2Command','--version'`

Data Types: `char` | `string`

`NamesOnly` — Boolean indicator to print only reference sequence names in output
`false` (default) | `true`

Boolean indicator to print only the reference sequence names in the output file, specified as `true` or `false`.

Example: `'NamesOnly',true`

Data Types: `logical`

`Summary` — Boolean indicator to print summary instead of actual reference sequences
`false` (default) | `true`

Boolean indicator to print a summary information about reference sequence names and lengths instead of actual sequences in the output file, specified as `true` or `false`.

Example: `'Summary',true`

Data Types: `logical`

Object Functions

getBowtie2Command	Translate object properties to Bowtie 2 options
getBowtie2Table	Retrieve table with object properties and equivalent Bowtie 2 options
run	Inspect Bowtie 2 index files

Examples

Inspect Bowtie 2 Index and Retrieve Reference Sequence Information

Get information about the reference sequence used to build the corresponding index files by using `bowtie2inspect`. An error message appears if you do not have the Bowtie 2 Support Package for Bioinformatics Toolbox installed when you run the function. Click the provided link to download the package from the Add-on menu.

```
bowtie2inspect('Dmel_chr4', 'Dmel_chr4_retrieved.fa');
```

By default, the output file `Dmel_chr4_retrieved.fa` contains the actual sequence of the reference used to build the index.

You can also get summary information about the reference name and lengths instead of the actual sequence.

Create an options object.

```
inspectOpt = Bowtie2InspectOptions;
```

Set the `Summary` property to `true`.

```
inspectOpt.Summary = true;
```

Run the function again using the specified option.

```
flag = bowtie2inspect('Dmel_chr4', 'Dmel_chr4_summary.fa',inspectOpt);
```

If the index inspection is successful, the function returns 0, and the output file now contains summary information of the reference sequence.

Alternatively, you can pass in a Bowtie 2 syntax string instead of using the option object.

```
flag = bowtie2inspect('Dmel_chr4', 'Dmel_chr4_summary2.fa','-s');
```

Version History

Introduced in R2018a

References

[1] Langmead, B., and S. Salzberg. "Fast gapped-read alignment with Bowtie 2." *Nature Methods*. 9, 2012, 357–359.

See Also

bowtie2 | bowtie2build | bowtie2inspect | Bowtie2AlignOptions | Bowtie2BuildOptions

External Websites

Bowtie 2 manual

bowtiebuild

(Removed) Generate index using Burrows-Wheeler transform

Note `bowtiebuild` has been removed. Use `bowtie2` and `bowtie2build` instead.

Syntax

```
bowtiebuild(input,indexBaseName)
bowtiebuild(input,indexBaseName,'BowtieBuildOptions',options)
```

Description

`bowtiebuild(input,indexBaseName)` builds an index using the reference sequence(s) in `input`, and saves it to the index file `indexBaseName`.

Note `bowtiebuild` runs on Mac and UNIX platforms only.

`bowtiebuild(input,indexBaseName,'BowtieBuildOptions',options)` specifies additional options.

Examples

Build a Bowtie Index

Download the *E. coli* genome from NCBI.

```
getgenbank('NC_008253','tofile','NC_008253.fna','SequenceOnly',true)
```

Built a Bowtie index with the base name `ECOLI`.

```
bowtiebuild('NC_008253.fna','ECOLI')
```

Input Arguments

`input` — FASTA-formatted files
character vector | string | string vector | cell array of character vectors

FASTA-formatted files with the reference sequences to be indexed, specified as a character vector, string, string vector, or cell array of character vectors. Use a cell array of character vectors or string vector to specify multiple files.

`indexBaseName` — Name for indexed reference file
character vector | string

Name for indexed reference file, specified as a character vector or string containing the path and base name for the resulting Bowtie index file.

`options` — Additional `bowtiebuild` options

character vector | string

Additional `bowtiebuild` options, specified as a character vector or string for any valid `bowtiebuild` options. Type `bowtiebuild('--help')` for available options.

Example: `'-t 5 -C'`

Tips

- More information on the Bowtie algorithm (Version 0.12.7) can be found at http://bowtie-bio.sourceforge.net/index.shtml.
- Some prebuilt index files for model organisms can be downloaded directly from the Bowtie repository.

Version History

Introduced in R2012b

R2023a: Removed

Errors starting in R2023a

`bowtiebuild` has been removed. Use `bowtie2` and `bowtie2build` instead.

R2022b: Warns

Warns starting in R2022b

`bowtiebuild` warns that it will be removed in a future release.

R2022a: To be removed

Not recommended starting in R2022a

`bowtiebuild` runs without a warning, but it will be removed in a future release.

See Also

`baminfo` | `BioMap` | `fastainfo` | `fastqinfo` | `samread` | `saminfo`

bwaindex

Create BWA indices from reference sequence

Syntax

```
bwaindex(referenceFile)
bwaindex(referenceFile,indexOptions)
bwaindex(referenceFile,Name,Value)
```

Description

`bwaindex(referenceFile)` creates BWA index files for the reference sequence in `referenceFile` [1][2]. By default, the function writes the index files to the same directory as `referenceFile`.

The index files are in the AMB, ANN, BWT, PAC, and SA file formats.

`bwaindex` requires the BWA Support Package for Bioinformatics Toolbox. If the support package is not installed, then the function provides a download link. For details, see “Bioinformatics Toolbox Software Support Packages”.

`bwaindex(referenceFile,indexOptions)` uses additional options specified by `indexOptions`.

`bwaindex(referenceFile,Name,Value)` uses additional options specified by one or more name-value pair arguments. For example, `bwaindex(referenceFile,'Algorithm','is')` specifies the linear-time algorithm.

Examples

Align Reads to Reference Sequence Using BWA

This example requires the BWA Support Package for Bioinformatics Toolbox™. If the support package is not installed, the software provides a download link. For details, see “Bioinformatics Toolbox Software Support Packages”.

Build a set of index files for the Drosophila genome. This example uses the reference sequence `Dmel_chr4.fa`, provided with the toolbox. The `'Prefix'` argument lets you define the prefix of the output index files. You can also include the file path information. For this example, define the prefix as `Dmel_chr4` and save the index files in the current directory.

```
bwaindex('Dmel_chr4.fa','Prefix','./Dmel_chr4');
```

As an alternative to specifying name-value pair arguments, you can use the `BWAIndexOptions` object to specify the indexing options.

```
indexOpt = BWAIndexOptions;
indexOpt.Prefix = './Dmel_chr4';
indexOpt.Algorithm = 'bwtsw';
bwaindex('Dmel_chr4.fa',indexOpt);
```

Once the index files are ready, map the read sequences to the reference using `bwamem`. Two pair-end read input files are already provided with the toolbox. Using name-value pair arguments, you can specify different alignment options, such as the number of parallel threads to use.

```
bwamem('Dmel_chr4','SRR6008575_10k_1.fq','SRR6008575_10k_2.fq','SRR6008575_10k_chr4.sam','NumThre
```

Alternatively, you can use `BWAMEMoptions` to specify the alignment options.

```
alignOpt = BWAMEMOptions;
alignOpt.NumThreads = 4;
bwamem('Dmel_chr4','SRR6008575_10k_1.fq','SRR6008575_10k_2.fq','SRR6008575_10k_chr4.sam',alignOp
```

Input Arguments

`referenceFile` — Reference file name

character vector | string

Reference file name, specified as a character vector or string. The file must be a FASTA-formatted file with the reference sequence information for indexing.

Data Types: `char` | `string`

`indexOptions` — Additional options for indexing

`BWAIndexOptions` object | character vector | string

Additional options for indexing, specified as a `BWAIndexOptions` object, character vector, or string. The character vector or string must be in the `bwa index` native syntax (prefixed by a dash). If you specify a `BWAIndexOptions` object, the function uses only those properties that are set or modified.

Data Types: `char` | `string`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `bwaindex(referenceFile,'Algorithm','bwtsw')` specifies to use the BWT-SW algorithm.

`Algorithm` — Algorithm to construct BWT index

`'is'` | `'bwtsw'`

Algorithm to construct the BWT (Burrows-Wheeler transform) index, specified as a character vector or string. Options are:

- `'is'` — Linear-time algorithm. The memory requirement for using this option is 5.37 times the size of the database. You cannot use this option if your database is larger than 2 GB.
- `'bwtsw'` — BWT-SW algorithm.

The default algorithm is chosen automatically based on the size of the reference genome.

Data Types: `char` | `string`

`BlockSize` — Number of bases processed per batch
`1e7` (default) | positive scalar

Number of bases processed per batch in the `bwtsw` algorithm, specified as a positive scalar.

Data Types: `double`

`ExtraCommand` — Additional commands
`""` (default) | character vector | string

Additional commands, specified as a character vector or string.

The commands must be in the native syntax (prefixed by one or two dashes). Use this option to apply undocumented flags and flags without corresponding MATLAB properties.

Example: `'ExtraCommand','-6'`

Data Types: `char` | `string`

`IncludeAll` — Flag to apply all available options
`false` (default) | true

Flag to include all available options with the corresponding default values when converting to the original options syntax, specified as `true` or `false`.

The original (native) syntax is prefixed by one or two dashes. By default, the function converts only the specified options. If the value is `true`, the software converts all available options, with default values for unspecified options, to the original syntax.

Note If you set `IncludeAll` to `true`, the software translates all available properties, with default values for unspecified properties. The only exception is that when the default value of a property is `NaN`, `Inf`, `[]`, `''`, or `""`, then the software does not translate the corresponding property.

Example: `'IncludeAll',true`

Data Types: `logical`

`Prefix` — Prefix for output index files
character vector | string

Prefix for the output index files, specified as a character vector or string. You can specify only the prefix or a file path and prefix. The default value is the same as the input FASTA file name.

Example: `'Prefix','D:/ngs/GRCh38_p12'`

Data Types: `char` | `string`

Version History

Introduced in R2020b

References

[1] Li, Heng, and Richard Durbin. “Fast and Accurate Short Read Alignment with Burrows-Wheeler Transform.” *Bioinformatics* 25, no. 14 (July 15, 2009): 1754–60. https://doi.org/10.1093/bioinformatics/btp324.

[2] Li, Heng, and Richard Durbin. “Fast and Accurate Long-Read Alignment with Burrows–Wheeler Transform.” *Bioinformatics* 26, no. 5 (March 1, 2010): 589–95. https://doi.org/10.1093/bioinformatics/btp698.

See Also

`BWAIndexOptions` | `bwamem`

Topics

“Bioinformatics Toolbox Software Support Packages”

BWAIndexOptions

Option set for `bwaindex`

Description

A `BWAIndexOptions` object contains options for the `bwaindex` function, which creates indices from a reference sequence [1][2].

Creation

Syntax

```
bwaindexOpt = BWAIndexOptions
bwaindexOpt = BWAIndexOptions(Name,Value)
bwaindexOpt = BWAIndexOptions(S)
```

Description

`bwaindexOpt = BWAIndexOptions` creates a `BWAIndexOptions` object with the default property values.

`BWAIndexOptions` requires the BWA Support Package for Bioinformatics Toolbox. If the support package is not installed, then the function provides a download link. For details, see “Bioinformatics Toolbox Software Support Packages”.

`bwaindexOpt = BWAIndexOptions(Name,Value)` sets the object properties on page 1-468 using one or more name-value pair arguments. Enclose each property name in quotes. For example, `bwaindexOpt = BWAIndexOptions('Prefix','dmel_chr4')` specifies the prefix for the index files.

`bwaindexOpt = BWAIndexOptions(S)` specifies optional parameters using a string or character vector `S`.

Input Arguments

`S` — `bwa index` options
character vector | string

`bwa index` options, specified as a character vector or string. `S` must be in the `bwa index` native syntax (prefixed by a dash).

Example: `'-a bwtsw -b 5000000'`

Properties

`Algorithm` — Algorithm to construct BWT index
empty string array (default) | `'is'` | `'bwtsw'`

Algorithm to construct the BWT (Burrows-Wheeler transform) index, specified as a character vector or string. Options are:

- `'is'` — Linear-time algorithm. The memory requirement for using this option is 5.37 times the size of the database. You cannot use this option if your database is larger than 2 GB.
- `'bwtsw'` — BWT-SW algorithm.

The default algorithm is chosen automatically based on the size of the reference genome.

Data Types: `char` | `string`

`BlockSize` — Number of bases processed per batch
`1e7` (default) | positive scalar

Number of bases processed per batch in the `bwtsw` algorithm, specified as a positive scalar.

Data Types: `double`

`ExtraCommand` — Additional commands
`""` (default) | character vector | string

Additional commands, specified as a character vector or string.

The commands must be in the native syntax (prefixed by one or two dashes). Use this option to apply undocumented flags and flags without corresponding MATLAB properties.

When the software converts the original flags to MATLAB properties, it stores any unrecognized flags in this property.

Example: `'-6'`

Data Types: `char` | `string`

`IncludeAll` — Flag to apply all available options
`false` (default) | `true`

Flag to include all available options with the corresponding default values during conversion to the original options syntax, specified as `true` or `false`.

The original (native) syntax is prefixed by one or two dashes. By default, the function converts only the specified options. If the value is `true`, the software converts all available options, with default values for unspecified options, to the original syntax.

Note If you set `IncludeAll` to `true`, the software translates all available properties, with default values for unspecified properties. The only exception is that when the default value of a property is `NaN`, `Inf`, `[]`, `''`, or `""`, then the software does not translate the corresponding property.

Data Types: `logical`

`Prefix` — Prefix for output index files
empty string array (default) | character vector | string

Prefix for the output index files, specified as a character vector or string. You can specify only the prefix or a file path and prefix. The default value is the same as the input FASTA file name.

Example: `'D:/ngs/GRCh38_p12'`

Data Types: char | string

Version — Supported version
string

This property is read-only.

Supported version of the original bwa software, returned as a string.

Example: "0.7.17"

Data Types: string

Object Functions

getCommand	Translate object properties to original options syntax
getOptionsTable	Return table with all properties and equivalent options in original syntax

Examples

Align Reads to Reference Sequence Using BWA

This example requires the BWA Support Package for Bioinformatics Toolbox™. If the support package is not installed, the software provides a download link. For details, see “Bioinformatics Toolbox Software Support Packages”.

Build a set of index files for the Drosophila genome. This example uses the reference sequence Dmel_chr4.fa, provided with the toolbox. The 'Prefix' argument lets you define the prefix of the output index files. You can also include the file path information. For this example, define the prefix as Dmel_chr4 and save the index files in the current directory.

```
bwaindex('Dmel_chr4.fa','Prefix','./Dmel_chr4');
```

As an alternative to specifying name-value pair arguments, you can use the BWAIndexOptions object to specify the indexing options.

```
indexOpt = BWAIndexOptions;
indexOpt.Prefix = './Dmel_chr4';
indexOpt.Algorithm = 'bwtsw';
bwaindex('Dmel_chr4.fa',indexOpt);
```

Once the index files are ready, map the read sequences to the reference using bwamem. Two pair-end read input files are already provided with the toolbox. Using name-value pair arguments, you can specify different alignment options, such as the number of parallel threads to use.

```
bwamem('Dmel_chr4','SRR6008575_10k_1.fq','SRR6008575_10k_2.fq','SRR6008575_10k_chr4.sam','NumThre
```

Alternatively, you can use BWAMEMoptions to specify the alignment options.

```
alignOpt = BWAMEMOptions;
alignOpt.NumThreads = 4;
bwamem('Dmel_chr4','SRR6008575_10k_1.fq','SRR6008575_10k_2.fq','SRR6008575_10k_chr4.sam',alignOp
```

Version History

Introduced in R2020b

References

[1] Li, Heng, and Richard Durbin. “Fast and Accurate Short Read Alignment with Burrows-Wheeler Transform.” *Bioinformatics* 25, no. 14 (July 15, 2009): 1754–60. https://doi.org/10.1093/bioinformatics/btp324.

[2] Li, Heng, and Richard Durbin. “Fast and Accurate Long-Read Alignment with Burrows–Wheeler Transform.” *Bioinformatics* 26, no. 5 (March 1, 2010): 589–95. https://doi.org/10.1093/bioinformatics/btp698.

See Also

`bwaindex`

Topics

“Bioinformatics Toolbox Software Support Packages”

bwamem

Map sequence reads to reference genome using BWA

Syntax

```
bwamem(indexBaseName,reads1,reads2,outputFileName)
bwamem(___,options)
bwamem(___,Name,Value)
```

Description

`bwamem(indexBaseName,reads1,reads2,outputFileName)` maps the sequencing reads from `reads1` and `reads2` against the reference sequence and writes the results to the output file `outputFileName`. The input `indexBaseName` represents the base name (prefix) of the reference index files [1][2].

`bwamem` requires the BWA Support Package for Bioinformatics Toolbox. If the support package is not installed, then the function provides a download link. For details, see “Bioinformatics Toolbox Software Support Packages”.

`bwamem(___,options)` uses the additional options specified by `options`. Specify these options after all other input arguments.

`bwamem(___,Name,Value)` uses additional options specified by one or more name-value pair arguments. For example, `'BandWidth',90` sets the maximum allowable gap length to 90.

Examples

Align Reads to Reference Sequence Using BWA

This example requires the BWA Support Package for Bioinformatics Toolbox™. If the support package is not installed, the software provides a download link. For details, see “Bioinformatics Toolbox Software Support Packages”.

Build a set of index files for the Drosophila genome. This example uses the reference sequence `Dmel_chr4.fa`, provided with the toolbox. The `'Prefix'` argument lets you define the prefix of the output index files. You can also include the file path information. For this example, define the prefix as `Dmel_chr4` and save the index files in the current directory.

```
bwaindex('Dmel_chr4.fa','Prefix','./Dmel_chr4');
```

As an alternative to specifying name-value pair arguments, you can use the `BWAIndexOptions` object to specify the indexing options.

```
indexOpt = BWAIndexOptions;
indexOpt.Prefix = './Dmel_chr4';
indexOpt.Algorithm = 'bwtsw';
bwaindex('Dmel_chr4.fa',indexOpt);
```

Once the index files are ready, map the read sequences to the reference using `bwamem`. Two pair-end read input files are already provided with the toolbox. Using name-value pair arguments, you can specify different alignment options, such as the number of parallel threads to use.

```
bwamem('Dmel_chr4','SRR6008575_10k_1.fq','SRR6008575_10k_2.fq','SRR6008575_10k_chr4.sam','NumThre
```

Alternatively, you can use `BWAMEMoptions` to specify the alignment options.

```
alignOpt = BWAMEMOptions;
alignOpt.NumThreads = 4;
bwamem('Dmel_chr4','SRR6008575_10k_1.fq','SRR6008575_10k_2.fq','SRR6008575_10k_chr4.sam',alignOp
```

Input Arguments

`indexBaseName` — Base name of reference index files
character vector | string

Base name (prefix) of the reference index files, specified as a character vector or string. For example, the base name of an index file `'Dmel_chr4.bwt'` is `'Dmel_chr4'`.

The index files are in the AMB, ANN, BWT, PAC, and SA file formats.

Example: `'Dmel_chr4'`

Data Types: `char` | `string`

`reads1` — Name of file with first mate reads or single-end reads
character vector | string

Name of the file with the first mate reads or single-end reads, specified as a character vector or string.

For paired-end data, the sequences in `reads1` must correspond read-for-read to sequences in `reads2`.

Example: `'SRR6008575_10k_1.fq'`

Data Types: `char` | `string`

`reads2` — Name of file with second mate reads
character vector | string | `[]`

Name of the file with the second mate reads, specified as a character vector or string.

Specify `reads2` as empty (`[]`, `''`, or `""`) if the data consists of single-end reads only.

Example: `'SRR6008575_10k_2.fq'`

Data Types: `char` | `string`

`outputFileName` — Output file name
character vector | string

Output file name, specified as a character vector or string. This file contains the mapping results.

Example: `'SRR6008575_10k_chr4.sam'`

Data Types: `char` | `string`

`options` — Additional options for mapping
`BWAMEMOptions` object | character vector | string

Additional options for mapping, specified as a `BWAMEMOptions` object, character vector, or string. The character vector or string must be in the `bwa mem` native syntax (prefixed by a dash). If you specify a `BWAMEMOptions` object, the software uses only those properties that are set or modified.

Data Types: `char` | `string`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `bwamem(indexbasename,reads1,reads2,outputfile,'BandWidth',90)` sets 90 as the maximum allowable gap.

`AlternativeHitsThreshold` — Threshold for determining which hits receive XA tag in output SAM file
`[5 200]` (default) | nonnegative integer | two-element numeric vector

Threshold for determining which hits receive an XA tag in the output SAM file, specified as a nonnegative integer *n* or two-element numeric vector `[n m]`, where *n* and *m* must be nonnegative integers.

If a read has less than *n* hits with a score greater than 80% of the best score for that read, all hits receive an XA tag in the output SAM file.

When you also specify *m*, the software returns up to *m* hits if the hit list contains a hit to an ALT contig.

Data Types: `double`

`AppendReadCommentsToSAM` — Flag to append FASTA or FASTQ comments to output SAM file
`false` (default) | `true`

Flag to append FASTA or FASTQ comments to the output SAM file, specified as `true` or `false`. The comments appear as text after a space in the file header.

Data Types: `logical`

`BandWidth` — Maximum allowable gap length
`100` (default) | nonnegative integer

Maximum allowable gap length, specified as a nonnegative integer.

Data Types: `double`

`BasesPerBatch` — Number of bases per batch
`[]` (default) | positive integer

Number of bases per batch, specified as a positive integer.

If you do not specify `BasesPerBatch`, the software uses `1e7 * NumThreads` by default. `NumThreads` is the number of parallel threads available when you run `bwamem`.

If you specify `BasesPerBatch`, the software uses that exact number and does not multiply the number by `NumThreads`. This rule applies regardless of whether you explicitly set `NumThreads` or not.

However, if you specify `NumThreads` but not `BasesPerBatch`, the software uses `1e7 * NumThreads`.

The batch size is proportional to the number of parallel threads in use. Using different numbers of threads might produce different outputs. Specifying this option helps with the reproducibility of results.

Data Types: `double`

`ClipPenalty` — Penalty for clipped alignments
`[5 5]` (default) | nonnegative integer | two-element numeric vector

Penalty for clipped alignments, specified as a nonnegative integer or two-element numeric vector. Each read has the best score for an alignment that spans the length of the read. The software does not clip alignments that do not span the length of the read and do not score higher than the sum of `ClipPenalty` and the best score of the full-length read.

Specify a nonnegative integer to set the same penalty for both `5'` and `3'` clipping.

Specify a two-element numeric vector to set different penalties for `5'` and `3'` clipping.

Data Types: `double`

`DropChainFraction` — Threshold for dropping chains relative to longest overlapping chain
`0.5` (default) | scalar between `0` and `1`

Threshold for dropping chains relative to the longest overlapping chain, specified as a scalar between `0` and `1`.

The software drops chains that are shorter than `DropChainFraction * (longest overlapping chain length)`.

Data Types: `double`

`DropChainLength` — Minimum number of bases
`0` (default) | nonnegative integer

Minimum number of bases in seeds forming a chain, specified as a nonnegative integer. The software drops chains shorter than `DropChainLength`.

Data Types: `double`

`ExtraCommand` — Additional commands
`""` (default) | character vector | string

Additional commands, specified as a character vector or string.

The commands must be in the native syntax (prefixed by one or two dashes). Use this option to apply undocumented flags and flags without corresponding MATLAB properties.

Example: `'ExtraCommand','-y'`

Data Types: `char` | `string`

`FastaHeaderToXR` — Flag to include FASTA header in XR tag
`false` (default) | `true`

Flag to include the FASTA header in the XR tag, specified as `true` or `false`.

Data Types: `logical`

`GapExtensionPenalty` — Gap extension penalty
`[1 1]` (default) | nonnegative integer | two-element numeric vector

Gap extension penalty, specified as a nonnegative integer or two-element numeric vector `[n m]`. *n* is the penalty for extending a deletion. *m* is the penalty for extending an insertion.

If you specify a nonnegative integer, the software uses it as the penalty for extending a deletion or an insertion.

Data Types: `double`

`GapOpenPenalty` — Gap opening penalty
`[6 6]` (default) | nonnegative integer | two-element numeric vector

Gap opening penalty, specified as a nonnegative integer or two-element numeric vector `[n m]`. *n* is the penalty for opening a deletion. *m* is the penalty for opening an insertion.

If you specify a nonnegative integer, the software uses it as the penalty for opening a deletion or an insertion.

Data Types: `double`

`HeaderInsert` — Text to insert into header of output SAM file
`[0x0 string]` (default) | character vector | string

Text to insert into the header of the output SAM file, specified as a character vector or string.

Use one of the following:

- Character vector or string that starts with `@` to insert the exact text to the SAM header
- Character vector or string that is a file name, where each line in the file must start with `@`

Data Types: `char` | `string`

`IncludeAll` — Flag to apply all available options
`false` (default) | `true`

Flag to include all available options with the corresponding default values when converting to the original syntax, specified as `true` or `false`.

The original (native) syntax is prefixed by one or two dashes. By default, the function converts only the specified options. If the value is `true`, the software converts all available options, with default values for unspecified options, to the original syntax.

Note If you set `IncludeAll` to `true`, the software translates all available properties, with default values for unspecified properties. The only exception is that when the default value of a property is `NaN`, `Inf`, `[]`, `''`, or `""`, then the software does not translate the corresponding property.

Data Types: `logical`

`InsertSizeStatistics` — Insert size distribution parameters
`[1x0 double]` (default) | four-element numeric array

Insert size distribution parameters, specified as a four-element numeric array `[`*`mean std max min`*`]`.

- *mean* is the mean insert size.
- *std* is the standard deviation.
- *max* is the maximum insert size.
- *min* is the minimum insert size.

If you specify *n* elements array, where *n* is less than four, the elements specify the first *n* distribution parameters. By default, the software infers unspecified parameters from data.

Data Types: `double`

`MarkShortSplitsSecond` — Flag to mark shorter split hits as secondary
`false` (default) | `true`

Flag to mark the shorter split hits as secondary in the SAM flag, specified as `true` or `false`.

Data Types: `logical`

`MarkSmallestCoordinatePrimary` — Flag to mark segment with smallest coordinates as primary
`false` (default) | `true`

Flag to mark the segment with the smallest coordinates as primary when the alignment is split, specified as `true` or `false`.

Data Types: `logical`

`MatchScore` — Score for sequence match
`1` (default) | nonnegative integer

Score for a sequence match, specified as a nonegative integer.

Data Types: `double`

`MaxMemOccurrence` — Maximum number of MEM occurrences
`500` (default) | positive integer

Maximum number of MEM (maximal exact match) occurrences for each read before it is discarded, specified as a positive integer.

Data Types: `double`

`MaxRoundsMateRescue` — Maximum number of rounds of mate rescue
`50` (default) | nonnegative integer

Maximum number of rounds of mate rescue for each read, specified as a nonnegative integer. The software uses the Smith-Waterman (SW) algorithm for the mate rescue.

Data Types: `double`

`MinSeedLength` — Minimum seed length
`19` (default) | positive integer

Minimum seed length, specified as a positive integer. The software discards any matches shorter than the minimum seed length.

Data Types: `double`

`MismatchPenalty` — Penalty for alignment mismatch
`4` (default) | nonnegative integer

Penalty for an alignment mismatch, specified as a nonnegative integer.

Data Types: `double`

`NumThreads` — Number of parallel threads
`1` (default) | positive integer

Number of parallel threads to use, specified as a positive integer. Threads are run on separate processors or cores. Increasing the number of threads generally improves the runtime significantly, but increases the memory footprint.

Data Types: `double`

`OutputAllAlignments` — Flag to return all found alignments
`false` (default) | `true`

Flag to return all found alignments including unpaired and paired-end reads, specified as `true` or `false`. If the value is `true`, the software returns all found alignments and marks them as secondary alignments.

Data Types: `logical`

`OutputScoreThreshold` — Score threshold for returning alignments
`30` (default) | positive integer

Score threshold for returning alignments, specified as a positive integer. Specify the minimum score that alignments must have to be in the output file.

Data Types: `double`

`ReadGroupLine` — Text to insert into read group header
`[0x0 string]` (default) | character vector | string

Text to insert into the read group (RG) header line in the output file, specified as a character vector or string.

Data Types: `char` | `string`

`ReadType` — Type of reads to align
`[0x0 string]` (default) | `'pacbio` | `'ont2d` | `'intractg'`

Type of reads to align, specified as a character vector or string. Each read type has different default parameter values to use during alignment. You can overwrite any parameters. Valid options are:

- `'pacbio'` — PacBio reads
- `'ont2d'` — Oxford nanopore 2D reads

- 'intractg' — Intra-species contigs

The parameter values are as follows.

'pacbio'	• MinSeedLength = 17 • DropChainLength = 40 • SeedSplitRatio = 10 • MatchScore = 1 • MismatchPenalty = 1 • GapOpenPenalty = 1 • GapExtensionPenalty = 1 • ClipPenalty = 0 The equivalent native syntax is '-k17 -W40 -r10 -A1 -B1 -O1 -E1 -L0'.
'ont2d'	• MinSeedLength = 14 • DropChainLength = 20 • SeedSplitRatio = 10 • MatchScore = 1 • MismatchPenalty = 1 • GapOpenPenalty = 1 • GapExtensionPenalty = 1 • ClipPenalty = 0 The equivalent native syntax is '-k14 -W20 -r10 -A1 -B1 -O1 -E1 -L0'.
'intractg'	• MismatchPenalty = 9 • GapOpenPenalty = 16 • ClipPenalty = 5 The equivalent native syntax is '-B9 -O16 -L5'.

Data Types: char | string

ReduceSupplementaryMAPQ — Flag to reduce mapping quality (MAPQ) score of supplementary alignments
true (default) | false

Flag to reduce the mapping quality (MAPQ) score of supplementary alignments, specified as true or false.

Data Types: logical

SeedSplitRatio — Threshold for reseeding
1.50 (default) | nonnegative integer

Threshold for reseeding, specified as a nonnegative integer. Specify the seed length at which reseeding happens relative to the minimum seed length MinSeedLength. Specifically, if a MEM (maximal exact match) is longer than MinSeedLength * SeedSplitRatio, reseeding occurs.

Data Types: double

`SkipMateRescue` — Flag to skip mate rescue
`false` (default) | `true`

Flag to skip mate rescue, specified as `true` or `false`. Mate rescue uses the Smith-Waterman (SW) algorithm to align unmapped reads with mates that are properly aligned.

Data Types: `logical`

`SkipPairing` — Flag to skip read pairing
`false` (default) | `true`

Flag to skip read pairing, specified as `true` or `false`. If `true`, for paired-end reads, the software uses the Smith-Waterman (SW) algorithm to rescue missing hits only and does not try to find hits that fit a proper pair.

Data Types: `logical`

`SmartPairing` — Flag to perform smart pairing
`false` (default) | `true`

Flag to perform smart pairing, specified as `true` or `false`. If the value is `true`, the software pairs adjacent reads that are in the same file and have the same name. Such FASTQ files are also known as interleaved files.

Data Types: `logical`

`SoftClipSupplementary` — Flag to soft clip supplemental alignments
`false` (default) | `true`

Flag to soft clip supplemental alignments, specified as `true` or `false`. If the value is `true`, the software soft clips both supplemental alignments and a primary alignment.

The default value is `false`, which means that the software soft clips the primary alignment and hard clips the supplemental alignments.

Data Types: `logical`

`TreatAltAsPrimary` — Flag to treat ALT contigs as part of primary assembly
`false` (default) | `true`

Flag to treat ALT contigs as part of the primary assembly, specified as `true` or `false`.

Data Types: `logical`

`UnpairedReadPenalty` — Penalty for mapping read pairs as unpaired
`17` (default) | nonnegative integer

Penalty for mapping read pairs as unpaired, specified as a nonnegative integer.

The alignment score for a paired read pair is `read1 score + read2 score - insert penalty`. The alignment score for an unpaired read pair is `read1 score + read2 score - UnpairedReadPenalty`. The software compares these two scores to force read pairing. A larger `UnpairedReadPenalty` value leads to a more aggressive read pairing.

Data Types: `double`

`Verbosity` — Verbosity level of information printed
`0` (default) | nonnegative integer

Verbosity level of information printed to the MATLAB command line while the software is running, specified as a nonnegative integer. Valid options are:

- 0 — For disabling all outputs to the command line.
- 1 — For printing error messages.
- 2 — For printing warning and error messages.
- 3 — For printing all messages.
- 4 — For debugging purposes only.

Data Types: `double`

`ZDropOff` — Cutoff for Smith-Waterman extension
`100` (default) | nonnegative integer

Cutoff for the Smith-Waterman (SW) extension, specified as a nonnegative integer. The software uses the following expression:

$|i - j| * MatchScore + ZDropOff$, where i and j are the current positions of the query and reference, respectively. When the difference between the best score and current extension score is larger than this expression value, the software terminates the SW extension.

Data Types: `double`

Version History

Introduced in R2020b

References

[1] Li, Heng, and Richard Durbin. “Fast and Accurate Short Read Alignment with Burrows-Wheeler Transform.” *Bioinformatics* 25, no. 14 (July 15, 2009): 1754–60. https://doi.org/10.1093/bioinformatics/btp324.

[2] Li, Heng, and Richard Durbin. “Fast and Accurate Long-Read Alignment with Burrows–Wheeler Transform.” *Bioinformatics* 26, no. 5 (March 1, 2010): 589–95. https://doi.org/10.1093/bioinformatics/btp698.

See Also

`BWAMEMOptions` | `bwaindex`

Topics
“Bioinformatics Toolbox Software Support Packages”

BWAMEMOptions

Option set for `bwamem`

Description

A `BWAMEMOptions` object contains options for the `bwamem` function, which aligns sequence reads to a reference genome [1][2].

Creation

Syntax

```
bwamemOpt = BWAMEMOptions
bwamemOpt = BWAMEMOptions(Name,Value)
bwamemOpt = BWAMEMOptions(S)
```

Description

`bwamemOpt = BWAMEMOptions` creates a `BWAMEMOptions` object with the default property values.

`BWAMEMOptions` requires the BWA Support Package for Bioinformatics Toolbox. If the support package is not installed, then the function provides a download link. For details, see “Bioinformatics Toolbox Software Support Packages”.

`bwamemOpt = BWAMEMOptions(Name,Value)` sets the object properties on page 1-482 using one or more name-value pair arguments. Enclose each property name in quotes. For example, `bwamemOpt = BWAMEMOptions('BandWidth',90)` sets the maximum allowable gap length to 90.

`bwamemOpt = BWAMEMOptions(S)` specifies optional parameters using a string or character vector `S`.

Input Arguments

S — bwamem options
character vector | string

`bwamem` options, specified as a character vector or string. `S` must be in the `bwa mem` option syntax (prefixed by one or two dashes).

Example: `'-k14 -W20 -r10'`

Properties

`AlternativeHitsThreshold` — Threshold for determining which hits receive XA tag in output SAM file
`[5 200]` (default) | nonnegative integer | two-element numeric vector

Threshold for determining which hits receive an XA tag in the output SAM file, specified as a nonnegative integer *n* or two-element numeric vector `[n m]`, where *n* and *m* must be nonnegative integers.

If a read has less than *n* hits with a score greater than 80% of the best score for that read, all hits receive an XA tag in the output SAM file.

When you also specify *m*, the software returns up to *m* hits if the hit list contains a hit to an ALT contig.

Data Types: `double`

`AppendReadCommentsToSAM` — Flag to append FASTA or FASTQ comments to output SAM file
`false` (default) | `true`

Flag to append FASTA or FASTQ comments to the output SAM file, specified as `true` or `false`. The comments appear as text after a space in the file header.

Data Types: `logical`

`BandWidth` — Maximum allowable gap length
`100` (default) | nonnegative integer

Maximum allowable gap length, specified as a nonnegative integer.

Data Types: `double`

`BasesPerBatch` — Number of bases per batch
`[]` (default) | positive integer

Number of bases per batch, specified as a positive integer.

If you do not specify `BasesPerBatch`, the software uses `1e7 * NumThreads` by default. `NumThreads` is the number of parallel threads available when you run `bwamem`.

If you specify `BasesPerBatch`, the software uses that exact number and does not multiply the number by `NumThreads`. This rule applies regardless of whether you explicitly set `NumThreads` or not.

However, if you specify `NumThreads` but not `BasesPerBatch`, the software uses `1e7 * NumThreads`.

The batch size is proportional to the number of parallel threads in use. Using different numbers of threads might produce different outputs. Specifying this option helps with the reproducibility of results.

Data Types: `double`

`ClipPenalty` — Penalty for clipped alignments
`[5 5]` (default) | nonnegative integer | two-element numeric vector

Penalty for clipped alignments, specified as a nonnegative integer or two-element numeric vector. Each read has the best score for an alignment that spans the length of the read. The software does not clip alignments that do not span the length of the read and do not score higher than the sum of `ClipPenalty` and the best score of the full-length read.

Specify a nonnegative integer to set the same penalty for both `5'` and `3'` clipping.

Specify a two-element numeric vector to set different penalties for `5'` and `3'` clipping.

Data Types: `double`

`DropChainFraction` — Threshold for dropping chains relative to longest overlapping chain
`0.5` (default) | scalar between `0` and `1`

Threshold for dropping chains relative to the longest overlapping chain, specified as a scalar between `0` and `1`.

The software drops chains that are shorter than `DropChainFraction * (longest overlapping chain length)`.

Data Types: `double`

`DropChainLength` — Minimum number of bases
`0` (default) | nonnegative integer

Minimum number of bases in seeds forming a chain, specified as a nonnegative integer. The software drops chains shorter than `DropChainLength`.

Data Types: `double`

`ExtraCommand` — Additional commands
`""` (default) | character vector | string

Additional commands, specified as a character vector or string.

The commands must be in the native syntax (prefixed by one or two dashes). Use this option to apply undocumented flags and flags without corresponding MATLAB properties.

When the software converts the original flags to MATLAB properties, it stores any unrecognized flags in this property.

Example: `'-y'`

Data Types: `char` | `string`

`FastaHeaderToXR` — Flag to include FASTA header in XR tag
`false` (default) | `true`

Flag to include the FASTA header in the XR tag, specified as `true` or `false`.

Data Types: `logical`

`GapExtensionPenalty` — Gap extension penalty
`[1 1]` (default) | nonnegative integer | two-element numeric vector

Gap extension penalty, specified as a nonnegative integer or two-element numeric vector `[`*n m*`]`. *n* is the penalty for extending a deletion. *m* is the penalty for extending an insertion.

If you specify a nonnegative integer, the software uses it as the penalty for extending a deletion or an insertion.

Data Types: `double`

`GapOpenPenalty` — Gap opening penalty
`[6 6]` (default) | nonnegative integer | two-element numeric vector

Gap opening penalty, specified as a nonnegative integer or two-element numeric vector `[n m]`. *n* is the penalty for opening a deletion. *m* is the penalty for opening an insertion.

If you specify a nonnegative integer, the software uses it as the penalty for opening a deletion or an insertion.

Data Types: `double`

`HeaderInsert` — Text to insert into header of output SAM file
`[0x0 string]` (default) | character vector | string

Text to insert into the header of the output SAM file, specified as a character vector or string.

Use one of the following:

- Character vector or string that starts with `@` to insert the exact text to the SAM header
- Character vector or string that is a file name, where each line in the file must start with `@`

Data Types: `char` | `string`

`IncludeAll` — Flag to use all object properties
`false` (default) | true

Flag to include all the object properties with the corresponding default values when converting to the original options syntax, specified as `true` or `false`. You can convert the properties to the original syntax prefixed by one or two dashes (such as `'-d 100 -e 80'`) by using `getCommand`. The default value `false` means that when you call `getCommand(optionsObject)`, it converts only the specified properties. If the value is `true`, `getCommand` converts all available properties, with default values for unspecified properties, to the original syntax.

Note If you set `IncludeAll` to `true`, the software translates all available properties, with default values for unspecified properties. The only exception is that when the default value of a property is `NaN`, `Inf`, `[]`, `''`, or `""`, then the software does not translate the corresponding property.

Example: `true`

Data Types: `logical`

`InsertSizeStatistics` — Insert size distribution parameters
`[1x0 double]` (default) | four-element numeric array

Insert size distribution parameters, specified as a four-element numeric array `[mean std max min]`.

- *mean* is the mean insert size.
- *std* is the standard deviation.
- *max* is the maximum insert size.
- *min* is the minimum insert size.

If you specify *n* elements array, where *n* is less than four, the elements specify the first *n* distribution parameters. By default, the software infers unspecified parameters from data.

Data Types: `double`

`MarkShortSplitsSecond` — Flag to mark shorter split hits as secondary
`false` (default) | `true`

Flag to mark the shorter split hits as secondary in the SAM flag, specified as `true` or `false`.

Data Types: `logical`

`MarkSmallestCoordinatePrimary` — Flag to mark segment with smallest coordinates as primary
`false` (default) | `true`

Flag to mark the segment with the smallest coordinates as primary when the alignment is split, specified as `true` or `false`.

Data Types: `logical`

`MatchScore` — Score for sequence match
`1` (default) | nonnegative integer

Score for a sequence match, specified as a nonegative integer.

Data Types: `double`

`MaxMemOccurrence` — Maximum number of MEM occurrences
`500` (default) | positive integer

Maximum number of MEM (maximal exact match) occurrences for each read before it is discarded, specified as a positive integer.

Data Types: `double`

`MaxRoundsMateRescue` — Maximum number of rounds of mate rescue
`50` (default) | nonnegative integer

Maximum number of rounds of mate rescue for each read, specified as a nonnegative integer. The software uses the Smith-Waterman (SW) algorithm for the mate rescue.

Data Types: `double`

`MinSeedLength` — Minimum seed length
`19` (default) | positive integer

Minimum seed length, specified as a positive integer. The software discards any matches shorter than the minimum seed length.

Data Types: `double`

`MismatchPenalty` — Penalty for alignment mismatch
`4` (default) | nonnegative integer

Penalty for an alignment mismatch, specified as a nonnegative integer.

Data Types: `double`

`NumThreads` — Number of parallel threads to use
`1` (default) | positive integer

Number of parallel threads to use, specified as a positive integer. Threads are run on separate processors or cores. Increasing the number of threads generally improves the runtime significantly, but increases the memory footprint.

Data Types: `double`

`OutputAllAlignments` — Flag to return all found alignments
`false` (default) | `true`

Flag to return all found alignments including unpaired and paired-end reads, specified as `true` or `false`. If the value is `true`, the software returns all found alignments and marks them as secondary alignments.

Data Types: `logical`

`OutputScoreThreshold` — Score threshold for returning alignments
`30` (default) | positive integer

Score threshold for returning alignments, specified as a positive integer. Specify the minimum score that alignments must have to be in the output file.

Data Types: `double`

`ReadGroupLine` — Text to insert into read group header
`[0x0 string]` (default) | character vector | string

Text to insert into the read group (RG) header line in the output file, specified as a character vector or string.

Data Types: `char` | `string`

`ReadType` — Type of reads to align
`[0x0 string]` (default) | `'pacbio` | `'ont2d` | `'intractg'`

Type of reads to align, specified as a character vector or string. Each read type has different default parameter values to use during alignment. You can overwrite any parameters. Valid options are:

- `'pacbio'` — PacBio reads
- `'ont2d'` — Oxford nanopore 2D reads
- `'intractg'` — Intra-species contigs

The parameter values are as follows.

'pacbio'	• MinSeedLength = 17 • DropChainLength = 40 • SeedSplitRatio = 10 • MatchScore = 1 • MismatchPenalty = 1 • GapOpenPenalty = 1 • GapExtensionPenalty = 1 • ClipPenalty = 0 The equivalent native syntax is '-k17 -W40 -r10 -A1 -B1 -O1 -E1 -L0'.
'ont2d'	• MinSeedLength = 14 • DropChainLength = 20 • SeedSplitRatio = 10 • MatchScore = 1 • MismatchPenalty = 1 • GapOpenPenalty = 1 • GapExtensionPenalty = 1 • ClipPenalty = 0 The equivalent native syntax is '-k14 -W20 -r10 -A1 -B1 -O1 -E1 -L0'.
'intractg'	• MismatchPenalty = 9 • GapOpenPenalty = 16 • ClipPenalty = 5 The equivalent native syntax is '-B9 -O16 -L5'.

Data Types: char | string

ReduceSupplementaryMAPQ — Flag to reduce mapping quality (MAPQ) score of supplementary alignments
true (default) | false

Flag to reduce the mapping quality (MAPQ) score of supplementary alignments, specified as true or false.

Data Types: logical

SeedSplitRatio — Threshold for reseeding
1.50 (default) | nonnegative integer

Threshold for reseeding, specified as a nonnegative integer. Specify the seed length at which reseeding happens relative to the minimum seed length MinSeedLength. Specifically, if a MEM (maximal exact match) is longer than MinSeedLength * SeedSplitRatio, reseeding occurs.

Data Types: double

SkipMateRescue — Flag to skip mate rescue
false (default) | true

Flag to skip mate rescue, specified as `true` or `false`. Mate rescue uses the Smith-Waterman (SW) algorithm to align unmapped reads with mates that are properly aligned.

Data Types: `logical`

`SkipPairing` — Flag to skip read pairing
`false` (default) | `true`

Flag to skip read pairing, specified as `true` or `false`. If `true`, for paired-end reads, the software uses the Smith-Waterman (SW) algorithm to rescue missing hits only and does not try to find hits that fit a proper pair.

Data Types: `logical`

`SmartPairing` — Flag to perform smart pairing
`false` (default) | `true`

Flag to perform smart pairing, specified as `true` or `false`. If the value is `true`, the software pairs adjacent reads that are in the same file and have the same name. Such FASTQ files are also known as interleaved files.

Data Types: `logical`

`SoftClipSupplementary` — Flag to soft clip supplemental alignments
`false` (default) | `true`

Flag to soft clip supplemental alignments, specified as `true` or `false`. If the value is `true`, the software soft clips both supplemental alignments and a primary alignment.

The default value is `false`, which means that the software soft clips the primary alignment and hard clips the supplemental alignments.

Data Types: `logical`

`TreatAltAsPrimary` — Flag to treat ALT contigs as part of primary assembly
`false` (default) | `true`

Flag to treat ALT contigs as part of the primary assembly, specified as `true` or `false`.

Data Types: `logical`

`UnpairedReadPenalty` — Penalty for mapping read pairs as unpaired
`17` (default) | nonnegative integer

Penalty for mapping read pairs as unpaired, specified as a nonnegative integer.

The alignment score for a paired read pair is `read1 score + read2 score - insert penalty`. The alignment score for an unpaired read pair is `read1 score + read2 score - UnpairedReadPenalty`. The software compares these two scores to force read pairing. A larger `UnpairedReadPenalty` value leads to a more aggressive read pairing.

Data Types: `double`

`Verbosity` — Verbosity level of information printed
`0` (default) | nonnegative integer

Verbosity level of information printed to the MATLAB command line while the software is running, specified as a nonnegative integer. Valid options are:

- 0 — For disabling all outputs to the command line.
- 1 — For printing error messages.
- 2 — For printing warning and error messages.
- 3 — For printing all messages.
- 4 — For debugging purposes only.

Data Types: `double`

`Version` — Supported version
string

This property is read-only.

Supported version of the original `bwa` software, returned as a string.

Example: `"0.7.17"`

Data Types: `string`

`ZDropOff` — Cutoff for Smith-Waterman extension
`100` (default) | nonnegative integer

Cutoff for the Smith-Waterman (SW) extension, specified as a nonnegative integer. The software uses the following expression:

$|i - j| * MatchScore + ZDropOff$, where i and j are the current positions of the query and reference, respectively. When the difference between the best score and current extension score is larger than this expression value, the software terminates the SW extension.

Data Types: `double`

Object Functions

getCommand	Translate object properties to original options syntax
getOptionsTable	Return table with all properties and equivalent options in original syntax

Examples

Align Reads to Reference Sequence Using BWA

This example requires the BWA Support Package for Bioinformatics Toolbox™. If the support package is not installed, the software provides a download link. For details, see “Bioinformatics Toolbox Software Support Packages”.

Build a set of index files for the Drosophila genome. This example uses the reference sequence `Dmel_chr4.fa`, provided with the toolbox. The `'Prefix'` argument lets you define the prefix of the output index files. You can also include the file path information. For this example, define the prefix as `Dmel_chr4` and save the index files in the current directory.

```
bwaindex('Dmel_chr4.fa','Prefix','./Dmel_chr4');
```

As an alternative to specifying name-value pair arguments, you can use the `BWAIndexOptions` object to specify the indexing options.

```
indexOpt = BWAIndexOptions;
indexOpt.Prefix = './Dmel_chr4';
indexOpt.Algorithm = 'bwtsw';
bwaindex('Dmel_chr4.fa',indexOpt);
```

Once the index files are ready, map the read sequences to the reference using `bwamem`. Two pair-end read input files are already provided with the toolbox. Using name-value pair arguments, you can specify different alignment options, such as the number of parallel threads to use.

```
bwamem('Dmel_chr4','SRR6008575_10k_1.fq','SRR6008575_10k_2.fq','SRR6008575_10k_chr4.sam','NumThre
```

Alternatively, you can use `BWAMEMoptions` to specify the alignment options.

```
alignOpt = BWAMEMOptions;
alignOpt.NumThreads = 4;
bwamem('Dmel_chr4','SRR6008575_10k_1.fq','SRR6008575_10k_2.fq','SRR6008575_10k_chr4.sam',alignOp
```

Version History

Introduced in R2020b

References

[1] Li, Heng, and Richard Durbin. “Fast and Accurate Short Read Alignment with Burrows-Wheeler Transform.” *Bioinformatics* 25, no. 14 (July 15, 2009): 1754–60. https://doi.org/10.1093/bioinformatics/btp324.

[2] Li, Heng, and Richard Durbin. “Fast and Accurate Long-Read Alignment with Burrows–Wheeler Transform.” *Bioinformatics* 26, no. 5 (March 1, 2010): 589–95. https://doi.org/10.1093/bioinformatics/btp698.

See Also

`bwamem` | `cufflinks` | `bwaindex`

Topics

“Bioinformatics Toolbox Software Support Packages”

celintensityread

Read probe intensities from Affymetrix CEL files

Syntax

```
ProbeStructure = celintensityread(CELFiles, CDFFile)

ProbeStructure = celintensityread(..., 'CELPath', CELPathValue, ...)
ProbeStructure = celintensityread(..., 'CDFPath', CDFPathValue, ...)
ProbeStructure = celintensityread(..., 'PMOnly', PMOnlyValue, ...)
ProbeStructure = celintensityread(..., 'Verbose', VerboseValue, ...)
```

Input Arguments

`CELFiles`	Any of the following: • Character vector or string specifying a single CEL file name. • `'*'`, which reads all CEL files in the current folder. • `' '`, which opens the Select CEL Files dialog box from which you select the CEL files. From this dialog box, you can press and hold **Ctrl** or **Shift** while clicking to select multiple CEL files. • Cell array of character vectors or string vector containing CEL file names.
`CDFFile`	Either of the following: • Character vector or string specifying a CDF file name. • `' '`, which opens the Select CDF File dialog box from which you select the CDF file.
`CELPathValue`	Character vector or string specifying the path and folder where the files specified in *`CELFiles`* are stored.
`CDFPathValue`	Character vector or string specifying the path and folder where the file specified in *`CDFFile`* is stored.
`PMOnlyValue`	Property to include or exclude the mismatch (MM) probe intensity values in the returned structure. Enter `true` to return only perfect match (PM) probe intensities. Enter `false` to return both PM and MM probe intensities. Default is `true`.
`VerboseValue`	Controls the display of a progress report showing the name of each CEL file as it is read. When *`VerboseValue`* is `false`, no progress report is displayed. Default is `true`.

Output Arguments

`ProbeStructure`	MATLAB structure containing information from the CEL files, including probe intensities, probe indices, and probe set IDs.

Description

`ProbeStructure` `= celintensityread(`*`CELFiles, CDFFile`*`)` reads the specified Affymetrix CEL files and the associated CDF library file (created from Affymetrix GeneChip arrays for expression or genotyping assays), and then creates *`ProbeStructure`*, a structure containing information from the CEL files, including probe intensities, probe indices, and probe set IDs. *`CELFiles`* is a character vector, string, string vector, or cell array of character vectors containing CEL file names. *`CDFFile`* is a character vector or string specifying a CDF file name.

If you set *`CELFiles`* to `'*'`, then it reads all CEL files in the current folder. If you set *`CELFiles`* to `' '`, then it opens the Select CEL Files dialog box from which you select the CEL files. From this dialog box, you can press and hold **Ctrl** or **Shift** while clicking to select multiple CEL files.

If you set *`CDFFile`* to `' '`, then it opens the Select CDF File dialog box from which you select the CDF file.

`ProbeStructure` `= celintensityread(..., '`*`PropertyName`*`',` *`PropertyValue`*`, ...)` calls `celintensityread` with optional properties that use property name/property value pairs. You can specify one or more properties in any order. Each *`PropertyName`* must be enclosed in single quotation marks and is case insensitive. These property name/property value pairs are as follows:

`ProbeStructure` `= celintensityread(..., 'CELPath',` *`CELPathValue`*`, ...)` specifies a path and folder where the files specified by *`CELFiles`* are stored.

`ProbeStructure` `= celintensityread(..., 'CDFPath',` *`CDFPathValue`*`, ...)` specifies a path and folder where the file specified by *`CDFFile`* is stored.

`ProbeStructure` `= celintensityread(..., 'PMOnly',` *`PMOnlyValue`*`, ...)` includes or excludes the mismatch (MM) probe intensity values. When *`PMOnlyValue`* is `true`, `celintensityread` returns only perfect match (PM) probe intensities. When *`PMOnlyValue`* is `false`, `celintensityread` returns both PM and MM probe intensities. Default is `true`.

Tip Reading a large number of CEL files and/or a large CEL file can require extended amounts of memory from the operating system.

- If you receive errors related to memory, try the following:
  - Increase the virtual memory (swap space) for your operating system as described in "Resolve "Out of Memory" Errors".
- If you receive errors related to Java® heap space, increase your Java heap space:
  - If you have MATLAB version 7.10 (R2010a) or later, see "Java Heap Memory Preferences".
  - If you have MATLAB version 7.9 (R2009b) or earlier, see https://www.mathworks.com/matlabcentral/answers/92813-how-do-i-increase-the-heap-space-for-the-java-vm-in-matlab.

`ProbeStructure` contains the following fields.

Field	Description
`CDFName`	File name of the Affymetrix CDF library file.

Field	Description
`CELNames`	Cell array of names of the Affymetrix CEL files.
`NumChips`	Number of CEL files read into the structure.
`NumProbeSets`	Number of probe sets in each CEL file.
`NumProbes`	Number of probes in each CEL file.
`ProbeSetIDs`	Cell array of the probe set IDs from the Affymetrix CDF library file.
`ProbeIndices`	Column vector containing probe indexing information. Probes within a probe set are numbered `0` through `N - 1`, where `N` is the number of probes in the probe set.
`GroupNumbers`	Column vector containing group numbers for probes within the probe set. For gene expression data, the group number for all probes is `1`. For SNP (genotyping) data, the group numbers for probes are: • `1` — Allele A – (sense) • `2` — Allele B – (sense) • `3` — Allele A + (antisense) • `4` — Allele B + (antisense)
`PMIntensities`	Matrix containing perfect match (PM) probe intensity values. Each row corresponds to a probe, and each column corresponds to a CEL file. The rows are ordered the same way as in `ProbeIndices`, and the columns are ordered the same way as in the *`CELFiles`* input argument.
`MMIntensities` (optional)	Matrix containing mismatch (MM) probe intensity values. Each row corresponds to a probe, and each column corresponds to a CEL file. The rows are ordered the same way as in `ProbeIndices`, and the columns are ordered the same way as in the *`CELFiles`* input argument.

`ProbeStructure` `= celintensityread(..., 'Verbose',` *`VerboseValue`*`, ...)` controls the display of a progress report showing the name of each CEL file as it is read. When *`VerboseValue`* is `false`, no progress report is displayed. Default is `true`.

Examples

The following example assumes that you have the `HG_U95Av2.CDF` library file stored at `D:\Affymetrix\LibFiles\HGGenome`, and that your current folder points to a location containing CEL files associated with this CDF library file. In this example, the `celintensityread` function reads all the CEL files in the current folder and a CDF file in a specified folder. The next command line uses the `rmabackadj` function to perform background adjustment on the PM probe intensities in the `PMIntensities` field of `PMProbeStructure`.

```
PMProbeStructure = celintensityread('*', 'HG_U95Av2.CDF',...
                       'CDFPath', 'D:\Affymetrix\LibFiles\HGGenome');
BackAdjustedMatrix = rmabackadj(PMProbeStructure.PMIntensities);
```

Version History

Introduced in R2006a

See Also

`affygcrma` | `affyinvarsetnorm` | `affyprobeseqread` | `affyread` | `affyrma` | `affysnpintensitysplit` | `agferead` | `gcrma` | `gcrmabackadj` | `gprread` | `ilmnbsread` | `probelibraryinfo` | `probesetlookup` | `probesetplot` | `probesetvalues` | `rmabackadj` | `rmasummary` | `sptread`

Topics

"Preprocessing Affymetrix Microarray Data at the Probe Level"
"Analyzing Affymetrix SNP Arrays for DNA Copy Number Variants"

cghcbs

Perform circular binary segmentation (CBS) on array-based comparative genomic hybridization (aCGH) data

Syntax

```
SegmentStruct = cghcbs(CGHData)

SegmentStruct = cghcbs(CGHData, ...'Alpha', AlphaValue, ...)
SegmentStruct = cghcbs(CGHData, ...'Permutations', PermutationsValue, ...)
SegmentStruct = cghcbs(CGHData, ...'Method', MethodValue, ...)
SegmentStruct = cghcbs(CGHData, ...'StoppingRule', StoppingRuleValue, ...)
SegmentStruct = cghcbs(CGHData, ...'Smooth', SmoothValue, ...)
SegmentStruct = cghcbs(CGHData, ...'Prune', PruneValue, ...)
SegmentStruct = cghcbs(CGHData, ...'Errsum', ErrsumValue, ...)
SegmentStruct = cghcbs(CGHData, ...'WindowSize', WindowSizeValue, ...)
SegmentStruct = cghcbs(CGHData, ...'SampleIndex', SampleIndexValue, ...)
SegmentStruct = cghcbs(CGHData, ...'Chromosome', ChromosomeValue, ...)
SegmentStruct = cghcbs(CGHData, ...'Showplot', ShowplotValue, ...)
SegmentStruct = cghcbs(CGHData, ...'Verbose', VerboseValue, ...)
```

Input Arguments

CGHData	Array-based comparative genomic hybridization (aCGH) data in either of the following forms: • Structure with the following fields: • `Sample` — Cell array of character vectors or string vector containing the sample names (optional). • `Chromosome` — Vector containing the chromosome numbers on which the clones are located. • `GenomicPosition` — Vector containing the genomic positions (in any unit) to which the clones are mapped. • `Log2Ratio` — Matrix containing $\log_2$ ratio of test to reference signal intensity for each clone. Each row corresponds to a clone, and each column corresponds to a sample. • Matrix in which each row corresponds to a clone. The first column contains the chromosome number, the second column contains the genomic position, and the remaining columns each contain the $\log_2$ ratio of test to reference signal intensity for a sample.
AlphaValue	Scalar that specifies the significance level for the statistical tests to accept change points. Default is `0.01`.
PermutationsValue	Scalar that specifies the number of permutations used for p-value estimation. Default is `10,000`.

MethodValue	Character vector or string that specifies the method to estimate the p-values. Choices are `'Perm'` or `'Hybrid'` (default). `'Perm'` does a full permutation, while `'Hybrid'` uses a faster, tail probability-based permutation. When using the `'Hybrid'` method, the `'Perm'` method is applied automatically when segment data length becomes less than 200.
StoppingRuleValue	Controls the use of a heuristic stopping rule, based on the method described by Venkatraman and Olshen (2007) on page 1-504, to declare a change without performing the full number of permutations for the p-value estimation, whenever it becomes very likely that a change has been detected. Choices are `true` or `false` (default). **Tip** Set this property to `true` to increase processing speed. Set this property to `false` to maximize accuracy.
SmoothValue	Controls the smoothing of outliers before segmenting using the procedure explained by Olshen et al. (2004) on page 1-504. Choices are `true` (default) or `false`.
PruneValue	Controls the elimination of change points identified due to local trends in the data that are not indicative of real copy number change, using the procedure explained by Olshen et al. (2004) on page 1-504. Choices are `true` or `false` (default).
ErrsumValue	Scalar that specifies the allowed proportional increase in the error sum of squares when eliminating change points using the `'Prune'` property. Commonly used values are `0.05` and `0.1`. Default is `0.05`.
WindowSizeValue	Scalar that specifies the size of the window (in data points) used to divide the data when using the `'Perm'` method on large data sets. Default is `200`.
SampleIndexValue	A single sample index or a vector of sample indices that specify the sample(s) to analyze. Default is all sample indices.
ChromosomeValue	A single chromosome number or a vector of chromosome numbers that specify the data to analyze. Default is all chromosome numbers.

`ShowplotValue`	Controls the display of plots of the segment means over the original data. Choices are either: • `true` — All chromosomes in all samples are plotted. If there are multiple samples in *`CGHData`*, then each sample is plotted in a separate Figure window. • `false` — No plot. • `W` — The layout displays all chromosomes in the whole genome in one plot in the Figure window. • `S` — The layout displays each chromosome in a subplot in the Figure window. • *`I`* — An integer specifying only one of the chromosomes in *`CGHData`* to be plotted. Default is: • `false` — When return values are specified. • `true` and `W` — When return values are not specified.
`VerboseValue`	Controls the display of a progress report of the analysis. Choices are `true` (default) or `false`.

Output Arguments

`SegmentStruct`	Structure containing segmentation information in the following fields: • `Sample` — Sample name from *`CGHData`* input argument. If the input argument does not include sample names, then sample names are assigned as `Sample1`, `Sample2`, and so forth. • `SegmentData` — Structure array containing segment data for the sample in the following fields: • `Chromosome` — Chromosome number on which the segment is located. • `Start` — Genomic position at the start of the segment (in the same units as used for the *`CGHData`* input). • `End` — Genomic position at the end of the segment (in the same units as used for the *`CGHData`* input). • `Mean` — Mean value of the $\log_2$ ratio of the test to reference signal intensity for the segment.

Description

`SegmentStruct` `= cghcbs(`*`CGHData`*`)` performs circular binary segmentation (CBS) on array-based comparative genomic hybridization (aCGH) data to determine the copy number alteration segments (neighboring regions of DNA that exhibit a statistical difference in copy number) and change points.

Note The CBS algorithm recursively splits chromosomes into segments based on a maximum t statistic estimated by permutation. This computation can be time consuming. If *n* = number of data points, then computation time ~ O(n^2).

`SegmentStruct = cghcbs(CGHData, ...'PropertyName', PropertyValue, ...)` calls `cghcbs` with optional properties that use property name/property value pairs. You can specify one or more properties in any order. Each *`PropertyName`* must be enclosed in single quotation marks and is case insensitive. These property name/property value pairs are as follows:

`SegmentStruct = cghcbs(CGHData, ...'Alpha', AlphaValue, ...)` specifies the significance level for the statistical tests to accept change points. Default is `0.01`.

`SegmentStruct = cghcbs(CGHData, ...'Permutations', PermutationsValue, ...)` specifies the number of permutations used for p-value estimation. Default is `10,000`.

`SegmentStruct = cghcbs(CGHData, ...'Method', MethodValue, ...)` specifies the method to estimate the p-values. Choices are `'Perm'` or `'Hybrid'` (default). `'Perm'` does a full permutation, while `'Hybrid'` uses a faster, tail probability-based permutation. When using the `'Hybrid'` method, the `'Perm'` method is applied automatically when segment data length becomes less than 200.

`SegmentStruct = cghcbs(CGHData, ...'StoppingRule', StoppingRuleValue, ...)` controls the use of a heuristic stopping rule, based on the method described by Venkatraman and Olshen (2007) on page 1-504, to declare a change without performing the full number of permutations for the p-value estimation, whenever it becomes very likely that a change has been detected. Choices are `true` or `false` (default).

`SegmentStruct = cghcbs(CGHData, ...'Smooth', SmoothValue, ...)` controls the smoothing of outliers before segmenting, using the procedure explained by Olshen et al. (2004) on page 1-504. Choices are `true` (default) or `false`.

`SegmentStruct = cghcbs(CGHData, ...'Prune', PruneValue, ...)` controls the elimination of change points identified due to local trends in the data that are not indicative of real copy number change, using the procedure explained by Olshen et al. (2004) on page 1-504. Choices are `true` or `false` (default).

`SegmentStruct = cghcbs(CGHData, ...'Errsum', ErrsumValue, ...)` specifies the allowed proportional increase in the error sum of squares when eliminating change points using the `'Prune'` property. Commonly used values are `0.05` and `0.1`. Default is `0.05`.

`SegmentStruct = cghcbs(CGHData, ...'WindowSize', WindowSizeValue, ...)` specifies the size of the window (in data points) used to divide the data when using the `'Perm'` method on large data sets. Default is `200`.

`SegmentStruct = cghcbs(CGHData, ...'SampleIndex', SampleIndexValue, ...)` analyzes only the sample(s) specified by *`SampleIndexValue`*, which can be a single sample index or a vector of sample indices. Default is all sample indices.

`SegmentStruct = cghcbs(CGHData, ...'Chromosome', ChromosomeValue, ...)` analyzes only the data on the chromosomes specified by *`ChromosomeValue`*, which can be a single chromosome number or a vector of chromosome numbers. Default is all chromosome numbers.

`SegmentStruct = cghcbs(CGHData, ...'Showplot', ShowplotValue, ...)` controls the display of plots of the segment means over the original data. Choices are `true`, `false`, `W`, `S`, or *`I`*, an integer specifying one of the chromosomes in *`CGHData`*. When *`ShowplotValue`* is `true`, all chromosomes in all samples are plotted. If there are multiple samples in *`CGHData`*, then each sample is plotted in a separate Figure window. When *`ShowplotValue`* is `W`, the layout displays all chromosomes in one plot in the Figure window. When *`ShowplotValue`* is `S`, the layout displays each chromosome in a subplot in the Figure window. When *`ShowplotValue`* is *`I`*, only the specified chromosome is plotted. Default is either:

- `false` — When return values are specified.
- `true` and `W` — When return values are not specified.

`SegmentStruct = cghcbs(CGHData, ...'Verbose', VerboseValue, ...)` controls the display of a progress report of the analysis. Choices are `true` (default) or `false`.

Examples

Perform circular binary segmentation on comparative genomic hybridization data

Analyze data from the Coriell cell line study

Load the array-based CGH (aCGH) data from the Coriell cell line study (Snijders, A. et al., 2001).

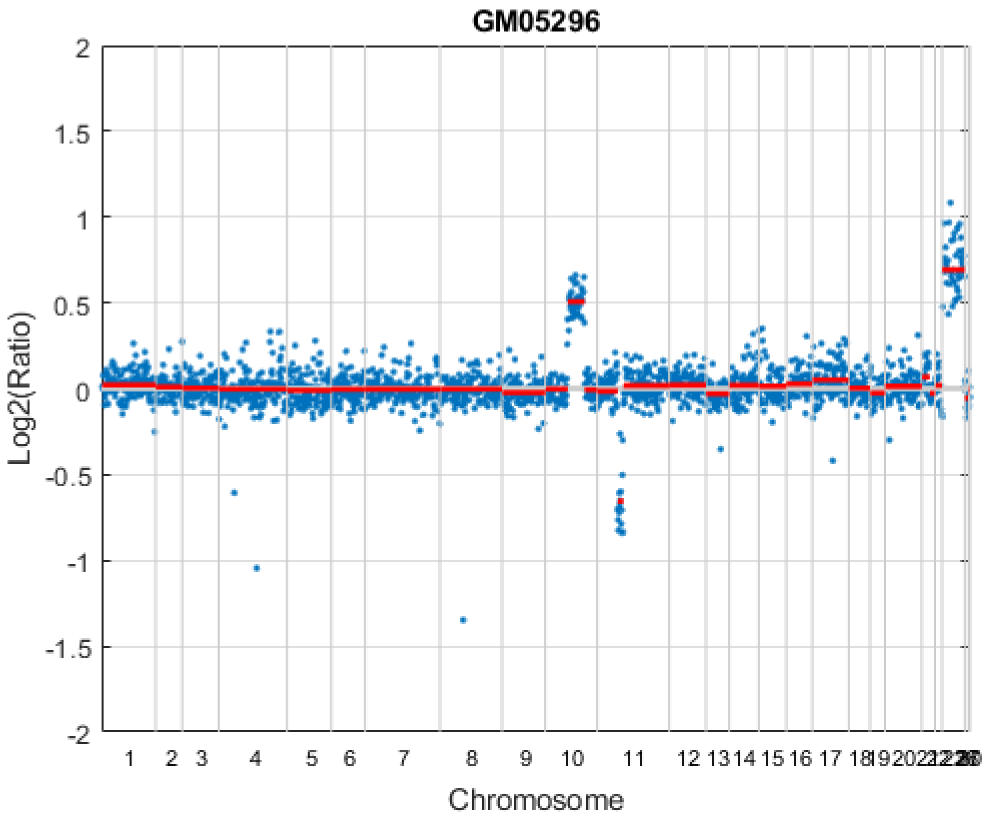
```
load coriell_baccgh
```

Analyze all chromosomes of sample 3 (GM05296) of the aCGH data and return segmentation data in a structure, S. Plot the segment means over the original data for all chromosomes of this sample.

```
S = cghcbs(coriell_data,'sampleindex',3,'showplot',true);
```

```
Analyzing: GM05296. Current chromosome 1
Analyzing: GM05296. Current chromosome 2
Analyzing: GM05296. Current chromosome 3
Analyzing: GM05296. Current chromosome 4
Analyzing: GM05296. Current chromosome 5
Analyzing: GM05296. Current chromosome 6
Analyzing: GM05296. Current chromosome 7
Analyzing: GM05296. Current chromosome 8
Analyzing: GM05296. Current chromosome 9
Analyzing: GM05296. Current chromosome 10
Analyzing: GM05296. Current chromosome 11
Analyzing: GM05296. Current chromosome 12
Analyzing: GM05296. Current chromosome 13
Analyzing: GM05296. Current chromosome 14
Analyzing: GM05296. Current chromosome 15
Analyzing: GM05296. Current chromosome 16
Analyzing: GM05296. Current chromosome 17
Analyzing: GM05296. Current chromosome 18
Analyzing: GM05296. Current chromosome 19
Analyzing: GM05296. Current chromosome 20
Analyzing: GM05296. Current chromosome 21
Analyzing: GM05296. Current chromosome 22
Analyzing: GM05296. Current chromosome 23
Analyzing: GM05296. Current chromosome 26
```

```
Analyzing: GM05296. Current chromosome 27
Analyzing: GM05296. Current chromosome 30
```



Chromosome 10 shows a gain, while chromosome 11 shows a loss.

Display copy number alteration regions aligned to a chromosome ideogram

Create a structure containing segment gain and loss information for chromosomes 10 and 11 from sample 3, making sure the segment data is in bp units. (You can determine copy number variance (CNV) information by exploring S, the structure of segments returned by the cghcbs function. For the `'CNVType'` field, use 1 to indicate a loss and 2 to indicate a gain.

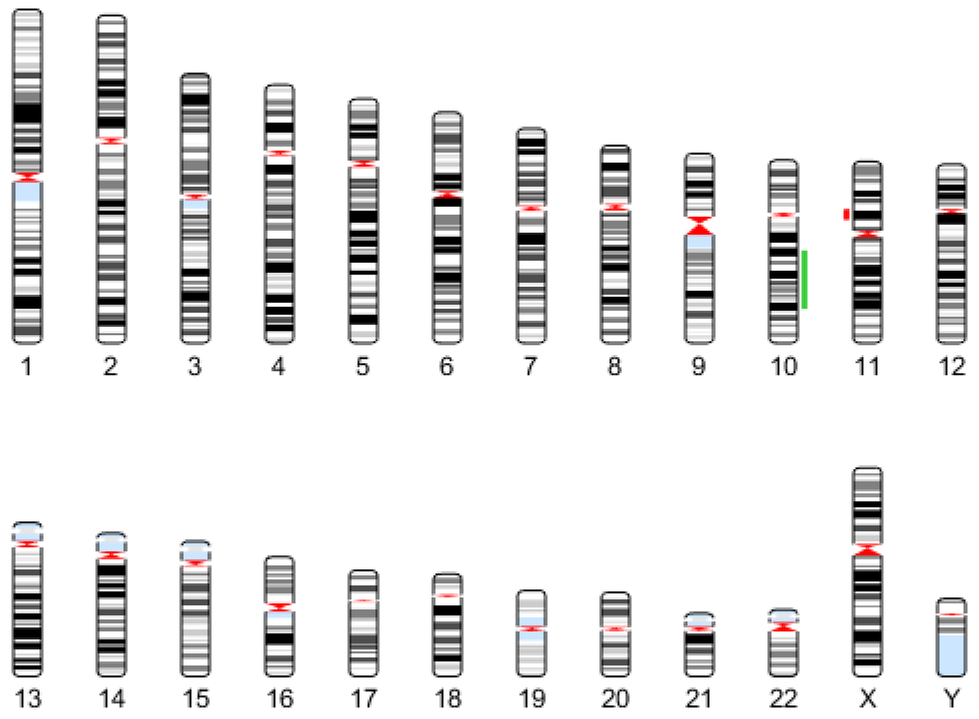
```
cnvStruct = struct('Chromosome', [10 11],...
 'CNVType', [2 1],...
 'Start', [S.SegmentData(10).Start(2),...
  S.SegmentData(11).Start(2)]*1000,...
 'End',   [S.SegmentData(10).End(2),...
  S.SegmentData(11).End(2)]*1000)
```

```
cnvStruct = 

  struct with fields:

    Chromosome: [10 11]
       CNVType: [2 1]
         Start: [66905000 35416000]
```

```
            End: [110412000 43357000]
```

Pass the structure to the `chromosomeplot` function using the `'CNV'` option to display the copy number gains (green) and losses (red) aligned to the human chromosome ideogram. Specify kb units for the display of segment information in the data tip.

```
chromosomeplot('hs_cytoBand.txt', 'CNV', cnvStruct, 'unit', 2)
```



Analyze data from a pancreatic cancer study

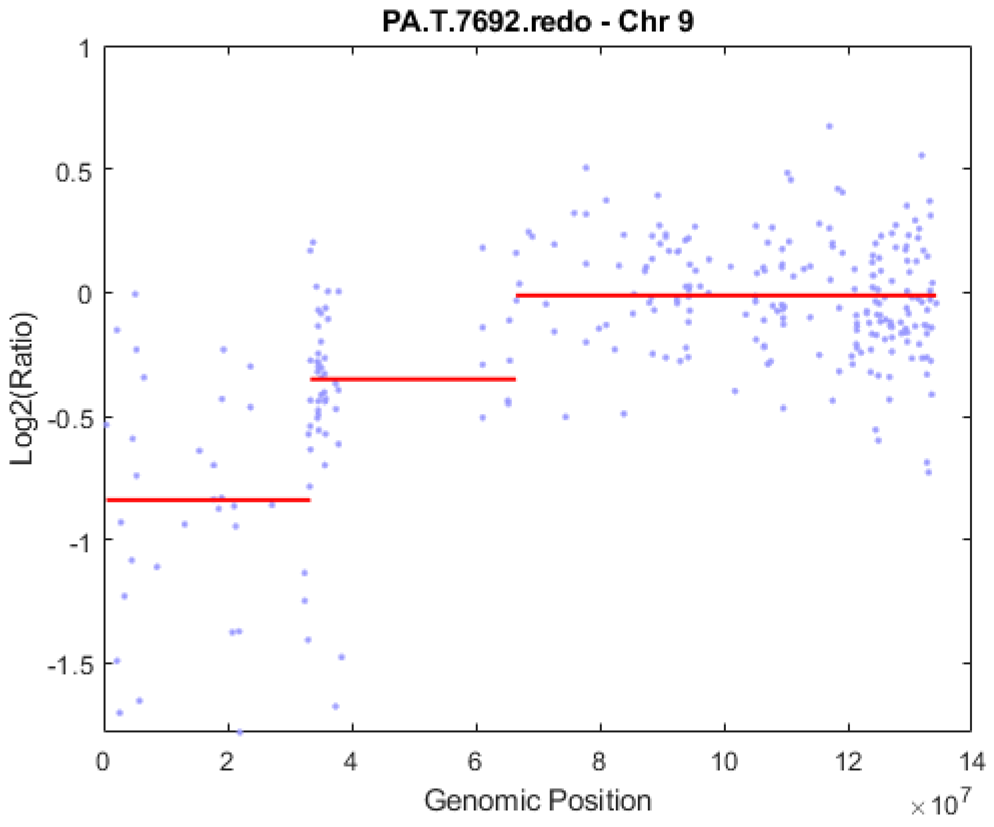
Load the aCGH data from a pancreatic cancer study (Aguirre, A. et al., 2004).

```
load pancrea_oligocgh
```

Analyze only chromosome 9 in sample 32 of the CGH data and return the segmentation data in a structure, PS. Plot the segment means over the original data for chromosome 9 in this sample.

```
PS = cghcbs(pancrea_data,'sampleindex',32,'chromosome',9,...
            'showplot',9);
```

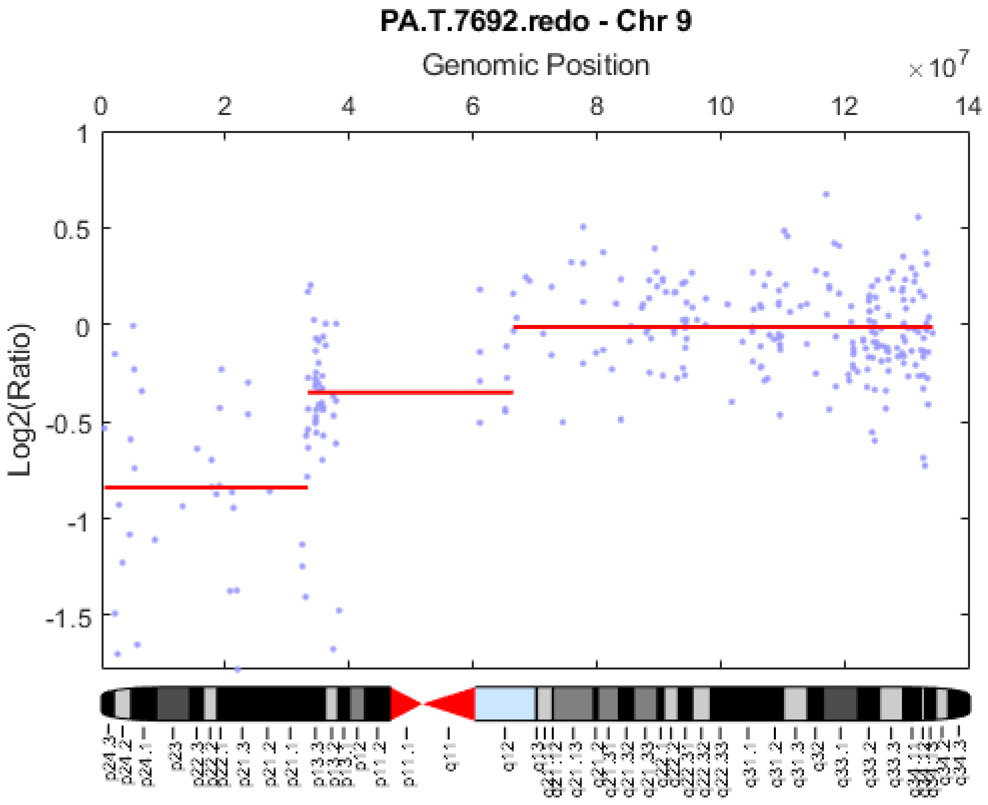
```
Analyzing: PA.T.7692.redo. Current chromosome 9
```



Chromosome 9 contains two segments that indicate losses. For more detailed information on interpreting the data, see Aguirre, A. et al. (2004).

Use the `chromosomeplot` function with the `'addtoplot'` option to add the ideogram of chromosome 9 for Homo sapiens to the plot of the segmentation data.

```
chromosomeplot('hs_cytoBand.txt', 9, 'addtoplot', gca)
```



Version History

Introduced in R2007b

References

[1] Olshen, A.B., Venkatraman, E.S., Lucito, R., and Wigler, M. (2004). Circular binary segmentation for the analysis of array-based DNA copy number data. Biostatistics *5, 4*, 557–572.

[2] Venkatraman, E.S., and Olshen, A.B. (2007). A Faster Circular Binary Segmentation Algorithm for the Analysis of Array CGH Data. Bioinformatics *23(6)*, 657–663.

[3] Venkatraman, E.S., and Olshen, A.B. (2006). DNAcopy: A Package for Analyzing DNA Copy Data. `https://www.bioconductor.org/packages/2.1/bioc/html/DNAcopy.html`

[4] Snijders, A.M., Nowak, N., Segraves, R., Blackwood, S., Brown, N., Conroy, J., Hamilton, G., Hindle, A.K., Huey, B., Kimura, K., Law, S., Myambo, K., Palmer, J., Ylstra, B., Yue, J.P., Gray, J.W., Jain, A.N., Pinkel, D., and Albertson, D.G. (2001). Assembly of microarrays for genome-wide measurement of DNA copy number. Nature Genetics *29*, 263–264.

[5] Aguirre, A.J., Brennan, C., Bailey, G., Sinha, R., Feng, B., Leo, C., Zhang, Y., Zhang, J., Gans, J.D., Bardeesy, N., Cauwels, C., Cordon-Cardo, C., Redston, M.S., DePinho, R.A., and Chin, L. (2004). High-resolution characterization of the pancreatic adenocarcinoma genome. PNAS *101, 24*, 9067–9072.

See Also

`chromosomeplot` | `cytobandread`

cghfreqplot

Display frequency of DNA copy number alterations across multiple samples

Syntax

```
FreqStruct = cghfreqplot(CGHData)

FreqStruct = cghfreqplot(CGHData, ...'Threshold', ThresholdValue, ...)
FreqStruct = cghfreqplot(CGHData, ...'Group', GroupValue, ...)
FreqStruct = cghfreqplot(CGHData, ...'Subgrp', SubgrpValue, ...)
FreqStruct = cghfreqplot(CGHData, ...'Subplot', SubplotValue, ...)
FreqStruct = cghfreqplot(CGHData, ...'Cutoff', CutoffValue, ...)
FreqStruct = cghfreqplot(CGHData, ...'Chromosome', ChromosomeValue, ...)
FreqStruct = cghfreqplot(CGHData, ...'IncludeX', IncludeXValue, ...)
FreqStruct = cghfreqplot(CGHData, ...'IncludeY', IncludeYValue, ...)
FreqStruct = cghfreqplot(CGHData, ...'Chrominfo', ChrominfoValue, ...)
FreqStruct = cghfreqplot(CGHData, ...'ShowCentr', ShowCentrValue, ...)
FreqStruct = cghfreqplot(CGHData, ...'Color', ColorValue, ...)
FreqStruct = cghfreqplot(CGHData, ...'YLim', YLimValue, ...)
FreqStruct = cghfreqplot(CGHData, ...'Titles', TitlesValue, ...)
```

Input Arguments

CGHData	Array-based comparative genomic hybridization (aCGH) data in either of the following forms: • Structure with the following fields: • `Sample` — Cell array of character vectors or string vector containing the sample names (optional). • `Chromosome` — Vector containing the chromosome numbers on which the clones are located. • `GenomicPosition` — Vector containing the genomic positions (in bp, kb, or mb units) to which the clones are mapped. • `Log2Ratio` — Matrix containing $\log_2$ ratio of test to reference signal intensity for each clone. Each row corresponds to a clone, and each column corresponds to a sample. • Matrix in which each row corresponds to a clone. The first column contains the chromosome number, the second column contains the genomic position, and the remaining columns each contain the $\log_2$ ratio of test to reference signal intensity for a sample.

`ThresholdValue`	Positive scalar or vector that specifies the gain/loss threshold. A clone is considered to be a gain if its $\log_2$ ratio is above *`ThresholdValue`*, and a loss if its $\log_2$ ratio is below negative *`ThresholdValue`*. The *`ThresholdValue`* is applied as follows: • If a positive scalar, it is the gain and loss threshold for all the samples. • If a two-element vector, the first element is the gain threshold for all samples, and the second element is the loss threshold for all samples. • If a vector of the same length as the number of samples, each element in the vector is considered as a unique gain and loss threshold for each sample. Default is `0.25`.
`GroupValue`	Specifies the sample groups to calculate the frequency from. Choices are: • A vector of sample column indices (for data with only one group). The samples specified in the vector are considered a group. • A cell array of vectors of sample column indices (for data divided into multiple groups). Each element in the cell array is considered a group. Default is a single group of all the samples in *`CGHData`*.
`SubgrpValue`	Controls the analysis of samples by subgroups. Choices are `true` (default) or `false`.
`SubplotValue`	Controls the display of all plots in one Figure window when more than one subgroup is analyzed. Choices are `true` (default) or `false` (displays plots in separate windows).
`CutoffValue`	Scalar or two-element numeric vector that specifies a cutoff, which controls the plotting of only the clones with frequency gains or losses greater than or equal to *`CutoffValue`*. If a two-element vector, the first element is the cutoff for gains, and the second element is for losses. Default is `0`.
`ChromosomeValue`	Single chromosome number or a vector of chromosome numbers that specify the chromosomes for which to display frequency plots. Default is all chromosomes in *`CGHData`*.
`IncludeXValue`	Controls the inclusion of the X chromosome in the analysis. Choices are `true` (default) or `false`.
`IncludeYValue`	Controls the inclusion of the Y chromosome in the analysis. Choices are `true` or `false` (default) .

`ChrominfoValue`	Cytogenetic banding information specified by either of the following: • Structure returned by the `cytobandread` function • Character vector or string specifying the file name of an NCBI ideogram text file or a UCSC Genome Browser cytoband text file Default is *Homo sapiens* cytogenetic banding information from the UCSC Genome Browser, NCBI Build 36.1 (`https://genome.UCSC.edu`).
`ShowCentrValue`	Controls the display of the centromere positions as vertical dashed lines in the frequency plot. Choices are `true` (default) or `false`. **Tip** The centromere positions are obtained from *`ChrominfoValue`*.
`ColorValue`	Color scheme for the vertical lines in the plot, indicating the frequency of the gains and losses, specified by either of the following: • Name of or handle to a function that returns a colormap • M-by-3 matrix containing RGB values. If M equals 1, then that single color is used for all gains and losses. If M equals 2 or more, then the first row is used for gains, the second row is used for losses, and remaining rows are ignored. For example, `[0 1 0;1 0 0]` specifies green for gain and red for loss. The default color scheme is a range of colors from pure green (gain = 1) through yellow (0) to pure red (loss = –1).
`YLimValue`	Two-element vector specifying the minimum and maximum values on the vertical axis. Default is `[1, -1]`.
`TitlesValue`	Character vector, string, string vector, or a cell array of character vectors that specifies titles for the group(s), which are added to the tops of the plot(s).

Output Arguments

FreqStruct	Structure containing frequency data in the following fields: • `Group` — Structure array, with each structure representing a group of samples. Each structure contains the following fields: • `Sample` — Cell array containing names of samples within the group. • `GainFrequency` — Column vector containing the average gain for each clone for a group of samples. • `LossFrequency` — Column vector containing the average loss for each clone for a group of samples. • `Chromosome` — Column vector containing the chromosome numbers on which the clones are located. • `GenomicPosition` — Column vector containing the genomic positions of the clones. **Tip** You can use this output structure as input to the `cghfreqplot` function.

Description

`*FreqStruct* = cghfreqplot(*CGHData*)` displays the frequency of copy number gain or loss across multiple samples for each clone on an array against their genomic position along the chromosomes.

`*FreqStruct* = cghfreqplot(*CGHData*, ...'*PropertyName*', *PropertyValue*, ...)` calls `cghfreqplot` with optional properties that use property name/property value pairs. You can specify one or more properties in any order. Each *PropertyName* must be enclosed in single quotation marks and is case insensitive. These property name/property value pairs are as follows:

`*FreqStruct* = cghfreqplot(*CGHData*, ...'Threshold', *ThresholdValue*, ...)` specifies the gain/loss threshold. A clone is considered to be a gain if its $\log_2$ ratio is above *ThresholdValue*, and a loss if its $\log_2$ ratio is below negative *ThresholdValue*.

The *ThresholdValue* is applied as follows:

- If a positive scalar, it is the gain and loss threshold for all the samples.
- If a two-element vector, the first element is the gain threshold for all samples, and the second element is the loss threshold for all samples.
- If a vector of the same length as the number of samples, each element in the vector is considered as a unique gain and loss threshold for each sample.

Default is `0.25`.

`*FreqStruct* = cghfreqplot(*CGHData*, ...'Group', *GroupValue*, ...)` specifies the sample groups to calculate the frequency from. Choices are:

- A vector of sample column indices (for data with only one group). The samples specified in the vector are considered a group.

- A cell array of vectors of sample column indices (for data divided into multiple groups). Each element in the cell array is considered a group.

Default is a single group of all the samples in `CGHData`.

`FreqStruct = cghfreqplot(CGHData, ...'Subgrp', SubgrpValue, ...)` controls the analysis of samples by subgroups. Choices are `true` (default) or `false`.

`FreqStruct = cghfreqplot(CGHData, ...'Subplot', SubplotValue, ...)` controls the display of all plots in one Figure window when more than one subgroup is analyzed. Choices are `true` (default) or `false` (displays plots in separate windows).

`FreqStruct = cghfreqplot(CGHData, ...'Cutoff', CutoffValue, ...)` specifies a cutoff value, which controls the plotting of only the clones with frequency gains or losses greater than or equal to `CutoffValue`. `CutoffValue` is a scalar or two-element numeric vector. If a two-element numeric vector, the first element is the cutoff for gains, and the second element is for losses. Default is `0`.

`FreqStruct = cghfreqplot(CGHData, ...'Chromosome', ChromosomeValue, ...)` displays the frequency plots only of chromosome(s) specified by `ChromosomeValue`, which can be a single chromosome number or a vector of chromosome numbers. Default is all chromosomes in `CGHData`.

`FreqStruct = cghfreqplot(CGHData, ...'IncludeX', IncludeXValue, ...)` controls the inclusion of the X chromosome in the analysis. Choices are `true` (default) or `false`.

`FreqStruct = cghfreqplot(CGHData, ...'IncludeY', IncludeYValue, ...)` controls the inclusion of the Y chromosome in the analysis. Choices are `true` or `false` (default).

`FreqStruct = cghfreqplot(CGHData, ...'Chrominfo', ChrominfoValue, ...)` specifies the cytogenetic banding information for the chromosomes. `ChrominfoValue` can be either of the following

- Structure returned by the `cytobandread` function
- Character vector or string specifying the file name of an NCBI ideogram text file or a UCSC Genome Browser cytoband text file

Default is *Homo sapiens* cytogenetic banding information from the UCSC Genome Browser, NCBI Build 36.1 (`https://genome.UCSC.edu`).

Tip You can download files containing cytogenetic G-banding data from the NCBI or UCSC Genome Browser web site. For example, you can download the cytogenetic banding data (`cytoBandIdeo.txt.gz`) for *Homo sapiens* from:

`https://hgdownload.cse.ucsc.edu/goldenPath/hg19/database/`

`FreqStruct = cghfreqplot(CGHData, ...'ShowCentr', ShowCentrValue, ...)` controls the display of the centromere positions as vertical dashed lines in the frequency plot. Choices are `true` (default) or `false`.

Tip The centromere positions are obtained from `ChrominfoValue`.

`FreqStruct = cghfreqplot(CGHData, ...'Color', ColorValue, ...)` specifies a color scheme for the vertical lines in the plot, indicating the frequency of the gains and losses. Choices are:

- Name of or handle to a function that returns a colormap.
- M-by-3 matrix containing RGB values. If M equals 1, then that single color is used for all gains and losses. If M equals 2 or more, then the first row is used for gains, the second row is used for losses, and remaining rows are ignored. For example, `[0 1 0;1 0 0]` specifies green for gain and red for loss.

The default color scheme is a range of colors from pure green (gain = 1) through yellow (0) to pure red (loss = –1).

`FreqStruct = cghfreqplot(CGHData, ...'YLim', YLimValue, ...)` specifies the y vertical limits for the frequency plot. *`YLimValue`* is a two-element vector specifying the minimum and maximum values on the vertical axis. Default is `[1, -1]`.

`FreqStruct = cghfreqplot(CGHData, ...'Titles', TitlesValue, ...)` specifies titles for the group(s), which are added to the tops of the plot(s). *`TitlesValue`* can be a character vector, string, string vector, or a cell array of character vectors.

Examples

Display the frequency of copy number alterations from multiple samples

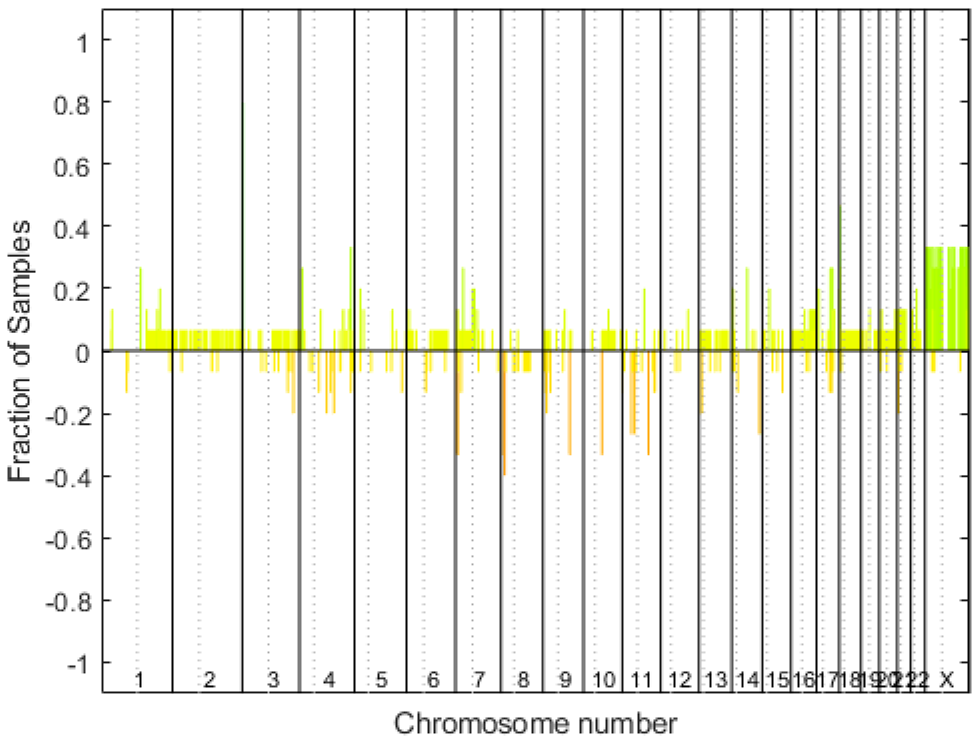
Plot data from the Coriell cell line study

Load the array-based CGH (aCGH) data from the Coriell cell line study (Snijders, A. et al., 2001).

```
load coriell_baccgh
```

Display a frequency plot of the copy number alterations across all samples.

```
Struct = cghfreqplot(coriell_data);
```



View data tips for the data, chromosomes, and centromeres. First click the **Data Cursor** button on the toolbar, then click the black chromosome boundary line, or a dotted centromere line in the plot. To delete this data tip, right-click it, then select **Delete Current Datatip**.

Display a color bar indicating the degree of gain or loss by clicking the **Insert Colorbar** button on the toolbar.

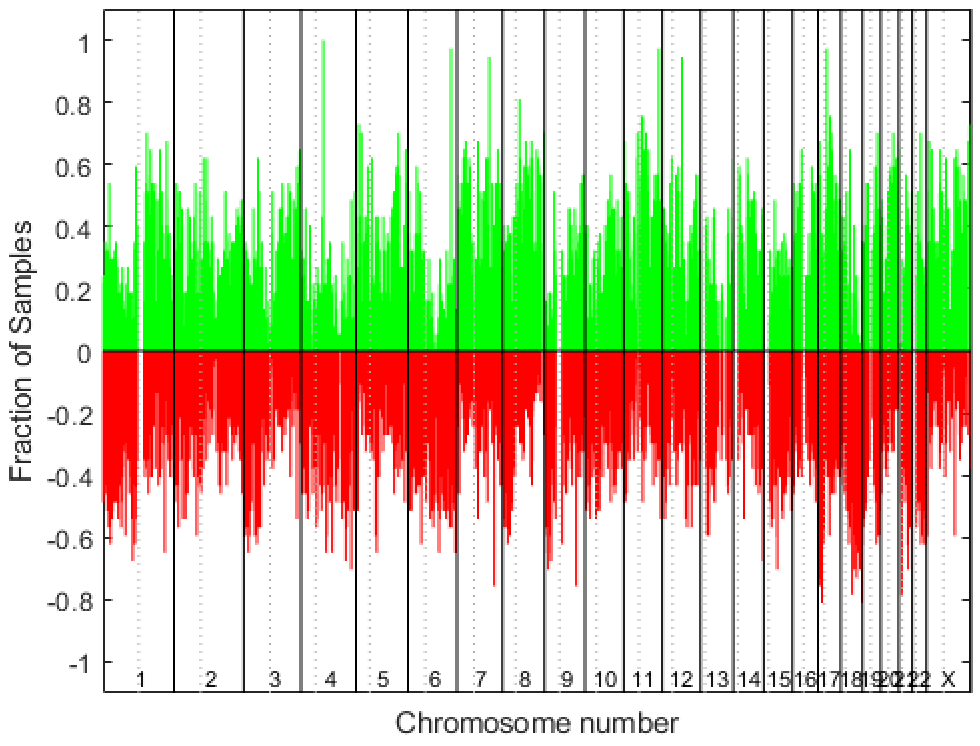
Plot data from a pancreatic cancer study

Load the aCGH data from a pancreatic cancer study (Aguirre, A. et al., 2004).

```
load pancrea_oligocgh
```

Display a frequency plot of the copy number alterations across all samples using a green and red color scheme.

```
cghfreqplot(pancrea_data, 'Color', [0 1 0; 1 0 0])
```



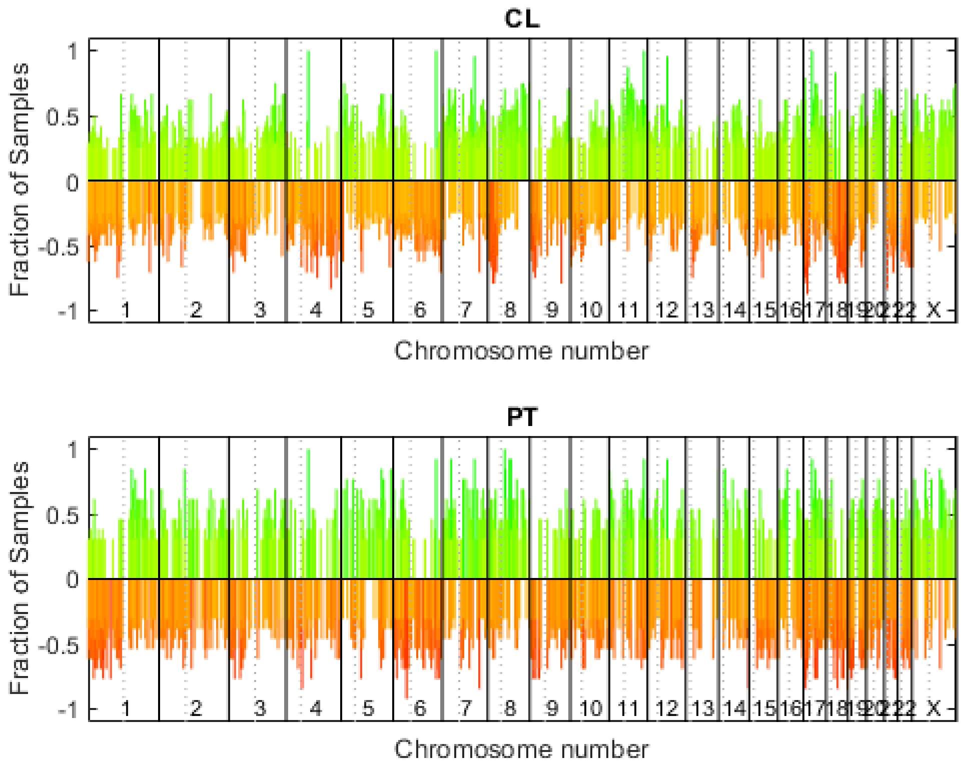
Plotting groups of aCGH Data

Define two groups of data.

```
grp1 = strncmp('PA.C', pancrea_data.Sample,4);
grp1_ind = find(grp1);
grp2 = strncmp('PA.T', pancrea_data.Sample,4);
grp2_ind = find(grp2);
```

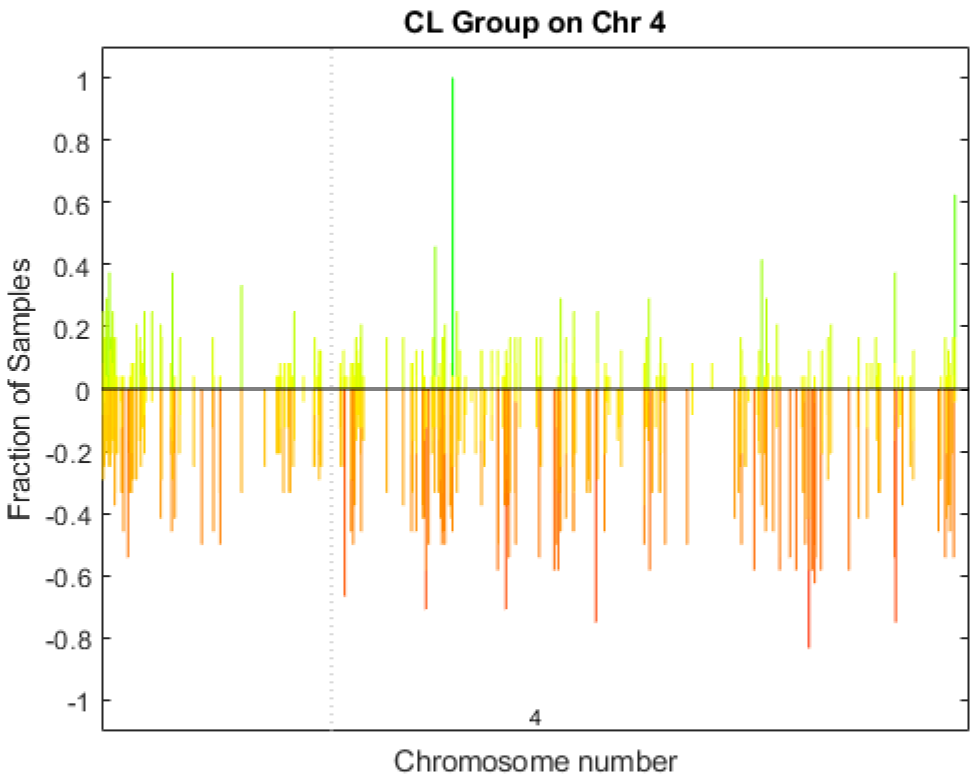
Display a frequency plot of the copy number alterations across all samples in the two groups and limit the plotting to only the clones with frequency gains or losses greater than or equal to 0.25.

```
SP = cghfreqplot(pancrea_data, 'Group', {grp1_ind, grp2_ind},...
                 'Title', {'CL', 'PT'}, 'Cutoff', 0.25);
```



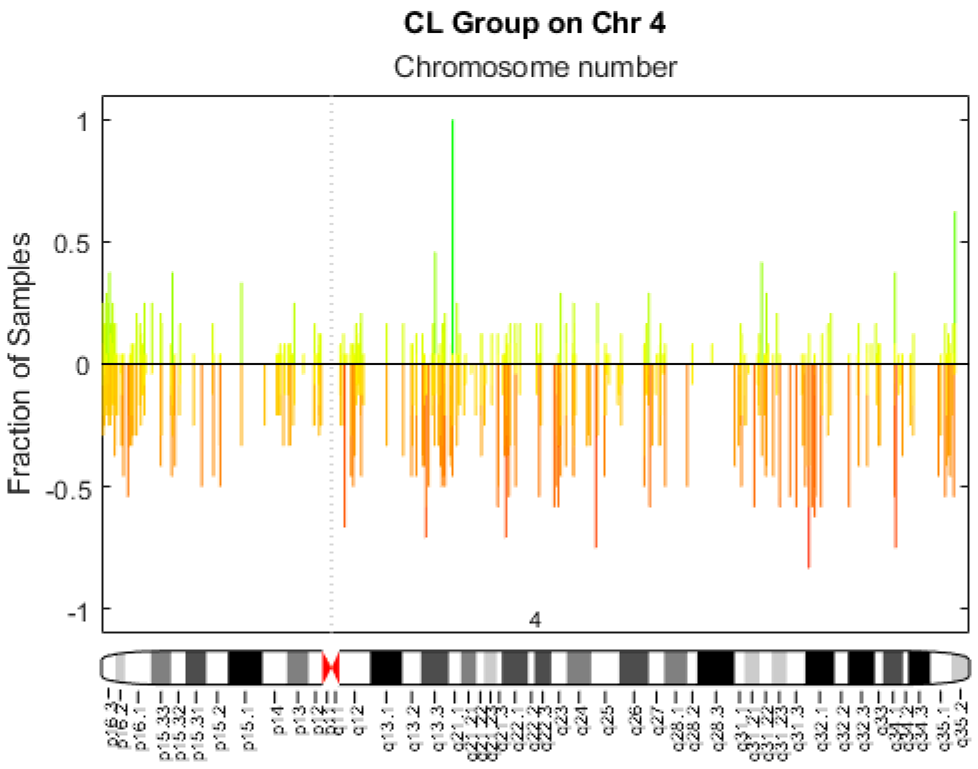
Display a frequency plot of the copy number alterations across all samples in the first group and limit the plot to chromosome 4 only.

```
SP = cghfreqplot(pancrea_data, 'Group', grp1_ind, ...
                 'Title', 'CL Group on Chr 4', 'Chromosome', 4);
```



Use the `chromosomeplot` function with the `'addtoplot'` option to add the ideogram of chromosome 4 for Homo sapiens to this frequency plot. Because the plot of the frequency data from the pancreatic cancer study is in kb units, use the `'Unit'` option to convert the ideogram data to kb units.

```
fh = gcf;
currentAxes = fh.CurrentAxes;
chromosomeplot('hs_cytoBand.txt', 4, 'addtoplot', currentAxes, 'Unit', 2);
```



Version History

Introduced in R2008a

References

[1] Snijders, A.M., Nowak, N., Segraves, R., Blackwood, S., Brown, N., Conroy, J., Hamilton, G., Hindle, A.K., Huey, B., Kimura, K., Law, S., Myambo, K., Palmer, J., Ylstra, B., Yue, J.P., Gray, J.W., Jain, A.N., Pinkel, D., and Albertson, D.G. (2001). Assembly of microarrays for genome-wide measurement of DNA copy number. Nature Genetics *29*, 263–264.

[2] Aguirre, A.J., Brennan, C., Bailey, G., Sinha, R., Feng, B., Leo, C., Zhang, Y., Zhang, J., Gans, J.D., Bardeesy, N., Cauwels, C., Cordon-Cardo, C., Redston, M.S., DePinho, R.A., and Chin, L. (2004). High-resolution characterization of the pancreatic adenocarcinoma genome. PNAS *101, 24*, 9067–9072.

See Also

`cghcbs` | `chromosomeplot` | `cytobandread`

chromosomeplot

Plot chromosome ideogram with G-banding pattern

Syntax

```
chromosomeplot(CytoData)
chromosomeplot(CytoData, ChromNum)

chromosomeplot(CytoData, ChromNum, ...,'Orientation', OrientationValue, ...)
chromosomeplot(CytoData, ChromNum, ...,'ShowBandLabel',
ShowBandLabelValue, ...)
chromosomeplot(CytoData, ChromNum, ...,'AddToPlot', AddToPlotValue, ...)
chromosomeplot(..., 'Unit', UnitValue, ...)
chromosomeplot(..., 'CNV', CNVValue, ...)
```

Arguments

CytoData	Either of the following: • Character vector or string specifying a file containing cytogenetic G-banding data (in bp units), such as an NCBI ideogram text file or a UCSC Genome Browser cytoband text file. • Structure containing cytogenetic G-banding data (in bp units) in the following fields: • `ChromLabels` • `BandStartBPs` • `BandEndBPs` • `BandLabels` • `GieStains` **Tip** Use the `cytobandread` function to create the structure to use for *`CytoData`*.
ChromNum	Scalar or character vector or string specifying a single chromosome to plot. Valid entries are integers, `'X'`, and `'Y'`. **Note** Setting *`ChromNum`* to `0` will plot ideograms for all chromosomes.
OrientationValue	Character vector or string or number that specifies the orientation of the ideogram of a single chromosome specified by *`ChromNum`*. Choices are `'Vertical'` or `1` (default) and `'Horizontal'` or `2`.
ShowBandLabelValue	Controls the display of band labels (such as `q25.3`) when plotting a single chromosome ideogram, specified by *`ChromNum`*. Choices are `true` (default) or `false`.

AddToPlotValue	Variable name of a figure axis to which to add the single chromosome ideogram, specified by *ChromNum*. **Note** If you use this property to add the ideogram to a plot of genomic data that is in units other than bp, use the `'Unit'` property to convert the ideogram data to the appropriate units. **Tip** Before printing a figure containing an added chromosome ideogram, change the background to white by issuing the following command: `set(gcf,'color','w')`
UnitValue	Integer that specifies the units (base pairs, kilo base pairs, or mega base pairs) for the starting and ending genomic positions. This unit is used in the data tip displayed when you hover the cursor over chromosomes in the ideogram. This unit can also be used when using the `'AddToPlot'` property to add the ideogram to a plot that is in units other than bp. Choices are `1` (bp), `2` (kb), or `3` (mb). Default is `1` (bp).
CNVValue	Controls the display of copy number variance (CNV) data, provided by *CNVValue*, aligned to the chromosome ideogram. Gains are shown in green to the right or above the ideogram, while losses are shown in red to the left or below the ideogram. *CNVValue* is a structure array containing the four fields described in the table below.

Description

`chromosomeplot(`*CytoData*`)` plots the ideogram of all chromosomes, using information from *CytoData*, a structure containing cytogenetic G-banding data (in bp units), or a character vector or string specifying a file containing cytogenetic G-banding data (in bp units), such as an NCBI ideogram text file or a UCSC Genome Browser cytoband text file. The G bands distinguish different areas of the chromosome. For example, for the *Homo sapiens* ideogram, possible G bands are:

- `gneg` — white
- `gpos25` — light gray
- `gpos50` — medium gray
- `gpos75` — dark gray
- `gpos100` — black
- `acen` — red (centromere)
- `stalk` — indented region (region with repeats)
- `gvar` — light blue

Darker bands are AT-rich, while lighter bands are GC-rich.

`chromosomeplot(`*CytoData*`, `*ChromNum*`)` plots the ideogram of a single chromosome specified by *ChromNum*.

`chromosomeplot(..., 'PropertyName', PropertyValue, ...)` calls `chromosomeplot` with optional properties that use property name/property value pairs. You can specify one or more properties in any order. Each *`PropertyName`* must be enclosed in single quotation marks and is case insensitive. These property name/property value pairs are as follows:

`chromosomeplot(CytoData, ChromNum, ...,'Orientation', OrientationValue, ...)` specifies the orientation of the ideogram of a single chromosome specified by *`ChromNum`*. Choices are `'Vertical'` or `1` (default) and `'Horizontal'` or `2`.

Note When plotting the ideogram of all chromosomes, the orientation is always vertical.

`chromosomeplot(CytoData, ChromNum, ...,'ShowBandLabel', ShowBandLabelValue, ...)` displays band labels (such as q25.3) when plotting a single chromosome ideogram, specified by *`ChromNum`*. Choices are `true` (default) or `false`.

`chromosomeplot(CytoData, ChromNum, ...,'AddToPlot', AddToPlotValue, ...)` adds the single chromosome ideogram, specified by *`ChromNum`*, to a figure axis specified by *`AddToPlotValue`*.

Note If you use this property to add the ideogram to a plot of genomic data that is in units other than bp, use the `'Unit'` property to convert the ideogram data to the appropriate units.

Tip Before printing a figure containing an added chromosome ideogram, change the background to white by issuing the following command:

```
set(gcf,'color','w')
```

`chromosomeplot(..., 'Unit', UnitValue, ...)` specifies the units (base pairs, kilo base pairs, or mega base pairs) for the starting and ending genomic positions. This unit is used in the data tip displayed when you hover the cursor over chromosomes in the ideogram. This unit can also be used when using the `'AddToPlot'` property to add the ideogram to a plot that is in units other than bp. Choices are `1` (bp), `2` (kb), or `3` (mb). Default is `1` (bp).

`chromosomeplot(..., 'CNV', CNVValue, ...)` displays copy number variance (CNV) data, provided by *`CNVValue`*, aligned to the chromosome ideogram. Gains are shown in green to the right or above the ideogram, while losses are shown in red to the left or below the ideogram. *`CNVValue`* is a structure array containing the following fields. Each field must contain the same number of elements.

Field	Description

Chromosome	Either of the following:
	• Numeric vector containing the chromosome number on which each CNV is located.
	Note For the sex chromosome, X, use *N*, where *N* = number of autosomes + 1. For the sex chromosome, Y, use *M*, where *M* = number of autosomes + 2. For example, for *Homo sapiens* use 23 for X and 24 for Y, and for *Mus musculus* (lab mouse), use 20 for X and 21 for Y.
	• Character array containing the chromosome number on which each CNV is located.
	Note Using a character array lets you use the characters X and Y (instead of numbers) for sex chromosomes. However, all elements in the array must be the same width, which may require you to add spaces to some character vectors. For example:
	`[' 1'; ' 2'; '10'; ' X']`
	Or you can use the `char` function with a cell array to create a character array of the chromosome numbers and letters. For example:
	`char({'1', '2', '10', 'X'})`

CNVType	Numeric vector containing the type of each CNV, either 1 (loss) or 2 (gain).
Start	Numeric vector containing the starting genomic position of each CNV. Units must be in base pairs.
End	Numeric vector containing the ending genomic position of each CNV. Units must be in base pairs.

Examples

Plot Chromosome Ideograms

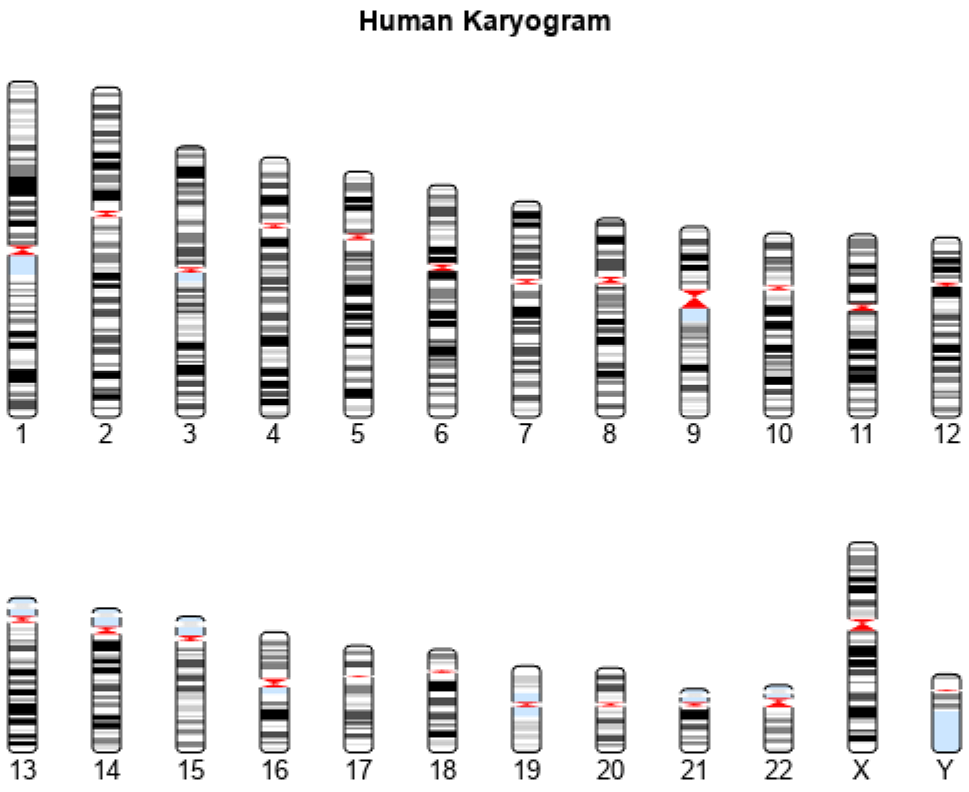
Read the cytogenetic banding information for *Homo sapiens* into a structure.

```
hs_cytobands = cytobandread('hs_cytoBand.txt')
```

```
hs_cytobands = struct with fields:
     ChromLabels: {862x1 cell}
    BandStartBPs: [862x1 int32]
      BandEndBPs: [862x1 int32]
      BandLabels: {862x1 cell}
       GieStains: {862x1 cell}
```

Plot the entire chromosome ideogram.

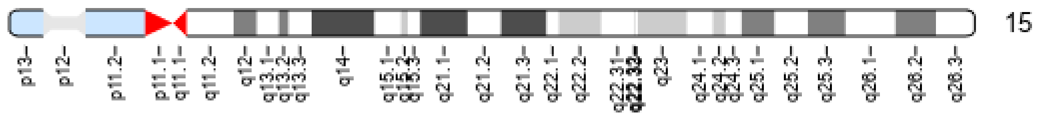
```
chromosomeplot(hs_cytobands);
title('Human Karyogram')
```



You can display the ideogram of a specific chromosome by right-clicking it in the plot, then selecting **Display in New Figure > Vertical** or **Horizontal**.

You can also programmatically display the ideogram of a specific chromosome, set the orientation, and the units used in the data tip to kilo base pairs.

```
chromosomeplot(hs_cytobands, 15, 'Orientation', 2, 'Unit', 2);
```



Hover over the chromosome to view a data tip. To get more information about a specific band, select the Data Cursor button on the toolbar and click the band in the plot. Use the context menu (right-click) to see more options such as deleting or creating a data tip.

Display copy number alteration data aligned to chromosome ideogram

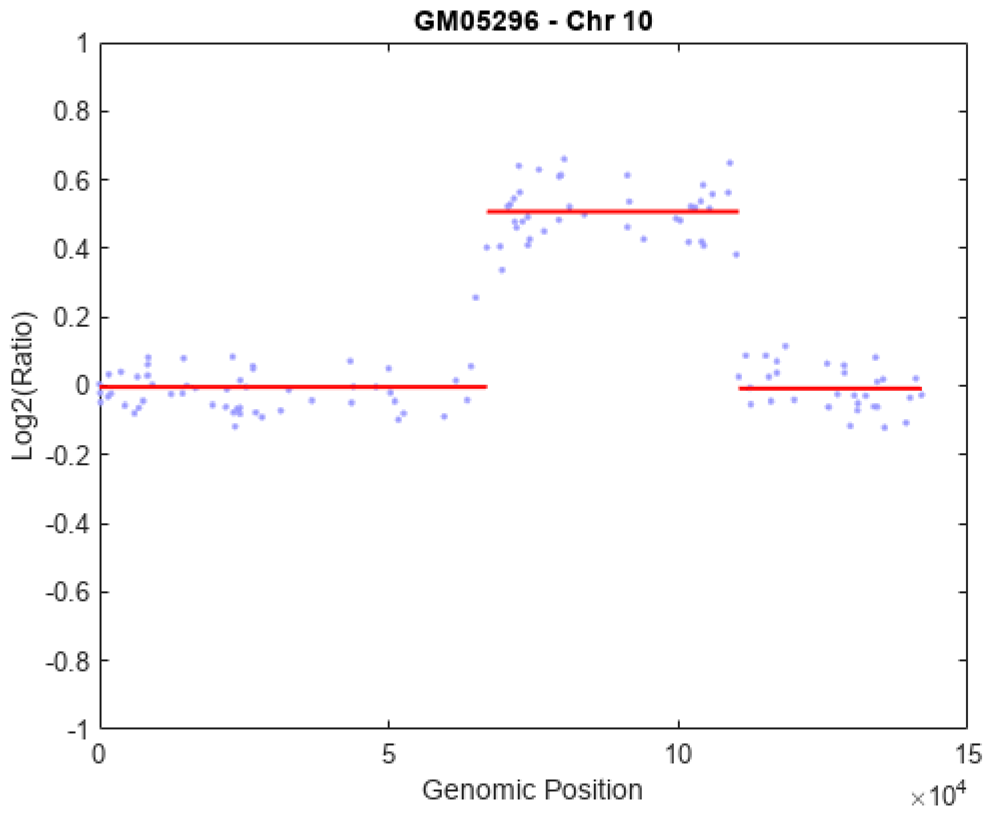
Load the array-based CGH (aCGH) data from the Coriell cell line study (Snijders, A. et al., 2001).

```
load coriell_baccgh
```

Use the `cghcbs` function to analyze chromosome 10 of sample 3 (GM05296) of the aCGH data and return copy number variance (CNV) data in a structure, S. Plot the segment means over the original data for only chromosome 10 of sample 3.

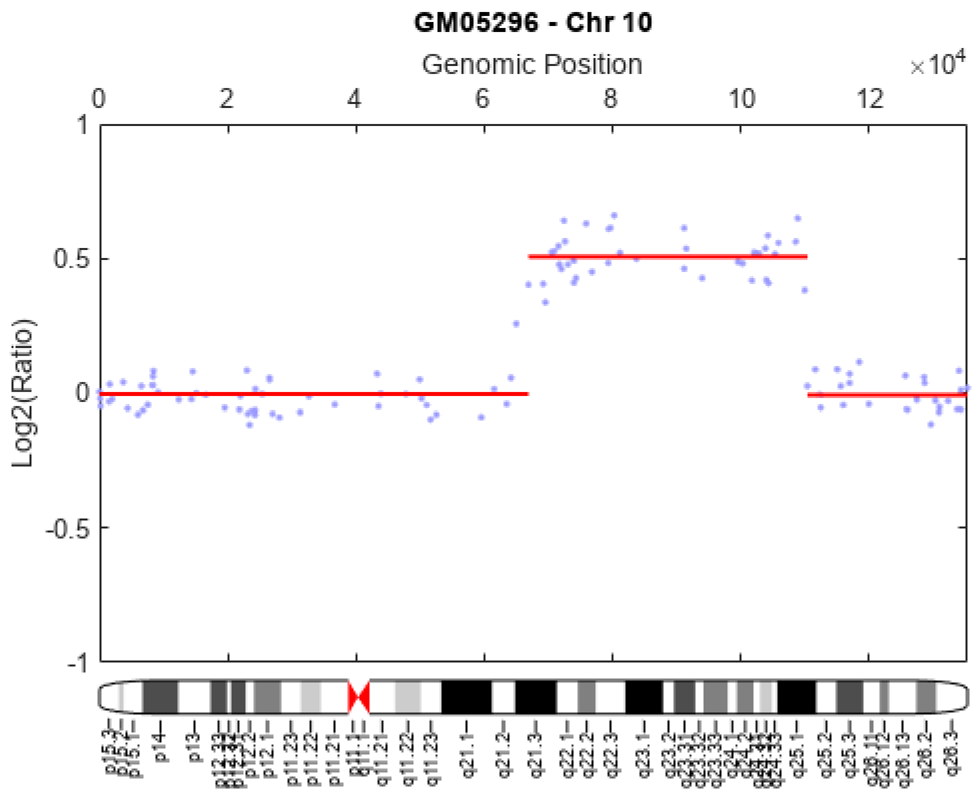
```
S = cghcbs(coriell_data,'sampleindex',3,'chromosome',10,...
           'showplot',10);
```

```
Analyzing: GM05296. Current chromosome 10
```



Use the chromosomeplot function with the 'addtoplot' option to add the ideogram of chromosome 10 for Homo sapiens to the plot. Because the plot of the CNV data from the Coriell cell line study is in kb units, use the 'Unit' property to convert the ideogram data to kb units.

```
set(gcf,'color','w'); % Set the background of the figure to white.
chromosomeplot('hs_cytoBand.txt', 10, 'addtoplot', gca,...
               'Unit', 2);
```



Version History

Introduced in R2007b

References

[1] Snijders, A.M., Nowak, N., Segraves, R., Blackwood, S., Brown, N., Conroy, J., Hamilton, G., Hindle, A.K., Huey, B., Kimura, K., Law, S., Myambo, K., Palmer, J., Ylstra, B., Yue, J.P., Gray, J.W., Jain, A.N., Pinkel, D., and Albertson, D.G. (2001). Assembly of microarrays for genome-wide measurement of DNA copy number. Nature Genetics *29*, 263–264.

See Also

`cghcbs` | `cytobandread`

cigar2align

Convert unaligned sequences to aligned sequences using signatures in CIGAR format

Syntax

```
Alignment = cigar2align(Seqs,Cigars)
[GapSeq,Indices] = cigar2align(Seqs,Cigars)
___ = cigar2align(Seqs,Cigars,Name,Value)
```

Description

`Alignment = cigar2align(Seqs,Cigars)` converts unaligned sequences in `Seqs` into `Alignment` using the information stored in `Cigars`.

`[GapSeq,Indices] = cigar2align(Seqs,Cigars)` converts unaligned sequences in `Seqs` into `GapSeq`, and also returns `Indices`, a vector of numeric indices, using the information stored in `Cigars`. When an alignment has many columns, this syntax uses less memory and is faster.

`___ = cigar2align(Seqs,Cigars,Name,Value)`, for any outputs, specifies additional options using one or more name-value arguments. For example, to have the output display gaps in the reference sequence, use `Alignment = cigar2align(Seqs,Cigars,GapsInRef=true)`.

Examples

Reconstruct Alignment Using cigar2align

Create a cell array of character vectors containing unaligned sequences, create a cell array of corresponding CIGAR-formatted character vectors associated with a reference sequence of ACGTATGC, and then reconstruct the alignment.

```
Seqs = {'ACGACTGC', 'ACGTTGC', 'AGGTATC'}; % unaligned sequences
Cigars = {'3M1D1M1I3M', '4M1D1P3M', '5M1P1M1D1M'}; % cigar-formatted
Alignment = cigar2align(Seqs, Cigars)
```

```
Alignment = 3x8 char array
    'ACG-ATGC'
    'ACGT-TGC'
    'AGGTAT-C'
```

Reconstruct the same alignment to display positions in the aligned sequences that correspond to gaps in the reference sequence.

```
Alignment2 = cigar2align(Seqs,Cigars,GapsInRef=true)
```

```
Alignment2 = 3x9 char array
    'ACG-ACTGC'
    'ACGT--TGC'
    'AGGTA-T-C'
```

Reconstruct the alignment adding an offset padding of 5.

```
Alignment3 = cigar2align(Seqs,Cigars,Start=[5 5 5],OffsetPad=true)
```

```
Alignment3 = 3x12 char array
    '    ACG-ATGC'
    '    ACGT-TGC'
    '    AGGTAT-C'
```

Use the two-output syntax to obtain the alignment and indices.

```
[GapSeq, Indices] = cigar2align(Seqs,Cigars)
```

```
GapSeq = 3x1 cell
    {'ACG-ATGC'}
    {'ACGT-TGC'}
    {'AGGTAT-C'}


Indices = 3×1

     1
     1
     1
```

Input Arguments

Seqs — Unaligned sequences
cell array of character vectors | string vector

Unaligned sequences, specified as a cell array of character vectors or as a string vector. `Seqs` must contain the same number of elements as `Cigars`.

Data Types: `string` | `cell`

`Cigars` — Formats for sequences
cell array of CIGAR–formatted character vectors | CIGAR–formatted string vector

Formats for sequences, specified as a cell array of valid CIGAR–formatted character vectors or a CIGAR–formatted string vector. `Cigars` must contain the same number of elements as `Seqs`.

Data Types: `string` | `cell`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `Alignment = cigar2align(Seqs,Cigars,GapsInRef=true)`

`GapsInRef` — Indication to display gaps
`false` (default) | `true`

indication to display positions in the aligned sequences that correspond to gaps in the reference sequence, specified as `false` (do not display gaps) or `true`. If your reference sequence has gaps and you set `GapsInRef` to `false`, and then later use `Alignment` as input to `align2cigar`, the returned CIGAR–formatted character vectors will not match the original ones.

Example: `true`

Data Types: `logical`

`OffsetPad` — Indication to add padding blanks to left of each aligned sequence
`false` (default) | `true`

Indication to add padding blanks to left of each aligned sequence, specified as `false` (do not add padding) or `true`. The added padding places blanks to the left of each aligned read sequence. The offset of the start position is from the first position of the reference sequence. When `false`, the matrix of aligned sequences starts at the start position of the leftmost aligned read sequence.

Example: `true`

Data Types: `logical`

`SoftClipping` — Indication to include characters in the aligned read sequences corresponding to soft clipping ends
`false` (default) | `true`

Indication to include characters in the aligned read sequences corresponding to soft clipping ends, specified as `false` (do not include) or `true`.

Example: `true`

Data Types: `logical`

`Start` — Reference sequence position at which each aligned sequence starts
position `1` of the reference sequence (default) | vector of positive integers

Reference sequence position at which each aligned sequence starts, specified as a vector of positive integers. By default, each aligned sequence starts at position 1 of the reference sequence.

Data Types: `single` | `double`

Output Arguments

`Alignment` — Aligned sequences
character array

Aligned sequences, returned as a character array. Each row of `Alignment` represents one aligned sequence. The number of rows of `Alignment` equals the number of character vectors in `Seqs`.

GapSeq — Aligned sequences
cell array

Aligned sequences without any leading or trailing whitespace, returned as a cell array of character vectors. The number character vectors in `GapSeq` equals the number of character vectors in `Seqs`.

`Indices` — Indices of starting columns in GapSeq
double vector

Indices of starting columns in `Alignment`, returned as a numeric vector. Sequences returned in `GapSeq` are identical to those in the `Alignment` output except that those in `GapSeq` have no leading or trailing whitespace.

Entries in `Indices` are not necessarily the same as the start positions in the reference sequence for each aligned sequence. This is because either of the following can hold:

- The reference sequence can be extended to account for insertions.
- An aligned sequence can have leading soft clippings, padding, or insertion characters.

Algorithms

When `cigar2align` reconstructs the alignment, it does not display hard clipped positions (H) or soft clipped positions (S). Also, it does not consider soft clipped positions as start positions for aligned sequences.

Alternative Functionality

If your CIGAR information is captured in the `Signature` property of a `BioMap` object, you can use the `getAlignment` method to construct the alignment.

Version History

Introduced in R2010b

See Also

`align2cigar` | `seqalignviewer` | `getBaseCoverage` | `getCompactAlignment` | `getAlignment` | `BioMap`

Topics

"Manage Sequence Read Data in Objects"

External Websites

Sequence Read Archive
SAM format specification

classperf

Evaluate classifier performance

Syntax

```
classperf
cp = classperf(groundTruth)
cp = classperf(groundTruth,classifierOutput)

classperf(cp,classifierOutput)
classperf(cp,classifierOutput,testIdx)

classperf(___,Name,Value)
```

Description

`classperf` without input arguments displays the properties of a `classperformance` object. For more information, see classperformance Properties.

`cp = classperf(groundTruth)` creates an empty `classperformance` object `cp` using the true labels `groundTruth` for every observation in your data set.

`cp = classperf(groundTruth,classifierOutput)` creates a `classperformance` object `cp` using the true labels `groundTruth`, and then updates the object properties based on the results of the classifier `classifierOutput`. Use this syntax when you want to know the classifier performance on a single validation run.

`classperf(cp,classifierOutput)` updates the `classperformance` object `cp` with the results of a classifier `classifierOutput`. Use this syntax to update the performance of the classifier iteratively, such as inside a `for` loop for multiple cross-validation runs.

`classperf(cp,classifierOutput,testIdx)` uses `testIdx` to compare the results of the classifier to the true labels and update the object `cp`. `testIdx` represents a subset of the true labels (ground truth) in the current validation.

`classperf(___,Name,Value)` specifies additional options with one or more `Name,Value` pair arguments. Specify these options after all other input arguments.

Examples

Perform 10-Fold Cross-Validation

Create indices for the 10-fold cross-validation and classify measurement data for the Fisher iris data set. The Fisher iris data set contains width and length measurements of petals and sepals from three species of irises.

Load the data set.

```
load fisheriris
```

Create indices for the 10-fold cross-validation.

```
indices = crossvalind('Kfold',species,10);
```

Initialize an object to measure the performance of the classifier.

```
cp = classperf(species);
```

Perform the classification using the measurement data and report the error rate, which is the ratio of the number of incorrectly classified samples divided by the total number of classified samples.

```
for i = 1:10
    test = (indices == i);
    train = ~test;
    class = classify(meas(test,:),meas(train,:),species(train,:));
    classperf(cp,class,test);
end
cp.ErrorRate
```

```
ans = 0.0200
```

Suppose you want to use the observation data from the `setosa` and `virginica` species only and exclude the `versicolor` species from cross-validation.

```
labels = {'setosa','virginica'};
indices = crossvalind('Kfold',species,10,'Classes',labels);
```

`indices` now contains zeros for the rows that belong to the `versicolor` species.

Perform the classification again.

```
for i = 1:10
    test = (indices == i);
    train = ~test;
    class = classify(meas(test,:),meas(train,:),species(train,:));
    classperf(cp,class,test);
end
cp.ErrorRate
```

```
ans = 0.0160
```

Classify Fisher Iris Data Using K-Nearest Neighbor

Load the data set.

```
load fisheriris
X = meas;
Y = species;
```

X is a numeric matrix that contains four petal measurements for 150 irises. Y contains the true class names (labels) of the corresponding iris species.

Initialize the `classperformance` object using the true labels.

```
cp = classperf(Y)
cp = 
  classperformance with properties:

                  ClassLabels: {3x1 cell}
                  GroundTruth: [150x1 double]
         NumberOfObservations: 150
            ValidationCounter: 0
           SampleDistribution: [150x1 double]
            ErrorDistribution: [150x1 double]
    SampleDistributionByClass: [3x1 double]
     ErrorDistributionByClass: [3x1 double]
               CountingMatrix: [4x3 double]
                  CorrectRate: NaN
                    ErrorRate: NaN
              LastCorrectRate: 0
                LastErrorRate: 0
             InconclusiveRate: NaN
               ClassifiedRate: NaN
                  Sensitivity: NaN
                  Specificity: NaN
      PositivePredictiveValue: NaN
      NegativePredictiveValue: NaN
           PositiveLikelihood: NaN
           NegativeLikelihood: NaN
                   Prevalence: NaN
              DiagnosticTable: [2x2 double]
                        Label: ''
                  Description: ''
               ControlClasses: [2x1 double]
                TargetClasses: 1
```

Perform the classification using the k-nearest neighbor classifier. Cross-validate the model 10 times by using 145 samples as the training set and 5 samples as the test set. After each cross-validation run, update the classifier performance object with the results.

```
for i = 1:10
    [train,test] = crossvalind('LeaveMOut',Y,5);
    mdl = fitcknn(X(train,:),Y(train),'NumNeighbors',3);
    predictions = predict(mdl,X(test,:));
    classperf(cp,predictions,test);
end
```

Report the classification error rate, which is a ratio of the number of incorrectly classified samples divided by the total number of classified samples.

```
cp.ErrorRate
```

```
ans = 0.0467
```

Input Arguments

groundTruth — True labels

vector of integers | logical vector | string vector | cell array of character vectors

True labels for all observations in your data set, specified as a vector of integers, logical vector, string vector, or cell array of character vectors.

`classifierOutput` — Classification results
vector of integers | logical vector | string vector | cell array of character vectors

Classification results from a classifier, specified as a vector of integers, logical vector, string vector, or cell array of character vectors. When `classifierOutput` is a cell array of character vectors or string vector, an empty character vector or string represents an inconclusive result. For a vector of integers, `NaN` represents an inconclusive result.

- If you do not specify `testIdx`, `classifierOutput` must be the same size and data type as `groundTruth`.
- If you specify `testIdx` as a vector of integers, `classifierOutput` must have the same number of elements as `testIdx`. If `testIdx` is a logical vector, the number of elements in `classifierOutput` must equal `sum(testIdx)`.

`cp` — Classifier performance information
`classperformance` object

Classifier performance information, specified as a `classperformance` object. For details, see classperformance Properties.

`testIdx` — Subset of true labels
vector of integers | logical vector

Subset of true labels (`groundTruth`), specified as a vector of integers or logical vector. The `testIdx` argument indicates a subset of true labels (from a test set). The function uses `testIdx` as an index vector to get a subset of labels from `groundTruth`, such as `groundTruth(testIdx)`.

- If `testIdx` is a logical vector, its length must equal the total number of observations (`cp.NumberOfObservations`).
- If `testIdx` is a vector of integers, it cannot contain duplicate integers, and each integer must be greater than `0` but less than or equal to the total number of observations.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `cp = classperf(groundTruth,classifierOutput,'Positive',[1 2 3])` specifies the labels for the target (diseased) classes.

`Positive` — Labels for target classes
vector of integers | logical vector | string vector | cell array of character vectors

Labels for the target classes, specified as the comma-separated pair consisting of `'Positive'` and a vector of integers, logical vector, string vector, or cell array of character vectors.

- If `groundTruth` is a vector of integers, the positive label and negative label (specified by the `'Negative'` name-value pair argument) must be vectors of integers.
- If `groundTruth` is a string vector or cell array of character vectors, the positive label and negative label can be string vectors, cell arrays of character vectors, or vectors of positive integers. The entries must be a subset of `grp2idx(groundTruth)`.

By default, the positive label corresponds to the first class returned by `grp2idx(groundTruth)` and the negative label corresponds to all other classes.

The function uses the positive label to set the `TargetClasses` property of the `cp` object.

The positive and negative labels are disjoint subsets of `unique(groundTruth)`. For example, suppose you have a data set that contains data from six patients. Five patients have ovarian, lung, prostate, skin, or brain cancer, and one patient does not have cancer. Then `ClassLabels = {'Ovarian', 'Lung', 'Prostate', 'Skin', 'Brain', 'Healthy'}`. You can test a classifier for lung cancer only by setting the positive label to `[2]` and the negative label to `[1 3 4 5 6]`. Alternatively, you can test for any type of cancer by setting the positive label to `[1 2 3 4 5]` and the negative label to `[6]`.

In clinical tests, the function counts inconclusive values (empty character vector `''` or `NaN`) as false negatives to calculate the specificity and as false positives to calculate the sensitivity. The function dose not count any tested observation with its true class not within the union of positive label and negative label. However, if the true class of a tested observation is within the union but its predicted class is not covered by `groundTruth`, the function counts that observation as inconclusive.

Example: `'Positive',[1 2]`

`Negative` — Labels for control classes
vector of integers | logical vector | string vector | cell array of character vectors

Labels for the control classes, specified as the comma-separated pair consisting of `'Negative'` and a vector of integers, logical vector, string vector, or cell array of character vectors.

- If `groundTruth` is a vector of integers, the positive label and negative label (specified by the `'Negative'` name-value pair argument) must be vectors of integers.
- If `groundTruth` is a string vector or cell array of character vectors, the positive label and negative label can be string vectors, cell arrays of character vectors, or vectors of positive integers. The entries must be a subset of `grp2idx(groundTruth)`.

By default, the positive label corresponds to the first class returned by `grp2idx(groundTruth)` and the negative label corresponds to all other classes.

The function uses the negative label to set the `ControlClasses` property of the `cp` object. For details on how the function uses the positive and negative labels, see “Positive” on page 1-0 .

Example: `'Negative',[3]`

Version History

Introduced before R2006a

See Also

classperformance Properties | `crossvalind` | `classify` | `grp2idx`

classperformance Properties

Classifier performance information

Description

To view the performance-related information of a classifier, create a `classperformance` object by using the `classperf` function. Use dot notation to access the object properties, such as `CorrectRate`, `ErrorRate`, `Sensitivity`, and `Specificity`.

Properties

Name and Description

`Label` — Name of classifier object
`''` (default) | character vector

Name of the classifier object, specified as a character vector. Use dot notation to set this property.

Example: `'cp_kfold'`

Data Types: `char`

`Description` — Description of object
`''` (default) | character vector

Description of the object, specified as a character vector. Use dot notation to set this property.

Example: `'performance_data_kfold'`

Data Types: `char`

True Labels and Indices

`ClassLabels` — Unique set of true labels
vector of positive integers | cell array of character vectors

This property is read-only.

Unique set of true labels from `groundTruth`, specified as a vector of positive integers or cell array of character vectors. This property is equivalent to the output when you run `unique(groundTruth)`.

Example: `{'ovarian','liver','normal'}`

Data Types: `double` | `cell`

`GroundTruth` — True labels for all observations
vector of positive integers | cell array of character vectors

This property is read-only.

True labels for all observations in your data set, specified as a vector of positive integers or cell array of character vectors.

Example: `{'ovarian','liver','normal','ovarian','ovarian','liver'}`

Data Types: `double` | `cell`

`NumberOfObservations` — Number of observations
positive integer

This property is read-only.

Number of observations in your data set, specified as a positive integer.

Example: `200`

Data Types: `double`

`ControlClasses` — Indices to control classes from true labels
vector of positive integers

Indices to the control classes from the true labels (`ClassLabels`), specified as a vector of positive integers. This property indicates the control (or negative) classes in the diagnostic test. By default, `ControlClasses` contains all classes other than the first class returned by `grp2idx(groundTruth)`.

You can set this property by using dot notation or the `'Negative'` name-value pair argument with the `classperf` function.

Example: `[3]`

Data Types: `double`

`TargetClasses` — Indices to target classes from true labels
vector of positive integers

Indices to the target classes from the true labels (`ClassLabels`), specified as a vector of positive integers. This property indicates the target (or positive) classes in the diagnostic test. By default, `TargetClasses` contains the first class returned by `grp2idx(groundTruth)`.

You can set this property by using dot notation or the `'Positive'` name-value pair argument with the `classperf` function.

Example: `[1 2]`

Data Types: `double`

Sample and Error Distributions

`SampleDistribution` — Number of evaluations for each sample
numeric vector

This property is read-only.

Number of evaluations for each sample during the validation, specified as a numeric vector. For example, if you use resubstitution, `SampleDistribution` is a vector of ones and `ValidationCounter` = 1. If you have a 10-fold cross-validation, `SampleDistribution` is also a vector of ones, but `ValidationCounter` = 10.

`SampleDistribution` is useful when performing Monte Carlo partitions of the test sets, and it can help determine if each sample is tested an equal number of times.

Example: `[0 0 2 0]`

Data Types: `double`

`ErrorDistribution` — Frequency of misclassification of each sample
numeric vector

This property is read-only.

Frequency of misclassification of each sample, specified as a numeric vector.

Example: `[0 0 1 0]`

Data Types: `double`

`SampleDistributionByClass` — Frequency of true classes during validation
numeric vector

This property is read-only.

Frequency of the true classes during the validation, specified as a numeric vector.

Example: `[10 10 0]`

Data Types: `double`

`ErrorDistributionByClass` — Frequency of errors for each class
numeric vector

This property is read-only.

Frequency of errors for each class during the validation, specified as a numeric vector.

Example: `[0 0 0]`

Data Types: `double`

Performance Statistics

`ValidationCounter` — Number of validations
positive integer

This property is read-only.

Number of validations, specified as a positive integer.

Example: `10`

Data Types: `double`

`CountingMatrix` — Classification confusion matrix
numeric array

This property is read-only.

Classification confusion matrix, specified as a numeric array. The order of the rows and columns in the matrix is the same as in `grp2idx(groundTruth)`. Columns represent the true classes, and rows represent the classifier prediction. The last row in `CountingMatrix` is reserved for counting inconclusive results.

Example: `[10 0 0;0 10 0; 0 0 0; 0 0 0]`

Data Types: `double`

`CorrectRate` — Correct rate of classifier
positive scalar

This property is read-only.

Correct rate of the classifier, specified as a positive scalar. `CorrectRate` is defined as the number of correctly classified samples divided by the number of classified samples. Inconclusive results are not counted.

Example: `1`

Data Types: `double`

`ErrorRate` — Error rate of classifier
positive scalar

This property is read-only.

Error rate of the classifier, specified as a positive scalar. `ErrorRate` is defined as the number of incorrectly classified samples divided by the number of classified samples. Inconclusive results are not counted.

Example: `0`

Data Types: `double`

`LastCorrectRate` — Correct rate of classifier during last run
positive scalar

This property is read-only.

Correct rate of the classifier during the last validation run, specified as a positive scalar. In contrast with `CorrectRate`, `LastCorrectRate` only applies to the evaluated samples from the most recent validation run of the classifier performance object.

Example: `1`

Data Types: `double`

`LastErrorRate` — Error rate of classifier during last validation
positive scalar

This property is read-only.

Error rate of the classifier during the last validation run, specified as a positive scalar. In contrast with `ErrorRate`, `LastErrorRate` only applies to the evaluated samples from the most recent validation run of the classifier performance object.

Example: `0`

Data Types: `double`

`InconclusiveRate` — Inconclusive rate of classifier
positive scalar

This property is read-only.

Inconclusive rate of the classifier, specified as a positive scalar. `InconclusiveRate` is defined as the number of nonclassified (inconclusive) samples divided by the total number of samples.

Example: `0`

Data Types: `double`

`ClassifiedRate` — Classified rate of classifier
positive scalar

This property is read-only.

Classified rate of the classifier, specified as a positive scalar. `ClassifiedRate` is defined as the number of classified samples divided by the total number of samples.

Example: `1`

Data Types: `double`

`Sensitivity` — Sensitivity of classifier
positive scalar

This property is read-only.

Sensitivity of the classifier, specified as a positive scalar. `Sensitivity` is defined as the number of correctly classified positive samples divided by the number of true positive samples.

Inconclusive results that are true positives are counted as errors for computing `Sensitivity`. In other words, inconclusive results can decrease the diagnostic value of the test.

Example: `1`

Data Types: `double`

`Specificity` — Specificity of classifier
positive scalar

This property is read-only.

Specificity of the classifier, specified as a positive scalar. `Specificity` is defined as the number of correctly classified negative samples divided by the number of true negative samples.

Inconclusive results that are true negatives are counted as errors for computing `Specificity`. In other words, inconclusive results can decrease the diagnostic value of the test.

Example: `0.8`

Data Types: `double`

`PositivePredictiveValue` — Positive predictive value of classifier
positive scalar

This property is read-only.

Positive predictive value of the classifier, specified as a positive scalar. `PositivePredictiveValue` is defined as the number of correctly classified positive samples divided by the number of positive classified samples.

Inconclusive results are classified as negative when computing `PositivePredictiveValue`.

Example: `1`

Data Types: `double`

`NegativePredictiveValue` — Negative predictive value of classifier
positive scalar

This property is read-only.

Negative predictive value of the classifier, specified as a positive scalar. `NegativePredictiveValue` is defined as the number of correctly classified negative samples divided by the number of negative classified samples.

Inconclusive results are classified as positive when computing `NegativePredictiveValue`.

Example: `1`

Data Types: `double`

`PositiveLikelihood` — Positive likelihood of classifier
positive scalar

This property is read-only.

Positive likelihood of the classifier, specified as a positive scalar. `PositiveLikelihood` is defined as `Sensitivity / (1 - Specificity)`.

Example: `5`

Data Types: `double`

`NegativeLikelihood` — Negative likelihood of classifier
positive scalar

This property is read-only.

Negative likelihood of the classifier, specified as a positive scalar. `NegativeLikelihood` is defined as `(1 - Sensitivity)/Specificity`.

Example: `0`

Data Types: `double`

`Prevalence` — Prevalence of classifier
positive scalar

This property is read-only.

Prevalence of the classifier, specified as a positive scalar. `Prevalence` is defined as the number of true positive samples divided by the total number of samples.

Example: `1`

Data Types: `double`

`DiagnosticTable` — Diagnostic table
2-by-2 numeric array

This property is read-only.

Diagnostic table, specified as a two-by-two numeric array. The first row indicates the number of samples classified as positive, with the number of true positives in the first column and the number of false positives in the second column. The second row indicates the number of samples classified as negative, with the number of false negatives in the first column and the number of true negatives in the second column.

Correct classifications appear in the diagonal elements and errors appear in the off-diagonal elements. Inconclusive results are considered errors and are counted in the off-diagonal elements. For an example, see “Diagnostic Table Example” on page 1-540.

Example: `[20 0;0 0]`

Data Types: `double`

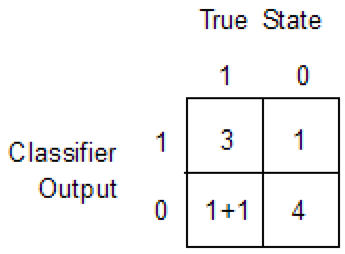
More About

Diagnostic Table Example

Suppose that a cancer study of 10 patients yields these results.

Patient	Classifier Output	Has Cancer
1	Positive	Yes
2	Positive	Yes
3	Positive	Yes
4	Positive	No
5	Negative	Yes
6	Negative	No
7	Negative	No
8	Negative	No
9	Negative	No
10	Inconclusive	Yes

Using these results, the function computes the `DiagnosticTable` as follows:


True State
1
0
Classifier Output
1
0
3
1
1+1
4

Version History

Introduced before R2006a

See Also

`classperf` | `crossvalind` | `classify` | `grp2idx`

cleave

Cleave amino acid sequence with enzyme

Syntax

```
Fragments = cleave(SeqAA, Enzyme)
Fragments = cleave(SeqAA, PeptidePattern, Position)
[Fragments, CuttingSites] = cleave(...)
[Fragments, CuttingSites, Lengths] = cleave(...)
[Fragments, CuttingSites, Lengths, Missed] = cleave(...)
cleave(..., 'PartialDigest', PartialDigestValue, ...)
cleave(..., 'MissedSites', MissedSitesValue, ...)
cleave(..., 'Exception', ExceptionValue, ...)
```

Input Arguments

SeqAA	One of the following: • Character vector or string containing single-letter codes specifying an amino acid sequence. • Row vector of integers specifying an amino acid sequence. • MATLAB structure containing a `Sequence` field that contains an amino acid sequence, such as returned by `fastaread`, `getgenpept`, `genpeptread`, `getpdb`, or `pdbread`. Examples: `'ARN'` or `[1 2 3]`.
Enzyme	Character vector or string specifying a name or abbreviation code for an enzyme or compound for which the literature specifies a cleavage rule. **Tip** Use the `cleavelookup` function to display the names of enzymes and compounds in the cleavage rule library.
PeptidePattern	Short amino acid sequence to search for in *SeqAA*, a larger sequence. *PeptidePattern* can be any of the following: • Character vector or string • Vector of integers • Regular expression
Position	Integer from `0` to the length of the *PeptidePattern*, that specifies a position in the *PeptidePattern* to cleave. **Note** Position `0` corresponds to the N terminal end of *PeptidePattern*.

PartialDigestValue	Value from `0` to `1` (default) specifying the probability that a cleavage site will be cleaved.
MissedSitesValue	Nonnegative integer specifying the maximum number of missed cleavage sites. The output includes all possible peptide fragments that can result from missing *MissedSitesValue* or less cleavage sites. Default is `0`, which is equivalent to an ideal digestion.
ExceptionValue	Regular expression specifying an exception rule to the cleavage rule associated with *Enzyme*. By default, exception rules are only applied in the case of trypsin, and all other enzymes have no exception rule, which is specified as an empty character vector. To prevent the use of the default exceptions for trypsin, use an empty character vector as the exception rule. To see the regular expression for trypsin's exception rules, check the Cleave Lookup table.

Output Arguments

Fragments	Cell array of character vectors representing the fragments from the cleavage.
CuttingSites	Numeric vector containing indices representing the cleavage sites. **Note** The `cleave` function adds a `0` to the list, so `numel(`*CuttingSites*`)==numel(`*Fragments*`)`. Use *CuttingSites* `+ 1` to point to the first amino acid of every fragment respective to the original sequence.
Lengths	Numeric vector containing the length of each fragment.
Missed	Numeric vector containing the number of missed cleavage sites for every peptide fragment.

Description

Fragments `= cleave(`*SeqAA*`,` *Enzyme*`)` cuts *SeqAA*, an amino acid sequence, into parts at the cleavage sites specific for *Enzyme*, a character vector or string specifying a name or abbreviation code for an enzyme or compound for which the literature specifies a cleavage rule. It returns *Fragments*, a cell array of character vectors representing the fragments from the cleavage.

Tip Use the `cleavelookup` function to display the names of enzymes and compounds in the cleavage rule library.

Fragments `= cleave(`*SeqAA*`,` *PeptidePattern*`,` *Position*`)` cuts *SeqAA*, an amino acid sequence, into parts at the cleavage sites specified by a peptide pattern and position.

`[`*Fragments*`,` *CuttingSites*`] = cleave(...)` returns a numeric vector containing indices representing the cleavage sites.

Note The `cleave` function adds a `0` to the list, so `numel(`*`CuttingSites`*`)==numel(`*`Fragments`*`)`. Use *`CuttingSites`* + `1` to point to the first amino acid of every fragment respective to the original sequence.

`[`*`Fragments`*`, `*`CuttingSites`*`, `*`Lengths`*`] = cleave(...)` returns a numeric vector containing the length of each fragment.

`[`*`Fragments`*`, `*`CuttingSites`*`, `*`Lengths`*`, `*`Missed`*`] = cleave(...)` returns a numeric vector containing the number of missed cleavage sites for every fragment.

`cleave(..., '`*`PropertyName`*`', `*`PropertyValue`*`, ...)` calls `cleave` with optional properties that use property name/property value pairs. You can specify one or more properties in any order. Enclose each *`PropertyName`* in single quotation marks. Each *`PropertyName`* is case insensitive. These property name/property value pairs are as follows:

`cleave(..., 'PartialDigest', `*`PartialDigestValue`*`, ...)` simulates a partial digestion where *`PartialDigestValue`* is the probability of a cleavage site being cut. *`PartialDigestValue`* is a value from `0` to `1` (default).

This table lists some common proteases and their cleavage sites.

Protease	Peptide Pattern	Position
Aspartic acid N	`D`	1
Chymotrypsin	`[WYF](?!P)`	1
Glutamine C	`[ED](?!P)`	1
Lysine C	`[K](?!P)`	1
Trypsin	`[KR](?!P)`	1

`cleave(..., 'MissedSites', `*`MissedSitesValue`*`, ...)` returns all possible peptide fragments that can result from missing *`MissedSitesValue`* or less cleavage sites. *`MissedSitesValue`* is a nonnegative integer. Default is `0`, which is equivalent to an ideal digestion.

`cleave(..., 'Exception', `*`ExceptionValue`*`, ...)` specifies an exception rule to the cleavage rule associated with *`Enzyme`*. *`ExceptionValue`* is a regular expression. By default, exception rules are only applied in the case of trypsin, and all other enzymes have no exception rule, which is specified as an empty character vector. To prevent the use of the default exceptions for trypsin, specify an empty character vector as the exception rule.

Examples

Cleave a sequence

This example shows how to cleave a sequence using trypsin.

Retrieve a protein sequence from the GenPept database.

```
S = getgenpept('AAA59174');
```

Cleave the sequence using trypsin's cleavage rules and all known exceptions.

```
parts = cleave(S.Sequence,'trypsin');
```

Display the first ten fragments.

```
parts(1:10)
```

```
ans =

    'MGTGGR'
    'R'
    'GAAAAPLLVAVAALLLGAAGHLYPGEVCPGMDIR'
    'NNLTR'
    'LHELENCSVIEGHLQILLMFK'
    'TRPEDFR'
    'DLSFPK'
    'LIMITDYLLLFR'
    'VYGLESLK'
    'DLFPNLTVIR'
```

Cleave the sequence using trypsin's cleavage rules and a single specific exception rule.

```
parts = cleave(S.Sequence,'trypsin','exception','KD');
parts(1:10)
```

```
ans =

    'MGTGGR'
    'R'
    'GAAAAPLLVAVAALLLGAAGHLYPGEVCPGMDIR'
    'NNLTR'
    'LHELENCSVIEGHLQILLMFK'
    'TRPEDFR'
    'DLSFPK'
    'LIMITDYLLLFR'
    'VYGLESLKDLFPNLTVIR'
    'GSR'
```

Cleave the sequence using one of trypsin's cleavage rules, which is to cleave after K or R when the next residue is not P.

```
[parts, sites, lengths] = cleave(S.Sequence,'[KR](?!P)',1);
for i = 1:10
    fprintf('%5d%5d   %s\n',sites(i),lengths(i),parts{i})
end
```

```
    0    6   MGTGGR
    6    1   R
    7   34   GAAAAPLLVAVAALLLGAAGHLYPGEVCPGMDIR
   41    5   NNLTR
   46   21   LHELENCSVIEGHLQILLMFK
   67    7   TRPEDFR
   74    6   DLSFPK
   80   12   LIMITDYLLLFR
   92    8   VYGLESLK
  100   10   DLFPNLTVIR
```

Cut the sequence using trypsin, allowing for 1 missed cleavage site.

```
[parts2, sites2, lengths2, missed] = cleave(S.Sequence,'trypsin','missedsites',1);
```

Display the first 10 fragments that have 1 missed cleavage site.

```
idx = find(missed);
for i = 1:10
    fprintf('%5d%5d   %s\n',sites2(idx(i)),lengths2(idx(i)),parts2{idx(i)})
end
```

```
    0    7   MGTGGRR
    6   35   RGAAAAPLLVAVAALLLGAAGHLYPGEVCPGMDIR
    7   39   GAAAAPLLVAVAALLLGAAGHLYPGEVCPGMDIRNNLTR
   41   26   NNLTRLHELENCSVIEGHLQILLMFK
   46   28   LHELENCSVIEGHLQILLMFKTRPEDFR
   67   13   TRPEDFRDLSFPK
   74   18   DLSFPKLIMITDYLLLFR
   80   20   LIMITDYLLLFRVYGLESLK
   92   18   VYGLESLKDLFPNLTVIR
  100   13   DLFPNLTVIRGSR
```

Version History

Introduced before R2006a

See Also

cleavelookup | rebasecuts | restrict | regexp

cleavelookup

Find cleavage rule for enzyme or compound

Syntax

```
cleavelookup
cleavelookup('Code', CodeValue)
cleavelookup('Name', NameValue)
```

Arguments

CodeValue	Character vector specifying a code representing an abbreviation code for an enzyme or compound. For valid codes, see the table Cleave Lookup.
NameValue	Character vector specifying an enzyme or compound name. For valid names, see the table Cleave Lookup.

Description

`cleavelookup` displays a table of abbreviation codes, cleavage positions, cleavage patterns, and full names of enzymes and compounds for which cleavage rules are specified by the cleavage rule library. Trysin’s exception rules are also listed in the table. For more information, see the ExPASy PeptideCutter tool.

Cleave Lookup

Code	Position	Pattern	Full Name
ARG-C	1	R	ARG-C proteinase
ASP-N	2	D	ASP-N endopeptidase
BNPS	1	W	BNPS-Skatole
CASP1	1	(?<=[FWYL]\w[HAT])D(?=[^PEDQKR])	Caspase 1
CASP2	1	(?<=DVA)D(?=[^PEDQKR])	Caspase 2
CASP3	1	(?<=DMQ)D(?=[^PEDQKR])	Caspase 3
CASP4	1	(?<=LEV)D(?=[^PEDQKR])	Caspase 4
CASP5	1	(?<=[LW]EH)D	Caspase 5
CASP6	1	(?<=VE[HI])D(?=[^PEDQKR])	Caspase 6
CASP7	1	(?<=DEV)D(?=[^PEDQKR])	Caspase 7
CASP8	1	(?<=[IL]ET)D(?=[^PEDQKR])	Caspase 8
CASP9	1	(?<=LEH)D	Caspase 9
CASP10	1	(?<=IEA)D	Caspase 10
CH-HI	1	([FY](?=[^P]))|(W(?=[^MP]))	Chymotrypsin-high specificity
CH-LO	1	([FLY](?=[^P]))|(W(?=[^MP]))| (M(?=[^PY]))|(H(?=[^DMPW]))	Chymotrypsin- low specificity
CLOST	1	R	Clostripain
CNBR	1	M	CNBR
ELANE	1	[AV]	Neutrophil elastase
ENTKIN	1	(?<=[DE][DE][DE])K	Enterokinase
FACTXA	1	(?<=[AFGILTVM][DE]G)R	Factor XA
FORMIC	1	D	Formic acid
GLUEND	1	E	Glutamyl endopeptidase
GRANB	1	(?<=IEP)D	Granzyme B
HYDROX	1	N(?=G)	Hydroxylamine
IODOB	1	W	Iodosobenzoic acid
LYSC	1	K	Lysc
NLATEV	1	(?<=Y\w)Q(?=[GS])	NLA in tobacco etch virus
NTCB	1	C	NTCB
PEPS	1	((?<=[^HKR][^P])[^R](?=[FL][^P]))|((?<=[^HKR][^P])[FL](?=\w[^P]))	Pepsin PH = 1.3
PEPS2	1	((?<=[^HKR][^P])[^R](?=[FLWY][^P]))|((?<=[^HKR][^P])[FLWY](?=\w[^P]))	Pepsin PH > 2

Code	Position	Pattern	Full Name
PROEND	1	`(?<=[HKR])P(?=[^P])`	Proline endopeptidase
PROTK	1	`[AEFILTVWY]`	Proteinase K
STAPHP	1	`(?<=[^E])E`	Staphylococcal peptidase I
THERMO	1	`[^DE](?=[AFILMV])`	Thermolysin
THROMB	1	`((?<=\w\wG)R(?=G))| ((?<=[AFGILTVM][AFGILTVWA]P)R(?=[^DE][^DE]))`	Thrombin
TRYPS	1	`((?<=\w)[KR](?=[^P]))| ((?<=W)K(?=P))|((?<=M)R(?=P))`	Trypsin
XTRYPS	1	`((?<=C)K[DHY])|((?<=D)K(?=D))|((?<=R)R(?=[HR]))|((?<=C)R(?=K))`	Trypsin exceptions

`cleavelookup('Code',` *`CodeValue`*`)` displays the cleavage position, cleavage pattern, and full name of the enzyme or compound specified by *`CodeValue`*, a character vector specifying an abbreviation code.

`cleavelookup('Name',` *`NameValue`*`)` displays the cleavage position, cleavage pattern, and abbreviation code of the enzyme or compound specified by *`NameValue`*, a character vector specifying an enzyme or compound name.

Examples

Example 1.1. Using cleavelookup with an Enzyme Name

Display the cleavage position, cleavage pattern, and abbreviation code of the enzyme Caspase 1.

```
cleavelookup('name', 'CASPASE 1')
```

```
ans =

1    (?<=[FWYL]\w[HAT])D(?=[^PEDQKR])    CASP1
```

Example 1.2. Using cleavelookup with an Abbreviation Code

Display the cleavage position, cleavage pattern, and full name of the enzyme with a abbreviation code of CASP1.

```
cleavelookup('code', 'CASP1')
```

```
ans =

1    (?<=[FWYL]\w[HAT])D(?=[^PEDQKR])    CASPASE 1
```

Version History

Introduced in R2008b

See Also

`cleave` | `rebasecuts` | `restrict`

cluster (phytree)

Validate clusters in phylogenetic tree

Syntax

```
LeafClusters = cluster(Tree, Threshold)
[LeafClusters, NodeClusters] = cluster(Tree, Threshold)
[LeafClusters, NodeClusters, Branches] = cluster(Tree, Threshold)
cluster(..., 'Criterion', CriterionValue, ...)
cluster(..., 'MaxClust', MaxClustValue, ...)
cluster(..., 'Distances', DistancesValue, ...)
```

Input Arguments

Tree	Phylogenetic tree object created, such as created with the `phytree` constructor function.
Threshold	Scalar specifying a threshold value.
CriterionValue	Character vector or string specifying the criterion to determine the number of clusters as a function of the species pairwise distances. Choices are: • `'maximum'` (default) — Maximum within cluster pairwise distance (W_{max}). Cluster splitting stops when $W_{max} \le$ *Threshold*. • `'median'` — Median within cluster pairwise distance (W_{med}). Cluster splitting stops when $W_{med} \le$ *Threshold*. • `'average'` — Average within cluster pairwise distance (W_{avg}). Cluster splitting stops when $W_{avg} \le$ *Threshold*. • `'ratio'` — Between/within cluster pairwise distance ratio, defined as BW_{rat} = (trace(B)/(k - 1)) / (trace(W)/(n - k)) where B and W are the between- and within-scatter matrices, respectively. k is the number of clusters, and n is the number of species in the tree. Cluster splitting stops when $BW_{rat} \ge$ *Threshold*. • `'gain'` — Within cluster pairwise distance gain, defined as W_{gain} = (trace(W_{old})/ (trace(W) - 1) * (n - k - 1)) where W and W_{old} are the within-scatter matrices for k and k - 1, respectively. k is the number of clusters, and n is the number of species in the tree. Cluster splitting stops when $W_{gain} \le$ *Threshold*. • `'silhouette'` — Average silhouette width (SW_{avg}). SW_{avg} ranges from `-1` to `+1`. Cluster splitting stops when $SW_{avg} \ge$ *Threshold*. For more information, see `silhouette`.

MaxClustValue	Positive integer specifying the maximum number of possible clusters for the tested partitions. Default is the number of leaves in the tree. **Tip** When using the `'maximum'`, `'median'`, or `'average'` criteria, set *Threshold* to `[]` (empty) to force `cluster` to return *MaxClustValue* clusters. It does so because such metrics monotonically decrease as *k* increases. **Tip** When using the `'ratio'`, `'gain'`, or `'silhouette'` criteria, you may find it hard to estimate an appropriate *Threshold* in advance. Set *Threshold* to `[]` (empty) to find the optimal number of clusters below *MaxClustValue*. Also, set *MaxClustValue* to a small value to avoid expensive computation due to testing all possible number of clusters.
DistancesValue	Matrix of pairwise distances, such as returned by the `seqpdist` function, containing biological distances between each pair of sequences. `cluster` substitutes this matrix for the patristic distances in *Tree*. For example, this matrix can contain the real sample pairwise distances.

Output Arguments

LeafClusters	Column vector containing a cluster index for each species (leaf) in *Tree*, a phylogenetic tree object.
NodeClusters	Column vector containing the cluster index for each leaf node and branch node in *Tree*. **Tip** Use the *LeafClusters* or *NodeClusters* output vectors with the handle returned by the `plot` method to modify graphic elements of the phylogenetic tree object. For more information, see “Examples” on page 1-553.
Branches	Two-column matrix containing, for each step in the algorithm, the index of the branch being considered and the value of the criterion. Each row corresponds to a step in the algorithm. The first column contains branch indices, and the second column contains criterion values. **Tip** To obtain the whole curve of the criterion versus the number of clusters in *Branches*, set *Threshold* to `[]` (empty) and do not specify a *MaxClustValue*. Be aware that computation of some criteria can be computationally intensive.

Description

`*LeafClusters* = cluster(*Tree*, *Threshold*)` returns a column vector containing a cluster index for each species (leaf) in a phylogenetic tree object. It determines the optimal number of clusters as follows:

- Starting with two clusters ($k = 2$), selects the partition that optimizes the criterion specified by the `'Criterion'` property

- Increments *k* by `1` and again selects the optimal partition
- Continues incrementing *k* and selecting the optimal partition until a criterion value = *`Threshold`* or *k* = the maximum number of clusters (that is, number of leaves)
- From all possible *k* values, selects the *k* value whose partition optimizes the criterion

`[`*`LeafClusters`*`, `*`NodeClusters`*`] = cluster(`*`Tree`*`, `*`Threshold`*`)` returns a column vector containing the cluster index for each leaf node and branch node in *`Tree`*.

`[`*`LeafClusters`*`, `*`NodeClusters`*`, `*`Branches`*`] = cluster(`*`Tree`*`, `*`Threshold`*`)` returns a two-column matrix containing, for each step in the algorithm, the index of the branch being considered and the value of the criterion. Each row corresponds to a step in the algorithm. The first column contains branch indices, and the second column contains criterion values.

`cluster(..., '`*`PropertyName`*`', `*`PropertyValue`*`, ...)` calls `cluster` with optional properties that use property name/property value pairs. You can specify one or more properties in any order. Enclose each *`PropertyName`* in single quotation marks. Each *`PropertyName`* is case insensitive. These property name/property value pairs are as follows:.

`cluster(..., 'Criterion', `*`CriterionValue`*`, ...)` specifies the criterion to determine the number of clusters as a function of the species pairwise distances.

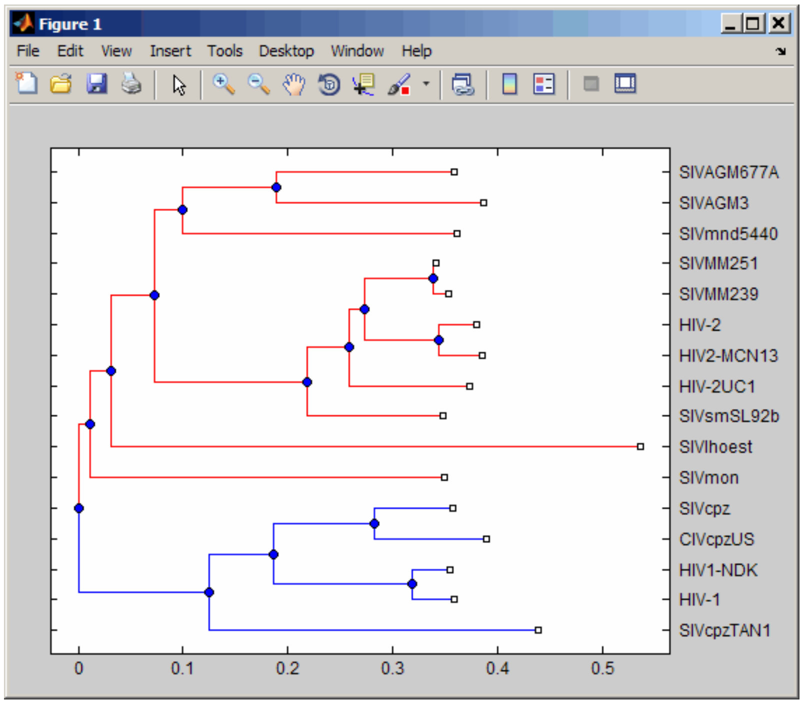
`cluster(..., 'MaxClust', `*`MaxClustValue`*`, ...)` specifies the maximum number of possible clusters for the tested partitions. Default is the number of leaves in the tree.

`cluster(..., 'Distances', `*`DistancesValue`*`, ...)` substitutes the patristic distances in *`Tree`* with a user-provided pairwise distance matrix.

Examples

Validate the clusters in a phylogenetic tree:

```
% Read sequences from a multiple alignment file into a MATLAB
% structure
gagaa = multialignread('aagag.aln');

% Build a phylogenetic tree from the sequences
gag_tree = seqneighjoin(seqpdist(gagaa),'equivar',gagaa);

% Validate the clusters in the tree and find the best partition
% using the 'gain' criterion
[i,j] = cluster(gag_tree,[],'criterion','gain','maxclust',10);

% Use the returned vector of indices to color the branches of each
% cluster in a plot of the tree
h = plot(gag_tree);
set(h.BranchLines(j==2),'Color','b')
set(h.BranchLines(j==1),'Color','r')
```



References

[1] Dudoit, S. and Fridlyan, J. (2002). A prediction-based resampling method for estimating the number of clusters in a dataset. Genome Biology *3(7)*, research 0036.1–0036.21.

[2] Theodoridis, S. and Koutroumbas, K. (1999). Pattern Recognition (Academic Press), pp. 434–435.

[3] Kaufman, L. and Rousseeuw, P.J. (1990). Finding Groups in Data: An Introduction to Cluster Analysis (New York, Wiley).

[4] Calinski, R. and Harabasz, J. (1974). A dendrite method for cluster analysis. Commun Statistics *3*, 1–27.

[5] Hartigan, J.A. (1985). Statistical theory in clustering. J Classification *2*, 63–76.

See Also

`phytree` | `phytreeread` | `phytreeviewer` | `seqlinkage` | `seqneighjoin` | `seqpdist` | `plot` | `view` | `cluster` | `silhouette`

Topics

phytree object on page 1-1449

clustergram

Object containing hierarchical clustering analysis data

Description

The `clustergram` function creates a `clustergram` object. The object contains hierarchical clustering analysis data that you can view in a heatmap and dendrogram.

Creation

Syntax

```
clustergram(data)
clustergram(data,Name,Value)
```

Description

`cgObj = clustergram(data)` performs hierarchical clustering analysis on the values in `data`. The returned clustergram object `cgObj` contains analysis data and displays a dendrogram and heatmap.

`cgObj = clustergram(data,Name,Value)` sets the object properties on page 1-555 using name-value pairs. For example, `clustergram(data,'Standardize','column')` standardizes the values along the columns of data. You can specify multiple name-value pairs. Enclose each property name in quotes.

Input Arguments

`data` — Source data
DataMatrix object | numeric matrix

Source data, specified as a DataMatrix object on page 1-734 or numeric matrix. Typically, if the matrix contains gene expression data, each row corresponds to a gene and each column corresponds to a sample.

Name-Value Pair Arguments

Use comma-separated name-value pair arguments to set the object properties. Enclose each property name in single quotes.

Example: `cg = clustergram(data,'Colormap',redbluecmap,'Annotate',true)`

Properties

`Standardize` — Dimension for standardizing data values
`'none'` (default) | `'row'` | `'column'` | `3` | `2` | `1`

Dimension for standardizing data values, specified as a character vector, string, or positive integer. Choices are:

- `'column'` or `1` — Standardize along the columns of data.
- `'row'` or `2` — Standardize along the rows of data.
- `'none'` or `3` — Do not standardize.

If you specify `'column'` or `'row'`, the function transforms the standardized values so that the mean is 0 and the standard deviation is 1 in the specified dimension.

Example: `'column'`

Data Types: `double` | `char` | `string`

`Symmetric` — Flag to make the heatmap color scale symmetric around zero
`true` (default) | `false`

Flag to make the heatmap color scale symmetric around zero, specified as `true` or `false`.

Example: `false`

Data Types: `logical`

`ImputeFun` — Name of function or function handle to impute missing data
character vector | cell array

Name of a function or function handle to impute missing data, specified as a character vector or cell array. If you specify a cell array, the first element must be the name of a function or function handle, and the remaining elements must be name-value pairs used as inputs to the function. Missing data points are colored gray in the heatmap.

If data points are missing, use this property to impute the missing values.. Otherwise, the `clustergram` function errors.

Example: `'func1'`

Data Types: `char`

`Colormap` — Heatmap colors
`redgreencmap` (default) | matrix | name of function handle

heatmap colors, specified as a three-column (*M*-by-3) matrix of red-green-blue (RGB) values or the name of a function handle that returns a colormap, such as `redgreencmap` or `redbluecmap`.

The default colormap is `redgreencmap`, in which red represents values above the mean, black represents the mean, and green represents values below the mean of a row (gene) across all columns (samples).

Example: `redbluecmap`

Data Types: `double` | `char`

`ColumnLabels` — Column labels
`[1x0 double]` (default) | string vector | cell array of character vectors | numeric vector

Column labels, specified as a string vector, cell array of character vectors, or numeric vector. The size of the vector must match the number of columns in the input `data`.

If the number of column labels is 200 or more, the labels do not appear in the clustergram plot.

Example: `["sample1","sample2","sample3"]`

Data Types: `double` | `string` | `cell`

`RowLabels` — Row labels
`[]` (default) | string vector | cell array of character vectors | numeric vector

Row labels, specified as a string vector, cell array of character vectors, or numeric vector. The size of the vector must match the number of rows in the input `data`.

If the number of row labels is 200 or more, the labels do not appear in the clustergram plot.

Example: `["gene1","gene2","gene3"]`

Data Types: `double` | `string` | `cell`

`ColumnLabelsRotate` — Orientation of column labels
`90` (default) | numeric scalar

Orientation of column labels, specified as a numeric scalar. Specify the value of rotation in degrees (positive angles cause counterclockwise rotation).

Example: `30`

Data Types: `double`

`RowLabelsRotate` — Orientation of row labels
0 (default) | numeric scalar

Orientation of row labels, specified as a numeric scalar. Specify the value of rotation in degrees (positive angles cause counterclockwise rotation).

Example: `30`

Data Types: `double`

`Annotate` — Flag to display data values in heatmap
`false` (default) | `true`

Flag to display data values in the heatmap, specified as `true` or `false`.

Example: `true`

Data Types: `logical`

`AnnotPrecision` — Display precision of data values
2 (default) | numeric scalar

Display precision of data values in the heatmap, specified as a numeric scalar. The default number of digits of precision is 2.

Example: 3

Data Types: `double`

`LabelsWithMarkers` — Flag to display colored markers for row and column labels
`false` (default) | `true`

Flag to display colored markers instead of colored text for the row and column labels, specified as `true` or `false`.

Example: `true`

Data Types: `logical`

`AnnotColor` — Text color of displayed data values
`'w'` (default) | character vector | string | three-element numeric vector

Text color of displayed data values in the heatmap, specified as a character vector, string, or three-element numeric vector. For example, to use cyan, you can enter `[0 1 1]`, `'c'`, `"c"`, `"cyan"`, or `'cyan'`. For details, see "Color Options" on page 1-571.

Example: `'red'`

Data Types: `char` | `string` | `double`

`DisplayRange` — Display range of standardize values
3 (default) | positive scalar

Display range of standardize values, specified as a positive scalar.

The default value `3`means that there is a color variation for values between `-3` and `3`, but values greater than `3` are the same color as `3`, and values less than `-3` are the same color as `-3`.

For example, if you specify `redgreencmap` for the `'Colormap'` property, pure red represents values greater than or equal to the specified display range value and pure green represents values less than or equal to the negative of the specified display range value.

Example: `3`

Data Types: `double`

`ColumnLabelsColor` — Color information for column labels
`[]` (default) | structure | structure array

Warning This property will be removed in a future release. Set `LabelsWithMarkers` to `true` for colored markers instead of colored texts.

Color information for column labels, specified as a structure or structure array.

For a single structure, you must specify the following fields.

- `Labels` — Cell array of character vectors specifying column labels listed in the `ColumnLabels` property.
- `Colors` — Character vector or string specifying a color for the column labels. If this field is empty, the default color (black) is used.

For a structure array, you must specify a single element in each field for each structure.

- `Labels` — Character vector or string specifying a column label listed in the `ColumnLabels` property.
- `Colors` — Character vector or string specifying a color for the column labels. If this field is empty, the default color (black) is used.

For more information on specifying colors, see "Color Options" on page 1-571.

Data Types: `struct`

`RowLabelsColor` — Color information for row labels
[ ] (default) | structure | structure array

Warning This property will be removed in a future release. Set `LabelsWithMarkers` to `true` for colored markers instead of colored texts.

Color information for row labels, specified as a structure or structure array.

For a single structure, you must specify the following fields.

- `Labels` — Cell array of character vectors specifying row labels listed in the `RowLabels` property.
- `Colors` — Character vector or string specifying a color for the row labels. If this field is empty, the default color (black) is used.

For a structure array, you must specify a single element in each field for each structure.

- `Labels` — Character vector or string specifying a row label listed in the `RowLabels` property.
- `Colors` — Character vector or string specifying a color for the row labels. If this field is empty, the default color (black) is used.

For more information on specifying colors, see "Color Options" on page 1-571.

`Cluster` — Dimension for data clustering
`'all'` (default) | `1` | `2` | `3` | `'column'` | `'row'`

Dimension for data clustering, specified as a positive integer, character vector, or string. Choices are:

- `'column'` or `1` — Cluster along the columns of data only, which results in clustered rows.
- `'row'` or `2` — Cluster along the rows of data only, which results in clustered columns.
- `'all'` or `3` — Cluster along the columns of data, then cluster along the rows of row-clustered data.

Example: 2

Data Types: `double` | `char` | `string`

`ColumnGroupMarker` — Information for annotating groups of columns
structure | structure array

Information for annotating groups of columns, specified as a structure or structure array.

If you specify a single structure, each field must contain a cell array of elements. If you specify a structure array, each structure must have a single element in each field.

The fields are :

- `GroupNumber` — Scalar specifying the column group number to annotate.
- `Annotation` — Character vector specifying text to annotate the column group.
- `Color` — Character vector or three-element vector of RGB values specifying a color to label the column group. For more information on specifying colors, see "Color Options" on page 1-571. If this field is empty, the default value is `'blue'`.

Data Types: `struct`

`ColumnPDist` — Distance metric to pass to `pdist` function
`'euclidean'` (default) | character vector | cell array

Distance metric to pass to the `pdist` function to calculate the pairwise distances between columns, specified as a character vector or cell array. Specify a cell array if the distance metric requires extra arguments. For example, to use the Minkowski distance with an exponent *p*, specify `{'minkowski',p}`.

Example: `'jaccard'`

Data Types: `char` | `cell`

`Dendrogram` — Color threshold information to pass to `dendrogram` function
scalar | two-element numeric vector | character vector | cell array of character vectors

Color threshold information to pass to the `dendrogram` function to create a dendrogram plot, specified as a scalar, two-element numeric vector, character vector, or cell array of character vectors. This option sets the `'ColorThreshold'` property of the dendrogram plot. If you specify a two-element numeric vector or cell array, the first element is for the rows, and the second element is for the columns.

Data Types: `double` | `cell`

`DisplayRatio` — Ratio of space that row and column dendrograms occupy
`1/5` (default) | scalar between `0` and `1` | two-element vector

Ratio of space that the row and column dendrograms occupy relative to the heatmap, specified as a scalar between `0` and `1` or two-element vector. If you specify a scalar, the function uses it as the ratio for both row and column dendrograms. If you specify a two-element vector, the function uses the first element for the ratio of the row dendrogram width to the heatmap width, and the second element for the ratio of the column dendrogram height to the heatmap height. The second element is ignored for one-dimensional clustergrams.

Example: `0.5`

Data Types: `double`

`Linkage` — Linkage method to create hierarchical cluster tree
`'average'` (default) | character vector | two-element cell array of character vectors

Linkage method passed to the `linkage` function to create the hierarchical cluster tree for rows and columns, specified as a character vector or two-element cell array of character vectors. If you specify a cell array, the function uses the first element for linkage between rows, and the second element for linkage between columns.

Example: `'centroid'`

Data Types: `char` | `cell`

`LogTrans` — Flag to $\log_2$ transform data
`false` (default) | `true`

Flag to $\log_2$ transform the data from natural scale, specified as `true` or `false`.

Example: `true`

Data Types: `logical`

`OptimalLeafOrder` — Flag to calculate optimal leaf order
`true` | `false`

Flag to calculate the optimal leaf order that maximizes the similarity between neighboring leaves, specified as `true` or `false`. The default value depends on the size of the input `data`. If the number of rows or columns in `data` exceeds 1500, the default value is `false`. Otherwise, the default value is `true`.

Disabling the optimal leaf ordering calculation can be useful when working with large datasets because this calculation consumes a lot of memory and time.

Example: `true`

Data Types: `logical`

`RowGroupMarker` — Information for annotating groups of rows
structure | structure array

Information for annotating groups of rows, specified as a structure or structure array.

If you specify a single structure, each field must contain a cell array of elements. If you specify a structure array, each structure must have a single element in each field.

The fields are

- `GroupNumber` — Scalar specifying the column group number to annotate.
- `Annotation` — Character vector specifying text to annotate the column group.
- `Color` — Character vector or three-element vector of RGB values specifying a color to label the column group. For more information on specifying colors, see “Color Options” on page 1-571. If this field is empty, the default value is `'blue'`.

Data Types: `struct`

`RowPDist` — Distance metric to pass to `pdist` function
`'euclidean'` (default) | character vector | cell array

Distance metric to pass to the `pdist` function to calculate the pairwise distances between rows, specified as a character vector or cell array. Specify a cell array if the distance metric requires extra arguments. For example, to use the Minkowski distance with an exponent *p*, specify `{'minkowski',`*p*`}`.

Example: `'jaccard'`

Data Types: `char` | `cell`

`ShowDendrogram` — Flag to show dendrogram tree diagrams with clustergram
`'on'` (default) | `'off'`

Flag to show the dendrogram tree diagrams with the clustergram, specified as `'on'` or `'off'`.

Example: `'off'`

Data Types: `char`

Object Functions

view	Display heatmap or clustergram

plot	Render heatmap or clustergram
addTitle	Add title to heatmap or clustergram
addXLabel	Label x-axis of heatmap or clustergram
addYLabel	Label y-axis of heatmap or clustergram
clusterGroup	Select cluster group

Examples

Perform Hierarchical Clustering on Gene Expression Data

Load microarray data containing gene expression levels of *Saccharomyces cerevisiae* (yeast) during the metabolic shift from fermentation to respiration [1].

```
load filteredyeastdata
```

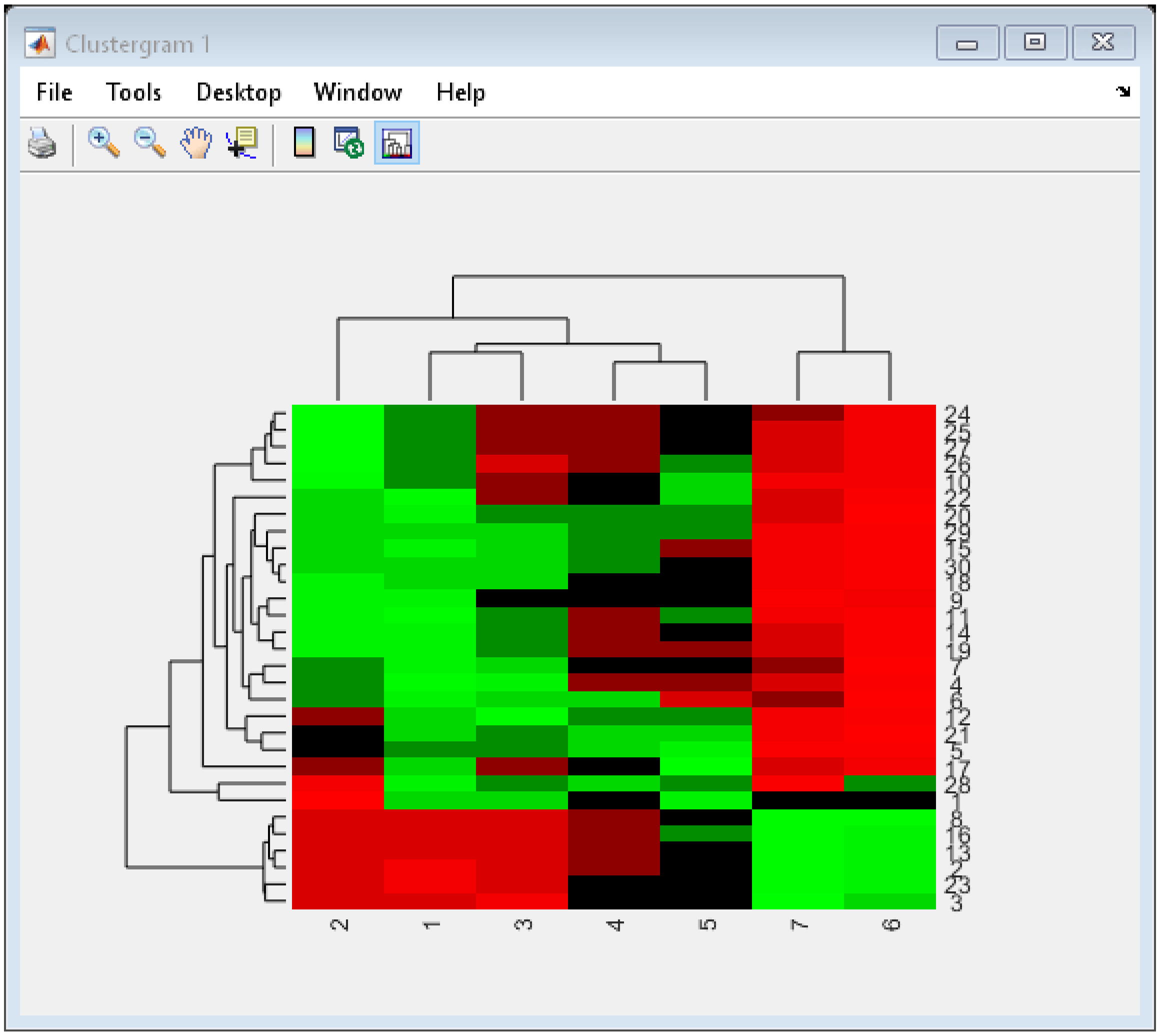
This MAT file includes three variables, which are added to the MATLAB® workspace:

- `yeastvalues` - A matrix of gene expression data from *Saccharomyces* -_cerevisiae_ during the metabolic shift from fermentation to respiration - `genes` - A cell array of GenBank® accession numbers for labeling the rows in `yeastvalues` - `times` - A vector of time values for labeling the columns in `yeastvalues`

Create a clustergram object to display the heat map from the gene expression data in the first 30 rows of the `yeastvalues` matrix and standardize along the rows of data.

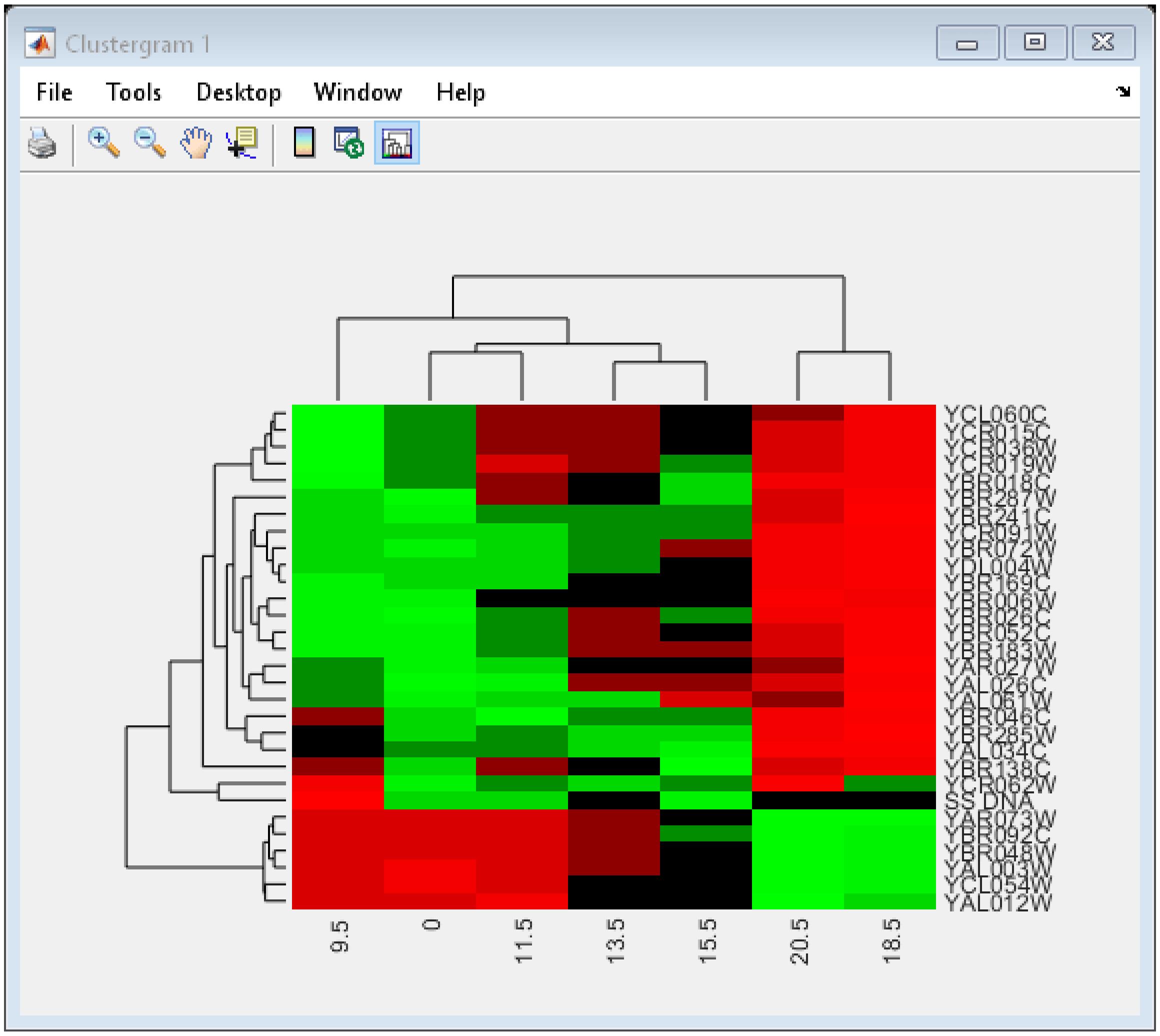
```
cgo = clustergram(yeastvalues(1:30,:),'Standardize','Row')
```

```
Clustergram object with 30 rows of nodes and 7 columns of nodes.
```



Use the `set` method and the `genes` and `times` vectors to add meaningful row and column labels to the clustergram.

```
set(cgo,'RowLabels',genes(1:30),'ColumnLabels',times)
```



Add a color bar to the clustergram by clicking the `Insert Colorbar` button on the toolbar.

View a data tip containing the intensity value, row label, and column label for a specific area of the heat map by clicking the `Data Cursor` button on the toolbar, then clicking an area in the heat map. To delete this data tip, right-click it, then select `Delete Current Datatip`.

Display intensity values for each area of the heat map by clicking the **Annotate** button on the toolbar. Click the **Annotate** button again to remove the intensity values.

```
Tip: If the amount of data is large enough, the cells within the clustergram
are too small to display the intensity annotations. Zoom in to see the
intensity annotations.
```

Remove the dendrogram tree diagrams from the figure by clicking the **Show Dendrogram** button on the toolbar. Click it again to display the dendrograms.

Use the `get` method to display the properties of the clustergram object, `cgo`.

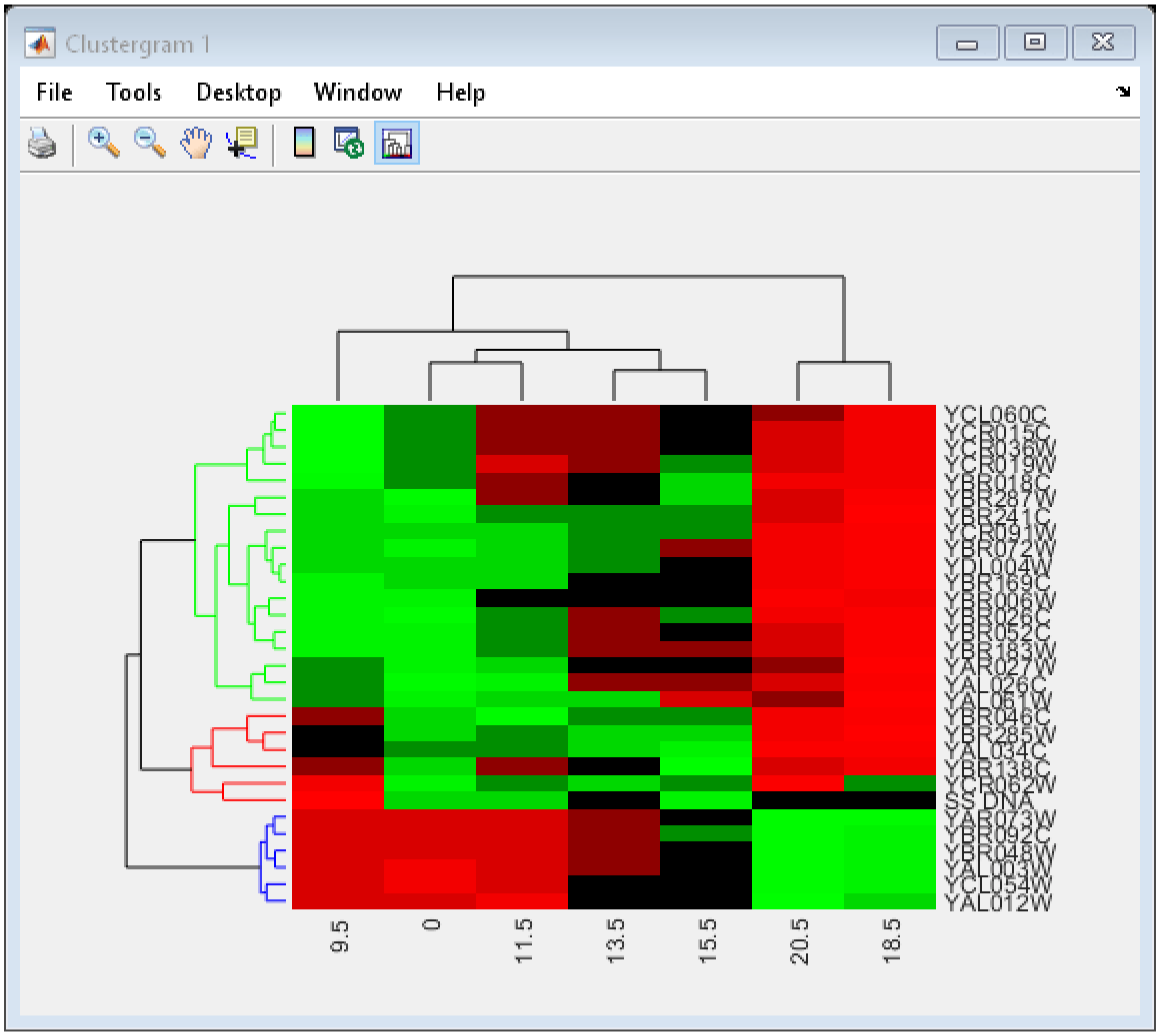
```
get(cgo)
```

```
               Cluster: 'ALL'
              RowPDist: {'Euclidean'}
```

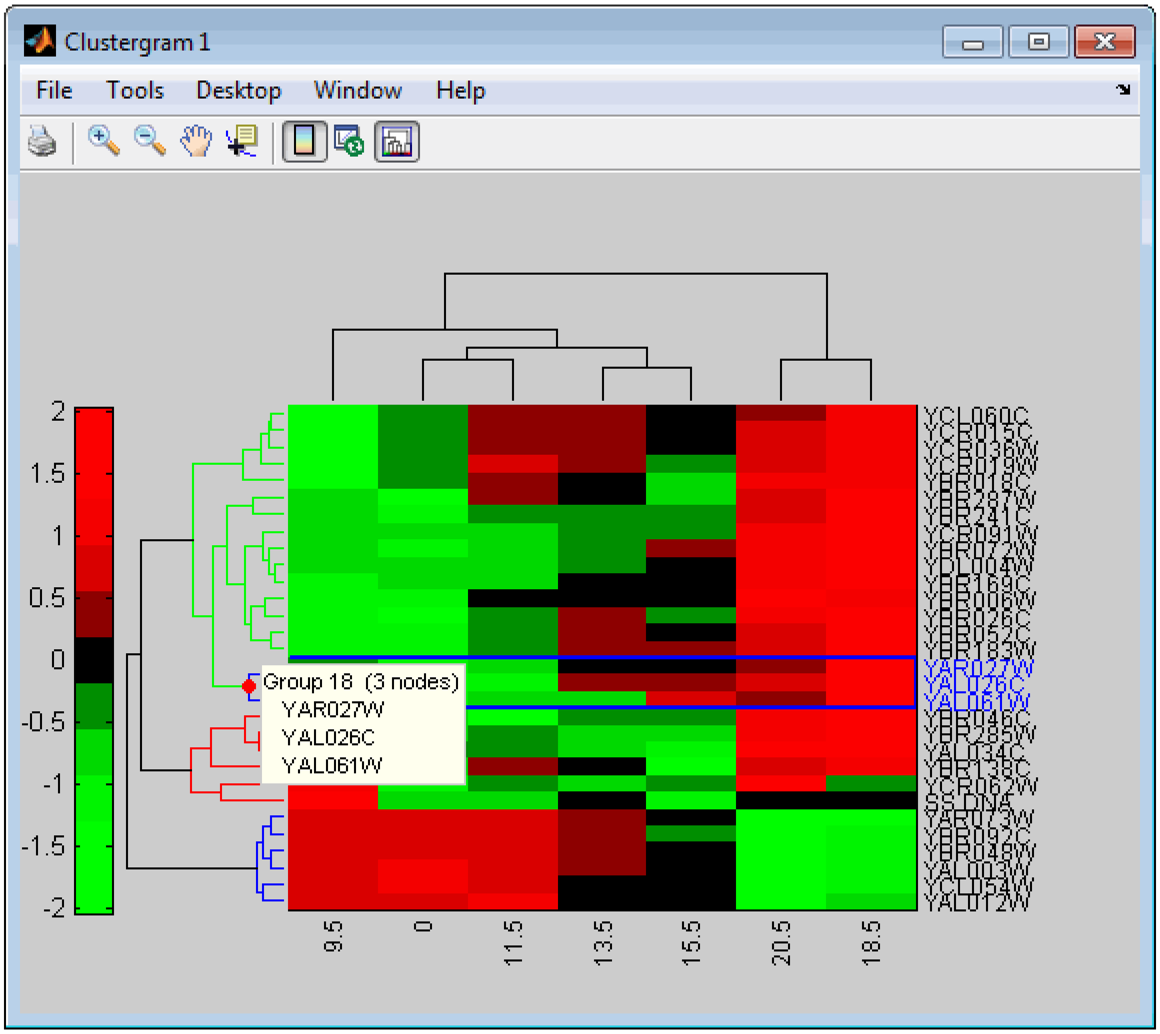
```
          ColumnPDist: {'Euclidean'}
              Linkage: {'Average'}
           Dendrogram: {}
      OptimalLeafOrder: 1
             LogTrans: 0
         DisplayRatio: [0.2000 0.2000]
       RowGroupMarker: []
    ColumnGroupMarker: []
       ShowDendrogram: 'on'
          Standardize: 'ROW'
            Symmetric: 1
         DisplayRange: 3
             Colormap: [11x3 double]
            ImputeFun: []
         ColumnLabels: {1x7 cell}
            RowLabels: {30x1 cell}
   ColumnLabelsRotate: 90
      RowLabelsRotate: 0
             Annotate: 'off'
       AnnotPrecision: 2
           AnnotColor: 'w'
    ColumnLabelsColor: []
       RowLabelsColor: []
    LabelsWithMarkers: 0
```

Change the clustering parameters by changing the linkage method and changing the color of the groups of nodes in the dendrogram whose linkage is less than a threshold of 3.

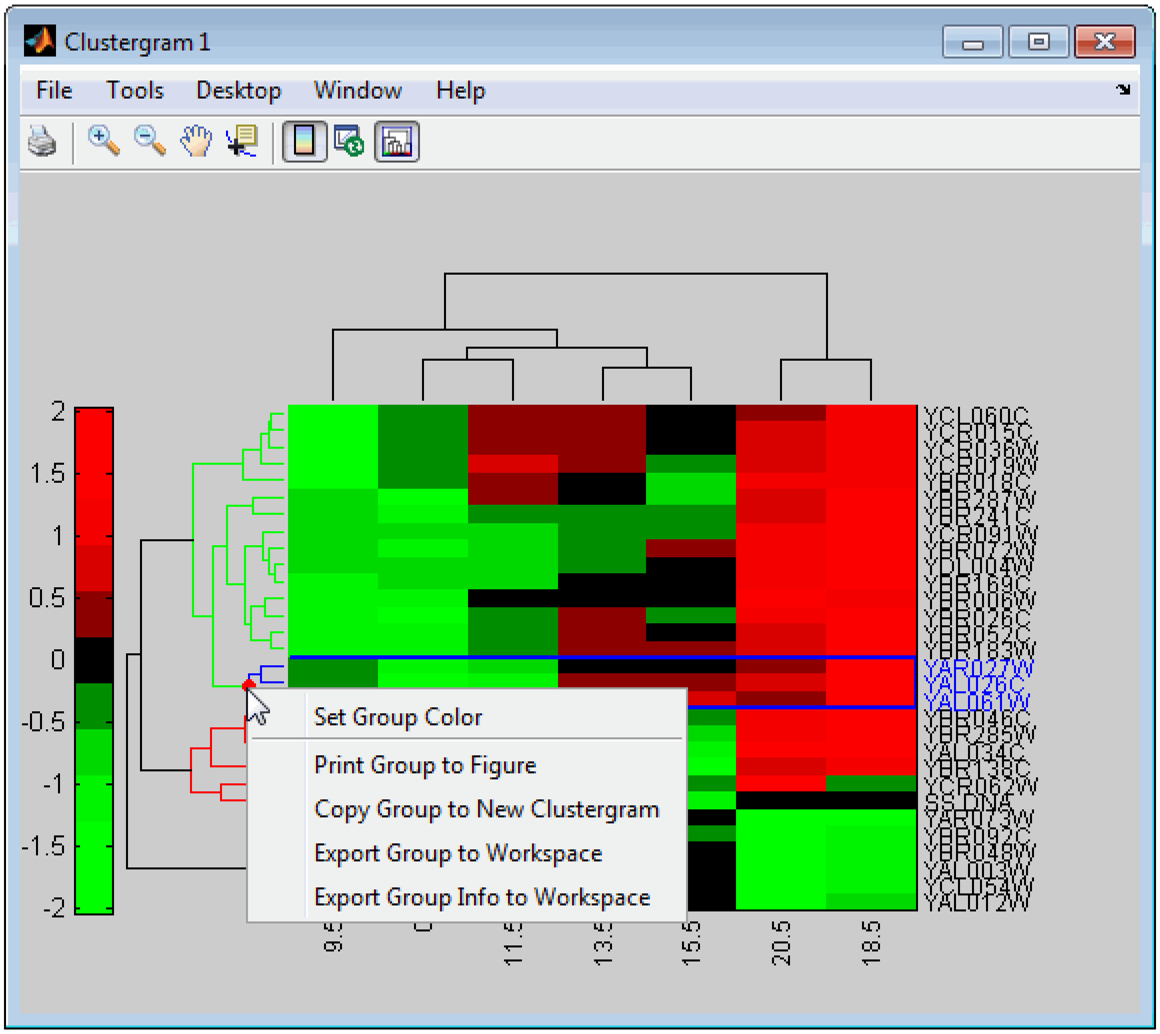
```
set(cgo,'Linkage','complete','Dendrogram',3)
```



Place the cursor on a branch node in the dendrogram to highlight (in blue) the group associated with it. Press and hold the mouse button to display a data tip listing the group number and the nodes (genes or samples) in the group.



Right-click a branch node in the dendrogram to display a menu of options.



The following options are available:

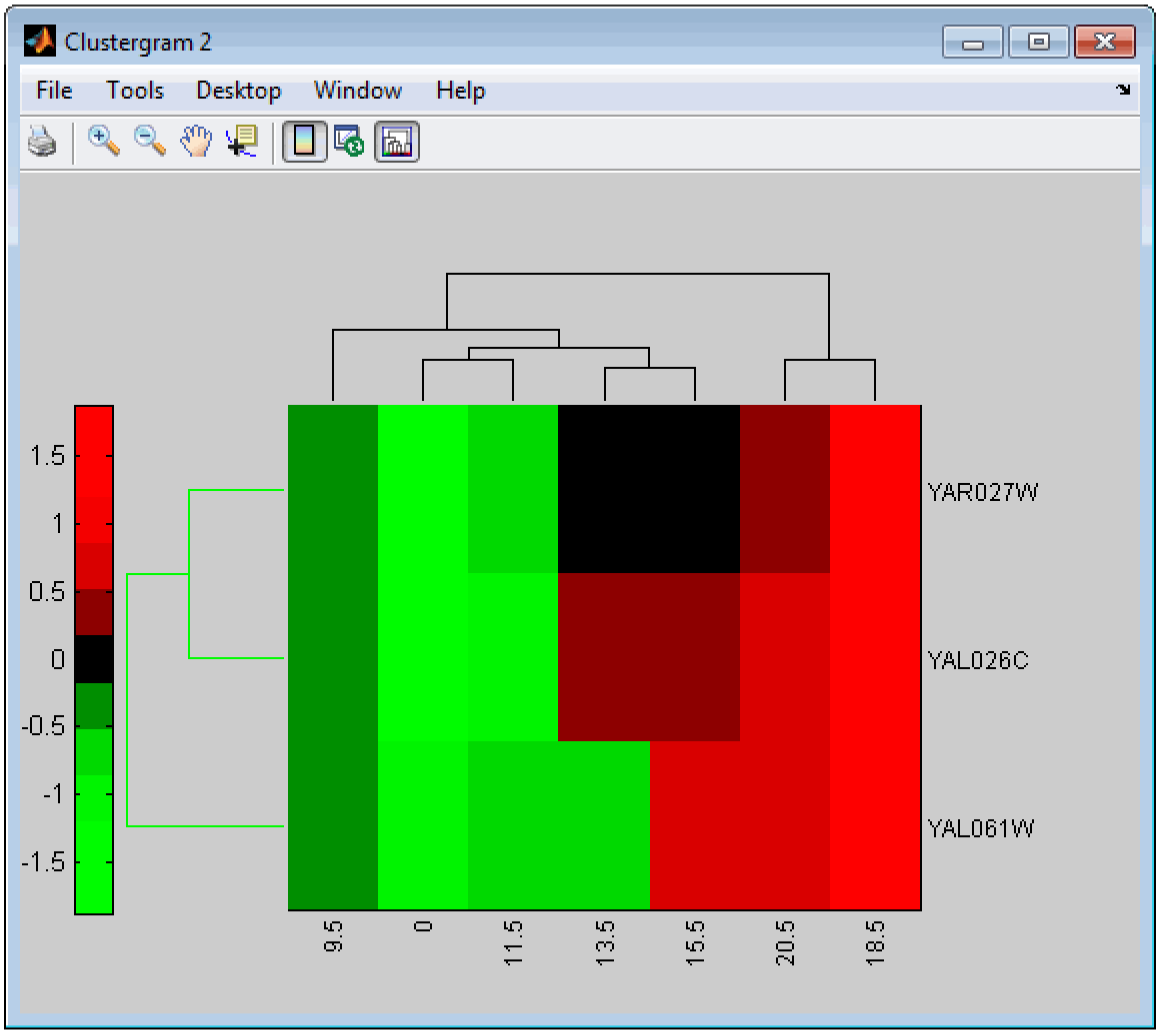
- **Set Group Color** - Change the cluster group color. - **Print Group to Figure** - Print the group to a figure window. - **Copy Group to New Clustergram** - Copy the group to a new clustergram window. - **Export Group to Workspace** - Create a clustergram object of the group in the MATLAB workspace. - **Export Group Info to Workspace** - Create a structure containing information about the group in the MATLAB workspace. The structure contains these fields:

- `GroupNames` - Cell array of character vectors containing the names of the row or column groups. - `RowNodeNames` - Cell array of character vectors containing the names of the row nodes. - `ColumnNodeNames` - Cell array of character vectors containing the names of the column nodes. - `ExprValues` - An M-by-N matrix of intensity values, where M and N are the number of row nodes and of column nodes respectively. If the matrix contains gene expression data, typically each row corresponds to a gene and each column corresponds to sample.

Create a clustergram object for Group 18 in the MATLAB workspace. Right-click Group 18, then select **Export Group to Workspace** . In the **Export to Workspace** dialog box, type `Group18`, then click **OK** .

Use the `view` method to view the clustergram object, `Group18`.

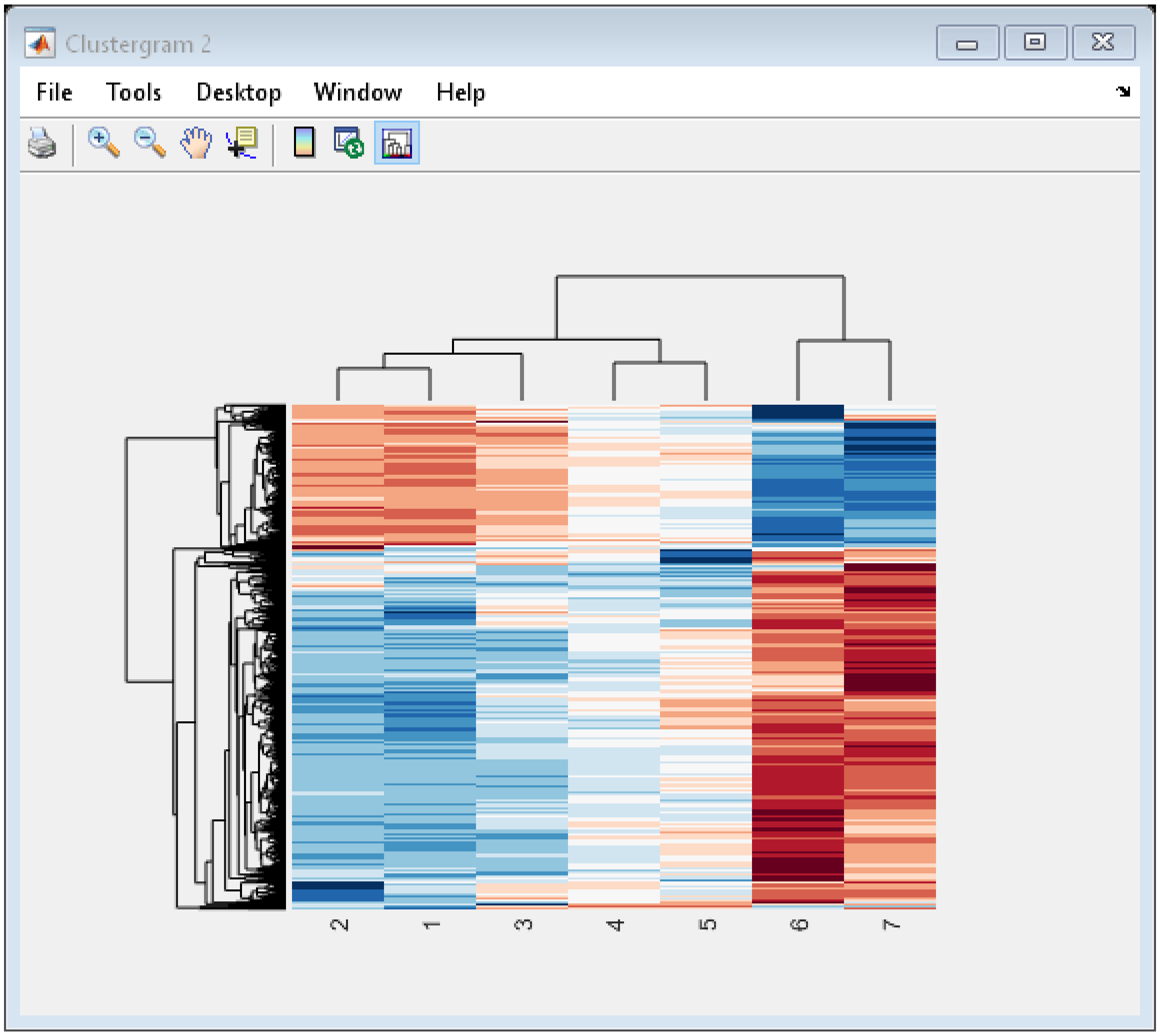
```
view(Group18)
```



View all the gene expression data using a diverging red and blue colormap and standardize along the rows of data.

```
cgo_all = clustergram(yeastvalues,'Colormap',redbluecmap,'Standardize','Row')
```

```
Clustergram object with 614 rows of nodes and 7 columns of nodes.
```

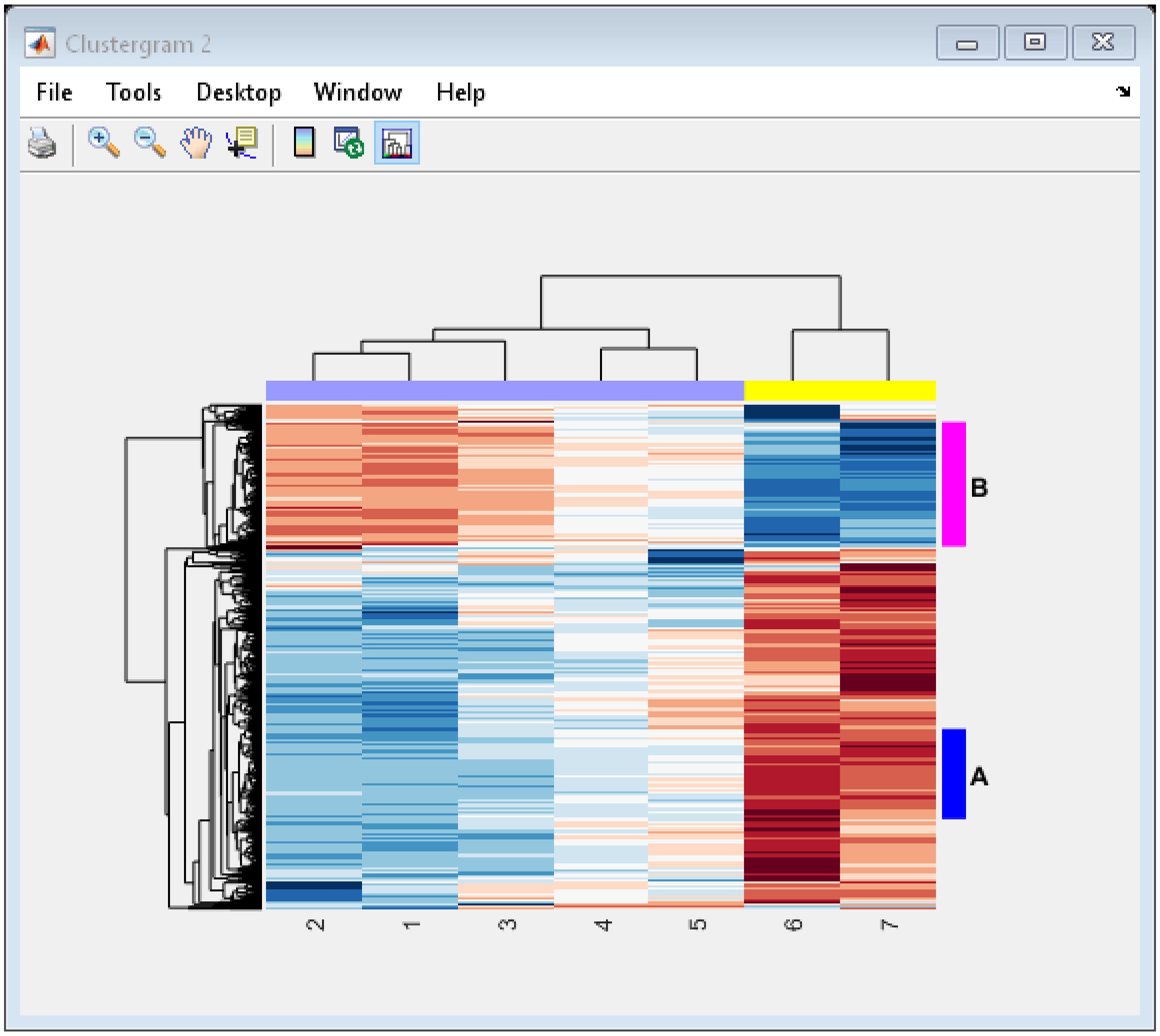


Create structure arrays to specify marker colors and annotations for two groups of rows (510 and 593) and two groups of columns (4 and 5).

```
rm = struct('GroupNumber',{510,593},'Annotation',{'A','B'},...
     'Color',{'b','m'});
cm = struct('GroupNumber',{4,5},'Annotation',{'Time1','Time2'},...
     'Color',{[1 1 0],[0.6 0.6 1]});
```

Use the `RowGroupMarker` and `ColumnGroupMarker` properties to add the color markers and annotations to the clustergram.

```
set(cgo_all,'RowGroupMarker',rm,'ColumnGroupMarker',cm)
```



More About

Color Options

The following lists the predefined colors and their RGB triplet equivalents. The short names and long names are character vectors that specify one of eight preset colors. The RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color; the intensities must be in the range [0 1].

RGB Triplet	Short Name	Long Name
`[1 1 0]`	`y`	`yellow`
`[1 0 1]`	`m`	`magenta`
`[0 1 1]`	`c`	`cyan`
`[1 0 0]`	`r`	`red`
`[0 1 0]`	`g`	`green`
`[0 0 1]`	`b`	`blue`

RGB Triplet	Short Name	Long Name
`[1 1 1]`	`w`	`white`
`[0 0 0]`	`k`	`black`

Version History

Introduced before R2006a

References

[1] DeRisi, J. L. “Exploring the Metabolic and Genetic Control of Gene Expression on a Genomic Scale.” *Science* 278, no. 5338 (October 24, 1997): 680–86.

See Also

`redbluecmap` | `redgreencmap` | `HeatMap`

clusterGroup

Select cluster group

Syntax

```
clusterGroup(cgObj1,groupIndex,dim)
cgObj2 = clusterGroup(cgObj1,groupIndex,dim)
cgObj2 = clusterGroup(cgObj1,groupIndex,dim,'InfoOnly',tf_InfoOnly)
cgObj2 = clusterGroup(cgObj1,groupIndex,dim,'Color',colorChoice)
```

Description

`clusterGroup(cgObj1,groupIndex,dim)` selects and highlights the cluster group specified by `groupIndex` along the row or column dimension `dim` in the Clustergram window.

`cgObj2 = clusterGroup(cgObj1,groupIndex,dim)` creates and returns the clustergram object `cgObj2` for the specified cluster group. This syntax is equivalent to selecting the `Export Group to Workspace` option from the context menu after right-clicking a group in the Clustergram window.

`cgObj2 = clusterGroup(cgObj1,groupIndex,dim,'InfoOnly',tf_InfoOnly)` specifies whether to return the cluster group as a structure or a clustergram object.

`cgObj2 = clusterGroup(cgObj1,groupIndex,dim,'Color',colorChoice)` specifies the color to use to highlight the dendrogram of the selected cluster group.

Examples

Select Cluster Group

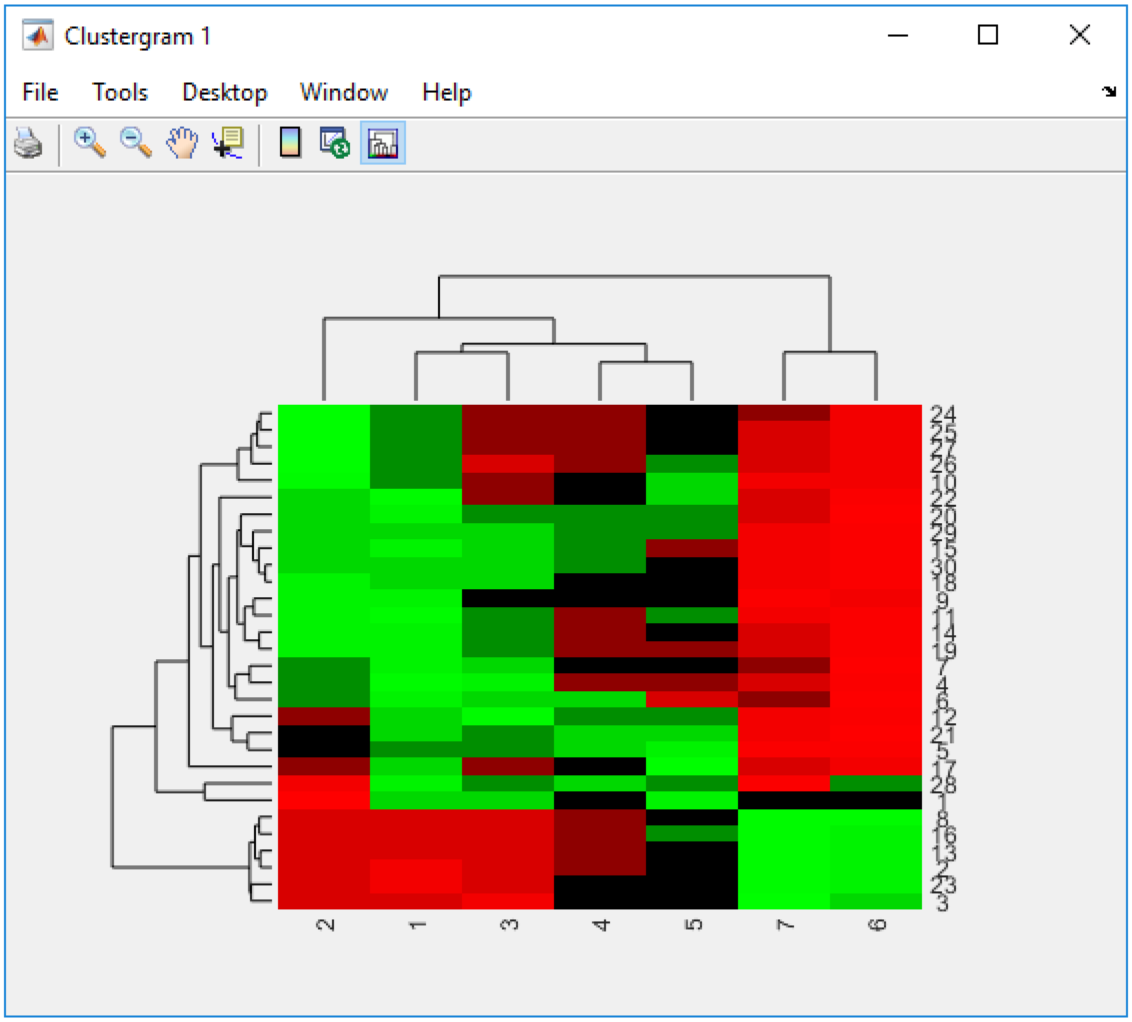
Load microarray data containing some measurements of gene expression levels.

```
load filteredyeastdata
```

Create a clustergram object and display a heatmap from the gene expression data.

```
cgo = clustergram(yeastvalues(1:30,:),'Standardize','Row')
```

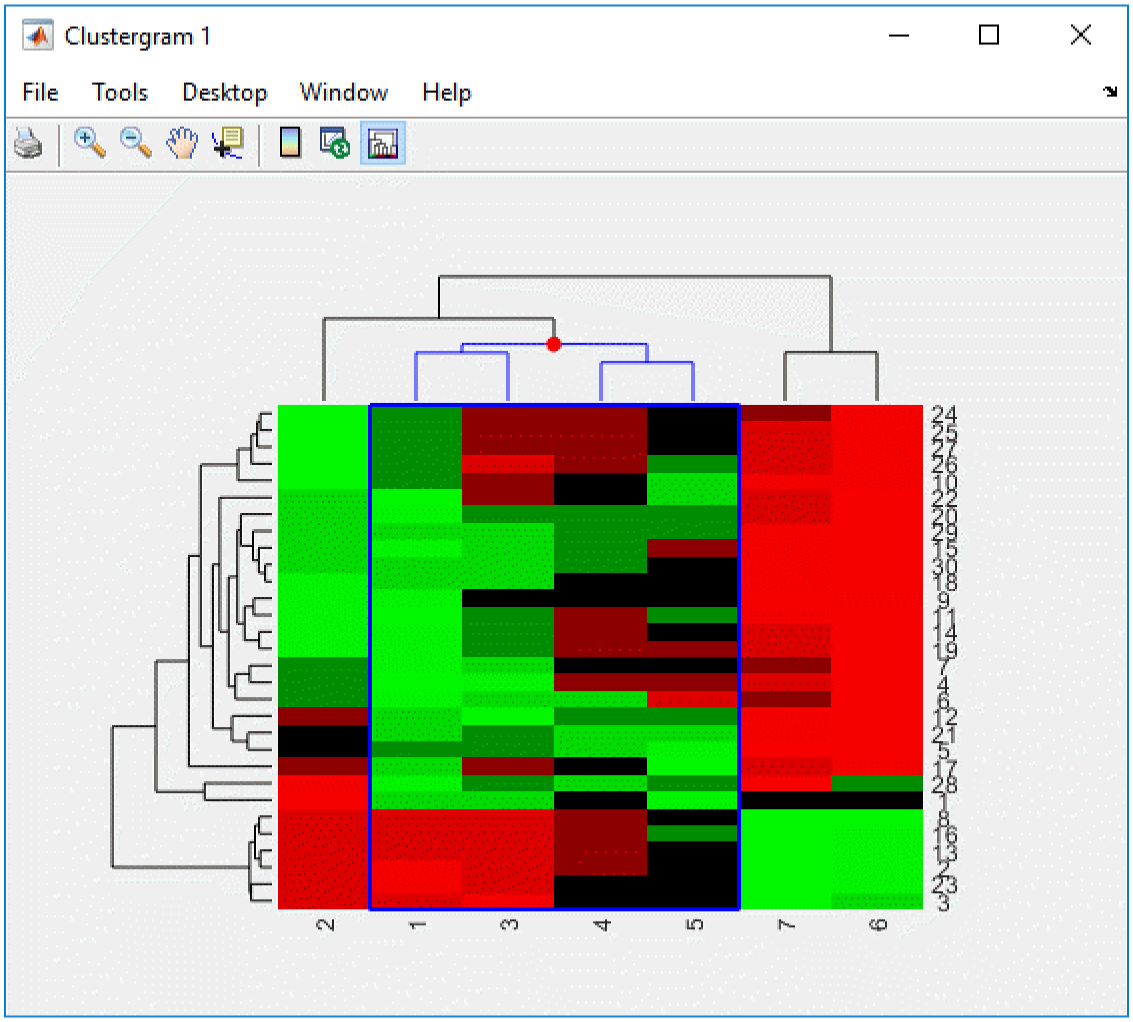
```
Clustergram object with 30 rows of nodes and 7 columns of nodes.
```



From the command line, use the `clusterGroup` function to select and highlight the Group 4 column cluster in the Clustergram window.

```
cgroup4 = clusterGroup(cgo,4,'column')
```

```
Clustergram object with 30(30) rows of nodes and 4(7) columns of nodes.
```



Input Arguments

`cgObj1` — Clustergram object
`clustergram` object

Clustergram object, specified as a `clustergram` object.

`groupIndex` — Group index for cluster group
positive integer

Group index for a cluster group in `cgObj1`, specified as a positive integer.

Example: 4

Data Types: `double`

`dim` — Dimension of cluster group
`'row'` | `'column'`

Dimension of the cluster group, specified as `'row'` or `'column'`.

Example: `'row'`

Data Types: `char`

`tf_InfoOnly` — Flag to return structure
`false` (default) | `true`

Flag to return a structure instead of a clustergram object, specified as `true` or `false`.

Example: `true`

Data Types: `logical`

`colorChoice` — Color to highlight dendrogram
character vector | three-element numeric vector

Color to highlight the dendrogram of the selected cluster group, specified as a character vector or three-element numeric vector of RGB values. For example, to use cyan, specify `[0 1 1]`, `'c'`, or `'cyan'`.

For more information on specifying colors, see “Color Options” on page 1-576.

Example: `'red'`

Data Types: `double` | `char`

Output Arguments

`cgObj2` — Selected cluster group
`clustergram` object | structure

Selected cluster group, returned as a clustergram object or structure. If you specify `'InfoOnly'` as `true`, the function returns `cgObj2` as a structure.

More About

Color Options

The following lists the predefined colors and their RGB triplet equivalents. The short names and long names are character vectors that specify one of eight preset colors. The RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color; the intensities must be in the range [0 1].

RGB Triplet	Short Name	Long Name
[1 1 0]	y	yellow
[1 0 1]	m	magenta
[0 1 1]	c	cyan
[1 0 0]	r	red
[0 1 0]	g	green
[0 0 1]	b	blue
[1 1 1]	w	white

RGB Triplet	Short Name	Long Name
`[0 0 0]`	`k`	`black`

Version History

Introduced in R2009b

See Also

`HeatMap` | `clustergram`

codonbias

Calculate codon frequency for each amino acid coded for in nucleotide sequence

Syntax

```
CodonFreq = codonbias(SeqNT)

CodonFreq = codonbias(SeqNT, ...'GeneticCode', GeneticCodeValue, ...)
CodonFreq = codonbias(SeqNT, ...'Frame', FrameValue, ...)
CodonFreq = codonbias(SeqNT, ...'Reverse', ReverseValue, ...)
CodonFreq = codonbias(SeqNT, ...'Ambiguous', AmbiguousValue, ...)
CodonFreq = codonbias(SeqNT, ...'Pie', PieValue, ...)
```

Input Arguments

SeqNT	One of the following: • Character vector or string specifying a nucleotide sequence • Row vector of integers specifying a nucleotide sequence • MATLAB structure containing a `Sequence` field that contains a nucleotide sequence, such as returned by `fastaread`, `fastqread`, `emblread`, `getembl`, `genbankread`, or `getgenbank` Valid characters include `A`, `C`, `G`, `T`, and `U`. `codonbias` does not count ambiguous nucleotides or gaps.
GeneticCodeValue	Integer, character vector, or string specifying a genetic code number or code name from the table Genetic Code. Default is `1` or `'Standard'`. **Tip** If you use a code name, you can truncate the name to the first two letters of the name.
FrameValue	Integer specifying a reading frame in the nucleotide sequence. Choices are `1` (default), `2`, or `3`.
ReverseValue	Controls the return of the codon frequency for the reverse complement sequence of the nucleotide sequence specified by *SeqNT*. Choices are `true` or `false` (default).

AmbiguousValue	Character vector or string specifying how to treat codons containing ambiguous nucleotide characters (`R`, `Y`, `K`, `M`, `S`, `W`, `B`, `D`, `H`, `V`, or `N`). Choices are: • `'ignore'` (default) — Skips codons containing ambiguous characters • `'prorate'` — Counts codons containing ambiguous characters and distributes them proportionately in the appropriate codon fields. For example, the counts for the codon `ART` are distributed evenly between the `AAT` and `AGT` fields. • `'warn'` — Skips codons containing ambiguous characters and displays a warning.
PieValue	Controls the creation of a figure of 20 pie charts, one for each amino acid. Choices are `true` or `false` (default).

Output Arguments

CodonFreq	MATLAB structure containing a field for each amino acid, each of which contains the associated codon frequencies as percentages.

Description

Many amino acids are coded by two or more nucleic acid codons. However, the probability that a specific codon (from all possible codons for an amino acid) is used to code an amino acid varies between sequences. Knowing the frequency of each codon in a protein coding sequence for each amino acid is a useful statistic.

`*CodonFreq* = codonbias(*SeqNT*)` calculates the codon frequency in percent for each amino acid coded for in *SeqNT*, a nucleotide sequence, and returns the results in *CodonFreq*, a MATLAB structure containing a field for each amino acid.

`*CodonFreq* = codonbias(*SeqNT*, ...'*PropertyName*', *PropertyValue*, ...)` calls `codonbias` with optional properties that use property name/property value pairs. You can specify one or more properties in any order. Each *PropertyName* must be enclosed in single quotation marks and is case insensitive. These property name/property value pairs are as follows:

`*CodonFreq* = codonbias(*SeqNT*, ...'GeneticCode', *GeneticCodeValue*, ...)` specifies a genetic code. Choices for *GeneticCodeValue* are an integer, character vector, or string specifying a code number or code name from the table Genetic Code. If you use a code name, you can truncate the name to the first two characters of the name. Default is `1` or `'Standard'`.

Tip If you use a code name, you can truncate the name to the first two letters of the name.

`*CodonFreq* = codonbias(*SeqNT*, ...'Frame', *FrameValue*, ...)` calculates the codon frequency in the reading frame specified by *FrameValue*, which can be `1` (default), `2`, or `3`.

`*CodonFreq* = codonbias(*SeqNT*, ...'Reverse', *ReverseValue*, ...)` controls the return of the codon frequency for the reverse complement of the nucleotide sequence specified by *SeqNT*. Choices are `true` or `false` (default).

`CodonFreq = codonbias(SeqNT, ...'Ambiguous', AmbiguousValue, ...)` specifies how to treat codons containing ambiguous nucleotide characters. Choices are `'ignore'` (default), `'prorate'`, and `'warn'`.

`CodonFreq = codonbias(SeqNT, ...'Pie', PieValue, ...)` controls the creation of a figure of 20 pie charts, one for each amino acid. Choices are `true` or `false` (default).

Genetic Code

Code Number	Code Name
1	`Standard`
2	`Vertebrate Mitochondrial`
3	`Yeast Mitochondrial`
4	`Mold`, `Protozoan`, `Coelenterate Mitochondrial`, and `Mycoplasma/Spiroplasma`
5	`Invertebrate Mitochondrial`
6	`Ciliate`, `Dasycladacean`, and `Hexamita Nuclear`
9	`Echinoderm Mitochondrial`
10	`Euplotid Nuclear`
11	`Bacterial` and `Plant Plastid`
12	`Alternative Yeast Nuclear`
13	`Ascidian Mitochondrial`
14	`Flatworm Mitochondrial`
15	`Blepharisma Nuclear`
16	`Chlorophycean Mitochondrial`
21	`Trematode Mitochondrial`
22	`Scenedesmus Obliquus Mitochondrial`
23	`Thraustochytrium Mitochondrial`

Examples

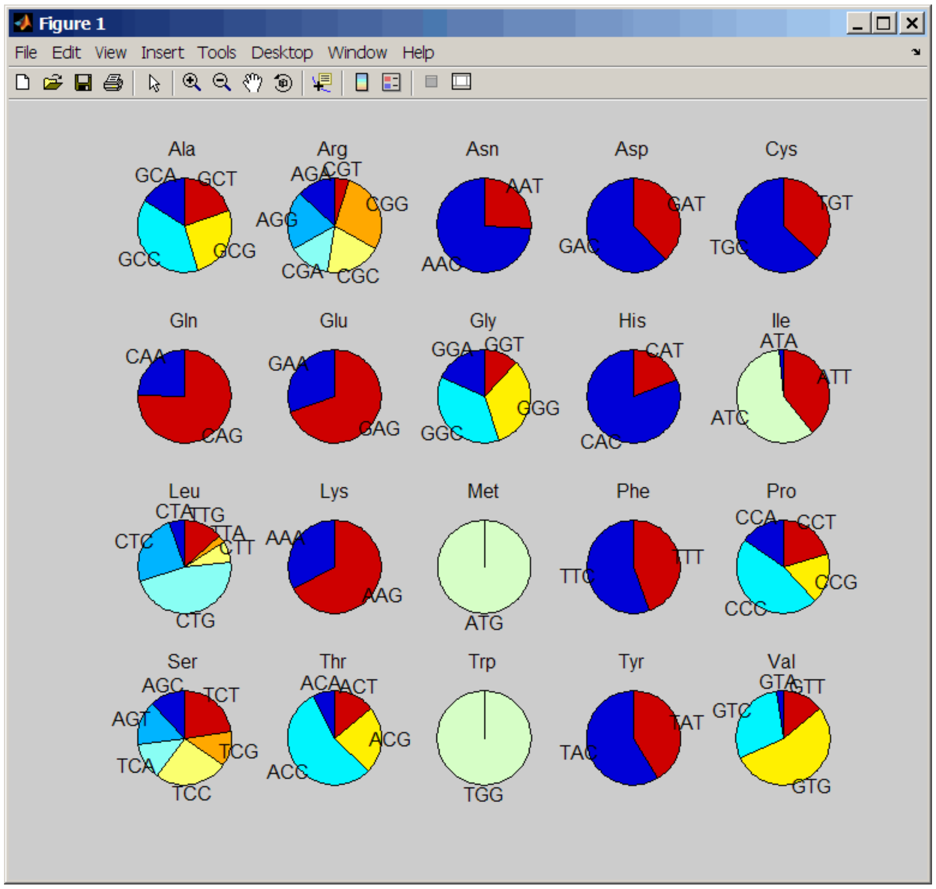
Calculate Codon Frequency for Each Amino Acid

Import a nucleotide sequence from the GenBank database into the MATLAB software. For example, retrieve the DNA sequence that codes for a human insulin receptor.

```
S = getgenbank('M10051');
```

Calculate the codon frequency for each amino acid coded for by the DNA sequence, and then plot the results.

```
cb = codonbias(S.Sequence,'PIE',true)
```



Get the codon frequency for the alanine (A) amino acid.

```
cb.Ala
```

```
ans =

    Codon: {'GCA'  "GCC'  "GCG'  'GCT'}
     Freq: [0.1600 0.3867 0.2533 02000]
```

Version History

Introduced before R2006a

See Also

aminolookup | codoncount | geneticcode | nt2aa

codoncount

Count codons in nucleotide sequence

Syntax

```
Codons = codoncount(SeqNT)
[Codons, CodonArray] = codoncount(SeqNT)

... = codoncount(SeqNT, ...'Frame', FrameValue, ...)
... = codoncount(SeqNT, ...'Reverse', ReverseValue, ...)
... = codoncount(SeqNT, ...'Ambiguous', AmbiguousValue, ...)
... = codoncount(SeqNT, ...'Figure', FigureValue, ...)
... = codoncount(SeqNT, ...'GeneticCode', GeneticCodeValue, ...)
```

Input Arguments

SeqNT	One of the following: • Character vector or string specifying a nucleotide sequence. For valid letter codes, see the table Mapping Nucleotide Letter Codes to Integers • Row vector of integers specifying a nucleotide sequence. For valid integers, see the table Mapping Nucleotide Integers to Letter Codes • MATLAB structure containing a `Sequence` field that contains a nucleotide sequence, such as returned by `fastaread`, `fastqread`, `emblread`, `getembl`, `genbankread`, or `getgenbank`. Examples: `'ACGT'` or `[1 2 3 4]`
FrameValue	Integer specifying a reading frame in the nucleotide sequence. Choices are `1` (default), `2`, or `3`.
ReverseValue	Controls the return of the codon count for the reverse complement sequence of the nucleotide sequence specified by *SeqNT*. Choices are `true` or `false` (default).

AmbiguousValue	Character vector or string specifying how to treat codons containing ambiguous nucleotide characters (`R`, `Y`, `K`, `M`, `S`, `W`, `B`, `D`, `H`, `V`, or `N`). Choices are: • `'ignore'` (default) — Skips codons containing ambiguous characters • `'bundle'` — Counts codons containing ambiguous characters and reports the total count in the `Ambiguous` field of the *`Codons`* output structure. • `'prorate'` — Counts codons containing ambiguous characters and distributes them proportionately in the appropriate codon fields containing standard nucleotide characters. For example, the counts for the codon `ART` are distributed evenly between the `AAT` and `AGT` fields. • `'warn'` — Skips codons containing ambiguous characters and displays a warning.
FigureValue	Controls the display of a heat map of the codon counts. Choices are `true` or `false` (default).
GeneticCodeValue	Integer, character vector, or string specifying a genetic code number or code name from the table Genetic Code. Default is `1` or `'Standard'`. You can also specify `'None'`. **Tip** If you use a code name, you can truncate the name to the first two letters of the name.

Output Arguments

Codons	MATLAB structure containing fields for the 64 possible codons (`AAA`, `AAC`, `AAG`, ..., `TTG`, `TTT`), which contain the codon counts in *`SeqNT`*.
CodonArray	A 4-by-4-by-4 array containing the raw count data for each codon. The three dimensions correspond to the three positions in the codon, and the indices to each element are represented by `1 = A`, `2 = C`, `3 = G`, and `4 = T`. For example, the element `(2,3,4)` in the array contains the number of `CGT` codons.

Description

`Codons` `= codoncount(`*`SeqNT`*`)` counts the codons in *`SeqNT`*, a nucleotide sequence, and returns the codon counts in *`Codons`*, a MATLAB structure containing fields for the 64 possible codons (`AAA`, `AAC`, `AAG`, ..., `TTG`, `TTT`).

- For sequences that have codons containing the character `U`, these codons are added to the corresponding codons containing a `T`.
- If the sequence contains gaps indicated by a hyphen (`-`), then codons containing gaps are ignored.
- If the sequence contains unrecognized characters, then codons containing these characters are ignored, and the following warning message appears:

```
Warning: Unknown symbols appear in the sequence. These will be ignored.
```

`[Codons, CodonArray] = codoncount(SeqNT)` returns `CodonArray`, a 4-by-4-by-4 array containing the raw count data for each codon. The three dimensions correspond to the three positions in the codon, and the indices to each element are represented by `1 = A`, `2 = C`, `3 = G`, and `4 = T`. For example, the element `(2,3,4)` in the array contains the number of `CGT` codons.

`... = codoncount(SeqNT, ...'PropertyName', PropertyValue, ...)` calls `codoncount` with optional properties that use property name/property value pairs. You can specify one or more properties in any order. Each `PropertyName` must be enclosed in single quotation marks and is case insensitive. These property name/property value pairs are as follows:

`... = codoncount(SeqNT, ...'Frame', FrameValue, ...)` counts the codons in the reading frame specified by `FrameValue`, which can be `1` (default), `2`, or `3`.

`... = codoncount(SeqNT, ...'Reverse', ReverseValue, ...)` controls the return of the codon count for the reverse complement sequence of `SeqNT`. Choices are `true` or `false` (default).

`... = codoncount(SeqNT, ...'Ambiguous', AmbiguousValue, ...)` specifies how to treat codons containing ambiguous nucleotide characters. Choices are:

- `'ignore'` (default)
- `'bundle'`
- `'prorate'`
- `'warn'`

`... = codoncount(SeqNT, ...'Figure', FigureValue, ...)` controls the display of a heat map of the codon counts. Choices are `true` or `false` (default).

`... = codoncount(SeqNT, ...'GeneticCode', GeneticCodeValue, ...)` controls the overlay of a grid on the heat map figure. The grid groups the synonymous codons according to `GeneticCodeValue`.

Examples

Count codons in a nucleotide sequence

```
seq = randseq(1000);
codons = codoncount(seq)
```

```
codons = struct with fields:
    AAA: 11
    AAC: 5
    AAG: 8
    AAT: 6
    ACA: 6
    ACC: 7
    ACG: 4
    ACT: 7
    AGA: 6
    AGC: 9
    AGG: 5
    AGT: 2
    ATA: 6
    ATC: 4
    ATG: 4
    ATT: 6
    CAA: 3
    CAC: 5
    CAG: 7
    CAT: 10
    CCA: 5
    CCC: 4
    CCG: 8
    CCT: 5
    CGA: 7
    CGC: 6
    CGG: 5
    CGT: 5
    CTA: 4
    CTC: 7
    CTG: 4
    CTT: 5
    GAA: 5
    GAC: 6
    GAG: 5
    GAT: 4
    GCA: 3
    GCC: 2
    GCG: 8
    GCT: 5
    GGA: 6
    GGC: 7
    GGG: 10
    GGT: 4
    GTA: 2
    GTC: 6
    GTG: 5
    GTT: 2
    TAA: 2
    TAC: 4
    TAG: 1
    TAT: 4
    TCA: 6
    TCC: 2
    TCG: 5
    TCT: 5
    TGA: 4
    TGC: 1
    TGG: 5
    TGT: 8
    TTA: 6
    TTC: 1
    TTG: 8
    TTT: 5
```

Count the codons in the second frame for the reverse complement of a sequence.

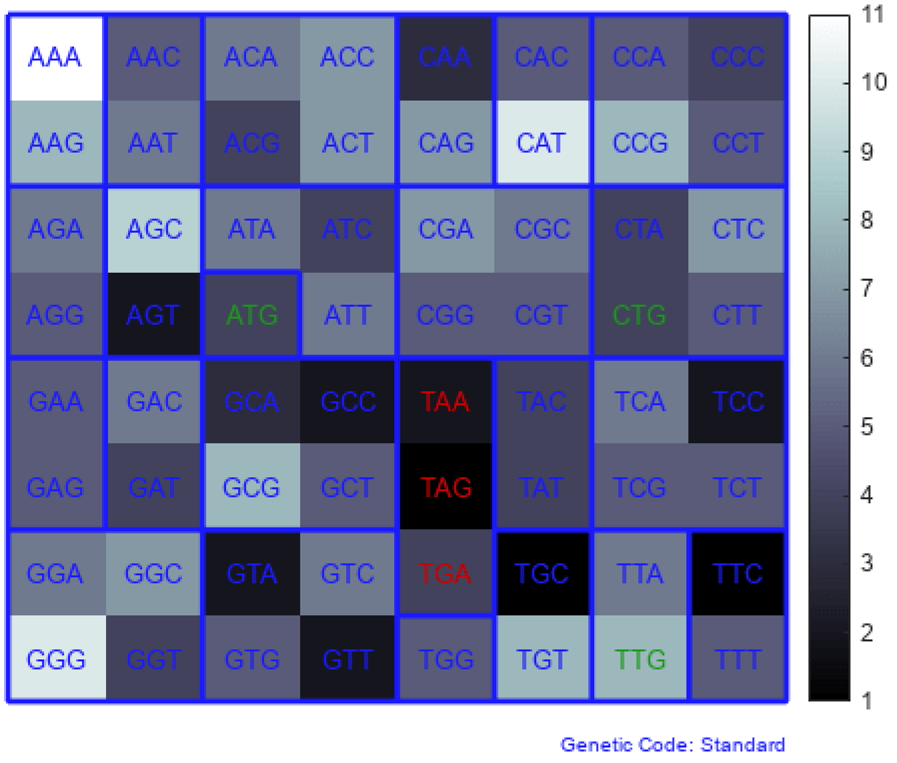
```
r2codons = codoncount(seq,'Frame',2,'Reverse',true)
```

```
r2codons = struct with fields:
    AAA: 5
    AAC: 2
    AAG: 5
    AAT: 6
    ACA: 8
    ACC: 4
    ACG: 5
    ACT: 2
    AGA: 5
    AGC: 5
    AGG: 5
    AGT: 7
    ATA: 4
    ATC: 4
    ATG: 10
    ATT: 6
    CAA: 8
    CAC: 5
    CAG: 4
    CAT: 4
    CCA: 5
    CCC: 10
    CCG: 5
    CCT: 5
    CGA: 5
    CGC: 8
    CGG: 8
    CGT: 4
    CTA: 1
    CTC: 5
    CTG: 7
    CTT: 8
    GAA: 1
    GAC: 6
    GAG: 7
    GAT: 4
    GCA: 1
    GCC: 7
    GCG: 6
    GCT: 9
    GGA: 2
    GGC: 2
    GGG: 4
    GGT: 7
    GTA: 4
    GTC: 6
    GTG: 5
    GTT: 5
    TAA: 6
    TAC: 2
    TAG: 4
    TAT: 6
    TCA: 4
    TCC: 6
    TCG: 7
    TCT: 6
    TGA: 6
    TGC: 3
    TGG: 5
    TGT: 6
    TTA: 2
    TTC: 5
    TTG: 3
    TTT: 11
```

Create a heat map of the codons and overlay a grid that groups the synonymous codons according to the standard genetic code.

```
codoncount(seq,'Figure', true);
```

```
AAA - 11     AAC -  5     AAG -  8     AAT -  6
ACA -  6     ACC -  7     ACG -  4     ACT -  7
AGA -  6     AGC -  9     AGG -  5     AGT -  2
ATA -  6     ATC -  4     ATG -  4     ATT -  6
CAA -  3     CAC -  5     CAG -  7     CAT - 10
CCA -  5     CCC -  4     CCG -  8     CCT -  5
CGA -  7     CGC -  6     CGG -  5     CGT -  5
CTA -  4     CTC -  7     CTG -  4     CTT -  5
GAA -  5     GAC -  6     GAG -  5     GAT -  4
GCA -  3     GCC -  2     GCG -  8     GCT -  5
GGA -  6     GGC -  7     GGG - 10     GGT -  4
GTA -  2     GTC -  6     GTG -  5     GTT -  2
TAA -  2     TAC -  4     TAG -  1     TAT -  4
TCA -  6     TCC -  2     TCG -  5     TCT -  5
TGA -  4     TGC -  1     TGG -  5     TGT -  8
TTA -  6     TTC -  1     TTG -  8     TTT -  5
```



Version History

Introduced before R2006a

See Also

`aacount` | `basecount` | `baselookup` | `codonbias` | `dimercount` | `nmercount` | `ntdensity` | `seqcomplement` | `seqrcomplement` | `seqreverse` | `seqwordcount`

colnames (DataMatrix)

Retrieve or set column names of DataMatrix object

Syntax

```
ReturnColNames = colnames(DMObj)
ReturnColNames = colnames(DMObj, ColIndices)
DMObjNew = colnames(DMObj, ColIndices, ColNames)
```

Input Arguments

DMObj	DataMatrix object, such as created by `DataMatrix` (object constructor).
ColIndices	One or more columns in *DMObj*, specified by any of the following: • Positive integer • Vector of positive integers • Character vector specifying a column name • Cell array of character vectors • Logical vector
ColNames	Column names specified by any of the following: • Numeric vector • Cell array of character vectors • Character array • Single character vector, which is used as a prefix for column names, with column numbers appended to the prefix • Logical `true` or `false` (default). If `true`, unique column names are assigned using the format `col1`, `col2`, `col3`, etc. If `false`, no column names are assigned. **Note** The number of elements in *ColNames* must equal the number of elements in *ColIndices*.

Output Arguments

ReturnColNames	Character vector or cell array of character vectors containing column names in *DMObj*.
DMObjNew	DataMatrix object created with names specified by *ColIndices* and *ColNames*.

Description

ReturnColNames = `colnames(`*DMObj*`)` returns *ReturnColNames*, a cell array of character vectors specifying the column names in *DMObj*, a DataMatrix object.

ReturnColNames = colnames(*DMObj*, *ColIndices*) returns the column names specified by *ColIndices*. *ColIndices* can be a positive integer, vector of positive integers, character vector specifying a column name, cell array of character vectors, or a logical vector.

DMObjNew = colnames(*DMObj*, *ColIndices*, *ColNames*) returns *DMObjNew*, a DataMatrix object with columns specified by *ColIndices* set to the names specified by *ColNames*. The number of elements in *ColIndices* must equal the number of elements in *ColNames*.

Version History

Introduced in R2008b

See Also

`DataMatrix` | `rownames`

Topics

DataMatrix object on page 1-734

combine

Class: `bioma.data.ExptData`
Package: `bioma.data`

Combine two ExptData objects

Syntax

`*NewEDObj* = combine(*EDObj1*, *EDObj2*)`

Description

`*NewEDObj* = combine(*EDObj1*, *EDObj2*)` combines data from two ExptData objects and returns a new ExptData object. The number and names of features (rows) in both ExptData objects must match. The number and names of samples (columns) in both ExptData objects must match.

Input Arguments

`EDObj#`

Object of the `bioma.data.ExptData` class.

Default:

See Also

`bioma.data.ExptData`

Topics

"Representing Expression Data Values in ExptData Objects"

combine

Class: `bioma.data.MetaData`
Package: `bioma.data`

Combine two MetaData objects

Syntax

`NewMDObj = combine(MDObj1, MDObj2)`

Description

`NewMDObj = combine(MDObj1, MDObj2)` combines data from two MetaData objects and returns a new MetaData object. The sample or feature names in the two MetaData objects being combined must be unique. The variable names in the two MetaData objects can be unique or the same. If a variable name is common to the two MetaData objects, then the variable occupies one column in the new MetaData object. Variable names unique to either of the two MetaData objects occupy their own column and contain values only for the samples or features where the variable is present.

Input Arguments

`MDObj#`

Object of the `bioma.data.MetaData` class.

Default:

See Also

`bioma.data.MetaData`

Topics

"Representing Sample and Feature Metadata in MetaData Objects"

combine

Class: `bioma.data.MIAME`
Package: `bioma.data`

Combine two MIAME objects

Syntax

`NewMIAMEObj = combine(MIAMEObj1, MIAMEObj2)`

Description

`NewMIAMEObj` `= combine(`*`MIAMEObj1`*`,` *`MIAMEObj2`*`)` combines data from two MIAME objects and returns a new MIAME object. The `combine` method concatenates the properties of the two objects together.

Input Arguments

`MIAMEObj#`

Object of the `bioma.data.MIAME` class.

Default:

Examples

Construct two MIAME objects, and then combine them:

```
% Create a MATLAB structure containing GEO Series data
geoStruct1 = getgeodata('GSE4616');
% Create a second MATLAB structure containing GEO Series data
geoStruct2 = getgeodata('GSE11287');
% Import bioma.data package to make constructor function
% available
import bioma.data.*
% Construct MIAME object from the first structure
MIAMEObj1 = MIAME(geoStruct1);
% Construct MIAME object from the second structure
MIAMEObj2 = MIAME(geoStruct2);
% Combine the two MIAME objects
newMIAMEObj = combine(MIAMEObj1, MIAMEObj2)
```

See Also

`bioma.data.MIAME`

Topics

"Representing Experiment Information in a MIAME Object"

combine

Combine two objects

Syntax

```
combinedData = combine(data1,data2)
newObj = combine(data1,data2,'Name',objName)
```

Description

`combinedData = combine(data1,data2)` combines sequence data from two objects of the same class (`BioRead` or `BioMap`) and returns the data `combinedData` in a new object. The `combine` function concatenates the properties of the two objects.

`newObj = combine(data1,data2,'Name',objName)` specifies the name `objName` for `newObj`.

Examples

Combine NGS Data

Load the first data set.

```
br1 = BioRead('ex1.sam')
```

```
br1 =
  BioRead with properties:

     Quality: [1501x1 File indexed property]
    Sequence: [1501x1 File indexed property]
      Header: [1501x1 File indexed property]
       NSeqs: 1501
        Name: ''
```

Load the second data set.

```
br2 = BioRead('ex2.sam')
```

```
br2 =
  BioRead with properties:

     Quality: [3307x1 File indexed property]
    Sequence: [3307x1 File indexed property]
      Header: [3307x1 File indexed property]
       NSeqs: 3307
        Name: ''
```

Combine the two data sets. Set the name of the new object to `'combinedData'`.

```
br3 = combine(br1,br2,'Name','combinedData')
```

```
br3 = 
  BioRead with properties:

     Quality: {4808x1 cell}
    Sequence: {4808x1 cell}
      Header: {4808x1 cell}
       NSeqs: 4808
        Name: 'combinedData'
```

Input Arguments

`data1` — First sequence read data
`BioRead` | `BioMap`

First sequence read data to be combined with the second sequence read data, specified as a `BioRead` or `BioMap` object.

`data2` — Second sequence read data
`BioRead` | `BioMap`

Second sequence read data to be combined with the first sequence read data, specified as a `BioRead` or `BioMap` object.

`objName` — Name of combined object
`''` (default) | character vector | string

Name of the combined object, specified as a character vector or string.

Version History
Introduced in R2010a

See Also
`BioMap` | `BioRead`

Topics
“Manage Sequence Read Data in Objects”

conncomp (biograph)

(Removed) Find strongly or weakly connected components in biograph object

Note The function has been removed. Use the `conncomp` function of `graph` or `digraph` instead.

Syntax

```
[S, C] = conncomp(BGObj)

[S, C] = conncomp(BGObj, ...'Directed', DirectedValue, ...)
[S, C] = conncomp(BGObj, ...'Weak', WeakValue, ...)
```

Arguments

`BGObj`	Biograph object created by `biograph` (object constructor).
`DirectedValue`	Property that indicates whether the graph is directed or undirected. Enter `false` for an undirected graph. This results in the upper triangle of the adjacency matrix being ignored. Default is `true`. A DFS-based algorithm computes the connected components. Time complexity is `O(N+E)`, where `N` and `E` are number of nodes and edges respectively.
`WeakValue`	Property that indicates whether to find weakly connected components or strongly connected components. A weakly connected component is a maximal group of nodes that are mutually reachable by violating the edge directions. Set *`WeakValue`* to `true` to find weakly connected components. Default is `false`, which finds strongly connected components. The state of this parameter has no effect on undirected graphs because weakly and strongly connected components are the same in undirected graphs. Time complexity is `O(N+E)`, where `N` and `E` are number of nodes and edges respectively.

Description

Tip For introductory information on graph theory functions, see “Graph Theory Functions”.

`[S, C] = conncomp(BGObj)` finds the strongly connected components of an N-by-N adjacency matrix extracted from a biograph object, *`BGObj`* using Tarjan's algorithm. A strongly connected component is a maximal group of nodes that are mutually reachable without violating the edge directions. The N-by-N adjacency matrix represents a directed graph; all nonzero entries in the matrix indicate the presence of an edge.

The number of components found is returned in *`S`*, and *`C`* is a vector indicating to which component each node belongs.

Tarjan's algorithm has a time complexity of `O(N+E)`, where `N` and `E` are the number of nodes and edges respectively.

`[S, C] = conncomp(BGObj, ...'PropertyName', PropertyValue, ...)` calls `conncomp` with optional properties that use property name/property value pairs. You can specify one or more properties in any order. Each *`PropertyName`* must be enclosed in single quotes and is case insensitive. These property name/property value pairs are as follows:

`[S, C] = conncomp(BGObj, ...'Directed', DirectedValue, ...)` indicates whether the graph is directed or undirected. Set *`DirectedValue`* to `false` for an undirected graph. This results in the upper triangle of the adjacency matrix being ignored. Default is `true`. A DFS-based algorithm computes the connected components. Time complexity is `O(N+E)`, where `N` and `E` are number of nodes and edges respectively.

`[S, C] = conncomp(BGObj, ...'Weak', WeakValue, ...)` indicates whether to find weakly connected components or strongly connected components. A weakly connected component is a maximal group of nodes that are mutually reachable by violating the edge directions. Set *`WeakValue`* to `true` to find weakly connected components. Default is `false`, which finds strongly connected components. The state of this parameter has no effect on undirected graphs because weakly and strongly connected components are the same in undirected graphs. Time complexity is `O(N+E)`, where `N` and `E` are number of nodes and edges respectively.

Note By definition, a single node can be a strongly connected component.

Note A directed acyclic graph (DAG) cannot have any strongly connected components larger than one.

Version History

Introduced in R2006b

R2022b: Removed

Errors starting in R2022b

The function has been removed. Use the `conncomp` function of `graph` or `digraph` instead.

R2022a: Warns

Warns starting in R2022a

The function issues a warning that it will be removed in a future release.

R2021b: To be removed

Not recommended starting in R2021b

The function runs without warning, but it will be removed in a future release.

References

[1] Tarjan, R.E., (1972). Depth first search and linear graph algorithms. SIAM Journal on Computing *1(2)*, 146–160.

[2] Sedgewick, R., (2002). Algorithms in C++, Part 5 Graph Algorithms (Addison-Wesley).

[3] Siek, J.G., Lee, L-Q, and Lumsdaine, A. (2002). The Boost Graph Library User Guide and Reference Manual, (Upper Saddle River, NJ:Pearson Education).

See Also

`conncomp`

cpgisland

Locate CpG islands in DNA sequence

Syntax

```
cpgStruct = cpgisland(SeqDNA)

cpgStruct = cpgisland(SeqDNA, ...'Window', WindowValue, ...)
cpgStruct = cpgisland(SeqDNA, ...'MinIsland', MinIslandValue, ...)
cpgStruct = cpgisland(SeqDNA, ...'GCmin', GCminValue, ...)
cpgStruct = cpgisland(SeqDNA, ...'CpGoe', CpGoeValue, ...)
cpgStruct = cpgisland(SeqDNA, ...'Plot', PlotValue, ...)
```

Input Arguments

SeqDNA	One of the following: • Character vector or string specifying a nucleotide sequence • Row vector of integers specifying a nucleotide sequence • MATLAB structure containing a `Sequence` field that contains a DNA nucleotide sequence, such as returned by `fastaread`, `fastqread`, `emblread`, `getembl`, `genbankread`, or `getgenbank` Valid characters include `A`, `C`, `G`, and `T`. `cpgisland` does not count ambiguous nucleotides or gaps.
WindowValue	Integer specifying the window size for calculating GC content and CpGobserved/CpGexpected ratios. Default is `100` bases. A smaller window size increases the noise in a plot.
MinIslandValue	Integer specifying the minimum number of consecutive marked bases to report as a CpG island. Default is `200` bases.
GCminValue	Value specifying the minimum GC percent in a window needed to mark a base. Choices are a value between `0` and `1`. Default is `0.5`.
CpGoeValue	Value specifying the minimum CpGobserved/CpGexpected ratio in each window needed to mark a base. Choices are a value between `0` and `1`. Default is `0.6`. This ratio is defined as: `CPGobs/CpGexp = (NumCpGs*Length)/(NumGs*NumCs)`
PlotValue	Controls the plotting of GC content, CpGoe content, CpG islands greater than the minimum island size, and all potential CpG islands for the specified criteria. Choices are `true` or `false` (default).

Output Arguments

cpgStruct	MATLAB structure containing the starting and ending bases of the CpG islands greater than the minimum island size.

Description

cpgStruct = cpgisland(*SeqDNA*) searches *SeqDNA*, a DNA nucleotide sequence, for CpG islands with a GC content greater than 50% and a CpGobserved/CpGexpected ratio greater than 60%. It marks bases meeting this criteria within a moving window of 100 DNA bases and then returns the results in *cpgStruct*, a MATLAB structure containing the starting and ending bases of the CpG islands greater than the minimum island size of 200 bases.

cpgStruct = cpgisland(*SeqDNA*, ...'*PropertyName*', *PropertyValue*, ...) calls cpgisland with optional properties that use property name/property value pairs. You can specify one or more properties in any order. Each *PropertyName* must be enclosed in single quotation marks and is case insensitive. These property name/property value pairs are as follows:

cpgStruct = cpgisland(*SeqDNA*, ...'Window', *WindowValue*, ...) specifies the window size for calculating GC content and CpGobserved/CpGexpected ratios. Default is 100 bases. A smaller window size increases the noise in a plot.

cpgStruct = cpgisland(*SeqDNA*, ...'MinIsland', *MinIslandValue*, ...) specifies the minimum number of consecutive marked bases to report as a CpG island. Default is 200 bases.

cpgStruct = cpgisland(*SeqDNA*, ...'GCmin', *GCminValue*, ...) specifies the minimum GC percent in a window needed to mark a base. Choices are a value between 0 and 1. Default is 0.5.

cpgStruct = cpgisland(*SeqDNA*, ...'CpGoe', *CpGoeValue*, ...) specifies the minimum CpGobserved/CpGexpected ratio in each window needed to mark a base. Choices are a value between 0 and 1. Default is 0.6. This ratio is defined as:

```
CPGobs/CpGexp = (NumCpGs*Length)/(NumGs*NumCs)
```

cpgStruct = cpgisland(*SeqDNA*, ...'Plot', *PlotValue*, ...) controls the plotting of GC content, CpGoe content, CpG islands greater than the minimum island size, and all potential CpG islands for the specified criteria. Choices are true or false (default).

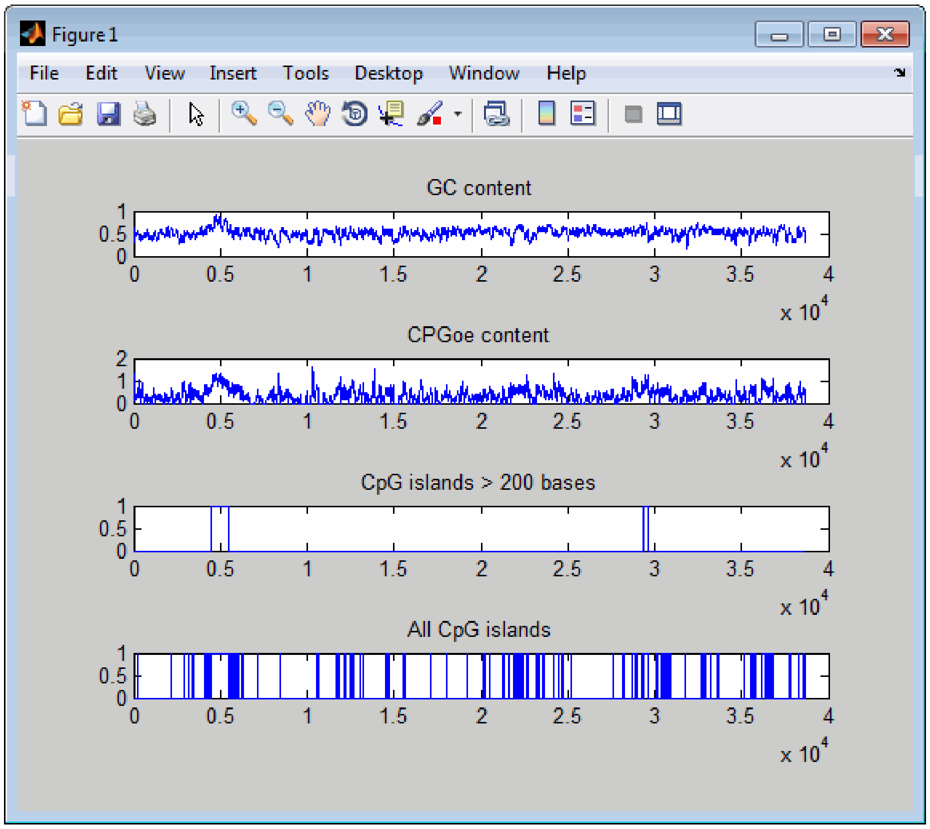
Examples

1. Import a nucleotide sequence from the GenBank database. For example, retrieve a sequence from *Homo sapiens* chromosome 12.

   ```
   S = getgenbank('AC156455');
   ```
2. Calculate the CpG islands in the sequence and plot the results.

   ```
   cpgisland(S.Sequence,'PLOT',true)

   ans =

       Starts: [4510 29359]
        Stops: [5468 29604]
   ```

   The CpG islands greater than 200 bases in length are listed and a plot displays.



Version History

Introduced before R2006a

See Also

`basecount` | `ntdensity` | `seqshoworfs`

crossvalind

Generate indices for training and test sets

Syntax

```
cvIndices = crossvalind(cvMethod,N,M)
[train,test] = crossvalind(cvMethod,N,M)
___ = crossvalind(___,Name,Value)
```

Description

`cvIndices = crossvalind(cvMethod,N,M)` returns the indices `cvIndices` after applying `cvMethod` on `N` observations using `M` as the selection parameter.

`[train,test] = crossvalind(cvMethod,N,M)` returns the logical vectors `train` and `test`, representing observations that belong to the training set and the test (evaluation) set, respectively. You can specify any supported method except `'Kfold'`, which accepts a scalar output only.

`___ = crossvalind(___,Name,Value)` specifies additional options using one or more name-value pair arguments in addition to the arguments in previous syntaxes. For example, `cvIndices = crossvalind('HoldOut',Groups,0.2,'Class',{'Cancer','Control'})` specifies to use observations from the 'Cancer' and 'Control' groups to generate indices that represent 20% of observations as the holdout set and 80% as the training set.

Examples

Perform 10-Fold Cross-Validation

Create indices for the 10-fold cross-validation and classify measurement data for the Fisher iris data set. The Fisher iris data set contains width and length measurements of petals and sepals from three species of irises.

Load the data set.

```
load fisheriris
```

Create indices for the 10-fold cross-validation.

```
indices = crossvalind('Kfold',species,10);
```

Initialize an object to measure the performance of the classifier.

```
cp = classperf(species);
```

Perform the classification using the measurement data and report the error rate, which is the ratio of the number of incorrectly classified samples divided by the total number of classified samples.

```
for i = 1:10
    test = (indices == i);
    train = ~test;
    class = classify(meas(test,:),meas(train,:),species(train,:));
    classperf(cp,class,test);
end
cp.ErrorRate
```

```
ans = 0.0200
```

Suppose you want to use the observation data from the `setosa` and `virginica` species only and exclude the `versicolor` species from cross-validation.

```
labels = {'setosa','virginica'};
indices = crossvalind('Kfold',species,10,'Classes',labels);
```

`indices` now contains zeros for the rows that belong to the `versicolor` species.

Perform the classification again.

```
for i = 1:10
    test = (indices == i);
    train = ~test;
    class = classify(meas(test,:),meas(train,:),species(train,:));
    classperf(cp,class,test);
end
cp.ErrorRate
```

```
ans = 0.0160
```

Perform Leave-One-Out Cross-Validation

Load the carbig data set.

```
load carbig;
x = Displacement;
y = Acceleration;
N = length(x);
```

Train a second degree polynomial model with the leave-one-out cross-validation, and evaluate the averaged cross-validation error. The function randomly selects one observation to hold out for the evaluation set, and using this method within a loop does not guarantee disjointed evaluation sets, and you may see a different CVerr for each run.

```
sse = 0; % Initialize the sum of squared error.
for i = 1:100
    [train,test] = crossvalind('LeaveMOut',N,1);
    yhat = polyval(polyfit(x(train),y(train),2),x(test));
    sse = sse + sum((yhat - y(test)).^2);
end
CVerr = sse / 100;
```

Input Arguments

`cvMethod` — Cross-validation method
character vector | string

Cross-validation method, specified as a character vector or string.

This table describes the valid cross-validation methods. Depending on the method, the third input argument (`M`) has different meanings and requirements.

cvMethod	M	Description
`'Kfold'`	`M` is the fold parameter, most commonly known as *K* in the *K*-fold cross-validation. `M` must be a positive integer. The default value is `5`.	The method uses *K*-fold cross-validation to generate indices. This method uses `M-1` folds for training and the last fold for evaluation. The method repeats this process `M` times, leaving one different fold for evaluation each time.
`'HoldOut'`	`M` is the proportion of observations to hold out for the test set. `M` must be a scalar between `0` and `1`. The default value is `0.5`, corresponding to a 50% holdout.	The method randomly selects approximately `N*M` observations to hold out for the test (evaluation) set. Using this method within a loop is similar to using *K*-fold cross-validation one time outside the loop, except that nondisjointed subsets are assigned to each evaluation.
`'LeaveMOut`	`M` is the number of observations to leave out for the test set. `M` must be a positive integer. The default value is `1`, corresponding to the leave-one-out cross-validation (LOOCV).	The method randomly selects `M` observations to hold out for the evaluation set. Using this cross-validation method within a loop does not guarantee disjointed evaluation sets. To guarantee disjointed evaluation sets, use `'Kfold'` instead.
`'Resubstitution'`	`M` must be specified as a two-element vector `[`*P*`,`*Q*`]`. Each element must be a scalar between `0` and `1`. The default value is `[1,1]`, corresponding to the full resubstitution.	The method randomly selects `N*`*P* observations for the evaluation set and `N*`*Q* observations for the training set. The method selects the sets while minimizing the number of observations used in both sets. *Q* `= 1-`*P* corresponds to the holdout `(100*`*P*`)%`.

Example: `'Kfold'`

Data Types: `char` | `string`

`N` — Total number of observations or grouping information
positive integer | vector of positive integers | logical vector | cell array of character vectors

Total number of observations or grouping information, specified as a positive integer, vector of positive integers, logical vector, or cell array of character vectors.

N can be a positive integer specifying the total number of samples in your data set, for instance.

N can also be a vector of positive integers or logical values, or a cell array of character vectors, containing grouping information or labels for your samples. The partition of the groups depends on the type of cross-validation. For 'Kfold', each group is divided into M subsets, approximately equal in size. For all other methods, approximately equal numbers of observations from each group are selected for the evaluation (test) set. The training set contains at least one observation from each group regardless of the cross-validation method you use.

Example: 100

Data Types: double | cell

M — Cross-validation parameter
positive scalar | positive integer | two-element vector

Cross-validation parameter, specified as a positive scalar between 0 and 1, positive integer, or two-element vector. Depending on the cross-validation method, the requirements for M differ. For details, see cvMethod.

Example: 5

Data Types: double

Name-Value Pair Arguments

Specify optional pairs of arguments as Name1=Value1,...,NameN=ValueN, where Name is the argument name and Value is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose Name *in quotes.*

Example: [train,test] = crossvalind('LeaveMOut',groups,1,'Min',3) specifies to have at least three observations in each group in the training set when performing the leave-one-out cross-validation.

Classes — Class or group information
vector of positive integers | character vector | string | string vector | cell array of character vectors

Class or group information, specified as the comma-separated pair consisting of 'Classes' and a vector of positive integers, character vector, string, string vector, or cell array of character vectors. This option lets you restrict the observations to only the specified groups.

This name-value pair argument is applicable only when you specify N as a grouping variable. The data type of 'Classes' must match that of N. For example, if you specify N as a cell array of character vectors containing class labels, you must use a cell array of character vectors to specify 'Classes'. The output arguments you specify contain the value 0 for observations belonging to excluded classes.

Example: 'Classes',{'Cancer','Control'}

Data Types: double | cell

Min — Minimum number of observations
1 (default) | positive integer

Minimum number of observations for each group in the training set, specified as the comma-separated pair consisting of 'Min' and a positive integer. Setting a large value can help to balance the training groups, but causes partial resubstitution when there are not enough observations.

This name-value pair argument is not applicable for the `'Kfold'` method.

Example: `'Min',3`

Data Types: `double`

Output Arguments

`cvIndices` — Cross-validation indices
vector

Cross-validation indices, returned as a vector.

If you are using `'Kfold'` as the cross-validation method, `cvIndices` contains equal (or approximately equal) proportions of the integers 1 through `M`, which define a partition of the `N` observations into `M` disjointed subsets.

For other cross-validation methods, `cvIndices` is a logical vector containing 1s for observations that belong to the training set and 0s for observations that belong to the test (evaluation) set.

`train` — Training set
logical vector

Training set, returned as a logical vector. This argument specifies which observations belong to the training set.

`test` — Test set
logical vector

Test set, returned as a logical vector. This argument specifies which observations belong to the test set.

Version History

Introduced before R2006a

See Also

`classperf` | `classify` | `grp2idx`

cuffcompare

Compare assembled transcripts across multiple experiments

Syntax

```
statsFile = cuffcompare(gtfFiles)
statsFile = cuffcompare(gtfFiles,compareOptions)
statsFile = cuffcompare(gtfFiles,Name,Value)
[statsFile,combinedGTF,lociFile,trackingFile] = cuffcompare( ___ )
```

Description

`statsFile = cuffcompare(gtfFiles)` compares the assembled transcripts in `gtfFiles` and returns summary statistics in the output file `statsFile` [1].

`cuffcompare` requires the Cufflinks Support Package for the Bioinformatics Toolbox. If the support package is not installed, then the function provides a download link. For details, see “Bioinformatics Toolbox Software Support Packages”.

`statsFile = cuffcompare(gtfFiles,compareOptions)` uses additional options specified by `compareOptions`.

`statsFile = cuffcompare(gtfFiles,Name,Value)` uses additional options specified by one or more name-value pair arguments. For example, `statsFile = cuffcompare(gtfFile,'OutputPrefix',"cuffComp")` appends the prefix `"cuffComp"` to the output file names.

`[statsFile,combinedGTF,lociFile,trackingFile] = cuffcompare( ___ )` returns the names of the output files using any of the input argument combinations in the previous syntaxes. By default, the function saves all files to the current directory.

Examples

Assemble Transcriptome and Perform Differential Expression Testing

Create a `CufflinksOptions` object to define cufflinks options, such as the number of parallel threads and the output directory to store the results.

```
cflOpt = CufflinksOptions;
cflOpt.NumThreads = 8;
cflOpt.OutputDirectory = "./cufflinksOut";
```

The SAM files provided for this example contain aligned reads for *Mycoplasma pneumoniae* from two samples with three replicates each. The reads are simulated 100bp-reads for two genes (`gyrA` and `gyrB`) located next to each other on the genome. All the reads are sorted by reference position, as required by `cufflinks`.

```
sams = ["Myco_1_1.sam","Myco_1_2.sam","Myco_1_3.sam",...
        "Myco_2_1.sam", "Myco_2_2.sam", "Myco_2_3.sam"];
```

Assemble the transcriptome from the aligned reads.

```
[gtfs,isofpkm,genes,skipped] = cufflinks(sams,cflOpt);
```

`gtfs` is a list of GTF files that contain assembled isoforms.

Compare the assembled isoforms using `cuffcompare`.

```
stats = cuffcompare(gtfs);
```

Merge the assembled transcripts using `cuffmerge`.

```
mergedGTF = cuffmerge(gtfs,'OutputDirectory','./cuffMergeOutput');
```

`mergedGTF` reports only one transcript. This is because the two genes of interest are located next to each other, and `cuffmerge` cannot distinguish two distinct genes. To guide `cuffmerge`, use a reference GTF (`gyrAB.gtf`) containing information about these two genes. If the file is not located in the same directory that you run `cuffmerge` from, you must also specify the file path.

```
gyrAB = which('gyrAB.gtf');
mergedGTF2 = cuffmerge(gtfs,'OutputDirectory','./cuffMergeOutput2',...
            'ReferenceGTF',gyrAB);
```

Calculate abundances (expression levels) from aligned reads for each sample.

```
abundances1 = cuffquant(mergedGTF2,["Myco_1_1.sam","Myco_1_2.sam","Myco_1_3.sam"],...
                        'OutputDirectory','./cuffquantOutput1');
abundances2 = cuffquant(mergedGTF2,["Myco_2_1.sam", "Myco_2_2.sam", "Myco_2_3.sam"],...
                        'OutputDirectory','./cuffquantOutput2');
```

Assess the significance of changes in expression for genes and transcripts between conditions by performing the differential testing using `cuffdiff`. The `cuffdiff` function operates in two distinct steps: the function first estimates abundances from aligned reads, and then performs the statistical analysis. In some cases (for example, distributing computing load across multiple workers), performing the two steps separately is desirable. After performing the first step with `cuffquant`, you can then use the binary CXB output file as an input to `cuffdiff` to perform statistical analysis. Because `cuffdiff` returns several files, specify the output directory is recommended.

```
isoformDiff = cuffdiff(mergedGTF2,[abundances1,abundances2],...
                      'OutputDirectory','./cuffdiffOutput');
```

Display a table containing the differential expression test results for the two genes `gyrB` and `gyrA`.

```
readtable(isoformDiff,'FileType','text')
```

```
ans =

  2×14 table

        test_id             gene_id         gene              locus              sample_1     sample_
    ________________    _____________    ______    _______________________    ________     _______

    'TCONS_00000001'    'XLOC_000001'    'gyrB'    'NC_000912.1:2868-7340'      'q1'          'q2'
    'TCONS_00000002'    'XLOC_000001'    'gyrA'    'NC_000912.1:2868-7340'      'q1'          'q2'
```

You can use `cuffnorm` to generate normalized expression tables for further analyses. `cuffnorm` results are useful when you have many samples and you want to cluster them or plot expression

levels for genes that are important in your study. Note that you cannot perform differential expression analysis using `cuffnorm`.

Specify a cell array, where each element is a string vector containing file names for a single sample with replicates.

```
alignmentFiles = {["Myco_1_1.sam","Myco_1_2.sam","Myco_1_3.sam"],...
                  ["Myco_2_1.sam", "Myco_2_2.sam", "Myco_2_3.sam"]}
isoformNorm = cuffnorm(mergedGTF2, alignmentFiles,...
                      'OutputDirectory', './cuffnormOutput');
```

Display a table containing the normalized expression levels for each transcript.

```
readtable(isoformNorm,'FileType','text')
```

```
ans =

  2×7 table

      tracking_id         q1_0          q1_2          q1_1          q2_1          q2_0
    ________________    __________    __________    __________    __________    __________    ___

    'TCONS_00000001'    1.0913e+05         78628    1.2132e+05    4.3639e+05    4.2228e+05    4.2
    'TCONS_00000002'    3.5158e+05    3.7458e+05    3.4238e+05    1.0483e+05    1.1546e+05    1.1
```

Column names starting with *q* have the format: *conditionX_N*, indicating that the column contains values for replicate *N* of *conditionX*.

Input Arguments

`gtfFiles` — Names of GTF files

string array | cell array of character vectors

Names of GTF files, specified as a string vector or cell array of character vectors. Each GTF file corresponds to a sample produced by `cufflinks`.

Example: `["Myco_1_1.transcripts.gtf","Myco_2_1.transcripts.gtf"]`

Data Types: `string` | `cell`

`compareOptions` — `cuffcompare` options

`CuffCompareOptions` object | character vector | string

`cuffcompare` options, specified as a `CuffCompareOptions` object, character vector, or string. The character vector or string must be in the original `cuffcompare` option syntax (prefixed by one or two dashes), such as `'-d 100 -e 80'` [1].

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `statsFile = cuffcompare(gtfFile,'OutputPrefix',"cuffComp",'MaxGroupingRange',90)`

`ConsensusPrefix` — Prefix for consensus transcript names
`"TCONS"` (default) | string | character vector

Prefix for consensus transcript names in the output `combined.gtf` file, specified as a string or character vector. This option must be a string or character vector with a non-zero length.

Example: `'ConsensusPrefix',"consensusTs"`

Data Types: `char` | `string`

`DiscardIntronRedundant` — Flag to ignore intron-redundant transfrags
`false` (default) | `true`

Flag to ignore intron-redundant transfrags if they have the same 5' end but different 3' ends, specified as `true` or `false`.

Example: `'DiscardIntronRedundant',true`

Data Types: `logical`

`DiscardSingleExonAll` — Flag to discard single-exon transfrags and reference transcripts
`false` (default) | `true`

Flag to discard single-exon transfrags and reference transcripts, specified as `true` or `false`.

Example: `'DiscardSingleExonAll',true`

Data Types: `logical`

`DiscardSingleExonReference` — Flag to discard single-exon reference transcripts
`false` (default) | `true`

Flag to discard single-exon reference transcripts, specified as `true` or `false`.

Example: `'DiscardSingleExonReference',true`

Data Types: `logical`

`ExtraCommand` — Additional commands
`""` (default) | character vector | string

The commands must be in the native syntax (prefixed by one or two dashes). Use this option to apply undocumented flags and flags without corresponding MATLAB properties.

Example: `'ExtraCommand',"--library-type fr-secondstrand"`

Data Types: `char` | `string`

`GTFManifest` — Name of text file containing list of GTF files to process
string | character vector

Name of the text file containing a list of GTF files to process, specified as a string or character vector. The file must contain one GTF file path per line. You can use this option as an alternative to passing an array of file names to `cuffcompare`.

Example: `'GTFManifest',"gtfManifestFile.txt"`

Data Types: `char` | `string`

`GenericGFF` — Flag to treat input GTF files as GFF
`false` (default) | `true`

Flag to treat input GTF files as GFF files, specified as `true` or `false`. Use this option when the input GFF or GTF files are not produced by `cufflinks`.

Example: `'GenericGFF',true`

Data Types: `logical`

`IncludeAll` — Flag to include all available options
`false` (default) | `true`

The original (native) syntax is prefixed by one or two dashes. By default, the function converts only the specified options. If the value is `true`, the software converts all available options, with default values for unspecified options, to the original syntax.

Note If you set `IncludeAll` to `true`, the software translates all available properties, with default values for unspecified properties. The only exception is that when the default value of a property is `NaN`, `Inf`, `[]`, `''`, or `""`, then the software does not translate the corresponding property.

Example: `'IncludeAll',true`

Data Types: `logical`

`IncludeContained` — Flag to include transfrags contained by other transfrags
`false` (default) | `true`

Flag to include transfrags contained by other transfrags in the same locus in the output `combined.gtf`, specified as `true` or `false`. By default, `cuffcompare` does not include these contained transfrags. If the value is `true`, the contained transfrags include a `contained_in` attribute indicating the first container transfrag found.

Example: `'IncludeContained',true`

Data Types: `logical`

`MaxAccuracyRange` — Number of bases from terminal exons to use when assessing exon accuracy
`100` (default) | positive integer

Number of bases from the free ends of terminal exons to use when assessing exon accuracy, specified as a positive integer.

Example: `'MaxAccuracyRange',80`

Data Types: `double`

`MaxGroupingRange` — Number of bases to use for grouping transcript start sites
`100` (default) | positive integer

Number of bases to use for grouping transcript start sites, specified as a positive integer.

Example: `'MaxGroupingRange',90`

Data Types: `double`

`OutputPrefix` — Prefix for `cuffcompare` output files
`"cuffcmp"` (default) | string | character vector

Prefix for `cuffcompare` output files, specified as a string or character vector. This option must be a string or character vector with a non-zero length.

Example: `'OutputPrefix',"cuffcompareOut"`

Data Types: `char` | `string`

`ReferenceGTF` — Name of GTF or GFF file containing reference transcripts
string | character vector

Name of the GTF or GFF file containing reference transcripts to compare to each sample, specified as a string or character vector. If you provide a file, the function compares each sample to the references in the file and marks isoforms as `overlapping`, `matching`, or `novel`. The function stores these tags in the output files `.refmap` and `.tmap` files.

Example: `'ReferenceGTF',"references.gtf"`

Data Types: `char` | `string`

`SequenceDirectory` — Name of directory containing FASTA sequences to classify input transcripts as repeats
string | character vector

Name of directory containing FASTA sequences to classify input transcripts as repeats, specified as a string or character vector. The directory must contain FASTA-format files with the underlying genomic sequences and contain one FASTA file per reference. Name each FASTA file after the chromosome with the extension `.fa` or `.fasta`.

Example: `'SequenceDirectory',"./SequenceDirectory/"`

Data Types: `char` | `string`

`SnCorrection` — Flag to consider only reference transcripts that overlap with input transfrags
`false` (default) | `true`

Flag to consider only reference transcripts that overlap with any of the input transfrags, specified as `true` or `false`. If the value is `true`:

- The function ignores any reference transcripts that do not overlap with any of the input transfrags.
- You must also specify the `ReferenceGTF` option.

Example: `'SnCorrection',true`

Data Types: `logical`

`SpCorrection` — Flag to consider only input transcripts that overlap with reference transcripts
`false` (default) | `true`

Flag to consider only input transcripts that overlap with any of the reference transcripts, specified as `true` or `false`. If the value is `true`:

- The function ignores any input transcripts that do not overlap with any of the reference transcripts and reports no novel loci.
- You must also specify the `ReferenceGTF` option.

Example: `'SpCorrection',true`

Data Types: `logical`

`SuppressMapFiles` — Flag to prevent creation of `.tmap` and `.refmap` files
`false` (default) | `true`

Flag to prevent the creation of `.tmap` and `.refmap` files, specified as `true` or `false`. Set the value to `true` to prevent the function from generating the files.

Example: `'SuppressMapFiles',true`

Data Types: `logical`

Output Arguments

`statsFile` — Name of text file containing statistics
`"cuffcmp.stats"`

Name of the text file containing statistics related to the accuracy of the transcripts in each sample, returned as string. The function performs the tests for sensitivity (Sn) and specificity (Sp) at various levels, including the nucleotide, exon, and intron levels, and reports the results in this file.

The default file name is `"cuffcmp.stats"`. If you specify `OutputPrefix`, the function uses it instead of `"cuffcmp"`.

`combinedGTF` — Name of file containing union of all transfrags in each sample
`"cuffcmp.combined.gtf"`

Name of the file containing the union of all transfrags in each sample, returned as a string.

The default file name is `"cuffcmp.combined.gtf"`. If you specify `OutputPrefix`, the function uses it instead of `"cuffcmp"`.

`lociFile` — Name of file with all processed loci
`"cuffcmp.loci"`

Name of file with all processed loci across all transcripts, returned as a string.

The default file name is `"cuffcmp.loci"`. If you specify `OutputPrefix`, the function uses it instead of `"cuffcmp"`.

`trackingFile` — Name of file containing transcripts with identical coordinates
`"cuffcmp.tracking"`

Name of the file containing transcripts with identical coordinates, introns, and strands, returned as a string.

The default file name is `"cuffcmp.tracking"`. If you specify `OutputPrefix`, the function uses it instead of `"cuffcmp"`.

Version History
Introduced in R2019a

References

[1] Trapnell, Cole, Brian A Williams, Geo Pertea, Ali Mortazavi, Gordon Kwan, Marijke J van Baren, Steven L Salzberg, Barbara J Wold, and Lior Pachter. “Transcript Assembly and Quantification by RNA-Seq Reveals Unannotated Transcripts and Isoform Switching during Cell Differentiation.” *Nature Biotechnology* 28, no. 5 (May 2010): 511–15.

See Also

`CuffCompareOptions` | `cufflinks`

Topics

“Bioinformatics Toolbox Software Support Packages”

External Websites

Cufflinks manual

CuffCompareOptions

Option set for `cuffcompare`

Description

A `CuffCompareOptions` object specifies options for the `cuffcompare` function, which compares assembled transcripts across several experiments [1].

Creation

Syntax

```
cuffcompareOpt = CuffCompareOptions
cuffcompareOpt = CuffCompareOptions(Name,Value)
cuffcompareOpt = CuffCompareOptions(S)
```

Description

`cuffcompareOpt = CuffCompareOptions` creates a `CuffCompareOptions` object with the default property values.

`CuffCompareOptions` requires the Cufflinks Support Package for the Bioinformatics Toolbox. If the support package is not installed, then the function provides a download link. For details, see "Bioinformatics Toolbox Software Support Packages".

`cuffcompareOpt = CuffCompareOptions(Name,Value)` sets the object properties on page 1-615 using one or more name-value pair arguments. Enclose each property name in quotes. For example, `cuffcompareOpt = CuffCompareOptions('SupressMapFiles',true)` prevents the creation of `.tmap` and `.refmap` files.

`cuffcompareOpt = CuffCompareOptions(S)` specifies optional parameters using the string or character vector `S`.

Input Arguments

S — `cuffcompare` options
string | character vector

`cuffcompare` options, specified as a string or character vector. `S` must be in the original `cuffcompare` option syntax (prefixed by one or two dashes).

Example: `'-d 100 -e 80'`

Properties

`ConsensusPrefix` — Prefix for consensus transcript names

`"TCONS"` (default) | string | character vector

Prefix for consensus transcript names in the output `combined.gtf` file, specified as a string or character vector. This option must be a string or character vector with a non-zero length.

Example: `"consensusTs"`

Data Types: `char` | `string`

`DiscardIntronRedundant` — Flag to ignore intron-redundant transfrags

`false` (default) | `true`

Flag to ignore intron-redundant transfrags if they have the same 5' end but different 3' ends, specified as `true` or `false`.

Example: `true`

Data Types: `logical`

`DiscardSingleExonAll` — Flag to discard single-exon transfrags and reference transcripts

`false` (default) | `true`

Flag to discard single-exon transfrags and reference transcripts, specified as `true` or `false`.

Example: `true`

Data Types: `logical`

`DiscardSingleExonReference` — Flag to discard single-exon reference transcripts

`false` (default) | `true`

Flag to discard single-exon reference transcripts, specified as `true` or `false`.

Example: `true`

Data Types: `logical`

`ExtraCommand` — Additional commands

`""` (default) | string | character vector

The commands must be in the native syntax (prefixed by one or two dashes). Use this option to apply undocumented flags and flags without corresponding MATLAB properties.

When the software converts the original flags to MATLAB properties, it stores any unrecognized flags in this property.

Example: `"--library-type fr-secondstrand"`

Data Types: `char` | `string`

`GTFManifest` — Name of text file containing list of GTF files to process

string | character vector

Name of the text file containing a list of GTF files to process, specified as a string or character vector. The file must contain one GTF file path per line. You can use this option as an alternative to passing an array of file names to `cuffcompare`.

Example: "gtfManifestFile.txt"

Data Types: char | string

GenericGFF — Flag to treat input GTF files as GFF

false (default) | true

Flag to treat input GTF files as GFF files, specified as true or false. Use this option when the input GFF or GTF files are not produced by cufflinks.

Example: true

Data Types: logical

IncludeAll — Flag to include all object properties when converting to original syntax

false (default) | true

Flag to include all the object properties with the corresponding default values when converting to the original options syntax, specified as true or false. You can convert the properties to the original syntax prefixed by one or two dashes (such as '-d 100 -e 80') by using getCommand. The default value false means that when you call getCommand(optionsObject), it converts only the specified properties. If the value is true, getCommand converts all available properties, with default values for unspecified properties, to the original syntax.

Note If you set IncludeAll to true, the software translates all available properties, with default values for unspecified properties. The only exception is that when the default value of a property is NaN, Inf, [], '', or "", then the software does not translate the corresponding property.

Example: true

Data Types: logical

IncludeContained — Flag to include transfrags contained by other transfrags

false (default) | true

Flag to include transfrags contained by other transfrags in the same locus in the output combined.gtf, specified as true or false. By default, cuffcompare does not include these contained transfrags. If the value is true, the contained transfrags include a contained_in attribute indicating the first container transfrag found.

Example: true

Data Types: logical

MaxAccuracyRange — Number of bases from terminal exons to use when assessing exon accuracy

100 (default) | positive integer

Number of bases from the free ends of terminal exons to use when assessing exon accuracy, specified as a positive integer.

Example: 80

Data Types: `double`

`MaxGroupingRange` — Number of bases to use for grouping transcript start sites

`100` (default) | positive integer

Number of bases to use for grouping transcript start sites, specified as a positive integer.

Example: `90`

Data Types: `double`

`OutputPrefix` — Prefix for `cuffcompare` output files

`"cuffcmp"` (default) | string | character vector

Prefix for `cuffcompare` output files, specified as a string or character vector. This option must be a string or character vector with a non-zero length.

Example: `"cuffcompareOut"`

Data Types: `char` | `string`

`ReferenceGTF` — Name of GTF or GFF file containing reference transcripts

string | character vector

Name of the GTF or GFF file containing reference transcripts to compare to each sample, specified as a string or character vector. If you provide a file, the function compares each sample to the references in the file and marks isoforms as `overlapping`, `matching`, or `novel`. The function stores these tags in the output files `.refmap` and `.tmap` files.

Example: `"references.gtf"`

Data Types: `char` | `string`

`SequenceDirectory` — Name of directory containing FASTA sequences to classify input transcripts as repeats

string | character vector

Name of directory containing FASTA sequences to classify input transcripts as repeats, specified as a string or character vector. The directory must contain FASTA-format files with the underlying genomic sequences and contain one FASTA file per reference. Name each FASTA file after the chromosome with the extension `.fa` or `.fasta`.

Example: `"./SequenceDirectory/"`

Data Types: `char` | `string`

`SnCorrection` — Flag to consider only reference transcripts that overlap with input transfrags

`false` (default) | `true`

Flag to consider only reference transcripts that overlap with any of the input transfrags, specified as `true` or `false`. If the value is `true`:

- The function ignores any reference transcripts that do not overlap with any of the input transfrags.
- You must also specify the `ReferenceGTF` option.

Example: `true`

Data Types: `logical`

`SpCorrection` — Flag to consider only input transcripts that overlap with reference transcripts

`false` (default) | `true`

Flag to consider only input transcripts that overlap with any of the reference transcripts, specified as `true` or `false`. If the value is `true`:

- The function ignores any input transcripts that do not overlap with any of the reference transcripts and reports no novel loci.
- You must also specify the `ReferenceGTF` option.

Example: `true`

Data Types: `logical`

`SuppressMapFiles` — Flag to prevent creation of `.tmap` and `.refmap` files

`false` (default) | `true`

Flag to prevent the creation of `.tmap` and `.refmap` files, specified as `true` or `false`. Set the value to `true` to prevent the function from generating the files.

Example: `true`

Data Types: `logical`

`Version` — Supported version
string

This property is read-only.

Supported version of the original cufflinks software, returned as a string.

Example: `"2.2.1"`

Data Types: `string`

Object Functions

getCommand	Translate object properties to original options syntax
getOptionsTable	Return table with all properties and equivalent options in original syntax

Examples

Create CuffCompareOptions Object

Create a `CuffCompareOptions` object with the default values.

```
opt = CuffCompareOptions;
```

Create an object using name-value pairs.

```
opt2 = CuffCompareOptions('GenericGFF',true,'MaxAccuracyRange',80)
```

Create an object using the original syntax.

```
opt3 = CuffCompareOptions('-d 100 -e 80')
```

Assemble Transcriptome and Perform Differential Expression Testing

Create a `CufflinksOptions` object to define cufflinks options, such as the number of parallel threads and the output directory to store the results.

```
cflOpt = CufflinksOptions;
cflOpt.NumThreads = 8;
cflOpt.OutputDirectory = "./cufflinksOut";
```

The SAM files provided for this example contain aligned reads for *Mycoplasma pneumoniae* from two samples with three replicates each. The reads are simulated 100bp-reads for two genes (`gyrA` and `gyrB`) located next to each other on the genome. All the reads are sorted by reference position, as required by `cufflinks`.

```
sams = ["Myco_1_1.sam","Myco_1_2.sam","Myco_1_3.sam",...
        "Myco_2_1.sam", "Myco_2_2.sam", "Myco_2_3.sam"];
```

Assemble the transcriptome from the aligned reads.

```
[gtfs,isofpkm,genes,skipped] = cufflinks(sams,cflOpt);
```

`gtfs` is a list of GTF files that contain assembled isoforms.

Compare the assembled isoforms using `cuffcompare`.

```
stats = cuffcompare(gtfs);
```

Merge the assembled transcripts using `cuffmerge`.

```
mergedGTF = cuffmerge(gtfs,'OutputDirectory','./cuffMergeOutput');
```

`mergedGTF` reports only one transcript. This is because the two genes of interest are located next to each other, and `cuffmerge` cannot distinguish two distinct genes. To guide `cuffmerge`, use a reference GTF (`gyrAB.gtf`) containing information about these two genes. If the file is not located in the same directory that you run `cuffmerge` from, you must also specify the file path.

```
gyrAB = which('gyrAB.gtf');
mergedGTF2 = cuffmerge(gtfs,'OutputDirectory','./cuffMergeOutput2',...
            'ReferenceGTF',gyrAB);
```

Calculate abundances (expression levels) from aligned reads for each sample.

```
abundances1 = cuffquant(mergedGTF2,["Myco_1_1.sam","Myco_1_2.sam","Myco_1_3.sam"],...
                        'OutputDirectory','./cuffquantOutput1');
abundances2 = cuffquant(mergedGTF2,["Myco_2_1.sam", "Myco_2_2.sam", "Myco_2_3.sam"],...
                        'OutputDirectory','./cuffquantOutput2');
```

Assess the significance of changes in expression for genes and transcripts between conditions by performing the differential testing using `cuffdiff`. The `cuffdiff` function operates in two distinct steps: the function first estimates abundances from aligned reads, and then performs the statistical analysis. In some cases (for example, distributing computing load across multiple workers), performing the two steps separately is desirable. After performing the first step with `cuffquant`, you can then use the binary CXB output file as an input to `cuffdiff` to perform statistical analysis. Because `cuffdiff` returns several files, specify the output directory is recommended.

```
isoformDiff = cuffdiff(mergedGTF2,[abundances1,abundances2],...
                       'OutputDirectory','./cuffdiffOutput');
```

Display a table containing the differential expression test results for the two genes `gyrB` and `gyrA`.

```
readtable(isoformDiff,'FileType','text')
```

```
ans =

  2×14 table

        test_id            gene_id         gene               locus               sample_1    sample_
    _________________    ______________    ______    _________________________    ________    _______

    'TCONS_00000001'    'XLOC_000001'    'gyrB'    'NC_000912.1:2868-7340'      'q1'        'q2'
    'TCONS_00000002'    'XLOC_000001'    'gyrA'    'NC_000912.1:2868-7340'      'q1'        'q2'
```

You can use `cuffnorm` to generate normalized expression tables for further analyses. `cuffnorm` results are useful when you have many samples and you want to cluster them or plot expression levels for genes that are important in your study. Note that you cannot perform differential expression analysis using `cuffnorm`.

Specify a cell array, where each element is a string vector containing file names for a single sample with replicates.

```
alignmentFiles = {["Myco_1_1.sam","Myco_1_2.sam","Myco_1_3.sam"],...
                  ["Myco_2_1.sam", "Myco_2_2.sam", "Myco_2_3.sam"]}
isoformNorm = cuffnorm(mergedGTF2, alignmentFiles,...
                       'OutputDirectory', './cuffnormOutput');
```

Display a table containing the normalized expression levels for each transcript.

```
readtable(isoformNorm,'FileType','text')
```

```
ans =

  2×7 table

      tracking_id          q1_0          q1_2          q1_1          q2_1          q2_0
    _________________    __________    __________    __________    __________    __________    __

    'TCONS_00000001'    1.0913e+05         78628    1.2132e+05    4.3639e+05    4.2228e+05    4.
    'TCONS_00000002'    3.5158e+05    3.7458e+05    3.4238e+05    1.0483e+05    1.1546e+05    1.
```

Column names starting with *q* have the format: *conditionX_N*, indicating that the column contains values for replicate *N* of *conditionX*.

Version History

Introduced in R2019a

References

[1] Trapnell, Cole, Brian A Williams, Geo Pertea, Ali Mortazavi, Gordon Kwan, Marijke J van Baren, Steven L Salzberg, Barbara J Wold, and Lior Pachter. "Transcript Assembly and Quantification by RNA-Seq Reveals Unannotated Transcripts and Isoform Switching during Cell Differentiation." *Nature Biotechnology* 28, no. 5 (May 2010): 511–15.

See Also

`cuffcompare` | `cufflinks`

Topics

"Bioinformatics Toolbox Software Support Packages"

External Websites

Cufflinks manual

cuffdiff

Identify significant changes in transcript expression

Syntax

```
cuffdiff(transcriptsAnnot,alignmentFiles)
cuffdiff(transcriptsAnnot,alignmentFiles,opt)
cuffdiff(transcriptsAnnot,alignmentFiles,Name,Value)
[isoformsDiff,geneDiff,tssDiff,cdsExp,splicingDiff,cdsDiff,promotersDiff] =
cuffdiff( ___ )
```

Description

`cuffdiff(transcriptsAnnot,alignmentFiles)` identifies significant changes in transcript expression between the samples in `alignmentFiles` using the transcript annotation file `transcriptsAnnot` [1].

`cuffdiff` requires the Cufflinks Support Package for the Bioinformatics Toolbox. If the support package is not installed, then the function provides a download link. For details, see “Bioinformatics Toolbox Software Support Packages”.

`cuffdiff(transcriptsAnnot,alignmentFiles,opt)` uses additional options specified by `opt`.

`cuffdiff(transcriptsAnnot,alignmentFiles,Name,Value)` uses additional options specified by one or more name-value pair arguments. For example, `cuffdiff('gyrAB.gtf', ["Myco_1_1.sam", "Myco_2_1.sam"],'NumThreads',5)` specifies five parallel threads.

`[isoformsDiff,geneDiff,tssDiff,cdsExp,splicingDiff,cdsDiff,promotersDiff] = cuffdiff( ___ )` returns the names of files containing differential expression test results using any of the input argument combinations in the previous syntaxes. By default, the function saves all files to the current directory.

Examples

Assemble Transcriptome and Perform Differential Expression Testing

Create a `CufflinksOptions` object to define cufflinks options, such as the number of parallel threads and the output directory to store the results.

```
cflOpt = CufflinksOptions;
cflOpt.NumThreads = 8;
cflOpt.OutputDirectory = "./cufflinksOut";
```

The SAM files provided for this example contain aligned reads for *Mycoplasma pneumoniae* from two samples with three replicates each. The reads are simulated 100bp-reads for two genes (`gyrA` and `gyrB`) located next to each other on the genome. All the reads are sorted by reference position, as required by `cufflinks`.

```
sams = ["Myco_1_1.sam","Myco_1_2.sam","Myco_1_3.sam",...
        "Myco_2_1.sam", "Myco_2_2.sam", "Myco_2_3.sam"];
```

Assemble the transcriptome from the aligned reads.

```
[gtfs,isofpkm,genes,skipped] = cufflinks(sams,cflOpt);
```

`gtfs` is a list of GTF files that contain assembled isoforms.

Compare the assembled isoforms using `cuffcompare`.

```
stats = cuffcompare(gtfs);
```

Merge the assembled transcripts using `cuffmerge`.

```
mergedGTF = cuffmerge(gtfs,'OutputDirectory','./cuffMergeOutput');
```

`mergedGTF` reports only one transcript. This is because the two genes of interest are located next to each other, and `cuffmerge` cannot distinguish two distinct genes. To guide `cuffmerge`, use a reference GTF (`gyrAB.gtf`) containing information about these two genes. If the file is not located in the same directory that you run `cuffmerge` from, you must also specify the file path.

```
gyrAB = which('gyrAB.gtf');
mergedGTF2 = cuffmerge(gtfs,'OutputDirectory','./cuffMergeOutput2',...
            'ReferenceGTF',gyrAB);
```

Calculate abundances (expression levels) from aligned reads for each sample.

```
abundances1 = cuffquant(mergedGTF2,["Myco_1_1.sam","Myco_1_2.sam","Myco_1_3.sam"],...
                        'OutputDirectory','./cuffquantOutput1');
abundances2 = cuffquant(mergedGTF2,["Myco_2_1.sam", "Myco_2_2.sam", "Myco_2_3.sam"],...
                        'OutputDirectory','./cuffquantOutput2');
```

Assess the significance of changes in expression for genes and transcripts between conditions by performing the differential testing using `cuffdiff`. The `cuffdiff` function operates in two distinct steps: the function first estimates abundances from aligned reads, and then performs the statistical analysis. In some cases (for example, distributing computing load across multiple workers), performing the two steps separately is desirable. After performing the first step with `cuffquant`, you can then use the binary CXB output file as an input to `cuffdiff` to perform statistical analysis. Because `cuffdiff` returns several files, specify the output directory is recommended.

```
isoformDiff = cuffdiff(mergedGTF2,[abundances1,abundances2],...
                      'OutputDirectory','./cuffdiffOutput');
```

Display a table containing the differential expression test results for the two genes `gyrB` and `gyrA`.

```
readtable(isoformDiff,'FileType','text')
```

```
ans =

  2×14 table

        test_id             gene_id         gene               locus               sample_1    sample_
    _________________    _____________    ______    _______________________    ________    _______

    'TCONS_00000001'    'XLOC_000001'    'gyrB'    'NC_000912.1:2868-7340'      'q1'        'q2'
    'TCONS_00000002'    'XLOC_000001'    'gyrA'    'NC_000912.1:2868-7340'      'q1'        'q2'
```

You can use `cuffnorm` to generate normalized expression tables for further analyses. `cuffnorm` results are useful when you have many samples and you want to cluster them or plot expression

levels for genes that are important in your study. Note that you cannot perform differential expression analysis using `cuffnorm`.

Specify a cell array, where each element is a string vector containing file names for a single sample with replicates.

```
alignmentFiles = {["Myco_1_1.sam","Myco_1_2.sam","Myco_1_3.sam"],...
                  ["Myco_2_1.sam", "Myco_2_2.sam", "Myco_2_3.sam"]}
isoformNorm = cuffnorm(mergedGTF2, alignmentFiles,...
                      'OutputDirectory', './cuffnormOutput');
```

Display a table containing the normalized expression levels for each transcript.

```
readtable(isoformNorm,'FileType','text')
```

```
ans =

  2×7 table

      tracking_id         q1_0          q1_2          q1_1          q2_1          q2_0        
    ________________    __________    __________    __________    __________    __________    __

    'TCONS_00000001'    1.0913e+05         78628    1.2132e+05    4.3639e+05    4.2228e+05    4.2
    'TCONS_00000002'    3.5158e+05    3.7458e+05    3.4238e+05    1.0483e+05    1.1546e+05    1.1
```

Column names starting with *q* have the format: *conditionX_N*, indicating that the column contains values for replicate *N* of *conditionX*.

Input Arguments

`transcriptsAnnot` — Name of transcript annotation file

string | character vector

Name of the transcript annotation file, specified as a string or character vector. The file can be a GTF or GFF file produced by `cufflinks`, `cuffcompare`, or another source of GTF annotations.

Example: `"gyrAB.gtf"`

Data Types: `char` | `string`

`alignmentFiles` — Names of SAM, BAM, or CXB files

string vector | cell array

Names of SAM, BAM, or CXB files containing alignment records for each sample, specified as a string vector or cell array. If you use a cell array, each element must be a string vector or cell array of character vectors specifying alignment files for every replicate of the same sample.

Example: `["Myco_1_1.sam", "Myco_2_1.sam"]`

Data Types: `char` | `string` | `cell`

opt — `cuffdiff` options

`CuffDiffOptions` object | string | character vector

`cuffdiff` options, specified as a `CuffDiffOptions` object, string, or character vector. The string or character vector must be in the original `cuffdiff` option syntax (prefixed by one or two dashes) [1].

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `cuffdiff('gyrAB.gtf',["Myco_1_1.sam", "Myco_2_1.sam"],'NumThreads',5,'DispersionMethod',"per-condition")`

`ConditionLabels` — Sample labels

string | string vector | character vector | cell array of character vectors

Sample labels, specified as a string, string vector, character vector, or cell array of character vectors. The number of labels must equal the number of samples or the value must be empty `[]`.

Example: `'ConditionLabels',["Control","Mutant1","Mutant2"]`

Data Types: `string` | `char` | `cell`

`ContrastFile` — Contrast file name

string | character vector

Contrast file name, specified as a string or character vector. The file must be a two-column tab-delimited text file, where each line indicates two conditions to compare using `cuffdiff`. The condition labels in the file must match either the labels specified for `ConditionLabels` or the sample names. The file must have a single header line as the first line, followed by one line for each contrast. An example of the contrast file format follows.

condition_A	condition_B
Control	Mutant1
Control	Mutant2

If you do not provide this file, `cuffdiff` compares every pair of input conditions, which can impact performance.

Example: `'ContrastFile',"contrast.txt"`

Data Types: `char` | `string`

`DispersionMethod` — Method to model variance in fragment counts

`"pooled"` (default) | `"per-condition"` | `"blind"` | `"poisson"`

Method to model the variance in fragment counts across replicates, specified as one of the following options:

- `"pooled"` — The function uses each replicated condition to build a model and averages these models into a global model for all conditions in the experiment.
- `"per-condition"` — The function produces a model for each condition. You can use this option only if all conditions have replicates.
- `"blind"` — The function treats all samples as replicates of a single global distribution and produces one model.
- `"poisson"` — Variance in fragment counts is a poisson model, where the fragment count is predicted to be the mean across replicates. This method is not recommended.

Select a method depending on whether you expect the variability in each group of samples to be similar.

- When comparing two groups where the first group has low cross-replicate variability and the second group has high variability, choose the `per-condition` method.
- If the conditions have similar levels of variability, choose the `pooled` method.
- If you have only a single replicate in each condition, choose the `blind` method.

Example: `'DispersionMethod',"blind"`

Data Types: `char` | `string`

`DoIsoformSwitch` — Flag to perform isoform switching tests
`true` (default) | `false`

Flag to perform isoform switching tests, specified as `true` or `false`. These tests estimate how much differential splicing exists in isoforms from a single primary transcript. By default, the value is `true` and the test results are saved in the output file `splicing.diff`.

Example: `'DoIsoformSwitch',false`

Data Types: `logical`

`EffectiveLengthCorrection` — Flag to normalize fragment counts
`true` (default) | `false`

Flag to normalize fragment counts to fragments per kilobase per million mapped reads (FPKM), specified as `true` or `false`.

Example: `'EffectiveLengthCorrection',false`

Data Types: `logical`

`ExtraCommand` — Additional commands
`""` (default) | string | character vector

The commands must be in the native syntax (prefixed by one or two dashes). Use this option to apply undocumented flags and flags without corresponding MATLAB properties.

Example: `'ExtraCommand','--library-type fr-secondstrand'`

Data Types: `char` | `string`

`FalseDiscoveryRate` — False discovery rate
`0.05` (default) | scalar between `0` and `1`

False discovery rate to use during statistical tests, specified as a scalar between `0` and `1`.

Example: `'FalseDiscoveryRate',0.01`

Data Types: `double`

`FragmentBiasCorrection` — Name of FASTA file with reference transcripts to detect bias
string | character vector

Name of the FASTA file with reference transcripts to detect bias in fragment counts, specified as a string or character vector. Library preparation can introduce sequence-specific bias into RNA-Seq experiments. Providing reference transcripts improves the accuracy of the transcript abundance estimates.

Example: `'FragmentBiasCorrection',"bias.fasta"`

Data Types: `char` | `string`

`FragmentLengthMean` — Expected mean fragment length in base pairs
`200` (default) | positive integer

Expected mean fragment length, specified as a positive integer. The default value is `200` base pairs. The function can learn the fragment length mean for each SAM file. Using this option is not recommended for paired-end reads.

Example: `'FragmentLengthMean',100`

Data Types: `double`

`FragmentLengthSD` — Expected standard deviation for fragment length distribution
`80` (default) | positive scalar

Expected standard deviation for the fragment length distribution, specified as a positive scalar. The default value is `80` base pairs. The function can learn the fragment length standard deviation for each SAM file. Using this option is not recommended for paired-end reads.

Example: `'FragmentLengthSD',70`

Data Types: `double`

`GenerateAnalysisDiff` — Flag to create differential analysis files
`true` (default) | `false`

Flag to create differential analysis files (`*.diff`), specified as `true` or `false`.

Example: `'GenerateAnalysisDiff',false`

Data Types: `logical`

`IncludeAll` — Flag to include all object properties
`false` (default) | `true`

Flag to include all the object properties with the corresponding default values when converting to the original options syntax, specified as `true` or `false`. You can convert the properties to the original syntax prefixed by one or two dashes (such as `'-d 100 -e 80'`) by using `getCommand`. The default value `false` means that when you call `getCommand(optionsObject)`, it converts only the specified properties. If the value is `true`, `getCommand` converts all available properties, with default values for unspecified properties, to the original syntax.

Note If you set `IncludeAll` to `true`, the software translates all available properties, with default values for unspecified properties. The only exception is that when the default value of a property is `NaN`, `Inf`, `[]`, `''`, or `""`, then the software does not translate the corresponding property.

Example: `'IncludeAll',true`

Data Types: `logical`

`IsoformShiftReplicates` — Minimum number of replicates to test genes for differential regulation
`3` (default) | positive integer

Minimum number of replicates to test genes for differential regulation, specified as a positive integer. The function skips the tests when the number of replicates is smaller than the specified value.

Example: `'IsoformShiftReplicates',2`

Data Types: `double`

`LengthCorrection` — Flag to correct by transcript length
`true` (default) | `false`

Flag to correct by the transcript length, specified as `true` or `false`. Set this value to `false` only when the fragment count is independent of the feature size, such as for small RNA libraries with no fragmentation and for 3' end sequencing, where all fragments have the same length.

Example: `'LengthCorrection',false`

Data Types: `logical`

`LibraryNormalizationMethod` — Method to normalize library size
`"geometric"` (default) | `"classic-fpkm"` | `"quartile"`

Method to normalize the library size, specified as one of the following options:

- `"geometric"` — The function scales the FPKM values by the median geometric mean of fragment counts across all libraries as described in [2].
- `"classic-fpkm"` — The function applies no scaling to the FPKM values or fragment counts.
- `"quartile"` — The function scales the FPKM values by the ratio of upper quartiles between fragment counts and the average value across all libraries.

Example: `'LibraryNormalizationMethod',"classic-fpkm"`

Data Types: `char` | `string`

`MaskFile` — Name of GTF or GFF file containing transcripts to ignore
string | character vector

Name of the GTF or GFF file containing transcripts to ignore during analysis, specified as a string or character vector. Some examples of transcripts to ignore include annotated rRNA transcripts, mitochondrial transcripts, and other abundant transcripts. Ignoring these transcripts improves the robustness of the abundance estimates.

Example: `'MaskFile',"excludes.gtf"`

Data Types: `char` | `string`

`MaxBundleFrags` — Maximum number of fragments to include for each locus before skipping
`500000` (default) | positive integer

Maximum number of fragments to include for each locus before skipping new fragments, specified as a positive integer. Skipped fragments are marked with the status `HIDATA` in the file `skipped.gtf`.

Example: `'MaxBundleFrags',400000`

Data Types: `double`

`MaxFragAlignments` — Maximum number of aligned reads to include for each fragment
`Inf` (default) | positive integer

Maximum number of aligned reads to include for each fragment before skipping new reads, specified as a positive integer. `Inf`, the default value, sets no limit on the maximum number of aligned reads.

Example: `'MaxFragAlignments',1000`

Data Types: `double`

`MaxMLEIterations` — Maximum number of iterations for maximum likelihood estimation
`5000` (default) | positive integer

Maximum number of iterations for the maximum likelihood estimation of abundances, specified as a positive integer.

Example: `'MaxMLEIterations',4000`

Data Types: `double`

`MinAlignmentCount` — Minimum number of alignments required in locus for significance testing
`10` (default) | positive integer

Minimum number of alignments required in a locus to perform the significance testing for differences between samples, specified as a positive integer.

Example: `'MinAlignmentCount',8`

Data Types: `double`

`MinIsoformFraction` — Minimum abundance of isoform to include in differential expression tests
`1e-5` (default) | scalar between `0` and `1`

Minimum abundance of an isoform to include in differential expression tests, specified as a scalar between `0` and `1`. For alternative isoforms quantified at below the specified value, the function rounds down the abundance to zero. The specified value is a fraction of the major isoform. The function performs this filtering after MLE estimation but before MAP estimation to improve the robustness of confidence interval generation and differential expression analysis. Using a parameter value other than the default is not recommended.

