MATLAB®

Data Import and Export

MATLAB®

R2020a

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

MATLAB® Data Import and Export

Revision History

September 2009	Online only	New for MATLAB 7.9 (Release 2009b)
March 2010	Online only	Revised for Version 7.10 (Release 2010a)
September 2010	Online only	Revised for Version 7.11 (Release 2010b)
April 2011	Online only	Revised for Version 7.12 (Release 2011a)
September 2011	Online only	Revised for Version 7.13 (Release 2011b)
March 2012	Online only	Revised for Version 7.14 (Release 2012a)
September 2012	Online only	Revised for Version 8.0 (Release 2012b)
March 2013	Online only	Revised for Version 8.1 (Release 2013a)
September 2013	Online only	Revised for Version 8.2 (Release 2013b)
March 2014	Online only	Revised for Version 8.3 (Release 2014a)
October 2014	Online only	Revised for Version 8.4 (Release 2014b)
March 2015	Online only	Revised for Version 8.5 (Release 2015a)
September 2015	Online only	Revised for Version 8.6 (Release 2015b)
October 2015	Online only	Rereleased for Version 8.5.1 (Release 2015aSP1)
March 2016	Online only	Revised for Version 9.0 (Release 2016a)
September 2016	Online only	Revised for Version 9.1 (Release 2016b)
March 2017	Online only	Revised for Version 9.2 (Release 2017a)
September 2017	Online only	Revised for Version 9.3 (Release 2017b)
March 2018	Online only	Revised for Version 9.4 (Release 2018a)
September 2018	Online only	Revised for Version 9.5 (Release 2018b)
March 2019	Online only	Revised for Version 9.6 (Release 2019a)
September 2019	Online only	Revised for Version 9.7 (Release 2019b)
March 2020	Online only	Revised for Version 9.8 (Release 2020a)

Contents

1 File Opening, Loading, and Saving

2 Text Files

3 Spreadsheets

4 Low-Level File I/O

5 Internet of Things (IoT) Data

6 Images

7 Scientific Data

8 Audio and Video

9 XML Documents

10 Memory-Mapping Data Files

11 Internet File Access and JSON

12 Large Data

13 TCP/IP Support in MATLAB

14 Bluetooth Low Energy Communication

File Opening, Loading, and Saving

Supported File Formats for Import and Export

The following table shows the file formats that you can import and export from the MATLAB application.

In addition to the functions in the table, you also can use the **Import Tool** to import text or spreadsheet file formats interactively.

File Content	Extension	Description	Import Function	Export Function
MATLAB formatted data	MAT	Saved MATLAB workspace	`load`	`save`
		Partial access of variables in MATLAB workspace	`matfile`	`matfile`
Text	any, including: CSV TXT	Comma delimited numbers	`readmatrix`	`writematrix`
		Delimited numbers	`readmatrix`	`writematrix`
		Delimited numbers, or a mix of text and numbers	`textscan`	none

		Column-oriented delimited numbers or a mix of text and numbers	`readtable`
			
			`readcell`
			
			`readvars`
				`writetable`
				
				`writecell`

Spreadsheet	XLS
	XLSX
	XLSM
	
	XLSB (Systems with Microsoft® Excel® for Windows® only)
	
	XLTM (import only)
	XLTX (import only)
	
	ODS (Systems with Microsoft Excel for Windows only)
		Column-oriented data in worksheet or range of spreadsheet	`readmatrix`
			
			`readtable`
			
			`readcell`
			
			`readvars`
				`writematrix`
				
				`writetable`
				
				`writecell`

Extensible Markup Language	XML	XML-formatted text	`xmlread`	`xmlwrite`
Data Acquisition Toolbox™ file	DAQ	Data Acquisition Toolbox	`daqread`	none
Scientific data	CDF	Common Data Format	See “Common Data Format”	See `cdflib`
	FITS	Flexible Image Transport System	See “FITS Files”	See “FITS Files”

File Content	Extension	Description	Import Function	Export Function
	HDF	Hierarchical Data Format, version 4, or HDF-EOS v. 2	See "HDF4 Files"	See "HDF4 Files"
	H5	HDF or HDF-EOS, version 5	See "HDF5 Files"	See "HDF5 Files"
	NC	Network Common Data Form (netCDF)	See "NetCDF Files"	See "NetCDF Files"
Image	BMP	Windows Bitmap	`imread`	`imwrite`
	GIF	Graphics Interchange Format		
	HDF	Hierarchical Data Format		
	JPEG JPG	Joint Photographic Experts Group		
	JP2 JPF JPX J2C J2K	JPEG 2000		
	PBM	Portable Bitmap		
	PCX	Paintbrush		
	PGM	Portable Graymap		
	PNG	Portable Network Graphics		
	PNM	Portable Any Map		
	PPM	Portable Pixmap		
	RAS	Sun™ Raster		
	TIFF TIF	Tagged Image File Format		
	XWD	X Window Dump		
	CUR	Windows Cursor resources	`imread`	none
	ICO	Windows Icon resources		
Audio (all platforms)	AU SND	NeXT/Sun sound	`audioread`	`audiowrite`
	AIFF	Audio Interchange File Format		
	AIFC	Audio Interchange File Format, with compression codecs		
	FLAC	Free Lossless Audio Codec		
	OGG	Ogg Vorbis		
	WAV	Microsoft WAVE sound		
Audio (Windows)	M4A MP4	MPEG-4	`audioread`	`audiowrite`

File Content	Extension	Description	Import Function	Export Function
	any	Formats supported by Microsoft Media Foundation	`audioread`	none
Audio (Mac)	M4A MP4	MPEG-4	`audioread`	`audiowrite`
Audio (Linux®)	any	Formats supported by GStreamer	`audioread`	none
Video (all platforms)	AVI	Audio Video Interleave	`VideoReader`	`VideoWriter`
	MJ2	Motion JPEG 2000		
Video (Windows)	MPG	MPEG-1	`VideoReader`	none
	ASF ASX WMV	Windows Media®		
	any	Formats supported by Microsoft DirectShow®		
Video (Windows 7 or later)	MP4 M4V	MPEG-4	`VideoReader`	`VideoWriter`
	MOV	QuickTime	`VideoReader`	none
	any	Formats supported by Microsoft Media Foundation		
Video (Mac)	MP4 M4V	MPEG-4	`VideoReader`	`VideoWriter`
	MPG	MPEG-1	`VideoReader`	none
	MOV	QuickTime		
	any	Formats supported by QuickTime, including `.3gp`, `.3g2`, and `.dv`		
Video (Linux)	any	Formats supported by your installed GStreamer plug-ins, including `.ogg`	`VideoReader`	none
Triangulation	STL	Stereolithography	`stlread`	`stlwrite`

You can use web services such as a RESTful or WSDL to read and write data in an internet media type format such as JSON, XML, image, or text. For more information, see:

- "Web Access"
- "WSDL (Web Services Description Language)"

Methods for Importing Data

In this section...

Caution When you import data into the MATLAB workspace, the new variables you create overwrite any existing variables in the workspace that have the same name.

Tools that Import Multiple File Formats

You can import data into MATLAB from a disk file or the system clipboard interactively.

To import data from a file, do one of the following:

- On the **Home** tab, in the **Variable** section, select **Import Data** .
- Double-click a file name in the Current Folder browser.
- Call `uiimport`.

To import data from the clipboard, do one of the following:

- On the Workspace browser title bar, click , and then select **Paste**.
- Call `uiimport`.

To import without invoking a graphical user interface, the easiest option is to use the `importdata` function.

For a complete list of the formats you can import interactively or with `importdata`, see "Supported File Formats for Import and Export" on page 1-2.

Importing Specific File Formats

MATLAB includes functions tailored to import specific file formats. Consider using format-specific functions instead of importing data interactively when you want to import only a portion of a file. Many of the format-specific functions provide options for selecting ranges or portions of data. Some format-specific functions allow you to request multiple optional outputs. This option is not available when you import interactively.

For a complete list of the format-specific functions, see "Supported File Formats for Import and Export" on page 1-2.

For binary data files, consider "Overview of Memory-Mapping" on page 10-2. Memory-mapping enables you to access file data using standard MATLAB indexing operations.

Alternatively, MATLAB toolboxes perform specialized import operations. For example, use Database Toolbox™ software for importing data from relational databases. Refer to the documentation on specific toolboxes to see the available import features.

Importing Data with Low-Level I/O

If the Import Wizard, `importdata`, and format-specific functions cannot read your data, use *low-level I/O functions* such as `fscanf` or `fread`. Low-level functions allow the most control over reading from a file, but require detailed knowledge of the structure of your data. For more information, see:

- "Import Text Data Files with Low-Level I/O" on page 4-2
- "Import Binary Data with Low-Level I/O" on page 4-8

Import Images, Audio, and Video Interactively

Import data interactively into MATLAB workspace.

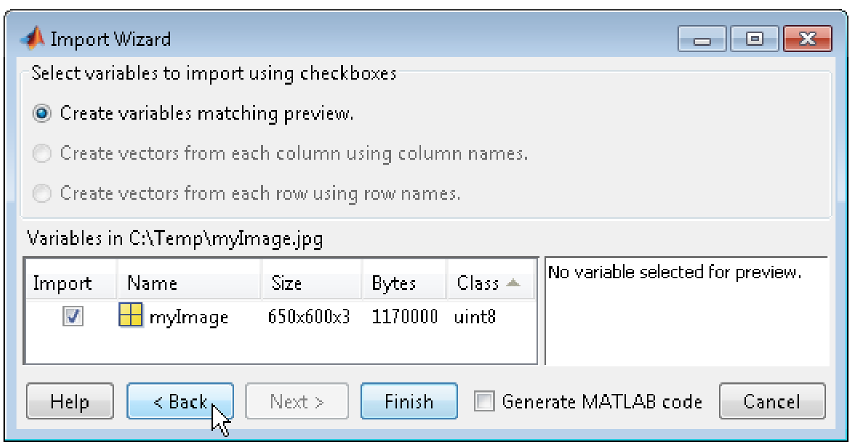
In this section...

Note For information on importing text files, see "Read Text File Data Using Import Tool" on page 2-4. For information on importing spreadsheets, see "Read Spreadsheet Data Using Import Tool" on page 3-4.

Viewing the Contents of a File

Start the Import Wizard by selecting **Import Data** or calling `uiimport`.

To view images or video, or to listen to audio, click the **Back** button on the first window that the Import Wizard displays.



The right pane of the new window includes a preview tab. Click the button in the preview tab to show an image or to play audio or video.

Specifying Variables

The Import Wizard generates default variable names based on the format and content of your data. You can change the variables in any of the following ways:

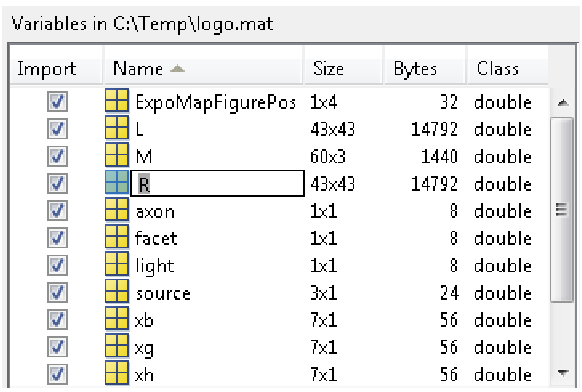
- "Renaming or Deselecting Variables" on page 1-8
- "Importing to a Structure Array" on page 1-8

The default variable name for data imported from the system clipboard is `A_pastespecial`.

If the Import Wizard detects a single variable in a file, the default variable name is the file name. Otherwise, the Import Wizard uses default variable names that correspond to the output fields of the `importdata` function. For more information on the output fields, see the `importdata` function reference page.

Renaming or Deselecting Variables

To override the default variable name, select the name and type a new one.


Variables in C:\Temp\logo.mat

Import	Name	Size	Bytes	Class
☑	ExpoMapFigurePos	1x4	32	double
☑	L	43x43	14792	double
☑	M	60x3	1440	double
☑	R	43x43	14792	double
☑	axon	1x1	8	double
☑	facet	1x1	8	double
☑	light	1x1	8	double
☑	source	3x1	24	double
☑	xb	7x1	56	double
☑	xg	7x1	56	double
☑	xh	7x1	56	double

To avoid importing a particular variable, clear the check box in the **Import** column.

Importing to a Structure Array

To import data into fields of a structure array rather than as individual variables, start the Import Wizard by calling `uiimport` with an output argument. For example, the sample file `durer.mat` contains three variables: `X`, `caption`, and `map`. If you issue the command

```
durerStruct = uiimport('durer.mat')
```

and click the **Finish** button, the Import Wizard returns a scalar structure with three fields:

```
durerStruct = 
          X: [648x509 double]
        map: [128x3 double]
    caption: [2x28 char]
```

To access a particular field, use dot notation. For example, view the `caption` field:

```
disp(durerStruct.caption)
```

MATLAB returns:

```
Albrecht Durer's Melancolia.
Can you find the matrix?
```

For more information, see "Access Data in Structure Array".

Generating Reusable MATLAB Code

To create a function that reads similar files without restarting the Import Wizard, select the **Generate MATLAB code** check box. When you click **Finish** to complete the initial import operation, MATLAB opens an Editor window that contains an unsaved function. The default function name is `importfile.m` or `importfileN.m`, where *N* is an integer.

The function in the generated code includes the following features:

- For text files, if you request vectors from rows or columns, the generated code also returns vectors.
- When importing from files, the function includes an input argument for the name of the file to import, `fileToRead1`.
- When importing into a structure array, the function includes an output argument for the name of the structure, `newData1`.

However, the generated code has the following limitations:

- If you rename or deselect any variables in the Import Wizard, the generated code does not reflect those changes.
- If you do not import into a structure array, the generated function creates variables in the base workspace. If you plan to call the generated function from within your own function, your function cannot access these variables. To allow your function to access the data, start the Import Wizard by calling `uiimport` with an output argument. Call the generated function with an output argument to create a structure array in the workspace of your function.

MATLAB does not automatically save the function. To save the file, select **Save**. For best results, use the function name with a `.m` extension for the file name.

See Also

`VideoReader` | `audioread` | `imread`

More About

- “Read Video Files” on page 8-8
- “Read and Write Audio Files” on page 8-2
- “Importing Images” on page 6-2

Import or Export a Sequence of Files

To import or export multiple files, create a control loop to process one file at a time. When constructing the loop:

- To build sequential file names, use `sprintf`.
- To find files that match a pattern, use `dir`.
- Use *function syntax* to pass the name of the file to the import or export function. (For more information, see "Command vs. Function Syntax".)

For example, to read files named `file1.txt` through `file20.txt` with `importdata`:

```
numfiles = 20;
mydata = cell(1, numfiles);

for k = 1:numfiles
  myfilename = sprintf('file%d.txt', k);
  mydata{k} = importdata(myfilename);
end
```

To read all files that match `*.jpg` with `imread`:

```
jpegFiles = dir('*.jpg');
numfiles = length(jpegFiles);
mydata = cell(1, numfiles);

for k = 1:numfiles
  mydata{k} = imread(jpegFiles(k).name);
end
```

Save and Load Parts of Variables in MAT-Files

In this section...

You can save and load parts of variables directly in MAT-files without loading them into memory using the `matfile` function. The primary advantage of using the `matfile` function over the `load` or `save` functions is that you can process parts of very large data sets that are otherwise too large to fit in memory. When working with these large variables, read and write as much data into memory as possible at a time. Otherwise, repeated file access can negatively impact the performance of your code.

Save and Load Using the matfile Function

This example shows how to load, modify, and save part of a variable in an existing MAT-file using the `matfile` function.

Create a Version 7.3 MAT-file with two variables, `A` and `B`.

```
A = rand(5);
B = magic(10);
save example.mat A B -v7.3;
clear A B
```

Construct a `MatFile` object from the MAT-file, `example.mat`. The `matfile` function creates a `MatFile` object that corresponds to the MAT-file and contains the properties of the `MatFile` object. By default, `matfile` only permits loading from existing MAT-files.

```
exampleObject = matfile('example.mat');
```

To enable saving, call `matfile` with the `Writable` parameter.

```
exampleObject = matfile('example.mat','Writable',true);
```

Alternatively, construct the object and set `Properties.Writable` in separate steps.

```
exampleObject = matfile('example.mat');
exampleObject.Properties.Writable = true;
```

Load the first row of `B` from `example.mat` into variable `firstRowB` and modify the data. When you index into objects associated with Version 7.3 MAT-files, MATLAB® loads only the part of the variable that you specify.

```
firstRowB = exampleObject.B(1,:);
firstRowB = 2 * firstRowB;
```

Update the values in the first row of variable `B` in `example.mat` using the values stored in `firstRowB`.

```
exampleObject.B(1,:) = firstRowB;
```

For very large files, the best practice is to read and write as much data into memory as possible at a time. Otherwise, repeated file access negatively impacts the performance of your code. For example, suppose that your file contains many rows and columns, and that loading a single row requires most of the available memory. Rather than updating one element at a time, update each row.

```
[nrowsB,ncolsB] = size(exampleObject,'B');
for row = 1:nrowsB
  exampleObject.B(row,:) = row * exampleObject.B(row,:);
end
```

If memory is not a concern, you can update the entire contents of a variable at a time.

```
exampleObject.B = 10 * exampleObject.B;
```

Alternatively, update a variable by calling the `save` function with the `-append` option. The `-append` option requests that the `save` function replace only the specified variable, `B`, and leave other variables in the file intact. This method always requires that you load and save the entire variable.

```
load('example.mat','B');
B(1,:) = 2 * B(1,:);
save('example.mat','-append','B');
```

Add a variable to the file using the `matlab.io.MatFile` object.

```
exampleObject.C = magic(8);
```

You also can add the variable by calling the `save` function with the `-append` option.

```
C = magic(8);
save('example.mat','-append','C');
clear C
```

Load Parts of Variables Dynamically

This example shows how to access parts of variables from a MAT-file dynamically. This is useful when working with MAT-files whose variables names are not always known.

Consider the example MAT-file, `topography.mat`, that contains one or more arrays with unknown names. Construct a `MatFile` object that corresponds to the file, `topography.mat`. Call `who` to get the variable names in the file.

```
exampleObject = matfile('topography.mat');
varlist = who(exampleObject)
```

```
varlist = 4x1 cell
    {'topo'      }
    {'topolegend'}
    {'topomap1'  }
    {'topomap2'  }
```

`varlist` is a cell array containing the names of the four variables in `topography.mat`.

The third and fourth variables, `topomap1` and `topomap2`, are both arrays containing topography data. Load the elevation data from the third column of each variable into a field of the structure array, `S`. For each field, specify a field name that is the original variable name prefixed by `elevationOf_`.

Then, access the data in each variable as properties of `exampleObject`. Because `varName` is a variable, enclose it in parentheses.

```
for index = 3:4
    varName = varlist{index};
    S(1).(['elevationOf_',varName]) = exampleObject.(varName)(:,3);
end
```

View the contents of the structure array, `S`.

```
S
```

```
S = struct with fields:
    elevationOf_topomap1: [64x1 double]
    elevationOf_topomap2: [128x1 double]
```

`S` has two fields, `elevationOf_topomap1` and `elevationOf_topomap2`, each containing a column vector.

Avoid Inadvertently Loading Entire Variables

When you do not know the size of a large variable in a MAT-file and want to load only parts of that variable at a time, avoid using the `end` keyword. Using the `end` keyword temporarily loads the entire contents of the variable in question into memory. For very large variables, loading takes a long time or generates `Out of Memory` errors. Instead, call the `size` method for `MatFile` objects.

For example, this code temporarily loads the entire contents of `B` in memory:

```
lastColB = exampleObject.B(:,end);
```

Use this code instead to improve performance:

```
[nrows,ncols] = size(exampleObject,'B');
lastColB = exampleObject.B(:,ncols);
```

Similarly, any time you refer to a variable with syntax of the form `matObj.varName`, such as `exampleObject.B`, MATLAB temporarily loads the entire variable into memory. Therefore, make sure to call the `size` method for `MatFile` objects with syntax such as:

```
[nrows,ncols] = size(exampleObject,'B');
```

rather than passing the entire contents of `exampleObject.B` to the `size` function,

```
[nrows,ncols] = size(exampleObject.B);
```

The difference in syntax is subtle, but significant.

Partial Loading and Saving Requires Version 7.3 MAT-Files

Any load or save operation that uses a `MatFile` object associated with a Version 7 or earlier MAT-file temporarily loads the entire variable into memory.

Use the `matfile` function to create files in Version 7.3 format. For example, this code

```
newfile = matfile('newfile.mat');
```

creates a MAT-file that supports partial loading and saving.

However, by default, the `save` function creates Version 7 MAT-files. Convert existing MAT-files to Version 7.3 by calling the `save` function with the `-v7.3` option, such as:

```
load('durer.mat');
save('mycopy_durer.mat','-v7.3');
```

To change your preferences to save new files in Version 7.3 format, access the **Environment** section on the **Home** tab, and click **Preferences**. Select **MATLAB** > **General** > **MAT-Files**. This preference is not available in MATLAB Online™.

See Also

`load` | `matfile` | `save`

More About

- "Save and Load Workspace Variables"
- "Growing Arrays Using matfile Function" on page 1-18
- "MAT-File Versions" on page 1-15

MAT-File Versions

In this section...

Overview of MAT-File Versions

MAT-files are binary MATLAB files that store workspace variables. Starting with MAT-file Version 4, there are several subsequent versions of MAT-files that support an increasing set of features. MATLAB releases R2006b and later all support all MAT-file versions.

By default, all save operations create Version 7 MAT-files. The only exception to this is when you create new MAT-files using the `matfile` function. In this case, the default MAT-file version is 7.3.

To identify or change the default MAT-file version, access the MAT-Files Preferences.

- On the **Home** tab, in the **Environment** section, click **Preferences**.
- Select **MATLAB** > **General** > **MAT-Files**.

The preferences apply to both the `save` function and the **Save** menu options.

The maximum size of a MAT-file is imposed only by your native file system.

This table lists and compares all MAT-file versions.

MAT-File Version	Supported MATLAB Releases	Supported Features	Compression	Maximum Size of Each Variable	Value of `version` argument in save function	Preference Option
Version 7.3	R2006b (Version 7.3) or later	Saving and loading parts of variables, and all Version 7 features	Yes	≥ 2 GB on 64-bit computers	`'-v7.3'`	**MATLAB Version 7.3 or later (save -v7.3)**
Version 7	R14 (Version 7.0) or later	Unicode® character encoding, which enables file sharing between systems that use different default character encoding schemes, and all Version 6 features.	Yes	2^31 bytes per variable	`'-v7'`	**MATLAB Version 7 or later (save -v7)**

MAT-File Version	Supported MATLAB Releases	Supported Features	Compression	Maximum Size of Each Variable	Value of version argument in save function	Preference Option
Version 6	R8 (Version 5) or later	N-dimensional arrays, cell arrays, structure arrays, variable names longer than 19 characters, and all Version 4 features.	No	2^31 bytes per variable	`'-v6'`	**MATLAB Version 5 or later (save -v6)**
Version 4	All	Two-dimensional `double`, character, and sparse arrays	No	100,000,000 elements per array, and 2^31 bytes per variable	`'-v4'`	n/a

Note Version 7.3 MAT-files use an HDF5 based format that requires some overhead storage to describe the contents of the file. For cell arrays, structure arrays, or other containers that can store heterogeneous data types, Version 7.3 MAT-files are sometimes larger than Version 7 MAT-files.

Save to Nondefault MAT-File Version

Save to a MAT-file version other than the default version when you want to:

- Allow access to the file using earlier versions of MATLAB.
- Take advantage of Version 7.3 MAT-file features.
- Reduce the time required to load and save some files by storing uncompressed data.
- Reduce the size of some files by storing compressed data.

To save to a MAT-file version other than the default version, specify a `version` as the last input to the `save` function. For example, to create a Version 6 MAT-file named `myfile.mat`, type:

```
save('myfile.mat','-v6')
```

Data Compression

Beginning with Version 7, MATLAB compresses data when writing to MAT-files to save storage space. Data compression and decompression slow down all save operations and some load operations. In most cases, the reduction in file size is worth the additional time spent.

In some cases, loading compressed data actually can be *faster* than loading uncompressed data. For example, consider a block of data in a numeric array saved to both a 10 MB compressed file and a 100 MB uncompressed file. Loading the first 10 MB takes the same amount of time for each file. Loading the remaining 90 MB from the uncompressed file takes nine times as long as loading the first 10 MB. Completing the load of the compressed file requires only the relatively short time to decompress the data.

The benefits of data compression are negligible in the following cases:

- The amount of data in each item is small relative to the complexity of its container. For example, simple numeric arrays take less time to compress and uncompress than cell or structure arrays of the same size. Compressing arrays that result in an uncompressed file size of less than 3 MB offers limited benefit, unless you are transferring data over a network.
- The data is random, with no repeated patterns or consistent values.

Accelerate Save and Load Operations for Version 7.3 MAT-Files

Version 7.3 MAT-files use an HDF5-based format that stores data in compressed chunks. The time required to load part of a variable from a Version 7.3 MAT-file depends on how that data is stored across one or more chunks. Each chunk that contains any portion of the data you want to load must be fully uncompressed to access the data. Rechunking your data can improve the performance of the load operation. To rechunk data, use the HDF5 command-line tools, which are part of the HDF5 distribution.

See Also

`matfile` | `save`

More About

- "Save and Load Workspace Variables"

Growing Arrays Using matfile Function

When writing a large number of large values to a MAT-file, the size of the file increases in a nonincremental way. This method of increase is expected. To minimize the number of times the file must grow and ensure optimal performance though, assign initial values to the array prior to populating it with data.

For example, suppose that you have a writable `MatFile` object.

```
fileName = 'matFileOfDoubles.mat';
matObj = matfile(fileName);
matObj.Properties.Writable = true;
```

Define parameters of the values to write. In this case, write one million values, fifty thousand at a time. The values should have a mean of 123.4, and a standard deviation of 56.7.

```
size = 1000000;
chunk = 50000;
mean = 123.4;
std = 56.7;
```

Assign an initial value of zero to the last element in the array prior to populating it with data.

```
matObj.data(1,size) = 0;
```

View the size of the file.

- On Windows systems, use `dir`.

  ```
  system('dir matFileOfDoubles.mat');
  ```
- On UNIX® systems, use `ls -ls`:

  ```
  system('ls -ls matFileOfDoubles.mat');
  ```

In this case, `matFileOfDoubles.mat` is less than 5000 bytes. Assigning an initial value to the last element of the array does not create a large file. It does, however, prepare your system for the potentially large size increase of `matFileOfDoubles.mat`.

Write data to the array, one chunk at a time.

```
nout = 0;
while(nout < size)
    fprintf('Writing %d of %d\n',nout,size);
    chunkSize = min(chunk,size-nout);
    data = mean + std * randn(1,chunkSize);
    matObj.data(1,(nout+1):(nout+chunkSize)) = data;
    nout = nout + chunkSize;
end
```

View the size of the file.

```
system('dir matFileOfDoubles.mat');
```

The file size is now larger because the array is populated with data.

See Also

`matfile`

More About

- “Save and Load Parts of Variables in MAT-Files” on page 1-11

Unexpected Results When Loading Variables Within a Function

If you have a function that loads data from a MAT-file and find that MATLAB does not return the expected results, check whether any variables in the MAT-file share the same name as a MATLAB function. Common variable names that conflict with function names include `i`, `j`, `mode`, `char`, `size`, and `path`.

These unexpected results occur because when you execute a function, MATLAB preprocesses all the code in the function before running it. However, calls to `load` are not preprocessed, meaning MATLAB has no knowledge of the variables in your MAT-file. Variables that share the same name as MATLAB functions are, therefore, preprocessed as MATLAB functions, causing the unexpected results. This is different from scripts, which MATLAB preprocesses and executes line by line, similar to the Command Window.

For example, consider a MAT-file with variables `height`, `width`, and `length`. If you load these variables in a function such as `findVolume`, MATLAB interprets the reference to `length` as a call to the MATLAB `length` function, and returns an error.

```
function vol = findVolume(myfile)
  load(myfile);
  vol = height * width * length;
end
```

```
Error using length
Not enough input arguments.
```

To avoid confusion, when defining your function, choose one (or more) of these approaches:

- Load the variables into a structure array. For example:

  ```
  function vol = findVolume(myfile)
    dims = load(myfile);
    vol = dims.height * dims.width * dims.length;
  end
  ```
- Explicitly include the names of variables in the call to the `load` function. For example:

  ```
  function vol = findVolume(myfile)
    load(myfile,'height','width','length')
    vol = height * width * length;
  end
  ```
- Initialize the variables within the function before calling `load`. To initialize a variable, assign it to an empty matrix or an empty character vector. For example:

  ```
  function vol = findVolume(myfile)
    height = [];
    width = [];
    length = [];
    load(myfile);
    vol = height * width * length;
  ```

To determine whether a particular variable name is associated with a MATLAB function, use the `exist` function. A return value of 5 determines that the name is a built-in MATLAB function.

See Also
`load`

More About

- "Save and Load Workspace Variables"

Create Temporary Files

Use the `tempdir` function to return the name of the folder designated to hold temporary files on your system. For example, issuing `tempdir` on The Open Group UNIX systems returns the `/tmp` folder.

Use the `tempname` function to return a file name in the temporary folder. The returned file name is a suitable destination for temporary data. For example, if you need to store some data in a temporary file, then you might issue the following command first:

```
fileID = fopen(tempname,'w');
```

In most cases, `tempname` generates a universally unique identifier (UUID). However, if you run MATLAB without JVM™, then `tempname` generates a random name using the CPU counter and time, and this name is not guaranteed to be unique.

Some systems delete temporary files every time you reboot the system. On other systems, designating a file as temporary means only that the file is not backed up.

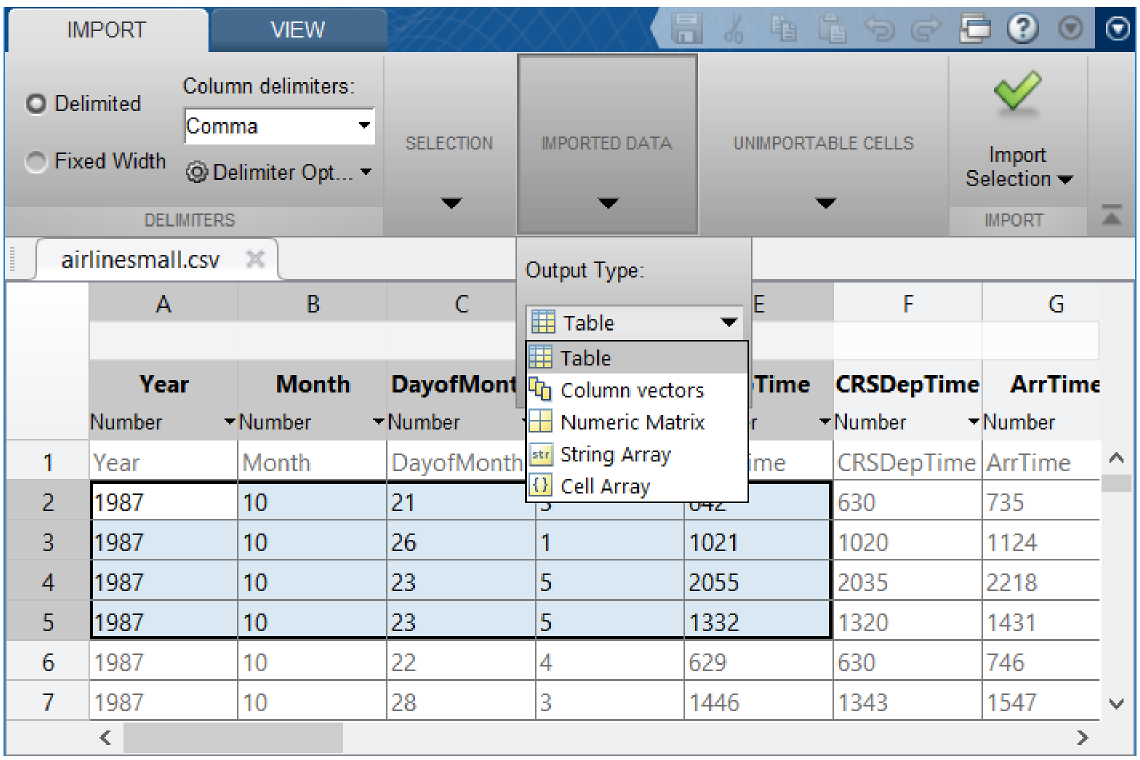
Text Files

Import Text Files

Text files often contain a mix of numeric and text data as well as variable and row names, which is best represented in MATLAB as a table. You can import tabular data from text files into a table using the **Import Tool** or the `readtable` function.

Import Text Files Using the Import Tool

The **Import Tool** allows you to import into a table or other data type. For example, read a subset of data from the sample file `airlinesmall.csv`. Open the file using the **Import Tool** and select options such as the range of data to import and the output type. Then, click on the **Import Selection** button  to import the data into the MATLAB workspace.



Import Text Files Using readtable

Alternatively, you can read tabular data from a text file into a table using the `readtable` function with the file name, for example:

```
T = readtable('airlinesmall.csv');
```

Display the first five rows and columns from the table.

```
T(1:5,1:5)
```

```
ans =
```

```
  5×5 table

    Year    Month    DayofMonth    DayOfWeek    DepTime 
    ____    _____    __________    _________    ________

    1987     10          21            3        {'642' }
    1987     10          26            1        {'1021'}
    1987     10          23            5        {'2055'}
    1987     10          23            5        {'1332'}
    1987     10          22            4        {'629' }
```

Import Data from Text Files as Other Data Types

In addition to tables, you can import tabular data from a text file into the MATLAB workspace as a timetable, a numeric matrix, a cell array, or separate column vectors. Based on the data type you need, use one of these functions.

Data Type of Output	Function
Timetable	`readtimetable`
Numeric Matrix	`readmatrix`
Cell Array	`readcell`
Separate Column Vectors	`readvars`

See Also

Import Tool | `readtable`

More About

- "Read Text File Data Using Import Tool" on page 2-4
- "Import Mixed Data from Text File into Table" on page 2-14
- "Access Data in Tables"

Read Text File Data Using Import Tool

In this section...

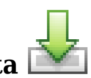
Import data from a text file by selecting data interactively. You also can repeat this import operation on multiple text files by using the generate code feature of the import tool.

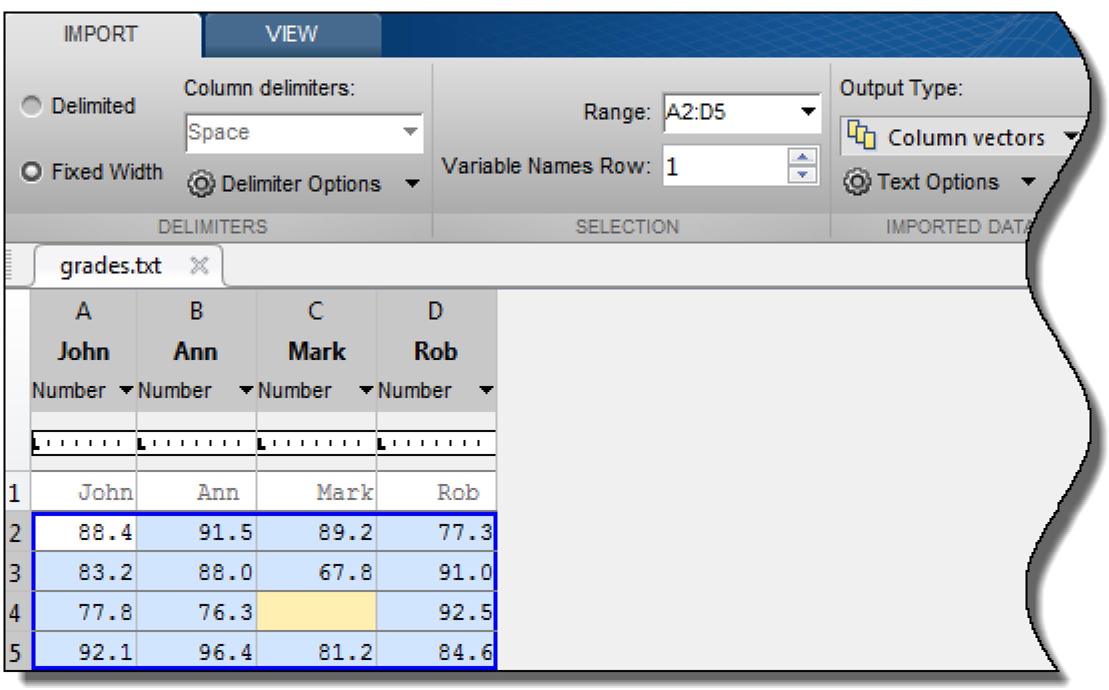
Select Data Interactively

This example shows how to import data from a text file with column headers and numeric data using the Import Tool. The file in the example, `grades.txt`, contains this data:

```
John    Ann     Mark    Rob
88.4    91.5    89.2    77.3
83.2    88.0    67.8    91.0
77.8    76.3            92.5
92.1    96.4    81.2    84.6
```

To create the file, copy and paste the data using any text editor.

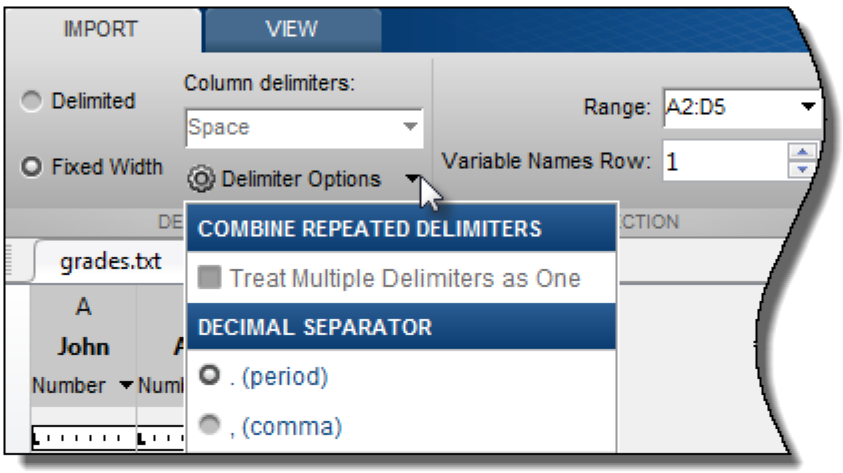
On the **Home** tab, in the **Variable** section, click **Import Data** . Alternatively, right-click the name of the file in the Current Folder browser and select **Import Data**. The Import Tool opens.



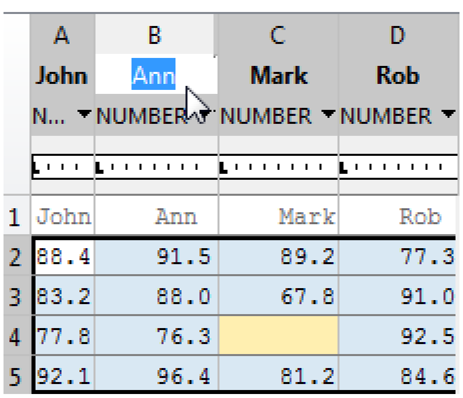
The Import Tool recognizes that `grades.txt` is a fixed width file. In the **Imported Data** section, select how you want the data to be imported. The following table indicates how data is imported depending on the option you select.

Option Selected	How Data is Imported
Table	Import selected data as a table.
Column vectors	Import each column of the selected data as an individual m-by-1 vector.
Numeric Matrix	Import selected data as an m-by-n numeric array.
String Array	Import selected data as a string array that contains text.
Cell Array	Import selected data as a cell array that can contain multiple data types, such as numeric data and text.

Under **Delimiter Options**, you can specify whether the Import Tool should use a period or a comma as the decimal separator for numeric values.

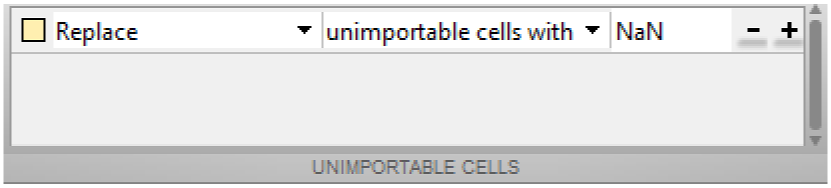


Double-click a variable name to rename it.



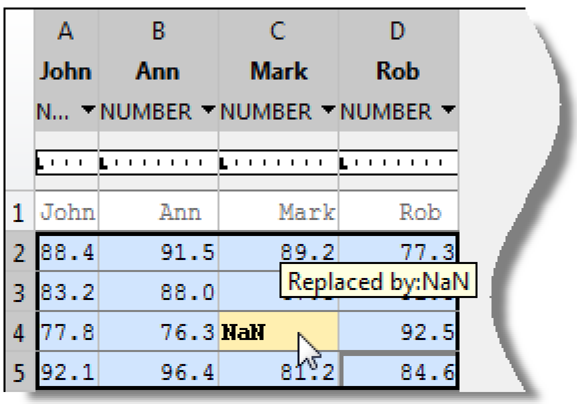
You also can use the **Variable Names Row** box in the **Selection** section to select the row in the text file that you want the Import Tool to use for variable names.

The Import Tool highlights unimportable cells. Unimportable cells are cells that contain data that cannot be imported in the format specified for that column. In this example, the cell at row 3, column C, is considered unimportable because a blank cell is not numeric. Highlight colors correspond to proposed rules to make the data fit into a numeric array. You can add, remove, reorder, or edit rules, such as changing the replacement value from NaN to another value.



All rules apply to the imported data only and do not change the data in the file. Any time you are importing into a matrix or into numeric column vectors and the range includes non-numeric data, then you must specify the rules.

To see how your data is imported, place the cursor over individual cells.



When you click the **Import Selection** button, the Import Tool creates variables in your workspace.

For more information on interacting with the Import Tool, watch this video.

Import Data from Multiple Text Files

To perform the same import operation on multiple files, use the code generation feature of the Import Tool. If you import a file one time and generate code from the Import Tool, you can use this code to make it easier to repeat the operation. The Import Tool generates a program script that you can edit and run to import the files, or a function that you can call for each file.

Suppose you have a set of text files in the current folder. The files are named `myfile01.txt` through `myfile25.txt`, and you want to import the data from each file, starting from the second row. Generate code to import the entire set of files as follows:

1. Open one of the files in the Import Tool.
2. Click **Import Selection**, and then select **Generate Function**. The Import Tool generates code similar to the following excerpt, and opens the code in the Editor.

```
function data = importfile(filename,startRow,endRow)
%IMPORTFILE Import numeric data from a text file as a matrix.
...
```

3 Save the function.

4 In a separate program file or at the command line, create a `for` loop to import data from each text file into a cell array named `myData`:

```
numFiles = 25;
startRow = 2;
endRow = inf;
myData = cell(1,numFiles);

for fileNum = 1:numFiles
    fileName = sprintf('myfile%02d.txt',fileNum);
    myData{fileNum} = importfile(fileName,startRow,endRow);
end
```

Each cell in `myData` contains an array of data from the corresponding text file. For example, `myData{1}` contains the data from the first file, `myfile01.txt`.

See Also

`readcell` | `readmatrix` | `readtable` | `readtimetable` | `readvars` | `textscan`

More About

- “Import Text Files” on page 2-2

Import Dates and Times from Text Files

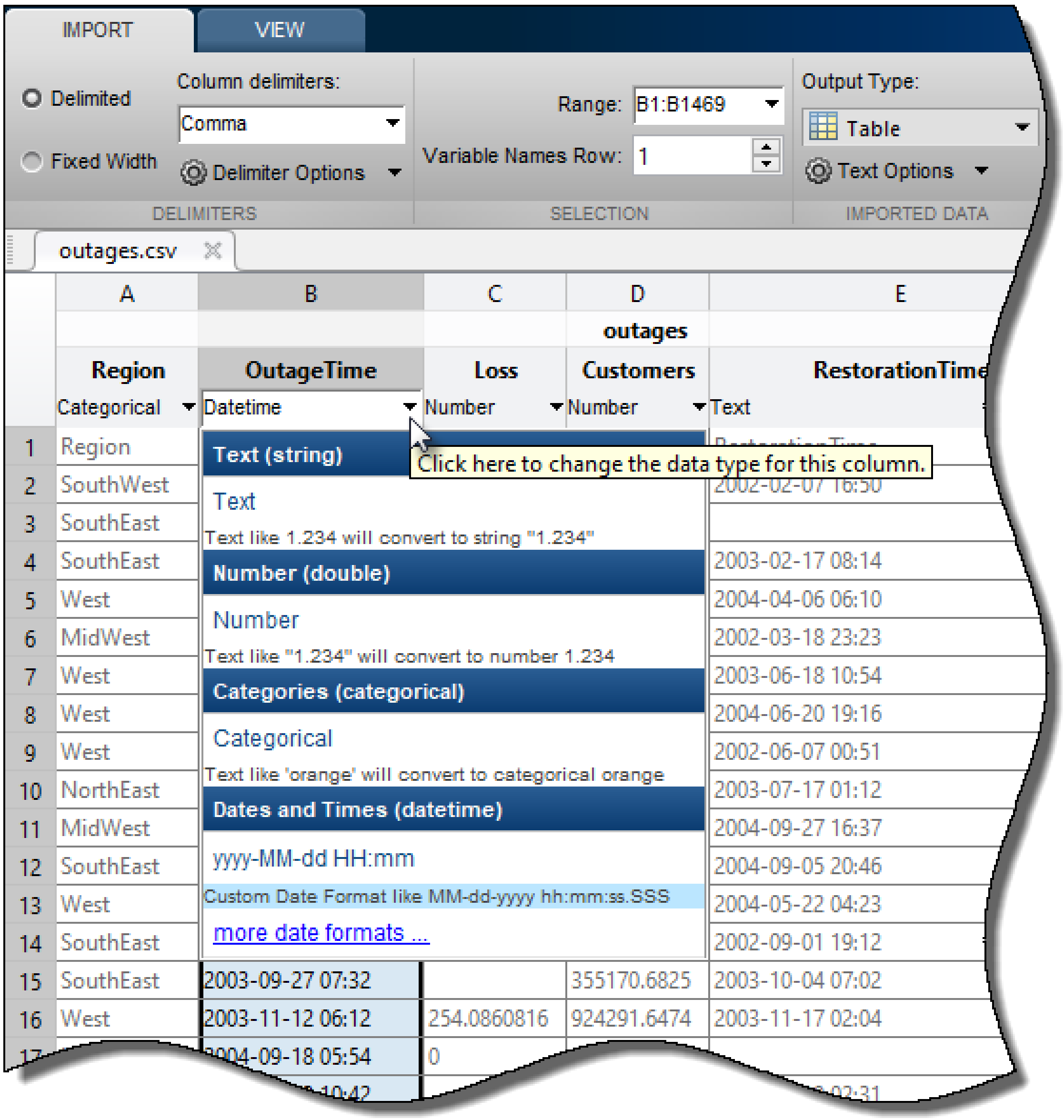
Import formatted dates and times (such as `'01/01/01'` or `'12:30:45'`) from column oriented tabular data in three ways.

- Import Tool — Interactively select and import dates and times.
- `readtable` function — Automatically detect variables with dates and times and import them into a table.
- Import Options — Use `readtable` with `detectImportOptions` function for more control over importing date and time variables. For example, you can specify properties such as `FillValue` and `DatetimeFormat`.

This example shows you how to import dates and times from text files using each of these methods.

Import Tool

Open the file `outages.csv` using the Import Tool. Specify the formats of dates and times using the drop-down menu for each column. You can select from a predefined date format, or enter a custom format. To import the `OutageTime` column, specify the custom format `yyyy-MM-dd HH:mm`. Then, click the **Import Selection** button to import the data into the workspace.



readtable Function

Use the `readtable` function and display `10` rows of the `OutageTime` variable. `readtable` automatically detects the date time variables and formats.

```
filename = 'outages.csv';
T = readtable(filename);
T.OutageTime(1:10)
```

```
ans = 10x1 datetime
   2002-02-01 12:18
   2003-01-23 00:49
   2003-02-07 21:15
   2004-04-06 05:44
   2002-03-16 06:18
   2003-06-18 02:49
```

```
   2004-06-20 14:39
   2002-06-06 19:28
   2003-07-16 16:23
   2004-09-27 11:09
```

Import Options

Use an import options object for more control over importing date and time variables. For example, change the date-time display format or specify a fill value for missing dates.

Create an import options object for the `outages.csv` file and display the variable import options for the variable `RestorationTime`. The `detectImportOptions` function automatically detects the data types of the variables.

```
opts = detectImportOptions(filename);
getvaropts(opts,'RestorationTime')
```

```
ans = 
  DatetimeVariableImportOptions with properties:

   Variable Properties:
              Name: 'RestorationTime'
              Type: 'datetime'
         FillValue: NaT
    TreatAsMissing: {}
         QuoteRule: 'remove'
          Prefixes: {}
          Suffixes: {}
    EmptyFieldRule: 'missing'

   Datetime Options:
    DatetimeFormat: 'default'
    DatetimeLocale: 'en_US'
       InputFormat: ''
          TimeZone: ''
```

Import the data and display the first `10` rows of the variable `RestorationTime`. The second row contains a `NaT`, indicating a missing date and time value.

```
T = readtable(filename,opts);
T.RestorationTime(1:10)
```

```
ans = 10x1 datetime
   2002-02-07 16:50
   NaT
   2003-02-17 08:14
   2004-04-06 06:10
   2002-03-18 23:23
   2003-06-18 10:54
   2004-06-20 19:16
   2002-06-07 00:51
   2003-07-17 01:12
   2004-09-27 16:37
```

To use a different date-time display format, update the `DatetimeFormat` property, and then replace missing values with the current date and time by using the `FillValue` property. Display the updated variable options.

```
opts = setvaropts(opts,'RestorationTime', ...
                       'DatetimeFormat','MMMM d, yyyy HH:mm:ss Z',...
                       'FillValue','now');
getvaropts(opts,'RestorationTime')
```

```
ans = 
  DatetimeVariableImportOptions with properties:

   Variable Properties:
              Name: 'RestorationTime'
              Type: 'datetime'
         FillValue: February 29, 2020 03:49:21 *
    TreatAsMissing: {}
         QuoteRule: 'remove'
          Prefixes: {}
          Suffixes: {}
    EmptyFieldRule: 'missing'

   Datetime Options:
    DatetimeFormat: 'MMMM d, yyyy HH:mm:ss Z'
    DatetimeLocale: 'en_US'
       InputFormat: ''
          TimeZone: ''
```

Read the data with the updated import options and display the first `10` rows of the variable.

```
T = readtable(filename,opts);
T.RestorationTime(1:10)
```

```
ans = 10x1 datetime
   2002-02-07 16:50
   2020-02-29 03:49
   2003-02-17 08:14
   2004-04-06 06:10
   2002-03-18 23:23
   2003-06-18 10:54
   2004-06-20 19:16
   2002-06-07 00:51
   2003-07-17 01:12
   2004-09-27 16:37
```

For more information on the `datetime` variable options, see the `setvaropts` reference page.

See Also

Import Tool | `detectImportOptions` | `readcell` | `readmatrix` | `readtable` | `readtimetable` | `readvars` | `setvaropts`

More About

- “Import Mixed Data from Text File into Table” on page 2-14

Import Numeric Data from Text Files into Matrix

Import numeric data as MATLAB arrays from files stored as comma-separated or delimited text files.

Import Comma-Separated Data

This example shows how to import comma-separated numeric data from a text file. Create a sample file, read all the data in the file, and then read only a subset starting from a specified location.

Create a sample file named `ph.dat` that contains comma-separated data and display the contents of the file.

```
A = 0.9*gallery('integerdata',99,[3 4],1);
writematrix(A,'ph.dat','Delimiter',',')
type('ph.dat')
```

```
85.5,54,74.7,34.2
63,75.6,46.8,80.1
85.5,39.6,2.7,38.7
```

Read the file using the `readmatrix` function. The function returns a 3-by-4 `double` array containing the data from the file.

```
M = readmatrix('ph.dat')
```

```
M = 3×4

   85.5000   54.0000   74.7000   34.2000
   63.0000   75.6000   46.8000   80.1000
   85.5000   39.6000    2.7000   38.7000
```

Import only the rectangular portion of data starting from the first row and third column in the file. Create an import options object and specify the columns and rows to import using the `SelectedVariableNames` and `DataLines` properties. Then, import the selected portion of the data from the file.

```
opts = detectImportOptions('ph.dat');
opts.SelectedVariableNames = {'Var3','Var4'};
opts.DataLines = [1 3];
readmatrix('ph.dat',opts)
```

```
ans = 3×2

   74.7000   34.2000
   46.8000   80.1000
    2.7000   38.7000
```

Import Delimited Numeric Data

This example shows how to import numeric data delimited by any single character using the `writematrix` function. Create a sample file, read the entire file, and then read a subset of the file starting at the specified location.

Create a tab-delimited file named `num.txt` that contains a 4-by-4 numeric array and display the contents of the file.

```
A = gallery('integerdata',99,[4,4],0);
writematrix(A,'num.txt','Delimiter','\t')
type('num.txt')
```

```
95	89	82	92
23	76	45	74
61	46	61	18
49	2	79	41
```

Read the entire file. The `readmatrix` function determines the delimiter automatically and returns a 4-by-4 `double` array.

```
M = readmatrix('num.txt')
```

```
M = 4×4

    95    89    82    92
    23    76    45    74
    61    46    61    18
    49     2    79    41
```

Read only the rectangular block of data beginning from the second row, third column, in the file. Create an import options object and specify the columns and rows to import using the `SelectedVariableNames` and `DataLines` properties. Then, import the selected portion of the data from the file.

```
opts = detectImportOptions('num.txt');
opts.SelectedVariableNames = {'Var3','Var4'};
opts.DataLines = [2 4];
readmatrix('num.txt',opts)
```

```
ans = 3×2

    45    74
    61    18
    79    41
```

See Also

`readcell` | `readmatrix` | `readtimetable` | `readvars`

More About

- "Import Text Files" on page 2-2

Import Mixed Data from Text File into Table

This example shows how to use the `readtable` function to import mixed text and numeric data into a table, specify the data types for the variables, and then append a new variable to the table.

Sample File Overview

The sample file, `outages.csv`, contains data representing electric utility outages in the US. The first few lines of the file are:

```
Region,OutageTime,Loss,Customers,RestorationTime,Cause

SouthWest,2002-01-20 11:49,672,2902379,2002-01-24 21:58,winter storm

SouthEast,2002-01-30 01:18,796,336436,2002-02-04 11:20,winter storm

SouthEast,2004-02-03 21:17,264.9,107083,2004-02-20 03:37,winter storm

West,2002-06-19 13:39,391.4,378990,2002-06-19 14:27,equipment fault
```

Read Text File

Import the data using `readtable` and display the first five rows. The `readtable` function automatically detects the delimiter and the variable types.

```
T = readtable('outages.csv');
head(T,5) % show first 5 rows of table
```

```
ans=5×6 table
       Region             OutageTime        Loss     Customers     RestorationTime            Cause
    _____________    ________________    ______    __________    ________________    _____________

    {'SouthWest'}    2002-02-01 12:18    458.98    1.8202e+06    2002-02-07 16:50    {'winter sto
    {'SouthEast'}    2003-01-23 00:49    530.14    2.1204e+05                 NaT    {'winter sto
    {'SouthEast'}    2003-02-07 21:15     289.4    1.4294e+05    2003-02-17 08:14    {'winter sto
    {'West'     }    2004-04-06 05:44    434.81    3.4037e+05    2004-04-06 06:10    {'equipment
    {'MidWest'  }    2002-03-16 06:18    186.44    2.1275e+05    2002-03-18 23:23    {'severe sto
```

Specify Variable Data Types Before Import

Updating the variable data types to the appropriate MATLAB data types can benefit your data, based on the type of variables in your file. For example, the first and sixth columns in `outages.csv` are categorical. By designating these two columns as `categorical` arrays you can leverage MATLAB functions for processing categorical data.

Designate and specify the data types of the variables in one of these ways:

- Specify the `Format` name-value pair in `readtable`
- Set the `VariableTypes` property of the import options for the file

Use the `Format` name-value pair to specify the variable data types, read the data, and display the first five rows. In the `%{yyyy-MM-dd HH:mm}D` part of the `formatSpec` specifier, the text between the curly braces describes the format of the date and time data. The values specified in `Format` designate the:

- First and last columns in the file as categorical data
- Second and fifth columns as formatted date and time data
- Third and fourth columns as floating-point values

```
formatSpec = '%C%{yyyy-MM-dd HH:mm}D%f%f%{yyyy-MM-dd HH:mm}D%C';
T = readtable('outages.csv','Format',formatSpec);
head(T,5)
```

```
ans=5×6 table
      Region           OutageTime          Loss      Customers       RestorationTime             Cause
    _________    ________________    ______    __________    ________________    _______________

    SouthWest    2002-02-01 12:18    458.98    1.8202e+06    2002-02-07 16:50    winter storm
    SouthEast    2003-01-23 00:49    530.14    2.1204e+05                 NaT    winter storm
    SouthEast    2003-02-07 21:15     289.4    1.4294e+05    2003-02-17 08:14    winter storm
    West         2004-04-06 05:44    434.81    3.4037e+05    2004-04-06 06:10    equipment fault
    MidWest      2002-03-16 06:18    186.44    2.1275e+05    2002-03-18 23:23    severe storm
```

Alternatively, specify the data types for the variables by using the `setvartype` function of the import options. First, create an import options object for the file. The data file contains different types of variables. Designate the first and last variables as `categorical` arrays, the second and fifth variables as `datetime` arrays, and the remaining variables as `double`.

```
opts = detectImportOptions('outages.csv');
varNames = opts.VariableNames ; % variable names
varTypes = {'categorical','datetime','double',...
                'double','datetime','categorical'};
opts = setvartype(opts,varNames,varTypes);
```

Import the data using `readtable` with `opts`, and then display the first five rows.

```
T = readtable('outages.csv',opts);
head(T,5)
```

```
ans=5×6 table
      Region           OutageTime          Loss      Customers       RestorationTime             Cause
    _________    ________________    ______    __________    ________________    _______________

    SouthWest    2002-02-01 12:18    458.98    1.8202e+06    2002-02-07 16:50    winter storm
    SouthEast    2003-01-23 00:49    530.14    2.1204e+05                 NaT    winter storm
    SouthEast    2003-02-07 21:15     289.4    1.4294e+05    2003-02-17 08:14    winter storm
    West         2004-04-06 05:44    434.81    3.4037e+05    2004-04-06 06:10    equipment fault
    MidWest      2002-03-16 06:18    186.44    2.1275e+05    2002-03-18 23:23    severe storm
```

Append New Variable to Table

Table `T` contains `OutageTime` and `RestorationTime`. Calculate the duration of each electrical outage and append this data to the table.

```
T.Duration = T.RestorationTime - T.OutageTime;
head(T,5)
```

```
ans=5×7 table
      Region           OutageTime          Loss      Customers       RestorationTime             Cause
    _________    ________________    ______    __________    ________________    _______________
```

```
    SouthWest    2002-02-01 12:18    458.98    1.8202e+06    2002-02-07 16:50    winter storm
    SouthEast    2003-01-23 00:49    530.14    2.1204e+05                 NaT    winter storm
    SouthEast    2003-02-07 21:15     289.4    1.4294e+05    2003-02-17 08:14    winter storm
    West         2004-04-06 05:44    434.81    3.4037e+05    2004-04-06 06:10    equipment fault
    MidWest      2002-03-16 06:18    186.44    2.1275e+05    2002-03-18 23:23    severe storm
```

See Also

`detectImportOptions` | `head` | `preview` | `readtable` | `readtimetable` | `setvaropts` | `setvartype`

More About

- “Create and Work with Tables”
- “Import Dates and Times from Text Files” on page 2-8
- “Access Data in Tables”

Import Block of Mixed Data from Text File into Table or Cell Array

This example reads a block of mixed text and numeric data from a text file, and then imports the block of data into a table or a cell array.

Data File Overview

The sample file `bigfile.txt` contains commented lines beginning with `##`. The data is arranged in five columns: The first column contains text indicating timestamps. The second, third, and fourth columns contain numeric data indicating temperature, humidity and wind speed. The last column contains descriptive text. Display the contents of the file `bigfile.txt`.

```
type('bigfile.txt')
```

```
## A    ID = 02476
## YKZ Timestamp Temp Humidity Wind Weather
06-Sep-2013 01:00:00    6.6    89    4    clear
06-Sep-2013 05:00:00    5.9    95    1    clear
06-Sep-2013 09:00:00    15.6    51    5    mainly clear
06-Sep-2013 13:00:00    19.6    37    10    mainly clear
06-Sep-2013 17:00:00    22.4    41    9    mostly cloudy
06-Sep-2013 21:00:00    17.3    67    7    mainly clear
## B    ID = 02477
## YVR Timestamp Temp Humidity Wind Weather
09-Sep-2013 01:00:00    15.2    91    8    clear
09-Sep-2013 05:00:00    19.1    94    7    n/a
09-Sep-2013 09:00:00    18.5    94    4    fog
09-Sep-2013 13:00:00    20.1    81    15    mainly clear
09-Sep-2013 17:00:00    20.1    77    17    n/a
09-Sep-2013 18:00:00    20.0    75    17    n/a
09-Sep-2013 21:00:00    16.8    90    25    mainly clear
## C    ID = 02478
## YYZ Timestamp Temp Humidity Wind Weather
```

Import Block of Data as Table

To import the data as a table, use `readtable` with import options.

Create an import options object for the file using the `detectImportOptions` function. Specify the location of the data using the `DataLines` property. For example, lines `3` through `8` contain the first block of data. Optionally, you can specify the names of the variables using the `VariableNames` property. Finally import the first block of data using `readtable` with the `opts` object.

```
opts = detectImportOptions('bigfile.txt');
opts.DataLines = [3 8];
opts.VariableNames = {'Timestamp','Temp',...
                      'Humidity','Wind','Weather'};
T_first = readtable('bigfile.txt',opts)
```

```
T_first=6×5 table
         Timestamp          Temp    Humidity    Wind         Weather
    ____________________    ____    ________    ____    _________________

    06-Sep-2013 01:00:00     6.6       89         4     {'clear'        }
    06-Sep-2013 05:00:00     5.9       95         1     {'clear'        }
```

```
    06-Sep-2013 09:00:00    15.6      51         5     {'mainly clear' }
    06-Sep-2013 13:00:00    19.6      37        10     {'mainly clear' }
    06-Sep-2013 17:00:00    22.4      41         9     {'mostly cloudy'}
    06-Sep-2013 21:00:00    17.3      67         7     {'mainly clear' }
```

Read the second block by updating the `DataLines` property to the location of the second block.

```
opts.DataLines = [11 17];
T_second = readtable('bigfile.txt',opts)
```

```
T_second=7×5 table
         Timestamp          Temp    Humidity    Wind        Weather
    ____________________    ____    ________    ____    ________________

    09-Sep-2013 01:00:00    15.2       91         8     {'clear'       }
    09-Sep-2013 05:00:00    19.1       94         7     {'n/a'         }
    09-Sep-2013 09:00:00    18.5       94         4     {'fog'         }
    09-Sep-2013 13:00:00    20.1       81        15     {'mainly clear'}
    09-Sep-2013 17:00:00    20.1       77        17     {'n/a'         }
    09-Sep-2013 18:00:00      20       75        17     {'n/a'         }
    09-Sep-2013 21:00:00    16.8       90        25     {'mainly clear'}
```

Import Block of Data as Cell Array

You can import the data as cell array using the `readcell` function with `detectImportOptions`, or by using the `textscan` function. First import the block of data using the `readcell` function and then perform the same import by using `textscan`.

To perform the import using the readcell function, create an import options object for the file using the `detectImportOptions` function. Specify the location of the data using the `DataLines` property. Then, perform the import operation using the `readcell` function and import options object `opts`.

```
opts = detectImportOptions('bigfile.txt');
opts.DataLines = [3 8]; % fist block of data
C = readcell('bigfile.txt',opts)
```

```
C=6×5 cell array
  Columns 1 through 4

    {[06-Sep-2013 01:00:00]}    {[ 6.6000]}    {[89]}    {[ 4]}
    {[06-Sep-2013 05:00:00]}    {[ 5.9000]}    {[95]}    {[ 1]}
    {[06-Sep-2013 09:00:00]}    {[15.6000]}    {[51]}    {[ 5]}
    {[06-Sep-2013 13:00:00]}    {[19.6000]}    {[37]}    {[10]}
    {[06-Sep-2013 17:00:00]}    {[22.4000]}    {[41]}    {[ 9]}
    {[06-Sep-2013 21:00:00]}    {[17.3000]}    {[67]}    {[ 7]}

  Column 5

    {'clear'        }
    {'clear'        }
    {'mainly clear' }
    {'mainly clear' }
    {'mostly cloudy'}
    {'mainly clear' }
```

To perform the import using the `textscan` function, specify the size of block using `N` and the format of the data fields using `formatSpec`. For example, use `'%s'` for text variables, `'%D'` for date and time variables, or `'%c'` for categorical variables. Use `fopen` to open the file. The function then returns a file identifier, `fileID`. Next, read from the file by using the `textscan` function.

```
N = 6;
formatSpec = '%D %f %f %f %c';
fileID = fopen('bigfile.txt');
```

Read the first block and display the contents of the variable `Humidity`.

```
C_first = textscan(fileID,formatSpec,N,'CommentStyle','##','Delimiter','\t')
```

```
C_first=1×5 cell array
  Columns 1 through 4

    {6x1 datetime}    {6x1 double}    {6x1 double}    {6x1 double}

  Column 5

    {6x1 char}
```

```
C_first{3}
```

```
ans = 6×1

    89
   NaN
    95
   NaN
    51
   NaN
```

Update the block size N, and read the second block. Display the contents of the fifth variable `Weather`.

```
N = 7;
C_second = textscan(fileID,formatSpec,N,'CommentStyle','##','Delimiter','\t')
```

```
C_second=1×5 cell array
  Columns 1 through 4

    {7x1 datetime}    {7x1 double}    {7x1 double}    {7x1 double}

  Column 5

    {7x1 char}
```

```
C_second{5}
```

```
ans = 7x1 char array
    'm'
    '...'
    'm'
    '...'
    'm'
```

```
'...'
'C'
```

Close the file.

```
fclose(fileID);
```

See Also

`detectImportOptions` | `fopen` | `readcell` | `readtable` | `textscan`

More About

- “Access Data in Cell Array”
- “Moving within a File” on page 4-10

Write Data to Text Files

In this section...

Export tabular data contained in tables, cell arrays, or numeric arrays from the MATLAB workspace to text files.

Export Table to Text File

You can export tabular data from MATLAB® workspace into a text file using the `writetable` function. Create a sample table, write the table to text file, and then write the table to text file with additional options.

Create a sample table, `T`, containing the variables `Pitch`, `Shape`, `Price` and `Stock`.

```
Pitch = [0.7;0.8;1;1.25;1.5];
Shape = {'Pan';'Round';'Button';'Pan';'Round'};
Price = [10.0;13.59;10.50;12.00;16.69];
Stock = [376;502;465;1091;562];
T = table(Pitch,Shape,Price,Stock)
```

```
T=5×4 table
    Pitch      Shape       Price     Stock
    _____    __________    _____     _____

     0.7     {'Pan'   }       10      376
     0.8     {'Round' }    13.59      502
       1     {'Button'}     10.5      465
    1.25     {'Pan'   }       12     1091
     1.5     {'Round' }    16.69      562
```

Export the table, `T`, to a text file named `tabledata.txt`. View the contents of the file. By default, `writetable` writes comma-separated data, includes table variable names as column headings.

```
writetable(T,'tabledata.txt');
type tabledata.txt
```

```
Pitch,Shape,Price,Stock
0.7,Pan,10,376
0.8,Round,13.59,502
1,Button,10.5,465
1.25,Pan,12,1091
1.5,Round,16.69,562
```

Create a table `T2` which includes row names using the `RowNames` name-value pair argument.

```
rowNames = {'M4';'M5';'M6';'M8';'M10'};
T2 = table(Pitch,Shape,Price,Stock,'RowNames',rowNames)
```

```
T2=5×4 table
           Pitch      Shape       Price     Stock
```

```
         _____    __________    _____    _____

    M4     0.7    {'Pan'   }       10     376
    M5     0.8    {'Round' }    13.59     502
    M6       1    {'Button'}     10.5     465
    M8    1.25    {'Pan'   }       12    1091
    M10    1.5    {'Round' }    16.69     562
```

Export T2 to a tab-delimited text file named `tabledata2.txt`. Use the `Delimiter` name-value pair argument to specify a tab delimiter, and the `WriteRowNames` name-value pair argument to include row names. View the contents of the file.

```
writetable(T2,'tabledata2.txt','Delimiter','\t','WriteRowNames',true);
type tabledata2.txt
```

```
Row	Pitch	Shape	Price	Stock
M4	0.7	Pan	10	376
M5	0.8	Round	13.59	502
M6	1	Button	10.5	465
M8	1.25	Pan	12	1091
M10	1.5	Round	16.69	562
```

Export Cell Array to Text File

You can export a cell array from MATLAB® workspace into a text file in one of these ways:

- Use the writecell function to export the cell array to a text file.
- Use `fprintf` to export the cell array by specifying the format of the output data.

Create a sample cell array `C`.

```
C = {'Atkins',32,77.3,'M';'Cheng',30,99.8,'F';'Lam',31,80.2,'M'}
```

```
C = 3×4 cell array
    {'Atkins'}    {[32]}    {[77.3000]}    {'M'}
    {'Cheng' }    {[30]}    {[99.8000]}    {'F'}
    {'Lam'   }    {[31]}    {[80.2000]}    {'M'}
```

Export the cell array using `writecell`.

```
writecell(C,'data.dat')
```

View the contents of the file.

```
type data.dat
```

```
Atkins,32,77.3,M
Cheng,30,99.8,F
Lam,31,80.2,M
```

Alternatively, import the cell array using `fprintf`. Open a file that you can write to named `celldata.dat`. Define `formatSpec` using the format specifiers to describe the pattern of the data in the file. Typical format specifiers include `'%s'` for a character vector, `'%d'` for an integer, or `'%f'` for a floating-point number. Separate each format specifier with a space to indicate a space delimiter for the output file. Include a newline character at the end of each row of data (`'\n'`).

```
fileID = fopen('celldata.dat','w');
formatSpec = '%s %d %2.1f %s\n';
```

Determine the size of `C` and export one row of data at a time using the `fprintf` function. Then close the file. `fprintf` writes a space-delimited file.

```
[nrows,ncols] = size(C);
for row = 1:nrows
    fprintf(fileID,formatSpec,C{row,:});
end
fclose(fileID);
```

View the contents of the file.

```
type celldata.dat
```

```
Atkins 32 77.3 M
Cheng 30 99.8 F
Lam 31 80.2 M
```

Export Numeric Array to Text File

You can export a numerical array to a text file using `writematrix`.

Create a numeric array A.

```
A = magic(5)/10
```

```
A = 5×5

    1.7000    2.4000    0.1000    0.8000    1.5000
    2.3000    0.5000    0.7000    1.4000    1.6000
    0.4000    0.6000    1.3000    2.0000    2.2000
    1.0000    1.2000    1.9000    2.1000    0.3000
    1.1000    1.8000    2.5000    0.2000    0.9000
```

Write the numeric array to `myData.dat` and specify the delimiter to be `';'`. Then, view the contents of the file.

```
writematrix(A,'myData.dat','Delimiter',';')
type myData.dat
```

```
1.7;2.4;0.1;0.8;1.5
2.3;0.5;0.7;1.4;1.6
0.4;0.6;1.3;2;2.2
1;1.2;1.9;2.1;0.3
1.1;1.8;2.5;0.2;0.9
```

See Also

`fprintf` | `type` | `writecell` | `writematrix` | `writetable` | `writetimetable`

Write to a Diary File

To keep an activity log of your MATLAB session, use the `diary` function. `diary` creates a verbatim copy of your MATLAB session in a disk file (excluding graphics).

For example, if you have the array `A` in your workspace,

```
A = [ 1 2 3 4; 5 6 7 8 ];
```

execute these commands at the MATLAB prompt to export this array using `diary`:

1. Turn on the `diary` function. Optionally, you can name the output file `diary` creates:

   ```
   diary my_data.out
   ```
2. Display the contents of the array you want to export. This example displays the array `A`. You could also display a cell array or other MATLAB class:

   ```
   A =
       1       2       3       4
       5       6       7       8
   ```
3. Turn off the `diary` function:

   ```
   diary off
   ```

   `diary` creates the file `my_data.out` and records all the commands executed in the MATLAB session until you turn it off:

   ```
   A =

        1       2       3       4
        5       6       7       8

   diary off
   ```
4. Open the `diary` file `my_data.out` in a text editor and remove the extraneous text, if desired.

Read Collection or Sequence of Text Files

When your data is stored across multiple text files, you can use `tabularTextDatastore` to manage and import the data. This example shows how to use `tabularTextDatastore` to read the data from the collection of text files all together, or to read one file at a time.

Data

For this example, the folder `C:\DataTxt` contains a collection of text files. Capture this location in the variable `location`. The data contains `10` text files, where each file contains `10` rows of data. The results differ based on your files and data.

```
location = 'C:\DataTxt';
dir(location)
```

```
.          File01.csv  File03.csv  File05.csv  File07.csv  File09.csv
..         File02.csv  File04.csv  File06.csv  File08.csv  File10.csv
```

Create Datastore

Create a datastore using the location of the files.

```
ds = tabularTextDatastore(location)
```

```
ds =
  TabularTextDatastore with properties:

                      Files: {
                             'C:\DataTxt\File01.csv';
                             'C:\DataTxt\File02.csv';
                             'C:\DataTxt\File03.csv'
                              ... and 7 more
                             }
               FileEncoding: 'UTF-8'
   AlternateFileSystemRoots: {}
          ReadVariableNames: true
              VariableNames: {'LastName', 'Gender', 'Age' ... and 7 more}
             DatetimeLocale: en_US

  Text Format Properties:
             NumHeaderLines: 0
                  Delimiter: ','
               RowDelimiter: '\r\n'
             TreatAsMissing: ''
               MissingValue: NaN

  Advanced Text Format Properties:
            TextscanFormats: {'%q', '%q', '%f' ... and 7 more}
                   TextType: 'char'
         ExponentCharacters: 'eEdD'
               CommentStyle: ''
                 Whitespace: ' \b\t'
    MultipleDelimitersAsOne: false

  Properties that control the table returned by preview, read, readall:
      SelectedVariableNames: {'LastName', 'Gender', 'Age' ... and 7 more}
            SelectedFormats: {'%q', '%q', '%f' ... and 7 more}
```

```
                   ReadSize: 20000 rows
```

Read Data from Datastore

Use the `read` or `readall` functions to import the data from the datastore. If the data from the collection fits in the memory, you can import it all at once using the `readall` function.

```
allData = readall(ds);
size(allData)
```

```
ans = 1×2

   100    10
```

Alternatively, import the data one file at a time using the `read` function. To control the amount of data imported, before you call read, adjust the `ReadSize` property of the datastore. Set the `ReadSize` to `'file'` or a positive integer.

- If `ReadSize` is `'file'`, then each call to `read` reads all the data one file at a time.
- If `ReadSize` is a positive integer, then each call to `read` reads the number of rows specified by `ReadSize`, or fewer, if it reaches the end of the data.

```
ds.ReadSize = 'file';
firstFile = read(ds) % reads first file
```

```
firstFile=10×10 table
     LastName       Gender      Age              Location              Height    Weight    Smoker
    __________    ________    ___    _____________________________    ______    ______    _______

    'Smith'       'Male'      38     'County General Hospital'          71        176      'TRUE'
    'Johnson'     'Male'      43     'VA Hospital'                      69        163      'FALSE'
    'Williams'    'Female'    38     'St. Mary's Medical Center'        64        131      'FALSE'
    'Jones'       'Female'    40     'VA Hospital'                      67        133      'FALSE'
    'Brown'       'Female'    49     'County General Hospital'          64        119      'FALSE'
    'Davis'       'Female'    46     'St. Mary's Medical Center'        68        142      'FALSE'
    'Miller'      'Female'    33     'VA Hospital'                      64        142      'TRUE'
    'Wilson'      'Male'      40     'VA Hospital'                      68        180      'FALSE'
    'Moore'       'Male'      28     'St. Mary's Medical Center'        68        183      'FALSE'
    'Taylor'      'Female'    31     'County General Hospital'          66        132      'FALSE'
```

```
secondFile = read(ds) % reads second file
```

```
secondFile=10×10 table
     LastName       Gender      Age              Location              Height    Weight    Smoker
    __________    ________    ___    _____________________________    ______    ______    _______

    'Anderson'    'Female'    45     'County General Hospital'          68        128      'FALSE'
    'Thomas'      'Female'    42     'St. Mary's Medical Center'        66        137      'FALSE'
    'Jackson'     'Male'      25     'VA Hospital'                      71        174      'FALSE'
    'White'       'Male'      39     'VA Hospital'                      72        202      'TRUE'
    'Harris'      'Female'    36     'St. Mary's Medical Center'        65        129      'FALSE'
    'Martin'      'Male'      48     'VA Hospital'                      71        181      'TRUE'
    'Thompson'    'Male'      32     'St. Mary's Medical Center'        69        191      'TRUE'
    'Garcia'      'Female'    27     'VA Hospital'                      69        131      'TRUE'
    'Martinez'    'Male'      37     'County General Hospital'          70        179      'FALSE'
```

```
    'Robinson'    'Male'      50     'County General Hospital'        68       172      'FALSE'
```

See Also
readcell | readmatrix | readtable | readtimetable | readvars | tabularTextDatastore

More About
- "Read and Analyze Large Tabular Text File" on page 12-96

Import Block of Numeric Data from Text File

This example shows how to read numeric data organized in blocks in a text file. Each block within the file can have a different format. You can read all the blocks as cell arrays, one block at a time, using `textscan`.