Example: `'MinIsoformFraction',1e-5`

Data Types: `double`

`MultiReadCorrection` — Flag to improve abundance estimation using rescue method
`false` (default) | `true`

Flag to improve abundance estimation for reads mapped to multiple genomic positions using the rescue method, specified as `true` or `false`. If the value is `false`, the function divides multimapped reads uniformly to all mapped positions. If the value is `true`, the function uses additional information, including gene abundance estimation, inferred fragment length, and fragment bias, to improve transcript abundance estimation.

The rescue method is described in [3].

Example: `'MultiReadCorrection',true`

Data Types: `logical`

`NormalizeCompatibleHits` — Flag to use only fragments compatible with reference transcript to calculate FPKM values
`true` (default) | `false`

Flag to use only fragments compatible with a reference transcript to calculate FPKM values, specified as `true` or `false`.

Example: `'NormalizeCompatibleHits',false`

Data Types: `logical`

`NormalizeTotalHits` — Flag to include all fragments to calculate FPKM values
`false` (default) | `true`

Flag to include all fragments to calculate FPKM values, specified as `true` or `false`. If the value is `true`, the function includes all fragments, including fragments without a compatible reference.

Example: `'NormalizeTotalHits',true`

Data Types: `logical`

`NumFragAssignmentDraws` — Number of fragment assignments to perform on each transcript
`50` (default) | positive integer

Number of fragment assignments to perform on each transcript, specified as a positive integer. For each fragment drawn from a transcript, the function performs the specified number of assignments probabilistically to determine the transcript assignment uncertainty and to estimate the variance-covariance matrix for the assigned fragment counts.

Example: `'NumFragAssignmentSamples',40`

Data Types: `double`

`NumFragDraws` — Number of draws from negative binomial random number generator
`100` (default) | positive integer

Number of draws from the negative binomial random number generator for each transcript, specified as a positive integer. Each draw is a number of fragments that the function probabilistically assigns to transcripts in the transcriptome to determine the assignment uncertainty and to estimate the variance-covariance matrix for assigned fragment counts.

Example: `'NumFragSamples',90`

Data Types: `double`

`NumThreads` — Number of parallel threads to use
`1` (default) | positive integer

Number of parallel threads to use, specified as a positive integer. Threads are run on separate processors or cores. Increasing the number of threads generally improves the runtime significantly, but increases the memory footprint.

Example: `'NumThreads',4`

Data Types: `double`

`OutputDirectory` — Directory to store analysis results
current directory (`"./"`) (default) | string | character vector

Directory to store analysis results, specified as a string or character vector.

Example: `'OutputDirectory',"./AnalysisResults/"`

Data Types: `char` | `string`

Seed — Seed for random number generator
`0` (default) | nonnegative integer

Seed for the random number generator, specified as a nonnegative integer. Setting a seed value ensures the reproducibility of the analysis results.

Example: `'Seed',10`

Data Types: `double`

`TimeSeries` — Flag to treat input samples as time series
`false` (default) | `true`

Flag to treat input samples as a time series rather than as independent experimental conditions, specified as `true` or `false`. If you set the value to `true`, you must provide samples in order of increasing time: the first SAM file must be for the first time point, the second SAM file for the second time point, and so on.

Example: `'TimeSeries',true`

Data Types: `logical`

Output Arguments

`isoformsDiff` — Name of file containing transcript-level differential expression results
`"./isoform_exp.diff"`

Name of a file containing transcript-level differential expression results, returned as a string.

The output string also includes the directory information defined by `OutputDirectory`. The default is the current directory. If you set `OutputDirectory` to `"/local/tmp/"`, the output becomes `"/local/tmp/isoform_exp.diff"`.

geneDiff — Name of file containing gene-level differential expression results
`"./gene_exp.diff"`

Name of a file containing gene-level differential expression results, returned as a string.

The output string also includes the directory information defined by `OutputDirectory`. The default is the current directory. If you set `OutputDirectory` to `"/local/tmp/"`, the output becomes `"/local/tmp/gene_exp.diff"`.

`tssDiff` — Name of file containing primary transcript differential expression results
`"./tss_group_exp.diff"`

Name of a file containing primary transcript differential expression results, returned as a string.

The output string also includes the directory information defined by `OutputDirectory`. The default is the current directory. If you set `OutputDirectory` to `"/local/tmp/"`, the output becomes `"/local/tmp/tss_group_exp.diff"`.

`cdsExp` — Name of file containing coding sequence differential expression results
`"./cds_exp.diff"`

Name of a file containing coding sequence differential expression results, returned as a string.

The output string also includes the directory information defined by `OutputDirectory`. The default is the current directory. If you set `OutputDirectory` to `"/local/tmp/"`, the output becomes `"/local/tmp/cds_exp.diff"`.

`splicingDiff` — Name of file containing differential splicing results for isoforms
`"./splicing.diff"`

Name of a file containing differential splicing results for isoforms, returned as a string.

The output string also includes the directory information defined by `OutputDirectory`. The default is the current directory. If you set `OutputDirectory` to `"/local/tmp/"`, the output becomes `"/local/tmp/splicing.diff"`.

`cdsDiff` — Name of file containing differential coding sequence output
`"./cds.diff"`

Name of a file containing differential coding sequence output, returned as a string.

The output string also includes the directory information defined by `OutputDirectory`. The default is the current directory. If you set `OutputDirectory` to `"/local/tmp/"`, the output becomes `"/local/tmp/cds.diff"`.

`promotersDiff` — Name of file containing information on differential promoter use
`"./promoters.diff"`

Name of a file containing information on differential promoter use that exists between samples, returned as a string.

The output string also includes the directory information defined by `OutputDirectory`. The default is the current directory. If you set `OutputDirectory` to `"/local/tmp/"`, the output becomes `"/local/tmp/promoters.diff"`.

Version History

Introduced in R2019a

References

[1] Trapnell, Cole, Brian A Williams, Geo Pertea, Ali Mortazavi, Gordon Kwan, Marijke J van Baren, Steven L Salzberg, Barbara J Wold, and Lior Pachter. "Transcript Assembly and Quantification by RNA-Seq Reveals Unannotated Transcripts and Isoform Switching during Cell Differentiation." *Nature Biotechnology* 28, no. 5 (May 2010): 511–15.

[2] Anders, Simon, and Wolfgang Huber. "Differential Expression Analysis for Sequence Count Data." *Genome Biology* 11, no. 10 (October 2010): R106. https://doi.org/10.1186/gb-2010-11-10-r106.

[3] Mortazavi, Ali, Brian A Williams, Kenneth McCue, Lorian Schaeffer, and Barbara Wold. “Mapping and Quantifying Mammalian Transcriptomes by RNA-Seq.” *Nature Methods* 5, no. 7 (July 2008): 621–28. https://doi.org/10.1038/nmeth.1226.

See Also

`CuffDiffOptions` | `cufflinks`

Topics

“Bioinformatics Toolbox Software Support Packages”

External Websites

Cufflinks manual

CuffDiffOptions

Option set for `cuffdiff`

Description

A `CuffDiffOptions` object sets options for the `cuffdiff` function, which identifies significant changes in transcript expression [1].

Creation

Syntax

```
cuffdiffOpt = CuffDiffOptions
cuffdiffOpt = CuffDiffOptions(Name,Value)
cuffdiffOpt = CuffDiffOptions(S)
```

Description

`cuffdiffOpt = CuffDiffOptions` creates a `CuffDiffOptions` object with the default property values.

`CuffDiffOptions` requires the Cufflinks Support Package for the Bioinformatics Toolbox. If the support package is not installed, then the function provides a download link. For details, see “Bioinformatics Toolbox Software Support Packages”.

`cuffdiffOpt = CuffDiffOptions(Name,Value)` sets the object properties on page 1-635 using one or more name-value pair arguments. Enclose each property name in quotes. For example, `cuffdiffOpt = CuffDiffOptions('SupressMapFiles',true)` prevents the creation of `.tmap` and `.refmap` files.

`cuffdiffOpt = CuffDiffOptions(S)` specifies optional parameters using a string or character vector `S`.

Input Arguments

S — `cuffdiff` options
string | character vector

`cuffdiff` options, specified as a string or character vector. `S` must be in the original `cuffdiff` option syntax (prefixed by one or two dashes).

Example: `'--seed 5'`

Properties

`ConditionLabels` — Sample labels

string | string vector | character vector | cell array of character vector

Sample labels, specified as a string, string vector, character vector, or cell array of character vectors. The number of labels must equal the number of samples or the value must be empty `[]`.

Example: `["Control","Mutant1","Mutant2"]`

Data Types: `string` | `char` | `cell`

`ContrastFile` — Contrast file name

string | character vector

Contrast file name, specified as a string or character vector. The file must be a two-column tab-delimited text file, where each line indicates two conditions to compare using `cuffdiff`. The condition labels in the file must match either the labels specified for `ConditionLabels` or the sample names. The file must have a single header line as the first line, followed by one line for each contrast. An example of the contrast file format follows.

condition_A	condition_B
Control	Mutant1
Control	Mutant2

If you do not provide this file, `cuffdiff` compares every pair of input conditions, which can impact performance.

Example: `"contrast.txt"`

Data Types: `char` | `string`

`DispersionMethod` — Method to model variance in fragment counts

`"pooled"` (default) | `"per-condition"` | `"blind"` | `"poisson"`

Method to model the variance in fragment counts across replicates, specified as one of the following options:

- `"pooled"` — The function uses each replicated condition to build a model and averages these models into a global model for all conditions in the experiment.
- `"per-condition"` — The function produces a model for each condition. You can use this option only if all conditions have replicates.
- `"blind"` — The function treats all samples as replicates of a single global distribution and produces one model.
- `"poisson"` — Variance in fragment counts is a poisson model, where the fragment count is predicted to be the mean across replicates. This method is not recommended.

Select a method depending on whether you expect the variability in each group of samples to be similar.

- When comparing two groups where the first group has low cross-replicate variability and the second group has high variability, choose the `per-condition` method.
- If the conditions have similar levels of variability, choose the `pooled` method.
- If you have only a single replicate in each condition, choose the `blind` method.

Example: `"blind"`

Data Types: `char` | `string`

`DoIsoformSwitch` — Flag to perform isoform switching tests

`true` (default) | `false`

Flag to perform isoform switching tests, specified as `true` or `false`. These tests estimate how much differential splicing exists in isoforms from a single primary transcript. By default, the value is `true` and the test results are saved in the output file `splicing.diff`.

Example: `false`

Data Types: `logical`

`EffectiveLengthCorrection` — Flag to normalize fragment counts

`true` (default) | `false`

Flag to normalize fragment counts to fragments per kilobase per million mapped reads (FPKM), specified as `true` or `false`.

Example: `false`

Data Types: `logical`

`ExtraCommand` — Additional commands

`""` (default) | string | character vector

The commands must be in the native syntax (prefixed by one or two dashes). Use this option to apply undocumented flags and flags without corresponding MATLAB properties.

When the software converts the original flags to MATLAB properties, it stores any unrecognized flags in this property.

Example: `'--library-type fr-secondstrand'`

Data Types: `char` | `string`

`FalseDiscoveryRate` — False discovery rate

`0.05` (default) | scalar between `0` and `1`

False discovery rate used during statistical tests, specified as a scalar between `0` and `1`.

Example: `0.01`

Data Types: `double`

`FragmentBiasCorrection` — Name of FASTA file with reference transcripts to detect bias

string | character vector

Name of the FASTA file with reference transcripts to detect bias in fragment counts, specified as a string or character vector. Library preparation can introduce sequence-specific bias into RNA-Seq experiments. Providing reference transcripts improves the accuracy of the transcript abundance estimates.

Example: `"bias.fasta"`

Data Types: `char` | `string`

`FragmentLengthMean` — Expected mean fragment length in base pairs

`200` (default) | positive integer

Expected mean fragment length, specified as a positive integer. The default value is `200` base pairs. The function can learn the fragment length mean for each SAM file. Using this option is not recommended for paired-end reads.

Example: `100`

Data Types: `double`

`FragmentLengthSD` — Expected standard deviation for fragment length distribution

`80` (default) | positive scalar

Expected standard deviation for the fragment length distribution, specified as a positive scalar. The default value is `80` base pairs. The function can learn the fragment length standard deviation for each SAM file. Using this option is not recommended for paired-end reads.

Example: `70`

Data Types: `double`

`GenerateAnalysisDiff` — Flag to create differential analysis files

`true` (default) | `false`

Flag to create differential analysis files (`*.diff`), specified as `true` or `false`.

Example: `false`

Data Types: `logical`

`IncludeAll` — Flag to use all object properties

`false` (default) | `true`

Flag to include all the object properties with the corresponding default values when converting to the original options syntax, specified as `true` or `false`. You can convert the properties to the original syntax prefixed by one or two dashes (such as `'-d 100 -e 80'`) by using `getCommand`. The default value `false` means that when you call `getCommand(optionsObject)`, it converts only the specified properties. If the value is `true`, `getCommand` converts all available properties, with default values for unspecified properties, to the original syntax.

Note If you set `IncludeAll` to `true`, the software translates all available properties, with default values for unspecified properties. The only exception is that when the default value of a property is `NaN`, `Inf`, `[]`, `''`, or `""`, then the software does not translate the corresponding property.

Example: `true`

Data Types: `logical`

`IsoformShiftReplicates` — Minimum number of replicates to test genes for differential regulation

`3` (default) | positive integer

Minimum number of replicates to test genes for differential regulation, specified as a positive integer. The function skips the tests when the number of replicates is smaller than the specified value.

Example: `2`

Data Types: `double`

`LengthCorrection` — Flag to correct by transcript length

`true` (default) | `false`

Flag to correct by the transcript length, specified as `true` or `false`. Set this value to `false` only when the fragment count is independent of the feature size, such as for small RNA libraries with no fragmentation and for 3' end sequencing, where all fragments have the same length.

Example: `false`

Data Types: `logical`

`LibraryNormalizationMethod` — Method to normalize library size

`"geometric"` (default) | `"classic-fpkm"` | `"quartile"`

Method to normalize the library size, specified as one of the following options:

- `"geometric"` — The function scales the FPKM values by the median geometric mean of fragment counts across all libraries as described in [2].
- `"classic-fpkm"` — The function applies no scaling to the FPKM values or fragment counts.
- `"quartile"` — The function scales the FPKM values by the ratio of upper quartiles between fragment counts and the average value across all libraries.

Example: `"classic-fpkm"`

Data Types: `char` | `string`

`MaskFile` — Name of GTF or GFF file containing transcripts to ignore

string | character vector

Name of the GTF or GFF file containing transcripts to ignore during analysis, specified as a string or character vector. Some examples of transcripts to ignore include annotated rRNA transcripts, mitochondrial transcripts, and other abundant transcripts. Ignoring these transcripts improves the robustness of the abundance estimates.

Example: `"excludes.gtf"`

Data Types: `char` | `string`

`MaxBundleFrags` — Maximum number of fragments to include for each locus before skipping

`500000` (default) | positive integer

Maximum number of fragments to include for each locus before skipping new fragments, specified as a positive integer. Skipped fragments are marked with the status HIDATA in the file `skipped.gtf`.

Example: `400000`

Data Types: `double`

`MaxFragAlignments` — Maximum number of aligned reads to include for each fragment

`Inf` (default) | positive integer

Maximum number of aligned reads to include for each fragment before skipping new reads, specified as a positive integer. `Inf`, the default value, sets no limit on the maximum number of aligned reads.

Example: `1000`

Data Types: `double`

`MaxMLEIterations` — Maximum number of iterations for maximum likelihood estimation

`5000` (default) | positive integer

Maximum number of iterations for the maximum likelihood estimation of abundances, specified as a positive integer.

Example: `4000`

Data Types: `double`

`MinAlignmentCount` — Minimum number of alignments required in locus for significance testing

`10` (default) | positive integer

Minimum number of alignments required in a locus to perform the significance testing for differences between samples, specified as a positive integer.

Example: `8`

Data Types: `double`

`MinIsoformFraction` — Minimum abundance of isoform to include in differential expression tests

`1e-5` (default) | scalar between `0` and `1`

Minimum abundance of an isoform to include in differential expression tests, specified as a scalar between `0` and `1`. For alternative isoforms quantified at below the specified value, the function rounds down the abundance to zero. The specified value is a fraction of the major isoform. The function performs this filtering after MLE estimation but before MAP estimation to improve the robustness of confidence interval generation and differential expression analysis. Using a parameter value other than the default is not recommended.

Example: `1e-5`

Data Types: `double`

`MultiReadCorrection` — Flag to improve abundance estimation using rescue method

`false` (default) | `true`

Flag to improve abundance estimation for reads mapped to multiple genomic positions using the rescue method, specified as `true` or `false`. If the value is `false`, the function divides multimapped reads uniformly to all mapped positions. If the value is `true`, the function uses additional information, including gene abundance estimation, inferred fragment length, and fragment bias, to improve transcript abundance estimation.

The rescue method is described in [3].

Example: `true`

Data Types: `logical`

`NormalizeCompatibleHits` — Flag to use only fragments compatible with reference transcript to calculate FPKM values

`true` (default) | `false`

Flag to use only fragments compatible with a reference transcript to calculate FPKM values, specified as `true` or `false`.

Example: `false`

Data Types: `logical`

`NormalizeTotalHits` — Flag to include all fragments to calculate FPKM values

`false` (default) | `true`

Flag to include all fragments to calculate FPKM values, specified as `true` or `false`. If the value is `true`, the function includes all fragments, including fragments without a compatible reference.

Example: `true`

Data Types: `logical`

`NumFragAssignmentDraws` — Number of fragment assignments to perform on each transcript

`50` (default) | positive integer

Number of fragment assignments to perform on each transcript, specified as a positive integer. For each fragment drawn from a transcript, the function performs the specified number of assignments probabilistically to determine the transcript assignment uncertainty and to estimate the variance-covariance matrix for the assigned fragment counts.

Example: `40`

Data Types: `double`

`NumFragDraws` — Number of draws from negative binomial random number generator

`100` (default) | positive integer

Number of draws from the negative binomial random number generator for each transcript, specified as a positive integer. Each draw is a number of fragments that the function probabilistically assigns to transcripts in the transcriptome to determine the assignment uncertainty and to estimate the variance-covariance matrix for assigned fragment counts.

Example: `90`

Data Types: `double`

`NumThreads` — Number of parallel threads to use

`1` (default) | positive integer

Number of parallel threads to use, specified as a positive integer. Threads are run on separate processors or cores. Increasing the number of threads generally improves the runtime significantly, but increases the memory footprint.

Example: `4`

Data Types: `double`

`OutputDirectory` — Directory to store analysis results

current directory (`"./"`) (default) | string | character vector

Directory to store analysis results, specified as a string or character vector.

Example: `"./AnalysisResults/"`

Data Types: `char` | `string`

`Seed` — Seed for random number generator

`0` (default) | nonnegative integer

Seed for the random number generator, specified as a nonnegative integer. Setting a seed value ensures the reproducibility of the analysis results.

Example: `10`

Data Types: `double`

`TimeSeries` — Flag to treat input samples as time series

`false` (default) | `true`

Flag to treat input samples as a time series rather than as independent experimental conditions, specified as `true` or `false`. If you set the value to `true`, you must provide samples in order of increasing time: the first SAM file must be for the first time point, the second SAM file for the second time point, and so on.

Example: `true`

Data Types: `logical`

`Version` — Supported version
string

This property is read-only.

Supported version of the original cufflinks software, returned as a string.

Example: "2.2.1"

Data Types: string

Object Functions

getCommand	Translate object properties to original options syntax
getOptionsTable	Return table with all properties and equivalent options in original syntax

Examples

Create CuffDiffOptions Object

Create a `CuffDiffOptions` object with the default values.

```
opt = CuffDiffOptions;
```

Create an object using name-value pairs.

```
opt2 = CuffDiffOptions('FalseDiscoveryRate',0.01,'NumThreads',4)
```

Create an object by using the original syntax.

```
opt3 = CuffDiffOptions('--FDR 0.01 --num-threads 4')
```

Assemble Transcriptome and Perform Differential Expression Testing

Create a `CufflinksOptions` object to define cufflinks options, such as the number of parallel threads and the output directory to store the results.

```
cflOpt = CufflinksOptions;
cflOpt.NumThreads = 8;
cflOpt.OutputDirectory = "./cufflinksOut";
```

The SAM files provided for this example contain aligned reads for *Mycoplasma pneumoniae* from two samples with three replicates each. The reads are simulated 100bp-reads for two genes (`gyrA` and `gyrB`) located next to each other on the genome. All the reads are sorted by reference position, as required by `cufflinks`.

```
sams = ["Myco_1_1.sam","Myco_1_2.sam","Myco_1_3.sam",...
        "Myco_2_1.sam", "Myco_2_2.sam", "Myco_2_3.sam"];
```

Assemble the transcriptome from the aligned reads.

```
[gtfs,isofpkm,genes,skipped] = cufflinks(sams,cflOpt);
```

`gtfs` is a list of GTF files that contain assembled isoforms.

Compare the assembled isoforms using `cuffcompare`.

```
stats = cuffcompare(gtfs);
```

Merge the assembled transcripts using `cuffmerge`.

```
mergedGTF = cuffmerge(gtfs,'OutputDirectory','./cuffMergeOutput');
```

mergedGTF reports only one transcript. This is because the two genes of interest are located next to each other, and cuffmerge cannot distinguish two distinct genes. To guide cuffmerge, use a reference GTF (gyrAB.gtf) containing information about these two genes. If the file is not located in the same directory that you run cuffmerge from, you must also specify the file path.

```
gyrAB = which('gyrAB.gtf');
mergedGTF2 = cuffmerge(gtfs,'OutputDirectory','./cuffMergeOutput2',...
            'ReferenceGTF',gyrAB);
```

Calculate abundances (expression levels) from aligned reads for each sample.

```
abundances1 = cuffquant(mergedGTF2,["Myco_1_1.sam","Myco_1_2.sam","Myco_1_3.sam"],...
                        'OutputDirectory','./cuffquantOutput1');
abundances2 = cuffquant(mergedGTF2,["Myco_2_1.sam", "Myco_2_2.sam", "Myco_2_3.sam"],...
                        'OutputDirectory','./cuffquantOutput2');
```

Assess the significance of changes in expression for genes and transcripts between conditions by performing the differential testing using cuffdiff. The cuffdiff function operates in two distinct steps: the function first estimates abundances from aligned reads, and then performs the statistical analysis. In some cases (for example, distributing computing load across multiple workers), performing the two steps separately is desirable. After performing the first step with cuffquant, you can then use the binary CXB output file as an input to cuffdiff to perform statistical analysis. Because cuffdiff returns several files, specify the output directory is recommended.

```
isoformDiff = cuffdiff(mergedGTF2,[abundances1,abundances2],...
                      'OutputDirectory','./cuffdiffOutput');
```

Display a table containing the differential expression test results for the two genes gyrB and gyrA.

```
readtable(isoformDiff,'FileType','text')
```

```
ans =

  2×14 table

        test_id            gene_id         gene              locus              sample_1     sample_
    ________________    _____________    ______    _________________________    ________    _______

    'TCONS_00000001'    'XLOC_000001'    'gyrB'    'NC_000912.1:2868-7340'        'q1'        'q2'
    'TCONS_00000002'    'XLOC_000001'    'gyrA'    'NC_000912.1:2868-7340'        'q1'        'q2'
```

You can use cuffnorm to generate normalized expression tables for further analyses. cuffnorm results are useful when you have many samples and you want to cluster them or plot expression levels for genes that are important in your study. Note that you cannot perform differential expression analysis using cuffnorm.

Specify a cell array, where each element is a string vector containing file names for a single sample with replicates.

```
alignmentFiles = {["Myco_1_1.sam","Myco_1_2.sam","Myco_1_3.sam"],...
                  ["Myco_2_1.sam", "Myco_2_2.sam", "Myco_2_3.sam"]}
isoformNorm = cuffnorm(mergedGTF2, alignmentFiles,...
                      'OutputDirectory', './cuffnormOutput');
```

Display a table containing the normalized expression levels for each transcript.

```
readtable(isoformNorm,'FileType','text')
```

```
ans =

  2×7 table

      tracking_id         q1_0          q1_2          q1_1          q2_1          q2_0
    _________________    __________    __________    __________    __________    __________    __

    'TCONS_00000001'    1.0913e+05         78628    1.2132e+05    4.3639e+05    4.2228e+05    4.
    'TCONS_00000002'    3.5158e+05    3.7458e+05    3.4238e+05    1.0483e+05    1.1546e+05    1.
```

Column names starting with *q* have the format: *conditionX_N*, indicating that the column contains values for replicate *N* of *conditionX*.

Version History

Introduced in R2019a

References

[1] Trapnell, Cole, Brian A Williams, Geo Pertea, Ali Mortazavi, Gordon Kwan, Marijke J van Baren, Steven L Salzberg, Barbara J Wold, and Lior Pachter. “Transcript Assembly and Quantification by RNA-Seq Reveals Unannotated Transcripts and Isoform Switching during Cell Differentiation.” *Nature Biotechnology* 28, no. 5 (May 2010): 511–15.

[2] Anders, Simon, and Wolfgang Huber. “Differential Expression Analysis for Sequence Count Data.” *Genome Biology* 11, no. 10 (October 2010): R106. https://doi.org/10.1186/gb-2010-11-10-r106.

[3] Mortazavi, Ali, Brian A Williams, Kenneth McCue, Lorian Schaeffer, and Barbara Wold. “Mapping and Quantifying Mammalian Transcriptomes by RNA-Seq.” *Nature Methods* 5, no. 7 (July 2008): 621–28. https://doi.org/10.1038/nmeth.1226.

See Also

`cufflinks` | `CufflinksOptions` | `cuffcompare` | `cuffdiff` | `cuffmerge` | `cuffnorm` | `cuffquant` | `cuffgtf2sam` | `cuffgffread`

Topics

“Bioinformatics Toolbox Software Support Packages”

External Websites

Cufflinks manual

cuffgffread

Filter and convert GFF and GTF files

Syntax

```
cuffgffread(input,output)
cuffgffread(input,output,opt)
cuffgffread(input,output,Name,Value)
```

Description

`cuffgffread(input,output)` reads the `input` GFF or GTF file and writes the mandatory columns to the `output` GFF file [1]. The function can also return the GTF-format file using the `'GTFOutput'` option.

`cuffgffread` requires the Cufflinks Support Package for the Bioinformatics Toolbox. If the support package is not installed, then the function provides a download link. For details, see "Bioinformatics Toolbox Software Support Packages".

`cuffgffread(input,output,opt)` uses the additional options specified by `opt`.

`cuffgffread(input,output,Name,Value)` uses additional options specified by one or more name-value pair arguments. For example, `cuffgffread('gyrAB.gtf','gyrAB.gff','PreserveAttributes',true)` retains all attributes in the output file.

Examples

Convert GTF to GFF Format

Convert a GTF file to a GFF file while retaining all attributes.

```
cuffgffread('gyrAB.gtf','gyrABOut.gff','PreserveAttributes',true)
```

You can also set the options using an object. For instance, specify the output to be in the GTF format.

```
opt = CuffGFFReadOptions;
opt.GTFOutput = true;
opt.PreserveAttributes = true;
cuffgffread('gyrAB.gtf','gyrABOut.gtf',opt);
```

Once you have the options object, you can retrieve the equivalent original options for all object properties using `getOptionsTable`.

```
getOptionsTable(opt)
```

```
ans =

  33×3 table
```

```
                                    PropertyName                FlagName          FlagShortNam
                             ____________________________    ________________    _____________

    AppendDescription        'AppendDescription'             '-A'                    ''
    CheckOppositeStrand      'CheckOppositeStrand'           '-B'                    ''
    CheckPhase               'CheckPhase'                    '-H'                    ''
    Cluster                  'Cluster'                       '--cluster-only'        ''
    CodingOnly               'CodingOnly'                    '-C'                    ''
    CollapseContainer        'CollapseContainer'             '-K'                    ''
    CollapseFull             'CollapseFull'                  '-Q'                    ''
    CoordinateRange          'CoordinateRange'               '-r'                    ''
    DiscardInvalidCDS        'DiscardInvalidCDS'             '-J'                    ''
    DiscardNonCanonicalSplice    'DiscardNonCanonicalSplice'     '-N'                    ''
    DiscardSingleExon        'DiscardSingleExon'             '-U'                    ''
    DiscardTerminatedCDS     'DiscardTerminatedCDS'          '-V'                    ''
    FastaCDSFile             'FastaCDSFile'                  '-x'                    ''
    FastaExonsFile           'FastaExonsFile'                '-w'                    ''
    FastaProteinFile         'FastaProteinFile'              '-y'                    ''
    FirstExonOnly            'FirstExonOnly'                 '-G'                    ''
    ForceExons               'ForceExons'                    '--force-exons'         ''
    FullyContained           'FullyContained'                '-R'                    ''
    GTFOutput                'GTFOutput'                     '-T'                    ''
    MaxIntronLength          'MaxIntronLength'               '-i'                    ''
    Merge                    'Merge'                         '--merge'               '-M'
    MergeCloseExons          'MergeCloseExons'               '-Z'                    ''
    MergeInfoFile            'MergeInfoFile'                 '-d'                    ''
    PreserveAttributes       'PreserveAttributes'            '-F'                    ''
    Pseudo                   'Pseudo'                        '--no-pseudo'           ''
    ReplacementTable         'ReplacementTable'              '-m'                    ''
    SequenceFile             'SequenceFile'                  '-g'                    ''
    SequenceInfo             'SequenceInfo'                  '-s'                    ''
    UrlDecode                'UrlDecode'                     '-D'                    ''
    UseEnsemblConversion     'UseEnsemblConversion'          '-L'                    ''
    UseNonTranscript         'UseNonTranscript'              '-O'                    ''
    UseTrackName             'UseTrackName'                  '-t'                    ''
    WriteCoordinates         'WriteCoordinates'              '-W'                    ''
```

Input Arguments

`input` — Input file name

string | character vector

Input file name, specified as a string or character vector. The file can be a GTF or GFF file.

Example: `'gyrAB.gtf'`

Data Types: `char` | `string`

`output` — Output file name

string | character vector

Output file name, specified as a string or character vector. By default, the output is a GFF file. Set `'GTFOutput'` to `true` to get a GTF output file.

Example: `'gyrAB.gff'`

Data Types: `char` | `string`

opt — `cuffgffread` options
`CuffGFFReadOptions` object | string | character vector

`cuffgffread` options, specified as a `CuffGFFReadOptions` object, string, or character vector. The string or character vector must be in the original `gffread` option syntax (prefixed by one or two dashes) [1].

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example:
`cuffgffread('gyrAB.gtf','gyrAB.gff','CoordinateRange','+NC_000912.1:4821..7340')`

`AppendDescription` — Flag to add file descriptions to `descr` attribute
`false` (default) | `true`

Flag to add file descriptions from sequence files to the `descr` attribute of the output GFF record, specified as `true` or `false`. Specify the sequence files using the `SequenceInfo` option.

Example: `'AppendDescription',true`

Data Types: `logical`

`CheckOppositeStrand` — Flag to check opposite strand when checking for in-frame stop codons
`false` (default) | `true`

Flag to check opposite strand when checking for in-frame stop codons, specified as `true` or `false`.

Example: `'CheckOppositeStrand',true`

Data Types: `logical`

`CheckPhase` — Flag to adjust coding sequence phase
`false` (default) | `true`

Flag to adjust coding sequence phase when checking for in-frame stop codons, specified as `true` or `false`.

Example: `'CheckPhase',true`

Data Types: `logical`

`Cluster` — Flag to cluster input transcripts into loci
`true` (default) | `false`

Flag to cluster the input transcripts into loci, specified as `true` or `false`. This option is the same as the `Merge` property, except that it does not collapse fully contained transcripts with identical introns.

Example: `'Cluster',false`

Data Types: `logical`

`CodingOnly` — Flag to discard transcripts with no coding sequence
`false` (default) | `true`

Flag to discard transcripts with no coding sequence feature (CDS), specified as `true` or `false`.

Example: `'CodingOnly',true`

Data Types: `logical`

`CollapseContainer` — Flag to collapse fully contained transcripts
`false` (default) | `true`

Flag to collapse fully contained transcripts that are shorter with fewer introns than the container, specified as `true` or `false`. This property applies only when you set `Merge` to `true`.

Example: `'CollapseContainer',true`

Data Types: `logical`

`CollapseFull` — Flag to collapse shorter transcripts overlapping at least 80% with another exon
`false` (default) | `true`

Flag to collapse shorter transcripts overlapping at least 80% with another single exon transcript, specified as `true` or `false`. This property applies only when you set `Merge` to `true`.

Example: `'CollapseFull',true`

Data Types: `logical`

`CoordinateRange` — Genomic range to filter transcripts
string | character vector

Genomic range to filter transcripts, specified as a string or character vector. The format must be `"[[<strand>]<chr>:]<start>..<end>"`, where `start` and `end` are genomic positions, `chr` is an optional chromosome or contig name, and an optional `strand` (`'+'` or `'-'`).

Example: `'CoordinateRange',“+NC_000912.1:4821..7340”`

Data Types: `char` | `string`

`DiscardInvalidCDS` — Flag to ignore mRNA transcripts either lacking start or stop codon or having in-frame stop codon
`false` (default) | `true`

Flag to ignore mRNA transcripts either lacking a start or stop codon or having an in-frame stop codon, specified as `true` or `false`.

Example: `'DiscardInvalidCDS',true`

Data Types: `logical`

`DiscardNonCanonicalSplice` — Flag to ignore multiexon mRNA transcripts that have intron with noncanonical splice sequence
`false` (default) | `true`

Flag to ignore multiexon mRNA transcripts that have an intron with a noncanonical splice sequence, specified as `true` or `false`. A noncanonical splice sequence is any splice sequence other than `"GT-AG"`, `"CG-AG"`, or `"AT-AC"`.

Example: `'DiscardNonCanonicalSplice',true`

Data Types: `logical`

`DiscardSingleExon` — Flag to ignore transcripts spanning single exon
`false` (default) | `true`

Flag to ignore transcripts spanning a single exon, specified as `true` or `false`.

Example: `'DiscardSingleExon',true`

Data Types: `logical`

`DiscardTerminatedCDS` — Flag to ignore transcripts with in-frame stop codon
`false` (default) | `true`

Flag to ignore transcripts with an in-frame stop codon, specified as `true` or `false`.

Example: `'DiscardTerminatedCDS',true`

Data Types: `logical`

`ExtraCommand` — Additional commands
`""` (default) | character vector | string

The commands must be in the native syntax (prefixed by one or two dashes). Use this option to apply undocumented flags and flags without corresponding MATLAB properties.

Example: `'ExtraCommand',"-E"`

Data Types: `char` | `string`

`FastaCDSFile` — Name of file to save spliced coding sequences
string | character vector

Name of a file to save the spliced coding sequences in the FASTA format, specified as a string or character vector.

Example: `'FastaCDSFile',"splicedCoding.FASTA"`

Data Types: `char` | `string`

`FastaExonsFile` — Name of file to save spliced exons
string | character vector

Name of a file to save the spliced exons in the FASTA format, specified as a string or character vector.

Example: `'FastaExonsFile',"splicedExon.FASTA"`

Data Types: `char` | `string`

`FastaProteinFile` — Name of file to save protein translation of coding sequences
string | character vector

Name of a file to save the protein translation of coding sequences in the FASTA format, specified as a string or character vector.

Example: `'FastaProteinFile',"translated.FASTA"`

Data Types: `char` | `string`

`FirstExonOnly` — Flag to parse additional attributes only from first exon
`false` (default) | `true`

Flag to parse additional attributes only from the first exon, specified as `true` or `false`.

Example: `'FirstExonOnly',true`

Data Types: `logical`

`ForceExons` — Flag to list lowest-level GFF features as exon features
`false` (default) | `true`

Flag to list the lowest-level GFF features as exon features in the output file, specified as `true` or `false`.

Example: `'ForceExons',true`

Data Types: `logical`

`FullyContained` — Flag to discard transcripts not contained fully
`false` (default) | `true`

Flag to discard transcripts not contained fully within the range, specified as `true` or `false`. Specify the range using the `CoordinateRange` option.

Example: `'FullyContained',true`

Data Types: `logical`

`GTFOutput` — Flag to output GTF-format transcript files
`false` (default) | `true`

Flag to output GTF-format transcript files, specified as `true` or `false`.

Example: `'GTFOutput',true`

Data Types: `logical`

`IncludeAll` — Flag to apply all available options
`false` (default) | `true`

The original (native) syntax is prefixed by one or two dashes. By default, the function converts only the specified options. If the value is `true`, the software converts all available options, with default values for unspecified options, to the original syntax.

Note If you set `IncludeAll` to `true`, the software translates all available properties, with default values for unspecified properties. The only exception is that when the default value of a property is `NaN`, `Inf`, `[]`, `''`, or `""`, then the software does not translate the corresponding property.

Example: `'IncludeAll',true`

Data Types: `logical`

`MaxIntronLength` — Maximum intron length for transcript to include in output
`Inf` (default) | positive integer

Maximum intron length for a transcript to include in the output file, specified as a positive integer. `Inf`, the default value, sets no limit on the intron length.

Example: `'MaxIntronLength',500`

Data Types: `double`

`Merge` — Flag to merge transcripts to loci
`false` (default) | `true`

Flag to merge transcripts into loci by collapsing transcripts with identical introns, specified as `true` or `false`.

Example: `'Merge',true`

Data Types: `logical`

`MergeCloseExons` — Flag to merge exons into single exon
`false` (default) | `true`

Flag to merge exons into a single exon when separated by fewer than 4 base-pair introns, specified as `true` or `false`.

Example: `'MergeCloseExons',true`

Data Types: `logical`

`MergeInfoFile` — Name of file to save information on duplicates when merging
string | character vector

Name of a file to save information on duplicates when merging, specified as a string or character vector. This property applies only when you set `Merge` to `true`.

Example: `'MergeInfoFile',"duplicates.txt"`

Data Types: `char` | `string`

`PreserveAttributes` — Flag to retain all attributes in output
`false` (default) | `true`

Flag to retain all attributes in the output file, specified as `true` or `false`.

Example: `'PreserveAttributes',true`

Data Types: `logical`

`Pseudo` — Flag to filter out records containing "pseudo"
`true` (default) | `false`

Flag to filter out records containing the word "pseudo," specified as `true` or `false`.

Example: `'Pseudo',false`

Data Types: `logical`

`ReplacementTable` — Name of file containing replacement table
string | character vector

Name of a file containing a replacement table, specified as a string or character vector. The table must have two columns, where the first column contains the original transcript IDs and the second column contains the new transcript IDs. An example table follows.

origTranscript1	newTranscript1
origTranscript2	newTranscript2
origTranscript3	newTranscript3

If you provide a replacement table, the function replaces the transcript IDs found in the first column with the new transcripts IDs from the second column and filters out those transcripts not found.

Example: `'ReplacementTable',"replaceTbl.txt"`

Data Types: `char` | `string`

`SequenceFile` — Name of FASTA-format file containing genomic sequences
string | character vector

Name of a FASTA-format file containing genomic sequences for all input mappings, specified as a string or character vector.

Example: `'SequenceFile',"seqs.fasta"`

Data Types: `char` | `string`

`SequenceInfo` — Name of tab-delimited file with additional information on input sequence
string | character vector

Name of a tab-delimited file with additional information on each input sequence, specified as a string or character vector. This file must have three columns: a sequence name column, a sequence length column, and a sequence description column. If `AppendDescription` is `true`, the sequence description is included as an attribute in the output GFF file.

Example: `'SequenceInfo',"seqinfo.txt"`

Data Types: `char` | `string`

`UrlDecode` — Flag to decode URL-encoded characters in attribute names
`false` (default) | `true`

Flag to decode url-encoded characters in attribute names, specified as `true` or `false`. For instance, "transcript%20description" is decoded to "transcript description".

Example: `'UrlDecode',true`

Data Types: `logical`

`UseEnsemblConversion` — Flag to use GTF-to-GFF3 conversion method from Ensembl
`false` (default) | `true`

Flag to use the GTF-to-GFF3 conversion method from Ensembl, specified as `true` or `false`.

Example: `'UseEnsemblConversion',true`

Data Types: `logical`

`UseNonTranscript` — Flag to include nontranscript GFF records in output file
`false` (default) | `true`

Flag to include nontranscript GFF records in the output file, specified as `true` or `false`.

Example: `'UseNonTranscript',true`

Data Types: `logical`

`UseTrackName` — Flag to use track name in second column of GFF output line
`false` (default) | `true`

Flag to use the track name in the second column of the GFF output line, specified as `true` or `false`.

Example: `'UseTrackName',true`

Data Types: `logical`

`WriteCoordinates` — Flag to write exon coordinates projected onto spliced sequence
`false` (default) | `true`

Flag to write the exon coordinates projected onto the spliced sequence, specified as `true` or `false`. This property applies only when `FastaExonsFile` or `FastaCDSFile` is specified.

Example: `'WriteCoordinates',true`

Data Types: `logical`

Version History

Introduced in R2019a

References

[1] Trapnell, Cole, Brian A Williams, Geo Pertea, Ali Mortazavi, Gordon Kwan, Marijke J van Baren, Steven L Salzberg, Barbara J Wold, and Lior Pachter. “Transcript Assembly and Quantification by RNA-Seq Reveals Unannotated Transcripts and Isoform Switching during Cell Differentiation.” *Nature Biotechnology* 28, no. 5 (May 2010): 511–15.

See Also

`cufflinks` | `CuffGFFReadOptions`

Topics
“Bioinformatics Toolbox Software Support Packages”

External Websites
Cufflinks manual

CuffGFFReadOptions

Option set for `cuffgffread`

Description

A `CuffGFFReadOptions` object contains options for the `cuffgffread` function, which filters and converts GFF and GTF files [1].

Creation

Syntax

```
cuffgffreadOpt = CuffGFFReadOptions
cuffgffreadOpt = CuffGFFReadOptions(Name,Value)
cuffgffreadOpt = CuffGFFReadOptions(S)
```

Description

`cuffgffreadOpt = CuffGFFReadOptions` creates a `CuffGFFReadOptions` object with the default property values.

`CuffGFFReadOptions` requires the Cufflinks Support Package for the Bioinformatics Toolbox. If the support package is not installed, then the function provides a download link. For details, see "Bioinformatics Toolbox Software Support Packages".

`cuffgffreadOpt = CuffGFFReadOptions(Name,Value)` sets the object properties on page 1-655 using one or more name-value pair arguments. Enclose each property name in quotes. For example, `cuffgffreadOpt = CuffGFFReadOptions('DiscardSingleExon',true)` discards transcripts spanning a single exon.

`cuffgffreadOpt = CuffGFFReadOptions(S)` specifies optional parameters using the string or character vector `S`.

Input Arguments

S — `cuffgffread` options
string | character vector

`cuffgffread` options, specified as a string or character vector. `S` must be in the original `gffread` option syntax (prefixed by one or two dashes).

Example: `'-U'`

Properties

`AppendDescription` — Flag to add file descriptions to `descr` attribute
`false` (default) | `true`

Flag to add file descriptions from sequence files to the `descr` attribute of the output GFF record, specified as `true` or `false`. Specify the sequence files using the `SequenceInfo` option.

Example: `true`

Data Types: `logical`

`CheckOppositeStrand` — Flag to check opposite strand when checking for in-frame stop codons
`false` (default) | `true`

Flag to check opposite strand when checking for in-frame stop codons, specified as `true` or `false`.

Example: `true`

Data Types: `logical`

`CheckPhase` — Flag to adjust coding sequence phase
`false` (default) | `true`

Flag to adjust coding sequence phase when checking for in-frame stop codons, specified as `true` or `false`.

Example: `true`

Data Types: `logical`

`Cluster` — Flag to cluster input transcripts into loci
`true` (default) | `false`

Flag to cluster the input transcripts into loci, specified as `true` or `false`. This option is the same as the `Merge` property, except that it does not collapse fully contained transcripts with identical introns.

Example: `false`

Data Types: `logical`

`CodingOnly` — Flag to discard transcripts with no coding sequence
`false` (default) | `true`

Flag to discard transcripts with no coding sequence feature (CDS), specified as `true` or `false`.

Example: `true`

Data Types: `logical`

`CollapseContainer` — Flag to collapse fully-contained transcripts
`false` (default) | `true`

Flag to collapse fully contained transcripts that are shorter with fewer introns than the container, specified as `true` or `false`. This property applies only when you set `Merge` to `true`.

Example: `true`

Data Types: `logical`

`CollapseFull` — Flag to collapse shorter transcripts overlapping at least 80% with another exon
`false` (default) | `true`

Flag to collapse shorter transcripts overlapping at least 80% with another single exon transcript, specified as `true` or `false`. This property applies only when you set `Merge` to `true`.

Example: `true`

Data Types: `logical`

`CoordinateRange` — Genomic range to filter transcripts
string | character vector

Genomic range to filter transcripts, specified as a string or character vector. The format must be `"[[<strand>]<chr>:]<start>..<end>"`, where `start` and `end` are genomic positions, `chr` is an optional chromosome or contig name, and an optional `strand` (`'+'` or `'-'`).

Example: “`+NC_000912.1:4821..7340`”

Data Types: `char` | `string`

`DiscardInvalidCDS` — Flag to ignore mRNA transcripts either lacking start or stop codon or having in-frame stop codon
`false` (default) | `true`

Flag to ignore mRNA transcripts either lacking a start or stop codon or having an in-frame stop codon, specified as `true` or `false`.

Example: `true`

Data Types: `logical`

`DiscardNonCanonicalSplice` — Flag to ignore multiexon mRNA transcripts that have intron with noncanonical splice sequence
`false` (default) | `true`

Flag to ignore multiexon mRNA transcripts that have an intron with a noncanonical splice sequence, specified as `true` or `false`. A noncanonical splice sequence is any splice sequence other than `"GT-AG"`, `"CG-AG"`, or `"AT-AC"`.

Example: `true`

Data Types: `logical`

`DiscardSingleExon` — Flag to ignore transcripts spanning single exon
`false` (default) | `true`

Flag to ignore transcripts spanning a single exon, specified as `true` or `false`.

Example: `true`

Data Types: `logical`

`DiscardTerminatedCDS` — Flag to ignore transcripts with in-frame stop codon
`false` (default) | `true`

Flag to ignore transcripts with an in-frame stop codon, specified as `true` or `false`.

Example: `true`

Data Types: `logical`

`ExtraCommand` — Additional commands
`""` (default) | character vector | string

The commands must be in the native syntax (prefixed by one or two dashes). Use this option to apply undocumented flags and flags without corresponding MATLAB properties.

When the software converts the original flags to MATLAB properties, it stores any unrecognized flags in this property.

Example: `"-E"`

Data Types: `char` | `string`

`FastaCDSFile` — Name of file to save spliced coding sequences
string | character vector

Name of a file to save the spliced coding sequences in the FASTA format, specified as a string or character vector.

Example: `"splicedCoding.FASTA"`

Data Types: `char` | `string`

`FastaExonsFile` — Name of file to save spliced exons
string | character vector

Name of a file to save the spliced exons in the FASTA format, specified as a string or character vector.

Example: `"splicedExon.FASTA"`

Data Types: `char` | `string`

`FastaProteinFile` — Name of file to save protein translation of coding sequences
string | character vector

Name of a file to save the protein translation of coding sequences in the FASTA format, specified as a string or character vector.

Example: `"translated.FASTA"`

Data Types: `char` | `string`

`FirstExonOnly` — Flag to parse additional attributes only from first exon
`false` (default) | `true`

Flag to parse additional attributes only from the first exon, specified as `true` or `false`.

Example: `true`

Data Types: `logical`

`ForceExons` — Flag to list lowest-level GFF features as exon features
`false` (default) | `true`

Flag to list the lowest-level GFF features as exon features in the output file, specified as `true` or `false`.

Example: `true`

Data Types: `logical`

`FullyContained` — Flag to discard transcripts not contained fully
`false` (default) | `true`

Flag to discard transcripts not contained fully within the range, specified as `true` or `false`. Specify the range using the `CoordinateRange` option.

Example: `true`

Data Types: `logical`

`GTFOutput` — Flag to output GTF-format transcript files
`false` (default) | `true`

Flag to output GTF-format transcript files, specified as `true` or `false`.

Example: `true`

Data Types: `logical`

`IncludeAll` — Flag to use all object properties
`false` (default) | `true`

Flag to include all the object properties with the corresponding default values when converting to the original options syntax, specified as `true` or `false`. You can convert the properties to the original syntax prefixed by one or two dashes (such as `'-d 100 -e 80'`) by using `getCommand`. The default value `false` means that when you call `getCommand(optionsObject)`, it converts only the specified properties. If the value is `true`, `getCommand` converts all available properties, with default values for unspecified properties, to the original syntax.

Note If you set `IncludeAll` to `true`, the software translates all available properties, with default values for unspecified properties. The only exception is that when the default value of a property is `NaN`, `Inf`, `[]`, `''`, or `""`, then the software does not translate the corresponding property.

Example: `true`

Data Types: `logical`

`MaxIntronLength` — Maximum intron length for transcript to be included in output
`Inf` (default) | positive integer

Maximum intron length for a transcript to include in the output file, specified as a positive integer. `Inf`, the default value, sets no limit on the intron length.

Example: `500`

Data Types: `double`

`Merge` — Flag to merge transcripts to loci
`false` (default) | `true`

Flag to merge transcripts into loci by collapsing transcripts with identical introns, specified as `true` or `false`.

Example: `true`

Data Types: `logical`

`MergeCloseExons` — Flag to merge exons into single exon
`false` (default) | `true`

Flag to merge exons into a single exon when separated by fewer than 4 base-pair introns, specified as `true` or `false`.

Example: `true`

Data Types: `logical`

`MergeInfoFile` — Name of file to save information on duplicates when merging
string | character vector

Name of a file to save information on duplicates when merging, specified as a string or character vector. This property applies only when you set `Merge` to `true`.

Example: `"duplicates.txt"`

Data Types: `char` | `string`

`PreserveAttributes` — Flag to retain all attributes in output
`false` (default) | `true`

Flag to retain all attributes in the output file, specified as `true` or `false`.

Example: `true`

Data Types: `logical`

`Pseudo` — Flag to filter out records containing "pseudo"
`true` (default) | `false`

Flag to filter out records containing the word "pseudo," specified as `true` or `false`.

Example: `false`

Data Types: `logical`

`ReplacementTable` — Name of file containing replacement table
string | character vector

Name of a file containing a replacement table, specified as a string or character vector. The table must have two columns, where the first column contains the original transcript IDs and the second column contains the new transcript IDs. An example table follows.

origTranscript1	newTranscript1
origTranscript2	newTranscript2
origTranscript3	newTranscript3

If you provide a replacement table, the function replaces the transcript IDs found in the first column with the new transcripts IDs from the second column and filters out those transcripts not found.

Example: `"replaceTbl.txt"`

Data Types: `char` | `string`

`SequenceFile` — Name of FASTA-format file containing genomic sequences
string | character vector

Name of a FASTA-format file containing genomic sequences for all input mappings, specified as a string or character vector.

Example: "seqs.fasta"

Data Types: char | string

SequenceInfo — Name of tab-delimited file with additional information on input sequence
string | character vector

Name of a tab-delimited file with additional information on each input sequence, specified as a string or character vector. This file must have three columns: a sequence name column, a sequence length column, and a sequence description column. If AppendDescription is true, the sequence description is included as an attribute in the output GFF file.

Example: "seqinfo.txt"

Data Types: char | string

UrlDecode — Flag to decode URL-encoded characters in attribute names
false (default) | true

Flag to decode url-encoded characters in attribute names, specified as true or false. For instance, "transcript%20description" is decoded to "transcript description".

Example: true

Data Types: logical

UseEnsemblConversion — Flag to use GTF-to-GFF3 conversion method from Ensembl
false (default) | true

Flag to use the GTF-to-GFF3 conversion method from Ensembl, specified as true or false.

Example: true

Data Types: logical

UseNonTranscript — Flag to include nontranscript GFF records in output file
false (default) | true

Flag to include nontranscript GFF records in the output file, specified as true or false.

Example: true

Data Types: logical

UseTrackName — Flag to use track name in second column of GFF output line
false (default) | true

Flag to use the track name in the second column of the GFF output line, specified as true or false.

Example: true

Data Types: logical

Version — Supported version
string

This property is read-only.

Supported version of the original cufflinks software, returned as a string.

Example: "2.2.1"

Data Types: `string`

`WriteCoordinates` — Flag to write exon coordinates projected onto spliced sequence
`false` (default) | `true`

Flag to write the exon coordinates projected onto the spliced sequence, specified as `true` or `false`. This property applies only when `FastaExonsFile` or `FastaCDSFile` is specified.

Example: `true`

Data Types: `logical`

Object Functions

getCommand	Translate object properties to original options syntax
getOptionsTable	Return table with all properties and equivalent options in original syntax

Examples

Create CuffGFFReadOptions Object

Create a `CuffGFFReadOptions` object with the default values.

```
opt = CuffGFFReadOptions;
```

Create an object using name-value pairs.

```
opt2 = CuffGFFReadOptions('DiscardSingleExon',true,'FastaExonsFile','exons.fa');
```

Create an object by using the original syntax.

```
opt3 = CuffGFFReadOptions('-U -w exons.fa')
```

Convert GTF to GFF Format

Convert a GTF file to a GFF file while retaining all attributes.

```
cuffgffread('gyrAB.gtf','gyrABOut.gff','PreserveAttributes',true)
```

You can also set the options using an object. For instance, specify the output to be in the GTF format.

```
opt = CuffGFFReadOptions;
opt.GTFOutput = true;
opt.PreserveAttributes = true;
cuffgffread('gyrAB.gtf','gyrABOut.gtf',opt);
```

Once you have the options object, you can retrieve the equivalent original options for all object properties using `getOptionsTable`.

```
getOptionsTable(opt)
```

```
ans =

  33×3 table
```

```
                                        PropertyName                 FlagName        FlagShortNam
                                 ___________________________    ________________    ____________

    AppendDescription            'AppendDescription'            '-A'                    ''
    CheckOppositeStrand          'CheckOppositeStrand'          '-B'                    ''
    CheckPhase                   'CheckPhase'                   '-H'                    ''
    Cluster                      'Cluster'                      '--cluster-only'        ''
    CodingOnly                   'CodingOnly'                   '-C'                    ''
    CollapseContainer            'CollapseContainer'            '-K'                    ''
    CollapseFull                 'CollapseFull'                 '-Q'                    ''
    CoordinateRange              'CoordinateRange'              '-r'                    ''
    DiscardInvalidCDS            'DiscardInvalidCDS'            '-J'                    ''
    DiscardNonCanonicalSplice    'DiscardNonCanonicalSplice'    '-N'                    ''
    DiscardSingleExon            'DiscardSingleExon'            '-U'                    ''
    DiscardTerminatedCDS         'DiscardTerminatedCDS'         '-V'                    ''
    FastaCDSFile                 'FastaCDSFile'                 '-x'                    ''
    FastaExonsFile               'FastaExonsFile'               '-w'                    ''
    FastaProteinFile             'FastaProteinFile'             '-y'                    ''
    FirstExonOnly                'FirstExonOnly'                '-G'                    ''
    ForceExons                   'ForceExons'                   '--force-exons'         ''
    FullyContained               'FullyContained'               '-R'                    ''
    GTFOutput                    'GTFOutput'                    '-T'                    ''
    MaxIntronLength              'MaxIntronLength'              '-i'                    ''
    Merge                        'Merge'                        '--merge'               '-M'
    MergeCloseExons              'MergeCloseExons'              '-Z'                    ''
    MergeInfoFile                'MergeInfoFile'                '-d'                    ''
    PreserveAttributes           'PreserveAttributes'           '-F'                    ''
    Pseudo                       'Pseudo'                       '--no-pseudo'           ''
    ReplacementTable             'ReplacementTable'             '-m'                    ''
    SequenceFile                 'SequenceFile'                 '-g'                    ''
    SequenceInfo                 'SequenceInfo'                 '-s'                    ''
    UrlDecode                    'UrlDecode'                    '-D'                    ''
    UseEnsemblConversion         'UseEnsemblConversion'         '-L'                    ''
    UseNonTranscript             'UseNonTranscript'             '-O'                    ''
    UseTrackName                 'UseTrackName'                 '-t'                    ''
    WriteCoordinates             'WriteCoordinates'             '-W'                    ''
```

Version History

Introduced in R2019a

References

[1] Trapnell, Cole, Brian A Williams, Geo Pertea, Ali Mortazavi, Gordon Kwan, Marijke J van Baren, Steven L Salzberg, Barbara J Wold, and Lior Pachter. "Transcript Assembly and Quantification by RNA-Seq Reveals Unannotated Transcripts and Isoform Switching during Cell Differentiation." *Nature Biotechnology* 28, no. 5 (May 2010): 511–15.

See Also

`cuffcompare` | `cufflinks`

Topics

"Bioinformatics Toolbox Software Support Packages"

External Websites
Cufflinks manual

cuffgtf2sam

Convert GTF files to SAM files

Syntax

```
cuffgtf2sam(input,output)
cuffgtf2sam(input,output,Name,Value)
```

Description

`cuffgtf2sam(input,output)` converts the assembled transcripts in the GTF file `input` to the SAM-format file `output` [1].

`cuffgtf2sam` requires the Cufflinks Support Package for the Bioinformatics Toolbox. If the support package is not installed, then the function provides a download link. For details, see “Bioinformatics Toolbox Software Support Packages”.

`cuffgtf2sam(input,output,Name,Value)` uses additional options specified by one or more name-value pair arguments. For example, `gtf2sam('hum37_2_1M.gtf','hum37_2_1M.sam','UseFPKM',true)` inserts the FPKM value into the SAM records.

Examples

Convert GTF to SAM

Convert a GTF file to a SAM file.

```
cuffgtf2sam('hum37_2_1M.gtf','hum37_2_1M.sam')
```

Input Arguments

`input` — Names of input files
string | character vector | string vector | cell array of character vectors

Names of input files, specified as a string, character vector, string vector, or cell array of character vectors.

Example: `'gyrAB.gtf'`

Data Types: `cell` | `char` | `string`

`output` — Output SAM file name
string | character vector

Output SAM file name, specified as a string or character vector.

Example: `'gyrAB.sam'`

Data Types: `char` | `string`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `gtf2sam('hum37_2_1M.gtf','hum37_2_1M.sam','UseFPKM',true)`

`ReferenceFASTA` — Name of reference FASTA file
string | character vector

Name of a reference FASTA file, specified as a string or character vector. If you specify a FASTA file, the function recreates the sequences of transcripts by comparing to the reference sequences in the provided FASTA file. If you do not specify `'ReferenceFASTA'`, the function omits the sequence information from the output SAM file.

Example: `'ReferenceFASTA',"ref.fasta"`

Data Types: `char` | `string`

`UseFPKM` — Flag to insert FPKM value
`false` (default) | `true`

Flag to insert the FPKM value into the SAM records instead of the isoform fraction, specified as `true` or `false`.

Example: `'UseFPKM',true`

Data Types: `logical`

Version History
Introduced in R2019a

References

[1] Trapnell, Cole, Brian A Williams, Geo Pertea, Ali Mortazavi, Gordon Kwan, Marijke J van Baren, Steven L Salzberg, Barbara J Wold, and Lior Pachter. “Transcript Assembly and Quantification by RNA-Seq Reveals Unannotated Transcripts and Isoform Switching during Cell Differentiation.” *Nature Biotechnology* 28, no. 5 (May 2010): 511–15.

[2] Li, H., B. Handsaker, A. Wysoker, T. Fennell, J. Ruan, N. Homer, G. Marth, G. Abecasis, R. Durbin, and 1000 Genome Project Data Processing Subgroup. “The Sequence Alignment/Map Format and SAMtools.” Bioinformatics 25, no. 16 (August 15, 2009): 2078–79.

See Also
`cufflinks` | `cuffgffread` | `samread` | `GTFAnnotation` | `GFFAnnotation`

Topics
“Bioinformatics Toolbox Software Support Packages”

External Websites

Cufflinks manual

cufflinks

Assemble transcriptome from aligned reads

Syntax

```
cufflinks(alignmentFiles)
cufflinks(alignmentFiles,cufflinksOptions)
cufflinks(alignmentFiles,Name,Value)
[transcripts,isoforms,genes,skippedTranscripts] = cufflinks( ___ )
```

Description

`cufflinks(alignmentFiles)` assembles a transcriptome from aligned reads in `alignmentFile` and quantifies the level of expression for each transcript [1]. By default, the function writes the results to a GTF file named `transcripts.gtf` in the current directory.

`cufflinks` requires the Cufflinks Support Package for the Bioinformatics Toolbox. If the support package is not installed, then the function provides a download link. For details, see “Bioinformatics Toolbox Software Support Packages”.

`cufflinks(alignmentFiles,cufflinksOptions)` uses additional options specified by `cufflinksOptions`.

`cufflinks(alignmentFiles,Name,Value)` uses additional options specified by one or more name-value pair arguments. For example, `cufflinks(alignmentFile,'TrimCoverageThreshold',5)` specifies the minimum average coverage for 3' end trimming.

`[transcripts,isoforms,genes,skippedTranscripts] = cufflinks( ___ )` returns the file names of the assembled transcriptome using any of the input argument combinations from the previous syntaxes. By default, the function saves all files to the current directory.

Examples

Assemble Transcriptome and Perform Differential Expression Testing

Create a `CufflinksOptions` object to define cufflinks options, such as the number of parallel threads and the output directory to store the results.

```
cflOpt = CufflinksOptions;
cflOpt.NumThreads = 8;
cflOpt.OutputDirectory = "./cufflinksOut";
```

The SAM files provided for this example contain aligned reads for *Mycoplasma pneumoniae* from two samples with three replicates each. The reads are simulated 100bp-reads for two genes (`gyrA` and `gyrB`) located next to each other on the genome. All the reads are sorted by reference position, as required by `cufflinks`.

```
sams = ["Myco_1_1.sam","Myco_1_2.sam","Myco_1_3.sam",...
        "Myco_2_1.sam", "Myco_2_2.sam", "Myco_2_3.sam"];
```

Assemble the transcriptome from the aligned reads.

```
[gtfs,isofpkm,genes,skipped] = cufflinks(sams,cflOpt);
```

`gtfs` is a list of GTF files that contain assembled isoforms.

Compare the assembled isoforms using `cuffcompare`.

```
stats = cuffcompare(gtfs);
```

Merge the assembled transcripts using `cuffmerge`.

```
mergedGTF = cuffmerge(gtfs,'OutputDirectory','./cuffMergeOutput');
```

`mergedGTF` reports only one transcript. This is because the two genes of interest are located next to each other, and `cuffmerge` cannot distinguish two distinct genes. To guide `cuffmerge`, use a reference GTF (`gyrAB.gtf`) containing information about these two genes. If the file is not located in the same directory that you run `cuffmerge` from, you must also specify the file path.

```
gyrAB = which('gyrAB.gtf');
mergedGTF2 = cuffmerge(gtfs,'OutputDirectory','./cuffMergeOutput2',...
            'ReferenceGTF',gyrAB);
```

Calculate abundances (expression levels) from aligned reads for each sample.

```
abundances1 = cuffquant(mergedGTF2,["Myco_1_1.sam","Myco_1_2.sam","Myco_1_3.sam"],...
                        'OutputDirectory','./cuffquantOutput1');
abundances2 = cuffquant(mergedGTF2,["Myco_2_1.sam", "Myco_2_2.sam", "Myco_2_3.sam"],...
                        'OutputDirectory','./cuffquantOutput2');
```

Assess the significance of changes in expression for genes and transcripts between conditions by performing the differential testing using `cuffdiff`. The `cuffdiff` function operates in two distinct steps: the function first estimates abundances from aligned reads, and then performs the statistical analysis. In some cases (for example, distributing computing load across multiple workers), performing the two steps separately is desirable. After performing the first step with `cuffquant`, you can then use the binary CXB output file as an input to `cuffdiff` to perform statistical analysis. Because `cuffdiff` returns several files, specify the output directory is recommended.

```
isoformDiff = cuffdiff(mergedGTF2,[abundances1,abundances2],...
                      'OutputDirectory','./cuffdiffOutput');
```

Display a table containing the differential expression test results for the two genes `gyrB` and `gyrA`.

```
readtable(isoformDiff,'FileType','text')
```

```
ans =

  2×14 table

        test_id            gene_id         gene             locus               sample_1    sample_
    ________________    _____________    ______    ________________________    ________    _______

    'TCONS_00000001'    'XLOC_000001'    'gyrB'    'NC_000912.1:2868-7340'       'q1'         'q2'
    'TCONS_00000002'    'XLOC_000001'    'gyrA'    'NC_000912.1:2868-7340'       'q1'         'q2'
```

You can use `cuffnorm` to generate normalized expression tables for further analyses. `cuffnorm` results are useful when you have many samples and you want to cluster them or plot expression

levels for genes that are important in your study. Note that you cannot perform differential expression analysis using `cuffnorm`.

Specify a cell array, where each element is a string vector containing file names for a single sample with replicates.

```
alignmentFiles = {["Myco_1_1.sam","Myco_1_2.sam","Myco_1_3.sam"],...
                  ["Myco_2_1.sam", "Myco_2_2.sam", "Myco_2_3.sam"]}
isoformNorm = cuffnorm(mergedGTF2, alignmentFiles,...
                      'OutputDirectory', './cuffnormOutput');
```

Display a table containing the normalized expression levels for each transcript.

```
readtable(isoformNorm,'FileType','text')
```

```
ans =

  2×7 table

      tracking_id          q1_0          q1_2          q1_1          q2_1          q2_0
    ________________    __________    __________    __________    __________    __________    __

    'TCONS_00000001'    1.0913e+05         78628    1.2132e+05    4.3639e+05    4.2228e+05    4.2
    'TCONS_00000002'    3.5158e+05    3.7458e+05    3.4238e+05    1.0483e+05    1.1546e+05    1.1
```

Column names starting with *q* have the format: *conditionX_N*, indicating that the column contains values for replicate *N* of *conditionX*.

Input Arguments

`alignmentFiles` — Names of SAM or BAM files

string | string vector | character vector | cell array of character vectors

Names of SAM or BAM files, specified as a string, string vector, character vector, or cell array of character vectors. The input files must be sorted by reference position.

Example: `'Myco_1_1.sam'`

Data Types: `char` | `string`

`cufflinksOptions` — Cufflinks options

`CufflinksOptions` object | character vector | string

Cufflinks options, specified as a `CufflinksOptions` object, character vector, or string. The character vector or string must be in the cufflinks option syntax (prefixed by one or two dashes) [1].

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example:
`cufflinks(alignmentFile,'TrimCoverageThreshold',5,'FragmentLengthMean',180)`

`EffectiveLengthCorrection` — Flag to normalize fragment counts
`true` (default) | `false`

Flag to normalize fragment counts to fragments per kilobase per million mapped reads (FPKM), specified as `true` or `false`.

Example: `'EffectiveLengthCorrection',false`

Data Types: `logical`

`ExtraCommand` — Additional commands
`""` (default) | string | character vector

Additional commands, specified as a string or character vector.

The commands must be in the native syntax (prefixed by one or two dashes). Use this option to apply undocumented flags and flags without corresponding MATLAB properties.

Example: `'ExtraCommand','--library-type fr-secondstrand'`

Data Types: `char` | `string`

`FauxReadTiling` — Flag to include reference transcripts in assembled output
`true` (default) | `false`

Flag to include reference transcripts in the assembled output as faux-reads during RABT (advanced reference annotation based transcript) assembly, specified as `true` or `false`.

Note The function only performs the RABT assembly if you specify `GTFGuide`. Otherwise, `FauxReadTiling`, regardless of being `true` or `false`, has no effect on the assembled transcript.

Example: `'FauxReadTiling',false`

Data Types: `logical`

`FragmentBiasCorrection` — Name of FASTA file with reference transcripts to detect bias
string | character vector

Name of the FASTA file with reference transcripts to detect bias in fragment counts, specified as a string or character vector. Library preparation can introduce sequence-specific bias into RNA-Seq experiments. Providing reference transcripts improves the accuracy of the transcript abundance estimates.

Example: `'FragmentBiasCorrection','ref.fasta'`

Data Types: `char` | `string`

`FragmentLengthMean` — Expected mean fragment length
`200` (default) | positive integer

Expected mean fragment length, specified as a positive integer. The default value is `200` base pairs. The function can learn the fragment length mean for each SAM file. Using this option is not recommended for paired-end reads.

Example: `'FragmentLengthMean',100`

Data Types: `double`

`FragmentLengthSD` — Expected standard deviation for fragment length distribution
`80` (default) | positive scalar

Expected standard deviation for the fragment length distribution, specified as a positive scalar. The default value is `80` base pairs. The function can learn the fragment length standard deviation for each SAM file. Using this option is not recommended for paired-end reads.

Example: `'FragmentLengthSTD',70`

Data Types: `double`

`GTFGuide` — Name of GTF file to guide RABT assembly
string | character vector

Name of a GTF file to guide the RABT assembly, specified as a string or character vector.

Example: `'GTFGuide','tr.gtf'`

Data Types: `char` | `string`

`IncludeAll` — Flag to apply all available options
`false` (default) | true

Flag to include all available options with the corresponding default values when converting to the original options syntax, specified as `true` or `false`.

The original (native) syntax is prefixed by one or two dashes. By default, the function converts only the specified options. If the value is `true`, the software converts all available options, with default values for unspecified options, to the original syntax.

Note If you set `IncludeAll` to `true`, the software translates all available properties, with default values for unspecified properties. The only exception is that when the default value of a property is `NaN`, `Inf`, `[]`, `''`, or `""`, then the software does not translate the corresponding property.

Example: `'IncludeAll',true`

Data Types: `logical`

`RABTOverhangTolerance` — Number of base pairs allowed to overlap with transcript intron
`8` (default) | positive integer

Number of base pairs from a read allowed to overlap with a transcript intron when determining if a read is mappable to another transcript during the RABT assembly, specified as a positive integer. The default value is `8`.

Note The function only performs the RABT assembly if you specify `GTFGuide`. Otherwise, `RABTOverhangTolerance` has no effect on the assembled transcript.

Example: `'IntronOverhangTolerance',10`

Data Types: `double`

`JunctionAlpha` — Alpha value in binomial test to filter false-positive alignments
`0.001` (default) | scalar between `0` and `1`

Alpha value in the binomial test to filter false-positive alignments, specified as a scalar between `0` and `1`.

Example: `'JunctionAlpha',0.005`

Data Types: `double`

`LengthCorrection` — Flag to correct by transcript length
`true` (default) | `false`

Flag to correct by the transcript length, specified as `true` or `false`. Set this value to `false` only when the fragment count is independent of the feature size, such as for small RNA libraries with no fragmentation and for 3' end sequencing, where all fragments have the same length.

Example: `'LengthCorrection',false`

Data Types: `logical`

`MaskFile` — Name of GTF or GFF file containing transcripts to ignore
string | character vector

Name of the GTF or GFF file containing transcripts to ignore during analysis, specified as a string or character vector. Some examples of transcripts to ignore include annotated rRNA transcripts, mitochondrial transcripts, and other abundant transcripts. Ignoring these transcripts improves the robustness of the abundance estimates.

Example: `'MaskFile','excludes.gtf'`

Data Types: `char` | `string`

`MaxBundleFrags` — Maximum number of fragments to include for each locus before skipping
`500000` (default) | positive integer

Maximum number of fragments to include for each locus before skipping new fragments, specified as a positive integer. Skipped fragments are marked with the status `HIDATA` in the file `skipped.gtf`.

Example: `'MaxBundleFrags',400000`

Data Types: `double`

`MaxBundleLength` — Maximum genomic length in base pairs for bundle
`3500000` (default) | positive integer

Maximum genomic length in base pairs for a bundle, specified as a positive integer.

Example: `'MaxBundleLength',3400000`

Data Types: `double`

`MaxFragAlignments` — Maximum number of aligned reads to include for each fragment
`Inf` (default) | positive integer

Maximum number of aligned reads to include for each fragment before skipping new reads, specified as a positive integer. `Inf`, the default value, sets no limit on the maximum number of aligned reads.

Example: `'MaxFragAlignments',1000`

Data Types: `double`

`MaxIntronLength` — Maximum number of bases in intron
`300000` (default) | positive integer

Maximum number of bases in an intron to report, specified as a positive integer. `cufflinks` also ignores SAM alignments with REF_SKIP CIGAR operations longer than this property value.

Example: `'MaxIntronLength',350000`

Data Types: `double`

`MaxMLEIterations` — Maximum number of iterations for maximum likelihood estimation
`5000` (default) | positive integer

Maximum number of iterations for the maximum likelihood estimation of abundances, specified as a positive integer.

Example: `'MaxMLEIterations',4000`

Data Types: `double`

`MinFragsPerTransfrag` — Minimum number of aligned RNA-Seq fragments to report
`10` (default) | positive integer

Minimum number of aligned RNA-Seq fragments to report on an assembled transfrag, specified as a positive integer.

Example: `'MinFragsPerTransfrag',15`

Data Types: `double`

`MinIntronLength` — Minimum number of base pairs for intron in genome
`50` (default) | positive integer

Minimum number of base pairs for an intron in the genome, specified as a positive integer.

Example: `'MinIntronLength',50`

Data Types: `double`

`MinIsoformFraction` — Cuffoff value to report abundance of isoform
`0.1` (default) | scalar between `0` and `1`

Cuffoff value to report the abundance of a particular isoform as a fraction of the most abundant isoform (major isoform), specified as a scalar between `0` and `1`. The function filters out transcripts with abundances below the specified value because isoforms expressed at low levels often cannot be assembled reliably. The default value is 0.1, or 10%, of the major isoform of the gene.

Example: `'MinIsoformFraction',0.20`

Data Types: `double`

`MultiReadCorrection` — Flag to improve abundance estimation using rescue method
`false` (default) | `true`

Flag to improve abundance estimation for reads mapped to multiple genomic positions using the rescue method, specified as `true` or `false`. If the value is `false`, the function divides multimapped reads uniformly to all mapped positions. If the value is `true`, the function uses additional information, including gene abundance estimation, inferred fragment length, and fragment bias, to improve transcript abundance estimation.

The rescue method is described in [2].

Example: `true`

Data Types: `logical`

`NormalizeCompatibleHits` — Flag to use only fragments compatible with reference transcript to calculate FPKM values
`false` (default) | `true`

Flag to use only fragments compatible with a reference transcript to calculate FPKM values, specified as `true` or `false`.

Example: `'NormalizeCompatibleHits',false`

Data Types: `logical`

`NormalizeTotalHits` — Flag to include all fragments to calculate FPKM values
`false` (default) | `true`

Flag to include all fragments to calculate FPKM values, specified as `true` or `false`. If the value is `true`, the function includes all fragments, including fragments without a compatible reference.

Example: `'NormalizeTotalHits',true`

Data Types: `logical`

`NumFragAssignmentDraws` — Number of fragment assignments to perform on each transcript
`50` (default) | positive integer

Number of fragment assignments to perform on each transcript, specified as a positive integer. For each fragment drawn from a transcript, the function performs the specified number of assignments probabilistically to determine the transcript assignment uncertainty and to estimate the variance-covariance matrix for the assigned fragment counts.

Example: `'NumFragAssignmentSamples',40`

Data Types: `double`

`NumFragDraws` — Number of draws from negative binomial random number generator
`100` (default) | positive integer

Number of draws from the negative binomial random number generator for each transcript, specified as a positive integer. Each draw is a number of fragments that the function probabilistically assigns to transcripts in the transcriptome to determine the assignment uncertainty and to estimate the variance-covariance matrix for assigned fragment counts.

Example: `'NumFragSamples',90`

Data Types: `double`

`NumThreads` — Number of parallel threads to use
`1` (default) | positive integer

Number of parallel threads to use, specified as a positive integer. Threads are run on separate processors or cores. Increasing the number of threads generally improves the runtime significantly, but increases the memory footprint.

Example: `'NumThreads',4`

Data Types: `double`

`OutputDirectory` — Directory to store analysis results
current directory (`"./"`) (default) | string | character vector

Directory to store analysis results, specified as a string or character vector.

Example: `'OutputDirectory',"./AnalysisResults/"`

Data Types: `char` | `string`

`OverhangTolerance` — Number of base pairs of overlap with intron
`8` (default) | positive integer

Number of base pairs of overlap with an intron that the function allows when determining if the read is compatible with another transcript, specified as a positive integer.

Example: `'OverhangTolerance',5`

Data Types: `double`

`RABTOverhangTolerance3` — Number of base pairs allowed to overhang 3' end of reference transcript
`600` (default) | positive integer

Number of base pairs allowed to overhang the 3' end of each reference transcript during the RABT assembly, specified as a positive integer. The function uses this property when deciding if an assembled transcript is novel or should be merged with the reference.

Note The function only performs the RABT assembly if you specify `GTFGuide`. Otherwise, `RABTOverhangTolerance3` has no effect on the assembled transcript.

Example: `'OverhangTolerance3',500`

Data Types: `double`

`OverlapRadius` — Distance between transfrags
`50` (default) | positive integer

Distance between transfrags, specified as a positive integer. If the distance is below the specified value, the function merges the transfrags. The default value is `50` base pairs.

Example: `'OverlapRadius',40`

Data Types: `double`

`PreMRNAFraction` — Threshold to include alignments in intronic intervals
`0.15` (default) | scalar between `0` and `1`

Threshold to include alignments in intronic intervals in the assembly, specified as a scalar between `0` and `1`. The function ignores the intronic alignments if the minimum depth of coverage divided by the number of spliced reads is below the specified value. Use this property to filter reads originating from incompletely spliced transcripts.

Example: `'PreMRNAFraction',0.10`

Data Types: `double`

`ReferenceGTF` — Name of GTF or GFF file containing reference annotation
string | character vector

Name of a GTF or GFF file containing reference annotation used to estimate isoform expression, specified as a string or character vector. If you provide a `ReferenceGTF` file, the function does not assemble any novel transcripts and ignores any alignments incompatible with the reference transcripts.

Example: `'ReferenceGTF',"isoest.gtf"`

Data Types: `char` | `string`

`Seed` — Seed for random number generator
`0` (default) | nonnegative integer

Seed for the random number generator, specified as a nonnegative integer. Setting a seed value ensures the reproducibility of the analysis results.

Example: `'Seed',10`

Data Types: `double`

`SmallAnchorFraction` — Minimum percentage of alignment on each side of splice junction
`0.09` (default) | scalar between `0` and `1`

Minimum percentage of alignment on each side of a splice junction, specified as a scalar between `0` and `1`. The function filters alignments with a percentage smaller than this property value prior to assembly.

Example: `'SmallAnchorFraction',0.1`

Data Types: `double`

`TranscriptPrefix` — Prefix for reported transfrags in output GTF file
`"CUFF"` (default) | string | character vector

Prefix for the reported transfrags in the output GTF file, specified as a string or character vector. This option must be a string or character vector with a non-zero length.

Example: `'TranscriptPrefix',"tfrags"`

Data Types: `char` | `string`

`TrimCoverageThreshold` — Minimum average coverage needed for 3' trimming
`10` (default) | positive integer

Minimum average coverage for 3' trimming, specified as a positive integer.

Example: `'TrimCoverageThreshold',8`

Data Types: `double`

`TrimDropoffFraction` — Minimum percentage of average coverage
`0.1` (default) | scalar between `0` and `1`

Minimum percentage of average coverage for trimming the 3' end of assembled transcripts, specified as a scalar between `0` and `1`.

Example: `'TrimDropoffFraction',0.15`

Data Types: `double`

Output Arguments

`transcripts` — Transcript file name
`"./transcripts.gtf"` (default)

Transcript file name, returned as a string. The name of the file is `"transcripts.gtf"`. The file contains the assembled isoforms, along with attributes describing the abundance of reads originating from each transcript.

The output string also includes the directory information defined by `OutputDirectory`. The default is the current directory. If you set `OutputDirectory` to `"/local/tmp/"`, the output becomes `"/local/tmp/transcripts.gtf"`.

`isoforms` — Estimated isoform-level expression file name
`"./isoforms.fpkm_tracking"` (default)

Estimated isoform-level expression file name, returned as a string. By default, the name of the file is `"isoforms.fpkm_tracking"`. The file contains estimates for isoform-level expression in `cufflinks` FPKM tracking format.

The output string also includes the directory information defined by `OutputDirectory`. The default is the current directory. If you set `OutputDirectory` to `"/local/tmp/"`, the output becomes `"/local/tmp/isoforms.fpkm_tracking"`.

genes — Estimated gene-level expression file name
`"./genes.fpkm_tracking"` (default)

Estimated gene-level expression file name, returned as a string. By default, the name of the file is `"genes.fpkm_tracking"`. The file contains estimates for gene-level expression in `cufflinks` FPKM tracking format.

The output string also includes the directory information defined by `OutputDirectory`. The default is the current directory. If you set `OutputDirectory` to `"/local/tmp/"`, the output becomes `"/local/tmp/genes.fpkm_tracking"`.

`skippedTranscripts` — Name of file containing skipped transcripts
`"./skipped.gtf"` (default)

Name of the file containing skipped transcripts when processing a locus, returned as a string. By default, the name of the file is `"skipped.gtf"`. The `'MaxBundleFrags'` option specifies the maximum number of transcripts (fragments) to include for each locus. After reaching the threshold, the function puts the skipped fragments in this file.

The output string also includes the directory information defined by `OutputDirectory`. The default is the current directory. If you set `OutputDirectory` to `"/local/tmp/"`, the output becomes `"/local/tmp/skipped.gtf"`.

Version History

Introduced in R2019a

References

[1] Trapnell, Cole, Brian A Williams, Geo Pertea, Ali Mortazavi, Gordon Kwan, Marijke J van Baren, Steven L Salzberg, Barbara J Wold, and Lior Pachter. “Transcript Assembly and Quantification by RNA-Seq Reveals Unannotated Transcripts and Isoform Switching during Cell Differentiation.” *Nature Biotechnology* 28, no. 5 (May 2010): 511–15.

[2] Mortazavi, Ali, Brian A Williams, Kenneth McCue, Lorian Schaeffer, and Barbara Wold. “Mapping and Quantifying Mammalian Transcriptomes by RNA-Seq.” *Nature Methods* 5, no. 7 (July 2008): 621–28. https://doi.org/10.1038/nmeth.1226.

See Also

`CufflinksOptions` | `cuffcompare` | `cuffdiff` | `cuffmerge` | `cuffnorm` | `cuffquant` | `cuffgffread` | `cuffgtf2sam` | `bowtie2` | `bwamem`

Topics

“Bioinformatics Toolbox Software Support Packages”

External Websites

Cufflinks manual

CufflinksOptions

Option set for `cufflinks`

Description

A `CufflinksOptions` object contains options for the `cufflinks` function, which assembles a transcriptome from aligned reads [1].

Creation

Syntax

```
cufflinksOpt = CufflinksOptions
cufflinksOpt = CufflinksOptions(Name,Value)
cufflinksOpt = CufflinksOptions(S)
```

Description

`cufflinksOpt = CufflinksOptions` creates a `CufflinksOptions` object with the default property values.

`CufflinksOptions` requires the Cufflinks Support Package for the Bioinformatics Toolbox. If the support package is not installed, then the function provides a download link. For details, see "Bioinformatics Toolbox Software Support Packages".

`cufflinksOpt = CufflinksOptions(Name,Value)` sets the object properties on page 1-680 using one or more name-value pair arguments. Enclose each property name in quotes. For example, `cufflinksOpt = CufflinksOptions('TrimCoverageThreshold',5)` specifies the minimum average coverage for 3' end trimming.

`cufflinksOpt = CufflinksOptions(S)` specifies optional parameters using a string or character vector `S`.

Input Arguments

S — Cufflinks options
character vector | string

Cufflinks options, specified as a character vector or string. `S` must be in the Cufflinks option syntax (prefixed by one or two dashes).

Example: `'--trim-3-avgcov-thresh 5'`

Properties

`EffectiveLengthCorrection` — Flag to normalize fragment counts

`true` (default) | `false`

Flag to normalize fragment counts to fragments per kilobase per million mapped reads (FPKM), specified as `true` or `false`.

Example: `false`

Data Types: `logical`

`ExtraCommand` — Additional commands

`""` (default) | character vector | string

Additional commands, specified as a character vector or string.

The commands must be in the native syntax (prefixed by one or two dashes). Use this option to apply undocumented flags and flags without corresponding MATLAB properties.

When the software converts the original flags to MATLAB properties, it stores any unrecognized flags in this property.

Example: `'--library-type fr-secondstrand'`

Data Types: `char` | `string`

`FauxReadTiling` — Flag to include reference transcripts in assembled output

`true` (default) | `false`

Flag to include reference transcripts in the assembled output as faux-reads during RABT (advanced reference annotation based transcript) assembly, specified as `true` or `false`.

Note The function only performs the RABT assembly if you specify `GTFGuide`. Otherwise, `FauxReadTiling`, regardless of being `true` or `false`, has no effect on the assembled transcript.

Example: `false`

Data Types: `logical`

`FragmentBiasCorrection` — Name of FASTA file with reference transcripts to detect bias

string | character vector

Name of the FASTA file with reference transcripts to detect bias in fragment counts, specified as a string or character vector. Library preparation can introduce sequence-specific bias into RNA-Seq experiments. Providing reference transcripts improves the accuracy of the transcript abundance estimates.

Example: `"bias.fasta"`

Data Types: `char` | `string`

`FragmentLengthMean` — Expected mean fragment length

`200` (default) | positive integer

Expected mean fragment length, specified as a positive integer. The default value is `200` base pairs. The function can learn the fragment length mean for each SAM file. Using this option is not recommended for paired-end reads.

Example: `100`

Data Types: `double`

`FragmentLengthSD` — Expected standard deviation for fragment length distribution

`80` (default) | positive scalar

Expected standard deviation for the fragment length distribution, specified as a positive scalar. The default value is `80` base pairs. The function can learn the fragment length standard deviation for each SAM file. Using this option is not recommended for paired-end reads.

Example: `70`

Data Types: `double`

`GTFGuide` — Name of GTF file to guide RABT assembly

string | character vector

Name of a GTF file to guide the RABT assembly, specified as a string or character vector.

Example: `'tr.gtf'`

Data Types: `char` | `string`

`IncludeAll` — Flag to use all object properties

`false` (default) | true

Flag to include all the object properties with the corresponding default values when converting to the original options syntax, specified as `true` or `false`. You can convert the properties to the original syntax prefixed by one or two dashes (such as `'-d 100 -e 80'`) by using `getCommand`. The default value `false` means that when you call `getCommand(optionsObject)`, it converts only the specified properties. If the value is `true`, `getCommand` converts all available properties, with default values for unspecified properties, to the original syntax.

Note If you set `IncludeAll` to `true`, the software translates all available properties, with default values for unspecified properties. The only exception is that when the default value of a property is `NaN`, `Inf`, `[]`, `''`, or `""`, then the software does not translate the corresponding property.

Example: `true`

Data Types: `logical`

`JunctionAlpha` — Alpha value in binomial test to filter false-positive alignments

`0.001` (default) | scalar between `0` and `1`

Alpha value in the binomial test to filter false-positive alignments, specified as a scalar between `0` and `1`.

Example: `0.005`

Data Types: `double`

`LengthCorrection` — Flag to correct by transcript length

`true` (default) | `false`

Flag to correct by the transcript length, specified as `true` or `false`. Set this value to `false` only when the fragment count is independent of the feature size, such as for small RNA libraries with no fragmentation and for 3' end sequencing, where all fragments have the same length.

Example: `false`

Data Types: `logical`

`MaskFile` — Name of GTF or GFF file containing transcripts to ignore

string | character vector

Name of the GTF or GFF file containing transcripts to ignore during analysis, specified as a string or character vector. Some examples of transcripts to ignore include annotated rRNA transcripts, mitochondrial transcripts, and other abundant transcripts. Ignoring these transcripts improves the robustness of the abundance estimates.

Example: `'excludes.gtf'`

Data Types: `char` | `string`

`MaxBundleFrags` — Maximum number of fragments to include for each locus before skipping

`500000` (default) | positive integer

Maximum number of fragments to include for each locus before skipping new fragments, specified as a positive integer. Skipped fragments are marked with the status `HIDATA` in the file `skipped.gtf`.

Example: `400000`

Data Types: `double`

`MaxBundleLength` — Maximum genomic length in base pairs for bundle

`3500000` (default) | positive integer

Maximum genomic length in base pairs for a bundle, specified as a positive integer.

Example: `3400000`

Data Types: `double`

`MaxFragAlignments` — Maximum number of aligned reads to include for each fragment

`Inf` (default) | positive integer

Maximum number of aligned reads to include for each fragment before skipping new reads, specified as a positive integer. `Inf`, the default value, sets no limit on the maximum number of aligned reads.

Example: `1000`

Data Types: `double`

`MaxIntronLength` — Maximum number of bases in intron

`300000` (default) | positive integer

Maximum number of bases in an intron to report, specified as a positive integer. `cufflinks` also ignores SAM alignments with REF_SKIP CIGAR operations longer than this property value.

Example: `350000`

Data Types: `double`

`MaxMLEIterations` — Maximum number of iterations for maximum likelihood estimation

`5000` (default) | positive integer

Maximum number of iterations for the maximum likelihood estimation of abundances, specified as a positive integer.

Example: `4000`

Data Types: `double`

`MinFragsPerTransfrag` — Minimum number of aligned RNA-Seq fragments to report

`10` (default) | positive integer

Minimum number of aligned RNA-Seq fragments to report on an assembled transfrag, specified as a positive integer.

Example: `15`

Data Types: `double`

`MinIntronLength` — Minimum number of base pairs for intron in genome

`50` (default) | positive integer

Minimum number of base pairs for an intron in the genome, specified as a positive integer.

Example: `50`

Data Types: `double`

`MinIsoformFraction` — Cuffoff value to report abundance of isoform

`0.1` (default) | scalar between `0` and `1`

Cuffoff value to report the abundance of a particular isoform as a fraction of the most abundant isoform (major isoform), specified as a scalar between `0` and `1`. The function filters out transcripts with abundances below the specified value because isoforms expressed at low levels often cannot be assembled reliably. The default value is 0.1, or 10%, of the major isoform of the gene.

Example: `0.20`

Data Types: `double`

`MultiReadCorrection` — Flag to improve abundance estimation using rescue method

`false` (default) | `true`

Flag to improve abundance estimation for reads mapped to multiple genomic positions using the rescue method, specified as `true` or `false`. If the value is `false`, the function divides multimapped reads uniformly to all mapped positions. If the value is `true`, the function uses additional information, including gene abundance estimation, inferred fragment length, and fragment bias, to improve transcript abundance estimation.

The rescue method is described in [2].

Example: `true`

Data Types: `logical`

`NormalizeCompatibleHits` — Flag to use only fragments compatible with reference transcript to calculate FPKM values

`false` (default) | `true`

Flag to use only fragments compatible with a reference transcript to calculate FPKM values, specified as `true` or `false`.

Example: `true`

Data Types: `logical`

`NormalizeTotalHits` — Flag to include all fragments to calculate FPKM values

`false` (default) | `true`

Flag to include all fragments to calculate FPKM values, specified as `true` or `false`. If the value is `true`, the function includes all fragments, including fragments without a compatible reference.

Example: `true`

Data Types: `logical`

`NumFragAssignmentDraws` — Number of fragment assignments to perform on each transcript

`50` (default) | positive integer

Number of fragment assignments to perform on each transcript, specified as a positive integer. For each fragment drawn from a transcript, the function performs the specified number of assignments probabilistically to determine the transcript assignment uncertainty and to estimate the variance-covariance matrix for the assigned fragment counts.

Example: `40`

Data Types: `double`

`NumFragDraws` — Number of draws from negative binomial random number generator

`100` (default) | positive integer

Number of draws from the negative binomial random number generator for each transcript, specified as a positive integer. Each draw is a number of fragments that the function probabilistically assigns to transcripts in the transcriptome to determine the assignment uncertainty and to estimate the variance-covariance matrix for assigned fragment counts.

Example: `90`

Data Types: `double`

`NumThreads` — Number of parallel threads to use

`1` (default) | positive integer

Number of parallel threads to use, specified as a positive integer. Threads are run on separate processors or cores. Increasing the number of threads generally improves the runtime significantly, but increases the memory footprint.

Example: `4`

Data Types: `double`

`OutputDirectory` — Directory to store analysis results

current directory (`"./"`) (default) | string | character vector

Directory to store analysis results, specified as a string or character vector.

Example: `"./AnalysisResults/"`

Data Types: `char` | `string`

`OverhangTolerance` — Number of base pairs of overlap with intron

`8` (default) | positive integer

Number of base pairs of overlap with an intron that the function allows when determining if the read is compatible with another transcript, specified as a positive integer.

Example: `5`

Data Types: `double`

`OverlapRadius` — Distance between transfrags

`50` (default) | positive integer

Distance between transfrags, specified as a positive integer. If the distance is below the specified value, the function merges the transfrags. The default value is `50` base pairs.

Example: `40`

Data Types: `double`

`PreMRNAFraction` — Threshold to include alignments in intronic intervals

`0.15` (default) | scalar between `0` and `1`

Threshold to include alignments in intronic intervals in the assembly, specified as a scalar between `0` and `1`. The function ignores the intronic alignments if the minimum depth of coverage divided by the number of spliced reads is below the specified value. Use this property to filter reads originating from incompletely spliced transcripts.

Example: `0.10`

Data Types: `double`

`RABTOverhangTolerance` — Number of base pairs allowed to overlap with transcript intron

`8` (default) | positive integer

Number of base pairs from a read allowed to overlap with a transcript intron when determining if a read is mappable to another transcript during the RABT assembly, specified as a positive integer. The default value is `8`.

Note The function only performs the RABT assembly if you specify `GTFGuide`. Otherwise, `RABTOverhangTolerance` has no effect on the assembled transcript.

Example: `10`

Data Types: `double`

`RABTOverhangTolerance3` — Number of base pairs allowed to overhang 3' end of reference transcript

`600` (default) | positive integer

Number of base pairs allowed to overhang the 3' end of each reference transcript during the RABT assembly, specified as a positive integer. The function uses this property when deciding if an assembled transcript is novel or should be merged with the reference.

Note The function only performs the RABT assembly if you specify `GTFGuide`. Otherwise, `RABTOverhangTolerance3` has no effect on the assembled transcript.

Example: `500`

Data Types: `double`

`ReferenceGTF` — Name of GTF or GFF file used to estimate isoform expression

string | character vector

Name of a GTF or GFF file containing reference annotation used to estimate isoform expression, specified as a string or character vector. If you provide a `ReferenceGTF` file, the function does not assemble any novel transcripts and ignores any alignments incompatible with the reference transcripts.

Example: `'isoest.gtf'`

Data Types: `char` | `string`

Seed — Seed for random number generator

`0` (default) | nonnegative integer

Seed for the random number generator, specified as a nonnegative integer. Setting a seed value ensures the reproducibility of the analysis results.

Example: `10`

Data Types: `double`

`SmallAnchorFraction` — Minimum percentage of alignment on each side of splice junction

`0.09` (default) | scalar between `0` and `1`

Minimum percentage of alignment on each side of a splice junction, specified as a scalar between `0` and `1`. The function filters alignments with a percentage smaller than this property value prior to assembly.

Example: `0.1`

Data Types: `double`

`TranscriptPrefix` — Prefix for reported transfrags in output GTF file

`"CUFF"` (default) | string | character vector

Prefix for the reported transfrags in the output GTF file, specified as a string or character vector. This option must be a string or character vector with a non-zero length.

Example: `"tfrags"`

Data Types: `char` | `string`

`TrimCoverageThreshold` — Minimum average coverage needed for 3' trimming

`10` (default) | positive integer

Minimum average coverage for 3' trimming, specified as a positive integer.

Example: `8`

Data Types: `double`

`TrimDropoffFraction` — Minimum percentage of average coverage

`0.1` (default) | scalar between `0` and `1`

Minimum percentage of average coverage for trimming the 3' end of assembled transcripts, specified as a scalar between `0` and `1`.

Example: `0.15`

Data Types: `double`

`Version` — Supported version
string

This property is read-only.

Supported version of the original cufflinks software, returned as a string.

Example: `"2.2.1"`

Data Types: `string`

Object Functions

getCommand	Translate object properties to original options syntax
getOptionsTable	Return table with all properties and equivalent options in original syntax

Examples

Create CufflinksOptions

Create a `CufflinksOptions` object with default values.

```
opt = CufflinksOptions;
```

Create an object using name-value pairs.

```
opt2 = CufflinksOptions('TranscriptPrefix',"MATLAB",'NumThreads',4)
```

Create an object by using the original `cufflinks` syntax.

```
opt3 = CufflinksOptions('--label MATLAB --num-threads 4')
```

Assemble Transcriptome and Perform Differential Expression Testing

Create a `CufflinksOptions` object to define cufflinks options, such as the number of parallel threads and the output directory to store the results.

```
cflOpt = CufflinksOptions;
cflOpt.NumThreads = 8;
cflOpt.OutputDirectory = "./cufflinksOut";
```

The SAM files provided for this example contain aligned reads for *Mycoplasma pneumoniae* from two samples with three replicates each. The reads are simulated 100bp-reads for two genes (`gyrA` and `gyrB`) located next to each other on the genome. All the reads are sorted by reference position, as required by `cufflinks`.

```
sams = ["Myco_1_1.sam","Myco_1_2.sam","Myco_1_3.sam",...
        "Myco_2_1.sam", "Myco_2_2.sam", "Myco_2_3.sam"];
```

Assemble the transcriptome from the aligned reads.

```
[gtfs,isofpkm,genes,skipped] = cufflinks(sams,cflOpt);
```

`gtfs` is a list of GTF files that contain assembled isoforms.

Compare the assembled isoforms using `cuffcompare`.

```
stats = cuffcompare(gtfs);
```

Merge the assembled transcripts using `cuffmerge`.

```
mergedGTF = cuffmerge(gtfs,'OutputDirectory','./cuffMergeOutput');
```

`mergedGTF` reports only one transcript. This is because the two genes of interest are located next to each other, and `cuffmerge` cannot distinguish two distinct genes. To guide `cuffmerge`, use a reference GTF (`gyrAB.gtf`) containing information about these two genes. If the file is not located in the same directory that you run `cuffmerge` from, you must also specify the file path.

```
gyrAB = which('gyrAB.gtf');
mergedGTF2 = cuffmerge(gtfs,'OutputDirectory','./cuffMergeOutput2',...
            'ReferenceGTF',gyrAB);
```

Calculate abundances (expression levels) from aligned reads for each sample.

```
abundances1 = cuffquant(mergedGTF2,["Myco_1_1.sam","Myco_1_2.sam","Myco_1_3.sam"],...
                        'OutputDirectory','./cuffquantOutput1');
abundances2 = cuffquant(mergedGTF2,["Myco_2_1.sam", "Myco_2_2.sam", "Myco_2_3.sam"],...
                        'OutputDirectory','./cuffquantOutput2');
```

Assess the significance of changes in expression for genes and transcripts between conditions by performing the differential testing using `cuffdiff`. The `cuffdiff` function operates in two distinct steps: the function first estimates abundances from aligned reads, and then performs the statistical analysis. In some cases (for example, distributing computing load across multiple workers), performing the two steps separately is desirable. After performing the first step with `cuffquant`, you can then use the binary CXB output file as an input to `cuffdiff` to perform statistical analysis. Because `cuffdiff` returns several files, specify the output directory is recommended.

```
isoformDiff = cuffdiff(mergedGTF2,[abundances1,abundances2],...
                      'OutputDirectory','./cuffdiffOutput');
```

Display a table containing the differential expression test results for the two genes `gyrB` and `gyrA`.

```
readtable(isoformDiff,'FileType','text')
```

```
ans =

  2×14 table

        test_id              gene_id         gene              locus               sample_1    sample_
    ________________    _______________    ______    _________________________    ________    _______

    'TCONS_00000001'    'XLOC_000001'     'gyrB'    'NC_000912.1:2868-7340'       'q1'         'q2'
    'TCONS_00000002'    'XLOC_000001'     'gyrA'    'NC_000912.1:2868-7340'       'q1'         'q2'
```

You can use `cuffnorm` to generate normalized expression tables for further analyses. `cuffnorm` results are useful when you have many samples and you want to cluster them or plot expression levels for genes that are important in your study. Note that you cannot perform differential expression analysis using `cuffnorm`.

Specify a cell array, where each element is a string vector containing file names for a single sample with replicates.

```
alignmentFiles = {["Myco_1_1.sam","Myco_1_2.sam","Myco_1_3.sam"],...
                  ["Myco_2_1.sam", "Myco_2_2.sam", "Myco_2_3.sam"]}
isoformNorm = cuffnorm(mergedGTF2, alignmentFiles,...
                      'OutputDirectory', './cuffnormOutput');
```

Display a table containing the normalized expression levels for each transcript.

```
readtable(isoformNorm,'FileType','text')
```

```
ans =
  2×7 table

     tracking_id         q1_0          q1_2          q1_1          q2_1          q2_0
    ________________    __________    __________    __________    __________    __________    ___

    'TCONS_00000001'    1.0913e+05         78628    1.2132e+05    4.3639e+05    4.2228e+05    4.2
    'TCONS_00000002'    3.5158e+05    3.7458e+05    3.4238e+05    1.0483e+05    1.1546e+05    1.1
```

Column names starting with *q* have the format: *conditionX_N*, indicating that the column contains values for replicate *N* of *conditionX*.

Version History

Introduced in R2019a

References

[1] Trapnell, Cole, Brian A Williams, Geo Pertea, Ali Mortazavi, Gordon Kwan, Marijke J van Baren, Steven L Salzberg, Barbara J Wold, and Lior Pachter. “Transcript Assembly and Quantification by RNA-Seq Reveals Unannotated Transcripts and Isoform Switching during Cell Differentiation.” *Nature Biotechnology* 28, no. 5 (May 2010): 511–15.

[2] Mortazavi, Ali, Brian A Williams, Kenneth McCue, Lorian Schaeffer, and Barbara Wold. “Mapping and Quantifying Mammalian Transcriptomes by RNA-Seq.” *Nature Methods* 5, no. 7 (July 2008): 621–28. https://doi.org/10.1038/nmeth.1226.

See Also

`cufflinks` | `cuffgffread` | `cuffgtf2sam` | `cuffcompare` | `cuffdiff` | `cuffmerge` | `cuffnorm` | `cuffquant`

Topics
“Bioinformatics Toolbox Software Support Packages”

External Websites
Cufflinks manual

cuffmerge

Merge RNA-seq assemblies

Syntax

```
mergedGTF = cuffmerge(gtfFiles)
mergedGTF = cuffmerge(gtfFiles,opt)
mergedGTF = cuffmerge(gtfFiles,Name,Value)
```

Description

`mergedGTF = cuffmerge(gtfFiles)` merges assembled transcriptome from two or more GTF files [1]. Merging GTF files is a required step to perform the downstream differential analysis with `cuffdiff`.

`cuffmerge` requires Python® 2 installed in your system.

`cuffmerge` requires the Cufflinks Support Package for the Bioinformatics Toolbox. If the support package is not installed, then the function provides a download link. For details, see “Bioinformatics Toolbox Software Support Packages”.

`mergedGTF = cuffmerge(gtfFiles,opt)` uses additional options specified by `opt`.

`mergedGTF = cuffmerge(gtfFiles,Name,Value)` uses additional options specified by one or more name-value pair arguments. For example, `cuffmerge(["Myco_1_1.transcripts.gtf","Myco_1_2.transcripts.gtf"],'NumThreads',5)` specifies to use five parallel threads.

Examples

Assemble Transcriptome and Perform Differential Expression Testing

Create a `CufflinksOptions` object to define cufflinks options, such as the number of parallel threads and the output directory to store the results.

```
cflOpt = CufflinksOptions;
cflOpt.NumThreads = 8;
cflOpt.OutputDirectory = "./cufflinksOut";
```

The SAM files provided for this example contain aligned reads for *Mycoplasma pneumoniae* from two samples with three replicates each. The reads are simulated 100bp-reads for two genes (`gyrA` and `gyrB`) located next to each other on the genome. All the reads are sorted by reference position, as required by `cufflinks`.

```
sams = ["Myco_1_1.sam","Myco_1_2.sam","Myco_1_3.sam",...
        "Myco_2_1.sam", "Myco_2_2.sam", "Myco_2_3.sam"];
```

Assemble the transcriptome from the aligned reads.

```
[gtfs,isofpkm,genes,skipped] = cufflinks(sams,cflOpt);
```

`gtfs` is a list of GTF files that contain assembled isoforms.

Compare the assembled isoforms using `cuffcompare`.

```
stats = cuffcompare(gtfs);
```

Merge the assembled transcripts using `cuffmerge`.

```
mergedGTF = cuffmerge(gtfs,'OutputDirectory','./cuffMergeOutput');
```

`mergedGTF` reports only one transcript. This is because the two genes of interest are located next to each other, and `cuffmerge` cannot distinguish two distinct genes. To guide `cuffmerge`, use a reference GTF (`gyrAB.gtf`) containing information about these two genes. If the file is not located in the same directory that you run `cuffmerge` from, you must also specify the file path.

```
gyrAB = which('gyrAB.gtf');
mergedGTF2 = cuffmerge(gtfs,'OutputDirectory','./cuffMergeOutput2',...
            'ReferenceGTF',gyrAB);
```

Calculate abundances (expression levels) from aligned reads for each sample.

```
abundances1 = cuffquant(mergedGTF2,["Myco_1_1.sam","Myco_1_2.sam","Myco_1_3.sam"],...
                        'OutputDirectory','./cuffquantOutput1');
abundances2 = cuffquant(mergedGTF2,["Myco_2_1.sam", "Myco_2_2.sam", "Myco_2_3.sam"],...
                        'OutputDirectory','./cuffquantOutput2');
```

Assess the significance of changes in expression for genes and transcripts between conditions by performing the differential testing using `cuffdiff`. The `cuffdiff` function operates in two distinct steps: the function first estimates abundances from aligned reads, and then performs the statistical analysis. In some cases (for example, distributing computing load across multiple workers), performing the two steps separately is desirable. After performing the first step with `cuffquant`, you can then use the binary CXB output file as an input to `cuffdiff` to perform statistical analysis. Because `cuffdiff` returns several files, specify the output directory is recommended.

```
isoformDiff = cuffdiff(mergedGTF2,[abundances1,abundances2],...
                      'OutputDirectory','./cuffdiffOutput');
```

Display a table containing the differential expression test results for the two genes `gyrB` and `gyrA`.

```
readtable(isoformDiff,'FileType','text')
```

```
ans =

  2×14 table

        test_id            gene_id        gene              locus              sample_1     sample_
    ________________    _____________    ______    _______________________    ________     _______

    'TCONS_00000001'    'XLOC_000001'    'gyrB'    'NC_000912.1:2868-7340'      'q1'          'q2'
    'TCONS_00000002'    'XLOC_000001'    'gyrA'    'NC_000912.1:2868-7340'      'q1'          'q2'
```

You can use `cuffnorm` to generate normalized expression tables for further analyses. `cuffnorm` results are useful when you have many samples and you want to cluster them or plot expression levels for genes that are important in your study. Note that you cannot perform differential expression analysis using `cuffnorm`.

Specify a cell array, where each element is a string vector containing file names for a single sample with replicates.

```
alignmentFiles = {["Myco_1_1.sam","Myco_1_2.sam","Myco_1_3.sam"],...
                  ["Myco_2_1.sam", "Myco_2_2.sam", "Myco_2_3.sam"]}
isoformNorm = cuffnorm(mergedGTF2, alignmentFiles,...
                      'OutputDirectory', './cuffnormOutput');
```

Display a table containing the normalized expression levels for each transcript.

```
readtable(isoformNorm,'FileType','text')
```

```
ans =

  2×7 table

      tracking_id          q1_0           q1_2           q1_1           q2_1           q2_0
    _________________    ___________    ___________    ___________    ___________    ___________    ___

    'TCONS_00000001'     1.0913e+05          78628     1.2132e+05     4.3639e+05     4.2228e+05     4.
    'TCONS_00000002'     3.5158e+05     3.7458e+05     3.4238e+05     1.0483e+05     1.1546e+05     1.
```

Column names starting with *q* have the format: *conditionX_N*, indicating that the column contains values for replicate *N* of *conditionX*.

Input Arguments

`gtfFiles` — Names of GTF files

string vector | cell array of character vectors

Names of GTF files, specified as a string vector or cell array of character vectors.

Example: `["Myco_1_1.transcripts.gtf", "Myco_1_2.transcripts.gtf"]`

Data Types: `string` | `cell`

`opt` — `cuffgffread` options

`CuffMergeOptions` object | string | character vector

`cuffgffread` options, specified as a `CuffMergeOptions` object, string, or character vector. The string or character vector must be in the original `cuffmerge` option syntax (prefixed by one or two dashes) [1].

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example:
`cuffmerge(["Myco_1_1.transcripts.gtf","Myco_1_2.transcripts.gtf"],'NumThreads',5)`

`ExtraCommand` — Additional commands

`""` (default) | string | character vector

The commands must be in the native syntax (prefixed by one or two dashes). Use this option to apply undocumented flags and flags without corresponding MATLAB properties.

Example: `'ExtraCommand','--library-type fr-secondstrand'`

Data Types: `char` | `string`

`IncludeAll` — Flag to apply all available options
`false` (default) | `true`

The original (native) syntax is prefixed by one or two dashes. By default, the function converts only the specified options. If the value is `true`, the software converts all available options, with default values for unspecified options, to the original syntax.

Note If you set `IncludeAll` to `true`, the software translates all available properties, with default values for unspecified properties. The only exception is that when the default value of a property is `NaN`, `Inf`, `[]`, `''`, or `""`, then the software does not translate the corresponding property.

Example: `'IncludeAll',true`

Data Types: `logical`

`MinIsoformFraction` — Minimum abundance of isoform to be included in merged assembly
`0.5` (default) | scalar between `0` and `1`

Minimum abundance of an isoform to be included in the merged assembly, specified as a scalar between `0` and `1`. This value is expressed as a percentage of the most abundant (major) isoform.

Example: `'MinIsoformFraction',0.4`

Data Types: `double`

`NumThreads` — Number of parallel threads to use
`1` (default) | positive integer

Number of parallel threads to use, specified as a positive integer. Threads are run on separate processors or cores. Increasing the number of threads generally improves the runtime significantly, but increases the memory footprint.

Example: `'NumThreads',4`

Data Types: `double`

`OutputDirectory` — Directory to store analysis results
current directory (`"./"`) (default) | string | character vector

Directory to store analysis results, specified as a string or character vector.

Example: `'OutputDirectory',"./AnalysisResults/"`

Data Types: `char` | `string`

`ReferenceGTF` — Name of optional reference annotation GTF file
string | character vector

Name of an optional reference annotation GTF file to be included in the combined assembly, specified as a string or character vector.

Example: `'ReferenceGTF',"ref.gtf"`

Data Types: `char` | `string`

`ReferenceSequence` — Name of directory or FASTA file containing genomic sequences
string | character vector

Name of a directory or FASTA file containing genomic DNA sequences for the reference, specified as a string or character vector.

- If you specify a directory, it must contain one FASTA file per contig. In other words, the directory must contain one FASTA file per reference chromosome, and each file must be named after the chromosome and have a `.fa` or `.fasta` extension.
- If you specify a FASTA file, it must contain all the reference sequences.

The function uses the provided sequences to improve transfrag classification and exclude artifacts.

Example: `'ReferenceSequence',"allrefs.fasta"`

Data Types: `char` | `string`

Output Arguments

`mergedGTF` — Name of output GTF file
`"./merged_asm/merged.gtf"`

Name of the output GTF file containing the merged transcriptome, returned as a string.

The output string also includes the directory information defined by `OutputDirectory`. By default, the function

- Creates the *merged_asm* subfolder in the current directory and saves the output file (`merged.gtf`) in that folder.
- Creates a subfolder named *logs* inside *merged_asm* folder and saves a log file.

If you set `OutputDirectory` to `"/local/tmp/"`, `mergedGTF` becomes `"/local/tmp/merged.gtf"`. The function also creates the *logs* folder inside the specified output directory.

Version History

Introduced in R2019a

References

[1] Trapnell, Cole, Brian A Williams, Geo Pertea, Ali Mortazavi, Gordon Kwan, Marijke J van Baren, Steven L Salzberg, Barbara J Wold, and Lior Pachter. “Transcript Assembly and Quantification by RNA-Seq Reveals Unannotated Transcripts and Isoform Switching during Cell Differentiation.” *Nature Biotechnology* 28, no. 5 (May 2010): 511–15.

See Also

`cufflinks` | `CuffMergeOptions`

Topics
“Bioinformatics Toolbox Software Support Packages”

External Websites
Cufflinks manual

CuffMergeOptions

Option set for `cuffmerge`

Description

A `CuffMergeOptions` object contains options for the `cuffmerge` function, which merges cufflinks transcript assemblies [1].

Creation

Syntax

```
cuffmergeOpt = CuffMergeOptions
cuffmergeOpt = CuffMergeOptions(Name,Value)
cuffmergeOpt = CuffMergeOptions(S)
```

Description

`cuffmergeOpt = CuffMergeOptions` creates a `CuffMergeOptions` object with the default property values.

`CuffMergeOptions` requires the Cufflinks Support Package for the Bioinformatics Toolbox. If the support package is not installed, then the function provides a download link. For details, see “Bioinformatics Toolbox Software Support Packages”.

`cuffmergeOpt = CuffMergeOptions(Name,Value)` sets the object properties on page 1-698 using one or more name-value pair arguments. Enclose each property name in quotes. For example, `cuffmergeOpt = CuffMergeOptions('NumThreads',8)` specifies eight parallel threads.

`cuffmergeOpt = CuffMergeOptions(S)` specifies optional parameters using a string or character vector `S`.

Input Arguments

`S` — `cuffmerge` options
string | character vector

`cuffmerge` options, specified as a string or character vector. `S` must be in the original `cuffmerge` option syntax (prefixed by one or two dashes).

Example: `'--num-thread 5'`

Properties

`ExtraCommand` — Additional commands

`""` (default) | string | character vector

The commands must be in the native syntax (prefixed by one or two dashes). Use this option to apply undocumented flags and flags without corresponding MATLAB properties.

When the software converts the original flags to MATLAB properties, it stores any unrecognized flags in this property.

Example: `'--library-type fr-secondstrand'`

Data Types: `char` | `string`

`IncludeAll` — Flag to use all object properties

`false` (default) | `true`

Flag to include all the object properties with the corresponding default values when converting to the original options syntax, specified as `true` or `false`. You can convert the properties to the original syntax prefixed by one or two dashes (such as `'-d 100 -e 80'`) by using `getCommand`. The default value `false` means that when you call `getCommand(optionsObject)`, it converts only the specified properties. If the value is `true`, `getCommand` converts all available properties, with default values for unspecified properties, to the original syntax.

Note If you set `IncludeAll` to `true`, the software translates all available properties, with default values for unspecified properties. The only exception is that when the default value of a property is `NaN`, `Inf`, `[]`, `''`, or `""`, then the software does not translate the corresponding property.

Example: `true`

Data Types: `logical`

`MinIsoformFraction` — Minimum abundance of isoform to be included in merged assembly

`0.5` (default) | scalar between `0` and `1`

Minimum abundance of an isoform to be included in the merged assembly, specified as a scalar between `0` and `1`. This value is expressed as a percentage of the most abundant (major) isoform.

Example: `0.4`

Data Types: `double`

`NumThreads` — Number of parallel threads to use

`1` (default) | positive integer

Number of parallel threads to use, specified as a positive integer. Threads are run on separate processors or cores. Increasing the number of threads generally improves the runtime significantly, but increases the memory footprint.

Example: `4`

Data Types: `double`

`OutputDirectory` — Directory to store analysis results

current directory (`"./"`) (default) | string | character vector

Directory to store analysis results, specified as a string or character vector.

Example: `"./AnalysisResults/"`

Data Types: `char` | `string`

`ReferenceGTF` — Name of optional reference annotation GTF file

string | character vector

Name of an optional reference annotation GTF file to be included in the combined assembly, specified as a string or character vector.

Example: `"ref.gtf"`

Data Types: `char` | `string`

`ReferenceSequence` — Name of directory or FASTA file containing genomic sequences

string | character vector

Name of a directory or FASTA file containing genomic DNA sequences for the reference, specified as a string or character vector.

- If you specify a directory, it must contain one FASTA file per contig. In other words, the directory must contain one FASTA file per reference chromosome, and each file must be named after the chromosome and have a `.fa` or `.fasta` extension.
- If you specify a FASTA file, it must contain all the reference sequences.

The function uses the provided sequences to improve transfrag classification and exclude artifacts.

Example: `"allrefs.fasta"`

Data Types: `char` | `string`

`Version` — Supported version
string

This property is read-only.

Supported version of the original cufflinks software, returned as a string.

Example: `"2.2.1"`

Data Types: `string`

Object Functions

getCommand	Translate object properties to original options syntax
getOptionsTable	Return table with all properties and equivalent options in original syntax

Examples

Create CuffMergeOptions Object

Create a `CuffMergeOptions` object with the default values.

```
opt = CuffMergeOptions;
```

Create an object using name-value pairs.

```
opt2 = CuffMergeOptions('OutputDirectory',"./merged",'MinIsoformFraction',0.1)
```

Create an object by using the original syntax.

```
opt3 = CuffMergeOptions('-o ./merged --min-isoform-fraction 0.1')
```

Assemble Transcriptome and Perform Differential Expression Testing

Create a `CufflinksOptions` object to define cufflinks options, such as the number of parallel threads and the output directory to store the results.

```
cflOpt = CufflinksOptions;
cflOpt.NumThreads = 8;
cflOpt.OutputDirectory = "./cufflinksOut";
```

The SAM files provided for this example contain aligned reads for *Mycoplasma pneumoniae* from two samples with three replicates each. The reads are simulated 100bp-reads for two genes (`gyrA` and `gyrB`) located next to each other on the genome. All the reads are sorted by reference position, as required by `cufflinks`.

```
sams = ["Myco_1_1.sam","Myco_1_2.sam","Myco_1_3.sam",...
        "Myco_2_1.sam", "Myco_2_2.sam", "Myco_2_3.sam"];
```

Assemble the transcriptome from the aligned reads.

```
[gtfs,isofpkm,genes,skipped] = cufflinks(sams,cflOpt);
```

`gtfs` is a list of GTF files that contain assembled isoforms.

Compare the assembled isoforms using `cuffcompare`.

```
stats = cuffcompare(gtfs);
```

Merge the assembled transcripts using `cuffmerge`.

```
mergedGTF = cuffmerge(gtfs,'OutputDirectory','./cuffMergeOutput');
```

`mergedGTF` reports only one transcript. This is because the two genes of interest are located next to each other, and `cuffmerge` cannot distinguish two distinct genes. To guide `cuffmerge`, use a reference GTF (`gyrAB.gtf`) containing information about these two genes. If the file is not located in the same directory that you run `cuffmerge` from, you must also specify the file path.

```
gyrAB = which('gyrAB.gtf');
mergedGTF2 = cuffmerge(gtfs,'OutputDirectory','./cuffMergeOutput2',...
            'ReferenceGTF',gyrAB);
```

Calculate abundances (expression levels) from aligned reads for each sample.

```
abundances1 = cuffquant(mergedGTF2,["Myco_1_1.sam","Myco_1_2.sam","Myco_1_3.sam"],...
                        'OutputDirectory','./cuffquantOutput1');
abundances2 = cuffquant(mergedGTF2,["Myco_2_1.sam", "Myco_2_2.sam", "Myco_2_3.sam"],...
                        'OutputDirectory','./cuffquantOutput2');
```

Assess the significance of changes in expression for genes and transcripts between conditions by performing the differential testing using `cuffdiff`. The `cuffdiff` function operates in two distinct steps: the function first estimates abundances from aligned reads, and then performs the statistical analysis. In some cases (for example, distributing computing load across multiple workers), performing the two steps separately is desirable. After performing the first step with `cuffquant`, you can then use the binary CXB output file as an input to `cuffdiff` to perform statistical analysis. Because `cuffdiff` returns several files, specify the output directory is recommended.

```
isoformDiff = cuffdiff(mergedGTF2,[abundances1,abundances2],...
                       'OutputDirectory','./cuffdiffOutput');
```

Display a table containing the differential expression test results for the two genes `gyrB` and `gyrA`.

```
readtable(isoformDiff,'FileType','text')
```

```
ans =

  2×14 table

        test_id              gene_id          gene               locus                sample_1     sample_
    __________________    _____________    ______    ___________________________    _________    _______

    'TCONS_00000001'     'XLOC_000001'    'gyrB'    'NC_000912.1:2868-7340'          'q1'          'q2'
    'TCONS_00000002'     'XLOC_000001'    'gyrA'    'NC_000912.1:2868-7340'          'q1'          'q2'
```

You can use `cuffnorm` to generate normalized expression tables for further analyses. `cuffnorm` results are useful when you have many samples and you want to cluster them or plot expression levels for genes that are important in your study. Note that you cannot perform differential expression analysis using `cuffnorm`.

Specify a cell array, where each element is a string vector containing file names for a single sample with replicates.

```
alignmentFiles = {["Myco_1_1.sam","Myco_1_2.sam","Myco_1_3.sam"],...
                  ["Myco_2_1.sam", "Myco_2_2.sam", "Myco_2_3.sam"]}
isoformNorm = cuffnorm(mergedGTF2, alignmentFiles,...
                       'OutputDirectory', './cuffnormOutput');
```

Display a table containing the normalized expression levels for each transcript.

```
readtable(isoformNorm,'FileType','text')
```

```
ans =

  2×7 table

      tracking_id           q1_0           q1_2           q1_1           q2_1           q2_0
    __________________    ___________    ___________    ___________    ___________    ___________    __

    'TCONS_00000001'     1.0913e+05          78628     1.2132e+05     4.3639e+05     4.2228e+05     4.
    'TCONS_00000002'     3.5158e+05     3.7458e+05     3.4238e+05     1.0483e+05     1.1546e+05     1.
```

Column names starting with *q* have the format: *conditionX_N*, indicating that the column contains values for replicate *N* of *conditionX*.

Version History

Introduced in R2019a

References

[1] Trapnell, Cole, Brian A Williams, Geo Pertea, Ali Mortazavi, Gordon Kwan, Marijke J van Baren, Steven L Salzberg, Barbara J Wold, and Lior Pachter. “Transcript Assembly and Quantification by RNA-Seq Reveals Unannotated Transcripts and Isoform Switching during Cell Differentiation.” *Nature Biotechnology* 28, no. 5 (May 2010): 511–15.

See Also

`CufflinksOptions` | `cuffcompare` | `cuffdiff` | `cuffmerge` | `cuffnorm` | `cuffquant` | `cuffgtf2sam` | `cuffgffread`

Topics

“Bioinformatics Toolbox Software Support Packages”

External Websites

Cufflinks manual

cuffnorm

Normalize transcript expression levels

Syntax

```
cuffnorm(transcriptsAnnot,alignmentFiles)
cuffnorm(transcriptsAnnot,alignmentFiles,opt)
cuffnorm(transcriptsAnnot,alignmentFiles,Name,Value)
[isoform,gene,tss,cds] = cuffnorm( ___ )
```

Description

`cuffnorm(transcriptsAnnot,alignmentFiles)` normalizes transcript expression to FPKM for the samples in `alignmentFiles` and corrects for differences in library size [1].

`cuffnorm` requires the Cufflinks Support Package for the Bioinformatics Toolbox. If the support package is not installed, then the function provides a download link. For details, see “Bioinformatics Toolbox Software Support Packages”.

`cuffnorm(transcriptsAnnot,alignmentFiles,opt)` uses additional options specified by `opt`.

`cuffnorm(transcriptsAnnot,alignmentFiles,Name,Value)` uses additional options specified by one or more name-value pair arguments. For example, `cuffnorm('gyrAB.gtf', ["Myco_1_1.sam", "Myco_2_1.sam"],'NumThreads',5)` specifies to use five parallel threads.

`[isoform,gene,tss,cds] = cuffnorm( ___ )` returns the names of files containing normalized results using any of the input argument combinations in the previous syntaxes. By default, the function saves all files to the current directory.

Examples

Assemble Transcriptome and Normalize Expression Levels

Create a `CufflinksOptions` object to define cufflinks options, such as the number of parallel threads and the output directory to store the results.

```
cflOpt = CufflinksOptions;
cflOpt.NumThreads = 8;
cflOpt.OutputDirectory = "./cufflinksOut";
```

The SAM files provided for this example contain aligned reads for *Mycoplasma pneumoniae* from two samples with three replicates each. The reads are simulated 100bp-reads for two genes (`gyrA` and `gyrB`) located next to each other on the genome. All the reads are sorted by reference position, as required by `cufflinks`.

```
sams = ["Myco_1_1.sam","Myco_1_2.sam","Myco_1_3.sam",...
        "Myco_2_1.sam", "Myco_2_2.sam", "Myco_2_3.sam"];
```

Assemble the transcriptome from the aligned reads.

```
[gtfs,isofpkm,genes,skipped] = cufflinks(sams,cflOpt);
```

`gtfs` is a list of GTF files that contain assembled isoforms.

Compare the assembled isoforms using `cuffcompare`.

```
stats = cuffcompare(gtfs);
```

Merge the assembled transcripts using `cuffmerge`.

```
mergedGTF = cuffmerge(gtfs,'OutputDirectory','./cuffMergeOutput');
```

`mergedGTF` reports only one transcript. This is because the two genes of interest are located next to each other, and `cuffmerge` cannot distinguish two distinct genes. To guide `cuffmerge`, use a reference GTF (`gyrAB.gtf`) containing information about these two genes. If the file is not located in the same directory that you run `cuffmerge` from, you must also specify the file path.

```
gyrAB = which('gyrAB.gtf');
mergedGTF2 = cuffmerge(gtfs,'OutputDirectory','./cuffMergeOutput2',...
            'ReferenceGTF',gyrAB);
```

Calculate abundances (expression levels) from aligned reads for each sample.

```
abundances1 = cuffquant(mergedGTF2,["Myco_1_1.sam","Myco_1_2.sam","Myco_1_3.sam"],...
                        'OutputDirectory','./cuffquantOutput1');
abundances2 = cuffquant(mergedGTF2,["Myco_2_1.sam", "Myco_2_2.sam", "Myco_2_3.sam"],...
                        'OutputDirectory','./cuffquantOutput2');
```

Assess the significance of changes in expression for genes and transcripts between conditions by performing the differential testing using `cuffdiff`. The `cuffdiff` function operates in two distinct steps: the function first estimates abundances from aligned reads, and then performs the statistical analysis. In some cases (for example, distributing computing load across multiple workers), performing the two steps separately is desirable. After performing the first step with `cuffquant`, you can then use the binary CXB output file as an input to `cuffdiff` to perform statistical analysis. Because `cuffdiff` returns several files, specify the output directory is recommended.

```
isoformDiff = cuffdiff(mergedGTF2,[abundances1,abundances2],...
                      'OutputDirectory','./cuffdiffOutput');
```

Display a table containing the differential expression test results for the two genes `gyrB` and `gyrA`.

```
readtable(isoformDiff,'FileType','text')

ans =

  2×14 table

        test_id             gene_id         gene              locus               sample_1    sample_
    _________________    _____________    ______    _________________________    ________    _______

    'TCONS_00000001'    'XLOC_000001'    'gyrB'    'NC_000912.1:2868-7340'      'q1'         'q2'
    'TCONS_00000002'    'XLOC_000001'    'gyrA'    'NC_000912.1:2868-7340'      'q1'         'q2'
```

You can use `cuffnorm` to generate normalized expression tables for further analyses. `cuffnorm` results are useful when you have many samples and you want to cluster them or plot expression levels for genes that are important in your study. Note that you cannot perform differential expression analysis using `cuffnorm`.

Specify a cell array, where each element is a string vector containing file names for a single sample with replicates.

```
alignmentFiles = {["Myco_1_1.sam","Myco_1_2.sam","Myco_1_3.sam"],...
                  ["Myco_2_1.sam", "Myco_2_2.sam", "Myco_2_3.sam"]}
isoformNorm = cuffnorm(mergedGTF2, alignmentFiles,...
                      'OutputDirectory', './cuffnormOutput');
```

Display a table containing the normalized expression levels for each transcript.

```
readtable(isoformNorm,'FileType','text')
```

```
ans =

  2×7 table

      tracking_id          q1_0          q1_2          q1_1          q2_1          q2_0
    _________________    __________    __________    __________    __________    __________    __

    'TCONS_00000001'     1.0913e+05         78628    1.2132e+05    4.3639e+05    4.2228e+05    4.2
    'TCONS_00000002'     3.5158e+05    3.7458e+05    3.4238e+05    1.0483e+05    1.1546e+05    1.1
```

Column names starting with *q* have the format: *conditionX_N*, indicating that the column contains values for replicate *N* of *conditionX*.

Input Arguments

`transcriptsAnnot` — Name of transcript annotation file
string | character vector

Name of the transcript annotation file, specified as a string or character vector. The file can be a GTF or GFF file produced by `cufflinks`, `cuffcompare`, or another source of GTF annotations.

Example: `"gyrAB.gtf"`

Data Types: `char` | `string`

`alignmentFiles` — Names of SAM, BAM, or CXB files
string vector | cell array

Names of SAM, BAM, or CXB files containing alignment records for each sample, specified as a string vector or cell array. If you use a cell array, each element must be a string vector or cell array of character vectors specifying alignment files for every replicate of the same sample.

Example: `["Myco_1_1.sam", "Myco_2_1.sam"]`

Data Types: `char` | `string` | `cell`

`opt` — `cuffnorm` options
`CuffNormOptions` object | string | character vector

`cuffnorm` options, specified as a `CuffNormOptions` object, string, or character vector. The string or character vector must be in the original `cuffnorm` option syntax (prefixed by one or two dashes) [1].

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `cuffnorm('gyrAB.gtf',["Myco_1_1.sam", "Myco_2_1.sam"],'NumThreads',5)`

`ExtraCommand` — Additional commands
`""` (default) | string | character vector

The commands must be in the native syntax (prefixed by one or two dashes). Use this option to apply undocumented flags and flags without corresponding MATLAB properties.

Example: `'ExtraCommand','--library-type fr-secondstrand'`

Data Types: `char` | `string`

`IncludeAll` — Flag to apply all available options
`false` (default) | `true`

The original (native) syntax is prefixed by one or two dashes. By default, the function converts only the specified options. If the value is `true`, the software converts all available options, with default values for unspecified options, to the original syntax.

Note If you set `IncludeAll` to `true`, the software translates all available properties, with default values for unspecified properties. The only exception is that when the default value of a property is `NaN`, `Inf`, `[]`, `''`, or `""`, then the software does not translate the corresponding property.

Example: `'IncludeAll',true`

Data Types: `logical`

`Labels` — Labels for samples
`[]` (default) | string | character vector | string vector | cell array of character vectors

Labels for samples, specified as a string, character vector, string vector, or cell array of character vectors. If you are providing labels, you must specify the same number of labels as input samples.

Example: `'Labels',["mutant1","mutant2"]`

Data Types: `char` | `string` | `cell`

`LibraryNormalizationMethod` — Method to normalize library size
`"geometric"` (default) | `"classic-fpkm"` | `"quartile"`

Method to normalize the library size, specified as one of the following options:

- `"geometric"` — The function scales the FPKM values by the median geometric mean of fragment counts across all libraries as described in [2].
- `"classic-fpkm"` — The function applies no scaling to the FPKM values or fragment counts.
- `"quartile"` — The function scales the FPKM values by the ratio of upper quartiles between fragment counts and the average value across all libraries.

Example: `'LibraryNormalizationMethod',"classic-fpkm"`

Data Types: `char` | `string`

`NormalizeCompatibleHits` — Flag to use only fragments compatible with reference transcript to calculate FPKM values
`true` (default) | `false`

Flag to use only fragments compatible with a reference transcript to calculate FPKM values, specified as `true` or `false`.

Example: `'NormalizeCompatibleHits',false`

Data Types: `logical`

`NormalizeTotalHits` — Flag to include all fragments to calculate FPKM values
`false` (default) | `true`

Flag to include all fragments to calculate FPKM values, specified as `true` or `false`. If the value is `true`, the function includes all fragments, including fragments without a compatible reference.

Example: `'NormalizeTotalHits',true`

Data Types: `logical`

`NumThreads` — Number of parallel threads to use
`1` (default) | positive integer

Number of parallel threads to use, specified as a positive integer. Threads are run on separate processors or cores. Increasing the number of threads generally improves the runtime significantly, but increases the memory footprint.

Example: `'NumThreads',4`

Data Types: `double`

`OutputDirectory` — Directory to store analysis results
current directory (`"./"`) (default) | string | character vector

Directory to store analysis results, specified as a string or character vector.

Example: `'OutputDirectory',"./AnalysisResults/"`

Data Types: `char` | `string`

`OutputFormat` — Format for result files
`"simple-table"` (default) | `"cuffdiff"`

Format for result files, specified as `"simple-table"` or `"cuffdiff"`.

- `"simple-table"` — The output is in tab-delimited table format.
- `"cuffdiff"` — The output is in the same form used by `cuffdiff`.

Example: `'OutputFormat',"cuffdiff"`

Data Types: `char` | `string`

`Seed` — Seed for random number generator
`0` (default) | nonnegative integer

Seed for the random number generator, specified as a nonnegative integer. Setting a seed value ensures the reproducibility of the analysis results.

Example: `'Seed',10`

Data Types: `double`

Output Arguments

`isoform` — Name of file containing normalized expression level for isoform
`"./isoforms.fpkm_table"`

Name of a file containing the normalized expression level for each isoform, returned as a string.

The output string also includes the directory information defined by `OutputDirectory`. The default is the current directory. If you set `OutputDirectory` to `"/local/tmp/"`, the output becomes `"/local/tmp/isoforms.fpkm_table"`.

gene — Name of file containing normalized expression level for gene
`"./genes.fpkm_table"`

Name of a file containing the normalized expression level for each gene, returned as a string.

The output string also includes the directory information defined by `OutputDirectory`. The default is the current directory. If you set `OutputDirectory` to `"/local/tmp/"`, the output becomes `"/local/tmp/genes.fpkm_table"`.

`tss` — Name of file containing normalized expression level for transcript start site
`"./tss_groups.fpkm_table"`

Name of a file containing the normalized expression level for each transcript start site (TSS), returned as a string.

The output string also includes the directory information defined by `OutputDirectory`. The default is the current directory. If you set `OutputDirectory` to `"/local/tmp/"`, the output becomes `"/local/tmp/tss_groups.fpkm_table"`.

cds — Name of file containing normalized expression level for coding sequence
`"./cds.fpkm_table"`

Name of a file containing the normalized expression level for each coding sequence, returned as a string.

The output string also includes the directory information defined by `OutputDirectory`. The default is the current directory. If you set `OutputDirectory` to `"/local/tmp/"`, the output becomes `"/local/tmp/cds.fpkm_table"`.

Version History

Introduced in R2019a

References

[1] Trapnell, Cole, Brian A Williams, Geo Pertea, Ali Mortazavi, Gordon Kwan, Marijke J van Baren, Steven L Salzberg, Barbara J Wold, and Lior Pachter. “Transcript Assembly and Quantification

by RNA-Seq Reveals Unannotated Transcripts and Isoform Switching during Cell Differentiation." *Nature Biotechnology* 28, no. 5 (May 2010): 511–15.

See Also
`cufflinks` | `CuffNormOptions`

Topics
"Bioinformatics Toolbox Software Support Packages"

External Websites
Cufflinks manual

CuffNormOptions

Option set for `cuffnorm`

Description

A `CuffNormOptions` object contains options for the `cuffnorm` function, which generates expression tables normalized for library size [1].

Creation

Syntax

```
cuffnormOpt = CuffNormOptions
cuffnormOpt = CuffNormOptions(Name,Value)
cuffnormOpt = CuffNormOptions(S)
```

Description

`cuffnormOpt = CuffNormOptions` creates a `CuffNormOptions` object with the default property values.

`CuffNormOptions` requires the Cufflinks Support Package for the Bioinformatics Toolbox. If the support package is not installed, then the function provides a download link. For details, see “Bioinformatics Toolbox Software Support Packages”.

`cuffnormOpt = CuffNormOptions(Name,Value)` sets the object properties on page 1-711 using one or more name-value pair arguments. Enclose each property name in quotes. For example, `cuffnormOpt = CuffNormOptions('NumThreads',8)` specifies to use eight parallel threads.

`cuffnormOpt = CuffNormOptions(S)` specifies optional parameters using the string or character vector `S`.

Input Arguments

`S` — `cuffmerge` options
string | character vector

`cuffmerge` options, specified as a string or character vector. `S` must be in the original `cuffmerge` option syntax (prefixed by one or two dashes).

Example: `'--seed 5'`

Properties

`ExtraCommand` — Additional commands

`""` (default) | string | character vector

The commands must be in the native syntax (prefixed by one or two dashes). Use this option to apply undocumented flags and flags without corresponding MATLAB properties.

When the software converts the original flags to MATLAB properties, it stores any unrecognized flags in this property.

Example: `'--library-type fr-secondstrand'`

Data Types: `char` | `string`

`IncludeAll` — Flag to use all object properties

`false` (default) | `true`

Flag to include all the object properties with the corresponding default values when converting to the original options syntax, specified as `true` or `false`. You can convert the properties to the original syntax prefixed by one or two dashes (such as `'-d 100 -e 80'`) by using `getCommand`. The default value `false` means that when you call `getCommand(optionsObject)`, it converts only the specified properties. If the value is `true`, `getCommand` converts all available properties, with default values for unspecified properties, to the original syntax.

Note If you set `IncludeAll` to `true`, the software translates all available properties, with default values for unspecified properties. The only exception is that when the default value of a property is `NaN`, `Inf`, `[]`, `''`, or `""`, then the software does not translate the corresponding property.

Example: `true`

Data Types: `logical`

`Labels` — Labels for samples

`[]` (default) | string | character vector | string vector | cell array of character vectors

Labels for samples, specified as a string, character vector, string vector, or cell array of character vectors. If you are providing labels, you must specify the same number of labels as input samples.

Example: `["mutant1","mutant2"]`

Data Types: `double` | `char` | `string` | `cell`

`LibraryNormalizationMethod` — Method to normalize library size

`"geometric"` (default) | `"classic-fpkm"` | `"quartile"`

Method to normalize the library size, specified as one of the following options:

- `"geometric"` — The function scales the FPKM values by the median geometric mean of fragment counts across all libraries as described in [2].
- `"classic-fpkm"` — The function applies no scaling to the FPKM values or fragment counts.
- `"quartile"` — The function scales the FPKM values by the ratio of upper quartiles between fragment counts and the average value across all libraries.

Example: `"classic-fpkm"`

Data Types: `char` | `string`

`NormalizeCompatibleHits` — Flag to use only fragments compatible with reference transcript to calculate FPKM values

`true` (default) | `false`

Flag to use only fragments compatible with a reference transcript to calculate FPKM values, specified as `true` or `false`.

Example: `false`

Data Types: `logical`

`NormalizeTotalHits` — Flag to include all fragments to calculate FPKM values

`false` (default) | `true`

Flag to include all fragments to calculate FPKM values, specified as `true` or `false`. If the value is `true`, the function includes all fragments, including fragments without a compatible reference.

Example: `true`

Data Types: `logical`

`NumThreads` — Number of parallel threads to use

`1` (default) | positive integer

Number of parallel threads to use, specified as a positive integer. Threads are run on separate processors or cores. Increasing the number of threads generally improves the runtime significantly, but increases the memory footprint.

Example: `4`

Data Types: `double`

`OutputDirectory` — Directory to store analysis results

current directory (`"./"`) (default) | string | character vector

Directory to store analysis results, specified as a string or character vector.

Example: `"./AnalysisResults/"`

Data Types: `char` | `string`

`OutputFormat` — Format for result files

`"simple-table"` (default) | `"cuffdiff"`

Format for result files, specified as `"simple-table"` or `"cuffdiff"`.

- `"simple-table"` — The output is in tab-delimited table format.
- `"cuffdiff"` — The output is in the same form used by `cuffdiff`.

Example: `"cuffdiff"`

Data Types: `char` | `string`

Seed — Seed for random number generator

`0` (default) | nonnegative integer

Seed for the random number generator, specified as a nonnegative integer. Setting a seed value ensures the reproducibility of the analysis results.

Example: `10`

Data Types: `double`

`Version` — Supported version

string

This property is read-only.

Supported version of the original cufflinks software, returned as a string.

Example: `"2.2.1"`

Data Types: `string`

Object Functions

getCommand	Translate object properties to original options syntax
getOptionsTable	Return table with all properties and equivalent options in original syntax

Examples

Create CuffNormOptions Object

Create a `CuffNormOptions` object with the default values.

```
opt = CuffNormOptions;
```

Create an object using name-value pairs.

```
opt2 = CuffNormOptions('OutputDirectory',"./norm",'Seed',20)
```

Create an object by using the original syntax.

```
opt3 = CuffNormOptions('--output-dir ./norm --seed 20')
```

Assemble Transcriptome and Normalize Expression Levels

Create a `CufflinksOptions` object to define cufflinks options, such as the number of parallel threads and the output directory to store the results.

```
cflOpt = CufflinksOptions;
cflOpt.NumThreads = 8;
cflOpt.OutputDirectory = "./cufflinksOut";
```

The SAM files provided for this example contain aligned reads for *Mycoplasma pneumoniae* from two samples with three replicates each. The reads are simulated 100bp-reads for two genes (`gyrA` and gyrB) located next to each other on the genome. All the reads are sorted by reference position, as required by cufflinks.

```
sams = ["Myco_1_1.sam","Myco_1_2.sam","Myco_1_3.sam",...
        "Myco_2_1.sam", "Myco_2_2.sam", "Myco_2_3.sam"];
```

Assemble the transcriptome from the aligned reads.

```
[gtfs,isofpkm,genes,skipped] = cufflinks(sams,cflOpt);
```

gtfs is a list of GTF files that contain assembled isoforms.

Compare the assembled isoforms using cuffcompare.

```
stats = cuffcompare(gtfs);
```

Merge the assembled transcripts using cuffmerge.

```
mergedGTF = cuffmerge(gtfs,'OutputDirectory','./cuffMergeOutput');
```

mergedGTF reports only one transcript. This is because the two genes of interest are located next to each other, and cuffmerge cannot distinguish two distinct genes. To guide cuffmerge, use a reference GTF (gyrAB.gtf) containing information about these two genes. If the file is not located in the same directory that you run cuffmerge from, you must also specify the file path.

```
gyrAB = which('gyrAB.gtf');
mergedGTF2 = cuffmerge(gtfs,'OutputDirectory','./cuffMergeOutput2',...
            'ReferenceGTF',gyrAB);
```

Calculate abundances (expression levels) from aligned reads for each sample.

```
abundances1 = cuffquant(mergedGTF2,["Myco_1_1.sam","Myco_1_2.sam","Myco_1_3.sam"],...
                        'OutputDirectory','./cuffquantOutput1');
abundances2 = cuffquant(mergedGTF2,["Myco_2_1.sam", "Myco_2_2.sam", "Myco_2_3.sam"],...
                        'OutputDirectory','./cuffquantOutput2');
```

Assess the significance of changes in expression for genes and transcripts between conditions by performing the differential testing using cuffdiff. The cuffdiff function operates in two distinct steps: the function first estimates abundances from aligned reads, and then performs the statistical analysis. In some cases (for example, distributing computing load across multiple workers), performing the two steps separately is desirable. After performing the first step with cuffquant, you can then use the binary CXB output file as an input to cuffdiff to perform statistical analysis. Because cuffdiff returns several files, specify the output directory is recommended.

```
isoformDiff = cuffdiff(mergedGTF2,[abundances1,abundances2],...
                      'OutputDirectory','./cuffdiffOutput');
```

Display a table containing the differential expression test results for the two genes gyrB and gyrA.

```
readtable(isoformDiff,'FileType','text')
```

```
ans =

  2×14 table

        test_id              gene_id          gene              locus                sample_1    sample_
    ________________    _____________    ______    _______________________    ________    _______
    'TCONS_00000001'    'XLOC_000001'    'gyrB'    'NC_000912.1:2868-7340'      'q1'         'q2'
    'TCONS_00000002'    'XLOC_000001'    'gyrA'    'NC_000912.1:2868-7340'      'q1'         'q2'
```

You can use `cuffnorm` to generate normalized expression tables for further analyses. `cuffnorm` results are useful when you have many samples and you want to cluster them or plot expression levels for genes that are important in your study. Note that you cannot perform differential expression analysis using `cuffnorm`.

Specify a cell array, where each element is a string vector containing file names for a single sample with replicates.

```
alignmentFiles = {["Myco_1_1.sam","Myco_1_2.sam","Myco_1_3.sam"],...
                  ["Myco_2_1.sam", "Myco_2_2.sam", "Myco_2_3.sam"]}
isoformNorm = cuffnorm(mergedGTF2, alignmentFiles,...
                      'OutputDirectory', './cuffnormOutput');
```

Display a table containing the normalized expression levels for each transcript.

```
readtable(isoformNorm,'FileType','text')
```

```
ans =

  2×7 table

      tracking_id          q1_0           q1_2           q1_1           q2_1           q2_0
    ________________    __________    __________    __________    __________    __________    __

    'TCONS_00000001'    1.0913e+05         78628    1.2132e+05    4.3639e+05    4.2228e+05    4.
    'TCONS_00000002'    3.5158e+05    3.7458e+05    3.4238e+05    1.0483e+05    1.1546e+05    1.
```

Column names starting with *q* have the format: *conditionX_N*, indicating that the column contains values for replicate *N* of *conditionX*.

Version History

Introduced in R2019a

References

[1] Trapnell, Cole, Brian A Williams, Geo Pertea, Ali Mortazavi, Gordon Kwan, Marijke J van Baren, Steven L Salzberg, Barbara J Wold, and Lior Pachter. "Transcript Assembly and Quantification by RNA-Seq Reveals Unannotated Transcripts and Isoform Switching during Cell Differentiation." *Nature Biotechnology* 28, no. 5 (May 2010): 511–15.

See Also

`CufflinksOptions` | `cuffcompare` | `cuffdiff` | `cuffmerge` | `cuffnorm` | `cuffquant` | `cuffgtf2sam` | `cuffgffread`

Topics
"Bioinformatics Toolbox Software Support Packages"

External Websites
Cufflinks manual

cuffquant

Quantify gene and transcript expression profiles

Syntax

```
cxbFile = cuffquant(transcriptsAnnot,alignmentFiles)
cxbFile = cuffquant(transcriptsAnnot,alignmentFiles,opt)
cxbFile = cuffquant(transcriptsAnnot,alignmentFiles,Name,Value)
```

Description

`cxbFile = cuffquant(transcriptsAnnot,alignmentFiles)` generates abundance estimates for the samples in `alignmentFiles` using the reference annotation file `transcriptsAnnot` [1]. You can use the generated CXB-format abundance (*.CXB) as input for `cuffdiff` to perform downstream differential expression analysis.

`cuffquant` requires the Cufflinks Support Package for the Bioinformatics Toolbox. If the support package is not installed, then the function provides a download link. For details, see “Bioinformatics Toolbox Software Support Packages”.

`cxbFile = cuffquant(transcriptsAnnot,alignmentFiles,opt)` uses additional options specified by `opt`.

`cxbFile = cuffquant(transcriptsAnnot,alignmentFiles,Name,Value)` uses additional options specified by one or more name-value pair arguments. For example, `cuffquant('gyrAB.gtf',["Myco_1_1.sam", "Myco_2_1.sam"],'NumThreads',5)` specifies to use five parallel threads.

Examples

Assemble Transcriptome and Perform Differential Expression Testing

Create a `CufflinksOptions` object to define cufflinks options, such as the number of parallel threads and the output directory to store the results.

```
cflOpt = CufflinksOptions;
cflOpt.NumThreads = 8;
cflOpt.OutputDirectory = "./cufflinksOut";
```

The SAM files provided for this example contain aligned reads for *Mycoplasma pneumoniae* from two samples with three replicates each. The reads are simulated 100bp-reads for two genes (`gyrA` and `gyrB`) located next to each other on the genome. All the reads are sorted by reference position, as required by `cufflinks`.

```
sams = ["Myco_1_1.sam","Myco_1_2.sam","Myco_1_3.sam",...
        "Myco_2_1.sam", "Myco_2_2.sam", "Myco_2_3.sam"];
```

Assemble the transcriptome from the aligned reads.

```
[gtfs,isofpkm,genes,skipped] = cufflinks(sams,cflOpt);
```

`gtfs` is a list of GTF files that contain assembled isoforms.

Compare the assembled isoforms using `cuffcompare`.

```
stats = cuffcompare(gtfs);
```

Merge the assembled transcripts using `cuffmerge`.

```
mergedGTF = cuffmerge(gtfs,'OutputDirectory','./cuffMergeOutput');
```

`mergedGTF` reports only one transcript. This is because the two genes of interest are located next to each other, and `cuffmerge` cannot distinguish two distinct genes. To guide `cuffmerge`, use a reference GTF (`gyrAB.gtf`) containing information about these two genes. If the file is not located in the same directory that you run `cuffmerge` from, you must also specify the file path.

```
gyrAB = which('gyrAB.gtf');
mergedGTF2 = cuffmerge(gtfs,'OutputDirectory','./cuffMergeOutput2',...
            'ReferenceGTF',gyrAB);
```

Calculate abundances (expression levels) from aligned reads for each sample.

```
abundances1 = cuffquant(mergedGTF2,["Myco_1_1.sam","Myco_1_2.sam","Myco_1_3.sam"],...
                        'OutputDirectory','./cuffquantOutput1');
abundances2 = cuffquant(mergedGTF2,["Myco_2_1.sam", "Myco_2_2.sam", "Myco_2_3.sam"],...
                        'OutputDirectory','./cuffquantOutput2');
```

Assess the significance of changes in expression for genes and transcripts between conditions by performing the differential testing using `cuffdiff`. The `cuffdiff` function operates in two distinct steps: the function first estimates abundances from aligned reads, and then performs the statistical analysis. In some cases (for example, distributing computing load across multiple workers), performing the two steps separately is desirable. After performing the first step with `cuffquant`, you can then use the binary CXB output file as an input to `cuffdiff` to perform statistical analysis. Because `cuffdiff` returns several files, specify the output directory is recommended.

```
isoformDiff = cuffdiff(mergedGTF2,[abundances1,abundances2],...
                      'OutputDirectory','./cuffdiffOutput');
```

Display a table containing the differential expression test results for the two genes `gyrB` and `gyrA`.

```
readtable(isoformDiff,'FileType','text')
```

```
ans =

  2×14 table

        test_id              gene_id          gene              locus               sample_1     sample_
    _________________    ______________    ______    ________________________    ________    _______

    'TCONS_00000001'    'XLOC_000001'    'gyrB'    'NC_000912.1:2868-7340'       'q1'          'q2'
    'TCONS_00000002'    'XLOC_000001'    'gyrA'    'NC_000912.1:2868-7340'       'q1'          'q2'
```

You can use `cuffnorm` to generate normalized expression tables for further analyses. `cuffnorm` results are useful when you have many samples and you want to cluster them or plot expression levels for genes that are important in your study. Note that you cannot perform differential expression analysis using `cuffnorm`.

Specify a cell array, where each element is a string vector containing file names for a single sample with replicates.

```
alignmentFiles = {["Myco_1_1.sam","Myco_1_2.sam","Myco_1_3.sam"],...
                  ["Myco_2_1.sam", "Myco_2_2.sam", "Myco_2_3.sam"]}
isoformNorm = cuffnorm(mergedGTF2, alignmentFiles,...
                       'OutputDirectory', './cuffnormOutput');
```

Display a table containing the normalized expression levels for each transcript.

```
readtable(isoformNorm,'FileType','text')
```

```
ans =

  2×7 table

      tracking_id          q1_0          q1_2          q1_1          q2_1          q2_0       
    _________________    __________    __________    __________    __________    __________    __

    'TCONS_00000001'     1.0913e+05         78628    1.2132e+05    4.3639e+05    4.2228e+05    4.
    'TCONS_00000002'     3.5158e+05    3.7458e+05    3.4238e+05    1.0483e+05    1.1546e+05    1.
```

Column names starting with *q* have the format: *conditionX_N*, indicating that the column contains values for replicate *N* of *conditionX*.

Input Arguments

`transcriptsAnnot` — Name of transcript annotation file

string | character vector

Name of the transcript annotation file, specified as a string or character vector. The file can be a GTF or GFF file produced by `cufflinks`, `cuffcompare`, or another source of GTF annotations.

Example: `"gyrAB.gtf"`

Data Types: `char` | `string`

`alignmentFiles` — Names of SAM, BAM, or CXB files

string vector | cell array

Names of SAM, BAM, or CXB files containing alignment records for each sample, specified as a string vector or cell array. If you use a cell array, each element must be a string vector or cell array of character vectors specifying alignment files for every replicate of the same sample.

Example: `["Myco_1_1.sam", "Myco_2_1.sam"]`

Data Types: `char` | `string` | `cell`

`opt` — `cuffquant` options

`CuffQuantOptions` object | string | character vector

`cuffquant` options, specified as a `CuffQuantOptions` object, string, or character vector. The string or character vector must be in the original `cuffquant` option syntax (prefixed by one or two dashes) [1].

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `cuffquant(transcripts,alignmentFiles,'NumThreads',4,'Seed',1)`

`EffectiveLengthCorrection` — Flag to normalize fragment counts
`true` (default) | false

Flag to normalize fragment counts to fragments per kilobase per million mapped reads (FPKM), specified as `true` or `false`.

Example: `'EffectiveLengthCorrection',false`

Data Types: `logical`

`ExtraCommand` — Additional commands
`""` (default) | string | character vector

The commands must be in the native syntax (prefixed by one or two dashes). Use this option to apply undocumented flags and flags without corresponding MATLAB properties.

Example: `'ExtraCommand','--library-type fr-secondstrand'`

Data Types: `char` | `string`

`FragmentBiasCorrection` — Name of FASTA file with reference transcripts to detect bias
string | character vector

Name of the FASTA file with reference transcripts to detect bias in fragment counts, specified as a string or character vector. Library preparation can introduce sequence-specific bias into RNA-Seq experiments. Providing reference transcripts improves the accuracy of the transcript abundance estimates.

Example: `'FragmentBiasCorrection',"bias.fasta"`

Data Types: `char` | `string`

`FragmentLengthMean` — Expected mean fragment length in base pairs
`200` (default) | positive integer

Expected mean fragment length, specified as a positive integer. The default value is `200` base pairs. The function can learn the fragment length mean for each SAM file. Using this option is not recommended for paired-end reads.

Example: `'FragmentLengthMean',100`

Data Types: `double`

`FragmentLengthSD` — Expected standard deviation for fragment length distribution
`80` (default) | positive scalar

Expected standard deviation for the fragment length distribution, specified as a positive scalar. The default value is `80` base pairs. The function can learn the fragment length standard deviation for each SAM file. Using this option is not recommended for paired-end reads.

Example: `'FragmentLengthSD',70`

Data Types: `double`

`IncludeAll` — Flag to apply all available options
`false` (default) | true

The original (native) syntax is prefixed by one or two dashes. By default, the function converts only the specified options. If the value is `true`, the software converts all available options, with default values for unspecified options, to the original syntax.

Note If you set `IncludeAll` to `true`, the software translates all available properties, with default values for unspecified properties. The only exception is that when the default value of a property is `NaN`, `Inf`, `[]`, `''`, or `""`, then the software does not translate the corresponding property.

Example: `'IncludeAll',true`

Data Types: `logical`

`LengthCorrection` — Flag to correct by transcript length
`true` (default) | `false`

Flag to correct by the transcript length, specified as `true` or `false`. Set this value to `false` only when the fragment count is independent of the feature size, such as for small RNA libraries with no fragmentation and for 3' end sequencing, where all fragments have the same length.

Example: `'LengthCorrection',false`

Data Types: `logical`

`MaskFile` — Name of GTF or GFF file containing transcripts to ignore
string | character vector

Name of the GTF or GFF file containing transcripts to ignore during analysis, specified as a string or character vector. Some examples of transcripts to ignore include annotated rRNA transcripts, mitochondrial transcripts, and other abundant transcripts. Ignoring these transcripts improves the robustness of the abundance estimates.

Example: `'MaskFile',"excludes.gtf"`

Data Types: `char` | `string`

`MaxBundleFrags` — Maximum number of fragments to include for each locus before skipping
`500000` (default) | positive integer

Maximum number of fragments to include for each locus before skipping new fragments, specified as a positive integer. Skipped fragments are marked with the status `HIDATA` in the file `skipped.gtf`.

Example: `'MaxBundleFrags',400000`

Data Types: `double`

`MaxFragAlignments` — Maximum number of aligned reads to include for each fragment
`Inf` (default) | positive integer

Maximum number of aligned reads to include for each fragment before skipping new reads, specified as a positive integer. `Inf`, the default value, sets no limit on the maximum number of aligned reads.

Example: `'MaxFragAlignments',1000`

Data Types: `double`

`MaxMLEIterations` — Maximum number of iterations for maximum likelihood estimation
`5000` (default) | positive integer

Maximum number of iterations for the maximum likelihood estimation of abundances, specified as a positive integer.

Example: `'MaxMLEIterations',4000`

Data Types: `double`

`MinAlignmentCount` — Minimum number of alignments required in locus for significance testing
`10` (default) | positive integer

Minimum number of alignments required in a locus to perform the significance testing for differences between samples, specified as a positive integer.

Example: `'MinAlignmentCount',8`

Data Types: `double`

`MultiReadCorrection` — Flag to improve abundance estimation using rescue method
`false` (default) | `true`

Flag to improve abundance estimation for reads mapped to multiple genomic positions using the rescue method, specified as `true` or `false`. If the value is `false`, the function divides multimapped reads uniformly to all mapped positions. If the value is `true`, the function uses additional information, including gene abundance estimation, inferred fragment length, and fragment bias, to improve transcript abundance estimation.

The rescue method is described in [2].

Example: `'MultiReadCorrection',true`

Data Types: `logical`

`NumThreads` — Number of parallel threads to use
`1` (default) | positive integer

Number of parallel threads to use, specified as a positive integer. Threads are run on separate processors or cores. Increasing the number of threads generally improves the runtime significantly, but increases the memory footprint.

Example: `'NumThreads',4`

Data Types: `double`

`OutputDirectory` — Directory to store analysis results
current directory (`"./"`) (default) | string | character vector

Directory to store analysis results, specified as a string or character vector.

Example: `"./AnalysisResults/"`

Data Types: `char` | `string`

`Seed` — Seed for random number generator
`0` (default) | nonnegative integer

Seed for the random number generator, specified as a nonnegative integer. Setting a seed value ensures the reproducibility of the analysis results.

Example: `10`

Data Types: `double`

Output Arguments

`cxbFile` — Name of abundances file
`"./abundances.cxb"`

Name of the abundances file, returned as a string.

The output string also includes the directory information defined by `OutputDirectory`. The default is the current directory. If you set `OutputDirectory` to `"/local/tmp/"`, the output becomes `"/local/tmp/abundances.cxb"`.

Version History

Introduced in R2019a

References

[1] Trapnell, Cole, Brian A Williams, Geo Pertea, Ali Mortazavi, Gordon Kwan, Marijke J van Baren, Steven L Salzberg, Barbara J Wold, and Lior Pachter. “Transcript Assembly and Quantification by RNA-Seq Reveals Unannotated Transcripts and Isoform Switching during Cell Differentiation.” *Nature Biotechnology* 28, no. 5 (May 2010): 511–15. https://doi.org/10.1038/nbt.1621.

[2] Mortazavi, Ali, Brian A Williams, Kenneth McCue, Lorian Schaeffer, and Barbara Wold. “Mapping and Quantifying Mammalian Transcriptomes by RNA-Seq.” *Nature Methods* 5, no. 7 (July 2008): 621–28. https://doi.org/10.1038/nmeth.1226.

See Also

`CuffQuantOptions` | `cufflinks` | `CufflinksOptions` | `cuffcompare` | `cuffdiff` | `cuffmerge` | `cuffnorm` | `cuffgffread` | `cuffgtf2sam`

Topics
“Bioinformatics Toolbox Software Support Packages”

External Websites
Cufflinks manual

CuffQuantOptions

Option set for `cuffquant`

Description

A `CuffQuantOptions` object contains options to run the `cuffquant` function, which quantifies gene and transcript expression data [1].

Creation

Syntax

```
cuffquantOpt = CuffQuantOptions
cuffquantOpt = CuffQuantOptions(Name,Value)
cuffquantOpt = CuffQuantOptions(S)
```

Description

`cuffquantOpt = CuffQuantOptions` creates a `CuffQuantOptions` object with default property values.

`CuffQuantOptions` requires the Cufflinks Support Package for the Bioinformatics Toolbox. If the support package is not installed, then the function provides a download link. For details, see "Bioinformatics Toolbox Software Support Packages".

`cuffquantOpt = CuffQuantOptions(Name,Value)` sets the object properties on page 1-724 using one or more name-value pair arguments. Enclose each property name in quotes. For example, `cuffquantOpt = CuffQuantOptions('NumThreads',8)` specifies to use eight parallel threads.

`cuffquantOpt = CuffQuantOptions(S)` specifies optional parameters using a string or character vector `S`.

Input Arguments

S — `Cuffquant` options
string | character vector

`Cuffquant` options, specified as a string or character vector. `S` must be in the `cuffquant` option syntax (prefixed by one or two dashes).

Example: `'--seed 5'`

Properties

`EffectiveLengthCorrection` — Flag to normalize fragment counts
`true` (default) | false

Flag to normalize fragment counts to fragments per kilobase per million mapped reads (FPKM), specified as `true` or `false`.

Example: `false`

Data Types: `logical`

`ExtraCommand` — Additional commands
`""` (default) | string | character vector

The commands must be in the native syntax (prefixed by one or two dashes). Use this option to apply undocumented flags and flags without corresponding MATLAB properties.

When the software converts the original flags to MATLAB properties, it stores any unrecognized flags in this property.

Example: `'--library-type fr-secondstrand'`

Data Types: `char` | `string`

`FragmentBiasCorrection` — Name of FASTA file with reference transcripts to detect bias
string | character vector

Name of the FASTA file with reference transcripts to detect bias in fragment counts, specified as a string or character vector. Library preparation can introduce sequence-specific bias into RNA-Seq experiments. Providing reference transcripts improves the accuracy of the transcript abundance estimates.

Example: `"bias.fasta"`

Data Types: `char` | `string`

`FragmentLengthMean` — Expected mean fragment length in base pairs
`200` (default) | positive integer

Expected mean fragment length, specified as a positive integer. The default value is `200` base pairs. The function can learn the fragment length mean for each SAM file. Using this option is not recommended for paired-end reads.

Example: `100`

Data Types: `double`

`FragmentLengthSD` — Expected standard deviation for fragment length distribution
`80` (default) | positive scalar

Expected standard deviation for the fragment length distribution, specified as a positive scalar. The default value is `80` base pairs. The function can learn the fragment length standard deviation for each SAM file. Using this option is not recommended for paired-end reads.

Example: `70`

Data Types: `double`

`IncludeAll` — Flag to use all object properties
`false` (default) | true

Flag to include all the object properties with the corresponding default values when converting to the original options syntax, specified as `true` or `false`. You can convert the properties to the original syntax prefixed by one or two dashes (such as `'-d 100 -e 80'`) by using `getCommand`. The default value `false` means that when you call `getCommand(optionsObject)`, it converts only the specified properties. If the value is `true`, `getCommand` converts all available properties, with default values for unspecified properties, to the original syntax.

Note If you set `IncludeAll` to `true`, the software translates all available properties, with default values for unspecified properties. The only exception is that when the default value of a property is `NaN`, `Inf`, `[]`, `''`, or `""`, then the software does not translate the corresponding property.

Example: `true`

Data Types: `logical`

`LengthCorrection` — Flag to correct by transcript length
`true` (default) | `false`

Flag to correct by the transcript length, specified as `true` or `false`. Set this value to `false` only when the fragment count is independent of the feature size, such as for small RNA libraries with no fragmentation and for 3' end sequencing, where all fragments have the same length.

Example: `false`

Data Types: `logical`

`MaskFile` — Name of GTF or GFF file containing transcripts to ignore
string | character vector

Name of the GTF or GFF file containing transcripts to ignore during analysis, specified as a string or character vector. Some examples of transcripts to ignore include annotated rRNA transcripts, mitochondrial transcripts, and other abundant transcripts. Ignoring these transcripts improves the robustness of the abundance estimates.

Example: `'excludes.gtf'`

Data Types: `char` | `string`

`MaxBundleFrags` — Maximum number of fragments to include for each locus before skipping
`500000` (default) | positive integer

Maximum number of fragments to include for each locus before skipping new fragments, specified as a positive integer. Skipped fragments are marked with the status `HIDATA` in the file `skipped.gtf`.

Example: `400000`

Data Types: `double`

`MaxFragAlignments` — Maximum number of aligned reads to include for each fragment
`Inf` (default) | positive integer

Maximum number of aligned reads to include for each fragment before skipping new reads, specified as a positive integer. `Inf`, the default value, sets no limit on the maximum number of aligned reads.

Example: `1000`

Data Types: `double`

`MaxMLEIterations` — Maximum number of iterations for maximum likelihood estimation
`5000` (default) | positive integer

Maximum number of iterations for the maximum likelihood estimation of abundances, specified as a positive integer.

Example: `4000`

Data Types: `double`

`MinAlignmentCount` — Minimum number of alignments in locus to perform significance testing
`10` (default) | positive integer

Minimum number of alignments required in a locus to perform the significance testing for differences between samples, specified as a positive integer.

Example: `8`

Data Types: `double`

`MultiReadCorrection` — Flag to improve abundance estimation using rescue method
`false` (default) | `true`

Flag to improve abundance estimation for reads mapped to multiple genomic positions using the rescue method, specified as `true` or `false`. If the value is `false`, the function divides multimapped reads uniformly to all mapped positions. If the value is `true`, the function uses additional information, including gene abundance estimation, inferred fragment length, and fragment bias, to improve transcript abundance estimation.

The rescue method is described in [2].

Example: `true`

Data Types: `logical`

`NumThreads` — Number of parallel threads to use
`1` (default) | positive integer

Number of parallel threads to use, specified as a positive integer. Threads are run on separate processors or cores. Increasing the number of threads generally improves the runtime significantly, but increases the memory footprint.

Example: `4`

Data Types: `double`

`OutputDirectory` — Directory to store analysis results
current directory (`"./"`) (default) | string | character vector

Directory to store analysis results, specified as a string or character vector.

Example: `"./AnalysisResults/"`

Data Types: `char` | `string`

`Seed` — Seed for random number generator
`0` (default) | nonnegative integer

Seed for the random number generator, specified as a nonnegative integer. Setting a seed value ensures the reproducibility of the analysis results.

Example: `10`

Data Types: `double`

`Version` — Supported version
string

This property is read-only.

Supported version of the original cufflinks software, returned as a string.

Example: `"2.2.1"`

Data Types: `string`

Object Functions

getCommand	Translate object properties to original options syntax
getOptionsTable	Return table with all properties and equivalent options in original syntax

Examples

Create CuffQuantOptions Object

Create a `CuffQuantOptions` object with the default values.

```
opt = CuffQuantOptions;
```

Create an object using name-value pairs.

```
opt2 = CuffQuantOptions('NumThreads',4,'MinAlignmentCount',50)
```

Create an object by using the original syntax.

```
opt3 = CuffQuantOptions('-p 4 --min-alignment-count 50')
```

Assemble Transcriptome and Perform Differential Expression Testing

Create a `CufflinksOptions` object to define cufflinks options, such as the number of parallel threads and the output directory to store the results.

```
cflOpt = CufflinksOptions;
cflOpt.NumThreads = 8;
cflOpt.OutputDirectory = "./cufflinksOut";
```

The SAM files provided for this example contain aligned reads for *Mycoplasma pneumoniae* from two samples with three replicates each. The reads are simulated 100bp-reads for two genes (`gyrA` and `gyrB`) located next to each other on the genome. All the reads are sorted by reference position, as required by `cufflinks`.

```
sams = ["Myco_1_1.sam","Myco_1_2.sam","Myco_1_3.sam",...
        "Myco_2_1.sam", "Myco_2_2.sam", "Myco_2_3.sam"];
```

Assemble the transcriptome from the aligned reads.

```
[gtfs,isofpkm,genes,skipped] = cufflinks(sams,cflOpt);
```

`gtfs` is a list of GTF files that contain assembled isoforms.

Compare the assembled isoforms using `cuffcompare`.

```
stats = cuffcompare(gtfs);
```

Merge the assembled transcripts using `cuffmerge`.

```
mergedGTF = cuffmerge(gtfs,'OutputDirectory','./cuffMergeOutput');
```

`mergedGTF` reports only one transcript. This is because the two genes of interest are located next to each other, and `cuffmerge` cannot distinguish two distinct genes. To guide `cuffmerge`, use a reference GTF (`gyrAB.gtf`) containing information about these two genes. If the file is not located in the same directory that you run `cuffmerge` from, you must also specify the file path.

```
gyrAB = which('gyrAB.gtf');
mergedGTF2 = cuffmerge(gtfs,'OutputDirectory','./cuffMergeOutput2',...
            'ReferenceGTF',gyrAB);
```

Calculate abundances (expression levels) from aligned reads for each sample.

```
abundances1 = cuffquant(mergedGTF2,["Myco_1_1.sam","Myco_1_2.sam","Myco_1_3.sam"],...
                        'OutputDirectory','./cuffquantOutput1');
abundances2 = cuffquant(mergedGTF2,["Myco_2_1.sam", "Myco_2_2.sam", "Myco_2_3.sam"],...
                        'OutputDirectory','./cuffquantOutput2');
```

Assess the significance of changes in expression for genes and transcripts between conditions by performing the differential testing using `cuffdiff`. The `cuffdiff` function operates in two distinct steps: the function first estimates abundances from aligned reads, and then performs the statistical analysis. In some cases (for example, distributing computing load across multiple workers), performing the two steps separately is desirable. After performing the first step with `cuffquant`, you can then use the binary CXB output file as an input to `cuffdiff` to perform statistical analysis. Because `cuffdiff` returns several files, specify the output directory is recommended.

```
isoformDiff = cuffdiff(mergedGTF2,[abundances1,abundances2],...
                      'OutputDirectory','./cuffdiffOutput');
```

Display a table containing the differential expression test results for the two genes `gyrB` and `gyrA`.

```
readtable(isoformDiff,'FileType','text')
```

```
ans =

  2×14 table

       test_id            gene_id         gene              locus              sample_1    sample_
    ________________    _____________    ______    ________________________    ________    _______

    'TCONS_00000001'    'XLOC_000001'    'gyrB'    'NC_000912.1:2868-7340'       'q1'        'q2'
    'TCONS_00000002'    'XLOC_000001'    'gyrA'    'NC_000912.1:2868-7340'       'q1'        'q2'
```

You can use `cuffnorm` to generate normalized expression tables for further analyses. `cuffnorm` results are useful when you have many samples and you want to cluster them or plot expression

levels for genes that are important in your study. Note that you cannot perform differential expression analysis using `cuffnorm`.

Specify a cell array, where each element is a string vector containing file names for a single sample with replicates.

```
alignmentFiles = {["Myco_1_1.sam","Myco_1_2.sam","Myco_1_3.sam"],...
                  ["Myco_2_1.sam", "Myco_2_2.sam", "Myco_2_3.sam"]}
isoformNorm = cuffnorm(mergedGTF2, alignmentFiles,...
                      'OutputDirectory', './cuffnormOutput');
```

Display a table containing the normalized expression levels for each transcript.

```
readtable(isoformNorm,'FileType','text')
```

```
ans =

  2×7 table

      tracking_id          q1_0          q1_2          q1_1          q2_1          q2_0

    ________________    __________    __________    __________    __________    __________    __

    'TCONS_00000001'    1.0913e+05         78628    1.2132e+05    4.3639e+05    4.2228e+05    4.
    'TCONS_00000002'    3.5158e+05    3.7458e+05    3.4238e+05    1.0483e+05    1.1546e+05    1.
```

Column names starting with *q* have the format: *conditionX_N*, indicating that the column contains values for replicate *N* of *conditionX*.

Version History

Introduced in R2019a

References

[1] Trapnell, Cole, Brian A Williams, Geo Pertea, Ali Mortazavi, Gordon Kwan, Marijke J van Baren, Steven L Salzberg, Barbara J Wold, and Lior Pachter. “Transcript Assembly and Quantification by RNA-Seq Reveals Unannotated Transcripts and Isoform Switching during Cell Differentiation.” *Nature Biotechnology* 28, no. 5 (May 2010): 511–15.

[2] Mortazavi, Ali, Brian A Williams, Kenneth McCue, Lorian Schaeffer, and Barbara Wold. “Mapping and Quantifying Mammalian Transcriptomes by RNA-Seq.” *Nature Methods* 5, no. 7 (July 2008): 621–28. https://doi.org/10.1038/nmeth.1226.

See Also

`CufflinksOptions` | `cuffcompare` | `cuffdiff` | `cuffmerge` | `cuffnorm` | `cuffquant` | `cuffgffread` | `cuffgtf2sam`

Topics

“Bioinformatics Toolbox Software Support Packages”

External Websites

Cufflinks manual

cytobandread

Read cytogenetic banding information

Syntax

`CytoStruct = cytobandread(File)`

Input Arguments

File	Character vector or string specifying a file containing cytogenetic G-banding data, such as an NCBI ideogram text file or a UCSC Genome Browser cytoband text file.

Output Arguments

CytoStruct	Structure containing cytogenetic G-banding data in the following fields: • `ChromLabels` • `BandStartBPs` • `BandEndBPs` • `BandLabels` • `GieStains`

Description

`CytoStruct = cytobandread(File)` reads *`File`*, which is a character vector or string specifying a file containing cytogenetic G-banding data, and returns *`CytoStruct`*, which is a structure containing the following fields.

Field	Description
`ChromLabels`	Cell array containing the chromosome label (number or letter) on which each band is located.
`BandStartBPs`	Column vector containing the number of the base pair at the start of each band.
`BandEndBPs`	Column vector containing the number of the base pair at the end of each band.
`BandLabels`	Cell array containing the FISH label of each band, for example, `p32.3`.

Field	Description
GieStains	Cell array containing the Giemsa staining result for each band. Possible stain results depend on the species. For example, for *Homo sapiens*, the possibilities are: • gneg • gpos25 • gpos50 • gpos75 • gpos100 • acen • stalk • gvar

Tip You can download files containing cytogenetic G-banding data from the NCBI or UCSC Genome Browser web site. For example, you can download the cytogenetic banding data (cytoBandIdeo.txt.gz) for *Homo sapiens* from:

https://hgdownload.cse.ucsc.edu/goldenPath/hg19/database/

Examples

Read cytogenetic banding information from a file

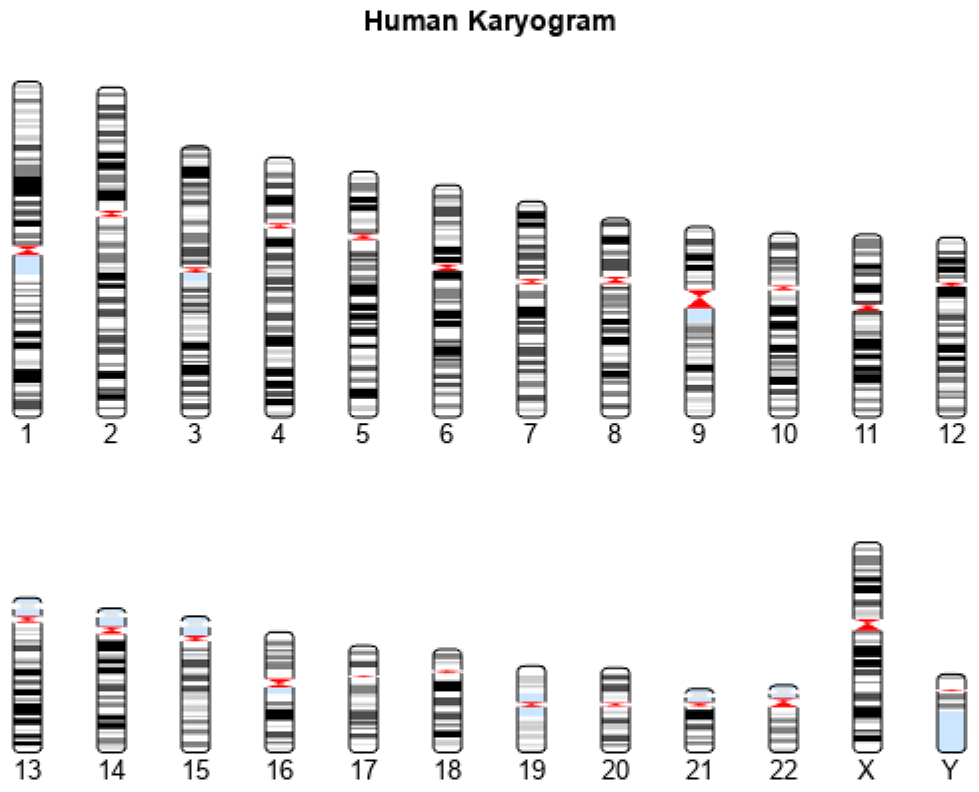
Read the cytogenetic banding information for *Homo sapiens* into a structure.

```
bands = cytobandread('hs_cytoBand.txt')
```

```
bands = struct with fields:
     ChromLabels: {862x1 cell}
    BandStartBPs: [862x1 int32]
      BandEndBPs: [862x1 int32]
      BandLabels: {862x1 cell}
       GieStains: {862x1 cell}
```

Plot the entire chromosome ideogram.

```
chromosomeplot(bands)
title('Human Karyogram')
```



Version History

Introduced in R2007b

See Also

`chromosomeplot`

DataMatrix object

Data structure encapsulating data and metadata from microarray experiment so that it can be indexed by gene or probe identifiers and by sample identifiers

Description

A DataMatrix object is a data structure encapsulating measurement data and feature metadata from a microarray experiment so that it can be indexed by gene or probe identifiers and by sample identifiers. A DataMatrix object stores experimental data in a matrix, with rows typically corresponding to gene names or probe identifiers, and columns typically corresponding to sample identifiers. A DataMatrix object also stores metadata, such as the gene names or probe identifiers and sample identifiers, in row names and column names.

You create a DataMatrix object using the object constructor function `DataMatrix`.

Property Summary

Properties of a DataMatrix Object

Property	Description
`Name`	Character vector that describes the DataMatrix object. Default is `''`.
`RowNames`	Empty array or cell array of character vectors that specifies the names for the rows, typically gene names or probe identifiers. The number of elements in the cell array must equal the number of rows in the matrix. Default is an empty array.
`ColNames`	Empty array or cell array of character vectors that specifies the names for the columns, typically sample identifiers. The number of elements in the cell array must equal the number of columns in the matrix.
`NRows`	Read-only. Positive number that specifies the number of rows in the matrix. **Note** You cannot modify this property directly. You can access it using the `get` method.
`NCols`	Read-only. Positive number that specifies the number of columns in the matrix. **Note** You cannot modify this property directly. You can access it using the `get` method.
`NDims`	Read-only. Positive number that specifies the number of dimensions in the matrix. **Note** You cannot modify this property directly. You can access it using the `get` method.
`ElementClass`	Read-only. Character vector that specifies the class type of the elements in the DataMatrix object, such as `single` or `double`. **Note** You cannot modify this property directly. You can access it using the `get` method.

Method Summary

General Methods of a DataMatrix Object

Method	Description
`colnames`	Retrieve or set column names of DataMatrix object.
`disp`	Display DataMatrix object.
`dmwrite`	Write DataMatrix object to text file.
`double`	Convert DataMatrix object to double-precision array.
`get`	Retrieve information about DataMatrix object.
`isempty`	Determine if DataMatrix object is empty.
`isfinite`	Determine if DataMatrix object elements are finite.
`isinf`	Determine if DataMatrix object elements are infinite.
`isnan`	Determine if DataMatrix object elements are NaN.
`isscalar`	Determine if DataMatrix object is scalar.
`isequal`	Test DataMatrix objects for equality.
`isequaln`	Test DataMatrix objects for equality, treating NaNs as equal.
`isvector`	Determine if DataMatrix object is vector.
`length`	Return length of DataMatrix object.
`ndims`	Return number of dimensions in DataMatrix object.
`numel`	Return number of elements in DataMatrix object.
`plot`	Draw 2-D line plot of DataMatrix object.
`rownames`	Retrieve or set row names of DataMatrix object.
`set`	Set property of DataMatrix object.
`single`	Convert DataMatrix object to single-precision array.
`size`	Return size of DataMatrix object.

Methods for Manipulating the Data in a DataMatrix Object

Method	Description
`cat`	Concatenate DataMatrix objects. The `horzcat` and `vertcat` methods implement special cases.
`horzcat`	Concatenate DataMatrix objects horizontally.
`sortcols`	Sort columns of DataMatrix object in ascending or descending order.
`sortrows`	Sort rows of DataMatrix object in ascending or descending order.
`subsasgn`	Subscripted assignment for DataMatrix object. To invoke this method, use parentheses or dot indexing described in "Accessing Data in DataMatrix Objects".
`subsref`	Subscripted reference for DataMatrix object. To invoke this method, use parentheses or dot indexing described in "Accessing Data in DataMatrix Objects".
`transpose`	Transpose DataMatrix object.
`vertcat`	Concatenate DataMatrix objects vertically.

Descriptive Statistics and Statistical Learning Methods

Method	Description
`kmeans`	K-means clustering.
`max`	Return maximum values in DataMatrix object.
`mean`	Return average or mean values in DataMatrix object.
`median`	Return median values in DataMatrix object.
`min`	Return minimum values in DataMatrix object.
`nanmax`	Return maximum values in DataMatrix object ignoring NaN values.
`nanmean`	Return average or mean values in DataMatrix object ignoring NaN values.
`nanmedian`	Return median values in DataMatrix object ignoring NaN values.
`nanmin`	Return minimum values in DataMatrix object ignoring NaN values.
`nanstd`	Return standard deviation values in DataMatrix object ignoring NaN values.
`nansum`	Return sum of elements in DataMatrix object ignoring NaN values.
`nanvar`	Return variance values in DataMatrix object ignoring NaN values.
`pca`	Principal component analysis on data.
`pdist`	Pairwise distance.
`std`	Return standard deviation values in DataMatrix object.
`sum`	Return sum of elements in DataMatrix object.
`var`	Return variance values in DataMatrix object.

Unary Methods — Exponential

Method	Description
`exp`	Exponential.
`log`	Natural logarithm.
`log10`	Common (base 10) logarithm.
`log2`	Base 2 logarithm and dissect floating-point numbers into exponent and mantissa.
`pow2`	Base 2 power and scale floating-point numbers.
`sqrt`	Square root.

Unary Methods — Integer

Method	Description
`ceil`	Round DataMatrix object toward infinity.
`fix`	Round DataMatrix object toward zero.
`floor`	Round DataMatrix object toward minus infinity.
`round`	Round DataMatrix object to nearest integer.

Unary Methods — Custom

Method	Description
`dmarrayfun`	Apply function to each element in DataMatrix object.

Binary Methods — Arithmetic Operator

Operator	Method	Description
`+`	`plus`	Add DataMatrix objects
`-`	`minus`	Subtract DataMatrix objects.
`.*`	`times`	Multiply DataMatrix objects.
`./`	`rdivide`	Right array divide DataMatrix objects.
`.\`	`ldivide`	Left array divide DataMatrix objects.
`.^`	`power`	Array power DataMatrix objects.

Binary Methods — Relational Operator

Operator	Method	Description
`<`	`lt`	Test DataMatrix objects for less than.
`<=`	`le`	Test DataMatrix objects for less than or equal to.
`>`	`gt`	Test DataMatrix objects for greater than.
`>=`	`ge`	Test DataMatrix objects for greater than or equal to.
`==`	`eq`	Test DataMatrix objects for equality.
`~=`	`ne`	Test DataMatrix objects for inequality.

Binary Methods — Custom

Method	Description
`dmbsxfun`	Apply element-by-element binary operation to two DataMatrix objects with singleton expansion enabled.

Examples

Example 1.3. Determining Properties and Property Values of a DataMatrix Object

You can display all properties and their current values of a DataMatrix object, *`DMobj`*, by using the following syntax:

```
get(DMobj)
```

You can return all properties and their current values of *`DMobj`*, a DataMatrix object, to *`DMstruct`*, a scalar structure in which each field name is a property of a DataMatrix object, and each field contains the value of that property, by using the following syntax:

```
DMstruct = get(DMobj)
```

You can return the value of a specific property of a DataMatrix object, *`DMobj`*, by using either of the following syntaxes:

```
PropertyValue = get(DMObj, 'PropertyName')
```

```
PropertyValue = DMObj.PropertyName
```

You can return the value of specific properties of a DataMatrix object, *`DMobj`*, by using the following syntax:

```
[Property1Value, Property2Value, ...] = get(DMobj, ...
'Property1Name', 'Property2Name', ...)
```

Example 1.4. Determining Possible Values of DataMatrix Object Properties

You can display possible values for all properties that have a fixed set of property values in a DataMatrix object, *`DMobj`*, by using the following syntax:

```
set(DMobj)
```

You can display possible values for a specific property that has a fixed set of property values in a DataMatrix object, *`DMobj`*, by using the following syntax:

```
set(DMObj,  'PropertyName')
```

Example 1.5. Specifying Properties of a DataMatrix Object

You can set a specific property of a DataMatrix object, *`DMObj`*, by using either of the following syntaxes:

```
DMObj = set(DMObj,  'PropertyName', PropertyValue)
```

```
DMObj.PropertyName = PropertyValue
```

You can set multiple properties of a DataMatrix object, *`DMobj`*, by using the following syntax:

```
set(DMobj,  'PropertyName1', PropertyValue1, ...
    'PropertyName2', PropertyValue2, ...)
```

Note For more examples of creating and using DataMatrix objects, see “Representing Expression Data Values in DataMatrix Objects”.

Version History

Introduced in R2008b

R2017b: `princomp` method has been renamed

Errors starting in R2017b

The `princomp` method of `DataMatrix object` has been renamed. Replace instances of `princomp` with `pca`.

See Also

`DataMatrix` | `colnames` | `disp` | `dmarrayfun` | `dmbsxfun` | `dmwrite` | `double` | `eq` | `ge` | `get` | `gt` | `horzcat` | `isequal` | `isequaln` | `ldivide` | `le` | `lt` | `max` | `mean` | `median` | `min` | `minus` | `ndims` | `ne` | `numel` | `plot` | `plus` | `power` | `rdivide` | `rownames` | `set` | `single` | `sortcols` | `sortrows` | `std` | `sum` | `times` | `var` | `vertcat`

DataMatrix

Create DataMatrix object

Syntax

```
DMobj = DataMatrix(Matrix)
DMobj = DataMatrix(Matrix, RowNames, ColumnNames)
DMobj = DataMatrix('File', FileName)

DMobj = DataMatrix(..., 'RowNames', RowNamesValue, ...)
DMobj = DataMatrix(..., 'ColNames', ColNamesValue, ...)
DMobj = DataMatrix(..., 'Name', NameValue, ...)
DMobj = DataMatrix('File', FileName, ...'Delimiter', DelimiterValue, ...)
DMobj = DataMatrix('File', FileName, ...'HLine', HLineValue, ...)
DMobj = DataMatrix('File', FileName, ...'Rows', RowsValue, ...)
DMobj = DataMatrix('File', FileName, ...'Columns', ColumnsValue, ...)
```

Arguments

Matrix	Two-dimensional numeric or logical array.
RowNames	Row names for the DataMatrix object, specified by a numeric vector, character array, string vector, or cell array of character vectors, whose elements are equal in number to the number of rows in *Matrix*. *RowNames* are typically gene names or probe identifiers from a microarray experiment. **Note** The row names do not need to be unique.
ColumnNames	Column names for the DataMatrix object, specified by a numeric vector, character array, string vector, or cell array of character vectors, whose elements are equal in number to the number of columns in *Matrix*. *ColumnNames* are typically sample identifiers from a microarray experiment. **Note** The column names do not need to be unique.
FileName	Character vector or string specifying a file name or a path and file name of a tab-delimited TXT or XLS file that contains table-oriented data and metadata. **Note** Typically, the first row of the table contains column names, the first column contains row names, and the numeric data starts at the `2,2` position. The `DataMatrix` function will detect if the first column does not contain row names, and read data from the first column. However, if the first row does not contain header text (column names), set the `HLine` property to `0`.

`RowNamesValue`, *`ColNamesValue`*	Row names or column names for the DataMatrix object. Choices are: • Numeric vector, character array, string vector, or a cell array of character vectors, whose elements are equal in number to the number of rows or number of columns of numeric data in the input matrix. • A character vector or string, which is used as a prefix for row or column names. Numbers will be appended to the prefix. • `true` — Unique row or column names will be assigned using the formats `row1`, `row2`, `row3`, etc., or `col1`, `col2`, `col3`, etc. • `false` — Default. No row or column names are assigned. **Note** The row or column names do not need to be unique.
`NameValue`	Character vector or string specifying a name for the DataMatrix object. Default is `''`.
`DelimiterValue`	Character vector or string specifying a delimiter symbol to use for the input file. Typical choices are: • `' '` • `'\t'` (default) • `','` • `';'` • `'|'`
`HLineValue`	Positive integer that specifies which row of the input file contains the column header text (column names). Default is `1`. When creating the DataMatrix object *`DMobj`*, the `DataMatrix` function loads data from (*`HLineValue`* `+ 1`) to the end of the file. **Tip** If the input file does not contain column header text (column names), set *`HLineValue`* to `0`.
`RowsValue`, *`ColumnsValue`*	A subset of rows or columns in `File`, for the `DataMatrix` function to use to create the DataMatrix object. Choices are: • Cell array of character vectors • Character array • String vector • Numeric or logical vector

Description

A DataMatrix object encapsulates measurement data and feature metadata from a microarray experiment so that it can be indexed by gene names or probe identifiers and by sample identifiers. For examples of creating and using DataMatrix objects, see “Representing Expression Data Values in DataMatrix Objects”.

Note The DataMatrix constructor function is part of the microarray object package. To make it available, type the following in the MATLAB command line:

```
import bioma.data.*
```

Otherwise, use `bioma.data.DataMatrix` instead of `DataMatrix`, in the following syntaxes.

`DMobj` `= DataMatrix(`*`Matrix`*`)` creates a DataMatrix object, *`DMobj`*, from *`Matrix`*, a two-dimensional numeric or logical array. *`Matrix`* can also be a DataMatrix object.

`DMobj` `= DataMatrix(`*`Matrix`*`,` *`RowNames`*`,` *`ColumnNames`*`)` creates a DataMatrix object, *`DMobj`*, from *`Matrix`*, a two-dimensional numeric or logical array, with row names and column names specified by *`RowNames`* and *`ColumnNames`*. *`RowNames`* and *`ColumnNames`* can be a numeric vector, character array, string vector, or cell array of character vectors, whose elements are equal in number to the number of rows and number of columns, respectively, in *`Matrix`*. *`RowNames`* are typically gene names or probe identifiers, while *`ColumnNames`* are typically sample identifiers.

Note The row or column names do not need to be unique.

`DMobj` `= DataMatrix('File',` *`FileName`*`)` creates a DataMatrix object, *`DMobj`*, from *`FileName`*, a character vector or string specifying a file name or a path and file name of a tab-delimited TXT or XLS file that contains table-oriented data and metadata.

Note Typically, the first row of the table contains column names, the first column contains row names, and the numeric data starts at the `2,2` position. The `DataMatrix` function will detect if the first column does not contain row names, and read data from the first column. However, if the first row does not contain header text (column names), set the `HLine` property to `0`.

`DMobj` `= DataMatrix(...,` *`'PropertyName'`*`,` *`PropertyValue`*`, ...)` calls `DataMatrix` with optional properties that use property name/property value pairs. You can specify one or more properties in any order. Each *`PropertyName`* must be enclosed in single quotation marks and is case insensitive. These property name/property value pairs are as follows:

`DMobj` `= DataMatrix(..., 'RowNames',` *`RowNamesValue`*`, ...)` specifies row names for *`DMobj`*. *`RowNamesValue`* can be any of the following:

- Numeric vector, character array, string vector, or a cell array of character vectors, whose elements are equal in number to the number of rows of numeric data in the input matrix.
- A character vector or string, which is used as a prefix for row names. Row numbers will be appended to the prefix.
- `true` — Unique row names will be assigned using the format `row1`, `row2`, `row3`, etc.
- `false` — Default. No row names are assigned.

Note The row names do not need to be unique.

`DMobj` `= DataMatrix(..., 'ColNames',` *`ColNamesValue`*`, ...)` specifies column names for *`DMobj`*. *`ColNamesValue`* can be any of the following:

- Numeric vector, character array, string vector, or a cell array of character vectors, whose elements are equal in number to the number of columns of numeric data in the input matrix.
- A character vector or string, which is used as a prefix for column names. Column numbers will be appended to the prefix.
- `true` — Unique column names will be assigned using the format `col1`, `col2`, `col3`, etc.
- `false` — Default. No column names are assigned.

Note The column names do not need to be unique.

`*DMobj* = DataMatrix(..., 'Name', *NameValue*, ...)` specifies a name for *`DMobj`*. Default is `''`.

`*DMobj* = DataMatrix('File', *FileName*, ...'Delimiter', *DelimiterValue*, ...)` specifies a delimiter symbol to use for the input file. Typical choices are:

- `' '`
- `'\t'` (default)
- `','`
- `';'`
- `'|'`

`*DMobj* = DataMatrix('File', *FileName*, ...'HLine', *HLineValue*, ...)` specifies which row of the input file contains the column header text (column names). *`HLineValue`* is a positive integer. Default is `1`. When creating the DataMatrix object *`DMobj`*, the `DataMatrix` function loads data from (*`HLineValue`* `+ 1`) to the end of the file.

Tip If the input file does not contain column header text (column names), set *`HLineValue`* to `0`.

`*DMobj* = DataMatrix('File', *FileName*, ...'Rows', *RowsValue*, ...)` specifies a subset of row names in `File` for the `DataMatrix` function to use to create *`DMobj`*. *`RowsValue`* can be a cell array of character vectors, a character array, string vector, or a numeric or logical vector.

`*DMobj* = DataMatrix('File', *FileName*, ...'Columns', *ColumnsValue*, ...)` specifies a subset of column names in `File` for the `DataMatrix` function to use to create *`DMobj`*. *`ColumnsValue`* can be a cell array of character vectors, string vector, character array, or numeric or logical vector.

Examples

For examples of creating and using DataMatrix objects, see “Representing Expression Data Values in DataMatrix Objects”.

Version History

Introduced in R2008b

See Also

`colnames` | `disp` | `dmarrayfun` | `dmbsxfun` | `dmwrite` | `double` | `eq` | `ge` | `get` | `gt` | `horzcat` | `isequal` | `isequaln` | `ldivide` | `le` | `lt` | `max` | `mean` | `median` | `min` | `minus` | `ndims` | `ne` | `numel` | `plot` | `plus` | `power` | `rdivide` | `rownames` | `set` | `single` | `sortcols` | `sortrows` | `std` | `sum` | `times` | `var` | `vertcat`

Topics

DataMatrix object on page 1-734

dayhoff

Return Dayhoff scoring matrix

Syntax

`ScoringMatrix = dayhoff`

Description

`ScoringMatrix = dayhoff` returns a `PAM250` type scoring matrix. The order of amino acids in the matrix is `A R N D C Q E G H I L K M F P S T W Y V B Z X *`.

Version History

Introduced before R2006a

See Also

`blosum` | `gonnet` | `localalign` | `nuc44` | `nwalign` | `pam` | `swalign`

dimercount

Count dimers in nucleotide sequence

Syntax

```
Dimers = dimercount(SeqNT)
[Dimers, Percent] = dimercount(SeqNT)

... = dimercount(SeqNT, 'Ambiguous', AmbiguousValue)
... = dimercount(SeqNT, 'Chart', ChartValue)
```

Input Arguments

SeqNT	One of the following: • Character vector or string specifying a nucleotide sequence. For valid letter codes, see the table Mapping Nucleotide Letter Codes to Integers. • Row vector of integers specifying a nucleotide sequence. For valid integers, see the table Mapping Nucleotide Integers to Letter Codes. • MATLAB structure containing a `Sequence` field that contains a nucleotide sequence, such as returned by `fastaread`, `fastqread`, `emblread`, `getembl`, `genbankread`, or `getgenbank`. Examples: `'ACGT'` or `[1 2 3 4]`
AmbiguousValue	Character vector or string specifying how to treat dimers containing ambiguous nucleotide characters (`R`, `Y`, `K`, `M`, `S`, `W`, `B`, `D`, `H`, `V`, or `N`). Choices are: • `'ignore'` (default) — Skips dimers containing ambiguous characters • `'bundle'` — Counts dimers containing ambiguous characters and reports the total count in the `Ambiguous` field of the *`Dimers`* output structure. • `'prorate'` — Counts dimers containing ambiguous characters and distributes them proportionately in the appropriate dimer fields containing standard nucleotide characters. For example, the counts for the dimer `AR` are distributed evenly between the `AA` and `AG` fields. • `'warn'` — Skips dimers containing ambiguous characters and displays a warning.
ChartValue	Character vector or string specifying a chart type. Choices are `'pie'` or `'bar'`.

Output Arguments

Dimers	MATLAB structure containing the fields AA, AC, AG, AT, CA, CC, CG, CT, GA, GC, GG, GT, TA, TC, TG, and TT, which contain the dimer counts in *SeqNT*.
Percent	A 4-by-4 matrix with the relative proportions of the dimers in *SeqNT*. The rows correspond to A, C, G, and T in the first element of the dimer, and the columns correspond to A, C, G, and T in the second element of the dimer.

Description

Dimers = dimercount(*SeqNT*) counts the nucleotide dimers in *SeqNT*, a nucleotide sequence, and returns the dimer counts in *Dimers*, a MATLAB structure containing the fields AA, AC, AG, AT, CA, CC, CG, CT, GA, GC, GG, GT, TA, TC, TG, and TT.

- For sequences that have dimers with the character U, these dimers are added to the corresponding dimers containing a T.
- If the sequence contains gaps indicated by a hyphen (-), the gaps are ignored, and the two characters on either side of the gap are counted as a dimer.
- If the sequence contains unrecognized characters, then dimers containing these characters are ignored, and the following warning message appears:

  ```
  Warning: Unknown symbols appear in the sequence. These will be ignored.
  ```

[*Dimers*, *Percent*] = dimercount(*SeqNT*) returns *Percent*, a 4-by-4 matrix with the relative proportions of the dimers in *SeqNT*. The rows correspond to A, C, G, and T in the first element of the dimer, and the columns correspond to A, C, G, and T in the second element of the dimer.

... = dimercount(*SeqNT*, 'Ambiguous', *AmbiguousValue*) specifies how to treat dimers containing ambiguous nucleotide characters. Choices are:

- 'ignore' (default)
- 'bundle'
- 'prorate'
- 'warn'

... = dimercount(*SeqNT*, 'Chart', *ChartValue*) creates a chart showing the relative proportions of the dimers. *ChartValue* can be 'pie' or 'bar'.

Examples

Count dimers in a nucleotide sequence

```
seq = randseq(100)
```

```
seq = 
'TTATGACGTTATTCTACTTTGATTGTGCGAGACAATGCTACCTTACCGGTCGGAACTCGATCGGTTGAACTCTATCACGCCTGGTCTTCGAAGTT
```

```
[Dimers, Percent] = dimercount(seq)
```

```
Dimers = struct with fields:
    AA: 4
```

```
    AC: 9
    AG: 3
    AT: 6
    CA: 3
    CC: 3
    CG: 8
    CT: 9
    GA: 8
    GC: 4
    GG: 4
    GT: 6
    TA: 7
    TC: 8
    TG: 7
    TT: 10

Percent = 4×4

    0.0404    0.0909    0.0303    0.0606
    0.0303    0.0303    0.0808    0.0909
    0.0808    0.0404    0.0404    0.0606
    0.0707    0.0808    0.0707    0.1010
```

Version History

Introduced before R2006a

See Also

aacount | basecount | baselookup | codoncount | nmercount | ntdensity

disp (DataMatrix)

Display DataMatrix object

Syntax

`disp(DMObj)`

Arguments

DMObj	DataMatrix object, such as created by `DataMatrix` (object constructor).

Description

`disp(DMObj)` displays the DataMatrix object *DMObj*, including row names and column names, without printing the DataMatrix object name.

Version History

Introduced in R2008b

See Also

`DataMatrix`

Topics

DataMatrix object on page 1-734

dmarrayfun (DataMatrix)

Apply function to each element in DataMatrix object

Syntax

```
DMObjNew1 = dmarrayfun(Func, DMObj1)
DMObjNew1 = dmarrayfun(Func, DMObj1, DMObj2, ...)
[DMObjNew1, DMObjNew2, ...] = dmarrayfun(Func, DMObj1, ...)

[DMObjNew1, ...] = dmarrayfun(Func, DMObj1, ...'UniformOutput',
UniformOutputValue, ...)
[DMObjNew1, ...] = dmarrayfun(Func, DMObj1, ...'DataMatrixOutput',
DataMatrixOutputValue, ...)
[DMObjNew1, ...] = dmarrayfun(Func, DMObj1, ...'Rows', RowsValue, ...)
[DMObjNew1, ...] = dmarrayfun(Func, DMObj1, ...'Columns', ColumnsValue, ...)
[DMObjNew1, ...] = dmarrayfun(Func, DMObj1, ...'ErrorHandler',
ErrorFuncHandle, ...)
```

Input Arguments

Func	Function handle for a function that returns one or more scalars, and returns values of the same class each time it is called.
DMObj1	DataMatrix object, such as created by `DataMatrix` (object constructor).
DMObj2	Either of the following: • DataMatrix object, such as created by `DataMatrix` (object constructor) • MATLAB numeric array **Note** *DMObj2* and subsequent input objects or arrays must be the same size (number of rows and columns) as *DMObj1*.
UniformOutputValue	Specifies whether *Func* must return output values without encapsulation in a cell array. Choices are `true` (default) or `false`. If `true`, `dmarrayfun` must return scalar values that can be concatenated into an array. These values can also be a cell array. If `false`, `dmarrayfun` returns a cell array (or multiple cell arrays), where the `I`,`J`th cell contains the value equal to *Func*(*DMObj1*(`I,J`),`...`).
DataMatrixOutputValue	Specifies whether return values must be DataMatrix objects. Choices are `true` (default) or `false`. If you set the `'UniformOutput'` property to `false`, this property is ignored.

RowsValue, *ColumnsValue*	Specifies the rows or columns to which to apply the function. Choices are: • Positive integer • Vector of positive integers • Character vector specifying a row or column name • Cell array of character vectors • Logical vector
ErrorFuncHandle	Specifies a function handle to a function that `dmarrayfun` calls if the call to *Func* fails.

Output Arguments

DMObjNew1, *DMObjNew2*	DataMatrix objects created from applying the function to each element in one or more DataMatrix objects. The size (number of rows and columns), row names, and column names will be the same as *DMObj1*.

Description

DMObjNew1 `= dmarrayfun(`*Func*`,` *DMObj1*`)` applies the function specified by *Func* to each element in *DMObj1*, a DataMatrix object, and returns the results in *DMObjNew1*, a new DataMatrix object. *DMObjNew1* has the same size (number of rows and columns), row names, and column names as *DMObj1*. The `I`,`J`th element of *DMObjNew1* is equal to *Func*`(`*DMObj1*`(I,J))`, where *Func* is a function handle for a function that takes one input argument, returns one scalar value, and returns values of the same class each time it is called.

DMObjNew1 `= dmarrayfun(`*Func*`,` *DMObj1*`,` *DMObj2*`, ...)` evaluates the function specified by *Func* using elements in *DMObj1*, *DMObj2*, etc. as input arguments. The `I`,`J`th element of *DMObjNew1* is equal to *Func*`(`*DMObj1*`(I,J),` *DMObj2*`(I,J), ...)`, where *Func* is a function handle for a function that takes multiple input arguments, returns one scalar, and returns values of the same class each time it is called.

`[`*DMObjNew1*`,` *DMObjNew2*`, ...] = dmarrayfun(`*Func*`,` *DMObj1*`, ...)` evaluates the function specified by *Func* using elements in *DMObj1*, and possibly other input arguments. *Func* is a function handle for a function that takes one or more input arguments, returns multiple scalars, and returns values of the same class each time it is called. It returns DataMatrix objects *DMObjNew1*, *DMObjNew2*, etc. with each one corresponding to one of the outputs of *Func*. The outputs of *Func* may be of different classes, however, but each output must be the same each time it is called.

`[`*DMObjNew1*`, ...] = dmarrayfun(`*Func*`,` *DMObj1*`, ...'`*PropertyName*`',` *PropertyValue*`, ...)` calls `dmarrayfun` with optional properties that use property name/property value pairs. You can specify one or more properties in any order. Each *PropertyName* must be enclosed in single quotation marks and is case insensitive. These property name/property value pairs are as follows:

`[`*DMObjNew1*`, ...] = dmarrayfun(`*Func*`,` *DMObj1*`, ...'UniformOutput',` *UniformOutputValue*`, ...)` specifies whether *Func* must return output values without encapsulation in a cell array. Choices are `true` (default) or `false`. If `true`, `dmarrayfun` must return scalar values that can be concatenated into an array. These values can also be a cell array. If `false`, `dmarrayfun` returns a cell array (or multiple cell arrays), where the `I,J`th cell contains the value equal to `Func(DMObj1(I,J),...)`.

`[DMObjNew1, ...] = dmarrayfun(Func, DMObj1, ...'DataMatrixOutput', DataMatrixOutputValue, ...)` specifies whether return values must be DataMatrix objects. Choices are `true` (default) or `false`. If you set the `'UniformOutput'` property to `false`, this property is ignored.

`[DMObjNew1, ...] = dmarrayfun(Func, DMObj1, ...'Rows', RowsValue, ...)` applies the function only to the rows in the DataMatrix object specified by `RowsValue`, which can be a positive integer, vector of positive integers, character vector specifying a row name, cell array of character vectors, or a logical vector.

`[DMObjNew1, ...] = dmarrayfun(Func, DMObj1, ...'Columns', ColumnsValue, ...)` applies the function only to the columns in the DataMatrix object specified by `ColumnsValue`, which can be a positive integer, vector of positive integers, character vector specifying a column name, cell array of character vectors, or a logical vector.

`[DMObjNew1, ...] = dmarrayfun(Func, DMObj1, ...'ErrorHandler', ErrorFuncHandle, ...)` specifies a function handle to a function that `dmarrayfun` calls if the call to `Func` fails. The error handling function will be called with these input arguments:

- Structure with the following fields:
  - `identifier` — Identifier of the error
  - `message` — Error message text
  - `index` — Linear index into the input array(s) at which the error occurred
- Set of input arguments at which the call to the function failed

If you do not specify `ErrorFuncHandle`, `dmarrayfun` rethrows the error from the call to `Func`.

Version History

Introduced in R2008b

See Also

`DataMatrix` | `dmbsxfun` | `arrayfun`

Topics

DataMatrix object on page 1-734

dmbsxfun (DataMatrix)

Apply element-by-element binary operation to two DataMatrix objects with singleton expansion enabled

Syntax

`DMObjNew = dmbsxfun(Func, DMObj1, DMObj2)`

Input Arguments

`Func`	Function handle for a function or a built-in function. For more information on built-in functions, see `bsxfun`.
`DMObj1`, *`DMObj2`*	Either of the following: • DataMatrix object, such as created by `DataMatrix` (object constructor) • MATLAB numeric array At least one of these input arguments must be a DataMatrix object.

Output Arguments

`DMObjNew`	DataMatrix object or MATLAB numeric array created from element-by-element binary operation of two DataMatrix objects with singleton expansion enabled.

Description

`DMObjNew` `= dmbsxfun(`*`Func`*`,` *`DMObj1`*`,` *`DMObj2`*`)` applies an element-by-element binary operation to the DataMatrix objects *`DMObj1`* and *`DMObj2`*, with singleton expansion enabled. *`Func`* is a function handle, and can be for a function or a built-in function. For more information on built-in functions, see `bsxfun`.

`DMObj1` and *`DMObj2`* can be DataMatrix objects or MATLAB numeric arrays; however, at least one of these input arguments must be a DataMatrix object. *`DMObj1`* and *`DMObj2`* must have the same number of rows or the same number or columns. If they don't have the same number of rows, then one must be a row vector and its rows are expanded down to be equal to the larger matrix. If they don't have the same number of columns, then one must be a column vector and its columns are expanded across to be equal to the larger matrix.

`DMObjNew` is a DataMatrix object, unless the larger input argument is a MATLAB numeric array; then *`DMObjNew`* is also a numeric array. The size (number of rows and columns) of *`DMObjNew`* is equal to the larger of the two input arguments. The row names and column names of *`DMObjNew`* come from the larger input argument, or, if both inputs are the same size, from the first input argument.

Examples

1 Use the `DataMatrix` constructor function to create a DataMatrix object.

```
A = bioma.data.DataMatrix(magic(3), 'RowNames', true, ...
                          'ColNames',true)
```

2 Use the built-in function `@minus` to subtract the column means from this DataMatrix object.

```
A = dmbsxfun(@minus, A, mean(A))
```

Version History

Introduced in R2008b

See Also

`DataMatrix` | `bsxfun`

Topics

DataMatrix object on page 1-734

dmNames

Class: `bioma.data.ExptData`
Package: `bioma.data`

Retrieve or set `Name` properties of DataMatrix objects in ExptData object

Syntax

```
DMNames = dmNames(EDObj)
DMNames = dmNames(EDObj, Subset)
NewEDObj = dmNames(EDObj, Subset, NewDMNames)
```

Description

`DMNames = dmNames(EDObj)` returns a cell array of character vectors specifying the `Name` properties of all the DataMatrix objects in an ExptData object.

`DMNames = dmNames(EDObj, Subset)` returns a cell array of character vectors specifying the `Name` properties of a subset of the DataMatrix objects in an ExptData object.

`NewEDObj = dmNames(EDObj, Subset, NewDMNames)` replaces the `Name` properties of DataMatrix objects specified by *`Subset`* in *`EDObj`*, an ExptData object, with *`NewDMNames`*, and returns *`NewEDObj`*, a new ExptData object.

Input Arguments

`EDObj`

Object of the `bioma.data.ExptData` class.

Default:

`Subset`

One of the following to specify the names of a subset of the DataMatrix objects in an ExptData object:

- Character vector specifying a name
- Cell array of character vectors specifying names
- Positive integer
- Vector of positive integers
- Logical vector

Default:

`NewDMNames`

New names for specific DataMatrix objects within an ExptData object, specified by one of the following:

- Numeric vector
- Character vector or cell array of character vectors
- Character vector, which `dmNames` uses as a prefix for the DataMatrix object names, with numbers appended to the prefix
- Logical `true` or `false` (default). If `true`, `dmNames` assigns unique names using the format `DM1`, `DM2`, etc.

The number of elements in *`NewDMNames`* must equal the number of DataMatrix objects specified by *`Subset`*.

Default:

Output Arguments

`DMNames`

Cell array of character vectors specifying the names of all or some of the DataMatrix objects in an ExptData object.

`NewEDObj`

Object of the `bioma.data.ExptData` class, returned after replacing names of specific DataMatrix objects.

Examples

Construct an ExptData object, and then retrieve the names of DataMatrix objects from it:

```
% Import bioma.data package to make constructor functions
% available
import bioma.data.*
% Create DataMatrix object from .txt file containing
% expression values from microarray experiment
dmObj = DataMatrix('File', 'mouseExprsData.txt');
% Construct ExptData object
EDObj = ExptData(dmObj);
% Retrieve DataMatrix object names
DMNames = dmNames(EDObj);
```

See Also

`bioma.data.ExptData` | `DataMatrix` | `elementNames` | `featureNames` | `sampleNames`

Topics

"Representing Expression Data Values in ExptData Objects"

dmwrite (DataMatrix)

Write DataMatrix object to text file

Syntax

```
dmwrite(DMObj, File)
dmwrite(..., 'Delimiter', DelimiterValue, ...)
dmwrite(..., 'Precision', PrecisionValue, ...)
dmwrite(..., 'Header', HeaderValue, ...)
dmwrite(..., 'Annotated', AnnotatedValue, ...)
dmwrite(..., 'Append', AppendValue, ...)
```

Arguments

DMObj	DataMatrix object, such as created by `DataMatrix` (object constructor).
File	Character vector specifying either a file name or a path and file name for saving the text file.
DelimiterValue	Character vector specifying a delimiter symbol to use as a matrix column separator. Typical choices are: • `' '` • `'\t'` (default) • `','` • `';'` • `'|'`
PrecisionValue	Precision for writing the data to the text file, specified by either: • Positive integer specifying the number of significant digits • C-style format character vector starting with `%`, such as `'%6.5f'` Default is `5`.
HeaderValue	Character vector specifying the first line of the text file. Default is the `Name` property for the DataMatrix object.
AnnotatedValue	Controls the writing of row and column names to the text file. Choices are `true` (default) or `false`.
AppendValue	Controls the appending of *DMObj* to *File* when it is an existing file. Choices are `true` or `false` (default). If `false`, `dmwrite` overwrites *File*.

Description

`dmwrite(`*DMObj*`, `*File*`)` writes a DataMatrix object to a text file using the delimiter `\t` to separate DataMatrix columns. `dmwrite` writes the data starting at the first column of the first row in the destination file.

`dmwrite(..., 'PropertyName', PropertyValue, ...)` calls `dmwrite` with optional properties that use property name/property value pairs. You can specify one or more properties in any order. Enclose each *`PropertyName`* in single quotation marks. Each *`PropertyName`* is case insensitive. These property name/property value pairs are as follows:

`dmwrite(..., 'Delimiter', DelimiterValue, ...)` specifies a delimiter symbol to use as a column separator for separating matrix columns. Default is `'\t'`.

`dmwrite(..., 'Precision', PrecisionValue, ...)` specifies the precision for writing the data to the text file. Default is `5`.

`dmwrite(..., 'Header', HeaderValue, ...)` specifies the first line of the text file. Default is the `Name` property for the DataMatrix object.

`dmwrite(..., 'Annotated', AnnotatedValue, ...)` controls the writing of row and column names to the text file. Choices are `true` (default) or `false`.

`dmwrite(..., 'Append', AppendValue, ...)` controls the appending of *`DMObj`* to *`File`* when it is an existing file. Choices are `true` or `false` (default). If `false`, `dmwrite` overwrites *`File`*.

Examples

Create a DataMatrix object and write the contents to a text file:

```
% Create a DataMatrix object
dmobj = bioma.data.DataMatrix(rand(2,3), {'Row1', 'Row2'}, ...
                                          {'Col1', 'Col2', 'Col3'})
% Write the DataMatrix object to a text file
dmwrite(dmobj,'testdm.txt')
```

Version History

Introduced in R2009b

See Also

`DataMatrix`

Topics

DataMatrix object on page 1-734

dna2rna

Convert DNA sequence to RNA sequence

Syntax

SeqRNA = `dna2rna(`*SeqDNA*`)`

Arguments

SeqDNA	DNA sequence specified by any of the following: • Character vector or string with the characters `A`, `C`, `G`, `T`, and ambiguous characters `R`, `Y`, `K`, `M`, `S`, `W`, `B`, `D`, `H`, `V`, `N`, • Row vector of integers from the table Mapping Nucleotide Integers to Letter Codes. • MATLAB structure containing a `Sequence` field that contains a DNA sequence, such as returned by `fastaread`, `fastqread`, `emblread`, `getembl`, `genbankread`, or `getgenbank`.

Description

SeqRNA = `dna2rna(`*SeqDNA*`)` converts a DNA sequence to an RNA sequence by converting any thymine nucleotides (`T`) in the DNA sequence to uracil nucleotides (`U`). The RNA sequence is returned in the same format as the DNA sequence. For example, if *SeqDNA* is a vector of integers, then so is *SeqRNA*.

Examples

Convert a DNA sequence to an RNA sequence

```
dna = randseq(100)
```

```
dna = 
'TTATGACGTTATTCTACTTTGATTGTGCGAGACAATGCTACCTTACCGGTCGGAACTCGATCGGTTGAACTCTATCACGCCTGGTCTTCGAAGTTA
```

```
rna = dna2rna(dna)
```

```
rna = 
'UUAUGACGUUAUUCUACUUUGAUUGUGCGAGACAAUGCUACCUUACCGGUCGGAACUCGAUCGGUUGAACUCUAUCACGCCUGGUCUUCGAAGUUA
```

Version History

Introduced before R2006a

See Also

`rna2dna` | `regexp` | `strrep`

dnds

Estimate synonymous and nonsynonymous substitution rates

Syntax

```
[Dn, Ds, Vardn, Vards] = dnds(SeqNT1, SeqNT2)

[Dn, Ds, Vardn, Vards] = dnds(SeqNT1, SeqNT2, ...'GeneticCode',
GeneticCodeValue, ...)
[Dn, Ds, Vardn, Vards] = dnds(SeqNT1, SeqNT2, ...'Method', MethodValue, ...)
[Dn, Ds, Vardn, Vards] = dnds(SeqNT1, SeqNT2, ...'Window', WindowValue, ...)
[Dn, Ds, Vardn, Vards] = dnds(SeqNT1, SeqNT2, ...'AdjustStops',
AdjustStopsValue, ...)
[Dn, Ds, Vardn, Vards] = dnds(SeqNT1, SeqNT2, ...'Verbose',
VerboseValue, ...)
```

Input Arguments

SeqNT1, *SeqNT2*	Nucleotide sequences. Enter a character vector, string, or a structure with the field `Sequence`.
GeneticCodeValue	Property to specify a genetic code. Enter a Code Number, a character vector, or string with a Code Name from the table Genetic Codes. If you use a Code Name, you can truncate it to the first two characters. Default is `1` or `Standard`.
MethodValue	Character vector or string specifying the method for calculating substitution rates. Choices are: • `NG` (default) — Nei-Gojobori method (1986) on page 1-766 uses the number of synonymous and nonsynonymous substitutions and the number of potentially synonymous and nonsynonymous sites. Based on the Jukes-Cantor model. • `LWL` — Li-Wu-Luo method (1985) on page 1-766 uses the number of transitional and transversional substitutions at three different levels of degeneracy of the genetic code. Based on Kimura's two-parameter model. • `PBL` — Pamilo-Bianchi-Li method (1993) on page 1-766 is similar to the Li-Wu-Luo method, but with bias correction. Use this method when the number of transitions is much larger than the number of transversions.
WindowValue	Integer specifying the sliding window size, in codons, for calculating substitution rates and variances.
AdjustStopsValue	Controls whether stop codons are excluded from calculations. Choices are `true` (default) or `false`.

VerboseValue	Property to control the display of the codons considered in the computations and their amino acid translations. Choices are `true` or `false` (default). **Tip** Specify `true` to use this display to manually verify the codon alignment of the two input sequences. The presence of stop codons (*) in the amino acid translation can indicate that *SeqNT1* and *SeqNT2* are not codon-aligned.

Output Arguments

Dn	Nonsynonymous substitution rate(s).
Ds	Synonymous substitution rate(s).
Vardn	Variance for the nonsynonymous substitution rate(s).
Vards	Variance for the synonymous substitutions rate(s).

Description

`[`*`Dn, Ds, Vardn, Vards`*`] = dnds(`*`SeqNT1, SeqNT2`*`)` estimates the synonymous and nonsynonymous substitution rates per site between the two homologous nucleotide sequences, *SeqNT1* and *SeqNT2*, by comparing codons using the Nei-Gojobori method.

`dnds` returns:

- *Dn* — Nonsynonymous substitution rate(s).
- *Ds* — Synonymous substitution rate(s).
- *Vardn* — Variance for the nonsynonymous substitution rate(s).
- *Vards* — Variance for the synonymous substitutions rate(s).

This analysis:

- Assumes that the nucleotide sequences, *SeqNT1* and *SeqNT2*, are codon-aligned, that is, do not have frame shifts

  Tip If your sequences are not codon-aligned, use the `nt2aa` function to convert them to amino acid sequences, use the `nwalign` function to globally align them, then use the `seqinsertgaps` function to recover the corresponding codon-aligned nucleotide sequences. For an example, see "Estimate synonymous and nonsynonymous substitution rates between two nucleotide sequences" on page 1-764.
- Excludes codons that include ambiguous nucleotide characters or gaps
- Considers the number of codons in the shorter of the two nucleotide sequences

Caution If *SeqNT1* and *SeqNT2* are too short or too divergent, saturation can be reached, and `dnds` returns `NaN`s and a warning message.

`[`*`Dn, Ds, Vardn, Vards`*`] = dnds(`*`SeqNT1, SeqNT2`*`, ...'`*`PropertyName`*`',` *`PropertyValue`*`, ...)` calls `dnds` with optional properties that use property name/property value pairs. You can specify one or more properties in any order. Each *PropertyName* must be enclosed in single quotation marks and is case insensitive. These property name/property value pairs are as follows:

`[Dn, Ds, Vardn, Vards] = dnds(SeqNT1, SeqNT2, ...'GeneticCode', GeneticCodeValue, ...)` calculates synonymous and nonsynonymous substitution rates using the specified genetic code. Enter a Code Number, a character vector or string with a Code Name from the table Genetic Codes. If you use a Code Name, you can truncate it to the first two characters. Default is `1` or `Standard`.

`[Dn, Ds, Vardn, Vards] = dnds(SeqNT1, SeqNT2, ...'Method', MethodValue, ...)` allows you to calculate synonymous and nonsynonymous substitution rates using the following algorithms:

- `NG` (default) — Nei-Gojobori method (1986) uses the number of synonymous and nonsynonymous substitutions and the number of potentially synonymous and nonsynonymous sites. Based on the Jukes-Cantor model.
- `LWL` — Li-Wu-Luo method (1985) uses the number of transitional and transversional substitutions at three different levels of degeneracy of the genetic code. Based on Kimura's two-parameter model.
- `PBL` — Pamilo-Bianchi-Li method (1993) is similar to the Li-Wu-Luo method, but with bias correction. Use this method when the number of transitions is much larger than the number of transversions.

`[Dn, Ds, Vardn, Vards] = dnds(SeqNT1, SeqNT2, ...'Window', WindowValue, ...)` performs the calculations over a sliding window, specified in codons. Each output is an array containing a rate or variance for each window.

`[Dn, Ds, Vardn, Vards] = dnds(SeqNT1, SeqNT2, ...'AdjustStops', AdjustStopsValue, ...)` controls whether stop codons are excluded from calculations. Choices are `true` (default) or `false`.

Tip When the `'AdjustStops'` property is set to `true`, the following are true:

- Stop codons are excluded from frequency tables.
- Paths containing stop codons are not counted in the Nei-Gojobori method.

`[Dn, Ds, Vardn, Vards] = dnds(SeqNT1, SeqNT2, ...'Verbose', VerboseValue, ...)` controls the display of the codons considered in the computations and their amino acid translations. Choices are `true` or `false` (default).

Tip Specify `true` to use this display to manually verify the codon alignment of the two input sequences, *SeqNT1* and *SeqNT2*. The presence of stop codons (*) in the amino acid translation can indicate that *SeqNT1* and *SeqNT2* are not codon-aligned.

Examples

Estimate synonymous and nonsynonymous substitution rates between two nucleotide sequences

This example shows how to estimate synonymous and nonsynonymous substitution rates between two nucleotide sequences that are not codon-aligned.

This example uses two nucleotide sequences representing the human HEXA gene (accession number: NM_000520) and mouse HEXA gene (accession number: AK080777).

If you have live internet connection, you can use `getgenbank` function to retrieve the sequence information from the NCBI data repository and load the data into MATLAB®.

```
humanHEXA = getgenbank('NM_000520');
mouseHEXA = getgenbank('AK080777');
```

For your convenience, MATLAB provides these two sequences in the following mat file. Note that data in public databases are frequently updated and curated, and the results in this example may slightly differ if you use the latest data.

```
load hexosaminidase.mat
```

Extract the coding regions from the two nucleotide sequences.

```
humanHEXA_cds = featureparse(humanHEXA,'feature','CDS','Sequence',true);
mouseHEXA_cds = featureparse(mouseHEXA,'feature','CDS','Sequence',true);
```

Align the amino acid sequences converted from the nucleotide sequences.

```
[sc,al] = nwalign(nt2aa(humanHEXA_cds),nt2aa(mouseHEXA_cds),'extendgap',1);
```

Use the `seqinsertgaps` function to copy the gaps from the aligned amino acid sequences to their corresponding nucleotide sequences, thus codon-aligning them.

```
humanHEXA_aligned = seqinsertgaps(humanHEXA_cds,al(1,:))
```

```
humanHEXA_aligned = 
'atgacaagctccaggctttggttttcgctgctgctggcggcagcgttcgcaggacgggcgacggccctctggccctggcctcagaacttccaaac
```

```
mouseHEXA_aligned = seqinsertgaps(mouseHEXA_cds,al(3,:))
```

```
mouseHEXA_aligned = 
'atggccggctgcaggctctgggtttcgctgctgctggcggcggcgttggcttgcttggccacggcactgtggccgtggccccagtacatccaaac
```

Estimate the synonymous and nonsynonymous substitutions rates of the codon-aligned nucleotide sequences and also display the codons considered in the computations and their amino acid translations.

```
[nonsynSubRate,synSubRate] = dnds(humanHEXA_aligned,mouseHEXA_aligned,'verbose',true)
```

```
DNDS:
Codons considered in the computations:
ATGACAAGCTCCAGGCTTTGGTTTTCGCTGCTGCTGGCGGCAGCGTTCGCAGGACGGGCGACGGCCCTCTGGCCCTGGCCTCAGAACTTCCAAACC
ATGGCCGGCTGCAGGCTCTGGGTTTCGCTGCTGCTGGCGGCGGCGTTGGCTTGCTTGGCCACGGCACTGTGGCCGTGGCCCCAGTACATCCAAACC
Translations:
M  T  S  S  R  L  W  F  S  L  L  L  A  A  A  F  A  G  R  A  T  A  L  W  P  W  P  Q  N  F  Q  T
M  A  G  C  R  L  W  V  S  L  L  L  A  A  A  L  A  C  L  A  T  A  L  W  P  W  P  Q  Y  I  Q  T

nonsynSubRate = 0.0933
```

```
synSubRate = 0.5181
```

Version History

Introduced before R2006a

References

[1] Li, W., Wu, C., and Luo, C. (1985). A new method for estimating synonymous and nonsynonymous rates of nucleotide substitution considering the relative likelihood of nucleotide and codon changes. Molecular Biology and Evolution *2(2)*, 150–174.

[2] Nei, M., and Gojobori, T. (1986). Simple methods for estimating the numbers of synonymous and nonsynonymous nucleotide substitutions. Molecular Biology and Evolution *3(5)*, 418–426.

[3] Nei, M., and Jin, L. (1989). Variances of the average numbers of nucleotide substitutions within and between populations. Molecular Biology and Evolution *6(3)*, 290–300.

[4] Nei, M., and Kumar, S. (2000). Synonymous and nonsynonymous nucleotide substitutions" in Molecular Evolution and Phylogenetics (Oxford University Press).

[5] Pamilo, P., and Bianchi, N. (1993). Evolution of the Zfx And Zfy genes: rates and interdependence between the genes. Molecular Biology and Evolution *10(2)*, 271–281.

See Also

`featureparse` | `nwalign` | `seqinsertgaps`

Topics

dndsml on page 1-767
geneticcode on page 1-880
nt2aa on page 1-1393
seqpdist on page 1-1727

dndsml

Estimate synonymous and nonsynonymous substitution rates using maximum likelihood method

Syntax

```
[Dn, Ds, Like] = dndsml(SeqNT1, SeqNT2)

[Dn, Ds, Like] = dndsml(SeqNT1, SeqNT2, ...'GeneticCode',
GeneticCodeValue, ...)
[Dn, Ds, Like] = dndsml(SeqNT1, SeqNT2, ...'Verbose', VerboseValue, ...)
```

Input Arguments

SeqNT1, *SeqNT2*	Nucleotide sequences. Enter a character vector, string, or a structure with the field `Sequence`.
GeneticCodeValue	Property to specify a genetic code. Enter a Code Number, a character vector, or string with a Code Name from the table Genetic Codes. If you use a Code Name, you can truncate it to the first two characters. Default is `1` or `Standard`.
VerboseValue	Property to control the display of the codons considered in the computations and their amino acid translations. Choices are `true` or `false` (default). **Tip** Specify `true` to use this display to manually verify the codon alignment of the two input sequences. The presence of stop codons (*) in the amino acid translation can indicate that *SeqNT1* and *SeqNT2* are not codon-aligned.

Output Arguments

Dn	Nonsynonymous substitution rate(s).
Ds	Synonymous substitution rate(s).
Like	Likelihood of estimate of substitution rates.

Description

`[Dn, Ds, Like] = dndsml(SeqNT1, SeqNT2)` estimates the synonymous and nonsynonymous substitution rates between the two homologous sequences, *SeqNT1* and *SeqNT2*, using the Goldman-Yang method (1994) on page 1-770. This maximum likelihood method estimates an explicit model for codon substitution that accounts for transition/transversion rate bias and base/codon frequency bias. Then it uses the model to correct synonymous and nonsynonymous counts to account for multiple substitutions at the same site. The maximum likelihood method is best suited when the sample size is significant (larger than 100 bases) and when the sequences being compared can have transition/transversion rate biases and base/codon frequency biases.

`dndsml` returns:

- *`Dn`* — Nonsynonymous substitution rate(s).
- *`Ds`* — Synonymous substitution rate(s).
- *`Like`* — Likelihood of this estimate.

This analysis:

- Assumes that the nucleotide sequences, *`SeqNT1`* and *`SeqNT2`*, are codon-aligned, that is, do not have frame shifts.

  Tip If your sequences are not codon-aligned, use the `nt2aa` function to convert them to amino acid sequences, use the `nwalign` function to globally align them, then use the `seqinsertgaps` function to recover the corresponding codon-aligned nucleotide sequences. For an example, see "Estimate synonymous and nonsynonymous substitution rates between two nucleotide sequences using maximum likelihood method" on page 1-768.
- Excludes any ambiguous nucleotide characters or codons that include gaps.
- Considers the number of codons in the shorter of the two nucleotide sequences.

Caution If *`SeqNT1`* and *`SeqNT2`* are too short or too divergent, saturation can be reached, and `dndsml` returns `NaN`s and a warning message.

`[`*`Dn, Ds, Like`*`] = dndsml(`*`SeqNT1, SeqNT2`*`, ...'`*`PropertyName`*`',` *`PropertyValue`*`, ...)` calls `dnds` with optional properties that use property name/property value pairs. You can specify one or more properties in any order. Each *`PropertyName`* must be enclosed in single quotation marks and is case insensitive. These property name/property value pairs are as follows:

`[`*`Dn, Ds, Like`*`] = dndsml(`*`SeqNT1, SeqNT2`*`, ...'GeneticCode',` *`GeneticCodeValue`*`, ...)` calculates synonymous and nonsynonymous substitution rates using the specified genetic code. Enter a Code Number, a character vector or string with a Code Name from the table Genetic Codes. If you use a Code Name, you can truncate it to the first two characters. Default is `1` or `Standard`.

`[`*`Dn, Ds, Like`*`] = dndsml(`*`SeqNT1, SeqNT2`*`, ...'Verbose',` *`VerboseValue`*`, ...)` controls the display of the codons considered in the computations and their amino acid translations. Choices are `true` or `false` (default).

Tip Specify `true` to use this display to manually verify the codon alignment of the two input sequences, *`SeqNT1`* and *`SeqNT2`*. The presence of stop codons (*) in the amino acid translation can indicate that *`SeqNT1`* and *`SeqNT2`* are not codon-aligned.

Examples

Estimate synonymous and nonsynonymous substitution rates between two nucleotide sequences using maximum likelihood method

This example shows how to estimate synonymous and nonsynonymous substitution rates between two nucleotide sequences that are not codon-aligned using maximum likelihood method.

This example uses two nucleotide sequences representing the human HEXA gene (accession number: NM_000520) and mouse HEXA gene (accession number: AK080777).

If you have live internet connection, you can use `getgenbank` function to retrieve the sequence information from the NCBI data repository and load the data into MATLAB®.

```
humanHEXA = getgenbank('NM_000520');
mouseHEXA = getgenbank('AK080777');
```

For your convenience, MATLAB provides these two sequences in the following mat file. Note that data in public databases are frequently updated and curated, and the results in this example may slightly differ if you use the latest data.

```
load hexosaminidase.mat
```

Extract the coding regions from the two nucleotide sequences.

```
humanHEXA_cds = featureparse(humanHEXA,'feature','CDS','Sequence',true);
mouseHEXA_cds = featureparse(mouseHEXA,'feature','CDS','Sequence',true);
```

Align the amino acid sequences converted from the nucleotide sequences.

```
[sc,al] = nwalign(nt2aa(humanHEXA_cds),nt2aa(mouseHEXA_cds),'extendgap',1);
```

Use the `seqinsertgaps` function to copy the gaps from the aligned amino acid sequences to their corresponding nucleotide sequences, thus codon-aligning them.

```
humanHEXA_aligned = seqinsertgaps(humanHEXA_cds,al(1,:))
```

```
humanHEXA_aligned = 
'atgacaagctccaggctttggttttcgctgctgctggcggcagcgttcgcaggacgggcgacggccctctggccctggcctcagaacttccaaac
```

```
mouseHEXA_aligned = seqinsertgaps(mouseHEXA_cds,al(3,:))
```

```
mouseHEXA_aligned = 
'atggccggctgcaggctctgggtttcgctgctgctggcggcggcgttggcttgcttggccacggcactgtggccgtggccccagtacatccaaac
```

Estimate the synonymous and nonsynonymous substitutions rates of the codon-aligned nucleotide sequences and also display the codons considered in the computations and their amino acid translations.

```
[nonsynSubRate,synSubRate] = dndsml(humanHEXA_aligned,mouseHEXA_aligned,'verbose',true)
```

```
DNDSML:
Codons considered in the computations:
ATGACAAGCTCCAGGCTTTGGTTTTCGCTGCTGCTGGCGGCAGCGTTCGCAGGACGGGCGACGGCCCTCTGGCCCTGGCCTCAGAACTTCCAAACC
ATGGCCGGCTGCAGGCTCTGGGTTTCGCTGCTGCTGGCGGCGGCGTTGGCTTGCTTGGCCACGGCACTGTGGCCGTGGCCCCAGTACATCCAAACC
Translations:
M  T  S  S  R  L  W  F  S  L  L  L  A  A  A  F  A  G  R  A  T  A  L  W  P  W  P  Q  N  F  Q  T  
M  A  G  C  R  L  W  V  S  L  L  L  A  A  A  L  A  C  L  A  T  A  L  W  P  W  P  Q  Y  I  Q  T  

Initial estimates: Kappa=3.301203, dn=0.093274, ds=0.518095, t=0.353716
ML estimates: Kappa=2.498253, omega(dn/ds)=0.185577, t=0.602465

nonsynSubRate = 0.0943

synSubRate = 0.5080
```

Version History

Introduced before R2006a

References

[1] Tamura, K., and Mei, M. (1993). Estimation of the number of nucleotide substitutions in the control region of mitochondrial DNA in humans and chimpanzees. Molecular Biology and Evolution *10*, 512–526.

[2] Yang, Z., and Nielsen, R. (2000). Estimating synonymous and nonsynonymous substitution rates under realistic evolutionary models. Molecular Biology and Evolution *17*, 32–43.

[3] Goldman, N., and Yang, Z. (1994). A Codon-based Model of Nucleotide Substitution for Protein-coding DNA Sequences. Mol. Biol. Evol. *11(5)*, 725–736.

See Also

`featureparse` | `nwalign` | `seqinsertgaps`

Topics

dnds on page 1-762
geneticcode on page 1-880
nt2aa on page 1-1393
seqpdist on page 1-1727

dolayout (biograph)

(Removed) Calculate node positions and edge trajectories

Note The `biograph` object and its functions have been removed. Use `graph` or `digraph` instead.

Syntax

`dolayout(`*`BGobj`*`)`

`dolayout(`*`BGobj`*`, 'Paths', `*`PathsOnlyValue`*`)`

Arguments

`BGobj`	Biograph object created by the `biograph` function (object constructor).
`PathsOnlyValue`	Controls the calculation of only the edge paths, leaving the nodes at their current positions. Choices are `true` or `false` (default).

Description

`dolayout(`*`BGobj`*`)` calls the layout engine to calculate the optimal position for each node so that its 2-D rendering is clean and uncluttered, and then calculates the best curves to represent the edges. The layout engine uses the following properties of the biograph object:

- `LayoutType` — Specifies the layout engine as `'hierarchical'`, `'equilibrium'`, or `'radial'`.
- `LayoutScale` — Rescales the sizes of the node before calling the layout engine. This gives more space to the layout and reduces the overlapping of nodes.
- `NodeAutoSize` — Controls precalculating the node size before calling the layout engine. When `NodeAutoSize` is set to `'on'`, the layout engine uses the node properties `FontSize` and `Shape`, and the biograph object property `LayoutScale` to precalculate the actual size of each node. When `NodeAutoSize` is set to `'off'`, the layout engine uses the node property `Size`.

For more information on the above properties, see Properties of a Biograph Object.

`dolayout(`*`BGobj`*`, 'Paths', `*`PathsOnlyValue`*`)` controls the calculation of only the edge paths, leaving the nodes at their current positions. Choices are `true` or `false` (default).

Version History

Introduced before R2006a

R2022b: Removed

Errors starting in R2022b

The `biograph` object and its functions have been removed. Use `graph` or `digraph` instead.

R2022a: Warns
Warns starting in R2022a

The function issues a warning that it will be removed in a future release.

R2021b: To be removed
Not recommended starting in R2021b

The function runs without warning, but it will be removed in a future release.

See Also
`graph` | `digraph`

double (DataMatrix)

Convert DataMatrix object to double-precision array

Syntax

```
B = double(DMObj)
B = double(DMObj, Rows)
B = double(DMObj, Rows, Cols)
```

Input Arguments

DMObj	DataMatrix object, such as created by `DataMatrix` (object constructor).
Rows, *Cols*	Row(s) or column(s) in *DMObj*, specified by one of the following: • Scalar • Vector of positive integers • Character vector specifying a row or column name • Cell array of row or column names • Logical vector

Output Arguments

B	MATLAB numeric array.

Description

`B = double(DMObj)` converts *DMObj*, a DataMatrix object, to a double-precision array, which it returns in *B*.

`B = double(DMObj, Rows)` converts a subset of *DMObj*, a DataMatrix object, specified by *Rows*, to a double-precision array, which it returns in*B*. *Rows* can be a positive integer, vector of positive integers, character vector specifying a row name, cell array of row names, or a logical vector.

`B = double(DMObj, Rows, Cols)` converts a subset of *DMObj*, a DataMatrix object, specified by *Rows* and *Cols*, to a double-precision array, which it returns in*B*. *Cols* can be a positive integer, vector of positive integers, character vector specifying a column name, cell array of column names, or a logical vector.

Version History

Introduced in R2008b

See Also

`DataMatrix` | `single`

Topics

DataMatrix object on page 1-734

elementData

Class: `bioma.ExpressionSet`
Package: `bioma`

Retrieve or set data element (DataMatrix object) in ExpressionSet object

Syntax

```
DMObj = elementData(ESObj, Element)
NewESObj = elementData(ESObj, Element, NewDMObj)
```

Description

`DMObj = elementData(ESObj, Element)` returns the DataMatrix object from an ExpressionSet object, specified by *`Element`*, a positive integer or a character vector specifying an element name.

`NewESObj = elementData(ESObj, Element, NewDMObj)` replaces the DataMatrix object specified by *`Element`* in *`ESObj`*, an ExpressionSet object, with *`NewDMObj`*, a new DataMatrix object, and returns *`NewESObj`*, a new ExpressionSet object.

Input Arguments

`ESObj`

Object of the `bioma.ExpressionSet` class.

Default:

`Element`

Element (DataMatrix object) in an ExpressionSet object, specified by either of the following:

- Positive integer
- Character vector specifying the element name

Default:

`NewDMObj`

Object of the DataMatrix on page 1-734 class. The sample names and feature names in *`NewDMObj`* must match the sample names and feature names in the DataMatrix object specified by *`Element`*.

Default:

Output Arguments

`DMObj`

Object of the DataMatrix on page 1-734 class, returned from the ExptData object of an ExpressionSet object.

`NewESObj`

Object of the `bioma.ExpressionSet` class, returned after replacing a specified data element (DataMatrix object).

Examples

Construct an ExpressionSet object, `ESObj`, as described in the “Examples” on page 1-0 section of the `bioma.ExpressionSet` class reference page. Extract a DataMatrix object from it:

```
% Extract first DataMatrix object
ExtractedDMObj = elementData(ESObj, 1);
```

See Also

`bioma.ExpressionSet` | `bioma.data.ExptData` | `DataMatrix`

Topics

“Managing Gene Expression Data in Objects”

elementData

Class: `bioma.data.ExptData`
Package: `bioma.data`

Retrieve or set data element (DataMatrix object) in ExptData object

Syntax

```
DMObj = elementData(EDObj, Element)
NewEDObj = elementData(EDObj, Element, NewDMObj)
```

Description

`DMObj = elementData(EDObj, Element)` returns the DataMatrix object from an ExptData object, specified by *`Element`*, a positive integer or character vector specifying an element name.

`NewEDObj = elementData(EDObj, Element, NewDMObj)` replaces the element (DataMatrix object) specified by *`Element`* in *`EDObj`*, an ExptData object, with *`NewDMObj`*, a new DataMatrix object, and returns *`NewEDObj`*, a new ExptData object.

Input Arguments

EDObj

Object of the `bioma.data.ExptData` class.

Default:

Element

Element (DataMatrix object) in an ExptData object, specified by either of the following:

- Positive integer
- Character vector specifying the element name

Default:

NewDMObj

Object of the DataMatrix on page 1-734 class. The sample names and feature names in *`NewDMObj`* must match the sample names and feature names of *`EDObj`*.

Default:

Output Arguments

DMObj

Object of the DataMatrix on page 1-734 class, returned from an ExptData object.

`NewEDObj`

Object of the `bioma.data.ExptData` class, returned after replacing a data element (DataMatrix object).

Examples

Construct an ExptData object, and then extract a DataMatrix object from it:

```
% Import bioma.data package to make constructor functions
% available
import bioma.data.*
% Create DataMatrix object from .txt file containing
% expression values from microarray experiment
dmObj = DataMatrix('File', 'mouseExprsData.txt');
% Construct ExptData object
EDObj = ExptData(dmObj);
% Extract first DataMatrix object
ExtractedDMObj = elementData(EDObj, 1);
```

See Also

`bioma.data.ExptData` | `DataMatrix`

Topics

“Representing Expression Data Values in ExptData Objects”

elementNames

Class: `bioma.ExpressionSet`
Package: `bioma`

Retrieve or set element names of DataMatrix objects in ExpressionSet object

Syntax

```
ElmtNames = elementNames(ESObj)
ElmtNames = elementNames(ESObj, Subset)
NewESObj = elementNames(ESObj, Subset, NewElmtNames)
```

Description

`ElmtNames = elementNames(ESObj)` returns a cell array of character vectors specifying the element names of all the data elements (DataMatrix objects) stored in the ExptData object in an ExpressionSet object.

`ElmtNames = elementNames(ESObj, Subset)` returns a cell array of character vectors specifying the element names of a subset of the data elements (DataMatrix objects) in the ExptData object in an ExpressionSet object.

`NewESObj = elementNames(ESObj, Subset, NewElmtNames)` replaces the element names of the data elements (DataMatrix objects) specified by *`Subset`* in *`ESObj`*, an ExpressionSet object, with *`NewElmtNames`*, and returns *`NewESObj`*, a new ExpressionSet object.

Input Arguments

`ESObj`

Object of the `bioma.ExpressionSet` class.

`Subset`

One of the following to specify the element names of a subset of the data elements (DataMatrix objects) in the ExptData object of an ExpressionSet object:

- Character vector or string specifying an element name
- Cell array of character vectors or string vector specifying element names
- Positive integer
- Vector of positive integers
- Logical vector

`NewElmtNames`

New element names for specific data elements (DataMatrix objects) within an ExpressionSet object, specified by one of the following:

- Numeric vector
- String vector, or cell array of character vectors
- Character vector or string, which `elementNames` uses as a prefix for the element names, with element numbers appended to the prefix
- Logical `true` or `false` (default). If `true`, `elementNames` assigns unique element names using the format `Elmt1`, `Elmt2`, etc.

The number of elements in *`NewElmtNames`* must equal the number of elements specified by *`Subset`*.

Output Arguments

`ElmtNames`

Cell array of character vectors specifying the element names of all or some of the data elements (DataMatrix objects) in the ExptData object of an ExpressionSet object.

`NewESObj`

Object of the `bioma.ExpressionSet` class, returned after replacing element names of specific data elements (DataMatrix objects).

Examples

Construct an ExpressionSet object, `ESObj`, as described in the “Examples” on page 1-0 section of the `bioma.ExpressionSet` class reference page. Retrieve the element names of the DataMatrix objects in it:

```
% Retrieve element names of DataMatrix objects
ENames = elementNames(ESObj);
```

See Also

`bioma.ExpressionSet` | `bioma.data.ExptData` | `DataMatrix` | `exptData`

Topics

“Managing Gene Expression Data in Objects”

elementNames

Class: `bioma.data.ExptData`
Package: `bioma.data`

Retrieve or set element names of DataMatrix objects in ExptData object

Syntax

```
ElmtNames = elementNames(EDObj)
ElmtNames = elementNames(EDObj, Subset)
NewEDObj = elementNames(EDObj, Subset, NewElmtNames)
```

Description

`ElmtNames` `= elementNames(`*`EDObj`*`)` returns a cell array of character vectors specifying the element names of all the data elements (DataMatrix objects) stored in an ExptData object.

`ElmtNames` `= elementNames(`*`EDObj`*`,` *`Subset`*`)` returns a cell array of character vectors specifying the element names of a subset of the data elements (DataMatrix objects) stored in an ExptData object.

`NewEDObj` `= elementNames(`*`EDObj`*`,` *`Subset`*`,` *`NewElmtNames`*`)` replaces the element names of the data elements (DataMatrix objects) specified by *`Subset`* in *`EDObj`*, an ExptData object, with *`NewElmtNames`*, and returns *`NewEDObj`*, a new ExptData object.

Input Arguments

`EDObj`

Object of the `bioma.data.ExptData` class.

Default:

`Subset`

One of the following to specify the element names of a subset of the data elements (DataMatrix objects) in an ExptData object:

- Character vector specifying an element name
- Cell array of character vectors specifying element names
- Positive integer
- Vector of positive integers
- Logical vector

Default:

`NewElmtNames`

New element names for specific data elements (DataMatrix objects) within an ExptData object, specified by one of the following:

- Numeric vector
- Character vector or cell array of character vectors
- Character vector, which `elementNames` uses as a prefix for the element names, with element numbers appended to the prefix
- Logical `true` or `false` (default). If `true`, `elementNames` assigns unique element names using the format `Elmt1`, `Elmt2`, etc.

The number of elements in *`NewElmtNames`* must equal the number of elements specified by *`Subset`*.

Default:

Output Arguments

`ElmtNames`

Cell array of character vectors specifying the element names of all or some of the data elements (DataMatrix objects) in an ExptData object.

`NewEDObj`

Object of the `bioma.data.ExptData` class, returned after replacing element names of specific data elements (DataMatrix objects).

Examples

Construct an ExptData object, and then retrieve the element names of DataMatrix objects from it:

```
% Import bioma.data package to make constructor functions
% available
import bioma.data.*
% Create DataMatrix object from .txt file containing
% expression values from microarray experiment
dmObj = DataMatrix('File', 'mouseExprsData.txt');
% Construct ExptData object
EDObj = ExptData(dmObj);
% Retrieve element names of DataMatrix objects
ENames = elementNames(EDObj);
```

See Also

`bioma.data.ExptData` | `DataMatrix` | `dmNames` | `featureNames` | `sampleNames`

Topics

"Representing Expression Data Values in ExptData Objects"

emblread

Read data from EMBL file

Syntax

```
EMBLData = emblread(File)
EMBLSeq = emblread (File, 'SequenceOnly', SequenceOnlyValue)
EMBLSeq = emblread (File, 'TimeOut', TimeOutValue)
```

Input Arguments

`File`	Either of the following: • Character vector or string specifying a file name, a path and file name, or a URL pointing to a file. The referenced file is an EMBL-formatted file. If you specify only a file name, that file must be on the MATLAB search path or in the MATLAB Current Folder. • Character vector or string that contains the text of an EMBL-formatted file **Tip** You can use the `getembl` function with the `'ToFile'` property to retrieve data from the European Molecular Biology Laboratory (EMBL) database and create an EMBL-formatted file.
`SequenceOnlyValue`	Controls the reading of only the sequence without the metadata. Choices are `true` or `false` (default).
`TimeOutValue`	Connection timeout in seconds, specified as a positive scalar. The default value is 5. For details, see here.

Output Arguments

`EMBLData`	Structure with fields corresponding to EMBL data.
`EMBLSeq`	Character vector representing the sequence.

Description

`EMBLData` `= emblread(`*`File`*`)` reads data from *`File`*, an EMBL-formatted file, and creates *`EMBLData`*, a MATLAB structure containing fields corresponding to the EMBL two-character line type code, based on release 107 of the EMBL-Bank flat file format. Each line type code is stored as a separate element in the structure. For a list of the EMBL two-character line type codes, see ftp://ftp.ebi.ac.uk/pub/databases/embl/doc/usrman.txt.

Note Topology information was not included in EMBL flat files before release 87 of the database. When reading a file created before release 87, EMBLREAD returns an empty `Identification.Topology` field.

Note The entry name is no longer displayed in the ID line of EMBL flat files in release 87. When reading a file created in release 87, EMBLREAD returns the accession number in the `Identification.EntryName` field.

`EMBLSeq = emblread (File, 'SequenceOnly', SequenceOnlyValue)` controls the reading of only the sequence without the metadata. Choices are `true` or `false` (default).

`EMBLSeq = emblread (File, 'TimeOut', TimeOutValue)` sets the connection timeout (in seconds) to read data from a remote file or URL.

Examples

Read sequence information from EMBL file

Download the sequence information from the web and save to a file.

```
out = getembl('X00558','ToFile','rat_protein.txt');
```

Read data from the EMBL file.

```
seqData = emblread('rat_protein.txt')
```

```
seqData = 

  struct with fields:

            Identification: [1×1 struct]
                 Accession: 'X00558'
           SequenceVersion: 'X00558.1'
               DateCreated: '13-JUN-1985  Rel. 06, Created '
               DateUpdated: '18-APR-2005  Rel. 83, Last updated, Version 4 '
               Description: 'Rat liver apolipoprotein A-I mRNA  apoA-I    ...'
                   Keyword: 'apolipoprotein; lipoprotein; signal peptide. ...'
           OrganismSpecies: 'Rattus norvegicus  Norway rat                ...'
    OrganismClassification: [3×75 char]
                 Organelle: ''
                 Reference: {[1×1 struct]}
    DatabaseCrossReference: [4×75 char]
                  Comments: ''
                  Assembly: ''
                   Feature: [22×75 char]
                 BaseCount: [1×1 struct]
                  Sequence: 'agctccgggggaggtcgcccacatccttcgggatgaaagctgcag...'
```

Version History

Introduced before R2006a

See Also

`fastaread` | `genbankread` | `genpeptread` | `getembl` | `pdbread` | `seqviewer`

eq (DataMatrix)

Test DataMatrix objects for equality

Syntax

```
T = eq(DMObj1, DMObj2)
T = DMObj1 == DMObj2
T = eq(DMObj1, B)
T = DMObj1 == B
T = eq(B, DMObj1)
T = B == DMObj1
```

Input Arguments

DMObj1, *DMObj2*	DataMatrix objects, such as created by `DataMatrix` (object constructor).
B	MATLAB numeric or logical array.

Output Arguments

T	Logical matrix of the same size as *DMObj1* and *DMObj2* or *DMObj1* and *B*. It contains logical `1` (true) where elements in the first input are equal to the corresponding element in the second input, and logical `0` (false) when they are not equal.

Description

`T = eq(DMObj1, DMObj2)` or the equivalent `T = DMObj1 == DMObj2` compares each element in DataMatrix object *DMObj1* to the corresponding element in DataMatrix object *DMObj2*, and returns *T*, a logical matrix of the same size as *DMObj1* and *DMObj2*, containing logical `1` (true) where elements in *DMObj1* are equal to the corresponding element in *DMObj2*, and logical `0` (false) when they are not equal. *DMObj1* and *DMObj2* must have the same size (number of rows and columns), unless one is a scalar (1-by-1 DataMatrix object). *DMObj1* and *DMObj2* can have different `Name` properties.

`T = eq(DMObj1, B)` or the equivalent `T = DMObj1 == B` compares each element in DataMatrix object *DMObj1* to the corresponding element in *B*, a numeric or logical array, and returns *T*, a logical matrix of the same size as *DMObj1* and *B*, containing logical `1` (true) where elements in *DMObj1* are equal to the corresponding element in *B*, and logical `0` (false) when they are not equal. *DMObj1* and *B* must have the same size (number of rows and columns), unless one is a scalar.

`T = eq(B, DMObj1)` or the equivalent `T = B == DMObj1` compares each element in *B*, a numeric or logical array, to the corresponding element in DataMatrix object *DMObj1*, and returns *T*, a logical matrix of the same size as *B* and *DMObj1*, containing logical `1` (true) where elements in *B* are equal to the corresponding element in *DMObj1*, and logical `0` (false) when they are not equal. *B* and *DMObj1* must have the same size (number of rows and columns), unless one is a scalar.

MATLAB calls `T = eq(X, Y)` for the syntax `T = X == Y` when *X* or *Y* is a DataMatrix object.

Version History
Introduced in R2008b

See Also
`DataMatrix` | `ne`

Topics
DataMatrix object on page 1-734

evalrasmolscript

(To be removed) Send RasMol script commands to Molecule Viewer window

Note will be removed in a future release.

Syntax

`evalrasmolscript(FigureHandle, Command)`

Arguments

FigureHandle	Figure handle to a molecule viewer returned by the `molviewer` function.
Command	Any of the following: • Character vector, string, string vector, or cell array of character vectors specifying RasMol script commands. If there are multiple commands, use a `;` to separate them. **Note** For a complete list of RasMol script commands, see `https://www.stolaf.edu/academics/chemapps/jmol/docs/` • Character vector or string specifying a file name or a path and file name of a text file containing Jmol script commands. If you specify only a file name, that file must be on the MATLAB search path or in the MATLAB Current Folder.

Description

`evalrasmolscript(FigureHandle, Command)` sends the RasMol script commands specified by *`Command`* to *`FigureHandle`*, the figure handle of a Molecule Viewer window created using the `molviewer` function.

Version History

Introduced in R2007a

R2023a: Warns
Warns starting in R2023a

`evalrasmolscript` issues a warning that it will be removed in a future release.

R2020b: To be removed
Not recommended starting in R2020b

`evalrasmolscript` runs without warning. But it will be removed in a future release.

See Also

`getpdb` | `pdbread` | `pdbwrite`

expressions

Class: `bioma.ExpressionSet`
Package: `bioma`

Retrieve or set `Expressions` DataMatrix object from ExpressionSet object

Syntax

`ExpressionsDMObj` `= expressions(`*`ESObj`*`)`
`NewESObj` `= expressions(`*`ESObj`*`, `*`NewDMObj`*`)`

Description

`ExpressionsDMObj` `= expressions(`*`ESObj`*`)` returns the `Expressions` element (DataMatrix object), which contains expression values, from an ExpressionSet object.

`NewESObj` `= expressions(`*`ESObj`*`, `*`NewDMObj`*`)` replaces the `Expressions` element (DataMatrix object) in *`ESObj`*, an ExpressionSet object, with *`NewDMObj`*, a new DataMatrix object, and returns *`NewESObj`*, a new ExpressionSet object.

Input Arguments

`ESObj`

Object of the `bioma.ExpressionSet` class.

Default:

`NewDMObj`

Object of the DataMatrix on page 1-734 class.

Default:

Output Arguments

`ExpressionsDMObj`

DataMatrix object containing the expression values from the `Expressions` DataMatrix object within an ExpressionSet object.

`NewESObj`

ExpressionSet object returned after replacing the `Expressions` DataMatrix object.

Examples

Construct an ExpressionSet object, `ESObj`, as described in the “Examples” on page 1-0 section of the `bioma.ExpressionSet` class reference page. Extract the `Expressions` DataMatrix object from it:

```
% Extract expression values from Expressions DataMatrix object
ExpressionsDMObj = expressions(ESObj);
```

See Also

`bioma.ExpressionSet` | `bioma.data.ExptData` | `DataMatrix`

Topics

"Managing Gene Expression Data in Objects"

exprprofrange

Calculate range of gene expression profiles

Syntax

```
Range = exprprofrange(Data)
[Range, LogRange] = exprprofrange(Data)

... = exprprofrange(Data, 'ShowHist', ShowHistValue)
```

Arguments

Data	DataMatrix object on page 1-734 or numeric matrix of expression values, where each row corresponds to a gene.
ShowHistValue	Controls the display of a histogram with range data. Default is: • `false` — When output values are specified. • `true` — When output values are not specified.

Description

`Range = exprprofrange(Data)` calculates the range of each expression profile in *Data*, a DataMatrix object on page 1-734 or numeric matrix of expression values, where each row corresponds to a gene.

`[Range, LogRange] = exprprofrange(Data)` returns the log range, that is, `log(max(prof))-log(min(prof))`, of each expression profile. If you do not specify output arguments, `exprprofrange` displays a histogram bar plot of the range.

`... = exprprofrange(Data, 'ShowHist', ShowHistValue)` controls the display of a histogram with range data. Choices for *ShowHistValue* are `true` or `false`.

Examples

Calculate range of gene expression profiles

Load microarray data containing gene expression levels of Saccharomyces cerevisiae (yeast).

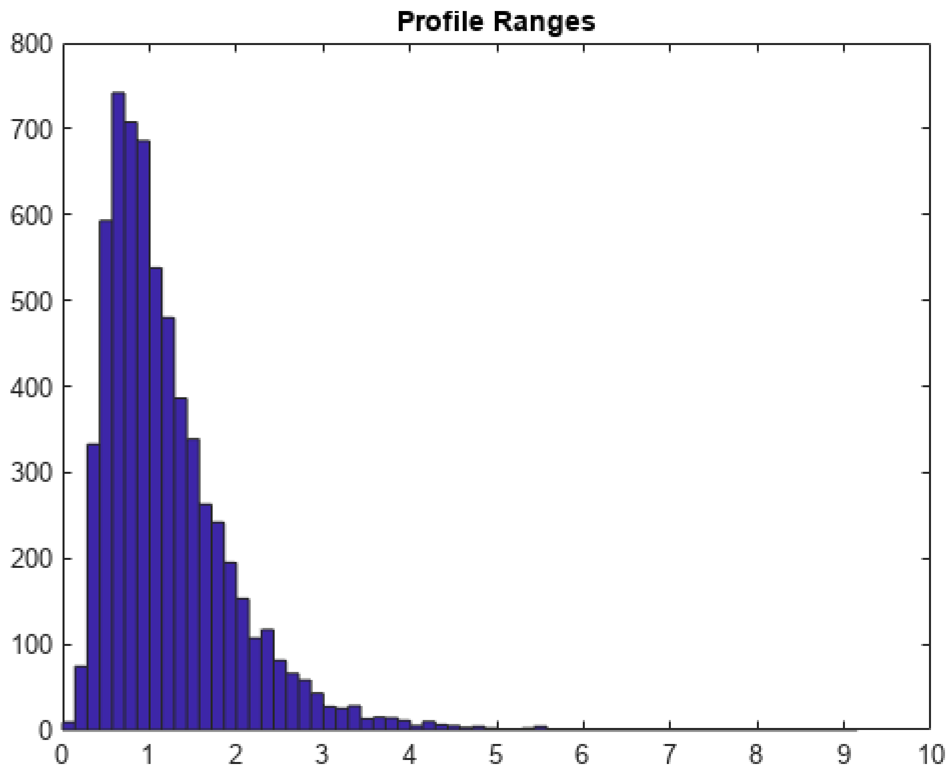
```
load yeastdata
```

This MAT-file includes three variables, which are added to the MATLAB® workspace:

- `yeastvalues` - A matrix of gene expression data
- genes - A cell array of GenBank® accession numbers for labeling the rows in yeastvalues
- times - A vector of time values for labeling the columns in yeastvalues

Calculate the range of expression profiles for yeast data as gene expression changes during the metabolic shift from fermentation to respiration. And display a histogram of the data.

```
range = exprprofrange(yeastvalues,'ShowHist',true);
```



Version History

Introduced before R2006a

See Also

`exprprofvar` | `generangefilter`

exprprofvar

Calculate variance of gene expression profiles

Syntax

```
Variance = exprprofvar(Data)

exprprofvar(..., 'PropertyName', PropertyValue,...)
exprprofvar(..., 'ShowHist', ShowHistValue)
```

Arguments

Data	DataMatrix object on page 1-734 or numeric matrix of expression values, where each row corresponds to a gene.
ShowHistValue	Controls the display of a histogram with variance data. Default is: • `false` — When output values are specified. • `true` — When output values are not specified.

Description

`Variance = exprprofvar(Data)` calculates the variance of each expression profile in *Data*, a DataMatrix object on page 1-734 or numeric matrix of expression values, where each row corresponds to a gene. If you do not specify output arguments, this function displays a histogram bar plot of the range.

`exprprofvar(..., 'PropertyName', PropertyValue,...)` defines optional properties using property name/value pairs.

`exprprofvar(..., 'ShowHist', ShowHistValue)` controls the display of a histogram with range data. Choices for *ShowHistValue* are `true` or `false`.

Examples

Calculate variance of gene expression profiles

Load microarray data containing gene expression levels of Saccharomyces cerevisiae (yeast).

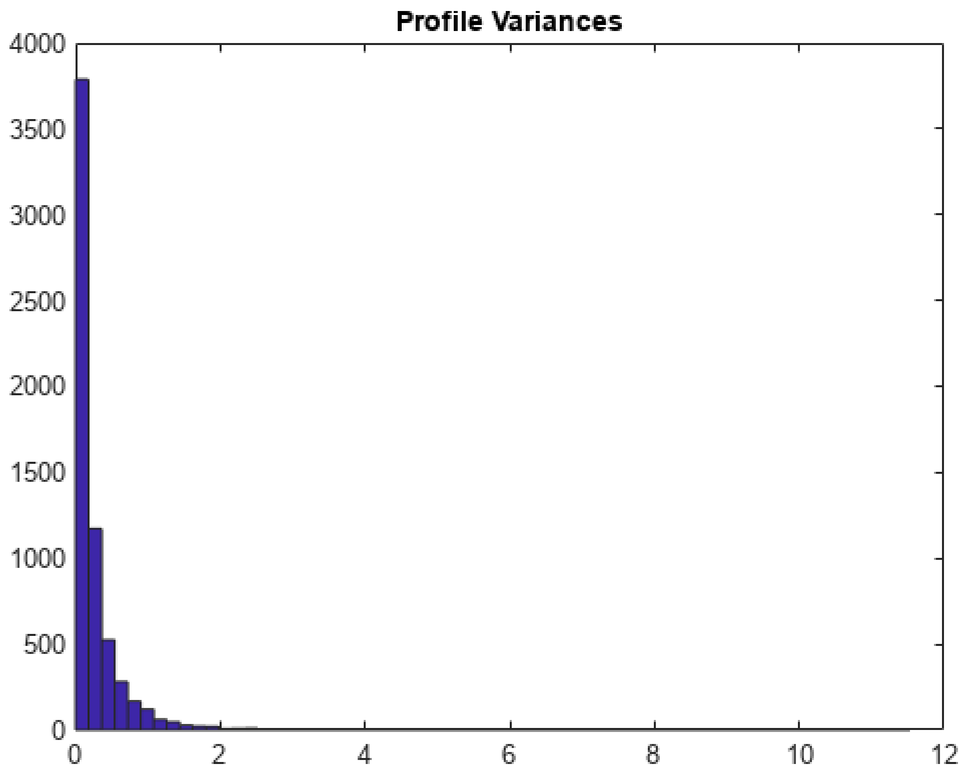
```
load yeastdata
```

This MAT-file includes three variables, which are added to the MATLAB® workspace:

- `yeastvalues` - A matrix of gene expression data
- genes - A cell array of GenBank® accession numbers for labeling the rows in yeastvalues
- times - A vector of time values for labeling the columns in yeastvalues

Calculate the variance of expression profiles for yeast data as gene expression changes during the metabolic shift from fermentation to respiration. And display a histogram of the data.

```
range = exprprofvar(yeastvalues,'ShowHist',true);
```



Version History

Introduced before R2006a

See Also

`exprprofrange` | `generangefilter` | `genevarfilter`

exprWrite

Class: `bioma.ExpressionSet`
Package: `bioma`

Write expression values in ExpressionSet object to text file

Syntax

```
exprWrite(ESObj, File)
exprWrite(..., 'Delimiter', DelimiterValue, ...)
exprWrite(..., 'Precision', PrecisionValue, ...)
exprWrite(..., 'Header', HeaderValue, ...)
exprWrite(..., 'Annotated', AnnotatedValue, ...)
exprWrite(..., 'Append', AppendValue, ...)
```

Description

`exprWrite(`*`ESObj`*`, `*`File`*`)` writes the expression values in the `Expressions` element (DataMatrix object) from an ExpressionSet object to a text file, using the delimiter `\t` to separate columns. `exprWrite` writes the data starting at the first column of the first row in the destination file.

`exprWrite(..., '`*`PropertyName`*`', `*`PropertyValue`*`, ...)` calls `exprWrite` with optional properties that use property name/property value pairs. You can specify one or more properties in any order. Enclose each *`PropertyName`* in single quotation marks. Each *`PropertyName`* is case insensitive. These property name/property value pairs are as follows:

`exprWrite(..., 'Delimiter', `*`DelimiterValue`*`, ...)` specifies a delimiter symbol to use as a column separator. Default is `'\t'`.

`exprWrite(..., 'Precision', `*`PrecisionValue`*`, ...)` specifies the precision for writing the data to the text file. Default is `5`.

`exprWrite(..., 'Header', `*`HeaderValue`*`, ...)` specifies the first line of the text file. Default is the `Name` property for the DataMatrix object.

`exprWrite(..., 'Annotated', `*`AnnotatedValue`*`, ...)` controls the writing of row and column names to the text file. Choices are `true` (default) or `false`.

`exprWrite(..., 'Append', `*`AppendValue`*`, ...)` controls the appending of the expression values to *`File`* when it is an existing file. Choices are `true` or `false` (default). If `false`, `exprWrite` overwrites *`File`*.

Input Arguments

`ESObj`

Object of the `bioma.ExpressionSet` class.

Default:

`File`

Character vector specifying either a file name or a path and file name for saving the expression values. If you specify only a file name, `exprWrite` saves the file to the MATLAB Current Folder.

Default:

`DelimiterValue`

Character vector specifying a delimiter symbol to use as a matrix column separator. Typical choices are:

- `' '`
- `'\t'` (default)
- `','`
- `';'`
- `'|'`

Default:

`PrecisionValue`

Precision for writing the data to the text file, specified by either:

- Positive integer specifying the number of significant digits
- C-style format character vector starting with `%`, such as `'%6.5f'`

Default: 5

`HeaderValue`

Character vector specifying the first line of the text file. Default is the `Name` property for the DataMatrix object.

Default:

`AnnotatedValue`

Controls the writing of row and column names to the text file. Choices are `true` (default) or `false`.

Default:

`AppendValue`

Controls the appending of the expression values to *`File`* when it is an existing file. Choices are `true` or `false` (default). If `false`, `exprWrite` overwrites *`File`*.

Default:

Examples

Construct an ExpressionSet object, `ESObj`, as described in the “Examples” on page 1-0 section of the `bioma.ExpressionSet` class reference page. Write the expression values in the ExpressionSet object to a text file:

```
% Write expression values to text file
exprWrite(ESObj, 'myexpressiondata.txt')
```

See Also

`bioma.ExpressionSet` | `bioma.data.ExptData` | `DataMatrix` | `dmwrite`

Topics

"Managing Gene Expression Data in Objects"

exptData

Class: `bioma.ExpressionSet`
Package: `bioma`

Retrieve or set experiment data in ExpressionSet object

Syntax

```
ExptDataObj = exptData(ESObj)
NewESObj = exptData(ESObj, NewExptDataObj)
```

Description

`ExptDataObj` `= exptData(`*`ESObj`*`)` returns the ExptData object stored in an ExpressionSet object.

`NewESObj` `= exptData(`*`ESObj`*`,` *`NewExptDataObj`*`)` replaces the ExptData object in *`ESObj`*, an ExpressionSet object, with *`NewExptDataObj`*, a new ExptData object, and returns *`NewESObj`*, a new ExpressionSet object.

Input Arguments

`ESObj`

Object of the `bioma.ExpressionSet` class.

Default:

`NewExptDataObj`

Object of the `bioma.data.ExptData` class.

Default:

Output Arguments

`ExptDataObj`

Object of the `bioma.data.ExptData` class.

`NewESObj`

Object of the `bioma.ExpressionSet` class, returned after replacing the ExptData object.

Examples

Construct an ExpressionSet object, `ESObj`, as described in the “Examples” on page 1-0 section of the `bioma.ExpressionSet` class reference page. Retrieve the ExptData object stored in the ExpressionSet object:

```
% Retrieve the ExptData object
NewEDObj = exptData(ESObj);
```

See Also

`bioma.ExpressionSet` | `bioma.data.ExptData` | `DataMatrix` | `featureData` | `sampleData`

Topics

“Managing Gene Expression Data in Objects”

exptInfo

Class: `bioma.ExpressionSet`
Package: `bioma`

Retrieve or set experiment information in ExpressionSet object

Syntax

```
MIAMEObj = exptInfo(ESObj)
NewESObj = exptInfo(ESObj, NewMIAMEObj)
```

Description

`MIAMEObj = exptInfo(ESObj)` returns a MIAME object containing experiment information from an ExpressionSet object.

`NewESObj = exptInfo(ESObj, NewMIAMEObj)` replaces the MIAME object in *`ESObj`*, an ExpressionSet object, with *`NewMIAMEObj`*, a new MIAME object, and returns *`NewESObj`*, a new ExpressionSet object.

Input Arguments

`ESObj`

Object of the `bioma.ExpressionSet` class.

Default:

`NewMIAMEObj`

Object of the `bioma.data.MIAME` class.

Default:

Output Arguments

`MIAMEObj`

Object of the `bioma.data.MIAME` class.

`NewESObj`

Object of the `bioma.ExpressionSet` class, returned after replacing the MIAME object.

Examples

Construct an ExpressionSet object, `ESObj`, as described in the “Examples” on page 1-0 section of the `bioma.ExpressionSet` class reference page. Retrieve the MIAME object stored in the ExpressionSet object:

```
% Retrieve the MIAME object
NewMIAMEObj = exptInfo(ESObj);
```

References

[1] Brazma, A., Hingamp, P., Quackenbush, J., Sherlock, G., Spellman, P., Stoeckert, C., et al. (2001). Minimum information about a microarray experiment (MIAME)—toward standards for microarray data. Nat Genet. 29, 365-371.

See Also

`bioma.ExpressionSet` | `bioma.data.MIAME`

Topics

"Managing Gene Expression Data in Objects"

fastainfo

Return information about FASTA file

Syntax

```
info = fastainfo(file)
info = fastainfo(file,'TimeOut',TO)
```

Description

`info = fastainfo(file)` returns a structure `info` containing summary information about the FASTA `file`.

`info = fastainfo(file,'TimeOut',TO)` sets the connection timeout (in seconds) to read data from a remote file or URL.

Examples

Get Summary Information about FASTA File

```
info = fastainfo('ex1ref.fasta')
```

```
info = struct with fields:
           Filename: 'ex1ref.fasta'
           FilePath: 'B:\matlab\toolbox\bioinfo\bioinfodata'
        FileModDate: '07-May-2010 16:12:06'
           FileSize: 1612
    NumberOfEntries: 1
             Header: 'Reference for ex1'
             Length: 1569
```

Input Arguments

`file` — FASTA file
character vector | string | character array

FASTA file, specified as one of the following:

- Character vector or string specifying a FASTA file name, path and name of a FASTA file, or URL pointing to a FASTA file. If you specify only a file name, the file must be on the MATLAB search path or in the current folder.
- Character array containing the text of a FASTA file.

Data Types: `char` | `string`

`TO` — Connection timeout
5 (default) | positive scalar

Connection timeout to read data from a remote file, specified as a positive scalar. For details, see here.

Data Types: `double`
Complex Number Support: Yes

Output Arguments

`info` — Summary information about FASTA file
structure

Summary information about the FASTA file, returned as a structure. The structure contains the following fields.

Field	Description
`Filename`	Name of the file.
`FilePath`	Path to the file.
`FileModDate`	Modification date of the file.
`FileSize`	Size of the file in bytes.
`NumberOfEntries`	Number of sequence entries in the file.
`Header`	If `file` contains only one sequence, then this is a character vector containing the header information from the FASTA file. Otherwise, this field is empty.
`Length`	If `file` contains only one sequence, then this is a scalar specifying the length of the sequence. Otherwise, this field is empty.

Version History

Introduced in R2009b

See Also

`fastaread` | `fastqinfo` | `fastawrite` | `fastqread` | `fastqwrite`

Topics
"Data Import"

fastaread

Read data from FASTA file

Syntax

```
fastaStruct = fastaread(file)
fastaStruct = fastaread(file,Name=Value)
[header,sequence] = fastaread( ___ )
```

Description

`fastaStruct = fastaread(file)` returns the sequence data from the input FASTA `file` as a structure.

`fastaStruct = fastaread(file,Name=Value)` uses additional options specified by one or more name-value arguments. For example, `seqdata = fastaread(fastafile,IgnoreGaps=true)` removes any gap symbol (`-` or `.`) from the sequences.

`[header,sequence] = fastaread( ___ )` returns the sequence data as separate variables: `header` and `sequence`. You can specify any of the input argument combinations in the previous syntaxes. If the file contains multiple sequences, `header` and `sequences` are cell arrays of sequence header and nucleotide or amino acid sequence information.

Examples

Read sequence data from FASTA files

Read the nucleotide sequence information of the human p53 tumor gene.

```
p53nt = fastaread("p53nt.txt")
```

```
p53nt = struct with fields:
      Header: 'gi|8400737|ref|NM_000546.2| Homo sapiens tumor protein p53 (Li-Fraumeni syndrome)
    Sequence: 'ACTTGTCATGGCGACTGTCCAGCTTTGTGCCAGGAGCCTCGCAGGGGTTGATGGGATTGGGGTTTTCCCCTCCCATGTGCT
```

Read the amino acid sequence information of p53 protein.

```
p53aa = fastaread("p53aa.txt")
```

```
p53aa = struct with fields:
      Header: 'gi|8400738|ref|NP_000537.2| tumor protein p53 [Homo sapiens]'
    Sequence: 'MEEPQSDPSVEPPLSQETFSDLWKLLPENNVLSPLPSQAMDDLMLSPDDIEQWFTEDPGPDEAPRMPEAAPRVAPAPAAPT
```

Read a block of entries from the 5th to 10th sequences from a FASTA file ignoring gaps from each sequence.

```
pf2 = fastaread('pf00002.fa',BlockRead=[5 10],IgnoreGaps=true)
```

```
pf2=6×1 struct array with fields:
    Header
```

Sequence

Input Arguments

`file` — Name of FASTA file or sequence information
character vector | character array | string scalar

Name of a FASTA-formatted file or sequence information, specified as a character vector, character array, or string scalar.

You specify either of the following:

- File name, a path and file name, or a URL pointing to a file. The referenced file is a FASTA-formatted file (ASCII text file). If you specify only a file name, that file must be on the MATLAB search path or in the MATLAB Current Folder.
- MATLAB character array that contains the text of a FASTA-formatted file.

A FASTA-formatted file begins with a right angle bracket (`>`) and a single line description. Following this description is the sequence information as a series of lines. Sequences must use the standard IUB/IUPAC amino acid and nucleotide letter codes.

For a list of codes, see `aminolookup` and `baselookup`.

Data Types: `char` | `string`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: `seqdata = fastaread(fastafile,TrimHeaders=true,TimeOut=10)`

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `seqdata = fastaread(fastafile,'TrimHeaders',true,'TimeOut',10)`

`IgnoreGaps` — Flag to remove gap symbols
`false` or `0` (default) | `true` or `1`

Flag to remove any gap symbols (`-` or `.`) from the sequences, specified as a logical `1` (`true`) or `0` (`false`).

`BlockRead` — Sequence entry or blocks to read from input file
positive integer | `1`-by-`2` of positive integers

Sequence entry or blocks to read from the input file with multiple sequences, specified as a positive integer or `1`-by-`2` vector of positive integers.

Specify a scalar positive integer *n* to read in the *n*th entry in the file.

Specify a two-element vector `[`*`m1 m2`*`]` to read in a block of entries starting at the *m1* entry and ending at the *m2* entry. Use `Inf` for *m2* to read all entries in the file starting at *m1*.

Data Types: `double`

TrimHeaders — Flag to trim header after first white space
false or 0 (default) | true or 1

Flag to trim the header after the first white space, specified as a logical 1 (true) or 0 (false). White space characters include a space (char(32)) and a tab (char(9)).

TimeOut — Connection time out to read from remote EMBL-EBI file
5 (default) | positive scalar

Connection time out in seconds to read from a remote EMBL-EBI file, specified as a positive scalar. For details, see here.

Data Types: double

Output Arguments

fastaStruct — Sequence data
structure

Sequence data, returned as a structure. The structure contains the following fields:

Field	Description
Header	Header information.
Sequence	Single letter-code representation of a nucleotide or amino acid sequence.

header — Sequence header information
character vector | cell array of character vectors

Sequence header information, returned as a character vector or cell array of character vectors.

Data Types: char | cell

sequence — Single letter-coded nucleotide or amino acid sequences
character vector | cell array of character vectors

Single letter-coded nucleotide or amino acid sequences, returned as a character vector or cell array of character vectors.

Data Types: char | cell

Version History

Introduced before R2006a

See Also

aminolookup | baselookup | BioIndexedFile | emblread | fastainfo | fastawrite | fastqinfo | fastqread | fastqwrite | genbankread | genpeptread | multialignread | saminfo | samread | seqprofile | seqviewer

fastawrite

Write to file using FASTA format

Syntax

```
fastawrite(File, Data)
fastawrite(File, Header, Sequence)
```

Arguments

File	Character vector or string specifying either a file name or a path and file name for saving the FASTA-formatted data. If you specify only a file name, `fastawrite` saves the file to the MATLAB Current Folder. If you specify an existing file, `fastawrite` appends the data to the file, instead of overwriting the file.
Data	Any of the following: • Character vector or string containing a sequence • MATLAB structure containing the fields `Header` and `Sequence` • MATLAB structure containing sequence information from the GenBank or GenPept database, such as returned by `genbankread`, `getgenbank`, `genpeptread`, or `getgenpept`. • Character array, where each row is a sequence.
Header	Character vector or string containing header information about the sequence. This text appears in the header of the FASTA-formatted file, *File*.
Sequence	Character vector or string containing an amino acid or nucleotide sequence using the standard IUB/IUPAC letter or integer codes. For a list of valid characters, see Amino Acid Lookup or Nucleotide Lookup.

Description

`fastawrite(`*File*`,` *Data*`)` writes the contents of *Data* to *File*, a FASTA-formatted file. If you specify an existing FASTA-formatted file,`fastawrite` appends the data to the file, instead of overwriting the file. For the FASTA-format specifications, visit https://www.ncbi.nlm.nih.gov/BLAST/fasta.shtml.

`fastawrite(`*File*`,` *Header*`,` *Sequence*`)` writes the specified header and sequence information to *File*, a FASTA-formatted file.

Tip To append FASTA-formatted data to an existing file, simply specify that file name. `fastawrite` adds the data to the end of the file.

If you are using `fastawrite` in a script, you can disable the append warning message by entering the following command lines before the `fastawrite` command:

```
warnState = warning %Save the current warning state
warning('off','Bioinfo:fastawrite:AppendToFile');
```

Then enter the following command line after the `fastawrite` command:

```
warning(warnState) %Reset warning state to previous settings
```

Examples

Example 1.6. Writing a Coding Region to a FASTA-Formatted File

1 Retrieve the sequence for the human p53 gene from the GenBank database.

```
seq = getgenbank('NM_000546');
```

2 Read the coordinates of the coding region in the CDS line.

```
start = seq.CDS.indices(1)

start =

   198

stop = seq.CDS.indices(2)

stop =

   1379
```

3 Extract the coding region.

```
codingSeq = seq.Sequence(start:stop);
```

4 Write the coding region to a FASTA-formatted file, specifying `Coding region for p53` for the Header in the file, and `p53coding.txt` for the file name.

```
fastawrite('p53coding.txt','Coding region for p53',codingSeq);
```

Example 1.7. Saving Multiple Sequences to a FASTA-Formatted File

1 Write two nucleotide sequences to a MATLAB structure containing the fields `Header` and `Sequence`.

```
data(1).Sequence = 'ACACAGGAAA';
data(1).Header = 'First sequence';
data(2).Sequence = 'ACGTCAGGTC';
data(2).Header = 'Second sequence';
```

2 Write the sequences to a FASTA-formatted file, specifying `my_sequences.txt` for the file name.

```
fastawrite('my_sequences.txt', data)
```

3 Display the FASTA-formatted file, `my_sequences.txt`.

```
type('my_sequences.txt')

>First sequence
ACACAGGAAA

>Second sequence
ACGTCAGGTC
```

Example 1.8. Appending Sequences to a FASTA-Formatted File

1 If you haven't already done so, create the FASTA-formatted file, `my_sequences.txt`, described previously.

2 Append a third sequence to the file.

```
fastawrite('my_sequences.txt','Third sequence','TACTGACTTC')
```

3 Display the FASTA-formatted file, `my_sequences.txt`.

```
type('my_sequences.txt')

>First sequence
ACACAGGAAA

>Second sequence
ACGTCAGGTC

>Third sequence
TACTGACTTC
```

Version History

Introduced before R2006a

See Also

`fastainfo` | `fastaread` | `fastqinfo` | `fastqread` | `fastqwrite` | `genbankread` | `genpeptread` | `getgenbank` | `getgenpept` | `multialignwrite` | `saminfo` | `samread` | `seqviewer` | `sffinfo` | `sffread`

fastqinfo

Return information about FASTQ file

Syntax

`InfoStruct = fastqinfo(File)`

Description

`InfoStruct = fastqinfo(File)` returns a MATLAB structure containing summary information about a FASTQ-formatted file.

Input Arguments

`File`

Character vector or string specifying a file name or path and file name of a FASTQ-formatted file. If you specify only a file name, that file must be on the MATLAB search path or in the current folder.

Default:

Output Arguments

`InfoStruct`

MATLAB structure containing summary information about a FASTQ-formatted file. The structure contains the following fields.

Field	Description
`Filename`	Name of the file.
`FilePath`	Path to the file
`FileModDate`	Modification date of the file.
`FileSize`	Size of the file in bytes.
`NumberOfEntries`	Number of sequence reads in the file.

Examples

Return a summary of the contents of a FASTQ file:

```
info = fastqinfo('SRR005164_1_50.fastq')

info = 

           Filename: 'SRR005164_1_50.fastq'
           FilePath: 'D:\2010_08_24_h11m43s32_job6027_pass\matlab\toolbox\bioinfo\biodemos'
        FileModDate: '03-Mar-2009 14:21:51'
```

```
          FileSize: 16702
     NumberOfEntries: 50
```

Version History

Introduced in R2009b

See Also

`fastqread` | `fastqwrite` | `fastainfo` | `fastaread` | `fastawrite` | `sffinfo` | `sffread` | `saminfo` | `samread` | `BioIndexedFile` | `BioRead`

Topics

"Working with Illumina/Solexa Next-Generation Sequencing Data"

External Websites

https://www.ncbi.nlm.nih.gov/Traces/sra/sra.cgi?cmd=show&f=main&m=main&s=main

fastqread

Read data from FASTQ file

Syntax

```
FASTQStruct = fastqread(File)
[Header, Sequence] = fastqread(File)
[Header, Sequence, Qual] = fastqread(File)

fastqread(..., 'Blockread', BlockreadValue, ...)
fastqread(..., 'HeaderOnly', HeaderOnlyValue, ...)
fastqread(..., 'TrimHeaders', TrimHeadersValue, ...)
```

Description

`FASTQStruct = fastqread(File)` reads a FASTQ-formatted file and returns the data in a MATLAB array of structures.

`[Header, Sequence] = fastqread(File)` returns only the header and sequence data in two separate variables.

`[Header, Sequence, Qual] = fastqread(File)` returns the data in three separate variables.

`fastqread(..., 'PropertyName', PropertyValue, ...)` calls `fastqread` with optional properties that use property name/property value pairs. You can specify one or more properties in any order. Enclose each *PropertyName* in single quotation marks. Each *PropertyName* is case insensitive. These property name/property value pairs are as follows:

`fastqread(..., 'Blockread', BlockreadValue, ...)` reads a single sequence entry or block of sequence entries from a FASTQ-formatted file containing multiple sequences.

`fastqread(..., 'HeaderOnly', HeaderOnlyValue, ...)` specifies whether to return only the header information.

`fastqread(..., 'TrimHeaders', TrimHeadersValue, ...)` specifies whether to trim the header to the first white space.

Input Arguments

`File`

Either of the following:

- Character vector or string specifying a file name or path and file name of a FASTQ-formatted file. If you specify only a file name, that file must be on the MATLAB search path or in the MATLAB Current Folder.
- MATLAB character array that contains the text of a FASTQ-formatted file.

Default:

BlockreadValue

Scalar or vector that controls the reading of a single sequence entry or block of sequence entries from a FASTQ-formatted file containing multiple sequences. Enter a scalar *N* to read the *N*th entry in the file. Enter a 1-by-2 vector `[M1, M2]` to read a block of entries starting at the *M1* entry and ending at the *M2* entry. To read all remaining entries in the file starting at the *M1* entry, enter a positive value for *M1* and enter `Inf` for *M2*.

Default:

HeaderOnlyValue

Specifies whether to return only the header information. Choices are `true` or `false` (default).

Default:

TrimHeadersValue

Specifies whether to trim the header after the first white space character. White space characters include a space (char(32)) and a tab (char(9)). Choices are `true` or `false` (default).

Default:

Output Arguments

FASTQStruct

Array of structures containing information from a FASTQ-formatted file. There is one structure for each sequence read or entry in the file. Each structure contains the following fields.

Field	Description
`Header`	Header information.
`Sequence`	Single letter-code representation of a nucleotide sequence.
`Quality`	ASCII representation of per-base quality scores for a nucleotide sequence.

Header

Variable containing header information or, if the FASTQ-formatted file contains multiple sequences, a cell array containing header information.

Sequence

Variable containing sequence information or, if the FASTQ-formatted file contains multiple sequences, a cell array containing sequence information.

Qual

Variable containing quality information or, if the FASTQ-formatted file contains multiple sequences, a cell array containing quality information.

Examples

Read a FASTQ file into an array of structures:

```
% Read the contents of a FASTQ-formatted file into
% an array of structures
reads = fastqread('SRR005164_1_50.fastq')

reads =

1x50 struct array with fields:
    Header
    Sequence
    Quality
```

Read a FASTQ file into three separate variables:

```
% Read the contents of a FASTQ-formatted file into
% three separate variables
[headers,seqs,quals] = fastqread('SRR005164_1_50.fastq');
```

Read a block of entries from a FASTQ file:

```
% Read the contents of reads 5 through 10 into
% an array of structures
reads_5_10 = fastqread('SRR005164_1_50.fastq', 'blockread', [5 10])

1x6 struct array with fields:
    Header
    Sequence
    Quality
```

More About

FASTQ-file Format

A FASTQ-formatted file contains nucleotide sequence and quality information on four lines:

- **Line 1** — Header information prefixed with an `@` symbol
- **Line 2** — Nucleotide sequence
- **Line 3** — Header information prefixed with a `+` symbol
- **Line 4** — ASCII representation of per-base quality scores for the nucleotide sequence using Phred or Solexa encoding

Version History
Introduced in R2009b

See Also
`fastqinfo` | `fastqwrite` | `fastainfo` | `fastaread` | `fastawrite` | `sffinfo` | `sffread` | `saminfo` | `samread` | `BioIndexedFile` | `BioRead`

Topics
"Working with Illumina/Solexa Next-Generation Sequencing Data"

External Websites
https://www.ncbi.nlm.nih.gov/Traces/sra/sra.cgi?cmd=show&f=main&m=main&s=main

fastqwrite

Write to file using FASTQ format

Syntax

```
fastqwrite(File, FASTQStruct)
fastqwrite(File, Header, Sequence, Qual)
```

Description

`fastqwrite(File, FASTQStruct)` writes the contents of a MATLAB structure or array of structures to a FASTQ-formatted file. If you specify an existing FASTQ-formatted file, `fastqwrite` appends the data to the file, instead of overwriting the file.

`fastqwrite(File, Header, Sequence, Qual)` writes header, sequence, and quality information to a FASTQ-formatted file.

Tip To append FASTQ-formatted data to an existing file, simply specify that file name. `fastqwrite` adds the data to the end of the file.

If you are using `fastqwrite` in a script, you can disable the append warning message by entering the following command lines before the `fastqwrite` command:

```
warnState = warning %Save the current warning state
warning('off','Bioinfo:fastqwrite:AppendToFile');
```

Then enter the following command line after the `fastqwrite` command:

```
warning(warnState) %Reset warning state to previous settings
```

Input Arguments

`File`

Character vector or string specifying either a file name or a path and file name for saving the FASTQ-formatted data. If you specify only a file name, `fastqwrite` saves the file to the MATLAB Current Folder. If you specify an existing file, `fastqwrite` appends the data to the file, instead of overwriting the file.

Default:

`FASTQStruct`

MATLAB structure or array of structures containing the fields `Header`, `Sequence`, and `Quality`, such as returned by `fastqread`.

Default:

Header

Character vector or string containing header information about the nucleotide sequence. This text appears in the header of the FASTQ-formatted file, *File*.

Default:

Sequence

Character vector or string containing a nucleotide sequence using the standard IUB/IUPAC letter or integer codes. For a list of valid characters, see Amino Acid Lookup or Nucleotide Lookup.

Default:

Qual

Character vector or string containing ASCII representation of per-base quality scores for a nucleotide sequence.

Default:

Examples

Write multiple sequences to a FASTQ file from an array of structures:

```
% Read the contents of a FASTQ-formatted file into
% an array of structures
reads = fastqread('SRR005164_1_50.fastq');
% Create another array of structures for the first five reads
reads5 = reads(1:5);
% Write the first five reads to a separate FASTQ-formatted file
fastqwrite('fiveReads.fastq', reads5)
```

Write a single sequence to a FASTQ file from separate variables:

```
% Create separate variables for the header, sequence, and
% quality information of a nucleotide sequence
h = 'MYSEQ-000_1_1_1_953_493';
s = 'GTTACCATGATGTTATTTCTTCATTTGGAGGTAAAA';
q = ']]]]]]]]]]]]]]]]]]]]]]T]]]]RJRZTQLOA';
% Write the information to a FASTQ-formatted file
fastqwrite('oneRead.fastq', h, s, q)
```

More About

FASTQ-file Format

A FASTQ-formatted file contains nucleotide sequence and quality information on four lines:

- **Line 1** — Header information prefixed with an @ symbol
- **Line 2** — Nucleotide sequence
- **Line 3** — Header information prefixed with a + symbol
- **Line 4** — ASCII representation of per-base quality scores for the nucleotide sequence using Phred or Solexa encoding

Version History

Introduced in R2009b

See Also

`fastqinfo` | `fastqread` | `fastainfo` | `fastaread` | `fastawrite` | `sffinfo` | `sffread` | `saminfo` | `samread` | `BioIndexedFile` | `BioRead`

Topics

"Working with Illumina/Solexa Next-Generation Sequencing Data"

External Websites

https://www.ncbi.nlm.nih.gov/Traces/sra/sra.cgi?cmd=show&f=main&m=main&s=main

featurecount

Compute the number of reads mapped to genomic features

Syntax

```
T = featurecount(GTFfile,Inputfile)
[T,S] = featurecount(___)
[___] = featurecount(___,Name,Value)
```

Description

`T = featurecount(GTFfile,Inputfile)` counts the number of reads in the BAM-formatted or SAM-formatted file `Inputfile` that map onto genomic features as specified in the GTF-formatted file `GTFfile`. `GTFfile` specifies the annotation file. `Inputfile` specifies the names of the BAM or SAM files to consider. The output `T` is a table where rows correspond to features and columns correspond to the input files. The elements of the table consist of the number of reads mapping to each feature for a given input file.

`[T,S] = featurecount(___)` returns a table `S` with a summary of assigned and unassigned alignment entries. If multiple input files are provided, each file is associated with a column.

`[___] = featurecount(___,Name,Value)` uses additional options specified by one or more `Name,Value` pair arguments.

Examples

Count the number of reads mapped to genomic features

Count reads from a sample SAM file that map to the features included in a GTF file. By default, `featurecount` maps the reads to exons, and summarizes the total number of reads at the gene level.

```
[t,s] = featurecount('Dmel_BDGP5_nohc.gtf','rnaseq_sample1.sam');
```

```
Processing GTF file Dmel_BDGP5_nohc.gtf ...
Processing SAM file rnaseq_sample1.sam ...
Processing reference chr2L ...
Processing reference chr2R ...
Processing reference chr3L ...
Processing reference chr3R ...
Processing reference chr4 ...
Processing reference chrX ...
Done.
```

Display the first 10 rows of count data.

```
t(1:10,:)
```

```
ans=10×3 table
          ID          Reference    rnaseq_sample1
    ______________    _________    ______________
    {'FBgn0002121'}    {'chr2L'}          9
    {'FBgn0067779'}    {'chr2L'}          2
    {'FBgn0005278'}    {'chr2L'}          4
    {'FBgn0031220'}    {'chr2L'}          4
    {'FBgn0025683'}    {'chr2L'}         13
    {'FBgn0053635'}    {'chr2L'}          2
    {'FBgn0016977'}    {'chr2L'}         22
    {'FBgn0086902'}    {'chr2L'}         27
    {'FBgn0031245'}    {'chr2L'}          2
    {'FBgn0024352'}    {'chr2L'}          2
```

The `ID` column contains the names of features (genes in this example). The `Reference` column lists the names of reference sequences for the features. The third column contains the total number of reads mapped to each feature for a given SAM file, that is, rnaseq_sample1.sam. By default, the table shows only those features (rows) and SAM files (columns) with non-zero read counts. Set 'ShowZeroCounts' to true to include those rows and columns with all zero counts in the output table.

`s` contains the summary statistics of assigned and unassigned reads from each SAM file. For instance, the `TotalEntries` row indicates the total number of alignment records from the given SAM file, and the `Assigned` row includes the number of reads that are assigned to features in the GTF file. For details about each row, refer to the Output Arguments section of the reference page.

```
s
```

```
s=9×1 table
                                    rnaseq_sample1
                                    ______________

    TotalEntries                        33354
    Assigned                            16399
    Unassigned_ambiguous                  167
    Unassigned_filtered                     0
    Unassigned_lowMappingQuality            0
    Unassigned_multiMapped                  0
    Unassigned_noFeature                16788
    Unassigned_supplementary                0
    Unassigned_unmapped                     0
```

Count reads without any summarization and disable displaying the progress messages.

```
[t2,s2] = featurecount('Dmel_BDGP5_nohc.gtf','rnaseq_sample1.sam', ...
                        'Summarization',false,'Verbose',false);
```

Notice the ID column of the output table now reports the feature attribute followed by the start and stop positions of each feature, separated by underscores.

```
t2(1:10,:)
```

```
ans=10×3 table
                 ID                 Reference    rnaseq_sample1
    _____________________________    _________    ______________

    {'FBgn0002121_12286_12928'  }    {'chr2L'}          3
    {'FBgn0002121_13683_14874'  }    {'chr2L'}          1
    {'FBgn0002121_14933_15711'  }    {'chr2L'}          3
    {'FBgn0067779_67044_67507'  }    {'chr2L'}         2
    {'FBgn0005278_108588_108809'}    {'chr2L'}         1
    {'FBgn0005278_110755_110877'}    {'chr2L'}         1
    {'FBgn0005278_112690_113369'}    {'chr2L'}         1
    {'FBgn0031220_117079_117759'}    {'chr2L'}         2
    {'FBgn0031220_118361_118874'}    {'chr2L'}         1
    {'FBgn0031220_118931_119076'}    {'chr2L'}         1
```

You can choose how to assign a read to a particular feature when the read overlaps with multiple features by setting the 'OverlapMethod' option. For instance, if you want to count a read only if it fully overlaps a feature, use the 'full' option.

```
[tFull, sFull] = featurecount('Dmel_BDGP5_nohc.gtf','rnaseq_sample1.sam', ...
                              'OverlapMethod','full','Verbose',false);
```

If you have paired-end data, you can count reads as fragments.

```
[tFrag,sFrag] = featurecount('Dmel_BDGP5_nohc.gtf','rnaseq_sample1.sam', ...
                             'CountFragments',true,'Verbose',false);
```

You can also count fragments from multiple SAM files.

```
[t2,s2] = featurecount('Dmel_BDGP5_nohc.gtf',...
          {'rnaseq_sample1.sam','rnaseq_sample2.sam'},'CountFragments',true, ...
          'Verbose',false);
```

Use the following options to count paired-end reads where at least one of the read mates are above a certain mapping quality threshold.

```
[t3,s3] = featurecount('Dmel_BDGP5_nohc.gtf',...
          'rnaseq_sample1.sam','CountFragments',true,'MinMappingQuality',20, ...
          'Verbose',false);
```

If the reads come from any strand-specific assay, you can specify such strand specificity during counting. For instance, if the protocol is stranded, the strand of the feature is compared with the strand of the read. Then only those reads that have the same strand as the overlapped feature are counted.

```
[t4,s4] = featurecount('Dmel_BDGP5_nohc.gtf',...
          'rnaseq_sample1.sam','StrandSpecificity','stranded','Verbose',false);
```

Input Arguments

`GTFfile` — GTF-formatted file name

character vector | string

GTF-formatted file name, specified as a character vector or string.

Example: `'Dmel_BDGP5_nohc.gtf'`

`Inputfile` — BAM-formatted or SAM-formatted file name

character vector | string | string vector | cell array of character vectors

BAM-formatted or SAM-formatted file name, specified as a character vector, string, string vector, or cell array of character vectors.

Example: `'rnaseq_sample1.sam'`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `'CountFragments',true` specifies to count reads as pairs of mates.

`Feature` — Feature type
`'exon'` (default) | character vector | string

Feature type, specified as a character vector or string. This is used to decide what feature to consider from the GTF file. Default is `'exon'`.

`Metafeature` — Attribute type
`'gene_id'` (default) | character vector | string

Attribute type, specified as a character vector or string. This is used to decide what attribute to consider from the GTF file for grouping features into metafeatures and summarizing the read count.

`Summarization` — Boolean variable indicating whether to summarize at the metafeature level
`true` (default) | `false`

Boolean variable indicating whether to summarize at the metafeature level, specified as `true` or `false`.

Default is `true`, meaning the function groups features into metafeatures and reports the read counts for metafeatures.

`Alias` — Name of file containing aliases of reference names
character vector | string

Name of file containing aliases of reference names, specified as a character vector or string. The file must be a tab-delimited file where the first column corresponds to the reference names used in the GTF file, and the second column corresponds to the reference names used in the input file(s). The names are case-sensitive. It is necessary to include only the reference names that are different in the GTF file and the input file. The file must contain only one alias term for any reference listed in the input file. By default, the reference names in the GTF file and those in the input files are assumed to be the same.

`CountFragments` — Boolean variable indicating whether to count reads as pairs of mates
`false` (default) | `true`

Boolean variable indicating whether to count reads as fragments, specified as `true` or `false`. Paired-end reads must have the same ID for the field `QNAME` in the input file, and the mutual order of mates is inferred by the appropriate bit in the `FLAG` field within the input file. Reads that have no valid mate either because the mate is unmapped or filtered out by input criteria are still counted if they satisfy the overlapping criteria.

Default is `false`, that is, the reads are counted as single-end reads, and their pairing information is ignored.

`StrandSpecificity` — Strand specificity of sequencing protocol
`'unstranded'` (default) | `'stranded'` | `'reverse'`

Strand specificity of the sequencing protocol, specified as `'unstranded'` (default), `'stranded'`, or `'reverse'`.

- If `'unstranded'`, the strand of the reads (or fragments) is ignored.
- If `'stranded'`, the strand of the reads (or fragments) is considered, and only those having the same strand as the feature they overlap are counted.
- If `'reverse'`, the opposite direction of the strand of the reads (or fragments) is considered, and only those having the opposite strand as the feature they overlap are counted.

When counting fragments (paired-end reads), the strand of the first mate is considered as the strand of the whole fragment. The mutual order of mates (first or second) is inferred from the appropriate bit in the `FLAG` field of the input file.

`MinOverlap` — Minimum number of overlapped bases required
`1` (default) | positive integer

Minimum number of overlapped bases required to assign a read to a feature, specified as a positive integer. When counting fragments, the sum of the overlaps from each end is used as the minimum number of overlapped bases.

`MinMappingQuality` — Minimum mapping quality for a given read
`0` (default) | non-negative integer

Minimum mapping quality for a given read to be considered for counting, specified as a non-negative integer. This corresponds to the `MAPQ` field in the input file. If counting fragments, at least one of the read mates must satisfy this criterion in order to be considered for counting.

`CountMultiOverlap` — Boolean variable indicating whether to count reads overlapping multiple features
`false` (default) | `true`

Boolean variable indicating whether to count reads overlapping multiple features, specified as `true` or `false` (default).

If `true`, a read (or fragment) overlapping multiple features is counted multiple times. During summarization at the metafeature level, a read (or fragment) is counted only once if it overlaps with multiple features belonging to the same metafeature as long as it does not overlap with other metafeaures.

`CountMultiMapped` — Counting option for reads having multiple mapping locations in the input file
`'primary'` (default) | `'none'` | `'all'`

Counting option for reads having multiple mapping locations in the input file, specified as `'primary'` (default), `'none'`, or `'all'`.

- If `'primary'`, only the primary alignment of a multi-mapped read is considered. The appropriate bit in the input file is used to identify primary alignments.
- If `'none'`, all alignments of a multi-mapped read are ignored. The NH tag is used to identify multi-mapped reads.

- If `'all'`, all alignments of a multi-mapped read are considered and counted multiple times.

`BothEndsMapped` — Boolean variable indicating whether a fragment must have both mates mapped
`false` (default) | `true`

Boolean variable indicating whether a fragment must have both mates mapped, specified as `true` or `false`. Mate mapping information is retrieved from the `FLAG` field in the input file. Default is `false`.

`ProperlyPaired` — Boolean variable indicating whether a fragment must be properly paired
`false` (default) | `true`

Boolean variable indicating whether a fragment must be properly paired, specified as `true` or `false`. Mate pairing information is retrieved from the `FLAG` field in the input file. Default is `false`.

`ShowZeroCounts` — Boolean variable indicating whether to report features or metafeatures with zero count
`false` (default) | `true`

Boolean variable indicating whether to report features or metafeatures with zero count for every input file in the output table, specified as `true` or `false`.

Default is `false`, that is, only rows with non-zero counts and columns with non-zero counts are included in the output table.

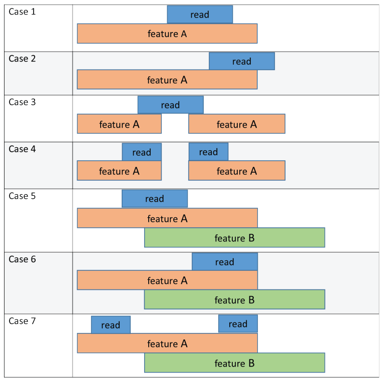
`OverlapMethod` — Method to use when assigning a given read to metafeature
`'partial'` (default) | `'full'` | `'max'` | `'hits'`

Method to use when assigning a given read to metafeature, specified as `'partial'`, `'full'`, `'max'`, or `'hits'`. If `'Summarization'` is set to `false`, then the reads are assigned to features, instead of metafeatures, based on the specified method.

In the following table, *R* refers to a read or fragment, and *M* refers to a metafeature.

Method	Description
`'partial'`	*R* is assigned to *M* if *R* overlaps (even partially) only with *M*. Otherwise *R* is considered ambiguous.
`'full'`	*R* is assigned to *M* if *R* is completely mapped only within *M*, that is, fully overlapping only M. Otherwise *R* is considered ambiguous
`'max'`	*R* is assigned to *M* if *R* satisfies the overlapping criteria only with M, or if *R* satisfies the overlapping criteria with several metafeatures but overlaps fully only with M.
`'hits'`	*R* is assigned to *M* if *R* overlaps even partially only *M*, or if *M* is the only metafeature with the highest number of features hit by *R*; otherwise *R* is considered ambiguous.

The following schematic diagram and table illustrate the outcome of these methods in conjunction with the `'CountMultiOverlap'` name-value pair argument. In the figure, the read refers to a short-read sequence from an input file, and feature A and feature B refers to features listed in a GTF file.



Each method column lists the feature that the read is assigned to based on the corresponding method. The `'CountMultiOverlap'` column indicates whether this name-value pair is set to `true` or `false` and if it has any effect in the outcome of each method.

	`'CountMultiOverlap'`	**`'partial'`**	**`'full'`**	**`'max'`**	**`'hits'`**
Case 1	No effect since the read maps only to one feature (feature A).	feature A	feature A	feature A	feature A
Case 2	No effect since the read maps only to one feature (feature A).	feature A	no feature	feature A	feature A
Case 3	No effect since the read maps only to one feature (feature A).	feature A	no feature	feature A	feature A
Case 4	No effect since the read maps only to one feature (feature A).	feature A	feature A	feature A	feature A
Case 5	`false`	ambiguous	feature A	feature A	ambiguous
	`true`	feature A, feature B	feature A	feature A	feature A, feature B
Case 6	`false`	ambiguous	ambiguous	ambiguous	ambiguous
	`true`	feature A, feature B	feature A, feature B	feature A, feature B	feature A, feature B
Case 7	`false`	Ambiguous	feature A	feature A	feature A
	`true`	feature A, feature B	feature A	feature A	feature A

no feature means that the read is not assigned to any feature. If you have specified the second output table `S`, its `Unassigned_noFeature` row is incremented by one for such occurrence. *ambiguous* means that the read is not assigned to any feature since it satisfies the overlapping criteria for multiple features, and the `Unassigned_ambiguous` row is incremented by one for such occurrence.

`UseParallel` — Boolean variable indicating whether to compute in parallel
`false` (default) | `true`

Boolean variable indicating whether to compute in parallel, specified as `true` or `false`.

In order to execute the computation in parallel, you must have Parallel Computing Toolbox. If a MATLAB parallel pool does not exist, one is automatically created when the auto-creation option is enabled in your parallel preferences. Otherwise, computation runs in the serial mode.

Default is `false`, that is, serial mode.

`Verbose` — Boolean variable indicating whether to display the progress of computation
`true` (default) | `false`

Boolean variable indicating whether to display the progress of computation, specified as `true` or `false`.

Output Arguments

T — Results containing sequence reads mapped to genomic features
table

Results containing sequence reads mapped to genomic features, returned as a table. The rows correspond to features, and columns correspond to the input files. The elements of the table consist of the number of reads mapped to each feature for a given input file. The table also reports the ID of each feature and the reference sequence for the feature.

When `'Summarization'` is set to false, the ID column of the table reports the metafeature attribute followed by the start and stop positions of each feature, separated by underscores.

S — Summary of assigned and unassigned alignment entries
table

Summary of assigned and unassigned alignment entries, returned as a table. Each column of the table corresponds to each input file provided. The table has the following rows:

Row Names	Description
`TotalEntries`	Number of records (or alignments) in the input file
`Assigned`	Number of reads or fragments that were assigned to features
`Unassigned_ambiguous`	Number of unassigned reads or fragments overlapping multiple features or metafeatures
`Unassigned_filtered`	Number of alignment records filtered by input criteria
`Unassigned_lowMappingQuality`	Number of alignment records filtered out due to low mapping quality

Row Names	Description
`Unassigned_multiMapped`	Number of alignment records not assigned because of corresponding reads mapped to multiple locations
`Unassigned_noFeature`	Number of reads or fragments not assigned to any features
`Unassigned_supplementary`	Number of alignment records not assigned because they are flagged as supplementary records for chimeric alignments
`Unassigned_unmapped`	Number of alignment records not assigned because corresponding reads are unmapped

Version History

Introduced in R2016a

Extended Capabilities

Automatic Parallel Support

Accelerate code by automatically running computation in parallel using Parallel Computing Toolbox™.

To run in parallel, set `'UseParallel'` to `true`.

For more information, see the `'UseParallel'` name-value pair argument.

See Also

`bowtie2` | `cufflinks`

Topics

"Count Features from NGS Reads"

featureData

Class: `bioma.ExpressionSet`
Package: `bioma`

Retrieve or set feature metadata in ExpressionSet object

Syntax

```
MetaDataObj = featureData(ESObj)
NewESObj = featureData(ESObj, NewMetaDataObj)
```

Description

`MetaDataObj = featureData(ESObj)` returns a MetaData object containing the feature metadata from an ExpressionSet object.

`NewESObj = featureData(ESObj, NewMetaDataObj)` replaces the feature metadata in *`ESObj`*, an ExpressionSet object, with *`NewMetaDataObj`*, and returns *`NewESObj`*, a new ExpressionSet object.

Input Arguments

`ESObj`

Object of the `bioma.ExpressionSet` class.

Default:

`NewMetaDataObj`

Object of the `bioma.data.MetaData` class, containing feature metadata, stored in two dataset arrays. The feature names and variable names in *`NewMetaDataObj`* must match the feature names and variable names in the *`MetaDataObj`* being replaced in the ExpressionSet object, *`ESObj`*.

Default:

Output Arguments

`MetaDataObj`

Object of the `bioma.data.MetaData` class, containing the feature metadata, stored in two dataset arrays.

`NewESObj`

Object of the `bioma.ExpressionSet` class, returned after replacing the MetaData object containing the feature metadata.

See Also
`bioma.ExpressionSet` | `bioma.data.MetaData` | `featureNames` | `sampleData`

Topics
"Managing Gene Expression Data in Objects"

featureNames

Class: `bioma.ExpressionSet`
Package: `bioma`

Retrieve or set feature names in ExpressionSet object

Syntax

```
FeatNames = featureNames(ESObj)
FeatNames = featureNames(ESObj, Subset)
NewESObj = featureNames(ESObj, Subset, NewFeatNames)
```

Description

`FeatNames = featureNames(ESObj)` returns a cell array of character vectors specifying all feature names in an ExpressionSet object.

`FeatNames = featureNames(ESObj, Subset)` returns a cell array of character vectors specifying a subset the feature names in an ExpressionSet object.

`NewESObj = featureNames(ESObj, Subset, NewFeatNames)` replaces the feature names specified by *`Subset`* in *`ESObj`*, an ExpressionSet object, with *`NewFeatNames`*, and returns *`NewESObj`*, a new ExpressionSet object.

Input Arguments

`ESObj`

Object of the `bioma.ExpressionSet` class.

`Subset`

One of the following to specify a subset of the feature names in an ExpressionSet object:

- Character vector or string specifying a feature name
- Cell array of character vectors or string vector specifying feature names
- Positive integer
- Vector of positive integers
- Logical vector

`NewFeatNames`

New feature names for specific feature names within an ExpressionSet object, specified by one of the following:

- Numeric vector
- Cell array of character vectors or string vector
- Character vector or string, which `featureNames` uses as a prefix for the feature names, with feature numbers appended to the prefix

- Logical `true` or `false` (default). If `true`, `featureNames` assigns unique feature names using the format `Feature1`, `Feature2`, etc.

The number of feature names in *`NewFeatNames`* must equal the number of features specified by *`Subset`*.

Output Arguments

`FeatNames`

Cell array of character vectors specifying all or some of the feature names in an ExpressionSet object. The feature names are the row names in the DataMatrix objects in the ExpressionSet object. The feature names are also the row names of the *`VarValues`* dataset array in the MetaData object in the ExpressionSet object.

`NewESObj`

Object of the `bioma.ExpressionSet` class, returned after replacing specific feature names.

See Also

`bioma.ExpressionSet` | `bioma.data.ExptData` | `DataMatrix` | `bioma.data.MetaData` | `sampleNames`

Topics

"Managing Gene Expression Data in Objects"

featureNames

Class: `bioma.data.ExptData`
Package: `bioma.data`

Retrieve or set feature names in ExptData object

Syntax

```
FeatNames = featureNames(EDObj)
FeatNames = featureNames(EDObj, Subset)
NewESObj = featureNames(EDObj, Subset, NewFeatNames)
```

Description

`FeatNames = featureNames(EDObj)` returns a cell array of character vectors specifying all feature names in an ExptData object.

`FeatNames = featureNames(EDObj, Subset)` returns a cell array of character vectors specifying a subset the feature names in an ExptData object.

`NewESObj = featureNames(EDObj, Subset, NewFeatNames)` replaces the feature names specified by *`Subset`* in *`EDObj`*, an ExptData object, with *`NewFeatNames`*, and returns *`NewEDObj`*, a new ExptData object.

Input Arguments

`EDObj`

Object of the `bioma.data.ExptData` class.

Default:

`Subset`

One of the following to specify a subset of the feature names in an ExptData object:

- Character vector specifying a feature name
- Cell array of character vectors specifying feature names
- Positive integer
- Vector of positive integers
- Logical vector

Default:

`NewFeatNames`

New feature names for specific feature names within an ExptData object, specified by one of the following:

- Numeric vector
- Character vector or cell array of character vectors
- Character vector, which `featureNames` uses as a prefix for the feature names, with feature numbers appended to the prefix
- Logical `true` or `false` (default). If `true`, `featureNames` assigns unique feature names using the format `Feature1`, `Feature2`, etc.

The number of feature names in *`NewFeatNames`* must equal the number of features specified by *`Subset`*.

Default:

Output Arguments

`FeatNames`

Cell array of character vectors specifying all or some of the feature names in an ExptData object. The feature names are the row names in the DataMatrix objects in the ExptData object.

`NewEDObj`

Object of the `bioma.data.ExptData` class, returned after replacing specific feature names.

Examples

Construct an ExptData object, and then retrieve the feature names from it:

```
% Import bioma.data package to make constructor functions
% available
import bioma.data.*
% Create DataMatrix object from .txt file containing
% expression values from microarray experiment
dmObj = DataMatrix('File', 'mouseExprsData.txt');
% Construct ExptData object
EDObj = ExptData(dmObj);
% Retrieve feature names
FNames = featureNames(EDObj);
```

See Also

`bioma.data.ExptData` | `DataMatrix` | `dmNames` | `elementNames` | `sampleNames`

Topics

"Representing Expression Data Values in ExptData Objects"

featureparse

Parse features from GenBank, GenPept, or EMBL data

Syntax

```
FeatStruct = featureparse(Features)

FeatStruct = featureparse(Features, ...'Feature', FeatureValue, ...)
FeatStruct = featureparse(Features, ...'Sequence', SequenceValue, ...)
```

Input Arguments

Features	Any of the following: • MATLAB structure with fields corresponding to GenBank, GenPept, or EMBL data, such as those returned by `genbankread`, `genpeptread`, `emblread`, `getgenbank`, `getgenpept`, or `getembl` • Character vector or character array containing the text from the Features section of a GenBank, GenPept, or EMBL-formatted file
FeatureValue	Name of a feature contained in *Features*. When specified, `featureparse` returns only the substructure that corresponds to this feature. If there are multiple features with the same *FeatureValue*, then *FeatStruct* is an array of structures.
SequenceValue	Property to control the extraction, when possible, of the sequences respective to each feature, joining and complementing pieces of the source sequence and storing them in the `Sequence` field of the returned structure, *FeatStruct*. When extracting the sequence from an incomplete CDS feature, `featureparse` uses the `codon_start` qualifier to adjust the frame of the sequence. Choices are `true` or `false` (default).

Output Arguments

FeatStruct	Output structure containing a field for every database feature. Each field name in *FeatStruct* matches the corresponding feature name in the GenBank, GenPept, or EMBL database, with the exceptions listed in the table below. Fields in *FeatStruct* contain substructures with feature qualifiers as fields. In the GenBank, GenPept, and EMBL databases, for each feature, the only mandatory qualifier is its location, which `featureparse` translates to the field `Location`. When possible, `featureparse` also translates this location to numeric indices, creating an `Indices` field. **Note** If you use the `Indices` field to extract sequence information, you may need to complement the sequences.

Description

FeatStruct = featureparse(*Features*) parses the features from *Features*, which contains GenBank, GenPept, or EMBL features. *Features* can be a:

- Character vector or string containing GenBank, GenPept, or EMBL features
- MATLAB character array including text describing GenBank, GenPept, or EMBL features
- MATLAB structure with fields corresponding to GenBank, GenPept, or EMBL data, such as those returned by `genbankread`, `genpeptread`, `emblread`, `getgenbank`, `getgenpept`, or `getembl`

FeatStruct is the output structure containing a field for every database feature. Each field name in *FeatStruct* matches the corresponding feature name in the GenBank, GenPept, or EMBL database, with the following exceptions.

Feature Name in GenBank, GenPept, or EMBL Database	Field Name in MATLAB Structure
`-10_signal`	`minus_10_signal`
`-35_signal`	`minus_35_signal`
`3'UTR`	`three_prime_UTR`
`3'clip`	`three_prime_clip`
`5'UTR`	`five_prime_UTR`
`5'clip`	`five_prime_clip`
`D-loop`	`D_loop`

Fields in *FeatStruct* contain substructures with feature qualifiers as fields. In the GenBank, GenPept, and EMBL databases, for each feature, the only mandatory qualifier is its location, which `featureparse` translates to the field `Location`. When possible, `featureparse` also translates this location to numeric indices, creating an `Indices` field.

Note If you use the `Indices` field to extract sequence information, you may need to complement the sequences.

FeatStruct = featureparse (*Features*, ...'*PropertyName*', *PropertyValue*, ...) calls `featureparse` with optional properties that use property name/property value pairs. You can specify one or more properties in any order. Each *PropertyName* must be enclosed in single quotation marks and is case insensitive. These property name/property value pairs are as follows:

FeatStruct = featureparse(*Features*, ...'Feature', *FeatureValue*, ...) returns only the substructure that corresponds to *FeatureValue*, the name of a feature contained in *Features*. If there are multiple features with the same *FeatureValue*, then *FeatStruct* is an array of structures.

FeatStruct = featureparse(*Features*, ...'Sequence', *SequenceValue*, ...) controls the extraction, when possible, of the sequences respective to each feature, joining and complementing pieces of the source sequence and storing them in the field `Sequence`. When extracting the sequence from an incomplete CDS feature, `featureparse` uses the `codon_start` qualifier to adjust the frame of the sequence. Choices are `true` or `false` (default).

Examples

Obtain Features from GenBank

Obtain all the features stored in a GeneBank file.

```
gbkStruct = genbankread('nm175642.txt');
features = featureparse(gbkStruct)
```

```
features = struct with fields:
    source: [1×1 struct]
      gene: [1×1 struct]
      exon: [1×31 struct]
       CDS: [1×1 struct]
       STS: [1×2 struct]
```

Get a subset of features from a GeneBank record. For example, obtain only the coding sequences (CDS) feature of two strains of the Influenza A virus (H5N1) from the GenBank database.

```
hk01 = getgenbank('AF509094');
vt04 = getgenbank('DQ094287');
hk01_cds = featureparse(hk01,'feature','CDS','Sequence',true);
vt04_cds = featureparse(vt04,'feature','CDS','Sequence',true);
```

Use `nt2aa` and `nwalign` to align the amino acid sequences converted from the corrresponding nucleotide sequences

```
[sc,al] = nwalign(nt2aa(hk01_cds),nt2aa(vt04_cds),'extendgap',1);
```

Use `seqinsertgaps` to copy the gaps from the aligned amino acide sequences to their corresponding nucleotide sequences to codon-align them.

```
hk01_aligned = seqinsertgaps(hk01_cds,al(1,:))
```

```
hk01_aligned = 
'caaaagcaggagattaaaatgaatccaaatcagaagataatgaccattggatcaatctgtatggtaatcggaatgattagcctggtgttacaaat
```

```
vt04_aligned = seqinsertgaps(vt04_cds,al(3,:))
```

```
vt04_aligned = 
'------------------atgaatccaaatcagaagataataaccatcggatcaatctgtatggtaactggaatagttagcttaatgttacaagt
```

Once you have code aligned the two sequences, use them as input to other functions, such as `dnds`, which calculates the synonymous and nonsynonymous substitutions rates of the codon-aligned nucleotide sequences. By setting `Verbose` to `true`, you can also display the codons considered in the computations and their amino acid translations.

```
[dn,ds] = dnds(hk01_aligned,vt04_aligned,'verbose',true)
```

```
DNDS:
Codons considered in the computations:
ATGAATCCAAATCAGAAGATAATGACCATTGGATCAATCTGTATGGTAATCGGAATGATTAGCCTGGTGTTACAAATTGGGAACATGATTTCAATA
ATGAATCCAAATCAGAAGATAATAACCATCGGATCAATCTGTATGGTAACTGGAATAGTTAGCTTAATGTTACAAGTTGGGAACATGATCTCAATA
Translations:
M  N  P  N  Q  K  I  M  T  I  G  S  I  C  M  V  I  G  M  I  S  L  V  L  Q  I  G  N  M  I  S  I
M  N  P  N  Q  K  I  I  T  I  G  S  I  C  M  V  T  G  I  V  S  L  M  L  Q  V  G  N  M  I  S  I
```

```
dn = 0.0397

ds = 0.1957
```

Version History

Introduced in R2006b

See Also

`emblread` | `genbankread` | `genpeptread` | `getgenbank` | `getgenpept`

featureVarDesc

Class: `bioma.ExpressionSet`
Package: `bioma`

Retrieve or set feature variable descriptions in ExpressionSet object

Syntax

```
DSVarDescriptions = featureVarDesc(ESObj)
NewESObj = featureVarDesc(ESObj, NewDSVarDescriptions)
```

Description

`DSVarDescriptions` `= featureVarDesc(`*`ESObj`*`)` returns a dataset array containing the feature variable names and descriptions from the MetaData object in an ExpressionSet object.

`NewESObj` `= featureVarDesc(`*`ESObj`*`,` *`NewDSVarDescriptions`*`)` replaces the feature variable descriptions in *`ESObj`*, an ExpressionSet object, with *`NewDSVarDescriptions`*, and returns *`NewESObj`*, a new ExpressionSet object.

Input Arguments

`ESObj`

Object of the `bioma.ExpressionSet` class.

Default:

`NewDSVarDescriptions`

Descriptions of the feature variable names, specified by either of the following:

- A new dataset array containing the feature variable names and descriptions. In this dataset array, each row corresponds to a variable. The first column contains the variable name, and the second column (`VariableDescription`) contains a description of the variable. The row names (variable names) must match the row names (variable names) in *`DSVarDescriptions`*, the dataset array being replaced in the MetaData object in the ExpressionSet object, *`ESObj`*.
- Cell array of character vectors containing descriptions of the feature variables. The number of elements in *`VarDesc`* must equal the number of row names (variable names) in *`DSVarDescriptions`*, the dataset array being replaced in the MetaData object in the ExpressionSet object, *`ESObj`*.

Default:

Output Arguments

`DSVarDescriptions`

A dataset array containing the feature variable names and descriptions from the MetaData object of an ExpressionSet object. In this dataset array, each row corresponds to a variable. The first column contains the variable name, and the second column (`VariableDescription`) contains a description of the variable.

`NewESObj`

Object of the `bioma.ExpressionSet` class, returned after replacing the dataset array containing the feature variable descriptions.

See Also

`bioma.ExpressionSet` | `bioma.data.MetaData` | `variableDesc`

Topics

"Managing Gene Expression Data in Objects"

featureVarNames

Class: `bioma.ExpressionSet`
Package: `bioma`

Retrieve or set feature variable names in ExpressionSet object

Syntax

```
FeatVarNames = featureVarNames(ESObj)
FeatVarNames = featureVarNames(ESObj, Subset)
NewESObj = featureVarNames(ESObj, Subset, NewFeatVarNames)
```

Description

`FeatVarNames = featureVarNames(ESObj)` returns a cell array of character vectors specifying all feature variable names in an ExpressionSet object.

`FeatVarNames = featureVarNames(ESObj, Subset)` returns a cell array of character vectors specifying a subset the feature variable names in an ExpressionSet object.

`NewESObj = featureVarNames(ESObj, Subset, NewFeatVarNames)` replaces the feature variable names specified by *`Subset`* in *`ESObj`*, an ExpressionSet object, with *`NewFeatVarNames`*, and returns *`NewESObj`*, a new ExpressionSet object.

Input Arguments

`ESObj`

Object of the `bioma.ExpressionSet` class.

Default:

`Subset`

One of the following to specify a subset of the feature variable names in an ExpressionSet object:

- Character vector or string specifying a feature variable name
- Cell array of character vectors or string vector specifying feature variable names
- Positive integer
- Vector of positive integers
- Logical vector

`NewFeatVarNames`

New feature variable names for specific feature variable names within an ExpressionSet object, specified by one of the following:

- Numeric vector

- Cell array of character vectors or string vector
- Character array
- Character vector or string, which `featureVarNames` uses as a prefix for the feature variable names, with feature variable numbers appended to the prefix
- Logical `true` or `false` (default). If `true`, `featureVarNames` assigns unique feature variable names using the format `Var1`, `Var2`, etc.

The number of feature variable names in *`NewFeatVarNames`* must equal the number of feature variable names specified by *`Subset`*.

Output Arguments

`FeatVarNames`

Cell array of character vectors specifying all or some of the feature variable names in an ExpressionSet object. The feature variable names are the column names of the *`VarValues`* dataset array. The feature variable names are also the row names of the *`VarDescriptions`* dataset array. Both dataset arrays are in the MetaData object in the ExpressionSet object.

`NewESObj`

Object of the `bioma.ExpressionSet` class, returned after replacing specific feature names.

See Also

`bioma.ExpressionSet` | `bioma.data.MetaData` | `sampleNames` | `featureNames` | `sampleVarNames`

Topics

"Managing Gene Expression Data in Objects"

featureVarValues

Class: `bioma.ExpressionSet`
Package: `bioma`

Retrieve or set feature variable data values in ExpressionSet object

Syntax

```
DSVarValues = featureVarValues(ESObj)
NewESObj = featureVarValues(ESObj, NewDSVarValues)
```

Description

`DSVarValues = featureVarValues(ESObj)` returns a dataset array containing the measured value of each variable per feature from the MetaData object of an ExpressionSet object.

`NewESObj = featureVarValues(ESObj, NewDSVarValues)` replaces the feature variable values in *`ESObj`*, an ExpressionSet object, with *`NewDSVarValues`*, and returns *`NewESObj`*, a new ExpressionSet object.

Input Arguments

`ESObj`

Object of the `bioma.ExpressionSet` class.

Default:

`NewDSVarValues`

A dataset array containing a value for each variable per feature. In this dataset array, the columns correspond to variables and rows correspond to feature. The row names (feature names) must match the row names (feature names) in *`DSVarValues`*, the dataset array being replaced in the MetaData object in the ExpressionSet object, *`ESObj`*.

Default:

Output Arguments

`DSVarValues`

A dataset array containing the measured value of each variable per feature from the MetaData object of an ExpressionSet object. In this dataset array, the columns correspond to variables and rows correspond to features.

`NewESObj`

Object of the `bioma.ExpressionSet` class, returned after replacing the dataset array containing the feature variable values.

See Also

`bioma.ExpressionSet` | `bioma.data.MetaData` | `variableValues`

Topics

"Managing Gene Expression Data in Objects"

featureview

Draw linear or circular map of features from GenBank structure

Syntax

```
featureview(GBStructure)
featureview(GBStructure, FeatList)
featureview(GBStructure, FeatList, Levels)
featureview(GBStructure, Levels)
[Handles, OutFeatList] = featureview(...)

featureview(..., 'FontSize', FontSizeValue, ...)
featureview(..., 'ColorMap', ColorMapValue, ...)
featureview(..., 'Qualifiers', QualifiersValue, ...)
featureview(..., 'ShowPositions', ShowPositionsValue, ...)
```

Arguments

GBStructure	GenBank structure, typically created using the `getgenbank` or the `genbankread` function.
FeatList	String vector or cell array of character vectors specifying feature names (from the list of all features in the GenBank structure) to include in or exclude from the map. • If *FeatList* is a string vector or cell array of character vectors containing feature names, these features are mapped. Any features in *FeatList* not found in the GenBank structure are ignored. • If *FeatList* includes `'-'` as the first element in the cell array or string vector, then the remaining strings (features) are not mapped. By default, *FeatList* is the a list of all features in the GenBank structure.
Levels	Vector of `N` integers, where `N` is the number of features. Each integer represents the level in the map for the corresponding feature. For example, if *Levels* = `[1, 1, 2, 3, 3]`, the first two features would appear on level 1, the third feature on level 2, and the fourth and fifth features on level 3. By default, *Levels* = `[1:N]`.
FontSizeValue	Scalar that sets the font size (points) for the annotations of the features. Default is `9`.

ColorMapValue	Three-column matrix, to specify a list of colors to use for each feature. This matrix replaces the default matrix, which specifies the following colors and order: blue, green, red, cyan, magenta, yellow, brown, light green, orange, purple, gold, and silver. In the matrix, each row corresponds to a color, and each column specifies red, green, and blue intensity respectively. Valid values for the RGB intensities are `0.0` to `1.0`.
QualifiersValue	Cell array of character vectors or string vector to specify an ordered list of qualifiers to search for in the structure and use as annotations. For each feature, the first matching qualifier found from the list is used for its annotation. If a feature does not include any of the qualifiers, no annotation displays for that feature. By default, *QualifiersValue = {'gene', 'product', 'locus_tag', 'note', 'db_xref', 'protein_id'}*. Provide your own *QualifiersValue* to limit or expand the list of qualifiers or change the search order. **Tip** Set *QualifiersValue = {}* to create a map with no annotations. **Tip** To determine all qualifiers available for a given feature, do either of the following: • Create the map, and then click a feature or its annotation to list all qualifiers for that feature. • Use the `featureparse` command to parse all the features into a new structure, and then use the `fieldnames` command to list the qualifiers for a specific feature.
ShowPositionsValue	Property to add the sequence position to the annotation label for each feature. Enter `true` to add the sequence position. Default is `false`.

Description

`featureview(`*`GBStructure`*`)` creates a linear or circular map of all features from a GenBank structure, typically created using the `getgenbank` or the `genbankread` function.

`featureview(`*`GBStructure`*`, `*`FeatList`*`)` creates a linear or circular map of a subset of features from a GenBank structure. *FeatList* lets you specify features (from the list of all features in the GenBank structure) to include in or exclude from the map.

- If *FeatList* is a cell array of features, these features are mapped. Any features in *FeatList* not found in the GenBank structure are ignored.
- If *FeatList* includes `'-'` as the first string in the cell array, then the remaining strings (features) are not mapped.

By default, *FeatList* is a list of all features in the GenBank structure.

`featureview(`*`GBStructure`*`, `*`FeatList`*`, `*`Levels`*`)` or `featureview(`*`GBStructure`*`, `*`Levels`*`)` indicates which level on the map each feature is drawn. Level 1 is the left-most (linear map) or inner-most (circular map) level, and level `N` is the right-most (linear map) or outer-most (circular map) level, where `N` is the number of features.

`Levels` is a vector of `N` integers, where `N` is the number of features. Each integer represents the level in the map for the corresponding feature. For example, if *`Levels`* `= [1, 1, 2, 3, 3]`, the first two features would appear on level 1, the third feature on level 2, and the fourth and fifth features on level 3. By default, *`Levels`* `= [1:N]`.

`[`*`Handles, OutFeatList`*`] = featureview(...)` returns a list of handles for each feature in *`OutFeatList`*. It also returns *`OutFeatList`*, which is a cell array of the mapped features.

Tip Use *`Handles`* and *`OutFeatList`* with the `legend` command to create a legend of features.

`featureview(..., '`*`PropertyName`*`', `*`PropertyValue`*`, ...)` defines optional properties that use property name/value pairs in any order. These property name/value pairs are as follows:

`featureview(..., 'FontSize', `*`FontSizeValue`*`, ...)` sets the font size (points) for the annotations of the features. Default *`FontSizeValue`* is `9`.

`featureview(..., 'ColorMap', `*`ColorMapValue`*`, ...)` specifies a list of colors to use for each feature. This matrix replaces the default matrix, which specifies the following colors and order: blue, green, red, cyan, magenta, yellow, brown, light green, orange, purple, gold, and silver. *`ColorMapValue`* is a three-column matrix, where each row corresponds to a color, and each column specifies red, green, and blue intensity respectively. Valid values for the RGB intensities are `0.0` to `1.0`.

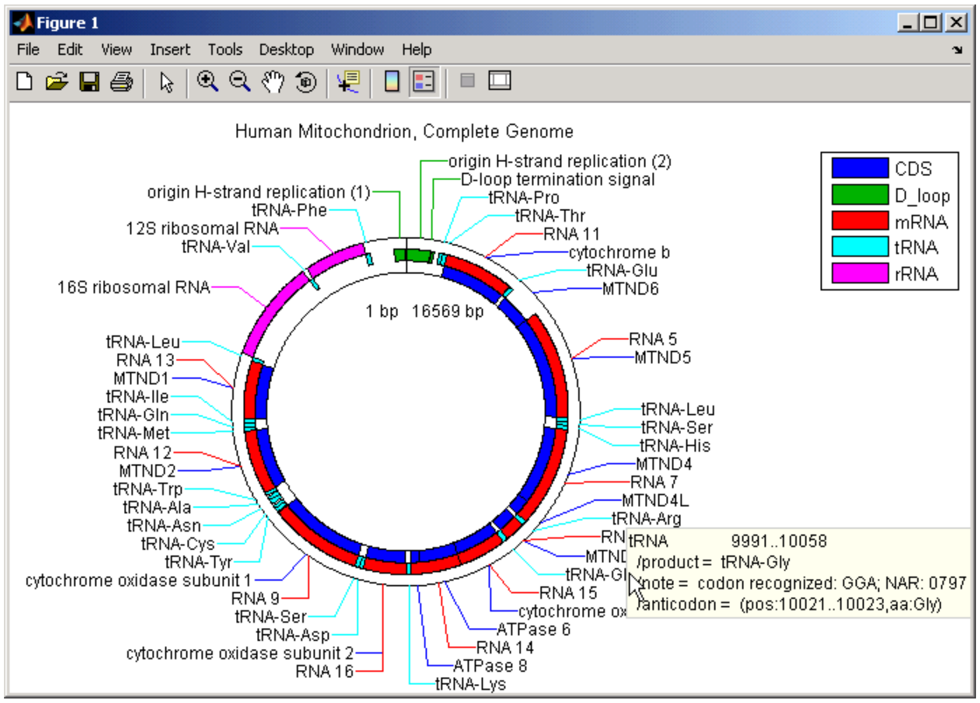
`featureview(..., 'Qualifiers', `*`QualifiersValue`*`, ...)` lets you specify an ordered list of qualifiers to search for and use as annotations. For each feature, the first matching qualifier found from the list is used for its annotation. If a feature does not include any of the qualifiers, no annotation displays for that feature. *`QualifiersValue`* is a cell array of character vectors or string vector. By default, *`QualifiersValue = {'gene', 'product', 'locus_tag', 'note', 'db_xref', 'protein_id'}`*. Provide your own *`QualifiersValue`* to limit or expand the list of qualifiers or change the search order.

Tip Set *`QualifiersValue = {}`* to create a map with no annotations.

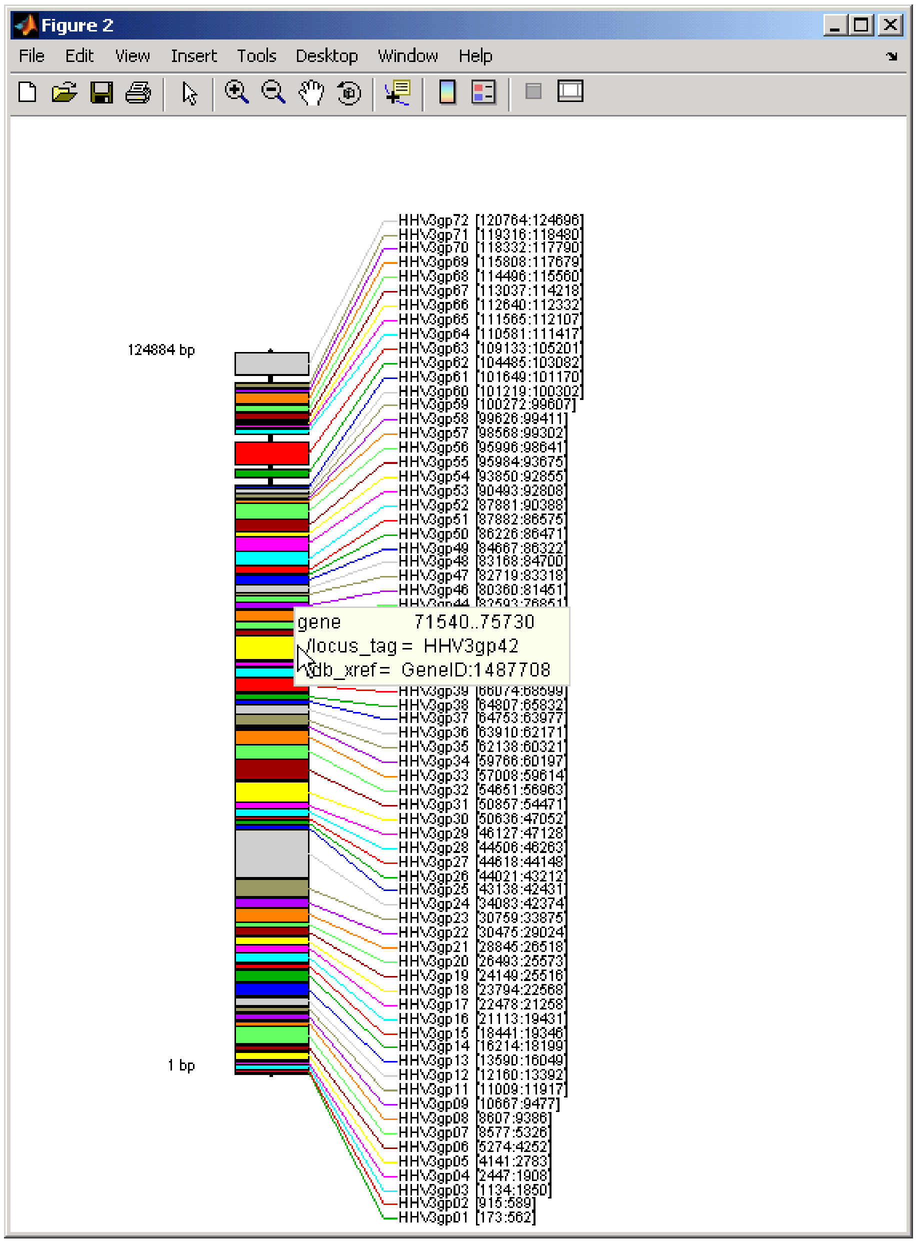
Tip To determine all qualifiers available for a given feature, do either of the following:

- Create the map, and then click a feature or its annotation to list all qualifiers for that feature.
- Use the `featureparse` command to parse all the features into a new structure, and then use the `fieldnames` command to list the qualifiers for a specific feature.

`featureview(..., 'ShowPositions', `*`ShowPositionsValue`*`, ...)` lets you add the sequence position to the annotation label. If *`ShowPositionsValue`* is `true`, sequence positions are added to the annotation labels. Default is `false`.



tRNA 9991..10058
/product = tRNA-Gly
/note = codon recognized: GGA; NAR: 0797
/anticodon = (pos:10021..10023,aa:Gly)



After creating a map:

- Click a feature or annotation to display a list of all qualifiers for that feature.
- Zoom the plot by clicking the following buttons:

  🔍+ or 🔍−

Examples

Example 1.9. Creating a Circular Map with a Legend

The following example creates a circular map of five different features mapped on three levels. It also uses outputs from the `featureview` function as inputs to the `legend` function to add a legend to the map.

```
GBStructure = getgenbank('J01415');
[Handles, OutFeatList] = featureview(GBStructure, ...
     {'CDS','D_loop','mRNA','tRNA','rRNA'}, [1 2 2 2 3])
legend(Handles, OutFeatList, 'interpreter', 'none', ...
     'location','bestoutside')
title('Human Mitochondrion, Complete Genome')
```

Example 1.10. Creating a Linear Map with Sequence Position Labels and Changed Font Size

The following example creates a linear map showing only the gene feature. It changes the font of the labels to seven points and includes the sequence position in the labels.

```
herpes = getgenbank('NC_001348');
featureview(herpes,{'gene'},'fontsize',7,'showpositions',true)
title('Genes in Human herpesvirus 3 (strain Dumas)')
```

Example 1.11. Determining Qualifiers for a Specific Feature

The following example uses the `getgenbank` function to create a GenBank structure, `GBStructure`. It then uses the `featureparse` function to parse the features in the GenBank structure into a new structure, `features`. It then uses the `fieldnames` function to return all qualifiers for one of the features, `D_loop`.

```
GenBankStructure = getgenbank('J01415');
features = featureparse (GenBankStructure)
features =

         source: [1x1 struct]
         D_loop: [1x2 struct]
     rep_origin: [1x3 struct]
    repeat_unit: [1x4 struct]
    misc_signal: [1x1 struct]
       misc_RNA: [1x1 struct]
      variation: [1x17 struct]
           tRNA: [1x22 struct]
           rRNA: [1x2 struct]
           mRNA: [1x10 struct]
            CDS: [1x13 struct]
       conflict: [1x1 struct]

fieldnames(features.D_loop)

ans =

    'Location'
    'Indices'
    'note'
    'citation'
```

Version History
Introduced in R2006b

See Also
`featureparse` | `genbankread` | `getgenbank` | `seqviewer`

galread

Read microarray data from GenePix array list file

Syntax

`GALData = galread(File)`

Arguments

`File`	GenePix® array list formatted file (GAL). Enter a character vector or string specifying a file name, or a path and file name.

Description

`galread` reads data from a GenePix formatted file into a MATLAB structure.

`GALData` `= galread(`*`File`*`)` reads in a GenePix array list formatted file (*`File`*) and creates a structure (`GALData`) containing the following fields.

Field
`Header`
`BlockData`
`IDs`
`Names`

The field `BlockData` is an `N`-by-`3` array. The columns of this array are the block data, the column data, and the row data respectively. For more information on the GAL format, see here.

Version History

Introduced before R2006a

See Also

`affyread` | `geoseriesread` | `geosoftread` | `gprread` | `ilmnbsread` | `imageneread` | `sptread`

gcrma

Perform GC Robust Multi-array Average (GCRMA) background adjustment, quantile normalization, and median-polish summarization on Affymetrix microarray probe-level data

Syntax

```
ExpressionMatrix = gcrma(PMMatrix, MMMatrix, ProbeIndices, AffinPM, AffinMM)
ExpressionMatrix = gcrma(PMMatrix, MMMatrix, ProbeIndices, SequenceMatrix)

ExpressionMatrix = gcrma(..., 'ChipIndex', ChipIndexValue, ...)
ExpressionMatrix = gcrma(..., 'OpticalCorr', OpticalCorrValue, ...)
ExpressionMatrix = gcrma(..., 'CorrConst', CorrConstValue, ...)
ExpressionMatrix = gcrma(..., 'Method', MethodValue, ...)
ExpressionMatrix = gcrma(..., 'TuningParam', TuningParamValue, ...)
ExpressionMatrix = gcrma(..., 'GSBCorr', GSBCorrValue, ...)
ExpressionMatrix = gcrma(..., 'Normalize', NormalizeValue, ...)
ExpressionMatrix = gcrma(..., 'Verbose', VerboseValue, ...)
```

Input Arguments

PMMatrix	Matrix of intensity values where each row corresponds to a perfect match (PM) probe and each column corresponds to an Affymetrix CEL file. (Each CEL file is generated from a separate chip. All chips should be of the same type.) **Tip** You can use the `PMIntensities` matrix returned by the `celintensityread` function.
MMMatrix	Matrix of intensity values where each row corresponds to a mismatch (MM) probe and each column corresponds to an Affymetrix CEL file. (Each CEL file is generated from a separate chip. All chips should be of the same type.) **Tip** You can use the `MMIntensities` matrix returned by the `celintensityread` function.
ProbeIndices	Column vector containing probe indices. Probes within a probe set are numbered 0 through *N* - 1, where *N* is the number of probes in the probe set. **Tip** You can use the `affyprobeseqread` function to generate this column vector.
AffinPM	Column vector of PM probe affinities. **Tip** You can use the `affyprobeaffinities` function to generate this column vector.

`AffinMM`	Column vector of MM probe affinities. **Tip** You can use the `affyprobeaffinities` function to generate this column vector.
`SequenceMatrix`	An *N*-by-25 matrix of sequence information for the perfect match (PM) probes on the Affymetrix GeneChip array, where *N* is the number of probes on the array. Each row corresponds to a probe, and each column corresponds to one of the 25 sequence positions. Nucleotides in the sequences are represented by one of the following integers: • `0` — None • `1` — A • `2` — C • `3` — G • `4` — T **Tip** You can use the `affyprobeseqread` function to generate this matrix. If you have this sequence information in letter representation, you can convert it to integer representation using the `nt2int` function.
`ChipIndexValue`	Positive integer specifying a column index in *`MMMatrix`*, which specifies a chip. This chip intensity data is used to compute probe affinities. Default is `1`.
`OpticalCorrValue`	Controls the use of optical background correction on the PM and MM intensity values in *`PMMatrix`* and *`MMMatrix`*. Choices are `true` (default) or `false`.
`CorrConstValue`	Value that specifies the correlation constant, rho, for background intensity for each PM/MM probe pair. Choices are any value `≥ 0` and `≤ 1`. Default is `0.7`.
`MethodValue`	Character vector or string that specifies the method to estimate the signal. Choices are `'MLE'`, a faster, ad hoc Maximum Likelihood Estimate method, or `'EB'`, a slower, more formal, empirical Bayes method. Default is `'MLE'`.
`TuningParamValue`	Value that specifies the tuning parameter used by the estimate method. This tuning parameter sets the lower bound of signal values with positive probability. Choices are a positive value. Default is `5` (MLE) or `0.5` (EB). **Tip** For information on determining a setting for this parameter, see Wu et al., 2004 on page 1-860.
`GSBCorrValue`	Specifies whether to perform gene-specific binding (GSB) correction using probe affinity data. Choices are `true` (default) or `false`. If there is no probe affinity information, this property is ignored.
`NormalizeValue`	Controls whether quantile normalization is performed on background adjusted data. Choices are `true` (default) or `false`.

VerboseValue	Controls the display of a progress report showing the number of each chip as it is completed. Choices are `true` (default) or `false`.

Output Arguments

ExpressionMatrix	Matrix of $\log_2$ expression values where each row corresponds to a gene (probe set) and each column corresponds to an Affymetrix CEL file, which represents a single chip.

Description

`ExpressionMatrix = gcrma(PMMatrix, MMMatrix, ProbeIndices, AffinPM, AffinMM)` performs GCRMA background adjustment, quantile normalization, and median-polish summarization on Affymetrix microarray probe-level data using probe affinity data. *ExpressionMatrix* is a matrix of $\log_2$ expression values where each row corresponds to a gene (probe set) and each column corresponds to an Affymetrix CEL file, which represents a single chip.

Note There is no column in *ExpressionMatrix* that contains probe set or gene information.

`ExpressionMatrix = gcrma(PMMatrix, MMMatrix, ProbeIndices, SequenceMatrix)` performs GCRMA background adjustment, quantile normalization, and Robust Multi-array Average (RMA) summarization on Affymetrix microarray probe-level data using probe sequence data to compute probe affinity data. *ExpressionMatrix* is a matrix of $\log_2$ expression values where each row corresponds to a gene (probe set) and each column corresponds to an Affymetrix CEL file, which represents a single chip.

Note If *AffinPM* and *AffinMM* affinity data and *SequenceMatrix* sequence data are not available, you can still use the `gcrma` function by entering an empty matrix for these inputs in the syntax.

`ExpressionMatrix = gcrma( ...'PropertyName', PropertyValue, ...)` calls `gcrma` with optional properties that use property name/property value pairs. You can specify one or more properties in any order. Each *PropertyName* must be enclosed in single quotes and is case insensitive. These property name/property value pairs are as follows:

`ExpressionMatrix = gcrma(..., 'ChipIndex', ChipIndexValue, ...)` computes probe affinities from MM probe intensity data from the chip with the specified column index in *MMMatrix*. Default *ChipIndexValue* is `1`. If *AffinPM* and *AffinMM* affinity data are provided, this property is ignored.

`ExpressionMatrix = gcrma(..., 'OpticalCorr', OpticalCorrValue, ...)` controls the use of optical background correction on the PM and MM intensity values in *PMMatrix* and *MMMatrix*. Choices are `true` (default) or `false`.

`ExpressionMatrix = gcrma(..., 'CorrConst', CorrConstValue, ...)` specifies the correlation constant, rho, for background intensity for each PM/MM probe pair. Choices are any value `≥ 0` and `≤ 1`. Default is `0.7`.

`ExpressionMatrix = gcrma(..., 'Method', MethodValue, ...)` specifies the method to estimate the signal. Choices are `MLE`, a faster, ad hoc Maximum Likelihood Estimate method, or `EB`, a slower, more formal, empirical Bayes method. Default is `MLE`.

`ExpressionMatrix = gcrma(..., 'TuningParam', TuningParamValue, ...)` specifies the tuning parameter used by the estimate method. This tuning parameter sets the lower bound of signal values with positive probability. Choices are a positive value. Default is `5` (MLE) or `0.5` (EB).

Tip For information on determining a setting for this parameter, see Wu et al., 2004 on page 1-860.

`ExpressionMatrix = gcrma(..., 'GSBCorr', GSBCorrValue, ...)` specifies whether to perform gene specific binding (GSB) correction using probe affinity data. Choices are `true` (default) or `false`. If there is no probe affinity information, this property is ignored.

`ExpressionMatrix = gcrma(..., 'Normalize', NormalizeValue, ...)` controls whether quantile normalization is performed on background adjusted data. Choices are `true` (default) or `false`.

`ExpressionMatrix = gcrma(..., 'Verbose', VerboseValue, ...)` controls the display of a progress report showing the number of each chip as it is completed. Choices are `true` (default) or `false`.

Examples

1. Load the MAT-file, included with the Bioinformatics Toolbox software, that contains Affymetrix data from a prostate cancer study. The variables in the MAT-file include `seqMatrix`, a matrix containing sequence information for PM probes, `pmMatrix` and `mmMatrix`, matrices containing PM and MM probe intensity values, and `probeIndices`, a column vector containing probe indexing information.

   ```
   load prostatecancerrawdata
   ```
2. Compute the Affymetrix PM and MM probe affinities from their sequences and MM probe intensities.

   ```
   [apm, amm] = affyprobeaffinities(seqMatrix, mmMatrix(:,1),...
                'ProbeIndices', probeIndices);
   ```
3. Perform GCRMA background adjustment, quantile normalization, and Robust Multi-array Average (RMA) summarization on the Affymetrix microarray probe-level data and create a matrix of expression values.

   ```
   expdata = gcrma(pmMatrix, mmMatrix, probeIndices, seqMatrix);
   ```

The `prostatecancerrawdata.mat` file used in this example contains data from Best et al., 2005 on page 1-855.

Version History

Introduced in R2007a

References

[1] Wu, Z., Irizarry, R.A., Gentleman, R., Murillo, F.M., and Spencer, F. (2004). A Model Based Background Adjustment for Oligonucleotide Expression Arrays. Journal of the American Statistical Association *99(468)*, 909–917.

[2] Wu, Z., and Irizarry, R.A. (2005). Stochastic Models Inspired by Hybridization Theory for Short Oligonucleotide Arrays. Proceedings of RECOMB 2004. J Comput Biol. *12(6)*, 882–93.

[3] Wu, Z., and Irizarry, R.A. (2005). A Statistical Framework for the Analysis of Microarray Probe-Level Data. Johns Hopkins University, Biostatistics Working Papers 73.

[4] Speed, T. (2006). Background models and GCRMA. Lecture 10, Statistics 246, University of California Berkeley.

[5] Best, C.J.M., Gillespie, J.W., Yi, Y., Chandramouli, G.V.R., Perlmutter, M.A., Gathright, Y., Erickson, H.S., Georgevich, L., Tangrea, M.A., Duray, P.H., Gonzalez, S., Velasco, A., Linehan, W.M., Matusik, R.J., Price, D.K., Figg, W.D., Emmert-Buck, M.R., and Chuaqui, R.F. (2005). Molecular alterations in primary prostate cancer after androgen ablation therapy. Clinical Cancer Research *11*, 6823–6834.

See Also

`affygcrma` | `affyprobeseqread` | `affyread` | `affyrma` | `celintensityread` | `gcrmabackadj` | `quantilenorm` | `rmabackadj` | `rmasummary`

gcrmabackadj

Perform GC Robust Multi-array Average (GCRMA) background adjustment on Affymetrix microarray probe-level data using sequence information

Syntax

```
PMMatrix_Adj = gcrmabackadj(PMMatrix, MMMatrix, AffinPM, AffinMM)
[PMMatrix_Adj, nsbStruct] = gcrmabackadj(PMMatrix, MMMatrix, AffinPM,
AffinMM)

... = gcrmabackadj( ...'OpticalCorr', OpticalCorrValue, ...)
... = gcrmabackadj( ...'CorrConst', CorrConstValue, ...)
... = gcrmabackadj( ...'Method', MethodValue, ...)
... = gcrmabackadj( ...'TuningParam', TuningParamValue, ...)
... = gcrmabackadj( ...'AddVariance', AddVarianceValue, ...)
... = gcrmabackadj( ...'GSBCorr', GSBCorrValue, ...)
... = gcrmabackadj( ...'Showplot', ShowplotValue, ...)
... = gcrmabackadj( ...'Verbose', VerboseValue, ...)
```

Input Arguments

PMMatrix	Matrix of intensity values where each row corresponds to a perfect match (PM) probe and each column corresponds to an Affymetrix CEL file. (Each CEL file is generated from a separate chip. All chips should be of the same type.) **Tip** You can use the `PMIntensities` matrix returned by the `celintensityread` function.
MMMatrix	Matrix of intensity values where each row corresponds to a mismatch (MM) probe and each column corresponds to an Affymetrix CEL file. (Each CEL file is generated from a separate chip. All chips should be of the same type.) **Tip** You can use the `MMIntensities` matrix returned by the `celintensityread` function.
AffinPM	Column vector of PM probe affinities, such as returned by the `affyprobeaffinities` function. Each row corresponds to a probe.
AffinMM	Column vector of MM probe affinities, such as returned by the `affyprobeaffinities` function. Each row corresponds to a probe.
OpticalCorrValue	Controls the use of optical background correction on the PM and MM probe intensity values in *PMMatrix* and *MMMatrix*. Choices are `true` (default) or `false`.

CorrConstValue	Value that specifies the correlation constant, rho, for log background intensity for each PM/MM probe pair. Choices are any value `≥ 0` and `≤ 1`. Default is `0.7`.
MethodValue	Character vector or string that specifies the method to estimate the signal. Choices are `'MLE'`, a faster, ad hoc Maximum Likelihood Estimate method, or `'EB'`, a slower, more formal, empirical Bayes method. Default is `'MLE'`.
TuningParamValue	Value that specifies the tuning parameter used by the estimate method. This tuning parameter sets the lower bound of signal values with positive probability. Choices are a positive value. Default is `5` (MLE) or `0.5` (EB). **Tip** For information on determining a setting for this parameter, see Wu et al., 2004 on page 1-860.
AddVarianceValue	Controls whether the signal variance is added to the weight function for smoothing low signal edge. Choices are `true` or `false` (default).
GSBCorrValue	Specifies whether to perform gene-specific binding (GSB) correction using probe affinity data. Choices are `true` (default) or `false`. If there is no probe affinity information, this property is ignored.
ShowplotValue	Controls the display of a plot showing the $\log_2$ of probe intensity values from a specified column (chip) in *MMMatrix*, versus probe affinities in *AffinMM*. Choices are `true`, `false`, or *I*, an integer specifying a column in *MMMatrix*. If set to `true`, the first column in *MMMatrix* is plotted. Default is: • `false` — When return values are specified. • `true` — When return values are not specified.
VerboseValue	Controls the display of a progress report showing the number of each chip as it is completed. Choices are `true` (default) or `false`.

Output Arguments

PMMatrix_Adj	Matrix of background adjusted PM (perfect match) intensity values.
nsbStruct	Structure containing nonspecific binding background parameters, estimated from the intensities and affinities of probes on an Affymetrix GeneChip array. *nsbStruct* includes the following fields: • `sigma` • `mu_pm` • `mu_mm`

Description

PMMatrix_Adj = `gcrmabackadj(`*PMMatrix*, *MMMatrix*, *AffinPM*, *AffinMM*`)` performs GCRMA background adjustment (including optical background correction and nonspecific binding correction) on Affymetrix microarray probe-level data, using probe sequence information and returns *PMMatrix_Adj*, a matrix of background adjusted PM (perfect match) intensity values.

Note If *`AffinPM`* and *`AffinMM`* data are not available, you can still use the `gcrmabackadj` function by entering empty column vectors for both of these inputs in the syntax.

`[`*`PMMatrix_Adj`*`,` *`nsbStruct`*`] = gcrmabackadj(`*`PMMatrix`*`,` *`MMMatrix`*`,` *`AffinPM`*`,` *`AffinMM`*`)` returns *`nsbStruct`*, a structure containing nonspecific binding background parameters, estimated from the intensities and affinities of probes on an Affymetrix GeneChip array. *`nsbStruct`* includes the following fields:

- `sigma`
- `mu_pm`
- `mu_mm`

`... = gcrmabackadj( ...'`*`PropertyName`*`',` *`PropertyValue`*`, ...)` calls `gcrmabackadj` with optional properties that use property name/property value pairs. You can specify one or more properties in any order. Each *`PropertyName`* must be enclosed in single quotation marks and is case insensitive. These property name/property value pairs are as follows:

`... = gcrmabackadj( ...'OpticalCorr',` *`OpticalCorrValue`*`, ...)` controls the use of optical background correction on the PM and MM probe intensity values in *`PMMatrix`* and *`MMMatrix`*. Choices are `true` (default) or `false`.

`... = gcrmabackadj( ...'CorrConst',` *`CorrConstValue`*`, ...)` specifies the correlation constant, rho, for log background intensity for each PM/MM probe pair. Choices are any value `≥ 0` and `≤ 1`. Default is `0.7`.

`... = gcrmabackadj( ...'Method',` *`MethodValue`*`, ...)` specifies the method to estimate the signal. Choices are `MLE`, a faster, ad hoc Maximum Likelihood Estimate method, or `EB`, a slower, more formal, empirical Bayes method. Default is `MLE`.

`... = gcrmabackadj( ...'TuningParam',` *`TuningParamValue`*`, ...)` specifies the tuning parameter used by the estimate method. This tuning parameter sets the lower bound of signal values with positive probability. Choices are a positive value. Default is `5` (MLE) or `0.5` (EB).

Tip For information on determining a setting for this parameter, see Wu et al., 2004 on page 1-860.

`... = gcrmabackadj( ...'AddVariance',` *`AddVarianceValue`*`, ...)` controls whether the signal variance is added to the weight function for smoothing low signal edge. Choices are `true` or `false` (default).

`... = gcrmabackadj( ...'GSBCorr',` *`GSBCorrValue`*`, ...)` specifies whether to perform gene specific binding (GSB) correction using probe affinity data. Choices are `true` (default) or `false`. If there is no probe affinity information, this property is ignored.

`... = gcrmabackadj( ...'Showplot',` *`ShowplotValue`*`, ...)` controls the display of a plot showing the $\log_2$ of probe intensity values from a specified column (chip) in *`MMMatrix`*, versus probe affinities in *`AffinMM`*. Choices are `true`, `false`, or *`I`*, an integer specifying a column in *`MMMatrix`*. If set to `true`, the first column in *`MMMatrix`* is plotted. Default is:

- `false` — When return values are specified.

- `true` — When return values are not specified.

`... = gcrmabackadj( ...'Verbose', VerboseValue, ...)` controls the display of a progress report showing the number of each chip as it is completed. Choices are `true` (default) or `false`.

Examples

1 Load the MAT-file, included with the Bioinformatics Toolbox software, that contains Affymetrix data from a prostate cancer study. The variables in the MAT-file include `seqMatrix`, a matrix containing sequence information for PM probes, `pmMatrix` and `mmMatrix`, matrices containing PM and MM probe intensity values, and `probeIndices`, a column vector containing probe indexing information.

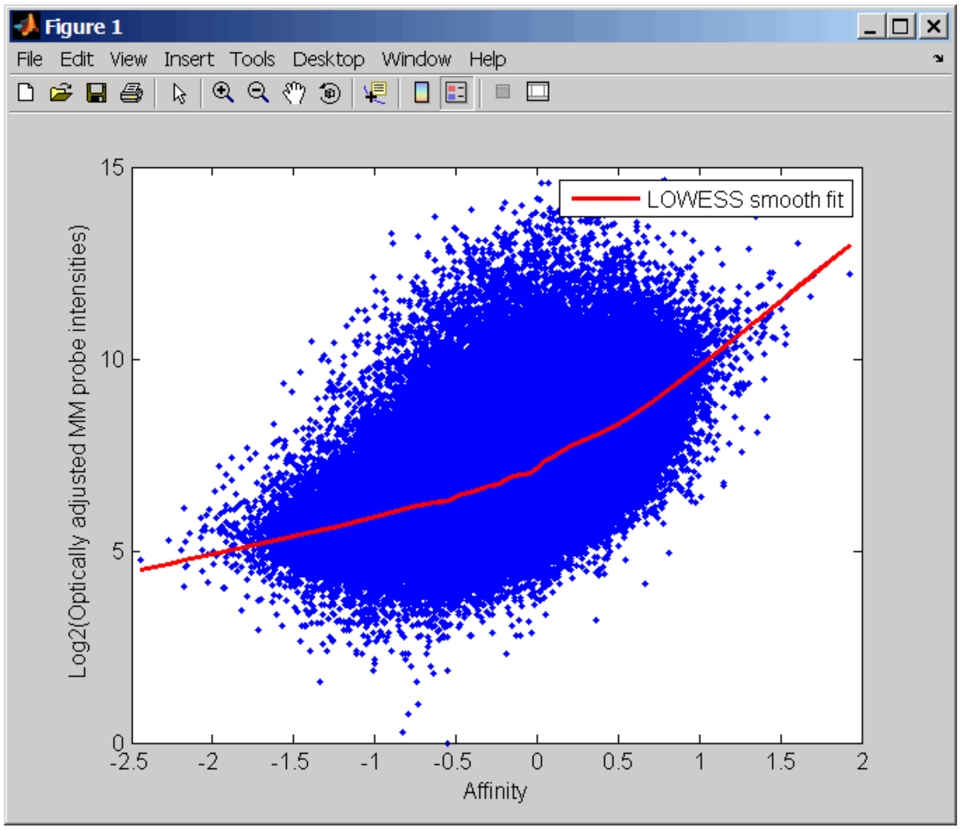
```
load prostatecancerrawdata
```

2 Compute the Affymetrix PM and MM probe affinities from their sequences and MM probe intensities.

```
[apm, amm] = affyprobeaffinities(seqMatrix, mmMatrix(:,1),...
             'ProbeIndices', probeIndices);
```

3 Perform GCRMA background adjustment on the Affymetrix microarray probe-level data, creating a matrix of background adjusted PM intensity values. Also, display a plot showing the $\log_2$ of probe intensity values from column 3 (chip 3) in `mmMatrix`, versus probe affinities in `amm`.

```
pms_adj = gcrmabackadj(pmMatrix, mmMatrix, apm, amm, 'showplot', 3);
```



4 Perform GCRMA background adjustment again, using the slower, more formal, empirical Bayes method.

```
pms_adj2 = gcrmabackadj(pmMatrix, mmMatrix, apm, amm, 'method', 'EB');
```

The `prostatecancerrawdata.mat` file used in this example contains data from Best et al., 2005.

Version History

Introduced in R2007a

References

[1] Wu, Z., Irizarry, R.A., Gentleman, R., Murillo, F.M., and Spencer, F. (2004). A Model Based Background Adjustment for Oligonucleotide Expression Arrays. Journal of the American Statistical Association *99(468)*, 909–917.

[2] Wu, Z., and Irizarry, R.A. (2005). Stochastic Models Inspired by Hybridization Theory for Short Oligonucleotide Arrays. Proceedings of RECOMB 2004. J Comput Biol. *12(6)*, 882–93.

[3] Wu, Z., and Irizarry, R.A. (2005). A Statistical Framework for the Analysis of Microarray Probe-Level Data. Johns Hopkins University, Biostatistics Working Papers 73.

[4] Wu, Z., and Irizarry, R.A. (2003). A Model Based Background Adjustment for Oligonucleotide Expression Arrays. RSS Workshop on Gene Expression, Wye, England, `https://biosun01.biostat.jhsph.edu/%7Eririzarr/Talks/gctalk.pdf`.

[5] Abd Rabbo, N.A., and Barakat, H.M. (1979). Estimation Problems in Bivariate Lognormal Distribution. Indian J. Pure Appl. Math *10(7)*, 815–825.

[6] Best, C.J.M., Gillespie, J.W., Yi, Y., Chandramouli, G.V.R., Perlmutter, M.A., Gathright, Y., Erickson, H.S., Georgevich, L., Tangrea, M.A., Duray, P.H., Gonzalez, S., Velasco, A., Linehan, W.M., Matusik, R.J., Price, D.K., Figg, W.D., Emmert-Buck, M.R., and Chuaqui, R.F. (2005). Molecular alterations in primary prostate cancer after androgen ablation therapy. Clinical Cancer Research *11*, 6823–6834.

See Also

`affygcrma` | `affyprobeseqread` | `affyread` | `celintensityread` | `probelibraryinfo`

ge (DataMatrix)

Test DataMatrix objects for greater than or equal to

Syntax

```
T = ge(DMObj1, DMObj2)
T = DMObj1 >= DMObj2
T = ge(DMObj1, B)
T = DMObj1 >= B
T = ge(B, DMObj1)
T = B >= DMObj1
```

Input Arguments

DMObj1, *DMObj2*	DataMatrix objects, such as created by `DataMatrix` (object constructor).
B	MATLAB numeric or logical array.

Output Arguments

T	Logical matrix of the same size as *DMObj1* and *DMObj2* or *DMObj1* and *B*. It contains logical `1` (true) where elements in the first input are greater than or equal to the corresponding element in the second input, and logical `0` (false) otherwise.

Description

`T = ge(DMObj1, DMObj2)` or the equivalent `T = DMObj1 >= DMObj2` compares each element in DataMatrix object *DMObj1* to the corresponding element in DataMatrix object *DMObj2*, and returns *T*, a logical matrix of the same size as *DMObj1* and *DMObj2*, containing logical `1` (true) where elements in *DMObj1* are greater than or equal to the corresponding element in *DMObj2*, and logical `0` (false) otherwise. *DMObj1* and *DMObj2* must have the same size (number of rows and columns), unless one is a scalar (1-by-1 DataMatrix object). *DMObj1* and *DMObj2* can have different `Name` properties.

`T = ge(DMObj1, B)` or the equivalent `T = DMObj1 >= B` compares each element in DataMatrix object *DMObj1* to the corresponding element in *B*, a numeric or logical array, and returns *T*, a logical matrix of the same size as *DMObj1* and *B*, containing logical `1` (true) where elements in *DMObj1* are greater than or equal to the corresponding element in *B*, and logical `0` (false) otherwise. *DMObj1* and *B* must have the same size (number of rows and columns), unless one is a scalar.

`T = ge(B, DMObj1)` or the equivalent `T = B >= DMObj1` compares each element in *B*, a numeric or logical array, to the corresponding element in DataMatrix object *DMObj1*, and returns *T*, a logical matrix of the same size as *B* and *DMObj1*, containing logical `1` (true) where elements in *B* are greater than or equal to the corresponding element in *DMObj1*, and logical `0` (false) otherwise. *B* and *DMObj1* must have the same size (number of rows and columns), unless one is a scalar.

MATLAB calls `T = ge(X, Y)` for the syntax `T = X >= Y` when *X* or *Y* is a DataMatrix object.

Version History

Introduced in R2008b

See Also

`DataMatrix` | `le`

Topics

DataMatrix object on page 1-734

genbankread

Read data from GenBank file

Syntax

```
GenBankData = genbankread(File)
GenBankData = genbankread(File, 'TimeOut', TimeOutValue)
```

Arguments

`File`	Either of the following: • Character vector or string specifying a file name, a path and file name, or a URL pointing to a file. The referenced file is a GenBank-formatted file (ASCII text file). If you specify only a file name, that file must be on the MATLAB search path or in the MATLAB Current Folder. • MATLAB character array or string vector that contains the text of a GenBank-formatted file. **Tip** You can use the `getgenbank` function with the `'ToFile'` property to retrieve sequence information from the GenBank database and create an GenBank-formatted file.
`TimeOutValue`	Connection timeout in seconds, specified as a positive scalar. The default value is 5. For details, see here.
`GenBankData`	MATLAB structure or array of structures containing fields corresponding to GenBank keywords.

Description

`GenBankData` `= genbankread(`*`File`*`)` reads a GenBank-formatted file, *`File`*, and creates *`GenBankData`*, a structure or array of structures, containing fields corresponding to the GenBank keywords. When *`File`* contains multiple entries, each entry is stored as a separate element in *`GenBankData`*. For a list of the GenBank keywords, see https://www.ncbi.nlm.nih.gov/Sitemap/samplerecord.html.

`GenBankData` `= genbankread(`*`File`*`, 'TimeOut',` *`TimeOutValue`*`)` sets the connection timeout (in seconds) to read data from a remote file or URL.

Examples

1 Retrieve sequence information for the HEXA gene, store the data in a file, and then read into the MATLAB software.

```
getgenbank('nm_000520', 'ToFile', 'TaySachs_Gene.txt')
s = genbankread('TaySachs_Gene.txt')

s =
                LocusName: 'NM_000520'
      LocusSequenceLength: '2437'
     LocusNumberofStrands: ''
            LocusTopology: 'linear'
        LocusMoleculeType: 'mRNA'
     LocusGenBankDivision: 'PRI'
    LocusModificationDate: '18-FEB-2009'
               Definition: [1x63 char]
                Accession: 'NM_000520'
                  Version: 'NM_000520.4'
                       GI: '189181665'
                  Project: []
                   DBLink: []
                 Keywords: []
                  Segment: []
                   Source: 'Homo sapiens (human)'
           SourceOrganism: [4x65 char]
                Reference: {1x10 cell}
                  Comment: [32x67 char]
                 Features: [147x74 char]
                      CDS: [1x1 struct]
                 Sequence: [1x2437 char]
```

2 Display the source organism for this sequence.

```
s.SourceOrganism

ans =

Homo sapiens
Eukaryota; Metazoa; Chordata; Craniata; Vertebrata; Euteleostomi;
Mammalia; Eutheria; Euarchontoglires; Primates; Haplorrhini;
Catarrhini; Hominidae; Homo.
```

Version History

Introduced before R2006a

See Also

emblread | fastaread | genpeptread | getgenbank | scfread | seqviewer

geneentropyfilter

Remove genes with low entropy expression values

Syntax

```
Mask = geneentropyfilter(Data)
[Mask, FData] = geneentropyfilter(Data)
[Mask, FData, FNames] = geneentropyfilter(Data, Names)

geneentropyfilter(..., 'Percentile', PercentileValue)
```

Arguments

Data	DataMatrix object on page 1-734 or numeric matrix where each row corresponds to the experimental results for one gene. Each column is the results for all genes from one experiment.
Names	Cell array of character vectors or string vector where each element corresponds to the name of a gene for each row of experimental data. *Names* has same number of rows as *Data* with each row containing the name or ID of the gene in the data set.
PercentileValue	Property to specify a percentile below which gene data is removed. Enter a value from `0` to `100`.

Description

`Mask = geneentropyfilter(Data)` identifies gene expression profiles in *Data* with entropy values less than the 10th percentile.

Mask is a logical vector with one element for each row in *Data*. The elements of *Mask* corresponding to rows with a variance greater than the threshold have a value of `1`, and those with a variance less than the threshold are `0`.

`[Mask, FData] = geneentropyfilter(Data)` returns *FData*, a filtered data matrix. You can also create *FData* using `FData = Data(Mask,:)`.

`[Mask, FData, FNames] = geneentropyfilter(Data, Names)` returns *FNames*, a filtered names array, where *Names* is a cell array of character vectors or string vector of the names of the genes corresponding to each row of *Data*. You can also create *FNames* using `FNames = Names(Mask)`.

Note If *Data* is a DataMatrix object with specified row names, you do not need to provide the second input *Names* to return the third output *FNames*.

`geneentropyfilter(..., 'Percentile', PercentileValue)` removes from *Data*, the experimental data, gene expression profiles with entropy values less than *PercentileValue*, the specified percentile.

Examples

1. Load the MAT-file, provided with the Bioinformatics Toolbox software, that contains yeast data. This MAT-file includes three variables: `yeastvalues`, a matrix of gene expression data, `genes`, a cell array of GenBank accession numbers for labeling the rows in `yeastvalues`, and `times`, a vector of time values for labeling the columns in `yeastvalues`

   ```
   load yeastdata
   ```
2. Remove genes with low entropy expression values.

   ```
   [fyeastvalues, fgenes] = geneentropyfilter(yeastvalues,genes);
   ```

Version History
Introduced before R2006a

References

[1] Kohane I.S., Kho A.T., Butte A.J. (2003), Microarrays for an Integrative Genomics, Cambridge, MA:MIT Press.

See Also
`exprprofrange` | `exprprofvar` | `genelowvalfilter` | `generangefilter` | `genevarfilter`

genelowvalfilter

Remove gene profiles with low absolute values

Syntax

```
Mask = genelowvalfilter(Data)
[Mask,FData] = genelowvalfilter(Data)
[Mask,FData,FNames] = genelowvalfilter(Data,geneNames)

[ ___ ] = genelowvalfilter( ___ ,Name,Value)
```

Description

`Mask = genelowvalfilter(Data)` returns a logical vector `Mask` identifying gene expression profiles in `Data` that have absolute expression levels in the lowest 10% of the data set.

Gene expression profile experiments have data where the absolute values are very low. The quality of this type of data is often bad due to large quantization errors or simply poor spot hybridization. Use this function to filter data.

`[Mask,FData] = genelowvalfilter(Data)` also returns `FData`, a data matrix containing filtered expression profiles.

`[Mask,FData,FNames] = genelowvalfilter(Data,geneNames)` also returns `FNames`, a cell array of filtered gene names or IDs. You have to specify `geneNames` to return `FNames` unless `Data` is a `DataMatrix` object with specified row names.

`[ ___ ] = genelowvalfilter( ___ ,Name,Value)` uses additional options specified as one or more optional name-value pair arguments.

Examples

Filter Out Genes with Low Absolute Expression Levels

Load the sample yeast data.

```
load yeastdata;
```

Retrieve the genes and corresponding expression data where absolute expression levels exceed the 10th percentile.

```
[mask,filteredData,filteredGenes] = genelowvalfilter(yeastvalues,genes);
```

Compare the number of filtered genes (`filteredGenes`) with the number of genes in the original data set (`genes`).

```
size (filteredGenes,1)
```

```
ans = 6394
```

Filter Out Genes with Low Absolute Expression Levels Using a Logical Vector

Load the sample yeast data.

```
load yeastdata;
```

Mark the genes that have low absolute expression levels below the 10th percentile of the data set.

```
mask = genelowvalfilter(yeastvalues);
```

The variable `genes` contains every gene names in the yeast data set. Use the generated logical vector `mask` to retrieve the genes where expression levels exceed the 10th percentile.

```
filteredGenes = genes(mask);
```

Extract corresponding expression profile data for the selected genes from the variable `yeastvalues`, which contains expression profiles of every gene in the yeast data set.

```
filteredData = yeastvalues(mask,:);
```

Filter Out Genes with Absolute Expression Levels that are Lower Than a User-Defined Threshold

Load the sample yeast data.

```
load yeastdata;
```

Retrieve the genes and corresponding expression data where absolute expression levels exceed the 30th percentile of the data set.

```
[mask,filteredData,filteredGenes] = genelowvalfilter(yeastvalues,genes,'Percentile',30);
```

Compare the number of filtered genes (`filteredGenes`) with the number of genes in the original data set (`genes`).

```
size (filteredGenes,1)
```

```
ans = 6384
```

Input Arguments

Data — Input data
DataMatrix object | numeric matrix

Input data, specified as a `DataMatrix` object or numeric matrix. Each row of the matrix corresponds to the experimental results for one gene. Each column represents the results for all genes from one experiment.

geneNames — Gene names or IDs
cell array of character vectors | string vector

Gene names or IDs, specified as a cell array of character vectors or string vector. The array has the same number of rows as `Data`. Each row contains the name or ID of the gene in the data set.

Note If `Data` is a `DataMatrix` object with specified row names, you do not need to provide the second input `geneNames` to return the third output `FNames`.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `'AbsValue',10.5` specifies `genelowvalfilter` to remove expression profiles with absolute values less than 10.5.

`Percentile` — Percentile value
10 (default) | scalar value in the range (0,100)

Percentile value, specified as a scalar value in the range (0 to 100). The function `genelowvalfilter` removes gene expression profiles with absolute values less than the percentile value, which is specified using `'Percentile'`.

Example: `'Percentile',50`

`AbsValue` — Absolute expression profile value
real number

Absolute expression profile value, specified as a real number. The function `genelowvalfilter` removes gene expression profiles with absolute values less than the absolute value, which is specified using `'AbsValue'`.

Example: `'AbsValue',10.5`

`AnyVal` — Logical indicator to select minimum or maximum absolute value
`false` (default) | `true`

Logical indicator to select the minimum or maximum absolute value, specified as `true` or `false`. Set the value to `true` to select the minimum absolute value. Set it to `false` to select the maximum absolute value.

Example: `'AnyVal',true`

Output Arguments

`Mask` — Logical vector
vector of 0s and 1s

Logical vector, returned as a vector of 0s and 1s for each row in `Data`. The elements of `Mask` with value `1` correspond to rows with absolute expression levels exceeding the threshold, and those with value `0` correspond to rows with absolute expression levels less than or equal to the threshold.

`FData` — Filtered data matrix
data matrix

Filtered data matrix, returned as a data matrix that contains gene expression profiles with absolute expression levels exceeding the threshold value. You can also create `FData` using `FData = Data(Mask,:)`.

FNames — Array of filtered gene names
cell array of character vectors | string vector

Array of filtered gene names, returned as a cell array of character vectors or string vector. It contains gene names or IDs corresponding to each row of `Data` that contains gene expression profiles with absolute expression levels exceeding the threshold value. You can also create `FNames` using `FNames = geneNames(Mask)`.

Version History

Introduced before R2006a

References

[1] Kohane, I.S., Kho, A.T., Butte, A.J. (2003). Microarrays for an Integrative Genomics, First Edition (Cambridge, MA: MIT Press).

See Also

`exprprofrange` | `exprprofvar` | `geneentropyfilter` | `generangefilter` | `genevarfilter`

geneont

Data structure containing Gene Ontology (GO) information

Description

A `geneont` object is a data structure containing Gene Ontology information.

You can explore and traverse Gene Ontology terms using “is_a” and “part_of” relationships.

Creation

Syntax

```
GeneontObj = geneont
GeneontObj = geneont('File',filename)
GeneontObj = geneont('Live',true)
GeneontObj = geneont('Live',true,'ToFile',filepath)
```

Description

`GeneontObj = geneont` creates `GeneontObj`, a `geneont` object, from the `gene_ontology.obo` file in the MATLAB current directory. The function also creates multiple term objects, one for each term in the `GeneontObj` object.

You can use parenthesis () indexing with either GO identifiers (numbers) or by GO terms (term objects) to create a subontology.

`GeneontObj` is a handle object. To learn how this affects your use of the object, see Copying Objects in the MATLAB Programming Fundamentals documentation.

`GeneontObj = geneont('File',filename)` creates `GeneontObj` from the `filename` file. `filename` must be on the MATLAB search path.

`GeneontObj = geneont('Live',true)` creates `GeneontObj` from the current version of the Gene Ontology database.

Note The full Gene Ontology database may take several minutes to download when you run this function using the `'Live'` property.

`GeneontObj = geneont('Live',true,'ToFile',filepath)` creates `GeneontObj` from the current version of the Gene Ontology database and saves the contents of this file to `filepath`.

Input Arguments

`'File'` — Name of a gene ontology file in the current directory
string | character vector

Name of a gene ontology file in the current directory, specified as a string or character vector. The file must be an Open Biomedical Ontology (OBO)-formatted file that is on the MATLAB search path.

`'Live'` — Indication to use the current online Gene Ontology database
`false` (default) | `true`

Indication to use the current online Gene Ontology database, specified as `false` (do not use the online database) or `true` (use the online database).

`'ToFile'` — File to save the online database
string | character vector

File to save the online database, specified as a string or character vector representing the path to a (new) file. If the value does not include path information, the file is saved to the current directory.

Properties

`date` — Date and time OBO file was last updated
character vector

This property is read-only.

Date and time the OBO file was last updated, returned as a character vector. The OBO file is the Open Biomedical Ontology file from which the `geneont` object was created.

Example: `'02:12:2008 19:30'`

Data Types: `char`

`default_namespace` — Namespace to which GO terms are assigned
character vector

This property is read-only.

Namespace to which GO terms are assigned, returned as a character vector. Currently, `'gene_ontology'` is the only possible namespace. However, other namespaces may be used in the future. Use this namespace information to determine the ontology namespace to which the GO terms in a `geneont` object are assigned.

Example: `'gene_ontology'`

Data Types: `char`

`format_version` — Format version of OBO file
character vector

This property is read-only.

Format version of the OBO file, returned as a character vector. The OBO file is the Open Biomedical Ontology file from which the `geneont` object was created.

Example: `'1.2'`

Data Types: `char`

`terms` — `term` objects of `GeneontObj`
column vector of handles

This property is read-only.

`term` objects of `GeneontObj`, returned as a column vector of handles.

Note Although `terms` is a column vector with handles to `term` objects, in the MATLAB Command Window, `terms` displays as a structure array, with one structure for each GO term in the `geneont` object.

Use the information in this structure to access (by GO ID) the terms of a `geneont` object and to view the properties of individual terms. For an example, see "Use terms to Explore Gene Ontology" on page 1-874.

Example: `'02:12:2008 19:30'`

Data Types: `char`

Object Functions

getancestors	Find terms that are ancestors of specified Gene Ontology (GO) term
getdescendents	Find terms that are descendents of specified Gene Ontology (GO) term
getmatrix	Convert geneont object into relationship matrix
getrelatives	Find terms that are relatives of specified Gene Ontology (GO) term

Examples

Create Gene Ontology Object and Explore GO Terms

Download the current version of the Gene Ontology database from the Web into a geneont object.

```
GeneontObj = geneont('Live',true)
```

```
Gene Ontology object with 47266 Terms.
```

Display information about the `geneont` object.

```
get(GeneontObj)
```

```
                 default_namespace: 'gene_ontology'
                    format_version: '1.2'
                      data_version: 'releases/2022-01-13'
                           version: ''
                              date: ''
                          saved_by: ''
                 auto_generated_by: ''
                         subsetdef: {17×1 cell}
                            import: ''
                    synonymtypedef: 'systematic_synonym "Systematic synonym" EXACT'
                           idspace: ''
    default_relationship_id_prefix: ''
                        id_mapping: ''
                            remark: ''
                           typeref: ''
                  unrecognized_tag: {'ontology'  'go'}
                             Terms: [47266×1 geneont.term]
```

Search for all GO terms in the geneont object that contain the character vector `ribosome` in the field `name`, and use the `geneont.terms` property to create a MATLAB® structure array of `term` objects containing those terms.

```
comparison = regexpi(get(GeneontObj.terms,'name'),'ribosome');
indices = find(~cellfun('isempty',comparison));
terms_with_ribosmome = GeneontObj.terms(indices)
```

```
  33×1 struct array with fields:

    id
    name
    ontology
    definition
    comment
    synonym
    is_a
    part_of
    obsolete
```

Note: Although the `terms` property is a column vector with handles to `term` objects, in the MATLAB Command Window, `terms` displays as a structure array, with one structure for each GO term in the `geneont` object.

Use `terms` to Explore Gene Ontology

Download the current version of the Gene Ontology database from the Web into a `geneont` object.

```
GeneontObj = geneont('Live',true)
```

```
Gene Ontology object with 47266 Terms.
```

View the `terms` property of `GeneontObj`.

```
GeneontObj.terms
```

```
  47266×1 struct array with fields:

    id
    name
    ontology
    definition
    comment
    synonym
    is_a
    part_of
    obsolete
```

Note: Although the `terms` property is an array of handles to term objects, in the MATLAB® Command Window, `terms` displays as a structure array, with one structure for each GO term in the `geneont` object.

View the properties of the `term` object in the 14,723rd position in the `geneont` object.

```
GeneontObj.terms(14723)
```

```
            id: 31445
          name: 'regulation of heterochromatin assembly'
      ontology: 'biological process'
    definition: '"Any process that modulates the frequency, rate, extent or location of heterochr
       comment: ''
       synonym: {'synonym'  '"regulation of heterochromatin formation" EXACT []'}
          is_a: [3×1 double]
       part_of: [0×1 double]
      obsolete: 0
```

Create a cell array of character vectors that list the ontology property for each term in the `geneont` object.

```
ontologies = get(GeneontObj.terms,'ontology');
```

Create a logical mask that identifies all the terms with an `ontology` property of `'cellular component'`.

```
mask = strcmp(ontologies,'cellular component');
```

Apply the logical mask to all the terms `GeneontObj` to return a structure array of `term` objects, containing only terms with an ontology property of `'cellular component'`.

```
cell_comp_terms = GeneontObj.terms(mask)
```

```
  4469×1 struct array with fields:

    id
    name
    ontology
    definition
    comment
    synonym
    is_a
    part_of
    obsolete
```

Create a subontology of all the cellular component terms by indexing into `GeneontObj` with the masked term objects.

```
subontology = GeneontObj(cell_comp_terms)
```

```
Gene Ontology object with 4469 Terms.
```

Index Into a `geneont` Object Using the GO Identifier

This example shows how to create a subontology by indexing into a `geneont` object by using the GO identifier.

Download the current version of the Gene Ontology database from the Web into a `geneont` object.

```
GeneontObj = geneont('LIVE', true)
```

```
Gene Ontology object with 47266 Terms.
```

Create a subontology by returning the terms with GO identifiers of GO:000005 through GO:000010.

```
subontology1 = GeneontObj(5:10)
```

```
Gene Ontology object with 6 Terms.
```

Create a subontology by returning the term with a GO identifier of GO:000100.

```
subontology2 = GeneontObj(100)
```

```
Gene Ontology object with 1 Terms.
```

Index Into a geneont Object Using the GO Term

You can create a subontology by indexing into a `geneont` object using the GO term.

Download the current version of the Gene Ontology database from the Web into a `geneont` object.

```
GeneontObj = geneont('LIVE', true)
```

```
Gene Ontology object with 47266 Terms.
```

Create an array of `term` objects containing the fifth through tenth terms of the `geneont` object.

```
termObject = GeneontObj.terms(5:10)
```

```
  6×1 struct array with fields:

    id
    name
    ontology
    definition
    comment
    synonym
    is_a
    part_of
    obsolete
```

Note: The GO term of 5 is the position of the term object in the `geneont` object, and is not necessarily the same as the `term` object with a GO identifier of GO:000005 used in the example "Index Into a `geneont` Object using the GO Identifier." This is because there are many terms that are obsolete and are not included as `term` objects in the `geneont` object.

Create a subontology by returning the fifth through tenth terms of the `geneont` object.

```
subontology3 = GeneontObj(termObject)
```

```
Gene Ontology object with 6 Terms.
```

Version History

Introduced before R2006a

See Also

`goannotread` | `num2goid` | `term`

Topics

"Data Formats and Databases"

generangefilter

Remove gene profiles with small profile ranges

Syntax

```
Mask = generangefilter(Data)
[Mask, FData] = generangefilter(Data)
[Mask, FData, FNames] = generangefilter(Data, Names)

generangefilter(..., 'Percentile', PercentileValue, ...)
generangefilter(..., 'AbsValue', AbsValueValue, ...)
generangefilter(..., 'LogPercentile', LogPercentileValue, ...)
generangefilter(..., 'LogValue', LogValueValue, ...)
```

Arguments

`Data`	DataMatrix object on page 1-734 or numeric matrix where each row corresponds to the experimental results for one gene. Each column is the results for all genes from one experiment.
`Names`	Cell array of character vectors or string vector where each element corresponds to the name of a gene for each row of experimental data. *`Names`* has same number of rows as *`Data`* with each row containing the name or ID of the gene in the data set.
`PercentileValue`	Property to specify a percentile below which gene expression profiles are removed. Enter a value from `0` to `100`.
`AbsValueValue`	Property to specify an absolute value below which gene expression profiles are removed.
`LogPercentileValue`	Property to specify the logarithm of a percentile.
`LogValueValue`	Property to specify the logarithm of an absolute value.

Description

`Mask` `= generangefilter(`*`Data`*`)` calculates the range for each gene expression profile in *`Data`*, a DataMatrix object on page 1-734 or matrix of the experimental data, and then identifies the expression profiles with ranges less than the 10th percentile.

`Mask` is a logical vector with one element for each row in *`Data`*. The elements of *`Mask`* corresponding to rows with a range greater than the threshold have a value of `1`, and those with a range less than the threshold are `0`.

`[`*`Mask`*`, `*`FData`*`] = generangefilter(`*`Data`*`)` returns *`FData`*, a filtered data matrix. You can also create *`FData`* using *`FData`* `= Data(`*`Mask`*`,:)`.

`[`*`Mask`*`, `*`FData`*`, `*`FNames`*`] = generangefilter(`*`Data`*`, `*`Names`*`)` returns *`FNames`*, a filtered names array, where *`Names`* is a cell array of character vectors or string vector of the names of the genes corresponding to each row in *`Data`*. You can also create *`FNames`* using *`FNames`* `= Names(`*`Mask`*`)`.

Note If *`Data`* is a DataMatrix object with specified row names, you do not need to provide the second input *`Names`* to return the third output *`FNames`*.

`generangefilter(..., '`*`PropertyName`*`', `*`PropertyValue`*`, ...)` calls `generangefilter` with optional properties that use property name/property value pairs. You can specify one or more properties in any order. Each *`PropertyName`* must be enclosed in single quotation marks and is case insensitive. These property name/property value pairs are as follows:

`generangefilter(..., 'Percentile', `*`PercentileValue`*`, ...)` removes from the experimental data (*`Data`*) gene expression profiles with ranges less than a specified percentile (*`PercentileValue`*).

`generangefilter(..., 'AbsValue', `*`AbsValueValue`*`, ...)` removes from *`Data`* gene expression profiles with ranges less than *`AbsValueValue`*.

`generangefilter(..., 'LogPercentile', `*`LogPercentileValue`*`, ...)` filters genes with profile ranges in the lowest percent of the log range (*`LogPercentileValue`*).

`generangefilter(..., 'LogValue', `*`LogValueValue`*`, ...)` filters genes with profile log ranges lower than *`LogValueValue`*.

Examples

1. Load the MAT-file, provided with the Bioinformatics Toolbox software, that contains yeast data. This MAT-file includes three variables: `yeastvalues`, a matrix of gene expression data, `genes`, a cell array of GenBank accession numbers for labeling the rows in `yeastvalues`, and `times`, a vector of time values for labeling the columns in `yeastvalues`

   ```
   load yeastdata
   ```

2. Remove gene profiles with small profile ranges.

   ```
   [mask, fyeastvalues, fgenes] = generangefilter(yeastvalues,genes);
   ```

Version History

Introduced before R2006a

References

[1] Kohane I.S., Kho A.T., Butte A.J. (2003), Microarrays for an Integrative Genomics, Cambridge, MA:MIT Press.

See Also

`exprprofrange` | `exprprofvar` | `geneentropyfilter` | `genelowvalfilter` | `genevarfilter`

geneticcode

Return nucleotide codon to amino acid mapping for genetic code

Syntax

```
Map = geneticcode
Map = geneticcode(GeneticCode)
```

Input Arguments

GeneticCode	Integer, character vector, or string specifying a genetic code number or code name from the table Genetic Code. Default is `1` or `'Standard'`. **Tip** If you use a code name, you can truncate the name to the first two letters of the name.

Output Arguments

Map	Structure containing the mapping of nucleotide codons to amino acids for the standard genetic code. The *Map* structure contains a field for each nucleotide codon.

Description

`Map = geneticcode` returns a structure containing the mapping of nucleotide codons to amino acids for the standard genetic code. The *Map* structure contains a field for each nucleotide codon.

`Map = geneticcode(GeneticCode)` returns a structure containing the mapping of nucleotide codons to amino acids for the specified genetic code. *GeneticCode* is either:

- An integer, character vector, or string specifying a code number or code name from the table Genetic Code
- The `transl_table` (code) number from the NCBI Web page describing genetic codes:

  `https://www.ncbi.nlm.nih.gov/Taxonomy/Utils/wprintgc.cgi?mode=c`

Tip If you use a code name, you can truncate the name to the first two letters of the name.

Genetic Code

Code Number	Code Name
1	Standard
2	Vertebrate Mitochondrial
3	Yeast Mitochondrial
4	Mold, Protozoan, Coelenterate Mitochondrial, and Mycoplasma/ Spiroplasma
5	Invertebrate Mitochondrial
6	Ciliate, Dasycladacean, and Hexamita Nuclear
9	Echinoderm Mitochondrial
10	Euplotid Nuclear
11	Bacterial and Plant Plastid
12	Alternative Yeast Nuclear
13	Ascidian Mitochondrial
14	Flatworm Mitochondrial
15	Blepharisma Nuclear
16	Chlorophycean Mitochondrial
21	Trematode Mitochondrial
22	Scenedesmus Obliquus Mitochondrial
23	Thraustochytrium Mitochondrial

Examples

Return the mapping of nucleotide codons to amino acids for the Flatworm Mitochondrial genetic code.

```
wormmap = geneticcode('Flatworm Mitochondrial');
```

Version History

Introduced before R2006a

References

[1] NCBI Web page describing genetic codes:

`https://www.ncbi.nlm.nih.gov/Taxonomy/Utils/wprintgc.cgi?mode=c`

See Also

aa2nt | aminolookup | baselookup | codonbias | dnds | dndsml | nt2aa | revgeneticcode | seqshoworfs | seqviewer

genevarfilter

Filter genes with small profile variance

Syntax

```
Mask = genevarfilter(Data)
[Mask, FData] = genevarfilter(Data)
[Mask, FData, FNames] = genevarfilter(Data, Names)

genevarfilter(..., 'Percentile', PercentileValue, ...)
genevarfilter(..., 'AbsValue', AbsValueValue, ...)
```

Arguments

Data	DataMatrix object on page 1-734 or numeric matrix where each row corresponds to a gene. If a matrix, the first column is the names of the genes, and each additional column is the results from an experiment.
Names	Cell array of character vectors or string vector where each element corresponds to the name of a gene for each row of experimental data. *Names* has same number of rows as *Data* with each row containing the name or ID of the gene in the data set.
PercentileValue	Specifies a percentile below which gene expression profiles are removed. Choices are integers from `0` to `100`. Default is `10`.
AbsValueValue	Property to specify an absolute value below which gene expression profiles are removed.

Description

Gene profiling experiments typically include genes that exhibit little variation in their profile and are generally not of interest. These genes are commonly removed from the data.

`Mask = genevarfilter(Data)` calculates the variance for each gene expression profile in *Data* and returns *Mask*, which identifies the gene expression profiles with a variance less than the 10th percentile. *Mask* is a logical vector with one element for each row in *Data*. The elements of *Mask* corresponding to rows with a variance greater than the threshold have a value of `1`, and those with a variance less than the threshold are `0`.

`[Mask, FData] = genevarfilter(Data)` calculates the variance for each gene expression profile in *Data* and returns *FData*, a filtered data matrix, in which the low-variation gene expression profiles are removed. You can also create *FData* using `FData = Data(Mask,:)`.

`[Mask, FData, FNames] = genevarfilter(Data, Names)` returns *FNames*, a filtered names array, in which the names associated with low-variation gene expression profiles are removed. *Names* is a cell array of character vectors or string vector of the names of the genes corresponding to each row in *Data*. You can also create *FNames* using `FNames = Names(Mask)`.

Note If *`Data`* is a DataMatrix object with specified row names, you do not need to provide the second input *`Names`* to return the third output *`FNames`*.

`genevarfilter(..., '`*`PropertyName`*`', `*`PropertyValue`*`, ...)` calls `genevarfilter` with optional properties that use property name/property value pairs. You can specify one or more properties in any order. Each *`PropertyName`* must be enclosed in single quotation marks and is case insensitive. These property name/property value pairs are as follows:

`genevarfilter(..., 'Percentile', `*`PercentileValue`*`, ...)` removes from *`Data`*, the experimental data, the gene expression profiles with a variance less than the percentile specified by *`PercentileValue`*. Choices are integers from `0` to `100`. Default is `10`.

`genevarfilter(..., 'AbsValue', `*`AbsValueValue`*`, ...)` removes from *`Data`* , the experimental data, the gene expression profiles with a variance less than *`AbsValueValue`*.

Examples

1. Load the MAT-file, provided with the Bioinformatics Toolbox software, that contains yeast data. This MAT-file includes three variables: `yeastvalues`, a matrix of gene expression data, `genes`, a cell array of GenBank accession numbers for labeling the rows in `yeastvalues`, and `times`, a vector of time values for labeling the columns in `yeastvalues`

   ```
   load yeastdata
   ```
2. Filter genes with a small profile variance.

   ```
   [fyeastvalues, fgenes] = genevarfilter(yeastvalues,genes);
   ```

Version History

Introduced before R2006a

References

[1] Kohane I.S., Kho A.T., Butte A.J. (2003), Microarrays for an Integrative Genomics, Cambridge, MA:MIT Press.

See Also

`exprprofrange` | `exprprofvar` | `generangefilter` | `geneentropyfilter` | `genelowvalfilter`

Topics

DataMatrix object on page 1-734

genpeptread

Read data from GenPept file

Syntax

```
GenPeptData = genpeptread(File)
GenPeptData = genpeptread(File, 'TimeOut', TimeOutValue)
```

Arguments

File	Either of the following: • Character vector or string specifying a file name, a path and file name, or a URL pointing to a file. The referenced file is a GenPept-formatted file. If you specify only a file name, that file must be on the MATLAB search path or in the MATLAB Current Folder. • MATLAB character array that contains the text of a GenPept-formatted file. **Tip** You can use the `getgenpept` function with the `'ToFile'` property to retrieve sequence information from the GenPept database and create an GenPept-formatted file.
TimeOutValue	Connection timeout in seconds, specified as a positive scalar. The default value is 5. For details, see here.
GenPeptData	MATLAB structure or array of structures containing fields corresponding to GenPept keywords.

Description

Note NCBI has changed the name of their protein search engine from GenPept to Entrez Protein. However, the function names in the Bioinformatics Toolbox software (`getgenpept` and `genpeptread`) are unchanged representing the still-used GenPept report format.

`GenPeptData = genpeptread(File)` reads a GenPept-formatted file, `File`, and creates *GenPeptData*, a structure or array of structures, containing fields corresponding to the GenPept keywords. When *File* contains multiple entries, each entry is stored as a separate element in *GenPeptData*. For a list of the GenPept keywords, see https://www.ncbi.nlm.nih.gov/Sitemap/samplerecord.html.

`GenPeptData = genpeptread(File, 'TimeOut', TimeOutValue)` sets the connection timeout (in seconds) to read data from a remote file or URL.

Examples

Retrieve sequence information for the protein coded by the HEXA gene, store the data in a file, and then read into the MATLAB software.

```
getgenpept('p06865', 'ToFile', 'TaySachs_Protein.txt')
genpeptread('TaySachs_Protein.txt')
```

Version History

Introduced before R2006a

See Also

`fastaread` | `genbankread` | `getgenpept` | `pdbread` | `seqviewer`

geoseriesread

Read Gene Expression Omnibus (GEO) Series (GSE) format data

Syntax

```
GEOData = geoseriesread(File)
GEOData = geoseriesread(File,'TimeOut',TimeOutValue)
```

Input Arguments

File	Either of the following: • Character vector or string specifying a file name, a path and file name, or a URL pointing to a file. The referenced file is a Gene Expression Omnibus (GEO) Series (GSE) format file. If you specify only a file name, that file must be on the MATLAB search path or in the MATLAB current folder. • Character array or column vector of strings that contains the text of a GEO Series (GSE) format file. **Tip** You can use the `getgeodata` function with the `'ToFile'` property to retrieve GEO Series (GSE) format data from the GEO database and create a GEO Series (GSE) format file.
TimeOutValue	Connection timeout in seconds, specified as a positive scalar. The default value is 5. For details, see here.

Output Arguments

GEOData	MATLAB structure containing the following fields: • `Header` — Header text from the GEO Series (GSE) format file, typically containing a description of the data or experiment information. • `Data` — DataMatrix object on page 1-734 containing the data from a GEO Series (GSE) format file. The columns and rows of the DataMatrix object correspond to the sample IDs and Ref IDs, respectively, from the GEO Series (GSE) format file.

Description

`GEOData = geoseriesread(File)` reads a Gene Expression Omnibus (GEO) Series (GSE) format file, and then creates a MATLAB structure, *GEOData*, with the following fields.

Fields	Description
`Header`	Header text from the GEO Series (GSE) format file, typically containing a description of the data or experiment information.

Fields	Description
Data	DataMatrix object on page 1-734 containing the data from a GEO Series (GSE) format file. The columns and rows of the DataMatrix object correspond to the sample IDs and Ref IDs, respectively, from the GEO Series (GSE) format file.

GEOData = geoseriesread(*File*,'TimeOut',*TimeOutValue*) sets the connection timeout (in seconds) to read data from a remote file or URL.

Examples

1 Retrieve Series (GSE) data from the GEO Web site and save it to a file.

```
geodata = getgeodata('GSE11287','ToFile','GSE11287.txt');
```

2 In a subsequent MATLAB session, you can access the Series (GSE) data from your local file, instead of retrieving it from the GEO Web site.

```
geodata = geoseriesread('GSE11287.txt')

geodata = 

    Header: [1x1 struct]
      Data: [45101x6 bioma.data.DataMatrix]
```

3 Access the sample IDs using the colnames property of a DataMatrix object.

```
sampleIDs = geodata.Data.colnames

sampleIDs = 

  'GSM284935'  'GSM284936'  'GSM284937'  'GSM284938'  'GSM284939'  'GSM284940'
```

Version History

Introduced in R2008b

See Also

affyread | agferead | galread | geosoftread | getgeodata | gprread | ilmnbsread | sptread

Topics

DataMatrix object on page 1-734

geosoftread

Read Gene Expression Omnibus (GEO) SOFT format data

Syntax

```
GEOSOFTData = geosoftread(File)
GEOSOFTData = geosoftread(File,'TimeOut',TimeOutValue)
```

Input Arguments

File	Either of the following: • Character vector or string specifying a file name, a path and file name, or a URL pointing to a file. The referenced file is a Gene Expression Omnibus (GEO) SOFT format Sample file (GSM), Data Set file (GDS), or Platform (GPL) file. If you specify only a file name, that file must be on the MATLAB search path or in the MATLAB Current Folder. • Character array or column vector of strings that contains the text of a GEO SOFT format file. **Tip** You can use the `getgeodata` function with the `'ToFile'` property to retrieve GEO SOFT format data from the GEO database and create a GEO SOFT format file.
TimeOutValue	Connection timeout in seconds, specified as a positive scalar. The default value is 5. For details, see here.

Output Arguments

GEOSOFTData	MATLAB structure containing information from a GEO SOFT format file.

Description

`GEOSOFTData` `= geosoftread(`*`File`*`)` reads a Gene Expression Omnibus (GEO) SOFT format Sample file (GSM), Data Set file (GDS), or Platform (GPL) file, and then creates a MATLAB structure, *`GEOSOFTData`*, with the following fields.

Fields	Description
`Scope`	Type of file read (SAMPLE, DATASET, or PLATFORM)
`Accession`	Accession number for record in GEO database.
`Header`	Microarray experiment information.
`ColumnDescriptions`	Cell array containing descriptions of columns in the data.
`ColumnNames`	Cell array containing names of columns in the data.
`Data`	Array containing microarray data.

Fields	Description
`Identifier` (GDS files only)	Cell array containing probe IDs.
`IDRef` (GDS files only)	Cell array containing indices to probes.

Note Currently, the `geosoftread` function supports Sample (GSM), Data Set (GDS), and Platform (GPL) records.

`GEOSOFTData = geosoftread(File,'TimeOut',TimeOutValue)` sets the connection timeout (in seconds) to read data from a remote file or URL.

Examples

Import Gene Expression Omnius SOFT-Formatted Data

Download the SOFT format sample (GSM) file from the Gene Expression Omnius GEO web site to a file.

```
geodata = getgeodata('GSM3258','ToFile','GSM3258.txt');
```

Use `geosoftread` to read a local copy of the GSM file, instead of accessing it from the GEO web site.

```
geodata = geosoftread('GSM3258.txt');
```

Download the GDS (GEO Data Set) file containing data for the molecular investigation of photosynthesis in proteobacteria.

```
gdsdata = geosoftread('GDS329.soft')
```

```
gdsdata = struct with fields:
                 Scope: 'DATASET'
             Accession: 'GDS329'
                Header: [1×1 struct]
    ColumnDescriptions: {6×1 cell}
           ColumnNames: {6×1 cell}
                 IDRef: {5355×1 cell}
            Identifier: {5355×1 cell}
                  Data: [5355×6 double]
```

Version History
Introduced before R2006a

See Also
`galread` | `getgeodata` | `geoseriesread` | `gprread` | `ilmnbsread` | `sptread`

get (biograph)

(To be removed) Retrieve information about biograph object

Note The `biograph` object and its methods will be removed in a future release. Use `graph` or `digraph` instead.

Syntax

```
get(BGobj)
BGStruct = get(BGobj)
PropertyValue = get(BGobj, 'PropertyName')
[Property1Value, Property2Value, ...] = get(BGobj, 'Property1Name',
'Property2Name', ...)
```

Input Arguments

BGobj	Biograph object created with the function `biograph`.
PropertyName	Property name for a biograph object.

Output Arguments

BGStruct	Scalar structure, in which each field name is a property of a biograph object, and each field contains the value of that property.
PropertyValue	Value of the property specified by *PropertyName*.

Description

`get(`*`BGobj`*`)` displays all properties and their current values of *BGobj*, a biograph object.

`BGStruct`` = get(`*`BGobj`*`)` returns all properties of *BGobj*, a biograph object, to *BGStruct*, a scalar structure, in which each field name is a property of a biograph object, and each field contains the value of that property.

`PropertyValue`` = get(`*`BGobj`*`, '`*`PropertyName`*`')` returns the value of the specified property of *BGobj*, a biograph object.

`[`*`Property1Value`*`, `*`Property2Value`*`, ...] = get(`*`BGobj`*`, '`*`Property1Name`*`', '`*`Property2Name`*`', ...)` returns the values of the specified properties of *BGobj*, a biograph object.

Properties of a Biograph Object

Property	Description
`ID`	Character vector to identify the biograph object. Default is `''`.
`Label`	Character vector to label the biograph object. Default is `''`.
`Description`	Character vector that describes the biograph object. Default is `''`.
`LayoutType`	Character vector that specifies the algorithm for the layout engine. Choices are: • `'hierarchical'` (default) — Uses a topological order of the graph to assign levels, and then arranges the nodes from top to bottom, while minimizing crossing edges. • `'radial'` — Uses a topological order of the graph to assign levels, and then arranges the nodes from inside to outside of the circle, while minimizing crossing edges. • `'equilibrium'` — Calculates layout by minimizing the energy in a dynamic spring system.
`EdgeType`	Character vector that specifies how edges display. Choices are: • `'straight'` • `'curved'` (default) • `'segmented'` **Note** Curved or segmented edges occur only when necessary to avoid obstruction by nodes. Biograph objects with `LayoutType` equal to `'equilibrium'` or `'radial'` cannot produce curved or segmented edges.
`Scale`	Positive number that post-scales the node coordinates. Default is `1`.
`LayoutScale`	Positive number that scales the size of the nodes before calling the layout engine. Default is `1`.
`EdgeTextColor`	Three-element numeric vector of RGB values. Default is `[0, 0, 0]`, which defines black.
`EdgeFontSize`	Positive number that sets the size of the edge font in points. Default is `8`.
`ShowArrows`	Controls the display of arrows with the edges. Choices are `'on'` (default) or `'off'`.
`ArrowSize`	Positive number that sets the size of the arrows in points. Default is `8`.
`ShowWeights`	Controls the display of text indicating the weight of the edges. Choices are `'on'` or `'off'` (default).

Property	Description
`ShowTextInNodes`	Character vector that specifies the node property used to label nodes when you display a biograph object using the `view` method. Choices are: • `'Label'` — Uses the `Label` property of the node object (default). • `'ID'` — Uses the `ID` property of the node object. • `'None'`
`NodeAutoSize`	Controls precalculating the node size before calling the layout engine. Choices are `'on'` (default) or `'off'`. **Note** Set it to `off` if you want to apply different node sizes by changing the `Size` property.
`NodeCallback`	User-defined callback for all nodes. Enter the name of a function, a function handle, or a cell array with multiple function handles. After using the `view` function to display the biograph object in the Biograph Viewer, you can double-click a node to activate the first callback, or right-click and select a callback to activate. Default is the anonymous function, `@(node) inspect(node)`, which displays the Property Inspector dialog box.
`EdgeCallback`	User-defined callback for all edges. Enter the name of a function, a function handle, or a cell array with multiple function handles. After using the `view` function to display the biograph object in the Biograph Viewer, you can right-click and select a callback to activate. Default is the anonymous function, `@(edge) inspect(edge)`, which displays the Property Inspector dialog box.
`CustomNodeDrawFcn`	Function handle to a customized function to draw nodes. Default is `[]`.
`Nodes`	Read-only column vector with handles to node objects of a biograph object. The size of the vector is the number of nodes. For properties of node objects, see Properties of a Node Object.
`Edges`	Read-only column vector with handles to edge objects of a biograph object. The size of the vector is the number of edges. For properties of edge objects, see Properties of an Edge Object.

Version History

Introduced in R2008b

R2021b: `biograph` and its methods will be removed

Not recommended starting in R2021b

The `biograph` object and its methods will be removed in a future release. Use `graph` or `digraph` instead.

See Also

`graph` | `digraph`

get

Retrieve property of object

Syntax

```
S = get(object)
propertyValues = get(object,propertyNames)
```

Description

`S = get(object)` returns a structure containing a field for each property of a `BioRead` or `BioMap` `object`.

`propertyValues = get(object,propertyNames)` returns the values of the properties specified by `propertyNames`.

Examples

Retrieve Sequencing Data from BioRead

Store read data from a SAM-formatted file in a `BioRead` object.

```
br = BioRead('ex1.sam')
```

```
br =
  BioRead with properties:

     Quality: [1501x1 File indexed property]
    Sequence: [1501x1 File indexed property]
      Header: [1501x1 File indexed property]
       NSeqs: 1501
        Name: ''
```

Retrieve the data from the object. `data` is a structure containing a field for each object property, such as `Quality`, `Sequence`, `Header`, `NSeqs`, and `Name`.

```
data = get(br)
```

```
data = struct with fields:
     Quality: {1501x1 cell}
    Sequence: {1501x1 cell}
      Header: {1501x1 cell}
       NSeqs: 1501
        Name: ''
```

Retrieve the header data only.

```
headers = get(br,'Header');
```

You can also retrieve a subset of properties.

```
subset = get(br,{'Header','NSeqs'});
```

Input Arguments

`object` — Object containing read data
`BioRead` object | `BioMap` object

Object containing the read data, specified as a `BioRead` or `BioMap` object.

Example: `bioreadObj`

`propertyNames` — Names of object properties
character vector | string | string vector | cell array of character vectors

Names of the object properties, specified as a character vector, string, string vector, or cell array of character vectors.

Example: `{'Quality','Sequence'}`

Output Arguments

`S` — Object property data
structure

Object property data, returned as a structure containing a field for each property of the object.

`propertyValues` — Property values
cell array

Property values, returned as a cell array.

Version History

Introduced in R2010a

See Also

`BioRead` | `BioMap`

Topics
“Manage Sequence Read Data in Objects”

get (DataMatrix)

Retrieve information about DataMatrix object

Syntax

```
get(DMObj)
DMStruct = get(DMObj)
PropertyValue = get(DMObj, 'PropertyName')
[Property1Value, Property2Value, ...] = get(DMObj, 'Property1Name',
'Property2Name', ...)
```

Input Arguments

DMObj	DataMatrix object, such as created by `DataMatrix` (object constructor).
PropertyName	Property name of a DataMatrix object.

Output Arguments

DMStruct	Scalar structure, in which each field name is a property of a DataMatrix object, and each field contains the value of that property.
PropertyValue	Value of the property specified by *PropertyName*.

Description

`get(`*DMObj*`)` displays all properties and their current values of *DMObj*, a DataMatrix object.

DMStruct `= get(`*DMObj*`)` returns all properties of *DMObj*, a DataMatrix object, to *DMStruct*, a scalar structure, in which each field name is a property of a DataMatrix object, and each field contains the value of that property.

PropertyValue `= get(`*DMObj*`, '`*PropertyName*`')` returns the value of the specified property of *DMObj*, a DataMatrix object.

`[`*Property1Value*`, `*Property2Value*`, ...] = get(`*DMObj*`, '`*Property1Name*`', '`*Property2Name*`', ...)` returns the values of the specified properties of *DMObj*, a DataMatrix object.

Properties of a DataMatrix Object

Property	Description
`Name`	Character vector that describes the DataMatrix object. Default is `''`.
`RowNames`	Empty array or cell array of character vectors that specifies the names for the rows, typically gene names or probe identifiers. The number of elements in the cell array must equal the number of rows in the matrix. Default is an empty array.
`ColNames`	Empty array or cell array of character vectors that specifies the names for the columns, typically sample identifiers. The number of elements in the cell array must equal the number of columns in the matrix.
`NRows`	Positive number that specifies the number of rows in the matrix.
`NCols`	Positive number that specifies the number of columns in the matrix.
`NDims`	Positive number that specifies the number of dimensions in the matrix.
`ElementClass`	Character vector that specifies the class type, such as `single` or `double`.

Examples

1. Load the MAT-file, provided with the Bioinformatics Toolbox software, that contains yeast data. This MAT-file includes three variables: `yeastvalues`, a matrix of gene expression data, `genes`, a cell array of GenBank accession numbers for labeling the rows in `yeastvalues`, and `times`, a vector of time values for labeling the columns in `yeastvalues`.

   ```
   load filteredyeastdata
   ```
2. Import the microarray object package so that the `DataMatrix` constructor function will be available.

   ```
   import bioma.data.*
   ```
3. Create a DataMatrix object from the gene expression data in the first 30 rows of the `yeastvalues` matrix. Use the `genes` column vector and `times` row vector to specify the row names and column names.

   ```
   dmo = DataMatrix(yeastvalues(1:30,:),genes(1:30,:),times);
   ```
4. Use the `get` method to display the properties of the DataMatrix object, `dmo`.

   ```
   get(dmo)

             Name: ''
         RowNames: {30x1 cell}
         ColNames: {'   0'  ' 9.5'  '11.5'  '13.5'  '15.5'  '18.5'  '20.5'}
            NRows: 30
            NCols: 7
            NDims: 2
     ElementClass: 'double'
   ```

Version History

Introduced in R2008b

See Also
`DataMatrix` | `set`

Topics
DataMatrix object on page 1-734

get (phytree)

Retrieve information about phylogenetic tree object

Syntax

```
[Value1, Value2,...] = get(Tree, 'Property1','Property2',...)
get(Tree)
V = get(Tree)
```

Arguments

Tree	Phytree object created with the function `phytree`.
Name	Property name for a `phytree` object.

Description

`[Value1, Value2,...] = get(Tree, 'Property1','Property2',...)` returns the specified properties from a phytree object (`Tree`).

Properties for a `phytree` object are listed in the following table.

Property	Description
`NumLeaves`	Number of leaves
`NumBranches`	Number of branches
`NumNodes`	Number of nodes (`NumLeaves + NumBranches`)
`Pointers`	Branch to leaf/branch connectivity list
`Distances`	Edge length for every leaf/branch
`LeafNames`	Names of the leaves
`BranchNames`	Names of the branches
`NodeNames`	Names of all the nodes

`get(Tree)` displays all property names and their current values for a phytree object (*Tree*).

`V = get(Tree)` returns a structure where each field name is the name of a property of a phytree object (*Tree*) and each field contains the value of that property.

Examples

1 Read in a phylogenetic tree from a file.

```
tr = phytreeread('pf00002.tree')
```

```
Phylogenetic tree object with 33 leaves (32 branches)
```

2 Get the names of the leaves.

```
protein_names = get(tr,'LeafNames')

protein_names = 

    'Q9YHC6_RANRI/126-382'
    'VIPR1_RAT/140-397'
    'VIPR_CARAU/100-359'
     ...
```

Version History

Introduced before R2006a

See Also

`phytree` | `phytreeread` | `getbyname` | `select`

Topics

phytree object on page 1-1449

getancestors (biograph)

(Removed) Find ancestors of a node in biograph object

Note The function has been removed. Use `predecessors` instead. Note that the function does not let you specify `NumGenerations`. You can use `distances` to find the distances from all nodes to the target node after setting the edge weight to 1, and filter the nodes by distance to the appropriate depth.

Syntax

```
Node = getancestors(BiographNode)
Node = getancestors(BiographNode, NumGenerations)
```

Arguments

BiographNode	Node in a biograph object.
NumGenerations	Number of generations. Enter a positive integer.

Description

`Node = getancestors(BiographNode)` returns a node (`BiographNode`) and all of its direct ancestors.

`Node = getancestors(BiographNode, NumGenerations)` finds the node (*BiographNode*) and its direct ancestors up to a specified number of generations (*NumGenerations*). If *NumGenerations* is `0`, the function returns the node itself.

Version History

Introduced before R2006a

R2022b: Removed

Errors starting in R2022b

The function has been removed. Use `predecessors` instead. Note that the function does not let you specify `NumGenerations`. You can use `distances` to find the distances from all nodes to the target node after setting the edge weight to 1, and filter the nodes by distance to the appropriate depth.

R2022a: Warns

Warns starting in R2022a

The function issues a warning that it will be removed in a future release.

R2021b: To be removed

Not recommended starting in R2021b

The function runs without warning, but it will be removed in a future release.

See Also

`graph` | `digraph` | `distances` | `predecessors`

getancestors

Find terms that are ancestors of specified Gene Ontology (GO) term

Syntax

```
AncestorIDs = getancestors(GeneontObj,ID)
[AncestorIDs,Counts] = getancestors(GeneontObj,ID)
___ = getancestors(GeneontObj,ID,Name,Value)
```

Description

`AncestorIDs = getancestors(GeneontObj,ID)` searches `GeneontObj`, a `geneont` object, for GO terms that are ancestors of the GO term(s) specified by `ID`, which is a GO term identifier or vector of identifiers. The result `AncestorIDs` is a vector of GO term identifiers including `ID`.

`[AncestorIDs,Counts] = getancestors(GeneontObj,ID)` also returns the number of times each ancestor is found. `Counts` is a column vector with the same number of elements as terms in `GeneontObj`.

Tip The `Counts` return value is useful when you tally counts in gene enrichment studies.

`___ = getancestors(GeneontObj,ID,Name,Value)`, for any output arguments, specifies additional options using one or more name-value arguments. For example, you can restrict the search to have up to two levels up in the gene ontology by specifying `AncestorIDs = getancestors(GeneontObj,ID,Height=2)`.

Examples

Get Ancestors of `geneont` Object

Download the current version of the Gene Ontology database from the Web into a `geneont` object.

```
GO = geneont('Live',true)
```

```
Gene Ontology object with 47266 Terms.
```

Retrieve the ancestors of the Gene Ontology term with an identifier of 46680.

```
ancestors = getancestors(GO,46680)
```

```
ancestors = 8×1

        8150
        9636
       10033
       14070
       17085
       42221
       46680
       50896
```

Create a subordinate Gene Ontology.

```
subontology = GO(ancestors)
```

```
Gene Ontology object with 8 Terms.
```

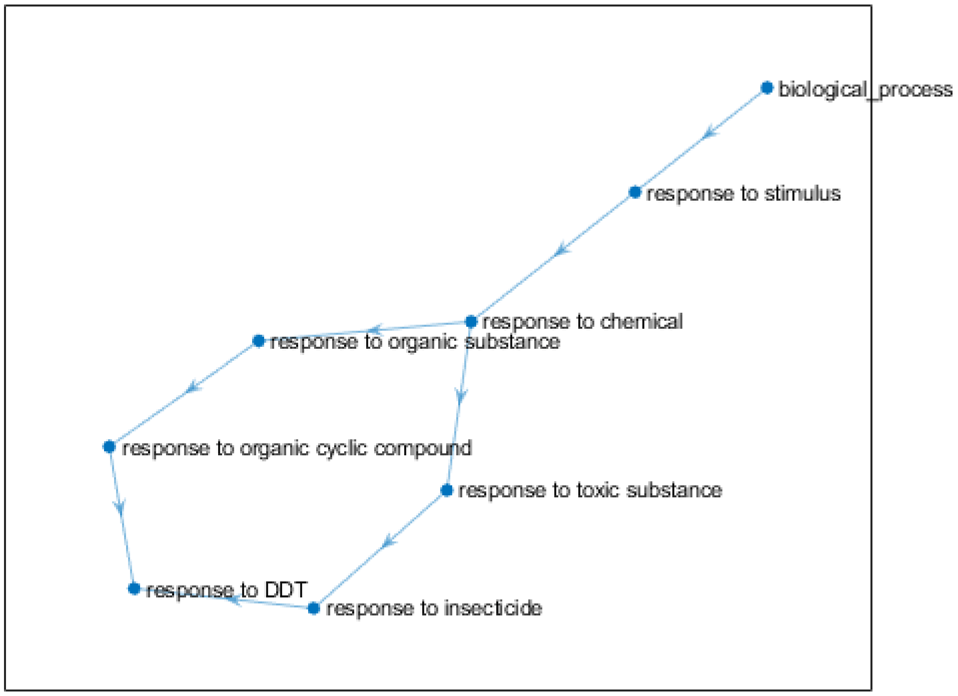
Create and display a report of the subordinate Gene Ontology terms, that includes the GO identifier and name.

```
rpt = get(subontology.terms,{'id','name'})
```

```
rpt=8×2 cell array
    {[ 8150]}    {'biological_process'                }
    {[ 9636]}    {'response to toxic substance'       }
    {[10033]}    {'response to organic substance'     }
    {[14070]}    {'response to organic cyclic compound'}
    {[17085]}    {'response to insecticide'           }
    {[42221]}    {'response to chemical'              }
    {[46680]}    {'response to DDT'                   }
    {[50896]}    {'response to stimulus'              }
```

View relationships of the subordinate Gene Ontology by using the `getmatrix` method to create a connection matrix to pass to the `digraph` function.

```
cm = getmatrix(subontology);
BG = digraph(cm,get(subontology.terms,"name"));
plot(BG,Interpreter="none",Layout="force")
```



Input Arguments

`GeneontObj` — Gene ontology object

output of `geneont`

Gene ontology object, specified as the output of the `geneont` command.

Example: `geneont('Live',true)`

`ID` — Gene ontology ID numbers

nonnegative integer vector

Gene ontology ID numbers, specified as a nonnegative integer vector.

Example: `5`

Data Types: `single` | `double`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Names are case-insensitive.

Example: `AncestorIDs = getancestors(GeneontObj,5,Height=2)`

`Height` — Number of search levels up in gene ontology
`Inf` (default) | positive integer

Number of search levels up in gene ontology, specified as a positive integer.

Example: `4`

Data Types: `single` | `double`

`Relationtype` — Relations to match
`'both'` (default) | `'is_a'` | `'part_of'`

Relations to match, specified as one of the following:

- `'is_a'`
- `'part_of'`
- `'both'`

Example: `'is_a'`

Data Types: `char` | `string`

`Exclude` — Indication to exclude `ID` from `AncestorIDs`
`false` (default) | `true`

Indication to exclude `ID` from `AncestorIDs`, specified as `false` or `true`. This argument can fail to apply when `getancestors` reaches the term while searching the ontology.

Example: `true`

Data Types: `logical`

Output Arguments

`AncestorIDs` — ID numbers of ancestors of `GeneontObj`
nonnegative integer vector

ID numbers of ancestors of `GeneontObj`, returned as a nonnegative integer vector. By default, `AncestorIDs` includes `ID`, unless you set the `Exclude` argument to `true`.

`Counts` — Number of times each relative is found
column vector of integers

Number of times each relative is found, returned as a column vector of integers. `Counts` has the same number of elements as terms in `GeneontObj`.

Version History

Introduced before R2006a

See Also

`geneont` | `goannotread` | `num2goid` | `term`

getblast

Retrieve BLAST report from NCBI website

Syntax

```
blastdata = getblast(RID)
blastdata = getblast(RID,Name,Value)
```

Description

`blastdata = getblast(RID)` retrieves `RID`, the Request ID for the NCBI BLAST report, and returns the report data in `blastdata`, a MATLAB structure. The Request ID, `RID`, must be recent because NCBI purges reports after 36 hours.

`blastdata = getblast(RID,Name,Value)` uses additional options specified by one or more name-value pair arguments.

Examples

Perform BLAST search

Perform a BLAST search on a protein sequence and save the results to an XML file.

Get a sequence from the Protein Data Bank and create a MATLAB structure.

```
S = getpdb('1CIV');
```

Use the structure as input for the BLAST search with a significance threshold of `1e-10`. The first output is the request ID, and the second output is the estimated time (in minutes) until the search is completed.

```
[RID1,ROTE] = blastncbi(S,'blastp','expect',1e-10);
```

Get the search results from the report. You can save the XML-formatted report to a file for an offline access. Use ROTE as the wait time to retrieve the results.

```
report1 = getblast(RID1,'WaitTime',ROTE,'ToFile','1CIV_report.xml')
```

```
Blast results are not available yet. Please wait ...

report1 = 

  struct with fields:

                RID: 'R49TJMCF014'
          Algorithm: 'BLASTP 2.6.1+'
           Database: 'nr'
            QueryID: 'Query_224139'
    QueryDefinition: 'unnamed protein product'
               Hits: [1×100 struct]
         Parameters: [1×1 struct]
         Statistics: [1×1 struct]
```

Use `blastread` to read BLAST data from the XML-formatted BLAST report file.

```
blastdata = blastread('1CIV_report.xml')
```

```
blastdata = 

  struct with fields:

                RID: ''
          Algorithm: 'BLASTP 2.6.1+'
           Database: 'nr'
            QueryID: 'Query_224139'
    QueryDefinition: 'unnamed protein product'
               Hits: [1×100 struct]
         Parameters: [1×1 struct]
         Statistics: [1×1 struct]
```

Alternatively, run the BLAST search with an NCBI accession number.

```
RID2 = blastncbi('AAA59174','blastp','expect',1e-10)
```

```
RID2 =

    'R49WAPMH014'
```

Get the search results from the report.

```
report2 = getblast(RID2)
```

```
Blast results are not available yet. Please wait ...

report2 = 

  struct with fields:

                RID: 'R49WAPMH014'
          Algorithm: 'BLASTP 2.6.1+'
           Database: 'nr'
            QueryID: 'AAA59174.1'
    QueryDefinition: 'insulin receptor precursor [Homo sapiens]'
               Hits: [1×100 struct]
         Parameters: [1×1 struct]
         Statistics: [1×1 struct]
```

Input Arguments

RID — Request ID for NCBI BLAST search
character vector | string

Request ID for retrieving results from a specific NCBI BLAST search, specified as a character vector or string.

Example: `'GTF033EZ015'`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `'ToFile','report.xml'` saves the results to a file named `report.xml`.

`ToFile` — Name of file to save report data to
character vector | string

Name of the file to save the report data to, specified as the comma-separated pair consisting of `'ToFile'` and a character vector or string. The file is XML-formatted by default.

Example: `'ToFile','Report.xml'`

`WaitTime` — Time to wait for report
0 (default) | nonnegative integer

Time (in minutes) to wait for the report from NCBI to be ready, specified as the comma-separated pair consisting of `'WaitTime'` and a nonnegative integer. If the report is still not ready after the specified time, an error is generated.

The default value is 0, that is, there is no delay in retrieving the report.

Tip Use the `RTOE`, request time of execution, returned by the `blastncbi` function as the wait time here.

Example: `'WaitTime',2`

`TimeOut` — Connection timeout
5 (default) | positive scalar

Connection timeout (in seconds) for each request, specified as a positive scalar. For details, see here.

Example: `'TimeOut',10`

Output Arguments

`blastdata` — BLAST report data
structure

BLAST report data, returned as a structure that contains the following fields:

Field	Description
`RID`	Request ID for retrieving results from a specific NCBI BLAST search
`Algorithm`	NCBI algorithm used to perform the BLAST search
`Database`	All databases searched
`QueryID`	Identifier of the query sequence
`QueryDefinition`	Definition of the query sequence
`Hits`	Structure containing information on the hit sequences, such as IDs, accession numbers, lengths, and HSPs (high-scoring segment pairs)
`Parameters`	Structure containing information on the input parameters used to perform the search
`Statistics`	Summary of statistical details about the performed search, such as lambda, kappa, and entropy values

More About

Hits

This table lists each field of `blastdata.Hits`.

Field	Description
`ID`	ID of the subject sequence that matched the query sequence
`Definition`	Description of the subject sequence
`Accession`	Accession of the subject sequence
`Length`	Length of the subject sequence
`Hsps`	Structure containing Information on the high-scoring segment pairs (HSPs)

Hits.Hsps

This table summarizes the fields of `Hits.Hsps`.

Field	Description
`Score`	Pairwise alignment score for a high-scoring segment pair between the query sequence and a subject sequence.
`BitScore`	Bit score for a high-scoring segment pair.
`Expect`	Expectation value for a high-scoring segment pair.
`Identities`	Number of identical or similar residues for a high-scoring segment pair between the query sequence and a subject sequence.

Field	Description
`Positives`	Number of identical or similar residues for a high-scoring sequence pair between the query sequence and a subject amino acid sequence. This field applies only to translated nucleotide or amino acid query sequences and databases.
`Gaps`	Nonaligned residues for a high-scoring segment pair.
`AlignmentLength`	Length of the alignment for a high-scoring segment pair.
`QueryIndices`	Indices of the query sequence residue positions for a high-scoring segment pair.
`SubjectIndices`	Indices of the subject sequence residue positions for a high-scoring segment pair.
`Frame`	Reading frame of the translated nucleotide sequence for a high-scoring segment pair.
`Alignment`	3-by-*N* character array showing the alignment for a high-scoring sequence pair between the query sequence and a subject sequence. The first row is the query sequence, the second row is the alignment, and the third row is the subject sequence.

Version History

Introduced before R2006a

R2017b: `'Alignments'` option has been removed

Errors starting in R2017b

The `'Alignments'` name-value pair has been removed. The number of hits returned in the output is controlled by the number of hits in the input BLAST report.

R2017b: `'Descriptions'` option has been removed

Errors starting in R2017b

The `'Descriptions'` name-value pair has been removed. The number of hits returned in the output is controlled by the number of hits in the input BLAST report.

R2017b: `'FileFormat'` option has been removed

Errors starting in R2017b

The `'FileFormat'` name-value pair has been removed. The file is XML-formatted automatically.

See Also

`blastncbi` | `blastread`

getBowtie2Command

Translate object properties to Bowtie 2 options

Syntax

```
S = getBowtie2Command(object)
S = getBowtie2Command(object,'IncludeAll',TF)
```

Description

`S = getBowtie2Command(object)` returns a character vector `S`, representing the Bowtie 2 option syntax that corresponds to the modified object properties. By default, the function translates only the modified properties.

`getBowtie2Command` requires the Bowtie 2 Support Package for Bioinformatics Toolbox. If this support package is not installed, then the function provides a download link. For details, see “Bioinformatics Toolbox Software Support Packages”.

`S = getBowtie2Command(object,'IncludeAll',TF)` specifies whether to translate all the object properties (`true`) or only the modified properties (`false`).

Examples

Retrieve Bowtie 2 Options

Create a `Bowtie2AlignOptions` object.

```
alignOpt = Bowtie2AlignOptions;
```

Modify the object properties. For example, specify to trim four residues from the `3'` and `5'` ends before aligning.

```
alignOpt.Trim3 = 4;
```

```
alignOpt.Trim5 = 4;
```

Retrieve the equivalent Bowtie 2 option syntax for the modified properties.

```
getbowtie2command(alignOpt)
```

```
ans =

    '-3 4 -5 4'
```

Input Arguments

`object` — Bowtie 2 options object
`Bowtie2AlignOptions` object | `Bowtie2InspectOptions` object | `Bowtie2BuildOptions` object

Bowtie 2 options object, specified as a `Bowtie2AlignOptions`, `Bowtie2InspectOptions`, or `Bowtie2BuildOptions` object.

Example: `alignOpt`

TF — Flag to translate all object properties
`false` (default) | `true`

Flag to translate all object properties, specified as `true` or `false`. If `true`, the function translates all the object properties. If `false`, the function translates only the modified properties.

Example: `true`

Data Types: `logical`

Output Arguments

S — Bowtie 2 option syntax
character vector

Bowtie 2 option syntax [1], returned as a character vector. The syntax is prefixed by one or two dashes.

Version History

Introduced in R2018a

References

[1] Langmead, B., and S. Salzberg. "Fast gapped-read alignment with Bowtie 2." *Nature Methods*. 9, 2012, 357–359.

See Also

`bowtie2` | `bowtie2inspect` | `bowtie2build` | `Bowtie2AlignOptions` | `Bowtie2BuildOptions` | `Bowtie2InspectOptions`

External Websites
Bowtie 2 manual

getBowtie2Table

Retrieve table with object properties and equivalent Bowtie 2 options

Syntax

`tbl = getBowtie2Table(object)`

Description

`tbl = getBowtie2Table(object)` returns a table with all the object properties and corresponding Bowtie 2 options [1].

`getBowtie2Table` requires the Bowtie 2 Support Package for Bioinformatics Toolbox. If this support package is not installed, then the function provides a download link. For details, see "Bioinformatics Toolbox Software Support Packages".

Examples

Retrieve Corresponding Bowtie 2 Options for All Object Properties

Create a `Bowtie2AlignOptions` object.

```
alignOpt = Bowtie2AlignOptions;
```

Retrieve the equivalent Bowtie 2 options for all the object properties.

```
getbowtie2table(alignOpt)
```

```
ans =

  47×2 table

         PropertyName              Bowtie2OptionName
    ______________________      ______________________

    'AllowDovetail'             '--dovetail'
    'AmbiguousPenalty'          '--np'
    'Encoding_Phred33'          '--phred33'
    'Encoding_Phred64'          '--phred64'
    'Encoding_Solexa'           '--solexa-quals'
    'ExcludeContain'            '--no-contain'
    'ExcludeDiscordant'         '--no-discordant'
    'ExcludeMixed'              '--no-mixed'
    'ExcludeOverlap'            '--no-overlap'
    'ExcludeUnaligned'          '--no-unal'
    'IgnoreQuality'             '--ignore-quals'
    'MatchBonus'                '--ma'
    'MaxAmbiguousFunction'      '--n-ceil'
    'MemoryMappedIndex'         '--mm'
    'MinScoreFunction'          '--score-min'
    'MismatchPenalty'           '--mp'
```

```
'Mode_EndToEnd'           '--end-to-end'
'Mode_Local'              '--local'
'NoGapPositions'          '--gbar'
'Nondeterministic'        '--non-deterministic'
'NumAlignments'           '-k'
'NumAlignments_All'       '-a'
'NumAlignments_Best'      ''
'NumReseedings'           '-R'
'NumSeedExtensions'       '-D'
'NumSeedMismatches'       '-N'
'NumThreads'              '--threads'
'NumThreads'              '-p'
'Offrate'                 '--offrate'
'Offrate'                 '-o'
'PadPositions'            '--dpad'
'ReadGapCosts'            '--rdg'
'ReadGroup'               '--rg'
'ReadGroupID'             '--rg-id'
'RefGapCosts'             '--rfg'
'Reorder'                 '--reorder'
'Seed'                    '--seed'
'SeedIntervalFunction'    '-i'
'SeedLength'              '-L'
'Skip'                    '--skip'
'Skip'                    '-s'
'Trim3'                   '--trim3'
'Trim3'                   '-3'
'Trim5'                   '--trim5'
'Trim5'                   '-5'
'UpTo'                    '--upto'
'UpTo'                    '-u'
```

Input Arguments

`object` — Bowtie 2 options object

`Bowtie2AlignOptions` object | `Bowtie2InspectOptions` object | `Bowtie2BuildOptions` object

Bowtie 2 options object, specified as a `Bowtie2AlignOptions`, `Bowtie2InspectOptions`, or `Bowtie2BuildOptions` object.

Example: `alignOpt`

Output Arguments

`tbl` — Object properties and corresponding Bowtie2 options

table

Object properties and the corresponding Bowtie2 options [1], returned as a table.

Version History

Introduced in R2018a

References

[1] Langmead, B., and S. Salzberg. "Fast gapped-read alignment with Bowtie 2." *Nature Methods*. 9, 2012, 357–359.

See Also

`bowtie2` | `bowtie2inspect` | `bowtie2build` | `Bowtie2AlignOptions` | `Bowtie2BuildOptions` | `Bowtie2InspectOptions`

External Websites

Bowtie 2 manual

getbyname (phytree)

Branches and leaves from phytree object

Syntax

```
S = getbyname(Tree, Expression)
S = getbyname(Tree, Key)
S = getbyname(Tree, Key, 'Exact', ExactValue)
```

Arguments

Tree	phytree object created by `phytree` function (object constructor) or `phytreeread` function.
Expression	Regular expression or cell array of regular expressions to search for in *Tree*.
Key	Character vector or cell array of character vectors to search for in *Tree*.
ExactValue	Controls whether the full exact node name must match the character vector(s), ignoring case. Choices are `true` or `false` (default). When `true`, *S* is a numeric column vector indicating which node names match a query exactly, in full.

Description

S = `getbyname(`*Tree*, *Expression*`)` searches the nodes names in *Tree*, a phytree object, for the regular expression(s) specified by *Expression*. It returns *S*, a logical matrix of size `NumNodes`-by-`M`, where `M` is either `1` or the length of *Expression*. Each row in *S* corresponds to a node, and each column corresponds to a query in *Expression*. The logical matrix *S* indicates the node names that match *Expression*, ignoring case.

S = `getbyname(`*Tree*, *Key*`)` searches the nodes names in *Tree*, a phytree object, for the character vector(s) specified by *Key*. It returns *S*, a logical matrix of size `NumNodes`-by-`M`, where `M` is either `1` or the length of *Key*. Each row in *S* corresponds to a node, and each column corresponds to a query in *Key*. The logical matrix *S* indicates the node names that match *Key*, ignoring case.

S = `getbyname(`*Tree*, *Key*, `'Exact'`, *ExactValue*`)` specifies whether the full exact node name must match the character vector(s), ignoring case. Choices are `true` or `false` (default). When `true`, *S* is a numeric column vector indicating which node names match a query exactly, in full.

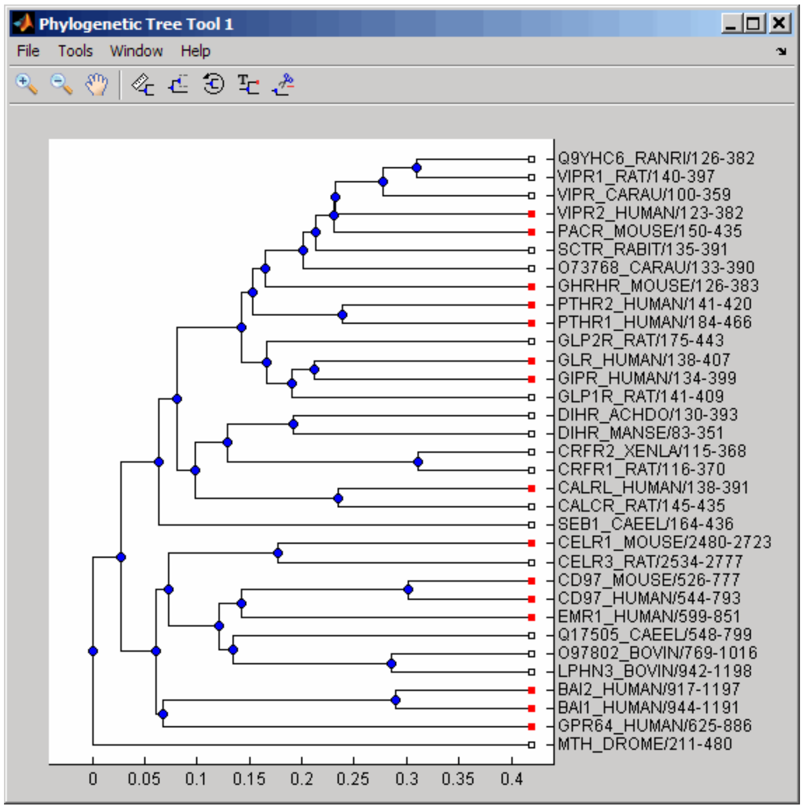
Examples

1 Read a phylogenetic tree file created from a protein family into a phytree object.

```
tr = phytreeread('pf00002.tree');
```

2 Determine all the mouse and human proteins by searching for nodes that include the character vectors `'mouse'` and `'human'` in their names.

```
sel = getbyname(tr,{'mouse','human'});
view(tr,any(sel,2));
```



Version History

Introduced before R2006a

See Also

`phytree` | `phytreeread` | `get` | `prune` | `select`

Topics

phytree object on page 1-1449

getcanonical (phytree)

Calculate canonical form of phylogenetic tree

Syntax

```
Pointers = getcanonical(Tree)
[Pointers, Distances, Names] = getcanonical(Tree)
```

Arguments

Tree	phytree object created by `phytree` function (object constructor).

Description

`Pointers = getcanonical(Tree)` returns the pointers for the canonical form of a phylogenetic tree (*Tree*). In a canonical tree the leaves are ordered alphabetically and the branches are ordered first by their width and then alphabetically by their first element. A canonical tree is isomorphic to all the trees with the same skeleton independently of the order of their leaves and branches.

`[Pointers, Distances, Names] = getcanonical(Tree)` returns, in addition to the pointers described above, the reordered distances (*Distances*) and node names (*Names*).

Examples

1 Create two phylogenetic trees with the same skeleton but slightly different distances.

```
b = [1 2; 3 4; 5 6; 7 8;9 10];
tr_1 = phytree(b,[.1 .2 .3 .3 .4 ]');
tr_2 = phytree(b,[.2 .1 .2 .3 .4 ]');
```

2 Plot the trees.

```
 plot(tr_1)
 plot(tr_2)
```

3 Check whether the trees have an isomorphic construction.

```
isequal(getcanonical(tr_1),getcanonical(tr_2))

ans =
     1
```

Version History

Introduced before R2006a

See Also

`phytree` | `phytreeread` | `getbyname` | `select` | `subtree`

Topics
phytree object on page 1-1449

getCommand

Translate object properties to original options syntax

Syntax

```
S = getCommand(optionsObject)
```

Description

`S = getCommand(optionsObject)` returns a string `S` representing the modified properties in `optionsObject` translated into the original syntax (prefixed by one or two dashes). By default, the function translates only the modified properties.

Note If you set `IncludeAll` to `true`, the software translates all available properties, with default values for unspecified properties. The only exception is that when the default value of a property is `NaN`, `Inf`, `[]`, `''`, or `""`, then the software does not translate the corresponding property.

Examples

Retrieve Options in Original Syntax

Create a `CufflinksOptions` object.

Note `getCommand` also works on other options objects. For a complete list of objects, see “optionsObject” on page 1-0 .

```
opt = CufflinksOptions;
```

Modify the object properties. For this example, specify the minimum average coverage for 3' end trimming and change the seed for the Cufflinks random number generator.

```
opt.TrimCoverageThreshold = 5;
```

```
opt.Seed = 1;
```

Retrieve the options translated into the original syntax.

```
s = getCommand(opt)
```

```
ans = 

    "--seed -2.3 --trim-3-avgcov-thresh 5"
```

Input Arguments

`optionsObject` — Options
`BWAIndexOptions` | `BWAMEMOptions` | `CufflinksOptions` | `CuffCompareOptions` | `CuffMergeOptions` | `CuffQuantOptions` | `CuffDiffOptions` | `CuffNormOptions` | `CuffGFFReadOptions`

Options, specified as a `BWAIndexOptions`, `BWAMEMOptions`, `CufflinksOptions`, `CuffCompareOptions`, `CuffMergeOptions`, `CuffQuantOptions`, `CuffDiffOptions`, `CuffNormOptions`, or `CuffGFFReadOptions` object.

Output Arguments

`S` — Options in original syntax
string

Options in the original syntax, returned as a string.

Version History

Introduced in R2019a

References

[1] Trapnell, C., B. Williams, G. Pertea, A. Mortazavi, G. Kwan, J. van Baren, S. Salzberg, B. Wold, and L. Pachter. 2010. Transcript assembly and quantification by RNA-Seq reveals unannotated transcripts and isoform switching during cell differentiation. *Nature Biotechnology*. 28:511–515.

[2] Li, Heng, and Richard Durbin. “Fast and Accurate Short Read Alignment with Burrows-Wheeler Transform.” *Bioinformatics* 25, no. 14 (July 15, 2009): 1754–60. https://doi.org/10.1093/bioinformatics/btp324.

[3] Li, Heng, and Richard Durbin. “Fast and Accurate Long-Read Alignment with Burrows–Wheeler Transform.” *Bioinformatics* 26, no. 5 (March 1, 2010): 589–95. https://doi.org/10.1093/bioinformatics/btp698.

See Also

`BWAIndexOptions` | `CuffCompareOptions` | `CuffDiffOptions` | `CuffGFFReadOptions` | `CufflinksOptions` | `CuffMergeOptions` | `CuffNormOptions` | `CuffQuantOptions`

getData

Create structure containing subset of data from `GTFAnnotation` or `GFFAnnotation` object

Syntax

```
AnnotStruct = getData(AnnotObj)
AnnotStruct = getData(AnnotObj,StartPos,EndPos)
AnnotStruct = getData(AnnotObj,Subset)
AnnotStruct = getData( ___ ,Name,Value)
```

Description

`AnnotStruct = getData(AnnotObj)` returns `AnnotStruct`, an array of structures containing data from all elements in `AnnotObj`. The fields in the return structures are the same as the elements in the `FieldNames` property of `AnnotObj`.

`AnnotStruct = getData(AnnotObj,StartPos,EndPos)` returns `AnnotStruct`, an array of structures containing data from a subset of the elements in `AnnotObj` that falls within each reference sequence range specified by `StartPos` and `EndPos`.

`AnnotStruct = getData(AnnotObj,Subset)` returns `AnnotStruct`, an array of structures containing subset of data from `AnnotObj` specified by `Subset`, a vector of integers.

`AnnotStruct = getData( ___ ,Name,Value)` returns `AnnotStruct`, an array of structures, using any of the input arguments in the previous syntaxes and additional options specified by one or more `Name,Value` pair arguments.

Examples

Retrieve Subsets of Data from a `GTFAnnotation` Object

Construct a `GTFAnnotation` object using a GTF-formatted file that is provided with Bioinformatics Toolbox™.

```
GTFAnnotObj = GTFAnnotation('hum37_2_1M.gtf');
```

Extract the annotation data for positions 668,000 through 680,000 from the reference sequence.

```
AnnotStruct1 = getData(GTFAnnotObj,668000,680000)
```

```
AnnotStruct1=18×1 struct array with fields:
    Reference
    Start
    Stop
    Feature
    Gene
    Transcript
    Source
    Score
    Strand
    Frame
    Attributes
```

Extract the first five annotations from the object.

```
AnnotStruct2 = getData(GTFAnnotObj,1:5)
```

```
AnnotStruct2=5×1 struct array with fields:
    Reference
    Start
    Stop
    Feature
    Gene
    Transcript
    Source
    Score
    Strand
    Frame
    Attributes
```

Retrieve Subsets of Data from a `GFFAnnotation` Object

Construct a `GFFAnnotation` object using a GFF-formatted file that is provided with Bioinformatics Toolbox™.

```
GFFAnnotObj = GFFAnnotation('tair8_1.gff');
```

Extract annotations for positions 10,000 through 20,000 from the reference sequence.

```
AnnotStruct1 = getData(GFFAnnotObj,10000,20000)
```

```
AnnotStruct1=9×1 struct array with fields:
    Reference
    Start
    Stop
    Feature
    Source
    Score
    Strand
    Frame
    Attributes
```

Extract the first five annotations from the object.

```
AnnotStruct2 = getData(GFFAnnotObj,1:5)
```

```
AnnotStruct2=5×1 struct array with fields:
    Reference
    Start
    Stop
    Feature
    Source
    Score
    Strand
    Frame
    Attributes
```

Input Arguments

`AnnotObj` — Feature annotations
`GTFAnnotation` object | `GFFAnnotation` object

Feature annotations, specified as a `GTFAnnotation` or `GFFAnnotation` object.

`StartPos` — Start of a range in each reference sequence in `AnnotObj`
nonnegative integer

Start of a range in each reference sequence in `AnnotObj`, specified as a nonnegative integer less than or equal to `EndPos`.

Data Types: `double`

`EndPos` — End of a range in each reference sequence in `AnnotObj`
nonnegative integer

End of a range in each reference sequence in `AnnotObj`, specified as a nonnegative integer greater than or equal to `StartPos`.

Data Types: `double`

`Subset` — Subset of data from `AnnotObj` to retrieve
vector of positive integers

Subset of data from `AnnotObj` to retrieve, specified as a vector of positive integers. Each integer must be less than or equal to the number of entries in the object.

Data Types: `double`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `AnnotStruct = getData(AnnotObj,"Reference","Chr")`

`Reference` — One or more reference sequences in `AnnotObj`
character vector | string | string vector | cell array of character vectors

One or more reference sequences in `AnnotObj`, specified as a character vector, string, string vector, or cell array of character vectors. Only annotations whose reference field matches one of the character vectors or strings are included in `AnnotStruct`.

Data Types: `char` | `string` | `cell`

`Feature` — One or more features in `AnnotObj`
character vector | string | string vector | cell array of character vectors

One or more features in AnnotObj, specified as a character vector, string, string vector, or cell array of character vectors. Only annotations whose feature field matches one of the character vectors or strings are included in AnnotStruct.

Data Types: char | string | cell

Gene — One or more genes in AnnotObj of type GTFAnnotation
character vector | string | string vector | cell array of character vectors

One or more genes in AnnotObj of type GTFAnnotation, specified as a character vector, string, string vector, or cell array of character vectors. Only annotations whose gene field matches one of the character vectors or strings are included in AnnotStruct.

Data Types: char | string | cell

Transcript — One or more transcripts in AnnotObj of type GTFAnnotation
character vector | string | string vector | cell array of character vectors

One or more transcripts in AnnotObj of type GTFAnnotation, specified as a character vector, string, string vector, or cell array of character vectors. Only annotations whose transcript field matches one of the character vectors or strings are included in AnnotStruct.

Data Types: char | string | cell

Overlap — Minimum number of base positions that annotation must overlap in the range
1 (default) | positive integer | "full" | "start"

Minimum number of base positions that annotation must overlap in the range, to be included in AnnotStruct, specified as a positive integer, "full" or "start". Use "full" when an annotation must be fully contained in the range to be included. Use "start" when an annotation's start position must lie within the range to be included.

Data Types: double | char | string

Output Arguments

AnnotStruct — Data from elements in AnnotObj
structure array

Data from elements in AnnotObj, returned as a structure array with these fields:

- Reference
- Start
- Stop
- Feature
- Gene (for AnnotObj of type GTFAnnotation)
- Transcript (for AnnotObj of type GTFAnnotation)
- Source
- Score
- Strand
- Frame

- `Attributes`

The fields are the same as the elements in the `FieldNames` property of `AnnotObj`. See GTF2.2: A Gene Annotation Format.

Tips

Using `getData` creates a structure, which provides better access to the annotation data than an object.

- You can access all field values in a structure.
- You can extract, assign, and delete field values.
- You can use linear indexing to access field values of specific annotations. For example, you can access the start value of only the fifth annotation.

Version History

Introduced in R2013a

See Also

`getExons` | `getFeatureNames` | `getGeneNames` | `getGenes` | `getIndex` | `getRange` | `getReferenceNames` | `getSegments` | `getSubset` | `getTranscriptNames` | `getTranscripts` | `GTFAnnotation` | `GFFAnnotation`

Topics

"Store and Manage Feature Annotations in Objects"

External Websites

GTF2.2: A Gene Annotation Format
GFF (General Feature Format) specifications

getdescendants (biograph)

(Removed) Find descendants of a node in biograph object

Note The function has been removed. Use `successors` instead. Note that the function does not let you specify `NumGenerations`. You can use `distances` to find the distances from all nodes to the target node after setting the edge weight to 1, and filter the nodes by distance to the appropriate depth.

Syntax

```
Nodes = getdescendants(BiographNode)
Nodes = getdescendants(BiographNode,NumGenerations)
```

Arguments

`BiographNode`	Node in a biograph object.
`NumGenerations`	Number of generations. Enter a positive integer.

Description

`Nodes` `= getdescendants(`*`BiographNode`*`)` finds a given node (*`BiographNode`*) all of its direct descendants.

`Nodes` `= getdescendants(`*`BiographNode,NumGenerations`*`)` finds the node (*`BiographNode`*) and all of its direct descendants up to a specified number of generations (*`NumGenerations`*). If the *`NumGenerations`* is `0`, the function returns the node itself.

Version History

Introduced before R2006a

R2022b: Removed

Errors starting in R2022b

`getdescendants (biograph)` will be removed in a future release. Use `successors` instead. Note that the function does not let you specify `NumGenerations`. You can use `distances` to find the distances from all nodes to the target node after setting the edge weight to 1, and filter the nodes by distance to the appropriate depth.

R2022a: Warns

Warns starting in R2022a

The function issues a warning that it will be removed in a future release.

R2021b: To be removed

Not recommended starting in R2021b

The function runs without warning, but it will be removed in a future release.

See Also

`graph` | `digraph` | `distances` | `successors`

getdescendents

Find terms that are descendents of specified Gene Ontology (GO) term

Syntax

```
DescendentIDs = getdescendents(GeneontObj,ID)
[DescendentIDs,Counts] = getdescendents(GeneontObj,ID)
___ = getdescendents(GeneontObj,ID,Name,Value)
```

Description

`DescendentIDs = getdescendents(GeneontObj,ID)` searches `GeneontObj`, a `geneont` object, for GO terms that are descendents of the GO term(s) specified by `ID`, which is a GO term identifier or vector of identifiers. The result `DescendentIDs` is a vector of GO term identifiers including `ID`.

`[DescendentIDs,Counts] = getdescendents(GeneontObj,ID)` also returns the number of times each descendent is found. `Counts` is a column vector with the same number of elements as terms in `GeneontObj`.

Tip The `Counts` return value is useful when you tally counts in gene enrichment studies.

`___ = getdescendents(GeneontObj,ID,Name,Value)`, for any output arguments, specifies additional options using one or more name-value arguments. For example, you can restrict the search to have up to two levels up in the gene ontology by specifying `DescendentIDs = getdescendents(GeneontObj,ID,Height=2)`.

Examples

Get Descendents of geneont Object

Download the current version of the Gene Ontology database from the Web into a `geneont` object.

```
GO = geneont('Live',true)
```

```
Gene Ontology object with 47266 Terms.
```

Retrieve the descendants of the "aldo-keto reductase activity" GO term with a GO identifier of 4033.

```
descendants = getdescendants(GO,4033)
```

```
descendants = 9×1

        4032
        4033
        8106
       32018
       32866
       32867
       50236
       52650
       52675
```

Create a subordinate Gene Ontology.

```
subontology = GO(descendants)
```

```
Gene Ontology object with 9 Terms.
```

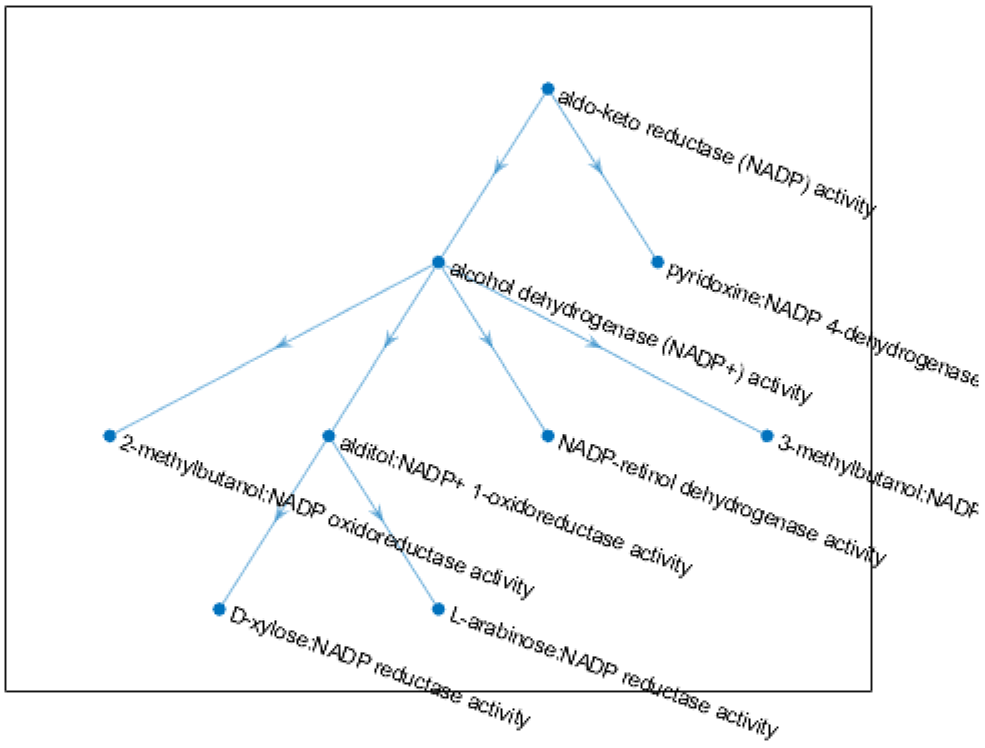
Create and display a report of the subordinate Gene Ontology terms, that includes the GO identifier and name.

```
rpt = [num2goid(cell2mat(get(subontology.terms,'id')))...
       get(subontology.terms,'name')]';
fprintf('%s --> %s \n',rpt{:})
```

```
GO:0004032 --> alditol:NADP+ 1-oxidoreductase activity
GO:0004033 --> aldo-keto reductase (NADP) activity
GO:0008106 --> alcohol dehydrogenase (NADP+) activity
GO:0032018 --> 2-methylbutanol:NADP oxidoreductase activity
GO:0032866 --> D-xylose:NADP reductase activity
GO:0032867 --> L-arabinose:NADP reductase activity
GO:0050236 --> pyridoxine:NADP 4-dehydrogenase activity
GO:0052650 --> NADP-retinol dehydrogenase activity
GO:0052675 --> 3-methylbutanol:NADP oxidoreductase activity
```

View relationships of the subordinate Gene Ontology by using the `getmatrix` method to create a connection matrix to pass to the `digraph` function.

```
cm = getmatrix(subontology);
BG = digraph(cm,get(subontology.terms,"name"));
plot(BG)
```


aldo-keto reductase (NADP) activity
alcohol dehydrogenase (NADP+) activity
pyridoxine:NADP 4-dehydrogenase
2-methylbutanol:NADP oxidoreductase activity
alditol:NADP+ 1-oxidoreductase activity
NADP-retinol dehydrogenase activity
3-methylbutanol:NADP
D-xylose:NADP reductase activity
L-arabinose:NADP reductase activity

Input Arguments

`GeneontObj` — Gene ontology object
output of `geneont`

Gene ontology object, specified as the output of the `geneont` command.

Example: `geneont('Live',true)`

`ID` — Gene ontology ID numbers
nonnegative integer vector

Gene ontology ID numbers, specified as a nonnegative integer vector.

Example: `5`

Data Types: `single` | `double`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Names are case-insensitive.

Example: `DescendentIDs = getdescendents(GeneontObj,5,Depth=2)`

`Depth` — Number of search levels in gene ontology
`Inf` (default) | positive integer

Number of search levels down in gene ontology, specified as a positive integer.

Example: `4`

Data Types: `single` | `double`

`Relationtype` — Relations to match
`'both'` (default) | `'is_a'` | `'part_of'`

Relations to match, specified as one of the following:

- `'is_a'`
- `'part_of'`
- `'both'`

Example: `'is_a'`

Data Types: `char` | `string`

`Exclude` — Indication to exclude `ID` from `DescendentIDs`
`false` (default) | `true`

Indication to exclude `ID` from `DescendentIDs`, specified as `false` or `true`. This argument can fail to apply when `getdescendents` reaches the term while searching the ontology.

Example: `true`

Data Types: `logical`

Output Arguments

`DescendentIDs` — ID numbers of descendents of `GeneontObj`
nonnegative integer vector

ID numbers of descendents of `GeneontObj`, returned as a nonnegative integer vector. By default, `DescendentIDs` includes `ID`, unless you set the `Exclude` argument to `true`.

`Counts` — Number of times each relative is found
column vector of integers

Number of times each relative is found, returned as a column vector of integers. `Counts` has the same number of elements as terms in `GeneontObj`.

Version History

Introduced before R2006a

See Also

`geneont` | `goannotread` | `num2goid` | `term`

getDictionary

Class: `BioIndexedFile`

Retrieve reference sequence names from SAM-formatted source file associated with BioIndexedFile object

Syntax

```
Dict = getDictionary(BioIFobj)
```

Description

`Dict` = `getDictionary(`*`BioIFobj`*`)` returns *`Dict`*, a cell array of unique character vectors specifying the names of the reference sequences in the SAM-formatted source file associated with *`BioIFobj`*, a BioIndexedFile object.

Input Arguments

`BioIFobj`

Object of the `BioIndexedFile` class.

Default:

Output Arguments

`Dict`

Cell array of unique character vectors specifying the reference sequence names in the SAM-formatted source file associated with *`BioIFobj`*, a BioIndexedFile object.

See Also

`BioIndexedFile` | `BioMap` | `getSubset`

Topics

“Work with Next-Generation Sequencing Data”
“Manage Sequence Read Data in Objects”

getedgesbynodeid (biograph)

(Removed) Get handles to edges in biograph object

Note The function has been removed. Use `findedge` instead.

Syntax

`*Edges* = getedgesbynodeid(*BGobj*,*SourceIDs*,*SinkIDs*)`

Arguments

`BGobj`	Biograph object.
`SourceIDs`, *`SinkIDs`*	Enter a character vector, cell array of character vectors, or an empty cell array (gets all edges).

Description

`Edges` `= getedgesbynodeid(`*`BGobj`*`,`*`SourceIDs`*`,`*`SinkIDs`*`)` gets the handles to the edges that connect the specified source nodes (*`SourceIDs`*) to the specified sink nodes (*`SinkIDs`*) in a biograph object.

Version History

Introduced before R2006a

R2022b: Removed
Errors starting in R2022b

The function has been removed. Use `findedge` instead.

R2022a: Warns
Warns starting in R2022a

The function issues a warning that it will be removed in a future release.

R2021b: To be removed
Not recommended starting in R2021b

The function runs without warning, but it will be removed in a future release.

See Also

`findedge` | `graph` | `digraph`

getembl

Retrieve sequence information from EMBL database

Syntax

```
EMBLData = getembl(AccessionNumber)

EMBLData = getembl(..., 'ToFile', ToFileValue, ...)
EMBLSeq = getembl(..., 'SequenceOnly', SequenceOnlyValue, ...)
EMBLSeq = getembl(..., 'TimeOut', TimeOutValue, ...)
```

Input Arguments

AccessionNumber	Unique identifier for a sequence record. Enter a unique combination of letters and numbers.
ToFileValue	Character vector specifying a file name or a path and file name to which to save the data. If you specify only a file name, the file is stored in the current folder.
SequenceOnlyValue	Controls the retrieving of only the sequence without the metadata. Choices are `true` or `false` (default).
TimeOutValue	Connection timeout in seconds, specified as a positive scalar. The default value is 5. For details, see here.

Output Arguments

EMBLData	MATLAB structure with fields corresponding to EMBL data.
EMBLSeq	MATLAB character vector representing the sequence.

Description

`getembl` retrieves information from the European Molecular Biology Laboratory (EMBL) database for nucleotide sequences. This database is maintained by the European Bioinformatics Institute (EBI). For more details about the EMBL database, see

`https://ena-docs.readthedocs.io/en/latest/retrieval/general-guide.html`

EMBLData `= getembl(`*AccessionNumber*`)` searches for the accession number in the EMBL database (`https://www.ebi.ac.uk/`) and returns *EMBLData*, a MATLAB structure with fields corresponding to the EMBL two-character line type code. Each line type code is stored as a separate element in the structure.

EMBLData contains the following fields.

Field
`Identification`

Field
`Accession`
`SequenceVersion`
`DateCreated`
`DateUpdated`
`Description`
`Keyword`
`OrganismSpecies`
`OrganismClassification`
`Organelle`
`Reference`
`DatabaseCrossReference`
`Comments`
`Assembly`
`Feature`
`BaseCount`
`Sequence`

`*EMBLData* = getembl(..., '*PropertyName*', *PropertyValue*, ...)` calls `getembl` with optional properties that use property name/property value pairs. You can specify one or more properties in any order. Each `*PropertyName*` must be enclosed in single quotation marks and is case insensitive. These property name/property value pairs are as follows:

`*EMBLData* = getembl(..., 'ToFile', *ToFileValue*, ...)` saves the information to an EMBL-formatted file. `*ToFileValue*` is a character vector specifying a file name or a path and file name to which to save the data. If you specify only a file name, the file is stored in the current folder.

Tip Read an EMBL-formatted file back into the MATLAB software using the `emblread` function.

`*EMBLSeq* = getembl(..., 'SequenceOnly', *SequenceOnlyValue*, ...)` controls the retrieving of only the sequence without the metadata. Choices are `true` or `false` (default).

`*EMBLSeq* = getembl(..., 'TimeOut', *TimeOutValue*, ...)` sets the connection timeout (in seconds) to retrieve data from EMBL database.

Examples

Retrieve data for the rat liver apolipoprotein A-I.

```
emblout = getembl('X00558')
```

Retrieve data for the rat liver apolipoprotein A-I and save it to the file `rat_protein`. If you specify a file name without a path, the file is stored in the current folder.

```
emblout = getembl('X00558','ToFile','c:\project\rat_protein.txt')
```

Retrieve only the sequence for the rat liver apolipoprotein A-I.

```
Seq = getembl('X00558','SequenceOnly',true)
```

Version History

Introduced before R2006a

See Also

`emblread` | `getgenbank` | `getgenpept` | `getpdb` | `seqviewer`

getEntryByIndex

Class: `BioIndexedFile`

Retrieve entries from source file associated with BioIndexedFile object using numeric index

Syntax

`Entries = getEntryByIndex(BioIFobj, Indices)`

Description

`Entries = getEntryByIndex(BioIFobj, Indices)` extracts entries from the source file associated with *`BioIFobj`*, a BioIndexedFile object. It extracts and concatenates the entries specified by *`Indices`*, a numeric vector of positive integers. It returns *`Entries`*, a character vector of concatenated entries. The value of each element in *`Indices`* must be less than or equal to the number of entries in the source file. A one-to-one relationship exists between the number and order of elements in *`Indices`* and the output *`Entries`*, even if *`Indices`* has repeated entries.

Input Arguments

`BioIFobj`

Object of the `BioIndexedFile` class.

Default:

`Indices`

Numeric vector of positive integers. The value of each element must be less than or equal to the number of entries in the source file associated with *`BioIFobj`*, the BioIndexedFile object.

Default:

Output Arguments

`Entries`

Character vector of concatenated entries extracted from the source file associated with *`BioIFobj`*, the BioIndexedFile object.

Examples

Construct a BioIndexedFile object to access a table containing cross-references between gene names and gene ontology (GO) terms:

```
% Create variable containing full absolute path of source file
sourcefile = which('yeastgenes.sgd');
% Create a BioIndexedFile object from the source file. Indicate
% the source file is a tab-delimited file where contiguous rows
% with the same key are considered a single entry. Store the
% index file in the Current Folder. Indicate that keys are
% located in column 3 and that header lines are prefaced with !
gene2goObj = BioIndexedFile('mrtab', sourcefile, '.', ...
                            'KeyColumn', 3, 'HeaderPrefix','!')
```

Return the first, third, and fifth entries from the source file:

```
% Access 1st, 3rd, and 5th entries
subset_entries = getEntryByIndex(gene2goObj, [1 3 5]);
```

Tips

Use this method to visualize and explore a subset of the entries in the source file for validation purposes.

See Also

`BioIndexedFile` | `getEntryByKey` | `getSubset`

Topics

"Work with Next-Generation Sequencing Data"

getEntryByKey

Class: `BioIndexedFile`

Retrieve entries from source file associated with BioIndexedFile object using alphanumeric key

Syntax

`Entries = getEntryByKey(BioIFobj, Key)`

Description

`Entries = getEntryByKey(BioIFobj, Key)` extracts entries from the source file associated with *`BioIFobj`*, a BioIndexedFile object. It extracts and concatenates the entries specified by *`Key`*, a character vector or cell array of character vectors specifying one or more alphanumeric keys. It returns *`Entries`*, a character vector of concatenated entries. If the keys in the source file are not unique, it returns all entries that match a specified key, all at the position of the key in the *`Key`* cell array. If the keys in the source file are unique, there is a one-to-one relationship between the number and order of elements in *`Key`* and the output *`Entries`*.

Input Arguments

`BioIFobj`

Object of the `BioIndexedFile` class.

Default:

`Key`

Character vector or cell array of character vectors specifying one or more keys in the source file associated with *`BioIFobj`*, the BioIndexedFile object.

Default:

Output Arguments

`Entries`

Character vector of concatenated entries extracted from the source file associated with *`BioIFobj`*, the BioIndexedFile object.

Examples

Construct a BioIndexedFile object to access a table containing cross-references between gene names and gene ontology (GO) terms:

```
% Create variable containing full absolute path of source file
sourcefile = which('yeastgenes.sgd');
% Create a BioIndexedFile object from the source file. Indicate
% the source file is a tab-delimited file where contiguous rows
% with the same key are considered a single entry. Store the
% index file in the Current Folder. Indicate that keys are
% located in column 3 and that header lines are prefaced with !
gene2goObj = BioIndexedFile('mrtab', sourcefile, '.', ...
                            'KeyColumn', 3, 'HeaderPrefix','!')
```

Return the entries from the source file that are specified by the keys AAC1 and AAD10:

```
% Access entries that have the keys AAC1 and AAD10
subset_entries = getEntryByKey(gene2goObj, {'AAC1' 'AAD10'});
```

Tips

Use this method to visualize and explore a subset of the entries in the source file for validation purposes.

See Also

`BioIndexedFile` | `getEntryByIndex` | `getKeys` | `getSubset`

Topics

"Work with Next-Generation Sequencing Data"

getExons

Return table of exons from `GTFAnnotation` object

Syntax

```
exons = getExons(AnnotObj)
[exons,junctions]= getExons(AnnotObj)
[ ___ ] = getExons(AnnotObj,"Reference",R)
[ ___ ] = getExons(AnnotObj,"Gene",G)
[ ___ ] = getExons(AnnotObj,"Transcript",T)
```

Description

`exons = getExons(AnnotObj)` returns `exons`, a table of existing exons in `AnnotObj`.

`[exons,junctions]= getExons(AnnotObj)` also returns `junctions`, a table of spliced junctions for each reference listed in `AnnotObj`.

`[ ___ ] = getExons(AnnotObj,"Reference",R)` returns the exons that belong to one or more references specified by `R`.

`[ ___ ] = getExons(AnnotObj,"Gene",G)` returns the exons that belong to one or more genes specified by `G`.

`[ ___ ] = getExons(AnnotObj,"Transcript",T)` returns the exons that belong to one or more transcripts specified by `T`.

Examples

Retrieve Exons from a GTF-formatted File

Create a `GTFAnnotation` object from a GTF-formatted file.

```
obj = GTFAnnotation('hum37_2_1M.gtf');
```

Get the list of gene names listed in the object.

```
gNames = getGeneNames(obj)
```

```
gNames = 28x1 cell
    {'uc002qvu.2'}
    {'uc002qvv.2'}
    {'uc002qvw.2'}
    {'uc002qvx.2'}
    {'uc002qvy.2'}
    {'uc002qvz.2'}
    {'uc002qwa.2'}
    {'uc002qwb.2'}
    {'uc002qwc.1'}
    {'uc002qwd.2'}
    {'uc002qwe.3'}
    {'uc002qwf.2'}
    {'uc002qwg.2'}
    {'uc002qwh.2'}
    {'uc002qwi.3'}
    {'uc002qwk.2'}
    {'uc002qwl.2'}
    {'uc002qwm.1'}
    {'uc002qwn.1'}
    {'uc002qwo.1'}
    {'uc002qwp.2'}
    {'uc002qwq.2'}
    {'uc010ewe.2'}
    {'uc010ewf.1'}
    {'uc010ewg.2'}
    {'uc010ewh.1'}
    {'uc010ewi.2'}
    {'uc010yim.1'}
```

Get a table of exons which belong to the first gene `uc002qvu.2`.

```
exons = getExons(obj,'Gene',gNames{1})
```

```
exons=8×7 table
      Transcript        GeneName          GeneID        Reference    Start      Stop     Strand
    ______________    __________    ______________    _________    ______    ______    ______

    {'uc002qvu.2'}    {0x0 char}    {'uc002qvu.2'}      chr2       218138    219001      -   
    {'uc002qvu.2'}    {0x0 char}    {'uc002qvu.2'}      chr2       224864    224920      -   
    {'uc002qvu.2'}    {0x0 char}    {'uc002qvu.2'}      chr2       229966    230044      -   
    {'uc002qvu.2'}    {0x0 char}    {'uc002qvu.2'}      chr2       231023    231191      -   
    {'uc002qvu.2'}    {0x0 char}    {'uc002qvu.2'}      chr2       233101    233229      -   
    {'uc002qvu.2'}    {0x0 char}    {'uc002qvu.2'}      chr2       234160    234272      -   
    {'uc002qvu.2'}    {0x0 char}    {'uc002qvu.2'}      chr2       247538    247602      -   
    {'uc002qvu.2'}    {0x0 char}    {'uc002qvu.2'}      chr2       249731    249852      -   
```

Input Arguments

`AnnotObj` — GTF annotation

`GTFAnnotation` object

GTF annotation, specified as a `GTFAnnotation` object.

R — Names of reference sequences

character vector | string | string vector | cell array of character vectors | categorical array

Names of reference sequences, specified as a character vector, string, string vector, cell array of character vectors, or categorical array.

The names must come from the `Reference` field of `AnnotObj`. If a name does not exist, the function provides a warning and ignores it.

Data Types: `char` | `string` | `cell` | `categorical`

G — Names of genes
character vector | string | string vector | cell array of character vectors | categorical array

Names of genes, specified as a character vector, string, string vector, cell array of character vectors, or categorical array.

The names must come from the `Gene` field of `AnnotObj`. If a name does not exist, the function provides a warning and ignores the name.

Data Types: `char` | `string` | `cell` | `categorical`

T — Names of transcripts
character vector | string | string vector | cell array of character vectors | categorical array

Names of transcripts, specified as a character vector, string, string vector, cell array of character vectors, or categorical array.

The names must come from the `Transcript` field of `AnnotObj`. If a name does not exist, the function gives a warning and ignores the name.

Data Types: `char` | `string` | `cell` | `categorical`

Output Arguments

`exons` — Exons in `AnnotObj`
table

Exons in `AnnotObj`, returned as a table. The table contains the following variables for each transcript.

Variable Name	Description
`Transcript`	Cell array of character vectors containing transcript IDs, obtained from the `Transcript` field of `AnnotObj`.
`GeneName`	Cell array of character vectors containing the names of expressed genes, obtained from the `Attributes` field of `AnnotObj`. This cell array can contain empty character vectors if the corresponding gene names are not found in `Attributes`.
`GeneID`	Cell array of character vectors containing the expressed gene IDs, obtained from the `Gene` field of `AnnotObj`.
`Reference`	Categorical array representing the names of reference sequences to which the expressed genes belong. The reference names are from the `Reference` field of `AnnotObj`.
`Start`	Start location of each exon.
`Stop`	Stop location of each exon.
`Strand`	Categorical array containing the strand of expressed gene.

`junctions` — Spliced junctions for each reference
table

Spliced junctions for each reference, returned as a table. The table contains the following variables for each junction.

Variable Name	Description
`Start`	Start location of each junction.
`Stop`	Stop location of each junction.
`Reference`	Categorical array representing the names of reference sequences to which the junctions belong. The reference names are from the `Reference` field of `AnnotObj`.

Version History

Introduced in R2014b

See Also

`getData` | `getFeatureNames` | `getGeneNames` | `getGenes` | `getIndex` | `getRange` | `getReferenceNames` | `getSegments` | `getSubset` | `getTranscriptNames` | `getTranscripts` | `GTFAnnotation` | `GFFAnnotation`

Topics

"Store and Manage Feature Annotations in Objects"

External Websites

GTF2.2: A Gene Annotation Format
GFF (General Feature Format) specifications

getFeatureNames

Retrieve unique feature names from `GTFAnnotation` or `GFFAnnotation` object

Syntax

`Features = getFeatureNames(AnnotObj)`

Description

`Features = getFeatureNames(AnnotObj)` returns `Features`, a cell array of character vectors specifying the unique feature names associated with annotations in `AnnotObj`.

Examples

Retrieve Feature Names from a GTFAnnotation Object

Construct a `GTFAnnotation` object from a GTF-formatted file that is provided with Bioinformatics Toolbox™, and then retrieve the feature names from the annotation object.

Construct a `GTFAnnotation` object from a GTF file.

```
GTFAnnotObj = GTFAnnotation('hum37_2_1M.gtf');
```

Retrieve feature names from the annotation object.

```
featureNames = getFeatureNames(GTFAnnotObj)
```

```
featureNames = 4x1 cell
    {'CDS'        }
    {'exon'       }
    {'start_codon'}
    {'stop_codon' }
```

Retrieve Feature Names from a GFFAnnotation Object

Construct a `GFFAnnotation` object from a GFF-formatted file that is provided with Bioinformatics Toolbox™, and then retrieve the feature names from the annotation object.

Construct a `GFFAnnotation` object from a GFF file.

```
GFFAnnotObj = GFFAnnotation('tair8_1.gff');
```

Retrieve feature names for the annotation object.

```
featureNames = getFeatureNames(GFFAnnotObj)
```

```
featureNames = 14x1 cell
    {'CDS'                      }
    {'exon'                   }
    {'five_prime_UTR'         }
    {'gene'                   }
    {'mRNA'                   }
    {'miRNA'                  }
    {'ncRNA'                  }
    {'protein'                }
    {'pseudogene'             }
    {'pseudogenic_exon'       }
    {'pseudogenic_transcript' }
    {'tRNA'                   }
    {'three_prime_UTR'        }
    {'transposable_element_gene'}
```

Input Arguments

`AnnotObj` — Feature annotations
`GTFAnnotation` object | `GFFAnnotation` object

Feature annotations, specified as a `GTFAnnotation` or `GFFAnnotation` object.

Output Arguments

`Features` — Unique feature names associated with annotations in `AnnotObj`
cell array of character vectors

Unique feature names associated with annotations in `AnnotObj`, returned as a cell array of character vectors.

Version History

Introduced in R2011b

See Also

`getData` | `getExons` | `getGeneNames` | `getGenes` | `getIndex` | `getRange` | `getReferenceNames` | `getSegments` | `getSubset` | `getTranscriptNames` | `getTranscripts` | `GTFAnnotation` | `GFFAnnotation`

Topics
“Store and Manage Feature Annotations in Objects”

External Websites
GTF2.2: A Gene Annotation Format
GFF (General Feature Format) specifications

getgenbank

Retrieve sequence information from GenBank database

Syntax

```
Data = getgenbank(AccessionNumber)
getgenbank(AccessionNumber)

Data = getgenbank(..., 'PartialSeq', PartialSeqValue, ...)
Data = getgenbank(..., 'ToFile', ToFileValue, ...)
Data = getgenbank(..., 'FileFormat', FileFormatValue, ...)
Data = getgenbank(..., 'SequenceOnly', SequenceOnlyValue, ...)
Data = getgenbank(..., 'TimeOut', TimeOutValue, ...)
```

Arguments

AccessionNumber	Character vector or string specifying a unique alphanumeric identifier for a sequence record.
PartialSeqValue	Two-element array of integers containing the start and end positions of the subsequence [*StartBP*, *EndBP*] that specifies a subsequence to retrieve. *StartBP* is an integer between 1 and *EndBP*. *EndBP* is an integer between *StartBP* and the length of the sequence.
ToFileValue	Character vector or string specifying either a file name or a path and file name for saving the GenBank data. If you specify only a file name, the file is saved to the MATLAB Current Folder.
FileFormatValue	Character vector or string specifying the format for the sequence information. Choices are: • `'GenBank'` — Default when *SequenceOnlyValue* is `false`. • `'FASTA'` — Default when *SequenceOnlyValue* is `true`. When `'FASTA'`, then *Data* contains only two fields, `Header` and `Sequence`.
SequenceOnlyValue	Controls the return of only the sequence as a character array. Choices are `true` or `false` (default).
TimeOutValue	Connection timeout in seconds, specified as a positive scalar. The default value is 5. For details, see here.

Description

`getgenbank` retrieves nucleotide information from the GenBank database. This database is maintained by the National Center for Biotechnology Information (NCBI). For more details about the GenBank database, see

`https://www.ncbi.nlm.nih.gov/Genbank/`

`Data = getgenbank(AccessionNumber)` searches for the accession number in the GenBank database and returns *`Data`*, a MATLAB structure containing information for the sequence.

Tip If an error occurs while retrieving the GenBank-formatted information, try rerunning the query. Errors can occur due to Internet connectivity issues that are unrelated to the GenBank record.

`getgenbank(AccessionNumber)` displays information in the MATLAB Command Window without returning data to a variable. The displayed information is only hyperlinks to the URLs used to search for and retrieve the data.

`getgenbank(..., 'PropertyName', PropertyValue, ...)` calls `getgenbank` with optional properties that use property name/property value pairs. You can specify one or more properties in any order. Each *`PropertyName`* must be enclosed in single quotation marks and is case insensitive. These property name/property value pairs are as follows:

`Data = getgenbank(..., 'PartialSeq', PartialSeqValue, ...)` returns the specified subsequence in the `Sequence` field of the MATLAB structure. *`PartialSeqValue`* is a two-element array of integers containing the start and end positions of the subsequence `[StartBP, EndBP]`. *`StartBP`* is an integer between 1 and *`EndBP`*. *`EndBP`* is an integer between *`StartBP`* and the length of the sequence.

`Data = getgenbank(..., 'ToFile', ToFileValue, ...)` saves the data returned from the GenBank database to a file. *`ToFileValue`* is a character vector or string specifying either a file name or a path and file name for saving the GenBank data. If you specify only a file name, the file is saved to the MATLAB Current Folder. The function does not append data to an existing file. Instead, it overwrites the contents of the existing file without warning.

Tip You can read a GenBank-formatted file back into MATLAB using the `genbankread` function.

`Data = getgenbank(..., 'FileFormat', FileFormatValue, ...)` returns the sequence in the specified format. Choices are `'GenBank'` or `'FASTA'`. When `'FASTA'`, then *`Data`* contains only two fields, `Header` and `Sequence`. `'GenBank'` is the default when *`SequenceOnlyValue`* is `false`. `'FASTA'` is the default when *`SequenceOnlyValue`* is `true`.

`Data = getgenbank(..., 'SequenceOnly', SequenceOnlyValue, ...)` returns only the sequence in *`Data`*, a character array. Choices are `true` or `false` (default).

Note If you use the `'SequenceOnly'` and `'ToFile'` properties together, the output is always a FASTA-formatted file.

`Data = getgenbank(..., 'TimeOut', TimeOutValue, ...)` sets the connection timeout (in seconds) to retrieve data from the GenBank database.

Examples

Example 1.12. Retrieving an RNA Sequence

To retrieve the sequence from chromosome 19 that codes for the human insulin receptor and store it in a structure, S, in the MATLAB Command Window, type:

```
S = getgenbank('M10051')

S =

                LocusName: 'HUMINSR'
      LocusSequenceLength: '4723'
     LocusNumberofStrands: ''
            LocusTopology: 'linear'
        LocusMoleculeType: 'mRNA'
     LocusGenBankDivision: 'PRI'
    LocusModificationDate: '06-JAN-1995'
               Definition: 'Human insulin receptor mRNA, complete cds.'
                Accession: 'M10051'
                  Version: 'M10051.1'
                       GI: '186439'
                  Project: []
                   DBLink: []
                 Keywords: 'insulin receptor; tyrosine kinase.'
                  Segment: []
                   Source: 'Homo sapiens (human)'
           SourceOrganism: [4x65 char]
                Reference: {[1x1 struct]}
                  Comment: [14x67 char]
                 Features: [51x74 char]
                      CDS: [1x1 struct]
                 Sequence: [1x4723 char]
                SearchURL: [1x67 char]
              RetrieveURL: [1x101 char]
```

Example 1.13. Retrieving a Partial RNA Sequence

By looking at the Features field of the structure returned, you can determine that the coding sequence is positions 139 through 4287. To retrieve only the coding sequence from chromosome 19 that codes for the human insulin receptor and store it in a structure, CDS, in the MATLAB Command Window, type:

```
CDS = getgenbank('M10051','PARTIALSEQ',[139,4287]);
```

Version History

Introduced before R2006a

R2019a: Appending data to existing file

Behavior changed in R2019a

The function no longer appends data to an existing file. The function now overwrites the contents of the existing file without warning.

See Also

genbankread | getembl | getgenpept | getpdb | seqviewer

getGenes

Return table of unique genes in `GTFAnnotation` object

Syntax

```
genes = getGenes(AnnotObj)
genes = getGenes(AnnotObj,"Reference",R)
genes = getGenes(AnnotObj,"Gene",G)
genes = getGenes(AnnotObj,"Transcript",T)
```

Description

`genes = getGenes(AnnotObj)` returns `genes`, a table of genes referenced by exons in `AnnotObj`.

`genes = getGenes(AnnotObj,"Reference",R)` returns one or more genes that belong to one or more references specified by `R`.

`genes = getGenes(AnnotObj,"Gene",G)` returns one or more genes specified by `G`.

`genes = getGenes(AnnotObj,"Transcript",T)` returns one or more genes that contains one or more transcripts specified by `T`.

Examples

Retrieve Genes from a GTF-formatted File

Create a `GTFAnnotation` object from a GTF-formatted file.

```
obj = GTFAnnotation("hum37_2_1M.gtf");
```

Retrieve unique reference names. In this case, there is only one reference sequence, which is chromosome 2 (`chr2`).

```
ref = getReferenceNames(obj)
```

```
ref = 1x1 cell array
    {'chr2'}
```

Get a table of all genes which belong to `chr2`.

```
genes = getGenes(obj,"Reference",ref)
```

```
genes=28×7 table
        GeneID           GeneName      Reference     Start      Stop     Strand    NumTranscripts
    ______________    __________    _________    ______    ______    ______    ______________

    {'uc010yim.1'}    {0x0 char}      chr2        41609     46385      -               1       
    {'uc002qvu.2'}    {0x0 char}      chr2       218138    249852      -               1       
    {'uc002qvv.2'}    {0x0 char}      chr2       218138    256690      -               1       
    {'uc002qvw.2'}    {0x0 char}      chr2       218138    260702      -               1       
    {'uc002qvx.2'}    {0x0 char}      chr2       218138    264068      -            1
    {'uc002qvy.2'}    {0x0 char}      chr2       218138    264068      -            1
    {'uc002qvz.2'}    {0x0 char}      chr2       218138    264392      -            1
    {'uc002qwa.2'}    {0x0 char}      chr2       218138    264743      -            1
    {'uc010ewe.2'}    {0x0 char}      chr2       218138    264810      -            1
    {'uc002qwb.2'}    {0x0 char}      chr2       239563    242178      -            1
    {'uc002qwc.1'}    {0x0 char}      chr2       243503    262786      -            1
    {'uc002qwd.2'}    {0x0 char}      chr2       264869    272481      +            1
    {'uc002qwe.3'}    {0x0 char}      chr2       264869    273148      +            1
    {'uc002qwg.2'}    {0x0 char}      chr2       264869    278280      +            1
    {'uc002qwh.2'}    {0x0 char}      chr2       264869    278280      +            1
    {'uc002qwf.2'}    {0x0 char}      chr2       264869    278280      +            1
       ⋮
```

Input Arguments

`AnnotObj` — GTF annotation
`GTFAnnotation` object

GTF annotation, specified as a `GTFAnnotation` object.

`R` — Names of reference sequences
character vector | string | string vector | cell array of character vectors | categorical array

Names of reference sequences, specified as a character vector, string, string vector, cell array of character vectors, or categorical array.

The names must come from the `Reference` field of `AnnotObj`. If a name does not exist, the function provides a warning and ignores it.

Data Types: `char` | `string` | `cell` | `categorical`

`G` — Names of genes
character vector | string | string vector | cell array of character vectors | categorical array

Names of genes, specified as a character vector, string, string vector, cell array of character vectors, or categorical array.

The names must come from the `Gene` field of `AnnotObj`. If a name does not exist, the function provides a warning and ignores the name.

Data Types: `char` | `string` | `cell` | `categorical`

`T` — Names of transcripts
character vector | string | string vector | cell array of character vectors | categorical array

Names of transcripts, specified as a character vector, string, string vector, cell array of character vectors, or categorical array.

The names must come from the `Transcript` field of `AnnotObj`. If a name does not exist, the function gives a warning and ignores the name.

Data Types: `char` | `string` | `cell` | `categorical`

Output Arguments

genes — Genes referenced by exons in `AnnotObj`
table

Genes referenced by exons in `AnnotObj`, returned as a table. The table contains the following variables for each gene.

Variable Name	Description
`GeneID`	Cell array of character vectors containing gene IDs as listed in `AnnotObj`, obtained from the `Gene` field of `AnnotObj`.
`GeneName`	Cell array of character vectors containing gene names, obtained from the `Attributes` field of `AnnotObj`. This cell array can contain empty character vectors if the corresponding gene names are not found in `Attributes`.
`Reference`	Categorical array representing the names of reference sequences to which the genes belong, obtained from the `Reference` field of `AnnotObj`.
`Start`	Start location of the first exon in each gene.
`Stop`	Stop location of the last exon in each gene.
`Strand`	Categorical array containing the strand of each gene.
`NumTranscripts`	Integer array listing the number of transcripts in each gene.

Version History

Introduced in R2014b

See Also

`getData` | `getExons` | `getFeatureNames` | `getGeneNames` | `getIndex` | `getRange` | `getReferenceNames` | `getSegments` | `getSubset` | `getTranscriptNames` | `getTranscripts` | `GTFAnnotation` | `GFFAnnotation`

Topics

“Store and Manage Feature Annotations in Objects”

External Websites

GTF2.2: A Gene Annotation Format
GFF (General Feature Format) specifications

getGeneNames

Retrieve unique gene names from `GTFAnnotation` object

Syntax

```
Genes = getGeneNames(AnnotObj)
```

Description

`Genes = getGeneNames(AnnotObj)` returns `Genes`, a cell array of character vectors specifying the unique gene names associated with annotations in `AnnotObj`.

Examples

Retrieve Gene Names from a GTFAnnotation Object

Construct a `GTFAnnotation` object from a GTF-formatted file that is provided with Bioinformatics Toolbox™, and then retrieve a list of the unique gene names from the object.

Construct a `GTFAnnotation` object from a GTF file.

```
GTFAnnotObj = GTFAnnotation('hum37_2_1M.gtf');
```

Get gene names from object.

```
geneNames = getGeneNames(GTFAnnotObj)
```

```
geneNames = 28x1 cell
    {'uc002qvu.2'}
    {'uc002qvv.2'}
    {'uc002qvw.2'}
    {'uc002qvx.2'}
    {'uc002qvy.2'}
    {'uc002qvz.2'}
    {'uc002qwa.2'}
    {'uc002qwb.2'}
    {'uc002qwc.1'}
    {'uc002qwd.2'}
    {'uc002qwe.3'}
    {'uc002qwf.2'}
    {'uc002qwg.2'}
    {'uc002qwh.2'}
    {'uc002qwi.3'}
    {'uc002qwk.2'}
    {'uc002qwl.2'}
    {'uc002qwm.1'}
    {'uc002qwn.1'}
    {'uc002qwo.1'}
    {'uc002qwp.2'}
    {'uc002qwq.2'}
    {'uc010ewe.2'}
    {'uc010ewf.1'}
    {'uc010ewg.2'}
    {'uc010ewh.1'}
    {'uc010ewi.2'}
    {'uc010yim.1'}
```

Input Arguments

`AnnotObj` — GTF annotation
`GTFAnnotation` object

GTF annotation, specified as a `GTFAnnotation` object.

Output Arguments

`Genes` — Unique gene names associated with annotations in `AnnotObj`
cell array of character vectors

Unique gene names associated with annotations in `AnnotObj`, returned as a cell array of character vectors.

Version History

Introduced in R2011b

See Also

`getData` | `getExons` | `getFeatureNames` | `getGenes` | `getIndex` | `getRange` | `getReferenceNames` | `getSegments` | `getSubset` | `getTranscriptNames` | `getTranscripts` | `GTFAnnotation` | `GFFAnnotation`

Topics
"Store and Manage Feature Annotations in Objects"

External Websites
GTF2.2: A Gene Annotation Format
GFF (General Feature Format) specifications

getgenpept

Retrieve sequence information from GenPept database

Syntax

```
Data = getgenpept(AccessionNumber)
getgenpept(AccessionNumber)

Data = getgenpept(..., 'PartialSeq', PartialSeqValue, ...)
Data = getgenpept(..., 'ToFile', ToFileValue, ...)
Data = getgenpept(..., 'FileFormat', FileFormatValue, ...)
Data = getgenpept(..., 'SequenceOnly', SequenceOnlyValue, ...)
Data = getgenpept(..., 'TimeOut, TimeOutValue, ...)
```

Arguments

AccessionNumber	Character vector specifying a unique alphanumeric identifier for a sequence record.
PartialSeqValue	Two-element array of integers containing the start and end positions of the subsequence [*StartAA*, *EndAA*] that specifies a subsequence to retrieve. *StartAA* is an integer between 1 and *EndAA*; *EndAA* is an integer between *StartAA* and the length of the sequence.
ToFileValue	Character vector specifying either a file name or a path and file name for saving the GenPept data. If you specify only a file name, the file is saved to the MATLAB Current Folder.
FileFormatValue	Character vector specifying the format for the sequence information. Choices are: • `'Genpept'` — Default when *SequenceOnlyValue* is `false`. • `'FASTA'` — Default when *SequenceOnlyValue* is `true`. When `'FASTA'`, then *Data* contains only two fields, `Header` and `Sequence`.
SequenceOnlyValue	Controls the return of only the sequence as a character array. Choices are `true` or `false` (default).
TimeOutValue	Connection timeout in seconds, specified as a positive scalar. The default value is 5. For details, see here.

Description

`getgenpept` retrieves a protein (amino acid) sequence information from the GenPept database, which is a translation of the nucleotide sequences in the GenBank database and is maintained by the National Center for Biotechnology Information (NCBI).

Note NCBI has changed the name of their protein search engine from GenPept to Entrez Protein. However, the function names in the Bioinformatics Toolbox software (`getgenpept` and `genpeptread`) are unchanged representing the still-used GenPept report format. For more information on GenPept data, visit https://www.ncbi.nlm.nih.gov/home/about/policies.shtml.

`Data = getgenpept(AccessionNumber)` searches for the accession number in the GenPept database and returns *`Data`*, a MATLAB structure containing information for the sequence.

Tip If an error occurs while retrieving the GenPept-formatted information, try rerunning the query. Errors can occur due to Internet connectivity issues that are unrelated to the GenPept record.

`getgenpept(AccessionNumber)` displays information in the MATLAB Command Window without returning data to a variable. The displayed information is only hyperlinks to the URLs used to search for and retrieve the data.

`getgenpept(..., 'PropertyName', PropertyValue, ...)` calls `getgenpept` with optional properties that use property name/property value pairs. You can specify one or more properties in any order. Each *`PropertyName`* must be enclosed in single quotation marks and is case insensitive. These property name/property value pairs are as follows:

`Data = getgenpept(..., 'PartialSeq', PartialSeqValue, ...)` returns the specified subsequence in the `Sequence` field of the MATLAB structure. *`PartialSeqValue`* is a two-element array of integers containing the start and end positions of the subsequence `[StartAA, EndAA]`. *`StartAA`* is an integer between 1 and *`EndAA`*; *`EndAA`* is an integer between *`StartAA`* and the length of the sequence.

`Data = getgenpept(..., 'ToFile', ToFileValue, ...)` saves the data returned from the GenPept database to a file. *`ToFileValue`* is a character vector specifying either a file name or a path and file name for saving the GenPept data. If you specify only a file name, the file is saved to the MATLAB Current Folder. The function does not append data to an existing file. Instead, it overwrites the contents of the existing file without warning.

Tip You can read a GenPept-formatted file back into MATLAB using the `genpeptread` function.

`Data = getgenpept(..., 'FileFormat', FileFormatValue, ...)` returns the sequence in the specified format. Choices are `'GenPept'` or `'FASTA'`. When `'FASTA'`, then *`Data`* contains only two fields, `Header` and `Sequence`. `'GenPept'` is the default when *`SequenceOnlyValue`* is `false`. `'FASTA'` is the default when *`SequenceOnlyValue`* is `true`.

`Data = getgenpept(..., 'SequenceOnly', SequenceOnlyValue, ...)` returns only the sequence in *`Data`*, a character array. Choices are `true` or `false` (default).

Note If you use the `'SequenceOnly'` and `'ToFile'` properties together, the output is always a FASTA-formatted file.

`Data = getgenpept(..., 'TimeOut, TimeOutValue, ...)` sets the connection timeout to retrieve data from the GenPept database.

Examples

Example 1.14. Retrieving a Peptide Sequence

To retrieve the sequence for the human insulin receptor and store it in a structure, `Seq`, in the MATLAB Command Window, type:

```
Seq = getgenpept('AAA59174')

Seq =

                LocusName: 'AAA59174'
     LocusSequenceLength: '1382'
    LocusNumberofStrands: ''
           LocusTopology: 'linear'
       LocusMoleculeType: ''
    LocusGenBankDivision: 'PRI'
   LocusModificationDate: '06-JAN-1995'
              Definition: 'insulin receptor precursor.'
               Accession: 'AAA59174'
                 Version: 'AAA59174.1'
                      GI: '307070'
                 Project: []
                DBSource: 'locus HUMINSR accession M10051.1'
                Keywords: ''
                  Source: 'Homo sapiens (human)'
          SourceOrganism: [4x65 char]
               Reference: {[1x1 struct]}
                 Comment: [14x67 char]
                Features: [40x64 char]
                Sequence: [1x1382 char]
               SearchURL: [1x104 char]
             RetrieveURL: [1x92 char]
```

Example 1.15. Retrieving a Partial Peptide Sequence

By looking at the `Features` field of the structure, you can determine that the furin-like repeats domain is positions 234 through 281. To retrieve only the furin-like repeats domain from the sequence for the human insulin receptor and store it in a structure, `Fur`, in the MATLAB Command Window, type:

```
Fur = getgenpept('AAA59174','PARTIALSEQ',[234,281]);
```

Version History

Introduced before R2006a

R2019a: Appending data to existing file

Behavior changed in R2019a

The function no longer appends data to an existing file. The function now overwrites the contents of the existing file without warning.

See Also

`genpeptread` | `getembl` | `getgenbank` | `getpdb`

getgeodata

Retrieve Gene Expression Omnibus (GEO) format data

Syntax

```
GEOData = getgeodata(AccessionNumber)

getgeodata(AccessionNumber, 'ToFile', ToFileValue)
GEOData = getgeodata(AccessionNumber,'TimeOut',TimeOutValue)
```

Input Arguments

AccessionNumber	Character vector specifying a unique identifier for a GEO Sample (GSM), Data Set (GDS), Platform (GPL), or Series (GSE) record in the GEO database. Next-Generation Sequencing data cannot be retrieved using this function. **Tip** • If you are unable to retrieve data for an accession number, increase your Java heap space: • If you have MATLAB version 7.10 (R2010a) or later, see “Java Heap Memory Preferences”. • If you have MATLAB version 7.9 (R2009b) or earlier, see https://www.mathworks.com/matlabcentral/answers/92813-how-do-i-increase-the-heap-space-for-the-java-vm-in-matlab. • Recently submitted data sets may not be available for immediate download. There can be a one- to two-day delay between an experiment being submitted to the GEO database and its availability on the FTP site.
ToFileValue	Character vector specifying a file name or path and file name for saving the data. If you specify only a file name, that file will be saved in the MATLAB Current Folder.
TimeOutValue	Connection timeout in seconds, specified as a positive scalar. The default value is 5. For details, see here.

Output Arguments

GEOData	MATLAB structure containing information for a GEO record retrieved from the GEO database.

Description

`GEOData = getgeodata(AccessionNumber)` searches the Gene Expression Omnibus database for the specified accession number of a Sample (GSM), Data Set (GDS), Platform (GPL), or Series (GSE) record and returns a MATLAB structure containing the following fields:

Field	Description
`Scope`	Type of data retrieved (SAMPLE, DATASET, PLATFORM, or SERIES)
`Accession`	Accession number for record in GEO database.
`Header`	Microarray experiment information.
`ColumnDescriptions`	Cell array containing descriptions of columns in the data.
`ColumnNames`	Cell array containing names of columns in the data.
`Data`	Array containing microarray data.
`Identifier` (GDS files only)	Cell array containing probe IDs.
`IDRef` (GDS files only)	Cell array containing indices to probes.

Note Currently, the `getgeodata` function supports Sample (GSM), Data Set (GDS), Platform (GPL), and Series (GSE) records.

`getgeodata(`*`AccessionNumber`*`, 'ToFile', `*`ToFileValue`*`)` saves the data returned from the database to a file.

Note You can read a GEO SOFT-formatted file back into the MATLAB software using the `geosoftread` function. You can read a GEO SERIES-formatted file back into the MATLAB software using the `geoseriesread` function.

For more information, see

`https://www.ncbi.nlm.nih.gov/About/disclaimer.html`

`GEOData`` = getgeodata(`*`AccessionNumber`*`,'TimeOut',`*`TimeOutValue`*`)` sets the connection timeout (in seconds) to retrieve data from the Gene Expression Omnibus database.

Examples

```
geoStruct = getgeodata('GSM1768')
```

Version History

Introduced before R2006a

See Also

`geoseriesread` | `geosoftread` | `getgenbank` | `getgenpept`

getHeader

Retrieve sequence headers from object

Syntax

```
headers = getHeader(object)
subsetHeaders = getHeader(object,subset)
```

Description

`headers = getHeader(object)` returns sequence `headers` (names) from a `BioRead` or `BioMap` object.

`subsetHeaders = getHeader(object,subset)` returns the header information `subsetHeaders` for only the object elements specified by `subset`.

Examples

Retrieve Sequence Header Information

Store read data from a SAM-formatted file in a BioRead object.

```
br = BioRead('ex1.sam')
```

```
br = 
  BioRead with properties:

     Quality: [1501x1 File indexed property]
    Sequence: [1501x1 File indexed property]
      Header: [1501x1 File indexed property]
       NSeqs: 1501
        Name: ''
```

Retrieve sequence header information.

```
allHeaders = getHeader(br);
```

Retrieve sequence header information from the first and third elements in the object.

```
subset = getHeader(br,[1 3])
```

```
subset = 2x1 cell
    {'B7_591:4:96:693:509'}
    {'EAS51_64:8:5:734:57'}
```

Use a logical vector to get the same information.

```
subset2 = getHeader(br,[true false true])
```

```
subset2 = 2x1 cell
    {'B7_591:4:96:693:509'}
    {'EAS51_64:8:5:734:57'}
```

Access each property of the object by using the dot notation.

```
allHeaders  = br.Header;
subset      = br.Header([1 3])
```

```
subset = 2x1 cell
    {'B7_591:4:96:693:509'}
    {'EAS51_64:8:5:734:57'}
```

Input Arguments

`object` — Object containing read data
`BioRead` object | `BioMap` object

Object containing the read data, specified as a `BioRead` or `BioMap` object.

Example: `bioreadObj`

`subset` — Subset of elements in object
vector of positive integers | logical vector | string vector | cell array of character vectors

Subset of elements in the object, specified as a vector of positive integers, logical vector, string vector, or cell array of character vectors containing valid sequence headers.

Example: `[1 3]`

Output Arguments

`headers` — Sequence headers
cell array of character vectors

Sequence headers, returned as a cell array of character vectors.

`subsetHeaders` — Sequence headers for subset of elements
cell array of character vectors

Sequence headers for a subset of elements from the object, returned as a cell array of character vectors.

Version History

Introduced in R2010a

See Also

`BioRead` | `BioMap`

Topics

"Manage Sequence Read Data in Objects"

gethmmalignment

Retrieve multiple sequence alignment associated with hidden Markov model (HMM) profile from PFAM database

Syntax

```
AlignStruct = gethmmalignment(PFAMName)
AlignStruct = gethmmalignment(PFAMAccessNumber)
AlignStruct = gethmmalignment(PFAMNumber)

AlignStruct = gethmmalignment(..., 'ToFile', ToFileValue, ...)
AlignStruct = gethmmalignment(..., 'Type', TypeValue, ...)
AlignStruct = gethmmalignment(..., 'IgnoreGaps', IgnoreGaps, ...)
AlignStruct = gethmmalignment(..., 'TimeOut', TimeOutValue, ...)
```

Input Arguments

PFAMName	Character vector specifying a protein family name (unique identifier) of an HMM profile record in the PFAM database. For example, `'7tm_2'`.
PFAMAccessNumber	Character vector specifying a protein family accession number of an HMM profile record in the PFAM database. For example, `'PF00002'`.
PFAMNumber	Integer specifying a protein family number of an HMM profile record in the PFAM database. For example, `2` is the protein family number for the protein family `PF00002`.
ToFileValue	Character vector specifying a file name or a path and file name for saving the data. If you specify only a file name, that file will be saved in the MATLAB Current Folder.
TypeValue	Character vector that specifies the set of alignments returned. Choices are: • `'full'` — Default. Returns all alignments that fit the HMM profile. • `'seed'` — Returns only the alignments used to generate the HMM profile.
IgnoreGapsValue	Controls the removal of the symbols `-` and `.` from the sequence. Choices are `true` or `false` (default).
TimeOutValue	Connection timeout in seconds, specified as a positive scalar. The default value is 5. For details, see here.

Output Arguments

AlignStruct	MATLAB structure array containing the multiple sequence alignment associated with an HMM profile.

Description

AlignStruct = gethmmalignment(*PFAMName*) searches the PFAM database (http://pfam.xfam.org/) for the HMM profile record represented by *PFAMName*, a protein family name, retrieves the multiple sequence alignment associated with the HMM profile, and returns *AlignStruct*, a MATLAB structure array, with each structure containing the following fields:

Field	Description
Header	Protein name
Sequence	Protein sequence

AlignStruct = gethmmalignment(*PFAMAccessNumber*) searches the PFAM database for the HMM profile record represented by *PFAMAccessNumber*, a protein family accession number, retrieves the multiple sequence alignment associated with the HMM profile, and returns *AlignStruct*, a MATLAB structure array.

AlignStruct = gethmmalignment(*PFAMNumber*) determines a protein family accession number from *PFAMNumber*, an integer, searches the PFAM database for the associated HMM profile record, retrieves the multiple sequence alignment associated with the HMM profile, and returns *AlignStruct*, a MATLAB structure array.

AlignStruct = gethmmalignment(..., '*PropertyName*', *PropertyValue*, ...) calls gethmmalignment with optional properties that use property name/property value pairs. You can specify one or more properties in any order. Each *PropertyName* must be enclosed in single quotation marks and is case insensitive. These property name/property value pairs are as follows:

AlignStruct = gethmmalignment(..., 'ToFile', *ToFileValue*, ...) saves the data returned from the PFAM database to a file specified by *ToFileValue*.

Note You can read a FASTA-formatted file containing PFAM data back into the MATLAB software using the fastaread function.

AlignStruct = gethmmalignment(..., 'Type', *TypeValue*, ...) specifies the set of alignments returned. Choices are:

- 'full' — Default. Returns all sequences that fit the HMM profile.
- 'seed' — Returns only the sequences used to generate the HMM profile.

AlignStruct = gethmmalignment(..., 'IgnoreGaps', *IgnoreGaps*, ...) controls the removal of the symbols - and . from the sequence. Choices are true or false (default).

AlignStruct = gethmmalignment(..., 'TimeOut', *TimeOutValue*, ...) sets the connection timeout (in seconds) to retrieve data from the PFAM database.

Examples

To retrieve a multiple alignment of the sequences used to train the HMM profile for global alignment to the 7-transmembrane receptor protein in the secretin family, enter:

```
pfamalign = gethmmalignment('PF00002','Type','seed')
pfamalign = 

  29×1 struct array with fields:

    Header
    Sequence
```

Version History

Introduced before R2006a

See Also

`fastaread` | `gethmmprof` | `gethmmtree` | `multialignread` | `multialignwrite` | `pfamhmmread`

gethmmprof

Retrieve hidden Markov model (HMM) profile from PFAM database

Syntax

```
HMMStruct = gethmmprof(PFAMName)
HMMStruct = gethmmprof(PFAMNumber)

HMMStruct = gethmmprof(..., 'ToFile', ToFileValue, ...)
HMMStruct = gethmmprof(..., 'Mode', ModeValue, ...)
HMMStruct = gethmmprof(..., 'TimeOut', TimeOutValue, ...)
```

Input Arguments

PFAMName	Character vector specifying a protein family name (unique identifier) of an HMM profile record in the PFAM database. For example, `'7tm_2'`.
PFAMNumber	Integer specifying a protein family number of an HMM profile record in the PFAM database. For example, `2` is the protein family number for the protein family `'PF00002'`.
ToFileValue	Character vector specifying a file name or a path and file name for saving the data. If you specify only a file name, that file will be saved in the MATLAB Current Folder.
ModeValue	Character vector that specifies the returned alignment mode. Choices are: • `'ls'` — Default. Global alignment mode. • `'fs'` — Local alignment mode.
TimeOutValue	Connection timeout in seconds, specified as a positive scalar. The default value is 5. For details, see here.

Output Arguments

HMMStruct	MATLAB structure containing information for an HMM profile retrieved from the PFAM database.

Description

Note `gethmmprof` retrieves information from PFAM-HMM profiles, from file format version HMMER2.0 to HMMER3/f.

`HMMStruct` `= gethmmprof(`*`PFAMName`*`)` searches the PFAM database (http://pfam.xfam.org/) for the record represented by *`PFAMName`* (a protein family name), retrieves the HMM profile information, and stores it in *`HMMStruct`*, a MATLAB structure containing the following fields corresponding to parameters of an HMM profile.

Field	Description
`Name`	The protein family name (unique identifier) of the HMM profile record in the PFAM database.
`PfamAccessionNumber`	The protein family accession number of the HMM profile record in the PFAM database.
`ModelDescription`	Description of the HMM profile.
`ModelLength`	The length of the profile (number of MATCH states).
`Alphabet`	The alphabet used in the model, `'AA'` or `'NT'`. **Note** `AlphaLength` is 20 for `'AA'` and 4 for `'NT'`.
`MatchEmission`	Symbol emission probabilities in the MATCH states. The format is a matrix of size `ModelLength`-by-`AlphaLength`, where each row corresponds to the emission distribution for a specific MATCH state.
`InsertEmission`	Symbol emission probabilities in the INSERT state. The format is a matrix of size `ModelLength`-by-`AlphaLength`, where each row corresponds to the emission distribution for a specific INSERT state.
`NullEmission`	Symbol emission probabilities in the MATCH and INSERT states for the NULL model. The format is a 1-by-`AlphaLength` row vector. **Note** NULL probabilities are also known as the background probabilities.

`BeginX`	BEGIN state transition probabilities.
	
	Format is a 1-by-`(ModelLength + 1)` row vector:
	
	`[B->D1 B->M1 B->M2 B->M3 .... B->Mend]`

`MatchX`	MATCH state transition probabilities.
	
	Format is a 4-by-`(ModelLength - 1)` matrix:
	
	`[M1->M2 M2->M3 ... M[end-1]->Mend;`
	` M1->I1 M2->I2 ... M[end-1]->I[end-1];`
	` M1->D2 M2->D3 ... M[end-1]->Dend;`
	` M1->E  M2->E  ... M[end-1]->E  ]`

`InsertX`	INSERT state transition probabilities.
	
	Format is a 2-by-`(ModelLength - 1)` matrix:
	
	`[ I1->M2 I2->M3 ... I[end-1]->Mend;`
	`  I1->I1 I2->I2 ... I[end-1]->I[end-1] ]`

Field	Description

`DeleteX`	DELETE state transition probabilities.
	
	Format is a 2-by-(`ModelLength - 1`) matrix:
	
	`[ D1->M2 D2->M3 ... D[end-1]->Mend ;`
	`  D1->D2 D2->D3 ... D[end-1]->Dend ]`

`FlankingInsertX`	Flanking insert states (N and C) used for LOCAL profile alignment.
	
	Format is a 2-by-2 matrix:
	
	`[N->B  C->T ;`
	` N->N  C->C]`

`LoopX`	Loop states transition probabilities used for multiple hits alignment.
	
	Format is a 2-by-2 matrix:
	
	`[E->C  J->B ;`
	` E->J  J->J]`

`NullX`	Null transition probabilities used to provide scores with log-odds values also for state transitions. Format is a 2-by-1 column vector: `[G->F ; G->G]`

`*HMMStruct* = gethmmprof(*PFAMNumber*)` determines a protein family accession number from *`PFAMNumber`* (an integer), searches the PFAM database for the associated record, retrieves the HMM profile information, and stores it in *`HMMStruct`*, a MATLAB structure.

`*HMMStruct* = gethmmprof(..., '*PropertyName*', *PropertyValue*, ...)` calls `gethmmprof` with optional properties that use property name/property value pairs. You can specify one or more properties in any order. Each *`PropertyName`* must be enclosed in single quotation marks and is case insensitive. These property name/property value pairs are as follows:

`*HMMStruct* = gethmmprof(..., 'ToFile', *ToFileValue*, ...)` saves the data returned from the PFAM database in a file specified by *`ToFileValue`*.

Note You can read an HMM-formatted file back into the MATLAB software using the `pfamhmmread` function.

`*HMMStruct* = gethmmprof(..., 'Mode', *ModeValue*, ...)` specifies the returned alignment mode. Choices are:

- `'ls'` (default) — Global alignment mode.
- `'fs'` — Local alignment mode.

For more information on HMM profile models, see “HMM Profile Model” on page 1-1108.

`*HMMStruct* = gethmmprof(..., 'TimeOut', *TimeOutValue*, ...)` sets the connection timeout (in seconds) to retrieve data the PFAM database.

Examples

To retrieve a hidden Markov model (HMM) profile for the global alignment of the 7-transmembrane receptor protein in the secretin family, enter:

```
hmm = gethmmprof('7tm_2')

hmm =

  struct with fields:

                 Name: '7tm_2'
    PfamAccessionNumber: 'PF00002.21'
       ModelDescription: '7 transmembrane receptor (Secretin family)'
            ModelLength: 241
               Alphabet: 'AA'
          MatchEmission: [241×20 double]
         InsertEmission: [241×20 double]
           NullEmission: [1×20 double]
                 BeginX: [242×1 double]
                 MatchX: [240×4 double]
                InsertX: [240×2 double]
                DeleteX: [240×2 double]
        FlankingInsertX: [2×2 double]
                  LoopX: [2×2 double]
                  NullX: [2×1 double]
```

Version History

Introduced before R2006a

See Also

gethmmalignment | hmmprofalign | hmmprofstruct | pfamhmmread | showhmmprof

gethmmtree

Retrieve phylogenetic tree data from PFAM database

Syntax

```
Tree = gethmmtree(PFAMName)
Tree = gethmmtree(PFAMAccessionNumber)
Tree = gethmmtree(PFAMNumber)

Tree = gethmmtree(...'ToFile', ToFileValue, ...)
Tree = gethmmtree(...'TimeOut',TimeOutValue)
```

Input Arguments

PFAMName	Character vector specifying a protein family name (unique identifier) of an HMM profile record in the PFAM database. For example, `'7tm_2'`.
PFAMAccessionNumber	Character vector specifying a protein family accession number of an HMM profile record in the PFAM database. For example, `'PF00002'`.
PFAMNumber	Integer specifying a protein family number of an HMM profile record in the PFAM database. For example, `2` is the protein family number for the protein family `PF0002`.
ToFileValue	Property to specify the location and file name for saving data. Enter either a file name or a path and file name supported by your system (ASCII text file).
TimeOutValue	Connection timeout in seconds, specified as a positive scalar. The default value is 5. For details, see here.

Output Arguments

Tree	An object containing a phylogenetic tree representative of the protein family.

Description

`Tree = gethmmtree(`*`PFAMName`*`)` searches the PFAM database for the record represented by *`PFAMName`*, a protein family name, retrieves information, and returns `Tree`, an object containing a phylogenetic tree representative of the protein family.

`Tree = gethmmtree(`*`PFAMAccessionNumber`*`)` searches the PFAM database for the record represented by *`PFAMAccessionNumber`*, a protein family accession number, retrieves information, and returns `Tree`, an object containing a phylogenetic tree representative of the protein family.

`Tree = gethmmtree(`*`PFAMNumber`*`)` determines a protein family accession number from *`PFAMNumber`*, an integer, searches the PFAM database for the associated record, retrieves information, and returns `Tree`, an object containing a phylogenetic tree representative of the protein family.

`Tree = gethmmtree(...'PropertyName', PropertyValue, ...)` calls `gethmmtree` with optional properties that use property name/property value pairs. You can specify one or more properties in any order. Each *`PropertyName`* must be enclosed in single quotation marks and is case insensitive. These property name/property value pairs are as follows:

`Tree = gethmmtree(...'ToFile', ToFileValue, ...)` saves the data returned from the PFAM database in the file *`ToFileValue`*.

Tip To download the `'seed'` tree, use `gethmmtree` without any extra input arguments. To obtain the `'full'` tree, you may use the `gethmmalignment` function to download the `'full'` alignment and build a tree using the `seqpdist` and `seqneighjoin` functions as illustrated in the following example.

`Tree = gethmmtree(...'TimeOut',TimeOutValue)` sets the connection timeout (in seconds) to retrieve data from the PFAM database.

Examples

Retrieve phylogenetic tree built from the multiple-aligned sequences used to train the HMM profile model for global alignment. The PFAM accession number `PF00002` is for the 7-transmembrane receptor protein in the secretin family.

```
tree  = gethmmtree('PF00002');
```

Recover the `'full'` tree for the same family by downloading the full multiple sequence alignment and building the tree using the `seqdist` and `seqneighjoin` functions. It may take some considerable amount of time to calculate the tree for large families.

```
seqs = gethmmalignment('PF00002','type','full');
dis = seqpdist(seqs);
tree = seqneighjoin(dis,'equivar',seqs);
```

Version History

Introduced before R2006a

See Also

`gethmmalignment` | `phytreeread`

getIndex

Return index array of annotations from `GTFAnnotation` or `GFFAnnotation` object

Syntax

```
Idx = getIndex(AnnotObj)
Idx = getIndex(AnnotObj,StartPos,EndPos)
Idx = getIndex( ___ ,Name,Value)
```

Description

`Idx = getIndex(AnnotObj)` returns an index array `Idx`, an array of integers containing the index of each annotation in `AnnotObj`.

`Idx = getIndex(AnnotObj,StartPos,EndPos)` returns an index array `Idx` for a subset of elements that falls within each reference sequence range specified by `StartPos` and `EndPos`.

`Idx = getIndex( ___ ,Name,Value)` returns an index array `Idx`, using any of the input arguments from the previous syntaxes and additional options specified by one or more `Name,Value` pair arguments.

Examples

Retrieve Indices of Annotations from a `GTFAnnotation` Object

Construct a `GTFAnnotation` object using a GTF-formatted file that is provided with Bioinformatics Toolbox™.

```
GTFAnnotObj = GTFAnnotation('hum37_2_1M.gtf');
```

Extract indices of annotations for positions 210,000 through 220,000 from the reference sequence.

```
Idx = getIndex(GTFAnnotObj,210000,220000)
```

```
Idx = 16×1

     7
    15
    16
    17
    36
    47
    48
    49
    69
    70
      ⋮
```

Retrieve Indices of Annotations from a GFFAnnotation Object

Construct a `GFFAnnotation` object using a GFF-formatted file that is provided with Bioinformatics Toolbox™.

```
GFFAnnotObj = GFFAnnotation('tair8_1.gff');
```

Extract indices of annotations or features for positions 10,000 through 20,000 from the reference sequence.

```
Idx = getIndex(GFFAnnotObj,10000,20000)
```

```
Idx = 9×1

    61
    62
    63
    64
    65
    66
    67
    68
    69
```

Input Arguments

`AnnotObj` — Feature annotations
`GTFAnnotation` object | `GFFAnnotation` object

Feature annotations, specified as a `GTFAnnotation`or `GFFAnnotation` object.

`StartPos` — Start of a range in each reference sequence in `AnnotObj`
nonnegative integer

Start of a range in each reference sequence in `AnnotObj`, specified as a nonnegative integer less than or equal to `EndPos`.

Data Types: `double`

`EndPos` — End of a range in each reference sequence in `AnnotObj`
nonnegative integer

End of a range in each reference sequence in `AnnotObj`, specified as a nonnegative integer greater than or equal to `StartPos`.

Data Types: `double`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example:

`Reference` — One or more reference sequences in `AnnotObj`
character vector | string | string vector | cell array of character vectors

One or more reference sequences in `AnnotObj`, specified as a character vector, string, string vector, or cell array of character vectors. Only annotations whose reference field matches one of the character vectors or strings are included in `Idx`.

Data Types: `char` | `string` | `cell`

`Feature` — One or more features in `AnnotObj`
character vector | string | string vector | cell array of character vectors

One or more features in `AnnotObj`, specified as a character vector, string, string vector, or cell array of character vectors. Only annotations whose feature field matches one of the character vectors or strings are included in `Idx`.

Data Types: `char` | `string` | `cell`

`Gene` — One or more genes in `AnnotObj` of type `GTFAnnotation`
character vector | string | string vector | cell array of character vectors

One or more genes in `AnnotObj` of type `GTFAnnotation`, specified as a character vector, string, string vector, or cell array of character vectors. Only annotations whose gene field matches one of the character vectors or strings are included in `Idx`.

Data Types: `char` | `string` | `cell`

`Transcript` — One or more transcripts in `AnnotObj` of type `GTFAnnotation`
character vector | string | string vector | cell array of character vectors

One or more transcripts in `AnnotObj` of type `GTFAnnotation`, specified as a character vector, string, string vector, or cell array of character vectors. Only annotations whose transcript field matches one of the character vectors or strings are included in `Idx`.

Data Types: `char` | `string` | `cell`

`Overlap` — Minimum number of base positions that annotation must overlap in the range
1 (default) | positive integer | `"full"` | `"start"`

Minimum number of base positions that annotation must overlap in the range, to be included in `Idx`, specified as a positive integer, `"full"` or `"start"`. Use `"full"` when an annotation must be fully contained in the range to be included. Use `"start"` when an annotation's start position must lie within the range to be included.

Data Types: `double` | `char` | `string`

Output Arguments

`Idx` — Indices of elements in `AnnotObj`
array of integers

Indices of elements in `AnnotObj`, returned as an array of integers.

Version History

Introduced in R2013a

See Also
`getData` | `getExons` | `getFeatureNames` | `getGeneNames` | `getGenes` | `getRange` | `getReferenceNames` | `getSegments` | `getSubset` | `getTranscriptNames` | `getTranscripts` | `GTFAnnotation` | `GFFAnnotation`

Topics
"Store and Manage Feature Annotations in Objects"

External Websites
GTF2.2: A Gene Annotation Format
GFF (General Feature Format) specifications

getIndexByKey

Class: `BioIndexedFile`

Retrieve indices from source file associated with BioIndexedFile object using alphanumeric key

Syntax

```
Indices = getIndexByKey(BioIFobj, Key)
[Indices, LogicalVals] = getIndexByKey(BioIFobj, Key)
```

Description

`Indices = getIndexByKey(BioIFobj, Key)` returns the indices of entries in the source file associated with *`BioIFobj`*, a BioIndexedFile object. It returns the indices of entries that have the keys specified by *`Key`*, a character vector or cell array of character vectors specifying one or more alphanumeric keys. It returns *`Indices`*, a numeric vector of the indices of entries that have the alphanumeric keys specified by *`Key`*. If the keys in the source file are not unique, it returns all indices of entries that match a specified key, all at the position of the key in the *`Key`* cell array. If the keys in the source file are unique, there is a one-to-one relationship between the number and order of elements in *`Key`* and the output *`Indices`*.

`[Indices, LogicalVals] = getIndexByKey(BioIFobj, Key)` returns a logical vector that indicates only the last match for each key, such that there is a one-to-one relationship between the number and order of elements in *`Key`* and *`Indices(LogicalVals)`*.

Input Arguments

`BioIFobj`

Object of the `BioIndexedFile` class.

Default:

`Key`

Character vector or cell array of character vectors specifying one or more keys in the source file associated with *`BioIFobj`*, the BioIndexedFile object.

Default:

Output Arguments

`Indices`

Numeric vector of the indices of entries in source file that have the alphanumeric keys specified by *`Key`*.

`LogicalVals`

Logical vector containing the same number of elements as *`Indices`*. The vector indicates only the last match for each key specified in *`Key`*, such that there is a one-to-one relationship between the number and order of elements in *`Key`* and *`Indices`*`(`*`LogicalVals`*`)`.

Tip Some files contain repeated keys. For example, SAM-formatted files use the same key for entries that are paired end reads. Use the *`Indices`*`(`*`LogicalVals`*`)` syntax to return only the last index of a repeated key. For more information, see “Examples” on page 1-978.

Examples

Construct a BioIndexedFile object to access a table containing cross-references between gene names and gene ontology (GO) terms:

```
% Create variable containing full absolute path of source file
sourcefile = which('yeastgenes.sgd');
% Create a BioIndexedFile object from the source file. Indicate
% the source file is a tab-delimited file where contiguous rows
% with the same key are considered a single entry. Store the
% index file in the Current Folder. Indicate that keys are
% located in column 3 and that header lines are prefaced with !
gene2goObj = BioIndexedFile('mrtab', sourcefile, '.', ...
                            'KeyColumn', 3, 'HeaderPrefix','!')
```

Return the indices for the entries in the source file that are specified by the keys AAC1 and AAD10.

```
% Access indices for entries that have the keys AAC1 and AAD10
indices = getIndexByKey(gene2goObj, {'AAC1' 'AAD10'})

indices =

           3
                 5
```

Construct a BioIndexedFile object to access a SAM-formatted file that has repeated keys.

```
% Create variable containing full absolute path of source file
samsourcefile = which('ex1.sam');
% Create a BioIndexedFile object from the source file. Store the
% index file in the Current Folder.
samObj = BioIndexedFile('sam', samsourcefile, '.')
```

Return only the last indices for the entries in the source file that are specified by two keys,'B7_593:7:15:244:876 and EAS56_65:4:296:78:421, both of which are repeated keys.

```
% Return all indices for entries that have two specific keys
[Indices, LogicalVal] = getIndexByKey(samObj, ...
                  {'B7_593:7:15:244:876', 'EAS56_65:4:296:78:421'})

Indices =

        3058
        3238
        3292
        3293
```

```
LogicalVal =

     0
     1
     0
     1

% Return only the last index for each key
LastIndices = Indices(LogicalVal)

LastIndices =

        3238
        3293
```

Tips

Use this method to determine the indices of specific entries with known keys.

See Also

`BioIndexedFile` | `getEntryByKey` | `getKeys` | `getSubset`

Topics

"Work with Next-Generation Sequencing Data"

getKeys

Class: `BioIndexedFile`

Retrieve alphanumeric keys from source file associated with BioIndexedFile object

Syntax

`Keys` `= getKeys(`*`BioIFobj`*`)`

Description

`Keys` `= getKeys(`*`BioIFobj`*`)` returns *`Keys`*, a cell array of character vectors specifying all the keys to the entries in the source file associated with *`BioIFobj`*, a BioIndexedFile object. The keys appear in the same order as they do in the source file, even if they are not unique.

Input Arguments

`BioIFobj`

Object of the `BioIndexedFile` class.

Default:

Output Arguments

`Keys`

Cell array of character vectors specifying all the keys to the entries in the source file. The keys appear in the same order as they do in the source file, even if they are not unique.

Examples

Construct a BioIndexedFile object to access a table containing cross-references between gene names and gene ontology (GO) terms:

```
% Create variable containing full absolute path of source file
sourcefile = which('yeastgenes.sgd');
% Create a BioIndexedFile object from the source file. Indicate
% the source file is a tab-delimited file where contiguous rows
% with the same key are considered a single entry. Store the
% index file in the Current Folder. Indicate that keys are
% located in column 3 and that header lines are prefaced with !
gene2goObj = BioIndexedFile('mrtab', sourcefile, '.', ...
                            'KeyColumn', 3, 'HeaderPrefix','!')
```

Retrieve all the keys for the entries in the source file, then view the first 12 keys:

```
% Retrieve all keys for entries in gene2goObj
keys = getKeys(gene2goObj);
```

```
% View the first 12 keys
keys(1:12)

ans =

    '15S_RRNA'
    '21S_RRNA'
    'AAC1'
    'AAC3'
    'AAD10'
    'AAD14'
    'AAD15'
    'AAD16'
    'AAD3'
    'AAD4'
    'AAD6'
    'AAH1'
```

Tips

Use this method to see a complete list of the alphanumeric keys, in the order they occur in the source file from which the BioIndexedFile object was created.

See Also

`BioIndexedFile` | `getEntryByKey` | `getIndexByKey` | `getSubset`

Topics

"Work with Next-Generation Sequencing Data"

getmatrix (biograph)

(Removed) Get connection matrix from biograph object

Note The function has been removed. Use `adjacency` instead.

Syntax

[*Matrix*, *ID*, *Distances*] = getmatrix(*BGObj*)

Arguments

BGObj	Biograph object created by `biograph` (object constructor).

Description

[*Matrix*, *ID*, *Distances*] = getmatrix(*BGObj*) converts the biograph object, *BiographObj*, into a logical sparse matrix, *Matrix*, in which 1 indicates that a node (row index) is connected to another node (column index). *ID* is a cell array of character vectors listing the `ID` properties for each node, and corresponds to the rows and columns of *Matrix*. *Distances* is a column vector with one entry for every nonzero entry in *Matrix* traversed column-wise and representing the respective `Weight` property for each edge.

Version History

Introduced in R2006b

R2022b: Removed

Errors starting in R2022b

The function has been removed. Use `adjacency` instead.

R2022a: Warns

Warns starting in R2022a

The function issues a warning that it will be removed in a future release.

R2021b: To be removed

Not recommended starting in R2021b

The function runs without warning, but it will be removed in a future release.

See Also

`adjacency` | `graph` | `digraph`

getmatrix

Convert `geneont` object into relationship matrix

Syntax

```
[Matrix] = getmatrix(GeneontObj)
[Matrix,ID] = getmatrix(GeneontObj)
[Matrix,ID,Relationship] = getmatrix(GeneontObj)
```

Description

`[Matrix] = getmatrix(GeneontObj)` converts a `geneont` object into a matrix of relationship values between nodes (row and column indices).

`[Matrix,ID] = getmatrix(GeneontObj)` also returns a column vector listing Gene Ontology IDs that correspond to the rows and columns of `Matrix`.

`[Matrix,ID,Relationship] = getmatrix(GeneontObj)` also returns a cell array of character vectors defining the types of relationships.

Examples

View Relationships in `geneont` Object

Download the current version of the Gene Ontology database from the Web into a `geneont` object.

```
GO = geneont('Live',true)
```

```
Gene Ontology object with 47266 Terms.
```

Retrieve the ancestors of the Gene Ontology term with an identifier of 46680.

```
ancestors = getancestors(GO,46680)
```

```
ancestors = 8×1

        8150
        9636
       10033
       14070
       17085
       42221
       46680
       50896
```

Create a subordinate Gene Ontology.

```
GeneontObj = GO(ancestors)
```

```
Gene Ontology object with 8 Terms.
```

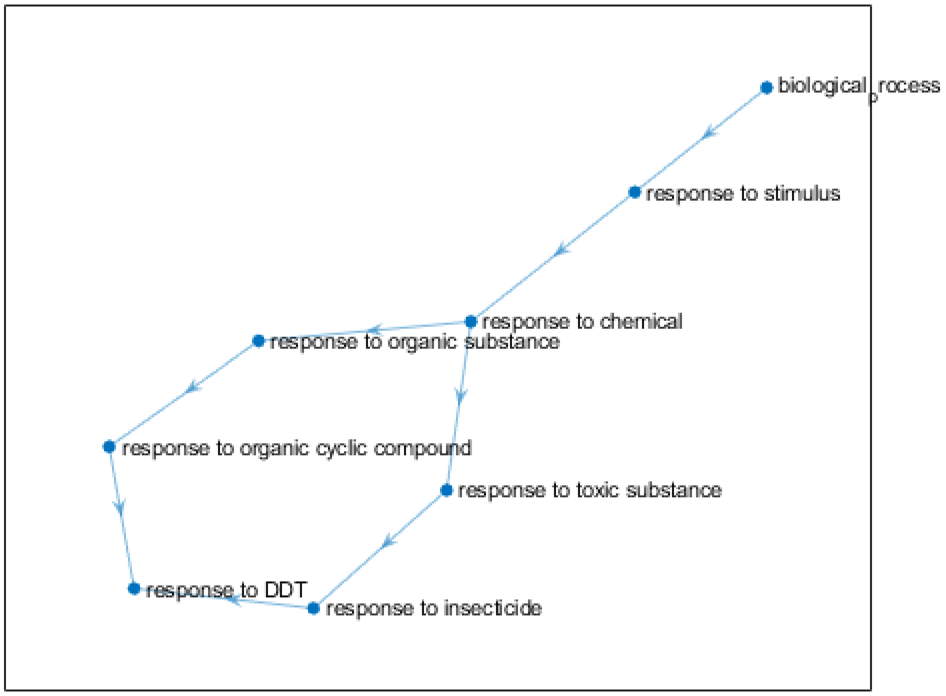
View the relationships in `GeneontObj`.

```
[Matrix,ID,Relationship] = getmatrix(GeneontObj)
```

```
Matrix =
   (6,2)        1
   (6,3)        1
   (3,4)        1
   (2,5)        1
   (8,6)        1
   (4,7)        1
   (5,7)        1
   (1,8)        1


ID = 8×1

         8150
         9636
        10033
        14070
        17085
        42221
        46680
        50896


Relationship = 1×6 cell
    {'is_a'}    {'negatively_regulates'}    {'part_of'}    {'positively_regulates'}    {'regulate
```

Plot the relationships.

```
BG = digraph(Matrix,get(GeneontObj.terms,"name"));
plot(BG,Layout="force")
```



Input Arguments

`GeneontObj` — Gene ontology object
output of `geneont`

Gene ontology object, specified as the output of the `geneont` command.

Example: `geneont('Live',true)`

Output Arguments

`Matrix` — Relationship values between nodes
sparse matrix

Relationship values between nodes (rows and columns) of `GeneontObj`, returned as a sparse matrix with the following entries:

- `0` — No relationship
- `1` — “is_a” relationship
- `2` — “part_of” relationship

`ID` — Gene ontology IDs
nonnegative integer column vector

Gene ontology IDs, returned as a nonnegative integer column vector. The entries in `ID` correspond to the entries in `Matrix`.

`Relationship` — Types of relationships
cell array of character vectors

Types of relationships, returned as a cell array of character vectors.

Version History

Introduced before R2006a

See Also

`geneont` | `goannotread` | `num2goid` | `term`

getmatrix (phytree)

Convert phytree object into relationship matrix

Syntax

[*Matrix*, *ID*, *Distances*] = getmatrix(*PhytreeObj*)

Arguments

PhytreeObj	phytree object created by `phytree` (object constructor).

Description

[*Matrix*, *ID*, *Distances*] = getmatrix(*PhytreeObj*) converts a phytree object, *PhytreeObj*, into a logical sparse matrix, *Matrix*, in which 1 indicates that a branch node (row index) is connected to its child (column index). The child can be either another branch node or a leaf node. *ID* is a column vector of strings listing the labels that correspond to the rows and columns of *Matrix*, with the labels from 1 to *Number of Leaves* being the leaf nodes, then the labels from *Number of Leaves* + 1 to *Number of Leaves* + *Number of Branches* being the branch nodes, and the label for the last branch node also being the root node. *Distances* is a column vector with one entry for every nonzero entry in *Matrix* traversed column-wise and representing the distance between the branch node and the child.

Examples

```
T = phytreeread('pf00002.tree')
[MATRIX, ID, DIST] = getmatrix(T);
```

Version History

Introduced in R2006b

See Also

`phytree` | `phytreeviewer` | `get` | `pdist` | `prune`

Topics

phytree object on page 1-1449

getweightmatrix (biograph)

(Removed) Get connection matrix with weights from biograph object

Note The function has been removed. Use `adjacency` instead.

Syntax

`[Matrix, ID] = getweightmatrix(BGObj)`

Arguments

BGObj	Biograph object created by `biograph` (object constructor).

Description

`[Matrix, ID] = getweightmatrix(BGObj)` converts the biograph object into a double sparse matrix, where non-zeros indicate the weight from the source node (row index) to the destination node (column index). *ID* is a list of the node's `'ID'` property and corresponds to the rows and columns of *Matrix*.

Version History

Introduced in R2006b

R2022b: Removed

Errors starting in R2022b

The function has been removed. Use `adjacency` instead.

R2022a: Warns

Warns starting in R2022a

The function issues a warning that it will be removed in a future release.

R2021b: To be removed

Not recommended starting in R2021b

The function runs without warning, but it will be removed in a future release.

See Also

`adjacency` | `graph` | `digraph`

getnewickstr (phytree)

Create Newick-formatted character vector

Syntax

```
nwk = getnewickstr(Tree)

getnewickstr(..., 'PropertyName', PropertyValue,...)
getnewickstr(..., 'Distances', DistancesValue)
getnewickstr(..., 'BranchNames', BranchNamesValue)
```

Arguments

`Tree`	Phytree object created with the function `phytree`.
`DistancesValue`	Property to control including or excluding distances in the output. Enter either `true` (include distances) or `false` (exclude distances). Default is `true`.
`BranchNamesValue`	Property to control including or excluding branch names in the output. Enter either `true` (include branch names) or `false` (exclude branch names). Default is `false`.

Description

`nwk = getnewickstr(Tree)` returns the Newick-formatted character vector of a phylogenetic tree object (`Tree`).

`getnewickstr(..., 'PropertyName', PropertyValue,...)` defines optional properties using property name/value pairs.

`getnewickstr(..., 'Distances', DistancesValue)`, when `DistancesValue` is `false`, excludes the distances from the output.

`getnewickstr(..., 'BranchNames', BranchNamesValue)`, when `BranchNamesValue` is `true`, includes the branch names in the output.

Examples

1 Create some random sequences.

```
seqs = int2nt(ceil(rand(10)*4));
```

2 Calculate pairwise distances.

```
dist = seqpdist(seqs,'alpha','nt');
```

3 Construct a phylogenetic tree.

```
tree = seqlinkage(dist);
```

4 Get the Newick-formatted character vector.

```
nwk  = getnewickstr(tree)
```

Version History
Introduced before R2006a

References

Information about the Newick tree format.

`https://evolution.genetics.washington.edu/phylip/newicktree.html`

See Also
`phytree` | `phytreeread` | `phytreeviewer` | `phytreewrite` | `seqlinkage` | `get` | `getbyname` | `getcanonical`

Topics
phytree object on page 1-1449

getnodesbyid (biograph)

(Removed) Get handles to nodes

Note The function has been removed. Use `findnode` instead.

Syntax

`*NodesHandles* = getnodesbyid(*BGobj*,*NodeIDs*)`

Arguments

BGobj	Biograph object.
NodeIDs	Enter a character vector or cell array of character vectors containing node identifications.

Description

`*NodesHandles* = getnodesbyid(*BGobj*,*NodeIDs*)` gets the handles for the specified nodes (*NodeIDs*) in a biograph object.

Version History

Introduced before R2006a

R2022b: Removed

Errors starting in R2022b

The function has been removed. Use `findnode` instead.

R2022a: Warns

Warns starting in R2022a

The function issues a warning that it will be removed in a future release.

R2021b: To be removed

Not recommended starting in R2021b

The function runs without warning, but it will be removed in a future release.

See Also

`findnode` | `graph` | `digraph`

getOptionsTable

Return table with all properties and equivalent options in original syntax

Syntax

`tbl = getOptionsTable(optionsObject)`

Description

`tbl = getOptionsTable(optionsObject)` returns a table with all the object properties and equivalent options in the original (native) syntax.

Examples

Retrieve Equivalent Original Options for Object Properties

Create a `CufflinksOptions` object.

Note `getOptionsTable` also works on other options objects. For a complete list of objects, see "optionsObject" on page 1-0 .

```
opt = CufflinksOptions;
```

Retrieve the equivalent original options for all object properties.

```
getOptionsTable(opt)
```

```
ans =

  35×3 table

                                         PropertyName                              FlagName
                                  ___________________________        _________________________________

    EffectiveLengthCorrection     'EffectiveLengthCorrection'        '--no-effective-length-correctio
    FauxReadTiling                'FauxReadTiling'                   '--no-faux-reads'
    FragmentBiasCorrection        'FragmentBiasCorrection'           '--frag-bias-correct'
    FragmentLengthMean            'FragmentLengthMean'               '--frag-len-mean'
    FragmentLengthSTD             'FragmentLengthSTD'                '--frag-len-std-dev'
    GTFGuide                      'GTFGuide'                         '--GTF-guide'
    IntronOverhangTolerance       'IntronOverhangTolerance'          '--intron-overhang-tolerance'
    JunctionAlpha                 'JunctionAlpha'                    '--junc-alpha'
    LengthCorrection              'LengthCorrection'                 '--no-length-correction'
    MaskFile                      'MaskFile'                         '--mask-file'
    MaxBundleFrags                'MaxBundleFrags'                   '--max-bundle-frags'
    MaxBundleLength               'MaxBundleLength'                  '--max-bundle-length'
    MaxFragAlignments             'MaxFragAlignments'                '--max-frag-multihits'
    MaxIntronLength               'MaxIntronLength'                  '--max-intron-length'
    MaxMLEIterations              'MaxMLEIterations'                 '--max-mle-iterations'
    MinFragsPerTransfrag        'MinFragsPerTransfrag'        '--min-frags-per-transfrag'
    MinIntronLength             'MinIntronLength'             '--min-intron-length'
    MinIsoformFraction          'MinIsoformFraction'          '--min-isoform-fraction'
    MultiReadCorrection         'MultiReadCorrection'         '--multi-read-correct'
    NormalizeCompatibleHits     'NormalizeCompatibleHits'     '--compatible-hits-norm'
    NormalizeTotalHits          'NormalizeTotalHits'          '--total-hits-norm'
    NumFragAssignmentSamples    'NumFragAssignmentSamples'    '--num-frag-assign-draws'
    NumFragSamples              'NumFragSamples'              '--num-frag-count-draws'
    NumThreads                  'NumThreads'                  '--num-threads'
    OutputDirectory             'OutputDirectory'             '--output-dir'
    OverhangTolerance           'OverhangTolerance'           '--overhang-tolerance'
    OverhangTolerance3          'OverhangTolerance3'          '--3-overhang-tolerance'
    OverlapRadius               'OverlapRadius'               '--overlap-radius'
    PreMRNAFraction             'PreMRNAFraction'             '--pre-mrna-fraction'
    ReferenceGTF                'ReferenceGTF'                '--GTF'
    Seed                        'Seed'                        '--seed'
    SmallAnchorFraction         'SmallAnchorFraction'         '--small-anchor-fraction'
    TranscriptPrefix            'TranscriptPrefix'            '--label'
    TrimCoverageThreshold       'TrimCoverageThreshold'       '--trim-3-avgcov-thresh'
    TrimDropoffFraction         'TrimDropoffFraction'         '--trim-3-dropoff-frac'
```

Input Arguments

`optionsObject` — Options
`BWAIndexOptions` | `BWAMEMOptions` | `CufflinksOptions` | `CuffCompareOptions` | `CuffMergeOptions` | `CuffQuantOptions` | `CuffDiffOptions` | `CuffNormOptions` | `CuffGFFReadOptions`

Options, specified as a `BWAIndexOptions`, `BWAMEMOptions`, `CufflinksOptions`, `CuffCompareOptions`, `CuffMergeOptions`, `CuffQuantOptions`, `CuffDiffOptions`, `CuffNormOptions`, or `CuffGFFReadOptions` object.

Output Arguments

`tbl` — Object properties and corresponding original options
table

Object properties and their corresponding original options, returned as a table.

Version History

Introduced in R2019a

References

[1] Trapnell, C., B. Williams, G. Pertea, A. Mortazavi, G. Kwan, J. van Baren, S. Salzberg, B. Wold, and L. Pachter. 2010. Transcript assembly and quantification by RNA-Seq reveals unannotated transcripts and isoform switching during cell differentiation. *Nature Biotechnology*. 28:511–515.

[2] Li, Heng, and Richard Durbin. “Fast and Accurate Short Read Alignment with Burrows-Wheeler Transform.” *Bioinformatics* 25, no. 14 (July 15, 2009): 1754–60. https://doi.org/10.1093/bioinformatics/btp324.

[3] Li, Heng, and Richard Durbin. “Fast and Accurate Long-Read Alignment with Burrows–Wheeler Transform.” *Bioinformatics* 26, no. 5 (March 1, 2010): 589–95. https://doi.org/10.1093/bioinformatics/btp698.

See Also

`BWAIndexOptions` | `BWAMEMOptions` | `CuffCompareOptions` | `CuffDiffOptions` | `CuffGFFReadOptions` | `CufflinksOptions` | `CuffMergeOptions` | `CuffNormOptions` | `CuffQuantOptions`

getpdb

Retrieve protein structure data from Protein Data Bank (PDB) database

Syntax

```
PDBStruct = getpdb(PDBid)

PDBStruct = getpdb(PDBid, ...'ToFile', ToFileValue, ...)
PDBStruct = getpdb(PDBid, ...'SequenceOnly', SequenceOnlyValue, ...)
PDBStruct = getpdb(PDBid, ...'TimeOut', TimeOutValue, ...)
```

Input Arguments

PDBid	Character vector or string specifying a unique identifier for a protein structure record in the PDB database. **Note** Each structure in the PDB database is represented by a four-character alphanumeric identifier. For example, `4hhb` is the identifier for hemoglobin.
ToFileValue	Character vector or string specifying a file name or a path and file name for saving the PDB-formatted data. If you specify only a file name, that file will be saved in the MATLAB Current Folder.
SequenceOnlyValue	Controls the return of the protein sequence only. Choices are `true` or `false` (default). If there is one sequence, it is returned as a character array. If there are multiple sequences, they are returned as a cell array.
TimeOutValue	Connection timeout in seconds, specified as a positive scalar. The default value is 5. For details, see here.

Output Arguments

PDBStruct	MATLAB structure containing a field for each PDB record.

Description

The Protein Data Bank (PDB) database is an archive of experimentally determined 3-D biological macromolecular structure data. `getpdb` retrieves protein structure data from the Protein Data Bank (PDB) database, which contains 3-D biological macromolecular structure data.

PDBStruct = getpdb(*PDBid*) searches the PDB database for the protein structure record specified by the identifier *PDBid* and returns the MATLAB structure *PDBStruct*, which contains a field for each PDB record. The following table summarizes the possible PDB records and the corresponding fields in the MATLAB structure *PDBStruct*:

PDB Database Record	Field in the MATLAB Structure
HEADER	Header
OBSLTE	Obsolete
TITLE	Title
CAVEAT	Caveat
COMPND	Compound
SOURCE	Source
KEYWDS	Keywords
EXPDTA	ExperimentData
AUTHOR	Authors
REVDAT	RevisionDate
SPRSDE	Superseded
JRNL	Journal
REMARK 1	Remark1
REMARK *N* **Note** *N* equals 2 through 999.	Remark*n* **Note** *n* equals 2 through 999.
DBREF	DBReferences
SEQADV	SequenceConflicts
SEQRES	Sequence
FTNOTE	Footnote
MODRES	ModifiedResidues
HET	Heterogen
HETNAM	HeterogenName
HETSYN	HeterogenSynonym
FORMUL	Formula
HELIX	Helix
SHEET	Sheet
TURN	Turn
SSBOND	SSBond
LINK	Link
HYDBND	HydrogenBond
SLTBRG	SaltBridge
CISPEP	CISPeptides
SITE	Site
CRYST1	Cryst1
ORIGXn	OriginX

PDB Database Record	Field in the MATLAB Structure
`SCALEn`	`Scale`
`MTRIXn`	`Matrix`
`TVECT`	`TranslationVector`
`MODEL`	`Model`
`ATOM`	`Atom`
`SIGATM`	`AtomSD`
`ANISOU`	`AnisotropicTemp`
`SIGUIJ`	`AnisotropicTempSD`
`TER`	`Terminal`
`HETATM`	`HeterogenAtom`
`CONECT`	`Connectivity`

`PDBStruct = getpdb(PDBid, ...'PropertyName', PropertyValue, ...)` calls `getpdb` with optional properties that use property name/property value pairs. You can specify one or more properties in any order. Each *`PropertyName`* must be enclosed in single quotation marks and is case insensitive. These property name/property value pairs are as follows:

`PDBStruct = getpdb(PDBid, ...'ToFile', ToFileValue, ...)` saves the data returned from the database to a PDB-formatted file, *`ToFileValue`*.

`PDBStruct = getpdb(PDBid, ...'SequenceOnly', SequenceOnlyValue, ...)` controls the return of the protein sequence only. Choices are `true` or `false` (default). If there is one sequence, it is returned as a character array. If there are multiple sequences, they are returned as a cell array.

`PDBStruct = getpdb(PDBid, ...'TimeOut', TimeOutValue, ...)` sets the connection timeout (in seconds) to retrieve data from the PDB database.

The Sequence Field

The `Sequence` field is also a structure containing sequence information in the following subfields:

- `NumOfResidues`
- `ChainID`
- `ResidueNames` — Contains the three-letter codes for the sequence residues.
- `Sequence` — Contains the single-letter codes for the sequence residues.

Note If the sequence has modified residues, then the `ResidueNames` subfield might not correspond to the standard three-letter amino acid codes. In this case, the `Sequence` subfield will contain the modified residue code in the position corresponding to the modified residue. The modified residue code is provided in the `ModifiedResidues` field.

The Model Field

The `Model` field is also a structure or an array of structures containing coordinate information. If the MATLAB structure contains one model, the `Model` field is a structure containing coordinate information for that model. If the MATLAB structure contains multiple models, the `Model` field is an array of structures containing coordinate information for each model. The `Model` field contains the following subfields:

- `Atom`
- `AtomSD`
- `AnisotropicTemp`
- `AnisotropicTempSD`
- `Terminal`
- `HeterogenAtom`

The Atom Field

The `Atom` field is also an array of structures containing the following subfields:

- `AtomSerNo`
- `AtomName`
- `altLoc`
- `resName`
- `chainID`
- `resSeq`
- `iCode`
- `X`
- `Y`
- `Z`
- `occupancy`
- `tempFactor`
- `segID`
- `element`
- `charge`
- `AtomNameStruct` — Contains three subfields: `chemSymbol`, `remoteInd`, and `branch`.

Examples

Retrieve the structure information for the electron transport (heme) protein that has a PDB identifier of `5CYT`, read the information into a MATLAB structure `pdbstruct`, and save the information to a PDB-formatted file `electron_transport.pdb` in the MATLAB Current Folder.

```
pdbstruct = getpdb('5CYT', 'ToFile', 'electron_transport.pdb')
```

Version History

Introduced before R2006a

See Also

getembl | getgenbank | getgenpept | pdbdistplot | pdbread | pdbsuperpose | pdbtransform | pdbwrite

getQuality

Retrieve sequence quality information from object

Syntax

```
quality = getQuality(object)
subsetQuality = getQuality(object,subset)
```

Description

`quality = getQuality(object)` returns sequence `quality` information from a `BioRead` or `BioMap` object.

`subsetQuality = getQuality(object,subset)` returns the sequence quality information `subsetQuality` for only the object elements specified by `subset`.

Examples

Retrieve Sequence Quality Information

Store read data from a SAM-formatted file in a BioRead object.

```
br = BioRead('ex1.sam')
```

```
br = 
  BioRead with properties:

     Quality: [1501x1 File indexed property]
    Sequence: [1501x1 File indexed property]
      Header: [1501x1 File indexed property]
       NSeqs: 1501
        Name: ''
```

Retrieve sequence quality information.

```
seqQuals = getQuality(br);
```

Retrieve sequence quality information from the first and third elements in the object.

```
seqQuals2 = getQuality(br,[1 3])
```

```
seqQuals2 = 2x1 cell
    {'<<<<<<<<<<<<<<<;<<<<<<<<<5<<<<<;:<;7'}
    {'<<<<<<<<<<<7;71<<;<;;<7;<<3;);3*8/5' }
```

Use a logical vector to get the same information.

```
seqQuals3 = getQuality(br,[true false true])
```

```
seqQuals3 = 2x1 cell
    {'<<<<<<<<<<<<<<<;<<<<<<<<<5<<<<<;:<;7'}
    {'<<<<<<<<<<<7;71<<;<;;<7;<<3;);3*8/5' }
```

You can use a header to get the quality of the corresponding sequence with that header. If multiple sequences have the same header, the function returns the quality information of all those sequences.

Get the quality information of the sequences with the header B7_591:4:96:693:509.

```
seqQuals4 = getQuality(br,{'B7_591:4:96:693:509'})
```

```
seqQuals4 = 1x1 cell array
    {'<<<<<<<<<<<<<<<;<<<<<<<<<5<<<<<;:<;7'}
```

Access each property of the object using the dot notation.

```
seqQuals = br.Quality;
seqQuals2   = br.Quality([1 3])
```

```
seqQuals2 = 2x1 cell
    {'<<<<<<<<<<<<<<<;<<<<<<<<<5<<<<<;:<;7'}
    {'<<<<<<<<<<<7;71<<;<;;<7;<<3;);3*8/5' }
```

Input Arguments

`object` — Object containing read data

`BioRead` object | `BioMap` object

Object containing the read data, specified as a `BioRead` or `BioMap` object.

Example: `bioreadObj`

`subset` — Subset of elements in object

vector of positive integers | logical vector | string vector | cell array of character vectors

Subset of elements in the object, specified as a vector of positive integers, logical vector, string vector, or cell array of character vectors containing valid sequence headers.

Example: `[1 3]`

Tip When you use a sequence header (or a cell array of headers) for `subset`, a repeated header specifies all elements with that header.

Output Arguments

`quality` — Sequence quality information

cell array of character vectors

Sequence quality information, returned as a cell array of character vectors. Each character is an ASCII-encoded value of the log probability of a base being incorrect.

`subsetQuality` — Sequence quality information for subset of elements
cell array of character vectors

Sequence quality information for a subset of elements from the object, returned as a cell array of character vectors.

Version History
Introduced in R2010a

See Also
`BioRead` | `BioMap`

Topics
"Manage Sequence Read Data in Objects"

getRange

Retrieve range of annotations from `GTFAnnotation` or `GFFAnnotation` object

Syntax

```
Range = getRange(AnnotObj)
```

Description

`Range = getRange(AnnotObj)` returns `Range`, a 1-by-2 numeric array specifying the minimum and maximum positions in the reference sequence covered by annotations in `AnnotObj`.

Examples

Retrieve Range of Annotations from a `GTFAnnotation` Object

Construct a `GTFAnnotation` object from a GTF-formatted file that is provided with Bioinformatics Toolbox™, and then return the range of the feature annotations.

Construct a `GTFAnnotation` object from a GTF file.

```
GTFAnnotObj = GTFAnnotation('hum37_2_1M.gtf');
```

Return first and last positions of reference associated with feature annotations.

```
range = getRange(GTFAnnotObj)
```

```
range = 1x2 int32 row vector

     41609   1371382
```

Retrieve Range of Annotations from a `GFFAnnotation` Object

Construct a `GFFAnnotation` object from a GFF-formatted file that is provided with Bioinformatics Toolbox™, and then return the range of the feature annotations.

Construct a `GFFAnnotation` object from a GFF file.

```
GFFAnnotObj = GFFAnnotation('tair8_1.gff');
```

Return first and last positions of reference associated with feature annotations.

```
range = getRange(GFFAnnotObj)
```

```
range = 1x2 int32 row vector

     3631   498516
```

Input Arguments

`AnnotObj` — Feature annotations
`GTFAnnotation` object | `GFFAnnotation` object

Feature annotations, specified as a `GTFAnnotation` or `GFFAnnotation` object.

Output Arguments

`Range` — Minimum and maximum positions in the reference sequence covered by annotations in `AnnotObj`
1-by-2 numeric array

Minimum and maximum positions in the reference sequence covered by annotations in `AnnotObj`, returned as a 1-by-2 numeric array.

Tips

- Use the `getSubset` method with the `Reference` name-value pair to return a `GFFAnnotation` object containing only one reference sequence. Then use this subsetted object as input to the `getRange` function.

Version History

Introduced in R2011b

See Also

`getData` | `getExons` | `getFeatureNames` | `getGeneNames` | `getGenes` | `getIndex` | `getReferenceNames` | `getSegments` | `getSubset` | `getTranscriptNames` | `getTranscripts` | `GTFAnnotation` | `GFFAnnotation`

Topics
"Store and Manage Feature Annotations in Objects"

External Websites
GTF2.2: A Gene Annotation Format
GFF (General Feature Format) specifications

getReferenceNames

Retrieve reference names from `GTFAnnotation` or `GFFAnnotation` object

Syntax

```
References = getReferenceNames(AnnotObj)
```

Description

`References = getReferenceNames(AnnotObj)` returns `References`, a cell array of character vectors specifying the names of all reference sequences in `AnnotObj`.

Examples

Retrieve Reference Names from a `GTFAnnotation` object

Construct a `GTFAnnotation` object from a GTF-formatted file that is provided with Bioinformatics Toolbox™, and then return the names of the reference sequences from the annotation object.

Construct a `GTFAnnotation` object from a GTF file.

```
GTFAnnotObj = GTFAnnotation('hum37_2_1M.gtf');
```

Return reference names for the annotation object.

```
refNames = getReferenceNames(GTFAnnotObj)
```

```
refNames = 1x1 cell array
    {'chr2'}
```

Retrieve Reference Names from a `GFFAnnotation` Object

Construct a `GFFAnnotation` object from a GFF-formatted file that is provided with Bioinformatics Toolbox™, and then return the names of the reference sequences from the annotation object.

Construct a `GFFAnnotation` object from a GFF file.

```
GFFAnnotObj = GFFAnnotation('tair8_1.gff');
```

Return reference names for the annotation object.

```
refNames = getReferenceNames(GFFAnnotObj)
```

```
refNames = 1x1 cell array
    {'Chr1'}
```

Input Arguments

`AnnotObj` — Feature annotations
`GTFAnnotation` object | `GFFAnnotation` object

Feature annotations, specified as a `GTFAnnotation` or `GFFAnnotation` object.

Output Arguments

`References` — Names of all reference sequences in `AnnotObj`
cell array of character vectors

Names of all reference sequences in `AnnotObj`, returned as a cell array of character vectors.

Version History

Introduced in R2011b

See Also

`getData` | `getExons` | `getFeatureNames` | `getGeneNames` | `getGenes` | `getIndex` | `getRange` | `getSegments` | `getSubset` | `getTranscriptNames` | `getTranscripts` | `GTFAnnotation` | `GFFAnnotation`

Topics
"Store and Manage Feature Annotations in Objects"

External Websites
GTF2.2: A Gene Annotation Format
GFF (General Feature Format) specifications

getrelatives (biograph)

(Removed) Find relatives of a node in biograph object

Note The function has been removed. Use `neighbors` instead. Note that the function does not let you specify `NumGenerations`. You can use `distances` to find the distances from all nodes to the target node after setting the edge weight to 1, and filter the nodes by distance to the appropriate depth.

Syntax

```
Nodes = getrelatives(BiographNode)
Nodes = getrelatives(BiographNode,NumGenerations)
```

Arguments

BiographNode	Node in a biograph object.
NumGenerations	Number of generations. Enter a positive integer.

Description

`Nodes` `= getrelatives(`*`BiographNode`*`)` finds all the direct relatives for a given node (*`BiographNode`*).

`Nodes` `= getrelatives(`*`BiographNode,NumGenerations`*`)` finds the direct relatives for a given node (*`BiographNode`*) up to a specified number of generations (*`NumGenerations`*). If the *`NumGenerations`* is `0`, the function returns the node itself.

Version History

Introduced before R2006a

R2022b: Removed

Errors starting in R2022b

The function has been removed. Use `neighbors` instead. Note that the function does not let you specify `NumGenerations`. You can use `distances` to find the distances from all nodes to the target node after setting the edge weight to 1, and filter the nodes by distance to the appropriate depth.

R2022a: Warns

Warns starting in R2022a

The function issues a warning that it will be removed in a future release.

R2021b: To be removed

Not recommended starting in R2021b

The function runs without warning, but it will be removed in a future release.

See Also

`graph` | `digraph` | `distances` | `neighbors`

getrelatives

Find terms that are relatives of specified Gene Ontology (GO) term

Syntax

```
RelativeIDs = getrelatives(GeneontObj,ID)
[RelativeIDs,Counts] = getrelatives(GeneontObj,ID)
___ = getrelatives(GeneontObj,ID,Name,Value)
```

Description

`RelativeIDs = getrelatives(GeneontObj,ID)` searches `GeneontObj`, a `geneont` object, for GO terms that are relatives of the GO term(s) specified by `ID`, which is a GO term identifier or vector of identifiers. The result `RelativeIDs` is a vector of GO term identifiers including `ID`.

`[RelativeIDs,Counts] = getrelatives(GeneontObj,ID)` also returns the number of times each relative is found.

Tip The `Counts` return value is useful when you tally counts in gene enrichment studies.

`___ = getrelatives(GeneontObj,ID,Name,Value)`, for any output arguments, specifies additional options using one or more name-value arguments. For example, you can restrict the search to have up to two levels up in the gene ontology by specifying `RelativeIDs = getrelatives(GeneontObj,ID,Height=2)`.

Examples

Search for Relatives in Gene Ontology

Download the current version of the Gene Ontology database from the Web into a `geneont` object.

```
GO = geneont('Live',true)
```

```
Gene Ontology object with 47266 Terms.
```

Retrieve the immediate relatives for the mitochondrial membrane GO term with a GO identifier of `31966`.

```
RelativeIDs = getrelatives(GO,31966,'levels',1)
```

```
RelativeIDs = 8×1

        5740
        5741
        5743
       31090
       31966
       44289
       90692
```

```
        98573
```

Create a subordinate Gene Ontology.

```
subontology = GO(RelativeIDs)
```

```
Gene Ontology object with 8 Terms.
```

Create a report of the subordinate Gene Ontology terms, that includes the GO identifier and name.

```
rpt = get(subontology.terms,{'id','name'})
```

```
rpt=8×2 cell array
    {[ 5740]}    {'mitochondrial envelope'                        }
    {[ 5741]}    {'mitochondrial outer membrane'                  }
    {[ 5743]}    {'mitochondrial inner membrane'                  }
    {[31090]}    {'organelle membrane'                            }
    {[31966]}    {'mitochondrial membrane'                        }
    {[44289]}    {'mitochondrial inner-outer membrane contact site'}
    {[90692]}    {'mitochondrial membrane scission site'          }
    {[98573]}    {'intrinsic component of mitochondrial membrane'  }
```

Retrieve all relatives for the mitochondrial outer membrane GO term with an identifier of `5741`.

```
RelativeIDs = getrelatives(GO,5741)
```

```
RelativeIDs = 7×1

         5741
        31306
        31315
        31966
        31968
        32473
        98799
```

Create a subordinate Gene Ontology.

```
subontology = GO(RelativeIDs)
```

```
Gene Ontology object with 7 Terms.
```

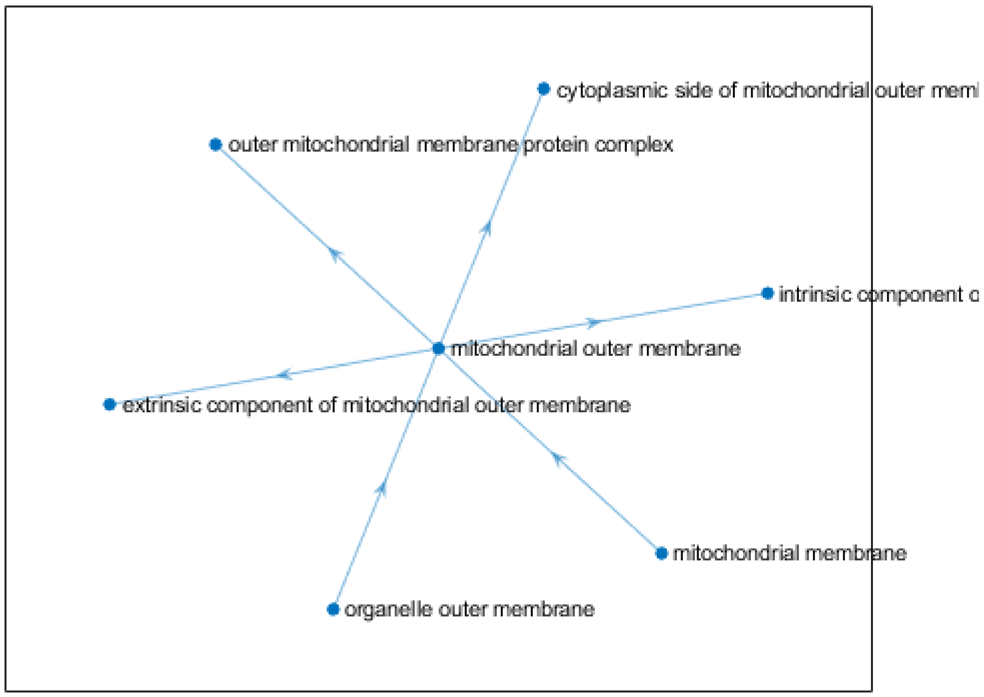
Create a report of the subordinate Gene Ontology terms.

```
rpt = get(subontology.terms,{'id','name'})
```

```
rpt=7×2 cell array
    {[ 5741]}    {'mitochondrial outer membrane'                        }
    {[31306]}    {'intrinsic component of mitochondrial outer membrane'}
    {[31315]}    {'extrinsic component of mitochondrial outer membrane'}
    {[31966]}    {'mitochondrial membrane'                              }
    {[31968]}    {'organelle outer membrane'                            }
    {[32473]}    {'cytoplasmic side of mitochondrial outer membrane'    }
    {[98799]}    {'outer mitochondrial membrane protein complex'        }
```

To view the relationships among the subordinate ontology, create a plot.

```
cm = getmatrix(subontology);
BG = digraph(cm,get(subontology.terms,"name"));
plot(BG,Layout="force")
```


cytoplasmic side of mitochondrial outer meml
outer mitochondrial membrane protein complex
intrinsic component o
mitochondrial outer membrane
extrinsic component of mitochondrial outer membrane
mitochondrial membrane
organelle outer membrane

Input Arguments

`GeneontObj` — Gene ontology object
output of `geneont`

Gene ontology object, specified as the output of the `geneont` command.

Example: `geneont('Live',true)`

`ID` — Gene ontology ID numbers
nonnegative integer vector

Gene ontology ID numbers, specified as a nonnegative integer vector.

Example: 5

Data Types: `single` | `double`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `RelativeIDs = getrelatives(GeneontObj,ID,Levels=1)`

`Height` — Number of search levels up in gene ontology
`Inf` (default) | positive integer

Number of search levels up in gene ontology, specified as a positive integer.

Example: 4

Data Types: `single` | `double`

`Depth` — Number of search levels in gene ontology
`Inf` (default) | positive integer

Number of search levels down in gene ontology, specified as a positive integer.

Example: 4

Data Types: `single` | `double`

`Levels` — Number of levels to search up and down in the gene ontology
`Inf` (default) | positive integer

Number of levels to search up and down in the gene ontology, specified as a positive integer. If you specify `Levels`, the value overrides any values in `Height` and `Depth`.

Example: 2

Data Types: `single` | `double`

`Relationtype` — Relations to match
`'both'` (default) | `'is_a'` | `'part_of'`

Relations to match, specified as one of the following:

- `'is_a'`
- `'part_of'`
- `'both'`

Example: `'is_a'`

Data Types: `char` | `string`

`Exclude` — Indication to exclude `ID` from `RelativeIDs`
`false` (default) | `true`

Indication to exclude `ID` from `RelativeIDs`, specified as `false` or `true`. This argument can fail to apply when `getrelatives` reaches the term while searching the ontology.

Example: `true`

Data Types: `logical`

Output Arguments

`RelativeIDs` — ID numbers of relatives of `GeneontObj`
nonnegative integer vector

ID numbers of relatives of `GeneontObj`, returned as a nonnegative integer vector. By default, `RelativeIDs` includes `ID`, unless you set the `Exclude` argument to `true`.

`Counts` — Number of times each relative is found
column vector of integers

Number of times each relative is found, returned as a column vector of integers. `Counts` has the same number of elements as terms in `GeneontObj`.

Version History
Introduced before R2006a

See Also
`geneont` | `goannotread` | `num2goid` | `term`

getSegments

Return table of non-overlapping segments from `GTFAnnotation` object

Syntax

```
segments = getSegments(AnnotObj)
[segments,transcriptIDs] = getSegments(AnnotObj)
[ ___ ] = getSegments(AnnotObj,"Reference",R)
[ ___ ] = getSegments(AnnotObj"Gene",G)
[ ___ ] = getSegments(AnnotObj,"Transcript",T)
```

Description

`segments = getSegments(AnnotObj)` returns `segments`, a table of non-overlapping segments of nucleotide sequences built by flattening the transcripts in `AnnotObj`. If an exon boundary is not the same in two or more transcripts of a gene, then the function creates two or more non-overlapping segments which cover all exons in the transcript.

`[segments,transcriptIDs] = getSegments(AnnotObj)` returns `transcriptIDs`, a cell array of character vectors containing all unique transcript IDs in `AnnotObj`.

`[ ___ ] = getSegments(AnnotObj,"Reference",R)` returns the segments that belong to one or more references specified by `R`.

`[ ___ ] = getSegments(AnnotObj"Gene",G)` returns the segments that belong to one of more genes specified by `G`.

`[ ___ ] = getSegments(AnnotObj,"Transcript",T)` returns the segments that belong to one or more transcripts specified by `T`.

Examples

Retrieve Segments from a GTF-formatted File

Create a `GTFAnnotation` object from a GTF-formatted file.

```
obj = GTFAnnotation('hum37_2_1M.gtf');
```

Retrieve unique reference names. In this case, there is only one reference sequence, which is chromosome 2 (`chr2`).

```
ref = getReferenceNames(obj)
```

```
ref = 1x1 cell array
    {'chr2'}
```

Get a table of all non-overlapping segments of nucleotide sequences which belong to `chr2`.

```
segments = getSegments(obj,"Reference",ref);
```

Input Arguments

`AnnotObj` — GTF annotation
`GTFAnnotation` object

GTF annotation, specified as a `GTFAnnotation` object.

`R` — Names of reference sequences
character vector | string | string vector | cell array of character vectors | categorical array

Names of reference sequences, specified as a character vector, string, string vector, cell array of character vectors, or categorical array.

The names must come from the `Reference` field of `AnnotObj`. If a name does not exist, the function provides a warning and ignores it.

Data Types: `char` | `string` | `cell` | `categorical`

`G` — Names of genes
character vector | string | string vector | cell array of character vectors | categorical array

Names of genes, specified as a character vector, string, string vector, cell array of character vectors, or categorical array.

The names must come from the `Gene` field of `AnnotObj`. If a name does not exist, the function provides a warning and ignores the name.

Data Types: `char` | `string` | `cell` | `categorical`

`T` — Names of transcripts
character vector | string | string vector | cell array of character vectors | categorical array

Names of transcripts, specified as a character vector, string, string vector, cell array of character vectors, or categorical array.

The names must come from the `Transcript` field of `AnnotObj`. If a name does not exist, the function gives a warning and ignores the name.

Data Types: `char` | `string` | `cell` | `categorical`

Output Arguments

`segments` — Non-overlapping segments
table

Non-overlapping segments, returned as a table. The table contains the following variables for each segments.

Variable Name	Description
`Start`	Start location of each segment.
`Stop`	Stop location of each segment.

Variable Name	Description
`Reference`	Categorical array representing the names of reference sequences to which the segments belong, obtained from the `Reference` field of `AnnotObj`.
`ExonIndicator`	Logical sparse matrix of segment versus exon. The rows represent segments. The columns are exons. If the *i*th segment is part of the *j*th exon, the element at position (*i*,*j*) is 1. Otherwise, it is 0.
`TranscriptIndicator`	Logical sparse matrix of segment versus transcript. The rows represent segments and the columns are transcripts. The element at position (*i*,*j*) is 1 if the *i*th segment is part of the *j*th transcript, and 0 otherwise.

`transcriptIDs` — Unique transcript IDs
cell array of character vectors

Unique transcript IDs, returned as a cell array of character vectors. The transcript IDs correspond to columns of the `TranscriptIndicator` variable of `segments`. For instance, the first element of `transcriptIDs` is the ID of the first column of `TranscriptIndicator` matrix.

Version History

Introduced in R2014b

See Also

`getData` | `getExons` | `getFeatureNames` | `getGeneNames` | `getGenes` | `getIndex` | `getRange` | `getReferenceNames` | `getSubset` | `getTranscriptNames` | `getTranscripts` | `GTFAnnotation` | `GFFAnnotation`

Topics
"Store and Manage Feature Annotations in Objects"

External Websites
GTF2.2: A Gene Annotation Format
GFF (General Feature Format) specifications

getSequence

Retrieve sequences from object

Syntax

```
seqs = getSequence(object)
subsetSeqs = getSequence(object,subset)
```

Description

`seqs = getSequence(object)` returns nucleotide sequences from a `BioRead` or `BioMap` object.

`subsetSeqs = getSequence(object,subset)` returns the sequences `subsetSeqs` for only the object elements specified by `subset`.

Examples

Retrieve Sequences from NGS Data

Store read data from a SAM-formatted file in a BioRead object.

```
br = BioRead('ex1.sam')
```

```
br = 
  BioRead with properties:

     Quality: [1501x1 File indexed property]
    Sequence: [1501x1 File indexed property]
      Header: [1501x1 File indexed property]
       NSeqs: 1501
        Name: ''
```

Retrieve the sequences (reads) from the object.

```
seqs = getSequence(br);
```

Retrieve the sequences from the first and third elements in the object.

```
seqs2 = getSequence(br,[1 3])
```

```
seqs2 = 2x1 cell
    {'CACTAGTGGCTCATTGTAAATGTGTGGTTTAACTCG'}
    {'AGTGGCTCATTGTAAATGTGTGGTTTAACTCGTCC' }
```

Use a logical vector to get the same information.

```
seqs3 = getSequence(br,[true false true])
```

```
seqs3 = 2x1 cell
    {'CACTAGTGGCTCATTGTAAATGTGTGGTTTAACTCG'}
    {'AGTGGCTCATTGTAAATGTGTGGTTTAACTCGTCC' }
```

You can use a header to get the corresponding sequences with that header. If multiple sequences have the same header, the function returns all those sequences.

Get the sequences with the header B7_591:4:96:693:509.

```
seqs4 = getSequence(br,{'B7_591:4:96:693:509'})
```

```
seqs4 = 1x1 cell array
    {'CACTAGTGGCTCATTGTAAATGTGTGGTTTAACTCG'}
```

Access each property of the object using the dot notation.

```
seqs = br.Sequence;
seq2   = br.Sequence([1 3])
```

```
seq2 = 2x1 cell
    {'CACTAGTGGCTCATTGTAAATGTGTGGTTTAACTCG'}
    {'AGTGGCTCATTGTAAATGTGTGGTTTAACTCGTCC' }
```

Input Arguments

`object` — Object containing read data
`BioRead` object | `BioMap` object

Object containing the read data, specified as a `BioRead` or `BioMap` object.

Example: `bioreadObj`

`subset` — Subset of elements in object
vector of positive integers | logical vector | string vector | cell array of character vectors

Subset of elements in the object, specified as a vector of positive integers, logical vector, string vector, or cell array of character vectors containing valid sequence headers.

Example: `[1 3]`

Tip When you use a sequence header (or a cell array of headers) for `subset`, a repeated header specifies all elements with that header.

Output Arguments

`seqs` — Nucleotide sequences
cell array of character vectors

Nucleotide sequences from the object, returned as a cell array of character vectors.

`subsetSeqs` — Nucleotide sequences from subset of elements
cell array of character vectors

Nucleotide sequences from a subset of elements from the object, returned as a cell array of character vectors.

Version History

Introduced in R2010a

See Also

`BioRead` | `BioMap`

Topics

“Manage Sequence Read Data in Objects”

getSubsequence

Retrieve partial sequences from object

Syntax

```
subSeqs = getSubsequence(object,subset,positions)
```

Description

`subSeqs = getSubsequence(object,subset,positions)` returns the partial sequences `subSeqs` for sequence positions specified by `positions` from only object elements specified by `subset`.

Examples

Retrieve Subsequences from NGS Data

Store read data from a SAM-formatted file in a BioRead object.

```
br = BioRead('ex1.sam')
```

```
br =
  BioRead with properties:

     Quality: [1501x1 File indexed property]
    Sequence: [1501x1 File indexed property]
      Header: [1501x1 File indexed property]
       NSeqs: 1501
        Name: ''
```

Retrieve the sequences (reads) from the object.

```
seqs = getSequence(br);
```

Retrieve the first, third, and fifth sequences from the object.

```
seqs2 = getSequence(br,[1 3 5])
```

```
seqs2 = 3x1 cell
    {'CACTAGTGGCTCATTGTAAATGTGTGGTTTAACTCG'}
    {'AGTGGCTCATTGTAAATGTGTGGTTTAACTCGTCC' }
    {'GCTCATTGTAAATGTGTGGTTTAACTCGTCCATGG' }
```

Retrieve the first five positions of those sequences.

```
seqs3 = getSubsequence(br,[1 3 5],[1:5])
```

```
seqs3 = 3x1 cell
    {'CACTA'}
    {'AGTGG'}
    {'GCTCA'}
```

You can use a header to get the corresponding sequences with that header. If multiple sequences have the same header, the function returns all of those sequences.

Get the first five positions of the sequences with the header B7_591:4:96:693:509.

```
seqs4 = getSubsequence(br,{'B7_591:4:96:693:509'},[1:5])
```

```
seqs4 = 1x1 cell array
    {'CACTA'}
```

Retrieve the first, fourth, and sixth positions of the first three sequences.

```
seq5 = getSubsequence(br,[1:3],[1 4 6])
```

```
seq5 = 3x1 cell
    {'CTG'}
    {'CGG'}
    {'AGC'}
```

Input Arguments

`object` — Object containing read data
`BioRead` object | `BioMap` object

Object containing the read data, specified as a `BioRead` or `BioMap` object.

Example: `bioreadObj`

`subset` — Subset of elements in object
vector of positive integers | logical vector | string vector | cell array of character vectors

Subset of elements in the object, specified as a vector of positive integers, logical vector, string vector, or cell array of character vectors containing valid sequence headers.

Example: `[1 3]`

Tip When you use a sequence header (or a cell array of headers) for `subset`, a repeated header specifies all elements with that header.

`positions` — Sequence positions
vector of positive integers | logical vector

Sequence positions, specified as a vector of positive integers or logical vector. The last position must be within the range of positions for each sequence specified by `subset`.

Example: `[2:10]`

Output Arguments

`subSeqs` — Subsequences from subset of elements
cell array of character vectors

Subsequences from a subset of elements, returned as a cell array of character vectors.

Version History

Introduced in R2010a

See Also

`BioRead` | `BioMap`

Topics
"Manage Sequence Read Data in Objects"

getSubset

Class: `BioIndexedFile`

Create object containing subset of elements from BioIndexedFile object

Syntax

```
NewObj = getSubset(BioIFObj, Indices)
NewObj = getSubset(BioIFObj, Keys)
```

Description

`NewObj` `= getSubset(`*`BioIFObj`*`,` *`Indices`*`)` returns *`NewObj`*, a new BioIndexedFile object that accesses a subset of entries in the source file associated with *`BioIFObj`*, a BioIndexedFile object. The entries are specified by *`Indices`*, a vector containing unique positive integers.

`NewObj` `= getSubset(`*`BioIFObj`*`,` *`Keys`*`)` returns *`NewObj`*, a new BioIndexedFile object that accesses a subset of entries in the source file associated with *`BioIFObj`*, a BioIndexedFile object. The entries are specified by *`Keys`*, a character vector or cell array of unique character vectors specifying keys.

Input Arguments

BioIFObj

Object of the `BioIndexedFile` class.

Default:

Indices

Vector containing unique positive integers that specify the entries in the source file to access with *`NewObj`*. The number of elements in *`Indices`* cannot exceed the number of entries indexed by *`BioIFObj`*. There is a one-to-one relationship between the elements in *`Indices`* and the entries that *`NewObj`* accesses.

Keys

Character vector or cell array of unique character vectors specifying keys that specify the entries in the source file to access with *`NewObj`*. The number of elements in *`Keys`* is less than or equal to the number of entries indexed by *`BioIFObj`*. If the keys in the source file are not unique, then all entries that match a given key are indexed by *`NewObj`*. In this case, there is not a one-to-one relationship between the elements in *`Keys`* and the entries that *`NewObj`* accesses. If the keys in the source file are unique, then there is a one-to-one relationship between the elements in *`Keys`* and the entries that *`NewObj`* accesses.

Output Arguments

`NewObj`

Object of the `BioIndexedFile` class.

Examples

Construct a BioIndexedFile object to access a table containing cross-references between gene names and gene ontology (GO) terms:

```
% Create a variable containing the full absolute path of the source file.
sourcefile = which('yeastgenes.sgd');
% Create a BioIndexedFile object from the source file. Indicate
% the source file is a tab-delimited file where contiguous rows
% with the same key are considered a single entry. Store the
% index file in the Current Folder. Indicate that keys are
% located in column 3 and that header lines are prefaced with !
gene2goObj = BioIndexedFile('mrtab', sourcefile, '.', ...
                            'KeyColumn', 3, 'HeaderPrefix','!')
```

Create a new BioIndexedFile object that accesses only the first 1,000 cross-references and reuses the same index file as `gene2goObj`:

```
% Create a new BioIndexedFile object.
gene2goSubset = getSubset(gene2goObj,1:1000);
```

Tips

Use this method to create a smaller, more manageable BioIndexedFile object.

See Also

`BioIndexedFile` | `getEntryByKey` | `getIndexByKey` | `getEntryByIndex` | `getKeys` | `read`

Topics

"Work with Next-Generation Sequencing Data"

getSubset

Retrieve subset of elements from object

Syntax

```
subset = getSubset(object,subset)
subset = getSubset(object,subset,Name,Value)
```

Description

`subset = getSubset(object,subset)` returns the sequence read data `subset` for only the object elements specified by `subset`.

`subset = getSubset(object,subset,Name,Value)` uses additional options specified by one or more name-value pair arguments. For example, you can specify whether to keep the data in memory.

Examples

Retrieve Subset of Elements from NGS Data

Store read data from a SAM-formatted file in a `BioRead` object. By default, the data remains in the source file, and `BioRead` uses an index file to access the data, making the process more memory efficient.

```
br = BioRead('ex1.sam')
```

```
br = 
  BioRead with properties:

     Quality: [1501x1 File indexed property]
    Sequence: [1501x1 File indexed property]
      Header: [1501x1 File indexed property]
       NSeqs: 1501
        Name: ''
```

Set the `'InMemory'` name-value pair argument to `true` to store the data in memory, enabling you to access the data faster and edit the properties of the object.

```
brInMemory = BioRead('ex1.sam','InMemory',true)
```

```
brInMemory = 
  BioRead with properties:

     Quality: {1501x1 cell}
    Sequence: {1501x1 cell}
      Header: {1501x1 cell}
       NSeqs: 1501
        Name: ''
```

Retrieve the second and third elements from the object `br`. By default, the resulting object `subset` is not placed in memory if the parent object `br` is not in memory. If `br` is already in memory, the resulting subset is placed in memory.

```
subset = getSubset(br,[2 3])
```

```
subset = 
  BioRead with properties:

     Quality: [2x1 File indexed property]
    Sequence: [2x1 File indexed property]
      Header: [2x1 File indexed property]
       NSeqs: 2
        Name: ''
```

Alternatively, you can keep the parent object `br` in the source file, and load the resulting subset in memory if the subset is small enough. You access the subset faster and update it as needed.

```
subsetInMemory = getSubset(br,[2 3],'InMemory',true)
```

```
subsetInMemory = 
  BioRead with properties:

     Quality: {2x1 cell}
    Sequence: {2x1 cell}
      Header: {2x1 cell}
       NSeqs: 2
        Name: ''
```

Update the header information of the first element.

```
subsetInMemory.Header(1)
```

```
ans = 1x1 cell array
    {'EAS54_65:7:152:368:113'}
```

```
subsetInMemory.Header(1) = {'NewHeader'};
subsetInMemory.Header(1)
```

```
ans = 1x1 cell array
    {'NewHeader'}
```

You can use a header to get the corresponding elements with that header. If multiple elements have the same header, the function returns all those elements.

Get all the elements with the header `'B7_591:4:96:693:509'` from the `br` object stored in memory.

```
subset2 = getSubset(brInMemory,{'B7_591:4:96:693:509'})
```

```
subset2 = 
  BioRead with properties:

     Quality: {'<<<<<<<<<<<<<<<;<<<<<<<<<5<<<<<;:<;7'}
    Sequence: {'CACTAGTGGCTCATTGTAAATGTGTGGTTTAACTCG'}
```

```
      Header: {'B7_591:4:96:693:509'}
       NSeqs: 1
        Name: ''
```

Input Arguments

`object` — Object containing read data
`BioRead` object | `BioMap` object

Object containing the read data, specified as a `BioRead` or `BioMap` object.

Example: `bioreadObj`

`subset` — Subset of elements in object
vector of positive integers | logical vector | string vector | cell array of character vectors

Subset of elements in the object, specified as a vector of positive integers, logical vector, string vector, or cell array of character vectors containing valid sequence headers.

Example: `[1 3]`

Tip When you use a sequence header (or a cell array of headers) for `subset`, a repeated header specifies all elements with that header.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `'InMemory',true` specifies to save the output object (`subset`) in memory.

`Name` — Name of object
`''` (default) | character vector | string

Name of the object, specified as the comma-separated pair consisting of `'Name'` and a character vector or string. The default is an empty character vector `''` (no name).

Example: `'Name','newData'`

`InMemory` — Logical flag to keep data in memory
`false` (default) | `true`

Logical flag to keep data in memory, specified as the comma-separated pair consisting of `'InMemory'` and `true` or `false`. Keeping the data in memory lets you access the resulting object `subset` faster and update its properties. If the data specified for `subset` is still large and does not fit in memory, set this name-value pair to `false` to use indexed access, which is more memory efficient but does not enable you to modify the properties.

If the parent `object` is already in memory, the resulting object `subset` is automatically placed in memory, and the function ignores this argument.

Example: `'InMemory',true`

`SelectReference` — References used to create subset of data
cell array of character vectors | string vector | vector of positive integers

References used to create the `subset` of data with only the reads mapped to those references, specified as the comma-separated pair consisting of `'SelectReference'` and a cell array of character vectors, string vector, or vector of positive integers.

Note This argument is for the `BioMap` objects only.

Example: `'SelectReference',{'RefSeq1'}`

Output Arguments

`subset` — Subset of elements
`BioRead` object | `BioMap` object

Subset of elements from the `object`, returned as a `BioRead` or `BioMap` object. If `object` is in memory, then `subset` is placed in memory. If `object` is indexed, then `subset` is indexed unless you set `'InMemory'` to `true`.

Version History

Introduced in R2010a

See Also

`BioRead` | `BioMap`

Topics

"Manage Sequence Read Data in Objects"

getSubset

Retrieve subset of elements from `GTFAnnotation` or `GFFAnnotation` object

Syntax

```
NewObj = getSubset(AnnotObj,StartPos,EndPos)
NewObj = getSubset(AnnotObj,Subset)
NewObj = getSubset( ___ ,Name,Value)
```

Description

`NewObj = getSubset(AnnotObj,StartPos,EndPos)` returns `NewObj`, a new object containing a subset of the elements from `AnnotObj` that falls within each reference sequence range specified by `StartPos` and `EndPos`.

`NewObj = getSubset(AnnotObj,Subset)` returns `NewObj`, a new object containing a subset of elements specified by `Subset`, a vector of integers.

`NewObj = getSubset( ___ ,Name,Value)` returns `NewObj`, a new object containing a subset of the elements from `AnnotObj`, using any of the input arguments from the previous syntaxes and additional options specified by one or more `Name,Value` pair arguments.

Examples

Retrieve Subsets of Data from a GTFAnnotation Object

Construct a `GTFAnnotation` object using a GTF-formatted file that is provided with Bioinformatics Toolbox™.

```
GTFAnnotObj = GTFAnnotation('hum37_2_1M.gtf');
```

Retrieve a subset of data from the first to fifth elements of `GTFAnnotObj`.

```
subsetGTF1 = getSubset(GTFAnnotObj,1:5)
```

```
subsetGTF1 = 
  GTFAnnotation with properties:

    FieldNames: {'Reference'  'Start'  'Stop'  'Feature'  'Gene'  'Transcript'  'Source'  'Score
    NumEntries: 5
```

Retrieve only the first, fifth and eighth elements of `GTFAnnotObj`.

```
subsetGTF2 = getSubset(GTFAnnotObj,[1 5 8])
```

```
subsetGTF2 = 
  GTFAnnotation with properties:

    FieldNames: {'Reference'  'Start'  'Stop'  'Feature'  'Gene'  'Transcript'  'Source'  'Score
    NumEntries: 3
```

Create a Subset of Data Containing Only CDS Features from a GTF-formatted File

Construct a `GTFAnnotation` object using a GTF-formatted file that is provided with Bioinformatics Toolbox™.

```
GTFAnnotObj = GTFAnnotation('hum37_2_1M.gtf');
```

Create a subset of the data containing only CDS features.

```
subsetGTF = getSubset(GTFAnnotObj,"Feature","CDS")
```

```
subsetGTF = 
  GTFAnnotation with properties:

    FieldNames: {'Reference'  'Start'  'Stop'  'Feature'  'Gene'  'Transcript'  'Source'  'Score
    NumEntries: 92
```

Retrieve Subsets of Data from a GFFAnnotation Object

Construct a `GFFAnnotation` object using a GFF-formatted file that is provided with Bioinformatics Toolbox™.

```
GFFAnnotObj = GFFAnnotation('tair8_1.gff');
```

Retrieve a subset of data from the first to fifth elements of `GFFAnnotObj`.

```
subsetGFF2 = getSubset(GFFAnnotObj,1:5)
```

```
subsetGFF2 = 
  GFFAnnotation with properties:

    FieldNames: {'Reference'  'Start'  'Stop'  'Feature'  'Source'  'Score'  'Strand'  'Frame'
    NumEntries: 5
```

Retrieve only the first, fifth, and eighth elements of `GFFAnnotObj`.

```
subsetGFF3 = getSubset(GFFAnnotObj,[1 5 8])
```

```
subsetGFF3 = 
  GFFAnnotation with properties:

    FieldNames: {'Reference'  'Start'  'Stop'  'Feature'  'Source'  'Score'  'Strand'  'Frame'
    NumEntries: 3
```

Create a Subset of Data Containing Only Protein Features from a GFF-formatted File

Construct a `GFFAnnotation` object using a GFF-formatted file that is provided with Bioinformatics Toolbox™.

```
GFFAnnotObj = GFFAnnotation('tair8_1.gff');
```

Create a subset of data containing only protein features.

```
subsetGFF1 = getSubset(GFFAnnotObj,"Feature","protein")
```

```
subsetGFF1 = 
  GFFAnnotation with properties:

    FieldNames: {'Reference'  'Start'  'Stop'  'Feature'  'Source'  'Score'  'Strand'  'Frame'
    NumEntries: 200
```

Input Arguments

`AnnotObj` — Feature annotations
`GTFAnnotation` object | `GFFAnnotation` object

Feature annotations, specified as a `GTFAnnotation` or `GFFAnnotation` object.

`StartPos` — Start of a range in each reference sequence in `AnnotObj`
nonnegative integer

Start of a range in each reference sequence in `AnnotObj`, specified as a nonnegative integer less than or equal to `EndPos`.

Data Types: `double`

`EndPos` — End of a range in each reference sequence in `AnnotObj`
nonnegative integer

End of a range in each reference sequence in `AnnotObj`, specified as a nonnegative integer greater than or equal to `StartPos`.

Data Types: `double`

`Subset` — Subset of data from `AnnotObj` to retrieve
vector of positive integers

Subset of data from `AnnotObj` to retrieve, specified as a vector of positive integers. Each integer must be less than or equal to the number of entries in the object.

Data Types: `double`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `NewObj = getSubset(AnnotObj,"Feature","CDS")`

`Reference` — One or more reference sequences in `AnnotObj`
character vector | string | string vector | cell array of character vectors

One or more reference sequences in `AnnotObj`, specified as a character vector, string, string vector, or cell array of character vectors. Only annotations whose reference field matches one of the character vectors or strings are included in `NewObj`.

Data Types: `char` | `string` | `cell`

`Feature` — One or more features in `AnnotObj`
character vector | string | string vector | cell array of character vectors

One or more features in `AnnotObj`, specified as a character vector, string, string vector, or cell array of character vectors. Only annotations whose feature field matches one of the character vectors or strings are included in `NewObj`.

Data Types: `char` | `string` | `cell`

`Gene` — One or more genes in `AnnotObj` of type `GTFAnnotation`
character vector | string | string vector | cell array of character vectors

One or more genes in `AnnotObj` of type `GTFAnnotation`, specified as a character vector, string, string vector, or cell array of character vectors. Only annotations whose gene field matches one of the character vectors or strings are included in `NewObj`.

Data Types: `char` | `string` | `cell`

`Transcript` — One or more transcripts in `AnnotObj` of type `GTFAnnotation`
character vector | string | string vector | cell array of character vectors

One or more transcripts in `AnnotObj` of type `GTFAnnotation`, specified as a character vector, string, string vector, or cell array of character vectors. Only annotations whose transcript field matches one of the character vectors or strings are included in `NewObj`.

Data Types: `char` | `string` | `cell`

`Overlap` — Minimum number of base positions that annotation must overlap in the range
1 (default) | positive integer | `"full"` | `"start"`

Minimum number of base positions that annotation must overlap in the range, to be included in `NewObj`, specified as a positive integer, `"full"` or `"start"`. Use `"full"` when an annotation must be fully contained in the range to be included. Use `"start"` when an annotation’s start position must lie within the range to be included.

Data Types: `double` | `char` | `string`

Output Arguments

`NewObj` — Subset of feature annotations
`GTFAnnotation` object | `GFFAnnotation` object

Subset of feature annotations, returned as a `GTFAnnotation` or `GFFAnnotation` object.

Tips

- The `getSubset` function selects annotations from the range specified by `StartPos` and `EndPos` for each reference sequence in `AnnotObj` unless you use the `Reference` name-value pair argument to limit the reference sequences.
- After creating a subsetted object, you can access the number of entries, range of reference sequences covered by annotations, field names, and reference names. To access the values of all fields, create a structure of the data using the `getData` function.

Version History

Introduced in R2013a

See Also

`getData` | `getExons` | `getFeatureNames` | `getGeneNames` | `getGenes` | `getIndex` | `getRange` | `getReferenceNames` | `getSegments` | `getTranscriptNames` | `getTranscripts` | `GTFAnnotation` | `GFFAnnotation`

Topics

"Store and Manage Feature Annotations in Objects"

External Websites

GTF2.2: A Gene Annotation Format
GFF (General Feature Format) specifications

getTranscripts

Return table of unique transcripts in `GTFAnnotation` object

Syntax

```
transcriptsTable = getTranscripts(AnnotObj)
transcriptsTable = getTranscripts(AnnotObj,"Reference",R)
transcriptsTable = getTranscripts(AnnotObj,"Gene",G)
transcriptsTable = getTranscripts(AnnotObj,"Transcript",T)
```

Description

`transcriptsTable = getTranscripts(AnnotObj)` returns `transcriptsTable`, a table of transcripts referenced by exons in `AnnotObj`.

`transcriptsTable = getTranscripts(AnnotObj,"Reference",R)` returns one or more transcripts that belong to the references specified by `R`.

`transcriptsTable = getTranscripts(AnnotObj,"Gene",G)` returns one or more transcripts that belong to the genes specified by `G`.

`transcriptsTable = getTranscripts(AnnotObj,"Transcript",T)` returns one or more transcripts specified by `T`.

Examples

Retrieve Transcripts from a GTF-formatted File

Create a `GTFAnnotation` object from a GTF-formatted file.

```
obj = GTFAnnotation('hum37_2_1M.gtf');
```

Get the list of gene names listed in the object.

```
gNames = getGeneNames(obj)
```

```
gNames = 28x1 cell
    {'uc002qvu.2'}
    {'uc002qvv.2'}
    {'uc002qvw.2'}
    {'uc002qvx.2'}
    {'uc002qvy.2'}
    {'uc002qvz.2'}
    {'uc002qwa.2'}
    {'uc002qwb.2'}
    {'uc002qwc.1'}
    {'uc002qwd.2'}
    {'uc002qwe.3'}
    {'uc002qwf.2'}
    {'uc002qwg.2'}
    {'uc002qwh.2'}
    {'uc002qwi.3'}
    {'uc002qwk.2'}
    {'uc002qwl.2'}
    {'uc002qwm.1'}
    {'uc002qwn.1'}
    {'uc002qwo.1'}
    {'uc002qwp.2'}
    {'uc002qwq.2'}
    {'uc010ewe.2'}
    {'uc010ewf.1'}
    {'uc010ewg.2'}
    {'uc010ewh.1'}
    {'uc010ewi.2'}
    {'uc010yim.1'}
```

Get a table of transcripts which belong to the first gene `uc002qvu.2`.

```
transcripts = getTranscripts(obj,"Gene",gNames{1})
```

```
transcripts=1×7 table
      Transcript        GeneName          GeneID          Reference    Start       Stop      Strand
    ______________    __________    ______________    _________    ______    ______    ______

    {'uc002qvu.2'}    {0x0 char}    {'uc002qvu.2'}      chr2       218138    249852      -
```

Input Arguments

`AnnotObj` — GTF annotation
`GTFAnnotation` object

GTF annotation, specified as a `GTFAnnotation` object.

R — Names of reference sequences
character vector | string | string vector | cell array of character vectors | categorical array

Names of reference sequences, specified as a character vector, string, string vector, cell array of character vectors, or categorical array.

The names must come from the `Reference` field of `AnnotObj`. If a name does not exist, the function provides a warning and ignores it.

Data Types: `char` | `string` | `cell` | `categorical`

G — Names of genes
character vector | string | string vector | cell array of character vectors | categorical array

Names of genes, specified as a character vector, string, string vector, cell array of character vectors, or categorical array.

The names must come from the `Gene` field of `AnnotObj`. If a name does not exist, the function provides a warning and ignores the name.

Data Types: `char` | `string` | `cell` | `categorical`

T — Names of transcripts
character vector | string | string vector | cell array of character vectors | categorical array

Names of transcripts, specified as a character vector, string, string vector, cell array of character vectors, or categorical array.

The names must come from the `Transcript` field of `AnnotObj`. If a name does not exist, the function gives a warning and ignores the name.

Data Types: `char` | `string` | `cell` | `categorical`

Output Arguments

`transcriptsTable` — Transcripts
table

Transcripts, returned as a table. The table contains the following variables for each transcript.

Variable Name	Description
`Transcript`	Cell array of character vectors containing transcript IDs, obtained from the `Transcript` field of `AnnotObj`.
`GeneName`	Cell array of character vectors containing the names of expressed genes, obtained from the `Attributes` field of `AnnotObj`. This cell array can contain empty character vectors if the corresponding gene names are not found in `Attributes`.
`GeneID`	Cell array of character vectors containing the expressed gene IDs, obtained from the `Gene` field of `AnnotObj`.
`Reference`	Categorical array representing the names of reference sequences to which the expressed genes belong. The reference names are from the `Reference` field of `AnnotObj`.
`Start`	Start location of the first exon in each transcript.
`Stop`	Stop location of the last exon in each transcript.
`Strand`	Categorical array containing the strand of expressed gene.

Version History

Introduced in R2014b

See Also

`getData` | `getExons` | `getFeatureNames` | `getGeneNames` | `getGenes` | `getIndex` | `getRange` | `getReferenceNames` | `getSegments` | `getSubset` | `getTranscriptNames` | `GTFAnnotation` | `GFFAnnotation`

Topics
"Store and Manage Feature Annotations in Objects"

External Websites
GTF2.2: A Gene Annotation Format
GFF (General Feature Format) specifications

getTranscriptNames

Retrieve unique transcript names from `GTFAnnotation` object

Syntax

```
Transcripts = getTranscriptNames(AnnotObj)
```

Description

`Transcripts = getTranscriptNames(AnnotObj)` returns `Transcripts`, a cell array of character vectors specifying the unique transcript names associated with annotations in `AnnotObj`.

Examples

Retrieve Transcript Names from a `GTFAnnotation` Object

Construct a `GTFAnnotation` object from a GTF-formatted file that is provided with Bioinformatics Toolbox™, and then retrieve a list of the unique transcript names from the object.

Construct a `GTFAnnotation` object from a GTF file.

```
GTFAnnotObj = GTFAnnotation('hum37_2_1M.gtf');
```

Get transcript names from object.

```
transcriptNames = getTranscriptNames(GTFAnnotObj)
```

```
transcriptNames = 28x1 cell
    {'uc002qvu.2'}
    {'uc002qvv.2'}
    {'uc002qvw.2'}
    {'uc002qvx.2'}
    {'uc002qvy.2'}
    {'uc002qvz.2'}
    {'uc002qwa.2'}
    {'uc002qwb.2'}
    {'uc002qwc.1'}
    {'uc002qwd.2'}
    {'uc002qwe.3'}
    {'uc002qwf.2'}
    {'uc002qwg.2'}
    {'uc002qwh.2'}
    {'uc002qwi.3'}
    {'uc002qwk.2'}
    {'uc002qwl.2'}
    {'uc002qwm.1'}
    {'uc002qwn.1'}
    {'uc002qwo.1'}
    {'uc002qwp.2'}
    {'uc002qwq.2'}
    {'uc010ewe.2'}
    {'uc010ewf.1'}
    {'uc010ewg.2'}
    {'uc010ewh.1'}
    {'uc010ewi.2'}
    {'uc010yim.1'}
```

Input Arguments

`AnnotObj` — GTF annotation
`GTFAnnotation` object

GTF annotation, specified as a `GTFAnnotation` object.

Output Arguments

`Transcripts` — Unique transcript names associated with annotations in `AnnotObj`
cell array of character vectors

Unique transcript names associated with annotations in `AnnotObj`, specified as a cell array of character vectors.

Version History

Introduced in R2011b

See Also

`getData` | `getExons` | `getFeatureNames` | `getGeneNames` | `getGenes` | `getIndex` | `getRange` | `getReferenceNames` | `getSegments` | `getSubset` | `getTranscripts` | `GTFAnnotation` | `GFFAnnotation`

Topics

"Store and Manage Feature Annotations in Objects"

External Websites

GTF2.2: A Gene Annotation Format
GFF (General Feature Format) specifications

GFFAnnotation

Contain General Feature Format (GFF) annotations

Description

The `GFFAnnotation` object contains annotations for one or more reference sequences, conforming to the GFF file format.

Each element in the object represents an annotation. Use the object properties and methods to filter annotations by feature, reference sequence, or reference sequence position. Use object methods to extract data for a subset of annotations into an array of structures.

Creation

Construct a `GFFAnnotation` object from a GFF- or GTF-formatted file.

`Annotobj = GFFAnnotation(File)` constructs a `GFFAnnotation` object `Annotobj` from `File`. Here, `File` is a character vector or string specifying a GFF- or GTF-formatted file.

Properties

`FieldNames` — Names of the available data fields for each annotation in the `GFFAnnotation` object
cell array of character vectors

Names of the available data fields for each annotation in the `GFFAnnotation` object, returned as a cell array of character vectors. This property is read only.

Data Types: `cell`

`NumEntries` — Number of annotations in the `GFFAnnotation` object
integer

Number of annotations in the `GFFAnnotation` object, returned as an integer. This property is read only.

Data Types: `double`

Object Functions

getData	Create structure containing subset of data from GTFAnnotation or GFFAnnotation object
getFeatureNames	Retrieve unique feature names from GTFAnnotation or GFFAnnotation object
getIndex	Return index array of annotations from GTFAnnotation or GFFAnnotation object
getRange	Retrieve range of annotations from GTFAnnotation or GFFAnnotation object
getReferenceNames	Retrieve reference names from GTFAnnotation or GFFAnnotation object
getSubset	Retrieve subset of elements from GTFAnnotation or GFFAnnotation object

Examples

Create GFFAnnotation Object

Construct a GFFAnnotation object from a GFF-formatted file that is provided with Bioinformatics Toolbox™.

```
GFFAnnotObj1 = GFFAnnotation('tair8_1.gff')
```

```
GFFAnnotObj1 = 
  GFFAnnotation with properties:

    FieldNames: {'Reference'  'Start'  'Stop'  'Feature'  'Source'  'Score'  'Strand'  'Frame'
    NumEntries: 3331
```

Construct a GFFAnnotation object from a GTF-formatted file that is provided with Bioinformatics Toolbox™.

```
GFFAnnotObj2 = GFFAnnotation('hum37_2_1M.gtf')
```

```
GFFAnnotObj2 = 
  GFFAnnotation with properties:

    FieldNames: {'Reference'  'Start'  'Stop'  'Feature'  'Source'  'Score'  'Strand'  'Frame'
    NumEntries: 308
```

Version History

Introduced in R2011b

See Also

GTFAnnotation

Topics

"Store and Manage Feature Annotations in Objects"

External Websites

GTF2.2: A Gene Annotation Format
GFF (General Feature Format) specifications

goannotread

Read annotations from Gene Ontology annotated file

Syntax

```
Annotation = goannotread(File)
Annotation = goannotread(File, ...'Fields', FieldsValue, ...)
Annotation = goannotread(File, ...'Aspect', AspectValue, ...)
```

Input Arguments

File	Character vector or string specifying a file name of a Gene Ontology (GO) annotated format (GAF) file.
FieldsValue	Character vector, string, string vector, or cell array of character vectors specifying one or more fields to read from the Gene Ontology annotated file. Default is to read all fields. Valid fields are listed below.
AspectValue	Character vector or string specifying one or more characters. Valid aspects are: • `P` — Biological process • `F` — Molecular function • `C` — Cellular component Default is `'CFP'`, which specifies to read all aspects.

Output Arguments

Annotation	MATLAB array of structures containing annotations from a Gene Ontology annotated file.

Description

Note The `goannotread` function supports GAF 1.0 and 2.0 file formats.

`Annotation = goannotread(File)` converts the contents of *File*, a Gene Ontology annotated file, into *Annotation*, an array of structures. Files should have the structure specified by the Gene Ontology consortium, available at:

`http://www.geneontology.org/`

`Annotation = goannotread(File, ...'PropertyName', PropertyValue, ...)` calls `goannotread` with optional properties that use property name/property value pairs. You can specify one or more properties in any order. Each *PropertyName* must be enclosed in single quotation marks and is case insensitive. These property name/property value pairs are as follows:

`Annotation = goannotread(File, ...'Fields', FieldsValue, ...)` specifies the fields to read from the Gene Ontology annotated file. `FieldsValue` is a character vector, string, string vector, or cell array of character vectors specifying one or more fields. Default is to read all fields. Valid fields are:

- `Database`
- `DB_Object_ID`
- `DB_Object_Symbol`
- `Qualifier`
- `GOid`
- `DBReference`
- `Evidence`
- `WithFrom`
- `Aspect`
- `DB_Object_Name`
- `Synonym`
- `DB_Object_Type`
- `Taxon`
- `Date`
- `Assigned_by`

`Annotation = goannotread(File, ...'Aspect', AspectValue, ...)` specifies the aspects to read from the Gene Ontology annotated file. `AspectValue` is a character vector or string specifying one or more characters. Valid aspects are:

- `P` — Biological process
- `F` — Molecular function
- `C` — Cellular component

Default is `'CFP'`, which specifies to read all aspects.

Examples

Read Annotations from Gene Ontology Annotated File

1. Download `gene_association.sgd.gz`, the file containing `GO` annotations for the gene products of *Saccharomyces cerevisiae*, from the yeast genome website to your MATLAB current folder.
2. Uncompress the file using the `gunzip` function.

   ```
   gunzip('gene_association.sgd.gz')
   ```
3. Load the file.

   ```
   SGDGenes = goannotread('gene_association.sgd');
   ```
4. Create a structure with `GO` annotations and display a list of the first five genes.

   ```
   S = struct2cell(SGDGenes);
   genes = S(3,1:5)'
   ```

```
genes =

    '15S_RRNA'
    '15S_RRNA'
    '15S_RRNA'
    '15S_RRNA'
    '21S_RRNA'
```

5 You can limit the annotations to genes related to molecular function (`F`) and to the fields for the gene symbol and the associated ID, that is, `DB_Object_Symbol` and `GOid`.

```
sgdSelect = goannotread('gene_association.sgd','Aspect','F','Fields',{'DB_Object_Symbol','GOi
```

```
sgdSelect =

  30701×1 struct array with fields:

    DB_Object_Symbol
    GOid
```

6 Create a list of genes and the associated GO terms.

```
selectGenes = {sgdSelect.DB_Object_Symbol};
selectGO = [sgdSelect.GOid];
```

Version History

Introduced before R2006a

See Also

`geneont` | `num2goid` | `geneont` | `getancestors` | `getdescendants` | `getmatrix` | `getrelatives`

gonnet

Return Gonnet scoring matrix

Syntax

```
gonnet
```

Description

`gonnet` returns the Gonnet matrix.

The Gonnet matrix is the recommended mutation matrix for initially aligning protein sequences. Matrix elements are ten times the logarithmic of the probability that the residues are aligned divided by the probability that the residues are aligned by chance, and then matrix elements are normalized to 250 PAM units.

Expected score = `-0.6152`, Entropy = `1.6845` bits, Lowest score = `-8`, Highest score = `14.2`

Order:

```
A  R  N  D  C  Q  E  G  H  I  L  K  M  F  P  S  T  W  Y  V  B  Z  X  *
```

Version History

Introduced before R2006a

References

[1] Gaston, H., Gonnet, M., Cohen, A., and Benner, S. (1992). Exhaustive matching of the entire protein sequence database. Science. *256*, 1443–1445.

See Also

`blosum` | `dayhoff` | `localalign` | `nuc44` | `nwalign` | `pam` | `swalign`

gprread

Read microarray data from GenePix Results (GPR) file

Syntax

```
GPRData = gprread(File)

gprread(..., 'PropertyName', PropertyValue,...)
gprread(..., 'CleanColNames', CleanColNamesValue)
```

Arguments

File	GenePix Results (GPR) formatted file. Enter a character vector or string specifying a file name, or a path and file name.
CleanColNamesValue	Controls the creation of column names that can be used as variable names.

Description

`GPRData = gprread(File)` reads GenePix results data from *File* and creates a MATLAB structure (*GPRData*).

`gprread(..., 'PropertyName', PropertyValue,...)` defines optional properties using property name/value pairs.

`gprread(..., 'CleanColNames', CleanColNamesValue)` controls the creation of column names that can be used as variable names. A GPR file may contain column names with spaces and some characters that the MATLAB software cannot use in MATLAB variable names. If *CleanColNamesValue* is `true`, `gprread` returns names in the field `ColumnNames` that are valid MATLAB variable names and names that you can use in functions. By default, *CleanColNamesValue* is `false` and the field `ColumnNames` may contain characters that are invalid for MATLAB variable names.

The field `Indices` of the structure contains indices that can be used for plotting heat maps of the data.

For more details on the GPR format, see

`https://mdc.custhelp.com/app/answers/detail/a_id/18883/kw/file%20format`

The function supports versions 3, 4, and 5 of the GenePix Results Format.

Examples

Read and display data from GenePix® result (GPR) file

This example shows how to read and display data from a GenePix® result (GPR) file.

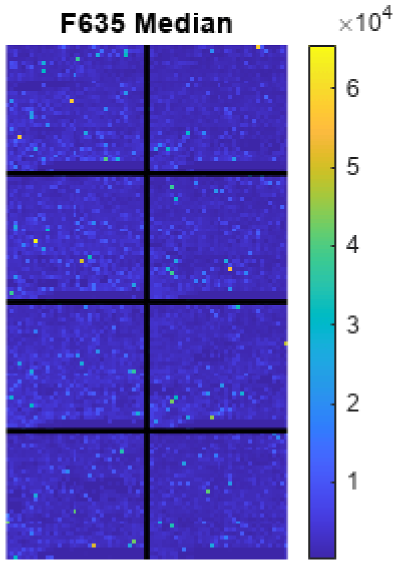
Read in a sample GPR file.

```
gprStruct = gprread('mouse_a1pd.gpr')
```

```
gprStruct = struct with fields:
         Header: [1x1 struct]
           Data: [9504x38 double]
         Blocks: [9504x1 double]
        Columns: [9504x1 double]
           Rows: [9504x1 double]
          Names: {9504x1 cell}
            IDs: {9504x1 cell}
    ColumnNames: {38x1 cell}
        Indices: [132x72 double]
          Shape: [1x1 struct]
```

Plot the median foreground intensity for the 635 nm channel.

```
maimage(gprStruct,'F635 Median')
```



Version History

Introduced before R2006a

See Also

affyread | agferead | celintensityread | galread | geoseriesread | geosoftread | ilmnbsread | imageneread | magetfield | sptread

graphallshortestpaths

(Removed) Find all shortest paths in graph

Note has been removed. Use `distances` instead.

Syntax

```
[dist] = graphallshortestpaths(G)

[dist] = graphallshortestpaths(G, ...'Directed', DirectedValue, ...)
[dist] = graphallshortestpaths(G, ...'Weights', WeightsValue, ...)
```

Arguments

G	N-by-N adjacency matrix that represents a graph. Nonzero entries in matrix *G* represent the weights of the edges.
DirectedValue	Property that indicates whether the graph is directed or undirected. Enter `false` for an undirected graph. This results in the upper triangle of the matrix being ignored. Default is `true`.
WeightsValue	Column vector that specifies custom weights for the edges in matrix *G*. It must have one entry for every nonzero value (edge) in matrix *G*. The order of the custom weights in the vector must match the order of the nonzero values in matrix *G* when it is traversed column-wise. This property lets you use zero-valued weights. By default, `graphallshortestpaths` gets weight information from the nonzero entries in matrix *G*.

Description

Tip For introductory information on graph theory functions, see “Graph Theory Functions”.

`[dist] = graphallshortestpaths(G)` finds the shortest paths between every pair of nodes in the graph represented by matrix *G*, using Johnson's algorithm. Input *G* is an N-by-N adjacency matrix that represents a graph. Nonzero entries in matrix *G* represent the weights of the edges.

Output *dist* is an N-by-N matrix where `dist(S,T)` is the distance of the shortest path from source node `S` to target node `T`. Elements in the diagonal of this matrix are always `0`, indicating the source node and target node are the same. A `0` not in the diagonal indicates that the distance between the source node and target node is `0`. An `Inf` indicates there is no path between the source node and the target node.

Johnson's algorithm has a time complexity of `O(N*log(N)+N*E)`, where `N` and `E` are the number of nodes and edges respectively.

`[...] = graphallshortestpaths (G, 'PropertyName', PropertyValue, ...)` calls `graphallshortestpaths` with optional properties that use property name/property value pairs. You can specify one or more properties in any order. Each `PropertyName` must be enclosed in single quotes and is case insensitive. These property name/property value pairs are as follows:

`[dist] = graphallshortestpaths(G, ...'Directed', DirectedValue, ...)` indicates whether the graph is directed or undirected. Set `DirectedValue` to `false` for an undirected graph. This results in the upper triangle of the matrix being ignored. Default is `true`.

`[dist] = graphallshortestpaths(G, ...'Weights', WeightsValue, ...)` lets you specify custom weights for the edges. `WeightsValue` is a column vector having one entry for every nonzero value (edge) in matrix `G`. The order of the custom weights in the vector must match the order of the nonzero values in matrix `G` when it is traversed column-wise. This property lets you use zero-valued weights. By default, `graphallshortestpaths` gets weight information from the nonzero entries in matrix `G`.

Version History

Introduced in R2006b

R2022b: Removed

Errors starting in R2022b

`graphallshortestpaths` has been removed. Use `distances` instead.

R2022a: Warns

Warns starting in R2022a

`graphallshortestpaths` issues a warning that it will be removed in a future release.

R2021b: To be removed

Not recommended starting in R2021b

`graphallshortestpaths` runs without warning, but it will be removed in a future release.

R2021b: Full matrices are supported

Behavior changed in R2021b

The function now supports full matrices in addition to sparse matrices.

References

[1] Johnson, D.B. (1977). Efficient algorithms for shortest paths in sparse networks. Journal of the ACM *24(1)*, 1-13.

[2] Siek, J.G., Lee, L-Q, and Lumsdaine, A. (2002). The Boost Graph Library User Guide and Reference Manual, (Upper Saddle River, NJ:Pearson Education).

See Also

`distances`

graphconncomp

(Removed) Find strongly or weakly connected components in graph

Note has been removed. Use `conncomp` instead.

Syntax

```
[S, C] = graphconncomp(G)

[S, C] = graphconncomp(G, ...'Directed', DirectedValue, ...)
[S, C] = graphconncomp(G, ...'Weak', WeakValue, ...)
```

Arguments

`G`	N-by-N adjacency matrix that represents a graph. Nonzero entries in matrix *`G`* indicate the presence of an edge.
`DirectedValue`	Property that indicates whether the graph is directed or undirected. Enter `false` for an undirected graph. This results in the upper triangle of the matrix being ignored. Default is `true`. A DFS-based algorithm computes the connected components. Time complexity is `O(N+E)`, where `N` and `E` are number of nodes and edges respectively.
`WeakValue`	Property that indicates whether to find weakly connected components or strongly connected components. A weakly connected component is a maximal group of nodes that are mutually reachable by violating the edge directions. Set *`WeakValue`* to `true` to find weakly connected components. Default is `false`, which finds strongly connected components. The state of this parameter has no effect on undirected graphs because weakly and strongly connected components are the same in undirected graphs. Time complexity is `O(N+E)`, where `N` and `E` are number of nodes and edges respectively.

Description

Tip For introductory information on graph theory functions, see “Graph Theory Functions”.

`[S, C] = graphconncomp(G)` finds the strongly connected components of the graph represented by matrix *`G`* using Tarjan's algorithm. A strongly connected component is a maximal group of nodes that are mutually reachable without violating the edge directions. Input *`G`* is an N-by-N adjacency matrix that represents a graph. Nonzero entries in matrix *`G`* indicate the presence of an edge.

The number of components found is returned in *`S`*, and *`C`* is a vector indicating to which component each node belongs.

Tarjan's algorithm has a time complexity of `O(N+E)`, where `N` and `E` are the number of nodes and edges respectively.

`[S, C] = graphconncomp(G, ...'PropertyName', PropertyValue, ...)` calls `graphconncomp` with optional properties that use property name/property value pairs. You can specify one or more properties in any order. Each *PropertyName* must be enclosed in single quotes and is case insensitive. These property name/property value pairs are as follows:

`[S, C] = graphconncomp(G, ...'Directed', DirectedValue, ...)` indicates whether the graph is directed or undirected. Set *directedValue* to `false` for an undirected graph. This results in the upper triangle of the matrix being ignored. Default is `true`. A DFS-based algorithm computes the connected components. Time complexity is `O(N+E)`, where `N` and `E` are number of nodes and edges respectively.

`[S, C] = graphconncomp(G, ...'Weak', WeakValue, ...)` indicates whether to find weakly connected components or strongly connected components. A weakly connected component is a maximal group of nodes that are mutually reachable by violating the edge directions. Set *WeakValue* to `true` to find weakly connected components. Default is `false`, which finds strongly connected components. The state of this parameter has no effect on undirected graphs because weakly and strongly connected components are the same in undirected graphs. Time complexity is `O(N+E)`, where `N` and `E` are number of nodes and edges respectively.

Note By definition, a single node can be a strongly connected component.

Note A directed acyclic graph (DAG) cannot have any strongly connected components larger than one.

Version History

R2022b: Removed
Errors starting in R2022b

`graphconncomp` has been removed. Use `conncomp` instead. Note that `graphconncomp` produces the same number of components (bins) but with different IDs than `conncomp`.

R2022a: Warns
Warns starting in R2022a

`graphconncomp` issues a warning that it will be removed in a future release .

R2021b: To be removed
Not recommended starting in R2021b

`graphconncomp` runs without warning, but it will be removed in a future release.

R2021b: Full matrices are supported
Behavior changed in R2021b

The function now supports full matrices in addition to sparse matrices.

References

[1] Tarjan, R.E., (1972). Depth first search and linear graph algorithms. SIAM Journal on Computing *1(2)*, 146–160.

[2] Sedgewick, R., (2002). Algorithms in C++, Part 5 Graph Algorithms (Addison-Wesley).

[3] Siek, J.G., Lee, L-Q, and Lumsdaine, A. (2002). The Boost Graph Library User Guide and Reference Manual, (Upper Saddle River, NJ:Pearson Education).

See Also

`conncomp`

graphisdag

(Removed) Test for cycles in directed graph

Note has been removed. Use `isdag` instead.

Syntax

`graphisdag(G)`

Arguments

G	N-by-N adjacency matrix that represents a directed graph. Nonzero entries in matrix *G* indicate the presence of an edge.

Description

Tip For introductory information on graph theory functions, see “Graph Theory Functions”.

`graphisdag(G)` returns logical 1 (`true`) if the directed graph represented by matrix *G* is a directed acyclic graph (DAG) and logical 0 (`false`) otherwise. *G* is an N-by-N adjacency matrix that represents a directed graph. Nonzero entries in matrix *G* indicate the presence of an edge.

Version History

Introduced in R2006b

R2022b: Removed

Errors starting in R2022b

`graphisdag` has been removed. Use `isdag` instead.

R2022a: Warns

Warns starting in R2022a

`graphisdag` issues a warning that it will be removed in a future release.

R2021b: To be removed

Not recommended starting in R2021b

`graphisdag` runs without warning, but it will be removed in a future release.

R2021b: Full matrices are supported

Behavior changed in R2021b

The function now supports full matrices in addition to sparse matrices.

References

[1] Siek, J.G., Lee, L-Q, and Lumsdaine, A. (2002). The Boost Graph Library User Guide and Reference Manual, (Upper Saddle River, NJ:Pearson Education).

See Also

`isdag`

graphisomorphism

(Removed) Find isomorphism between two graphs

Note has been removed. Use `isomorphism` instead.

Syntax

```
[Isomorphic, Map] = graphisomorphism(G1, G2)
[Isomorphic, Map] = graphisomorphism(G1, G2,'Directed', DirectedValue)
```

Arguments

`G1`	N-by-N adjacency matrix that represents a directed or undirected graph. Nonzero entries in matrix *`G1`* indicate the presence of an edge.
`G2`	N-by-N adjacency matrix that represents a directed or undirected graph. *`G2`* must be the same (directed or undirected) as *`G1`*.
`DirectedValue`	Property that indicates whether the graphs are directed or undirected. Enter `false` when both *`G1`* and *`G2`* are undirected graphs. In this case, the upper triangles of the matrices *`G1`* and *`G2`* are ignored. Default is `true`, meaning that both graphs are directed.

Description

Tip For introductory information on graph theory functions, see “Graph Theory Functions”.

`[`*`Isomorphic`*`, `*`Map`*`] = graphisomorphism(`*`G1`*`, `*`G2`*`)` returns logical 1 (`true`) in *`Isomorphic`* if *`G1`* and *`G2`* are isomorphic graphs, and logical 0 (`false`) otherwise. A graph isomorphism is a 1-to-1 mapping of the nodes in the graph *`G1`* and the nodes in the graph *`G2`* such that adjacencies are preserved. *`G1`* and *`G2`* are both N-by-N adjacency matrices that represent directed or undirected graphs. Return value *`Isomorphic`* is Boolean. When *`Isomorphic`* is `true`, *`Map`* is a row vector containing the node indices that map from *`G2`* to *`G1`* for one possible isomorphism. When *`Isomorphic`* is `false`, *`Map`* is empty. The worst-case time complexity is `O(N!)`, where `N` is the number of nodes.

`[`*`Isomorphic`*`, `*`Map`*`] = graphisomorphism(`*`G1`*`, `*`G2`*`,'Directed', `*`DirectedValue`*`)` indicates whether the graphs are directed or undirected. Set *`DirectedValue`* to `false` when both *`G1`* and *`G2`* are undirected graphs. In this case, the upper triangles of the matrices *`G1`* and *`G2`* are ignored. Default is `true`, meaning that both graphs are directed.

Version History

Introduced in R2006b

R2022b: Removed

Errors starting in R2022b

`graphisomorphism` has been removed. Use `isomorphism` instead.

R2022a: Warns
Warns starting in R2022a

`graphisomorphism` issues a warning that it will be removed in a future release.

R2021b: To be removed
Not recommended starting in R2021b

`graphisomorphism` runs without warning, but it will be removed in a future release.

R2021b: Full matrices are supported
Behavior changed in R2021b

The function now supports full matrices in addition to sparse matrices.

References

[1] Fortin, S. (1996). The Graph Isomorphism Problem. Technical Report, 96-20, Dept. of Computer Science, University of Alberta, Edomonton, Alberta, Canada.

[2] McKay, B.D. (1981). Practical Graph Isomorphism. Congressus Numerantium *30*, 45-87.

[3] Siek, J.G., Lee, L-Q, and Lumsdaine, A. (2002). The Boost Graph Library User Guide and Reference Manual, (Upper Saddle River, NJ:Pearson Education).

graphisspantree

(Removed) Determine if tree is spanning tree

Note has been removed. For details, see “Version History”.

Syntax

TF = graphisspantree(*G*)

Arguments

G	N-by-N adjacency matrix whose lower triangle represents an undirected graph. Nonzero entries in matrix *G* indicate the presence of an edge.

Description

Tip For introductory information on graph theory functions, see “Graph Theory Functions”.

`TF = graphisspantree(G)` returns logical 1 (`true`) if *G* is a spanning tree, and logical 0 (`false`) otherwise. A spanning tree must touch all the nodes and must be acyclic. *G* is an N-by-N sparse matrix whose lower triangle represents an undirected graph. Nonzero entries in matrix *G* indicate the presence of an edge.

Version History

Introduced in R2006b

R2022b: Removed

Errors starting in R2022b

`graphisspantree` has been removed. A graph is a spanning tree if and only if all nodes are reachable from an arbitrary start node, and `E == N-1`, where *E* is the number of edges and *N* is the number of nodes. You can use either `bfsearch` or `dfsearch` to check if such conditions are true for a given graph.

R2022a: Warns

Warns starting in R2022a

`graphisspantree` issues a warning that it will be removed in a future release.

R2021b: To be removed

Not recommended starting in R2021b

`graphisspantree` runs without warning, but it will be removed in a future release.

R2021b: Full matrices are supported
Behavior changed in R2021b

The function now supports full matrices in addition to sparse matrices.

References

[1] Siek, J.G., Lee, L-Q, and Lumsdaine, A. (2002). The Boost Graph Library User Guide and Reference Manual, (Upper Saddle River, NJ:Pearson Education).

See Also

`bfsearch` | `dfsearch`

graphmaxflow

(Removed) Calculate maximum flow in directed graph

Note has been removed. Use `maxflow` instead. For details, see “Version History”.

Syntax

```
[MaxFlow, FlowMatrix, Cut] = graphmaxflow(G, SNode, TNode)

[...] = graphmaxflow(G, SNode, TNode, ...'Capacity', CapacityValue, ...)
[...] = graphmaxflow(G, SNode, TNode, ...'Method', MethodValue, ...)
```

Arguments

`G`	N-by-N adjacency matrix that represents a directed graph. Nonzero entries in matrix `G` represent the capacities of the edges.
`SNode`	Node in *`G`*.
`TNode`	Node in *`G`*.
`CapacityValue`	Column vector that specifies custom capacities for the edges in matrix *`G`*. It must have one entry for every nonzero value (edge) in matrix *`G`*. The order of the custom capacities in the vector must match the order of the nonzero values in matrix *`G`* when it is traversed column-wise. By default, `graphmaxflow` gets capacity information from the nonzero entries in matrix *`G`*.
`MethodValue`	Character vector or string that specifies the algorithm used to find the minimal spanning tree (MST). Choices are: • `'Edmonds'` — Uses the Edmonds and Karp algorithm, the implementation of which is based on a variation called the *labeling algorithm*. Time complexity is `O(N*E^2)`, where `N` and `E` are the number of nodes and edges respectively. • `'Goldberg'` — Default algorithm. Uses the Goldberg algorithm, which uses the generic method known as *preflow-push*. Time complexity is `O(N^2*sqrt(E))`, where `N` and `E` are the number of nodes and edges respectively.

Description

Tip For introductory information on graph theory functions, see “Graph Theory Functions”.

`[`*`MaxFlow`*`, `*`FlowMatrix`*`, `*`Cut`*`] = graphmaxflow(`*`G`*`, `*`SNode`*`, `*`TNode`*`)` calculates the maximum flow of directed graph *`G`* from node *`SNode`* to node *`TNode`*. Input *`G`* is an N-by-N adjacency matrix that represents a directed graph. Nonzero entries in matrix `G` represent the capacities of the edges. Output *`MaxFlow`* is the maximum flow, and *`FlowMatrix`* is a adjacency matrix with all the flow values for every edge. `FlowMatrix`(*X*,*Y*) is the flow from node `X` to node `Y`. Output `Cut` is a logical row vector indicating the nodes connected to `SNode` after calculating the minimum cut between `SNode` and `TNode`. If several solutions to the minimum cut problem exist, then `Cut` is a matrix.

Tip The algorithm that determines `Cut`, all minimum cuts, has a time complexity of `O(2^N)`, where *N* is the number of nodes. If this information is not needed, use the `graphmaxflow` function without the third output.

`[...] = graphmaxflow(G, SNode, TNode, ...'PropertyName', PropertyValue, ...)` calls `graphmaxflow` with optional properties that use property name/property value pairs. You can specify one or more properties in any order. Each `PropertyName` must be enclosed in single quotes and is case insensitive. These property name/property value pairs are as follows:

`[...] = graphmaxflow(G, SNode, TNode, ...'Capacity', CapacityValue, ...)` lets you specify custom capacities for the edges. `CapacityValue` is a column vector having one entry for every nonzero value (edge) in matrix `G`. The order of the custom capacities in the vector must match the order of the nonzero values in matrix `G` when it is traversed column-wise. By default, `graphmaxflow` gets capacity information from the nonzero entries in matrix `G`.

`[...] = graphmaxflow(G, SNode, TNode, ...'Method', MethodValue, ...)` lets you specify the algorithm used to find the minimal spanning tree (MST). Choices are:

- `'Edmonds'` — Uses the Edmonds and Karp algorithm, the implementation of which is based on a variation called the *labeling algorithm*. Time complexity is `O(N*E^2)`, where `N` and `E` are the number of nodes and edges respectively.
- `'Goldberg'` — Default algorithm. Uses the Goldberg algorithm, which uses the generic method known as *preflow-push*. Time complexity is `O(N^2*sqrt(E))`, where `N` and `E` are the number of nodes and edges respectively.

Version History

Introduced in R2006b

R2022b: Removed
Errors starting in R2022b

`graphmaxflow` has been removed. Use `maxflow` instead. Note that `maxflow` returns only one solution while `graphmaxflow` returns multiple solutions if they exist.

R2022a: Warns
Warns starting in R2022a

`graphmaxflow` issues a warning that it will be removed in a future release.

R2021b: To be removed
Not recommended starting in R2021b

`graphmaxflow` runs without warning, but it will be removed in a future release.

R2021b: Full matrices are supported
Behavior changed in R2021b

The function now supports full matrices in addition to sparse matrices.

References

[1] Edmonds, J. and Karp, R.M. (1972). Theoretical improvements in the algorithmic efficiency for network flow problems. Journal of the ACM *19*, 248-264.

[2] Goldberg, A.V. (1985). A New Max-Flow Algorithm. MIT Technical Report MIT/LCS/TM-291, Laboratory for Computer Science, MIT.

[3] Siek, J.G., Lee, L-Q, and Lumsdaine, A. (2002). The Boost Graph Library User Guide and Reference Manual, (Upper Saddle River, NJ:Pearson Education).

graphminspantree

(Removed) Find minimal spanning tree in graph

Note has been removed. Use `minspantree` instead. For details, see "Version History".

Syntax

```
[Tree, pred] = graphminspantree(G)
[Tree, pred] = graphminspantree(G, R)

[Tree, pred] = graphminspantree(..., 'Method', MethodValue, ...)
[Tree, pred] = graphminspantree(..., 'Weights', WeightsValue, ...)
```

Arguments

G	N-by-N adjacency matrix that represents an undirected graph. Nonzero entries in matrix *G* represent the weights of the edges.
R	Scalar between 1 and the number of nodes.

Description

Tip For introductory information on graph theory functions, see "Graph Theory Functions".

`[Tree, pred] = graphminspantree(G)` finds an acyclic subset of edges that connects all the nodes in the undirected graph *G* and for which the total weight is minimized. Weights of the edges are all nonzero entries in the lower triangle of the N-by-N adjacency matrix *G*. Output *Tree* is a spanning tree represented by an adjacency matrix. Output *pred* is a vector containing the predecessor nodes of the minimal spanning tree (MST), with the root node indicated by `0`. The root node defaults to the first node in the largest connected component. This computation requires an extra call to the `graphconncomp` function.

`[Tree, pred] = graphminspantree(G, R)` sets the root of the minimal spanning tree to node `R`.

`[Tree, pred] = graphminspantree(..., 'PropertyName', PropertyValue, ...)` calls `graphminspantree` with optional properties that use property name/property value pairs. You can specify one or more properties in any order. Each *PropertyName* must be enclosed in single quotes and is case insensitive. These property name/property value pairs are as follows:

`[Tree, pred] = graphminspantree(..., 'Method', MethodValue, ...)` lets you specify the algorithm used to find the minimal spanning tree (MST). Choices are:

- `'Kruskal'` — Grows the minimal spanning tree (MST) one edge at a time by finding an edge that connects two trees in a spreading forest of growing MSTs. Time complexity is `O(E+X*log(N))`, where `X` is the number of edges no longer than the longest edge in the MST, and `N` and `E` are the number of nodes and edges respectively.

- `'Prim'` — Default algorithm. Grows the minimal spanning tree (MST) one edge at a time by adding a minimal edge that connects a node in the growing MST with any other node. Time complexity is `O(E*log(N))`, where `N` and `E` are the number of nodes and edges respectively.

Note When the graph is unconnected, Prim's algorithm returns only the tree that contains R, while Kruskal's algorithm returns an MST for every component.

`[`*`Tree`*`, `*`pred`*`] = graphminspantree(..., 'Weights', `*`WeightsValue`*`, ...)` lets you specify custom weights for the edges. *`WeightsValue`* is a column vector having one entry for every nonzero value (edge) in matrix *`G`*. The order of the custom weights in the vector must match the order of the nonzero values in matrix *`G`* when it is traversed column-wise. By default, `graphminspantree` gets weight information from the nonzero entries in matrix *`G`*.

Version History

Introduced in R2006b

R2022b: Removed

Errors starting in R2022b

`graphminspantree` has been removed. Use `minspantree` instead. When you run `minspantree` by setting the `'Type'` name-value argument to `'forest'`, the returned list of predecessors includes predecessor node IDs even though they are not reachable from the original root. `graphminspantree` sets the predecessors for those unreachable nodes from the root to be `NaN`.

R2022a: Warns

Warns starting in R2022a

`graphminspantree` issues a warning that it will be removed in a future release.

R2021b: To be removed

Not recommended starting in R2021b

`graphminspantree` runs without warning, but it will be removed in a future release.

R2021b: Full matrices are supported

Behavior changed in R2021b

The function now supports full matrices in addition to sparse matrices.

References

[1] Kruskal, J.B. (1956). On the Shortest Spanning Subtree of a Graph and the Traveling Salesman Problem. Proceedings of the American Mathematical Society *7*, 48-50.

[2] Prim, R. (1957). Shortest Connection Networks and Some Generalizations. Bell System Technical Journal *36*, 1389-1401.

[3] Siek, J.G. Lee, L-Q, and Lumsdaine, A. (2002). The Boost Graph Library User Guide and Reference Manual, (Upper Saddle River, NJ:Pearson Education).

See Also

`minspantree`

graphpred2path

(Removed) Convert predecessor indices to paths

Note has been removed.

Syntax

`path = graphpred2path(pred, D)`

Arguments

pred	Row vector or matrix of predecessor node indices. The value of the root (or source) node in *pred* must be 0.
D	Destination node in *pred*.

Description

Tip For introductory information on graph theory functions, see “Graph Theory Functions”.

`path = graphpred2path(pred, D)` traces back a path by following the predecessor list in *pred* starting at destination node *D*.

The value of the root (or source) node in *pred* must be 0. If a NaN is found when following the predecessor nodes, `graphpred2path` returns an empty path.

If *pred* is a ...	And *D* is a ...	Then *path* is a ...
row vector of predecessor node indices	scalar	row vector listing the nodes from the root (or source) to *D*.
	row vector	row cell array with every column containing the path to the destination for every element in *D*.
matrix	scalar	column cell array with every row containing the path for every row in *pred*.
	row vector	matrix cell array with every row containing the paths for the respective row in *pred*, and every column containing the paths to the respective destination in *D*.

Note If *D* is omitted, the paths to all the destinations are calculated for every predecessor listed in *pred*.

Version History

Introduced in R2006b

R2022b: Removed
Errors starting in R2022b

`graphpred2path` has been removed.

R2022a: Warns
Warns starting in R2022a

`graphpred2path` issues a warning that it will be removed in a future release.

R2021b: To be removed
Not recommended starting in R2021b

`graphpred2path` runs without warning, but it will be removed in a future release.