File Format Overview

The information in the sample text file, `test80211.txt`, is the result from a wireless network communication quality test. The sample file consists of four lines of introduction followed by several blocks of data. Each block represents a different environment (for example, mobile, indoor, outdoor) and has the following format:

- Two header lines of description
- The text, `Num SNR=`, followed by a numeric value, `m`
- Numeric data organized in a table of `m` columns and an arbitrary number of rows (The data is comma-delimited.)
- The text, `*EOB`, denoting the end of the block

For example, a block of data is formatted like this:

* Indoor2

* SNR Vs test No

Num SNR=3

,-5.00E+00,-4.00E+00,

1.00E+00,3.32E-07,9.12E-07

2.00E+00,1.49E-07,2.44E-07

3.00E+00,6.04E-07,2.53E-07

4.00E+00,1.53E-07,4.25E-07

5.00E+00,1.82E-07,1.83E-07

6.00E+00,6.27E-07,8.21E-07

7.00E+00,9.10E-08,1.53E-08

8.00E+00,8.73E-07,6.45E-07

9.00E+00,4.40E-07,1.33E-07

*EOB

The numeric data represents error rates over a range of noise levels for a number of independent tests. The first column indicates the test number. To view the entire sample file, type at the command line:

```
open test80211.txt
```

Open Text File for Reading

Open the file and create a file identifier.

```
fileID = fopen('test80211.txt','r');
```

Read Introduction Lines

Read the four introductory lines, which contain text delimited by a newline character. `textscan` returns a 1-by-1 cell array containing a 4-by-1 cell array of character vectors.

```
Intro = textscan(fileID,'%s',4,'Delimiter','\n')
```

```
Intro = 1x1 cell array
    {4x1 cell}
```

View the contents of the first cell.

```
disp(Intro{1})
```

```
    {'*CCX'                      }
    {'*CCX WiFi conformance test'}
    {'*CCX BER Results'          }
    {'*CCX'                      }
```

Read Each Block

For each block, we want to read a header, the numeric value `m`, column headers for the data, then the data itself. First, initialize the block index.

```
Block = 1;
```

Read each block of data in a `while` loop. The loop executes until the end of the file is reached and `~feof` returns `false`. The `textscan` function returns the data in each block as a cell array named `InputText`. Convert each cell array to a numeric array using `cell2mat` and store the numeric array in a cell array named `Data`. A cell array allows the storage of different size blocks.

```
while (~feof(fileID))                               % For each block:

   fprintf('Block: %s\n', num2str(Block))           % Print block number to the screen
   InputText = textscan(fileID,'%s',2,'delimiter','\n');  % Read 2 header lines
   HeaderLines{Block,1} = InputText{1};
   disp(HeaderLines{Block});                        % Display header lines

   InputText = textscan(fileID,'Num SNR = %f');     % Read the numeric value
                                                    % following the text, Num SNR =
   NumCols = InputText{1};                          % Specify that this is the
                                                    % number of data columns

   FormatString = repmat('%f',1,NumCols);           % Create format string
                                                    % based on the number
                                                    % of columns
   InputText = textscan(fileID,FormatString, ...    % Read data block
      'delimiter',',');

   Data{Block,1} = cell2mat(InputText);
   [NumRows,NumCols] = size(Data{Block});           % Determine size of table
```

```
   disp(cellstr(['Table data size: ' ...
      num2str(NumRows) ' x ' num2str(NumCols)]));
   disp(' ');                                          % New line

   eob = textscan(fileID,'%s',1,'delimiter','\n');  % Read and discard end-of-block marker
   Block = Block+1;                                    % Increment block index
end
```

```
Block: 1
    {'*       Mobile1'       }
    {'*       SNR Vs test No'}

    {'Table data size: 30 x 19'}


Block: 2
    {'*       Mobile2'       }
    {'*       SNR Vs test No'}

    {'Table data size: 30 x 9'}


Block: 3
    {'*       Mobile3'       }
    {'*       SNR Vs test No'}

    {'Table data size: 31 x 15'}


Block: 4
    {'*       Mobile4'       }
    {'*       SNR Vs test No'}

    {'Table data size: 28 x 19'}


Block: 5
    {'*       Mobile5'       }
    {'*       SNR Vs test No'}

    {'Table data size: 32 x 18'}


Block: 6
    {'*       Mobile6'       }
    {'*       SNR Vs test No'}

    {'Table data size: 30 x 19'}


Block: 7
    {'*       Mobile7'       }
    {'*       SNR Vs test No'}

    {'Table data size: 30 x 11'}


Block: 8
```

```
    {'*       Mobile8'       }
    {'*       SNR Vs test No'}

    {'Table data size: 20 x 18'}


Block: 9
    {'*       Indoor0'       }
    {'*       SNR Vs test No'}

    {'Table data size: 9 x 3'}


Block: 10
    {'*       Indoor1'       }
    {'*       SNR Vs test No'}

    {'Table data size: 22 x 6'}


Block: 11
    {'*       Indoor2'       }
    {'*       SNR Vs test No'}

    {'Table data size: 25 x 3'}


Block: 12
    {'*       Indoor3'       }
    {'*       SNR Vs test No'}

    {'Table data size: 21 x 18'}


Block: 13
    {'*       Outdoor1'      }
    {'*       SNR Vs test No'}

    {'Table data size: 20 x 18'}


Block: 14
    {'*       Outdoor2'      }
    {'*       SNR Vs test No'}

    {'Table data size: 23 x 3'}


Block: 15
    {'*       Outdoor3'      }
    {'*       SNR Vs test No'}

    {'Table data size: 22 x 18'}


Block: 16
    {'*       Outdoor4'      }
    {'*       SNR Vs test No'}
```

```
    {'Table data size: 21 x 18'}


Block: 17
    {'*       Outdoor5'      }
    {'*       SNR Vs test No'}

    {'Table data size: 18 x 5'}
```

Close Text File

```
fclose(fileID);
```

Total Number of Blocks

Determine the number of blocks in the file.

```
NumBlocks = Block-1
```

```
NumBlocks = 17
```

View Numeric Data

Display the numeric data in one of the blocks using short scientific notation.

First, store the current Command Window output display format.

```
user_format = get(0, 'format');
```

Change the display format to short scientific notation.

```
format shortE
```

Display the header lines for the ninth block and the numeric data.

```
Block = 9;
disp(HeaderLines{Block});
```

```
    {'*       Indoor0'       }
    {'*       SNR Vs test No'}
```

```
fprintf('SNR        %d        %d\n',Data{Block,1}(1,2:end))
```

```
SNR        -7        -6
```

```
disp(Data{Block,1}(2:end,2:end));
```

```
   9.0600e-07   6.7100e-07
   3.1700e-07   3.5400e-07
   2.8600e-07   1.9600e-07
   1.4800e-07   7.3400e-07
   3.9500e-08   9.6600e-07
   7.9600e-07   7.8300e-07
   4.0000e-07   8.8100e-07
   3.0100e-07   2.9700e-07
```

Restore the original Command Window output display format.

```
set(0, 'format', user_format);
```

See Also

`textscan`

More About

- "Import Block of Mixed Data from Text File into Table or Cell Array" on page 2-17

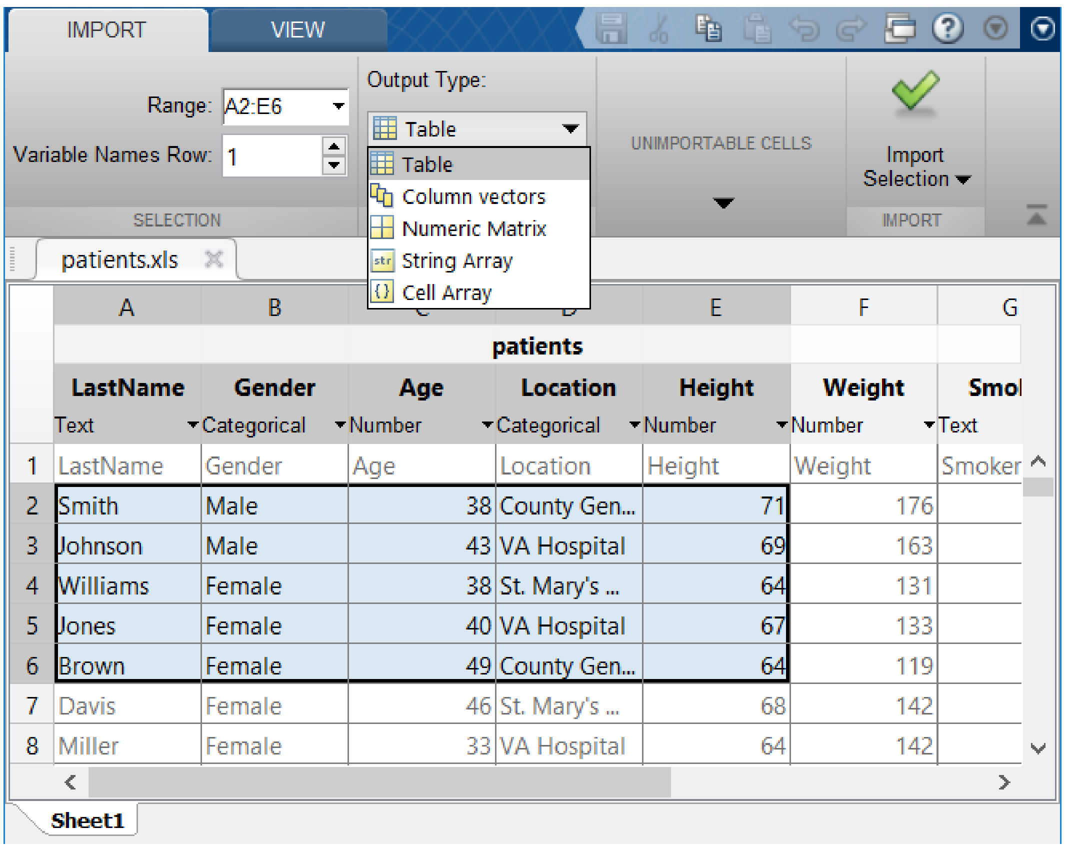
Spreadsheets

Import Spreadsheets

Spreadsheets often contain a mix of numeric and text data as well as variable and row names, which is best represented in MATLAB as a table. You can import data into a table using the **Import Tool** or the `readtable` function.

Import Spreadsheet Data Using the Import Tool

The **Import Tool** allows you to import into a table or other data type. For example, read data from the sample spreadsheet file `patients.xls` as a table in MATLAB. Open the file using the **Import Tool** and select options such as the range of data and the output type. Then, click the **Import Selection** button  to import the data into the MATLAB workspace.



Import Spreadsheet Data Using readtable

Alternatively, you can read spreadsheet data into a table using the `readtable` function with the file name, for example:

```
T = readtable('patients.xls');
```

You can also select the range of data to import by specifying the range parameter. For example, read the first five rows and columns of the spreadsheet. Specify the range in Excel notation as `'A1:E5'`.

```
T = readtable('patients.xls','Range','A1:E5')
```

```
T =

  4×5 table

      LastName        Gender      Age             Location              Height
    ____________    __________    ___    _____________________________    ______

    {'Smith'   }    {'Male'  }    38     {'County General Hospital'  }      71
    {'Johnson' }    {'Male'  }    43     {'VA Hospital'              }      69
    {'Williams'}    {'Female'}    38     {'St. Mary's Medical Center'}      64
    {'Jones'   }    {'Female'}    40     {'VA Hospital'              }      67
```

Import Spreadsheet Data as Other Data Types

In addition to tables, you can import your spreadsheet data into the MATLAB workspace as a timetable, a numeric matrix, a cell array, or separate column vectors. Based on the data type you need, use one of these functions.

Data Type of Output	Function
Timetable	`readtimetable`
Numeric Matrix	`readmatrix`
Cell Array	`readcell`
Separate Column Vectors	`readvars`

See Also
Import Tool | `readtable`

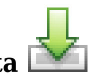
More About
- “Read Spreadsheet Data Using Import Tool” on page 3-4
- “Read Spreadsheet Data into Table” on page 3-9
- “Access Data in Tables”

Read Spreadsheet Data Using Import Tool

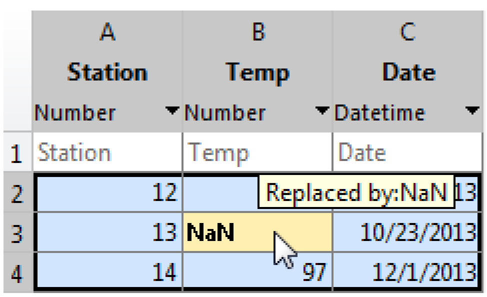
In this section...

This example shows how to import data from a spreadsheet into the workspace using the Import Tool and also to import data from the clipboard.

Select Data Interactively

On the **Home** tab, in the **Variable** section, click **Import Data** . Alternatively, in the Current Folder browser, double-click the name of a file with an extension of `.xls`, `.xlsx`, `.xlsb`, or `.xlsm`. The Import Tool opens.

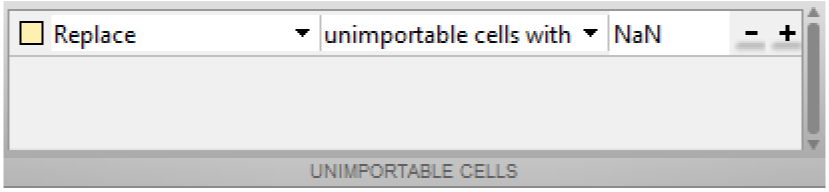
Select the data you want to import. For example, the data in the following figure corresponds to data for three column vectors. You can edit the variable name within the tab, and you can select noncontiguous sections of data for the same variable.



On the **Import** tab, in the **Output Type** section, select how you want the data to be imported. The option you select dictates the data type of the imported data.

Option Selected	How Data Is Imported
Column vectors	Import each column of the selected data as an individual `m`-by-1 vector.
Numeric Matrix	Import selected data as an `m`-by-`n` numeric array.
String Array	Import selected data as an `m`-by-`n` string array.
Cell Array	Import selected data as a cell array that can contain multiple data types, such as numeric data and text.
Table	Import selected data as a table.

If you choose to import the data as a matrix or as numeric column vectors, the tool highlights any nonnumeric data in the worksheet. Each highlight color corresponds to a proposed rule to make the data fit into a numeric array. For example, you can replace nonnumeric values with `NaN`. Also, you can see how your data will be imported when you place the cursor over individual cells.



You can add, remove, reorder, or edit rules, such as changing the replacement value from `NaN` to another value. All rules apply to the imported data only and do not change the data in the file. Specify rules any time the range includes nonnumeric data and you are importing into a matrix or numeric column vectors.

Any cells that contain `#Error?` correspond to formula errors in your spreadsheet file, such as division by zero. The Import Tool regards these cells as nonnumeric.

When you click the **Import Selection** button , the Import Tool creates variables in your workspace.

For more information on interacting with the Import Tool, watch this video.

Import Data from Multiple Spreadsheets

If you plan to perform the same import operation on multiple files, you can generate code from the Import Tool to make it easier to repeat the operation. On all platforms, the Import Tool can generate a program script that you can edit and run to import the files. On Microsoft Windows systems with Excel software, the Import Tool can generate a function that you can call for each file.

For example, suppose that you have a set of spreadsheets in the current folder named `myfile01.xlsx` through `myfile25.xlsx`, and you want to import the same range of data, `A2:G100`, from the first worksheet in each file. Generate code to import the entire set of files as follows:

1. Open one of the files in the Import Tool.
2. From the **Import Selection** button, select **Generate Function**. The Import Tool generates code similar to the following excerpt, and opens the code in the Editor.

   ```
   function data = importfile(workbookFile, sheetName, range)
   %IMPORTFILE    Import numeric data from a spreadsheet
   ...
   ```
3. Save the function.
4. In a separate program file or at the command line, create a `for` loop to import data from each spreadsheet into a cell array named `myData`:

   ```
   numFiles = 25;
   range = 'A2:G100';
   sheet = 1;
   myData = cell(1,numFiles);

   for fileNum = 1:numFiles
       fileName = sprintf('myfile%02d.xlsx',fileNum);
       myData{fileNum} = importfile(fileName,sheet,range);
   end
   ```

Each cell in `myData` contains an array of data from the corresponding worksheet. For example, `myData{1}` contains the data from the first file, `myfile01.xlsx`.

Paste Data from Clipboard

In addition to importing data interactively, you can also paste spreadsheet data from the clipboard into MATLAB.

First, select and copy your spreadsheet data in Microsoft Excel, then use one of the following methods:

- On the Workspace browser title bar, click , and then select **Paste**.
- Open an existing variable in the Variables editor, right-click, and then select **Paste Excel Data**.
- Call `uiimport -pastespecial`.

See Also

`detectImportOptions` | `readcell` | `readmatrix` | `readtable` | `readvars`

More About

- "Define Import Options for Tables" on page 3-17
- "Read Spreadsheet Data into Array or Individual Variables" on page 3-7

Read Spreadsheet Data into Array or Individual Variables

The best way to represent spreadsheet data in MATLAB® is in a table, which can store a mix of numeric and text data. However, sometimes you need to import spreadsheet data as a matrix, a cell array, or separate variables. Based on your data and the data type you need in the MATLAB® workspace, use one of these functions:

- `readmatrix` — Import homogeneous numeric or text data as a matrix.
- `readcell` — Import mixed numeric and text data as a cell array.
- `readvars` — Import spreadsheet columns as separate variables.

Read Spreadsheet Data into Matrix

Import numeric data from `basic_matrix.xls` into a matrix.

```
M = readmatrix('basic_matrix.xls')
```

```
M = 5×4

     6     8     3     1
     5     4     7     3
     1     6     7    10
     4     2     8     2
     2     7     5     9
```

You can also select the data to import from the spreadsheet by specifying the `Sheet` and `Range` parameters. For example, specify the `Sheet` parameter as `'Sheet1'` and the `Range` parameter as `'B1:D3'`. The `readmatrix` function reads a 3-by-3 subset of the data, starting at the element in the first row and second column of the sheet named `'Sheet1'`.

```
M = readmatrix('basic_matrix.xls','Sheet','Sheet1','Range','B1:D3')
```

```
M = 3×3

     8     3     1
     4     7     3
     6     7    10
```

Read Spreadsheet Data into Cell Array

Import the mixed tabular data from `airlinesmall_subset.xlsx` into a cell array.

```
C = readcell('airlinesmall_subset.xlsx');
whos C
```

```
  Name          Size                 Bytes  Class    Attributes

  C          1339x29               4277290  cell
```

You can also select the data to import from the spreadsheet by specifying the `Sheet` and `Range` parameters. For example, specify the `Sheet` parameter as `'2007'` and the `Range` parameter as `'G2:I11'`. The `readcell` function imports ten rows of data for variables in columns 7, 8, and 9, from the worksheet named `'2007'`.

```
subC = readcell('airlinesmall_subset.xlsx','Sheet','2007','Range','G2:I11')
```

```
subC=10×3 cell array
    {[ 935]}    {[ 935]}    {'WN'}
    {[1041]}    {[1040]}    {'WN'}
    {[1430]}    {[1500]}    {'WN'}
    {[ 940]}    {[ 950]}    {'WN'}
    {[1515]}    {[1515]}    {'WN'}
    {[2042]}    {[2035]}    {'WN'}
    {[2116]}    {[2130]}    {'WN'}
    {[1604]}    {[1605]}    {'WN'}
    {[1258]}    {[1230]}    {'WN'}
    {[1134]}    {[1145]}    {'WN'}
```

Read Spreadsheet Data Columns as Separate Variables

Import the first three columns from `airlinesmall_subset.xlsx` as separate workspace variables.

```
[Year,Month,DayOfMonth] = readvars('airlinesmall_subset.xlsx');
whos  Year Month DayOfMonth
```

```
  Name                 Size              Bytes  Class     Attributes

  DayOfMonth        1338x1               10704  double              
  Month             1338x1               10704  double              
  Year              1338x1               10704  double              
```

You can also select which subset to import from the spreadsheet by specifying the `Sheet` and `Range` parameters. For example, import ten rows of the column `DayOfMonth` from the worksheet named `'2004'`. Specify the column and number of rows using the `Range` parameter.

```
DayOfMonth = readvars('airlinesmall_subset.xlsx','Sheet','2004','Range','C2:C11')
```

```
DayOfMonth = 10×1

    26
    10
    21
    24
    20
    20
     1
     2
    30
    11
```

See Also

`readcell` | `readmatrix` | `readtable` | `readvars`

More About

- “Read Spreadsheet Data Using Import Tool” on page 3-4
- “Read Spreadsheet Data into Table” on page 3-9
- “Read Collection or Sequence of Spreadsheet Files” on page 3-12

Read Spreadsheet Data into Table

The best way to represent spreadsheet data in MATLAB® is in a table, which can store a mix of numeric and text data, as well as variable and row names. You can read data into tables interactively or programmatically. To interactively select data, click **Import Data** on the **Home** tab, in the **Variable** section. To programmatically import data, use one of these functions:

- `readtable` — Read a single worksheet.
- `spreadsheetDatastore` — Read multiple worksheets or files.

This example shows how to import spreadsheet data programmatically using both functions. The sample data, `airlinesmall_subset.xlsx`, contains one sheet for each year between 1996 and 2008. The sheet names correspond to the year, such as `2003`.

Read All Data from Worksheet

Call `readtable` to read all the data in the worksheet called 2008, and then display only the first 10 rows and columns. Specify the worksheet name using the `Sheet` name-value pair argument. If your data is on the first worksheet in the file, you do not need to specify `Sheet`.

```
T = readtable('airlinesmall_subset.xlsx','Sheet','2008');
T(1:10,1:10)
```

```
ans=10×10 table
    Year    Month    DayofMonth    DayOfWeek    DepTime    CRSDepTime    ArrTime    CRSArrTime
    ____    _____    __________    _________    _______    __________    _______    __________

    2008      1           3            4         1012         1010        1136         1135
    2008      1           4            5         1303         1300        1411         1415
    2008      1           6            7         2134         2115        2242         2220
    2008      1           7            1         1734         1655          54           30
    2008      1           8            2         1750         1755        2018         2035
    2008      1           9            3          640          645         855          905
    2008      1          10            4         1943         1945        2039         2040
    2008      1          11            5         1303         1305        1401         1400
    2008      1          13            7         1226         1230        1415         1400
    2008      1          14            1         1337         1340        1623         1630
```

Read Selected Range from Specific Worksheet

From the worksheet named `1996`, read only 10 rows of data from the first 5 columns by specifying a range, `'A1:E11'`. The `readtable` function returns a 10-by-5 table.

```
T_selected = readtable('airlinesmall_subset.xlsx','Sheet','1996','Range','A1:E11')
```

```
T_selected=10×5 table
    Year    Month    DayofMonth    DayOfWeek    DepTime
    ____    _____    __________    _________    _______

    1996      1          18            4         2117
    1996      1          12            5         1252
    1996      1          16            2         1441
    1996      1           1            1         2258
    1996      1           4            4         1814
    1996      1          31            3         1822
    1996      1          18            4          729
```

```
    1996      1          26            5         1704
    1996      1          11            4         1858
    1996      1           7            7         2100
```

Convert Variables to Datetimes, Durations, or Categoricals

During the import process, `readtable` automatically detects the data types of the variables. However, if your data contains nonstandard dates, durations, or repeated labels, then you can convert those variable to their correct data types. Converting variables to their correct data types lets you perform efficient computations and comparisons and improves memory usage. For instance, represent the variables `Year`, `Month`, and `DayofMonth` as one `datetime` variable, the `UniqueCarrier` as `categorical`, and `ArrDelay` as `duration` in minutes.

```
data = T(:,{'Year','Month','DayofMonth','UniqueCarrier','ArrDelay'});
data.Date = datetime(data.Year,data.Month,data.DayofMonth);
data.UniqueCarrier = categorical(data.UniqueCarrier);
data.ArrDelay = minutes(data.ArrDelay);
```

Find the day of the year with the longest delay, and then display the date.

```
ind = find(data.ArrDelay == max(data.ArrDelay));
data.Date(ind)
```

```
ans = datetime
   07-Apr-2008
```

Read All Worksheets from Spreadsheet File

A datastore is useful for processing arbitrarily large amounts of data that are spread across multiple worksheets or multiple spreadsheet files. You can perform data import and data processing through the datastore.

Create a datastore from the collection of worksheets in `airlinesmall_subset.xlsx`, select the variables to import, and then preview the data.

```
ds = spreadsheetDatastore('airlinesmall_subset.xlsx');
ds.SelectedVariableNames = {'Year','Month','DayofMonth','UniqueCarrier','ArrDelay'};
preview(ds)
```

```
ans=8×5 table
    Year    Month    DayofMonth    UniqueCarrier    ArrDelay
    ____    _____    __________    _____________    ________

    1996      1          18           {'HP'}            6   
    1996      1          12           {'HP'}           11   
    1996      1          16           {'HP'}          -13   
    1996      1           1           {'HP'}            1   
    1996      1           4           {'US'}           -9   
    1996      1          31           {'US'}            9   
    1996      1          18           {'US'}           -2   
    1996      1          26           {'NW'}          -10   
```

Before importing data, you can specify what data types to use. For this example, import `UniqueCarrier` as a categorical variable.

```
ds.SelectedVariableTypes(4) = {'categorical'};
```

Import data using the `readall` or `read` functions. The `readall` function requires that all the data fit into memory, which is true for the sample data. After the import, compute the maximum arrival delay for this dataset.

```
alldata = readall(ds);
max(alldata.ArrDelay)/60
```

```
ans = 15.2333
```

For large data sets, import portions of the file using the `read` function. For more information, see **Read Collection or Sequence of Spreadsheet Files**.

See Also

`readtable` | `spreadsheetDatastore`

More About

- “Read Spreadsheet Data Using Import Tool” on page 3-4
- “Read Spreadsheet Data into Array or Individual Variables” on page 3-7
- “Read Collection or Sequence of Spreadsheet Files” on page 3-12

Read Collection or Sequence of Spreadsheet Files

When you have data stored across multiple spreadsheet files, use `spreadsheetDatastore` to manage and import the data. After creating the datastore, you can read all the data from the collection simultaneously, or you can read one file at a time.

Data

If the folder `C:\Data` contains a collection of spreadsheet files, then capture the location of the data in `location`. The data used in this example contains `10` spreadsheet files, where each file contains `10` rows of data. Your results will differ based on your files and data.

```
location = 'C:\Data';
dir(location)
```

```
.           ..          File01.xls  File02.xls  File03.xls  File04.xls  File05.xls  File06.xls  F
```

Create Datastore

Create a datastore using the location of the files.

```
ds = spreadsheetDatastore(location)
```

```
ds = 
  SpreadsheetDatastore with properties:

                      Files: {
                             'C:\Data\File01.xls';
                             'C:\Data\File02.xls';
                             'C:\Data\File03.xls'
                              ... and 7 more
                             }
   AlternateFileSystemRoots: {}
                     Sheets: ''
                      Range: ''

  Sheet Format Properties:
             NumHeaderLines: 0
          ReadVariableNames: true
              VariableNames: {'LastName', 'Gender', 'Age' ... and 7 more}
              VariableTypes: {'char', 'char', 'double' ... and 7 more}

  Properties that control the table returned by preview, read, readall:
      SelectedVariableNames: {'LastName', 'Gender', 'Age' ... and 7 more}
      SelectedVariableTypes: {'char', 'char', 'double' ... and 7 more}
                   ReadSize: 'file'
```

Read Data from Datastore

Use the `read` or `readall` functions to import the data from the datastore. If the data from the collection fits in the memory, then you can import it all at once using the `readall` function.

```
allData = readall(ds);
size(allData)
```

```
ans = 1×2
```

```
   100    10
```

Alternatively, you can import the data one file at a time using the `read` function. To control the amount of data imported, before you call `read`, adjust the `ReadSize` property of the datastore. You can set the `ReadSize` to `'file'`, `'sheet'`, or a positive integer.

- If `ReadSize` is `'file'`, then each call to `read` returns data one file at a time.
- If `ReadSize` is `'sheet'`, then each call to `read` returns data one sheet at a time.
- If `ReadSize` is a positive integer, then each call to `read` returns the number of rows specified by `ReadSize`, or fewer if it reaches the end of the data.

```
ds.ReadSize = 'file';
firstFile = read(ds) % reads first file
```

```
firstFile=10×10 table
     LastName       Gender      Age              Location               Height    Weight    Smoker
    __________    ________    ___    _____________________________    ______    ______    _______

    'Smith'       'Male'      38     'County General Hospital'          71       176      'true' 
    'Johnson'     'Male'      43     'VA Hospital'                      69       163      'false'
    'Williams'    'Female'    38     'St. Mary's Medical Center'        64       131      'false'
    'Jones'       'Female'    40     'VA Hospital'                      67       133      'false'
    'Brown'       'Female'    49     'County General Hospital'          64       119      'false'
    'Davis'       'Female'    46     'St. Mary's Medical Center'        68       142      'false'
    'Miller'      'Female'    33     'VA Hospital'                      64       142      'true' 
    'Wilson'      'Male'      40     'VA Hospital'                      68       180      'false'
    'Moore'       'Male'      28     'St. Mary's Medical Center'        68       183      'false'
    'Taylor'      'Female'    31     'County General Hospital'          66       132      'false'
```

```
secondFile = read(ds) % reads second file
```

```
secondFile=10×10 table
     LastName       Gender      Age              Location               Height    Weight    Smoker
    __________    ________    ___    _____________________________    ______    ______    _______

    'Anderson'    'Female'    45     'County General Hospital'          68       128      'false'
    'Thomas'      'Female'    42     'St. Mary's Medical Center'        66       137      'false'
    'Jackson'     'Male'      25     'VA Hospital'                      71       174      'false'
    'White'       'Male'      39     'VA Hospital'                      72       202      'true' 
    'Harris'      'Female'    36     'St. Mary's Medical Center'        65       129      'false'
    'Martin'      'Male'      48     'VA Hospital'                      71       181      'true' 
    'Thompson'    'Male'      32     'St. Mary's Medical Center'        69       191      'true' 
    'Garcia'      'Female'    27     'VA Hospital'                      69       131      'true' 
    'Martinez'    'Male'      37     'County General Hospital'          70       179      'false'
    'Robinson'    'Male'      50     'County General Hospital'          68       172      'false'
```

See Also
`readtable` | `spreadsheetDatastore`

More About
- “Read Spreadsheet Data into Table” on page 3-9

Write Data to Excel Spreadsheets

In this section...

Write Tabular Data to Spreadsheet File

To export a table in the workspace to a Microsoft® Excel® spreadsheet file, use the `writetable` function. You can export data from the workspace to any worksheet in the file, and to any location within that worksheet. By default, `writetable` writes your table data to the first worksheet in the file, starting at cell `A1`.

For example, create a sample table of column-oriented data and display the first five rows.

```
load patients.mat
T = table(LastName,Age,Weight,Smoker);
T(1:5,:)
```

```
ans=5×4 table
      LastName       Age    Weight    Smoker
    ____________    ___    ______    ______

    {'Smith'   }    38      176      true  
    {'Johnson' }    43      163      false 
    {'Williams'}    38      131      false 
    {'Jones'   }    40      133      false 
    {'Brown'   }    49      119      false 
```

Write table `T` to the first sheet in a new spreadsheet file named `patientdata.xlsx`, starting at cell `D1`. To specify the portion of the worksheet you want to write to, use the `Range` name-value pair argument. By default, `writetable` writes the table variable names as column headings in the spreadsheet file.

```
filename = 'patientdata.xlsx';
writetable(T,filename,'Sheet',1,'Range','D1')
```

Write the table `T` without the variable names to a new sheet called `'MyNewSheet'`. To write the data without the variable names, specify the name-value pair `WriteVariableNames` as `false`.

```
writetable(T,filename,'Sheet','MyNewSheet','WriteVariableNames',false);
```

Write Numeric and Text Data to Spreadsheet File

To export a numeric array and a cell array to a Microsoft Excel spreadsheet file, use the `writematrix` or `writecell` functions. You can export data in individual numeric and text workspace variables to any worksheet in the file, and to any location within that worksheet. By default, the import functions write your matrix data to the first worksheet in the file, starting at cell `A1`.

For example, create a sample array of numeric data, `A`, and a sample cell array of text and numeric data, `C`.

```
A = magic(5)
C = {'Time', 'Temp'; 12 98; 13 'x'; 14 97}
```

```
A =

    17    24     1     8    15
    23     5     7    14    16
     4     6    13    20    22
    10    12    19    21     3
    11    18    25     2     9


C = 

    'Time'    'Temp'
    [  12]    [  98]
    [  13]    'x'   
    [  14]    [  97]
```

Write array `A` to the 5-by-5 rectangular region, `E1:I5`, on the first sheet in a new spreadsheet file named `testdata.xlsx`.

```
filename = 'testdata.xlsx';
writematrix(A,filename,'Sheet',1,'Range','E1:I5')
```

Write cell array `C` to a rectangular region that starts at cell `B2` on a worksheet named `Temperatures`. You can specify range using only the first cell.

```
writecell(C,filename,'Sheet','Temperatures','Range','B2');
```

`writecell` displays a warning because the worksheet, `Temperatures`, did not previously exist, but you can disable this warning.

Disable Warning When Adding New Worksheet

If the target worksheet does not exist in the file, then the `writetable` and `writecell` functions display this warning:

```
Warning: Added specified worksheet.
```

For information on how to suppress warning messages, see “Suppress Warnings”.

Format Cells in Excel Files

To write data to Excel files on Windows systems with custom formats (such as fonts or colors), access the COM server directly using `actxserver` rather than `writetable`, `writetimetable`, `writematrix`, or `writecell`. For example, Technical Solution 1-QLD4K uses `actxserver` to establish a connection between MATLAB and Excel, write data to a worksheet, and specify the colors of the cells.

For more information, see “Get Started with COM”.

See Also
`writecell` | `writematrix` | `writetable`

Define Import Options for Tables

Typically, you can import tables using the readtable function. However, sometimes importing tabular data requires additional control over the import process. For example, you might want to select the variables to import or handle rows with missing or error-causing data. To control the import process, you can create an import options object. The object has properties that you can adjust based on your import needs.

Create Import Options

To create an import options object for a sample data set, airlinesmall.csv, use the detectImportOptions function. The detectImportOptions function creates a DelimitedTextImportOptions object for this text file. For a full list of properties of the import options object, see the detectImportOptions reference page.

```
opts = detectImportOptions('airlinesmall.csv');
```

Customize Table-Level Import Options

The import options object has properties that you can adjust to control the import process. Some properties apply to the entire table while others apply to specific variables. Properties that affect the entire table include rules to manage error-causing or missing data. For example, remove rows with data that cause import errors by setting the ImportErrorRule to 'omitrow'. Replace missing values by setting the MissingRule to 'fill'. The FillValue property value determines what value replaces the missing values. For example, you can replace missing values with NaN.

```
opts.ImportErrorRule = 'omitrow';
opts.MissingRule = 'fill';
```

Customize Variable-Level Import Options

To get and set options for specific variables use the getvaropts, setvartype, and setvaropts functions. For example, view the current options for the variables named FlightNum, Origin, Dest, and ArrDelay, using the getvaropts function.

```
getvaropts(opts,{'FlightNum','Origin','Dest','ArrDelay'});
```

Change the data types for the variables using the setvartype function:

- Since the values in the variable FlightNum are identifiers for the flight and not numerical values, change its data type to char.
- Since the variables Origin and Dest designate a finite set of repeating text values, change their data type to categorical.

```
opts = setvartype(opts,{'FlightNum','Origin','Dest','ArrDelay'},...
                       {'char','categorical','categorical','single'});
```

Change other properties using the setvaropts function:

- For the FlightNum variable, remove any leading white spaces from the text by setting the WhiteSpaceRule property to trimleading.
- For the ArrDelay variable, replace fields containing 0 or NA with the value specified in FillValue property by setting the TreatAsMissing property.

```
opts = setvaropts(opts,'FlightNum','WhitespaceRule','trimleading');
opts = setvaropts(opts,'ArrDelay','TreatAsMissing',{'0','NA'});
```

Import Table

Specify the variables to get, import them using `readtable`, and display the first 8 rows of the table.

```
opts.SelectedVariableNames = {'FlightNum','Origin','Dest','ArrDelay'};
T = readtable('airlinesmall.csv',opts);
T(1:8,:)
```

```
ans=8×4 table
    FlightNum    Origin    Dest    ArrDelay
    _________    ______    ____    ________

    {'1503'}      LAX      SJC         8
    {'1550'}      SJC      BUR         8
    {'1589'}      SAN      SMF        21
    {'1655'}      BUR      SJC        13
    {'1702'}      SMF      LAX         4
    {'1729'}      LAX      SJC        59
    {'1763'}      SAN      SFO         3
    {'1800'}      SEA      LAX        11
```

See Also

`DelimitedTextImportOptions` | `SpreadsheetImportOptions` | `detectImportOptions` | `getvaropts` | `readcell` | `readmatrix` | `readtable` | `readvars` | `setvaropts` | `setvartype`

More About

- “Read Spreadsheet Data Using Import Tool” on page 3-4
- “Read Spreadsheet Data into Table” on page 3-9

Low-Level File I/O

Import Text Data Files with Low-Level I/O

Overview

Low-level file I/O functions allow the most control over reading or writing data to a file. However, these functions require that you specify more detailed information about your file than the easier-to-use *high-level functions*, such as `importdata`. For more information on the high-level functions that read text files, see "Import Text Files" on page 2-2.

If the high-level functions cannot import your data, use one of the following:

- `fscanf`, which reads formatted data in a text or ASCII file; that is, a file you can view in a text editor. For more information, see "Reading Data in a Formatted Pattern" on page 4-2.
- `fgetl` and `fgets`, which read one line of a file at a time, where a newline character separates each line. For more information, see "Reading Data Line-by-Line" on page 4-4.
- `fread`, which reads a stream of data at the byte or bit level. For more information, see "Import Binary Data with Low-Level I/O" on page 4-8.

For additional information, see:

- "Testing for End of File (EOF)" on page 4-5
- "Opening Files with Different Character Encodings" on page 4-7

Note The low-level file I/O functions are based on functions in the ANSI® Standard C Library. However, MATLAB includes *vectorized* versions of the functions, to read and write data in an array with minimal control loops.

Reading Data in a Formatted Pattern

To import text files that `importdata` and `textscan` cannot read, consider using `fscanf`. The `fscanf` function requires that you describe the format of your file, but includes many options for this format description.

For example, create a text file `mymeas.dat` as shown. The data in `mymeas.dat` includes repeated sets of times, dates, and measurements. The header text includes the number of sets of measurements, `N`:

```
Measurement Data
N=3

12:00:00
```

```
01-Jan-1977
4.21  6.55  6.78  6.55
9.15  0.35  7.57  NaN
7.92  8.49  7.43  7.06
9.59  9.33  3.92  0.31
09:10:02
23-Aug-1990
2.76  6.94  4.38  1.86
0.46  3.17  NaN   4.89
0.97  9.50  7.65  4.45
8.23  0.34  7.95  6.46
15:03:40
15-Apr-2003
7.09  6.55  9.59  7.51
7.54  1.62  3.40  2.55
NaN   1.19  5.85  5.05
6.79  4.98  2.23  6.99
```

Opening the File

As with any of the low-level I/O functions, before reading, open the file with `fopen`, and obtain a file identifier. By default, `fopen` opens files for read access, with a permission of `'r'`.

When you finish processing the file, close it with `fclose(`*`fid`*`)`.

Describing the Data

Describe the data in the file with format specifiers, such as `'%s'` for text, `'%d'` for an integer, or `'%f'` for a floating-point number. (For a complete list of specifiers, see the `fscanf` reference page.)

To skip literal characters in the file, include them in the format description. To skip a data field, use an asterisk (`'*'`) in the specifier.

For example, consider the header lines of `mymeas.dat`:

```
Measurement Data   % skip the first 2 words, go to next line:  %*s %*s\n
N=3                % ignore 'N=', read integer:  N=%d\n
                   % go to next line:  \n
12:00:00
01-Jan-1977
4.21  6.55  6.78  6.55
...
```

To read the headers and return the single value for `N`:

```
N = fscanf(fid, '%*s %*s\nN=%d\n\n', 1);
```

Specifying the Number of Values to Read

By default, `fscanf` reapplies your format description until it cannot match the description to the data, or it reaches the end of the file.

Optionally, specify the number of values to read, so that `fscanf` does not attempt to read the entire file. For example, in `mymeas.dat`, each set of measurements includes a fixed number of rows and columns:

```
measrows = 4;
meascols = 4;
meas  = fscanf(fid, '%f', [measrows, meascols])';
```

Creating Variables in the Workspace

There are several ways to store `mymeas.dat` in the MATLAB workspace. In this case, read the values into a structure. Each element of the structure has three fields: `mtime`, `mdate`, and `meas`.

Note `fscanf` fills arrays with numeric values in column order. To make the output array match the orientation of numeric data in a file, transpose the array.

```
filename = 'mymeas.dat';
measrows = 4;
meascols = 4;

% open the file
fid = fopen(filename);

% read the file headers, find N (one value)
N = fscanf(fid, '%*s %*s\nN=%d\n\n', 1);

% read each set of measurements
for n = 1:N
    mystruct(n).mtime = fscanf(fid, '%s', 1);
    mystruct(n).mdate = fscanf(fid, '%s', 1);

    % fscanf fills the array in column order,
    % so transpose the results
    mystruct(n).meas  = ...
      fscanf(fid, '%f', [measrows, meascols])';
end

% close the file
fclose(fid);
```

Reading Data Line-by-Line

MATLAB provides two functions that read lines from files and store them as character vectors: `fgetl` and `fgets`. The `fgets` function copies the line along with the newline character to the output, but `fgetl` does not.

The following example uses `fgetl` to read an entire file one line at a time. The function `litcount` determines whether a given character sequence (`literal`) appears in each line. If it does, the function prints the entire line preceded by the number of times the literal appears on the line.

```
function y = litcount(filename, literal)
% Count the number of times a given literal appears in each line.

fid = fopen(filename);
y = 0;
tline = fgetl(fid);
while ischar(tline)
   matches = strfind(tline, literal);
   num = length(matches);
```

```
   if num > 0
      y = y + num;
      fprintf(1,'%d:%s\n',num,tline);
   end
   tline = fgetl(fid);
end
fclose(fid);
```

Create an input data file called `badpoem`:

```
Oranges and lemons,
Pineapples and tea.
Orangutans and monkeys,
Dragonflys or fleas.
```

To find out how many times `'an'` appears in this file, call `litcount`:

```
litcount('badpoem','an')
```

This returns:

```
2: Oranges and lemons,
1: Pineapples and tea.
3: Orangutans and monkeys,
ans =
     6
```

Testing for End of File (EOF)

When you read a portion of your data at a time, you can use `feof` to check whether you have reached the end of the file. `feof` returns a value of `1` when the file pointer is at the end of the file. Otherwise, it returns `0`.

Note Opening an empty file does *not* move the file position indicator to the end of the file. Read operations, and the `fseek` and `frewind` functions, move the file position indicator.

Testing for EOF with feof

When you use `textscan`, `fscanf`, or `fread` to read portions of data at a time, use `feof` to check whether you have reached the end of the file.

For example, suppose that the hypothetical file `mymeas.dat` has the following form, with no information about the number of measurement sets. Read the data into a structure with fields for `mtime`, `mdate`, and `meas`:

```
12:00:00
01-Jan-1977
4.21  6.55  6.78  6.55
9.15  0.35  7.57  NaN
7.92  8.49  7.43  7.06
9.59  9.33  3.92  0.31
09:10:02
23-Aug-1990
2.76  6.94  4.38  1.86
0.46  3.17  NaN   4.89
```

```
0.97  9.50  7.65  4.45
8.23  0.34  7.95  6.46
```

To read the file:

```
filename = 'mymeas.dat';
measrows = 4;
meascols = 4;

% open the file
fid = fopen(filename);

% make sure the file is not empty
finfo = dir(filename);
fsize = finfo.bytes;

if fsize > 0

    % read the file
    block = 1;
    while ~feof(fid)
        mystruct(block).mtime = fscanf(fid, '%s', 1);
        mystruct(block).mdate = fscanf(fid, '%s', 1);

        % fscanf fills the array in column order,
        % so transpose the results
        mystruct(block).meas  = ...
          fscanf(fid, '%f', [measrows, meascols])';

        block = block + 1;
    end

end

% close the file
fclose(fid);
```

Testing for EOF with fgetl and fgets

If you use `fgetl` or `fgets` in a control loop, `feof` is not always the best way to test for end of file. As an alternative, consider checking whether the value that `fgetl` or `fgets` returns is a character vector.

For example, the function `litcount` described in "Reading Data Line-by-Line" on page 4-4 includes the following `while` loop and `fgetl` calls :

```
y = 0;
tline = fgetl(fid);
while ischar(tline)
   matches = strfind(tline, literal);
   num = length(matches);
   if num > 0
      y = y + num;
      fprintf(1,'%d:%s\n',num,tline);
   end
   tline = fgetl(fid);
end
```

This approach is more robust than testing `~feof(fid)` for two reasons:

- If `fgetl` or `fgets` find data, they return a character vector. Otherwise, they return a number (`-1`).
- After each read operation, `fgetl` and `fgets` check the next character in the file for the end-of-file marker. Therefore, these functions sometimes set the end-of-file indicator *before* they return a value of `-1`. For example, consider the following three-line text file. Each of the first two lines ends with a newline character, and the third line contains only the end-of-file marker:

```
123
456
```

  Three sequential calls to `fgetl` yield the following results:

```
t1 = fgetl(fid);    % t1 = '123', feof(fid) = false
t2 = fgetl(fid);    % t2 = '456', feof(fid) = true
t3 = fgetl(fid);    % t3 = -1,    feof(fid) = true
```

  This behavior does not conform to the ANSI specifications for the related C language functions.

Opening Files with Different Character Encodings

Encoding schemes support the characters required for particular alphabets, such as those for Japanese or European languages. Common encoding schemes include US-ASCII or UTF-8.

If you do not specify an encoding scheme when opening a file for reading, `fopen` uses auto character-set detection to determine the encoding. If you do not specify an encoding scheme when opening a file for writing, `fopen` defaults to using UTF-8 in order to provide interoperability between all platforms and locales without data loss or corruption. For more information, see .

To determine the default, open a file, and call `fopen` again with the syntax:

```
[filename, permission, machineformat, encoding] = fopen(fid);
```

If you specify an encoding scheme when you open a file, the following functions apply that scheme: `fscanf`, `fprintf`, `fgetl`, `fgets`, `fread`, and `fwrite`.

For a complete list of supported encoding schemes, and the syntax for specifying the encoding, see the `fopen` reference page.

Import Binary Data with Low-Level I/O

In this section...

Low-Level Functions for Importing Data

Low-level file I/O functions allow the most direct control over reading or writing data to a file. However, these functions require that you specify more detailed information about your file than the easier-to-use *high-level functions*. For a complete list of high-level functions and the file formats they support, see "Supported File Formats for Import and Export" on page 1-2.

If the high-level functions cannot import your data, use one of the following:

- `fscanf`, which reads formatted data in a text or ASCII file; that is, a file you can view in a text editor. For more information, see "Reading Data in a Formatted Pattern" on page 4-2.
- `fgetl` and `fgets`, which read one line of a file at a time, where a newline character separates each line. For more information, see "Reading Data Line-by-Line" on page 4-4.
- `fread`, which reads a stream of data at the byte or bit level. For more information, see "Reading Binary Data in a File" on page 4-8.

Note The low-level file I/O functions are based on functions in the ANSI Standard C Library. However, MATLAB includes *vectorized* versions of the functions, to read and write data in an array with minimal control loops.

Reading Binary Data in a File

As with any of the low-level I/O functions, before importing, open the file with `fopen`, and obtain a file identifier. When you finish processing a file, close it with `fclose(`*`fileID`*`)`.

By default, `fread` reads a file 1 byte at a time, and interprets each byte as an 8-bit unsigned integer (`uint8`). `fread` creates a column vector, with one element for each byte in the file. The values in the column vector are of class `double`.

For example, consider the file `nine.bin`, created as follows:

```
fid = fopen('nine.bin','w');
fwrite(fid, [1:9]);
fclose(fid);
```

To read all data in the file into a 9-by-1 column vector of class `double`:

```
fid = fopen('nine.bin');
col9 = fread(fid);
fclose(fid);
```

Changing the Dimensions of the Array

By default, `fread` reads all values in the file into a column vector. However, you can specify the number of values to read, or describe a two-dimensional output matrix.

For example, to read `nine.bin`, described in the previous example:

```
fid = fopen('nine.bin');

% Read only the first six values
col6 = fread(fid, 6);

% Return to the beginning of the file
frewind(fid);

% Read first four values into a 2-by-2 matrix
frewind(fid);
two_dim4 = fread(fid, [2, 2]);

% Read into a matrix with 3 rows and
% unspecified number of columns
frewind(fid);
two_dim9 = fread(fid, [3, inf]);

% Close the file
fclose(fid);
```

Describing the Input Values

If the values in your file are not 8-bit unsigned integers, specify the size of the values.

For example, consider the file `fpoint.bin`, created with double-precision values as follows:

```
myvals = [pi, 42, 1/3];

fid = fopen('fpoint.bin','w');
fwrite(fid, myvals, 'double');
fclose(fid);
```

To read the file:

```
fid = fopen('fpoint.bin');

% read, and transpose so samevals = myvals
samevals = fread(fid, 'double')';

fclose(fid);
```

For a complete list of precision descriptions, see the `fread` function reference page.

Saving Memory

By default, `fread` creates an array of class `double`. Storing double-precision values in an array requires more memory than storing characters, integers, or single-precision values.

To reduce the amount of memory required to store your data, specify the class of the array using one of the following methods:

- Match the class of the input values with an asterisk (`'*'`). For example, to read single-precision values into an array of class `single`, use the command:

```
mydata = fread(fid,'*single')
```

- Map the input values to a new class with the `'=>'` symbol. For example, to read `uint8` values into an `uint16` array, use the command:

```
mydata = fread(fid,'uint8=>uint16')
```

For a complete list of precision descriptions, see the `fread` function reference page.

Reading Portions of a File

MATLAB low-level functions include several options for reading portions of binary data in a file:

- Read a specified number of values at a time, as described in “Changing the Dimensions of the Array” on page 4-9. Consider combining this method with “Testing for End of File” on page 4-10.
- Move to a specific location in a file to begin reading. For more information, see “Moving within a File” on page 4-10.
- Skip a certain number of bytes or bits after each element read. For an example, see “Write and Read Complex Numbers” on page 4-21.

Testing for End of File

When you open a file, MATLAB creates a pointer to indicate the current position within the file.

Note Opening an empty file does *not* move the file position indicator to the end of the file. Read operations, and the `fseek` and `frewind` functions, move the file position indicator.

Use the `feof` function to check whether you have reached the end of a file. `feof` returns a value of `1` when the file pointer is at the end of the file. Otherwise, it returns `0`.

For example, read a large file in parts:

```
filename = 'largedata.dat';        % hypothetical file
segsize = 10000;

fid = fopen(filename);

while ~feof(fid)
    currData = fread(fid, segsize);
    if ~isempty(currData)
        disp('Current Data:');
        disp(currData);
    end
end

fclose(fid);
```

Moving within a File

To read or write selected portions of data, move the file position indicator to any location in the file. For example, call `fseek` with the syntax

```
fseek(fid,offset,origin);
```

where:

- *fid* is the file identifier obtained from fopen.
- *offset* is a positive or negative offset value, specified in bytes.
- *origin* specifies the location from which to calculate the position:

'bof'	Beginning of file
'cof'	Current position in file
'eof'	End of file

Alternatively, to move easily to the beginning of a file:

```
frewind(fid);
```

Use ftell to find the current position within a given file. ftell returns the number of bytes from the beginning of the file.

For example, create a file five.bin:

```
A = 1:5;
fid = fopen('five.bin','w');
fwrite(fid, A,'short');
fclose(fid);
```

Because the call to fwrite specifies the short format, each element of A uses two storage bytes in five.bin.

Reopen five.bin for reading:

```
fid = fopen('five.bin','r');
```

Move the file position indicator forward 6 bytes from the beginning of the file:

```
status = fseek(fid,6,'bof');
```

File Position	bof	1	2	3	4	5	6	7	8	9	10	eof
File Contents		0	1	0	2	0	3	0	4	0	5	
File Position Indicator							↑					

Read the next element:

```
four = fread(fid,1,'short');
```

The act of reading advances the file position indicator. To determine the current file position indicator, call ftell:

```
position = ftell(fid)

position =
     8
```

File Position	bof	1	2	3	4	5	6	7	8	9	10	eof
File Contents		0	1	0	2	0	3	0	4	0	5	
File Position Indicator									↑			

To move the file position indicator back 4 bytes, call `fseek` again:

```
status = fseek(fid,-4,'cof');
```

File Position	bof	1	2	3	4	5	6	7	8	9	10	eof
File Contents		0	1	0	2	0	3	0	4	0	5	
File Position Indicator					↑							

Read the next value:

```
three = fread(fid,1,'short');
```

Reading Files Created on Other Systems

Different operating systems store information differently at the byte or bit level:

- *Big-endian* systems store bytes starting with the largest address in memory (that is, they start with the big end).
- *Little-endian* systems store bytes starting with the smallest address (the little end).

Windows systems use little-endian byte ordering, and UNIX systems use big-endian byte ordering.

To read a file created on an opposite-endian system, specify the byte ordering used to create the file. You can specify the ordering in the call to open the file, or in the call to read the file.

For example, consider a file with double-precision values named `little.bin`, created on a little-endian system. To read this file on a big-endian system, use one (or both) of the following commands:

- Open the file with

  ```
  fid = fopen('little.bin', 'r', 'l')
  ```
- Read the file with

  ```
  mydata = fread(fid, 'double', 'l')
  ```

where `'l'` indicates little-endian ordering.

If you are not sure which byte ordering your system uses, call the `computer` function:

```
[cinfo, maxsize, ordering] = computer
```

The returned *`ordering`* is `'L'` for little-endian systems, or `'B'` for big-endian systems.

Export to Text Data Files with Low-Level I/O

In this section...

Write to Text Files Using fprintf

This example shows how to create text files, including combinations of numeric and character data and nonrectangular files, using the low-level `fprintf` function.

`fprintf` is based on its namesake in the ANSI® Standard C Library. However, MATLAB® uses a vectorized version of `fprintf` that writes data from an array with minimal control loops.

Open the File

Create a sample matrix `y` with two rows.

```
x = 0:0.1:1;
y = [x; exp(x)];
```

Open a file for writing with `fopen` and obtain a file identifier, `fileID`. By default, `fopen` opens a file for read-only access, so you must specify the permission to write or append, such as `'w'` or `'a'`.

```
fileID = fopen('exptable.txt','w');
```

Write to the File

Write a title, followed by a blank line using the `fprintf` function. To move to a new line in the file, use `'\n'`.

```
fprintf(fileID, 'Exponential Function\n\n');
```

Note: Some Windows® text editors, including Microsoft® Notepad, require a newline character sequence of `'\r\n'` instead of `'\n'`. However, `'\n'` is sufficient for Microsoft Word or WordPad.

Write the values in `y` in column order so that two values appear in each row of the file. `fprintf` converts the numbers or characters in the array inputs to text according to your specifications. Specify `'%f'` to print floating-point numbers.

```
fprintf(fileID,'%f %f\n',y);
```

Other common conversion specifiers include `'%d'` for integers or `'%s'` for characters. `fprintf` reapplies the conversion information to cycle through all values of the input arrays in column order.

Close the file using `fclose` when you finish writing.

```
fclose(fileID);
```

View the contents of the file using the `type` function.

```
type exptable.txt
```

```
Exponential Function

0.000000 1.000000
0.100000 1.105171
0.200000 1.221403
0.300000 1.349859
0.400000 1.491825
0.500000 1.648721
0.600000 1.822119
0.700000 2.013753
0.800000 2.225541
0.900000 2.459603
1.000000 2.718282
```

Additional Formatting Options

Optionally, include additional information in the call to `fprintf` to describe field width, precision, or the order of the output values. For example, specify the field width and number of digits to the right of the decimal point in the exponential table.

```
fileID = fopen('exptable_new.txt', 'w');

fprintf(fileID,'Exponential Function\n\n');
fprintf(fileID,'%6.2f %12.8f\n', y);

fclose(fileID);
```

View the contents of the file.

```
type exptable_new.txt
```

```
Exponential Function

  0.00   1.00000000
  0.10   1.10517092
  0.20   1.22140276
  0.30   1.34985881
  0.40   1.49182470
  0.50   1.64872127
  0.60   1.82211880
  0.70   2.01375271
  0.80   2.22554093
  0.90   2.45960311
  1.00   2.71828183
```

Append To or Overwrite Existing Text Files

This example shows how to append values to an existing text file, rewrite the entire file, and overwrite only a portion of the file.

By default, `fopen` opens files with read access. To change the type of file access, use the permission specifier in the call to `fopen`. Possible permission specifiers include:

- `'r'` for reading
- `'w'` for writing, discarding any existing contents of the file
- `'a'` for appending to the end of an existing file

To open a file for both reading and writing or appending, attach a plus sign to the permission, such as 'w+' or 'a+'. If you open a file for both reading and writing, you must call fseek or frewind between read and write operations.

Append to Existing Text File

Create a file named changing.txt.

```
fileID = fopen('changing.txt','w');
fmt = '%5d %5d %5d %5d\n';
fprintf(fileID,fmt, magic(4));
fclose(fileID);
```

The current contents of changing.txt are:

16 5 9 4

2 11 7 14

3 10 6 15

13 8 12 1

Open the file with permission to append.

```
fileID = fopen('changing.txt','a');
```

Write the values [55 55 55 55] at the end of file:

```
fprintf(fileID,fmt,[55 55 55 55]);
```

Close the file.

```
fclose(fileID);
```

View the contents of the file using the type function.

```
type changing.txt
```

```
   16     5     9     4
    2    11     7    14
    3    10     6    15
   13     8    12     1
   55    55    55    55
```

Overwrite Entire Text File

A text file consists of a contiguous set of characters, including newline characters. To replace a line of the file with a different number of characters, you must rewrite the line that you want to change and all subsequent lines in the file.

Replace the first line of changing.txt with longer, descriptive text. Because the change applies to the first line, rewrite the entire file.

```
replaceLine = 1;
numLines = 5;
newText = 'This file originally contained a magic square';
```

```
fileID = fopen('changing.txt','r');
mydata = cell(1, numLines);
for k = 1:numLines
   mydata{k} = fgetl(fileID);
end
fclose(fileID);

mydata{replaceLine} = newText;

fileID = fopen('changing.txt','w');
fprintf(fileID,'%s\n',mydata{:});
fclose(fileID);
```

View the contents of the file.

```
type changing.txt
```

```
This file originally contained a magic square
    2    11     7    14
    3    10     6    15
   13     8    12     1
   55    55    55    55
```

Overwrite Portion of Text File

Replace the third line of changing.txt with `[33 33 33 33]`. If you want to replace a portion of a text file with exactly the same number of characters, you do not need to rewrite any other lines in the file.

```
replaceLine = 3;
myformat = '%5d %5d %5d %5d\n';
newData = [33 33 33 33];
```

Move the file position marker to the correct line.

```
fileID = fopen('changing.txt','r+');
for k=1:(replaceLine-1);
   fgetl(fileID);
end
```

Call `fseek` between read and write operations.

```
fseek(fileID,0,'cof');

fprintf(fileID, myformat, newData);
fclose(fileID);
```

View the contents of the file.

```
type changing.txt
```

```
This file originally contained a magic square
    2    11     7    14
   33    33    33    33
```

```
    13     8    12     1
    55    55    55    55
```

Open Files with Different Character Encodings

Encoding schemes support the characters required for particular alphabets, such as those for Japanese or European languages. Common encoding schemes include US-ASCII or UTF-8.

If you do not specify an encoding scheme when opening a file for reading, `fopen` uses auto character-set detection to determine the encoding. If you do not specify an encoding scheme when opening a file for writing, `fopen` defaults to using UTF-8 in order to provide interoperability between all platforms and locales without data loss or corruption. For more information, see .

To determine the default, open a file, and call `fopen` again with the syntax:

```
[filename, permission, machineformat, encoding] = fopen(fid);
```

If you specify an encoding scheme when you open a file, the following functions apply that scheme: `fscanf`, `fprintf`, `fgetl`, `fgets`, `fread`, and `fwrite`.

For a complete list of supported encoding schemes, and the syntax for specifying the encoding, see the `fopen` reference page.

See Also

`fopen` | `fprintf` | `fseek`

More About

- “Formatting Text”
- “Write Data to Text Files” on page 2-21

Export Binary Data with Low-Level I/O

In this section...

Low-Level Functions for Exporting Data

Low-level file I/O functions allow the most direct control over reading or writing data to a file. However, these functions require that you specify more detailed information about your file than the easier-to-use *high-level functions*. For a complete list of high-level functions and the file formats they support, see "Supported File Formats for Import and Export" on page 1-2.

If the high-level functions cannot export your data, use one of the following:

- `fprintf`, which writes formatted data to a text or ASCII file; that is, a file you can view in a text editor or import into a spreadsheet. For more information, see "Export to Text Data Files with Low-Level I/O" on page 4-13.
- `fwrite`, which writes a stream of binary data to a file. For more information, see "Write Binary Data to a File" on page 4-18.

Note The low-level file I/O functions are based on functions in the ANSI Standard C Library. However, MATLAB includes *vectorized* versions of the functions, to read and write data in an array with minimal control loops.

Write Binary Data to a File

This example shows how to use the `fwrite` function to export a stream of binary data to a file.

Create a file named `nine.bin` with the integers from 1 to 9. As with any of the low-level I/O functions, before writing, open or create a file with `fopen` and obtain a file identifier.

```
fileID = fopen('nine.bin','w');
fwrite(fileID, [1:9]);
```

By default, `fwrite` writes values from an array in column order as 8-bit unsigned integers (`uint8`).

When you finish processing a file, close it with `fclose`.

```
fclose(fileID);
```

Create a file with double-precision values. You must specify the precision of the values if the values in your matrix are not 8-bit unsigned integers.

```
mydata = [pi 42 1/3];

fileID = fopen('double.bin','w');
```

```
fwrite(fileID,mydata,'double');
fclose(fileID);
```

Overwrite or Append to an Existing Binary File

This example shows how to overwrite a portion of an existing binary file and append values to the file.

By default, `fopen` opens files with read access. To change the type of file access, use the permission specifier in the call to `fopen`. Possible permission specifiers include:

- `'r'` for reading
- `'w'` for writing, discarding any existing contents of the file
- `'a'` for appending to the end of an existing file

To open a file for both reading and writing or appending, attach a plus sign to the permission, such as `'w+'` or `'a+'`. If you open a file for both reading and writing, you must call `fseek` or `frewind` between read and write operations.

Overwrite a Portion of an Existing File

Create a file named `magic4.bin`, specifying permission to write and read.

```
fileID = fopen('magic4.bin','w+');
fwrite(fileID,magic(4));
```

The original magic(4) matrix is:

`16 2 3 13`

`5 11 10 8`

`9 7 6 12`

`4 14 15 1`

The file contains 16 bytes, 1 for each value in the matrix.

Replace the values in the second column of the matrix with the vector, `[44 44 44 44]`. To do this, first seek to the fourth byte from the beginning of the file using `fseek`.

```
fseek(fileID,4,'bof');
```

Write the vector `[44 44 44 44]` using `fwrite`.

```
fwrite(fileID,[44 44 44 44]);
```

Read the results from the file into a 4-by-4 matrix.

```
frewind(fileID);
newdata = fread(fileID,[4,4])
```

```
newdata = 4×4

    16    44     3    13
     5    44    10     8
```

```
     9    44     6    12
     4    44    15     1
```

Close the file.

```
fclose(fileID);
```

Append Binary Data to Existing File

Append the values `[55 55 55 55]` to `magic4.bin`. First. open the file with permission to append and read.

```
fileID = fopen('magic4.bin','a+');
```

Write values at end of file.

```
fwrite(fileID,[55 55 55 55]);
```

Read the results from the file into a 4-by-5 matrix.

```
frewind(fileID);
appended = fread(fileID, [4,5])
```

```
appended = 4×5

    16    44     3    13    55
     5    44    10     8    55
     9    44     6    12    55
     4    44    15     1    55
```

Close the file.

```
fclose(fileID);
```

Create a File for Use on a Different System

Different operating systems store information differently at the byte or bit level:

- *Big-endian* systems store bytes starting with the largest address in memory (that is, they start with the big end).
- *Little-endian* systems store bytes starting with the smallest address (the little end).

Windows systems use little-endian byte ordering, and UNIX systems use big-endian byte ordering.

To create a file for use on an opposite-endian system, specify the byte ordering for the target system. You can specify the ordering in the call to open the file, or in the call to write the file.

For example, to create a file named `myfile.bin` on a big-endian system for use on a little-endian system, use one (or both) of the following commands:

- Open the file with

  ```
  fid = fopen('myfile.bin', 'w', 'l')
  ```
- Write the file with

```
fwrite(fid, mydata, precision, 'l')
```

where `'l'` indicates little-endian ordering.

If you are not sure which byte ordering your system uses, call the `computer` function:

```
[cinfo, maxsize, ordering] = computer
```

The returned *`ordering`* is `'L'` for little-endian systems, or `'B'` for big-endian systems.

Write and Read Complex Numbers

This example shows how to write and read complex numbers in binary files.

The available precision values for `fwrite` do not explicitly support complex numbers. To store complex numbers in a file, separate the real and imaginary components and write them separately to the file. There are two ways to do this:

- Write all real components followed by all imaginary components
- Interleave the components

Use the approach that allows you to read the data in your target application.

Separate Real and Imaginary Components

Create an array that contains complex values.

```
nrows = 5;
ncols = 5;
z = complex(rand(nrows, ncols), rand(nrows, ncols))
```

```
z = 5×5 complex

   0.8147 + 0.7577i   0.0975 + 0.7060i   0.1576 + 0.8235i   0.1419 + 0.4387i   0.6557 + 0.4898i
   0.9058 + 0.7431i   0.2785 + 0.0318i   0.9706 + 0.6948i   0.4218 + 0.3816i   0.0357 + 0.4456i
   0.1270 + 0.3922i   0.5469 + 0.2769i   0.9572 + 0.3171i   0.9157 + 0.7655i   0.8491 + 0.6463i
   0.9134 + 0.6555i   0.9575 + 0.0462i   0.4854 + 0.9502i   0.7922 + 0.7952i   0.9340 + 0.7094i
   0.6324 + 0.1712i   0.9649 + 0.0971i   0.8003 + 0.0344i   0.9595 + 0.1869i   0.6787 + 0.7547i
```

Separate the complex values into real and imaginary components.

```
z_real = real(z);
z_imag = imag(z);
```

Write All Real Components Followed By Imaginary Components

Write all the real components, `z_real`, followed by all the imaginary components, `z_imag`, to a file named `complex_adj.bin`.

```
adjacent = [z_real z_imag];

fileID = fopen('complex_adj.bin', 'w');
fwrite(fileID,adjacent,'double');
fclose(fileID);
```

Read the values from the file using `fread`.

```
fileID = fopen('complex_adj.bin');
same_real = fread(fileID, [nrows, ncols], 'double');
same_imag = fread(fileID, [nrows, ncols], 'double');
fclose(fileID);

same_z = complex(same_real, same_imag);
```

Interleave Real and Imaginary Components

An alternative approach is to interleave the real and imaginary components for each value. `fwrite` writes values in column order, so build an array that combines the real and imaginary parts by alternating rows.

First, preallocate the interleaved array.

```
interleaved = zeros(nrows*2, ncols);
```

Alternate real and imaginary data.

```
newrow = 1;
for row = 1:nrows
    interleaved(newrow,:) = z_real(row,:);
    interleaved(newrow + 1,:) = z_imag(row,:);
    newrow = newrow + 2;
end
```

Write the interleaved values to a file named `complex_int.bin`.

```
fileID = fopen('complex_int.bin','w');
fwrite(fileID, interleaved, 'double');
fclose(fileID);
```

Open the file for reading and read the real values from the file. The fourth input to `fread` tells the function to skip the specified number of bytes after reading each value.

```
fileID = fopen('complex_int.bin');
same_real = fread(fileID, [nrows, ncols], 'double', 8);
```

Return to the first imaginary value in the file. Then, read all the imaginary data.

```
fseek(fileID, 8, 'bof');
same_imag = fread(fileID, [nrows, ncols], 'double', 8);
fclose(fileID);

same_z = complex(same_real, same_imag);
```

See Also

`fopen` | `fread` | `fseek` | `fwrite`

More About

- "Moving within a File" on page 4-10

Internet of Things (IoT) Data

Aggregate Data in ThingSpeak Channel

This example shows how to aggregate data to a lower time resolution in a ThingSpeak™ channel to remove irregularity. Irregularity in a data can be caused due to several factors such as event driven sensing, malfunctioning of sensors, or network latencies.

Read Data

ThingSpeak channel 22641 contains tide and weather data measured once a minute at Ockway Bay, Cape Cod. Field 2 of the channel contains air temperature data. Read the air temperature data for the past 3 hours from channel 22641 using the `thingSpeakRead` function.

```
datetimeStop = dateshift(datetime('now'),'start','hour');
datetimeStart = dateshift(datetime('now'),'start','hour') - hours(3);

data = thingSpeakRead(22641,'DateRange',[datetimeStart,datetimeStop],...
    'Fields',2,'outputFormat','timetable');
```

Aggregate the Data

Data is measured once every minute. However, due to network latency associated with the measurement system, the actual timestamps can be greater than or less than a minute apart. Further, for the application of interest, the frequency of data measured every minute is high. Data at an hourly time resolution is sufficient. You can use the `retime` function to aggregate the data for each hour to a single value. You can use the maximum value for each hour to aggregate the data. Preview the first four values of the data with `head`.

```
dataHourly = retime(data,'hourly','max');
head(dataHourly,4)
```

```
ans =

  3×1 timetable

         Timestamps          AirTemperatureC
    ____________________    _______________

    03-Jan-2019 14:00:00          7.5
    03-Jan-2019 15:00:00          6.9
    03-Jan-2019 16:00:00          6.4
```

Send Data to ThingSpeak

Change the channelID and the writeAPIKey to send data to your channel

```
channelID=17504;
writeAPIKey='23ZLGOBBU9TWHG2H';
thingSpeakWrite(channelID,data,'writeKey',writeAPIKey);
```

See Also

`retime` | `thingSpeakRead` | `thingSpeakWrite`

Regularize Irregularly Sampled Data

This example shows how to regularize irregularly sampled data to have a constant time period between measurements. You update timestamps of data read from a ThingSpeak™ channel to remove irregularity, then write the data to a channel. Timestamp variations in measured data introduced due to network latencies or hardware resets can affect data preprocessing and data analytics algorithms. Many algorithms require regularly sampled data to work correctly.

Read Data from the Weather Station Channel

ThingSpeak channel 12397 contains data from the MathWorks® weather station, located in Natick, Massachusetts. The data is collected once every minute. Field 4 of the channel contains air temperature data. To check for irregularly sampled data, read the air temperature data from channel 12397 using the `thingSpeakRead` function.

```
data = thingSpeakRead(12397,'NumMin',60,'Fields',4,'outputFormat','timetable');
```

Check for Irregularly Sampled Data

The data for the last 60 minutes read from channel 12397 is stored in as a timetable. Use `isregular` function to check if the channel data is regularly sampled. If data is irregularly sampled, generate a regularly spaced time vector for the time period of interest. Generate a new time vector using `linspace` with the `startTime`, `stopTime`, and the number of measurements.

```
regularFlag = isregular(data,'Time');

if ~regularFlag
    startTime = data.Timestamps(1);
    stopTime  = data.Timestamps(end);
    newTimeVector = linspace(startTime,stopTime,height(data));
    data.Timestamps = newTimeVector;
end
```

Send Data to ThingSpeak

Send the processed data to a ThingSpeak channel using the `thingSpeakWrite` function.

```
% Change the channelID and the writeAPIKey to send data to your channel.
channelID=17504;
writeAPIKey='23ZLGOBBU9TWHG2H';
thingSpeakWrite(channelID,data,'WriteKey',writeAPIKey);
```

See Also

`linspace` | `thingSpeakRead` | `thingSpeakWrite`

Plot Data Read from ThingSpeak Channel

This example shows how to read data from a public ThingSpeak™ channel and create a simple plot visualization from the results.

Read Data from ThingSpeak Channel

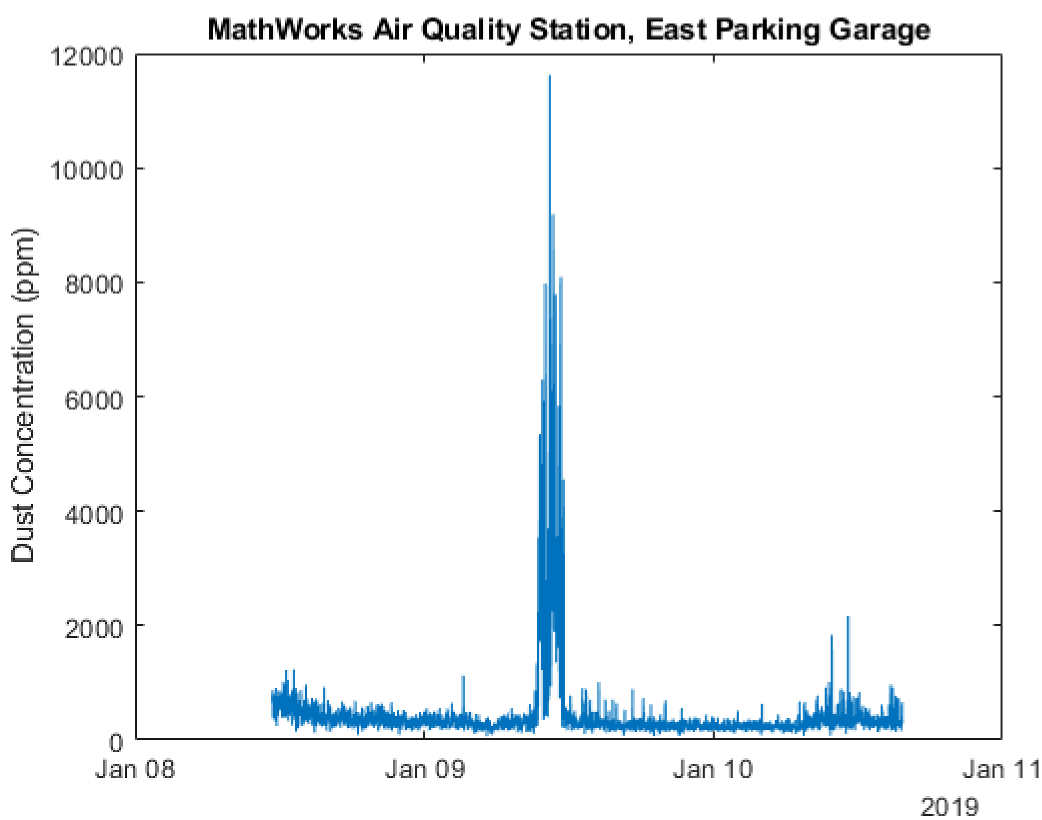
ThingSpeak channel 102698 contains air quality data from a parking garage in Natick Massachusetts. Field 5 is a measure of dust concentration.

```
[dustData,Timestamps]=thingSpeakRead(102698,'Fields',5,'NumPoints',3000);
```

Plot the Dust Concentration Over Time

Use `plot` to visualize the data. Use `ylabel` and `title` to add labels to your plot.

```
plot(Timestamps,dustData);
ylabel('Dust Concentration (ppm)');
title('MathWorks Air Quality Station, East Parking Garage');
```



During business days, you can see spikes in the dust concentration at times when cars arrive or depart.

See Also

`plot` | `thingSpeakRead` | `thingSpeakWrite` | `title` | `ylabel`

Read ThingSpeak Data and Predict Battery Discharge Time with Linear Fit

This example shows how to read battery data from a ThingSpeak™ channel and analyze the data to determine the remaining battery life. Use a linear fit to predict the date that the battery will fail, and then write the remaining time in days to another ThingSpeak Channel. You read data for a 12 V battery connected to a microprocessor reporting its voltage to ThingSpeak every half hour. Then use regression to predict the day and time when the battery will fail.

Read Data from ThingSpeak Channel

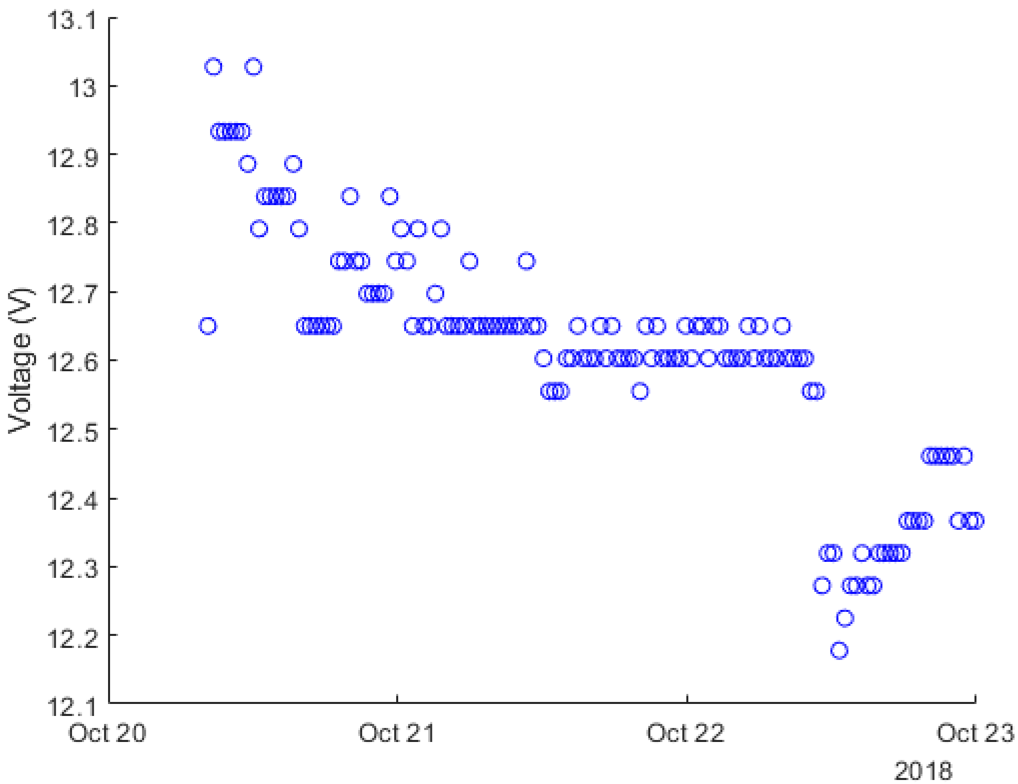
Start by storing channel and date information in variables, and then use `thingSpeakRead` to read the data. Channel 592680 shows the scaled measurement of voltage from a 12 V battery. Use the `DateRange` name-value pair to use a specific selection of data.

```
batteryChannelID = 592680;
startDate = datetime('Oct 20, 2018');
endDate = datetime('Oct 23, 2018');
batteryData = thingSpeakRead(batteryChannelID,'DateRange',[startDate endDate],'Outputformat','Ti
```

Convert the Data for Fitting and Plot

The channel stores raw data from the device. Convert the analog-to-digital converter (ADC) measurement to voltage using the experimentally determined conversion factor 14.6324. Then use `scatter` to generate a plot.