References

[1] Siek, J.G., Lee, L-Q, and Lumsdaine, A. (2002). The Boost Graph Library User Guide and Reference Manual, (Upper Saddle River, NJ:Pearson Education).

graphshortestpath

(Removed) Solve shortest path problem in graph

Note has been removed. Use `shortestpath` or `shortestpathtree` instead.

Syntax

```
[dist,path,pred] = graphshortestpath(G,S)
[ ___ ] = graphshortestpath(G,S,D)
[ ___ ] = graphshortestpath( ___ ,Name,Value)
```

Description

`[dist,path,pred] = graphshortestpath(G,S)` determines the shortest paths from the source node *S* to all other nodes in the graph `G`. `dist` contains the distances from the source node to all other nodes. `path` contains the shortest paths to every node. `pred` contains the predecessor nodes of the shortest paths.

`[ ___ ] = graphshortestpath(G,S,D)` determines the shortest path from the source node `S` to the target node `D` and returns any of the output arguments from the previous syntax.

`[ ___ ] = graphshortestpath( ___ ,Name,Value)` specifies additional options using one or more name-value pair arguments. Specify name-value pair arguments after any of the input argument combinations in the previous syntaxes.

Input Arguments

`G` — Adjacency matrix
matrix

Adjacency matrix, specified as an *N*-by-*N* matrix that represents a graph. Nonzero entries in the matrix represent the weights of the edges.

Data Types: `double` | `single` | `logical`

`S` — Source node
numeric node index

Source node, specified as a numeric node index.

Example: 2

Data Types: `double`

`D` — Destination node
numeric node index

Destination node, specified as a numeric node index.

Example: 5

Data Types: `double`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `[dist,path,pred] = graphshortestpath(G,1,5,'Method',Acyclic')` assumes `G` to be a directed acyclic graph when finding the shortest path from node `1` to node `5`.

`Method` — Shortest path algorithm
`'Dijkstra'` (default) | `'Bellman-Ford'` | `'BFS'` | `'Acyclic'`

Shortest path algorithm, specified as the comma-separated pair consisting of `'Method'` and one of these options.

Option	Description
`'Dijkstra'` (default)	This algorithm assumes that all edge weights are positive values in `G`. The time complexity is `O(log(`*N*`)*`*E*`)`, where *N* and *E* are the number of nodes and the number of edges respectively.
`'BFS'`	This breadth-first search algorithm assumes that all weights are equal and that edges are nonzero entries in the adjacency matrix `G`. The time complexity is `O(`*N*+*E*`)`.
`'Bellman-Ford'`	This algorithm assumes that all edge weights are nonzero entries in `G`. The time complexity is `O(`*N***E*`)`.
`'Acyclic'`	This algorithm assumes that `G` is a directed acyclic graph and all edge weights are nonzero entries in `G`. The time complexity is `O(N+E)`.

Example: `'Method','Acyclic'`

Data Types: `char` | `string`

`Directed` — Directed or undirected graph flag
`true` (default) | `false`

Directed or undirected graph flag, specified as a comma-separated pair consisting of `'Directed'` and `true` or `false`. If `false`, the function ignores the upper triangle of the adjacency matrix `G`.

Example: `'Directed',false`

Data Types: `logical`

`Weights` — Custom weights for edges in `G`
column vector

Custom weights for edges in the matrix G, specified as a comma-separated pair consisting of `'Weights'` and a column vector. The vector must meet the following conditions.

- It must have one entry for every nonzero value (edge) in the matrix `G`.
- The order of the custom weights in the vector must match the order of the nonzero values in `G` when it is traversed columnwise.

You can specify zero-valued weights. By default, the function obtains the weight information from the nonzero entries in `G`.

Example: `'Weights',[1 2.3 1.3 0 4]`

Data Types: `double`

Output Arguments

`dist` — Distances from source node to all other nodes in graph
numeric scalar | numeric vector

Distances from the source node to all other nodes in the graph, returned as a numeric scalar or vector. `dist` is returned as a scalar if you specify a destination node as the third input argument.

The function returns `Inf` for nonreachable nodes and `0` for the source node.

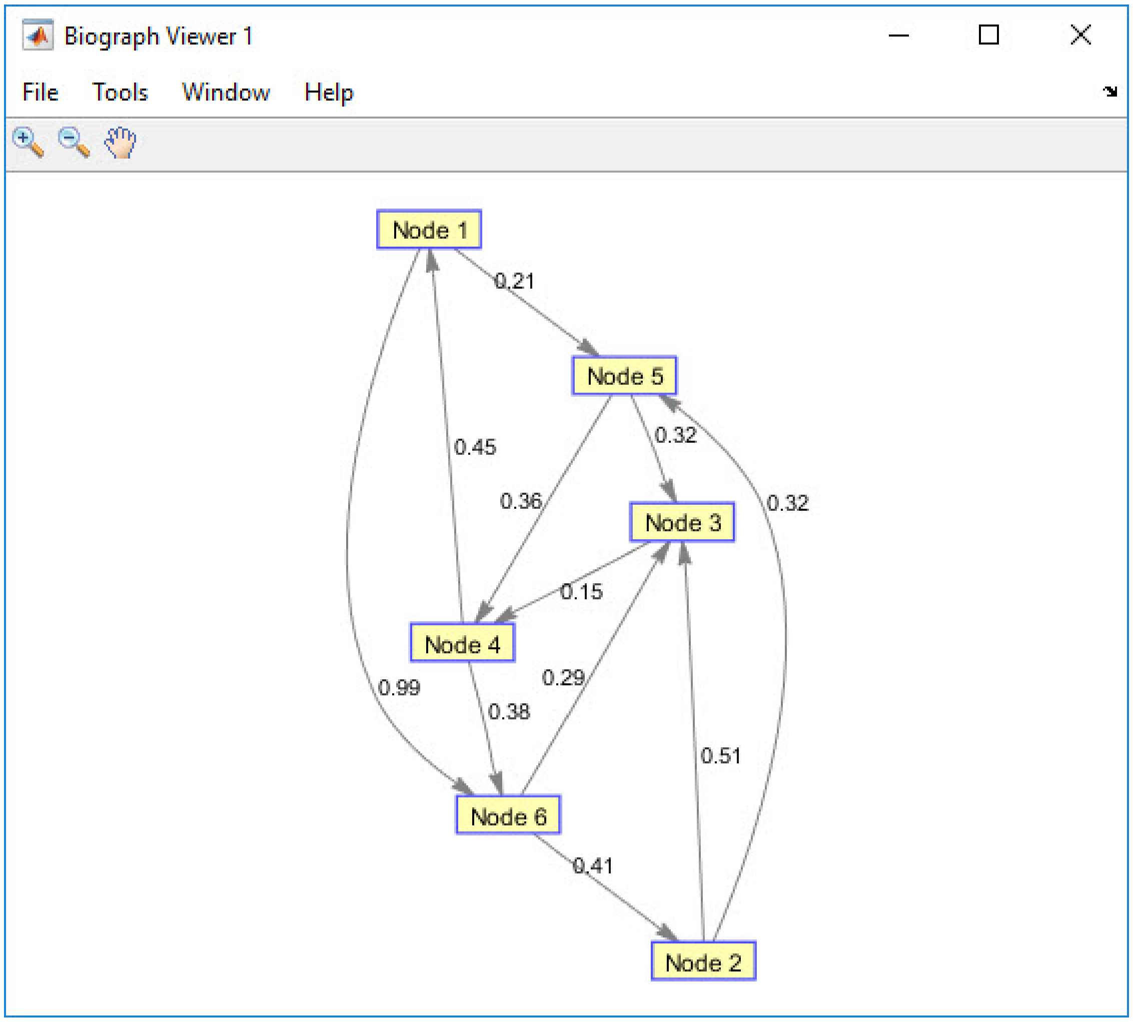
`path` — Shortest paths from source node to all other nodes
vector | cell array

Shortest paths from the source node to all other nodes, returned as a vector or cell array. It is returned as a vector if you specify a destination node. Each number represents a node index in the graph.

`pred` — Predecessor nodes of shortest paths
vector

Predecessor nodes of the shortest paths, returned as a vector.

You can use `pred` to determine the shortest paths from the source node to all other nodes. Suppose that you have a directed graph with 6 nodes.



The function finds that the shortest path from node 1 to node 6 is `path = [1 5 4 6]` and `pred = [0 6 5 5 1 4]`. Now you can determine the shortest paths from node 1 to any other node within the graph by indexing into `pred`. For example, to figure out the shortest path from node 1 to node 2, you can query `pred` with the destination node as the first query, then use the returned answer to get the next node. Repeat this procedure until the query answer is 0, which indicates the source node.

```
pred(2) = 6; pred(6) = 4; pred(4) = 5; pred(5) = 1; pred(1) = 0;
```

The results indicate that the shortest path from node 1 to node 2 is `1->5->4->6->2`.

Version History

Introduced in R2006b

R2022b: Removed

Errors starting in R2022b

`graphshortestpath` has been removed. Use `shortestpath` or `shortestpathtree` instead.

R2022a: Warns

Warns starting in R2022a

`graphshortestpath` issues a warning that it will be removed in a future release.

R2021b: To be removed
Not recommended starting in R2021b

`graphshortestpath` runs without warning, but it will be removed in a future release.

R2021b: Full matrices are supported
Behavior changed in R2021b

The function now supports full matrices in addition to sparse matrices.

R2021b: Edges with infinite weight are now included in the shortest path
Behavior changed in R2021b

The function uses edges with infinite weight and include them in the shortest path if no paths with finite length exist.

References

[1] Dijkstra, E. W. "A Note on Two Problems in Connexion with Graphs." Numerische Mathematik. Vol. 1, Number 1, 1959, pp. 269–271.

[2] Bellman, R. "On a Routing Problem." Quarterly of Applied Mathematics. Vol. 16, Number 1, pp. 87–90.

[3] Siek, J. G., L. Q. Lee, and A. Lumsdaine. The Boost Graph Library: User Guide and Reference Manual. Upper Saddle River, NJ: Pearson Education, 2002.

See Also

`shortestpath` | `shortestpathtree`

graphtopoorder

(Removed) Perform topological sort of directed acyclic graph

Note has been removed. Use `toposort` instead.

Syntax

`*order* = graphtopoorder(*G*)`

Arguments

G	N-by-N adjacency matrix that represents a directed acyclic graph. Nonzero entries in matrix *G* indicate the presence of an edge.

Description

Tip For introductory information on graph theory functions, see “Graph Theory Functions”.

`*order* = graphtopoorder(*G*)` returns an index vector with the order of the nodes sorted topologically. In topological order, an edge can exist between a source node `u` and a destination node `v`, if and only if `u` appears before `v` in the vector *`order`*. *G* is an N-by-N adjacency matrix that represents a directed acyclic graph (DAG). Nonzero entries in matrix *G* indicate the presence of an edge.

Version History

Introduced in R2006b

R2022b: Removed

Errors starting in R2022b

`graphtopoorder` has been removed. Use `toposort` instead.

R2022a: Warns

Warns starting in R2022a

`graphtopoorder` issues a warning that it will be removed in a future release.

R2021b: To be removed

Not recommended starting in R2021b

`graphtopoorder` runs without warning, but it will be removed in a future release.

R2021b: Full matrices are supported

Behavior changed in R2021b

The function now supports full matrices in addition to sparse matrices.

References

[1] Siek, J.G., Lee, L-Q, and Lumsdaine, A. (2002). The Boost Graph Library User Guide and Reference Manual, (Upper Saddle River, NJ:Pearson Education).

See Also

`toposort`

graphtraverse

(Removed) Traverse graph by following adjacent nodes

Note has been removed. Use `bfsearch` or `dfsearch` instead. For details, see “Version History”.

Syntax

```
[disc, pred, closed] = graphtraverse(G, S)

[...] = graphtraverse(G, S, ...'Depth', DepthValue, ...)
[...] = graphtraverse(G, S, ...'Directed', DirectedValue, ...)
[...] = graphtraverse(G, S, ...'Method', MethodValue, ...)
```

Arguments

G	N-by-N adjacency matrix that represents a directed graph. Nonzero entries in matrix *G* indicate the presence of an edge.
S	Integer that indicates the source node in graph *G*.
DepthValue	Integer that indicates a node in graph *G* that specifies the depth of the search. Default is `Inf` (infinity).
DirectedValue	Property that indicates whether graph *G* is directed or undirected. Enter `false` for an undirected graph. This results in the upper triangle of the adjacency matrix being ignored. Default is `true`.
MethodValue	Character vector or string that specifies the algorithm used to traverse the graph. Choices are: • `'BFS'` — Breadth-first search. Time complexity is `O(N+E)`, where `N` and `E` are number of nodes and edges respectively. • `'DFS'` — Default algorithm. Depth-first search. Time complexity is `O(N+E)`, where `N` and `E` are number of nodes and edges respectively.

Description

Tip For introductory information on graph theory functions, see “Graph Theory Functions”.

`[`*`disc`*`, `*`pred`*`, `*`closed`*`] = graphtraverse(`*`G`*`, `*`S`*`)` traverses graph `G` starting from the node indicated by integer `S`. *G* is an N-by-N adjacency matrix that represents a directed graph. Nonzero entries in matrix *G* indicate the presence of an edge. *`disc`* is a vector of node indices in the order in which they are discovered. *`pred`* is a vector of predecessor node indices (listed in the order of the node indices) of the resulting spanning tree. *`closed`* is a vector of node indices in the order in which they are closed.

`[...] = graphtraverse(`*`G`*`, `*`S`*`, ...'`*`PropertyName`*`', `*`PropertyValue`*`, ...)` calls `graphtraverse` with optional properties that use property name/property value pairs. You can specify one or more properties in any order. Each *PropertyName* must be enclosed in single quotes and is case insensitive. These property name/property value pairs are as follows:

`[...] = graphtraverse(G, S, ...'Depth', DepthValue, ...)` specifies the depth of the search. *DepthValue* is an integer indicating a node in graph *G*. Default is `Inf` (infinity). The `'Depth'` name-value argument is no longer supported for the `'DFS'` method.

`[...] = graphtraverse(G, S, ...'Directed', DirectedValue, ...)` indicates whether the graph is directed or undirected. Set *DirectedValue* to `false` for an undirected graph. This results in the upper triangle of the adjacency matrix being ignored. Default is `true`.

`[...] = graphtraverse(G, S, ...'Method', MethodValue, ...)` lets you specify the algorithm used to traverse the graph. Choices are:

- `'BFS'` — Breadth-first search. Time complexity is `O(N+E)`, where `N` and `E` are number of nodes and edges respectively.
- `'DFS'` — Default algorithm. Depth-first search. Time complexity is `O(N+E)`, where `N` and `E` are number of nodes and edges respectively. The `'Depth'` name-value argument is no longer supported for the `'DFS'` method.

Version History

Introduced in R2006b

R2022b: Removed

Errors starting in R2022b

`graphtraverse` has been removed. Use `bfsearch` or `dfsearch` instead. Note that `bfsearch` and `dfsearch` do not have the `'Depth'` option. An example of an alternative approach would be as follows:

```
s = [1 1 2 2 2 3 4 7 8 8 8 8];
t = [3 4 7 5 6 2 6 2 9 10 11 12];
g = digraph(s,t);
v = bfsearch(g,1);
d = distances(g,1,v);
maxDepth = 2;
v2 = v(d <= maxDepth);
```

R2022a: Warns

Warns starting in R2022a

`graphtraverse` issues a warning that it will be removed in a future release.

R2021b: To be removed

Not recommended starting in R2021b

`graphtraverse` runs without warning, but it will be removed in a future release.

R2021b: `'Depth'` option for `'DFS'` method is no longer supported

Behavior changed in R2021b

The `'Depth'` name-value argument is no longer supported for the `'DFS'` method.

R2021b: Full matrices are supported
Behavior changed in R2021b

The function now supports full matrices in addition to sparse matrices.

References

[1] Sedgewick, R., (2002). Algorithms in C++, Part 5 Graph Algorithms (Addison-Wesley).

[2] Siek, J.G., Lee, L-Q, and Lumsdaine, A. (2002). The Boost Graph Library User Guide and Reference Manual, (Upper Saddle River, NJ:Pearson Education).

See Also

`bfsearch` | `dfsearch`

gt (DataMatrix)

Test DataMatrix objects for greater than

Syntax

```
T = gt(DMObj1, DMObj2)
T = DMObj1 > DMObj2
T = gt(DMObj1, B)
T = DMObj1 > B
T = gt(B, DMObj1)
T = B > DMObj1
```

Input Arguments

DMObj1, *DMObj2*	DataMatrix objects, such as created by `DataMatrix` (object constructor).
B	MATLAB numeric or logical array.

Output Arguments

T	Logical matrix of the same size as *DMObj1* and *DMObj2* or *DMObj1* and *B*. It contains logical `1` (true) where elements in the first input are greater than the corresponding element in the second input, and logical `0` (false) otherwise.

Description

T = `gt(`*DMObj1*, *DMObj2*`)` or the equivalent *T* = *DMObj1* > *DMObj2* compares each element in DataMatrix object *DMObj1* to the corresponding element in DataMatrix object *DMObj2*, and returns *T*, a logical matrix of the same size as *DMObj1* and *DMObj2*, containing logical `1` (true) where elements in *DMObj1* are greater than the corresponding element in *DMObj2*, and logical `0` (false) otherwise. *DMObj1* and *DMObj2* must have the same size (number of rows and columns), unless one is a scalar (1-by-1 DataMatrix object). *DMObj1* and *DMObj2* can have different `Name` properties.

T = `gt(`*DMObj1*, *B*`)` or the equivalent *T* = *DMObj1* > *B* compares each element in DataMatrix object *DMObj1* to the corresponding element in *B*, a numeric or logical array, and returns *T*, a logical matrix of the same size as *DMObj1* and *B*, containing logical `1` (true) where elements in *DMObj1* are greater than the corresponding element in *B*, and logical `0` (false) otherwise. *DMObj1* and *B* must have the same size (number of rows and columns), unless one is a scalar.

T = `gt(`*B*, *DMObj1*`)` or the equivalent *T* = *B* > *DMObj1* compares each element in *B*, a numeric or logical array, to the corresponding element in DataMatrix object *DMObj1*, and returns *T*, a logical matrix of the same size as *B* and *DMObj1*, containing logical `1` (true) where elements in *B* are greater than the corresponding element in *DMObj1*, and logical `0` (false) otherwise. *B* and *DMObj1* must have the same size (number of rows and columns), unless one is a scalar.

MATLAB calls *T* = `gt(`*X*, *Y*`)` for the syntax *T* = *X* > *Y* when *X* or *Y* is a DataMatrix object.

Version History
Introduced in R2008b

See Also
`DataMatrix` | `lt`

Topics
DataMatrix object on page 1-734

GTFAnnotation

Contain Gene Transfer Format (GTF) annotations

Description

The `GTFAnnotation` object contains annotations for one or more reference sequences, conforming to the GTF file format.

Each element in the object represents an annotation. Use the object properties and methods to filter annotations by feature, reference sequence, or reference sequence position. Use object methods to extract data for a subset of annotations into an array of structures.

Creation

Construct a `GTFAnnotation` object from a GTF-formatted file.

`Annotobj = GTFAnnotation(File)` constructs a `GTFAnnotation` object `Annotobj` from `File`. Here, `File` is a character vector or string specifying a GTF-formatted file.

Properties

`FieldNames` — Names of the available data fields for each annotation in the `GTFAnnotation` object
cell array of character vectors

Names of the available data fields for each annotation in the `GTFAnnotation` object, returned as a cell array of character vectors. This property is read only.

Data Types: `cell`

`NumEntries` — Number of annotations in the `GTFAnnotation` object
integer

Number of annotations in the `GTFAnnotation` object, returned as an integer. This property is read only.

Data Types: `double`

Object Functions

getData	Create structure containing subset of data from GTFAnnotation or GFFAnnotation object
getExons	Return table of exons from GTFAnnotation object
getFeatureNames	Retrieve unique feature names from GTFAnnotation or GFFAnnotation object
getGeneNames	Retrieve unique gene names from GTFAnnotation object
getGenes	Return table of unique genes in GTFAnnotation object
getIndex	Return index array of annotations from GTFAnnotation or GFFAnnotation object

getRange	Retrieve range of annotations from GTFAnnotation or GFFAnnotation object
getReferenceNames	Retrieve reference names from GTFAnnotation or GFFAnnotation object
getSegments	Return table of non-overlapping segments from GTFAnnotation object
getSubset	Retrieve subset of elements from GTFAnnotation or GFFAnnotation object
getTranscriptNames	Retrieve unique transcript names from GTFAnnotation object
getTranscripts	Return table of unique transcripts in GTFAnnotation object

Examples

Create `GTFAnnotation` Object

Construct a `GTFAnnotation` object from a GTF-formatted file that is provided with Bioinformatics Toolbox™.

```
GTFAnnotObj = GTFAnnotation('hum37_2_1M.gtf')
```

```
GTFAnnotObj = 
  GTFAnnotation with properties:

    FieldNames: {'Reference'  'Start'  'Stop'  'Feature'  'Gene'  'Transcript'  'Source'  'Score
    NumEntries: 308
```

Version History

Introduced in R2011b

See Also

`GFFAnnotation`

Topics

"Store and Manage Feature Annotations in Objects"

External Websites

GTF2.2: A Gene Annotation Format
GFF (General Feature Format) specifications

HeatMap

Object containing matrix and heatmap display properties

Description

The `HeatMap` function creates a `HeatMap` object. You can use the object to display a heatmap (2-D color image) of matrix data.

Creation

Syntax

```
HeatMap(data)
HeatMap(data,Name,Value)
```

Description

`hmObj` = `HeatMap(data)` displays a heatmap (2-D color image) of `data` and returns an object containing the data and display properties.

`hmObj` = `HeatMap(data,Name,Value)` sets the object properties on page 1-1080 using name-value pairs. For example, `HeatMap(data,'Annotate',true)` displays data values in the heatmap. You can specify multiple name-value pairs. Enclose each property name in quotes.

Input Arguments

`data` — Heatmap data
DataMatrix object | numeric matrix

Heatmap data, specified as a DataMatrix object on page 1-734 or numeric matrix.

Name-Value Pair Arguments

Use comma-separated name-value pair arguments to set the object properties on page 1-1080. Enclose each property name in single quotes.

Example: `hm = HeatMap(data,'Colormap',redbluecmap,'Annotate',true)`

Properties

`Standardize` — Dimension for standardizing data values
`'none'` (default) | `'row'` | `'column'` | `3` | `2` | `1`

Dimension for standardizing data values, specified as a character vector, string, or positive integer. Choices are:

- `'column'` or `1` — Standardize along the columns of data.

- `'row'` or `2` — Standardize along the rows of data.
- `'none'` or `3` — Do not standardize.

If you specify `'column'` or `'row'`, the function transforms the standardized values so that the mean is 0 and the standard deviation is 1 in the specified dimension.

Example: `'column'`

Data Types: `double` | `char` | `string`

`Symmetric` — Flag to make the heatmap color scale symmetric around zero
`true` (default) | `false`

Flag to make the heatmap color scale symmetric around zero, specified as `true` or `false`.

Example: `false`

Data Types: `logical`

`DisplayRange` — Display range of standardize values
maximum absolute value in `data` (default) | positive scalar

Display range of standardize values, specified as a positive scalar. The default is the maximum absolute value in the input `data`.

For example, if you specify `3`, there is a color variation for values between `-3` and `3`, but values greater than `3` are the same color as `3`, and values less than `-3` are the same color as `-3`.

For example, if you specify `redgreencmap` for the `'Colormap'` property, pure red represents values greater than or equal to the specified display range value and pure green represents values less than or equal to the negative of the specified display range value.

Example: `3`

Data Types: `double`

`Colormap` — Heatmap colors
`redgreencmap` (default) | matrix | name of function handle

heatmap colors, specified as a three-column (*M*-by-3) matrix of red-green-blue (RGB) values or the name of a function handle that returns a colormap, such as `redgreencmap` or `redbluecmap`.

The default colormap is `redgreencmap`, in which red represents values above the mean, black represents the mean, and green represents values below the mean of a row (gene) across all columns (samples).

Example: `redbluecmap`

Data Types: `double` | `char`

`ImputeFun` — Name of function or function handle to impute missing data
character vector | cell array

Name of a function or function handle to impute missing data, specified as a character vector or cell array. If you specify a cell array, the first element must be the name of a function or function handle, and the remaining elements must be name-value pairs used as inputs to the function. Missing data points are colored gray in the heatmap.

If data points are missing, you can use this property to impute the missing values.

Example: `'func1'`

Data Types: `char`

`ColumnLabels` — Column labels
`[1x0 double]` (default) | string vector | cell array of character vectors | numeric vector

Column labels, specified as a string vector, cell array of character vectors, or numeric vector. The size of the vector must match the number of columns in the input `data`.

Example: `["sample1","sample2","sample3"]`

Data Types: `double` | `string` | `cell`

`RowLabels` — Row labels
`[]` (default) | string vector | cell array of character vectors | numeric vector

Row labels, specified as a string vector, cell array of character vectors, or numeric vector. The size of the vector must match the number of rows in the input `data`.

Example: `["gene1","gene2","gene3"]`

Data Types: `double` | `string` | `cell`

`ColumnLabelsRotate` — Orientation of column labels
`90` (default) | numeric scalar

Orientation of column labels, specified as a numeric scalar. Specify the value of rotation in degrees (positive angles cause counterclockwise rotation).

Example: `30`

Data Types: `double`

`RowLabelsRotate` — Orientation of row labels
0 (default) | numeric scalar

Orientation of row labels, specified as a numeric scalar. Specify the value of rotation in degrees (positive angles cause counterclockwise rotation).

Example: `30`

Data Types: `double`

`Annotate` — Flag to display data values in heatmap
`false` (default) | `true`

Flag to display data values in the heatmap, specified as `true` or `false`.

Example: `true`

Data Types: `logical`

`AnnotPrecision` — Display precision of data values
2 (default) | numeric scalar

Display precision of data values in the heatmap, specified as a numeric scalar. The default number of digits of precision is `2`.

Example: 3

Data Types: double

AnnotColor — Text color of displayed data values
'w' (default) | character vector | string | three-element numeric vector

Text color of displayed data values in the heatmap, specified as a character vector, string, or three-element numeric vector. For example, to use cyan, you can enter [0 1 1], 'c', "c", "cyan", or 'cyan'. For details, see "Color Options" on page 1-1092.

Example: 'red'

Data Types: char | string | double

ColumnLabelsColor — Color information for column labels
[] (default) | structure | structure array

Warning This property will be removed in a future release. Set LabelsWithMarkers to true for colored markers instead of colored texts.

Color information for column labels, specified as a structure or structure array.

For a single structure, you must specify the following fields:

- Labels — Cell array of character vectors specifying column labels listed in the ColumnLabels property.
- Colors — Character vector or string specifying a color for the column labels. If this field is empty, the default color (black) is used.

For a structure array, you must specify a single element in each field for each structure.

- Labels — Character vector or string specifying a column label listed in the ColumnLabels property.
- Colors — Character vector or string specifying a color for the column labels. If this field is empty, the default color (black) is used.

For more information on specifying colors, see "Color Options" on page 1-1092.

Data Types: struct

RowLabelsColor — Color information for row labels
[] (default) | structure | structure array

Warning This property will be removed in a future release. Set LabelsWithMarkers to true for colored markers instead of colored texts.

Color information for row labels, specified as a structure or structure array.

For a single structure, it must have following fields.

- Labels – Cell array of character vectors specifying row labels listed in the RowLabels property.

- `Colors` – Character vector or string specifying a color for the row labels. If this field is empty, the default color (black) is used.

For a structure array, each structure must have a single element in each field.

- `Labels` – Character vector or string specifying a row label listed in the `RowLabels` property.
- `Colors` – Character vector or string specifying a color for the row labels. If this field is empty, the default color (black) is used.

For more information on specifying colors, see “Color Options” on page 1-1092.

`LabelsWithMarkers` — Flag to display colored markers for row and column labels
`false` (default) | `true`

Flag to display colored markers instead of colored text for the row and column labels, specified as `true` or `false`.

Example: `true`

Data Types: `logical`

Object Functions

view	Display heatmap or clustergram
plot	Render heatmap or clustergram
addTitle	Add title to heatmap or clustergram
addXLabel	Label x-axis of heatmap or clustergram
addYLabel	Label y-axis of heatmap or clustergram

Examples

Plot Heatmap of Data Matrix

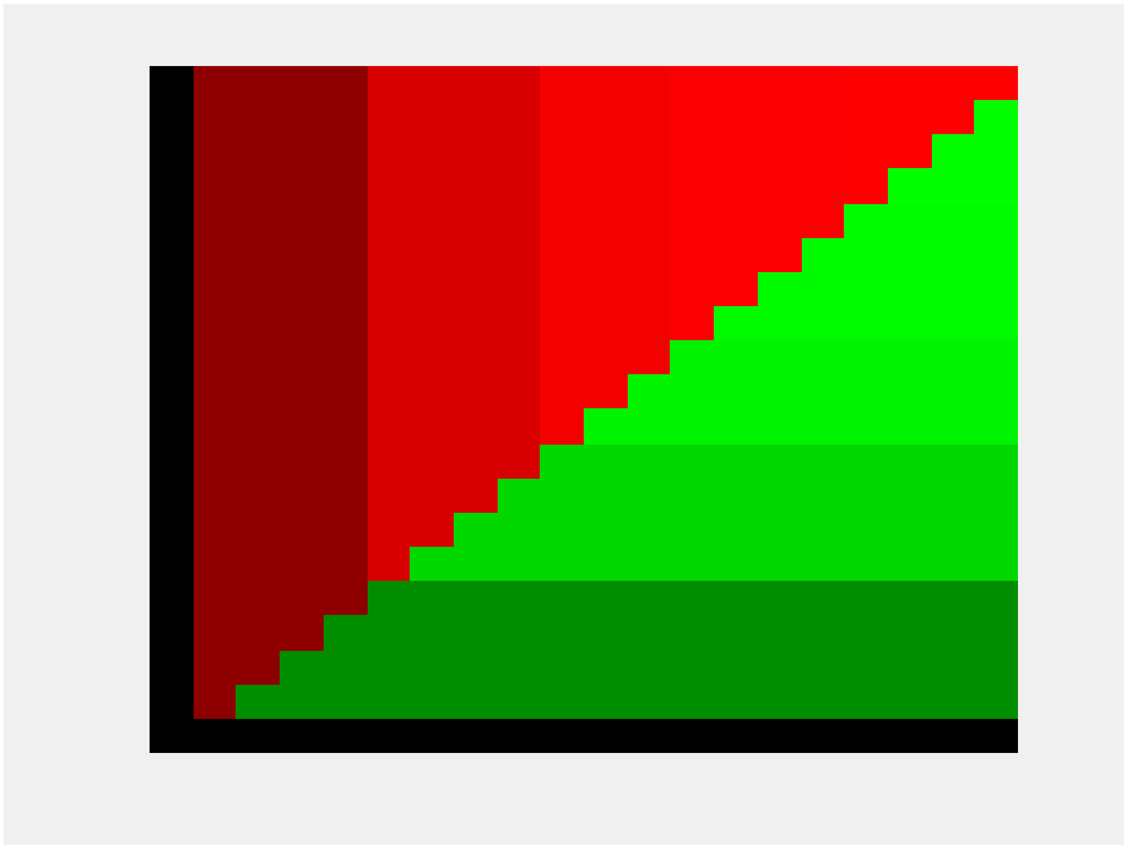
Create a matrix of data.

```
data = gallery('invhess',20);
```

Display a 2-D color heatmap of the data.

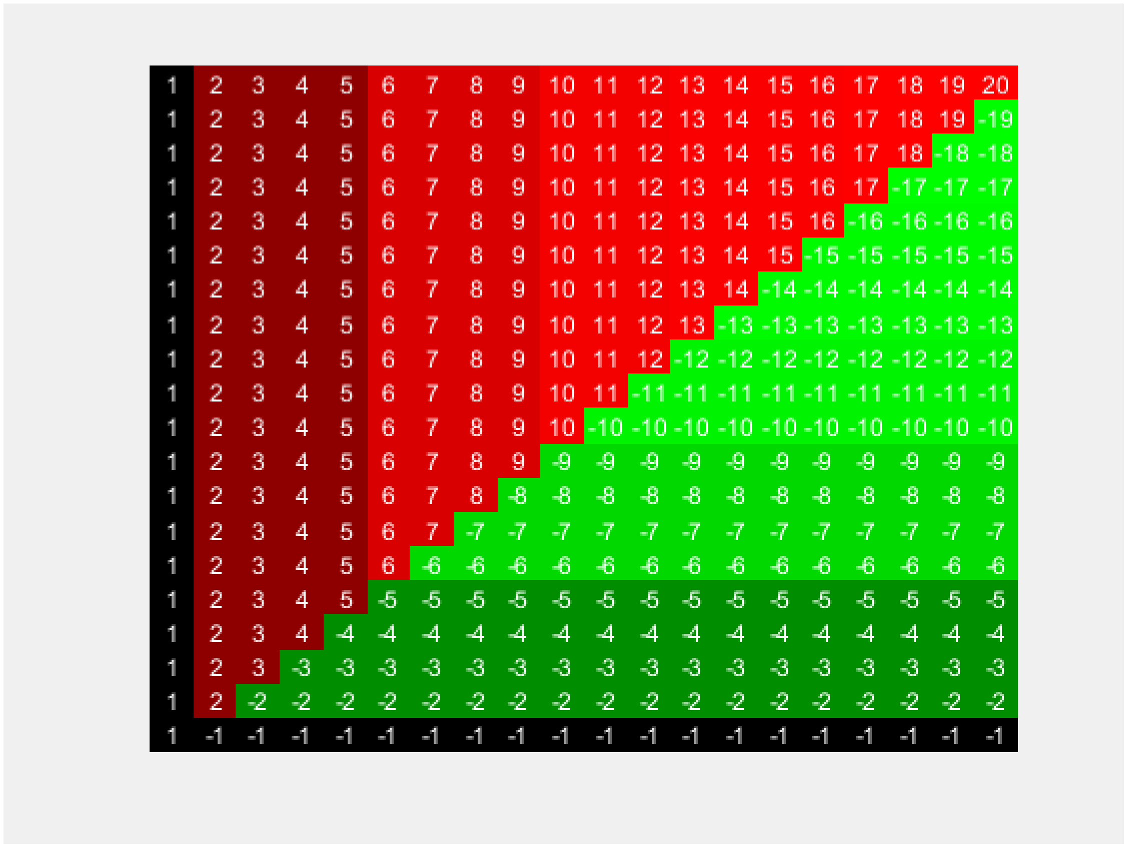
```
hmo = HeatMap(data);
```

```
             Standardize: '[column | row | {none}]'
               Symmetric: '[true | false].'
            DisplayRange: 'Scalar.'
                Colormap: []
               ImputeFun: 'string -or- function handle -or- cell array'
            ColumnLabels: 'Cell array of strings, or an empty cell array'
               RowLabels: 'Cell array of strings, or an empty cell array'
      ColumnLabelsRotate: []
         RowLabelsRotate: []
                Annotate: '[on | {off}]'
          AnnotPrecision: []
              AnnotColor: []
       ColumnLabelsColor: 'A structure array.'
          RowLabelsColor: 'A structure array.'
       LabelsWithMarkers: '[true | false].'
    ColumnLabelsLocation: '[ top | {bottom} ]'
       RowLabelsLocation: '[ {left} | right ]'
```



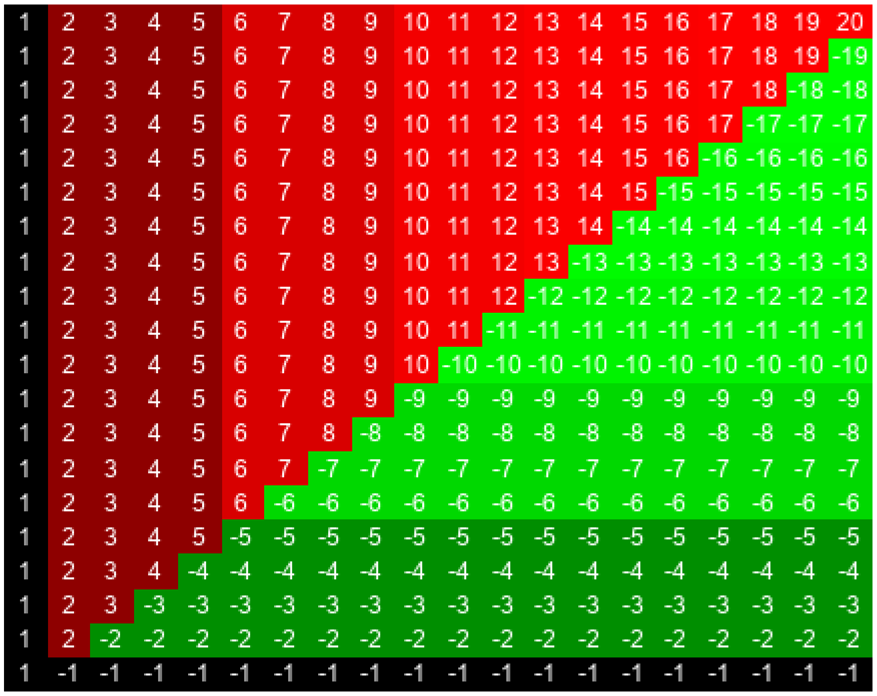
Display the data values in the heatmap.

```
hmo.Annotate = true;
view(hmo)
```



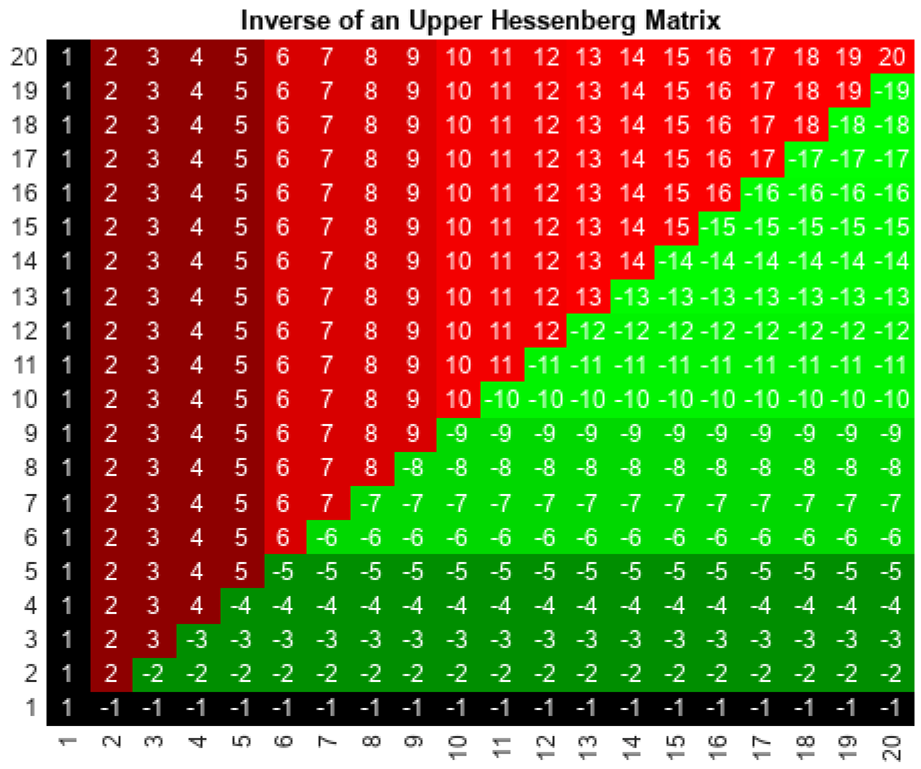
Use the `plot` function to display the heatmap in another figure specified by the figure handle `fH`.

```
fH = figure;
hA = plot(hmo,fH);
```



Use the returned axes handle `hA` to specify the axes properties.

```
hA.Title.String = 'Inverse of an Upper Hessenberg Matrix';
hA.XTickLabelMode = 'auto';
hA.YTickLabelMode = 'auto';
```



Add Custom Title and Labels to Heatmap

Load a sample of gene expression data.

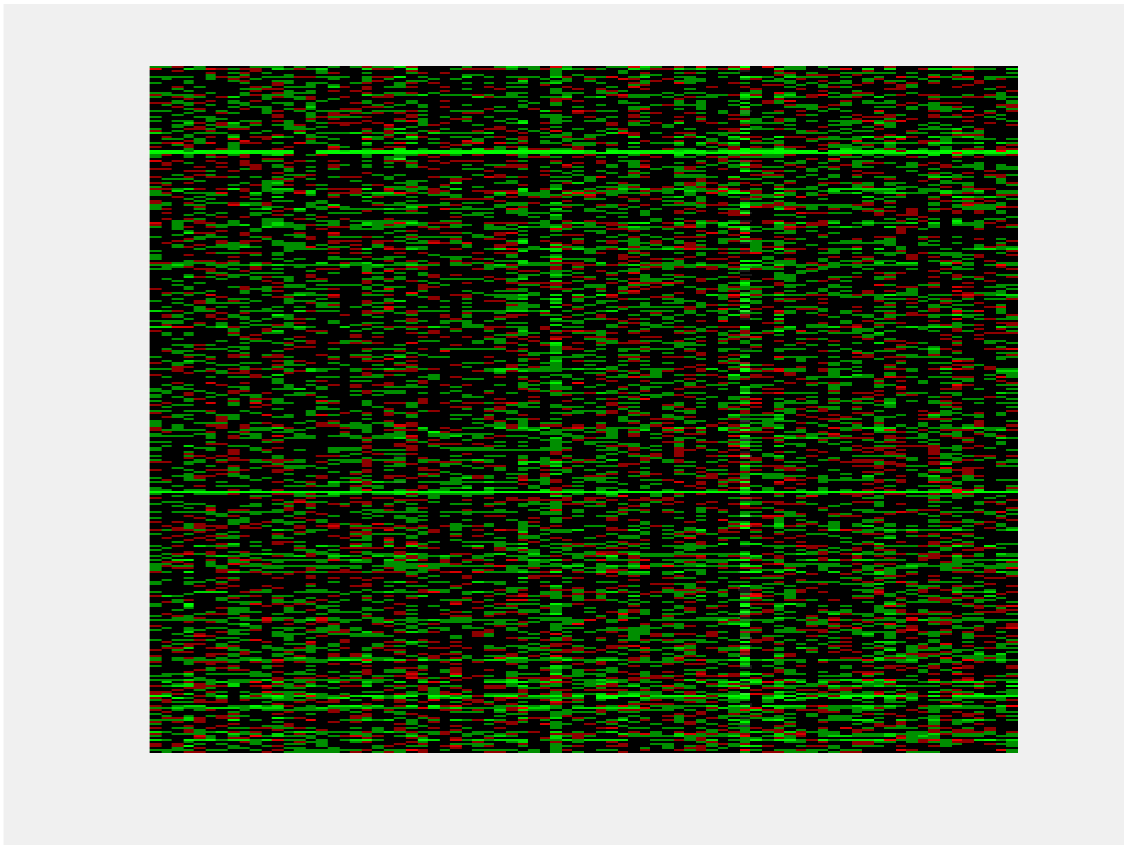
```
load bc_train_filtered
```

Display a heatmap of the gene expression values for 4918 genes from 78 samples.

```
hmo = HeatMap(bcTrainData.Log10Ratio);
```

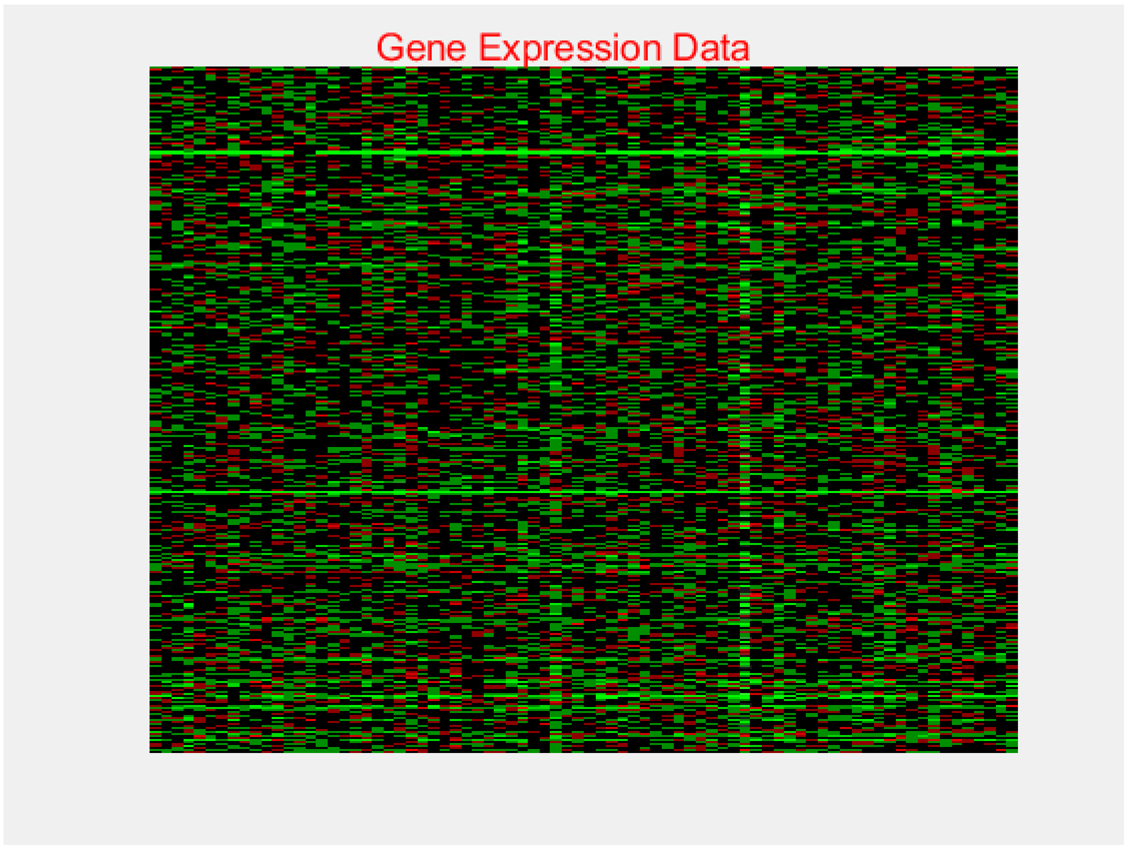
```
          Standardize: '[column | row | {none}]'
            Symmetric: '[true | false].'
         DisplayRange: 'Scalar.'
             Colormap: []
            ImputeFun: 'string -or- function handle -or- cell array'
         ColumnLabels: 'Cell array of strings, or an empty cell array'
            RowLabels: 'Cell array of strings, or an empty cell array'
  ColumnLabelsRotate: []
     RowLabelsRotate: []
            Annotate: '[on | {off}]'
       AnnotPrecision: []
          AnnotColor: []
   ColumnLabelsColor: 'A structure array.'
      RowLabelsColor: 'A structure array.'
   LabelsWithMarkers: '[true | false].'
```

```
    ColumnLabelsLocation: '[ top | {bottom} ]'
       RowLabelsLocation: '[ {left} | right ]'
```



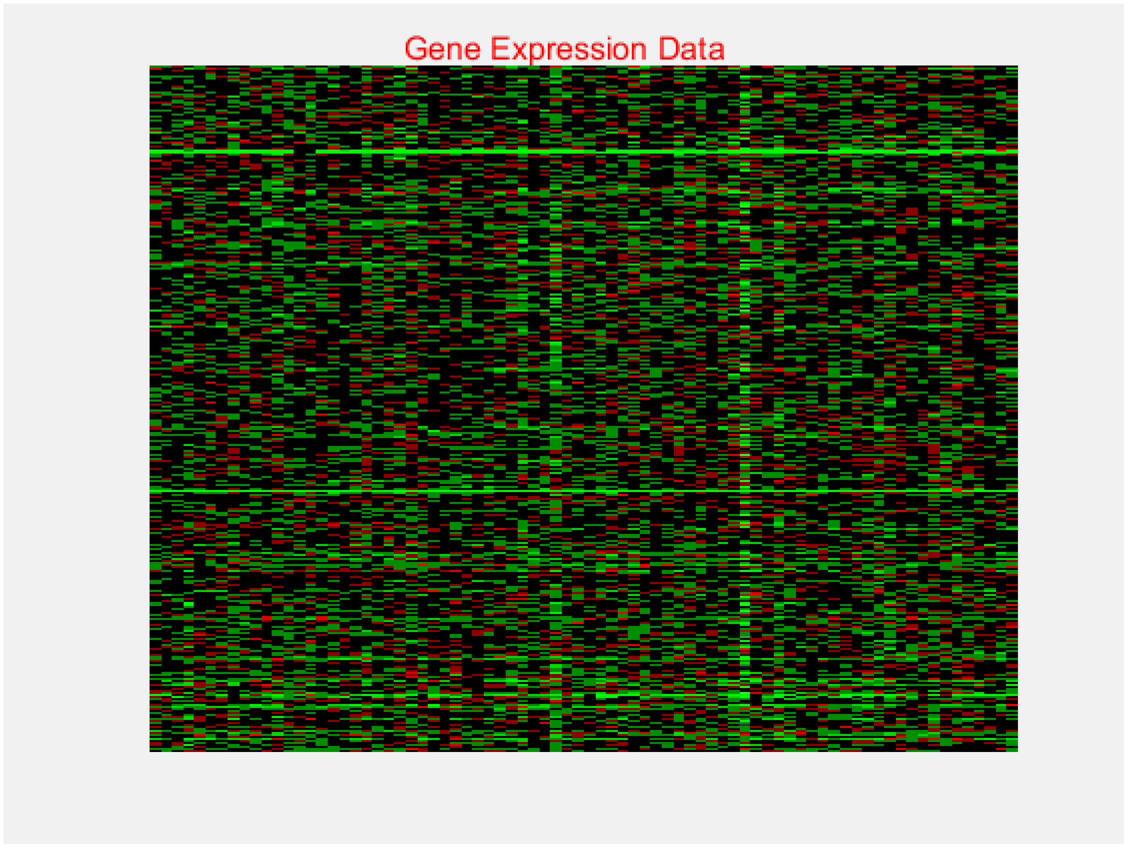
Add a title to the heatmap in red.

```
title = addTitle(hmo,'Gene Expression Data','Color','red');
```



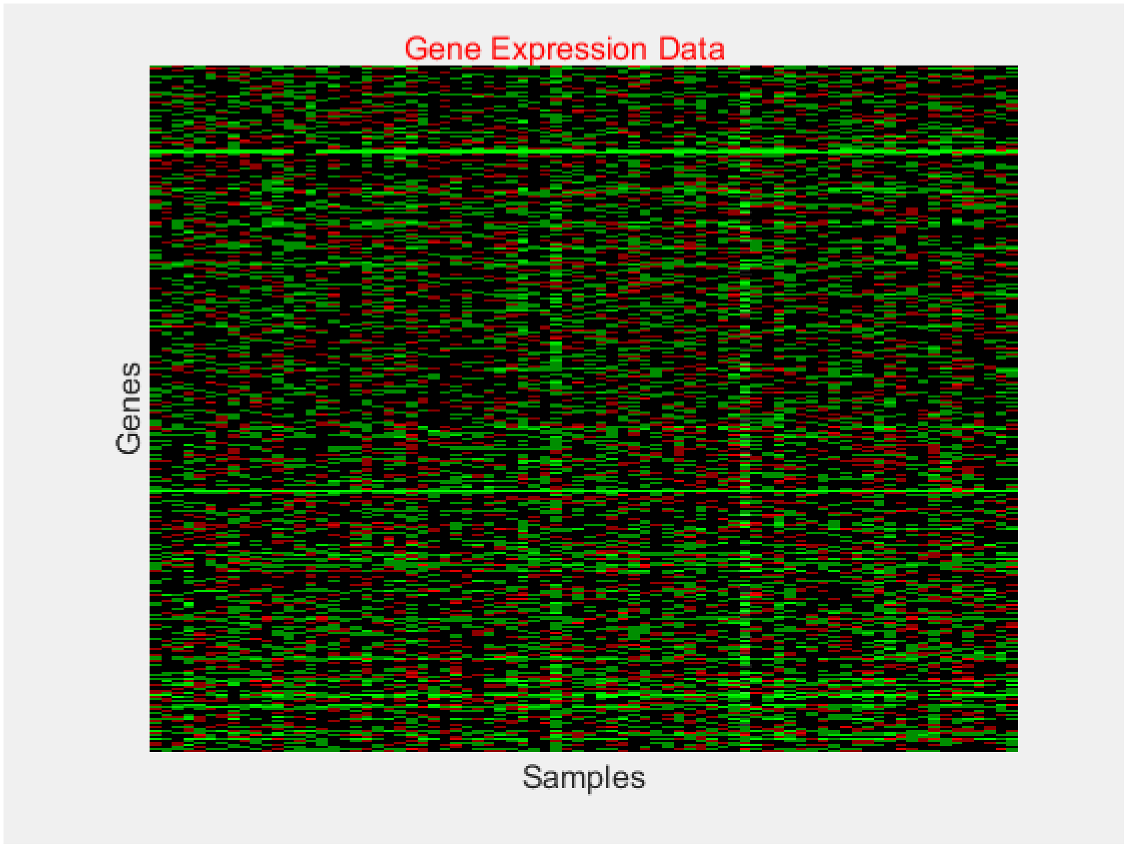
Change the title font size.

```
title.FontSize = 12;
```



Add labels to the x-axis and y-axis.

```
addXLabel(hmo,'Samples','FontSize',12);
addYLabel(hmo,'Genes','FontSize',12);
```



More About

Color Options

The following lists the predefined colors and their RGB triplet equivalents. The short names and long names are character vectors that specify one of eight preset colors. The RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color; the intensities must be in the range [0 1].

RGB Triplet	Short Name	Long Name
`[1 1 0]`	`y`	`yellow`
`[1 0 1]`	`m`	`magenta`
`[0 1 1]`	`c`	`cyan`
`[1 0 0]`	`r`	`red`
`[0 1 0]`	`g`	`green`
`[0 0 1]`	`b`	`blue`
`[1 1 1]`	`w`	`white`
`[0 0 0]`	`k`	`black`

Version History

Introduced in R2009b

See Also

`redbluecmap` | `redgreencmap` | `clustergram`

hmmprofalign

Align query sequence to profile using hidden Markov model alignment

Syntax

```
Score = hmmprofalign(Model, Seq)
[Score, Alignment] = hmmprofalign(Model, Seq)
[Score, Alignment, Pointer] = hmmprofalign(Model, Seq)

hmmprofalign(..., 'ShowScore', ShowScoreValue, ...)
hmmprofalign(..., 'Flanks', FlanksValue, ...)
hmmprofalign(..., 'ScoreFlanks', ScoreFlanksValue, ...)
hmmprofalign(..., 'ScoreNullTransitions', ScoreNullTransitionsValue, ...)
```

Arguments

Model	Hidden Markov model created with the function `hmmprofstruct`.
Seq	Amino acid or nucleotide sequence. You can also enter a structure with the field `Sequence`.
ShowScoreValue	Controls the display of the scoring space and the winning path. Choices are `true` or `false` (default).
FlanksValue	Controls the inclusion of the symbols generated by the FLANKING INSERT states in the output sequence. Choices are `true` or `false` (default).
ScoreFlanksValue	Controls the inclusion of the transition probabilities for the flanking states in the raw score. Choices are `true` or `false` (default).
ScoreNullTransitionsValue	Controls the adjustment of the raw score using the null model for transitions (`Model.NullX`). Choices are `true` or `false` (default).

Description

`Score = hmmprofalign(Model, Seq)` returns the score for the optimal alignment of the query amino acid or nucleotide sequence (`Seq`) to the profile hidden Markov model (`Model`). Scores are computed using log-odd ratios for emission probabilities and log probabilities for state transitions.

`[Score, Alignment] = hmmprofalign(Model, Seq)` also returns a character vector showing the optimal profile alignment.

Uppercase letters and dashes correspond to MATCH and DELETE states respectively (the combined count is equal to the number of states in the model). Lowercase letters are emitted by the INSERT states. For more information about the HMM profile, see `hmmprofstruct`.

`[Score, Alignment, Pointer] = hmmprofalign(Model, Seq)` also returns a vector of the same length as the profile model with indices pointing to the respective symbols of the query sequence. Null pointers (`NaN`) mean that such states did not emit a symbol in the aligned sequence because they represent model jumps from the BEGIN state of a MATCH state, model jumps from the from a MATCH state to the END state, or because the alignment passed through DELETE states.

`hmmprofalign(..., '`*`PropertyName`*`', `*`PropertyValue`*`, ...)` calls `hmmprofalign` with optional properties that use property name/property value pairs. You can specify one or more properties in any order. Each *`PropertyName`* must be enclosed in single quotation marks and is case insensitive. These property name/property value pairs are as follows:

`hmmprofalign(..., 'ShowScore', `*`ShowScoreValue`*`, ...)`, when *`ShowScoreValue`* is `true`, displays the scoring space and the winning path.

`hmmprofalign(..., 'Flanks', `*`FlanksValue`*`, ...)`, when *`FlanksValue`* is `true`, includes the symbols generated by the FLANKING INSERT states in the output sequence.

`hmmprofalign(..., 'ScoreFlanks', `*`ScoreFlanksValue`*`, ...)`, when *`ScoreFlanksValue`* is `true`, includes the transition probabilities for the flanking states in the raw score.

`hmmprofalign(..., 'ScoreNullTransitions', `*`ScoreNullTransitionsValue`*`, ...)`, when *`ScoreNullTransitionsValue`* is `true`, adjusts the raw score using the null model for transitions (`Model.NullX`).

Note Multiple target alignment is not supported in this implementation. All the `Model.LoopX` probabilities are ignored.

Examples

Align query sequence to profile using HMM model alignment

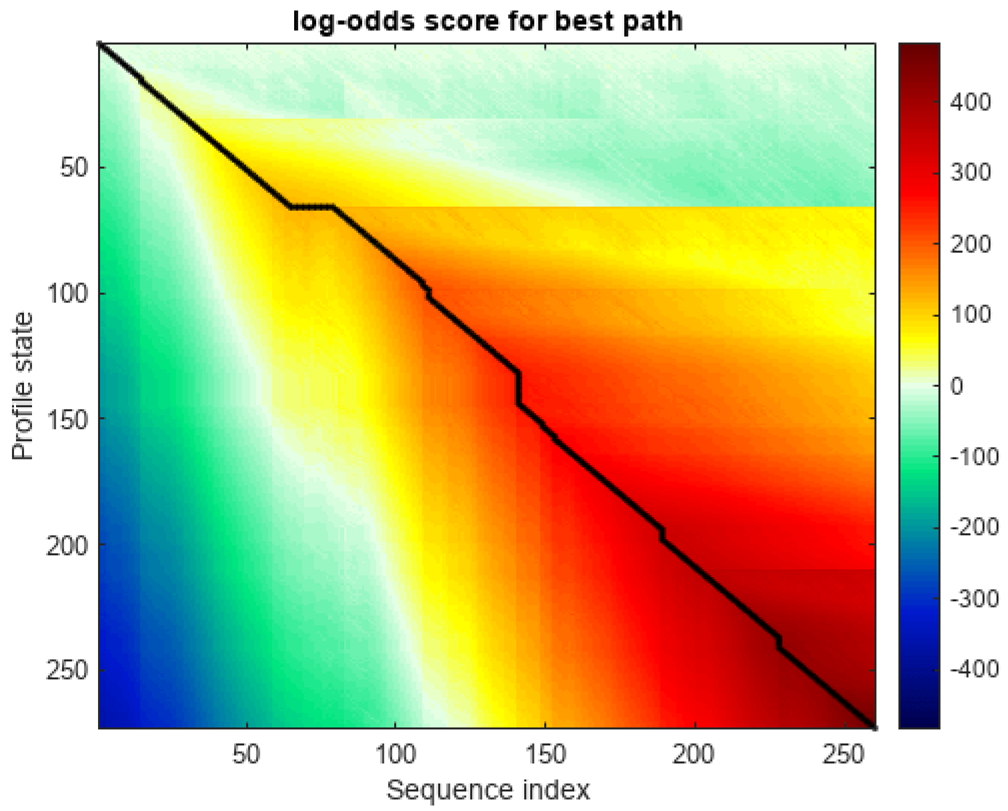
This example shows how to align a query sequence to a HMM model profile using HMM model alignment.

Load the HMM profile structure of the 7 transmembrane receptor (Secretin family).

```
load('hmm_model_examples','model_7tm_2');
```

Load an example sequence and align it to the profile structure using the HMM alignment.

```
load('hmm_model_examples','sequences');
humanSeq = sequences(1).Sequence;
[a,s]=hmmprofalign(model_7tm_2,humanSeq,'showscore',true)
```



```
a = 483.7231

s =
'YILVKAIYTLGYSVS-LMSLATGSIILCLFRKLHCTRNYIHLNLFLSFILRAISVLVKDDVLYSSSgtlhcpdqpsswvgCKLSLVFLQYCIMANI
```

Version History

Introduced before R2006a

See Also

`gethmmprof` | `hmmprofestimate` | `hmmprofgenerate` | `hmmprofgenerate` | `hmmprofstruct` | `pfamhmmread` | `showhmmprof` | `multialign` | `profalign`

hmmprofestimate

Estimate profile hidden Markov model (HMM) parameters using pseudocounts

Syntax

```
hmmprofestimate(Model, MultipleAlignment,'PropertyName',PropertyValue,...)

hmmprofestimate(..., 'A', AValue)

hmmprofestimate(..., 'Ax', AxValue)

hmmprofestimate(..., 'BE', BEValue)

hmmprofestimate(..., 'BDx', BDxValue)
```

Arguments

`Model`	Hidden Markov model created with the function `hmmprofstruc`.
`MultipleAlignment`	Array of sequences. `Sequences` can also be a structured array with the aligned sequences in a field `Aligned` or `Sequences` and the optional names in a field `Header` or `Name`.
`A`	Property to set the pseudocount weight A. Default value is `20`.
`Ax`	Property to set the pseudocount weight Ax. Default value is 20.
`BE`	Property to set the background symbol emission probabilities. Default values are taken from `Model.NullEmission`.
`BMx`	Property to set the background transition probabilities from any MATCH state ([`M->M` `M->I` `M->D`]). Default values are taken from `hmmprofstruct`.
`BDx`	Property to set the background transition probabilities from any DELETE state ([`D->M` `D->D`]). Default values are taken from `hmmprofstruct`.

Description

`hmmprofestimate(Model, MultipleAlignment, 'PropertyName', PropertyValue...)` returns a structure with the fields containing the updated estimated parameters of a profile HMM. Symbol emission and state transition probabilities are estimated using the real counts and weighted pseudocounts obtained with the background probabilities. Default weight is `A=20`, the default background symbol emission for match and insert states is taken from `Model.NullEmission`, and the default background transition probabilities are the same as default transition probabilities returned by `hmmprofstruct`.

Model Construction: Multiple aligned sequences should contain uppercase letters and dashes indicating the model MATCH and DELETE states agreeing with `Model.ModelLength`. If model state annotation is missing, but `MultipleAlignment` is space aligned, then a "maximum entropy" criteria is used to select `Model.ModelLength` states.

Note Insert and flank insert transition probabilities are not estimated, but can be modified afterwards using `hmmprofstruct`.

`hmmprofestimate(..., 'A',` *`AValue`*`)` sets the pseudocount weight `A = Avalue` when estimating the symbol emission probabilities. Default value is `20`.

`hmmprofestimate(...,'Ax',` *`AxValue`*`)` sets the pseudocount weight `Ax = Axvalue` when estimating the transition probabilities. Default value is 20.

`hmmprofestimate(...,'BE',` *`BEValue`*`)` sets the background symbol emission probabilities. Default values are taken from `Model.NullEmission`.

`hmmprofestimate(...,'BMx',` *`BMxValue`*`)` sets the background transition probabilities from any MATCH state ([`M->M` `M->I` `M->D`]). Default values are taken from `hmmprofstruct`.

`hmmprofestimate(..., 'BDx',` *`BDxValue`*`)` sets the background transition probabilities from any DELETE state ([`D->M` `D->D`]). Default values are taken from `hmmprofstruct`.

Version History

Introduced before R2006a

See Also

`hmmprofalign` | `hmmprofstruct` | `showhmmprof`

hmmprofgenerate

Generate random sequence drawn from profile hidden Markov model (HMM)

Syntax

```
Sequence = hmmprofgenerate(Model)
[Sequence, Profptr] = hmmprofgenerate(Model)

... = hmmprofgenerate(Model, ...'Align', AlignValue, ...)
... = hmmprofgenerate(Model, ...'Flanks', FlanksValue, ...)
... = hmmprofgenerate(Model, ...'Signature', SignatureValue, ...)
```

Arguments

Model	Hidden Markov model created with the `hmmprofstruct` function.
AlignValue	Controls the use of uppercase letters for matches and lowercase letters for inserted letters. Choices are `true` or `false` (default).
FlanksValue	Controls the inclusion of the symbols generated by the FLANKING INSERT states in the output sequence. Choices are `true` or `false` (default).
SignatureValue	Controls the return of the most likely path and symbols. Choices are `true` or `false` (default).

Description

`Sequence` `= hmmprofgenerate(`*`Model`*`)` returns a sequence of amino acids or nucleotides drawn from the profile *`Model`*. The length, alphabet, and probabilities of the *`Model`* are stored in a structure. For more information about this structure, see `hmmprofstruct`.

`[`*`Sequence`*`, `*`Profptr`*`] = hmmprofgenerate(`*`Model`*`)` returns a vector of the same length as the profile model pointing to the respective states in the output sequence. Null pointers (0) mean that such states do not exist in the output sequence, either because they are never touched (i.e., jumps from the BEGIN state to MATCH states or from MATCH states to the END state), or because DELETE states are not in the output sequence (not aligned output; see below).

`... = hmmprofgenerate(`*`Model`*`, ...'`*`PropertyName`*`', `*`PropertyValue`*`, ...)` calls `hmmprofgenerate` with optional properties that use property name/property value pairs. You can specify one or more properties in any order. Each *`PropertyName`* must be enclosed in single quotes and is case insensitive. These property name/property value pairs are as follows:

`... = hmmprofgenerate(`*`Model`*`, ...'Align', `*`AlignValue`*`, ...)` if `Align` is `true`, the output sequence is aligned to the model as follows: uppercase letters and dashes correspond to MATCH and DELETE states respectively (the combined count is equal to the number of states in the model). Lowercase letters are emitted by the INSERT or FLANKING INSERT states. If *`AlignValue`* is `false`, the output is a sequence of uppercase symbols. The default value is `true`.

`... = hmmprofgenerate(Model, ...'Flanks', FlanksValue, ...)` if `Flanks` is `true`, the output sequence includes the symbols generated by the FLANKING INSERT states. The default value is `false`.

`... = hmmprofgenerate(Model, ...'Signature', SignatureValue, ...)` if *`SignatureValue`* is `true`, returns the most likely path and symbols. The default value is `false`.

Examples

```
load('hmm_model_examples','model_7tm_2') % load a model example
rand_sequence = hmmprofgenerate(model_7tm_2)
```

Version History

Introduced before R2006a

See Also

`hmmprofalign` | `hmmprofstruct` | `showhmmprof`

hmmprofmerge

Displays a set of HMM profile alignments

Syntax

```
hmmprofmerge(Sequences)
hmmprofmerge(Sequences, Names)
hmmprofmerge(Sequences, Names, Scores)
```

Arguments

Sequences	Array of sequences. *Sequences* can also be a structured array with the aligned sequences in a field `Aligned` or `Sequences`, and the optional names in a field `Header` or `Name`.
Names	Names for the sequences. Enter a cell array of character vectors.
Scores	Pairwise alignment scores from the function `hmmprofalign`. Enter a vector of values with the same length as the number of sequences in *Sequences*.

Description

`hmmprofmerge(`*`Sequences`*`)` opens your default Web browser and displays a set of prealigned sequences to an HMM model profile. The output is aligned corresponding to the HMM states.

- **Match states** — Uppercase letters
- **Insert states** — Lowercase letters or asterisks (*)
- **Delete states** — Dashes

Periods (.) are added at positions corresponding to inserts in other sequences. The input sequences must have the same number of profile states, that is, the joint count of capital letters and dashes must be the same.

`hmmprofmerge(`*`Sequences`*`, `*`Names`*`)` labels the sequences with `Names`.

`hmmprofmerge(`*`Sequences`*`, `*`Names`*`, `*`Scores`*`)` sorts the displayed sequences using `Scores`.

Examples

```
load('hmm_model_examples','model_7tm_2')  %load model
load('hmm_model_examples','sequences')  %load sequences

for ind =1:length(sequences)
    [scores(ind),sequences(ind).Aligned] =...
        hmmprofalign(model_7tm_2,sequences(ind).Sequence);
    end
hmmprofmerge(sequences, scores)
```

Version History

Introduced before R2006a

See Also

`hmmprofalign` | `hmmprofstruct`

hmmprofstruct

Create or edit hidden Markov model (HMM) profile structure

Syntax

```
Model = hmmprofstruct(Length)
Model = hmmprofstruct(Length, Field1, Field1Value, Field2, Field2Value, ...)
NewModel = hmmprofstruct(Model, Field1, Field1Value, Field2,
Field2Value, ...)
```

Input Arguments

Length	Number of match states in the model.
Model	MATLAB structure containing fields for the parameters of an HMM profile created with the `hmmprofstruct` function.
Field	Character vector or string containing a field name in the structure *Model*. See the table below for field names.
FieldValue	Value associated with *Field*. See the table below for descriptions.

Output Arguments

Model	MATLAB structure containing fields for the parameters of an HMM profile.

Description

Model = `hmmprofstruct(`*Length*`)` returns *Model*, a MATLAB structure containing fields for the parameters of an HMM profile. *Length* specifies the number of match states in the model. All other required parameters are set to the default values.

Model = `hmmprofstruct(`*Length*, *Field1*, *Field1Value*, *Field2*, *Field2Value*, ...`)` returns an HMM profile structure using the specified parameters. All other required parameters are set to default values.

NewModel = `hmmprofstruct(`*Model*, *Field1*, *Field1Value*, *Field2*, *Field2Value*, ...`)` returns an updated HMM profile structure using the specified parameters. All other parameters are taken from the input *Model*.

HMM Profile Structure

The MATLAB structure *Model* contains the following fields, which are the required and optional parameters of an HMM profile. All probability values are in the [0 1] range.

Field	Description
`ModelLength`	Integer specifying the length of the profile (number of MATCH states).

Field	Description
`Alphabet`	Character vector or string specifying the alphabet used in the model. Choices are `'AA'` (default) or `'NT'`. **Note** `AlphaLength` is 20 for `'AA'` and 4 for `'NT'`.
`MatchEmission`	Symbol emission probabilities in the MATCH states. Either of the following: • A matrix of size `ModelLength`-by-`AlphaLength`, where each row corresponds to the emission distribution for a specific MATCH state. Defaults to uniform distributions. • A structure containing residue counts, such as returned by `aacount` or `basecount`.
`InsertEmission`	Symbol emission probabilities in the INSERT state. Either of the following: • A matrix of size `ModelLength`-by-`AlphaLength`, where each row corresponds to the emission distribution for a specific INSERT state. Defaults to uniform distributions. • A structure containing residue counts, such as returned by `aacount` or `basecount`.
`NullEmission`	Symbol emission probabilities in the MATCH and INSERT states for the NULL model. Either of the following: • A 1-by-`AlphaLength` row vector. Defaults to a uniform distribution. • A structure containing residue counts, such as returned by `aacount` or `basecount`. **Note** The NULL model is used to compute the log-odds ratio at every state and avoid overflow when propagating the probabilities through the model. **Note** NULL probabilities are also known as the background probabilities.

Field	Description

`BeginX`	BEGIN state transition probabilities.
	
	Format is a 1-by-(`ModelLength + 1`) row vector:
	
	`[B->D1 B->M1 B->M2 B->M3 .... B->Mend]`
	
	Note If necessary, `hmmprofstruct` will normalize the data such that the sum of the transition probabilities from the BEGIN state equals 1:
	
	`sum(`*`Model`*`.BeginX) = 1`
	
	For fragment profiles:
	
	`sum(`*`Model`*`.BeginX(3:end)) = 0`
	
	Default is `[0.01 0.99 0 0 ... 0]`.

`MatchX`	MATCH state transition probabilities.
	
	Format is a 4-by-(`ModelLength - 1`) matrix:
	
	`[M1->M2 M2->M3 ... M[end-1]->Mend;`
	` M1->I1 M2->I2 ... M[end-1]->I[end-1];`
	` M1->D2 M2->D3 ... M[end-1]->Dend;`
	` M1->E  M2->E  ... M[end-1]->E  ]`
	
	Note If necessary, `hmmprofstruct` will normalize the data such that the sum of the transition probabilities from every MATCH state equals 1:
	
	`sum(`*`Model`*`.MatchX) = [ 1 1 ... 1 ]`
	
	For fragment profiles:
	
	`sum(`*`Model`*`.MatchX(4,:)) = 0`
	
	Default is `repmat([0.998 0.001 0.001 0],ModelLength-1,1)`.

Field	Description

`InsertX`	INSERT state transition probabilities.
	
	Format is a 2-by-(`ModelLength - 1`) matrix:
	
	`[ I1->M2 I2->M3 ... I[end-1]->Mend;`
	`  I1->I1 I2->I2 ... I[end-1]->I[end-1] ]`
	
	Note If necessary, `hmmprofstruct` will normalize the data such that the sum of the transition probabilities from every INSERT state equals 1:
	
	`sum(`*`Model`*`.InsertX) = [ 1 1 ... 1 ]`
	
	Default is `repmat([0.5 0.5],ModelLength-1,1)`.

`DeleteX`	DELETE state transition probabilities.
	
	Format is a 2-by-(`ModelLength - 1`) matrix:
	
	`[ D1->M2 D2->M3 ... D[end-1]->Mend ;`
	`  D1->D2 D2->D3 ... D[end-1]->Dend ]`
	
	Note If necessary, `hmmprofstruct` will normalize the data such that the sum of the transition probabilities from every DELETE state equals 1:
	
	`sum(`*`Model`*`.DeleteX) = [ 1 1 ... 1 ]`
	
	Default is `repmat([0.5 0.5],ModelLength-1,1)`.

Field	Description

`FlankingInsertX`	Flanking insert states (N and C) used for LOCAL profile alignment.
	
	Format is a 2-by-2 matrix:
	
	`[N->B  C->T ;`
	` N->N  C->C]`
	
	Note If necessary, `hmmprofstruct` will normalize the data such that the sum of the transition probabilities from Flanking Insert states equals 1:
	
	`sum(`*`Model`*`.FlankingInsertsX) = [1 1]`
	
	Note To force global alignment use:
	
	`Model``.FlankingInsertsX = [1 1; 0 0]`
	
	Default is `[0.01 0.01; 0.99 0.99]`.

`LoopX`	Loop states transition probabilities used for multiple hits alignment.
	
	Format is a 2-by-2 matrix:
	
	`[E->C  J->B ;`
	` E->J  J->J]`
	
	Note If necessary, `hmmprofstruct` will normalize the data such that the sum of the transition probabilities from Loop states equals 1:
	
	`sum(`*`Model`*`.LoopX) = [1 1]`
	
	Default is `[0.5 0.01; 0.5 0.99]`.

`NullX`	Null transition probabilities used to provide scores with log-odds values also for state transitions.
	
	Format is a 2-by-1 column vector:
	
	`[G->F ; G->G]`
	
	Note If necessary, `hmmprofstruct` will normalize the data such that the sum of the transition probabilities from Null states equals 1:
	
	`sum(`*`Model`*`.NullX) = 1`
	
	Default is `[0.01; 0.99]`.

Field	Description
`IDNumber`	Optional. User-assigned identification number.
`Description`	Optional. User-assigned description of the model.

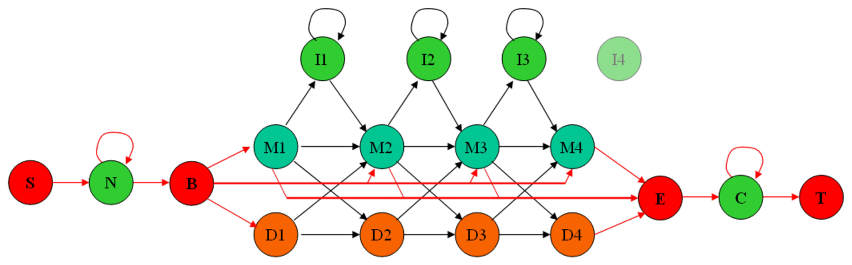
HMM Profile Model

An HMM profile model is a common statistical tool for modeling structured sequences composed of symbols. These symbols include randomness in both the output (emission of symbols) and the state transitions of the process. Markov models are generally represented by state diagrams.

The following figure is a state diagram for an HMM profile of length four. INSERT, MATCH, and DELETE states are in the center section.

- INSERT state represents the excess of one or more symbols in the target sequence that are not included in the profile.
- MATCH state means that the target sequence is aligned to the profile at the specific location.
- DELETE state represents a gap or symbol absence in the target sequence (also known as a silent state because it does not emit any symbols).

Flanking states (`S`, `N`, `B`, `E`, `C`, `T`) are used for proper modeling of the ends of the sequence, either for global, local or fragment alignment of the profile. `S`, `B`, `E`, and `T` are silent, while `N` and `C` are used to insert symbols at the flanks.



Examples

Example 1.16. Creating an HMM Profile Structure

Create an HMM profile structure with 100 MATCH states, using the amino acid alphabet.

```
hmmProfile = hmmprofstruct(100,'Alphabet','AA')
```

```
hmmProfile = 

        ModelLength: 100
           Alphabet: 'AA'
      MatchEmission: [100x20 double]
     InsertEmission: [100x20 double]
       NullEmission: [1x20 double]
             BeginX: [101x1 double]
             MatchX: [99x4 double]
            InsertX: [99x2 double]
            DeleteX: [99x2 double]
    FlankingInsertX: [2x2 double]
```

```
             LoopX: [2x2 double]
             NullX: [2x1 double]
```

Example 1.17. Editing an HMM Profile Structure

1. Use the `pfamhmmread` function to create an HMM profile structure from `pf00002.ls`, a PFAM HMM-formatted file included with the software.

   ```
   hmm02 = pfamhmmread('pf00002.ls');
   ```
2. Modify the HMM profile structure to force a global alignment by setting the looping transition probabilities in the flanking insert states to zero.

   ```
   hmm02 = hmmprofstruct(hmm02,'FlankingInsertX',[0 0;1 1]);
   hmm02.FlankingInsertX

   ans =

        0     0
        1     1
   ```

Version History

Introduced before R2006a

See Also

`aacount` | `basecount` | `gethmmprof` | `hmmprofalign` | `hmmprofestimate` | `hmmprofgenerate` | `hmmprofmerge` | `pfamhmmread` | `showhmmprof`

horzcat (DataMatrix)

Concatenate DataMatrix objects horizontally

Syntax

```
DMObjNew = horzcat(DMObj1, DMObj2, ...)
DMObjNew = (DMObj1, DMObj2, ...)
DMObjNew = horzcat(DMObj1, B, ...)
DMObjNew = (DMObj1, B, ...)
```

Input Arguments

DMObj1, *DMObj2*	DataMatrix objects, such as created by `DataMatrix` (object constructor).
B	MATLAB numeric or logical array.

Output Arguments

DMObjNew	DataMatrix object created by horizontal concatenation.

Description

DMObjNew = `horzcat(`*DMObj1*, *DMObj2*, `...)` or the equivalent *DMObjNew* = (*DMObj1*, *DMObj2*, `...`) horizontally concatenates the DataMatrix objects *DMObj1* and *DMObj2* into *DMObjNew*, another DataMatrix object. *DMObj1* and *DMObj2* must have the same number of rows. The row names and the order of rows for *DMObjNew* are the same as *DMObj1*. The row names of *DMObj2* and any other DataMatrix object input arguments are not preserved. The columns names for *DMObjNew* are the column names of *DMObj1*, *DMObj2*, and other DataMatrix object input arguments.

DMObjNew = `horzcat(`*DMObj1*, *B*, `...)` or the equivalent *DMObjNew* = (*DMObj1*, *B*, `...`) horizontally concatenates the DataMatrix object *DMObj1* and a numeric or logical array *B* into *DMObjNew*, another DataMatrix object. *DMObj1* and *B* must have the same number of rows. The row names for *DMObjNew* are the same as *DMObj1*. The row names of *DMObj2* and any other DataMatrix object input arguments are not preserved. The column names for *DMObjNew* are the column names of *DMObj1* and empty for the columns from *B*.

MATLAB calls *DMObjNew* = `horzcat(`*X1*, *X2*, *X3*, `...)` for the syntax *DMObjNew* = [*X1*, *X2*, *X3*, `...`] when any one of *X1*, *X2*, *X3*, etc. is a DataMatrix object.

Version History

Introduced in R2008b

See Also

`DataMatrix` | `vertcat`

Topics

DataMatrix object on page 1-734

ilmnbslookup

Look up Illumina BeadStudio target (probe) sequence and annotation information

Syntax

```
AnnotStruct = ilmnbslookup(AnnotationFile, ID)
AnnotStruct = ilmnbslookup(AnnotationFile, ID, 'LookUpField',
LookUpFieldValue)
```

Input Arguments

AnnotationFile	Character vector or string specifying a file name or a path and file name of an Illumina® annotation file (CSV, BGX, or TXT format). If you specify only a file name, that file must be on the MATLAB search path or in the current folder. **Tip** You can download Illumina annotation files, such as `HumanRef-8_V3_0_R0_11282963_A.bgx`, from the Illumina Web site.
ID	Character vector, string, string vector, or cell array of character vectors representing a unique identifier(s) for one or more targets (probes) on an Illumina microarray. **Tip** By default, *ID* must match the `Search_key` field in *AnnotationFile*. However, you can use an identifier that corresponds to any of the fields in *AnnotationFile*, then set the `'LookUpField'` property appropriately. For example, if you want to look up annotation information for the targets (probes) on chromosome 7 only, set *ID* to `'7'`, then set *LookUpFieldValue* to `'Chromosome'`. For a list of all fields in *AnnotationFile*, see the following tables.
LookUpFieldValue	Field in *AnnotationFile* where `ilmnbslookup` looks for the specified *ID*. Default is the `Search_key` field. **Tip** Set this property so that it corresponds to the *ID* you use as input.

Output Arguments

AnnotStruct	Structure containing the probe sequence and annotation information for one or more targets (probes) specified by *ID*, and by *AnnotationFile*, an Illumina annotation file. *AnnotStruct* contains the same fields as *AnnotationFile*. The fields are described in the following two tables.

Description

AnnotStruct = ilmnbslookup(*AnnotationFile*, *ID*) returns *AnnotStruct*, a structure containing probe sequence and annotation information for one or more targets (probes) specified by *ID*, and by *AnnotationFile*, an Illumina annotation file (CSV, BGX, or TXT format).

AnnotStruct contains the same fields as *AnnotationFile*. The fields are described in the following two tables.

Structure Created from Illumina CSV Annotation File

Field	Description
Search_key	Internal identifier for the target, useful for custom design array
Target	Unique identifier for the target
ProbeId	Illumina probe identifier
Gid	GenBank identifier for the gene
Transcript	Illumina internal transcript identifier
Accession	GenBank accession number for the gene
Symbol	Typically, the gene symbol
Type	Probe type
Start	Starting position of the probe sequence in the GenBank record
Probe_Sequence	Sequence of the probe
Definition	Definition field from the GenBank record
Ontology	Gene Ontology terms associated with the gene
Synonym	Synonyms for the gene (from the GenBank record)

Structure Created from a BGX or TXT Annotation File

Field	Description
`Accession`	GenBank accession number for the gene
`Array_Address_Id`	Decoder identifier
`Chromosome`	Chromosome on which the gene is located
`Cytoband`	Cytogenetic banding region of the chromosome on which the gene associated with the target is located
`Definition`	Definition field from the GenBank record
`Entrez_Gene_ID`	Entrez Gene database identifier for the gene
`GI`	GenBank identifier for the gene
`ILMN_Gene`	Illuminainternal gene symbol
`Obsolete_Probe_Id`	Probe identifier before BGX annotation files
`Ontology_Component`	Gene Ontology cellular components associated with the gene
`Ontology_Function`	Gene Ontology molecular functions associated with the gene
`Ontology_Process`	Gene Ontology biological processes associated with the gene
`Probe_Chr_Orientation`	Orientation of the probe on the NCBI genome build
`Probe_Coordinates`	Genomic position of the probe on the NCBI genome build
`Probe_Id`	Illuminaprobe identifier
`Probe_Sequence`	Sequence of the probe
`Probe_Start`	Start position of the probe relative to the `5'` end of the source transcript sequence
`Probe_Type`	Information about what the probe is targeting
`Protein_Product`	NCBI protein accession number
`RefSeq_ID`	Identifier from the NCBI RefSeq database
`Reporter_Composite_map`	Information associated with control probes
`Reporter_Group_Name`	Information associated with control probes
`Reporter_Group_id`	Information associated with control probes
`Search_Key`	Internal identifier for the target, useful for custom design array
`Source`	Source from which the transcript sequence was obtained
`Source_Reference_ID`	Source's identifier
`Species`	Species associated with the gene
`Symbol`	Typically, the gene symbol
`Synonyms`	Synonyms for the gene (from the GenBank record)
`Transcript`	Illuminainternal transcript identifier
`Unigene_ID`	Identifier from the NCBI UniGene database

`AnnotStruct = ilmnbslookup(AnnotationFile, ID, 'LookUpField', LookUpFieldValue)` looks for *`ID`* in the annotation file in the field specified by *`LookUpFieldValue`*. Default is the `Search_key` field.

Examples

Note The gene expression file, `TumorAdjacent-probe-raw.txt`, and the annotation file, `HumanRef-8_V3_0_R0_11282963_A.bgx`, used in the following examples are not provided with the Bioinformatics Toolbox software.

Example 1.18. Look Up Annotation Information for a Single Target (Probe)

1. Read the contents of a tab-delimited file exported from the Illumina BeadStudio™ software into a MATLAB structure.

```
ilmnStruct = ilmnbsread('TumorAdjacent-probe-raw.txt')

ilmnStruct =

             Header: [1x1 struct]
           TargetID: {22184x1 cell}
        ColumnNames: {1x37 cell}
               Data: [22184x37 double]
    TextColumnNames: {1x23 cell}
           TextData: {22184x23 cell}
```

2. Find the number of the `Search_key` column in the `TextColumnNames` cell array, which is returned in the `ilmnStruct` structure by the `ilmnbsread` function.

```
srchCol = find(strcmpi('Search_Key',ilmnStruct.TextColumnNames))

srchCol =

     1
```

3. Look up the probe sequence and annotation information for the 10th entry in the annotation file, `HumanRef-8_V3_0_R0_11282963_A.bgx`.

```
annotation = ilmnbslookup('HumanRef-8_V3_0_R0_11282963_A.bgx',...
                           ilmnStruct.TextData{10,srchCol})
annotation =

                 Accession: 'NM_144670.2'
          Array_Address_Id: '0004050154'
                Chromosome: '12'
                  Cytoband: '12p13.31b'
                Definition: 'Homo sapiens alpha-2-macroglobulin-like 1 (A2ML1), mRNA.'
            Entrez_Gene_ID: '144568'
                        GI: '74271844'
                 ILMN_Gene: 'A2ML1'
         Obsolete_Probe_Id: ''
        Ontology_Component: ''
         Ontology_Function: 'endopeptidase inhibitor activity [goid 4866] [evidence IEA]'
          Ontology_Process: ''
      Probe_Chr_Orientation: '+'
         Probe_Coordinates: '8920412-8920461'
                  Probe_Id: 'ILMN_2136495'
            Probe_Sequence: 'TGTAATCGCAGCCCCTTGGAAGGCCAAGGCAGGAGAATCGCCTCAACACT'
               Probe_Start: '4889'
                Probe_Type: 'S'
           Protein_Product: 'NP_653271.2'
                 RefSeq_ID: 'NM_144670.2'
    Reporter_Composite_map: ''
       Reporter_Group_Name: ''
         Reporter_Group_id: ''
                Search_Key: 'ILMN_17375'
                    Source: 'RefSeq'
       Source_Reference_ID: 'NM_144670.2'
                   Species: 'Homo sapiens'
                    Symbol: 'A2ML1'
                  Synonyms: [1x141 char]
                Transcript: 'ILMN_17375'
                Unigene_ID: ''
```

Example 1.19. Look Up Annotation Information for a Subset of Targets (Probes)

Use the `ilmnbslookup` function with the `'LookUpField'` property to look up the annotation information for all targets located on chromosome 12 in the annotation file, `HumanRef-8_V3_0_R0_11282963_A.bgx`.

```
chr12annotation = ilmnbslookup('HumanRef-8_V3_0_R0_11282963_A.bgx',...
                               '12','LookUpField','Chromosome')

chr12annotation = 

                 Accession: {1x1186 cell}
          Array_Address_Id: {1x1186 cell}
                Chromosome: {1x1186 cell}
                  Cytoband: {1x1186 cell}
                Definition: {1x1186 cell}
            Entrez_Gene_ID: {1x1186 cell}
                        GI: {1x1186 cell}
                 ILMN_Gene: {1x1186 cell}
         Obsolete_Probe_Id: {1x1186 cell}
        Ontology_Component: {1x1186 cell}
         Ontology_Function: {1x1186 cell}
          Ontology_Process: {1x1186 cell}
     Probe_Chr_Orientation: {1x1186 cell}
         Probe_Coordinates: {1x1186 cell}
                  Probe_Id: {1x1186 cell}
            Probe_Sequence: {1x1186 cell}
               Probe_Start: {1x1186 cell}
                Probe_Type: {1x1186 cell}
           Protein_Product: {1x1186 cell}
                 RefSeq_ID: {1x1186 cell}
    Reporter_Composite_map: ''
       Reporter_Group_Name: ''
         Reporter_Group_id: ''
                Search_Key: {1x1186 cell}
                    Source: {1x1186 cell}
       Source_Reference_ID: {1x1186 cell}
                   Species: {1x1186 cell}
                    Symbol: {1x1186 cell}
                  Synonyms: {1x1186 cell}
                Transcript: {1x1186 cell}
                Unigene_ID: {1x1186 cell}
```

The output structure indicates that there are 1,186 targets located on chromosome 12.

Version History

Introduced in R2008a

See Also

`ilmnbsread`

ilmnbsread

Read gene expression data exported from Illumina BeadStudio software

Syntax

```
IlmnStruct = ilmnbsread(File)

IlmnStruct = ilmnbsread(File, ...'Columns', ColumnsValue, ...)
IlmnStruct = ilmnbsread(File, ...'HeaderOnly', HeaderOnlyValue, ...)
IlmnStruct = ilmnbsread(File, ...'CleanColNames', CleanColNamesValue, ...)
```

Input Arguments

File	Character vector or string specifying a file name or a path and file name of a tab-delimited file or comma-separated expression data file exported from Illumina BeadStudio software. If you specify only a file name, that file must be on the MATLAB search path or in the current folder.
ColumnsValue	Cell array that specifies the column names to read. Default is all column names.
HeaderOnlyValue	Controls the population of only the `Header`, `ColumnNames`, and `TextColumnNames` fields in *IlmnStruct*. Choices are `true` or `false` (default).
CleanColNamesValue	Controls the conversion of any `ColumnNames` containing spaces or characters that cannot be used as MATLAB variable names, to valid MATLAB variable names. Choices are `true` or `false` (default).

Output Arguments

IlmnStruct	MATLAB structure containing data exported from Illumina BeadStudio software.

Description

IlmnStruct = `ilmnbsread(`*File*`)` reads *File*, a tab-delimited or comma-separated expression data file exported from the Illumina BeadStudio software, and creates *IlmnStruct*, a MATLAB structure containing the following fields.

Field	Description
`Header`	Character vector containing a description of the data.
`TargetID`	Cell array containing unique identifiers for targets on an Illumina gene expression microarray.

Field	Description
`ColumnNames`	Cell array containing names of the columns that contain numeric data in the tab-delimited file exported from the Illumina BeadStudio software.
`Data`	Matrix containing numeric microarray data for each target on an Illumina gene expression microarray. **Note** `ColumnNames` and `Data` have the same number of columns.
`TextColumnNames`	Cell array containing names of the columns that contain nonnumeric data in the tab-delimited file exported from the Illumina BeadStudio software. This field can be empty.
`TextData`	Cell array containing nonnumeric microarray data (such as annotations) for each target on an Illumina gene expression microarray. This field can be empty. **Note** `TextColumnNames` and `TextData` have the same number of columns.

`*IlmnStruct* = ilmnbsread(*File*, ...'*PropertyName*', *PropertyValue*, ...)` calls `ilmnbsread` with optional properties that use property name/property value pairs. You can specify one or more properties in any order. Each *`PropertyName`* must be enclosed in single quotation marks and is case insensitive. These property name/property value pairs are as follows:

`*IlmnStruct* = ilmnbsread(*File*, ...'Columns', *ColumnsValue*, ...)` reads the data only from the columns specified by *`ColumnsValue`*, a cell array of column names. Default behavior is to read data from all columns.

`*IlmnStruct* = ilmnbsread(*File*, ...'HeaderOnly', *HeaderOnlyValue*, ...)` controls the population of only the `Header`, `ColumnNames`, and `TextColumnNames` fields in *`IlmnStruct`*. Choices are `true` or `false` (default).

`*IlmnStruct* = ilmnbsread(*File*, ...'CleanColNames', *CleanColNamesValue*, ...)` controls the conversion of any `ColumnNames` containing spaces or characters that cannot be used as MATLAB variable names, to valid MATLAB variable names. Choices are `true` or `false` (default).

Tip Use the `'CleanColNames'` property if you plan to use the `ColumnNames` field as variable names.

Examples

Note The gene expression file, `TumorAdjacent-probe-raw.txt` used in the following example is not provided with the Bioinformatics Toolbox software.

Read the contents of a tab-delimited file exported from the Illumina BeadStudio software into a MATLAB structure.

```
ilmnStruct = ilmnbsread('TumorAdjacent-probe-raw.txt')

ilmnStruct = 

             Header: [1x1 struct]
           TargetID: {22184x1 cell}
        ColumnNames: {1x37 cell}
               Data: [22184x37 double]
    TextColumnNames: {1x23 cell}
           TextData: {22184x23 cell}
```

Version History
Introduced in R2008a

See Also
affyread | agferead | celintensityread | galread | geoseriesread | geosoftread | gprread | ilmnbslookup | imageneread | magetfield | sptread

imageneread

Read microarray data from ImaGene Results file

Syntax

`imagenedata = imageneread(File)`

`imagenedata = imageneread(..., 'CleanColNames', CleanColNamesValue, ...)`

Arguments

`File`	ImaGene® Results formatted file. Enter a character vector or string specifying a file name, or a path and file name.
`CleanColNamesValue`	Controls the conversion of any `ColumnNames` containing spaces or characters that cannot be used as MATLAB variable names, to valid MATLAB variable names. Choices are `true` or `false` (default).

Description

`imagenedata` `= imageneread(`*`File`*`)` reads ImaGene results data from *`File`* and creates *`imagenedata`*, a MATLAB structure containing the following fields.

Field
`HeaderAA`
`Data`
`Blocks`
`Rows`
`Columns`
`Fields`
`IDs`
`ColumnNames`
`Indices`
`Shape`

`imagenedata` `= imageneread(..., '`*`PropertyName`*`', `*`PropertyValue`*`, ...)` calls `imageneread` with optional properties that use property name/property value pairs. You can specify one or more properties in any order. Each *`PropertyName`* must be enclosed in single quotation marks and is case insensitive. These property name/property value pairs are as follows:

`imagenedata` `= imageneread(..., 'CleanColNames', `*`CleanColNamesValue`*`, ...)` controls the conversion of any `ColumnNames` containing spaces or characters that cannot be used as MATLAB variable names, to valid MATLAB variable names. Choices are `true` or `false` (default).

The field `Indices` of the structure contains indices that you can use for plotting heat maps of the data with the function `image` or `imagesc`.

For more details on the ImaGene format and example data, see the ImaGene documentation.

Examples

In the following example, the file `cy3.txt` is not provided.

1. Read in a sample ImaGene Results file. Note that the example file, `cy3.txt`, is not provided with the Bioinformatics Toolbox software.

   ```
   cy3Data = imageneread('cy3.txt');
   ```
2. Plot the signal mean.

   ```
   maimage(cy3Data,'Signal Mean');
   ```
3. Read in a sample ImaGene Results file. Note that the example file, `cy5.txt`, is not provided with the Bioinformatics Toolbox software.

   ```
   cy5Data = imageneread('cy5.txt');
   ```
4. Create a loglog plot of the signal median from two ImaGene Results files.

   ```
   sigMedianCol = find(strcmp('Signal Median',cy3Data.ColumnNames));
   cy3Median = cy3Data.Data(:,sigMedianCol);
   cy5Median = cy5Data.Data(:,sigMedianCol);
   maloglog(cy3Median,cy5Median,'title','Signal Median');
   ```

Version History

Introduced before R2006a

See Also

`gprread` | `ilmnbsread` | `maboxplot` | `maimage` | `sptread`

int2aa

Convert amino acid sequence from integer to letter representation

Syntax

```
SeqChar = int2aa(SeqInt)
SeqChar = int2aa(SeqInt, 'Case', CaseValue)
```

Input Arguments

`SeqInt`	Row vector of integers specifying an amino acid sequence. For valid integers, see the table Mapping Amino Acid Integers to Letter Codes. Integers are arbitrarily assigned to IUB/IUPAC letters.
`CaseValue`	Character vector or string specifying the upper or lower case. Choices are `'upper'` (default) or `'lower'`.

Output Arguments

`SeqChar`	Amino acid sequence specified by a character vector of single-letter codes.

Description

`SeqChar` `= int2aa(`*`SeqInt`*`)` converts *`SeqInt`*, a row vector of integers specifying an amino acid sequence, to *`SeqChar`*, a character vector or string of single-letter codes specifying the same amino acid sequence. For valid integers, see the table Mapping Amino Acid Integers to Letter Codes.

`SeqChar` `= int2aa(`*`SeqInt`*`, 'Case', `*`CaseValue`*`)` specifies the upper or lower case. Choices are `'upper'` (default) or `'lower'`.

Mapping Amino Acid Integers to Letter Codes

Amino Acid	Integer	Code
Alanine	1	A
Arginine	2	R
Asparagine	3	N
Aspartic acid (Aspartate)	4	D
Cysteine	5	C
Glutamine	6	Q
Glutamic acid (Glutamate)	7	E
Glycine	8	G
Histidine	9	H
Isoleucine	10	I
Leucine	11	L
Lysine	12	K
Methionine	13	M
Phenylalanine	14	F
Proline	15	P
Serine	16	S
Threonine	17	T
Tryptophan	18	W
Tyrosine	19	Y
Valine	20	V
Asparagine or Aspartic acid (Aspartate)	21	B
Glutamine or Glutamic acid (Glutamate)	22	Z
Unknown amino acid (any amino acid)	23	X
Translation stop	24	*
Gap of indeterminate length	25	-
Unknown (any integer not in table)	0 or ≥ 26	?

Examples

Convert an amino acid sequence from integer to letter representation.

```
s = int2aa([13 1 17 11 1 21])
```

```
s =

MATLAB
```

Version History

Introduced before R2006a

See Also

`aa2int` | `aminolookup` | `int2nt` | `isotopicdist` | `nt2int`

int2nt

Convert nucleotide sequence from integer to letter representation

Syntax

```
SeqChar = int2nt(SeqInt)

SeqChar = int2nt(SeqInt, ...'Alphabet', AlphabetValue, ...)
SeqChar = int2nt(SeqInt, ...'Unknown', UnknownValue, ...)
SeqChar = int2nt(SeqInt, ...'Case', CaseValue, ...)
```

Input Arguments

SeqInt	Row vector of integers specifying a nucleotide sequence. For valid integers, see the table Mapping Nucleotide Integers to Letter Codes. Integers are arbitrarily assigned to IUB/IUPAC letters.
AlphabetValue	Character vector or string specifying a nucleotide alphabet. Choices are: • `'DNA'` (default) — Uses the symbols `A`, `C`, `G`, and `T`. • `'RNA'` — Uses the symbols `A`, `C`, `G`, and `U`.
UnknownValue	Character to represent unknown nucleotides, that is `0` or integers ≥ `17`. Choices are any character other than the nucleotide characters `A`, `C`, `G`, `T`, and `U` and the ambiguous nucleotide characters `N`, `R`, `Y`, `K`, `M`, `S`, `W`, `B`, `D`, `H`, and `V`. Default is `*`.
CaseValue	Character vector or string specifying the upper or lower case. Choices are `'upper'` (default) or `'lower'`.

Output Arguments

SeqChar	Nucleotide sequence specified by a character vector of codes.

Description

`SeqChar = int2nt(SeqInt)` converts *`SeqInt`*, a row vector of integers specifying a nucleotide sequence, to *`SeqChar`*, a character vector of codes specifying the same nucleotide sequence. For valid codes, see the table Mapping Nucleotide Integers to Letter Codes.

Mapping Nucleotide Integers to Letter Codes

Nucleotide	Integer	Code
Adenosine	1	A
Cytidine	2	C
Guanine	3	G
Thymidine	4	T
Uridine (if `'Alphabet'` set to `'RNA'`)	4	U
Purine (A or G)	5	R
Pyrimidine (T or C)	6	Y
Keto (G or T)	7	K
Amino (A or C)	8	M
Strong interaction (3 H bonds) (G or C)	9	S
Weak interaction (2 H bonds) (A or T)	10	W
Not A (C or G or T)	11	B
Not C (A or G or T)	12	D
Not G (A or C or T)	13	H
Not T or U (A or C or G)	14	V
Any nucleotide (A or C or G or T or U)	15	N
Gap of indeterminate length	16	-
Unknown (any integer not in table)	0 or ≥ 17	* (default)

SeqChar = int2nt(*SeqInt*, ...*PropertyName*', *PropertyValue*, ...) calls `int2nt` with optional properties that use property name/property value pairs. You can specify one or more properties in any order. Each *PropertyName* must be enclosed in single quotation marks and is case insensitive. These property name/property value pairs are as follows:

SeqChar = int2nt(*SeqInt*, ...'Alphabet', *AlphabetValue*, ...) specifies a nucleotide alphabet. *AlphabetValue* can be `'DNA'`, which uses the symbols A, C, G, and T, or `'RNA'`, which uses the symbols A, C, G, and U. Default is `'DNA'`.

SeqChar = int2nt(*SeqInt*, ...'Unknown', *UnknownValue*, ...) specifies the character to represent unknown nucleotides, that is 0 or integers ≥ 17. *UnknownValue* can be any character other than the nucleotide characters A, C, G, T, and U and the ambiguous nucleotide characters N, R, Y, K, M, S, W, B, D, H, and V. Default is *.

SeqChar = int2nt(*SeqInt*, ...'Case', *CaseValue*, ...) specifies the upper or lower case. *CaseValue* can be `'upper'` (default) or `'lower'`.

Examples

- Convert a nucleotide sequence from integer to letter representation.

  ```
  s = int2nt([1 2 4 3 2 4 1 3 2])
  ```

```
s =
ACTGCTAGC
```

- Convert a nucleotide sequence from integer to letter representation and define # as the symbol for unknown numbers 17 and greater.

```
si = [1 2 4 20 2 4 40 3 2];
s = int2nt(si, 'unknown', '#')

s =
ACT#CT#GC
```

Version History

Introduced before R2006a

See Also

aa2int | baselookup | int2aa | nt2int

isdag (biograph)

(Removed) Test for cycles in biograph object

Note The function has been removed. Use the `isdag` function of `graph` or `digraph` instead.

Syntax

`isdag(`*`BGObj`*`)`

Arguments

`BGObj`	Biograph object created by `biograph` (object constructor).

Description

Tip For introductory information on graph theory functions, see “Graph Theory Functions”.

`isdag(`*`BGObj`*`)` returns logical 1 (`true`) if an N-by-N adjacency matrix extracted from a biograph object, *`BGObj`*, is a directed acyclic graph (DAG) and logical 0 (`false`) otherwise. In the N-by-N sparse matrix, all nonzero entries indicate the presence of an edge.

Version History

Introduced in R2006b

R2022b: Removed
Warns starting in R2022b

The function has been removed. Use the `isdag` function of `graph` or `digraph` instead.

R2022a: Warns
Warns starting in R2022a

The function issues a warning that it will be removed in a future release.

R2021b: To be removed
Not recommended starting in R2021b

The function runs without warning, but it will be removed in a future release.

References

[1] Siek, J.G., Lee, L-Q, and Lumsdaine, A. (2002). The Boost Graph Library User Guide and Reference Manual, (Upper Saddle River, NJ:Pearson Education).

See Also

`isdag` | `graph` | `digraph`

isempty

Class: `bioma.data.ExptData`
Package: `bioma.data`

Determine whether ExptData object is empty

Syntax

`TF = isempty(EDObj)`

Description

`TF = isempty(EDObj)` returns logical `1` (`true`) if *`EDObj`* is an empty ExptData object. Otherwise, it returns logical `0` (`false`). An empty ExptData object contains no data elements.

Input Arguments

`EDObj`

Object of the `bioma.data.ExptData` class.

Default:

Examples

Construct an ExptData object, and then check to see if it is empty:

```
% Import bioma.data package to make constructor functions
% available
import bioma.data.*
% Create DataMatrix object from .txt file containing
% expression values from microarray experiment
dmObj = DataMatrix('File', 'mouseExprsData.txt');
% Construct ExptData object
EDObj = ExptData(dmObj);
% Determine if ExptData object is empty
isempty(EDObj)
```

See Also

`bioma.data.ExptData`

Topics

“Representing Expression Data Values in ExptData Objects”

isempty

Class: `bioma.data.MetaData`
Package: `bioma.data`

Determine whether MetaData object is empty

Syntax

`TF = isempty(MDObj)`

Description

`TF = isempty(MDObj)` returns logical `1` (`true`) if `MDObj` is an empty MetaData object. Otherwise, it returns logical `0` (`false`). An empty MetaData object contains no variable names, values, or descriptions.

Input Arguments

`MDObj`

Object of the `bioma.data.MetaData` class.

Default:

Examples

Construct a MetaData object, and then check to see if it is empty:

```
% Import bioma.data package to make constructor function
% available
import bioma.data.*
% Construct MetaData object from .txt file
MDObj2 = MetaData('File', 'mouseSampleData.txt', 'VarDescChar', '#');
% Determine if MetaData object is empty
isempty(MDObj2)
```

See Also
`bioma.data.MetaData`

Topics
"Representing Sample and Feature Metadata in MetaData Objects"

isempty

Class: bioma.data.MIAME
Package: bioma.data

Determine whether MIAME object is empty

Syntax

TF = isempty(*MIAMEObj*)

Description

TF = isempty(*MIAMEObj*) returns logical 1 (true) if *MIAMEObj* is an empty MIAME object. Otherwise, it returns logical 0 (false). All properties are empty in an empty MIAME object.

Input Arguments

MIAMEObj

Object of the bioma.data.MIAME class.

Default:

Examples

Construct a MIAME object, and then check to see if it is empty:

```
% Create a MATLAB structure containing GEO Series data
geoStruct = getgeodata('GSE4616');
% Import bioma.data package to make constructor function
% available
import bioma.data.*
% Construct MIAME object
MIAMEObj = MIAME(geoStruct);
% Determine if MIAME object is empty
isempty(MIAMEObj)
```

See Also
bioma.data.MIAME

Topics
“Representing Experiment Information in a MIAME Object”

isequal (DataMatrix)

Test DataMatrix objects for equality

Syntax

```
TF = isequal(DMObj1, DMObj2)
TF = isequal(DMObj1, DMObj2, DMObj3, ...)
```

Input Arguments

DMObj1, *DMObj2*, *DMObj3*	DataMatrix objects, such as created by `DataMatrix` (object constructor).

Output Arguments

TF	Logical value indicating if inputs are numerically equal (have the same contents), have the same size (same `NRows` and `NCols` properties), and have the same `RowNames` and `ColNames` properties. `NaN`s are not considered equal to each other.

Description

`TF = isequal(DMObj1, DMObj2)` returns logical `1` (`true`) if the input DataMatrix objects, *DMObj1* and *DMObj2*, meet the following:

- Are numerically equal (have the same contents)
- Have the same size (same `NRows` and `NCols` properties)
- Have the same `RowNames` and `ColNames` properties

Otherwise, it returns logical `0` (`false`). *DMObj1* and *DMObj2* do not have to have the same `Name` property. `NaN`s are not considered equal to each other.

`TF = isequal(DMObj1, DMObj2, DMObj3, ...)` returns logical `1` (`true`) if all input DataMatrix objects, *DMObj1*, *DMObj2*, *DMObj3*, etc. meet the following:

- Are numerically equal (have the same contents)
- Have the same size (same `NRows` and `NCols` properties)
- Have the same `RowNames` and `ColNames` properties

Otherwise, it returns logical `0` (`false`). The input DataMatrix objects do not have to have the same `Name` property. `NaN`s are not considered equal to each other.

Version History

Introduced in R2008b

See Also

`DataMatrix` | `isequaln`

Topics

DataMatrix object on page 1-734

isequaln (DataMatrix)

Test DataMatrix objects for equality, treating NaNs as equal

Syntax

```
TF = isequaln(DMObj1, DMObj2)
TF = isequaln(DMObj1, DMObj2, DMObj3, ...)
```

Input Arguments

DMObj1, *DMObj2*, *DMObj3*	DataMatrix objects, such as created by `DataMatrix` (object constructor).

Output Arguments

TF	Logical value indicating if inputs are numerically equal (have the same contents), have the same size (same `NRows` and `NCols` properties), and have the same `RowNames` and `ColNames` properties. `NaN`s are considered equal to each other.

Description

`TF = isequaln(DMObj1, DMObj2)` returns logical `1` (`true`) if the input DataMatrix objects, *`DMObj1`* and *`DMObj2`*, meet the following:

- Are numerically equal (have the same contents)
- Have the same size (same `NRows` and `NCols` properties)
- Have the same `RowNames` and `ColNames` properties

Otherwise, it returns logical `0` (`false`). *`DMObj1`* and *`DMObj2`* do not need to have the same `Name` property. `NaN`s are considered equal to each other.

`TF = isequaln(DMObj1, DMObj2, DMObj3, ...)` returns logical `1` (`true`) if all input DataMatrix objects, *`DMObj1`*, *`DMObj2`*, *`DMObj3`*, etc. meet the following:

- Are numerically equal (have the same contents)
- Have the same size (same `NRows` and `NCols` properties)
- Have the same `RowNames` and `ColNames` properties

Otherwise, it returns logical `0` (`false`). The input DataMatrix objects do not need to have the same `Name` property. `NaN`s are considered equal to each other.

Version History

Introduced in R2012b

See Also
`DataMatrix` | `isequal`

Topics
DataMatrix object on page 1-734

isequalwithequalnans (DataMatrix)

Test DataMatrix objects for equality, treating NaNs as equal

Syntax

```
TF = isequalwithequalnans(DMObj1, DMObj2)
TF = isequalwithequalnans(DMObj1, DMObj2, DMObj3, ...)
```

Input Arguments

DMObj1, *DMObj2*, *DMObj3*	DataMatrix objects, such as created by `DataMatrix` (object constructor).

Output Arguments

TF	Logical value indicating if inputs are numerically equal (have the same contents), have the same size (same `NRows` and `NCols` properties), and have the same `RowNames` and `ColNames` properties. `NaN`s are considered equal to each other.

Description

`TF = isequalwithequalnans(`*`DMObj1`*`, `*`DMObj2`*`)` returns logical `1` (`true`) if the input DataMatrix objects, *`DMObj1`* and *`DMObj2`*, meet the following:

- Are numerically equal (have the same contents)
- Have the same size (same `NRows` and `NCols` properties)
- Have the same `RowNames` and `ColNames` properties

Otherwise, it returns logical `0` (`false`). *`DMObj1`* and *`DMObj2`* do not have to have the same `Name` property. `NaN`s are considered equal to each other.

`TF = isequalwithequalnans(`*`DMObj1`*`, `*`DMObj2`*`, `*`DMObj3`*`, ...)` returns logical `1` (`true`) if all input DataMatrix objects, *`DMObj1`*, *`DMObj2`*, *`DMObj3`*, etc. meet the following:

- Are numerically equal (have the same contents)
- Have the same size (same `NRows` and `NCols` properties)
- Have the same `RowNames` and `ColNames` properties

Otherwise, it returns logical `0` (`false`). The input DataMatrix objects do not have to have the same `Name` property. `NaN`s are considered equal to each other.

Version History

Introduced in R2008b

See Also
`DataMatrix` | `isequal`

Topics
DataMatrix object on page 1-734

isoelectric

Estimate isoelectric point for amino acid sequence

Syntax

```
pI = isoelectric(SeqAA)
[pI Charge] = isoelectric(SeqAA)

isoelectric(..., 'PropertyName', PropertyValue,...)
isoelectric(..., 'PKVals', PKValsValue)
isoelectric(..., 'Charge', ChargeValue)
isoelectric(..., 'Chart', ChartValue)
```

Arguments

SeqAA	Amino acid sequence. Enter a character vector, string, or a vector of integers from the table Mapping Amino Acid Letter Codes to Integers. Examples: `'ARN'` or `[1 2 3]`.
PKValsValue	Character vector or string specifying a file name or path and file name of a PK file containing a table of pK values for amino acids, which .`isoelectric` uses to estimate the isoelectric point (*pI*) of an amino acid sequence. For an example of a PK file format, type `edit Emboss.pK` in the MATLAB command line.
ChargeValue	Property to select a specific pH for estimating charge. Enter a number between `0` and `14`. Default is `7.2`.
ChartValue	Controls the plotting a graph of charge versus pH. Enter `true` or `false`.

Description

`pI = isoelectric(SeqAA)` returns the estimated isoelectric point (*pI*) for an amino acid sequence using the following pK values:

```
N_term     8.6
K         10.8
R         12.5
H          6.5
D          3.9
E          4.1
C          8.5
Y         10.1
C_term     3.6
```

The isoelectric point is the pH at which the protein has a net charge of zero.

`[pI Charge] = isoelectric(SeqAA)` returns the estimated isoelectric point (*pI*) for an amino acid sequence and the estimated charge for a given pH (default is typical intracellular pH `7.2`).

The estimates are skewed by the underlying assumptions that all amino acids are fully exposed to the solvent, that neighboring peptides have no influence on the pK of any given amino acid, and that the constitutive amino acids, as well as the N- and C-termini, are unmodified. Cysteine residues participating in disulfide bridges also affect the true pI and are not considered here. By default, `isoelectric` uses the EMBOSS amino acid pK table, or you can substitute other values using the property `PKVals`.

- If the sequence contains ambiguous amino acid characters (b z * –), `isoelectric` ignores the characters and displays a warning message.

  ```
  Warning: Symbols other than the standard 20 amino acids
  appear in the sequence.
  ```
- If the sequence contains undefined amino acid characters (`i j o`) , `isoelectric` ignores the characters and displays a warning message.

  ```
  Warning: Sequence contains unknown characters. These will
  be ignored.
  ```

`isoelectric(..., 'PropertyName', PropertyValue,...)` defines optional properties using property name/value pairs.

`isoelectric(..., 'PKVals', PKValsValue)` uses pK values stored in a *`PKValValues`*, a PK file, to estimate the isoelectric point (*`pI`*) of an amino acid sequence. For an example of a PK file format, type `edit Emboss.pK` in the MATLAB command line.

`isoelectric(..., 'Charge', ChargeValue)` returns the estimated charge of a sequence for a given pH (*`ChargeValue`*).

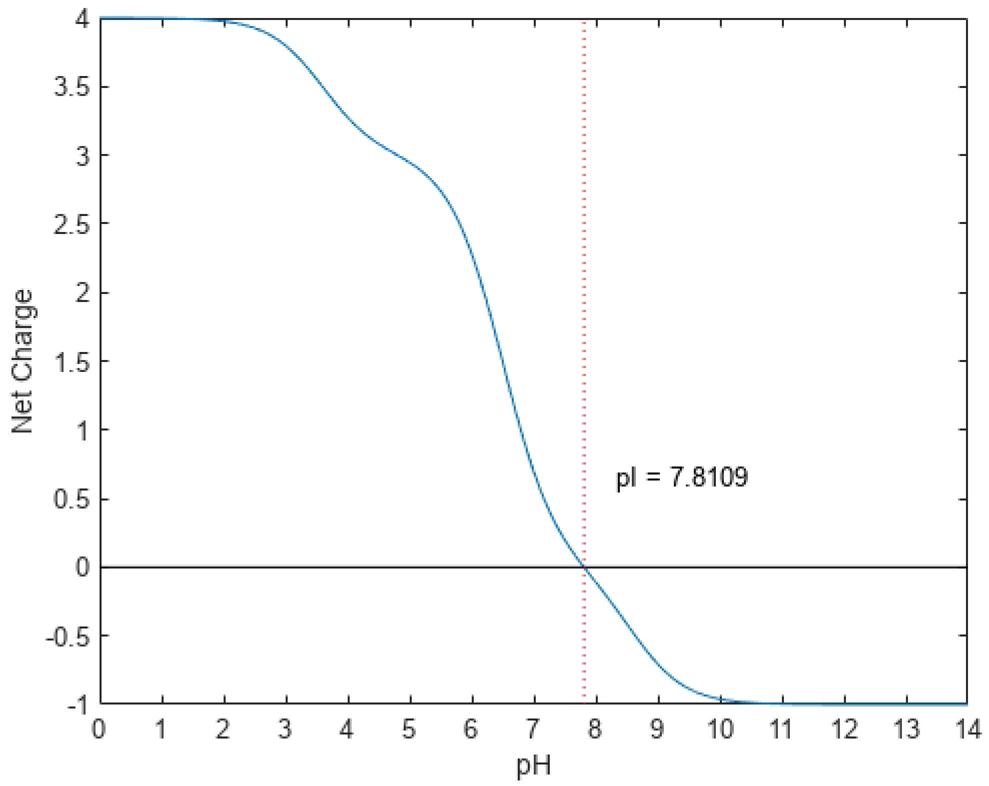
`isoelectric(..., 'Chart', ChartValue)` when *`ChartValue`* is `true`, returns a graph plotting the charge of the protein versus the pH of the solvent.

Examples

Estimate Isoelectric Point for Amino Acid Sequence

Calculate the isoelectric point using the pK values stored in a PK file and plot the charge against the pH for a short sequence.

```
isoelectric('PQGGGGWGQPHGGGWGQPHGGGGWGQGGSHSQG','PKVal','Emboss.pK','CHART',true)
```



```
ans = 7.8109
```

Also estimate its charge at a specific pH.

```
[pI,Charge] = isoelectric('PQGGGGWGQPHGGGWGQPHGGGGWGQGGSHSQG','Charge',7.3)
```

```
pI = 7.8109
```

```
Charge = 0.3629
```

Version History

Introduced before R2006a

See Also

aacount | molweight

isomorphism (biograph)

(Removed) Find isomorphism between two biograph objects

Note The function has been removed. Use the `isomorphism` function of `graph` or `digraph` instead.

Syntax

```
[Isomorphic, Map] = isomorphism(BGObj1, BGObj2)
[Isomorphic, Map] = isomorphism(BGObj1, BGObj2,'Directed', DirectedValue)
```

Arguments

BGObj1	Biograph object created by `biograph` (object constructor).
BGObj2	Biograph object created by `biograph` (object constructor).
DirectedValue	Property that indicates whether the graphs are directed or undirected. Enter `false` when both *BGObj1* and *BGObj2* produce undirected graphs. In this case, the upper triangles of the sparse matrices extracted from *BGObj1* and *BGObj2* are ignored. Default is `true`, meaning that both graphs are directed.

Description

Tip For introductory information on graph theory functions, see “Graph Theory Functions”.

`[`*Isomorphic*`,` *Map*`] = isomorphism(`*BGObj1*`,` *BGObj2*`)` returns logical 1 (`true`) in *Isomorphic* if two N-by-N adjacency matrices extracted from biograph objects *BGObj1* and *BGObj2* are isomorphic graphs, and logical 0 (`false`) otherwise. A graph isomorphism is a 1-to-1 mapping of the nodes in the graph from *BGObj1* and the nodes in the graph from *BGObj2* such that adjacencies are preserved. Return value *Isomorphic* is Boolean. When *Isomorphic* is `true`, *Map* is a row vector containing the node indices that map from *BGObj2* to *BGObj1*. When *Isomorphic* is `false`, the worst-case time complexity is `O(N!)`, where `N` is the number of nodes.

`[`*Isomorphic*`,` *Map*`] = isomorphism(`*BGObj1*`,` *BGObj2*`,'Directed',` *DirectedValue*`)` indicates whether the graphs are directed or undirected. Set *DirectedValue* to `false` when both *BGObj1* and *BGObj2* produce undirected graphs. In this case, the upper triangles of the sparse matrices extracted from *BGObj1* and *BGObj2* are ignored. The default is `true`, meaning that both graphs are directed.

Version History

Introduced in R2006b

R2022b: Removed

Errors starting in R2022b

The function has been removed. Use the `isomorphism` function of `graph` or `digraph` instead.

R2022a: Warns
Warns starting in R2022a

The function issues a warning that it will be removed in a future release.

R2021b: To be removed
Not recommended starting in R2021b

The function runs without warning, but it will be removed in a future release.

References

[1] Fortin, S. (1996). The Graph Isomorphism Problem. Technical Report, 96-20, Dept. of Computer Science, University of Alberta, Edomonton, Alberta, Canada.

[2] McKay, B.D. (1981). Practical Graph Isomorphism. Congressus Numerantium *30*, 45-87.

[3] Siek, J.G., Lee, L-Q, and Lumsdaine, A. (2002). The Boost Graph Library User Guide and Reference Manual, (Upper Saddle River, NJ:Pearson Education).

See Also

`isomorphism` | `graph` | `digraph`

isotopicdist

Calculate high-resolution isotope mass distribution and density function

Syntax

```
[MD, Info, DF] = isotopicdist(SeqAA)
[MD, Info, DF] = isotopicdist(Compound)
[MD, Info, DF] = isotopicdist(Formula)

isotopicdist(..., 'NTerminal', NTerminalValue, ...)
isotopicdist(..., 'CTerminal', CTerminalValue, ...)
isotopicdist(..., 'Resolution', ResolutionValue, ...)
isotopicdist(..., 'FFTResolution', FFTResolutionValue, ...)
isotopicdist(..., 'FFTRange', FFTRangeValue, ...)
isotopicdist(..., 'FFTLocation', FFTLocationValue, ...)
isotopicdist(..., 'NoiseThreshold', NoiseThresholdValue, ...)
isotopicdist(..., 'ShowPlot', ShowPlotValue, ...)
```

Description

`[MD, Info, DF] = isotopicdist(SeqAA)` analyzes a peptide sequence and returns a matrix containing the expected mass distribution; a structure containing the monoisotopic mass, average mass, most abundant mass, nominal mass, and empirical formula; and a matrix containing the expected density function.

`[MD, Info, DF] = isotopicdist(Compound)` analyzes a compound specified by a numeric vector or matrix.

`[MD, Info, DF] = isotopicdist(Formula)` analyzes a compound specified by an empirical chemical formula represented by the structure *`Formula`*. The field names in *`Formula`* must be valid element symbols and are case sensitive. The respective values in *`Formula`* are the number of atoms for each element. *`Formula`* can also be an array of structures that specifies multiple formulas. The field names can be in any order within a structure. However, if there are multiple structures, the order must be the same in each.

`isotopicdist(..., 'PropertyName', PropertyValue, ...)` calls `isotopicdist` with optional properties that use property name/property value pairs. You can specify one or more properties in any order. Enclose each *`PropertyName`* in single quotation marks. Each *`PropertyName`* is case insensitive. These property name/property value pairs are as follows:

`isotopicdist(..., 'NTerminal', NTerminalValue, ...)` modifies the N-terminal of the peptide.

`isotopicdist(..., 'CTerminal', CTerminalValue, ...)` modifies the C-terminal of the peptide.

`isotopicdist(..., 'Resolution', ResolutionValue, ...)` specifies the approximate resolution of the instrument, given as the Gaussian width (in daltons) at full width at half height (FWHH).

`isotopicdist(..., 'FFTResolution', FFTResolutionValue, ...)` specifies the number of data points per dalton, to compute the fast Fourier transform (FFT) algorithm.

`isotopicdist(..., 'FFTRange', FFTRangeValue, ...)` specifies the absolute range (window size) in daltons for the FFT algorithm and output density function.

`isotopicdist(..., 'FFTLocation', FFTLocationValue, ...)` specifies the location of the FFT range (window) defined by *`FFTRangeValue`*. It specifies this location by setting the location of the lower limit of the range, relative to the location of the monoisotopic peak, which is computed by `isotopicdist`.

`isotopicdist(..., 'NoiseThreshold', NoiseThresholdValue, ...)` removes points in the mass distribution that are smaller than `1/`*`NoiseThresholdValue`* times the most abundant mass.

`isotopicdist(..., 'ShowPlot', ShowPlotValue, ...)` controls the display of a plot of the mass distribution.

Input Arguments

`SeqAA`

Peptide sequence specified by either a:

- Character vector or string of single-letter codes
- Cell array of character vectors or string vector that specifies multiple peptide sequences

Tip You can use the `getgenpept` and `genpeptread` functions to retrieve peptide sequences from the GenPept database or a GenPept-formatted file. You can then use the `cleave` function to perform an insilico digestion on a peptide sequence. The `cleave` function creates a cell array of character vectors representing peptide fragments, which you can submit to the `isotopicdist` function.

Default:

`Compound`

Compound specified by either a:

- Numeric vector of form `[C H N O S]`, where `C`, `H`, `N`, `O`, and `S` are nonnegative numbers that represent the number of atoms of carbon, hydrogen, nitrogen, oxygen, and sulfur respectively in a compound.
- M-by-5 numeric matrix that specifies multiple compounds, with each row corresponding to a compound and each column corresponding to an atom.

Default:

`Formula`

Chemical formula specified by either a:

- Structure whose field names are valid element symbols and case sensitive. Their respective values are the number of atoms for each element.

- Array of structures that specifies multiple formulas.

Note If *`Formula`* is a single structure, the order of the fields does not matter. If *`Formula`* is an array of structures, then the order of the fields must be the same in each structure.

Default:

`NTerminalValue`

Modification for the N-terminal of the peptide, specified by either:

- One of `'none'`, `'amine'` (default), `'formyl'`, or `'acetyl'`
- Custom modification specified by an empirical formula, represented by a structure. The structure must have field names that are valid element symbols and case sensitive. Their respective values are the number of atoms for each element.

`CTerminalValue`

Modification for the C-terminal of the peptide, specified by either:

- One of `'none'`, `'freeacid'` (default), or `'amide'`
- Custom modification specified by an empirical formula, represented by a structure. The structure must have field names that are valid element symbols and case sensitive. Their respective values are the number of atoms for each element.

`ResolutionValue`

Value in daltons specifying the approximate resolution of the instrument, given as the Gaussian width at full width half height (FWHH).

Default: `1/8` Da

`FFTResolutionValue`

Value specifying the number of data points per dalton, used to compute the FFT algorithm.

Default: `1000`

`FFTRangeValue`

Value specifying the absolute range (window size) in daltons for the FFT algorithm and output density function. By default, this value is automatically estimated based on the weight of the molecule. The actual FFT range used internally by `isotopicdist` is further increased such that *`FFTRangeValue * FFTResolutionValue`* is a power of two.

Tip Increase the *`FFTRangeValue`* if the signal represented by the *`DF`* output appears to be truncated.

Tip Ultrahigh resolution allows you to resolve micropeaks that have the same nominal mass, but slightly different exact masses. To achieve ultrahigh resolution, increase *`FFTResolutionValue`* and reduce *`ResolutionValue`*, but ensure that *`FFTRangeValue * FFTResolutionValue`* is within the available memory.

Default:

`FFTLocationValue`

Fraction that specifies the location of the FFT range (window) defined by *`FFTRangeValue`*. It specifies this location by setting the location of the lower limit of the FFT range, relative to the location of the monoisotopic peak, which is computed by `isotopicdist`. The location of the lower limit of the FFT range is set to the mass of the monoistopic peak `- (`*`FFTLocationValue * FFTRangeValue`*`)`.

Tip You may need to shift the FFT range to the left in rare cases where a compound contains an element, such as Iron or Argon, whose most abundant isotope is not the lightest one.

Default: `1/16`

`NoiseThresholdValue`

Value that removes points in the mass distribution that are smaller than `1/`*`NoiseThresholdValue`* times the most abundant mass.

Default: `1e6`

`ShowPlotValue`

Controls the display of a plot of the isotopic mass distribution. Choices are `true`, `false`, or *`I`*, which is an integer specifying a compound. If set to `true`, the first compound is plotted. Default is:

- `false` — When you specify return values.
- `true` — When you do not specify return values.

Default:

Output Arguments

`MD`

Mass distribution represented by a two-column matrix in which each row corresponds to an isotope. The first column lists the isotopic mass, and the second column lists the probability for that mass.

`Info`

Structure containing mass information for the peptide sequence or compound in the following fields:

- `NominalMass`
- `MonoisotopicMass`
- `ObservedAverageMass` — Estimated from the *`DF`* signal output, using instrument resolution specified by the `'Resolution'` property.
- `CalculatedAverageMass` — Calculated directly from the input formula, assuming perfect instrument resolution.
- `MostAbundantMass`

- `Formula` — Structure containing the number of atoms of each element.

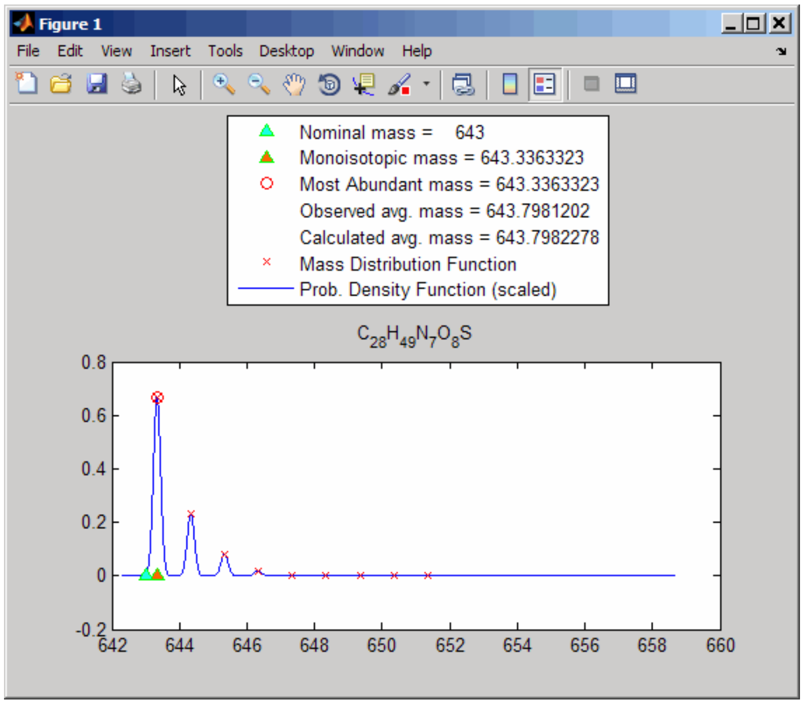
DF

Density function represented by a two-column matrix in which each row corresponds to an m/z value. The first column lists the mass, and the second column lists the relative intensity of the signal at that mass.

Examples

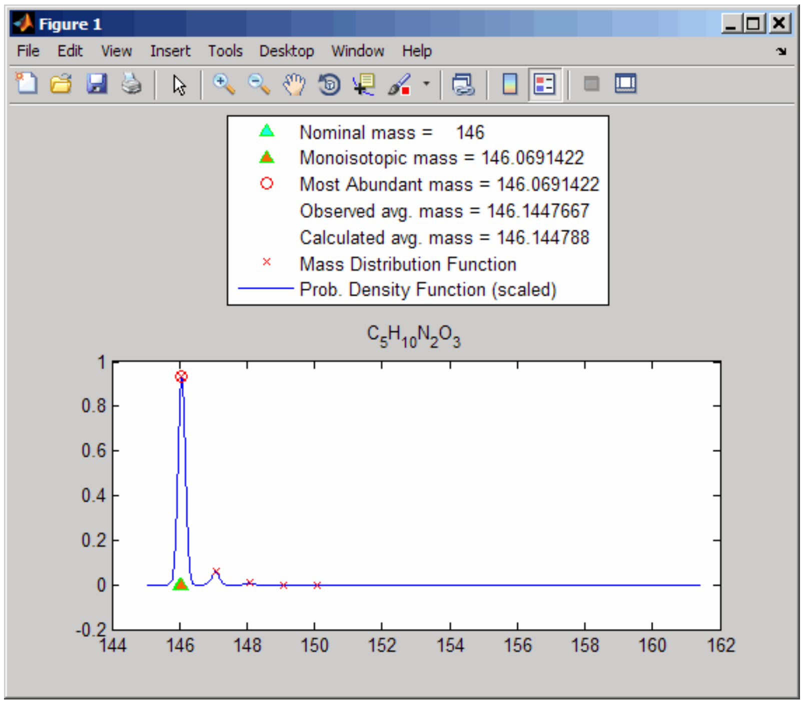
Calculate and display the isotopic mass distribution of the peptide sequence `MATLAP` with an Acetyl N-terminal and an Amide C-terminal:

```
MD = isotopicdist('MATLAP','nterm','Acetyl','cterm','Amide', ...
                  'showplot',true)

MD =

  643.3363    0.6676
  644.3388    0.2306
  645.3378    0.0797
  646.3386    0.0181
  647.3396    0.0033
  648.3409    0.0005
  649.3423    0.0001
  650.3439    0.0000
  651.3455    0.0000
```



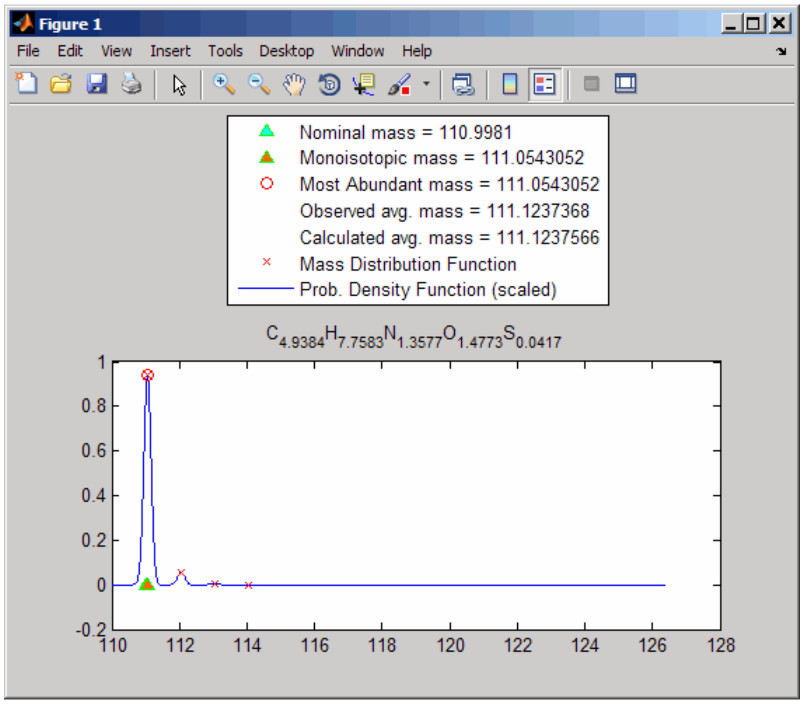
Calculate and display the isotopic mass distribution of Glutamine ($C_5H_{10}N_2O_3$):

```
MD = isotopicdist([5 10 2 3 0],'showplot',true)
MD =

  146.0691    0.9328
  147.0715    0.0595
  148.0733    0.0074
  149.0755    0.0004
  150.0774    0.0000
```



Display the isotopic mass distribution of the "averagine" model, whose molecular formula represents the statistical occurrences of amino acids from all known proteins:

```
isotopicdist([4.9384 7.7583 1.3577 1.4773 0.0417])
```



More About

Average Mass

Sum of the average atomic masses of the constituent elements in a molecule.

Monoisotopic Mass

Sum of the masses of the atoms in a molecule using the unbound, ground-state, rest mass of the principle (most abundant) isotope for each element instead of the isotopic average mass.

Most Abundant Mass

Mass of the molecule with the most-highly represented isotope distribution, based on the natural abundance of the isotopes.

Nominal Mass

Sum of the integer masses (ignoring the mass defect) of the most abundant isotope of each element in a molecule.

Version History

Introduced in R2009b

References

[1] Rockwood, A. L., Van Orden, S. L., and Smith, R. D. (1995). Rapid Calculation of Isotope Distributions. Anal. Chem. *67:15*, 2699–2704.

[2] Rockwood, A. L., Van Orden, S. L., and Smith, R. D. (1996). Ultrahigh Resolution Isotope Distribution Calculations. Rapid Commun. Mass Spectrum *10*, 54–59.

[3] Senko, M.W., Beu, S. C., and McLafferty, F. W. (1995). Automated assignment of charge states from resolved isotopic peaks for multiply charged ions. J. Am. Soc. Mass Spectrom. *6*, 52–56.

[4] Senko, M.W., Beu, S. C., and McLafferty, F. W. (1995). Determination of monoisotopic masses and ion populations for large biomolecules from resolved isotopic distributions. J. Am. Soc. Mass Spectrom. *6*, 229–233.

See Also

`cleave` | `getgenpept` | `genpeptread` | `int2aa` | `nt2aa` | `aminolookup` | `cleavelookup` | `molweight`

isspantree (biograph)

(Removed) Determine if tree created from biograph object is spanning tree

Note The function has been removed. A graph is a spanning tree if and only if all nodes are reachable from an arbitrary start node, and `E == N-1`, where *E* is the number of edges and *N* is the number of nodes. You can use either `bfsearch` or `dfsearch` to check if such conditions are true for a given graph.

Syntax

```
TF = isspantree(BGObj)
```

Arguments

BGObj	Biograph object created by `biograph` (object constructor).

Description

Tip For introductory information on graph theory functions, see “Graph Theory Functions”.

`TF = isspantree(BGObj)` returns logical 1 (`true`) if the N-by-N adjacency matrix extracted from a biograph object, *BGObj*, is a spanning tree, and logical 0 (`false`) otherwise. A spanning tree must touch all the nodes and must be acyclic. The lower triangle of the N-by-N adjacency matrix represents an undirected graph, and all nonzero entries indicate the presence of an edge.

Note The function ignores the direction of the edges in the Biograph object.

Version History

Introduced in R2006b

R2022b: Removed
Errors starting in R2022b

The function has been removed. A graph is a spanning tree if and only if all nodes are reachable from an arbitrary start node, and `E == N-1`, where *E* is the number of edges and *N* is the number of nodes. You can use either `bfsearch` or `dfsearch` to check if such conditions are true for a given graph.

R2022a: Warns
Warns starting in R2022a

The function issues a warning that it will be removed in a future release.

R2021b: To be removed
Not recommended starting in R2021b

The function runs without warning, but it will be removed in a future release.

References

[1] Siek, J.G., Lee, L-Q, and Lumsdaine, A. (2002). The Boost Graph Library User Guide and Reference Manual, (Upper Saddle River, NJ:Pearson Education).

See Also

`bfsearch` | `dfsearch` | `graph` | `digraph`

jcampread

Read JCAMP-DX-formatted files

Syntax

`JCAMPStruct = jcampread(File)`

Input Arguments

File	Either of the following: • Character vector or string specifying a file name, a path and file name, or a URL pointing to a file. The referenced file is a JCAMP-DX-formatted file (ASCII text file). If you specify only a file name, that file must be on the MATLAB search path or in the current folder. • MATLAB character array that contains the text of a JCAMP-DX-formatted file.

Output Arguments

JCAMPStruct	MATLAB structure containing information from a JCAMP-DX-formatted file.

Description

JCAMP-DX is a file format for infrared, NMR, and mass spectrometry data from the Joint Committee on Atomic and Molecular Physical Data (JCAMP). `jcampread` supports reading data from files saved with Versions 4.24, 5, or 6 of the JCAMP-DX format. For more details, see:

`http://www.jcamp-dx.org/`

`JCAMPStruct` `= jcampread(`*`File`*`)` reads data from *`File`*, a JCAMP-DX-formatted file, and creates *`JCAMPStruct`*, a MATLAB structure containing the following fields.

Field
`Title`
`DataType`
`DataClass` (version 5.00 and above)
`Origin`
`Owner`
`Blocks`
`Notes`

The `Blocks` field of the structure is an array of structures corresponding to each set of data in the file. These structures have the following fields.

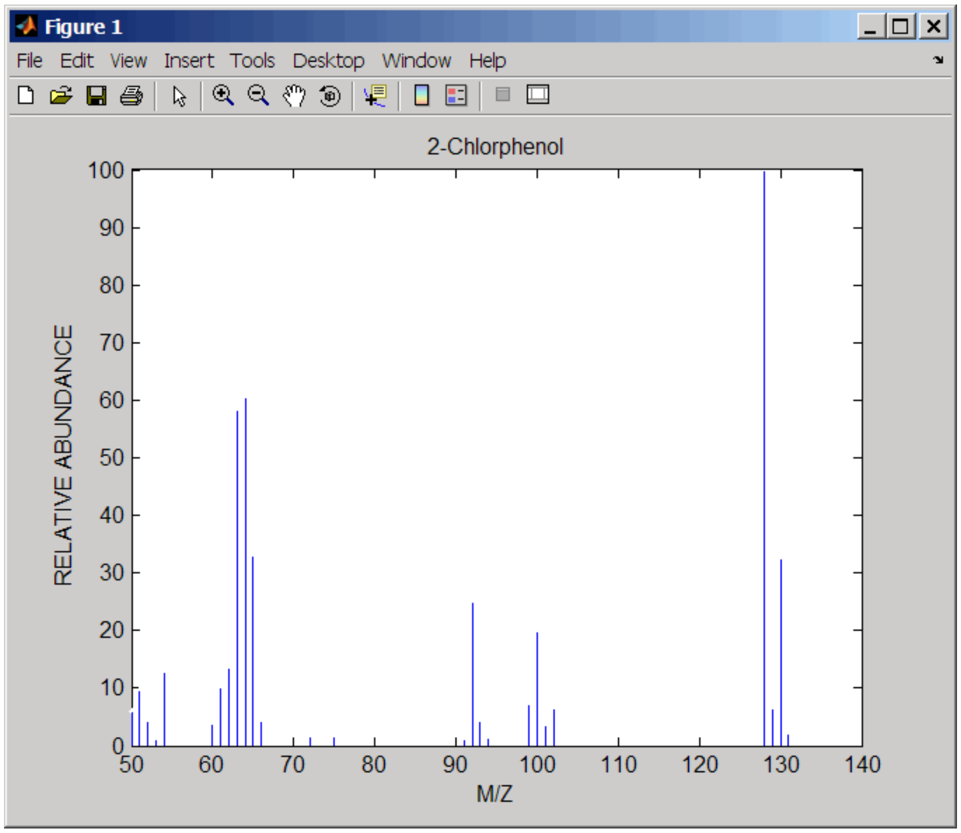
Field
`XData`
`YData`
`ZData` (if multiple blocks)
`XUnits`
`YUnits`
`ZUnits` (if multiple blocks)
`Notes`

Examples

1. Open a Web browser to http://www.jcamp-dx.org/testdata.html.
2. Download the `testdata.zip` file to your MATLAB Current Folder.
3. Extract `isas_ms1.dx`, a JCAMP-DX-formatted file, from the `testdata.zip` file to your MATLAB Current Folder.
4. Read the data from the JCAMP-DX-formatted file, `isas_ms1.dx`, into the MATLAB software

   ```
   jcampStruct = jcampread('isas_ms1.dx')

   jcampStruct = 

           Title: '2-Chlorphenol'
        DataType: 'MASS SPECTRUM'
       DataClass: 'PEAKTABLE'
          Origin: 'H. Mayer, ISAS Dortmund'
           Owner: 'COPYRIGHT (C) 1993 by ISAS Dortmund, FRG'
          Blocks: [1x1 struct]
           Notes: {8x2 cell}
   ```
5. Plot the mass spectrum.

   ```
   data = jcampStruct.Blocks(1);
   stem(data.XData,data.YData, '.', 'MarkerEdgeColor','w');
   title(jcampStruct.Title);
   xlabel(data.XUnits);
   ylabel(data.YUnits);
   ```



Version History

Introduced before R2006a

See Also

`mslowess` | `mssgolay` | `msviewer` | `mzcdfread` | `mzxmlread` | `tgspcread`

joinseq

Join two sequences to produce shortest supersequence

Syntax

SeqNT3 = joinseq(*SeqNT1*, *SeqNT2*)

Arguments

SeqNT1, *SeqNT2*	Nucleotide sequences. Enter a character vector or string for each sequence.

Description

SeqNT3 = joinseq(*SeqNT1*, *SeqNT2*) creates a new sequence that is the shortest supersequence of *SeqNT1* and *SeqNT2*. If there is no overlap between the sequences, then *SeqNT2* is concatenated to the end of *SeqNT1*. If the length of the overlap is the same at both ends of the sequence, then the overlap at the end of *SeqNT1* and the start of *SeqNT2* is used to join the sequences.

If *SeqNT1* is a subsequence of *SeqNT2*, then *SeqNT2* is returned as the shortest supersequence and vice versa.

Examples

Join two sequences that contain an overlap.

```
seq1 = 'ACGTAAA';
seq2 = 'AAATGCA';
joined = joinseq(seq1,seq2)

joined =
    ACGTAAATGCA
```

Version History

Introduced before R2006a

See Also

cat | strcat | strfind

knnimpute

Impute missing data using nearest-neighbor method

Syntax

```
imputedData = knnimpute(data)
imputedData = knnimpute(data,k)
imputedData = knnimpute(data,k,Name,Value)
```

Description

`imputedData = knnimpute(data)` returns `imputedData` after replacing `NaN`s in the input `data` with the corresponding value from the nearest-neighbor column. If the corresponding value from the nearest-neighbor column is also `NaN`, the next nearest column is used. The function calculates the Euclidean distance between observation columns by using only the rows with no `NaN` values. Thus, the data must have at least one row that contains no `NaN`.

`imputedData = knnimpute(data,k)` replaces `NaN`s in `Data` with a weighted mean of the `k` nearest-neighbor columns. The weights are inversely proportional to the distances from the neighboring columns.

`imputedData = knnimpute(data,k,Name,Value)` uses additional options specified by one or more name-value pair arguments. For example, `imputedData = knnimpute(data,k,'Distance','mahalanobis')` uses the Mahalanobis distance to compute the nearest-neighbor columns.

Examples

Impute Missing Data Using KNN

The function `knnimpute` replaces NaNs in the input data with the corresponding value from the nearest-neighbor column. Consider the following matrix.

```
A = [1 2 5;4 5 7;NaN -1 8;7 6 0]
```

```
A = 4×3

     1     2     5
     4     5     7
   NaN    -1     8
     7     6     0
```

A(3,1) is NaN, and because column 2 is the closest column to column 1 in the Euclidean distance, `knnimpute` replaces the (3,1) entry of column 1 with the corresponding entry from column 2, which is -1.

```
results = knnimpute(A)
```

```
results = 4×3
```

```
     1     2     5
     4     5     7
    -1    -1     8
     7     6     0
```

The data must have at least one row without any NaN values for `knnimpute` to work. If all rows have NaN values, you can add a row where every observation (column) has identical values and call `knnimpute` on the updated matrix to replace the NaN values with the average of all column values for a given row.

```
B = [NaN 2 1; 3 NaN 1; 1 8 NaN]
```

```
B = 3×3

   NaN     2     1
     3   NaN     1
     1     8   NaN
```

```
B(4,:) = ones(1,3)
```

```
B = 4×3

   NaN     2     1
     3   NaN     1
     1     8   NaN
     1     1     1
```

```
imputed = knnimpute(B)
```

```
imputed = 4×3

    1.5000    2.0000    1.0000
    3.0000    2.0000    1.0000
    1.0000    8.0000    4.5000
    1.0000    1.0000    1.0000
```

You can then remove the added row.

```
imputed(4,:) = []
```

```
imputed = 3×3

    1.5000    2.0000    1.0000
    3.0000    2.0000    1.0000
    1.0000    8.0000    4.5000
```

Load a sample biological data set and imputes missing values in `yeastvalues,` where each row represents each gene and each column represents an experimental condition or observation.

```
load yeastdata
```

Remove data for empty spots where gene labels are set to `'EMPTY'`.

```
emptySpots = strcmp('EMPTY',genes);
yeastvalues(emptySpots,:) = [];
```

`knnimpute` uses the next nearest column if the corresponding value from the nearest-neighbor column is also NaN. However, if all columns are NaNs, the function generates a warning for each row and keeps the rows instead of deleting the whole row in the returned output. The sample data contains some rows with all NaNs. Remove those rows to avoid the warnings.

```
yeastvalues(~any(~isnan(yeastvalues),2),:) = [];
```

Impute missing values.

```
imputedData1 = knnimpute(yeastvalues);
```

Check if there any NaN left after imputing data.

```
sum(any(isnan(imputedData1),2))
```

```
ans = 0
```

Use the 5-nearest neighbor search to get the nearest column.

```
imputedData2 = knnimpute(yeastvalues,5);
```

Change the distance metric to use the Minknowski distance.

```
imputedData3 = knnimpute(yeastvalues,5,'Distance','minkowski');
```

You can also specify the parameter for the distance metric. For instance, specify a different exponent (say 5) for the Minknowski distance.

```
imputedData4 = knnimpute(yeastvalues,5,'Distance','minkowski','DistArgs',5);
```

Input Arguments

data — Input data

matrix

Input data, specified as a matrix. The data must have at least one row that contains no `NaN` because the function calculates the Euclidean distance between observation columns by using only the rows with no `NaN` values.

Data Types: `double`

k — Number of nearest neighbors

positive integer

Number of nearest neighbors, specified as a positive integer.

Data Types: `double`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `imputedData = knnimpute(data,k,'Distance','mahalanobis')`

`Distance` — Distance metric
character vector | string | function handle

Distance metric, specified as a character vector, string, or function handle, as described in the following table.

Use the `'DistArgs'` name-value pair in conjunction to specify parameters for the distance function. For instance, to specify a different exponent (say 5) for the Minknowski distance, use: `output = knnimpute(data,3,'Distance','minkowski','DistArgs',5)`.

Value	Description
`'euclidean'`	Euclidean distance (default).
`'squaredeuclidean'`	Squared Euclidean distance. (This option is provided for efficiency only. It does not satisfy the triangle inequality.)
`'seuclidean'`	Standardized Euclidean distance. Each coordinate difference between observations is scaled by dividing by the corresponding element of the standard deviation, `S = nanstd(X)`. Use `'DistArgs'` to specify another value for `S`.
`'mahalanobis'`	Mahalanobis distance using the sample covariance of `X`, `C = nancov(X)`. Use `'DistArgs'` to specify another value for `C`, where the matrix `C` is symmetric and positive definite.
`'cityblock'`	City block distance.
`'minkowski'`	Minkowski distance. The default exponent is 2. Use `DistParameter` to specify a different exponent `P`, where `P` is a positive scalar value of the exponent.
`'chebychev'`	Chebychev distance (maximum coordinate difference).
`'cosine'`	One minus the cosine of the included angle between points (treated as vectors).
`'correlation'`	One minus the sample correlation between points (treated as sequences of values).
`'hamming'`	Hamming distance, which is the percentage of coordinates that differ.
`'jaccard'`	One minus the Jaccard coefficient, which is the percentage of nonzero coordinates that differ.
`'spearman'`	One minus the sample Spearman's rank correlation between observations (treated as sequences of values).

Value	Description
`@`*`distfun`*	Custom distance function handle. A distance function has the form `function D2 = distfun(ZI,ZJ)` `% calculation of distance` `...` where • `ZI` is a `1`-by-`n` vector containing a single observation. • `ZJ` is an `m2`-by-`n` matrix containing multiple observations. `distfun` must accept a matrix `ZJ` with an arbitrary number of observations. • `D2` is an `m2`-by-`1` vector of distances, and `D2(k)` is the distance between observations `ZI` and `ZJ(k,:)`. If your data is not sparse, you can generally compute distance more quickly by using a built-in distance instead of a function handle.

See `pdist` for more details.

Example: `'Distance','cosine'`

Data Types: `char` | `string` | `function_handle`

`DistArgs` — Distance metric parameter values
positive scalar | cell array

Distance metric parameter values, specified as a positive scalar or cell array of values. Use `'DistArgs'` together with `'Distance'` to specify parameters for the distance function. For instance, to specify a different exponent (say 5) for the Minknowski distance, use: `output = knnimpute(data,3,'Distance','minkowski','DistArgs',5)`

Example: `'DistArgs',3`

Data Types: `double` | `cell`

`Weights` — Weights used in weighted mean calculation
numeric vector of length `k`

Weights used in the weighted mean calculation, specified as a numeric vector of length `k`.

Example: `'Weights',[0.3 0.5 0.2]`

Data Types: `double`

`Median` — Flag to use median of k nearest neighbors
`true` | `false`

Flag to use the median of `k` nearest neighbors instead of the weighted mean, specified as `true` or `false`.

Example: `'Median',true`

Data Types: `logical`

Output Arguments

`imputedData` — Results after replacing NaNs
numeric matrix

Results after replacing NaNs from the input `data` with the corresponding value from the nearest-neighbor column, returned as a numeric matrix.

Version History

Introduced before R2006a

References

[1] Speed, T. (2003). Statistical Analysis of Gene Expression Microarray Data (Chapman & Hall/CRC).

[2] Hastie, T., Tibshirani, R., Sherlock, G., Eisen, M., Brown, P., and Botstein, D. (1999). “Imputing missing data for gene expression arrays”, Technical Report, Division of Biostatistics, Stanford University.

[3] Troyanskaya, O., Cantor, M., Sherlock, G., Brown, P., Hastie, T., Tibshirani, R., Botstein, D., and Altman, R. (2001). Missing value estimation methods for DNA microarrays. Bioinformatics *17(6)*, 520–525.

See Also

`isnan` | `nanmean` | `nanmedian` | `pdist`

ldivide (DataMatrix)

Left array divide DataMatrix objects

Syntax

```
DMObjNew = ldivide(DMObj1, DMObj2)
DMObjNew = DMObj1 .\ DMObj2
DMObjNew = ldivide(DMObj1, B)
DMObjNew = DMObj1 .\ B
DMObjNew = ldivide(B, DMObj1)
DMObjNew = B .\ DMObj1
```

Input Arguments

DMObj1, *DMObj2*	DataMatrix objects, such as created by `DataMatrix` (object constructor).
B	MATLAB numeric or logical array.

Output Arguments

DMObjNew	DataMatrix object created by left array division.

Description

DMObjNew = `ldivide(`*DMObj1*, *DMObj2*`)` or the equivalent *DMObjNew* = *DMObj1* `.\` *DMObj2* performs an element-by-element left array division of the DataMatrix objects *DMObj1* and *DMObj2* and places the results in *DMObjNew*, another DataMatrix object. In other words, `ldivide` divides each element in *DMObj2* by the corresponding element in *DMObj1*. *DMObj1* and *DMObj2* must have the same size (number of rows and columns), unless one is a scalar (1-by-1 DataMatrix object). The size (number of rows and columns), row names, and column names for *DMObjNew* are the same as *DMObj1*, unless *DMObj1* is a scalar; then they are the same as *DMObj2*.

DMObjNew = `ldivide(`*DMObj1*, *B*`)` or the equivalent *DMObjNew* = *DMObj1* `.\` *B* performs an element-by-element left array division of the DataMatrix object *DMObj1* and *B*, a numeric or logical array, and places the results in *DMObjNew*, another DataMatrix object. In other words, `ldivide` divides each element in *B* by the corresponding element in *DMObj1*. *DMObj1* and *B* must have the same size (number of rows and columns), unless *B* is a scalar. The size (number of rows and columns), row names, and column names for *DMObjNew* are the same as *DMObj1*.

DMObjNew = `ldivide(`*B*, *DMObj1*`)` or the equivalent *DMObjNew* = *B* `.\` *DMObj1* performs an element-by-element left array division of *B*, a numeric or logical array, and the DataMatrix object *DMObj1*, and places the results in *DMObjNew*, another DataMatrix object. In other words, `ldivide` divides each element in *DMObj1* by the corresponding element in *B*.*DMObj1* and *B* must have the same size (number of rows and columns), unless *B* is a scalar. The size (number of rows and columns), row names, and column names for *DMObjNew* are the same as *DMObj1*.

Note Arithmetic operations between a scalar DataMatrix object and a nonscalar array are not supported.

MATLAB calls `DMObjNew = ldivide(X, Y)` for the syntax `DMObjNew = X .\ Y` when *X* or *Y* is a DataMatrix object.

Version History

Introduced in R2008b

See Also

`DataMatrix` | `rdivide` | `times`

Topics

DataMatrix object on page 1-734

le (DataMatrix)

Test DataMatrix objects for less than or equal to

Syntax

```
T = le(DMObj1, DMObj2)
T = DMObj1 <= DMObj2
T = le(DMObj1, B)
T = DMObj1 <= B
T = le(B, DMObj1)
T = B <= DMObj1
```

Input Arguments

DMObj1, *DMObj2*	DataMatrix objects, such as created by `DataMatrix` (object constructor).
B	MATLAB numeric or logical array.

Output Arguments

T	Logical matrix of the same size as *DMObj1* and *DMObj2* or *DMObj1* and *B*. It contains logical `1` (true) where elements in the first input are less than or equal to the corresponding element in the second input, and logical `0` (false) otherwise.

Description

`T = le(DMObj1, DMObj2)` or the equivalent `T = DMObj1 <= DMObj2` compares each element in DataMatrix object *DMObj1* to the corresponding element in DataMatrix object *DMObj2*, and returns *T*, a logical matrix of the same size as *DMObj1* and *DMObj2*, containing logical `1` (true) where elements in *DMObj1* are less than or equal to the corresponding element in *DMObj2*, and logical `0` (false) otherwise. *DMObj1* and *DMObj2* must have the same size (number of rows and columns), unless one is a scalar (1-by-1 DataMatrix object). *DMObj1* and *DMObj2* can have different `Name` properties.

`T = le(DMObj1, B)` or the equivalent `T = DMObj1 <= B` compares each element in DataMatrix object *DMObj1* to the corresponding element in *B*, a numeric or logical array, and returns *T*, a logical matrix of the same size as *DMObj1* and *B*, containing logical `1` (true) where elements in *DMObj1* are less than or equal to the corresponding element in *B*, and logical `0` (false) otherwise. *DMObj1* and *B* must have the same size (number of rows and columns), unless one is a scalar.

`T = le(B, DMObj1)` or the equivalent `T = B <= DMObj1` compares each element in *B*, a numeric or logical array, to the corresponding element in DataMatrix object *DMObj1*, and returns *T*, a logical matrix of the same size as *B* and *DMObj1*, containing logical `1` (true) where elements in *B* are less than or equal to the corresponding element in *DMObj1*, and logical `0` (false) otherwise. *B* and *DMObj1* must have the same size (number of rows and columns), unless one is a scalar.

MATLAB calls `T = le(X, Y)` for the syntax `T = X <= Y` when *X* or *Y* is a DataMatrix object.

Version History
Introduced in R2008b

See Also
`DataMatrix` | `ge`

Topics
DataMatrix object on page 1-734

localalign

Return local optimal and suboptimal alignments between two sequences

Syntax

```
AlignStruct = localalign(Seq1, Seq2)
AlignStruct = localalign(Seq1, Seq2, ...'NumAln', NumAlnValue, ...)
AlignStruct = localalign(Seq1, Seq2, ...'MinScore', MinScoreValue, ...)
AlignStruct = localalign(Seq1, Seq2, ...'Percent', PercentValue, ...)
AlignStruct = localalign(Seq1, Seq2, ...'DoAlignment', DoAlignmentValue, ...)
AlignStruct = localalign(Seq1, Seq2, ...'Alphabet', AlphabetValue, ...)
AlignStruct = localalign(Seq1, Seq2, ...'ScoringMatrix',
ScoringMatrixValue, ...)
AlignStruct = localalign(Seq1, Seq2, ...'Scale', ScaleValue, ...)
AlignStruct = localalign(Seq1, Seq2, ...'GapOpen', GapOpenValue, ...)
```

Description

`AlignStruct = localalign(Seq1, Seq2)` returns information about the first optimal (highest scoring) local alignment between two sequences in a MATLAB structure.

`AlignStruct = localalign(Seq1, Seq2, ...'PropertyName', PropertyValue, ...)` calls `localalign` with optional properties that use property name/property value pairs. You can specify one or more properties in any order. Enclose each *`PropertyName`* in single quotation marks. Each *`PropertyName`* is case insensitive. These property name/property value pairs are as follows:

`AlignStruct = localalign(Seq1, Seq2, ...'NumAln', NumAlnValue, ...)` returns information about one or more nonintersecting, local alignments (optimal and suboptimal). It limits the number of alignments to return by specifying the number of local alignments to return. It returns the alignments in decreasing order according to their score.

`AlignStruct = localalign(Seq1, Seq2, ...'MinScore', MinScoreValue, ...)` returns information about nonintersecting, local alignments (optimal and suboptimal), whose score is greater than *`MinScoreValue`*.

`AlignStruct = localalign(Seq1, Seq2, ...'Percent', PercentValue, ...)` returns information about one or more nonintersecting local alignments (optimal and suboptimal), whose scores are within *`PercentValue`* percent of the highest score. It returns the alignments in decreasing order according to their score.

`AlignStruct = localalign(Seq1, Seq2, ...'DoAlignment', DoAlignmentValue, ...)` specifies whether to include the pairwise alignments in the `Alignment` field of the output structure. Choices are `true` (default) or `false`.

`AlignStruct = localalign(Seq1, Seq2, ...'Alphabet', AlphabetValue, ...)` specifies the type of sequences. Choices are `'AA'` (default) or `'NT'`.

`AlignStruct = localalign(Seq1, Seq2, ...'ScoringMatrix', ScoringMatrixValue, ...)` specifies the scoring matrix to use for the local alignment.

`AlignStruct = localalign(Seq1, Seq2, ...'Scale', ScaleValue, ...)` specifies a scale factor applied to the output scores, thereby controlling the units of the output scores. Choices are any positive value. Default is `1`, which does not change the units of the output score.

`AlignStruct = localalign(Seq1, Seq2, ...'GapOpen', GapOpenValue, ...)` specifies the penalty for opening a gap in the alignment. Choices are any positive value. Default is `8`.

Input Arguments

Seq1

First amino acid or nucleotide sequence specified by any of the following:

- Character vector or string of letters representing amino acids or nucleotides, such as returned by `int2aa` or `int2nt`
- Vector of integers representing amino acids or nucleotides, such as returned by `aa2int` or `nt2int`
- MATLAB structure containing a `Sequence` field, such as returned by `fastaread`, `fastqread`, `emblread`, `getembl`, `genbankread`, `getgenbank`, `getgenpept`, `genpeptread`, `getpdb`, `pdbread`, or `sffread`

Tip For help with letter and integer representations of amino acids and nucleotides, see Amino Acid Lookup or Nucleotide Lookup.

Default:

Seq2

Second amino acid or nucleotide sequence, which `localalign` aligns with *`Seq1`*.

Default:

NumAlnValue

Positive scalar (< or = `2^12`) specifying the number of alignments to return. `localalign` returns the top *`NumAlnValue`* local, nonintersecting alignments (optimal and suboptimal). If the number of optimal alignments is greater than *`NumAlnValue`*, then `localalign` returns the first *`NumAlnValue`* alignments based on their order in the trace back matrix.

Note If you specify a *`NumAlnValue`*, you cannot specify a *`MinScoreValue`* or *`PercentValue`*.

Tip Use *`NumAlnValue`* to return multiple alignments when you are aligning low complexity sequences and must consider several local alignments.

Default: `1`

MinScoreValue

Positive scalar specifying the minimum score of local, nonintersecting alignments (optimal and suboptimal) to return.

Note If you specify a *`MinScoreValue`*, you cannot specify a *`NumAlnValue`* or *`PercentValue`*.

Tip Use *`MinScoreValue`* to return suboptimal alignments, for example when you are interested in accounting for sequencing errors or imperfect scoring matrices.

Default:

`PercentValue`

Positive scalar between `0` and `100` that limits the return of local, nonintersecting alignments (optimal and suboptimal) to those alignments with a score within *`PercentValue`* percent of the highest score. For example, if the highest score is `10.5` and you specify `5` for *`PercentValue`*, then `localalign` determines a minimum score of `10.5 – (10.5 * 0.05) = 9.975`. It returns all alignments with a score of `9.975` or higher.

Note If you specify a *`PercentValue`*, you cannot specify a *`NumAlnValue`* or *`MinScoreValue`*.

Tip Use *`PercentValue`* to return optimal and suboptimal alignments when you do not know how similar the two sequences are or how well they score against a given scoring matrix.

Default:

`DoAlignmentValue`

Controls the inclusion of the pairwise alignments in the `Alignment` field of the output structure. Choices are `true` (default) or `false`.

Default:

`AlphabetValue`

Character vector or string specifying the type of sequences. Choices are `'AA'` (default) or `'NT'`.

Default:

`ScoringMatrixValue`

Either of the following:

- Character vector or string specifying the scoring matrix to use for the local alignment. Choices for amino acid sequences are:
  - `'BLOSUM62'`
  - `'BLOSUM30'` increasing by `5` up to `'BLOSUM90'`
  - `'BLOSUM100'`
  - `'PAM10'` increasing by `10` up to `'PAM500'`
  - `'DAYHOFF'`
  - `'GONNET'`

  Default is:

  - `'BLOSUM50'` — When *`AlphabetValue`* equals `'AA'`
  - `'NUC44'` — When *`AlphabetValue`* equals `'NT'`

  Note The previous scoring matrices, provided with the software, also include a structure containing a scale factor that converts the units of the output score to bits. You can also use the `'Scale'` property to specify an additional scale factor to convert the output score from bits to another unit.

- Matrix representing the scoring matrix to use for the local alignment, such as returned by the `blosum`, `pam`, `dayhoff`, `gonnet`, or `nuc44` function.

  Note If you use a scoring matrix that you created or was created by one of the previous functions, the matrix does not include a scale factor. The output score is returned in the same units as the scoring matrix. You can use the `'Scale'` property to specify a scale factor to convert the output score to another unit.

Note If you need to compile `localalign` into a stand-alone application or software component using MATLAB Compiler™, use a matrix instead of a character vector or string for *`ScoringMatrixValue`*.

Default:

`ScaleValue`

Positive value that specifies a scale factor that is applied to the output scores, thereby controlling the units of the output scores.

For example, if the output score is initially determined in bits, and you enter `log(2)` for *`ScaleValue`*, then `localalign` returns *`Score`* in nats.

Default is `1`, which does not change the units of the output score.

Note If the `'ScoringMatrix'` property also specifies a scale factor, then `localalign` uses it first to scale the output score. It then applies the scale factor specified by *`ScaleValue`* to rescale the output score.

Tip Before comparing alignment scores from multiple alignments, ensure that the scores are in the same units. Use the `'Scale'` property to control the units of the output scores.

Default:

`GapOpenValue`

Positive value specifying the penalty for opening a gap in the alignment.

Default: `8`

Output Arguments

`AlignStruct`

MATLAB structure or array of structures containing information about the local optimal and suboptimal alignments between two sequences. Each structure represents an optimal or suboptimal alignment and contains the following fields.

Field	Description
`Score`	Score for the local optimal or suboptimal alignment.
`Start`	1-by-2 vector of indices indicating the starting point in each sequence for the alignment.
`Stop`	1-by-2 vector of indices indicating the stopping point in each sequence for the alignment.
`Alignment`	3-by-N character array showing the two sequences, *`Seq1`* and *`Seq2`*, in the first and third rows. It also shows symbols representing the optimal or suboptimal local alignment between the two sequences in the second row.

Examples

Limit the number of alignments to return between two sequences by specifying the number of alignments:

```
% Create variables containing two amino acid sequences.
Seq1 = 'VSPAGMASGYDPGKA';
Seq2 = 'IPGKATREYDVSPAG';

% Use the NumAln property to return information about the
% top three local alignments.
struct1 = localalign(Seq1, Seq2, 'numaln', 3)

struct1 =

        Score: [3x1 double]
        Start: [3x2 double]
         Stop: [3x2 double]
    Alignment: {3x1 cell}

% View the scores of the first and second alignments.
struct1.Score(1:2)

ans =

   11.0000
    9.6667

% View the first alignment.
struct1.Alignment{1}

ans =

VSPAG
|||||
VSPAG
```

Limit the number of alignments to return between two sequences by specifying a minimum score:

```
% Create variables containing two amino acid sequences.
Seq1 = 'VSPAGMASGYDPGKA';
Seq2 = 'IPGKATREYDVSPAG';

% Use the MinScore property to return information about
% only local alignments with a score greater than 8.
% Use the DoAlignment property to exclude the actual alignments.
struct2 = localalign(Seq1,Seq2,'minscore',8,'doalignment',false)

struct2 =

    Score: [2x1 double]
    Start: [2x2 double]
     Stop: [2x2 double]
```

Limit the number of alignments to return between two sequences by specifying a percentage from the maximum score:

```
% Create variables containing two amino acid sequences.
Seq1 = 'VSPAGMASGYDPGKA';
Seq2 = 'IPGKATREYDVSPAG';

% Use the Percent property to return information about only
% local alignments with a score within 15% of the maximum score.
struct3 = localalign(Seq1, Seq2, 'percent', 15)

struct3 =

        Score: [2x1 double]
        Start: [2x2 double]
         Stop: [2x2 double]
    Alignment: {2x1 cell}
```

Specify a scoring matrix and gap opening penalty when aligning two sequences:

```
% Create variables containing two nucleotide sequences.
Seq1 = 'CCAATCTACTACTGCTTGCAGTAC';
Seq2 = 'AGTCCGAGGGCTACTCTACTGAAC';

% Create a scoring matrix with a match score of 10 and a mismatch
% score of -9
sm = [10 -9 -9 -9;
      -9 10 -9 -9;
      -9 -9 10 -9;
      -9 -9 -9 10];

% Use the ScoringMatrix and GapOpen properties when returning
% information about the top three local alignments.
struct4 = localalign(Seq1, Seq2, 'alpha', 'nt', ...
         'scoringmatrix', sm, 'gapopen', 20, 'numaln', 3)

struct4 =

        Score: [3x1 double]
        Start: [3x2 double]
         Stop: [3x2 double]
    Alignment: {3x1 cell}
```

More About

Nonintersecting Alignments

Alignments having no matches or mismatches in common.

Optimal Alignment

An alignment with the highest score.

Suboptimal Alignment

An alignment with a score less than the highest score.

Version History

Introduced in R2009b

References

[1] Barton, G. (1993). An efficient algorithm to locate all locally optimal alignments between two sequences allowing for gaps. CABIOS *9*, 729–734.

See Also

`multialignread` | `fastaread` | `gethmmalignment` | `seqalignviewer` | `multialign` | `nwalign` | `swalign` | `blosum` | `pam` | `dayhoff` | `gonnet` | `nuc44`

Topics
"Aligning Pairs of Sequences"
"View and Align Multiple Sequences"
Amino Acid Lookup
Nucleotide Lookup

External Websites
https://www.ncbi.nlm.nih.gov/
https://web.expasy.org/docs/swiss-prot_guideline.html
https://www.rcsb.org/
https://www.ncbi.nlm.nih.gov/Traces/sra/sra.cgi?cmd=show&f=main&m=main&s=main

lt (DataMatrix)

Test DataMatrix objects for less than

Syntax

```
T = lt(DMObj1, DMObj2)
T = DMObj1 < DMObj2
T = lt(DMObj1, B)
T = DMObj1 < B
T = lt(B, DMObj1)
T = B < DMObj1
```

Input Arguments

`DMObj1`, `DMObj2`	DataMatrix objects, such as created by `DataMatrix` (object constructor).
`B`	MATLAB numeric or logical array.

Output Arguments

`T`	Logical matrix of the same size as `DMObj1` and `DMObj2` or `DMObj1` and `B`. It contains logical `1` (true) where elements in the first input are less than the corresponding element in the second input, and logical `0` (false) otherwise.

Description

`T = lt(DMObj1, DMObj2)` or the equivalent `T = DMObj1 < DMObj2` compares each element in DataMatrix object `DMObj1` to the corresponding element in DataMatrix object `DMObj2`, and returns `T`, a logical matrix of the same size as `DMObj1` and `DMObj2`, containing logical `1` (true) where elements in `DMObj1` are less than the corresponding element in `DMObj2`, and logical `0` (false) otherwise. `DMObj1` and `DMObj2` must have the same size (number of rows and columns), unless one is a scalar (1-by-1 DataMatrix object). `DMObj1` and `DMObj2` can have different `Name` properties.

`T = lt(DMObj1, B)` or the equivalent `T = DMObj1 < B` compares each element in DataMatrix object `DMObj1` to the corresponding element in `B`, a numeric or logical array, and returns `T`, a logical matrix of the same size as `DMObj1` and `B`, containing logical `1` (true) where elements in `DMObj1` are less than the corresponding element in `B`, and logical `0` (false) otherwise. `DMObj1` and `B` must have the same size (number of rows and columns), unless one is a scalar.

`T = lt(B, DMObj1)` or the equivalent `T = B < DMObj1` compares each element in `B`, a numeric or logical array, to the corresponding element in DataMatrix object `DMObj1`, and returns `T`, a logical matrix of the same size as `B` and `DMObj1`, containing logical `1` (true) where elements in `B` are less than the corresponding element in `DMObj1`, and logical `0` (false) otherwise. `B` and `DMObj1` must have the same size (number of rows and columns), unless one is a scalar.

MATLAB calls `T = lt(X, Y)` for the syntax `T = X < Y` when `X` or `Y` is a DataMatrix object.

Version History

Introduced in R2008b

See Also

`DataMatrix` | `gt`

Topics

DataMatrix object on page 1-734

maboxplot

Create box plot for microarray data

Syntax

```
maboxplot(MAData)
maboxplot(MAData, ColumnName)
maboxplot(MAStruct, FieldName)
H = maboxplot(...)
[H, HLines] = maboxplot(...)

maboxplot(..., 'Title', TitleValue, ...)
maboxplot(..., 'Notch', NotchValue, ...)
maboxplot(..., 'Symbol', SymbolValue, ...)
maboxplot(..., 'Orientation', OrientationValue, ...)
maboxplot(..., 'WhiskerLength', WhiskerLengthValue, ...)
maboxplot(..., 'BoxPlot', BoxPlotValue, ...)
```

Arguments

MAData	DataMatrix object on page 1-734, numeric array, or a structure containing a field called `Data`. The values in the columns of *MAData* will be used to create box plots. If a DataMatrix object, the column names are used as labels in the box plot.
ColumnName	An array of column names corresponding to the data in *MAData* used as labels in the box plot.
MAStruct	A microarray data structure.
FieldName	A field within the microarray data structure, *MAStruct*. The values in the field *FieldName* will be used to create box plots.
TitleValue	Character vector or string to use as the title for the plot. The default title is *FieldName*.
NotchValue	Logical specifying the type of boxes drawn. Choices are: • `true` — Notched boxes • `false` — Square boxes Default is `false`.
OrientationValue	Character vector or string specifying the orientation of the box plot. Choices are: • `'Vertical'` • `'Horizontal'` (default)

WhiskerLengthValue	Value specifying the maximum length of the whiskers as a function of the interquartile range (IQR). The whisker extends to the most extreme data value within *WhiskerLengthValue**IQR of the box. Default = 1.5. If *WhiskerLengthValue* equals 0, then maboxplot displays all data values outside the box, using the plotting symbol Symbol.
BoxPlotValue	A cell array of property name/property value pairs to pass to the Statistics and Machine Learning Toolbox™ boxplot function, which creates the box plot. For valid pairs, see the boxplot function.

Description

maboxplot(*MAData*) displays a box plot of the values in the columns of *MAData*. *MAData* can be a DataMatrix object on page 1-734, numeric array, or a structure containing a field called Data, containing microarray data.

maboxplot(*MAData, ColumnName*) labels the box plot column names.

maboxplot(*MAStruct, FieldName*) displays a box plot of the values in the field FieldName in the microarray data structure *MAStruct*. If *MAStruct* is block based, maboxplot creates a box plot of the values in the field *FieldName* for each block.

Note If you provide *MAStruct*, without providing *FieldName*, maboxplot uses the Signal element in the ColumnNames field of *MAStruct*, if Affymetrixdata, or the first element in the ColumnNames field of *MAStruct*, otherwise.

H = maboxplot(...) returns the handle of the box plot axes.

[*H*, *HLines*] = maboxplot(...) returns the handles of the lines used to separate the different blocks in the image.

maboxplot(..., '*PropertyName*', *PropertyValue*, ...) calls maboxplot with optional properties that use property name/property value pairs. You can specify one or more properties in any order. Each *PropertyName* must be enclosed in single quotation marks and is case insensitive. These property name/property value pairs are as follows:

maboxplot(..., 'Title', *TitleValue*, ...) allows you to specify the title of the plot. The default *TitleValue* is FieldName.

maboxplot(..., 'Notch', *NotchValue*, ...) if *NotchValue* is true, draws notched boxes. The default is false to show square boxes.

maboxplot(..., 'Symbol', *SymbolValue*, ...) allows you to specify the symbol used for outlier values. The default Symbol is '+'.

maboxplot(..., 'Orientation', *OrientationValue*, ...) allows you to specify the orientation of the box plot. The choices are 'Vertical' and 'Horizontal'. The default is 'Vertical'.

maboxplot(..., 'WhiskerLength', *WhiskerLengthValue*, ...) allows you to specify the whisker length for the box plot. *WhiskerLengthValue* defines the maximum length of the whiskers as a function of the interquartile range (IQR) (default = `1.5`). The whisker extends to the most extreme data value within `WhiskerLength*IQR` of the box. If *`WhiskerLengthValue`* equals `0`, then `maboxplot` displays all data values outside the box, using the plotting symbol `Symbol`.

`maboxplot(..., 'BoxPlot', `*`BoxPlotValue`*`, ...)` allows you to specify arguments to pass to the `boxplot` function, which creates the box plot. *`BoxPlotValue`* is a cell array of property name/property value pairs. For valid pairs, see the `boxplot` function.

Examples

Display Box Plots for Microarray Data

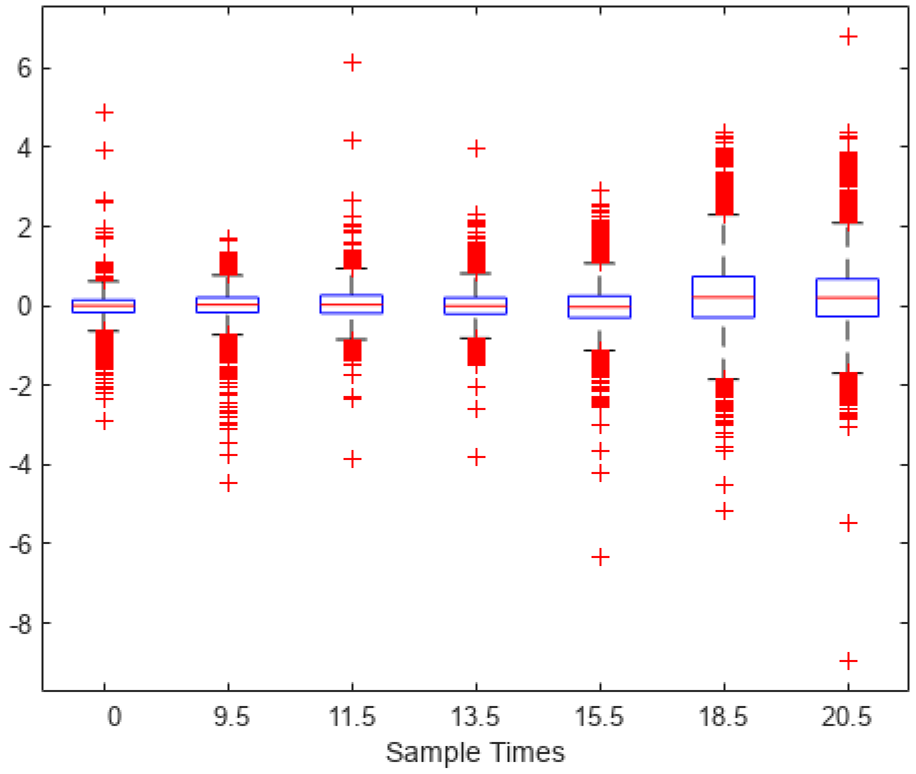
This example shows how to display box plots for microarray data.

Load the MAT-file, provided with the Bioinformatics Toolbox™ software, that contains yeast data. This MAT-file includes three variables: `yeastvalues` , a matrix of gene expression data, `genes` , a cell array of GenBank® accession numbers for labeling the rows in yeastvalues, and `times` , a vector of time values for labeling the columns in yeastvalues.

```
load yeastdata
```

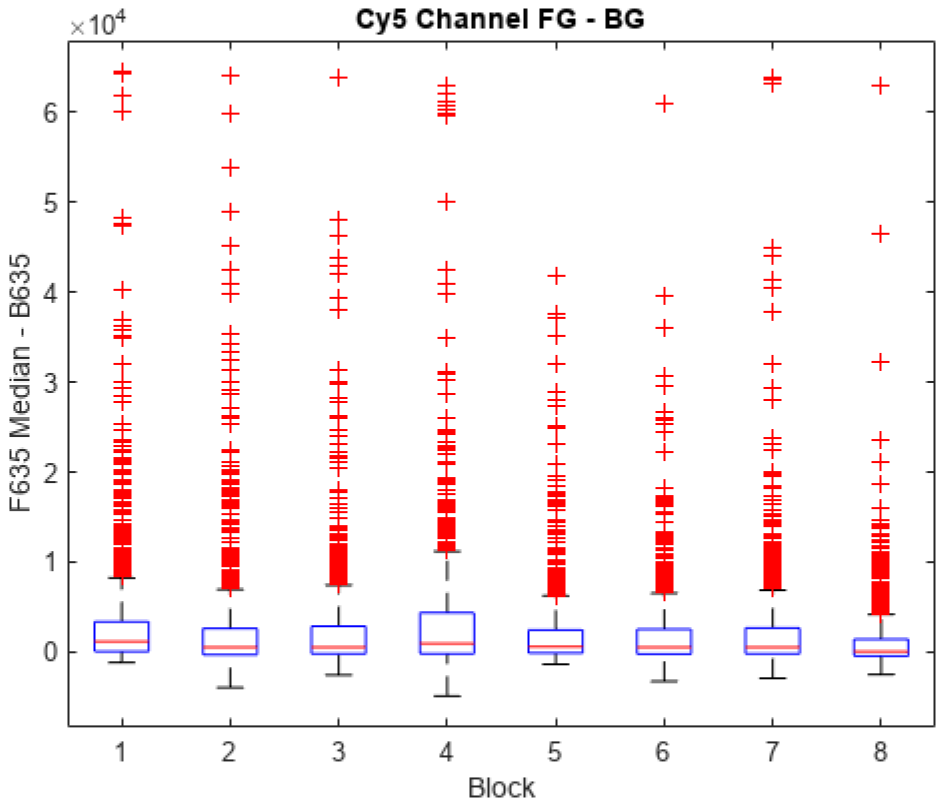
Show the box plot of gene expression data.

```
maboxplot(yeastvalues,times);
xlabel('Sample Times');
```



Use the `gprread` function to create a structure containing microarray data, and plot the data using name-value pair arguments of the `maboxplot` function.

```
madata = gprread('mouse_a1wt.gpr');
maboxplot(madata,'F635 Median - B635','TITLE', 'Cy5 Channel FG - BG');
```



Version History

Introduced before R2006a

See Also

`magetfield` | `maimage` | `mairplot` | `maloglog` | `malowess` | `manorm` | `mavolcanoplot` | `boxplot`

mafdr

Estimate positive false discovery rate for multiple hypothesis testing

Syntax

```
FDR = mafdr(PValues)
FDR = mafdr(PValues,Name,Value)
[FDR,Q] = mafdr(PValues,___)
[FDR,Q,aPrioriProb] = mafdr(PValues,___)
[FDR,Q,aPrioriProb,R_squared] = mafdr(PValues,'Method','polynomial',___)
```

Description

`FDR = mafdr(PValues)` returns `FDR` that contains a positive false discovery rate (pFDR) for each entry in `PValues` using the procedure introduced by Storey (2002) [1]. `PValues` contains one p-value for each feature (for example, a gene) in a data set.

`FDR = mafdr(PValues,Name,Value)` uses additional options specified by one or more name-value pair arguments. For example, `'Showplot',true` displays diagnostic plots of calculated results.

`[FDR,Q] = mafdr(PValues,___)` also returns hypothesis testing error measures `Q` for all p-values. Optionally, you can specify one or more name-value pair arguments.

`[FDR,Q,aPrioriProb] = mafdr(PValues,___)` also returns `aPrioriProb`, the estimated *a priori* probability that the null hypothesis $\widehat{\pi}_0$ is true.

`[FDR,Q,aPrioriProb,R_squared] = mafdr(PValues,'Method','polynomial',___)` also returns `R_squared`, the square of correlation coefficient. Use the polynomial method to get the R-squared value.

Examples

Estimate Positive False Discovery Rate for Multiple Hypothesis Testing

Estimate the positive FDR using data from a prostate cancer study (Best et al., 2005). The data contains probe intensity data from Affymetrix® HG-U133A GeneChip® arrays.

Load the gene expression data. It contains two variables, `dependentData` and `independentData` that are two matrices of gene expression values from two experimental conditions.

```
load prostatecancerexpdata
```

Use `mattest` to calculate the p-values for gene expression values in the two matrices.

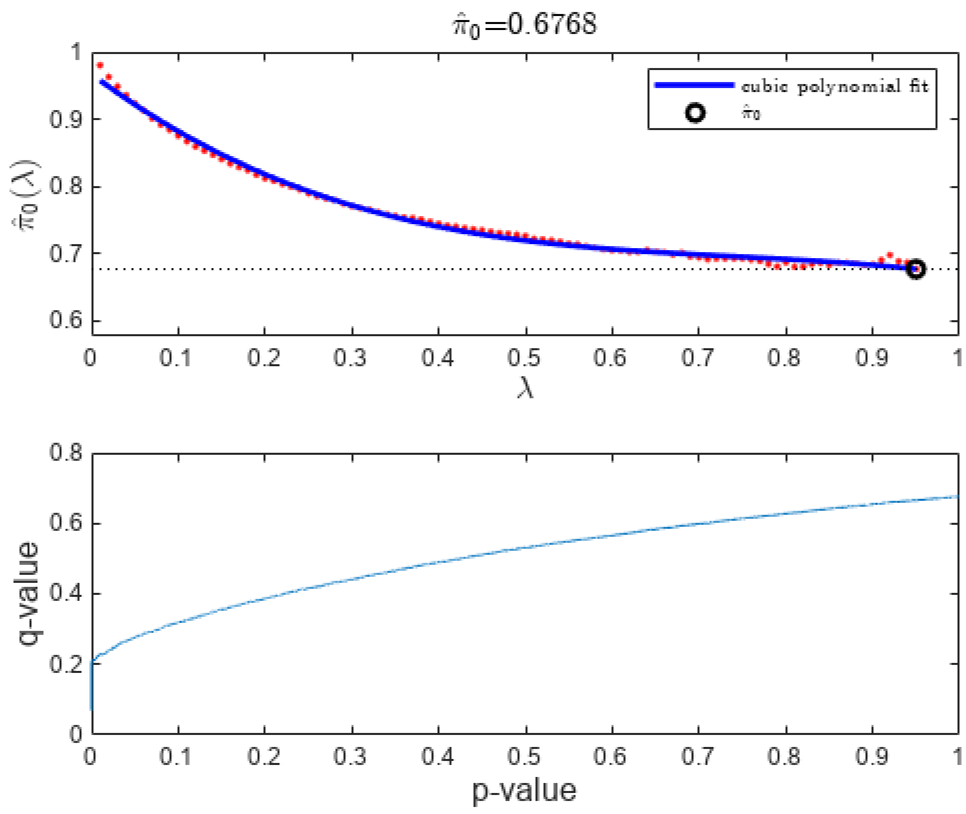
```
pvalues = mattest(dependentData,independentData,'permute',true);
```

Use `mafdr` to calculate the positive FDR values.

```
fdr = mafdr(pvalues);
```

Calculate the q-values, *a priori* probability (that the null hypothesis is true), and R-squared value. You must use the polynomial method to get the R-squared value. Plot the data by setting `'Showplot'` to `true`.

```
[fdr,q,priori,R2] = mafdr(pvalues,'Method','polynomial','Showplot',true);
```



Input Arguments

PValues — P-values for all features

column vector | `DataMatrix` object

P-values for all features in a data set, specified as a column vector or a DataMatrix object on page 1-734. You can use the first output of the `mattest` function.

Data Types: `double`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `fdr = mafdr(pvals,'Lambda',0.5,'Showplot',true)` specifies the tuning parameter value of 0.5 to estimate a prior probability and displays the quality statistics plots.

`BHFDR` — Flag to use linear step-up procedure
`false` (default) | `true`

Flag to use the linear step-up procedure introduced by Benjamini and Hochberg (1995) [2], specified as the comma-separated pair consisting of `'BHFDR'` and `true` or `false`. The default value is `false`, that is, the function uses the procedure introduced by Storey (2002) [1].

If `true`:

- The function uses the Benjamini and Hochberg method.
- The function ignores the `'Method'` and `'Lambda'` name-value pair arguments.
- Specify only one output argument, that is, `FDR`.
- If you also set `'Showplot'` to `true`, then the function plots only the q-values versus p-values. For details, see “Showplot” on page 1-0 .

Example: `'BHFDR',true`

Data Types: `logical`

`Lambda` — Tuning parameter
`[0.01:0.01:0.95]` (default) | positive scalar | vector

Tuning parameter used to estimate the *a priori* probability that the null hypothesis is true, specified as the comma-separated pair consisting of `'Lambda'` and a positive scalar or vector with four or more values. The scalar value or each value in the vector must be between 0 and 1.

- If you specify a single value, then the function ignores the `'Method'` name-value pair argument.
- If you specify a vector of values, then the function chooses the optimal value using the method specified by the `'Method'` name-value pair argument.

Example: `'Lambda'[0.01:0.1:0.95]`

Data Types: `double`

`Method` — Method to choose Lambda value
`'bootstrap'` (default) | `'polynomial'`

Method to choose the Lambda value from a range of values, specified as the comma-separated pair consisting of `'Method'` and `'bootstrap'` or `'polynomial'`.

Example: `'Method','polynomial'`

Data Types: `char` | `string`

`Showplot` — Flag to display diagnostic plots
`false` (default) | `true`

Flag to display two diagnostic plots, specified as the comma-separated pair consisting of `'Showplot'` and `true` or `false`.

If true, the function displays two plots:

- Estimated *a priori* probability that the null hypothesis $\widehat{\pi}_0(\lambda)$ is true versus the tuning parameter (λ) with a cubic polynomial fitting curve

- q-values versus p-values

If you also set `'BHFDR'` to `true`, the function displays only the second plot.

Example: `'Showplot',true`

Data Types: `logical`

Output Arguments

`FDR` — Positive FDR values
vector | `DataMatrix` object

Positive FDR values, returned as a vector or DataMatrix object.

If `PValues` is a column vector, then `FDR` is a column vector.

If `PValues` is a `DataMatrix` object, then `FDR` is a `DataMatrix` object.

`Q` — Q-values
column vector

Q-values, returned as a column vector. `Q` contains the measures of hypothesis testing error for all observations in `PValues`.

`aPrioriProb` — Estimated a priori probability
positive scalar

Estimated *a priori* probability that the null hypothesis $\widehat{\pi}_0$ is true, returned as a positive scalar.

`R_squared` — Square of correlation coefficient
positive scalar

Square of the correlation coefficient, returned as a positive scalar. Specify `'Method'` as `'polynomial'` to get this fourth output.

Version History

Introduced in R2007a

References

[1] Storey, John D. "A Direct Approach to False Discovery Rates." *Journal of the Royal Statistical Society: Series B (Statistical Methodology)* 64, no. 3 (August 2002): 479–98.

[2] Benjamini, Y., and Hochberg, Y. 1995. Controlling the false discovery rate: A practical and powerful approach to multiple testing. J. Royal Stat. Soc. 57:289–300.

[3] Best, C.J.M., Gillespie, J.W., Yi, Y., Chandramouli, G.V.R., Perlmutter, M.A., Gathright, Y., Erickson, H.S., Georgevich, L., Tangrea, M.A., Duray, P.H., Gonzalez, S., Velasco, A., Linehan, W.M., Matusik, R.J., Price, D.K., Figg, W.D., Emmert-Buck, M.R., and Chuaqui, R.F. 2005. Molecular alterations in primary prostate cancer after androgen ablation therapy. Clin. Cancer Res. 11:6823–6831.

[4] Storey, J.D., and Tibshirani, R. 2003. Statistical significance for genomewide studies. Proc. Nat. Acad. Sci. 100:9440–9445.

[5] Storey, J.D., Taylor, J.E., and Siegmund, D. 2004. Strong control, conservative point estimation and simultaneous conservative consistency of false discovery rates: A unified approach. J. Royal Stat. Soc. 66:187–205.

See Also

`affygcrma` | `affyrma` | `gcrma` | `mairplot` | `maloglog` | `mapcaplot` | `mattest` | `mavolcanoplot` | `rmasummary`

magetfield

Extract data from microarray structure

Syntax

`magetfield(MAStruct, FieldName)`

Arguments

MAStruct	Microarray structure.
FieldName	A column in *MAStruct*.

Description

`magetfield(MAStruct, FieldName)` extracts data for *FieldName*, a column in *MAStruct*, microarray structure.

The benefit of this function is to hide the details of extracting a column of data from a structure created with one of the microarray reader functions (`gprread`, `agferead`, `sptread`, `imageneread`).

Examples

```
maStruct = gprread('mouse_a1wt.gpr');
cy5data = magetfield(maStruct,'F635 Median');
cy3data = magetfield(maStruct,'F532 Median');
mairplot(cy5data,cy3data,'title','R vs G IR plot');
```

Version History

Introduced before R2006a

See Also

`agferead` | `gprread` | `ilmnbsread` | `imageneread` | `maboxplot` | `mairplot` | `maloglog` | `malowess` | `sptread`

maimage

Spatial image for microarray data

Syntax

```
maimage(X, FieldName)
H = maimage(...)
[H, HLines] = maimage(...)

maimage(..., 'PropertyName', PropertyValue,...)
maimage(..., 'Title', TitleValue)
maimage(..., 'ColorBar', ColorBarValue)
maimage(..., 'HandleGraphicsPropertyName' PropertyValue)
```

Arguments

X	A microarray data structure.
FieldName	A field in the microarray data structure *X*.
TitleValue	A character vector or string to use as the title for the plot. The default title is `FieldName`.
ColorBarValue	Property to control displaying a color bar in the Figure window. Enter either `true` or `false`. The default value is `false`.

Description

`maimage(`*`X`*`, `*`FieldName`*`)` displays an image of field *`FieldName`* from microarray data structure *`X`*. Microarray data can be GenPix Results (GPR) format. After creating the image, click a data point to display the value and ID, if known.

`H = maimage(...)` returns the handle of the image.

`[H, HLines] = maimage(...)` returns the handles of the lines used to separate the different blocks in the image.

`maimage(..., '`*`PropertyName`*`', `*`PropertyValue`*`,...)` defines optional properties using property name/value pairs.

`maimage(..., 'Title', `*`TitleValue`*`)` allows you to specify the title of the plot. The default title is `FieldName`.

`maimage(..., 'ColorBar', `*`ColorBarValue`*`)`, when *`ColorBarValue`* is `true`, a color bar is shown. If *`ColorBarValue`* is `false`, no color bar is shown. The default is for the color bar to be shown.

`maimage(..., '`*`HandleGraphicsPropertyName`*`' `*`PropertyValue`*`)` allows you to pass optional Handle Graphics® property name/value pairs to the function. For example, a name/value pair for color could be `maimage(..., 'color' 'r')`.

Examples

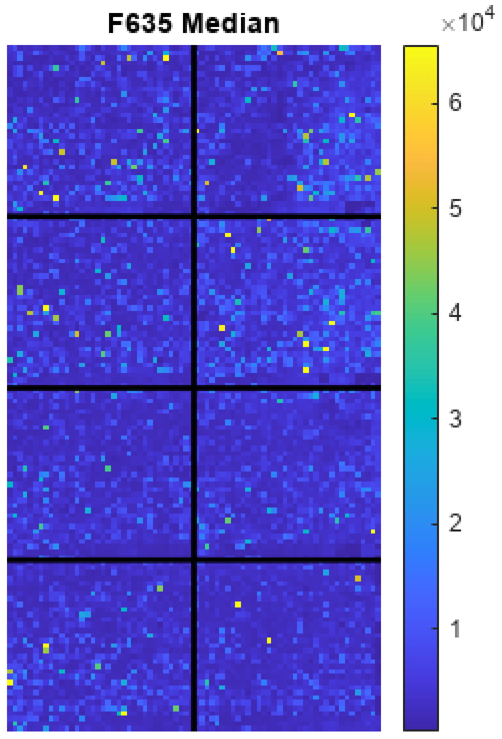
Generate Spatial Image and Change Colormap of Microarray Data

Read in a sample GPR file.

```
madata = gprread('mouse_a1wt.gpr');
```

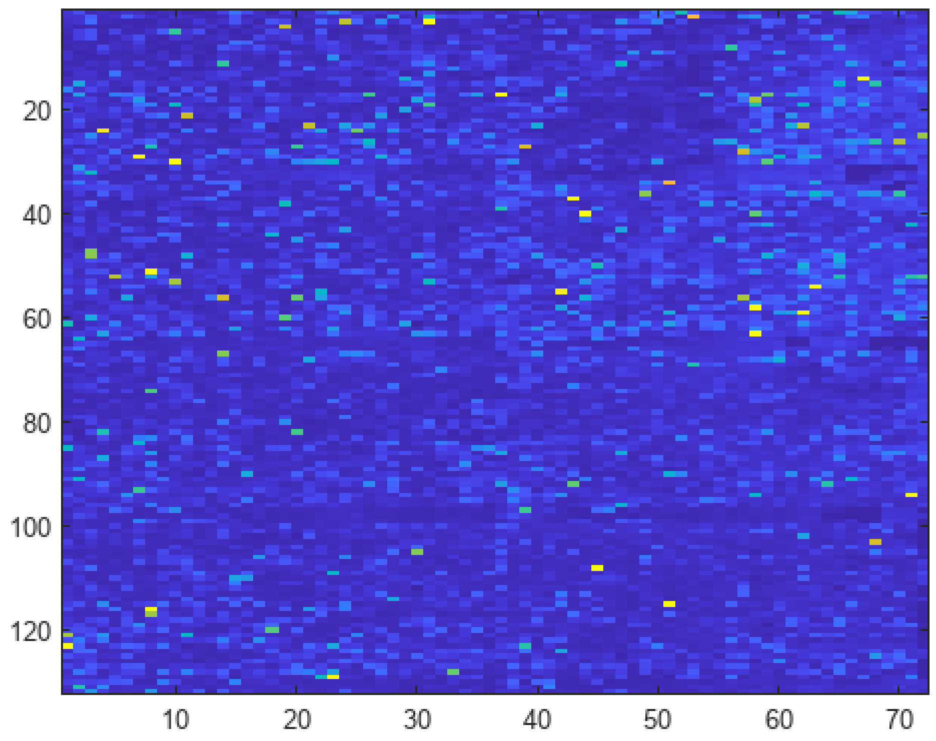
Plot the median foreground intensity for the 635 nm channel.

```
maimage(madata,'F635 Median')
```



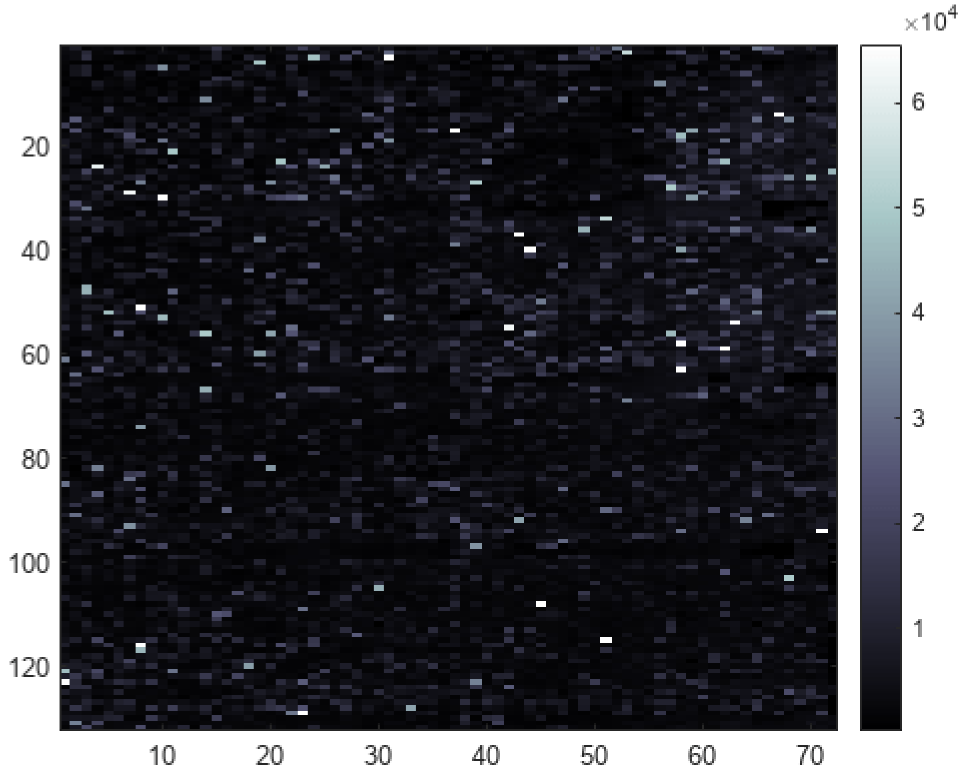
Alternatively, create a similar plot using more basic graphics commands.

```
F635Median = magetfield(madata,'F635 Median');
figure
imagesc(F635Median(madata.Indices));
```



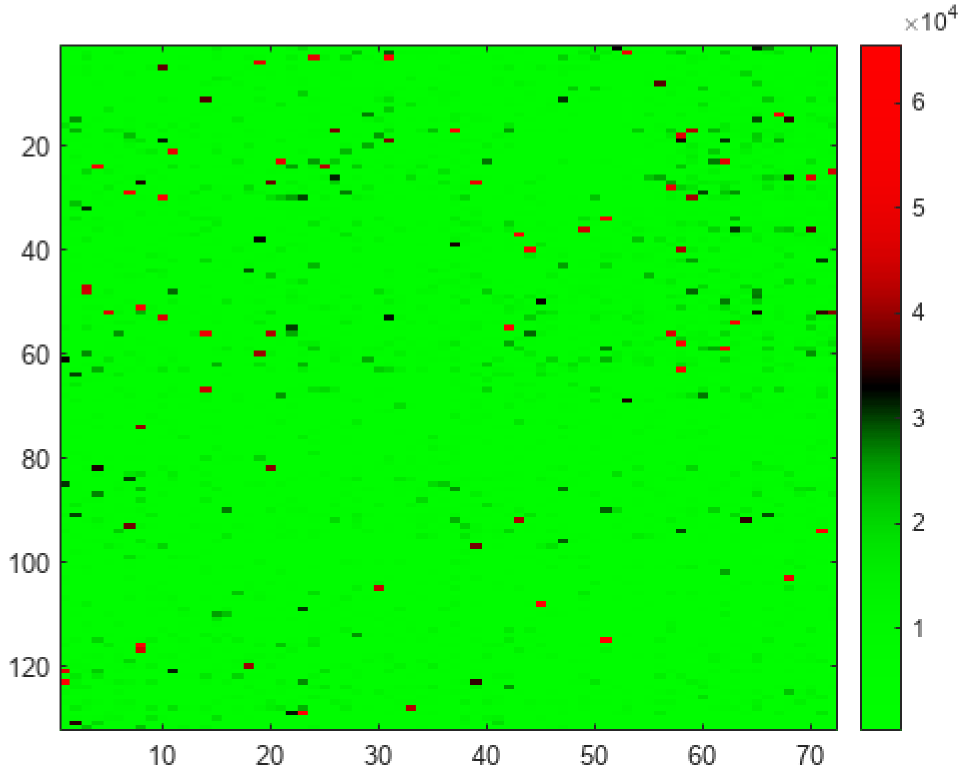
Change the colormap to one of the preset colors (for details, see “map”) and add a color bar.

```
colormap('bone')
colorbar
```



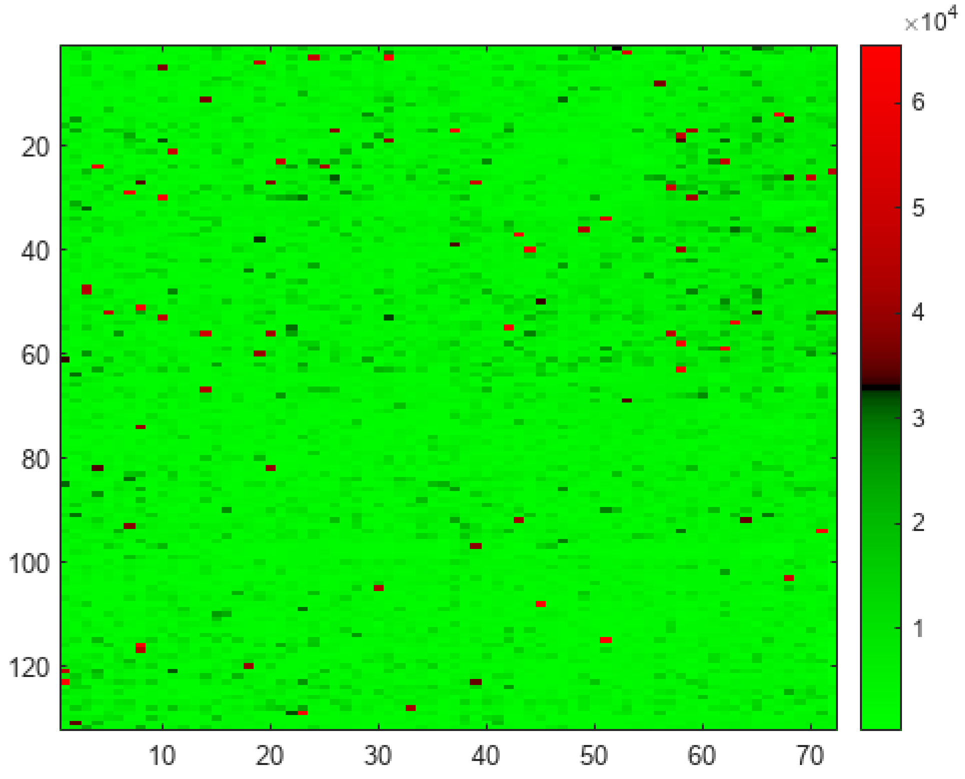
Change the colormap to red and green.

```
colormap(redgreencmap)
```



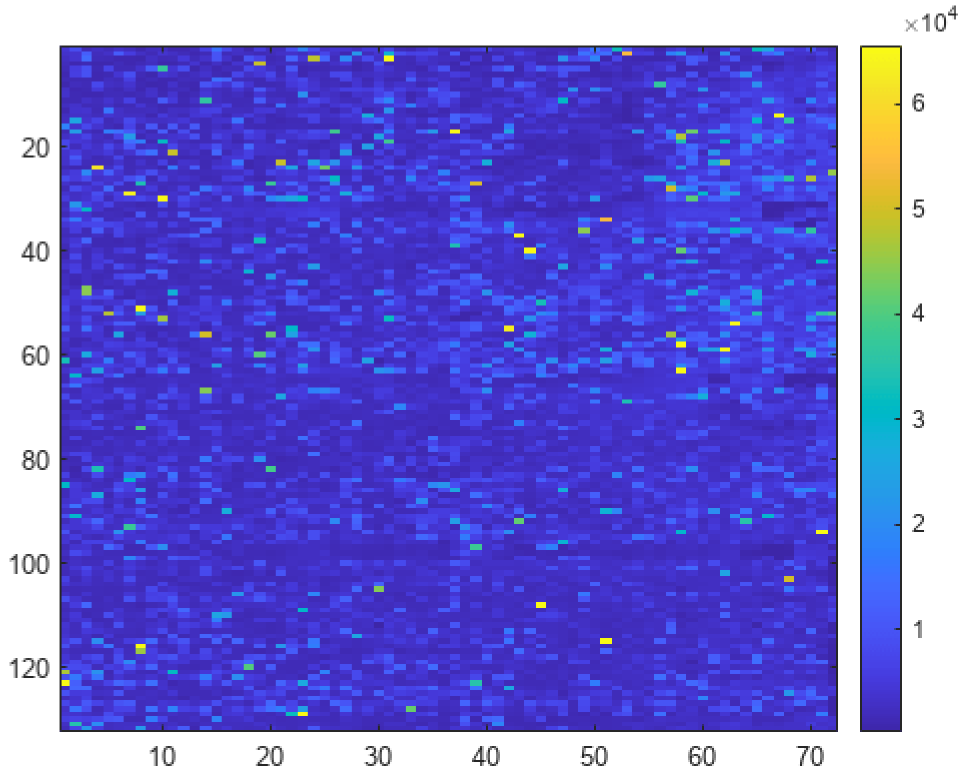
You can also change the color interpolation method.

```
colormap(redgreencmap(256,'Interpolation','cubic'))
```



Reset the colormap to the default value.

```
colormap('default')
```



Version History

Introduced before R2006a

See Also

`maboxplot` | `magetfield` | `mairplot` | `maloglog` | `malowess` | `imagesc`

mainvarsetnorm

Perform rank invariant set normalization on gene expression values from two experimental conditions or phenotypes

Syntax

```
NormDataY = mainvarsetnorm(DataX, DataY)

NormDataY = mainvarsetnorm(..., 'Thresholds', ThresholdsValue, ...)
NormDataY = mainvarsetnorm(..., 'Exclude', ExcludeValue, ...)
NormDataY = mainvarsetnorm(..., 'Percentile', PercentileValue, ...)
NormDataY = mainvarsetnorm(..., 'Iterate', IterateValue, ...)
NormDataY = mainvarsetnorm(..., 'Method', MethodValue, ...)
NormDataY = mainvarsetnorm(..., 'Span', SpanValue, ...)
NormDataY = mainvarsetnorm(..., 'Showplot', ShowplotValue, ...)
```

Arguments

DataX	Vector of gene expression values from a single experimental condition or phenotype, where each row corresponds to a gene. These data points are used as the baseline.
DataY	Vector of gene expression values from a single experimental condition or phenotype, where each row corresponds to a gene. These data points will be normalized using the baseline.
ThresholdsValue	Vector that sets the thresholds for the lowest average rank and the highest average rank between the two data sets. The average rank for each data point is determined by first converting the values in *DataX* and *DataY* to ranks, then averaging the two ranks for each data point. Then, the threshold for each data point is determined by interpolating between the threshold for the lowest average rank and the threshold for the highest average rank. **Note** These individual thresholds are used to determine the rank invariant set, which is a set of data points, each having a proportional rank difference (prd) smaller than its predetermined threshold. For more information on the rank invariant set, see “Description” on page 1-1195. *ThresholdsValue* is a 1-by-2 vector [*LT*, *HT*], where *LT* is the threshold for the lowest average rank and *HT* is threshold for the highest average rank. Select these two thresholds empirically to limit the spread of the invariant set, but allow enough data points to determine the normalization relationship. Values must be between `0` and `1`. Default is [`0.03, 0.07`].

ExcludeValue	Property to filter the invariant set of data points, by excluding the data points whose average rank (between *DataX* and *DataY*) is in the highest *N* ranked averages or lowest *N* ranked averages.
PercentileValue	Property to stop the iteration process when the number of data points in the invariant set reaches *N* percent of the total number of input data points. Default is `1`. **Note** If you do not use this property, the iteration process continues until no more data points are eliminated.
IterateValue	Property to control the iteration process for determining the invariant set of data points. Enter `true` to repeat the process until either no more data points are eliminated, or a predetermined percentage of data points (*PercentileValue*) is reached. Enter `false` to perform only one iteration of the process. Default is `true`. **Tip** Select `false` for smaller data sets, typically less than 200 data points.
MethodValue	Property to select the smoothing method used to normalize the data. Enter `'lowess'` or `'runmedian'`. Default is `'lowess'`.
SpanValue	Property to set the window size for the smoothing method. If *SpanValue* is less than 1, the window size is that percentage of the number of data points. If *SpanValue* is equal to or greater than 1, the window size is of size *SpanValue*. Default is `0.05`, which corresponds to a window size equal to 5% of the total number of data points in the invariant set.
ShowplotValue	Property to control the plotting of a pair of M-A scatter plots (before and after normalization). M is the ratio between *DataX* and *DataY*. A is the average of *DataX* and *DataY*. Enter `true` to create the pair of M-A scatter plots. Default is `false`.

Description

`NormDataY = mainvarsetnorm(DataX, DataY)` normalizes the values in *DataY*, a vector of gene expression values, to a reference vector, *DataX*, using the invariant set method. *NormDataY* is a vector of normalized gene expression values from *DataY*.

Specifically, `mainvarsetnorm`:

- Determines the proportional rank difference (*prd*) for each pair of ranks, *RankX* and *RankY*, from the two vectors of gene expression values, *DataX* and *DataY*.

  prd = abs(*RankX* - *RankY*)

- Determines the invariant set of data points by selecting data points whose proportional rank differences (*prd*) are below *threshold*, which is a predetermined threshold for a given data point (defined by the *ThresholdsValue* property). It optionally repeats the process until either no more data points are eliminated, or a predetermined percentage of data points is reached.

  The invariant set is data points with a *prd* < *threshold*.

- Uses the invariant set of data points to calculate the lowess or running median smoothing curve, which is used to normalize the data in `DataY`.

Note If `DataX` or `DataY` contains NaN values, then `NormDataY` will also contain NaN values at the corresponding positions.

Tip `mainvarsetnorm` is useful for correcting for dye bias in two-color microarray data.

`NormDataY = mainvarsetnorm(..., 'PropertyName', PropertyValue, ...)` calls `mainvarsetnorm` with optional properties that use property name/property value pairs. You can specify one or more properties in any order. Each `PropertyName` must be enclosed in single quotation marks and is case insensitive. These property name/property value pairs are as follows:

`NormDataY = mainvarsetnorm(..., 'Thresholds', ThresholdsValue, ...)` sets the thresholds for the lowest average rank and the highest average rank between the two data sets. The average rank for each data point is determined by first converting the values in `DataX` and `DataY` to ranks, then averaging the two ranks for each data point. Then, the threshold for each data point is determined by interpolating between the threshold for the lowest average rank and the threshold for the highest average rank.

Note These individual thresholds are used to determine the rank invariant set, which is a set of data points, each having a proportional rank difference (prd) smaller than its predetermined threshold. For more information on the rank invariant set, see “Description” on page 1-1195.

`ThresholdsValue` is a 1-by-2 vector [`LT, HT`], where `LT` is the threshold for the lowest average rank and `HT` is threshold for the highest average rank. Select these two thresholds empirically to limit the spread of the invariant set, but allow enough data points to determine the normalization relationship. Values must be between `0` and `1`. Default is [`0.03, 0.07`].

`NormDataY = mainvarsetnorm(..., 'Exclude', ExcludeValue, ...)` filters the invariant set of data points, by excluding the data points whose average rank (between `DataX` and `DataY`) is in the highest `N` ranked averages or lowest `N` ranked averages.

`NormDataY = mainvarsetnorm(..., 'Percentile', PercentileValue, ...)` stops the iteration process when the number of data points in the invariant set reaches `N` percent of the total number of input data points. Default is `1`.

Note If you do not use this property, the iteration process continues until no more data points are eliminated.

`NormDataY = mainvarsetnorm(..., 'Iterate', IterateValue, ...)` controls the iteration process for determining the invariant set of data points. When `IterateValue` is `true`, `mainvarsetnorm` repeats the process until either no more data points are eliminated, or a predetermined percentage of data points (`PercentileValue`) is reached. When `IterateValue` is `false`, performs only one iteration of the process. Default is `true`.

Tip Select `false` for smaller data sets, typically less than 200 data points.

`NormDataY = mainvarsetnorm(..., 'Method', MethodValue, ...)` selects the smoothing method for normalizing the data. When *`MethodValue`* is `'lowess'`, `mainvarsetnorm` uses the lowess method. When *`MethodValue`* is `'runmedian'`, `mainvarsetnorm` uses the running median method. Default is `'lowess'`.

`NormDataY = mainvarsetnorm(..., 'Span', SpanValue, ...)` sets the window size for the smoothing method. If *`SpanValue`* is less than 1, the window size is that percentage of the number of data points. If *`SpanValue`* is equal to or greater than 1, the window size is of size *`SpanValue`*. Default is `0.05`, which corresponds to a window size equal to 5% of the total number of data points in the invariant set.

`NormDataY = mainvarsetnorm(..., 'Showplot', ShowplotValue, ...)` determines whether to plot a pair of M-A scatter plots (before and after normalization). M is the ratio between *`DataX`* and *`DataY`*. A is the average of *`DataX`* and *`DataY`*. When *`ShowplotValue`* is `true`, `mainvarsetnorm` plots the M-A scatter plots. Default is `false`.

Examples

Normalize Microarray Data

This example illustrates how to correct for dye bias or scanning differences between two channels of data from a two-color microarray experiment.

Read microarray data from a sample GPR file.

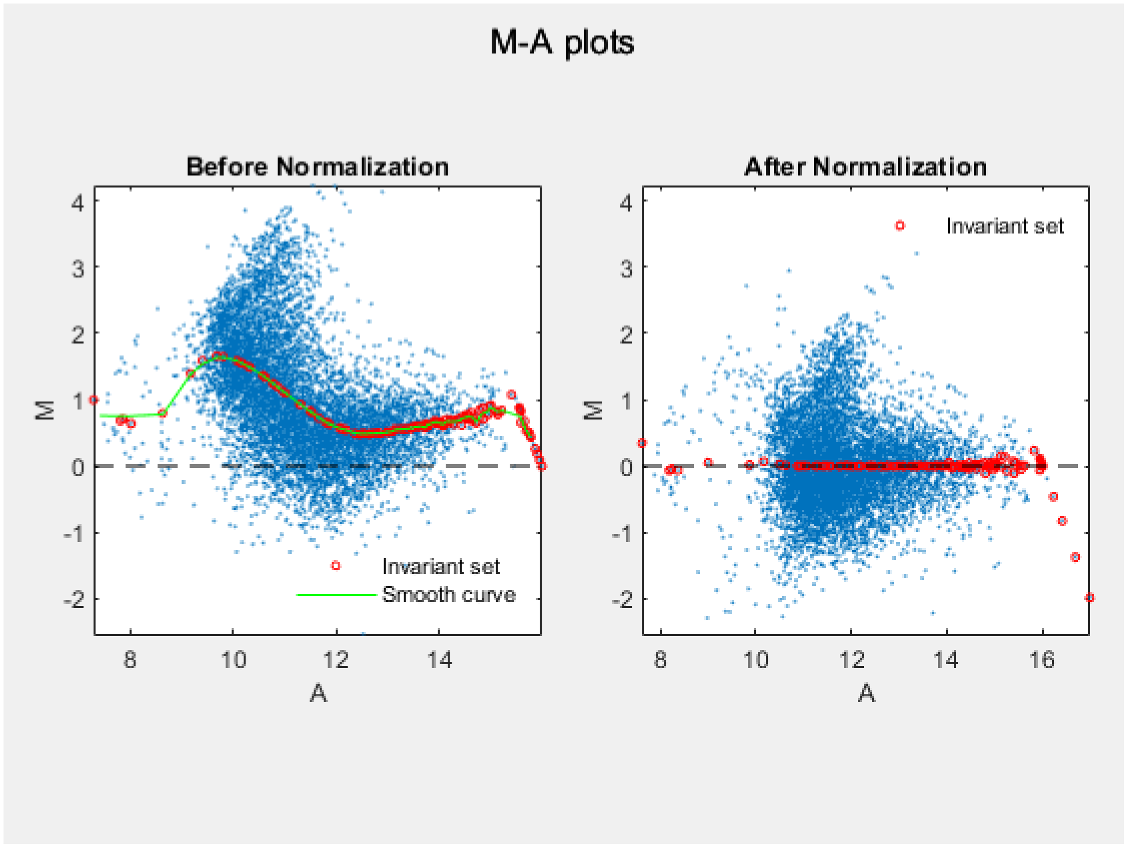
```
maStruct = gprread('mouse_a1wt.gpr');
```

Extract gene expression values from two different experimental conditions.

```
cy5data = magetfield(maStruct, 'F635 Median');
cy3data = magetfield(maStruct, 'F532 Median');
```

Normalize `cy3data` using `cy5data` as reference and plot the results.

```
Normcy3data = mainvarsetnorm(cy5data, cy3data, 'showplot', true);
```



Under perfect experimental conditions, data points with equal expression values would fall along the M = 0 line, which represents a gene expression ratio of 1. However, dye bias caused the measured values in one channel to be higher than the other channel, as seen in the Before normalization plot. Normalization corrected the variance, as seen in the After normalization plot.

Version History

Introduced in R2006a

References

[1] Tseng, G.C., Oh, Min-Kyu, Rohlin, L., Liao, J.C., and Wong, W.H. (2001) Issues in cDNA microarray analysis: quality filtering, channel normalization, models of variations and assessment of gene effects. Nucleic Acids Research. *29*, 2549-2557.

[2] Hoffmann, R., Seidl, T., and Dugas, M. (2002) Profound effect of normalization on detection of differentially expressed genes in oligonucleotide microarray data analysis. Genome Biology. *3(7)*: research 0033.1-0033.11.

See Also

`affyinvarsetnorm` | `malowess` | `manorm` | `quantilenorm`

mairplot

Create intensity versus ratio scatter plot of microarray data

Syntax

```
mairplot(DataX, DataY)
[Intensity, Ratio] = mairplot(DataX, DataY)
[Intensity, Ratio, H] = mairplot(DataX, DataY)

... = mairplot(..., 'Type', TypeValue, ...)
... = mairplot(..., 'LogTrans', LogTransValue, ...)
... = mairplot(..., 'FactorLines', FactorLinesValue, ...)
... = mairplot(..., 'Title', TitleValue, ...)
... = mairplot(..., 'Labels', LabelsValue, ...)
... = mairplot(..., 'Normalize', NormalizeValue, ...)
... = mairplot(..., 'LowessOptions', LowessOptionsValue, ...)
... = mairplot(..., 'Showplot', ShowplotValue, ...)
... = mairplot(..., 'PlotOnly', PlotOnlyValue, ...)
```

Input Arguments

DataX, *DataY*	DataMatrix object on page 1-734 or vector of gene expression values where each row corresponds to a gene. For example, in a two-color microarray experiment, *DataX* could be cy3 intensity values and *DataY* could be cy5 intensity values.
TypeValue	Character vector or string that specifies the plot type. Choices are `'IR'` (plots $\log_{10}$ of the product of the *DataX* and *DataY* intensities versus $\log_2$ of the intensity ratios) or `'MA'` (plots $(1/2)\log_2$ of the product of the *DataX* and *DataY* intensities versus $\log_2$ of the intensity ratios). Default is `'IR'`.
LogTransValue	Controls the conversion of data in *X* and *Y* from natural scale to $\log_2$ scale. Set *LogTransValue* to `false`, when the data is already $\log_2$ scale. Default is `true`, which assumes the data is natural scale.
FactorLinesValue	Adds lines to the plot showing a factor of *N* change. Default is 2, which corresponds to a level of 1 and -1 on a $\log_2$ scale. **Tip** You can also change the factor lines interactively, after creating the plot.
TitleValue	Character vector or string that specifies a title for the plot.
LabelsValue	Cell array of character vectors or string vector containing labels for the data. If labels are defined, then clicking a point on the plot shows the label corresponding to that point.

NormalizeValue	Controls the display of lowess normalized ratio values. Enter `true` to display to lowess normalized ratio values. Default is `false`. **Tip** You can also normalize the data from the MAIR Plot window, after creating the plot.
LowessOptionsValue	Cell array of one, two, or three property name/value pairs in any order that affect the lowess normalization. Choices for property name/value pairs are: • `'Order'`, *`OrderValue`* • `'Robust'`, *`RobustValue`* • `'Span'`, *`SpanValue`* For more information on the preceding property name/value pairs, see `malowess`.
ShowplotValue	Controls the display of the scatter plot. Choices are `true` (default) or `false`.
PlotOnlyValue	Controls the display of the scatter plot without user interface components. Choices are `true` or `false` (default). **Note** If you set the `'PlotOnly'` property to `true`, you can still display labels for data points by clicking a data point, and you can still adjust the horizontal fold change lines by click-dragging the lines.

Output Arguments

Intensity	DataMatrix object on page 1-734 or vector containing intensity values for the microarray gene expression data, calculated as: • $\log_{10}$ of the product of the *`DataX`* and *`DataY`* intensities (when `Type` is `'IR'`) • (1/2)$\log_2$ of the product of the *`DataX`* and *`DataY`* intensities (when `Type` is `'MA'`) **Note** If *`DataX`* or *`DataY`* is a DataMatrix object, then *`Intensity`* is also a DataMatrix object with the same properties.
Ratio	DataMatrix object on page 1-734 or vector containing ratios of the microarray gene expression data, calculated as `log2(`*`DataX./ DataY`*`)`. **Note** If *`DataX`* or *`DataY`* is a DataMatrix object, then *`Ratio`* is also a DataMatrix object with the same properties.
H	Handle of the plot.

Description

`mairplot(`*`DataX`*`, `*`DataY`*`)` creates a scatter plot that plots $\log_{10}$ of the product of the *`DataX`* and *`DataY`* intensities versus $\log_2$ of the intensity ratios.

`[`*`Intensity`*`, `*`Ratio`*`] = mairplot(`*`DataX`*`, `*`DataY`*`)` returns the intensity and ratio values. If you set `'Normalize'` to `true`, the returned ratio values are normalized.

`[`*`Intensity`*`, `*`Ratio`*`, `*`H`*`] = mairplot(`*`DataX`*`, `*`DataY`*`)` returns the handle of the plot.

`...`` = mairplot(..., '`*`PropertyName`*`', `*`PropertyValue`*`, ...)` calls `mairplot` with optional properties that use property name/property value pairs. You can specify one or more properties in any order. Each *`PropertyName`* must be enclosed in single quotation marks and is case insensitive. These property name/property value pairs are as follows:

`...`` = mairplot(..., 'Type', `*`TypeValue`*`, ...)` specifies the plot type. Choices are `'IR'` (plots $\log_{10}$ of the product of the *`DataX`* and *`DataY`* intensities versus $\log_2$ of the intensity ratios) or `'MA'` (plots $(1/2)\log_2$ of the product of the *`DataX`* and *`DataY`* intensities versus $\log_2$ of the intensity ratios). Default is `'IR'`.

`...`` = mairplot(..., 'LogTrans', `*`LogTransValue`*`, ...)` controls the conversion of data in *`X`* and *`Y`* from natural to $\log_2$ scale. Set *`LogTransValue`* to `false`, when the data is already $\log_2$ scale. Default is `true`, which assumes the data is natural scale.

`...`` = mairplot(..., 'FactorLines', `*`FactorLinesValue`*`, ...)` adds lines to the plot showing a factor of *`N`* change. Default is 2, which corresponds to a level of 1 and -1 on a $\log_2$ scale.

Tip You can also change the factor lines interactively, after creating the plot.

`...`` = mairplot(..., 'Title', `*`TitleValue`*`, ...)` specifies a title for the plot.

`...`` = mairplot(..., 'Labels', `*`LabelsValue`*`, ...)` specifies a cell array of character vectors or string vector of labels for the data. If labels are defined, then clicking a point on the plot shows the label corresponding to that point.

`...`` = mairplot(..., 'Normalize', `*`NormalizeValue`*`, ...)` controls the display of lowess normalized ratio values. Enter `true` to display to lowess normalized ratio values. Default is `false`.

Tip You can also normalize the data from the MAIR Plot window, after creating the plot.

`...`` = mairplot(..., 'LowessOptions', `*`LowessOptionsValue`*`, ...)` lets you specify up to three property name/value pairs (in any order) that affect the lowess normalization. Choices for property name/value pairs are:

- `'Order', `*`OrderValue`*
- `'Robust', `*`RobustValue`*
- `'Span', `*`SpanValue`*

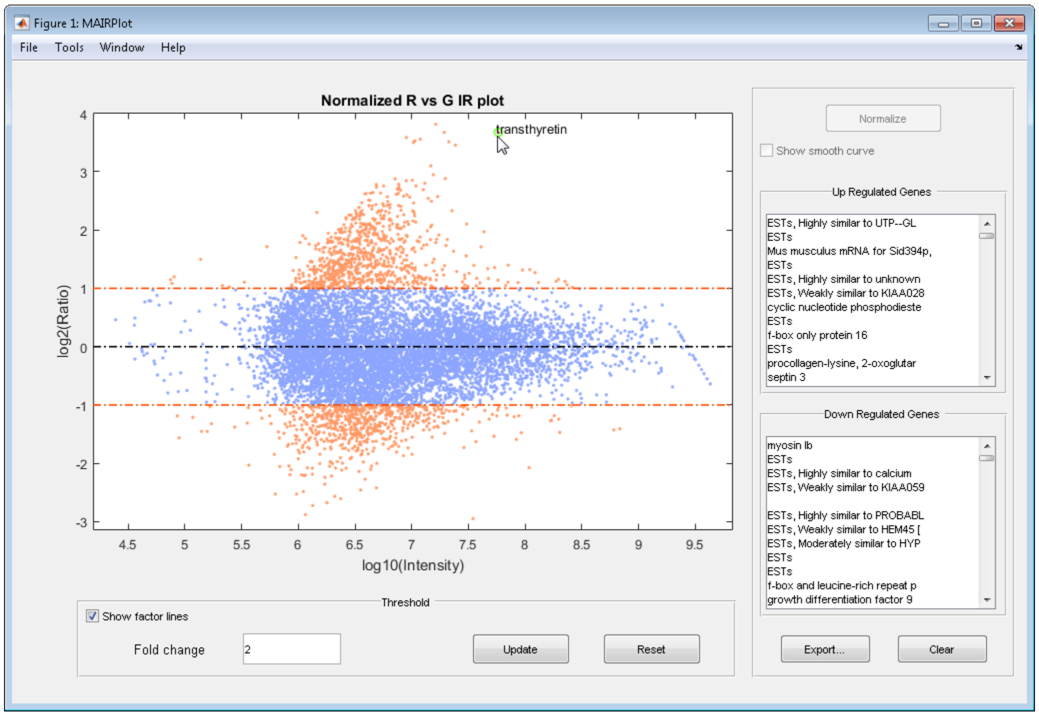
For more information on the previous three property name/value pairs, see the `malowess` function.

`... = mairplot(..., 'Showplot', ShowplotValue, ...)` controls the display of the scatter plot. Choices are `true` (default) or `false`.

`... = mairplot(..., 'PlotOnly', PlotOnlyValue, ...)` controls the display of the scatter plot without user interface components. Choices are `true` or `false` (default).

Note If you set the `'PlotOnly'` property to `true`, you can still display labels for data points by clicking a data point, and you can still adjust the horizontal fold change lines by click-dragging the lines.

Following is an IR plot of normalized data.



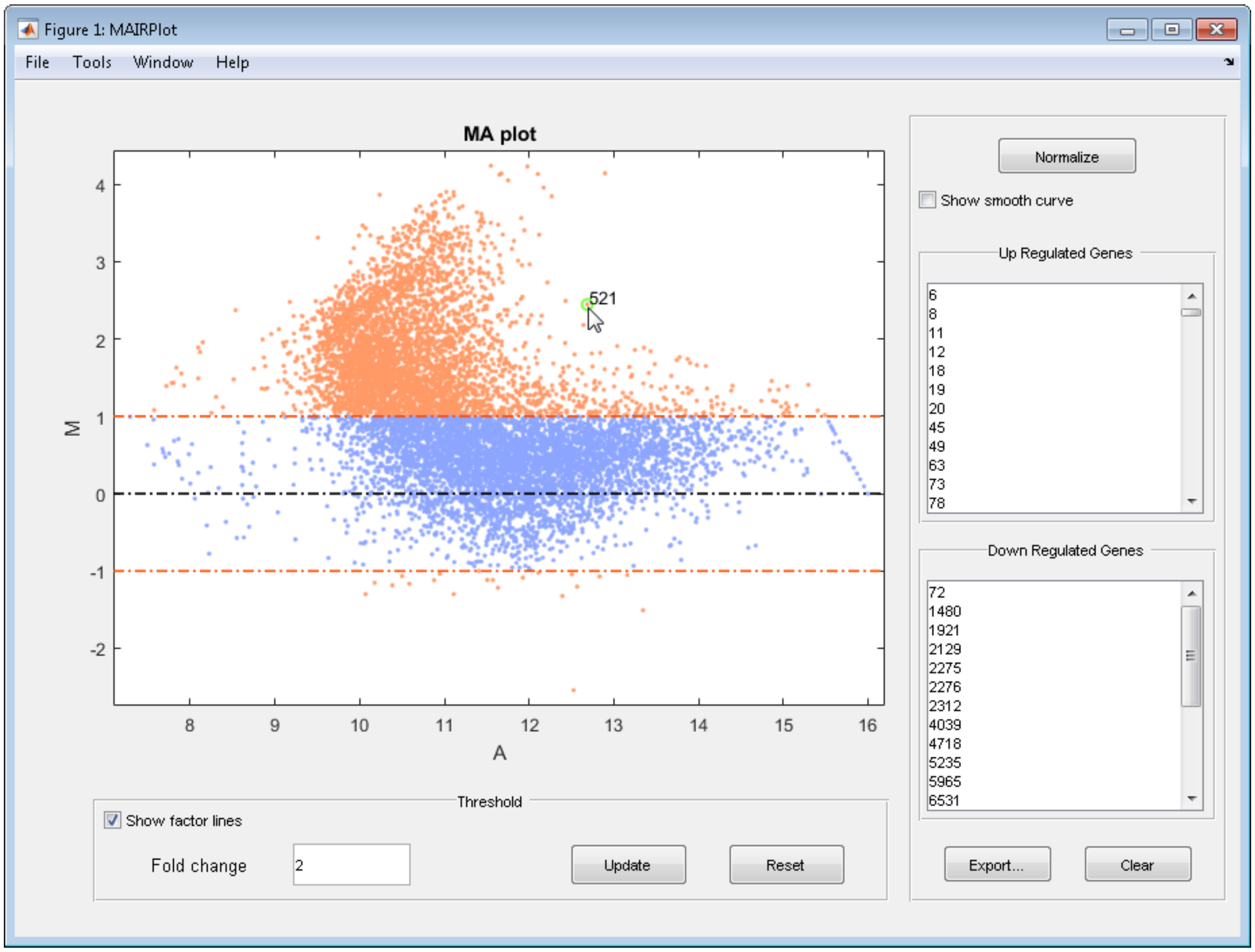
Up Regulated Genes:
ESTs, Highly similar to UTP--GL
ESTs
Mus musculus mRNA for Sid394p,
ESTs
ESTs, Highly similar to unknown
ESTs, Weakly similar to KIAA028
cyclic nucleotide phosphodieste
ESTs
f-box only protein 16
ESTs
procollagen-lysine, 2-oxoglutar
septin 3

Down Regulated Genes:
myosin Ib
ESTs
ESTs, Highly similar to calcium
ESTs, Weakly similar to KIAA059

ESTs, Highly similar to PROBABL
ESTs, Weakly similar to HEM45 [
ESTs, Moderately similar to HYP
ESTs
ESTs
f-box and leucine-rich repeat p
growth differentiation factor 9

[✓] Show factor lines
Fold change 2

Following is an MA plot of unnormalized data.



The intensity versus ratio scatter plot displays the following:

- $\log_{10}$ (Intensity) versus $\log_2$ (Ratio) scatter plot of genes.
- Two horizontal fold change lines at a fold change level of 2, which corresponds to a ratio of 1 and –1 on a $\log_2$ (Ratio) scale. (Lines will be at different fold change levels, if you used the `'FactorLines'` property.)
- Data points for genes that are considered differentially expressed (outside of the fold change lines) appear in orange.

After you display the intensity versus ratio scatter plot, you can interactively do the following:

- Adjust the horizontal fold change lines by click-dragging one line or entering a value in the **Fold Change** text box, then clicking **Update**.
- Display labels for data points by clicking a data point.
- Select a gene from the **Up Regulated** or **Down Regulated** list to highlight the corresponding data point in the plot. Press and hold **Ctrl** or **Shift** to select multiple genes.
- Zoom the plot by selecting **Tools > Zoom In** or **Tools > Zoom Out**.
- View lists of significantly up-regulated and down-regulated genes, and optionally, export the gene labels and indices to a structure in the MATLAB Workspace by clicking **Export**.

- Normalize the data by clicking the **Normalize** button, then selecting whether to show the normalized plot in a separate window. If you show the normalized plot in a separate window, the **Show smooth curve** check box becomes available in the original (unnormalized) plot.

Tip To select different lowess normalization options before normalizing, select **Tools > Set LOWESS Normalization Options**, then enter options in the Options for LOWESS dialog box.

Examples

1. Use the gprread function to create a structure containing microarray data.

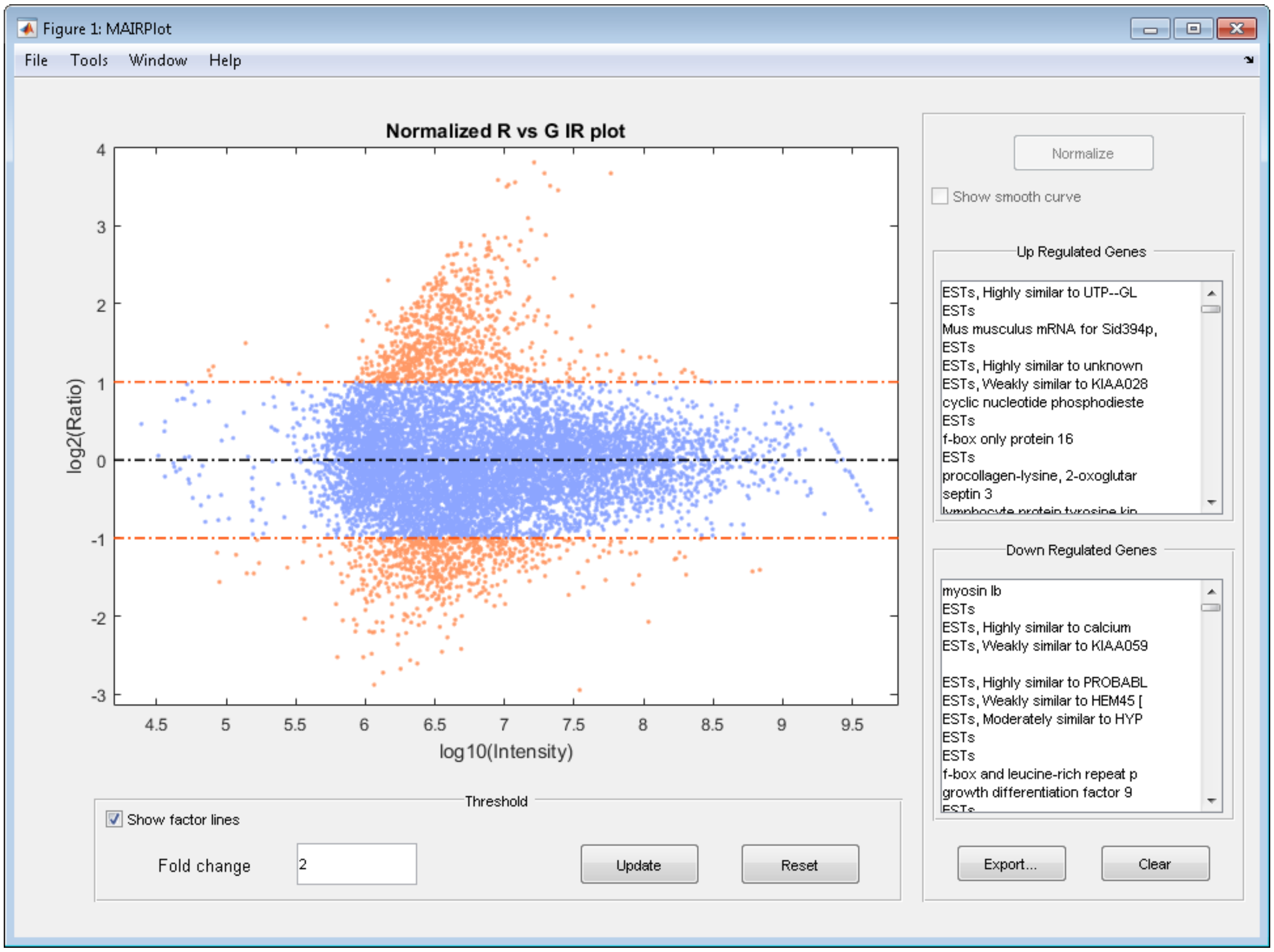
```
maStruct = gprread('mouse_a1wt.gpr');
```

2. Use the magetfield function to extract the green (cy3) and red (cy5) signals from the structure.

```
cy5data = magetfield(maStruct,'F635 Median');
cy3data = magetfield(maStruct,'F532 Median');
```

3. Create an intensity versus ratio scatter plot of the cy3 and cy5 data. Normalize the data and add a title and labels:

```
mairplot(cy5data, cy3data, 'Normalize', true, ...
                'Title','Normalized R vs G IR plot', ...
                'Labels', maStruct.Names)
```



Fold change: 2

Up Regulated Genes:
ESTs, Highly similar to UTP--GL
ESTs
Mus musculus mRNA for Sid394p,
ESTs
ESTs, Highly similar to unknown
ESTs, Weakly similar to KIAA028
cyclic nucleotide phosphodieste
ESTs
f-box only protein 16
ESTs
procollagen-lysine, 2-oxoglutar
septin 3
lymphocyte protein tyrosine kin

Down Regulated Genes:
myosin Ib
ESTs
ESTs, Highly similar to calcium
ESTs, Weakly similar to KIAA059

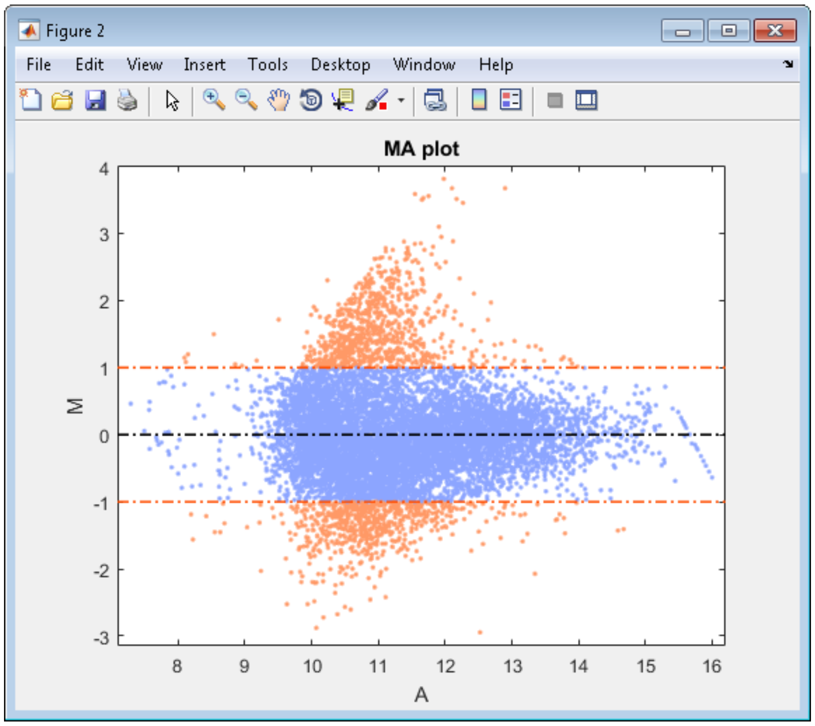
ESTs, Highly similar to PROBABL
ESTs, Weakly similar to HEM45 [
ESTs, Moderately similar to HYP
ESTs
ESTs
f-box and leucine-rich repeat p
growth differentiation factor 9
ESTs

4 Return intensity values and ratios without displaying the plot.

```
[intensities, ratios] = mairplot(cy5data, cy3data, 'Showplot', false);
```

5 Create a normalized MA plot of the cy3 and cy5 data without the user interface components.

```
mairplot(cy5data, cy3data, 'Normalize', true, ...
                'Type','MA','PlotOnly',true)
```



Version History

Introduced before R2006a

References

[1] Quackenbush, J. (2002). Microarray Data Normalization and Transformation. Nature Genetics *Suppl. 32*, 496–501.

[2] Dudoit, S., Yang, Y.H., Callow, M.J., and Speed, T.P. (2002). Statistical Methods for Identifying Differentially Expressed Genes in Replicated cDNA Microarray Experiments. Statistica Sinica *12*, 111–139.

See Also

`maboxplot` | `magetfield` | `maimage` | `mainvarsetnorm` | `maloglog` | `malowess` | `manorm` | `mattest` | `mavolcanoplot`

maloglog

Create loglog plot of microarray data

Syntax

```
maloglog(X, Y)

maloglog(X, Y, ...'FactorLines', N, ...)
maloglog(X, Y, ...'Title', TitleValue, ...)
maloglog(X, Y, ...'Labels', LabelsValues, ...)
maloglog(X, Y, ...'HandleGraphicsName', HGValue, ...)
H = maloglog(...)
```

Arguments

X, *Y*	DataMatrix object on page 1-734 or numeric array of microarray expression values from a single experimental condition.
N	Property to add two lines to the plot showing a factor of *N* change.
`TitleValue`	A character vector or string to use as the title for the plot.
`LabelsValue`	A cell array of character vectors or string vector containing labels for the data in *X* and *Y*. If you specify `LabelsValue`, then clicking a data point in the plot shows the label corresponding to that point.

Description

`maloglog(X, Y)` creates a loglog scatter plot of *X* versus *Y*. *X* and *Y* are DataMatrix object on page 1-734s or numeric arrays of microarray expression values from two different experimental conditions.

`maloglog(X, Y, ...'PropertyName', PropertyValue, ...)` calls `maloglog` with optional properties that use property name/property value pairs. You can specify one or more properties in any order. Each `PropertyName` must be enclosed in single quotation marks and is case insensitive. These property name/property value pairs are as follows:

`maloglog(X, Y, ...'FactorLines', N, ...)` adds two lines to the plot showing a factor of *N* change.

`maloglog(X, Y, ...'Title', TitleValue, ...)` allows you to specify a title for the plot.

`maloglog(X, Y, ...'Labels', LabelsValues, ...)` allows you to specify a cell array of character vectors or string vector containing labels for the data. If `LabelsValues` is defined, then clicking a data point in the plot shows the label corresponding to that point.

`maloglog(X, Y, ...'HandleGraphicsName', HGValue, ...)` allows you to pass optional Handle Graphics property name/property value pairs to the function.

`H = maloglog(...)` returns the handle to the plot.

Examples

```
maStruct = gprread('mouse_a1wt.gpr');
Red = magetfield(maStruct,'F635 Median');
Green = magetfield(maStruct,'F532 Median');
maloglog(Red,Green,'title','Red vs Green');
% Add factorlines and labels
figure
maloglog(Red,Green,'title','Red vs Green',...
                'FactorLines',2,'LABELS',maStruct.Names);
% Now create a normalized plot
figure
maloglog(manorm(Red),manorm(Green),'title',...
                'Normalized Red vs Green','FactorLines',2,...
                'LABELS',maStruct.Names);
```

Version History

Introduced before R2006a

See Also

`maboxplot` | `magetfield` | `mainvarsetnorm` | `maimage` | `mairplot` | `malowess` | `manorm` | `mattest` | `mavolcanoplot` | `loglog`

malowess

Smooth microarray data using Lowess method

Syntax

```
YSmooth = malowess(X, Y)

YSmooth = malowess(X, Y, ...'Order', OrderValue, ...)
YSmooth = malowess(X, Y, ...'Robust', RobustValue, ...)
YSmooth = malowess(X, Y, ...'Span', SpanValue, ...)
```

Arguments

X, *Y*	DataMatrix object on page 1-734 or numeric vector containing scatter data.
OrderValue	Property to select the order of the algorithm. Enter either `1` (linear fit) or `2` (quadratic fit). The default order is `1`.
RobustValue	Property to select a robust fit. Enter either `true` or `false`.
SpanValue	Property to specify the window size. The default value is `0.05` (5% of total points in *X*)

Description

YSmooth = `malowess(`*X*, *Y*`)` smooths scatter data in *X* and *Y* using the Lowess smoothing method. The default window size is 5% of the length of *X*. *YSmooth* is a numeric vector or, if *Y* is a DataMatrix object, also a DataMatrix object with the same properties as *Y*.

YSmooth = `malowess(`*X*, *Y*, `...'`*PropertyName*`',` *PropertyValue*`, ...)` calls `malowess` with optional properties that use property name/property value pairs. You can specify one or more properties in any order. Each *PropertyName* must be enclosed in single quotation marks and is case insensitive. These property name/property value pairs are as follows:

YSmooth = `malowess(`*X*, *Y*, `...'Order',` *OrderValue*`, ...)` chooses the order of the algorithm. Note that the Curve Fitting Toolbox™ software refers to Lowess smoothing of order `2` as Loess smoothing.

YSmooth = `malowess(`*X*, *Y*, `...'Robust',` *RobustValue*`, ...)` uses a robust fit when *RobustValue* is set to `true`. This option can take a long time to calculate.

YSmooth = `malowess(`*X*, *Y*, `...'Span',` *SpanValue*`, ...)` modifies the window size for the smoothing function. If *SpanValue* is less than `1`, the window size is taken to be a fraction of the number of points in the data. If *SpanValue* is greater than `1,` the window is of size *SpanValue*.

Examples

```
maStruct = gprread('mouse_a1wt.gpr');
cy5data = magetfield(maStruct, 'F635 Median');
```

```
cy3data = magetfield(maStruct, 'F532 Median');
[x,y] = mairplot(cy5data, cy3data);
drawnow
ysmooth = malowess(x,y);
hold on;
plot(x, ysmooth, 'rx')
ynorm = y - ysmooth;
```

Version History

Introduced before R2006a

See Also

affyinvarsetnorm | maboxplot | magetfield | maimage | mainvarsetnorm | mairplot | maloglog | manorm | quantilenorm | robustfit

manorm

Normalize microarray data

Syntax

```
XNorm = manorm(X)
XNorm = manorm(MAStruct, FieldName)
[XNorm, ColVal] = manorm(...)

manorm(..., 'Method', MethodValue, ...)
manorm(..., 'Extra_Args', Extra_ArgsValue, ...)
manorm(..., 'LogData', LogDataValue, ...)
manorm(..., 'Percentile', PercentileValue, ...)
manorm(..., 'Global', GlobalValue, ...)
manorm(..., 'StructureOutput', StructureOutputValue, ...)
manorm(..., 'NewColumnName', NewColumnNameValue, ...)
```

Arguments

X	Numeric array or DataMatrix object on page 1-734 of microarray data.
MAStruct	Microarray structure.
FieldName	Field.

Description

`XNorm = manorm(X)` scales the values in each column of *X*, a numeric array or DataMatrix object on page 1-734 of microarray data, by dividing by the mean column intensity. *XNorm* is a vector, matrix, or DataMatrix object on page 1-734 of normalized microarray data.

`XNorm = manorm(MAStruct, FieldName)` scales the data in *MAStruct*, a microarray structure, for a field specified by *FieldName*, for each block or print-tip by dividing each block by the mean column intensity. The output is a matrix with each column corresponding to the normalized data for each block.

`[XNorm, ColVal] = manorm(...)` returns the values used to normalize the data.

`manorm(..., 'PropertyName', PropertyValue, ...)` calls `manorm` with optional properties that use property name/property value pairs. You can specify one or more properties in any order. Each *PropertyName* must be enclosed in single quotation marks and is case insensitive. These property name/property value pairs are as follows:

`manorm(..., 'Method', MethodValue, ...)` allows you to choose the method for scaling or centering the data. *MethodValue* can be `'Mean'`(default), `'Median'`, `'STD'` (standard deviation), `'MAD'` (median absolute deviation), or a function handle. If you pass a function handle, then the function should ignore NaNs and must return a single value per column of the input data.

`manorm(..., 'Extra_Args', Extra_ArgsValue, ...)` allows you to pass extra arguments to the function *MethodValue*. *Extra_ArgsValue* must be a cell array.

`manorm(..., 'LogData', LogDataValue, ...)`, when *`LogDataValue`* is `true`, works with log ratio data in which case the mean (or *`MethodValue`*) of each column is subtracted from the values in the columns, instead of dividing the column by the normalizing value.

`manorm(..., 'Percentile', PercentileValue, ...)` only uses the percentile (*`PercentileValue`*) of the data preventing large outliers from skewing the normalization. If *`PercentileValue`* is a vector containing two values, then the range from the *`PercentileValue(1)`* percentile to the *`PercentileValue(2)`* percentile is used. The default value is `100`, that is to use all the data in the data set.

`manorm(..., 'Global', GlobalValue, ...)` when *`GlobalValue`* is `true`, normalizes the values in the data set by the global mean (or *`MethodValue`*) of the data, as opposed to normalizing each column or block of the data independently.

`manorm(..., 'StructureOutput', StructureOutputValue, ...)`, when *`StructureOutputValue`* is `true`, the input data is a structure returns the input structure with an additional data field for the normalized data.

`manorm(..., 'NewColumnName', NewColumnNameValue, ...)`, when using `StructureOutput`, allows you to specify the name of the column that is appended to the list of `ColumnNames` in the structure. The default behavior is to prefix `'Block Normalized'` to `FieldName`.

Examples

```
maStruct = gprread('mouse_a1wt.gpr');
% Extract some data of interest.
Red = magetfield(maStruct,'F635 Median');
Green = magetfield(maStruct,'F532 Median');
% Create a log-log plot.
maloglog(Red,Green,'factorlines',true)
% Center the data.
normRed = manorm(Red);
normGreen = manorm(Green);
% Create a log-log plot of the centered data.
figure
maloglog(normRed,normGreen,'title','Normalized','factorlines',true)

% Alternatively, you can work directly with the structure
normRedBs = manorm(maStruct,'F635 Median - B635');
normGreenBs = manorm(maStruct,'F532 Median - B532');
% Create a log-log plot of the centered data. This includes some
% zero values so turn off the warning.
figure
w = warning('off','Bioinfo:maloglog:ZeroValues');
warning('off','Bioinfo:maloglog:NegativeValues');
maloglog(normRedBs,normGreenBs,'title',...
                'Normalized Background-Subtracted Median Values',...
                'factorlines',true)
        warning(w);
```

Version History

Introduced before R2006a

See Also

`affyinvarsetnorm` | `maboxplot` | `magetfield` | `mainvarsetnorm` | `mairplot` | `maloglog` | `malowess` | `quantilenorm` | `rmasummary`

mapcaplot

Create Principal Component Analysis (PCA) plot of microarray data

Syntax

```
mapcaplot(data)
mapcaplot(data,labels)
```

Description

`mapcaplot(data)` creates 2-D scatter plots of principal components of `data`. Once you plot the principal components, you can:

- Select principal components for the *x* and *y* axes from the drop-down list below each scatter plot.
- Click a data point to display its label.
- Select a subset of data points by dragging a box around them. Points in the selected region and the corresponding points in the other axes are then highlighted. The labels of the selected data points appear in the list box.
- Select a label in the list box to highlight the corresponding data point in the plot. Press and hold **Ctrl** or **Shift** to select multiple data points.
- Export the gene labels and indices to the MATLAB workspace.

`mapcaplot(data,labels)` labels the data points in the PCA plots using `labels`, instead of the row numbers.

Examples

Create PCA Plot of Microarray Data

Create a PCA plot to visualize genes involved during the metabolic shift from fermentation to respiration of yeast (*Saccharomyces cerevisiae*).

Load the data file that contains filtered yeast microarray data. The data comes from an experiment (DeRisi et al., 1997) that used DNA microarrays to study temporal gene expression of these genes. Expression levels were measured at seven time points during the diauxic shift.

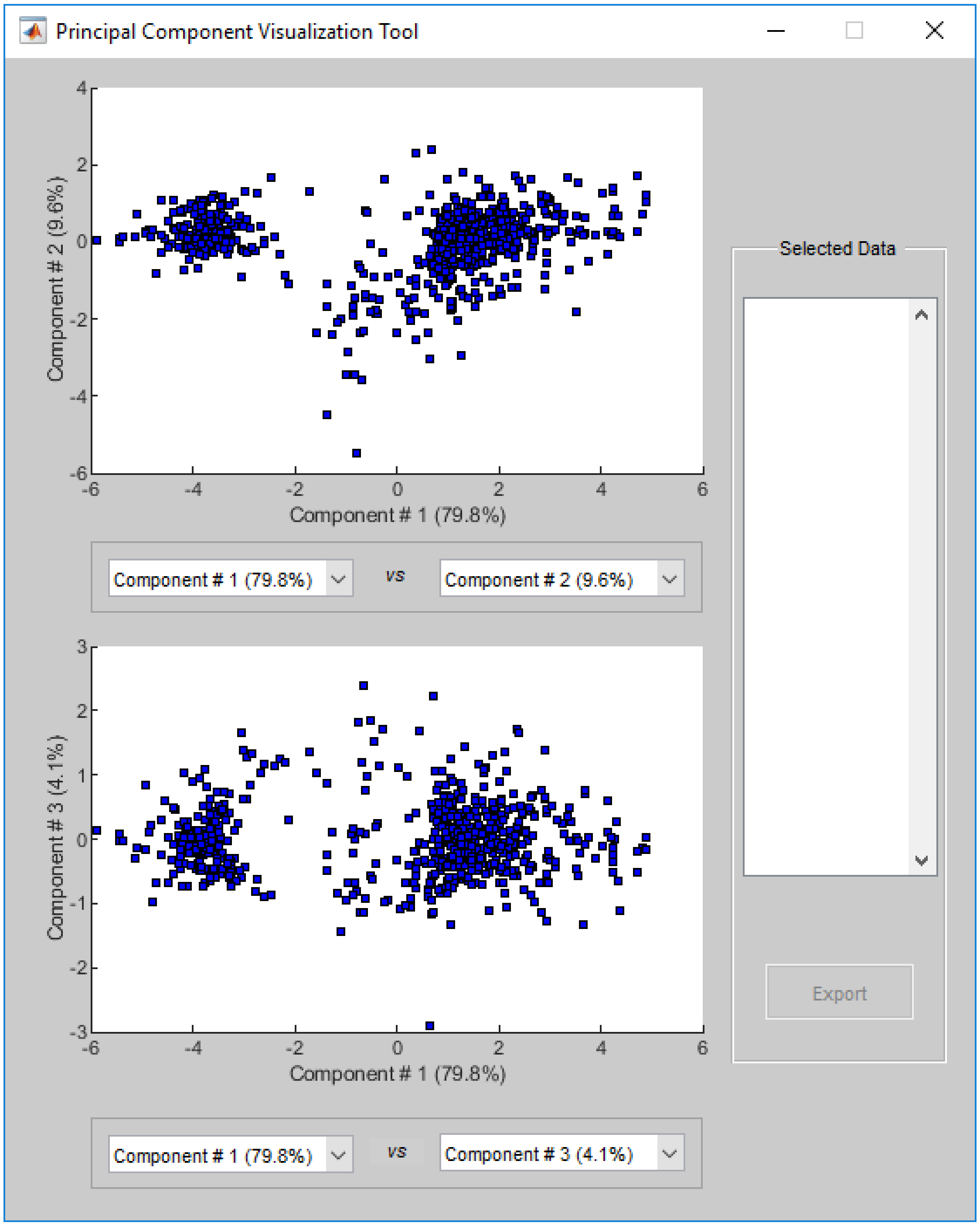
```
load filteredyeastdata
```

This MAT-file includes three variables:

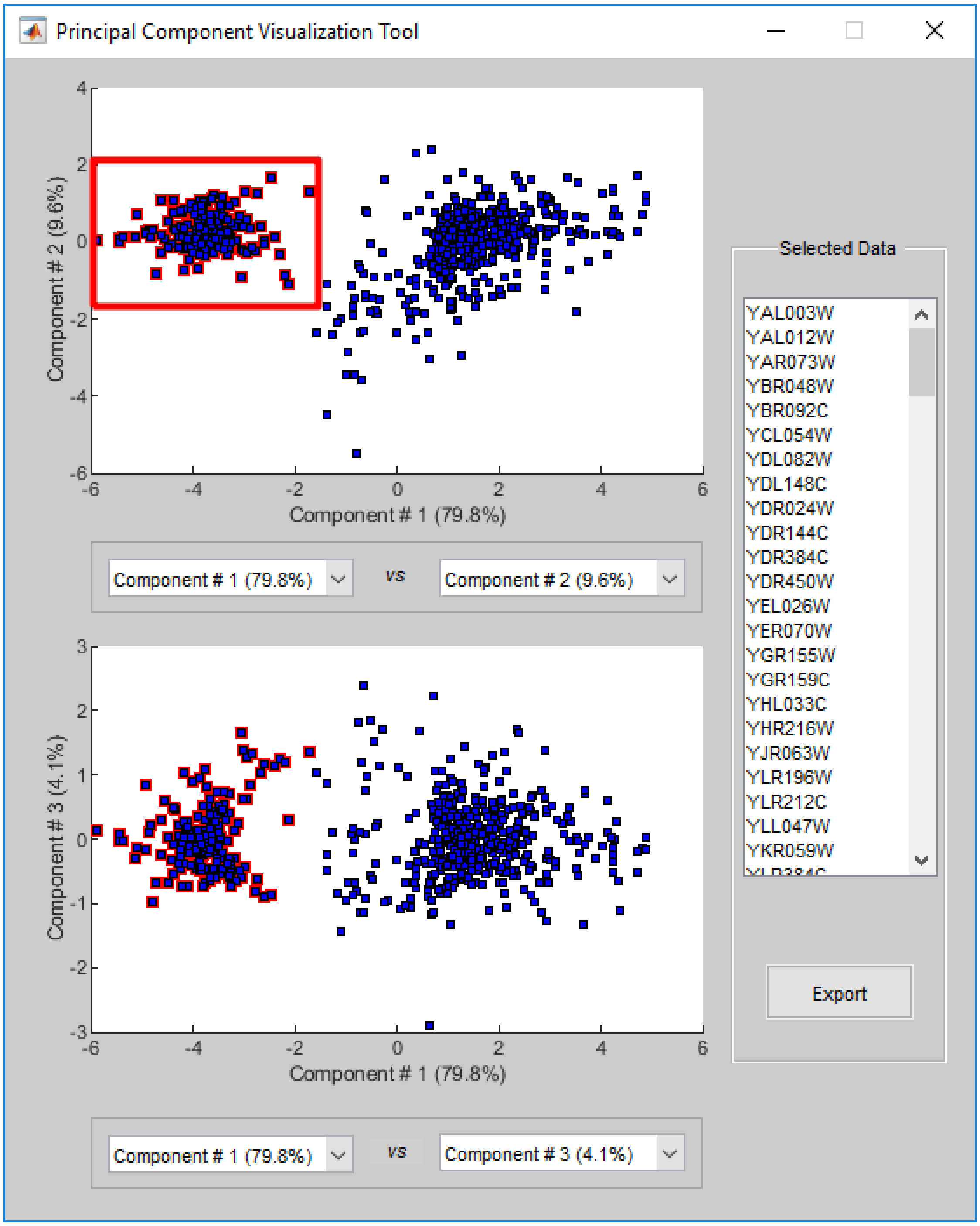
- *yeastvalues* — A matrix of gene expression data from Saccharomyces cerevisiae (yeast) during the metabolic shift from fermentation to respiration
- *genes* — A cell array of GenBank® accession numbers for labeling the rows in yeastvalues
- *times* — A vector of time values for labeling the columns in yeastvalues

Perform PCA on the expression data and plot the result.

```
mapcaplot(yeastvalues, genes)
```



Select a subset of data points by dragging a box around them. The data points are highlighted and their corresponding labels appear in **Selected Data**. You can then export the selected data to the workspace by selecting **Export**.



Input Arguments

data — Microarray expression profile data
numeric array | `DataMatrix` object

Microarray expression profile data, specified as a numeric array or `DataMatrix object`.

Data Types: `double`

`labels` — Data point labels
cell array of character vectors | string vector

Data point labels, specified as a cell array of character vectors or string vector.

Data Types: `string` | `cell`

Version History
Introduced before R2006a

References

[1] DeRisi, J.L., Iyer, V.R., and Brown, P.O. (1997). Exploring the metabolic and genetic control of gene expression on a genomic scale. Science 278, 680–686s.

See Also
`clustergram` | `mattest` | `mavolcanoplot` | `pca`

mattest

Perform two-sample t-test to evaluate differential expression of genes from two experimental conditions or phenotypes

Syntax

```
PValues = mattest(DataX, DataY)
[PValues, TScores] = mattest(DataX, DataY)
[PValues, TScores, DFs] = mattest(DataX, DataY)

... = mattest(..., 'VarType', VarTypeValue, ...)
... = mattest(..., 'Permute', PermuteValue, ...)
... = mattest(..., 'Bootstrap', BootstrapValue, ...)
... = mattest(..., 'Showhist', ShowhistValue, ...)
... = mattest(..., 'Showplot', ShowplotValue, ...)
... = mattest(..., 'Labels', LabelsValue, ...)
```

Input Arguments

`DataX`, *`DataY`*	DataMatrix object on page 1-734 or a matrix of gene expression values where each row corresponds to a gene and each column corresponds to a replicate. *`DataX`* and *`DataY`* must have the same number of rows and are assumed to be normally distributed in each class with equal variances. *`DataX`* contains data from one experimental condition and *`DataY`* contains data from a different experimental condition. For example, *`DataX`* could be expression values from cancer cells, and *`DataY`* could be expression values from normal cells.
`VarTypeValue`	Character vector that specifies the variance type of the test. *`VarTypeValue`* can be `'equal'` or `'unequal'` (default). If set to `'equal'`, `mattest` performs the test assuming the two samples have equal variances. If set to `'unequal'`, `mattest` performs the test assuming the two samples have unknown and unequal variances.
`PermuteValue`	Controls whether permutation tests are run, and if so, how many. Choices are `true`, `false` (default), or any integer greater than `2`. If set to `true`, the number of permutations is `1000`.
`BootstrapValue`	Controls whether bootstrap tests are run, and if so, how many. Choices are `true`, `false` (default), or any integer greater than `2`. If set to `true`, the number of bootstrap tests is `1000`.
`ShowhistValue`	Controls the display of histograms of t-score distributions and p-value distributions. Choices are `true` or `false` (default).
`ShowplotValue`	Controls the display of a normal t-score quantile plot. Choices are `true` or `false` (default). In the t-score quantile plot, data points with t-scores `> (1 - 1/(2N))` or `< 1/(2N)` display with red circles. `N` is the total number of genes.

LabelsValue	Cell array of character vectors or string vector containing labels (typically gene names or probe set IDs) for each row in *DataX* and *DataY*. The labels display if you click a data point in the t-score quantile plot.

Output Arguments

PValues	One of the following: • Column vector of p-values for each gene in *DataX* and *DataY* (if both inputs are matrices). • DataMatrix object on page 1-734 with row names the same as the first input DataMatrix object and a column name of `p-values` (if at least one input is a DataMatrix object).
TScores	Column vector of t-scores for each gene in *DataX* and *DataY*.
DFs	Column vector containing the degree of freedom for each gene in *DataX* and *DataY*.

Description

`PValues = mattest(DataX, DataY)` performs an unpaired t-test for differential expression with a standard two-tailed and two-sample t-test on every gene in *DataX* and *DataY* and returns a p-value for each gene. *DataX* and *DataY* are either a DataMatrix object on page 1-734 or a matrix of gene expression values, in which each row corresponds to a gene, and each column corresponds to a replicate. *DataX* contains data from one experimental condition and *DataY* contains data from another experimental condition. *DataX* and *DataY* must have the same number of rows and are assumed to be normally distributed in each class. *PValues* is a column vector of p-values for each gene, or, if at least one of the inputs is a DataMatrix object, a DataMatrix object with row names the same as the first input DataMatrix object and a column name of `p-values`.

`[PValues, TScores] = mattest(DataX, DataY)` also returns a t-score for each gene in *DataX* and *DataY*. *TScores* is a column vector of t-scores for each gene.

`[PValues, TScores, DFs] = mattest(DataX, DataY)` also returns *DFs*, a column vector containing the degree of freedom for each gene across both data sets, *DataX* and *DataY*.

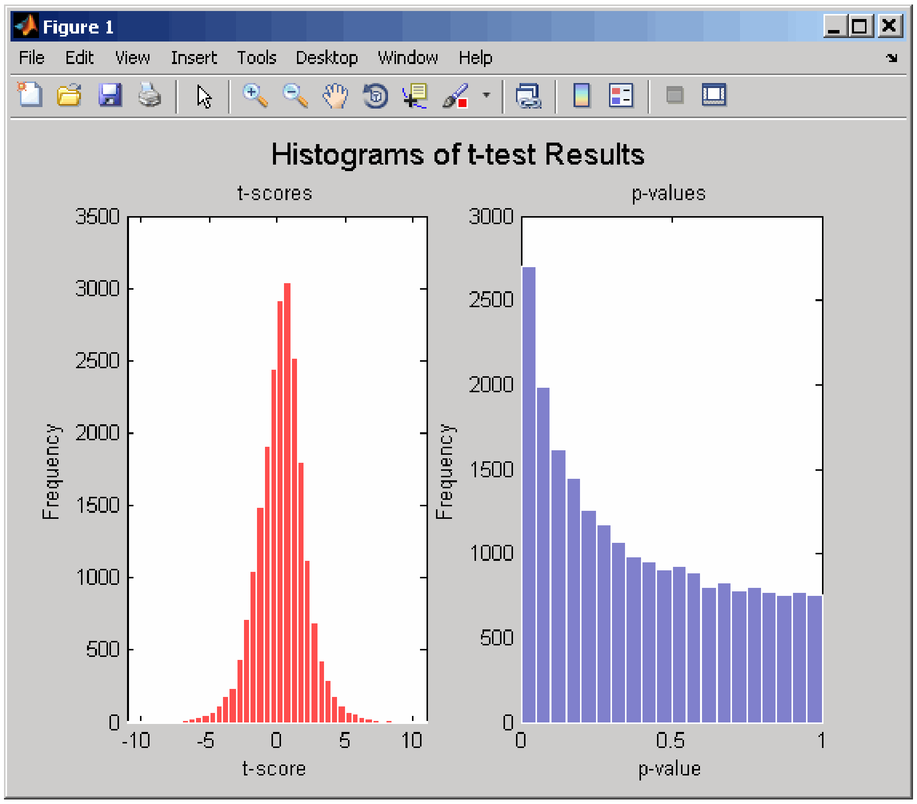
`... = mattest(..., 'PropertyName', PropertyValue, ...)` calls `mattest` with optional properties that use property name/property value pairs. You can specify one or more properties in any order. Each *PropertyName* must be enclosed in single quotation marks and is case insensitive. These property name/property value pairs are as follows:

`... = mattest(..., 'VarType', VarTypeValue, ...)` specifies the variance type of the test. *VarTypeValue* can be `'equal'` or `'unequal'` (default). If set to `'equal'`, `mattest` performs the test assuming the two samples have equal variances. If set to `'unequal'`, `mattest` performs the test assuming the two samples have unknown and unequal variances.

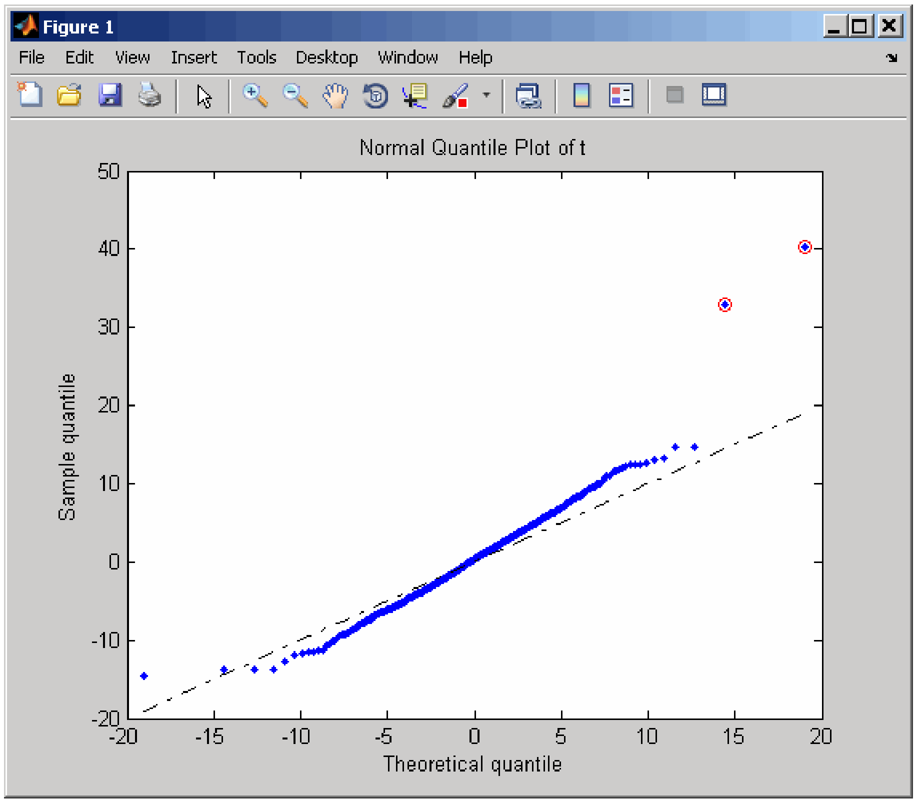
`... = mattest(..., 'Permute', PermuteValue, ...)` controls whether permutation tests are run, and if so, how many. *PermuteValue* can be `true`, `false` (default), or any integer greater than `2`. If set to `true`, the number of permutations is `1000`.

`... = mattest(..., 'Bootstrap', `*`BootstrapValue`*`, ...)` controls whether bootstrap tests are run, and if so, how many. *`BootstrapValue`* can be `true`, `false` (default), or any integer greater than 2. If set to `true`, the number of bootstrap tests is `1000`.

`... = mattest(..., 'Showhist', `*`ShowhistValue`*`, ...)` controls the display of histograms of t-score distributions and p-value distributions. When *`ShowhistValue`* is `true`, `mattest` displays histograms. Default is `false`.



`... = mattest(..., 'Showplot', `*`ShowplotValue`*`, ...)` controls the display of a normal t-score quantile plot. When *`ShowplotValue`* is `true`, `mattest` displays a quantile-quantile plot. Default is `false`. In the t-score quantile plot, the black diagonal line represents the sample quantile being equal to the theoretical quantile. Data points of genes considered to be differentially expressed lie farther away from this line. Specifically, data points with t-scores > `(1 - 1/(2N))` or < `1/(2N)` display with red circles. `N` is the total number of genes.



`... = mattest(..., 'Labels', LabelsValue, ...)` controls the display of labels when you click a data point in the t-score quantile plot. *`LabelsValue`* is a cell array of character vectors or string vector containing labels (typically gene names or probe set IDs) for each row in *`DataX`* and *`DataY`*.

Examples

1 Load the MAT-file, included with the Bioinformatics Toolbox software, that contains Affymetrix data from a prostate cancer study, specifically probe intensity data from Affymetrix HG-U133A GeneChip arrays. The two variables in the MAT-file, `dependentData` and `independentData`, are two matrices of gene expression values from two experimental conditions.

```
load prostatecancerexpdata
```

2 Calculate the p-values and t-scores for the gene expression values in the two matrices and display a normal t-score quantile plot.

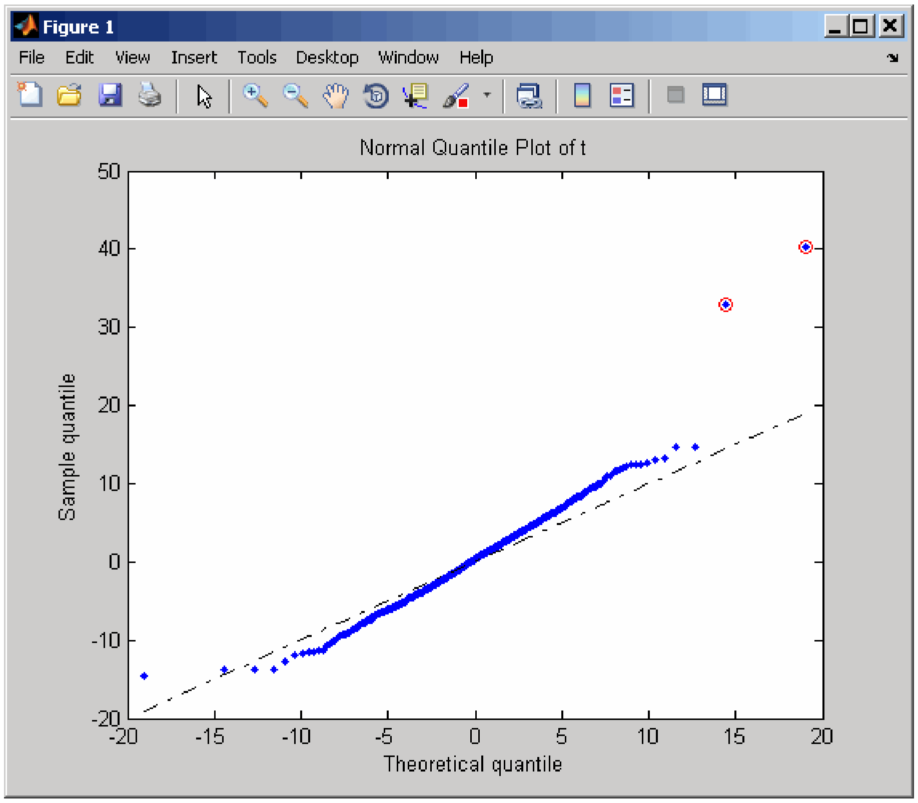
```
[pvalues,tscores] = mattest(dependentData, independentData,...
                    'showplot',true);
```

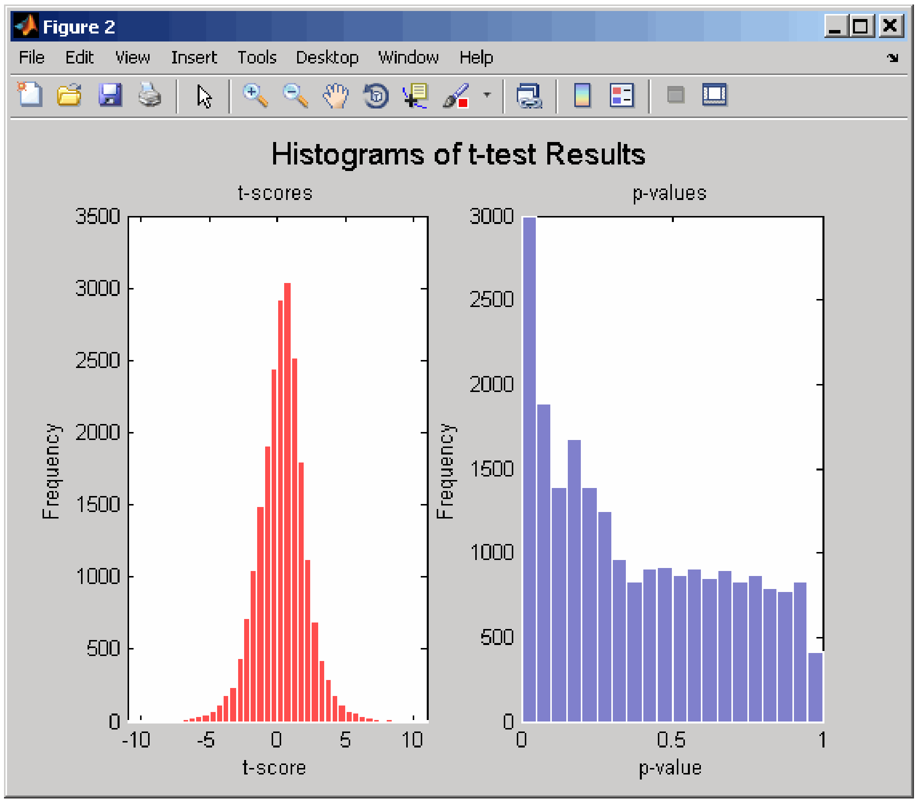
3 Calculate the p-values and t-scores again using permutation tests (1000 permutations) and displaying histograms of t-score distributions and p-value distributions.

```
[pvalues,tscores] = mattest(dependentData,independentData,...
                    'permute',true,'showhist',true,...
                    'showplot',true);
```

4 Calculate the p-values and t-scores again using bootstrap tests (2000 tests) and displaying histograms of t-score distributions and p-value distributions.

```
[pvalues,tscores] = mattest(dependentData,independentData,...
                            'bootstrap',2000,'showhist',true,...
                            'showplot',true);
```





The `prostatecancerexpdata.mat` file used in this example contains data from Best et al., 2005.

Version History

Introduced in R2006a

References

[1] Review Literature: Huber, W., von Heydebreck, A., Sültmann, H., Poustka, A., and Vingron, M. (2002). Variance stabilization applied to microarray data calibration and to the quantification of differential expression. Bioinformatics *18 (Suppl. 1)*, S96–S104.

[2] Best, C.J.M., Gillespie, J.W., Yi, Y., Chandramouli, G.V.R., Perlmutter, M.A., Gathright, Y., Erickson, H.S., Georgevich, L., Tangrea, M.A., Duray, P.H., Gonzalez, S., Velasco, A., Linehan, W.M., Matusik, R.J., Price, D.K., Figg, W.D., Emmert-Buck, M.R., and Chuaqui, R.F. (2005). Molecular alterations in primary prostate cancer after androgen ablation therapy. Clinical Cancer Research *11*, 6823–6834.

See Also

`affygcrma` | `affyrma` | `maboxplot` | `mafdr` | `mainvarsetnorm` | `mairplot` | `maloglog` | `malowess` | `manorm` | `mavolcanoplot` | `rmasummary`

mavolcanoplot

Create significance versus gene expression ratio (fold change) scatter plot of microarray data

Syntax

```
mavolcanoplot(DataX, DataY, PValues)
SigStructure = mavolcanoplot(DataX, DataY, PValues)

... mavolcanoplot(..., 'Labels', LabelsValue, ...)
... mavolcanoplot(..., 'LogTrans', LogTransValue, ...)
... mavolcanoplot(..., 'PCutoff', PCutoffValue, ...)
... mavolcanoplot(..., 'Foldchange', FoldchangeValue, ...)
... mavolcanoplot(..., 'PlotOnly', PlotOnlyValue, ...)
```

Input Arguments

DataX, *DataY*	DataMatrix object on page 1-734, matrix, or vector of gene expression values from a single experimental condition. If a DataMatrix object or a matrix, each row is a gene, each column is a sample, and an average expression value is calculated for each gene. **Note** If the values in *DataX* or *DataY* are natural scale, use the `LogTrans` property to convert them to log $_2$ scale.
PValues	Either of the following: • Column vector of p-values for each feature (for example, gene) in a data set, such as returned by `mattest`. • DataMatrix object on page 1-734 containing p-values for each feature (for example, gene) in a data set, such as returned by `mattest`.
LabelsValue	Cell array of character vectors or string vector containing labels (typically gene names or probe set IDs) for the data. After creating the plot, you can click a data point to display the label associated with it. If you do not provide a *LabelsValue*, data points are labeled with row numbers from *DataX* and *DataY*.
LogTransValue	Property to control the conversion of data in *DataX* and *DataY* from natural scale to log $_2$ scale. Enter `true` to convert data to log $_2$ scale, or `false`. Default is `false`, which assumes data is already log $_2$ scale.

PCutoffValue	Lets you specify a cutoff p-value to define data points that are statistically significant. This value is displayed graphically as a horizontal line on the plot. Default is `0.05`, which is equivalent to 1.3010 on the $-\log_{10}$ (p-value) scale. The value must be between 0 and 1. **Note** You can also change the p-value cutoff interactively after creating the plot.
FoldchangeValue	Lets you specify a ratio fold change to define data points that are differentially expressed. Default is `2`, which corresponds to a ratio of 1 and –1 on a $\log_2$ (ratio) scale. **Note** You can also change the fold change interactively after creating the plot.
PlotOnlyValue	Controls the display of the volcano plot without user interface components. Choices are `true` or `false` (default). **Note** If you set the `'PlotOnly'` property to `true`, you can still display labels for data points by clicking a data point, and you can still adjust vertical fold change lines and the horizontal p-value cutoff line by click-dragging the lines.

Output Arguments

SigStructure	Structure containing information for genes that are considered to be both statistically significant (above the p-value cutoff) and significantly differentially expressed (outside of the fold change values). The fields are listed below.

Description

`mavolcanoplot(`*`DataX`*`, `*`DataY`*`, `*`PValues`*`)` creates a scatter plot of gene expression data, plotting significance versus fold change of gene expression ratios of two data sets, *`DataX`* and *`DataY`*. It plots significance as the $-\log_{10}$ (p-value) from the input, *`PValues`*. *`DataX`* and *`DataY`* can be vectors, matrices, or DataMatrix object on page 1-734s. *`PValues`* is a column vector or DataMatrix object on page 1-734.

`SigStructure`` = mavolcanoplot(`*`DataX`*`, `*`DataY`*`, `*`PValues`*`)` returns a structure containing information for genes that are considered to be both statistically significant (above the p-value cutoff) and significantly differentially expressed (outside of the fold change values). The fields within *`SigStructure`* are sorted by p-value and include:

- `Name`
- `PCutoff`
- `FCThreshold`
- `GeneLabels`
- `PValues`

- `FoldChanges`

Note The fields `PValues` and `FoldChanges` will be either vectors or DataMatrix objects depending on the type of input *`PValues`*.

`... mavolcanoplot(..., '`*`PropertyName`*`', `*`PropertyValue`*`, ...)` defines optional properties that use property name/value pairs in any order. These property name/value pairs are as follows:

`... mavolcanoplot(..., 'Labels', `*`LabelsValue`*`, ...)` lets you provide a cell array of character vectors or string vector containing labels (typically gene names or probe set IDs) for the data. After creating the plot, you can click a data point to display the label associated with it. If you do not provide a *`LabelsValue`*, data points are labeled with row numbers from *`DataX`* and *`DataY`*.

`... mavolcanoplot(..., 'LogTrans', `*`LogTransValue`*`, ...)` controls the conversion of data from *`DataX`* and *`DataY`* to $\log_2$ scale. When *`LogTransValue`* is `true`, `mavolcanoplot` converts data from natural to $\log_2$ scale. Default is `false`, which assumes the data is already $\log_2$ scale.

`... mavolcanoplot(..., 'PCutoff', `*`PCutoffValue`*`, ...)` lets you specify a p-value cutoff to define data points that are statistically significant. This value displays graphically as a horizontal line on the plot. Default is `0.05`, which is equivalent to 1.3010 on the $-\log_{10}$ (p-value) scale.

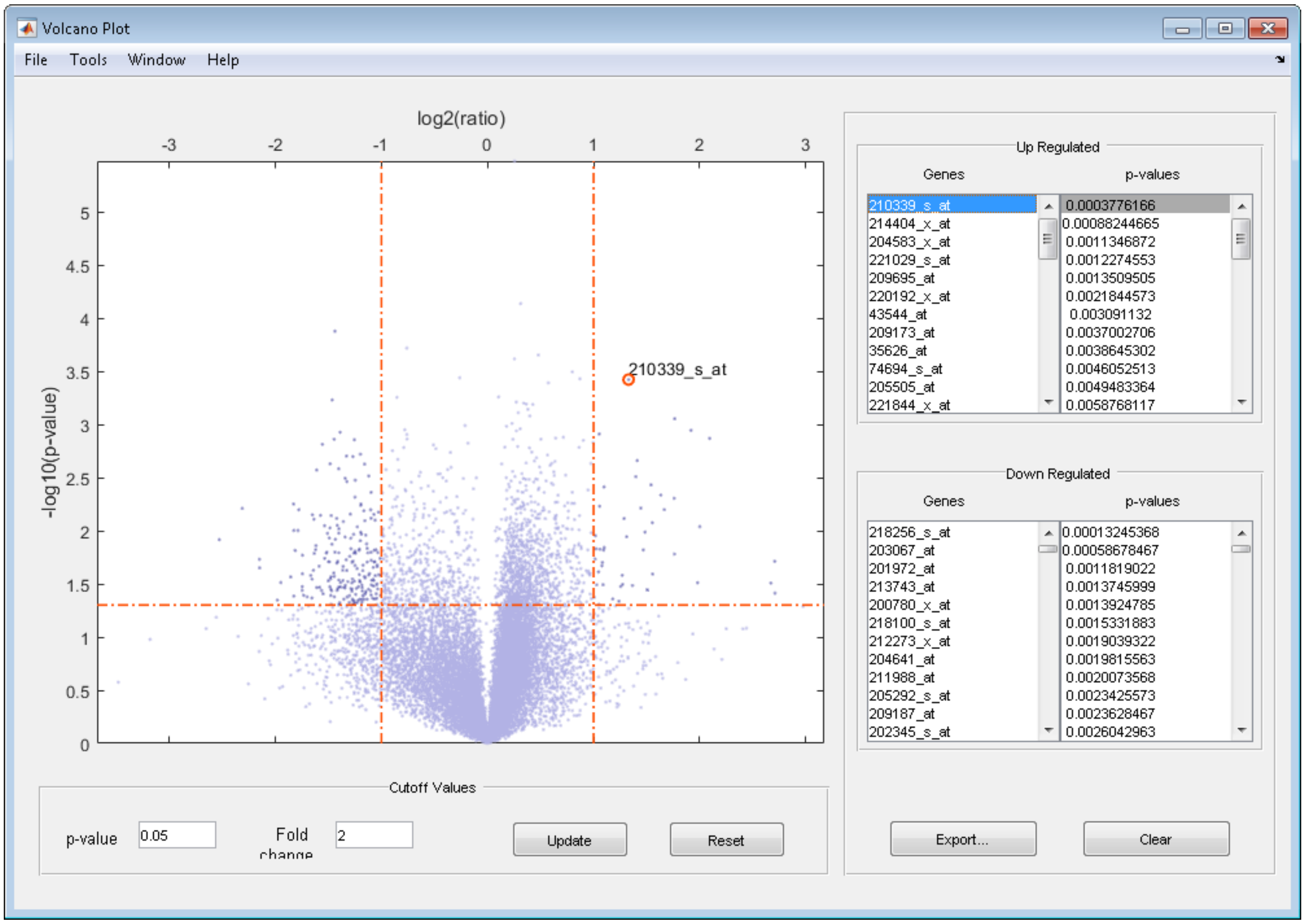
Note You can also change the p-value cutoff interactively after creating the plot.

`... mavolcanoplot(..., 'Foldchange', `*`FoldchangeValue`*`, ...)` lets you specify a ratio fold change to define data points that are differentially expressed. Fold changes display graphically as two vertical lines on the plot. Default is `2`, which corresponds to a ratio of 1 and –1 on a $\log_2$ (ratio) scale.

Note You can also change the fold change interactively after creating the plot.

`... mavolcanoplot(..., 'PlotOnly', `*`PlotOnlyValue`*`, ...)` controls the display of the volcano plot without user interface components. Choices are `true` or `false` (default).

Note If you set the `'PlotOnly'` property to `true`, you can still display labels for data points by clicking a data point, and you can still adjust vertical fold change lines and the horizontal p-value cutoff line by click-dragging the lines.



Up Regulated

Genes	p-values
210339_s_at	0.0003776166
214404_x_at	0.00088244665
204583_x_at	0.0011346872
221029_s_at	0.0012274553
209695_at	0.0013509505
220192_x_at	0.0021844573
43544_at	0.003091132
209173_at	0.0037002706
35626_at	0.0038645302
74694_s_at	0.0046052513
205505_at	0.0049483364
221844_x_at	0.0058768117

Down Regulated

Genes	p-values
218256_s_at	0.00013245368
203067_at	0.00058678467
201972_at	0.0011819022
213743_at	0.0013745999
200780_x_at	0.0013924785
218100_s_at	0.0015331883
212273_x_at	0.0019039322
204641_at	0.0019815563
211988_at	0.0020073568
205292_s_at	0.0023425573
209187_at	0.0023628467
202345_s_at	0.0026042963

The volcano plot displays the following:

- $-\log_{10}$ (p-value) versus $\log_2$ (ratio) scatter plot of genes
- Two vertical fold change lines at a fold change level of 2, which corresponds to a ratio of 1 and –1 on a $\log_2$ (ratio) scale. (Lines will be at different fold change levels, if you used the `'Foldchange'` property.)
- One horizontal line at the 0.05 p-value level, which is equivalent to 1.3010 on the $-\log_{10}$ (p-value) scale. (The line will be at a different p-value level, if you used the `'PCutoff'` property.)
-

After you display the volcano scatter plot, you can interactively:

- Adjust the vertical fold change lines by click-dragging one line or entering a value in the **Fold Change** text box.
- Adjust the horizontal p-value cutoff line by click-dragging or entering a value in the **p-value Cutoff** text box.
- Display labels for data points by clicking a data point.
- Select a gene from the **Up Regulated** or **Down Regulated** list to highlight the corresponding data point in the plot. Press and hold **Ctrl** or **Shift** to select multiple genes.

- Zoom the plot by selecting **Tools > Zoom In** or **Tools > Zoom Out**.
- View lists of significantly up-regulated and down-regulated genes and their associated p-values, and optionally, export the labels, p-values, and fold changes to a structure in the MATLAB Workspace by clicking **Export**.

Examples

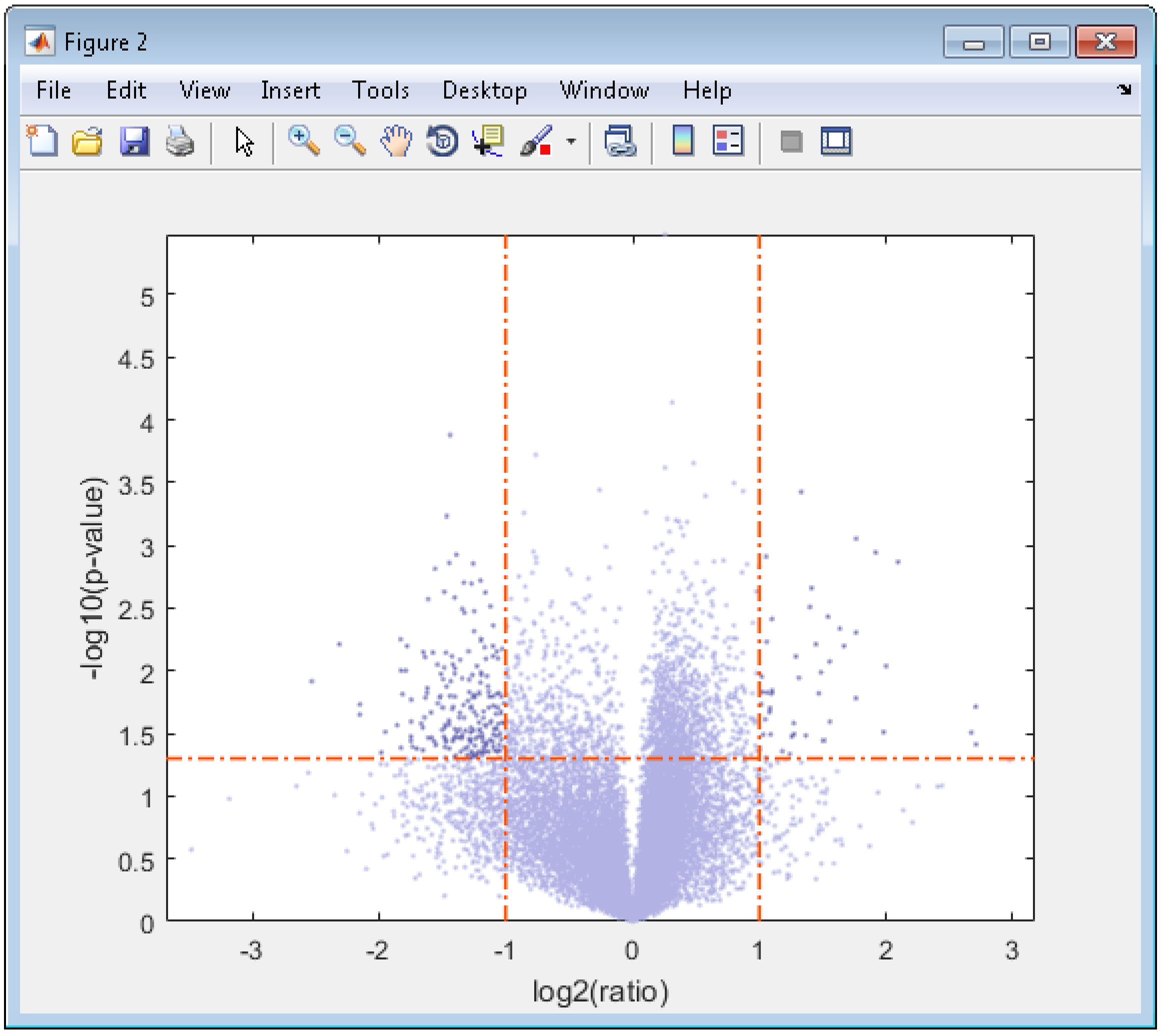
1. Load a MAT-file, included with the Bioinformatics Toolbox software, which contains Affymetrix data variables, including `dependentData` and `independentData`, two matrices of gene expression values from two experimental conditions.

   ```
   load prostatecancerexpdata
   ```
2. Use the `mattest` function to calculate p-values for the gene expression values in the two matrices.

   ```
   pvalues = mattest(dependentData, independentData);
   ```
3. Using the two matrices, the `pvalues` calculated by `mattest`, and the `probesetIDs` column vector of labels provided, use `mavolcanoplot` to create a significance versus gene expression ratio scatter plot of the microarray data from the two experimental conditions.

   ```
   mavolcanoplot(dependentData, independentData, pvalues,...
   'Labels', probesetIDs)
   ```
4. View the volcano plot without the user interface components.

   ```
   mavolcanoplot(dependentData, independentData, pvalues,...
   'Labels', probesetIDs,'Plotonly', true)
   ```



The `prostatecancerexpdata.mat` file used in the previous example contains data from Best et al., 2005.

Version History

Introduced in R2006a

References

[1] Cui, X., Churchill, G.A. (2003). Statistical tests for differential expression in cDNA microarray experiments. Genome Biology *4*, 210.

[2] Best, C.J.M., Gillespie, J.W., Yi, Y., Chandramouli, G.V.R., Perlmutter, M.A., Gathright, Y., Erickson, H.S., Georgevich, L., Tangrea, M.A., Duray, P.H., Gonzalez, S., Velasco, A., Linehan, W.M., Matusik, R.J., Price, D.K., Figg, W.D., Emmert-Buck, M.R., and Chuaqui, R.F. (2005). Molecular alterations in primary prostate cancer after androgen ablation therapy. Clinical Cancer Research *11*, 6823–6834.

See Also

maboxplot | maimage | mainvarsetnorm | mairplot | maloglog | malowess | manorm | mapcaplot | mattest

max (DataMatrix)

Return maximum values in DataMatrix object

Syntax

```
M = max(DMObj1)
[M, Indices] = max(DMObj1)
[M, Indices, Names] = max(DMObj1)
... = max(DMObj1, [], Dim)
MA = max(DMObj1, DMObj2)
```

Input Arguments

DMObj1, *DMObj2*	DataMatrix objects, such as created by `DataMatrix` (object constructor). **Note** *DMObj1* and *DMObj2* must be the same size, unless one is a scalar.
Dim	Scalar specifying the dimension of *DMObj* to return the maximum values. Choices are: • `1` — Default. Returns a row vector containing a maximum value for each column. • `2` — Returns a column vector containing a maximum value for each row.

Output Arguments

M	One of the following: • Scalar specifying the maximum value in *DMObj* when it contains vector of data • Row vector containing the maximum value for each column in *DMObj* (when *Dim* = `1`) • Column vector containing the maximum value for each row in *DMObj* (when *Dim* = `2`)
Indices	Either of the following: • Positive integer specifying the index of the maximum value in a DataMatrix object containing a vector of data • Vector containing the indices for the maximum value in each column (if *Dim* = `1`) or row (if *Dim* = `2`) in a DataMatrix object containing a matrix of data
Names	Vector of the row names (if *Dim* = `1`) or column names (if *Dim* = `2`) corresponding to the maximum value in each column or each row of a DataMatrix object.

MA	Numeric array created from the maximum elements in either of the following: • Two DataMatrix objects • A DataMatrix object and a numeric array

Description

`M = max(DMObj1)` returns the maximum value(s) in *DMObj1*, a DataMatrix object. If *DMObj1* contains a vector of data, *M* is a scalar. If *DMObj1* contains a matrix of data, *M* is a row vector containing a maximum value in each column.

`[M, Indices] = max(DMObj1)` returns *Indices*, the indices of the maximum value(s) in *DMObj1*, a DataMatrix object. If *DMObj1* contains a vector of data, *Indices* is a positive integer. If *DMObj1* contains a matrix of data, *Indices* is a vector containing the indices for the maximum value in each column (if *Dim* = 1) or row (if *Dim* = 2). If there are multiple maximum values in a column or row, the index for the first value is returned.

`[M, Indices, Names] = max(DMObj1)` returns *Names*, a vector of the row names (if *Dim* = 1) or column names (if *Dim* = 2) corresponding to the maximum value in each column or each row of*DMObj1*, a DataMatrix object. If there are multiple maximum values in a column or row, the row or column name for the first value is returned.

`... = max(DMObj1, [], Dim)` specifies which dimension to return the maximum values for, that is each column or each row in a DataMatrix object. If *Dim* = 1, returns *M*, a row vector containing the maximum value in each column. If *Dim* = 2, returns *M*, a column vector containing the maximum value in each row. Default *Dim* = 1.

`MA = max(DMObj1, DMObj2)` returns *MA*, a numeric array containing the larger of the two values from each position of *DMObj1* and *DMObj2*. *DMObj1* and *DMObj2* can both be DataMatrix objects, or one can be a DataMatrix object and the other a numeric array. They must be the same size, unless one is a scalar. *MA* has the same size (number of rows and columns) as the first nonscalar input.

Version History

Introduced in R2008b

See Also

`DataMatrix` | `min` | `sum`

Topics

DataMatrix object on page 1-734

maxflow (biograph)

(Removed) Calculate maximum flow in biograph object

Note The function has been removed. Use the `maxflow` function of `graph` or `digraph` instead.

Syntax

```
[MaxFlow, FlowMatrix, Cut] = maxflow(BGObj, SNode, TNode)

[...] = maxflow(BGObj, SNode, TNode, ...'Capacity', CapacityValue, ...)
[...] = maxflow(BGObj, SNode, TNode, ...'Method', MethodValue, ...)
```

Arguments

BGObj	Biograph object created by `biograph` (object constructor).
SNode	Node in a directed graph represented by an N-by-N adjacency matrix extracted from biograph object, *BGObj*.
TNode	Node in a directed graph represented by an N-by-N adjacency matrix extracted from biograph object, *BGObj*.
CapacityValue	Column vector that specifies custom capacities for the edges in the N-by-N adjacency matrix. It must have one entry for every nonzero value (edge) in the N-by-N adjacency matrix. The order of the custom capacities in the vector must match the order of the nonzero values in the N-by-N adjacency matrix when it is traversed column-wise. By default, `maxflow` gets capacity information from the nonzero entries in the N-by-N adjacency matrix.
MethodValue	Character vector or string that specifies the algorithm used to find the minimal spanning tree (MST). Choices are: • `'Edmonds'` — Uses the Edmonds and Karp algorithm, the implementation of which is based on a variation called the *labeling algorithm*. Time complexity is `O(N*E^2)`, where `N` and `E` are the number of nodes and edges respectively. • `'Goldberg'` — Default algorithm. Uses the Goldberg algorithm, which uses the generic method known as *preflow-push*. Time complexity is `O(N^2*sqrt(E))`, where `N` and `E` are the number of nodes and edges respectively.

Description

Tip For introductory information on graph theory functions, see “Graph Theory Functions”.

`[MaxFlow, FlowMatrix, Cut] = maxflow(BGObj, SNode, TNode)` calculates the maximum flow of a directed graph represented by an N-by-N adjacency matrix extracted from a biograph object, *`BGObj`*, from node *`SNode`* to node *`TNode`*. Nonzero entries in the matrix determine the capacity of the edges. Output *`MaxFlow`* is the maximum flow, and *`FlowMatrix`* is a sparse matrix with all the flow values for every edge. *`FlowMatrix`*(*X*,*Y*) is the flow from node *X* to node *Y*. Output *`Cut`* is a logical row vector indicating the nodes connected to *`SNode`* after calculating the minimum cut between *`SNode`* and *`TNode`*. If several solutions to the minimum cut problem exist, then *`Cut`* is a matrix.

Tip The algorithm that determines *`Cut`*, all minimum cuts, has a time complexity of `O(2^N)`, where *N* is the number of nodes. If this information is not needed, use the `maxflow` method without the third output.

`[...] = maxflow(BGObj, SNode, TNode, ...'PropertyName', PropertyValue, ...)` calls `maxflow` with optional properties that use property name/property value pairs. You can specify one or more properties in any order. Each *`PropertyName`* must be enclosed in single quotes and is case insensitive. These property name/property value pairs are as follows:

`[...] = maxflow(BGObj, SNode, TNode, ...'Capacity', CapacityValue, ...)` lets you specify custom capacities for the edges. *`CapacityValue`* is a column vector having one entry for every nonzero value (edge) in the N-by-N adjacency matrix. The order of the custom capacities in the vector must match the order of the nonzero values in the matrix when it is traversed column-wise. By default, `graphmaxflow` gets capacity information from the nonzero entries in the matrix.

`[...] = maxflow(BGObj, SNode, TNode, ...'Method', MethodValue, ...)` lets you specify the algorithm used to find the minimal spanning tree (MST). Choices are:

- `'Edmonds'` — Uses the Edmonds and Karp algorithm, the implementation of which is based on a variation called the *labeling algorithm*. Time complexity is `O(N*E^2)`, where `N` and `E` are the number of nodes and edges respectively.
- `'Goldberg'` — Default algorithm. Uses the Goldberg algorithm, which uses the generic method known as *preflow-push*. Time complexity is `O(N^2*sqrt(E))`, where `N` and `E` are the number of nodes and edges respectively.

Version History

Introduced in R2006b

R2022b: Removed
Errors starting in R2022b

The function has been removed. Use the `maxflow` function of `graph` or `digraph` instead.

R2022a: Warns
Warns starting in R2022a

The function issues a warning that it will be removed in a future release.

R2021b: To be removed
Not recommended starting in R2021b

The function runs without warning, but it will be removed in a future release.

References

[1] Edmonds, J. and Karp, R.M. (1972). Theoretical improvements in the algorithmic efficiency for network flow problems. Journal of the ACM *19*, 248-264.

[2] Goldberg, A.V. (1985). A New Max-Flow Algorithm. MIT Technical Report MIT/LCS/TM-291, Laboratory for Computer Science, MIT.

[3] Siek, J.G., Lee, L-Q, and Lumsdaine, A. (2002). The Boost Graph Library User Guide and Reference Manual, (Upper Saddle River, NJ:Pearson Education).

See Also

`maxflow` | `graph` | `digraph`

mean (DataMatrix)

Return average or mean values in DataMatrix object

Syntax

```
M = mean(DMObj)
M = mean(DMObj, Dim)
M = mean(DMObj, Dim, IgnoreNaN)
```

Input Arguments

DMObj	DataMatrix object, such as created by `DataMatrix` (object constructor).
Dim	Scalar specifying the dimension of *DMObj* to calculate the means. Choices are: • `1` — Default. Returns mean values for elements in each column. • `2` — Returns mean values for elements in each row.
IgnoreNaN	Specifies if NaNs should be ignored. Choices are `true` (default) or `false`.

Output Arguments

M	Either of the following: • Row vector containing the mean values from elements in each column in *DMObj* (when *Dim* = `1`) • Column vector containing the mean values from elements in each row in *DMObj* (when *Dim* = `2`)

Description

M = `mean(`*DMObj*`)` returns the mean values of the elements in the columns of a DataMatrix object, treating NaNs as missing values. *M* is a row vector containing the mean values for elements in each column in *DMObj*.

M = `mean(`*DMObj*`, `*Dim*`)` returns the mean values of the elements in the columns or rows of a DataMatrix object, as specified by *Dim*. If *Dim* = `1`, returns *M*, a row vector containing the mean values for elements in each column in *DMObj*. If *Dim* = `2`, returns *M*, a column vector containing the mean values for elements in each row in *DMObj*. Default *Dim* = `1`.

M = `mean(`*DMObj*`, `*Dim*`, `*IgnoreNaN*`)` specifies if NaNs should be ignored. *IgnoreNaN* can be `true` (default) or `false`.

Version History

Introduced in R2008b

See Also

`DataMatrix` | `max` | `median` | `min` | `sum`

Topics

DataMatrix object on page 1-734

median (DataMatrix)

Return median values in DataMatrix object

Syntax

```
Med = median(DMObj)
Med = median(DMObj, Dim)
Med = median(DMObj, Dim, IgnoreNaN)
```

Input Arguments

DMObj	DataMatrix object, such as created by `DataMatrix` (object constructor).
Dim	Scalar specifying the dimension of *DMObj* to calculate the medians. Choices are: • `1` — Default. Returns median values for elements in each column. • `2` — Returns median values for elements in each row.
IgnoreNaN	Specifies if NaNs should be ignored. Choices are `true` (default) or `false`.

Output Arguments

Med	Either of the following: • Row vector containing the median values from elements in each column in *DMObj* (when *Dim* = 1) • Column vector containing the median values from elements in each row in *DMObj* (when *Dim* = 2)

Description

Med = median(*DMObj*) returns the median values of the elements in the columns of a DataMatrix object, treating NaNs as missing values. *Med* is a row vector containing the median values for elements in each column in *DMObj*.

Med = median(*DMObj*, *Dim*) returns the median values of the elements in the columns or rows of a DataMatrix object, as specified by *Dim*. If *Dim* = 1, returns *Med*, a row vector containing the median values for elements in each column in *DMObj*. If *Dim* = 2, returns *Med*, a column vector containing the median values for elements in each row in *DMObj*. Default *Dim* = 1.

Med = median(*DMObj*, *Dim*, *IgnoreNaN*) specifies if NaNs should be ignored. *IgnoreNaN* can be `true` (default) or `false`.

Version History

Introduced in R2008b

See Also

`DataMatrix` | `max` | `mean` | `min` | `sum`

Topics

DataMatrix object on page 1-734

metafeatures

Attractor metagene algorithm for feature engineering using mutual information-based learning

Syntax

```
M = metafeatures(X)
[M,W] = metafeatures(X)
[M,W,GSorted] = metafeatures(X,G)
[M,W,GSorted,GSortedInd] = metafeatures( ___ )
[ ___ ] = metafeatures( ___ ,Name,Value)
[ ___ ] = metafeatures(T)
[ ___ ] = metafeatures(T,Name,Value)
```

Description

`M = metafeatures(X)` returns the weighted sums of features `M` in `X` using the attractor metagene algorithm described in [1].

M is a *r*-by-*n* matrix. *r* is the number of metafeatures identified during each repetition of the algorithm. The default number of repetitions is 1. By default, only unique metafeatures are returned in M. If multiple repetitions result in the same metafeature, then just one copy is returned in `M`. *n* is the number of samples (patients or time points).

`X` is a *p*-by-*n* numeric matrix. *p* is the number of variables, features, or genes. In other words, rows of `X` correspond to variables, such as measurements of gene expression for different genes. Columns correspond to different samples, such as patients or time points.

`[M,W] = metafeatures(X)` returns a *p*-by-*r* matrix `W` containing metafeatures weights. `M = W'*X`. *p* is the number of variables. *r* is the number of unique metafeatures or the number of times the algorithm is repeated (the default is 1).

`[M,W,GSorted] = metafeatures(X,G)` uses a *p*-by-1 cell array of character vectors or string vector `G` containing the variable names and returns a *p*-by-*r* cell array of variable names `GSorted` sorted by the decreasing weight.

The *i*th column of `GSorted` lists the feature (variable) names in order of their contributions to the *i*th metafeature.

`[M,W,GSorted,GSortedInd] = metafeatures( ___ )` returns the indices `GSortedInd` such that `GSorted` = `G`(`GSortedInd`).

`[ ___ ] = metafeatures( ___ ,Name,Value)` uses additional options specified by one or more `Name,Value` pair arguments.

`[ ___ ] = metafeatures(T)` uses a *p*-by-*n* table `T`. Gene names are the row names of the table. `M = W'*T{:,:}`.

`[ ___ ] = metafeatures(T,Name,Value)` uses additional options specified by one or more `Name,Value` pair arguments.

Note It is possible that the number of metafeatures (*r*) returned in `M` can be fewer than the number of replicates (repetitions). Even though you may have set the number of replicates to a positive integer greater than 1, if each repetition returns the same metafeature, then *r* is 1, and `M` is 1-by-*n*. This is because, by default, the function returns only unique metafeatures. If you prefer to get all metafeatures, set `'ReturnUnique'` to `false`. A metafeature is considered unique if the Pearson correlation between it and all previously found metafeatures is less than the `'UniqueTolerance'` value (the default value is `0.98`).

Examples

Apply Attractor Metagene Algorithm to Gene Expression Data

Load the breast cancer gene expression data. The data was retrieved from the Cancer Genome Atlas (TCGA) on May 20, 2014 and contains gene expression data of 17814 genes for 590 different patients. The expression data is stored in the variable `geneExpression`. The gene names are stored in the variable `geneNames`.

```
load TCGA_Breast_Gene_Expression
```

The data has several NaN values.

```
sum(sum(isnan(geneExpression)))
```

```
ans =

        1695
```

Use the *k*-nearest neighbor imputation method to replace missing data with the corresponding value from an average of the *k* columns that are nearest.

```
geneExpression = knnimpute(geneExpression,3);
```

There are three common drivers of breast cancer: ERBB2, estrogen, and progestrone. `metafeatures` allows you to seed the starting weights to focus on the genes of interest. In this case, set the weight for each of these genes to 1 in three different rows of `startValues`. Each row corresponds to initial values for a different replicate (repetition).

```
erbb         = find(strcmp('ERBB2',geneNames));
estrogen     = find(strcmp('ESR1',geneNames));
progestrone  = find(strcmp('PGR',geneNames));

startValues = zeros(size(geneExpression,1),3);
startValues(erbb,1)        = 1;
startValues(estrogen,2)    = 1;
startValues(progestrone,3) = 1;
```

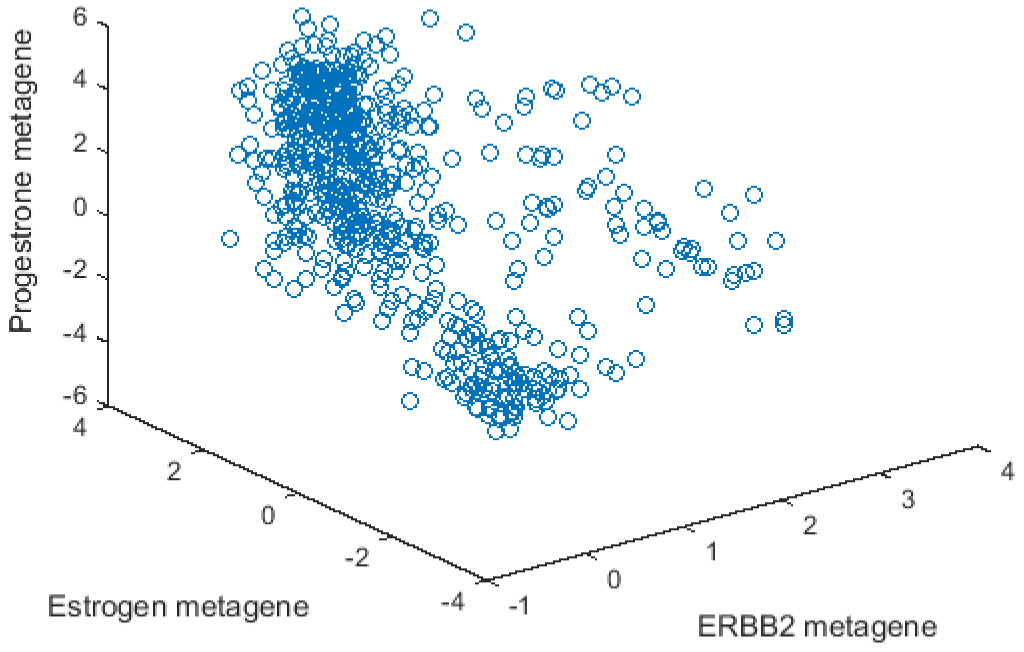
Apply the attractor metagene algorithm to the imputed data.

```
[meta, weights, genes_sorted] = metafeatures(geneExpression,geneNames,'start',startValues);
```

The variable `meta` has the value of three metagenes discovered for each sample. Plot these three metagenes to gain insight into the nature of gene regulation across different phenotypes of breast cancer.

```
plot3(meta(1,:),meta(2,:),meta(3,:),'o')
xlabel('ERBB2 metagene')
```

```
ylabel('Estrogen metagene')
zlabel('Progestrone metagene')
```



Based on the plot, observe the following.

- There is a group of points clustered together with low values for all three metagenes. Based on mRNA levels, these could be triple-negative or basal type breast cancer.
- There is a group of points that have high estrogen receptor metagene expression and span across both high and low progestrone metagene expression. There are no points with high progestrone metagene expression and low estrogen metagene expression. This is consistent with the observation that ER-/PR+ breast cancers are extremely rare [3].
- The remaining points are the ERBB2 positive cancers. They have less representation in this data set than the hormone-driven and triple negative cancers.

Input Arguments

X — Data

numeric matrix

Data, specified as a numeric matrix. Rows of X correspond to variables, such as measurements of gene expression. Columns correspond to different samples, such as patients or time points.

G — Variable names

cell array of character vectors | string vector

Variable names, specified as a cell array of character vectors or string vector.

T — Data
table

Data, specified as a table. The row names of the table correspond to the names of features or genes, and the columns represent different samples, such as patients or time points.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `'Replicates',5` specifies to repeat the algorithm five times.

`Alpha` — Tuning parameter for the number of metafeatures
5 (default) | positive scalar

Tuning parameter for the number of metafeatures, specified as the comma-separated pair consisting of `'Alpha'` and a positive number. This parameter controls the nonlinearity of the function that calculates the weights as described in the “Attractor Metagene Algorithm” on page 1-1244. As alpha increases, the number of metafeatures tends to increase. This parameter is often the most important parameter to adjust in the analysis of a data set.

Example: `'Alpha',3`

`Start` — Option for choosing initial weights
`'random'` (default) | `'robust'` | matrix

Option for choosing initial weights, specified as the comma-separated pair consisting of `'Start'` and a character vector, string, or matrix. This table summarizes the available options.

Option	Description
`'random'`	Initialize the weights to a vector of positive weights chosen uniformly at random and scaled such that they sum to 1. Choose a different initial weight vector for each replicate. This option is the default.
`'robust'`	If `X` or `T` has *n* columns, run the algorithm *n* times. On the *i*th evaluation of the algorithm, the weights are initialized to all zeros with the exception of the *i*th weight, which is set to 1. This option is useful when you are attempting to find all metafeatures of a data set.
matrix	*n*-by-*r* matrix of initial weights. The algorithm runs *r* times. The weights in the *i*th run of the algorithm are initialized to the *i*th column of the matrix.

Example: `'Start','robust'`

`Replicates` — Number of times to repeat the algorithm
1 (default) | positive integer

Number of times to repeat the algorithm, specified as the comma-separated pair consisting of `'Replicates'` and a positive integer. This option is valid only with the `'random'` start option. The default is 1.

Example: `'Replicates',2`

`ReturnUnique` — Unique metafeatures flag
`true` (default) | `false`

Unique metafeatures flag, specified as the comma-separated pair consisting of `'ReturnUnique'` and `true` or `false`. If true, then only the unique metafeatures are returned. The default is `true`.

This option is useful when the algorithm is repeated multiple times. By setting this option to `true`, you choose to look at just the unique metafeatures since the same set of metafeatures can be discovered for different initializations.

A metafeature is considered unique if the Pearson correlation between it and all previously found metafeatures is less than the `'UniqueTolerance'` value (the default value is `0.98`).

To run the algorithm multiple times, set the `'Replicates'` name-value pair argument or the `'Start'` option to `'robust'` or a matrix with more than 1 row.

Example: `'ReturnUnique',false`

`UniqueTolerance` — Tolerance for metafeature uniqueness
0.98 (default) | real number between 0 and 1

Tolerance for metafeature uniqueness, specified as the comma-separated pair consisting of `'UniqueTolerance'` and a real number between 0 and 1.

A metafeature is considered unique if the Pearson correlation between it and all previously found metafeatures is less than the `'UniqueTolerance'` value.

Example: `'UniqueTolerance',0.90`

`Options` — Options for controlling the algorithm
`[]` (default) | structure

Options for controlling the algorithm, specified as the comma-separated pair consisting of `'Options'` and a structure. This table summarizes these options.

Option	Description
`Display`	Level of output display. Choices are `'off'` or `'iter'`. The default is `'off'`.
`MaxIter`	Maximum number of iterations allowed. The default is 100.
`Tolerance`	If `M` changes by less than the tolerance in an iteration, then the algorithm stops. The default is `1e-6`.
`Streams`	A `RandStream` object. If you do not specify any streams, metafeatures uses the default random stream.
`UseParallel`	Logical value indicating whether to perform calculations in parallel if a parallel pool and Parallel Computing Toolbox are available. For problems with large data sets relative to the available system memory, running in parallel can degrade performance. The default is `false`.

Example: `'Options',struct('Display','iter')`

Output Arguments

`M` — Metafeatures
numeric matrix

Metafeatures, returned as a numeric matrix. It is an *r*-by-*n* matrix containing the weighted sums of the features in `X`. *r* is the number of replicates performed by the algorithm. *n* is the number of different samples such as time points or patients.

Note It is possible that the number of metafeatures (*r*) returned in `M` can be fewer than the number of replicates (repetitions). Even though you may have set the number of replicates to a positive integer greater than 1, if each repetition returns the same metafeature, then *r* is 1, and `M` is 1-by-*n*. This is because, by default, the function returns only unique metafeatures. If you prefer to get all metafeatures, set `'ReturnUnique'` to `false`. A metafeature is considered unique if the Pearson correlation between it and all previously found metafeatures is less than the `'UniqueTolerance'` value (the default value is `0.98`).

`W` — Metafeatures weights
numeric matrix

Metafeatures weights, returned as a numeric matrix. It is a *p*-by-*r* matrix. *p* is the number of variables. *r* is the number of replicates performed by the algorithm.

`GSorted` — Sorted variable names
cell array of character vectors

Sorted variable names, returned as a cell array of character vectors. It is a *p*-by-*r* cell array. The names are sorted by decreasing weight. The *i*th column of the `GSorted` lists the variable names in order of their contributions to *i*th metafeature.

If `GSorted` is requested without `G` or if `T.Properties.RowNames` is empty, then the algorithm names each variable (feature) as `Var`*i*, which corresponds to the *i*th row of `X`.

`GSortedInd` — Index to GSorted
matrix

Index to `GSorted`, returned as a matrix of indices. It is a *p*-by-*r* matrix. The indices satisfy `GSorted = G(GSortedInd)` or `GSorted = T.Properties.RowNames(GSortedInd)`.

More About

Attractor Metagene Algorithm

The attractor metagene algorithm [1] is an iterative algorithm that converges to metagenes with important features. A metagene is defined as any weighted sum of gene expression using a nonlinear distance metric. The distance metric is a nonlinear variant of mutual information using binning and splines as described in [2]. In fact, the use of mutual information as a distance metric is one of major benefits of this algorithm since mutual information is a robust information theoretic approach to determine the statistical dependence between variables. Therefore, it is useful for analyzing relationships among gene expression. Another advantage is that the results of the algorithm tend to be more clearly linked with a phenotype defined by gene expression.

The algorithm is initialized by either random or user-specified weights and proceeds in these steps.

1. The estimate of a metagene during the *i*th iteration of the algorithm is $M_i = W_i * G$, where W_i is a vector of weights of size 1-by-*p* (number of genes), and G is the gene expression matrix of size *p*-by-*n* (number of samples).
2. Update the weights by $W_{j,\,i+1} = J(M_i, G_j)$, where $W_{j,i+1}$ is the *j*th element of W_{i+1}, G_j is the *j*th row of *G*, and *J* is a similarity metric, which is defined as follows.

   - If the Pearson correlation between M_i and G_j is greater than 0, then $J(M_i, G_j) = I(M_i, G_j)^{\alpha}$, where $I(M_i, G_j)$ is the measure of mutual information between two genes with minimum value 0 and maximum value 1, and α is any nonnegative number.
   - If the correlation is less than or equal to 0, then $J(M_i, G_j) = 0$.

The algorithm iterates until the change in W_i between iterations is less than the defined tolerance, that is, $\|W_i - W_{i-1}\| < tolerance$ or the maximum number of iterations is reached.

The Role of α

In the similarity metric of the algorithm, the parameter α controls the degree of nonlinearity. As α increases, the number of metagenes tends to increase. If α is sufficiently large, then each gene approximately becomes an attractor metagene. If α is zero, then all weights remain equal to each other. Therefore, there is only one attractor metagene representing the average of all genes.

Therefore, adjusting α for the data set under consideration is a key step in fine tuning the algorithm. In the case of [1], using the TCGA data from several types of cancer to identify attractor metagenes, α value of 5 resulted in between 50 and 150 attractor metagenes discovered from the data.

Version History

Introduced in R2014b

References

[1] Cheng, W-Y., Ou Yang, T-H., and Anastassiou, D. (2013). Biomolecular events in cancer revealed by attractor metagenes. PLoS Computational Biology 9(2): e1002920.

[2] Daub, C., Steuer, R., Selbig, J., and Kloska, S. (2004). Estimating mutual information using B-spline functions – an improved similarity measure for analysing gene expression data. BMC Bioinformatics 5, 118.

[3] Hefti, M.M., Hu, R., Knoblauch, N.W., Collins, L.C., Haibe-Kains, B., Tamimi, R.M., and Beck, A.H. (2013). Estrogen receptor negative/progesterone receptor positive breast cancer is not a reproducible subtype. Breast Cancer Research. 15:R68.

Extended Capabilities

Automatic Parallel Support

Accelerate code by automatically running computation in parallel using Parallel Computing Toolbox™.

To run in parallel, set the `'UseParallel'` option to `true`.

Set the `'UseParallel'` field of the options structure to `true` and specify the `'Options'` name-value pair argument in the call to this function.

For example: `'Options',struct('UseParallel',true)`

For more information, see the `'Options'` name-value pair argument.

See Also

`relieff` | `sequentialfs` | `rankfeatures` | `randfeatures`

Topics

"Identifying Biomolecular Subgroups Using Attractor Metagenes"

microplateplot

Display visualization of microtiter plate

Syntax

```
microplateplot(Data)
Handle = microplateplot(...)

microplateplot(Data, ...'RowLabels', RowLabelsValue, ...)
microplateplot(Data, ...'ColumnLabels', ColumnLabelsValue, ...)
microplateplot(Data, ...'TextLabels', TextLabelsValue, ...)
microplateplot(Data, ...'TextFontSize', TextFontSizeValue, ...)
microplateplot(Data, ...'MissingValueColor', MissingValueColorValue, ...)
microplateplot(Data, ...'ToolTipFormat', ToolTipFormatValue, ...)
```

Description

`microplateplot(`*`Data`*`)` displays an image of a microtiter plate with each well colored according to intensity values, such as from a plate reader.

`Handle` `= microplateplot(...)` returns the handle to the axes of the plot.

`microplateplot(..., '`*`PropertyName`*`',` *`PropertyValue`*`, ...)` calls `microplateplot` with optional properties that use property name/property value pairs. You can specify one or more properties in any order. Enclose each *`PropertyName`* in single quotation marks. Each *`PropertyName`* is case insensitive. These property name/property value pairs are as follows:

`microplateplot(`*`Data`*`, ...'RowLabels',` *`RowLabelsValue`*`, ...)` lets you specify labels for the rows of data.

`microplateplot(`*`Data`*`, ...'ColumnLabels',` *`ColumnLabelsValue`*`, ...)` lets you specify labels for the columns of data.

`microplateplot(`*`Data`*`, ...'TextLabels',` *`TextLabelsValue`*`, ...)` lets you specify text to overlay of the wells in the image.

`microplateplot(`*`Data`*`, ...'TextFontSize',` *`TextFontSizeValue`*`, ...)` lets you specify the font size of the text you specify with the `'TextLabels'` property.

`microplateplot(`*`Data`*`, ...'MissingValueColor',` *`MissingValueColorValue`*`, ...)` lets you specify the color of wells with missing values (NaN values).

`microplateplot(`*`Data`*`, ...'ToolTipFormat',` *`ToolTipFormatValue`*`, ...)` lets you specify the format of the text used in the well tooltips. The well tooltips display the actual value from the input matrix when you click a well. *`ToolTipFormatValue`* is a format string, such as used by the `sprintf` function. Default is `'Value: %.3f'`, which specifies including three digits to the right of the decimal in fixed-point notation.

Input Arguments

`Data`

DataMatrix object on page 1-734 or matrix containing intensity values, such as from a plate reader.

Tip For help importing data from a spreadsheet or data file into a MATLAB matrix, see “Import Text Files”.

Note The `microplateplot` function converts any nonnumeric symbols or characters in the matrix to NaN values.

Default:

`RowLabelsValue`

Cell array of character vectors or string vector that specifies labels for the rows of data. Default is the first *`N`*letters of the alphabet, where *`N`* is the number of rows in *`Data`*. If there are more than 26 rows in *`Data`*, then the default is `AA, AB, ..., ZZ`. If *`Data`* is a DataMatrix object, then the default is the row labels of *`Data`*.

Default:

`ColumnLabelsValue`

Cell array of character vectors or string vector that specifies labels for the columns of data. Default is `1, 2, ...,` *`M`* , where *`M`* is the number of columns in *`Data`*. If *`Data`* is a DataMatrix object, then the default is the column labels of *`Data`*.

Default:

`TextLabelsValue`

Cell array of character vectors or string vector the same size as *`Data`* that specifies text to overlay on the wells of the image.

Default:

`TextFontSizeValue`

Positive integer specifying the font size of the text you specify with the `'TextLabels'` property. Default font size is determined automatically based on the size of the Figure window.

Default:

`MissingValueColorValue`

Three-element numeric vector of RGB values that specifies the color of wells with missing values (NaN values). Default is `[0, 0, 0]`, which defines black.

Default:

`ToolTipFormatValue`

Format string, such as used by the `sprintf` function, that specifies the format of the text used in the well tooltips. The well tooltips display the actual value from the input matrix when you click a well.

Default: `'Value: %.3f'`, which specifies including three digits to the right of the decimal in fixed-point notation.

Output Arguments

`Handle`

Handle to the axes of the plot.

Tip Use the *`Handle`* output with the `set` function and the `'YDir'` or `'XDir'` property to reverse the order of the `A` through `H` labels or `1` through `12` labels respectively. Note that in the microplate plot, the default order for the `A` through `H` labels, or `'YDir'` property, is `'reverse'` (top to bottom), and the default order for the `1` through `12` labels, or `'XDir'` property, is `'normal'` (left to right). For more information on the `'XDir'` and `'YDir'` properties, see Axes.

Examples

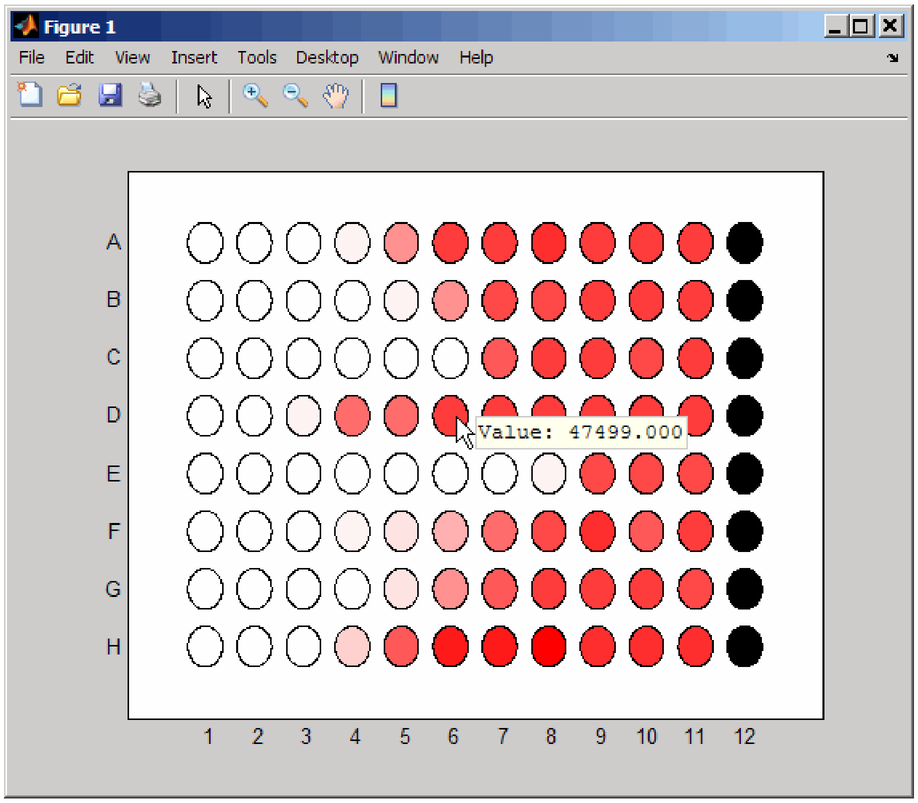
Example 1.20. Creating a Plot of a Microplate, Changing the Colormap, Viewing Well Values, and Adding Text Labels

1. Load a MAT-file, included with the Bioinformatics Toolbox software, which contains two variables: `assaydata`, an 8-by-12 matrix of data values from a microtiter plate, and `whiteToRed`, a 64-by-3 matrix that defines a colormap.

   ```
   load microPlateAssay
   ```
2. Create a visualization of the data from the microtiter plate.

   ```
   microplateplot(assaydata)
   ```
3. Change the visualization to use a white-to-red colormap, and then view a tooltip displaying the value of well D6 by clicking the well.

   ```
   colormap(whiteToRed)
   ```



Notice that all wells in column 12 are black, indicating missing data.

4 Overlay an X on well E8.

a Create an empty cell array.

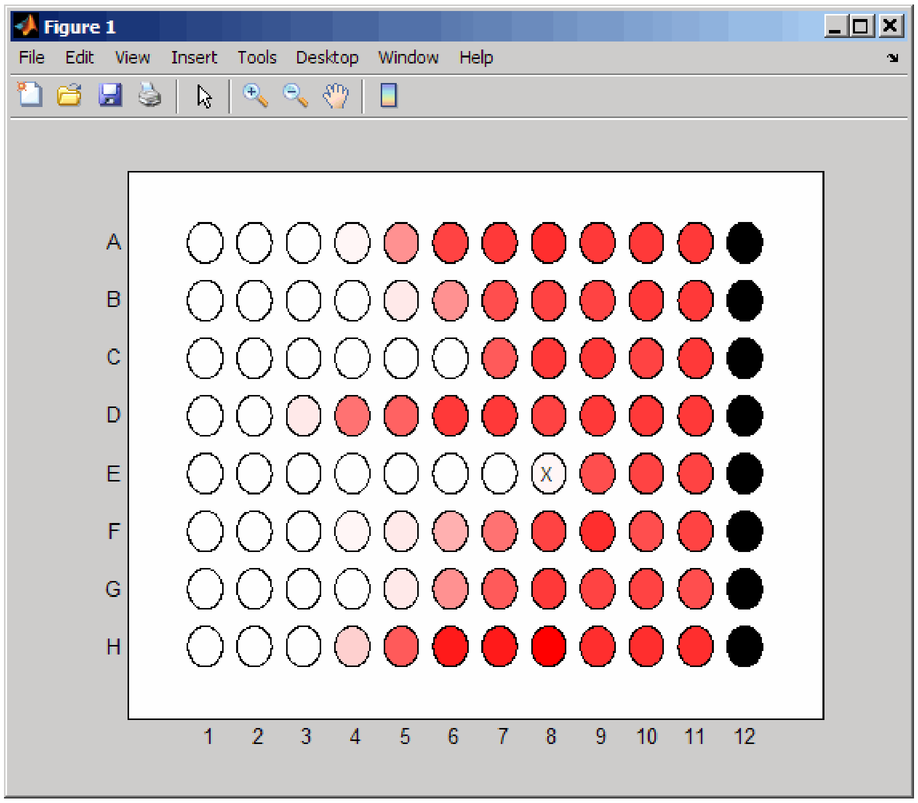
```
mask = cell(8,12);
```

b Add the string `'X'` to the cell in the fifth row and eighth column of the array.

```
mask{5,8} = 'X';
```

c Pass the cell array to the `microplateplot` function using the `'TextLabels'` property.

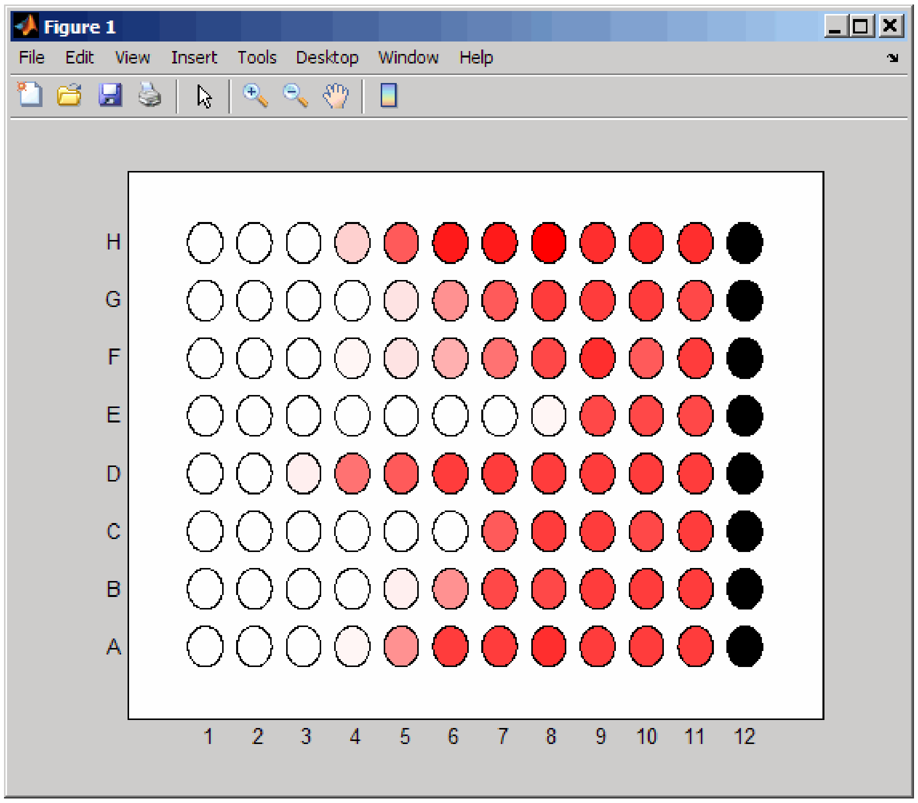
```
microplateplot(assaydata,'TEXTLABELS',mask);
```



Example 1.21. Changing the Order of Row Labels in the Plot

1. If you have not already done so, create a plot of a microplate as described previously.
2. Return a handle to the axes of the plot, and then reverse the order of the row letter labels.

```
h = microplateplot(assaydata);
set(h,'YDir','normal')
```



Example 1.22. Adding a Title and Axis Labels to the Plot

For information on adding a title and *x*-axis and *y*-axis labels to your plot, see “Add Title and Axis Labels to Chart”.

Example 1.23. Printing and Exporting the Plot

For information on printing or exporting your plot, see “Printing and Saving”.

Version History

Introduced in R2009a

See Also

`imagesc` | `sprintf` | `set`

Topics

“Printing and Saving”

min (DataMatrix)

Return minimum values in DataMatrix object

Syntax

```
M = min(DMObj1)
[M, Indices] = min(DMObj1)
[M, Indices, Names] = min(DMObj1)
... = min(DMObj1, [], Dim)
MA = min(DMObj1, DMObj2)
```

Input Arguments

DMObj1, *DMObj2*	DataMatrix objects, such as created by `DataMatrix` (object constructor). **Note** *DMObj1* and *DMObj2* must be the same size, unless one is a scalar.
Dim	Scalar specifying the dimension of *DMObj* to return the minimum values. Choices are: • `1` — Default. Returns a row vector containing a minimum value for each column. • `2` — Returns a column vector containing a minimum value for each row.

Output Arguments

M	One of the following: • Scalar specifying the minimum value in *DMObj* when it contains vector of data • Row vector containing the minimum value for each column in *DMObj* (when *Dim* = 1) • Column vector containing the minimum value for each row in *DMObj* (when *Dim* = 2)
Indices	Either of the following: • Positive integer specifying the index of the minimum value in a DataMatrix object containing a vector of data • Vector containing the indices for the minimum value in each column (if *Dim* = 1) or row (if *Dim* = 2) in a DataMatrix object containing a matrix of data
Names	Vector of the row names (if *Dim* = 1) or column names (if *Dim* = 2) corresponding to the minimum value in each column or each row of a DataMatrix object.

MA	Numeric array created from the minimum elements in either of the following: • Two DataMatrix objects • A DataMatrix object and a numeric array

Description

`M = min(DMObj1)` returns the minimum value(s) in `DMObj1`, a DataMatrix object. If `DMObj1` contains a vector of data, `M` is a scalar. If `DMObj1` contains a matrix of data, `M` is a row vector containing a minimum value in each column.

`[M, Indices] = min(DMObj1)` returns `Indices`, the indices of the minimum value(s) in `DMObj1`, a DataMatrix object. If `DMObj1` contains a vector of data, `Indices` is a positive integer. If `DMObj1` contains a matrix of data, `Indices` is a vector containing the indices for the minimum value in each column (if `Dim` = `1`) or row (if `Dim` = `2`). If there are multiple minimum values in a column or row, the index for the first value is returned.

`[M, Indices, Names] = min(DMObj1)` returns `Names`, a vector of the row names (if `Dim` = `1`) or column names (if `Dim` = `2`) corresponding to the minimum value in each column or each row of`DMObj1`, a DataMatrix object. If there is more than one minimum value in a column or row, the row or column name for the first value is returned.

`... = min(DMObj1, [], Dim)` specifies which dimension to return the minimum values for, that is each column or each row in a DataMatrix object. If `Dim` = `1`, returns `M`, a row vector containing the minimum value in each column. If `Dim` = `2`, returns `M`, a column vector containing the minimum value in each row. Default `Dim` = `1`.

`MA = min(DMObj1, DMObj2)` returns `MA`, a numeric array containing the smaller of the two values from each position of `DMObj1` and `DMObj2`. `DMObj1` and `DMObj2` can both be DataMatrix objects, or one can be a DataMatrix object and the other a numeric array. They must be the same size, unless one is a scalar. `MA` has the same size (number of rows and columns) as the first nonscalar input.

Version History

Introduced in R2008b

See Also

`DataMatrix` | `max` | `sum`

Topics

DataMatrix object on page 1-734

minspantree (biograph)

(Removed) Find minimal spanning tree in biograph object

Note The function has been removed. Use the `minspantree` function of `graph` or `digraph` instead.

Syntax

```
[Tree, pred] = minspantree(BGObj)
[Tree, pred] = minspantree(BGObj, R)

[Tree, pred] = minspantree(..., 'Method', MethodValue, ...)
[Tree, pred] = minspantree(..., 'Weights', WeightsValue, ...)
```

Arguments

BGObj	Biograph object created by `biograph` (object constructor).
R	Scalar between 1 and the number of nodes.

Description

`[`*`Tree`*`, `*`pred`*`] = minspantree(`*`BGObj`*`)` finds an acyclic subset of edges that connects all the nodes in the undirected graph represented by an N-by-N adjacency matrix extracted from a biograph object, *`BGObj`*, and for which the total weight is minimized. Weights of the edges are all nonzero entries in the lower triangle of the N-by-N sparse matrix. Output *`Tree`* is a spanning tree represented by a sparse matrix. Output *`pred`* is a vector containing the predecessor nodes of the minimal spanning tree (MST), with the root node indicated by `0`. The root node defaults to the first node in the largest connected component. This computation requires an extra call to the `graphconncomp` function.

Note The function ignores the direction of the edges in the Biograph object.

`[`*`Tree`*`, `*`pred`*`] = minspantree(`*`BGObj`*`, `*`R`*`)` sets the root of the minimal spanning tree to node `R`.

`[`*`Tree`*`, `*`pred`*`] = minspantree(..., '`*`PropertyName`*`', `*`PropertyValue`*`, ...)` calls `minspantree` with optional properties that use property name/property value pairs. You can specify one or more properties in any order. Each *`PropertyName`* must be enclosed in single quotes and is case insensitive. These property name/property value pairs are as follows:

`[`*`Tree`*`, `*`pred`*`] = minspantree(..., 'Method', `*`MethodValue`*`, ...)` lets you specify the algorithm used to find the minimal spanning tree (MST). Choices are:

- `'Kruskal'` — Grows the minimal spanning tree (MST) one edge at a time by finding an edge that connects two trees in a spreading forest of growing MSTs. Time complexity is `O(E+X*log(N))`, where `X` is the number of edges no longer than the longest edge in the MST, and `N` and `E` are the number of nodes and edges respectively.

- `'Prim'` — Default algorithm. Grows the minimal spanning tree (MST) one edge at a time by adding a minimal edge that connects a node in the growing MST with any other node. Time complexity is `O(E*log(N))`, where `N` and `E` are the number of nodes and edges respectively.

Note When the graph is unconnected, Prim's algorithm returns only the tree that contains R, while Kruskal's algorithm returns an MST for every component.

`[`*`Tree`*`, `*`pred`*`] = minspantree(..., 'Weights', `*`WeightsValue`*`, ...)` lets you specify custom weights for the edges. *`WeightsValue`* is a column vector having one entry for every nonzero value (edge) in the N-by-N sparse matrix. The order of the custom weights in the vector must match the order of the nonzero values in the N-by-N sparse matrix when it is traversed column-wise. By default, `minspantree` gets weight information from the nonzero entries in the N-by-N sparse matrix.

Version History

Introduced in R2006b

R2022b: Removed

Errors starting in R2022b

The function has been removed. Use the `minspantree` function of `graph` or `digraph` instead.

R2022a: Warns

Warns starting in R2022a

The function issues a warning that it will be removed in a future release.

R2021b: To be removed

Not recommended starting in R2021b

The function runs without warning, but it will be removed in a future release.

References

[1] Kruskal, J.B. (1956). On the Shortest Spanning Subtree of a Graph and the Traveling Salesman Problem. Proceedings of the American Mathematical Society 7, 48-50.

[2] Prim, R. (1957). Shortest Connection Networks and Some Generalizations. Bell System Technical Journal *36*, 1389-1401.

[3] Siek, J.G. Lee, L-Q, and Lumsdaine, A. (2002). The Boost Graph Library User Guide and Reference Manual, (Upper Saddle River, NJ:Pearson Education).

See Also

`minspantree` | `graph` | `digraph`

minus (DataMatrix)

Subtract DataMatrix objects

Syntax

```
DMObjNew = minus(DMObj1, DMObj2)
DMObjNew = DMObj1 - DMObj2
DMObjNew = minus(DMObj1, B)
DMObjNew = DMObj1 - B
DMObjNew = minus(B, DMObj1)
DMObjNew = B - DMObj1
```

Input Arguments

DMObj1, *DMObj2*	DataMatrix objects, such as created by `DataMatrix` (object constructor).
B	MATLAB numeric or logical array.

Output Arguments

DMObjNew	DataMatrix object created by subtraction.

Description

DMObjNew = `minus(`*DMObj1*, *DMObj2*`)` or the equivalent *DMObjNew* = *DMObj1* - *DMObj2* performs an element-by-element subtraction of the DataMatrix object *DMObj2* from the DataMatrix object *DMObj1* and places the results in *DMObjNew*, another DataMatrix object. *DMObj1* and *DMObj2* must have the same size (number of rows and columns), unless one is a scalar (1-by-1 DataMatrix object). The size (number of rows and columns), row names, and column names for *DMObjNew* are the same as *DMObj1*, unless *DMObj1* is a scalar; then they are the same as *DMObj2*.

DMObjNew = `minus(`*DMObj1*, *B*`)` or the equivalent *DMObjNew* = *DMObj1* - *B* performs an element-by-element subtraction of *B*, a numeric or logical array, from the DataMatrix object *DMObj1*, and places the results in *DMObjNew*, another DataMatrix object. *DMObj1* and *B* must have the same size (number of rows and columns), unless *B* is a scalar. The size (number of rows and columns), row names, and column names for *DMObjNew* are the same as *DMObj1*.

DMObjNew = `minus(`*B*, *DMObj1*`)` or the equivalent *DMObjNew* = *B* - *DMObj1* performs an element-by-element subtraction of the DataMatrix object *DMObj1* from *B*, a numeric or logical array, and places the results in *DMObjNew*, another DataMatrix object. *DMObj1* and *B* must have the same size (number of rows and columns), unless *B* is a scalar. The size (number of rows and columns), row names, and column names for *DMObjNew* are the same as *DMObj1*.

Note Arithmetic operations between a scalar DataMatrix object and a nonscalar array are not supported.

MATLAB calls `DMObjNew = minus(X, Y)` for the syntax `DMObjNew = X - Y` when *X* or *Y* is a DataMatrix object.

Version History

Introduced in R2008b

See Also

`DataMatrix` | `plus`

Topics

DataMatrix object on page 1-734

molweight

Calculate molecular weight of amino acid sequence

Syntax

```
molweight(SeqAA)
```

Arguments

SeqAA	Amino acid sequence. Enter a character vector, string, or a vector of integers from the tableAmino Acid Lookup. Examples: `'ARN'`, `[1 2 3]`. You can also enter a structure with the field `Sequence`.

Description

`molweight(`*`SeqAA`*`)` calculates the molecular weight for the amino acid sequence *SeqAA*.

Examples

1 Retrieve an amino acid sequence from the NCBI GenPept database.

```
rhodopsin = getgenpept('NP_000530');
```

2 Calculate the molecular weight of the sequence.

```
rhodopsinMW = molweight(rhodopsin)

rhodopsinMW =

  3.8892e+004
```

Version History

Introduced before R2006a

See Also

`aacount` | `atomiccomp` | `isoelectric` | `isotopicdist` | `proteinplot`

molviewer

(To be removed) Display and manipulate 3-D molecule structure

Note will be removed in a future release.

Syntax

```
molviewer
molviewer(File)
molviewer(pdbID)
molviewer(pdbStruct)
FigureHandle = molviewer(...)
```

Input Arguments

File	Character vector or string specifying one of the following: • File name of a file on the MATLAB search path or in the MATLAB Current Folder • Path and file name • URL pointing to a file (URL must begin with a protocol such as http://, ftp://, or file://) The referenced file is a molecule model file, such as a Protein Data Bank (PDB)-formatted file (ASCII text file). Valid file types include: • PDB • MOL (MDL) • SDF • XYZ • SMOL • JVXL • CIF/mmCIF
pdbID	Character vector or string specifying a unique identifier for a protein structure record in the PDB database. **Note** Each structure in the PDB database is represented by a four-character alphanumeric identifier. For example, `4hhb` is the identifier for hemoglobin.
pdbStruct	A structure containing a field for each PDB record, such as returned by the `getpdb` or `pdbread` function.

Output Arguments

FigureHandle	Figure handle to the Molecule Viewer.

Description

`molviewer` opens the Molecule Viewer app. You can display 3-D molecular structures by selecting **File > Open**, **File > Load PDB ID**, or **File > Open URL**.

`molviewer(`*`File`*`)` reads the data in a molecule model file, *`File`*, and opens the Molecule Viewer app displaying the 3-D molecular structure for viewing and manipulation.

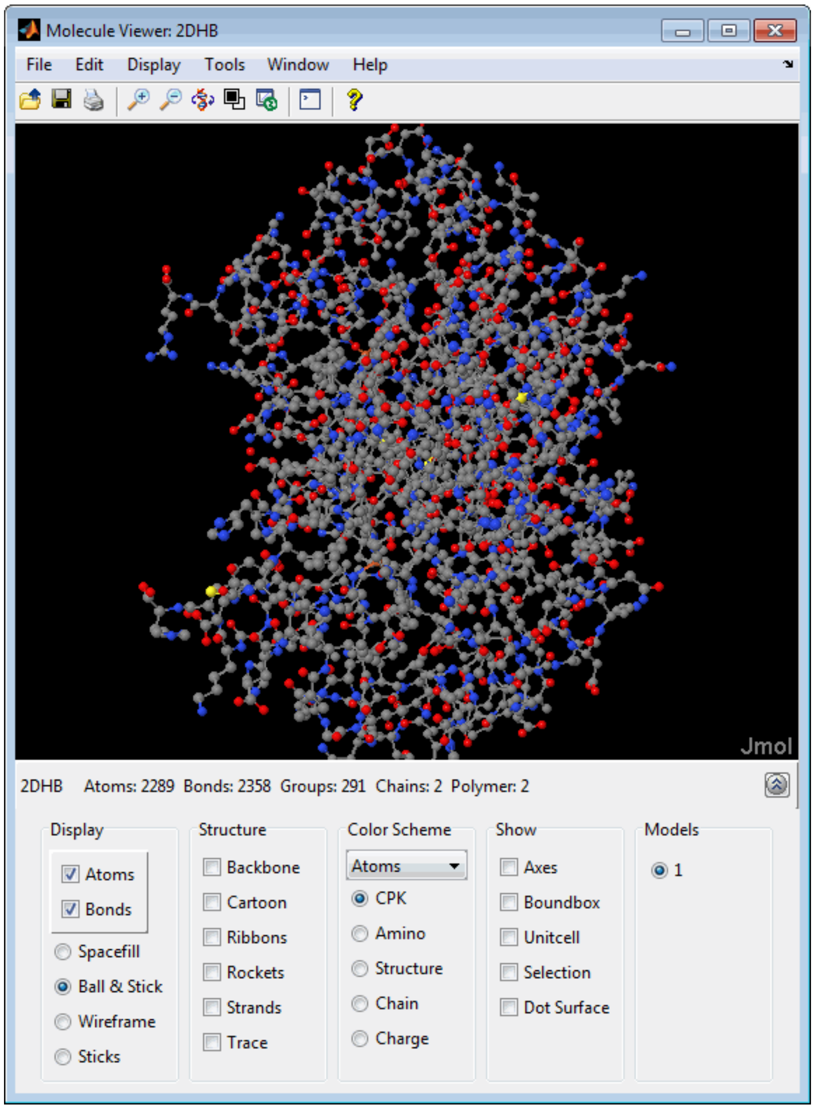
`molviewer(`*`pdbID`*`)` retrieves the structural data of a protein, *`pdbID`*, from the PDB database and opens the Molecule Viewer app displaying the 3-D molecular structure for viewing and manipulation.

`molviewer(`*`pdbStruct`*`)` reads the data from *`pdbStruct`*, a structure containing a field for each PDB record, and opens the Molecule Viewer app displaying a 3-D molecular structure for viewing and manipulation.

`FigureHandle`` = molviewer(...)` returns the figure handle to the Molecule Viewer window.

Tip You can pass the *`FigureHandle`* to the `evalrasmolscript` function, which sends RasMol script commands to the Molecule Viewer window.

Tip If you receive any errors related to memory or Java heap space, try increasing your Java heap space as described at https://www.mathworks.com/matlabcentral/answers/92813-how-do-i-increase-the-heap-space-for-the-java-vm-in-matlab.



2DHB Atoms: 2289 Bonds: 2358 Groups: 291 Chains: 2 Polymer: 2

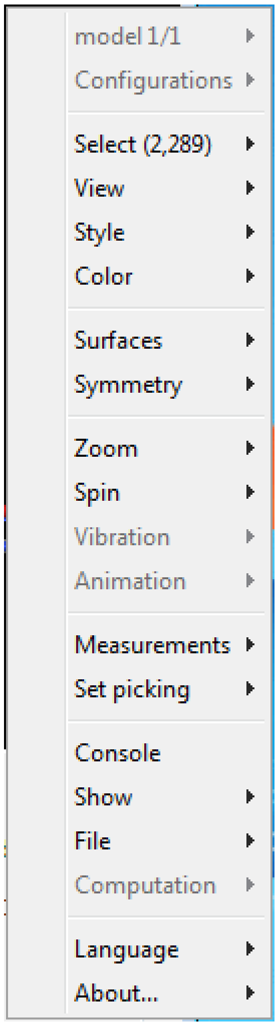
Display	Structure	Color Scheme	Show	Models
✓ Atoms	☐ Backbone	Atoms	☐ Axes	● 1
✓ Bonds	☐ Cartoon	● CPK	☐ Boundbox	
○ Spacefill	☐ Ribbons	○ Amino	☐ Unitcell	
● Ball & Stick	☐ Rockets	○ Structure	☐ Selection	
○ Wireframe	☐ Strands	○ Chain	☐ Dot Surface	
○ Sticks	☐ Trace	○ Charge		

After displaying the 3-D molecule structure, you can:

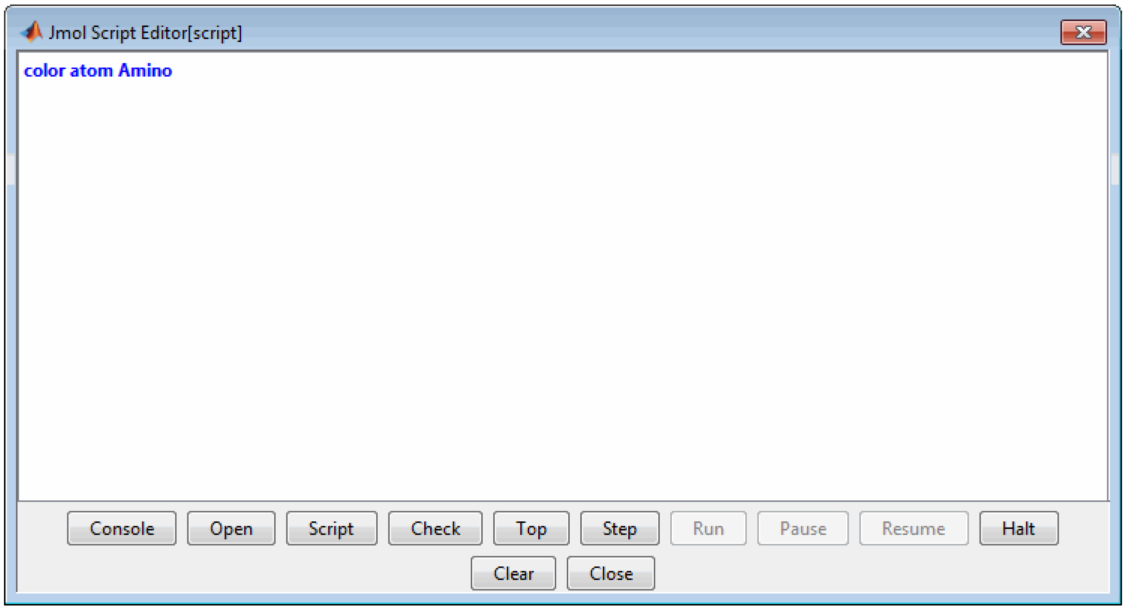
- Hover the mouse over a subcomponent of the molecule to display an identification label for it.
- Spin and rotate the molecule at different angles by click-dragging it.
- Spin the molecule in the *x*-*z* plane by clicking .
- Spin the molecule in the *x*-*y* plane by pressing and holding the **Shift** key, then click-dragging left and right.
- Zoom in a stepless fashion by pressing and holding the **Shift** key, then click-dragging up and down.
- Zoom in a stepwise fashion by clicking the figure, then turning the mouse scroll wheel, or by clicking the following buttons:

  or

- Move the molecule by pressing and holding **Ctrl** + **Alt**, then click-dragging.
- Change the background color between black and white by clicking .
- Reset the molecule position by clicking .
- Show or hide the Control Panel by clicking .
- Manipulate and annotate the 3-D structure by selecting options in the Control Panel or, for a complete list of options, by right-clicking the Molecule Viewer window to select commands:



- Display the Jmol Script Console by clicking .



```
color atom Amino
```

Note There is a known bug with the **Open** button of the script editor that prevents loading a Rasmol script interactively. Instead use the `evalrasmolscript` function which sends RasMol script commands to the **Molecule Viewer** app. Also, you can copy and paste the script commands into the script console.

Examples

View the H5N1 influenza virus hemagglutinin molecule, whose structural information is located at `www.rcsb.org/pdb/files/2FK0.pdb.gz`.

```
molviewer('https://www.rcsb.org/pdb/files/2FK0.pdb.gz')
```

View the molecule with a PDB identifier of `2DHB`.

```
molviewer('2DHB')
```

View the molecule with a PDB identifier of `4hhb`, and create a figure handle for the Molecule Viewer.

```
FH = molviewer('4hhb')
```

Use the `getpdb` function to retrieve protein structure data from the PDB database and create a MATLAB structure. Then view the protein molecule.

```
pdbstruct = getpdb('1vqx')
molviewer(pdbstruct)
```

Version History

Introduced in R2007a

R2023a: Warns

Warns starting in R2023a

`molviewer` issues a warning that it will be removed in a future release.

R2020b: To be removed
Not recommended starting in R2020b

`molviewer` runs without warning. But it will be removed in a future release.

See Also

`evalrasmolscript` | `getpdb` | `pdbread` | `pdbsuperpose` | `pdbtransform` | `pdbwrite`

msalign

Align peaks in signal to reference peaks

Syntax

```
IntensitiesOut = msalign(X, Intensities, RefX)

... = msalign(..., 'Rescaling', RescalingValue, ...)
... = msalign(..., 'Weights', WeightsValue, ...)
... = msalign(..., 'MaxShift', MaxShiftValue, ...)
... = msalign(..., 'WidthOfPulses', WidthOfPulsesValue, ...)
... = msalign(..., 'WindowSizeRatio', WindowSizeRatioValue, ...)
... = msalign(..., 'Iterations', IterationsValue, ...)
... = msalign(..., 'GridSteps', GridStepsValue, ...)
... = msalign(..., 'SearchSpace', SearchSpaceValue, ...)
... = msalign(..., 'ShowPlot', ShowPlotValue, ...)
[IntensitiesOut, RefXOut] = msalign(..., 'Group', GroupValue, ...)
```

Input Arguments

X	Vector of separation-unit values for a set of signals with peaks. The number of elements in the vector equals the number of rows in the matrix *Intensities*. The separation unit can quantify wavelength, frequency, distance, time, or m/z depending on the instrument that generates the signal data.
Intensities	Matrix of intensity values for a set of peaks that share the same separation-unit range. Each row corresponds to a separation-unit value, and each column corresponds to either a set of signals with peaks or a retention time. The number of rows equals the number of elements in vector *X*.
RefX	Vector of separation-unit values of known reference masses in a sample signal. **Tip** For reference peaks, select compounds that are not expected to have significant shifts among the different signals. For example, in mass spectrometry, select compounds that do not undergo structural transformation, such as phosphorylation. Doing so increases the accuracy of your alignment and lets you detect compounds that exhibit structural transformations among the sample signal.
RescalingValue	Controls the rescaling of *X*. Choices are `true` (default) or `false`. When `false`, the output signal is aligned only to the reference peaks by using constant shifts. By default, `msalign` estimates a rescaling factor, unless *RefX* contains only one reference peak.
WeightsValue	Vector of positive values, with the same number of elements as *RefX*. The default vector is `ones(size(`*RefX*`))`.

`MaxShiftValue`	Two-element vector, in which the first element is negative and the second element is positive, that specifies the lower and upper limits of a range, in separation units, relative to each peak. No peak shifts beyond these limits. Default is `[-100 100]`.
`WidthOfPulsesValue`	Positive value that specifies the width, in separation units, for all the Gaussian pulses used to build the correlating synthetic signal. The point of the peak where the Gaussian pulse reaches `60.65`% of its maximum is set to the width specified by `WidthOfPulsesValue`. Default is `10`.
`WindowSizeRatioValue`	Positive value that specifies a scaling factor that determines the size of the window around every alignment peak. The synthetic signal is compared to the input signal only within these regions, which saves computation time. The size of the window is given in separation-units by `WidthOfPulsesValue` * `WindowSizeRatioValue`. Default is `2.5`, which means at the limits of the window, the Gaussian pulses have a value of `4.39`% of their maximum.
`IterationsValue`	Positive integer that specifies the number of refining iterations. At every iteration, the search grid is scaled down to improve the estimates. Default is `5`.
`GridStepsValue`	Positive integer that specifies the number of steps for the search grid. At every iteration, the search area is divided by `GridStepsValue^2`. Default is `20`.
`SearchSpaceValue`	Character vector or string that specifies the type of search space. Choices are: • `'regular'` — Default. Evenly spaced lattice. • `'latin'` — Random Latin hypercube with `GridStepsValue^2` samples.
`ShowPlotValue`	Controls the display of a plot of an original and aligned signal over the reference masses specified by `RefX`. Choices are `true`, `false`, or `I`, an integer specifying the index of a signal in `Intensities`. If you set to `true`, the first signal in `Intensities` is plotted. Default is: • `false` — When return values are specified. • `true` — When return values are not specified.

GroupValue	Controls the creation of *RefXOut*, a new vector of separation-unit values to be used as reference masses for aligning the peaks. This vector is created by adjusting the values in *RefX*, based on the sample data from multiple signals in *Intensities*, such that the overall shifting and scaling of the peaks is minimized. Choices are `true` or `false` (default). **Tip** Set *GroupValue* to `true` only if *Intensities* contains data for a large number of signals, and you are not confident of the separation-unit values used for your reference peaks in *RefX*. Leave *GroupValue* set to `false` if you are confident of the separation-unit values used for your reference peaks in *RefX*.

Output Arguments

IntensitiesOut	Matrix of intensity values for a set of peaks that share the same separation-unit range. Each row corresponds to a separation-unit value, and each column corresponds to either a set of signals with peaks or a retention time. The intensity values represent a shifting and scaling of the data.
RefXOut	Vector of separation-unit values of reference masses, calculated from *RefX* and the sample data from multiple signals in *Intensities*, when you set *GroupValue* to `true`.

Description

Tip Use the following syntaxes with data from any separation technique that produces signal data, such as spectroscopy, NMR, electrophoresis, chromatography, or mass spectrometry.

IntensitiesOut `= msalign(`*X*`,` *Intensities*`,` *RefX*`)` aligns the peaks in raw, noisy signal data, represented by *Intensities* and *X*, to reference peaks, provided by *RefX*. First, it creates a synthetic signal from the reference peaks using Gaussian pulses centered at the separation-unit values specified by *RefX*. Then, it shifts and scales the separation-unit scale to find the maximum alignment between the input signals and the synthetic signal. (It uses an iterative multiresolution grid search until it finds the best scale and shift factors for each signal.) Once the new separation-unit scale is determined, the corrected signals are created by resampling their intensities at the original separation-unit values, creating *IntensitiesOut*, a vector or matrix of corrected intensity values. The resampling method preserves the shape of the peaks.

Tip The `msalign` function works best with three to five reference peaks that you know will appear in the signal. If you use a single reference peak (internal standard), there is a possibility of aligning sample peaks to the incorrect reference peaks as `msalign` both scales and shifts the *X* vector. If using a single reference peak, you might need to only shift the *X* vector. To do this, use *IntensitiesOut* `= interp1(`*X*`,` *Intensities*`,` *X*`-(`*ReferencePeak*`-`*ExperimentalPeak*`))`.

`... = msalign(..., '`*PropertyName*`',` *PropertyValue*`, ...)` calls `msalign` with optional properties that use property name/property value pairs. You can specify one or more properties in any order. Each *PropertyName* must be enclosed in single quotation marks and is case insensitive. These property name/property value pairs are as follows:

`... = msalign(..., 'Rescaling', `*`RescalingValue`*`, ...)` controls the rescaling of *X*. Choices are `true` (default) or `false`. When `false`, the output signal is aligned only to the reference peaks by using constant shifts. By default, `msalign` estimates a rescaling factor, unless *RefX* contains only one reference peak.

`... = msalign(..., 'Weights', `*`WeightsValue`*`, ...)` specifies the relative weight for each mass in *RefX*, the vector of reference separation-unit values. *WeightsValue* is a vector of positive values, with the same number of elements as *RefX*. The default vector is `ones(size(`*`RefX`*`))`, which means each reference peak is weighted equally, so that more intense reference peaks have a greater effect in the alignment algorithm. If you have a less intense reference peak, you can increase its weight to emphasize it more in the alignment algorithm.

`... = msalign(..., 'MaxShift', `*`MaxShiftValue`*`, ...)` specifies the lower and upper limits of the range, in separation units, relative to each peak. No peak shifts beyond these limits. *MaxShiftValue* is a two-element vector, in which the first element is negative and the second element is positive. Default is `[-100 100]`.

Note Use these values to tune the robustness of the algorithm. Ideally, you should keep the range within the maximum expected shift. If you try to correct larger shifts by increasing the limits, you increase the possibility of picking incorrect peaks to align to the reference masses.

`... = msalign(..., 'WidthOfPulses', `*`WidthOfPulsesValue`*`, ...)` specifies the width, in separation units, for all the Gaussian pulses used to build the correlating synthetic signal. The point of the peak where the Gaussian pulse reaches `60.65`% of its maximum is set to the width you specify with *WidthOfPulsesValue*. Choices are any positive value. Default is `10`. *WidthOfPulsesValue* may also be a function handle. The function is evaluated at the respective separation-unit values and returns a variable width for the pulses. Its evaluation should give reasonable values from `0` to `max(abs(Range))`; otherwise, the function returns an error.

Note Tuning the spread of the Gaussian pulses controls a tradeoff between robustness (wider pulses) and precision (narrower pulses). However, the spread of the pulses is unrelated to the shape of the observed peaks in the signal. The purpose of the pulse spread is to drive the optimization algorithm.

`... = msalign(..., 'WindowSizeRatio', `*`WindowSizeRatioValue`*`, ...)` specifies a scaling factor that determines the size of the window around every alignment peak. The synthetic signal is compared to the sample signal only within these regions, which saves computation time. The size of the window is given in separation units by *`WidthOfPulsesValue * WindowSizeRatioValue`*. Choices are any positive value. Default is `2.5`, which means at the limits of the window, the Gaussian pulses have a value of `4.39`% of their maximum.

`... = msalign(..., 'Iterations', `*`IterationsValue`*`, ...)` specifies the number of refining iterations. At every iteration, the search grid is scaled down to improve the estimates. Choices are any positive integer. Default is `5`.

`... = msalign(..., 'GridSteps', `*`GridStepsValue`*`, ...)` specifies the number of steps for the search grid. At every iteration, the search area is divided by *GridStepsValue*`^2`. Choices are any positive integer. Default is `20`.

`... = msalign(..., 'SearchSpace', SearchSpaceValue, ...)` specifies the type of search space. Choices are:

- `'regular'` — Default. Evenly spaced lattice.
- `'latin'` — Random Latin hypercube with *`GridStepsValue`*^2 samples.

`... = msalign(..., 'ShowPlot', ShowPlotValue, ...)` controls the display of a plot of an original and aligned signal over the reference masses specified by *`RefX`*. Choices are `true`, `false`, or *`I`*, an integer specifying the index of a signal in *`Intensities`*. If set to `true`, the first signal in *`Intensities`* is plotted. Default is:

- `false` — When return values are specified.
- `true` — When return values are not specified.

`[IntensitiesOut, RefXOut] = msalign(..., 'Group', GroupValue, ...)` controls the creation of *`RefXOut`*, a new vector of separation-unit values to use as reference masses for aligning the peaks. This vector is created by adjusting the values in *`RefX`*, based on the sample data from multiple signals in *`Intensities`*, such that the overall shifting and scaling of the peaks is minimized. Choices are `true` or `false` (default).

Tip Set *`GroupValue`* to `true` only if *`Intensities`* contains data for a large number of signals, and you are not confident of the separation-unit values used for your reference peaks in *`RefX`*. Leave *`GroupValue`* set to `false` if you are confident of the separation-unit values used for your reference peaks in *`RefX`*.

Examples

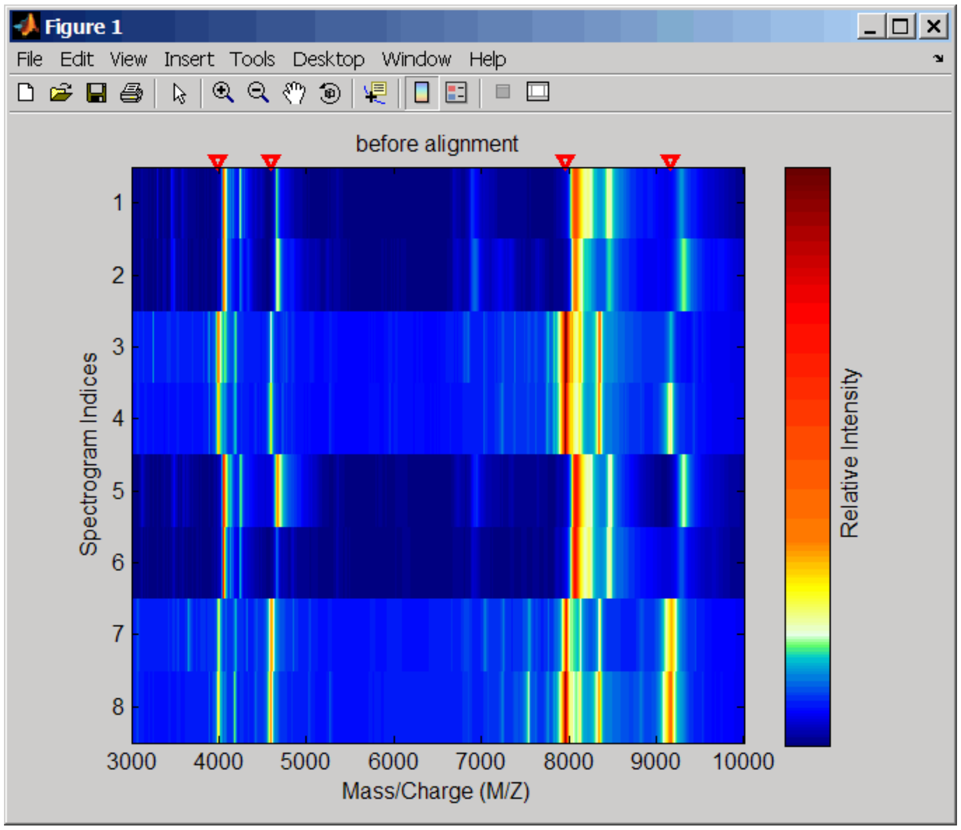
Example 1.24. Aligning a Mass Spectrum with Three or More Reference Peaks

1 Load a MAT-file, included with the Bioinformatics Toolbox software, that contains sample data, reference masses, and parameter data for synthetic peak width.

```
load sample_lo_res
R = [3991.4 4598 7964 9160];
W = [60 100 60 100];
```

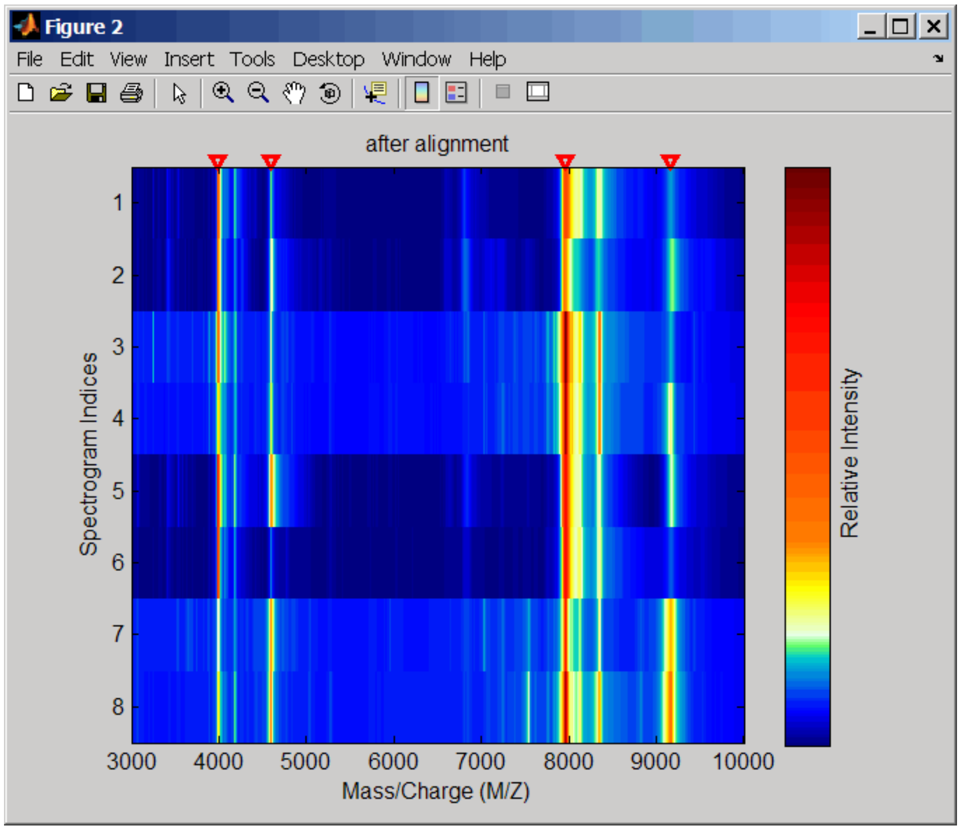
2 Display a color image of the mass spectra before alignment.

```
msheatmap(MZ_lo_res,Y_lo_res,'markers',R,'range',[3000 10000])
title('before alignment')
```



3 Align spectra with reference masses and display a color image of mass spectra after alignment.

```
YA = msalign(MZ_lo_res,Y_lo_res,R,'weights',W);
msheatmap(MZ_lo_res,YA,'markers',R,'range',[3000 10000])
title('after alignment')
```

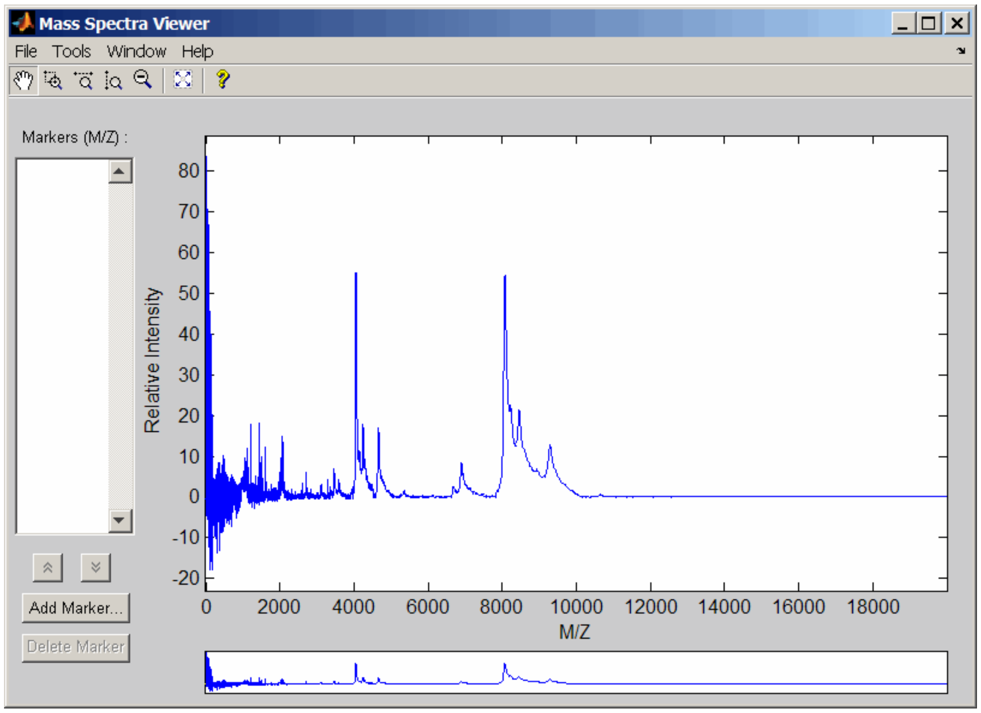


Example 1.25. Aligning a Mass Spectrum with One Reference Peak

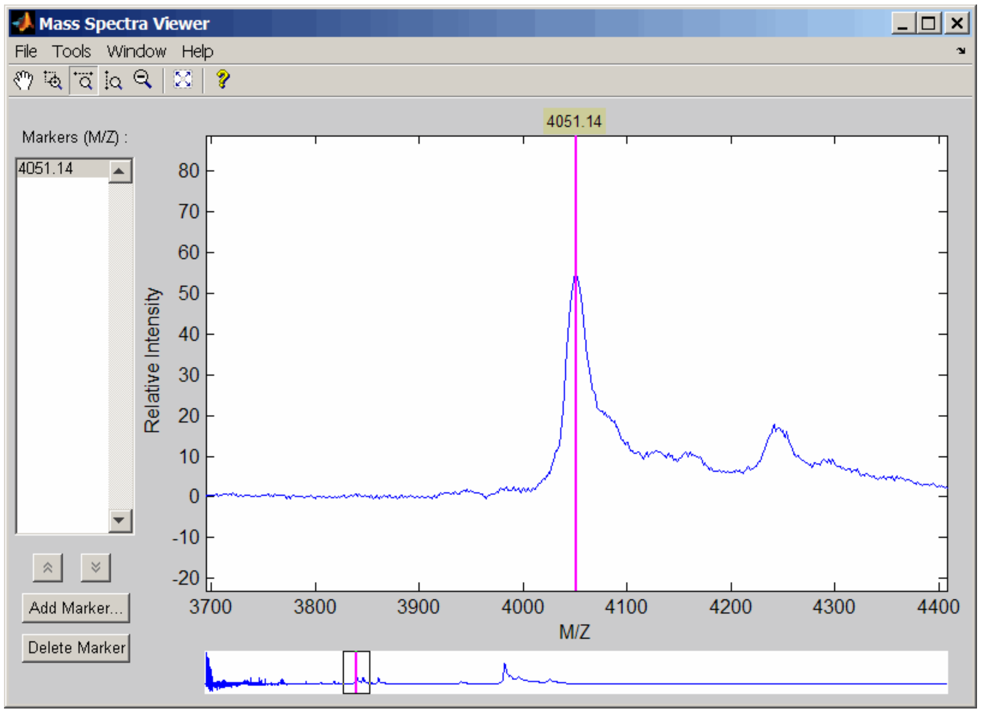
It is not recommended to use the `msalign` function if you have only one reference peak. Instead, use the following procedure, which shifts the *X* input vector, but does not scale it.

1 Load sample data and view the first sample spectrum.

```
load sample_lo_res
MZ = MZ_lo_res;
Y = Y_lo_res(:,1);
msviewer(MZ, Y)
```



2 Use the tall peak around 4000 m/z as the reference peak. To determine the reference peak's m/z value, click , and then click-drag to zoom in on the peak. Right-click in the center of the peak, and then click **Add Marker** to label the peak with its m/z value.

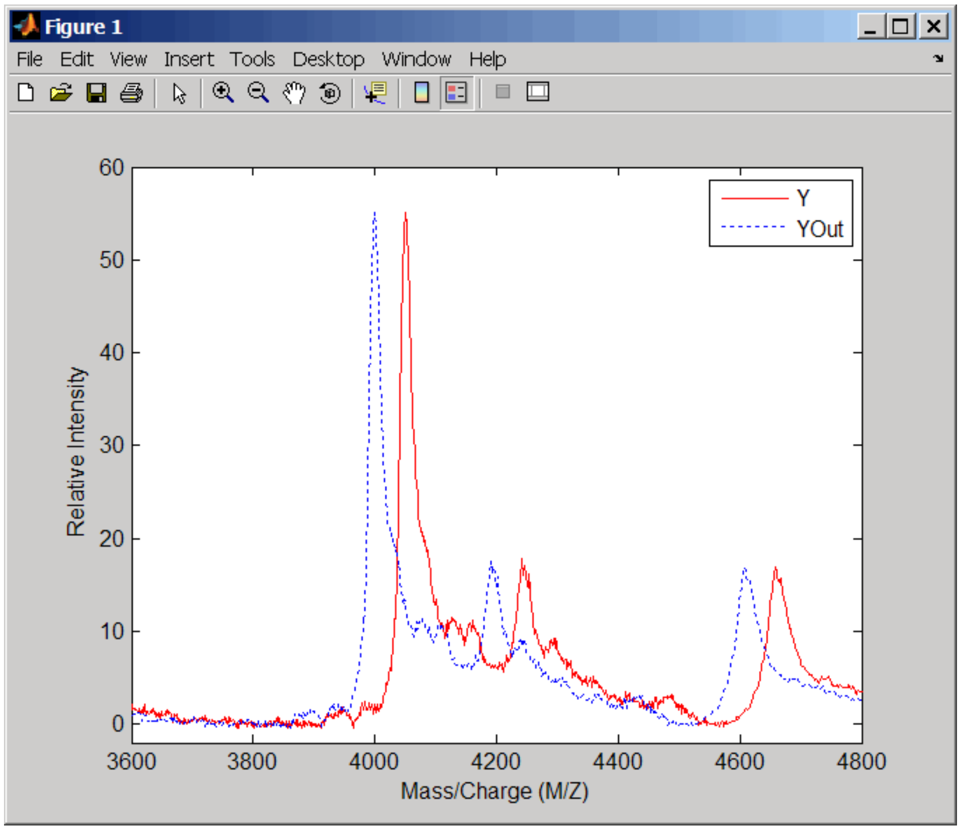


3 Shift a spectrum by the difference between RP, the known reference mass of 4000 m/z, and SP, the experimental mass of 4051.14 m/z.

```
RP = 4000;
SP = 4051.14;
YOut = interp1(MZ, Y, MZ-(RP-SP));
```

4 Plot the original spectrum in red and the shifted spectrum in blue and zoom in on the reference peak.

```
plot(MZ,Y,'r',MZ,YOut,'b:')
xlabel('Mass/Charge (M/Z)')
ylabel('Relative Intensity')
legend('Y','YOut')
axis([3600 4800 -2 60])
```



Version History

Introduced before R2006a

References

[1] Monchamp, P., Andrade-Cetto, L., Zhang, J.Y., and Henson, R. (2007) Signal Processing Methods for Mass Spectrometry. In Systems Bioinformatics: An Engineering Case-Based Approach, G. Alterovitz and M.F. Ramoni, eds. (Artech House Publishers).

See Also

`mspalign` | `msbackadj` | `msdotplot` | `msheatmap` | `mslowess` | `msnorm` | `mspeaks` | `msresample` | `msppresample` | `mssgolay` | `msviewer`

Topics

“Mass Spectrometry and Bioanalytics”
“Preprocessing Raw Mass Spectrometry Data”
“Visualizing and Preprocessing Hyphenated Mass Spectrometry Data Sets for Metabolite and Protein/Peptide Profiling”
“Differential Analysis of Complex Protein and Metabolite Mixtures using Liquid Chromatography/Mass Spectrometry (LC/MS)”

msbackadj

Correct baseline of signal with peaks

Syntax

```
yOut = msbackadj(X,Intensities)
yOut = msbackadj(X,Intensities,Name,Value)
```

Description

`yOut = msbackadj(X,Intensities)` adjusts the variable baseline of a raw signal with peaks by performing the following steps.

1. Estimate the baseline within multiple shifted windows of width 200 separation units.
2. Regress the varying baseline to the window points using a spline approximation.
3. Adjust the baseline of the peak signals supplied by the input `Intensities`.
4. Return the adjusted intensity values in the output matrix `yOut`.

`yOut = msbackadj(X,Intensities,Name,Value)` sets additional options specified by one or more name-value pair arguments. For example, `msbackadj(X,Intensities,'WindowSize',300)` sets the width of the shifting window to 300 separation units.

Examples

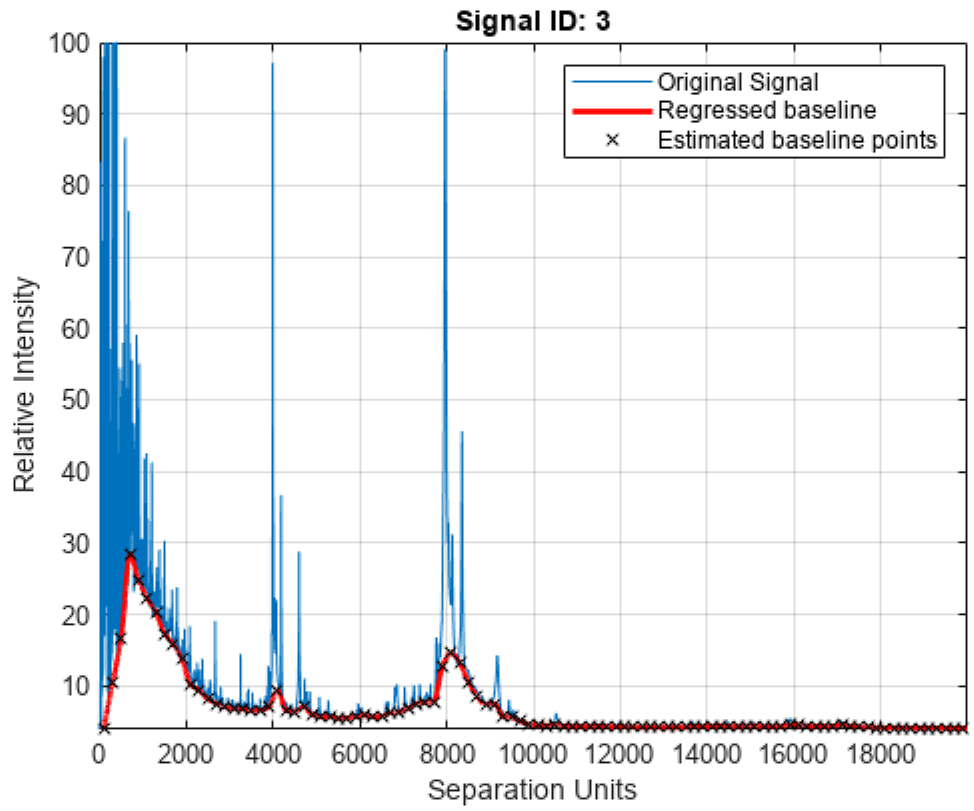
Adjust Baseline of Mass Spectrometry Data

Load a sample mass spec data including `MZ_lo_res`, a vector of m/z values, and `Y_lo_res`, a matrix of intensity values.

```
load sample_lo_res
```

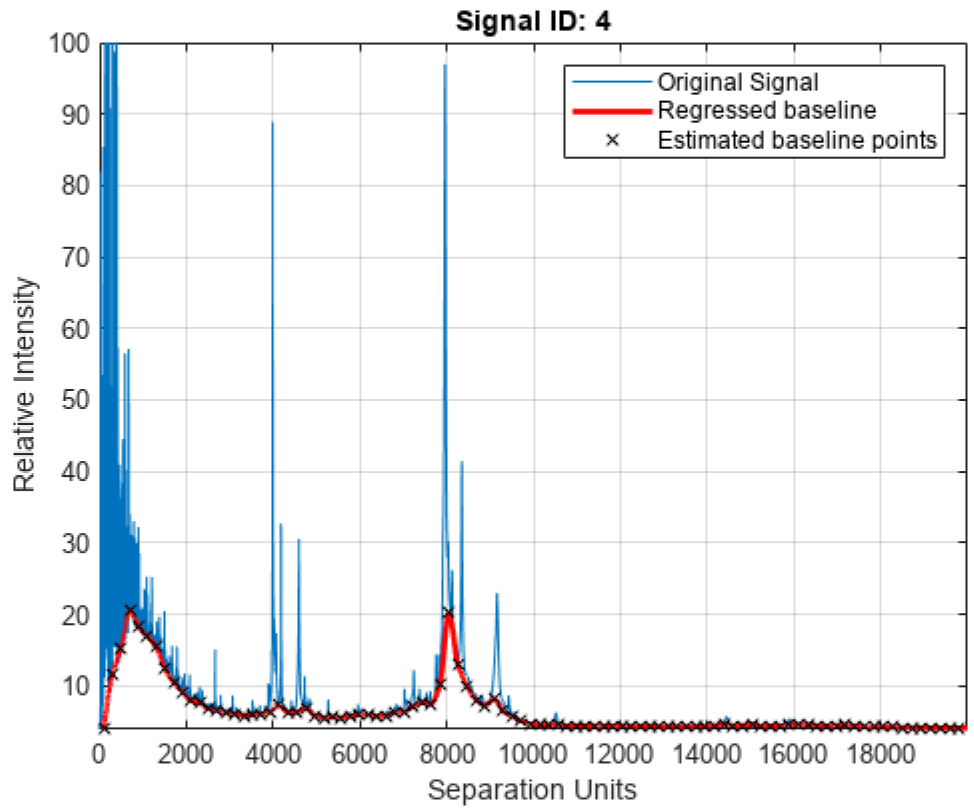
Adjust the baseline of a group of spectrograms and show only the third spectrum and its estimated background.

```
YB = msbackadj(MZ_lo_res,Y_lo_res,'ShowPlot',3);
```



Estimate the baseline for every spectrum in `Y_lo_res` using an anonymous function to describe an m/z dependent parameter. Then plot the estimated background for the fourth spectrum.

```
wf = @(mz) 200 + .001 .* mz;
msbackadj(MZ_lo_res,Y_lo_res,'StepSize',wf,'ShowPlot',4);
```



Input Arguments

X — Separation-unit values

vector

Separation-unit values for a set of signals with peaks, specified as a vector without any `Inf` or `NaN` values.

The number of elements in the vector equals the number of rows in `Intensities`. The separation unit can quantify wavelength, frequency, distance, time, or m/z ratio depending on the instrument that generates the signal data.

Data Types: `double`

`Intensities` — Intensity values for set of peaks

numeric matrix

Intensity values for a set of peaks that share separation-unit range, specified as a numeric matrix.

Each row corresponds to a separation-unit value, and each column corresponds to either a set of signals with peaks or a retention time. The number of rows equals the number of elements in X. The signal data can come from any separation technique, such as spectroscopy, NMR, electrophoresis, chromatography, or mass spectrometry.

Data Types: `double`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example:

`WindowSize` — Shifting window size
200 (default) | positive scalar | function handle

Shifting window size, specified as a positive scalar or function handle. By default, `msbackadj` estimates baseline points for windows with a width of 200 separation units.

If you specify a function handle, the function is evaluated at the respective `X` values and returns a variable width for the window. Specifying a function handle is useful when the resolution of the signal is dissimilar at different regions.

The result of `msbackadj` depends on the window size and step size. Define the parameters based on the width of your peaks in the signal and the presence of possible drifts. If you have wider peaks towards the end of the signal, consider using variable window sizes and/or step sizes.

Example: `'WindowSize',300`

Data Types: `double` | `function_handle`

`StepSize` — Step size for shifting window
200 (default) | positive scalar | function handle

Step size for the shifting window, specified as a positive scalar or function handle. By default, `msbackadj` estimates baseline points for windows placed every 200 separation units.

If you specify a function handle, the function is evaluated at the respective separation-unit values and returns the distance between adjacent windows.

Example: `'StepSize',150`

Data Types: `double` | `function_handle`

`RegressionMethod` — Method to regress window estimated points
`'pchip'` (default) | `'linear'` `'spline'`

Method to regress the window estimated points to a soft curve, specified as one of the following:

- `'pchip'` — Shape-preserving piecewise cubic interpolation. The interpolated value at a query point is based on a shape-preserving piecewise cubic interpolation of the values at neighboring grid points.
- `'linear'` — Linear interpolation. The interpolated value at a query point is based on linear interpolation of the values at neighboring grid points in each respective dimension.
- `'spline'` — Spline interpolation. The interpolated value at a query point is based on a cubic interpolation of the values at neighboring grid points in each respective dimension.

Example: `'RegressionMethod','linear'`

Data Types: `char` | `string`

`EstimationMethod` — Method to find likely baseline value
`'quantile'` (default) | `'em'`

Method to find likely baseline (background) value in every window, specified as one of the following:

- `'quantile'` — Quantile value is set to 10%.
- `'em'` — Every sample is the independent and identically distributed (i.i.d) draw of any of two normal distributed classes (background or peaks). Because the class label is hidden, the distributions are estimated with an Expectation-Maximization algorithm. The ultimate baseline value is the mean of the background class.

Example: `'EstimationMethod','em'`

Data Types: `char` | `string`

`SmoothMethod` — Method to smooth curve of estimated points
`'none'` (default) | `'lowess'` | `'loess'` | `'rlowess'` | `'rloess'`

Method to smooth the curve of estimated points, specified as one of the following:

- `'none'` — No smoothing.
- `'lowess'` — Linear fit.
- `'loess'` — Quadratic fit.
- `'rlowess'` — Robust linear fit.
- `'rloess'` — Robust quadratic fit.

Example: `'SmoothMethod','lowess'`

Data Types: `char` | `string`

`QuantileValue` — Quantile value
`0.10` (default) | positive scalar between `0` and `1`

Quantile value, specified as a positive scalar between `0` and `1`.

Example: `'QuantileValue',0.2`

Data Types: `double`

`PreserveHeights` — Flag to preserve height of tallest peak
`false` (default) | true

Flag to preserve the height of the tallest peak in the signal, specified as `true` or `false`. By default, peak heights are not preserved.

Example: `'PreserveHeights',true`

Data Types: `logical`

`ShowPlot` — Flag to plot regressed baseline, original signal, and estimated baseline points
`false` | `true` | positive integer

Flag to plot the regressed baseline, original signal, and estimated baseline points, specified as `true`, `false`, or a positive integer.

The default behavior is as follows:

- When you call `msbackadj` without an output argument, the plot is shown. Only the first signal from the input `Intensities` is plotted.
- When you call `msbackadj` with an output argument, the plot is not shown. But you can get the plot by also setting `'ShowPlot'` to `true`.

You can also specify an index to one of the signals (columns) in `Intensities` to show the corresponding plot of that signal.

Example: `'ShowPlot',5`

Data Types: `double` | `logical`

Output Arguments

`yOut` — Adjusted intensity values
matrix

Adjusted intensity values, returned as a matrix.

Version History

Introduced before R2006a

See Also

`mspalign` | `msdotplot` | `msalign` | `msheatmap` | `mslowess` | `msnorm` | `mspeaks` | `msresample` | `msppresample` | `mssgolay` | `msviewer`

Topics
“Mass Spectrometry and Bioanalytics”
“Preprocessing Raw Mass Spectrometry Data”
“Visualizing and Preprocessing Hyphenated Mass Spectrometry Data Sets for Metabolite and Protein/Peptide Profiling”
“Differential Analysis of Complex Protein and Metabolite Mixtures using Liquid Chromatography/Mass Spectrometry (LC/MS)”

msdotplot

Plot set of peak lists from LC/MS or GC/MS data set

Syntax

```
msdotplot(Peaklist, Times)
msdotplot(FigHandle, Peaklist, Times)
msdotplot(..., 'Quantile', QuantileValue)
PlotHandle = msdotplot(...)
```

Input Arguments

Peaklist	Cell array of peak lists, where each element is a two-column matrix with m/z values in the first column and ion intensity values in the second column. Each element corresponds to a spectrum or retention time. **Tip** You can use the `mzxml2peaks` function to create the *Peaklist* cell array.
Times	Vector of retention times associated with an LC/MS or GC/MS data set. The number of elements in *Times* equals the number of elements in the cell array *Peaklist*. **Tip** You can use the `mzxml2peaks` function to create the *Times* vector.
FigHandle	Handle to an open Figure window such as one created by the `msheatmap` function.
QuantileValue	Value that specifies a percentage. When peaks are ranked by intensity, only those that rank above this percentage are plotted. Choices are any value ≥ `0` and ≤ `1`. Default is `0`. For example, setting *QuantileValue* = `0` plots all peaks, and setting *QuantileValue* = `0.8` plots only the 20% most intense peaks.

Output Arguments

PlotHandle	Handle to the line series object (figure plot).

Description

`msdotplot(`*Peaklist*`,` *Times*`)` plots a set of peak lists from a liquid chromatography/mass spectrometry (LC/MS) or gas chromatography/mass spectrometry (GC/MS) data set represented by *Peaklist*, a cell array of peak lists, where each element is a two-column matrix with m/z values in the first column and ion intensity values in the second column, and *Times*, a vector of retention times associated with the spectra. *Peaklist* and *Times* have the same number of elements. The data is plotted into any existing figure generated by the `msheatmap` function; otherwise, the data is plotted into a new Figure window.

`msdotplot(`*`FigHandle`*`, `*`Peaklist`*`, `*`Times`*`)` plots the set of peak lists into the axes contained in an open Figure window with the handle *`FigHandle`*.

Tip This syntax is useful to overlay a dot plot on top of a heat map of mass spectrometry data created with the `msheatmap` function.

`msdotplot(..., 'Quantile', `*`QuantileValue`*`)` plots only the most intense peaks, specifically those in the percentage above the specified *`QuantileValue`*. Choices are any value `≥ 0` and `≤ 1`. Default is `0`. For example, setting *`QuantileValue`* = `0` plots all peaks, and setting *`QuantileValue`* = `0.8` plots only the 20% most intense peaks.

`PlotHandle`` = msdotplot(...)` returns a handle to the line series object (figure plot). You can use this handle as input to the `get` function to display a list of the plot's properties. You can use this handle as input to the `set` function to change the plot's properties, including showing and hiding points.

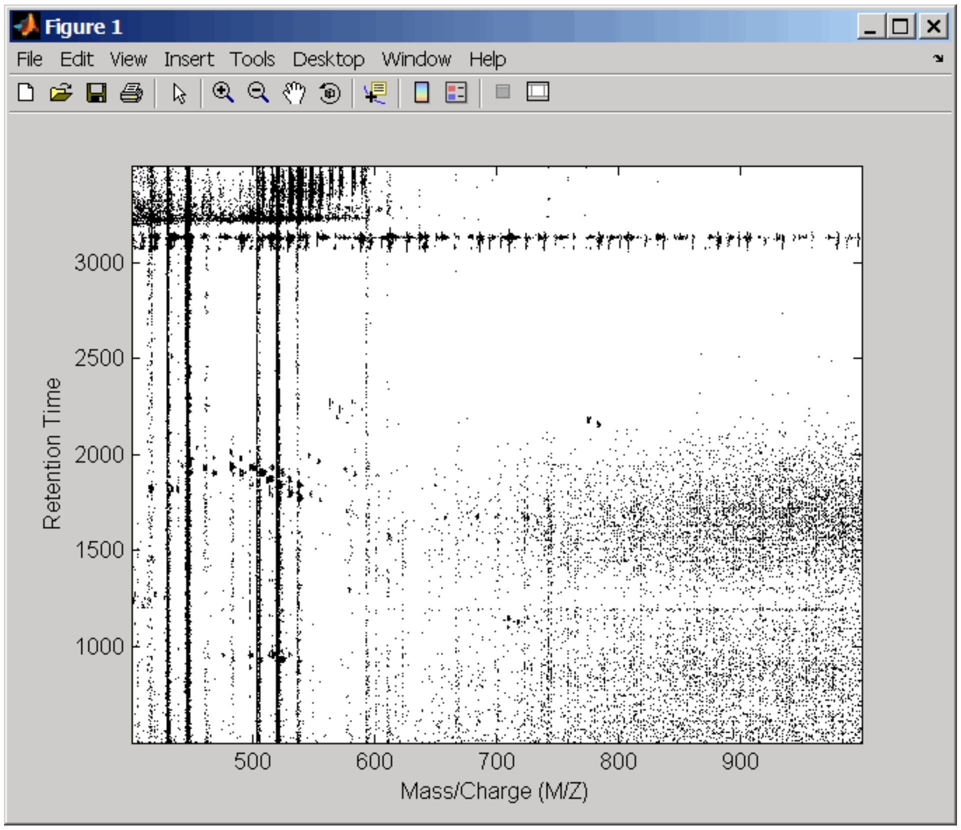
Examples

1 Load a MAT-file, included with the Bioinformatics Toolbox software, which contains LC/MS data variables, including `peaks` and `ret_time`. `peaks` is a cell array of peak lists, where each element is a two-column matrix of m/z values and ion intensity values, and each element corresponds to a spectrum or retention time. `ret_time` is a column vector of retention times associated with the LC/MS data set.

```
load lcmsdata
```

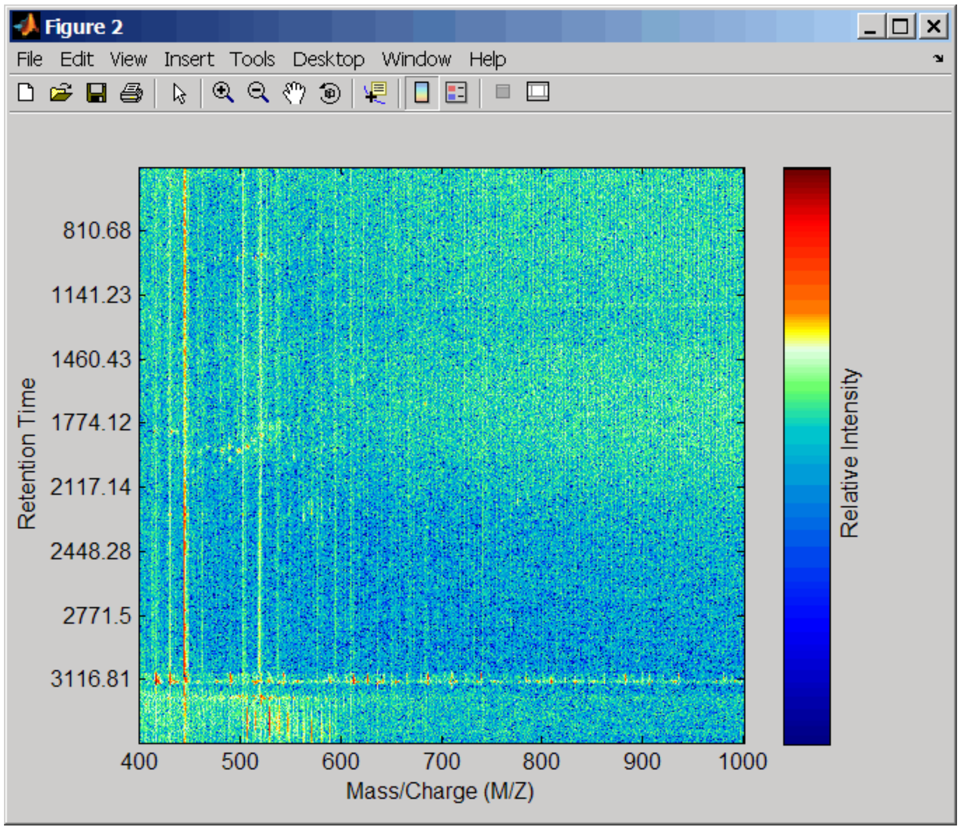
2 Create a dot plot with only the 5% most intense peaks.

```
msdotplot(ms_peaks,ret_time,'Quantile',0.95)
```



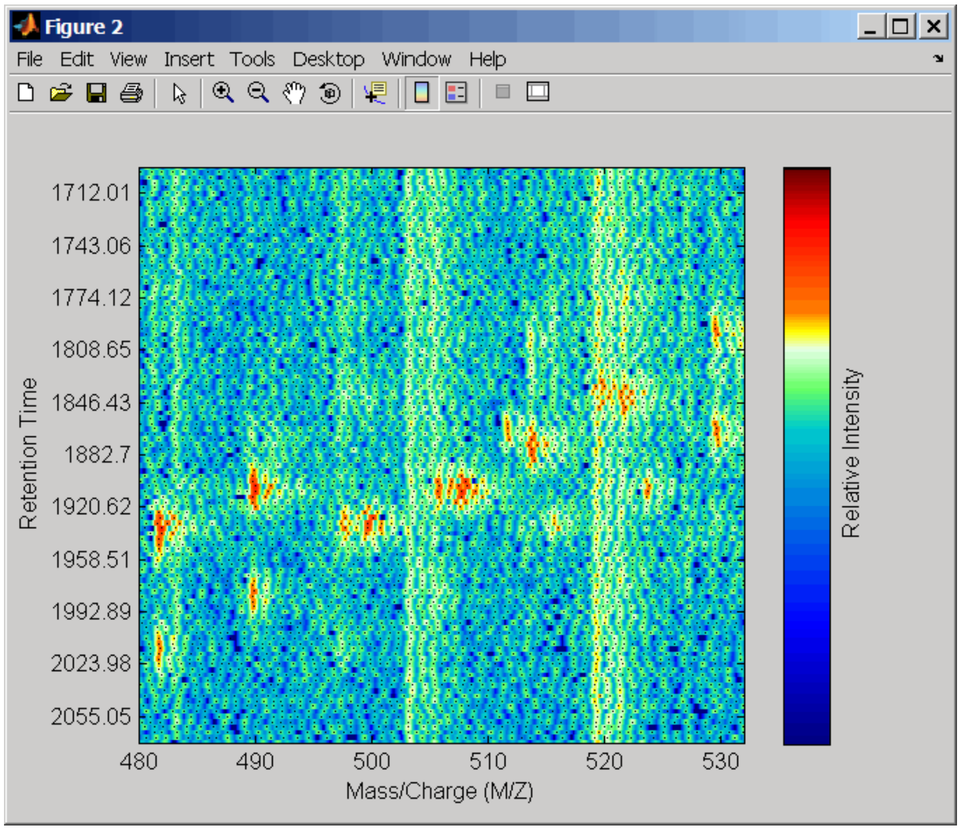
3 Resample the data, then create a heat map of the LC/MS data.

```
[MZ,Y] = msppresample(ms_peaks,5000);
msheatmap(MZ,ret_time,log(Y))
```



4 Overlay the dot plot on the heat map, and then zoom in to see the detail.

```
msdotplot(ms_peaks,ret_time)
axis([480 532 375 485])
```



Version History

Introduced in R2007a

See Also

`mspalign` | `msbackadj` | `msalign` | `msheatmap` | `mslowess` | `msnorm` | `mspeaks` | `msresample` | `msppresample` | `mssgolay` | `msviewer`

Topics

"Mass Spectrometry and Bioanalytics"
"Preprocessing Raw Mass Spectrometry Data"
"Visualizing and Preprocessing Hyphenated Mass Spectrometry Data Sets for Metabolite and Protein/Peptide Profiling"
"Differential Analysis of Complex Protein and Metabolite Mixtures using Liquid Chromatography/Mass Spectrometry (LC/MS)"

msheatmap

Create pseudocolor image of set of mass spectra

Syntax

```
msheatmap(MZ, Intensities)
msheatmap(MZ, Times, Intensities)

msheatmap(..., 'Midpoint', MidpointValue, ...)
msheatmap(..., 'Range', RangeValue, ...)
msheatmap(..., 'Markers', MarkersValue, ...)
msheatmap(..., 'SpecIdx', SpecIdxValue, ...)
msheatmap(..., 'Group', GroupValue, ...)
msheatmap(..., 'Resolution', ResolutionValue, ...)
```

Arguments

MZ	Column vector of common mass/charge (m/z) values for a set of spectra. The number of elements in the vector equals the number of rows in the matrix *Intensities*. **Note** You can use the `msppresample` function to create the *MZ* vector.
Times	Column vector of retention times associated with a liquid chromatography/mass spectrometry (LC/MS) or gas chromatography/mass spectrometry (GC/MS) data set. The number of elements in the vector equals the number of columns in the matrix *Intensities*. The retention times are used to label the *y*-axis of the heat map. **Tip** You can use the `mzxml2peaks` function to create the *Times* vector.
Intensities	Matrix of intensity values for a set of mass spectra that share the same m/z range. Each row corresponds to an m/z value, and each column corresponds to a spectrum or retention time. The number of rows equals the number of elements in vector *MZ*. The number of columns equals the number of elements in vector *Times*. **Note** You can use the `msppresample` function to create the *Intensities* matrix.

MidpointValue	Value specifying a quantile of the ion intensity values to fall below the midpoint of the colormap, meaning they do not represent peaks. `msheatmap` uses a custom colormap where cool colors represent nonpeak regions, white represents the midpoint, and warm colors represent peaks. Choices are any value `≥ 0` and `≤ 1`. Default is: • `0.99` — For LC/MS or GC/MS data or when input *T* is provided. This means that 1% of the pixels are warm colors and represent peaks. • `0.95` — For non-LC/MS or non-GC/MS data or when input *T* is not provided. This means that 5% of the pixels are warm colors and represent peaks. **Tip** You can also change the midpoint interactively after creating the heat map by right-clicking the color bar, selecting **Interactive Colormap Shift**, and then click-dragging the cursor vertically on the color bar. This technique is useful when comparing multiple heat maps.
RangeValue	1-by-2 vector specifying the m/z range for the *x*-axis of the heat map. *RangeValue* must be within `[min(`*MZ*`) max(`*MZ*`)]`. Default is the full range `[min(`*MZ*`) max(`*MZ*`)]`.
MarkersValue	Vector of m/z values to mark on the top horizontal axis of the heat map. Default is `[]`.
SpecIdxValue	Either of the following: • Vector of values with the same number of elements as columns (spectra) in the matrix *Intensities*. • Cell array of character vectors or string vector with the same number of elements as columns (spectra) in the matrix *Intensities*. Each value or character vector or string specifies a label for the corresponding spectrum. These values or character vectors or strings are used to label the *y*-axis of the heat map. **Note** If input *Times* is provided, it is assumed that *Intensities* contains LC/MS or GC/MS data, and *SpecIdxValue* is ignored.
GroupValue	Either of the following: • Vector of values with the same number of elements as rows in the matrix *Intensities* • Cell array of character vectors or string vector with the same number of elements as rows (spectra) in the matrix *Intensities* Each value, character vector, or string specifies a group to which the corresponding spectrum belongs. The spectra are sorted and combined into groups along the *y*-axis in the heat map. **Note** If input *Times* is provided, it is assumed that *Intensities* contains LC/MS or GC/MS data, and *GroupValue* is ignored.

ResolutionValue	Value specifying the horizontal resolution of the heat map image. Increase this value to enhance details. Decrease this value to reduce memory usage. Default is: • `0.5` — When *MZ* contains > 2,500 elements. • `0.05` — When *MZ* contains <= 2,500 elements.

Description

`msheatmap(`*`MZ`*`, `*`Intensities`*`)` displays a pseudocolor heat map image of the intensities for the spectra in matrix *`Intensities`*.

`msheatmap(`*`MZ`*`, `*`Times`*`, `*`Intensities`*`)` displays a pseudocolor heat map image of the intensities for the spectra in matrix *`Intensities`*, using the retention times in vector *`Times`* to label the *y*-axis.

`msheatmap(..., '`*`PropertyName`*`', `*`PropertyValue`*`, ...)` calls `msheatmap` with optional properties that use property name/property value pairs. You can specify one or more properties in any order. Each *`PropertyName`* must be enclosed in single quotation marks and is case insensitive. These property name/property value pairs are as follows:

`msheatmap(..., 'Midpoint', `*`MidpointValue`*`, ...)` specifies a quantile of the ion intensity values to fall below the midpoint of the colormap, meaning they do not represent peaks. `msheatmap` uses a custom colormap where cool colors represent nonpeak regions, white represents the midpoint, and warm colors represent peaks. Choices are any value between `0` and `1`. Default is:

- `0.99` — For LC/MS or GC/MS data or when input *T* is provided. This means that 1% of the pixels are warm colors and represent peaks.
- `0.95` — For non-LC/MS or non-GC/MS data or when input *T* is not provided. This means that 5% of the pixels are warm colors and represent peaks.

Tip You can also change the midpoint interactively after creating the heat map by right-clicking the color bar, selecting **Interactive Colormap Shift**, then click-dragging the cursor vertically on the color bar. This technique is useful when comparing multiple heat maps.

`msheatmap(..., 'Range', `*`RangeValue`*`, ...)` specifies the m/z range for the *x*-axis of the heat map. *`RangeValue`* is a 1-by-2 vector that must be within `[min(`*`MZ`*`) max(`*`MZ`*`)]`. Default is the full range `[min(`*`MZ`*`) max(`*`MZ`*`)]`.

`msheatmap(..., 'Markers', `*`MarkersValue`*`, ...)` places markers along the top horizontal axis of the heat map for the m/z values specified in the vector *`MarkersValue`*. Default is `[]`.

`msheatmap(..., 'SpecIdx', `*`SpecIdxValue`*`, ...)` labels the spectra along the *y*-axis in the heat map. The labels are specified by *`SpecIdxValue`*, a vector of values, cell array of character vectors, or string vector. The number of values or character vectors or strings is the same as the number of columns (spectra) in the matrix *`Intensities`*. Each value or character vector or string specifies a label for the corresponding spectrum.

`msheatmap(..., 'Group', `*`GroupValue`*`, ...)` sorts and combines spectra into groups along the *y*-axis in the heat map. The groups are specified by *`GroupValue`*, a vector of values, cell array of character vectors, or string vector. The number of values, character vectors, or strings is the same as the number of rows in the matrix *`Intensities`*. Each value or character vector or string specifies a group to which the corresponding spectrum belongs. Default is `[1:numSpectra]`.

`msheatmap(..., 'Resolution', ResolutionValue, ...)` specifies the horizontal resolution of the heat map image. Increase this value to enhance details. Decrease this value to reduce memory usage. Default is:

- 0.5 — When *MZ* contains > 2,500 elements.
- 0.05 — When *MZ* contains <= 2,500 elements.

Examples

Example 1.26. SELDI-TOF Data

1 Load SELDI-TOF sample data.

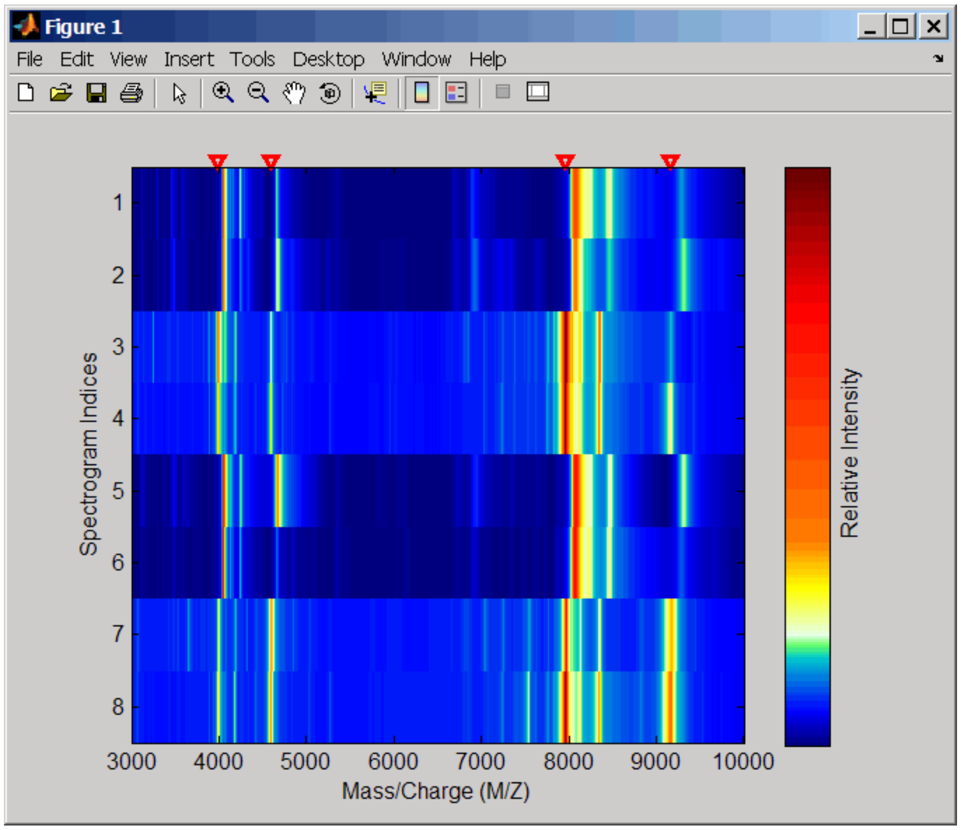
```
load sample_lo_res
```

2 Create a vector of four m/z values to mark along the top horizontal axis of the heat map.

```
M = [3991.4 4598 7964 9160];
```

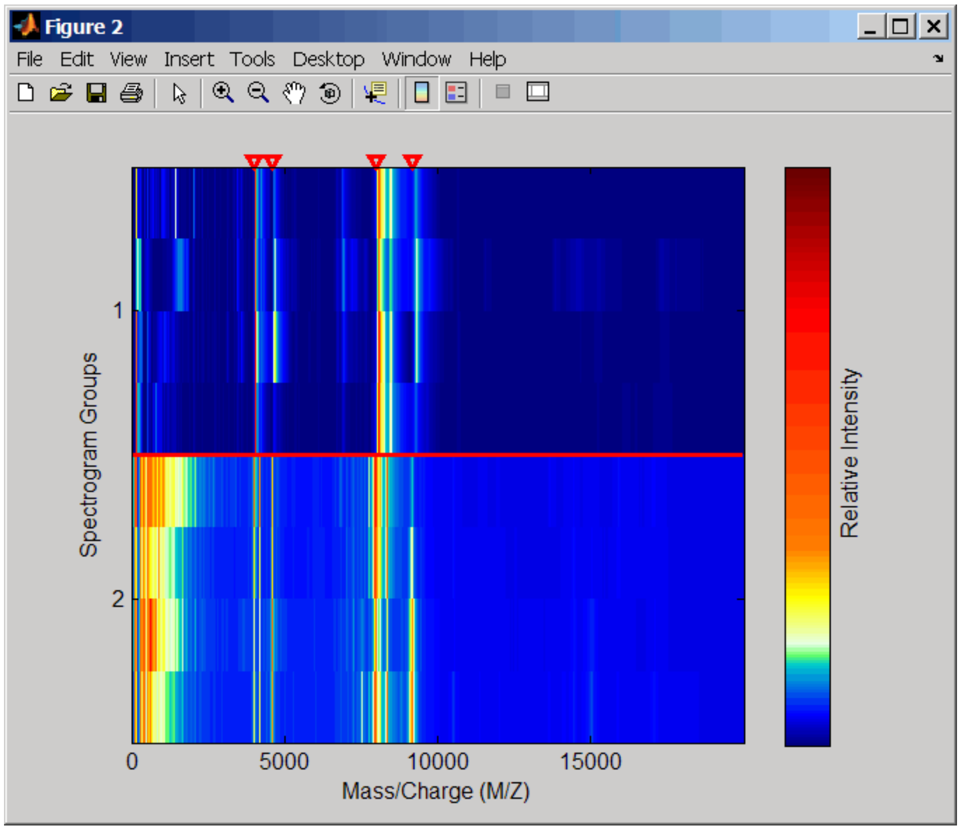
3 Display the heat map with m/z markers and a limited m/z range.

```
msheatmap(MZ_lo_res,Y_lo_res,'markers',M,'range',[3000 10000])
```



4 Display the heat map again grouping each spectrum into one of two groups.

```
TwoGroups = [1 1 2 2 1 1 2 2];
msheatmap(MZ_lo_res,Y_lo_res,'markers',M,'group',TwoGroups)
```



Example 1.27. Liquid Chromatography/Mass Spectrometry (LC/MS) Data

1 Load LC/MS sample data.

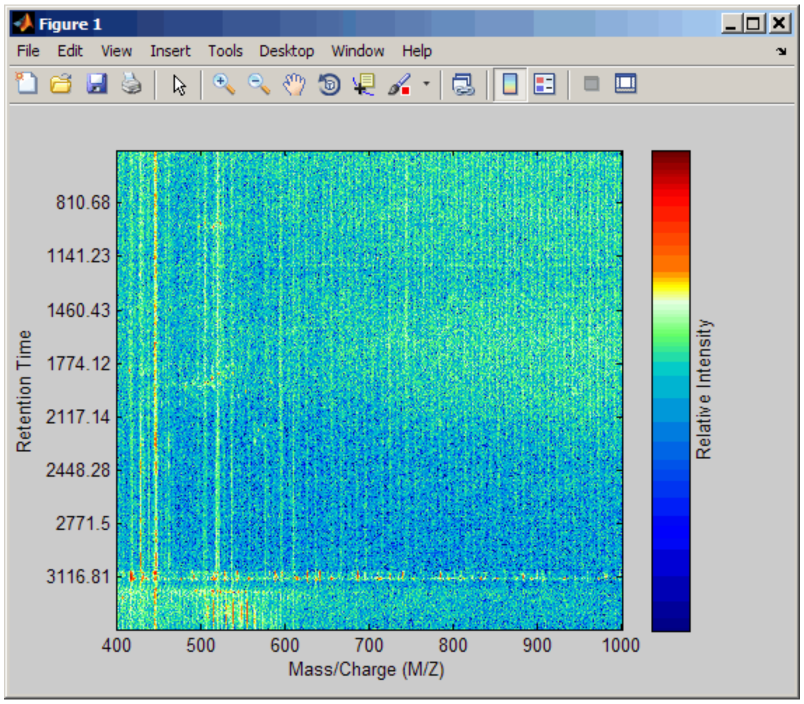
```
load lcmsdata
```

2 Resample the peak lists to create a vector of m/z values and a matrix of intensity values.

```
[MZ, Intensities] = msppresample(ms_peaks, 5000);
```

3 Display the heat map showing mass spectra at different retention times.

```
msheatmap(MZ, ret_time, log(Intensities))
```



Version History

Introduced before R2006a

See Also

`mspalign` | `msbackadj` | `msdotplot` | `msalign` | `mslowess` | `msnorm` | `mspeaks` | `msresample` | `msppresample` | `mssgolay` | `msviewer`

Topics

"Mass Spectrometry and Bioanalytics"
"Preprocessing Raw Mass Spectrometry Data"
"Visualizing and Preprocessing Hyphenated Mass Spectrometry Data Sets for Metabolite and Protein/Peptide Profiling"
"Differential Analysis of Complex Protein and Metabolite Mixtures using Liquid Chromatography/Mass Spectrometry (LC/MS)"

mslowess

Smooth signal with peaks using nonparametric method

Syntax

```
Yout = mslowess(X, Intensities)

mslowess(..., 'Order', OrderValue, ...)
mslowess(..., 'Span', SpanValue, ...)
mslowess(..., 'Kernel', KernelValue, ...)
mslowess(..., 'RobustIterations', RobustIterationsValue, ...)
mslowess(..., 'ShowPlot', ShowPlotValue, ...)
```

Arguments

X	Vector of separation-unit values for a set of signals with peaks. The number of elements in the vector equals the number of rows in the matrix *Intensities*. The separation unit can quantify wavelength, frequency, distance, time, or m/z depending on the instrument that generates the signal data.
Intensities	Matrix of intensity values for a set of peaks that share the same separation-unit range. Each row corresponds to a separation-unit value, and each column corresponds to either a set of signals with peaks or a retention time. The number of rows equals the number of elements in vector *X*.

Description

Tip Use the following syntaxes with data from any separation technique that produces signal data, such as spectroscopy, NMR, electrophoresis, chromatography, or mass spectrometry.

`Yout = mslowess(X, Intensities)` smooths raw noisy signal data, *Intensities*, using a locally weighted linear regression (Lowess) method with a default span of `10` samples.

Note `mslowess` assumes the input vector, *X*, may not have uniformly spaced separation units. Therefore, the sliding window for smoothing is centered using the closest samples in terms of the *X* value and not in terms of the *X* index.

Note When the input vector, *X*, does not have repeated values or NaN values, the algorithm is approximately twice as fast.

`mslowess(X, Intensities, ...'PropertyName', PropertyValue, ...)` calls `mslowess` with optional properties that use property name/property value pairs. You can specify one or more properties in any order. Each *PropertyName* must be enclosed in single quotation marks and is case insensitive. These property name/property value pairs are as follows:

`mslowess(..., 'Order', OrderValue, ...)` specifies the order (*OrderValue*) of the Lowess smoother. Enter `1` (linear polynomial fit or Lowess), `2` (quadratic polynomial fit or Loess), or `0` (equivalent to a weighted local mean estimator and presumably faster because only a mean computation is performed instead of a least-squares regression). The default value is `1`.

Note Curve Fitting Toolbox software also refers to Lowess smoothing of order 2 as Loess smoothing.

`mslowess(..., 'Span', SpanValue, ...)` specifies the window size for the smoothing kernel. If *SpanValue* is greater than `1`, the window is equal to *SpanValue* number of samples independent of the separation-unit vector, *X*. The default value is `10` samples. Higher values will smooth the signal more at the expense of computation time. If *SpanValue* is less than `1`, the window size is taken to be a fraction of the number of points in the data. For example, when *SpanValue* is `0.005`, the window size is equal to `0.50`% of the number of points in *X*.

`mslowess(..., 'Kernel', KernelValue, ...)` selects the function specified by *KernelValue* for weighting the observed intensities. Samples close to the separation-unit location being smoothed have the most weight in determining the estimate. *KernelValue* can be any of the following character vectors (or strings):

- `'tricubic'` (default) — `(1 - (dist/dmax).^3).^3`
- `'gaussian'` — `exp(-(2*dist/dmax).^2)`
- `'linear'` — `1-dist/dmax`

`mslowess(..., 'RobustIterations', RobustIterationsValue, ...)` specifies the number of iterations (*RobustValue*) for a robust fit. If *RobustIterationsValue* is `0` (default), no robust fit is performed. For robust smoothing, small residual values at every span are outweighed to improve the new estimate. `1` or `2` robust iterations are usually adequate, while larger values might be computationally expensive.

Note For an *X* vector that has uniformly spaced separation units, a nonrobust smoothing with *OrderValue* equal to `0` is equivalent to filtering the signal with the kernel vector.

`mslowess(..., 'ShowPlot', ShowPlotValue, ...)` plots the smoothed signal over the original signal. When you call `mslowess` without output arguments, the signals are plotted unless *ShowPlotValue* is `false`. When *ShowPlotValue* is `true`, only the first signal in *Intensities* is plotted. *ShowPlotValue* can also contain an index to one of the signals in *Intensities*.

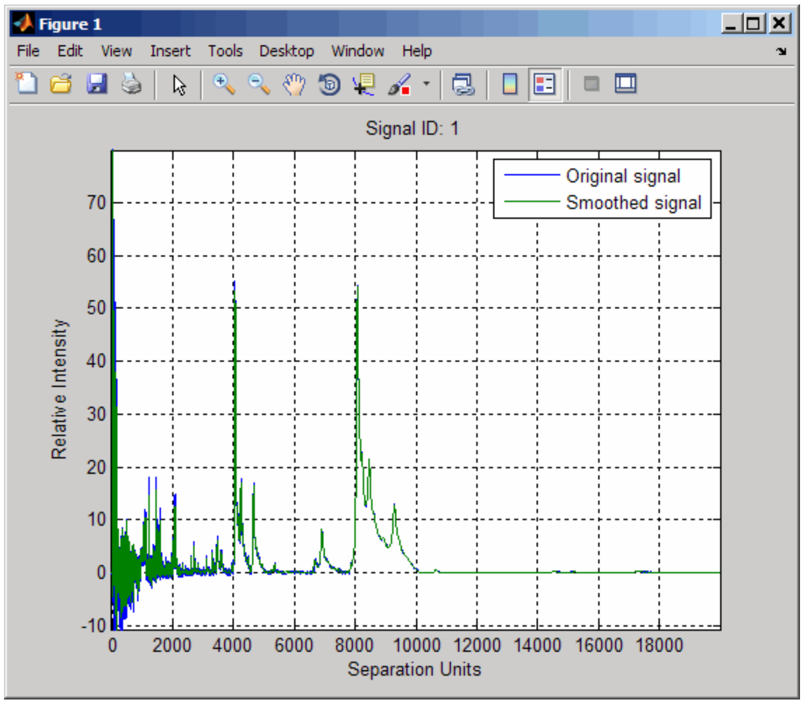
Examples

1. Load a MAT-file, included with the Bioinformatics Toolbox software, that contains some sample data.

```
load sample_lo_res
```

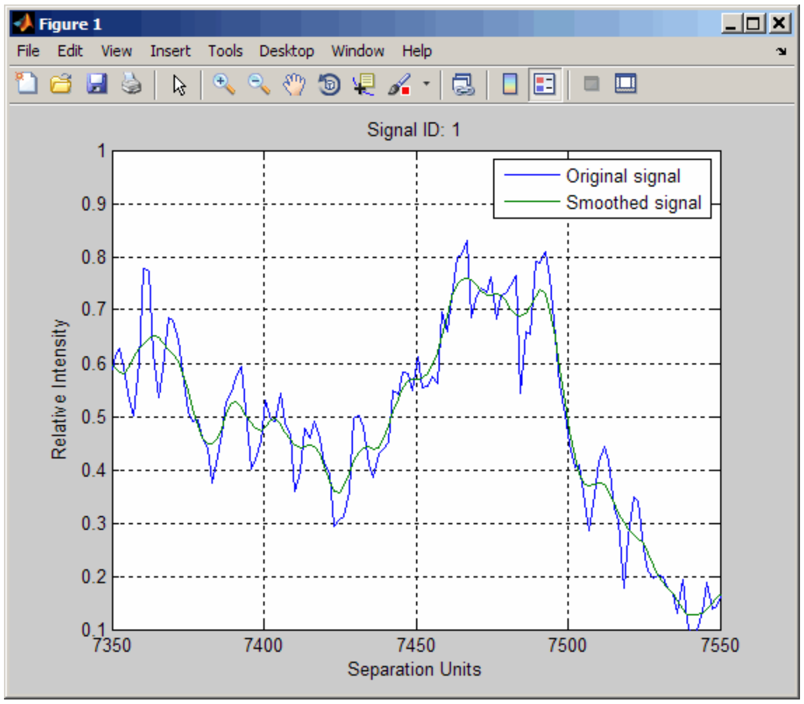
2. Smooth the spectra and draw a figure of the first spectrum with original and smoothed signals.

```
YS = mslowess(MZ_lo_res,Y_lo_res,'Showplot',true);
```



3 Zoom in on a region of the figure to see the difference in the original and smoothed signals.

```
axis([7350 7550 0.1 1.0])
```



Version History

Introduced before R2006a

See Also

`mspalign` | `msbackadj` | `msdotplot` | `msalign` | `msheatmap` | `msnorm` | `mspeaks` | `msresample` | `msppresample` | `mssgolay` | `msviewer`

Topics

“Mass Spectrometry and Bioanalytics”
“Preprocessing Raw Mass Spectrometry Data”
“Visualizing and Preprocessing Hyphenated Mass Spectrometry Data Sets for Metabolite and Protein/Peptide Profiling”
“Differential Analysis of Complex Protein and Metabolite Mixtures using Liquid Chromatography/Mass Spectrometry (LC/MS)”

msnorm

Normalize set of signals with peaks

Syntax

```
yOut = msnorm(X,Intensities)
[yOut,normParams] = msnorm(X,Intensities)
yOut = msnorm(X,Intensities,NormParameters)
[ ___ ] = msnorm(X,Intensities,Name,Value)
```

Description

`yOut = msnorm(X,Intensities)` normalizes a group of signals with peaks by standardizing the area under the curve (AUC) to the group median and returns the normalized data `yOut`.

`[yOut,normParams] = msnorm(X,Intensities)` also returns the normalization parameters `normParams`, which you can use to normalize another group of signals.

`yOut = msnorm(X,Intensities,NormParameters)` uses the parameter information `NormParameters` from a previous normalization to normalize a new set of signals. The function uses the same parameters to select the separation-unit positions and output scale from the previous normalization. If you specified a consensus proportion using the `'Consensus'` name-value pair argument in the previous normalization, the function selects no new separation-unit positions and performs normalization using the same separation-unit positions.

`[ ___ ] = msnorm(X,Intensities,Name,Value)` uses additional options specified by one or more name-value pair arguments and returns any of the output arguments in previous syntaxes. For example, `out = msnorm(X,Y,'Quantile',[0.9 1])` sets the lower (0.9) and upper (1) quantile limit to use only the largest 10% of intensities in each signal to compute the AUC.

Examples

AUC Normalization

This example shows how to normalize the area under the curve of every mass spectrum from the mass spec data.

Load a MAT-file, included with the Bioinformatics Toolbox™ software, that contains sample mass spec data, including MZ_lo_res, a vector of m/z values, and Y_lo_res, a matrix of intensity values.

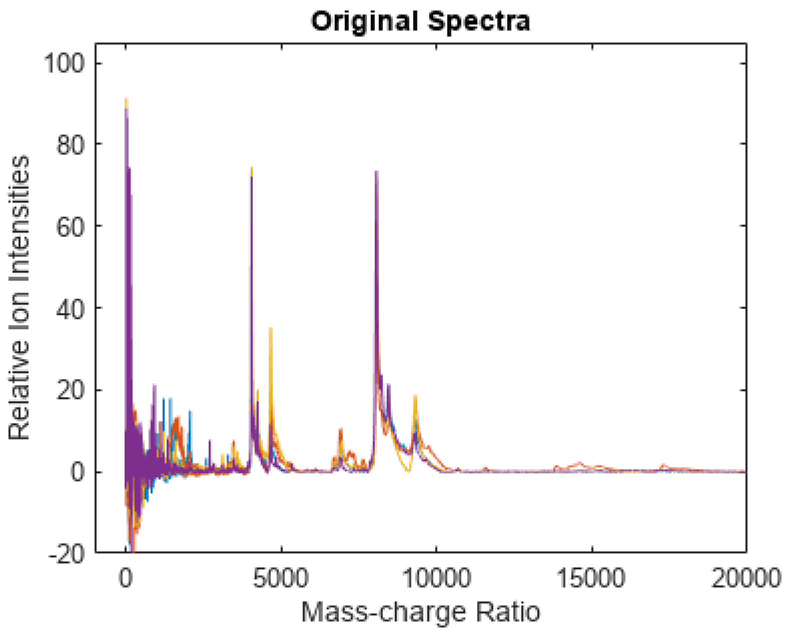
```
load sample_lo_res
```

Create a subset (four signals) of the data.

```
MZ = MZ_lo_res;
Y = Y_lo_res(:,[1 2 5 6]);
```

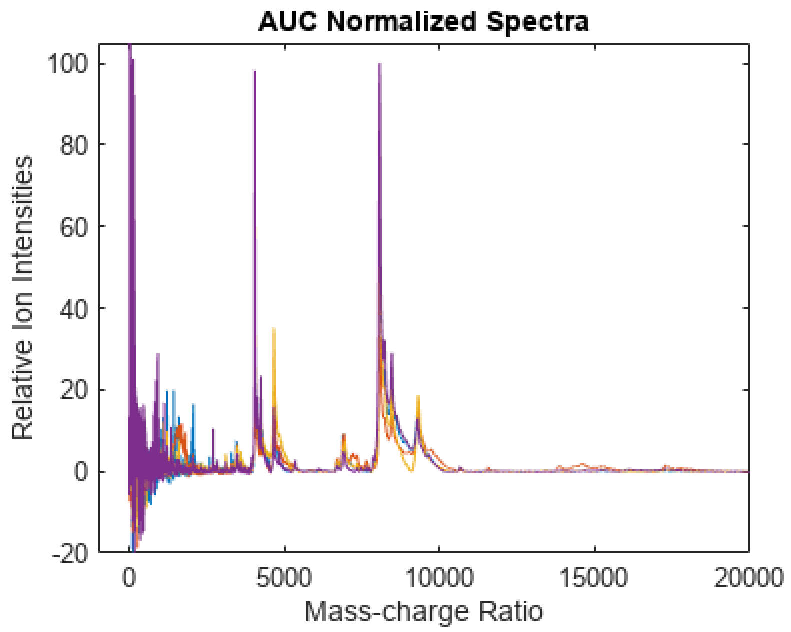
Plot the four spectra.

```
plot(MZ, Y)
axis([-1000 20000 -20 105])
xlabel('Mass-charge Ratio')
ylabel('Relative Ion Intensities')
title('Original Spectra')
```



Normalize the area under the curve (AUC) of every spectrum to the median, eliminating low-mass (m/z < 1,000) noise, and post-rescaling such that the maximum intensity is 100. Plot the four spectra.

```
Y1 = msnorm(MZ,Y,'Limits',[1000 inf],'Max',100);
plot(MZ, Y1)
axis([-1000 20000 -20 105])
xlabel('Mass-charge Ratio')
ylabel('Relative Ion Intensities')
title('AUC Normalized Spectra')
```



Maximum Intensity Normalization

This example shows how to normalize the ion intensity of every spectrum from the mass spec data.

Load a MAT-file, included with the Bioinformatics Toolbox™ software, that contains sample mass spec data, including MZ_lo_res, a vector of m/z values, and Y_lo_res, a matrix of intensity values.

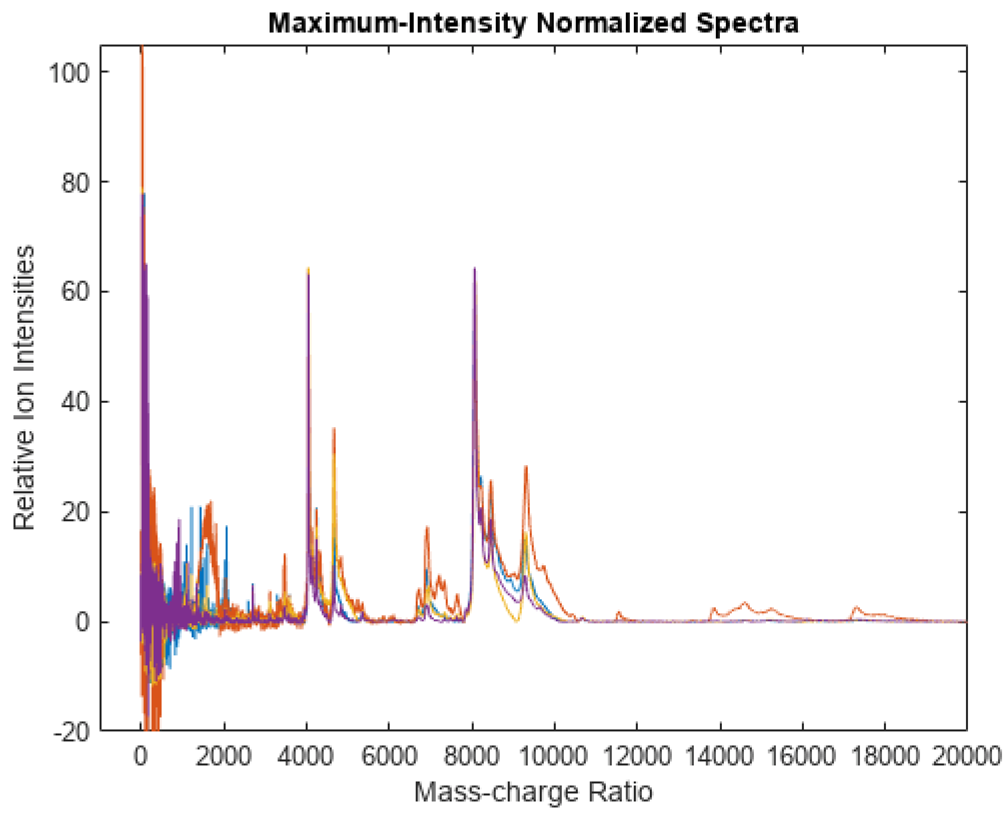
```
load sample_lo_res
```

Create a subset (four signals) of the data.

```
MZ = MZ_lo_res;
Y = Y_lo_res(:,[1 2 5 6]);
```

Normalize the ion intensity of every spectrum to the maximum intensity of the single highest peak from any of the spectra in the range above 1000 m/z. Plot the four spectra.

```
Y2 = msnorm(MZ,Y,'QUANTILE', [1 1],'LIMITS',[1000 inf]);
plot(MZ, Y2)
axis([-1000 20000 -20 105])
xlabel('Mass-charge Ratio')
ylabel('Relative Ion Intensities')
title('Maximum-Intensity Normalized Spectra')
```



Quantile Normalization

This example shows how to perform quantile normalization for mass spec data.

Load a MAT-file, included with the Bioinformatics Toolbox™ software, that contains sample mass spec data, including MZ_lo_res, a vector of m/z values, and Y_lo_res, a matrix of intensity values.

```
load sample_lo_res
```

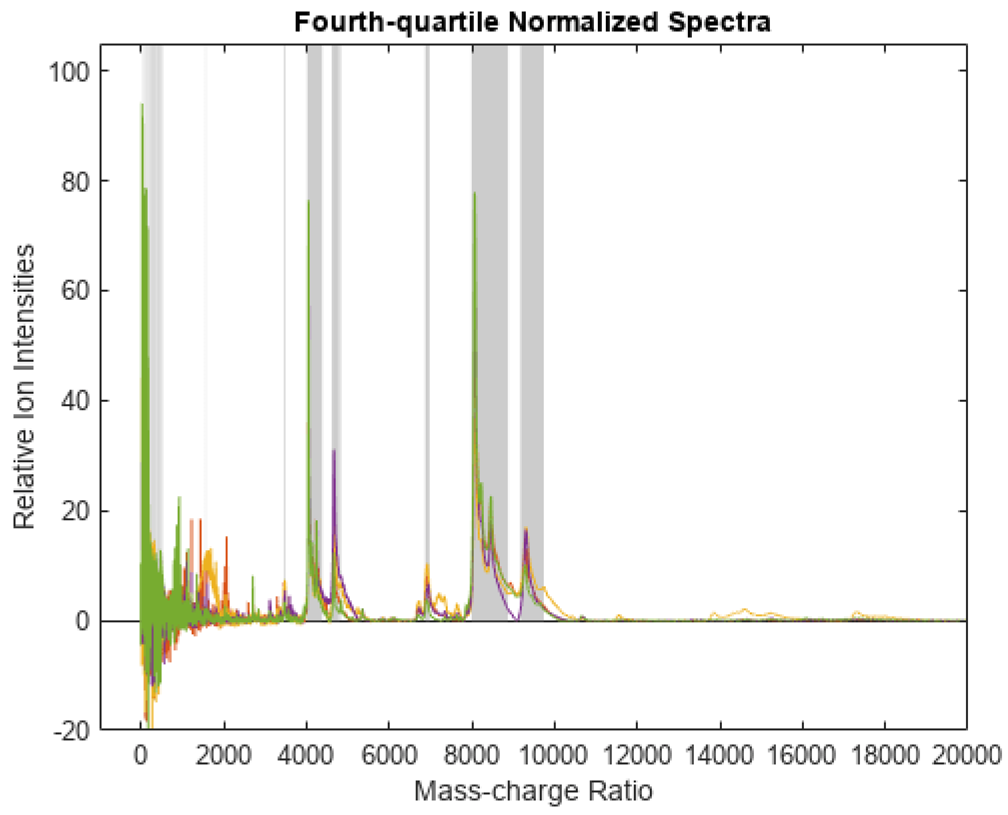
Create a subset (four signals) of the data.

```
MZ = MZ_lo_res;
Y = Y_lo_res(:,[1 2 5 6]);
```

Normalize using the data in the m/z regions where the intensities are within the fourth quartile in at least 90% of the spectrograms. Note that you can use the normalization parameters in the second output to normalize another set of data in the same m/z regions. Plot the four spectra.

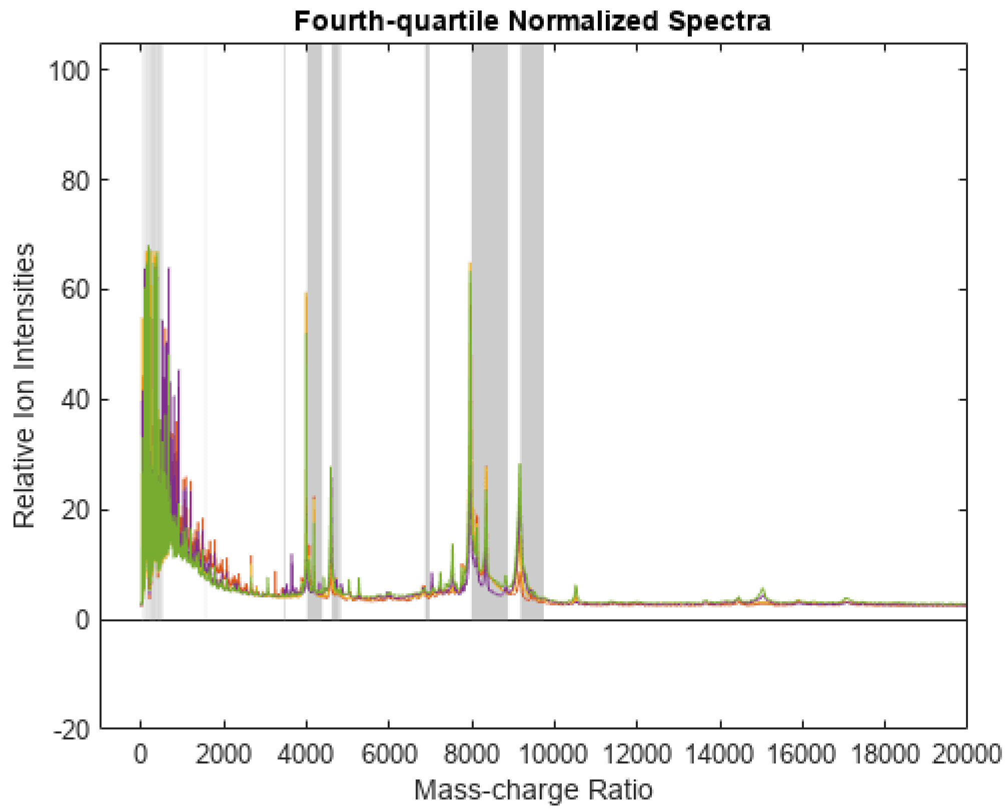
```
[Y3,S] = msnorm(MZ,Y,'Quantile',[0.75 1],'Consensus',0.9);
area(MZ,S.Xh.*1000,'LineStyle','None','FaceColor',[.8 .8 .8])
hold on
plot(MZ, Y3)
hold off
axis([-1000 20000 -20 105])
xlabel('Mass-charge Ratio')
```

```
ylabel('Relative Ion Intensities')
title('Fourth-quartile Normalized Spectra')
```



Use the normalization parameters in the second output of the previous step to normalize a different subset of data (four signals) using the data in the same m/z regions as the previous data set. Plot the four spectra.

```
Y4 = msnorm(MZ,Y_lo_res(:,[3 4 7 8]),S);

area(MZ,S.Xh.*1000,'LineStyle','None','FaceColor',[.8 .8 .8])
hold on
plot(MZ, Y4)
hold off
axis([-1000 20000 -20 105])
xlabel('Mass-charge Ratio')
ylabel('Relative Ion Intensities')
title('Fourth-quartile Normalized Spectra')
```



Input Arguments

X — Vector of separation-unit values for signals with peaks
vector

Vector of separation-unit values for a set of signals with peaks, specified as a vector.

Data Types: `double`

`Intensities` — Intensity values for set of peaks
matrix

Intensity values for a set of peaks that share the same separation-unit range, specified as a matrix. Each row is a separation-unit value and each column is either a set of signals with peaks or a retention time. The number of rows in `Intensities` must equal the number of elements in the input vector `X`.

Data Types: `double`

`NormParameters` — Normalization parameters
structure

Normalization parameters to normalize another group of signals, specified as a structure. `NormParameters` is a structure returned by `msnorm` from a previous normalization call.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `out = msnorm(X,Y,'Quantile',[0.9 1])` sets the lower (0.9) and upper (1) quantile limit to use only the largest 10% of intensities in each signal to compute the AUC.

`Quantile` — Quantile limits to reduce separation-unit values in `X`

`[0 1]` (default) | 1-by-2 vector | scalar between `0` and `1`

Quantile limits to reduce the set of separation-unit values in `X`, specified as a `1`-by-`2` vector or a scalar between `0` and `1`.

If you specify a vector, the first element is the lower limit and the second element is the upper limit. For example, `[0.9 1]` means that the function uses only the largest 10% of intensities in each signal to compute the AUC. The default value `[0 1]` means that the function uses the whole AUC, instead of limiting the intensities to a particular quantile.

If you specify a scalar value, it represents the lower quantile limit. The upper quantile limit is automatically set to `1`.

Example: `'Quantile',[0.8 1]`

Data Types: `double`

`Limits` — Separation-unit range to pick normalization points

`[min(X) max(X)]` (default) | 1-by-2 vector

Separation-unit range to pick normalization points, specified as a `1`-by-`2` vector. The default value `[min(X) max(X)]` selects all available points from `X`. If you specify a lower or upper limit as a value that is not within the available range `[min(X) max(X)]`, the function sets the lower limit to `min(X)` and the upper limit to `max(X)`.

This parameter is useful to eliminate noise from the AUC calculation. For instance, you can exclude the matrix noise that appears in the low-mass region (m/z values less than `1000`) of a SELDI mass spectrometer by setting the limit to `[1000 max(X)]`.

Example: `'Limits',[900 max(X)]`

Data Types: `double`

`Consensus` — Minimal percentage of intensity values within quantile limits

scalar between `0` and `1`

Minimal percentage of intensity values within the quantile limits that a separation-unit position must have to be included in the AUC calculation, specified as a scalar between `0` and `1`. The same separation-unit positions are then used to normalize all the signals. Use this parameter to eliminate low-intensity peaks and noise from the normalization.

For instance, to select m/z regions whose intensities are within the third quantile in at least 90% of the spectrograms, set `'Quantile'` and `'Consensus'` as follows: `yOut = msnorm(MZ,Y,'Quantile',[0.5 0.75],'Consensus',0.9)`.

Example: `'Consensus',0.8`

Data Types: `double`

`Method` — Method for normalizing AUC of every signal
`'Median'` (default) | `'Mean'`

Method for normalizing the AUC of every signal, specified as `'Median'` or `'Mean'`.

Example: `'Method','Mean'`

Data Types: `char` | `string`

`Max` — Overall maximum intensity to scale to after normalization
scalar

Overall maximum intensity to scale to after normalizing each signal individually, specified as a scalar. If you do not specify this parameter, no postscaling is performed.

Note If you specify this value and also set `'Quantile'` to `[1 1]`, then a single point (peak height of the tallest peak) is normalized to the specified maximum value.

Example: `'Max'`

Data Types: `double`

Output Arguments

`yOut` — Normalized intensity values
matrix

Normalized intensity values, returned as a matrix.

`normParams` — Normalization parameters
structure

Normalization parameters that you can use to normalize another group of signals, returned as a structure.

Version History

Introduced before R2006a

See Also

`mspalign` | `msbackadj` | `msdotplot` | `msalign` | `msheatmap` | `mslowess` | `mspeaks` | `msresample` | `msppresample` | `mssgolay` | `msviewer`

Topics
"Mass Spectrometry and Bioanalytics"
"Preprocessing Raw Mass Spectrometry Data"
"Visualizing and Preprocessing Hyphenated Mass Spectrometry Data Sets for Metabolite and Protein/Peptide Profiling"

"Differential Analysis of Complex Protein and Metabolite Mixtures using Liquid Chromatography/ Mass Spectrometry (LC/MS)"

mspalign

Align mass spectra from multiple peak lists from LC/MS or GC/MS data set

Syntax

```
[CMZ, AlignedPeaks] = mspalign(Peaklist)

[CMZ, AlignedPeaks] = mspalign(Peaklist, ...'Quantile', QuantileValue, ...)
[CMZ, AlignedPeaks] = mspalign(Peaklist, ...'EstimationMethod',
EstimationMethodValue, ...)
[CMZ, AlignedPeaks] = mspalign(Peaklist, ...'CorrectionMethod',
CorrectionMethodValue, ...)
[CMZ, AlignedPeaks] = mspalign(Peaklist, ...'ShowEstimation',
ShowEstimationValue, ...)
```

Input Arguments

Peaklist	Cell array of peak lists from a liquid chromatography/mass spectrometry (LC/MS) or gas chromatography/mass spectrometry (GC/MS) data set. Each element in the cell array is a two-column matrix with m/z values in the first column and ion intensity values in the second column. Each element corresponds to a spectrum or retention time. **Note** You can use the `mzxml2peaks` function or the `mspeaks` function to create the *`Peaklist`* cell array.
QuantileValue	Value that determines which peaks are selected by the estimation method to create *`CMZ`*, the vector of common m/z values. Choices are any value `≥ 0` and `≤ 1`. Default is `0.95`.
EstimationMethodValue	Character vector or string specifying the method to estimate *`CMZ`*, the vector of common mass/charge (m/z) values. Choices are: • `histogram` — Default method. Peak locations are clustered using a kernel density estimation approach. The peak ion intensity is used as a weighting factor. The center of all the clusters conform to the *`CMZ`* vector. • `regression` — Takes a sample of the distances between observed significant peaks and regresses the inter-peak distance to create the *`CMZ`* vector with similar inter-element distances.

CorrectionMethodValue	Character vector or string specifying the method to align each peak list to the *CMZ* vector. Choices are: • `nearest-neighbor` — Default method. For each common peak in the *CMZ* vector, its counterpart in each peak list is the peak that is closest to the common peak's m/z value. • `shortest-path` — For each common peak in the *CMZ* vector, its counterpart in each peak list is selected using the shortest path algorithm.
ShowEstimationValue	Controls the display of an assessment plot relative to the estimation method and the vector of common mass/charge (m/z) values. Choices are `true` or `false`. Default is either: • `false` — When return values are specified. • `true` — When return values are not specified.

Output Arguments

CMZ	Vector of common mass/charge (m/z) values estimated by the `mspalign` function.
AlignedPeaks	Cell array of peak lists, with the same form as *Peaklist*, but with corrected m/z values in the first column of each matrix.

Description

`[`*`CMZ`*`, `*`AlignedPeaks`*`] = mspalign(`*`Peaklist`*`)` aligns mass spectra from multiple peak lists (centroided data), by first estimating *CMZ*, a vector of common mass/charge (m/z) values estimated by considering the peaks in all spectra in *Peaklist*, a cell array of peak lists, where each element corresponds to a spectrum or retention time. It then aligns the peaks in each spectrum to the values in *CMZ*, creating *AlignedPeaks*, a cell array of aligned peak lists.

`[`*`CMZ`*`, `*`AlignedPeaks`*`] = mspalign(`*`Peaklist`*`, ...'`*`PropertyName`*`', `*`PropertyValue`*`, ...)` calls `mspalign` with optional properties that use property name/property value pairs. You can specify one or more properties in any order. Each *PropertyName* must be enclosed in single quotation marks and is case insensitive. These property name/property value pairs are as follows:

`[`*`CMZ`*`, `*`AlignedPeaks`*`] = mspalign(`*`Peaklist`*`, ...'Quantile', `*`QuantileValue`*`, ...)` determines which peaks are selected by the estimation method to create *CMZ*, the vector of common m/z values. Choices are a scalar between `0` and `1`. Default is `0.95`.

`[`*`CMZ`*`, `*`AlignedPeaks`*`] = mspalign(`*`Peaklist`*`, ...'EstimationMethod', `*`EstimationMethodValue`*`, ...)` specifies the method used to estimate *CMZ*, the vector of common mass/charge (m/z) values. Choices are:

- `histogram` — Default method. Peak locations are clustered using a kernel density estimation approach. The peak ion intensity is used as a weighting factor. The center of all the clusters conform to the *CMZ* vector.
- `regression` — Takes a sample of the distances between observed significant peaks and regresses the inter-peak distance to create the *CMZ* vector with similar inter-element distances.

`[`*`CMZ`*`, `*`AlignedPeaks`*`] = mspalign(`*`Peaklist`*`, ...'CorrectionMethod', `*`CorrectionMethodValue`*`, ...)` specifies the method used to align each peak list to the *`CMZ`* vector. Choices are:

- `nearest-neighbor` — Default method. For each common peak in the *`CMZ`* vector, its counterpart in each peak list is the peak that is closest to the common peak's m/z value.
- `shortest-path` — For each common peak in the *`CMZ`* vector, its counterpart in each peak list is selected using the shortest path algorithm.

`[`*`CMZ`*`, `*`AlignedPeaks`*`] = mspalign(`*`Peaklist`*`, ...'ShowEstimation', `*`ShowEstimationValue`*`, ...)` controls the display of an assessment plot relative to the estimation method and the estimated vector of common mass/charge (m/z) values. Choices are `true` or `false`. Default is either:

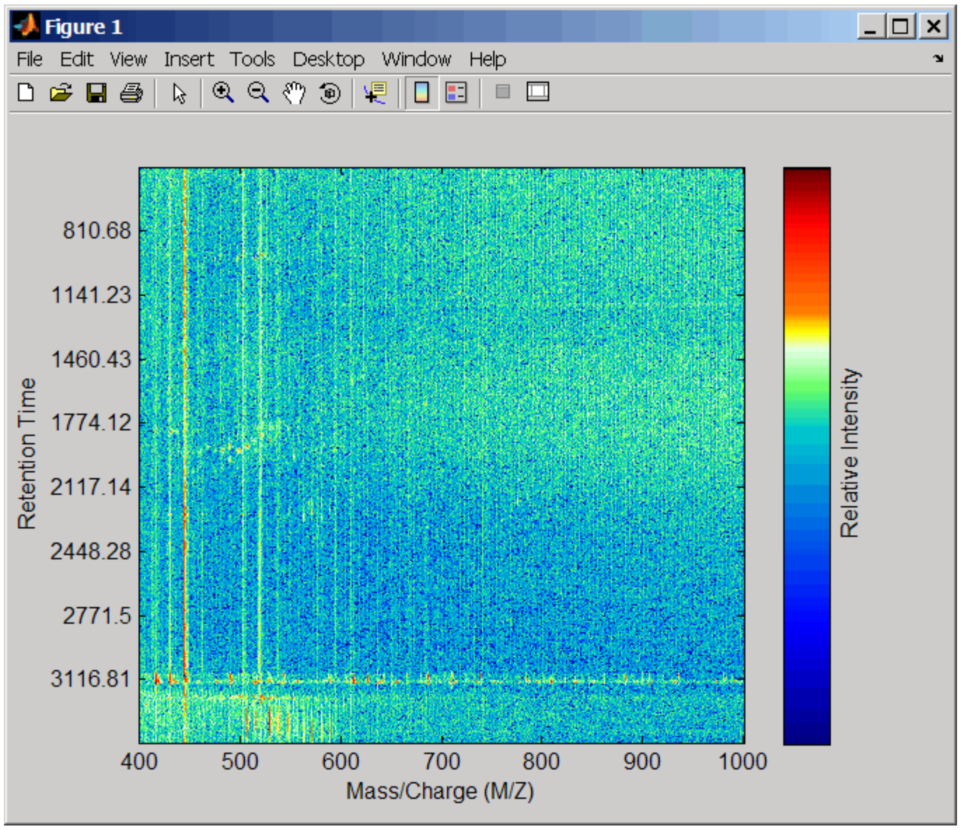
- `false` — When return values are specified.
- `true` — When return values are not specified.

Examples

1. Load a MAT-file, included with the Bioinformatics Toolbox software, which contains liquid chromatography/mass spectrometry (LC/MS) data variables, including `peaks` and `ret_time`. `peaks` is a cell array of peak lists, where each element is a two-column matrix of m/z values and ion intensity values, and each element corresponds to a spectrum or retention time. `ret_time` is a column vector of retention times associated with the LC/MS data set.

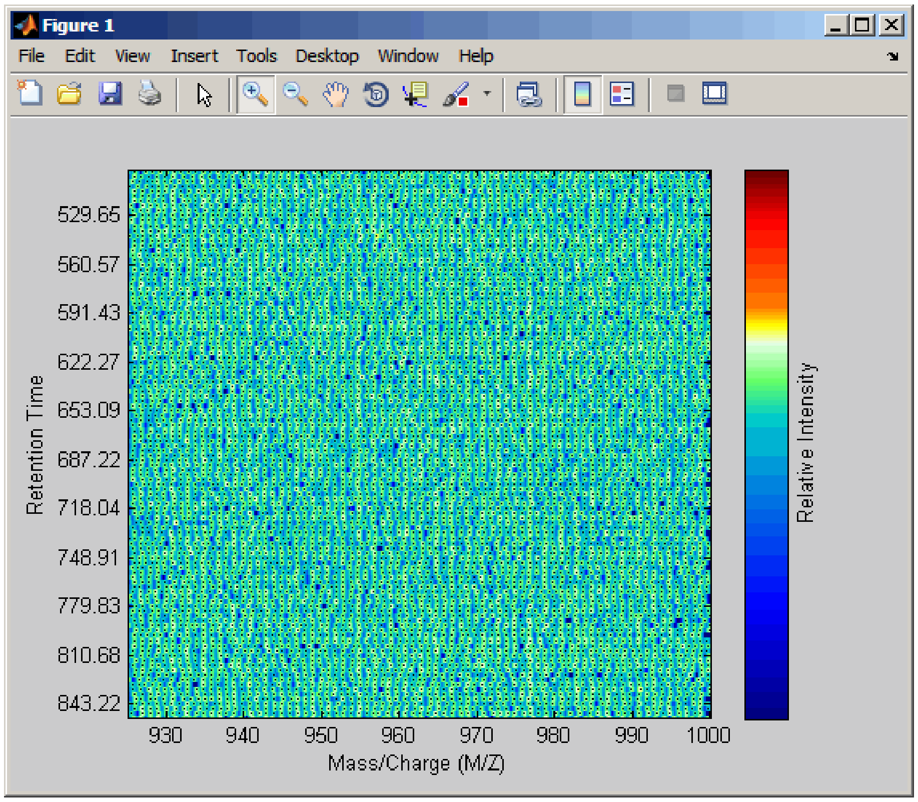
   ```
   load lcmsdata
   ```
2. Resample the unaligned data, display it in a heat map, and then overlay a dot plot.

   ```
   [MZ,Y] = msppresample(ms_peaks,5000);
   msheatmap(MZ,ret_time,log(Y))
   ```



```
msdotplot(ms_peaks,ret_time)
```

3 Click the Zoom In button, and then click the dot plot two or three times to zoom in and see how the dots representing peaks overlay the heat map image.

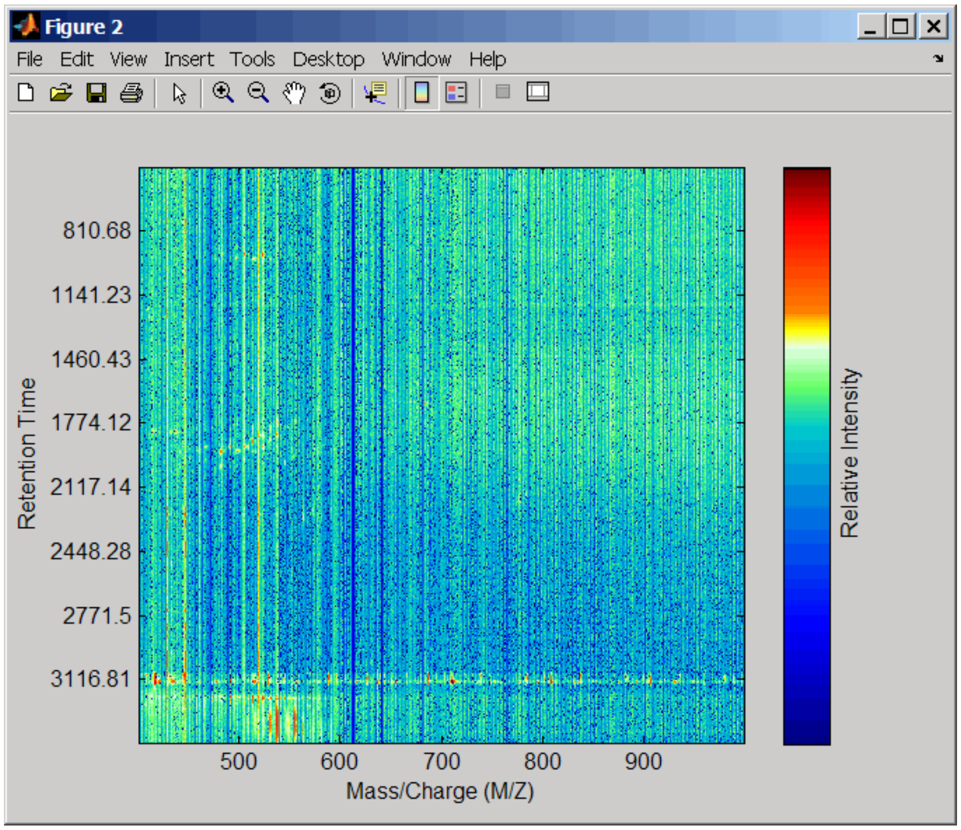


4 Align the peak lists from the mass spectra using the default estimation and correction methods.

```
[CMZ, aligned_peaks] = mspalign(ms_peaks);
```

5 Resample the unaligned data, display it in a heat map, and then overlay a dot plot.

```
[MZ2,Y2] = msppresample(aligned_peaks,5000);
msheatmap(MZ2,ret_time,log(Y2))
```



```
msdotplot(aligned_peaks,ret_time)
```

6 Link the axes of the two heat plots and zoom in to observe the detail to compare the unaligned and aligned LC/MS data sets.

```
linkaxes(findobj(0,'Tag','MSHeatMap'))
axis([480 532 375 485])
```

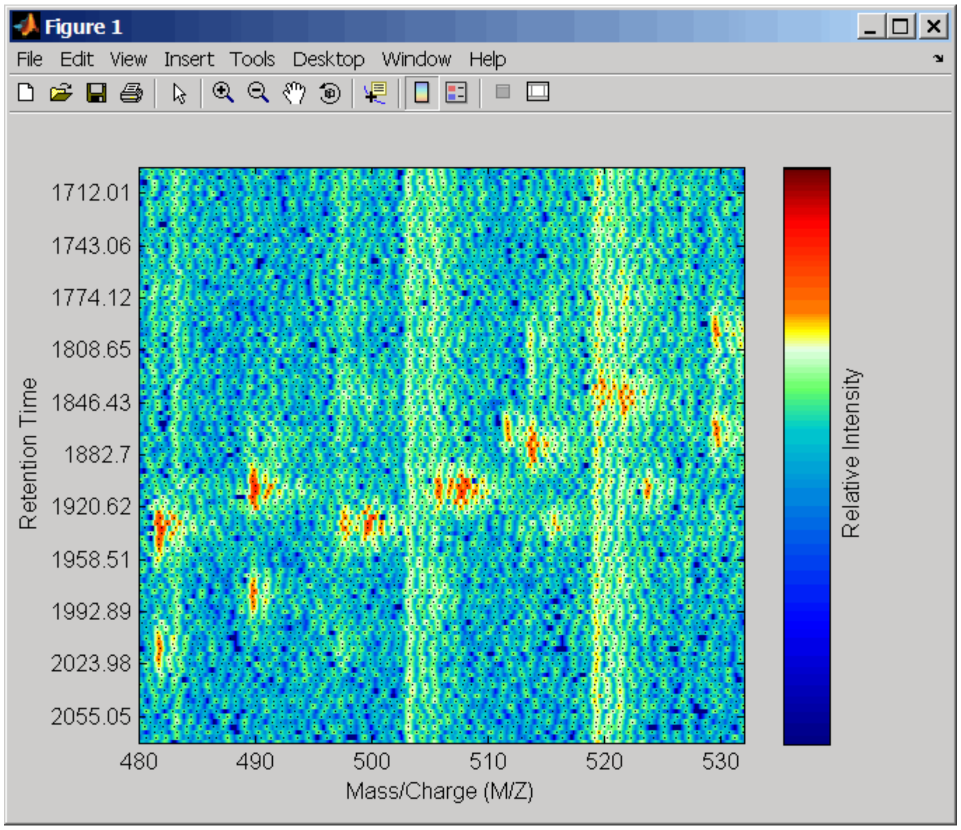
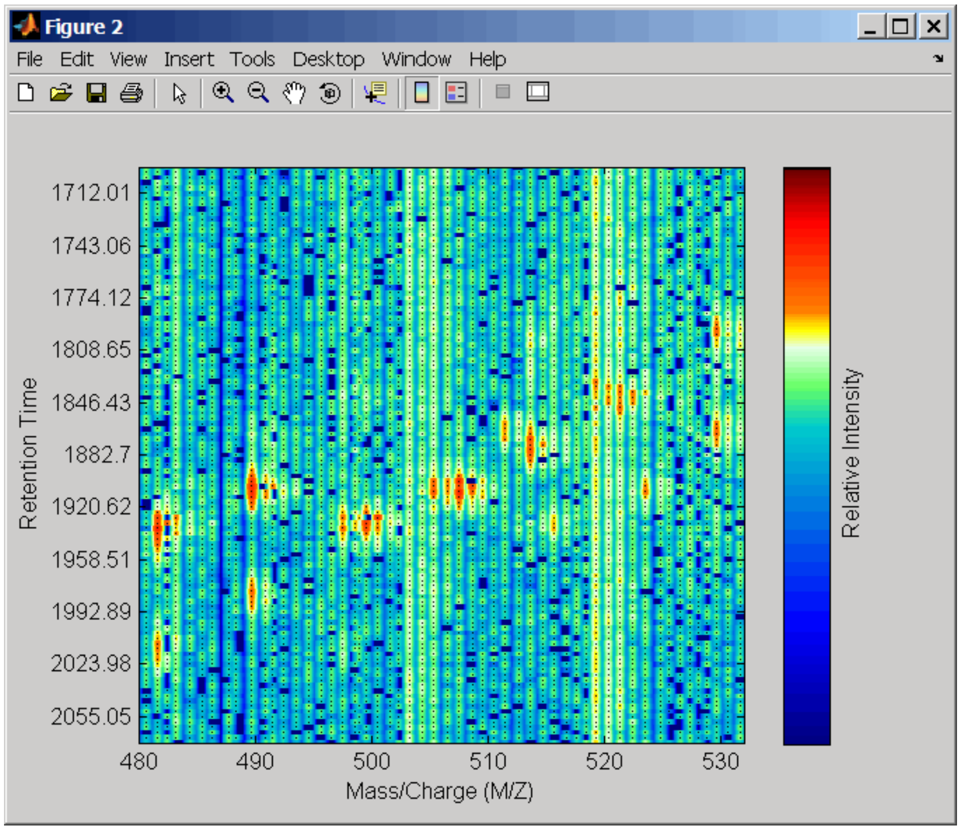


Figure 1
File Edit View Insert Tools Desktop Window Help
Retention Time
1712.01
1743.06
1774.12
1808.65
1846.43
1882.7
1920.62
1958.51
1992.89
2023.98
2055.05
480
490
500
510
520
530
Mass/Charge (M/Z)
Relative Intensity



Version History

Introduced in R2007a

References

[1] Jeffries, N. (2005) Algorithms for alignment of mass spectrometry proteomic data. Bioinfomatics *21:14*, 3066–3073.

[2] Purvine, S., Kolker, N., and Kolker, E. (2004) Spectral Quality Assessment for High-Throughput Tandem Mass Spectrometry Proteomics. OMICS: A Journal of Integrative Biology *8:3*, 255–265.

See Also

`msbackadj` | `msdotplot` | `msalign` | `msheatmap` | `mslowess` | `msnorm` | `mspeaks` | `msresample` | `msppresample` | `mssgolay` | `msviewer`

Topics

“Mass Spectrometry and Bioanalytics”
“Preprocessing Raw Mass Spectrometry Data”
“Visualizing and Preprocessing Hyphenated Mass Spectrometry Data Sets for Metabolite and Protein/Peptide Profiling”

"Differential Analysis of Complex Protein and Metabolite Mixtures using Liquid Chromatography/ Mass Spectrometry (LC/MS)"

mspeaks

Convert raw peak data to peak list (centroided data)

Syntax

```
Peaklist = mspeaks(X, Intensities)
[Peaklist, PFWHH] = mspeaks(X, Intensities)
[Peaklist, PFWHH, PExt] = mspeaks(X, Intensities)
mspeaks(X, Intensities, ...'Base', BaseValue, ...)
mspeaks(X, Intensities, ...'Levels', LevelsValue, ...)
mspeaks(X, Intensities, ...'NoiseEstimator', NoiseEstimatorValue, ...)
mspeaks(X, Intensities, ...'Multiplier', MultiplierValue, ...)
mspeaks(X, Intensities, ...'Denoising', DenoisingValue, ...)
mspeaks(X, Intensities, ...'PeakLocation', PeakLocationValue, ...)
mspeaks(X, Intensities, ...'FWHHFilter', FWHHFilterValue, ...)
mspeaks(X, Intensities, ...'OverSegmentationFilter',
OverSegmentationFilterValue, ...)
mspeaks(X, Intensities, ...'HeightFilter', HeightFilterValue, ...)
mspeaks(X, Intensities, ...'ShowPlot', ShowPlotValue, ...)
mspeaks(X, Intensities, ...'Style', StyleValue, ...)
```

Description

`Peaklist = mspeaks(X, Intensities)` finds relevant peaks in raw, noisy peak signal data, and creates *`Peaklist`*, a two-column matrix, containing the separation-axis value and intensity for each peak. *`X`* is a vector of separation-unit values for a set of signals with peaks. *`Intensities`* is a matrix of intensity values for a set of peaks that share the same separation-unit range.

`[Peaklist, PFWHH] = mspeaks(X, Intensities)` returns *`PFWHH`*, a two-column matrix indicating the left and right locations of the full width at half height (FWHH) markers for each peak. For any peak not resolved at FWHH, `mspeaks` returns the peak shape extents instead. When *`Intensities`* includes multiple signals, then *`PFWHH`* is a cell array of matrices.

`[Peaklist, PFWHH, PExt] = mspeaks(X, Intensities)` returns *`PExt`*, a two-column matrix indicating the left and right locations of the peak shape extents determined after wavelet denoising. When *`Intensities`* includes multiple signals, then *`PExt`* is a cell array of matrices.

`mspeaks(X, Intensities, ...'PropertyName', PropertyValue, ...)` calls `mspeaks` with optional properties that use property name/property value pairs. You can specify one or more properties in any order. Enclose each *`PropertyName`* in single quotation marks. Each *`PropertyName`* is case insensitive. These property name/property value pairs are as follows:

`mspeaks(X, Intensities, ...'Base', BaseValue, ...)` specifies the wavelet base.

`mspeaks(X, Intensities, ...'Levels', LevelsValue, ...)` specifies the number of levels for the wavelet decomposition.

`mspeaks(X, Intensities, ...'NoiseEstimator', NoiseEstimatorValue, ...)` specifies the method to estimate the threshold, `T`, to filter out noisy components in the first high-band decomposition (`y_h`).

`mspeaks(`*`X`*`, `*`Intensities`*`, ...'Multiplier', `*`MultiplierValue`*`, ...)` specifies the threshold multiplier constant.

`mspeaks(`*`X`*`, `*`Intensities`*`, ...'Denoising', `*`DenoisingValue`*`, ...)` controls the use of wavelet denoising to smooth the signal. Choices are `true` (default) or `false`.

`mspeaks(`*`X`*`, `*`Intensities`*`, ...'PeakLocation', `*`PeakLocationValue`*`, ...)` specifies the proportion of the peak height to use to select the points used to compute the centroid separation-axis value of the respective peak. *`PeakLocationValue`* must be a value `≥ 0` and `≤ 1`. Default is `1.0`.

`mspeaks(`*`X`*`, `*`Intensities`*`, ...'FWHHFilter', `*`FWHHFilterValue`*`, ...)` specifies the minimum full width at half height (FWHH), in separation units, for reported peaks. Peaks with FWHH below this value are excluded from the output list *`Peaklist`*.

`mspeaks(`*`X`*`, `*`Intensities`*`, ...'OverSegmentationFilter',` *`OverSegmentationFilterValue`*`, ...)` specifies the minimum distance, in separation units, between neighboring peaks. When the signal is not smoothed appropriately, multiple maxima can appear to represent the same peak. Increase this filter value to join oversegmented peaks into a single peak.

`mspeaks(`*`X`*`, `*`Intensities`*`, ...'HeightFilter', `*`HeightFilterValue`*`, ...)` specifies the minimum height for reported peaks. Peaks with heights below this value are excluded from the output list *`Peaklist`*.

`mspeaks(`*`X`*`, `*`Intensities`*`, ...'ShowPlot', `*`ShowPlotValue`*`, ...)` controls the display of a plot of the original and the smoothed signal, with the peaks included in the output matrix *`Peaklist`* marked.

`mspeaks(`*`X`*`, `*`Intensities`*`, ...'Style', `*`StyleValue`*`, ...)` specifies the style for marking the peaks in the plot.

`mspeaks` finds peaks in data from any separation technique that produces signal data, such as spectroscopy, nuclear magnetic resonance (NMR), electrophoresis, chromatography, or mass spectrometry.

Input Arguments

X

Vector of separation-unit values for a set of signals with peaks. The number of elements in the vector equals the number of rows in the matrix *`Intensities`*. The separation unit can quantify wavelength, frequency, distance, time, or m/z depending on the instrument that generates the signal data.

Default:

Intensities

Matrix of intensity values for a set of peaks that share the same separation-unit range. Each row corresponds to a separation-unit value, and each column corresponds to either a set of signals with peaks or a retention time. The number of rows equals the number of elements in vector *`X`*.

Default:

`BaseValue`

Integer from `2` to `20` that specifies the wavelet base.

Default: `4`

`LevelsValue`

Integer from `1` to `12` that specifies the number of levels for the wavelet decomposition.

Default: `10`

`NoiseEstimatorValue`

Character vector, string, or scalar that specifies the method to estimate the threshold, `T`, to filter out noisy components in the first high-band decomposition (`y_h`). Choices are:

- `mad` — Default. Median absolute deviation, which calculates `T = sqrt(2*log(`*`n`*`))*mad(`*`y_h`*`) / 0.6745`, where *`n`* = the number of rows in the *`Intensities`* matrix.
- `std` — Standard deviation, which calculates `T = std(`*`y_h`*`)`.
- A positive real value.

Default:

`MultiplierValue`

Positive real value that specifies the threshold multiplier constant.

Default: `1.0`

`DenoisingValue`

Controls the use of wavelet denoising to smooth the signal. Choices are `true` (default) or `false`.

Tip If your data was previously smoothed, for example, with the `mslowess` or `mssgolay` function, you do not need to use wavelet denoising. Set this property to `false`.

Default:

`PeakLocationValue`

Value that specifies the proportion of the peak height to use to select the points to compute the centroid separation-axis value of the respective peak. The value must be `≥ 0` and `≤ 1`.

Note When *`PeakLocationValue`* = `1.0`, the peak location is at the maximum of the peak. When *`PeakLocationValue`* = `0`, `mspeaks` computes the peak location with all the points from the closest minimum to the left of the peak to the closest minimum to the right of the peak.

Default: `1.0`

`FWHHFilterValue`

Positive real value that specifies the minimum full width at half height (FWHH), in separation units, for reported peaks. Peaks with FWHH below this value are excluded from the output list *`Peaklist`*.

Default: `0`

`OverSegmentationFilterValue`

Positive real value that specifies the minimum distance, in separation units, between neighboring peaks. When the signal is not smoothed appropriately, multiple maxima can appear to represent the same peak. Increase this filter value to join oversegmented peaks into a single peak.

Default: `0`

`HeightFilterValue`

Positive real value that specifies the minimum height for reported peaks.

Default: `0`

`ShowPlotValue`

Controls the display of a plot of the original signal and the smoothed signal, with the peaks included in the output matrix *`Peaklist`* marked. Choices are `true`, `false`, or *`I`*, an integer specifying the index of a spectrum in *`Intensities`*. If set to `true`, the first spectrum in *`Intensities`* is plotted. Default is:

- `false` — When you specify return values.
- `true` — When you do not specify return values.

Default:

`StyleValue`

Character vector or string specifying the style for marking the peaks in the plot. Choices are:

- `'peak'` (default) — Places a marker at the peak crest.
- `'exttriangle'` — Draws a triangle using the peak crest and the extents.
- `'fwhhtriangle'` — Draws a triangle using the peak crest and the FWHH points.
- `'extline'` — Places a marker at the peak crest and vertical lines at the extents.
- `'fwhhline'` — Places a marker at the peak crest and a horizontal line at FWHH.

Default:

Output Arguments

`Peaklist`

Two-column matrix where each row corresponds to a peak. The first column contains separation-unit values (indicating the location of peaks along the separation axis). The second column contains intensity values. When *`Intensities`* includes multiple signals, then *`Peaklist`* is a cell array of matrices, each containing a peak list.

`PFWHH`

Two-column matrix indicating the left and right locations of the full width at half height (FWHH) markers for each peak. For any peak not resolved at FWHH, `mspeaks` returns the peak shape extents instead. When *`Intensities`* includes multiple signals, then *`PFWHH`* is a cell array of matrices.

`PExt`

Two-column matrix indicating the left and right locations of the peak shape extents determined after wavelet denoising. When *`Intensities`* includes multiple signals, then *`PExt`* is a cell array of matrices.

Examples

1 Load a MAT-file, included with the Bioinformatics Toolbox software, that contains two mass spectrometry data variables, `MZ_lo_res` and `Y_lo_res`. `MZ_lo_res` is a vector of m/z values for a set of spectra. `Y_lo_res` is a matrix of intensity values for a set of mass spectra that share the same m/z range.

```
load sample_lo_res
```

2 Adjust the baseline of the eight spectra stored in `Y_lo_res`.

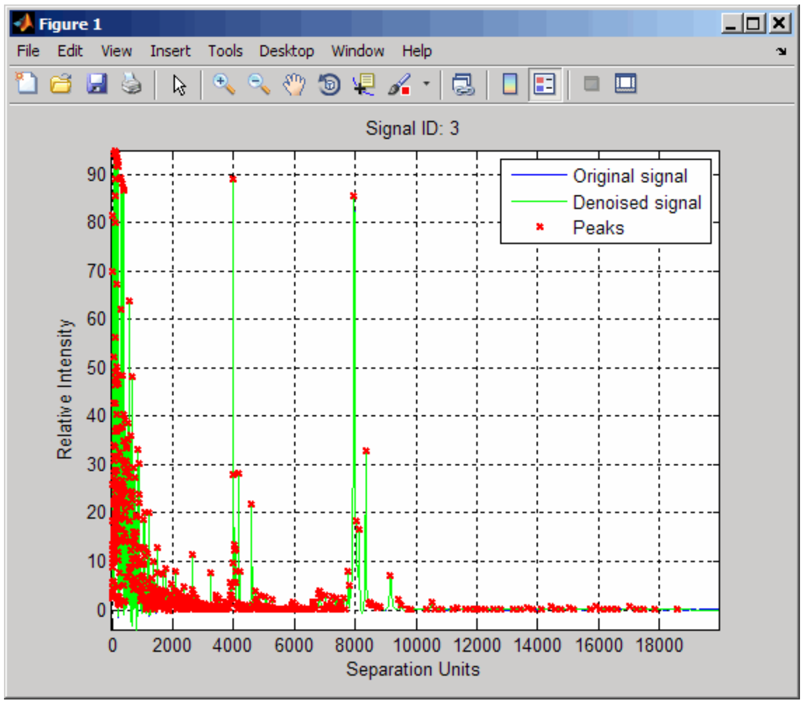
```
YB = msbackadj(MZ_lo_res,Y_lo_res);
```

3 Convert the raw mass spectrometry data to a peak list by finding the relevant peaks in each spectrum.

```
P = mspeaks(MZ_lo_res,YB);
```

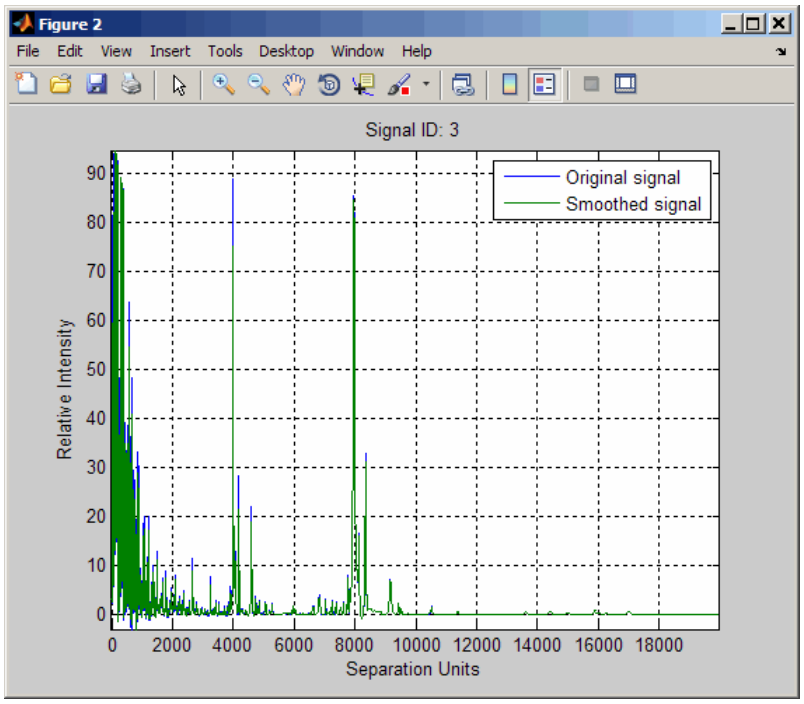
4 Plot the third spectrum in `YB`, the matrix of baseline-corrected intensity values, with the detected peaks marked.

```
P = mspeaks(MZ_lo_res,YB,'SHOWPLOT',3);
```

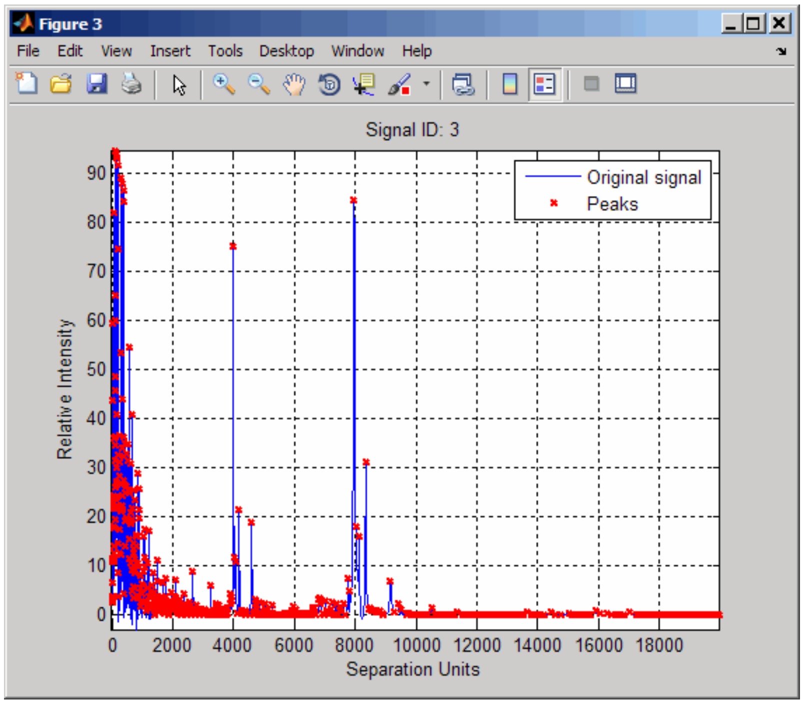


5 Smooth the signal using the `mslowess` function. Then convert the smoothed data to a peak list by finding relevant peaks and plot the third spectrum.

```
YS = mslowess(MZ_lo_res,YB,'SHOWPLOT',3);
```

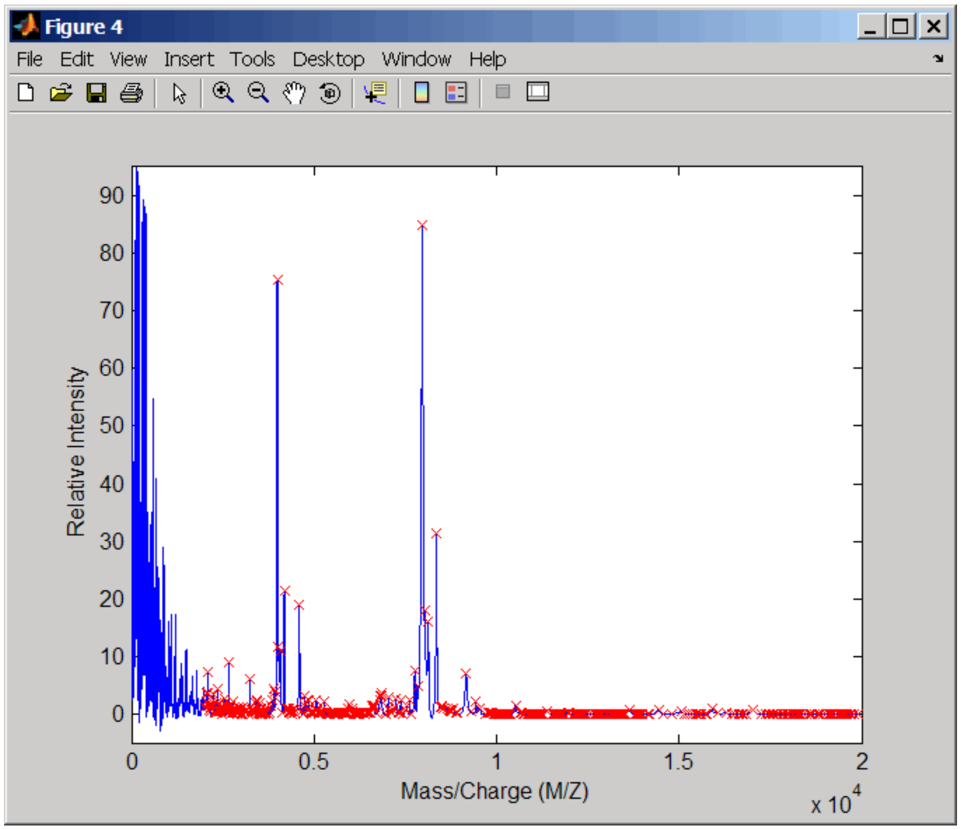


```
P = mspeaks(MZ_lo_res,YS,'DENOISING',false,'SHOWPLOT',3);
```



6 Use the `cellfun` function to remove all peaks with m/z values less than 2000 from the eight peaks listed in output `P`. Then plot the peaks of the third spectrum (in red) over its smoothed signal (in blue).

```
Q = cellfun(@(p) p(p(:,1)>2000,:),P,'UniformOutput',false);
figure
plot(MZ_lo_res,YS(:,3),'b',Q{3}(:,1),Q{3}(:,2),'rx')
xlabel('Mass/Charge (M/Z)')
ylabel('Relative Intensity')
axis([0 20000 -5 95])
```



Algorithms

`mspeaks` converts raw peak data to a peak list (centroided data) by:

1. Smoothing the signal using undecimated wavelet transform with Daubechies coefficients
2. Assigning peak locations
3. Estimating noise
4. Eliminating peaks that do not satisfy specified criteria

Version History

Introduced in R2007a

References

[1] Morris, J.S., Coombes, K.R., Koomen, J., Baggerly, K.A., and Kobayash, R. (2005) Feature extraction and quantification for mass spectrometry in biomedical applications using the mean spectrum. Bioinfomatics *21:9*, 1764–1775.

[2] Yasui, Y., Pepe, M., Thompson, M.L., Adam, B.L., Wright, G.L., Qu, Y., Potter, J.D., Winget, M., Thornquist, M., and Feng, Z. (2003) A data-analytic strategy for protein biomarker discovery: profiling of high-dimensional proteomic data for cancer detection. Biostatistics *4:3*, 449–463.

[3] Donoho, D.L., and Johnstone, I.M. (1995) Adapting to unknown smoothness via wavelet shrinkage. J. Am. Statist. Asso. *90*, 1200–1224.

[4] Strang, G., and Nguyen, T. (1996) Wavelets and Filter Banks (Wellesley: Cambridge Press).

[5] Coombes, K.R., Tsavachidis, S., Morris, J.S., Baggerly, K.A., Hung, M.C., and Kuerer, H.M. (2005) Improved peak detection and quantification of mass spectrometry data acquired from surface-enhanced laser desorption and ionization by denoising spectra with the undecimated discrete wavelet transform. Proteomics *5(16)*, 4107–4117.

See Also
`mspalign` | `msbackadj` | `msdotplot` | `msalign` | `msheatmap` | `mslowess` | `msnorm` | `msresample` | `msppresample` | `mssgolay` | `msviewer`

Topics
"Mass Spectrometry and Bioanalytics"
"Preprocessing Raw Mass Spectrometry Data"
"Visualizing and Preprocessing Hyphenated Mass Spectrometry Data Sets for Metabolite and Protein/Peptide Profiling"
"Differential Analysis of Complex Protein and Metabolite Mixtures using Liquid Chromatography/Mass Spectrometry (LC/MS)"

msppresample

Resample signal with peaks while preserving peaks

Syntax

```
[X, Intensities] = msppresample(Peaklist, N)

msppresample(Peaklist, N, ...'Range', RangeValue, ...)
msppresample(Peaklist, N, ...'FWHH', FWHHValue, ...)
msppresample(Peaklist, N, ...'ShowPlot', ShowPlotValue, ...)
```

Input Arguments

Peaklist	Either of the following: • Two-column matrix, where the first column contains separation-unit values and the second column contains intensity values. The separation unit can quantify wavelength, frequency, distance, time, or m/z depending on the instrument that generates the signal data. • Cell array of peak lists, where each element is a two-column matrix of separation-unit values and intensity values, and each element corresponds to a signal or retention time. **Tip** You can use the `mzxml2peaks` function or the `mspeaks` function to create the *Peaklist* matrix or cell array.
N	Integer specifying the number of equally spaced points (separation-unit values) in the resampled signal.
RangeValue	1-by-2 vector specifying the minimum and maximum separation-unit values for the output matrix *Intensities*. *RangeValue* must be within `[min(`*inputSU*`) max(`*inputSU*`)]`, where *inputSU* is the concatenated separation-unit values from the input *Peaklist*. Default is the full range `[min(`*inputSU*`) max(`*inputSU*`)]`.
FWHHValue	Value that specifies the full width at half height (FWHH) in separation units. The FWHH is used to convert each peak to a Gaussian shaped curve. Default is `median(diff(`*inputSU*`))/2`, where *inputSU* is the concatenated separation-unit values from the input *Peaklist*. The default is a rough approximation of resolution observed in the input data, *Peaklist*. **Tip** To ensure that the resolution of the peaks is preserved, set *FWHHValue* to half the distance between the two peaks of interest that are closest to each other.

ShowPlotValue	Controls the display of a plot of an original and resampled signal. Choices are `true`, `false`, or *I*, an integer specifying the index of a signal in *Intensities*. If you set to `true`, the first signal in *Intensities* is plotted. Default is: • `false` — When return values are specified. • `true` — When return values are not specified.

Output Arguments

X	Vector of equally spaced, common separation-unit values for a set of signals with peaks. The number of elements in the vector equals *N*, or the number of rows in matrix *Intensities*.
Intensities	Matrix of reconstructed intensity values for a set of peaks that share the same separation-unit range. Each row corresponds to a separation-unit value, and each column corresponds to either a set of signals with peaks or a retention time. The number of rows equals *N*, or the number of elements in vector *X*.

Description

Tip Use the following syntaxes with data from any separation technique that produces signal data, such as spectroscopy, NMR, electrophoresis, chromatography, or mass spectrometry.

`[`*X*`, `*Intensities*`] = msppresample(`*Peaklist*`, `*N*`)` resamples *Peaklist*, a peak list, by converting centroided peaks to a semicontinuous, raw signal that preserves peak information. The resampled signal has *N* equally spaced points. Output *X* is a vector of *N* elements specifying the equally spaced, common separation-unit values for the set of signals with peaks. Output *Intensities* is a matrix of reconstructed intensity values for a set of peaks that share the same separation-unit range. Each row corresponds to a separation-unit value, and each column corresponds to either a set of signals with peaks or a retention time. The number of rows equals *N*.

`msppresample` uses a Gaussian kernel to reconstruct the signal. The intensity at any given separation-unit value is taken from the maximum intensity of any contributing (overlapping) peaks.

Tip `msppresample` is useful to prepare a set of signals for imaging functions such as `msheatmap` and preprocessing functions such as `msbackadj` and `msnorm`.

`msppresample(`*Peaklist*`, `*N*`, ... '`*PropertyName*`', `*PropertyValue*`, ...)` calls `msppresample` with optional properties that use property name/property value pairs. You can specify one or more properties in any order. Each *PropertyName* must be enclosed in single quotation marks and is case insensitive. These property name/property value pairs are as follows:

`msppresample(`*Peaklist*`, `*N*`, ...'Range', `*RangeValue*`, ...)` specifies a separation-unit range for the output matrix *Intensities* using the minimum and maximum separation values specified in the 1-by-2 vector *RangeValue*. *RangeValue* must be within `[min(`*inputSU*`)` `max(`*`inputSU`*`)]`, where *`inputSU`* is the concatenated separation-unit values from the input *`Peaklist`*. Default is the full range `[min(`*`inputSU`*`) max(`*`inputSU`*`)]`

`msppresample(`*`Peaklist`*`, `*`N`*`, ...'FWHH', `*`FWHHValue`*`, ...)` sets the full width at half height (FWHH) in separation units. The FWHH is used to convert each peak to a Gaussian shaped curve. Default is `median(diff(`*`inputSU`*`))/2`, where *`inputSU`* is the concatenated separation-unit values from the input *`Peaklist`*. The default is a rough approximation of resolution observed in the input data, *`Peaklist`*.

Tip To ensure that the resolution of the peaks is preserved, set *`FWHHValue`* to half the distance between the two peaks of interest that are closest to each other.

`msppresample(`*`Peaklist`*`, `*`N`*`, ...'ShowPlot', `*`ShowPlotValue`*`, ...)` controls the display of a plot of an original and resampled signal. Choices are `true`, `false`, or *`I`*, an integer specifying the index of a signal in *`Intensities`*. If you set to `true`, the first signal in *`Intensities`* is plotted. Default is:

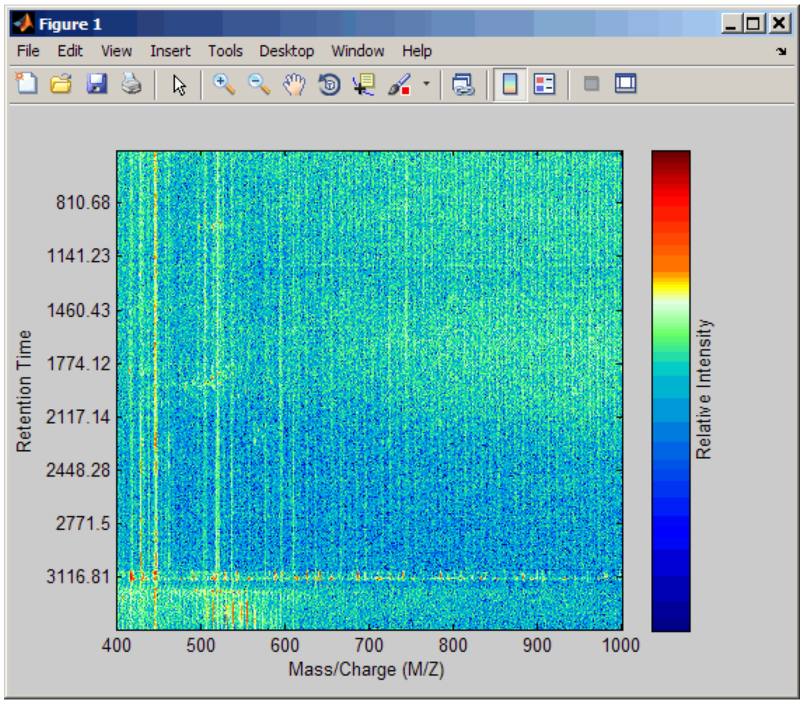
- `false` — When return values are specified.
- `true` — When return values are not specified.

Examples

1. Load a MAT-file, included with the Bioinformatics Toolbox software, that contains liquid chromatography/mass spectrometry (LC/MS) data variables. It includes `peaks`, a cell array of peak lists, where each element is a two-column matrix of m/z values and ion intensity values, and each element corresponds to a spectrum or retention time.

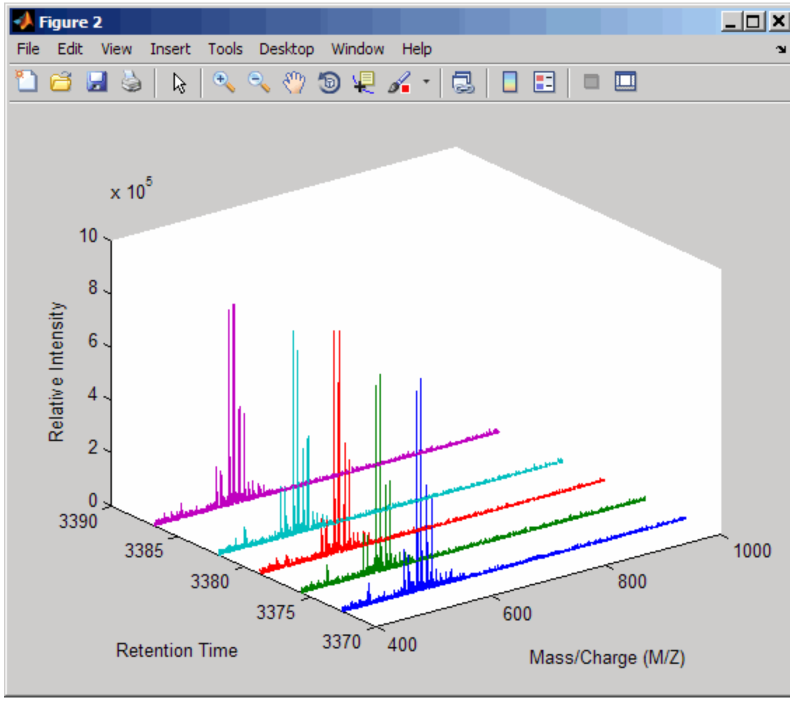
   ```
   load lcmsdata
   ```
2. Resample the data, specifying 5000 m/z values in the resampled signal. Then create a heat map of the LC/MS data.

   ```
   [MZ,Y] = msppresample(ms_peaks,5000);
   msheatmap(MZ,ret_time,log(Y))
   ```



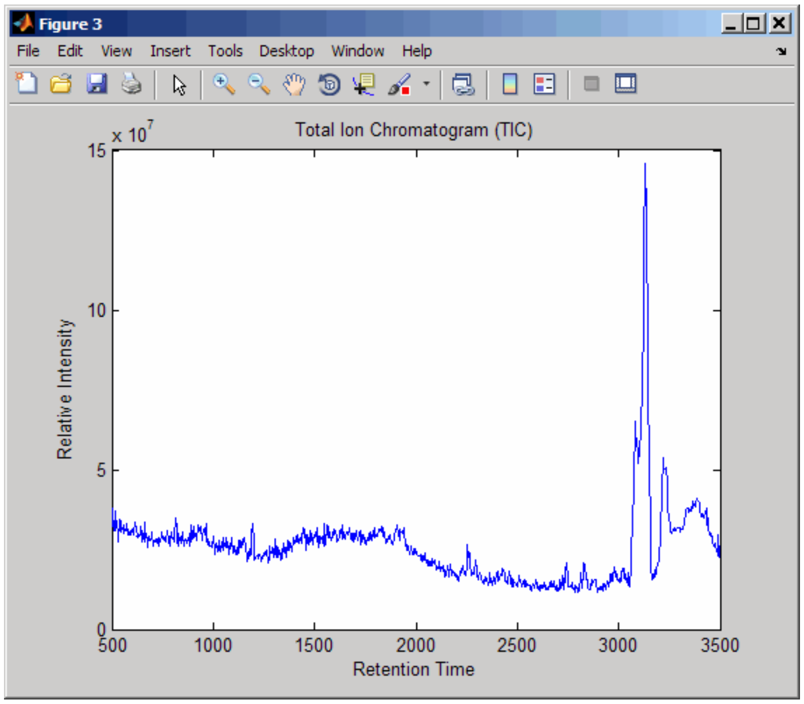
3 Plot the reconstructed profile spectra between two retention times.

```
figure
t1 = 3370;
t2 = 3390;
h = find(ret_time>t1 & ret_time<t2);
[MZ,Y] = msppresample(ms_peaks(h),10000);
plot3(repmat(MZ,1,numel(h)),repmat(ret_time(h)',10000,1),Y)
xlabel('Mass/Charge (M/Z)')
ylabel('Retention Time')
zlabel('Relative Intensity')
```



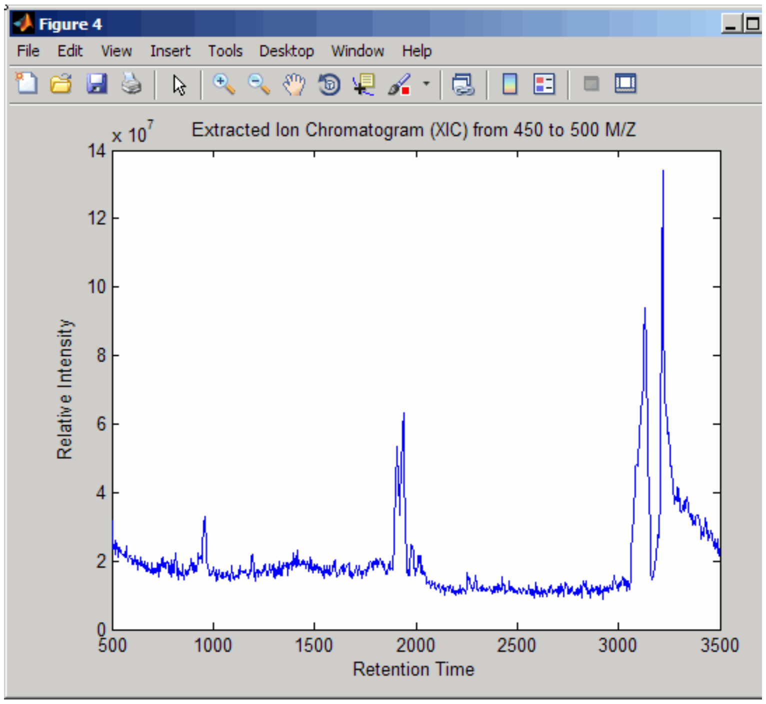
4 Resample the data to plot the Total Ion Chromatogram (TIC).

```
figure
[MZ,Y] = msppresample(ms_peaks,5000);
plot(ret_time,sum(Y))
title('Total Ion Chromatogram (TIC)')
xlabel('Retention Time')
ylabel('Relative Intensity')
```



5 Resample the data to plot the Extracted Ion Chromatogram (XIC) in the 450 to 500 m/z range.

```
figure
[MZ,Y] = msppresample(ms_peaks,5000,'Range',[450 500]);
plot(ret_time,sum(Y))
title('Extracted Ion Chromatogram (XIC) from 450 to 500 M/Z')
xlabel('Retention Time')
ylabel('Relative Intensity')
```



Version History

Introduced in R2007a

See Also

`mspalign` | `msbackadj` | `msdotplot` | `msalign` | `msheatmap` | `mslowess` | `msnorm` | `mspeaks` | `msresample` | `mssgolay` | `msviewer`

Topics

“Mass Spectrometry and Bioanalytics”
“Preprocessing Raw Mass Spectrometry Data”
“Visualizing and Preprocessing Hyphenated Mass Spectrometry Data Sets for Metabolite and Protein/Peptide Profiling”
“Differential Analysis of Complex Protein and Metabolite Mixtures using Liquid Chromatography/Mass Spectrometry (LC/MS)”

msresample

Resample signal with peaks

Syntax

```
[Xout, Intensitiesout] = msresample(X, Intensities, N)

msresample(..., 'Uniform', UniformValue, ...)
msresample(..., 'Range', RangeValue, ...)
msresample(..., 'RangeWarnOff', RangeWarnOffValue, ...)
msresample(..., 'Missing', MissingValue, ...)
msresample(..., 'Window', WindowValue, ...)
msresample(..., 'Cutoff', CutoffValue, ...)
msresample(..., 'ShowPlot', ShowPlotValue, ...)
```

Arguments

X	Vector of separation-unit values for a set of signals with peaks. The number of elements in the vector equals the number of rows in the matrix *Intensities*. The separation unit can quantify wavelength, frequency, distance, time, or m/z depending on the instrument that generates the signal data.
Intensities	Matrix of intensity values for a set of peaks that share the same separation-unit range. Each row corresponds to a separation-unit value, and each column corresponds to either a set of signals with peaks or a retention time. The number of rows equals the number of elements in vector *X*.
N	Positive integer specifying the total number of samples.

Description

Tip Use the following syntaxes with data from any separation technique that produces signal data, such as spectroscopy, NMR, electrophoresis, chromatography, or mass spectrometry.

`[Xout, Intensitiesout] = msresample(X, Intensities, N)` resamples raw noisy signal data, *Intensities*. The output signal has *N* samples with a spacing that increases linearly within the range `[min(X) max(X)]`. *X* can be a linear or a quadratic function of its index. When you set input arguments such that down-sampling takes place, `msresample` applies a lowpass filter before resampling to minimize aliasing.

For the antialias filter, `msresample` uses a linear-phase FIR filter with a least-squares error minimization. The cutoff frequency is set by the largest down-sampling ratio when comparing the same regions in the *X* and *Xout* vectors.

Tip `msresample` is particularly useful when you have signals with different separation-unit vectors and you want to match the scales.

`msresample(..., 'PropertyName', PropertyValue, ...)` calls `msresample` with optional properties that use property name/property value pairs. You can specify one or more properties in any order. Each *`PropertyName`* must be enclosed in single quotes and is case insensitive. These property name/property value pairs are as follows:

`msresample(..., 'Uniform', UniformValue, ...)`, when *`UniformValue`* is `true`, it forces the vector *`X`* to be uniformly spaced. The default value is `false`.

`msresample(..., 'Range', RangeValue, ...)` specifies a 1-by-2 vector with the separation-unit range for the output signal, *`Intensitiesout`*. *`RangeValue`* must be within `[min(X) max(X)]`. Default value is the full range `[min(X) max(X)]`. When *`RangeValue`* values exceed the values in *`X`*, `msresample` extrapolates the signal with zeros and returns a warning message.

`msresample(..., 'RangeWarnOff', RangeWarnOffValue, ...)` controls the return of a warning message when *`RangeValue`* values exceed the values in *`X`*. *`RangeWarnOffValue`* can be `true` or `false` (default).

`msresample(..., 'Missing', MissingValue, ...)`, when *`MissingValue`* is `true`, analyzes the input vector, *`X`*, for dropped samples. The default value is `false`. If the down-sample factor is large, checking for dropped samples might not be worth the extra computing time. Dropped samples can only be recovered if the original separation-unit values follow a linear or a quadratic function of the *`X`* vector index.

`msresample(..., 'Window', WindowValue, ...)` specifies the window used when calculating parameters for the lowpass filter. Enter `'Flattop'`, `'Blackman'`, `'Hamming'`, or `'Hanning'`. The default value is `'Flattop'`.

`msresample(..., 'Cutoff', CutoffValue, ...)` specifies the cutoff frequency. Enter a scalar value from `0` to `1` (Nyquist frequency or half the sampling frequency). By default, `msresample` estimates the cutoff value by inspecting the separation-unit vectors, *`X`* and *`XOut`*. However, the cutoff frequency might be underestimated if *`X`* has anomalies.

`msresample(..., 'ShowPlot', ShowPlotValue, ...)` plots the original and the resampled signal. When `msresample` is called without output arguments, the signals are plotted unless *`ShowPlotValue`* is `false`. When *`ShowPlotValue`* is `true`, only the first signal in *`Intensities`* is plotted. *`ShowPlotValue`* can also contain an index to one of the signals in *`Intensities`*.

Tip LC/MS data analysis requires extended amounts of memory from the operating system.

- If you receive errors related to memory, try the following:
  - Increase the virtual memory (swap space) for your operating system as described in “Resolve “Out of Memory” Errors”.
- If you receive errors related to Java heap space, increase your Java heap space:
  - If you have MATLAB version 7.10 (R2010a) or later, see “Java Heap Memory Preferences”.
  - If you have MATLAB version 7.9 (R2009b) or earlier, see https://www.mathworks.com/matlabcentral/answers/92813-how-do-i-increase-the-heap-space-for-the-java-vm-in-matlab.

Examples

Resample Mass Spectrometry Data

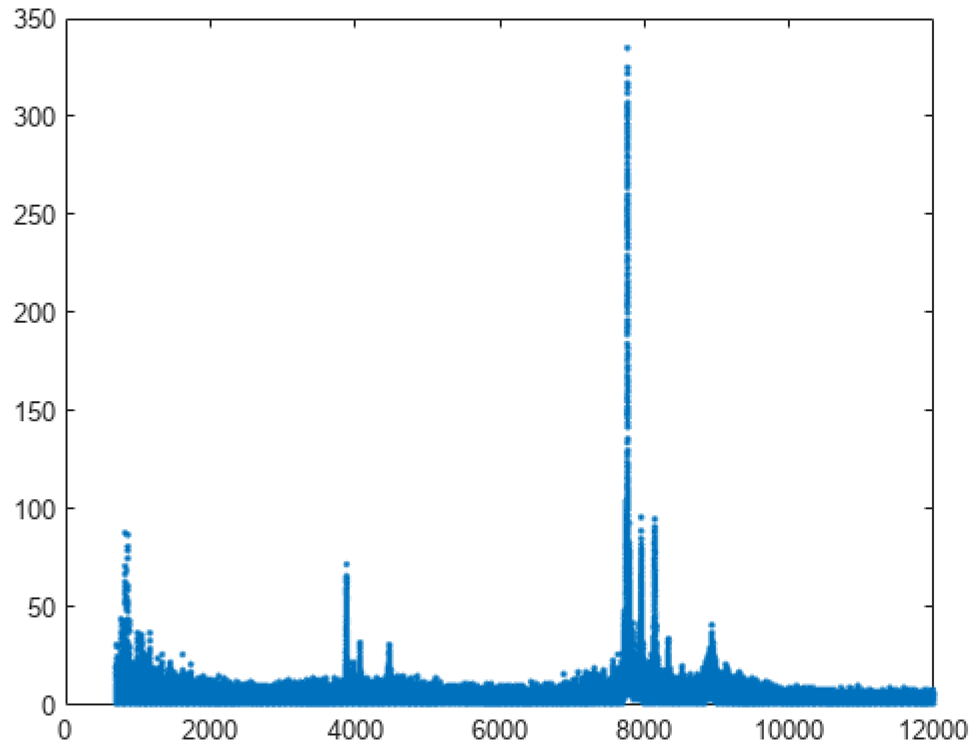
This example shows how to resample mass spec data.

Load a MAT-file, included with Bioinformatics Toolbox™, that contains mass spectrometry data, and then extract m/z and intensity value vectors.

```
load sample_hi_res;
mz = MZ_hi_res;
y = Y_hi_res;
```

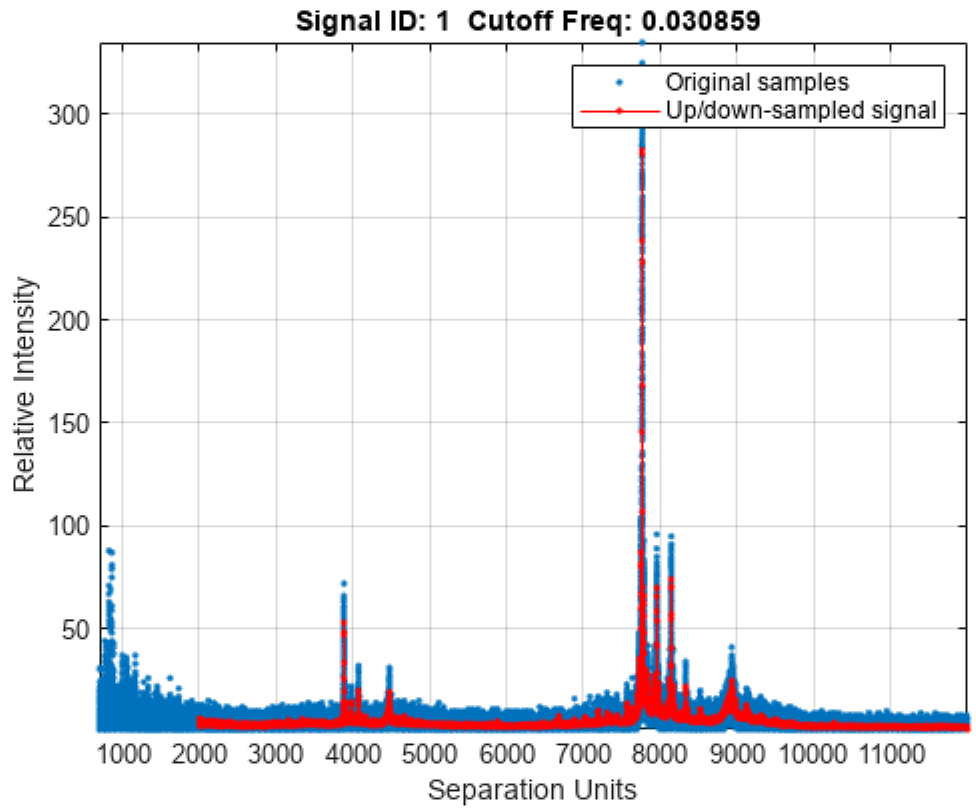
Plot the original data.

```
plot(mz, y, '.')
```



Resample the spectrogram to have 10000 samples between 2000 and maximum m/z value in the data set, and show both the resampled and original data.

```
[mz1,y1] = msresample(mz, y, 10000, 'range',[2000 max(mz)],'SHOWPLOT',true);
```



Version History

Introduced before R2006a

See Also

`mspalign` | `msbackadj` | `msdotplot` | `msalign` | `msheatmap` | `mslowess` | `msnorm` | `mspeaks` | `msppresample` | `mssgolay` | `msviewer`

Topics

"Mass Spectrometry and Bioanalytics"
"Preprocessing Raw Mass Spectrometry Data"
"Visualizing and Preprocessing Hyphenated Mass Spectrometry Data Sets for Metabolite and Protein/Peptide Profiling"
"Differential Analysis of Complex Protein and Metabolite Mixtures using Liquid Chromatography/Mass Spectrometry (LC/MS)"

mssgolay

Smooth signal with peaks using least-squares polynomial

Syntax

```
Yout = mssgolay(X, Intensities)

mssgolay(X, Intensities, ...'Span', SpanValue, ...)
mssgolay(X, Intensities, ...'Degree', DegreeValue, ...)
mssgolay(X, Intensities, ...'ShowPlot', ShowPlotValue, ...)
```

Arguments

X	Vector of separation-unit values for a set of signals with peaks. The number of elements in the vector equals the number of rows in the matrix *Intensities*. The separation unit can quantify wavelength, frequency, distance, time, or m/z depending on the instrument that generates the signal data.
Intensities	Matrix of intensity values for a set of peaks that share the same separation-unit range. Each row corresponds to a separation-unit value, and each column corresponds to either a set of signals with peaks or a retention time. The number of rows equals the number of elements in vector *X*.

Description

Tip Use the following syntaxes with data from any separation technique that produces signal data, such as spectroscopy, NMR, electrophoresis, chromatography, or mass spectrometry.

`Yout = mssgolay(X, Intensities)` smooths raw noisy signal data, *Intensities*, using a least-squares digital polynomial filter (Savitzky and Golay filters). The default span or frame is `15` samples.

`mssgolay(X, Intensities, ...'PropertyName', PropertyValue, ...)` calls `mssgolay` with optional properties that use property name/property value pairs. You can specify one or more properties in any order. Each *PropertyName* must be enclosed in single quotation marks and is case insensitive. These property name/property value pairs are as follows:

`mssgolay(X, Intensities, ...'Span', SpanValue, ...)` modifies the frame size for the smoothing function. If *SpanValue* is greater than `1`, the window is the size of *SpanValue* in samples independent of the *X* vector. Higher values smooth the signal more with an increase in computation time. If *SpanValue* is less than `1`, the window size is a fraction of the number of points in the input data, *X*. For example, if *SpanValue* is `0.05`, the window size is equal to `5%` of the number of points in *X*.

Note The original algorithm by Savitzky and Golay assumes the input vector, *X*, has uniformly spaced separation units, while `mssgolay` also allows one that is not uniformly spaced. Therefore, the sliding frame for smoothing is centered using the closest samples in terms of the *X* value and not in terms of the *X* index.

When the input vector, *X*, does not have repeated values or NaN values, the algorithm is approximately twice as fast.

When the input vector, *X*, is evenly spaced, the least-squares fitting is performed once so that the signal is filtered with the same coefficients, and the speed of the algorithm increases considerably.

If the input vector, *X*, is evenly spaced and *SpanValue* is even, span is incremented by `1` to include both edge samples in the frame.

`mssgolay(`*X*`, `*Intensities*`, ...'Degree', `*DegreeValue*`, ...)` specifies the degree of the polynomial (*DegreeValue*) fitted to the points in the moving frame. The default value is `2`. *DegreeValue* must be smaller than *SpanValue*.

`mssgolay(`*X*`, `*Intensities*`, ...'ShowPlot', `*ShowPlotValue*`, ...)` plots smoothed signals over the original. When `mssgolay` is called without output arguments, the signals are plotted unless *ShowPlotValue* is `false`. When *ShowPlotValue* is `true`, only the first signal in *Intensities* is plotted. *ShowPlotValue* can also contain an index to one of the signals in *Intensities*.

Examples

Smooth Mass Spectrometry Data

This example shows how to smooth mass spectrometry data using least-squares polynomial approach.

Load a MAT-file, included with Bioinformatics Toolbox™, that contains mass spectrometry data including `MZ_lo_res` , a vector of m/z values for a set of spectra, and `Y_lo_res` , a matrix of intensity values for a set of mass spectra that share the same m/z charge.

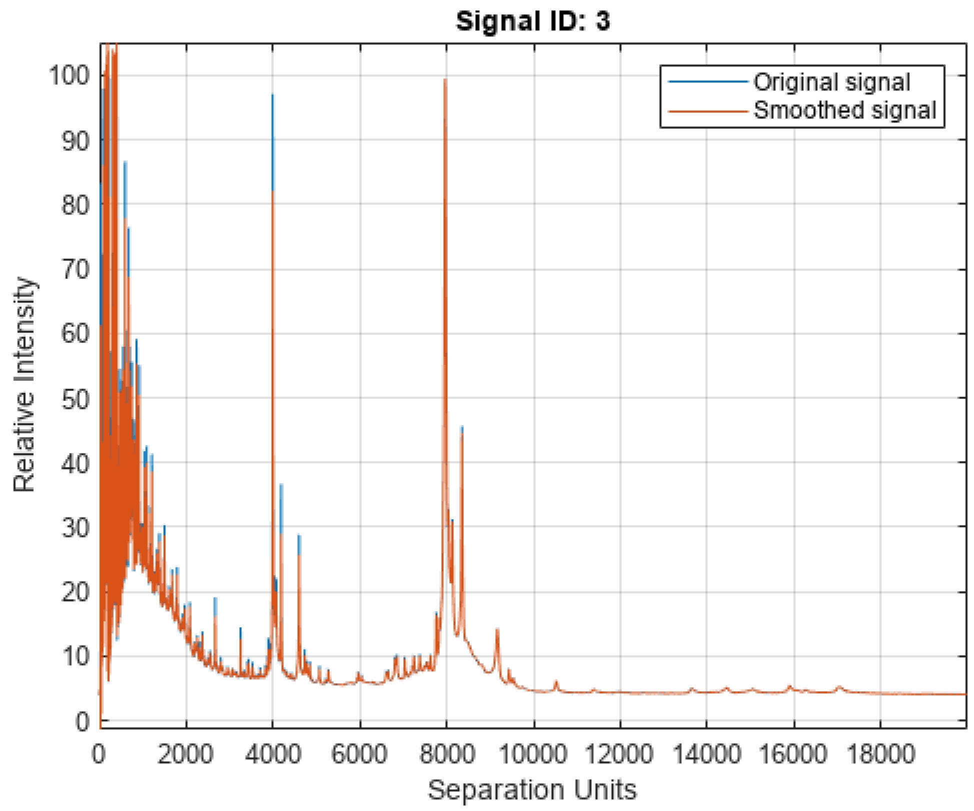
```
load sample_lo_res
```

Apply least-squares polynomial smoothing to the data.

```
YS = mssgolay(MZ_lo_res, Y_lo_res);
```

Plot the third sample/spectrogram in `Y_lo_res` , and its smoothed signal.

```
mssgolay(MZ_lo_res,Y_lo_res,'SHOWPLOT',3);
```



Version History

Introduced before R2006a

See Also

`mspalign` | `msbackadj` | `msdotplot` | `msalign` | `msheatmap` | `mslowess` | `msnorm` | `mspeaks` | `msresample` | `msppresample` | `msviewer`

Topics

“Mass Spectrometry and Bioanalytics”
“Preprocessing Raw Mass Spectrometry Data”
“Visualizing and Preprocessing Hyphenated Mass Spectrometry Data Sets for Metabolite and Protein/Peptide Profiling”
“Differential Analysis of Complex Protein and Metabolite Mixtures using Liquid Chromatography/Mass Spectrometry (LC/MS)”

msviewer

Explore mass spectrum or set of mass spectra

Syntax

```
msviewer(MZ , Intensities)
msviewer(..., 'Markers', MarkersValue)
msviewer(..., 'Group', GroupValue)
```

Arguments

`MZ`	Column vector of common mass/charge (m/z) values for a set of spectra. The number of elements in the vector equals the number of rows in the matrix *`Intensities`*.
`Intensities`	Matrix of intensity values for a set of mass spectra that share the same m/z range. Each row corresponds to an m/z value, and each column corresponds to a spectrum or retention time. The number of rows equals the number of elements in vector *`MZ`*.
`MarkersValue`	Column vector to specify a list of marker positions from the mass/charge vector *`MZ`*.
`GroupValue`	Either of the following: • Vector of values with the same number of elements as rows in the matrix *`Intensities`* • Cell array of character vectors or string vector with the same number of elements as rows (spectra) in the matrix *`Intensities`* Each value, character vector, or string specifies a group to which the corresponding spectrum belongs. Spectra from the same group are plotted with the same color. Default is `[1:numSpectra]`.

Description

`msviewer(`*`MZ`*` , `*`Intensities`*`)` displays the MS Viewer, which lets you view and explore a mass spectrum defined by *`MZ`* and *`Intensities`*.

`msviewer(..., 'Markers', `*`MarkersValue`*`)` specifies a list of marker positions from the mass/charge vector, *`MZ`*, for exploration and easy navigation. Enter a column vector with *`MZ`* values.

`msviewer(..., 'Group', `*`GroupValue`*`)` specifies a group to which the spectra belong. The groups are specified by *`GroupValue`*, a vector of values or cell array of character vectors or string vector. The number of values or character vectors or strings is the same as the number of rows in the matrix *`Intensities`*. Each value or character vector specifies a group to which the corresponding spectrum belongs. Spectra from the same group are plotted with the same color. Default is `[1:numSpectra]`.

The MS Viewer includes the following features:

- Plot mass spectra. The spectra are plotted with different colors according to their group labels.
- An overview displays a full spectrum, and a box indicates the region that is currently displayed in the main window.
- Five different zoom in options, one zoom out option, and a reset view option resize the spectrum.
- Add/focus/move/delete marker operations
- Import/Export markers from/to MATLAB workspace
- Print and preview the spectra plot
- Print the spectra plot to a MATLAB Figure window

MSViewer has five components:

- Menu bar: **File**, **Tools**, **Window**, and **Help**
- Toolbar: Move marker, Zoom XY, Zoom X, Zoom Y, Zoom out, Reset view, and Help
- Main window: display the spectra
- Overview window: display the overview of a full spectrum (the average of all spectra in display)
- Marker control panel: a list of markers, Add Marker, Delete Marker, up and down buttons

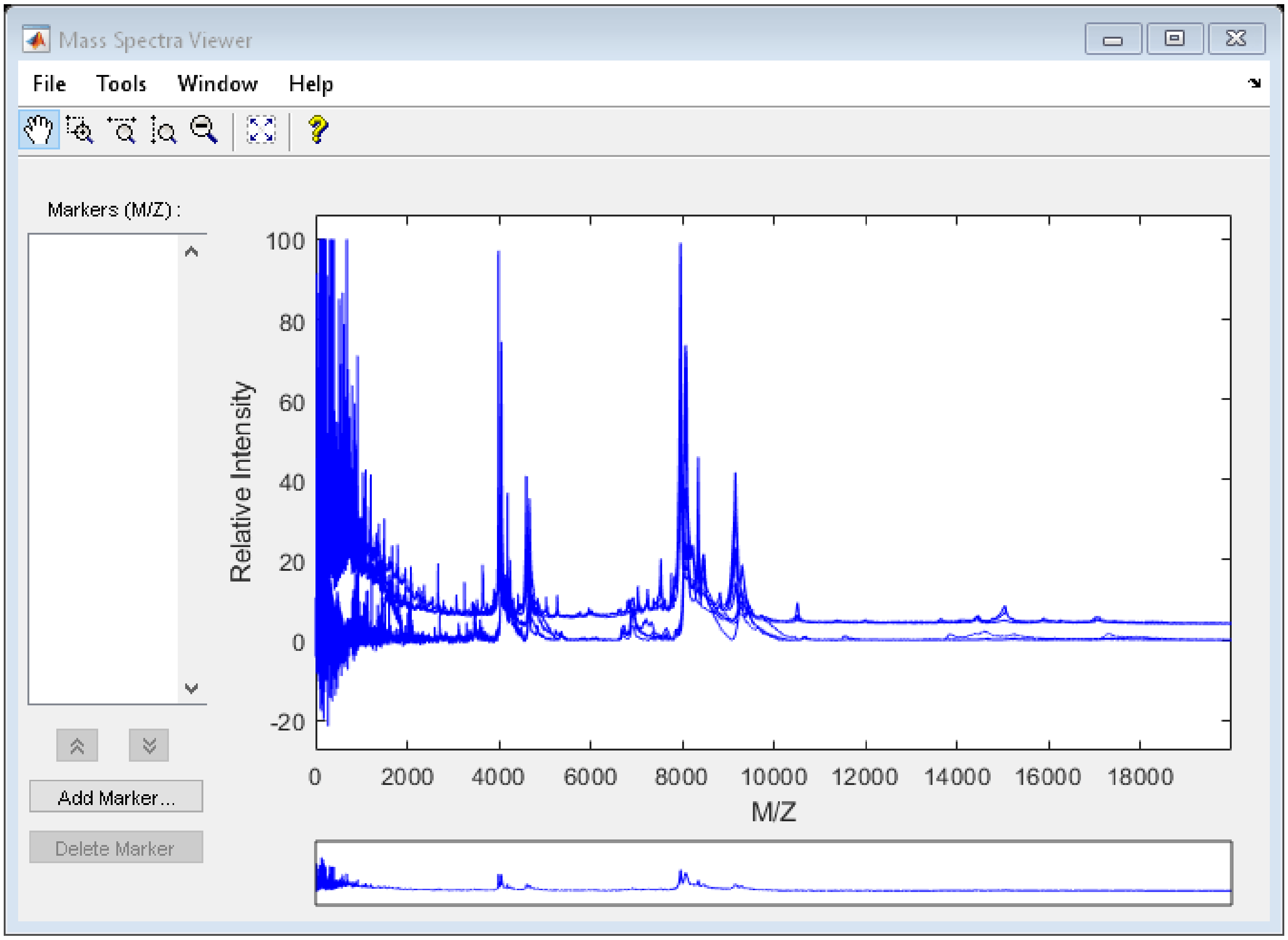
Examples

Plot Mass Spectra Data

This example shows how to plot mass spectra data.

Load and plot a sample mass spectra data.

```
load sample_lo_res
msviewer(MZ_lo_res, Y_lo_res)
```



Add a marker by pointing to a mass peak, right-clicking, and then clicking **Add Marker**.

The **File** menu has the following options.

- **Import Markers from Workspace** - Opens the Import Markers From MATLAB® Workspace dialog. The dialog displays a list of double Mx1 or 1xM variables. If the selected variable is out of range, the viewer displays an error message.
- **Export Markers to Workspace** - Opens the Export Markers to MATLAB® Workspace dialog. Enter a variable name for the markers. All markers are saved. If thre is no marker available, this menu item is disabled.
- **Print to Figure** - Prints the spectra plot in the main display to a MATLAB® figure window.

The **Tools** menu has the following options.

- **Add Marker** - Opens the Add Marker dialog where you can enter an m/z marker.
- **Delete Marker** - Removes the currently selected m/z marker from the **Markers** (m/z) list.
- **Next Marker** or **Previous Marker** - Moves the selection up and down the **Markers** list.

- **Zoom XY**, **Zoom X**, **Zoom Y**, or **Zoom Out** - Changes the cursor from an arrow to a crosshair. Left-click and drag a rectangle box over an area and then release it. The display zooms the area covered by the box.

From the range window at the bottom, move the view box to a new location.

Version History

Introduced before R2006a

See Also

`mspalign` | `msbackadj` | `msdotplot` | `msalign` | `msheatmap` | `mslowess` | `msnorm` | `mspeaks` | `msresample` | `msppresample` | `mssgolay`

Topics

“Mass Spectrometry and Bioanalytics”
“Preprocessing Raw Mass Spectrometry Data”
“Visualizing and Preprocessing Hyphenated Mass Spectrometry Data Sets for Metabolite and Protein/Peptide Profiling”
“Differential Analysis of Complex Protein and Metabolite Mixtures using Liquid Chromatography/Mass Spectrometry (LC/MS)”

multialign

Align multiple sequences using progressive method

Syntax

```
SeqsMultiAligned = multialign(Seqs)
SeqsMultiAligned = multialign(Seqs, Tree)

multialign(..., 'PropertyName', PropertyValue,...)
multialign(..., 'Weights', WeightsValue)
multialign(..., 'ScoringMatrix', ScoringMatrixValue)
multialign(..., 'SMInterp', SMInterpValue)
multialign(..., 'GapOpen', GapOpenValue)
multialign(..., 'ExtendGap', ExtendGapValue)
multialign(..., 'DelayCutoff', DelayCutoffValue)
multialign(..., 'UseParallel', UseParallelValue)
multialign(..., 'Verbose', VerboseValue)
multialign(..., 'ExistingGapAdjust', ExistingGapAdjustValue)
multialign(..., 'TerminalGapAdjust', TerminalGapAdjustValue)
```

Input Arguments

Seqs	Vector of structures with the fields `'Sequence'` for the residues and `'Header'` or `'Name'` for the labels. *Seqs* can also be a string vector, cell array of character vectors, or character array.
Tree	Phylogenetic tree calculated with the `seqlinkage on page 1-1713` or `seqneighjoin on page 1-1723` function.
WeightsValue	Property to select the sequence weighting method. Enter `'THG'` (default) or `'equal'`.

`ScoringMatrixValue`	Either of the following: • Character vector or string specifying the scoring matrix to use for the alignment. Choices for amino acid sequences are: • `'BLOSUM62'` • `'BLOSUM30'` increasing by `5` up to `'BLOSUM90'` • `'BLOSUM100'` • `'PAM10'` increasing by `10` up to `'PAM500'` • `'DAYHOFF'` • `'GONNET'` Default is: • `'BLOSUM80'` to `'BLOSUM30'` series — When *`AlphabetValue`* equals `'AA'` • `'NUC44'` — When *`AlphabetValue`* equals `'NT'` **Note** The above scoring matrices, provided with the software, also include a structure containing a scale factor that converts the units of the output score to bits. You can also use the `'Scale'` property to specify an additional scale factor to convert the output score from bits to another unit. • Matrix representing the scoring matrix to use for the alignment. It can be a matrix, such as returned by the `blosum`, `pam`, `dayhoff`, `gonnet`, or `nuc44` function. It can also be an *M*-by-*M* matrix or *M*-by-*M*-by-*N* array of matrices with *N* user-defined scoring matrices. **Note** If you use a scoring matrix that you created or was created by one of the above functions, the matrix does not include a scale factor. The output score will be returned in the same units as the scoring matrix. When passing your own series of scoring matrices, ensure they share the same scale. **Note** If you need to compile `multialign` into a stand-alone application or software component using MATLAB Compiler, use a matrix instead of a character vector or string for *`ScoringMatrixValue`*.
`SMInterpValue`	Property to specify whether linear interpolation of the scoring matrices is on or off. When `false`, the scoring matrix is assigned to a fixed range depending on the distances between the two profiles (or sequences) being aligned. Default is `true`.

GapOpenValue	Scalar or a function specified using `@`. If you enter a function, `multialign` passes four values to the function: the average score for two matched residues (`sm`), the average score for two mismatched residues (`sx`), and, the length of both profiles or sequences (`len1`, `len2`). Default is `@(sm,sx,len1,len2) 5*sm`.
ExtendGapValue	Scalar or a function specified using `@`. If you enter a function, `multiialign` passes four values to the function: the average score for two matched residues (`sm`), the average score for two mismatched residues (`sx`), and the length of both profiles or sequences (`len1`, `len2`). Default is `@(sm,sx,len1,len2) sm/4`.
DelayCutoffValue	Property to specify the threshold delay of divergent sequences. Default is unity where sequences with the closest sequence farther than the median distance are delayed.
UseParallelValue	Controls the computation of the pairwise alignments using `parfor`-loops. When `true`, and Parallel Computing Toolbox is installed and a `parpool` is open, computation occurs in parallel. If there are no open `parpool`, but automatic creation is enabled in the Parallel Preferences, the default pool will be automatically open and computation occurs in parallel. If Parallel Computing Toolbox is installed, but there are no open `parpool` and automatic creation is disabled, then computation uses `parfor`-loops in serial mode. If Parallel Computing Toolbox is not installed, then computation uses `parfor`-loops in serial mode. Default is `false`, which uses for-loops in serial mode.
VerboseValue	Property to control displaying the sequences with sequence information. Default is `false`.
ExistingGapAdjustValue	Property to control automatic adjustment based on existing gaps. Default is `true`.
TerminalGapAdjustValue	Property to adjust the penalty for opening a gap at the ends of the sequence. Default is `false`.

Output Arguments

SeqsMultiAligned	Vector of structures (same as *Seqs*) but with the field `'Sequence'` updated with the alignment. When *Seqs* is a cell or char array, *SeqsMultiAligned* is a char array with the output alignment following the same order as the input.

Description

`*SeqsMultiAligned* = multialign(*Seqs*)` performs a progressive multiple alignment for a set of sequences (*Seqs*). Pairwise distances between sequences are computed after pairwise alignment with the Gonnet scoring matrix and then by counting the proportion of sites at which each pair of sequences are different (ignoring gaps). The guide tree is calculated by the neighbor-joining method assuming equal variance and independence of evolutionary distance estimates.

`SeqsMultiAligned = multialign(Seqs, Tree)` uses a tree (`Tree`) as a guide for the progressive alignment. The sequences (`Seqs`) should have the same order as the leaves in the tree (`Tree`) or use a field (`'Header'` or `'Name'`) to identify the sequences.

`multialign(..., 'PropertyName', PropertyValue,...)` enters optional arguments as property name/property value pairs. Specify one or more properties in any order. Enclose each `PropertyName` in single quotation marks. Each `PropertyName` is case insensitive. These property name/property value pairs are as follows:

`multialign(..., 'Weights', WeightsValue)` selects the sequence weighting method. Weights emphasize highly divergent sequences by scaling the scoring matrix and gap penalties. Closer sequences receive smaller weights.

Values of the property `Weights` are:

- `'THG'` (default) — Thompson-Higgins-Gibson method using the phylogenetic tree branch distances weighted by their thickness.
- `'equal'` — Assigns the same weight to every sequence.

`multialign(..., 'ScoringMatrix', ScoringMatrixValue)` selects the scoring matrix (`ScoringMatrixValue`) for the progressive alignment. Match and mismatch scores are interpolated from the series of scoring matrices by considering the distances between the two profiles or sequences being aligned. The first matrix corresponds to the smallest distance, and the last matrix to the largest distance. Intermediate distances are calculated using linear interpolation.

`multialign(..., 'SMInterp', SMInterpValue)`, when `SMInterpValue` is `false`, turns off the linear interpolation of the scoring matrices. Instead, each supplied scoring matrix is assigned to a fixed range depending on the distances between the two profiles or sequences being aligned.

`multialign(..., 'GapOpen', GapOpenValue)` specifies the initial penalty for opening a gap.

`multialign(..., 'ExtendGap', ExtendGapValue)` specifies the initial penalty for extending a gap.

`multialign(..., 'DelayCutoff', DelayCutoffValue)` specifies a threshold to delay the alignment of divergent sequences whose closest neighbor is farther than

```
(DelayCutoffValue) * (median patristic distance between sequences)
```

`multialign(..., 'UseParallel', UseParallelValue)` specifies whether to use `parfor`-loops when computing the pairwise alignments. When `true`, and Parallel Computing Toolbox is installed and a `parpool` is open, computation occurs in parallel. If there are no open `parpool`, but automatic creation is enabled in the Parallel Preferences, the default pool will be automatically open and computation occurs in parallel. If Parallel Computing Toolbox is installed, but there are no open `parpool` and automatic creation is disabled, then computation uses `parfor`-loops in serial mode. If Parallel Computing Toolbox is not installed, then computation uses `parfor`-loops in serial mode. Default is `false`, which uses for-loops in serial mode.

`multialign(..., 'Verbose', VerboseValue)`, when `VerboseValue` is `true`, turns on verbosity.

The remaining input optional arguments are analogous to the function `profalign on page 1-1520` and are used through every step of the progressive alignment of profiles.

`multialign(..., 'ExistingGapAdjust', ExistingGapAdjustValue)`, when *`ExistingGapAdjustValue`* is `false`, turns off the automatic adjustment based on existing gaps of the position-specific penalties for opening a gap.

When *`ExistingGapAdjustValue`* is `true`, for every profile position, `profalign` proportionally lowers the penalty for opening a gap toward the penalty of extending a gap based on the proportion of gaps found in the contiguous symbols and on the weight of the input profile.

`multialign(..., 'TerminalGapAdjust', TerminalGapAdjustValue)`, when *`TerminalGapAdjustValue`* is `true`, adjusts the penalty for opening a gap at the ends of the sequence to be equal to the penalty for extending a gap.

Examples

Align multiple sequences

This example shows how to align multiple protein sequences.

Use the `fastaread` function to read p53samples.txt, a FASTA-formatted file included with Bioinformatics Toolbox™, which contains p53 protein sequences of seven species.

```
p53 = fastaread('p53samples.txt')
```

```
p53=7×1 struct array with fields:
    Header
    Sequence
```

Compute the pairwise distances between each pair of sequences using the 'GONNET' scoring matrix.

```
dist = seqpdist(p53,'ScoringMatrix','GONNET');
```

Build a phylogenetic tree using an unweighted average distance (UPGMA) method. This tree will be used as a guiding tree in the next step of progressive alignment.

```
tree = seqlinkage(dist,'average',p53)
```

```
    Phylogenetic tree object with 7 leaves (6 branches)
```

Perform progressive alignment using the PAM family scoring matrices.

```
ma = multialign(p53,tree,'ScoringMatrix',...
                {'pam150','pam200','pam250'})
```

```
ma=7×1 struct array with fields:
    Header
    Sequence
```

Align Nucleotide Sequences

1. Enter an array of sequences.

   ```
   seqs = {'CACGTAACATCTC','ACGACGTAACATCTTCT','AAACGTAACATCTCGC'};
   ```
2. Promote terminations with gaps in the alignment.

   ```
   multialign(seqs,'terminalGapAdjust',true)

   ans =
   --CACGTAACATCTC--
   ACGACGTAACATCTTCT
   -AAACGTAACATCTCGC
   ```
3. Compare the alignment without termination gap adjustment.

   ```
   multialign(seqs)

   ans =
   CA--CGTAACATCT--C
   ACGACGTAACATCTTCT
   AA-ACGTAACATCTCGC
   ```

Version History
Introduced before R2006a

Extended Capabilities

Automatic Parallel Support
Accelerate code by automatically running computation in parallel using Parallel Computing Toolbox™.

To run in parallel, set `'UseParallel'` to `true`.

For more information, see the `'UseParallel'` name-value pair argument.

See Also
`align2cigar` | `hmmprofalign` | `multialignread` | `multialignwrite` | `nwalign` | `profalign` | `seqprofile` | `seqconsensus` | `seqneighjoin`

multialignread

Read multiple sequence alignment file

Syntax

```
S = multialignread(File)
[Headers, Sequences] = multialignread(File)
... = multialignread(File, 'IgnoreGaps', IgnoreGapsValue)
... = multialignread(File, 'TimeOut', TimeOutValue)
```

Input Arguments

File	Multiple sequence alignment file specified by one of the following: • File name or path and file name • URL pointing to a file • MATLAB character array that contains the text of a multiple sequence alignment file You can read common multiple sequence alignment file types, such as ClustalW (`.aln`), GCG (`.msf`), and PHYLIP.
IgnoreGapsValue	Controls removing gap symbols, such as `'-'` or `'.'`, from the sequences. Choices are `true` or `false` (default).
TimeOutValue	Connection timeout in seconds, specified as a positive scalar. The default value is 5. For details, see here.

Output Arguments

S	MATLAB structure array containing the following fields: • `Header` — Header information from the file. • `Sequence` — Amino acid or nucleotide sequences.
Headers	Cell array containing the header information from the file.
Sequences	Cell array containing the amino acid or nucleotide sequences.

Description

`S = multialignread(File)` reads a multiple sequence alignment file. The file contains multiple sequence lines that start with a sequence header followed by an optional number (not used by `multialignread`) and a section of the sequence. The multiple sequences are broken into blocks with the same number of blocks for every sequence. To view an example multiple sequence alignment file, type `open aagag.aln` at the MATLAB command line.

The output, *S*, is a structure array where `S.Header` contains the header information and `S.Sequence` contains the amino acid or nucleotide sequences.

`[Headers, Sequences] = multialignread(File)` reads the file into separate variables, `Headers` and `Sequences`, which are cell arrays containing header information and amino acid or nucleotide sequences, respectively.

`... = multialignread(File, 'IgnoreGaps', IgnoreGapsValue)` controls the removal of any gap symbol, such as `'-'` or `'.'`, from the sequences. Choices are `true` or `false` (default).

`... = multialignread(File, 'TimeOut', TimeOutValue)` sets the connection timeout (in seconds) to read data from a remote file or URL.

Examples

Read a multiple sequence alignment of the gag polyprotein for several HIV strains.

```
gagaa = multialignread('aagag.aln')

gagaa = 

1x16 struct array with fields:
    Header
    Sequence
```

Version History

Introduced before R2006a

See Also

`fastaread` | `gethmmalignment` | `multialign` | `seqalignviewer` | `multialignwrite` | `seqconsensus` | `seqdisp` | `seqprofile`

multialignwrite

Write multiple alignment to file

Syntax

```
multialignwrite(File, Alignment)

multialignwrite(..., 'Format', FormatValue, ...)
multialignwrite(..., 'Header', HeaderValue, ...)
multialignwrite(..., 'WriteCount', WriteCountValue, ...)
```

Description

`multialignwrite(`*`File`*`, `*`Alignment`*`)` writes the contents of an alignment to a ClustalW ALN-formatted (default) or MSF-formatted file.

`multialignwrite(..., '`*`PropertyName`*`', `*`PropertyValue`*`, ...)` calls `multialignwrite` with optional properties that use property name/property value pairs. You can specify one or more properties in any order. Enclose each *`PropertyName`* in single quotation marks. Each *`PropertyName`* is case insensitive. These property name/property value pairs are as follows:

`multialignwrite(..., 'Format', `*`FormatValue`*`, ...)` specifies the format of the file. *`FormatValue`* can be `'ALN'` (default) or `'MSF'`.

`multialignwrite(..., 'Header', `*`HeaderValue`*`, ...)` specifies the first line of the file. The default for *`HeaderValue`* is `'MATLAB multiple sequence alignment'`.

`multialignwrite(..., 'WriteCount', `*`WriteCountValue`*`, ...)` specifies whether to add the residue counts to the end of each line. *`WriteCountValue`* can be `true` (default) or `false`.

Input Arguments

`Alignment`

An alignment, such as returned by the `multialign` function, represented by a vector of structures, each containing the fields `Header` and `Sequence`.

`File`

Character vector or string specifying either a file name or a path and file name for saving the data. If you specify only a file name, the file is saved to the MATLAB Current Folder browser.

Tip If you use an `.msf` extension when supplying a file name for *`File`*, the data is written to an MSF-formatted file. Otherwise, the data is written to a ClustalW ALN-formatted file.

Below the columns of the ClustalW ALN-formatted file, symbols can appear that denote:

- `*` — Residues or nucleotides in the column are identical in all sequences in the alignment.
- `:` — Conserved substitutions exist in the column for all sequences in the alignment.
- `.` — Semiconserved substitutions exist in the column for all sequences in the alignment.

For more information on these symbols and the groups of residues considered conserved and semiconserved, see section 12 in “Changes since version 1.6” at `https://web.mit.edu/seven/src/clustalw-1.82/README`.

Default:

`FormatValue`

Character vector or string that specifies the format of *`File`*. Choices are `'ALN'` (default) or `'MSF'`.

Tip You can also write to an MSF-formatted file by using an `.msf` extension when supplying a file name for *`File`*.

Default:

`HeaderValue`

Character vector or string that specifies the first line of the file.

Tip Use the `'Header'` property if your file header must be a specific format for a third-party software application.

Default: `'MATLAB multiple sequence alignment'`

`WriteCountValue`

Specifies whether to add the residue counts to the end of each line. Choices are `true` (default) or `false`.

Default:

Examples

1. Use the `fastaread` function to read `p53samples.txt`, a FASTA-formatted file included with the Bioinformatics Toolbox software, which contains seven cellular tumor antigen p53 sequences.

   ```
   p53 = fastaread('p53samples.txt')

   p53 =

   7x1 struct array with fields:
       Header
       Sequence
   ```
2. Use the `multialign` function to align the seven cellular tumor antigen p53 sequences.

   ```
   ma = multialign(p53,'verbose',true);
   ```

3 Write the alignment to a file named `p53.aln`.

```
multialignwrite('p53.aln',ma)
```

Version History

Introduced in R2008b

See Also

`fastaread` | `fastawrite` | `gethmmalignment` | `multialign` | `multialignread` | `seqalignviewer` | `phytreewrite` | `seqconsensus` | `seqdisp` | `seqprofile`

mzcdf2peaks

Convert mzCDF structure to peak list

Syntax

`[Peaklist, Times] = mzcdf2peaks(mzCDFStruct)`

Input Arguments

mzCDFStruct	MATLAB structure containing information from a netCDF file, such as one created by the `mzcdfread` function. Its fields correspond to the variables and global attributes in a netCDF file. If a netCDF variable contains local attributes, an additional field is created, with the name of the field being the variable name appended with `_attributes`. The number and names of the fields will vary, depending on the mass spectrometer software, but typically there are `mass_values` and `intensity_values` fields.

Output Arguments

Peaklist	Either of the following: • Two-column matrix, where the first column contains mass/charge (m/z) values and the second column contains ion intensity values. • Cell array of peak lists, where each element is a two-column matrix of m/z values and ion intensity values, and each element corresponds to a spectrum or retention time.
Times	Scalar of vector of retention times associated with a liquid chromatography/mass spectrometry (LC/MS) or gas chromatography/mass spectrometry (GC/MS) data set. If *Times* is a vector, the number of elements equals the number of peak lists contained in *Peaklist*.

Description

`[Peaklist, Times] = mzcdf2peaks(mzCDFStruct)` extracts peak information from *mzCDFStruct*, a MATLAB structure containing information from a netCDF file, such as one created by the `mzcdfread` function, and creates *Peaklist*, a single matrix or a cell array of matrices containing mass/charge (m/z) values and ion intensity values, and *Times*, a scalar or vector of retention times associated with a liquid chromatography/mass spectrometry (LC/MS) or gas chromatography/mass spectrometry (GC/MS) data set.

mzCDFStruct contains fields that correspond to the variables and global attributes in a netCDF file. If a netCDF variable contains local attributes, an additional field is created, with the name of the field being the variable name appended with `_attributes`. The number and names of the fields will vary, depending on the mass spectrometer software, but typically there are `mass_values` and `intensity_values` fields.

Examples

In the following example, the file `results.cdf` is not provided.

1. Use the `mzcdfread` function to read a netCDF file into the MATLAB software as a structure. Then extract the peak information from the structure.

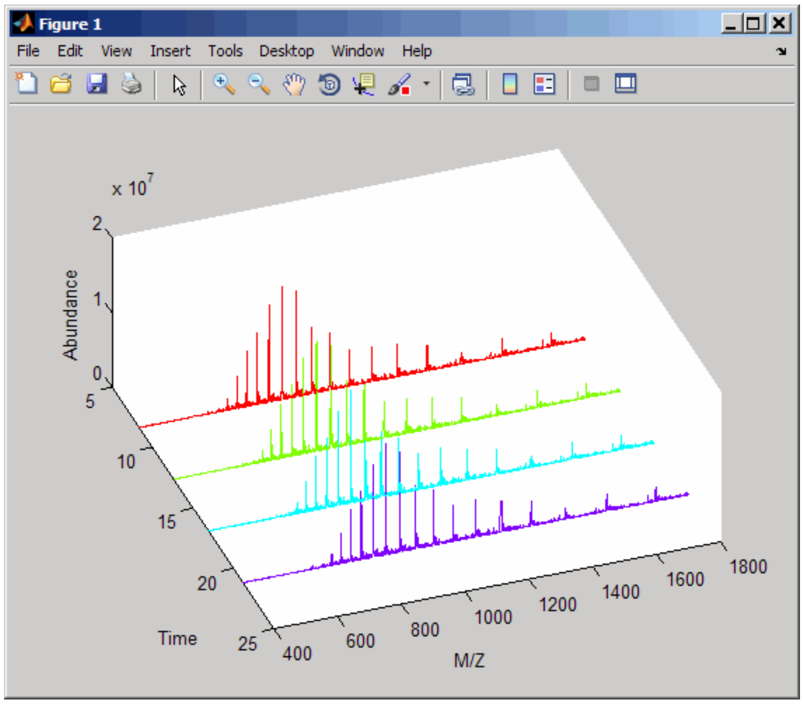
```
mzcdf_struct = mzcdfread('results.cdf');
[peaks,time] = mzcdf2peaks(mzcdf_struct)

peaks = 

    [7008x2 single]
    [7008x2 single]
    [7008x2 single]
    [7008x2 single]

time =

    8.3430
   12.6130
   16.8830
   21.1530
```

2. Create a color map containing a color for each peak list (retention time).

```
colors = hsv(numel(peaks));
```

3. Create a 3-D figure of the peaks and add labels to it.

```
figure
hold on

for i = 1:numel(peaks)
    t = repmat(time(i),size(peaks{i},1),1);
    plot3(t,peaks{i}(:,1),peaks{i}(:,2),'color',colors(i,:))
end

view(70,60)
xlabel('Time')
ylabel(mzcdf_struct.mass_axis_label)
zlabel(mzcdf_struct.intensity_axis_label)
```



Version History

Introduced in R2008b

See Also

`msdotplot` | `mspalign` | `msppresample` | `mzcdfread`

mzcdfinfo

Return information about netCDF file containing mass spectrometry data

Syntax

`InfoStruct` `= mzcdfinfo(`*`File`*`)`

Input Arguments

`File`	Character vector or string containing a file name, or a path and file name, of a netCDF file that contains mass spectrometry data and conforms to the ANDI/MS or the ASTM E2077-00 (2005) standard specification or earlier specifications. If you specify only a file name, that file must be on the MATLAB search path or in the current folder.

Output Arguments

`InfoStruct`	MATLAB structure containing information from a netCDF file. It includes the fields in the following table.

Description

`InfoStruct` `= mzcdfinfo(`*`File`*`)` returns a MATLAB structure, *`InfoStruct`*, containing summary information about a netCDF file, *`File`*.

`File` is a character vector or string containing a file name, or a path and file name, of a netCDF file that contains mass spectrometry data. The file must conform to the ANDI/MS or the ASTM E2077-00 (2005) standard specification or earlier specifications.

`InfoStruct` includes the following fields.

Field	Description
`Filename`	Name of the netCDF file.
`FileTimeStamp`	Date time stamp of the netCDF file.
`FileSize`	Size of the file in bytes.
`NumberOfScans`	Number of scans in the file.
`StartTime`	Run start time.
`EndTime`	Run end time.
`TimeUnits`	Units for time.
`GlobalMassMin`	Minimum m/z value in all scans.
`GlobalMassMax`	Maximum m/z value in all scans.

Field	Description
`GlobalIntensityMin`	Minimum intensity value in all scans.
`GlobalIntensityMax`	Maximum intensity value in all scans.
`ExperimentType`	Indicates if data is raw or centroided.

Note If any of the associated attributes are not in the netCDF file (because they are optional in the specifications), the value for that field will be set to `N/A` or `NaN`.

Examples

In the following example, the file `results.cdf` is not provided.

Return a MATLAB structure containing summary information about a netCDF file.

```
info = mzcdfinfo('results.cdf')

info = 

              Filename: 'results.cdf'
         FileTimeStamp: '19930703134354-700'
              FileSize: 339892
         NumberOfScans: 4
             StartTime: 8.3430
               EndTime: 21.1530
             TimeUnits: 'N/A'
         GlobalMassMin: 399.9990
         GlobalMassMax: 1.8000e+003
    GlobalIntensityMin: NaN
    GlobalIntensityMax: NaN
        ExperimentType: 'Continuum Mass Spectrum'
```

Version History

Introduced in R2008b

See Also

`mzcdfread`

mzcdfread

Read mass spectrometry data from netCDF file

Syntax

```
mzCDFStruct = mzcdfread(File)

mzCDFStruct = mzcdfread(File, ...'TimeRange', TimeRangeValue, ...)
mzCDFStruct = mzcdfread(File, ...'ScanIndices', ScanIndicesValue, ...)
mzCDFStruct = mzcdfread(File, ...'Verbose', VerboseValue, ...)
```

Input Arguments

File	Character vector or string containing a file name, or a path and file name, of a netCDF file that contains mass spectrometry data and conforms to the ANDI/MS or the ASTM E2077-00 (2005) standard specification or earlier specifications. If you specify only a file name, that file must be on the MATLAB search path or in the current folder.
TimeRangeValue	Two-element numeric array `[Start End]` that specifies the time range in *`File`* for which to read spectra. Default is to read spectra from all times `[0 Inf]`. **Tip** Time units are indicated in the netCDF global attributes. For summary information about the time ranges in a netCDF file, use the `mzcdfinfo` function. **Note** If you specify a *`TimeRangeValue`*, you cannot specify a *`ScanIndicesValue`*.
ScanIndicesValue	Positive integer, vector of integers, or a two-element numeric array `[Start_Ind End_Ind]` that specifies a scan, multiple scans, or a range of scans in *`File`* to read. `Start_Ind` and `End_Ind` are each positive integers indicating a scan index number. `Start_Ind` must be less than `End_Ind`. Default is to read all scans. **Tip** For information about the scan indices in a netCDF file, check the `NumberOfScans` field in the structure returned by the `mzcdfinfo` function. **Note** If you specify a *`ScanIndicesValue`*, you cannot specify *`TimeRangeValue`*.
VerboseValue	Controls the display of the progress of the reading of *`File`*. Choices are `true` (default) or `false`.

Output Arguments

`mzCDFStruct`	MATLAB structure containing mass spectrometry information from a netCDF file. Its fields correspond to the variables and global attributes in a netCDF file. If a netCDF variable contains local attributes, an additional field is created, with the name of the field being the variable name appended with `_attributes`. The number and names of the fields will vary, depending on the mass spectrometer software, but typically there are `mass_values` and `intensity_values` fields.

Description

`mzCDFStruct` `= mzcdfread(`*`File`*`)` reads a netCDF file, *`File`*, and then creates a MATLAB structure, *`mzCDFStruct`*.

`File` is a character vector or string containing a file name, or a path and file name, of a netCDF file that contains mass spectrometry data. The file must conform to the ANDI/MS or the ASTM E2077-00 (2005) standard specification or earlier specifications.

`mzCDFStruct` contains fields that correspond to the variables and global attributes in a netCDF file. If a netCDF variable contains local attributes, an additional field is created, with the name of the field being the variable name appended with `_attributes`. The number and names of the fields will vary, depending on the mass spectrometer software, but typically there are `mass_values` and `intensity_values` fields.

Tip LC/MS data analysis requires extended amounts of memory from the operating system.

- If you receive errors related to memory, try the following:
  - Increase the virtual memory (swap space) for your operating system as described in “Resolve “Out of Memory” Errors”.
- If you receive errors related to Java heap space, increase your Java heap space:
  - If you have MATLAB version 7.10 (R2010a) or later, see “Java Heap Memory Preferences”.
  - If you have MATLAB version 7.9 (R2009b) or earlier, see https://www.mathworks.com/matlabcentral/answers/92813-how-do-i-increase-the-heap-space-for-the-java-vm-in-matlab.

`mzCDFStruct` `= mzcdfread(`*`File`*`, ...'`*`PropertyName`*`',` *`PropertyValue`*`, ...)` calls `mzcdfread` with optional properties that use property name/property value pairs. You can specify one or more properties in any order. Each *`PropertyName`* must be enclosed in single quotation marks and is case insensitive. These property name/property value pairs are as follows:

`mzCDFStruct` `= mzcdfread(`*`File`*`, ...'TimeRange',` *`TimeRangeValue`*`, ...)` specifies the range of time in *`File`* to read. *`TimeRangeValue`* is a two-element numeric array `[Start End]`. Default is to read spectra from all times `[0 Inf]`.

Tip Time units are indicated in the netCDF global attributes. For summary information about the time ranges in a netCDF file, use the `mzcdfinfo` function.

Note If you specify a *`TimeRangeValue`*, you cannot specify *`ScanIndicesValue`*.

`mzCDFStruct` `= mzcdfread(`*`File`*`, ...'ScanIndices', `*`ScanIndicesValue`*`, ...)` specifies a scan, multiple scans, or range of scans in *`File`* to read. *`ScanIndicesValue`* is a positive integer, vector of integers, or a two-element numeric array `[Start_Ind End_Ind]`. `Start_Ind` and `End_Ind` are each positive integers indicating a scan index number. `Start_Ind` must be less than `End_Ind`. Default is to read all scans.

Tip For information about the scan indices in a netCDF file, check the `NumberOfScans` field in the structure returned by the `mzcdfinfo` function.

Note If you specify a *`ScanIndicesValue`*, you cannot specify a *`TimeRangeValue`*.

`mzCDFStruct` `= mzcdfread(`*`File`*`, ...'Verbose', `*`VerboseValue`*`, ...)` controls the progress display when reading *`File`*. Choices are `true` (default) or `false`.

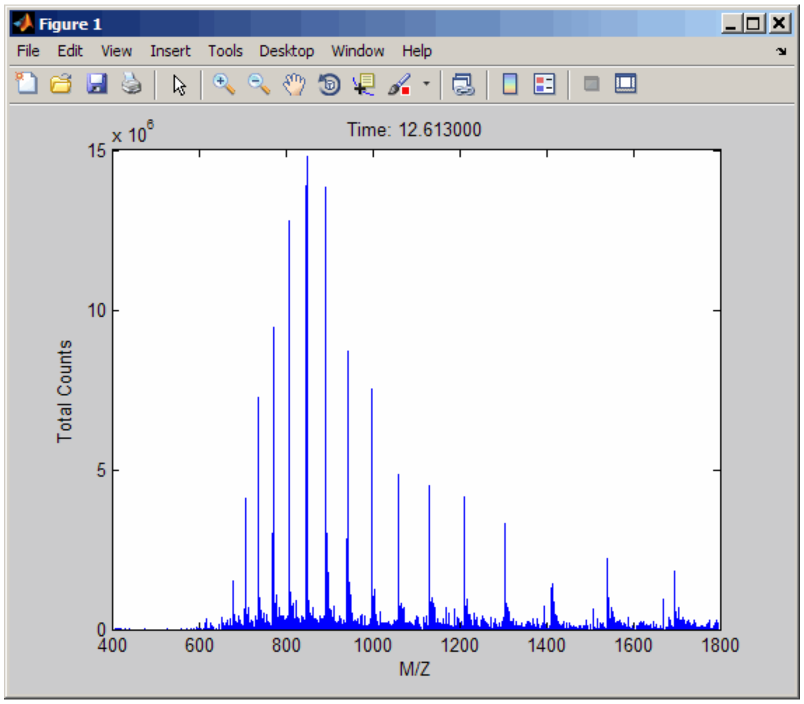
Examples

In the following example, the file `results.cdf` is not provided.

1. Read a netCDF file into the MATLAB software as a structure.

   ```
   out = mzcdfread('results.cdf');
   ```
2. View the second scan in the netCDF file by creating separate variables containing the intensity and m/z values, and then plotting these values. Add a title and *x*- and *y*-axis labels using fields in the output structure.

   ```
   idx1 = out.scan_index(2)+1;
   idx2 = out.scan_index(3);
   y = out.intensity_values(idx1:idx2);
   z = out.mass_values(idx1:idx2);
   stem(z,y,'marker','none')

   title(sprintf('Time: %f',out.scan_acquisition_time(2)))
   xlabel(out.mass_axis_units)
   ylabel(out.intensity_axis_units)
   ```



Version History

Introduced in R2008b

See Also

`jcampread` | `mzcdf2peaks` | `mzcdfinfo` | `mzxmlread` | `tgspcread`

mzxml2peaks

Convert mzXML structure to peak list

Syntax

`[Peaklist, Times] = mzxml2peaks(mzXMLStruct)`

`[Peaklist, Times] = mzxml2peaks(mzXMLStruct, 'Levels', LevelsValue)`

Input Arguments

mzXMLStruct	MATLAB structure containing information from an mzXML file, such as one created by the `mzxmlread` function. It includes the fields shown in the table below.
LevelsValue	Positive integer or vector of integers that specifies the level(s) of spectra in *mzXMLStruct* to convert, assuming the spectra are from tandem MS data sets. Default is `1`, which converts only the first-level spectra, that is, spectra containing precursor ions. Setting *LevelsValue* to `2` converts only the second-level spectra, which are the fragment spectra (created from a precursor ion).

Output Arguments

Peaklist	Either of the following: • Two-column matrix, where the first column contains mass/charge (m/z) values and the second column contains ion intensity values. • Cell array of peak lists, where each element is a two-column matrix of m/z values and ion intensity values, and each element corresponds to a spectrum or retention time.
Times	Vector of retention times associated with a liquid chromatography/mass spectrometry (LC/MS) or gas chromatography/mass spectrometry (GC/MS) data set. The number of elements in *Times* equals the number of elements in *Peaklist*.

Description

`[Peaklist, Times] = mzxml2peaks(mzXMLStruct)` extracts peak information from *mzXMLStruct*, a MATLAB structure containing information from an mzXML file, such as one created by the `mzxmlread` function, and creates *Peaklist*, a cell array of matrices containing mass/charge (m/z) values and ion intensity values, and *Times*, a vector of retention times associated with a liquid chromatography/mass spectrometry (LC/MS) or gas chromatography/mass spectrometry (GC/MS) data set. *mzXMLStruct* includes the following fields:

Field	Description
`scan`	Structure array containing the data pertaining to each individual scan, such as mass spectrometry level, total ion current, polarity, precursor mass (when it applies), and the spectrum data.
`index`	Structure containing indices to the positions of scan elements in the XML document.
`mzXML`	Structure containing: • Information in the root element of the mzXML schema, such as instrument details, experiment details, and preprocessing method • URLs pointing to schemas for the individual scans • Indexing approach • Digital signature calculated for the current instance of the document

`[`*`Peaklist`*`,` *`Times`*`] = mzxml2peaks(`*`mzXMLStruct`*`, 'Levels',` *`LevelsValue`*`)` specifies the level(s) of the spectra in *`mzXMLStruct`* to convert, assuming the spectra are from tandem MS data sets. Default is `1`, which converts only the first-level spectra, that is, spectra containing precursor ions. Setting *`LevelsValue`* to `2` converts only the second-level spectra, which are the fragment spectra (created from a precursor ion).

Examples

Note In the following example, the file `results.mzxml` is not provided. Sample mzXML files can be found at:

- The Sashimi Project
- Peptide Atlas Repository at the Institute for Systems Biology (ISB)

1. Use the `mzxmlread` function to read an mzXML file into the MATLAB software as structure. Then extract the peak information of only the first-level ions from the structure.

   ```
   mzxml_struct = mzxmlread('results.mzxml');
   [peaks,time] = mzxml2peaks(mzxml_struct);
   ```
2. Create a dot plot of the LC/MS data.

   ```
   msdotplot(peaks,time)
   ```

Version History

Introduced in R2007a

See Also

`msdotplot` | `mspalign` | `msppresample` | `mzxmlread`

mzxmlinfo

Return information about mzXML file

Syntax

`InfoStruct = mzxmlinfo(File)`

`InfoStruct = mzxmlinfo(File, 'NumOfLevels', NumOfLevelsValue)`

Input Arguments

File	Character vector or string containing a file name, or a path and file name, of an mzXML file that conforms to the mzXML 2.1 specification or earlier specifications. If you specify only a file name, that file must be on the MATLAB search path or in the current folder.
NumOfLevelsValue	Controls the return of `NumOfLevels`, an additional field in *InfoStruct*, that contains the number of mass spectrometry (MS) levels of spectra in *File*. Choices are `true` or `false` (default).

Output Arguments

InfoStruct	MATLAB structure containing information from an mzXML file. It includes the fields shown in the table below.

Description

InfoStruct = `mzxmlinfo(`*File*`)` returns a MATLAB structure, *InfoStruct*, containing summary information about an mzXML file, *File*.

File is a character vector or string containing a file name, or a path and file name, of an mzXML file. The file must conform to the mzXML 2.1 specification or earlier specifications. You can view the mzXML 2.1 specification at:

`http://sashimi.sourceforge.net/schema_revision/mzXML_2.1/Doc/mzXML_2.1_tutorial.pdf`

InfoStruct includes the following fields.

Field	Description
`Filename`	Name of the mzXML file.
`FileModDate`	Modification date of the file.
`FileSize`	Size of the file in bytes.
`NumberOfScans`	Number of scans in the file.*
`StartTime`	Run start time.*

Field	Description
`EndTime`	Run end time.*
`DataProcessingIntensityCutoff`	Minimum mass/charge (m/z) intensity value.*
`DataProcessingCentroided`	Indicates if data is centroided.*
`DataProcessingDeisotoped`	Indicates if data is deisotoped.*
`DataProcessing ChargeDeconvoluted`	Indicates if data is deconvoluted.*
`DataProcessingSpotIntegration`	For LC/MALDI experiments, indicates if peaks eluting over multiple spots have been integrated into a single spot.*

* — These fields contain `N/A` if the mzXML file does not include the associated attributes. The associated attributes are optional in the mzXML file, per the mzXML 2.1 specification.

`*InfoStruct* = mzxmlinfo(*File*, 'NumOfLevels', *NumOfLevelsValue*)` controls the return of `NumOfLevels`, an additional field in *`mzXMLInfo`*, that contains the number of mass spectrometry levels of spectra in *`File`*. Choices are `true` or `false` (default).

Examples

Note In the following example, the file `results.mzxml` is not provided. Sample mzXML files can be found at:

- The Sashimi Project
- Peptide Atlas Repository at the Institute for Systems Biology (ISB)

Return a MATLAB structure containing summary information about an mzXML file.

```
info = mzxmlinfo('results.mzxml');

info =

                         Filename: 'results.mzxml'
                      FileModDate: '07-May-2008 13:39:12'
                         FileSize: 10607
                    NumberOfScans: 2
                        StartTime: 'PT0.00683333S'
                          EndTime: 'PT200.036S'
    DataProcessingIntensityCutoff: 'N/A'
         DataProcessingCentroided: 'false'
         DataProcessingDeisotoped: 'N/A'
 DataProcessingChargeDeconvoluted: 'N/A'
    DataProcessingSpotIntegration: 'N/A'
```

Return a MATLAB structure containing summary information, including the number of mass spectrometry levels, about an mzXML file.

```
info = mzxmlinfo('results.mzxml','numoflevels',true);
info = 

                          Filename: 'results.mzxml'
                       FileModDate: '07-May-2008 13:39:12'
                          FileSize: 10607
                     NumberOfScans: 2
                         StartTime: 'PT0.00683333S'
                           EndTime: 'PT200.036S'
     DataProcessingIntensityCutoff: 'N/A'
          DataProcessingCentroided: 'false'
          DataProcessingDeisotoped: 'N/A'
  DataProcessingChargeDeconvoluted: 'N/A'
     DataProcessingSpotIntegration: 'N/A'
                  NumberOfMSLevels: 2
```

Version History

Introduced in R2008b

See Also

`mzxmlread`

mzxmlread

Read data from mzXML file

Syntax

```
mzXMLStruct = mzxmlread(myFile)
mzXMLStruct = mzxmlread(myFile,Name,Value)
```

Description

`mzXMLStruct = mzxmlread(myFile)` returns a MATLAB structure, `mzXMLStruct`, from an mzXML file, `myFile`.

`mzXMLStruct = mzxmlread(myFile,Name,Value)` reads an mzXML file, `myFile`, and then returns a MATLAB structure, `mzXMLStruct`, using additional options specified by one or more `Name,Value` pair arguments.

Examples

Create a MATLAB Structure from an mzXML File

In this example, the file `results_1.mzxml` is not provided. You can find sample mzXML files at:

- The Sashimi Project
- Peptide Atlas Repository at the Institute for Systems Biology (ISB)

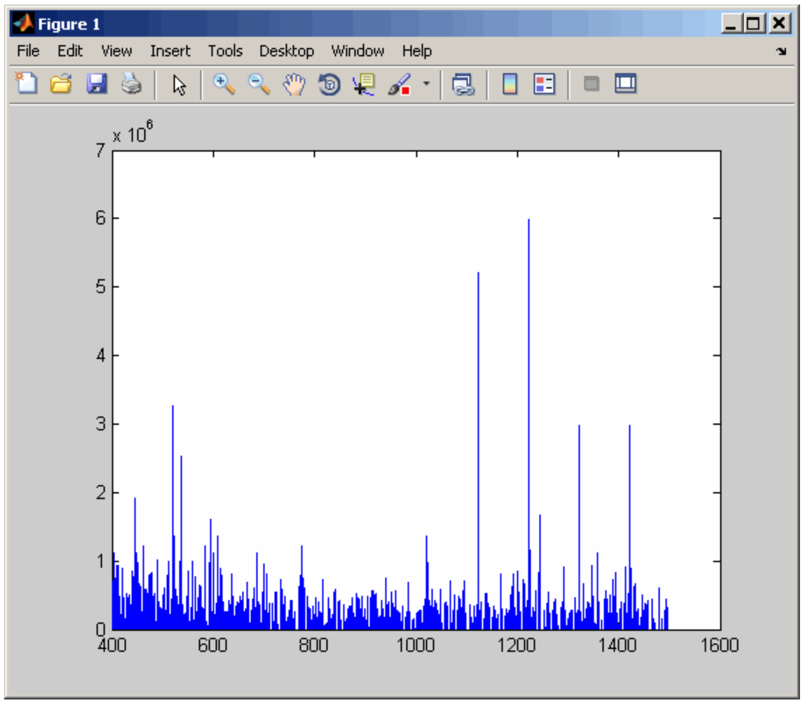
Read an mzXML file into a MATLAB structure.

```
out = mzxmlread('results_1.mzxml')
```

```
out = 

     scan: [2000x1 struct]
    mzXML: [1x1 struct]
    index: [1x1 struct]
```

View the first scan in the mzXML file by creating separate variables containing the mass-to-charge ratio (`mz_ratio`) and intensity (`Y`) values respectively. Then plot these values.

```
mz_ratio = out.scan(1).peaks.mz(1:2:end);
Y = out.scan(1).peaks.mz(2:2:end);
stem(mz_ratio,Y,'marker','none')
```



Extract One or Multiple Scans from an mzXML Structure

In this example, the file `results_2.mzxml` is not provided. You can find sample mzXML files at:

- The Sashimi Project
- Peptide Atlas Repository at the Institute for Systems Biology (ISB)

Read an mzXML file into a MATLAB structure, extracting a scan at index 1000.

```
out1 = mzxmlread('results_2.mzxml','ScanIndices',1000)
```

```
out1 = 

     scan: [1x1 struct]
    mzXML: [1x1 struct]
    index: [1x1 struct]
```

Read an mzXML file into a MATLAB structure, extracting multiple scans at indices 1000, 1500, and 2000.

```
out2 = mzxmlread('results_2.mzxml','ScanIndices',[1000 1500 2000])
```

```
out2 = 

     scan: [3x1 struct]
    mzXML: [1x1 struct]
    index: [1x1 struct]
```

Read an mzXML file into a MATLAB structure, extracting a range of scans from indices 1000 to 2000.

```
out3 = mzxmlread('results_2.mzxml','ScanIndices',[1000:2000])
```

```
out3 = 

    scan: [1001x1 struct]
    mzXML: [1x1 struct]
    index: [1x1 struct]
```

Input Arguments

`myFile` — Input file
character vector | string

Input file, specified as a character vector or string containing an mzXML file name. The file must conform to the mzXML 2.1 or earlier specifications. You can read the mzXML 2.1 specification here:

http://sashimi.sourceforge.net/schema_revision/mzXML_2.1/Doc/mzXML_2.1_tutorial.pdf

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `'TimeRange',[5.0 10.0],'Verbose',true`

`Levels` — Spectra levels
positive integer | vector of integers

Spectra levels, specified as a positive integer or vector of integers indicating which scans to extract scans from `myFile`. By default, `mzxmlread` reads all spectra levels.

For summary information about the levels of spectra in an mzXML file, use the `mzxmlinfo` function.

If you are using the `'Levels'` name-value pair argument, then you cannot use `'ScanIndices'`.

Example: `'Levels',5`

`TimeRange` — Range of time
two-element numeric array

Range of time, specified as a two-element numeric array, such as `[Start End]` indicating which scans to extract from `myFile`. The `Start` and `End` scalar values must be between the `startTime` and `endTime` attributes of the `msRun` element in `myFile`. The `Start` scalar value must be less than `End`. By default, `mzxmlread` reads all scans.

For summary information about the time ranges in an mzXML file, use the `mzxmlinfo` function.

If you are using `'TimeRange'` name-value pair argument, then you cannot use `'ScanIndices'`.

Example: `'TimeRange',[5.1 10.2]`

`ScanIndices` — Scan indices
positive integer | vector of positive integers

Scan indices, specified as a positive integer or vector of positive integers indicating which scans to extract from `myFile`. Use an integer to specify a single scan, or a vector of integers to specify multiple scans. By default, `mzxmlread` reads all scans.

For summary information about the time ranges in an mzXML file, use the `mzxmlinfo` function.

If you are using the `'ScanIndices'` name-value pair argument, then you cannot use `'Levels'` or `'TimeRange'`.

Example: `'ScanIndices',7000`

`Verbose` — Verbose mode
`false` (default) | `0` | `true` | `1`

Verbose mode, specified as `true` (`1`), or `false` (`0`). When `'Verbose'` is set to `true`, `mzxmlread` displays the progress while reading `myFile`.

Example: `'Verbose',true`

Output Arguments

`mzXMLStruct` — Structure from mzXML file
MATLAB structure

Structure from an mzXML file, returned as a MATLAB structure. `mzXMLStruct` has the following fields:

Field	Description
`scan`	Structure array containing the data pertaining to each individual scan, such as mass spectrometry level, total ion current, polarity, precursor mass (when it applies), and the spectrum data.
`index`	Structure containing indices to the positions of scan elements in the XML document.
`mzXML`	Structure containing all of the following: • Information in the root element of the mzXML schema, such as instrument details, experiment details, and preprocessing methods • URLs pointing to schemas for each scan • Indexing approach • Digital signature calculated for the current instance of the document

Tips

LC/MS data analysis requires extended amounts of memory from the operating system.

- If you receive errors related to memory, try the following:
  - Increase the virtual memory (swap space) for your operating system as described in “Resolve “Out of Memory” Errors”.
- If you receive errors related to Java heap space, increase your Java heap space:
  - If you have MATLAB version 7.10 (R2010a) or later, see “Java Heap Memory Preferences”.

- If you have MATLAB version 7.9 (R2009b) or earlier, see https://www.mathworks.com/matlabcentral/answers/92813-how-do-i-increase-the-heap-space-for-the-java-vm-in-matlab.

Version History

Introduced in R2006b

See Also

`jcampread` | `mzxml2peaks` | `mzxmlinfo` | `tgspcread` | `xmlread`

nbintest

Unpaired hypothesis test for count data with small sample sizes

Syntax

```
test = nbintest(X,Y)
test = nbintest(X,Y,Name,Value)
```

Description

`test = nbintest(X,Y)` performs a hypothesis test that two independent samples of short-read count data, in each row of X and Y, come from distributions with equal means under the assumptions that:

- Short-read counts are modeled using the negative binomial distribution.
- Variance and mean of data in each row are linked through a regression function along all the rows.

`X` and `Y` must have the same number of rows and at least 2 columns, but not necessarily the same number of columns. Rows of `X` and `Y` correspond to variables, features, or genes, such as measurements of gene expression for different genes. Columns are usually time points or patients.

`test` is a `NegativeBinomialTest` object with two-sided p-values stored in the `pValue` property.

Use this function when you want to perform an unpaired hypothesis test for short-read count data (from high-throughput assays such as RNA-Seq or ChIP-Seq) with small sample sizes (in the order of tens at most). For instance, use this function to decide if observed differences in read counts between two conditions are significant for given genes.

`test = nbintest(X,Y,Name,Value)` uses additional options specified by one or more `Name,Value` pair arguments.

Note It is recommended that you use the diagnostic plots of the `NegativeBinomialTest` object returned by `nbintest` before interpreting the p-values. These plots allow you to see if the model assumption is correct, and the variance link used is appropriate for the data.

Examples

Perform unpaired hypothesis test for short-read count data

This example shows how to perform an unpaired hypothesis test for synthetic short-read count data from two different biological conditions.

The data in this example contains synthetic gene count data for 5000 genes, representing two different biological conditions, such as diseased and normal cells. For each condition, there are five samples. Only 10% of the genes (500 genes) are differentially expressed. Specifically, half of them (250 genes) are exactly 3-fold overexpressed. The other 250 genes are 3-fold underexpressed. The rest of the gene expression data is generated from the same negative binomial distribution for both conditions. Each sample also has a different size factor (that is, the coverage or sampling depth).

Load the data.

```
load('nbintest_data.mat','K','H0');
```

The variable K contains gene count data. The rows represent genes, and the columns represent samples. In this case, the first five columns represent samples from the first condition. The other five columns represent samples from the second condition. Display the first few rows of K.

```
K(1:5,:)
```

```
ans = 5×10

       13683        14140         8281        14309        12208         8045         9446        11317
       16028        16805         9813        16486        14076         9901        10927        13348
         814          862          492          910          758          521          573          753
       15870        16453         9857        16454        14267         9671        10997        13624
        9422         9393         5734         9598         8174         5381         6315         7752
```

In this example, the null hypothesis is true when the gene is not differentially expressed. The variable H0 contains boolean indicators that indicate for which genes the null hypothesis is true (marked as 1). In other words, H0 contains known labels that you will use later to compare with predicted results.

```
sum(H0)
```

```
ans = 4500
```

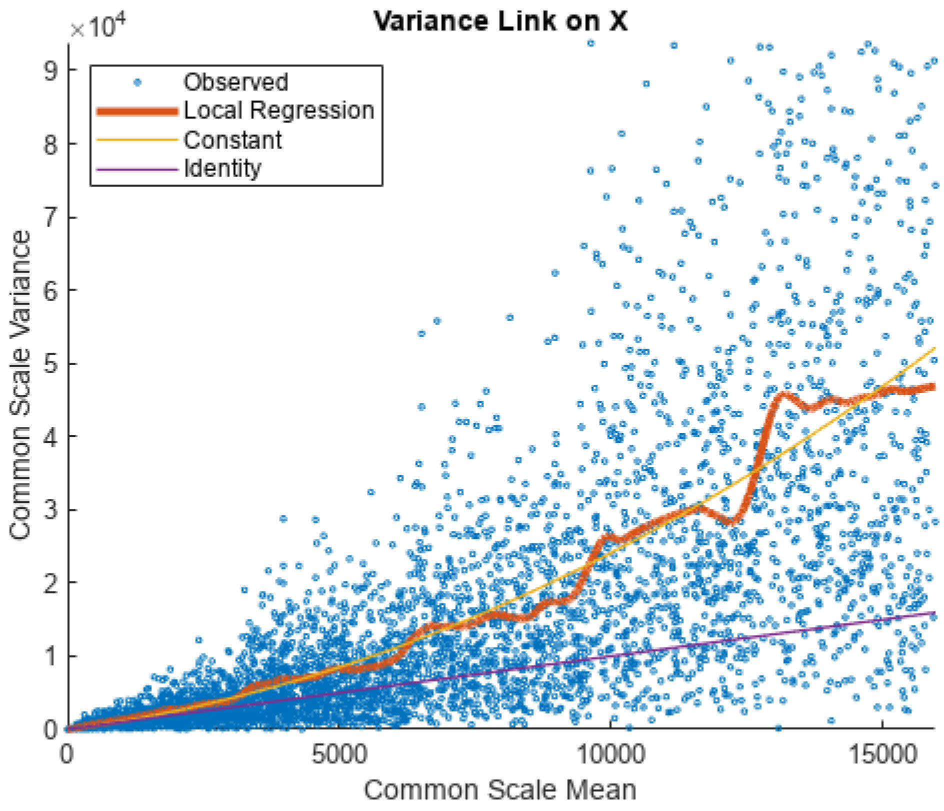
Out of 5000 genes, 4500 are not differentially expressed in this synthetic data.

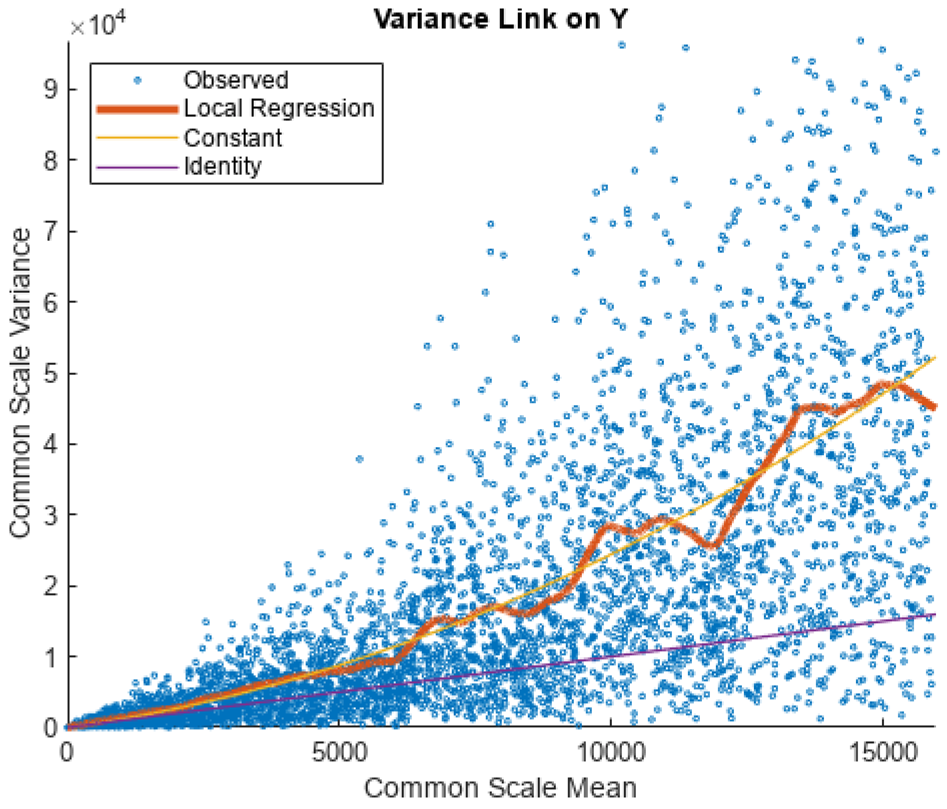
Run an unpaired hypothesis test for samples from two conditions using `nbintest`. The assumption is that the data came from a negative binomial distribution, where the variance is linked to the mean via a locally-regressed smooth function of the mean as described in [1] by setting `'VarianceLink'` to `'LocalRegression'`.

```
tLocal = nbintest(K(:,1:5),K(:,6:10),'VarianceLink','LocalRegression');
```

Use `plotVarianceLink` to plot a scatter plot for each experimental condition (for X and Y conditions), with the sample variance on the common scale versus the estimate of the condition-dependent mean. Use a linear scale for both axes. Include curves for all other linkage options by setting `'Compare'` to `true`.

```
plotVarianceLink(tLocal,'Scale','linear','Compare',true)
```

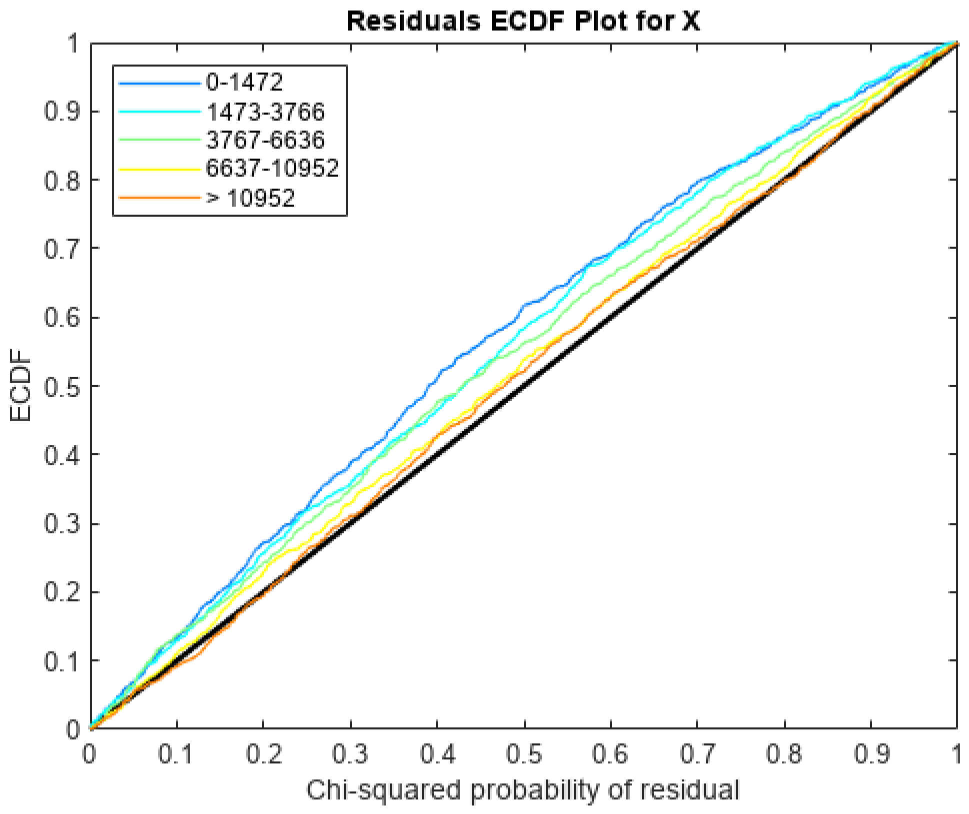

Variance Link on X
×10^4
Observed
Local Regression
Constant
Identity
Common Scale Variance
Common Scale Mean
0
1
2
3
4
5
6
7
8
9
0
5000
10000
15000

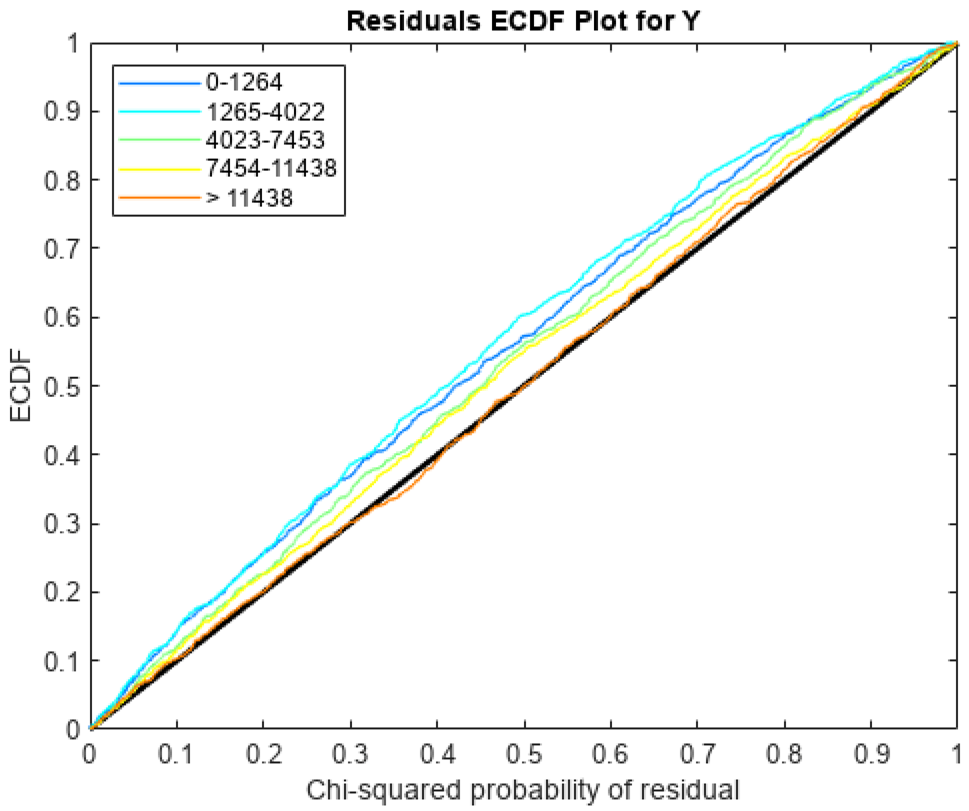


The `Identity` line represents the Poisson model, where the variance is identical to the mean as described in [3]. Observe that the data seems to be overdispersed (that is, most points are above the `Identity` line). The `Constant` line represents the negative binomial model, where the variance is the sum of the shot noise term (mean) and a constant multiplied by the squared mean as described in [2]. The `Local Regression` and `Constant` linkage options appear to fit better with the overdispersed data.

Use `plotChiSquaredFit` to assess the goodness-of-fit for variance regression. It plots the empirical CDF (ecdf) of the chi-squared probabilities. The probabilities are the ratio between the observed and the estimated variance stratified by short-read count levels into five equal-sized bins.

```
plotChiSquaredFit(tLocal)
```


Residuals ECDF Plot for X
0-1472
1473-3766
3767-6636
6637-10952
> 10952
ECDF
Chi-squared probability of residual
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1



Each figure shows five ecdf curves. Each curve represents one of the five short-read count levels. For instance, the blue line represents the ecdf curve for a low short-read counts between 0 and 1264. The red line represents high counts (more than 11438).

One way to interpret the curves is to check if the ecdf curves are above the diagonal line. If they are above the line, then the variance is overestimated. If they are below the line, then the variance is underestimated. In both figures, the variance seems to be correctly estimated for higher counts (that is, the red line follows the diagonal line), but slightly overestimated for lower count levels.

To assess the performance of the hypothesis test, construct a confusion matrix using the known labels and the predicted p-values.

```
confusionmat(H0,(tLocal.pValue > .001))
```

```
ans = 2×2

         493            7
           5         4495
```

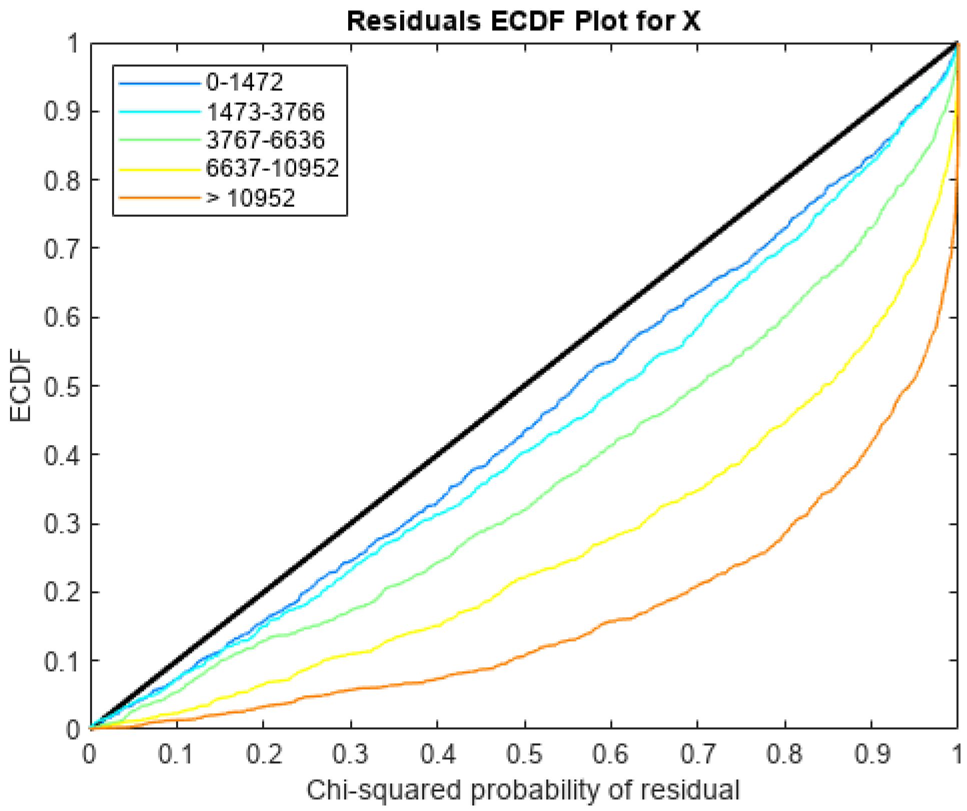
Out of 500 differentially expressed genes, 493 are correctly predicted (true positives) and 7 of them are incorrectly predicted as not-differentially expressed genes (false negatives). Out of 4500 genes that are not differentially expressed, 4495 are correctly predicted (true negatives) and 5 of them are incorrectly predicted as differentially expressed genes (false positives).

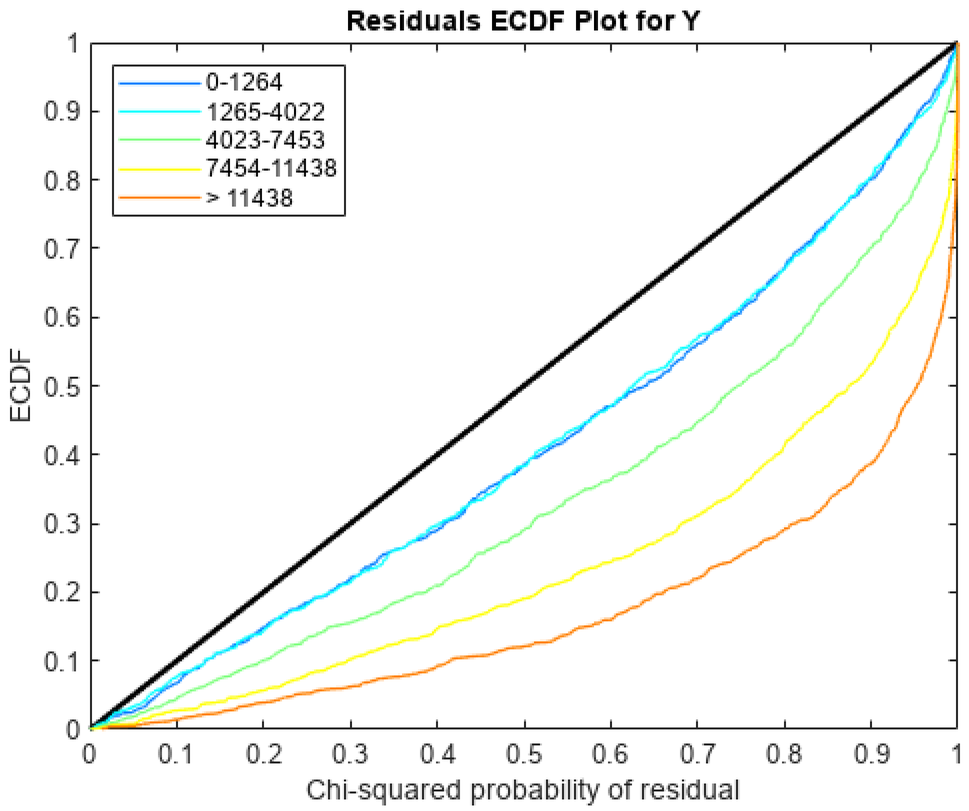
For a comparison, run the hypothesis test again assuming that counts are modeled by the Poisson distribution, where the variance is identical to the mean.

```
tPoisson = nbintest(K(:,1:5),K(:,6:10),'VarianceLink','Identity');
```

Plot the ecdf curves. Observe that all the curves are below the diagonal line, implying that the variance is underestimated. Therefore, the negative binomial model fits the data better.

```
plotChiSquaredFit(tPoisson)
```





Input Arguments

X — Gene expression values from the first experimental condition
matrix | table

Gene expression values from the first experimental condition, specified as a matrix or table. For instance, X can represent gene expression values from cancer cells.

Note X and Y must have the same number of rows and at least 2 columns, but not necessarily the same number of columns. Rows of X and Y correspond to genes (or features), such as measurements of gene expression for different genes. Columns are usually time points or patients.

Y — Gene expression values from the second experimental condition
matrix | table

Gene expression values from the second experimental condition, specified as a matrix or table. For instance, Y can represent gene expression values from normal cells.

Note X and Y must have the same number of rows and at least 2 columns, but not necessarily the same number of columns. Rows of X and Y correspond to genes (or features), such as measurements of gene expression for different genes. Columns are time points or patients.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `'VarianceLink','Identity'` specifies that the variance is equal to the mean when defining the linkage between the two.

`VarianceLink` — Linkage type between the variance and mean

`'LocalRegression'` (default) | `'Constant'` | `'Identity'`

Linkage type between the variance and mean, specified as a comma-separated pair consisting of `'VarianceLink'` and a character vector or string. This table summarizes the available linkage options.

Linkage Option	Description
`'LocalRegression'`	The variance is the sum of the shot noise term (mean) and a locally regressed nonparametric smooth function of the mean as described in [1]. This option is the default. Use this option if your data is overdispersed and has more than 1000 rows (genes).
`'Constant'`	The variance is the sum of the shot noise term (mean) and a constant multiplied by the squared mean as described in [2]. This method uses all the rows in the data to estimate the constant. Use this option if your data is overdispersed and has less than 1000 rows.
`'Identity'`	The variance is equal to the mean as described in [3]. Counts are therefore modeled by the Poisson distribution individually for each row of `X` and `Y`. Use this option if your data has few genes and the regression between the variance and mean is not possible because of very small number of samples or replicates. This option is not recommended for overdispersed data.

Example: `'VarianceLink','Constant'`

`PooledVariance` — Logical flag to pool variance across both conditions

`false` (default) | `true`

Logical flag to pool variance across both conditions, specified as `true` or `false`. By default, the variance is estimated separately for each condition.

Example: `'PooledVariance',true`

`SizeFactor` — Size (scaling) factor of each column in X and Y

`[]` (default) | cell array of two vectors

Size (scaling) factor of each column in `X` and `Y`, specified as a cell array of two vectors such as `{SX,SY}`. `SX` and `SY` are numeric vectors with sizes equal to `size(X,2)` and `size(Y,2)`. `SX` , `SY`, or both can be a scalar indicating that all columns share the same size factor.

In a high-throughput sequencing library, the size factor is an estimation of the coverage or the sampling depth. The default is an empty array `[]`, meaning the size factor is estimated as the median of the ratio of the sample's counts to the geometric mean of each row in `X` or `Y`. Rows with zero geometric mean are ignored.

Example: `'SizeFactor',{[1.2,0.5,0.8],[0.8,1.1,1.5]}`

Output Arguments

`test` — Hypothesis test results
`NegativeBinomialTest` object

Hypothesis test results, returned as a `NegativeBinomialTest` object. Use this object to create diagnostic plots and access p-values.

Version History

Introduced in R2014b

References

[1] Anders, S., and Huber, W. (2010). Differential Expression Analysis for Sequence Count Data. Genome Biology, 11(10):R106.

[2] Robinson, M.D., and Smyth, G.K. (2008). Small-sample Estimation of Negative Binomial Dispersion, with Applications to SAGE data. Biostatistics, 9:321-332.

[3] Marioni, J.C., Mason, C.E., Mane, S.M., Stephens, M., and Gilad, Y. (2008). RNA-seq: an Assessment of Technical Reproducibility and Comparison with Gene Expression Arrays. Genome Research, 16:1509-1517.

See Also

`mattest` | `NegativeBinomialTest` | `plotVarianceLink` | `plotChiSquaredFit`

Topics
“Negative Binomial Distribution”

NegativeBinomialTest

Unpaired hypothesis test result

Description

A `NegativeBinomialTest` object, returned by the `nbintest` function, contains the results of an unpaired hypothesis test for short-read count data with small sample sizes. Use this object to access p-values of the test or to create diagnostic plots.

Creation

`nbintest` returns the unpaired hypothesis test result as a `NegativeBinomialTest` object. You cannot construct this object directly.

Properties

`pValue` — Two-sided p-values
column vector

Two-sided p-values, specified as a column vector, for every row of the inputs to `nbintest`.

`VarianceLink` — Linkage type between the variance and mean
`'LocalRegression'` (default) | `'Constant'` | `'Identity'`

This property is read-only.

Linkage type between the variance and mean, specified as a character vector or string. This table summarizes the available linkage options.

Linkage Option	Description
`'LocalRegression'`	The variance is the sum of the shot noise term (mean) and a locally regressed nonparametric smooth function of the mean as described in [1]. This option is the default. Use this option if your data contains several rows (genes), such as more than 1000 rows.
`'Constant'`	The variance is the sum of the shot noise term (mean) and a constant multiplied by the squared mean as described in [2]. This method uses all the rows in the data to estimate the constant. Use this option if your data has fewer rows, that is, less than 1000 rows, and is overdispersed.
`'Identity'`	The variance is equal to the mean as described in [3]. Counts are therefore modeled by the Poisson distribution individually for each row of `X` and `Y`. Use this option to compare the results of the other two options.

`PooledVariance` — Logical flag to pool variance between both conditions
0 (default) | 1

This property is read-only.

Logical flag to pool variance between both conditions, specified as 1 (`true`) or 0 (`false`). The default is 0, meaning the variance is estimated separately for each condition.

`SizeFactors` — Size (scaling) factor of each column in X and Y
cell array of two vectors

This property is read-only.

Size (scaling) factor of each column in `X` and `Y`, specified as a cell array of two vectors, such as `{SX,SY}`. `SX` and `SY` are numeric vectors with sizes equal to `size(X,2)` and `size(Y,2)`.

Note These properties are read-only. Run `nbintest` to change them.

Object Functions

plotVarianceLink	Plot the sample variance versus the estimate of the condition-dependent mean
plotChiSquaredFit	Plot goodness-of-fit for variance regression

Examples

Perform unpaired hypothesis test for short-read count data

This example shows how to perform an unpaired hypothesis test for synthetic short-read count data from two different biological conditions.

The data in this example contains synthetic gene count data for 5000 genes, representing two different biological conditions, such as diseased and normal cells. For each condition, there are five samples. Only 10% of the genes (500 genes) are differentially expressed. Specifically, half of them (250 genes) are exactly 3-fold overexpressed. The other 250 genes are 3-fold underexpressed. The rest of the gene expression data is generated from the same negative binomial distribution for both conditions. Each sample also has a different size factor (that is, the coverage or sampling depth).

Load the data.

```
load('nbintest_data.mat','K','H0');
```

The variable `K` contains gene count data. The rows represent genes, and the columns represent samples. In this case, the first five columns represent samples from the first condition. The other five columns represent samples from the second condition. Display the first few rows of `K`.

```
K(1:5,:)
```

```
ans = 5×10

       13683        14140         8281        14309        12208         8045         9446        11317
       16028        16805         9813        16486        14076         9901        10927        13348
         814          862          492          910          758          521          573          753
       15870        16453         9857        16454        14267         9671        10997        13624
        9422         9393         5734         9598         8174         5381         6315         7752
```

In this example, the null hypothesis is true when the gene is not differentially expressed. The variable `H0` contains boolean indicators that indicate for which genes the null hypothesis is true (marked as 1). In other words, H0 contains known labels that you will use later to compare with predicted results.

```
sum(H0)
```

```
ans = 4500
```

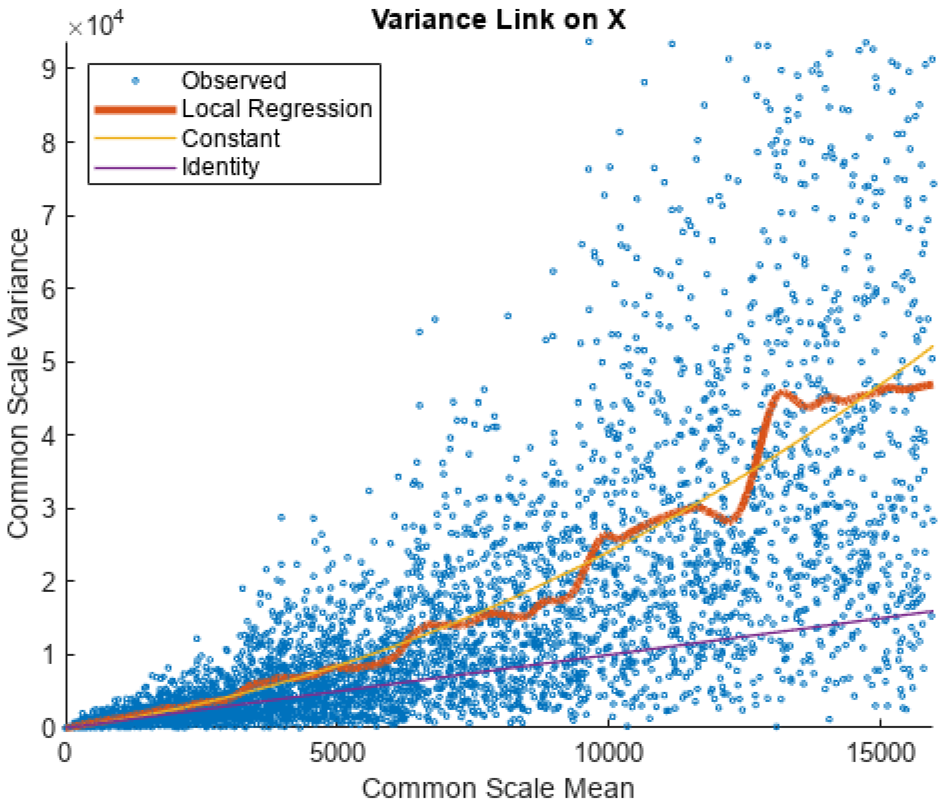
Out of 5000 genes, 4500 are not differentially expressed in this synthetic data.

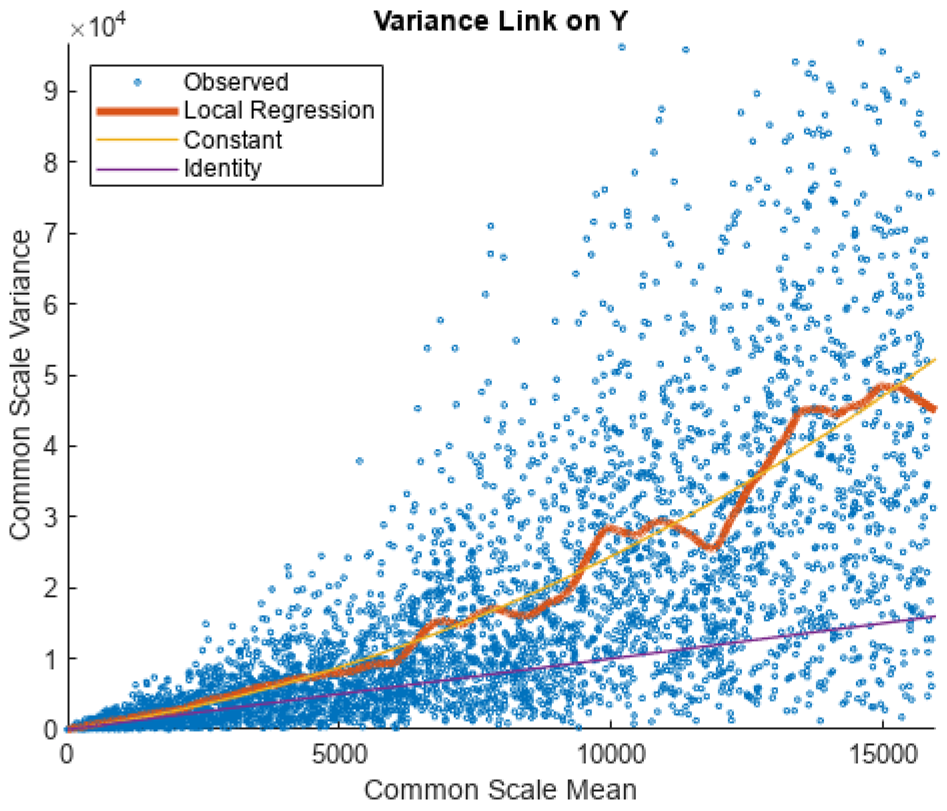
Run an unpaired hypothesis test for samples from two conditions using `nbintest`. The assumption is that the data came from a negative binomial distribution, where the variance is linked to the mean via a locally-regressed smooth function of the mean as described in [1] by setting `'VarianceLink'` to `'LocalRegression'`.

```
tLocal = nbintest(K(:,1:5),K(:,6:10),'VarianceLink','LocalRegression');
```

Use `plotVarianceLink` to plot a scatter plot for each experimental condition (for X and Y conditions), with the sample variance on the common scale versus the estimate of the condition-dependent mean. Use a linear scale for both axes. Include curves for all other linkage options by setting `'Compare'` to `true`.

```
plotVarianceLink(tLocal,'Scale','linear','Compare',true)
```

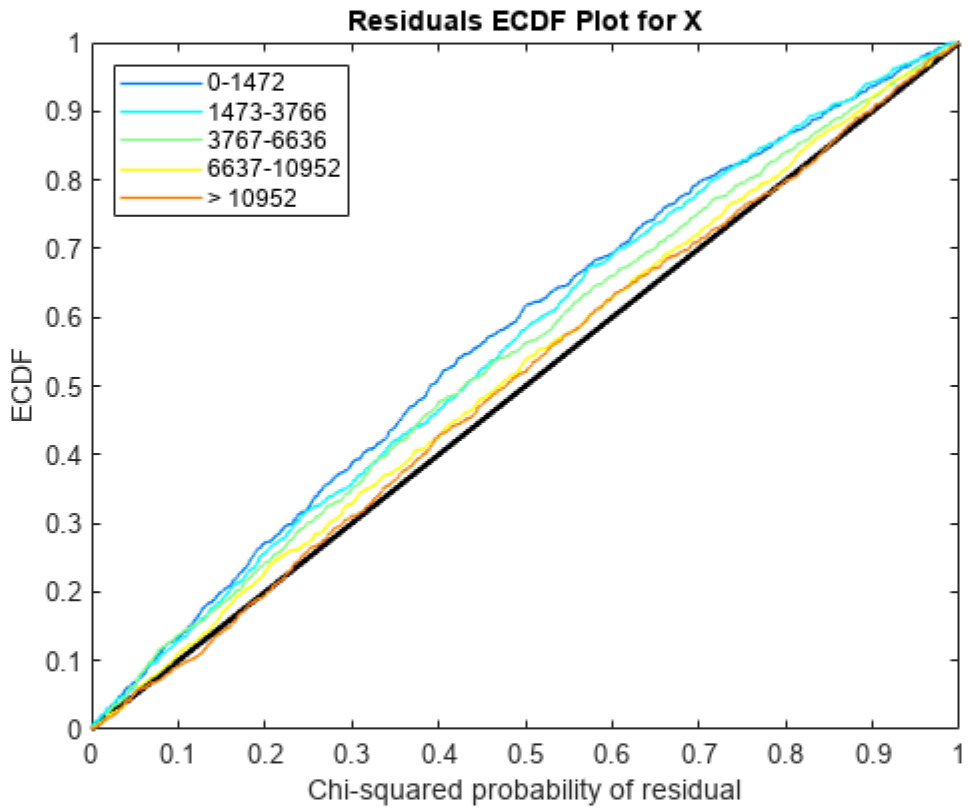


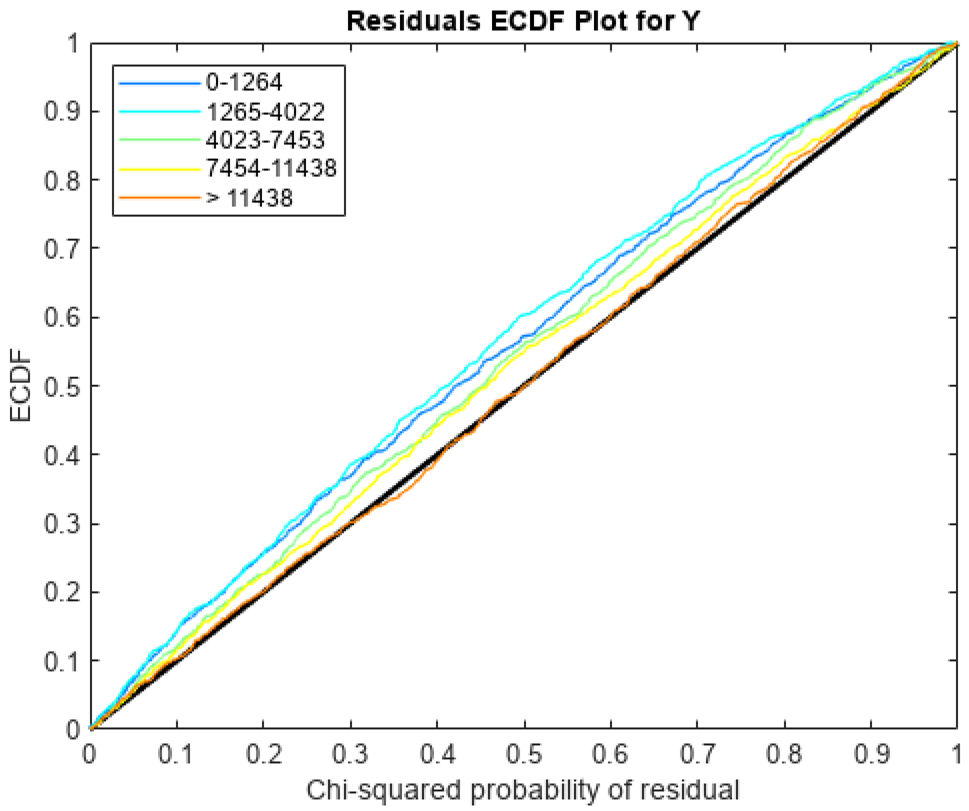


The `Identity` line represents the Poisson model, where the variance is identical to the mean as described in [3]. Observe that the data seems to be overdispersed (that is, most points are above the `Identity` line). The `Constant` line represents the negative binomial model, where the variance is the sum of the shot noise term (mean) and a constant multiplied by the squared mean as described in [2]. The `Local Regression` and `Constant` linkage options appear to fit better with the overdispersed data.

Use `plotChiSquaredFit` to assess the goodness-of-fit for variance regression. It plots the empirical CDF (ecdf) of the chi-squared probabilities. The probabilities are the ratio between the observed and the estimated variance stratified by short-read count levels into five equal-sized bins.

```
plotChiSquaredFit(tLocal)
```


Residuals ECDF Plot for X
0-1472
1473-3766
3767-6636
6637-10952
> 10952
ECDF
Chi-squared probability of residual
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1



Each figure shows five ecdf curves. Each curve represents one of the five short-read count levels. For instance, the blue line represents the ecdf curve for a low short-read counts between 0 and 1264. The red line represents high counts (more than 11438).

One way to interpret the curves is to check if the ecdf curves are above the diagonal line. If they are above the line, then the variance is overestimated. If they are below the line, then the variance is underestimated. In both figures, the variance seems to be correctly estimated for higher counts (that is, the red line follows the diagonal line), but slightly overestimated for lower count levels.

To assess the performance of the hypothesis test, construct a confusion matrix using the known labels and the predicted p-values.

```
confusionmat(H0,(tLocal.pValue > .001))
```

```
ans = 2×2

         493           7
           5        4495
```

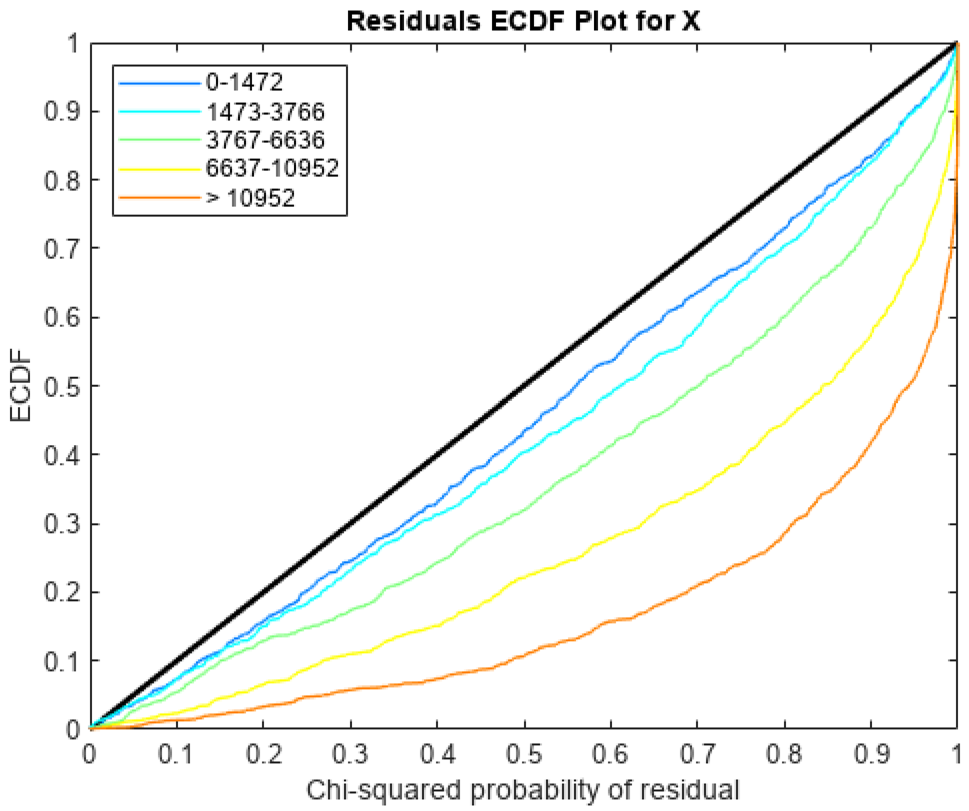
Out of 500 differentially expressed genes, 493 are correctly predicted (true positives) and 7 of them are incorrectly predicted as not-differentially expressed genes (false negatives). Out of 4500 genes that are not differentially expressed, 4495 are correctly predicted (true negatives) and 5 of them are incorrectly predicted as differentially expressed genes (false positives).

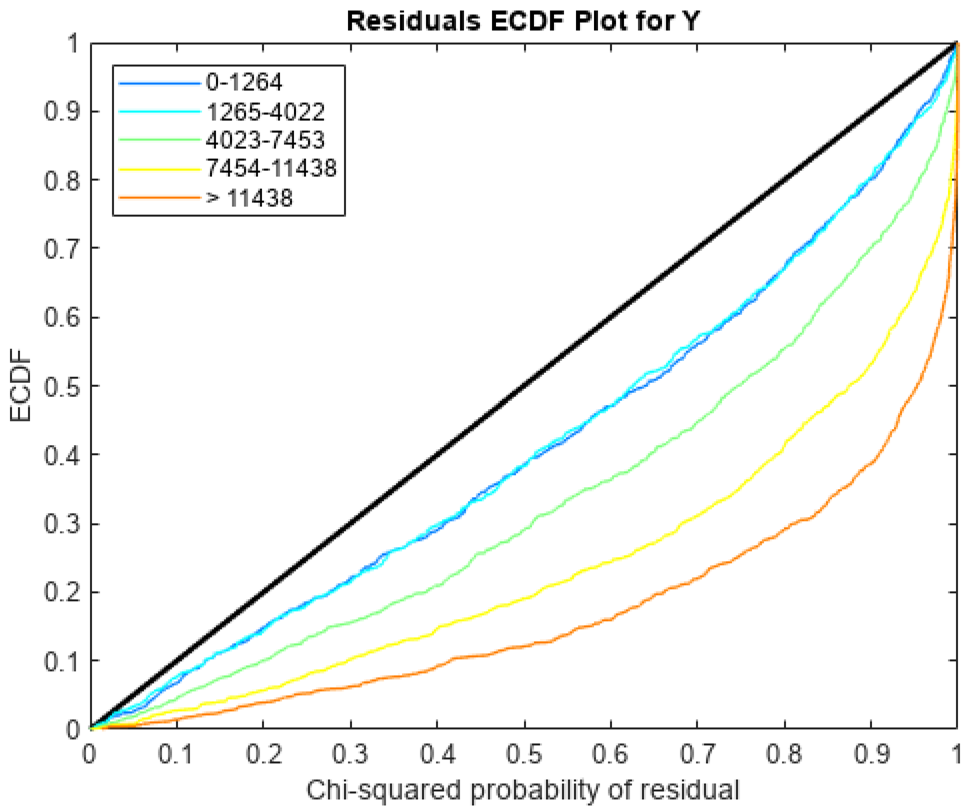
For a comparison, run the hypothesis test again assuming that counts are modeled by the Poisson distribution, where the variance is identical to the mean.

```
tPoisson = nbintest(K(:,1:5),K(:,6:10),'VarianceLink','Identity');
```

Plot the ecdf curves. Observe that all the curves are below the diagonal line, implying that the variance is underestimated. Therefore, the negative binomial model fits the data better.

```
plotChiSquaredFit(tPoisson)
```





Version History

Introduced in R2014b

References

[1] Anders, S., and Huber, W. (2010). Differential Expression Analysis for Sequence Count Data. Genome Biology, 11(10):R106.

[2] Robinson, M.D., and Smyth, G.K. (2008). Small-sample Estimation of Negative Binomial Dispersion, with Applications to SAGE data. Biostatistics, 9:321-332.

[3] Marioni, J.C., Mason, C.E., Mane, S.M., Stephens, M., and Gilad, Y. (2008). RNA-seq: an Assessment of Technical Reproducibility and Comparison with Gene Expression Arrays. Genome Research, 16:1509-1517.

See Also

`nbintest` | `mattest`

Topics

"Negative Binomial Distribution"

ndims (DataMatrix)

Return number of dimensions in DataMatrix object

Syntax

`N = ndims(DMObj)`

Input Arguments

DMObj	DataMatrix object, such as created by `DataMatrix` (object constructor).

Output Arguments

N	Positive integer representing the number of dimensions in *DMObj*. The number of dimensions in a DataMatrix object is always 2.

Description

`N = ndims(DMObj)` returns the number of dimensions in *DMObj*, a DataMatrix object. The number of dimensions in a DataMatrix object is always 2.

Version History

Introduced in R2008b

See Also

`DataMatrix`

Topics

DataMatrix object on page 1-734

ne (DataMatrix)

Test DataMatrix objects for inequality

Syntax

```
T = ne(DMObj1, DMObj2)
T = DMObj1 ~= DMObj2
T = ne(DMObj1, B)
T = DMObj1 ~= B
T = ne(B, DMObj1)
T = B ~= DMObj1
```

Input Arguments

DMObj1, *DMObj2*	DataMatrix objects, such as created by `DataMatrix` (object constructor).
B	MATLAB numeric or logical array.

Output Arguments

T	Logical matrix of the same size as *DMObj1* and *DMObj2* or *DMObj1* and *B*. It contains logical `1` (true) where elements in the first input are not equal to the corresponding element in the second input, and logical `0` (false) when they are equal.

Description

`T = ne(DMObj1, DMObj2)` or the equivalent `T = DMObj1 ~= DMObj2` compares each element in DataMatrix object *DMObj1* to the corresponding element in DataMatrix object *DMObj2*, and returns *T*, a logical matrix of the same size as *DMObj1* and *DMObj2*, containing logical `1` (true) where elements in *DMObj1* are not equal to the corresponding element in *DMObj2*, and logical `0` (false) when they are equal. *DMObj1* and *DMObj2* must have the same size (number of rows and columns), unless one is a scalar (1-by-1 DataMatrix object). *DMObj1* and *DMObj2* can have different `Name` properties.

`T = ne(DMObj1, B)` or the equivalent `T = DMObj1 ~= B` compares each element in DataMatrix object *DMObj1* to the corresponding element in *B*, a numeric or logical array, and returns *T*, a logical matrix of the same size as *DMObj1* and *B*, containing logical `1` (true) where elements in *DMObj1* are not equal to the corresponding element in *B*, and logical `0` (false) when they are equal. *DMObj1* and *B* must have the same size (number of rows and columns), unless one is a scalar.

`T = ne(B, DMObj1)` or the equivalent `T = B ~= DMObj1` compares each element in *B*, a numeric or logical array, to the corresponding element in DataMatrix object *DMObj1*, and returns *T*, a logical matrix of the same size as *B* and *DMObj1*, containing logical `1` (true) where elements in *B* are not equal to the corresponding element in *DMObj1*, and logical `0` (false) when they are equal. *B* and *DMObj1* must have the same size (number of rows and columns), unless one is a scalar.

MATLAB calls `T = ne(X, Y)` for the syntax `T = X ~= Y` when *X* or *Y* is a DataMatrix object.

Version History

Introduced in R2008b

See Also

`DataMatrix` | `eq`

Topics

DataMatrix object on page 1-734

nmercount

Count n-mers in nucleotide or amino acid sequence

Syntax

```
Nmer = nmercount(Seq, Length)
Nmer = nmercount(Seq, Length, C)
```

Input Arguments

Seq	One of the following: • Character vector or string specifying a nucleotide sequence or amino acid sequence. For valid letter codes, see the table Mapping Nucleotide Letter Codes to Integers or the table Mapping Amino Acid Letter Codes to Integers. • MATLAB structure containing a `Sequence` field that contains a nucleotide sequence or an amino acid sequence, such as returned by `fastaread`, `fastqread`, `emblread`, `getembl`, `genbankread`, `getgenbank`, `getgenpept`, `genpeptread`, `getpdb`, or `pdbread`.
Length	Integer specifying the length of n-mer to count.

Output Arguments

Nmer	Cell array containing the n-mer counts in *Seq*.

Description

Nmer = `nmercount(`*Seq*`,` *Length*`)` counts the n-mers or patterns of a specific length in *Seq*, a nucleotide sequence or amino acid sequence, and returns the n-mer counts in a cell array.

Nmer = `nmercount(`*Seq*`,` *Length*`,` *C*`)` returns only the n-mers with cardinality of at least *C*.

Examples

1 Use the `getgenpept` function to retrieve the amino acid sequence for the human insulin receptor.

```
S = getgenpept('AAA59174','SequenceOnly',true);
```

2 Count the number of four-mers in the amino acid sequence and display the first 20 rows in the cell array.

```
nmers = nmercount(S,4);
nmers(1:20,:)

ans =
    'APES'    [2]
```

```
        'DFRD'    [2]
        'ESLK'    [2]
        'FRDL'    [2]
        'GNYS'    [2]
        'LKEL'    [2]
        'SHCQ'    [2]
        'SLKD'    [2]
        'SVRI'    [2]
        'TDYL'    [2]
        'TSLA'    [2]
        'TVIN'    [2]
        'VING'    [2]
        'VPLD'    [2]
        'YALV'    [2]
        'AAAA'    [1]
        'AAAP'    [1]
        'AAEI'    [1]
        'AAEL'    [1]
        'AAFP'    [1]
```

Version History

Introduced before R2006a

See Also

`aacount` | `basecount` | `codoncount` | `dimercount`

nt2aa

Convert nucleotide sequence to amino acid sequence

Syntax

```
SeqAA = nt2aa(SeqNT)

SeqAA = nt2aa(..., 'Frame', FrameValue, ...)
SeqAA = nt2aa(..., 'GeneticCode', GeneticCodeValue, ...)
SeqAA = nt2aa(..., 'AlternativeStartCodons',
AlternativeStartCodonsValue, ...)
SeqAA = nt2aa(..., 'ACGTOnly', ACGTOnlyValue, ...)
```

Input Arguments

SeqNT	One of the following: • Character vector or string containing single-letter codes specifying a nucleotide sequence. For valid letter codes, see the table Mapping Nucleotide Letter Codes to Integers. • Row vector of integers specifying a nucleotide sequence. For valid integers, see the table Mapping Nucleotide Integers to Letter Codes. • MATLAB structure containing a `Sequence` field that contains a nucleotide sequence, such as returned by `fastaread`, `fastqread`, `emblread`, `getembl`, `genbankread`, or `getgenbank` **Note** Hyphens are valid only if the codon to which it belongs represents a gap, that is, the codon contains all hyphens. Example: `ACT---TGA` **Tip** Do not use a sequence with hyphens if you specify `'all'` for *`FrameValue`*.
FrameValue	Integer, character vector, or string specifying a reading frame in the nucleotide sequence. Choices are `1`, `2`, `3`, or `'all'`. Default is `1`. If *`FrameValue`* is `'all'`, then *`SeqAA`* is a 3-by-1 cell array.

GeneticCodeValue	Integer, character vector, or string specifying a genetic code number or code name from the table Genetic Code. Default is `1` or `'Standard'`. **Tip** If you use a code name, you can truncate the name to the first two letters of the name.
AlternativeStartCodonsValue	Controls the translation of alternative codons. Choices are `true` (default) or `false`.
ACGTOnlyValue	Controls the behavior of ambiguous nucleotide characters (`R`, `Y`, `K`, `M`, `S`, `W`, `B`, `D`, `H`, `V`, and `N`) and unknown characters. *ACGTOnlyValue* can be `true` (default) or `false`. • If `true`, then the function errors if any of these characters are present. • If `false`, then the function tries to resolve ambiguities. If it cannot, it returns `X` for the affected codon.

Output Arguments

SeqAA	Amino acid sequence specified by a character vector of single-letter codes.

Description

`SeqAA = nt2aa(SeqNT)` converts a nucleotide sequence, specified by *SeqNT*, to an amino acid sequence, returned in *SeqAA*, using the standard genetic code.

`SeqAA = nt2aa(SeqNT, ...'PropertyName', PropertyValue, ...)` calls `nt2aa` with optional properties that use property name/property value pairs. You can specify one or more properties in any order. Each *PropertyName* must be enclosed in single quotation marks and is case insensitive. These property name/property value pairs are as follows:

`SeqAA = nt2aa(..., 'Frame', FrameValue, ...)` converts a nucleotide sequence for a specific reading frame to an amino acid sequence. Choices are `1`, `2`, `3`, or `'all'`. Default is `1`. If *FrameValue* is `'all'`, then output *SeqAA* is a 3-by-1 cell array.

`SeqAA = nt2aa(..., 'GeneticCode', GeneticCodeValue, ...)` specifies a genetic code to use when converting a nucleotide sequence to an amino acid sequence. *GeneticCodeValue* can be an integer, character vector, or string specifying a code number or code name from the table Genetic Code. Default is `1` or `'Standard'`. The amino acid to nucleotide codon mapping for the Standard genetic code is shown in the table Standard Genetic Code.

Tip If you use a code name, you can truncate the name to the first two letters of the name.