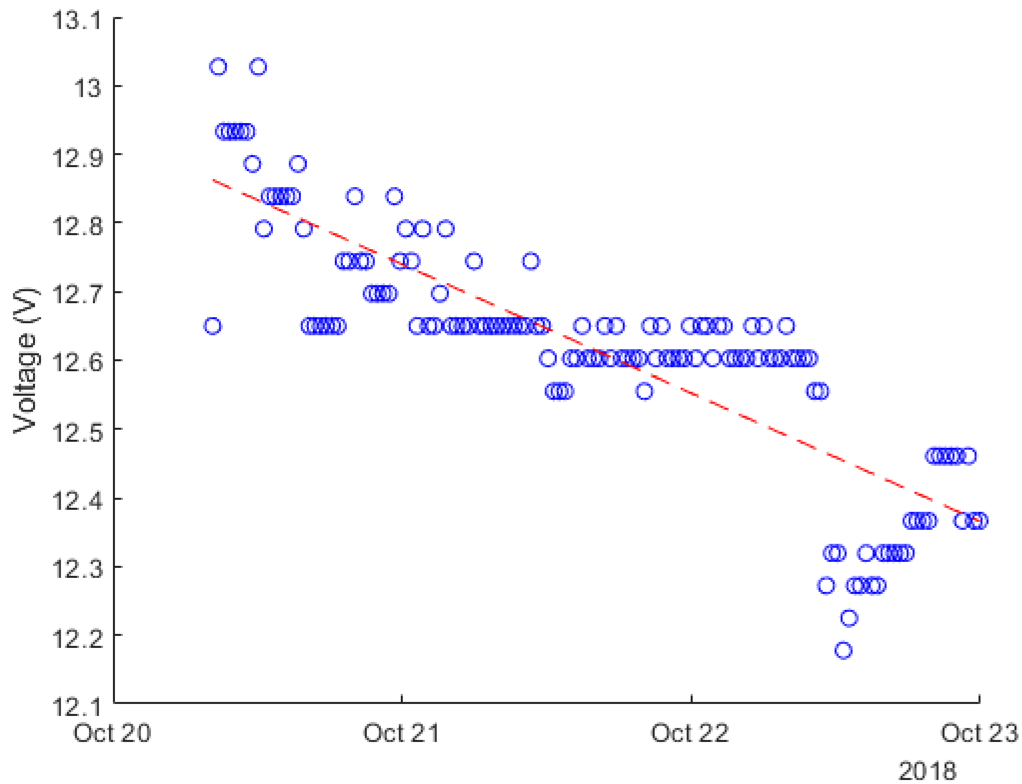
```
myVoltage = 14.6324 * batteryData.Voltage;
scatter(batteryData.Timestamps,myVoltage,'b');
ylabel('Voltage (V)');
hold on
```



Fit the Data

The timetable datetime format is useful for reading and plotting. To fit the data, the datetime needs to be in numeric format. Use `datenum` to convert the timestamps into a number of days, and subtract the starting number to keep the values low. Use `polyfit` to perform linear regression on the data, and `polyval` to evaluate the fit at the existing time values. Add the fit line to the previous plot.

```
battTimes = datenum(batteryData.Timestamps);
battTimes= battTimes-battTimes(1);
myFit=polyfit(battTimes,myVoltage,1);
fitLine=polyval(myFit,battTimes);
plot(batteryData.Timestamps,fitLine,'r--');
```



Predict Discharge Time

The battery should not be discharged below 10.4 V. Find the number of days until the fit line will intersect with this voltage.

```
endDays = (10.4-myFit(2))/myFit(1)
```

```
endDays = 13.1573
```

There are just over 13 days until the battery dies.

Write Prediction to ThingSpeak

The `thingSpeakWrite` function writes the result to a ThingSpeak channel. Return the output from `thingSpeakWrite` to ensure a successful write operation. Change the `writeChannelID` and `writeAPIKey` to write to your own channel.

```
writeChannelID = 17504;
writeAPIKey='23ZLGOBBU9TWHG2H';
result = thingSpeakWrite(writeChannelID,round(endDays,4),'WriteKey',writeAPIKey)
```

```
result = struct with fields:
         Field1: '13.1573'
         Field2: []
         Field3: []
         Field4: []
         Field5: []
         Field6: []
```

```
      Field7: []
      Field8: []
    Latitude: []
   Longitude: []
   ChannelID: 17504
     Created: 03-Jun-2019 15:24:43
 LastEntryID: 50018
    Altitude: []
```

The result shows the successful write operation and reports the data that was written.

See Also

datetime | datnum | polyfit | polyval | scatter | thingSpeakRead | thingSpeakWrite

Images

Importing Images

To import data into the MATLAB workspace from a graphics file, use the imread function. Using this function, you can import data from files in many standard file formats, including the Tagged Image File Format (TIFF), Graphics Interchange Format (GIF), Joint Photographic Experts Group (JPEG), and Portable Network Graphics (PNG) formats. For a complete list of supported formats, see the imread reference page.

This example reads the image data stored in a file in JPEG format into the MATLAB workspace as the array I:

```
I = imread('ngc6543a.jpg');
```

imread represents the image in the workspace as a multidimensional array of class uint8. The dimensions of the array depend on the format of the data. For example, imread uses three dimensions to represent RGB color images:

```
whos I
  Name      Size                           Bytes  Class

  I       650x600x3                      1170000  uint8 array

Grand total is 1170000 elements using 1170000 bytes
```

For more control over reading TIFF files, use the Tiff object—see "Reading Image Data and Metadata from TIFF Files" on page 6-3 for more information.

Getting Information About Image Files

If you have a file in a standard graphics format, use the imfinfo function to get information about its contents. The imfinfo function returns a structure containing information about the file. The fields in the structure vary with the file format, but imfinfo always returns some basic information including the file name, last modification date, file size, and format.

This example returns information about a file in Joint Photographic Experts Group (JPEG) format:

```
info = imfinfo('ngc6543a.jpg')

info =

           Filename: 'matlabroot\toolbox\matlab\demos\ngc6543a.jpg'
        FileModDate: '01-Oct-1996 16:19:44'
           FileSize: 27387
             Format: 'jpg'
      FormatVersion: ''
              Width: 600
             Height: 650
           BitDepth: 24
          ColorType: 'truecolor'
    FormatSignature: ''
    NumberOfSamples: 3
       CodingMethod: 'Huffman'
      CodingProcess: 'Sequential'
            Comment: {'CREATOR: XV Version 3.00b  Rev: 6/15/94  Quality =...'}
```

Reading Image Data and Metadata from TIFF Files

While you can use `imread` to import image data and metadata from TIFF files, the function does have some limitations. For example, a TIFF file can contain multiple images and each images can have multiple subimages. While you can read all the images from a multi-image TIFF file with `imread`, you cannot access the subimages. Using the `Tiff` object, you can read image data, metadata, and subimages from a TIFF file. When you construct a `Tiff` object, it represents your connection with a TIFF file and provides access to many of the routines in the LibTIFF library.

A step-by-step example of using `Tiff` object methods and properties to read subimages from a TIFF file follows. To get the most out of the `Tiff` object, familiarize yourself with the TIFF specification and technical notes. See `LibTIFF - TIFF Library and Utilities`.

Reading Subimages from a TIFF File

A TIFF file can contain one or more image file directories (IFD). Each IFD contains image data and the metadata (tags) associated with the image. Each IFD can contain one or more subIFDs, which also can contain image data and metadata. These subimages are typically reduced-resolution (thumbnail) versions of the image data in the IFD containing the subIFDs.

To read the subimages in an IFD, you must get the location of the subimage from the `SubIFD` tag. The `SubIFD` tag contains an array of byte offsets that point to the subimages. You then can pass the address of the subIFD to the `setSubDirectory` method to make the subIFD the current IFD. Most `Tiff` object methods operate on the current IFD.

1. Open a TIFF file that contains images and subimages using the `Tiff` object constructor. This example uses the TIFF file created in “Creating TIFF File Subdirectories” on page 6-8, which contains one IFD directory with two subIFDs. The `Tiff` constructor opens the TIFF file, and makes the first subIFD in the file the current IFD:

   ```
   t = Tiff('my_subimage_file.tif','r');
   ```
2. Retrieve the locations of subIFDs associated with the current IFD. Use the `getTag` method to get the value of the `SubIFD` tag. This method returns an array of byte offsets that specify the location of subIFDs:

   ```
   offsets = getTag(t,'SubIFD')
   ```
3. Navigate to the first subimage. First, set the currentIFD to the directory containing the first subimage:

   ```
   dirNum = 1;
   setDirectory(t,dirNum);
   ```
4. Then, navigate to the first subIFD using the `setSubDirectory` method. Specify the byte offset of the subIFD as an argument. This call makes the subIFD the current IFD:

   ```
   setSubDirectory(t,offsets(1));
   ```
5. Read the image data from the current IFD (the first subIFD) the same way you read any other IFD in the file:

   ```
   subimage_one = read(t);
   ```
6. View the first subimage:

   ```
   imagesc(subimage_one)
   ```
7. Navigate to the second subimage. First, reset the currentIFD to the directory containing the second subimage:

   ```
   setDirectory(t,dirNum);
   ```

8 Then, navigate to the second subIFD using the `setSubDirectory` method. Specify the byte offset of the second subIFD:

```
setSubDirectory(t,offsets(2));
```

9 Read the image data from the current IFD (the second subIFD) as you would with any other IFD in the file:

```
subimage_two = read(t);
```

10 View the second subimage:

```
imagesc(subimage_two)
```

11 Close the `Tiff` object:

```
close(t);
```

See Also

`Tiff`

External Websites

- "Exporting to Images" on page 6-5

Exporting to Images

To export data from the MATLAB workspace using one of the standard graphics file formats, use the `imwrite` function. Using this function, you can export data in formats such as the Tagged Image File Format (TIFF), Joint Photographic Experts Group (JPEG), and Portable Network Graphics (PNG). For a complete list of supported formats, see the `imwrite` reference page.

The following example writes a multidimensional array of `uint8` data `I` from the MATLAB workspace into a file in TIFF format. The class of the output image written to the file depends on the format specified. For most formats, if the input array is of class `uint8`, `imwrite` outputs the data as 8-bit values. See the `imwrite` reference page for details.

```
whos I
  Name      Size                           Bytes  Class

  I       650x600x3                      1170000  uint8 array

Grand total is 1170000 elements using 1170000 bytes
imwrite(I, 'my_graphics_file.tif','tif');
```

Note `imwrite` supports different syntaxes for several of the standard formats. For example, with TIFF file format, you can specify the type of compression MATLAB uses to store the image. See the `imwrite` reference page for details.

For more control writing data to a TIFF file, use the `Tiff` object—see "Exporting Image Data and Metadata to TIFF Files" on page 6-5 for more information.

Exporting Image Data and Metadata to TIFF Files

While you can use `imwrite` to export image data and metadata (tags) to Tagged Image File Format (TIFF) files, the function does have some limitations. For example, when you want to modify image data or metadata in the file, you must write the all the data to the file. You cannot write only the updated portion. Using the `Tiff` object, you can write portions of the image data and modify or add individual tags to a TIFF file. When you construct a `Tiff` object, it represents your connection with a TIFF file and provides access to many of the routines in the LibTIFF library.

The following sections provide step-by-step examples of using `Tiff` object methods and properties to perform some common tasks with TIFF files. To get the most out of the `Tiff` object, you must be familiar with the TIFF specification and technical notes. View this documentation at `LibTIFF - TIFF Library and Utilities`.

Creating a New TIFF File

1. Create some image data. This example reads image data from a JPEG file included with MATLAB:

   ```
   imgdata = imread('ngc6543a.jpg');
   ```
2. Create a new TIFF file by constructing a `Tiff` object, specifying the name of the new file as an argument. To create a file you must specify either write mode (`'w'`) or append mode (`'a'`):

   ```
   t = Tiff('myfile.tif','w');
   ```

   When you create a new TIFF file, the `Tiff` constructor creates a file containing an image file directory (IFD). A TIFF file uses this IFD to organize all the data and metadata associated with a particular image. A TIFF file can contain multiple IFDs. The `Tiff` object makes the IFD it creates the current IFD. `Tiff` object methods operate on the current IFD. You can navigate among IFDs in a TIFF file and specify which IFD is the current IFD using `Tiff` object methods.

3. Set required TIFF tags using the `setTag` method of the `Tiff` object. These required tags specify information about the image, such as its length and width. To break the image data into strips, specify a value for the `RowsPerStrip` tag. To break the image data into tiles, specify values for the `TileWidth` and `TileLength` tags. The example creates a structure that contains tag names and values and passes that to `setTag`. You also can set each tag individually.

```
tagstruct.ImageLength = size(imgdata,1);
tagstruct.ImageWidth = size(imgdata,2);
tagstruct.Photometric = Tiff.Photometric.RGB;
tagstruct.BitsPerSample = 8;
tagstruct.SamplesPerPixel = 3;
tagstruct.RowsPerStrip = 16;
tagstruct.PlanarConfiguration = Tiff.PlanarConfiguration.Chunky;
tagstruct.Software = 'MATLAB';
tagstruct % display tagstruct
setTag(t,tagstruct)
```

   For information about supported TIFF tags and how to set their values, see “Setting Tag Values” on page 6-9. For example, the `Tiff` object supports properties that you can use to set the values of certain properties. This example uses the `Tiff` object `PlanarConfiguration` property to specify the correct value for the chunky configuration: `Tiff.PlanarConfiguration.Chunky`.

4. Write the image data and metadata to the current directory using the `write` method of the `Tiff` object.

```
write(t,imgdata);
```

   If you wanted to put multiple images into your file, call the `writeDirectory` method right after performing this write operation. The `writeDirectory` method sets up a new image file directory in the file and makes this new directory the current directory.

5. Close your connection to the file by closing the `Tiff` object:

```
close(t);
```

6. Test that you created a valid TIFF file by using the `imread` function to read the file, and then display the image:

```
imagesc(imread('myfile.tif'));
```

Writing a Strip or Tile of Image Data

Note You can only modify a strip or a tile of image data if the data is not compressed.

---

1. Open an existing TIFF file for modification by creating a `Tiff` object. This example uses the file created in “Creating a New TIFF File” on page 6-5. The `Tiff` constructor returns a handle to a `Tiff` object.

```
t = Tiff('myfile.tif','r+');
```

2. Generate some data to write to a strip in the image. This example creates a three-dimensional array of zeros that is the size of a strip. The code uses the number of rows in a strip, the width of the image, and the number of samples per pixel as dimensions. The array is an array of `uint8` values.

```
width = getTag(t,'ImageWidth');
height = getTag(t,'RowsPerStrip');
numSamples = getTag(t,'SamplesPerPixel');
stripData = zeros(height,width,numSamples,'uint8');
```

If the image data had a tiled layout, you would use the `TileWidth` and `TileLength` tags to specify the dimensions.

3. Write the data to a strip in the file using the `writeEncodedStrip` method. Specify the index number that identifies the strip you want to modify. The example picks strip 18 because it is easier to see the change in the image.

```
writeEncodedStrip(t,18,stripData);
```

If the image had a tiled layout, you would use the `writeEncodedTile` method to modify the tile.

4. Close your connection to the file by closing the `Tiff` object.

```
close(t);
```

5. Test that you modified a strip of the image in the TIFF file by using the `imread` function to read the file, and then display the image.

```
modified_imgdata = imread('myfile.tif');
imagesc(modified_imgdata)
```

Note the black strip across the middle of the image.

Modifying TIFF File Metadata (Tags)

1. Open an existing TIFF file for modification using the `Tiff` object. This example uses the file created in “Creating a New TIFF File” on page 6-5. The `Tiff` constructor returns a handle to a `Tiff` object.

```
t = Tiff('myfile.tif','r+');
```

2. Verify that the file does not contain the `Artist` tag, using the `getTag` method. This code should issue an error message saying that it was unable to retrieve the tag.

```
artist_value = getTag(t,'Artist');
```

3. Add the `Artist` tag using the `setTag` method.

```
setTag(t,'Artist','Pablo Picasso');
```

4. Write the new tag data to the TIFF file using the `rewriteDirectory` method. Use the `rewriteDirectory` method when modifying existing metadata in a file or adding new metadata to a file.

```
rewriteDirectory(t);
```

5. Close your connection to the file by closing the `Tiff` object.

```
close(t);
```

6. Test your work by reopening the TIFF file and getting the value of the `Artist` tag, using the `getTag` method.

```
t = Tiff('myfile.tif', 'r');

getTag(t,'Artist')

ans =

Pablo Picasso
```

```
close(t);
```

Creating TIFF File Subdirectories

1 Create some image data. This example reads image data from a JPEG file included with MATLAB. The example then creates two reduced-resolution (thumbnail) versions of the image data.

```
imgdata = imread('ngc6543a.jpg');
%
% Reduce number of pixels by a half.
img_half = imgdata(1:2:end,1:2:end,:);
%
% Reduce number of pixels by a third.
img_third = imgdata(1:3:end,1:3:end,:);
```

2 Create a new TIFF file by constructing a `Tiff` object and specifying the name of the new file as an argument. To create a file you must specify either write mode (`'w'`) or append mode (`'a'`). The `Tiff` constructor returns a handle to a `Tiff` object.

```
t = Tiff('my_subimage_file.tif','w');
```

3 Set required TIFF tags using the `setTag` method of the `Tiff` object. These required tags specify information about the image, such as its length and width. To break the image data into strips, specify a value for the `RowsPerStrip` tag. To break the image data into tiles, use the `TileWidth` and `TileLength` tags. The example creates a structure that contains tag names and values and passes that to `setTag`. You can also set each tag individually.

To create subdirectories, you must set the `SubIFD` tag, specifying the number of subdirectories you want to create. Note that the number you specify isn't the value of the `SubIFD` tag. The number tells the `Tiff` software to create a `SubIFD` that points to two subdirectories. The actual value of the `SubIFD` tag will be the byte offsets of the two subdirectories.

```
tagstruct.ImageLength = size(imgdata,1);
tagstruct.ImageWidth = size(imgdata,2);
tagstruct.Photometric = Tiff.Photometric.RGB;
tagstruct.BitsPerSample = 8;
tagstruct.SamplesPerPixel = 3;
tagstruct.RowsPerStrip = 16;
tagstruct.PlanarConfiguration = Tiff.PlanarConfiguration.Chunky;
tagstruct.Software = 'MATLAB';
tagstruct.SubIFD = 2 ;  % required to create subdirectories
tagstruct  % display tagstruct
setTag(t,tagstruct)
```

For information about supported TIFF tags and how to set their values, see “Setting Tag Values” on page 6-9. For example, the `Tiff` object supports properties that you can use to set the values of certain properties. This example uses the `Tiff` object `PlanarConfiguration` property to specify the correct value for the chunky configuration: `Tiff.PlanarConfiguration.Chunky`.

4 Write the image data and metadata to the current directory using the `write` method of the `Tiff` object.

```
write(t,imgdata);
```

5 Set up the first subdirectory by calling the `writeDirectory` method. The `writeDirectory` method sets up the subdirectory and make the new directory the current directory. Because you specified that you wanted to create two subdirectories, `writeDirectory` sets up a subdirectory.

```
writeDirectory(t);
```

6 Set required tags, just as you did for the regular directory. According to the LibTIFF API, a subdirectory cannot contain a `SubIFD` tag.

```
tagstruct2.ImageLength = size(img_half,1);
tagstruct2.ImageWidth = size(img_half,2);
tagstruct2.Photometric = Tiff.Photometric.RGB;
tagstruct2.BitsPerSample = 8;
tagstruct2.SamplesPerPixel = 3;
tagstruct2.RowsPerStrip = 16;
tagstruct2.PlanarConfiguration = Tiff.PlanarConfiguration.Chunky;
tagstruct2.Software = 'MATLAB';
tagstruct2  % display tagstruct2
setTag(t,tagstruct2)
```

7 Write the image data and metadata to the subdirectory using the `write` method of the `Tiff` object.

```
write(t,img_half);
```

8 Set up the second subdirectory by calling the `writeDirectory` method. The `writeDirectory` method sets up the subdirectory and makes it the current directory.

```
writeDirectory(t);
```

9 Set required tags, just as you would for any directory. According to the LibTIFF API, a subdirectory cannot contain a `SubIFD` tag.

```
tagstruct3.ImageLength = size(img_third,1);
tagstruct3.ImageWidth = size(img_third,2);
tagstruct3.Photometric = Tiff.Photometric.RGB;
tagstruct3.BitsPerSample = 8;
tagstruct3.SamplesPerPixel = 3;
tagstruct3.RowsPerStrip = 16;
tagstruct3.PlanarConfiguration = Tiff.PlanarConfiguration.Chunky;
tagstruct3.Software = 'MATLAB';
tagstruct3  % display tagstruct3
setTag(t,tagstruct3)
```

10 Write the image data and metadata to the subdirectory using the `write` method of the `Tiff` object:

```
write(t,img_third);
```

11 Close your connection to the file by closing the `Tiff` object:

```
close(t);
```

Setting Tag Values

The following table lists all the TIFF tags that the `Tiff` object supports and includes information about their MATLAB class and size. For certain tags, the table also indicates the set of values that the `Tiff` object supports, which is a subset of all the possible values defined by the TIFF specification. You can use the `Tiff` properties structure to specify the supported values for these tags. For example, use `Tiff.Compression.JPEG` to specify JPEG compression. See the Tiff reference page for a full list of properties.

Table 1: Supported TIFF Tags

TIFF Tag	Class	Size	Supported Values	Notes
Artist	char	1xN		
BitsPerSample	double	1x1	1,8,16,32,64	See Table 2 on page 6-14
ColorMap	double	256x3	Values should be normalized between 0–1. Stored internally as uint16 values.	Photometric must be Palette
Compression	double	1x1	None: 1 CCITTRLE: 2 CCITTFax3: 3 CCITTFax4: 4 LZW: 5 JPEG: 7 CCITTRLEW: 32771 PackBits: 32773 Deflate: 32946 AdobeDeflate: 8	See Table 3 on page 6-15.
Copyright	char	1xN		
DateTime	char	1x19	Return value is padded to 19 chars if required.	
DocumentName	char	1xN		
DotRange	double	1x2		Photometric must be Separated
ExtraSamples	double	1xN	Unspecified: 0 AssociatedAlpha: 1 UnassociatedAlpha: 2	See Table 4 on page 6-15.
FillOrder	double	1x1		
GeoAsciiParamsTag	char	1xN		
GeoDoubleParamsTag	double	1xN		
GeoKeyDirectoryTag	double	Nx4		
Group3Options	double	1x1		Compression must be CCITTFax3
Group4Options	double	1x1		Compression must be CCITTFax4
HalfToneHints	double	1x2		
HostComputer	char	1xn		
ICCProfile	uint8	1xn		
ImageDescription	char	1xn		
ImageLength	double	1x1		

TIFF Tag	Class	Size	Supported Values	Notes
ImageWidth	double	1x1		
InkNames	char cell array	1x NumInks		Photometric must be Separated
InkSet	double	1x1	CMYK: 1 MultiInk: 2	Photometric must be Separated
JPEGQuality	double	1x1	A value between 1 and 100	
Make	char	1xn		
MaxSampleValue	double	1x1	0–65,535	
MinSampleValue	double	1x1	0–65,535	
Model	char	1xN		
ModelPixelScaleTag	double	1x3		
ModelTiepointTag	double	Nx6		
ModelTransformationMatrixTag	double	1x16		
NumberOfInks	double	1x1		Must be equal to SamplesPerPixel
Orientation	double	1x1	TopLeft: 1 TopRight: 2 BottomRight: 3 BottomLeft: 4 LeftTop: 5 RightTop: 6 RightBottom: 7 LeftBottom: 8	
PageName	char	1xN		
PageNumber	double	1x2		
Photometric	double	1x1	MinIsWhite: 0 MinIsBlack: 1 RGB: 2 Palette: 3 Mask: 4 Separated: 5 YCbCr: 6 CIELab: 8 ICCLab: 9 ITULab: 10	See Table 2 on page 6-14.
Photoshop	uint8	1xN		
PlanarConfiguration	double	1x1	Chunky: 1 Separate: 2	
PrimaryChromaticities	double	1x6		
ReferenceBlackWhite	double	1x6		

TIFF Tag	Class	Size	Supported Values	Notes
ResolutionUnit	double	1x1		
RICHTIFFIPTC	uint8	1xN		
RowsPerStrip	double	1x1		
RPCCoefficientTag	double	1x92	92-element row vector	See Table 6 on page 6-16
SampleFormat	double	1x1	Uint: 1 Int: 2 IEEEFP: 3	See Table 2 on page 6-13
SamplesPerPixel	double	1x1		
SMaxSampleValue	double	1x1	Range of MATLAB data type specified for Image data	
SMinSampleValue	double	1x1	Range of MATLAB data type specified for Image data	
Software	char	1xN		
StripByteCounts	double	1xN		Read-only
StripOffsets	double	1xN		Read-only
SubFileType	double	1x1	Default : 0 ReducedImage: 1 Page: 2 Mask: 4	
SubIFD	double	1x1		
TargetPrinter	char	1xN		
Thresholding	double	1x1	BiLevel: 1 HalfTone: 2 ErrorDiffuse: 3	Photometric can be either: MinIsWhite MinIsBlack
TileByteCounts	double	1xN		Read-only
TileLength	double	1x1	Must be a multiple of 16	
TileOffsets	double	1xN		Read-only
TileWidth	double	1x1	Must be a multiple of 16	
TransferFunction	double	See note[1]	Each value should be within 0–2^16-1	SamplePerPixel can be either 1 or 3
WhitePoint	double	1x2		Photometric can be: RGB Palette YCbCr CIELab ICCLab ITULab

TIFF Tag	Class	Size	Supported Values	Notes
XMP	char	1xn		N>5
XPostion	double	1x1		
XResolution	double	1x1		
YCbCrCoefficents	double	1x3		Photometric must be YCbCr
YCbCrPositioning	double	1x1	Centered: 1 Cosited: 2	Photometric must be YCbCr
YCbCrSubSampling	double	1x2		Photometric must be YCbCr
YPosition	double	1x1		
YResolution	double	1x1		
ZipQuality	double	1x1	Value between 1 and 9	

[1]Size is 1x2^BitsPerSample or3x2^BitsPerSample.

Table 2: Valid SampleFormat Values for BitsPerSample Settings

BitsPerSample	SampleFormat	MATLAB Data Type
1	Uint	logical
8	Uint, Int	uint8, int8
16	Uint, Int	uint16, int16
32	Uint, Int, IEEEFP	uint32, int32, single
64	IEEEFP	double

Table 3: Valid SampleFormat Values for BitsPerSample and Photometric Combinations

	BitsPerSample Values				
Photometric Values	**1**	**8**	**16**	**32**	**64**
MinIsWhite	Uint	Uint/Int	Uint Int	Uint Int IEEEFP	IEEEFP
MinIsBlack	Uint	Uint/Int	Uint Int	Uint Int IEEEFP	IEEEFP
RGB		Uint	Uint	Uint IEEEFP	IEEEFP
Pallette		Uint	Uint		
Mask	Uint				
Separated		Uint	Uint	Uint IEEEFP	IEEEFP
YCbCr		Uint	Uint	Uint IEEEFP	IEEEFP
CIELab		Uint	Uint		
ICCLab		Uint	Uint		
ITULab		Uint	Uint		

Table 4: Valid SampleFormat Values for BitsPerSample and Compression Combinations

	BitsPerSample Values				
Compression Values	**1**	**8**	**16**	**32**	**64**
None	Uint	Uint Int	Uint Int	Uint Int IEEEFP	IEEEFP
CCITTRLE	Uint				
CCITTFax3	Uint				
CCITTFax4	Uint				
LZW	Uint	Uint Int	Uint Int	Uint Int IEEEFP	IEEEFP
JPEG		Uint Int			
CCITTRLEW	Uint				
PackBits	Uint	Uint Int	Uint Int	Uint Int IEEEFP	IEEEFP
Deflate	Uint	Uint Int	Uint Int	Uint Int IEEEFP	IEEEFP
AdobeDeflate	Uint	Uint Int	Uint Int	Uint Int IEEEFP	IEEEFP

Table 5: Valid SamplesPerPixel Values for Photometric Settings

Photometric Values	SamplesPerPixel[1]
MinIsWhite	1+
MinIsBlack	1+
RGB	3+
Pallette	1
Mask	1
Separated	1+
YCbCr	3
CIELab	3+
ICCLab	3+
ITULab	3+

Table 6: List of RPCCoefficientTag Value Descriptions

Index Value in 92-Element Vector	Value Descriptions[1]	Units
1	Root mean square bias error	meters per horizontal axis
2	Root mean square random error	meters per horizontal axis
3	Line offset	pixels
4	Sample offset	pixels
5	Geodetic latitude offset	degrees
6	Geodetic longitude offset	degrees
7	Geodetic height offset	meters
8	Line scale factor	pixels
9	Sample scale factor	pixels
10	Geodetic latitude scale	degrees
11	Geodetic longitude scale	degrees
12	Geodetic height scale factor	meters
13 through 32	Numerator coefficients of *r(n)*, a rational polynomial equation [2]	
33 through 52	Denominator coefficients of the rational polynomial equation *r(n)*	
53 through 72	Numerator coefficients of *c(n)*, a rational polynomial equation [2]	
73 through 92	Denominator coefficients of the rational polynomial equation *c(n)*	

[1]To specify the values in this vector using the `RPCCoefficientTag` object, see `RPCCoefficientTag` in the Mapping Toolbox™.

[2]Equations *r(n)* and *c(n)* represent the normalized row and column values of a generic rigorous projection model.

See Also

`Tiff`

External Websites

- "Importing Images" on page 6-2

Scientific Data

Import CDF Files Using Low-Level Functions

This example shows how to use low-level functions to read data from a CDF file. The MATLAB® low-level CDF functions correspond to routines in the CDF C API library. To use the MATLAB CDF low-level functions effectively, you must be familiar with the CDF C interface.

Open CDF File

Open the sample CDF File, `example.cdf`.

```
cdfid = cdflib.open('example.cdf');
```

Get Information About File Contents

Use `cdflib.inquire` to get information about the number of variables in the file, the number of global attributes, and the number of attributes with variable scope.

```
info = cdflib.inquire(cdfid)
```

```
info = struct with fields:
     encoding: 'IBMPC_ENCODING'
     majority: 'ROW_MAJOR'
       maxRec: 23
      numVars: 6
    numvAttrs: 1
    numgAttrs: 3
```

Get Information About Variables

Use `cdflib.inqurieVar` to get information about the individual variables in the file. Variable ID numbers start at zero.

```
info  = cdflib.inquireVar(cdfid,0)
```

```
info = struct with fields:
           name: 'Time'
       datatype: 'cdf_epoch'
    numElements: 1
           dims: []
    recVariance: 1
    dimVariance: []
```

```
info  = cdflib.inquireVar(cdfid,1)
```

```
info = struct with fields:
           name: 'Longitude'
       datatype: 'cdf_int1'
    numElements: 1
           dims: [2 2]
    recVariance: 0
    dimVariance: [1 0]
```

Read Variable Data Into Workspace

Read the data in a variable into the MATLAB workspace. The first variable contains CDF Epoch time values. The low-level interface returns these as `double` values.

```
data_time = cdflib.getVarRecordData(cdfid,0,0)
```

```
data_time = 6.3146e+13
```

Convert the time value to a date vector.

```
timeVec = cdflib.epochBreakdown(data_time)
```

```
timeVec = 7×1

        2001
           1
           1
           0
           0
           0
           0
```

Read Global Attribute From File

Determine which attributes in the CDF file are global.

```
info = cdflib.inquireAttr(cdfid,0)
```

```
info = struct with fields:
         name: 'SampleAttribute'
        scope: 'GLOBAL_SCOPE'
    maxgEntry: 4
     maxEntry: -1
```

Read the value of the attribute. You must use the `cdflib.getAttrgEntry` function for global attributes.

```
value = cdflib.getAttrgEntry(cdfid,0,0)
```

```
value = 
'This is a sample entry.'
```

Close CDF File

Use `cdflib.close` to close the CDF file.

```
cdflib.close(cdfid);
```

See Also

`cdflib` | `cdfread`

External Websites

- CDF website

Represent CDF Time Values

This example shows how to extract date information from a CDF epoch object. CDF represents time differently than MATLAB®. CDF represents date and time as the number of milliseconds since 1-Jan-0000. This is called an epoch in CDF terminology. To represent CDF dates, MATLAB uses an object called a CDF epoch object. MATLAB also can represent a date and time as a datetime value or as a serial date number, which is the number of days since 0-Jan-0000. To access the time information in a CDF object, convert to one of these other representations.

Read the sample CDF file, `example.cdf`.

```
data = cdfread('example.cdf');
whos
```

```
  Name       Size            Bytes  Class    Attributes

  data      24x6             23904  cell
```

`cdfread` returns a cell array.

Extract the date information from the first CDF epoch object returned in the cell array, `data`, using the `todatenum` function.

```
m_datenum = todatenum(data{1})
```

```
m_datenum = 730852
```

Convert the MATLAB serial date number to a datetime value.

```
m_datetime = datetime(m_datenum,'ConvertFrom','datenum')
```

```
m_datetime = datetime
   01-Jan-2001
```

See Also

`cdfread` | `datetime` | `todatenum`

Import CDF Files Using High-Level Functions

This example shows how to use high-level MATLAB® functions to import the sample CDF file, `example.cdf`. High-level functions provide a simpler interface to accessing CDF files.

Get Information About Contents of CDF File

Get information about the contents of a CDF file using the `cdfinfo` function. Because `cdfinfo` creates temporary files, ensure that your current folder is writable before using the function.

```
info = cdfinfo('example.cdf')
```

```
info = struct with fields:
              Filename: 'example.cdf'
           FileModDate: '10-May-2010 21:35:01'
              FileSize: 1310
                Format: 'CDF'
         FormatVersion: '2.7.0'
          FileSettings: [1x1 struct]
              Subfiles: {}
             Variables: {6x6 cell}
      GlobalAttributes: [1x1 struct]
    VariableAttributes: [1x1 struct]
```

`cdfinfo` returns a structure containing general information about the file and detailed information about the variables and attributes in the file. In this example, the `Variables` field indicates the number of variables in the file.

View the contents of the `Variables` field.

```
vars = info.Variables
```

```
vars=6×6 cell array
  Columns 1 through 5

    {'Time'            }    {1x2 double}    {[24]}    {'epoch' }    {'T/'    }
    {'Longitude'       }    {1x2 double}    {[ 1]}    {'int8'  }    {'F/FT'  }
    {'Latitude'        }    {1x2 double}    {[ 1]}    {'int8'  }    {'F/TF'  }
    {'Data'            }    {1x3 double}    {[ 1]}    {'double'}    {'T/TTT' }
    {'multidimensional'}    {1x4 double}    {[ 1]}    {'uint8' }    {'T/TTTT'}
    {'Temperature'     }    {1x2 double}    {[10]}    {'int16' }    {'T/TT'  }

  Column 6

    {'Full'}
    {'Full'}
    {'Full'}
    {'Full'}
    {'Full'}
    {'Full'}
```

The first variable, `Time`, consists of 24 records containing CDF epoch data. The next two variables, `Longitude` and `Latitude`, each have only one associated record containing `int8` data.

Read All Data from CDF File

Use the `cdfread` function to read all of the data in the CDF file.

```
data = cdfread('example.cdf');
whos data
```

```
  Name        Size              Bytes  Class     Attributes

  data       24x6               23904  cell
```

`cdfread` returns the data in a cell array. The columns of data correspond to the variables. The rows correspond to the records associated with a variable.

Read Data from Specific Variables

Read only the `Longitude` and `Latitude` variables from the CDF file. To read the data associated with particular variables, use the `'Variable'` parameter. Specify the names of the variables in a cell array of character vectors. Variable names are case sensitive.

```
var_long_lat = cdfread('example.cdf','Variable',{'Longitude','Latitude'});
whos var_long_lat
```

```
  Name                Size              Bytes  Class     Attributes

  var_long_lat       1x2                  216  cell
```

Combine Records to Speed Up Read Operations

By default, `cdfread` creates a cell array with a separate element for every variable and every record in each variable, padding the records dimension to create a rectangular cell array. When working with large data sets, you can speed up read operations by specifying the `'CombineRecords'` parameter to reduce the number of elements in the cell array that `cdfread` returns. When you set the `'CombineRecords'` parameter to `true`, the `cdfread` function creates a separate element for each variable but saves time by putting all the records associated with a variable in a single cell array element.

```
data_combined = cdfread('example.cdf','CombineRecords',true);
```

Compare the sizes of the cell arrays returned by `cdfread`.

```
whos data*
```

```
  Name                 Size              Bytes  Class     Attributes

  data                24x6               23904  cell
  data_combined        1x6                8080  cell
```

Reading all the data from the example file without the `CombineRecords` parameter returns a 24-by-6 cell array, where the columns represent variables and the rows represent the records for each variable. Reading the data from the same file with `'CombineRecords'` set to `true` returns a 1-by-6 cell array.

When combining records, the dimensions of the data in the cell change. In this example, the `Time` variable has 24 records, each of which is a scalar value. In the `data_combined` cell array, the combined element contains a 24-by-1 vector of values.

Read CDF Epoch Values as Serial Date Numbers

By default, `cdfread` creates a MATLAB `cdfepoch` object for each CDF epoch value in the file. Speed up read operations by setting the 'ConvertEpochToDatenum' name-value pair argument to `true`, to return CDF epoch values as MATLAB serial date numbers.

```
data_datenums = cdfread('example.cdf','ConvertEpochToDatenum',true);
whos data*
```

```
  Name                  Size            Bytes  Class    Attributes

  data                 24x6             23904  cell               
  data_combined         1x6              8080  cell               
  data_datenums        24x6             19872  cell               
```

See Also

`cdfinfo` | `cdfread`

External Websites

- CDF website

Export to CDF Files

This example shows how to export data to a CDF file using MATLAB® CDF low-level functions. The MATLAB functions correspond to routines in the CDF C API library.

To use the MATLAB CDF low-level functions effectively, you must be familiar with the CDF C interface. Also, CDF files do not support non-ASCII encoded inputs. Therefore, variable names, attributes names, variable values, and attribute values must have 7-bit ASCII encoding.

Create New CDF File

Create a new CDF file named `my_file.cdf` using `cdflib.create`. This function corresponds to the CDF library C API routine, `CDFcreateCDF`.

```
cdfid = cdflib.create('my_file.cdf');
```

`cdflib.create` returns a file identifier, `cdfid`.

Create Variables in CDF File

Create variables named `Time` and `Latitude` using `cdflib.createVar`. This function corresponds to the CDF library C API routine, `CDFcreatezVar`.

```
time_id = cdflib.createVar(cdfid,'Time','cdf_int4',1,[],true,[]);
lat_id = cdflib.createVar(cdfid,'Latitude','cdf_int2',1,181,true,true);
```

`cdflib.createVar` returns a numeric identifier for each variable.

Create a variable named `Image`.

```
dimSizes = [20 10];
image_id = cdflib.createVar(cdfid,'Image','cdf_int4',1,...
    dimSizes,true,[true true]);
```

Write to Variables

Write data to the first and second records of the `Time` variable. Record numbers are zero-based. The `cdflib.putVarRecordData` function corresponds to the CDF library C API routine, `CDFputzVarRecordData`.

```
cdflib.putVarRecordData(cdfid,time_id,0,int32(23));
cdflib.putVarRecordData(cdfid,time_id,1,int32(24));
```

Write data to the `Latitude` variable.

```
data = int16([-90:90]);
recspec = [0 1 1];
dimspec = { 0 181 1 };
cdflib.hyperPutVarData(cdfid,lat_id,recspec,dimspec,data);
```

Write data to the `Image` variable.

```
recspec = [0 3 1];
dimspec = { [0 0], [20 10], [1 1] };
data = reshape(int32([0:599]), [20 10 3]);
cdflib.hyperPutVarData(cdfid,image_id,recspec,dimspec,data);
```

Write to Global Attribute

Create a global attribute named TITLE using cdflib.createAttr. This function corresponds to the CDF library C API routine, CDFcreateAttr.

```
titleAttrNum = cdflib.createAttr(cdfid,'TITLE','global_scope');
```

cdflib.createAttr returns a numeric identifier for the attribute. Attribute numbers are zero-based.

Write values to entries in the global attribute.

```
cdflib.putAttrEntry(cdfid,titleAttrNum,0,'CDF_CHAR','cdf Title');
cdflib.putAttrEntry(cdfid,titleAttrNum,1,'CDF_CHAR','Author');
```

Write to Attributes Associated with Variables

Create attributes associated with variables in the CDF file.

```
fieldAttrNum = cdflib.createAttr(cdfid,'FIELDNAM','variable_scope');
unitsAttrNum = cdflib.createAttr(cdfid,'UNITS','variable_scope');
```

Write to attributes of the Time variable.

```
cdflib.putAttrEntry(cdfid,fieldAttrNum,time_id,...
    'CDF_CHAR','Time of observation');
cdflib.putAttrEntry(cdfid,unitsAttrNum,time_id,...
    'CDF_CHAR','Hours');
```

Get Information About CDF File

Get information about the file using cdflib.inquire. This function corresponds to the CDF library C API routines, CDFinquireCDF and CDFgetNumgAttributes.

```
info = cdflib.inquire(cdfid)
```

```
info = struct with fields:
     encoding: 'IBMPC_ENCODING'
     majority: 'ROW_MAJOR'
       maxRec: 2
      numVars: 3
    numvAttrs: 2
    numgAttrs: 1
```

cdflib.inquire returns a structure array that includes information about the data encoding and the number of variables and attributes in the file.

Close CDF File

Close the CDF File using cdflib.close. This function corresponds to the CDF library C API routine, CDFcloseCDF. You must close a CDF to guarantee that all modifications you made since opening the CDF are written to the file.

```
cdflib.close(cdfid);
```

See Also

cdflib

External Websites

- CDF website

Map NetCDF API Syntax to MATLAB Syntax

MATLAB netcdf package of low-level functions and its correspondence with the NetCDF C library.

MATLAB provides access to the routines in the NetCDF C library through a set of low-level functions that are grouped into a package called netcdf. Use the functions in this package to read and write data to and from NetCDF files. To use the MATLAB NetCDF functions effectively, you should be familiar with the NetCDF C interface.

Usually, the MATLAB functions in the netcdf package correspond directly to routines in the NetCDF C library. For example, the MATLAB function netcdf.open corresponds to the NetCDF library routine nc_open. In some cases, one MATLAB function corresponds to a group of NetCDF library functions. For example, instead of creating MATLAB versions of every NetCDF library nc_put_att_*type* function, where *type* represents a data type, MATLAB uses one function, netcdf.putAtt, to handle all supported data types.

To call one of the functions in the netcdf package, you must prefix the function name with the package name. The syntax of the MATLAB functions is similar to the NetCDF library routines. However, the NetCDF C library routines use input parameters to return data, while their MATLAB counterparts use one or more return values. For example, this is the function signature of the nc_open routine in the NetCDF library:

```
int nc_open (const char *path, int omode, int *ncidp); /* C syntax */
```

The NetCDF file identifier is returned in the ncidp argument.

This is the signature of the corresponding MATLAB function, netcdf.open:

```
ncid = netcdf.open(filename, mode)
```

Like its NetCDF C library counterpart, the MATLAB NetCDF function accepts a file name and a constant that specifies the access mode. However, that the MATLAB netcdf.open function returns the file identifier, ncid, as a return value.

The MATLAB NetCDF functions automatically choose the MATLAB class that best matches the NetCDF data type. This table shows the default mapping.

NetCDF Data Type	MATLAB Class
'NC_BYTE'	int8 or uint8[a]
'NC_CHAR'	char
'NC_SHORT'	int16
'NC_INT'	int32
'NC_FLOAT'	single
'NC_DOUBLE'	double

a. NetCDF interprets byte data as either signed or unsigned.

You can override the default and specify the class of the return data by using an optional argument to the netcdf.getVar function.

See Also

More About

- “Import NetCDF Files and OPeNDAP Data” on page 7-13
- “Export to NetCDF Files” on page 7-20

External Websites

- NetCDF website

Import NetCDF Files and OPeNDAP Data

Read data from a NetCDF file using the high-level functions, and then read the file by using the `netcdf` package low-level functions.

In this section...

MATLAB NetCDF Capabilities

Network Common Data Form (NetCDF) is a set of software libraries and machine-independent data formats that support the creation, access, and sharing of array-oriented scientific data. NetCDF is used by a wide range of engineering and scientific fields that want a standard way to store data so that it can be shared.

MATLAB high-level functions simplify the process of importing data from a NetCDF file or an OPeNDAP NetCDF data source. MATLAB low-level functions enable more control over the importing process, by providing access to the routines in the NetCDF C library. To use the low-level functions effectively, you should be familiar with the NetCDF C Interface. The NetCDF documentation is available at the Unidata website.

Note For information about importing Common Data Format (CDF) files, which have a separate, incompatible format, see "Import CDF Files Using Low-Level Functions" on page 7-2.

Read from NetCDF File Using High-Level Functions

This example shows how to display and read the contents of a NetCDF file, using high-level functions.

Display the contents of the sample NetCDF file, `example.nc`.

```
ncdisp('example.nc')
```

```
Source:
           \\matlabroot\toolbox\matlab\demos\example.nc
Format:
           netcdf4
Global Attributes:
           creation_date = '29-Mar-2010'
Dimensions:
           x = 50
           y = 50
           z = 5
Variables:
    avagadros_number
           Size:       1x1
           Dimensions:
           Datatype:   double
           Attributes:
```

```
                       description = 'this variable has no dimensions'
    temperature
           Size:       50x1
           Dimensions: x
           Datatype:   int16
           Attributes:
                       scale_factor = 1.8
                       add_offset   = 32
                       units        = 'degrees_fahrenheight'
    peaks
           Size:       50x50
           Dimensions: x,y
           Datatype:   int16
           Attributes:
                       description = 'z = peaks(50);'
Groups:
    /grid1/
        Attributes:
                   description = 'This is a group attribute.'
        Dimensions:
                   x    = 360
                   y    = 180
                   time = 0     (UNLIMITED)
        Variables:
            temp
                   Size:       []
                   Dimensions: x,y,time
                   Datatype:   int16

    /grid2/
        Attributes:
                   description = 'This is another group attribute.'
        Dimensions:
                   x    = 360
                   y    = 180
                   time = 0     (UNLIMITED)
        Variables:
            temp
                   Size:       []
                   Dimensions: x,y,time
                   Datatype:   int16
```

`ncdisp` displays all the groups, dimensions, and variable definitions in the file. Unlimited dimensions are identified with the label, `UNLIMITED`.

Read data from the `peaks` variable.

```
peaksData  = ncread('example.nc','peaks');
```

Display information about the `peaksData` output.

```
whos peaksData
```

```
  Name              Size            Bytes  Class    Attributes

  peaksData      50x50               5000  int16
```

Read the `description` attribute associated with the variable.

```
peaksDesc  = ncreadatt('example.nc','peaks','description')
```

```
peaksDesc =

z = peaks(50);
```

Create a three-dimensional surface plot of the variable data. Use the value of the `description` attribute as the title of the figure.

```
surf(double(peaksData))
title(peaksDesc);
```

Read the `description` attribute associated with the `/grid1/` group. Specify the group name as the second input to the `ncreadatt` function.

```
g = ncreadatt('example.nc','/grid1/','description')
```

```
g =

This is a group attribute.
```

Read the global attribute, `creation_date`. For global attributes, specify the second input argument to `ncreadatt` as `'/'`.

```
creation_date = ncreadatt('example.nc','/','creation_date')
```

```
creation_date =

29-Mar-2010
```

Find All Unlimited Dimensions in NetCDF File

This example shows how to find all unlimited dimensions in a group in a NetCDF file, using high-level functions.

Get information about the `/grid2/` group in the sample file, `example.nc`, using the `ncinfo` function.

```
ginfo = ncinfo('example.nc','/grid2/')
```

```
ginfo =

      Filename: '\\matlabroot\toolbox\matlab\demos\example.nc'
          Name: 'grid2'
    Dimensions: [1x3 struct]
     Variables: [1x1 struct]
    Attributes: [1x1 struct]
        Groups: []
        Format: 'netcdf4'
```

`ncinfo` returns a structure array containing information about the group.

Get a vector of the Boolean values that indicate the unlimited dimensions for this group.

```
unlimDims = [ginfo.Dimensions.Unlimited]
```

```
unlimDims =

     0     0     1
```

Use the `unlimDims` vector to display the unlimited dimension.

```
disp(ginfo.Dimensions(unlimDims))
```

```
         Name: 'time'
       Length: 0
    Unlimited: 1
```

Read from NetCDF File Using Low-Level Functions

This example shows how to get information about the dimensions, variables, and attributes in a NetCDF file using MATLAB low-level functions in the `netcdf` package. To use these functions effectively, you should be familiar with the NetCDF C Interface.

Open NetCDF File

Open the sample NetCDF file, `example.nc`, using the `netcdf.open` function, with read-only access.

```
ncid = netcdf.open('example.nc','NC_NOWRITE')
```

```
ncid = 65536
```

`netcdf.open` returns a file identifier.

Get Information About NetCDF File

Get information about the contents of the file using the `netcdf.inq` function. This function corresponds to the `nc_inq` function in the NetCDF library C API.

```
[ndims,nvars,natts,unlimdimID] = netcdf.inq(ncid)
```

```
ndims = 3

nvars = 3

natts = 1

unlimdimID = -1
```

`netcdf.inq` returns the number of dimensions, variables, and global attributes in the file, and returns the identifier of the unlimited dimension in the file. An unlimited dimension can grow.

Get the name of the global attribute in the file using the `netcdf.inqAttName` function. This function corresponds to the `nc_inq_attname` function in the NetCDF library C API. To get the name of an attribute, you must specify the ID of the variable the attribute is associated with and the attribute number. To access a global attribute, which is not associated with a particular variable, use the constant `'NC_GLOBAL'` as the variable ID.

```
global_att_name = netcdf.inqAttName(ncid,...
    netcdf.getConstant('NC_GLOBAL'),0)
```

```
global_att_name =
'creation_date'
```

Get information about the data type and length of the attribute using the `netcdf.inqAtt` function. This function corresponds to the `nc_inq_att` function in the NetCDF library C API. Again, specify the variable ID using `netcdf.getConstant('NC_GLOBAL')`.

```
[xtype,attlen] = netcdf.inqAtt(ncid,...
    netcdf.getConstant('NC_GLOBAL'),global_att_name)
```

```
xtype = 2

attlen = 11
```

Get the value of the attribute, using the `netcdf.getAtt` function.

```
global_att_value = netcdf.getAtt(ncid,...
    netcdf.getConstant('NC_GLOBAL'),global_att_name)
```

```
global_att_value = 
'29-Mar-2010'
```

Get information about the first dimension in the file, using the `netcdf.inqDim` function. This function corresponds to the `nc_inq_dim` function in the NetCDF library C API. The second input to `netcdf.inqDim` is the dimension ID, which is a zero-based index that identifies the dimension. The first dimension has the index value `0`.

```
[dimname,dimlen] = netcdf.inqDim(ncid,0)
```

```
dimname = 
'x'

dimlen = 50
```

`netcdf.inqDim` returns the name and length of the dimension.

Get information about the first variable in the file using the `netcdf.inqVar` function. This function corresponds to the `nc_inq_var` function in the NetCDF library C API. The second input to `netcdf.inqVar` is the variable ID, which is a zero-based index that identifies the variable. The first variable has the index value `0`.

```
[varname,vartype,dimids,natts] = netcdf.inqVar(ncid,0)
```

```
varname = 
'avagadros_number'

vartype = 6

dimids =

     []

natts = 1
```

`netcdf.inqVar` returns the name, data type, dimension ID, and the number of attributes associated with the variable. The data type information returned in `vartype` is the numeric value of the NetCDF data type constants, such as, `NC_INT` and `NC_BYTE`. See the NetCDF documentation for information about these constants.

Read Data from NetCDF File

Read the data associated with the variable, `avagadros_number`, in the example file, using the `netcdf.getVar` function. The second input to `netcdf.getVar` is the variable ID, which is a zero-based index that identifies the variable. The `avagadros_number` variable has the index value `0`.

```
A_number = netcdf.getVar(ncid,0)
```

```
A_number = 6.0221e+23
```

View the data type of `A_number`.

```
whos A_number
```

```
  Name          Size            Bytes  Class     Attributes

  A_number      1x1                 8  double
```

The functions in the `netcdf` package automatically choose the MATLAB class that best matches the NetCDF data type, but you can also specify the class of the return data by using an optional argument to `netcdf.getVar`.

Read the data associated with `avagadros_number` and return the data as class `single`.

```
A_number = netcdf.getVar(ncid,0,'single');
whos A_number
```

```
  Name          Size            Bytes  Class     Attributes

  A_number      1x1                 4  single
```

Close NetCDF File

Close the NetCDF file, `example.nc`.

```
netcdf.close(ncid)
```

See Also

`ncdisp` | `ncinfo` | `ncread` | `ncreadatt`

More About

- “Map NetCDF API Syntax to MATLAB Syntax” on page 7-11

External Websites

- NetCDF C Interface

Resolve Errors Reading OPeNDAP Data

When you have trouble reading OPeNDAP data, consider these factors.

- OPeNDAP data is being pulled over the network from a server on the Internet. Pulling large data could be slow. Speed and reliability depends on their network connection
- OPeNDAP capability does not support proxy servers or any authentication
- Failure to open an OPeNDAP link could have multiple causes:
  - Invalid URL
  - Local machine firewall/network firewall does not allow any external connections.
  - Local machine firewall/network firewall does not allow external connections on the OPeNDAP protocol.
  - Remote server is down.
  - Remote server will not serve the amount of data being requested. In this case, you can read data in subsets or chunks.
  - Remote server is incorrectly configured.

Export to NetCDF Files

Create, merge, and write NetCDF files using high-level functions and the `netcdf` package low-level functions.

In this section...

MATLAB NetCDF Capabilities

Network Common Data Form (NetCDF) is a set of software libraries and machine-independent data formats that support the creation, access, and sharing of array-oriented scientific data. NetCDF is used by a wide range of engineering and scientific fields that want a standard way to store data so that it can be shared.

MATLAB high-level functions make it easy to export data to a netCDF file. MATLAB low-level functions provide access to the routines in the NetCDF C library. To use the low-level functions effectively, you should be familiar with the NetCDF C Interface. The NetCDF documentation is available at the Unidata website.

Note For information about exporting to Common Data Format (CDF) files, which have a separate and incompatible format, see "Export to CDF Files" on page 7-8.

Create New NetCDF File From Existing File or Template

This example shows how to create a new NetCDF file that contains the variable, dimension, and group definitions of an existing file, but uses a different format.

Create a file containing one variable, using the `nccreate` function.

```
nccreate('myfile.nc','myvar')
```

Write data to the file.

```
A = 99;
ncwrite('myfile.nc','myvar',A)
```

Read the variable, dimension, and group definitions from the file using `ncinfo`. This information defines the file's *schema*.

```
S = ncinfo('myfile.nc');
```

Get the format of the file.

```
file_fmt = S.Format
```

```
file_fmt =
'netcdf4_classic'
```

Change the value of the `Format` field in the structure, `S`, to another supported NetCDF format.

```
S.Format = 'netcdf4';
```

Create a new version of the file that uses the new format, using the `ncwriteschema` function. A schema defines the structure of the file but does not contain any of the data that was in the original file.

```
ncwriteschema('newfile.nc',S)
S = ncinfo('newfile.nc');
```

Note: When you convert a file's format using `ncwriteschema`, you might get a warning message if the original file format includes fields that are not supported by the new format. For example, the netcdf4 format supports fill values but the NetCDF classic format does not. In these cases, `ncwriteschema` still creates the file, but omits the field that is undefined in the new format.

View the format of the new file.

```
new_fmt = S.Format
```

```
new_fmt =
'netcdf4'
```

The new file, `newfile.nc`, contains the variable and dimension definitions of `myfile.nc`, but does not contain the data.

Write data to the new file.

```
ncwrite('newfile.nc','myvar',A)
```

Merge Two NetCDF Files

This example shows how to merge two NetCDF files using high-level functions. The combined file contains the variable and dimension definitions of the files that are combined, but does not contain the data in these original files.

Create a NetCDF file named `ex1.nc` and define a variable named `myvar`. Then, write data to the variable and display the file contents.

```
nccreate('ex1.nc','myvar');
ncwrite('ex1.nc','myvar',55)
ncdisp('ex1.nc')
```

```
Source:
           pwd\ex1.nc
Format:
           netcdf4_classic
Variables:
    myvar
           Size:       1x1
           Dimensions:
           Datatype:   double
```

Create a second file and define a variable named `myvar2`. Then, write data to the variable and display the file contents.

```
nccreate('ex2.nc','myvar2');
ncwrite('ex2.nc','myvar2',99)
ncdisp('ex2.nc')
```

```
Source:
           pwd\ex2.nc
Format:
           netcdf4_classic
Variables:
    myvar2
           Size:       1x1
           Dimensions:
           Datatype:   double
```

Get the schema of each of the files, using the `ncinfo` function.

```
info1 = ncinfo('ex1.nc')
```

```
info1 =

      Filename: 'pwd\ex1.nc'
          Name: '/'
    Dimensions: []
     Variables: [1x1 struct]
    Attributes: []
        Groups: []
        Format: 'netcdf4_classic'
```

```
info2 = ncinfo('ex2.nc')
```

```
info2 =

      Filename: 'pwd\ex2.nc'
          Name: '/'
    Dimensions: []
     Variables: [1x1 struct]
    Attributes: []
        Groups: []
        Format: 'netcdf4_classic'
```

Create a new NetCDF file that uses the schema of the first example file, using the `ncwriteschema` function. Then, display the file contents.

```
ncwriteschema('combined.nc',info1)
ncdisp('combined.nc')
```

```
Source:
           pwd\combined.nc
Format:
           netcdf4_classic
Variables:
    myvar
           Size:       1x1
           Dimensions:
           Datatype:   double
           Attributes:
                       _FillValue = 9.969209968386869e+36
```

Add the schema from `ex2.nc` to `combined.nc`, using `ncwriteschema`.

```
ncwriteschema('combined.nc',info2)
```

View the contents of the combined file.

```
ncdisp('combined.nc')
```

```
Source:
           pwd\combined.nc
Format:
           netcdf4_classic
Variables:
    myvar
           Size:       1x1
           Dimensions:
           Datatype:   double
           Attributes:
                       _FillValue = 9.969209968386869e+36
    myvar2
           Size:       1x1
           Dimensions:
           Datatype:   double
           Attributes:
                       _FillValue = 9.969209968386869e+36
```

The file contains the `myvar` variable defined in the first example file and the `myvar2` variable defined in the second file.

Write Data to NetCDF File Using Low-Level Functions

This example shows how to use low-level functions to write data to a NetCDF file. The MATLAB® low-level functions provide access to the routines in the NetCDF C library. MATLAB groups the functions into a package, called `netcdf`. To call one of the functions in the package, you must prefix the function name with the package name.

To use the MATLAB NetCDF functions effectively, you should be familiar with the information about the NetCDF C Interface.

To run this example, you must have write permission in your current folder.

Create a 1-by-50 variable of numeric values named `my_data` in the MATLAB workspace. The vector is of class `double`.

```
my_data = linspace(0,49,50);
```

Create a NetCDF file named `my_file.nc`, using the `netcdf.create` function. The `NOCLOBBER` parameter is a NetCDF file access constant that indicates that you do not want to overwrite an existing file with the same name.

```
ncid = netcdf.create('my_file.nc','NOCLOBBER');
```

`netcdf.create` returns a file identifier, `ncid`. When you create a NetCDF file, the file opens in define mode. You must be in define mode to define dimensions and variables.

Define a dimension in the file, using the `netcdf.defDim` function. This function corresponds to the `nc_def_dim` function in the NetCDF library C API. You must define dimensions in the file before you can define variables and write data to the file. In this case, define a dimension named `my_dim` with length 50.

```
dimid = netcdf.defDim(ncid,'my_dim',50)
```

```
dimid = 0
```

`netcdf.defDim` returns a dimension identifier that corresponds to the new dimension. Identifiers are zero-based indexes.

Define a variable named `my_var` on the dimension, using the `netcdf.defVar` function. This function corresponds to the `nc_def_var` function in the NetCDF library C API. Specify the NetCDF data type of the variable, in this case, `NC_BYTE`.

```
varid = netcdf.defVar(ncid,'my_var','NC_BYTE',dimid)
```

```
varid = 0
```

`netcdf.defVar` returns a variable identifier that corresponds to `my_var`.

Take the NetCDF file out of define mode. To write data to a file, you must be in data mode.

```
netcdf.endDef(ncid)
```

Write the data from the MATLAB workspace into the variable in the NetCDF file, using the `netcdf.putVar` function. The data in the workspace is of class `double` but the variable in the NetCDF file is of type `NC_BYTE`. The MATLAB NetCDF functions automatically do the conversion.

```
netcdf.putVar(ncid,varid,my_data)
```

Close the file, using the `netcdf.close` function.

```
netcdf.close(ncid)
```

Verify that the data was written to the file by opening the file and reading the data from the variable into a new variable in the MATLAB workspace.

```
ncid2 = netcdf.open('my_file.nc','NC_NOWRITE');
x = netcdf.getVar(ncid2,0);
```

View the data type of `x`.

```
whos x
```

```
  Name      Size            Bytes  Class    Attributes

  x        50x1                50  int8
```

MATLAB stores data in column-major order while the NetCDF C API uses row-major order. `x` represents the data stored in the NetCDF file and is therefore 50-by-1 even though the original vector in the MATLAB workspace, `my_data`, is 1-by-50. Because you stored the data in the NetCDF file as `NC_BYTE`, MATLAB reads the data from the variable into the workspace as class `int8`.

Close the file.

```
netcdf.close(ncid2)
```

See Also

More About

- “Map NetCDF API Syntax to MATLAB Syntax” on page 7-11

External Websites

- NetCDF C Interface

Importing Flexible Image Transport System (FITS) Files

The FITS file format is the standard data format used in astronomy, endorsed by both NASA and the International Astronomical Union (IAU). For more information about the FITS standard, go to the FITS Web site, `https://fits.gsfc.nasa.gov/`.

The FITS file format is designed to store scientific data sets consisting of multidimensional arrays (1-D spectra, 2-D images, or 3-D data cubes) and two-dimensional tables containing rows and columns of data. A data file in FITS format can contain multiple components, each marked by an ASCII text header followed by binary data. The first component in a FITS file is known as the *primary*, which can be followed by any number of other components, called *extensions*, in FITS terminology. For a complete list of extensions, see `fitsread`.

To get information about the contents of a Flexible Image Transport System (FITS) file, use the `fitsinfo` function. The `fitsinfo` function returns a structure containing the information about the file and detailed information about the data in the file.

To import data into the MATLAB workspace from a Flexible Image Transport System (FITS) file, use the `fitsread` function. Using this function, you can import the primary data in the file or you can import the data in any of the extensions in the file, such as the `Image` extension, as shown in this example.

1. Determine which extensions the FITS file contains, using the `fitsinfo` function.

   ```
   info = fitsinfo('tst0012.fits')

   info =

          Filename: 'matlabroot\tst0012.fits'
       FileModDate: '12-Mar-2001 19:37:46'
          FileSize: 109440
          Contents: {'Primary'  'Binary Table'  'Unknown'  'Image'  'ASCII Table'}
       PrimaryData: [1x1 struct]
       BinaryTable: [1x1 struct]
           Unknown: [1x1 struct]
             Image: [1x1 struct]
        AsciiTable: [1x1 struct]
   ```

   The `info` structure shows that the file contains several extensions including the `Binary Table`, `ASCII Table`, and `Image` extensions.
2. Read data from the file.

   To read the `Primary` data in the file, specify the filename as the only argument:

   ```
   pdata = fitsread('tst0012.fits');
   ```

   To read any of the extensions in the file, you must specify the name of the extension as an optional parameter. This example reads the `Binary Table` extension from the FITS file:

   ```
   bindata = fitsread('tst0012.fits','binarytable');
   ```

Importing HDF5 Files

In this section...

Overview

Hierarchical Data Format, Version 5, (HDF5) is a general-purpose, machine-independent standard for storing scientific data in files, developed by the National Center for Supercomputing Applications (NCSA). HDF5 is used by a wide range of engineering and scientific fields that want a standard way to store data so that it can be shared. For more information about the HDF5 file format, read the HDF5 documentation available at the HDF Web site (`https://www.hdfgroup.org`).

MATLAB provides two methods to import data from an HDF5 file:

- High-level functions that make it easy to import data, when working with numeric datasets
- Low-level functions that enable more complete control over the importing process, by providing access to the routines in the HDF5 C library

---

Note For information about importing to HDF4 files, which have a separate, incompatible format, see "Import HDF4 Files Programmatically" on page 7-43.

Using the High-Level HDF5 Functions to Import Data

MATLAB includes several functions that you can use to examine the contents of an HDF5 file and import data from the file into the MATLAB workspace.

Note You can only use the high-level functions to read numeric datasets or attributes. To read non-numeric datasets or attributes, you must use the low-level interface on page 7-32.

- `h5disp` — View the contents of an HDF5 file
- `h5info` — Create a structure that contains all the metadata defining an HDF5 file
- `h5read` — Read data from a variable in an HDF5 file
- `h5readatt` — Read data from an attribute associated with a variable in an HDF5 file or with the file itself (a global attribute).

For details about how to use these functions, see their reference pages, which include examples. The following sections illustrate some common usage scenarios.

Determining the Contents of an HDF5 File

HDF5 files can contain data and metadata, called *attributes*. HDF5 files organize the data and metadata in a hierarchical structure similar to the hierarchical structure of a UNIX file system.

In an HDF5 file, the directories in the hierarchy are called *groups*. A group can contain other groups, data sets, attributes, links, and data types. A data set is a collection of data, such as a multidimensional numeric array or string. An attribute is any data that is associated with another entity, such as a data set. A link is similar to a UNIX file system symbolic link. Links are a way to reference objects without having to make a copy of the object.

Data types are a description of the data in the data set or attribute. Data types tell how to interpret the data in the data set.

To get a quick view into the contents of an HDF5 file, use the `h5disp` function.

```
h5disp('example.h5')

HDF5 example.h5
Group '/'
    Attributes:
        'attr1':  97 98 99 100 101 102 103 104 105 0
        'attr2':  2x2 H5T_INTEGER
    Group '/g1'
        Group '/g1/g1.1'
            Dataset 'dset1.1.1'
                Size:  10x10
                MaxSize:  10x10
                Datatype:   H5T_STD_I32BE (int32)
                ChunkSize:  []
                Filters:  none
                Attributes:
                    'attr1':  49 115 116 32 97 116 116 114 105 ...
                    'attr2':  50 110 100 32 97 116 116 114 105 ...
            Dataset 'dset1.1.2'
                Size:  20
                MaxSize:  20
                Datatype:   H5T_STD_I32BE (int32)
                ChunkSize:  []
                Filters:  none
        Group '/g1/g1.2'
            Group '/g1/g1.2/g1.2.1'
                Link 'slink'
                    Type:  soft link
    Group '/g2'
        Dataset 'dset2.1'
            Size:  10
            MaxSize:  10
            Datatype:   H5T_IEEE_F32BE (single)
            ChunkSize:  []
            Filters:  none
        Dataset 'dset2.2'
            Size:  5x3
            MaxSize:  5x3
            Datatype:   H5T_IEEE_F32BE (single)
            ChunkSize:  []
            Filters:  none
                    .
                    .
                    .
```

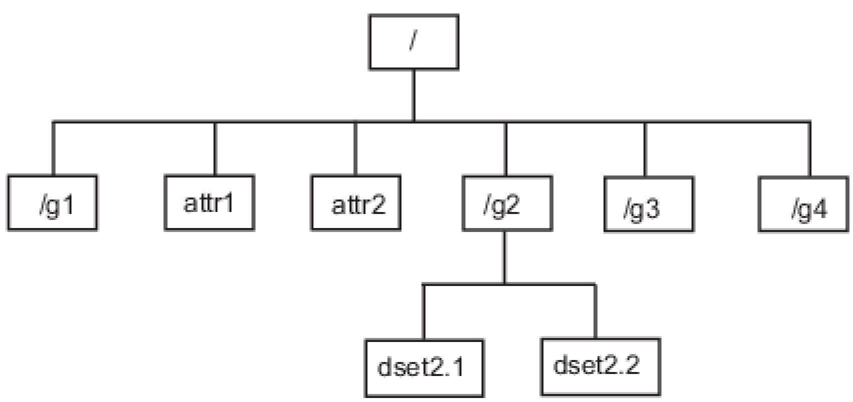
To explore the hierarchical organization of an HDF5 file, use the `h5info` function. `h5info` returns a structure that contains various information about the HDF5 file, including the name of the file.

```
info = h5info('example.h5')
info =

         Filename: 'matlabroot\matlab\toolbox\matlab\demos\example.h5'
          Name: '/'
        Groups: [4x1 struct]
      Datasets: []
     Datatypes: []
         Links: []
    Attributes: [2x1 struct]
```

By looking at the `Groups` and `Attributes` fields, you can see that the file contains four groups and two attributes. The `Datasets`, `Datatypes`, and `Links` fields are all empty, indicating that the root group does not contain any data sets, data types, or links. To explore the contents of the sample HDF5 file further, examine one of the structures in `Groups`. The following example shows the contents of the second structure in this field.

```
level2 = info.Groups(2)

level2 =

          Name: '/g2'
        Groups: []
      Datasets: [2x1 struct]
     Datatypes: []
         Links: []
    Attributes: []
```

In the sample file, the group named `/g2` contains two data sets. The following figure illustrates this part of the sample HDF5 file organization.



To get information about a data set, such as its name, dimensions, and data type, look at either of the structures returned in the `Datasets` field.

```
dataset1 = level2.Datasets(1)

dataset1 =
      Filename: 'matlabroot\example.h5'
          Name: '/g2/dset2.1'
          Rank: 1
      Datatype: [1x1 struct]
          Dims: 10
       MaxDims: 10
```

```
        Layout: 'contiguous'
    Attributes: []
         Links: []
     Chunksize: []
     Fillvalue: []
```

Importing Data from an HDF5 File

To read data or metadata from an HDF5 file, use the `h5read` function. As arguments, specify the name of the HDF5 file and the name of the data set. (To read the value of an attribute, you must use `h5readatt`.)

To illustrate, this example reads the data set, `/g2/dset2.1` from the HDF5 sample file `example.h5`.

```
data = h5read('example.h5','/g2/dset2.1')

data =

    1.0000
    1.1000
    1.2000
    1.3000
    1.4000
    1.5000
    1.6000
    1.7000
    1.8000
    1.9000
```

Mapping HDF5 Datatypes to MATLAB Datatypes

When the `h5read` function reads data from an HDF5 file into the MATLAB workspace, it maps HDF5 data types toMATLAB data types, as shown in the table below.

HDF5 Data Type	h5read Returns
Bit-field	Array of packed 8-bit integers
Float	MATLAB single and double types, provided that they occupy 64 bits or fewer
Integer types, signed and unsigned	Equivalent MATLAB integer types, signed and unsigned
Opaque	Array of `uint8` values
Reference	Returns the actual data pointed to by the reference, not the value of the reference.
Strings, fixed-length and variable length	Cell array of character vectors
Enums	Cell array of character vectors, where each enumerated value is replaced by the corresponding member name
Compound	1-by-1 struct array; the dimensions of the dataset are expressed in the fields of the structure.
Arrays	Array of values using the same datatype as the HDF5 array. For example, if the array is of signed 32-bit integers, the MATLAB array will be of type `int32`.

The example HDF5 file included with MATLAB includes examples of all these datatypes.

For example, the data set `/g3/string` is a string.

```
h5disp('example.h5','/g3/string')
HDF5 example.h5
Dataset 'string'
    Size:  2
    MaxSize:  2
    Datatype:   H5T_STRING
        String Length: 3
        Padding: H5T_STR_NULLTERM
        Character Set: H5T_CSET_ASCII
        Character Type: H5T_C_S1
    ChunkSize:  []
    Filters:  none
    FillValue:  ''
```

Now read the data from the file, MATLAB returns it as a cell array of character vectors.

```
s = h5read('example.h5','/g3/string')

s =

    'ab '
    'de '

>> whos s
  Name      Size            Bytes  Class    Attributes

  s         2x1               236  cell
```

The compound data types are always returned as a 1-by-1 struct. The dimensions of the data set are expressed in the fields of the struct. For example, the data set `/g3/compound2D` is a compound datatype.

```
h5disp('example.h5','/g3/compound2D')
HDF5 example.h5
Dataset 'compound2D'
    Size:  2x3
    MaxSize:  2x3
    Datatype:   H5T_COMPOUND
        Member 'a':  H5T_STD_I8LE (int8)
        Member 'b':  H5T_IEEE_F64LE (double)
    ChunkSize:  []
    Filters:  none
    FillValue:  H5T_COMPOUND
```

Now read the data from the file, MATLAB returns it as a 1-by-1 struct.

```
data = h5read('example.h5','/g3/compound2D')

data =

    a: [2x3 int8]
    b: [2x3 double]
```

Read an HDF5 Dataset with Dynamically Loaded Filters

In R2015a and later releases, MATLAB supports reading HDF5 datasets that are written using a third-party filter. To read the datasets using the dynamically loaded filter feature, you must:

- Install the HDF5 filter plugin on your system as a shared library or a DLL.
- Set the `HDF5_PLUGIN_PATH` environment variable to point to the installation.

For more information see, HDF5 Dynamically Loaded Filters.

Note Writing HDF5 datasets using dynamically loaded filters is not supported.

Using the Low-Level HDF5 Functions to Import Data

MATLAB provides direct access to dozens of functions in the HDF5 library with *low-level* functions that correspond to the functions in the HDF5 library. In this way, you can access the features of the HDF5 library from MATLAB, such as reading and writing complex data types and using the HDF5 subsetting capabilities. For more information, see "Using the MATLAB Low-Level HDF5 Functions to Export Data" on page 7-34.

Exporting to HDF5 Files

In this section...

Overview

Hierarchical Data Format, Version 5, (HDF5) is a general-purpose, machine-independent standard for storing scientific data in files, developed by the National Center for Supercomputing Applications (NCSA). HDF5 is used by a wide range of engineering and scientific fields that want a standard way to store data so that it can be shared. For more information about the HDF5 file format, read the HDF5 documentation available at the HDF Web site (`https://www.hdfgroup.org`).

MATLAB provides two methods to export data to an HDF5 file:

- High-level functions that simplify the process of exporting data, when working with numeric datasets
- Low-level functions that provide a MATLAB interface to routines in the HDF5 C library

Note For information about exporting to HDF4 files, which have a separate and incompatible format, see "Export to HDF4 Files" on page 7-51.

Using the MATLAB High-Level HDF5 Functions to Export Data

The easiest way to write data or metadata from the MATLAB workspace to an HDF5 file is to use these MATLAB high-level functions.

Note You can use the high-level functions only with numeric data. To write nonnumeric data, you must use the low-level interface on page 7-34.

- `h5create` — Create an HDF5 dataset
- `h5write` — Write data to an HDF5 dataset
- `h5writeatt` — Write data to an HDF5 attribute

For details about how to use these functions, see their reference pages, which include examples. The following sections illustrate some common usage scenarios.

Writing a Numeric Array to an HDF5 Dataset

This example creates an array and then writes the array to an HDF5 file.

1. Create a MATLAB variable in the workspace. This example creates a 5-by-5 array of `uint8` values.

   ```
   testdata = uint8(magic(5))
   ```

2 Create the HDF5 file and the dataset, using `h5create`.

```
h5create('my_example_file.h5', '/dataset1', size(testdata))
```

3 Write the data to the HDF5 file.

```
h5write('my_example_file.h5', '/dataset1', testdata)
```

Using the MATLAB Low-Level HDF5 Functions to Export Data

MATLAB provides direct access to dozens of functions in the HDF5 library with *low-level* functions that correspond to the functions in the HDF5 library. In this way, you can access the features of the HDF5 library from MATLAB, such as reading and writing complex data types and using the HDF5 subsetting capabilities.

The HDF5 library organizes the library functions into collections, called *interfaces*. For example, all the routines related to working with files, such as opening and closing, are in the H5F interface, where *F* stands for file. MATLAB organizes the low-level HDF5 functions into classes that correspond to each HDF5 interface. For example, the MATLAB functions that correspond to the HDF5 file interface (H5F) are in the @H5F class folder.

The following sections provide more detail about how to use the MATLAB HDF5 low-level functions.

- "Map HDF5 Function Syntax to MATLAB Function Syntax" on page 7-34
- "Map Between HDF5 Data Types and MATLAB Data Types" on page 7-36
- "Report Data Set Dimensions" on page 7-36
- "Write Data to HDF5 Data Set Using MATLAB Low-Level Functions" on page 7-37
- "Write a Large Data Set" on page 7-38
- "Preserve Correct Layout of Your Data" on page 7-39

Note This section does not describe all features of the HDF5 library or explain basic HDF5 programming concepts. To use the MATLAB HDF5 low-level functions effectively, refer to the official HDF5 documentation available at `https://www.hdfgroup.org`.

Map HDF5 Function Syntax to MATLAB Function Syntax

In most cases, the syntax of the MATLAB low-level HDF5 functions matches the syntax of the corresponding HDF5 library functions. For example, the following is the function signature of the `H5Fopen` function in the HDF5 library. In the HDF5 function signatures, `hid_t` and `herr_t` are HDF5 types that return numeric values that represent object identifiers or error status values.

```
hid_t H5Fopen(const char *name, unsigned flags, hid_t access_id) /* C syntax */
```

In MATLAB, each function in an HDF5 interface is a method of a MATLAB class. The following shows the signature of the corresponding MATLAB function. First note that, because it's a method of a class, you must use the dot notation to call the MATLAB function: `H5F.open`. This MATLAB function accepts the same three arguments as the HDF5 function: a character vector containing the name, an HDF5-defined constant for the flags argument, and an HDF5 property list ID. You use property lists to specify characteristics of many different HDF5 objects. In this case, it's a file access property list. Refer to the HDF5 documentation to see which constants can be used with a particular function and note that, in MATLAB, constants are passed as character vectors.

```
file_id = H5F.open(name, flags, plist_id)
```

There are, however, some functions where the MATLAB function signature is different than the corresponding HDF5 library function. The following describes some general differences that you should keep in mind when using the MATLAB low-level HDF5 functions.

- **HDF5 output parameters become MATLAB return values** — Some HDF5 library functions use function parameters to return data. Because MATLAB functions can return multiple values, these output parameters become return values. To illustrate, the HDF5 `H5Dread` function returns data in the `buf` parameter.

  ```
  herr_t H5Dread(hid_t dataset_id,
                       hid_t mem_type_id,
                       hid_t mem_space_id,
                       hid_t file_space_id,
                       hid_t xfer_plist_id,
                       void * buf )   /* C syntax */
  ```

  The corresponding MATLAB function changes the output parameter `buf` into a return value. Also, in the MATLAB function, the nonzero or negative value `herr_t` return values become MATLAB errors. Use MATLAB `try-catch` statements to handle errors.

  ```
  buf = H5D.read(dataset_id,
                        mem_type_id,
                          mem_space_id,
                         file_space_id,
                          plist_id)
  ```

- **String length parameters are unnecessary** — The length parameter, used by some HDF5 library functions to specify the length of a string parameter, is not necessary in the corresponding MATLAB function. For example, the `H5Aget_name` function in the HDF5 library includes a buffer as an output parameter and the size of the buffer as an input parameter.

  ```
  ssize_t H5Aget_name(hid_t attr_id,
                      size_t buf_size,
                      char *buf ) /* C syntax */
  ```

  The corresponding MATLAB function changes the output parameter `buf` into a return value and drops the `buf_size` parameter.

  ```
  buf = H5A.get_name(attr_id)
  ```

- **Use an empty array to specify NULL** — Wherever HDF5 library functions accept the value `NULL`, the corresponding MATLAB function uses empty arrays (`[]`). For example, the `H5Dfill` function in the HDF5 library accepts the value `NULL` in place of a specified fill value.

  ```
  herr_t H5Dfill(const void *fill,
                 hid_t fill_type_id, void *buf,
                 hid_t buf_type_id,
                 hid_t space_id ) /* C syntax */
  ```

  When using the corresponding MATLAB function, you can specify an empty array (`[]`) instead of `NULL`.