`SeqAA = nt2aa(..., 'AlternativeStartCodons', AlternativeStartCodonsValue, ...)` controls the translation of alternative start codons. By default, *`AlternativeStartCodonsValue`* is set to `true`, and if the first codon of a sequence is a known alternative start codon, the codon is translated to methionine.

If this option is set to `false`, then an alternative start codon at the start of a sequence is translated to its corresponding amino acid in the genetic code that you specify, which might not necessarily be methionine. For example, in the human mitochondrial genetic code, AUA and AUU are known to be alternative start codons. For more information on alternative start codons, visit https://www.ncbi.nlm.nih.gov/Taxonomy/Utils/wprintgc.cgi?mode=t#SG1.

For more information about alternative start codons, see:

`www.ncbi.nlm.nih.gov/Taxonomy/Utils/wprintgc.cgi?mode=t#SG1`

Genetic Code

Code Number	Code Name
1	`Standard`
2	`Vertebrate Mitochondrial`
3	`Yeast Mitochondrial`
4	`Mold`, `Protozoan`, `Coelenterate Mitochondrial`, and `Mycoplasma/Spiroplasma`
5	`Invertebrate Mitochondrial`
6	`Ciliate`, `Dasycladacean`, and `Hexamita Nuclear`
9	`Echinoderm Mitochondrial`
10	`Euplotid Nuclear`
11	`Bacterial` and `Plant Plastid`
12	`Alternative Yeast Nuclear`
13	`Ascidian Mitochondrial`
14	`Flatworm Mitochondrial`
15	`Blepharisma Nuclear`
16	`Chlorophycean Mitochondrial`
21	`Trematode Mitochondrial`
22	`Scenedesmus Obliquus Mitochondrial`
23	`Thraustochytrium Mitochondrial`

Standard Genetic Code

Amino Acid Name	Amino Acid Code	Nucleotide Codon
Alanine	A	GCT GCC GCA GCG
Arginine	R	CGT CGC CGA CGG AGA AGG
Asparagine	N	AAT AAC
Aspartic acid (Aspartate)	D	GAT GAC
Cysteine	C	TGT TGC
Glutamine	Q	CAA CAG
Glutamic acid (Glutamate)	E	GAA GAG
Glycine	G	GGT GGC GGA GGG
Histidine	H	CAT CAC
Isoleucine	I	ATT ATC ATA
Leucine	L	TTA TTG† CTT CTC CTA CTG† † indicates alternative start codon for the Standard Genetic Code as defined here. If you are using `nt2aa`, alternative start codons are converted to methionine (M) by default when one of these codons are the first codon of a sequence. To change this default behavior, set the `AlternativeStartCodons` name-value argument of `nt2aa` to `false`.
Lysine	K	AAA AAG
Methionine	M	ATG
Phenylalanine	F	TTT TTC
Proline	P	CCT CCC CCA CCG
Serine	S	TCT TCC TCA TCG AGT AGC
Threonine	T	ACT ACC ACA ACG
Tryptophan	W	TGG
Tyrosine	Y	TAT TAC
Valine	V	GTT GTC GTA GTG
Asparagine or Aspartic acid (Aspartate)	B	Random codon from `D` and `N`
Glutamine or Glutamic acid (Glutamate)	Z	Random codon from `E` and `Q`
Unknown amino acid (any amino acid)	X	Random codon
Translation stop	*	TAA TAG TGA
Gap of indeterminate length	-	---

Amino Acid Name	Amino Acid Code	Nucleotide Codon
Unknown character (any character or symbol not in table)	?	???

SeqAA = nt2aa(..., 'ACGTOnly', *ACGTOnlyValue*, ...) controls the behavior of ambiguous nucleotide characters (R, Y, K, M, S, W, B, D, H, V, and N) and unknown characters. *ACGTOnlyValue* can be true (default) or false. If true, then the function errors if any of these characters are present. If false, then the function tries to resolve ambiguities. If it cannot, it returns X for the affected codon.

Examples

Example 1.28. Converting the ND1 Gene

1 Use the getgenbank function to retrieve genomic information for the human mitochondrion from the GenBank database and store it in a MATLAB structure .

```
mitochondria = getgenbank('NC_012920')

mitochondria = 

                LocusName: 'NC_012920'
      LocusSequenceLength: '16569'
     LocusNumberofStrands: ''
            LocusTopology: 'circular'
        LocusMoleculeType: 'DNA'
     LocusGenBankDivision: 'PRI'
    LocusModificationDate: '05-MAR-2010'
               Definition: 'Homo sapiens mitochondrion, complete genome.'
                Accession: 'NC_012920 AC_000021'
                  Version: 'NC_012920.1'
                       GI: '251831106'
                  Project: []
                   DBLink: 'Project:30353'
                 Keywords: []
                  Segment: []
                   Source: 'mitochondrion Homo sapiens (human)'
           SourceOrganism: [4x65 char]
                Reference: {1x7 cell}
                  Comment: [24x67 char]
                 Features: [933x74 char]
                      CDS: [1x13 struct]
                 Sequence: [1x16569 char]
                SearchURL: [1x70 char]
              RetrieveURL: [1x104 char]
```

2 Determine the name and location of the first gene in the human mitochondrion.

```
mitochondria.CDS(1).gene

ans =

ND1

mitochondria.CDS(1).location
```

```
ans =

3307..4262
```

3 Extract the sequence for the ND1 gene from the nucleotide sequence.

```
ND1gene = mitochondria.Sequence(3307:4262);
```

4 Convert the ND1 gene on the human mitochondria genome to an amino acid sequence using the Vertebrate Mitochondrial genetic code.

```
protein1 = nt2aa(ND1gene,'GeneticCode', 2);
```

5 Use the `getgenpept` function to retrieve the same amino acid sequence from the GenPept database.

```
protein2 = getgenpept('YP_003024026', 'SequenceOnly', true);
```

6 Use the `isequal` function to compare the two amino acid sequences.

```
isequal (protein1, protein2)

ans =

     1
```

Example 1.29. Converting the ND2 Gene

1 Use the `getgenbank` function to retrieve the nucleotide sequence for the human mitochondrion from the GenBank database.

```
mitochondria = getgenbank('NC_012920');
```

2 Determine the name and location of the second gene in the human mitochondrion.

```
mitochondria.CDS(2).gene

ans =

ND2

mitochondria.CDS(2).location

ans =

4470..5511
```

3 Extract the sequence for the ND2 gene from the nucleotide sequence.

```
ND2gene = mitochondria.Sequence(4470:5511);
```

4 Convert the ND2 gene on the human mitochondria genome to an amino acid sequence using the Vertebrate Mitochondrial genetic code.

```
protein1 = nt2aa(ND2gene,'GeneticCode', 2);
```

Note In the `ND2gene` nucleotide sequence, the first codon is `ATT`, which is translated to `M`, while the subsequent `ATT` codons are translated to `I`. If you set `'AlternativeStartCodons'` to `false`, then the first `ATT` codon is translated to `I`, the corresponding amino acid in the Vertebrate Mitochondrial genetic code.

5 Use the `getgenpept` function to retrieve the same amino acid sequence from the GenPept database.

```
protein2 = getgenpept('YP_003024027', 'SequenceOnly', true);
```

6 Use the `isequal` function to compare the two amino acid sequences.

```
isequal (protein1, protein2)

ans =

     1
```

Example 1.30. Converting a Sequence with Ambiguous Characters

If you have a sequence with ambiguous or unknown nucleotide characters, you can set the `'ACGTOnly'` property to `false` to have the `nt2aa` function try to resolve them:

```
nt2aa('agttgccgacgcgcncar','ACGTOnly', false)

ans =

SCRRAQ
```

Version History

Introduced before R2006a

See Also

`aa2nt` | `aminolookup` | `baselookup` | `codonbias` | `dnds` | `dndsml` | `geneticcode` | `isotopicdist` | `revgeneticcode` | `seqviewer`

nt2int

Convert nucleotide sequence from letter to integer representation

Syntax

```
SeqInt = nt2int(SeqChar)

SeqInt = nt2int(SeqChar, ...'Unknown', UnknownValue, ...)
SeqInt = nt2int(SeqChar, ...'ACGTOnly', ACGTOnlyValue, ...)
```

Input Arguments

SeqChar	One of the following: • Character vector or string specifying a nucleotide sequence. For valid letter codes, see the table Mapping Nucleotide Letter Codes to Integers. Integers are arbitrarily assigned to IUB/IUPAC letters. • MATLAB structure containing a `Sequence` field that contains a nucleotide sequence, such as returned by `fastaread`, `fastqread`, `emblread`, `getembl`, `genbankread`, or `getgenbank`.
UnknownValue	Integer to represent unknown nucleotides. Choices are integers ≥ `0` and ≤ `255`. Default is `0`.
ACGTOnlyValue	Controls the prohibition of ambiguous nucleotides. Choices are `true` or `false` (default). If *ACGTOnlyValue* is `true`, you can enter only the characters `A`, `C`, `G`, `T`, and `U`.

Output Arguments

SeqInt	Nucleotide sequence specified by a row vector of integers.

Description

SeqInt = `nt2int(`*SeqChar*`)` converts *SeqChar*, a character vector or string specifying a nucleotide sequence, to *SeqInt*, a row vector of integers specifying the same nucleotide sequence. For valid codes, see the table Mapping Nucleotide Letter Codes to Integers. Unknown characters (characters not in the table) are mapped to `0`. Gaps represented with hyphens are mapped to `16`.

SeqInt = `nt2int(`*SeqChar*`, ...'`*PropertyName*`',` *PropertyValue*`, ...)` calls `nt2int` with optional properties that use property name/property value pairs. You can specify one or more properties in any order. Each *PropertyName* must be enclosed in single quotation marks and is case insensitive. These property name/property value pairs are as follows:

SeqInt = `nt2int(`*SeqChar*`, ...'Unknown',` *UnknownValue*`, ...)` specifies an integer to represent unknown nucleotides. *UnknownValue* can be an integer ≥ `0` and ≤ `255`. Default is `0`.

`SeqInt = nt2int(SeqChar, ...'ACGTOnly', ACGTOnlyValue, ...)` controls the prohibition of ambiguous nucleotides (N, R, Y, K, M, S, W, B, D, H, and V). Choices are `true` or `false` (default). If *`ACGTOnlyValue`* is `true`, you can enter only the characters A, C, G, T, and U.

Mapping Nucleotide Letter Codes to Integers

Nucleotide	Code	Integer
Adenosine	A	1
Cytidine	C	2
Guanine	G	3
Thymidine	T	4
Uridine (if `'Alphabet'` set to `'RNA'`)	U	4
Purine (A or G)	R	5
Pyrimidine (T or C)	Y	6
Keto (G or T)	K	7
Amino (A or C)	M	8
Strong interaction (3 H bonds) (G or C)	S	9
Weak interaction (2 H bonds) (A or T)	W	10
Not A (C or G or T)	B	11
Not C (A or G or T)	D	12
Not G (A or C or T)	H	13
Not T or U (A or C or G)	V	14
Any nucleotide (A or C or G or T or U)	N	15
Gap of indeterminate length	-	16
Unknown (any character not in table)	*	0 (default)

Examples

Example 1.31. Converting a Simple Sequence

Convert a nucleotide sequence from letters to integers.

```
s = nt2int('ACTGCTAGC')
```

```
s =
     1     2     4     3     2     4     1     3     2
```

Example 1.32. Converting a Random Sequence

1 Create a random character vector to represent a nucleotide sequence.

```
SeqChar = randseq(20)
```

```
SeqChar =

TTATGACGTTATTCTACTTT
```

2 Convert the nucleotide sequence from letter to integer representation.

```
SeqInt = nt2int(SeqChar)

SeqInt =

  Columns 1 through 13
     4    4    1    4    3    1    2    3    4    4    1    4    4

  Columns 14 through 20
     2    4    1    2    4    4    4
```

Version History

Introduced before R2006a

See Also

aa2int | baselookup | int2aa | int2nt

ntdensity

Plot density of nucleotides along sequence

Syntax

```
ntdensity(SeqNT)
Density = ntdensity(SeqNT)

... = ntdensity(..., 'Window', WindowValue, ...)
[Density, HighCG] = ntdensity(..., 'CGThreshold', CGThresholdValue, ...)
```

Arguments

SeqNT	One of the following: • Character vector or string specifying a nucleotide sequence. For valid letter codes, see the table Mapping Nucleotide Letter Codes to Integers. • Row vector of integers specifying a nucleotide sequence. For valid integers, see the table Mapping Nucleotide Integers to Letter Codes. • MATLAB structure containing a `Sequence` field that contains a nucleotide sequence, such as returned by `emblread`, `fastaread`, `fastqread`, `genbankread`, `getembl`, or `getgenbank`. **Note** Although you can submit a sequence with nucleotides other than `A`, `C`, `G`, and `T`, `ntdensity` plots only `A`, `C`, `G`, and `T`.
WindowValue	Value that specifies the window length for the density calculation. Default is `length(`*SeqNT*`)/20`.
CGThresholdValue	Controls the return of indices for regions where the `CG` content of *SeqNT* is greater than *CGThresholdValue*. Default is `5`.

Description

`ntdensity(`*SeqNT*`)` plots the density of nucleotides `A`, `C`, `G`, and `T` in sequence *SeqNT*.

Density `= ntdensity(`*SeqNT*`)` returns a MATLAB structure with the density of nucleotides `A`, `C`, `G`, and `T`.

`... = ntdensity(`*SeqNT*`, ...'`*PropertyName*`', `*PropertyValue*`, ...)` calls `ntdensity` with optional properties that use property name/property value pairs. You can specify one or more properties in any order. Each *PropertyName* must be enclosed in single quotation marks and is case insensitive. These property name/property value pairs are as follows:

`... = ntdensity(..., 'Window', `*WindowValue*`, ...)` uses a window of length *WindowValue* for the density calculation. Default *WindowValue* is `length(`*SeqNT*`)/20`.

`[Density, HighCG] = ntdensity(..., 'CGThreshold', CGThresholdValue, ...)` returns indices for regions where the CG content of *`SeqNT`* is greater than *`CGThresholdValue`*. Default *`CGThresholdValue`* is 5.

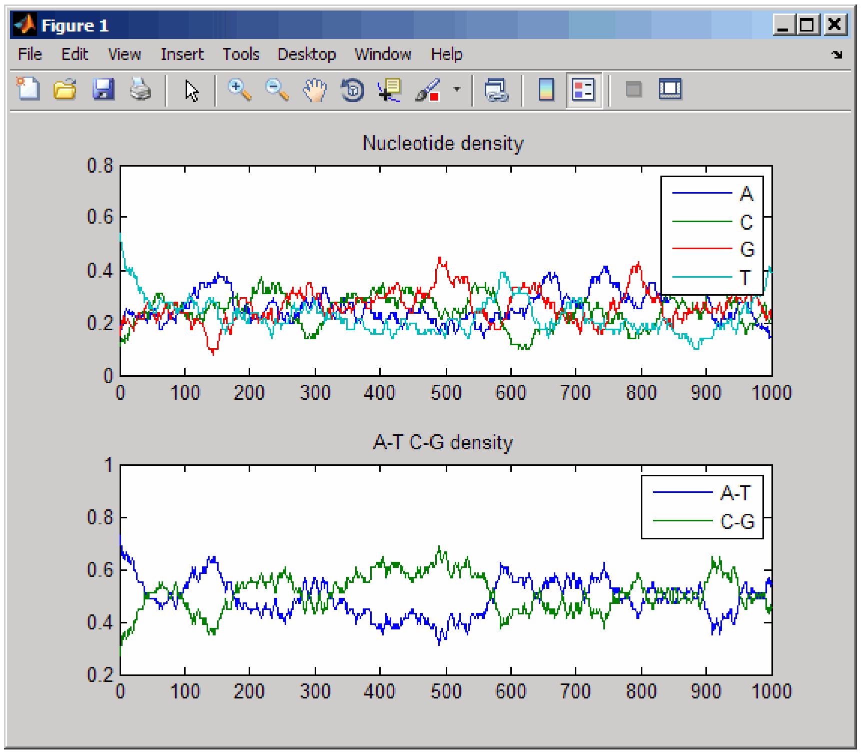
Examples

1 Create a random character vector to represent a nucleotide sequence.

```
s = randseq(1000, 'alphabet', 'dna');
```

2 Plot the density of nucleotides along the sequence.

```
ntdensity(s)
```



Version History

Introduced before R2006a

See Also

`basecount` | `codoncount` | `cpgisland` | `dimercount` | `filter`

nuc44

Return NUC44 scoring matrix for nucleotide sequences

Syntax

```
ScoringMatrix = nuc44
[ScoringMatrix, MatrixInfo] = nuc44
```

Description

`ScoringMatrix = nuc44` returns the scoring matrix. The `nuc44` scoring matrix uses ambiguous nucleotide codes and probabilities rounded to the nearest integer.

Scale = `0.277316`

Expected score = `-1.7495024`, Entropy = `0.5164710` bits

Lowest score = `-4`, Highest score = `5`

Order: `A C G T R Y K M S W B D H V N`

`[ScoringMatrix, MatrixInfo] = nuc44` returns a structure with information about the matrix with fields `Name` and `Order`.

Note The NUC44 scoring matrix is supplied by NCBI and is used by the BLAST suite of programs. For more information, see `ftp://ftp.ncbi.nih.gov/blast/matrices/`.

Version History

Introduced before R2006a

See Also

`blosum` | `dayhoff` | `gonnet` | `localalign` | `nwalign` | `pam` | `swalign`

num2goid

Convert numbers to Gene Ontology IDs

Syntax

GOIDs = num2goid(*X*)

Description

GOIDs = num2goid(*X*) converts the numbers in *X* to a cell array of character vectors with Gene Ontology IDs. IDs are seven-digit numbers preceded by the prefix GO:, which is the standard used by the Gene Ontology database.

Examples

Get the Gene Ontology IDs of the following numbers.

```
 t = [5575 5622 5623 5737 5840 30529 43226 43228 43229 43232 43234];
 ids = num2goid(t)
```

```
Columns 1 through 4

    'GO:0005575'    'GO:0005622'    'GO:0005623'    'GO:0005737'

  Columns 5 through 8

    'GO:0005840'    'GO:0030529'    'GO:0043226'    'GO:0043228'

  Columns 9 through 11

    'GO:0043229'    'GO:0043232'    'GO:0043234'
```

Version History

Introduced before R2006a

See Also

geneont | goannotread | geneont | getancestors | getdescendants | getmatrix | getrelatives

numel (DataMatrix)

Return number of elements in DataMatrix object

Syntax

```
N = numel(DMObj)
Ns = numel(DMObj, Index1, Index2)
```

Input Arguments

DMObj	DataMatrix object, such as created by `DataMatrix` (object constructor).
Index1	A row or range of rows in *DMObj* specified by a positive integer or a range using the format `x:y`, where `x` is the first row and `y` is the last row.
Index2	A column or range of columns in *DMObj* specified by a positive integer or a range using the format `x:y`, where `x` is the first column and `y` is the last column.

Output Arguments

N	Positive integer representing the number of elements in *DMObj*, a DataMatrix object.
Ns	Positive integer representing the number of subscripted elements in *DMObj*, a DataMatrix object.

Description

`N = numel(DMObj)` returns `1`. To find the number of elements in *DMObj*, a DataMatrix object, use either of the following syntaxes:

```
prod(size(DMObj))
numel(DMObj,':',':')
```

`Ns = numel(DMObj, Index1, Index2)` returns the number of subscripted elements in *DMObj*, a DataMatrix object. *Index1* specifies a row or range of rows in *DMObj*. *Index2* specifies a column or range of columns in *DMObj*.

Version History

Introduced in R2008b

See Also

`DataMatrix`

Topics

DataMatrix object on page 1-734

nwalign

Globally align two sequences using Needleman-Wunsch algorithm

Syntax

```
Score = nwalign(Seq1,Seq2)
Score = nwalign(Seq1,Seq2,Name=Value)
[Score,Alignment] = nwalign(Seq1,Seq2, ___ )
[Score,Alignment,Start] = nwalign(Seq1,Seq2, ___ )
```

Description

`Score = nwalign(Seq1,Seq2)` returns the optimal global alignment score in bits after aligning two sequences `Seq1` and `Seq2`. The scale factor used to calculate the score is provided by `ScoringMatrix`.

`Score = nwalign(Seq1,Seq2,Name=Value)` uses additional options specified by one or more name-value arguments.

`[Score,Alignment] = nwalign(Seq1,Seq2, ___ )` also returns a character array `Alignment` showing the alignment of `Seq1` and `Seq2`.

`[Score,Alignment,Start] = nwalign(Seq1,Seq2, ___ )` also returns a vector of indices `Start` as `[1;1]` indicating the starting point in each sequence for the alignment.

Examples

Perform global alignment of two sequences

Globally align two amino acid sequences using the `BLOSUM50` (default) scoring matrix and the default values for the `GapOpen` and `ExtendGap` properties. Return the optimal global alignment score in bits and the alignment character array.

```
seq1 = "VSPAGMASGYD";
seq2 = "IPGKASYD";
[Score, Alignment] = nwalign(seq1,seq2)
```

```
Score = 7.3333

Alignment = 3x11 char array
    'VSPAGMASGYD'
    ': | | || ||'
    'I-P-GKAS-YD'
```

Specify the `PAM250` scoring matrix and a gap open penalty of `5`.

```
[Score,Alignment] = nwalign(seq1,seq2,ScoringMatrix="PAM250",GapOpen=5)
```

```
Score = 6
```

```
Alignment = 3x11 char array
    'VSPAGMASGYD'
    ': | |:|| ||'
    'I-P-GKAS-YD'
```

Return the *Score* in nat units (nats) by specifying a scale factor of `log(2)`.

```
[Score,Alignment] = nwalign(seq1,seq2,Scale=log(2))
```

```
Score = 5.0831

Alignment = 3x11 char array
    'VSPAGMASGYD'
    ': | | || ||'
    'I-P-GKAS-YD'
```

Input Arguments

Seq1 — Amino or nucleotide sequence to align
character vector | string scalar | vector of integers | structure

Amino or nucleotide sequence to align, specified as a character vector or string scalar, vector of integers, or structure.

You can specify:

- Character vector or string scalar representing an amino acid or nucleotide sequence, such as the output from `int2aa` or `int2nt`.
- Vector of integers representing an amino acid or nucleotide sequence, such as the output from `aa2int` or `nt2int`,
- Structure containing a `Sequence` field.

Tip For help with letter and integer representations of amino acids and nucleotides, see Amino Acid Lookup or Nucleotide Lookup.

Data Types: `char` | `string` | `double` | `struct`

Seq2 — Amino or nucleotide sequence to align
character vector | string scalar | vector of integers | structure

Amino or nucleotide sequence to align, specified as a character vector or string scalar, vector of integers, or structure. For details, see `Seq1`.

Data Types: `char` | `string` | `double` | `struct`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: `[s,a] = nwalign("HEAGAWGHEE","PAWHEAE",GapOpen=5,ShowScore=true)` specifies to use the value of 5 as a penalty for gap opening and to show the scoring space and winning path.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `[s,a] = nwalign("HEAGAWGHEE","PAWHEAE",'GapOpen',5,'ShowScore',true)`

`Alphabet` — Type of sequence
`"AA"` (default) | `"NT"`

Type of sequence, specified as `"AA"` (amino acid) or `"NT"` (nucleotide).

Data Types: `char` | `string`

`ScoringMatrix` — Scoring matrix for global alignment
`"BLOSUM50"` (for amino acid sequences) (default) | character vector | string scalar | numeric matrix

Scoring matrix for the global alignment, specified as a character vector, string scalar, or numeric matrix.

You can specify a scoring matrix name. Valid choices are:

- `"BLOSUM50"` (default for amino acid sequences)
- `"NUC44"` (default for nucleotide sequences)
- `"BLOSUM62"`
- `"BLOSUM30"` increasing by `5` up to `"BLOSUM90"`
- `"BLOSUM100"`
- `"PAM10"` increasing by `10` up to `"PAM500"`
- `"DAYHOFF"`
- `"GONNET"`

Note The above scoring matrices, provided with the software, also include a scale factor that converts the units of the output score to bits. You can also specify the `Scale` name-value argument to specify an additional scale factor to convert the output score from bits to another unit.

You can also specify a numeric matrix, such as the one returned by the `blosum`, `pam`, `dayhoff`, `gonnet`, or `nuc44` function.

Note

- If you use a scoring matrix that you created or was created by one of these scoring matrix functions, the matrix does not include a scale factor. The output score will be returned in the same units as the scoring matrix. You can use the `Scale` name-value argument to specify a scale factor to convert the output score to another unit.
- If you need to compile `nwalign` into a standalone application or software component using MATLAB Compiler, use a numeric matrix instead of the scoring matrix name.

Data Types: `double` | `char` | `string`

`Scale` — Scale factor applied to output score
`1` (default) | numeric scalar | numeric vector

Scale factor applied to the output score, specified as a numeric scalar or vector. If you specify a vector, the function returns `Score` as a vector of the same length. By default, there is no scaling or change in the units of the output score.

Use this argument to control the units of the output scores. For example, if the output score is initially determined in bits, you can specify `Scale=log(2)` to return the output score in nats instead.

Note

- If the `ScoringMatrix` argument also specifies a scale factor, then the function uses it first to scale the output score, then applies the scale factor specified by the `Scale` argument to rescale the output score.
- Before comparing alignment scores from multiple alignments, ensure that the scores are in the same units.

Data Types: `double`

`GapOpen` — Penalty for opening gap
`8` (default) | positive scalar

Penalty for opening a gap, specified as a positive scalar.

Data Types: `double`

`ExtendGap` — Penalty for extending gap
positive scalar

Penalty for extending a gap using the affine gap penalty scheme, specified as a positive scalar.

If you specify this value, the function uses the affine gap penalty scheme, that is, it scores the first gap using the `GapOpen` value and scores subsequent gaps using the `ExtendGap` value. If you do not specify this value, the function scores all gaps equally, using the `GapOpen` penalty.

Data Types: `double`

`Glocal` — Flag to perform semiglobal alignment
`false` or `0` (default) | `true` or `1`

Flag to perform a semiglocal alignment, specified as a numeric or logical `1` (`true`) or 0 (`false`).

In a semiglobal alignment, gap penalties at the end of the sequences are null.

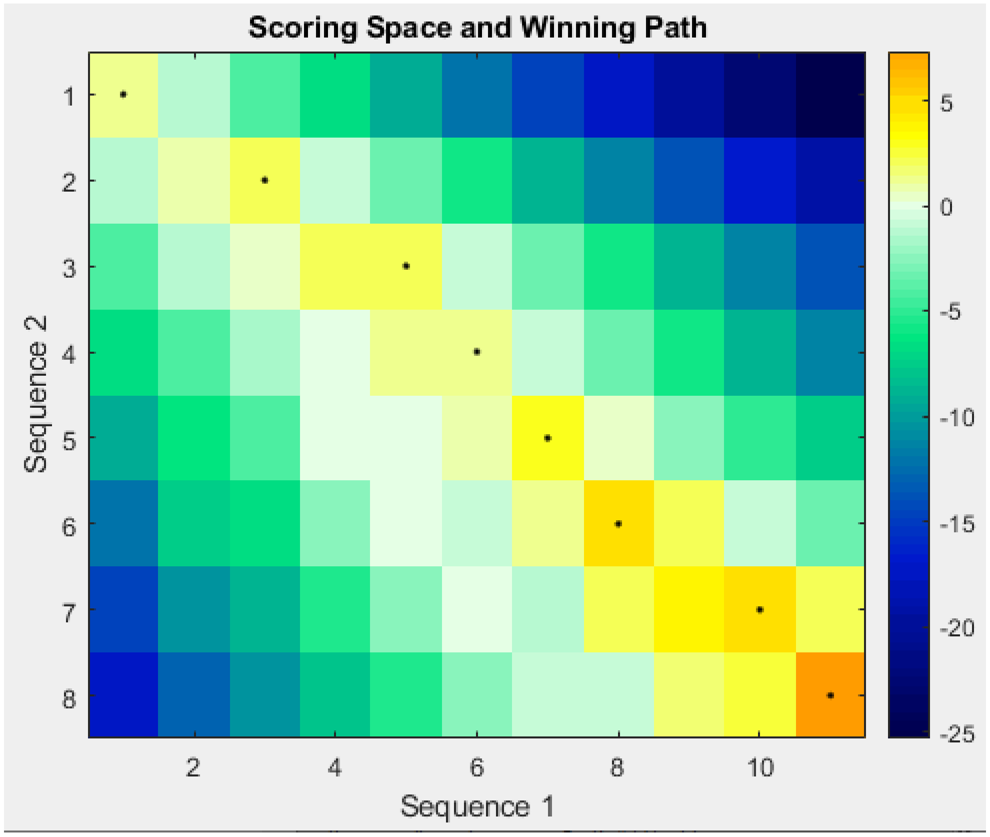
`Showscore` — Flag to display scoring space and winning path of alignment
`false` or `0` (default) | `true` or `1`

Flag to display the scoring space and winning path of the alignment, specified as a numeric or logical `1` (`true`) or 0 (`false`).

The scoring space is a heat map displaying the best scores for all the partial alignments of two sequences. The color of each (`n1,n2`) coordinate in the scoring space represents the best score for the pairing of subsequences `Seq1(1:n1)` and `Seq2(1:n2)`, where `n1` is a position in `Seq1` and `n2` is a position in `Seq2`. The best score for a pairing of specific subsequences is determined by scoring all possible alignments of the subsequences by summing matches and gap penalties.

The winning path is represented by black dots in the scoring space, and it illustrates the pairing of positions in the optimal global alignment. The color of the last point (lower right) of the winning path represents the optimal global alignment score for the two sequences and is the `Score` output.

Note The scoring space visually indicates if there are potential alternate winning paths, which is useful when aligning sequences with big gaps. Visual patterns in the scoring space can also indicate a possible sequence rearrangement.



Output Arguments

`Score` — Optimal global alignment score

numeric scalar | numeric vector

Optimal global alignment score, returned as a numeric scalar or vector. It is returned as a vector when you specify a numeric vector for the `Scale` name-value argument.

`Alignment` — Aligned sequences

character array

Aligned sequences, returned as a character array. The first and third rows are `Seq1` and `Seq2`, respectively. The second row shows symbols representing the optimal global alignment for two sequences. The symbol `|` indicates amino acids or nucleotides that match exactly. The symbol `:` indicates amino acids or nucleotides that are related as defined by the scoring matrix (nonmatches with a zero or positive scoring matrix value).

`Start` — Starting point in each sequence for alignment
`[1;1]`

Starting point in each sequence for the alignment, returned as a vector of indices. Because the function performs a global alignment, `Start` is always returned as `[1;1]`. The function returns this output to be consistent with the `swalign` function.

Version History

Introduced before R2006a

References

[1] Durbin, Richard, Sean R. Eddy, Anders Krogh, and Graeme Mitchison. *Biological Sequence Analysis: Probabilistic Models of Proteins and Nucleic Acids*. 1st ed. Cambridge University Press, 1998.

See Also

`aa2int` | `aminolookup` | `baselookup` | `blosum` | `dayhoff` | `gonnet` | `int2aa` | `int2nt` | `localalign` | `multialign` | `nt2aa` | `nt2int` | `nuc44` | `pam` | `profalign` | `seqdotplot` | `swalign`

oligoprop

Calculate sequence properties of DNA oligonucleotide

Syntax

```
SeqProperties = oligoprop(SeqNT)

SeqProperties = oligoprop(SeqNT, ...'Salt', SaltValue, ...)
SeqProperties = oligoprop(SeqNT, ...'Temp', TempValue, ...)
SeqProperties = oligoprop(SeqNT, ...'Primerconc', PrimerconcValue, ...)
SeqProperties = oligoprop(SeqNT, ...'HPBase', HPBaseValue, ...)
SeqProperties = oligoprop(SeqNT, ...'HPLoop', HPLoopValue, ...)
SeqProperties = oligoprop(SeqNT, ...'Dimerlength', DimerlengthValue, ...)
```

Input Arguments

SeqNT	DNA oligonucleotide sequence represented by any of the following: • Character vector or string containing the letters `A`, `C`, `G`, `T`, or `N` • Vector of integers containing the integers `1`, `2`, `3`, `4`, or `15` • Structure containing a `Sequence` field that contains a nucleotide sequence
SaltValue	Value that specifies a salt concentration in moles/liter for melting temperature calculations. Default is `0.05` moles/liter.
TempValue	Value that specifies the temperature in degrees Celsius for nearest-neighbor calculations of free energy. Default is `25` degrees Celsius.
PrimerconcValue	Value that specifies the concentration in moles/liter for melting temperature calculations. Default is `50e-6` moles/liter.
HPBaseValue	Value that specifies the minimum number of paired bases that form the neck of the hairpin. Default is `4` base pairs.
HPLoopValue	Value that specifies the minimum number of bases that form the loop of a hairpin. Default is `2` bases.
DimerlengthValue	Value that specifies the minimum number of aligned bases between the sequence and its reverse. Default is `4` bases.

Output Arguments

SeqProperties	Structure containing the sequence properties for a DNA oligonucleotide.

Description

`SeqProperties = oligoprop(SeqNT)` returns the sequence properties for a DNA oligonucleotide as a structure with the following fields:

Field	Description
GC	Percent GC content for the DNA oligonucleotide. Ambiguous N characters in *SeqNT* are considered to potentially be any nucleotide. If *SeqNT* contains ambiguous N characters, GC is the midpoint value, and its uncertainty is expressed by GCdelta.
GCdelta	The difference between GC (midpoint value) and either the maximum or minimum value GC could assume. The maximum and minimum values are calculated by assuming all N characters are G/C or not G/C, respectively. Therefore, GCdelta defines the possible range of GC content.
Hairpins	H-by-length(*SeqNT*) matrix of characters displaying all potential hairpin structures for the sequence *SeqNT*. Each row is a potential hairpin structure of the sequence, with the hairpin forming nucleotides designated by capital letters. H is the number of potential hairpin structures for the sequence. Ambiguous N characters in *SeqNT* are considered to potentially complement any nucleotide.
Dimers	D-by-length(*SeqNT*) matrix of characters displaying all potential dimers for the sequence *SeqNT*. Each row is a potential dimer of the sequence, with the self-dimerizing nucleotides designated by capital letters. D is the number of potential dimers for the sequence. Ambiguous N characters in *SeqNT* are considered to potentially complement any nucleotide.
MolWeight	Molecular weight of the DNA oligonucleotide. Ambiguous N characters in *SeqNT* are considered to potentially be any nucleotide. If *SeqNT* contains ambiguous N characters, MolWeight is the midpoint value, and its uncertainty is expressed by MolWeightdelta.
MolWeightdelta	The difference between MolWeight (midpoint value) and either the maximum or minimum value MolWeight could assume. The maximum and minimum values are calculated by assuming all N characters are G or C, respectively. Therefore, MolWeightdelta defines the possible range of molecular weight for *SeqNT*.
Tm	A vector with melting temperature values, in degrees Celsius, calculated by six different methods, listed in the following order: • Basic (Marmur et al., 1962 on page 1-1418) • Salt adjusted (Howley et al., 1979 on page 1-1418) • Nearest-neighbor (Breslauer et al., 1986 on page 1-1418) • Nearest-neighbor (SantaLucia Jr. et al., 1996 on page 1-1418) • Nearest-neighbor (SantaLucia Jr., 1998 on page 1-1418) • Nearest-neighbor (Sugimoto et al., 1996 on page 1-1418) Ambiguous N characters in *SeqNT* are considered to potentially be any nucleotide. If *SeqNT* contains ambiguous N characters, Tm is the midpoint value, and its uncertainty is expressed by Tmdelta.
Tmdelta	A vector containing the differences between Tm (midpoint value) and either the maximum or minimum value Tm could assume for each of the six methods. Therefore, Tmdelta defines the possible range of melting temperatures for *SeqNT*.

Field	Description
`Thermo`	4-by-3 matrix of thermodynamic calculations. The rows correspond to nearest-neighbor parameters from: • Breslauer et al., 1986 on page 1-1418 • SantaLucia Jr. et al., 1996 on page 1-1418 • SantaLucia Jr., 1998 on page 1-1418 • Sugimoto et al., 1996 on page 1-1418 The columns correspond to: • delta `H` — Enthalpy in kilocalories per mole, kcal/mol • delta `S` — Entropy in calories per mole-degrees Kelvin, cal/(K)(mol) • delta `G` — Free energy in kilocalories per mole, kcal/mol Ambiguous `N` characters in *`SeqNT`* are considered to potentially be any nucleotide. If *`SeqNT`* contains ambiguous `N` characters, `Thermo` is the midpoint value, and its uncertainty is expressed by `Thermodelta`.
`Thermodelta`	4-by-3 matrix containing the differences between `Thermo` (midpoint value) and either the maximum or minimum value `Thermo` could assume for each calculation and method. Therefore, `Thermodelta` defines the possible range of thermodynamic values for *`SeqNT`*.

`SeqProperties` `= oligoprop(`*`SeqNT`*`, ...'`*`PropertyName`*`',` *`PropertyValue`*`, ...)` calls `oligoprop` with optional properties that use property name/property value pairs. You can specify one or more properties in any order. Each *`PropertyName`* must be enclosed in single quotation marks and is case insensitive. These property name/property value pairs are as follows:

`SeqProperties` `= oligoprop(`*`SeqNT`*`, ...'Salt',` *`SaltValue`*`, ...)` specifies a salt concentration in moles/liter for melting temperature calculations. Default is `0.05` moles/liter.

`SeqProperties` `= oligoprop(`*`SeqNT`*`, ...'Temp',` *`TempValue`*`, ...)` specifies the temperature in degrees Celsius for nearest-neighbor calculations of free energy. Default is `25` degrees Celsius.

`SeqProperties` `= oligoprop(`*`SeqNT`*`, ...'Primerconc',` *`PrimerconcValue`*`, ...)` specifies the concentration in moles/liter for melting temperatures. Default is `50e-6` moles/liter.

`SeqProperties` `= oligoprop(`*`SeqNT`*`, ...'HPBase',` *`HPBaseValue`*`, ...)` specifies the minimum number of paired bases that form the neck of the hairpin. Default is `4` base pairs.

`SeqProperties` `= oligoprop(`*`SeqNT`*`, ...'HPLoop',` *`HPLoopValue`*`, ...)` specifies the minimum number of bases that form the loop of a hairpin. Default is `2` bases.

`SeqProperties` `= oligoprop(`*`SeqNT`*`, ...'Dimerlength',` *`DimerlengthValue`*`, ...)` specifies the minimum number of aligned bases between the sequence and its reverse. Default is `4` bases.

Examples

Example 1.33. Calculating Properties for a DNA Sequence

1 Create a random sequence.

```
seq = randseq(25)

seq =

TAGCTTCATCGTTGACTTCTACTAA
```

2 Calculate sequence properties of the sequence.

```
S1 = oligoprop(seq)

S1 =

               GC: 36
          GCdelta: 0
         Hairpins: [0x25 char]
           Dimers: 'tAGCTtcatcgttgacttctactaa'
        MolWeight: 7.5820e+003
   MolWeightdelta: 0
               Tm: [52.7640 60.8629 62.2493 55.2870 54.0293 61.0614]
          Tmdelta: [0 0 0 0 0 0]
           Thermo: [4x3 double]
      Thermodelta: [4x3 double]
```

3 List the thermodynamic calculations for the sequence.

```
S1.Thermo

ans =

 -178.5000 -477.5700  -36.1125
 -182.1000 -497.8000  -33.6809
 -190.2000 -522.9000  -34.2974
 -191.9000 -516.9000  -37.7863
```

Example 1.34. Calculating Properties for a DNA Sequence with Ambiguous Characters

1 Calculate sequence properties of the sequence ACGTAGAGGACGTN.

```
S2 = oligoprop('ACGTAGAGGACGTN')

S2 =

               GC: 53.5714
          GCdelta: 3.5714
         Hairpins: 'ACGTagaggACGTn'
           Dimers: [3x14 char]
        MolWeight: 4.3329e+003
   MolWeightdelta: 20.0150
               Tm: [38.8357 42.2958 57.7880 52.4180 49.9633 55.1330]
          Tmdelta: [1.4643 1.4643 10.3885 3.4633 0.2829 3.8074]
           Thermo: [4x3 double]
      Thermodelta: [4x3 double]
```

2 List the potential dimers for the sequence.

```
S2.Dimers

ans =

ACGTagaggacgtn
```

```
    ACGTagaggACGTn
    acgtagagGACGTN
```

Version History

Introduced before R2006a

References

[1] Breslauer, K.J., Frank, R., Blöcker, H., and Marky, L.A. (1986). Predicting DNA duplex stability from the base sequence. Proceedings of the National Academy of Science USA *83*, 3746–3750.

[2] Chen, S.H., Lin, C.Y., Cho, C.S., Lo, C.Z., and Hsiung, C.A. (2003). Primer Design Assistant (PDA): A web-based primer design tool. Nucleic Acids Research *31(13)*, 3751–3754.

[3] Howley, P.M., Israel, M.A., Law, M., and Martin, M.A. (1979). A rapid method for detecting and mapping homology between heterologous DNAs. Evaluation of polyomavirus genomes. The Journal of Biological Chemistry *254(11)*, 4876–4883.

[4] Marmur, J., and Doty, P. (1962). Determination of the base composition of deoxyribonucleic acid from its thermal denaturation temperature. Journal Molecular Biology *5*, 109–118.

[5] Panjkovich, A., and Melo, F. (2005). Comparison of different melting temperature calculation methods for short DNA sequences. Bioinformatics *21(6)*, 711–722.

[6] SantaLucia Jr., J., Allawi, H.T., and Seneviratne, P.A. (1996). Improved Nearest-Neighbor Parameters for Predicting DNA Duplex Stability. Biochemistry *35*, 3555–3562.

[7] SantaLucia Jr., J. (1998). A unified view of polymer, dumbbell, and oligonucleotide DNA nearest-neighbor thermodynamics. Proceedings of the National Academy of Science USA *95*, 1460–1465.

[8] Sugimoto, N., Nakano, S., Yoneyama, M., and Honda, K. (1996). Improved thermodynamic parameters and helix initiation factor to predict stability of DNA duplexes. Nucleic Acids Research *24(22)*, 4501–4505.

[9] http://www.basic.northwestern.edu/biotools/oligocalc.html for weight calculations.

See Also

`palindromes`

Topics

isoelectric on page 1-1139
molweight on page 1-1259
ntdensity on page 1-1403
randseq on page 1-1551

palindromes

Find palindromes in sequence

Syntax

```
[Position, Length] = palindromes(SeqNT)
[Position, Length, Pal] = palindromes(SeqNT)

... = palindromes(SeqNT, ..., 'Length', LengthValue, ...)
... = palindromes(SeqNT, ..., 'Complement', ComplementValue, ...)
```

Arguments

SeqNT	One of the following: • Character vector or string specifying a nucleotide sequence. For valid letter codes, see the table Mapping Nucleotide Letter Codes to Integers. • Row vector of integers specifying a nucleotide sequence. For valid integers, see the table Mapping Nucleotide Integers to Letter Codes. • MATLAB structure containing a `Sequence` field that contains a nucleotide sequence, such as returned by `emblread`, `fastaread`, `fastqread`, `genbankread`, `getembl`, or `getgenbank`.
LengthValue	Integer specifying a minimum length for palindromes. Default is `6`.
ComplementValue	Controls the return of complementary palindromes, that is, where the elements match their complementary pairs `A`-`T` (or `U`) and `C`-`G` instead of an exact nucleotide match. Choices are `true` or `false` (default).

Description

`[`*`Position`*`, `*`Length`*`] = palindromes(`*`SeqNT`*`)` finds all palindromes in sequence *`SeqNT`* with a length greater than or equal to `6`, and returns the starting indices, *`Position`*, and the lengths of the palindromes, *`Length`*.

`[`*`Position`*`, `*`Length`*`, `*`Pal`*`] = palindromes(`*`SeqNT`*`)` also returns a cell array, *`Pal`*, of the palindromes.

`... = palindromes(`*`SeqNT`*`, ...'`*`PropertyName`*`', `*`PropertyValue`*`, ...)` calls `palindromes` with optional properties that use property name/property value pairs. You can specify one or more properties in any order. Each *`PropertyName`* must be enclosed in single quotation marks and is case insensitive. These property name/property value pairs are as follows:

`... = palindromes(`*`SeqNT`*`, ..., 'Length', `*`LengthValue`*`, ...)` finds all palindromes longer than or equal to *`LengthValue`*. Default is `6`.

`... = palindromes(`*`SeqNT`*`, ..., 'Complement', `*`ComplementValue`*`, ...)` controls the return of complementary palindromes, that is, where the elements match their complementary pairs

`A-T` (or `A-U`) and `C-G` instead of an exact nucleotide match. Choices for *`ComplementValue`* are `true` or `false` (default).

Examples

Find the palindromes in a simple nucleotide sequence.

```
[p,l,s] = palindromes('GCTAGTAACGTATATATAAT')

p =
    11
    12
l =
     7
     7
s =
    'TATATAT'
    'ATATATA'
```

Find the complementary palindromes in a simple nucleotide sequence.

```
[pc,lc,sc] = palindromes('TAGCTTGTCACTGAGGCCA',...
                         'Complement',true)
pc =
     8
lc =
     7
sc =
    'TCACTGA'
```

Find the palindromes in a random nucleotide sequence.

```
a = randseq(100)

a =
TAGCTTCATCGTTGACTTCTACTAA
AAGCAAGCTCCTGAGTAGCTGGCCA
AGCGAGCTTGCTTGTGCCCGGCTGC
GGCGGTTGTATCCTGAATACGCCAT

[pos,len,pal]=palindromes(a)

pos =
    74
len =
     6
pal =
    'GCGGCG'
```

Version History

Introduced before R2006a

See Also

`seqcomplement` | `seqrcomplement` | `seqreverse` | `regexp` | `strfind`

pam

Return Point Accepted Mutation (PAM) scoring matrix

Syntax

```
ScoringMatrix = pam(N)
[ScoringMatrix, MatrixInfo] = pam(N)

... = pam(N, ...'Extended', ExtendedValue, ...)
... = pam(N, ...'Order', OrderValue, ...)
```

Arguments

N	Integer specifying the PAM scoring matrix to return. Choices are `10:10:500`. **Tip** Entering a larger value for *N* allows for sequence alignments with larger evolutionary distances.
ExtendedValue	Controls the return of the ambiguous characters (`B`, `Z`, and `X`), and the stop character (`*`), in addition to the `20` standard amino acid characters. Choices are `true` or `false` (default).
OrderValue	Character vector or string that controls the order of amino acids in the scoring matrix. Choices are a character vector or string with at least the `20` standard amino acids. The default order of the output is `A R N D C Q E G H I L K M F P S T W Y V B Z X *`. If *OrderValue* does not contain the characters `B`, `Z`, `X`, and `*`, then these characters are not returned.

Description

`*ScoringMatrix* = pam(*N*)` returns the PAM*N* scoring matrix for amino acid sequences.

`[*ScoringMatrix*, *MatrixInfo*] = pam(*N*)` returns a structure with information about the PAM matrix. The fields in the structure are `Name`, `Scale`, `Entropy`, `Expected`, and `Order`.

`... = pam(*N*, ...'*PropertyName*', *PropertyValue*, ...)` calls `pam` with optional properties that use property name/property value pairs. You can specify one or more properties in any order. Each *PropertyName* must be enclosed in single quotation marks and is case insensitive. These property name/property value pairs are as follows:

`... = pam(*N*, ...'Extended', *ExtendedValue*, ...)` controls the return of the ambiguous characters (`B`, `Z`, and `X`), and the stop character (`*`), in addition to the `20` standard amino acid characters. Choices are `true` or `false` (default).

`... = pam(*N*, ...'Order', *OrderValue*, ...)` controls the order of amino acids in the returned scoring matrix. Choices are a character vector or string with at least the `20` standard amino acids. The default ordering of the output is `A R N D C Q E G H I L K M F P S T W Y V B Z X *`. If *`OrderValue`* does not contain the extended characters `B`, `Z`, `X`, and `*`, then these characters are not returned.

PAM50 substitution matrix in `1/2` bit units, Expected score = `-3.70`, Entropy = `2.00` bits, Lowest score = `-13`, Highest score = `13`.

PAM250 substitution matrix in `1/3` bit units, Expected score = `-0.844`, Entropy = `0.354` bits, Lowest score = `-8`, Highest score = `17`.

Examples

Return the PAM50 matrix.

```
PAM50 = pam(50)
```

Return the PAM250 matrix and specify the order of amino acids in the matrix.

```
PAM250 = pam(250,'Order','CSTPAGNDEQHRKMILVFYW')
```

Version History

Introduced before R2006a

See Also

`blosum` | `dayhoff` | `gonnet` | `localalign` | `nuc44` | `nwalign` | `swalign`

pdbdistplot

Visualize intermolecular distances in Protein Data Bank (PDB) file

Syntax

```
pdbdistplot(PDBid)
pdbdistplot(PDBid, Distance)
pdbdistplot(___,'Chain',ChainID)
pdbdistplot(___,'Model',ModelNum)
pdbdistplot(___,'Hetero',TF)
```

Arguments

PDBid	Any of the following: • Character vector or string specifying a unique identifier for a protein structure record. • Name of a variable for a MATLAB structure containing PDB information for a molecular structure, such as returned by `getpdb` or `pdbread`. • Name of file containing PDB information for a molecular structure, such as created by `getpdb` with the `'ToFile'` property. **Note** Each structure in the PDB database is represented by a four-character alphanumeric identifier. For example, `4hhb` is the identification code for hemoglobin.
Distance	Threshold distance in angstroms shown on a spy plot. Default is `7`.
ChainID	Any of the following: • Character vector or string specifying the chain ID to consider. • Cell arrays of character vectors or string vector specifying the list of chain IDs to consider. *ChainID* is case-sensitive. By default, all chains included in the model are considered.
ModelNum	Positive integer specifying which model to consider. Default is 1.

Description

`pdbdistplot` displays the distances between atoms and between residues in a PDB structure.

`pdbdistplot(`*`PDBid`*`)` retrieves the structure specified by *PDBid* from the PDB database and creates a heat map showing inter–residue distances and a spy plot showing the residues where the minimum distances apart are less than `7` angstroms. If multiple chains are present in *PDBid*, separate plots are created.

`pdbdistplot(`*`PDBid`*`, `*`Distance`*`)` specifies the threshold distance shown on a spy plot. Default is `7`.

`pdbdistplot(___,'Chain',ChainID)` specifies the chains to consider. By default, all chains included in the model are considered.

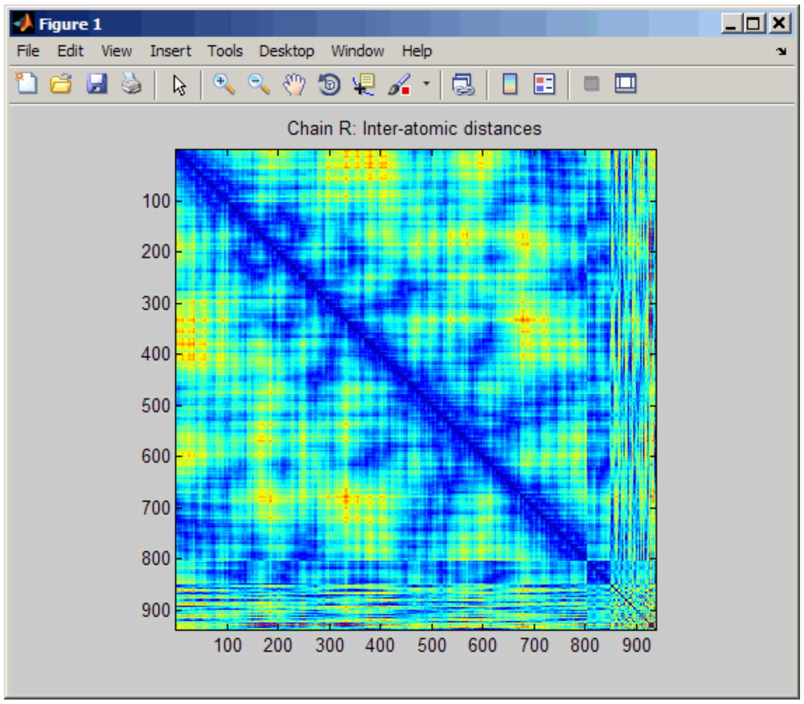
`pdbdistplot(___,'Model',ModelNum)` specifies which PDB structural model to consider. Default is 1.

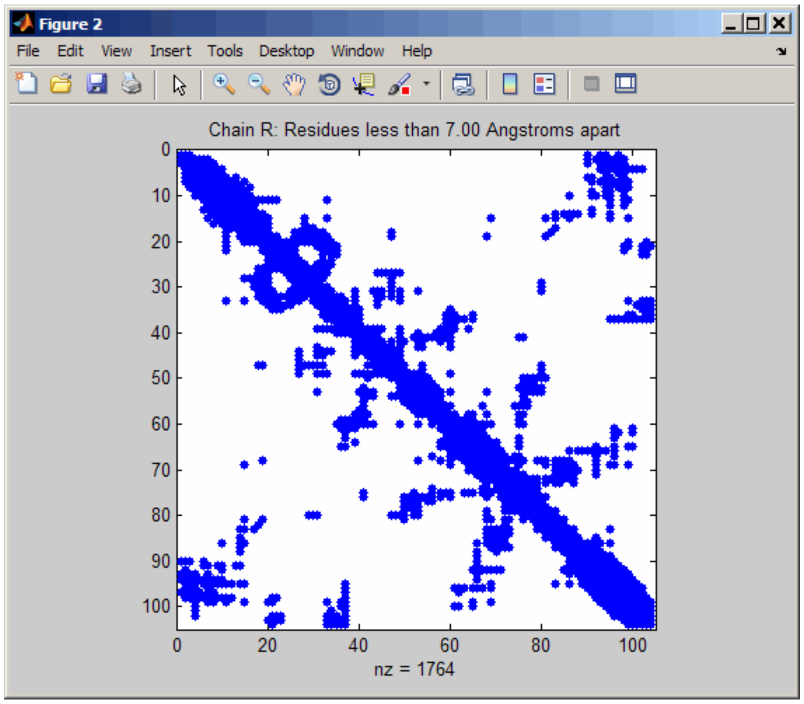
`pdbdistplot(___,'Hetero',TF)` specifies whether to include hetero atoms in the plot of residue interactions. *TF* is logical, `true` or `false`. Default is `false`.

Examples

Display a heat map of the inter–residue distances and a spy plot at 7 angstroms of the protein cytochrome C from albacore tuna.

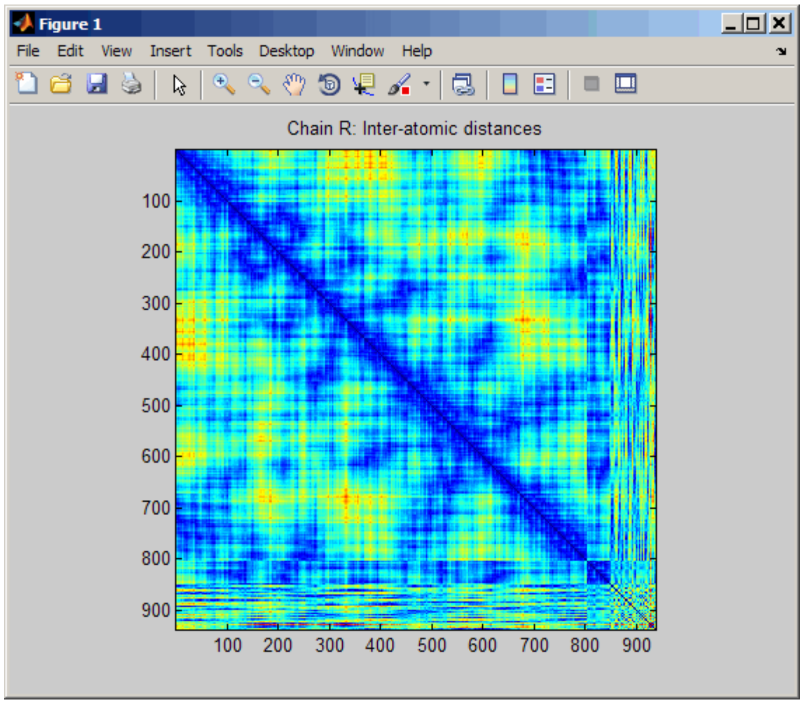
```
pdbdistplot('5CYT');
```

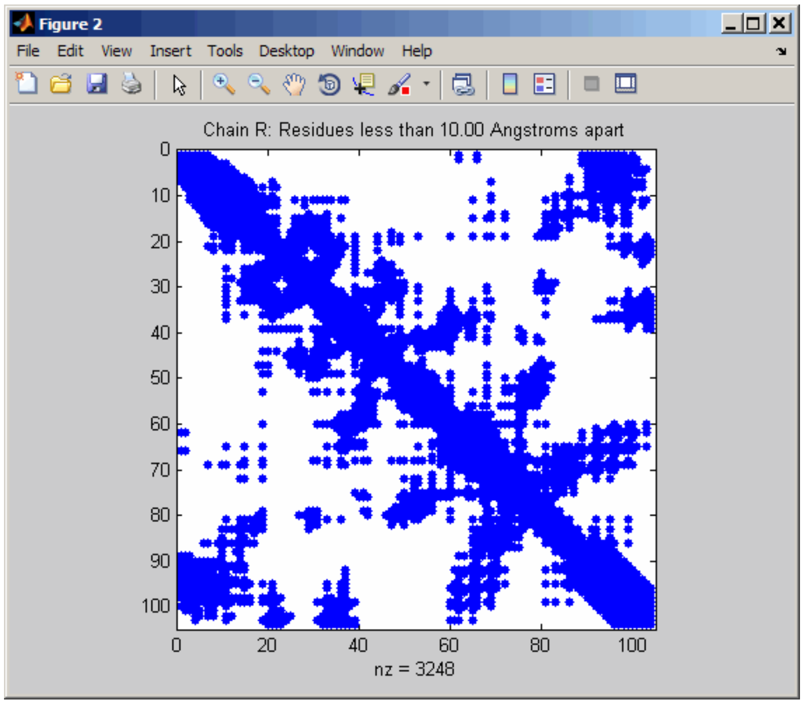




Display a spy plot at `10` angstroms of the same structure.

```
pdbdistplot('5CYT',10);
```





Version History

Introduced before R2006a

See Also

`getpdb` | `pdbread` | `proteinplot` | `ramachandran`

pdbread

Read data from Protein Data Bank (PDB) file

Syntax

```
PDBStruct = pdbread(File)

PDBStruct = pdbread(File, 'ModelNum', ModelNumValue)
PDBStruct = pdbread(File,'TimeOut', TimeOutValue)
```

Input Arguments

`File`	Either of the following: • Character vector or string specifying a file name, a path and file name, or a URL pointing to a file. The referenced file is a Protein Data Bank (PDB)-formatted file (ASCII text file). If you specify only a file name, that file must be on the MATLAB search path or in the MATLAB Current Folder. • Character array or column vector of strings that contains the text of a PDB-formatted file. **Tip** You can use the `getpdb` function with the `'ToFile'` property to retrieve protein structure data from the PDB database and create a PDB-formatted file.
`ModelNumValue`	Positive integer specifying a model in a PDB-formatted file.
`TimeOutValue`	Connection timeout in seconds, specified as a positive scalar. The default value is 5. For details, see here.

Output Arguments

`PDBStruct`	MATLAB structure containing a field for each PDB record.

Description

The Protein Data Bank (PDB) database is an archive of experimentally determined 3-D biological macromolecular structure data. For more information about the PDB format, see:

`https://www.wwpdb.org/documentation/file-format`

`PDBStruct` `= pdbread(`*`File`*`)` reads the data from PDB-formatted text file *`File`* and stores the data in the MATLAB structure, *`PDBStruct`*, which contains a field for each PDB record. The following table summarizes the possible PDB records and the corresponding fields in the MATLAB structure *`PDBStruct`*:

PDB Database Record	Field in the MATLAB Structure
`HEADER`	`Header`
`OBSLTE`	`Obsolete`
`TITLE`	`Title`
`CAVEAT`	`Caveat`
`COMPND`	`Compound`
`SOURCE`	`Source`
`KEYWDS`	`Keywords`
`EXPDTA`	`ExperimentData`
`AUTHOR`	`Authors`
`REVDAT`	`RevisionDate`
`SPRSDE`	`Superseded`
`JRNL`	`Journal`
`REMARK 1`	`Remark1`
`REMARK` *N* **Note** *N* equals 2 through 999.	`Remark`*n* **Note** *n* equals 2 through 999.
`DBREF`	`DBReferences`
`SEQADV`	`SequenceConflicts`
`SEQRES`	`Sequence`
`FTNOTE`	`Footnote`
`MODRES`	`ModifiedResidues`
`HET`	`Heterogen`
`HETNAM`	`HeterogenName`
`HETSYN`	`HeterogenSynonym`
`FORMUL`	`Formula`
`HELIX`	`Helix`
`SHEET`	`Sheet`
`TURN`	`Turn`
`SSBOND`	`SSBond`
`LINK`	`Link`
`HYDBND`	`HydrogenBond`
`SLTBRG`	`SaltBridge`
`CISPEP`	`CISPeptides`
`SITE`	`Site`
`CRYST1`	`Cryst1`
`ORIGXn`	`OriginX`

PDB Database Record	Field in the MATLAB Structure
`SCALEn`	`Scale`
`MTRIXn`	`Matrix`
`TVECT`	`TranslationVector`
`MODEL`	`Model`
`ATOM`	`Atom`
`SIGATM`	`AtomSD`
`ANISOU`	`AnisotropicTemp`
`SIGUIJ`	`AnisotropicTempSD`
`TER`	`Terminal`
`HETATM`	`HeterogenAtom`
`CONECT`	`Connectivity`

`PDBStruct = pdbread(File, 'ModelNum', ModelNumValue)` reads only the model specified by *`ModelNumValue`* from the PDB-formatted text file *`File`* and stores the data in the MATLAB structure *`PDBStruct`*. If *`ModelNumValue`* does not correspond to an existing mode number in *`File`*, then `pdbread` reads the coordinate information of all the models.

`PDBStruct = pdbread(File,'TimeOut', TimeOutValue)` sets the connection timeout (in seconds) to read data from the PDB database.

The Sequence Field

The `Sequence` field is also a structure containing sequence information in the following subfields:

- `NumOfResidues`
- `ChainID`
- `ResidueNames` — Contains the three-letter codes for the sequence residues.
- `Sequence` — Contains the single-letter codes for the sequence residues.

Note If the sequence has modified residues, then the `ResidueNames` subfield might not correspond to the standard three-letter amino acid codes. In this case, the `Sequence` subfield will contain the modified residue code in the position corresponding to the modified residue. The modified residue code is provided in the `ModifiedResidues` field.

The Model Field

The `Model` field is also a structure or an array of structures containing coordinate information. If the MATLAB structure contains one model, the `Model` field is a structure containing coordinate information for that model. If the MATLAB structure contains multiple models, the `Model` field is an array of structures containing coordinate information for each model. The `Model` field contains the following subfields:

- `Atom`
- `AtomSD`

- `AnisotropicTemp`
- `AnisotropicTempSD`
- `Terminal`
- `HeterogenAtom`

The Atom Field

The `Atom` field is also an array of structures containing the following subfields:

- `AtomSerNo`
- `AtomName`
- `altLoc`
- `resName`
- `chainID`
- `resSeq`
- `iCode`
- `X`
- `Y`
- `Z`
- `occupancy`
- `tempFactor`
- `segID`
- `element`
- `charge`
- `AtomNameStruct` — Contains three subfields: `chemSymbol`, `remoteInd`, and `branch`.

Examples

1. Use the `getpdb` function to retrieve structure information from the Protein Data Bank (PDB) for the nicotinic receptor protein with identifier `1abt`, and then save the data to the PDB-formatted file `nicotinic_receptor.pdb` in the MATLAB Current Folder.

   ```
   getpdb('1abt', 'ToFile', 'nicotinic_receptor.pdb');
   ```
2. Read the data from the `nicotinic_receptor.pdb` file into a MATLAB structure `pdbstruct`.

   ```
   pdbstruct = pdbread('nicotinic_receptor.pdb');
   ```
3. Read only the second model from the `nicotinic_receptor.pdb` file into a MATLAB structure `pdbstruct_Model2`.

   ```
   pdbstruct_Model2 = pdbread('nicotinic_receptor.pdb', 'ModelNum', 2);
   ```
4. View the atomic coordinate information in the model fields of both MATLAB structures `pdbstruct` and `pdbstruct_Model2`.

   ```
   pdbstruct.Model

   ans =

   1x4 struct array with fields:
   ```

```
        MDLSerNo
        Atom
        Terminal

    pdbstruct_Model2.Model

    ans = 

        MDLSerNo: 2
            Atom: [1x1205 struct]
        Terminal: [1x2 struct]
```

5 Read the data from a URL into a MATLAB structure, `gfl_pdbstruct`.

```
gfl_pdbstruct = pdbread('http://www.rcsb.org/pdb/files/1gfl.pdb');
```

Version History

Introduced before R2006a

See Also

`genpeptread` | `getpdb` | `pdbdistplot` | `pdbsuperpose` | `pdbtransform` | `pdbwrite`

pdbsuperpose

Superpose 3-D structures of two proteins

Syntax

```
pdbsuperpose(PDB1, PDB2)
Dist = pdbsuperpose(PDB1, PDB2)
[Dist, RMSD] = pdbsuperpose(PDB1, PDB2)
[Dist, RMSD, Transf] = pdbsuperpose(PDB1, PDB2)
[Dist, RMSD, Transf, PBD2TX] = pdbsuperpose(PDB1, PDB2)

... = pdbsuperpose(..., 'ModelNum', ModelNumValue, ...)
... = pdbsuperpose(..., 'Scale', ScaleValue, ...)
... = pdbsuperpose(..., 'Translate', TranslateValue, ...)
... = pdbsuperpose(..., 'Reflection', ReflectionValue, ...)
... = pdbsuperpose(..., 'SeqAlign', SeqAlignValue, ...)
... = pdbsuperpose(..., 'Segment', SegmentValue, ...)
... = pdbsuperpose(..., 'Apply', ApplyValue, ...)
... = pdbsuperpose(..., 'Display', DisplayValue, ...)
```

Input Arguments

PDB1, *PDB2*	Protein structures represented by any of the following: • Character vector or string specifying a unique identifier for a protein structure record in the Protein Data Bank (PDB) database. • Variable containing a PDB-formatted MATLAB structure, such as returned by `getpdb` or `pdbread`. • Character vector or string specifying a file name or, a path and file name. The referenced file is a PDB-formatted file. If you specify only a file name, that file must be on the MATLAB search path or in the MATLAB Current Folder.
ModelNumValue	Two-element numeric array whose elements correspond to models in *PDB1* and *PDB2* respectively when *PDB1* or *PDB2* contains multiple models. It specifies the models to consider in the superposition. By default, the first model in each structure is considered.
ScaleValue	Specifies whether to include a scaling component in the linear transformation. Choices are `true` or `false` (default).
TranslateValue	Specifies whether to include a translation component in the linear transformation. Choices are `true` (default) or `false`.

ReflectionValue	Specifies whether to include a reflection component in the linear transformation. Choices are: • `true` — Include reflection component. • `false` — Exclude reflection component. • `'best'` — Default. May or may not include the reflection component, depending on the best fit solution.
SeqAlignValue	Specifies whether to perform a local sequence alignment and then use only the portions of the structures corresponding to the segments that align to compute the linear transformation. Choices are `true` (default) or `false`. **Note** If you set the `'SeqAlign'` property to `true`, you can also specify the following properties used by the `swalign` function: • `'ScoringMatrix'` • `'GapOpen'` • `'ExtendGap'` For more information on these properties, see `swalign`.
SegmentValue	Specifies the boundaries and the chain of two subsequences to consider for computing the linear transformation. *SegmentValue* is a cell array of character vectors with the following format: `{'start1-stop1:chain1', 'start2-stop2:chain2'}` You can omit the boundaries to indicate the entire chain, such as in `{'chain1', 'start2-stop2:chain2'}`. You can specify only one pair of segments at any given time, and the specified segments are assumed to contain the same number of alpha carbon atoms.
ApplyValue	Specifies the extent to which the linear transformation should be applied. Choices are: • `'all'` — Default. Apply the linear transformation to the entire PDB2 structure. • `'chain'` — Apply the linear transformation to the specified chain only. • `'segment'` — Apply the linear transformation to the specified segment only.
DisplayValue	**Warning** This name-value argument will be removed in a future release. Specifies whether to display the original *PDB1* structure and the resulting transformed *PDB2TX* structure in the Molecule Viewer window using the `molviewer` function. Each structure is represented as a separate model. Choices are `true` or `false` (default).

Output Arguments

Dist	Value representing a dissimilarity measure given by the sum of the squared errors between *PDB1* and *PDB2*. For more information, see `procrustes`.
RMSD	Scalar representing the root mean square distance between the coordinates of the *PDB1* structure and the transformed *PDB2* structure, considering only the atoms used to compute the linear transformation.
Transf	Linear transformation computed to superpose the chain of *PDB2* to the chain of *PDB1*. *Transf* is a MATLAB structure with the following fields: • `T` — Orthogonal rotation and reflection component. • `b` — Scale component. • `c` — Translation component. **Note** Only alpha carbon atom coordinates are used to compute the linear transformation. **Tip** You can use the *Transf* output as input to the `pdbtransform` function.
PDB2TX	PDB-formatted MATLAB structure that represents the coordinates in the transformed *PDB2* protein structure.

Description

`pdbsuperpose(`*PDB1*`, `*PDB2*`)` computes and applies a linear transformation to superpose the coordinates of the protein structure represented in *PDB2* to the coordinates of the protein structure represented in *PDB1*. *PDB1* and *PDB2* are protein structures represented by any of the following:

- Character vector or string specifying a unique identifier for a protein structure record in the PDB database.
- Variable containing a PDB-formatted MATLAB structure, such as returned by `getpdb` or `pdbread`.
- Character vector or string specifying a file name or a path and file name. The referenced file is a PDB-formatted file. If you specify only a file name, that file must be on the MATLAB search path or in the MATLAB Current Folder.

Alpha carbon atom coordinates of single chains for each structure are considered to compute the linear transformation (translation, reflection, orthogonal rotation, and scaling). By default, the first chain in each structure is considered to compute the transformation, and the transformation is applied to the entire molecule. By default, the original *PDB1* structure and the resulting transformed *PDB2* structure are displayed as separate models in the Molecule Viewer window using the `molviewer` function.

Dist `= pdbsuperpose(`*PDB1*`, `*PDB2*`)` returns a dissimilarity measure given by the sum of the squared errors between *PDB1* and *PDB2*. For more information, see `procrustes`.

`[`*Dist*`, `*RMSD*`] = pdbsuperpose(`*PDB1*`, `*PDB2*`)` also returns *RMSD*, the root mean square distance between the coordinates of the *PDB1* structure and the transformed *PDB2* structure, considering only the atoms used to compute the linear transformation.

[*Dist*, *RMSD*, *Transf*] = pdbsuperpose(*PDB1*, *PDB2*) also returns *Transf*, the linear transformation computed to superpose the chain of *PDB2* to the chain of *PDB1*. *Transf* is a MATLAB structure with the following fields:

- T — Orthogonal rotation and reflection component.
- b — Scale component.
- c — Translation component.

Note Only alpha carbon atom coordinates are used to compute the linear transformation.

[*Dist*, *RMSD*, *Transf*, *PBD2TX*] = pdbsuperpose(*PDB1*, *PDB2*) also returns *PBD2TX*, a PDB-formatted MATLAB structure that represents the coordinates in the transformed *PDB2* protein structure.

... = pdbsuperpose(..., '*PropertyName*', *PropertyValue*, ...) calls pdbsuperpose with optional properties that use property name/property value pairs. You can specify one or more properties in any order. Each *PropertyName* must be enclosed in single quotation marks and is case insensitive. These property name/property value pairs are as follows:

... = pdbsuperpose(..., 'ModelNum', *ModelNumValue*, ...) specifies the models to consider in the superposition when *PDB1* or *PDB2* contains multiple models. *ModelNumValue* is a two-element numeric array whose elements correspond to the models in *PDB1* and *PDB2* respectively. By default, the first model in each structure is considered.

... = pdbsuperpose(..., 'Scale', *ScaleValue*, ...) specifies whether to include a scaling component in the linear transformation. Choices are true or false (default).

... = pdbsuperpose(..., 'Translate', *TranslateValue*, ...) specifies whether to include a translation component in the linear transformation. Choices are true (default) or false.

... = pdbsuperpose(..., 'Reflection', *ReflectionValue*, ...) specifies whether to include a reflection component in the linear transformation. Choices are true (include reflection component), false (exclude reflection component), or 'best' (may or may not include the reflection component, depending on the best fit solution). Default is 'best'.

... = pdbsuperpose(..., 'SeqAlign', *SeqAlignValue*, ...) specifies whether to perform a local sequence alignment and then use only the portions of the structures corresponding to the segments that align to compute the linear transformation. Choices are true (default) or false.

Note If you set the 'SeqAlign' property to true, you can also specify the following properties used by the swalign function:

- 'ScoringMatrix'
- 'GapOpen'
- 'ExtendGap'

For more information on these properties, see swalign.

... = pdbsuperpose(..., 'Segment', *SegmentValue*, ...) specifies the boundaries and the chain of two subsequences to consider for computing the linear transformation. *SegmentValue* is a cell array of character vectors with the following format: `{'start1-stop1:chain1', 'start2-stop2:chain2'}`. You can omit the boundaries to indicate the entire chain, such as in `{'chain1', 'start2-stop2:chain2'}`. You can specify only one pair of segments at any given time, and the specified segments are assumed to contain the same number of alpha carbon atoms.

`... = pdbsuperpose(..., 'Apply', ApplyValue, ...)` specifies the extent to which the linear transformation should be applied. Choices are `'all'` (apply the linear transformation to the entire PDB2 structure), `'chain'` (apply the linear transformation to the specified chain only), or `'segment'` (apply the linear transformation to the specified segment only). Default is `'all'`.

`... = pdbsuperpose(..., 'Display', DisplayValue, ...)` specifies whether to display the original *PDB1* structure and the resulting transformed *PDB2TX* structure in the Molecule Viewer window using the `molviewer` function. Each structure is represented as a separate model. Choices are `true` (default) or `false`.

Examples

Example 1.35. Superposing Two Hemoglobin Structures

1 Use the `getpdb` function to retrieve protein structure data from the Protein Data Bank (PDB) database for two hemoglobin structures.

```
str1 = getpdb('1dke');
str2 = getpdb('4hhb');
```

2 Superpose the first model of the two hemoglobin structures, applying the transformation to the entire molecule.

```
d = pdbsuperpose(str1, str2, 'model', [1 1], 'apply', 'all');
```

3 Superpose the two hemoglobin structures (each containing four chains), computing and applying the linear transformation chain by chain. Do not display the structures.

```
strtx = str2;
chainList1 = {str1.Sequence.ChainID};
chainList2 = {str2.Sequence.ChainID};
for i = 1:4
    [d(i), rmsd(i), tr(i), strtx] = pdbsuperpose(str1, strtx,  ...
                             'segment', {chainList1{i}; chainList2{i}}, ...
                             'apply', 'chain', 'display', false);
end
```

Example 1.36. Superposing Two Chains of a Thioredoxin Structure

Superpose chain B on chain A of a thioredoxin structure (PDBID = 2trx), and then apply the transformation only to chain B.

```
[d, rmsd, tr] = pdbsuperpose('2trx', '2trx', 'segment', {'A', 'B'}, ...
                             'apply', 'chain')

d =

    0.0028

rmsd =

    0.6604

tr =

    T: [3x3 double]
    b: 1
    c: [109x3 double]
```

Example 1.37. Superposing Two Calmodulin Structures

Superpose two calmodulin structures according to the linear transformation obtained using two 20 residue-long segments.

```
pdbsuperpose('1a29', '1cll', 'segment', {'10-30:A', '10-30:A'})

ans =

    0.1945
```

Version History

Introduced in R2008b

R2023a: `display` name-value argument of `pdbsuperpose` will be removed in a future release

Warns starting in R2023a

The `display` name-value argument of `pdbsuperpose` will be removed in a future release.

R2023a: `pdbsuperpose` no longer opens Molecule Viewer

Behavior changed in R2023a

`pdbsuperpose` no longer opens the **Molecule Viewer** app to display the superposed 3-D protein structures. In other words, the default value of the `display` name-value argument is now `false`.

See Also

`getpdb` | `pdbread` | `pdbtransform` | `swalign` | `procrustes`

pdbtransform

Apply linear transformation to 3-D structure of molecule

Syntax

```
pdbtransform(PDB, Transf)
PDBTX = pdbtransform(PDB, Transf)

... = pdbtransform(..., 'ModelNum', ModelNumValue, ...)
... = pdbtransform(..., 'Segment', SegmentValue, ...)
```

Input Arguments

PDB	Protein structure represented by any of the following: • Character vector or string specifying a unique identifier for a protein structure record in the Protein Data Bank (PDB) database. • Variable containing a PDB-formatted MATLAB structure, such as returned by `getpdb` or `pdbread`. • Character vector or string specifying a file name or a path and file name. The referenced file is a PDB-formatted file. If you specify only a file name, that file must be on the MATLAB search path or in the MATLAB Current Folder.
Transf	MATLAB structure representing a linear transformation, which is applied to the coordinates of the molecule represented by *PDB*. *Transf* contains the following fields: • `T` — Orthogonal rotation and reflection component. • `b` — Scale component. • `c` — Translation component. **Tip** You can use the *Transf* structure returned by the `pdbsuperpose` function as input.
ModelNumValue	Positive integer that specifies the model to which to apply the transformation, when *PDB* contains multiple models. By default, the first model is considered.
SegmentValue	Specifies the extent to which the linear transformation is applied. *SegmentValue* can be either: • `'all'` — The transformation is applied to the entire PDB input. • Character vector or string specifying the boundaries and the chain to consider. It uses either of the following formats: `'start-stop:chain'` or `'chain'`. Omitting the boundaries indicates the entire chain.

Output Arguments

PDBTX	Transformed PDB-formatted MATLAB structure.

Description

`pdbtransform(`*`PDB`*`, `*`Transf`*`)` applies the linear transformation specified in *`Transf`*, a MATLAB structure representing a linear transformation, to the coordinates of the molecule represented by *`PDB`*, which can be any of the following:

- Character vector or string specifying a unique identifier for a protein structure record in the PDB database.
- Variable containing a PDB-formatted MATLAB structure, such as returned by `getpdb` or `pdbread`.
- Character vector or string specifying a file name or a path and file name. The referenced file is a PDB-formatted file. If you specify only a file name, that file must be on the MATLAB search path or in the MATLAB Current Folder.

`PDBTX`` = pdbtransform(`*`PDB`*`, `*`Transf`*`)` returns *`PDBTX`*, the transformed PDB-formatted MATLAB structure.

`... = pdbtransform(...'`*`PropertyName`*`', `*`PropertyValue`*`, ...)` calls `pdbtransform` with optional properties that use property name/property value pairs. You can specify one or more properties in any order. Each *`PropertyName`* must be enclosed in single quotation marks and is case insensitive. These property name/property value pairs are as follows:

`... = pdbtransform(..., 'ModelNum', `*`ModelNumValue`*`, ...)` specifies the model to which to apply the transformation, when *`PDB`* contains multiple models. *`ModelNumValue`* is a positive integer. By default, the first model is considered.

`... = pdbtransform(..., 'Segment', `*`SegmentValue`*`, ...)` specifies the extent to which the linear transformation is applied. *`SegmentValue`* can be either:

- `'all'` — The transformation is applied to the entire PDB input.
- Character vector or string specifying the boundaries and the chain to consider. It uses either of the following formats: `'start-stop:chain'` or `'chain'`. Omitting the boundaries indicates the entire chain.

Examples

1. Create a MATLAB structure that defines a linear transformation.

   ```
   transf.T = eye(3);  transf.b = 1;  transf.c = [11.8 -2.8 -32.3];
   ```
2. Apply the linear transformation to chain B in the thioredoxin structure, with a PDB identifier of 2trx.

   ```
   pdbtx = pdbtransform('2trx', transf, 'segment', 'B');
   ```

Version History

Introduced in R2008b

See Also

`getpdb` | `pdbread` | `pdbsuperpose` | `procrustes`

pdbwrite

Write to file using Protein Data Bank (PDB) format

Syntax

```
pdbwrite(File, PDBStruct)
PDBArray = pdbwrite(File, PDBStruct)
```

Input Arguments

`File`	Character vector or string specifying either a file name or a path and file name for saving the PDB-formatted data. If you specify only a file name, the file is saved to the MATLAB Current Folder.
`PDBStruct`	MATLAB structure containing 3-D protein structure coordinate data, created initially by using the `getpdb` or `pdbread` functions. **Note** You can edit this structure to modify its 3-D protein structure data. The coordinate information is stored in the `Model` field of *`PDBStruct`*.

Output Arguments

`PDBArray`	Character array in which each row corresponds to a line in a PDB record.

Description

`pdbwrite(`*`File`*`, `*`PDBStruct`*`)` writes the contents of the MATLAB structure *`PDBStruct`* to a PDB-formatted file (ASCII text file) whose path and file name are specified by *`File`*. In the output file, *`File`*, the atom serial numbers are preserved. The atomic coordinate records are ordered according to their atom serial numbers.

`PDBArray`` = pdbwrite(`*`File`*`, `*`PDBStruct`*`)` saves the formatted PDB record, converted from the contents of the MATLAB structure *`PDBStruct`*, to *`PDBArray`*, a character array in which each row corresponds to a line in a PDB record.

Note You can edit *`PDBStruct`* to modify its 3-D protein structure data. The coordinate information is stored in the `Model` field of *`PDBStruct`*.

Examples

1 Use the `getpdb` function to retrieve structure information from the Protein Data Bank (PDB) for the green fluorescent protein with identifier `1GFL` , and store the data in the MATLAB structure `gflstruct`.

```
gflstruct = getpdb('1GFL');
```

2. Find the *x*-coordinate of the first atom.

   ```
   gflstruct.Model.Atom(1).X

   ans =

     -14.0930
   ```

3. Edit the *x*-coordinate of the first atom.

   ```
   gflstruct.Model.Atom(1).X = -18;
   ```

   Note Do not add or remove any `Atom` fields, because the `pdbwrite` function does not allow the number of elements in the structure to change.

4. Write the modified MATLAB structure `gflstruct` to a new PDB-formatted file `modified_gfl.pdb` in the `Work` folder on your `C` drive.

   ```
   pdbwrite('c:\work\modified_gfl.pdb', gflstruct);
   ```

5. Use the `pdbread` function to read the modified PDB file into a MATLAB structure, then confirm that the *x*-coordinate of the first atom has changed.

   ```
   modified_gflstruct = pdbread('c:\work\modified_gfl.pdb')
   modified_gflstruct.Model.Atom(1).X

   ans =

      -18
   ```

Version History

Introduced in R2007a

See Also

`getpdb` | `pdbread`

pdist (phytree)

Calculate pairwise patristic distances in phytree object

Syntax

```
D = pdist(Tree)
[D, C] = pdist(Tree)

pdist(..., 'Nodes', NodesValue, ...)
pdist(..., 'Squareform', SquareformValue, ...)
pdist(..., 'Criteria', CriteriaValue, ...)
```

Arguments

Tree	phytree object created by `phytree` function (object constructor) or `phytreeread` function.
NodesValue	Character vector or string specifying the nodes included in the computation. Choices are `'leaves'` (default) or `'all'`.
SquareformValue	Controls the creation of a square matrix. Choices are `true` or `false` (default).
CriteriaValue	Character vector or string specifying the criteria used to relate pairs. Choices are `'distance'` (default) or `'levels'`.

Description

`D = pdist(Tree)` returns *D*, a vector containing the patristic distances between every possible pair of leaf nodes of *Tree*, a phylogenetic tree object. The patristic distances are computed by following paths through the branches of the tree and adding the patristic branch distances originally created with the `seqlinkage` function.

The output vector *D* is arranged in the order `((2,1), (3,1), ..., (M,1), (3,2), ..., (M,2), ..., (M,M-1))` (the lower-left triangle of the full `M`-by-`M` distance matrix). To get the distance between the `I`th and `J`th nodes `(I > J)`, use the formula `D((J-1)*(M-J/2)+I-J)`. `M` is the number of leaves.

`[D, C] = pdist(Tree)` returns in *C*, the index of the closest common parent nodes for every possible pair of query nodes.

`pdist(..., 'PropertyName', PropertyValue, ...)` calls `pdist` with optional properties that use property name/property value pairs. You can specify one or more properties in any order. Each *PropertyName* must be enclosed in single quotation marks and is case insensitive. These property name/property value pairs are as follows:

`pdist(..., 'Nodes', NodesValue, ...)` specifies the nodes included in the computation. Choices are `'leaves'` (default) or `'all'`. When *NodesValue* is `'leaves'`, the output is ordered as before, but `M` is the total number of nodes in the tree (`NumLeaves+NumBranches`).

`pdist(..., 'Squareform', *SquareformValue*, ...)` controls the creation of a square matrix. Choices are `true` or `false` (default). When *`SquareformValue`* is `true`, `pdist` converts the output into a square-formatted matrix, so that `D(I,J)` denotes the distance between the `I`th and the `J`th nodes. The output matrix is symmetric and has a zero diagonal.

`pdist(..., 'Criteria', *CriteriaValue*, ...)` changes the criteria used to relate pairs. *`CriteriaValue`* can be `'distance'` (default) or `'levels'`.

Examples

1. Read a phylogenetic tree file into a phytree object.

   ```
   tr = phytreeread('pf00002.tree')
   ```
2. Calculate the tree distances between pairs of leaves.

   ```
   dist = pdist(tr,'nodes','leaves','squareform',true)
   ```

Version History

Introduced before R2006a

See Also

`phytree` | `phytreeread` | `phytreeviewer` | `seqlinkage` | `seqpdist`

Topics

phytree object on page 1-1449

pfamhmmread

Read data from PFAM HMM-formatted file

Syntax

```
HMMStruct = pfamhmmread(File)
HMMStruct = pfamhmmread(File,'TimeOut',TimeOutValue)
```

Input Arguments

File	Character vector or string specifying a file name, a path and file name, a URL pointing to a file, or the text of a PFAM-HMM-formatted file. The referenced file is a PFAM HMM-formatted file. If you specify only a file name, that file must be on the MATLAB search path or in the current folder. **Tip** You can use the `gethmmprof` function with the `'ToFile'` property to retrieve HMM profile information from the PFAM database and create a PFAM HMM-formatted file.
TimeOutValue	Connection timeout in seconds, specified as a positive scalar. The default value is 5. For details, see here.

Output Arguments

HMMStruct	MATLAB structure containing information from a PFAM HMM-formatted file.

Description

Note `pfamhmmread` reads PFAM-HMM formatted files, from file format version HMMER2.0 to HMMER3/f.

`HMMStruct` `= pfamhmmread(`*`File`*`)` reads *`File`*, a PFAM HMM-formatted file, and converts it to *`HMMStruct`*, a MATLAB structure containing the following fields corresponding to parameters of an HMM profile:

Field	Description
`Name`	The protein family name (unique identifier) of the HMM profile record in the PFAM database.
`PfamAccessionNumber`	The protein family accession number of the HMM profile record in the PFAM database.
`ModelDescription`	Description of the HMM profile.
`ModelLength`	The length of the profile (number of MATCH states).

Field	Description
`Alphabet`	The alphabet used in the model, `'AA'` or `'NT'`. **Note** `AlphaLength` is 20 for `'AA'` and 4 for `'NT'`.
`MatchEmission`	Symbol emission probabilities in the MATCH states. The format is a matrix of size `ModelLength`-by-`AlphaLength`, where each row corresponds to the emission distribution for a specific MATCH state.
`InsertEmission`	Symbol emission probabilities in the INSERT state. The format is a matrix of size `ModelLength`-by-`AlphaLength`, where each row corresponds to the emission distribution for a specific INSERT state.
`NullEmission`	Symbol emission probabilities in the MATCH and INSERT states for the NULL model. The format is a 1-by-`AlphaLength` row vector. **Note** NULL probabilities are also known as the background probabilities.
`BeginX`	BEGIN state transition probabilities. Format is a 1-by-`(ModelLength + 1)` row vector: `[B->D1 B->M1 B->M2 B->M3 .... B->Mend]`
`MatchX`	MATCH state transition probabilities. Format is a 4-by-`(ModelLength - 1)` matrix: `[M1->M2 M2->M3 ... M[end-1]->Mend;` `M1->I1 M2->I2 ... M[end-1]->I[end-1];` `M1->D2 M2->D3 ... M[end-1]->Dend;` `M1->E  M2->E  ... M[end-1]->E  ]`
`InsertX`	INSERT state transition probabilities. Format is a 2-by-`(ModelLength - 1)` matrix: `[ I1->M2 I2->M3 ... I[end-1]->Mend;` `I1->I1 I2->I2 ... I[end-1]->I[end-1] ]`
`DeleteX`	DELETE state transition probabilities. Format is a 2-by-`(ModelLength - 1)` matrix: `[ D1->M2 D2->M3 ... D[end-1]->Mend ;` `D1->D2 D2->D3 ... D[end-1]->Dend ]`

Field	Description

`FlankingInsertX`	Flanking insert states (N and C) used for LOCAL profile alignment.
	
	Format is a 2-by-2 matrix:
	
	`[N->B  C->T ;`
	` N->N  C->C]`

`LoopX`	Loop states transition probabilities used for multiple hits alignment.
	
	Format is a 2-by-2 matrix:
	
	`[E->C  J->B ;`
	` E->J  J->J]`

`NullX`	Null transition probabilities used to provide scores with log-odds values also for state transitions. Format is a 2-by-1 column vector: `[G->F ; G->G]`

HMMStruct = pfamhmmread(*File*,'TimeOut',*TimeOutValue*) sets the connection timeout (in seconds) to retrieve data from the PFAM database.

For more information on HMM profile models, see “HMM Profile Model” on page 1-1108.

Examples

Read a locally saved PFAM HMM-formatted file into a MATLAB structure.

```
pfamhmmread('pf00002.ls')

ans =

                   Name: '7tm_2'
    PfamAccessionNumber: 'PF00002.15'
       ModelDescription: '7 transmembrane receptor (Secretin family)'
            ModelLength: 293
               Alphabet: 'AA'
          MatchEmission: [293x20 double]
         InsertEmission: [293x20 double]
           NullEmission: [1x20 double]
                 BeginX: [294x1 double]
                 MatchX: [292x4 double]
                InsertX: [292x2 double]
                DeleteX: [292x2 double]
        FlankingInsertX: [2x2 double]
                  LoopX: [2x2 double]
                  NullX: [2x1 double]
```

Version History

Introduced before R2006a

See Also

`gethmmalignment` | `gethmmprof` | `hmmprofalign` | `hmmprofstruct` | `showhmmprof`

phytree object

Data structure containing phylogenetic tree

Description

A phytree object is a data structure containing a phylogenetic tree. Phylogenetic trees are binary rooted trees, which means that each branch is the parent of two other branches, two leaves, or one branch and one leaf. A phytree object can be ultrametric or nonultrametric.

Method Summary

Following are methods of a phytree object:

cluster (phytree)	Validate clusters in phylogenetic tree
get (phytree)	Retrieve information about phylogenetic tree object
getbyname (phytree)	Branches and leaves from phytree object
getcanonical (phytree)	Calculate canonical form of phylogenetic tree
getmatrix (phytree)	Convert phytree object into relationship matrix
getnewickstr (phytree)	Create Newick-formatted character vector
pdist (phytree)	Calculate pairwise patristic distances in phytree object
plot (phytree)	Draw phylogenetic tree
prune (phytree)	Remove branch nodes from phylogenetic tree
reorder (phytree)	Reorder leaves of phylogenetic tree
reroot (phytree)	Change root of phylogenetic tree
select (phytree)	Select tree branches and leaves in phytree object
subtree (phytree)	Extract phylogenetic subtree
view (phytree)	View phylogenetic tree
weights (phytree)	Calculate weights for phylogenetic tree

Property Summary

Note You cannot modify these properties directly. You can access these properties using the `get` method.

Property	Description
`NumLeaves`	Number of leaves
`NumBranches`	Number of branches
`NumNodes`	Number of nodes (`NumLeaves + NumBranches`)
`Pointers`	Branch to leaf/branch connectivity list

Property	Description
`Distances`	Edge length for every leaf/branch
`LeafNames`	Names of the leaves
`BranchNames`	Names of the branches
`NodeNames`	Names of all the nodes

Version History

Introduced in R2006b

See Also

`phytree` | `phytreeread` | `phytreeviewer` | `phytreewrite` | `seqlinkage` | `seqneighjoin` | `seqpdist` | `cluster` | `get` | `getbyname` | `getcanonical` | `getmatrix` | `getnewickstr` | `pdist` | `plot` | `prune` | `reroot` | `select` | `subtree` | `view` | `weights`

phytree

Create phytree object

Syntax

```
Tree = phytree(B)
Tree = phytree(B, D)
Tree = phytree(B, C)
Tree = phytree(BC)
Tree = phytree(..., N)
Tree = phytree
```

Arguments

B	Numeric array of size `[NUMBRANCHES X 2]` in which every row represents a branch of the tree. It contains two pointers to the branch or leaf nodes, which are its children.
C	Column vector with distances for every branch.
D	Column vector with distances from every node to their parent branch.
BC	Combined matrix with pointers to branches or leaves, and distances of branches.
N	Cell array with the names of leaves and branches.

Description

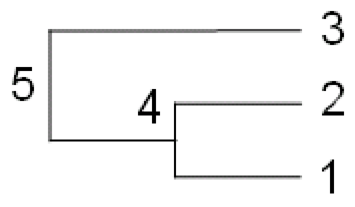
`Tree = phytree(B)` creates an ultrametric phylogenetic tree object. In an ultrametric phylogenetic tree object, all leaves are the same distance from the root.

B is a numeric array of size `[NUMBRANCHES X 2]` in which every row represents a branch of the tree and it contains two pointers to the branch or leaf nodes, which are its children.

Leaf nodes are numbered from `1` to `NUMLEAVES` and branch nodes are numbered from `NUMLEAVES + 1` to `NUMLEAVES + NUMBRANCHES`. Note that because only binary trees are allowed, `NUMLEAVES = NUMBRANCHES + 1`.

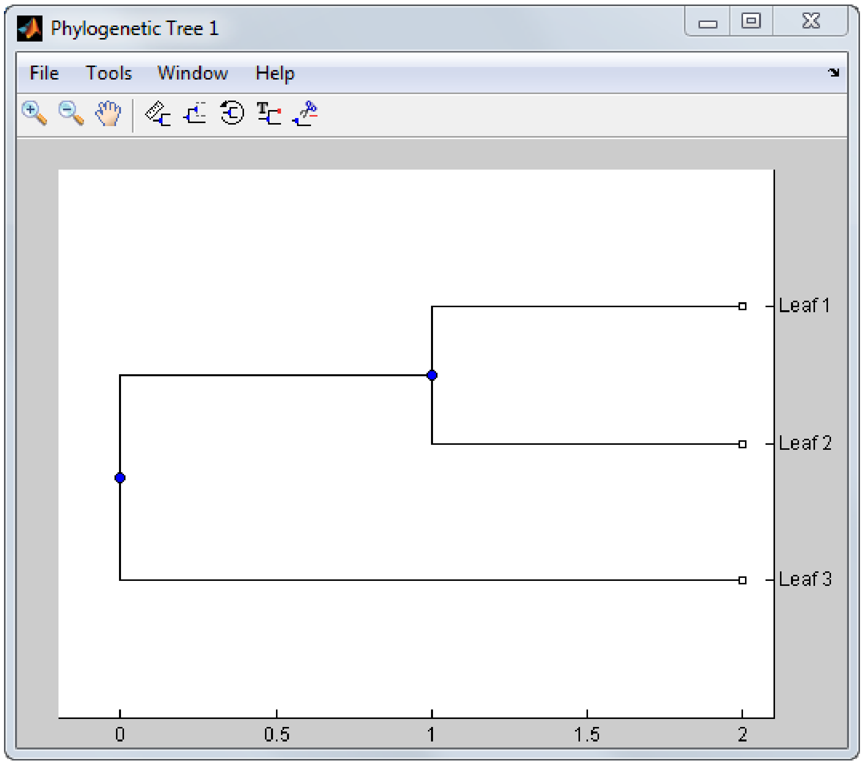
Branches are defined in chronological order (for example, `B(i,:) > NUMLEAVES + i`). As a consequence, the first row can only have pointers to leaves, and the last row must represent the root branch. Parent-child distances are set to `1`, unless the child is a leaf and to satisfy the ultrametric condition of the tree its distance is increased.

Given a tree with three leaves and two branches as an example.



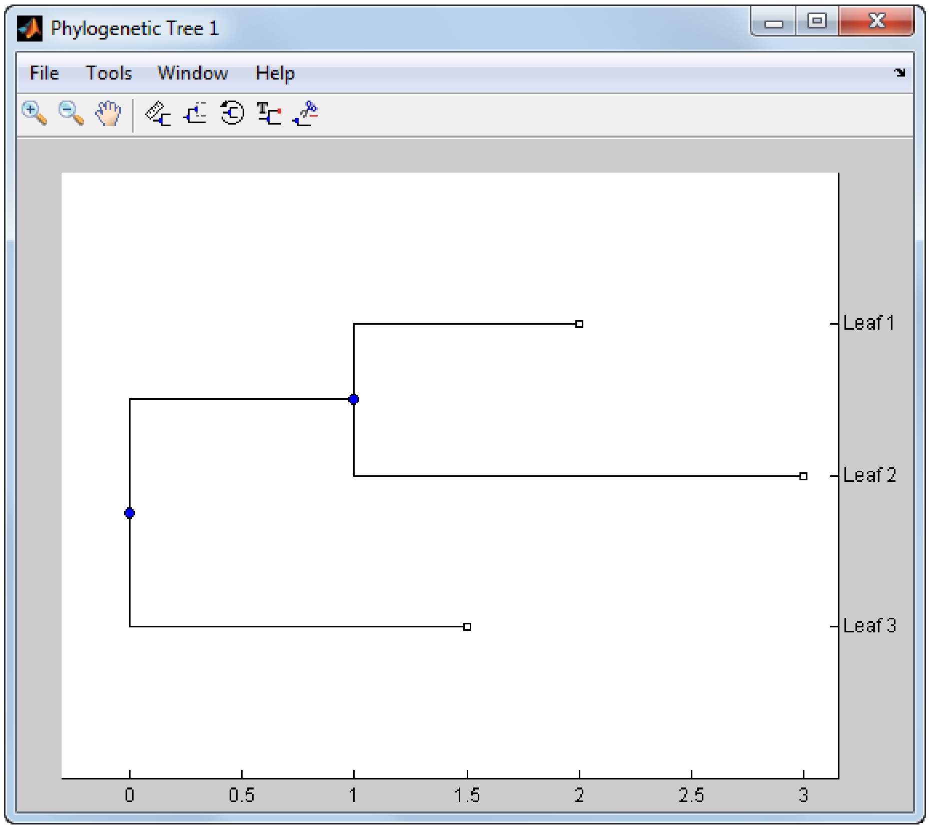
In the MATLAB Command Window, type

```
B = [1 2 ; 3 4]

 B =

     1     2
     3     4

tree = phytree(B)

 Phylogenetic tree object with 3 leaves (2 branches)

view(tree)
```



`Tree = phytree(B, D)` creates an additive (ultrametric or nonultrametric) phylogenetic tree object with branch distances defined by *D*. *D* is a numeric array of size `[NUMNODES X 1]` with the distances of every child node (leaf or branch) to its parent branch equal to `NUMNODES = NUMLEAVES + NUMBRANCHES`. The last distance in *D* is the distance of the root node and is meaningless.

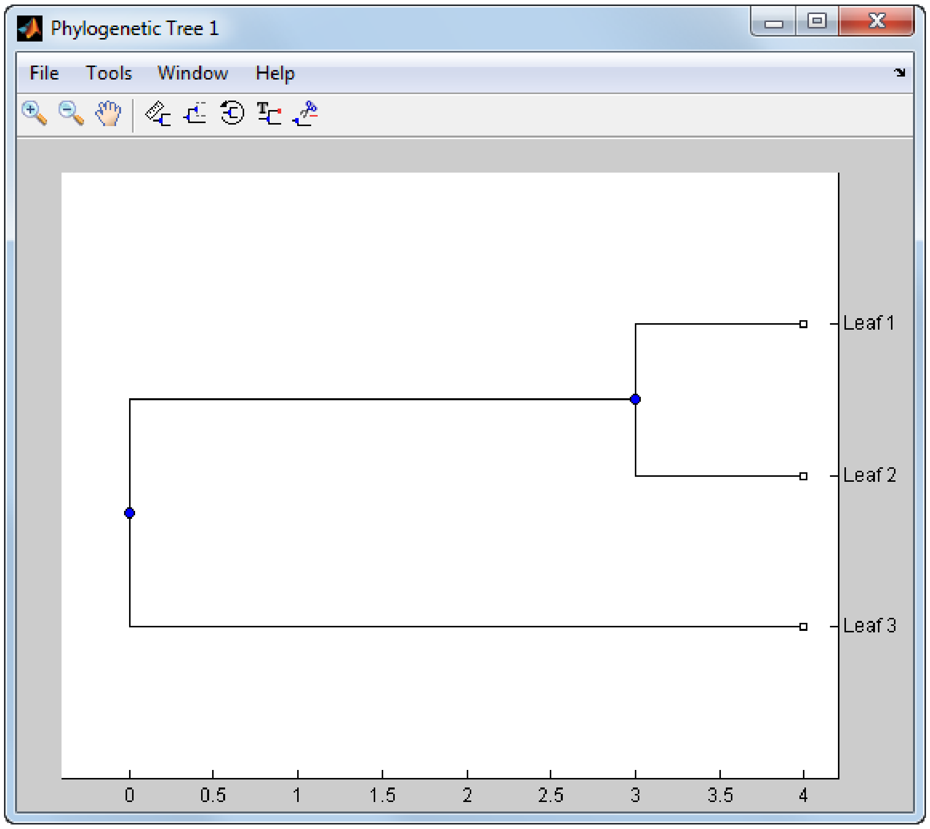
```
b = [1 2 ; 3 4 ]

b =

     1     2
     3     4

d = [1; 2; 1.5; 1; 0]

d =
    1.0000
    2.0000
    1.5000
    1.0000
         0

view(phytree(b,d))
```



`Tree = phytree(B, C)` creates an ultrametric phylogenetic tree object with distances between branches and leaves defined by `C`. `C` is a numeric array of size `[NUMBRANCHES X 1]`, which contains the distance from each branch to the leaves. In ultrametric trees, all of the leaves are at the same location (same distance to the root).

```
b = [1 2 ; 3 4]

b =

     1     2
     3     4

c = [1 4]'

c =

     1
     4
```

```
view(phytree(b,c))
```



`Tree` `= phytree(`*`BC`*`)` creates an ultrametric phylogenetic binary tree object with branch pointers in `BC(:,[1 2])` and branch coordinates in `BC(:,3)`. Same as `phytree(B,C)`.

`Tree` `= phytree(...,` *`N`*`)` specifies the names for the leaves and/or the branches. *N* is a string vector or cell array of character vectors. If `NUMEL(N)==NUMLEAVES`, then the names are assigned chronologically to the leaves. If `NUMEL(N)==NUMBRANCHES`, the names are assigned to the branch nodes. If `NUMEL(N)==NUMLEAVES + NUMBRANCHES`, all the nodes are named. Unassigned names default to `'Leaf #'` and/or `'Branch #'` as required.

`Tree` `= phytree` creates an empty phylogenetic tree object.

Examples

Create a Phylogenetic Tree

This example shows how to create a phylogenetic tree from a multiple sequence alignment file.

Read a multiple sequence alignment file.

```
Sequences = multialignread('aagag.aln');
```

Calculate the distance between each pair of sequences.

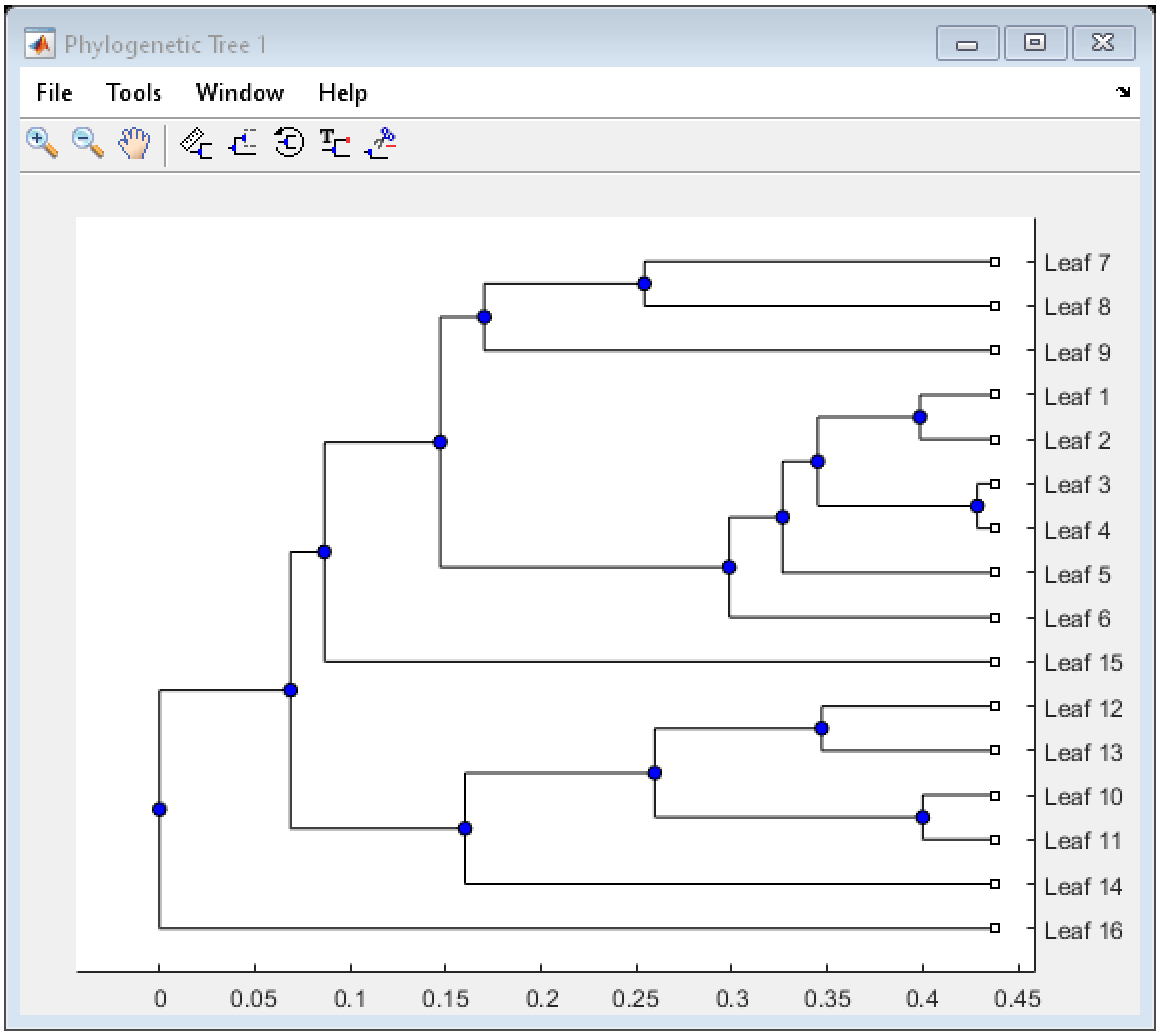
```
distances = seqpdist(Sequences);
```

Construct a phylogenetic tree object from the pairwise distances calculated previously.

```
tree = seqlinkage(distances);
```

View the phylogenetic tree.

```
phytreeviewer(tree)
```



Version History

Introduced in R2006a

See Also

phytreeread | phytreeviewer | phytreewrite | seqlinkage | seqneighjoin | seqpdist | cluster | get | getbyname | getcanonical | getmatrix | getnewickstr | pdist | plot | prune | reroot | select | subtree | view | weights

Topics
phytree object on page 1-1449

phytreeread

Read phylogenetic tree file

Syntax

```
Tree = phytreeread(File)
Tree = phytreeread(File,'TimeOut',TimeOutValue)
[Tree, Boot]= phytreeread(File)
```

Input Arguments

File	Character vector or string specifying a Newick- or Nexus-formatted tree file (ASCII text file) name, a path and file name, or a URL pointing to a file. For the Nexus tree format, only one tree from the first `TREES` block is read. This is either the last default tree (marked by an asterisk *) or the first tree, if no default trees exist.
TimeOutValue	Connection timeout in seconds, specified as a positive scalar. The default value is 5. For details, see here.

Output Arguments

Tree	`phytree` object created with the function `phytree`.
Boot	Column vector of bootstrap values for each tree node specified in *`File`*. If *`File`* does not specify a bootstrap value for a node, it returns a NaN value for that node. `phytreeread` considers the following values in *`File`* to be bootstrap values: • Values within square brackets (`[]`) after the branch or leaf node lengths (for Newick-formatted trees only) • Values that appear instead of branch or leaf node labels (for both Newick- and Nexus-formatted trees)

Description

`Tree = phytreeread(File)` reads a Newick- or Nexus-formatted tree file and returns a phytree object on page 1-1449 containing data from the file.

`Tree = phytreeread(File,'TimeOut',TimeOutValue)` sets the connection timeout (in seconds) to read data from a remote file or URL.

The NEWICK tree format can be found at:

`https://evolution.genetics.washington.edu/phylip/newicktree.html`

The NEXUS tree format can be found at:

`https://wiki.christophchamp.com/index.php/NEXUS_file_format`

Note This implementation allows only binary trees. Non-binary trees are translated into a binary tree with extra branches of length 0.

`[Tree, Boot]= phytreeread(File)` returns *`Boot`*, a column vector of bootstrap values for each tree node specified in *`File`*. If *`File`* does not specify a bootstrap value for a node, it returns a NaN value for that node. `phytreeread` considers the following values in *`File`* to be bootstrap values:

- Values within square brackets (`[]`) after the branch or leaf node lengths (for Newick-formatted trees only)
- Values that appear instead of branch or leaf node labels (for both Newick- and Nexus-formatted trees)

Examples

```
tr = phytreeread('pf00002.tree')
```

```
 Phylogenetic tree object with 33 leaves (32 branches)
```

```
tr2 = phytreeread('pf00002.nex')
```

```
Phylogenetic tree object with 33 leaves (32 branches)
```

Version History

Introduced before R2006a

See Also

`phytree` | `gethmmtree` | `phytreeviewer` | `phytreewrite`

Topics

phytree object on page 1-1449

phytreeviewer

Visualize, edit, and explore phylogenetic tree data

Syntax

```
phytreeviewer
phytreeviewer(Tree)
phytreeviewer(File)
```

Description

`phytreeviewer` opens the Phylogenetic Tree app that allows you to view, edit, and explore phylogenetic tree data.

`phytreeviewer(Tree)` loads a `phytree` object `Tree` into the app.

`phytreeviewer(File)` loads data from a Newick or ClustalW tree formatted file into the app.

Input Arguments

`Tree` — Phylogenetic tree
`Phytree` object

Phylogenetic tree, specified as a `Phytree` object created with the functions `phytree` or `phytreeread`.

`File` — Newick or ClustalW tree formatted file
character vector | string

Newick or ClustalW tree formatted file, specified as a character vector or string containing the file name, a path and file name, or a URL pointing to the file.

Examples

View a Phylogenetic Tree

This example shows how to view a phylogenetic tree.

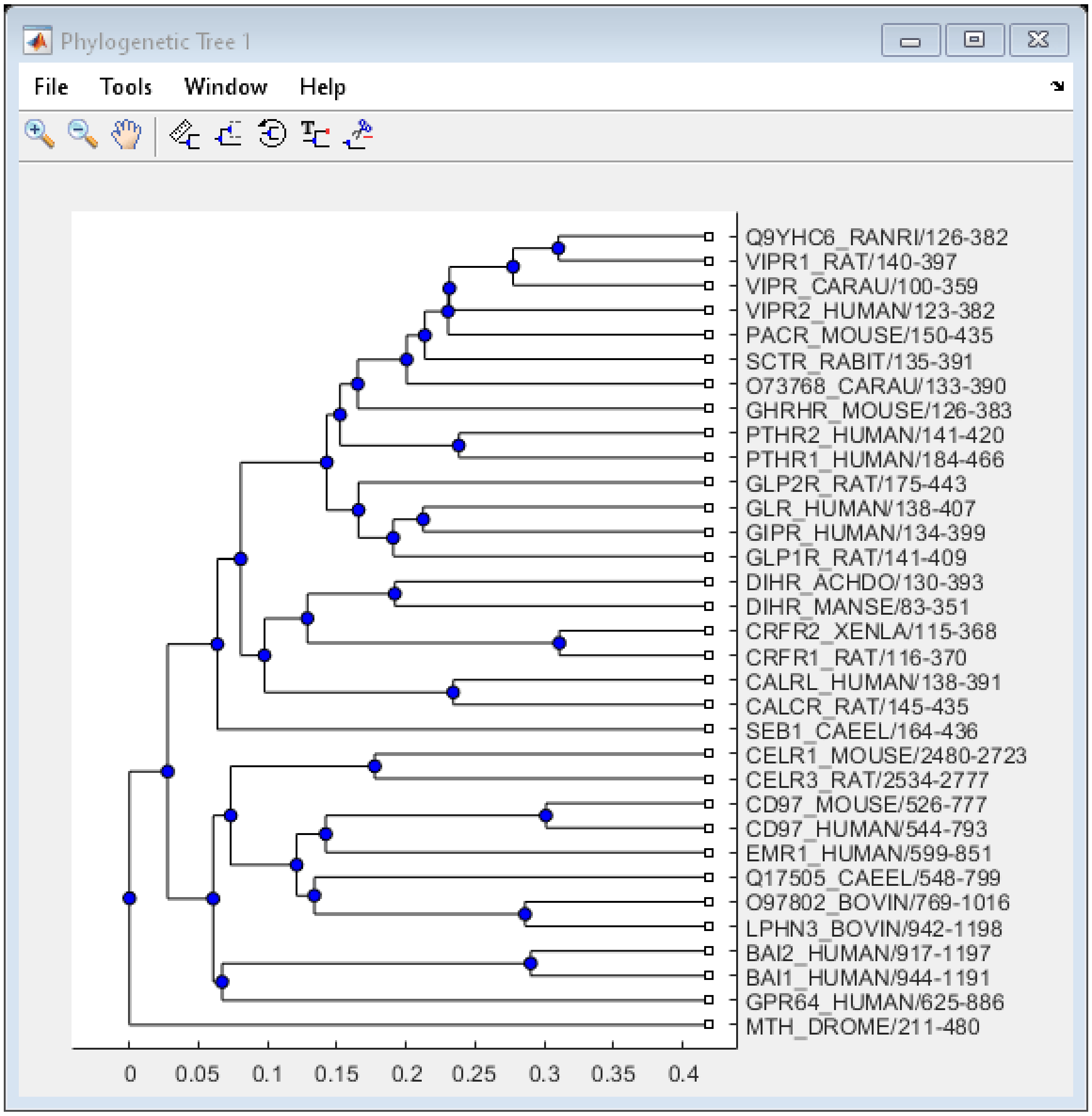
Load a sample phylogenetic tree.

```
tr= phytreeread('pf00002.tree')
```

```
    Phylogenetic tree object with 33 leaves (32 branches)
```

View the phylogenetic tree.

```
phytreeviewer(tr)
```



Alternatively, you can click Phylogenetic Tree on the **Apps** tab to open the app, and view the phylogenetic tree object `tr` .

Version History

Introduced in R2012b

See Also

`phytree` | `phytreeread` | `phytreewrite` | `cluster` | `plot` | `view`

phytreewrite

Write phylogenetic tree object to Newick-formatted file

Syntax

```
phytreewrite(File, Tree)
phytreewrite(Tree)

phytreewrite(..., 'Distances', DistancesValue, ...)
phytreewrite(..., 'BranchNames', BranchNamesValue, ...)
```

Arguments

`File`	Character vector or string specifying a Newick-formatted file (ASCII text file) name, a path and file name, or a URL pointing to a file.
`Tree`	Phylogenetic tree object, either created with `phytree` (object constructor function) or imported using the `phytreeread` function.

Description

`phytreewrite(`*`File`*`, `*`Tree`*`)` copies the contents of a `phytree` object from the MATLAB workspace to a file. Data in the file uses the Newick format for describing trees.

`phytreewrite(`*`Tree`*`)` opens the Save Phylogenetic Tree As dialog box for you to enter or select a file name.

`phytreewrite(..., '`*`PropertyName`*`', `*`PropertyValue`*`, ...)` calls `phytreewrite` with optional properties that use property name/property value pairs. You can specify one or more properties in any order. Enclose each *`PropertyName`* in single quotation marks. Each *`PropertyName`* is case insensitive. These property name/property value pairs are as follows:

`phytreewrite(..., 'Distances', `*`DistancesValue`*`, ...)` specifies whether to exclude the distances from the output. *`DistancesValue`* can be `true` (default) or `false`.

`phytreewrite(..., 'BranchNames', `*`BranchNamesValue`*`, ...)` specifies whether to exclude the branch names from the output. *`BranchNamesValue`* can be `true` (default) or `false`.

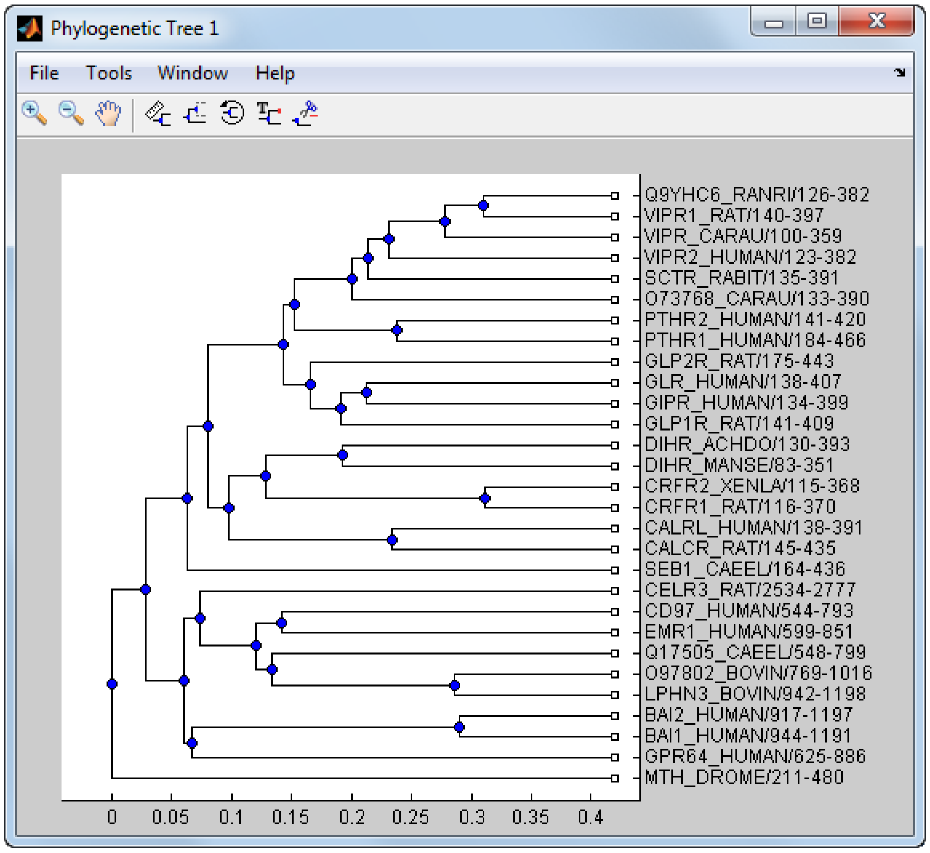
Examples

Read tree data from a Newick-formatted file.

```
tr = phytreeread('pf00002.tree')

 Phylogenetic tree object with 33 leaves (32 branches)
```

Remove all the mouse proteins and view the pruned tree.

```
ind = getbyname(tr,'mouse');
tr = prune(tr,ind);
view(tr)
```



Write pruned tree data to a file.

```
phytreewrite('newtree.tree',tr)
```

Version History

Introduced before R2006a

See Also

`multialignwrite` | `phytree` | `phytreeread` | `phytreeviewer` | `seqlinkage` | `phytree object`

Topics

`getnewickstr` on page 1-989

plot (DataMatrix)

Draw 2-D line plot of DataMatrix object

Syntax

```
plot(DMObj1)
plot(DMObj1, DMObj2)
plot(..., LineSpec)
```

Arguments

DMObj1, *DMObj2*	DataMatrix objects, such as created by `DataMatrix` (object constructor). **Note** If both *DMObj1* and *DMObj2* are input arguments, one of the inputs can be a MATLAB numeric array.
LineSpec	Character vector specifying a line style, marker symbol, and color of the plotted lines. For more information on these specifiers, see “Line Specifications” on page 1-1463.

Description

`plot(DMObj1)` plots the columns of a DataMatrix object *DMObj1* versus their index.

`plot(DMObj1, DMObj2)` plots the data from *DMObj1* and *DMObj2*, two DataMatrix objects, or one DataMatrix object and one MATLAB numeric array.

- If *DMObj1* and *DMObj2* are both vectors, they must have the same number of elements, and `plot` plots one vector versus the other vector, creating a single line.
- If one is a vector and one a scalar, `plot` plots discrete points vertically or horizontally, at the scalar value.
- If one is a vector and one a matrix, the number of elements in the vector must equal either the number of rows or the number of columns in the matrix, and `plot` plots the vector versus each row or column in the matrix.
- If both are matrices, they must have the same size (number of rows and columns), and `plot` plots each column in *DMObj1* versus the corresponding column in *DMObj2*.

`plot(..., LineSpec)` plots all lines as defined by *LineSpec*, a character vector specifying a line style, marker symbol, and/or color.

More About

Line Specifications

The following tables list available line styles, markers, and colors.

Line Style	Description	Resulting Line
"-"	Solid line	——
"--"	Dashed line	- - - -
":"	Dotted line	⋯⋯
"-."	Dash-dotted line	-·-·-·

Marker	Description	Resulting Marker
"o"	Circle	○
"+"	Plus sign	+
"*"	Asterisk	✳
"."	Point	•
"x"	Cross	×
"_"	Horizontal line	—
"|"	Vertical line	|
"square"	Square	□
"diamond"	Diamond	◇
"^"	Upward-pointing triangle	△
"v"	Downward-pointing triangle	▽
">"	Right-pointing triangle	▷
"<"	Left-pointing triangle	◁
"pentagram"	Pentagram	☆
"hexagram"	Hexagram	✡

Color Name	Short Name	RGB Triplet	Appearance
"red"	"r"	[1 0 0]	🟥
"green"	"g"	[0 1 0]	🟩
"blue"	"b"	[0 0 1]	🟦
"cyan"	"c"	[0 1 1]	🟦
"magenta"	"m"	[1 0 1]	🟪

Color Name	Short Name	RGB Triplet	Appearance
`"yellow"`	`"y"`	`[1 1 0]`	🟨
`"black"`	`"k"`	`[0 0 0]`	⬛
`"white"`	`"w"`	`[1 1 1]`	⬜

Version History

Introduced in R2008b

See Also

`DataMatrix` | `plot`

Topics

DataMatrix object on page 1-734

plot

Render heatmap or clustergram

Syntax

```
plot(hm_cg_object)
plot(hm_cg_object,hFig)
hAxes = plot( ___ )
```

Description

`plot(hm_cg_object)` renders a heatmap or clustergram of `hm_cg_object`.

`plot(hm_cg_object,hFig)` displays a heatmap or clustergram in a MATLAB figure specified by the figure handle `hFig`.

`hAxes = plot( ___ )` returns the handle to the axes of the heatmap or clustergram figure using any of the input argument combinations from the previous syntaxes.

Examples

Plot Heatmap of Data Matrix

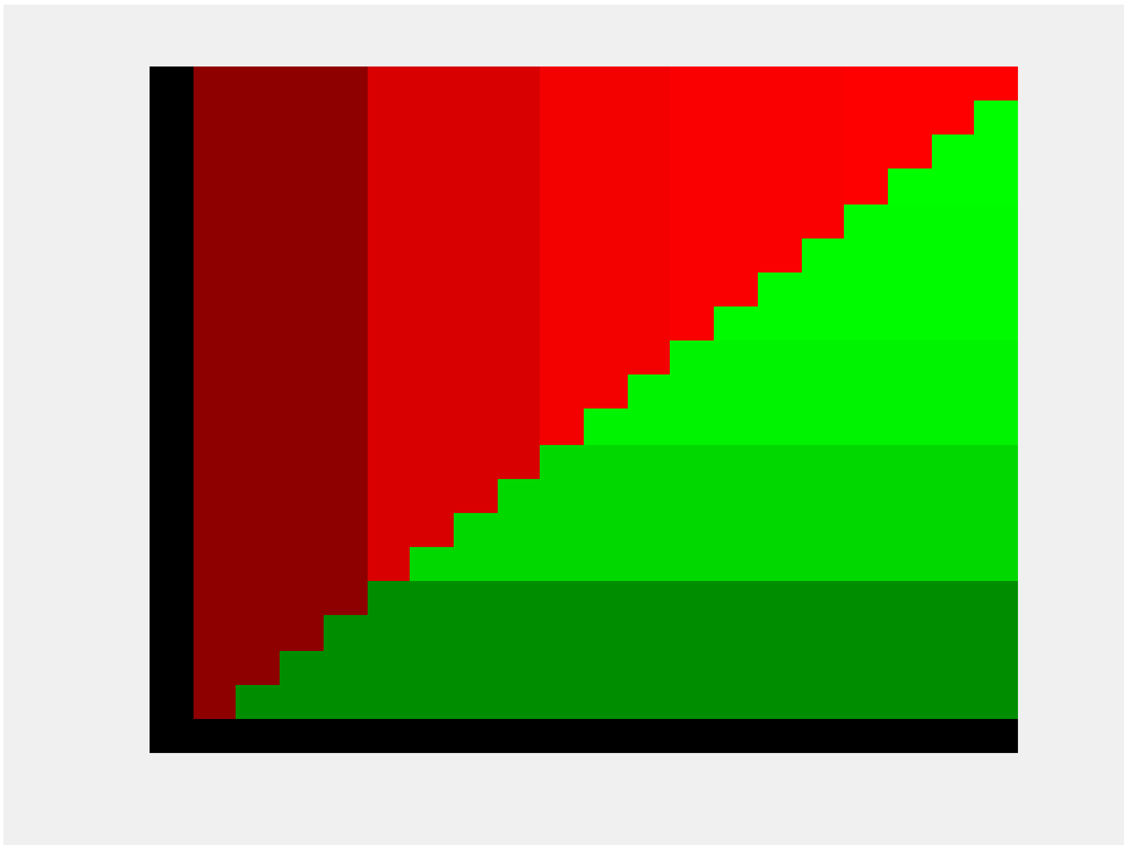
Create a matrix of data.

```
data = gallery('invhess',20);
```

Display a 2-D color heatmap of the data.

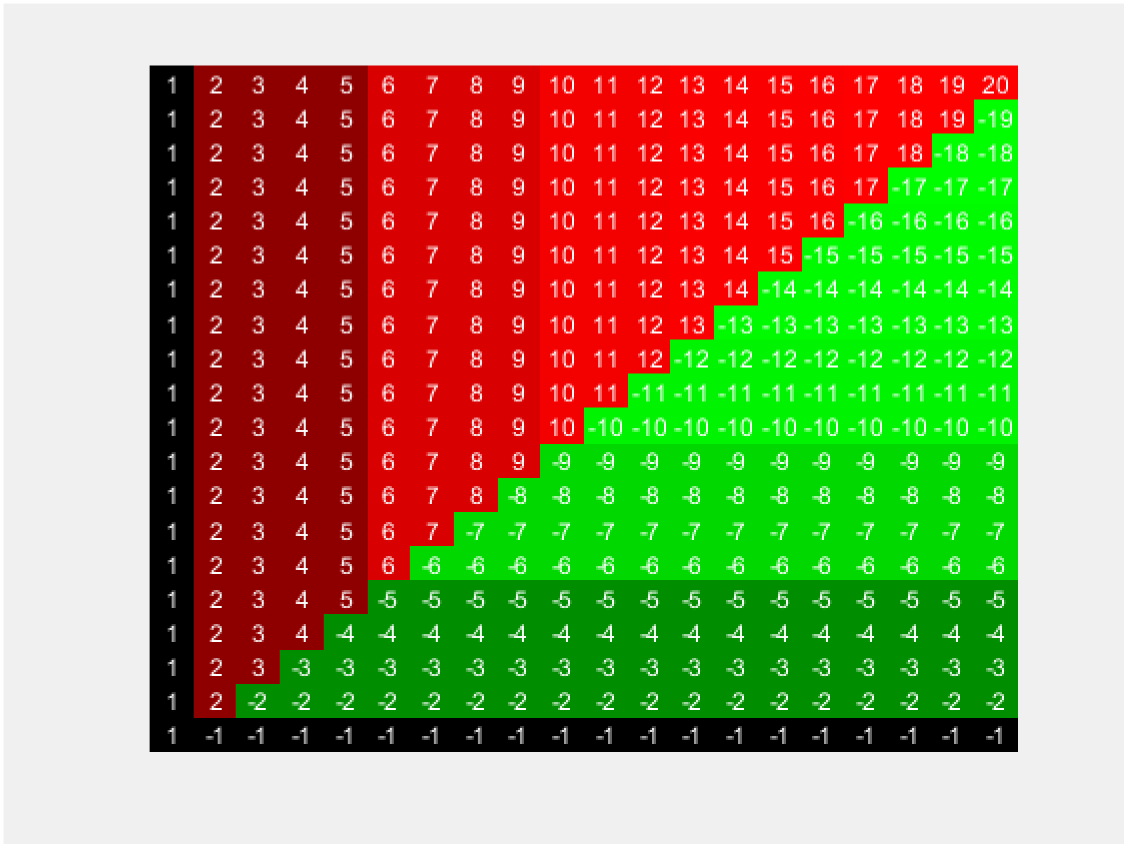
```
hmo = HeatMap(data);
```

```
             Standardize: '[column | row | {none}]'
               Symmetric: '[true | false].'
            DisplayRange: 'Scalar.'
                Colormap: []
               ImputeFun: 'string -or- function handle -or- cell array'
            ColumnLabels: 'Cell array of strings, or an empty cell array'
               RowLabels: 'Cell array of strings, or an empty cell array'
      ColumnLabelsRotate: []
         RowLabelsRotate: []
                Annotate: '[on | {off}]'
          AnnotPrecision: []
              AnnotColor: []
       ColumnLabelsColor: 'A structure array.'
          RowLabelsColor: 'A structure array.'
       LabelsWithMarkers: '[true | false].'
    ColumnLabelsLocation: '[ top | {bottom} ]'
       RowLabelsLocation: '[ {left} | right ]'
```



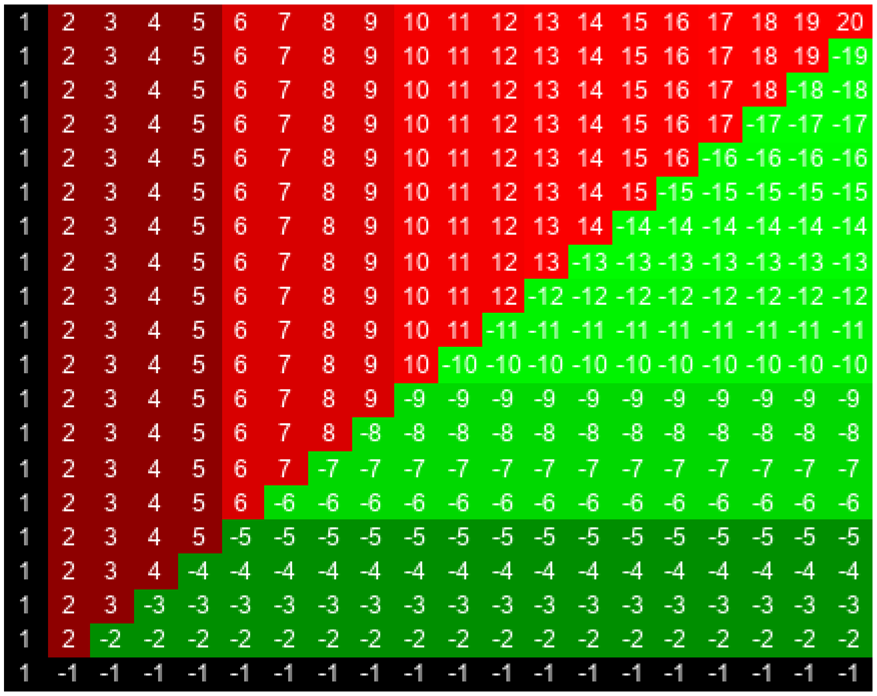
Display the data values in the heatmap.

```
hmo.Annotate = true;
view(hmo)
```



Use the `plot` function to display the heatmap in another figure specified by the figure handle `fH`.

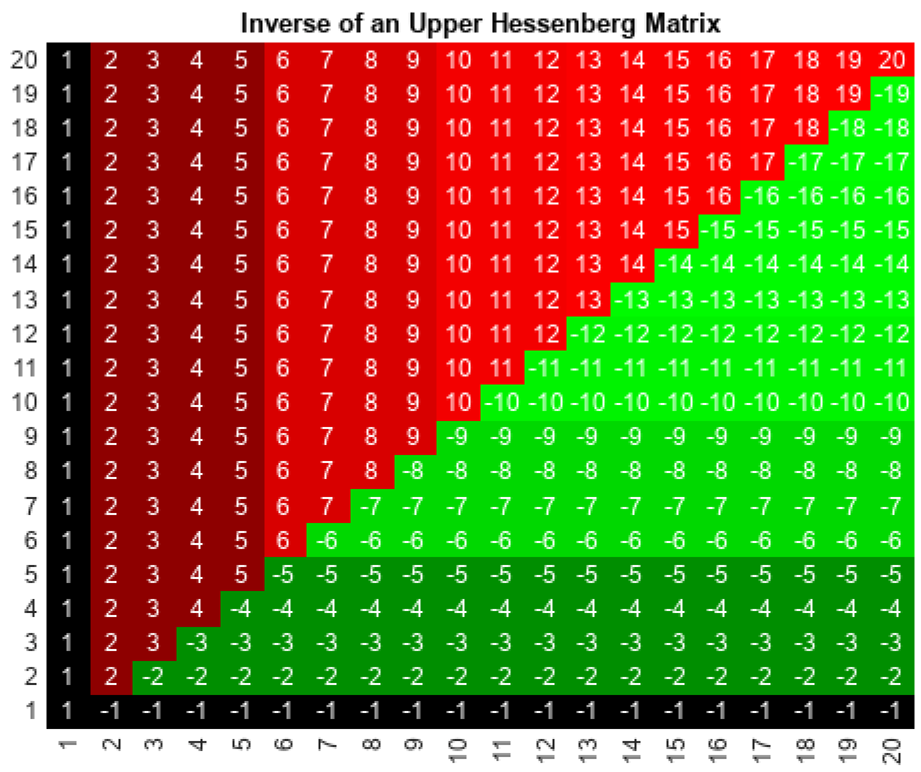
```
fH = figure;
hA = plot(hmo,fH);
```



1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	-19
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	-18	-18
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	-17	-17	-17
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	-16	-16	-16	-16
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	-15	-15	-15	-15	-15
1	2	3	4	5	6	7	8	9	10	11	12	13	14	-14	-14	-14	-14	-14	-14
1	2	3	4	5	6	7	8	9	10	11	12	13	-13	-13	-13	-13	-13	-13	-13
1	2	3	4	5	6	7	8	9	10	11	12	-12	-12	-12	-12	-12	-12	-12	-12
1	2	3	4	5	6	7	8	9	10	11	-11	-11	-11	-11	-11	-11	-11	-11	-11
1	2	3	4	5	6	7	8	9	10	-10	-10	-10	-10	-10	-10	-10	-10	-10	-10
1	2	3	4	5	6	7	8	9	-9	-9	-9	-9	-9	-9	-9	-9	-9	-9	-9
1	2	3	4	5	6	7	8	-8	-8	-8	-8	-8	-8	-8	-8	-8	-8	-8	-8
1	2	3	4	5	6	7	-7	-7	-7	-7	-7	-7	-7	-7	-7	-7	-7	-7	-7
1	2	3	4	5	6	-6	-6	-6	-6	-6	-6	-6	-6	-6	-6	-6	-6	-6	-6
1	2	3	4	5	-5	-5	-5	-5	-5	-5	-5	-5	-5	-5	-5	-5	-5	-5	-5
1	2	3	4	-4	-4	-4	-4	-4	-4	-4	-4	-4	-4	-4	-4	-4	-4	-4	-4
1	2	3	-3	-3	-3	-3	-3	-3	-3	-3	-3	-3	-3	-3	-3	-3	-3	-3	-3
1	2	-2	-2	-2	-2	-2	-2	-2	-2	-2	-2	-2	-2	-2	-2	-2	-2	-2	-2
1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1

Use the returned axes handle hA to specify the axes properties.

```
hA.Title.String = 'Inverse of an Upper Hessenberg Matrix';
hA.XTickLabelMode = 'auto';
hA.YTickLabelMode = 'auto';
```



Input Arguments

`hm_cg_object` — Heatmap or clustergram object
`HeatMap` object | `clustergram` object

Heatmap or clustergram object, specified as a `HeatMap` object or `clustergram` object.

`hFig` — Handle to figure
figure handle

Handle to a figure to display the heatmap or clustergram, specified as a figure handle.

Output Arguments

`hAxes` — Axes of heatmap or clustergram figure
`axes` object

Axes of the heatmap or clustergram figure, returned as an `axes` object.

Version History

Introduced in R2009b

See Also

`HeatMap` | Axes | `clustergram`

plotChiSquaredFit

Plot goodness-of-fit for variance regression

Syntax

```
plotChiSquaredFit(test)
plotChiSquaredFit(test,Name,Value)
H = plotChiSquaredFit( ___ )
```

Description

`plotChiSquaredFit(test)` plots the empirical CDF of the chi-squared probabilities of the ratio between the observed and the estimated variance stratified by count levels into five equal-sized bins. Use this plot to assess the goodness-of-fit.

`test`, an output of the `nbintest` function, is a `NegativeBinomialTest` object. It contains results from an unpaired hypothesis test for two independent samples.

Note If the `'VarianceLink'` name-value pair argument was set to `'Identity'` when you ran `nbintest`, then the chi-squared probability is computed using the ratio between the observed variance to the mean.

`plotChiSquaredFit(test,Name,Value)` uses a name-value pair argument.

`H = plotChiSquaredFit( ___ )` returns handles to axes.

Examples

Perform unpaired hypothesis test for short-read count data

This example shows how to perform an unpaired hypothesis test for synthetic short-read count data from two different biological conditions.

The data in this example contains synthetic gene count data for 5000 genes, representing two different biological conditions, such as diseased and normal cells. For each condition, there are five samples. Only 10% of the genes (500 genes) are differentially expressed. Specifically, half of them (250 genes) are exactly 3-fold overexpressed. The other 250 genes are 3-fold underexpressed. The rest of the gene expression data is generated from the same negative binomial distribution for both conditions. Each sample also has a different size factor (that is, the coverage or sampling depth).

Load the data.

```
load('nbintest_data.mat','K','H0');
```

The variable `K` contains gene count data. The rows represent genes, and the columns represent samples. In this case, the first five columns represent samples from the first condition. The other five columns represent samples from the second condition. Display the first few rows of `K`.

```
K(1:5,:)
```

```
ans = 5×10

       13683        14140         8281        14309        12208         8045         9446        11317
       16028        16805         9813        16486        14076         9901        10927        13348
         814          862          492          910          758          521          573          753
       15870        16453         9857        16454        14267         9671        10997        13624
        9422         9393         5734         9598         8174         5381         6315         7752
```

In this example, the null hypothesis is true when the gene is not differentially expressed. The variable `H0` contains boolean indicators that indicate for which genes the null hypothesis is true (marked as 1). In other words, H0 contains known labels that you will use later to compare with predicted results.

```
sum(H0)
```

```
ans = 4500
```

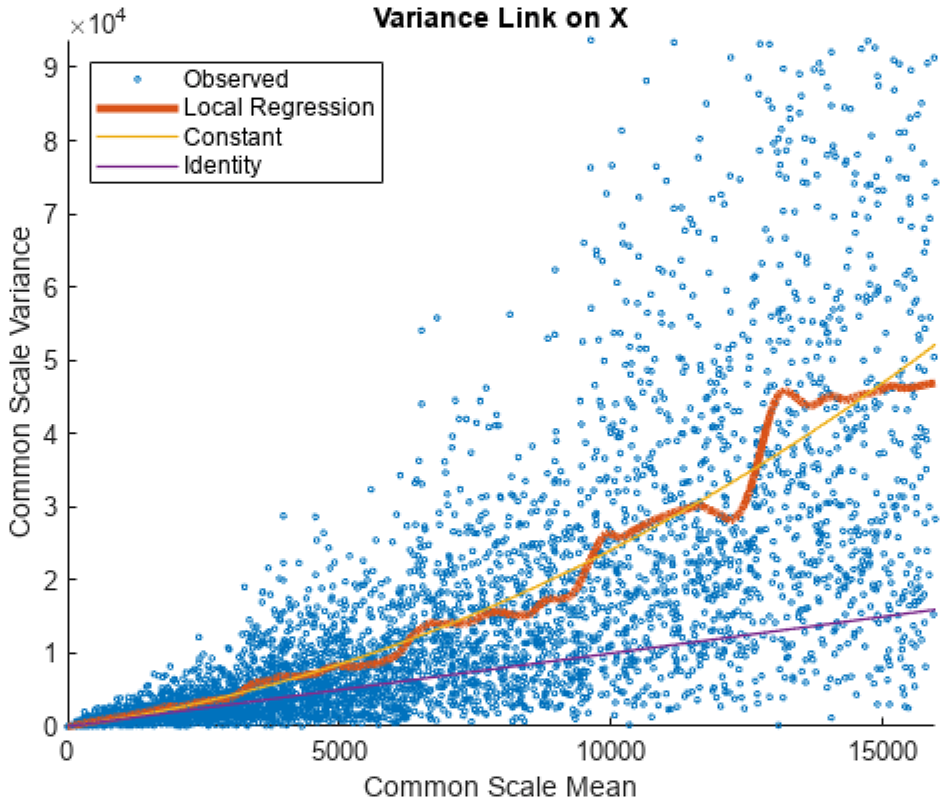
Out of 5000 genes, 4500 are not differentially expressed in this synthetic data.

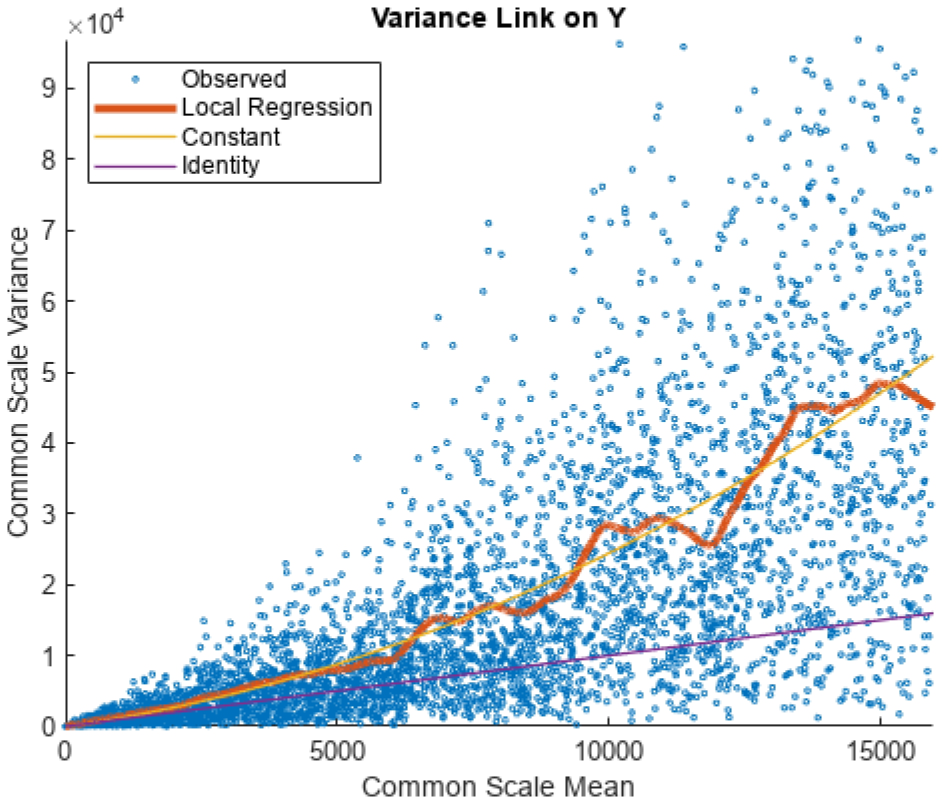
Run an unpaired hypothesis test for samples from two conditions using `nbintest`. The assumption is that the data came from a negative binomial distribution, where the variance is linked to the mean via a locally-regressed smooth function of the mean as described in [1] by setting `'VarianceLink'` to `'LocalRegression'`.

```
tLocal = nbintest(K(:,1:5),K(:,6:10),'VarianceLink','LocalRegression');
```

Use `plotVarianceLink` to plot a scatter plot for each experimental condition (for X and Y conditions), with the sample variance on the common scale versus the estimate of the condition-dependent mean. Use a linear scale for both axes. Include curves for all other linkage options by setting `'Compare'` to `true`.

```
plotVarianceLink(tLocal,'Scale','linear','Compare',true)
```

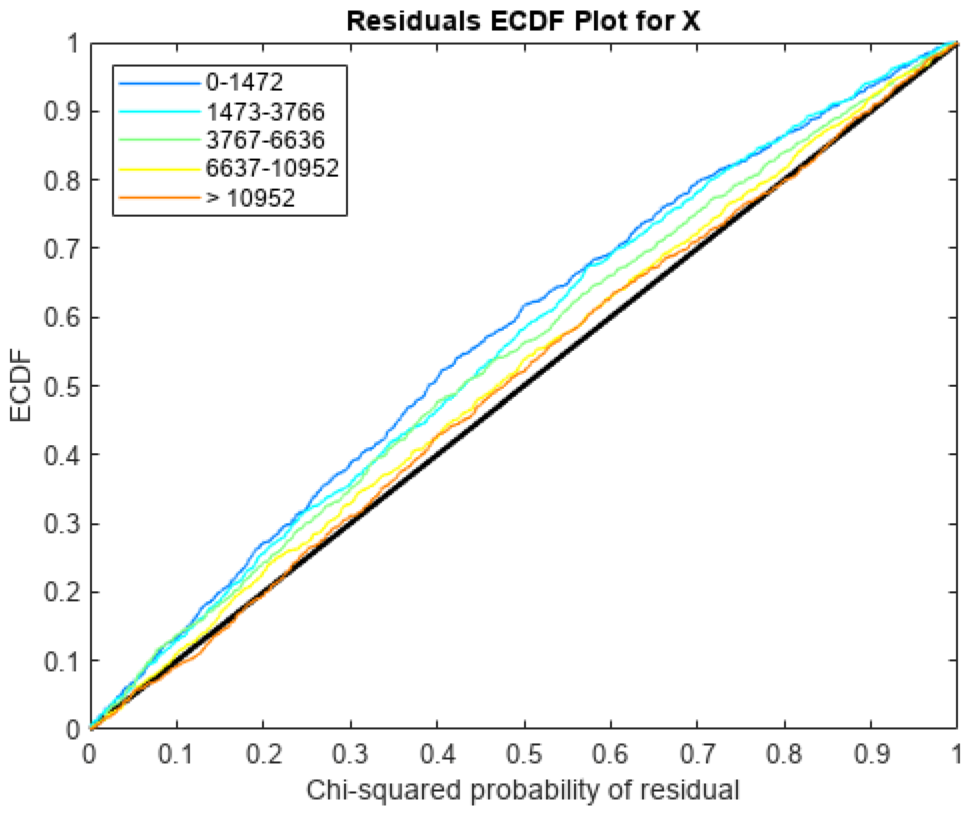

Variance Link on X
×10^4
Observed
Local Regression
Constant
Identity
Common Scale Variance
Common Scale Mean
0
1
2
3
4
5
6
7
8
9
0
5000
10000
15000

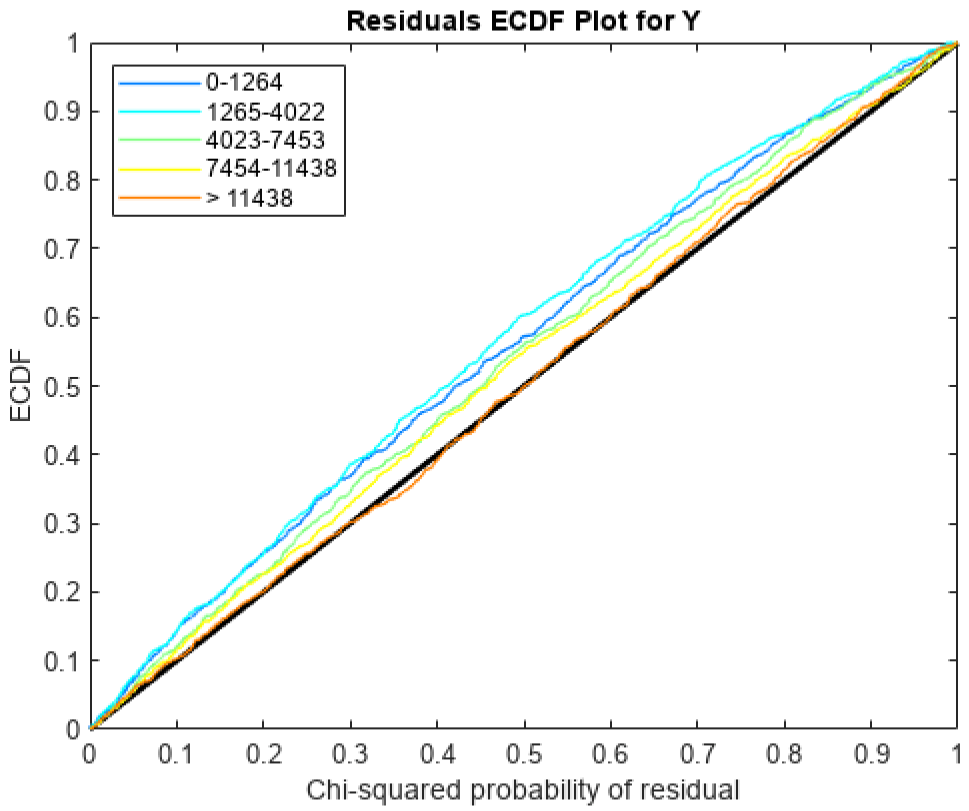


The `Identity` line represents the Poisson model, where the variance is identical to the mean as described in [3]. Observe that the data seems to be overdispersed (that is, most points are above the `Identity` line). The `Constant` line represents the negative binomial model, where the variance is the sum of the shot noise term (mean) and a constant multiplied by the squared mean as described in [2]. The `Local Regression` and `Constant` linkage options appear to fit better with the overdispersed data.

Use `plotChiSquaredFit` to assess the goodness-of-fit for variance regression. It plots the empirical CDF (ecdf) of the chi-squared probabilities. The probabilities are the ratio between the observed and the estimated variance stratified by short-read count levels into five equal-sized bins.

```
plotChiSquaredFit(tLocal)
```


Residuals ECDF Plot for X
0-1472
1473-3766
3767-6636
6637-10952
> 10952
ECDF
Chi-squared probability of residual
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1



Each figure shows five ecdf curves. Each curve represents one of the five short-read count levels. For instance, the blue line represents the ecdf curve for a low short-read counts between 0 and 1264. The red line represents high counts (more than 11438).

One way to interpret the curves is to check if the ecdf curves are above the diagonal line. If they are above the line, then the variance is overestimated. If they are below the line, then the variance is underestimated. In both figures, the variance seems to be correctly estimated for higher counts (that is, the red line follows the diagonal line), but slightly overestimated for lower count levels.

To assess the performance of the hypothesis test, construct a confusion matrix using the known labels and the predicted p-values.

```
confusionmat(H0,(tLocal.pValue > .001))
```

```
ans = 2×2

         493           7
           5        4495
```

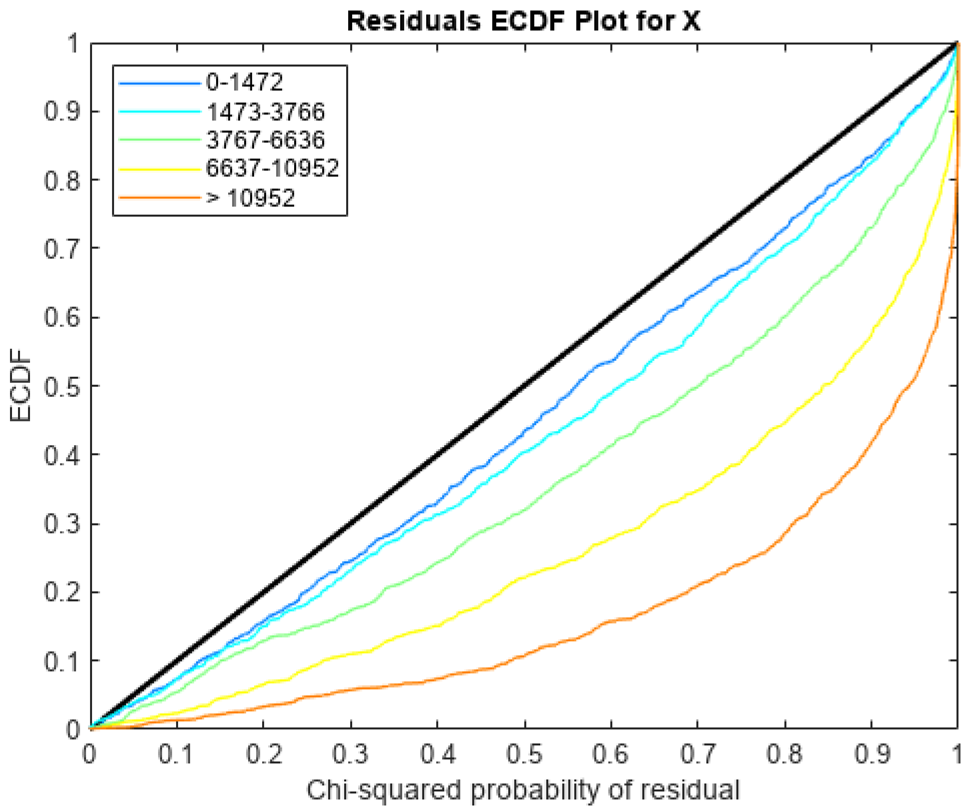
Out of 500 differentially expressed genes, 493 are correctly predicted (true positives) and 7 of them are incorrectly predicted as not-differentially expressed genes (false negatives). Out of 4500 genes that are not differentially expressed, 4495 are correctly predicted (true negatives) and 5 of them are incorrectly predicted as differentially expressed genes (false positives).

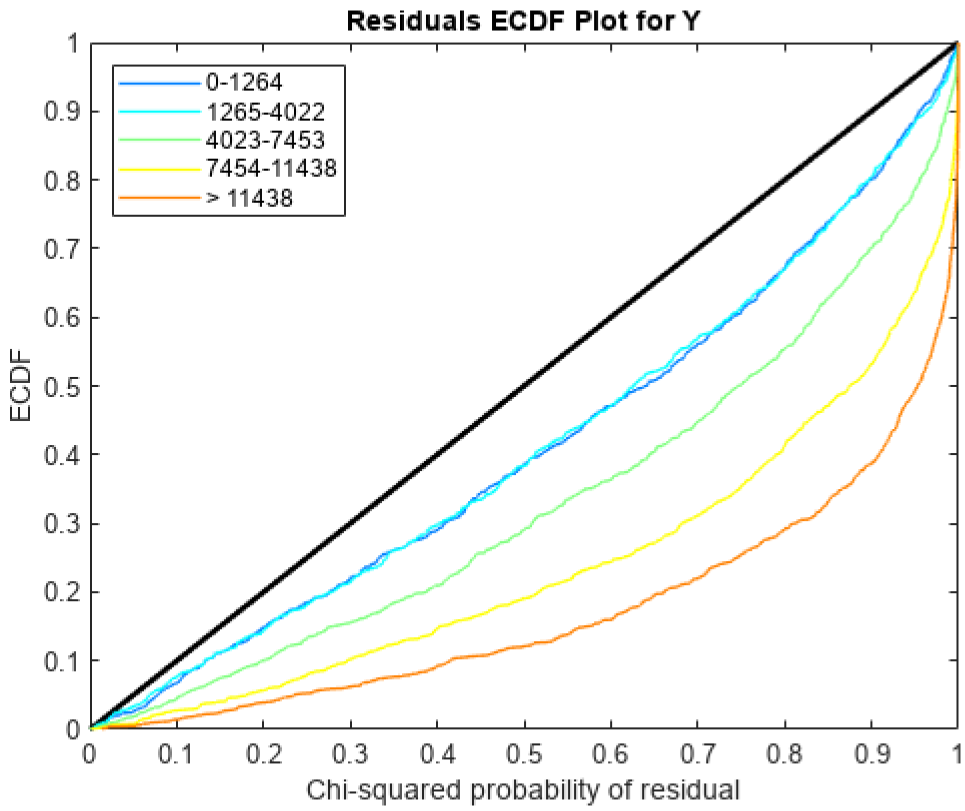
For a comparison, run the hypothesis test again assuming that counts are modeled by the Poisson distribution, where the variance is identical to the mean.

```
tPoisson = nbintest(K(:,1:5),K(:,6:10),'VarianceLink','Identity');
```

Plot the ecdf curves. Observe that all the curves are below the diagonal line, implying that the variance is underestimated. Therefore, the negative binomial model fits the data better.

```
plotChiSquaredFit(tPoisson)
```





Input Arguments

`test` — Unpaired hypothesis test result
`NegativeBinomialTest` object (default)

Unpaired hypothesis test results, specified as a `NegativeBinomialTest` object. test is returned by the `nbintest` function.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `'NumBins',4`

`NumBins` — Number of equal-sized bins
`5` (default) | positive integer

Number of equal-sized bins, specified as a comma-separated pair consisting of `'NumBins'` and a positive integer.

Example: `'NumBins',3`

Output Arguments

`H` — Handles to axes
vector of handles

Handles to axes, specified as a vector of handles.

Version History

Introduced in R2014b

See Also

`nbintest` | `NegativeBinomialTest` | `plotVarianceLink` | `mattest`

plot (phytree)

Draw phylogenetic tree

Syntax

```
plot(Tree)
plot(Tree, ActiveBranches)
H = plot(...)
plot(..., 'Type', TypeValue, ...)
plot(..., 'Orientation', OrientationValue, ...)
plot(..., 'Rotation', RotationValue, ...)
plot(..., 'BranchLabels', BranchLabelsValue, ...)
plot(..., 'LeafLabels', LeafLabelsValue, ...)
plot(..., 'TerminalLabels', TerminalLabelsValue, ...)
plot(..., 'LLRotation', LLRotationValue, ...)
```

Input Arguments

Tree	Phylogenetic tree object created, such as created with the `phytree` constructor function.
ActiveBranches	Logical array of size `numBranches`-by-`1` indicating the active branches, which are displayed in the Figure window.
TypeValue	Character vector or string specifying a method for drawing the phylogenetic tree. Choices are: • `'square'` (default) • `'angular'` • `'radial'` • `'equalangle'` • `'equaldaylight'`
OrientationValue	Character vector or string specifying the position of the root node, and hence the orientation of a phylogram or cladogram tree, when the `'Type'` property is `'square'` or `'angular'`. Choices are: • `'left'` (default) • `'right'` • `'top'` • `'bottom'`
RotationValue	Scalar between `0` (default) and `360` specifying rotation angle (in degrees) of the phylogenetic tree in the Figure window, when the `'Type'` property is `'radial'`, `'equalangle'`, or `'equaldaylight'`.
BranchLabelsValue	Controls the display of branch labels next to branch nodes. Choices are `true` or `false` (default).

`LeafLabelsValue`	Controls the display of leaf labels next to leaf nodes. Choices are `true` or `false`. Default is: • `true` — When the `'Type'` property is `'radial'`, `'equalangle'`, or `'equaldaylight'` • `false` — When the `'Type'` property is `'square'` or `'angular'`
`TerminalLabels`	Controls the display of terminal labels over the axis tick labels, when the `'Type'` property is `'square'` or `'angular'`. Choices are `true` (default) or `false`.
`LLRotationValue`	Controls the rotation of leaf labels so that the text aligns to the root node, when the `'Type'` property is `'radial'`, `'equalangle'`, or `'equaldaylight'`. Choices are `true` or `false` (default).

Output Arguments

`H`	Structure with handles to seven graph elements. The structure includes the following fields: • `axes` • `BranchLines` • `BranchDots` • `LeafDots` • `branchNodeLabels` • `leafNodeLabels` • `terminalNodeLabels` **Tip** Use the `set` function with the handles in this structure and their related properties to modify the plot. For more information on the properties you can modify using the `axes` handle, see Axes. For more information on the properties you can modify using the `BranchLines`, `BranchDots`, or `LeafDots` handle, see Primitive Line. For more information on the properties you can modify using the `branchNodeLabels`, `leafNodeLabels`, or `terminalNodeLabels` handle, see Text.

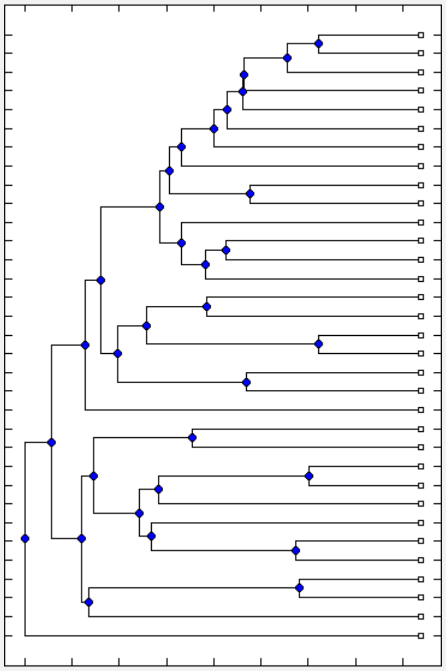
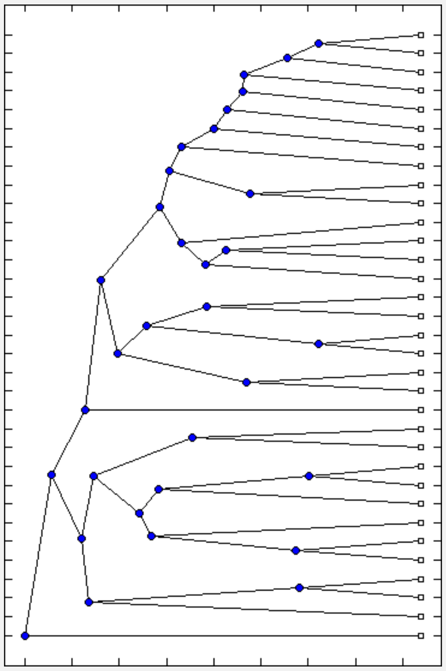
Description

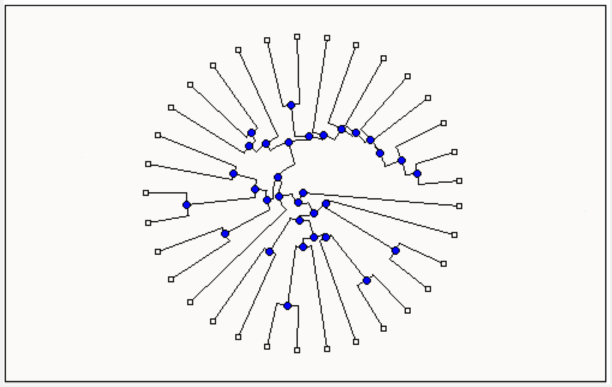
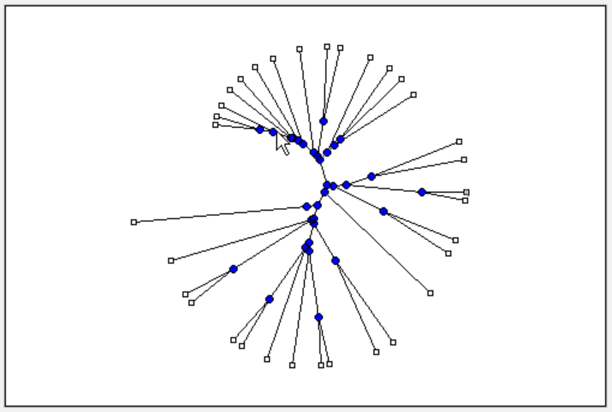
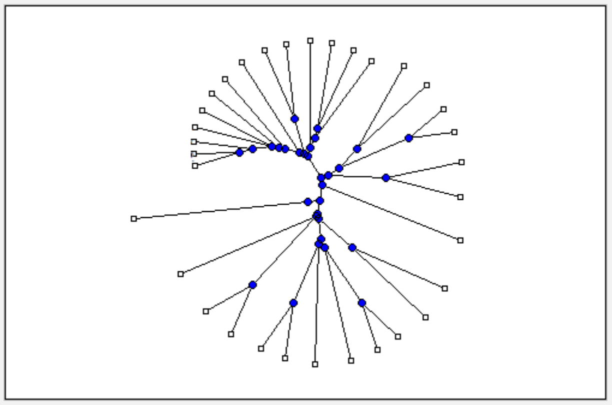
`plot(`*`Tree`*`)` draws a phylogenetic tree object into a figure as a phylogram. The significant distances between branches and nodes are in the horizontal direction. Vertical distances are arbitrary and have no significance.

`plot(`*`Tree`*`, `*`ActiveBranches`*`)` hides the nonactive branches and all of their descendants in the Figure window. *`ActiveBranches`* is a logical array of size `numBranches`-by-`1` indicating the active branches.

`H = plot(...)` returns a structure with handles to seven graph elements.

`plot(..., 'Type', TypeValue, ...)` specifies a method for rendering the phylogenetic tree. Choices are as follows.

Rendering Type	**Description**
`'square'` (default)	
`'angular'`	

Rendering Type	Description
`'radial'`	
`'equalangle'`	 **Tip** This rendering type hides the significance of the root node and emphasizes clusters, thereby making it useful for visually assessing clusters and detecting outliers.
`'equaldaylight'`	 **Tip** This rendering type hides the significance of the root node and emphasizes clusters, thereby making it useful for visually assessing clusters and detecting outliers.

`plot(..., 'Orientation', *OrientationValue*, ...)` specifies the orientation of the root node, and hence the orientation of a phylogram or cladogram phylogenetic tree in the Figure window, when the `'Type'` property is `'square'` or `'angular'`.

`plot(..., 'Rotation', *RotationValue*, ...)` specifies the rotation angle (in degrees) of the phylogenetic tree in the Figure window, when the `'Type'` property is `'radial'`, `'equalangle'`, or `'equaldaylight'`. Choices are any scalar between `0` (default) and `360`.

`plot(..., 'BranchLabels', *BranchLabelsValue*, ...)` hides or displays branch labels next to the branch nodes. Choices are `true` or `false` (default).

`plot(..., 'LeafLabels', *LeafLabelsValue*, ...)` hides or displays leaf labels next to the leaf nodes. Choices are `true` or `false`. Default is:

- `true` — When the `'Type'` property is `'radial'`, `'equalangle'`, or `'equaldaylight'`
- `false` — When the `'Type'` property is `'square'` or `'angular'`

`plot(..., 'TerminalLabels', *TerminalLabelsValue*, ...)` hides or displays terminal labels over the axis tick labels, when the `'Type'` property is `'square'` or `'angular'`. Choices are `true` (default) or `false`.

`plot(..., 'LLRotation', *LLRotationValue*, ...)` controls the rotation of leaf labels so that the text aligns to the root node, when the `'Type'` property is `'radial'`, `'equalangle'`, or `'equaldaylight'`. Choices are `true` or `false` (default).

Examples

```
% Create a phytree object from a file
tr = phytreeread('pf00002.tree')
% Plot the tree and return a structure with handles to the
% graphic elements of the phytree object
h = plot(tr,'Type','radial')

% Modify the font size and color of the leaf node labels
% by using one of the handles in the return structure
set(h.leafNodeLabels,'FontSize',6,'Color',[1 0 0])
```

Version History
Introduced before R2006a

See Also
`phytree` | `phytreeread` | `phytreeviewer` | `seqlinkage` | `seqneighjoin` | `cluster` | `view`

Topics
phytree object on page 1-1449

plotVarianceLink

Plot the sample variance versus the estimate of the condition-dependent mean

Syntax

```
plotVarianceLink(test)
plotVarianceLink(test,Name,Value)
H = plotVarianceLink( ___ )
```

Description

`plotVarianceLink(test)` displays one scatter plot for each experimental condition with the sample variance on the common scale versus the estimate of the condition-dependent mean.

`test`, an output of the `nbintest` function, is a `NegativeBinomialTest` object, containing results from an unpaired hypothesis test for two independent samples.

If the `'PooledVariance'` name-value pair argument was set to `true` when you ran `nbintest`, then `plotVarianceLink` plots only one scatter plot. The function also plots the variance regression according to the model specified by the `'VarianceLink'` name-value pair argument of `nbintest`.

`plotVarianceLink(test,Name,Value)` uses one or more name-value pair arguments.

`H = plotVarianceLink( ___ )` returns handles to axes.

Examples

Perform unpaired hypothesis test for short-read count data

This example shows how to perform an unpaired hypothesis test for synthetic short-read count data from two different biological conditions.

The data in this example contains synthetic gene count data for 5000 genes, representing two different biological conditions, such as diseased and normal cells. For each condition, there are five samples. Only 10% of the genes (500 genes) are differentially expressed. Specifically, half of them (250 genes) are exactly 3-fold overexpressed. The other 250 genes are 3-fold underexpressed. The rest of the gene expression data is generated from the same negative binomial distribution for both conditions. Each sample also has a different size factor (that is, the coverage or sampling depth).

Load the data.

```
load('nbintest_data.mat','K','H0');
```

The variable `K` contains gene count data. The rows represent genes, and the columns represent samples. In this case, the first five columns represent samples from the first condition. The other five columns represent samples from the second condition. Display the first few rows of `K`.

```
K(1:5,:)
```

```
ans = 5×10
```

```
        13683        14140         8281        14309        12208         8045         9446        11317
        16028        16805         9813        16486        14076         9901        10927        13348
          814          862          492          910          758          521          573          753
        15870        16453         9857        16454        14267         9671        10997        13624
         9422         9393         5734         9598         8174         5381         6315         7752
```

In this example, the null hypothesis is true when the gene is not differentially expressed. The variable `H0` contains boolean indicators that indicate for which genes the null hypothesis is true (marked as 1). In other words, H0 contains known labels that you will use later to compare with predicted results.

```
sum(H0)
```

```
ans = 4500
```

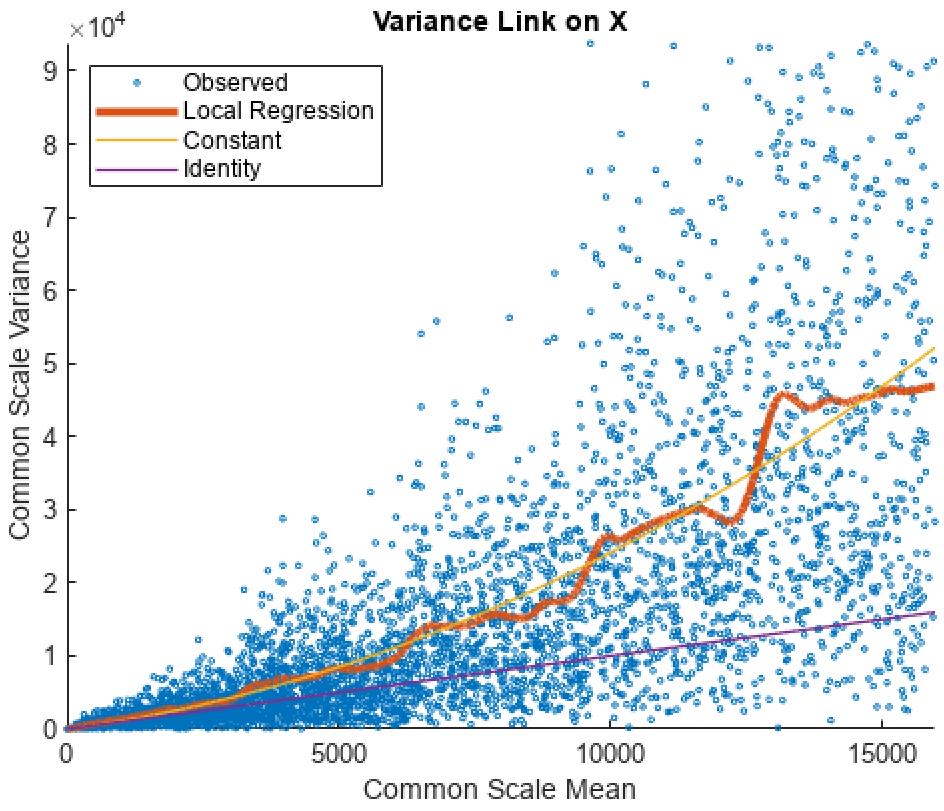
Out of 5000 genes, 4500 are not differentially expressed in this synthetic data.

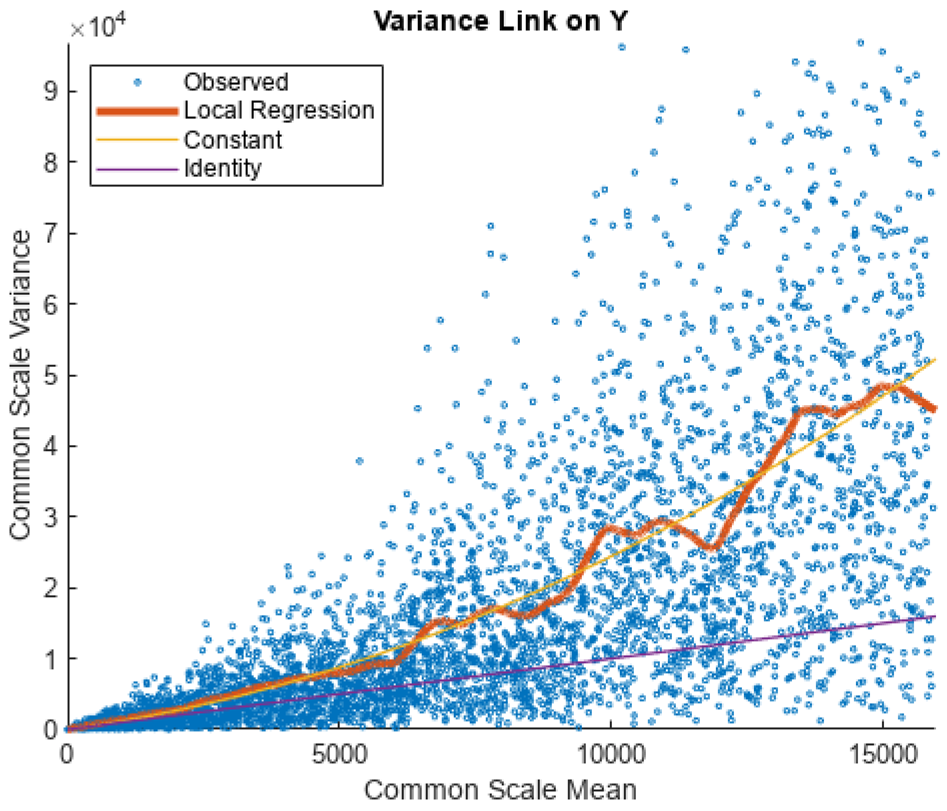
Run an unpaired hypothesis test for samples from two conditions using `nbintest`. The assumption is that the data came from a negative binomial distribution, where the variance is linked to the mean via a locally-regressed smooth function of the mean as described in [1] by setting `'VarianceLink'` to `'LocalRegression'`.

```
tLocal = nbintest(K(:,1:5),K(:,6:10),'VarianceLink','LocalRegression');
```

Use `plotVarianceLink` to plot a scatter plot for each experimental condition (for X and Y conditions), with the sample variance on the common scale versus the estimate of the condition-dependent mean. Use a linear scale for both axes. Include curves for all other linkage options by setting `'Compare'` to `true`.

```
plotVarianceLink(tLocal,'Scale','linear','Compare',true)
```

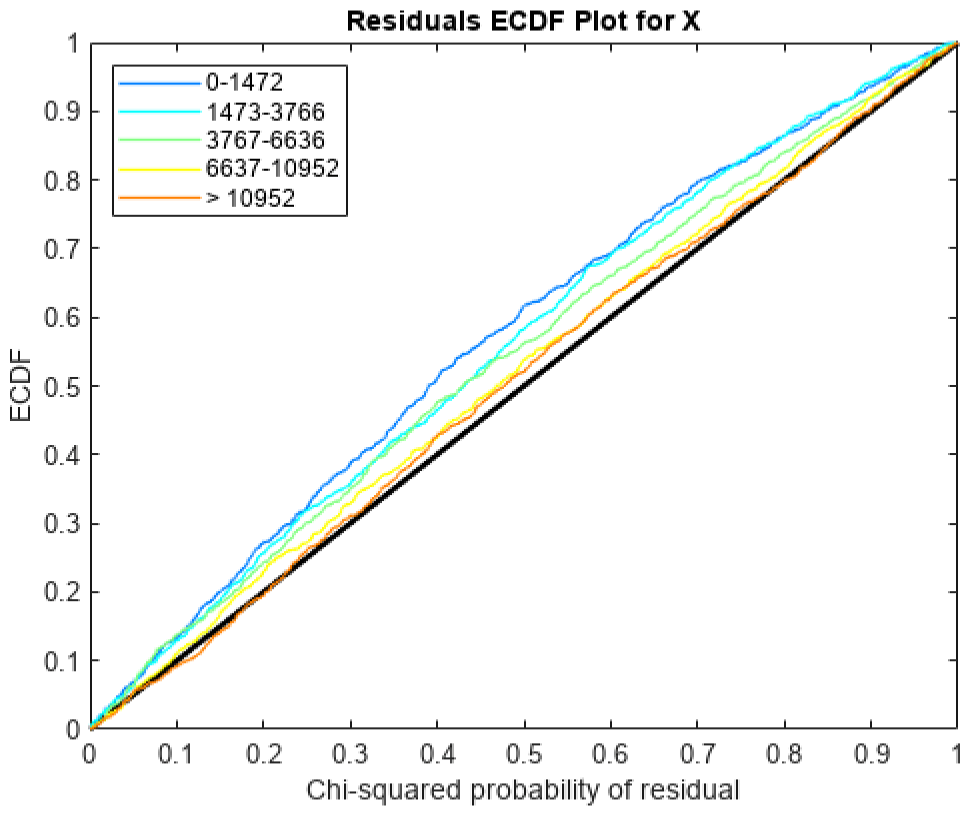

Variance Link on X
×10^4
9
8
7
6
5
4
3
2
1
0
Common Scale Variance
0
5000
10000
15000
Common Scale Mean
Observed
Local Regression
Constant
Identity

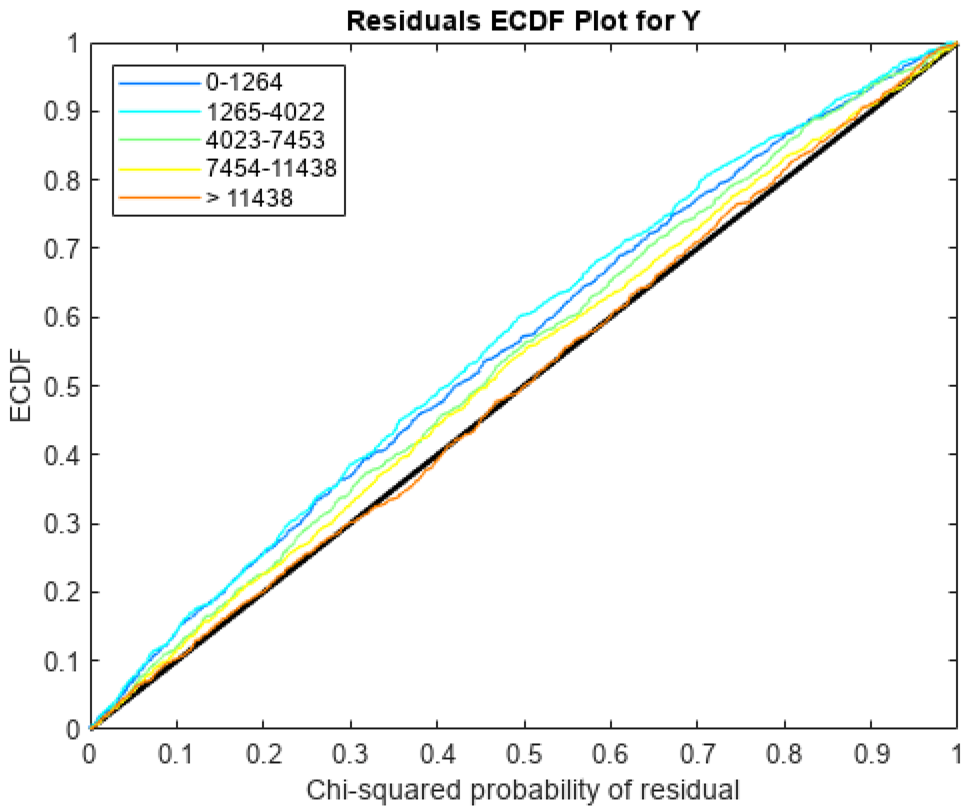


The `Identity` line represents the Poisson model, where the variance is identical to the mean as described in [3]. Observe that the data seems to be overdispersed (that is, most points are above the `Identity` line). The `Constant` line represents the negative binomial model, where the variance is the sum of the shot noise term (mean) and a constant multiplied by the squared mean as described in [2]. The `Local Regression` and `Constant` linkage options appear to fit better with the overdispersed data.

Use `plotChiSquaredFit` to assess the goodness-of-fit for variance regression. It plots the empirical CDF (ecdf) of the chi-squared probabilities. The probabilities are the ratio between the observed and the estimated variance stratified by short-read count levels into five equal-sized bins.

```
plotChiSquaredFit(tLocal)
```


Residuals ECDF Plot for X
0-1472
1473-3766
3767-6636
6637-10952
> 10952
ECDF
Chi-squared probability of residual
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1



Each figure shows five ecdf curves. Each curve represents one of the five short-read count levels. For instance, the blue line represents the ecdf curve for a low short-read counts between 0 and 1264. The red line represents high counts (more than 11438).

One way to interpret the curves is to check if the ecdf curves are above the diagonal line. If they are above the line, then the variance is overestimated. If they are below the line, then the variance is underestimated. In both figures, the variance seems to be correctly estimated for higher counts (that is, the red line follows the diagonal line), but slightly overestimated for lower count levels.

To assess the performance of the hypothesis test, construct a confusion matrix using the known labels and the predicted p-values.

```
confusionmat(H0,(tLocal.pValue > .001))
```

```
ans = 2×2

         493           7
           5        4495
```

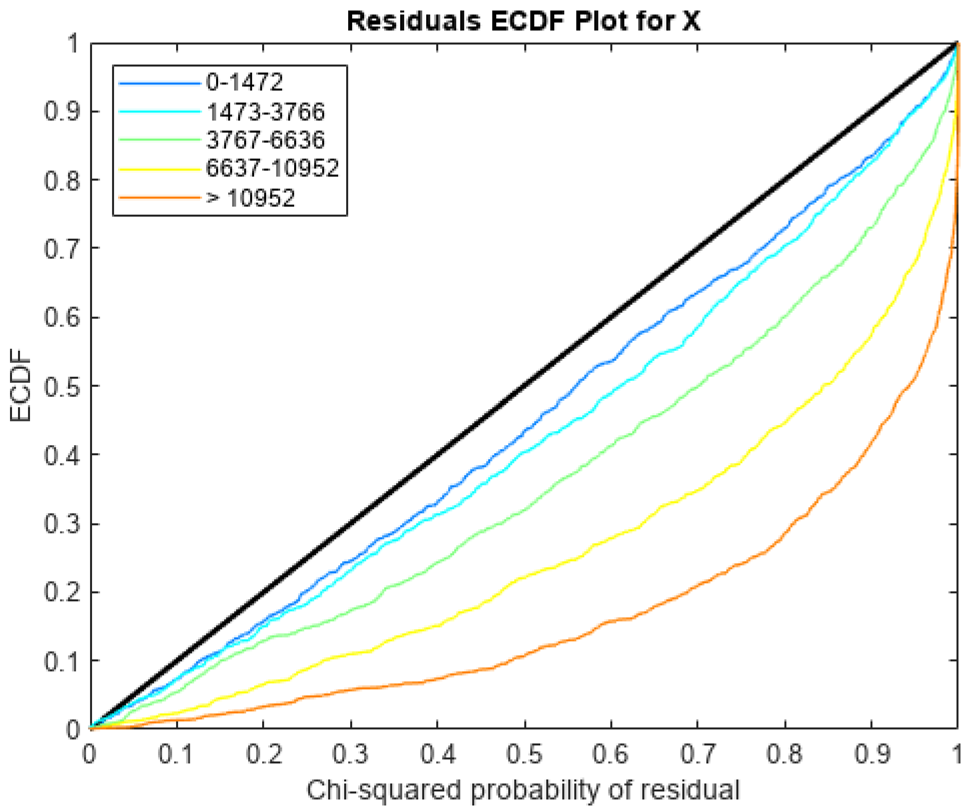
Out of 500 differentially expressed genes, 493 are correctly predicted (true positives) and 7 of them are incorrectly predicted as not-differentially expressed genes (false negatives). Out of 4500 genes that are not differentially expressed, 4495 are correctly predicted (true negatives) and 5 of them are incorrectly predicted as differentially expressed genes (false positives).

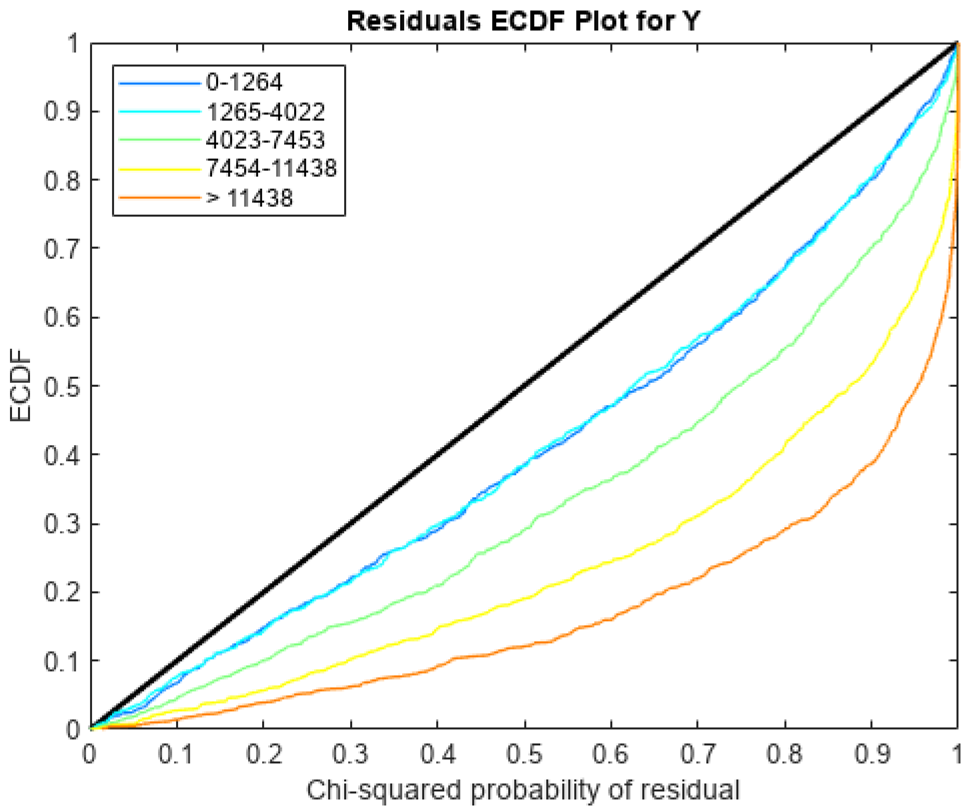
For a comparison, run the hypothesis test again assuming that counts are modeled by the Poisson distribution, where the variance is identical to the mean.

```
tPoisson = nbintest(K(:,1:5),K(:,6:10),'VarianceLink','Identity');
```

Plot the ecdf curves. Observe that all the curves are below the diagonal line, implying that the variance is underestimated. Therefore, the negative binomial model fits the data better.

```
plotChiSquaredFit(tPoisson)
```





Input Arguments

`test` — Unpaired hypothesis test result
`NegativeBinomialTest` object (default)

Unpaired hypothesis test results, specified as a `NegativeBinomialTest` object. test is returned by the `nbintest` function.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `'Compare',true,'Scale','linear'`

`Compare` — Logical flag to add a curve with the variance link for other models
`false` (default) | `true`

Logical flag to add a curve with the variance link for other models, specified as a comma-separated pair consisting of `'Compare'` and `true` or `false`. When it is set to `true`, the plot shows curves for all the available linkage options, that is, `'LocalRegression'`, `'Constant'`, and `'Identity'`.

Example: `'Compare',true`

`Scale` — Scale for both axes
`'log'` (default) | `'linear'`

Scale for both axes, specified as a comma-separated pair consisting of `'Scale'` and `'log'` or `'linear'`.

Example: `'Scale','linear'`

Output Arguments

`H` — Handles to axes
vector of handles

Handles to axes, specified as a vector of handles.

Version History

Introduced in R2014b

See Also

`nbintest` | `NegativeBinomialTest` | `plotChiSquaredFit` | `mattest`

plus (DataMatrix)

Add DataMatrix objects

Syntax

```
DMObjNew = plus(DMObj1, DMObj2)
DMObjNew = DMObj1 + DMObj2
DMObjNew = plus(DMObj1, B)
DMObjNew = DMObj1 + B
DMObjNew = plus(B, DMObj1)
DMObjNew = B + DMObj1
```

Input Arguments

DMObj1, *DMObj2*	DataMatrix objects, such as created by `DataMatrix` (object constructor).
B	MATLAB numeric or logical array.

Output Arguments

DMObjNew	DataMatrix object created by addition.

Description

`DMObjNew = plus(DMObj1, DMObj2)` or the equivalent `DMObjNew = DMObj1 + DMObj2` performs an element-by-element addition of the DataMatrix objects *DMObj1* and *DMObj2* and places the results in *DMObjNew*, another DataMatrix object. *DMObj1* and *DMObj2* must have the same size (number of rows and columns), unless one is a scalar (1-by-1 DataMatrix object). The size (number of rows and columns), row names, and column names for *DMObjNew* are the same as *DMObj1*, unless *DMObj1* is a scalar; then they are the same as *DMObj2*.

`DMObjNew = plus(DMObj1, B)` or the equivalent `DMObjNew = DMObj1 + B` performs an element-by-element addition of *DMObj1*, a DataMatrix object, and *B*, a numeric or logical array, and places the results in *DMObjNew*, another DataMatrix object. *DMObj1* and *B* must have the same size (number of rows and columns), unless *B* is a scalar. The size (number of rows and columns), row names, and column names for *DMObjNew* are the same as *DMObj1*.

`DMObjNew = plus(B, DMObj1)` or the equivalent `DMObjNew = B + DMObj1` performs an element-by-element addition of *B*, a numeric or logical array, and *DMObj1*, a DataMatrix object, and places the results in *DMObjNew*, another DataMatrix object. *DMObj1* and *B* must have the same size (number of rows and columns), unless *B* is a scalar. The size (number of rows and columns), row names, and column names for *DMObjNew* are the same as *DMObj1*.

Note Arithmetic operations between a scalar DataMatrix object and a nonscalar array are not supported.

MATLAB calls `DMObjNew = plus(X, Y)` for the syntax `DMObjNew = X + Y` when *X* or *Y* is a DataMatrix object.

Version History

Introduced in R2008b

See Also

`DataMatrix` | `minus`

Topics

DataMatrix object on page 1-734

power (DataMatrix)

Array power DataMatrix objects

Syntax

```
DMObjNew = power(DMObj1, DMObj2)
DMObjNew = DMObj1 .^ DMObj2
DMObjNew = power(DMObj1, B)
DMObjNew = DMObj1 .^ B
DMObjNew = power(B, DMObj1)
DMObjNew = B .^ DMObj1
```

Input Arguments

DMObj1, *DMObj2*	DataMatrix objects, such as created by `DataMatrix` (object constructor).
B	MATLAB numeric or logical array.

Output Arguments

DMObjNew	DataMatrix object created by array power.

Description

`DMObjNew = power(DMObj1, DMObj2)` or the equivalent `DMObjNew = DMObj1 .^ DMObj2` performs an element-by-element power of the DataMatrix objects *DMObj1* and *DMObj2* and places the results in *DMObjNew*, another DataMatrix object. In other words, `power` raises each element in *DMObj1* by the corresponding element in *DMObj2*. *DMObj1* and *DMObj2* must have the same size (number of rows and columns), unless one is a scalar (1-by-1 DataMatrix object). The size (number of rows and columns), row names, and column names for *DMObjNew* are the same as *DMObj1*, unless *DMObj1* is a scalar; then they are the same as *DMObj2*.

`DMObjNew = power(DMObj1, B)` or the equivalent `DMObjNew = DMObj1 .^ B` performs an element-by-element power of the DataMatrix object *DMObj1* and *B*, a numeric or logical array, and places the results in *DMObjNew*, another DataMatrix object. In other words, `power` raises each element in *DMObj1* by the corresponding element in *B*. *DMObj1* and *B* must have the same size (number of rows and columns), unless *B* is a scalar. The size (number of rows and columns), row names, and column names for *DMObjNew* are the same as *DMObj1*.

`DMObjNew = power(B, DMObj1)` or the equivalent `DMObjNew = B .^ DMObj1` performs an element-by-element power of *B*, a numeric or logical array, and the DataMatrix object *DMObj1*, and places the results in *DMObjNew*, another DataMatrix object. In other words, `power` raises each element in *B* by the corresponding element in *DMObj1*.*DMObj1* and *B* must have the same size (number of rows and columns), unless *B* is a scalar. The size (number of rows and columns), row names, and column names for *DMObjNew* are the same as *DMObj1*.

Note Arithmetic operations between a scalar DataMatrix object and a nonscalar array are not supported.

MATLAB calls `DMObjNew = power(X, Y)` for the syntax `DMObjNew = X .^ Y` when *X* or *Y* is a DataMatrix object.

Version History

Introduced in R2008b

See Also

`DataMatrix` | `times`

Topics

DataMatrix object on page 1-734

preset

Set combination of alignment options

Syntax

```
preset(object,P)
```

Description

`preset(object,P)` sets a combination of alignment options (object properties) to predefined values for typical tradeoffs between execution time and sensitivity. The function sets the following object properties: `NumReseedings`, `NumSeedExtensions`, `NumSeedMismatches`, `SeedIntervalFunction`, `SeedLength`, and `Mode`.

`preset` requires the Bowtie 2 Support Package for Bioinformatics Toolbox. If this support package is not installed, then the function provides a download link. For details, see “Bioinformatics Toolbox Software Support Packages”.

Examples

Map Reads to Reference Sequence Using Fast Alignment Options

Build a set of index files for the Drosophila genome. An error message appears if you do not have the Bioinformatics Toolbox Interface for Bowtie Aligner support package installed when you run the function. Click the provided link to download the package from the Add-on menu.

For this example, the reference sequence `Dmel_chr4.fa` is already provided with the toolbox.

```
status = bowtie2build('Dmel_chr4.fa', 'Dmel_chr4_index');
```

If the index build is successful, the function returns `0` and creates the index files (`*.bt2`) in the current folder. The files have the prefix `'Dmel_chr4_index'`.

Once the index is ready, map the read sequences to the reference. The paired-end read files (`SRR6008575_10k_1.fq` and `SRR6008575_10k_2.fq`) are already provided with the toolbox.

Create an options object.

```
 alignOpt = Bowtie2AlignOptions;
```

Some preset options define a combination of values for several alignment parameters at the same time. Use an empty string to see the list of all the preset options.

```
preset(alignOpt,'')
```

```
ans =

  8×6 table

                          NumSeedExtensions    NumReseedings    NumSeedMismatches    SeedLength
```

```
                                   _________________    _____________    _________________    __________

    Fast                                  10                  2                  0                22
    LocalFast                             10                  2                  0                22
    LocalSensitive                        15                  2                  0                20
    LocalVeryFast                          5                  1                  0                25
    LocalVerySensitive                    20                  3                  0                20
    Sensitive                             15                  2                  0                22
    VeryFast                               5                  1                  0                22
    VerySensitive                         20                  3                  0                20
```

Align using the `'Fast'` option, which makes the alignment process faster but less sensitive and less accurate.

```
preset(alignOpt,'Fast');
```

Map reads to the reference using the specified alignment options.

```
flag = run(alignOpt,'Dmel_chr4','SRR6008575_10k_1.fq','SRR6008575_10k_2.fq','SRR6008575_10k_chr4_
```

The output is a SAM-formatted file that contains the mapping results.

Input Arguments

`object` — Alignment options

`Bowtie2AlignOptions` object

Alignment options, specified as a `Bowtie2AlignOptions` object.

Example: `alignOpt`

`P` — Preset option

`'Sensitive'` object (default) | `'VerySensitive'` | `'Fast'` | `'VeryFast'` | `'LocalFast'` | `'LocalVeryFast'` | `'LocalSensitive'` | `'LocalVerySensitive'`

Preset option, specified as a character vector. Valid options are `'Sensitive'`, `'VerySensitive'`, `'Fast'`, `'VeryFast'`, `'LocalFast'`, `'LocalVeryFast'`, `'LocalSensitive'`, `'LocalVerySensitive'`, and `''`. Use an empty character vector `''` to display the predefined values for each preset option.

Example: `'LocalFast'`

Version History

Introduced in R2018a

References

[1] Langmead, B., and S. Salzberg. "Fast gapped-read alignment with Bowtie 2." *Nature Methods*. 9, 2012, 357–359.

See Also

`bowtie2` | `bowtie2inspect` | `bowtie2build` | `Bowtie2AlignOptions` | `Bowtie2BuildOptions` | `Bowtie2InspectOptions`

External Websites

Bowtie 2 manual

probelibraryinfo

Create table of probe set library information

Syntax

ProbeInfo = probelibraryinfo(*CELStruct*, *CDFStruct*)

Input Arguments

CELStruct	Structure created by the `affyread` function from an Affymetrix CEL file.
CDFStruct	Structure created by the `affyread` function from an Affymetrix CDF library file associated with the CEL file.

Output Arguments

ProbeInfo	Three-column matrix with the same number of rows as the `Probes` field of the *CELStruct*. • Column 1 — Probe set ID/name to which the probe belongs. (Probes that do not belong to a probe set in the CDF library file have probe set ID/name equal to `0`.) • Column 2 — Contains the probe pair number. • Column 3 — Indicates if the probe is a perfect match (`1`) or mismatch (`-1`) probe.

Description

ProbeInfo = probelibraryinfo(*CELStruct*, *CDFStruct*) creates a table of information linking the probe data from *CELStruct*, a structure created from an Affymetrix CEL file, with probe set information from *CDFStruct*, a structure created from an Affymetrix CDF file.

Note Affymetrix probe pair indexing is `0`-based, while MATLAB software indexing is `1`-based. The output from `probelibraryinfo` is `1`-based.

Examples

Retrieve Probe Set Library Information

This example shows how to extract probe set library information from Affymetrix® GeneChip® microarray data.

You need some sample data files from here. This example uses the sample data from the *E. coli* Antisense Genome Array. Extract the data files from the DTT archive using the Data Transfer Tool.

You also need to download the corresponding library file for the sample. For this example, Ecoli_ASv2.CDF is used as for the *E. coli* Antisense Genome Array. You may already have these files if you have any Affymetrix GeneChip software installed on your machine. If not, get the library files by downloading and unzipping the *E. coli* Antisense Genome Array zip file.

Read the contents of a CEL file into a MATLAB structure.

```
celStruct = affyread('Ecoli-antisense-121502.CEL');
```

Read the contents of a CDF file into a MATLAB structure.

```
cdfStruct = affyread('C:\LibFiles\Ecoli_ASv2.CDF');
```

Extract probe set library information.

```
probeInfo = probelibraryinfo(celStruct, cdfStruct);
```

Determine the probe set to which the 1104th probe belongs.

```
cdfStruct.ProbeSets(probeInfo(1104,1)).Name
```

```
ans =

    'thrA_b0002_at'
```

Version History

Introduced before R2006a

See Also

affyread | celintensityread | probesetlookup | probesetplot | probesetvalues

Topics

“Working with Affymetrix Data”

probesetlink

(Removed) Display probe set information on NetAffx Web site

Note `probesetlink` has been removed.

Syntax

```
probesetlink(AffyStruct, PS)
URL = probesetlink(AffyStruct, PS)

probesetlink(AffyStruct, PS, ...'Source', SourceValue, ...)
probesetlink(AffyStruct, PS, ...'Browser', BrowserValue, ...)
URL = probesetlink(AffyStruct, PS, ...'NoDisplay', NoDisplayValue, ...)
```

Input Arguments

AffyStruct	Structure created by the `affyread` function from an Affymetrix CHP file or an Affymetrix CDF library file.
PS	Probe set index or the probe set ID/name.
SourceValue	Controls the linking to the data source (for example, GenBank or Flybase) for the probe set (instead of linking to the NetAffx® Web site). Choices are `true` or `false` (default). **Note** This property requires the GIN library file associated with the CHP or CDF file to be located in the same folder as the CDF library file.
BrowserValue	Controls the display of the probe set information in your system's default Web browser. Choices are `true` or `false` (default).
NoDisplayValue	Controls the return of *URL* without opening a Web browser. Choices are `true` or `false` (default).

Output Arguments

URL	URL for the probe set information.

Description

`probesetlink(`*`AffyStruct`*`, `*`PS`*`)` opens a Web Browser window displaying information on the NetAffx Web site about a probe set specified by *PS*, a probe set index or the probe set ID/name, and *AffyStruct*, a structure created from an Affymetrix CHP file or Affymetrix CDF library file.

URL `= probesetlink(`*`AffyStruct`*`, `*`PS`*`)` also returns the URL (linking to the NetAffx Web site) for the probe set information.

`probesetlink(`*`AffyStruct`*`, `*`PS`*`, ...'`*`PropertyName`*`', `*`PropertyValue`*`, ...)` calls `probesetlink` with optional properties that use property name/property value pairs. You can specify one or more properties in any order. Each *PropertyName* must be enclosed in single quotation marks and is case insensitive. These property name/property value pairs are as follows:

`probesetlink(AffyStruct, PS, ...'Source', SourceValue, ...)` controls the linking to the data source (for example, GenBank or Flybase) for the probe set (instead of linking to the NetAffx Web site). Choices are `true` or `false` (default).

Note The `'Source'` property requires the GIN library file associated with the CHP or CDF file to be located in the same folder as the CDF library file.

`probesetlink(AffyStruct, PS, ...'Browser', BrowserValue, ...)` controls the display of the probe set information in your system's default Web browser. Choices are `true` or `false` (default).

`URL = probesetlink(AffyStruct, PS, ...'NoDisplay', NoDisplayValue, ...)` controls the return of the URL without opening a Web browser. Choices are `true` or `false` (default).

Note The NetAffx Web site requires you to register and provide a user name and password.

Version History

Introduced before R2006a

R2023a: Removed

Errors starting in R2023a

`probesetlink` has been removed.

See Also

`affyread` | `celintensityread` | `probelibraryinfo` | `probesetlookup` | `probesetplot` | `probesetvalues`

Topics

"Working with Affymetrix Data"

probesetlookup

Look up information for Affymetrix probe set

Syntax

`PSStruct = probesetlookup(AffyStruct, ID)`

Input Arguments

`AffyStruct`	Structure created by the `affyread` function from an Affymetrix CHP file or an Affymetrix CDF library file for expression assays.
`ID`	Character vector, string, string vector, or cell array of character vectors specifying one or more probe set IDs/names or gene IDs.

Output Arguments

`PSStruct`	Structure or array of structures containing the following fields for a probe set: • `Identifier` — Gene ID associated with the probe set • `ProbeSetName` — Probe set ID/name • `CDFIndex` — Index into the CDF structure for the probe set • `GINIndex` — Index into the GIN structure for the probe set • `Description` — Description of the probe set • `Source` — Source(s) of the probe set • `SourceURL` — Source URL(s) for the probe set

Description

`PSStruct = probesetlookup(AffyStruct, ID)` returns a structure or an array of structures containing information for an Affymetrix probe set specified by `ID`, a character vector, string, ,string vector, or cell array of character vectors specifying one or more probe set IDs/names or gene IDs, and by *`AffyStruct`*, a structure created from an Affymetrix CHP file or Affymetrix CDF library file for expression assays.

Note This function works with CHP files and CDF files for expression assays only. It requires that the GIN library file associated with the CHP file or CDF file to be located in the same folder as the CDF library file.

Examples

The following example uses the CDF library file from the *E. coli* Antisense Genome array, which you can download from:

```
https://tools.thermofisher.com/content/sfs/supportFiles/ecoli_antisense_libraryfile.zip
```

The following example assumes that the `Ecoli_ASv2.CDF` library file is stored at `D:\Affymetrix\LibFiles\Ecoli`.

1. Read the contents of a CDF library file into a MATLAB structure.

   ```
   cdfStruct = affyread('D:\Affymetrix\LibFiles\Ecoli\Ecoli_ASv2.CDF');
   ```

2. Look up the gene ID (`Identifier`) associated with the `argG_b3172_at` probe set.

   ```
   probesetlookup(cdfStruct,'argG_b3172_at')

   ans = 

         Identifier: '3315278'
       ProbeSetName: 'argG_b3172_at'
           CDFIndex: 5213
           GINIndex: 3074
        Description: [1x82 char]
             Source: 'NCBI EColi Genome'
          SourceURL: [1x74 char]
   ```

Version History

Introduced before R2006a

See Also

`affyread` | `celintensityread` | `probelibraryinfo` | `probesetplot` | `probesetvalues` | `rmabackadj`

probesetplot

Plot Affymetrix probe set intensity values

Syntax

```
probesetplot(CELStruct, CDFStruct, PS)

probesetplot(CELStruct, CDFStruct, PS, ...'GeneName', GeneNameValue, ...)
probesetplot(CELStruct, CDFStruct, PS, ...'Field', FieldValue, ...)
probesetplot(CELStruct, CDFStruct, PS, ...'ShowStats', ShowStatsValue, ...)
```

Arguments

CELStruct	Structure created by the `affyread` function from an Affymetrix CEL file.
CDFStruct	Structure created by the `affyread` function from an Affymetrix CDF library file associated with the CEL file.
PS	Probe set index or the probe set ID/name.
GeneNameValue	Controls whether the probe set name or the gene name is used for the title of the plot. Choices are `true` or `false` (default). **Note** The `'GeneName'` property requires the GIN library file associated with the CEL and CDF files to be located in the same folder as the CDF library file from which *CDFStruct* was created.
FieldValue	Character vector or string specifying the type of data to plot. Choices are: • `'Intensity'` (default) • `'StdDev'` • `'Background'` • `'Pixels'` • `'Outlier'`
ShowStatsValue	Controls whether the mean and standard deviation lines are included in the plot. Choices are `true` or `false` (default).

Description

`probesetplot(CELStruct, CDFStruct, PS)` plots the PM (perfect match) and MM (mismatch) intensity values for a specified probe set. *CELStruct* is a structure created by the `affyread` function from an Affymetrix CEL file. *CDFStruct* is a structure created by the `affyread` function from an Affymetrix CDF library file associated with the CEL file. *PS* is the probe set index or the probe set ID/name.

Note MATLAB software uses 1-based indexing for probe set numbers, while the Affymetrix CDF file uses 0-based indexing for probe set numbers. For example, `CDFStruct.ProbeSets(1)` has a `ProbeSetNumber` of `0` in the `ProbePairs` field.

`probesetplot(`*`CELStruct`*`, `*`CDFStruct`*`, `*`PS`*`, ...'`*`PropertyName`*`', `*`PropertyValue`*`, ...)` calls `probesetplot` with optional properties that use property name/property value pairs. You can specify one or more properties in any order. Each *`PropertyName`* must be enclosed in single quotation marks and is case insensitive. These property name/property value pairs are as follows:

`probesetplot(`*`CELStruct`*`, `*`CDFStruct`*`, `*`PS`*`, ...'GeneName', `*`GeneNameValue`*`, ...)` controls whether the probe set name or the gene name is used for the title of the plot. Choices are `true` or `false` (default).

Note The `'GeneName'` property requires the GIN library file associated with the CEL and CDF files to be located in the same folder as the CDF library file from which *`CDFStruct`* was created.

`probesetplot(`*`CELStruct`*`, `*`CDFStruct`*`, `*`PS`*`, ...'Field', `*`FieldValue`*`, ...)` specifies the type of data to plot. Choices are:

- `'Intensity'` (default)
- `'StdDev'`
- `'Background'`
- `'Pixels'`
- `'Outlier'`

`probesetplot(`*`CELStruct`*`, `*`CDFStruct`*`, `*`PS`*`, ...'ShowStats', `*`ShowStatsValue`*`, ...)` controls whether the mean and standard deviation lines are included in the plot. Choices are `true` or `false` (default).

Examples

Plot Affymetrix™ Probe Set Intensity Values

You need some sample data files from here. This example uses the sample data from the *E. coli* Antisense Genome Array. Extract the data files from the DTT archive using the Data Transfer Tool.

You also need to download the corresponding library file for the sample. For this example, Ecoli_ASv2.CDF is used as for the *E. coli* Antisense Genome Array. You may already have these files if you have any Affymetrix GeneChip software installed on your machine. If not, get the library files by downloading and unzipping the *E. coli* Antisense Genome Array zip file.

Read the contents of a CEL file into a MATLAB structure.

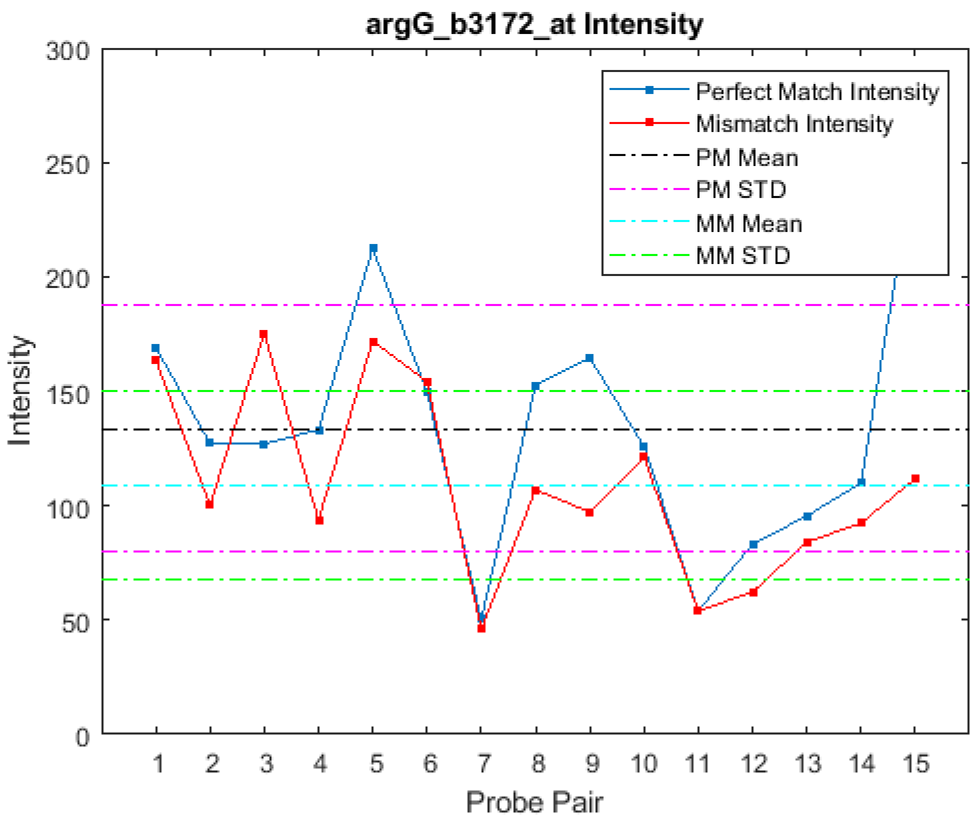
```
celStruct = affyread('Ecoli-antisense-121502.CEL');
```

Read the contents of a CDF file into a MATLAB structure.

```
cdfStruct = affyread('C:\LibFiles\Ecoli_ASv2.CDF');
```

Plot the PM and MM intensity values of the argG_b3172_at probe set, including the mean and standard deviation.

```
probesetplot(celStruct, cdfStruct, 'argG_b3172_at','showstats', true)
```



Version History

Introduced before R2006a

See Also

affyread | celintensityread | probesetlookup | probesetvalues

Topics

"Working with Affymetrix Data"

probesetvalues

Create table of Affymetrix probe set intensity values

Syntax

```
PSValues = probesetvalues(CELStruct, CDFStruct, PS)
PSValues = probesetvalues(CELStruct, CDFStruct, PS, 'Background',
BackgroundValue)
ColumnNames = probesetvalues
```

Input Arguments

CELStruct	Structure created by the `affyread` function from an Affymetrix CEL file.
CDFStruct	Structure created by the `affyread` function from an Affymetrix CDF library file associated with the CEL file.
PS	Probe set index or the probe set ID/name.
BackgroundValue	Controls the background correction in the calculation. Choices are: • `true` (default) — Background values from the `Background` field in the *PSValues* matrix are used to calculate the probe intensity values. • `false` — Background values are not calculated. • A vector of precalculated background values (such as returned by the `zonebackadj` function) whose length is equal to the number of probes in *CELStruct*. These background values are used to calculate the probe intensity values. **Tip** Including background correction in the calculation of the probe intensity values can be slow. Therefore, setting `'Background'` to `false` can speed up the calculation. However, the values returned in the `'Background'` field of the *PSValues* matrix will be zero.

Output Arguments

PSValues	Twenty-column matrix with one row for each probe pair in the probe set.
ColumnNames	Cell array of character vectors containing the column names of the *PSValues* matrix. This is returned only when you call `probesetvalues` with no input arguments.

Description

PSValues = probesetvalues(*CELStruct*, *CDFStruct*, *PS*) creates a table of intensity values for *PS*, a probe set, from the probe-level data in *CELStruct*, a structure created by the `affyread` function from an Affymetrix CEL file. `PS` is a probe set index or probe set ID/name from `CDFStruct`, a structure created by the `affyread` function from an Affymetrix CDF library file associated with the CEL file. `PSValues` is a twenty-column matrix with one row for each probe pair in the probe set. The columns correspond to the following fields.

Column	Field	Description
1	`'ProbeSetNumber'`	Number identifying the probe set to which the probe pair belongs.
2	`'ProbePairNumber'`	Index of the probe pair within the probe set.
3	`'UseProbePair'`	This field is for backward compatibility only and is not currently used.
4	`'Background'`	Estimated background of probe intensity values of the probe pair.
5	`'PMPosX'`	*x*-coordinate of the perfect match probe.
6	`'PMPosY'`	*y*-coordinate of the perfect match probe.
7	`'PMIntensity'`	Intensity value of the perfect match probe.
8	`'PMStdDev'`	Standard deviation of intensity value of the perfect match probe.
9	`'PMPixels'`	Number of pixels in the cell containing the perfect match probe.
10	`'PMOutlier'`	True/false flag indicating if the perfect match probe was marked as an outlier.
11	`'PMMasked'`	True/false flag indicating if the perfect match probe was masked.
12	`'MMPosX'`	*x*-coordinate of the mismatch probe.
13	`'MMPosY'`	*y*-coordinate of the mismatch probe.
14	`'MMIntensity'`	Intensity value of the mismatch probe.
15	`'MMStdDev'`	Standard deviation of intensity value of the mismatch probe.
16	`'MMPixels'`	Number of pixels in the cell containing the mismatch probe.
17	`'MMOutlier'`	True/false flag indicating if the mismatch probe was marked as an outlier.
18	`'MMMasked'`	True/false flag indicating if the mismatch probe was masked.
19	`'GroupNumber'`	Number identifying the group to which the probe pair belongs. For expression arrays, this is always `1`. For genotyping arrays, this is typically `1` (allele A, sense), `2` (allele B, sense), `3` (allele A, antisense), or `4` (allele B, antisense).
20	`'Direction'`	Number identifying the direction of the probe pair. `1` = sense and `2` = antisense.

Note MATLAB software uses 1-based indexing for probe set numbers, while the Affymetrix CDF file uses 0-based indexing for probe set numbers. For example, `CDFStruct.ProbeSets(1)` has a `ProbeSetNumber` of `0` in the `ProbePairs` field.

`*PSValues* = probesetvalues(*CELStruct*, *CDFStruct*, *PS*, 'Background', *BackgroundValue*)` controls the background correction in the calculation. *`BackgroundValue`* can be:

- `true` (default) — Background values from the `Background` field in the *`PSValues`* matrix are used to calculate the probe intensity values.
- `false` — Background values are not calculated.
- A vector of precalculated background values (such as returned by the `zonebackadj` function) whose length is equal to the number of probes in *`CELStruct`*. These background values are used to calculate the probe intensity values.

Tip Including background correction in the calculation of the probe intensity values can be slow. Therefore, setting `'Background'` to `false` can speed up the calculation. However, the values returned in the `'Background'` field of the *`PSValues`* matrix will be zero.

`*ColumnNames* = probesetvalues` returns a cell array of character vectors containing the column names of the *`PSValues`* matrix. *`ColumnNames`* is returned only when you call `probesetvalues` without input arguments. The information contained in *`ColumnNames`* is common to all Affymetrix GeneChip arrays.

Examples

Retrieve Affymetrix™ Probe Set Intensity Values

For this example, you need some sample data files from here. This example uses the sample data from the *E. coli* Antisense Genome Array. Extract the data files from the DTT archive using the Data Transfer Tool.

You also need to download the corresponding library file for the sample. For this example, Ecoli_ASv2.CDF is used as for the *E. coli* Antisense Genome Array. You may already have these files if you have any Affymetrix GeneChip software installed on your machine. If not, get the library files by downloading and unzipping the *E. coli* Antisense Genome Array zip file.

Read the contents of a CEL file into a MATLAB structure.

```
celStruct = affyread('Ecoli-antisense-121502.CEL');
```

Read the contents of a CDF file into a MATLAB structure.

```
cdfStruct = affyread('C:\LibFiles\Ecoli_ASv2.CDF');
```

Use the `zonebackadj` function to return a matrix or cell array of vectors containing the estimated background values for each probe.

```
[baData,zones,background] = zonebackadj(celStruct,'cdf',cdfStruct);
```

Create a table of intensity values for the `argG_b3172_at` probe set.

```
psvals = probesetvalues(celStruct, cdfStruct, 'argG_b3172_at',...
         'background',background)


psvals =

   1.0e+03 *

  Columns 1 through 7

    5.2120         0         0    0.0454    0.4300    0.1770    0.1690
    5.2120    0.0010         0    0.0455    0.4310    0.1770    0.1273
    5.2120    0.0020         0    0.0455    0.4320    0.1770    0.1270
    5.2120    0.0030         0    0.0455    0.4330    0.1770    0.1333
    5.2120    0.0040         0    0.0455    0.4340    0.1770    0.2123
    5.2120    0.0050         0    0.0455    0.4350    0.1770    0.1495
    5.2120    0.0060         0    0.0455    0.4360    0.1770    0.0503
    5.2120    0.0070         0    0.0456    0.4370    0.1770    0.1525
    5.2120    0.0080         0    0.0456    0.4380    0.1770    0.1645
    5.2120    0.0090         0    0.0456    0.4390    0.1770    0.1260
    5.2120    0.0100         0    0.0456    0.4400    0.1770    0.0540
    5.2120    0.0110         0    0.0456    0.4410    0.1770    0.0833
    5.2120    0.0120         0    0.0457    0.4420    0.1770    0.0955
    5.2120    0.0130         0    0.0457    0.4430    0.1770    0.1100
    5.2120    0.0140         0    0.0457    0.4440    0.1770    0.2510

  Columns 8 through 14

    0.0354    0.0250         0         0    0.4300    0.1780    0.1635
    0.0218    0.0300         0         0    0.4310    0.1780    0.1003
    0.0237    0.0300         0         0    0.4320    0.1780    0.1750
    0.0259    0.0360         0         0    0.4330    0.1780    0.0940
    0.0433    0.0360         0         0    0.4340    0.1780    0.1718
    0.0275    0.0360         0         0    0.4350    0.1780    0.1540
    0.0112    0.0300         0         0    0.4360    0.1780    0.0460
    0.0377    0.0360         0         0    0.4370    0.1780    0.1070
    0.0312    0.0360         0         0    0.4380    0.1780    0.0973
    0.0234    0.0360         0         0    0.4390    0.1780    0.1213
    0.0112    0.0360         0         0    0.4400    0.1780    0.0540
    0.0174    0.0360         0         0    0.4410    0.1780    0.0623
    0.0171    0.0300         0         0    0.4420    0.1780    0.0840
    0.0196    0.0360         0         0    0.4430    0.1780    0.0925
    0.0460    0.0360         0         0    0.4440    0.1780    0.1118

  Columns 15 through 20

    0.0241    0.0300         0         0    0.0010    0.0020
    0.0146    0.0360         0         0    0.0010    0.0020
    0.0286    0.0360         0         0    0.0010    0.0020
    0.0227    0.0300         0         0    0.0010    0.0020
    0.0365    0.0300         0         0    0.0010    0.0020
    0.0303    0.0300         0         0    0.0010    0.0020
    0.0098    0.0250         0         0    0.0010    0.0020
    0.0210    0.0360         0         0    0.0010    0.0020
    0.0219    0.0360         0         0    0.0010    0.0020
    0.0253    0.0360         0         0    0.0010    0.0020
    0.0129    0.0360         0         0    0.0010    0.0020
    0.0125    0.0360         0         0    0.0010    0.0020
    0.0186    0.0300         0         0    0.0010    0.0020
    0.0220    0.0360         0         0    0.0010    0.0020
    0.0207    0.0360         0         0    0.0010    0.0020
```

Version History

Introduced before R2006a

See Also

`affyread` | `celintensityread` | `probelibraryinfo` | `probesetlookup` | `probesetplot` | `rmabackadj` | `zonebackadj`

Topics

"Working with Affymetrix Data"

processTable

Package: bioinfo.pipeline

Return information about all processes in pipeline

Syntax

```
t = processTable(pipeline)
t = processTable(pipeline,blocks)
t = processTable( ___ ,Expanded=tf)
```

Description

t = processTable(pipeline) returns a table t that contains information about all processes in a pipeline, including block-specific run start and end times and run statuses.

t = processTable(pipeline,blocks) returns a table t that contains information only for the specified blocks.

t = processTable(___ ,Expanded=tf) specifies whether to expand t to contain one row for each independent run of a block, or to collapse multiple runs of the same block into a single row.

Examples

Check Bioinformatics Pipeline Run Status and Results

Import the pipeline and block objects needed for the example.

```
import bioinfo.pipeline.Pipeline
import bioinfo.pipeline.blocks.*
```

Create a pipeline.

```
P = Pipeline;
```

Create a FileChooser block that takes in a SAM file as an input.

```
samFile = FileChooser(which("Myco_1_1.sam"));
```

Create a SeqTrim block.

```
trimsequences = SeqTrim;
```

Add blocks to the pipeline.

```
addBlock(P,[samFile,trimsequences]);
```

Check the names of the output port and input port of the blocks to connect.

```
samFile.Outputs
```

```
ans = struct with fields:
    Files: [1×1 bioinfo.pipeline.Output]
```

```
trimsequences.Inputs
```

```
ans = struct with fields:
    FASTQFiles: [1×1 bioinfo.pipeline.Input]
```

Connect the ports.

```
connect(P,samFile,trimsequences,["Files","FASTQFiles"]);
```

The block returns an incomplete result if the block has not yet run.

```
results(P,trimsequences)
```

```
ans = 
  Incomplete pipeline result.
```

Run the pipeline and check the run status of each block in the process table. The `samFile` block had no error, but the `SeqTrim` block generated an error as indicated by the `Status` column. The `SeqTrim` block generated an error because it was expecting a FASTQ file as an input but received a SAM file instead.

```
run(P);
t = processTable(P,Expanded=true)
```

```
t=2×5 table
         Block           Status             RunStart                   RunEnd                RunErrors
    _______________    _________    ____________________    ____________________    _____________

    "FileChooser_1"    Completed    22-Jan-2023 17:32:44    22-Jan-2023 17:32:44    {0×0 MExcept
    "SeqTrim_1"        Error        22-Jan-2023 17:32:45    22-Jan-2023 17:32:45    {1×1 MExcept
```

You can see the printed error message at the MATLAB command line. You can also enter the following commands to see the message.

```
seqTrimInfo = t(2,:);
seqTrimInfo.RunErrors{:}
```

Input Arguments

pipeline — Bioinformatics pipeline

`bioinfo.pipeline.Pipeline` object

Bioinformatics pipeline, specified as a `bioinfo.pipeline.Pipeline` object.

blocks — Pipeline blocks

`bioinfo.pipeline.Block` object | vector of objects | character vector | string scalar | ...

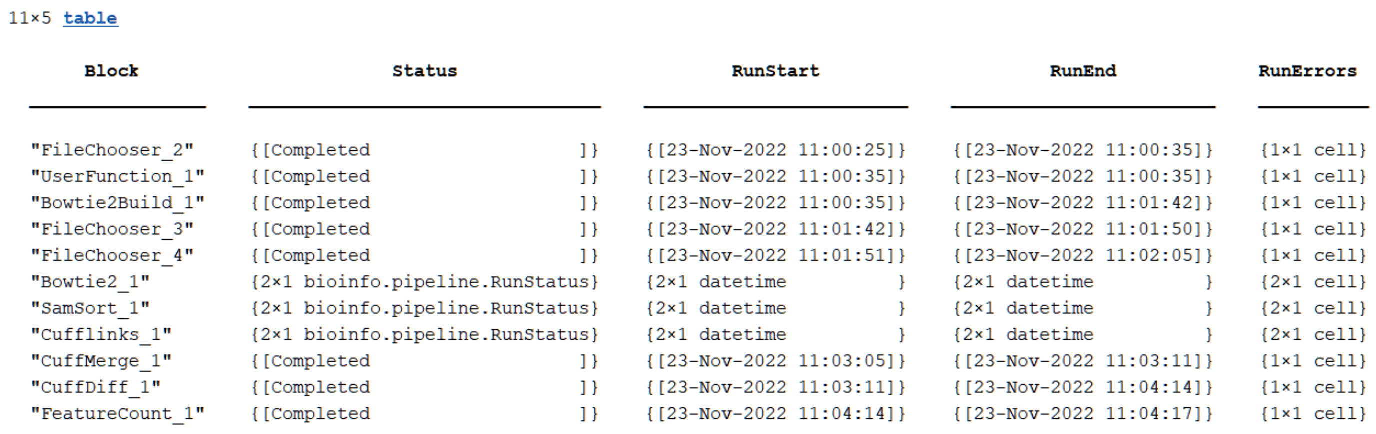
Pipeline blocks, specified as a `bioinfo.pipeline.Block` object, vector of block objects, character vector, string scalar, string vector, or cell array of character vectors representing block names.

`tf` — Flag to contain one row per each independent block run
`false` or 0 (default) | `true` or 1

Flag to contain one row per each independent block run or group all runs of the same block into a single row, specified as a numeric or logical 1 (`true`) or 0 (`false`).

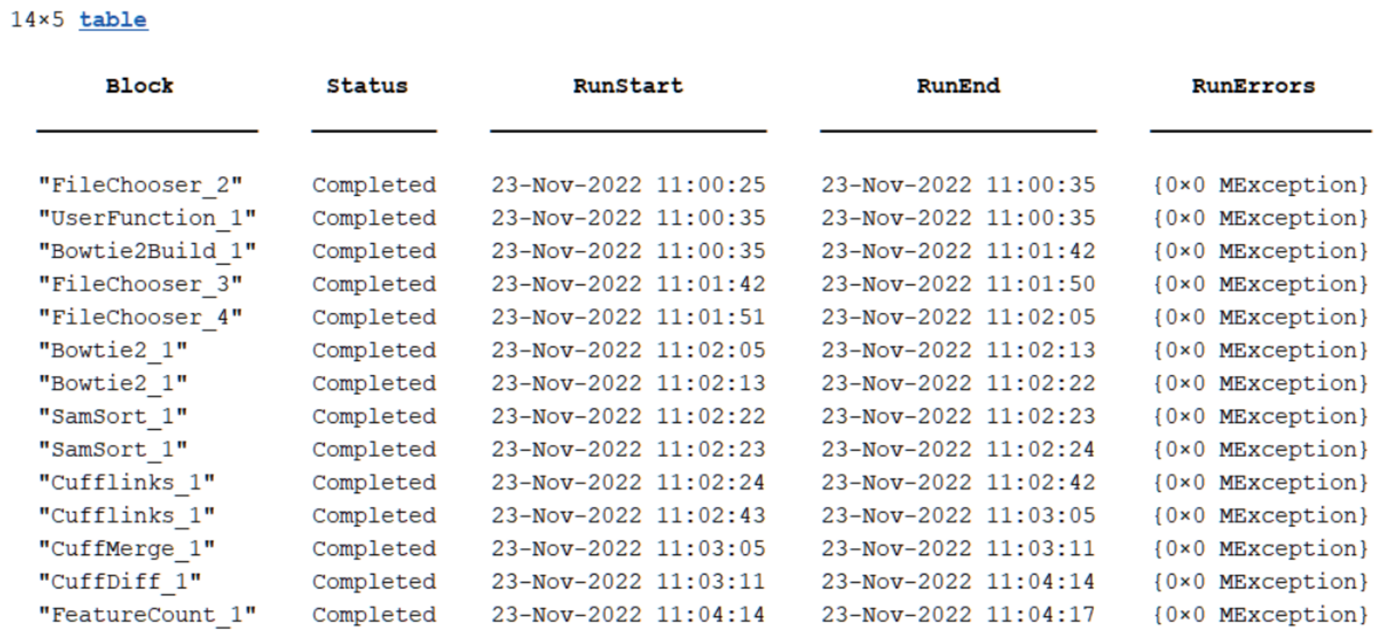
Each row in the table represents a block, and there are corresponding *Status*, *RunStart*, *RunEnd*, and *RunErrors* columns for each block.

By default, the function groups (or collapses) all runs of the same block into a single row. If a block is run *n* independent times (per its `SplitDimension` property of the input port), then the table variable value for each column is an *n*-by-1 cell array. For instance, the next figure is an example of a (default) collapsed process table (`Expanded = false`). Note that for the *Bowtie2_1*, *SamSort_1* and *Cufflinks_1* blocks, the table variable values are 2-by-1 cell arrays because each of these blocks were run 2 times independently.



```
11×5 table

       Block                    Status                       RunStart                   RunEnd             RunErrors
   _______________    ________________________________   ________________________   ________________________   __________

   "FileChooser_2"     {[Completed                   ]}   {[23-Nov-2022 11:00:25]}   {[23-Nov-2022 11:00:35]}   {1×1 cell}
   "UserFunction_1"    {[Completed                   ]}   {[23-Nov-2022 11:00:35]}   {[23-Nov-2022 11:00:35]}   {1×1 cell}
   "Bowtie2Build_1"    {[Completed                   ]}   {[23-Nov-2022 11:00:35]}   {[23-Nov-2022 11:01:42]}   {1×1 cell}
   "FileChooser_3"     {[Completed                   ]}   {[23-Nov-2022 11:01:42]}   {[23-Nov-2022 11:01:50]}   {1×1 cell}
   "FileChooser_4"     {[Completed                   ]}   {[23-Nov-2022 11:01:51]}   {[23-Nov-2022 11:02:05]}   {1×1 cell}
   "Bowtie2_1"         {2×1 bioinfo.pipeline.RunStatus}   {2×1 datetime          }   {2×1 datetime          }   {2×1 cell}
   "SamSort_1"         {2×1 bioinfo.pipeline.RunStatus}   {2×1 datetime          }   {2×1 datetime          }   {2×1 cell}
   "Cufflinks_1"       {2×1 bioinfo.pipeline.RunStatus}   {2×1 datetime          }   {2×1 datetime          }   {2×1 cell}
   "CuffMerge_1"       {[Completed                   ]}   {[23-Nov-2022 11:03:05]}   {[23-Nov-2022 11:03:11]}   {1×1 cell}
   "CuffDiff_1"        {[Completed                   ]}   {[23-Nov-2022 11:03:11]}   {[23-Nov-2022 11:04:14]}   {1×1 cell}
   "FeatureCount_1"    {[Completed                   ]}   {[23-Nov-2022 11:04:14]}   {[23-Nov-2022 11:04:17]}   {1×1 cell}
```

The next figure is the same table with the expanded cell array (`Expanded = true`).



```
14×5 table

       Block           Status          RunStart                RunEnd              RunErrors
   ________________    _________   ____________________   ____________________   _________________

   "FileChooser_2"     Completed   23-Nov-2022 11:00:25   23-Nov-2022 11:00:35   {0×0 MException}
   "UserFunction_1"    Completed   23-Nov-2022 11:00:35   23-Nov-2022 11:00:35   {0×0 MException}
   "Bowtie2Build_1"    Completed   23-Nov-2022 11:00:35   23-Nov-2022 11:01:42   {0×0 MException}
   "FileChooser_3"     Completed   23-Nov-2022 11:01:42   23-Nov-2022 11:01:50   {0×0 MException}
   "FileChooser_4"     Completed   23-Nov-2022 11:01:51   23-Nov-2022 11:02:05   {0×0 MException}
   "Bowtie2_1"         Completed   23-Nov-2022 11:02:05   23-Nov-2022 11:02:13   {0×0 MException}
   "Bowtie2_1"         Completed   23-Nov-2022 11:02:13   23-Nov-2022 11:02:22   {0×0 MException}
   "SamSort_1"         Completed   23-Nov-2022 11:02:22   23-Nov-2022 11:02:23   {0×0 MException}
   "SamSort_1"         Completed   23-Nov-2022 11:02:23   23-Nov-2022 11:02:24   {0×0 MException}
   "Cufflinks_1"       Completed   23-Nov-2022 11:02:24   23-Nov-2022 11:02:42   {0×0 MException}
   "Cufflinks_1"       Completed   23-Nov-2022 11:02:43   23-Nov-2022 11:03:05   {0×0 MException}
   "CuffMerge_1"       Completed   23-Nov-2022 11:03:05   23-Nov-2022 11:03:11   {0×0 MException}
   "CuffDiff_1"        Completed   23-Nov-2022 11:03:11   23-Nov-2022 11:04:14   {0×0 MException}
   "FeatureCount_1"    Completed   23-Nov-2022 11:04:14   23-Nov-2022 11:04:17   {0×0 MException}
```

Output Arguments

`t` — Table containing information on pipeline processes
table

Table containing information on pipeline processes, returned as a table.

Each row in the table represents a block, and there are corresponding *Status*, *RunStart*, *RunEnd*, and *RunErrors* columns for each block.

Version History

Introduced in R2023a

See Also

`bioinfo.pipeline.Block` | `bioinfo.pipeline.Pipeline` | **Biopipeline Designer**

profalign

Align two profiles using Needleman-Wunsch global alignment

Syntax

```
Prof = profalign(Prof1, Prof2)
[Prof, H1, H2] = profalign(Prof1, Prof2)

profalign(..., 'ScoringMatrix', ScoringMatrixValue, ...)
profalign(..., 'GapOpen', {G1Value, G2Value}, ...)
profalign(..., 'ExtendGap', {E1Value, E2Value}, ...)
profalign(..., 'ExistingGapAdjust', ExistingGapAdjustValue, ...)
profalign(..., 'TerminalGapAdjust', TerminalGapAdjustValue, ...)
profalign(..., 'ShowScore', ShowScoreValue, ...)
```

Description

`Prof = profalign(Prof1, Prof2)` returns a new profile (`Prof`) for the optimal global alignment of two profiles (`Prof1`, `Prof2`). The profiles (`Prof1`, `Prof2`) are numeric arrays of size `[(4 or 5 or 20 or 21) x Profile Length]` with counts or weighted profiles. Weighted profiles are used to down-weight similar sequences and up-weight divergent sequences. The output profile is a numeric matrix of size `[(5 or 21) x New Profile Length]` where the last row represents gaps. Original gaps in the input profiles are preserved. The output profile is the result of adding the aligned columns of the input profiles.

`[Prof, H1, H2] = profalign(Prof1, Prof2)` returns pointers that indicate how to rearrange the columns of the original profiles into the new profile.

`profalign(..., 'PropertyName', PropertyValue, ...)` calls `profalign` with optional properties that use property name/property value pairs. You can specify one or more properties in any order. Each *`PropertyName`* must be enclosed in single quotation marks and is case insensitive. These property name/property value pairs are as follows:

`profalign(..., 'ScoringMatrix', ScoringMatrixValue, ...)` defines the scoring matrix to be used for the alignment.

`ScoringMatrixValue` can be either of the following:

- Character vector or string specifying the scoring matrix to use for the alignment. Choices for amino acid sequences are:
  - `'BLOSUM62'`
  - `'BLOSUM30'` increasing by `5` up to `'BLOSUM90'`
  - `'BLOSUM100'`
  - `'PAM10'` increasing by `10` up to `'PAM500'`
  - `'DAYHOFF'`
  - `'GONNET'`

Default is:

- `'BLOSUM50'` — When *`AlphabetValue`* equals `'AA'`
- `'NUC44'` — When *`AlphabetValue`* equals `'NT'`

Note The above scoring matrices, provided with the software, also include a structure containing a scale factor that converts the units of the output score to bits. You can also use the `'Scale'` property to specify an additional scale factor to convert the output score from bits to another unit.

- Matrix representing the scoring matrix to use for the alignment, such as returned by the `blosum`, `pam`, `dayhoff`, `gonnet`, or `nuc44` function.

Note If you use a scoring matrix that you created or was created by one of the above functions, the matrix does not include a scale factor. The output score will be returned in the same units as the scoring matrix.

Note If you need to compile `profalign` into a stand-alone application or software component using MATLAB Compiler, use a matrix instead of a character vector or string for *`ScoringMatrixValue`*.

`profalign(..., 'GapOpen', {`*`G1Value`*`, `*`G2Value`*`}, ...)` sets the penalties for opening a gap in the first and second profiles respectively. *`G1Value`* and *`G2Value`* can be either scalars or vectors. When using a vector, the number of elements is one more than the length of the input profile. Every element indicates the position specific penalty for opening a gap between two consecutive symbols in the sequence. The first and the last elements are the gap penalties used at the ends of the sequence. The default gap open penalties are `{10,10}`.

`profalign(..., 'ExtendGap', {`*`E1Value`*`, `*`E2Value`*`}, ...)` sets the penalties for extending a gap in the first and second profile respectively. *`E1Value`* and *`E2Value`* can be either scalars or vectors. When using a vector, the number of elements is one more than the length of the input profile. Every element indicates the position specific penalty for extending a gap between two consecutive symbols in the sequence. The first and the last elements are the gap penalties used at the ends of the sequence. If `ExtendGap` is not specified, then extensions to gaps are scored with the same value as `GapOpen`.

`profalign(..., 'ExistingGapAdjust', `*`ExistingGapAdjustValue`*`, ...)`, if *`ExistingGapAdjustValue`* is `false`, turns off the automatic adjustment based on existing gaps of the position-specific penalties for opening a gap. When *`ExistingGapAdjustValue`* is `true` (default), for every profile position, `profalign` proportionally lowers the penalty for opening a gap toward the penalty of extending a gap based on the proportion of gaps found in the contiguous symbols and on the weight of the input profile.

`profalign(..., 'TerminalGapAdjust', `*`TerminalGapAdjustValue`*`, ...)`, when *`TerminalGapAdjustValue`* is `true`, adjusts the penalty for opening a gap at the ends of the sequence to be equal to the penalty for extending a gap. Default is `false`.

`profalign(..., 'ShowScore', `*`ShowScoreValue`*`, ...)`, when *`ShowScoreValue`* is `true`, displays the scoring space and the winning path.

Examples

1. Read in sequences and create profiles.

```
ma1 = ['RGTANCDMQDA';'RGTAHCDMQDA';'RRRAPCDL-DA'];
ma2 = ['RGTHCDLADAT';'RGTACDMADAA'];
p1  = seqprofile(ma1,'gaps','all','counts',true);
p2  = seqprofile(ma2,'counts',true);
```

2. Merge two profiles into a single one by aligning them.

```
p = profalign(p1,p2);
seqlogo(p)
```

3. Use the output pointers to generate the multiple alignment.

```
[p, h1, h2] = profalign(p1,p2);
ma = repmat('-',5,12);
ma(1:3,h1) = ma1;
ma(4:5,h2) = ma2;
disp(ma)
```

4. Increase the gap penalty before cysteine in the second profile.

```
gapVec = 10 + [p2(aa2int('C'),:) 0] * 10
p3 = profalign(p1,p2,'gapopen',{10,gapVec});
seqlogo(p3)
```

5. Add a new sequence to a profile without inserting new gaps into the profile.

```
gapVec = [0 inf(1,11) 0];
p4 = profalign(p3,seqprofile('PLHFMSVLWDVQQWP'),...
               'gapopen',{gapVec,10});
seqlogo(p4)
```

Version History

Introduced before R2006a

See Also

multialign | seqconsensus

Topics

hmmprofalign on page 1-1094
nwalign on page 1-1408
seqprofile on page 1-1735

proteinplot

Open Protein Plot window to investigate properties of amino acid sequence

Syntax

```
proteinplot
proteinplot (SeqAA)
```

Arguments

SeqAA	Either of the following: • Character vector or string containing single-letter codes specifying an amino acid sequence. For valid letter codes, see the table Mapping Amino Acid Letter Codes to Integers. Unknown characters are mapped to `0`. • MATLAB structure containing a `Sequence` field that contains an amino acid sequence, such as returned by `fastaread`, `getgenpept`, `genpeptread`, `getpdb`, or `pdbread`.

Description

The Protein Plot window lets you analyze and compare properties of a single amino acid sequence. It displays smoothed line plots of various properties such as the hydrophobicity of the amino acids in the sequence.

`proteinplot` opens the Protein Plot window.

`proteinplot (`*`SeqAA`*`)` opens the Protein Plot window and loads *SeqAA*, an amino acid sequence, into the window.

Tip You can analyze and compare properties of an amino acid sequence from the MATLAB command line also by using the `proteinpropplot` function.

Examples

Example 1.38. Importing Sequences into the Protein Plot Window

You can import a sequence into the Protein Plot window from the MATLAB command line.

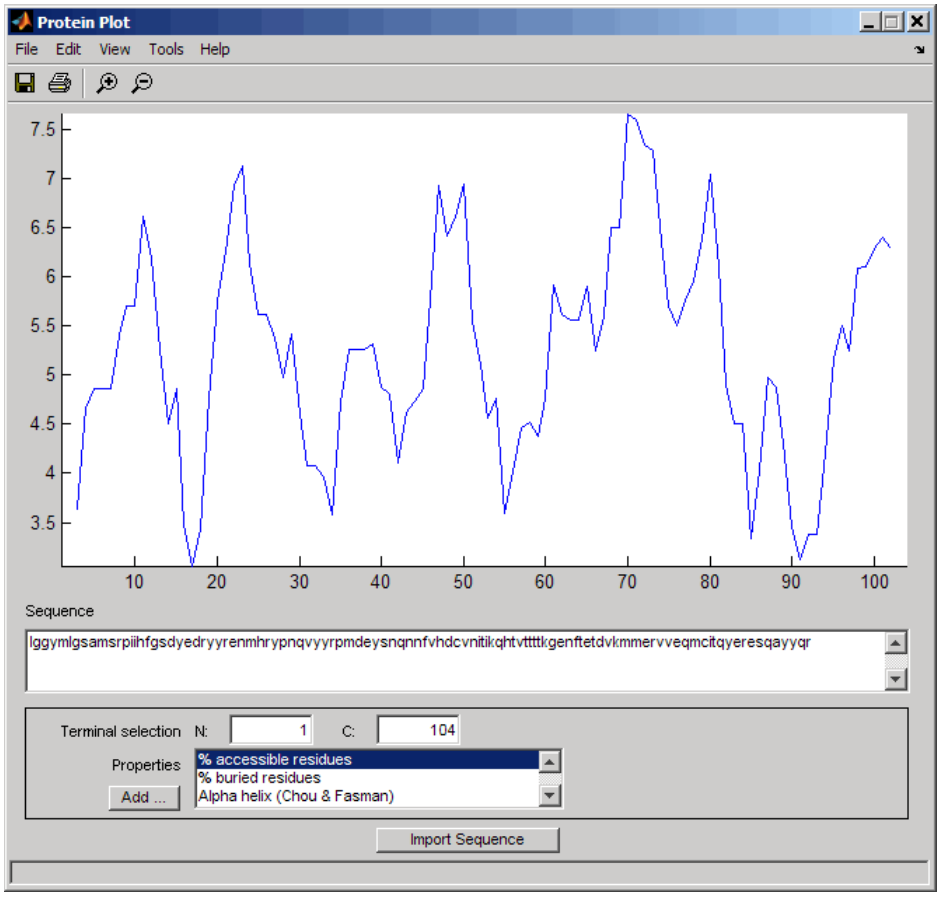
1 Retrieve an amino acid sequence from the Protein Data Bank (PDB) database.

```
prion = getpdb('1HJM', 'SEQUENCEONLY', true);
```

2 Load the amino acid sequence into the Protein Plot window.

```
proteinplot(prion)
```

The Protein Plot window opens, and the sequence appears in the **Sequence** text box.



Sequence: lggymlgsamsrpiihfgsdyedryyrenmhrypnqvyyrpmdeysnqnnfvhdcvnitikqhtvttttkgenftetdvkmmervveqmcitqyeresqayyqr

Terminal selection N: 1 C: 104

Properties: % accessible residues, % buried residues, Alpha helix (Chou & Fasman)

You can import a sequence after the Protein Plot window is open by doing either of the following:

- Type or paste an amino acid sequence into the **Sequence** text box.
- Click the **Import Sequence** button to open the Import dialog box From the **Import From** list, select one of the following:

  - **Workspace** — To select a variable from the MATLAB Workspace
  - **Text File** — To select a text file
  - **FASTA File** — To select a FASTA-formatted file
  - **GenPept File** — To select a GenPept-formatted file
  - **GenPept Database** — To specify an accession number in the GenPept database

Example 1.39. Viewing Properties of Amino Acids

Select a property from the **Properties** drop-down list box to display a smoothed plot of the property values along the sequence. You can select multiple properties from the list by holding down **Shift** or **Ctrl** while selecting properties. When you select two properties, the plots are displayed using two y-axes, with one y-axis on the left and one on the right. For all other selections, a single *y*-axis is displayed. When displaying one or two properties, the *y* values displayed are the actual property values. When displaying three or more properties, the values are normalized to the range 0–1.

Example 1.40. Accessing Information About the Properties

You can access information about the properties from the **Help** menu.

1 Select **Help > References**. The Help browser opens with a list of properties and references.
2 Scroll down to locate the property of interest.

Example 1.41. Using Other Features in the Protein Plot Window

The **Terminal Selection** boxes (N and C) let you choose to plot only part of the sequence. By default, all of the sequence is plotted.

You can add your own properties by clicking on the **Add** button next to the **Properties** list. This opens a Property dialog box that lets you specify the value for each of the amino acids. The **Display Text** box lets you specify the text that will be displayed in the **Properties** list on the main Protein Plot window. You can also save the property values to a file for future use by typing a file name in the **Filename** text box.

The default smoothing method is an unweighted linear moving average with a window length of five residues. You can change this by selecting **Edit > Filter Window Options**. The dialog box lets you select the **Window Size** from `5` to `29` residues. Increasing the window size produces a smoother plot. You can modify the shape of the smoothing window by changing the **Edge Weight** factor. And you can choose the smoothing function to be a linear moving average, an exponential moving average or a linear Lowess smoothing.

The **File** menu lets you import a sequence, save the plot that you have created to a Figure file, export the data values in the figure to a workspace variable or to a MAT-file, export the figure to a normal Figure window for customizing, or print the figure.

The **Edit** menu lets you create a new property, to reset the property values to the default values, and to modify the smoothing parameters with the **Configuration Values** menu item.

The **View** menu lets you turn the toolbar on and off, and to add a legend to the plot.

The **Tools** menu lets you zoom in and zoom out of the plot, to view **Data Statistics** such as mean, minimum and maximum values of the plot, and to normalize the values of the plot from `0` to `1`.

The **Help** menu lets you view this document and to see the references for the sequence properties included with the Protein Plot window.

Version History

Introduced before R2006a

See Also

`aacount` | `atomiccomp` | `molweight` | `pdbdistplot` | `proteinpropplot` | `seqviewer` | `yyaxis`

proteinpropplot

Plot properties of amino acid sequence

Syntax

```
proteinpropplot (SeqAA)

proteinpropplot(SeqAA, ...'PropertyTitle', PropertyTitleValue, ...)
proteinpropplot(SeqAA, ...'Startat', StartatValue, ...)
proteinpropplot(SeqAA, ...'Endat', EndatValue, ...)
proteinpropplot(SeqAA, ...'Smoothing', SmoothingValue, ...)
proteinpropplot(SeqAA, ...'EdgeWeight', EdgeWeightValue, ...)
proteinpropplot(SeqAA, ...'WindowLength', WindowLengthValue, ...)
```

Arguments

SeqAA	Amino acid sequence. Enter any of the following: • Character vector or string containing letters representing an amino acid • Vector of integers representing an amino acid, such as returned by `aa2int` • Structure containing a `Sequence` field that contains an amino acid sequence, such as returned by `getembl`, `getgenpept`, or `getpdb`
PropertyTitleValue	Character vector or string that specifies the property to plot. Default is `Hydrophobicity (Kyte & Doolittle)`. To display a list of properties to plot, enter a empty character vector or empty string for *PropertyTitleValue*. For example, type: `proteinpropplot(sequence, 'propertytitle', '')` **Tip** To access references for the properties, view the `proteinpropplot` file.
StartatValue	Integer that specifies the starting point for the plot from the N-terminal end of the amino acid sequence *SeqAA*. Default is `1`.
EndatValue	Integer that specifies the ending point for the plot from the N-terminal end of the amino acid sequence *SeqAA*. Default is `length(SeqAA)`.
SmoothingValue	Character vector or string the specifies the smoothing method. Choices are: • `linear` (default) • `exponential` • `lowess`

EdgeWeightValue	Value that specifies the edge weight used for linear and exponential smoothing methods. Decreasing this value emphasizes peaks in the plot. Choices are any value ≥0 and ≤1. Default is 1.
WindowLengthValue	Integer that specifies the window length for the smoothing method. Increasing this value gives a smoother plot that shows less detail. Default is 11.

Description

`proteinpropplot (`*`SeqAA`*`)` displays a plot of the hydrophobicity (Kyte and Doolittle, 1982 on page 1-1529) of the residues in sequence `SeqAA`.

`proteinpropplot(`*`SeqAA`*`, ...'`*`PropertyName`*`', `*`PropertyValue`*`, ...)` calls `proteinpropplot` with optional properties that use property name/property value pairs. You can specify one or more properties in any order. Each *`PropertyName`* must be enclosed in single quotation marks and is case insensitive. These property name/property value pairs are as follows:

`proteinpropplot(`*`SeqAA`*`, ...'PropertyTitle', `*`PropertyTitleValue`*`, ...)` specifies a property to plot for the amino acid sequence *`SeqAA`*. Default is `Hydrophobicity (Kyte & Doolittle)`. To display a list of possible properties to plot, enter an empty character vector or empty string for *`PropertyTitleValue`*. For example, type:

```
proteinpropplot(sequence, 'propertytitle', '')
```

Tip To access references for the properties, view the `proteinpropplot` file.

`proteinpropplot(`*`SeqAA`*`, ...'Startat', `*`StartatValue`*`, ...)` specifies the starting point for the plot from the N-terminal end of the amino acid sequence *`SeqAA`*. Default is `1`.

`proteinpropplot(`*`SeqAA`*`, ...'Endat', `*`EndatValue`*`, ...)` specifies the ending point for the plot from the N-terminal end of the amino acid sequence *`SeqAA`*. Default is `length(`*`SeqAA`*`)`.

`proteinpropplot(`*`SeqAA`*`, ...'Smoothing', `*`SmoothingValue`*`, ...)` specifies the smoothing method. Choices are:

- `linear` (default)
- `exponential`
- `lowess`

`proteinpropplot(`*`SeqAA`*`, ...'EdgeWeight', `*`EdgeWeightValue`*`, ...)` specifies the edge weight used for linear and exponential smoothing methods. Decreasing this value emphasizes peaks in the plot. Choices are any value ≥`0` and ≤`1`. Default is `1`.

`proteinpropplot(`*`SeqAA`*`, ...'WindowLength', `*`WindowLengthValue`*`, ...)` specifies the window length for the smoothing method. Increasing this value gives a smoother plot that shows less detail. Default is `11`.

Examples

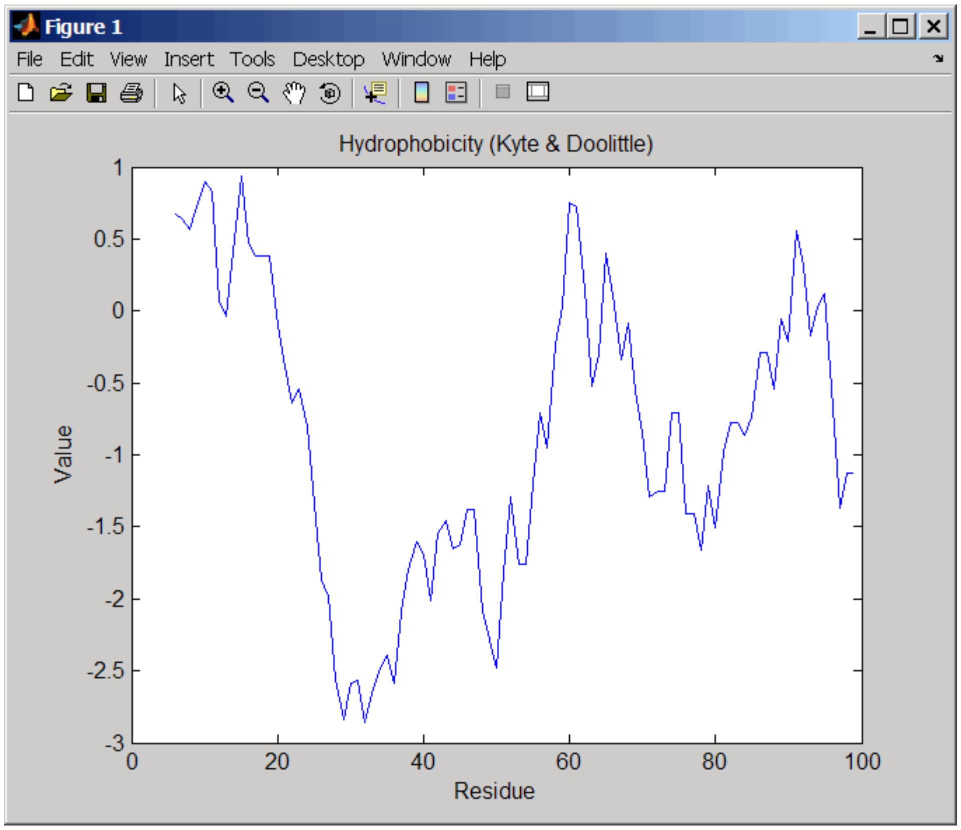
Example 1.42. Plotting Hydrophobicity

1 Use the `getpdb` function to retrieve a protein sequence.

```
prion = getpdb('1HJM', 'SEQUENCEONLY', true);
```

2 Plot the hydrophobicity (Kyte and Doolittle, 1982 on page 1-1529) of the residues in the sequence.

```
proteinpropplot(prion)
```



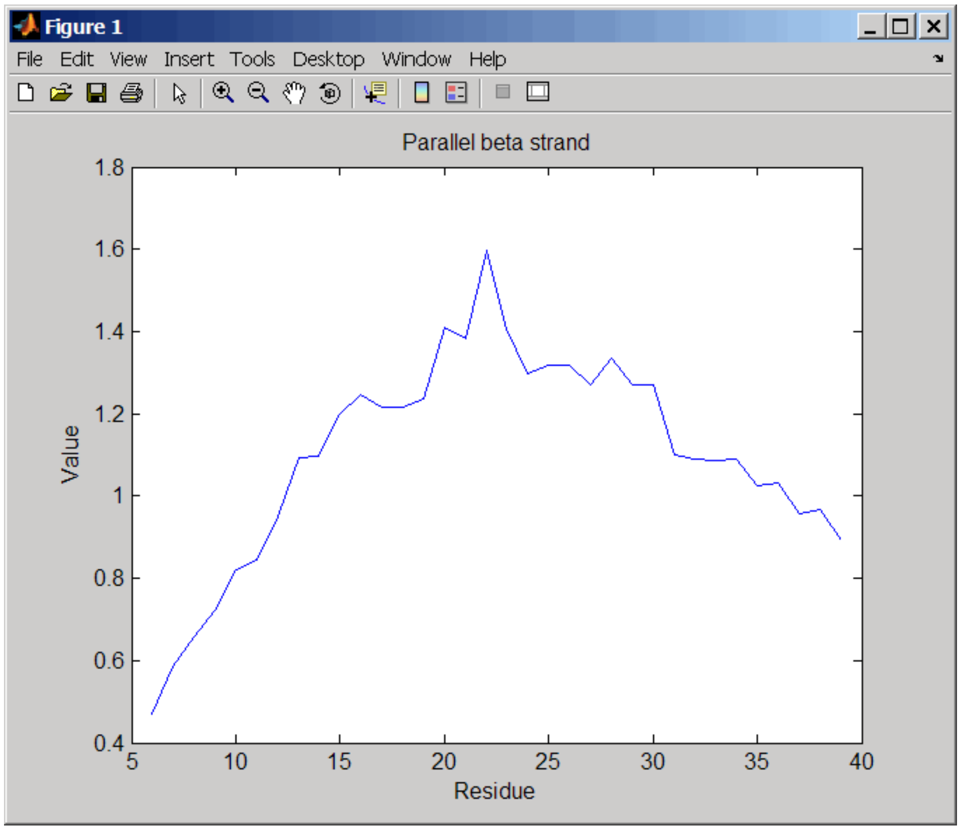
Example 1.43. Plotting Parallel Beta Strand

1 Use the `getgenpept` function to retrieve a protein sequence.

```
s = getgenpept('aad50640');
```

2 Plot the conformational preference for parallel beta strand for the residues in the sequence.

```
proteinpropplot(s,'propertytitle','Parallel beta strand')
```



Version History

Introduced in R2007a

References

[1] Kyte, J., and Doolittle, R.F. (1982). A simple method for displaying the hydropathic character of a protein. J Mol Biol *157(1)*, 105–132.

See Also

`aacount` | `atomiccomp` | `molweight` | `pdbdistplot` | `proteinplot` | `ramachandran` | `seqviewer` | `plotyy`

prune (phytree)

Remove branch nodes from phylogenetic tree

Syntax

```
T2 = prune(T1, Nodes)
T2 = prune(T1, Nodes, 'Mode','Exclusive')
```

Arguments

T1	Phylogenetic object created with the `phytree` constructor function.
Nodes	Nodes to remove from tree.
Mode	Property to control the method of pruning. Enter either `'Inclusive'` or `'Exclusive'`. The default value is `'Inclusive'`.

Description

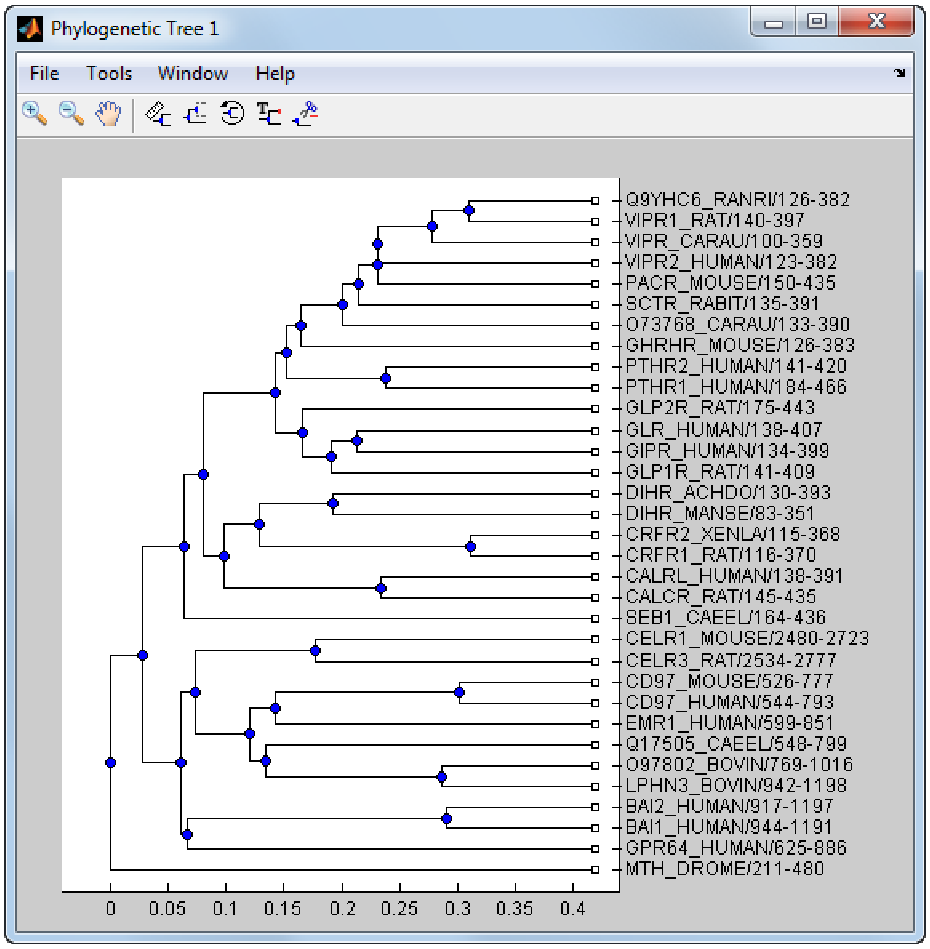
`T2 = prune(T1, Nodes)`removes the nodes listed in the vector *Nodes* from the tree *T1*. `prune` removes any branch or leaf nodes listed in *Nodes* and all their descendants from the tree *T1*, and returns the modified tree *T2*. The parent nodes are connected to the 'brothers' as required. `Nodes` in the tree are labeled as `[1:numLeaves]` for the leaves and as `[numLeaves+1:numLeaves+numBranches]` for the branches. *Nodes* can also be a logical array of size `[numLeaves+numBranches x 1]` indicating the nodes to be removed.

`T2 = prune(T1, Nodes, 'Mode','Exclusive')` changes the `Mode` property for pruning to `'Exclusive'` and removes only the descendants of the nodes listed in the vector *Nodes*. Nodes that do not have a predecessor become leaves in the list *Nodes*. In this case, pruning is the process of reducing a tree by turning some branch nodes into leaf nodes, and removing the leaf nodes under the original branch.

Examples

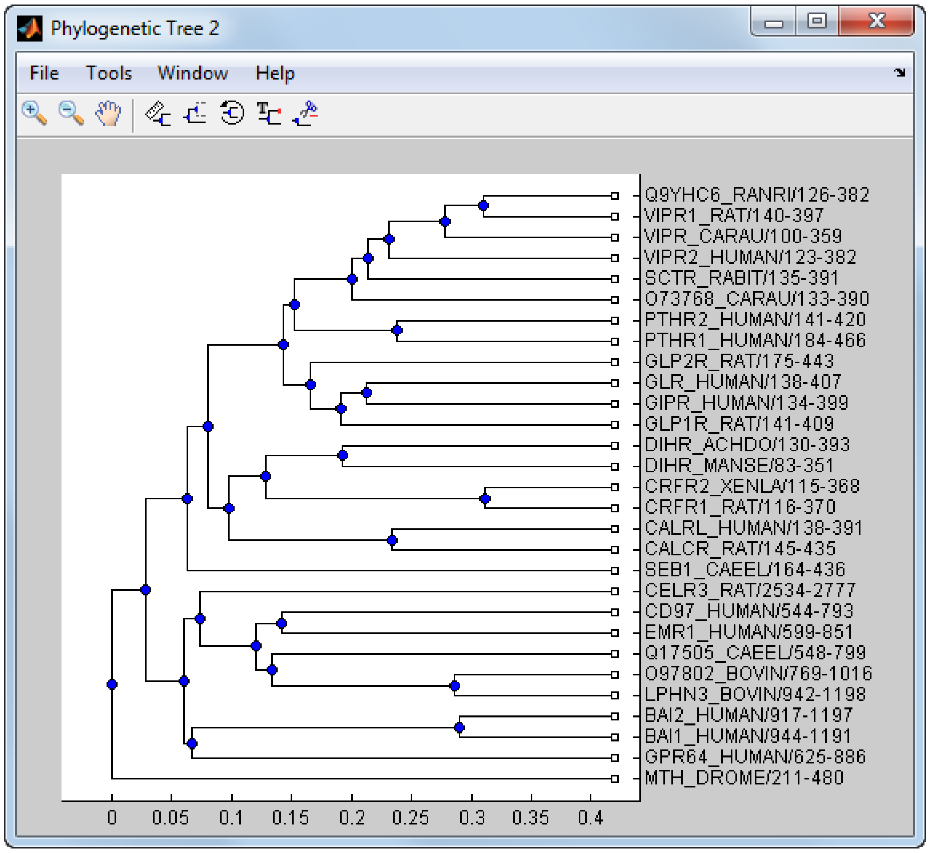
Load a phylogenetic tree created from a protein family

```
tr = phytreeread('pf00002.tree');
view(tr)
```



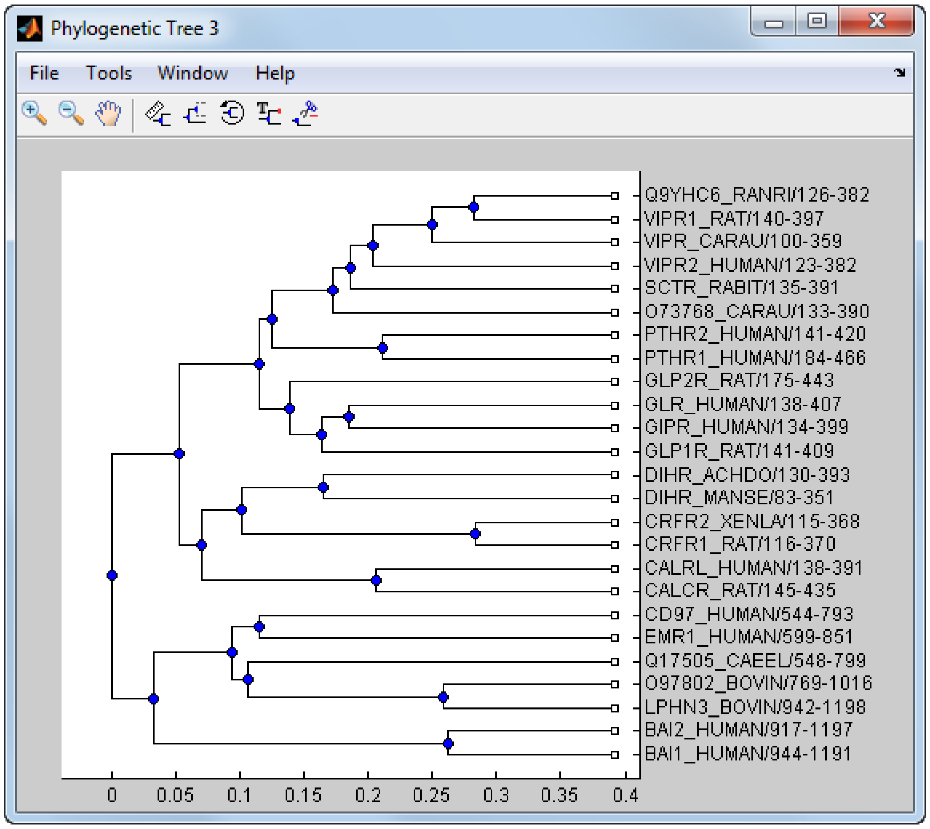
Remove all the 'mouse' proteins

```
ind = getbyname(tr,'mouse');
tr = prune(tr,ind);
view(tr)
```



Remove potential outliers in the tree

```
[sel,sel_leaves] = select(tr,'criteria','distance',...
                            'threshold',.3,...
                            'reference','leaves',...
                            'exclude','leaves',...
                            'propagate','toleaves');
tr = prune(tr,~sel_leaves)
view(tr)
```



Version History

Introduced before R2006a

See Also

`phytree` | `phytreeviewer` | `select` | `get`

Topics

phytree object on page 1-1449

pubMedID

Class: `bioma.ExpressionSet`
Package: `bioma`

Retrieve or set PubMed IDs in ExpressionSet object

Syntax

```
PMIDs = pubMedID(ESObj)
NewESObj = pubMedID(ESObj, NewPMIDs)
```

Description

`PMIDs = pubMedID(ESObj)` returns a character vector or cell array of character vectors containing the PubMed IDs from a MIAME object in an ExpressionSet object.

`NewESObj = pubMedID(ESObj, NewPMIDs)` replaces the PubMed IDs in the MIAME object in *`ESObj`*, an ExpressionSet object, with *`NewPMIDs`*, a character vector or cell array of character vectors specifying new PubMed IDs, and returns *`NewESObj`*, a new ExpressionSet object.

Input Arguments

`ESObj`

Object of the `bioma.ExpressionSet` class.

Default:

`NewPMIDs`

Character vector or cell array of character vectors containing new PubMed IDs.

Default:

Output Arguments

`PMIDs`

Character vector or cell array of character vectors containing the PubMed IDs from a MIAME object in an ExpressionSet object.

`NewESObj`

Object of the `bioma.ExpressionSet` class, returned after replacing the PubMed IDs.

Examples

Construct an ExpressionSet object, `ESObj`, as described in the “Examples” on page 1-0 section of the `bioma.ExpressionSet` class reference page. Retrieve the PubMed identifiers stored in the MIAME object stored in the ExpressionSet object:

```
% Retrieve PubMed IDs from the MIAME object
PMIDs = pubMedID(ESObj)
```

See Also

`bioma.ExpressionSet` | `bioma.data.MIAME`

Topics

“Managing Gene Expression Data in Objects”

External Websites

https://pubmed.ncbi.nlm.nih.gov/

quantilenorm

Quantile normalization over multiple arrays

Syntax

NormData = `quantilenorm(`*Data*`)`
NormData = `quantilenorm(...,'MEDIAN', true)`
NormData = `quantilenorm(...,'DISPLAY', true)`

Description

NormData = `quantilenorm(`*Data*`)`, where the columns of *Data* correspond to separate chips, normalizes the distributions of the values in each column.

Note If *Data* contains NaN values, then *NormData* will also contain NaN values at the corresponding positions.

NormData = `quantilenorm(...,'MEDIAN', true)` takes the median of the ranked values instead of the mean.

NormData = `quantilenorm(...,'DISPLAY', true)` plots the distributions of the columns and of the normalized data.

Examples

```
load yeastdata
normYeastValues = quantilenorm(yeastvalues,'display',1);
```

Version History

Introduced before R2006a

See Also

`affygcrma` | `affyrma` | `malowess` | `manorm` | `rmabackadj` | `rmasummary`

ramachandran

Draw Ramachandran plot for Protein Data Bank (PDB) data

Syntax

```
ramachandran(PDBid)
ramachandran(File)
ramachandran(PDBStruct)
RamaStruct = ramachandran(...)

ramachandran(..., 'Chain', ChainValue, ...)
ramachandran(..., 'Plot', PlotValue, ...)
ramachandran(..., 'Model', ModelValue, ...)
ramachandran(..., 'Glycine', GlycineValue, ...)
ramachandran(..., 'Regions', RegionsValue, ...)
ramachandran(..., 'RegionDef', RegionDefValue, ...)
```

Input Arguments

PDBid	Character vector or string specifying a unique identifier for a protein structure record in the PDB database. **Note** Each structure in the PDB database is represented by a four-character alphanumeric identifier. For example, `4hhb` is the identifier for hemoglobin.
File	Character vector or string specifying a file name or a path and file name. The referenced file is a Protein Data Bank (PDB)-formatted file. If you specify only a file name, that file must be on the MATLAB search path or in the MATLAB Current Directory.
PDBStruct	MATLAB structure containing PDB-formatted data, such as returned by `getpdb` or `pdbread`.
ChainValue	Character vector, string, string vector, or cell array of character vectors that specifies the chain(s) to compute the torsion angles for and plot. Choices are: • `'All'` (default) — Torsion angles for all chains are computed and plotted. • A character vector or string specifying the chain ID, which is case sensitive. • A cell array of character vectors or string vector specifying chain IDs, which are case sensitive.

PlotValue	Character vector or string specifying how to plot chains. Choices are: • `'None'` — Plots nothing. • `'Separate'` — Plots torsion angles for all specified chains in separate plots. • `'Combined'` (default) — Plots torsion angles for all specified chains in one combined plot.
ModelValue	Integer that specifies the structure model to consider. Default is `1`.
GlycineValue	Controls the highlighting of glycine residues with a circle in the plot. Choices are `true` or `false` (default).
RegionsValue	Controls the drawing of Ramachandran reference regions in the plot. Choices are `true` or `false` (default). The default regions are core right-handed alpha, core beta, core left-handed alpha, and allowed, with the core regions corresponding to data points of preferred values of psi/phi angle pairs, and the allowed regions corresponding to possible, but disfavored values of psi/phi angle pairs, based on simple energy considerations. The boundaries of these default regions are based on the calculations by Morris et al., 1992. **Note** If using the default colormap, red = right-handed core alpha, core beta, and core left-handed alpha, while yellow = allowed.
RegionDefValue	MATLAB structure or array of structures (if specifying multiple regions) containing information (name, color, and boundaries) for custom reference regions in a Ramachandran plot. Each structure must contain the following fields: • `Name` — Character vector or string specifying a name for the region. • `Color` — Character vector or string or three-element numeric vector of RGB values specifying a color for the region in the plot. • `Patch` — A 2-by-N matrix of values, the first row containing torsion angle phi (Φ) values, and the second row containing torsion angle psi (Ψ) values. When psi/phi angle pairs are plotted, the data points specify boundaries for the region. N is the number of data points needed to define the region. **Tip** If you specify custom reference regions in which a smaller region is contained or covered by a larger region, list the structure for the smaller region first in the array so that it is plotted last and visible in the plot.

Output Arguments

RamaStruct	MATLAB structure or array of structures (if protein contains multiple chains). Each structure contains the following fields: • `Angles` • `ResidueNum` • `ResidueName` • `Chain` • `HPoints` For descriptions of the fields, see the following table.

Description

A Ramachandran plot is a plot of the torsion angle phi, Φ, (torsion angle between the `C-N-CA-C` atoms) versus the torsion angle psi, Ψ, (torsion angle between the `N-CA-C-N` atoms) for each residue of a protein sequence.

`ramachandran(`*`PDBid`*`)` generates the Ramachandran plot for the protein specified by the PDB database identifier *`PDBid`*.

`ramachandran(`*`File`*`)` generates the Ramachandran plot for the protein specified by *`File`*, a PDB-formatted file.

`ramachandran(`*`PDBStruct`*`)` generates the Ramachandran plot for the protein stored in *`PDBStruct`*, a MATLAB structure containing PDB-formatted data, such as returned by `getpdb` or `pdbread`.

`RamaStruct` `= ramachandran(...)` returns a MATLAB structure or array of structures (if protein contains multiple chains). Each structure contains the following fields.

Field	Description
`Angles`	Three-column matrix containing the torsion angles phi (Φ), psi (Ψ), and omega (ω) for each residue in the sequence, ordered by residue sequence number. The number of rows in the matrix is equal to the number of rows in the `ResidueNum` column vector, which can be used to determine which residue corresponds to each row in the `Angles` matrix. **Note** The `Angles` matrix contains a row for each number in the range of residue sequence numbers, including residue sequence numbers missing from the PDB file. Rows corresponding to residue sequence numbers missing from the PDB file contain the value `NaN`.

Field	Description
`ResidueNum`	Column vector containing the residue sequence numbers from the PDB file. **Note** The `ResidueNum` vector starts with one of the following: • The lowest residue sequence number (if the lowest residue sequence number is negative or zero) • The number `1` (if the lowest residue sequence number is positive) The `ResidueNum` vector ends with the highest residue sequence number and includes all numbers in the range, including residue sequence numbers missing from the PDB file. The angles listed in the `Angles` matrix are in the same order as the residue sequence numbers in the `ResidueNum` vector. Therefore, you can use the `ResidueNum` vector to determine which residue corresponds to each row in the `Angles` matrix.
`ResidueName`	Column vector containing the residue names for the protein.
`Chain`	A character vector or string specifying the chains in the protein.
`HPoints`	Handle to the data points in the plot.

`ramachandran(..., '`*`PropertyName`*`', `*`PropertyValue`*`, ...)` calls `ramachandran` with optional properties that use property name/property value pairs. You can specify one or more properties in any order. Each *`PropertyName`* must be enclosed in single quotation marks and is case insensitive. These property name/property value pairs are as follows:

`ramachandran(..., 'Chain', `*`ChainValue`*`, ...)` specifies the chain(s) to compute the torsion angles for and plot. Choices are:

- `'All'` (default) — Torsion angles for all chains are computed and plotted.
- A character vector or string specifying the chain ID, which is case sensitive.
- A cell array of character vectors or string vector specifying chain IDs, which are case sensitive.

`ramachandran(..., 'Plot', `*`PlotValue`*`, ...)` specifies how to plot chains. Choices are:

- `'None'` — Plots nothing.
- `'Separate'` — Plots torsion angles for all specified chains in separate plots.
- `'Combined'` (default) — Plots torsion angles for all specified chains in one combined plot.

`ramachandran(..., 'Model', `*`ModelValue`*`, ...)` specifies the structure model to consider. Default is `1`.

`ramachandran(..., 'Glycine', `*`GlycineValue`*`, ...)` controls the highlighting of glycine residues with a circle in the plot. Choices are `true` or `false` (default).

`ramachandran(..., 'Regions', `*`RegionsValue`*`, ...)` controls the drawing of Ramachandran reference regions in the plot. Choices are `true` or `false` (default).

The default regions are core right-handed alpha, core beta, core left-handed alpha, and allowed, with the core regions corresponding to data points of preferred values of psi/phi angle pairs, and the allowed regions corresponding to possible, but disfavored values of psi/phi angle pairs, based on simple energy considerations. The boundaries of these default regions are based on the calculations by Morris et al., 1992.

Note If using the default colormap, then red = core right-handed alpha, core beta, and core left-handed alpha, while yellow = allowed.

`ramachandran(..., 'RegionDef', `*`RegionDefValue`*`, ...)` specifies information (name, color, and boundary) for custom reference regions in a Ramachandran plot. *`RegionDefValue`* is a MATLAB structure or array of structures containing the following fields:

- `Name` — Character vector or string specifying a name for the region.
- `Color` — Character vector or string or three-element numeric vector of RGB values specifying a color for the region in the plot.
- `Patch` — A 2-by-N matrix of values, the first row containing torsion angle phi (Φ) values, and the second row containing torsion angle psi (Ψ) values. When psi/phi angle pairs are plotted, the data points specify a boundary for the region. N is the number of data points needed to define the region.

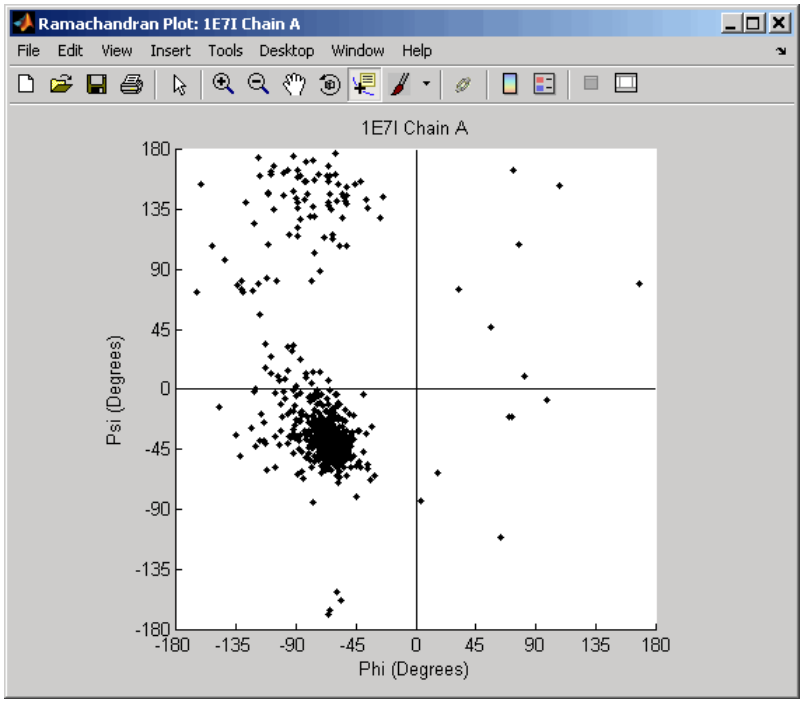
Tip If you specify custom reference regions in which a smaller region is contained or covered by a larger region, list the structure for the smaller region first in the array so that it is plotted last and visible in the plot.

Examples

Example 1.44. Drawing a Ramachandran Plot

Draw the Ramachandran plot for the human serum albumin complexed with octadecanoic acid, which has a PDB database identifier of `1E7I`.

```
ramachandran('1E7I')
```



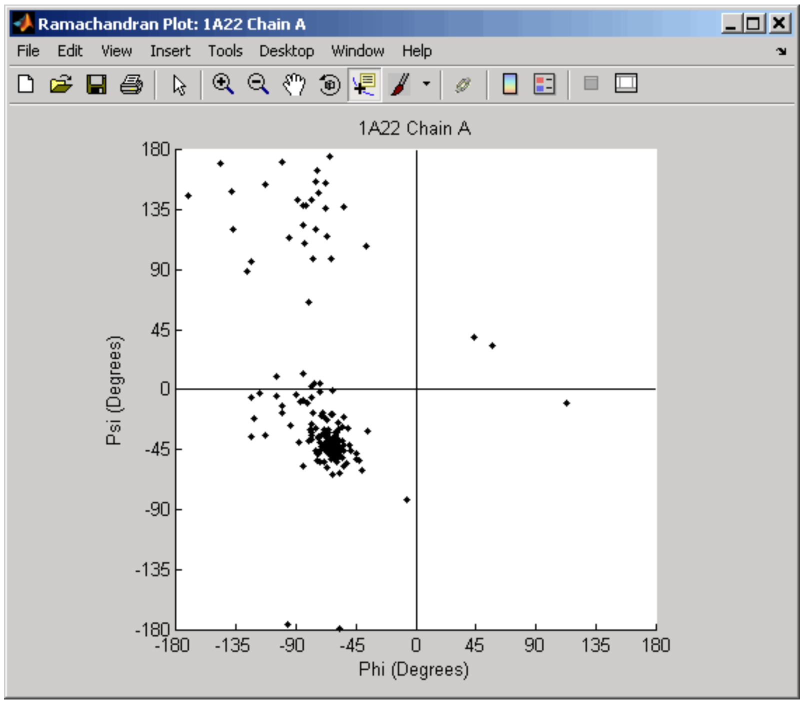
Example 1.45. Drawing a Ramachandran Plot for a Specific Chain

1 Use the `getpdb` function to retrieve protein structure data for the human growth hormone from the PDB database, and save the information to a file.

```
getpdb('1a22','ToFile','1a22.pdb');
```

2 Compute the torsion angles and draw the Ramachandran plot for chain A of the human growth hormone, represented in the pdb file, `1a22.pdb`.

```
ChainA1a22Struct = ramachandran('1a22.pdb','chain','A')

ChainA1a22Struct = 

         Angles: [191x3 double]
     ResidueNum: [191x1 double]
    ResidueName: {191x1 cell}
          Chain: 'A'
        HPoints: 370.0012
```



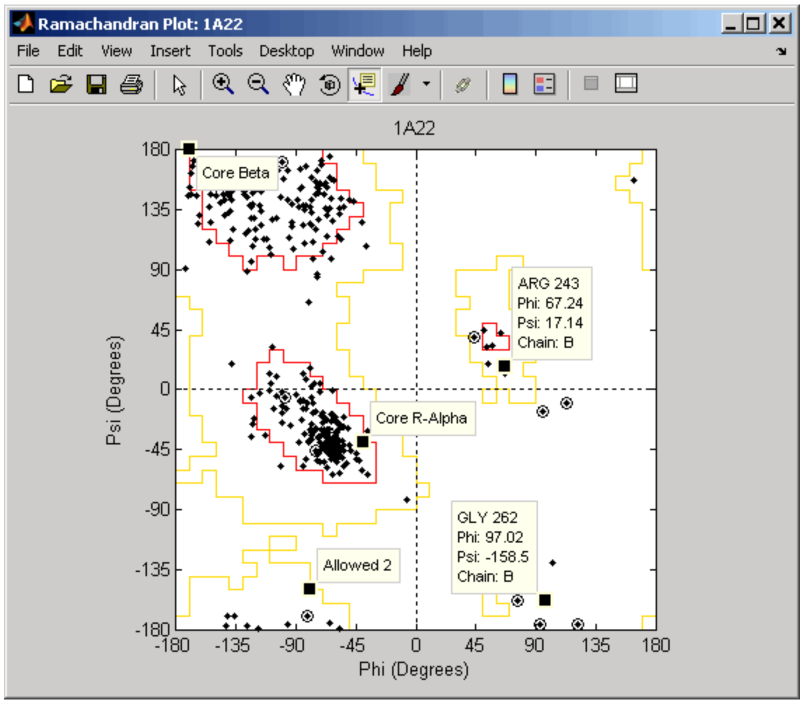
Example 1.46. Drawing Ramachandran Plots with Highlighted Glycine Residues and Ramachandran Regions

1. Use the `getpdb` function to retrieve protein structure data for the human growth hormone from the PDB database, and store the information in a structure.

   ```
   Struct1a22 = getpdb('1a22');
   ```

2. Draw a combined Ramachandran plot for all chains of the human growth hormone, represented in the pdb structure, `1a22Struct`. Highlight the glycine residues (with a circle), and draw the reference Ramachandran regions in the plot.

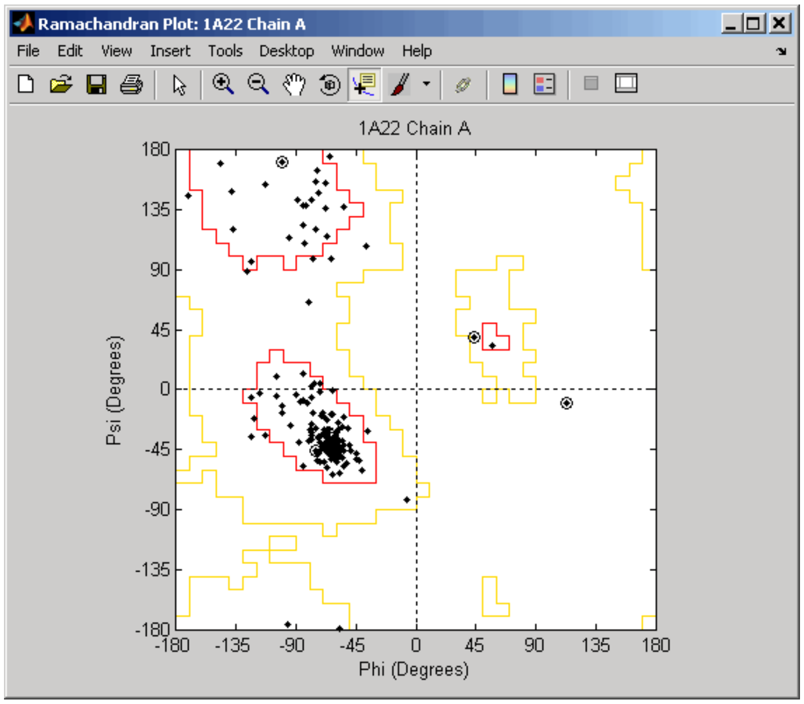
   ```
   ramachandran(Struct1a22,'glycine',true,'regions',true);
   ```

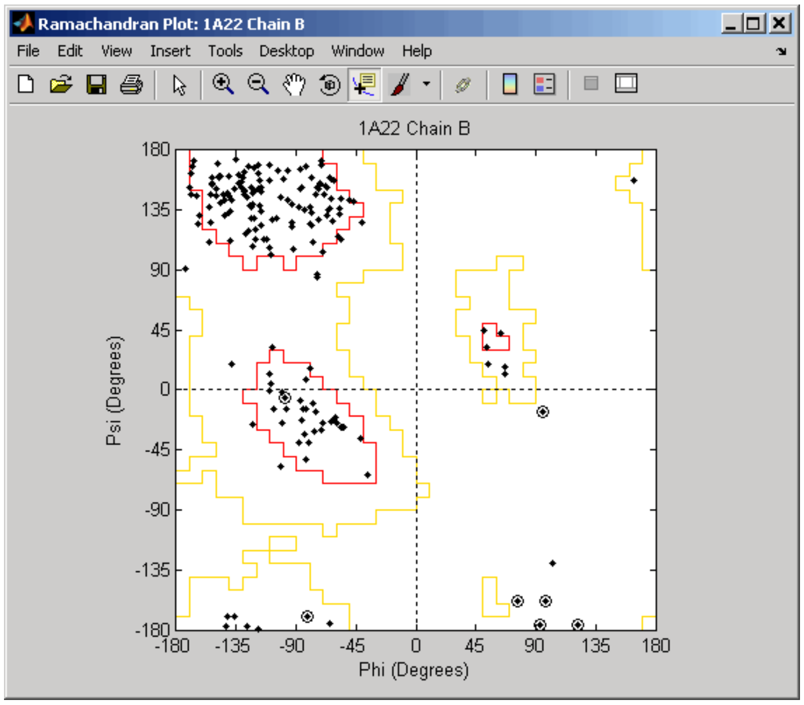


Tip Click a data point to display a data tip with information about the residue. Click a region to display a data tip defining the region. Press and hold the **Alt** key to display multiple data tips.

3 Draw a separate Ramachandran plot for each chain of the human growth hormone, represented in the pdb structure, `1a22Struct`. Highlight the glycine residues (with a circle) and draw the reference Ramachandran regions in the plot.

```
ramachandran(Struct1a22,'plot','separate','chain','all',...
             'glycine',true,'regions',true)
```





Example 1.47. Writing a Tab-Delimited Report File from a Ramachandran Structure

1 Create an array of two structures containing torsion angles for chains A and D in the Calcium/Calmodulin-dependent protein kinase, which has a PDB database identifier of `1hkx`.

```
a = ramachandran('1hkx', 'chain', {'A', 'D'})
```

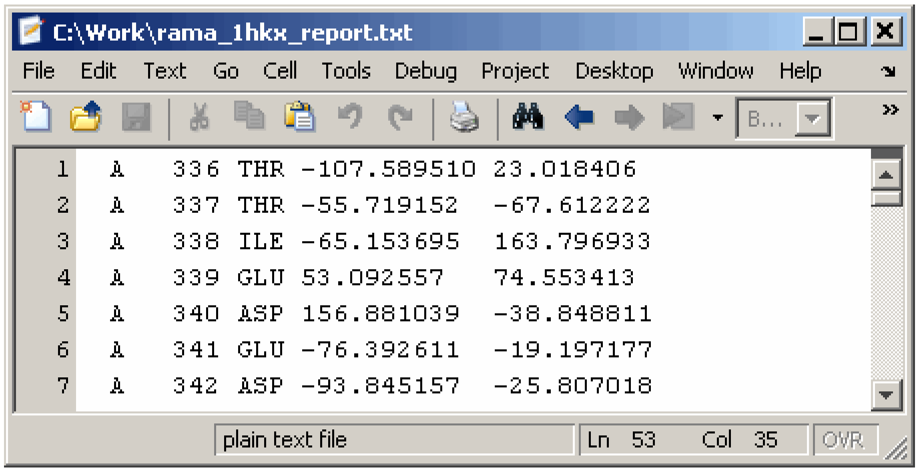
```
a = 

1x2 struct array with fields:
    Angles
    ResidueNum
    ResidueName
    Chain
    HPoints
```

2 Write a tab-delimited report file containing torsion angles phi (Φ) and psi (Ψ) for chains A and D in the Calcium/Calmodulin-dependent protein kinase.

```
fid = fopen('rama_1hkx_report.txt', 'wt');

for c = 1:numel(a)
    for i = 1:length(a(c).Angles)
        if ~all(isnan(a(c).Angles(i,:)))
            fprintf(fid,'%s\t%d\t%s\t%f\t%f\n', a(c).Chain, ...
                a(c).ResidueNum(i), a(c).ResidueName{i}, ...
                a(c).Angles(i,1:2));
        end
    end
end

fclose(fid);
```

3 View the file you created in the MATLAB Editor.

```
edit rama_1hkx_report.txt
```


C:\Work\rama_1hkx_report.txt
File Edit Text Go Cell Tools Debug Project Desktop Window Help
B...
plain text file
Ln 53 Col 35
OVR

```
A    336 THR -107.589510 23.018406
A    337 THR -55.719152  -67.612222
A    338 ILE -65.153695  163.796933
A    339 GLU 53.092557   74.553413
A    340 ASP 156.881039  -38.848811
A    341 GLU -76.392611  -19.197177
A    342 ASP -93.845157  -25.807018
```

Version History

Introduced before R2006a

References

[1] Morris, A.L., MacArthur, M.W., Hutchinson, E.G., and Thornton, J.M. (1992). Stereochemical Quality of Protein Structure Coordinates. PROTEINS: Structure, Function, and Genetics *12*, 345–364.

See Also

getpdb | pdbdistplot | pdbread | proteinpropplot

randfeatures

Generate randomized subset of features

Syntax

```
[IDX, Z] = randfeatures(X, Group, 'PropertyName', PropertyValue...)
randfeatures(..., 'Classifier', C)
randfeatures(..., 'ClassOptions', CO)
randfeatures(..., 'PerformanceThreshold', PT)
randfeatures(..., 'ConfidenceThreshold', CT)
randfeatures(..., 'SubsetSize', SS)
randfeatures(..., 'PoolSize', PS)
randfeatures(..., 'NumberOfIndices', N)
randfeatures(..., 'CrossNorm', CN)
randfeatures(..., 'Verbose', VerboseValue)
```

Description

`[IDX, Z] = randfeatures(X, Group, 'PropertyName', PropertyValue...)` performs a randomized subset feature search reinforced by classification. `randfeatures` randomly generates subsets of features used to classify the samples. Every subset is evaluated with the apparent error. Only the best subsets are kept, and they are joined into a single final pool. The cardinality for every feature in the pool gives the measurement of the significance.

`X` contains the training samples. Every column of `X` is an observed vector. `Group` contains the class labels. `Group` can be a numeric vector, a cell array of character vectors or string vector; `numel(Group)` must be the same as the number of columns in `X`, and `numel(unique(Group))` must be greater than or equal to `2`. `Z` is the classification significance for every feature. `IDX` contains the indices after sorting `Z`; i.e., the first one points to the most significant feature.

`randfeatures(..., 'Classifier', C)` sets the classifier. Options are

```
'da'   (default)  Discriminant analysis
'knn'             K nearest neighbors
```

`randfeatures(..., 'ClassOptions', CO)` is a cell with extra options for the selected classifier. When you specify the discriminant analysis model (`'da'`) as a classifier, `randfeatures` uses the `classify` function with its default parameters. For the KNN classifier, `randfeatures` uses `fitcknn` with the following default options. `{'Distance','correlation','NumNeighbors',5}`.

`randfeatures(..., 'PerformanceThreshold', PT)` sets the correct classification threshold used to pick the subsets included in the final pool. For the `'da'` model, the default is `0.8`. For the `'knn'` model, the default is `0.7`.

`randfeatures(..., 'ConfidenceThreshold', CT)` uses the posterior probability of the discriminant analysis to invalidate classified subvectors with low confidence. When using the `'da'` model, the default is `0.95.^(number of classes)`. When using the `'knn'` model, the default is 1, meaning any classified subvector must have all *k* neighbors classified to the same class in order to be kept in the pool.

`randfeatures(..., 'SubsetSize', SS)` sets the number of features considered in every subset. Default is `20`.

`randfeatures(..., 'PoolSize', PS)` sets the targeted number of accepted subsets for the final pool. Default is `1000`.

`randfeatures(..., 'NumberOfIndices', N)` sets the number of output indices in `IDX`. Default is the same as the number of features.

`randfeatures(..., 'CrossNorm', CN)` applies independent normalization across the observations for every feature. Cross-normalization ensures comparability among different features, although it is not always necessary because the selected classifier properties might already account for this. Options are

```
'none' (default)  Intensities are not cross-normalized.
'meanvar'         x_new = (x - mean(x))/std(x)
'softmax'         x_new = (1+exp((mean(x)-x)/std(x)))^-1
'minmax'          x_new = (x - min(x))/(max(x)-min(x))
```

`randfeatures(..., 'Verbose', VerboseValue)`, when `Verbose` is `true`, turns off verbosity. Default is `true`.

Examples

Find a reduced set of genes that is sufficient for classification of all the cancer types in the t-matrix NCI60 data set. Load sample data.

```
load NCI60tmatrix
```

Select features.

```
I = randfeatures(X,GROUP,'SubsetSize',15,'Classifier','da');
```

Test features with a linear discriminant classifier.

```
C = classify(X(I(1:25),:)',X(I(1:25),:)',GROUP);
cp = classperf(GROUP,C);
cp.CorrectRate
```

```
ans =

     1
```

Version History

Introduced before R2006a

References

[1] Li, L., Umbach, D.M., Terry, P., and Taylor, J.A. (2003). Application of the GA/KNN method to SELDI proteomics data. PNAS. 20, 1638-1640.

[2] Liu, H., Motoda, H. (1998). Feature Selection for Knowledge Discovery and Data Mining, Kluwer Academic Publishers.

[3] Ross, D.T. et.al. (2000). Systematic Variation in Gene Expression Patterns in Human Cancer Cell Lines. Nature Genetics. 24 (3), 227-235.

See Also

`classperf` | `crossvalind` | `rankfeatures` | `classify` | `sequentialfs`

randseq

Generate random sequence from finite alphabet

Syntax

```
Seq = randseq(SeqLength)

Seq = randseq(SeqLength, ...'Alphabet', AlphabetValue, ...)
Seq = randseq(SeqLength, ...'Weights', WeightsValue, ...)
Seq = randseq(SeqLength, ...'FromStructure', FromStructureValue, ...)
Seq = randseq(SeqLength, ...'Case', CaseValue, ...)
Seq = randseq(SeqLength, ...'DataType', DataTypeValue, ...)
```

Arguments

`SeqLength`	Integer that specifies the number of nucleotides or amino acids in the random sequence .
`AlphabetValue`	Character vector or string that specifies the alphabet for the sequence. Choices are `'dna'`(default), `'rna'`, or `'amino'`.
`WeightsValue`	Property to specify a weighted random sequence.
`FromStructureValue`	Property to specify a weighted random sequence using output structures from the functions from `basecount`, `dimercount`, `codoncount`, or `aacount`.
`CaseValue`	Character vector or string that specifies the case of letters in a sequence when `Alphabet` is `'char'`. Choices are`'upper'` (default) or `'lower'`.
`DataTypeValue`	Character vector or string that specifies the data type for a sequence. Choices are `'char'`(default) for letter sequences, and `'uint8'` or `'double'` for numeric sequences. Creates a sequence as an array of *`DataType`*.

Description

`Seq` `= randseq(`*`SeqLength`*`)` creates a random sequence with a length specified by *`SeqLength`*.

`Seq` `= randseq(`*`SeqLength`*`, ...'`*`PropertyName`*`', `*`PropertyValue`*`, ...)` calls `randseq` with optional properties that use property name/property value pairs. You can specify one or more properties in any order. Each *`PropertyName`* must be enclosed in single quotation marks and is case insensitive. These property name/property value pairs are as follows:

`Seq` `= randseq(`*`SeqLength`*`, ...'Alphabet', `*`AlphabetValue`*`, ...)` generates a sequence from a specific alphabet.

`Seq` `= randseq(`*`SeqLength`*`, ...'Weights', `*`WeightsValue`*`, ...)` creates a weighted random sequence where the `i`th letter of the sequence alphabet is selected with weight `W(i)`. The

weight vector is usually a probability vector or a frequency count vector. Note that the `ith` element of the nucleotide alphabet is given by `int2nt(i)`, and the `ith` element of the amino acid alphabet is given by `int2aa(i)`.

`Seq = randseq(SeqLength, ...'FromStructure', FromStructureValue, ...)` creates a weighted random sequence with weights given by the output structure from `basecount`, `dimercount`, `codoncount`, or `aacount`.

`Seq = randseq(SeqLength, ...'Case', CaseValue, ...)` specifies the case for a letter sequence.

`Seq = randseq(SeqLength, ...'DataType', DataTypeValue, ...)` specifies the data type for the sequence array.

Examples

Generate a random DNA sequence.

```
randseq(20)

ans =
TAGCTGGCCAAGCGAGCTTG
```

Generate a random RNA sequence.

```
randseq(20,'alphabet','rna')

ans =
GCUGCGGCGGUUGUAUCCUG
```

Generate a random protein sequence.

```
randseq(20,'alphabet','amino')

ans =
DYKMCLYEFGMFGHFTGHKK
```

Version History

Introduced before R2006a

See Also

`hmmgenerate` | `randsample` | `rand` | `randperm`

rankfeatures

Rank key features by class separability criteria

Syntax

```
IDX = rankfeatures(X,GROUP)
IDX = rankfeatures(X,GROUP,Name=Value)
[IDX,Z] = rankfeatures(X,GROUP,___)
```

Description

`IDX = rankfeatures(X,GROUP)` ranks the features in `X` using an independent evaluation criterion for binary classification. `X` is a matrix where every column is an observed vector and the number of rows corresponds to the original number of features. `GROUP` contains the class labels. `IDX` is a list of indices to the rows of `X` with the most significant features.

`IDX = rankfeatures(X,GROUP,Name=Value)` uses additional options specified by one or more name-value arguments.

`[IDX,Z] = rankfeatures(X,GROUP,___)` also returns a list of absolute values of the criterion used for every feature.

Examples

Find a reduced set of genes to differentiate breast cancer cells

Find a reduced set of genes that is sufficient for differentiating breast cancer cells from all other types of cancer in the t-matrix NCI60 data set.

Load sample data.

```
load NCI60tmatrix
```

Get a logical index vector to the breast cancer cells.

```
BC = GROUP == 8;
```

Select features.

```
I = rankfeatures(X,BC,NumberOfIndices=12);
```

Test features with a linear discriminant classifier.

```
C = classify(X(I,:)',X(I,:)',double(BC));
cp = classperf(BC,C);
cp.CorrectRate
```

```
ans = 1
```

Use cross-correlation weighting to further reduce the required number of genes.

```
I = rankfeatures(X,BC,'CCWeighting',0.7,'NumberOfIndices',8);
C = classify(X(I,:)',X(I,:)',double(BC));
cp = classperf(BC,C);
cp.CorrectRate

ans = 1
```

Find discriminant peaks of two groups of signals

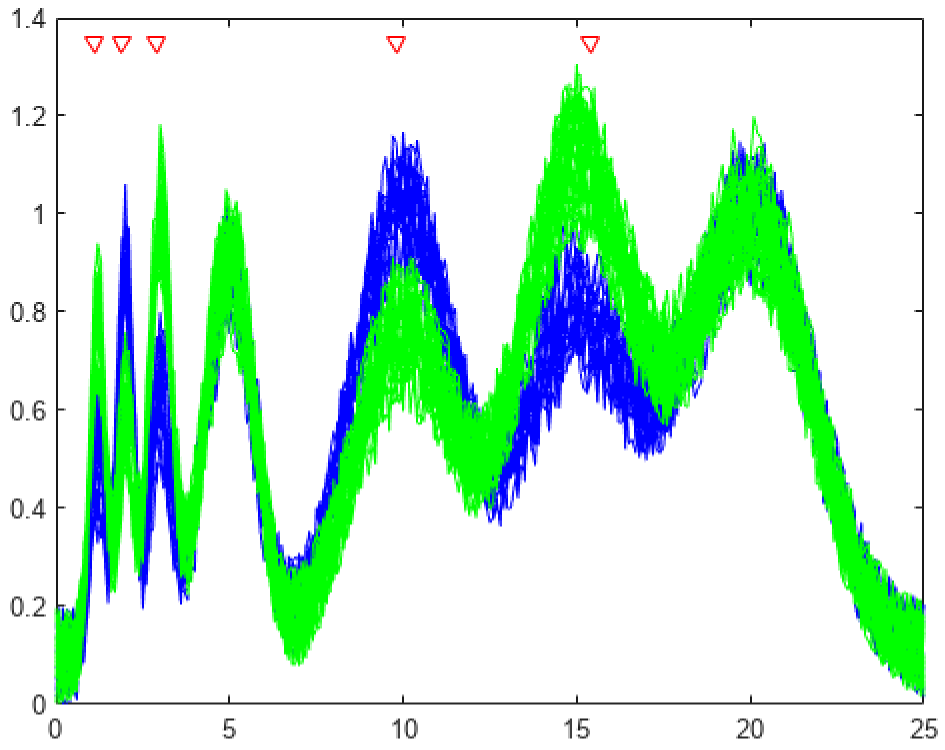
Find the discriminant peaks of two groups of signals with Gaussian pulses modulated by two different sources.

Load data.

```
load GaussianPulses
```

Specify the regional information to outweigh Z-value of features as a function handle. Set the number of output indices to 5.

```
f = rankfeatures(y',grp,NWeighting=@(x) x/10+5,NumberOfIndices=5);
plot(t,y(grp==1,:),'b',t,y(grp==2,:),'g',t(f),1.35,'vr');
```



Input Arguments

X — Sample data
numeric matrix

Sample data, specified as a numeric matrix. Each column is an observed vector, and each row is a feature.

Data Types: `double`

GROUP — Class labels
numeric vector | string vector | cell array of character vectors

Class labels, specified as a numeric vector, string vector, or cell array of character vectors. `numel(GROUP)` is the same as the number of columns in `X`. `GROUP` must have only two unique values. If it contains any `NaN` values, the function ignores the corresponding observation vector in `X`.

Data Types: `double` | `string` | `cell`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: `[idx,x] = rankfeatures(x,groups,Criterion="entrophy",NWeighting=0.2)` specifies to use the relative entropy as the criterion to assess the feature significance and regional information value of 0.2 to outweigh the Z-value of potential features.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `[idx,x] = rankfeatures(x,groups,'Criterion',"entrophy",'NWeighting',0.2)`

Criterion — Criterion to assess significance of feature
`"ttest"` (default) | "entrophy" | `"bhattacharyya"` | `"roc"` | `"wilcoxon"`

Criterion to assess the significance of each feature for separating two labeled groups, specified as one of the following:

- `"ttest"` — Absolute value two-sample t-test with pooled variance estimate.
- `"entropy"` — Relative entropy, also known as Kullback-Leibler distance or divergence.
- `"bhattacharyya"` — Minimum attainable classification error or Chernoff bound.
- `"roc"` — Area between the empirical receiver operating characteristic (ROC) curve and the random classifier slope.
- `"wilcoxon"` — Absolute value of the standardized u-statistic of a two-sample unpaired Wilcoxon test, also known as Mann-Whitney.

Note `"ttest"`, `"entropy"`, and `"bhattacharyya"` assume normal distributed classes while `"roc"` and `"wilcoxon"` are nonparametric tests. All tests are feature independent.

Data Types: `char` | `string`

`CCWeighting` — Correlation information to outweigh Z-value of features
`0` (default) | numeric scalar between `0` and `1`

Correlation information to outweigh the Z-value of potential features, specified as a numeric scalar between `0` and `1`.

The function uses $Z \times (1 - \alpha \times \rho)$ to calculate the weight, where ρ is the average of the absolute values of the cross-correlation coefficient between the candidate feature and all previously selected features. α is the `CCWeighting` value that sets the weighting factor.

By default, α is `0`, and the function does not weight the potential features. A large value of ρ (close to 1) outweighs the significance statistic, meaning that features are highly correlated with the features already picked are less likely to be included in the output list.

Data Types: `double`

`NWeighting` — Regional information to outweigh Z-value of features
`0` (default) | nonnegative scalar | function handle

Regional information to outweigh the Z-value of potential features, specified as a nonnegative scalar or function handle.

The function uses $Z \times \left(1 - e^{-\left(\frac{D}{\beta}\right)^2}\right)$ to calculate the weight, where D is the distance (in rows) between the candidate feature and previously selected features. β is the `NWeighting` value that sets the weighting factor. β must be greater than or equal to `0`.

By default, β is `0`, and the function does not weight the potential features. A small value of D (close to `0`) outweighs the significance statistics of only close features. This means that features that are close to already picked features are less likely to be included in the output list. This option is useful for extracting features from time series with temporal correlation.

β can also be a function of the feature location, specified using `@` or an anonymous function. In both cases `rankfeatures` passes the row position of the feature to the specified function and expects back a value greater than or equal to `0`.

Note You can use `CCWeighting` and `NWeighting` together.

Data Types: `double` | `function_handle`

`NumberOfIndices` — Number of output indices
positive scalar

Number of output indices in `IDX`, specified as a positive scalar. By default, the number of indices is the same as the number of features when α and β are `0`. Otherwise, the number of indices is set to `20`.

Data Types: `double`

`CrossNorm` — Method for independent normalization across observations
`"none"` (default) | `"meanvar"` | `"softmax"` | `"minmax"`

Method for independent normalization across observations for every feature, specified as one of the following:

- `"none"` (default) — No normalization.
- `"meanvar"` — $X_{new} = \frac{X - \mu}{\sigma}$
- `"softmax"` — $X_{new} = \frac{1}{1 + e^{\left(\frac{\mu - X}{\sigma}\right)}}$
- `"minmax"` — $X_{new} = \frac{X - X_{\min}}{X_{\max} - X_{\min}}$

In these equations, μ = `mean(`X`)`, σ = `std(`X`)`, X_{min} = `min(X)`, and X_{max} = `max(X)`.

Cross-normalization ensures comparability among different features although it is not always necessary because the selected criterion might already account for this.

Data Types: `char` | `string`

Output Arguments

`IDX` — List of indices
numeric vector

List of indices to the rows of X with the most significant features, returned as a numeric vector.

`Z` — List of absolute values of criterion for features
numeric vector

List of absolute values of the `Criterion` used for the features, returned as a numeric vector.

Version History

Introduced before R2006a

References

[1] Theodoridis, Sergios, and Konstantinos Koutroumbas. *Pattern Recognition*. San Diego: Academic Press, 1999: 341-342.

[2] Liu, Huan, and Hiroshi Motoda. *Feature Selection for Knowledge Discovery and Data Mining*. Kluwer International Series in Engineering and Computer Science 454. Boston: Kluwer Academic Publishers, 1998.

[3] Ross, Douglas T., Uwe Scherf, Michael B. Eisen, Charles M. Perou, Christian Rees, Paul Spellman, Vishwanath Iyer, et al. “Systematic Variation in Gene Expression Patterns in Human Cancer Cell Lines.” *Nature Genetics* 24, no. 3 (March 2000): 227–35.

See Also

`classperf` | `crossvalind` | `randfeatures` | `classify` | `sequentialfs`

rdivide (DataMatrix)

Right array divide DataMatrix objects

Syntax

```
DMObjNew = rdivide(DMObj1, DMObj2)
DMObjNew = DMObj1 ./ DMObj2
DMObjNew = rdivide(DMObj1, B)
DMObjNew = DMObj1 ./ B
DMObjNew = rdivide(B, DMObj1)
DMObjNew = B ./ DMObj1
```

Input Arguments

DMObj1, *DMObj2*	DataMatrix objects, such as created by `DataMatrix` (object constructor).
B	MATLAB numeric or logical array.

Output Arguments

DMObjNew	DataMatrix object created by right array division.

Description

`DMObjNew = rdivide(DMObj1, DMObj2)` or the equivalent `DMObjNew = DMObj1 ./ DMObj2` performs an element-by-element right array division of the DataMatrix objects *DMObj1* and *DMObj2* and places the results in *DMObjNew*, another DataMatrix object. In other words, `rdivide` divides each element in *DMObj1* by the corresponding element in *DMObj2*. *DMObj1* and *DMObj2* must have the same size (number of rows and columns), unless one is a scalar (1-by-1 DataMatrix object). The size (number of rows and columns), row names, and column names for *DMObjNew* are the same as *DMObj1*, unless *DMObj1* is a scalar; then they are the same as *DMObj2*.

`DMObjNew = rdivide(DMObj1, B)` or the equivalent `DMObjNew = DMObj1 ./ B` performs an element-by-element right array division of the DataMatrix object *DMObj1* and *B*, a numeric or logical array, and places the results in *DMObjNew*, another DataMatrix object. In other words, `rdivide` divides each element in *DMObj1* by the corresponding element in *B*. *DMObj1* and *B* must have the same size (number of rows and columns), unless *B* is a scalar. The size (number of rows and columns), row names, and column names for *DMObjNew* are the same as *DMObj1*.

`DMObjNew = rdivide(B, DMObj1)` or the equivalent `DMObjNew = B ./ DMObj1` performs an element-by-element right array division of *B*, a numeric or logical array, and the DataMatrix object *DMObj1*, and places the results in *DMObjNew*, another DataMatrix object. In other words, `rdivide` divides each element in *B* by the corresponding element in *DMObj1*.*DMObj1* and *B* must have the same size (number of rows and columns), unless *B* is a scalar. The size (number of rows and columns), row names, and column names for *DMObjNew* are the same as *DMObj1*.

Note Arithmetic operations between a scalar DataMatrix object and a nonscalar array are not supported.

MATLAB calls `DMObjNew = rdivide(X, Y)` for the syntax `DMObjNew = X ./ Y` when *X* or *Y* is a DataMatrix object.

Version History

Introduced in R2008b

See Also

`DataMatrix` | `ldivide` | `times`

Topics

DataMatrix object on page 1-734

read

Class: `BioIndexedFile`

Read one or more entries from source file associated with BioIndexedFile object

Syntax

```
Output = read(BioIFobj, Indices)
Output = read(BioIFobj, Key)
```

Description

`Output = read(BioIFobj, Indices)` reads the entries specified by *`Indices`* from the source file associated with *`BioIFobj`*, a BioIndexedFile object. *`Indices`* is a vector of positive integers specifying indices to entries in the source file. The `read` method reads and parses the entries using the function specified by the `Interpreter` property of the BioIndexedFile object. A one-to-one relationship exists between the number and order of elements in *`Indices`* and *`Output`*, even if *`Indices`* has repeated entries. *`Output`* is a structure or an array of structures containing the parsed data returned by the interpreter function.

`Output = read(BioIFobj, Key)` reads the entries specified by *`Key`* from the source file associated with *`BioIFobj`*, a BioIndexedFile object. *`Key`* is a character vector or cell array of character vectors specifying one or more keys to entries in the source file. The `read` method reads and parses the entries using the function specified by the `Interpreter` property of the BioIndexedFile object. If the keys in the source file are not unique, the `read` method reads all entries that match a specified key, all at the position of the key in the *`Key`* cell array. If the keys in the source file are unique, there is a one-to-one relationship between the number and order of elements in *`Key`* and *`Output`*.

Input Arguments

`BioIFobj`

Object of the `BioIndexedFile` class.

Default:

`Indices`

Vector of positive integers specifying indices to entries in the source file associated with *`BioIFobj`*, the BioIndexedFile object. The number of elements in *`Indices`* must be less than or equal to the number of entries in the source file. There is a one-to-one relationship between the number and order of elements in *`Indices`* and *`Output`*, even if *`Indices`* has repeated entries.

Default:

`Key`

Character vector or cell array of character vectors specifying one or more keys in the source file.

Default:

Output Arguments

`Output`

Structure or an array of structures containing the parsed data returned by the interpreter function.

Examples

Construct a BioIndexedFile object to access a table containing cross-references between gene names and gene ontology (GO) terms:

```
% Create variable containing full absolute path of source file
sourcefile = which('yeastgenes.sgd');
% Create a BioIndexedFile object from the source file. Indicate
% the source file is a tab-delimited file where contiguous rows
% with the same key are considered a single entry. Store the
% index file in the Current Folder. Indicate that keys are
% located in column 3 and that header lines are prefaced with !
gene2goObj = BioIndexedFile('mrtab', sourcefile, '.', ...
                            'KeyColumn', 3, 'HeaderPrefix','!')
```

Read the GO term from all entries that are associated with the gene YAT2:

```
% Access entries that have the string YAT2 in their keys
YAT2_entries = getEntryByKey(gene2goObj, 'YAT2');
% Adjust the object interpreter to return only the column
% containing the GO term
gene2goObj.Interpreter = @(x) regexp(x,'GO:\d+','match')
% Parse the entries with a key of YAT2 and return all GO terms
% from those entries
GO_YAT2_entries = read(gene2goObj, 'YAT2')

GO_YAT2_entries =

'GO:0004092'  'GO:0005737'  'GO:0006066'  'GO:0006066'  'GO:0009437'
```

Tips

Before using the `read` method, make sure the `Interpreter` property of the BioIndexedFile object is set appropriately. The `Interpreter` property is a handle to a function that parses the entries in the source file. The interpreter function must accept a character vector of one or more concatenated entries and return a structure or an array of structures containing the interpreted data.

If the BioIndexedFile object was created from a source file with an application-specific format such as `'SAM'`, `'FASTQ'`, or `'FASTA'`, the default `Interpreter` property is a handle to a function appropriate for that file type and typically does not require you to change it. If the BioIndexedFile object was created from a source file with a `'TABLE'`, `'MRTAB'`, or `'FLAT'` format, then the default `Interpreter` property is `[]`, which means the interpreter is an anonymous function in which the output is equivalent to the input.

For information on setting the `Interpreter` property, see `BioIndexedFile` class.

See Also

`BioIndexedFile` | `getSubset`

Topics

"Work with Next-Generation Sequencing Data"

rebasecuts

Find restriction enzymes that cut nucleotide sequence

Syntax

```
[Enzymes, Sites] = rebasecuts(SeqNT)
rebasecuts(SeqNT, Group)
rebasecuts(SeqNT, [Q, R])
rebasecuts(SeqNT, S)
```

Input Arguments

SeqNT	Nucleotide sequence.
Group	Cell array of character vectors or string vector representing the names of valid restriction enzymes.
Q, *R*	Base positions that limit the search to all sites between base *Q* and base *R*.
S	Base position that limits the search to all sites after base *S*.

Output Arguments

Enzymes	Cell array of character vectors containing the names of restriction enzymes from REBASE®, the Restriction Enzyme Database.
Sites	Vector of cut sites identified with the base position number before every cut.

Description

`[Enzymes, Sites] = rebasecuts(SeqNT)` finds all the restriction enzymes that cut *SeqNT*, a nucleotide sequence.

`rebasecuts(SeqNT, Group)` limits the search to *Group*, a list of enzymes.

`rebasecuts(SeqNT, [Q, R])` limits the search to those enzymes that cut after the base position specified by *Q* and before the base position specified by *R*.

`rebasecuts(SeqNT, S)` limits the search to those enzymes that cut just after the base position specified by *S*.