- **Use cell arrays to specify multiple constants** — Some functions in the HDF5 library require you to specify an array of constants. For example, in the `H5Screate_simple` function, to specify that a dimension in the data space can be unlimited, you use the constant `H5S_UNLIMITED` for the dimension in `maxdims`. In MATLAB, because you pass constants as character vectors, you must use a cell array of character vectors to achieve the same result. The following code fragment provides an example of using a cell array of character vectors to specify this constant for each dimension of this data space.

  ```
  ds_id = H5S.create_simple(2,[3 4],{'H5S_UNLIMITED' 'H5S_UNLIMITED'});
  ```

Map Between HDF5 Data Types and MATLAB Data Types

When the HDF5 low-level functions read data from an HDF5 file or write data to an HDF5 file, the functions map HDF5 data types to MATLAB data types automatically.

For *atomic* data types, such as commonly used binary formats for numbers (integers and floating point) and characters (ASCII), the mapping is typically straightforward because MATLAB supports similar types. See the table Mapping Between HDF5 Atomic Data Types and MATLAB Data Types for a list of these mappings.

Mapping Between HDF5 Atomic Data Types and MATLAB Data Types

HDF5 Atomic Data Type	MATLAB Data Type
Bit-field	Array of packed 8-bit integers
Float	MATLAB single and double types, provided that they occupy 64 bits or fewer
Integer types, signed and unsigned	Equivalent MATLAB integer types, signed and unsigned
Opaque	Array of `uint8` values
Reference	Array of `uint8` values
String	MATLAB character arrays

For *composite* data types, such as aggregations of one or more atomic data types into structures, multidimensional arrays, and variable-length data types (one-dimensional arrays), the mapping is sometimes ambiguous with reference to the HDF5 data type. In HDF5, a 5-by-5 data set containing a single `uint8` value in each element is distinct from a 1-by-1 data set containing a 5-by-5 array of `uint8` values. In the first case, the data set contains 25 observations of a single value. In the second case, the data set contains a single observation with 25 values. In MATLAB both of these data sets are represented by a 5-by-5 matrix.

If your data is a complex data set, you might need to create HDF5 data types directly to make sure that you have the mapping you intend. See the table Mapping Between HDF5 Composite Data Types and MATLAB Data Types for a list of the default mappings. You can specify the data type when you write data to the file using the `H5Dwrite` function. See the HDF5 data type interface documentation for more information.

Mapping Between HDF5 Composite Data Types and MATLAB Data Types

HDF5 Composite Data Type	MATLAB Data Type
Array	Extends the dimensionality of the data type which it contains. For example, an array of integers in HDF5 would map onto a two dimensional array of integers in MATLAB.
Compound	MATLAB structure. Note: All structures representing HDF5 data in MATLAB are scalar.
Enumeration	Array of integers which each have an associated name
Variable Length	MATLAB 1-D cell arrays

Report Data Set Dimensions

The MATLAB low-level HDF5 functions report data set dimensions and the shape of data sets differently than the MATLAB high-level functions. For ease of use, the MATLAB high-level functions report data set dimensions consistent with MATLAB column-major indexing. To be consistent with the HDF5 library, and to support the possibility of nested data sets and complicated data types, the MATLAB low-level functions report array dimensions using the C row-major orientation.

Write Data to HDF5 Data Set Using MATLAB Low-Level Functions

This example shows how to use the MATLAB® HDF5 low-level functions to write a data set to an HDF5 file and then read the data set from the file.

Create a 2-by-3 array of data to write to an HDF5 file.

```
testdata = [1 3 5; 2 4 6];
```

Create a new HDF5 file named `my_file.h5` in the system temp folder. Use the MATLAB `H5F.create` function to create a file. This MATLAB function corresponds to the HDF5 function, `H5Fcreate`. As arguments, specify the name you want to assign to the file, the type of access you want to the file (`'H5F_ACC_TRUNC'` in this case), and optional additional characteristics specified by a file creation property list and a file access property list. In this case, use default values for these property lists (`'H5P_DEFAULT'`). Pass C constants to the MATLAB function as character vectors.

```
filename = fullfile(tempdir,'my_file.h5');
fileID = H5F.create(filename,'H5F_ACC_TRUNC','H5P_DEFAULT','H5P_DEFAULT');
```

`H5F.create` returns a file identifier corresponding to the HDF5 file.

Create the data set in the file to hold the MATLAB variable. In the HDF5 programming model, you must define the data type and dimensionality (data space) of the data set as separate entities. First, use the `H5T.copy` function to specify the data type used by the data set, in this case, `double`. This MATLAB function corresponds to the HDF5 function, `H5Tcopy`.

```
datatypeID = H5T.copy('H5T_NATIVE_DOUBLE');
```

`H5T.copy` returns a data type identifier.

Create a data space using `H5S.create_simple`, which corresponds to the HDF5 function, `H5Screate_simple`. The first input, `2`, is the rank of the data space. The second input is an array specifying the size of each dimension of the dataset. Because HDF5 stores data in row-major order and the MATLAB array is organized in column-major order, you should reverse the ordering of the dimension extents before using `H5Screate_simple` to preserve the layout of the data. You can use `fliplr` for this purpose.

```
dims = size(testdata);
dataspaceID = H5S.create_simple(2,fliplr(dims),[]);
```

`H5S.create_simple` returns a data space identifier, `dataspaceID`. Note that other software programs that use row-major ordering (such as `H5DUMP` from the HDF Group) might report the size of the dataset to be 3-by-2 instead of 2-by-3.

Create the data set using `H5D.create`, which corresponds to the HDF5 function, `H5Dcreate`. Specify the file identifier, the name you want to assign to the data set, the data type identifier, the data space identifier, and a data set creation property list identifier as arguments. `'H5P_DEFAULT'` specifies the default property list settings.

```
dsetname = 'my_dataset';
datasetID = H5D.create(fileID,dsetname,datatypeID,dataspaceID,'H5P_DEFAULT');
```

`H5D.create` returns a data set identifier, `datasetID`.

Write the data to the data set using `H5D.write`, which corresponds to the HDF5 function, `H5Dwrite`. The input arguments are the data set identifier, the memory data type identifier, the memory space identifier, the data space identifier, the transfer property list identifier and the name of the MATLAB variable to write to the data set. The constant, `'H5ML_DEFAULT'`, specifies automatic mapping to HDF5 data types. The constant, `'H5S_ALL'`, tells `H5D.write` to write all the data to the file.

```
H5D.write(datasetID,'H5ML_DEFAULT','H5S_ALL','H5S_ALL',...
    'H5P_DEFAULT',testdata);
```

Close the data set, data space, data type, and file objects. If used inside a MATLAB function, these identifiers are closed automatically when they go out of scope.

```
H5D.close(datasetID);
H5S.close(dataspaceID);
H5T.close(datatypeID);
H5F.close(fileID);
```

Open the HDF5 file in order to read the data set you wrote. Use `H5F.open` to open the file for read-only access. This MATLAB function corresponds to the HDF5 function, `H5Fopen`.

```
fileID = H5F.open(filename,'H5F_ACC_RDONLY','H5P_DEFAULT');
```

Open the data set to read using `H5D.open`, which corresponds to the HDF5 function, `H5Dopen`. Specify as arguments the file identifier and the name of the data set, defined earlier in the example.

```
datasetID = H5D.open(fileID,dsetname);
```

Read the data into the MATLAB workspace using `H5D.read`, which corresponds to the HDF5 function, `H5Dread`. The input arguments are the data set identifier, the memory data type identifier, the memory space identifier, the data space identifier, and the transfer property list identifier.

```
returned_data = H5D.read(datasetID,'H5ML_DEFAULT',...
    'H5S_ALL','H5S_ALL','H5P_DEFAULT');
```

Compare the original MATLAB variable, `testdata`, with the variable just created, `returned_data`.

```
isequal(testdata,returned_data)
```

```
ans = logical
   1
```

The two variables are the same.

Write a Large Data Set

To write a large data set, you must use the chunking capability of the HDF5 library. To do this, create a property list and use the `H5P.set_chunk` function to set the chunk size in the property list. Suppose the dimensions of your data set are `[2^16 2^16]` and the chunk size is 1024-by-1024. You then pass the property list as the last argument to the data set creation function, `H5D.create`, instead of using the `H5P_DEFAULT` value.

```
dims = [2^16 2^16];
plistID = H5P.create('H5P_DATASET_CREATE'); % create property list

chunk_size = min([1024 1024], dims); % define chunk size
H5P.set_chunk(plistID, fliplr(chunk_size)); % set chunk size in property list

datasetID = H5D.create(fileID, dsetname, datatypeID, dataspaceID, plistID);
```

Preserve Correct Layout of Your Data

When you use any of the following functions that deal with dataspaces, you should flip dimension extents to preserve the correct layout of the data.

- `H5D.set_extent`
- `H5P.get_chunk`
- `H5P.set_chunk`
- `H5S.create_simple`
- `H5S.get_simple_extent_dims`
- `H5S.select_hyperslab`
- `H5T.array_create`
- `H5T.get_array_dims`

Working with Non-ASCII Characters in HDF5 Files

To enable sharing of HDF5 files across multiple locales, MATLAB supports the use of non-ASCII characters in HDF5 files. This example shows you how to:

- Create HDF5 files containing dataset and attribute names that have non-ASCII characters using the high-level functions.
- Create variable-length string datasets containing non-ASCII characters using the low-level functions.

Create Dataset and Attribute Names Containing Non-ASCII Characters

Create an HDF5 file containing a dataset name and an attribute name that contains non-ASCII characters. To check if the dataset and attribute names appear as expected, write data to the dataset, and display the file information.

Create a dataset with a name (/数据集) that includes non-ASCII characters.

```
dsetName = ['/' char([25968 25454 38598])];
dsetDims = [5 2];
h5create('outfile.h5',['/grp1' dsetName],dsetDims,...
                                'TextEncoding','UTF-8');
```

Write data to the file.

```
dataToWrite = rand(dsetDims);
h5write('outfile.h5',['/grp1' dsetName],dataToWrite);
```

Create an attribute name (屬性名稱) that includes non-ASCII characters and assign a value to the attribute.

```
attrName = char([25967 25453 38597]);
h5writeatt('outfile.h5','/',attrName,'I am an attribute',...
                                      'TextEncoding','UTF-8');
```

Display information about the file and check if the attribute name and dataset name appear correctly.

```
h5disp('outfile.h5')
```

```
HDF5 outfile.h5
Group '/'
    Attributes:
        '/屬性名稱':  'I am an attribute'
    Group '/grp1'
        Dataset '数据集'
            Size:  5x2
            MaxSize:  5x2
            Datatype:   H5T_IEEE_F64LE (double)
            ChunkSize:  []
            Filters:  none
            FillValue:  0.000000
```

Create Variable-Length String Data Containing Non-ASCII Characters

Create a variable-length string dataset to store data containing non-ASCII characters using the low-level functions. Write the data to the dataset. Check if the data is written correctly.

Create data containing non-ASCII characters.

```
dataToWrite = {char([12487 12540 12479]) 'hello' ...
                   char([1605 1585 1581 1576 1575]); ...
               'world' char([1052 1080 1088])    ...
                   char([954 972 963 956 959 962])};
disp(dataToWrite)
```

```
    'データ'    'hello'    'مرحبا'
    'world'    'Мир'      'κόσμος'
```

To write this data into a file, create an HDF5 file, define a group name, and a dataset name within the group.

Create the HDF5 file.

```
fileName = 'outfile.h5';
fileID = H5F.create(fileName,'H5F_ACC_TRUNC',...
                     'H5P_DEFAULT', 'H5P_DEFAULT');
```

To create the group containing non-ASCII characters in its name, first, configure the link creation property.

```
lcplID = H5P.create('H5P_LINK_CREATE');
H5P.set_char_encoding(lcplID,H5ML.get_constant_value('H5T_CSET_UTF8'));
plist = 'H5P_DEFAULT';
```

Then, create the group (グループ).

```
grpName = char([12464 12523 12540 12503]);
grpID = H5G.create(fileID,grpName,lcplID,plist,plist);
```

Create a dataset that contains variable-length string data with non-ASCII characters. First, configure its data type.

```
typeID = H5T.copy('H5T_C_S1');
H5T.set_size(typeID,'H5T_VARIABLE');
H5T.set_cset(typeID,H5ML.get_constant_value('H5T_CSET_UTF8'));
```

Now create the dataset by specifying its name, data type, and dimensions.

```
dsetName = 'datasetUtf8';
dataDims = [2 3];
h5DataDims = fliplr(dataDims);
h5MaxDims = h5DataDims;
spaceID = H5S.create_simple(2,h5DataDims,h5MaxDims);
dsetID = H5D.create(grpID,dsetName,typeID,spaceID,...
             'H5P_DEFAULT','H5P_DEFAULT','H5P_DEFAULT');
```

Write the data to the dataset.

```
H5D.write(dsetID,'H5ML_DEFAULT','H5S_ALL',...
               'H5S_ALL','H5P_DEFAULT',dataToWrite);
```

Read the data back.

```
dataRead = h5read('outfile.h5',['/' grpName '/' dsetName])
```

```
dataRead =

  2×3 cell array

    {'データ'}    {'hello'}    {'مرحبا' }
    {'world'}    {'Мир'  }    {'κόσμος'}
```

Check if data in the file matches the written data.

```
isequal(dataRead,dataToWrite)
```

```
ans =

  logical

   1
```

Close ids.

```
H5D.close(dsetID);
H5S.close(spaceID);
H5T.close(typeID);
H5G.close(grpID);
H5P.close(lcplID);
H5F.close(fileID);
```

See Also

H5A.get_name | H5I.get_name | H5L.get_name_by_idx | H5L.get_val | H5R.get_name | h5create | h5disp | h5info | h5writeatt

Import HDF4 Files Programmatically

In this section...

Overview

Hierarchical Data Format (HDF4) is a general-purpose, machine-independent standard for storing scientific data in files, developed by the National Center for Supercomputing Applications (NCSA). For more information about these file formats, read the HDF documentation at the HDF Web site (`www.hdfgroup.org`).

HDF-EOS is an extension of HDF4 that was developed by the National Aeronautics and Space Administration (NASA) for storage of data returned from the Earth Observing System (EOS). For more information about this extension to HDF4, see the HDF-EOS documentation at the NASA Web site (`www.hdfeos.org`).

MATLAB includes several options for importing HDF4 files, discussed in the following sections.

Note For information about importing HDF5 data, which is a separate, incompatible format, see "Importing HDF5 Files" on page 7-27.

Using the MATLAB HDF4 High-Level Functions

To import data from an HDF or HDF-EOS file, you can use the MATLAB HDF4 high-level function `hdfread`. The `hdfread` function provides a programmatic way to import data from an HDF4 or HDF-EOS file that still hides many of the details that you need to know if you use the low-level HDF functions, described in "Import HDF4 Files Using Low-Level Functions" on page 7-47.

This section describes these high-level MATLAB HDF functions, including

- "Using hdfinfo to Get Information About an HDF4 File" on page 7-43
- "Using hdfread to Import Data from an HDF4 File" on page 7-44

To export data to an HDF4 file, you must use the MATLAB HDF4 low-level functions.

Using hdfinfo to Get Information About an HDF4 File

To get information about the contents of an HDF4 file, use the `hdfinfo` function. The `hdfinfo` function returns a structure that contains information about the file and the data in the file.

This example returns information about a sample HDF4 file included with MATLAB:

```
info = hdfinfo('example.hdf')
```

```
info =

      Filename: 'matlabroot\example.hdf'
    Attributes: [1x2 struct]
        Vgroup: [1x1 struct]
```

```
         SDS: [1x1 struct]
       Vdata: [1x1 struct]
```

To get information about the data sets stored in the file, look at the SDS field.

Using hdfread to Import Data from an HDF4 File

To use the hdfread function, you must specify the data set that you want to read. You can specify the filename and the data set name as arguments, or you can specify a structure returned by the hdfinfo function that contains this information. The following example shows both methods. For information about how to import a subset of the data in a data set, see “Reading a Subset of the Data in a Data Set” on page 7-45.

1. Determine the names of data sets in the HDF4 file, using the hdfinfo function.

   ```
   info = hdfinfo('example.hdf')
   ```

   ```
   info = 

         Filename: 'matlabroot\example.hdf'
       Attributes: [1x2 struct]
           Vgroup: [1x1 struct]
              SDS: [1x1 struct]
            Vdata: [1x1 struct]
   ```

   To determine the names and other information about the data sets in the file, look at the contents of the SDS field. The Name field in the SDS structure gives the name of the data set.

   ```
   dsets = info.SDS
   ```

   ```
   dsets = 

          Filename: 'example.hdf'
              Type: 'Scientific Data Set'
              Name: 'Example SDS'
              Rank: 2
          DataType: 'int16'
        Attributes: []
              Dims: [2x1 struct]
             Label: {}
       Description: {}
             Index: 0
   ```

2. Read the data set from the HDF4 file, using the hdfread function. Specify the name of the data set as a parameter to the function. Note that the data set name is case sensitive. This example returns a 16-by-5 array:

   ```
   dset = hdfread('example.hdf', 'Example SDS')
   ```

   ```
   dset =

        3      4      5      6      7
        4      5      6      7      8
        5      6      7      8      9
        6      7      8      9     10
        7      8      9     10     11
        8      9     10     11     12
        9     10     11     12     13
       10     11     12     13     14
   ```

```
11      12      13      14      15
12      13      14      15      16
13      14      15      16      17
14      15      16      17      18
15      16      17      18      19
16      17      18      19      20
17      18      19      20      21
18      19      20      21      22
```

Alternatively, you can specify the specific field in the structure returned by `hdfinfo` that contains this information. For example, to read a scientific data set, use the `SDS` field.

```
dset = hdfread(info.SDS);
```

Reading a Subset of the Data in a Data Set

To read a subset of a data set, you can use the optional `'index'` parameter. The value of the index parameter is a cell array of three vectors that specify the location in the data set to start reading, the skip interval (e.g., read every other data item), and the amount of data to read (e.g., the length along each dimension). In HDF4 terminology, these parameters are called the *start*, *stride*, and *edge* values.

For example, this code

- Starts reading data at the third row, third column (`[3 3]`).
- Reads every element in the array (`[]`).
- Reads 10 rows and 2 columns (`[10 2]`).

```
subset = hdfread('Example.hdf','Example SDS',...
                 'Index',{[3 3],[],[10 2 ]})

subset =

     7      8
     8      9
     9     10
    10     11
    11     12
    12     13
    13     14
    14     15
    15     16
    16     17
```

Map HDF4 to MATLAB Syntax

Each HDF4 API includes many individual routines that you use to read data from files, write data to files, and perform other related functions. For example, the HDF4 Scientific Data (SD) API includes separate C routines to open (SDopen), close (SDend), and read data (SDreaddata). For the SD API and the HDF-EOS GD and SW APIs, MATLAB provides functions that map to individual C routines in the HDF4 library. These functions are implemented in the matlab.io.hdf4.sd, matlab.io.hdfeos.gd, and matlab.io.hdfeos.sw packages. For example, the SD API includes the C routine SDendaccess to close an HDF4 data set:

```
status = SDendaccess(sds_id); /* C code */
```

To call this routine from MATLAB, use the MATLAB function, matlab.io.hdf4.sd.endAccess. The syntax is similar:

```
sd.endAccess(sdsID)
```

For the remaining supported HDF4 APIs, MATLAB provides a single function that serves as a gateway to all the routines in the particular HDF4 API. For example, the HDF Annotations (AN) API includes the C routine ANend to terminate access to an AN interface:

```
status = ANend(an_id); /* C code */
```

To call this routine from MATLAB, use the MATLAB function associated with the AN API, hdfan. You must specify the name of the routine, minus the API acronym, as the first argument and pass any other required arguments to the routine in the order they are expected. For example,

```
status = hdfan('end',an_id);
```

Some HDF4 API routines use output arguments to return data. Because MATLAB does not support output arguments, you must specify these arguments as return values.

For example, the ANget_tagref routine returns the tag and reference number of an annotation in two output arguments, ann_tag and ann_ref. Here is the C code:

```
status = ANget_tagref(an_id,index,annot_type,ann_tag,ann_ref);
```

To call this routine from MATLAB, change the output arguments into return values:

```
[tag,ref,status] = hdfan('get_tagref',AN_id,index,annot_type);
```

Specify the return values in the same order as they appear as output arguments. The function status return value is always specified as the last return value.

Import HDF4 Files Using Low-Level Functions

This example shows how to read data from a Scientific Data Set in an HDF4 file, using the functions in the `matlat.io.hdf4.sd` package. In HDF4 terminology, the numeric arrays stored in HDF4 files are called data sets.

Add Package to Import List

Add the `matlab.io.hdf4.*` path to the import list.

```
import matlab.io.hdf4.*
```

Subsequent calls to functions in the `matlat.io.hdf4.sd` package need only be prefixed with `sd`, rather than the entire package path.

Open HDF4 File

Open the example HDF4 file, `sd.hdf`, and specify read access, using the `matlab.io.hdf4.sd.start` function. This function corresponds to the SD API routine, `SDstart`.

```
sdID = sd.start('sd.hdf','read');
```

`sd.start` returns an HDF4 SD file identifier, `sdID`.

Get Information About HDF4 File

Get the number of data sets and global attributes in the file, using the `matlab.io.hdf4.sd.fileInfo` function. This function corresponds to the SD API routine, `SDfileinfo`.

```
[ndatasets,ngatts] = sd.fileInfo(sdID)
```

```
ndatasets = 4
```

```
ngatts = 1
```

The file, `sd.hdf`, contains four data sets and one global attribute,

Get Attributes from HDF4 File

Get the contents of the first global attribute. HDF4 uses zero-based indexing, so an index value of 0 specifies the first index.

HDF4 files can optionally include information, called *attributes*, that describes the data that the file contains. Attributes associated with an entire HDF4 file are *global* attributes. Attributes associated with a data set are *local* attributes.

```
attr = sd.readAttr(sdID,0)
```

```
attr =
'02-Sep-2010 11:13:16'
```

Select Data Sets to Import

Determine the index number of the data set named `temperature`. Then, get the identifier of that data set.

```
idx = sd.nameToIndex(sdID,'temperature');
sdsID = sd.select(sdID,idx);
```

`sd.select` returns an HDF4 SD data set identifier, `sdsID`.

Get Information About Data Set

Get information about the data set identified by `sdsID` using the `matlab.io.hdf4.sd.getInfo` function. This function corresponds to the SD API routine, `SDgetinfo`.

```
[name,dims,datatype,nattrs] = sd.getInfo(sdsID)
```

```
name = 
'temperature'

dims = 1×2

    20    10


datatype = 
'double'

nattrs = 11
```

`sd.getInfo` returns information about the name, size, data type, and number of attributes of the data set.

Read Entire Data Set

Read the entire contents of the data set specified by the data set identifier, `sdsID`.

```
data = sd.readData(sdsID);
```

Read Portion of Data Set

Read a 2-by-4 portion of the data set, starting from the first column in the second row. Use the `matlab.io.hdf4.sd.readData` function, which corresponds to the SD API routine, `SDreaddata`. The `start` input is a vector of index values specifying the location in the data set where you want to start reading data. The `count` input is a vector specifying the number of elements to read along each data set dimension.

```
start = [0 1];
count = [2 4];
data2 = sd.readData(sdsID,start,count)
```

```
data2 = 2×4

    21    41    61    81
    22    42    62    82
```

Close HDF4 Data Set

Close access to the data set, using the `matlab.io.hdf4.sd.endAccess` function. This function corresponds to the SD API routine, `SDendaccess`. You must close access to all the data sets in and HDF4 file before closing the file.

```
sd.endAccess(sdsID)
```

Close HDF4 File

Close the HDF4 file using the `matlab.io.hdf4.sd.close` function. This function corresponds to the SD API routine, `SDend`.

```
sd.close(sdID)
```

See Also

`sd.close` | `sd.endAccess` | `sd.fileInfo` | `sd.getInfo` | `sd.readData` | `sd.start`

More About

- “Map HDF4 to MATLAB Syntax” on page 7-46

About HDF4 and HDF-EOS

Hierarchical Data Format (HDF4) is a general-purpose, machine-independent standard for storing scientific data in files, developed by the National Center for Supercomputing Applications (NCSA). For more information about these file formats, read the HDF documentation at the HDF Web site (`www.hdfgroup.org`).

HDF-EOS is an extension of HDF4 that was developed by the National Aeronautics and Space Administration (NASA) for storage of data returned from the Earth Observing System (EOS). For more information about this extension to HDF4, see the HDF-EOS documentation at the NASA Web site (`www.hdfeos.org`).

HDF4 Application Programming Interfaces (APIs) are libraries of C routines. To import or export data, you must use the functions in the HDF4 API associated with the particular HDF4 data type you are working with. Each API has a particular programming model, that is, a prescribed way to use the routines to write data sets to the file. MATLAB functions allow you to access specific HDF4 APIs.

To use the MATLAB HDF4 functions effectively, you must be familiar with the HDF library. For detailed information about HDF4 features and routines, refer to the documentation at the HDF Web site.

Export to HDF4 Files

In this section...

Write MATLAB Data to HDF4 File

This example shows how to write MATLAB® arrays to a Scientific Data Set in an HDF4 file.

Add Package to Import List

Add the `matlab.io.hdf4.*` path to the import list.

```
import matlab.io.hdf4.*
```

Prefix subsequent calls to functions in the `matlat.io.hdf4.sd` package with `sd`, rather than the entire package path.

Create HDF4 File

Create a new HDF4 file using the `matlab.io.hdf4.sd.start` function. This function corresponds to the SD API routine, `SDstart`.

```
sdID = sd.start('mydata.hdf','create');
```

`sd.start` creates the file and returns a file identifier named `sdID`.

To open an existing file instead of creating a new one, call `sd.start` with `'write'` access instead of `'create'`.

Create HDF4 Data Set

Create a data set in the file for each MATLAB array you want to export. If you are writing to an existing data set, you can skip ahead to the next step. In this example, create one data set for the array of sample data, `A`, using the `matlab.io.hdf4.sd.create` function. This function corresponds to the SD API routine, `SDcreate`. The `ds_type` argument is a character vector specifying the MATLAB data type of the data set.

```
A = [1 2 3 4 5 ; 6 7 8 9 10 ; 11 12 13 14 15];
ds_name = 'A';
ds_type = 'double';
ds_dims = size(A);
sdsID = sd.create(sdID,ds_name,ds_type,ds_dims);
```

`sd.create` returns an HDF4 SD data set identifier, `sdsID`.

Write MATLAB Data to HDF4 File

Write data in `A` to the data set in the file using the `matlab.io.hdf4.sd.writedata` function. This function corresponds to the SD API routine, `SDwritedata`. The `start` argument specifies the zero-based starting index.

```
start = [0 0];
sd.writeData(sdsID,start,A);
```

sd.writeData queues the write operation. Queued operations execute when you close the HDF4 file.

Write MATLAB Data to Portion of Data Set

Replace the second row of the data set with the vector B. Use a start input value of [1 0] to begin writing at the second row, first column. start uses zero-based indexing.

```
B = [9 9 9 9 9];
start = [1 0];
sd.writeData(sdsID,start,B);
```

Write Metadata to HDF4 File

Create a global attribute named creation_date, with a value that is the current date and time. Use the matlab.io.hdf4.sd.setAttr function, which corresponds to the SD API routine, SDsetattr.

```
sd.setAttr(sdID,'creation_date',datestr(now));
```

sd.Attr creates a file attribute, also called a global attribute, associated with the HDF4 file identified by sdID.

Associate a predefined attribute, cordsys, to the data set identified by sdsID. Possible values of this attribute include the text strings 'cartesian', 'polar', and 'spherical'.

```
attr_name = 'cordsys';
attr_value = 'polar';
sd.setAttr(sdsID,attr_name,attr_value);
```

Close HDF4 Data Set

Close access to the data set, using the matlab.io.hdf4.sd.endAccess function. This function corresponds to the SD API routine, SDendaccess. You must close access to all the data sets in and HDF4 file before closing the file.

```
sd.endAccess(sdsID);
```

Close HDF4 File

Close the HDF4 file using the matlab.io.hdf4.sd.close function. This function corresponds to the SD API routine, SDend.

```
sd.close(sdID);
```

Closing an HDF4 file executes all the write operations that have been queued using SDwritedata.

Manage HDF4 Identifiers

MATLAB supports utility functions that make it easier to use HDF4 in the MATLAB environment.

- “View All Open HDF4 Identifiers” on page 7-53
- “Close All Open HDF4 Identifiers” on page 7-53

View All Open HDF4 Identifiers

Use the gateway function to the MATLAB HDF4 utility API, `hdfml`, and specify the name of the `listinfo` routine as an argument to view all the currently open HDF4 identifiers. MATLAB updates this list whenever HDF identifiers are created or closed. In this example only two identifiers are open.

```
hdfml('listinfo')
```

```
No open RI identifiers
No open GR identifiers
No open grid identifiers
No open grid file identifiers
No open annotation identifiers
No open AN identifiers
Open scientific dataset identifiers:
    262144
Open scientific data file identifiers:
    393216
No open Vdata identifiers
No open Vgroup identifiers
No open Vfile identifiers
No open point identifiers
No open point file identifiers
No open swath identifiers
No open swath file identifiers
No open access identifiers
No open file identifiers
```

Close All Open HDF4 Identifiers

Close all the currently open HDF4 identifiers in a single call using the gateway function to the MATLAB HDF4 utility API, `hdfml`. Specify the name of the `closeall` routine as an argument:

```
hdfml('closeall')
```

See Also

`hdfml` | `sd.close` | `sd.create` | `sd.endAccess` | `sd.setAttr` | `sd.start` | `sd.writeData`

More About

- “Map HDF4 to MATLAB Syntax” on page 7-46

Audio and Video

Read and Write Audio Files

Write data to an audio file, get information about the file, and then read the data back into the MATLAB workspace.

Write to Audio File

Load sample data from the file, `handel.mat`

```
load handel.mat
```

The workspace now contains a matrix of audio data, `y`, and a sample rate, `Fs`.

Use the `audiowrite` function to write the data to a WAVE file named `handel.wav` in the current folder.

```
audiowrite('handel.wav',y,Fs)
clear y Fs
```

The `audiowrite` function also can write to other audio file formats such as OGG, FLAC, and MPEG-4 AAC.

Get Information About Audio File

Use the `audioinfo` function to get information about the WAVE file, `handel.wav`.

```
info = audioinfo('handel.wav')
```

```
info = 
             Filename: 'pwd\handel.wav'
    CompressionMethod: 'Uncompressed'
          NumChannels: 1
           SampleRate: 8192
         TotalSamples: 73113
             Duration: 8.9249
                Title: []
              Comment: []
               Artist: []
        BitsPerSample: 16
```

`audioinfo` returns a 1-by-1 structure array. The `SampleRate` field indicates the sample rate of the audio data, in hertz. The `Duration` field indicates the duration of the file, in seconds.

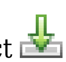
Read Audio File

Use the `audioread` function to read the file, `handel.wav`. The `audioread` function can support WAVE, OGG, FLAC, AU, MP3, and MPEG-4 AAC files.

```
[y,Fs] = audioread('handel.wav');
```

Play the audio.

```
sound(y,Fs)
```

You also can read WAV, AU, or SND files interactively. Select  **Import Data** or double-click the file name in the Current Folder browser.

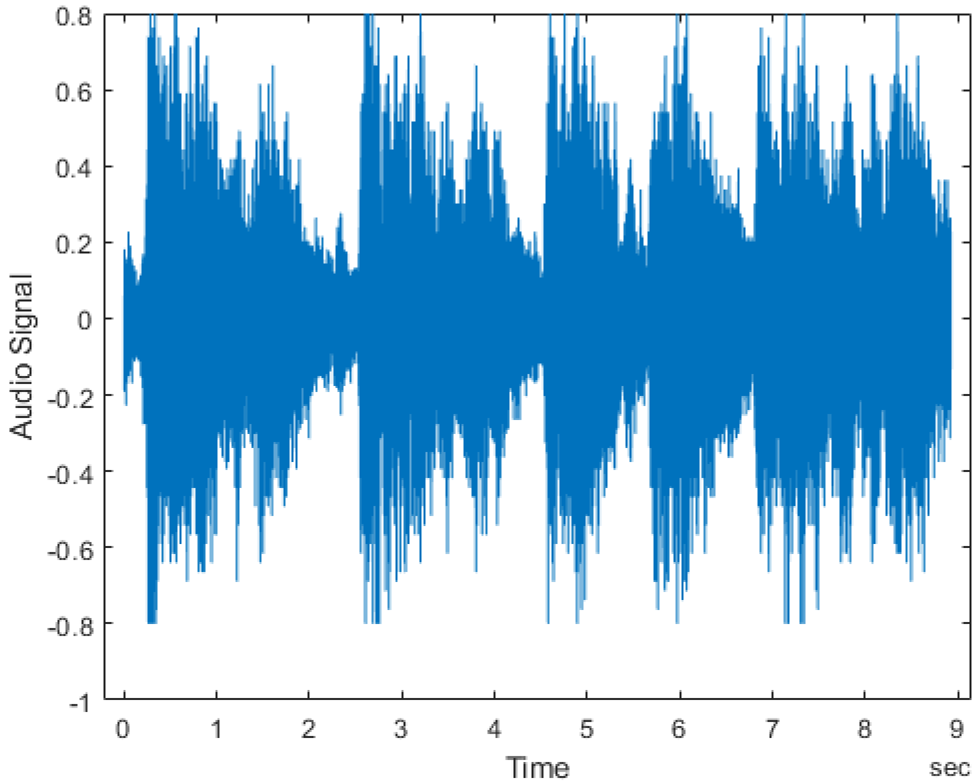
Plot Audio Data

Create a vector t the same length as y, that represents elapsed time.

```
t = 0:seconds(1/Fs):seconds(info.Duration);
t = t(1:end-1);
```

Plot the audio data as a function of time.

```
plot(t,y)
xlabel('Time')
ylabel('Audio Signal')
```



See Also

audioinfo | audioread | audiowrite

Related Examples

- "Import Images, Audio, and Video Interactively" on page 1-7

Record and Play Audio

Record and play audio data for processing in MATLAB from audio input and output devices on your system.

In this section...

Record Audio

Record data from an audio input device such as a microphone connected to your system:

1. Create an `audiorecorder` object.
2. Call the `record` or `recordblocking` method, where:
   - `record` returns immediate control to the calling function or the command prompt even as recording proceeds. Specify the length of the recording in seconds, or end the recording with the `stop` method. Optionally, call the `pause` and `resume` methods. The recording is performed asynchronously.
   - `recordblocking` retains control until the recording is complete. Specify the length of the recording in seconds. The recording is performed synchronously.
3. Create a numeric array corresponding to the signal data using the `getaudiodata` method.

The following examples show how to use the `recordblocking` and `record` methods.

Record Microphone Input

This example shows how to record microphone input, play back the recording, and store the recorded audio signal in a numeric array. You must first connect a microphone to your system.

Create an `audiorecorder` object named `recObj` for recording audio input.

```
recObj = audiorecorder
```

```
recObj = 

  audiorecorder with properties:

          SampleRate: 8000
       BitsPerSample: 8
         NumChannels: 1
            DeviceID: -1
       CurrentSample: 1
        TotalSamples: 0
             Running: 'off'
            StartFcn: []
             StopFcn: []
            TimerFcn: []
         TimerPeriod: 0.0500
                 Tag: ''
```

```
          UserData: []
              Type: 'audiorecorder'
```

`audiorecorder` creates an 8000 Hz, 8-bit, 1-channel `audiorecorder` object.

Record your voice for 5 seconds.

```
disp('Start speaking.')
recordblocking(recObj, 5);
disp('End of Recording.');
```

Play back the recording.

```
play(recObj);
```

Store data in double-precision array, `y`.

```
y = getaudiodata(recObj);
```

Plot the audio samples.

```
plot(y);
```

Record Two Channels from Different Sound Cards

To record audio independently from two different sound cards, with a microphone connected to each:

1. Call `audiodevinfo` to list the available sounds cards. For example, this code returns a structure array containing all input and output audio devices on your system:

   ```
   info = audiodevinfo;
   ```

   Identify the sound cards you want to use by name, and note their `ID` values.
2. Create two `audiorecorder` objects. For example, this code creates the `audiorecorder` object, `recorder1`, for recording a single channel from device 3 at 44.1 kHz and 16 bits per sample. The `audiorecorder` object, `recorder2`, is for recording a single channel from device 4 at 48 kHz:

   ```
   recorder1 = audiorecorder(44100,16,1,3);
   recorder2 = audiorecorder(48000,16,1,4);
   ```
3. Record each audio channel separately.

   ```
   record(recorder1);
   record(recorder2);
   pause(5);
   ```

   The recordings occur simultaneously as the first call to `record` does not block.
4. Stop the recordings.

   ```
   stop(recorder1);
   stop(recorder2);
   ```

Specify the Quality of the Recording

By default, an `audiorecorder` object uses a sample rate of 8000 hertz, a depth of 8 bits (8 bits per sample), and a single audio channel. These settings minimize the required amount of data storage. For higher quality recordings, increase the sample rate or bit depth.

For example, typical compact disks use a sample rate of 44,100 hertz and a 16-bit depth. Create an `audiorecorder` object to record in stereo (two channels) with those settings:

```
myRecObj = audiorecorder(44100, 16, 2);
```

For more information on the available properties and values, see the `audiorecorder` reference page.

Play Audio

After you import or record audio, MATLAB supports several ways to listen to the data:

- For simple playback using a single function call, use `sound` or `soundsc`. For example, load a sample MAT-file that contains signal and sample rate data, and listen to the audio:

  ```
  load chirp.mat;
  sound(y, Fs);
  ```
- For more flexibility during playback, including the ability to pause, resume, or define callbacks, use the `audioplayer` function. Create an `audioplayer` object, then call methods to play the audio. For example, listen to the `gong` sample file:

  ```
  load gong.mat;
  gong = audioplayer(y, Fs);
  play(gong);
  ```

  For an additional example, see “Record or Play Audio within a Function” on page 8-6.

If you do not specify the sample rate, `sound` plays back at 8192 hertz. For any playback, specify smaller sample rates to play back more slowly, and larger sample rates to play back more quickly.

Note Most sound cards support sample rates between approximately 5,000 and 48,000 hertz. Specifying sample rates outside this range can produce unexpected results.

Record or Play Audio within a Function

If you create an `audioplayer` or `audiorecorder` object inside a function, the object exists only for the duration of the function. For example, create a player function called `playFile` and a simple callback function `showSeconds`:

```
function playFile(myfile)
   load(myfile);

   obj = audioplayer(y, Fs);
   obj.TimerFcn = 'showSeconds';
   obj.TimerPeriod = 1;

   play(obj);
end

function showSeconds
   disp('tick')
end
```

Call `playFile` from the command prompt to play the file `handel.mat`:

```
playFile('handel.mat')
```

At the recorded sample rate of 8192 samples per second, playing the 73113 samples in the file takes approximately 8.9 seconds. However, the `playFile` function typically ends before playback completes, and clears the `audioplayer` object `obj`.

To ensure complete playback or recording, consider the following options:

- Use `playblocking` or `recordblocking` instead of `play` or `record`. The blocking methods retain control until playing or recording completes. If you block control, you cannot issue any other commands or methods (such as `pause` or `resume`) during the playback or recording.
- Create an output argument for your function that generates an object in the base workspace. For example, modify the `playFile` function to include an output argument:

  ```
  function obj = playFile(myfile)
  ```

  Call the function:

  ```
  h = playFile('handel.mat');
  ```

  Because `h` exists in the base workspace, you can pause playback from the command prompt:

  ```
  pause(h)
  ```

See Also

`audioplayer` | `audiorecorder` | `sound` | `soundsc`

More About

- "Read and Write Audio Files" on page 8-2

Read Video Files

Read frames from a video starting at a specific time or frame index, read frames within a specified interval, or read all the frames in the video.

Read Frames Beginning at Specified Time or Frame Index

Read part of a video file starting 0.5 second from the beginning of the file. Then, read the video starting from frame index 100 to the end of the video file.

Construct a `VideoReader` object associated with the sample file `'xylophone.mp4'`.

```
vidObj = VideoReader('xylophone.mp4');
```

Specify that reading should begin 0.5 second from the beginning of the file by setting the `CurrentTime` property.

```
vidObj.CurrentTime = 0.5;
```

Read video frames until the end of the file is reached by using the `readFrame` method.

```
while hasFrame(vidObj)
    vidFrame = readFrame(vidObj);
    imshow(vidFrame)
    pause(1/vidObj.FrameRate);
end
```

Alternatively, you can read frames from a video starting at a specified frame index to the end of the video by using the `read` method. Specify the indices to read as `[100 Inf]`. The `read` method returns all the frames starting at 100 to the end of the video file.

```
vidframes = read(vidObj,[100 Inf]);
```

Read Frames Within Specified Interval

Read a part of a video file by specifying the time or frame interval.

Read the video frames between 0.6 and 0.9 seconds. First, create a video reader object and a structure array to hold the frames.

```
vidObj = VideoReader('xylophone.mp4');
s = struct('cdata',zeros(vidObj.Height,vidObj.Width,3,'uint8'),'colormap',[]);
```

Then, specify that reading should begin 0.6 second from the beginning of the file by setting the `CurrentTime` property.

```
vidObj.CurrentTime = 0.6;
```

Read one frame at a time until `CurrentTime` reaches 0.9 second. Append data from each video frame to the structure array. View the number of frames in the structure array. `s` is a 1-by-10 structure indicating that 10 frames were read. For information on displaying the frames in the structure `s` as a movie, see the `movie` function reference page.

```
k = 1;
while vidObj.CurrentTime <= 0.9
    s(k).cdata = readFrame(vidObj);
    k = k+1;
end
whos s
```

```
  Name      Size               Bytes  Class     Attributes

  s         1x10             2305344  struct
```

Alternatively, you can read all the frames in a specified interval by using frame indices. For example, specify the second argument of `read` as `[18 27]`. The `read` method returns a `FrameSize`-by-10 array indicating that 10 frames were read.

```
frames = read(vidObj,[18 27]);
whos frames
```

```
  Name          Size                    Bytes  Class    Attributes

  frames       240x320x3x10            2304000  uint8
```

Read All Frames

Read all the frames from video, one frame at a time or all the frames at once.

Create a video reader object and display the total number of frames in the video.

```
vidObj = VideoReader('xylophone.mp4');
vidObj.NumFrames
```

```
ans = 141
```

Read all the frames, one frame at a time, by using the `readFrame` method, and display the frames.

```
while hasFrame(vidObj)
   frame = readFrame(vidObj);
   imshow(frame)
```

```
    pause(1/vidObj.FrameRate);
end
```

Alternatively, you can read all the video frames at once. The `read` method returns a `FrameSize`-by-141 array of video frames.

```
allFrames = read(vidObj);
whos allFrames
```

```
  Name                Size                    Bytes  Class    Attributes

  allFrames       240x320x3x141            32486400  uint8
```

Troubleshooting and Tips For Video Reading

- The `hasFrame` method might return logical `1` (true) when the value of the `CurrentTime` property is equal to the value of the `Duration` property. This is due to a limitation in the underlying APIs used.
- Seeking to the last frame in a video file by setting the `CurrentTime` property to a value close to the `Duration` value is not recommended. For some files, this operation returns an error indicating that the end-of-file has been reached, even though the `CurrentTime` value is less than the `Duration` value. This typically occurs if the file duration is larger than the duration of the video stream, and there is no video available to read near the end of the file.
- Use of the `Duration` property to limit the reading of data from a video file is not recommended. Use the `hasFrame` method to check whether there is a frame available to read. It is best to read data until the file reports that there are no more frames available to read.
- Video Reading Performance on Windows® Systems: To achieve better video reader performance on Windows for `MP4` and `MOV` files, MATLAB® uses the system's graphics hardware for decoding. However, in some cases using the graphics card for decoding can result in poorer performance depending on the specific graphics hardware on the system. If you notice slower video reader performance on your system, turn off the hardware acceleration by typing:

`matlab.video.read.UseHardwareAcceleration('off')`. You can reenable hardware acceleration by typing: `matlab.video.read.UseHardwareAcceleration('on')`.

See Also

`VideoReader` | `mmfileinfo` | `movie` | `read` | `readFrame`

More About

- "Supported Video and Audio File Formats" on page 8-12

Supported Video and Audio File Formats

Video and audio files in MATLAB and their supported file formats and codecs.

Video Data in MATLAB

What Are Video Files?

For video data, the term “file format” often refers to either the *container format* or the *codec*. A container format describes the layout of the file, while a codec describes how to encode/decode the video data. Many container formats can hold data encoded with different codecs.

To read a video file, any application must:

- Recognize the container format (such as AVI).
- Have access to the codec that can decode the video data stored in the file. Some codecs are part of standard Windows and Macintosh system installations, and allow you to play video in Windows Media Player or QuickTime. In MATLAB, `VideoReader` can access most, but not all, of these codecs.
- Properly use the codec to decode the video data in the file. `VideoReader` cannot always read files associated with codecs that were not part of your original system installation.

Formats That VideoReader Supports

Use `VideoReader` to read video files in MATLAB. The file formats that `VideoReader` supports vary by platform, and have no restrictions on file extensions.

Platforms	File Formats
All Platforms	AVI, including uncompressed, indexed, grayscale, and Motion JPEG-encoded video (`.avi`) Motion JPEG 2000 (`.mj2`)
All Windows	MPEG-1 (`.mpg`) Windows Media Video (`.wmv`, `.asf`, `.asx`) Any format supported by Microsoft DirectShow
Windows 7 or later	MPEG-4, including H.264 encoded video (`.mp4`, `.m4v`) Apple QuickTime Movie (`.mov`) Any format supported by Microsoft Media Foundation

Platforms	File Formats
Macintosh	Most formats supported by QuickTime Player, including: MPEG-1 (`.mpg`) MPEG-4, including H.264 encoded video (`.mp4`, `.m4v`) Apple QuickTime Movie (`.mov`) 3GPP 3GPP2 AVCHD DV **Note:** For OS X Yosemite (Version 10.10) and later, MPEG-4/H.264 files written using `VideoWriter`, play correctly, but display an inexact frame rate.
Linux	Any format supported by your installed plug-ins for GStreamer 1.0 or higher, as listed on https://gstreamer.freedesktop.org/documentation/plugins_doc.html, including Ogg Theora (`.ogg`).

View Codec Associated with Video File

This example shows how to view the codec associated with a video file, using the `mmfileinfo` function.

Store information about the sample video file, `shuttle.avi`, in a structure array named `info`. The `info` structure contains the following fields: `Filename`, `Path`, `Duration`, `Audio` and `Video`.

```
info = mmfileinfo('shuttle.avi');
```

Show the properties in the command window by displaying the fields of the `info` structure. For example, to view information under the `Video` field, type `info.Video`

```
info.Video
```

```
ans = struct with fields:
    Format: 'MJPG'
    Height: 288
     Width: 512
```

The file, `shuttle.avi`, uses the Motion JPEG codec.

Troubleshooting: Errors Reading Video File

You might be unable to read a video file if MATLAB cannot access the appropriate codec. 64-bit applications use 64-bit codec libraries, while 32-bit applications use 32-bit codec libraries. For example, when working with 64-bit MATLAB, you cannot read video files that require access to a 32-bit codec installed on your system. To read these files, try one of the following:

- Install a 64-bit codec that supports this file format. Then, try reading the file using 64-bit MATLAB.

- Re-encode the file into a different format with a 64-bit codec that is installed on your computer.

Sometimes, `VideoReader` cannot open a video file for reading on Windows platforms. This might occur if you have installed a third-party codec that overrides your system settings. Uninstall the codec and try opening the video file in MATLAB again.

Audio Data in MATLAB

What Are Audio Files?

The audio signal in a file represents a series of *samples* that capture the amplitude of the sound over time. The *sample rate* is the number of discrete samples taken per second and given in hertz. The precision of the samples, measured by the *bit depth* (number of bits per sample), depends on the available audio hardware.

MATLAB audio functions read and store single-channel (mono) audio data in an *m*-by-1 column vector, and stereo data in an *m*-by-2 matrix. In either case, *m* is the number of samples. For stereo data, the first column contains the left channel, and the second column contains the right channel.

Typically, each sample is a double-precision value between -1 and 1. In some cases, particularly when the audio hardware does not support high bit depths, audio files store the values as 8-bit or 16-bit integers. The range of the sample values depends on the available number of bits. For example, samples stored as `uint8` values can range from 0 to 255 (2^8 – 1). The MATLAB `sound` and `soundsc` functions support only single- or double-precision values between -1 and 1. Other audio functions support multiple data types, as indicated on the function reference pages.

Formats That audioReader Supports

Use `audioread` to read audio files in MATLAB. The `audioread` function supports these file formats.

Platform Support	File Format
All platforms	WAVE (`.wav`)
	OGG (`.ogg`)
	FLAC (`.flac`)
	AU (`.au`)
	AIFF (`.aiff`, `.aif`)
	AIFC (`.aifc`)
Windows 7 (or later), Macintosh, and Linux	MP3 (`.mp3`)
	MPEG-4 AAC (`.m4a`, `.mp4`)

On Windows platforms prior to Windows 7, `audioread` does not read WAVE files with MP3 encoded data.

On Windows 7 (or later) platforms, `audioread` might also read any files supported by Windows Media Foundation.

On Linux platforms, `audioread` might also read any files supported by GStreamer.

`audioread` can extract audio from MPEG-4 (`.mp4`, `.m4v`) video files on Windows 7 or later, Macintosh, and Linux, and from Windows Media Video (`.wmv`) and AVI (`.avi`) files on Windows 7 (or later) and Linux platforms.

See Also

`VideoReader` | `audioinfo` | `audioread` | `mmfileinfo`

More About

- “Read Video Files” on page 8-8
- “Read and Write Audio Files” on page 8-2

Convert Between Image Sequences and Video

Convert between video files and sequences of image files using `VideoReader` and `VideoWriter`.

The sample file named `shuttle.avi` contains 121 frames. Convert the frames to image files using `VideoReader` and the `imwrite` function. Then, convert the image files to an AVI file using `VideoWriter`.

Setup

Create a temporary working folder to store the image sequence.

```
workingDir = tempname;
mkdir(workingDir)
mkdir(workingDir,'images')
```

Create VideoReader

Create a `VideoReader` to use for reading frames from the file.

```
shuttleVideo = VideoReader('shuttle.avi');
```

Create the Image Sequence

Loop through the video, reading each frame into a width-by-height-by-3 array named `img`. Write out each image to a JPEG file with a name in the form `imgN.jpg`, where N is the frame number.

| img001.jpg|

| img002.jpg|

| ...|

| img121.jpg|

```
ii = 1;

while hasFrame(shuttleVideo)
   img = readFrame(shuttleVideo);
   filename = [sprintf('%03d',ii) '.jpg'];
   fullname = fullfile(workingDir,'images',filename);
   imwrite(img,fullname)    % Write out to a JPEG file (img1.jpg, img2.jpg, etc.)
   ii = ii+1;
end
```

Find Image File Names

Find all the JPEG file names in the `images` folder. Convert the set of image names to a cell array.

```
imageNames = dir(fullfile(workingDir,'images','*.jpg'));
imageNames = {imageNames.name}';
```

Create New Video with the Image Sequence

Construct a `VideoWriter` object, which creates a Motion-JPEG AVI file by default.

```
outputVideo = VideoWriter(fullfile(workingDir,'shuttle_out.avi'));
outputVideo.FrameRate = shuttleVideo.FrameRate;
open(outputVideo)
```

Loop through the image sequence, load each image, and then write it to the video.

```
for ii = 1:length(imageNames)
   img = imread(fullfile(workingDir,'images',imageNames{ii}));
   writeVideo(outputVideo,img)
end
```

Finalize the video file.

```
close(outputVideo)
```

View the Final Video

Construct a reader object.

```
shuttleAvi = VideoReader(fullfile(workingDir,'shuttle_out.avi'));
```

Create a MATLAB movie struct from the video frames.

```
ii = 1;
while hasFrame(shuttleAvi)
   mov(ii) = im2frame(readFrame(shuttleAvi));
   ii = ii+1;
end
```

Resize the current figure and axes based on the video's width and height, and view the first frame of the movie.

```
figure
imshow(mov(1).cdata, 'Border', 'tight')
```

Play back the movie once at the video's frame rate.

```
movie(mov,1,shuttleAvi.FrameRate)
```

Credits

Video of the Space Shuttle courtesy of NASA.

XML Documents

Importing XML Documents

To read an XML file from your local disk or from a URL, use the `xmlread` function. `xmlread` returns the contents of the file in a Document Object Model (DOM) node. For more information, see:

- "What Is an XML Document Object Model (DOM)?" on page 9-2
- "Example — Finding Text in an XML File" on page 9-3

What Is an XML Document Object Model (DOM)?

In a Document Object Model, every item in an XML file corresponds to a node. The properties and methods for DOM nodes (that is, the way you create and access nodes) follow standards set by the World Wide Web consortium.

For example, consider this sample XML file:

```
<productinfo>

<!-- This is a sample info.xml file. -->

<list>

<listitem>
<label color="blue">Import Wizard</label>
<callback>uiimport</callback>
<icon>ApplicationIcon.GENERIC_GUI</icon>
</listitem>

<listitem>
<label color="red">Profiler</label>
<callback>profile viewer</callback>
<icon>ApplicationIcon.PROFILER</icon>
</listitem>

</list>
</productinfo>
```

The information in the file maps to the following types of nodes in a DOM:

- *Element nodes* — Corresponds to tag names. In the sample `info.xml` file, these tags correspond to element nodes:
  - `productinfo`
  - `list`
  - `listitem`
  - `label`
  - `callback`
  - `icon`

  In this case, the `list` element is the parent of `listitem` element child nodes. The `productinfo` element is the root element node.
- Text nodes — Contains values associated with element nodes. Every text node is the child of an element node. For example, the `Import Wizard` text node is the child of the first `label` element node.

- Attribute nodes — Contains name and value pairs associated with an element node. For example, in the first `label` element node, `color` is the name of an attribute and `blue` is its value. Attribute nodes are not parents or children of any nodes.
- Comment nodes — Includes additional text in the file, in the form `<!--Sample comment-->`.
- Document nodes — Corresponds to the entire file. Use methods on the document node to create new element, text, attribute, or comment nodes.

For a complete list of the methods and properties of DOM nodes, see the `org.w3c.dom` package description at `https://docs.oracle.com/javase/7/docs/api`.

Example — Finding Text in an XML File

The full *matlabroot*/toolbox/matlab/general/info.xml file contains several `listitem` elements, such as:

```
<listitem>
<label>Import Wizard</label>
<callback>uiimport</callback>
<icon>ApplicationIcon.GENERIC_GUI</icon>
</listitem>
```

One of the `label` elements has the child text `Plot Tools`. Suppose that you want to find the text for the `callback` element in the same `listitem`. Follow these steps:

1 Initialize your variables, and call `xmlread` to obtain the document node:

```
findLabel = 'Plot Tools';
findCbk = '';

xDoc = xmlread(fullfile(matlabroot, ...
               'toolbox','matlab','general','info.xml'));
```

2 Find all the `listitem` elements. The `getElementsByTagName` method returns a deep list that contains information about the child nodes:

```
allListitems = xDoc.getElementsByTagName('listitem');
```

Note Lists returned by DOM methods use zero-based indexing.

3 For each `listitem`, compare the text for the `label` element to the text you want to find. When you locate the correct `label`, get the `callback` text:

```
for k = 0:allListitems.getLength-1
   thisListitem = allListitems.item(k);

   % Get the label element. In this file, each
   % listitem contains only one label.
   thisList = thisListitem.getElementsByTagName('label');
   thisElement = thisList.item(0);

   % Check whether this is the label you want.
   % The text is in the first child node.
   if strcmp(thisElement.getFirstChild.getData, findLabel)
       thisList = thisListitem.getElementsByTagName('callback');
       thisElement = thisList.item(0);
       findCbk = char(thisElement.getFirstChild.getData);
```

```
            break;
        end

    end
```

4 Display the final results:

```
if ~isempty(findCbk)
    msg = sprintf('Item "%s" has a callback of "%s."',...
                  findLabel, findCbk);
else
    msg = sprintf('Did not find the "%s" item.', findLabel);
end
disp(msg);
```

For an additional example that creates a structure array to store data from an XML file, see the `xmlread` function reference page.

Exporting to XML Documents

To write data to an XML file, use the `xmlwrite` function. `xmlwrite` requires that you describe the file in a Document Object Model (DOM) node. For an introduction to DOM nodes, see "What Is an XML Document Object Model (DOM)?" on page 9-2

For more information, see:

- "Creating an XML File" on page 9-5
- "Updating an Existing XML File" on page 9-6

Creating an XML File

Although each file is different, these are common steps for creating an XML document:

1 Create a document node and define the root element by calling this method:

```
docNode = com.mathworks.xml.XMLUtils.createDocument('root_element');
```

2 Get the node corresponding to the root element by calling `getDocumentElement`. The root element node is required for adding child nodes.

3 Add element, text, comment, and attribute nodes by calling methods on the document node. Useful methods include:

- `createElement`
- `createTextNode`
- `createComment`
- `setAttribute`

For a complete list of the methods and properties of DOM nodes, see the `org.w3c.dom` package description at `https://docs.oracle.com/javase/7/docs/api`.

4 As needed, define parent/child relationships by calling `appendChild` on the parent node.

Tip Text nodes are always children of element nodes. To add a text node, call `createTextNode` on the document node, and then call `appendChild` on the parent element node.

Example — Creating an XML File with xmlwrite

Suppose that you want to create an `info.xml` file for the Upslope Area Toolbox (described in "Display Custom Documentation"), as follows:

```
<?xml version="1.0" encoding="utf-8"?>
<toc version="2.0">
   <tocitem target="upslope_product_page.html">Upslope Area Toolbox<!-- Functions -->
      <tocitem target="demFlow_help.html">demFlow</tocitem>
      <tocitem target="facetFlow_help.html">facetFlow</tocitem>
      <tocitem target="flowMatrix_help.html">flowMatrix</tocitem>
      <tocitem target="pixelFlow_help.html">pixelFlow</tocitem>
   </tocitem>
</toc>
```

To create this file using `xmlwrite`, follow these steps:

1 Create the document node and root element, `toc`:

```
docNode = com.mathworks.xml.XMLUtils.createDocument('toc');
```

2 Identify the root element, and set the `version` attribute:

```
toc = docNode.getDocumentElement;
toc.setAttribute('version','2.0');
```

3 Add the `tocitem` element node for the product page. Each `tocitem` element in this file has a `target` attribute and a child text node:

```
product = docNode.createElement('tocitem');
product.setAttribute('target','upslope_product_page.html');
product.appendChild(docNode.createTextNode('Upslope Area Toolbox'));
toc.appendChild(product)
```

4 Add the comment:

```
product.appendChild(docNode.createComment(' Functions '));
```

5 Add a `tocitem` element node for each function, where the `target` is of the form *`function`*`_help.html`:

```
functions = {'demFlow','facetFlow','flowMatrix','pixelFlow'};
for idx = 1:numel(functions)
    curr_node = docNode.createElement('tocitem');

    curr_file = [functions{idx} '_help.html'];
    curr_node.setAttribute('target',curr_file);

    % Child text is the function name.
    curr_node.appendChild(docNode.createTextNode(functions{idx}));
    product.appendChild(curr_node);
end
```

6 Export the DOM node to `info.xml`, and view the file with the `type` function:

```
xmlwrite('info.xml',docNode);
type('info.xml');
```

Updating an Existing XML File

To change data in an existing file, call `xmlread` to import the file into a DOM node. Traverse the node and add or change data using methods defined by the World Wide Web consortium, such as:

- `getElementsByTagName`
- `getFirstChild`
- `getNextSibling`
- `getNodeName`
- `getNodeType`

When the DOM node contains all your changes, call `xmlwrite` to overwrite the file.

For a complete list of the methods and properties of DOM nodes, see the `org.w3c.dom` package description at `https://docs.oracle.com/javase/7/docs/api`.

For examples that use these methods, see:

- “Example — Finding Text in an XML File” on page 9-3
- “Example — Creating an XML File with xmlwrite” on page 9-5
- `xmlread` and `xmlwrite`

Memory-Mapping Data Files

Overview of Memory-Mapping

In this section...

What Is Memory-Mapping?

Memory-mapping is a mechanism that maps a portion of a file, or an entire file, on disk to a range of addresses within an application's address space. The application can then access files on disk in the same way it accesses dynamic memory. This makes file reads and writes faster in comparison with using functions such as `fread` and `fwrite`.

Benefits of Memory-Mapping

The principal benefits of memory-mapping are efficiency, faster file access, the ability to share memory between applications, and more efficient coding.

Faster File Access

Accessing files via memory map is faster than using I/O functions such as `fread` and `fwrite`. Data are read and written using the virtual memory capabilities that are built in to the operating system rather than having to allocate, copy into, and then deallocate data buffers owned by the process.

MATLAB does not access data from the disk when the map is first constructed. It only reads or writes the file on disk when a specified part of the memory map is accessed, and then it only reads that specific part. This provides faster random access to the mapped data.

Efficiency

Mapping a file into memory allows access to data in the file as if that data had been read into an array in the application's address space. Initially, MATLAB only allocates address space for the array; it does not actually read data from the file until you access the mapped region. As a result, memory-mapped files provide a mechanism by which applications can access data segments in an extremely large file without having to read the entire file into memory first.

Efficient Coding Style

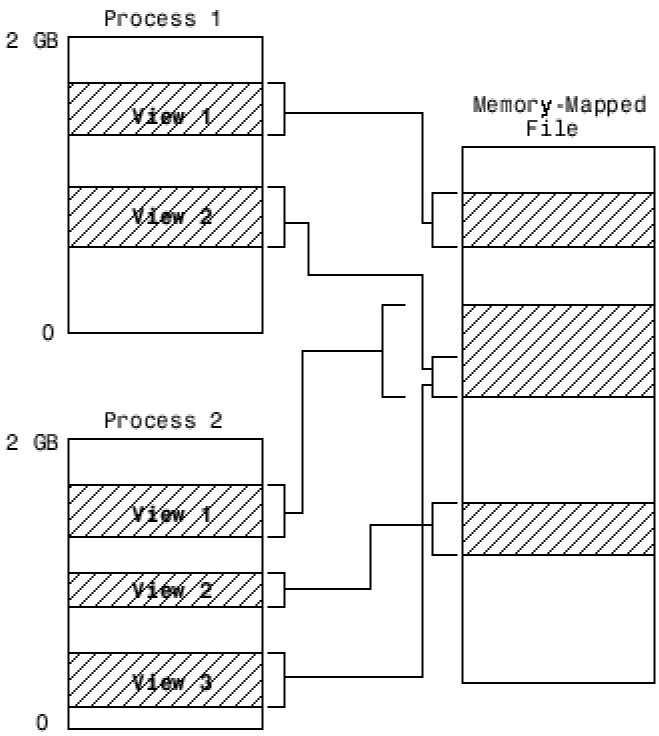
Memory-mapping in your MATLAB application enables you to access file data using standard MATLAB indexing operations. Once you have mapped a file to memory, you can read the contents of that file using the same type of MATLAB statements used to read variables from the MATLAB workspace. The contents of the mapped file appear as if they were an array in the currently active workspace. You simply index into this array to read or write the desired data from the file. Therefore, you do not need explicit calls to the `fread` and `fwrite` functions.

In MATLAB, if `x` is a memory-mapped variable, and `y` is the data to be written to a file, then writing to the file is as simple as

```
x.Data = y;
```

Sharing Memory Between Applications

Memory-mapped files also provide a mechanism for sharing data between applications, as shown in the figure below. This is achieved by having each application map sections of the same file. You can use this feature to transfer large data sets between MATLAB and other applications.



Also, within a single application, you can map the same segment of a file more than once.

When to Use Memory-Mapping

Just how much advantage you get from mapping a file to memory depends mostly on the size and format of the file, the way in which data in the file is used, and the computer platform you are using.

When Memory-Mapping Is Most Useful

Memory-mapping works best with binary files, and in the following scenarios:

- For large files that you want to access randomly one or more times
- For small files that you want to read into memory once and access frequently
- For data that you want to share between applications
- When you want to work with data in a file as if it were a MATLAB array

When the Advantage Is Less Significant

The following types of files do not fully use the benefits of memory-mapping:

- Formatted binary files like HDF or TIFF that require customized readers are not good for memory-mapping. Describing the data contained in these files can be a very complex task. Also, you cannot access data directly from the mapped segment, but must instead create arrays to hold the data.
- Text or ASCII files require that you convert the text in the mapped region to an appropriate type for the data to be meaningful. This takes up additional address space.
- Files that are larger than several hundred megabytes in size consume a significant amount of the virtual address space needed by MATLAB to process your program. Mapping files of this size may result in MATLAB reporting out-of-memory errors more often. This is more likely if MATLAB has been running for some time, or if the memory used by MATLAB becomes fragmented.

Maximum Size of a Memory Map

Due to limits set by the operating system and MATLAB, the maximum amount of data you can map with a single instance of a memory map is 2 gigabytes on 32-bit systems, and 256 terabytes on 64-bit systems. If you need to map more than this limit, you can either create separate maps for different regions of the file, or you can move the window of one map to different locations in the file.

Byte Ordering

Memory-mapping works only with data that have the same byte ordering scheme as the native byte ordering of your operating system. For example, because both Linus Torvalds' Linux and Microsoft Windows systems use little-endian byte ordering, data created on a Linux system can be read on Windows systems. You can use the `computer` function to determine the native byte ordering of your current system.

Map File to Memory

Create a Simple Memory Map

Suppose you want to create a memory map for a file named `records.dat`, using the `memmapfile` function.

Create a sample file named `records.dat`, containing 5000 values.

```
myData = gallery('uniformdata', [5000,1], 0);

fileID = fopen('records.dat','w');
fwrite(fileID, myData,'double');
fclose(fileID);
```

Next, create the memory map. Use the `Format` name-value pair argument to specify that the values are of type `double`. Use the `Writable` name-value pair argument to allow write access to the mapped region.

```
m = memmapfile('records.dat', ...
      'Format', 'double',     ...
      'Writable', true)
```

```
m = 

    Filename: 'd:\matlab\records.dat'
    Writable: true
      Offset: 0
      Format: 'double'
      Repeat: Inf
        Data: 5000x1 double array
```

MATLAB creates a `memmapfile` object, `m`. The `Format` property indicates that read and write operations to the mapped region treat the data in the file as a sequence of double-precision numbers. The `Data` property contains the 5000 values from the file, `records.dat`. You can change the value of any of the properties, except for `Data`, after you create the memory map, `m`.

For example, change the starting position of the memory map, `m`. Begin the mapped region 1024 bytes from the start of the file by changing the value of the `Offset` property.

```
m.Offset = 1024
```

```
m = 

    Filename: 'd:\matlab\records.dat'
    Writable: true
      Offset: 1024
      Format: 'double'
```

```
      Repeat: Inf
        Data: 4872x1 double array
```

Whenever you change the value of a memory map property, MATLAB remaps the file to memory. The Data property now contains only 4872 values.

Specify Format of Your Mapped Data

By default, MATLAB considers all the data in a mapped file to be a sequence of unsigned 8-bit integers. However, your data might be of a different data type. When you call the memmapfile function, use the Format name-value pair argument to indicate another data type. The value of Format can either be a character vector that identifies a single class used throughout the mapped region, or a cell array that specifies more than one class.

Suppose you map a file that is 12 kilobytes in length. Data read from this file can be treated as a sequence of 6,000 16-bit (2-byte) integers, or as 1,500 8-byte double-precision floating-point numbers, to name just a few possibilities. You also could read this data as a combination of different types: for example, as 4,000 8-bit (1-byte) integers followed by 1,000 64-bit (8-byte) integers. You can determine how MATLAB will interpret the mapped data by setting the Format property of the memory map when you call the memmapfile function.

MATLAB arrays are stored on disk in column-major order. The sequence of array elements is column 1, row 1; column 1, row 2; column 1, last row; column 2, row 1, and so on. You might need to transpose or rearrange the order of array elements when reading or writing via a memory map.

Map Multiple Data Types and Arrays

If the region you are mapping comprises segments of varying data types or array shapes, you can specify an individual format for each segment. Specify the value of the Format name-value pair argument as an n-by-3 cell array, where n is the number of segments. Each row in the cell array corresponds to a segment. The first cell in the row identifies the data type to apply to the mapped segment. The second cell contains the array dimensions to apply to the segment. The third cell contains the field name for referencing that segment. For a memory map, m, use the following syntax:

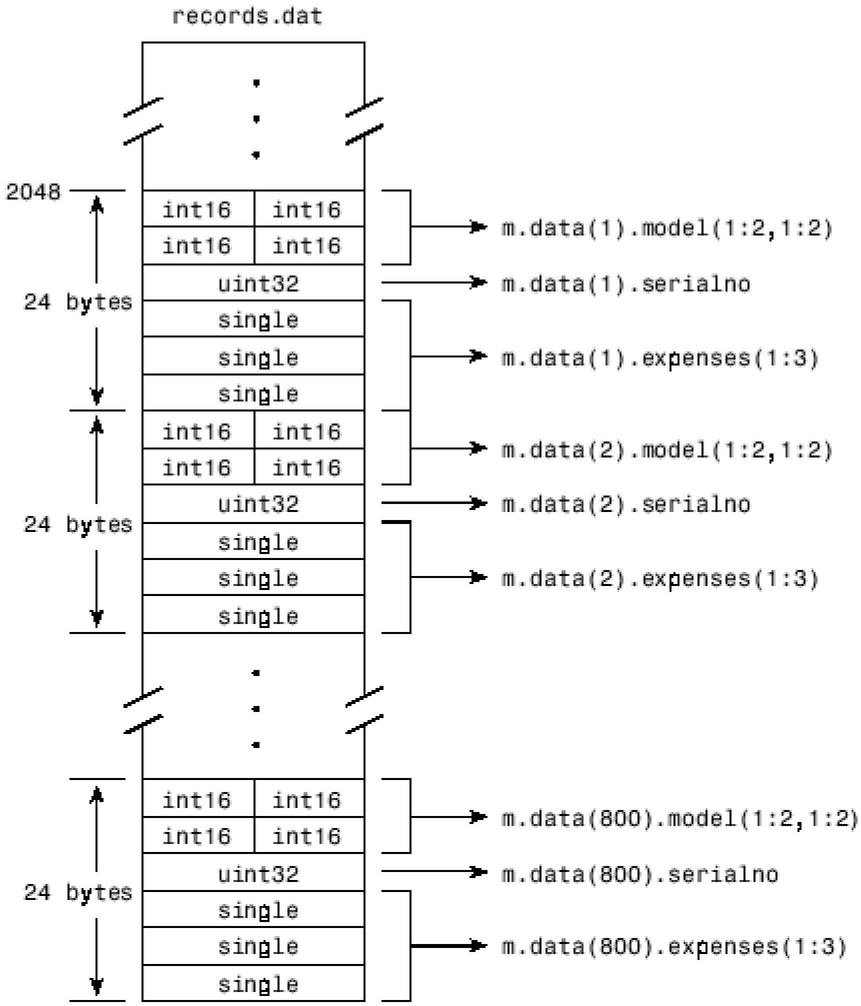
```
m = memmapfile(filename,                                ...
            'Format', {                                 ...
               datatype1, dimensions1, fieldname1;      ...
               datatype2, dimensions2, fieldname2;      ...
                   :           :            :           ...
               datatypeN, dimensionsN, fieldnameN})
```

Suppose you have a file that is 40,000 bytes in length. The following code maps the data beginning at the 2048th byte. The Format value is a 3-by-3 cell array that maps the file data to three different classes: int16, uint32, and single.

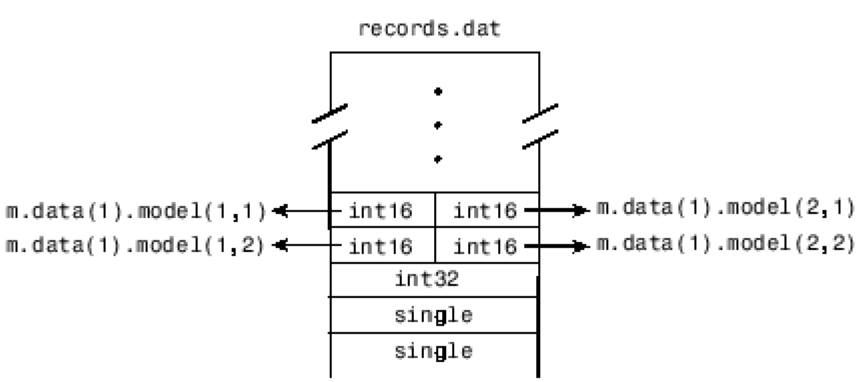
```
m = memmapfile('records.dat',        ...
      'Offset', 2048,                ...
      'Format', {                    ...
         'int16'  [2 2] 'model';     ...
         'uint32' [1 1] 'serialno';  ...
         'single' [1 3] 'expenses'});
```

In this case, memmapfile maps the int16 data as a 2-by-2 matrix that you can access using the field name, model. The uint32 data is a scalar value accessed using the field name, serialno. The `single` data is a 1-by-3 matrix named `expenses`. Each of these fields belongs to the 800-by-1 structure array, `m.Data`.

This figure shows the mapping of the example file.



The next figure shows the ordering of the array elements more closely. In particular, it illustrates that MATLAB arrays are stored on the disk in column-major order. The sequence of array elements in the mapped file is row 1, column 1; row 2, column 1; row 1, column 2; and row 2, column 2.



If the data in your file is not stored in this order, you might need to transpose or rearrange the order of array elements when reading or writing via a memory map.

Select File to Map

You can change the value of the `Filename` property at any time after constructing the `memmapfile` object. You might want to do this if:

- You want to use the same `memmapfile` object on more than one file.
- You save your `memmapfile` object to a MAT-file, and then later load it back into MATLAB in an environment where the mapped file has been moved to a different location. This requires that you modify the path segment of the `Filename` to represent the new location.

Update the path in the `Filename` property for a memory map using dot notation. For example, to specify a new path, `f:\testfiles\records.dat` for a memory map, `m`, type:

```
m.Filename = 'f:\testfiles\records.dat'
```

See Also

`memmapfile`

More About

- "Read from Mapped File" on page 10-9
- "Write to Mapped File" on page 10-14

Read from Mapped File

This example shows how to create two different memory maps, and then read from each of the maps using the appropriate syntax. Then, it shows how to modify map properties and analyze your data.

You can read the contents of a file that you mapped to memory using the same MATLAB® commands you use to read variables from the MATLAB workspace. By accessing the `Data` property of the memory map, the contents of the mapped file appear as an array in the currently active workspace. To read the data you want from the file, simply index into the array. For better performance, copy the `Data` field to a variable, and then read the mapped file using this variable:

```
dataRef = m.Data;

for k = 1 : N

y(k) = dataRef(k);

end
```

By contrast, reading directly from the `memmapfile` object is slower:

```
for k = 1 : N

y(k) = m.Data(k);

end
```

Read from Memory Map as Numeric Array

First, create a sample data file named `records.dat` that contains a 5000-by-1 matrix of double-precision floating-point numbers.

```
randData = gallery('uniformdata',[5000,1],0);

fileID = fopen('records.dat','w');
fwrite(fileID,randData,'double');
fclose(fileID);
```

Map 100 double-precision floating-point numbers from the file to memory, and then read a portion of the mapped data. Create the memory map, `m`. Specify an `Offset` value of 1024 to begin the map 1024 bytes from the start of the file. Specify a `Repeat` value of 100 to map 100 values.

```
m = memmapfile('records.dat','Format','double', ...
      'Offset',1024,'Repeat',100);
```

Copy the `Data` property to a variable, `d`. Then, show the format of `d`.

```
d = m.Data;

whos d
```

```
  Name        Size            Bytes  Class     Attributes

  d         100x1               800  double
```

The mapped data is an 800-byte array because there are 100 `double` values, each requiring 8 bytes.

Read a selected set of numbers from the file by indexing into the vector, `d`.

```
d(15:20)
```

```
ans = 6×1

    0.8392
    0.6288
    0.1338
    0.2071
    0.6072
    0.6299
```

Read from Memory Map as Nonscalar Structure

Map portions of data in the file, `records.dat`, as a sequence of multiple data types.

Call the `memmapfile` function to create a memory map, `m`.

```
m = memmapfile('records.dat',  ...
    'Format', {              ...
       'uint16' [5 8] 'x';   ...
       'double' [4 5] 'y' });
```

The `Format` parameter tells `memmapfile` to treat the first 80 bytes of the file as a 5-by-8 matrix of `uint16` values, and the 160 bytes after that as a 4-by-5 matrix of `double` values. This pattern repeats until the end of the file is reached.

Copy the `Data` property to a variable, `d`.

```
d = m.Data
```

```
d=166×1 struct array with fields:
    x
    y
```

`d` is a 166-element structure array with two fields. `d` is a nonscalar structure array because the file is mapped as a repeating sequence of multiple data types.

Examine one structure in the array to show the format of each field.

```
d(3)
```

```
ans = struct with fields:
    x: [5x8 uint16]
    y: [4x5 double]
```

Read the `x` field of that structure from the file.

```
d(3).x
```

```
ans = 5x8 uint16 matrix

   19972   47529   19145   16356   46507   47978   35550   16341
   60686   51944   16362   58647   35418   58072   16338   62509
   51075   16364   54226   34395    8341   16341   33787   57669
```

```
   16351   35598    6686   11480   16357   28709   36239    5932
   44292   15577   41755   16362   30311   31712   54813   16353
```

MATLAB formats the block of data as a 5-by-8 matrix of `uint16` values, as specified by the `Format` property.

Read the `y` field of that structure from the file.

```
d(3).y
```

```
ans = 4×5

    0.7271    0.3704    0.6946    0.5226    0.2714
    0.3093    0.7027    0.6213    0.8801    0.2523
    0.8385    0.5466    0.7948    0.1730    0.8757
    0.5681    0.4449    0.9568    0.9797    0.7373
```

MATLAB formats the block of data as a 4-by-5 matrix of `double` values.

Modify Map Properties and Analyze Data

This part of the example shows how to plot the Fourier transform of data read from a file via a memory map. It then modifies several properties of the existing map, reads from a different part of the data file, and plots a histogram from that data.

Create a sample file named `double.dat`.