REBASE, the Restriction Enzyme Database, is a collection of information about restriction enzymes and related proteins. For more information about REBASE, see:

`http://rebase.neb.com/rebase/rebase.html`

Examples

1 Create a nucleotide sequence.

```
seq = 'AGAGGGGTACGCGCTCTGAAAAGCGGGAACCTCGTGGCGCTTTATTAA';
```

2 Find all possible enzymes and cleavage sites in the sequence.

```
 [enzymes, sites] = rebasecuts(seq)
```

3 Find where restriction enzymes `CfoI` and `Tru9I` cut the sequence.

```
[enzymes, sites] = rebasecuts(seq, {'CfoI','Tru9I'})

enzymes = 

    'CfoI'
    'CfoI'
    'Tru9I'

sites =

    13
    39
    45
```

4 Find all possible enzymes that cut after base 7.

```
enzymes  = rebasecuts(seq, 7)

enzymes = 

    'Csp6I'
    'CviQI'
    'RsaNI'
```

5 Find all possible enzymes that cut between bases 11 and 37.

```
enzymes  = rebasecuts(seq, [11 37])

enzymes = 

    'AccII'
    'AspLEI'
    'BmiI'
    'Bsh1236I'
    'BspFNI'
    'BspLI'
    'BstFNI'
    'BstHHI'
    'BstUI'
    'CfoI'
    'FnuDII'
    'GlaI'
    'HhaI'
    'Hin6I'
    'HinP1I'
    'Hpy188I'
    'HspAI'
    'MvnI'
    'NlaIV'
    'PspN4I'
    'SetI'
```

Version History

Introduced before R2006a

References

[1] Roberts, R.J., Vincze, T., Posfai, J., and Macelis, D. (2007). REBASE—enzymes and genes for DNA restriction and modification. Nucl. Acids Res. *35*, D269–D270.

[2] Official REBASE Web site: `http://rebase.neb.com`.

See Also

`cleave` | `cleavelookup` | `restrict` | `seq2regexp` | `regexp`

redbluecmap

Create red and blue colormap

Syntax

```
redbluecmap(length)
redbluecmap
```

Description

`redbluecmap(length)` returns a `length`-by-`3` matrix containing a red and blue diverging color palette. Low values are dark blue, values in the center of the map are white, and high values are dark red. `length` must be a positive integer between 3 and 11.

`redbluecmap` returns an `11`-by-`3` matrix representing 11 colors.

Examples

Perform Hierarchical Clustering on Gene Expression Data

Load microarray data containing gene expression levels of *Saccharomyces cerevisiae* (yeast) during the metabolic shift from fermentation to respiration [1].

```
load filteredyeastdata
```

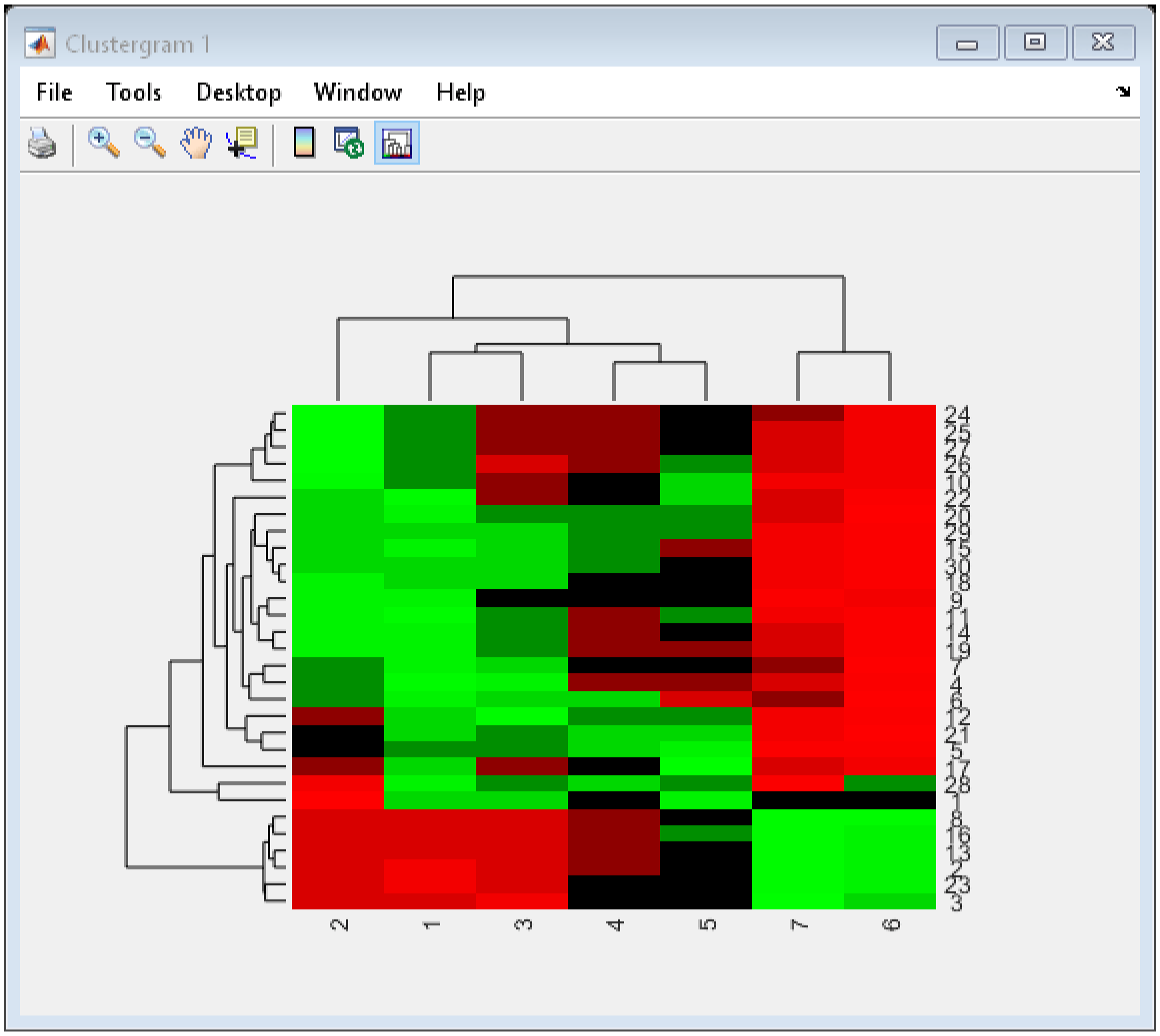
This MAT file includes three variables, which are added to the MATLAB® workspace:

- `yeastvalues` - A matrix of gene expression data from *Saccharomyces* -_cerevisiae_ during the metabolic shift from fermentation to respiration - `genes` - A cell array of GenBank® accession numbers for labeling the rows in `yeastvalues` - `times` - A vector of time values for labeling the columns in `yeastvalues`

Create a clustergram object to display the heat map from the gene expression data in the first 30 rows of the `yeastvalues` matrix and standardize along the rows of data.

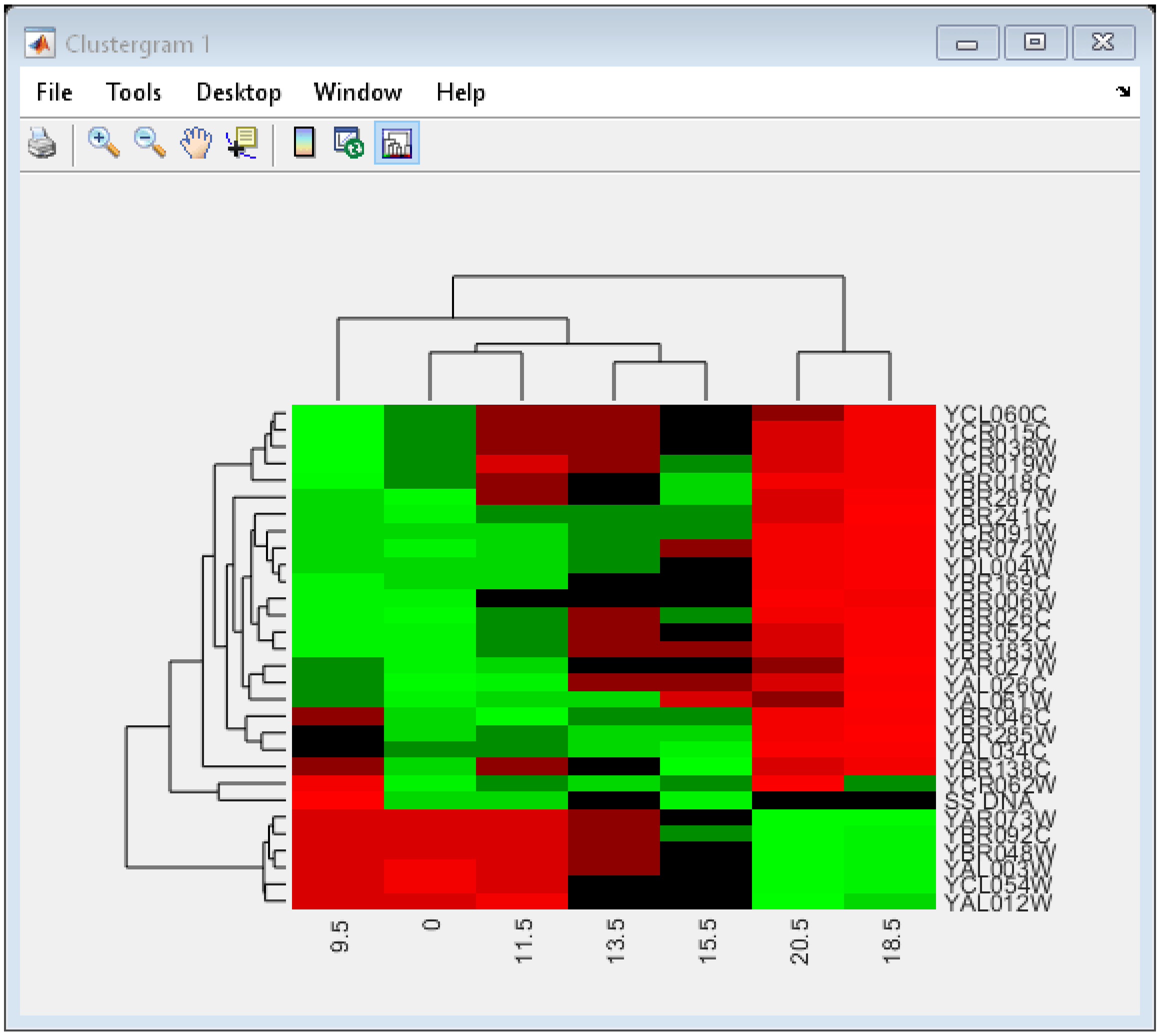
```
cgo = clustergram(yeastvalues(1:30,:),'Standardize','Row')
```

```
Clustergram object with 30 rows of nodes and 7 columns of nodes.
```



Use the `set` method and the `genes` and `times` vectors to add meaningful row and column labels to the clustergram.

```
set(cgo,'RowLabels',genes(1:30),'ColumnLabels',times)
```



Add a color bar to the clustergram by clicking the `Insert Colorbar` button on the toolbar.

View a data tip containing the intensity value, row label, and column label for a specific area of the heat map by clicking the `Data Cursor` button on the toolbar, then clicking an area in the heat map. To delete this data tip, right-click it, then select `Delete Current Datatip`.

Display intensity values for each area of the heat map by clicking the **Annotate** button on the toolbar. Click the **Annotate** button again to remove the intensity values.

```
Tip: If the amount of data is large enough, the cells within the clustergram
are too small to display the intensity annotations. Zoom in to see the
intensity annotations.
```

Remove the dendrogram tree diagrams from the figure by clicking the **Show Dendrogram** button on the toolbar. Click it again to display the dendrograms.

Use the `get` method to display the properties of the clustergram object, `cgo`.

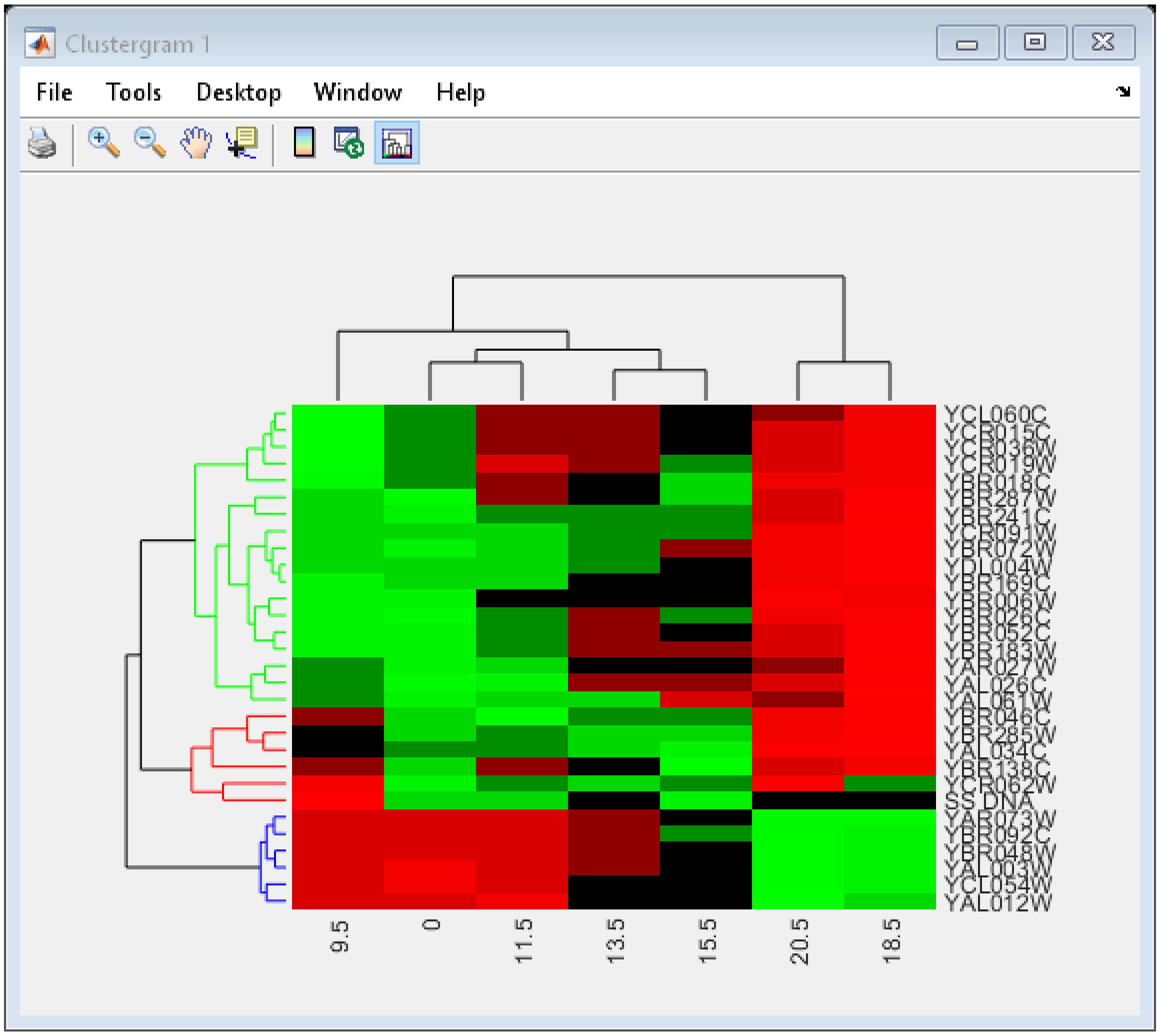
```
get(cgo)
```

```
               Cluster: 'ALL'
              RowPDist: {'Euclidean'}
```

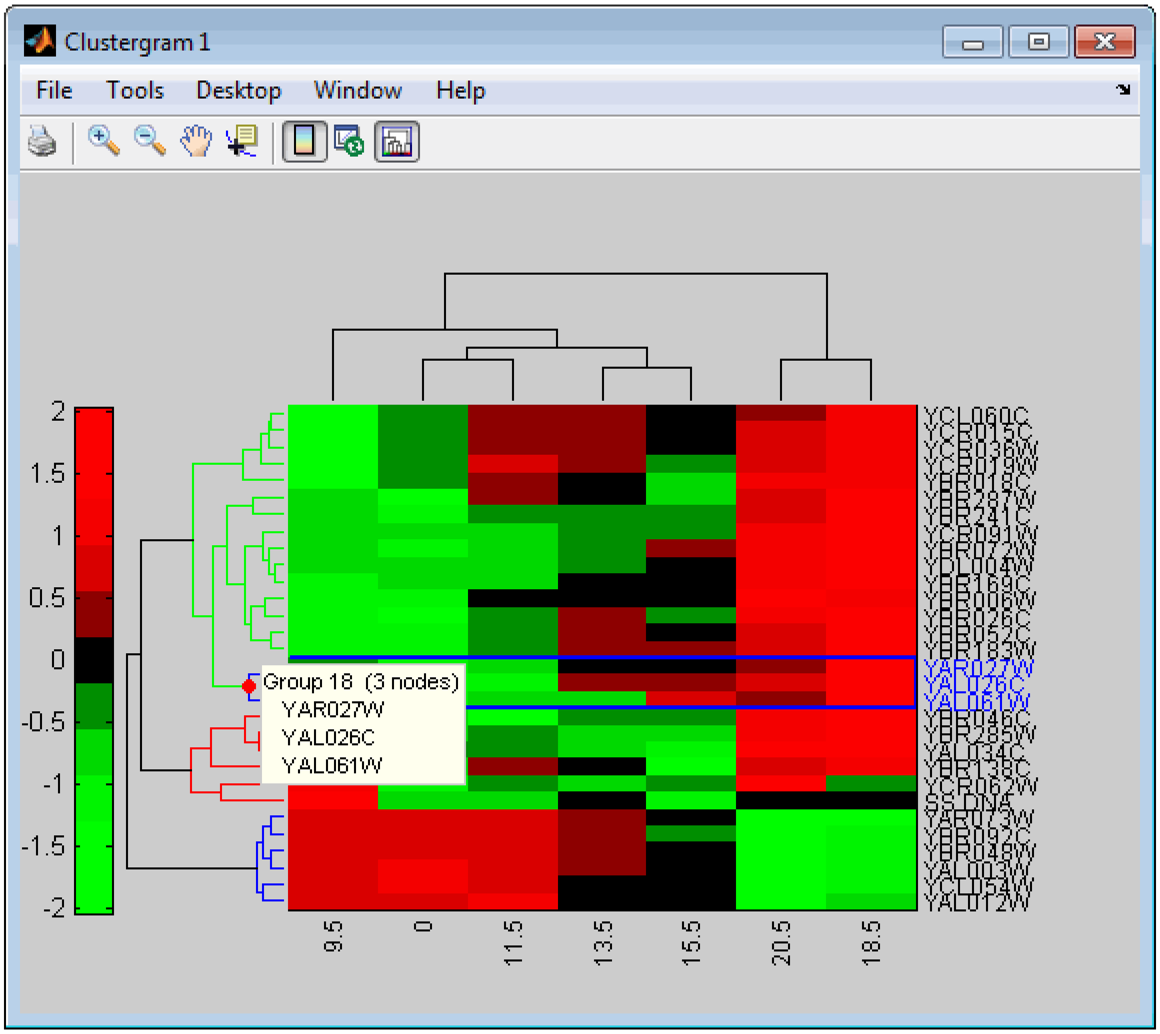
```
          ColumnPDist: {'Euclidean'}
              Linkage: {'Average'}
           Dendrogram: {}
      OptimalLeafOrder: 1
             LogTrans: 0
         DisplayRatio: [0.2000 0.2000]
       RowGroupMarker: []
    ColumnGroupMarker: []
       ShowDendrogram: 'on'
          Standardize: 'ROW'
            Symmetric: 1
         DisplayRange: 3
             Colormap: [11x3 double]
            ImputeFun: []
         ColumnLabels: {1x7 cell}
            RowLabels: {30x1 cell}
   ColumnLabelsRotate: 90
      RowLabelsRotate: 0
             Annotate: 'off'
       AnnotPrecision: 2
           AnnotColor: 'w'
    ColumnLabelsColor: []
       RowLabelsColor: []
    LabelsWithMarkers: 0
```

Change the clustering parameters by changing the linkage method and changing the color of the groups of nodes in the dendrogram whose linkage is less than a threshold of 3.

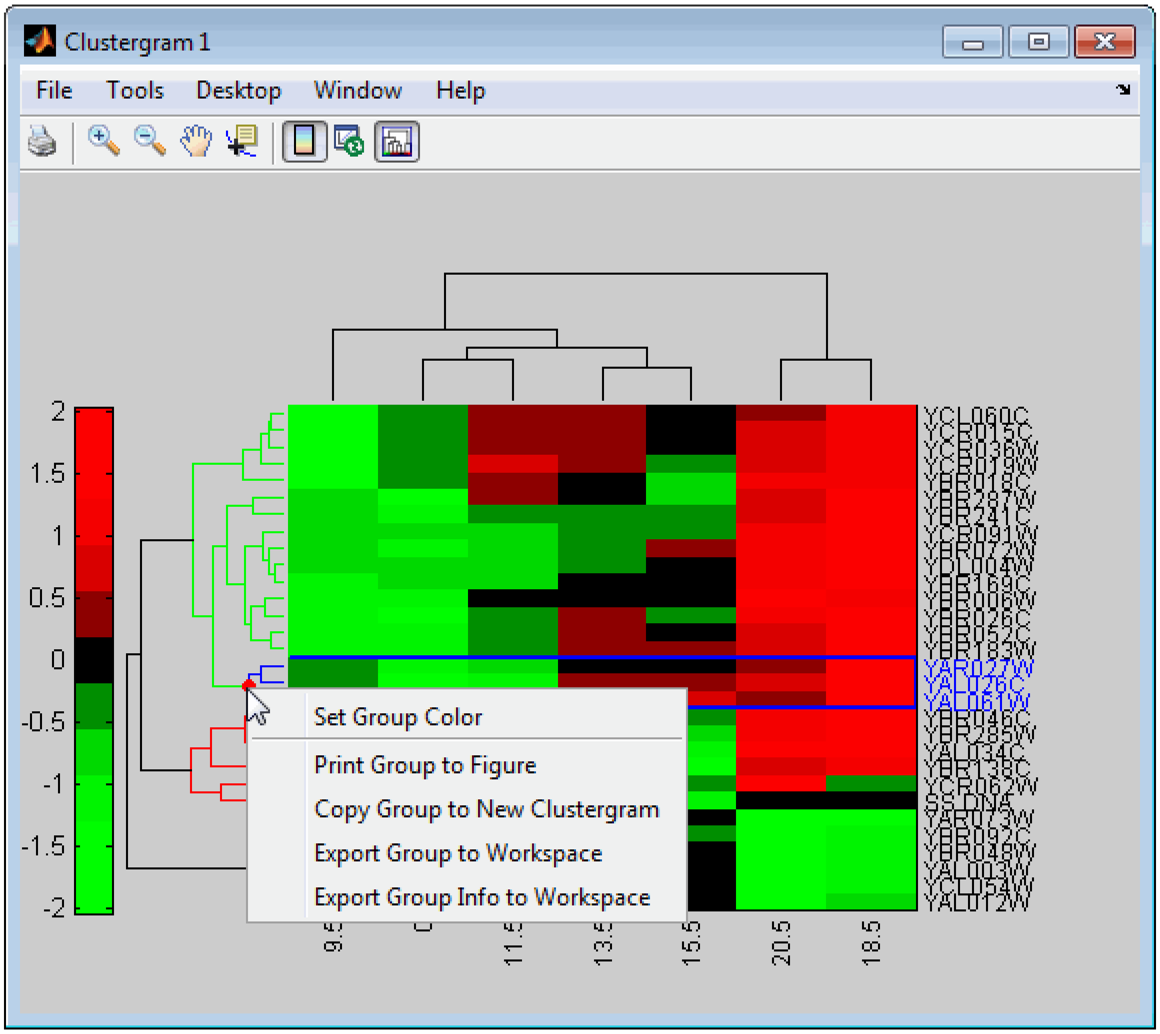
```
set(cgo,'Linkage','complete','Dendrogram',3)
```



Place the cursor on a branch node in the dendrogram to highlight (in blue) the group associated with it. Press and hold the mouse button to display a data tip listing the group number and the nodes (genes or samples) in the group.



Right-click a branch node in the dendrogram to display a menu of options.



The following options are available:

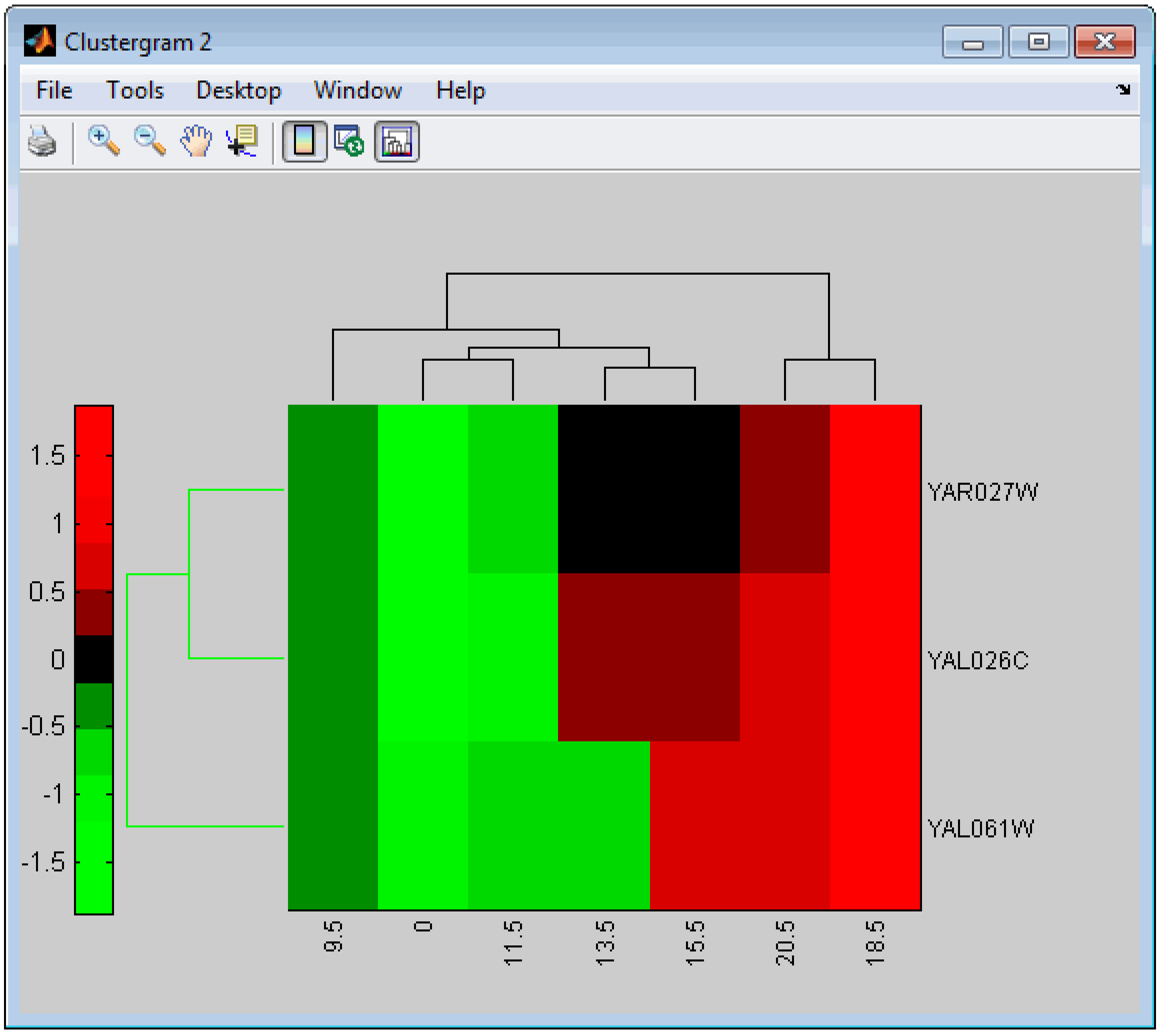
- **Set Group Color** - Change the cluster group color. - **Print Group to Figure** - Print the group to a figure window. - **Copy Group to New Clustergram** - Copy the group to a new clustergram window. - **Export Group to Workspace** - Create a clustergram object of the group in the MATLAB workspace. - **Export Group Info to Workspace** - Create a structure containing information about the group in the MATLAB workspace. The structure contains these fields:

- `GroupNames` - Cell array of character vectors containing the names of the row or column groups. - `RowNodeNames` - Cell array of character vectors containing the names of the row nodes. - `ColumnNodeNames` - Cell array of character vectors containing the names of the column nodes. - `ExprValues` - An M-by-N matrix of intensity values, where M and N are the number of row nodes and of column nodes respectively. If the matrix contains gene expression data, typically each row corresponds to a gene and each column corresponds to sample.

Create a clustergram object for Group 18 in the MATLAB workspace. Right-click Group 18, then select **Export Group to Workspace** . In the **Export to Workspace** dialog box, type `Group18`, then click **OK** .

Use the `view` method to view the clustergram object, `Group18`.

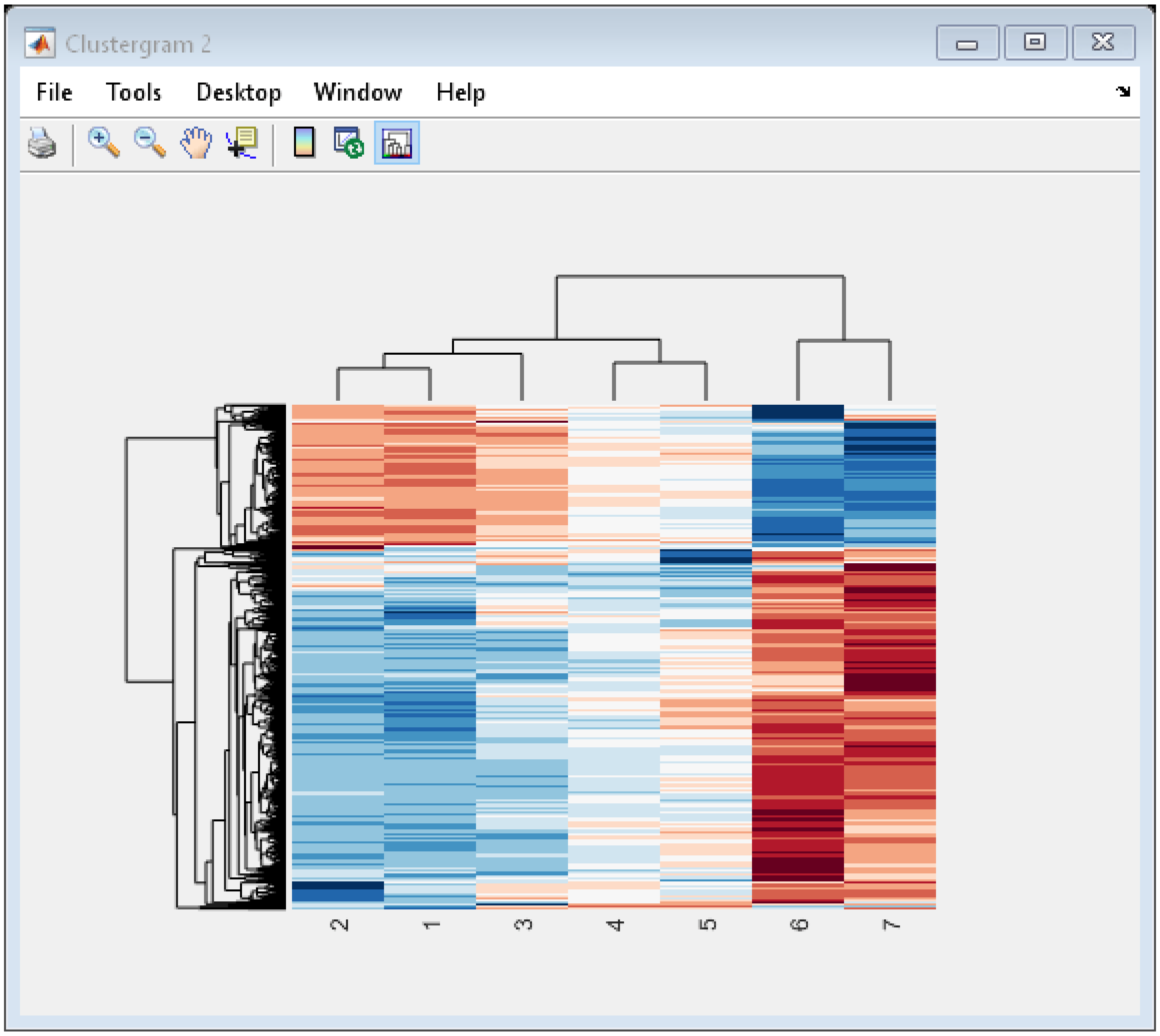
```
view(Group18)
```



View all the gene expression data using a diverging red and blue colormap and standardize along the rows of data.

```
cgo_all = clustergram(yeastvalues,'Colormap',redbluecmap,'Standardize','Row')
```

```
Clustergram object with 614 rows of nodes and 7 columns of nodes.
```

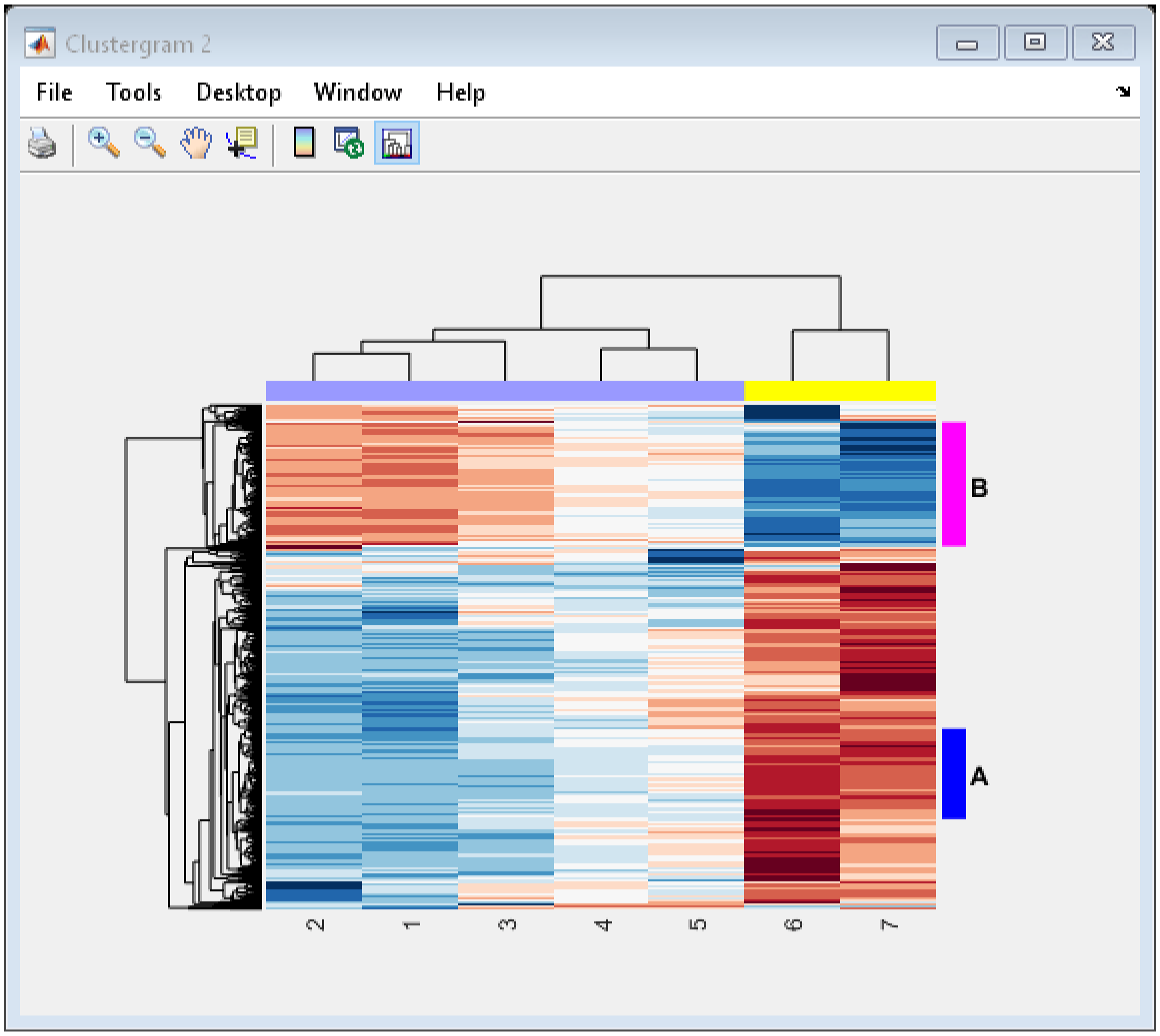


Create structure arrays to specify marker colors and annotations for two groups of rows (510 and 593) and two groups of columns (4 and 5).

```
rm = struct('GroupNumber',{510,593},'Annotation',{'A','B'},...
     'Color',{'b','m'});
cm = struct('GroupNumber',{4,5},'Annotation',{'Time1','Time2'},...
     'Color',{[1 1 0],[0.6 0.6 1]});
```

Use the `RowGroupMarker` and `ColumnGroupMarker` properties to add the color markers and annotations to the clustergram.

```
set(cgo_all,'RowGroupMarker',rm,'ColumnGroupMarker',cm)
```



Input Arguments

`length` — Number of colors in colormap
11 (default) | positive integer between 3 and 11

Number of colors in the colormap, specified as a positive integer between 3 and 11.

Example: 4

Data Types: `double`

Version History

Introduced in R2008a

References

[1] DeRisi, J. L. “Exploring the Metabolic and Genetic Control of Gene Expression on a Genomic Scale.” *Science* 278, no. 5338 (October 24, 1997): 680–86.

See Also

`clustergram` | `redgreencmap` | `colormap` | Colormap Editor

Topics

“Analyzing Illumina Bead Summary Gene Expression Data”

redgreencmap

Create red and green colormap

Syntax

```
redgreencmap(length)
redgreencmap(length,'Interpolation',interpMethod)
```

Description

`redgreencmap(length)` returns a `length`-by-3 matrix containing a red and green colormap. Low values are bright green, values in the center of the map are black, and high values are red.

`redgreencmap(length,'Interpolation',interpMethod)` specifies the algorithm to use for color interpolation.

Examples

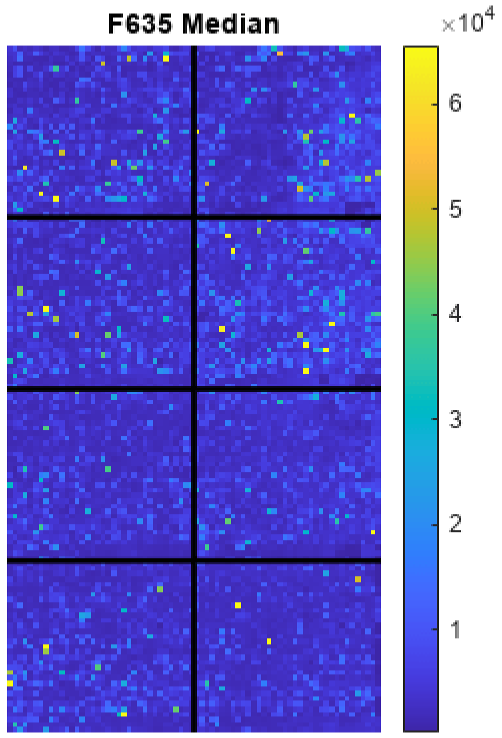
Generate Spatial Image and Change Colormap of Microarray Data

Read in a sample GPR file.

```
madata = gprread('mouse_a1wt.gpr');
```

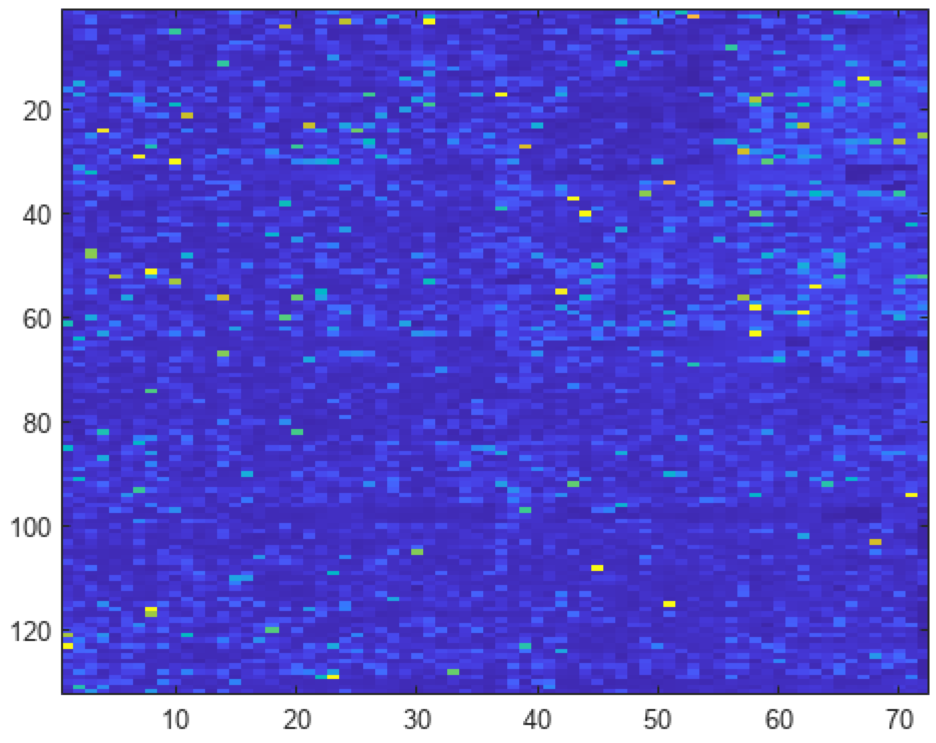
Plot the median foreground intensity for the 635 nm channel.

```
maimage(madata,'F635 Median')
```



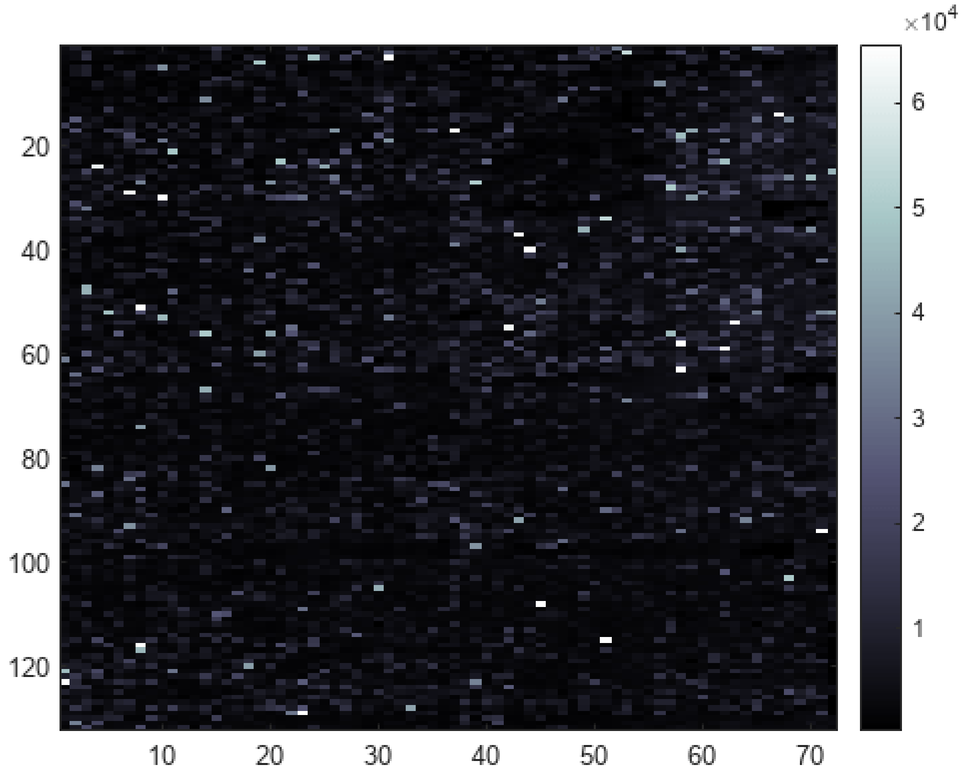
Alternatively, create a similar plot using more basic graphics commands.

```
F635Median = magetfield(madata,'F635 Median');
figure
imagesc(F635Median(madata.Indices));
```



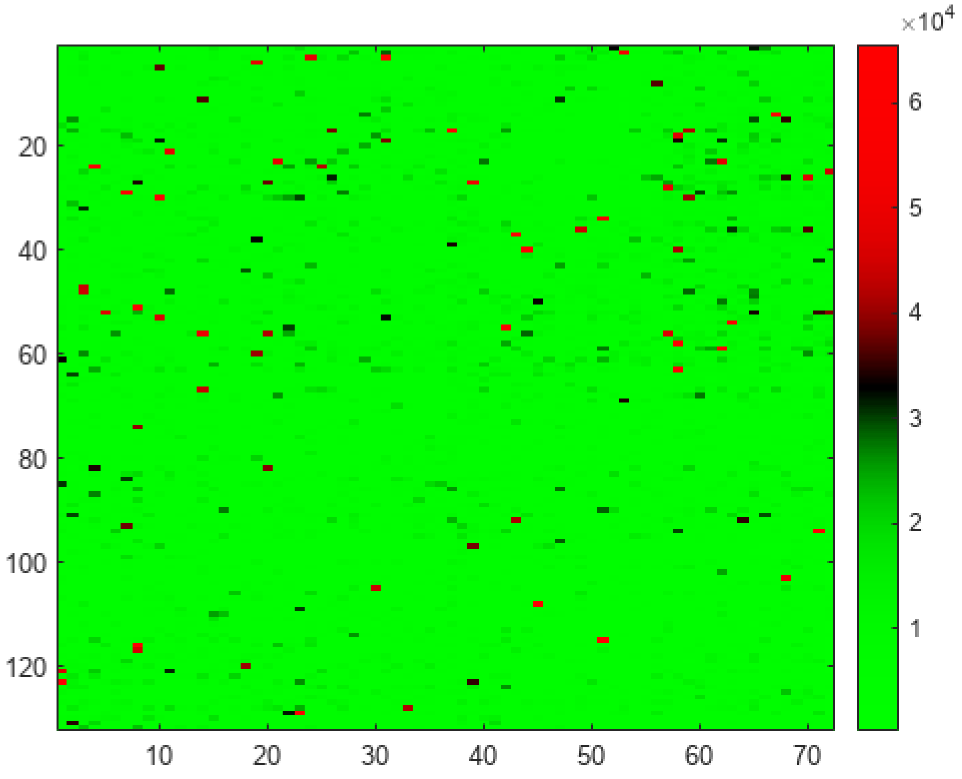
Change the colormap to one of the preset colors (for details, see “map”) and add a color bar.

```
colormap('bone')
colorbar
```



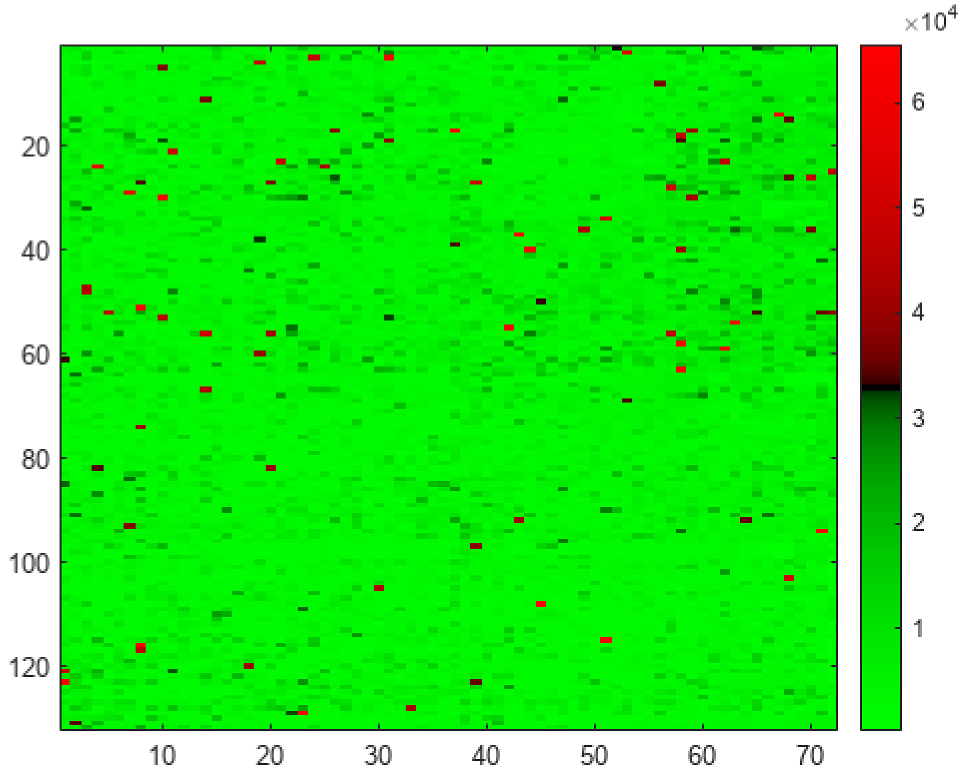
Change the colormap to red and green.

```
colormap(redgreencmap)
```



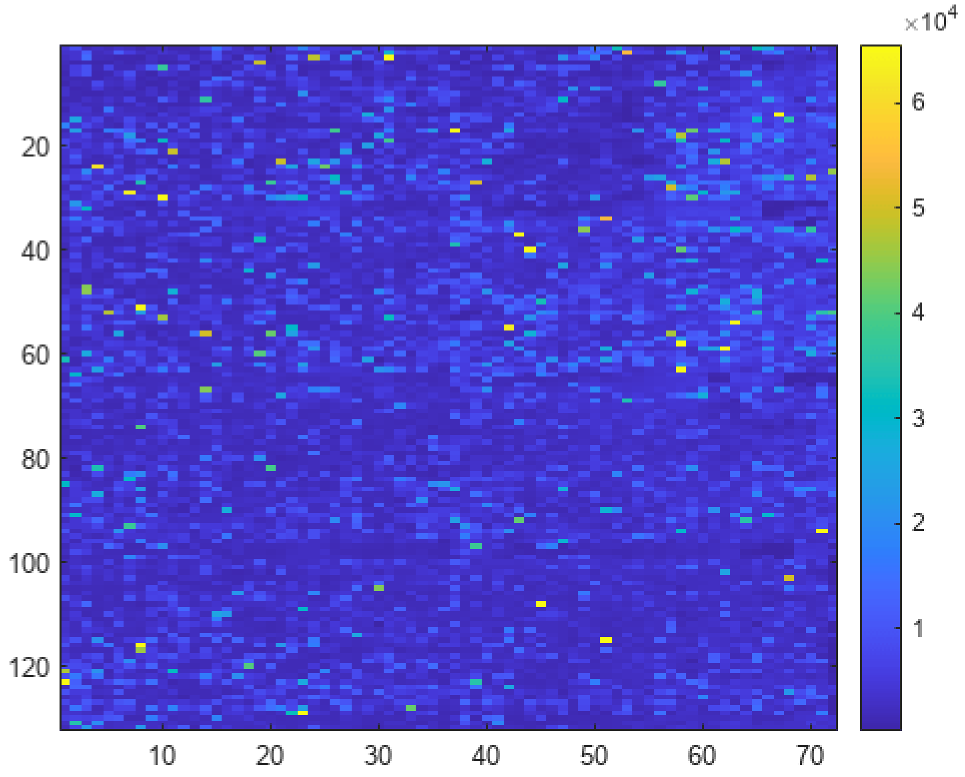
You can also change the color interpolation method.

```
colormap(redgreencmap(256,'Interpolation','cubic'))
```



Reset the colormap to the default value.

```
colormap('default')
```



Input Arguments

`length` — Number of colors in colormap
`[]` | nonnegative integer

Number of colors in the colormap, specified as empty `[]` or a nonnegative integer. If `length` is empty, the function automatically sets the value to the length of the colormap of the current figure.

Example: 64

Data Types: `double`

`interpMethod` — Interpolation algorithm
`'sigmoid'` (default) | `'linear'` | `'quadratic'` | `'cubic'`

Interpolation algorithm for color interpolation, specified as `'sigmoid'` (which uses `tanh`), `'linear'`, `'quadratic'`, or `'cubic'`.

Example: `'cubic'`

Data Types: `char` | `string`

Version History

Introduced before R2006a

See Also

`clustergram` | `redbluecmap` | `colormap` | Colormap Editor

Topics

"Analyzing Illumina Bead Summary Gene Expression Data"

reorder (phytree)

Reorder leaves of phylogenetic tree

Syntax

```
Tree1Reordered = reorder(Tree1, Order)
[Tree1Reordered, OptimalOrder] = reorder(Tree1, Order, 'Approximate',
ApproximateValue)
[Tree1Reordered, OptimalOrder] = reorder(Tree1, Tree2)
```

Input Arguments

Tree1, *Tree2*	Phytree objects.
Order	Vector with position indices for each leaf.
ApproximateValue	Controls the use of the optimal leaf-ordering calculation to find the closest order possible to the suggested one without dividing the clades or producing crossing branches. Enter `true` to use the calculation. Default is `false`.

Output Arguments

Tree1Reordered	Phytree object with reordered leaves.
OptimalOrder	Vector of position indices for each leaf in *Tree1Reordered*, determined by the optimal leaf-ordering calculation.

Description

Tree1Reordered = `reorder(`*Tree1*, *Order*`)` reorders the leaves of the phylogenetic tree *Tree1*, without modifying its structure and distances, creating a new phylogenetic tree, *Tree1Reordered*. *Order* is a vector of position indices for each leaf. If *Order* is invalid, that is, if it divides the clades (or produces crossing branches), then `reorder` returns an error message.

[*Tree1Reordered*, *OptimalOrder*] = `reorder(`*Tree1*, *Order*, `'Approximate'`, *ApproximateValue*`)` controls the use of the optimal leaf-ordering calculation, which finds the best approximate order closest to the suggested one, without dividing the clades or producing crossing branches. Enter `true` to use the calculation and return *Tree1Reordered*, the reordered tree, and *OptimalOrder*, a vector of position indices for each leaf in *Tree1Reordered*, determined by the optimal leaf-ordering calculation. Default is `false`.

[*Tree1Reordered*, *OptimalOrder*] = `reorder(`*Tree1*, *Tree2*`)` uses the optimal leaf-ordering calculation to reorder the leaves in *Tree1* such that it matches the order of leaves in *Tree2* as closely as possible, without dividing the clades or producing crossing branches. *Tree1Reordered* is the reordered tree, and *OptimalOrder* is a vector of position indices for each leaf in *Tree1Reordered*, determined by the optimal leaf-ordering calculation

Examples

Example 1.48. Reordering Leaves Using a Valid Order

1 Create and view a phylogenetic tree.

```
b = [1 2; 3 4; 5 6; 7 8; 9 10];
tree = phytree(b)
   Phylogenetic tree object with 6 leaves (5 branches)
view(tree)
```

2 Reorder the leaves on the phylogenetic tree, and then view the reordered tree.

```
 treeReordered = reorder(tree, [5, 6, 3, 4, 1, 2])
 view(treeReordered)
```

Example 1.49. Finding Best Approximate Order When Using an Invalid Order

1 Create a phylogenetic tree by reading a Newick-formatted tree file (ASCII text file).

```
tree = phytreeread('pf00002.tree')
    Phylogenetic tree object with 33 leaves (32 branches)
```

2 Create a row vector of the leaf names in alphabetical order.

```
[dummy,order] = sort(get(tree,'LeafNames'));
```

3 Reorder the phylogenetic tree to match as closely as possible the row vector of alphabetically ordered leaf names, without dividing the clades or having crossing branches.

```
treeReordered = reorder(tree,order,'approximate',true)
    Phylogenetic tree object with 33 leaves (32 branches)
```

4 View the original and the reordered phylogenetic trees.

```
view(tree)
view(treeReordered)
```

Example 1.50. Reordering Leaves to Match Leaf Order in Another Phylogenetic Tree

1 Create a phylogenetic tree by reading sequence data from a FASTA file, calculating the pairwise distances between sequences, and then using the neighbor-joining method.

```
seqs = fastaread('pf00002.fa')

seqs =

33x1 struct array with fields:
    Header
    Sequence

dist = seqpdist(seqs,'method','jukes-cantor','indels','pair');
NJtree = seqneighjoin(dist,'equivar',seqs)
    Phylogenetic tree object with 33 leaves (32 branches)
```

2 Create another phylogenetic tree from the same sequence data and pairwise distances between sequences, using the single linkage method.

```
HCtree = seqlinkage(dist,'single',seqs)
    Phylogenetic tree object with 33 leaves (32 branches)
```

3 Use the optimal leaf-ordering calculation to reorder the leaves in `HCtree` such that it matches the order of leaves in `NJtree` as closely as possible, without dividing the clades or having crossing branches.

```
HCtree_reordered = reorder(HCtree,NJtree)
    Phylogenetic tree object with 33 leaves (32 branches)
```

4 View the reordered phylogenetic tree and the tree used to reorder it.

```
view(HCtree_reordered)
view(NJtree)
```

Version History
Introduced in R2007a

See Also
`phytree` | `get` | `getbyname` | `prune`

Topics
phytree object on page 1-1449

reroot (phytree)

Change root of phylogenetic tree

Syntax

```
Tree2 = reroot(Tree1)
Tree2 = reroot(Tree1, Node)
Tree2 = reroot(Tree1, Node, Distance)
```

Arguments

Tree1	Phylogenetic tree (`phytree` object) created with the function `phytree`.
Node	Node index returned by the phytree object method `getbyname`.
Distance	Distance from the reference branch.

Description

`Tree2 = reroot(Tree1)` changes the root of a phylogenetic tree (*Tree1*) using a midpoint method. The midpoint is the location where the mean values of the branch lengths, on either side of the tree, are equalized. The original root is deleted from the tree.

`Tree2 = reroot(Tree1, Node)` changes the root of a phylogenetic tree (*Tree1*) to a branch node using the node index (*Node*). The new root is placed at half the distance between the branch node and its parent.

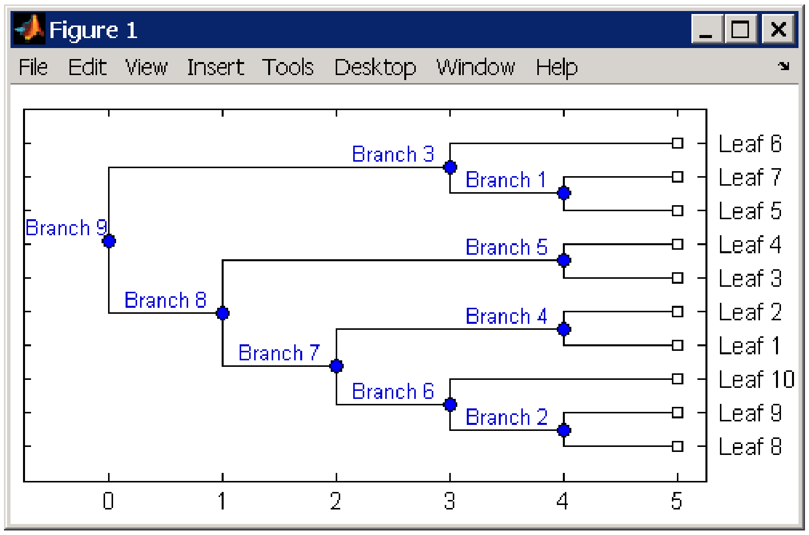
`Tree2 = reroot(Tree1, Node, Distance)` changes the root of a phylogenetic tree (*Tree1*) to a new root at a given distance (*Distance*) from the reference branch node (*Node*) toward the original root of the tree. Note: The new branch representing the root in the new tree (`Tree2`) is labeled `'Root'`.

Examples

1 Create an ultrametric tree.

```
tr_1 = phytree([5 7;8 9;6 11; 1 2;3 4;10 12;...
                14 16; 15 17;13 18])
plot(tr_1,'branchlabels',true)
```

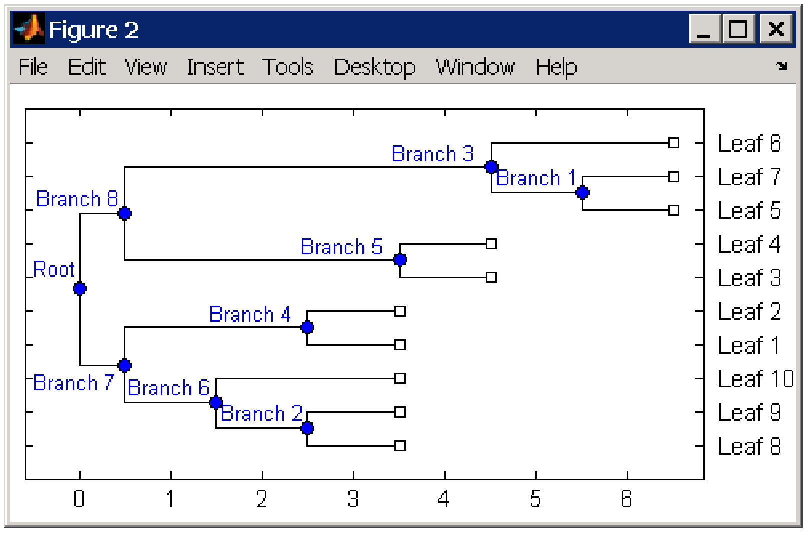
A figure with the phylogenetic tree displays.



2 Place the root at `'Branch 7'`.

```
sel = getbyname(tr_1,'Branch 7');
tr_2 =  reroot(tr_1,sel)
plot(tr_2,'branchlabels',true)
```

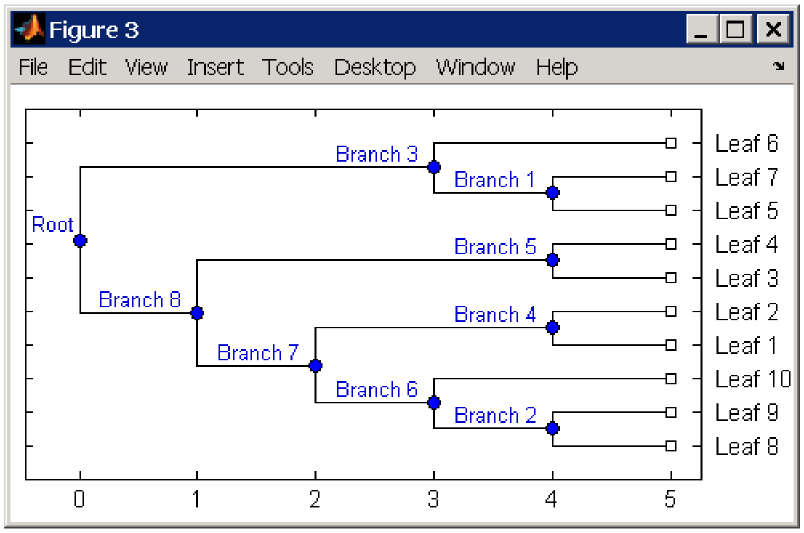
A figure of a phylogenetic tree displays with the root moved to the center of branch 7.



3 Move the root to a branch that makes the tree as ultrametric as possible.

```
tr_3 = reroot(tr_2)
plot(tr_3,'branchlabels',true)
```

A figure of the new tree displays with the root moved from the center of branch 7 to branch 8.



Version History

Introduced before R2006a

See Also

`phytree` | `seqneighjoin` | `get` | `getbyname` | `prune` | `select`

Topics

phytree object on page 1-1449

restrict

Split nucleotide sequence at restriction site

Syntax

```
Fragments = restrict(SeqNT, Enzyme)
Fragments = restrict(SeqNT, NTPattern, Position)
[Fragments, CuttingSites] = restrict(...)
[Fragments, CuttingSites, Lengths] = restrict(...)

... = restrict(..., 'PartialDigest', PartialDigestValue)
```

Arguments

SeqNT	One of the following: • Character vector or string specifying a nucleotide sequence. For valid letter codes, see the table Mapping Nucleotide Letter Codes to Integers. • Row vector of integers specifying a nucleotide sequence. For valid integers, see the table Mapping Nucleotide Integers to Letter Codes. • MATLAB structure containing a `Sequence` field that contains a nucleotide sequence, such as returned by `fastaread`, `fastqread`, `emblread`, `getembl`, `genbankread`, or `getgenbank`.
Enzyme	Character vector or string specifying a name of a restriction enzyme from REBASE, the Restriction Enzyme Database. **Tip** Some enzymes specify cutting rules for both a strand and its complement strand. `restrict` applies the cutting rule only for the 5' —> 3' strand. For a workaround to applying an enzyme cutting rule for both strands, see “Examples” on page 1-1592.
NTPattern	Short nucleotide sequence recognition pattern to search for in *SeqNT*, a larger sequence. *NTPattern* can be either of the following: • Character vector or string • Regular expression

`Position`	Either of the following: • Integer specifying a position in the *`SeqNT`* to cut, relative to *`NTPattern`*. • Two-element vector specifying two positions in the *`SeqNT`* to cut, relative to *`NTPattern`*. **Note** Position `0` corresponds to a cut before the first base of *`NTPattern`*.
`PartialDigestValue`	Value from `0` to `1` (default) specifying the probability that a cleavage site will be cut.

Description

`Fragments` `= restrict(`*`SeqNT`*`, `*`Enzyme`*`)` cuts *`SeqNT`*, a nucleotide sequence, into fragments at the restriction sites of *`Enzyme`*, a restriction enzyme. The `restrict` function stores the return values in *`Fragments`*, a cell array of sequences.

`Fragments` `= restrict(`*`SeqNT`*`, `*`NTPattern`*`, `*`Position`*`)` cuts *`SeqNT`*, a nucleotide sequence, into fragments at restriction sites specified by *`NTPattern`*, a nucleotide recognition pattern, and *`Position`*.

`[`*`Fragments`*`, `*`CuttingSites`*`] = restrict(...)` returns a numeric vector with the indices representing the cutting sites. The `restrict` function adds a `0` to the beginning of the *`CuttingSites`* vector so that the number of elements in *`CuttingSites`* equals the number of elements in *`Fragments`*. You can use *`CuttingSites`* `+ 1` to point to the first base of every fragment respective to the original sequence.

`[`*`Fragments`*`, `*`CuttingSites`*`, `*`Lengths`*`] = restrict(...)` returns a numeric vector with the lengths of every fragment.

`... = restrict(..., 'PartialDigest', `*`PartialDigestValue`*`)` simulates a partial digest where each restriction site in the sequence has a *`PartialDigestValue`* or probability of being cut.

REBASE, the Restriction Enzyme Database, is a collection of information about restriction enzymes and related proteins. For more information about REBASE or to search REBASE for the name of a restriction enzyme, see:

`http://rebase.neb.com/rebase/rebase.html`

Examples

Example 1.51. Splitting a Nucleotide Sequence by Specifying an Enzyme

1 Enter a nucleotide sequence.

```
Seq = 'AGAGGGGTACGCGCTCTGAAAAGCGGGAACCTCGTGGCGCTTTATTAA';
```

2 Use the restriction enzyme `HspAI` (which specifies a recognition sequence of `GCGC` and a cleavage position of `1`) to cleave the nucleotide sequence.

```
fragmentsEnzyme = restrict(Seq,'HspAI')
```

MATLAB returns:

```
fragmentsEnzyme =

    'AGAGGGGTACG'
    'CGCTCTGAAAAGCGGGAACCTCGTGG'
    'CGCTTTATTAA'
```

Example 1.52. Splitting a Nucleotide Sequence by Specifying a Pattern and Position

1. Enter a nucleotide sequence.

   ```
   Seq = 'AGAGGGGTACGCGCTCTGAAAAGCGGGAACCTCGTGGCGCTTTATTAA';
   ```
2. Use the sequence pattern GCGC with the point of cleavage at position 3 to cleave the nucleotide sequence.

   ```
   fragmentsPattern = restrict(Seq,'GCGC',3)
   ```

   MATLAB returns:

   ```
   fragmentsPattern =

       'AGAGGGGTACGCG'
       'CTCTGAAAAGCGGGAACCTCGTGGCG'
       'CTTTATTAA'
   ```

Example 1.53. Splitting a Nucleotide Sequence by Specifying a Regular Expression for the Pattern

1. Enter a nucleotide sequence.

   ```
   Seq = 'AGAGGGGTACGCGCTCTGAAAAGCGGGAACCTCGTGGCGCTTTATTAA';
   ```
2. Use a regular expression to specify the sequence pattern.

   ```
   fragmentsRegExp = restrict(Seq,'GCG[^C]',3)
   ```

   MATLAB returns:

   ```
   fragmentsRegExp =

       'AGAGGGGTACGCGCTCTGAAAAGCG'
       'GGAACCTCGTGGCGCTTTATTAA'
   ```

Example 1.54. Returning the Cutting Sites and Fragment Lengths

1. Enter a nucleotide sequence.

   ```
   Seq = 'AGAGGGGTACGCGCTCTGAAAAGCGGGAACCTCGTGGCGCTTTATTAA';
   ```
2. Capture the cutting sites and fragment lengths as well as the fragments.

   ```
   [fragments, cut_sites, lengths] = restrict(Seq,'HspAI')
   ```

   MATLAB returns:

   ```
   fragments =
       'AGAGGGGTACG'
       'CGCTCTGAAAAGCGGGAACCTCGTGG'
       'CGCTTTATTAA'
   ```

```
cut_sites =
     0
    11
    37

lengths =
    11
    26
    11
```

Example 1.55. Splitting a Double-Stranded Nucleotide Sequence

Some enzymes specify cutting rules for both a strand and its complement strand. `restrict` applies the cutting rule only for the 5' —> 3' strand. You can apply this rule manually for the complement strand.

1. Enter a nucleotide sequence.

   ```
   seq = 'CCCGCNNNNNNN';
   ```
2. Use the `seqcomplement` function to determine the complement strand, which is in the 3' —> 5' direction.

   ```
   seqc = seqcomplement(seq)
   ```

   MATLAB returns:

   ```
   seqc =

   GGGCGNNNNNNN
   ```
3. Cut the first strand using the restriction enzyme `FauI` (which specifies a recognition sequence pattern of `CCCGC` and a cleavage position of `9`).

   ```
   cuts_strand1 = restrict(seq, 'FauI')
   ```

   MATLAB returns:

   ```
   cuts_strand1 =

       'CCCGCNNNN'
       'NNN'
   ```
4. Cut the complement strand according the rule specified by `FauI` (which specifies a recognition sequence pattern of `GGGCG` with the point of cleavage at position `11`).

   ```
   cuts_strand2 = restrict(seqc, 'GGGCG', 11)
   ```

   MATLAB returns:

   ```
   cuts_strand2 =

       'GGGCGNNNNNN'
       'N'
   ```

Version History

Introduced before R2006a

References

[1] Roberts, R.J., Vincze, T., Posfai, J., and Macelis, D. (2007). REBASE—enzymes and genes for DNA restriction and modification. Nucl. Acids Res. *35*, D269–D270.

[2] Official REBASE Web site: `http://rebase.neb.com`.

See Also

`cleave` | `cleavelookup` | `rebasecuts` | `seq2regexp` | `seqcomplement` | `regexp`

revgeneticcode

Return reverse mapping (amino acid to nucleotide codon) for genetic code

Syntax

```
Map = revgeneticcode
Map = revgeneticcode(GeneticCode)

Map = revgeneticcode(..., 'Alphabet', AlphabetValue, ...)
Map = revgeneticcode(..., 'ThreeLetterCodes', ThreeLetterCodesValue, ...)
```

Input Arguments

GeneticCode	Integer, character vector, or string specifying a genetic code number or code name from the table Genetic Code. Default is `1` or `'Standard'`. **Tip** If you use a code name, you can truncate the name to the first two letters of the name.
AlphabetValue	Character vector or string specifying the nucleotide alphabet to use in the map. Choices are: • `'DNA'` (default) — Uses the symbols `A`, `C`, `G`, and `T`. • `'RNA'` — Uses the symbols `A`, `C`, `G`, and `U`.
ThreeLetterCodesValue	Controls the use of three-letter amino acid codes as field names in the return structure *Map*. Choices are `true` for three-letter codes or `false` for one-letter codes. Default is `false`.

Output Arguments

Map	Structure containing the reverse mapping of amino acids to nucleotide codons for the standard genetic code. The *Map* structure contains a field for each amino acid.

Description

Map = `revgeneticcode` returns a structure containing the reverse mapping of amino acids to nucleotide codons for the standard genetic code. The *Map* structure contains a field for each amino acid.

Map = `revgeneticcode(`*GeneticCode*`)` returns a structure containing the reverse mapping of amino acids to nucleotide codons for the specified genetic code. *GeneticCode* is either:

- An integer, character vector, or string specifying a code number or code name from the table Genetic Code

- The `transl_table` (code) number from the NCBI Web page describing genetic codes:

  `https://www.ncbi.nlm.nih.gov/Taxonomy/Utils/wprintgc.cgi?mode=c`

Tip If you use a code name, you can truncate the name to the first two letters of the name.

`*Map* = revgeneticcode(..., '*PropertyName*', *PropertyValue*, ...)` calls `revgeneticcode` with optional properties that use property name/property value pairs. You can specify one or more properties in any order. Each `*PropertyName*` must be enclosed in single quotation marks and is case insensitive. These property name/property value pairs are as follows:

`*Map* = revgeneticcode(..., 'Alphabet', *AlphabetValue*, ...)` specifies the nucleotide alphabet to use in the map. `*AlphabetValue*` can be `'DNA'`, which uses the symbols `A`, `C`, `G`, and `T`, or `'RNA'`, which uses the symbols `A`, `C`, `G`, and `U`. Default is `'DNA'`.

`*Map* = revgeneticcode(..., 'ThreeLetterCodes', *ThreeLetterCodesValue*, ...)` controls the use of three-letter amino acid codes as field names in the return structure `*Map*`. `*ThreeLetterCodesValue*` can be `true` for three-letter codes or `false` for one-letter codes. Default is `false`.

Genetic Code

Code Number	Code Name
1	`Standard`
2	`Vertebrate Mitochondrial`
3	`Yeast Mitochondrial`
4	`Mold`, `Protozoan`, `Coelenterate Mitochondrial`, and `Mycoplasma/ Spiroplasma`
5	`Invertebrate Mitochondrial`
6	`Ciliate`, `Dasycladacean`, and `Hexamita Nuclear`
9	`Echinoderm Mitochondrial`
10	`Euplotid Nuclear`
11	`Bacterial` and `Plant Plastid`
12	`Alternative Yeast Nuclear`
13	`Ascidian Mitochondrial`
14	`Flatworm Mitochondrial`
15	`Blepharisma Nuclear`
16	`Chlorophycean Mitochondrial`
21	`Trematode Mitochondrial`
22	`Scenedesmus Obliquus Mitochondrial`
23	`Thraustochytrium Mitochondrial`

Examples

- Return the reverse mapping of amino acids to nucleotide codons for the `Standard` genetic code.

```
map = revgeneticcode
```

```
map = 

      Name: 'Standard'
         A: {'GCT'  'GCC'  'GCA'  'GCG'}
         R: {'CGT'  'CGC'  'CGA'  'CGG'  'AGA'  'AGG'}
         N: {'AAT'  'AAC'}
         D: {'GAT'  'GAC'}
         C: {'TGT'  'TGC'}
         Q: {'CAA'  'CAG'}
         E: {'GAA'  'GAG'}
         G: {'GGT'  'GGC'  'GGA'  'GGG'}
         H: {'CAT'  'CAC'}
         I: {'ATT'  'ATC'  'ATA'}
         L: {'TTA'  'TTG'  'CTT'  'CTC'  'CTA'  'CTG'}
         K: {'AAA'  'AAG'}
         M: {'ATG'}
         F: {'TTT'  'TTC'}
         P: {'CCT'  'CCC'  'CCA'  'CCG'}
         S: {'TCT'  'TCC'  'TCA'  'TCG'  'AGT'  'AGC'}
         T: {'ACT'  'ACC'  'ACA'  'ACG'}
         W: {'TGG'}
         Y: {'TAT'  'TAC'}
         V: {'GTT'  'GTC'  'GTA'  'GTG'}
     Stops: {'TAA'  'TAG'  'TGA'}
    Starts: {'TTG'  'CTG'  'ATG'}
```

- Return the reverse mapping of amino acids to nucleotide codons for the Mold, Protozoan, Coelenterate Mitochondrial, and Mycoplasma/Spiroplasma genetic code, using the rna alphabet.

```
moldmap = revgeneticcode(4,'Alphabet','rna');
```

- Return the reverse mapping of amino acids to nucleotide codons for the Flatworm Mitochondrial genetic code, using three-letter codes for the field names in the return structure.

```
wormmap = revgeneticcode('Flatworm Mitochondrial',...
                          'ThreeLetterCodes',true);
```

Version History

Introduced before R2006a

References

[1] NCBI Web page describing genetic codes:

`https://www.ncbi.nlm.nih.gov/Taxonomy/Utils/wprintgc.cgi?mode=c`

See Also

`aa2nt` | `aminolookup` | `baselookup` | `geneticcode` | `nt2aa`

rmabackadj

Perform background adjustment on Affymetrix microarray probe-level data using Robust Multi-array Average (RMA) procedure

Syntax

```
BackAdjustedMatrix = rmabackadj(PMData)

BackAdjustedMatrix = rmabackadj(..., 'Method', MethodValue, ...)
BackAdjustedMatrix = rmabackadj(..., 'Truncate', TruncateValue, ...)
BackAdjustedMatrix = rmabackadj(..., 'Showplot', ShowplotValue, ...)
```

Input Arguments

PMData	Matrix of intensity values where each row corresponds to a perfect match (PM) probe and each column corresponds to an Affymetrix CEL file. (Each CEL file is generated from a separate chip. All chips should be of the same type.)
MethodValue	Specifies the estimation method for the background adjustment model parameters. Enter either `'RMA'` (to use estimation method described by Bolstad, 2005) or `'MLE'` (to estimate the parameters using maximum likelihood). Default is `'RMA'`.
TruncateValue	Specifies the background noise model. Enter either `true` (use a truncated Gaussian distribution) or `false` (use a nontruncated Gaussian distribution). Default is `true`.
ShowplotValue	Controls the plotting of a histogram showing the distribution of PM probe intensity values (blue) and the convoluted probability distribution function (red), with estimated parameters mu, sigma and alpha. Enter either `'all'` (plot a histogram for each column or chip) or specify a subset of columns (chips) by entering the column number, list of numbers, or range of numbers.

Output Arguments

BackAdjustedMatrix	Matrix of background-adjusted probe intensity values.

Description

`BackAdjustedMatrix = rmabackadj(PMData)` returns the background adjusted values of probe intensity values in the matrix, *PMData*. Note that each row in *PMData* corresponds to a perfect match (PM) probe and each column in *PMData* corresponds to an Affymetrix CEL file. (Each CEL file is generated from a separate chip. All chips should be of the same type.) Details on the background adjustment are described by Bolstad, 2005.

`BackAdjustedMatrix = rmabackadj(..., 'PropertyName', PropertyValue, ...)` calls `rmabackadj` with optional properties that use property name/property value pairs. You can specify one or more properties in any order. Each *PropertyName* must be enclosed in single quotation marks and is case insensitive. These property name/property value pairs are as follows:

BackAdjustedMatrix = rmabackadj(..., 'Method', *MethodValue*, ...) specifies the estimation method for the background adjustment model parameters. When *MethodValue* is 'RMA', rmabackadj implements the estimation method described by Bolstad, 2005. When *MethodValue* is 'MLE', rmabackadj estimates the parameters using maximum likelihood. Default is 'RMA'.

BackAdjustedMatrix = rmabackadj(..., 'Truncate', *TruncateValue*, ...) specifies the background noise model used. When *TruncateValue* is false, rmabackadj uses nontruncated Gaussian as the background noise model. Default is true.

BackAdjustedMatrix = rmabackadj(..., 'Showplot', *ShowplotValue*, ...) lets you plot a histogram showing the distribution of PM probe intensity values (blue) and the convoluted probability distribution function (red), with estimated parameters mu, sigma and alpha. When *ShowplotValue* is 'all', rmabackadj plots a histogram for each column or chip. When *ShowplotValue* is a number, list of numbers, or range of numbers, rmabackadj plots a histogram for the indicated column number (chip).

For example:

- (..., 'Showplot', 3,...) plots the intensity values in column 3 of *PMData*.
- (..., 'Showplot', [3,5,7],...) plots the intensity values in columns 3, 5, and 7 of *PMData*.
- (..., 'Showplot', 3:9,...) plots the intensity values in columns 3 to 9 of *PMData*.

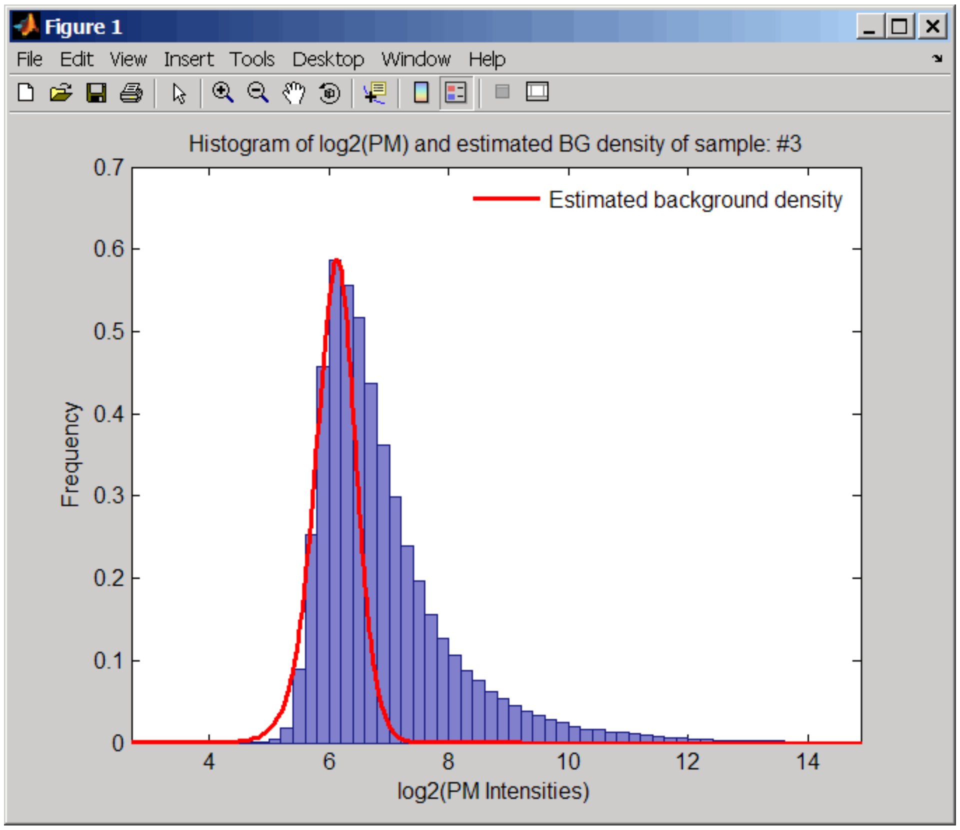


Figure 1
File Edit View Insert Tools Desktop Window Help
Histogram of log2(PM) and estimated BG density of sample: #3
Estimated background density
Frequency
0 0.1 0.2 0.3 0.4 0.5 0.6 0.7
4 6 8 10 12 14
log2(PM Intensities)

Examples

1. Load a MAT-file, included with the Bioinformatics Toolbox software, which contains Affymetrix probe-level data, including `pmMatrix`, a matrix of PM probe intensity values from multiple CEL files.

   ```
   load prostatecancerrawdata
   ```
2. Perform background adjustment on the PM probe intensity values in the matrix, `pmMatrix`, creating a new matrix, `BackgroundAdjustedMatrix`.

   ```
    BackgroundAdjustedMatrix = rmabackadj(pmMatrix);
   ```
3. Perform background adjustment on the PM probe intensity values in only column 3 of the matrix, `pmMatrix`, creating a new matrix, `BackgroundAdjustedChip3`.

   ```
    BackgroundAdjustedChip3 = rmabackadj(pmMatrix(:,3));
   ```

The `prostatecancerrawdata.mat` file used in the previous example contains data from Best et al., 2005.

Version History

Introduced in R2006a

References

[1] Irizarry, R.A., Hobbs, B., Collin, F., Beazer-Barclay, Y.D., Antonellis, K.J., Scherf, U., Speed, T.P. (2003). Exploration, Normalization, and Summaries of High Density Oligonucleotide Array Probe Level Data. Biostatistics *4*, 249–264.

[2] Bolstad, B. (2005). “affy: Built-in Processing Methods” `https://www.bioconductor.org/packages/2.1/bioc/vignettes/affy/ inst/doc/builtinMethods.pdf`

[3] Best, C.J.M., Gillespie, J.W., Yi, Y., Chandramouli, G.V.R., Perlmutter, M.A., Gathright, Y., Erickson, H.S., Georgevich, L., Tangrea, M.A., Duray, P.H., Gonzalez, S., Velasco, A., Linehan, W.M., Matusik, R.J., Price, D.K., Figg, W.D., Emmert-Buck, M.R., and Chuaqui, R.F. (2005). Molecular alterations in primary prostate cancer after androgen ablation therapy. Clinical Cancer Research *11*, 6823–6834.

See Also

`affyinvarsetnorm` | `affyread` | `affyrma` | `celintensityread` | `probelibraryinfo` | `probesetlookup` | `probesetvalues` | `quantilenorm` | `rmasummary`

rmasummary

Calculate gene expression values from Affymetrix microarray probe-level data using Robust Multi-array Average (RMA) procedure

Syntax

```
ExpressionMatrix = rmasummary(ProbeIndices, Data)

ExpressionMatrix = rmasummary(ProbeIndices, Data, 'Output', OutputValue)
```

Arguments

ProbeIndices	Column vector of probe indices. The convention for probe indices is, for each probe set, to label each probe 0 to *N* – 1, where *N* is the number of probes in the probe set. **Tip** Use the `ProbeIndices` field in the structure returned by `celintensityread` as the *`ProbeIndices`* input.
Data	Matrix of natural-scale intensity values where each row corresponds to a perfect match (PM) probe and each column corresponds to an Affymetrix CEL file. (Each CEL file is generated from a separate chip. All chips should be of the same type.) **Tip** Using a single-precision matrix for *`Data`* decreases memory usage. **Tip** You can use the matrix from the `PMIntensities` field in the structure returned by `celintensityread` as the *`Data`* input. However, first ensure the matrix has been background adjusted, using the `rmabackadj` or `gcrmabackadj`function, and normalized, using the `quantilenorm` function.
OutputValue	Specifies the scale of the returned gene expression values. *`OutputValue`* can be: • `'log'` • `'log2'` • `'log10'` • `'linear'` • `@`*`functionname`* In the last instance, the data is transformed as defined by the function *`functionname`*. Default is `'log2'`.

Description

`ExpressionMatrix` `= rmasummary(`*`ProbeIndices`*`,` *`Data`*`)` returns gene (probe set) expression values after calculating them from natural-scale probe intensities in the matrix *`Data`*, using the column vector of probe indices, *`ProbeIndices`*. Note that each row in *`Data`* corresponds to a perfect match (PM) probe, and each column corresponds to an Affymetrix CEL file. (Each CEL file is generated from a separate chip. All chips should be of the same type.) Note that the column vector *`ProbeIndices`* designates probes within each probe set by labeling each probe 0 to *N* – 1, where *N* is the number of probes in the probe set. Note that each row in *`ExpressionMatrix`* corresponds to a gene (probe set) and each column in *`ExpressionMatrix`* corresponds to an Affymetrix CEL file, which represents a single chip.

For a given probe set *n*, with *J* probe pairs, let *Yijn* denote the background-adjusted, base 2 log transformed and quantile-normalized PM probe intensity value of chip *i* and probe *j*. *Yijn* follows a linear additive model:

$Yijn = Uin + Ajn + Eijn$; $i = 1, \ldots, I$; $j = 1, \ldots, J$; $n = 1, \ldots, N$

where:

Uin = Gene expression of the probe set *n* on chip *i*

Ajn = Probe affinity effect for the *j*th probe in the probe set

Eijn = Residual for the *j*th probe on the *i*th chip

The RMA method assumes $A1 + A2 + \ldots + AJ = 0$ for all probe sets. A robust procedure, median polish, estimates *Ui* as the log scale measure of expression.

Note There is no column in *`ExpressionMatrix`* that contains probe set or gene information.

`ExpressionMatrix` `= rmasummary(...,` *`'PropertyName'`*`,` *`PropertyValue`*`, ...)` calls `rmasummary` with optional properties that use property name/property value pairs. You can specify one or more properties in any order. Each *`PropertyName`* must be enclosed in single quotation marks and is case insensitive. These property name/property value pairs are as follows:

`ExpressionMatrix` `= rmasummary(`*`ProbeIndices`*`,` *`Data`*`, 'Output',` *`OutputValue`*`)` specifies the scale of the returned gene expression values. *`OutputValue`* can be:

- `'log'`
- `'log2'`
- `'log10'`
- `'linear'`
- `@`*`functionname`*

In the last instance, the data is transformed as defined by the function *`functionname`*. Default is `'log2'`.

Examples

1 Load a MAT-file, included with the Bioinformatics Toolbox software, which contains Affymetrix data variables, including `pmMatrix`, a matrix of PM probe intensity values from multiple CEL files.

```
load prostatecancerrawdata
```

2 Perform background adjustment on the PM probe intensity values in the matrix, `pmMatrix`, using the `rmabackadj` function, thereby creating a new matrix, `BackgroundAdjustedMatrix`.

```
BackgroundAdjustedMatrix = rmabackadj(pmMatrix);
```

3 Normalize the data in `BackgroundAdjustedMatrix`, using the `quantilenorm` function.

```
NormMatrix = quantilenorm(BackgroundAdjustedMatrix);
```

4 Calculate gene expression values from the probe intensities in `NormMatrix`, creating a new matrix, `ExpressionMatrix`. (Use the `probeIndices` column vector provided to supply information on the probe indices.)

```
ExpressionMatrix = rmasummary(probeIndices, NormMatrix);
```

The `prostatecancerrawdata.mat` file used in the previous example contains data from Best et al., 2005.

Version History

Introduced in R2006a

References

[1] Irizarry, R.A., Hobbs, B., Collin, F., Beazer-Barclay, Y.D., Antonellis, K.J., Scherf, U., Speed, T.P. (2003). Exploration, Normalization, and Summaries of High Density Oligonucleotide Array Probe Level Data. Biostatistics. *4*, 249–264.

[2] Mosteller, F., and Tukey, J. (1977). Data Analysis and Regression (Reading, Massachusetts: Addison-Wesley Publishing Company), pp. 165–202.

[3] Best, C.J.M., Gillespie, J.W., Yi, Y., Chandramouli, G.V.R., Perlmutter, M.A., Gathright, Y., Erickson, H.S., Georgevich, L., Tangrea, M.A., Duray, P.H., Gonzalez, S., Velasco, A., Linehan, W.M., Matusik, R.J., Price, D.K., Figg, W.D., Emmert-Buck, M.R., and Chuaqui, R.F. (2005). Molecular alterations in primary prostate cancer after androgen ablation therapy. Clinical Cancer Research *11*, 6823–6834.

See Also

`affygcrma` | `affyinvarsetnorm` | `affyrma` | `celintensityread` | `gcrmabackadj` | `mainvarsetnorm` | `malowess` | `manorm` | `quantilenorm` | `rmabackadj`

rna2dna

Convert RNA sequence to DNA sequence

Syntax

SeqDNA = rna2dna(*SeqRNA*)

Arguments

SeqRNA	RNA sequence specified by any of the following: • Character vector or string with the characters `A`, `C`, `G`, `U`, and ambiguous characters `R`, `Y`, `K`, `M`, `S`, `W`, `B`, `D`, `H`, `V`, `N`, • Row vector of integers from the table Mapping Nucleotide Integers to Letter Codes. • MATLAB structure containing a `Sequence` field that contains an RNA sequence, such as returned by `fastaread`, `fastqread`, `emblread`, `getembl`, `genbankread`, or `getgenbank`.

Description

SeqDNA = rna2dna(*SeqRNA*) converts an RNA sequence to a DNA sequence by converting any uracil nucleotides (`U`) in the RNA sequence to thymine nucleotides (`T`). The DNA sequence is returned in the same format as the RNA sequence. For example, if *SeqRNA* is a vector of integers, then so is *SeqDNA*.

Examples

Convert an RNA sequence to a DNA sequence.

```
rna2dna('ACGAUGAGUCAUGCUU')

ans =
ACGATGAGTCATGCTT
```

Version History

Introduced before R2006a

See Also

`dna2rna` | `regexp` | `strrep`

rnaconvert

Convert secondary structure of RNA sequence between bracket and matrix notations

Syntax

`RNAStruct2` `= rnaconvert(`*`RNAStruct`*`)`

Input Arguments

`RNAStruct`	Secondary structure of an RNA sequence represented by either: • Bracket notation • Connectivity matrix **Tip** Use the `rnafold` function to create *`RNAStruct`*.

Output Arguments

`RNAStruct2`	Secondary structure of an RNA sequence represented by either: • **Bracket notation** — Character vector or string containing dots and brackets, where each dot represents an unpaired base, while a pair of equally nested, opening and closing brackets represents a base pair. • **Connectivity matrix** — Binary, upper-triangular matrix, where *`RNAmatrix`*`(i, j) = 1` if and only if the *`i`*th residue in the RNA sequence *`Seq`* is paired with the *`j`*th residue of *`Seq`*.

Description

`RNAStruct2` `= rnaconvert(`*`RNAStruct`*`)` returns *`RNAStruct2`*, the secondary structure of an RNA sequence, in matrix notation (if *`RNAStruct`* is in bracket notation), or in bracket notation (if *`RNAStruct`* is in matrix notation).

Examples

Example 1.56. Converting from Bracket to Matrix Notation

1 Create a character vector representing a secondary structure of an RNA sequence in bracket notation.

```
Bracket = '(((..((((.......)))).((.....)).))).';
```

2 Convert the secondary structure to a connectivity matrix representation.

```
Matrix = rnaconvert(Bracket);
```

Example 1.57. Converting from Matrix to Bracket Notation

1 Create a connectivity matrix representing a secondary structure of an RNA sequence.

```
    Matrix2 = zeros(12);
    Matrix2(1,12) = 1;
    Matrix2(2,11) = 1;
    Matrix2(3,10) = 1;
    Matrix2(4,9) = 1;
```

2 Convert the secondary structure to bracket notation.

```
Bracket2 = rnaconvert(Matrix2)

Bracket2 =

((((....))))
```

Version History

Introduced in R2007b

See Also

`rnafold` | `rnaplot`

rnafold

Predict minimum free-energy secondary structure of RNA sequence

Syntax

```
rnafold(Seq)
RNAbracket = rnafold(Seq)
[RNAbracket, Energy] = rnafold(Seq)
[RNAbracket, Energy, RNAmatrix] = rnafold(Seq)

... = rnafold(Seq, ...'MinLoopSize', MinLoopSizeValue, ...)
... = rnafold(Seq, ...'NoGU', NoGUValue, ...)
... = rnafold(Seq, ...'Progress', ProgressValue, ...)
```

Input Arguments

Seq	Either of the following: • Character vector or string specifying an RNA sequence. • MATLAB structure containing a `Sequence` field that specifies an RNA sequence.
MinLoopSizeValue	Integer specifying the minimum size of the loops (in bases) to be considered when computing the free energy. Default is `3`.
NoGUValue	Controls whether GU or UG pairs are forbidden to form. Choices are `true` or `false` (default).
ProgressValue	Controls the display of a progress bar during the computation of the minimum free-energy secondary structure. Choices are `true` or `false` (default).

Output Arguments

RNAbracket	Character vector of dots and brackets indicating the bracket notation for the minimum-free energy secondary structure of an RNA sequence. In the bracket notation, each dot represents an unpaired base, while a pair of equally nested, opening and closing brackets represents a base pair.
Energy	Value specifying the energy (in kcal/mol) of the minimum free-energy secondary structure of an RNA sequence.
RNAmatrix	Connectivity matrix representing the minimum free-energy secondary structure of an RNA sequence. A binary, upper-triangular matrix where `RNAmatrix(i, j) = 1` if and only if the *i*th residue in the RNA sequence *Seq* is paired with the *j*th residue of *Seq*.

Description

`rnafold(`*`Seq`*`)` predicts and displays the secondary structure (in bracket notation) associated with the minimum free energy for the RNA sequence, *`Seq`*, using the thermodynamic nearest-neighbor approach.

Note For long sequences, this prediction can be time consuming. For example, a 600-nucleotide sequence can take several minutes, and sequences greater than 1000 nucleotides can take over 1 hour, depending on your system.

`RNAbracket`` = rnafold(`*`Seq`*`)` predicts and returns the secondary structure associated with the minimum free energy for the RNA sequence, *`Seq`*, using the thermodynamic nearest-neighbor approach. The returned structure, *`RNAbracket`*, is in bracket notation, that is a vector of dots and brackets, where each dot represents an unpaired base, while a pair of equally nested, opening and closing brackets represents a base pair.

`[`*`RNAbracket, Energy`*`] = rnafold(`*`Seq`*`)` also returns *`Energy`*, the energy value (in kcal/mol) of the minimum free-energy secondary structure of the RNA sequence.

`[`*`RNAbracket, Energy, RNAmatrix`*`] = rnafold(`*`Seq`*`)` also returns *`RNAmatrix`*, a connectivity matrix representing the secondary structure associated with the minimum free energy. *`RNAmatrix`* is an upper triangular matrix where *`RNAmatrix`*`(i, j) = 1` if and only if the *`i`*th residue in the RNA sequence *`Seq`* is paired with the *`j`*th residue of *`Seq`*.

`... = rnafold(`*`Seq`*`, ...'`*`PropertyName`*`', `*`PropertyValue`*`, ...)` calls `rnafold` with optional properties that use property name/property value pairs. You can specify one or more properties in any order. Each *`PropertyName`* must be enclosed in single quotation marks and is case insensitive. These property name/property value pairs are as follows:

`...`` = rnafold(`*`Seq`*`, ...'MinLoopSize', `*`MinLoopSizeValue`*`, ...)` specifies the minimum size of the loops (in bases) to be considered when computing the free energy. Default is `3`.

`...`` = rnafold(`*`Seq`*`, ...'NoGU', `*`NoGUValue`*`, ...)` controls whether GU or UG pairs are forbidden to form. Choices are `true` or `false` (default).

`...`` = rnafold(`*`Seq`*`, ...'Progress', `*`ProgressValue`*`, ...)` controls the display of a progress bar during the computation of the minimum free-energy secondary structure. Choices are `true` or `false` (default).

Examples

Determine the minimum free-energy secondary structure (in both bracket and matrix notation) and the energy value of the following RNA sequence:

```
seq = 'ACCCCCUCCUUCCUUGGAUCAAGGGGCUCAA';
[bracket, energy, matrix] = rnafold(seq);bracket

bracket =

..(((((...((....))...))))).....
```

Version History

Introduced in R2007b

References

[1] Wuchty, S., Fontana, W., Hofacker, I., and Schuster, P. (1999). Complete suboptimal folding of RNA and the stability of secondary structures. Biopolymers *49*, 145–165.

[2] Matthews, D., Sabina, J., Zuker, M., and Turner, D. (1999). Expanded sequence dependence of thermodynamic parameters improves prediction of RNA secondary structure. J. Mol. Biol. *288*, 911–940.

See Also

`rnaconvert` | `rnaplot`

rnaplot

Draw secondary structure of RNA sequence

Syntax

```
rnaplot(RNA2ndStruct)
ha = rnaplot(RNA2ndStruct)
[ha, H] = rnaplot(RNA2ndStruct)

rnaplot(RNA2ndStruct, ...'Sequence', SequenceValue, ...)
rnaplot(RNA2ndStruct, ...'Format', FormatValue, ...)
rnaplot(RNA2ndStruct, ...'Selection', SelectionValue, ...)
rnaplot(RNA2ndStruct, ...'ColorBy', ColorByValue, ...)
```

Input Arguments

RNA2ndStruct	Secondary structure of an RNA sequence represented by either: • Character vector or string specifying bracket notation • Connectivity matrix **Tip** Use the `rnafold` function to create *RNA2ndStruct*.
SequenceValue	Sequence of the RNA secondary structure being plotted, specified by either of the following: • Character vector or string • Structure containing a `Sequence` field that contains an RNA sequence This information is used in the data tip displayed by clicking a base in the plot of the RNA secondary structure *RNA2ndStruct*. This information is required if you specify the `'Diagram'` format or if you specify to highlight any of the following paired selections: `'AU'`, `'UA'`, `'GC'`, `'CG'`, `'GU'` or `'UG'`.
FormatValue	Character vector or string specifying the format of the plot. Choices are: • `'Circle'` (default) • `'Diagram'` • `'Dotdiagram'` • `'Graph'` • `'Mountain'` • `'Tree'` **Note** If you specify `'Diagram'`, you must also use the `'Sequence'` property to provide the RNA sequence.

SelectionValue	Either of the following: • Numeric array specifying the indices of residues to highlight in the plot. • Character vector or string specifying the subset of residues to highlight in the plot. Choices are: • `'Paired'` • `'Unpaired'` • `'AU'` or `'UA'` • `'GC'` or `'CG'` • `'GU'` or `'UG'` **Note** If you specify `'AU'`, `'UA'`, `'GC'`, `'CG'`, `'GU'`, or `'UG'`, you must also use the `'Sequence'` property to provide the RNA sequence.
ColorByValue	Character vector or string specifying a color scheme for the plot. Choices are: • `'State'` (default) — Color by pair state: paired bases and unpaired bases. • `'Residue'` — Color by residue type (A, C, G, and U). • `'Pair'` — Color by pair type (AU/UA, GC/CG, and GU/UG). **Note** If you specify `'residue'` or `'pair'`, you must also use the `'Sequence'` property to provide the RNA sequence. **Note** Because internal nodes of a tree correspond to paired residues, you cannot specify `'residue'` if you specify `'Tree'` for the `'Format'` property.

Output Arguments

ha	Handle to the figure axis.

H	A structure of handles containing a subset of the following fields, based on what you specify for the `'Selection'` and `'ColorBy'` properties: • `Paired` • `Unpaired` • `A` • `C` • `G` • `U` • `AU` • `GC` • `GU` • `Selected`

Description

`rnaplot(`*`RNA2ndStruct`*`)` draws the RNA secondary structure specified by *`RNA2ndStruct`*, the secondary structure of an RNA sequence represented by a character vector or string specifying bracket notation or a connectivity matrix.

`ha`` = rnaplot(`*`RNA2ndStruct`*`)` returns *`ha`*, a handle to the figure axis.

`[`*`ha`*`, `*`H`*`] = rnaplot(`*`RNA2ndStruct`*`)` also returns *`H`*, a structure of handles, which you can use to graph elements in a MATLAB Figure window.

Tip Use the handles returned in *`H`* to change properties of the graph elements, such as color, marker size, and marker type.

`H` contains a subset of the following fields, based on what you specify for the `'Selection'` and `'ColorBy'` properties.

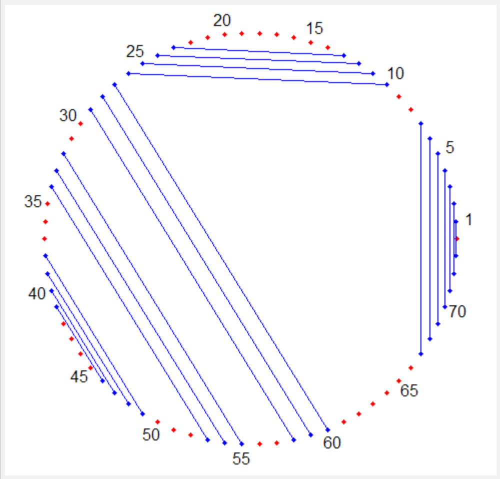
Field	Description
`Paired`	Handles to all paired residues
`Unpaired`	Handles to all unpaired residues
`A`	Handles to all A residues
`C`	Handles to all C residues
`G`	Handles to all G residues
`U`	Handles to all U residues
`AU`	Handles to all AU or UA pairs
`GC`	Handles to all GC or CG pairs
`GU`	Handles to all GU or UG pairs
`Selected`	Handles to all selected residues

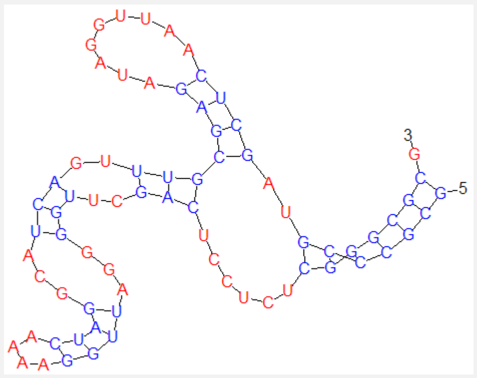
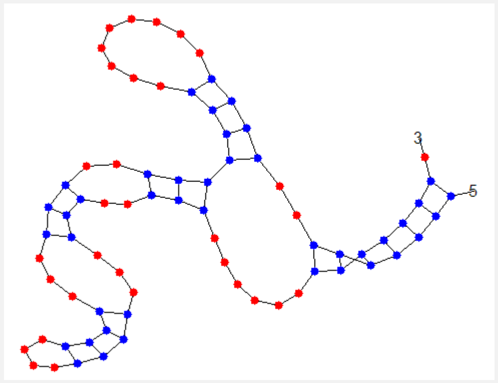
`rnaplot(`*`RNA2ndStruct`*`, ...'`*`PropertyName`*`', `*`PropertyValue`*`, ...)` calls `rnaplot` with optional properties that use property name/property value pairs. You can specify one or more properties in any order. Each *`PropertyName`* must be enclosed in single quotation marks and is case insensitive. These property name/property value pairs are as follows:

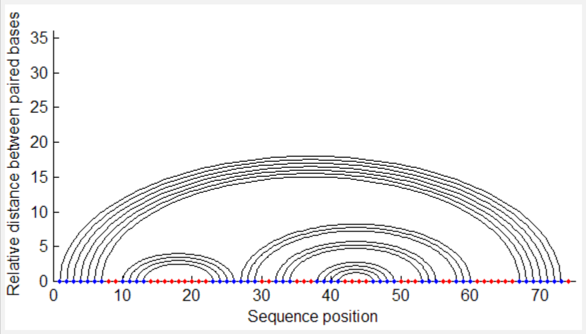
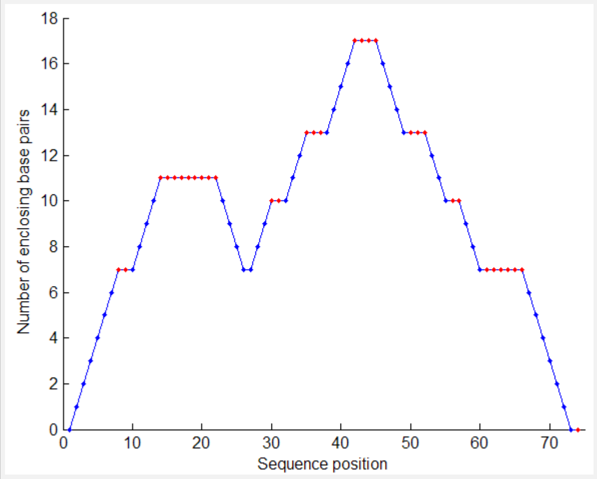
`rnaplot(`*`RNA2ndStruct`*`, ...'Sequence', `*`SequenceValue`*`, ...)` draws the RNA secondary structure specified by *`RNA2ndStruct`*, and annotates it with the sequence positions supplied by *`SequenceValue`*, the RNA sequence specified by a character vector, string, or a structure containing a `Sequence` field.

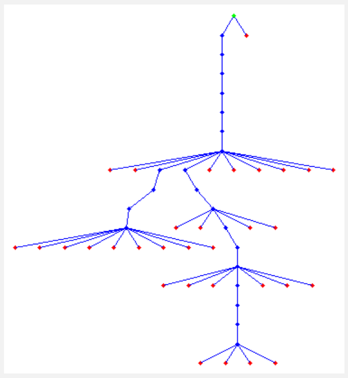
`rnaplot(`*`RNA2ndStruct`*`, ...'Format', `*`FormatValue`*`, ...)` draws the RNA secondary structure specified by *`RNA2ndStruct`*, using the format specified by *`FormatValue`*.

`FormatValue` is a character vector or string specifying the format of the plot. Choices are as follows.

Format	Description
`'Circle'` (default)	Each base is represented by a dot on the circumference of a circle of arbitrary size. Lines connect bases that pair with each other.  20 15 25 10 30 5 35 1 40 70 45 65 50 55 60

Format	Description
`'Diagram'`	Two-dimensional representation of the RNA secondary structure. Each base is represented and identified by a letter. The backbone and hydrogen bonds between base pairs are represented by lines.  **Note** If you specify `'Diagram'`, you must also use the `'Sequence'` property to provide the RNA sequence.
`'Dotdiagram'`	Two-dimensional representation of the RNA secondary structure. Each base is represented and identified by a dot. The backbone and hydrogen bonds between base pairs are represented by lines. 

Format	Description
'Graph'	Bases are displayed in their sequence position along the abscissa (*x*-axis) of a graph. Semi-elliptical lines connect bases that pair with each other. The height of the lines is proportional to the distance between paired bases. 
'Mountain'	Each base is represented by a dot in a two-dimensional plot, where the base position is in the abscissa (*x*-axis) and the number of base pairs enclosing a given base is in the ordinate (*y*-axis). 

Format	Description
`'Tree'`	Each base is represented by a node in a tree graph. Leaf nodes indicate unpaired bases, while each internal node indicates a base pair. The tree root is a fictitious node, not associated with any base in the secondary structure. 

`rnaplot(`*`RNA2ndStruct`*`, ...'Selection', `*`SelectionValue`*`, ...)` draws the RNA secondary structure specified by *`RNA2ndStruct`*, highlighting a subset of residues specified by *`SelectionValue`*. *`SelectionValue`* can be either:

- Numeric array specifying the indices of residues to highlight in the plot.
- Character vector or string specifying the subset of residues to highlight in the plot. Choices are:
  - `'Paired'`
  - `'Unpaired'`
  - `'AU'` or `'UA'`
  - `'GC'` or `'CG'`
  - `'GU'` or `'UG'`

Note If you specify `'AU'`, `'UA'`, `'GC'`, `'CG'`, `'GU'`, or `'UG'`, you must also use the `'Sequence'` property to provide the RNA sequence.

`rnaplot(`*`RNA2ndStruct`*`, ...'ColorBy', `*`ColorByValue`*`, ...)` draws the RNA secondary structure specified by *`RNA2ndStruct`*, using a color scheme specified by *`ColorByValue`*, a character vector or string indicating a color scheme. Choices are:

- `'State'` (default) — Color by pair state: paired bases and unpaired bases.
- `'Residue'` — Color by residue type (A, C, G, and U).
- `'Pair'` — Color by pair type (AU/UA, GC/CG, and GU/UG).

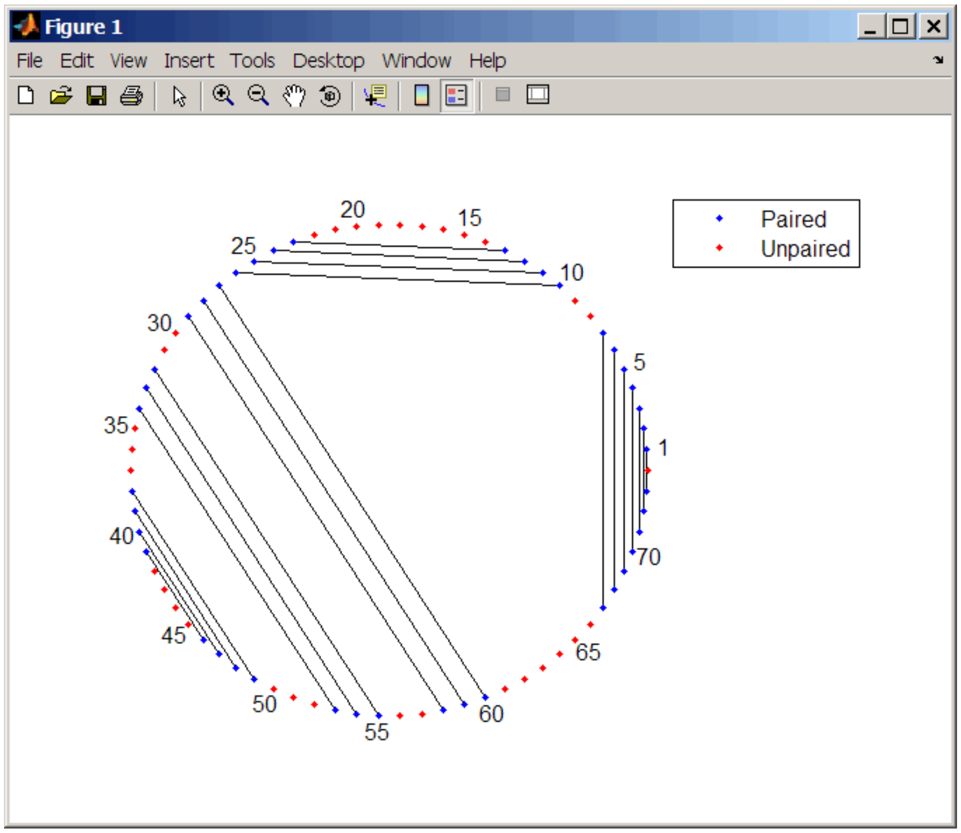
Note If you specify `'Residue'` or `'Pair'`, you must also use the `'Sequence'` property to provide the RNA sequence.

Note Because internal nodes of a tree correspond to paired residues, you cannot specify `'Residue'` if you specify `'Tree'` for the `'Format'` property.

Examples

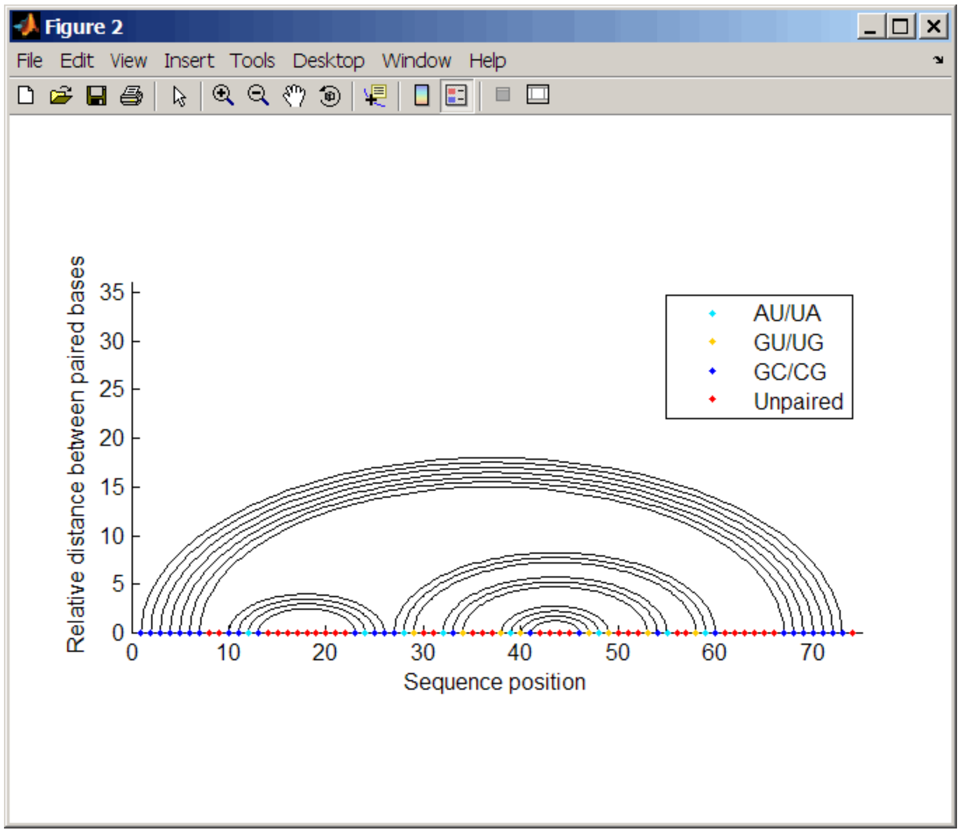
1 Determine the minimum free-energy secondary structure of an RNA sequence and plot it in circle format:

```
seq = 'GCGCCCGUAGCUCAAUUGGAUAGAGCGUUUGACUACGGAUCAAAAGGUUAGGGGUUCGACUCCUCUCGGGCGCG';
ss = rnafold(seq);
rnaplot(ss)
```



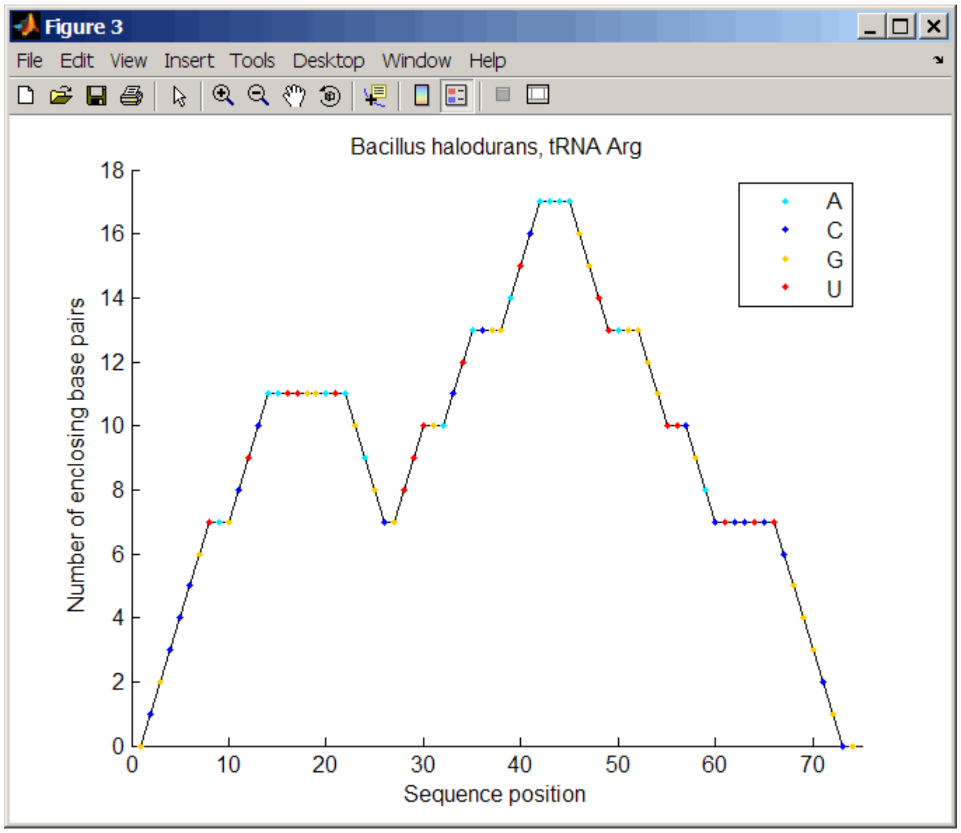
2 Plot the RNA sequence secondary structure in graph format and color it by pair type.

```
rnaplot(ss, 'sequence', seq, 'format', 'graph', 'colorby', 'pair')
```



3 Plot the RNA sequence secondary structure in mountain format and color it by residue type. Use the handle to add a title to the plot.

```
ha = rnaplot(ss, 'sequence', seq, 'format', 'mountain',...
             'colorby', 'residue')
title(ha, 'Bacillus halodurans, tRNA Arg')
```



4 Mutate the first six positions in the sequence and observe the effect the change has on the secondary structure by highlighting the first six residues.

```
seqMut = seq;
seqMut(1:6) = 'AAAAAA';
ssMut = rnafold(seqMut);
rnaplot(ss, 'sequence', seq, 'format', 'dotdiagram', 'selection', 1:6);
rnaplot(ssMut, 'sequence', seqMut, 'format', 'dotdiagram', 'selection', 1:6);
```

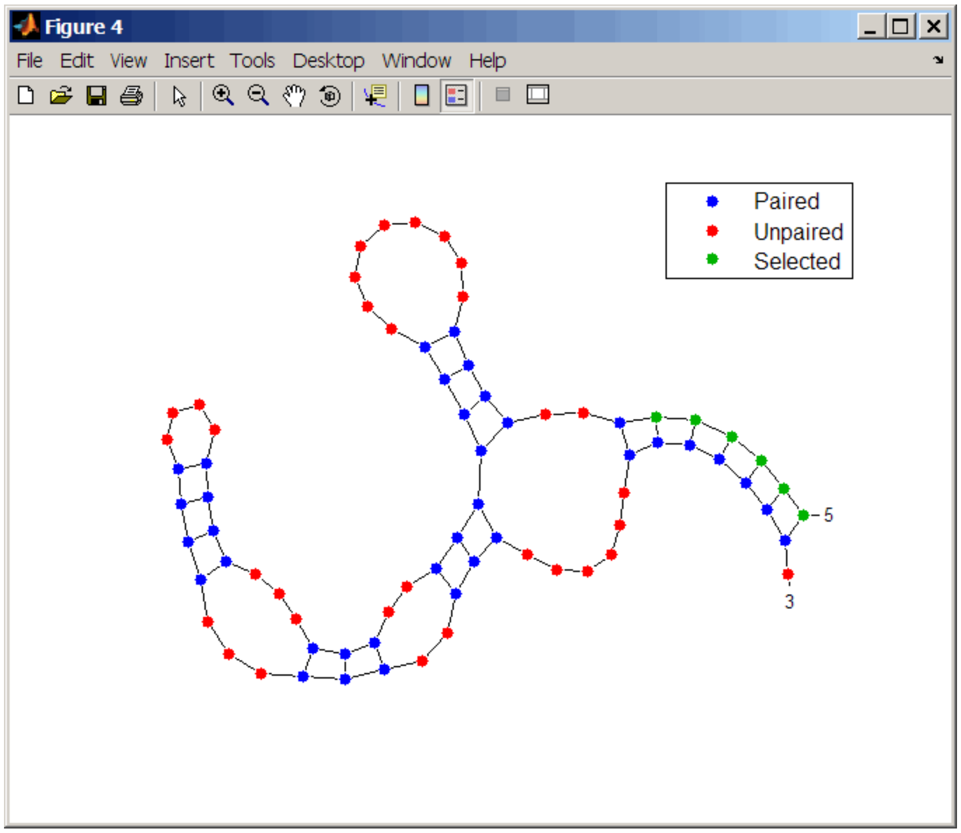
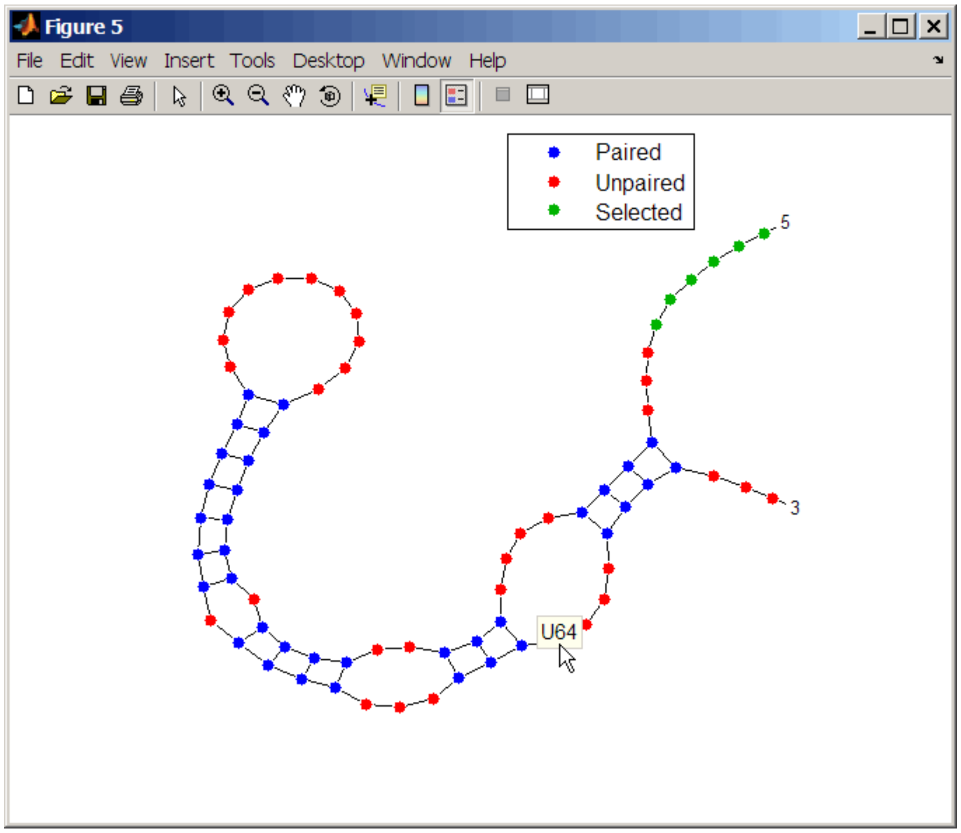


Figure 4
File Edit View Insert Tools Desktop Window Help
Paired
Unpaired
Selected
5
3



Tip If necessary, click-drag the legend to prevent it from covering the plot. Click a base in the plot to display a data tip with information on that base.

Version History

Introduced in R2007b

See Also

`rnaconvert` | `rnafold`

rnaseqde

Perform differential expression analysis on RNA-seq count data

Syntax

```
diffTable = rnaseqde(countTable,conditionVariables1,conditionVariables2)
diffTable = rnaseqde(___,Name=Value)
```

Description

`diffTable = rnaseqde(countTable,conditionVariables1,conditionVariables2)` uses the RNA-seq count data in `countTable` and performs differential expression analysis between the conditions (or samples) in `conditionVariables1` and `conditionVariables2`. The rows of `countTable` represent genes (or features), and the columns represent the conditions. The function first divides each condition by a library size factor to normalize the count prior to performing the hypothesis test. This size factor is equal to the median ratio of each feature over the geometric mean of the feature in all conditions. The function uses an exact test to determine the differences between two groups of counts that are each assumed to follow a negative binomial distribution [1].

`diffTable = rnaseqde(___,Name=Value)` specifies additional options using one or more name-value arguments.

Examples

Perform Differential Analysis on RNA-Seq Data

Use RNA-seq count data that consists of two biological replicates of the control (untreated) samples and two biological replicates of the knock-down (treated) samples [6]. Load the table with read counts for genes.

```
load("pasilla_count_noMM.mat","geneCountTable")
```

Display the first few rows of the table.

```
head(geneCountTable,10)
```

```
         ID           Reference    untreated3    untreated4    treated2    treated3
    _____________    _________    __________    __________    ________    ________

    "FBgn0000003"      "3R"              0             1            1           2
    "FBgn0000008"      "2R"            142           117          138         132
    "FBgn0000014"      "3R"             20            12           10          19
    "FBgn0000015"      "3R"              2             4            0           1
    "FBgn0000017"      "3L"           6591          5127         4809        6027
    "FBgn0000018"      "2L"            469           530          492         574
    "FBgn0000024"      "3R"              5             6           10           8
    "FBgn0000028"      "X"               0             0            2           1
    "FBgn0000032"      "3R"           1160          1143         1138        1415
    "FBgn0000036"      "3R"              0             0            0           1
```

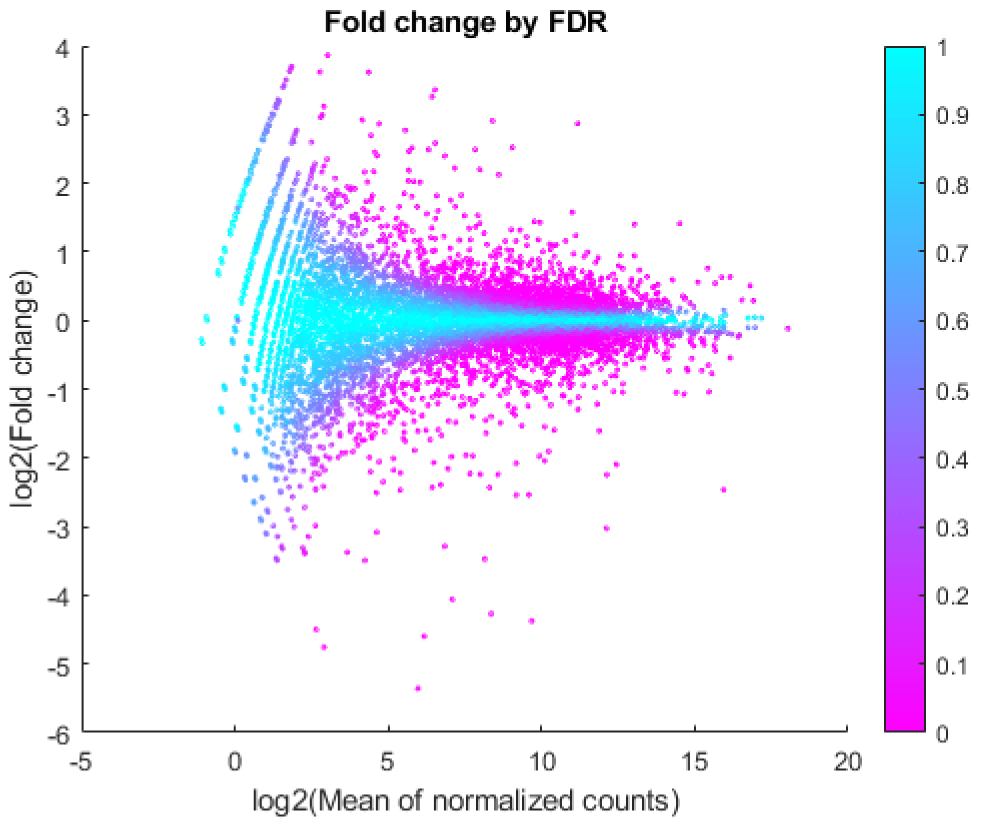
Perform the differential analysis of the control and treated samples using the read count data for genes. Specify both replicates for each condition. The input `geneCountTable` has an `ID` column. Optionally, you can append this column to the output table by using `IDColumns` .

```
diffTable = rnaseqde(geneCountTable,["untreated3", "untreated4"],...
                     ["treated2", "treated3"],IDColumns="ID");
head(diffTable,5)
```

```
          ID             Mean1       Mean2      Log2FoldChange      PValue        AdjustedPValue
    _____________       _______     _______    ______________     _________      ______________

    "FBgn0000003"       0.51415      1.3808          1.4253               1                  1
    "FBgn0000008"         135.9      129.48       -0.069799         0.67298            0.89231
    "FBgn0000014"        16.838      13.384        -0.33119          0.6421            0.87234
    "FBgn0000015"        3.1234     0.42413         -2.8806         0.22776            0.57215
    "FBgn0000017"        6151.8      5117.4        -0.26559       0.0014429           0.014207
```

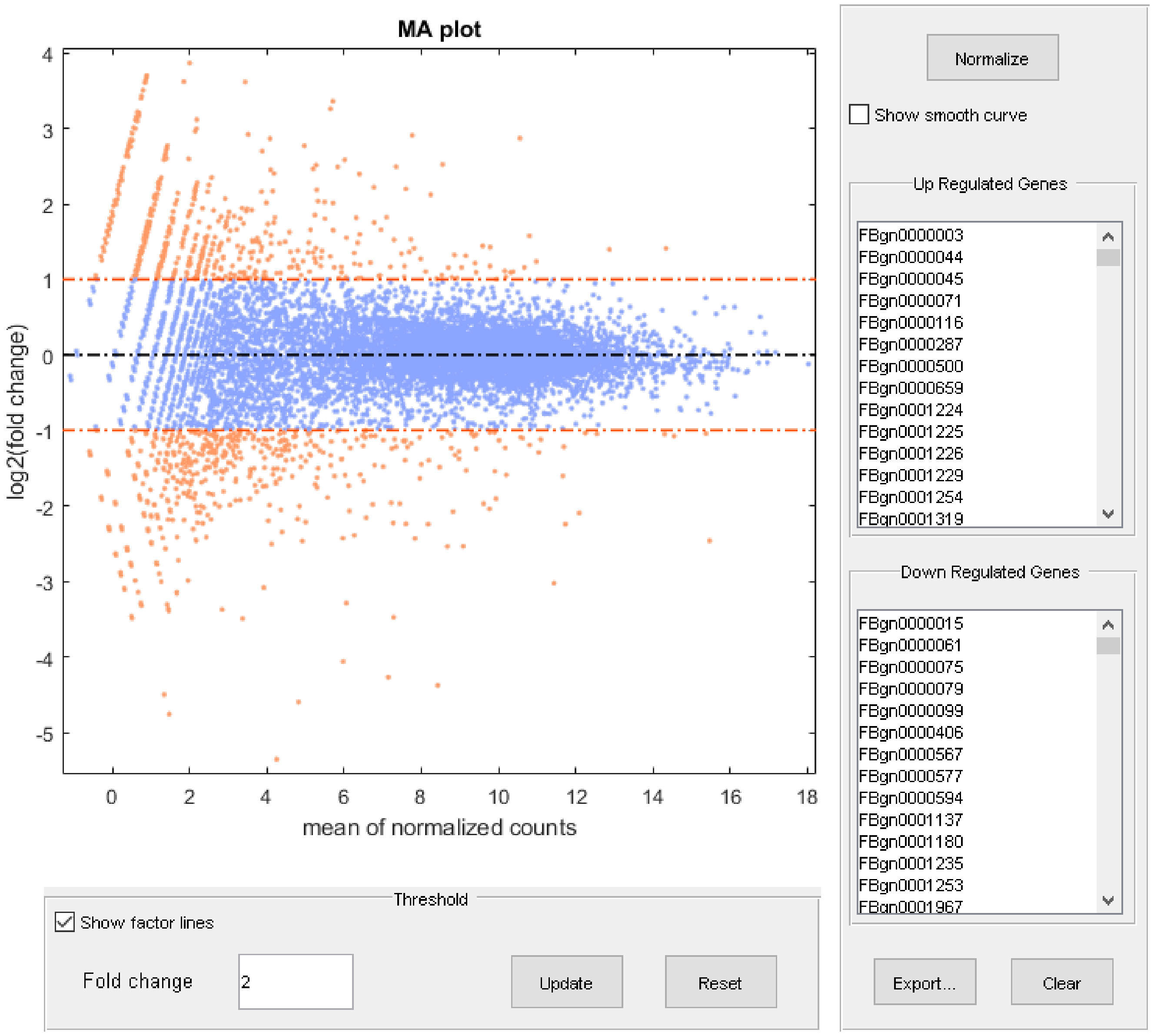
Look at the difference in gene expression between two conditions by displaying the log2 fold change for each gene.

```
figure
scatter(log2(mean([diffTable.Mean1,diffTable.Mean2], 2)),diffTable.Log2FoldChange,3,diffTable.Ad
colormap(flipud(cool(256)))
colorbar;
ylabel("log2(Fold change)")
xlabel("log2(Mean of normalized counts)")
title("Fold change by FDR")
```



You can also annotate the values in the plot with the corresponding gene names, interactively select genes, and export gene lists to the workspace.

```
warnSettings = warning('off','bioinfo:mairplot:ZeroValues');
mairplot(diffTable.Mean2,diffTable.Mean1,Labels=geneCountTable.ID,Type="MA");
set(get(gca,"Xlabel"),"String","mean of normalized counts")
set(get(gca,"Ylabel"),"String","log2(fold change)")
```



Show factor lines ✓
Fold change: 2

Up Regulated Genes:
FBgn0000003
FBgn0000044
FBgn0000045
FBgn0000071
FBgn0000116
FBgn0000287
FBgn0000500
FBgn0000659
FBgn0001224
FBgn0001225
FBgn0001226
FBgn0001229
FBgn0001254
FBgn0001319

Down Regulated Genes:
FBgn0000015
FBgn0000061
FBgn0000075
FBgn0000079
FBgn0000099
FBgn0000406
FBgn0000567
FBgn0000577
FBgn0000594
FBgn0001137
FBgn0001180
FBgn0001235
FBgn0001253
FBgn0001967

```
warning(warnSettings);
```

Input Arguments

`countTable` — RNA-seq count data
table

RNA-seq count data, specified as a table. It contains counts of genomic features for all replicates of each condition.

Data Types: `table`

`conditionVariables1` — Table variables for first condition
cell array of character vectors | string vector | numeric vector

Table variables for the first condition, specified as a cell array of character vectors, string vector, or numeric vector. You can specify a list of table variable names of the corresponding columns in `countTable` or a numeric vector containing the indices of the columns.

Data Types: `cell` | `double` | `string`

`conditionVariables2` — Table variables for second condition
cell array of character vectors | string vector | numeric vector

Table variables for the second condition, specified as a cell array of character vectors, string vector, or numeric vector. You can specify a list of table variable names of the corresponding columns in `countTable` or a numeric vector containing the indices of the columns.

Data Types: `cell` | `double` | `string`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `dt = rnaseqde(ct,[1 3],[4 6],FDRMethod="storey")` specifies to use the Storey procedure as the false discovery rate method.

`VarianceLink` — Linkage type between variance and mean
`"local"` (default) | `"constant"` | `"identity"`

Linkage type between the variance and mean, specified as a character vector or string. This table summarizes the available linkage options.

Linkage option	Description	When to use
`"local"`	The variance is the sum of the shot noise term (mean) and a locally regressed nonparametric smooth function of the mean as described in [1]. This option is the default.	Use this option if your data is overdispersed and has more than 1000 rows (genes).

Linkage option	Description	When to use
`"constant"`	The variance is the sum of the shot noise term (mean) and a constant multiplied by the squared mean as described in [2]. This method uses all the rows in the data to estimate the constant.	Use this option if your data is overdispersed and has less than 1000 rows.
`"identity"`	The variance is equal to the mean as described in [3]. Counts are therefore modeled by the Poisson distribution individually for each row of the conditions.	Use this option if your data has few genes and the regression between the variance and mean is not possible because of a very small number of samples or replicates. This option is not recommended for overdispersed data.

`FDRMethod` — False discovery rate method to adjust p-values
`"bh"` (default) | `"storey"`

False discovery rate method to adjust p-values, specified as `"bh"` or `"storey"`. By default, the function uses `"bh"`, which is the linear step-up procedure introduced by Benjamini and Hochberg [4]. The `"storey"` method adjusts p-values using the procedure introduced by Storey [5].

`IDColumns` — Columns from input table to append to output table
character vector | string | string vector | cell array of character vectors | numeric vector

Columns from the input `countTable` to append to the output `diffTable`, specified as a character vector, string, string vector, cell array of character vectors, or numeric vector. You can specify the column names or a numeric vector containing the indices of the columns in `countTable`. The function appends the columns to the left side of the output table.

Data Types: `double` | `char` | `string` | `cell`

Output Arguments

`diffTable` — Differential expression analysis results
table

Differential expression analysis results, returned as a table with these columns:

- `Mean1` — Mean normalized counts for the samples specified in `conditionVariables1`
- `Mean2` — Mean normalized counts for the samples specified in `conditionVariables2`
- `Log2FoldChange` — Log2 ratio of `Mean2` over `Mean1`
- `PValue` — P-value output from the hypothesis test
- `AdjustedPValue` — Adjusted p-value, calculated using the method specified in `'FDRMethod'`

Version History

Introduced in R2021b

References

[1] Anders, Simon, and Wolfgang Huber. “Differential Expression Analysis for Sequence Count Data.” *Genome Biology* 11, no. 10 (October 2010): R106. https://doi.org/10.1186/gb-2010-11-10-r106.

[2] Robinson, Mark D., and Gordon K. Smyth. “Small-Sample Estimation of Negative Binomial Dispersion, with Applications to SAGE Data.” *Biostatistics* 9, no. 2 (July 11, 2007): 321–32.

[3] Marioni, J. C., C. E. Mason, S. M. Mane, M. Stephens, and Y. Gilad. “RNA-Seq: An Assessment of Technical Reproducibility and Comparison with Gene Expression Arrays.” *Genome Research* 18, no. 9 (July 30, 2008): 1509–17.

[4] Benjamini, Y., and Hochberg, Y. 1995. Controlling the false discovery rate: A practical and powerful approach to multiple testing. J. Royal Stat. Soc. 57:289–300.

[5] Storey, John D. “A Direct Approach to False Discovery Rates.” *Journal of the Royal Statistical Society: Series B (Statistical Methodology)* 64, no. 3 (August 2002): 479–98.

[6] Brooks, A. N., L. Yang, M. O. Duff, K. D. Hansen, J. W. Park, S. Dudoit, S. E. Brenner, and B. R. Graveley. “Conservation of an RNA Regulatory Map between Drosophila and Mammals.” *Genome Research* 21, no. 2 (February 1, 2011): 193–202.

See Also

`nbintest` | `mairplot`

Topics

“Identifying Differentially Expressed Genes from RNA-Seq Data”
“Negative Binomial Distribution”

rownames (DataMatrix)

Retrieve or set row names of DataMatrix object

Syntax

```
ReturnRowNames = rownames(DMObj)
ReturnRowNames = rownames(DMObj, RowIndices)
DMObjNew = rownames(DMObj, RowIndices, RowNames)
```

Input Arguments

DMObj	DataMatrix object, such as created by `DataMatrix` (object constructor).
RowIndices	One or more rows in *DMObj*, specified by any of the following: • Positive integer • Vector of positive integers • Character vector specifying a row name • Cell array of character vectors • Logical vector
RowNames	Row names specified by any of the following: • Numeric vector • Cell array of character vectors • Character array • Single character vector, which is used as a prefix for row names, with row numbers appended to the prefix • Logical `true` or `false` (default). If `true`, unique row names are assigned using the format `row1`, `row2`, `row3`, etc. If `false`, no row names are assigned. **Note** The number of elements in *RowNames* must equal the number of elements in *RowIndices*.

Output Arguments

ReturnRowNames	Character vector or cell array of character vectors containing row names in *DMObj*.
DMObjNew	DataMatrix object created with names specified by *RowIndices* and *RowNames*.

Description

ReturnRowNames = `rownames(`*DMObj*`)` returns *ReturnRowNames*, a cell array of character vectors specifying the row names in *DMObj*, a DataMatrix object.

ReturnRowNames = rownames(*DMObj*, *RowIndices*) returns the row names specified by *RowIndices*. *RowIndices* can be a positive integer, vector of positive integers, character vector specifying a row name, cell array of character vectors, or a logical vector.

DMObjNew = rownames(*DMObj*, *RowIndices*, *RowNames*) returns *DMObjNew*, a DataMatrix object with rows specified by *RowIndices* set to the names specified by *RowNames*. The number of elements in *RowIndices* must equal the number of elements in *RowNames*.

Version History

Introduced in R2008b

See Also

`DataMatrix` | `colnames`

Topics

DataMatrix object on page 1-734

run

Map sequence reads to reference sequence using Bowtie 2

Syntax

```
run(object,indexBaseName,reads1,reads2,outputFileName)
run( ___ ,'IncludeAll',TF)
flag = run( ___ )
```

Description

`run(object,indexBaseName,reads1,reads2,outputFileName)` maps the sequencing reads from `reads1` and `reads2` against the reference sequence and writes the results to the output file `outputFileName`. The input `indexBaseName` represents the base name (prefix) of the reference index files. `object` is a `Bowtie2AlignOptions` object.

`run` requires the Bowtie 2 Support Package for Bioinformatics Toolbox. If this support package is not installed, then the function provides a download link. For details, see “Bioinformatics Toolbox Software Support Packages”.

`run( ___ ,'IncludeAll',TF)` specifies whether to use all object properties and their corresponding values when running `bowtie2`. Specify this option after all other input arguments. By default, only the modified properties are used to run `bowtie2`.

`flag = run( ___ )` returns an exit `flag` of the function using any of the input arguments in the previous syntaxes.

Examples

Align Reads to Reference Sequence Using Bowtie 2

First build a set of index files for the Drosophila genome. An error message appears if you do not have the Bioinformatics Toolbox Interface for Bowtie Aligner support package installed when you run the function. Click the provided link to download the package from the Add-on menu.

For this example, the reference sequence `Dmel_chr4.fa` is already provided with the toolbox.

```
status = bowtie2build('Dmel_chr4.fa', 'Dmel_chr4_index');
```

If the index build is successful, the function returns `0` and creates the index files (`*.bt2`) in the current folder. The files have the prefix `'Dmel_chr4_index'`.

Once the index is ready, map the read sequences to the reference. The pair-end read files (`SRR6008575_10k_1.fq` and `SRR6008575_10k_2.fq`) are already provided with the toolbox.

Create an options object.

```
 alignOpt = Bowtie2AlignOptions;
```

Trim four residues from the `3'` end before aligning.

```
 alignOpt.Trim3 = 4;
```

Map reads to the reference using the specified alignment option.

```
flag = run(alignOpt,'Dmel_chr4','SRR6008575_10k_1.fq','SRR6008575_10k_2.fq','SRR6008575_10k_chr4_
```

The output is a SAM-formatted file that contains the mapping results.

Input Arguments

`object` — Alignment options

`Bowtie2AlignOptions` object

Alignment options, specified as a `Bowtie2AlignOptions` object.

Example: `'alignOpt'`

`indexBaseName` — Base name of reference index files

character vector | string

Base name (prefix) of the reference index files, specified as a character vector or string. The index files are in the `BT2` or `BT21` format.

Example: `'Dmel_chr4'`

Data Types: `char` | `string`

`reads1` — Names of files with first mate reads

string array | cell array of character vectors

Names of files with the first mate reads or single-end reads, specified as a string array or cell array of character vectors.

For paired-end data, sequences in `reads1` must correspond file-for-file and read-for-read to sequences in `reads2`.

Example: `'SRR6008575_10k_1.fq'`

Data Types: `char` | `string`

`reads2` — Names of files with second mate reads

string array | cell array of character vectors

Names of files with the second mate reads, specified as a string array or cell array of character vectors.

Specify `reads2` as an empty character vector or string (`''` or `""`) if the data consists of single-end reads only.

Example: `'SRR6008575_10k_2.fq'`

Data Types: `char` | `string`

`outputFileName` — Output file name

character vector | string

Output file name, specified as a character vector or string. This file contains the mapping results.

Example: `'SRR6008575_10k_chr4.sam'`

Data Types: `char` | `string`

TF — Flag to use all object properties and their corresponding values
`false` (default) | `true`

Flag to use all object properties and their corresponding values when you run the function, specified as `true` or `false`. By default, only the modified properties are used.

Example: `true`

Output Arguments

`flag` — Exit status
integer

Exit status of the function, returned as an integer. `flag` is `0` if the function runs without errors or warning. Otherwise, it is nonzero.

Version History

Introduced in R2018a

References

[1] Langmead, B., and S. Salzberg. "Fast gapped-read alignment with Bowtie 2." *Nature Methods*. 9, 2012, 357–359.

See Also

`bowtie2` | `bowtie2inspect` | `bowtie2build` | `Bowtie2AlignOptions` | `Bowtie2BuildOptions` | `Bowtie2InspectOptions`

External Websites
Bowtie 2 manual

run

Build Bowtie 2 index files

Syntax

```
run(object,referenceFileNames,indexBaseName)
run(___,'IncludeAll',TF)
flag = run(___)
```

Description

`run(object,referenceFileNames,indexBaseName)` builds Bowtie 2 index files from the reference sequence information saved in the FASTA files specified by `referenceFileNames`.

`run` requires the Bowtie 2 Support Package for Bioinformatics Toolbox. If this support package is not installed, then the function provides a download link. For details, see "Bioinformatics Toolbox Software Support Packages".

`run(___,'IncludeAll',TF)` specifies whether to use all object properties and their corresponding values when running `bowtie2build`. Specify this option after all other input arguments. By default, only the modified properties are used to run the function.

`flag = run(___)` returns an exit `flag` of the function using any of the input arguments in the previous syntaxes.

Examples

Build Bowtie 2 Index Files for Reference Sequence

Build a set of index files for the Drosophila genome. An error message appears if you do not have the Bioinformatics Toolbox Interface for Bowtie Aligner support package installed when you run the function. Click the provided link to download the package from the Add-on menu.

Create an options object.

```
buildOpt = Bowtie2BuildOptions;
```

Build the index files using the `run` function. For this example, the reference sequence `Dmel_chr4.fa` is already provided with the toolbox.

```
flag = run(buildOpt,'Dmel_chr4.fa', 'Dmel_chr4_index');
```

If the index build is successful, the function returns `0` and creates the index files (`*.bt2`) in the current folder. The files have the prefix `'Dmel_chr4_index'`.

Input Arguments

`object` — Options to build index files
`Bowtie2BuildOptions` object

Options to build the index files, specified as a `Bowtie2BuildOptions` object.

Example: `'buildOpt'`

`referenceFileNames` — Names of files with reference sequence information
string | character vector | string array | cell array of character vectors

Names of files with reference sequence information, specified as a string, character vector, string array, or cell array of character vectors.

Example: `'Dmel_chr4.fa'`

Data Types: `char` | `string` | `cell`

`indexBaseName` — Base name of reference index files
character vector | string

Base name (prefix) of the reference index files, specified as a character vector or string. The index files are in the `BT2` or `BT21` format.

Example: `'Dmel_chr4'`

Data Types: `char` | `string`

TF — Flag to use all object properties and their corresponding values
`false` (default) | `true`

Flag to use all object properties and their corresponding values when you run the function, specified as `true` or `false`. By default, only the modified properties are used.

Example: `true`

Output Arguments

`flag` — Exit status
integer

Exit status of the function, returned as an integer. `flag` is `0` if the function runs without errors or warning. Otherwise, it is nonzero.

Version History

Introduced in R2018a

References

[1] Langmead, B., and S. Salzberg. "Fast gapped-read alignment with Bowtie 2." *Nature Methods*. 9, 2012, 357–359.

See Also

`bowtie2` | `bowtie2inspect` | `bowtie2build` | `Bowtie2AlignOptions` | `Bowtie2BuildOptions` | `Bowtie2InspectOptions`

External Websites
Bowtie 2 manual

run

Inspect Bowtie 2 index files

Syntax

```
run(object,indexBaseName,outputFileName)
run( ___ ,'IncludeAll',TF)
flag = run( ___ )
```

Description

`run(object,indexBaseName,outputFileName)` inspects Bowtie 2 index files with the prefix `indexBaseName`, checks the original reference sequences used to build the index, and saves the reference sequences in the output file `outputFileName`. The argument `object` is a `Bowtie2InspectOptions` object.

`run` requires the Bowtie 2 Support Package for Bioinformatics Toolbox. If this support package is not installed, then the function provides a download link. For details, see “Bioinformatics Toolbox Software Support Packages”.

`run( ___ ,'IncludeAll',TF)` specifies whether to use all object properties and their corresponding values when you run `bowtie2inspect`. Specify this option after all other input arguments. By default, only the modified properties are used to run `bowtie2inspect`.

`flag = run( ___ )` returns an exit `flag` of the function using any of the input arguments in the previous syntaxes.

Examples

Inspect Bowtie 2 Index and Retrieve Reference Sequence Information

Get information about the reference sequence used to build the corresponding index files. An error message appears if you do not have the Bioinformatics Toolbox Interface for Bowtie Aligner support package installed when you run the function. Click the provided link to download the package from the Add-on menu.

Create an options object.

```
inspectOpt = Bowtie2InspectOptions;
```

Inspect the index and get information about the reference sequence. By default, the output file `Dmel_chr4_retrieved.fa` contains the actual sequence of the reference used to build the index. If the function runs without any warning or error, it returns `0`.

```
flag = run(inspectOpt,'Dmel_chr4', 'Dmel_chr4_retrieved.fa');
```

Input Arguments

`object` — Options to inspect index files
`Bowtie2InspectOptions` object

Options to inspect the index files, specified as a `Bowtie2InspectOptions` object.

Example: `'inspectOpt'`

`indexBaseName` — Base name of reference index files
character vector | string

Base name (prefix) of the reference index files, specified as a character vector or string. The index files are in the `BT2` or `BT21` format.

Example: `'Dmel_chr4'`

Data Types: `char` | `string`

`outputFileName` — Name of output file
string | character vector

Name of an output file, specified as a string or character vector. By default, the output file contains the reference sequences that are used to build the index files.

Example: `'refSeq.fa'`

Data Types: `char` | `string`

`TF` — Flag to use all object properties and their corresponding values
`false` (default) | `true`

Flag to use all object properties and their corresponding values when you run the function, specified as `true` or `false`. By default, only the modified properties are used.

Example: `true`

Output Arguments

`flag` — Exit status
integer

Exit status of the function, returned as an integer. `flag` is `0` if the function runs without errors or warning. Otherwise, it is nonzero.

Version History

Introduced in R2018a

References

[1] Langmead, B., and S. Salzberg. "Fast gapped-read alignment with Bowtie 2." *Nature Methods*. 9, 2012, 357–359.

See Also

`bowtie2` | `bowtie2inspect` | `bowtie2build` | `Bowtie2AlignOptions` | `Bowtie2BuildOptions` | `Bowtie2InspectOptions`

External Websites

Bowtie 2 manual

run

Package: bioinfo.pipeline

Run pipeline

Syntax

```
run(pipeline)
run(pipeline,inputStruct)
run( ___ ,Name=Value)
```

Description

run(pipeline) runs the pipeline. The pipeline must have all the input ports satisfied on page 1-1648.

run(pipeline,inputStruct) runs the pipeline using the structure inputStruct as an input. This syntax is one of three ways on page 1-1648 to satisfy input ports by matching the field names of inputStruct to unconnected inport port names in the pipeline.

run(___ ,Name=Value) uses additional options specified by one or more name-value arguments for any of the above syntaxes.

Examples

Create a Simple Pipeline to Plot Sequence Quality Data

Import the Pipeline and block objects needed for the example.

```
import bioinfo.pipeline.Pipeline
import bioinfo.pipeline.blocks.*
```

Create a pipeline.

```
qcpipeline = Pipeline;
```

Select an input FASTQ file using a FileChooser block.

```
fastqfile = FileChooser(which("SRR005164_1_50.fastq"));
```

Create a SeqFilter block.

```
sequencefilter = SeqFilter;
```

Define the filtering threshold value. Specifically, filter out sequences with a total of more than 10 low-quality bases, where a base is considered a low-quality base if its quality score is less than 20.

```
sequencefilter.Options.Threshold = [10 20];
```

Add the blocks to the pipeline.

```
addBlock(qcpipeline,[fastqfile,sequencefilter]);
```

Connect the output of the first block to the input of the second block. To do so, you need to first check the input and output port names of the corresponding blocks.

View the `Outputs` (port of the first block) and `Inputs` (port of the second block).

```
fastqfile.Outputs
```

```
ans = struct with fields:
    Files: [1×1 bioinfo.pipeline.Output]
```

```
sequencefilter.Inputs
```

```
ans = struct with fields:
    FASTQFiles: [1×1 bioinfo.pipeline.Input]
```

Connect the `Files` output port of the `fastqfile` block to the `FASTQFiles` port of `sequencefilter` block.

```
connect(qcpipeline,fastqfile,sequencefilter,["Files","FASTQFiles"]);
```

Next, create a `UserFunction` block that calls the `seqqcplot` function to plot the quality data of the filtered sequence data. In this case, `inputFile` is the required argument for the `seqqcplot` function. The required argument name can be anything as long as it is a valid variable name.

```
qcplot = UserFunction("seqqcplot",RequiredArguments="inputFile",OutputArguments="figureHandle");
```

Alternatively, you can also use dot notation to set up your `UserFunction` block.

```
qcplot = UserFunction;
qcplot.RequiredArguments = "inputFile";
qcplot.Function = "seqqcplot";
qcplot.OutputArguments = "figureHandle";
```

Add the block.

```
addBlock(qcpipeline,qcplot);
```

Check the port names of `sequencefilter` block and `qcplot` block.

```
sequencefilter.Outputs
```

```
ans = struct with fields:
    FilteredFASTQFiles: [1×1 bioinfo.pipeline.Output]
         NumFilteredIn: [1×1 bioinfo.pipeline.Output]
        NumFilteredOut: [1×1 bioinfo.pipeline.Output]
```

```
qcplot.Inputs
```

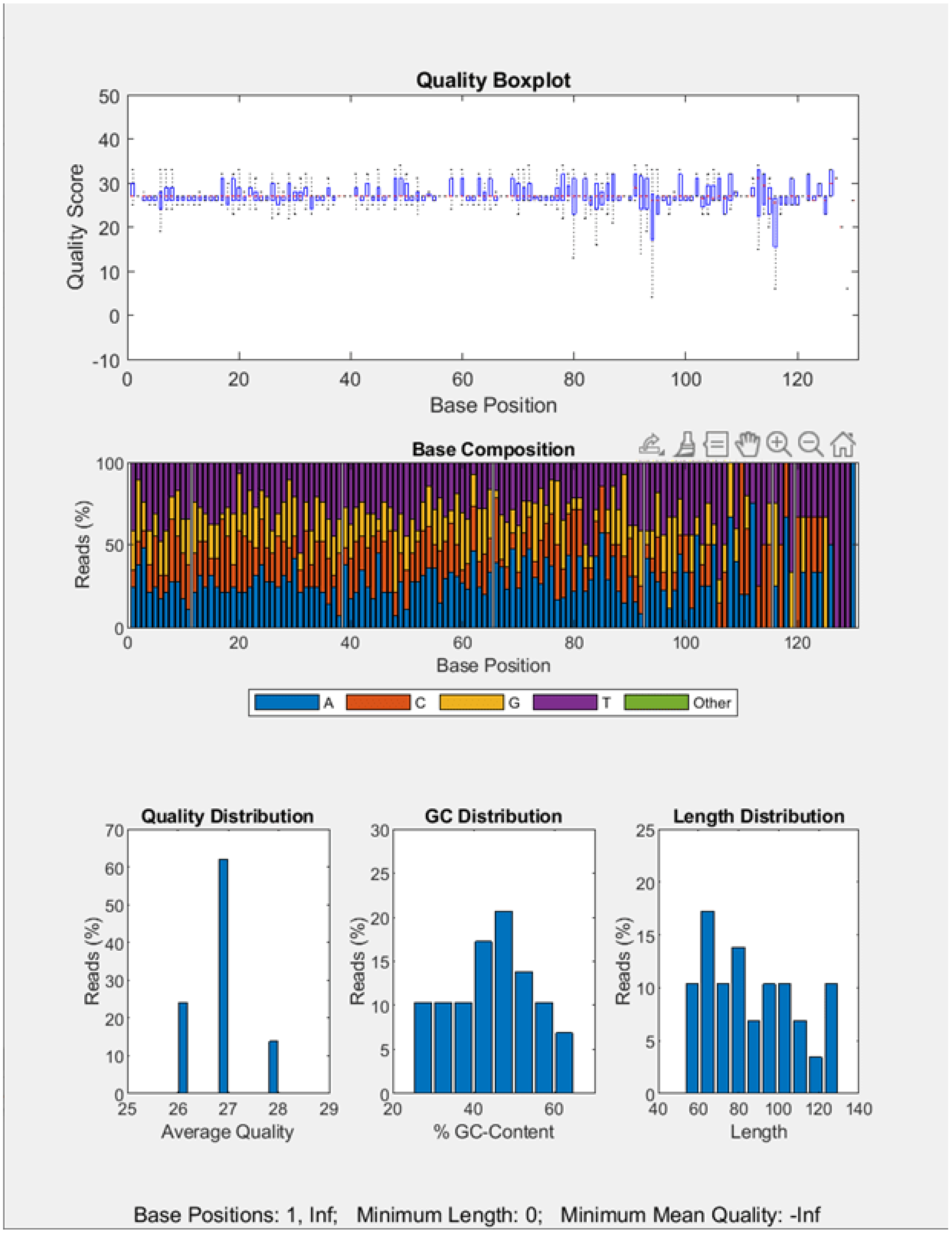
```
ans = struct with fields:
    inputFile: [1×1 bioinfo.pipeline.Input]
```

Connect the `FilteredFASTQFiles` port of the `sequencefilter` block to the `inputFile` port of the `qcplot` block.

```
connect(qcpipeline,sequencefilter,qcplot,["FilteredFASTQFiles","inputFile"]);
```

Run the pipeline to plot the sequence quality data.

```
run(qcpipeline);
```



Base Positions: 1, Inf;   Minimum Length: 0;   Minimum Mean Quality: -Inf

Run Bioinformatics Pipeline Using Input Structure

Import the Pipeline and block objects needed for the example.

```
import bioinfo.pipeline.Pipeline
import bioinfo.pipeline.blocks.*
```

Create a pipeline.

```
P = Pipeline;
```

Create a `Bowtie2Build` block to build index files for the reference genome.

```
bowtie2build = Bowtie2Build;
```

Create a `Bowtie2` block to map the read sequences to the reference sequence.

```
bowtie2 = Bowtie2;
```

Add the blocks to the pipeline.

```
addBlock(P,[bowtie2build,bowtie2],["bowtie2build","bowtie2"]);
```

Get the list of names of all the required input ports from every block in the pipeline that are needed to be set or connected. `IndexBaseName` is an input port of both `bowtie2build` and `bowtie2` block. `Reads1File` is the input port of the `bowtie2` block and `ReferenceFASTAFile` is the input of `bowtie2build` block.

```
portnames = inputNames(P)
```

```
portnames = 1×3 string
    "IndexBaseName"    "Reads1Files"    "ReferenceFASTAFiles"
```

Some blocks have optional input ports. To see the names of these ports, set `IncludeOptional=true`. For instance, the `Bowtie2` block has an optional input port (`Reads2Files`) that accept files for the second mate reads when you have paired-end read data.

```
allportnames = inputNames(P,IncludeOptional=true)
```

```
allportnames = 1×4 string
    "IndexBaseName"    "Reads1Files"    "Reads2Files"    "ReferenceFASTAFiles"
```

Create an input structure to set the input port values of the `bowtie2` and `bowtie2build` blocks. Specifically, set `IndexBaseName` to `"Dmel_chr4"` which is the base name for the reference index files for the Drosophila genome. Set `Reads1Files` to `"SRR6008575_10k_1.fq"` and `Reads2Files` to `"SRR6008575_10k_2.fq"`. Set `ReferenceFASTAFile` to "`Dmel_chr4.fa`". These read files are already provided with the toolbox.

```
inputStruct.IndexBaseName = "Dmel_chr4";
inputStruct.Reads1Files   = "SRR6008575_10k_1.fq";
inputStruct.Reads2Files   = "SRR6008575_10k_2.fq";
inputStruct.ReferenceFASTAFiles = "Dmel_chr4.fa";
```

Optionally, you can compile and check if the input structure is set up correctly. Note that this compilation also happens automatically when you run the pipeline.

```
compile(P,inputStruct);
```

Run the pipeline using the structure as an input.

```
run(P,inputStruct);
```

Get the bowtie2 block result after the pipeline finishes running.

```
wait(P);
mappedFile = results(P,bowtie2)
```

```
mappedFile = struct with fields:
    SAMFile: [1×1 bioinfo.pipeline.datatypes.File]
```

The Bowtie2 block generates a SAM file that contains the mapped results. To see the location of the file, use unwrap.

```
unwrap(mappedFile.SAMFile)
```

Input Arguments

pipeline — Bioinformatics pipeline
bioinfo.pipeline.Pipeline object

Bioinformatics pipeline, specified as a bioinfo.pipeline.Pipeline object.

inputStruct — Input structure to satisfy input ports
structure

Input structure to satisfy on page 1-1648 unconnected input ports, specified as a structure.

The field names of inputStruct must match the names of unconnected ports in the pipeline.

Tip Use inputNames to get the list of names for all unconnected input ports and use them as field names in inputStruct.

Data Types: struct

Name-Value Pair Arguments

Specify optional pairs of arguments as Name1=Value1,...,NameN=ValueN, where Name is the argument name and Value is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: run(pipeline,UseParallel=true) runs the pipeline in parallel.

ResultsDirectory — Location to store pipeline results
PipelineResults folder in the current working directory (default) | character vector | string scalar

Location to store the pipeline results, specified as a character vector or string scalar. The default location is the PipelineResults folder within the current working directory (pwd). In the PipelineResults folder, results from each block of the pipeline are stored separately in a subfolder that is named after the block name.

If you rerun the pipeline with the same results directory, what happens to the existing results depends on `RunMode`:

- When `RunMode=Minimal` (default), the existing results are reused unless the block has become stale.
- When `RunMode=Full`, the existing results are always overwritten.

Data Types: `char` | `string`

`DisplayLevel` — Information to print to command line
`"Warn"` or 2 (default) | `"off"` or 0 | `"Error"` or 1 | ...

Information to print to the MATLAB command line while the pipeline is running, specified as one of the following:

- `"Off"` or 0 — Display no messages.
- `"Error"` or 1 — Display only error messages.
- `"Warn"` or 2 — Display warnings and errors.
- `"Info"` or 3 — Display warnings, errors, and pipeline run progress information.
- `"Debug"` or 4 — Display more detailed debugging information.

Data Types: `double` | `char` | `string`

`UseParallel` — Flag to run pipeline in parallel
`false` or 0 (default) | `true` or 1

Flag to run the pipeline in parallel, specified as a numeric or logical 1 (`true`) or 0 (`false`). Parallel Computing Toolbox is required to run in parallel.

Note Only process-based pools are supported. Thread-based pools are not.

Data Types: `double` | `logical`

RunMode — Run mode of pipeline
`"Minimal"` (default) | `"Full"`

Run mode of the pipeline, specified as one of the following:

- `"Minimal"` — The pipeline runs only the blocks for which one of the following statements is true:
  - The block has not been run before or its results have been deleted.
  - You have modified the block since the last time it ran.
  - Input data, including new runtime inputs, to the block has changed since the last run.
  - The block has one or more upstream blocks which have run since the last time the block was run.

  Tip If you specify a subset of blocks to run using `To`, `From`, and `Only` name-value arguments, these rules are applied only to those selected blocks. It is recommended that you use the default run mode `"Minimal"` because skipping up-to-date blocks can save significant time running the pipeline, especially when the pipeline has long-running blocks that do not need to rerun.

- `"Full"` — The pipeline runs all blocks even if they have previously computed results.

Data Types: `char` | `string`

`From` — Starting blocks
`bioinfo.pipeline.Block` object | vector of objects | character vector | string scalar | string vector | cell array of character vector

Starting blocks when you run the pipeline, specified as a `bioinfo.pipeline.Block` object or vector of block objects. You can also specify a character vector, string scalar, string vector, or cell array of character vectors representing block names. By default, the pipeline runs every block that needs to be run as defined by the `Minimal` run mode.

If you specify this argument, the pipeline starts running from the specified blocks and all the downstream blocks.

If you specify both `To` and `From` blocks, there must exist one block between the blocks specified by `To` and `From`.

You cannot use this argument together with the `Only` name-value argument.

`To` — Ending blocks
`bioinfo.pipeline.Block` object | vector of objects | character vector | string scalar | string vector | cell array of character vector

Ending blocks when you run the pipeline, specified as a `bioinfo.pipeline.Block` object, vector of block objects, character vector, string scalar, string vector, or cell array of character vectors representing block names. By default, the pipeline runs every block that needs to be run as defined by the `Minimal` run mode.

If you specify this argument, the pipeline runs all the upstream blocks and stops at the specified blocks.

If you specify both `To` and `From` blocks, there must exist one block between the blocks specified by `To` and `From`.

You cannot use this argument together with the `Only` name-value argument.

`Only` — Only blocks to run
`bioinfo.pipeline.Block` object | vector of objects | string scalar | ...

Only blocks to run, specified as a `bioinfo.pipeline.Block` object, vector of block objects, character vector, string scalar, string vector, or cell array of character vectors representing block names. By default, the pipeline runs every block that needs to be run as defined by the `Minimal` run mode.

If you specify this argument, the pipeline runs only the specified blocks.

You cannot use this argument together with the `To` or `From` name-value arguments.

`SaveResults` — Blocks with results that are saved to MAT-files
`"-all"` (default) | `bioinfo.pipeline.Block` object | vector of objects | string scalar | ...

Blocks with results that are saved to MAT-files, specified as a `bioinfo.pipeline.Block` object, vector of block objects, character vector, string scalar, string vector, or cell array of character vectors representing block names.

By default (`SaveResults = "-all"`), results from each block are saved in the corresponding MAT-file in the block folder.

More About

Satisfy Input Ports

All required input ports of every block in a pipeline must be satisfied before you can run the pipeline.

To satisfy an input port, you must do one of the following:

- Connect to another port.
- Set the value of the input port, that is, `myBlock.Inputs.PropertyName.Value`. For example, consider a `BamSort` block. To specify the name of a BAM file as the block input value, set the value as `bamsortBlock.Inputs.BAMFile.Value = "ex1.bam"`.
- Pass in an input structure by calling `run(`*`pipeline`*`,`*`inputStruct`*`)`, where *inputStruct* has the field name equivalent to the input port name and the field value as the input port value.

Version History

Introduced in R2023a

See Also

`bioinfo.pipeline.Pipeline` | **Biopipeline Designer**

saminfo

Return information about SAM file

Syntax

```
InfoStruct = saminfo(File)
InfoStruct = saminfo(File,Name,Value)
```

Description

`InfoStruct` = `saminfo(`*`File`*`)` returns a MATLAB structure containing summary information about a SAM-formatted file.

`InfoStruct = saminfo(File,Name,Value)` returns a MATLAB structure with additional options specified by one or more `Name,Value` pair arguments.

Input Arguments

`File`

Character vector or string specifying a file name or path and file name of a SAM-formatted file. If you specify only a file name, that file must be on the MATLAB search path or in the Current Folder.

Default:

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

`NumOfReads`

Logical that controls the inclusion of a `NumReads` field in *`InfoStruct`*, the output structure.

Note Setting `NumOfReads` to `true` can significantly increase the time to create the output structure.

Default: `false`

`ScanDictionary`

Logical that controls the scanning of the SAM-formatted file to determine the reference names and the number of reads aligned to each reference. If `true`, the `ScannedDictionary` and `ScannedDictionaryCount` fields contain this information.

Default: `false`

Output Arguments

`InfoStruct`

MATLAB structure containing summary information about a SAM-formatted file. The structure contains these fields.

Field	Description
`Filename`	Name of the SAM-formatted file.
`FilePath`	Path to the file.
`FileSize`	Size of the file in bytes.
`FileModDate`	Modification date of the file.
`NumReads*`	Number of sequence reads in the file.
`ScannedDictionary*`	Cell array of character vectors specifying the names of the reference sequences in the SAM-formatted file.
`ScannedDictionaryCount*`	Cell array specifying the number of reads aligned to each reference sequence.
`Header**`	Structure containing the file format version, sort order, and group order.
`SequenceDictionary**`	Structure containing the: • Sequence name • Sequence length • Genome assembly identifier • MD5 checksum of sequence • URI of sequence • Species
`ReadGroup**`	Structure containing the: • Read group identifier • Sample • Library • Description • Platform unit • Predicted median insert size • Sequencing center • Date • Platform
`Program**`	Structure containing the: • Program name • Version • Command line

* — The NumReads field is empty if you do not set the NumOfReads name-value pair argument to true. The ScannedDictionary and ScannedDictionaryCount fields are empty if you do not set the ScanDictionary name-value pair argument to true.

** — These structures and their fields appear in the output structure only if they are in the SAM file. The information in these structures depends on the information in the SAM file.

Examples

Return information about the ex1.sam file included with Bioinformatics Toolbox:

```
info = saminfo('ex1.sam')

info = 

                  Filename: 'ex1.sam'
                  FilePath: [1x89 char]
                  FileSize: 254270
               FileModDate: '12-May-2011 14:23:25'
                    Header: [1x1 struct]
        SequenceDictionary: [1x1 struct]
                 ReadGroup: [1x2 struct]
                  NumReads: []
         ScannedDictionary: {0x1 cell}
    ScannedDictionaryCount: [0x1 uint64]
```

Return information about the ex1.sam file including the number of sequence reads:

```
info = saminfo('ex1.sam','numofreads', true)

info = 

                  Filename: 'ex1.sam'
                  FilePath: [1x89 char]
                  FileSize: 254270
               FileModDate: '12-May-2011 14:23:25'
                    Header: [1x1 struct]
        SequenceDictionary: [1x1 struct]
                 ReadGroup: [1x2 struct]
                  NumReads: 1501
         ScannedDictionary: {0x1 cell}
    ScannedDictionaryCount: [0x1 uint64]
```

Tips

Use saminfo to investigate the size and content of a SAM file before using the samread function to read the file contents into a MATLAB structure.

Version History

Introduced in R2010a

See Also

samread | fastqread | fastqwrite | fastqinfo | fastainfo | fastaread | fastawrite | sffinfo | sffread | BioIndexedFile | BioMap

Topics
"Manage Sequence Read Data in Objects"

External Websites
Sequence Read Archive
SAM format specification

samplealign

Align two data sets containing sequential observations by introducing gaps

Syntax

```
[I, J] = samplealign(X, Y)

[I, J] = samplealign(X, Y, ...'Band', BandValue, ...)
[I, J] = samplealign(X, Y, ...'Width', WidthValue, ...)
[I, J] = samplealign(X, Y, ...'Gap', GapValue, ...)
[I, J] = samplealign(X, Y, ...'Quantile', QuantileValue, ...)
[I, J] = samplealign(X, Y, ...'Distance', DistanceValue, ...)
[I, J] = samplealign(X, Y, ...'Weights', WeightsValue, ...)
[I, J] = samplealign(X, Y, ...'ShowConstraints', ShowConstraintsValue, ...)
[I, J] = samplealign(X, Y, ...'ShowNetwork', ShowNetworkValue, ...)
[I, J] = samplealign(X, Y, ...'ShowAlignment', ShowAlignmentValue, ...)
```

Input Arguments

X, *Y*	Matrices of data where rows correspond to observations or samples, and columns correspond to features or dimensions. *X* and *Y* can have a different number of rows, but they must have the same number of columns. The first column is the reference dimension and must contain unique values in ascending order. The reference dimension could contain sample indices of the observations or a measurable value, such as time.
BandValue	Either of the following: • Scalar. • Function specified using `@(z)`, where *z* is the mid-point between a given observation in one data set and a given observation in the other data set. *BandValue* specifies a maximum allowable distance between observations (along the reference dimension only) in the two data sets, thus limiting the number of potential matches between observations in two data sets. If *S* is the value in the reference dimension for a given observation (row) in one data set, then that observation is matched only with observations in the other data set whose values in the reference dimension fall within *S* ± *BandValue*. Then, only these potential matches are passed to the algorithm for further scoring. Default *BandValue* is `Inf`.

WidthValue	Either of the following: • Two-element vector, [*U*, *V*] • Scalar that is used for both *U* and *V* *WidthValue* limits the number of potential matches between observations in two data sets; that is, each observation in *X* is scored to the closest *U* observations in *Y*, and each observation in *Y* is scored to the closest *V* observations in *X*. Then, only these potential matches are passed to the algorithm for further scoring. Closeness is measured using only the first column (reference dimension) in each data set. Default is `Inf` if `'Band'` is specified; otherwise default is `10`.
GapValue	Any of the following: • Cell array, {*G*, *H*}, where *G* is either a scalar or a function handle specified using `@(X)`, and *H* is either a scalar or a function handle specified using `@(Y)`. The functions `@(X)` and `@(Y)` must calculate the penalty for each observation (row) when it is matched to a gap in the other data set. The functions `@(X)` and `@(Y)` must return a column vector with the same number of rows as *X* or *Y*, containing the gap penalty for each observation (row). • Single function handle specified using `@(Z)`, which is used for both *G* and *H*. The function `@(Z)` must calculate the penalty for each observation (row) in both *X* and *Y* when it is matched to a gap in the other data set. The function `@(Z)` must take as arguments *X* and *Y*. The function `@(Z)` must return a column vector with the same number of rows as *X* or *Y*, containing the gap penalty for each observation (row). • Scalar that is used for both *G* and *H*. *GapValue* specifies the position-dependent terms for assigning gap penalties. The calculated value, *GPX*, is the gap penalty for matching observations from the first data set *X* to gaps inserted in the second data set *Y*, and is the product of two terms: *GPX* = *G* * *QMS*. The term *G* takes its value as a function of the observations in *X*. Similarly, *GPY* is the gap penalty for matching observations from *Y* to gaps inserted in *X*, and is the product of two terms: *GPY* = *H* * *QMS*. The term *H* takes its value as a function of the observations in *Y*. By default, the term *QMS* is the 0.75 quantile of the score for the pairs of observations that are potential matches (that is, pairs that comply with the `'Band'` and `'Width'` constraints). Default *GapValue* is `1`.
QuantileValue	Scalar between `0` and `1` that specifies the quantile value used to calculate the term *QMS*, which is used by the `'Gap'` property to calculate gap penalties. Default is `0.75`.

`DistanceValue`	Function handle specified using `@(`*`R`*`,`*`S`*`)`. The function `@(`*`R`*`,`*`S`*`)` must: • Calculate the distance between pairs of observations that are potential matches. • Take as arguments, *R* and *S*, matrices that have the same number of rows and columns, and whose paired rows represent all potential matches of observations in *X* and *Y* respectively. • Return a column vector of positive values with the same number of elements as rows in *R* and *S*. Default is the Euclidean distance between the pairs. **Caution** All columns in *X* and *Y*, including the reference dimension, are considered when calculating distances. If you do not want to include the reference dimension in the distance calculations, use the `'Weight'` property to exclude it.
`WeightsValue`	Either of the following: • Logical row vector with the same number of elements as columns in *X* and *Y*, that specifies columns in *X* and *Y*. • Numeric row vector with the same number of elements as columns in *X* and *Y*, that specifies the relative weights of the columns (features). This property controls the inclusion/exclusion of columns (features) or the emphasis of columns (features) when calculating the distance score between observations that are potential matches, that is, when using the `'Distance'` property. Default is a logical row vector with all elements set to `true`. **Tip** Using a numeric row vector for *`WeightsValue`* and setting some values to `0` can simplify the distance calculation when the data sets have many columns (features). **Note** The weight values are not considered when using the `'Band'`, `'Width'`, or `'Gap'` property.
`ShowConstraintsValue`	Controls the display of the search space constrained by the specified `'Band'` and `'Width'` input parameters, thereby giving an indication of the memory required to run the algorithm with the specific `'Band'` and `'Width'` parameters on your data sets. Choices are `true` or `false` (default).
`ShowNetworkValue`	Controls the display of the dynamic programming network, the match scores, the gap penalties, and the winning path. Choices are `true` or `false` (default).

ShowAlignmentValue	Controls the display of the first and second columns of the *X* and *Y* data sets in the abscissa and the ordinate respectively, of a two-dimensional plot. Choices are `true`, `false` (default), or an integer specifying a column of the *X* and *Y* data sets to plot as the ordinate.

Output Arguments

I	Column vector containing indices of rows (observations) in *X* that match to a row (observation) in *Y*. Missing indices indicate that row (observation) is matched to a gap.
J	Column vector containing indices of rows (observations) in *Y* that match to a row (observation) in *X*. Missing indices indicate that row (observation) is matched to a gap.

Description

`[I, J] = samplealign(X, Y)` aligns the observations in two matrices of data, *X* and *Y*, by introducing gaps. *X* and *Y* are matrices of data where rows correspond to observations or samples, and columns correspond to features or dimensions. *X* and *Y* can have different number of rows, but must have the same number of columns. The first column is the reference dimension and must contain unique values in ascending order. The reference dimension could contain sample indices of the observations or a measurable value, such as time. The `samplealign` function uses a dynamic programming algorithm to minimize the sum of positive scores resulting from pairs of observations that are potential matches and the penalties resulting from the insertion of gaps. Return values *I* and *J* are column vectors containing indices that indicate the matches for each row (observation) in *X* and *Y* respectively.

Tip If you do not specify return values, `samplealign` does not run the dynamic programming algorithm. Running `samplealign` without return values, but setting the `'ShowConstraints'`, `'ShowNetwork'`, or `'ShowAlignment'` property to `true`, lets you explore the constrained search space, the dynamic programming network, or the aligned observations, without running into potential memory problems.

`[I, J] = samplealign(X, Y, ...'PropertyName', PropertyValue, ...)` calls `samplealign` with optional properties that use property name/property value pairs. You can specify one or more properties in any order. Each *PropertyName* must be enclosed in single quotation marks and is case insensitive. These property name/property value pairs are as follows:

`[I, J] = samplealign(X, Y, ...'Band', BandValue, ...)` specifies a maximum allowable distance between observations (along the reference dimension only) in the two data sets, thus limiting the number of potential matches between observations in the two data sets. If *S* is the value in the reference dimension for a given observation (row) in one data set, then that observation is matched only with observations in the other data set whose values in the reference dimension fall within *S* ± *BandValue*. Then, only these potential matches are passed to the algorithm for further scoring. *BandValue* can be a scalar or a function specified using `@(z)`, where *z* is the mid-point between a given observation in one data set and a given observation in the other data set. Default *BandValue* is `Inf`.

This constraint reduces the time and memory complexity of the algorithm from O(*MN*) to O(sqrt(*MN*)**K*), where *M* and *N* are the number of observations in *X* and *Y* respectively, and *K* is a small constant such that *K*<<*M* and *K*<<*N*. Adjust *BandValue* to the maximum expected shift between the reference dimensions in the two data sets, that is, between *X*(:,1) and *Y*(:,1).

`[I, J] = samplealign(X, Y, ...'Width', WidthValue, ...)` limits the number of potential matches between observations in two data sets; that is, each observation in *X* is scored to the closest *U* observations in *Y*, and each observation in *Y* is scored to the closest *V* observations in *X*. Then, only these potential matches are passed to the algorithm for further scoring. *WidthValue* is either a two-element vector, [*U*, *V*] or a scalar that is used for both *U* and *V*. Closeness is measured using only the first column (reference dimension) in each data set. Default is `Inf` if `'Band'` is specified; otherwise default is `10`.

This constraint reduces the time and memory complexity of the algorithm from O(*MN*) to O(sqrt(*MN*)*sqrt(*UV*)), where *M* and *N* are the number of observations in *X* and *Y* respectively, and *U* and *V* are small such that *U*<<*M* and *V*<<*N*.

Note If you specify both `'Band'` and `'Width'`, only pairs of observations that meet both constraints are considered potential matches and passed to the algorithm for scoring.

Tip Specify `'Width'` when you do not have a good estimate for the `'Band'` property. To get an indication of the memory required to run the algorithm with specific `'Band'` and `'Width'` parameters on your data sets, run `samplealign`, but do not specify return values and set `'ShowConstraints'` to `true`.

`[I, J] = samplealign(X, Y, ...'Gap', GapValue, ...)` specifies the position-dependent terms for assigning gap penalties.

GapValue is any of the following:

- Cell array, {*G*, *H*}, where *G* is either a scalar or a function handle specified using `@(X)`, and *H* is either a scalar or a function handle specified using `@(Y)`. The functions `@(X)` and `@(Y)` must calculate the penalty for each observation (row) when it is matched to a gap in the other data set. The functions `@(X)` and `@(Y)` must return a column vector with the same number of rows as *X* or *Y*, containing the gap penalty for each observation (row).
- Single function handle specified using `@(Z)`, that is used for both *G* and *H*. The function `@(Z)` must calculate the penalty for each observation (row) in both *X* and *Y* when it is matched to a gap in the other data set. The function `@(Z)` must take as arguments *X* and *Y*. The function `@(Z)` must return a column vector with the same number of rows as *X* or *Y*, containing the gap penalty for each observation (row).
- Scalar that is used for both *G* and *H*.

The calculated value, *GPX*, is the gap penalty for matching observations from the first data set *X* to gaps inserted in the second data set *Y*, and is the product of two terms: *GPX* = *G* * *QMS*. The term *G* takes its value as a function of the observations in *X*. Similarly, *GPY* is the gap penalty for matching observations from *Y* to gaps inserted in *X*, and is the product of two terms: *GPY* = *H* * *QMS*. The term *H* takes its value as a function of the observations in *Y*. By default, the term *QMS* is the 0.75 quantile of the score for the pairs of observations that are potential matches (that is, pairs that comply with the `'Band'` and `'Width'` constraints).

If *G* and *H* are positive scalars, then *GPX* and *GPY* are independent of the observation where the gap is being inserted.

Default *GapValue* is `1`, that is, both *G* and *H* are `1`, which indicates that the default penalty for gap insertions in both sequences is equivalent to the quantile (set by the `'Quantile'` property, default = `0.75`) of the score for the pairs of observations that are potential matches.

Note *GapValue* defaults to a relatively safe value. However, the success of the algorithm depends on the fine tuning of the gap penalties, which is application dependent. When the gap penalties are large relative to the score of the correct matches, `samplealign` returns alignments with fewer gaps, but with more incorrectly aligned regions. When the gap penalties are smaller, the output alignment contains longer regions with gaps and fewer matched observations. Set `'ShowNetwork'` to `true` to compare the gap penalties to the score of matched observations in different regions of the alignment.

`[I, J] = samplealign(X, Y, ...'Quantile', QuantileValue, ...)` specifies the quantile value used to calculate the term *QMS*, which is used by the `'Gap'` property to calculate gap penalties. *QuantileValue* is a scalar between `0` and `1`. Default is `0.75`.

Tip Set *QuantileValue* to an empty array (`[]`) to make the gap penalties independent of *QMS*, that is, *GPX* and *GPY* are functions of only the *G* and *H* input parameters respectively.

`[I, J] = samplealign(X, Y, ...'Distance', DistanceValue, ...)` specifies a function to calculate the distance between pairs of observations that are potential matches. *DistanceValue* is a function handle specified using `@(R,S)`. The function `@(R,S)` must take as arguments, *R* and *S*, matrices that have the same number of rows and columns, and whose paired rows represent all potential matches of observations in *X* and *Y* respectively. The function `@(R,S)` must return a column vector of positive values with the same number of elements as rows in *R* and *S*. Default is the Euclidean distance between the pairs.

Caution All columns in *X* and *Y*, including the reference dimension, are considered when calculating distances. If you do not want to include the reference dimension in the distance calculations, use the `'Weight'` property to exclude it.

`[I, J] = samplealign(X, Y, ...'Weights', WeightsValue, ...)` controls the inclusion/exclusion of columns (features) or the emphasis of columns (features) when calculating the distance score between observations that are potential matches, that is when using the `'Distance'` property. *WeightsValue* can be a logical row vector that specifies columns in *X* and *Y*. *WeightsValue* can also be a numeric row vector with the same number of elements as columns in *X* and *Y*, that specifies the relative weights of the columns (features). Default is a logical row vector with all elements set to `true`.

Tip Using a numeric row vector for *WeightsValue* and setting some values to `0` can simplify the distance calculation when the data sets have many columns (features).

Note The weight values are not considered when computing the constrained alignment space, that is when using the `'Band'` or `'Width'` properties, or when calculating the gap penalties, that is when using the `'Gap'` property.

`[I, J] = samplealign(X, Y, ...'ShowConstraints', ShowConstraintsValue, ...)` controls the display of the search space constrained by the input parameters `'Band'` and `'Width'`, giving an indication of the memory required to run the algorithm with specific `'Band'` and `'Width'` on your data sets. Choices are `true` or `false` (default).

`[I, J] = samplealign(X, Y, ...'ShowNetwork', ShowNetworkValue, ...)` controls the display of the dynamic programming network, the match scores, the gap penalties, and the winning path. Choices are `true` or `false` (default).

`[I, J] = samplealign(X, Y, ...'ShowAlignment', ShowAlignmentValue, ...)` controls the display of the first and second columns of the *X* and *Y* data sets in the abscissa and the ordinate respectively, of a two-dimensional plot. Links between all the potential matches that meet the constraints are displayed, and the matches belonging to the output alignment are highlighted. Choices are `true`, `false` (default), or an integer specifying a column of the *X* and *Y* data sets to plot as the ordinate.

Examples

Example 1.58. Warping a sine wave with a smooth function to more closely follow cyclical sunspot activity

1. Load `sunspot.dat`, a data file included with the MATLAB software, that contains the variable `sunspot`, which is a two-column matrix containing variations in sunspot activity over the last 300 years. The first column is the reference dimension (years), and the second column contains sunspot activity values. Sunspot activity is cyclical, reaching a maximum about every 11 years.

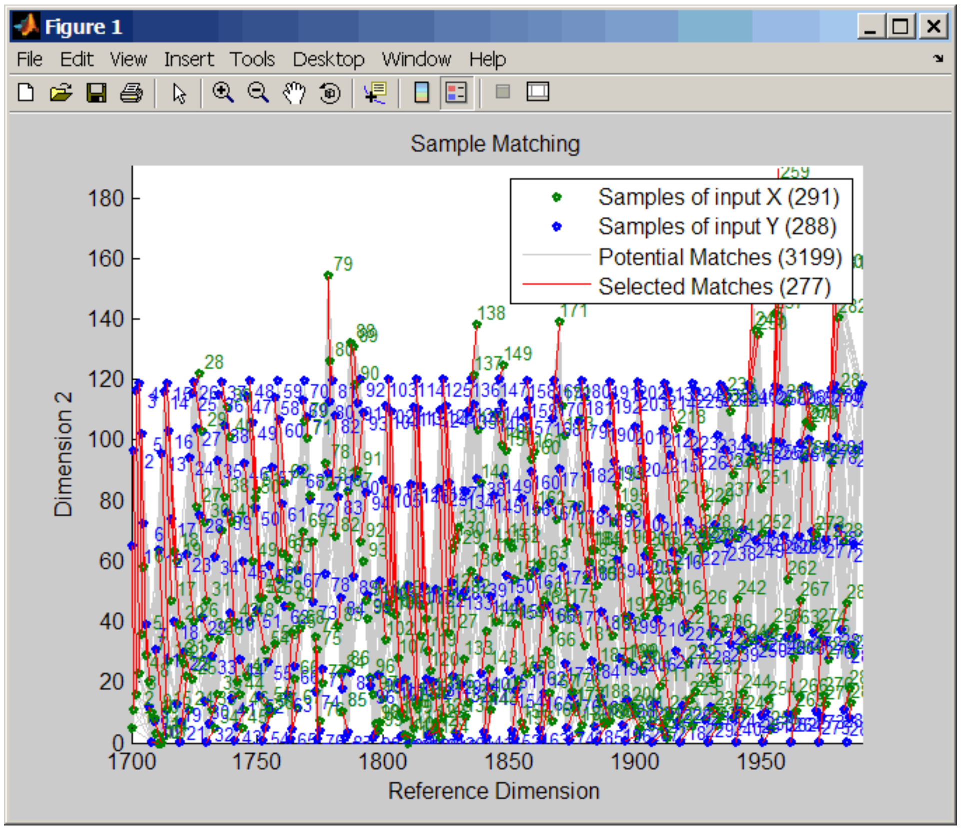
```
load sunspot.dat
```

2. Create a sine wave with a known period of sunspot activity.

```
years = (1700:1990)';
T = 11.038;
f = @(y) 60 + 60 * sin(y*(2*pi/T));
```

3. Align the observations between the sine wave and the sunspot activity by introducing gaps.

```
[i,j] = samplealign([years f(years)],sunspot,'weights',...
                    [0 1],'showalignment',true);
```

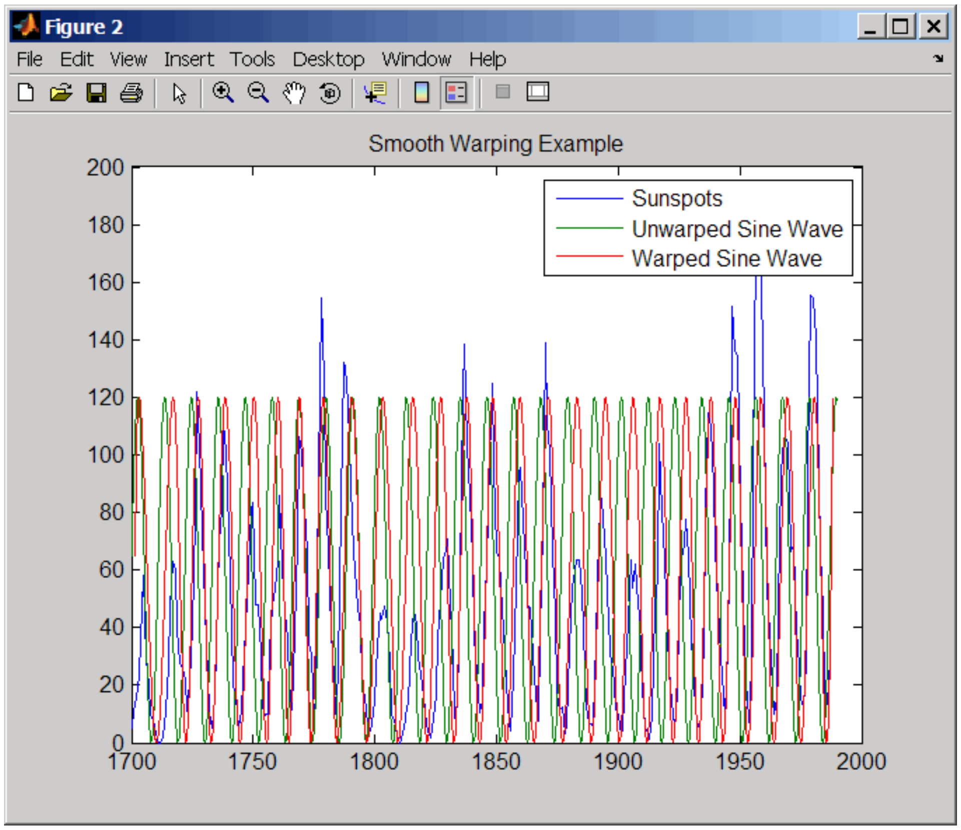


4 Estimate a smooth function to warp the sine wave.

```
[p,s,mu] = polyfit(years(i),years(j),15);
wy = @(y) polyval(p,(y-mu(1))./mu(2));
```

5 Plot the sunspot cycles, unwarped sine wave, and warped sine wave.

```
years = (1700:1/12:1990)';
figure
plot(sunspot(:,1),sunspot(:,2),years,f(years),wy(years),...
     f(years))
legend('Sunspots','Unwarped Sine Wave','Warped Sine Wave')
title('Smooth Warping Example')
```



Example 1.59. Recovering a nonlinear warping between two signals containing noisy Gaussian peaks

1 Create two signals with noisy Gaussian peaks.

```
rng('default')
peakLoc = [30 60 90 130 150 200 230 300 380 430];
peakInt = [7 1 3 10 3 6 1 8 3 10];
time = 1:450;
comp = exp(-(bsxfun(@minus,time,peakLoc')./5).^2);
sig_1 = (peakInt + rand(1,10)) * comp + rand(1,450);
sig_2 = (peakInt + rand(1,10)) * comp + rand(1,450);
```

2 Define a nonlinear warping function.

```
wf = @(t) 1 + (t<=100).*0.01.*(t.^2) + (t>100).*...
     (310+150*tanh(t./100-3));
```

3 Warp the second signal to distort it.

```
sig_2 = interp1(time,sig_2,wf(time),'pchip');
```

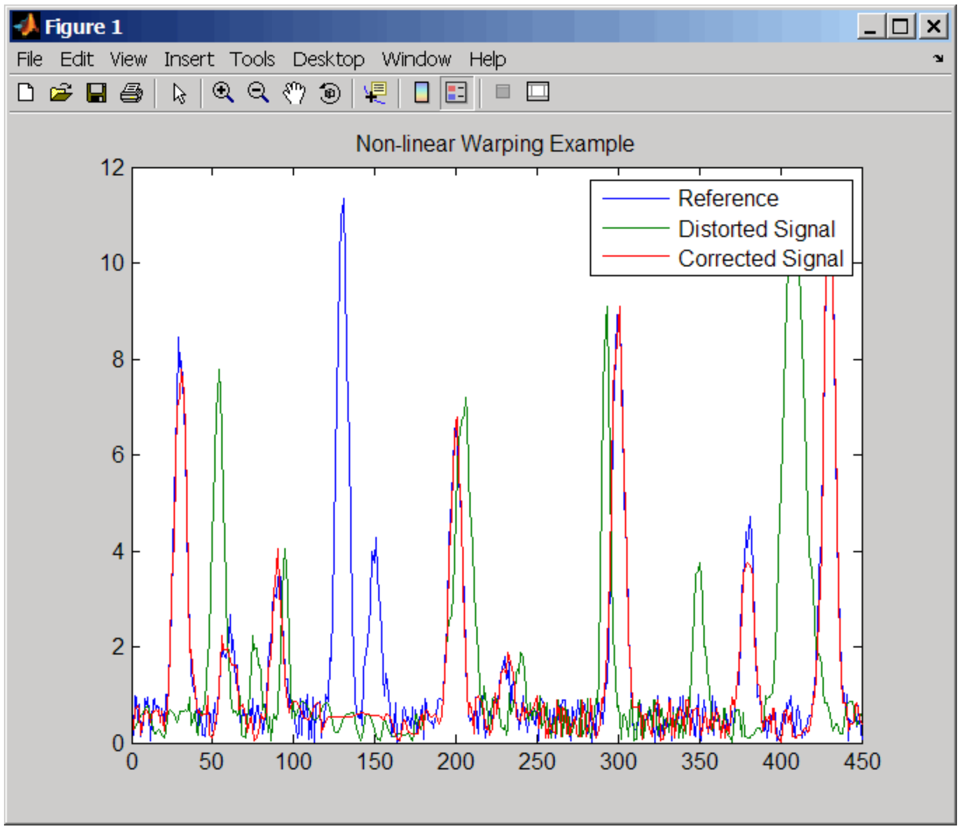
4 Align the observations between the two signals by introducing gaps.

```
[i,j] = samplealign([time;sig_1]',[time;sig_2]',...
                    'weights',[0,1],'band',35,'quantile',.5);
```

5 Plot the reference signal, distorted signal, and warped (corrected) signal.

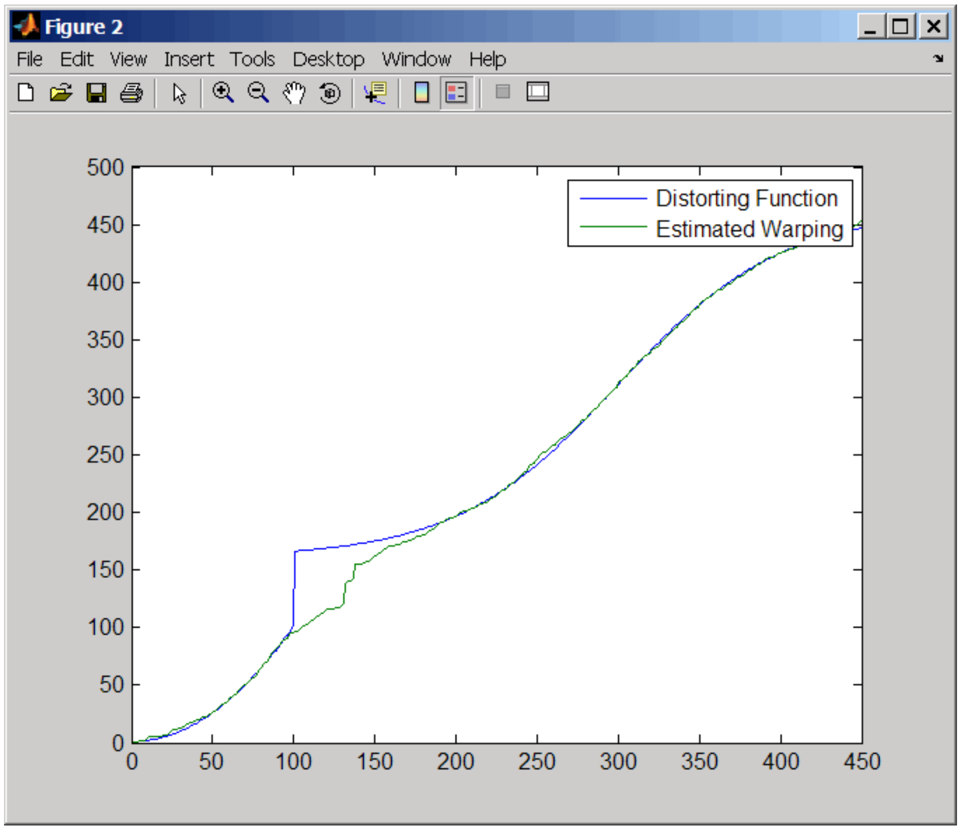
```
figure
sig_3 = interp1(time,sig_2,interp1(i,j,time,'pchip'),'pchip');
```

```
plot(time,sig_1,time,sig_2,time,sig_3)
legend('Reference','Distorted Signal','Corrected Signal')
title('Non-linear Warping Example')
```



6 Plot the real and the estimated warping functions.

```
figure
plot(time,wf(time),time,interp1(j,i,time,'pchip'))
legend('Distorting Function','Estimated Warping')
```



Note For examples of using function handles for the `Band`, `Gap`, and `Distance` properties, see "Visualizing and Preprocessing Hyphenated Mass Spectrometry Data Sets for Metabolite and Protein/Peptide Profiling".

Version History

Introduced in R2007b

References

[1] Myers, C.S. and Rabiner, L.R. (1981). A comparative study of several dynamic time-warping algorithms for connected word recognition. The Bell System Technical Journal *60:7*, 1389–1409.

[2] Sakoe, H. and Chiba, S. (1978). Dynamic programming algorithm optimization for spoken word recognition. IEEE Trans. Acoustics, Speech and Signal Processing *ASSP-26(1)*, 43–49.

See Also

`msalign` | `msheatmap` | `mspalign` | `msppresample` | `msresample`

sampleData

Class: `bioma.ExpressionSet`
Package: `bioma`

Retrieve or set sample metadata in ExpressionSet object

Syntax

```
MetaDataObj = sampleData(ESObj)
NewESObj = sampleData(ESObj, NewMetaDataObj)
```

Description

`MetaDataObj = sampleData(ESObj)` returns a MetaData object containing the sample metadata from an ExpressionSet object.

`NewESObj = sampleData(ESObj, NewMetaDataObj)` replaces the sample metadata in *`ESObj`*, an ExpressionSet object, with *`NewMetaDataObj`*, and returns *`NewESObj`*, a new ExpressionSet object.

Input Arguments

`ESObj`

Object of the `bioma.ExpressionSet` class.

Default:

`NewMetaDataObj`

Object of the `bioma.data.MetaData` class, containing sample metadata, stored in two dataset arrays. The sample names and variable names in *`NewMetaDataObj`* must match the sample names and variable names in the *`MetaDataObj`* being replaced in the ExpressionSet object, *`ESObj`*.

Default:

Output Arguments

`MetaDataObj`

Object of the `bioma.data.MetaData` class, containing the sample metadata, stored in two dataset arrays.

`NewESObj`

Object of the `bioma.ExpressionSet` class, returned after replacing the MetaData object containing the sample metadata.

Examples

Construct an ExpressionSet object, `ESObj`, as described in the “Examples” on page 1-0 section of the `bioma.ExpressionSet` class reference page. Retrieve the MetaData object that contains sample metadata, stored in the ExpressionSet object:

```
% Retrieve the sample data
NewMDObj = sampleData(ESObj);
```

See Also

`bioma.ExpressionSet` | `bioma.data.ExptData` | `sampleNames` | `featureData`

Topics

“Managing Gene Expression Data in Objects”

sampleNames

Class: `bioma.ExpressionSet`
Package: `bioma`

Retrieve or set sample names in ExpressionSet object

Syntax

`SamNames` `= sampleNames(`*`ESObj`*`)`
`SamNames` `= sampleNames(`*`ESObj`*`, `*`Subset`*`)`
`NewESObj` `= sampleNames(`*`ESObj`*`, `*`Subset`*`, `*`NewSamNames`*`)`

Description

`SamNames` `= sampleNames(`*`ESObj`*`)` returns a cell array of strings specifying all sample names in an ExpressionSet object.

`SamNames` `= sampleNames(`*`ESObj`*`, `*`Subset`*`)` returns a cell array of strings specifying a subset the sample names in an ExpressionSet object.

`NewESObj` `= sampleNames(`*`ESObj`*`, `*`Subset`*`, `*`NewSamNames`*`)` replaces the sample names specified by *`Subset`* in *`ESObj`*, an ExpressionSet object, with *`NewSamNames`*, and returns *`NewESObj`*, a new ExpressionSet object.

Input Arguments

`ESObj`

Object of the `bioma.ExpressionSet` class.

`Subset`

One of the following to specify a subset of the sample names in an ExpressionSet object:

- String or character vector specifying a sample name
- Cell array of character vectors or string vector specifying sample names
- Positive integer
- Vector of positive integers
- Logical vector

`NewSamNames`

New sample names for specific sample names within an ExpressionSet object, specified by one of the following:

- Numeric vector
- Cell array of character vectors or string vector
- Character vector or string, which `sampleNames` uses as a prefix for the sample names, with sample numbers appended to the prefix

- Logical `true` or `false` (default). If `true`, `sampleNames` assigns unique sample names using the format `Sample1`, `Sample2`, etc.

The number of sample names in *`NewSamNames`* must equal the number of samples specified by *`Subset`*.

Output Arguments

`SamNames`

Cell array of strings specifying all or some of the sample names in an ExpressionSet object. The sample names are the column names in the DataMatrix objects in the ExpressionSet object. The sample names are also the row names of the *`VarValues`* dataset array in the MetaData object in the ExpressionSet object.

`NewESObj`

Object of the `bioma.ExpressionSet` class, returned after replacing specific sample names.

Examples

Construct an ExpressionSet object, `ESObj`, as described in the “Examples” on page 1-0 section of the `bioma.ExpressionSet` class reference page. Retrieve the sample names from it:

```
% Retrieve the sample names
SNames = sampleNames(ESObj);
```

See Also

`bioma.ExpressionSet` | `bioma.data.ExptData` | `DataMatrix` | `bioma.data.MetaData` | `featureNames`

Topics

“Managing Gene Expression Data in Objects”

sampleNames

Class: `bioma.data.ExptData`
Package: `bioma.data`

Retrieve or set sample names in ExptData object

Syntax

```
SamNames = sampleNames(EDObj)
SamNames = sampleNames(EDObj, Subset)
NewEDObj = sampleNames(EDObj, Subset, NewSamNames)
```

Description

`SamNames = sampleNames(EDObj)` returns a cell array of character vectors specifying all sample names in an ExptData object.

`SamNames = sampleNames(EDObj, Subset)` returns a cell array of character vectors specifying a subset the sample names in an ExptData object.

`NewEDObj = sampleNames(EDObj, Subset, NewSamNames)` replaces the sample names specified by *`Subset`* in *`EDObj`*, an ExptData object, with *`NewSamNames`*, and returns *`NewEDObj`*, a new ExptData object.

Input Arguments

`EDObj`

Object of the `bioma.data.ExptData` class.

Default:

`Subset`

One of the following to specify a subset of the sample names in an ExptData object:

- Character vector specifying a sample name
- Cell array of character vectors specifying sample names
- Positive integer
- Vector of positive integers
- Logical vector

Default:

`NewSamNames`

New sample names for specific sample names within an ExptData object, specified by one of the following:

- Numeric vector
- Character vector or cell array of character vectors
- Character vector, which `sampleNames` uses as a prefix for the sample names, with sample numbers appended to the prefix
- Logical `true` or `false` (default). If `true`, `sampleNames` assigns unique sample names using the format `Sample1`, `Sample2`, etc.

The number of sample names in *`NewSamNames`* must equal the number of samples specified by *`Subset`*.

Default:

Output Arguments

`SamNames`

Cell array of character vectors specifying all or some of the sample names in an ExptData object. The sample names are the column names in the DataMatrix objects in the ExptData object.

`NewEDObj`

Object of the `bioma.data.ExptData` class, returned after replacing specific sample names.

Examples

Construct an ExptData object, and then retrieve the sample names from it:

```
% Import bioma.data package to make constructor functions
% available
import bioma.data.*
% Create DataMatrix object from .txt file containing
% expression values from microarray experiment
dmObj = DataMatrix('File', 'mouseExprsData.txt');
% Construct ExptData object
EDObj = ExptData(dmObj);
% Retrieve sample names
SNames = sampleNames(EDObj);
```

See Also

`bioma.data.ExptData` | `DataMatrix` | `dmNames` | `elementNames` | `featureNames`

Topics

"Representing Expression Data Values in ExptData Objects"

sampleNames

Class: `bioma.data.MetaData`
Package: `bioma.data`

Retrieve or set sample names in MetaData object

Syntax

```
SamFeatNames = sampleNames(MDObj)
SamFeatNames = sampleNames(MDObj, Subset)
NewMDObj = sampleNames(MDObj, Subset, NewSamFeatNames)
```

Description

`SamFeatNames = sampleNames(MDObj)` returns a cell array of character vectors specifying all sample names in a MetaData object.

`SamFeatNames = sampleNames(MDObj, Subset)` returns a cell array of character vectors specifying a subset the sample names in a MetaData object.

`NewMDObj = sampleNames(MDObj, Subset, NewSamFeatNames)` replaces the sample names specified by *`Subset`* in *`MDObj`*, a MetaData object, with *`NewSamFeatNames`*, and returns *`NewMDObj`*, a new MetaData object.

Input Arguments

`MDObj`

Object of the `bioma.data.MetaData` class.

`Subset`

One of the following to specify a subset of the sample names in a MetaData object:

- Character vector specifying a sample name
- Cell array of character vectors specifying sample names
- Positive integer
- Vector of positive integers
- Logical vector

`NewSamFeatNames`

New sample names for specific names within a MetaData object, specified by one of the following:

- Numeric vector
- Cell array of character vectors or character array
- Character vector which `sampleNames` uses as a prefix for the sample or feature names, with numbers appended to the prefix

- Logical `true` or `false` (default). If `true`, `sampleNames` assigns unique names using the format `Sample1`, `Sample2`, etc.

The number of names in *`NewSamFeatNames`* must equal the number of samples specified by *`Subset`*.

Output Arguments

`SamFeatNames`

Cell array of character vectors specifying all or some of the sample names in a MetaData object. The sample names are also the row names of the *`VarValues`* dataset array in the MetaData object.

`NewMDObj`

Object of the `bioma.data.MetaData` class, returned after replacing specific sample names.

Examples

Construct a MetaData object, and then retrieve the sample names from it:

```
% Import bioma.data package to make constructor function
% available
import bioma.data.*
% Construct MetaData object from .txt file
MDObj2 = MetaData('File', 'mouseSampleData.txt', 'VarDescChar', '#');
% Retrieve the sample names
SNames = sampleNames(MDObj2)
```

See Also
`bioma.data.MetaData` | `variableDesc` | `variableValues` | `variableNames`

Topics
“Representing Sample and Feature Metadata in MetaData Objects”

sampleVarDesc

Class: `bioma.ExpressionSet`
Package: `bioma`

Retrieve or set sample variable descriptions in ExpressionSet object

Syntax

```
DSVarDescriptions = sampleVarDesc(ESObj)
NewESObj = sampleVarDesc(ESObj, NewDSVarDescriptions)
```

Description

`DSVarDescriptions` `= sampleVarDesc(`*`ESObj`*`)` returns a dataset array containing the sample variable names and descriptions from the MetaData object in an ExpressionSet object.

`NewESObj` `= sampleVarDesc(`*`ESObj`*`,` *`NewDSVarDescriptions`*`)` replaces the sample variable descriptions in *`ESObj`*, an ExpressionSet object, with *`NewDSVarDescriptions`*, and returns *`NewESObj`*, a new ExpressionSet object.

Input Arguments

`ESObj`

Object of the `bioma.ExpressionSet` class.

Default:

`NewDSVarDescriptions`

Descriptions of the sample variable names, specified by either of the following:

- A new dataset array containing the sample variable names and descriptions. In this dataset array, each row corresponds to a variable. The first column contains the variable name, and the second column (`VariableDescription`) contains a description of the variable. The row names (variable names) must match the row names (variable names) in *`DSVarDescriptions`*, the dataset array being replaced in the MetaData object in the ExpressionSet object, *`ESObj`*.
- Cell array of character vectors containing descriptions of the sample variables. The number of elements in *`VarDesc`* must equal the number of row names (variable names) in *`DSVarDescriptions`*, the dataset array being replaced in the MetaData object in the ExpressionSet object, *`ESObj`*.

Default:

Output Arguments

`DSVarDescriptions`

A dataset array containing the sample variable names and descriptions from the MetaData object of an ExpressionSet object. In this dataset array, each row corresponds to a variable. The first column contains the variable name, and the second column (`VariableDescription`) contains a description of the variable.

`NewESObj`

Object of the `bioma.ExpressionSet` class, returned after replacing the dataset array containing the sample variable descriptions.

Examples

Construct an ExpressionSet object, `ESObj`, as described in the “Examples” on page 1-0 section of the `bioma.ExpressionSet` class reference page. Retrieve the sample variable descriptions in the ExpressionSet object:

```
% Retrieve the sample variable descriptions
SVarDescriptions = sampleVarDesc(ESObj)
```

See Also

`bioma.ExpressionSet` | `bioma.data.MetaData` | `variableDesc`

Topics

“Managing Gene Expression Data in Objects”

sampleVarNames

Class: `bioma.ExpressionSet`
Package: `bioma`

Retrieve or set sample variable names in ExpressionSet object

Syntax

```
SamVarNames = sampleVarNames(ESObj)
SamVarNames = sampleVarNames(ESObj, Subset)
NewESObj = sampleVarNames(ESObj, Subset, NewSamVarNames)
```

Description

`SamVarNames = sampleVarNames(ESObj)` returns a cell array of character vectors specifying all sample variable names in an ExpressionSet object.

`SamVarNames = sampleVarNames(ESObj, Subset)` returns a cell array of character vectors specifying a subset the sample variable names in an ExpressionSet object.

`NewESObj = sampleVarNames(ESObj, Subset, NewSamVarNames)` replaces the sample variable names specified by *`Subset`* in *`ESObj`*, an ExpressionSet object, with *`NewSamVarNames`*, and returns *`NewESObj`*, a new ExpressionSet object.

Input Arguments

`ESObj`

Object of the `bioma.ExpressionSet` class.

`Subset`

One of the following to specify a subset of the sample variable names in an ExpressionSet object:

- Character vector or string specifying a sample variable name
- Cell array of character vectors or string vector specifying sample variable names
- Positive integer
- Vector of positive integers
- Logical vector

`NewSamVarNames`

New sample variable names for specific sample variable names within an ExpressionSet object, specified by one of the following:

- Numeric vector
- String vector or cell array of character vectors
- Character vector or string, which `sampleVarNames` uses as a prefix for the sample variable names, with sample variable numbers appended to the prefix

- Logical `true` or `false` (default). If `true`, `sampleVarNames` assigns unique sample variable names using the format `Var1`, `Var2`, etc.

The number of sample variable names in *`NewSamVarNames`* must equal the number of sample variable names specified by *`Subset`*.

Output Arguments

`SamVarNames`

Cell array of character vectors specifying all or some of the sample variable names in an ExpressionSet object. The sample variable names are the column names of the *`VarValues`* dataset array. The sample variable names are also the row names of the *`VarDescriptions`* dataset array. Both dataset arrays are in the MetaData object in the ExpressionSet object.

`NewESObj`

Object of the `bioma.ExpressionSet` class, returned after replacing specific sample names.

Examples

Construct an ExpressionSet object, `ESObj`, as described in the “Examples” on page 1-0 section of the `bioma.ExpressionSet` class reference page. Retrieve the sample variable names from the ExpressionSet object:

```
% Retrieve the sample variable names
VNames = sampleVarNames(ESObj)
```

See Also

`bioma.ExpressionSet` | `bioma.data.MetaData` | `sampleNames` | `featureNames` | `featureVarNames`

Topics

“Managing Gene Expression Data in Objects”

sampleVarValues

Class: `bioma.ExpressionSet`
Package: `bioma`

Retrieve or set sample variable values in ExpressionSet object

Syntax

```
DSVarValues = sampleVarValues(ESObj)
NewESObj = sampleVarValues(ESObj, NewDSVarValues)
```

Description

`DSVarValues = sampleVarValues(ESObj)` returns a dataset array containing the measured value of each variable per sample from the MetaData object of an ExpressionSet object.

`NewESObj = sampleVarValues(ESObj, NewDSVarValues)` replaces the sample variable values in *`ESObj`*, an ExpressionSet object, with *`NewDSVarValues`*, and returns *`NewESObj`*, a new ExpressionSet object.

Input Arguments

`ESObj`

Object of the `bioma.ExpressionSet` class.

Default:

`NewDSVarValues`

A new dataset array containing a value for each variable per sample. In this dataset array, the columns correspond to variables and rows correspond to samples. The row names (sample names) must match the row names (sample names) in *`DSVarValues`*, the dataset array being replaced in the MetaData object in the ExpressionSet object, *`ESObj`*.

Default:

Output Arguments

`DSVarValues`

A dataset array containing the measured value of each variable per sample from the MetaData object of an ExpressionSet object. In this dataset array, the columns correspond to variables and rows correspond to samples.

`NewESObj`

Object of the `bioma.ExpressionSet` class, returned after replacing the dataset array containing the sample variable values.

Examples

Construct an ExpressionSet object, `ESObj`, as described in the “Examples” on page 1-0 section of the `bioma.ExpressionSet` class reference page. Retrieve the sample variable values in ExpressionSet object:

```
% Retrieve the sample variable values
SVarValues = sampleVarValues(ESObj);
```

See Also

`bioma.ExpressionSet` | `bioma.data.MetaData` | `variableValues`

Topics

“Managing Gene Expression Data in Objects”

samread

Read data from SAM file

Syntax

```
SAMStruct = samread(File)
[SAMStruct, HeaderStruct]= samread(File)
... = samread(File,'ParameterName',ParameterValue)
```

Description

`SAMStruct = samread(File)` reads a SAM-formatted file and returns the data in a MATLAB array of structures.

`[SAMStruct, HeaderStruct]= samread(File)` returns the alignment and header data in two separate variables.

`... = samread(File,'ParameterName',ParameterValue)` accepts one or more comma-separated parameter name/value pairs. Specify *`ParameterName`* inside single quotes.

Input Arguments

`File`

Character vector or string specifying a file name, path and file name of a SAM-formatted file, or the text of a SAM-formatted file. If you specify only a file name, that file must be on the MATLAB search path or in the current folder.

Parameter Name/Value Pairs

`Tags`

Controls the reading of the optional tags in addition to the first 11 fields for each alignment in the SAM-formatted file. Choices are `true` (default) or `false`.

Default:

`ReadGroup`

Character vector or string specifying the read group ID for which to read alignment records from. Default is to read records from all groups.

Tip For a list of the read groups (if present), return the header information in a separate *`Header`* structure and view the `ReadGroup` field in this structure.

Default:

`BlockRead`

Scalar or vector that controls the reading of a single sequence entry or block of sequence entries from a SAM-formatted file containing multiple sequences. Enter a scalar *`N`*, to read the *`N`*th entry in the file. Enter a 1-by-2 vector [*`M1, M2`*], to read a block of entries starting at the *`M1`* entry and ending at the *`M2`* entry. To read all remaining entries in the file starting at the *`M1`* entry, enter a positive value for *`M1`* and enter `Inf` for *`M2`*.

Default:

Output Arguments

`SAMStruct`

An *N*-by-1 array of structures containing sequence alignment and mapping information from a SAM-formatted file, where *N* is the number of alignment records stored in the SAM-formatted file. Each structure contains the following fields.

Field	Description
`QueryName`	Name of read sequence (if unpaired) or name of sequence pair (if paired). **Tip** You can use this information to populate the `Header` property of the BioMap object.
`Flag`	Integer indicating the bit-wise information that specifies the status of each of 11 flags described by the SAM format specification. **Tip** You can use the `bitget` function to determine the status of a specific SAM flag.
`ReferenceName`	Name of the reference sequence.
`Position`	Position (one-based offset) of the forward reference sequence where the left-most base of the alignment of the read sequence starts.
`MappingQuality`	Integer specifying the mapping quality score for the read sequence.
`CigarString`	CIGAR-formatted character vector representing how the read sequence aligns with the reference sequence.
`MateReferenceName`	Name of the reference sequence associated with the mate. If this name is the same as `ReferenceName`, then this value is `=`. If there is no mate, then this value is `*`.
`MatePosition`	Position (one-based offset) of the forward reference sequence where the left-most base of the alignment of the mate of the read sequence starts.

Field	Description
`InsertSize`	The number of base positions between the read sequence and its mate, when both are mapped to the same reference sequence. Otherwise, this value is `0`.
`Sequence`	Character vector containing the letter representations of the read sequence. It is the reverse-complement if the read sequence aligns to the reverse strand of the reference sequence.
`Quality`	Character vector containing the ASCII representation of the per-base quality score for the read sequence. The quality score is reversed if the read sequence aligns to the reverse strand of the reference sequence.
`Tags`	List of applicable SAM tags and their values.

`HeaderStruct`

Structure containing header information for the SAM-formatted file in the following fields.

Field	Description
`Header*`	Structure containing the file format version, sort order, and group order.
`SequenceDictionary*`	Structure containing the: • Sequence name • Sequence length • Genome assembly identifier • MD5 checksum of sequence • URI of sequence • Species
`ReadGroup*`	Structure containing the: • Read group identifier • Sample • Library • Description • Platform unit • Predicted median insert size • Sequencing center • Date • Platform

Field	Description
`Program*`	Structure containing the: • Program name • Version • Command line

* — These structures and their fields appear in the output structure only if they are present in the SAM file. The information in these structures depends on the information present in the SAM file.

Examples

Read the header information and the alignment data from the `ex1.sam` file included with Bioinformatics Toolbox, and then return the information in two separate variables:

```
[data header] = samread('ex1.sam');
```

Read a block of entries, excluding the tags, from the `ex1.sam` file, and then return the information in an array of structures:

```
% Read entries 5 through 10 and do not include the tags
data = samread('ex1.sam','blockread', [5 10], 'tags', false);
```

Tips

- Use the `saminfo` function to investigate the size and content of a SAM-formatted file before using the `samread` function to read the file contents into a MATLAB array of structures.
- If your SAM-formatted file is too large to read using available memory, try one of the following:
  - Use the `BlockRead` parameter with the `samread` function to read a subset of entries.
  - Create a BioIndexedFile object from the SAM-formatted file, then access the entries using methods of the `BioIndexedFile` class.
- Use the *`SAMStruct`* output argument that `samread` returns to create a `BioMap` object, which lets you explore, access, filter, and manipulate all or a subset of the data, before doing subsequent analyses or viewing the data.

Version History
Introduced in R2010a

See Also
`saminfo` | `soapread` | `fastqread` | `bamread` | `baminfo` | `fastqwrite` | `fastqinfo` | `fastainfo` | `fastaread` | `fastawrite` | `sffinfo` | `sffread` | `BioIndexedFile` | `BioMap`

Topics
"Manage Sequence Read Data in Objects"
"Work with Next-Generation Sequencing Data"

External Websites
Sequence Read Archive

SAM format specification

samsort

Sort SAM files

Syntax

```
sortedFile = samsort(inFile)
samsort(inFile,outFile)
```

Description

`sortedFile = samsort(inFile)` sorts a SAM file `inFile` and returns the name of the sorted SAM file `sortedFile`. The function sorts the alignment records by the reference sequence name first, and then by position within the reference.

`samsort(inFile,outFile)` sorts `inFile` and produces a sorted SAM file named `outFile`.

Examples

Sort SAM File

Sort a sample SAM file. The sorted file has the same base name as the input file, but with the extension "`.sorted.sam`". By default, the sorted file is saved in the current directory.

```
sortedFile = samsort("ex1.sam")
```

```
sortedFile = 
"ex1.sorted.sam"
```

You can change the name of output file by specifying it as the second input.

```
samsort("ex1.sam","sortedEx1.sam")
```

```
ans = 
"sortedEx1.sam"
```

You can also save the output file to an existing directory by providing the file path information.

```
mkdir("./OutputEx1SAM");
samsort("ex1.sam","./OutputEx1SAM/sortedEx1.sam")
```

```
ans = 
"./OutputEx1SAM/sortedEx1.sam"
```

Input Arguments

`inFile` — Name of input SAM file to sort
character vector | string

Name of the input SAM file to sort, specified as a string or character vector. You can specify a file name or a path and file name. The file name must have the extension `.sam`.

Example: `"./InputData/ex1.sam"`

Data Types: `char` | `string`

`outFile` — Name of output SAM file
character vector | string

Name of the output SAM file, specified as a string or character vector. You can specify a file name or a path and file name. The file name must have the extension `.sam`. The file is saved in the current directory by default unless you specify the path information. If you specify the file path, the listed directories must exist before you run the function.

Example: `"./OutputData/ex1Sorted.sam"`

Data Types: `char` | `string`

Output Arguments

`sortedFile` — Name of output SAM file
string

Name of the output SAM file, returned as a string. `sortedFile` has the same base name as `inFile` but with the extension `.sorted.sam`. The file is saved in the current directory by default.

Version History

Introduced in R2019b

See Also

`bamsort` | `bamread` | `samread` | `saminfo` | `BioMap`

Topics
“Data Import”
“Manage Sequence Read Data in Objects”

External Websites
Sequence Read Archive
SAM format specification

scfread

Read trace data from SCF file

Syntax

```
Sample = scfread(File)
[Sample, Probability] = scfread(File)
[Sample, Probability, Comments] = scfread(File)
[A, C, G, T] = scfread (File)
[A, C, G, T, ProbA, ProbC, ProbG, ProbT] = scfread (File)
[A, C, G, T, ProbA, ProbC, ProbG, ProbT, Comments, PkIndex, Base] = scfread
(File)
```

Arguments

`File`	Character vector or string specifying the file name or a path and file name of an SCF formatted file.

Description

`scfread` reads data from an SCF formatted file into MATLAB structures.

`Sample` `= scfread(`*`File`*`)` reads an SCF formatted file and returns the sample data in the structure `Sample`, which contains the following fields:

Field	Description
`A`	Column vector containing intensity of A fluorescence tag
`C`	Column vector containing intensity of C fluorescence tag
`G`	Column vector containing intensity of G fluorescence tag
`T`	Column vector containing intensity of T fluorescence tag

`[`*`Sample`*`, `*`Probability`*`] = scfread(`*`File`*`)` also returns the probability data in the structure `Probability`, which contains the following fields:

Field	Description
`peak_index`	Column vector containing the position in the SCF file for the start of the data for each peak
`prob_A`	Column vector containing the probability of each base in the sequence being an A
`prob_C`	Column vector containing the probability of each base in the sequence being a C
`prob_G`	Column vector containing the probability of each base in the sequence being a G

Field	Description
prob_T	Column vector containing the probability of each base in the sequence being a T
base	Column vector containing the called bases for the sequence

[*Sample*, *Probability*, *Comments*] = scfread(*File*) also returns the comment information from the SCF file in a character array Comments.

[*A*, *C*, *G*, *T*] = scfread (*File*) returns the sample data for the four bases in separate variables.

[*A*, *C*, *G*, *T*, *ProbA*, *ProbC*, *ProbG*, *ProbT*] = scfread (*File*) also returns the probabilities data for the four bases in separate variables.

[*A*, *C*, *G*, *T*, *ProbA*, *ProbC*, *ProbG*, *ProbT*, *Comments*, *PkIndex*, *Base*] = scfread (*File*) also returns the peak indices and called bases in separate variables.

SCF files store data from DNA sequencing instruments. Each file includes sample data, sequence information, and the relative probabilities of each of the four bases.

Examples

```
[sampleStruct, probStruct, Comments] = scfread('sample.scf')
sampleStruct =

    A: [10827x1 double]
    C: [10827x1 double]
    G: [10827x1 double]
    T: [10827x1 double]


probStruct =

    peak_index: [742x1 double]
        prob_A: [742x1 double]
        prob_C: [742x1 double]
        prob_G: [742x1 double]
        prob_T: [742x1 double]
          base: [742x1 char]


Comments =

SIGN=A=121,C=103,G=119,T=82
SPAC= 16.25
PRIM=0
MACH=Arkansas_SN312
DYEP=DT3700POP5{BD}v2.mob
NAME=HCIUP1D61207
LANE=6
GELN=
PROC=
RTRK=
CONV=phred version=0.990722.h
```

```
COMM=
SRCE=ABI 373A or 377
```

Version History

Introduced before R2006a

See Also

`genbankread` | `traceplot`

select (phytree)

Select tree branches and leaves in phytree object

Syntax

```
S = select(Tree, N)
[S, Selleaves, Selbranches] = select(...)

select(..., 'Reference', ReferenceValue, ...)
select(..., 'Criteria', CriteriaValue, ...)
select(..., 'Threshold', ThresholdValue, ...)
select(..., 'Exclude', ExcludeValue, ...)
select(..., 'Propagate', PropagateValue, ...)
```

Arguments

Tree	Phylogenetic tree (`phytree` object) created with the function `phytree`.
N	Number of closest nodes to the root node.
ReferenceValue	Property to select a reference point for measuring distance.
CriteriaValue	Property to select a criteria for measuring distance.
ThresholdValue	Property to select a distance value. Nodes with distances below this value are selected.
ExcludeValue	Property to remove (exclude) branch or leaf nodes from the output. Enter `'none'`, `'branches'`, or `'leaves'`. The default value is `'none'`.
PropagateValue	Property to select propagating nodes toward the leaves or the root.
S	Logical vector for all selected nodes.
Selleaves	Logical vector for selected leaves.
Selbranches	Logical vector for selected branches.

Description

`S = select(Tree, N)` returns a logical vector (*S*) of size `[NumNodes x 1]` indicating the *N* closest nodes to the root node of a `phytree` object (`Tree`) where `NumNodes = NumLeaves + NumBranches`. The first criterion used is branch levels, then patristic distance (also known as tree distance). By default, `select` uses `Inf` as the value of `N`, and `select(Tree)` returns a vector with values of `true`.

`[S, Selleaves, Selbranches] = select(...)` returns two additional logical vectors, one for the selected leaves and one for the selected branches.

`select(..., 'PropertyName', PropertyValue, ...)` uses additional options specified as one or more name-value pair arguments. Each *PropertyName* must be enclosed in single quotation marks and is case insensitive. These name-value pairs are as follows:

`select(..., 'Reference', ReferenceValue, ...)` changes the reference point(s) to measure the closeness. *ReferenceValue* can be `'root'` (default) or `'leaves'` or an index that points to any node of the tree. When using `'leaves'`, a node can have multiple distances to its descendant leaves (nonultrametric tree). If so, `select` considers the minimum distance to any descendant leaf.

`select(..., 'Criteria', CriteriaValue, ...)` changes the criteria used to measure closeness. If *CriteriaValue* `= 'levels'` (default), the first criterion is branch levels and then patristic distance. If *CriteriaValue* `= 'distance'`, the first criterion is patristic distance and then branch levels.

`select(..., 'Threshold', ThresholdValue, ...)` selects all the nodes where closeness is less than or equal to the threshold value (*ThresholdValue*). You can use either `'Criteria'` or `'Reference'` in conjunction with this name-value pair. If `N` is not specified, then `N = Inf`. Otherwise you can limit the number of selected nodes by `N`.

`select(..., 'Exclude', ExcludeValue, ...)` sets a postfilter which excludes all the branch nodes from *S* when *ExcludeValue* `= 'branches'` or excludes all the leave nodes when *ExcludeValue* `= 'leaves'`. The default is `'none'`.

`select(..., 'Propagate', PropagateValue, ...)` activates a postfunctionality that propagates the selected nodes to the leaves when *PropagateValue* is set to `'toleaves'` or toward the root finding a common ancestor when *PropagateValue* is set to `'toroot'`. The default value is `'none'`. *PropagateValue* may also be `'both'`. The `'Propagate'` property acts after the `'Exclude'` name-value pair.

Examples

```
% Load a phylogenetic tree created from a protein family:
tr = phytreeread('pf00002.tree');

% To find close products for a given protein (e.g. vipr2_human):
ind = getbyname(tr,'vipr2_human');
[sel,sel_leaves] = select(tr,'criteria','distance',...
                          'threshold',0.6,'reference',ind);
view(tr,sel_leaves)

% To find potential outliers in the tree, use
[sel,sel_leaves] = select(tr,'criteria','distance',...
                             'threshold',.3,...
                             'reference','leaves',...
                             'exclude','leaves',...
                             'propagate','toleaves');
view(tr,~sel_leaves)
```

Version History

Introduced before R2006a

See Also

`phytree` | `phytreeviewer` | `get` | `pdist` | `prune`

Topics
phytree object on page 1-1449

seq2regexp

Convert sequence with ambiguous characters to regular expression

Syntax

```
RegExp = seq2regexp(Seq)

RegExp = seq2regexp(Seq, ...'Alphabet', AlphabetValue, ...)
RegExp = seq2regexp(Seq, ...'Ambiguous', AmbiguousValue, ...)
```

Input Arguments

Seq	Either of the following: • Character vector or string containing codes specifying an amino acid or nucleotide sequence. • Structure containing a `Sequence` field that contains an amino acid or nucleotide sequence, such as returned by `fastaread`, `fastqread`, `getembl`, `getgenbank`, `getgenpept`, or `getpdb`.
AlphabetValue	Character vector or string specifying the sequence alphabet. Choices are: • `'NT'` (default) — Nucleotide • `'AA'` — Amino acid
AmbiguousValue	Controls whether ambiguous characters are included in *RegExp*, the regular expression return value. Choices are: • `true` (default) — Include ambiguous characters in the return value • `false` — Return only unambiguous characters

Output Arguments

RegExp	Character vector of codes specifying an amino acid or nucleotide sequence in regular expression format using IUB/IUPAC codes.

Description

`RegExp = seq2regexp(Seq)` converts ambiguous amino acid or nucleotide symbols in a sequence to a regular expression format using IUB/IUPAC codes.

`RegExp = seq2regexp(Seq, ...'PropertyName', PropertyValue, ...)` calls `seq2regexp` with optional properties that use property name/property value pairs. You can specify one or more properties in any order. Each *PropertyName* must be enclosed in single quotation marks and is case insensitive. These property name/property value pairs are as follows:

`RegExp = seq2regexp(Seq, ...'Alphabet', AlphabetValue, ...)` specifies the sequence alphabet. *AlphabetValue* can be either 'NT' for nucleotide sequences or 'AA' for amino acid sequences. Default is 'NT'.

RegExp = seq2regexp(*Seq*, ...'Ambiguous', *AmbiguousValue*, ...) controls whether ambiguous characters are included in *RegExp*, the regular expression return value. Choices are true (default) or false. For example:

- If *Seq* = 'ACGTK', and *AmbiguousValue* is true , the MATLAB software returns ACGT[GTK] with the unambiguous characters G and T and the ambiguous character K.
- If *Seq* = 'ACGTK', and *AmbiguousValue* is false, the MATLAB software returns ACGT[GT] with only the unambiguous characters.

Nucleotide Conversion

Nucleotide Code	Nucleotide	Conversion
A	Adenosine	A
C	Cytosine	C
G	Guanine	G
T	Thymidine	T
U	Uridine	U
R	Purine	[AG]
Y	Pyrimidine	[TC]
K	Keto	[GT]
M	Amino	[AC]
S	Strong interaction (3 H bonds)	[GC]
W	Weak interaction (2 H bonds)	[AT]
B	Not A	[CGT]
D	Not C	[AGT]
H	Not G	[ACT]
V	Not T or U	[ACG]
N	Any nucleotide	[ACGT]
-	Gap of indeterminate length	-
?	Unknown	?

Amino Acid Conversion

Amino Acid Code	Amino Acid	Conversion
B	Asparagine or Aspartic acid (Aspartate)	[DN]
Z	Glutamine or Glutamic acid (Glutamate)	[EQ]
X	Any amino acid	[A R N D C Q E G H I L K M F P S T W Y V]

Examples

1 Convert a nucleotide sequence to a regular expression.

```
seq2regexp('ACWTMAN')

ans =
AC[ATW]T[ACM]A[ACGTRYKMSWBDHVN]
```

2 Convert the same nucleotide sequence, but remove ambiguous characters from the regular expression.

```
seq2regexp('ACWTMAN', 'ambiguous', false)

ans =
AC[AT]T[AC]A[ACGT]
```

Version History

Introduced before R2006a

See Also

`restrict` | `seqwordcount` | `regexp` | `regexpi`

seqalignviewer

Visualize and edit multiple sequence alignment

Syntax

```
seqalignviewer
seqalignviewer(Alignment)
seqalignviewer(Alignment,Name,Value)
```

Description

`seqalignviewer` opens the Sequence Alignment app, where you can display and interactively adjust multiple sequence alignments.

`seqalignviewer(Alignment)` loads a group of previously multiply aligned sequences into the app, where you can view and interactively adjust the alignment.

`seqalignviewer(Alignment,Name,Value)` opens the app with additional options specified by one or more `Name,Value` pair arguments.

Tip If gaps are available after you have selected a block from aligned sequences, then there are three regions that you can drag and move horizontally:

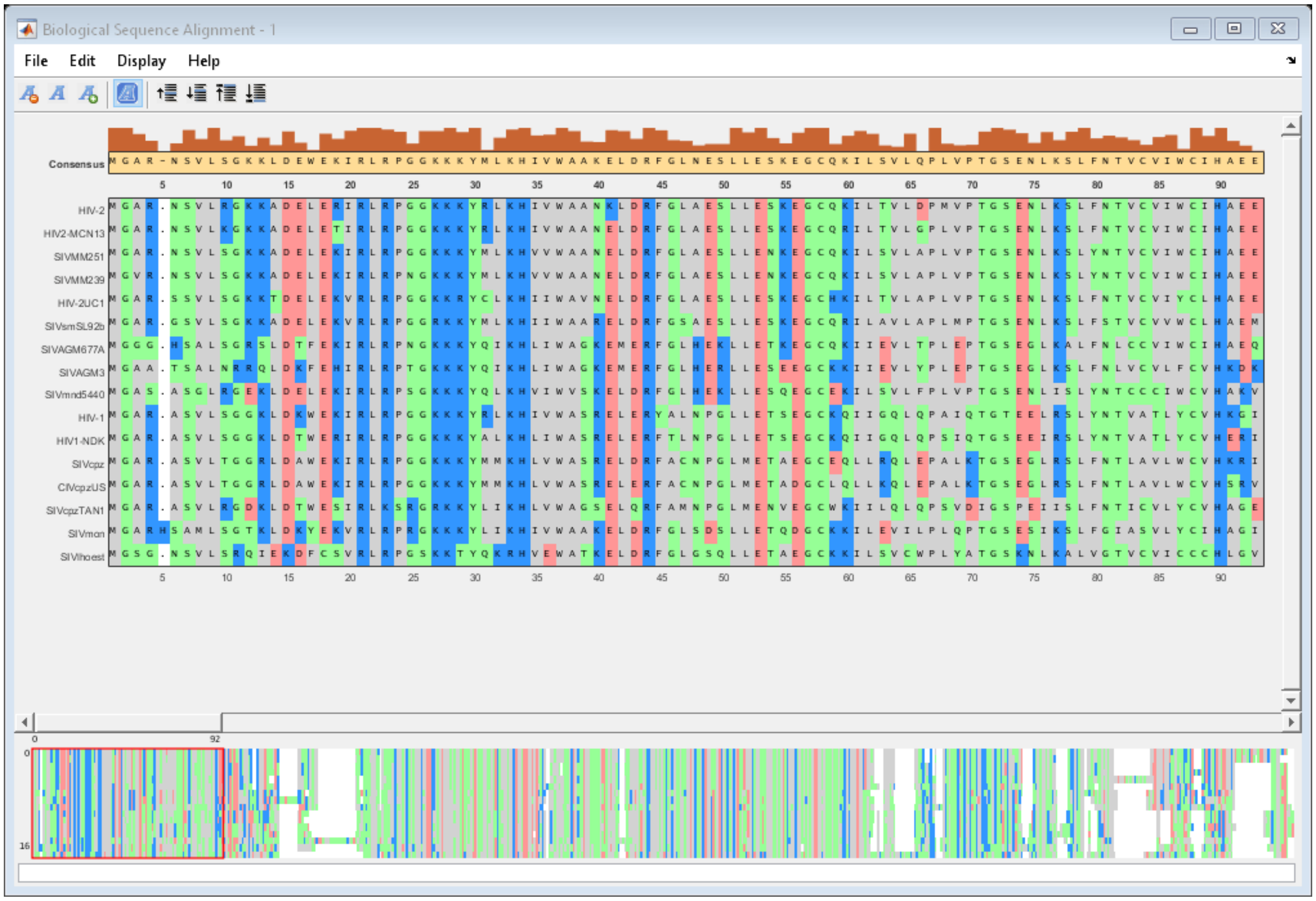
- Selected block
- Block on the left of the selection
- Block on the right of the selection

Examples

View a Multiple Sequence Alignment File

Load and view a multiple sequence alignment file.

```
seqalignviewer('aagag.aln')
```



Alternatively, you can click Sequence Alignment on the **Apps** tab to open the app, and view the alignment data.

You can also generate a phylogenetic tree from aligned sequences from within the app. Select **Display** > **View Tree** .

Input Arguments

`Alignment` — Multiple sequence alignment (MSA) data

structure | character array | character vector | string vector | 3-by-N character array

Multiple sequence alignment (MSA) data, specified as:

- MATLAB structure containing a `Sequence` field, such as returned by `fastaread`, `gethmmalignment`, `multialign`, or `multialignread`
- Character array or column vector of strings containing MSA data, such as returned by `multialign`
- Character vector specifying a file or URL containing MSA data
- 3-by-N character array showing the pairwise alignment of two sequences, such as returned by `nwalign` or `swalign`.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `'Alphabet','AA'` specifies that the aligned sequences are amino acid sequences.

`Alphabet` — Type of aligned sequences
`'AA'` | `'NT'`

Type of aligned sequences, specified as `'AA'` for amino acid sequences or `'NT'` for nucleotide sequences. If you do not specify the type, `seqalignviewer` attempts to determine the correct type. If it cannot, it defaults to `'AA'`.

Example: `'Alphabet','AA'`

`SeqHeaders` — List of names to label sequences in alignment window
array of structures containing a `Header` or `Name` field | cell array of character vectors | string vector

List of names to label the sequences in the alignment window, specified as a MATLAB array of structures containing a `Header` or `Name` field, cell array of character vectors, or string vector. The number of elements in either array must be the same as the number of sequences in the alignment data `Alignment`.

Example: `'SeqHeaders',names`

Version History
Introduced in R2012b

R2017b: `'R2012b'` option in `seqalignviewer` has been removed
Errors starting in R2017b

The `'R2012b'` name-value pair has been removed. The default version of `seqalignviewer` runs more robustly than the previous version (R2012b).

See Also
`cigar2align` | `fastaread` | `gethmmalignment` | `multialign` | `multialignread` | `multialignwrite` | `seqviewer` | `nwalign` | `swalign`

Topics
“View and Align Multiple Sequences”
“Investigating the Bird Flu Virus”

seqcomplement

Calculate complementary strand of nucleotide sequence

Syntax

SeqC = `seqcomplement(`*SeqNT*`)`

Arguments

SeqNT	Nucleotide sequence specified by any of the following: • Character vector or string containing the letters`A`, `C`, `G`, `T`, `U`, and ambiguous characters `R`, `Y`, `K`, `M`, `S`, `W`, `B`, `D`, `H`, `V`, `N`. • Row vector of integers from the table Mapping Nucleotide Integers to Letter Codes. • MATLAB structure containing a `Sequence` field that contains a nucleotide sequence, such as returned by `emblread`, `fastaread`, `fastqread`, `genbankread`, `getembl`, or `getgenbank`.

Description

SeqC = `seqcomplement(`*SeqNT*`)` calculates the complementary strand of a DNA or RNA nucleotide sequence. The return sequence, *SeqC*, is in the same format as *SeqNT*. For example, if *SeqNT* is a vector of integers, then so is *SeqC*.

Nucleotide in ***SeqNT***	Converts to This Nucleotide in ***SeqC***
`A`	`T` or `U`
`C`	`G`
`G`	`C`
`T` or `U`	`A`

Examples

Return the complement of a DNA nucleotide sequence.

```
s = 'ATCG';
seqcomplement(s)

ans =
TAGC
```

Version History

Introduced before R2006a

See Also

`codoncount` | `palindromes` | `seqrcomplement` | `seqreverse` | `seqviewer`

seqconsensus

Calculate consensus sequence

Syntax

```
CSeq = seqconsensus(Seqs)
[CSeq, Score] = seqconsensus(Seqs)
CSeq = seqconsensus(Profile)

seqconsensus(..., 'PropertyName', PropertyValue,...)
seqconsensus(..., 'ScoringMatrix', ScoringMatrixValue)
```

Arguments

`Seqs`	Set of multiply aligned amino acid or nucleotide sequences. Enter a character array, string vector, cell array of character vectors, or an array of structures with the field `Sequence`.
`Profile`	Sequence profile. Enter a profile from the function `seqprofile`. `Profile` is a matrix of size `[20 (or 4) x Sequence Length]` with the frequency or count of amino acids (or nucleotides) for every position. `Profile` can also have 21 (or 5) rows if gaps are included in the consensus.

ScoringMatrixValue	Either of the following: • Character vector or string specifying the scoring matrix to use for the alignment. Choices for amino acid sequences are: • `'BLOSUM62'` • `'BLOSUM30'` increasing by `5` up to `'BLOSUM90'` • `'BLOSUM100'` • `'PAM10'` increasing by `10` up to `'PAM500'` • `'DAYHOFF'` • `'GONNET'` Default is: • `'BLOSUM50'` — When *AlphabetValue* equals `'AA'` • `'NUC44'` — When *AlphabetValue* equals `'NT'` **Note** The above scoring matrices, provided with the software, also include a structure containing a scale factor that converts the units of the output score to bits. You can also use the `'Scale'` property to specify an additional scale factor to convert the output score from bits to another unit. • A `21x21`, `5x5`, `20x20`, or `4x4` numeric array. For the gap-included cases, gap scores (last row/column) are set to `mean(diag(ScoringMatrix))` for a gap matching with another gap, and set to `mean(nodiag(ScoringMatrix))` for a gap matching with another symbol. **Note** If you use a scoring matrix that you created, the matrix does not include a scale factor. The output score will be returned in the same units as the scoring matrix. **Note** If you need to compile `seqconsensus` into a stand-alone application or software component using MATLAB Compiler, use a matrix instead of a character vector or string for *ScoringMatrixValue*.

Description

CSeq = `seqconsensus(`*Seqs*`)`, for a multiply aligned set of sequences (*Seqs*), returns a character vector with the consensus sequence (*CSeq*). The frequency of symbols (`20` amino acids, `4` nucleotides) in the set of sequences is determined with the function `seqprofile`. For ambiguous nucleotide or amino acid symbols, the frequency or count is added to the standard set of symbols.

[*CSeq*, *Score*] = `seqconsensus(`*Seqs*`)` returns the conservation score of the consensus sequence. Scores are computed with the scoring matrix `BLOSUM50` for amino acids or `NUC44` for nucleotides. Scores are the average euclidean distance between the scored symbol and the M-dimensional consensus value. `M` is the size of the alphabet. The consensus value is the profile weighted by the scoring matrix.

`*CSeq* = seqconsensus(*Profile*)` returns a character vector with the consensus sequence (`*CSeq*`) from a sequence profile (`*Profile*`).

`seqconsensus(..., '*PropertyName*', *PropertyValue*,...)` defines optional properties using property name/value pairs.

`seqconsensus(..., 'ScoringMatrix', *ScoringMatrixValue*)` specifies the scoring matrix.

The following input parameters are analogous to the function `seqprofile` when the alphabet is restricted to `'AA'` or `'NT'`.

`seqconsensus(..., 'Alphabet', *AlphabetValue*)`

`seqconsensus(..., 'Gaps', *GapsValue*)`

`seqconsensus(..., 'Ambiguous', *AmbiguousValue*)`

`seqconsensus(..., 'Limits', *LimitsValue*)`

Examples

```
seqs = fastaread('pf00002.fa');
[C,S] = seqconsensus(seqs,'limits',[50 60],'gaps','all')
```

Version History

Introduced before R2006a

See Also

`fastaread` | `multialignread` | `multialignwrite` | `profalign` | `seqdisp` | `seqprofile`

seqdisp

Format long sequence output for easy viewing

Syntax

```
seqdisp(Seq)
seqdisp(Seq, ...'Row', RowValue, ...)
seqdisp(Seq, ...'Column', ColumnValue, ...)
seqdisp(Seq, ...'ShowNumbers', ShowNumbersValue, ...)
```

Arguments

`Seq`	Nucleotide or amino acid sequence represented by any of the following: • Character array • String vector • Character vector containing the FASTA file name • MATLAB structure with the field `Sequence` Multiply aligned sequences are allowed. FASTA files can have the file extension `fa`, `fasta`, `fas`, `fsa`, or `fst`.
`RowValue`	Integer that specifies the length of each row. Default is `60`.
`ColumnValue`	Integer that specifies the column width or number of symbols before displaying a space. Default is `10`.
`ShowNumbersValue`	Controls the display of numbers at the start of each row. Choices are `true` (default) to show numbers, or `false` to hide numbers.

Description

`seqdisp(`*`Seq`*`)` displays a sequence in rows, with a default row length of 60 and a default column width of `10`.

`seqdisp(`*`Seq`*`, ...'`*`PropertyName`*`', `*`PropertyValue`*`, ...)` calls `seqdisp` with optional properties that use property name/property value pairs. You can specify one or more properties in any order. Each *`PropertyName`* must be enclosed in single quotation marks and is case insensitive. These property name/property value pairs are as follows:

`seqdisp(`*`Seq`*`, ...'Row', `*`RowValue`*`, ...)` specifies the length of each row for the displayed sequence.

`seqdisp(`*`Seq`*`, ...'Column', `*`ColumnValue`*`, ...)` specifies the number of letters to display before adding a space. *`RowValue`* must be larger than and evenly divisible by *`ColumnValue`*.

`seqdisp(`*`Seq`*`, ...'ShowNumbers', `*`ShowNumbersValue`*`, ...)` controls the display of numbers at the start of each row. Choices are `true` (default) to show numbers, or `false` to hide numbers.

Examples

Read sequence information from the GenBank database. Display the sequence in rows with 50 letters, and within a row, separate every 10 letters with a space.

```
mouseHEXA = getgenbank('AK080777');
seqdisp(mouseHEXA, 'Row', 50, 'Column', 10)
```

Create and save a FASTA file with two sequences, and then display it.

```
hdr = ['Sequence A'; 'Sequence B'];
seq = ['TAGCTGRCCAAGGCCAAGCGAGCTTN';'ATCGACYGGTTCCGGTTCGCTCGAAN']
fastawrite('local.fa', hdr, seq);
seqdisp('local.fa', 'ShowNumbers', false')

ans =
>Sequence A
 1  TAGCTGRCCA AGGCCAAGCG AGCTTN
>Sequence B
 1  ATCGACYGGT TCCGGTTCGC TCGAAN
```

Version History

Introduced before R2006a

See Also

multialignread | multialignwrite | seqconsensus | seqlogo | seqprofile | seqshoworfs | seqviewer | getgenbank

seqdotplot

Create dot plot of two sequences

Syntax

```
seqdotplot(Seq1, Seq2)
seqdotplot(Seq1,Seq2, Window, Number)
Matches = seqdotplot(...)
[Matches, Matrix] = seqdotplot(...)
```

Arguments

Seq1, *Seq2*	Nucleotide or amino acid sequences. Enter a character vector or string for each sequence. Do not enter a vector of integers. You can also enter a structure with the field `Sequence`.
Window	Enter an integer for the size of a window.
Number	Enter an integer for the number of characters within the window that match.

Description

`seqdotplot(`*`Seq1`*`, `*`Seq2`*`)` plots a figure that visualizes the match between two sequences.

`seqdotplot(`*`Seq1`*`,`*`Seq2`*`, `*`Window`*`, `*`Number`*`)` plots sequence matches when there are at least *`Number`* matches in a window of size *`Window`*.

When plotting nucleotide sequences, start with a `Window` of `11` and `Number` of `7`.

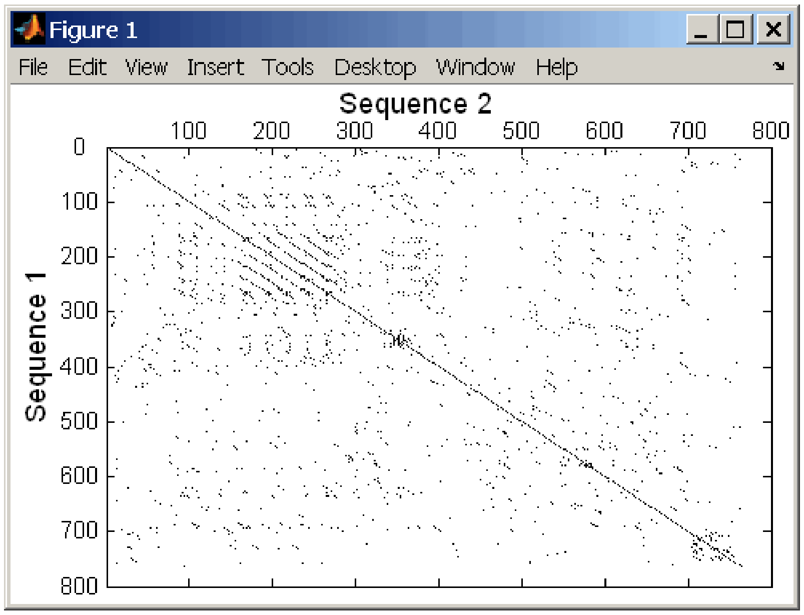
`Matches`` = seqdotplot(...)` returns the number of dots in the dot plot matrix.

`[`*`Matches`*`, `*`Matrix`*`] = seqdotplot(...)` returns the dot plot as a sparse matrix.

Examples

This example shows the similarities between the prion protein (PrP) nucleotide sequences of two ruminants, the moufflon and the golden takin.

```
moufflon = getgenbank('AB060288','Sequence',true);
takin = getgenbank('AB060290','Sequence',true);
seqdotplot(moufflon,takin,11,7)
```



Note For the correct interpretation of a dot plot, your monitor's display resolution must be able to contain the sequence lengths. If the resolution is not adequate, `seqdotplot` resizes the image and returns a warning.

```
Matches = seqdotplot(moufflon,takin,11,7)
Matches =
        5552

[Matches, Matrix] = seqdotplot(moufflon,takin,11,7)
```

Version History

Introduced before R2006a

See Also

`nwalign` | `swalign`

seqfilter

Filter out sequences based on specified criterion

Syntax

```
seqfilter(fastqFile)
seqfilter(fastqFile,Name,Value)
[outFiles,nSeqIn,nSeqOut] = seqfilter( ___ )
```

Description

`seqfilter(fastqFile)` applies a filtering criterion to the sequences in `fastqFile` and saves the sequences that meet the criterion in a new FASTQ file. By default, the sequences that pass the criterion are saved under file names with the suffix `'_filtered'` appended. If you do not specify any criterion, the function filters sequences using the default.

`seqfilter(fastqFile,Name,Value)` uses additional options specified by one or more `Name,Value` pair arguments.

`[outFiles,nSeqIn,nSeqOut] = seqfilter( ___ )` returns a cell array `outFiles` with the names of output files. `nSeqIn` and `nSeqOut` represent the numbers of sequences included and excluded from each input file, respectively.

Examples

Filter next-generation sequencing data

Filter out sequences with more than 10% of low quality bases, where a base is considered low quality when its quality score is less than 20.

```
 [outFile,in,out] = seqfilter('SRR005164_1_50.fastq',...
                              'Method','MaxPercentLowQualityBases',...
                              'Threshold',[10 20]) ;
```

Check the number of sequences saved in the output file.

```
in
```

```
in = 39
```

Check the number of sequences filtered out.

```
out
```

```
out = 11
```

Filter out sequences having an average quality score of below 20.

```
[outFile,in,out] = seqfilter('SRR005164_1_50.fastq',...
                             'Method','MeanQuality',...
                             'Threshold',20);
```

Apply the filtering criterion to every 10 bases as a sliding window.

```
[outFile,in,out] = seqfilter('SRR005164_1_50.fastq',...
                             'Method','MeanQuality',...
                             'Threshold',20,'WindowSize',10);
```

Filter out sequences with less than 100 bases.

```
[outFile,in,out] = seqfilter('SRR005164_1_50.fastq',...
                             'Method','MinLength',...
                             'Threshold',100);
```

Input Arguments

`fastqFile` — Names of FASTQ files with sequence and quality information

character vector | string | string vector | cell array of character vectors

Names of FASTQ-formatted files with sequence and quality information, specified as a character vector, string, string vector, or cell array of character vectors.

Example: `'SRR005164_1_50.fastq'`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `'Method','MaxNumberLowQualityBases','Threshold',[5 15]` specifies to filter out sequences with a total of more than 5 low-quality bases, where a base is considered a low-quality base if its quality score is less than 15.

`Method` — Criterion to filter sequences

`'MaxNumberLowQualityBases'` (default) | `'MaxPercentLowQualityBases'` | `'MeanQuality'` | `'MinLength'`

Criterion to filter sequences, specified as one of the following options. Specify only one filtering criterion per function call.

- `'MaxNumberLowQualityBases'`– applies a maximum threshold on the number of low-quality bases allowed.
- `'MaxPercentLowQualityBases'`– applies a maximum threshold on the percentage of low-quality bases allowed.
- `'MeanQuality'`– applies a minimum threshold on the average base quality across each sequence.
- `'MinLength'`– applies a minimum threshold on the sequence length.

Use this name-value pair argument together with `'Threshold'` to specify the appropriate threshold value. Depending on the filtering criterion, the corresponding value for `'Threshold'` can be a scalar or two-element vector. See the `'Threshold'` option for the default values. If you do not specify `'Threshold'`, then the function uses the default threshold value of the specified method. For each filtering criterion, the function uses the base quality encoding format specified by the `'Encoding'` name-value pair argument.

`Threshold` — Threshold value for filtering criterion
scalar | vector

Threshold value for the filtering criterion, specified as a scalar or vector. Use this name-value pair to define the threshold value for the filtering criterion specified by `'Method'`.

Depending on the filtering criterion, the corresponding value for `'Threshold'` can be a scalar or two-element vector. If you do not specify `'Threshold'`, then the function uses the default threshold value of the corresponding method. For each filtering criterion, the function uses the encoding format of the base quality specified by the `'Encoding'` name-value pair argument.

`'Method'`	`'Threshold'`	Default `'Threshold'` value
`'MaxNumberLowQualityBases'`	Two-element vector `[`*V1 V2*`]`. *V1* is a nonnegative integer that specifies the maximum number of low-quality bases allowed. *V2* specifies the minimum base quality. Any base with quality less than *V2* is considered a low-quality base. Any sequence containing a number of low-quality bases greater than *V1* is filtered out and not saved in the output file.	`[0 10]`
`'MaxPercentLowQualityBases'`	Two-element vector `[`*V1 V2*`]`. *V1* is a scalar between 0 and 100 that specifies the maximum percentage of low-quality bases allowed. *V2* specifies the minimum base quality. Any base with quality less than *V2* is considered a low-quality base. Any sequence containing a percentage of low-quality bases greater than *V1* is filtered out and not saved in the output file.	`[0 10]`
`'MeanQuality'`	Positive scalar that specifies the minimum threshold on the average base quality across each sequence. Any sequence with average base quality less than this value is filtered out.	`0`
`'MinLength'`	Nonnegative integer that specifies the minimum threshold on the sequence length allowed. Any sequence with length less than this value is filtered out.	`1`

`WindowSize` — Size of sliding window to apply filtering criterion to sequence
`Inf` (default) | positive integer

Size of the sliding window to apply the filtering criterion to a sequence, specified as a positive integer. The size of the window corresponds to the number of bases that the function uses at one time to apply the criterion. If any window fails the criterion, the whole sequence is discarded.

The default is `Inf`, that is, the filtering criterion is applied to the whole sequence.

`Encoding` — Base quality encoding format
`'Illumina18'` (default) | `'Sanger'` | `'Solexa'` | `'Illumina13'` | `'Illumina15'`

Base quality encoding format, specified as a character vector or string.

`OutputDir` — Relative or absolute path to output file directory
character vector | string

Relative or absolute path to the output file directory, specified as a character vector or string. The default is the current directory.

Example: `'OutputDir','F:\results'`

`OutputSuffix` — Suffix to use in output file name
`'_filtered'` (default) | character vector | string

Suffix to use in the output file name, specified as a character vector or string. It is inserted after the input file name and before the file extension. The default is `'_filtered'`.

`PairedFiles` — Whether to consider input files as pairs for paired-end sequence data
`false` (default) | `true`

Whether to consider the input files as pairs for paired-end sequence data, specified as `true` or `false`.

If `true`, the input files are read as pairs, and the sequence data is maintained in sync between the files. That is, if a sequence is filtered out in the first file, the corresponding sequence in the paired file is also filtered out.

`WriteSingleton` — Whether to save singleton sequences in a separate output file
`false` (default) | `true`

Whether to save singleton sequences in a separate output file, specified as `true` or `false`. To set this to `true`, the `'PairedFiles'` option must also be set to `true`.

A singleton sequence is the sequence that pass the filtering criterion but its corresponding sequence in the paired file does not. If `true`, singleton sequences are saved in a separate file with the suffix `'_singleton'`. The default is `false`, meaning that, only sequences that pass the filtering criterion in both input files of a given pair are saved in the output files.

`UseParallel` — Boolean indicating whether to perform computation in parallel
`false` (default) | `true`

Boolean indicating whether to perform computation in parallel, specified as `true` or `false`.

For parallel computing, you must have Parallel Computing Toolbox. If a parallel pool does not exist, one is created automatically when the auto-creation option is enabled in your parallel preferences. Otherwise, computation runs in serial mode.

Note There is a cost associated with sharing large input files across workers in a distributed environment. In some cases, running in parallel may not be beneficial in terms of performance.

Example: `'UseParallel',true`

`OverWrite` — Flag to overwrite existing files
`false` or 0 (default) | `true` or 1

Flag to overwrite existing files, specified as a numeric or logical 1 (`true`) or 0 (`false`).

When the value is `false` and a file matching one of the output file names already exists, the function generates an error.

Data Types: `double` | `logical`

Output Arguments

`outFiles` — Output file names
cell array of character vectors

Output file names, returned as a cell array of character vectors.

`nSeqIn` — Number of sequences selected from each input file
scalar | vector

Number of sequences selected from each input file, returned as a scalar or an *n*-by-`1` vector where *n* is the number of input files. If there are multiple input files, the order within `nSeqIn` corresponds to the order of the input files.

`nSeqOut` — Number of sequences excluded from each input file
scalar | vector

Number of sequences excluded from each input file, returned as a scalar or an *n*-by-`1` vector where *n* is the number of input files. If there are multiple input files, the order within `nSeqOut` corresponds to the order of the input files.

Version History

Introduced in R2016b

Extended Capabilities

Automatic Parallel Support
Accelerate code by automatically running computation in parallel using Parallel Computing Toolbox™.

To run in parallel, set `'UseParallel'` to `true`.

For more information, see the `'UseParallel'` name-value pair argument.

See Also

`seqtrim` | `seqsplit` | `seqsplitpe`

seqinsertgaps

Insert gaps into nucleotide or amino acid sequence

Syntax

```
NewSeq = seqinsertgaps(Seq, Positions)
NewSeq = seqinsertgaps(Seq, GappedSeq)
NewSeq = seqinsertgaps(Seq, GappedSeq, Relationship)
```

Input Arguments

Seq	Either of the following: • Character vector or string specifying a nucleotide or amino acid sequence • MATLAB structure containing a `Sequence` field
Positions	Vector of integers to specify the positions in *Seq* before which to insert a gap.
GappedSeq	Either of the following: • Character vector or string specifying a nucleotide or amino acid sequence • MATLAB structure containing a `Sequence` field
Relationship	Integer specifying the relationship between *Seq* and *GappedSeq*. Choices are: • `1` — Both sequences use the same alphabet, that is both are nucleotide sequences or both are amino acid sequences. • `3` — *Seq* contains nucleotides representing codons and *GappedSeq* contains amino acids (default).

Output Arguments

NewSeq	Sequence with gaps inserted, represented by a character vector specifying a nucleotide or amino acid sequence.

Description

NewSeq = `seqinsertgaps(`*Seq*, *Positions*`)` inserts gaps in the sequence *Seq* before the positions specified by the integers in the vector *Positions*.

NewSeq = `seqinsertgaps(`*Seq*, *GappedSeq*`)` finds the gap positions in the sequence *GappedSeq*, then inserts gaps in the corresponding positions in the sequence *Seq*.

NewSeq = `seqinsertgaps(`*Seq*, *GappedSeq*, *Relationship*`)` specifies the relationship between *Seq* and *GappedSeq*. Enter `1` for *Relationship* when both sequences use the same alphabet, that is both are nucleotide sequences or both are amino acid sequences. Enter `3` for *Relationship* when *Seq* contains nucleotides representing codons and *GappedSeq* contains amino acids. Default is `3`.

Examples

1. Retrieve two nucleotide sequences from the GenBank database for the neuraminidase (NA) protein of two strains of the Influenza A virus (H5N1).

   ```
    hk01 = getgenbank('AF509094');
    vt04 = getgenbank('DQ094287');
   ```
2. Extract the coding region from the two nucleotide sequences.

   ```
   hk01_cds = featureparse(hk01,'feature','CDS','Sequence',true);
   vt04_cds = featureparse(vt04,'feature','CDS','Sequence',true);
   ```
3. Align the amino acids sequences converted from the nucleotide sequences.

   ```
    [sc,al]=nwalign(nt2aa(hk01_cds),nt2aa(vt04_cds),'extendgap',1);
   ```
4. Use the `seqinsertgaps` function to copy the gaps from the aligned amino acid sequences to their corresponding nucleotide sequences, thus codon-aligning them.

   ```
    hk01_aligned = seqinsertgaps(hk01_cds,al(1,:))
    vt04_aligned = seqinsertgaps(vt04_cds,al(3,:))
   ```
5. Once you have code aligned the two sequences, you can use them as input to other functions such as `dnds`, which calculates the synonymous and nonsynonymous substitutions rates of the codon-aligned nucleotide sequences. By setting `Verbose` to `true`, you can also display the codons considered in the computations and their amino acid translations.

   ```
   [dn,ds] = dnds(hk01_aligned,vt04_aligned,'verbose',true)
   ```

Version History

Introduced in R2007a

See Also

`featureparse` | `dnds` | `dndsml` | `int2aa` | `int2nt`

seqlinkage

Construct phylogenetic tree from pairwise distances

Syntax

```
PhyloTree = seqlinkage(Distances)
PhyloTree = seqlinkage(Distances, Method)
PhyloTree = seqlinkage(Distances, Method, Names)
```

Arguments

`Distances`	Matrix or vector of pairwise distances, such as returned by the `seqpdist` function.
`Method`	Character vector or string that specifies a distance method. Choices are: • `'single'` • `'complete'` • `'average'` (default) • `'weighted'` • `'centroid'` • `'median'`
`Names`	Specifies alternative labels for leaf nodes. Choices are: • Vector of structures, each with a `Header` or `Name` field • Cell array of character vectors or string vector The elements must be unique. The number of elements must comply with the number of samples used to generate the pairwise distances in *`Dist`*.

Description

`PhyloTree` `= seqlinkage(`*`Distances`*`)` returns a phylogenetic tree object from the pairwise distances, *`Distances`*, between the species or products. *`Distances`* is a matrix or vector of pairwise distances, such as returned by the `seqpdist` function.

`PhyloTree` `= seqlinkage(`*`Distances`*`, `*`Method`*`)` creates a phylogenetic tree object using a specified patristic distance method. The available methods are:

`'single'`	Nearest distance (single linkage method)
`'complete'`	Furthest distance (complete linkage method)
`'average'` (default)	Unweighted Pair Group Method Average (UPGMA, group average).
`'weighted'`	Weighted Pair Group Method Average (WPGMA)
`'centroid'`	Unweighted Pair Group Method Centroid (UPGMC)

'median'	Weighted Pair Group Method Centroid (WPGMC)

PhyloTree = seqlinkage(*Distances*, *Method*, *Names*) passes a list of unique names to label the leaf nodes (for example, species or products) in a phylogenetic tree object.

Examples

Build Phylogenetic Tree from Pairwise Distances

Create an array of structures representing a multiple alignment of amino acids:

```
seqs = fastaread('pf00002.fa');
```

Measure the Jukes-Cantor pairwise distances between sequences:

```
distances = seqpdist(seqs,'method','jukes-cantor','indels','pair');
```

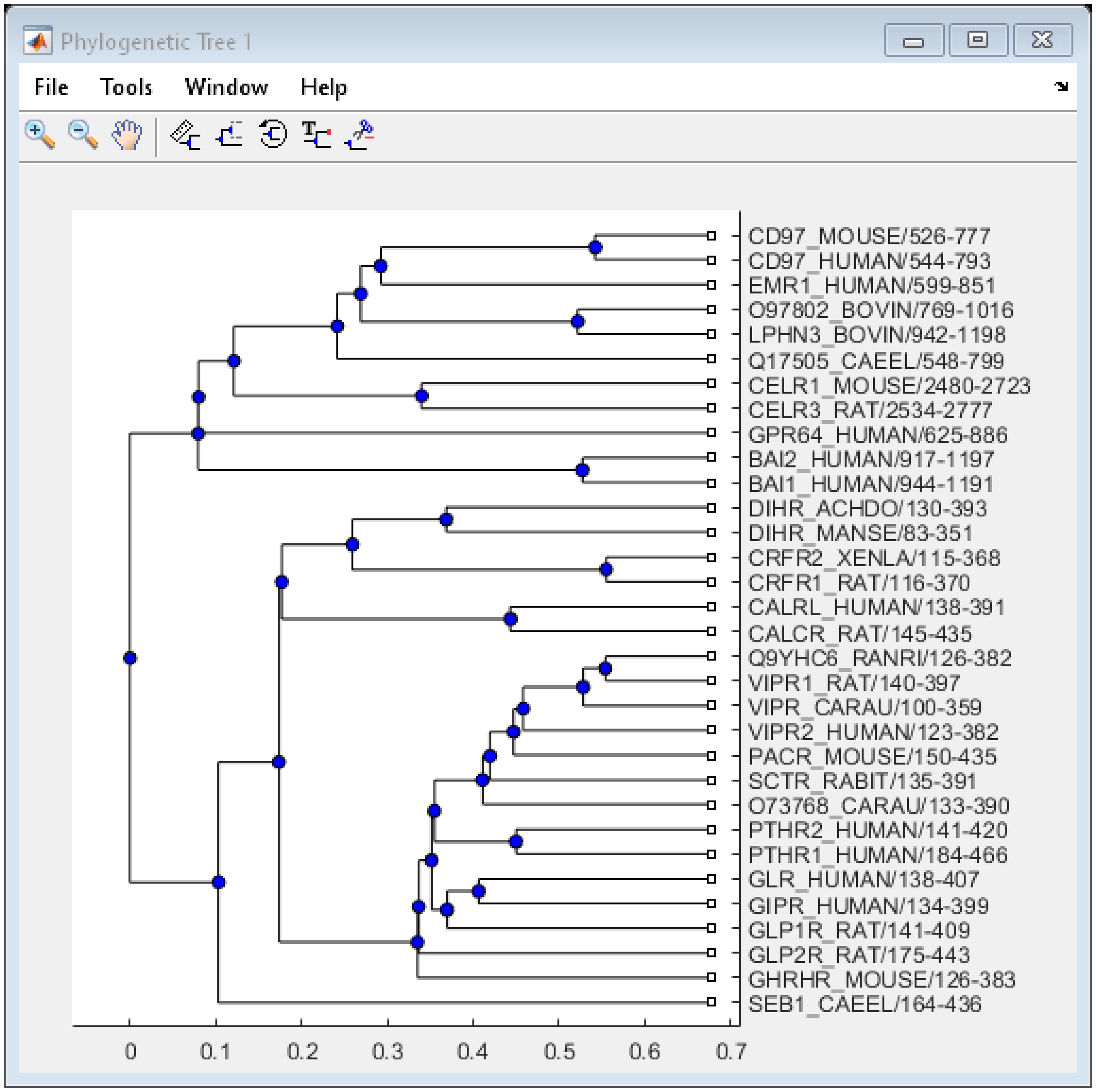
Build the phylogenetic tree for the multiple sequence alignment from calculated pairwise distances. Specify the method to compute the distances of the new nodes to all other nodes. Provide leaf names:

```
phylotree = seqlinkage(distances,'single',seqs)
```

```
    Phylogenetic tree object with 32 leaves (31 branches)
```

View the phylogenetic tree:

```
view(phylotree)
```



Version History

Introduced before R2006a

R2017b: `seqlinkage` correctly computes the input pairwise distances

Behavior changed in R2017b

For the R2017a or earlier versions, `seqlinkage` incorrectly doubled the input pairwise distances when building a tree. This bug has been fixed in R2017b.

If you have been previously selecting on page 1-1688 a subset of the tree returned by `seqlinkage` with a distance threshold, consider dividing the threshold by 2.

Note that the tree topology has always been computed correctly and not affected by this bug.

See Also

phytree | phytreewrite | seqpdist | seqneighjoin | cluster | plot | view

seqlogo

Display sequence logo for nucleotide or amino acid sequences

Syntax

```
seqlogo(Seqs)
seqlogo(Profile)
WgtMatrix = seqlogo(...)
[WgtMatrix, Handle] = seqlogo(...)

seqlogo(..., 'Displaylogo', DisplaylogoValue, ...)
seqlogo(..., 'Alphabet', AlphabetValue, ...)
seqlogo(..., 'Startat', StartatValue, ...)
seqlogo(..., 'Endat', EndatValue, ...)
seqlogo(..., 'SSCorrection', SSCorrectionValue, ...)
```

Input Arguments

Seqs	Set of pairwise or multiply aligned nucleotide or amino acid sequences, represented by any of the following: • Character array • Cell array of character vectors • String vector • Array of structures containing a `Sequence` field
Profile	Sequence profile distribution matrix with the frequency of nucleotides or amino acids for every column in the multiple alignment, such as returned by the `seqprofile` function. The size of the frequency distribution matrix is: • For nucleotides — `[4 x sequence length]` • For amino acids — `[20 x sequence length]` If gaps were included, *Profile* may have `5` rows (for nucleotides) or `21` rows (for amino acids), but `seqlogo` ignores gaps.
DisplaylogoValue	Controls the display of a sequence logo. Choices are `true` (default) or `false`.
AlphabetValue	Character vector or string specifying the type of sequence (nucleotide or amino acid). Choices are `'NT'` (default) or`'AA'`.
StartatValue	Positive integer that specifies the starting position for the sequences in *Seqs*. Default starting position is `1`.
EndatValue	Positive integer that specifies the ending position for the sequences in *Seqs*. Default ending position is the maximum length of the sequences in *Seqs*.

SSCorrectionValue	Controls the use of small sample correction in the estimation of the number of bits. Choices are `true` (default) or `false`.

Output Arguments

WgtMatrix	Cell array containing the symbol list in *Seqs* or *Profile* and the weight matrix used to graphically display the sequence logo.
Handle	Handle to the sequence logo figure.

Description

`seqlogo(`*Seqs*`)` displays a sequence logo for *Seqs*, a set of aligned sequences. The logo graphically displays the sequence conservation at a particular position in the alignment of sequences, measured in bits. The maximum sequence conservation per site is `log2(4)` bits for nucleotide sequences and `log2(20)` bits for amino acid sequences. If the sequence conservation value is zero or negative, no logo is displayed in that position.

`seqlogo(`*Profile*`)` displays a sequence logo for *Profile*, a sequence profile distribution matrix with the frequency of nucleotides or amino acids for every column in the multiple alignment, such as returned by the `seqprofile` function.

Color Code for Nucleotides

Nucleotide	Color
A	Green
C	Blue
G	Yellow
T, U	Red
Other	Purple

Color Code for Amino Acids

Amino Acid	Chemical Property	Color
G S T Y C Q N	Polar	Green
A V L I P W F M	Hydrophobic	Orange
D E	Acidic	Red
K R H	Basic	Blue
Other	—	Tan

WgtMatrix `= seqlogo(...)` returns a cell array of unique symbols in the sequence *Seqs* or *Profile*, and the information weight matrix used to graphically display the logo.

`[`*WgtMatrix*`,` *Handle*`] = seqlogo(...)` returns a handle to the sequence logo figure.

`seqlogo(`*Seqs*`, ...'`*PropertyName*`',` *PropertyValue*`, ...)` calls `seqpdist` with optional properties that use property name/property value pairs. You can specify one or more properties in any order. Each *PropertyName* must be enclosed in single quotation marks and is case insensitive. These property name/property value pairs are as follows:

`seqlogo(..., 'Displaylogo', *DisplaylogoValue*, ...)` controls the display of a sequence logo. Choices are `true` (default) or `false`.

`seqlogo(..., 'Alphabet', *AlphabetValue*, ...)` specifies the type of sequence (nucleotide or amino acid). Choices are `'NT'` (default) or`'AA'`.

Note If you provide amino acid sequences to `seqlogo`, you must set `Alphabet` to `'AA'`.

`seqlogo(..., 'Startat', *StartatValue*, ...)` specifies the starting position for the sequences in *`Seqs`*. Default starting position is `1`.

`seqlogo(..., 'Endat', *EndatValue*, ...)` specifies the ending position for the sequences in *`Seqs`*. Default ending position is the maximum length of the sequences in *`Seqs`*.

`seqlogo(..., 'SSCorrection', *SSCorrectionValue*, ...)` controls the use of small sample correction in the estimation of the number of bits. Choices are `true` (default) or `false`.

Note A simple calculation of bits tends to overestimate the conservation at a particular location. To compensate for this overestimation, when `SSCorrection` is set to `true`, a rough estimate is applied as an approximate correction. This correction works better when the number of sequences is greater than 50.

Examples

Display a Sequence Logo for Aligned Nucleotide Sequences

This example shows how to display a sequence logo for a set of aligned nucleotide sequences.

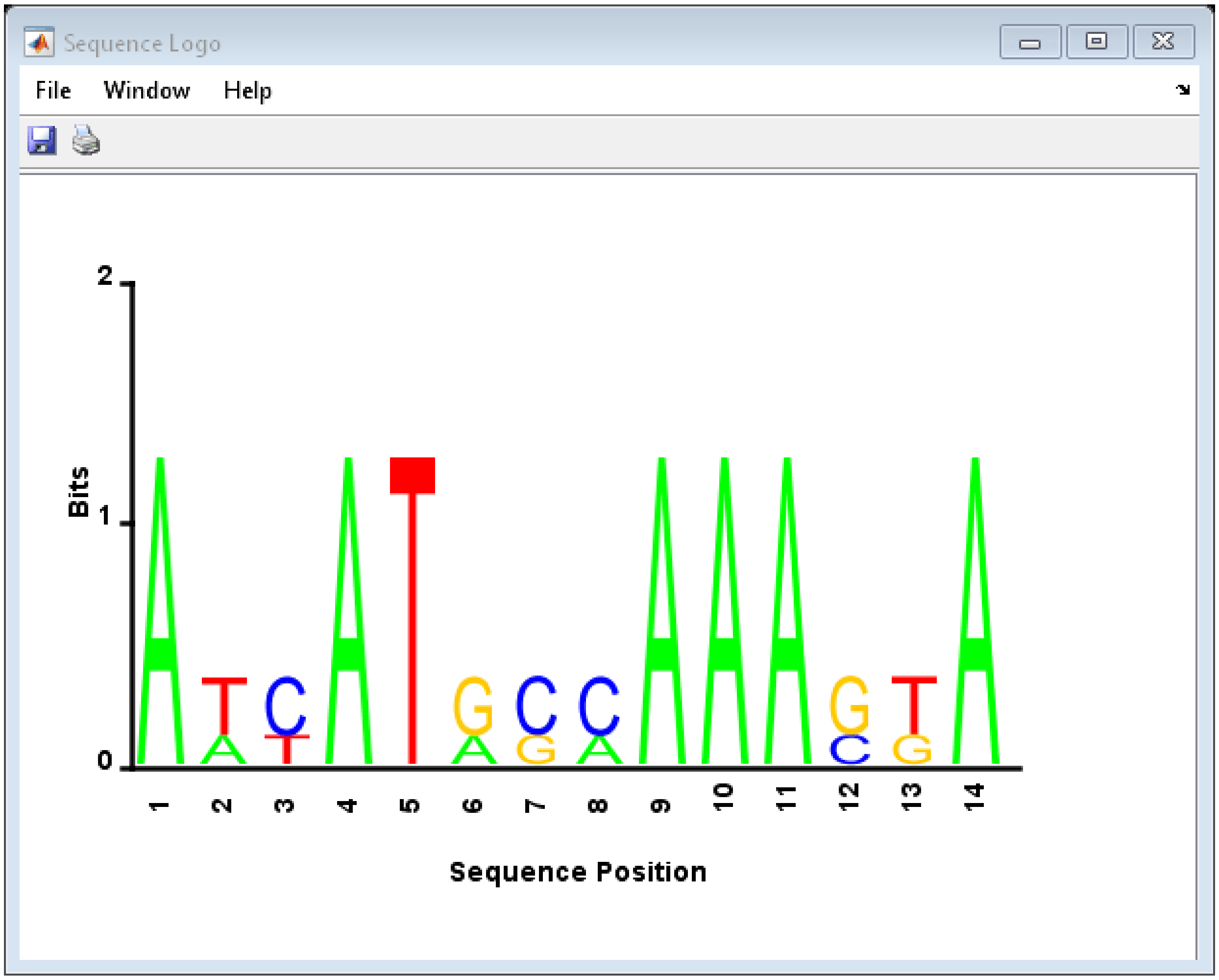
Create a series of aligned nucleotide sequences.

```
S = {'ATTATAGCAAACTA',...
     'AACATGCCAAAGTA',...
     'ATCATGCAAAAGGA'}
```

```
S =

  1x3 cell array

    {'ATTATAGCAAACTA'}    {'AACATGCCAAAGTA'}    {'ATCATGCAAAAGGA'}
```

Display the sequence logo.

```
seqlogo(S)
```



Display a Sequence Logo for Aligned Amino Acid Sequences

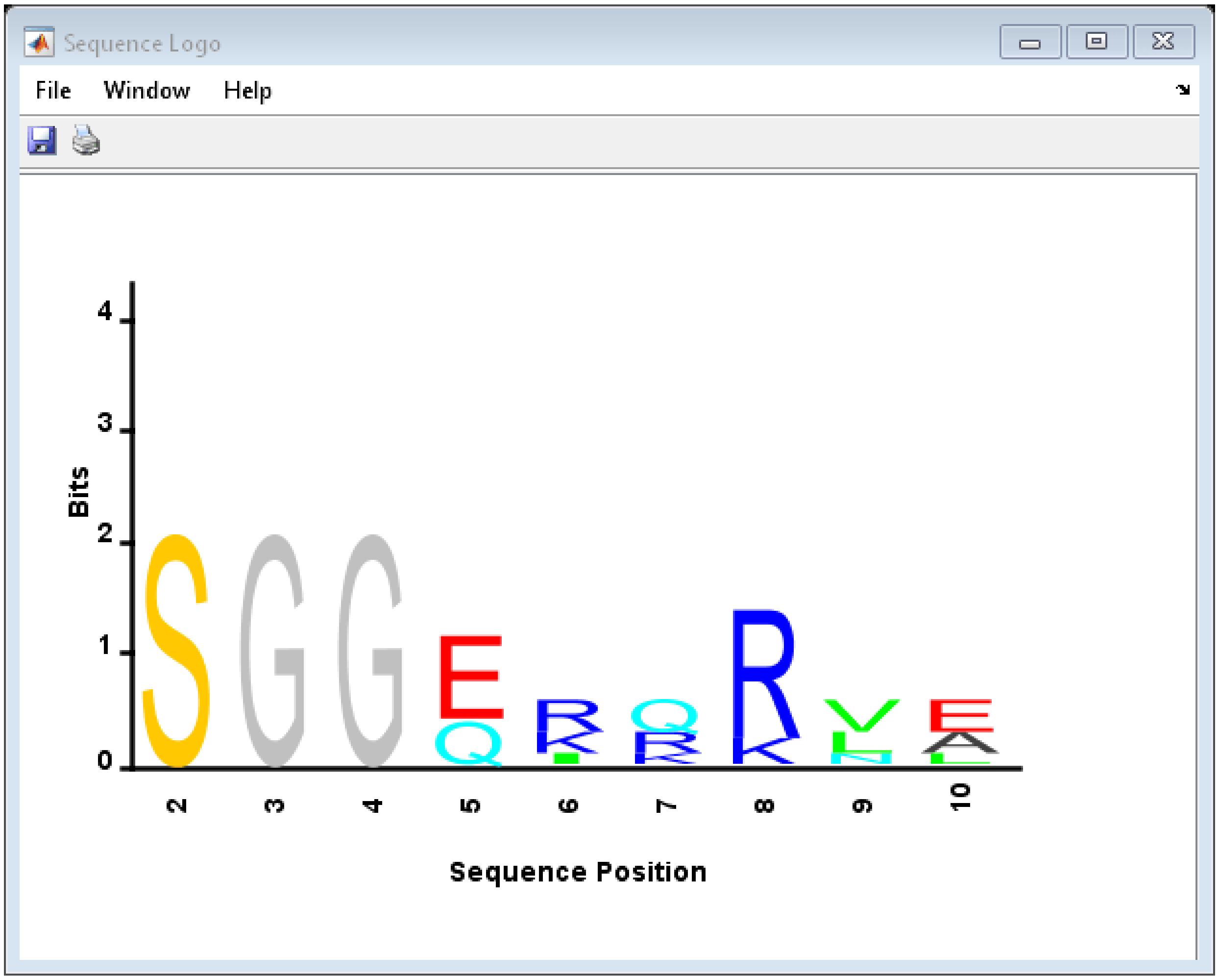
This example shows how to display a sequence logo for a set of aligned amino acid sequences.

Create a series of aligned amino acid sequences.

```
S2 = {'LSGGQRQRVAIARALAL',...
      'LSGGEKQRVAIARALMN',...
      'LSGGQIQRVLLARALAA',...
      'LSGGERRRLEIACVLAL',...
      'FSGGEKKKNELWQMLAL',...
      'LSGGERRRLEIACVLAL'};
```

Display the sequence logo, specifying an amino acid sequence and limiting the logo to sequence positions 2 through 10.

```
seqlogo(S2, 'alphabet', 'aa', 'startAt', 2, 'endAt', 10)
```



Version History

Introduced before R2006a

References

[1] Schneider, T.D., and Stephens, R.M. (1990). Sequence Logos: A new way to display consensus sequences. Nucleic Acids Research *18*, 6097–6100.

See Also

`seqconsensus` | `seqdisp` | `seqprofile`

seqmatch

Find matches for every character vector in library

Syntax

`Index = seqmatch(keyword, Library)`

Description

`Index = seqmatch(keyword, Library)` looks through `Library` to find character vectors or string that begin with `keyword`. `Index` contains the index to the first occurrence for every character vector or string in the query. `keyword` and `Library` must be cell arrays of character vectors or string vectors.

Examples

```
lib = {'VIPS_HUMAN', 'SCCR_RABIT', 'CALR_PIG' ,'VIPR_RAT', 'PACR_MOUSE'};
query = {'CALR','VIP'};
h = seqmatch(query,lib);
lib(h)
```

```
ans = 

    'CALR_PIG'    'VIPS_HUMAN'
```

Version History

Introduced before R2006a

See Also

`regexp` | `strncmp`

seqneighjoin

Construct phylogenetic tree using neighbor-joining method

Syntax

```
PhyloTree = seqneighjoin(Distances)
PhyloTree = seqneighjoin(Distances, Method)
PhyloTree = seqneighjoin(Distances, Method, Names)

PhyloTree = seqneighjoin(..., 'Reroot', RerootValue)
```

Input Arguments

Distances	Matrix or vector containing biological distances between pairs of sequences, such as returned by the `seqpdist` function.
Method	Character vector or string specifying a method to compute the distances between nodes. Choices are `'equivar'` (default) or `'firstorder'`.
Names	Either of the following: • Vector of structures with the fields `Header` and `Name` • Cell array of character vectors or string vector The number of elements must equal the number of samples used to generate the pairwise distances in *`Distances`*.

Description

`PhyloTree` `= seqneighjoin(`*`Distances`*`)` computes *`PhyloTree`*, a phylogenetic tree object, from *`Distances`*, pairwise distances between the species or products, using the neighbor-joining method.

`PhyloTree` `= seqneighjoin(`*`Distances`*`, `*`Method`*`)` specifies *`Method`*, a method to compute the distances of the new nodes to all other nodes at every iteration. The general expression to calculate the distances between the new node, `n`, after joining `i` and `j` and all other nodes (`k`), is given by

```
D(n,k) =  a*D(i,k) + (1-a)*D(j,k) - a*D(n,i) - (1-a)*D(n,j)
```

This expression is guaranteed to find the correct tree with additive data (minimum variance reduction).

Choices for *`Method`* are:

Method	Description
`equivar` (default)	Assumes equal variance and independence of evolutionary distance estimates (a = 1/2), such as in the original neighbor-joining algorithm by Saitou and Nei, JMBE (1987) or as in Studier and Keppler, JMBE (1988).

Method	Description
`firstorder`	Assumes a first-order model of the variances and covariances of evolutionary distance estimates, with `'a'` being adjusted at every iteration to a value between `0` and `1`, such as in Gascuel, JMBE (1997).

`PhyloTree` `= seqneighjoin(`*`Distances`*`,` *`Method`*`,` *`Names`*`)` passes *`Names`*, a list of names (such as species or products), to label the leaf nodes in the phylogenetic tree object.

`PhyloTree` `= seqneighjoin(..., 'Reroot',` *`RerootValue`*`)` specifies whether to reroot *`PhyloTree`*. Choices are `true` (default) or `false`. When *`RerootValue`* is `false`, `seqneighjoin` excludes rerooting the resulting tree, which is useful for observing the original linkage order followed by the algorithm. By default `seqneighjoin` reroots the resulting tree using the midpoint method.

Examples

Build Phylogenetic Tree using Neighbor Joining Method

Create an array of structures representing a multiple alignment of amino acids:

```
seqs = fastaread('pf00002.fa');
```

Measure the Jukes-Cantor pairwise distances between sequences.

```
distances = seqpdist(seqs,'method','jukes-cantor','indels','pair');
```

Use the output argument `distances`, a vector containing biological distances between each pair of sequences, as an input argument to `seqneighjoin`.

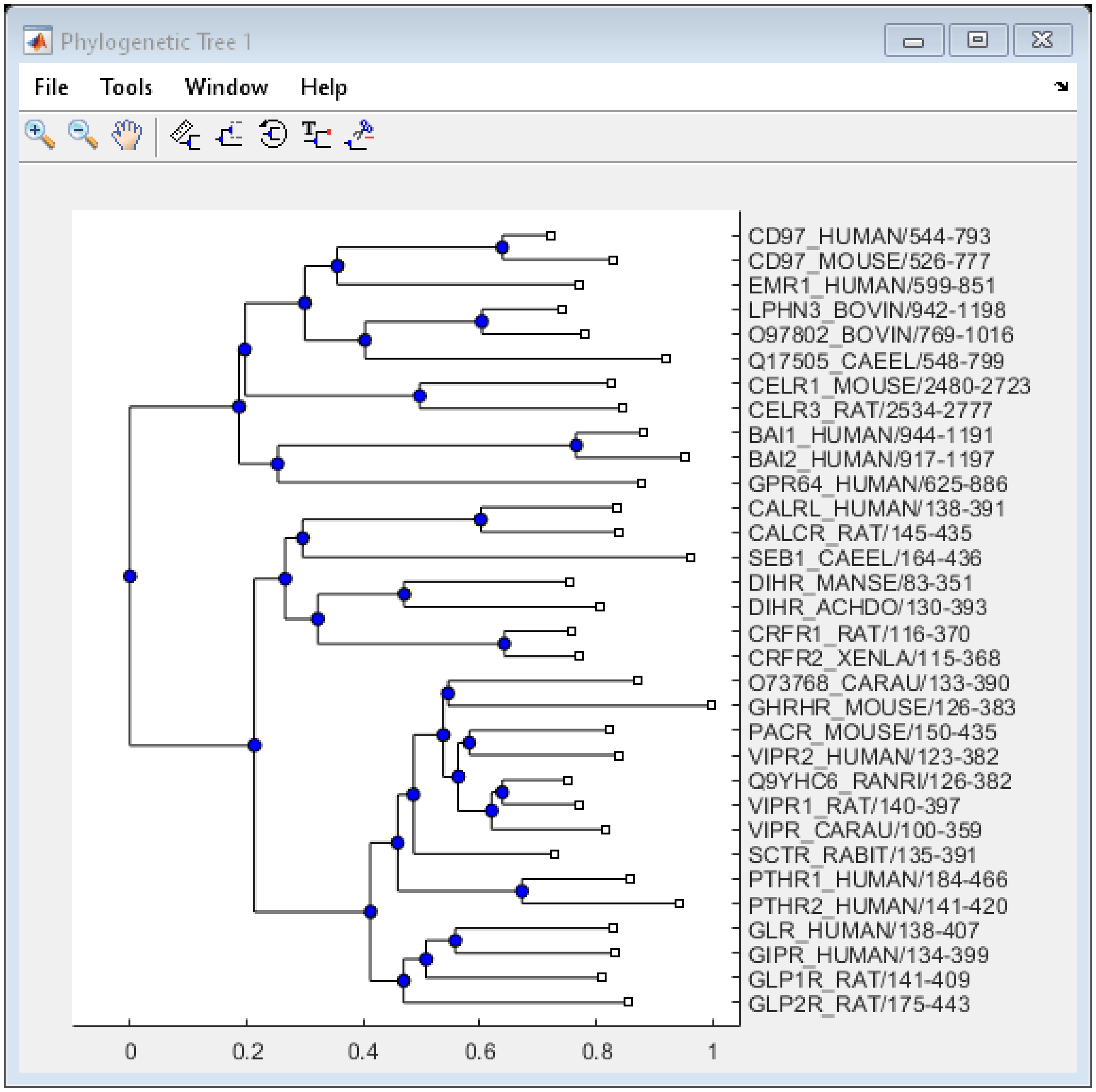
Build the phylogenetic tree for the multiple sequence alignment using the neighbor-joining algorithm. Specify the method to compute the distances of the new nodes to all other nodes.

```
phylotree = seqneighjoin(distances,'equivar',seqs)
```

```
    Phylogenetic tree object with 32 leaves (31 branches)
```

View the phylogenetic tree:

```
view(phylotree)
```



Version History

Introduced before R2006a

References

[1] Saitou, N., and Nei, M. (1987). The neighbor-joining method: A new method for reconstructing phylogenetic trees. Molecular Biology and Evolution *4(4)*, 406–425.

[2] Gascuel, O. (1997). BIONJ: An improved version of the NJ algorithm based on a simple model of sequence data. Molecular Biology and Evolution *14* 685–695.

[3] Studier, J.A., Keppler, K.J. (1988). A note on the neighbor-joining algorithm of Saitou and Nei. Molecular Biology and Evolution *5(6)* 729–731.

See Also
`multialign` | `phytree` | `seqlinkage` | `seqpdist` | `cluster` | `plot` | `reroot` | `view`

seqpdist

Calculate pairwise distance between sequences

Syntax

```
D = seqpdist(Seqs)

D = seqpdist(Seqs, ...'PropertyName', PropertyValue, ...)

D = seqpdist(Seqs, ...'Method', MethodValue, ...)
D = seqpdist(Seqs, ...'Indels', IndelsValue, ...)
D = seqpdist(Seqs, ...'OptArgs', OptArgsValue, ...)
D = seqpdist(Seqs, ...'PairwiseAlignment', PairwiseAlignmentValue, ...)
D = seqpdist(Seqs, ...'UseParallel', UseParallelValue, ...)
D = seqpdist(Seqs, ...'SquareForm', SquareFormValue ...)
D = seqpdist(Seqs, ...'Alphabet', AlphabetValue, ...)
D = seqpdist(Seqs, ...'ScoringMatrix', ScoringMatrixValue, ...)
D = seqpdist(Seqs, ...'Scale', ScaleValue, ...)
D = seqpdist(Seqs, ...'GapOpen', GapOpenValue, ...)
D = seqpdist(Seqs, ...'ExtendGap', ExtendGapValue, ...)
```

Input Arguments

Seqs	Any of the following: • Cell array of character vectors or string vector containing nucleotide or amino acid sequences • Vector of structures containing a `Sequence` field • Matrix of characters, in which each row corresponds to a nucleotide or amino acid sequence
MethodValue	Character vector or string that specifies the method to calculate pairwise distances. Default is `'Jukes-Cantor'`.
IndelsValue	Character vector or string that specifies how to treat sites with gaps. Default is `'score'`.
OptArgsValue	Character vector or cell array that specifies one or more input arguments required or accepted by the distance method specified by the `Method` property.

PairwiseAlignmentValue	Controls the global pairwise alignment of input sequences (using the `nwalign` function), while ignoring the multiple alignment of the input sequences (if any). Choices are `true` or `false`. Default is: • `true` — When all input sequences do not have the same length. • `false` — When all input sequences have the same length. **Tip** If your input sequences are the same length, `seqpdist` assumes they are aligned. If they are not aligned, do one of the following: • Align the sequences before passing them to `seqpdist`, for example, using the `multialign` function. • Set `PairwiseAlignment` to `true` when using `seqpdist`.
UseParallelValue	Controls the calculation of the pairwise distances using `parfor`-loops. When `true`, and Parallel Computing Toolbox is installed and a `parpool` is open, computation occurs in parallel. If there are no open `parpool`, but automatic creation is enabled in the Parallel Preferences, the default pool will be automatically open and computation occurs in parallel. If Parallel Computing Toolbox is installed, but there are no open `parpool` and automatic creation is disabled, then computation uses `parfor`-loops in serial mode. If Parallel Computing Toolbox is not installed, then computation uses `parfor`-loops in serial mode. Default is `false`, which uses for-loops in serial mode.
SquareFormValue	Controls the conversion of the output into a square matrix. Choices are `true` or `false` (default).
AlphabetValue	Character vector or string specifying the type of sequence (nucleotide or amino acid). Choices are `'NT'` or `'AA'` (default).

ScoringMatrixValue	Either of the following: • Character vector or string specifying the scoring matrix to use for the alignment. Choices for amino acid sequences are: • 'BLOSUM62' • 'BLOSUM30' increasing by 5 up to 'BLOSUM90' • 'BLOSUM100' • 'PAM10' increasing by 10 up to 'PAM500' • 'DAYHOFF' • 'GONNET' Default is: • 'BLOSUM50' — When *AlphabetValue* equals 'AA' • 'NUC44' — When *AlphabetValue* equals 'NT' **Note** The above scoring matrices, provided with the software, also include a structure containing a scale factor that converts the units of the output score to bits. You can also use the 'Scale' property to specify an additional scale factor to convert the output score from bits to another unit. • Matrix representing the scoring matrix to use for the alignment, such as returned by the blosum, pam, dayhoff, gonnet, or nuc44 function. **Note** If you use a scoring matrix that you created or was created by one of the above functions, the matrix does not include a scale factor. The output score will be returned in the same units as the scoring matrix. You can use the 'Scale' property to specify a scale factor to convert the output score to another unit. **Note** If you need to compile seqpdist into a stand-alone application or software component using MATLAB Compiler, use a matrix instead of a character vector or string for *ScoringMatrixValue*.
ScaleValue	Positive value that specifies the scale factor used to return the score in arbitrary units. If the scoring matrix information also provides a scale factor, then both are used.
GapOpenValue	Positive integer that specifies the penalty for opening a gap in the alignment. Default is 8.
ExtendedGapValue	Positive integer that specifies the penalty for extending a gap. Default is equal to *GapOpenValue*.

Output Arguments

D	Vector that contains biological distances between each pair of sequences stored in the `M` elements of *`Seqs`*.

Description

`D` `= seqpdist(`*`Seqs`*`)` returns *`D`*, a vector containing biological distances between each pair of sequences stored in the `M` sequences of *`Seqs`*, a cell array of sequences, a vector of structures, or a matrix or sequences.

`D` is a `1`-by-`(M*(M-1)/2)` row vector corresponding to the `M*(M-1)/2` pairs of sequences in *`Seqs`*. The output *`D`* is arranged in the order `((2,1),(3,1),..., (M,1),(3,2),...(M,2),... (M,M-1))`. This is the lower-left triangle of the full `M`-by-`M` distance matrix. To get the distance between the *`I`*th and the *`J`*th sequences for `I > J`, use the formula `D((J-1)*(M-J/2)+I-J)`.

`D` `= seqpdist(`*`Seqs`*`, ...'`*`PropertyName`*`', `*`PropertyValue`*`, ...)` calls `seqpdist` with optional properties that use property name/property value pairs. Specify one or more properties in any order. Enclose each *`PropertyName`* in single quotation marks. Each *`PropertyName`* is case insensitive. These property name/property value pairs are as follows:

`D` `= seqpdist(`*`Seqs`*`, ...'Method', `*`MethodValue`*`, ...)` specifies a method to compute distances between each sequence pair. Choices are shown in the following tables.

Methods for Nucleotides and Amino Acids

Method	Description
`p-distance`	Proportion of sites at which the two sequences are different. `p` is close to `1` for poorly related sequences, and `p` is close to `0` for similar sequences. `d = p`

`Jukes-Cantor` (default)	Maximum likelihood estimate of the number of substitutions between two sequences. `p` is described with the method `p-distance`.
	
	For nucleotides:
	
	`d = -3/4 log(1-p * 4/3)`
	
	For amino acids:
	
	`d = -19/20 log(1-p * 20/19)`

`alignment-score`	Distance (`d`) between two sequences (`1, 2`) is computed from the pairwise alignment score between the two sequences (`score12`), and the pairwise alignment score between each sequence and itself (`score11`, `score22`) as follows: `d = (1-score12/score11)* (1-score12/score22)` This option does not imply that prealigned input sequences will be realigned, it only scores them. Use with care; this distance method does not comply with the ultrametric condition. In the rare case where the score between sequences is greater than the score when aligning a sequence with itself, then `d = 0`

Methods with No Scoring of Gaps (Nucleotides Only)

Method	Description
`Tajima-Nei`	Maximum likelihood estimate considering the background nucleotide frequencies. It can be computed from the input sequences or given by setting `OptArgs` to `[gA gC gG gT]`. `gA`, `gC`, `gG`, `gT` are scalar values for the nucleotide frequencies.
`Kimura`	Considers separately the transitional nucleotide substitution and the transversional nucleotide substitution.
`Tamura`	Considers separately the transitional nucleotide substitution, the transversional nucleotide substitution, and the GC content. GC content can be computed from the input sequences or given by setting `OptArgs` to the proportion of GC content (scalar value from `0` to `1`).
`Hasegawa`	Considers separately the transitional nucleotide substitution, the transversional nucleotide substitution, and the background nucleotide frequencies. Background frequencies can be computed from the input sequences or given by setting the `OptArgs` property to `[gA gC gG gT]`.
`Nei-Tamura`	Considers separately the transitional nucleotide substitution between purines, the transitional nucleotide substitution between pyrimidines, the transversional nucleotide substitution, and the background nucleotide frequencies. Background frequencies can be computed from the input sequences or given by setting the `OptArgs` property to `[gA gC gG gT]`.

Methods with No Scoring of Gaps (Amino Acids Only)

Method	Description
`Poisson`	Assumes that the number of amino acid substitutions at each site has a Poisson distribution.
`Gamma`	Assumes that the number of amino acid substitutions at each site has a Gamma distribution with parameter `a`. Set `a` using the `OptArgs` property. Default is `2`.

You can also specify a user-defined distance function using `@`, for example, `@distfun`. The distance function must have the form:

```
function D = distfun(S1, S2, OptArgsValue)
```

The `distfun` function takes the following arguments:

- *`S1`* , *`S2`* — Two sequences of the same length (nucleotide or amino acid).
- *`OptArgsValue`* — Optional problem-dependent arguments.

The `distfun` function returns a scalar that represents the distance between *`S1`* and *`S2`*.

`D` `= seqpdist(`*`Seqs`*`, ...'Indels', `*`IndelsValue`*`, ...)` specifies how to treat sites with gaps. Choices are:

- `score` (default) — Scores these sites either as a point mutation or with the alignment parameters, depending on the method selected.
- `pairwise-del` — For every pairwise comparison, it ignores the sites with gaps.
- `complete-del` — Ignores all the columns in the multiple alignment that contain a gap. This option is available only if you provided a multiple alignment as the input *`Seqs`*.

`D` `= seqpdist(`*`Seqs`*`, ...'OptArgs', `*`OptArgsValue`*`, ...)` passes one or more arguments required or accepted by the distance method specified by the `Method` property. Use a character vector or cell array to pass one or more input arguments. For example, provide the nucleotide frequencies for the `Tajima-Nei` distance method, instead of computing them from the input sequences.

`D` `= seqpdist(`*`Seqs`*`, ...'PairwiseAlignment', `*`PairwiseAlignmentValue`*`, ...)` controls the global pairwise alignment of input sequences (using the `nwalign` function), while ignoring the multiple alignment of the input sequences (if any). Default is:

- `true` — When all input sequences do not have the same length.
- `false` — When all input sequences have the same length.

Tip If your input sequences have the same length, `seqpdist` assumes they are aligned. If they are not aligned, do one of the following:

- Align the sequences before passing them to `seqpdist`, for example, using the `multialign` function.
- Set `PairwiseAlignment` to `true` when using `seqpdist`.

`D` `= seqpdist(`*`Seqs`*`, ...'UseParallel', `*`UseParallelValue`*`, ...)` specifies whether to use `parfor`-loops when calculating the pairwise distances. When `true`, and Parallel Computing

Toolbox is installed and a `parpool` is open, computation occurs in parallel. If there are no open `parpool`, but automatic creation is enabled in the Parallel Preferences, the default pool will be automatically open and computation occurs in parallel. If Parallel Computing Toolbox is installed, but there are no open `parpool` and automatic creation is disabled, then computation uses `parfor`-loops in serial mode. If Parallel Computing Toolbox is not installed, then computation uses `parfor`-loops in serial mode. Default is `false`, which uses for-loops in serial mode.

`*D* = seqpdist(*Seqs*, ...'SquareForm', *SquareFormValue* ...)` controls the conversion of the output into a square matrix such that `*D*(*I*,*J*)` denotes the distance between the *I*th and *J*th sequences. The square matrix is symmetric and has a zero diagonal. Choices are `true` or `false` (default). Setting `Squareform` to `true` is the same as using the `squareform` function in Statistics and Machine Learning Toolbox .

`*D* = seqpdist(*Seqs*, ...'Alphabet', *AlphabetValue*, ...)` specifies the type of sequence (nucleotide or amino acid). Choices are `'NT'` or `'AA'` (default).

The remaining input properties are available when the `Method` property equals `'alignment-score'` or the `PairwiseAlignment` property equals `true`.

`*D* = seqpdist(*Seqs*, ...'ScoringMatrix', *ScoringMatrixValue*, ...)` specifies the scoring matrix to use for the global pairwise alignment. Default is:

- `'NUC44'` — When *`AlphabetValue`* equals `'NT'`.
- `'BLOSUM50'` — When *`AlphabetValue`* equals `'AA'`.

`*D* = seqpdist(*Seqs*, ...'Scale', *ScaleValue*, ...)` specifies the scale factor used to return the score in arbitrary units. Choices are any positive value. If the scoring matrix information also provides a scale factor, then both are used.

`*D* = seqpdist(*Seqs*, ...'GapOpen', *GapOpenValue*, ...)` specifies the penalty for opening a gap in the alignment. Choices are any positive integer. Default is `8`.

`*D* = seqpdist(*Seqs*, ...'ExtendGap', *ExtendGapValue*, ...)` specifies the penalty for extending a gap in the alignment. Choices are any positive integer. Default is equal to *`GapOpenValue`*.

Examples

1. Read amino acid alignment data into a MATLAB structure.

```
seqs = fastaread('pf00002.fa');
```

2. For every possible pair of sequences in the multiple alignment, ignore sites with gaps and score with the scoring matrix `PAM250`.

```
dist = seqpdist(seqs,'Method','alignment-score',...
                'Indels','pairwise-delete',...
                'ScoringMatrix','pam250');
```

3. Force the realignment of each sequence pair ignoring the provided multiple alignment.

```
dist = seqpdist(seqs,'Method','alignment-score',...
                'Indels','pairwise-delete',...
                'ScoringMatrix','pam250',...
                'PairwiseAlignment',true);
```

4 Measure the Jukes-Cantor pairwise distances after realigning each sequence pair, counting the gaps as point mutations.

```
dist = seqpdist(seqs,'Method','jukes-cantor',...
                'Indels','score',...
                'Scoringmatrix','pam250',...
                'PairwiseAlignment',true);
```

Version History

Introduced before R2006a

Extended Capabilities

Automatic Parallel Support

Accelerate code by automatically running computation in parallel using Parallel Computing Toolbox™.

To run in parallel, set `'UseParallel'` to `true`.

For more information, see the `'UseParallel'` name-value pair argument.

See Also

`fastaread` | `dnds` | `dndsml` | `multialign` | `nwalign` | `phytree` | `seqlinkage` | `pdist`

Topics

`phytree object`

seqprofile

Calculate sequence profile from set of multiply aligned sequences

Syntax

```
Profile = seqprofile(Seqs)
[Profile, Symbols] = seqprofile(Seqs)

seqprofile(Seqs, ...'Alphabet', AlphabetValue, ...)
seqprofile(Seqs, ...'Counts', CountsValue, ...)
seqprofile(Seqs, ...'Gaps', GapsValue, ...)
seqprofile(Seqs, ...'Ambiguous', AmbiguousValue, ...)
seqprofile(Seqs, ...'Limits', LimitsValue, ...)
```

Arguments

Seqs	Set of multiply aligned sequences represented by any of the following:. • Character array • Cell array of character vectors • String vector • Array of structures containing the field `Sequence`
AlphabetValue	Character vector or string specifying the sequence alphabet. Choices are: • `'NT'` — Nucleotides • `'AA'` — Amino acids (default) • `'none'` — No alphabet When `Alphabet` is `'none'`, the symbol list is based on the observed symbols. Each character can be any symbol, except for a hyphen (-) and a period (.), which are reserved for gaps.
CountsValue	Controls returning frequency (ratio of counts/total counts) or counts. Choices are `true` (counts) or `false` (frequency). Default is `false`.
GapsValue	Character vector or string that controls the counting of gaps in a sequence. Choices are: • `'all'` — Counts all gaps • `'noflanks'` — Counts all gaps except those at the flanks of every sequence • `'none'` — Default. Counts no gaps.
AmbiguousValue	Controls counting ambiguous symbols. Enter `'Count'` to add partial counts to the standard symbols.
LimitsValue	Specifies whether to use part of the sequence. Enter a `[1x2]` vector with the first position and the last position to include in the profile. Default is `[1,SeqLength]`.

Description

`Profile = seqprofile(Seqs)` returns `Profile`, a matrix of size `[20 (or 4) x SequenceLength]` with the frequency of amino acids (or nucleotides) for every column in the multiple alignment. The order of the rows is given by

- 4 nucleotides — `A C G T/U`
- 20 amino acids — `A R N D C Q E G H I L K M F P S T W Y V`

`[Profile, Symbols] = seqprofile(Seqs)` returns `Symbols`, a unique symbol list where every symbol in the list corresponds to a row in `Profile`, the profile.

`seqprofile(Seqs, ...'PropertyName', PropertyValue, ...)` calls `seqprofile` with optional properties that use property name/property value pairs. You can specify one or more properties in any order. Each `PropertyName` must be enclosed in single quotation marks and is case insensitive. These property name/property value pairs are as follows:

`seqprofile(Seqs, ...'Alphabet', AlphabetValue, ...)` selects a nucleotide alphabet, amino acid alphabet, or no alphabet.

`seqprofile(Seqs, ...'Counts', CountsValue, ...)` when `Counts` is `true`, returns the counts instead of the frequency.

`seqprofile(Seqs, ...'Gaps', GapsValue, ...)` appends a row to the bottom of a profile (`Profile`) with the count for gaps.

`seqprofile(Seqs, ...'Ambiguous', AmbiguousValue, ...)` when `Ambiguous` is `'count'`, counts the ambiguous amino acid symbols (`B Z X`) and nucleotide symbols (`R Y K M S W B D H V N`) with the standard symbols. For example, the amino acid `X` adds a `1/20` count to every row while the amino acid `B` counts as `1/2` at the `D` and `N` rows.

`seqprofile(Seqs, ...'Limits', LimitsValue, ...)` specifies the start and end positions for the profile relative to the indices of the multiple alignment.

Examples

Calculate Sequence Profile

Create an array of structures representing a multiple alignment of amino acids:

```
seqs = fastaread('pf00002.fa');
```

Return the sequence profile and symbol list from position 50 through 55 of the set of multiply aligned sequences, counting all gaps.

```
[Profile2,Symbols2] = seqprofile(seqs,'limits',[50 55],'gaps','all')
```

```
Profile2 = 21×6

    0.0312    0.0312    0.1562    0.4375    0.1250    0.2188
         0         0    0.3750         0         0         0
         0         0    0.0938    0.1562         0         0
         0         0         0    0.0312         0         0
         0    0.0625         0         0    0.0312         0
         0         0         0    0.0312         0         0
         0         0         0    0.1250         0         0
    0.0312         0    0.0625         0         0         0
         0         0         0         0         0         0
    0.4688    0.0625         0         0    0.3125    0.1562
      ⋮

Symbols2 =
'ARNDCQEGHILKMFPSTWYV-'
```

Version History

Introduced before R2006a

See Also

`fastaread` | `multialignread` | `multialignwrite` | `seqconsensus` | `seqdisp` | `seqlogo`

seqqcplot

Create quality control plots for sequence and quality data

Syntax

```
seqqcplot(dataSource)
seqqcplot(dataSource,type)
seqqcplot(dataSource,type,encoding)
seqqcplot( ___ ,Name,Value)
H = seqqcplot( ___ )
```

Description

`seqqcplot(dataSource)` generates a figure with quality control (QC) plots of sequence and quality data from `dataSource`. The figure contains the following types of QC plots.

- Box plot for the average quality score at each sequence position
- Bar plot for the sequence base composition at each sequence position
- Histogram of the average sequence quality score distribution
- Histogram of the GC-content distribution
- Histogram of the sequence length distribution

In the figure, you can click a specific plot to open it in a separate window.

`seqqcplot(dataSource,type)` generates a QC plot specified by `type`.

`seqqcplot(dataSource,type,encoding)` also specifies the encoding format of the base quality in the input file.

`seqqcplot( ___ ,Name,Value)` uses any of the input arguments in the previous syntaxes and additional options specified by one or more `Name,Value` pair arguments.

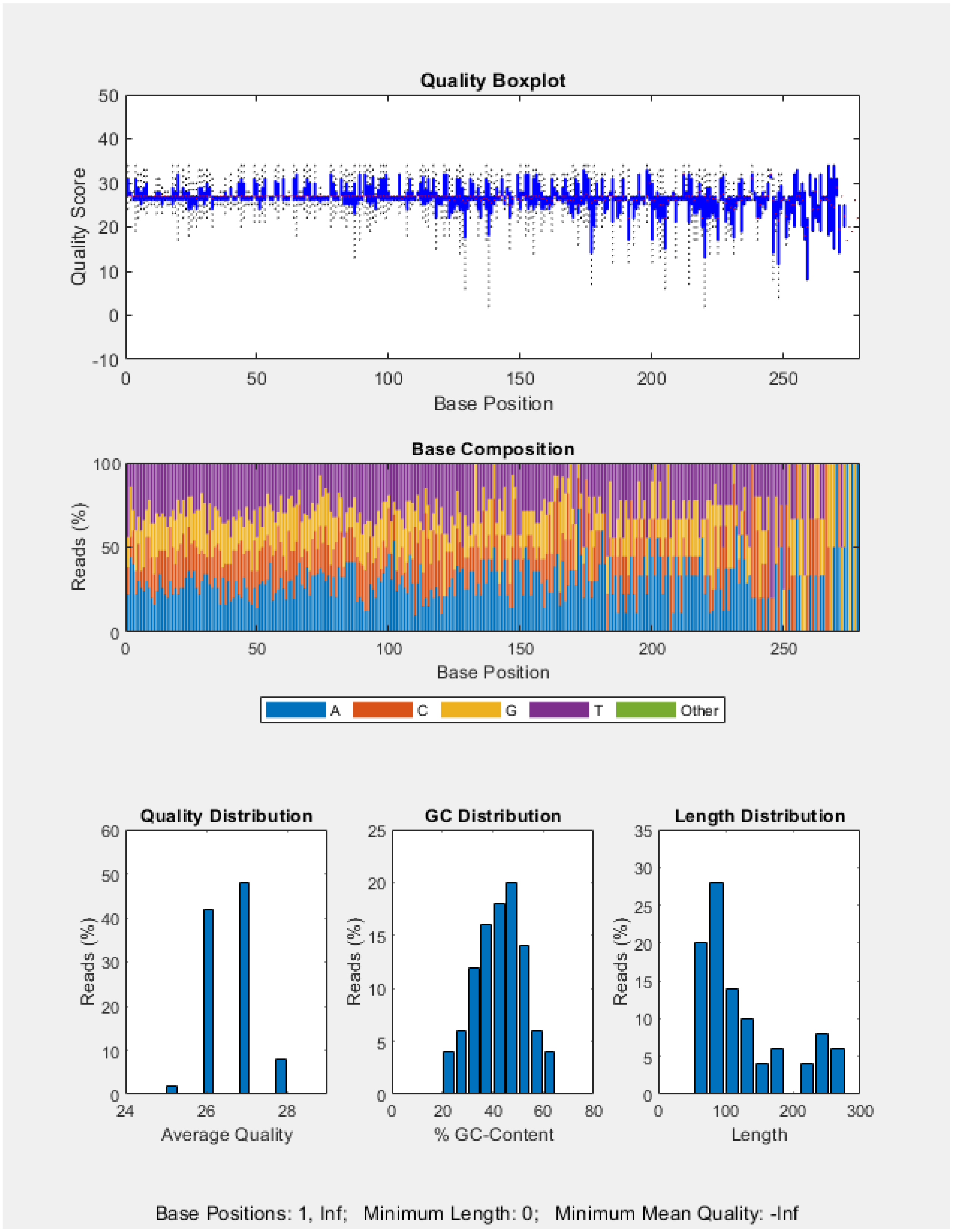
`H = seqqcplot( ___ )` returns the figure handle `H` of the output figure.