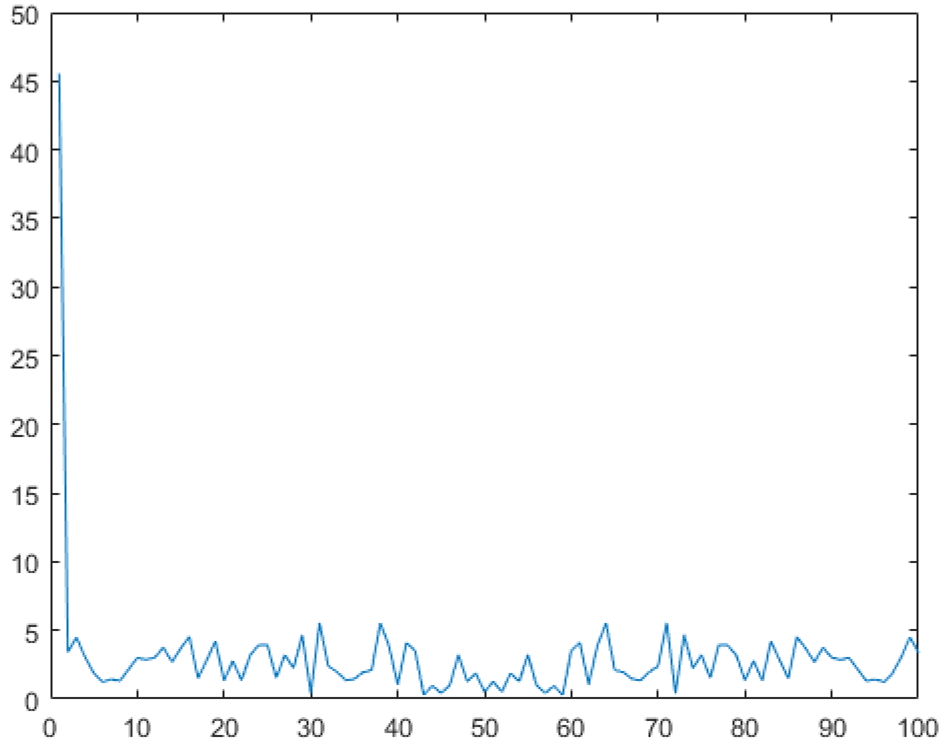
```
randData = gallery('uniformdata',[5000,1],0);
fileID = fopen('double.dat','w');
fwrite(fileID,randData,'double');
fclose(fileID);
```

Create a `memmapfile object` of 1,000 elements of type `double`, starting at the 1025th byte.

```
m = memmapfile('double.dat','Offset',1024,  ...
      'Format','double','Repeat',1000);
```

Copy the `Data` property to a variable, `k`. Then, get data associated with the map and plot the FFT of the first 100 values of the map.

```
k = m.Data;
plot(abs(fft(k(1:100))))
```



This is the first time that data is referenced and is when the actual mapping of the file to the MATLAB address space takes place.

Change the map properties, but continue using the same file. Whenever you change the value of a memory map property, MATLAB remaps the file to memory.

```
m.Offset = 4096;
m.Format = 'single';
m.Repeat = 800;
```

`m` is now a `memmapfile` object of 800 elements of type `single`. The map now begins at the 4096th byte in the file, `records.dat`.

Read from the portion of the file that begins at the 4096th byte, and calculate the maximum value of the data. This command maps a new region and unmaps the previous region.

```
X = max(m.Data)
```

```
X = single
    7.5449e+37
```

See Also

`memmapfile`

More About

- "Map File to Memory" on page 10-5
- "Write to Mapped File" on page 10-14

Write to Mapped File

This example shows how to create three different memory maps, and then write to each of the maps using the appropriate syntax. Then, it shows how to work with copies of your mapped data.

You can write to a file using the same MATLAB commands you use to access variables in the MATLAB workspace. By accessing the `Data` property of the memory map, the contents of the mapped file appear as an array in the currently active workspace. Simply index into this array to write data to the file. The syntax to use when writing to mapped memory depends on the format of the `Data` property of the memory map.

In this section...

Write to Memory Mapped as Numeric Array

First, create a sample file named `records.dat`, in your current folder.

```
myData = gallery('uniformdata', [5000,1], 0);

fileID = fopen('records.dat','w');
fwrite(fileID, myData,'double');
fclose(fileID);
```

Map the file as a sequence of 16-bit-unsigned integers. Use the `Format` name-value pair argument to specify that the values are of type `uint16`.

```
m = memmapfile('records.dat', ...
      'Offset',20,            ...
      'Format','uint16',      ...
      'Repeat',15);
```

Because the file is mapped as a sequence of a single class (`uint16`), `Data` is a numeric array.

Ensure that you have write permission to the mapped file. Set the `Writable` property of the memory map, `m`, to `true`.

```
m.Writable = true;
```

Create a matrix `X` that is the same size as the `Data` property, and write it to the mapped part of the file. All of the usual MATLAB indexing and class rules apply when assigning values to data via a memory map. The class that you assign to must be big enough to hold the value being assigned.

```
X = uint16(1:1:15);
m.Data = X;
```

`X` is a 1-by-15 vector of integer values ranging from 1 to 15.

Verify that new values were written to the file. Specify an `Offset` value of 0 to begin reading from the beginning of the file. Specify a `Repeat` value of 35 to view a total of 35 values. Use the `reshape` function to display the values as a 7-by-5 matrix.

```
m.Offset = 0;
m.Repeat = 35;
reshape(m.Data,5,7)'
```

```
ans = 7x5 uint16 matrix

   47662   34773   26485   16366   58664
   25170   38386   16333   14934    9028
       1       2       3       4       5
       6       7       8       9      10
      11      12      13      14      15
   10085   14020   16349   37120   31342
   62110   16274    9357   44395   18679
```

The values in `X` have been written to the file, `records.dat` .

Write to Memory Mapped as Scalar Structure

Map a region of the file, `records.dat`, as a 300-by-8 matrix of type `uint16` that can be referenced by the field name, `x`, followed by a 200-by-5 matrix of type `double` that can be reference by the field name, `y`. Specify write permission to the mapped file using the `Writable` name-value pair argument.

```
m = memmapfile('records.dat',     ...
      'Format', {                 ...
         'uint16' [300 8] 'x';    ...
         'double' [200 5] 'y' },  ...
      'Repeat', 1, 'Writable', true);
```

View the `Data` property

```
m.Data
```

```
ans = struct with fields:
    x: [300x8 uint16]
    y: [200x5 double]
```

`Data` is a scalar structure array. This is because the file, `records.dat`, is mapped as containing multiple data types that do not repeat.

Replace the matrix in the field, `x`, with a matrix of all ones.

```
 m.Data.x = ones(300,8,'uint16');
```

Write to Memory Mapped as Nonscalar Structure

Map the file, `records.dat`, as a 25-by-8 matrix of type `uint16` followed by a 15-by-5 matrix of type `double`. Repeat the pattern 20 times.

```
 m = memmapfile('records.dat',    ...
      'Format', {               ...
```

```
        'uint16' [5 4] 'x';    ...
        'double' [15 5] 'y' },  ...
    'Repeat', 20, 'Writable', true);
```

View the Data property

```
m.Data
```

```
ans=20×1 struct array with fields:
    x
    y
```

Data is a nonscalar structure array, because the file is mapped as a repeating sequence of multiple data types.

Write an array of all ones to the field named x in the 12th element of Data.

```
m.Data(12).x = ones(5,4,'uint16');
```

For the 12th element of Data, write the value, 50, to all elements in rows 3 to 5 of the field, x.

```
m.Data(12).x(3:5,1:end) = 50;
```

View the field, x, of the 12th element of Data.

```
m.Data(12).x
```

```
ans = 5x4 uint16 matrix

    1    1    1    1
    1    1    1    1
   50   50   50   50
   50   50   50   50
   50   50   50   50
```

Syntaxes for Writing to Mapped File

The syntax to use when writing to mapped memory depends on the format of the Data property of the memory map. View the properties of the memory map by typing the name of the memmapfile object.

This table shows the syntaxes for writing a matrix, X, to a memory map, m.

Format of the Data Property	Syntax for Writing to Mapped File
Numeric array **Example:** `15x1 uint16 array`	`m.Data = X;`

Scalar (1-by-1) structure array
 	
Example:
 	
`1x1 struct array with fields:`
`x`
`y`
	`m.Data.`*`fieldname`*` = X;`
	
	`fieldname` is the name of a field.

Format of the Data Property	Syntax for Writing to Mapped File

Nonscalar (n-by-1) structure array
 	
Example:
 	
`20x1 struct array with fields:`
`x`
`y`
	`m.Data(k).`*`fieldname`*` = X;`
	
	`k` is a scalar index and *`fieldname`* is the name of a field.

The class of `X` and the number of elements in `X` must match those of the `Data` property or the field of the `Data` property being accessed. You cannot change the dimensions of the `Data` property after you have created the memory map using the `memmapfile` function. For example, you cannot diminish or expand the size of an array by removing or adding a row from the mapped array, `m.Data`.

If you map an entire file and then append to that file after constructing the map, the appended data is not included in the mapped region. If you need to modify the dimensions of data that you have mapped to a memory map, `m`, you must either modify the `Format` or `Repeat` properties for `m`, or recreate `m` using the `memmapfile` function.

Note To successfully modify a mapped file, you must have write permission for that file. If you do not have write permission, attempting to write to the file generates an error, even if the `Writable` property is `true`.

Work with Copies of Your Mapped Data

This part of the example shows how to work with copies of your mapped data. The data in variable `d` is a copy of the file data mapped by `m.Data(2)`. Because it is a copy, modifying array data in `d` does not modify the data contained in the file.

Create a sample file named `double.dat`.

```
myData = gallery('uniformdata',[5000,1],0) * 100;
fileID = fopen('double.dat','w');
fwrite(fileID,myData,'double');
fclose(fileID);
```

Map the file as a series of `double` matrices.

```
m = memmapfile('double.dat',    ...
      'Format', {               ...
         'double' [5 5] 'x';    ...
         'double' [4 5] 'y' });
```

View the values in `m.Data(2).x`.

```
m.Data(2).x
```

```
ans = 5×5

   50.2813   19.3431   69.7898   49.6552   66.0228
   70.9471   68.2223   37.8373   89.9769   34.1971
   42.8892   30.2764   86.0012   82.1629   28.9726
   30.4617   54.1674   85.3655   64.4910   34.1194
   18.9654   15.0873   59.3563   81.7974   53.4079
```

Copy the contents of `m.Data` to the variable, `d`.

```
d = m.Data;
```

Write all zeros to the field named `x` in the copy.

```
d(2).x(1:5,1:5) = 0;
```

Verify that zeros are written to `d(2).x`

```
d(2).x
```

```
ans = 5×5

     0     0     0     0     0
     0     0     0     0     0
     0     0     0     0     0
     0     0     0     0     0
     0     0     0     0     0
```

Verify that the data in the mapped file is not changed.

```
m.Data(2).x
```

```
ans = 5×5

   50.2813   19.3431   69.7898   49.6552   66.0228
   70.9471   68.2223   37.8373   89.9769   34.1971
   42.8892   30.2764   86.0012   82.1629   28.9726
   30.4617   54.1674   85.3655   64.4910   34.1194
   18.9654   15.0873   59.3563   81.7974   53.4079
```

See Also
`memmapfile`

More About
- “Map File to Memory” on page 10-5
- “Read from Mapped File” on page 10-9

Delete Memory Map

In this section...

Ways to Delete a Memory Map

To clear a `memmapfile` object from memory, do any of the following:

- Reassign another value to the `memmapfile` object's variable
- Clear the `memmapfile` object's variable from memory
- Exit the function scope in which the `memmapfile` object was created

The Effect of Shared Data Copies On Performance

When you assign the `Data` field of the `memmapfile` object to a variable, MATLAB makes a shared data copy of the mapped data. This is very efficient because no memory actually gets copied. In the following statement, `d` is a shared data copy of the data mapped from the file:

```
d = m.Data;
```

When you finish using the mapped data, make sure to clear any variables that share data with the mapped file before clearing the `memmapfile` object itself. If you clear the object first, then the sharing of data between the file and dependent variables is broken, and the data assigned to such variables must be copied into memory before the object is cleared. If access to the mapped file was over a network, then copying this data to local memory can take considerable time. Therefore, if you assign `m.Data` to the variable, `d`, you should be sure to clear `d` before clearing `m` when you are finished with the memory map.

Share Memory Between Applications

This example shows how to implement two separate MATLAB processes that communicate with each other by writing and reading from a shared file. They share the file by mapping part of their memory space to a common location in the file. A write operation to the memory map belonging to the first process can be read from the map belonging to the second, and vice versa.

One MATLAB process (running `send.m`) writes a message to the file via its memory map. It also writes the length of the message to byte 1 in the file, which serves as a means of notifying the other process that a message is available. The second process (running `answer.m`) monitors byte 1 and, upon seeing it set, displays the received message, puts it into uppercase, and echoes the message back to the sender.

Prior to running the example, copy the `send` and `answer` functions to files `send.m` and `answer.m` in your current working directory.

The send Function

This function prompts you to enter text and then, using memory-mapping, passes the text to another instance of MATLAB that is running the `answer` function.

```
function send
% Interactively send a message to ANSWER using memmapfile class.

filename = fullfile(tempdir, 'talk_answer.dat');

% Create the communications file if it is not already there.
if ~exist(filename, 'file')
    [f, msg] = fopen(filename, 'wb');
    if f ~= -1
        fwrite(f, zeros(1,256), 'uint8');
        fclose(f);
    else
        error('MATLAB:demo:send:cannotOpenFile', ...
              'Cannot open file "%s": %s.', filename, msg);
    end
end

% Memory map the file.
m = memmapfile(filename, 'Writable', true, 'Format', 'uint8');

while true
    % Set first byte to zero, indicating a message is not
    % yet ready.
    m.Data(1) = 0;

    str = input('Enter text (or RETURN to end): ', 's');

    len = length(str);
    if (len == 0)
        disp('Terminating SEND function.')
        break;
    end

    % Warn if the message is longer than 255 characters.
    if len > 255
```

```
        warning('ml:ml','SEND input will be truncated to 255 characters.');
    end
    str = str(1:min(len,255));  % Limit message to 255 characters.
    len = length(str); % Update len if str has been truncated.

    % Update the file via the memory map.
    m.Data(2:len+1) = str;
    m.Data(1)=len;


    % Wait until the first byte is set back to zero,
    % indicating that a response is available.
    while (m.Data(1) ~= 0)
        pause(.25);
    end

    % Display the response.
    disp('response from ANSWER is:')
    disp(char(m.Data(2:len+1))')

end
```

The answer Function

The `answer` function starts a server that, using memory-mapping, watches for a message from `send`. When the message is received, `answer` replaces the message with an uppercase version of it, and sends this new message back to `send`. To use `answer`, call it with no inputs.

```
function answer
% Respond to SEND using memmapfile class.

disp('ANSWER server is awaiting message');

filename = fullfile(tempdir, 'talk_answer.dat');

% Create the communications file if it is not already there.
if ~exist(filename, 'file')
    [f, msg] = fopen(filename, 'wb');
    if f ~= -1
        fwrite(f, zeros(1,256), 'uint8');
        fclose(f);
    else
        error('MATLAB:demo:answer:cannotOpenFile', ...
              'Cannot open file "%s": %s.', filename, msg);
    end
end

% Memory map the file.
m = memmapfile(filename, 'Writable', true, 'Format', 'uint8');

while true
    % Wait until the first byte is not zero.
    while m.Data(1) == 0
        pause(.25);
    end

    % The first byte now contains the length of the message.
    % Get it from m.
```

```
    msg = char(m.Data(2:1+double(m.Data(1))))';

    % Display the message.
    disp('Received message from SEND:')
    disp(msg)

    % Transform the message to all uppercase.
    m.Data(2:1+double(m.Data(1))) = upper(msg);

    % Signal to SEND that the response is ready.
    m.Data(1) = 0;
end
```

Running the Example

To see what the example looks like when it is run, first, start two separate MATLAB sessions on the same computer system. Call the `send` function with no inputs in one MATLAB session. Call the `answer` function in the other session, to create a map in each of the processes' memory to the common file.

Run `send` in the first MATLAB session.

```
send
```

```
Enter text (or RETURN to end):
```

Run `answer` in the second MATLAB session.

```
answer
```

```
ANSWER server is awaiting message
```

Next, enter a message at the prompt displayed by the `send` function. MATLAB writes the message to the shared file. The second MATLAB session, running the `answer` function, loops on byte 1 of the shared file and, when the byte is written by `send`, `answer` reads the message from the file via its memory map. The `answer` function then puts the message into uppercase and writes it back to the file, and `send` (waiting for a reply) reads the message and displays it.

`send` writes a message and reads the uppercase reply.

```
Hello. Is there anybody out there?
```

```
response from ANSWER is:
HELLO. IS THERE ANYBODY OUT THERE?
Enter text (or RETURN to end):
```

`answer` reads the message from `send`.

```
Received message from SEND:
Hello.  Is there anybody out there?
```

Enter a second message at the prompt display by the `send` function. `send` writes the second message to the file.

```
I received your reply.
```

```
response from ANSWER is:
I RECEIVED YOUR REPLY.
Enter text (or RETURN to end):
```

`answer` reads the second message, put it into uppercase, and then writes the message to the file.

```
Received message from SEND:
I received your reply.
```

In the first instance of MATLAB, press **Enter** to exit the example.

```
Terminating SEND function.
```

11

Internet File Access and JSON

Server Authentication

MATLAB provides programmatic interfaces to these Web service interfaces.

- RESTful (Representational state transfer)—Use the `webread`, `webwrite`, and `websave` functions in “Web Access” to read content from RESTful Web services.
- HTTP (Hypertext Transfer Protocol)—Use the “HTTP Interface” API to implement advanced HTTP messaging semantics.

To use a proxy server, see “Proxy Server Authentication” on page 11-4.

Server Authentication For RESTful Web Services

NTLM and Kerberos are not supported on Linux and macOS platforms.

Authentication	Platform	`weboptions`	`weboptions` Arguments	System Setup
Basic	Windows Linux macOS	Required	`Username` and `Password`	N/A
Digest	Windows Linux macOS	Required	`Username` and `Password`	N/A
NTLM	Windows	Optional	Do not specify `Username` or `Password`	Logged into Windows domain
Kerberos	Windows	Optional	Do not specify `Username` or `Password`	Logged into Kerberos domain

Server Authentication For HTTP Web Services

Kerberos is not supported on Linux and macOS platforms.

Server Authentication	Platform	matlab.net.http `HTTPOptions` Object	matlab.net.http `Credentials` Properties	System Setup
Basic Digest	Windows Linux macOS	Credentials property	Username and Password	N/A
NTLM	Windows	Credentials property	`Username` and `Password` ignored	Logged into Windows domain
NTLM	Linux macOS	Credentials property	Username and Password	N/A
Kerberos	Windows	Credentials property	`Username` and `Password` ignored	Logged into Kerberos domain

See Also

`matlab.net.http.AuthenticationScheme` | `matlab.net.http.Credentials` | `matlab.net.http.HTTPOptions`

More About

- “Proxy Server Authentication” on page 11-4
- “Web Access”
- “HTTP Interface”

Proxy Server Authentication

MATLAB provides programmatic interfaces to these Web service interfaces.

- RESTful (Representational state transfer)—Use the `webread`, `webwrite`, and `websave` functions in “Web Access” to read content from RESTful Web services.
- HTTP (Hypertext Transfer Protocol)—Use the “HTTP Interface” API to implement advanced HTTP messaging semantics.

To authenticate to a server, see “Server Authentication” on page 11-2.

RESTful Web Services

MATLAB supports Basic, Digest, and NTLM proxy authentication types. To specify proxy server settings, choose one of these:

1. “Use MATLAB Web Preferences For Proxy Server Settings” on page 11-4
2. “Use System Settings For Proxy Server Settings” on page 11-5

If you specify the values using Web preferences, then MATLAB ignores system settings.

HTTP Web Services

MATLAB supports Basic, Digest, and NTLM proxy authentication types. To specify proxy server settings, choose one of these:

1. If you specify a ProxyURI in a `matlab.net.http.HTTPOptions` object, then set the Username and Password properties in matlab.net.http.Credentials.
2. “Use MATLAB Web Preferences For Proxy Server Settings” on page 11-4
3. “Use System Settings For Proxy Server Settings” on page 11-5

MATLAB chooses the first setting in this list.

Use MATLAB Web Preferences For Proxy Server Settings

You can specify proxy server settings using MATLAB “Web Preferences”.

Note Settings in Web Preferences override system settings.

To specify the proxy server settings:

1. On the **Home** tab, in the **Environment** section, click **Preferences**. Select **MATLAB > Web**.
2. Select the **Use a proxy server to connect to the Internet** check box.
3. Specify values for **Proxy host** and **Proxy port**.

   Examples of acceptable formats for the host are: `172.16.10.8` and `ourproxy`. For the port, enter an integer only, such as `22`. If you do not know the values for your proxy server, ask your system or network administrator for the information.

If your proxy server requires a user name and password, select the **Use a proxy with authentication** check box. Then enter your proxy user name and password.

4 Ensure that your settings work by clicking the **Test connection** button.

MATLAB attempts to connect to `https://www.mathworks.com`:

- If MATLAB can access the Internet, **Success!** appears next to the button.
- If MATLAB cannot access the Internet, **Failed!** appears next to the button. Correct the values you entered and try again. If you still cannot connect, try using the values you used when you authenticated your MATLAB license.

5 Click **OK** to accept the changes.

6 Restart MATLAB to enable the changes.

Use System Settings For Proxy Server Settings

If no proxy is specified in MATLAB Web preferences, then MATLAB uses the proxy set in the operating system preferences.

Operating System	System Certs
Windows	Windows Certificate Store
macOS	macOS KeyChain
Linux	Environment variables `http_proxy` and `https_proxy`

To specify proxy server settings in system preferences, refer to your Windows, Linux, or macOS operating system documentation.

MATLAB does not take into account proxy exceptions which you configure in Windows.

See Also

`matlab.net.http.Credentials` | `matlab.net.http.HTTPOptions`

More About

- “Server Authentication” on page 11-2
- “Web Access”
- “HTTP Interface”

MATLAB and Web Services Security

This topic describes how MATLAB handles security for web services. For a complete description of computer security, you need to consult external resources.

MATLAB Does Not Verify Certificate Chains

For HTTPS connections, the `webread`, `webwrite`, and `websave` functions verify that the certificate domain matches the host name of the web service. These functions do not verify the certificate chain. For a complete description of computer security, you need to consult external resources.

See Also

`webread` | `websave` | `webwrite`

Download Data from Web Service

This example shows how to download data from a web service with the `webread` function. The World Bank provides various climate data via the World Bank Climate Data API. A call to this API returns data in JSON format. `webread` converts JSON objects to structures that are convenient for analysis in MATLAB.

Use `webread` to read USA average annual temperatures into a structure array.

```
api = 'http://climatedataapi.worldbank.org/climateweb/rest/v1/';
url = [api 'country/cru/tas/year/USA'];
S = webread(url)
```

```
S =

112x1 struct array with fields:

    year
    data
```

`webread` converted the data to a structure array with 112 elements. Each structure contains the temperature for a given year, from 1901 to 2012.

```
S(1)
```

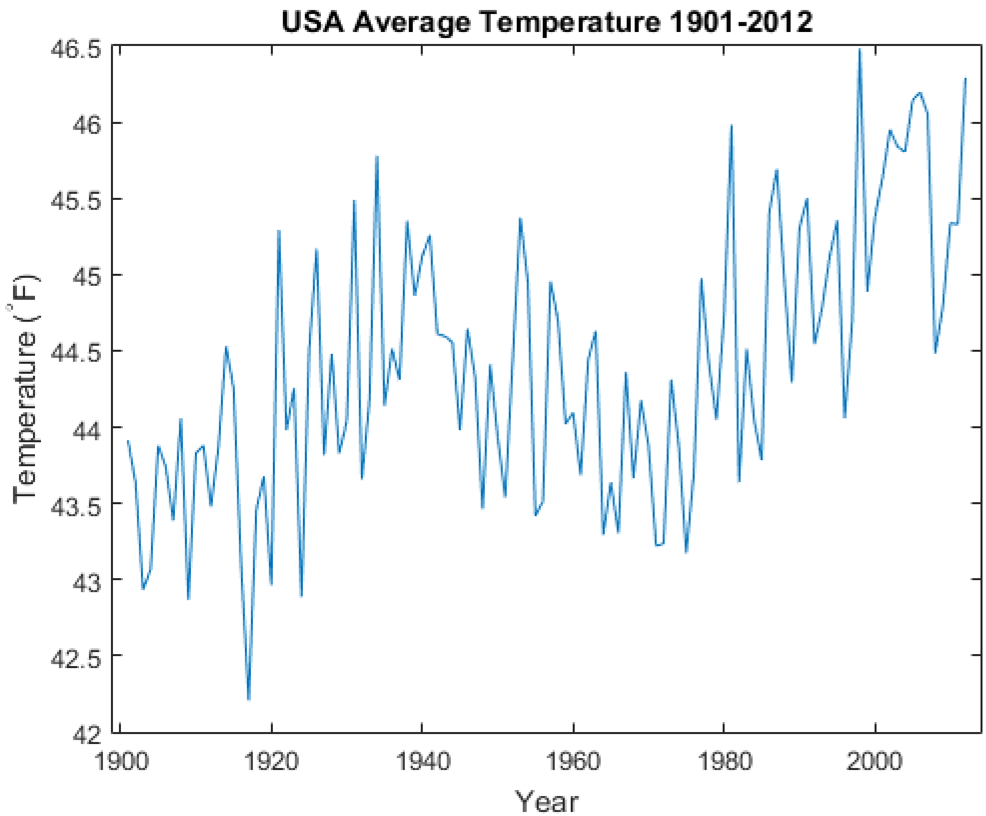
```
ans =

    year: 1901
    data: 6.6187
```

```
S(112)
```

```
ans =

    year: 2012
    data: 7.9395
```

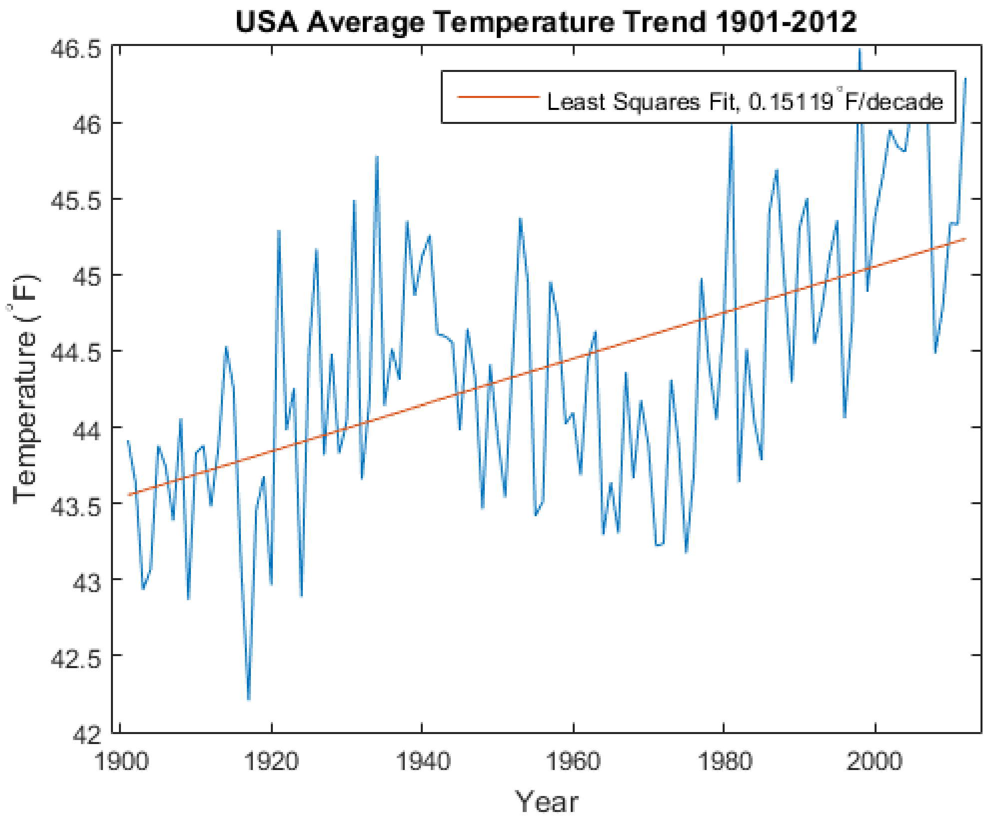
Plot the average temperature per year. Convert the temperatures and years to numeric arrays. Convert the years to a datetime object for ease of plotting, and convert the temperatures to degrees Fahrenheit.

```
temps = [S.data];
temps = 9/5 * temps + 32;
years = [S.year];
yearstoplot = datetime(years,1,1);
figure
plot(yearstoplot, temps);
title('USA Average Temperature 1901-2012')
xlabel('Year')
ylabel('Temperature (^{\circ}F)')
xmin = datetime(1899,1,1);
xmax = datetime(2014,1,1);
xlim([xmin xmax])
```



Overplot a least-squares fit of a line to the temperatures.

```
p = polyfit(years,temps,1);
ptemps = polyval(p,years);
deltat = p(1);
hold on
fl = plot(yearstoplot, ptemps);
xlim([xmin xmax])
title('USA Average Temperature Trend 1901-2012')
xlabel('Year')
ylabel('Temperature (^{\circ}F)')
deltat = num2str(10.0*deltat);
legend(fl,['Least Squares Fit, ', deltat, '^{\circ}F/decade'])
hold off
```



API and data courtesy of the World Bank: Climate Data API. (See World Bank: Climate Data API for more information about the API, and World Bank: Terms of Use.)

Convert Data from Web Service

This example shows how to download data from a web service and use a function as a content reader with `webread`.

The National Geophysical Data Center (NGDC) provides various geophysical and space weather data via a web service. Among other data sets, the NGDC aggregates sunspot numbers published by the American Association of Variable Star Observers (AAVSO). Use `webread` to download sunspot numbers for every year since 1945.

```
api = 'http://www.ngdc.noaa.gov/stp/space-weather/';
url = [api 'solar-data/solar-indices/sunspot-numbers/' ...
       'american/lists/list_aavso-arssn_yearly.txt'];
spots = webread(url);
whos('spots')
```

```
  Name       Size              Bytes  Class    Attributes

  spots      1x1269             2538  char
```

The NGDC web service returns the sunspot data as text. By default, `webread` returns the data as a character array.

```
spots(1:100)
```

```
ans =

        American
   Year     SSN
    1945    32.3
    1946    99.9
    1947   170.9
    1948   166.6
```

`webread` can use a function to return the data as a different type. You can use `readtable` with `webread` to return the sunspot data as a table.

Create a `weboptions` object that specifies a function for `readtable`.

```
myreadtable = @(filename)readtable(filename,'HeaderLines',1, ...
    'Format','%f%f','Delimiter','space','MultipleDelimsAsOne',1);
options = weboptions('ContentReader',myreadtable);
```

For this data, call `readtable` with several `Name,Value` input arguments to convert the data. For example, `Format` indicates that each row has two numbers. Spaces are delimiters, and multiple consecutive spaces are treated as a single delimiter. To call `readtable` with these input arguments, wrap `readtable` and the arguments in a new function, `myreadtable`. Create a `weboptions` object with `myreadtable` as the content reader.

Download sunspot data and return the data as a table.

```
spots = webread(url,options);
whos('spots')
```

```
  Name        Size             Bytes  Class    Attributes

  spots      76x2               2932  table
```

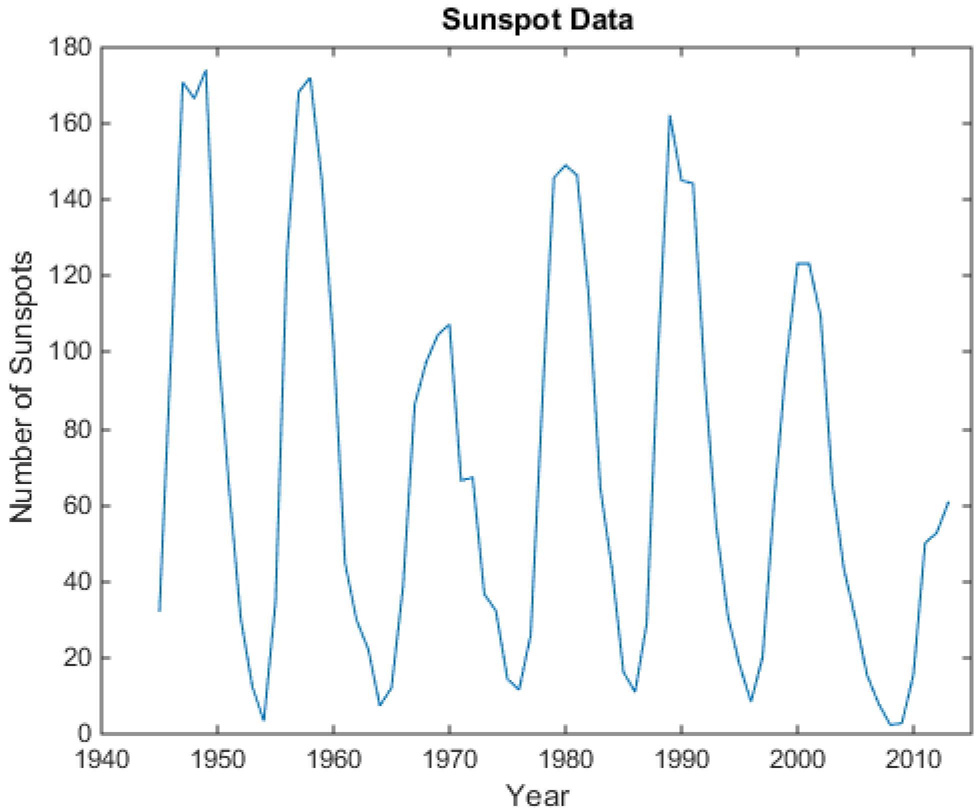
Display the sunspot data by column and row.

```
spots(1:4,{'Year','SSN'})
```

```
ans = 

    Year     SSN 
    ____    _____

    1945     32.3
    1946     99.9
    1947    170.9
    1948    166.6
```

Plot sunspot numbers by year. Use table functions to select sunspot numbers up to the year 2013. Convert the `Year` and `SSN` columns to arrays and plot them.

```
rows = spots.Year < 2014;
vars = {'Year','SSN'};
spots = spots(rows,vars);
year = spots.Year;
numspots = spots.SSN;
figure
plot(year,numspots);
title('Sunspot Data');
xlabel('Year');
ylabel('Number of Sunspots');
xlim([1940 2015])
ylim([0 180])
```



Aggregated data and web service courtesy of the NGDC. Sunspot data courtesy of the AAVSO, originally published in AAVSO Sunspot Counts: 1943-2013, AAVSO Solar Section (R. Howe, Chair).

- See NGDC Privacy Policy, Disclaimer, and Copyright for NGDC terms of service.
- See AAVSO Solar Section for more information on AAVSO solar data, including terms of use.

Download Web Page and Files

MATLAB provides two functions for reading content from RESTful web services: `webread` and `websave`. With the `webread` function, you can read the contents of a web page to a character array in the MATLAB workspace. With the `websave` function, you can save web page content to a file.

Because it can create a character array in the workspace, the `webread` function is useful for working with the contents of web pages in MATLAB. The `websave` function is useful for saving web pages to a local folder.

Note When `webread` returns HTML as a character array, remember that only the HTML in that specific web page is retrieved. The hyperlink targets, images, and so on, are not retrieved.

If you need to pass parameters to a web page, the `webread` and `websave` functions let you define the parameters as `Name, Value` pair arguments. For more information, see the `webread` and `websave` reference pages.

Example — Use the webread Function

The following procedure demonstrates how to retrieve the contents of the web page listing the files submitted to the MATLAB Central™ File Exchange, https://www.mathworks.com/matlabcentral/fileexchange/. It assigns the results to a character array, `fullList`:

```
filex = 'https://www.mathworks.com/matlabcentral/fileexchange/';
fullList = webread(filex);
```

Retrieve a list of only those files uploaded to the File Exchange within the past seven days that contain the word Simulink®. Set `duration` and `term` as parameters that `webread` passes to the web page.

```
filex = 'https://www.mathworks.com/matlabcentral/fileexchange/';
recent = webread(filex,'duration',7,'term','simulink');
```

Example — Use the websave Function

The following example builds on the procedure in the previous section, but saves the content to a file:

```
% Locate the list of files at the MATLAB Central File Exchange
% uploaded within the past 7 days, that contain "Simulink."
filex = 'https://www.mathworks.com/matlabcentral/fileexchange/';

% Save the Web content to a file.
recent = websave('contains_simulink.html',filex, ...
    'duration',7,'term','simulink');
```

MATLAB saves the web page as `contains_simulink.html`. The output argument `recent` contains the full path to `contains_simulink.html`. Call the `web` function to display `contains_simulink.html` in a browser.

```
web(recent)
```

This page has links to files uploaded to the MATLAB Central File Exchange.

Call Web Services from Functions

You can call `webread` from functions you define. Best practice is to allow your function to pass HTTP request options to `webread`.

This code sample shows how to download climate data for a country. The sample defines a function in a file named `worldBankTemps.m` that downloads annual temperatures from the World Bank and converts them to degrees Fahrenheit. You can pass additional HTTP request parameters with the `options` input argument. `options` is a `weboptions` object that `worldBankTemps` passes to `webread`. You can call `worldBankTemps` with a country name only when you do not need to define any other HTTP request parameters.

```
function temperatures = worldBankTemps(country,options)
% Get World Bank temperatures for a country, for example, 'USA'.
api = 'http://climatedataapi.worldbank.org/climateweb/rest/v1/';
api = [api 'country/cru/tas/year/'];
country = [api country];

% The options object contains additional HTTP
% request parameters. If worldBankTemps was
% not passed options as an input argument,
% create a default weboptions object.
if ~exist('options','var')
    options = weboptions;
end
s = webread(country,options);

% Convert data to arrays
temperatures = struct('Years',[],'DegreesInFahrenheit',[]);
temperatures(1).Years = [s.year];
temperatures(1).DegreesInFahrenheit = [s.data];

% Convert temperatures to Fahrenheit
temperatures(1).DegreesInFahrenheit = temperatures(1).DegreesInFahrenheit * 9/5 + 32;
end
```

To get temperature data for the USA, call `worldBankTemps`. If the connection to the World Bank web service times out, the service returns an error message.

```
S = worldBankTemps('USA')
```

```
Error using webread (line 112)
The connection to
URL 'http://climatedataapi.worldbank.org/climateweb/rest/v1/country/cru/tas/year/USA'
timed out after 5.0 seconds. Set options.Timeout to a higher value.
```

If you create `options` and set its `Timeout` property to 60 seconds, then you can call `worldBankTemps` again with `options` as an input argument. `worldBankTemps` passes `options` to `webread` as an input argument. This time `webread` keeps the connection open for a maximum of 60 seconds.

```
options = weboptions('Timeout',60);
S = worldBankTemps('USA',options)
```

```
S = 

                  Years: [1x112 double]
    DegreesInFahrenheit: [1x112 double]
```

If your code does not allow you to pass request options to `webread`, that limits your ability to respond to error messages returned by web services.

Error Messages Concerning Web Service Options

When you use a web service function in MATLAB the function might return an error message that advises you to set a property of `options`, such as `options.Timeout`. This table shows some typical error messages that refer to `options` properties and actions you can take in response.

Error Message Contains Phrase	Action To Be Taken
`Set options.Timeout to a higher value.`	`options = weboptions('Timeout',60)` `data = webread(url,options)`
`Set options.ContentType to 'json'.`	`options = weboptions('ContentType','json')` `data = webread(url,options)`
`...the provided authentication parameters, options.Username and options.Password, are incorrect.`	`options = weboptions('Username','`*`your username`*`','Password','`*`your password`*`')` `data = webread(url,options)`

Send Email

To send an email from MATLAB, use the `sendmail` function. You can also attach files to an email, which lets you mail files directly from MATLAB. To use `sendmail`, set up your email address and your SMTP server information with the `setpref` function.

The `setpref` function defines two mail-related preferences:

- Email address: This preference sets your email address that will appear on the message.

  ```
  setpref('Internet','E_mail','youraddress@yourserver.com');
  ```
- SMTP server: This preference sets your outgoing SMTP server address, which can be almost any email server that supports the Post Office Protocol (POP) or the Internet Message Access Protocol (IMAP).

  ```
  setpref('Internet','SMTP_Server','mail.server.network');
  ```

Find your outgoing SMTP server address in your email account settings in your email client application. You can also contact your system administrator for the information.

Once you have properly configured MATLAB, you can use the `sendmail` function. The `sendmail` function requires at least two arguments: the recipient's email address and the email subject.

```
sendmail('recipient@someserver.com','Hello From MATLAB!');
```

You can supply multiple email addresses using a cell array of character vectors.

```
sendmail({'recipient@someserver.com','recipient2@someserver.com'}, ...
    'Hello From MATLAB!');
```

You can specify a message body.

```
sendmail('recipient@someserver.com','Hello From MATLAB!', ...
    'Thanks for using sendmail.');
```

You can attach files to an email.

```
sendmail('recipient@someserver.com','Hello from MATLAB!', ...
    'Thanks for using sendmail.','C:\yourFileSystem\message.txt');
```

You cannot attach a file without including a message. However, the message can be empty.

You can attach multiple files to an email.

```
sendmail('recipient@someserver.com','Hello from MATLAB!', ...
    'Thanks for using sendmail.',{'C:\yourFileSystem\message.txt', ...
    'C:\yourFileSystem\message2.txt'});
```

See Also

`sendmail` | `setpref`

Perform FTP File Operations

This example shows how to use an FTP object to connect to an FTP server and perform remote file operations. To perform any file operation on an FTP server, follow these steps:

1. Connect to the server using the `ftp` function.
2. Perform operations using the appropriate MATLAB® FTP functions, such as the `cd`, `dir`, and `mget` functions. Specify the FTP object for all operations.
3. When you finish work on the server, close the connection using the `close` function.

The National Centers for Environmental Information (NCEI) maintain an anonymous FTP service providing public access to geophysical data. Access the FTP server to list its contents, download a file, and list contents of a subfolder.

First, open the connection.

```
ftpobj = ftp('ftp.ngdc.noaa.gov')
```

```
ftpobj = 

  FTP Object
     host: ftp.ngdc.noaa.gov
     user: anonymous
      dir: /
     mode: binary
```

List the contents of the top-level folder on the FTP server.

```
dir(ftpobj)
```

```

DMSP                         Solid_Earth                  google12c4c939d7b90761.html  mgg
INDEX.txt                    coastwatch                   hazards                      pub
README.txt                   dmsp4alan                    index.html                   tmp
STP                          ftp.html                     international                wdc
Snow_Ice                     geomag                       ionosonde
```

Download the file named `INDEX.txt` using the `mget` function. `mget` copies the file to the current MATLAB folder on your local machine. To view the contents of your copy of the file, use the `type` function.

```
mget(ftpobj,'INDEX.txt');
type INDEX.txt
```

```
                    National Centers for Environmental Information (NCEI),
                    formerly the National Geophysical Data Center (NGDC)

                          INDEX of anonymous ftp area
                               ftp.ngdc.noaa.gov

DIRECTORY/FILE DESCRIPTION OF CONTENTS
-------------- -----------------------------------------------------------------
pub/           Public access area
DMSP/          Defense Meteorological Satellite Data Archive
geomag/        Geomagnetism and geomagnetics models
```

```
hazards/       Natural Hazards data, volcanoes, tsunamis, earthquakes
international/ International program information on IAGA/Oersted/wdc
ionosonde/     Ionosonde data
mgg/           Limited Marine Geology and Geophysics (most data in http area)
OD/            Office of the Director
Snow_Ice/      Snow and Ice Data Center
Solid_Earth/   Historic Solid Earth Geophysics
STP/           Solar-Terrestrial Physics
tmp/           Pickup area for temporary outgoing data
wdc/           World Data Service for Geophysics, formerly World Data Centers
-------------- ----------------------------------------------------------------
Please see file README.txt in this directory for more information and how to
contact NCEI. Direct E-mail inquiries to ncei.info@noaa.gov

Also see our web site: http://www.ngdc.noaa.gov/

NCEI is part of the:
U.S. Department of Commerce, National Oceanic and Atmospheric Administration (NOAA),
National Environmental Satellite, Data and Information Service (NESDIS)
```

Change to the subfolder named `pub` on the FTP server.

```
cd(ftpobj,'pub')
```

```
ans = 
'/pub'
```

List the contents. `pub` is now the current folder on the FTP server. However, note that the current MATLAB folder on your local machine has not changed. When you specify an FTP object using functions such as `cd` and `dir`, the operations take place on the FTP server, not your local machine.

```
dir(ftpobj)
```

```
WebCD     coast     glac_lib  krm       outgoing  results   rgon
```

Close the connection to the FTP server.

```
close(ftpobj)
```

FTP service courtesy of the NCEI. See the NCEI Privacy Policy, Disclaimer, and Copyright for NCEI terms of service.

See Also

`cd` | `close` | `dir` | `ftp` | `mget`

Related Examples

- “Download Data from Web Service” on page 11-7
- “Download Web Page and Files” on page 11-13
- “Send Email” on page 11-16
- “Web Browsers and MATLAB”

Display Hyperlinks in the Command Window

In this section...

Create Hyperlinks to Web Pages

When you create a hyperlink to a Web page, append a full hypertext address on a single line as input to the `disp` or `fprintf` command. For example, the following command:

```
disp('<a href = "https://www.mathworks.com">The MathWorks Web Site</a>')
```

displays the following hyperlink in the Command Window:

```
The MathWorks Web Site
```

When you click this hyperlink, a MATLAB Web browser opens and displays the requested page.

Transfer Files Using FTP

To create a link to an FTP site, enter the site address as input to the `disp` command as follows:

```
disp('<a href = "ftp://ftp.mathworks.com">The MathWorks FTP Site</a>')
```

This command displays the following as a link in the Command Window:

```
The MathWorks FTP Site
```

When you click the link, a MATLAB browser opens and displays the requested FTP site.

Customize JSON Encoding for MATLAB Classes

This example shows how to customize the `jsonencode` function for a user-defined MATLAB class.

This class `Person.m` has a public property `Name` and a private property `Age`. If you call `jsonencode` to encode the data, the function only converts the public property.

```
classdef Person
    properties
        Name;
    end
    properties (Access = private)
        Age;
    end
    methods
        function obj = Person(name,age)
            obj.Name = name;
            obj.Age = age;
        end
    end
end
```

1. Display a JSON-encoded instance of `Person`.

   ```
   obj = Person('Luke',19);
   jsonencode(obj)
   ```

   ```
   ans =

       '{"Name":"Luke"}'
   ```
2. To display the private property `Age`, customize `jsonencode` by adding this function to the `methods` block:

   ```
   function json = jsonencode(obj)
       s = struct("Name", obj.Name, "Age", obj.Age);
       json = jsonencode(s);
   end
   ```

   The signature of the function must match the `jsonencode` signature, which takes a class object as input and returns a string or a character vector in JSON format.
3. Display the customized object.

   ```
   obj = Person('Luke',19);
   jsonencode(obj)
   ```

   ```
   ans =

       '{"Name":"Luke","Age":19}'
   ```

See Also

`jsonencode`

Large Data

Getting Started with MapReduce

As the number and type of data acquisition devices grows annually, the sheer size and rate of data being collected is rapidly expanding. These big data sets can contain gigabytes or terabytes of data, and can grow on the order of megabytes or gigabytes per day. While the collection of this information presents opportunities for insight, it also presents many challenges. Most algorithms are not designed to process big data sets in a reasonable amount of time or with a reasonable amount of memory. MapReduce allows you to meet many of these challenges to gain important insights from large data sets.

In this section...

What Is MapReduce?

MapReduce is a programming technique for analyzing data sets that do not fit in memory. You may be familiar with Hadoop® MapReduce, which is a popular implementation that works with the Hadoop Distributed File System (HDFS™). MATLAB provides a slightly different implementation of the MapReduce technique with the `mapreduce` function.

`mapreduce` uses a datastore to process data in small blocks that individually fit into memory. Each block goes through a Map phase, which formats the data to be processed. Then the intermediate data blocks go through a Reduce phase, which aggregates the intermediate results to produce a final result. The Map and Reduce phases are encoded by *map* and *reduce* functions, which are primary inputs to `mapreduce`. There are endless combinations of map and reduce functions to process data, so this technique is both flexible and extremely powerful for tackling large data processing tasks.

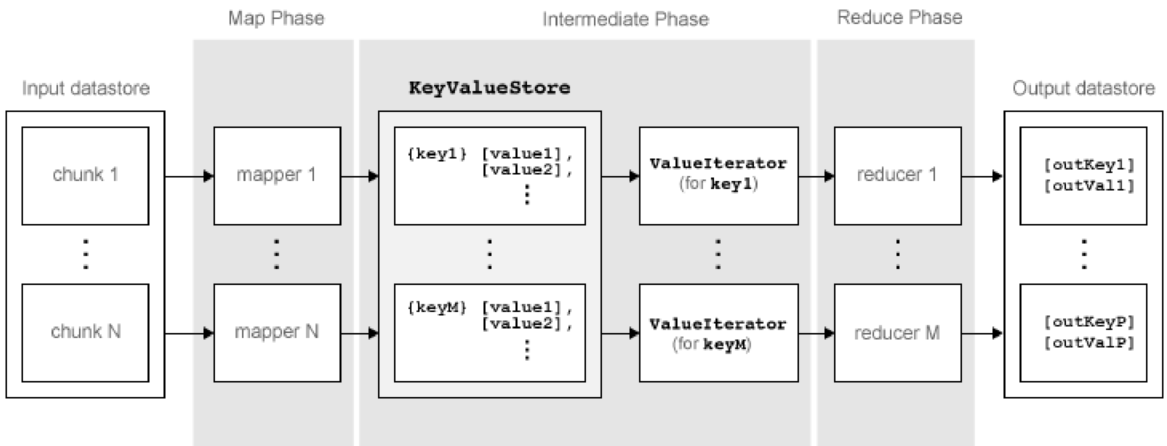
`mapreduce` lends itself to being extended to run in several environments. For more information about these capabilities, see “Speed Up and Deploy MapReduce Using Other Products” on page 12-17.

The utility of the `mapreduce` function lies in its ability to perform calculations on large collections of data. Thus, `mapreduce` is not well-suited for performing calculations on *normal* sized data sets which can be loaded directly into computer memory and analyzed with traditional techniques. Instead, use `mapreduce` to perform a statistical or analytical calculation on a data set that does not fit in memory.

Each call to the map or reduce function by `mapreduce` is independent of all others. For example, a call to the map function cannot depend on inputs or results from a previous call to the map function. It is best to break up such calculations into multiple calls to `mapreduce`.

MapReduce Algorithm Phases

`mapreduce` moves each block of data in the input datastore through several phases before reaching the final output. The following figure outlines the phases of the algorithm for `mapreduce`.



The algorithm has the following steps:

1. `mapreduce` reads a block of data from the input datastore using `[data,info] = read(ds)`, and then calls the map function to work on that block.
2. The map function receives the block of data, organizes it or performs a precursory calculation, and then uses the `add` and `addmulti` functions to add key-value pairs to an intermediate data storage object called a `KeyValueStore`. The number of calls to the map function by `mapreduce` is equal to the number of blocks in the input datastore.
3. After the map function works on all of the blocks of data in the datastore, `mapreduce` groups all of the values in the intermediate `KeyValueStore` object by unique key.
4. Next, `mapreduce` calls the reduce function once for each unique key added by the map function. Each unique key can have many associated values. `mapreduce` passes the values to the reduce function as a `ValueIterator` object, which is an object used to iterate over the values. The `ValueIterator` object for each unique key contains all the associated values for that key.
5. The reduce function uses the `hasnext` and `getnext` functions to iterate through the values in the `ValueIterator` object one at a time. Then, after aggregating the intermediate results from the map function, the reduce function adds final key-value pairs to the output using the `add` and `addmulti` functions. The order of the keys in the output is the same as the order in which the reduce function adds them to the final `KeyValueStore` object. That is, `mapreduce` does not explicitly sort the output.

Note The reduce function writes the final key-value pairs to a final `KeyValueStore` object. From this object, `mapreduce` pulls the key-value pairs into the output datastore, which is a `KeyValueDatastore` object by default.

Example MapReduce Calculation

This example uses a simple calculation (the mean travel distance in a set of flight data) to illustrate the steps needed to run `mapreduce`.

Prepare Data

The first step to using `mapreduce` is to construct a datastore for the data set. Along with the map and reduce functions, the datastore for a data set is a required input to `mapreduce`, since it allows `mapreduce` to process the data in blocks.

`mapreduce` works with most types of datastores. For example, create a `TabularTextDatastore` object for the `airlinesmall.csv` data set.

```
ds = tabularTextDatastore('airlinesmall.csv','TreatAsMissing','NA')
```

```
ds = 

  TabularTextDatastore with properties:

                      Files: {
                             ' ...\matlab\toolbox\matlab\demos\airlinesmall.csv'
                             }
                    Folders: {
                             ' ...\matlab\toolbox\matlab\demos'
                             }
               FileEncoding: 'UTF-8'
   AlternateFileSystemRoots: {}
      PreserveVariableNames: false
          ReadVariableNames: true
              VariableNames: {'Year', 'Month', 'DayofMonth' ... and 26 more}
             DatetimeLocale: en_US

  Text Format Properties:
             NumHeaderLines: 0
                  Delimiter: ','
               RowDelimiter: '\r\n'
             TreatAsMissing: 'NA'
               MissingValue: NaN

  Advanced Text Format Properties:
            TextscanFormats: {'%f', '%f', '%f' ... and 26 more}
                   TextType: 'char'
         ExponentCharacters: 'eEdD'
               CommentStyle: ''
                 Whitespace: ' \b\t'
    MultipleDelimitersAsOne: false

  Properties that control the table returned by preview, read, readall:
      SelectedVariableNames: {'Year', 'Month', 'DayofMonth' ... and 26 more}
            SelectedFormats: {'%f', '%f', '%f' ... and 26 more}
                   ReadSize: 20000 rows
                 OutputType: 'table'
                   RowTimes: []

  Write-specific Properties:
     SupportedOutputFormats: ["txt"    "csv"    "xlsx"    "xls"    "parquet"    "parq"]
        DefaultOutputFormat: "txt"
```

Several of the previously described options are useful in the context of `mapreduce`. The `mapreduce` function executes `read` on the datastore to retrieve data to pass to the map function. Therefore, you can use the `SelectedVariableNames`, `SelectedFormats`, and `ReadSize` options to directly configure the block size and type of data that `mapreduce` passes to the map function.

For example, to select the `Distance` (total flight distance) variable as the only variable of interest, specify `SelectedVariableNames`.

```
ds.SelectedVariableNames = 'Distance';
```

Now, whenever the `read`, `readall`, or `preview` functions act on `ds`, they will return only information for the `Distance` variable. To confirm this, you can preview the first few rows of data in the datastore. This allows you to examine the format of the data that the `mapreduce` function will pass to the map function.

```
preview(ds)
```

```
ans =

  8×1 table

    Distance
    ________
```

```
308
296
480
296
373
308
447
954
```

To view the *exact* data that `mapreduce` will pass to the map function, use `read`.

For additional information and a complete summary of the available options, see “Datastore”.

Write Map and Reduce Functions

The `mapreduce` function automatically calls the map and reduce functions during execution, so these functions must meet certain requirements to run properly.

1. The inputs to the map function are `data`, `info`, and `intermKVStore`:

   - `data` and `info` are the result of a call to the `read` function on the input datastore, which `mapreduce` executes automatically before each call to the map function.
   - `intermKVStore` is the name of the intermediate `KeyValueStore` object to which the map function needs to add key-value pairs. The `add` and `addmulti` functions use this object name to add key-value pairs. If none of the calls to the map function add key-value pairs to `intermKVStore`, then `mapreduce` does not call the reduce function and the resulting datastore is empty.

   A simple example of a map function is:

   ```
   function MeanDistMapFun(data, info, intermKVStore)
       distances = data.Distance(~isnan(data.Distance));
       sumLenValue = [sum(distances)  length(distances)];
       add(intermKVStore, 'sumAndLength', sumLenValue);
   end
   ```

   This map function has only three lines, which perform some straightforward roles. The first line filters out all `NaN` values in the block of distance data. The second line creates a two-element vector with the total distance and count for the block, and the third line adds that vector of values to `intermKVStore` with the key, `'sumAndLength'`. After this map function runs on all of the blocks of data in `ds`, the `intermKVStore` object contains the total distance and count for each block of distance data.

   Save this function in your current folder as `MeanDistMapFun.m`.
2. The inputs to the reduce function are `intermKey`, `intermValIter`, and `outKVStore`:

   - `intermKey` is for the active key added by the map function. Each call to the reduce function by `mapreduce` specifies a new unique key from the keys in the intermediate `KeyValueStore` object.
   - `intermValIter` is the `ValueIterator` associated with the active key, `intermKey`. This `ValueIterator` object contains all of the values associated with the active key. Scroll through the values using the `hasnext` and `getnext` functions.
   - `outKVStore` is the name for the final `KeyValueStore` object to which the reduce function needs to add key-value pairs. `mapreduce` takes the output key-value pairs from `outKVStore`

and returns them in the output datastore, which is a `KeyValueDatastore` object by default. If none of the calls to the reduce function add key-value pairs to `outKVStore`, then `mapreduce` returns an empty datastore.

A simple example of a reduce function is:

```
function MeanDistReduceFun(intermKey, intermValIter, outKVStore)
    sumLen = [0 0];
    while hasnext(intermValIter)
        sumLen = sumLen + getnext(intermValIter);
    end
    add(outKVStore, 'Mean', sumLen(1)/sumLen(2));
end
```

This reduce function loops through each of the distance and count values in `intermValIter`, keeping a running total of the distance and count after each pass. After this loop, the reduce function calculates the overall mean flight distance with a simple division, and then adds a single key to `outKVStore`.

Save this function in your current folder as `MeanDistReduceFun.m`.

For information about writing more advanced map and reduce functions, see “Write a Map Function” on page 12-9 and “Write a Reduce Function” on page 12-13.

Run mapreduce

After you have a datastore, a map function, and a reduce function, you can call `mapreduce` to perform the calculation. To calculate the average flight distance in the data set, call `mapreduce` using `ds`, `MeanDistMapFun`, and `MeanDistReduceFun`.

```
outds = mapreduce(ds, @MeanDistMapFun, @MeanDistReduceFun);
```

```
********************************
*      MAPREDUCE PROGRESS      *
********************************
Map   0% Reduce   0%
Map  16% Reduce   0%
Map  32% Reduce   0%
Map  48% Reduce   0%
Map  65% Reduce   0%
Map  81% Reduce   0%
Map  97% Reduce   0%
Map 100% Reduce   0%
Map 100% Reduce 100%
```

By default, the `mapreduce` function displays progress information at the command line and returns a `KeyValueDatastore` object that points to files in the current folder. You can adjust all three of these options using the `Name,Value` pair arguments for `'OutputFolder'`, `'OutputType'`, and `'Display'`. For more information, see the reference page for `mapreduce`.

View Results

Use the `readall` function to read the key-value pairs from the output datastore.

```
readall(outds)
```

```
ans =
```

```
  1×2 table

      Key            Value     
    ________    ____________

    {'Mean'}    {[702.1630]}
```

See Also

`mapreduce` | `tabularTextDatastore`

Related Examples

- “Build Effective Algorithms with MapReduce” on page 12-18

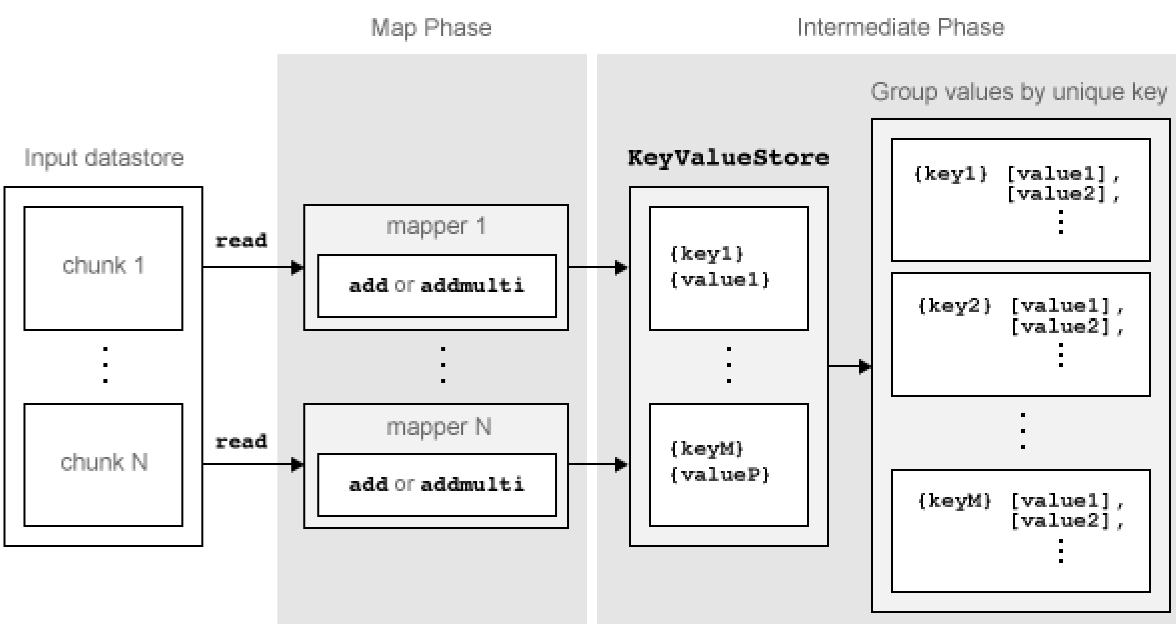
Write a Map Function

In this section...

Role of Map Function in MapReduce

`mapreduce` requires both an input map function that receives blocks of data and that outputs intermediate results, and an input reduce function that reads the intermediate results and produces a final result. Thus, it is normal to break up a calculation into two related pieces for the map and reduce functions to fulfill separately. For example, to find the maximum value in a data set, the map function can find the maximum value in each block of input data, and then the reduce function can find the single maximum value among all of the intermediate maxima.

This figure shows the Map phase of the `mapreduce` algorithm.



The Map phase of the `mapreduce` algorithm has the following steps:

1. `mapreduce` reads a single block of data using the `read` function on the input datastore, then calls the map function to work on the block.
2. The map function then works on the individual block of data and adds one or more key-value pairs to the intermediate `KeyValueStore` object using the `add` or `addmulti` functions.
3. `mapreduce` repeats this process for each of the blocks of data in the input datastore, so that the total number of calls to the map function is equal to the number of blocks of data. The `ReadSize` property of the datastore determines the number of data blocks.

The Map phase of the `mapreduce` algorithm is complete when the map function processes each of the blocks of data in the input datastore. The result of this phase of the `mapreduce` algorithm is a

KeyValueStore object that contains all of the key-value pairs added by the map function. After the Map phase, mapreduce prepares for the Reduce phase by grouping all the values in the KeyValueStore object by unique key.

Requirements for Map Function

mapreduce automatically calls the map function for each block of data in the input datastore. The map function must meet certain basic requirements to run properly during these automatic calls. These requirements collectively ensure the proper movement of data through the Map phase of the mapreduce algorithm.

The inputs to the map function are data, info, and intermKVStore:

- data and info are the result of a call to the read function on the input datastore, which mapreduce executes automatically before each call to the map function.
- intermKVStore is the name of the intermediate KeyValueStore object to which the map function needs to add key-value pairs. The add and addmulti functions use this object name to add key-value pairs. If the map function does not add any key-value pairs to the intermKVStore object, then mapreduce does not call the reduce function and the resulting datastore is empty.

In addition to these basic requirements for the map function, the key-value pairs added by the map function must also meet these conditions:

1. Keys must be numeric scalars, character vectors, or strings. Numeric keys cannot be NaN, complex, logical, or sparse.
2. All keys added by the map function must have the same class.
3. Values can be any MATLAB object, including all valid MATLAB data types.

Note The above key-value pair requirements may differ when using other products with mapreduce. See the documentation for the appropriate product to get product-specific key-value pair requirements.

Sample Map Functions

Here are a few illustrative map functions used in mapreduce examples.

Identity Map Function

A map function that simply returns what mapreduce passes to it is called an *identity mapper*. An identity mapper is useful to take advantage of the grouping of values by unique key before doing calculations in the reduce function. The identityMapper mapper file is one of the mappers used in the example “Tall Skinny QR (TSQR) Matrix Factorization Using MapReduce” on page 12-73.

```
function identityMapper(data, info, intermKVStore)
  % This mapper function simply copies the data and add them to the
  % intermKVStore as intermediate values.
  x = data.Value{:,:};
  add(intermKVStore,'Identity', x);
end
```

Simple Map Function

One of the simplest examples of a nonidentity mapper is `maxArrivalDelayMapper`, which is the mapper for the example “Find Maximum Value with MapReduce” on page 12-32. For each chunk of input data, this mapper calculates the maximum arrival delay and adds a key-value pair to the intermediate `KeyValueStore`.

```
function maxArrivalDelayMapper (data, info, intermKVStore)
  partMax = max(data.ArrDelay);
  add(intermKVStore, 'PartialMaxArrivalDelay',partMax);
end
```

Advanced Map Function

A more advanced example of a mapper is `statsByGroupMapper`, which is the mapper for the example “Compute Summary Statistics by Group Using MapReduce” on page 12-61. This mapper uses a nested function to calculate several statistical quantities (count, mean, variance, and so on) for each chunk of input data, and then adds several key-value pairs to the intermediate `KeyValueStore` object. Also, this mapper uses four input arguments, whereas `mapreduce` only accepts a map function with three input arguments. To get around this, pass in the extra parameter using an anonymous function during the call to `mapreduce`, as outlined in the example.

```
function statsByGroupMapper(data, ~, intermKVStore, groupVarName)
  % Data is a n-by-3 table. Remove missing values first
  delays = data.ArrDelay;
  groups = data.(groupVarName);
  notNaN =~isnan(delays);
  groups = groups(notNaN);
  delays = delays(notNaN);
  % Find the unique group levels in this chunk
  [intermKeys,~,idx] = unique(groups, 'stable');
  % Group delays by idx and apply @grpstatsfun function to each group
  intermVals = accumarray(idx,delays,size(intermKeys),@grpstatsfun);
  addmulti(intermKVStore,intermKeys,intermVals);
  function out = grpstatsfun(x)
    n = length(x); % count
    m = sum(x)/n; % mean
    v = sum((x-m).^2)/n; % variance
    s = sum((x-m).^3)/n; % skewness without normalization
    k = sum((x-m).^4)/n; % kurtosis without normalization
    out = {[n, m, v, s, k]};
  end
end
```

More Map Functions

For more information about common programming patterns in map or reduce functions, see “Build Effective Algorithms with MapReduce” on page 12-18.

See Also

`add` | `addmulti` | `mapreduce` | `tabularTextDatastore`

More About

- `KeyValueStore`
- “Write a Reduce Function” on page 12-13

- “Getting Started with MapReduce” on page 12-3

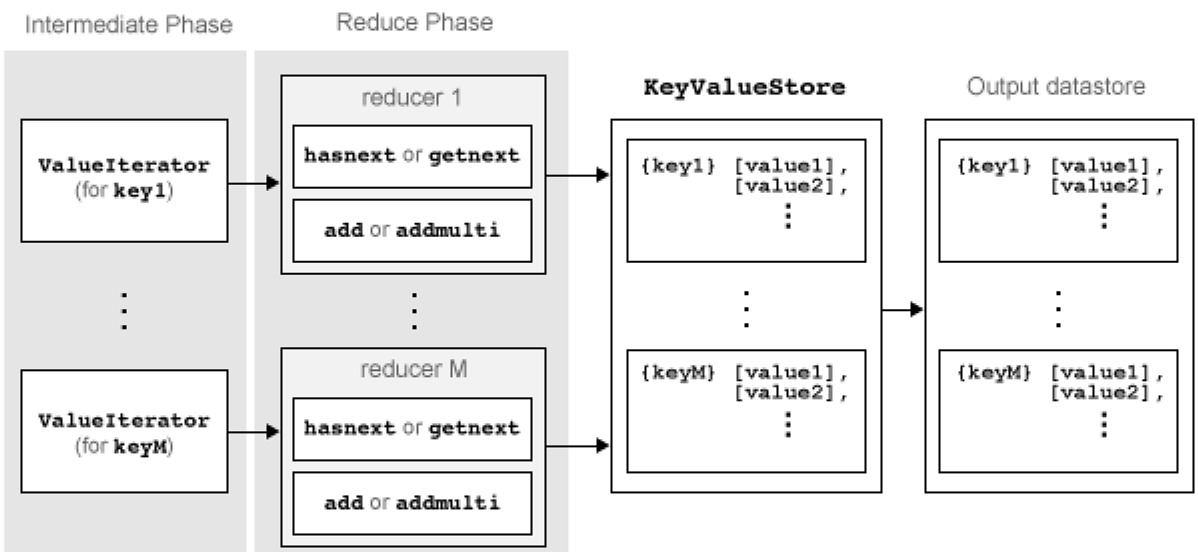
Write a Reduce Function

In this section...
"Role of the Reduce Function in MapReduce" on page 12-13
"Requirements for Reduce Function" on page 12-14
"Sample Reduce Functions" on page 12-14

Role of the Reduce Function in MapReduce

`mapreduce` requires both an input map function that receives blocks of data and that outputs intermediate results, and an input reduce function that reads the intermediate results and produces a final result. Thus, it is normal to break up a calculation into two related pieces for the map and reduce functions to fulfill separately. For example, to find the maximum value in a data set, the map function can find the maximum value in each block of input data, and then the reduce function can find the single maximum value among all of the intermediate maxima.

This figure shows the Reduce phase of the `mapreduce` algorithm.



The Reduce phase of the `mapreduce` algorithm has the following steps:

1. The result of the Map phase of the `mapreduce` algorithm is an intermediate `KeyValueStore` object that contains all of the key-value pairs added by the map function. Before calling the reduce function, `mapreduce` groups the values in the intermediate `KeyValueStore` object by unique key. Each unique key in the intermediate `KeyValueStore` object results in a single call to the reduce function.
2. For each key, `mapreduce` creates a `ValueIterator` object that contains all of the values associated with that key.
3. The reduce function scrolls through the values from the `ValueIterator` object using the `hasnext` and `getnext` functions, which are typically used in a `while` loop.
4. After performing a summary calculation, the reduce function adds one or more key-value pairs to the final `KeyValueStore` object using the `add` and `addmulti` functions.

The Reduce phase of the mapreduce algorithm is complete when the reduce function processes all of the unique intermediate keys and their associated values. The result of this phase of the mapreduce algorithm (similar to the Map phase) is a KeyValueStore object containing all of the final key-value pairs added by the reduce function. After the Reduce phase, mapreduce pulls the key-value pairs from the KeyValueStore and returns them in a datastore (a KeyValueDatastore object by default). The key-value pairs in the output datastore are not in sorted order; they appear in the same order as they were added by the reduce function.

Requirements for Reduce Function

mapreduce automatically calls the reduce function for each unique key in the intermediate KeyValueStore object, so the reduce function must meet certain basic requirements to run properly during these automatic calls. These requirements collectively ensure the proper movement of data through the Reduce phase of the mapreduce algorithm.

The inputs to the reduce function are intermKey, intermValIter, and outKVStore:

- intermKey is one of the unique keys added by the map function. Each call to the reduce function by mapreduce specifies a new unique key from the keys in the intermediate KeyValueStore object.
- intermValIter is the ValueIterator object associated with the active key, intermKey. This ValueIterator object contains all of the values associated with the active key. Scroll through the values using the hasnext and getnext functions.
- outKVStore is the name for the final KeyValueStore object to which the reduce function needs to add key-value pairs. The add and addmulti functions use this object name to add key-value pairs to the output. mapreduce takes the output key-value pairs from outKVStore and returns them in the output datastore, which is a KeyValueDatastore object by default. If the reduce function does not add any key-value pairs to outKVStore, then mapreduce returns an empty datastore.

In addition to these basic requirements for the reduce function, the key-value pairs added by the reduce function must also meet these conditions:

1. Keys must be numeric scalars, character vectors, or strings. Numeric keys cannot be NaN, logical, complex, or sparse.
2. All keys added by the reduce function must have the same class, but that class may differ from the class of the keys added by the map function.
3. If the OutputType argument of mapreduce is 'Binary' (the default), then a value added by the reduce function can be any MATLAB object, including all valid MATLAB data types.
4. If the OutputType argument of mapreduce is 'TabularText', then a value added by the reduce function can be a numeric scalar, character vector, or string. In this case, the value cannot be NaN, complex, logical, or sparse.

Note The above key-value pair requirements may differ when using other products with mapreduce. See the documentation for the appropriate product to get product-specific key-value pair requirements.

Sample Reduce Functions

Here are a few illustrative reduce functions used in mapreduce examples.

Simple Reduce Function

One of the simplest examples of a reducer is maxArrivalDelayReducer, which is the reducer for the example “Find Maximum Value with MapReduce” on page 12-32. The map function in this example finds the maximum arrival delay in each chunk of input data. Then the reduce function finishes the task by finding the single maximum value among all of the intermediate maxima. To find the maximum value, the reducer scrolls through the values in the ValueIterator object and compares each value to the current maximum. mapreduce only calls this reducer function once, since the mapper adds a single unique key to the intermediate KeyValueStore object. The reduce function adds a single key-value pair to the output.

```
function maxArrivalDelayReducer(intermKey, intermValIter, outKVStore)
  % intermKey is 'PartialMaxArrivalDelay'. intermValIter is an iterator of
  % all values that has the key 'PartialMaxArrivalDelay'.
  maxVal = -Inf;
  while hasnext(intermValIter)
    maxVal = max(getnext(intermValIter), maxVal);
  end
  % The key-value pair added to outKVStore will become the output of mapreduce
  add(outKVStore,'MaxArrivalDelay',maxVal);
end
```

Advanced Reduce Function

A more advanced example of a reducer is statsByGroupReducer, which is the reducer for the example “Compute Summary Statistics by Group Using MapReduce” on page 12-61. The map function in this example groups the data in each input using an extra parameter (airline carrier, month, and so on), and then calculates several statistical quantities for each group of data. The reduce function finishes the task by retrieving the statistical quantities and concatenating them into long vectors, and then using the vectors to calculate the final statistical quantities for count, mean, variance, skewness, and kurtosis. The reducer stores these values as fields in a structure, so that each unique key has a structure of statistical quantities in the output.

```
function statsByGroupReducer(intermKey, intermValIter, outKVStore)
% Reducer function for the StatisticsByGroupMapReduceExample.

% Copyright 2014 The MathWorks, Inc.

n = [];
m = [];
v = [];
s = [];
k = [];

% get all sets of intermediate statistics
while hasnext(intermValIter)
    value = getnext(intermValIter);
    n = [n; value(1)];
    m = [m; value(2)];
    v = [v; value(3)];
    s = [s; value(4)];
    k = [k; value(5)];
end
% Note that this approach assumes the concatenated intermediate values fit
% in memory. Refer to the reducer function, covarianceReducer,  of the
% CovarianceMapReduceExample for an alternative pairwise reduction approach

% combine the intermediate results
count = sum(n);
meanVal = sum(n.*m)/count;
d = m - meanVal;
variance = (sum(n.*v) + sum(n.*d.^2))/count;
skewnessVal = (sum(n.*s) + sum(n.*d.*(3*v + d.^2)))./(count*variance^(1.5));
kurtosisVal = (sum(n.*k) + sum(n.*d.*(4*s + 6.*v.*d +d.^3)))./(count*variance^2);
```

```
outValue = struct('Count',count, 'Mean',meanVal, 'Variance',variance,...
                 'Skewness',skewnessVal, 'Kurtosis',kurtosisVal);

% add results to the output datastore
add(outKVStore,intermKey,outValue);
```

More Reduce Functions

For more information about common programming patterns in map or reduce functions, see “Build Effective Algorithms with MapReduce” on page 12-18.

See Also

`add` | `addmulti` | `getnext` | `hasnext` | `mapreduce` | `tabularTextDatastore`

More About

- `KeyValueStore`
- `ValueIterator`
- “Write a Map Function” on page 12-9
- “Getting Started with MapReduce” on page 12-3

Speed Up and Deploy MapReduce Using Other Products

In this section...

Execution Environment

To use `mapreduce` with Parallel Computing Toolbox™, MATLAB Parallel Server™, or MATLAB Compiler™, use the `mapreducer` configuration function to change the execution environment for `mapreduce`. This enables you to start small to verify your map and reduce functions, then quickly scale up to run larger calculations.

Running in Parallel

Parallel Computing Toolbox can immediately speed up your `mapreduce` algorithms by using the full processing power of multicore computers to execute applications with a parallel pool of workers. If you already have Parallel Computing Toolbox installed, then you probably do not need to do anything special to take advantage of these capabilities. For more information about using `mapreduce` with Parallel Computing Toolbox, see "Run mapreduce on a Parallel Pool" (Parallel Computing Toolbox).

MATLAB Parallel Server enables you to run the same applications on a remote computer cluster. For more information, including how to configure MATLAB Parallel Server to support Hadoop clusters, see "Tall Arrays and mapreduce" (Parallel Computing Toolbox).

Application Deployment

MATLAB Compiler enables you to create standalone `mapreduce` applications or deployable archives, which you can share with colleagues or deploy to production Hadoop systems.

For more information, see "MapReduce Applications on Hadoop Clusters" (MATLAB Compiler).

See Also

`gcmr` | `mapreducer`

Build Effective Algorithms with MapReduce

The `mapreduce` example files that ship with MATLAB illustrate different programming techniques. You can use these examples as a starting point to quickly prototype similar `mapreduce` calculations.

Note The associated files for these examples are all in the `toolbox/matlab/demos/` folder.

Example Link	Primary File	Description	Notable Programming Techniques
"Find Maximum Value with MapReduce" on page 12-32	`MaxMapReduceExample.m`	Find maximum arrival delay	One intermediate key and minimal computation.
"Compute Mean Value with MapReduce" on page 12-35	`MeanMapReduceExample.m`	Find mean arrival delay	One intermediate key with intermediate state (accumulating intermediate sum and count).
"Create Histograms Using MapReduce" on page 12-43	`VisualizationMapReduceExample.m`	Visualize data using histograms	Low-volume summaries of data, sufficient to generate a graphic and gain preliminary insights.
"Compute Mean by Group Using MapReduce" on page 12-38	`MeanByGroupMapReduceExample.m`	Compute mean arrival delay for each day of the week	Perform simple computations on subgroups of input data using several intermediate keys.
"Compute Maximum Average HSV of Images with MapReduce" on page 12-78	`HueSaturationValueExample.m`	Determine average maximum hue, saturation, and brightness in an image collection	Analyzes an image datastore using three intermediate keys. The outputs are filenames, which can be used to view the images.
"Simple Data Subsetting Using MapReduce" on page 12-50	`SubsettingMapReduceExample.m`	Create single table from subset of large data set	Extraction of subset of large data set to look for patterns. The procedure is generalized using a parameterized map function to pass in the subsetting criteria.

Example Link	Primary File	Description	Notable Programming Techniques
"Using MapReduce to Compute Covariance and Related Quantities" on page 12-56	`CovarianceMapReduceExample.m`	Compute covariance and related quantities	Calculate several intermediate values and store them with the same key. Use covariance to obtain a correlation matrix and regression coefficients, and to perform principal components analysis.
"Compute Summary Statistics by Group Using MapReduce" on page 12-61	`StatisticsByGroupMapReduceExample.m`	Compute summary statistics organized by group	Use an anonymous function to pass an extra grouping parameter to a parameterized map function. This parameterization allows you to quickly recalculate statistics using different grouping variables.
"Using MapReduce to Fit a Logistic Regression Model" on page 12-67	`LogitMapReduceExample.m`	Fit simple logistic regression model	Chain multiple `mapreduce` calls to carry out an iterative regression algorithm. An anonymous function passes information from one iteration to the next to supply information directly to the map function.
"Tall Skinny QR (TSQR) Matrix Factorization Using MapReduce" on page 12-73	`TSQRMapReduceExample.m`	Tall skinny QR decomposition	Chain multiple `mapreduce` calls to perform multiple iterations of factorizations. Also use the `info` input argument of the map function to compute intermediate numeric keys.

Debug MapReduce Algorithms

This example shows how to debug your `mapreduce` algorithms in MATLAB using a simple example file, `MaxMapReduceExample.m`. Debugging enables you to follow the movement of data between the different phases of `mapreduce` execution and inspect the state of all intermediate variables.

In this section...

Set Breakpoint

Set one or more breakpoints in your map or reduce function files so you can examine the variable values where you think the problem is. For more information, see "Set Breakpoints".

Open the file `maxArrivalDelayMapper.m`.

```
edit maxArrivalDelayMapper.m
```

Set a breakpoint on line 9. This breakpoint causes execution of `mapreduce` to pause right before each call to the map function adds a key-value pair to the intermediate `KeyValueStore` object, named `intermKVStore`.

```
1   function maxArrivalDelayMapper (data, info, intermKVStore)
2   % Mapper function for the MaxMapreduceExample.
3
4   % Copyright 1984-2014 The MathWorks, Inc.
5
6   % Data is an n-by-1 table of the ArrDelay. As the data source is tabular,
7   % the return of read is a table object.
8 - partMax = max(data.ArrDelay);
9 ● add(intermKVStore, 'PartialMaxArrivalDelay',partMax);
```

Execute mapreduce

Run the `mapreduce` example file `MaxMapReduceExample.m`. Specify `mapreducer(0)` to ensure that the algorithm does not run in parallel, since parallel execution of `mapreduce` using Parallel Computing Toolbox ignores breakpoints.

```
mapreducer(0);
MaxMapReduceExample
```

MATLAB stops execution of the file when it encounters the breakpoint in the map function. During the pause in execution, you can hover over the different variable names in the map function, or type one of the variable names at the command line to inspect the values.

In this case, the display indicates that, as yet, there are no key-value pairs in `intermKVStore`.

```
1   function maxArrivalDelayMapper (data, info, intermKVStore)
2   % Mapper function for the MaxMapreduceExample.
3
4   % Copyright 1984-2014 The MathWorks, Inc.
5
6   % Data is an n-by-1 table of the ArrDelay. As the data source is tabular,
7   % the return of read is a table object.
8 - partMax = max(data.ArrDelay);
9   add(intermKVStore, 'PartialMaxArrivalDelay',partMax);
```

```
intermKVStore: 1x1 matlab.mapreduce.KeyValueStore =

KeyValueStore with no key-value pairs.

Keys must be numeric scalars or strings, and values may be any type.

Use add or addmulti to add more key-value pairs.
```

Step Through Map Function

1 Continue past the breakpoint. You can use `dbstep` to execute a single line, or `dbcont` to continue execution until MATLAB encounters another breakpoint. Alternatively, you can click **Step** or **Continue** in the **Editor** tab. For more information about all the available options, see “Debug a MATLAB Program”.

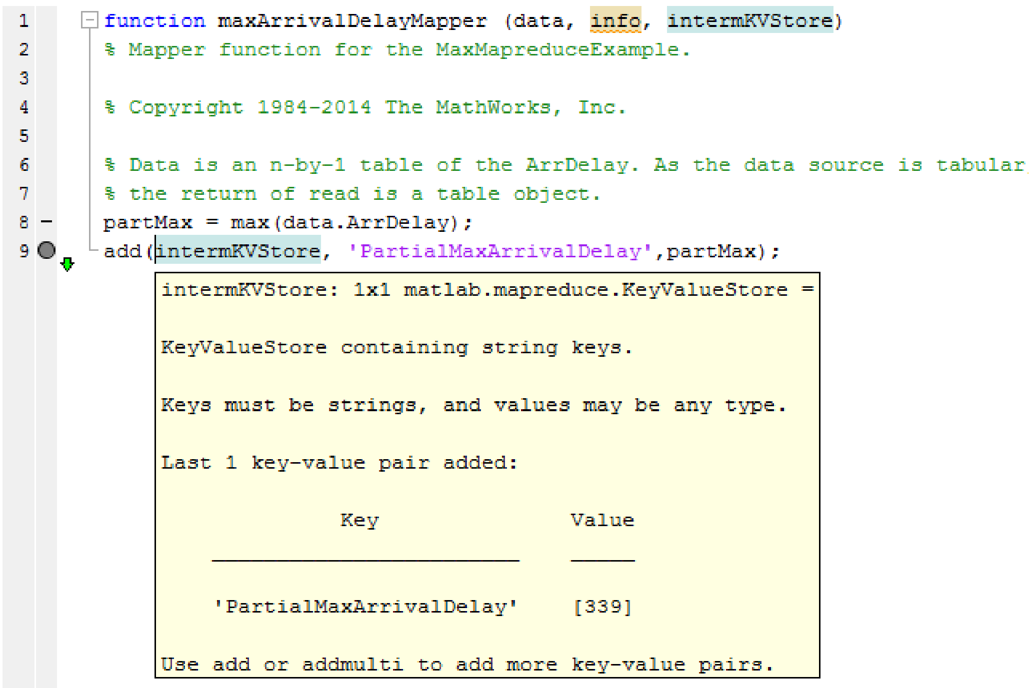
   In this case, use `dbstep` (or click **Step**) to execute only line 9, which adds a key-value pair to `intermKVStore`. Inspect the new display for `intermKVStore`.

```
1   function maxArrivalDelayMapper (data, info, intermKVStore)
2   % Mapper function for the MaxMapreduceExample.
3
4   % Copyright 1984-2014 The MathWorks, Inc.
5
6   % Data is an n-by-1 table of the ArrDelay. As the data source is tabular,
7   % the return of read is a table object.
8 - partMax = max(data.ArrDelay);
9   add(intermKVStore, 'PartialMaxArrivalDelay',partMax);
```

```
intermKVStore: 1x1 matlab.mapreduce.KeyValueStore =

KeyValueStore containing string keys.

Keys must be strings, and values may be any type.

Last 1 key-value pair added:

              Key                  Value
    ________________________    _____

    'PartialMaxArrivalDelay'    [186]

Use add or addmulti to add more key-value pairs.
```

2 Now, use `dbcont` (or click **Continue**) to continue execution of `mapreduce`. During the *next* call to the map function, MATLAB halts again on line 9. The new display for `intermKVStore` indicates that it does not contain any key-value pairs, because the display is meant to show only the *most recent* key-value pairs that are added in the current call to the map (or reduce) function.

3 Step past line 9 again using `dbstep` (or click **Step**) to add the next key-value pair to `intermKVStore`, and inspect the new display for the variable. MATLAB displays only the key-value pair added during the current call to the map function.



4 Complete the debugging of the map function by removing the breakpoint and closing the file `maxArrivalDelayMapper.m`.

Step Through Reduce Function

1 You can use the same process to set breakpoints and step through execution of a reduce function. The reduce function for this example is `maxArrivalDelayReducer.m`. Open this file for editing.

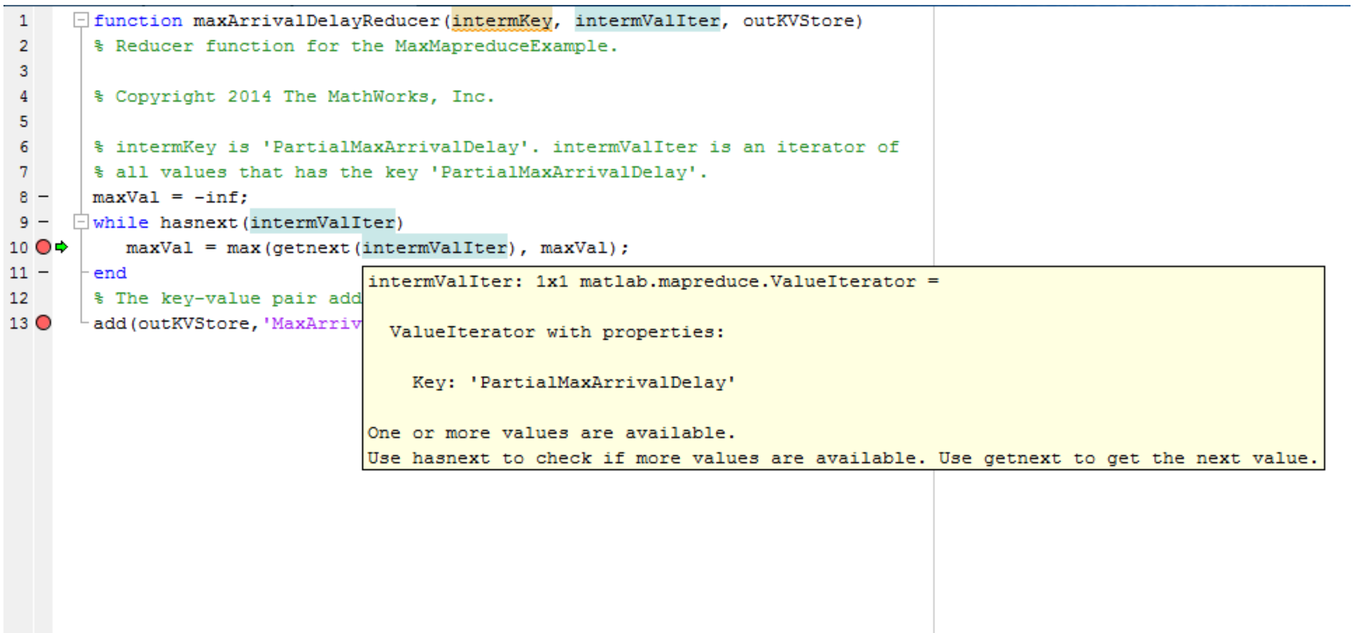
```
edit maxArrivalDelayReducer.m
```

2 Set two breakpoints: one on line 10, and one on line 13. This enables you to inspect the `ValueIterator` and the final key-value pairs added to the output, `outKVStore`.

3 Run the main example file.

```
MaxMapReduceExample
```

4 The execution of the example will pause when the breakpoint on line 10 is encountered. The debug display for the `ValueIterator` indicates the active key and whether any values remain to be retrieved.



5 Now, remove the breakpoint on line 10 and use `dbcont` (or click **Continue**) to continue execution of the example until the next breakpoint is reached (on line 13). Since this reduce function continually compares each new value from the `ValueIterator` to the global maximum, `mapreduce` execution ends by adding a single key-value pair to `outKVStore`.

6 Use `dbstep` (or click **Step**) to execute line 13 only. The display for `outKVStore` shows the global maximum value that `mapreduce` will return as the final answer.

```
function maxArrivalDelayReducer(intermKey, intermValIter, outKVStore)
% Reducer function for the MaxMapreduceExample.

% Copyright 2014 The MathWorks, Inc.

% intermKey is 'PartialMaxArrivalDelay'. intermValIter is an iterator of
% all values that has the key 'PartialMaxArrivalDelay'.
maxVal = -inf;
while hasnext(intermValIter)
    maxVal = max(getnext(intermValIter), maxVal);
end
% The key-value pair added to outKVStore will become the output of mapreduce
add(outKVStore,'MaxArrivalDelay',maxVal);
```

```
outKVStore: 1x1 matlab.mapreduce.KeyValueStore =

KeyValueStore containing string keys.

Keys must be strings, and values may be any type.

Last 1 key-value pair added:

          Key             Value
    _________________     ______

    'MaxArrivalDelay'     [1014]

Use add or addmulti to add more key-value pairs.
```

7 Now use `dbcont` (or click **Continue**) to advance execution, enabling the example to finish running. `mapreduce` returns the final results.

```
Map 100% Reduce 100%

ans =

          Key             Value
    _________________     ______

    'MaxArrivalDelay'     [1014]
```

For a complete guide to debugging in MATLAB, see “Debugging and Analysis”.

See Also

`mapreduce`

More About

- `KeyValueStore`
- `ValueIterator`
- “Getting Started with MapReduce” on page 12-3

Analyze Big Data in MATLAB Using MapReduce

This example shows how to use the `mapreduce` function to process a large amount of file-based data. The MapReduce algorithm is a mainstay of many modern "big data" applications. This example operates on a single computer, but the code can scale up to use Hadoop®.

Throughout this example, the data set is a collection of records from the American Statistical Association for USA domestic airline flights between 1987 and 2008. If you have experimented with "big data" before, you may already be familiar with this data set. A small subset of this data set is included with MATLAB® to allow you to run this and other examples.

Introduction to Datastores

Creating a datastore allows you to access a collection of data in a block-based manner. A datastore can process arbitrarily large amounts of data, and the data can even be spread across multiple files. You can create a datastore for many file types, including a collection of tabular text files (demonstrated here), a SQL database (Database Toolbox™ required) or a Hadoop® Distributed File System (HDFS™).

Create a datastore for a collection of tabular text files and preview the contents.

```
ds = tabularTextDatastore('airlinesmall.csv');
dsPreview = preview(ds);
dsPreview(:,10:15)
```

```
ans=8×6 table
    FlightNum    TailNum    ActualElapsedTime    CRSElapsedTime    AirTime    ArrDelay
    _________    _______    _________________    ______________    _______    ________

      1503       {'NA'}             53                 57          {'NA'}         8
      1550       {'NA'}             63                 56          {'NA'}         8
      1589       {'NA'}             83                 82          {'NA'}        21
      1655       {'NA'}             59                 58          {'NA'}        13
      1702       {'NA'}             77                 72          {'NA'}         4
      1729       {'NA'}             61                 65          {'NA'}        59
      1763       {'NA'}             84                 79          {'NA'}         3
      1800       {'NA'}            155                143          {'NA'}        11
```

The datastore automatically parses the input data and makes a best guess as to the type of data in each column. In this case, use the `'TreatAsMissing'` name-value pair argument to replace the missing values correctly. For numeric variables (such as `'AirTime'`), `tabularTextDatastore` replaces every instance of `'NA'` with a `NaN` value, which is the IEEE arithmetic representation for Not-a-Number.

```
ds = tabularTextDatastore('airlinesmall.csv', 'TreatAsMissing', 'NA');
ds.SelectedFormats{strcmp(ds.SelectedVariableNames, 'TailNum')} = '%s';
ds.SelectedFormats{strcmp(ds.SelectedVariableNames, 'CancellationCode')} = '%s';
dsPreview = preview(ds);
dsPreview(:,{'AirTime','TaxiIn','TailNum','CancellationCode'})
```

```
ans=8×4 table
    AirTime    TaxiIn    TailNum    CancellationCode
    _______    ______    _______    ________________

      NaN       NaN      {'NA'}          {'NA'}
```

```
        NaN         NaN        {'NA'}            {'NA'}
        NaN         NaN        {'NA'}            {'NA'}
        NaN         NaN        {'NA'}            {'NA'}
        NaN         NaN        {'NA'}            {'NA'}
        NaN         NaN        {'NA'}            {'NA'}
        NaN         NaN        {'NA'}            {'NA'}
        NaN         NaN        {'NA'}            {'NA'}
```

Scan for rows of interest

Datastore objects contain an internal pointer to keep track of which block of data the `read` function returns next. Use the `hasdata` and `read` functions to step through the entire data set, and filter the data set to only the rows of interest. In this case, the rows of interest are flights on United Airlines ("UA") departing from Boston ("BOS").

```
subset = [];

while hasdata(ds)
    t = read(ds);
    t = t(strcmp(t.UniqueCarrier, 'UA') & strcmp(t.Origin, 'BOS'), :);
    subset = vertcat(subset, t);
end

subset(1:10,[9,10,15:17])
```

```
ans=10×5 table
    UniqueCarrier    FlightNum    ArrDelay    DepDelay    Origin 
    _____________    _________    ________    ________    _______

       {'UA'}           121          -9           0       {'BOS'}
       {'UA'}          1021          -9          -1       {'BOS'}
       {'UA'}           519          15           8       {'BOS'}
       {'UA'}           354           9           8       {'BOS'}
       {'UA'}           701         -17           0       {'BOS'}
       {'UA'}           673          -9          -1       {'BOS'}
       {'UA'}            91          -3           2       {'BOS'}
       {'UA'}           335          18           4       {'BOS'}
       {'UA'}          1429           1          -2       {'BOS'}
       {'UA'}            53          52          13       {'BOS'}
```

Introduction to `mapreduce`

MapReduce is an algorithmic technique to "divide and conquer" big data problems. In MATLAB, `mapreduce` requires three input arguments:

1. A datastore to read data from
2. A "mapper" function that is given a subset of the data to operate on. The output of the map function is a partial calculation. `mapreduce` calls the mapper function one time for each block in the datastore, with each call operating independently.
3. A "reducer" function that is given the aggregate outputs from the mapper function. The reducer function finishes the computation begun by the mapper function, and outputs the final answer.

This is an over-simplification to some extent, since the output of a call to the mapper function can be shuffled and combined in interesting ways before being passed to the reducer function. This will be examined later in this example.

Use `mapreduce` to perform a computation

A simple use of `mapreduce` is to find the longest flight time in the entire airline data set. To do this:

1. The "mapper" function computes the maximum of each block from the datastore.
2. The "reducer" function then computes the maximum value among all of the maxima computed by the calls to the mapper function.

First, reset the datastore and filter the variables to the one column of interest.

```
reset(ds);
ds.SelectedVariableNames = {'ActualElapsedTime'};
```

Write the mapper function, `maxTimeMapper.m`. It takes three input arguments:

1. The input data, which is a table obtained by applying the `read` function to the datastore.
2. A collection of configuration and contextual information, `info`. This can be ignored in most cases, as it is here.
3. An intermediate data storage object, which records the results of the calculations from the mapper function. Use the `add` function to add Key/Value pairs to this intermediate output. In this example, the name of the key (`'MaxElapsedTime'`) is arbitrary.

Save the following mapper function (`maxTimeMapper.m`) in your current folder.

```
function maxTimeMapper(data, ~, intermKVStore)
  maxElapsedTime = max(data{:,:});
  add(intermKVStore, "MaxElapsedTime", maxElapsedTime)
end
```

Next, write the reducer function. It also takes three input arguments:

1. A set of input "keys". Keys will be discussed further below, but they can be ignored in some simple problems, as they are here.
2. An intermediate data input object that `mapreduce` passes to the reducer function. This data is in the form of Key/Value pairs, and you use the `hasnext` and `getnext` functions to iterate through the values for each key.
3. A final output data storage object. Use the `add` and `addmulti` functions to directly add Key/Value pairs to the output.

Save the following reducer function (`maxTimeReducer.m`) in your current folder.

```
function maxTimeReducer(~, intermValsIter, outKVStore)
  maxElapsedTime = -Inf;
  while(hasnext(intermValsIter))
    maxElapsedTime = max(maxElapsedTime, getnext(intermValsIter));
  end
  add(outKVStore, "MaxElapsedTime", maxElapsedTime);
end
```

Once the mapper and reducer functions are written and saved in your current folder, you can call `mapreduce` using the datastore, mapper function, and reducer function. If you have Parallel Computing Toolbox (PCT), MATLAB will automatically start a pool and parallelize execution. Use the `readall` function to display the results of the MapReduce algorithm.

```
result = mapreduce(ds, @maxTimeMapper, @maxTimeReducer);
```

```
********************************
*      MAPREDUCE PROGRESS      *
********************************
Map   0% Reduce   0%
Map  16% Reduce   0%
Map  32% Reduce   0%
Map  48% Reduce   0%
Map  65% Reduce   0%
Map  81% Reduce   0%
Map  97% Reduce   0%
Map 100% Reduce   0%
Map 100% Reduce 100%
```

```
readall(result)
```

```
ans=1×2 table
           Key              Value  
    __________________    ________

    {'MaxElapsedTime'}    {[1650]}
```

Use of keys in mapreduce

The use of keys is an important and powerful feature of `mapreduce`. Each call to the mapper function adds intermediate results to one or more named "buckets", called keys. The number of calls to the mapper function by `mapreduce` corresponds to the number of blocks in the datastore.

If the mapper function adds values to multiple keys, this leads to multiple calls to the reducer function, with each call working on only one key's intermediate values. The `mapreduce` function automatically manages this data movement between the map and reduce phases of the algorithm.

This flexibility is useful in many contexts. The example below uses keys in a relatively obvious way for illustrative purposes.

Calculating group-wise metrics with mapreduce

The behavior of the mapper function in this application is more complex. For every flight carrier found in the input data, use the `add` function to add a vector of values. This vector is a count of the number of flights for that carrier on each day in the 21+ years of data. The carrier code is the key for this vector of values. This ensures that all of the data for each carrier will be grouped together when `mapreduce` passes it to the reducer function.

Save the following mapper function (`countFlightsMapper.m`) in your current folder.

```
function countFlightsMapper(data, ~, intermKVStore)
  dayNumber = days((datetime(data.Year, data.Month, data.DayofMonth) - datetime(1987,10,1)))+1;
  daysSinceEpoch = days(datetime(2008,12,31) - datetime(1987,10,1))+1;
  [airlineName, ~, airlineIndex] = unique(data.UniqueCarrier, 'stable');

  for i = 1:numel(airlineName)
    dayTotals = accumarray(dayNumber(airlineIndex==i), 1, [daysSinceEpoch, 1]);
    add(intermKVStore, airlineName{i}, dayTotals);
  end
end
```

The reducer function is less complex. It simply iterates over the intermediate values and adds the vectors together. At completion, it outputs the values in this aggregate vector. Note that the reducer function does not need to sort or examine the `intermediateKeysIn` values; each call to the reducer function by `mapreduce` only passes the values for one airline carrier.

Save the following reducer function (`countFlightsReducer.m`) in your current folder.

```
function countFlightsReducer(intermKeysIn, intermValsIter, outKVStore)
  daysSinceEpoch = days(datetime(2008,12,31) - datetime(1987,10,1))+1;
  dayArray = zeros(daysSinceEpoch, 1);

  while hasnext(intermValsIter)
    dayArray = dayArray + getnext(intermValsIter);
  end
  add(outKVStore, intermKeysIn, dayArray);
end
```

Reset the datastore and select the variables of interest. Once the mapper and reducer functions are written and saved in your current folder, you can call `mapreduce` using the datastore, mapper function, and reducer function.

```
reset(ds);
ds.SelectedVariableNames = {'Year', 'Month', 'DayofMonth', 'UniqueCarrier'};
result = mapreduce(ds, @countFlightsMapper, @countFlightsReducer);
```

```
********************************
*      MAPREDUCE PROGRESS      *
********************************
Map   0% Reduce   0%
Map  16% Reduce   0%
Map  32% Reduce   0%
Map  48% Reduce   0%
Map  65% Reduce   0%
Map  81% Reduce   0%
Map  97% Reduce   0%
Map 100% Reduce   0%
Map 100% Reduce  10%
Map 100% Reduce  21%
Map 100% Reduce  31%
Map 100% Reduce  41%
Map 100% Reduce  52%
Map 100% Reduce  62%
Map 100% Reduce  72%
Map 100% Reduce  83%
Map 100% Reduce  93%
Map 100% Reduce 100%
```

```
result = readall(result);
```

In case this example was run with only the sample data set, load the results of the `mapreduce` algorithm run on the entire data set.

```
load airlineResults
```

Visualizing the results

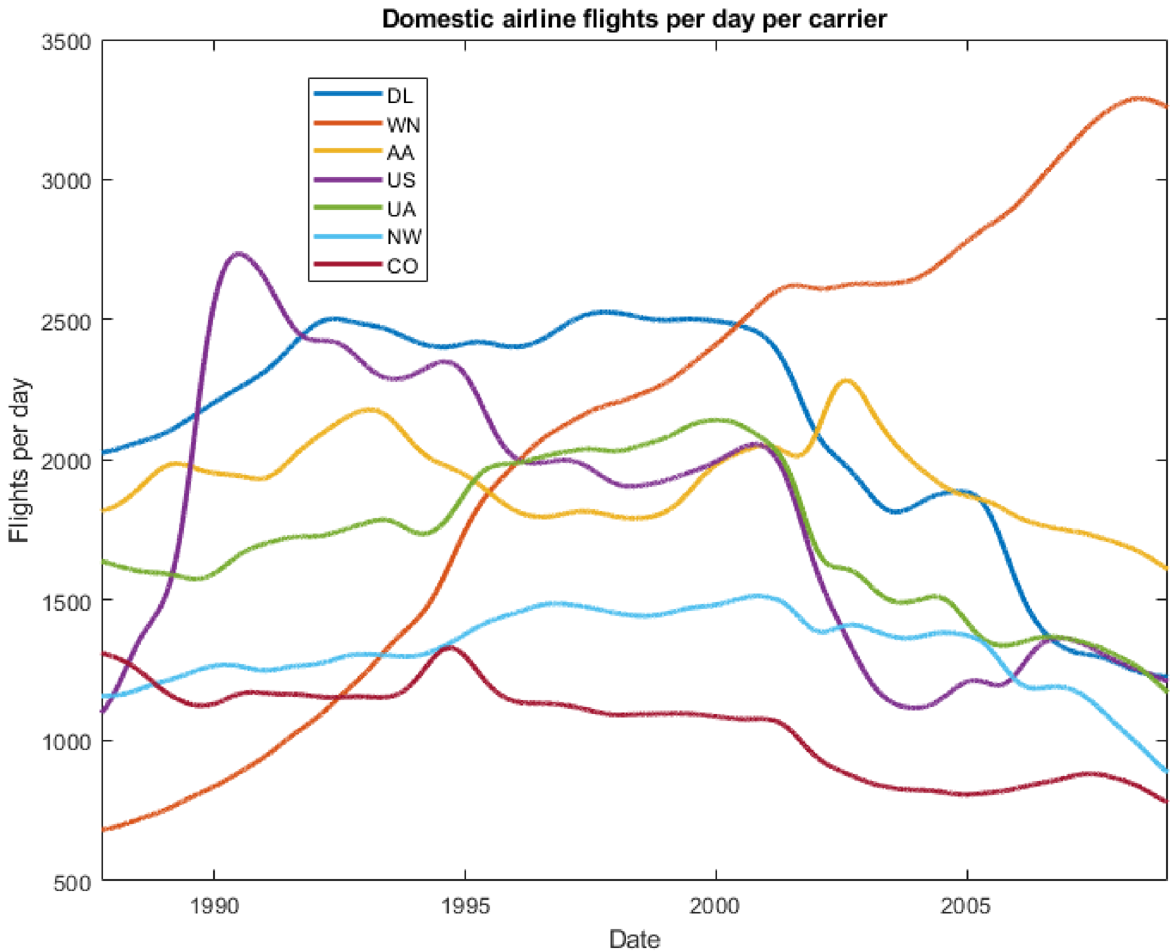
Using only the top 7 carriers, smooth the data to remove the effects of weekend travel. This would otherwise clutter the visualization.

```
lines = result.Value;
lines = horzcat(lines{:});
```

```
[~,sortOrder] = sort(sum(lines), 'descend');
lines = lines(:,sortOrder(1:7));
result = result(sortOrder(1:7),:);

lines(lines==0) = nan;
lines = smoothdata(lines,'gaussian');
```

Plot the data.

```
figure('Position',[1 1 800 600]);
plot(datetime(1987,10,1):caldays(1):datetime(2008,12,31),lines,'LineWidth',2)
title ('Domestic airline flights per day per carrier')
xlabel('Date')
ylabel('Flights per day')
legend(result.Key, 'Location', 'Best')
```



The plot shows the emergence of Southwest Airlines (WN) during this time period.

Learning more

This example only scratches the surface of what is possible with `mapreduce`. See the documentation for `mapreduce` for more information, including information on using it with Hadoop and MATLAB® Parallel Server™.

Local Functions

Listed here are the local functions that `mapreduce` applies to the data.

```
function maxTimeMapper(data, ~, intermKVStore)
  maxElapsedTime = max(data{:,:});
  add(intermKVStore, "MaxElapsedTime", maxElapsedTime)
end
%-----------------------------------------------------------------------
function maxTimeReducer(~, intermValsIter, outKVStore)
  maxElapsedTime = -Inf;
  while(hasnext(intermValsIter))
    maxElapsedTime = max(maxElapsedTime, getnext(intermValsIter));
  end
  add(outKVStore, "MaxElapsedTime", maxElapsedTime);
end
%-----------------------------------------------------------------------
function countFlightsMapper(data, ~, intermKVStore)
  dayNumber = days((datetime(data.Year, data.Month, data.DayofMonth) - datetime(1987,10,1)))+1;
  daysSinceEpoch = days(datetime(2008,12,31) - datetime(1987,10,1))+1;
  [airlineName, ~, airlineIndex] = unique(data.UniqueCarrier, 'stable');

  for i = 1:numel(airlineName)
    dayTotals = accumarray(dayNumber(airlineIndex==i), 1, [daysSinceEpoch, 1]);
    add(intermKVStore, airlineName{i}, dayTotals);
  end
end
%-----------------------------------------------------------------------
function countFlightsReducer(intermKeysIn, intermValsIter, outKVStore)
  daysSinceEpoch = days(datetime(2008,12,31) - datetime(1987,10,1))+1;
  dayArray = zeros(daysSinceEpoch, 1);

  while hasnext(intermValsIter)
    dayArray = dayArray + getnext(intermValsIter);
  end
  add(outKVStore, intermKeysIn, dayArray);
end
%-----------------------------------------------------------------------
```

See Also

`mapreduce` | `tabularTextDatastore`

More About

- “Getting Started with MapReduce” on page 12-3
- “Build Effective Algorithms with MapReduce” on page 12-18

Find Maximum Value with MapReduce

This example shows how to find the maximum value of a single variable in a data set using `mapreduce`. It demonstrates the simplest use of `mapreduce` since there is only one key and minimal computation.

Prepare Data

Create a datastore using the `airlinesmall.csv` data set. This 12-megabyte data set contains 29 columns of flight information for several airline carriers, including arrival and departure times. In this example, select `ArrDelay` (flight arrival delay) as the variable of interest.

```
ds = tabularTextDatastore('airlinesmall.csv', 'TreatAsMissing', 'NA');
ds.SelectedVariableNames = 'ArrDelay';
```

The datastore treats `'NA'` values as missing, and replaces the missing values with `NaN` values by default. Additionally, the `SelectedVariableNames` property allows you to work with only the selected variable of interest, which you can verify using `preview`.

```
preview(ds)
```

```
ans=8×1 table
    ArrDelay
    ________

        8
        8
       21
       13
        4
       59
        3
       11
```

Run MapReduce

The `mapreduce` function requires a map function and a reduce function as inputs. The mapper receives blocks of data and outputs intermediate results. The reducer reads the intermediate results and produces a final result.

In this example, the mapper finds the maximum arrival delay in each block of data. The mapper then stores these maximum values as the intermediate values associated with the key `'PartialMaxArrivalDelay'`.

Display the map function file.

```
function maxArrivalDelayMapper (data, info, intermKVStore)
  partMax = max(data.ArrDelay);
  add(intermKVStore, 'PartialMaxArrivalDelay',partMax);
end
```

The reducer receives a list of the maximum arrival delays for each block and finds the overall maximum arrival delay from the list of values. `mapreduce` only calls this reducer once, since the mapper only adds a single unique key. The reducer uses `add` to add a final key-value pair to the output.

Display the reduce function file.

```
function maxArrivalDelayReducer(intermKey, intermValIter, outKVStore)
  % intermKey is 'PartialMaxArrivalDelay'. intermValIter is an iterator of
  % all values that has the key 'PartialMaxArrivalDelay'.
  maxVal = -Inf;
  while hasnext(intermValIter)
    maxVal = max(getnext(intermValIter), maxVal);
  end
  % The key-value pair added to outKVStore will become the output of mapreduce
  add(outKVStore,'MaxArrivalDelay',maxVal);
end
```

Use `mapreduce` to apply the map and reduce functions to the datastore, `ds`.

```
maxDelay = mapreduce(ds, @maxArrivalDelayMapper, @maxArrivalDelayReducer);
```

```
********************************
*      MAPREDUCE PROGRESS      *
********************************
Map   0% Reduce   0%
Map  16% Reduce   0%
Map  32% Reduce   0%
Map  48% Reduce   0%
Map  65% Reduce   0%
Map  81% Reduce   0%
Map  97% Reduce   0%
Map 100% Reduce   0%
Map 100% Reduce 100%
```

`mapreduce` returns a datastore, `maxDelay`, with files in the current folder.

Read the final result from the output datastore, `maxDelay`.

```
readall(maxDelay)
```

```
ans=1×2 table
            Key               Value
    ___________________    ________

    {'MaxArrivalDelay'}    {[1014]}
```

Local Functions

Listed here are the map and reduce functions that `mapreduce` applies to the data.

```
function maxArrivalDelayMapper (data, info, intermKVStore)
  partMax = max(data.ArrDelay);
  add(intermKVStore, 'PartialMaxArrivalDelay',partMax);
end
%----------------------------------------------------------------------
function maxArrivalDelayReducer(intermKey, intermValIter, outKVStore)
  % intermKey is 'PartialMaxArrivalDelay'. intermValIter is an iterator of
  % all values that has the key 'PartialMaxArrivalDelay'.
  maxVal = -Inf;
  while hasnext(intermValIter)
    maxVal = max(getnext(intermValIter), maxVal);
  end
```

```
  % The key-value pair added to outKVStore will become the output of mapreduce
  add(outKVStore,'MaxArrivalDelay',maxVal);
end
%--------------------------------------------------------------------
```

See Also

mapreduce | tabularTextDatastore

More About

- “Getting Started with MapReduce” on page 12-3
- “Build Effective Algorithms with MapReduce” on page 12-18

Compute Mean Value with MapReduce

This example shows how to compute the mean of a single variable in a data set using `mapreduce`. It demonstrates a simple use of `mapreduce` with one key, minimal computation, and an intermediate state (accumulating intermediate sum and count).

Prepare Data

Create a datastore using the `airlinesmall.csv` data set. This 12-megabyte data set contains 29 columns of flight information for several airline carriers, including arrival and departure times. In this example, select `ArrDelay` (flight arrival delay) as the variable of interest.

```
ds = tabularTextDatastore('airlinesmall.csv', 'TreatAsMissing', 'NA');
ds.SelectedVariableNames = 'ArrDelay';
```

The datastore treats `'NA'` values as missing, and replaces the missing values with `NaN` values by default. Additionally, the `SelectedVariableNames` property allows you to work with only the selected variable of interest, which you can verify using `preview`.

```
preview(ds)
```

```
ans=8×1 table
    ArrDelay
    ________

       8
       8
      21
      13
       4
      59
       3
      11
```

Run MapReduce

The `mapreduce` function requires a map function and a reduce function as inputs. The mapper receives blocks of data and outputs intermediate results. The reducer reads the intermediate results and produces a final result.

In this example, the mapper finds the count and sum of the arrival delays in each block of data. The mapper then stores these values as the intermediate values associated with the key `"PartialCountSumDelay"`.

Display the map function file.

```
function meanArrivalDelayMapper (data, info, intermKVStore)
  % Data is an n-by-1 table of the ArrDelay. Remove missing values first:
  data(isnan(data.ArrDelay),:) = [];

  % Record the partial counts and sums and the reducer will accumulate them.
  partCountSum = [length(data.ArrDelay), sum(data.ArrDelay)];
  add(intermKVStore, "PartialCountSumDelay",partCountSum);
end
```

The reducer accepts the count and sum for each block stored by the mapper. It sums up the values to obtain the total count and total sum. The overall mean arrival delay is a simple division of the values.

`mapreduce` only calls this reducer once, since the mapper only adds a single unique key. The reducer uses `add` to add a single key-value pair to the output.

Display the reduce function file.

```
function meanArrivalDelayReducer(intermKey, intermValIter, outKVStore)
  count = 0;
  sum = 0;
  while hasnext(intermValIter)
    countSum = getnext(intermValIter);
    count = count + countSum(1);
    sum = sum + countSum(2);
  end
  meanDelay = sum/count;

  % The key-value pair added to outKVStore will become the output of mapreduce
  add(outKVStore,"MeanArrivalDelay",meanDelay);
end
```

Use `mapreduce` to apply the map and reduce functions to the datastore, `ds`.

```
meanDelay = mapreduce(ds, @meanArrivalDelayMapper, @meanArrivalDelayReducer);
```

```
********************************
*      MAPREDUCE PROGRESS      *
********************************
Map   0% Reduce   0%
Map  16% Reduce   0%
Map  32% Reduce   0%
Map  48% Reduce   0%
Map  65% Reduce   0%
Map  81% Reduce   0%
Map  97% Reduce   0%
Map 100% Reduce   0%
Map 100% Reduce 100%
```

`mapreduce` returns a datastore, `meanDelay`, with files in the current folder.

Read the final result from the output datastore, `meanDelay`.

```
readall(meanDelay)
```

```
ans=1×2 table
            Key                Value
    ____________________    __________

    {'MeanArrivalDelay'}    {[7.1201]}
```

Local Functions

Listed here are the map and reduce functions that `mapreduce` applies to the data.

```
function meanArrivalDelayMapper (data, info, intermKVStore)
  % Data is an n-by-1 table of the ArrDelay. Remove missing values first:
  data(isnan(data.ArrDelay),:) = [];

  % Record the partial counts and sums and the reducer will accumulate them.
  partCountSum = [length(data.ArrDelay), sum(data.ArrDelay)];
```

```
  add(intermKVStore, "PartialCountSumDelay",partCountSum);
end
%-------------------------------------------------------------------------
function meanArrivalDelayReducer(intermKey, intermValIter, outKVStore)
  count = 0;
  sum = 0;
  while hasnext(intermValIter)
    countSum = getnext(intermValIter);
    count = count + countSum(1);
    sum = sum + countSum(2);
  end
  meanDelay = sum/count;

  % The key-value pair added to outKVStore will become the output of mapreduce
  add(outKVStore,"MeanArrivalDelay",meanDelay);
end
%-------------------------------------------------------------------------
```

See Also

mapreduce | tabularTextDatastore

More About

- "Getting Started with MapReduce" on page 12-3
- "Build Effective Algorithms with MapReduce" on page 12-18

Compute Mean by Group Using MapReduce

This example shows how to compute the mean by group in a data set using `mapreduce`. It demonstrates how to do computations on subgroups of data.

Prepare Data

Create a datastore using the `airlinesmall.csv` data set. This 12-megabyte data set contains 29 columns of flight information for several airline carriers, including arrival and departure times. In this example, select `DayOfWeek` and `ArrDelay` (flight arrival delay) as the variables of interest.

```
ds = tabularTextDatastore('airlinesmall.csv', 'TreatAsMissing', 'NA');
ds.SelectedVariableNames = {'ArrDelay', 'DayOfWeek'};
```

The datastore treats `'NA'` values as missing, and replaces the missing values with `NaN` values by default. Additionally, the `SelectedVariableNames` property allows you to work with only the selected variables of interest, which you can verify using `preview`.

```
preview(ds)
```

```
ans=8×2 table
    ArrDelay    DayOfWeek
    ________    _________

        8           3    
        8           1    
       21           5    
       13           5    
        4           4    
       59           3    
        3           4    
       11           6    
```

Run MapReduce

The `mapreduce` function requires a map function and a reduce function as inputs. The mapper receives blocks of data and outputs intermediate results. The reducer reads the intermediate results and produces a final result.

In this example, the mapper computes the count and sum of delays by the day of week in each block of data, and then stores the results as intermediate key-value pairs. The keys are integers (1 to 7) representing the days of the week and the values are two-element vectors representing the count and sum of the delay of each day.

Display the map function file.

```
function meanArrivalDelayByDayMapper(data, ~, intermKVStore)
  % Data is an n-by-2 table: first column is the DayOfWeek and the second
  % is the ArrDelay. Remove missing values first.
  delays = data.ArrDelay;
  day = data.DayOfWeek;
  notNaN = ~isnan(delays);
  day = day(notNaN);
  delays = delays(notNaN);

  % find the unique days in this chunk
```

```
  [intermKeys,~,idx] = unique(day, 'stable');

  % group delays by idx and apply @grpstatsfun function to each group
  intermVals = accumarray(idx,delays,size(intermKeys),@countsum);
  addmulti(intermKVStore,intermKeys,intermVals);

  function out = countsum(x)
    n = length(x); % count
    s = sum(x); % mean
    out = {[n, s]};
  end
end
```

After the Map phase, `mapreduce` groups the intermediate key-value pairs by unique key (in this case, day of the week). Thus, each call to the reducer works on the values associated with one day of the week. The reducer receives a list of the intermediate count and sum of delays for the day specified by the input key (`intermKey`) and sums up the values into the total count, `n` and total sum `s`. Then, the reducer calculates the overall mean, and adds one final key-value pair to the output. This key-value pair represents the mean flight arrival delay for one day of the week.

Display the reduce function file.

```
function meanArrivalDelayByDayReducer(intermKey, intermValIter, outKVStore)
  n = 0;
  s = 0;

  % get all sets of intermediate results
  while hasnext(intermValIter)
    intermValue = getnext(intermValIter);
    n = n + intermValue(1);
    s = s + intermValue(2);
  end

  % accumulate the sum and count
  mean = s/n;
  % add results to the output datastore
  add(outKVStore,intermKey,mean);
end
```

Use `mapreduce` to apply the map and reduce functions to the datastore, `ds`.

```
meanDelayByDay = mapreduce(ds, @meanArrivalDelayByDayMapper, ...
                               @meanArrivalDelayByDayReducer);
```

```
********************************
*      MAPREDUCE PROGRESS      *
********************************
Map   0% Reduce   0%
Map  16% Reduce   0%
Map  32% Reduce   0%
Map  48% Reduce   0%
Map  65% Reduce   0%
Map  81% Reduce   0%
Map  97% Reduce   0%
Map 100% Reduce   0%
Map 100% Reduce  14%
Map 100% Reduce  29%
Map 100% Reduce  43%
```

```
Map 100% Reduce  57%
Map 100% Reduce  71%
Map 100% Reduce  86%
Map 100% Reduce 100%
```

`mapreduce` returns a datastore, `meanDelayByDay`, with files in the current folder.

Read the final result from the output datastore, `meanDelayByDay`.

```
result = readall(meanDelayByDay)
```

```
result=7×2 table
    Key      Value
    ___    __________

     3     {[7.0038]}
     1     {[7.0833]}
     5     {[9.4193]}
     4     {[9.3185]}
     6     {[4.2095]}
     2     {[5.8569]}
     7     {[6.5241]}
```

Organize Results

The integer keys (1 to 7) represent the days of the week. To organize the results more, convert the keys to a categorical array, retrieve the numeric values from the single element cells, and rename the variable names of the resulting table.

```
result.Key = categorical(result.Key, 1:7, ...
               {'Mon','Tue','Wed','Thu','Fri','Sat','Sun'});
result.Value = cell2mat(result.Value);
result.Properties.VariableNames = {'DayOfWeek', 'MeanArrDelay'}
```

```
result=7×2 table
    DayOfWeek    MeanArrDelay
    _________    ____________

       Wed          7.0038
       Mon          7.0833
       Fri          9.4193
       Thu          9.3185
       Sat          4.2095
       Tue          5.8569
       Sun          6.5241
```

Sort the rows of the table by mean flight arrival delay. This reveals that Saturday is the best day of the week to travel, whereas Friday is the worst.

```
result = sortrows(result,'MeanArrDelay')
```

```
result=7×2 table
    DayOfWeek    MeanArrDelay
    _________    ____________

       Sat          4.2095
       Tue          5.8569
```

```
    Sun          6.5241
    Wed          7.0038
    Mon          7.0833
    Thu          9.3185
    Fri          9.4193
```

Local Functions

Listed here are the map and reduce functions that `mapreduce` applies to the data.

```
function meanArrivalDelayByDayMapper(data, ~, intermKVStore)
  % Data is an n-by-2 table: first column is the DayOfWeek and the second
  % is the ArrDelay. Remove missing values first.
  delays = data.ArrDelay;
  day = data.DayOfWeek;
  notNaN = ~isnan(delays);
  day = day(notNaN);
  delays = delays(notNaN);

  % find the unique days in this chunk
  [intermKeys,~,idx] = unique(day, 'stable');

  % group delays by idx and apply @grpstatsfun function to each group
  intermVals = accumarray(idx,delays,size(intermKeys),@countsum);
  addmulti(intermKVStore,intermKeys,intermVals);

  function out = countsum(x)
    n = length(x); % count
    s = sum(x); % mean
    out = {[n, s]};
  end
end
%-------------------------------------------------------------------------
function meanArrivalDelayByDayReducer(intermKey, intermValIter, outKVStore)
  n = 0;
  s = 0;

  % get all sets of intermediate results
  while hasnext(intermValIter)
    intermValue = getnext(intermValIter);
    n = n + intermValue(1);
    s = s + intermValue(2);
  end

  % accumulate the sum and count
  mean = s/n;
  % add results to the output datastore
  add(outKVStore,intermKey,mean);
end
%-------------------------------------------------------------------------
```

See Also
`mapreduce` | `tabularTextDatastore`

More About

- "Getting Started with MapReduce" on page 12-3
- "Build Effective Algorithms with MapReduce" on page 12-18

Create Histograms Using MapReduce

This example shows how to visualize patterns in a large data set without having to load all of the observations into memory simultaneously. It demonstrates how to compute lower volume summaries of the data that are sufficient to generate a graphic.

Histograms are a common visualization technique that give an empirical estimate of the probability density function (pdf) of a variable. Histograms are well-suited to a big data environment, because they can reduce the size of raw input data to a vector of counts. Each count is the number of observations that falls within each of a set of contiguous, numeric intervals or bins.

The `mapreduce` function computes counts separately on multiple blocks of the data. Then `mapreduce` sums the counts from all blocks. The map function and reduce function are both extremely simple in this example. Nevertheless, you can build flexible visualizations with the summary information that they collect.

Prepare Data

Create a datastore using the `airlinesmall.csv` data set. This 12-megabyte data set contains 29 columns of flight information for several airline carriers, including arrival and departure times. In this example, select `ArrDelay` (flight arrival delay) as the variable of interest.

```
ds = tabularTextDatastore('airlinesmall.csv', 'TreatAsMissing', 'NA');
ds.SelectedVariableNames = 'ArrDelay';
```

The datastore treats `'NA'` values as missing, and replaces the missing values with `NaN` values by default. Additionally, the `SelectedVariableNames` property allows you to work with only the selected variable of interest, which you can verify using `preview`.

```
preview(ds)
```

```
ans=8×1 table
    ArrDelay
    ________

        8
        8
       21
       13
        4
       59
        3
       11
```

Run MapReduce

The `mapreduce` function requires a map function and a reduce function as inputs. The mapper receives blocks of data and outputs intermediate results. The reducer reads the intermediate results and produces a final result.

In this example, the mapper collects the counts of flights with various amounts of arrival delay by accumulating the arrival delays into bins. The bins are defined by the fourth input argument to the map function, `edges`.

Display the map function file.

```
function visualizationMapper(data, ~, intermKVStore, edges)
  % Count how many flights have arrival delay in each interval specified by
  % the EDGES vector, and add these counts to INTERMKVSTORE.
  counts = histc(data.ArrDelay, edges);
  add(intermKVStore, 'Null', counts);
end
```

The bin size of the histogram is important. Bins that are too wide can obscure important details in the data set. Bins that are too narrow can lead to a noisy histogram. When working with very large data sets, it is best to avoid making multiple passes over the data to try out different bin widths. A simple way to avoid making multiple passes is to collect counts with bins that are narrow. Then, to get wider bins, you can aggregate adjacent bin counts without reprocessing the raw data. The flight arrival delays are reported in 1-minute increments, so define 1-minute bins from -60 minutes to 599 minutes.

```
edges = -60:599;
```

Create an anonymous function to configure the map function to use the bin edges. The anonymous function allows you to specialize the map function by specifying a particular value for its fourth input argument. Then, you can call the map function via the anonymous function, using only the three input arguments that the `mapreduce` function expects.

```
ourVisualizationMapper = ...
    @(data, info, intermKVstore) visualizationMapper(data, info, intermKVstore, edges);
```

Display the reduce function file. The reducer sums the counts stored by the mapper.

```
function visualizationReducer(~, intermValList, outKVStore)
  if hasnext(intermValList)
    outVal = getnext(intermValList);
  else
    outVal = [];
  end
  while hasnext(intermValList)
    outVal = outVal + getnext(intermValList);
  end
  add(outKVStore, 'Null', outVal);
end
```

Use `mapreduce` to apply the map and reduce functions to the datastore, `ds`.

```
result = mapreduce(ds, ourVisualizationMapper, @visualizationReducer);
```

```
********************************
*      MAPREDUCE PROGRESS      *
********************************
Map   0% Reduce   0%
Map  16% Reduce   0%
Map  32% Reduce   0%
Map  48% Reduce   0%
Map  65% Reduce   0%
Map  81% Reduce   0%
Map  97% Reduce   0%
Map 100% Reduce   0%
Map 100% Reduce 100%
```

`mapreduce` returns an output datastore, `result`, with files in the current folder.

Organize Results

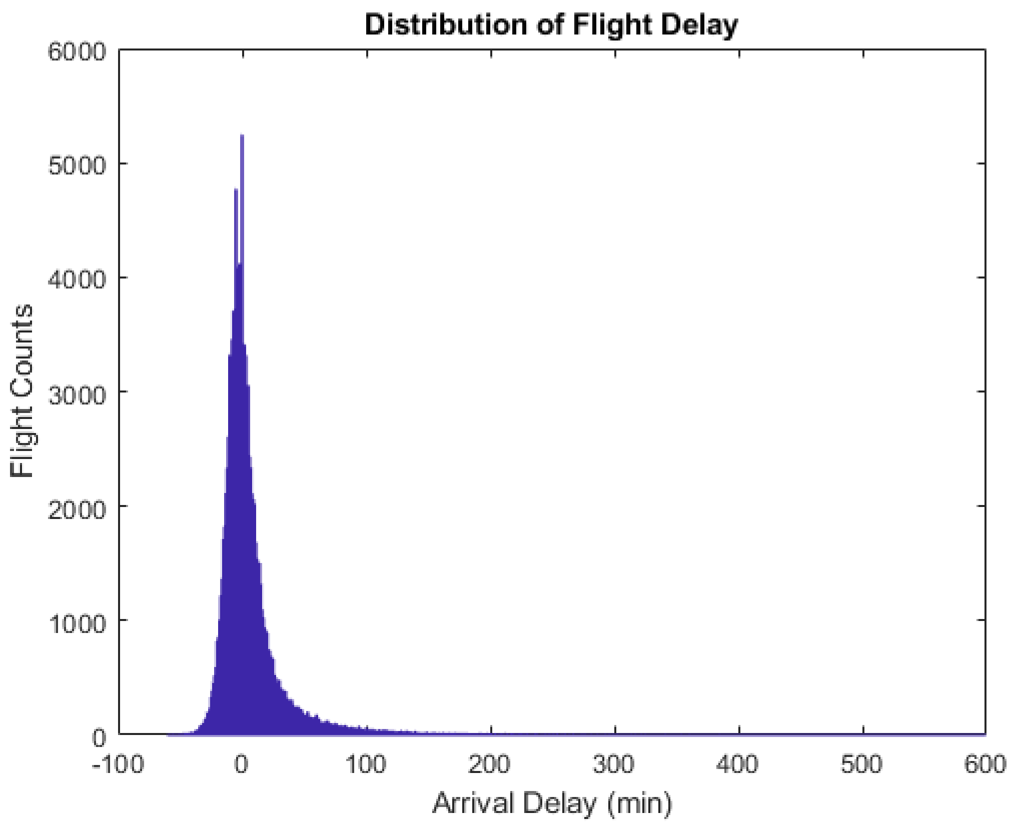
Read the final bin count results from the output datastore.

```
r = readall(result);
counts = r.Value{1};
```

Visualize Results

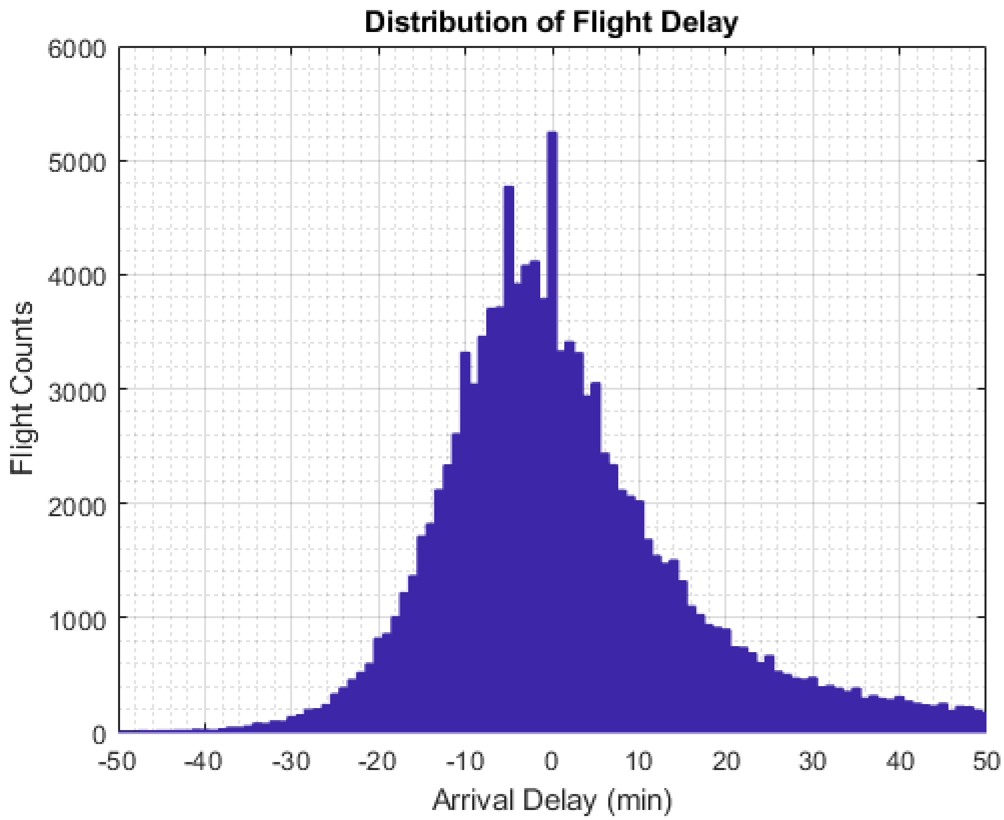
Plot the raw bin counts using the whole range of the data (apart from a few outliers excluded by the mapper).

```
bar(edges, counts, 'hist');
title('Distribution of Flight Delay')
xlabel('Arrival Delay (min)')
ylabel('Flight Counts')
```



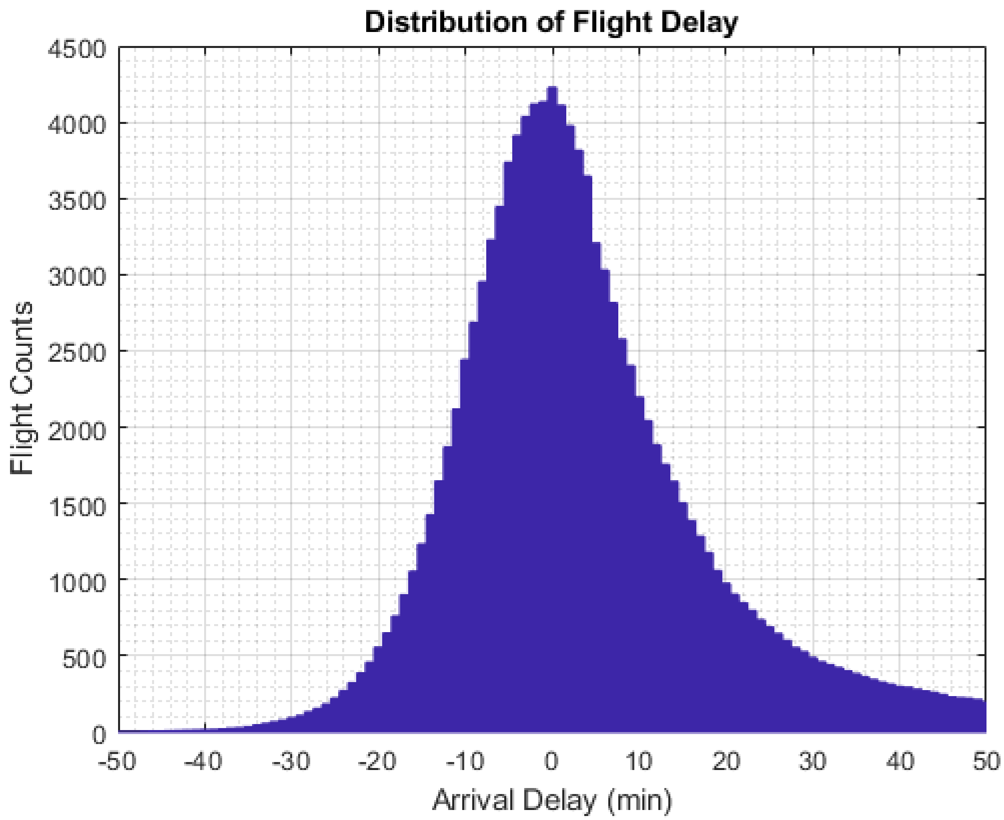
The histogram has long tails. Look at a restricted bin range to better visualize the delay distribution of the majority of flights. Zooming in a bit reveals there is a reporting artifact; it is common to round delays to 5-minute increments.

```
xlim([-50,50]);
grid on
grid minor
```



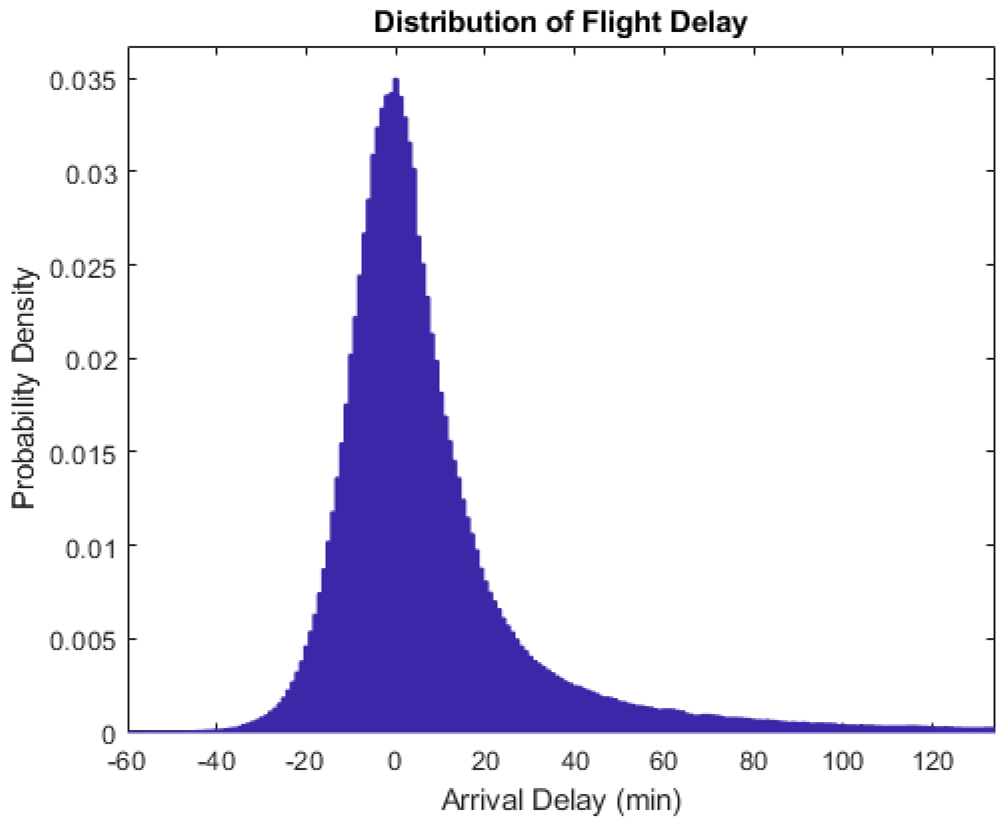
Smooth the counts with a moving average filter to remove the 5-minute recording artifact.

```
smoothCounts = filter( (1/5)*ones(1,5), 1, counts);
figure
bar(edges, smoothCounts, 'hist')
xlim([-50,50]);
title('Distribution of Flight Delay')
xlabel('Arrival Delay (min)')
ylabel('Flight Counts')
grid on
grid minor
```



To give the graphic a better balance, do not display the top 1% of most-delayed flights. You can tailor the visualization in many ways without reprocessing the complete data set, assuming that you collected the appropriate information during the full pass through the data.

```
empiricalCDF = cumsum(counts);
empiricalCDF = empiricalCDF / empiricalCDF(end);
quartile99 = find(empiricalCDF>0.99, 1, 'first');
low99 = 1:quartile99;

figure
empiricalPDF = smoothCounts(low99) / sum(smoothCounts);
bar(edges(low99), empiricalPDF, 'hist');

xlim([-60,edges(quartile99)]);
ylim([0, max(empiricalPDF)*1.05]);
title('Distribution of Flight Delay')
xlabel('Arrival Delay (min)')
ylabel('Probability Density')
```



Local Functions

Listed here are the map and reduce functions that `mapreduce` applies to the data.

```
function visualizationMapper(data, ~, intermKVStore, edges)
  % Count how many flights have arrival delay in each interval specified by
  % the EDGES vector, and add these counts to INTERMKVSTORE.
  counts = histc(data.ArrDelay, edges);
  add(intermKVStore, 'Null', counts);
end
%--------------------------------------------------------------
function visualizationReducer(~, intermValList, outKVStore)
  if hasnext(intermValList)
    outVal = getnext(intermValList);
  else
    outVal = [];
  end
  while hasnext(intermValList)
    outVal = outVal + getnext(intermValList);
  end
  add(outKVStore, 'Null', outVal);
end
%--------------------------------------------------------------
```

See Also
`mapreduce` | `tabularTextDatastore`

More About

- "Getting Started with MapReduce" on page 12-3
- "Build Effective Algorithms with MapReduce" on page 12-18

Simple Data Subsetting Using MapReduce

This example shows how to extract a subset of a large data set.

There are two aspects of subsetting, or performing a query. One is selecting a subset of the variables (columns) in the data set. The other is selecting a subset of the observations, or rows.

In this example, the selection of variables takes place in the definition of the datastore. (The map function could perform a further sub-selection of variables, but that is not within the scope of this example). In this example, the role of the map function is to perform the selection of observations. The role of the reduce function is to concatenate the subsetted records extracted by each call to the map function. This approach assumes that the data set can fit in memory after the Map phase.

Prepare Data

Create a datastore using the `airlinesmall.csv` data set. This 12-megabyte data set contains 29 columns of flight information for several airline carriers, including arrival and departure times. This example uses 15 variables out of the 29 variables available in the data.

```
ds = tabularTextDatastore('airlinesmall.csv', 'TreatAsMissing', 'NA');
ds.SelectedVariableNames = ds.VariableNames([1 2 5 9 12 13 15 16 17 ...
    18 20 21 25 26 27]);
ds.SelectedVariableNames
```

```
ans = 1x15 cell
  Columns 1 through 4

    {'Year'}    {'Month'}    {'DepTime'}    {'UniqueCarrier'}

  Columns 5 through 8

    {'ActualElapsedTime'}    {'CRSElapsedTime'}    {'ArrDelay'}    {'DepDelay'}

  Columns 9 through 13

    {'Origin'}    {'Dest'}    {'TaxiIn'}    {'TaxiOut'}    {'CarrierDelay'}

  Columns 14 through 15

    {'WeatherDelay'}    {'NASDelay'}
```

The datastore treats `'NA'` values as missing, and replaces the missing values with `NaN` values by default. Additionally, the `SelectedVariableNames` property allows you to work with only the specified variables of interest, which you can verify using `preview`.

```
preview(ds)
```

```
ans=8×15 table
    Year    Month    DepTime    UniqueCarrier    ActualElapsedTime    CRSElapsedTime    ArrDelay
    ____    _____    _______    _____________    _________________    ______________    ________

    1987     10        642         {'PS'}                53                 57                8
    1987     10       1021         {'PS'}                63                 56                8
    1987     10       2055         {'PS'}                83                 82               21
    1987     10       1332         {'PS'}                59                 58               13
    1987     10        629         {'PS'}                77                 72                4
```

```
    1987      10        1446          {'PS'}                61                  65             59
    1987      10         928          {'PS'}                84                  79              3
    1987      10         859          {'PS'}               155                 143             11
```

Run MapReduce

The `mapreduce` function requires a map function and a reduce function as inputs. The mapper receives blocks of data and outputs intermediate results. The reducer reads the intermediate results and produces a final result.

In this example, the mapper receives a table with the variables described by the `SelectedVariableNames` property in the datastore. Then, the mapper extracts flights that had a high amount of delay after pushback from the gate. Specifically, it identifies flights with a duration exceeding 2.5 times the length of the scheduled duration. The mapper ignores flights prior to 1995, because some of the variables of interest for this example were not collected before that year.

Display the map function file.

```
function subsettingMapper(data, ~, intermKVStore)
  % Select flights from 1995 and later that had exceptionally long
  % elapsed flight times (including both time on the tarmac and time in
  % the air).
  idx = data.Year > 1994 & (data.ActualElapsedTime - data.CRSElapsedTime)...
    > 1.50 * data.CRSElapsedTime;
  intermVal = data(idx,:);

  add(intermKVStore,'Null',intermVal);
end
```

The reducer receives the subsetted observations obtained from the mapper and simply concatenates them into a single table. The reducer returns one key (which is relatively meaningless) and one value (the concatenated table).

Display the reduce function file.

```
function subsettingReducer(~, intermValList, outKVStore)
  % get all intermediate results from the list
  outVal = {};

  while hasnext(intermValList)
    outVal = [outVal; getnext(intermValList)];
  end
  % Note that this approach assumes the concatenated intermediate values (the
  % subset of the whole data) fit in memory.

  add(outKVStore, 'Null', outVal);
end
```

Use `mapreduce` to apply the map and reduce functions to the datastore, `ds`.

```
result = mapreduce(ds, @subsettingMapper, @subsettingReducer);
```

```
********************************
*      MAPREDUCE PROGRESS      *
********************************
Map   0% Reduce   0%
Map  16% Reduce   0%
```

```
Map  32% Reduce   0%
Map  48% Reduce   0%
Map  65% Reduce   0%
Map  81% Reduce   0%
Map  97% Reduce   0%
Map 100% Reduce   0%
Map 100% Reduce 100%
```

`mapreduce` returns an output datastore, `result`, with files in the current folder.

Display Results

Look for patterns in the first 10 variables that were pulled from the data set. These variables identify the airline, the destination, and the arrival airports, as well as some basic delay information.

```
r = readall(result);
tbl = r.Value{1};
tbl(:,1:10)
```

```
ans=37×10 table
    Year    Month    DepTime    UniqueCarrier    ActualElapsedTime    CRSElapsedTime    ArrDelay
    ____    _____    _______    _____________    _________________    ______________    ________

    1995      6       1601         {'US'}               162                 58             118
    1996      6       1834         {'CO'}               241                 75             220
    1997      1        730         {'DL'}               110                 43             137
    1997      4       1715         {'UA'}               152                 57             243
    1997      9       2232         {'NW'}               143                 50             115
    1997     10       1419         {'CO'}               196                 58             157
    1998      3       2156         {'DL'}               139                 49             146
    1998     10       1803         {'NW'}               291                 81             213
    2000      5        830         {'WN'}               140                 55              85
    2000      8       1630         {'CO'}               357                123             244
    2002      6       1759         {'US'}               260                 67             192
    2003      3       1214         {'XE'}               214                 84             124
    2003      3        604         {'XE'}               175                 60             114
    2003      4       1556         {'MQ'}               142                 52             182
    2003      5       1954         {'US'}               127                 48              78
    2003      7       1250         {'FL'}               261                 95             166
      ⋮
```

Looking at the first record, a U.S. Air flight departed the gate 14 minutes after its scheduled departure time and arrived 118 minutes late. The flight experienced a delay of 104 minutes after pushback from the gate which is the difference between `ActualElapsedTime` and `CRSElapsedTime`.

There is one anomalous record. In February of 2006, a JetBlue flight had a departure time of 3:24 a.m. and an elapsed flight time of 1650 minutes, but an arrival delay of only 415 minutes. This might be a data entry error.

Otherwise, there are no clear cut patterns concerning when and where these exceptionally delayed flights occur. No airline, time of year, time of day, or single airport dominates. Some intuitive patterns, such as O'Hare (ORD) in the winter months, are certainly present.

Delay Patterns

Beginning in 1995, the airline system performance data began including measurements of how much delay took place in the taxi phases of a flight. Then, in 2003, the data also began to include certain causes of delay.

Examine these two variables in closer detail.

```
tbl(:,[1,7,8,11:end])
```

ans=*37×8 table*

Year	ArrDelay	DepDelay	TaxiIn	TaxiOut	CarrierDelay	WeatherDelay	NASDelay
1995	118	14	7	101	NaN	NaN	NaN
1996	220	54	12	180	NaN	NaN	NaN
1997	137	70	2	12	NaN	NaN	NaN
1997	243	148	4	38	NaN	NaN	NaN
1997	115	22	4	98	NaN	NaN	NaN
1997	157	19	6	95	NaN	NaN	NaN
1998	146	56	9	47	NaN	NaN	NaN
1998	213	3	11	205	NaN	NaN	NaN
2000	85	0	5	51	NaN	NaN	NaN
2000	244	10	4	273	NaN	NaN	NaN
2002	192	-1	6	217	NaN	NaN	NaN
2003	124	-6	13	131	NaN	NaN	NaN
2003	114	-1	8	106	NaN	NaN	NaN
2003	182	92	9	106	NaN	NaN	NaN
2003	78	-1	5	90	NaN	NaN	NaN
2003	166	0	11	170	0	0	166
⋮							

For these exceptionally delayed flights, the great majority of delay occurs during taxi out, on the tarmac. Moreover, the major cause of the delay is *NASDelay*. NAS delays are holds imposed by the national aviation authorities on departures headed for an airport that is forecast to be unable to handle all scheduled arrivals at the time the flight is scheduled to arrive. NAS delay programs in effect at any given time are posted at https://www.fly.faa.gov/ois/.

Preferably, when NAS delays are imposed, boarding of the aircraft is simply delayed. Such a delay would show up as a departure delay. However, for most of the flights selected for this example, the delays took place largely after departure from the gate, leading to a taxi delay.

Rerun MapReduce

The previous map function had the subsetting criteria hard-wired in the function file. A new map function would have to be written for any new query, such as flights departing San Francisco on a given day.

A generic mapper can be more adaptive by separating out the subsetting criteria from the map function definition and using an anonymous function to configure the mapper for each query. This generic mapper uses a fourth input argument that supplies the desired query variable.

Display the generic map function file.

```
function subsettingMapperGeneric(data, ~, intermKVStore, subsetter)
  intermKey = 'Null';
```

```
  intermVal = data(subsetter(data), :);
  add(intermKVStore,intermKey,intermVal);
end
```

Create an anonymous function that performs the same selection of rows that is hard-coded in `subsettingMapper`.

```
inFlightDelay150percent = ...
   @(data) data.Year > 1994 & ...
   (data.ActualElapsedTime-data.CRSElapsedTime) > 1.50*data.CRSElapsedTime;
```

Since the `mapreduce` function requires the map and reduce functions to accept exactly three inputs, use another anonymous function to specify the fourth input to the mapper, `subsettingMapperGeneric`. Subsequently, you can use this anonymous function to call `subsettingMapperGeneric` using only three arguments (the fourth is implicit).

```
configuredMapper = ...
    @(data, info, intermKVStore) subsettingMapperGeneric(data, info, ...
    intermKVStore, inFlightDelay150percent);
```

Use `mapreduce` to apply the generic map function to the input datastore.

```
result2 = mapreduce(ds, configuredMapper, @subsettingReducer);
```

```
********************************
*      MAPREDUCE PROGRESS      *
********************************
Map   0% Reduce   0%
Map  16% Reduce   0%
Map  32% Reduce   0%
Map  48% Reduce   0%
Map  65% Reduce   0%
Map  81% Reduce   0%
Map  97% Reduce   0%
Map 100% Reduce   0%
Map 100% Reduce 100%
```

`mapreduce` returns an output datastore, `result2`, with files in the current folder.

Verify Results

Confirm that the generic mapper gets the same result as with the hard-wired subsetting logic.

```
r2 = readall(result2);
tbl2 = r2.Value{1};

if isequaln(tbl, tbl2)
    disp('Same results with the configurable mapper.')
else
    disp('Oops, back to the drawing board.')
end
```

```
Same results with the configurable mapper.
```

Local Functions

Listed here are the map and reduce functions that `mapreduce` applies to the data.

```
function subsettingMapper(data, ~, intermKVStore)
  % Select flights from 1995 and later that had exceptionally long
```

```
  % elapsed flight times (including both time on the tarmac and time in
  % the air).
  idx = data.Year > 1994 & (data.ActualElapsedTime - data.CRSElapsedTime)...
    > 1.50 * data.CRSElapsedTime;
  intermVal = data(idx,:);

  add(intermKVStore,'Null',intermVal);
end
%--------------------------------------------------------------------------
function subsettingReducer(~, intermValList, outKVStore)
  % get all intermediate results from the list
  outVal = {};

  while hasnext(intermValList)
    outVal = [outVal; getnext(intermValList)];
  end
  % Note that this approach assumes the concatenated intermediate values (the
  % subset of the whole data) fit in memory.

  add(outKVStore, 'Null', outVal);
end
%--------------------------------------------------------------------------
function subsettingMapperGeneric(data, ~, intermKVStore, subsetter)
  intermKey = 'Null';
  intermVal = data(subsetter(data), :);
  add(intermKVStore,intermKey,intermVal);
end
%--------------------------------------------------------------------------
```

See Also

mapreduce | tabularTextDatastore

More About

- “Getting Started with MapReduce” on page 12-3
- “Build Effective Algorithms with MapReduce” on page 12-18

Using MapReduce to Compute Covariance and Related Quantities

This example shows how to compute the mean and covariance for several variables in a large data set using `mapreduce`. It then uses the covariance to perform several follow-up calculations that do not require another iteration over the entire data set.

Prepare Data

Create a datastore using the `airlinesmall.csv` data set. This 12-megabyte data set contains 29 columns of flight information for several airline carriers, including arrival and departure times. In this example, select `ActualElapsedTime` (total flight time), `Distance` (total flight distance), `DepDelay` (flight departure delay), and `ArrDelay` (flight arrival delay) as the variables of interest.

```
ds = tabularTextDatastore('airlinesmall.csv', 'TreatAsMissing', 'NA');
ds.SelectedVariableNames = {'ActualElapsedTime', 'Distance', ...
                                     'DepDelay', 'ArrDelay'};
```

The datastore treats `'NA'` values as missing, and replaces the missing values with `NaN` values by default. Additionally, the `SelectedVariableNames` property allows you to work with only the selected variables of interest, which you can verify using `preview`.

```
preview(ds)
```

ans=8×4 table

ActualElapsedTime	Distance	DepDelay	ArrDelay
53	308	12	8
63	296	1	8
83	480	20	21
59	296	12	13
77	373	-1	4
61	308	63	59
84	447	-2	3
155	954	-1	11

Run MapReduce

The `mapreduce` function requires a map function and a reduce function as inputs. The mapper receives blocks of data and outputs intermediate results. The reducer reads the intermediate results and produces a final result.

In this example, the mapper computes the count, mean, and covariance for the variables in each block of data in the datastore, `ds`. Then, the mapper stores the computed values for each block as an intermediate key-value pair consisting of a single key with a cell array containing the three computed values.

Display the map function file.

```
function covarianceMapper(t,~,intermKVStore)
  % Get data from input table and remove any rows with missing values
  x = t{:,:};
  x = x(~any(isnan(x),2),:);
```

```
  % Compute and save the count, mean, and covariance
  n = size(x,1);
  m = mean(x,1);
  c = cov(x,1);

  % Store values as a single item in the intermediate key/value store
  add(intermKVStore,'key',{n m c})
end
```

The reducer combines the intermediate results for each block to obtain the count, mean, and covariance for each variable of interest in the entire data set. The reducer stores the final key-value pairs for the keys `'count'`, `'mean'`, and `'cov'` with the corresponding values for each variable.

Display the reduce function file.

```
function covarianceReducer(~,intermValIter,outKVStore)
  % We will combine results computed in the mapper for different chunks of
  % the data, updating the count, mean, and covariance each time we add a new
  % chunk.

  % First, initialize everything to zero (scalar 0 is okay)
  n1 = 0; % no rows so far
  m1 = 0; % mean so far
  c1 = 0; % covariance so far

  while hasnext(intermValIter)
    % Get the next chunk, and extract the count, mean, and covariance
    t = getnext(intermValIter);
    n2 = t{1};
    m2 = t{2};
    c2 = t{3};

    % Use weighting formulas to update the values so far
    n = n1+n2;                     % new count
    m = (n1*m1 + n2*m2) / n;       % new mean

    % New covariance is a weighted combination of the two covariance, plus
    % additional terms that relate to the difference in means
    c1 = (n1*c1 + n2*c2 + n1*(m1-m)'*(m1-m) + n2*(m2-m)'*(m2-m))/ n;

    % Store the new mean and count for the next iteration
    m1 = m;
    n1 = n;
  end

  % Save results in the output key/value store
  add(outKVStore,'count',n1);
  add(outKVStore,'mean',m1);
  add(outKVStore,'cov',c1);
end
```

Use `mapreduce` to apply the map and reduce functions to the datastore, `ds`.

```
outds = mapreduce(ds, @covarianceMapper, @covarianceReducer);
```

```
********************************
*      MAPREDUCE PROGRESS      *
********************************
```

```
Map   0% Reduce   0%
Map  16% Reduce   0%
Map  32% Reduce   0%
Map  48% Reduce   0%
Map  65% Reduce   0%
Map  81% Reduce   0%
Map  97% Reduce   0%
Map 100% Reduce   0%
Map 100% Reduce 100%
```

`mapreduce` returns a datastore, `outds`, with files in the current folder.