Examples

Create Quality Control Plots for Sequence and Quality Data

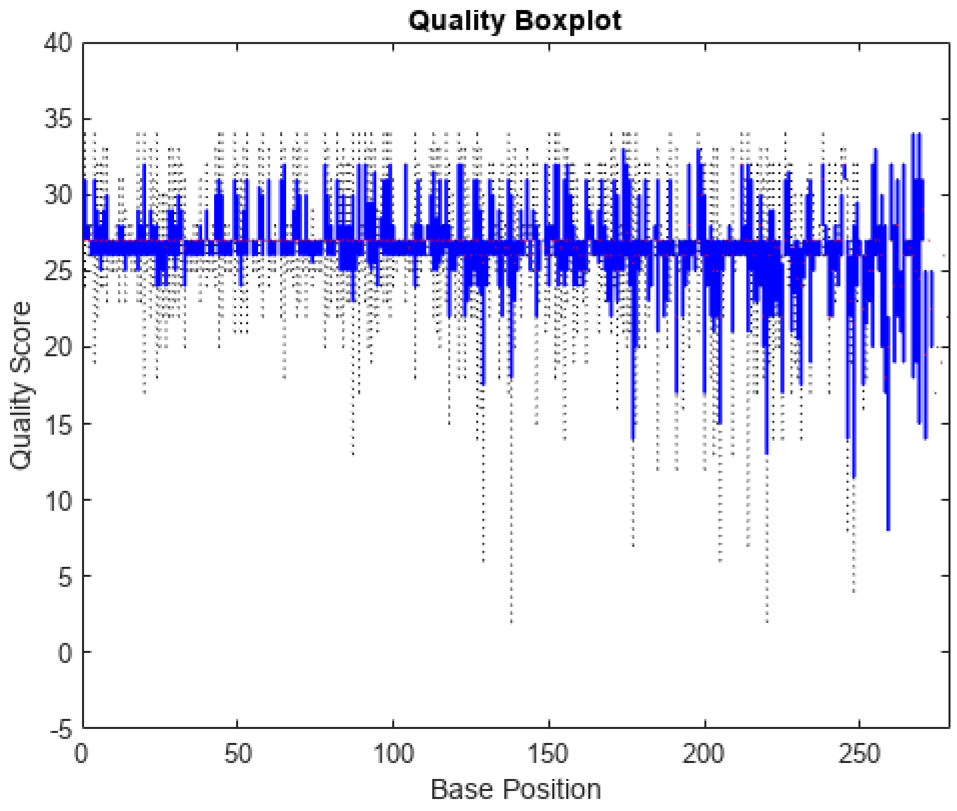
Plot quality control plots for sequence statistics and quality data from a FASTQ file.

```
seqqcplot('SRR005164_1_50.fastq');
```


Quality Boxplot
Quality Score
Base Position
Base Composition
Reads (%)
Base Position
A
C
G
T
Other
Quality Distribution
Reads (%)
Average Quality
GC Distribution
Reads (%)
% GC-Content
Length Distribution
Reads (%)
Length
Base Positions: 1, Inf; Minimum Length: 0; Minimum Mean Quality: -Inf

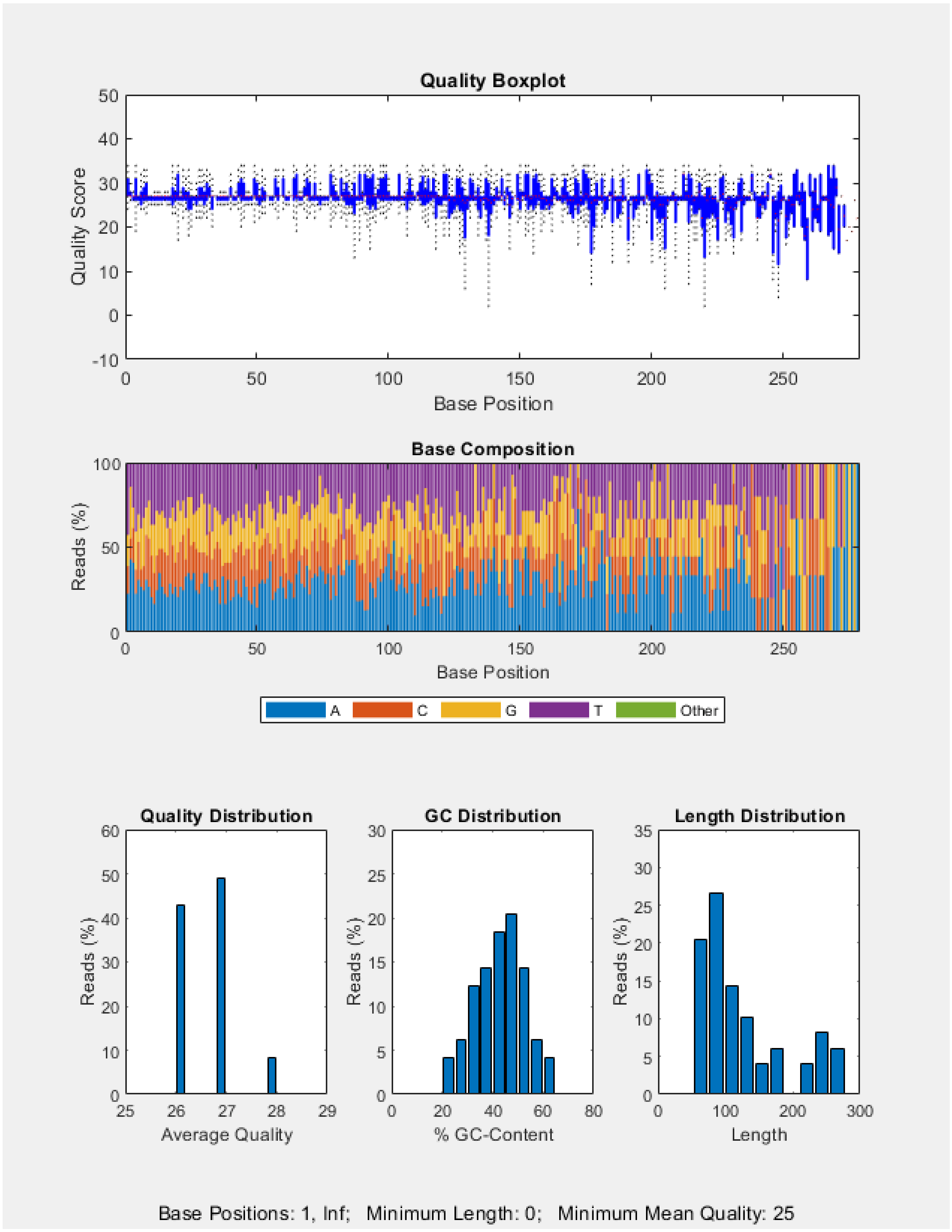
Plot only the box plot of average quality score for each sequence position.

```
seqqcplot('SRR005164_1_50.fastq','QualityBoxplot');
```



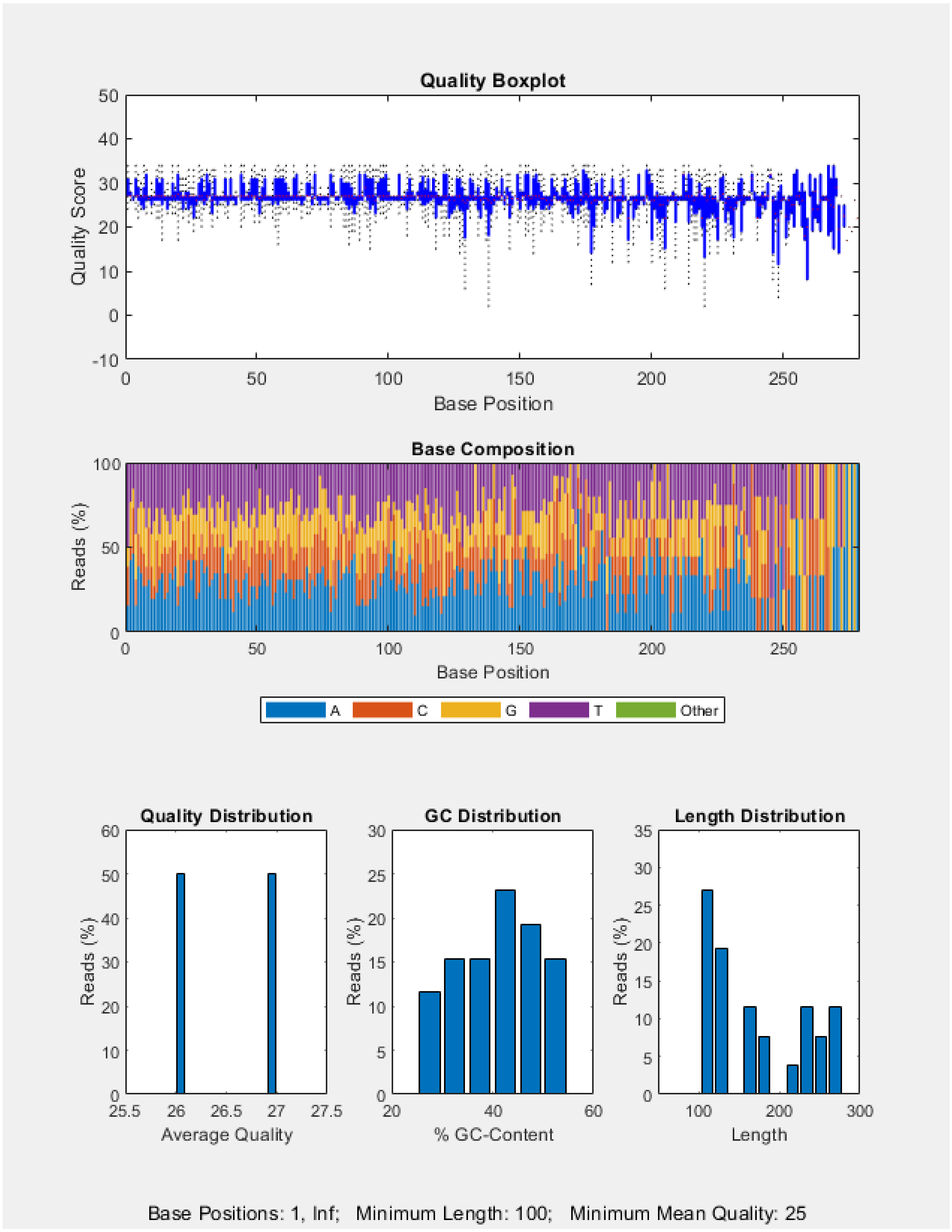
Plot the quality data of sequences with a minimum mean quality of 25.

```
seqqcplot('SRR005164_1_50.fastq','MeanQuality',25);
```


Quality Boxplot
Quality Score
Base Position
Base Composition
Reads (%)
Base Position
A
C
G
T
Other
Quality Distribution
Reads (%)
Average Quality
GC Distribution
Reads (%)
% GC-Content
Length Distribution
Reads (%)
Length
Base Positions: 1, Inf; Minimum Length: 0; Minimum Mean Quality: 25

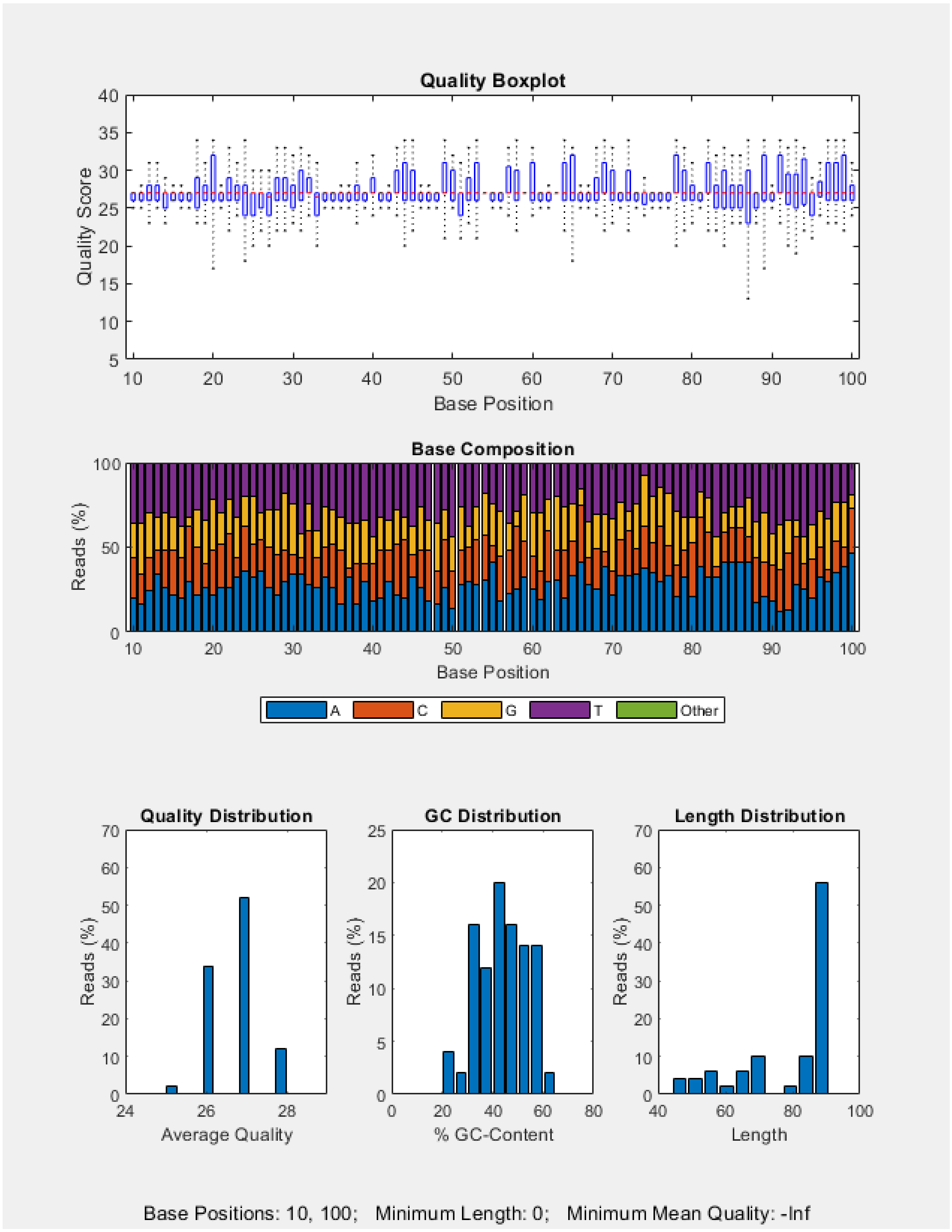
Plot the data of sequences having a minimum mean quality of 25 and a minimum sequence length of 100.

```
seqqcplot('SRR005164_1_50.fastq','MeanQuality',25,'MinLength',100);
```


Quality Boxplot
Quality Score
Base Position
Base Composition
Reads (%)
Base Position
A
C
G
T
Other
Quality Distribution
Reads (%)
Average Quality
GC Distribution
Reads (%)
% GC-Content
Length Distribution
Reads (%)
Length
Base Positions: 1, Inf;   Minimum Length: 100;   Minimum Mean Quality: 25

Produce QC plots for the quality data corresponding to the subsequences from base position 10 to 100.

```
seqqcplot('SRR005164_1_50.fastq','BasePositions',[10 100]);
```


Quality Boxplot
Quality Score
Base Position
Base Composition
Reads (%)
Base Position
A
C
G
T
Other
Quality Distribution
Reads (%)
Average Quality
GC Distribution
Reads (%)
% GC-Content
Length Distribution
Reads (%)
Length
Base Positions: 10, 100; Minimum Length: 0; Minimum Mean Quality: -Inf

Input Arguments

`dataSource` — Sequence and quality information
`BioMap` object | `BioRead` object | character vector | string | string vector | cell array of character vector

Sequence and quality information, specified as a `BioMap` object, `BioRead` object, character vector, string, string vector, or cell array of character vectors representing the names of FASTQ, SAM, or BAM files.

`seqqcplot` uses the read quality data, instead of the alignment quality, if you specify SAM or BAM files, a `BioRead` or `BioMap` object.

Example: `'SRR005164_1_50.fastq'`

`type` — Name of QC plot to generate
`'Summary'` (default) | `'QualityBoxplot'` | `'CompositionLine'` | `'CompositionBar'` | `'QualityDistribution'` | `'GCDistribution'` | `'LengthDistribution'`

Name of the QC plot to generate, specified as one of the following:

Name of QC Plot	Description
`'QualityBoxplot'`	Box plot for the average quality score at each sequence position.
`'CompositionLine'`	Line plot for the sequence base composition at each sequence position.
`'CompositionBar'`	Bar plot for the sequence base composition at each sequence position.
`'QualityDistribution'`	Histogram of the average sequence quality score distribution.
`'GCDistribution'`	Histogram of the GC-content distribution.
`'LengthDistribution'`	Histogram of the sequence length distribution.
`'Summary'`	Summary figure containing all available QC plots, except the `'CompositionLine'` plot. The figure also shows the values of name-value pairs that were used to generate the plots. If name-value pairs were not specified, it shows the corresponding default values instead.

By default, all available QC plots are plotted as subplots in a figure. To open a specific subplot in a separate figure window, click the subplot.

Example: `'QualityBoxplot'`

`encoding` — Encoding format of base quality
`'Illumina18'` (default) | `'Sanger'` | `'Solexa'` | `'Illumina13'` | `'Illumina15'` | `'Illumina19'`

Encoding format of the base quality, specified as one of the following:

- `'Sanger'`
- `'Solexa'`
- `'Illumina13'`

- `'Illumina15'`
- `'Illumina18'`
- `'Illumina19'`

Example: `'Sanger'`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `'MeanQuality',5`

`MeanQuality` — Minimum threshold on average base quality across each sequence
`-Inf` (default) | numeric scalar

Minimum threshold on the average base quality across each sequence, specified as a numeric scalar. The function considers only sequences with average quality score equal to or greater than the threshold. The threshold value is interpreted according to the specified encoding format. Default is `-Inf`, that is, any sequence is considered.

Example: `'MeanQuality',5`

`MinLength` — Minimum threshold on sequence length
`0` (default) | nonnegative numeric scalar

Minimum threshold on the sequence length, specified as a nonnegative numeric scalar. The function considers only sequences with length equal to or greater than the threshold.

Example: `'MinLength',100`

`BasePositions` — Base position range for subsequences
`[1 Inf]` (default) | two-element vector

Base position range for subsequences, specified as a two-element vector. The function considers only the subsequences in the specified position range. Default is `[1 Inf]`, that is, the entire length of each sequence is considered.

Example: `'BasePositions',[5 50]`

Output Arguments

`H` — Handle to output figure
figure handle

Handle to the output figure, returned as a figure handle.

Version History

Introduced in R2017a

See Also

`seqtrim` | `seqfilter` | `seqsplit` | `seqsplitpe`

seqrcomplement

Calculate reverse complementary strand of nucleotide sequence

Syntax

SeqRC = seqrcomplement(*SeqNT*)

Arguments

SeqNT	Nucleotide sequence specified by any of the following: • Character vector or string with the letters A, C, G, T, U, and ambiguous characters R, Y, K, M, S, W, B, D, H, V, N. • Row vector of integers from the table Mapping Nucleotide Integers to Letter Codes. • MATLAB structure containing a Sequence field that contains a nucleotide sequence, such as returned by emblread, fastaread, fastqread, genbankread, getembl, or getgenbank.

Description

SeqRC = seqrcomplement(*SeqNT*) calculates the reverse complementary strand of a DNA or RNA nucleotide sequence. The return sequence, *SeqRC*, reads from 3' --> 5' and is in the same format as *SeqNT*. For example, if *SeqNT* is a vector of integers, then so is *SeqRC*.

Nucleotide in *SeqNT*	Converts to This Nucleotide in *SeqRC*
A	T or U
C	G
G	C
T or U	A

Examples

Return the reverse complement of a DNA nucleotide sequence.

```
s = 'ATCG'
seqrcomplement(s)

ans =
CGAT
```

Version History

Introduced before R2006a

See Also

`codoncount` | `palindromes` | `seqcomplement` | `seqreverse` | `seqviewer`

seqreverse

Calculate reverse strand of nucleotide sequence

Syntax

`SeqR = seqreverse(SeqNT)`

Arguments

SeqNT	Nucleotide sequence specified by any of the following: • Character vector or string with the letters `A`, `C`, `G`, `T`, `U`, and ambiguous characters `R`, `Y`, `K`, `M`, `S`, `W`, `B`, `D`, `H`, `V`, `N`. • Row vector of integers from the table Mapping Nucleotide Integers to Letter Codes. • MATLAB structure containing a `Sequence` field that contains a nucleotide sequence, such as returned by `emblread`, `fastaread`, `fastqread`, `genbankread`, `getembl`, or `getgenbank`.

Description

SeqR `= seqreverse(`*SeqNT*`)` calculates the reverse strand of a DNA or RNA nucleotide sequence. The return sequence, *SeqR*, reads from 3' --> 5' and is in the same format as *SeqNT*. For example, if *SeqNT* is a vector of integers, then so is *SeqR*.

Examples

Return the reverse strand of a DNA nucleotide sequence.

```
s = 'ATCG'
seqreverse(s)
ans =
GCTA
```

Version History

Introduced before R2006a

See Also

`codoncount` | `palindromes` | `seqcomplement` | `seqrcomplement` | `seqviewer` | `fliplr`

seqshoworfs

Display open reading frames in sequence

Syntax

```
seqshoworfs(SeqNT)

seqshoworfs(SeqNT, ...'Frames', FramesValue, ...)
seqshoworfs(SeqNT, ...'GeneticCode', GeneticCodeValue, ...)
seqshoworfs(SeqNT, ...'MinimumLength', MinimumLengthValue, ...)
seqshoworfs(SeqNT, ...'AlternativeStartCodons', AlternativeStartCodonsValue,
...)
seqshoworfs(SeqNT, ...'Color', ColorValue, ...)
seqshoworfs(SeqNT, ...'Columns', ColumnsValue, ...)
```

Arguments

SeqNT	Nucleotide sequence. Enter either a character vector or string with `A`, `T` (`U`), `G`, `C`, and ambiguous characters `R`, `Y`, `K`, `M`, `S`, `W`, `B`, `D`, `H`, `V`, `N`, or a vector of integers. You can also enter a structure with the field `Sequence`.
FramesValue	Property to select the frame. Enter `1`, `2`, `3`, `-1`, `-2`, `-3`, enter a vector with integers, or `'all'`. The default value is the vector `[1 2 3]`. Frames `-1`, `-2`, and `-3` correspond to the first, second, and third reading frames for the reverse complement.
GeneticCodeValue	Genetic code name. Enter a code number or a code name from the table Genetic Code.
MinimumLengthValue	Property to set the minimum number of codons in an ORF.
AlternativeStartCodonsValue	Property to control using alternative start codons. Enter either `true` or `false`. The default value is `false`.

ColorValue	Color to highlight the reading frame. Specify one of the following: • Three-element numeric vector of RGB values • Character vector or string containing a predefined single-letter color code • Character vector or string containing a predefined color name For example, to use cyan, enter `[0 1 1]`, `'c'`, or `'cyan'`. For more information on specifying colors, see “Color Options” on page 1-1755. To specify different colors for the three reading frames, use a 1-by-3 cell array of color values. If you are displaying reverse complement reading frames, then use a 1-by-6 cell array of color values. The default color scheme is blue for the first reading frame, red for the second, and green for the third.
ColumnsValue	Property to specify the number of columns in the output.

Genetic Code

Code Number	Code Name
1	`Standard`
2	`Vertebrate Mitochondrial`
3	`Yeast Mitochondrial`
4	`Mold`, `Protozoan`, `Coelenterate Mitochondrial`, and `Mycoplasma/ Spiroplasma`
5	`Invertebrate Mitochondrial`
6	`Ciliate`, `Dasycladacean`, and `Hexamita Nuclear`
9	`Echinoderm Mitochondrial`
10	`Euplotid Nuclear`
11	`Bacterial` and `Plant Plastid`
12	`Alternative Yeast Nuclear`
13	`Ascidian Mitochondrial`
14	`Flatworm Mitochondrial`
15	`Blepharisma Nuclear`
16	`Chlorophycean Mitochondrial`
21	`Trematode Mitochondrial`
22	`Scenedesmus Obliquus Mitochondrial`
23	`Thraustochytrium Mitochondrial`

Description

`seqshoworfs` identifies and highlights all open reading frames using the standard or an alternative genetic code.

`seqshoworfs(`*`SeqNT`*`)` displays the sequence with all open reading frames highlighted, and it returns a structure of start and stop positions for each ORF in each reading frame. The standard genetic code is used with start codon `'AUG'` and stop codons `'UAA'`, `'UAG'`, and `'UGA'`.

`seqshoworfs(`*`SeqNT`*`, ...'`*`PropertyName`*`',` *`PropertyValue`*`, ...)` calls `seqshoworfs` with optional properties that use property name/property value pairs. You can specify one or more properties in any order. Each *`PropertyName`* must be enclosed in single quotes and is case insensitive. These property name/property value pairs are as follows:

`seqshoworfs(`*`SeqNT`*`, ...'Frames',` *`FramesValue`*`, ...)` specifies the reading frames to display. The default is to display the first, second, and third reading frames with ORFs highlighted in each frame.

`seqshoworfs(`*`SeqNT`*`, ...'GeneticCode',` *`GeneticCodeValue`*`, ...)` specifies the genetic code to use for finding open reading frames.

`seqshoworfs(`*`SeqNT`*`, ...'MinimumLength',` *`MinimumLengthValue`*`, ...)` sets the minimum number of codons for an ORF to be considered valid. The default value is `10`.

`seqshoworfs(`*`SeqNT`*`, ...'AlternativeStartCodons',` *`AlternativeStartCodonsValue`*`, ...)` uses alternative start codons if `AlternativeStartCodons` is set to `true`. For example, in the human mitochondrial genetic code, `AUA` and `AUU` are known to be alternative start codons. For more details on alternative start codons, see

`https://www.ncbi.nlm.nih.gov/Taxonomy/Utils/wprintgc.cgi?mode=t#SG1`

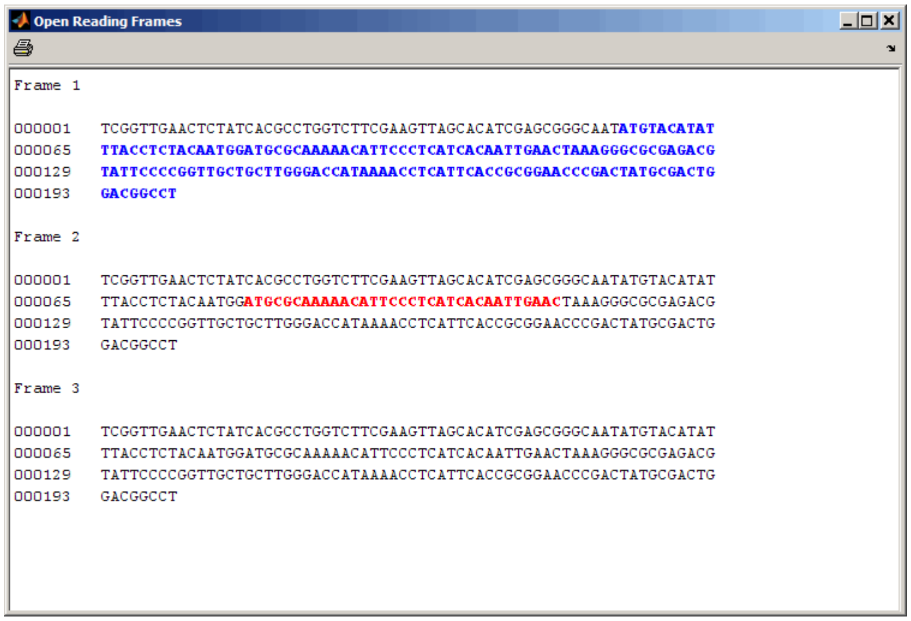
`seqshoworfs(`*`SeqNT`*`, ...'Color',` *`ColorValue`*`, ...)` specifies the color used to highlight the open reading frames in the output display. The default color scheme is blue for the first reading frame, red for the second, and green for the third.

`seqshoworfs(`*`SeqNT`*`, ...'Columns',` *`ColumnsValue`*`, ...)` specifies how many columns per line to use in the output. The default value is `64`.

Examples

Display the open reading frames in a random nucleotide sequence.

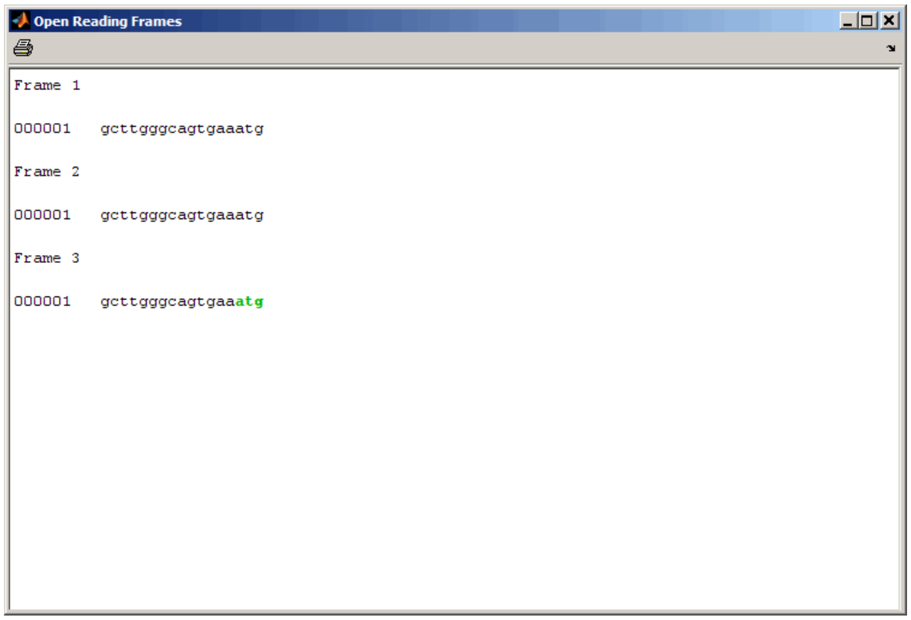
```
s = randseq(200, 'alphabet', 'dna');
seqshoworfs(s);
```



```
Frame 1

000001   TCGGTTGAACTCTATCACGCCTGGTCTTCGAAGTTAGCACATCGAGCGGGCAATATGTACATAT
000065   TTACCTCTACAATGGATGCGCAAAAACATTCCCTCATCACAATTGAACTAAAGGGCGCGAGACG
000129   TATTCCCCGGTTGCTGCTTGGGACCATAAAACCTCATTCACCGCGGAACCCGACTATGCGACTG
000193   GACGGCCT

Frame 2

000001   TCGGTTGAACTCTATCACGCCTGGTCTTCGAAGTTAGCACATCGAGCGGGCAATATGTACATAT
000065   TTACCTCTACAATGGATGCGCAAAAACATTCCCTCATCACAATTGAACTAAAGGGCGCGAGACG
000129   TATTCCCCGGTTGCTGCTTGGGACCATAAAACCTCATTCACCGCGGAACCCGACTATGCGACTG
000193   GACGGCCT

Frame 3

000001   TCGGTTGAACTCTATCACGCCTGGTCTTCGAAGTTAGCACATCGAGCGGGCAATATGTACATAT
000065   TTACCTCTACAATGGATGCGCAAAAACATTCCCTCATCACAATTGAACTAAAGGGCGCGAGACG
000129   TATTCCCCGGTTGCTGCTTGGGACCATAAAACCTCATTCACCGCGGAACCCGACTATGCGACTG
000193   GACGGCCT
```

Display the open reading frames in a GenBank sequence.

```
HLA_DQB1 = getgenbank('NM_002123');
seqshoworfs(HLA_DQB1.Sequence);
```



```
Frame 1

000001   gcttgggcagtgaaatg

Frame 2

000001   gcttgggcagtgaaatg

Frame 3

000001   gcttgggcagtgaaatg
```

More About

Color Options

The following lists the predefined colors and their RGB triplet equivalents. The short names and long names are character vectors that specify one of eight preset colors. The RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color; the intensities must be in the range [0 1].

RGB Triplet	Short Name	Long Name
`[1 1 0]`	`y`	`yellow`
`[1 0 1]`	`m`	`magenta`
`[0 1 1]`	`c`	`cyan`
`[1 0 0]`	`r`	`red`
`[0 1 0]`	`g`	`green`
`[0 0 1]`	`b`	`blue`
`[1 1 1]`	`w`	`white`
`[0 0 0]`	`k`	`black`

Version History

Introduced before R2006a

See Also

`codoncount` | `cpgisland` | `geneticcode` | `seqdisp` | `seqviewer` | `seqwordcount` | `regexp`

seqsplit

Split sequences into separate files based on barcodes

Syntax

```
seqsplit(fastqFile,barcodeFile)
seqsplit(___,Name,Value)
[outFiles,N] = seqsplit(___)
```

Description

`seqsplit(fastqFile,barcodeFile)` splits sequences in `fastqFile` according to the barcodes in `barcodeFile` and saves the sequences in separate files. By default, the output file name consists of the input file name followed by the barcode identifier. Sequences that do not match any provided barcodes, or that match multiple barcodes ambiguously, are saved in a file with the suffix `'_unmatched'` instead of the barcode identifier.

`seqsplit(___,Name,Value)` uses additional options specified by one or more `Name,Value` pair arguments.

`[outFiles,N] = seqsplit(___)` returns the names of output files in a cell array `outFiles`. `N` represents a vector containing the numbers of sequences saved in each output file.

Examples

Split sequences into separate files based on barcodes

Create a tab-delimited file with barcode IDs and barcode sequences.

```
barcodeInfo = {'ID1', 'AAAAC'; 'ID2', 'AGATT'; 'ID3', 'GACTT'};
writetable(cell2table(barcodeInfo), 'barcodeExample.txt', ...
       'Delimiter', '\t', 'WriteVariableNames', false);
```

Split sequences into separate output files based on the barcode sequences. By default, the function assumes that the barcode is located at the 5' end of each sequence, and no mismatches are allowed during barcode matching.

```
[outFiles, N] = seqsplit('SRR005164_1_50.fastq', 'barcodeExample.txt');
```

Check the number of sequences in each output file after splitting.

```
N
```

```
N = 3×1

     2
     1
     1
```

Allow up to two mismatches during the barcode matching.

```
[outFiles, N] = seqsplit('SRR005164_1_50.fastq', 'barcodeExample.txt', ...
         'MaxMismatches',2,'OutputSuffix','_MM2_split');
```

```
N
```

```
N = 3×1

     5
     9
     5
```

Input Arguments

fastqFile — Names of FASTQ files with sequence and quality information
character vector | string | string vector | cell array of character vectors

Names of FASTQ-formatted files with sequence and quality information, specified as a character vector, string, string vector, or cell array of character vectors.

Example: 'SRR005164_1_50.fastq'

barcodeFile — Name of barcode files with barcode information
character vector | string

Name of barcode file with barcode information, specified as a character vector or string. The file must be tab-formatted, containing barcode IDs and barcode sequences. Each ID must be followed by a barcode sequence, and all barcode sequences must have the same length.

Example: 'barcodeExample.txt'

Name-Value Pair Arguments

Specify optional pairs of arguments as Name1=Value1,...,NameN=ValueN, where Name is the argument name and Value is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose Name *in quotes.*

Example: 'MaxMismatches',2 specifies to allow up to 2 mismatches during barcode matching.

MaxMismatches — Maximum number of mismatches allowed during barcode matching
0 (default) | nonnegative integer

Maximum number of mismatches allowed during barcode matching, specified as a nonnegative integer. The default is 0, that is, no mismatches are allowed.

BarcodeFormat — Type of barcode to match
5 (default) | 3

Type of barcode to match, specified as 3 or 5. A value of 5 corresponds to the barcode located at the 5' end of each sequence, and 3 corresponds to the 3' end.

Example:

RemoveBarcode — Whether to remove the barcode
true (default) | false

Whether to remove the barcode and corresponding quality information from the matched sequences, specified as `true` or `false`. The default is `true`.

`WriteUnmatched` — Whether to save unmatched sequences
`false` (default) | `true`

Whether to save unmatched sequences and corresponding quality information in a separate output file, specified as `true` or `false`. The output file name has the suffix `'_unmatched'` instead of the barcode ID.

`OutputDir` — Relative or absolute path to output file directory
character vector | string

Relative or absolute path to the output file directory, specified as a character vector or string. The default is the current directory.

Example: `'OutputDir','F:\results'`

`OutputSuffix` — Suffix to use in output file name
`'_split'` (default) | character vector | string

Suffix to use in the output file name, specified as a character vector or string. It is inserted after the input file name and before the barcode ID. The default is `'_split'`.

`UseParallel` — Whether to perform computation in parallel
`false` (default) | `true`

Whether to perform computation in parallel, specified as `true` or `false`.

For parallel computing, you must have Parallel Computing Toolbox. If a parallel pool does not exist, one is created automatically when the auto-creation option is enabled in your parallel preferences. Otherwise, computation runs in serial mode.

Note There is a cost associated with sharing large input files across workers in a distributed environment. In some cases, running in parallel may not be beneficial in terms of performance.

Example: `'UseParallel',true`

Output Arguments

`outFiles` — Output file names
cell array of character vectors

Output file names, returned as a cell array of character vectors. By default, the name of each output file consists of the input file name followed by the output suffix (`'_split'`) and the barcode identifier.

`N` — Numbers of sequences saved in each output file
scalar | vector

Numbers of sequences saved in each output file, returned as a scalar or an *n*-by-`1` vector, where *n* is the number of output files. If there are multiple output files, the order within `N` corresponds to the order of the output files.

Version History

Introduced in R2016b

Extended Capabilities

Automatic Parallel Support
Accelerate code by automatically running computation in parallel using Parallel Computing Toolbox™.

To run in parallel, set `'UseParallel'` to `true`.

For more information, see the `'UseParallel'` name-value pair argument.

See Also

`seqtrim` | `seqfilter` | `seqsplitpe`

seqsplitpe

Split merged paired-end sequences into separate files

Syntax

```
seqsplitpe(fastqFile)
seqsplitpe(___,Name,Value)
[outFiles,N] = seqsplitpe(___)
```

Description

`seqsplitpe(fastqFile)` splits merged paired-end sequences from `fastqFile` into two separate files. Each sequence is split in the middle. The first half of the sequence is saved in the first output file and the other half in the second output file. By default, each output file name consists of the input file name appended with a suffix `'_1'` or `'_2'` before the file extension.

`seqsplitpe(___,Name,Value)` uses additional options specified by one or more `Name,Value` pair arguments.

`[outFiles,N] = seqsplitpe(___)` returns the names of output files in a cell array `outFiles`. `N` represents a vector containing the numbers of sequences saved in each output file.

Examples

Split merged paired-end sequences into separate files

Split each of the paired-end sequences in half, and store each half in separate output files.

```
[outFiles, N] = seqsplitpe('SXX123456_merged.fastq');
```

Check the number of sequences in each output file.

```
N
```

```
N = 2×1

    50
    50
```

Input Arguments

`fastqFile` — Names of FASTQ files with sequence and quality information
character vector | string | string vector | cell array of character vectors

Names of FASTQ files with sequence and quality information, specified as a character vector, string, string vector, or cell array of character vectors.

Example: `'SRR005164_1_50.fastq'`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `'OutputSuffix','PairedEnd_split'` specifies to use the custom suffix in the output file names.

`OutputDir` — Relative or absolute path to output file directory
character vector | string

Relative or absolute path to the output file directory, specified as a character vector or string. The default is the current directory.

Example: `'OutputDir','F:\results'`

`OutputSuffix` — Custom suffix to use in output file names
`''` (default) | character vector | string

Custom suffix to use in the output file names, specified as a character vector or string. It is inserted after the input file name and before the suffix `'_1'` or `'_2'`. The default is `''`.

Example: `'OutputSuffix','_MisMatches2'`

`UseParallel` — Boolean indicating whether to perform computation in parallel
`false` (default) | `true`

Boolean indicating whether to perform computation in parallel, specified as `true` or `false`.

For parallel computing, you must have Parallel Computing Toolbox. If a parallel pool does not exist, one is created automatically when the auto-creation option is enabled in your parallel preferences. Otherwise, computation runs in serial mode.

Note There is a cost associated with sharing large input files across workers in a distributed environment. In some cases, running in parallel may not be beneficial in terms of performance.

Example: `'UseParallel',true`

Output Arguments

`outFiles` — Output file names
cell array of character vectors

Output file names, returned as a cell array of character vectors. By default, the name of each output file consists of the input file name appended with a suffix `'_1'` or `'_2'` before the file extension.

`N` — Number of sequences saved in each output file
vector

Number of sequences saved in each output file, returned as an *n*-by-`1` vector where *n* is the number of output files. If there are multiple output files, the order within `N` corresponds to the order of the output files.

Version History

Introduced in R2016b

Extended Capabilities

Automatic Parallel Support

Accelerate code by automatically running computation in parallel using Parallel Computing Toolbox™.

To run in parallel, set `'UseParallel'` to `true`.

For more information, see the `'UseParallel'` name-value pair argument.

See Also

`seqtrim` | `seqfilter` | `seqsplit`

seqtrim

Trim sequences based on specified criterion

Syntax

```
seqtrim(fastqFile)
seqtrim(fastqFile,Name,Value)
[outFiles,nSeqTrimmed,nSeqUntrimmed] = seqtrim( ___ )
```

Description

`seqtrim(fastqFile)` trims the sequences in `fastqFile` and saves the trimmed sequences in new FASTQ files. By default, the trimmed sequences are saved under file names with the suffix `'_trimmed'` appended. If you do not specify any trimming criterion, the function trims sequences using the default.

`seqtrim(fastqFile,Name,Value)` uses additional options specified by one or more `Name,Value` pair arguments.

`[outFiles,nSeqTrimmed,nSeqUntrimmed] = seqtrim( ___ )` returns a cell array `outFiles` with the names of output files. `nSeqTrimmed` and `nSeqUntrimmed` represent the numbers of sequences trimmed and untrimmed from each input file, respectively.

Examples

Trim sequences in a FASTQ file

Trim each sequence when the number of bases with quality below 20 is greater than 3 within a sliding window of size 25.

```
[outFile,nt,unt] =  seqtrim('SRR005164_1_50.fastq', 'Method', 'MaxNumberLowQualityBases', ...
                 'Threshold', [3 20], 'WindowSize', 25);
```

Check the number of sequences that were trimmed.

```
nt
```

```
nt = 36
```

Check the number of sequences that were untrimmed.

```
unt
```

```
unt = 14
```

Trim the first 10 bases of each sequence.

```
[outfile,nt] = seqtrim('SRR005164_1_50.fastq','Method','Termini', ...
                     'Threshold',[10 0]);
```

Trim the last 5 bases.

```
[outfile,nt] = seqtrim('SRR005164_1_50.fastq','Method','Termini', ...
                       'Threshold',[0 5]);
```

Trim each sequence at position 50.

```
[outfile,nt] = seqtrim('SRR005164_1_50.fastq','Method','BasePositions', ...
                       'Threshold',[1 50]);
```

Trim each sequence when the running average base quality becomes less than 20.

```
[outFile,nt,unt] = seqtrim('SRR005164_1_50.fastq','Method','MeanQuality', ...
     'Threshold',20)
```

Trim each sequence when the percentage of bases with quality below 10 is more than 15.

```
[outFile,nt,unt] = seqtrim('SRR005164_1_50.fastq','Method','MaxPercentLowQualityBases', ...
     'Threshold',[15 10])
```

Input Arguments

`fastqFile` — Names of FASTQ files with sequence and quality information
character vector | string | string vector | cell array of character vectors

Names of FASTQ-formatted files with sequence and quality information, specified as a character vector, string, string vector, or cell array of character vectors.

Example: `'SRR005164_1_50.fastq'`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `'Method','MaxNumberLowQualityBases','Threshold',[3 20]` specifies to trim each sequence when the number of bases with quality below 20 is greater than 3.

`Method` — Criterion to trim sequences
`'MaxNumberLowQualityBases'` (default) | `'MaxPercentLowQualityBases'` | `'MeanQuality'` | `'BasePositions'` | `'Termini'`

Criterion to trim sequences, specified as one of the following options. Specify only one trimming criterion per function call.

- `'MaxNumberLowQualityBases'`– applies a maximum threshold on the number of low-quality bases allowed before trimming a sequence starting at the `5'` end.
- `'MaxPercentLowQualityBases'`– applies a maximum threshold on the percentage of low-quality bases allowed before trimming a sequence starting at the `5'` end.
- `'MeanQuality'`– applies a minimum threshold on the running average base quality allowed before trimming a sequence starting at the `5'` end.
- `'BasePositions'`– trims each sequence according to the base positions (first base and last base) starting at the `5'` end.

- `'Termini'`– trims each sequence from either the `5'` or `3'` end or from both ends.

Use this name-value pair argument together with `'Threshold'` to specify the appropriate threshold value. Depending on the trimming criterion, the corresponding value for `'Threshold'` varies. See the `'Threshold'` option for the default values.

Note Sequences resulting in empty sequences after trimming are saved in the output files as empty sequences. To remove empty sequences from files, use the `seqfilter` function with the `'MinLength'` option set to the value of `1`.

`Threshold` — Threshold value for trimming criterion

scalar | vector

Threshold value for the trimming criterion, specified as a scalar or vector. Use this name-value pair to define the threshold value for the trimming criterion specified by `'Method'`.

Depending on the trimming criterion, the corresponding value for `'Threshold'` can be a scalar or two-element vector. If you do not specify `'Threshold'`, then the function uses the default threshold value of the corresponding method. For each trimming criterion, the function uses the encoding format of the base quality specified by the `'Encoding'` name-value pair argument.

`'Method'`	`'Threshold'`	Default `'Threshold'` value
`'MaxNumberLowQualityBases'`	Two-element vector `[`*V1 V2*`]`. *V1* is a nonnegative integer that specifies the maximum number of low-quality bases allowed before trimming. *V2* specifies the minimum base quality. Any base with quality less than *V2* is considered a low-quality base.	`[0 10]`
`'MaxPercentLowQualityBases'`	Two-element vector `[`*V1 V2*`]`. *V1* is a scalar between 0 and 100 that specifies the maximum percentage of low quality bases allowed before trimming. *V2* specifies the minimum base quality. Any base with quality less than *V2* is considered a low-quality base.	`[0 10]`
`'MeanQuality'`	Positive scalar that specifies the minimum threshold on the running average base quality allowed before trimming a sequence starting at the `5'` end.	`0`

'Method'	'Threshold'	Default 'Threshold' value
'BasePositions'	Two-element vector [*V1 V2*], where *V1* and *V2* are positive integers specifying the base positions to start trimming at the 5' end and 3' end, respectively. To trim only the 5' end of each sequence before position *V1*, use [*V1* Inf]. To trim only the 3' end of each sequence after position *V2*, use [1 *V2*].	[1 Inf], that is, each sequence is left untrimmed.
'Termini'	Two-element vector [*V1 V2*], where V1 and V2 are nonnegative integers specifying the number of bases to trim at the 5' end and the 3' end, respectively. To trim *V1* bases at the 5' end only, use [*V1* 0]. To trim *V2* bases at the 3' end only, use [0 *V2*].	[0 0], that is, each sequence is left untrimmed.

WindowSize — Size of sliding window to apply filtering criterion to sequence
Inf (default) | positive integer

Size of the sliding window to apply the trimming criterion to a sequence, specified as a positive integer. The size of the window corresponds to the number of bases that the function uses at one time to apply the criterion. Any given sequence is trimmed before the first base of the window that violates the given criterion.

The sliding window can be applied to the following methods:

- 'MaxNumberLowQualityBases',
- 'MaxPercentLowQualityBases', and
- 'MeanQuality'.

Note Sequences shorter than the size of the window are saved in the output file as empty sequences. To remove empty sequences from files, use the seqfilter function with the 'MinLength' option set to the value of 1.

Encoding — Base quality encoding format
'Illumina18' (default) | 'Sanger' | 'Solexa' | 'Illumina13' | 'Illumina15'

Base quality encoding format, specified as a character vector or string.

OutputDir — Relative or absolute path to output file directory
character vector | string

Relative or absolute path to the output file directory, specified as a character vector or string. The default is the current directory.

Example: `'OutputDir','F:\results'`

`OutputSuffix` — Suffix to use in output file name
`'_trimmed'` (default) | character vector | string

Suffix to use in the output file name, specified as a character vector or string. It is inserted after the input file name and before the file extension. The default is `'_trimmed'`.

`UseParallel` — Boolean indicating whether to perform computation in parallel
`false` (default) | `true`

Boolean indicating whether to perform computation in parallel, specified as `true` or `false`.

For parallel computing, you must have Parallel Computing Toolbox. If a parallel pool does not exist, one is created automatically when the auto-creation option is enabled in your parallel preferences. Otherwise, computation runs in serial mode.

Note

- There is a cost associated with sharing large input files across workers in a distributed environment. In some cases, running in parallel may not be beneficial in terms of performance.
- During parallel computations, the work is divided by files, not by sequences, meaning that, for a single large file, running in parallel does not make a difference.

Example: `'UseParallel',true`

`OverWrite` — Flag to overwrite existing files
`false` or 0 (default) | `true` or 1

Flag to overwrite existing files, specified as a numeric or logical 1 (`true`) or 0 (`false`).

When the value is `false` and a file matching one of the output file names already exists, the function generates an error.

Data Types: `double` | `logical`

Output Arguments

`outFiles` — Output file names
cell array of character vectors

Output file names, returned as a cell array of character vectors.

`nSeqTrimmed` — Number of sequences trimmed from each input file
scalar | vector

Number of sequences trimmed from each input file, returned as a scalar or an *n*-by-1 vector where *n* is the number of input files. If there are multiple input files, the order within `nSeqTrimmed` corresponds to the order of the input files.

`nSeqUntrimmed` — Number of sequences untrimmed from each input file
scalar | vector

Number of sequences untrimmed from each input file, returned as a scalar or an *n*-by-`1` vector where *n* is the number of input files. If there are multiple input files, the order within `nSeqUntrimmed` corresponds to the order of the input files.

Version History

Introduced in R2016b

Extended Capabilities

Automatic Parallel Support

Accelerate code by automatically running computation in parallel using Parallel Computing Toolbox™.

To run in parallel, set `'UseParallel'` to `true`.

For more information, see the `'UseParallel'` name-value pair argument.

See Also

`seqsplit` | `seqsplitpe` | `seqfilter`

seqviewer

Visualize and interactively explore biological sequences

Syntax

```
seqviewer
seqviewer(Seq)
seqviewer(Seq,Name,Value)

seqviewer('close')
```

Description

`seqviewer` opens the Sequence Viewer app.

`seqviewer(Seq)` loads a sequence `Seq` into the app, where you can view and interactively explore the sequence.

`seqviewer(Seq,Name,Value)` opens the app with additional options specified by one or more `Name,Value` pair arguments.

`seqviewer('close')` closes the Sequence Viewer app.

Input Arguments

Seq — Amino acid or nucleotide sequence
character vector | string | row vector of integers | structure

Amino acid or nucleotide sequence, specified as:

- Character vector or string containing single-letter codes or a file name with an extension of .gbk, .gpt, .fasta, .fa, or .ebi.
- Row vector of integers
- MATLAB structure containing a `Sequence` field that contains an amino acid or nucleotide sequence, such as returned by `fastaread`, `fastqread`, `getgenpept`, `genpeptread`, `getpdb`, `pdbread`, `emblread`, `getembl`, `genbankread`, or `getgenbank`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `'Alphabet','AA'` specifies that the aligned sequences are amino acid sequences.

`Alphabet` — Type of aligned sequences
`'AA'` | `'NT'`

Type of aligned sequences, specified as `'AA'` for amino acid sequences or `'NT'` for nucleotide sequences.

Example: `'Alphabet','AA'`

Examples

Open and View a Biological Sequence

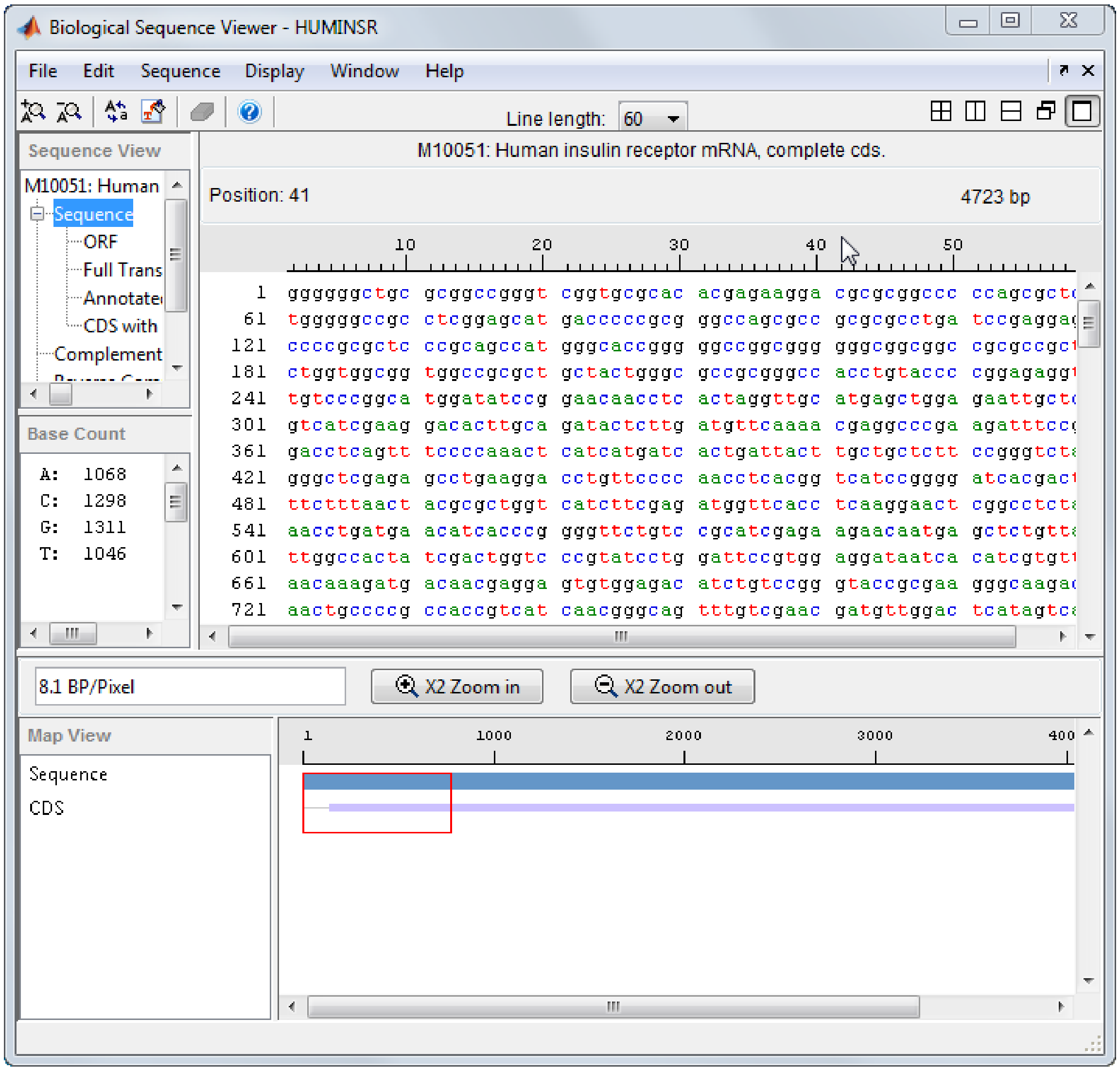
Retrieve a sequence from the GenBank database.

```
S = getgenbank('M10051');
```

Load the sequence into the Sequence Viewer app.

```
seqviewer(S)
```

Alternatively, you can click Sequence Viewer on the **Apps** tab to open the app, and view the biological sequence `S`.



M10051: Human insulin receptor mRNA, complete cds.

Position: 41 4723 bp

```
A:   1068
C:   1298
G:   1311
T:   1046
```

```
            10        20        30        40        50
  1 gggggctgc gcggccgggt cggtgcgcac acgagaagga cgcgcggccc ccagcgct
 61 tgggggccgc ctcggagcat gaccccgcg ggccagcgcc gcgcgcctga tccgagga
121 ccccgcgctc ccgcagccat gggcaccggg ggccggcggg gggcggcggc cgcgccgc
181 ctggtggcgg tggccgcgct gctactgggc gccgcgggcc acctgtaccc cggagagg
241 tgtcccggca tggatatccg gaacaacctc actaggttgc atgagctgga gaattgct
301 gtcatcgaag gacacttgca gatactcttg atgttcaaaa cgaggcccga agatttcc
361 gacctcagtt tccccaaact catcatgatc actgattact tgctgctctt ccgggtct
421 gggctcgaga gcctgaagga cctgttcccc aacctcacgg tcatccgggg atcacgac
481 ttctttaact acgcgctggt catcttcgag atggttcacc tcaaggaact cggcctct
541 aacctgatga acatcacccg gggttctgtc cgcatcgaga agaacaatga gctctgtt
601 ttggccacta tcgactggtc ccgtatcctg gattccgtgg aggataatca catcgtgt
661 aacaaagatg acaacgagga gtgtggagac atctgtccgg gtaccgcgaa gggcaaga
721 aactgccccg ccaccgtcat caacgggcag tttgtcgaac gatgttggac tcatagtc
```

8.1 BP/Pixel

```
1        1000        2000        3000        400
```

Close the app.

```
seqviewer('close')
```

Version History

Introduced before R2006a

See Also

aa2nt | aacount | aminolookup | basecount | baselookup | dimercount | emblread | fastaread | fastawrite | genbankread | geneticcode | genpeptread | getembl | getgenbank | getgenpept | nt2aa | proteinplot | seqcomplement | seqdisp | seqrcomplement | seqreverse | seqshoworfs | seqwordcount

Topics

"Exploring a Nucleotide Sequence Using the Sequence Viewer App"

seqwordcount

Count number of occurrences of word in sequence

Syntax

`seqwordcount(Seq, Word)`

Arguments

Seq	Character vector or string containing a nucleotide or amino acid sequence. You can also enter a structure with the field `Sequence`.
Word	Enter a short sequence of characters.

Description

`seqwordcount(Seq, Word)` counts the number of times that a word appears in a sequence, and then returns the number of occurrences of that word.

If *Word* contains nucleotide or amino acid symbols that represent multiple possible symbols (ambiguous characters), then `seqwordcount` counts all matches. For example, the symbol `R` represents either `G` or `A` (purines). For another example, if `word` equals `'ART'`, then `seqwordcount` counts occurrences of both `'AAT'` and `'AGT'`.

Examples

`seqwordcount` does not count overlapping patterns multiple times. In the following example, `seqwordcount` reports three matches. `TATATATA` is counted as two distinct matches, not three overlapping occurrences.

```
seqwordcount('GCTATAACGTATATATAT','TATA')

ans =
     3
```

The following example reports two matches (`'TAGT'` and `'TAAT'`). `B` is the ambiguous code for `G`, `T`, or `C`, while `R` is an ambiguous code for `G` and `A`.

```
seqwordcount('GCTAGTAACGTATATATAAT','BART')

ans =
     2
```

Version History

Introduced before R2006a

See Also
`codoncount` | `seqshoworfs` | `seqviewer` | `seq2regexp` | `strfind`

set (biograph)

(To be removed) Set property of biograph object

Note The `biograph` object and its methods will be removed in a future release. Use `graph` or `digraph` instead.

Syntax

```
set(BGobj)
set(BGobj, 'PropertyName')
set(BGobj, 'PropertyName', PropertyValue)
set(BGobj, 'Property1Name', Property1Value, 'Property2Name',
Property2Value, ...)
```

Arguments

BGobj	Biograph object created with the function `biograph`.
PropertyName	Property name for a biograph object.
PropertyValue	Value of the property specified by *PropertyName*.

Description

`set(`*`BGobj`*`)` displays possible values for all properties that have a fixed set of property values in *BGobj*, a biograph object.

`set(`*`BGobj`*`, '`*`PropertyName`*`')` displays possible values for a specific property that has a fixed set of property values in *BGobj*, a biograph object.

`set(`*`BGobj`*`, '`*`PropertyName`*`', `*`PropertyValue`*`)` sets the specified property of *BGobj*, a biograph object.

`set(`*`BGobj`*`, '`*`Property1Name`*`', `*`Property1Value`*`, '`*`Property2Name`*`', `*`Property2Value`*`, ...)` sets the specified properties of *BGobj*, a biograph object.

Properties of a Biograph Object

Property	Description
`ID`	Character vector to identify the biograph object. Default is `''`.
`Label`	Character vector to label the biograph object. Default is `''`.
`Description`	Character vector that describes the biograph object. Default is `''`.
`LayoutType`	Character vector that specifies the algorithm for the layout engine. Choices are: • `'hierarchical'` (default) — Uses a topological order of the graph to assign levels, and then arranges the nodes from top to bottom, while minimizing crossing edges. • `'radial'` — Uses a topological order of the graph to assign levels, and then arranges the nodes from inside to outside of the circle, while minimizing crossing edges. • `'equilibrium'` — Calculates layout by minimizing the energy in a dynamic spring system.
`EdgeType`	Character vector that specifies how edges display. Choices are: • `'straight'` • `'curved'` (default) • `'segmented'` **Note** Curved or segmented edges occur only when necessary to avoid obstruction by nodes. Biograph objects with `LayoutType` equal to `'equilibrium'` or `'radial'` cannot produce curved or segmented edges.
`Scale`	Positive number that post-scales the node coordinates. Default is `1`.
`LayoutScale`	Positive number that scales the size of the nodes before calling the layout engine. Default is `1`.
`EdgeTextColor`	Three-element numeric vector of RGB values. Default is `[0, 0, 0]`, which defines black.
`EdgeFontSize`	Positive number that sets the size of the edge font in points. Default is `8`.
`ShowArrows`	Controls the display of arrows with the edges. Choices are `'on'` (default) or `'off'`.
`ArrowSize`	Positive number that sets the size of the arrows in points. Default is `8`.
`ShowWeights`	Controls the display of text indicating the weight of the edges. Choices are `'on'` or `'off'` (default).

Property	Description
`ShowTextInNodes`	Character vector that specifies the node property used to label nodes when you display a biograph object using the `view` method. Choices are: • `'Label'` — Uses the `Label` property of the node object (default). • `'ID'` — Uses the `ID` property of the node object. • `'None'`
`NodeAutoSize`	Controls precalculating the node size before calling the layout engine. Choices are `'on'` (default) or `'off'`. **Note** Set it to `off` if you want to apply different node sizes by changing the `Size` property.
`NodeCallback`	User-defined callback for all nodes. Enter the name of a function, a function handle, or a cell array with multiple function handles. After using the `view` function to display the biograph object in the Biograph Viewer, you can double-click a node to activate the first callback, or right-click and select a callback to activate. Default is the anonymous function, `@(node) inspect(node)`, which displays the Property Inspector dialog box.
`EdgeCallback`	User-defined callback for all edges. Enter the name of a function, a function handle, or a cell array with multiple function handles. After using the `view` function to display the biograph object in the Biograph Viewer, you can right-click and select a callback to activate. Default is the anonymous function, `@(edge) inspect(edge)`, which displays the Property Inspector dialog box.
`CustomNodeDrawFcn`	Function handle to a customized function to draw nodes. Default is `[]`.
`Nodes`	Read-only column vector with handles to node objects of a biograph object. The size of the vector is the number of nodes. For properties of node objects, see Properties of a Node Object.
`Edges`	Read-only column vector with handles to edge objects of a biograph object. The size of the vector is the number of edges. For properties of edge objects, see Properties of an Edge Object.

Version History

Introduced in R2008b

R2021b: `biograph` and its methods will be removed

Not recommended starting in R2021b

The `biograph` object and its methods will be removed in a future release. Use `graph` or `digraph` instead.

See Also

`graph` | `digraph`

set

Set property of object

Syntax

```
newObject = set(object,Name,Value)
set(object,propertyName)
set(object)
allProperties = set(object)
```

Description

`newObject = set(object,Name,Value)` returns a new object that is a copy of `object` with properties set to the values specified by using one or more name-value pairs. Use single quotes around the property name. For example, `newObj = set(brObj,'Sequence',{'ACTCAG','GTCATG'})` specifies the `Sequence` property of `brObj`. You can specify any property name, except `NSeqs`. See `BioRead` or `BioMap` for their properties.

`set(object,propertyName)` displays all possible values for the specified property `PropName` of the object.

`set(object)` displays all properties of the object and their possible values.

`allProperties = set(object)` returns the structure `allProperties` containing all properties of `object` and their possible values.

Examples

Modify NGS Data

Store read data from a SAM-formatted file in a BioRead object. Set `'InMemory'` to `true` to load the object into memory so that you can modify its properties.

```
br = BioRead('SRR005164_1_50.fastq','InMemory',true)
```

```
br = 
  BioRead with properties:

     Quality: {50x1 cell}
    Sequence: {50x1 cell}
      Header: {50x1 cell}
       NSeqs: 50
        Name: ''
```

Check the list of object properties and their possible values. For example, the `Header` property takes a cell array of strings as its value.

```
allProperties = set(br)
allProperties = struct with fields:
     Quality: 'Cell array of strings.'
    Sequence: 'Cell array of strings.'
      Header: 'Cell array of strings.'
       NSeqs: 'Non negative integer.'
        Name: 'String.'
```

Specify custom header information that follows the pattern Header_1, Header_2, ... ,Header_50.

```
headers = cell(50,1);
for i = 1:50
    headers(i) = {['Header_' int2str(i)]};
end
```

Set the header property of the `br` object. Use the same object as the output to update an existing object.

```
br = set(br,'Header',headers)
```

```
br = 
  BioRead with properties:

     Quality: {50x1 cell}
    Sequence: {50x1 cell}
      Header: {50x1 cell}
       NSeqs: 50
        Name: ''
```

```
br.Header(1)
```

```
ans = 1x1 cell array
    {'Header_1'}
```

Alternatively, you can set the property by using the dot notation.

```
br.Header = headers
```

```
br = 
  BioRead with properties:

     Quality: {50x1 cell}
    Sequence: {50x1 cell}
      Header: {50x1 cell}
       NSeqs: 50
        Name: ''
```

Input Arguments

`object` — Object containing read data

BioRead object | BioMap object

Object containing the read data, specified as a `BioRead` or `BioMap` object. If the object is not stored in memory, you cannot modify its properties, except the `Name` property.

Example: `readData`

`propertyName` — Name of object property
character vector | string

Name of the object property, specified as a character vector or string.

Example: `'Sequence'`

Output Arguments

`newObject` — New object with updated properties
`BioRead` object | `BioMap` object

New object with updated properties, returned as a `BioRead` or `BioMap` object.

`allProperties` — Structure containing all properties and values
`struct`

Structure containing all properties and their possible values, returned as a `struct`. The field names are the property names, and the values are cell arrays of possible property values.

Version History

Introduced in R2010a

See Also

`BioRead` | `BioMap`

Topics
"Manage Sequence Read Data in Objects"

set (DataMatrix)

Set property of DataMatrix object

Syntax

```
set(DMObj)
set(DMObj, 'PropertyName')
DMObj = set(DMObj, 'PropertyName', PropertyValue)
DMObj = set(DMObj, 'Property1Name', Property1Value, 'Property2Name',
Property2Value, ...)
```

Arguments

DMObj	DataMatrix object, such as created by `DataMatrix` (object constructor).
PropertyName	Property name of a DataMatrix object.
PropertyValue	Value of the property specified by *PropertyName*.

Description

`set(`*`DMObj`*`)` displays possible values for all properties that have a fixed set of property values in *DMObj*, a DataMatrix object.

`set(`*`DMObj`*`, '`*`PropertyName`*`')` displays possible values for a specific property that has a fixed set of property values in *DMObj*, a DataMatrix object.

`DMObj`` = set(`*`DMObj`*`, '`*`PropertyName`*`', `*`PropertyValue`*`)` sets the specified property of *DMObj*, a DataMatrix object.

`DMObj`` = set(`*`DMObj`*`, '`*`Property1Name`*`', `*`Property1Value`*`, '`*`Property2Name`*`',`
`Property2Value``, ...)` sets the specified properties of *DMObj*, a DataMatrix object.

Properties of a DataMatrix Object

Property	Description
`Name`	Character vector that describes the DataMatrix object. Default is `''`.
`RowNames`	Empty array or cell array of character vectors that specifies the names for the rows, typically gene names or probe identifiers. The number of elements in the cell array must equal the number of rows in the matrix. Default is an empty array.
`ColNames`	Empty array or cell array of character vectors that specifies the names for the columns, typically sample identifiers. The number of elements in the cell array must equal the number of columns in the matrix.
`NRows`	Positive number that specifies the number of rows in the matrix. **Note** You cannot modify this property directly. You can access it using the `get` method.
`NCols`	Positive number that specifies the number of columns in the matrix. **Note** You cannot modify this property directly. You can access it using the `get` method.
`NDims`	Positive number that specifies the number of dimensions in the matrix. **Note** You cannot modify this property directly. You can access it using the `get` method.
`ElementClass`	Character vector that specifies the class type, such as `single` or `double`. **Note** You cannot modify this property directly. You can access it using the `get` method.

Examples

1 Load the MAT-file, provided with the Bioinformatics Toolbox software, that contains yeast data. This MAT-file includes three variables: `yeastvalues`, a matrix of gene expression data, `genes`, a cell array of GenBank accession numbers for labeling the rows in `yeastvalues`, and `times`, a vector of time values for labeling the columns in `yeastvalues`.

```
load filteredyeastdata
```

2 Import the microarray object package so that the `DataMatrix` constructor function will be available.

```
import bioma.data.*
```

3 Create a DataMatrix object from the gene expression data in the first 30 rows of the `yeastvalues` matrix.

```
dmo = DataMatrix(yeastvalues(1:30,:));
```

4 Use the `get` method to display the properties of the DataMatrix object, `dmo`.

```
get(dmo)

           Name: ''
        RowNames: []
        ColNames: []
           NRows: 30
           NCols: 7
           NDims: 2
    ElementClass: 'double'
```

Notice that the `RowNames` and `ColNames` fields are empty.

5 Use the `set` method and the `genes` and `times` variables to specify row names and column names for the DataMatrix object, `dmo`.

```
dmo = set(dmo,'RowNames',genes(1:30),'ColNames',times)
```

6 Use the `get` method to display the properties of the DataMatrix object, `dmo`.

```
get(dmo)

           Name: ''
        RowNames: {30x1 cell}
        ColNames: {'   0'  ' 9.5'  '11.5'  '13.5'  '15.5'  '18.5'  '20.5'}
           NRows: 30
           NCols: 7
           NDims: 2
    ElementClass: 'double'
```

Version History

Introduced in R2008b

See Also

`DataMatrix` | `get`

Topics

DataMatrix object on page 1-734

setHeader

Update header information of reads

Syntax

```
newObject = setHeader(object,headerInfo)
newObject = setHeader(object,headerInfo,subset)
```

Description

`newObject = setHeader(object,headerInfo)` returns a new object that is a copy of `object` with the `Header` property set to `headerInfo`.

`newObject = setHeader(object,headerInfo,subset)` returns a new object that is a copy of `object` with the `Header` property of a subset of elements set to `headerInfo`. A one-to-one relationship must exist between the number and order of elements in `headerInfo` and `subset`.

Examples

Update Header Information

Store read data from a SAM-formatted file in a BioRead object. Set `'InMemory'` to `true` to load the object into memory so that you can modify its properties.

```
br = BioRead('SRR005164_1_50.fastq','InMemory',true)
```

```
br =
  BioRead with properties:

     Quality: {50x1 cell}
    Sequence: {50x1 cell}
      Header: {50x1 cell}
       NSeqs: 50
        Name: ''
```

Check the header information for the first three elements of the object.

```
br.Header(1:3)
```

```
ans = 3x1 cell
    {'SRR005164.1'}
    {'SRR005164.2'}
    {'SRR005164.3'}
```

Define custom headers for the first three elements.

```
headers = {'Header1','Header2','Header3'};
```

Set the header information of the first three elements. `br2` is a copy of `br` with the headers updated. If you need to update the br object itself, set it as the output of the function.

```
br2 = setHeader(br,headers,[1:3]);
br2.Header(1:3)
```

```
ans = 3x1 cell
    {'Header1'}
    {'Header2'}
    {'Header3'}
```

You can also update the headers of the br object directly using dot notation.

```
br.Header(1:3) = headers;
br.Header(1:3)
```

```
ans = 3x1 cell
    {'Header1'}
    {'Header2'}
    {'Header3'}
```

Input Arguments

`object` — Object containing read data
BioRead object | BioMap object

Object containing the read data, specified as a `BioRead` or `BioMap` object. If the object is not stored in memory, you cannot modify its properties, except the `Name` property.

Example: `readData`

`headerInfo` — Header information
cell array of character vectors | string vector

Header information of the reads, specified as a cell array of character vectors or string vector.

Example: `{'H1','H2','H3'}`

`subset` — Subset of elements in object
vector of positive integers | logical vector | string vector | cell array of character vectors

Subset of elements in the object, specified as a vector of positive integers, logical vector, string vector, or cell array of character vectors containing valid sequence headers.

Example: `[1 3]`

Tip When you use a sequence header (or a cell array of headers) for `subset`, a repeated header specifies all elements with that header.

Output Arguments

`newObject` — New object with updated properties
BioRead object | BioMap object

New object with updated properties, returned as a `BioRead` or `BioMap` object.

Version History

Introduced in R2010a

See Also

`BioRead` | `BioMap`

Topics

"Manage Sequence Read Data in Objects"

setQuality

Update quality information

Syntax

```
newObject = setQuality(object,qualityInfo)
newObject = setQuality(object,qualityInfo,subset)
```

Description

`newObject = setQuality(object,qualityInfo)` returns a new object that is a copy of `object` with the `Quality` property set to `qualityInfo`.

`newObject = setQuality(object,qualityInfo,subset)` returns a new object that is a copy of `object` with the `Quality` property of a subset of elements set to `qualityInfo`. A one-to-one relationship must exist between the number and order of elements in `qualityInfo` and `subset`.

Examples

Update Quality Information

Store read data from a SAM-formatted file in a BioRead object. Set `'InMemory'` to `true` to load the object into memory so that you can modify its properties.

```
br = BioRead('SRR005164_1_50.fastq','InMemory',true)
```

```
br = 
  BioRead with properties:

     Quality: {50x1 cell}
    Sequence: {50x1 cell}
      Header: {50x1 cell}
       NSeqs: 50
        Name: ''
```

Check the quality information for the first three elements of the object.

```
br.Quality(1:3)
```

```
ans = 3x1 cell
    {'<A<<@=+><'<<<<>8<>8<<<>;&<=7>8=9#<;<?9<<<<?9<<<<A;<<A<<<@:<<<<>7<?<)<;A;'
    {'A<<<;<<<<@:<<<;;A;;A;@=*<<<<A>+<<<A;<<<<<<;;;<;<<@;@9<;<<B<B?.@9<@:A>,<?<)<<<A<<B@81*"<<<<
    {'<<<<<<;A;<<<<@=*:?9CA90(<6;9;<;<;A<<<;B<<B@5&<;<<A;:A<<B@/A;<<;:<<@:;<;<<<;?9<<<<<<;<:;<1<
```

Generate random quality scores for the first three reads. Assume that the quality scores are between 0 (equivalent to the ASCII code of 33) and 60 (equivalent to the ASCII code of 93).

```
qualities = cell(3,1);
rng('default');
```

```
for i = 1:3
    qualities{i} = char(floor((93 - 33) * rand(1,length(br.Quality{i}))) + 33);
end
qualities

qualities = 3x1 cell
    {'QW(WF&1AZZ*[Z>Q):WPZH#SYINM8H+K"1#&RJ4Z#;7NP,>;GKN1IH*(>Z5D.N0?JVZA))0S0'
    {'Q/X5,0E=6RDAX2NN7C%$@OY(C=!5*P3@*E0HJM<&.W*RA\%;'Z!ORU&80Q:W+0))UCA)TF6?9%/(,/:#WY>>5W7'O8/
    {'?RPG7Q@6YUBFD-3=.S,.+.;3X:,W[;'09D0EK.(24:?&0Q"XL>C/<ZA@.>FI87\#VWP&05I)L'H>OKWV5J,"M?=WEF
```

Update the quality information of the first three elements. `br2` is a copy of `br` with the updated quality scores. If you need to update the br object itself, set it as the output of the function.

```
br2 = setQuality(br,qualities,[1:3]);
br2.Quality(1:3)

ans = 3x1 cell
    {'QW(WF&1AZZ*[Z>Q):WPZH#SYINM8H+K"1#&RJ4Z#;7NP,>;GKN1IH*(>Z5D.N0?JVZA))0S0'
    {'Q/X5,0E=6RDAX2NN7C%$@OY(C=!5*P3@*E0HJM<&.W*RA\%;'Z!ORU&80Q:W+0))UCA)TF6?9%/(,/:#WY>>5W7'O8/
    {'?RPG7Q@6YUBFD-3=.S,.+.;3X:,W[;'09D0EK.(24:?&0Q"XL>C/<ZA@.>FI87\#VWP&05I)L'H>OKWV5J,"M?=WEF
```

You can update the quality scores of the br object directly by using dot notation.

```
br.Quality(1:3) = qualities;
br.Quality(1:3)

ans = 3x1 cell
    {'QW(WF&1AZZ*[Z>Q):WPZH#SYINM8H+K"1#&RJ4Z#;7NP,>;GKN1IH*(>Z5D.N0?JVZA))0S0'
    {'Q/X5,0E=6RDAX2NN7C%$@OY(C=!5*P3@*E0HJM<&.W*RA\%;'Z!ORU&80Q:W+0))UCA)TF6?9%/(,/:#WY>>5W7'O8/
    {'?RPG7Q@6YUBFD-3=.S,.+.;3X:,W[;'09D0EK.(24:?&0Q"XL>C/<ZA@.>FI87\#VWP&05I)L'H>OKWV5J,"M?=WEF
```

Input Arguments

`object` — Object containing read data

`BioRead` object | `BioMap` object

Object containing the read data, specified as a `BioRead` or `BioMap` object. If the object is not stored in memory, you cannot modify its properties, except the `Name` property.

Example: `readData`

`qualityInfo` — Quality information

cell array of character vectors

Quality information of reads, specified as a cell array of character vectors or string vector.

Example: `{'<A<<@=+>','A<<<;<<'}`

`subset` — Subset of elements in object

vector of positive integers | logical vector | string vector | cell array of character vectors

Subset of elements in the object, specified as a vector of positive integers, logical vector, string vector, or cell array of character vectors containing valid sequence headers.

Example: `[1 3]`

Tip When you use a sequence header (or a cell array of headers) for `subset`, a repeated header specifies all elements with that header.

Output Arguments

`newObject` — New object with updated properties
`BioRead` object | `BioMap` object

New object with updated properties, returned as a `BioRead` or `BioMap` object.

Version History

Introduced in R2010a

See Also

`BioRead` | `BioMap`

Topics
"Manage Sequence Read Data in Objects"

setSequence

Update read sequences

Syntax

```
newObject = setSequence(object,sequenceInfo)
newObject = setSequence(object,sequenceInfo,subset)
```

Description

`newObject = setSequence(object,sequenceInfo)` returns a new object that is a copy of `object` with the `Sequence` property set to `sequenceInfo`.

`newObject = setSequence(object,sequenceInfo,subset)` returns a new object that is a copy of `object` with the `Sequence` property of a subset of elements set to `sequenceInfo`. A one-to-one relationship must exist between the number and order of elements in `sequenceInfo` and `subset`.

Examples

Update Read Sequences

Store read data from a SAM-formatted file in a BioRead object. Set `'InMemory'` to `true` to load the object into memory so that you can modify its properties.

```
br = BioRead('SRR005164_1_50.fastq','InMemory',true)
```

```
br = 
  BioRead with properties:

     Quality: {50x1 cell}
    Sequence: {50x1 cell}
      Header: {50x1 cell}
       NSeqs: 50
        Name: ''
```

Check the read sequences of the first three elements of the object.

```
br.Sequence(1:3)
```

```
ans = 3x1 cell
    {'TGGCTTTAAAGCAGAACTTGTGAAAGAAGGAAAGCATTATGATTATCTGGCTAAGCTTAGCATTGTTTAGAA'
    {'TTACACTATCCTCTGATTACCAAAGACGTTTCTCGGTCATACAGACAGTCCTTGAGCAAGGGAAGAATTTATTTGCAGGCAAAAAAGTGT
    {'CACGAGCGGTATATTTGCCTTTTTGTGCTGTGATTCGATTCTTTTCTCTCCTCCACCCAAGCGAGCTTGCTCACGAAGTGCGATGAGCTC
```

Generate random sequences for the first three reads. Use the `randseq` function to generate random sequences with the same length as the original sequences.

```
sequenceInfo = cell(3,1);
rng('default');
```

```
for i = 1:3
    sequenceInfo{i} = randseq(length(br.Quality{i}));
end
sequenceInfo
```

```
sequenceInfo = 3x1 cell
    {'TTATGACGTTATTCTACTTTGATTGTGCGAGACAATGCTACCTTACCGGTCGGAACTCGATCGGTTGAACTC'
    {'TATCACGCCTGGTCTTCGAAGTTAGCACATCGAGCGGGCAATATGTACATATTTACCTCTACAATGGATGCGCAAAAACATTCCCTCATC
    {'GTTGCTGCTTGGGACCATAAAACCTCATTCACCGCGGAACCCGACTATGCGACTGGACGGCCTATTTACCGAGAGCTGTTCGAAGGCTGG
```

Update the sequences of the first three elements. `br2` is a copy of `br` with updated read sequences. If you need to update the br object itself, set it as the output of the function.

```
br2 = setSequence(br,sequenceInfo,[1:3]);
br2.Sequence(1:3)
```

```
ans = 3x1 cell
    {'TTATGACGTTATTCTACTTTGATTGTGCGAGACAATGCTACCTTACCGGTCGGAACTCGATCGGTTGAACTC'
    {'TATCACGCCTGGTCTTCGAAGTTAGCACATCGAGCGGGCAATATGTACATATTTACCTCTACAATGGATGCGCAAAAACATTCCCTCATC
    {'GTTGCTGCTTGGGACCATAAAACCTCATTCACCGCGGAACCCGACTATGCGACTGGACGGCCTATTTACCGAGAGCTGTTCGAAGGCTGG
```

You can also update the sequences of the br object directly using dot notation.

```
br.Sequence(1:3) = sequenceInfo;
br.Sequence(1:3)
```

```
ans = 3x1 cell
    {'TTATGACGTTATTCTACTTTGATTGTGCGAGACAATGCTACCTTACCGGTCGGAACTCGATCGGTTGAACTC'
    {'TATCACGCCTGGTCTTCGAAGTTAGCACATCGAGCGGGCAATATGTACATATTTACCTCTACAATGGATGCGCAAAAACATTCCCTCATC
    {'GTTGCTGCTTGGGACCATAAAACCTCATTCACCGCGGAACCCGACTATGCGACTGGACGGCCTATTTACCGAGAGCTGTTCGAAGGCTGG
```

Input Arguments

`object` — Object containing read data

`BioRead` object | `BioMap` object

Object containing the read data, specified as a `BioRead` or `BioMap` object. If the object is not stored in memory, you cannot modify its properties, except the `Name` property.

Example: `readData`

`sequenceInfo` — Read sequences

cell array of character vectors | string vector

Read sequences, specified as a cell array of character vectors or string vector containing nucleotide sequences.

Example: `{'TGGCTTC','AAAGCAGTACG'}`

`subset` — Subset of elements in object

vector of positive integers | logical vector | string vector | cell array of character vectors

Subset of elements in the object, specified as a vector of positive integers, logical vector, string vector, or cell array of character vectors containing valid sequence headers.

Example: [1 3]

Tip When you use a sequence header (or a cell array of headers) for `subset`, a repeated header specifies all elements with that header.

Output Arguments

`newObject` — New object with updated properties
`BioRead` object | `BioMap` object

New object with updated properties, returned as a `BioRead` or `BioMap` object.

Version History

Introduced in R2010a

See Also

`BioRead` | `BioMap`

Topics
“Manage Sequence Read Data in Objects”

setSubsequence

Update partial sequences

Syntax

`newObject = setSubsequence(object,subsequences,subset,positions)`

Description

`newObject = setSubsequence(object,subsequences,subset,positions)` returns a new object that is a copy of `object` with the partial sequences of a subset of elements set to `subsequences`. The `positions` argument specifies the sequence positions to be updated by `subsequences`. A one-to-one relationship must exist between the number and order of elements in `subsequences` and `subset`.

Examples

Update Partial Sequences

Store read data from a SAM-formatted file in a BioRead object. Set `'InMemory'` to `true` to load the object into memory so that you can modify its properties.

```
br = BioRead('SRR005164_1_50.fastq','InMemory',true)
```

```
br = 
  BioRead with properties:

     Quality: {50x1 cell}
    Sequence: {50x1 cell}
      Header: {50x1 cell}
       NSeqs: 50
        Name: ''
```

Assume that you want to update the sequences of the first two reads partially (for example, the first five positions). First check the existing sequences.

```
br.Sequence(1:2)
```

```
ans = 2x1 cell
    {'TGGCTTTAAAGCAGAACTTGTGAAAGAAGGAAAGCATTATGATTATCTGGCTAAGCTTAGCATTGTTTAGAA'
    {'TTACACTATCCTCTGATTACCAAAGACGTTTCTCGGTCATACAGACAGTCCTTGAGCAAGGGAAGAATTTATTTGCAGGCAAAAAAGTGT
```

Define the subsequences. Each subsequence must have the same length.

```
subSequences = {'ATTCG','TACTA'}
```

```
subSequences = 1x2 cell
    {'ATTCG'}    {'TACTA'}
```

Update the first five positions of the first two reads. The number of positions must equal the length of each subsequence. In this example, the total number of positions is five, as is the length of each subsequence. `br2` is a copy of `br` with updated read sequences. If you need to update the br object itself, set it as the output of the function.

```
positions   = [1:5];
subset      = [1 2];
br2         = setSubsequence(br,subSequences,subset,positions);
br2.Sequence(1:2)

ans = 2x1 cell
    {'ATTCGTTAAAGCAGAACTTGTGAAAGAAGGAAAGCATTATGATTATCTGGCTAAGCTTAGCATTGTTTAGAA'
    {'TACTACTATCCTCTGATTACCAAAGACGTTTCTCGGTCATACAGACAGTCCTTGAGCAAGGGAAGAATTTATTTGCAGGCAAAAAAGTGT
```

Input Arguments

`object` — Object containing read data
`BioRead` object | `BioMap` object

Object containing the read data, specified as a `BioRead` or `BioMap` object. If the object is not stored in memory, you cannot modify its properties, except the `Name` property.

Example: `readData`

`subsequences` — Partial read sequences
cell array of character vectors | string vector

Partial read sequences, specified as a cell array of character vectors or string vector. Each character vector or string (that is, each sequence) must be the same length.

Example: `{'TGGCTTC','AAAGCAG'}`

`subset` — Subset of elements in object
vector of positive integers | logical vector | string vector | cell array of character vectors

Subset of elements in the object, specified as a vector of positive integers, logical vector, string vector, or cell array of character vectors containing valid sequence headers.

Example: `[1 3]`

Tip When you use a sequence header (or a cell array of headers) for `subset`, a repeated header specifies all elements with that header.

`positions` — Sequence positions
vector of positive integers | logical vector

Sequence positions, specified as a vector of positive integers or logical vector. The number of positions must equal the length of every character vector or string in `subsequences`.

Example: `[1:5]`

Output Arguments

`newObject` — New object with updated properties
`BioRead` object | `BioMap` object

New object with updated properties, returned as a `BioRead` or `BioMap` object.

Version History

Introduced in R2010a

See Also

`BioRead` | `BioMap`

Topics

"Manage Sequence Read Data in Objects"

setSubset

Update elements of object

Syntax

```
newObject = setSubset(object,elements,subset)
```

Description

`newObject = setSubset(object,elements,subset)` returns a new object that is a copy of `object` with a subset of elements set to `elements`. A one-to-one relationship must exist between the number and order of elements in `elements` and `subset`.

Examples

Update Elements of BioRead Object

Construct two BioRead objects, one with 10 elements, and one with 2 elements. Trim the headers to the first white space.

```
struct1 = fastqread('SRR005164_1_50.fastq',...
                    'blockread', [1 10], 'trimheaders', true);
struct2 = fastqread('SRR005164_1_50.fastq',...
                    'blockread', [11 12], 'trimheaders', true);
brObj1  = BioRead(struct1)
```

```
brObj1 =
  BioRead with properties:

     Quality: {10x1 cell}
    Sequence: {10x1 cell}
      Header: {10x1 cell}
       NSeqs: 10
        Name: ''
```

```
brObj2  = BioRead(struct2)
```

```
brObj2 =
  BioRead with properties:

     Quality: {2x1 cell}
    Sequence: {2x1 cell}
      Header: {2x1 cell}
       NSeqs: 2
        Name: ''
```

Replace the first two elements in `brObj1` with the elements in `brObj2`. The object `brObj2` must contain the same number of elements as the number of elements in `subset` (in this case, 2).

```
subset = [1:2];
brObj1 = setSubset(brObj1,brObj2,subset)

brObj1 = 
  BioRead with properties:

     Quality: {10x1 cell}
    Sequence: {10x1 cell}
      Header: {10x1 cell}
       NSeqs: 10
        Name: ''
```

Input Arguments

`object` — Object containing read data
BioRead object | BioMap object

Object containing the read data, specified as a `BioRead` or `BioMap` object. If the object is not stored in memory, you cannot modify its properties, except the `Name` property.

Example: `readData`

`elements` — Object containing information related to read data
BioRead object | BioMap object

Object containing information related to the read data, specified as a `BioRead` or `BioMap` object. The object must contain the same number of elements as the number of elements in `subset`.

Example: `brObject`

`subset` — Subset of elements in object
vector of positive integers | logical vector | string vector | cell array of character vectors

Subset of elements in the object, specified as a vector of positive integers, logical vector, string vector, or cell array of character vectors containing valid sequence headers.

Example: `[1 3]`

Tip When you use a sequence header (or a cell array of headers) for `subset`, a repeated header specifies all elements with that header.

Output Arguments

`newObject` — New object with updated properties
BioRead object | BioMap object

New object with updated properties, returned as a `BioRead` or `BioMap` object.

Version History

Introduced in R2010a

See Also

`BioRead` | `BioMap`

Topics

“Manage Sequence Read Data in Objects”

sffinfo

Return information about SFF file

Syntax

InfoStruct = sffinfo(*File*)

Description

InfoStruct = sffinfo(*File*) returns a MATLAB structure containing summary information about a Standard Flowgram Format (SFF) file.

Input Arguments

File

Character vector or string specifying a file name or path and file name of an SFF file produced by version 1.0 of the Genome Sequencer System data analysis software from 454 Life Sciences. If you specify only a file name, that file must be on the MATLAB search path or in the current folder.

Output Arguments

InfoStruct

MATLAB structure containing summary information about an SFF file. The structure contains the following fields.

Field	Description
Filename	Name of the file.
FileModDate	Modification date of the file.
FileSize	Size of the file in bytes.
Version	Version number of the file.
FlowgramCode	Code of the format used to encode flowgram values.
NumberOfReads	Number of sequence reads in the file.
NumberOfFlowsPerRead	Number of flows for each read.
FlowChars	Bases used in each flow.
KeySequence	Character vector of bases in the key sequence.

Examples

The SFF file, SRR013472.sff, used in this example is not provided with the Bioinformatics Toolbox software. You can download sample SFF files from:

```
https://trace.ncbi.nlm.nih.gov/Traces/index.html
```

Return a summary of the contents of an SFF file:

```
info = sffinfo('SRR013472.sff')

info = 

                Filename: 'SRR013472.sff'
             FileModDate: '23-Feb-2009 15:14:36'
                FileSize: 6632392
                 Version: [0 0 0 1]
            FlowgramCode: 1
           NumberOfReads: 3546
    NumberOfFlowsPerRead: 440
               FlowChars: [1x440 char]
             KeySequence: 'TCAG'
```

Version History

Introduced in R2009b

See Also

`fastqread` | `fastqwrite` | `fastqinfo` | `fastainfo` | `fastaread` | `fastawrite` | `sffread` | `saminfo` | `samread`

External Websites

https://trace.ncbi.nlm.nih.gov/Traces/sra/

sffread

Read data from SFF file

Syntax

```
SFFStruct = sffread(File)

sffread(..., 'Blockread', BlockreadValue, ...)
sffread(..., 'Feature', FeatureValue, ...)
```

Description

`SFFStruct = sffread(File)` reads a Standard Flowgram Format (SFF) file and returns the data in a MATLAB array of structures.

`sffread(..., 'PropertyName', PropertyValue, ...)` calls `sffread` with optional properties that use property name/property value pairs. You can specify one or more properties in any order. Enclose each *`PropertyName`* in single quotation marks. Each *`PropertyName`* is case insensitive. These property name/property value pairs are as follows:

`sffread(..., 'Blockread', BlockreadValue, ...)` reads a single sequence entry or block of sequence entries from an SFF file containing multiple sequences.

`sffread(..., 'Feature', FeatureValue, ...)` specifies the information to include in the return structure.

Input Arguments

`File`

Character vector or string specifying a file name or path and file name of an SFF file produced by version 1.0 of the Genome Sequencer System data analysis software from 454 Life Sciences. If you specify only a file name, that file must be on the MATLAB search path or in the current folder.

Default:

`BlockreadValue`

Scalar or vector that controls the reading of a single sequence entry or block of sequence entries from an SFF file containing multiple sequences. Enter a scalar *`N`*, to read the *`N`*th entry in the file. Enter a 1-by-2 vector [*`M1, M2`*], to read a block of entries starting at the *`M1`* entry and ending at the *`M2`* entry. To read all remaining entries in the file starting at the *`M1`* entry, enter a positive value for *`M1`* and enter `Inf` for *`M2`*.

Default:

`FeatureValue`

Character vector or string specifying the information to include in the output structure. The character vector or string includes letters from the alphabet `H`, `S`, `Q`, `C`, `F`, and `I`, which represent the fields `Header`, `Sequence`, `Quality`, `Clipping`, `FlowgramValue`, and `FlowgramIndex`, respectively.

Default: `'HSQ'`

Output Arguments

`SFFStruct`

Array of structures containing information from an SFF file. There is one structure for each read or entry in the file. Each structure contains one or more of the following fields.

Field	Description
`Header`	Universal accession number.
`Sequence`	Numeric representation of nucleotide sequence.
`Quality`	Per-base quality scores.
`Clipping`	Clipping boundary positions.
`FlowgramValue`	Sequence of flowgram intensity values.
`FlowgramIndex`	Sequence of flowgram intensity indices.

Examples

The SFF file, `SRR013472.sff`, used in these examples is not provided with the Bioinformatics Toolbox software. You can download sample SFF files from:

https://trace.ncbi.nlm.nih.gov/Traces/index.html

Read an entire SFF file:

```
% Read the contents of an entire SFF file into an
% array of structures
reads = sffread('SRR013472.sff')

reads =

3546x1 struct array with fields:
    Header
    Sequence
    Quality
```

Read a block of entries from an SFF file:

```
% Read only the header and sequence information of the
% first five reads from an SFF file into an array of structures
reads5 = sffread('SRR013472.sff', 'block', [1 5], 'feature', 'hs')

reads5 =

5x1 struct array with fields:
    Header
    Sequence
```

Version History

Introduced in R2009b

See Also

`fastqread` | `fastqwrite` | `fastqinfo` | `fastainfo` | `fastaread` | `fastawrite` | `sffinfo` | `saminfo` | `samread`

External Websites

https://trace.ncbi.nlm.nih.gov/Traces/sra/

shortestpath (biograph)

(Removed) Solve shortest path problem in biograph object

Note The function has been removed. Use the `shortestpath` function of `graph` or `digraph` instead.

Syntax

```
[dist, path, pred] = shortestpath(BGObj, S)
[dist, path, pred] = shortestpath(BGObj, S, T)

[...] = shortestpath(..., 'Directed', DirectedValue, ...)
[...] = shortestpath(..., 'Method', MethodValue, ...)
[...] = shortestpath(..., 'Weights', WeightsValue, ...)
```

Arguments

BGObj	Biograph object created by `biograph` (object constructor).
S	Node in graph represented by an N-by-N adjacency matrix extracted from a biograph object, *BGObj*.
T	Node in graph represented by an N-by-N adjacency matrix extracted from a biograph object, *BGObj*.
DirectedValue	Property that indicates whether the graph represented by the N-by-N adjacency matrix extracted from a biograph object, *BGObj*, is directed or undirected. Enter `false` for an undirected graph. This results in the upper triangle of the sparse matrix being ignored. Default is `true`.
MethodValue	Character vector or string that specifies the algorithm used to find the shortest path. Choices are: • `'Bellman-Ford'` — Assumes weights of the edges to be nonzero entries in the N-by-N adjacency matrix. Time complexity is `O(N*E)`, where `N` and `E` are the number of nodes and edges respectively. • `'BFS'` — Breadth-first search. Assumes all weights to be equal, and nonzero entries in the N-by-N adjacency matrix to represent edges. Time complexity is `O(N+E)`, where `N` and `E` are the number of nodes and edges respectively. • `'Acyclic'` — Assumes the graph represented by the N-by-N adjacency matrix extracted from a biograph object, *BGObj*, to be a directed acyclic graph and that weights of the edges are nonzero entries in the N-by-N adjacency matrix. Time complexity is `O(N+E)`, where `N` and `E` are the number of nodes and edges respectively. • `'Dijkstra'` — Default algorithm. Assumes weights of the edges to be positive values in the N-by-N adjacency matrix. Time complexity is `O(log(N)*E)`, where `N` and `E` are the number of nodes and edges respectively.

WeightsValue	Column vector that specifies custom weights for the edges in the N-by-N adjacency matrix extracted from a biograph object, *BGObj*. It must have one entry for every nonzero value (edge) in the N-by-N adjacency matrix. The order of the custom weights in the vector must match the order of the nonzero values in the N-by-N adjacency matrix when it is traversed column-wise. This property lets you use zero-valued weights. By default, `shortestpaths` gets weight information from the nonzero entries in the N-by-N adjacency matrix.

Description

Tip For introductory information on graph theory functions, see "Graph Theory Functions".

`[`*dist*`,` *path*`,` *pred*`] = shortestpath(`*BGObj*`,` *S*`)` determines the single-source shortest paths from node *S* to all other nodes in the graph represented by an N-by-N adjacency matrix extracted from a biograph object, *BGObj*. Weights of the edges are all nonzero entries in the N-by-N adjacency matrix. *dist* are the `N` distances from the source to every node (using `Inf`s for nonreachable nodes and `0` for the source node). *path* contains the winning paths to every node. *pred* contains the predecessor nodes of the winning paths.

`[`*dist*`,` *path*`,` *pred*`] = shortestpath(`*BGObj*`,` *S*`,` *T*`)` determines the single source-single destination shortest path from node *S* to node *T*.

`[...] = shortestpath(..., '`*PropertyName*`',` *PropertyValue*`, ...)` calls `shortestpath` with optional properties that use property name/property value pairs. You can specify one or more properties in any order. Each *PropertyName* must be enclosed in single quotes and is case insensitive. These property name/property value pairs are as follows:

`[...] = shortestpath(..., 'Directed',` *DirectedValue*`, ...)` indicates whether the graph represented by the N-by-N adjacency matrix extracted from a biograph object, *BGObj*, is directed or undirected. Set *DirectedValue* to `false` for an undirected graph. This results in the upper triangle of the sparse matrix being ignored. Default is `true`.

`[...] = shortestpath(..., 'Method',` *MethodValue*`, ...)` lets you specify the algorithm used to find the shortest path. Choices are:

- `'Bellman-Ford'` — Assumes weights of the edges to be nonzero entries in the N-by-N adjacency matrix. Time complexity is `O(N*E)`, where `N` and `E` are the number of nodes and edges respectively.
- `'BFS'` — Breadth-first search. Assumes all weights to be equal, and nonzero entries in the N-by-N adjacency matrix to represent edges. Time complexity is `O(N+E)`, where `N` and `E` are the number of nodes and edges respectively.
- `'Acyclic'` — Assumes the graph represented by the N-by-N adjacency matrix extracted from a biograph object, *BGObj*, to be a directed acyclic graph and that weights of the edges are nonzero entries in the N-by-N adjacency matrix. Time complexity is `O(N+E)`, where `N` and `E` are the number of nodes and edges respectively.
- `'Dijkstra'` — Default algorithm. Assumes weights of the edges to be positive values in the N-by-N adjacency matrix. Time complexity is `O(log(N)*E)`, where `N` and `E` are the number of nodes and edges respectively.

`[...] = shortestpath(..., 'Weights', WeightsValue, ...)` lets you specify custom weights for the edges. *`WeightsValue`* is a column vector having one entry for every nonzero value (edge) in the N-by-N adjacency matrix extracted from a biograph object, *`BGObj`*. The order of the custom weights in the vector must match the order of the nonzero values in the N-by-N adjacency matrix when it is traversed column-wise. This property lets you use zero-valued weights. By default, `shortestpath` gets weight information from the nonzero entries in the N-by-N adjacency matrix.

Version History

Introduced in R2006b

R2022b: Removed
Errors starting in R2022b

The function has been removed. Use the `shortestpath` function of `graph` or `digraph` instead.

R2022a: Warns
Warns starting in R2022a

The function issues a warning that it will be removed in a future release.

R2021b: To be removed
Not recommended starting in R2021b

The function runs without warning, but it will be removed in a future release.

References

[1] Dijkstra, E.W. (1959). A note on two problems in connexion with graphs. Numerische Mathematik *1*, 269–271.

[2] Bellman, R. (1958). On a Routing Problem. Quarterly of Applied Mathematics *16(1)*, 87–90.

[3] Siek, J.G., Lee, L-Q, and Lumsdaine, A. (2002). The Boost Graph Library User Guide and Reference Manual, (Upper Saddle River, NJ:Pearson Education).

See Also

`shortestpath` | `graph` | `digraph`

showhmmprof

Plot hidden Markov model (HMM) profile

Syntax

```
showhmmprof(Model)

showhmmprof(Model, ...'Scale', ScaleValue, ...)
showhmmprof(Model, ...'Order', OrderValue, ...)
```

Arguments

`Model`	Hidden Markov model created by the function `gethmmprof` or `pfamhmmread`.
`ScaleValue`	Property to select a probability scale. Enter one of the following values: • `'logprob'` — Log probabilities • `'prob'` — Probabilities • `'logodds'` — Log-odd ratios
`OrderValue`	Property to specify the order of the amino acid alphabet. Enter a character vector or string with the 20 standard amino acids characters `A R N D C Q E G H I L K M F P S T W Y V`. The ambiguous characters `B Z X` are not allowed.

Description

`showhmmprof(`*`Model`*`)` plots a profile hidden Markov model described by the structure *`Model`*.

`showhmmprof(..., '`*`PropertyName`*`', `*`PropertyValue`*`, ...)` calls `showhmmprof` with optional properties that use property name/property value pairs. You can specify one or more properties in any order. Each *`PropertyName`* must be enclosed in single quotation marks and is case insensitive. These property name/property value pairs are as follows:

`showhmmprof(`*`Model`*`, ...'Scale', `*`ScaleValue`*`, ...)` specifies the scale to use. If log probabilities (*`ScaleValue`*`='logprob'`), probabilities (*`ScaleValue`*`='prob'`), or log-odd ratios (*`ScaleValue`*`='logodds'`). To compute the log-odd ratios, the null model probabilities are used for symbol emission and equally distributed transitions are used for the null transition probabilities. The default *`ScaleValue`* is `'logprob'`.

`showhmmprof(`*`Model`*`, ...'Order', `*`OrderValue`*`, ...)` specifies the order in which the symbols are arranged along the vertical axis. This option allows you reorder the alphabet and group the symbols according to their properties.

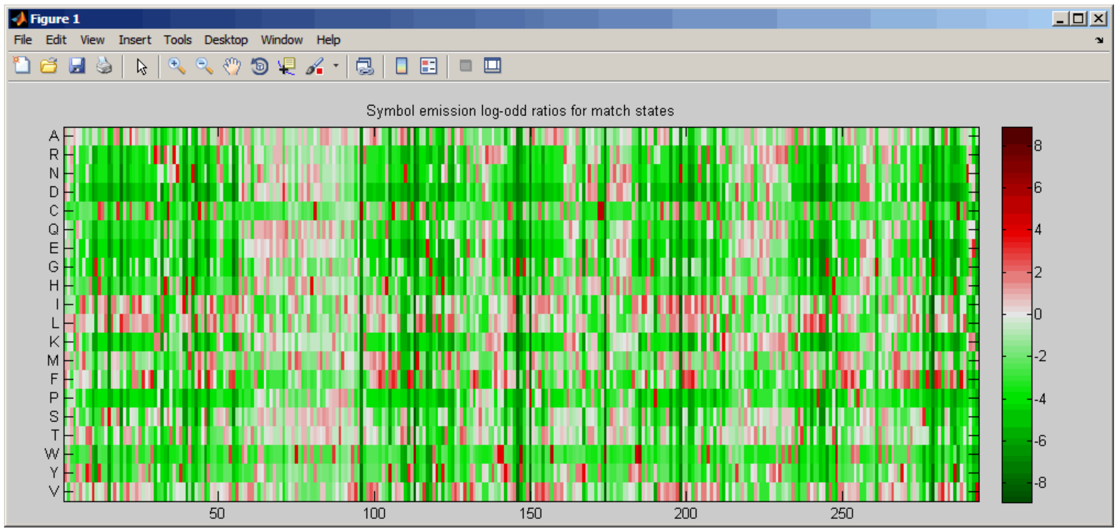
Examples

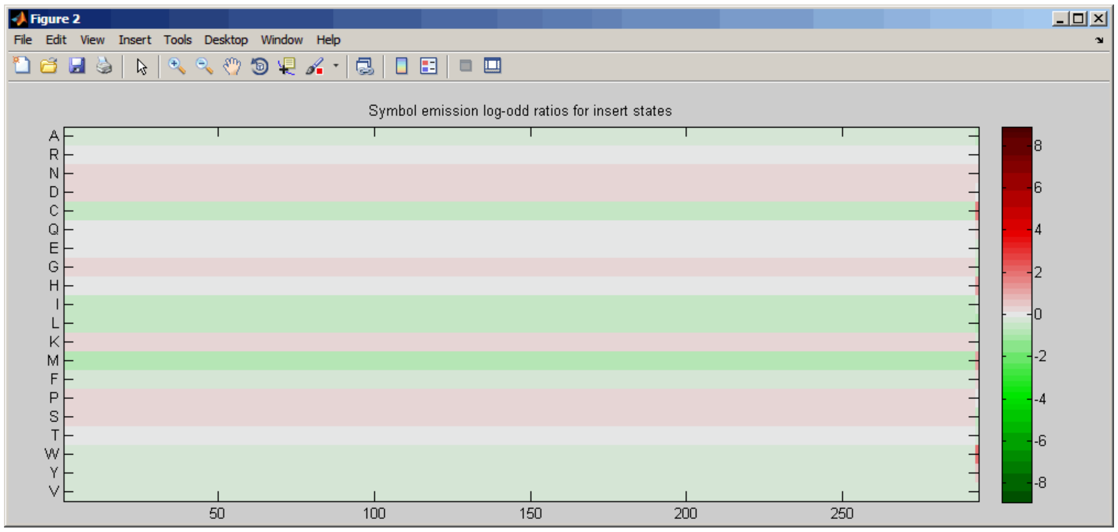
1 Load a model example.

```
model = pfamhmmread('pf00002.ls');
```

2 Plot the profile.

```
showhmmprof(model, 'Scale', 'logodds')
```





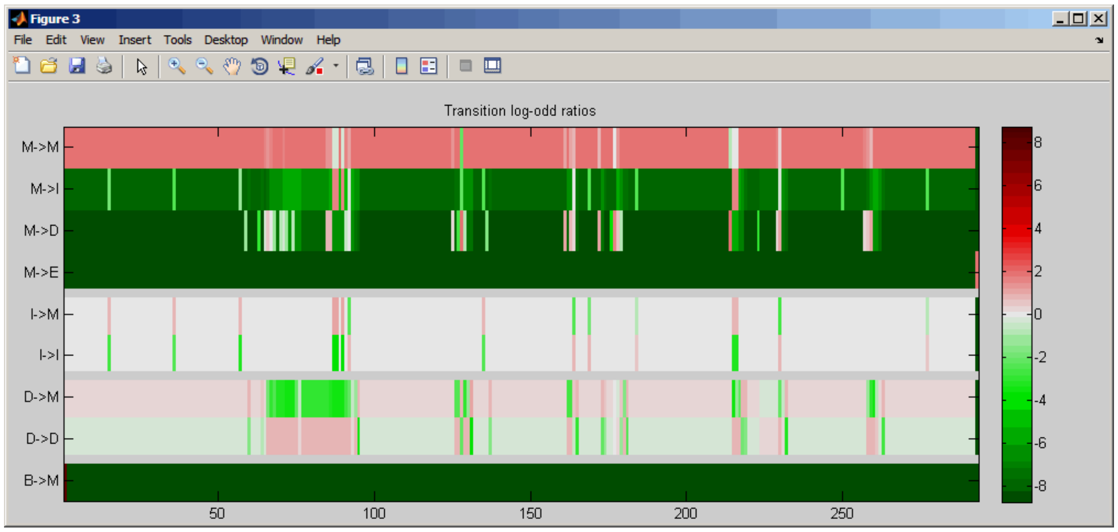


Figure 3
File Edit View Insert Tools Desktop Window Help
Transition log-odd ratios
M->M
M->I
M->D
M->E
I->M
I->I
D->M
D->D
B->M
50
100
150
200
250
8
6
4
2
0
-2
-4
-6
-8

3 Order the alphabet by hydrophobicity.

```
hydrophobic = 'IVLFCMAGTSWYPHNDQEKR';
```

4 Plot the profile.

```
showhmmprof(model, 'Order', hydrophobic)
```

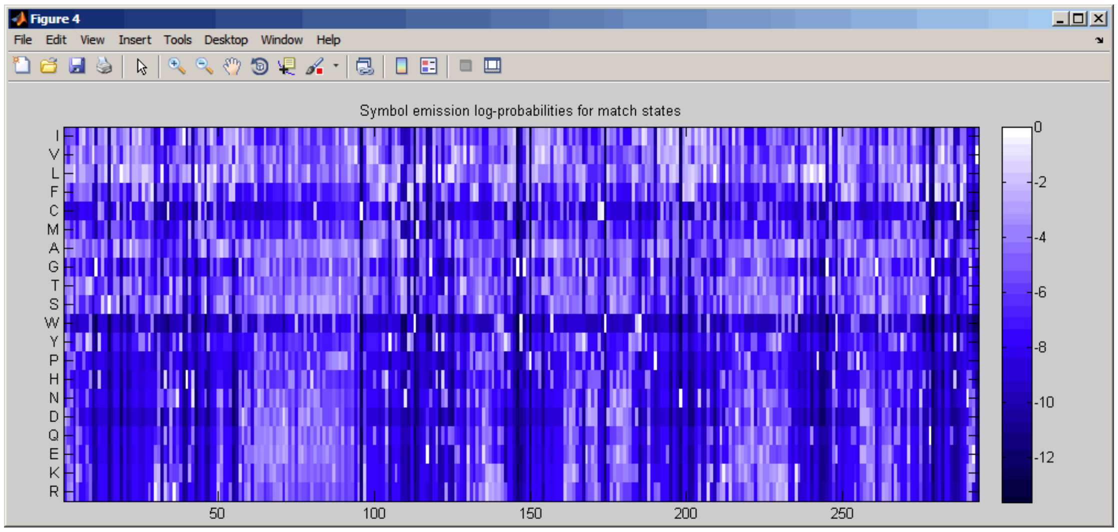
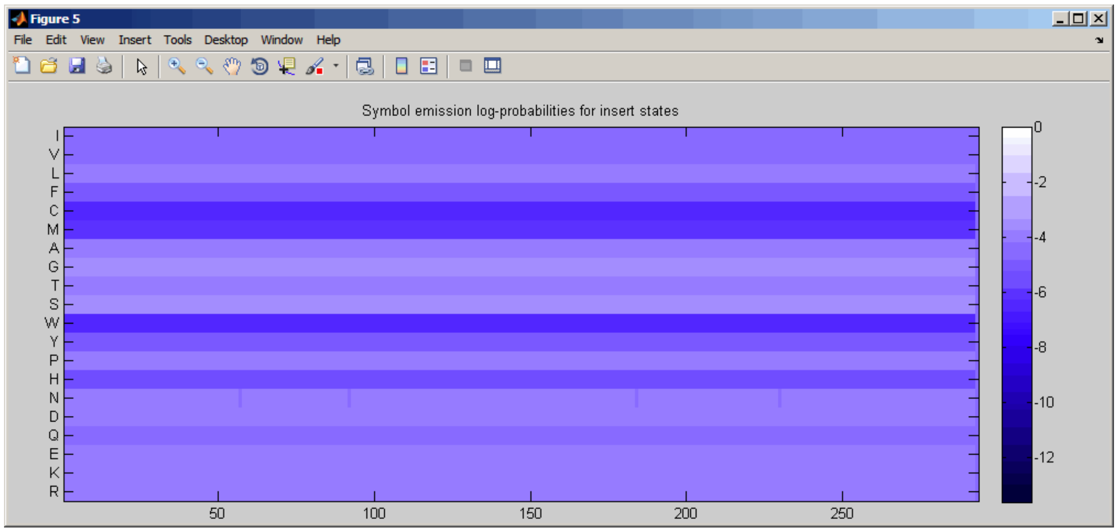
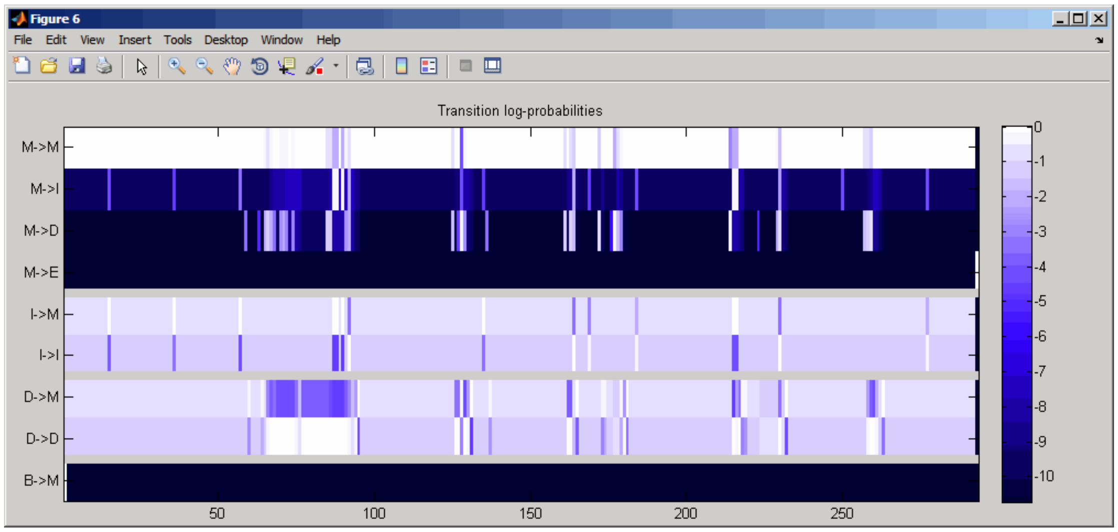


Figure 4
File Edit View Insert Tools Desktop Window Help
Symbol emission log-probabilities for match states
I
V
L
F
C
M
A
G
T
S
W
Y
P
H
N
D
Q
E
K
R
50
100
150
200
250
0
-2
-4
-6
-8
-10
-12





Version History

Introduced before R2006a

See Also

`gethmmprof` | `hmmprofalign` | `hmmprofestimate` | `hmmprofgenerate` | `hmmprofstruct` | `pfamhmmread`

single (DataMatrix)

Convert DataMatrix object to single-precision array

Syntax

```
B = single(DMObj)
B = single(DMObj, Rows)
B = single(DMObj, Rows, Cols)
```

Input Arguments

DMObj	DataMatrix object, such as created by `DataMatrix` (object constructor).
Rows, *Cols*	Row(s) or column(s) in *DMObj*, specified by one of the following: • Scalar • Vector of positive integers • Character vector specifying a row or column name • Cell array of row or column names • Logical vector

Output Arguments

B	MATLAB numeric array.

Description

`B = single(DMObj)` converts *DMObj*, a DataMatrix object, to a single-precision array, which it returns in *B*.

`B = single(DMObj, Rows)` converts a subset of *DMObj*, a DataMatrix object, specified by *Rows*, to a single-precision array, which it returns in *B*. *Rows* can be a positive integer, vector of positive integers, character vector specifying a row name, cell array of row names, or a logical vector.

`B = single(DMObj, Rows, Cols)` converts a subset of *DMObj*, a DataMatrix object, specified by *Rows* and *Cols*, to a single-precision array, which it returns in*B*. *Cols* can be a positive integer, vector of positive integers, character vector specifying a column name, cell array of column names, or a logical vector.

Version History

Introduced in R2008b

See Also

`DataMatrix` | `double`

Topics

DataMatrix object on page 1-734

size

Class: `bioma.ExpressionSet`
Package: `bioma`

Return size of ExpressionSet object

Syntax

```
NFeatSam = size(ESObj)
[NFeatures, NSamples] = size(ESObj)
DimLength = size(ESObj, Dim)
```

Description

`NFeatSam` `=` `size(`*`ESObj`*`)` returns a two-element row vector containing the number of features and number of samples in an ExpressionSet object.

`[`*`NFeatures`*`,` *`NSamples`*`] = size(`*`ESObj`*`)` returns the number of features and number of samples in an ExpressionSet object as separate variables.

`DimLength` `= size(`*`ESObj`*`,` *`Dim`*`)` returns the length of the dimension specified by *`Dim`*.

Input Arguments

`ESObj`

Object of the `bioma.ExpressionSet` class.

Default:

`Dim`

Scalar specifying the dimension of the ExpressionSet object. Choices are:

- `1` — Features
- `2` — Samples

Default:

Examples

Construct an ExpressionSet object, `ESObj`, as described in the “Examples” on page 1-0 section of the `bioma.ExpressionSet` class reference page. Determine the number of features and samples in the ExpressionSet object:

```
% Retrieve the number of features and samples
NumFeatSam = size(ESObj)
```

See Also

`bioma.ExpressionSet` | `bioma.data.ExptData` | `DataMatrix`

Topics
“Managing Gene Expression Data in Objects”

size

Class: `bioma.data.ExptData`
Package: `bioma.data`

Return size of ExptData object

Syntax

```
NFeatSam = size(EDObj)
[NFeatures, NSamples] = size(EDObj)
DimLength = size(EDObj, Dim)
```

Description

`NFeatSam = size(EDObj)` returns a two-element row vector containing the number of features and number of samples in an ExptData object.

`[NFeatures, NSamples] = size(EDObj)` returns the number of features and number of samples in an ExptData object as separate variables.

`DimLength = size(EDObj, Dim)` returns the length of the dimension specified by *`Dim`*.

Input Arguments

`EDObj`

Object of the `bioma.data.ExptData` class.

Default:

`Dim`

Scalar specifying the dimension of the ExptData object. Choices are:

- `1` — Features
- `2` — Samples

Default:

Examples

Construct an ExptData object, and then determine the number of features and samples in it:

```
% Import bioma.data package to make constructor functions
% available
import bioma.data.*
% Create DataMatrix object from .txt file containing
% expression values from microarray experiment
dmObj = DataMatrix('File', 'mouseExprsData.txt');
% Construct ExptData object
```

```
EDObj = ExptData(dmObj);
% Retrieve the number of features and samples
NumFeatSam = size(EDObj)
```

See Also

`bioma.data.ExptData`

Topics

"Representing Expression Data Values in ExptData Objects"

size

Class: `bioma.data.MetaData`
Package: `bioma.data`

Return size of MetaData object

Syntax

```
NSamVar = size(MDObj)
[NSamples, NVariables] = size(MDObj)
DimLength = size(MDObj, Dim)
```

Description

`NSamVar = size(MDObj)` returns a two-element row vector containing the number of samples or features and number of variables in a MetaData object.

`[NSamples, NVariables] = size(MDObj)` returns the number of samples or features and the number of variables in a MetaData object as separate variables.

`DimLength = size(MDObj, Dim)` returns the length of the dimension specified by *`Dim`*.

Input Arguments

`MDObj`

Object of the `bioma.data.MetaData` class.

Default:

`Dim`

Scalar specifying the dimension of the MetaData object. Choices are:

- `1` — Samples
- `2` — Variables

Default:

Examples

Construct a MetaData object, and then determine the number of samples and variables in it:

```
% Import bioma.data package to make constructor function
% available
import bioma.data.*
% Construct MetaData object from .txt file
MDObj2 = MetaData('File', 'mouseSampleData.txt', 'VarDescChar', '#');
% Retrieve the number of samples and variables
NumSamVar = size(MDObj2)
```

See Also

`bioma.data.MetaData`

Topics

"Representing Sample and Feature Metadata in MetaData Objects"

soapread

Read data from Short Oligonucleotide Analysis Package (SOAP) file

Syntax

```
SOAPStruct = soapread(File)
SOAPStruct = soapread(File,Name,Value)
```

Description

`SOAPStruct = soapread(File)` reads `File`, a SOAP-formatted file (version 2.15) and returns the data in `SOAPStruct`, a MATLAB array of structures.

`SOAPStruct = soapread(File,Name,Value)` specifies additional options using one or more name-value arguments. For example, to read entry 10 of the file, `SOAPStruct = soapread(File,BlockRead=10)`.

Examples

Read and Examine a SOAP-Formatted File

Read the alignment records (entries) from the `sample01.soap` file.

```
data = soapread("sample01.soap")
```

```
data=17×1 struct array with fields:
    QueryName
    Sequence
    Quality
    NumHits
    PairedEndSourceFile
    Length
    Strand
    ReferenceName
    Position
    AlignDetails
```

View the quality score for the 6th entry.

```
data(6).Quality
```

```
ans = 
'<>.>>>8>;:1>>>3>6>'
```

Determine the strand direction (forward or reverse) of the reference sequence to which the 12th entry aligns

```
data(12).Strand
```

```
ans = 
'-'
```

Modify SOAP-File Reading

Read a block of six alignment records (entries) from the `sample01.soap` file.

```
data_5_10 = soapread('sample01.soap',BlockRead=[5 10])
```

```
data_5_10=6×1 struct array with fields:
    QueryName
    Sequence
    Quality
    NumHits
    PairedEndSourceFile
    Length
    Strand
    ReferenceName
    Position
    AlignDetails
```

Input Arguments

`File` — File to read

file path | file name

File to read, specified as a path to a SOAP-formatted file (version 2.15) or as a file name. If you specify only a file name, that file must be on the MATLAB search path or in the current folder.

Data Types: `char` | `string`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

The names are case-insensitive. For example, you can use `aligndetails` instead of `AlignDetails`.

Example: `SOAPStruct = soapread(File,BlockRead=10)`

`AlignDetails` — Indication to include the `AlignDetails` field in the `SOAPStruct` output argument

`true` (default) | `false`

Indication to include the `AlignDetails` field in the `SOAPStruct` output argument, specified as `true` (include the field) or `false` (do not include the field).

Example: `false`

Data Types: `logical`

`BlockRead` — Entries to read

`[1 Inf]` (default) | positive integer | two-element positive integer vector

Entries to read, specified as a positive integer or as a two-element positive integer vector.

- To read entry `N` in `File`, specify a positive integer `N`.
- To read the block of entries starting at `N1` and ending at `N2`, specify a positive integer vector `[N1 N2]` with `N1 < N2`. To read all the entries starting at `N1`, specify `Inf` for `N2`.

Example: `[10,19]`

Data Types: `single` | `double`

Output Arguments

`SOAPStruct` — Sequence alignment and mapping information
array of structures

Sequence alignment and mapping information, returned as an *N*-by-1 array of structures, where *N* is the number of alignment records stored in `File`. Each structure contains the following fields.

Field	Description
`QueryName`	Name of aligned read sequence.
`Sequence`	Character vector containing the letter representations of the read sequence. It is the reverse-complement if the read sequence aligns to the reverse strand of the reference sequence.
`Quality`	Character vector containing the ASCII representation of the per-base quality score for the read sequence. The quality score is reversed if the read sequence aligns to the reverse strand of the reference sequence.
`NumHits`	The number of *total* instances where this read sequence aligned to an identical length of bases on another area of the reference sequence.
`PairedEndSourceFile`	Flag (a or b) specifying which source file to which the read sequence belongs. This field applies only to read sequences that are paired in the alignment.
`Length`	Scalar specifying the length of the read sequence.
`Strand`	+ or − specifying direction (forward or reverse) of reference sequence to which the read sequence aligns.
`ReferenceName`	Name or numeric ID of the reference sequence to which the read sequence aligns.
`Position`	Position (one-based offset) of the forward reference sequence where the left-most base of the alignment of the read sequence starts.
`AlignDetails`	Information on mismatches, insertions, and deletions in the alignment. For SOAP-formatted files v2.15, this field includes CIGAR strings.

Tips

If your SOAP-formatted file is too large to read using available memory, try either of the following:

- Use the `BlockRead` name-value pair arguments to read a subset of entries.
- Create a `BioIndexedFile` object from the SOAP-formatted file (using `'TABLE'` for the `Format`), and then access the entries using methods of the `BioIndexedFile` class.

Version History

Introduced in R2010b

References

[1] Li, R., Yu, C., Li, Y., Lam, T., Yiu, S., Kristiansen, K., and Wang, J. (2009). SOAP2: an improved ultrafast tool for short read alignment. Bioinformatics *25, 15*, 1966–1967.

[2] Li, R., Li, Y., Kristiansen, K., and Wang, J. (2008). SOAP: short oligonucleotide alignment program. Bioinformatics *24(5)*, 713–714.

See Also

`samread` | `bamread` | `fastqread`

Topics

"Work with Next-Generation Sequencing Data"

External Websites

Sequence Read Archive

sortcols (DataMatrix)

Sort columns of DataMatrix object in ascending or descending order

Syntax

```
DMObjNew = sortcols(DMObj1)
DMObjNew = sortcols(DMObj1, Row)
DMObjNew = sortcols(DMObj1, 'ColName')
DMObjNew = sortcols(DMObj1, ..., Mode)
[DMObjNew, Indices] = sortcols(DMObj1, ...)
```

Input Arguments

DMObj1	DataMatrix object, such as created by `DataMatrix` (object constructor).
Row	One or more rows in *DMObj1* by which to sort the columns. Choices are: • Positive integer • Vector of positive integers • Character vector or string specifying a row name • Cell array of character vectors or string vector specifying multiple row names • Logical vector
`'ColName'`	Character vector or string that specifies to sort the columns by the column names.
Mode	Character vector or string specifying the order by which to sort the columns. Choices are `'ascend'` (default) or `'descend'`.

Output Arguments

DMObjNew	DataMatrix object created from sorting the columns of another DataMatrix object.
Indices	Index vector that links *DMObj1* to *DMObjNew*. In other words, *DMObjNew* = *DMObj1*`(:,idx)`.

Description

DMObjNew `= sortcols(`*DMObj1*`)` sorts the columns in *DMObj1* in ascending order based on the elements in the first row. For any columns that have equal elements in a row, sorting is based on the row immediately below.

DMObjNew `= sortcols(`*DMObj1*`,` *Row*`)` sorts the columns in *DMObj1* in ascending order based on the elements in the specified row. Any columns that have equal elements in the specified row are sorted based on the elements in the next specified row.

DMObjNew `= sortcols(`*DMObj1*`, 'ColName')` sorts the columns in *DMObj1* in ascending order according to the column names.

`DMObjNew = sortcols(DMObj1, ..., Mode)` specifies the order of the sort. `Mode` can be `'ascend'` (default) or `'descend'`.

`[DMObjNew, Indices] = sortcols(DMObj1, ...)` returns `Indices`, an index vector that links `DMObj1` to `DMObjNew`. In other words, `DMObjNew = DMObj1(:,idx)`.

Version History

Introduced in R2008b

See Also

`DataMatrix` | `sortrows`

Topics

DataMatrix object on page 1-734

sortrows (DataMatrix)

Sort rows of DataMatrix object in ascending or descending order

Syntax

```
DMObjNew = sortrows(DMObj1)
DMObjNew = sortrows(DMObj1, Column)
DMObjNew = sortrows(DMObj1, 'RowName')
DMObjNew = sortrows(DMObj1, ..., Mode)
[DMObjNew, Indices] = sortrows(DMObj1, ...)
```

Input Arguments

`DMObj1`	DataMatrix object, such as created by `DataMatrix` (object constructor).
`Column`	One or more columns in *`DMObj1`* by which to sort the rows. Choices are: • Positive integer • Vector of positive integers • Character vector or string specifying a column name • Cell array of character vectors or string vector specifying multiple column names • Logical vector
`'RowName'`	Character vector or string that specifies to sort the rows by the row names.
`Mode`	Character vector or string specifying the order by which to sort the rows. Choices are `'ascend'` (default) or `'descend'`.

Output Arguments

`DMObjNew`	DataMatrix object created from sorting the rows of another DataMatrix object.
`Indices`	Index vector that links *`DMObj1`* to *`DMObjNew`*. In other words, *`DMObjNew`* `=` *`DMObj1`*`(idx,:)`.

Description

`DMObjNew` `= sortrows(`*`DMObj1`*`)` sorts the rows in *`DMObj1`* in ascending order based on the elements in the first column. For any rows that have equal elements in a column, sorting is based on the column immediately to the right.

`DMObjNew` `= sortrows(`*`DMObj1`*`,` *`Column`*`)` sorts the rows in *`DMObj1`* in ascending order based on the elements in the specified column. Any rows that have equal elements in the specified column are sorted based on the elements in the next specified column.

`DMObjNew` `= sortrows(`*`DMObj1`*`, 'RowName')` sorts the rows in *`DMObj1`* in ascending order according to the row names.

DMObjNew = sortrows(*DMObj1*, ..., *Mode*) specifies the order of the sort. *Mode* can be 'ascend' (default) or 'descend'.

[*DMObjNew*, *Indices*] = sortrows(*DMObj1*, ...) returns *Indices*, an index vector that links *DMObj1* to *DMObjNew*. In other words, *DMObjNew* = *DMObj1*(idx,:).

Version History

Introduced in R2008b

See Also

DataMatrix | sortcols

Topics

DataMatrix object on page 1-734

sptread

Read data from SPOT file

Syntax

SPOTData = sptread(*File*)

SPOTData = sptread(*File*, 'CleanColNames', *CleanColNamesValue*)

Arguments

File	Character vector or string specifying a file name, a path and file name, or a URL pointing to a file. The referenced file is a SPOT-formatted file (ASCII text file). If you specify only a file name, that file must be on the MATLAB search path or in the MATLAB Current Folder.
CleanColNamesValue	Controls the use of valid MATLAB variable names.

Description

SPOTData = sptread(*File*) reads *File*, a SPOT-formatted file, and creates *SPOTData*, a MATLAB structure containing the following fields:

```
Header
Data
Blocks
Columns
Rows
IDs
ColumnNames
Indices
Shape
```

SPOTData = sptread(*File*, 'CleanColNames', *CleanColNamesValue*) controls the use of valid MATLAB variable names. The column names in the SPOT-formatted file contain periods and some characters that cannot be used in MATLAB variable names. If you plan to use the column names as variable names in a function, use this option with `CleanColNames` set to `true` and the function will return the field `ColumnNames` with valid variable names.

The `Indices` field of the structure includes the indices that you can use for plotting heat maps of the data.

Examples

1 Read in a sample SPOT file and plot the median foreground intensity for the 635 nm channel. Note that the example file `spotdata.txt` is not provided with the Bioinformatics Toolbox software.

```
spotStruct = sptread('spotdata.txt')
maimage(spotStruct,'Rmedian');
```

2 Alternately, create a similar plot using more basic graphics commands.

```
Rmedian = magetfield(spotStruct,'Rmedian');
imagesc(Rmedian(spotStruct.Indices));
colormap bone
colorbar
```

Version History

Introduced before R2006a

See Also

affyread | agferead | celintensityread | geoseriesread | geosoftread | gprread | ilmnbsread | imageneread | maboxplot | magetfield

std (DataMatrix)

Return standard deviation values in DataMatrix object

Syntax

```
S = std(DMObj)
S = std(DMObj, Flag)
S = std(DMObj, Flag, Dim)
S = std(DMObj, Flag, Dim, IgnoreNaN)
```

Input Arguments

DMObj	DataMatrix object, such as created by `DataMatrix` (object constructor).
Flag	Scalar specifying how to normalize the data. Choices are: • `0` — Default. Normalizes using a sample size of *N* – `1`, unless *N* = `1`, in which case, normalizes using a sample size of `1`. • `1` — Normalizes using a sample size of *N*. *N* = the number of elements in each column or row, as specified by *`Dim`*. For more information on the normalization equations, see the function `std`.
Dim	Scalar specifying the dimension of *`DMObj`* to calculate the standard deviations. Choices are: • `1` — Default. Returns standard deviation values for elements in each column. • `2` — Returns standard deviation values for elements in each row.
IgnoreNaN	Specifies if NaNs should be ignored. Choices are `true` (default) or `false`.

Output Arguments

S	Either of the following: • Row vector containing the standard deviation values from elements in each column in *`DMObj`* (when *`Dim`* = `1`) • Column vector containing the standard deviation values from elements in each row in *`DMObj`* (when *`Dim`* = 2)

Description

`S = std(DMObj)` returns the standard deviation values of the elements in the columns of a DataMatrix object, treating NaNs as missing values. The data is normalized using a sample size of *N* – `1`, where *N* = the number of elements in each column. *`S`* is a row vector containing the standard deviation values for elements in each column in *`DMObj`*.

`S = std(DMObj, Flag)` specifies how to normalize the data. If `Flag` = `0`, normalizes using a sample size of *N* – `1`. If `Flag` = `1`, normalizes using a sample size of *N*. *N* = the number of elements in each column or row, as specified by `Dim`. For more information on the normalization equations, see the function `std`. Default `Flag` = `0`.

`S = std(DMObj, Flag, Dim)` returns the standard deviation values of the elements in the columns or rows of a DataMatrix object, as specified by `Dim`. If `Dim` = `1`, returns `S`, a row vector containing the standard deviation values for elements in each column in `DMObj`. If `Dim` = `2`, returns `S`, a column vector containing the standard deviation values for elements in each row in `DMObj`. Default `Dim` = `1`.

`S = std(DMObj, Flag, Dim, IgnoreNaN)` specifies if NaNs should be ignored. `IgnoreNaN` can be `true` (default) or `false`.

Version History

Introduced in R2008b

See Also

`DataMatrix` | `mean` | `median` | `var`

Topics

DataMatrix object on page 1-734

subtree (phytree)

Extract phylogenetic subtree

Syntax

Tree2 = subtree(*Tree1*, *Nodes*)

Description

Tree2 = subtree(*Tree1*, *Nodes*) extracts a new subtree (*Tree2*) where the new root is the first common ancestor of the *Nodes* vector from *Tree1*. Nodes in the tree are indexed as `[1:NUMLEAVES]` for the leaves and as `[NUMLEAVES+1:NUMLEAVES+NUMBRANCHES]` for the branches. Nodes can also be a logical array of following sizes `[NUMLEAVES+NUMBRANCHES x 1]`, `[NUMLEAVES x 1]` or `[NUMBRANCHES x 1]`.

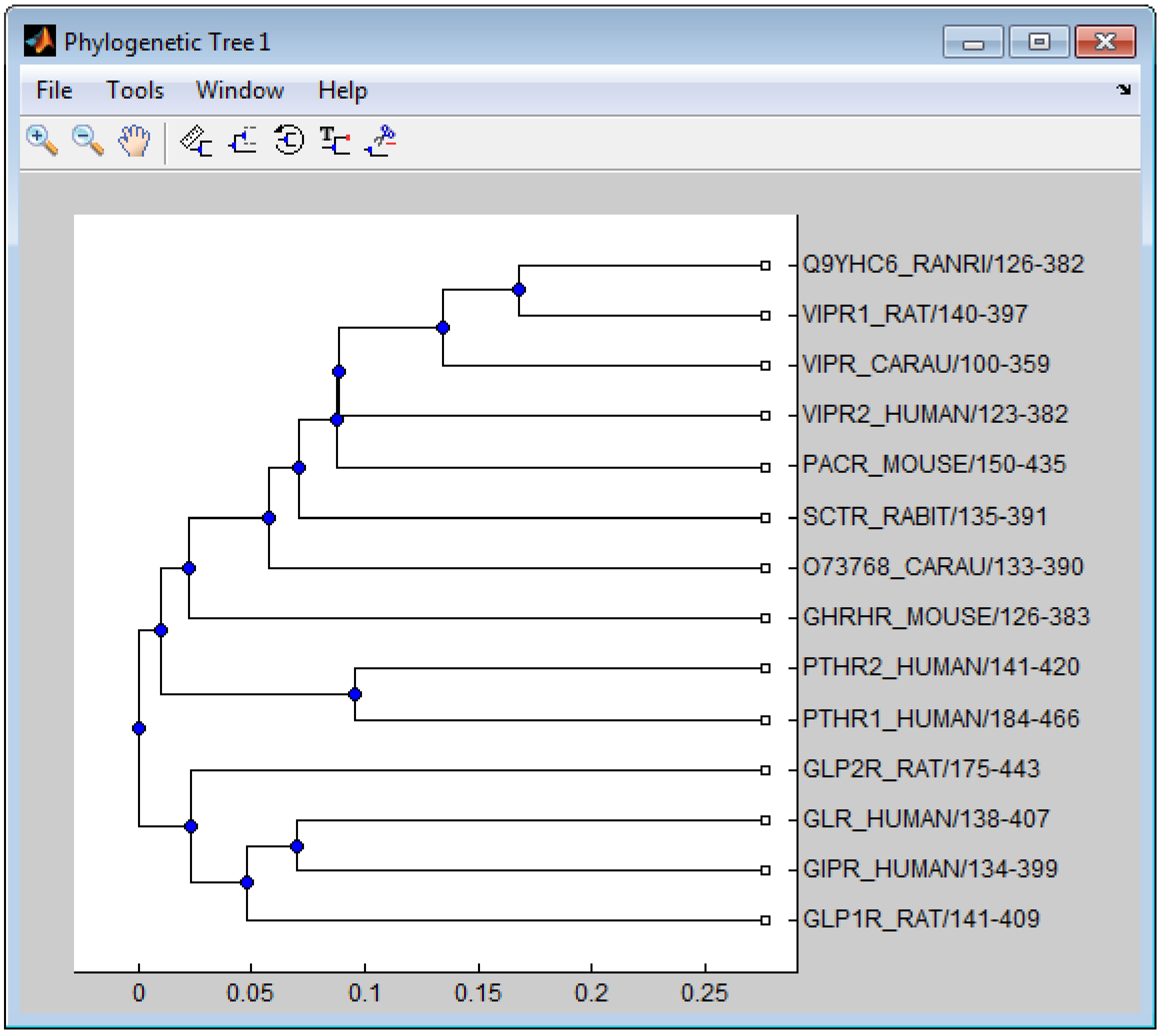
Examples

1 Load a phylogenetic tree created from a protein family.

```
tr = phytreeread('pf00002.tree');
```

2 Get the subtree that contains the VIPR2 and GLR human proteins.

```
sel = getbyname(tr,{'vipr2_human','glr_human'});
sel = any(sel,2);
tr =  subtree(tr,sel);
view(tr)
```



Version History

Introduced before R2006a

See Also

`phytree` | `get` | `getbyname` | `prune` | `select`

Topics

phytree object on page 1-1449

sum (DataMatrix)

Return sum of elements in DataMatrix object

Syntax

```
S = sum(DMObj)
S = sum(DMObj, Dim)
S = sum(DMObj, Dim, IgnoreNaN)
```

Input Arguments

DMObj	DataMatrix object, such as created by `DataMatrix` (object constructor).
Dim	Scalar specifying the dimension of *DMObj* to calculate the sums. Choices are: • `1` — Default. Returns sum of elements in each column. • `2` — Returns sum of elements in each row.
IgnoreNaN	Specifies if NaNs should be ignored. Choices are `true` (default) or `false`.

Output Arguments

S	Either of the following: • Row vector containing the sums of the elements in each column in *DMObj* (when *Dim* = `1`) • Column vector containing the sums of the elements in each row in *DMObj* (when *Dim* = `2`)

Description

`S = sum(DMObj)` returns the sum of the elements in the columns of a DataMatrix object, treating NaNs as missing values. *S* is a row vector containing the sums of the elements in each column in *DMObj*. If the values in *DMObj* are `single`s, then *S* is a `single`; otherwise, *S* is a `double`.

`S = sum(DMObj, Dim)` returns the sum of the elements in the columns or rows of a DataMatrix object, as specified by *Dim*. If *Dim* = `1`, returns *S*, a row vector containing the sums of the elements in each column in *DMObj*. If *Dim* = `2`, returns *S*, a column vector containing the sums of the elements in each row in *DMObj*. Default *Dim* = `1`.

`S = sum(DMObj, Dim, IgnoreNaN)` specifies if NaNs should be ignored. *IgnoreNaN* can be `true` (default) or `false`.

Version History

Introduced in R2008b

See Also

`DataMatrix` | `max` | `min`

Topics

DataMatrix object on page 1-734

swalign

Locally align two sequences using Smith-Waterman algorithm

Syntax

```
Score = swalign(Seq1, Seq2)
[Score, Alignment] = swalign(Seq1, Seq2)
[Score, Alignment, Start] = swalign(Seq1, Seq2)

... = swalign(Seq1,Seq2, ...'Alphabet', AlphabetValue)
... = swalign(Seq1,Seq2, ...'ScoringMatrix', ScoringMatrixValue, ...)
... = swalign(Seq1,Seq2, ...'Scale', ScaleValue, ...)
... = swalign(Seq1,Seq2, ...'GapOpen', GapOpenValue, ...)
... = swalign(Seq1,Seq2, ...'ExtendGap', ExtendGapValue, ...)
... = swalign(Seq1,Seq2, ...'Showscore', ShowscoreValue, ...)
```

Input Arguments

Seq1, *Seq2*	Amino acid or nucleotide sequences. Enter any of the following: • Character vector or string of letters representing amino acids or nucleotides, such as returned by `int2aa` or `int2nt` • Vector of integers representing amino acids or nucleotides, such as returned by `aa2int` or `nt2int` • Structure containing a `Sequence` field **Tip** For help with letter and integer representations of amino acids and nucleotides, see Amino Acid Lookup or Nucleotide Lookup.
AlphabetValue	Character vector or string specifying the type of sequence. Choices are `'AA'` (default) or `'NT'`.

ScoringMatrixValue	Either of the following: • Character vector or string specifying the scoring matrix to use for the local alignment. Choices for amino acid sequences are: • `'BLOSUM62'` • `'BLOSUM30'` increasing by `5` up to `'BLOSUM90'` • `'BLOSUM100'` • `'PAM10'` increasing by `10` up to `'PAM500'` • `'DAYHOFF'` • `'GONNET'` Default is: • `'BLOSUM50'` — When *AlphabetValue* equals `'AA'` • `'NUC44'` — When *AlphabetValue* equals `'NT'` **Note** The above scoring matrices, provided with the software, also include a structure containing a scale factor that converts the units of the output score to bits. You can also use the `'Scale'` property to specify an additional scale factor to convert the output score from bits to another unit. • Matrix representing the scoring matrix to use for the local alignment, such as returned by the `blosum`, `pam`, `dayhoff`, `gonnet`, or `nuc44` function. **Note** If you use a scoring matrix that you created or was created by one of the above functions, the matrix does not include a scale factor. The output score will be returned in the same units as the scoring matrix. You can use the `'Scale'` property to specify a scale factor to convert the output score to another unit. **Note** If you need to compile `swalign` into a stand-alone application or software component using MATLAB Compiler, use a matrix instead of a character vector or string for *ScoringMatrixValue*.

`ScaleValue`	Positive value that specifies a scale factor that is applied to the output score. For example, if the output score is initially determined in bits, and you enter `log(2)` for *`ScaleValue`*, then `swalign` returns *`Score`* in nats. Default is `1`, which does not change the units of the output score. **Note** If the `'ScoringMatrix'` property also specifies a scale factor, then `swalign` uses it first to scale the output score, then applies the scale factor specified by *`ScaleValue`* to rescale the output score. **Tip** Before comparing alignment scores from multiple alignments, ensure the scores are in the same units. You can use the `'Scale'` property to control the units of the output scores.
`GapOpenValue`	Positive value specifying the penalty for opening a gap in the alignment. Default is `8`.
`ExtendGapValue`	Positive value specifying the penalty for extending a gap using the affine gap penalty scheme. **Note** If you specify this value, `swalign` uses the affine gap penalty scheme, that is, it scores the first gap using the *`GapOpenValue`* and scores subsequent gaps using the *`ExtendGapValue`*. If you do not specify this value, `swalign` scores all gaps equally, using the *`GapOpenValue`* penalty.
`ShowscoreValue`	Controls the display of the scoring space and the winning path of the alignment. Choices are `true` or `false` (default).

Output Arguments

`Score`	Optimal local alignment score in bits.
`Alignment`	3-by-N character array showing the two sequences, *`Seq1`* and *`Seq2`*, in the first and third rows, and symbols representing the optimal local alignment between them in the second row.
`Start`	2-by-1 vector of indices indicating the starting point in each sequence for the alignment.

Description

`Score` `= swalign(`*`Seq1`*`, `*`Seq2`*`)` returns the optimal local alignment score in bits. The scale factor used to calculate the score is provided by the scoring matrix.

`[`*`Score`*`, `*`Alignment`*`] = swalign(`*`Seq1`*`, `*`Seq2`*`)` returns a 3-by-N character array showing the two sequences, *`Seq1`* and *`Seq2`*, in the first and third rows, and symbols representing the optimal local alignment between them in the second row. The symbol `|` indicates amino acids or nucleotides that match exactly. The symbol `:` indicates amino acids or nucleotides that are related as defined by the scoring matrix (nonmatches with a zero or positive scoring matrix value).

`[`*`Score, Alignment, Start`*`] = swalign(`*`Seq1, Seq2`*`)` returns a 2-by-1 vector of indices indicating the starting point in each sequence for the alignment.

`... = swalign(`*`Seq1`*`,`*`Seq2`*`, ...'`*`PropertyName`*`', `*`PropertyValue`*`, ...)` calls `swalign` with optional properties that use property name/property value pairs. You can specify one or more properties in any order. Each *`PropertyName`* must be enclosed in single quotation marks and is case insensitive. These property name/property value pairs are as follows:

`... = swalign(`*`Seq1`*`,`*`Seq2`*`, ...'Alphabet', `*`AlphabetValue`*`)` specifies the type of sequences. Choices are `'AA'` (default) or `'NT'`.

`... = swalign(`*`Seq1`*`,`*`Seq2`*`, ...'ScoringMatrix', `*`ScoringMatrixValue`*`, ...)` specifies the scoring matrix to use for the local alignment. Default is:

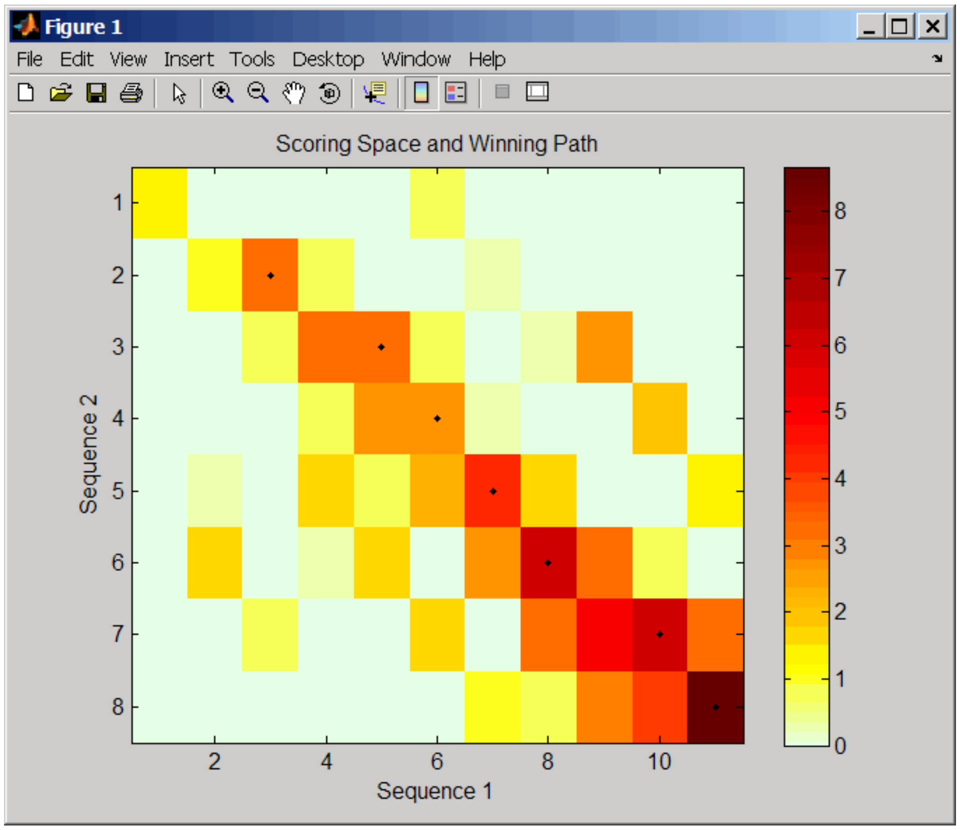
- `'BLOSUM50'` — When *`AlphabetValue`* equals `'AA'`
- `'NUC44'` — When *`AlphabetValue`* equals `'NT'`

`... = swalign(`*`Seq1`*`,`*`Seq2`*`, ...'Scale', `*`ScaleValue`*`, ...)` specifies a scale factor that is applied to the output score, thereby controlling the units of the output score. Choices are any positive value.

`... = swalign(`*`Seq1`*`,`*`Seq2`*`, ...'GapOpen', `*`GapOpenValue`*`, ...)` specifies the penalty for opening a gap in the alignment. Choices are any positive value. Default is `8`.

`... = swalign(`*`Seq1`*`,`*`Seq2`*`, ...'ExtendGap', `*`ExtendGapValue`*`, ...)` specifies the penalty for extending a gap using the affine gap penalty scheme. Choices are any positive value.

`... = swalign(`*`Seq1`*`,`*`Seq2`*`, ...'Showscore', `*`ShowscoreValue`*`, ...)` controls the display of the scoring space and winning path of the alignment. Choices are `true` or `false` (default).



The scoring space is a heat map displaying the best scores for all the partial alignments of two sequences. The color of each (`n1,n2`) coordinate in the scoring space represents the best score for the pairing of subsequences `Seq1(s1:n1)` and `Seq2(s2:n2)`, where `n1` is a position in `Seq1`, `n2` is a position in `Seq2`, `s1` is any position in `Seq1` between `1:n1`, and `s2` is any position in `Seq2` between `1:n2`. The best score for a pairing of specific subsequences is determined by scoring all possible alignments of the subsequences by summing matches and gap penalties.

The winning path is represented by black dots in the scoring space, and it illustrates the pairing of positions in the optimal local alignment. The color of the last point (lower right) of the winning path represents the optimal local alignment score for the two sequences and is the *`Score`* output returned by `swalign`.

Note The scoring space visually shows tandem repeats, small segments that potentially align, and partial alignments of domains from rearranged sequences.

Examples

1 Locally align two amino acid sequences using the `BLOSUM50` (default) scoring matrix and the default values for the `GapOpen` and `ExtendGap` properties. Return the optimal local alignment score in bits and the alignment character array.

```
[Score, Alignment] = swalign('VSPAGMASGYD','IPGKASYD')

Score =
     8.6667

Alignment =

PAGMASGYD
| | || ||
P-GKAS-YD
```

2 Locally align two amino acid sequences specifying the `PAM250` scoring matrix and a gap open penalty of `5`.

```
[Score, Alignment] = swalign('HEAGAWGHEE','PAWHEAE',...
                             'ScoringMatrix', 'pam250',...
                             'GapOpen',5)

Score =

     8
Alignment =

GAWGHE
:|| ||
PAW-HE
```

3 Locally align two amino acid sequences returning the *`Score`* in nat units (nats) by specifying a scale factor of `log(2)`.

```
[Score, Alignment] = swalign('HEAGAWGHEE','PAWHEAE','Scale',log(2))

Score =

    6.4694

Alignment =

AWGHE
|| ||
AW-HE
```

Version History

Introduced before R2006a

References

[1] Durbin, R., Eddy, S., Krogh, A., and Mitchison, G. (1998). Biological Sequence Analysis (Cambridge University Press).

[2] Smith, T., and Waterman, M. (1981). Identification of common molecular subsequences. Journal of Molecular Biology *147*, 195–197.

See Also

`aa2int` | `aminolookup` | `baselookup` | `blosum` | `dayhoff` | `gonnet` | `int2aa` | `int2nt` | `localalign` | `multialign` | `nt2aa` | `nt2int` | `nuc44` | `nwalign` | `pam` | `pdbsuperpose` | `seqdotplot`

term

Data structure containing information about Gene Ontology (GO) term

Description

A `term` object is a data structure containing information about a Gene Ontology (GO) term. You can explore and traverse Gene Ontology terms using "is_a" and "part_of" relationships.

A `term` object has handle copy semantics. To learn how this affects your use of the class, see "Copying Objects" in the MATLAB Programming Fundamentals documentation.

Creation

A `term` object is part of the `terms` property of a `Gene Ontology` object. Create a `Gene Ontology` object using the `geneont` command.

Properties

`definition` — Definition of the GO term
character vector

This property is read-only.

Definition of the GO term. specified as a character vector.

Tip If you know the GO identifier (for example, 314) of a term object, instead of its index or position number (for example, 287), you can use the following syntax to display the definition of a term object:

```
GeneontObj(314).terms.definition
```

For an example, see "Investigate Gene Ontology Term Properties" on page 1-1844.

Data Types: `char`

`id` — GO identifier of GO term
numeric scalar

This property is read-only.

GO identifier of GO term, specified as a numeric scalar.

Tip You can use the `num2goid` function to convert `id` to a GO ID character vector formatted as a 7-digit number preceded by the prefix `GO:`, which is the standard used by the Gene Ontology database.

Tip You can use the `id` property for a GO term as input to methods of a `geneont` object, such as `getancestors`, `getdescendants`, and `getrelatives`.

Data Types: `single` | `double`

`is_a` — GO identifiers of GO terms that have an “is a” relationship with this GO term
numeric array

This property is read-only.

GO identifiers of GO terms that have an “is a” relationship with this GO term, specified as a numeric array.

Tip You can also use the `getancestors` method of a geneont object with the `'Relationtype'` property set to `'is_a'` to determine term objects with an “is a” relationship.

Tip If you know the GO identifier (for example, 42321) of a term object, instead of its index or position number (for example, 18703), you can use the following syntax to display the `is_a` property of a term object:

```
GeneontObj(42321).terms.is_a
```

For an example, see “Investigate Gene Ontology Term Properties” on page 1-1844.

Data Types: `single` | `double`

`name` — Name of GO term
character vector

This property is read-only.

Name of GO term, specified as a character vector.

Data Types: `char` | `string`

`obsolete` — Indication that term is obsolete
logical value

This property is read-only.

Indication that term is obsolete, specified as a logical value.

Tip If you know the GO identifier (for example, 8) of a term object, instead of its index or position number (for example, 7), you can use the following syntax to display the obsolete status of a term object:

```
GeneontObj(8).terms.obsolete
```

For an example, see “Investigate Gene Ontology Term Properties” on page 1-1844.

Data Types: `logical`

`ontology` — Ontology of GO term
`'molecular function'` | `'biological process'` | `'cellular component'`

This property is read-only.

Ontology of GO term, specified as one of these values:

- `'molecular function'`
- `'biological process'`
- `'cellular component'`

Use the `ontology` property to determine the ontology of term objects, or to access or filter term objects by ontology.

Tip If you know the GO identifier (for example, 179) of a term object, instead of its index or position number (for example, 155), you can use the following syntax to display the ontology of a term object:

```
GeneontObj(179).terms.ontology
```

For an example, see "Investigate Gene Ontology Term Properties" on page 1-1844.

Data Types: `char`

`part_of` — GO identifiers of GO terms that have a "part of" relationship with this GO term
numeric array

This property is read-only.

GO identifiers of GO terms that have a "part of" relationship with this GO term, specified as a numeric array.

Tip You can also use the `getancestors` method of a `geneont` object with the `'Relationtype'` property set to `'part_of'` to determine term objects with a "part of" relationship.

Tip If you know the GO identifier (for example, 42321) of a term object, instead of its index or position number (for example, 18703), you can use the following syntax to display the `part_of` property of a term object:

```
GeneontObj(42321).terms.part_of
```

For an example, see "Investigate Gene Ontology Term Properties" on page 1-1844.

Data Types: `single` | `double`

`synonym` — GO terms that are synonyms of this GO term
two-column cell array

This property is read-only.

GO terms that are synonyms of this GO term, specified as a two-column cell array. The first column contains a character vector specifying the type of synonym, such as `'exact_synonym'`, `'related_synonym'`, `'broad_synonym'`, `'narrow_synonym'`, or `'alt_id'`. The second column contains the GO identifier of the synonymous term or a character vector describing the synonymous term.

Tip If you know the GO identifier (for example, 398) of a term object, instead of its index or position number (for example, 352), you can use the following syntax to display the synonym of a term object:

```
GeneontObj(398).terms.synonym
```

For an example, see "Investigate Gene Ontology Term Properties" on page 1-1844.

Data Types: `cell`

Examples

Investigate Gene Ontology Term Properties

Investigate some of a properties of Gene ontology (GO) `term` object. First download the current Gene Ontology database into a `geneont` object.

```
GeneontObj = geneont('LIVE', true)
```

```
Gene Ontology object with 47331 Terms.
```

Investigate Ontology

Display the ontology of the `term` object in the 155th position in `GeneontObj`.

```
GeneontObj.terms(155).ontology
```

```
ans =
'biological process'
```

Create a cell array of character vectors that list the `ontology` property for each `term` in `GeneontObj`.

```
ontologies = get(GeneontObj.terms,'ontology');
size(ontologies)
```

```
ans = 1×2

       47331           1
```

Create a logical mask that identifies all the terms with an `ontology` property of `'cellular component'`.

```
mask = strcmp(ontologies,'cellular component');
```

Apply the logical mask to all the terms in `GeneontObj` to return a structure containing only terms with an `ontology` property of `'cellular component'`.

```
cell_comp_terms = GeneontObj.terms(mask);
size(cell_comp_terms)
```

```
ans = 1×2

        4470           1
```

Investigate Names

Display the `name` property of the `term` object in the 157th position in `GeneontObj`.

```
GeneontObj.terms(157).name
```

```
ans = 
'obsolete activation of MAPKK (pseudohyphal growth)'
```

Find the index or position number of the `term` object whose `name` property is `'membrane'`.

```
membrane_index = find(strcmp(get(GeneontObj.terms,'name'),'membrane'))
```

```
membrane_index = 9883
```

Use this index or position number and the `id` property to determine the GO identifier of the `term` object.

```
membrane_goid = GeneontObj.terms(membrane_index).id
```

```
membrane_goid = 16020
```

Use this GO identifier as input to the `getrelatives` method to find the GO identifiers of other `term` objects that are immediate relatives of the `term` object whose name property is `'membrane'`.

```
relative_ids = getrelatives(GeneontObj,membrane_goid)
```

```
relative_ids = 23×1

        5628
        5642
        5886
       16020
       19867
       19898
       31090
       31224
       34045
       34357
      ⋮
```

List the `name` properties of these `term` objects.

```
lst = GeneontObj(relative_ids).terms;
get(lst,'name')
```

```
ans = 23×1 cell
    {'prospore membrane'                                               }
    {'annulate lamellae'                                               }
    {'plasma membrane'                                                 }
    {'membrane'                                                        }
    {'outer membrane'                                                  }
    {'extrinsic component of membrane'                                 }
    {'organelle membrane'                                              }
    {'intrinsic component of membrane'                                 }
    {'phagophore assembly site membrane'                               }
    {'photosynthetic membrane'                                         }
    {'ascus membrane'                                                  }
    {'nuclear outer membrane-endoplasmic reticulum membrane network'}
    {'coated membrane'                                              }
    {'prospore membrane leading edge'                               }
    {'respirasome'                                                  }
    {'leaflet of membrane bilayer'                                  }
    {'side of membrane'                                             }
    {'plasma membrane region'                                       }
    {'membrane protein complex'                                     }
    {'membrane microdomain'                                         }
    {'cellular anatomical entity'                                   }
    {'pathogen-containing vacuole membrane'                         }
    {'spore inner membrane'                                         }
```

Investigate `part_of` and `is_a` Relationships

Display the `term` objects to which the `term` object in the 18,703rd position has an “is a” relationship.

```
GeneontObj.terms(18703).is_a
```

```
ans = 30282
```

Display the `term` objects to which the `term` object in the 18,703rd position has a “part of” relationship.

```
GeneontObj.terms(18703).part_of
```

```
ans = 43931
```

Investigate Synonyms

Display the `term` objects that are synonymous to the `term` object in the third position in `GeneontObj`.

```
synonyms = GeneontObj.terms(3).synonym
```

```
synonyms = 3×2 cell
    {'alt_id' }    {'GO:0019952'                                 }
    {'alt_id' }    {'GO:0050876'                                 }
    {'synonym'}    {'"reproductive physiological process" EXACT []'}
```

Display the text of the third synonym.

```
synonyms(3,2)
```

```
ans = 1×1 cell array
    {'"reproductive physiological process" EXACT []'}
```

Display the `term` objects that are synonymous to the `term` object in the 352nd position of `GeneontObj`.

```
GeneontObj.terms(352).synonym
```

```
ans = 1×2 cell
    {'synonym'}    {'"histone proline isomerization" EXACT []'}
```

Investigate Obsolete Status

Display the `obsolete` status of the `term` object in the third and seventh positions of `GeneontObj`.

```
GeneontObj.terms(3).obsolete
```

```
ans = logical
   0
```

```
GeneontObj.terms(7).obsolete
```

```
ans = logical
   1
```

Create a cell array of logicals that list the `obsolete` property for each `term` in `GeneontObj`.

```
obsolescence = get(GeneontObj.terms,'obsolete');
sum(cell2mat(obsolescence))
```

```
ans = 3718
```

The number of `obsolete` terms is 3718.

Create a logical mask from the cell array that identifies all the nonobsolete terms.

```
mask = ~cell2mat(obsolescence);
```

Apply the logical mask to all the terms in `GeneontObj` to return a structure containing only terms that are not obsolete.

```
nonobsolete_terms = GeneontObj.terms(mask)
```

```
  43613×1 struct array with fields:

    id
    name
    ontology
    definition
    comment
    synonym
    is_a
    part_of
    obsolete
```

Version History

Introduced before R2006a

See Also

`geneont`

tgspcinfo

Return information about SPC file

Syntax

`InfoStruct = tgspcinfo(File)`

Description

`InfoStruct = tgspcinfo(File)` returns a MATLAB structure containing summary information about a Galactic SPC file from Thermo Scientific®.

Input Arguments

`File`

Character vector or string specifying a file name or path and file name of an SPC file. If you specify only a file name, that file must be on the MATLAB search path or in the current folder.

Default:

Output Arguments

`InfoStruct`

MATLAB structure containing the following fields:

Field	Description
`Filename`	Name of the SPC file.
`FileSize`	Size of the SPC file in bytes.
`ExperimentType`	Experimental technique used to create the data.
`NumDataPoints`	Number of data points (*y* data values) in the SPC file.
`XFirst`	First *x* data value in the SPC file.
`XLast`	Last *x* data value in the SPC file.
`NumScans`	Number of scans or subfiles in the SPC file.
`XLabel`	Label for the *x* data values.
`YLabel`	Label for the *y* data values.
`ZLabel`	Label for the *z* data values.
`CollectionTime`	Date and time the scans were collected.
`CollectionTimeDatenum`	Date and time the scans were collected in serial date number format. For more information, see `datenum`.
`Resolution`	Instrument resolution.

Field	Description
SourceInstrument	Name or model of the instrument used to collect data.
InterferogramPeakPointNumber	Peak point number for interferograms. It is 0 for scans that are not interferograms.
Comment	User-provided comments.
CustomAxisUnitLabel	User-provided labels for the axis units.
SubScanHeaders	Header information for subfiles or scans, including scan index, next scan index, and *w* data value.
ZValues	Vector containing the *z* data values of all scans in the SPC file.

Examples

This example assumes that you already have an SPC file to use. `sample.spc` file is not provided with the Bioinformatics Toolbox software.

Return information about an SPC file:

```
% Return information about an SPC file named sample.spc
info = tgspcinfo('sample.spc')

Reading header for file: SAMPLE.SPC
File contains 1 scans

info =

                        Filename: 'SAMPLE.SPC'
                        FileSize: 48380
                  ExperimentType: 'General SPC'
                   NumDataPoints: 12031
                          XFirst: 6.2998e+003
                           XLast: 499.9531
                        NumScans: 1
                          XLabel: 'Wavenumber (cm-1)'
                          YLabel: 'Absorbance'
                          ZLabel: 'Arbitrary'
                  CollectionTime: '08-Mar-1993 15:13:00'
           CollectionTimeDatenum: 7.2800e+005
                      Resolution: '  .00   '
                SourceInstrument: ''
    InterferogramPeakPointNumber: 0
                         Comment: [1x74 char]
             CustomAxisUnitLabel: ''
                  SubScanHeaders: [1x1 struct]
                         ZValues: 0
```

Version History

Introduced in R2009b

See Also

`tgspcread` | `datenum`

tgspcread

Read data from SPC file

Syntax

```
SPCStruct = tgspcread(File)
tgspcread(..., 'ZRange', ZRangeValue, ...)
tgspcread(..., 'ScanIndices', ScanIndicesValue, ...)
tgspcread(..., 'Verbose', VerboseValue, ...)
```

Description

`SPCStruct = tgspcread(File)` reads a Galactic SPC file from Thermo Scientific, and returns the data in a MATLAB structure.

`tgspcread(..., 'PropertyName', PropertyValue, ...)` calls `tgspcread` with optional properties that use property name/property value pairs. You can specify one or more properties in any order. Enclose each *`PropertyName`* in single quotation marks. Each *`PropertyName`* is case insensitive. These property name/property value pairs are as follows:

`tgspcread(..., 'ZRange', ZRangeValue, ...)` specifies a range of *z* data values in the SPC file from which to extract scans.

`tgspcread(..., 'ScanIndices', ScanIndicesValue, ...)` specifies a scan, multiple scans, or range of scans in the SPC file to read.

`tgspcread(..., 'Verbose', VerboseValue, ...)` controls the display of the progress of the reading of the SPC file. Choices are `true` (default) or `false`.

Input Arguments

`File`

Character vector or string specifying a file name or path and file name of an SPC file that conforms to the Thermo Scientific Universal Data Format Specification. If you specify only a file name, that file must be on the MATLAB search path or in the current folder.

Default:

`ZRangeValue`

Two-element numeric array `[Start End]` that specifies the range of *z* data values in *`File`* to read. *`Start`* and *`End`* must be positive scalars, and *`Start`* must be less than *`End`*. Default is to extract all scans.

Tip For summary information about the *z* data values in an SPC file, use the `tgspcinfo` function.

Note If you specify a *`ZRangeValue`*, you cannot specify a *`ScanIndicesValue`*.

Default:

`ScanIndicesValue`

Positive integer, vector of integers, or a two-element numeric array [*`Start_Ind`*`:` *`End_Ind`*] that specifies a scan, multiple scans, or a range of scans in *`File`* to read. *`Start_Ind`* and *`End_Ind`* are each positive integers indicating a scan index. *`Start_Ind`* must be less than *`End_Ind`*. Default is to read all scans.

Tip For summary information about the scan indices in an SPC file, check the `NumScans` field in the structure returned by the `tgspcinfo` function.

Note If you specify a *`ScanIndicesValue`*, you cannot specify a *`ZRangeValue`*.

Default:

`VerboseValue`

Controls the display of the progress of the reading of *`File`*. Choices are `true` (default) or `false`.

Default:

Output Arguments

`SPCStruct`

Structure containing information from an SPC file. The structure contains the following fields.

Field	Description
`Header`	Structure containing the following fields: • `Filename` — Name of the SPC file. • `FileSize` — Size of the SPC file in bytes. • `ExperimentType` — Experimental technique used to create the data. • `NumDataPoints` — Number of data points (*y* data values) in the SPC file. • `XFirst` — First *x* data value in the SPC file. • `XLast` — Last *x* data value in the SPC file. • `NumScans` — Number of scans or subfiles in the SPC file. • `XLabel` — Label for the *x* data values. • `YLabel` — Label for the *y* data values. • `ZLabel` — Label for the *z* data values. • `CollectionTime` — Date and time the scan data were collected. • `CollectionTimeDatenum` — Date and time the scan data were collected in serial date number format. For more information, see `datenum`. • `Resolution` — Instrument resolution. • `SourceInstrument` — Name or model of instrument used to collect data. • `InterferogramPeakPointNumber` — Peak point number for interferograms. It is `0` for scans that are not interferograms. • `Comment` — User-provided comments. • `CustomAxisUnitLabel` — User-provided labels for the axis units. • `SubScanHeaders` — Header information for subfiles or scans, including scan index, next scan index, and *w* data value. • `ZValues` — Vector containing the *z* data values of all scans in the SPC file.
`X`	Vector or cell array containing the *x* data values. If all scans share the same *x* data values, then `X` is a vector. If scans have different *x* data values, then `X` is a cell array.
`Y`	Vector, matrix, or cell array containing the *y* data values. If there is only one scan, then `Y` is a vector. If there are multiple scans that share the same *x* data values, then `Y` is a matrix. If there are multiple scans having different *x* data values, then `Y` is a cell array.
`Z`	Vector containing the *z* data values of scans read from the SPC file

Examples

This example assumes that you already have an SPC file to use. `sample.spc` file is not provided with the Bioinformatics Toolbox software.

Read an SPC file:

```
% Read the contents of an SPC file into a MATLAB structure
out = tgspcread('results.spc')

File contains 1 scans

out = 

    Header: [1x1 struct]
         X: [12031x1 single]
         Y: [12031x1 double]
         Z: 0
```

Plot an SPC file:

```
% Plot the first scan in the SPC file:
plot(out.X,out.Y(:,1));
```

Version History

Introduced in R2009b

See Also

`tgspcinfo` | `jcampread` | `mzcdfinfo` | `mzcdf2peaks` | `mzcdfread` | `mzxmlread` | `mzxml2peaks` | `mzxmlinfo` | `datenum`

times (DataMatrix)

Multiply DataMatrix objects

Syntax

```
DMObjNew = times(DMObj1, DMObj2)
DMObjNew = DMObj1 .* DMObj2
DMObjNew = times(DMObj1, B)
DMObjNew = DMObj1 .* B
DMObjNew = times(B, DMObj1)
DMObjNew = B .* DMObj1
```

Input Arguments

DMObj1, *DMObj2*	DataMatrix objects, such as created by `DataMatrix` (object constructor).
B	MATLAB numeric or logical array.

Output Arguments

DMObjNew	DataMatrix object created by multiplication.

Description

`DMObjNew = times(DMObj1, DMObj2)` or the equivalent `DMObjNew = DMObj1 .* DMObj2` performs an element-by-element multiplication of the DataMatrix objects *DMObj1* and *DMObj2* and places the results in *DMObjNew*, another DataMatrix object. *DMObj1* and *DMObj2* must have the same size (number of rows and columns), unless one is a scalar (1-by-1 DataMatrix object). The size (number of rows and columns), row names, and column names for *DMObjNew* are the same as *DMObj1*, unless *DMObj1* is a scalar; then they are the same as *DMObj2*.

`DMObjNew = times(DMObj1, B)` or the equivalent `DMObjNew = DMObj1 .* B` performs an element-by-element multiplication of the DataMatrix object *DMObj1* and *B*, a numeric or logical array, and places the results in *DMObjNew*, another DataMatrix object. *DMObj1* and *B* must have the same size (number of rows and columns), unless *B* is a scalar. The size (number of rows and columns), row names, and column names for *DMObjNew* are the same as *DMObj1*.

`DMObjNew = times(B, DMObj1)` or the equivalent `DMObjNew = B .* DMObj1` performs an element-by-element multiplication of *B*, a numeric or logical array, and the DataMatrix object *DMObj1*, and places the results in *DMObjNew*, another DataMatrix object. *DMObj1* and *B* must have the same size (number of rows and columns), unless *B* is a scalar. The size (number of rows and columns), row names, and column names for *DMObjNew* are the same as *DMObj1*.

Note Arithmetic operations between a scalar DataMatrix object and a nonscalar array are not supported.

MATLAB calls `DMObjNew = times(X, Y)` for the syntax `DMObjNew = X .* Y` when `X` or `Y` is a DataMatrix object.

Version History

Introduced in R2008b

See Also

`DataMatrix` | `minus` | `plus` | “Arithmetic Operations”

Topics

DataMatrix object on page 1-734

topoorder (biograph)

(Removed) Perform topological sort of directed acyclic graph extracted from biograph object

Note The function has been removed. Use `toposort` instead.

Syntax

`*order* = topoorder(*BGObj*)`

Arguments

`BGObj`	Biograph object created by `biograph` (object constructor).

Description

Tip For introductory information on graph theory functions, see “Graph Theory Functions”.

`order` `= topoorder(`*`BGObj`*`)` returns an index vector with the order of the nodes sorted topologically. In topological order, an edge can exist between a source node `u` and a destination node `v`, if and only if `u` appears before `v` in the vector *`order`*. *`BGObj`* is a biograph object from which an N-by-N adjacency matrix is extracted and represents a directed acyclic graph (DAG). In the N-by-N sparse matrix, all nonzero entries indicate the presence of an edge.

Version History

Introduced in R2006b

R2022b: Removed

Errors starting in R2022b

The function has been removed. Use `toposort` instead.

R2022a: Warns

Warns starting in R2022a

The function issues a warning that it will be removed in a future release.

R2021b: To be removed

Not recommended starting in R2021b

The function runs without warning, but it will be removed in a future release.

References

[1] Siek, J.G., Lee, L-Q, and Lumsdaine, A. (2002). The Boost Graph Library User Guide and Reference Manual, (Upper Saddle River, NJ:Pearson Education).

See Also

`toposort` | `graph` | `digraph`

traceplot

Draw nucleotide trace plots

Syntax

```
traceplot(TraceStructure)
traceplot(A, C, G, T)
h = traceplot(...)
```

Description

`traceplot(`*`TraceStructure`*`)` creates a trace plot from data in a structure with fields `A`, `C`, `G`, and `T`.

`traceplot(`*`A`*`, `*`C`*`, `*`G`*`, `*`T`*`)` creates a trace plot from data in vectors *`A`*, *`C`*, *`G`*, and *`T`*.

`h = traceplot(...)` returns a structure with the handles of the lines corresponding to `A`, `C`, `G`, `T`.

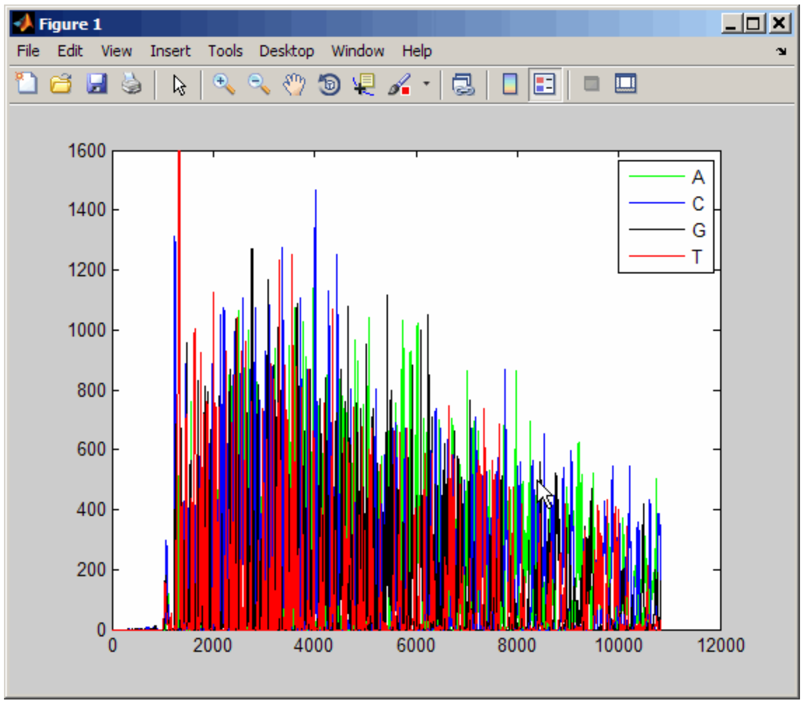
Examples

1 Read trace data from an SCF-formatted file into a MATLAB structure.

```
tstruct = scfread('sample.scf')

tstruct =

    A: [10827x1 double]
    C: [10827x1 double]
    G: [10827x1 double]
    T: [10827x1 double]
```

2 Draw a nucleotide trace plot of the data.

```
traceplot(tstruct)
```



Version History

Introduced before R2006a

See Also

`scfread`

traverse (biograph)

(Removed) Traverse biograph object by following adjacent nodes

Note The function has been removed. Use `bfsearch` or `dfsearch` instead.

Syntax

```
[disc, pred, closed] = traverse(BGObj, S)

[...] = traverse(BGObj, S, ...'Depth', DepthValue, ...)
[...] = traverse(BGObj, S, ...'Directed', DirectedValue, ...)
[...] = traverse(BGObj, S, ...'Method', MethodValue, ...)
```

Arguments

BGObj	Biograph object created by `biograph` (object constructor).
S	Integer that indicates the source node in *BGObj*.
DepthValue	Integer that indicates a node in *BGObj* that specifies the depth of the search. Default is `Inf` (infinity).
DirectedValue	Property that indicates whether graph represented by an N-by-N adjacency matrix extracted from a biograph object, *BGObj* is directed or undirected. Enter `false` for an undirected graph. This results in the upper triangle of the sparse matrix being ignored. Default is `true`.
MethodValue	Character vector that specifies the algorithm used to traverse the graph. Choices are: • `'BFS'` — Breadth-first search. Time complexity is `O(N+E)`, where `N` and `E` are number of nodes and edges respectively. • `'DFS'` — Default algorithm. Depth-first search. Time complexity is `O(N+E)`, where `N` and `E` are number of nodes and edges respectively.

Description

Tip For introductory information on graph theory functions, see “Graph Theory Functions”.

`[`*`disc`*`,` *`pred`*`,` *`closed`*`] = traverse(`*`BGObj`*`,` *`S`*`)` traverses the directed graph represented by an N-by-N adjacency matrix extracted from a biograph object, *BGObj*, starting from the node indicated by integer S. In the N-by-N sparse matrix, all nonzero entries indicate the presence of an edge. *disc* is a vector of node indices in the order in which they are discovered. *pred* is a vector of predecessor node indices (listed in the order of the node indices) of the resulting spanning tree. *closed* is a vector of node indices in the order in which they are closed.

`[...] = traverse(`*`BGObj`*`,` *`S`*`, ...'`*`PropertyName`*`',` *`PropertyValue`*`, ...)` calls `traverse` with optional properties that use property name/property value pairs. You can specify one or more properties in any order. Each *PropertyName* must be enclosed in single quotes and is case insensitive. These property name/property value pairs are as follows:

`[...] = traverse(`*`BGObj`*`, `*`S`*`, ...'Depth', `*`DepthValue`*`, ...)` specifies the depth of the search. *`DepthValue`* is an integer indicating a node in the graph represented by the N-by-N adjacency matrix extracted from a biograph object, *`BGObj`*. Default is `Inf` (infinity).

`[...] = traverse(`*`BGObj`*`, `*`S`*`, ...'Directed', `*`DirectedValue`*`, ...)` indicates whether the graph represented by the N-by-N adjacency matrix extracted from a biograph object, *`BGObj`* is directed or undirected. Set *`DirectedValue`* to `false` for an undirected graph. This results in the upper triangle of the sparse matrix being ignored. Default is `true`.

`[...] = traverse(`*`BGObj`*`, `*`S`*`, ...'Method', `*`MethodValue`*`, ...)` lets you specify the algorithm used to traverse the graph represented by the N-by-N adjacency matrix extracted from a biograph object, *`BGObj`*. Choices are:

- `'BFS'` — Breadth-first search. Time complexity is `O(N+E)`, where `N` and `E` are number of nodes and edges respectively.
- `'DFS'` — Default algorithm. Depth-first search. Time complexity is `O(N+E)`, where `N` and `E` are number of nodes and edges respectively.

Version History

Introduced in R2006b

R2022b: Removed

Errors starting in R2022b

The function has been removed. Use `bfsearch` or `dfsearch` instead.

R2022a: Warns

Warns starting in R2022a

The function issues a warning that it will be removed in a future release.

R2021b: To be removed

Not recommended starting in R2021b

The function runs without warning, but it will be removed in a future release.

References

[1] Sedgewick, R., (2002). Algorithms in C++, Part 5 Graph Algorithms (Addison-Wesley).

[2] Siek, J.G., Lee, L-Q, and Lumsdaine, A. (2002). The Boost Graph Library User Guide and Reference Manual, (Upper Saddle River, NJ:Pearson Education).

See Also

`bfsearch` | `dfsearch` | `graph` | `digraph`

var (DataMatrix)

Return variance values in DataMatrix object

Syntax

```
V = var(DMObj)
V = var(DMObj, Flag)
V = var(DMObj, Wgt)
V = var(..., Dim)
V = var(..., Dim, IgnoreNaN)
```

Input Arguments

`DMObj`	DataMatrix object, such as created by `DataMatrix` (object constructor).
`Flag`	Scalar specifying how to normalize the data. Choices are: • `0` — Default. Normalizes using a sample size of *N* – `1`, unless *N* = `1`, in which case, normalizes using a sample size of `1`. • `1` — Normalizes using a sample size of *N*. *N* = the number of elements in each column or row, as specified by *`Dim`*. For more information on the normalization equations, see the function `std`.
`Wgt`	Weight vector equal in length to the dimension over which `var` operates (specified by *`Dim`*. It is used to compute the variance.
`Dim`	Scalar specifying the dimension of *`DMObj`* to calculate the variances. Choices are: • `1` — Default. Returns variance values for elements in each column. • `2` — Returns variance values for elements in each row.
`IgnoreNaN`	Specifies if NaNs should be ignored. Choices are `true` (default) or `false`.

Output Arguments

`V`	An unbiased estimator of the variance within the columns or rows of a DataMatrix object. It can be either of the following: • Row vector containing the variance values from elements in each column in *`DMObj`* (when *`Dim`* = `1`) • Column vector containing the variance values from elements in each row in *`DMObj`* (when *`Dim`* = `2`)

Description

`V` `= var(`*`DMObj`*`)` returns the variance values of the elements in the columns of a DataMatrix object, treating NaNs as missing values. The data is normalized using a sample size of *N* – `1`, where *N* = the

number of elements in each column. *V* is a row vector containing the variance values for elements in each column in *DMObj*. The variance is the square of the standard deviation.

V = var(*DMObj*, *Flag*) specifies how to normalize the data. If *Flag* = 0, normalizes using a sample size of *N* – 1. If *Flag* = 1, normalizes using a sample size of *N*. *N* = the number of elements in each column or row, as specified by *Dim*. For more information on the normalization equations, see the function std. Default *Flag* = 0.

V = var(*DMObj*, *Wgt*) computes the variance using *Wgt*, a weight vector whose length must equal the length of the dimension over which var operates (specified by *Dim*). All elements in *Wgt* must be nonnegative. The var function normalizes *Wgt* to sum of 1.

V = var(..., *Dim*) returns the variance values of the elements in the columns or rows of a DataMatrix object, as specified by *Dim*. If *Dim* = 1, returns *V*, a row vector containing the variance values for elements in each column in *DMObj*. If *Dim* = 2, returns *V*, a column vector containing the variance values for elements in each row in *DMObj*. Default *Dim* = 1.

V = var(..., *Dim*, *IgnoreNaN*) specifies if NaNs should be ignored. *IgnoreNaN* can be true (default) or false.

Version History

Introduced in R2008b

See Also

DataMatrix | mean | median | std

Topics

DataMatrix object on page 1-734

variableDesc

Class: `bioma.data.MetaData`
Package: `bioma.data`

Retrieve or set variable descriptions for samples in MetaData object

Syntax

```
DSVarDescriptions = variableDesc(MDObj)
NewMDObj = variableDesc(MDObj, NewDSVarDescriptions)
```

Description

`DSVarDescriptions` `= variableDesc(`*`MDObj`*`)` returns a dataset array containing the variable names and descriptions for samples from a MetaData object.

`NewMDObj` `= variableDesc(`*`MDObj`*`,` *`NewDSVarDescriptions`*`)` replaces the sample variable descriptions in *`MDObj`*, a MetaData object, with *`NewDSVarDescriptions`*, and returns *`NewMDObj`*, a new MetaData object.

Input Arguments

`MDObj`

Object of the `bioma.data.MetaData` class.

Default:

`NewDSVarDescriptions`

Descriptions of the sample variable names, specified by one of the following:

- A new dataset array containing the variable names and descriptions for samples. In this dataset array, each row corresponds to a variable. The first column contains the variable name, and the second column (`VariableDescription`) contains a description of the variable. The row names (variable names) must match the row names (variable names) in *`DSVarDescriptions`*, the dataset array being replaced in the MetaData object, *`MDObj`*.
- Cell array of character vectors containing descriptions of the variables. The number of elements in *`VarDesc`* must equal the number of row names (variable names) in *`DSVarDescriptions`*, the dataset array being replaced in the MetaData object, *`MDObj`*.

Default:

Output Arguments

`DSVarDescriptions`

A dataset array containing the variable names and descriptions from a MetaData object. In this dataset array, each row corresponds to a sample variable. The first column contains the variable name, and the second column (`VariableDescription`) contains a description of the variable.

`NewMDObj`

Object of the `bioma.data.MetaData` class, returned after replacing the dataset array containing the sample variable descriptions.

Examples

Construct a MetaData object, and then retrieve the sample variable descriptions from it:

```
% Import bioma.data package to make constructor function
% available
import bioma.data.*
% Construct MetaData object from .txt file
MDObj2 = MetaData('File', 'mouseSampleData.txt', 'VarDescChar', '#');
% Retrieve the sample variable descriptions
VarDescriptions = variableDesc(MDObj2)
```

See Also

`bioma.data.MetaData` | `sampleNames` | `variableValues` | `variableNames`

Topics

“Representing Sample and Feature Metadata in MetaData Objects”

variableNames

Class: `bioma.data.MetaData`
Package: `bioma.data`

Retrieve or set variable names for samples in MetaData object

Syntax

```
VarNames = variableNames(MDObj)
VarNames = variableNames(MDObj, Subset)
NewMDObj = variableNames(MDObj, Subset, NewVarNames)
```

Description

`VarNames = variableNames(MDObj)` returns a cell array of character vectors specifying all variable names in a MetaData object.

`VarNames = variableNames(MDObj, Subset)` returns a cell array of character vectors specifying a subset the variable names in a MetaData object.

`NewMDObj = variableNames(MDObj, Subset, NewVarNames)` replaces the variable names specified by *`Subset`* in *`MDObj`*, a MetaData object, with *`NewVarNames`*, and returns *`NewMDObj`*, a new MetaData object.

Input Arguments

`MDObj`

Object of the `bioma.data.MetaData` class.

`Subset`

One of the following to specify a subset of the variable names in a MetaData object:

- Character vector or string specifying a variable name
- Cell array of character vectors or string vector specifying variable names
- Positive integer
- Vector of positive integers
- Logical vector

`NewVarNames`

New variable names for specific sample or feature variable names within a MetaData object, specified by one of the following:

- Numeric vector
- Cell array of character vectors or character array
- Character vector, which `variableNames` uses as a prefix for the variable names, with variable numbers appended to the prefix

- Logical `true` or `false` (default). If `true`, `variableNames` assigns unique variable names using the format `Var1`, `Var2`, etc.

The number of variable names in *`NewVarNames`* must equal the number of variable names specified by *`Subset`*.

Output Arguments

`VarNames`

Cell array of character vectors specifying all variable names in a MetaData object.

`NewMDObj`

Object of the `bioma.data.MetaData` class, returned after replacing the variable names.

Examples

Construct a MetaData object, and then retrieve the sample variable names from it:

```
% Import bioma.data package to make constructor function
% available
import bioma.data.*
% Construct MetaData object from .txt file
MDObj2 = MetaData('File', 'mouseSampleData.txt', 'VarDescChar', '#');
% Retrieve the sample variable names
VNames = variableNames(MDObj2)
```

See Also

`bioma.data.MetaData` | `sampleNames` | `variableValues` | `variableDesc`

Topics

“Representing Sample and Feature Metadata in MetaData Objects”

variableValues

Class: `bioma.data.MetaData`
Package: `bioma.data`

Retrieve or set variable values for samples in MetaData object

Syntax

```
DSVarValues = variableValues(MDObj)
NewMDObj = variableValues(MDObj, NewDSVarValues)
```

Description

`DSVarValues` `= variableValues(`*`MDObj`*`)` returns a dataset array containing the measured value of each variable per sample from a MetaData object.

`NewMDObj` `= variableValues(`*`MDObj`*`,` *`NewDSVarValues`*`)` replaces the sample variable values in *`MDObj`*, a MetaData object, with *`NewDSVarValues`*, and returns *`NewMDObj`*, a new MetaData object.

Input Arguments

`MDObj`

Object of the `bioma.data.MetaData` class.

Default:

`NewDSVarValues`

A new dataset array containing a value for each variable per sample. In this dataset array, the columns correspond to variables and rows correspond to samples. The row names (sample names) must match the row names (sample names) in *`DSVarValues`*, the dataset array being replaced in the MetaData object, *`MDObj`*.

Default:

Output Arguments

`DSVarValues`

A dataset array containing the measured value of each variable per sample from a MetaData object. In this dataset array, the columns correspond to variables and rows correspond to samples.

`NewMDObj`

Object of the `bioma.data.MetaData` class, returned after replacing the dataset array containing the sample variable values.

Examples

Construct a MetaData object, and then retrieve the sample variable values from it:

```
% Import bioma.data package to make constructor function
% available
import bioma.data.*
% Construct MetaData object from .txt file
MDObj2 = MetaData('File', 'mouseSampleData.txt', 'VarDescChar', '#');
% Retrieve the sample variable values
VarValues = variableValues(MDObj2)
```

See Also

`bioma.data.MetaData` | `sampleNames` | `variableNames` | `variableDesc`

Topics

"Representing Sample and Feature Metadata in MetaData Objects"

varValuesTable

Class: `bioma.data.MetaData`
Package: `bioma.data`

Create 2-D graphic table GUI of variable values in MetaData object

Syntax

`Handle` `= varValuesTable(`*`MDObj`*`)`
`Handle` `= varValuesTable(`*`MDObj`*`, `*`ParentHandle`*`)`

Description

`Handle` `= varValuesTable(`*`MDObj`*`)` creates a 2-D graphic table containing variable data from a MetaData object and returns a uitable handle to the table.

`Handle` `= varValuesTable(`*`MDObj`*`, `*`ParentHandle`*`)` specifies the parent handle to the table. The parent can be a figure or uipanel handle.

Input Arguments

`MDObj`

Object of the `bioma.data.MetaData` class.

Default:

`ParentHandle`

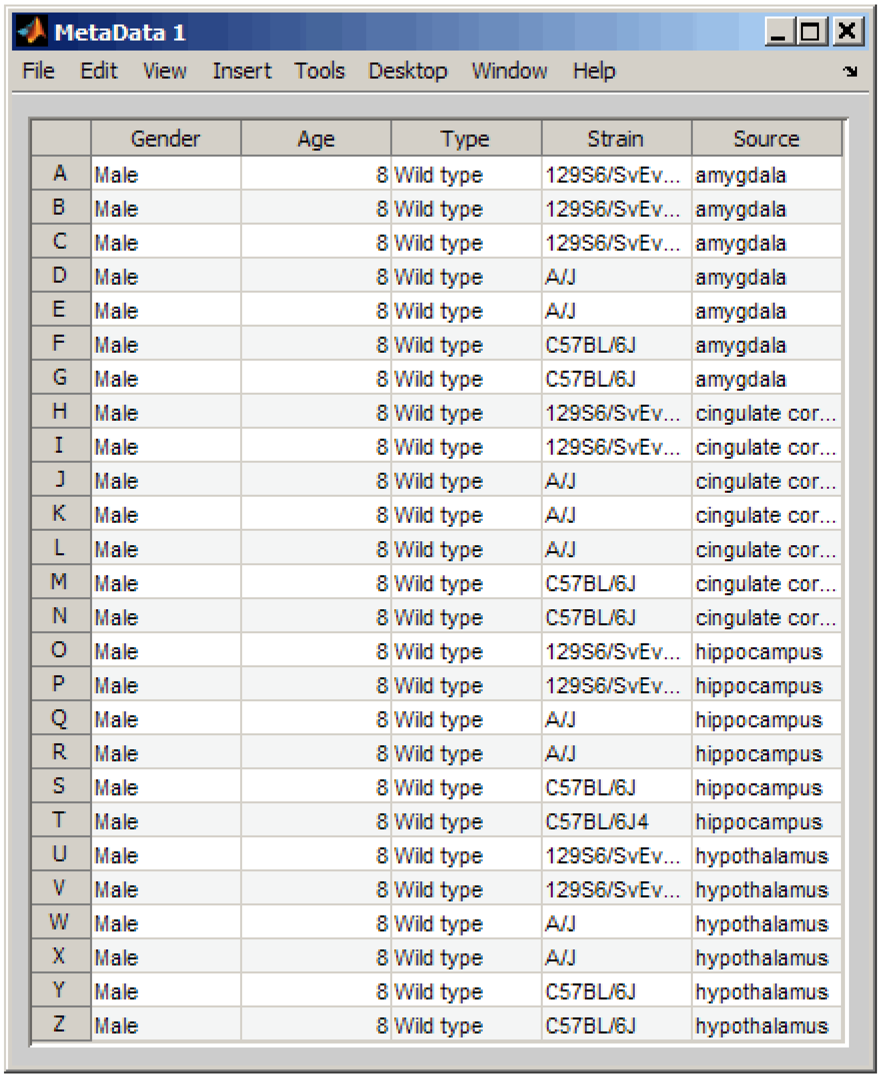
Figure or uipanel handle to be the parent handle to the table.

Default:

Examples

Construct a MetaData object, and then create a 2-D table of the variable values from it:

```
% Import bioma.data package to make constructor function
% available
import bioma.data.*
% Construct MetaData object from .txt file
MDObj2 = MetaData('File', 'mouseSampleData.txt', 'VarDescChar', '#');
% Retrieve the sample variable values in a table
handle = varValuesTable(MDObj2)
```



	Gender	Age	Type	Strain	Source
A	Male	8	Wild type	129S6/SvEv...	amygdala
B	Male	8	Wild type	129S6/SvEv...	amygdala
C	Male	8	Wild type	129S6/SvEv...	amygdala
D	Male	8	Wild type	A/J	amygdala
E	Male	8	Wild type	A/J	amygdala
F	Male	8	Wild type	C57BL/6J	amygdala
G	Male	8	Wild type	C57BL/6J	amygdala
H	Male	8	Wild type	129S6/SvEv...	cingulate cor...
I	Male	8	Wild type	129S6/SvEv...	cingulate cor...
J	Male	8	Wild type	A/J	cingulate cor...
K	Male	8	Wild type	A/J	cingulate cor...
L	Male	8	Wild type	A/J	cingulate cor...
M	Male	8	Wild type	C57BL/6J	cingulate cor...
N	Male	8	Wild type	C57BL/6J	cingulate cor...
O	Male	8	Wild type	129S6/SvEv...	hippocampus
P	Male	8	Wild type	129S6/SvEv...	hippocampus
Q	Male	8	Wild type	A/J	hippocampus
R	Male	8	Wild type	A/J	hippocampus
S	Male	8	Wild type	C57BL/6J	hippocampus
T	Male	8	Wild type	C57BL/6J4	hippocampus
U	Male	8	Wild type	129S6/SvEv...	hypothalamus
V	Male	8	Wild type	129S6/SvEv...	hypothalamus
W	Male	8	Wild type	A/J	hypothalamus
X	Male	8	Wild type	A/J	hypothalamus
Y	Male	8	Wild type	C57BL/6J	hypothalamus
Z	Male	8	Wild type	C57BL/6J	hypothalamus

See Also

`bioma.data.MetaData`

Topics

"Representing Sample and Feature Metadata in MetaData Objects"

vertcat (DataMatrix)

Concatenate DataMatrix objects vertically

Syntax

```
DMObjNew = vertcat(DMObj1, DMObj2, ...)
DMObjNew = (DMObj1; DMObj2; ...)
DMObjNew = vertcat(DMObj1, B, ...)
DMObjNew = (DMObj1, B, ...)
```

Input Arguments

DMObj1, *DMObj2*	DataMatrix objects, such as created by `DataMatrix` (object constructor).
B	MATLAB numeric or logical array.

Output Arguments

DMObjNew	DataMatrix object created by vertical concatenation.

Description

DMObjNew = `vertcat(`*DMObj1*`, `*DMObj2*`, ...)` or the equivalent *DMObjNew* `= (`*DMObj1*`;` *DMObj2*`; ...)` vertically concatenates the DataMatrix objects *DMObj1* and *DMObj2* into *DMObjNew*, another DataMatrix object. *DMObj1* and *DMObj2* must have the same number of columns. The column names and the order of columns for *DMObjNew* are the same as *DMObj1*. The column names of *DMObj2* and any other DataMatrix object input arguments are not preserved. The row names for *DMObjNew* are the row names of *DMObj1*, *DMObj2*, and other DataMatrix object input arguments.

DMObjNew = `vertcat(`*DMObj1*`, `*B*`, ...)` or the equivalent *DMObjNew* `= (`*DMObj1*`, `*B*`, ...)` vertically concatenates the DataMatrix object *DMObj1* and a numeric or logical array *B* into *DMObjNew*, another DataMatrix object. *DMObj1* and *B* must have the same number of columns. The column names for *DMObjNew* are the same as *DMObj1*. The column names of *DMObj2* and any other DataMatrix object input arguments are not preserved. The row names for *DMObjNew* are the row names of *DMObj1* and empty for the rows from *B*.

MATLAB calls *DMObjNew* `= vertcat(`*X1*`, `*X2*`, `*X3*`, ...)` for the syntax *DMObjNew* `= [`*X1*`; `*X2*`; `*X3*`; ...]` when any one of *X1*, *X2*, *X3*, etc. is a DataMatrix object.

Version History

Introduced in R2008b

See Also

`DataMatrix` | `horzcat`

Topics
DataMatrix object on page 1-734

view (biograph)

(Removed) Draw figure from biograph object

Note The function has been removed. Use the `plot` function of `graph` or `digraph` instead. The name-value arguments of the `plot` function can be used to control the properties of the graph. The arguments can be used as replacements for many of the `biograph` object properties to change the appearance of the graph.

Syntax

```
view(BGobj)
BGobjHandle = view(BGobj)
```

Arguments

`BGobj`	Biograph object created with the function `biograph`.

Description

`view(`*`BGobj`*`)` opens a Figure window and draws a graph represented by a biograph object (*`BGobj`*). When the biograph object is already drawn in the Figure window, this function only updates the graph properties.

`BGobjHandle` `= view(`*`BGobj`*`)` returns a handle to a deep copy of the biograph object (`BGobj`) in the Figure window. When updating an existing figure, you can use the returned handle to change object properties programmatically or from the command line. When you close the Figure window, the handle is no longer valid. The original biograph object (*`BGobj`*) is left unchanged.

Version History

Introduced before R2006a

R2022b: Removed

Errors starting in R2022b

The function has been removed. Use the `plot` function of `graph` or `digraph` instead. The name-value arguments of the `plot` function can be used to control the properties of the graph. The arguments can be used as replacements for many of the `biograph` object properties to change the appearance of the graph.

R2022a: Warns

Warns starting in R2022a

The function issues a warning that it will be removed in a future release.

R2021b: To be removed

Not recommended starting in R2021b

The function runs without warning, but it will be removed in a future release.

See Also

`plot` | `graph` | `digraph`

view

Display heatmap or clustergram

Syntax

```
view(hm_cg_object)
```

Description

`view(hm_cg_object)` displays a heatmap or clustergram of `hm_cg_object`.

Examples

Plot Heatmap of Data Matrix

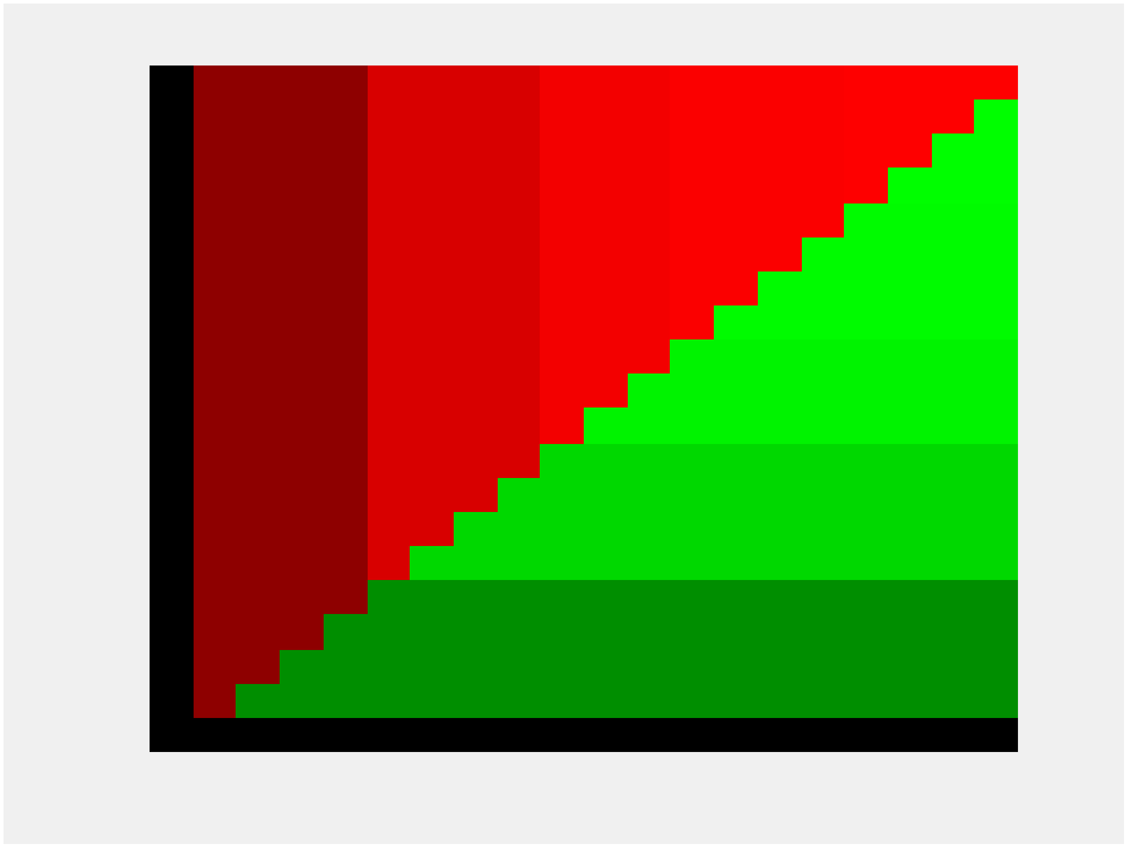
Create a matrix of data.

```
data = gallery('invhess',20);
```

Display a 2-D color heatmap of the data.

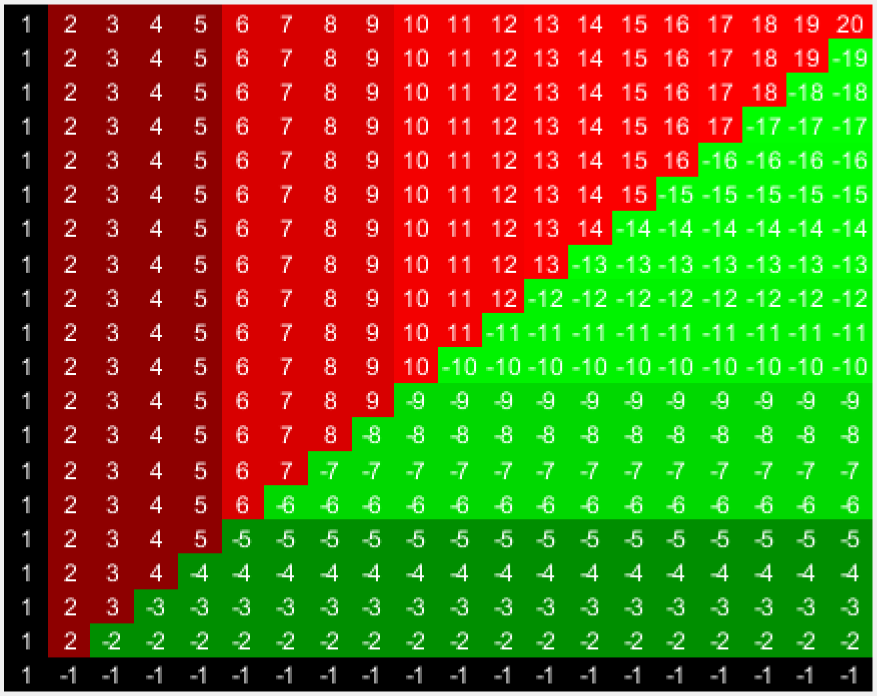
```
hmo = HeatMap(data);
```

```
             Standardize: '[column | row | {none}]'
               Symmetric: '[true | false].'
            DisplayRange: 'Scalar.'
                Colormap: []
               ImputeFun: 'string -or- function handle -or- cell array'
            ColumnLabels: 'Cell array of strings, or an empty cell array'
               RowLabels: 'Cell array of strings, or an empty cell array'
      ColumnLabelsRotate: []
         RowLabelsRotate: []
                Annotate: '[on | {off}]'
          AnnotPrecision: []
              AnnotColor: []
       ColumnLabelsColor: 'A structure array.'
          RowLabelsColor: 'A structure array.'
       LabelsWithMarkers: '[true | false].'
    ColumnLabelsLocation: '[ top | {bottom} ]'
       RowLabelsLocation: '[ {left} | right ]'
```



Display the data values in the heatmap.

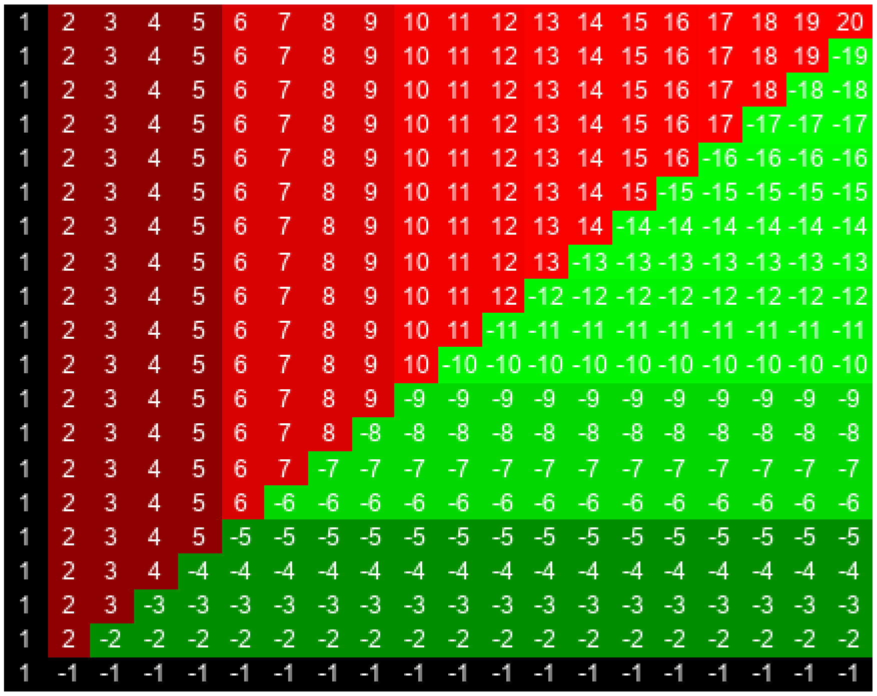
```
hmo.Annotate = true;
view(hmo)
```



1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	-19
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	-18	-18
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	-17	-17	-17
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	-16	-16	-16	-16
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	-15	-15	-15	-15	-15
1	2	3	4	5	6	7	8	9	10	11	12	13	14	-14	-14	-14	-14	-14	-14
1	2	3	4	5	6	7	8	9	10	11	12	13	-13	-13	-13	-13	-13	-13	-13
1	2	3	4	5	6	7	8	9	10	11	12	-12	-12	-12	-12	-12	-12	-12	-12
1	2	3	4	5	6	7	8	9	10	11	-11	-11	-11	-11	-11	-11	-11	-11	-11
1	2	3	4	5	6	7	8	9	10	-10	-10	-10	-10	-10	-10	-10	-10	-10	-10
1	2	3	4	5	6	7	8	9	-9	-9	-9	-9	-9	-9	-9	-9	-9	-9	-9
1	2	3	4	5	6	7	8	-8	-8	-8	-8	-8	-8	-8	-8	-8	-8	-8	-8
1	2	3	4	5	6	7	-7	-7	-7	-7	-7	-7	-7	-7	-7	-7	-7	-7	-7
1	2	3	4	5	6	-6	-6	-6	-6	-6	-6	-6	-6	-6	-6	-6	-6	-6	-6
1	2	3	4	5	-5	-5	-5	-5	-5	-5	-5	-5	-5	-5	-5	-5	-5	-5	-5
1	2	3	4	-4	-4	-4	-4	-4	-4	-4	-4	-4	-4	-4	-4	-4	-4	-4	-4
1	2	3	-3	-3	-3	-3	-3	-3	-3	-3	-3	-3	-3	-3	-3	-3	-3	-3	-3
1	2	-2	-2	-2	-2	-2	-2	-2	-2	-2	-2	-2	-2	-2	-2	-2	-2	-2	-2
1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1

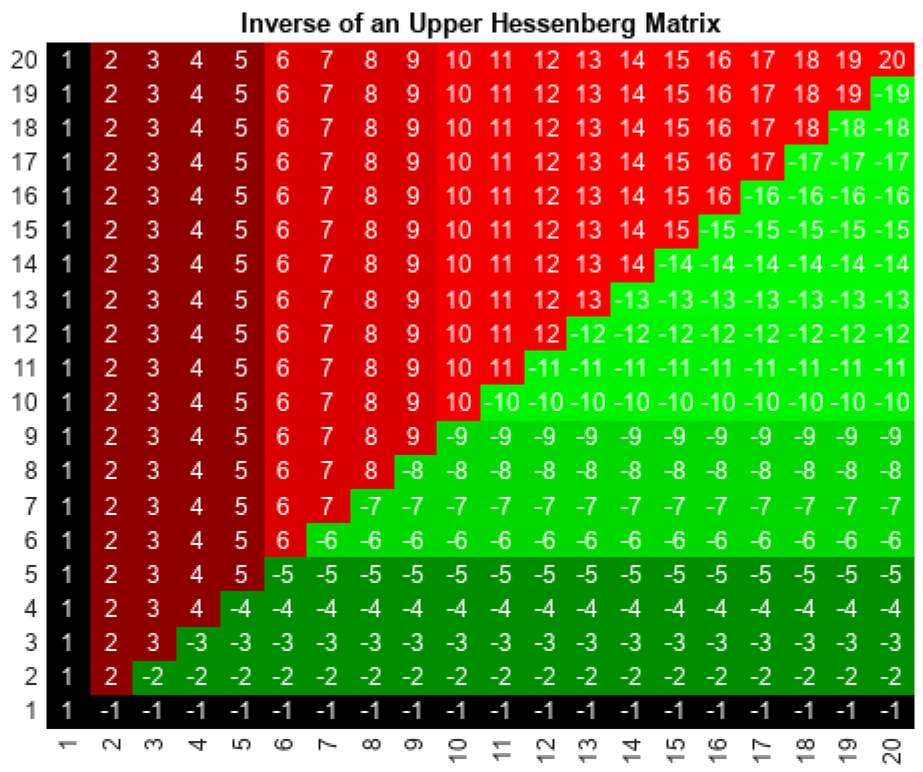
Use the `plot` function to display the heatmap in another figure specified by the figure handle `fH`.

```
fH = figure;
hA = plot(hmo,fH);
```



Use the returned axes handle hA to specify the axes properties.

```
hA.Title.String = 'Inverse of an Upper Hessenberg Matrix';
hA.XTickLabelMode = 'auto';
hA.YTickLabelMode = 'auto';
```



Input Arguments

`hm_cg_object` — Heatmap or clustergram object
`HeatMap` object | `clustergram` object

Heatmap or clustergram object, specified as a `HeatMap` object or `clustergram` object.

Version History

Introduced in R2009b

See Also

`HeatMap` | `clustergram`

view (phytree)

View phylogenetic tree

Syntax

```
view(Tree)
view(Tree, IntNodes)
```

Arguments

Tree	Phylogenetic tree (`phytree` object) created with the function `phytree`.
IntNodes	Nodes from the `phytree` object to initially display in the *Tree*.

Description

`view(`*`Tree`*`)` opens the Phylogenetic Tree window and draws a tree from data in a `phytree` object (*Tree*). The significant distances between branches and nodes are in the horizontal direction. Vertical distances have no significance and are selected only for display purposes. You can access tools to edit and analyze the tree from the Phylogenetic Tree menu bar or by using the left and right mouse buttons.

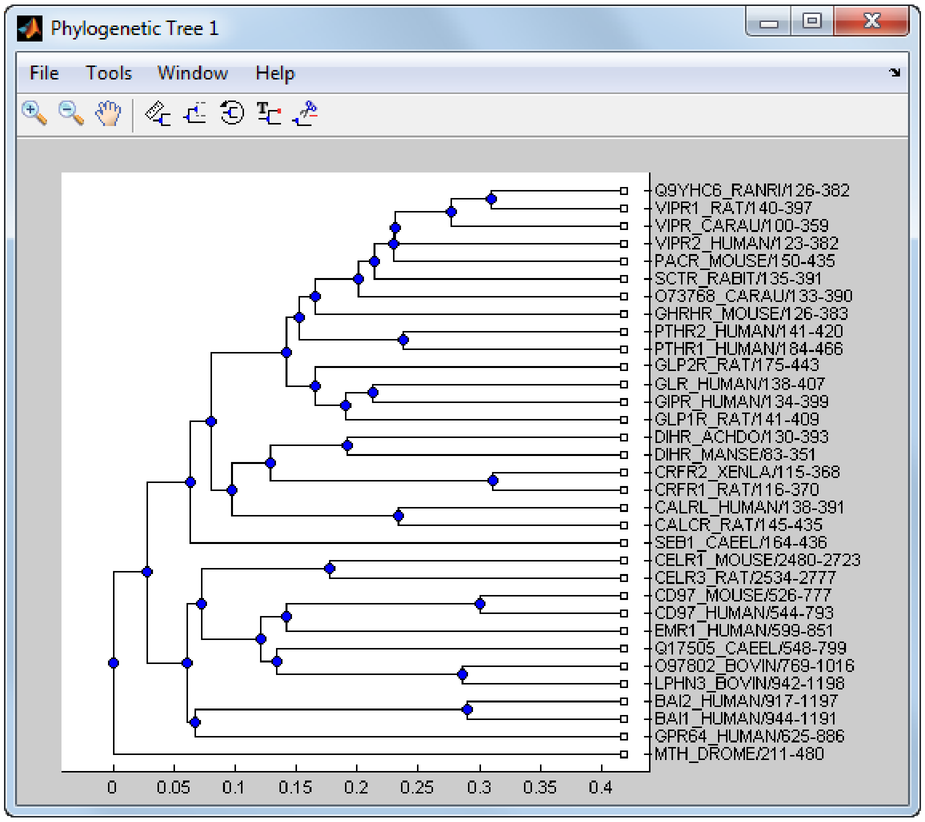
`view(`*`Tree, IntNodes`*`)` opens the Phylogenetic Tree window with an initial selection of nodes specified by *IntNodes*. *IntNodes* can be a logical array of any of the following sizes: `NumLeaves + NumBranches x 1`, `NumLeaves x 1`, or `NumBranches x 1`. *IntNodes* can also be a list of indices.

Examples

View a Phylogenetic Tree

Load and view a sample phylogenetic tree.

```
tr = phytreeread('pf00002.tree');
view(tr)
```



Version History

Introduced before R2006a

See Also

`phytree` | `phytreeread` | `phytreeviewer` | `seqlinkage` | `seqneighjoin` | `cluster` | `plot`

Topics

phytree object on page 1-1449

weights (phytree)

Calculate weights for phylogenetic tree

Syntax

```
W = weights(Tree)
```

Arguments

Tree	Phylogenetic tree (`phytree` object) created with the function `phytree`.

Description

`W = weights(Tree)` calculates branch proportional weights for every leaf in a tree (*Tree*) using the Thompson-Higgins-Gibson method. The distance of every segment of the tree is adjusted by dividing it by the number of leaves it contains. The sequence weights are the result of normalizing to unity the new patristic distances between every leaf and the root.

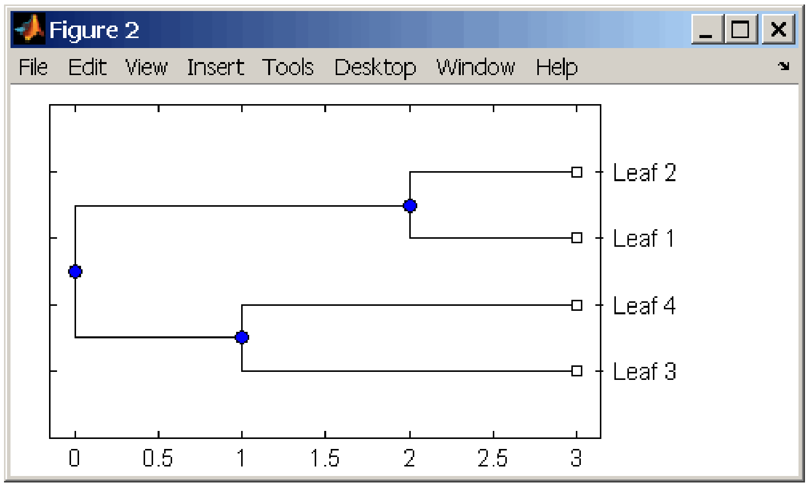
Examples

1 Create an ultrametric tree with specified branch distances.

```
bd = [1 2 3]';
tr_1 = phytree([1 2;3 4;5 6],bd)
```

2 View the tree.

```
view(tr_1)
```



3 Display the calculated weights.

```
weights(tr_1)

ans =
    1.0000
    1.0000
    0.8000
    0.8000
```

Version History

Introduced before R2006a

References

[1] Thompson JD, Higgins DG, Gibson TJ (1994), "CLUSTAL W: Improving the sensitivity of progressive multiple sequence alignment through sequence weighting, position-specific gap penalties and weight matrix choice," Nucleic Acids Research, 22(22):4673-4680.

[2] Henikoff S, Henikoff JG (1994), “Position-based sequence weights,” Journal Molecular Biology, 243(4):574-578.

See Also

`multialign` | `phytree` | `profalign` | `seqlinkage`

Topics

phytree object on page 1-1449

wait

Package: `bioinfo.pipeline`

Wait for running blocks to complete

Syntax

```
wait(pipeline)
wait(pipeline,blocks)
```

Description

`wait(pipeline)` waits for the running blocks in the pipeline to complete or waits until the blocks are guaranteed not to complete for various reasons, including upstream errors or blocks not being selected to run, before executing the subsequent command.

`wait(pipeline,blocks)` waits for the specified `blocks` to complete or waits until the blocks are guaranteed not to complete.

Examples

Wait for Running Blocks in Bioinformatics Pipeline

Import the Pipeline and block objects needed for the example.

```
import bioinfo.pipeline.Pipeline
import bioinfo.pipeline.blocks.*
```

Create a pipeline.

```
P = Pipeline;
```

Add some pause blocks for illustration purposes. Each block pauses with 10 seconds and 3 seconds, respectively.

```
PB1 = UserFunction(@() pause(10));
PB2 = UserFunction(@() pause(3));
addBlock(P,[PB1,PB2]);
```

Run the pipeline in parallel.

```
run(P,UseParallel=true);
```

```
Starting parallel pool (parpool) using the 'Processes' profile ...
Connected to parallel pool with 4 workers.
```

While the pipeline is still running and if you immediately execute the next commands to get the block results, you get a `Running` status instead.

```
results = processTable(P,Expanded=true)
```

```
results=2×5 table
         Block            Status            RunStart           RunEnd        RunErrors     
    ________________    _______    ____________________    ______    ________________

    "UserFunction_1"    Running    23-Nov-2022 17:56:37     NaT      {0×0 MException}
    "UserFunction_2"    Running    23-Nov-2022 17:56:37     NaT      {0×0 MException}
```

```
PB1Status = results(1,:);
PB1Status.Status
```

```
ans = 
  RunStatus enumeration

    Running
```

Instead use the `wait` function to wait for the running blocks to finish first and return the completed results afterwards.

```
wait(P);
results = processTable(P,Expanded=true)
```

```
results=2×5 table
         Block             Status              RunStart                 RunEnd                RunErrors
    ________________    _________    ____________________    ____________________    ___________

    "UserFunction_1"    Completed    23-Nov-2022 17:56:37    23-Nov-2022 17:56:48    {0×0 MExcept
    "UserFunction_2"    Completed    23-Nov-2022 17:56:37    23-Nov-2022 17:56:41    {0×0 MExcept
```

```
PB1Status = results(1,:);
PB1Status.Status
```

```
ans = 
  RunStatus enumeration

    Completed
```

Input Arguments

`pipeline` — Bioinformatics pipeline
`bioinfo.pipeline.Pipeline` object

Bioinformatics pipeline, specified as a `bioinfo.pipeline.Pipeline` object.

`blocks` — Pipeline blocks
`bioinfo.pipeline.Block` object | vector of objects | character vector | string scalar | ...

Pipeline blocks, specified as a `bioinfo.pipeline.Block` object, vector of block objects, character vector, string scalar, string vector, or cell array of character vectors representing block names.

Version History

Introduced in R2023a

See Also

`cancel` | `fetchResults` | `results` | `bioinfo.pipeline.Block` | `bioinfo.pipeline.Pipeline` | **Biopipeline Designer**

write

Write contents of BioRead or BioMap object to file

Syntax

```
write(object,fName)
write(object,fName,Name,Value)
```

Description

`write(object,fName)` writes the contents of a BioRead or BioMap object to a file named `fName`.

`write(object,fName,Name,Value)` uses additional options specified by one or more name-value pair arguments. For example, `write(object,'data','Format','FASTQ')` saves the contents of the object in the FASTQ-formatted file `data.fastq`.

Examples

Save NGS Data to a File

Create a `BioRead` object from an FASTQ file.

```
BRObj = BioRead('SRR005164_1_50.fastq');
```

Extract the first 10 elements from `BRObj` and store them in a new object.

```
subsetBRObj = getSubset(BRObj, [1:10]);
```

Write the contents of the subset data to a file named `subsetData.fastq` in the local `C` drive. By default, the file is FASTQ-formatted because the object contains the quality data.

```
write(subsetBRObj, 'C:\subsetData');
```

Input Arguments

`object` — Object containing read data
`BioRead` object | `BioMap` object

Object containing the read data, specified as a `BioRead` or `BioMap` object.

Example: `bioreadObj`

`fName` — Name of file
character vector | string

Name of the file where the contents of `object` are written, specified as a character vector or string.

The function adds the file extension automatically depending on the type of data the object contains. If you provide the extension, the function checks for consistency between the extension and the data format of the object. The file name can be prefixed by a file path. If the path is missing, the function writes the file to the same folder where the source file is located or to the current folder if the data is in memory.

Example: `'output'`

Data Types: `char`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `write(object,'data','Format','FASTQ')` saves the contents of the object in the FASTQ-formatted file `data.fastq`.

`Format` — File format
`'FASTQ'` | `'FASTA'` | `'SAM'` | `'BAM'`

File format, specified as a comma-separated pair consisting of `'Format'` and a character vector or string.

For `BioRead` objects, available formats are `'FASTA'` and `'FASTQ'`. The default format is `'FASTA'` if the object does not contain qualities, that is, the `Quality` property is empty. Otherwise, the default format is `'FASTQ'`.

For `BioMap` objects, available formats are `'FASTA'`, `'FASTQ'`, `'SAM'`, and `'BAM'` (default).

Example: `'Format','FASTA'`

Data Types: `char`

`Overwrite` — Boolean indicator to overwrite existing file
`false` (default) | `true`

Boolean indicator to overwrite an existing file, specified as a comma-separated pair consisting of `'Overwrite'` and `true` or `false`. If `true`, the function overwrites the file and deletes any respective index file (`*.idx,*.bai,*.linearindex`) or ordered file (`*.ordered.bam, *.ordered.sam`) that is no longer needed.

Example: `'Overwrite',true`

Data Types: `logical`

Version History
Introduced in R2010a

See Also
`BioMap` | `BioRead`

Topics
“Manage Sequence Read Data in Objects”

zonebackadj

Perform background adjustment on Affymetrix microarray probe-level data using zone-based method

Syntax

```
BackAdjustedData = zonebackadj(Data)
[BackAdjustedData, ZoneStruct] = zonebackadj(Data)
[BackAdjustedData, ZoneStruct, Background] = zonebackadj(Data)

... = zonebackadj(Data, ...'NumZones', NumZonesValue, ...)
... = zonebackadj(Data, ...'Percent', PercentValue, ...)
... = zonebackadj(Data, ...'SmoothFactor', SmoothFactorValue, ...)
... = zonebackadj(Data, ...'NoiseFrac', NoiseFracValue, ...)
... = zonebackadj(Data, ...'CDF', CDFValue, ...)
... = zonebackadj(Data, ...'Mask', MaskValue, ...)
... = zonebackadj(Data, ...'Showplot', ShowplotValue, ...)
```

Input Arguments

Data	Either of the following: • MATLAB structure containing probe intensities from an Affymetrix CEL file, such as returned by `affyread` when used to read a CEL file. • Array of MATLAB structures containing probe intensities from multiple Affymetrix CEL files.
NumZonesValue	Scalar or two-element vector that specifies the number of zones to use in the background adjustment. If a scalar, it must be a square number. If a two-element vector, the first element specifies the number of rows and the second element specifies the number of columns in a nonsquare grid. Default is `16`.
PercentValue	Value that specifies a percentage, *P*, such that the lowest *P* percent of ranked intensity values from each zone is used to estimate the background for that zone. Default is `2`.
SmoothFactorValue	Value that specifies the smoothing factor used in the calculation of the weighted average of the contributions of each zone to the background of a point. Default is `100`.
NoiseFracValue	Value that specifies the noise fraction, `NF`, such that the background-adjusted value is given by `max((Intensity - WeightedBackground), NF*LocalNoiseEstimate)`. Default is `0.5`.

`CDFValue`	Either of the following: • Character vector or string specifying a file name or path and file name of an Affymetrix CDF library file. If you specify only a file name, the file must be on the MATLAB search path or in the current folder. • MATLAB structure containing information from an Affymetrix CDF library file, such as returned by `affyread` when used to read a CDF file. The CDF library file or structure specifies control cells, which are not used in the background estimates.
`MaskValue`	Logical vector that specifies which cells to mask and not use in the background estimates. In the vector, `0 = not masked` and `1 = masked`. Defaults are the values in the `Masked` column of the `Probes` field of the CEL file.
`ShowplotValue`	Controls the plotting of an image of the background estimates. Choices are `true` or `false` (default).

Output Arguments

`BackAdjustedData`	Matrix or cell array of vectors containing background-adjusted probe intensity values.
`ZoneStruct`	MATLAB structure containing the centers of the zones used to perform the background adjustment and the estimates of the background values at the center of each zone.
`Background`	Matrix or cell array of vectors containing the estimated background values for each probe.

Description

`BackAdjustedData` `= zonebackadj(`*`Data`*`)` returns the background-adjusted probe intensities from *`Data`*, which contains probe intensities from Affymetrix CEL files. Details of the background adjustment are described in Statistical Algorithms Description Document.

`[`*`BackAdjustedData`*`, `*`ZoneStruct`*`] = zonebackadj(`*`Data`*`)` also returns a structure containing the centers of the zones used to perform the background adjustment and the estimates of the background values at the center of each zone.

`[`*`BackAdjustedData`*`, `*`ZoneStruct`*`, `*`Background`*`] = zonebackadj(`*`Data`*`)` also returns a matrix or cell array of vectors containing the estimated background values for each probe.

`... = zonebackadj(`*`Data`*`, ...'`*`PropertyName`*`', `*`PropertyValue`*`, ...)` calls `zonebackadj` with optional properties that use property name/property value pairs. You can specify one or more properties in any order. Each *`PropertyName`* must be enclosed in single quotation marks and is case insensitive. These property name/property value pairs are as follows:

`... = zonebackadj(`*`Data`*`, ...'NumZones', `*`NumZonesValue`*`, ...)` specifies the number of zones to use in the background adjustment. *`NumZonesValue`* can be either a scalar that is a square number or a two-element array in which the first element specifies the number of rows and the second element specifies the number of columns in a nonsquare grid. Default is `16`.

`... = zonebackadj(`*`Data`*`, ...'Percent', `*`PercentValue`*`, ...)` specifies a percentage, *P*, such that the lowest *P* percent of ranked intensity values from each zone is used to estimate the background for that zone. Default is `2`.

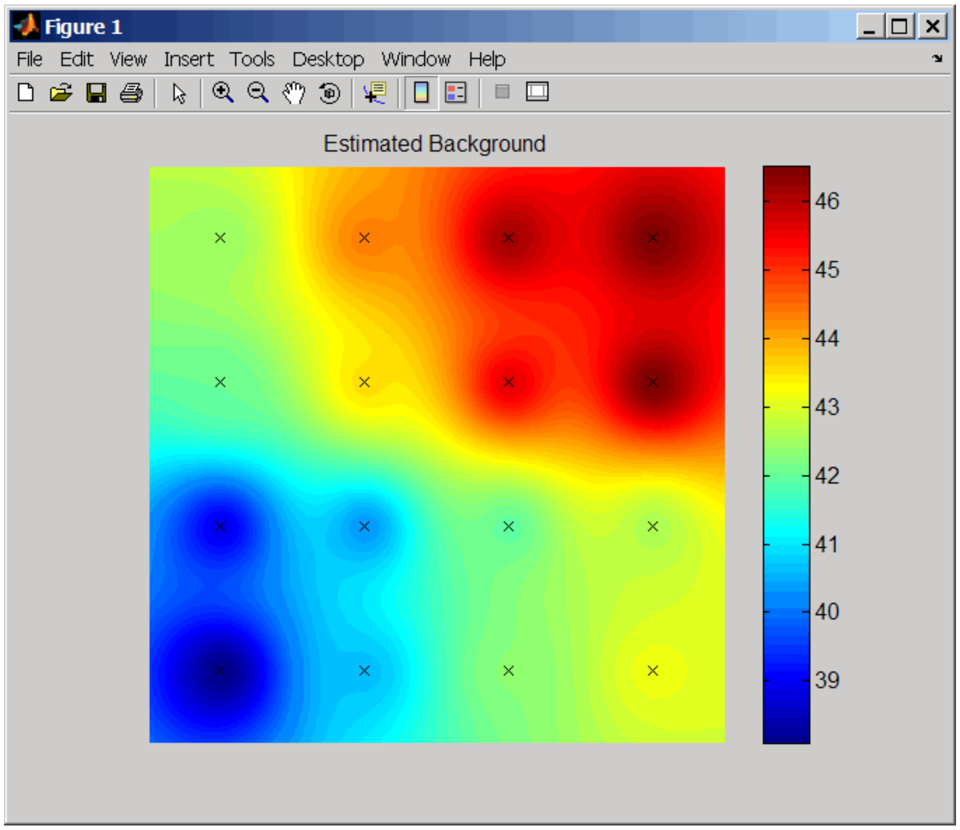
`... = zonebackadj(`*`Data`*`, ...'SmoothFactor', `*`SmoothFactorValue`*`, ...)` specifies the smoothing factor used in the calculation of the weighted average of the contributions of each zone to the background of a point, thus providing a smooth transition between zones. Default is `100`.

`... = zonebackadj(`*`Data`*`, ...'NoiseFrac', `*`NoiseFracValue`*`, ...)` specifies the noise fraction, such that the background-adjusted value is given by `max((Intensity - WeightedBackground), NF*LocalNoiseEstimate)`, where `NF` is *`NoiseFracValue`*. Default is `0.5`.

`... = zonebackadj(`*`Data`*`, ...'CDF', `*`CDFValue`*`, ...)` specifies an Affymetrix CDF library file or structure, which specifies control cells, which are not used in the background estimates.

`... = zonebackadj(`*`Data`*`, ...'Mask', `*`MaskValue`*`, ...)` specifies a logical vector of that specifies which cells to mask and not use in the background estimates. In the vector, `0 = not masked` and `1 = masked`. Defaults are the values in the `Masked` column of the `Probes` field of the CEL file.

`... = zonebackadj(`*`Data`*`, ...'Showplot', `*`ShowplotValue`*`, ...)` plots an image of the background estimates. Choices are `true` or `false` (default).



Examples

Retrieve Affymetrix™ Probe Set Intensity Values

For this example, you need some sample data files from here. This example uses the sample data from the *E. coli* Antisense Genome Array. Extract the data files from the DTT archive using the Data Transfer Tool.

You also need to download the corresponding library file for the sample. For this example, Ecoli_ASv2.CDF is used as for the *E. coli* Antisense Genome Array. You may already have these files if you have any Affymetrix GeneChip software installed on your machine. If not, get the library files by downloading and unzipping the *E. coli* Antisense Genome Array zip file.

Read the contents of a CEL file into a MATLAB structure.

```
celStruct = affyread('Ecoli-antisense-121502.CEL');
```

Read the contents of a CDF file into a MATLAB structure.

```
cdfStruct = affyread('C:\LibFiles\Ecoli_ASv2.CDF');
```

Use the `zonebackadj` function to return a matrix or cell array of vectors containing the estimated background values for each probe.

```
[baData,zones,background] = zonebackadj(celStruct,'cdf',cdfStruct);
```

Create a table of intensity values for the `argG_b3172_at` probe set.

```
psvals = probesetvalues(celStruct, cdfStruct, 'argG_b3172_at',...
         'background',background)
```

```
psvals =

   1.0e+03 *

  Columns 1 through 7

    5.2120         0         0    0.0454    0.4300    0.1770    0.1690
    5.2120    0.0010         0    0.0455    0.4310    0.1770    0.1273
    5.2120    0.0020         0    0.0455    0.4320    0.1770    0.1270
    5.2120    0.0030         0    0.0455    0.4330    0.1770    0.1333
    5.2120    0.0040         0    0.0455    0.4340    0.1770    0.2123
    5.2120    0.0050         0    0.0455    0.4350    0.1770    0.1495
    5.2120    0.0060         0    0.0455    0.4360    0.1770    0.0503
    5.2120    0.0070         0    0.0456    0.4370    0.1770    0.1525
    5.2120    0.0080         0    0.0456    0.4380    0.1770    0.1645
    5.2120    0.0090         0    0.0456    0.4390    0.1770    0.1260
    5.2120    0.0100         0    0.0456    0.4400    0.1770    0.0540
    5.2120    0.0110         0    0.0456    0.4410    0.1770    0.0833
    5.2120    0.0120         0    0.0457    0.4420    0.1770    0.0955
    5.2120    0.0130         0    0.0457    0.4430    0.1770    0.1100
    5.2120    0.0140         0    0.0457    0.4440    0.1770    0.2510

  Columns 8 through 14

    0.0354    0.0250         0         0    0.4300    0.1780    0.1635
    0.0218    0.0300         0         0    0.4310    0.1780    0.1003
    0.0237    0.0300         0         0    0.4320    0.1780    0.1750
    0.0259    0.0360         0         0    0.4330    0.1780    0.0940
    0.0433    0.0360         0         0    0.4340    0.1780    0.1718
    0.0275    0.0360         0         0    0.4350    0.1780    0.1540
    0.0112    0.0300         0         0    0.4360    0.1780    0.0460
    0.0377    0.0360         0         0    0.4370    0.1780    0.1070
    0.0312    0.0360         0         0    0.4380    0.1780    0.0973
    0.0234    0.0360         0         0    0.4390    0.1780    0.1213
    0.0112    0.0360         0         0    0.4400    0.1780    0.0540
    0.0174    0.0360         0         0    0.4410    0.1780    0.0623
    0.0171    0.0300         0         0    0.4420    0.1780    0.0840
    0.0196    0.0360         0         0    0.4430    0.1780    0.0925
    0.0460    0.0360         0         0    0.4440    0.1780    0.1118

  Columns 15 through 20

    0.0241    0.0300         0         0    0.0010    0.0020
    0.0146    0.0360         0         0    0.0010    0.0020
    0.0286    0.0360         0         0    0.0010    0.0020
    0.0227    0.0300         0         0    0.0010    0.0020
    0.0365    0.0300         0         0    0.0010    0.0020
    0.0303    0.0300         0         0    0.0010    0.0020
    0.0098    0.0250         0         0    0.0010    0.0020
    0.0210    0.0360         0         0    0.0010    0.0020
    0.0219    0.0360         0         0    0.0010    0.0020
    0.0253    0.0360         0         0    0.0010    0.0020
    0.0129    0.0360         0         0    0.0010    0.0020
    0.0125    0.0360         0         0    0.0010    0.0020
    0.0186    0.0300         0         0    0.0010    0.0020
    0.0220    0.0360         0         0    0.0010    0.0020
    0.0207    0.0360         0         0    0.0010    0.0020
```

Version History

Introduced in R2007b

References

[1] Statistical Algorithms Description Document, https://tools.thermofisher.com/content/sfs/brochures/sadd_whitepaper.pdf

See Also

`affyinvarsetnorm` | `affyread` | `celintensityread` | `gcrma` | `gcrmabackadj` | `probelibraryinfo` | `probesetlookup` | `probesetvalues` | `quantilenorm` | `rmabackadj` | `rmasummary`

Topics

"Working with Affymetrix Data"

Biopipeline Designer

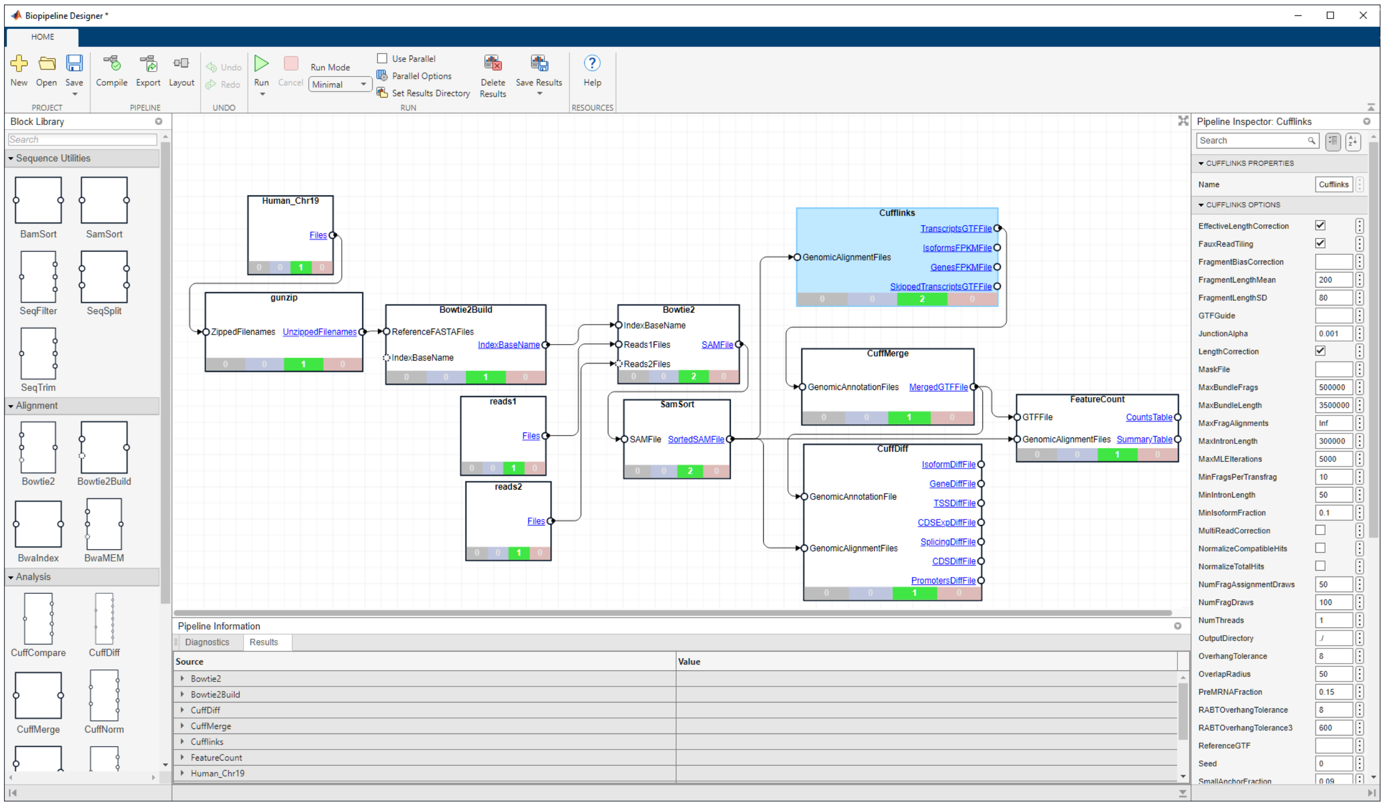
Build and run bioinformatics pipelines

Description

The **Biopipeline Designer** app lets you build and run end-to-end bioinformatics workflows interactively. It provides a block diagram editor to build the pipeline using built-in or custom blocks, where each block is a step that is needed in your workflow to reach the final goal of your analysis. For example, the goal could be to identify differentially expressed genes from RNA-Seq data. Some of the steps needed for such analysis could be preprocessing of read count data and counting genetic features, and you can define individual blocks to perform these necessary steps and connect such blocks to form a complete pipeline.

Using the app, you can create various bioinformatics pipelines to analyze genomic data. For instance, you can:

- Filter read sequences based on some criteria, such as read quality or length.
- Map reads to a reference using the built-in `Bowtie2` and `BwaMEM` blocks.
- Count the number of reads mapped to genomic features.
- Assemble transcriptomes, quantify transcript expression profiles, and identify significant changes in transcript expression using `Cufflinks`, `CuffQuant`, `CuffDiff`, and other related blocks.
- Create a custom block to represent any arbitrary MATLAB function and use in your pipeline, such as for performing differential analysis on RNA-seq count data or plotting the analysis results.



Pipeline Inspector: Cufflinks

CUFFLINKS PROPERTIES

Name	Cufflinks

CUFFLINKS OPTIONS

EffectiveLengthCorrection	✓
FauxReadTiling	✓
FragmentBiasCorrection	
FragmentLengthMean	200
FragmentLengthSD	80
GTFGuide	
JunctionAlpha	0.001
LengthCorrection	✓
MaskFile	
MaxBundleFrags	500000
MaxBundleLength	3500000
MaxFragAlignments	Inf
MaxIntronLength	300000
MaxMLEIterations	5000
MinFragsPerTransfrag	10
MinIntronLength	50
MinIsoformFraction	0.1
MultiReadCorrection	☐
NormalizeCompatibleHits	☐
NormalizeTotalHits	☐
NumFragAssignmentDraws	50
NumFragDraws	100
NumThreads	1
OutputDirectory	./
OverhangTolerance	8
OverlapRadius	50
PreMRNAFraction	0.15
RABTOverhangTolerance	8
RABTOverhangTolerance3	600
ReferenceGTF	
Seed	0
SmallAnchorFraction	0.09

Source	Value
Bowtie2	
Bowtie2Build	
CuffDiff	
CuffMerge	
Cufflinks	
FeatureCount	
Human_Chr19	

Open the Biopipeline Designer App

MATLAB command prompt: Enter `biopipelineDesigner`.

Examples

Create Simple Pipeline to Plot Sequence Quality Data Using Biopipeline Designer

This example shows how to create a bioinformatics pipeline in the **Biopipeline Designer** app that loads sequence read data, filters some sequences based on quality, and displays the quality statistics of the filtered data.

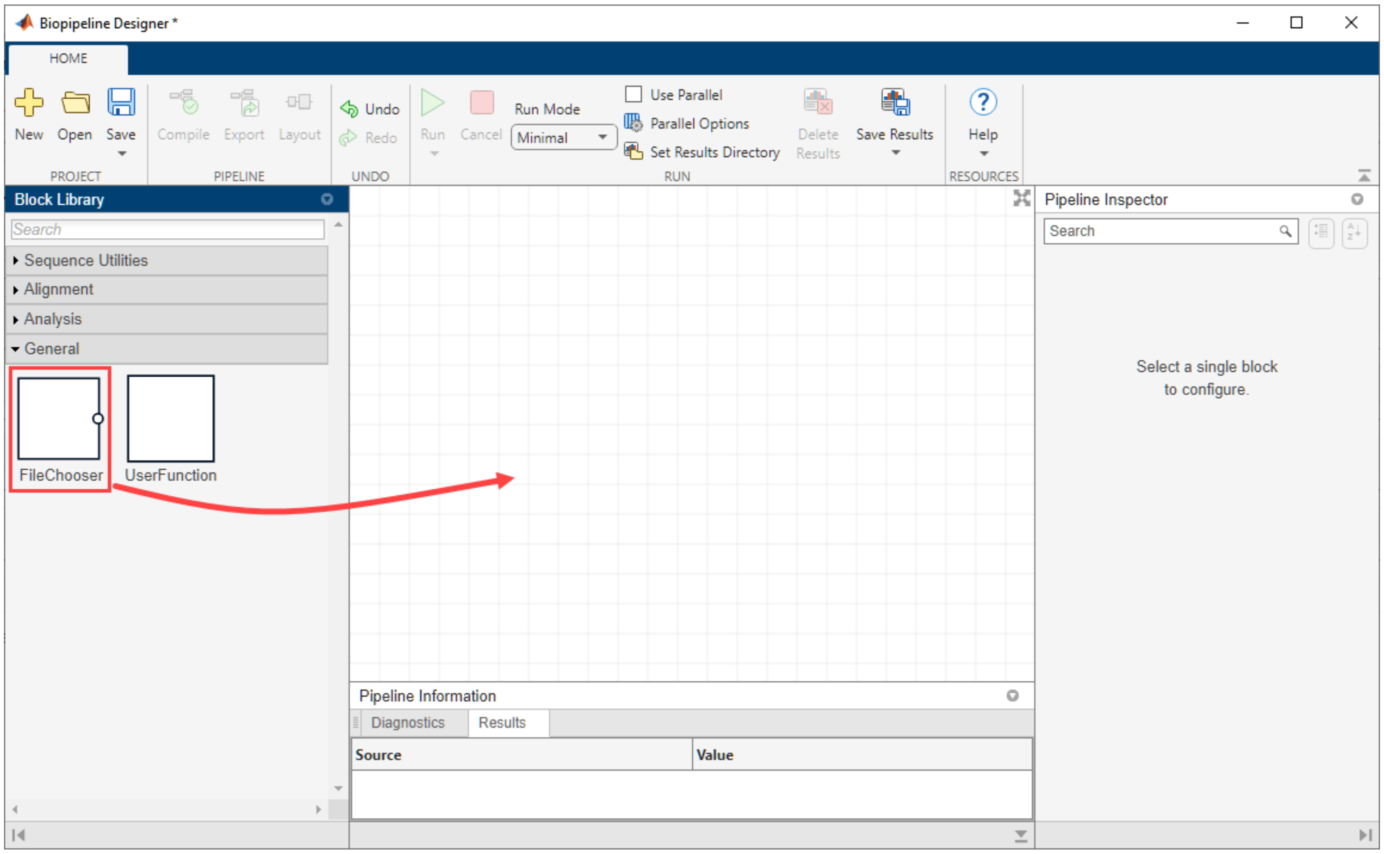
Open Biopipeline Designer App

Enter the following at the MATLAB® command line.

```
biopipelineDesigner
```

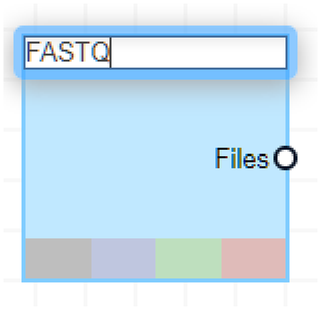
Select Input File Using FileChooser Block

In the **Block Library** panel of the app, scroll down to the **General** section. Drag the **FileChooser** block onto the diagram.



You can also use the **Search** box to look for specific built-in blocks in the **Block Library**.

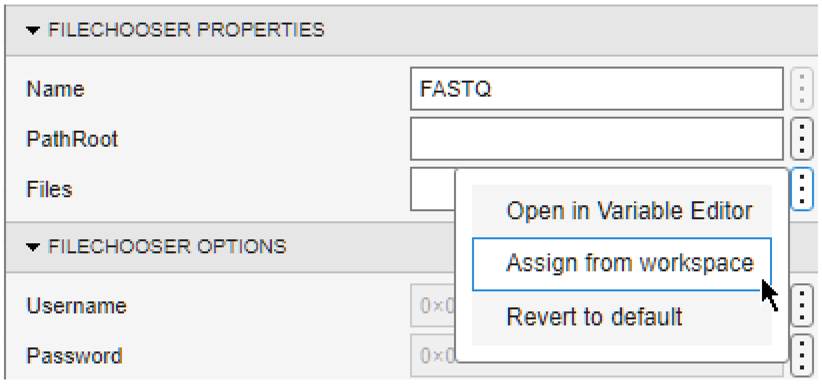
Double-click the block name `FileChooser_1` and rename as `FASTQ`.



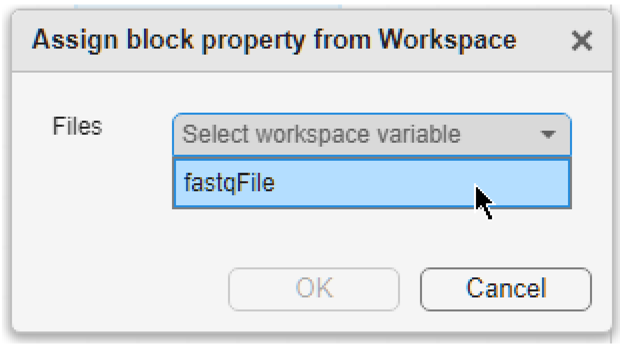
Run the following command at the MATLAB command line to create a variable that contains the full file path to the provided sequence read data.

```
fastqFile = which("SRR005164_1_50.fastq");
```

In the app, click the **FASTQ** block. In the **Pipeline Inspector** pane, under **FileChooser Properties**, click the vertical three-dot menu next to the **Files** property. Select `Assign from workspace`.

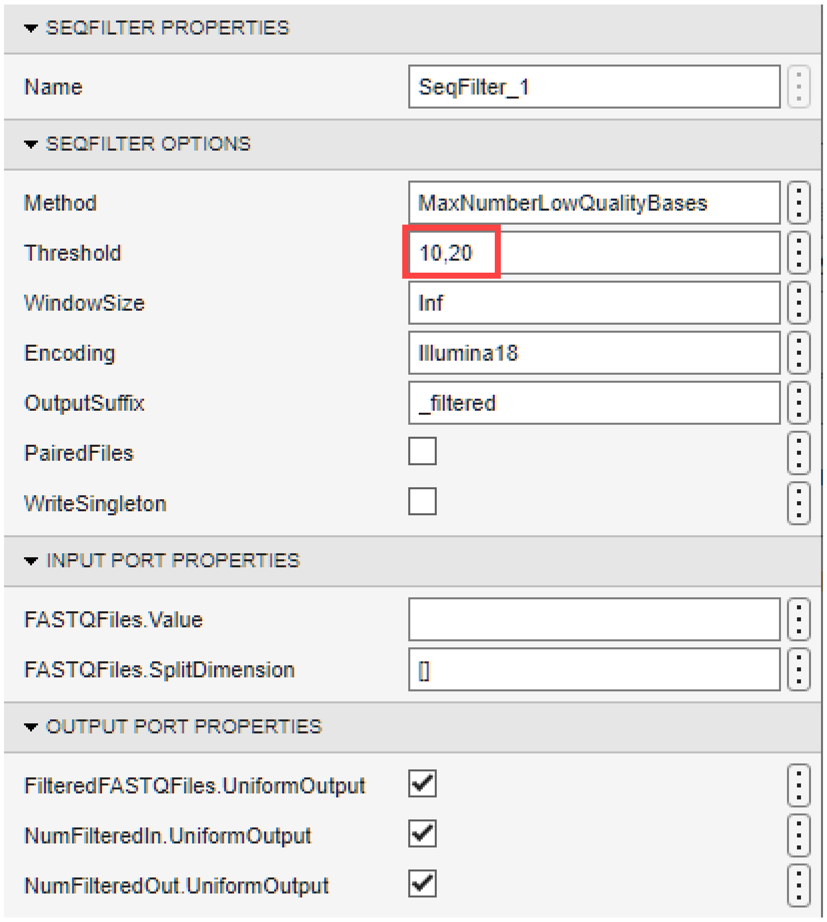


Select `fastqFile` from the list. Click **OK**.


Assign block property from Workspace
Files
Select workspace variable
fastqFile
OK
Cancel

Filter Sequences Based on Quality

In the **Block Library** panel, under the **Sequence Utilities** section, drag the **SeqFilter** block onto the diagram. This block can filter sequences based on some specifications. The **Pipeline Inspector** panel shows the default values of the block properties and filtering options. In the **SeqFilter Options** section, change **Threshold** to `10,20`. Keep the other options as default. This `10,20` threshold value filters out any sequences with more than 10 low quality bases, where a base is considered low quality when its quality score is less than 20. For details, see `SeqFilterOptions`.



▾ SEQFILTER PROPERTIES

Name	SeqFilter_1

▾ SEQFILTER OPTIONS

Method	MaxNumberLowQualityBases
Threshold	10,20
WindowSize	Inf
Encoding	Illumina18
OutputSuffix	_filtered
PairedFiles	☐
WriteSingleton	☐

▾ INPUT PORT PROPERTIES

FASTQFiles.Value	
FASTQFiles.SplitDimension	[]

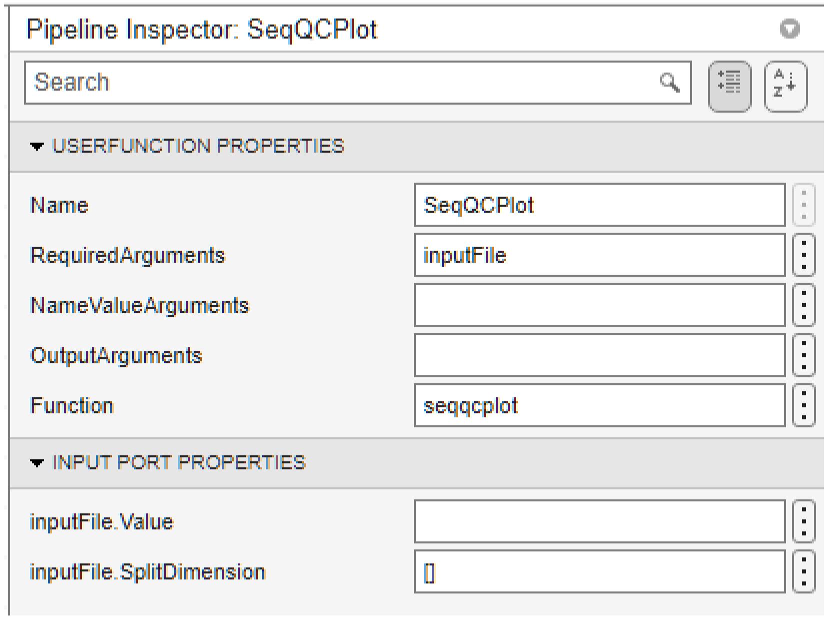
▾ OUTPUT PORT PROPERTIES

FilteredFASTQFiles.UniformOutput	✓
NumFilteredIn.UniformOutput	✓
NumFilteredOut.UniformOutput	✓

Plot Sequence Quality Data

Create a custom (`bioinfo.pipeline.blocks.UserFunction`) block that calls an existing MATLAB function `seqqcplot` to plot the quality statistics of the filtered data.

1. In the **Block Library** panel, under the **General** section, drag and drop the **UserFunction** block onto the diagram.
2. Rename the block to **SeqQCPlot**.
3. In the **Pipeline Inspector** pane, under **UserFunction Properties**, set the **RequiredArguments** to `inputFile` and **Function** to *seqqcplot*.


Pipeline Inspector: SeqQCPlot

Search

USERFUNCTION PROPERTIES

Name	SeqQCPlot
RequiredArguments	inputFile
NameValueArguments	
OutputArguments	
Function	seqqcplot

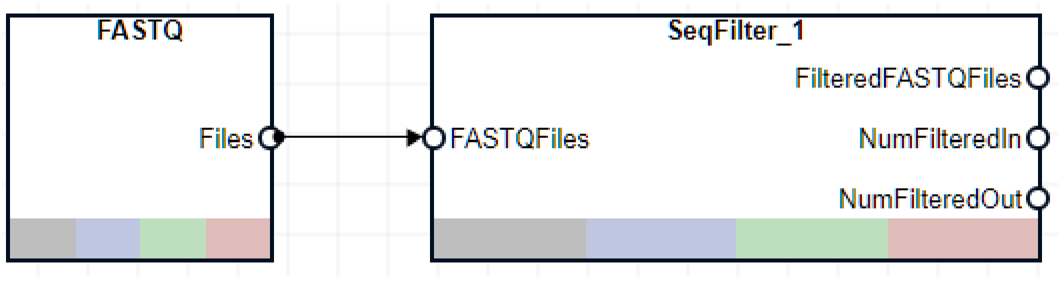
INPUT PORT PROPERTIES

inputFile.Value	
inputFile.SplitDimension	[]

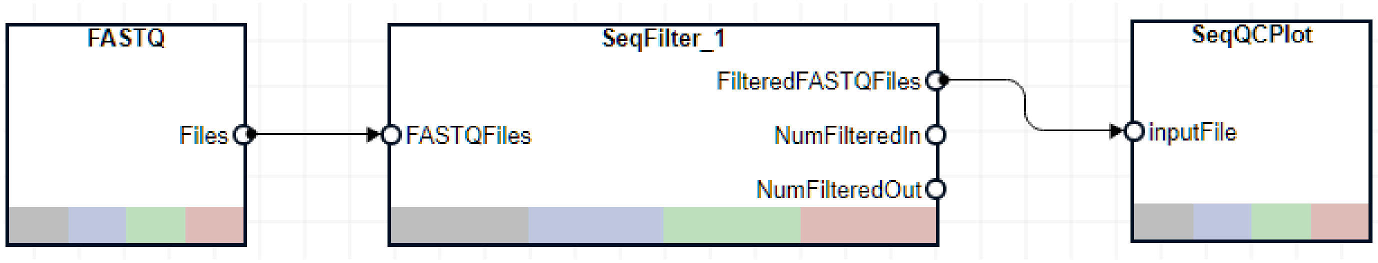
Connect Blocks and Run Pipeline

After setting up the blocks, you can now connect them to complete the pipeline.

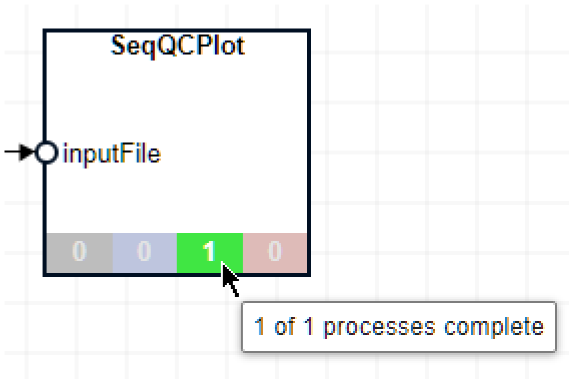
Drag an arrow from the **Files** output port of **FASTQ** to the **FASTQFiles** port of **SeqFilter_1**.



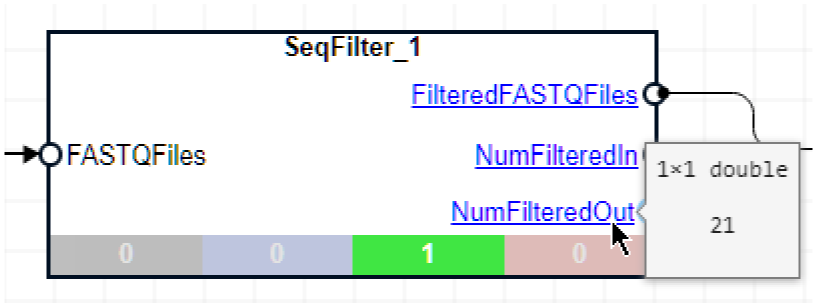
Next connect the **FilteredFASTQFiles** port to **inputFile** port.



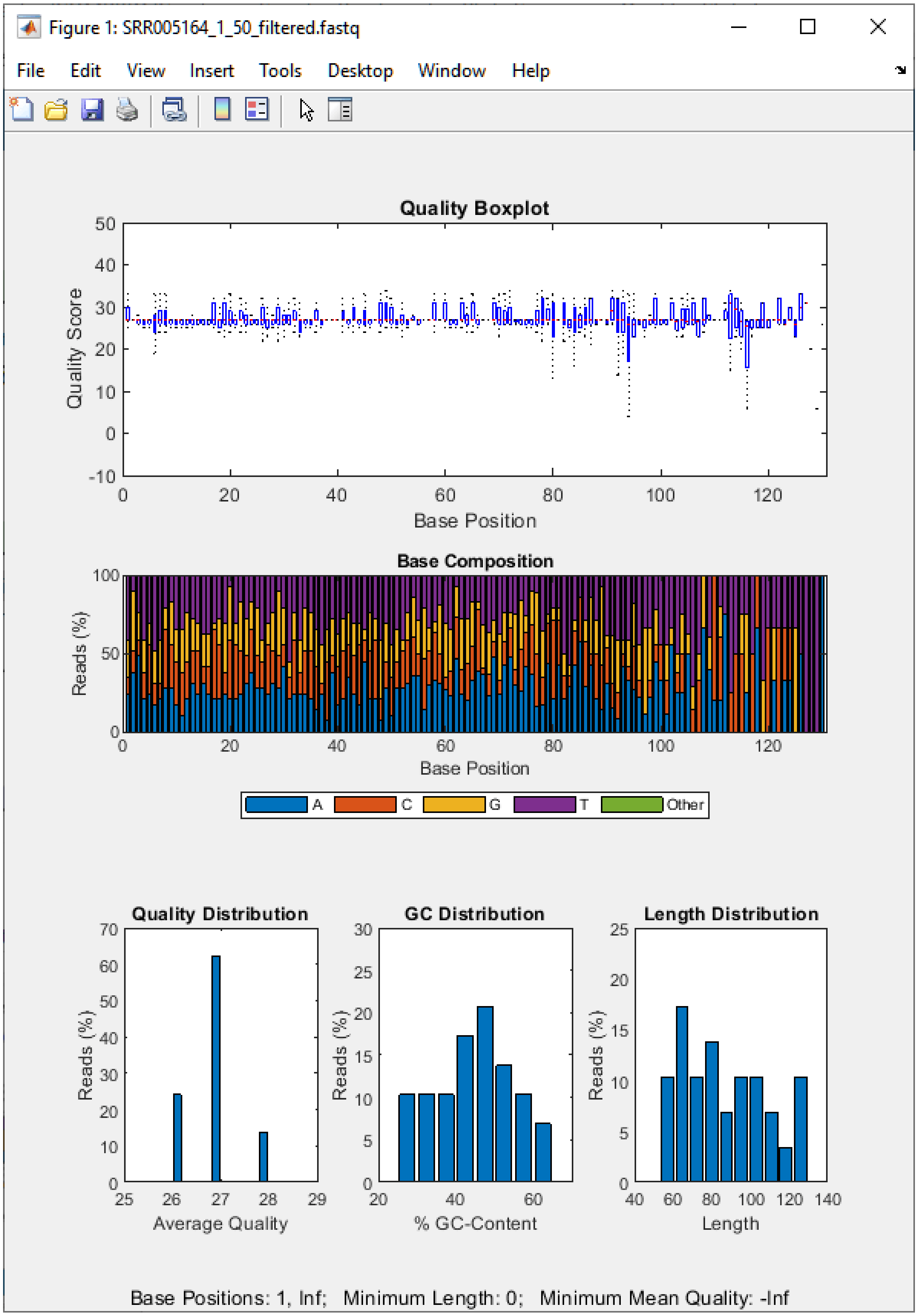
On the toolstrip of the app, click **Run**. During the run, you can see the progress of each block at its status bar. Point to a color-coded section with a number to see its meaning.



After the run, you can click each output port name of a block to see the output value. For example, click **NumFilteredOut** to see the total number of reads that were filtered out by the block.

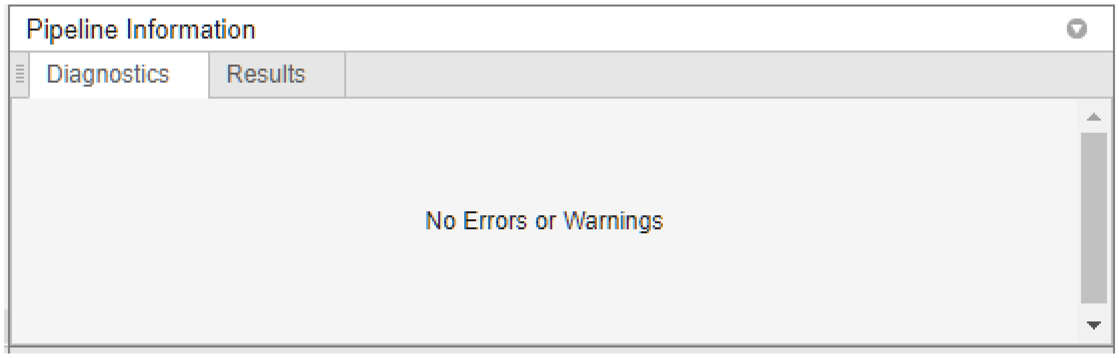


The app generates the following figure, which contains quality statistics plots of the filtered data.



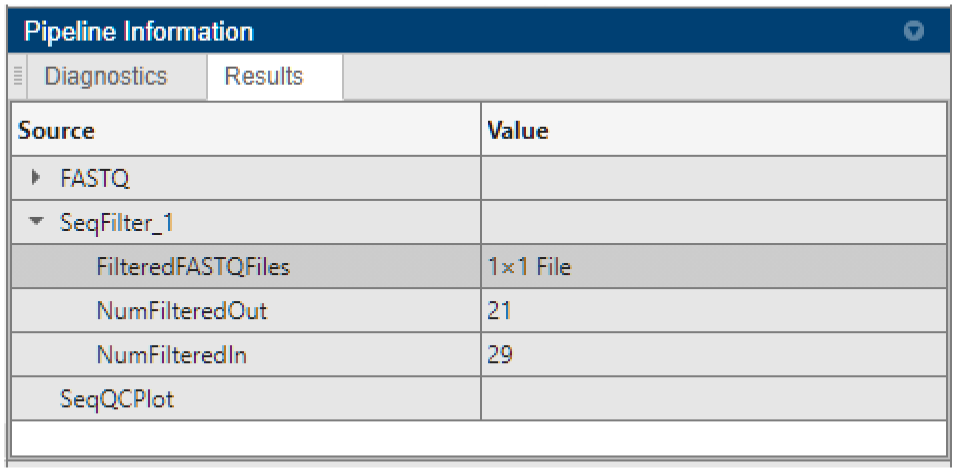
Base Positions: 1, Inf; Minimum Length: 0; Minimum Mean Quality: -Inf

If there are any errors or warnings, the app shows them in the **Diagnostics** tab of the **Pipeline Information** panel, which is at the bottom of the diagram.



No Errors or Warnings

Click the **Results** tab. In the **Source** column, expand **SeqFilter_1** to see the block results, such as the filtered FASTQ file and the number of sequences that are selected and filtered out.

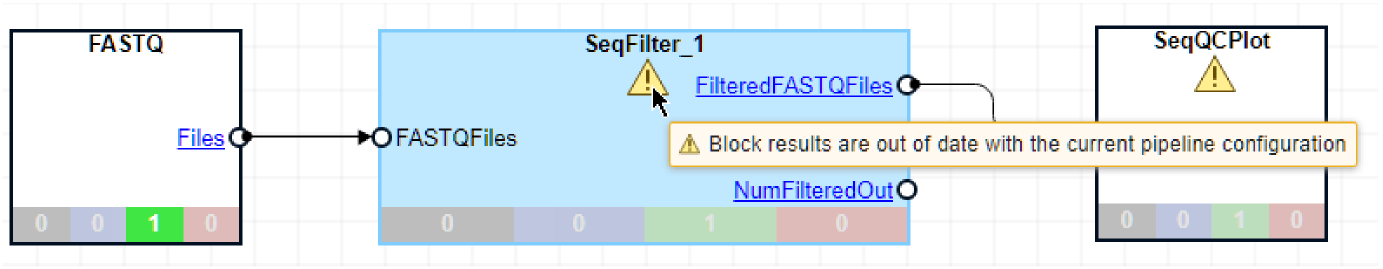


Source	Value
FASTQ	
SeqFilter_1	
FilteredFASTQFiles	1×1 File
NumFilteredOut	21
NumFilteredIn	29
SeqQCPlot	

Rerun Pipeline with Different Filtering Threshold

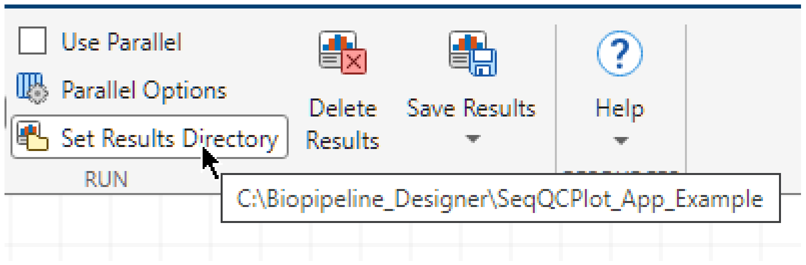
You can specify a different threshold to filter sequences and rerun the pipeline. The app is aware of which blocks in the pipeline have changed and which other blocks, such as downstream blocks, are affected as a result. Hence, on subsequent runs, it reruns only those blocks that are needed, instead of every block in the pipeline. For details, see “Bioinformatics Pipeline Run Mode”.

Click **SeqFilter_1**. In the **Pipeline Inspector** panel, change its **Threshold** option to `5,20`. This setting now filters out any sequence with more than 5 low quality bases, where a base is considered low quality when its score is less than 20. Both **SeqFilter** and **SeqQCPlot** blocks now have a warning icon to indicate that the results are now out of date due to the change to the **SeqFilter** block.



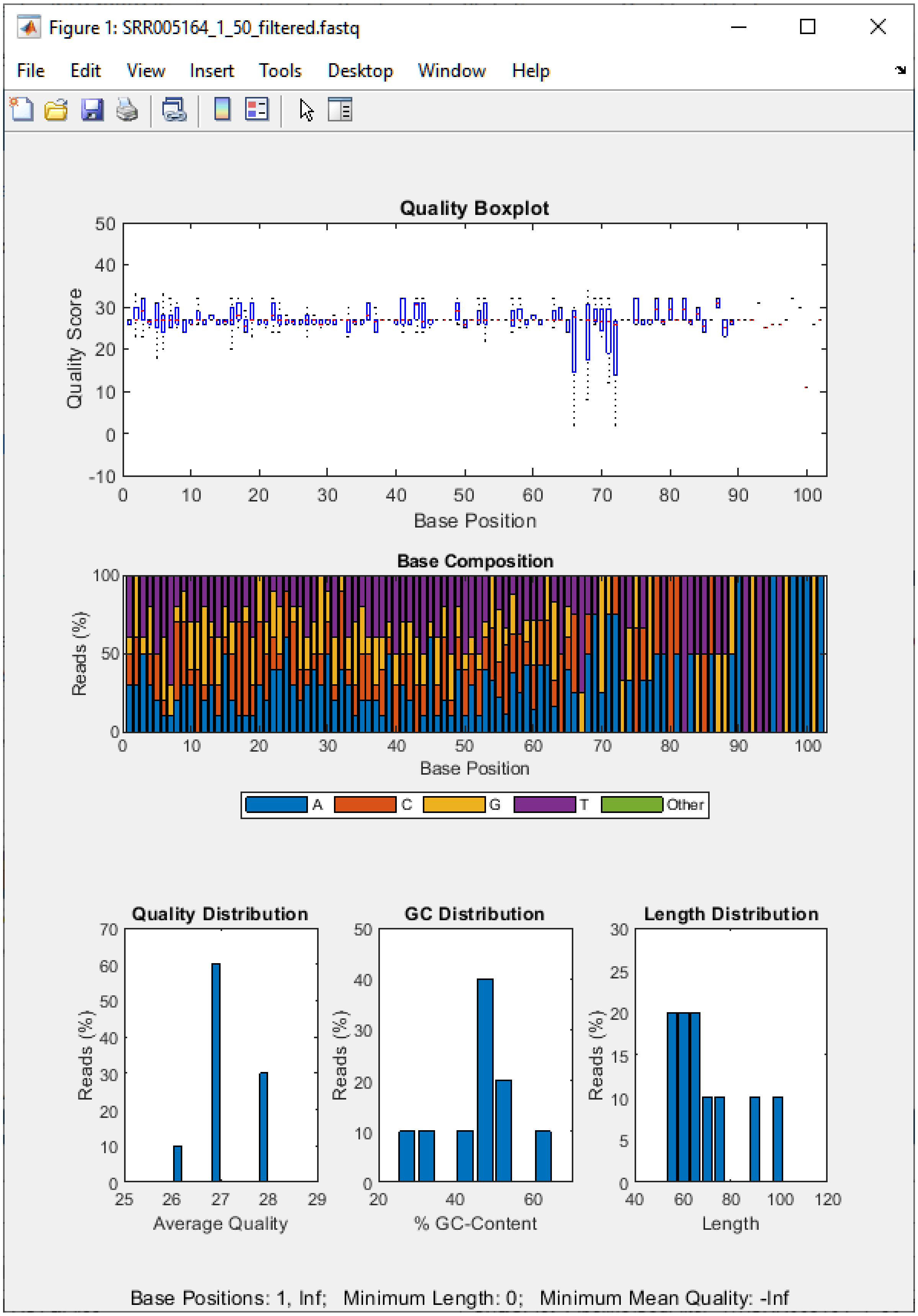
⚠ Block results are out of date with the current pipeline configuration

By default, the app saves the pipeline results in the `PipelineResults` folder in the current directory. It contains the pipeline results from the previous run before you changed the filtering threshold. If you want to save the rerun results to a different folder and avoid overwriting the previous results, you can change the directory location. Click **Set Results Directory** on the **Home** tab and set the directory to a different location, such as `C:\Biopipeline_Designer\SeqQCPlot_App_Example`. If you point to the button, the app shows the directory location.



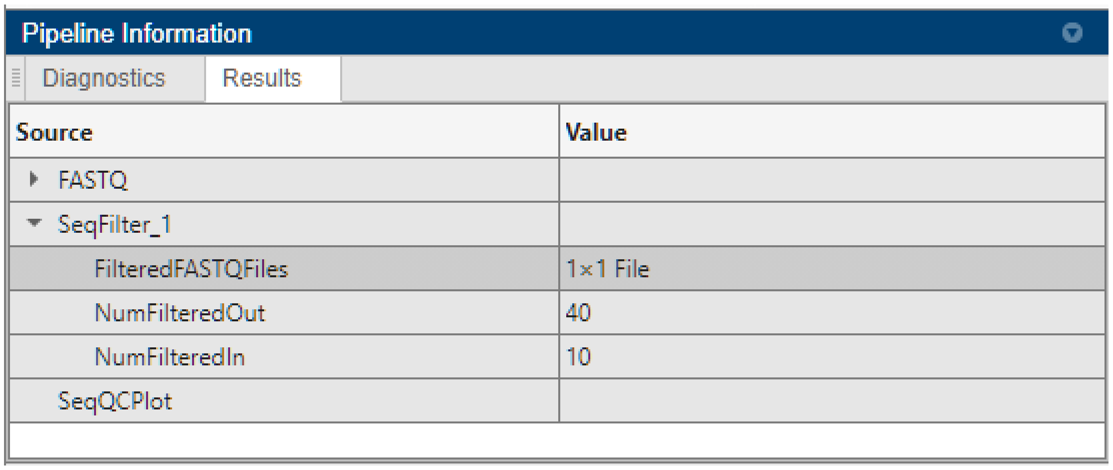
C:\Biopipeline_Designer\SeqQCPlot_App_Example

Click **Run**. The app generates the following figure. During this run, the app does not rerun the **FASTQ** block because it is not needed. It only reruns the other two blocks.



Base Positions: 1, Inf; Minimum Length: 0; Minimum Mean Quality: -Inf

Go to the **Results** tab of the **Pipeline Information** to check the new results.



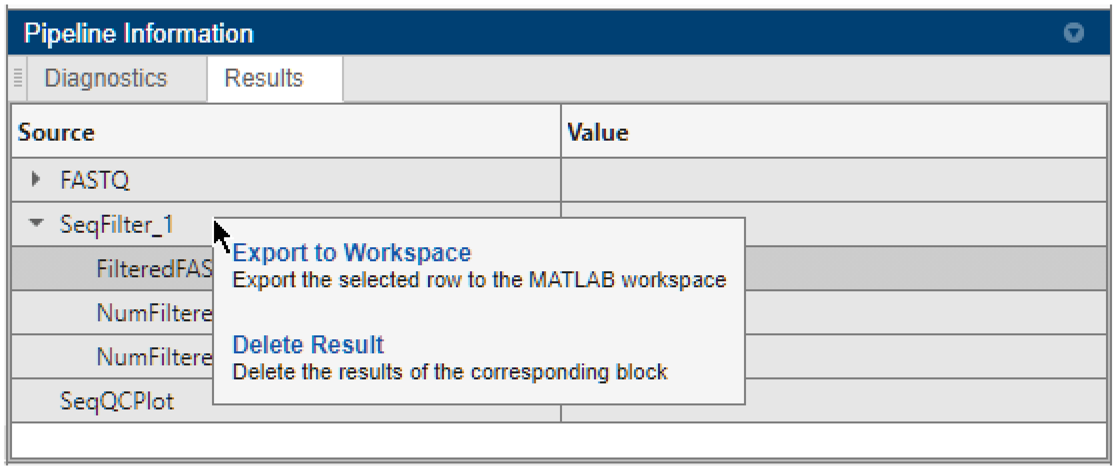
Pipeline Information

Diagnostics | Results

Source	Value
FASTQ	
SeqFilter_1	
FilteredFASTQFiles	1×1 File
NumFilteredOut	40
NumFilteredIn	10
SeqQCPlot	

Export Results

You can export each output of a block or every output of a block to the MATLAB workspace by selecting **Export to Workspace** from the context (right-click) menu of the corresponding row in the **Results** table. To export all outputs of a block, right-click at the block level.



- “Count RNA-Seq Reads Using Biopipeline Designer”

Programmatic Use

`biopipelineDesigner` opens the **Pipeline Designer** app.

`biopipelineDesigner(P)` opens the `bioinfo.pipeline.Pipeline` object `P` in the **Pipeline Designer** app.

`biopipelineDesigner(plprjFile)` opens a bioinformatics pipeline project file `plprjFile` in the **Pipeline Designer** app. `plprjFile` is a string or character vector specifying a file name or path and file name of a pipeline project PLPRJ file. If you specify only a file name, the file must be on the MATLAB search path or in the current folder.

Version History

Introduced in R2023a

See Also

Apps
Genomics Viewer

Objects
`bioinfo.pipeline.Pipeline`

Topics
“Count RNA-Seq Reads Using Biopipeline Designer”
“Bioinformatics Pipeline Run Mode”
“Bioinformatics Pipeline SplitDimension”
“Bioinformatics Toolbox Software Support Packages”

Genomics Viewer

View NGS sequences and annotations

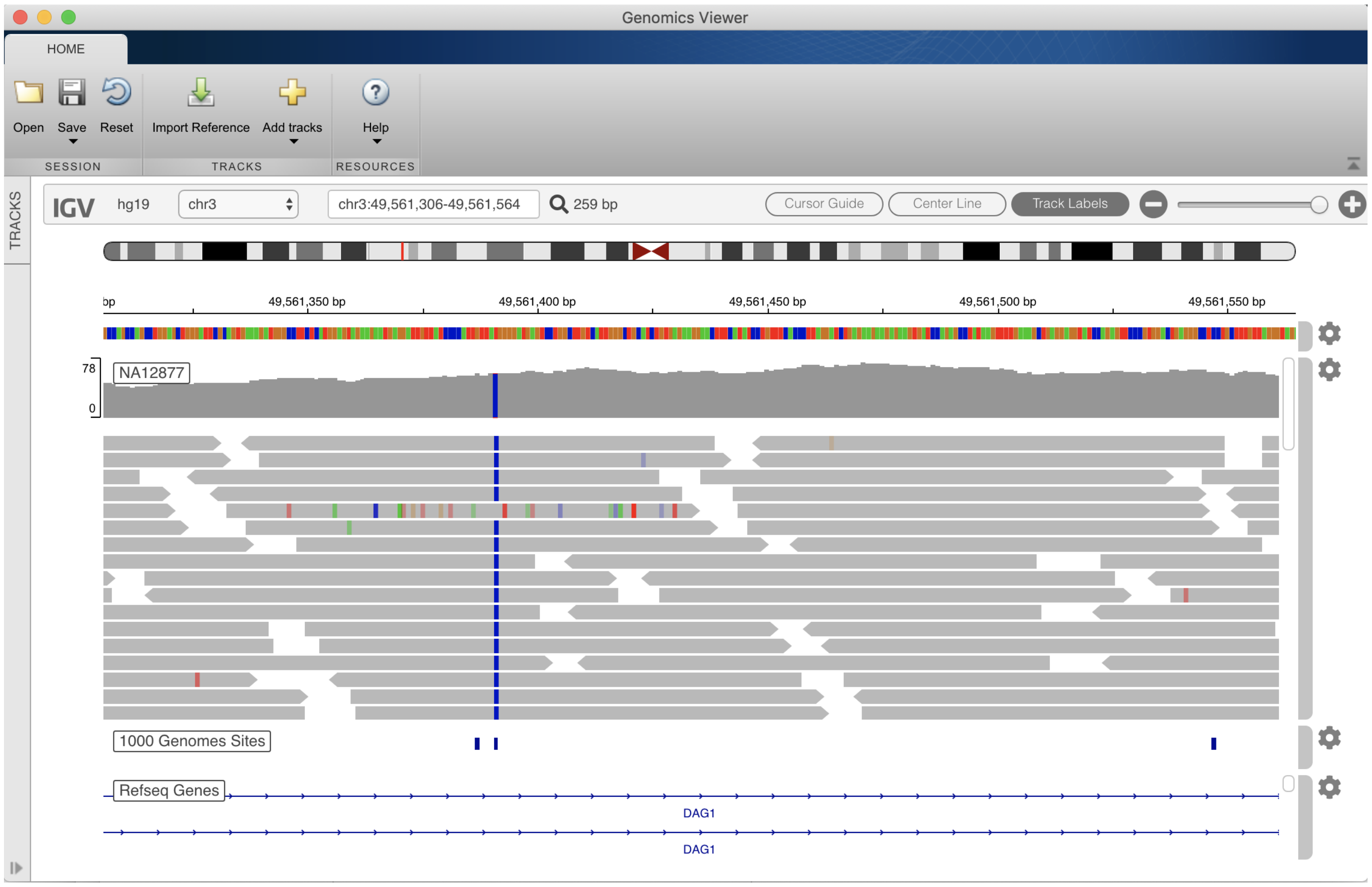
Description

The **Genomics Viewer** app lets you view and explore integrated genomic data with an embedded version of the Integrative Genomics Viewer (IGV) [1][2]. The genomic data include NGS read alignments, genome variants, and segmented copy number data.

Using the app, you can:

- Visualize short-read data (`.BAM` or `.CRAM`) aligned to a reference sequence and compare multiple data sets aligned to a common reference sequence.
- View coverage of different regions of the reference sequence.
- Investigate quality and other details of aligned reads.
- Display nonquantitative genome annotations (`.BED`, `.GFF`, `.GFF3`, and `.GTF`).
- Load structural variants (`.VCF`) and visualize genetic alterations, such as insertions and deletions.
- View segmented copy number data (`.SEG`) and quantitative genomic data (`.WIG`, `.BIGWIG`, and `.BEDGRAPH`), such as ChIP peaks and alignment coverage.

The app requires an internet connection.



Open the Genomics Viewer App

- MATLAB Toolstrip: On the **Apps** tab, under **Computational Biology**, click the app icon.
- MATLAB command prompt: Enter `genomicsViewer`.

Examples

- “Visualize NGS Data Using Genomics Viewer App”

Version History

Introduced in R2019b

References

[1] Robinson, J., H. Thorvaldsdóttir, W. Winckler, M. Guttman, E. Lander, G. Getz, J. Mesirov. 2011. Integrative Genomics Viewer. *Nature Biotechnology*. 29:24–26.

[2] Thorvaldsdóttir, H., J. Robinson, J. Mesirov. 2013. Integrative Genomics Viewer (IGV): High-performance genomics data visualization and exploration. *Briefings in Bioinformatics*. 14:178–192.

See Also

Apps
Sequence Alignment | Sequence Viewer

Topics
“Visualize NGS Data Using Genomics Viewer App”

Molecule Viewer

(To be removed) Display and manipulate 3-D molecule structure

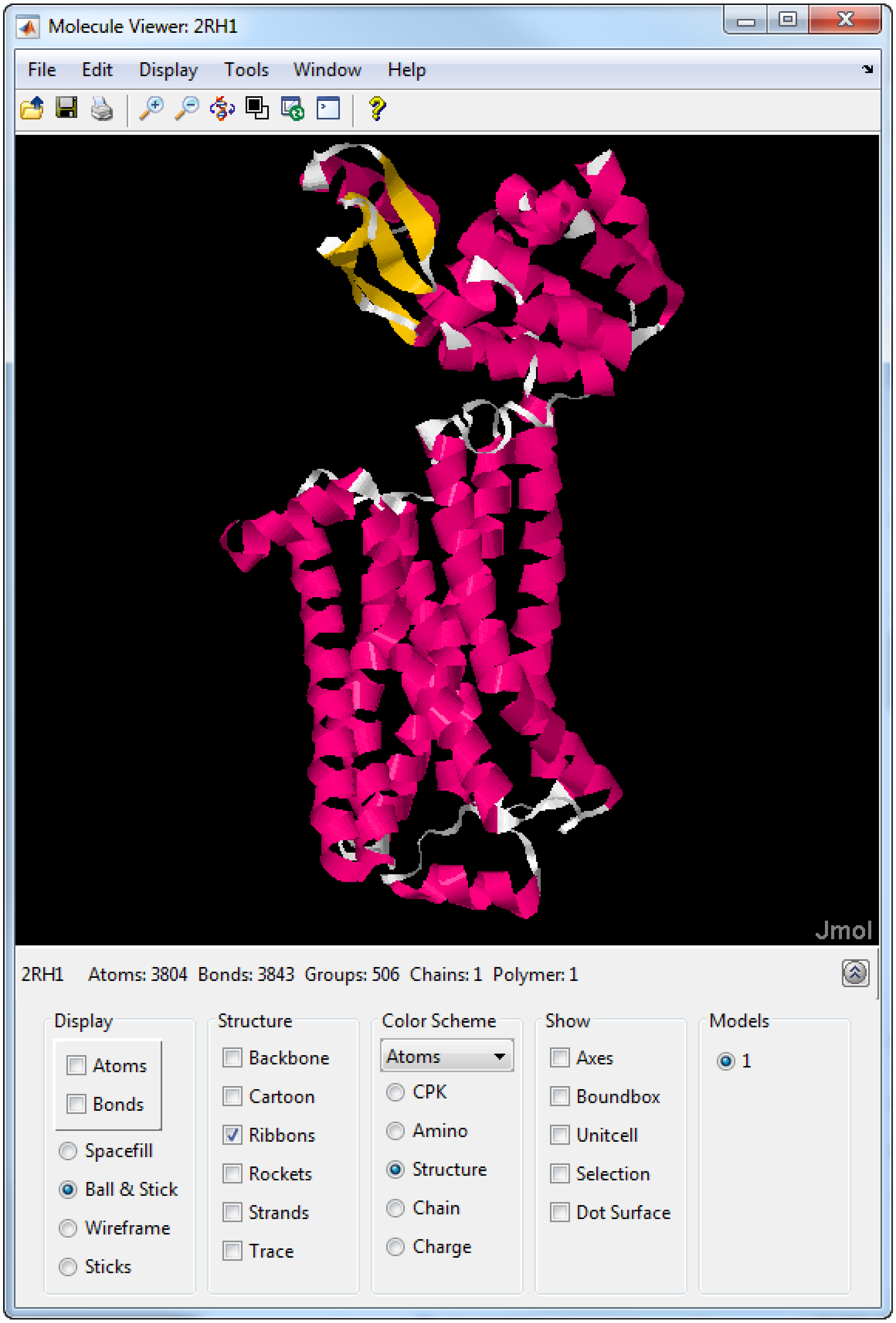
Note The app will be removed in a future release.

Description

The **Molecule Viewer** app lets you display and manipulate 3-D molecular structures.

You can:

- Import structural information directly from the Protein Data Bank (PDB) database or other supported files.
- Measure distances and dihedral angles.
- Display molecular surfaces, such as van der Waals or solvent-accessible surfaces.
- Select different visualization and color schemes to display a molecule, such as the ribbon or backbone representation.
- Run RasMol script commands from within the app.



2RH1 Atoms: 3804 Bonds: 3843 Groups: 506 Chains: 1 Polymer: 1

Display
- Atoms [ ]
- Bonds [ ]
- Spacefill
- [✓ Ball & Stick]
- Wireframe
- Sticks

Structure
- Backbone [ ]
- Cartoon [ ]
- Ribbons [✓]
- Rockets [ ]
- Strands [ ]
- Trace [ ]

Color Scheme
- Atoms
- CPK
- Amino
- [✓ Structure]
- Chain
- Charge

Show
- Axes [ ]
- Boundbox [ ]
- Unitcell [ ]
- Selection [ ]
- Dot Surface [ ]

Models
- [✓ 1]

Open the Molecule Viewer App

- MATLAB Toolstrip: On the **Apps** tab, under **Computational Biology**, click the app icon.
- MATLAB command prompt: Enter `molviewer`.

Programmatic Use

`molviewer` opens the Molecule Viewer app.

`molviewer(file)` reads the structural information from `file` and shows the 3-D molecular structure in the Molecule Viewer app.

`molviewer(pdbID)` retrieves the structural data for a protein from the PDB database using its `pdbID` and shows the 3-D molecular structure in the Molecule Viewer app.

Version History

Introduced in R2007a

R2023a: Warns
Warns starting in R2023a

The app issues a warning that it will be removed in a future release.

R2020b: To be removed
Not recommended starting in R2020b

The app runs without warning. But it will be removed in a future release.

Sequence Alignment

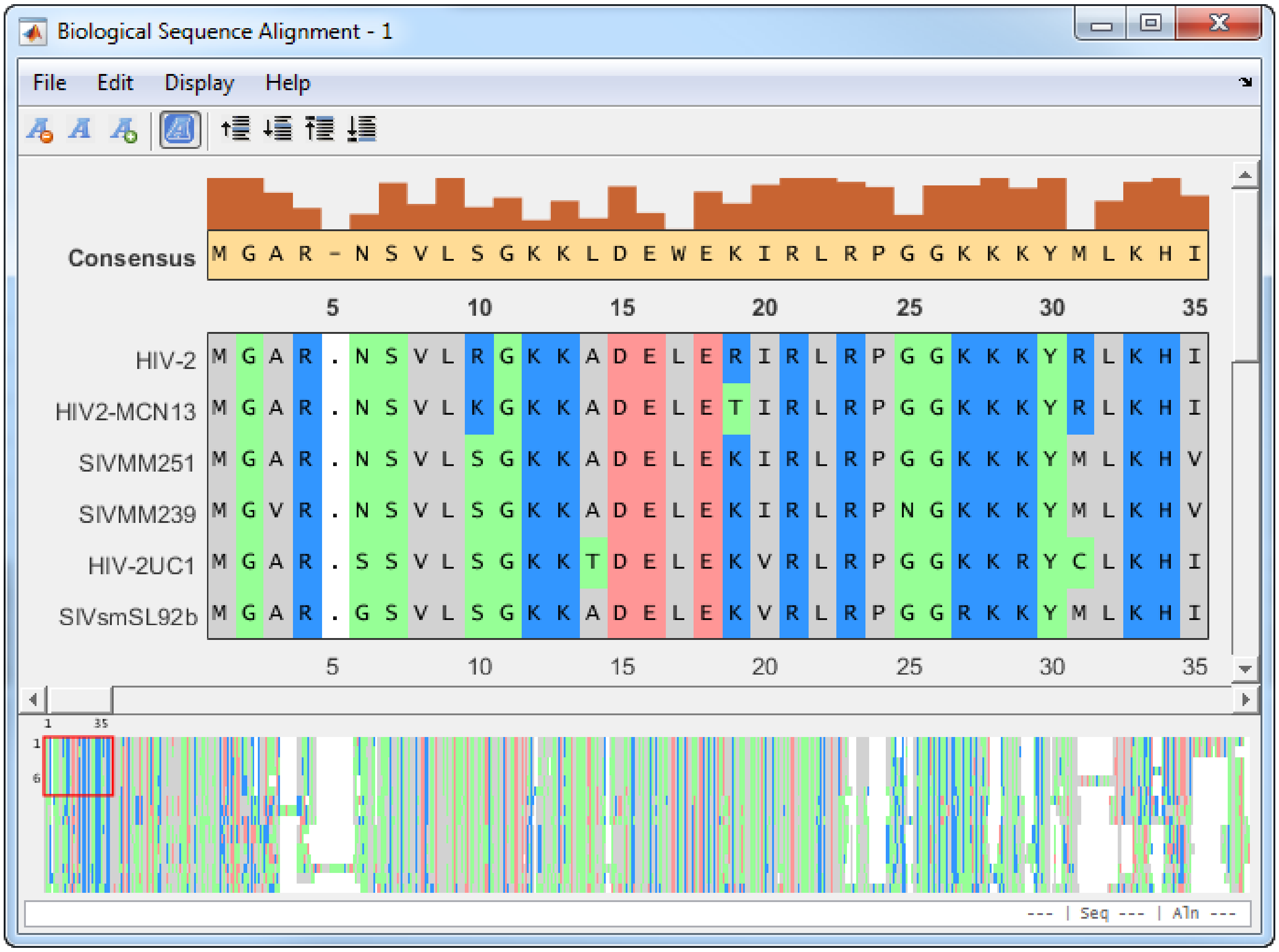
Visualize and edit multiple sequence alignments

Description

The **Sequence Alignment** app lets you visualize and edit multiple sequence alignments.

You can:

- Inspect the sequence alignment and make manual adjustments.
- View the consequence sequence information and export it to a file or MATLAB workspace.
- Generate a phylogenetic tree from aligned sequences.



```
                      5    10   15   20   25   30   35
Consensus   M G A R - N S V L S G K K L D E W E K I R L R P G G K K K Y M L K H I
HIV-2       M G A R . N S V L R G K K A D E L E R I R L R P G G K K K Y R L K H I
HIV2-MCN13  M G A R . N S V L K G K K A D E L E T I R L R P G G K K K Y R L K H I
SIVMM251    M G A R . N S V L S G K K A D E L E K I R L R P G G K K K Y M L K H V
SIVMM239    M G V R . N S V L S G K K A D E L E K I R L R P N G K K K Y M L K H V
HIV-2UC1    M G A R . S S V L S G K K T D E L E K V R L R P G G K K R Y C L K H I
SIVsmSL92b  M G A R . G S V L S G K K A D E L E K V R L R P G G R K K Y M L K H I
```

--- | Seq --- | Aln ---

Open the Sequence Alignment App

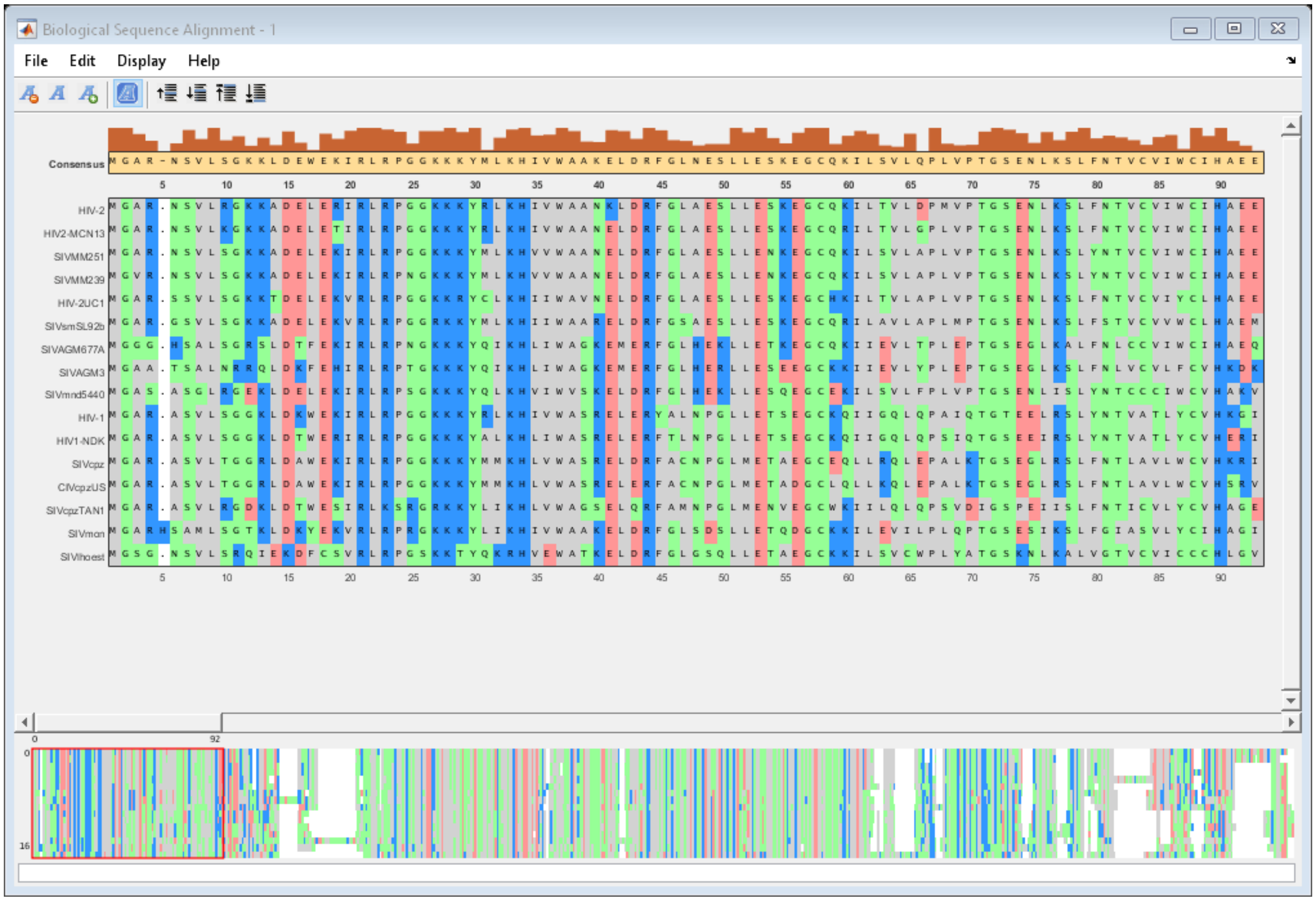
- MATLAB Toolstrip: On the **Apps** tab, under **Computational Biology**, click the app icon.
- MATLAB command prompt: Enter `seqalignviewer`.

Examples

View a Multiple Sequence Alignment File

Load and view a multiple sequence alignment file.

```
seqalignviewer('aagag.aln')
```



```
Consensus   MGAR-NSVLSGKKLDEWEKIRLRPGGKKKYMLKHIVWAAKELDRFGLNESLLESKEGCQKILSVLQPLVPTGSENLKSLFNTVCVIWCIHAEE
HIV-2       MGAR.NSVLRGKKADELERIRLRPGGKKKYRLKHIVWAANKLDRFGLAESLLESKEGCQKILTVLDPMVPTGSENLKSLFNTVCVIWCIHAEE
HIV2-MCN13  MGAR.NSVLKGKKADELETIRLRPGGKKKYRLKHIVWAANELDRFGLAESLLESKEGCQRILTVLGPLVPTGSENLKSLFNTVCVIWCIHAEE
SIVMM251    MGAR.NSVLSGKKADELEKIRLRPGGKKKYMLKHVVWAANELDRFGLAESLLENKEGCQKILSVLAPLVPTGSENLKSLYNTVCVIWCIHAEE
SIVMM239    MGVR.NSVLSGKKADELEKIRLRPNGKKKYMLKHVVWAANELDRFGLAESLLENKEGCQKILSVLAPLVPTGSENLKSLYNTVCVIWCIHAEE
HIV-2UC1    MGAR.SSVLSGKKTDELEKVRLRPGGKKRYCLKHIIWAVNELDRFGLAESLLESKEGCHKILTVLAPLVPTGSENLKSLFNTVCVIYCLHAEE
SIVsmSL92b  MGAR.GSVLSGKKADELEKVRLRPGGRKKYMLKHIIWAARELDRFGSAESLLESKEGCQRILAVLAPLMPTGSENLKSLFSTVCVVWCLHAEM
SIVAGM677A  MGGG.HSALSGRSLDTFEKIRLRPNGKKKYQIKHLIWAGKEMERFGLHEKLLETKEGCQKIIEVLTPLEPTGSEGLKALFNLCCVIWCIHAEQ
SIVAGM3     MGAA.TSALNRRQLDKFEHIRLRPTGKKKYQIKHLIWAGKEMERFGLHERLLESEEGCKKIIEVLYPLEPTGSEGLKSLFNLVCVLFCVHKDK
SIVmnd5440  MGAS.ASGLRGEKLDELEKIRLRPSGKKKYQLKHVIWVSKELDRFGLHEKLLESQEGCEKILSVLFPLVPTGSENLISLYNTCCCIWCVHAKV
HIV-1       MGAR.ASVLSGGKLDKWEKIRLRPGGKKKYRLKHIVWASRELERYALNPGLLETSEGCKQIIGQLQPAIQTGTEELRSLYNTVATLYCVHKGI
HIV1-NDK    MGAR.ASVLSGGKLDTWERIRLRPGGKKKYALKHLIWASRELERFTLNPGLLETSEGCKQIIGQLQPSIQTGSEEIRSLYNTVATLYCVHERI
SIVcpz      MGAR.ASVLTGGRLDAWEKIRLRPGGKKKYMMKHLVWASRELDRFACNPGLMETAEGCEQLLRQLEPALKTGSEGLRSLFNTLAVLWCVHKRI
CIVcpzUS    MGAR.ASVLTGGRLDAWEKIRLRPGGKKKYMMKHLVWASRELERFACNPGLMETADGCLQLLKQLEPALKTGSEGLRSLFNTLAVLWCVHSRV
SIVcpzTAN1  MGAR.ASVLRGDKLDTWESIRLKSRGRKKYLIKHLVWAGSELQRFAMNPGLMENVEGCWKIILQLQPSVDIGSPEIISLFNTICVLYCVHAGE
SIVmon      MGARHSAMLSGTKLDKYEKVRLRPRGKKKYLIKHIVWAAKELDRFGLSDSLLETQDGCKKILEVILPLQPTGSESIKSLFGIASVLYCIHAGI
SIVlhoest   MGSG.NSVLSRQIEKDFCSVRLRPGSKKTYQKRHVEWATKELDRFGLGSQLLETAEGCKKILSVCWPLYATGSKNLKALVGTVCVICCCHLGV
```

Alternatively, you can click Sequence Alignment on the **Apps** tab to open the app, and view the alignment data.

You can also generate a phylogenetic tree from aligned sequences from within the app. Select **Display > View Tree** .

- “View and Align Multiple Sequences”

Programmatic Use

`seqalignviewer` opens the **Sequence Alignment** app.

`seqalignviewer(Alignment)` loads multiple sequence alignment data `Alignment` into the app. `Alignment` can be one of the following:

- A MATLAB structure containing a `Sequence` field, such as returned by `fastaread`, `gethmmalignment`, `multialign`, or `multialignread`
- A MATLAB character array containing MSA data, such as returned by `multialign`
- A string specifying a file or URL that contains MSA data
- A 3-by-N character array showing the pairwise alignment of two sequences, such as the array returned by `nwalign` or `swalign`.

`seqalignviewer('close')` closes the app.

Version History

Introduced in R2012b

See Also

Apps
Sequence Viewer | Genomics Viewer

Functions
`seqalignviewer`

Topics
“View and Align Multiple Sequences”

Phylogenetic Tree

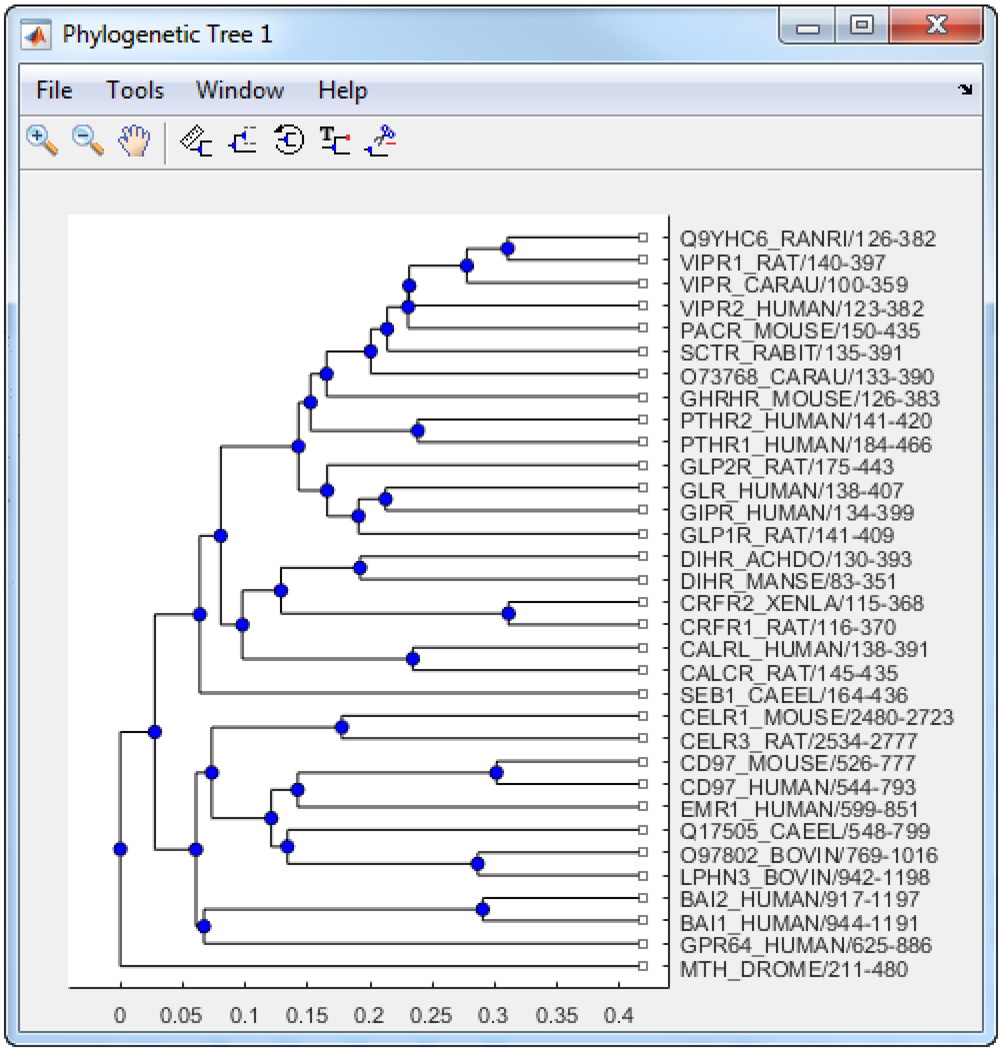
Visualize, edit, and explore phylogenetic tree data

Description

The **Phylogenetic Tree** app lets you visualize and explore phylogenetic tree data.

You can:

- Prune, reorder, and rename branches.
- Collapse and expand branches.
- Inspect and measure the path length between leaf nodes.



Open the Phylogenetic Tree App

- MATLAB Toolstrip: On the **Apps** tab, under **Computational Biology**, click the app icon.
- MATLAB command prompt: Enter `phytreeviewer`.

Examples

- “Using the Phylogenetic Tree App”

Programmatic Use

`phytreeviewer` opens the **Phylogenetic Tree** app.

`phytreeviewer(Tree)` loads a `phytree` object into the app.

`phytreeviewer(File)` loads data from a Newick or ClustalW file into the app.

Version History

Introduced in R2012b

See Also

Apps
Sequence Alignment | **Sequence Viewer** | **Genomics Viewer**

Functions
`phytreeviewer`

Topics
“Using the Phylogenetic Tree App”

Sequence Viewer

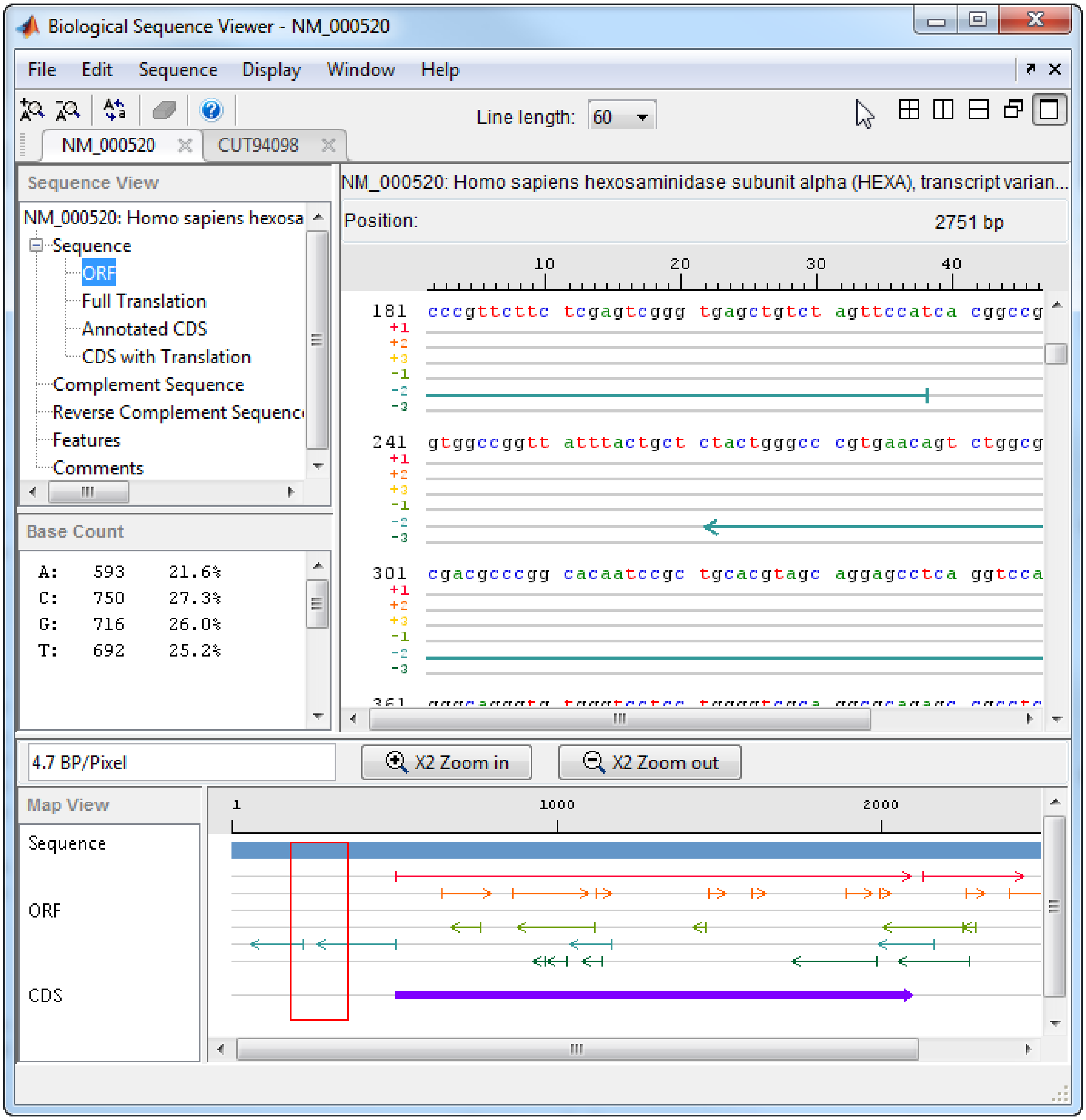
Visualize and interactively explore biological sequences

Description

The **Sequence Viewer** app lets you visualize and explore amino acid or nucleotide sequences.

You can:

- Import sequences from the NCBI or EMBL databases directly.
- View and explore various sequence information (such as ORFs and CDSs) and basic statistics (such as percentages of base counts or amino acid counts).
- View the complement and reverse complement sequences, and other sequence features such as genes and exons.
- Extract protein coding sections of a nucleotide sequence and export them to the MATLAB workspace.
- Search for characteristic words or sequence patterns using regular expressions.



Line length: 60

NM_000520: Homo sapiens hexosa

NM_000520: Homo sapiens hexosaminidase subunit alpha (HEXA), transcript varian...

Position: 2751 bp

```
181  cccgttcttc tcgagtcggg tgagctgtct agttccatca cggccg
241  gtggccggtt atttactgct ctactgggcc cgtgaacagt ctggcg
301  cgacgcccgg cacaatccgc tgcacgtagc aggagcctca ggtcca
```

```
A:   593   21.6%
C:   750   27.3%
G:   716   26.0%
T:   692   25.2%
```

4.7 BP/Pixel

Open the Sequence Viewer App

- MATLAB Toolstrip: On the **Apps** tab, under **Computational Biology**, click the app icon.
- MATLAB command prompt: Enter `seqviewer`.

Examples

Open and View a Biological Sequence

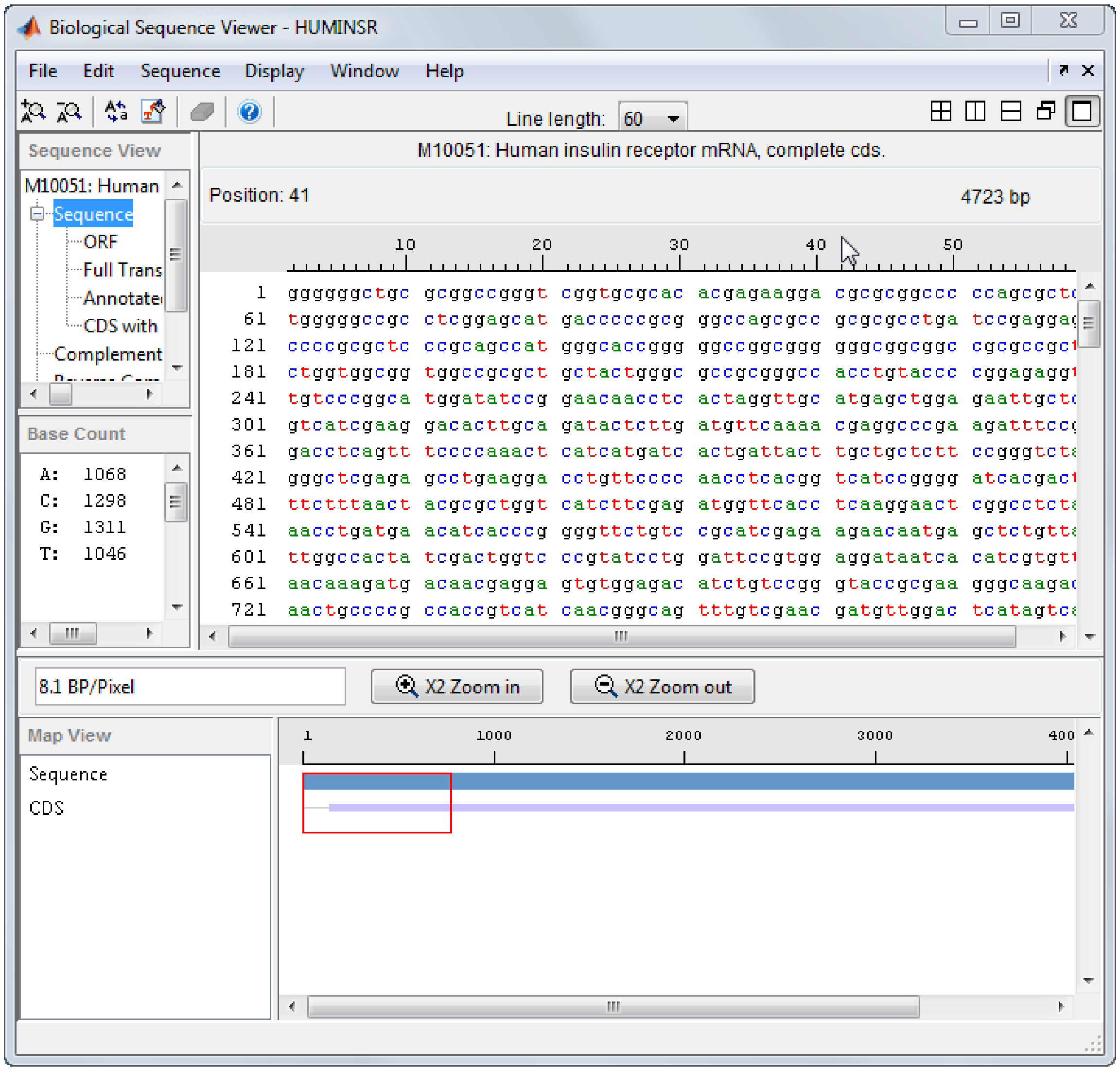
Retrieve a sequence from the GenBank database.

```
S = getgenbank('M10051');
```

Load the sequence into the Sequence Viewer app.

```
seqviewer(S)
```

Alternatively, you can click Sequence Viewer on the **Apps** tab to open the app, and view the biological sequence S.



Line length: 60

M10051: Human insulin receptor mRNA, complete cds.

Position: 41 4723 bp

```
A:   1068
C:   1298
G:   1311
T:   1046
```

```
             10         20         30         40         50
  1 gggggctgc gcggccgggt cggtgcgcac acgagaagga cgcgcggccc ccagcgct
 61 tgggggccgc ctcggagcat gaccccccgcg ggccagcgcc gcgcgcctga tccgagga
121 ccccgcgctc ccgcagccat gggcaccggg ggccggcggg gggcggcggc cgcgccgc
181 ctggtggcgg tggccgcgct gctactgggc gccgcgggcc acctgtaccc cggagagg
241 tgtcccggca tggatatccg gaacaacctc actaggttgc atgagctgga gaattgct
301 gtcatcgaag gacacttgca gatactcttg atgttcaaaa cgaggcccga agatttcc
361 gacctcagtt tccccaaact catcatgatc actgattact tgctgctctt ccgggtct
421 gggctcgaga gcctgaagga cctgttcccc aacctcacgg tcatccgggg atcacgac
481 ttctttaact acgcgctggt catcttcgag atggttcacc tcaaggaact cggcctct
541 aacctgatga acatcacccg gggttctgtc cgcatcgaga agaacaatga gctctgtt
601 ttggccacta tcgactggtc ccgtatcctg gattccgtgg aggataatca catcgtgt
661 aacaaagatg acaacgagga gtgtggagac atctgtccgg gtaccgcgaa gggcaaga
721 aactgccccg ccaccgtcat caacgggcag tttgtcgaac gatgttggac tcatagtc
```

8.1 BP/Pixel

Close the app.

```
seqviewer('close')
```

- “Exploring a Nucleotide Sequence Using the Sequence Viewer App”

Programmatic Use

`seqviewer` opens the Sequence Viewer app.

`seqviewer(Seq)` loads a amino acid or nucleotide sequence `Seq` into the app. `Seq` can be one of the following:

- A string of single-letter codes
- A row vector of integers
- A MATLAB structure containing a `Sequence` field that contains an amino acid or nucleotide sequence, such as the structure returned by `fastaread`, `fastqread`, `getgenpept`, `genpeptread`, `getpdb`, `pdbread`, `emblread`, `getembl`, `genbankread`, or `getgenbank`
- A string specifying a file name with an extension of .gbk, .gpt, .fasta, .fa, or .ebi.

Version History

Introduced before R2006a

See Also

Apps
Sequence Alignment

Functions
`seqviewer`

Topics
“Exploring a Nucleotide Sequence Using the Sequence Viewer App”