View the results of the `mapreduce` call by using the `readall` function on the output datastore.

```
results = readall(outds)
```

```
results=3×2 table
       Key           Value
    _________    ____________

    {'count'}    {[  120664]}
    {'mean' }    {1x4 double}
    {'cov'  }    {4x4 double}
```

```
Count = results.Value{1};
MeanVal = results.Value{2};
Covariance = results.Value{3};
```

Compute Correlation Matrix

The covariance, mean, and count values are useful to perform further calculations. Compute a correlation matrix by finding the standard deviations and normalizing them to correlation form.

```
s = sqrt(diag(Covariance));
Correlation = Covariance ./ (s*s')
```

```
Correlation = 4×4

    1.0000    0.9666    0.0278    0.0902
    0.9666    1.0000    0.0216    0.0013
    0.0278    0.0216    1.0000    0.8748
    0.0902    0.0013    0.8748    1.0000
```

The elapsed time (first column) and distance (second column) are highly correlated, since `Correlation(2,1) = 0.9666`. The departure delay (third column) and arrival delay (fourth column) are also highly correlated, since `Correlation(4,3) = 0.8748`.

Compute Regression Coefficients

Compute some regression coefficients to predict the arrival delay, `ArrDelay`, using the other three variables as predictors.

```
slopes = Covariance(1:3,1:3)\Covariance(1:3,4);
intercept = MeanVal(4) - MeanVal(1:3)*slopes;
b = table([intercept; slopes], 'VariableNames', {'Estimate'}, ...
    'RowNames', {'Intercept','ActualElapsedTime','Distance','DepDelay'})
```

```
b=4×1 table
                         Estimate
                         _________

    Intercept             -19.912
    ActualElapsedTime     0.56278
    Distance            -0.068721
    DepDelay              0.94689
```

Perform PCA

Use `svd` to perform PCA (principal components analysis). PCA is a technique for finding a lower dimensional summary of a data set. The following calculation is a simplified version of PCA, but more options are available from the `pca` and `pcacov` functions in Statistics and Machine Learning Toolbox™.

You can carry out PCA using either the covariance or correlation. In this case, use the correlation since the difference in scale of the variables is large. The first two components capture most of the variance.

```
[~,latent,pcacoef] = svd(Correlation);
latent = diag(latent)
```

```
latent = 4×1

    2.0052
    1.8376
    0.1407
    0.0164
```

Display the coefficient matrix. Each column of the coefficients matrix describes how one component is defined as a linear combination of the standardized original variables. The first component is mostly an average of the first two variables, with some additional contribution from the other variables. Similarly, the second component is mostly an average of the last two variables.

```
pcacoef
```

```
pcacoef = 4×4

   -0.6291    0.3222   -0.2444   -0.6638
   -0.6125    0.3548    0.2591    0.6572
   -0.3313   -0.6244    0.6673   -0.2348
   -0.3455   -0.6168   -0.6541    0.2689
```

Local Functions

Listed here are the map and reduce functions that `mapreduce` applies to the data.

```
function covarianceMapper(t,~,intermKVStore)
  % Get data from input table and remove any rows with missing values
  x = t{:,:};
  x = x(~any(isnan(x),2),:);

  % Compute and save the count, mean, and covariance
  n = size(x,1);
```

```
  m = mean(x,1);
  c = cov(x,1);

  % Store values as a single item in the intermediate key/value store
  add(intermKVStore,'key',{n m c})
end
%-----------------------------------------------------------------
function covarianceReducer(~,intermValIter,outKVStore)
  % We will combine results computed in the mapper for different chunks of
  % the data, updating the count, mean, and covariance each time we add a new
  % chunk.

  % First, initialize everything to zero (scalar 0 is okay)
  n1 = 0; % no rows so far
  m1 = 0; % mean so far
  c1 = 0; % covariance so far

  while hasnext(intermValIter)
    % Get the next chunk, and extract the count, mean, and covariance
    t = getnext(intermValIter);
    n2 = t{1};
    m2 = t{2};
    c2 = t{3};

    % Use weighting formulas to update the values so far
    n = n1+n2;                      % new count
    m = (n1*m1 + n2*m2) / n;        % new mean

    % New covariance is a weighted combination of the two covariance, plus
    % additional terms that relate to the difference in means
    c1 = (n1*c1 + n2*c2 + n1*(m1-m)'*(m1-m) + n2*(m2-m)'*(m2-m))/ n;

    % Store the new mean and count for the next iteration
    m1 = m;
    n1 = n;
  end

  % Save results in the output key/value store
  add(outKVStore,'count',n1);
  add(outKVStore,'mean',m1);
  add(outKVStore,'cov',c1);
end
%-----------------------------------------------------------------
```

See Also

mapreduce | tabularTextDatastore

More About

- “Getting Started with MapReduce” on page 12-3
- “Build Effective Algorithms with MapReduce” on page 12-18

Compute Summary Statistics by Group Using MapReduce

This example shows how to compute summary statistics organized by group using mapreduce. It demonstrates the use of an anonymous function to pass an extra grouping parameter to a parameterized map function. This parameterization allows you to quickly recalculate the statistics using a different grouping variable.

Prepare Data

Create a datastore using the airlinesmall.csv data set. This 12-megabyte data set contains 29 columns of flight information for several airline carriers, including arrival and departure times. For this example, select Month, UniqueCarrier (airline carrier ID), and ArrDelay (flight arrival delay) as the variables of interest.

```
ds = tabularTextDatastore('airlinesmall.csv', 'TreatAsMissing', 'NA');
ds.SelectedVariableNames = {'Month', 'UniqueCarrier', 'ArrDelay'};
```

The datastore treats 'NA' values as missing, and replaces the missing values with NaN values by default. Additionally, the SelectedVariableNames property allows you to work with only the selected variables of interest, which you can verify using preview.

```
preview(ds)
```

```
ans=8×3 table
    Month    UniqueCarrier    ArrDelay
    _____    _____________    ________

     10         {'PS'}            8
     10         {'PS'}            8
     10         {'PS'}           21
     10         {'PS'}           13
     10         {'PS'}            4
     10         {'PS'}           59
     10         {'PS'}            3
     10         {'PS'}           11
```

Run MapReduce

The mapreduce function requires a map function and a reduce function as inputs. The mapper receives blocks of data and outputs intermediate results. The reducer reads the intermediate results and produces a final result.

In this example, the mapper computes the grouped statistics for each block of data and stores the statistics as intermediate key-value pairs. Each intermediate key-value pair has a key for the group level and a cell array of values with the corresponding statistics.

This map function accepts four input arguments, whereas the mapreduce function requires the map function to accept exactly three input arguments. The call to mapreduce (below) shows how to pass in this extra parameter.

Display the map function file.

```
function statsByGroupMapper(data, ~, intermKVStore, groupVarName)
  % Data is a n-by-3 table. Remove missing values first
  delays = data.ArrDelay;
```

```
  groups = data.(groupVarName);
  notNaN =~isnan(delays);
  groups = groups(notNaN);
  delays = delays(notNaN);

  % Find the unique group levels in this chunk
  [intermKeys,~,idx] = unique(groups, 'stable');

  % Group delays by idx and apply @grpstatsfun function to each group
  intermVals = accumarray(idx,delays,size(intermKeys),@grpstatsfun);
  addmulti(intermKVStore,intermKeys,intermVals);

  function out = grpstatsfun(x)
    n = length(x); % count
    m = sum(x)/n; % mean
    v = sum((x-m).^2)/n; % variance
    s = sum((x-m).^3)/n; % skewness without normalization
    k = sum((x-m).^4)/n; % kurtosis without normalization
    out = {[n, m, v, s, k]};
  end
end
```

After the Map phase, `mapreduce` groups the intermediate key-value pairs by unique key (in this case, the airline carrier ID), so each call to the reduce function works on the values associated with one airline. The reducer receives a list of the intermediate statistics for the airline specified by the input key (`intermKey`) and combines the statistics into separate vectors: `n`, `m`, `v`, `s`, and `k`. Then, the reducer uses these vectors to calculate the count, mean, variance, skewness, and kurtosis for a single airline. The final key is the airline carrier code, and the associated values are stored in a structure with five fields.

Display the reduce function file.

```
function statsByGroupReducer(intermKey, intermValIter, outKVStore)
  n = [];
  m = [];
  v = [];
  s = [];
  k = [];

  % Get all sets of intermediate statistics
  while hasnext(intermValIter)
    value = getnext(intermValIter);
    n = [n; value(1)];
    m = [m; value(2)];
    v = [v; value(3)];
    s = [s; value(4)];
    k = [k; value(5)];
  end
  % Note that this approach assumes the concatenated intermediate values fit
  % in memory. Refer to the reducer function, covarianceReducer,  of the
  % CovarianceMapReduceExample for an alternative pairwise reduction approach

  % Combine the intermediate results
  count = sum(n);
  meanVal = sum(n.*m)/count;
  d = m - meanVal;
  variance = (sum(n.*v) + sum(n.*d.^2))/count;
```

```
  skewnessVal = (sum(n.*s) + sum(n.*d.*(3*v + d.^2)))./(count*variance^(1.5));
  kurtosisVal = (sum(n.*k) + sum(n.*d.*(4*s + 6.*v.*d +d.^3)))./(count*variance^2);

  outValue = struct('Count',count, 'Mean',meanVal, 'Variance',variance,...
                 'Skewness',skewnessVal, 'Kurtosis',kurtosisVal);

  % Add results to the output datastore
  add(outKVStore,intermKey,outValue);
end
```

Use `mapreduce` to apply the map and reduce functions to the datastore, `ds`. Since the parameterized map function accepts four inputs, use an anonymous function to pass in the airline carrier IDs as the fourth input.

```
outds1 = mapreduce(ds, ...
    @(data,info,kvs)statsByGroupMapper(data,info,kvs,'UniqueCarrier'), ...
    @statsByGroupReducer);
```

```
********************************
*      MAPREDUCE PROGRESS      *
********************************
Map   0% Reduce   0%
Map  16% Reduce   0%
Map  32% Reduce   0%
Map  48% Reduce   0%
Map  65% Reduce   0%
Map  81% Reduce   0%
Map  97% Reduce   0%
Map 100% Reduce   0%
Map 100% Reduce  10%
Map 100% Reduce  21%
Map 100% Reduce  31%
Map 100% Reduce  41%
Map 100% Reduce  52%
Map 100% Reduce  62%
Map 100% Reduce  72%
Map 100% Reduce  83%
Map 100% Reduce  93%
Map 100% Reduce 100%
```

`mapreduce` returns a datastore, `outds1`, with files in the current folder.

Read the final results from the output datastore.

```
r1 = readall(outds1)
```

```
r1=29×2 table
       Key           Value
    __________    ____________

    {'PS'    }    {1x1 struct}
    {'TW'    }    {1x1 struct}
    {'UA'    }    {1x1 struct}
    {'WN'    }    {1x1 struct}
    {'EA'    }    {1x1 struct}
    {'HP'    }    {1x1 struct}
    {'NW'    }    {1x1 struct}
    {'PA (1)'}    {1x1 struct}
```

```
    {'PI'     }    {1x1 struct}
    {'CO'     }    {1x1 struct}
    {'DL'     }    {1x1 struct}
    {'AA'     }    {1x1 struct}
    {'US'     }    {1x1 struct}
    {'AS'     }    {1x1 struct}
    {'ML (1)'}    {1x1 struct}
    {'AQ'     }    {1x1 struct}
      ⋮
```

Organize Results

To organize the results better, convert the structure containing the statistics into a table and use the carrier IDs as the row names. `mapreduce` returns the key-value pairs in the same order as they were added by the reduce function, so sort the table by carrier ID.

```
statsByCarrier = struct2table(cell2mat(r1.Value), 'RowNames', r1.Key);
statsByCarrier = sortrows(statsByCarrier, 'RowNames')
```

```
statsByCarrier=29×5 table
              Count      Mean       Variance    Skewness    Kurtosis
              _____    _______    ________    ________    ________

    9E          507     5.3669      1889.5      6.2676      61.706
    AA        14578     6.9598        1123      6.0321      93.085
    AQ          153     1.0065      230.02      3.9905      28.383
    AS         2826     8.0771         717      3.6547      24.083
    B6          793     11.936      2087.4      4.0072       27.45
    CO         7999      7.048      1053.8      4.6601      41.038
    DH          673      7.575      1491.7      2.9929      15.461
    DL        16284     7.4971      697.48      4.4746      41.115
    EA          875     8.2434      1221.3      5.2955      43.518
    EV         1655     10.028      1325.4      2.9347      14.878
    F9          332     8.4849      1138.6      4.2983      30.742
    FL         1248     9.5144      1360.4      3.6277      21.866
    HA          271    -1.5387      323.27      8.4245      109.63
    HP         3597     7.5897      744.51      5.2534      50.004
    ML (1)       69    0.15942      169.32      2.8354      16.559
    MQ         3805     8.8591      1530.5       7.054      105.51
      ⋮
```

Change Grouping Parameter

The use of an anonymous function to pass in the grouping variable allows you to quickly recalculate the statistics with a different grouping.

For this example, recalculate the statistics and group the results by `Month`, instead of by the carrier IDs, by simply passing the `Month` variable into the anonymous function.

```
outds2 = mapreduce(ds, ...
    @(data,info,kvs)statsByGroupMapper(data,info,kvs,'Month'), ...
    @statsByGroupReducer);
```

```
********************************
*      MAPREDUCE PROGRESS      *
********************************
```

```
Map   0% Reduce   0%
Map  16% Reduce   0%
Map  32% Reduce   0%
Map  48% Reduce   0%
Map  65% Reduce   0%
Map  81% Reduce   0%
Map  97% Reduce   0%
Map 100% Reduce   0%
Map 100% Reduce  17%
Map 100% Reduce  33%
Map 100% Reduce  50%
Map 100% Reduce  67%
Map 100% Reduce  83%
Map 100% Reduce 100%
```

Read the final results and organize them into a table.

```
r2 = readall(outds2);
r2 = sortrows(r2,'Key');
statsByMonth = struct2table(cell2mat(r2.Value));
mon = {'Jan','Feb','Mar','Apr','May','Jun', ...
       'Jul','Aug','Sep','Oct','Nov','Dec'};
statsByMonth.Properties.RowNames = mon
```

statsByMonth=*12×5 table*

	Count	Mean	Variance	Skewness	Kurtosis
Jan	9870	8.5954	973.69	4.1142	35.152
Feb	9160	7.3275	911.14	4.7241	45.03
Mar	10219	7.5536	976.34	5.1678	63.155
Apr	9949	6.0081	1077.4	8.9506	170.52
May	10180	5.2949	737.09	4.0535	30.069
Jun	10045	10.264	1266.1	4.8777	43.5
Jul	10340	8.7797	1069.7	5.1428	64.896
Aug	10470	7.4522	908.64	4.1959	29.66
Sep	9691	3.6308	664.22	4.6573	38.964
Oct	10590	4.6059	684.94	5.6407	74.805
Nov	10071	5.2835	808.65	8.0297	186.68
Dec	10281	10.571	1087.6	3.8564	28.823

Local Functions

Listed here are the map and reduce functions that `mapreduce` applies to the data.

```
function statsByGroupMapper(data, ~, intermKVStore, groupVarName)
  % Data is a n-by-3 table. Remove missing values first
  delays = data.ArrDelay;
  groups = data.(groupVarName);
  notNaN =~isnan(delays);
  groups = groups(notNaN);
  delays = delays(notNaN);

  % Find the unique group levels in this chunk
  [intermKeys,~,idx] = unique(groups, 'stable');

  % Group delays by idx and apply @grpstatsfun function to each group
```

```
  intermVals = accumarray(idx,delays,size(intermKeys),@grpstatsfun);
  addmulti(intermKVStore,intermKeys,intermVals);

  function out = grpstatsfun(x)
    n = length(x); % count
    m = sum(x)/n; % mean
    v = sum((x-m).^2)/n; % variance
    s = sum((x-m).^3)/n; % skewness without normalization
    k = sum((x-m).^4)/n; % kurtosis without normalization
    out = {[n, m, v, s, k]};
  end
end
%-------------------------------------------------------------------
function statsByGroupReducer(intermKey, intermValIter, outKVStore)
  n = [];
  m = [];
  v = [];
  s = [];
  k = [];

  % Get all sets of intermediate statistics
  while hasnext(intermValIter)
    value = getnext(intermValIter);
    n = [n; value(1)];
    m = [m; value(2)];
    v = [v; value(3)];
    s = [s; value(4)];
    k = [k; value(5)];
  end
  % Note that this approach assumes the concatenated intermediate values fit
  % in memory. Refer to the reducer function, covarianceReducer,  of the
  % CovarianceMapReduceExample for an alternative pairwise reduction approach

  % Combine the intermediate results
  count = sum(n);
  meanVal = sum(n.*m)/count;
  d = m - meanVal;
  variance = (sum(n.*v) + sum(n.*d.^2))/count;
  skewnessVal = (sum(n.*s) + sum(n.*d.*(3*v + d.^2)))./(count*variance^(1.5));
  kurtosisVal = (sum(n.*k) + sum(n.*d.*(4*s + 6.*v.*d +d.^3)))./(count*variance^2);

  outValue = struct('Count',count, 'Mean',meanVal, 'Variance',variance,...
                 'Skewness',skewnessVal, 'Kurtosis',kurtosisVal);

  % Add results to the output datastore
  add(outKVStore,intermKey,outValue);
end
%-------------------------------------------------------------------
```

See Also

mapreduce | tabularTextDatastore

More About

- “Getting Started with MapReduce” on page 12-3
- “Build Effective Algorithms with MapReduce” on page 12-18

Using MapReduce to Fit a Logistic Regression Model

This example shows how to use `mapreduce` to carry out simple logistic regression using a single predictor. It demonstrates chaining multiple `mapreduce` calls to carry out an iterative algorithm. Since each iteration requires a separate pass through the data, an anonymous function passes information from one iteration to the next to supply information directly to the mapper.

Prepare Data

Create a datastore using the `airlinesmall.csv` data set. This 12-megabyte data set contains 29 columns of flight information for several airline carriers, including arrival and departure times. In this example, the variables of interest are `ArrDelay` (flight arrival delay) and `Distance` (total flight distance).

```
ds = tabularTextDatastore('airlinesmall.csv', 'TreatAsMissing', 'NA');
ds.SelectedVariableNames = {'ArrDelay', 'Distance'};
```

The datastore treats `'NA'` values as missing, and replaces the missing values with `NaN` values by default. Additionally, the `SelectedVariableNames` property allows you to work with only the specified variables of interest, which you can verify using `preview`.

```
preview(ds)
```

```
ans=8×2 table
    ArrDelay    Distance
    ________    ________

       8          308   
       8          296   
      21          480   
      13          296   
       4          373   
      59          308   
       3          447   
      11          954   
```

Perform Logistic Regression

Logistic regression is a way to model the probability of an event as a function of another variable. In this example, logistic regression models the probability of a flight being more than 20 minutes late as a function of the flight distance, in thousands of miles.

To accomplish this logistic regression, the map and reduce functions must collectively perform a weighted least-squares regression based on the current coefficient values. The mapper computes a weighted sum of squares and cross product for each block of input data.

Display the map function file.

```
function logitMapper(b,t,~,intermKVStore)
  % Get data input table and remove any rows with missing values
  y = t.ArrDelay;
  x = t.Distance;
  t = ~isnan(x) & ~isnan(y);
  y = y(t)>20;                 % late by more than 20 min
  x = x(t)/1000;               % distance in thousands of miles
```

```
  % Compute the linear combination of the predictors, and the estimated mean
  % probabilities, based on the coefficients from the previous iteration
  if ~isempty(b)
    % Compute xb as the linear combination using the current coefficient
    % values, and derive mean probabilities mu from them
    xb = b(1)+b(2)*x;
    mu = 1./(1+exp(-xb));
  else
    % This is the first iteration. Compute starting values for mu that are
    % 1/4 if y=0 and 3/4 if y=1. Derive xb values from them.
    mu = (y+.5)/2;
    xb = log(mu./(1-mu));
  end

  % To perform weighted least squares, compute a sum of squares and cross
  % products matrix:
  %      (X'*W*X) = (X1'*W1*X1) + (X2'*W2*X2) + ... + (Xn'*Wn*Xn),
  % where X = [X1;X2;...;Xn]  and  W = [W1;W2;...;Wn].
  %
  % The mapper receives one chunk at a time and computes one of the terms on
  % the right hand side. The reducer adds all of the terms to get the
  % quantity on the left hand side, and then performs the regression.
  w = (mu.*(1-mu));                  % weights
  z = xb + (y - mu) .* 1./w;         % adjusted response

  X = [ones(size(x)),x,z];           % matrix of unweighted data
  wss = X' * bsxfun(@times,w,X);     % weighted cross-products X1'*W1*X1

  % Store the results for this part of the data.
  add(intermKVStore, 'key', wss);
end
```

The reducer computes the regression coefficient estimates from the sums of squares and cross products.

Display the reduce function file.

```
function logitReducer(~,intermValIter,outKVStore)
  % We will operate over chunks of the data, updating the count, mean, and
  % covariance each time we add a new chunk
  old = 0;

  % We want to perform weighted least squares. We do this by computing a sum
  % of squares and cross products matrix
  %      M = (X'*W*X) = (X1'*W1*X1) + (X2'*W2*X2) + ... + (Xn'*Wn*Xn)
  % where X = X1;X2;...;Xn]  and  W = [W1;W2;...;Wn].
  %
  % The mapper has computed the terms on the right hand side. Here in the
  % reducer we just add them up.

  while hasnext(intermValIter)
    new = getnext(intermValIter);
    old = old+new;
  end
  M = old;  % the value on the left hand side

  % Compute coefficients estimates from M. M is a matrix of sums of squares
```

```
  % and cross products for [X Y] where X is the design matrix including a
  % constant term and Y is the adjusted response for this iteration. In other
  % words, Y has been included as an additional column of X. First we
  % separate them by extracting the X'*W*X part and the X'*W*Y part.
  XtWX = M(1:end-1,1:end-1);
  XtWY = M(1:end-1,end);

  % Solve the normal equations.
  b = XtWX\XtWY;

  % Return the vector of coefficient estimates.
  add(outKVStore, 'key', b);
end
```

Run MapReduce

Run `mapreduce` iteratively by enclosing the calls to `mapreduce` in a loop. The loop runs until the convergence criteria are met, with a maximum of five iterations.

```
% Define the coefficient vector, starting as empty for the first iteration.
b = [];

for iteration = 1:5
    b_old = b;
    iteration

    % Here we will use an anonymous function as our mapper. This function
    % definition includes the value of b computed in the previous
    % iteration.
    mapper = @(t,ignore,intermKVStore) logitMapper(b,t,ignore,intermKVStore);
    result = mapreduce(ds, mapper, @logitReducer, 'Display', 'off');

    tbl = readall(result);
    b = tbl.Value{1}

    % Stop iterating if we have converged.
    if ~isempty(b_old) && ...
       ~any(abs(b-b_old) > 1e-6 * abs(b_old))
       break
    end
end
```

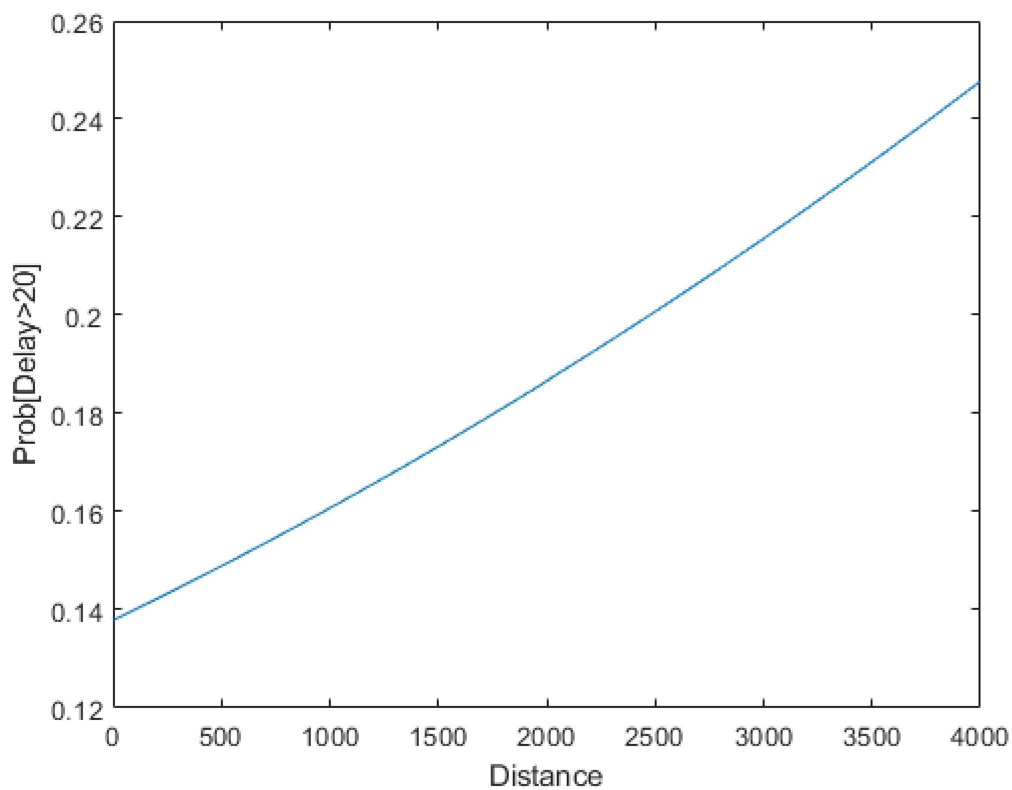
```
iteration = 1

b = 2×1

   -1.7674
    0.1209


iteration = 2

b = 2×1

   -1.8327
    0.1807


iteration = 3
```

```
b = 2×1

   -1.8331
    0.1806


iteration = 4

b = 2×1

   -1.8331
    0.1806
```

View Results

Use the resulting regression coefficient estimates to plot a probability curve. This curve shows the probability of a flight being more than 20 minutes late as a function of the flight distance.

```
xx = linspace(0,4000);
yy = 1./(1+exp(-b(1)-b(2)*(xx/1000)));
plot(xx,yy);
xlabel('Distance');
ylabel('Prob[Delay>20]')
```



Local Functions

Listed here are the map and reduce functions that `mapreduce` applies to the data.

```
function logitMapper(b,t,~,intermKVStore)
  % Get data input table and remove any rows with missing values
  y = t.ArrDelay;
  x = t.Distance;
  t = ~isnan(x) & ~isnan(y);
  y = y(t)>20;                 % late by more than 20 min
  x = x(t)/1000;               % distance in thousands of miles

  % Compute the linear combination of the predictors, and the estimated mean
  % probabilities, based on the coefficients from the previous iteration
  if ~isempty(b)
    % Compute xb as the linear combination using the current coefficient
    % values, and derive mean probabilities mu from them
    xb = b(1)+b(2)*x;
    mu = 1./(1+exp(-xb));
  else
    % This is the first iteration. Compute starting values for mu that are
    % 1/4 if y=0 and 3/4 if y=1. Derive xb values from them.
    mu = (y+.5)/2;
    xb = log(mu./(1-mu));
  end

  % To perform weighted least squares, compute a sum of squares and cross
  % products matrix:
  %      (X'*W*X) = (X1'*W1*X1) + (X2'*W2*X2) + ... + (Xn'*Wn*Xn),
  % where X = [X1;X2;...;Xn]  and  W = [W1;W2;...;Wn].
  %
  % The mapper receives one chunk at a time and computes one of the terms on
  % the right hand side. The reducer adds all of the terms to get the
  % quantity on the left hand side, and then performs the regression.
  w = (mu.*(1-mu));                  % weights
  z = xb + (y - mu) .* 1./w;         % adjusted response

  X = [ones(size(x)),x,z];           % matrix of unweighted data
  wss = X' * bsxfun(@times,w,X);     % weighted cross-products X1'*W1*X1

  % Store the results for this part of the data.
  add(intermKVStore, 'key', wss);
end
%-----------------------------------------------------------------------------
function logitReducer(~,intermValIter,outKVStore)
  % We will operate over chunks of the data, updating the count, mean, and
  % covariance each time we add a new chunk
  old = 0;

  % We want to perform weighted least squares. We do this by computing a sum
  % of squares and cross products matrix
  %      M = (X'*W*X) = (X1'*W1*X1) + (X2'*W2*X2) + ... + (Xn'*Wn*Xn)
  % where X = X1;X2;...;Xn]  and  W = [W1;W2;...;Wn].
  %
  % The mapper has computed the terms on the right hand side. Here in the
  % reducer we just add them up.

  while hasnext(intermValIter)
    new = getnext(intermValIter);
    old = old+new;
  end
  M = old;  % the value on the left hand side
```

```
  % Compute coefficients estimates from M. M is a matrix of sums of squares
  % and cross products for [X Y] where X is the design matrix including a
  % constant term and Y is the adjusted response for this iteration. In other
  % words, Y has been included as an additional column of X. First we
  % separate them by extracting the X'*W*X part and the X'*W*Y part.
  XtWX = M(1:end-1,1:end-1);
  XtWY = M(1:end-1,end);

  % Solve the normal equations.
  b = XtWX\XtWY;

  % Return the vector of coefficient estimates.
  add(outKVStore, 'key', b);
end
%-----------------------------------------------------------------------------
```

See Also

mapreduce | tabularTextDatastore

More About

- “Getting Started with MapReduce” on page 12-3
- “Build Effective Algorithms with MapReduce” on page 12-18

Tall Skinny QR (TSQR) Matrix Factorization Using MapReduce

This example shows how to compute a tall skinny QR (TSQR) factorization using `mapreduce`. It demonstrates how to chain `mapreduce` calls to perform multiple iterations of factorizations, and uses the `info` argument of the map function to compute numeric keys.

Prepare Data

Create a datastore using the `airlinesmall.csv` data set. This 12-megabyte data set contains 29 columns of flight information for several airline carriers, including arrival and departure times. In this example, the variables of interest are `ArrDelay` (flight arrival delay), `DepDelay` (flight departure delay) and `Distance` (total flight distance).

```
ds = tabularTextDatastore('airlinesmall.csv', 'TreatAsMissing', 'NA');
ds.ReadSize = 1000;
ds.SelectedVariableNames = {'ArrDelay', 'DepDelay', 'Distance'};
```

The datastore treats `'NA'` values as missing and replaces the missing values with `NaN` values by default. The `ReadSize` property lets you specify how to partition the data into blocks. Additionally, the `SelectedVariableNames` property allows you to work with only the specified variables of interest, which you can verify using `preview`.

```
preview(ds)
```

```
ans=8×3 table
    ArrDelay    DepDelay    Distance
    ________    ________    ________

        8          12         308   
        8           1         296   
       21          20         480   
       13          12         296   
        4          -1         373   
       59          63         308   
        3          -2         447   
       11          -1         954   
```

Chain MapReduce Calls

The implementation of the multi-iteration TSQR algorithm needs to chain consecutive `mapreduce` calls. To demonstrate the general chaining design pattern, this example uses two `mapreduce` iterations. The output from the map function calls is passed into a large set of reducers, and then the output of these reducers becomes the input for the next `mapreduce` iteration.

First MapReduce Iteration

In the first iteration, the map function, `tsqrMapper`, receives one block (the ith) of data, which is a table of size $N_i \times 3$. The mapper computes the R matrix of this block of data and stores it as an intermediate result. Then, `mapreduce` aggregates the intermediate results by unique key before sending them to the reduce function. Thus, `mapreduce` sends all intermediate R matrices with the same key to the same reducer.

Since the reducer uses `qr`, which is an in-memory MATLAB function, it's best to first make sure that the R matrices fit in memory. This example divides the dataset into eight partitions. The `mapreduce` function reads the data in blocks and passes the data along with some meta information to the map function. The `info` input argument is the second input to the map function and it contains the read offset and file size information that are necessary to generate the key,

```
key = ceil(offset/fileSize/numPartitions).
```

Display the map function file.

```
function tsqrMapper(data, info, intermKVStore)
  x = data{:,:};
  x(any(isnan(x),2),:) = [];% Remove missing values
  [~, r] = qr(x,0);

  % intermKey = randi(4); % random integer key for partitioning intermediate results
  intermKey = computeKey(info, 8);
  add(intermKVStore,intermKey, r);

  function key = computeKey(info, numPartitions)
    fileSize = info.FileSize; % total size of the underlying data file
    partitionSize = fileSize/numPartitions; % size in bytes of each partition
    offset = info.Offset; % offset in bytes of the current read
    key = ceil(offset/partitionSize);
  end
end
```

The reduce function receives a list of the intermediate *R* matrices, vertically concatenates them, and computes the *R* matrix of the concatenated matrix.

Display the reduce function file.

```
function tsqrReducer(intermKey, intermValIter, outKVStore)
  x = [];

  while (intermValIter.hasnext)
    x = [x;intermValIter.getnext];
  end
  % Note that this approach assumes the concatenated intermediate values fit
  % in memory. Consider increasing the number of reduce tasks (increasing the
  % number of partitions in the tsqrMapper) and adding more iterations if it
  % does not fit in memory.

  [~, r] =qr(x,0);

  add(outKVStore,intermKey,r);
end
```

Use `mapreduce` to apply the map and reduce functions to the datastore, `ds`.

```
outds1 = mapreduce(ds, @tsqrMapper, @tsqrReducer);
```

```
********************************
*      MAPREDUCE PROGRESS      *
********************************
Map   0% Reduce   0%
Map  10% Reduce   0%
Map  20% Reduce   0%
Map  30% Reduce   0%
Map  40% Reduce   0%
Map  50% Reduce   0%
```

```
Map  60% Reduce   0%
Map  70% Reduce   0%
Map  80% Reduce   0%
Map  90% Reduce   0%
Map 100% Reduce   0%
Map 100% Reduce  11%
Map 100% Reduce  22%
Map 100% Reduce  33%
Map 100% Reduce  44%
Map 100% Reduce  56%
Map 100% Reduce  67%
Map 100% Reduce  78%
Map 100% Reduce  89%
Map 100% Reduce 100%
```

`mapreduce` returns an output datastore, `outds1`, with files in the current folder.

Second MapReduce Iteration

The second iteration uses the output of the first iteration, `outds1`, as its input. This iteration uses an identity mapper, `identityMapper`, which simply copies over the data using a single key, `'Identity'`.

Display the identity mapper file.

```
function identityMapper(data, info, intermKVStore)
  % This mapper function simply copies the data and add them to the
  % intermKVStore as intermediate values.
  x = data.Value{:,:};
  add(intermKVStore,'Identity', x);
end
```

The reducer function is the same in both iterations. The use of a single key by the map function means that `mapreduce` only calls the reduce function once in the second iteration.

Use `mapreduce` to apply the identity mapper and the same reducer to the output from the first `mapreduce` call.

```
outds2 = mapreduce(outds1, @identityMapper, @tsqrReducer);
```

```
********************************
*      MAPREDUCE PROGRESS      *
********************************
Map   0% Reduce   0%
Map  11% Reduce   0%
Map  22% Reduce   0%
Map  33% Reduce   0%
Map  44% Reduce   0%
Map  55% Reduce   0%
Map  66% Reduce   0%
Map  77% Reduce   0%
Map  88% Reduce   0%
Map 100% Reduce   0%
Map 100% Reduce 100%
```

View Results

Read the final results from the output datastore.

```
r = readall(outds2);
r.Value{:}
```

```
ans = 3×3
10^5 ×

    0.1091    0.0893    0.5564
         0   -0.0478   -0.4890
         0         0    3.0130
```

Local Functions

Listed here are the map and reduce functions that `mapreduce` applies to the data.

```
function tsqrMapper(data, info, intermKVStore)
  x = data{:,:};
  x(any(isnan(x),2),:) = [];% Remove missing values
  [~, r] = qr(x,0);

  % intermKey = randi(4); % random integer key for partitioning intermediate results
  intermKey = computeKey(info, 8);
  add(intermKVStore,intermKey, r);

  function key = computeKey(info, numPartitions)
    fileSize = info.FileSize; % total size of the underlying data file
    partitionSize = fileSize/numPartitions; % size in bytes of each partition
    offset = info.Offset; % offset in bytes of the current read
    key = ceil(offset/partitionSize);
  end
end
%--------------------------------------------------------------------------------
function tsqrReducer(intermKey, intermValIter, outKVStore)
  x = [];

  while (intermValIter.hasnext)
    x = [x;intermValIter.getnext];
  end
  % Note that this approach assumes the concatenated intermediate values fit
  % in memory. Consider increasing the number of reduce tasks (increasing the
  % number of partitions in the tsqrMapper) and adding more iterations if it
  % does not fit in memory.

  [~, r] =qr(x,0);

  add(outKVStore,intermKey,r);
end
%--------------------------------------------------------------------------------
function identityMapper(data, info, intermKVStore)
  % This mapper function simply copies the data and add them to the
  % intermKVStore as intermediate values.
  x = data.Value{:,:};
  add(intermKVStore,'Identity', x);
end
%--------------------------------------------------------------------------------
```

Reference

1 Paul G. Constantine and David F. Gleich. 2011. Tall and skinny QR factorizations in MapReduce architectures. In Proceedings of the Second International Workshop on MapReduce and Its Applications (MapReduce '11). ACM, New York, NY, USA, 43-50. DOI=10.1145/1996092.1996103 https://doi.acm.org/10.1145/1996092.1996103

See Also

`mapreduce` | `tabularTextDatastore`

More About

- “Getting Started with MapReduce” on page 12-3
- “Build Effective Algorithms with MapReduce” on page 12-18

Compute Maximum Average HSV of Images with MapReduce

This example shows how to use `ImageDatastore` and `mapreduce` to find images with maximum hue, saturation and brightness values in an image collection.

Prepare Data

Create a datastore using the images in `toolbox/matlab/demos` and `toolbox/matlab/imagesci`. The selected images have the extensions `.jpg`, `.tif` and `.png`.

```
demoFolder = fullfile(matlabroot, 'toolbox', 'matlab', 'demos');
imsciFolder = fullfile(matlabroot, 'toolbox', 'matlab', 'imagesci');
```

Create a datastore using the folder paths, and filter which images are included in the datastore using the `FileExtensions` Name-Value pair.

```
ds = imageDatastore({demoFolder, imsciFolder}, ...
    'FileExtensions', {'.jpg', '.tif', '.png'});
```

Find Average Maximum HSV from All Images

One way to find the maximum average hue, saturation, and brightness values in the collection of images is to use `readimage` within a for-loop, processing the images one at a time. For an example of this method, see “Read and Analyze Image Files” on page 12-98.

This example uses `mapreduce` to accomplish the same task, however, the `mapreduce` method is highly scalable to larger collections of images. While the for-loop method is reasonable for small collections of images, it does not scale well to a large collection of images.

Scale to MapReduce

- The `mapreduce` function requires a map function and a reduce function as inputs.
- The map function receives blocks of data and outputs intermediate results.
- The reduce function reads the intermediate results and produces a final result.

Map function

- In this example, the map function stores the image data and the average HSV values as intermediate values.
- The intermediate values are associated with 3 keys, `'Average Hue'`, `'Average Saturation'` and `'Average Brightness'`.

```
function hueSaturationValueMapper(data, info, intermKVStore)
  if ~ismatrix(data)
    hsv = rgb2hsv(data);

    % Extract Hue values
    h = hsv(:,:,1);

    % Extract Saturation values
    s = hsv(:,:,2);

    % Extract Brightness values
    v = hsv(:,:,3);
```

```
    % Find average of HSV values
    avgH = mean(h(:));
    avgS = mean(s(:));
    avgV = mean(v(:));

    % Add intermediate key-value pairs
    add(intermKVStore, 'Average Hue', struct('Filename', info.Filename, 'Avg', avgH));
    add(intermKVStore, 'Average Saturation', struct('Filename', info.Filename, 'Avg', avgS));
    add(intermKVStore, 'Average Brightness', struct('Filename', info.Filename, 'Avg', avgV));
  end
end
```

Reduce function

- The reduce function receives a list of the image file names along with the respective average HSV values and finds the overall maximum values of average hue, saturation and brightness values.
- `mapreduce` only calls this reduce function 3 times, since the map function only adds three unique keys.
- The reduce function uses `add` to add a final key-value pair to the output. For example, `'Maximum Average Hue'` is the key and the respective file name is the value.

```
function hueSaturationValueReducer(key, intermValIter, outKVSTore)
  maxAvg = 0;
  maxImageFilename = '';

  % Loop over values for each key
  while hasnext(intermValIter)
    value = getnext(intermValIter);
    % Compare values to determine maximum
    if value.Avg > maxAvg
      maxAvg = value.Avg;
      maxImageFilename = value.Filename;
    end
  end

  % Add final key-value pair
  add(outKVSTore, ['Maximum ' key], maxImageFilename);
end
```

Run MapReduce

Use `mapreduce` to apply the map and reduce functions to the datastore, `ds`.

```
maxHSV = mapreduce(ds, @hueSaturationValueMapper, @hueSaturationValueReducer);
```

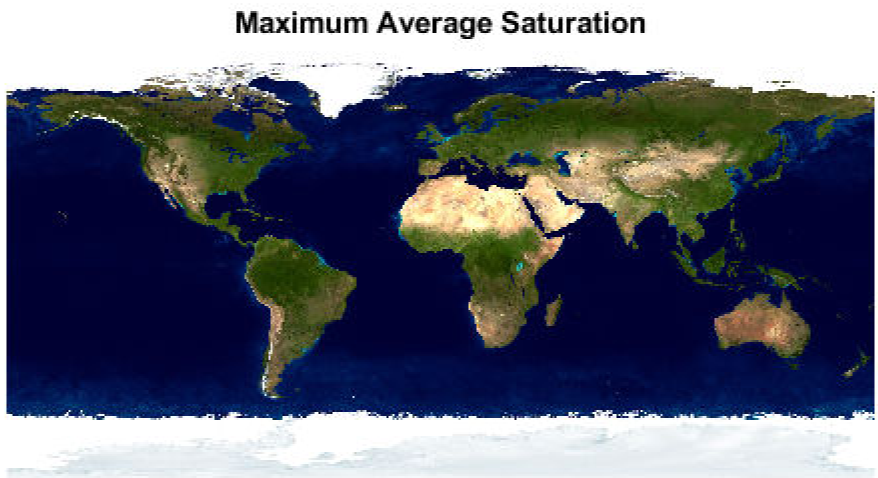
```
********************************
*      MAPREDUCE PROGRESS      *
********************************
Map   0% Reduce   0%
Map  12% Reduce   0%
Map  25% Reduce   0%
Map  37% Reduce   0%
Map  50% Reduce   0%
Map  62% Reduce   0%
Map  75% Reduce   0%
Map  87% Reduce   0%
Map 100% Reduce   0%
```

```
Map 100% Reduce  33%
Map 100% Reduce  67%
Map 100% Reduce 100%
```

`mapreduce` returns a datastore, `maxHSV`, with files in the current folder.

Read and display the final result from the output datastore, `maxHSV`. Use `find` and `strcmp` to find the file index from the `Files` property.

```
tbl = readall(maxHSV);
for i = 1:height(tbl)
    figure;
    idx = find(strcmp(ds.Files, tbl.Value{i}));
    imshow(readimage(ds, idx), 'InitialMagnification', 'fit');
    title(tbl.Key{i});
end
```


Maximum Average Saturation

Maximum Average Brightness

Local Functions

Listed here are the map and reduce functions that `mapreduce` applies to the data.

```
function hueSaturationValueMapper(data, info, intermKVStore)
  if ~ismatrix(data)
    hsv = rgb2hsv(data);

    % Extract Hue values
    h = hsv(:,:,1);

    % Extract Saturation values
    s = hsv(:,:,2);

    % Extract Brightness values
    v = hsv(:,:,3);

    % Find average of HSV values
    avgH = mean(h(:));
    avgS = mean(s(:));
    avgV = mean(v(:));

    % Add intermediate key-value pairs
    add(intermKVStore, 'Average Hue', struct('Filename', info.Filename, 'Avg', avgH));
    add(intermKVStore, 'Average Saturation', struct('Filename', info.Filename, 'Avg', avgS));
    add(intermKVStore, 'Average Brightness', struct('Filename', info.Filename, 'Avg', avgV));
  end
end
```

```
%--------------------------------------------------------------------------------------
function hueSaturationValueReducer(key, intermValIter, outKVSTore)
  maxAvg = 0;
  maxImageFilename = '';

  % Loop over values for each key
  while hasnext(intermValIter)
    value = getnext(intermValIter);
    % Compare values to determine maximum
    if value.Avg > maxAvg
      maxAvg = value.Avg;
      maxImageFilename = value.Filename;
    end
  end

  % Add final key-value pair
  add(outKVSTore, ['Maximum ' key], maxImageFilename);
end
%--------------------------------------------------------------------------------------
```

See Also

imageDatastore | mapreduce | tall

More About

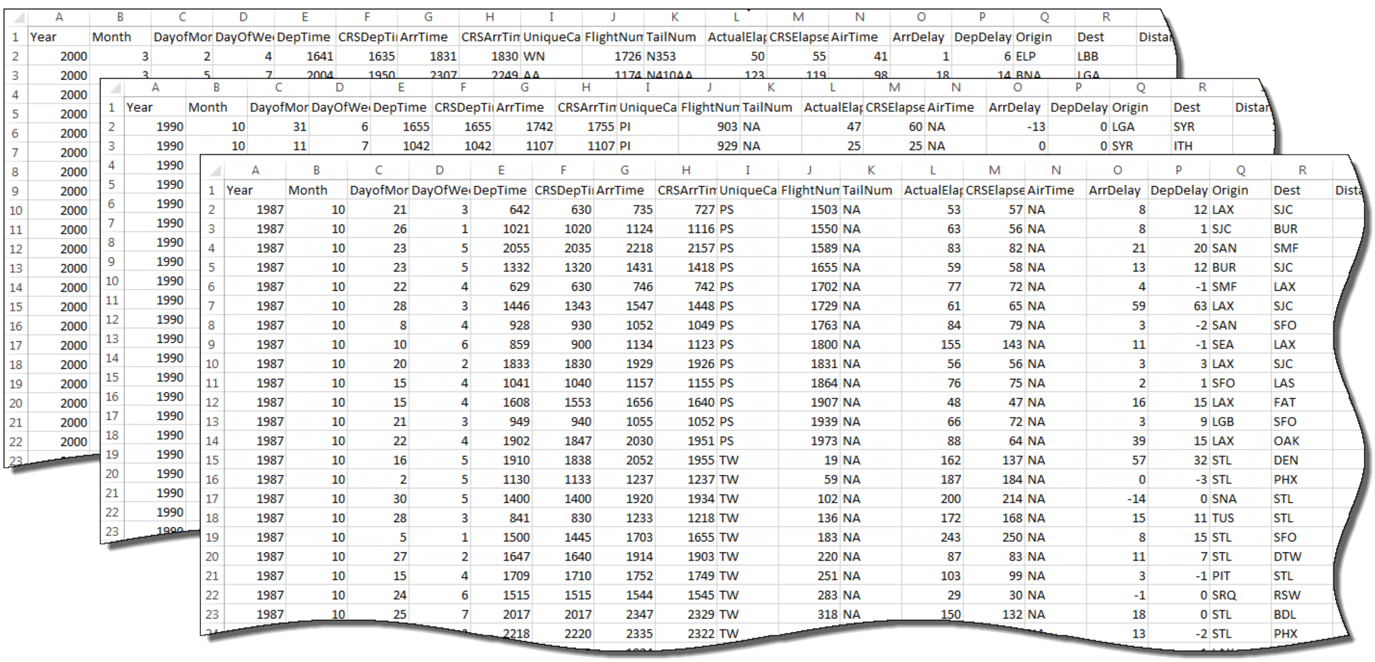
- “Getting Started with MapReduce” on page 12-3
- “Build Effective Algorithms with MapReduce” on page 12-18
- “Tall Arrays for Out-of-Memory Data” on page 12-136
- “Getting Started with Datastore” on page 12-84

Getting Started with Datastore

In this section...

What Is a Datastore?

A datastore is an object for reading a single file or a collection of files or data. The datastore acts as a repository for data that has the same structure and formatting. For example, each file in a datastore must contain data of the same type (such as numeric or text) appearing in the same order, and separated by the same delimiter.



A datastore is useful when:

- Each file in the collection might be too large to fit in memory. A datastore allows you to read and analyze data from each file in smaller portions that do fit in memory.
- Files in the collection have arbitrary names. A datastore acts as a repository for files in one or more folders. The files are not required to have sequential names.

You can create a datastore based on the type of data or application. The different types of datastores contain properties pertinent to the type of data that they support. For example, see the following table for a list of MATLAB datastores. For a complete list of datastores, see "Select Datastore for File Format or Application" on page 12-88.

Type of File or Data	Datastore Type
Text files containing column-oriented data, including CSV files.	`TabularTextDatastore`

Type of File or Data	Datastore Type
Image files, including formats that are supported by `imread` such as JPEG and PNG.	`ImageDatastore`
Spreadsheet files with a supported Excel format such as `.xlsx`.	`SpreadsheetDatastore`
Key-value pair data that are inputs to or outputs of `mapreduce`.	`KeyValueDatastore`
Parquet files containing column-oriented data.	`ParquetDatastore`
Custom file formats. Requires a provided function for reading data.	`FileDatastore`
Datastore for checkpointing `tall` arrays.	`TallDatastore`

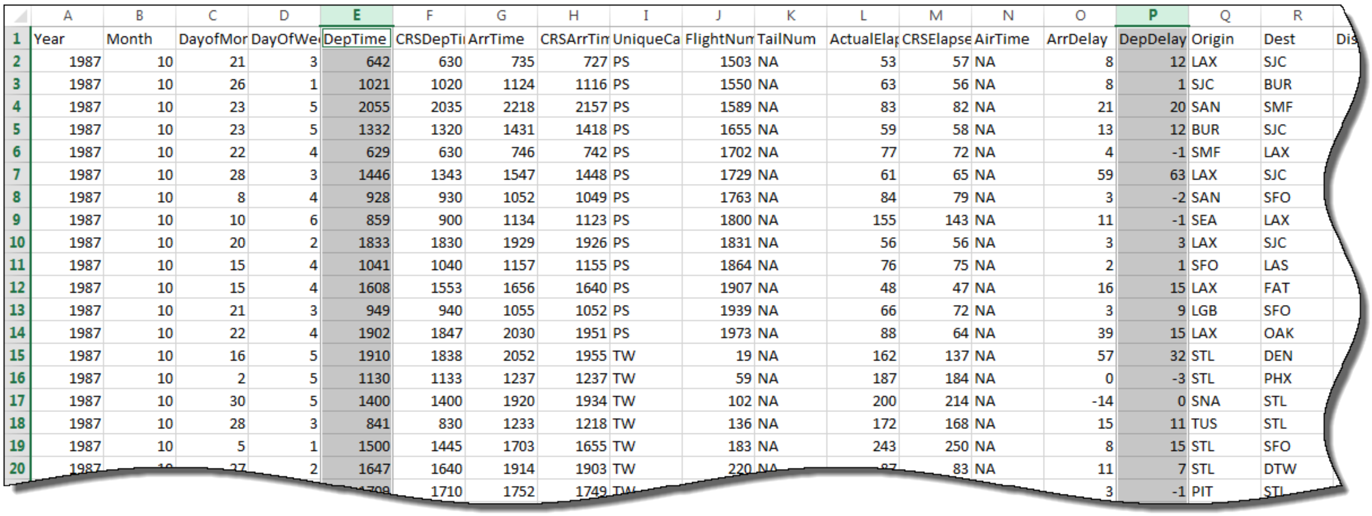
Create and Read from a Datastore

Use the `tabularTextDatastore` function to create a datastore from the sample file `airlinesmall.csv`, which contains departure and arrival information about individual airline flights. The result is a `TabularTextDatastore` object.

```
ds = tabularTextDatastore('airlinesmall.csv')
```

```
ds = 

  TabularTextDatastore with properties:

                      Files: {
                             ' ...\matlab\toolbox\matlab\demos\airlinesmall.csv'
                             }
                    Folders: {
                             ' ...\matlab\toolbox\matlab\demos'
                             }
               FileEncoding: 'UTF-8'
   AlternateFileSystemRoots: {}
      PreserveVariableNames: false
          ReadVariableNames: true
              VariableNames: {'Year', 'Month', 'DayofMonth' ... and 26 more}
             DatetimeLocale: en_US

  Text Format Properties:
             NumHeaderLines: 0
                  Delimiter: ','
               RowDelimiter: '\r\n'
             TreatAsMissing: ''
               MissingValue: NaN

  Advanced Text Format Properties:
            TextscanFormats: {'%f', '%f', '%f' ... and 26 more}
                   TextType: 'char'
         ExponentCharacters: 'eEdD'
               CommentStyle: ''
                 Whitespace: ' \b\t'
    MultipleDelimitersAsOne: false

  Properties that control the table returned by preview, read, readall:
      SelectedVariableNames: {'Year', 'Month', 'DayofMonth' ... and 26 more}
            SelectedFormats: {'%f', '%f', '%f' ... and 26 more}
                   ReadSize: 20000 rows
                 OutputType: 'table'
                   RowTimes: []

  Write-specific Properties:
     SupportedOutputFormats: ["txt"    "csv"    "xlsx"    "xls"    "parquet"    "parq"]
        DefaultOutputFormat: "txt"
```

After creating the datastore, you can preview the data without having to load it all into memory. You can specify variables (columns) of interest using the `SelectedVariableNames` property to preview or read only those variables.



	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	
1	Year	Month	DayofMor	DayOfWe	DepTime	CRSDepTi	ArrTime	CRSArrTin	UniqueCa	FlightNun	TailNum	ActualElap	CRSElapse	AirTime	ArrDelay	DepDelay	Origin	Dest	Dis
2	1987	10	21	3	642	630	735	727	PS	1503	NA	53	57	NA	8	12	LAX	SJC	
3	1987	10	26	1	1021	1020	1124	1116	PS	1550	NA	63	56	NA	8	1	SJC	BUR	
4	1987	10	23	5	2055	2035	2218	2157	PS	1589	NA	83	82	NA	21	20	SAN	SMF	
5	1987	10	23	5	1332	1320	1431	1418	PS	1655	NA	59	58	NA	13	12	BUR	SJC	
6	1987	10	22	4	629	630	746	742	PS	1702	NA	77	72	NA	4	-1	SMF	LAX	
7	1987	10	28	3	1446	1343	1547	1448	PS	1729	NA	61	65	NA	59	63	LAX	SJC	
8	1987	10	8	4	928	930	1052	1049	PS	1763	NA	84	79	NA	3	-2	SAN	SFO	
9	1987	10	10	6	859	900	1134	1123	PS	1800	NA	155	143	NA	11	-1	SEA	LAX	
10	1987	10	20	2	1833	1830	1929	1926	PS	1831	NA	56	56	NA	3	3	LAX	SJC	
11	1987	10	15	4	1041	1040	1157	1155	PS	1864	NA	76	75	NA	2	1	SFO	LAS	
12	1987	10	15	4	1608	1553	1656	1640	PS	1907	NA	48	47	NA	16	15	LAX	FAT	
13	1987	10	21	3	949	940	1055	1052	PS	1939	NA	66	72	NA	3	9	LGB	SFO	
14	1987	10	22	4	1902	1847	2030	1951	PS	1973	NA	88	64	NA	39	15	LAX	OAK	
15	1987	10	16	5	1910	1838	2052	1955	TW	19	NA	162	137	NA	57	32	STL	DEN	
16	1987	10	2	5	1130	1133	1237	1237	TW	59	NA	187	184	NA	0	-3	STL	PHX	
17	1987	10	30	5	1400	1400	1920	1934	TW	102	NA	200	214	NA	-14	0	SNA	STL	
18	1987	10	28	3	841	830	1233	1218	TW	136	NA	172	168	NA	15	11	TUS	STL	
19	1987	10	5	1	1500	1445	1703	1655	TW	183	NA	243	250	NA	8	15	STL	SFO	
20	1987	10	27	2	1647	1640	1914	1903	TW	220	NA	87	83	NA	11	7	STL	DTW	
					1708	1710	1752	1749	TW						3	-1	PIT	STL	

```
ds.SelectedVariableNames = {'DepTime','DepDelay'};
preview(ds)
```

```
ans = 

  8×2 table

    DepTime    DepDelay
    _______    ________

      642         12   
     1021          1   
     2055         20   
     1332         12   
      629         -1   
     1446         63   
      928         -2   
      859         -1   
```

You can specify the values in your data which represent missing values. In `airlinesmall.csv`, missing values are represented by `NA`.

```
ds.TreatAsMissing = 'NA';
```

If all of the data in the datastore for the variables of interest fit in memory, you can read it using the `readall` function.

```
T = readall(ds);
```

Otherwise, read the data in smaller subsets that do fit in memory, using the `read` function. By default, the `read` function reads from a `TabularTextDatastore` 20,000 rows at a time. However, you can change this value by assigning a new value to the `ReadSize` property.

```
ds.ReadSize = 15000;
```

Reset the datastore to the initial state before re-reading, using the `reset` function. By calling the `read` function within a `while` loop, you can perform intermediate calculations on each subset of data, and then aggregate the intermediate results at the end. This code calculates the maximum value of the `DepDelay` variable.

```
reset(ds)
X = [];
while hasdata(ds)
      T = read(ds);
      X(end+1) = max(T.DepDelay);
end
maxDelay = max(X)
```

```
maxDelay =

        1438
```

If the data in each individual file fits in memory, you can specify that each call to `read` should read one complete file rather than a specific number of rows.

```
reset(ds)
ds.ReadSize = 'file';
X = [];
while hasdata(ds)
      T = read(ds);
      X(end+1) = max(T.DepDelay);
end
maxDelay = max(X);
```

In addition to reading subsets of data in a datastore, you can apply map and reduce functions to the datastore using `mapreduce` or create a tall array using `tall`. For more information, see “Getting Started with MapReduce” on page 12-3 and “Tall Arrays for Out-of-Memory Data” on page 12-136.

See Also

`KeyValueDatastore` | `fileDatastore` | `imageDatastore` | `mapreduce` | `spreadsheetDatastore` | `tabularTextDatastore` | `tall`

Related Examples

- “Select Datastore for File Format or Application” on page 12-88
- “Read and Analyze Large Tabular Text File” on page 12-96
- “Read and Analyze Image Files” on page 12-98

Select Datastore for File Format or Application

A datastore is a repository for collections of data that are too large to fit in memory. Each file format and application uses a different type of datastore, which contains properties pertinent to the type of data or application that it supports. MATLAB provides datastores for standard file formats such as Excel files and datastores for specific applications such as Deep Learning. In addition to the existing datastores, if your data is in a proprietary format, then you can develop a customized datastore using the custom datastore framework.

Datastores for Standard File Formats

For a collection of data in standard file format use one of these options.

Datastore	Description
`TabularTextDatastore`	Text files containing column-oriented data, including CSV files
`SpreadsheetDatastore`	Spreadsheet files with a supported Excel format such as `.xlsx`
`ImageDatastore`	Image files, including formats that are supported by `imread` such as JPEG and PNG
`ParquetDatastore`	Parquet files containing column-oriented data
`FileDatastore`	Files with nonstandard file format Requires a custom file reading function

Transform or combine existing datastores.

Datastore	Description
`CombinedDatastore`	Datastore to combine data read from multiple underlying datastores
`TransformedDatastore`	Datastore to transform underlying datastore

Datastores to integrate with MapReduce and tall arrays.

Datastore	Description
`KeyValueDatastore`	Key-value pair data that are inputs to or outputs of `mapreduce`
`TallDatastore`	Datastore for checkpointing `tall` arrays

Datastores for Specific Applications

Based on your application use one of these datastores.

Application	Datastore	Description	Toolbox Required
Simulink Model Data	`SimulationDatastore`	Datastore for simulation input and output data that you use with a Simulink model	Simulink
Simulation Ensemble and Predictive Maintenance Data	`simulationEnsembleDatastore`	Datastore to manage simulation ensemble data	Predictive Maintenance Toolbox™
	`fileEnsembleDatastore`	Datastore to manage ensemble data in custom file format	Predictive Maintenance Toolbox
Measurement Data Format (MDF) Files	`mdfDatastore`	Datastore for collection of MDF files	Vehicle Network Toolbox™
	`mdfDatastore`	Datastore for collection of MDF files	Powertrain Blockset™
Deep Learning Datastores for preprocessing image or sequence data	`pixelLabelDatastore`	Datastore for pixel label data	Computer Vision Toolbox™ and Deep Learning Toolbox™
	`pixelLabelImageDatastore`	Datastore for training semantic segmentation networks Datastore is nondeterministic	Computer Vision Toolbox and Deep Learning Toolbox
	`boxLabelDatastore`	Datastore for bounding box label data	Computer Vision Toolbox and Deep Learning Toolbox
	`signalDatastore`	Datastore for collection of signal files	Signal Processing Toolbox™ and Deep Learning Toolbox
	`randomPatchExtractionDatastore`	Datastore for extracting random patches from images or pixel label images Datastore is nondeterministic	Image Processing Toolbox™ and Deep Learning Toolbox
	`denoisingImageDatastore`	Datastore to train an image denoising deep neural network Datastore is nondeterministic	Image Processing Toolbox and Deep Learning Toolbox

Application	Datastore	Description	Toolbox Required
	`augmentedImageDatastore`	Datastore for resizing and augmenting training images Datastore is nondeterministic	Deep Learning Toolbox
Audio Data	`audioDatastore`	Datastore for collection of audio files	Audio Toolbox™
Out-of-Memory Image Data	`bigimageDatastore`	Datastore to manage blocks of a single image that is too large to fit in memory	Image Processing Toolbox
Database Data	`databaseDatastore`	Datastore for collections of data in a relational database	Database Toolbox

Custom File Formats

For a collection of data in a custom file format, if each individual file fits in memory, use `FileDatastore` along with your custom file reading function. Otherwise, develop your own fully customized datastore for custom or proprietary data using the `matlab.io.Datastore` class. See “Develop Custom Datastore” on page 12-107.

Nondeterministic Datastores

Datastores that do not return the exact same data for a call to the `read` function after a call to the `reset` function are nondeterministic datastores. Do not use nondeterministic datastores with `tall` arrays, `mapreduce`, or any other code that requires reading the data more than once.

Some applications require data that is randomly augmented or transformed. For example, the `augmentedImageDatastore` datastore, from the deep learning application augments training image data with randomized preprocessing operations to help prevent the network from overfitting and memorizing the exact details of the training images. The output of this datastore is different every time you perform a `read` operation after a call to `reset`.

See Also

`FileDatastore` | `ImageDatastore` | `SpreadsheetDatastore` | `TabularTextDatastore` | `TallDatastore` | `tall`

More About

- “Getting Started with Datastore” on page 12-84
- “Tall Arrays for Out-of-Memory Data” on page 12-136
- “Develop Custom Datastore” on page 12-107

Work with Remote Data

In MATLAB, you can read and write data to and from a remote location, such as cloud storage in Amazon S3 (Simple Storage Service), Microsoft Azure Storage Blob, and Hadoop Distributed File System (HDFS).

You can access remote data using datastore objects. Use the datastore to examine part of your data from your desktop version of MATLAB. Then, after prototyping your code locally, you can scale up to a cluster or cloud. Scaling up improves execution efficiency as it is more efficient to run large calculations in the same location as the data. To write data to a remote location, you can use the `write` function on a tall or distributed array.

Amazon S3

MATLAB enables you to use Amazon S3 as an online file storage web service offered by Amazon Web Services. You can use data stored on Amazon S3 with datastore objects such as `ImageDatastore`, `FileDatastore`, `SpreadsheetDatastore`, or `TabularTextDatastore`. When you specify the location of the data, you must specify the full path to the files or folders using a uniform resource locator (URL) of the form

`s3://`*`bucketname/path_to_file`*

`bucketname` is the name of the container and *`path_to_file`* is the path to the file or folders.

Amazon S3 provides data storage through web services interfaces. You can use a *bucket* as a container to store objects in Amazon S3.

Set Up Access

To work with remote data in Amazon S3, you must set up access first:

1. Sign up for an Amazon Web Services (AWS) root account. See Amazon Web Services: Account.
2. Using your AWS root account, create an IAM (Identity and Access Management) user. See Creating an IAM User in Your AWS Account.
3. Generate an access key to receive an access key ID and a secret access key. See Managing Access Keys for IAM Users.
4. Configure your machine with the AWS access key ID, secret access key, and region using the AWS Command Line Interface tool from https://aws.amazon.com/cli/. Alternatively, set the environment variables directly by using `setenv`:

   - `AWS_ACCESS_KEY_ID` and `AWS_SECRET_ACCESS_KEY` — Authenticate and enable use of Amazon S3 services. (You generated this pair of access key variables in step 3.)
   - `AWS_DEFAULT_REGION` (optional) — Select the geographic region of your bucket. The value of this environment variable is typically determined automatically, but the bucket owner might require that you set it manually.
   - `AWS_SESSION_TOKEN` (optional) — Specify the session token if you are using temporary security credentials, such as with AWS® Federated Authentication.

If you are using Parallel Computing Toolbox, you must ensure the cluster has been configured to access S3 services. You can copy your client environment variables to the workers on a cluster by setting `EnvironmentVariables` in parpool, batch, createJob or in the Cluster Profile Manager.

Read Data from Amazon S3

You can read data from Amazon S3 using datastore objects. For example, create an `ImageDatastore`, read a specified image from the datastore, and then display the image to screen.

```
setenv('AWS_ACCESS_KEY_ID', 'YOUR_AWS_ACCESS_KEY_ID');
setenv('AWS_SECRET_ACCESS_KEY', 'YOUR_AWS_SECRET_ACCESS_KEY');

ds = imageDatastore('s3://bucketname/image_datastore/jpegfiles', ...
  'IncludeSubfolders', true, 'LabelSource', 'foldernames');
img = ds.readimage(1);
imshow(img)
```

Write Data to Amazon S3

To write data to Amazon S3, call the `write` function on a distributed or tall array, and provide the full path to a folder in the cloud storage. The following example shows how to read tabular data from Amazon S3 into a tall array, preprocess it by removing missing entries and sorting, and then write it back to Amazon S3.

```
setenv('AWS_ACCESS_KEY_ID', 'YOUR_AWS_ACCESS_KEY_ID');
setenv('AWS_SECRET_ACCESS_KEY', 'YOUR_AWS_SECRET_ACCESS_KEY');

ds = tabularTextDatastore('s3://bucketname/dataset/airlinesmall.csv', ...
  'TreatAsMissing', 'NA', 'SelectedVariableNames', {'ArrDelay'});
tt = tall(ds);
tt = sortrows(rmmissing(tt));
write('s3://bucketname/preprocessedData/',tt);
```

To read your tall data back, use the `datastore` function.

```
ds = datastore('s3://bucketname/preprocessedData/');
tt = tall(ds);
```

Microsoft Azure Storage Blob

MATLAB enables you to use Windows Azure® Storage Blob (WASB) as an online file storage web service offered by Microsoft. You can use data stored on Azure with datastore objects such as `ImageDatastore`, `FileDatastore`, `SpreadsheetDatastore`, or `TabularTextDatastore`. When you specify the location of the data, you must specify the full path to the files or folders using a uniform resource locator (URL) of the form

`wasbs://`*`container@account/path_to_file/file.ext`*

`container@account` is the name of the container and *`path_to_file`* is the path to the file or folders.

Azure provides data storage through web services interfaces. You can use a *blob* to store data files in Azure. See Introduction to Azure for more information.

Set Up Access

To work with remote data in Azure storage, you must set up access first:

1. Sign up for a Microsoft Azure account, see Microsoft Azure Account.
2. Set up your authentication details by setting exactly one of the two following environment variables using `setenv`:

- `MW_WASB_SAS_TOKEN` — Authentication via Shared Access Signature (SAS)

  Obtain an SAS. For details, see the "Get the SAS for a blob container" section in Manage Azure Blob Storage resources with Storage Explorer.

  In MATLAB, set `MW_WASB_SAS_TOKEN` to the SAS query string. For example,

  ```
  setenv MW_WASB_SAS_TOKEN '?st=2017-04-11T09%3A45%3A00Z&se=2017-05-12T09%3A45%3A00Z&sp=rl&sv=2015-12-11&sr=c&sig=
  ```

  You must set this string to a valid SAS token generated from the Azure Storage web UI or Explorer.
- `MW_WASB_SECRET_KEY` — Authentication via one of the Account's two secret keys

  Each Storage Account has two secret keys that permit administrative privilege access. This same access can be given to MATLAB without having to create an SAS token by setting the `MW_WASB_SECRET_KEY` environment variable. For example:

  ```
  setenv MW_WASB_SECRET_KEY '1234567890ABCDEF1234567890ABCDEF1234567890ABCDEF'
  ```

If you are using Parallel Computing Toolbox, you must copy your client environment variables to the workers on a cluster by setting `EnvironmentVariables` in parpool, batch, createJob or in the Cluster Profile Manager.

For more information, see Use Azure storage with Azure HDInsight clusters.

Read Data from Azure

To read data from a WASB location, use a datastore object. To produce the file location, start with the filename `file.ext`, and prefix it with the file path `/path_to_file` and your account name `wasbs://container@account/`. The complete data location uses the following syntax:

`wasbs://`*`container@account/path_to_file/file.ext`*

`container@account` is the name of the container and *`path_to_file`* is the path to the file or folders.

For example, if you have a file `airlinesmall.csv` in a folder `/airline` on a test storage account `wasbs://blobContainer@storageAccount.blob.core.windows.net/`, then you can create a datastore by using:

```
location = 'wasbs://blobContainer@storageAccount.blob.core.windows.net/airline/airlinesmall.csv';
```

```
ds = tabularTextDatastore(location, 'TreatAsMissing', 'NA', ...
  'SelectedVariableNames', {'ArrDelay'});
```

You can use Azure for all calculations datastores support, including direct reading, `mapreduce`, tall arrays and deep learning. For example, create an `ImageDatastore`, read a specified image from the datastore, and then display the image to screen.

```
setenv('MW_WASB_SAS_TOKEN', 'YOUR_WASB_SAS_TOKEN');
ds = imageDatastore('wasbs://YourContainer@YourAccount.blob.core.windows.net/', ...
  'IncludeSubfolders', true, 'LabelSource', 'foldernames');
img = ds.readimage(1);
imshow(img)
```

Write Data to Azure

To write data to Azure, call the `write` function on a distributed or tall array, and provide the full path to a folder in the cloud storage. The following example shows how to read tabular data from Azure into a tall array, preprocess it by removing missing entries and sorting, and then write it back to Azure.

```
setenv('MW_WASB_SAS_TOKEN', 'YOUR_WASB_SAS_TOKEN');

ds = tabularTextDatastore('wasbs://YourContainer@YourAccount.blob.core.windows.net/dataset/airli
  'TreatAsMissing', 'NA', 'SelectedVariableNames', {'ArrDelay'});
tt = tall(ds);
tt = sortrows(rmmissing(tt));
write('wasbs://YourContainer@YourAccount.blob.core.windows.net/preprocessedData/',tt);
```

To read your tall data back, use the `datastore` function.

```
ds = datastore('wasbs://YourContainer@YourAccount.blob.core.windows.net/preprocessedData/');
tt = tall(ds);
```

Hadoop Distributed File System

Specify Location of Data

You also can create a datastore for a collection of text files or sequence files that reside on the Hadoop Distributed File System (HDFS) using the `tabularTextDatastore` function. When you specify the location of the data, you must specify the full path to the files or folders using a uniform resource locator (URL) of one of these forms:

hdfs:/*path_to_file*

hdfs:///*path_to_file*

hdfs://*hostname*/*path_to_file*

hostname is the name of the host or server and *path_to_file* is the path to the file or folders. Specifying the *hostname* is optional. When you do not specify the *hostname*, Hadoop uses the default host name associated with the Hadoop Distributed File System (HDFS) installation in MATLAB.

For example, both these commands create a datastore for the file, `file1.txt`, in a folder named `data` located at a host named `myserver`:

- `ds = tabularTextDatastore('hdfs:///data/file1.txt')`
- `ds = tabularTextDatastore('hdfs://myserver/data/file1.txt')`

If *hostname* is specified, it must correspond to the namenode defined by the `fs.default.name` property in the Hadoop XML configuration files for your Hadoop cluster.

Optionally, you can include the port number. For example, this location specifies a host named `myserver` with port `7867`, containing the file `file1.txt` in a folder named `data`:

`'hdfs://myserver:7867/data/file1.txt'`

The specified port number must match the port number set in your HDFS configuration.

Set Hadoop Environment Variable

Before reading from HDFS, use the `setenv` function to set the appropriate environment variable to the folder where Hadoop is installed. This folder must be accessible from the current machine.

- Hadoop v1 only — Set the `HADOOP_HOME` environment variable.
- Hadoop v2 only — Set the `HADOOP_PREFIX` environment variable.
- If you work with both Hadoop v1 and Hadoop v2, or if the `HADOOP_HOME` and `HADOOP_PREFIX` environment variables are not set, then set the `MATLAB_HADOOP_INSTALL` environment variable.

For example, use this command to set the `HADOOP_HOME` environment variable. *`hadoop-folder`* is the folder where Hadoop is installed, and *`/mypath/`* is the path to that folder.

```
setenv('HADOOP_HOME','/mypath/hadoop-folder');
```

HDFS data on Hortonworks or Cloudera

If your current machine has access to HDFS data on Hortonworks or Cloudera®, then you do not have to set the `HADOOP_HOME` or `HADOOP_PREFIX` environment variables. MATLAB automatically assigns these environment variables when using Hortonworks or Cloudera application edge nodes.

Prevent Clearing Code from Memory

When reading from HDFS or when reading Sequence files locally, the `datastore` function calls the `javaaddpath` command. This command does the following:

- Clears the definitions of all Java classes defined by files on the dynamic class path
- Removes all global variables and variables from the base workspace
- Removes all compiled scripts, functions, and MEX-functions from memory

To prevent persistent variables, code files, or MEX-files from being cleared, use the `mlock` function.

Write Data to HDFS

Use the `write` function to write your tall and distributed arrays to a Hadoop Distributed File System. When you call this function on a distributed or tall array, you must specify the full path to a HDFS folder. The following example shows how to read tabular data from HDFS into a tall array, preprocess it by removing missing entries and sorting, and then write it back to HDFS.

```
ds = tabularTextDatastore('hdfs://myserver/some/path/dataset/airlinesmall.csv', ...
  'TreatAsMissing', 'NA', 'SelectedVariableNames', {'ArrDelay'});
tt = tall(ds);
tt = sortrows(rmmissing(tt));
write('hdfs://myserver/some/path/preprocessedData/',tt);
```

To read your tall data back, use the `datastore` function.

```
ds = datastore('hdfs://myserver/some/path/preprocessedData/');
tt = tall(ds);
```

See Also

`datastore` | `imageDatastore` | `imread` | `imshow` | `javaaddpath` | `mlock` | `setenv` | `tabularTextDatastore` | `write`

Related Examples

- “Read and Analyze Hadoop Sequence File” on page 12-105
- “Upload Deep Learning Data to the Cloud” (Parallel Computing Toolbox)

Read and Analyze Large Tabular Text File

This example shows how to create a datastore for a large text file containing tabular data, and then read and process the data one block at a time or one file at a time.

Create a Datastore

Create a datastore from the sample file `airlinesmall.csv` using the `tabularTextDatastore` function. When you create the datastore, you can specify that the text, `NA`, in the data is treated as missing data.

```
ds = tabularTextDatastore('airlinesmall.csv','TreatAsMissing','NA');
```

You can modify the properties of the datastore by changing its properties. Modify the `MissingValue` property to specify that missing values are treated as 0.

```
ds.MissingValue = 0;
```

In this example, select the variable for the arrival delay, `ArrDelay`, as the variable of interest.

```
ds.SelectedVariableNames = 'ArrDelay';
```

Preview the data using the `preview` function. This function does not affect the state of the datastore.

```
data = preview(ds)
```

```
data=8×1 table
    ArrDelay
    ________

        8
        8
       21
       13
        4
       59
        3
       11
```

Read Subsets of Data

By default, `read` reads from a `TabularTextDatastore` 20000 rows at a time. To read a different number of rows in each call to `read`, modify the `ReadSize` property of `ds`.

```
ds.ReadSize = 15000;
```

Read subsets of the data from `ds` using the `read` function in a `while` loop. The loop executes until `hasdata(ds)` returns `false`.

```
sums = [];
counts = [];
while hasdata(ds)
    T = read(ds);

    sums(end+1) = sum(T.ArrDelay);
    counts(end+1) = length(T.ArrDelay);
end
```

Compute the average arrival delay

```
avgArrivalDelay = sum(sums)/sum(counts)
```

```
avgArrivalDelay = 6.9670
```

Reset the datastore to allow rereading of the data.

```
reset(ds)
```

Read One File At a Time

A datastore can contain multiple files, each with a different number of rows. You can read from the datastore one complete file at a time by setting the `ReadSize` property to `'file'`.

```
ds.ReadSize = 'file';
```

When you change the value of `ReadSize` from a number to `'file'` or vice versa, MATLAB resets the datastore.

Read from `ds` using the `read` function in a `while` loop, as before, and compute the average arrival delay.

```
sums = [];
counts = [];
while hasdata(ds)
    T = read(ds);

    sums(end+1) = sum(T.ArrDelay);
    counts(end+1) = length(T.ArrDelay);
end
avgArrivalDelay = sum(sums)/sum(counts)
```

```
avgArrivalDelay = 6.9670
```

See Also

`mapreduce` | `tabularTextDatastore` | `tall`

Related Examples

- “Tall Arrays for Out-of-Memory Data” on page 12-136
- “Getting Started with MapReduce” on page 12-3

Read and Analyze Image Files

This example shows how to create a datastore for a collection of images, read the image files, and find the images with the maximum average hue, saturation, and brightness (HSV). For a similar example on image processing using the `mapreduce` function, see "Compute Maximum Average HSV of Images with MapReduce" on page 12-78.

Identify two MATLAB® directories and create a datastore containing images with `.jpg`, `.tif`, and `.png` extensions in those directories.

```
location1 = fullfile(matlabroot,'toolbox','matlab','demos');
location2 = fullfile(matlabroot,'toolbox','matlab','imagesci');

ds = imageDatastore({location1,location2},'FileExtensions',{'.jpg','.tif','.png'});
```

Initialize the maximum average HSV values and the corresponding image data.

```
maxAvgH = 0;
maxAvgS = 0;
maxAvgV = 0;

dataH = 0;
dataS = 0;
dataV = 0;
```

For each image in the collection, read the image file and calculate the average HSV values across all of the image pixels. If an average value is larger than that of a previous image, then record it as the new maximum (`maxAvgH`, `maxAvgS`, or `maxAvgV`) and record the corresponding image data (`dataH`, `dataS`, or `dataV`).

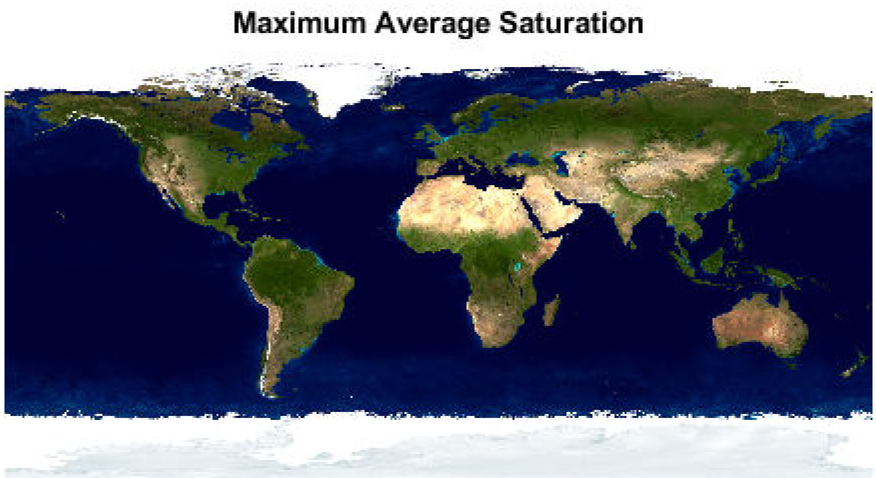
```
for i = 1:length(ds.Files)
    data = readimage(ds,i);     % Read the ith image
    if ~ismatrix(data)          % Only process 3-dimensional color data
        hsv = rgb2hsv(data);    % Compute the HSV values from the RGB data

        h = hsv(:,:,1);         % Extract the HSV values
        s = hsv(:,:,2);
        v = hsv(:,:,3);

        avgH = mean(h(:));      % Find the average HSV values across the image
        avgS = mean(s(:));
        avgV = mean(v(:));

        if avgH > maxAvgH       % Check for new maximum average hue
           maxAvgH = avgH;
           dataH = data;
        end

        if avgS > maxAvgS       % Check for new maximum average saturation
           maxAvgS = avgS;
           dataS = data;
        end

        if avgV > maxAvgV       % Check for new maximum average brightness
           maxAvgV = avgV;
           dataV = data;
        end
```

```
    end
end
```

View the images with the largest average hue, saturation, and brightness.

```
imshow(dataH,'InitialMagnification','fit');
title('Maximum Average Hue')
```

```
figure
imshow(dataS,'InitialMagnification','fit');
title('Maximum Average Saturation');
```




```
figure
imshow(dataV,'InitialMagnification','fit');
title('Maximum Average Brightness');
```

Maximum Average Brightness

See Also

`imageDatastore` | `mapreduce` | `tall`

Related Examples

- "Tall Arrays for Out-of-Memory Data" on page 12-136
- "Getting Started with MapReduce" on page 12-3
- "Compute Maximum Average HSV of Images with MapReduce" on page 12-78

Read and Analyze MAT-File with Key-Value Data

This example shows how to create a datastore for key-value pair data in a MAT-file that is the output of `mapreduce`. Then, the example shows how to read all the data in the datastore and sort it. This example assumes that the data in the MAT-file fits in memory.

Create a datastore from the sample file, `mapredout.mat`, using the `datastore` function. The sample file contains unique keys representing airline carrier codes and corresponding values that represent the number of flights operated by that carrier.

```
ds = datastore('mapredout.mat');
```

`datastore` returns a `KeyValueDatastore`. The `datastore` function automatically determines the appropriate type of datastore to create.

Preview the data using the `preview` function. This function does not affect the state of the datastore.

```
preview(ds)
```

```
ans=1×2 table
     Key        Value
    ______    _________

    {'AA'}    {[14930]}
```

Read all of the data in `ds` using the `readall` function. The `readall` function returns a table with two columns, `Key` and `Value`.

```
T = readall(ds)
```

```
T=29×2 table
       Key           Value
    __________    _________

    {'AA'    }    {[14930]}
    {'AS'    }    {[ 2910]}
    {'CO'    }    {[ 8138]}
    {'DL'    }    {[16578]}
    {'EA'    }    {[  920]}
    {'HP'    }    {[ 3660]}
    {'ML (1)'}    {[   69]}
    {'NW'    }    {[10349]}
    {'PA (1)'}    {[  318]}
    {'PI'    }    {[  871]}
    {'PS'    }    {[   83]}
    {'TW'    }    {[ 3805]}
    {'UA'    }    {[13286]}
    {'US'    }    {[13997]}
    {'WN'    }    {[15931]}
    {'AQ'    }    {[  154]}
      ⋮
```

`T` contains all the airline and flight data from the datastore in the same order in which the data was read. The table variables, `Key` and `Value`, are cell arrays.

Convert `Value` to a numeric array.

```
T.Value = cell2mat(T.Value)
```

```
T=29×2 table
        Key         Value
    ___________    _____

    {'AA'     }    14930
    {'AS'     }     2910
    {'CO'     }     8138
    {'DL'     }    16578
    {'EA'     }      920
    {'HP'     }     3660
    {'ML (1)'}        69
    {'NW'     }    10349
    {'PA (1)'}       318
    {'PI'     }      871
    {'PS'     }       83
    {'TW'     }     3805
    {'UA'     }    13286
    {'US'     }    13997
    {'WN'     }    15931
    {'AQ'     }      154
      ⋮
```

Assign new names to the table variables.

```
T.Properties.VariableNames = {'Airline','NumFlights'};
```

Sort the data in T by the number of flights.

```
T = sortrows(T,'NumFlights','descend')
```

```
T=29×2 table
    Airline    NumFlights
    _______    __________

    {'DL'}       16578
    {'WN'}       15931
    {'AA'}       14930
    {'US'}       13997
    {'UA'}       13286
    {'NW'}       10349
    {'CO'}        8138
    {'MQ'}        3962
    {'TW'}        3805
    {'HP'}        3660
    {'OO'}        3090
    {'AS'}        2910
    {'XE'}        2357
    {'EV'}        1699
    {'OH'}        1457
    {'FL'}        1263
      ⋮
```

View a summary of the sorted table.

```
summary(T)
```

```
Variables:

    Airline: 29x1 cell array of character vectors

    NumFlights: 29x1 double

        Values:

            Min            69
            Median       1457
            Max         16578
```

Reset the datastore to allow rereading of the data.

```
reset(ds)
```

See Also

`KeyValueDatastore` | `datastore` | `mapreduce` | `tall`

Related Examples

- "Tall Arrays for Out-of-Memory Data" on page 12-136
- "Getting Started with MapReduce" on page 12-3

Read and Analyze Hadoop Sequence File

This example shows how to create a datastore for a Sequence file containing key-value data. Then, you can read and process the data one block at a time. Sequence files are outputs of `mapreduce` operations that use Hadoop.

Set the appropriate environment variable to the location where Hadoop is installed. In this case, set the `MATLAB_HADOOP_INSTALL` environment variable.

```
setenv('MATLAB_HADOOP_INSTALL','/mypath/hadoop-folder')
```

`hadoop-folder` is the folder where Hadoop is installed and *`mypath`* is the path to that folder.

Create a datastore from the sample file, `mapredout.seq`, using the `datastore` function. The sample file contains unique keys representing airline carrier codes and corresponding values that represent the number of flights operated by that carrier.

```
ds = datastore('mapredout.seq')
```

```
ds = 
  KeyValueDatastore with properties:

       Files: {
              ' ...\matlab\toolbox\matlab\demos\mapredout.seq'
              }
    ReadSize: 1 key-value pairs
    FileType: 'seq'
```

`datastore` returns a `KeyValueDatastore`. The `datastore` function automatically determines the appropriate type of datastore to create.

Set the `ReadSize` property to six so that each call to `read` reads at most six key-value pairs.

```
ds.ReadSize = 6;
```

Read subsets of the data from `ds` using the `read` function in a `while` loop. For each subset of data, compute the sum of the values. Store the sum for each subset in an array named `sums`. The `while` loop executes until `hasdata(ds)` returns `false`.

```
sums = [];
while hasdata(ds)
    T = read(ds);
    T.Value = cell2mat(T.Value);
    sums(end+1) = sum(T.Value);
end
```

View the last subset of key-value pairs read.

```
T
```

```
T = 

      Key       Value
    ________    _____

    'WN'        15931
    'XE'         2357
```

```
    'YV'          849
    'ML (1)'       69
    'PA (1)'      318
```

Compute the total number of flights operated by all carriers.

```
numflights = sum(sums)
```

```
numflights =

      123523
```

See Also

KeyValueDatastore | datastore | mapreduce | tall

Related Examples

- “Getting Started with MapReduce” on page 12-3
- “Tall Arrays for Out-of-Memory Data” on page 12-136

Develop Custom Datastore

This topic shows how to implement a custom datastore for file-based data. Use this framework only when writing your own custom datastore interface. Otherwise, for standard file formats, such as images or spreadsheets, use an existing datastore from MATLAB. For more information, see “Getting Started with Datastore” on page 12-84.

Overview

To build your custom datastore interface, use the custom datastore classes and objects. Then, use the custom datastore to bring your data into MATLAB and leverage the MATLAB big data capabilities such as `tall`, `mapreduce`, and Hadoop.

Designing your custom datastore involves inheriting from one or more abstract classes and implementing the required methods. The specific classes and methods you need depend on your processing needs.

Processing Needs	Classes
Datastore for Serial Processing in MATLAB	`matlab.io.Datastore` See “Implement Datastore for Serial Processing” on page 12-108
Datastore with support for Parallel Computing Toolbox and MATLAB Parallel Server	`matlab.io.Datastore` and `matlab.io.datastore.Partitionable` See “Add Support for Parallel Processing” on page 12-110
Datastore with support for Hadoop	`matlab.io.Datastore` and `matlab.io.datastore.HadoopLocationBased` See “Add Support for Hadoop” on page 12-111
Datastore with support for shuffling samples in a datastore in random order	`matlab.io.Datastore` and `matlab.io.datastore.Shuffleable` See “Add Support for Shuffling” on page 12-112
Datastore with support for writing files via `writeall`	`matlab.io.Datastore` and `matlab.io.datastore.FileWritable` (Optionally, inheriting from `matlab.io.datastore.FoldersPropertyProvider` adds support for a `Folders` property.) See “Add Support for Writing Data” on page 12-113

Start by implementing datastore for serial processing, and then add support for parallel processing, Hadoop, shuffling, or writing.

Implement Datastore for Serial Processing

To implement a custom datastore named `MyDatastore`, create a script `MyDatastore.m`. The script must be on the MATLAB path and should contain code that inherits from the appropriate class and defines the required methods. The code for creating a datastore for serial processing in MATLAB must:

- Inherit from the base class `matlab.io.Datastore`.
- Define these methods: `hasdata`, `read`, `reset`, and `progress`.
- Define additional properties and methods based on your data processing and analysis needs.

For a sample implementation, follow these steps.

Steps | **Implementation**

Inherit from the base class `Datastore`.

```
classdef MyDatastore < matlab.io.Datastore

    properties (Access = private)
        CurrentFileIndex double
        FileSet matlab.io.datastore.DsFileSet
    end
```

Add this property to create a datastore on one machine that works seamlessly on another machine or cluster that possibly has a different file system or operating system.

Add methods to get and set this property in the methods section.

```
    % Property to support saving, loading, and processing of
    % datastore on different file system machines or clusters.
    % In addition, define the methods get.AlternateFileSystemRoots()
    % and set.AlternateFileSystemRoots() in the methods section.
    properties(Dependent)
        AlternateFileSystemRoots
    end
```

Implement the function `MyDatastore` that creates the custom datastore.

```
    methods % begin methods section

        function myds = MyDatastore(location,altRoots)
            myds.FileSet = matlab.io.datastore.DsFileSet(location,...
                'FileExtensions','.bin', ...
                'FileSplitSize',8*1024);
            myds.CurrentFileIndex = 1;

            if nargin == 2
                 myds.AlternateFileSystemRoots = altRoots;
            end

            reset(myds);
        end
```

Implement the `hasdata` method.

```
        function tf = hasdata(myds)
            % Return true if more data is available.
            tf = hasfile(myds.FileSet);
        end
```

Steps	Implementation
Implement the `read` method. This method uses `MyFileReader`, which is a function that you must create to read your proprietary file format . See "Create Function to Read Your Proprietary File Format" on page 12-110.	see code below
Implement the `reset` method.	see code below
Define the methods to get and set the `AlternateFileSystemRoots` property. You must reset the datastore in the `set` method.	see code below
Implement the `progress` method.	see code below

Implement the `read` method.

```
function [data,info] = read(myds)
    % Read data and information about the extracted data.
    if ~hasdata(myds)
        error(sprintf(['No more data to read.\nUse the reset ',...
            'method to reset the datastore to the start of ' ,...
            'the data. \nBefore calling the read method, ',...
            'check if data is available to read ',...
            'by using the hasdata method.']))

    end

    fileInfoTbl = nextfile(myds.FileSet);
    data = MyFileReader(fileInfoTbl);
    info.Size = size(data);
    info.FileName = fileInfoTbl.FileName;
    info.Offset = fileInfoTbl.Offset;

    % Update CurrentFileIndex for tracking progress
    if fileInfoTbl.Offset + fileInfoTbl.SplitSize >= ...
            fileInfoTbl.FileSize
        myds.CurrentFileIndex = myds.CurrentFileIndex + 1 ;
    end


end
```

Implement the `reset` method.

```
function reset(myds)
    % Reset to the start of the data.
    reset(myds.FileSet);
    myds.CurrentFileIndex = 1;
end
```

Define the methods to get and set the `AlternateFileSystemRoots` property.

You must reset the datastore in the `set` method.

```
% Before defining these methods, add the AlternateFileSystemRoots
% property in the properties section

% Getter for AlternateFileSystemRoots property
function altRoots = get.AlternateFileSystemRoots(myds)
    altRoots = myds.FileSet.AlternateFileSystemRoots;
end

% Setter for AlternateFileSystemRoots property
function set.AlternateFileSystemRoots(myds,altRoots)
    try
      % The DsFileSet object manages the AlternateFileSystemRoots
      % for your datastore
      myds.FileSet.AlternateFileSystemRoots = altRoots;

      % Reset the datastore
      reset(myds);
    catch ME
      throw(ME);
    end
end

end
```

Implement the `progress` method.

```
methods (Hidden = true)
    function frac = progress(myds)
        % Determine percentage of data read from datastore
        if hasdata(myds)
           frac = (myds.CurrentFileIndex-1)/...
                         myds.FileSet.NumFiles;
        else
          frac = 1;
        end
    end
end
```

Steps	Implementation

Implement the `copyElement` method when you use the `DsFileSet` object as a property in your datastore.	<pre> methods (Access = protected)
	        % If you use the DsFileSet object as a property, then
	        % you must define the copyElement method. The copyElement
	        % method allows methods such as readall and preview to
	        % remain stateless
	        function dscopy = copyElement(ds)
	            dscopy = copyElement@matlab.mixin.Copyable(ds);
	            dscopy.FileSet = copy(ds.FileSet);
	        end
	    end</pre>

End the `classdef` section.	`end`

Create Function to Read Your Proprietary File Format

The implementation of the `read` method of your custom datastore uses a function called `MyFileReader`. You must create this function to read your custom or proprietary data. Build this function using `DsFileReader` object and its methods. For instance, create a function that reads binary files.

```
function data = MyFileReader(fileInfoTbl)
% create a reader object using the FileName
reader = matlab.io.datastore.DsFileReader(fileInfoTbl.FileName);

% seek to the offset
seek(reader,fileInfoTbl.Offset,'Origin','start-of-file');

% read fileInfoTbl.SplitSize amount of data
data = read(reader,fileInfoTbl.SplitSize);
end
```

Add Support for Parallel Processing

To add support for parallel processing with Parallel Computing Toolbox and MATLAB Parallel Server, update your implementation code in `MyDatastore.m` to:

- Inherit from an additional class `matlab.io.datastore.Partitionable`.
- Define two additional methods: `maxpartitions` and `partition`.

For a sample implementation, follow these steps.

Steps	Implementation

Update the `classdef` section to inherit from the `Partitionable` class.	<pre>classdef MyDatastore < matlab.io.Datastore & ...
	                       matlab.io.datastore.Partitionable
	     .
	     .
	     .</pre>

Steps	Implementation

Add the definition for `partition` to the `methods` section.	`methods`
	`    .`
	`    .`
	`    .`
	`    function subds = partition(myds,n,ii)`
	`        subds = copy(myds);`
	`        subds.FileSet = partition(myds.FileSet,n,ii);`
	`        reset(subds);`
	`    end`
	`end`

Add definition for `maxpartitions` to the `methods` section.	`methods (Access = protected)`
	`    function n = maxpartitions(myds)`
	`      n = maxpartitions(myds.FileSet);`
	`    end`
	`end`

End `classdef`.	`end`

Add Support for Hadoop

To add support for Hadoop, update your implementation code in `MyDatastore.m` to:

- Inherit from an additional class `matlab.io.datastore.HadoopLocationBased`.
- Define two additional methods: `getLocation` and `initializeDatastore`.

For a sample implementation, follow these steps.

Steps	Implementation

Update the `classdef` section to inherit from the `HadoopLocationBased` class.	`classdef MyDatastore < matlab.io.Datastore & ...`
	`                       matlab.io.datastore.HadoopLocationBased`
	`    .`
	`    .`
	`    .`

Steps	Implementation

Add the definition for `getLocation`, `initializeDatastore`, and `isfullfile` (optional) to the `methods` section.	<pre>methods (Hidden = true)
	    .
	    .
	    .
	    function initializeDatastore(myds,hadoopInfo)
	       import matlab.io.datastore.DsFileSet;
	       myds.FileSet = DsFileSet(hadoopInfo,...
	            'FileSplitSize',myds.FileSet.FileSplitSize);
	       reset(myds);
	    end
	
	    function loc = getLocation(myds)
	       loc = myds.FileSet;
	    end
	
	    % isfullfile method is optional
	    function tf = isfullfile(myds)
	       tf = isequal(myds.FileSet.FileSplitSize,'file');
	    end
	
	end</pre>

End the `classdef` section.	`end`

Add Support for Shuffling

To add support for shuffling, update your implementation code in `MyDatastore.m` to:

- Inherit from an additional class `matlab.io.datastore.Shuffleable`.
- Define the additional method `shuffle`.

For a sample implementation, follow these steps.

Steps	Implementation

Update the `classdef` section to inherit from the `Shuffleable` class.	<pre>classdef MyDatastore < matlab.io.Datastore & ...
	                      matlab.io.datastore.Shuffleable
	    .
	    .
	    .</pre>

Steps	Implementation

Add the definition for `shuffle` to the existing `methods` section.	<pre>methods
	
	    % previously defined methods
	    .
	    .
	    .
	
	    function dsNew = shuffle(ds)
	        % dsNew = shuffle(ds) shuffles the files and the
	        % corresponding labels in the datastore.
	
	        % Create a copy of datastore
	        dsNew = copy(ds);
	        dsNew.Datastore = copy(ds.Datastore);
	        fds = dsNew.Datastore;
	
	        % Shuffle files and corresponding labels
	        numObservations = dsNew.NumObservations;
	        idx = randperm(numObservations);
	        fds.Files = fds.Files(idx);
	        dsNew.Labels = dsNew.Labels(idx);
	    end
	
	end</pre>

End the `classdef` section.	`end`

Add Support for Writing Data

To add support for writing data, update your implementation code in `MyDatastore.m` to follow these requirements:

- Inherit from an additional class `matlab.io.datastore.FileWritable`.
- Initialize the properties `SupportedOutputFormats` and `DefaultOutputFormat`.
- Implement a `write` method if the datastore writes data to a custom format.
- Implement a `getFiles` method if the datastore does not have a `Files` property.
- Implement a `getFolders` method if the datastore does not have a `Folders` property.
- The output location is validated as a string. If your datastore requires further validation, you must implement a `validateOutputLocation` method.
- If the datastore is meant for files that require multiple reads per file, then you must implement the methods `getCurrentFilename` and `currentFileIndexComparator`.
- For Parquet file writing, the custom datastore must have a property named `ReadSize` with the value `file`. If the datastore does not have this property, then it must implement the method `isSingleReadPerFile`.
- Optionally, inherit from another class `matlab.io.datastore.FoldersPropertyProvider` to add support for a `Folders` property (and thus the `FolderLayout` name-value pair of `writeall`). If you do this, then you can use the `populateFoldersFromLocation` method in the datastore constructor to populate the `Folders` property.
- To add support for the `'UseParallel'` option of `writeall`, you must subclass from both `matlab.io.datastore.FileWritable` and `matlab.io.datastore.Partitionable` and

implement a `partition` method in the subclass that supports the syntax `partition(ds,'Files',index)`.

For a sample implementation that inherits from `matlab.io.datastore.FileWritable`, follow these steps.

Steps	Implementation

Update the `classdef` section to inherit from the `FileWritable` class.	`classdef MyDatastore < matlab.io.Datastore & ...`
	`matlab.io.datastore.FileWritable`
	`.`
	`.`
	`.`

Initialize the properties `SupportedOutputFormats` and `DefaultOutputFormat`. In this example, the datastore supports all of the output formats of `ImageDatastore`, as well as a custom format `"dcm"`, which is also declared as the default output format.	`properties (Constant)`
	`SupportedOutputFormats = ...`
	`[matlab.io.datastore.ImageDatastore.SupportedOutputFormats, "dcm"];`
	`DefaultOutputFormat = "dcm";`
	`end`

Add definitions for `getFiles` and `getFolders` to the existing `methods` section. These methods are required when the datastore does not have `Files` or `Folders` properties.	`methods (Access = protected)`
	`function files = getFiles(ds)`
	`files = {'data/folder/file1', 'data/folder/file2',...};`
	`end`
	`function folders = getFolders(ds)`
	`folders = {'data/folder1/', 'data/folder2/',...};`
	`end`
	`end`

Add a `write` method when the datastore intends to write data to a custom format. In this example, the method switches between using a custom write function for `"dcm"` and the built-in write function for known formats.	`methods(Access = protected)`
	`function tf = write(myds, data, writeInfo, outFmt, varargin)`
	`if outFmt == "dcm" % use custom write fcn for dcm format`
	`dicomwrite(data, writeInfo.SuggestedOutputName, varargin{:});`
	`else % callback into built-in for known formats`
	`write@matlab.io.datastore.FileWritable(myds, data, ...`
	`writeInfo, outFmt, varargin{:});`
	`end`
	`tf = true;`
	`end`
	`end`

End the `classdef` section.	`end`

For a longer example class that inherits from both `matlab.io.datastore.FileWritable` and `matlab.io.datastore.FoldersPropertyProvider`, see “Develop Custom Datastore for DICOM Data” on page 12-124.

Validate Custom Datastore

After following the instructions presented here, the implementation step of your custom datastore is complete. Before using this custom datastore, qualify it using the guidelines presented in “Testing Guidelines for Custom Datastores” on page 12-116.

See Also

`matlab.io.Datastore` | `matlab.io.datastore.DsFileReader` | `matlab.io.datastore.DsFileSet` | `matlab.io.datastore.FileWritable` | `matlab.io.datastore.FoldersPropertyProvider` | `matlab.io.datastore.HadoopLocationBased` | `matlab.io.datastore.Partitionable` | `matlab.io.datastore.Shuffleable`

More About

- “Developing Classes — Typical Workflow”
- “Create and Share Toolboxes”
- “Create Help for Classes”
- “Develop Custom Datastore for DICOM Data” on page 12-124

Testing Guidelines for Custom Datastores

All datastores that are derived from the custom datastore classes share some common behaviors. This test procedure provides guidelines to test the minimal set of behaviors and functionalities that all custom datastores should have. You will need additional tests to qualify any unique functionalities of your custom datastore.

If you have developed your custom datastore based on instructions in “Develop Custom Datastore” on page 12-107, then follow these test procedures to qualify your custom datastore. First perform the unit tests, followed by the workflow tests:

- Unit tests qualify the datastore constructor and methods.
- Workflow tests qualify the datastore usage.

For all these test cases:

- Unless specified in the test description, assume that you are testing a nonempty datastore `ds`.
- Verify the test cases on the file extensions, file encodings, and data locations (like Hadoop) that your custom datastore is designed to support.

Unit Tests

Construction

The unit test guidelines for the datastore constructor are as follows.

Test Case Description	Expected Output
Check if your custom datastore constructor works with the minimal required inputs.	Datastore object of your custom datastore type with the minimal expected properties and methods
Check if your datastore object `ds` has `matlab.io.Datastore` as one of its superclasses. Run this command: `isa(ds,'matlab.io.Datastore')`	`1` or `true`
Call your custom datastore constructor with the required inputs and any supported input arguments and name-value pair arguments.	Datastore object of your custom datastore type with the minimal expected properties and methods

read

Unit test guidelines for the `read` method

Test Case Description	Expected Output
Call the `read` method on a datastore object `ds`. `t = read(ds);`	Data from the beginning of the datastore If you specify read size, then the size of the returned data is equivalent to read size.

<table>
<tr><th>Test Case Description</th><th>Expected Output</th></tr>
<tr><td>Call the <code>read</code> method again on the datastore object.

<code>t = read(ds);</code></td><td>Data starting from the end point of the previous read operation

If you specify read size, then the size of the returned data is equivalent to read size.</td></tr>
<tr><td>Continue calling the <code>read</code> method on the datastore object in a while loop.

<code>while(hasdata(ds))</code>
<code> t = read(ds);</code>
<code>end</code></td><td>No errors

Correct data in the correct format</td></tr>
<tr><td>When data is available to read, check the <code>info</code> output (if any) of the <code>read</code> method.

Call a datastore object <code>ds</code>.

<code>[t,info] = read(ds);</code></td><td>No error

<code>info</code> contains the expected information

<code>t</code> contains the expected data</td></tr>
<tr><td>When no more data is available to read, call <code>read</code> on the datastore object.</td><td>Either expected output or an error message based on your custom datastore implementation.</td></tr>
</table>

readall

Unit test guidelines for the `readall` method

<table>
<tr><th>Test Case Description</th><th>Expected Output</th></tr>
<tr><td>Call the <code>readall</code> method on the datastore object.</td><td>All data</td></tr>
<tr><td>Call the <code>readall</code> method on the datastore object, when <code>hasdata(ds)</code> is <code>false</code>.

Read from the datastore until <code>hasdata(ds)</code> is <code>false</code>, and then call the <code>readall</code> method.

<code>while(hasdata(ds))</code>
<code> t = read(ds);</code>
<code>end</code>

<code>readall(ds)</code></td><td>All data</td></tr>
</table>

hasdata

Unit test guidelines for the `hasdata` method

Test Case Description	Expected Output
Call the `hasdata` method on the datastore object before making any calls to `read`	`true`
Call the `hasdata` method on the datastore object after making a few calls to `read`, but before all the data is read	`true`

Test Case Description	Expected Output
When more data is available to read, call the `readall` method, and then call the `hasdata` method.	`true`
When no more data is available to read, call the `hasdata` method.	`false`

reset

Unit test guidelines for the `reset` method

Test Case Description	Expected Output

Call the `reset` method on the datastore object before making any calls to the `read` method.
 	
Verify that the `read` method returns the appropriate data after a call to the `reset` method.
 	
`reset(ds);`
`t = read(ds);`
	No errors
	
	The `read` returns data from the beginning of the datastore.
	
	If you specify read size, then the size of the returned data is equivalent to read size.

When more data is available to read, call the `reset` method after making a few calls to the `read` method. Verify that the `read` method returns the appropriate data after making a call to the `reset` method.	No errors The `read` method returns data from the beginning of the datastore. If you specify read size, then the size of the returned data is equivalent to read size.
When more data is available to read, call the `reset` method after making a call to the `readall` method. Verify that the `read` method returns the appropriate data after making a call to the `reset` method.	No errors The `read` method returns data from the beginning of the datastore. If you specify read size, then the size of the returned data is equivalent to read size.
When no more data is available to read, call the `reset` method on the datastore object and then call the `read` method Verify that `read` returns the appropriate data after a call to the `reset` method.	No errors The `read` method returns data from the beginning of the datastore. If you specify read size, then the size of the returned data is equivalent to read size.

progress

Unit test guidelines for the `progress` method

Test Case Description	Expected Output
Call the `progress` method on the datastore object before making any calls to the `read` method.	`0` or an expected output based on your custom datastore implementation.

Test Case Description	Expected Output

Call the `progress` method on the datastore object after making a call to `readall`, but before making any calls to `read`
 	
`readall(ds);`
`progress(ds)`
	`0` or an expected output based on your custom datastore implementation.

Call the `progress` method on the datastore object after making a few calls to `read` and while more data is available to read.	A fraction between `0` and `1` or an expected output based on your custom datastore implementation.
Call the `progress` method on the datastore object when no more data is available to read.	`1` or an expected output based on your custom datastore implementation.

preview

Unit test guidelines for the `preview` method

Test Case Description	Expected Output
Call `preview` on the datastore object before making any calls to `read`.	The `preview` method returns the expected data from the beginning of the datastore, based on your custom datastore implementation.
Call `preview` on the datastore object after making a few calls to `read` and while more data is available to read.	The `preview` method returns the expected data from the beginning of the datastore, based on your custom datastore implementation.
Call `preview` on the datastore object after making a call to `readall` and while more data is available to read.	The `preview` method returns the expected data from the beginning of the datastore, based on your custom datastore implementation.
Call `preview` on the datastore object after making a few calls to `read` and a call to `reset`.	The `preview` method returns the expected data from the beginning of the datastore, based on your custom datastore implementation.
Call `preview` on the datastore object when no more data is available to read.	The `preview` method returns the expected data from the beginning of the datastore, based on your custom datastore implementation.
Call `preview` after making a few calls to `read` method and then call `read` again.	The `read` method returns data starting from the end point of the previous read operation. If you specify read size, then the size of the returned data is equivalent to read size.
Call `preview`, and then call `readall` on the datastore.	The `readall` method returns all the data from the datastore.

Test Case Description	Expected Output
While datastore has data available to read, call `preview`, and then call `hasdata`.	The `hasdata` method returns `true`.

partition

Unit test guidelines for the `partition` method

Test Case Description	Expected Output

Call `partition` on the datastore object `ds` with a valid number of partitions and a valid partition index.
 	
Call `read` on a partition of the datastore and verify the data.
 	
`subds = partition(ds,n,index)`
`read(subds)`
 	
Verify that the partition is valid.
 	
`isequal(properties(ds),properties(subds))`
`isequal(methods(ds),methods(subds))`
	The `partition` method partitions the datastore into `n` partitions and returns the partition corresponding to the specified `index`.
	
	The returned partition `subds` must be a datastore object of your custom datastore.
	
	The partitioned datastore `subds` must have the same methods and properties as the original datastore.
	
	The `isequal` statement returns `true`.
	
	Calling `read` on the partition returns data starting from the beginning of the partition.
	
	If you specify read size, then the size of the returned data is equivalent to read size.

Call `partition` on the datastore object `ds` with number of partitions specified as `1` and `index` of returned partition specified as `1`.
 	
Verify the data returned by calling `read` and `preview` on a partition of the partitioned datastore.
 	
`subds = partition(ds,1,1)`
`isequal(properties(ds),properties(subds))`
`isequal(methods(ds),methods(subds))`
`isequaln(read(subds),read(ds))`
`isequaln(preview(subds),preview(ds))`
	The partition `subds` must be a datastore object of your custom datastore.
	
	The partition `subds` must have the same methods and properties as the original datastore `ds`.
	
	The `isequal` and `isequaln` statements returns `true`.

Call `partition` on the partition `subds` with a valid number of partitions and a valid partition index.	The repartitioning of a partition of the datastore should work without errors.

initializeDatastore

If your datastore inherits from `matlab.io.datastore.HadoopFileBased`, then verify the behavior of `initializeDatastore` using the guidelines in this table.

<table>
<tr><th>Test Case Description</th><th>Expected Output</th></tr>
<tr><td>
Call <code>initializeDatastore</code> on the datastore object <code>ds</code> with a valid <code>info</code> struct.
The <code>info</code> struct contains these fields:
<code>FileName</code>
<code>Offset</code>
<code>Size</code>
<code>FileName</code> is of data type <code>char</code> and the fields <code>Offset</code> and <code>Size</code> are of the data type double.
For example, initialize the <code>info</code> struct, and then call <code>initializeDatastore</code> on the datastore object <code>ds</code>.
<pre>info = struct('FileName','myFileName.ext',...
                         'Offset',0,'Size',500)
initializeDatastore(ds,info)</pre>
Verify the initialization by examining the properties of your datastore object.
<pre>ds</pre>
</td><td>The <code>initializeDatastore</code> method initializes the custom datastore object <code>ds</code> with the necessary information from the <code>info</code> struct.</td></tr>
</table>

getLocation

If your datastore inherits from `matlab.io.datastore.HadoopFileBased`, then verify the behavior of `getLocation` using these guidelines.

<table>
<tr><th>Test Case Description</th><th>Expected Output</th></tr>
<tr><td>
Call <code>getLocation</code> on the datastore object.
<pre>location = getLocation(ds)</pre>
Based on your custom datastore implementation, the <code>location</code> output is either of these:
List of files or directories
a <code>matlab.io.datastore.DsFileSet</code> object
If <code>location</code> is a <code>matlab.io.datastore.DsFileSet</code> object, then call <code>resolve</code> to verify the files in the <code>location</code> output.
<pre>resolve(location)</pre>
</td><td>The <code>getLocation</code> method returns the location of files in Hadoop.</td></tr>
</table>

isfullfile

If your datastore inherits from `matlab.io.datastore.HadoopFileBased`, then verify the behavior of `isfullfile` using these guidelines.

Test Case Description	Expected Output
Call `isfullfile` on the datastore object.	Based on your custom datastore implementation, the `isfullfile` method returns `true` or `false`.

Workflow Tests

Verify your workflow tests in the appropriate environment.

- If your datastore inherits only from `matlab.io.Datastore`, then verify all workflow tests in a local MATLAB session.
- If your datastore has parallel processing support (inherits from `matlab.io.datastore.Partitionable`), then verify your workflow tests in parallel execution environments, such as Parallel Computing Toolbox and MATLAB Parallel Server.
- If your datastore has Hadoop support (inherits from `matlab.io.datastore.HadoopFileBased`), then verify your workflow tests in a Hadoop cluster.

Tall Workflow

Testing guidelines for the `tall` workflow

Test Case Description	Expected Output
Create a tall array by calling `tall` on the datastore object `ds`. `t = tall(ds)`	The `tall` function returns an output that is the same data type as the output of the `read` method of the datastore.
For this test step, create a datastore object with data that fits in your system memory. Then, create a tall array using this datastore object. `t = tall(ds)`	No errors The function returns an output of the correct data type (not of a `tall` data type).
If your data is numeric, then apply an appropriate function like the `mean` function to both the `ds` and `t`, then compare the results. If your data is of the data type `string` or `categorical`, then apply the `unique` function on a column of `ds` and a column of `t`, then compare the results. Apply `gather` and verify the result. For examples, see “Big Data Workflow Using Tall Arrays and Datastores” (Parallel Computing Toolbox).	The function returns the same result whether it is applied to `ds` or to `t`.

MapReduce Workflow

Testing guidelines for the MapReduce workflow

Test Case Description	Expected Output
Call `mapreduce` on the datastore object `ds`. `outds = mapreduce(ds,@mapper,@reducer)` For more information, see `mapreduce`. To support the use of the `mapreduce` function, the `read` method of your custom datastore must return both the `info` and the `data` output arguments.	No error The MapReduce operation returns the expected result

Next Steps

Note This test procedure provides guidelines to test the minimal set of behaviors and functionalities for custom datastores. Additional tests are necessary to qualify any unique functionalities of your custom datastore.

After you complete the implementation and validation of your custom datastore, your custom datastore is ready to use.

- To add help for your custom datastore implementation, see “Create Help for Classes”.
- To share your custom datastore with other users, see “Create and Share Toolboxes”.

See Also

`matlab.io.Datastore` | `matlab.io.datastore.HadoopLocationBased` | `matlab.io.datastore.Partitionable`

More About

- “Develop Custom Datastore” on page 12-107
- “Create and Share Toolboxes”
- “Create Help for Classes”

Develop Custom Datastore for DICOM Data

This example shows how to develop a custom datastore that supports writing operations. The datastore is named `DICOMDatastore` because it supports DICOM ® (Digital Imaging and Communications in Medicine) data, which is an international standard for medical imaging information.

Developing Custom Datastores

The topic “Develop Custom Datastore” on page 12-107 describes the general process for creating a custom datastore, as well as the specific requirements to add different pieces of functionality. There are a variety of superclasses you can subclass from depending on what pieces of functionality you need (parallel evaluation, writing operations, shuffling, and so on). In particular, you can add support for writing operations by subclassing from `matlab.io.datastore.FileWritable`. However, for the broadest set of writing functionality, you must also subclass from `matlab.io.datastore.FoldersPropertyProvider`, which adds a `Folders` property to the datastore. The complete requirements to add writing support to a custom datastore are covered in “Add Support for Writing Data” on page 12-113.

Class Definition

This table contains code and explanations for the `DICOMDatastore` class.

`classdef DICOMDatastore < matlab.io.Datastore & ...`
`                         matlab.io.datastore.FileWritable & ...`
`                         matlab.io.datastore.FoldersPropertyProvider`
	Class Header
	
	`DICOMDatastore` inherits from `Datastore` for basic functionality, as well as from `FileWritable` and `FoldersPropertyProvider` to enable file writing capabilities.

`    properties`
`        Files matlab.io.datastore.FileSet`
`    end`
	Public Properties
	
	`DICOMDatastore` defines a public `Files` property that is a `FileSet` object. `DICOMDatastore` inherits a `Folders` property from `FoldersPropertyProvider`, so that property does not need to be initialized.

`    properties (Constant)`
`        SupportedOutputFormats = ...`
`          [matlab.io.datastore.ImageDatastore.SupportedOutputFormats, "dcm"];`
`        DefaultOutputFormat = "dcm";`
`    end`
	`DICOMDatastore` defines `SupportedOutputFormats` and `DefaultOutputFormat` as constant properties with default values. `"dcm"` is a custom format for DICOM data.

```
    methods(Access = public)

        function myds = DICOMDatastore(location)
            % The class constructor to set properties of the datastore.
            myds.Files = matlab.io.datastore.FileSet(location, ...
                "IncludeSubfolders", true);
            populateFoldersFromLocation(myds,location);
            reset(myds);
        end
```

Public Methods

The public methods section defines common datastore methods that the class uses to manipulate data. Public methods are externally accessible, so class users of `DICOMDatastore` can call these methods (in addition to other public methods inherited from the superclasses).

The constructor `DICOMDatastore` creates a new `DICOMDatastore` object by setting values for the `Files` and `Folders` properties.

- Use `FileSet` to set the value of the `Files` property.
- Use the `populateFoldersFromLocation` method of `FoldersPropertyProvider` to set the value of the `Folders` property.

```
        function tf = hasdata(myds)
            %HASDATA   Returns true if more data is available.
            %   Return logical scalar indicating availability of data.
            %   This method should be called before calling read. This
            %   is an abstract method and must be implemented by the
            %   subclasses. hasdata is used in conjunction with read to
            %   read all the data within the datastore.
            tf = hasNextFile(myds.Files);
        end
```

The `hasdata`, `read`, `reset`, and `progress` methods define the infrastructure for the datastore to work with small chunks of data at a time. These are abstract methods that must be implemented by the subclass.

```
        function [data, info] = read(myds)
            %READ   Read data and information about the extracted data.
            %   Return the data extracted from the datastore in the
            %   appropriate form for this datastore. Also return
            %   information about where the data was extracted from in
            %   the datastore. Both the outputs are required to be
            %   returned from the read method and can be of any type.
            %   info is recommended to be a struct with information
            %   about the chunk of data read. data represents the
            %   underlying class of tall, if tall is created on top of
            %   this datastore. This is an abstract method and must be
            %   implemented by the subclasses.

            % In this example, the read method reads data from the
            % datastore using a custom reader function, MyFileReader,
            % which takes the resolved filenames as input.
            if ~hasdata(myds)
                error("No more data to read.\nUse reset method to " ...
                    + "reset the datastore to the start of the data. Before " ...
                    + "calling the read method, check if data is available " ...
                    + "to read by using the hasdata method.");
            end

            file = nextfile(myds.Files);
            try
                data = dicomread(file.Filename);
            catch ME
                error("%s has failed", file.FileName);
            end

            info.FileSize = size(data);
            info.Filename = file.Filename;
        end
```

```
        function reset(myds)
            %RESET   Reset to the start of the data.
            %   Reset the datastore to the state where no data has been
            %   read from it. This is an abstract method and must be
            %   implemented by the subclasses.

            % In this example, the datastore is reset to point to the
            % first file (and first partition) in the datastore.
            reset(myds.Files);
        end
```

```
        function frac = progress(myds)
            %PROGRESS   Percentage of consumed data between 0.0 and 1.0.
            %   Return fraction between 0.0 and 1.0 indicating progress as a
            %   double. The provided example implementation returns the
            %   ratio of the index of the current file from FileSet
            %   to the number of files in FileSet. A simpler
            %   implementation can be used here that returns a 1.0 when all
            %   the data has been read from the datastore, and 0.0
            %   otherwise.
            %
            %   See also matlab.io.Datastore, read, hasdata, reset, readall,
            %   preview.
            frac = progress(myds.Files);
        end

    end
```

```
methods(Access = protected)

        function dsCopy = copyElement(myds)
            %COPYELEMENT   Create a deep copy of the datastore
            %   Create a deep copy of the datastore. We need to call
            %   copy on the datastore's property FileSet because it is
            %   a handle object. Creating a deep copy allows methods
            %   such as readall and preview, which call the copy method,
            %   to remain stateless.
            dsCopy = copyElement@matlab.mixin.Copyable(myds);
            dsCopy.Files = copy(myds.Files);
        end
```

Protected Methods

Protected methods redefine methods that were inherited by the class, and they are only accessible to `DICOMDatastore`. For more information, see “Modify Inherited Methods”.

The protected `copyElement` method is required whenever `FileSet` is used to define properties. The `copyElement` method allows methods such as `readall` and `preview` to remain stateless.

```
        function tf = write(myds, data, writeInfo, outFmt, varargin)
            if outFmt == "dcm"
                dicomwrite(data, writeInfo.SuggestedOutputName, varargin{:});
            else
                write@matlab.io.datastore.FileWritable(myds, data, ...
                    writeInfo, outFmt, varargin{:});
            end
            tf = true;
        end
```

The protected `write` method writes out chunks of data. Since `DICOMDatastore` supports `ImageDatastore` formats as well as the custom format `"dcm"`, the `write` method uses different functions to write the data depending on the output format.

```
        function files = getFiles(myds)
            files = myds.Files.FileInfo.Filename;
        end

    end
```

The protected `getFiles` method is necessary since `DICOMDatastore` uses `FileSet` objects for the `Files` property. The `Files` property is generally required to return a `cellstr`, so the `getFiles` method uses the `FileSet` object to generate a `cellstr` of the file paths.

```
end
```

End the `classdef` section.

Expand for Class Code

```
classdef DICOMDatastore <   matlab.io.Datastore & ...
                            matlab.io.datastore.FileWritable & ...
                            matlab.io.datastore.FoldersPropertyProvider
    properties
        Files matlab.io.datastore.FileSet
    end
```

```
    properties (Constant)
        SupportedOutputFormats = ...
            [matlab.io.datastore.ImageDatastore.SupportedOutputFormats, "dcm"];
        DefaultOutputFormat = "dcm";
    end

    methods(Access = public)
        function myds = DICOMDatastore(location)
            % The class constructor to set properties of the datastore.
            myds.Files = matlab.io.datastore.FileSet(location, ...
                "IncludeSubfolders", true);
            populateFoldersFromLocation(myds,location);
            reset(myds);
        end

        function tf = hasdata(myds)
            %HASDATA   Returns true if more data is available.
            %   Return logical scalar indicating availability of data.
            %   This method should be called before calling read. This
            %   is an abstract method and must be implemented by the
            %   subclasses. hasdata is used in conjunction with read to
            %   read all the data within the datastore.
            tf = hasNextFile(myds.Files);
        end

        function [data, info] = read(myds)
            %READ   Read data and information about the extracted data.
            %   Return the data extracted from the datastore in the
            %   appropriate form for this datastore. Also return
            %   information about where the data was extracted from in
            %   the datastore. Both the outputs are required to be
            %   returned from the read method and can be of any type.
            %   info is recommended to be a struct with information
            %   about the chunk of data read. data represents the
            %   underlying class of tall, if tall is created on top of
            %   this datastore. This is an abstract method and must be
            %   implemented by the subclasses.

            % In this example, the read method reads data from the
            % datastore using a custom reader function, MyFileReader,
            % which takes the resolved filenames as input.
            if ~hasdata(myds)
                error("No more data to read.\nUse reset method to " ...
                    + "reset the datastore to the start of the data. Before " ...
                    + "calling the read method, check if data is available " ...
                    + "to read by using the hasdata method.");
            end

            file = nextfile(myds.Files);
            try
                data = dicomread(file.Filename);
            catch ME
                error("%s has failed", file.FileName);
            end

            info.FileSize = size(data);
            info.Filename = file.Filename;
        end

        function reset(myds)
            %RESET   Reset to the start of the data.
            %   Reset the datastore to the state where no data has been
            %   read from it. This is an abstract method and must be
            %   implemented by the subclasses.

            % In this example, the datastore is reset to point to the
            % first file (and first partition) in the datastore.
            reset(myds.Files);
        end

        function frac = progress(myds)
            %PROGRESS   Percentage of consumed data between 0.0 and 1.0.
            %   Return fraction between 0.0 and 1.0 indicating progress as a
            %   double. The provided example implementation returns the
            %   ratio of the index of the current file from FileSet
            %   to the number of files in FileSet. A simpler
            %   implementation can be used here that returns a 1.0 when all
            %   the data has been read from the datastore, and 0.0
            %   otherwise.
            %
            %   See also matlab.io.Datastore, read, hasdata, reset, readall,
```

```
            %   preview.
            frac = progress(myds.Files);
        end
    end

    methods(Access = protected)
        function dsCopy = copyElement(myds)
            %COPYELEMENT   Create a deep copy of the datastore
            %   Create a deep copy of the datastore. We need to call
            %   copy on the datastore's property FileSet because it is
            %   a handle object. Creating a deep copy allows methods
            %   such as readall and preview, which call the copy method,
            %   to remain stateless.
            dsCopy = copyElement@matlab.mixin.Copyable(myds);
            dsCopy.Files = copy(myds.Files);
        end

        function tf = write(myds, data, writeInfo, outFmt, varargin)
            if outFmt == "dcm"
                dicomwrite(data, writeInfo.SuggestedOutputName, varargin{:});
            else
                write@matlab.io.datastore.FileWritable(myds, data, ...
                    writeInfo, outFmt, varargin{:});
            end
            tf = true;
        end

        function files = getFiles(myds)
            files = myds.Files.FileInfo.Filename;
        end
    end
end
```

Using the DICOMDatastore Class

After you implement the `DICOMDatastore` class, you can use the constructor to create a new `DICOMDatastore` object that references the location of a set of DICOM files. For example, if you have DICOM files in the folder `C:\Data\DICOM\series-000001\`:

```
ds = DICOMDatastore("C:\Data\DICOM\series-000001")
```

```
ds = 

  DICOMDatastore with properties:

                     Files: [1×1 matlab.io.datastore.FileSet]
    SupportedOutputFormats: ["png"    "jpg"    "jpeg"    "tif"    "tiff"    "dcm"]
       DefaultOutputFormat: "dcm"
                   Folders: {'C:\Data\DICOM\series-000001'}
```

Class users of `DICOMDatastore` have access to these public methods:

```
methods(ds)
```

```
Methods for class DICOMDatastore:

DICOMDatastore   copy      isPartitionable  preview    read     reset      writeall
combine          hasdata   isShuffleable    progress   readall  transform

Methods of DICOMDatastore inherited from handle.
```

In particular, with support for `writeall`, you can write the files to a new location:

```
writeall(ds,"C:\Data2\DICOM\")
```

This command creates copies of the datastore files in the folder `C:\Data2\DICOM\series-000001`.

For general information on authoring classes in MATLAB, see “Classes”.

See Also

`matlab.io.Datastore` | `matlab.io.datastore.FileWritable` | `matlab.io.datastore.FoldersPropertyProvider`

More About

- “Develop Custom Datastore” on page 12-107
- “Testing Guidelines for Custom Datastores” on page 12-116

Set Up Datastore for Processing on Different Machines or Clusters

You can create and save a datastore on a platform that loads and works seamlessly on a different platform by setting up the 'AlternateFileSystemRoots' property of the datastore. Use this property when:

- You create a datastore on a local machine, and need to access and process the data on another machine (possibly running a different operating system).
- You process your datastore with parallel and distributed computing involving different platforms, cloud or cluster machines.

This example demonstrates the use of the 'AlternateFileSystemRoots' property for TabularTextDatastore. However, you can use the same syntax for any of these datastores: SpreadsheetDatastore, ImageDatastore, ParquetDatastore, FileDatastore, KeyValueDatastore, and TallDatastore. To use the 'AlternateFileSystemRoots' functionality for custom datastores, see matlab.io.datastore.DsFileSet and “Develop Custom Datastore” on page 12-107.

Save Datastore and Load on Different File System Platform

Create a datastore on one file system that loads and works seamlessly on a different machine (possibly of a different operating system). For example, create a datastore on a Windows machine, save it, and then load it on a Linux machine.

First, before you create and save the datastore, identify the root paths for your data on the different platforms. The root paths will differ based on the machine or file system. For instance, if you have data on your local machine and a copy of the data on a cluster, then get the root paths for accessing the data:

- "Z:\DataSet" for your local Windows machine.
- "/nfs-bldg001/DataSet" for your Linux cluster.

Then, associate these root paths by using the 'AlternateFileSystemRoots' parameter of the datastore.

```
altRoots = ["Z:\DataSet","/nfs-bldg001/DataSet"];
ds = tabularTextDatastore('Z:\DataSet','AlternateFileSystemRoots',altRoots);
```

Examine the Files property of datastore. In this instance, the Files property contains the location of your data as accessed by your Windows machine.

```
ds.Files
```

```
ans =

  5×1 cell array

    {'Z:\DataSet\datafile01.csv'}
    {'Z:\DataSet\datafile02.csv'}
    {'Z:\DataSet\datafile03.csv'}
    {'Z:\DataSet\datafile04.csv'}
    {'Z:\DataSet\datafile05.csv'}
```

Save the datastore.

```
save ds_saved_on_Windows.mat ds
```

Load the datastore on a Linux platform and examine the `Files` property. Since the root path `'Z:\DataSet'` is not accessible on the Linux cluster, at load time, the datastore function automatically updates the root paths based on the values specified in the `'AlternateFileSystemRoots'` parameter. The `Files` property of the datastore now contains the updated root paths for your data on the Linux cluster.

```
load ds_saved_on_Windows.mat
ds.Files
```

```
ans =

  5×1 cell array

    {'/nfs-bldg001/DataSet/datafile01.csv'}
    {'/nfs-bldg001/DataSet/datafile02.csv'}
    {'/nfs-bldg001/DataSet/datafile03.csv'}
    {'/nfs-bldg001/DataSet/datafile04.csv'}
    {'/nfs-bldg001/DataSet/datafile05.csv'}
```

You can now process and analyze this datastore on your Linux machine.

Process Datastore Using Parallel and Distributed Computing

To process your datastore with parallel and distributed computing that involves different platforms, cloud or cluster machines, you must predefine the `'AlternateFileSystemRoots'` parameter. This example demonstrates how to create a datastore on your local machine, analyze a small portion of the data, and then use Parallel Computing Toolbox and MATLAB Parallel Server to scale up the analysis to the entire dataset.

Create a datastore and assign a value to the `'AlternateFileSystemRoots'` property. To set the value for the `'AlternateFileSystemRoots'` property, identify the root paths for your data on the different platforms. The root paths differ based on the machine or file system. For example, identify the root paths for data access from your machine and your cluster:

- `"Z:\DataSet"` from your local Windows Machine.
- `"/nfs-bldg001/DataSet"` from the MATLAB Parallel Server Linux Cluster.

Then, associate these root paths using the `AlternateFileSystemRoots` property.

```
altRoots = ["Z:\DataSet","/nfs-bldg001/DataSet"];
ds = tabularTextDatastore('Z:\DataSet','AlternateFileSystemRoots',altRoots);
```

Analyze a small portion of the data on your local machine. For instance, get a partitioned subset of the data, clean the data by removing any missing entries, and examine a plot of the variables.

```
tt = tall(partition(ds,100,1));
summary(tt);
% analyze your data
tt = rmmissing(tt);
plot(tt.MyVar1,tt.MyVar2)
```

Scale up your analysis to the entire dataset by using MATLAB Parallel Server cluster (Linux cluster). For instance, start a worker pool using the cluster profile, and then perform analysis on the entire dataset by using parallel and distributed computing capabilities.

```
parpool('MyMjsProfile')
tt = tall(ds);
summary(tt);
% analyze your data
tt = rmmissing(tt);
plot(tt.MyVar1,tt.MyVar2)
```

See Also

FileDatastore | ImageDatastore | KeyValueDatastore | SpreadsheetDatastore | TabularTextDatastore | TallDatastore | datastore

More About

- “Getting Started with Datastore” on page 12-84
- “Work with Remote Data” on page 12-91

Apache Parquet Data Type Mappings

MATLAB represents column-oriented data with tables and timetables. Each variable in a table or timetable can have a different data type and any number of columns. Column vectors are the most common shape of table and timetable variables.

The Apache™ Parquet file format is used for column-oriented heterogeneous data. Similar to MATLAB tables and timetables, each of the columns in a Parquet file can have different data types.

Despite their similarity, the permitted data types in MATLAB tables and timetables do not always map perfectly to the permitted data types in Parquet files. In some cases, it is necessary for MATLAB to perform data type conversions to retain information in the data (such as missing values). This conversion can sometimes result in a loss of precision in the data.

In general, MATLAB tables and timetables have these behaviors when they are converted to Parquet files:

- Table properties set on the original table are not saved.
- Table row names or timetable row times are converted into a new table variable before being written.
- When reading a variable name from a Parquet file, invalid table variable names are converted to valid table variable names.

The following tables summarize the representable data types in MATLAB tables and timetables, as well as how those variables are represented in Parquet files. These data type mappings can go in both directions (MATLAB → Parquet and Parquet → MATLAB), unless otherwise noted. Parquet files use a small number of primitive (or *physical*) data types. The *logical* types extend the physical types by specifying how they should be interpreted. Parquet data types not covered here are not supported for reading from or writing to Parquet files (JSON, BSON, binary, and so on).

Numeric Data Types

MATLAB Table or Timetable Variable Type	Apache Parquet Data Type		Notes
	Physical Type	Logical Type	
`double`	`DOUBLE`	`NONE`	MATLAB converts any missing floating-point numbers in a Parquet file into `NaN` values.
`single`	`FLOAT`	`NONE`	
`int8`	`INT32`	`INT_8`	When reading a Parquet file, if an array with integral type contains missing values, then the array is converted into the MATLAB `double` data type instead of an integer data type. The missing values are set to `NaN`. For 64-bit integers, this conversion can result in truncation of values that are larger in magnitude than `flintmax`.
`uint8`		`UINT_8`	
`int16`		`INT_16`	
`uint16`		`UINT_16`	
`int32`		`NONE`	
`uint32`		`UINT_32`	
`int64`	`INT64`	`NONE`	
`uint64`		`UINT_64`	

MATLAB Table or Timetable Variable Type	Apache Parquet Data Type		Notes
	Physical Type	Logical Type	
logical	BOOLEAN	NONE	When reading a Parquet file, if an array with BOOLEAN type contains missing values, then the array is converted into the MATLAB double data type instead of the logical data type. The missing values are set to NaN.

Text Data Types

MATLAB Table or Timetable Variable Type	Apache Parquet Data Type		Notes
	Physical Type	Logical Type	
categorical	BYTE_ARRAY	UTF8	Categorical arrays are converted into string arrays when written to Parquet files. Any <undefined> categorical values are converted to <missing> strings before being written.
string			string, char, and cellstr are all mapped to the same Parquet data type, and that data type is always read into MATLAB as a string array.
char			
cellstr (cell array of character vectors)			

Date and Time Data Types

MATLAB Table or Timetable Variable Type	Apache Parquet Data Type		Notes
	Physical Type	Logical Type	
datetime	INT32	DATE	MATLAB datetime arrays written to a Parquet file use TIMESTAMP_MICROS format and have precision truncated to 1 microsecond. Display format settings are not saved.
	INT64	TIMESTAMP_MILLIS	
		TIMESTAMP_MICROS	
duration	INT32	TIME_MILLIS	MATLAB duration arrays written to a

MATLAB Table or Timetable Variable Type	Apache Parquet Data Type		Notes
	Physical Type	Logical Type	
	INT64	TIME_MICROS	Parquet file use TIME_MICROS format and have precision truncated to 1 microsecond. Display format settings are not saved.

See Also
parquetread | parquetwrite | write

Tall Arrays for Out-of-Memory Data

Tall arrays are used to work with out-of-memory data that is backed by a `datastore`. Datastores enable you to work with large data sets in small blocks that individually fit in memory, instead of loading the entire data set into memory at once. Tall arrays extend this capability to enable you to work with out-of-memory data using common functions.

What is a Tall Array?

Since the data is not loaded into memory all at once, tall arrays can be arbitrarily large in the first dimension (that is, they can have any number of rows). Instead of writing special code that takes into account the huge size of the data, such as with techniques like MapReduce, tall arrays let you work with large data sets in an intuitive manner that is similar to the way you would work with in-memory MATLAB arrays. Many core operators and functions work the same with tall arrays as they do with in-memory arrays. MATLAB works with small blocks of the data at a time, handling all of the data chunking and processing in the background, so that common expressions, such as `A+B`, work with big data sets.

Benefits of Tall Arrays

Unlike in-memory arrays, tall arrays typically remain unevaluated until you request that the calculations be performed using the `gather` function. This *deferred evaluation* allows you to work quickly with large data sets. When you eventually request output using `gather`, MATLAB combines the queued calculations where possible and takes the minimum number of passes through the data. The number of passes through the data greatly affects execution time, so it is recommended that you request output only when necessary.

Note Since `gather` returns results as in-memory MATLAB arrays, standard memory considerations apply. MATLAB might run out of memory if the result returned by `gather` is too large.

Creating Tall Tables

Tall tables are like in-memory MATLAB tables, except that they can have any number of rows. To create a tall table from a large data set, you first need to create a datastore for the data. If the datastore `ds` contains tabular data, then `tall(ds)` returns a tall table or tall timetable containing the data. See “Datastore” for more information about creating datastores.

Create a spreadsheet datastore that points to a tabular file of airline flight data. For folders that contain a collection of files, you can specify the entire folder location, or use the wildcard character, `'*.csv'`, to include multiple files with the same file extension in the datastore. Clean the data by treating `'NA'` values as missing data so that `tabularTextDatastore` replaces them with `NaN` values. Also, set the format of a few text variables to `%s` so that `tabularTextDatastore` reads them as cell arrays of character vectors.

```
ds = tabularTextDatastore('airlinesmall.csv');
ds.TreatAsMissing = 'NA';
ds.SelectedFormats{strcmp(ds.SelectedVariableNames,'TailNum')} = '%s';
ds.SelectedFormats{strcmp(ds.SelectedVariableNames,'CancellationCode')} = '%s';
```

Create a tall table from the datastore. When you perform calculations on this tall table, the underlying datastore reads blocks of data and passes them to the tall table to process. Neither the datastore nor the tall table retain any of the underlying data.

```
tt = tall(ds)
```

```
tt =

  M×29 tall table

    Year    Month    DayofMonth    DayOfWeek    DepTime    CRSDepTime    ArrTime    CRSArrTime
    ____    _____    __________    _________    _______    __________    _______    __________

    1987    10       21            3             642        630           735        727
    1987    10       26            1            1021       1020          1124       1116
    1987    10       23            5            2055       2035          2218       2157
    1987    10       23            5            1332       1320          1431       1418
    1987    10       22            4             629        630           746        742
    1987    10       28            3            1446       1343          1547       1448
    1987    10        8            4             928        930          1052       1049
    1987    10       10            6             859        900          1134       1123
    :       :        :             :            :          :             :          :
    :       :        :             :            :          :             :          :
```

The display indicates that the number of rows, `M`, is currently unknown. MATLAB displays some of the rows, and the vertical ellipses `:` indicate that more rows exist in the tall table that are not currently being displayed.

Creating Tall Timetables

If the data you are working with has a time associated with each row of data, then you can use a tall timetable to work on the data. For information on creating tall timetables, see Extended Capabilities (`timetable`).

In this case, the tall table `tt` has times associated with each row, but they are broken down into several table variables such as `Year`, `Month`, `DayofMonth`, and so on. Combine all of these pieces of datetime information into a single new tall datetime variable `Dates`, which is based on the departure times `DepTime`. Then, create a tall timetable using `Dates` as the row times. Since `Dates` is the only datetime variable in the table, the `table2timetable` function automatically uses it for the row times.

```
hrs = (tt.DepTime - mod(tt.DepTime,100))/100;
mins = mod(tt.DepTime,100);
tt.Dates = datetime(tt.Year, tt.Month, tt.DayofMonth, hrs, mins, 0);
tt(:,1:8) = [];
TT = table2timetable(tt)
```

```
TT =

  M×21 tall timetable

           Dates           UniqueCarrier    FlightNum    TailNum    ActualElapsedTime    CRSEla
    ____________________    _____________    _________    _______    _________________    ______

    21-Oct-1987 06:42:00    'PS'             1503         'NA'        53                   57
    26-Oct-1987 10:21:00    'PS'             1550         'NA'        63                   56
    23-Oct-1987 20:55:00    'PS'             1589         'NA'        83                   82
    23-Oct-1987 13:32:00    'PS'             1655         'NA'        59                   58
    22-Oct-1987 06:29:00    'PS'             1702         'NA'        77                   72
    28-Oct-1987 14:46:00    'PS'             1729         'NA'        61                   65
    08-Oct-1987 09:28:00    'PS'             1763         'NA'        84                   79
```

```
    10-Oct-1987 08:59:00    'PS'             1800        'NA'      155                   143
    :                       :                :           :         :                     :
    :                       :
```

Creating Tall Arrays

When you extract a variable from a tall table or tall timetable, the result is a tall array of the appropriate underlying data type. A tall array can be a numeric, logical, datetime, duration, calendar duration, categorical, string, or cell array. Also, you can convert an in-memory array `A` into a tall array with `tA = tall(A)`. The in-memory array `A` must have one of the supported data types.

Extract the arrival delay `ArrDelay` from the tall timetable `TT`. This creates a new tall array variable with underlying data type double.

```
a = TT.ArrDelay
```

```
a =

  M×1 tall double column vector

     8
     8
    21
    13
     4
    59
     3
    11
    :
    :
```

The `classUnderlying` and `isaUnderlying` functions are useful to determine the underlying data type of a tall array.

Deferred Evaluation

One important aspect of tall arrays is that as you work with them, most operations are not performed immediately. These operations appear to execute quickly, because the actual computation is deferred until you specifically request that the calculations be performed. You can trigger evaluation of a tall array with either the `gather` function (to bring the result into memory) or the `write` function (to write the result to disk). This deferred evaluation is important because even a simple command like `size(X)` executed on a tall array with a billion rows is not a quick calculation.

As you work with tall arrays, MATLAB keeps track of all of the operations to be carried out. This information is then used to optimize the number of passes through the data that will be required when you request output with the `gather` function. Thus, it is normal to work with unevaluated tall arrays and request output only when you require it. For more information, see “Deferred Evaluation of Tall Arrays” on page 12-142.

Calculate the mean and standard deviation of the arrival delay. Use these values to construct the upper and lower thresholds for delays that are within one standard deviation of the mean. Notice that the result of each operation indicates that the array has not been calculated yet.

```
m = mean(a,'omitnan')
```

```
m =

  tall double

    ?

Preview deferred. Learn more.
```

```
s = std(a,'omitnan')
```

```
s =

  tall

    ?

Preview deferred. Learn more.
```

```
one_sigma_bounds = [m-s m m+s]
```

```
one_sigma_bounds =

  M×N×... tall array

    ?    ?    ?    ...
    ?    ?    ?    ...
    ?    ?    ?    ...
    :    :    :
    :    :    :

Preview deferred. Learn more.
```

Evaluation with gather

The benefit of delayed evaluation is that when the time comes for MATLAB to perform the calculations, it is often possible to combine the operations in such a way that the number of passes through the data is minimized. So even if you perform many operations, MATLAB only makes extra passes through the data when absolutely necessary.

The `gather` function forces evaluation of all queued operations and brings the resulting output into memory. For this reason, you can think of `gather` as a bridge between tall arrays and in-memory arrays. For example, you cannot control `if` or `while` loops using a tall logical array, but once the array is evaluated with `gather` it becomes an in-memory logical array that you can use in these contexts.

Since `gather` returns the entire result in MATLAB, you should make sure that the result will fit in memory.

Use `gather` to calculate `one_sigma_bounds` and bring the result into memory. In this case, `one_sigma_bounds` requires several operations to calculate, but MATLAB combines the operations into one pass through the data. Since the data in this example is small, `gather` executes quickly. However, the elimination of passes through the data becomes more valuable as the size of your data increases.

```
sig1 = gather(one_sigma_bounds)
```

```
Evaluating tall expression using the Local MATLAB Session:
- Pass 1 of 1: Completed in 1.5 sec
```

```
Evaluation completed in 1.8 sec

sig1 =

  -23.4572    7.1201   37.6975
```

You can specify multiple inputs and outputs to `gather` if you want to evaluate several tall arrays at once. This technique is faster than calling `gather` multiple times. For example, calculate the minimum and maximum arrival delay. Computed separately, each value requires a pass through the data to calculate for a total of two passes. However, computing both values simultaneously requires only one pass through the data.

```
[max_delay, min_delay] = gather(max(a),min(a))
```

```
Evaluating tall expression using the Local MATLAB Session:
- Pass 1 of 1: Completed in 1.1 sec
Evaluation completed in 1.1 sec

max_delay =

        1014


min_delay =

   -64
```

These results indicate that on average, most flights arrive about 7 minutes late. But it is within one standard deviation for a flight to be up to 37 minutes late or 23 minutes early. The quickest flight in the data set arrived about an hour early, and the latest flight was delayed by many hours.

Saving, Loading, and Checkpointing Tall Arrays

The `save` function saves the *state* of a tall array, but does not copy any of the data. The resulting `.mat` file is typically small. However, the original data files must be available in the same location in order to subsequently use `load`.

The `write` function makes a copy of the data and saves the copy as a collection of files, which can consume a large amount of disk space. `write` executes all pending operations on the tall array to calculate the values prior to writing. Once `write` copies the data, it is independent of the original raw data. Therefore, you can recreate the tall array from the written files even if the original raw data is no longer available.

You can recreate the tall array from the written files by creating a new datastore that points to the location where the files were written. This functionality enables you to create *checkpoints* or *snapshots* of tall array data. Creating a checkpoint is a good way to save the results of preprocessing your data, so that the data is in a form that is more efficient to load.

If you have a tall array `TA`, then you can write it to the folder `location` with the command:

```
write(location,TA);
```

Later, to reconstruct `TA` from the written files, use the commands:

```
ds = datastore(location);
TA = tall(ds);
```

Additionally, you can use the `write` function to trigger evaluation of a tall array and write the results to disk. This use of `write` is similar to `gather`, however, `write` does not bring any results into memory.

Supporting Functions

Most core functions work the same way with tall arrays as they do with in-memory arrays. However, in some cases the way that a function works with tall arrays is special or has limitations. You can tell whether a function supports tall arrays, and if it has any limitations, by looking at the bottom of the reference page for the function in the **Extended Capabilities** section (for an example, see `filloutliers`).

For a filtered list of all MATLAB functions that support tall arrays, see Function List (Tall Arrays).

Tall arrays also are supported by several toolboxes, enabling you to do things like write machine learning algorithms, deploy standalone apps, and run calculations in parallel or on a cluster. For more information, see “Extend Tall Arrays with Other Products” on page 12-175.

See Also

`datastore` | `gather` | `mapreducer` | `table` | `tall`

More About

- “Index and View Tall Array Elements” on page 12-147
- “Visualization of Tall Arrays” on page 12-161

Deferred Evaluation of Tall Arrays

One of the differences between tall arrays and in-memory MATLAB arrays is that tall arrays typically remain *unevaluated* until you request that calculations be performed. (The exceptions to this rule include plotting functions like `plot` and `histogram` and some statistical fitting functions like `fitlm`, which automatically evaluate tall array inputs.) While a tall array is in an unevaluated state, MATLAB might not know its size, its data type, or the specific values it contains. However, you can still use unevaluated arrays in your calculations as if the values were known. This allows you to work quickly with large data sets instead of waiting for each command to execute. For this reason, it is recommended that you use `gather` only when you require output.

MATLAB keeps track of all the operations you perform on unevaluated tall arrays as you enter them. When you eventually call `gather` to evaluate the queued operations, MATLAB uses the history of unevaluated commands to optimize the calculation by minimizing the number of passes through the data. Used properly, this optimization can save huge amounts of execution time by eliminating unnecessary passes through large data sets.

Display of Unevaluated Tall Arrays

The display of unevaluated tall arrays varies depending on how much MATLAB knows about the array and its values. There are three pieces of information reflected in the display:

- **Array size** — Unknown dimension sizes are represented by the variables `M` or `N` in the display. If no dimension sizes are known, then the size appears as `MxNx....`.
- **Array data type** — If the array has an unknown underlying data type, then its type appears as `tall array`. If the type is known, it is listed as, for example, `tall double array`.
- **Array values** — If the array values are unknown, then they appear as `?`. Known values are displayed.

MATLAB might know all, some, or none of these pieces of information about a given tall array, depending on the nature of the calculation.

For example, if the array has a known data type but unknown size and values, then the unevaluated tall array might look like this:

```
M×N×... tall double array

    ?    ?    ?    ...
    ?    ?    ?    ...
    ?    ?    ?    ...
    :    :    :
    :    :    :
```

If the type and relative size are known, then the display could be:

```
 1×N tall char array

    ?    ?    ?   ...
```

If some of the data is known, then MATLAB displays the known values:

```
  100×3 tall double matrix

    0.8147    0.1622    0.6443
```

```
    0.9058    0.7943    0.3786
    0.1270    0.3112    0.8116
    0.9134    0.5285    0.5328
    0.6324    0.1656    0.3507
    0.0975    0.6020    0.9390
    0.2785    0.2630    0.8759
    0.5469    0.6541    0.5502
      :         :         :
      :         :         :
```

Evaluation with gather

The `gather` function is used to evaluate tall arrays. `gather` accepts tall arrays as inputs and returns in-memory arrays as outputs. For this reason, you can think of this function as a bridge between tall arrays and in-memory arrays. For example, you cannot control `if` or `while` loop statements using a tall logical array, but once the array is evaluated with `gather` it becomes an in-memory logical value that you can use in these contexts.

`gather` performs all queued operations on a tall array and returns the *entire* result in memory. Since `gather` returns results as in-memory MATLAB arrays, standard memory considerations apply. MATLAB might run out of memory if the result returned by `gather` is too large.

Most of the time you can use `gather` to see the entire result of a calculation, particularly if the calculation includes a reduction operation such as `sum` or `mean`. However, if the result is too large to fit in memory, then you can use `gather(head(X))` or `gather(tail(X))` to perform the calculation and look at only the first or last few rows of the result.

Resolve Errors with gather

If you enter an erroneous command and `gather` fails to evaluate a tall array variable, then you must delete the variable from your workspace and recreate the tall array using *only* valid commands. This is because MATLAB keeps track of all the operations you perform on unevaluated tall arrays as you enter them. The only way to make MATLAB “forget” about an erroneous statement is to reconstruct the tall array from scratch.

Example: Calculate Size of Tall Array

This example shows what an unevaluated tall array looks like, and how to evaluate the array.

Create a datastore for the data set `airlinesmall.csv`. Convert the datastore into a tall table and then calculate the size.

```
varnames = {'ArrDelay', 'DepDelay', 'Origin', 'Dest'};
ds = tabularTextDatastore('airlinesmall.csv', 'TreatAsMissing', 'NA', ...
'SelectedVariableNames', varnames);
tt = tall(ds)
```

```
tt =

  M×4 tall table

    ArrDelay    DepDelay    Origin    Dest 
    ________    ________    ______    _____

        8          12       'LAX'     'SJC'
```

```
         8             1          'SJC'      'BUR'
        21            20          'SAN'      'SMF'
        13            12          'BUR'      'SJC'
         4            -1          'SMF'      'LAX'
        59            63          'LAX'      'SJC'
         3            -2          'SAN'      'SFO'
        11            -1          'SEA'      'LAX'
        :             :             :          :
        :             :             :          :
```

```
s = size(tt)
```

```
s =

  1×2 tall double row vector

    ?    ?

Preview deferred. Learn more.
```

Calculating the size of a tall array returns a small answer (a 1-by-2 vector), but the display indicates that an entire pass through the data is still required to calculate the size of `tt`.

Use the `gather` function to fully evaluate the tall array and bring the results into memory. As the command executes, there is a dynamic progress display in the command window that is particularly helpful with long calculations.

Note Always ensure that the result returned by `gather` will be able to fit in memory. If you use `gather` directly on a tall array without reducing its size using a function such as `mean`, then MATLAB might run out of memory.

```
tableSize = gather(s)
```

```
Evaluating tall expression using the Local MATLAB Session:
- Pass 1 of 1: Completed in 0.42 sec
Evaluation completed in 0.48 sec

tableSize =

      123523           4
```

Example: Multi-pass Calculations with Tall Arrays

This example shows how several calculations can be combined to minimize the total number of passes through the data.

Create a datastore for the data set `airlinesmall.csv`. Convert the datastore into a tall table.

```
varnames = {'ArrDelay', 'DepDelay', 'Origin', 'Dest'};
ds = tabularTextDatastore('airlinesmall.csv', 'TreatAsMissing', 'NA', ...
'SelectedVariableNames', varnames);
tt = tall(ds)
```

```
tt =

  M×4 tall table
```

```
    ArrDelay    DepDelay    Origin    Dest 
    ________    ________    ______    _____

       8           12       'LAX'     'SJC'
       8            1       'SJC'     'BUR'
      21           20       'SAN'     'SMF'
      13           12       'BUR'     'SJC'
       4           -1       'SMF'     'LAX'
      59           63       'LAX'     'SJC'
       3           -2       'SAN'     'SFO'
      11           -1       'SEA'     'LAX'
      :            :          :         :
      :            :          :         :
```

Subtract the mean value of `DepDelay` from `ArrDelay` to create a new variable `AdjArrDelay`. Then calculate the mean value of `AdjArrDelay` and subtract this mean value from `AdjArrDelay`. If these calculations were all evaluated separately, then MATLAB would require four passes through the data.

```
AdjArrDelay = tt.ArrDelay - mean(tt.DepDelay,'omitnan');
AdjArrDelay = AdjArrDelay - mean(AdjArrDelay,'omitnan')
```

```
AdjArrDelay =

  M×1 tall double column vector

    ?
    ?
    ?
    :
    :

Preview deferred. Learn more.
```

Evaluate `AdjArrDelay` and view the first few rows. Because some calculations can be combined, only three passes through the data are required.

```
gather(head(AdjArrDelay))
```

```
Evaluating tall expression using the Local MATLAB Session:
- Pass 1 of 3: Completed in 0.4 sec
- Pass 2 of 3: Completed in 0.39 sec
- Pass 3 of 3: Completed in 0.23 sec
Evaluation completed in 1.2 sec

ans =

    0.8799
    0.8799
   13.8799
    5.8799
   -3.1201
   51.8799
   -4.1201
    3.8799
```

Summary of Behavior and Recommendations

1. Tall arrays remain unevaluated until you request output using `gather`.
2. Use `gather` in most cases to evaluate tall array calculations. If you believe the result of the calculations might not fit in memory, then use `gather(head(X))` or `gather(tail(X))` instead.
3. Work primarily with unevaluated tall arrays and request output only when necessary. The more queued calculations there are that are unevaluated, the more optimization MATLAB can do to minimize the number of passes through the data.
4. If you enter an erroneous tall array command and `gather` fails to evaluate a tall array variable, then you must delete the variable from your workspace and recreate the tall array using *only* valid commands.

See Also

`gather` | `write`

More About

- "Tall Arrays for Out-of-Memory Data" on page 12-136

Index and View Tall Array Elements

Tall arrays are too large to fit in memory, so it is common to view subsets of the data rather than the entire array. This page shows techniques to extract and view portions of a tall array.

Extract Top Rows of Array

Use the `head` function to extract the first rows in a tall array. `head` does not force evaluation of the array, so you must use `gather` to view the result.

```
tt = tall(table(randn(1000,1),randn(1000,1),randn(1000,1)))
```

```
tt =

  1,000×3 tall table

      Var1        Var2        Var3  
    ________    ________    ________

     0.53767      0.6737     0.29617
      1.8339    -0.66911      1.2008
     -2.2588    -0.40032      1.0902
     0.86217     -0.6718     -0.3587
     0.31877     0.57563    -0.12993
     -1.3077    -0.77809     0.73374
    -0.43359     -1.0636     0.12033
     0.34262     0.55298      1.1363
        :           :           :
        :           :           :
```

```
t_head = gather(head(tt))
```

```
t_head =

  8×3 table

      Var1        Var2        Var3  
    ________    ________    ________

     0.53767      0.6737     0.29617
      1.8339    -0.66911      1.2008
     -2.2588    -0.40032      1.0902
     0.86217     -0.6718     -0.3587
     0.31877     0.57563    -0.12993
     -1.3077    -0.77809     0.73374
    -0.43359     -1.0636     0.12033
     0.34262     0.55298      1.1363
```

Extract Bottom Rows of Array

Similarly, you can use the `tail` function to extract the bottom rows in a tall array.

```
t_tail = gather(tail(tt))
```

```
t_tail =
```

```
8×3 table

      Var1        Var2        Var3  
    ________    ________    ________

     0.64776     0.47349    -0.27077
    -0.31763      1.3656     0.43966
       1.769     -1.6378    -0.50614
      1.5106      2.0237    -0.18435
     0.16401     0.77779       0.402
    -0.28276     -0.5489     0.53923
      1.1522    -0.12601    -0.73359
     -1.1465     0.29958    -0.26837
```

Indexing Tall Arrays

All tall arrays support parentheses indexing. When you index a tall array using parentheses, such as `T(A)` or `T(A,B)`, the result is a new tall array containing only the specified rows and columns (or variables).

Like most other operations on tall arrays, indexing expressions are not evaluated immediately. You must use `gather` to evaluate the indexing operation. For more information, see "Deferred Evaluation of Tall Arrays" on page 12-142.

You can perform these types of indexing in the first dimension of a tall array:

- `B = A(:,…)`, where `:` selects all rows in `A`.
- `B = A(idx,…)`, where `idx` is a tall numeric column vector or non-tall numeric vector.
- `B = A(L,…)`, where `L` is a tall or non-tall logical array of the same height as `A`. For example, you can use relational operators, such as `tt(tt.Var1 < 10,:)`. When you index a tall array with a tall logical array, there are a few requirements. Each of the tall arrays:
  - Must be the same size in the first dimension.
  - Must be derived from a single tall array.
  - Must not have been indexed differently in the first dimension.
- `B = A(P:D:Q,…)` or `B = A(P:Q,…)`, where `P:D:Q` and `P:Q` are valid `colon` indexing expressions.
  - `head(tt,k)` provides a shortcut for `tt(1:k,:)`.
  - `tail(tt,k)` provides a shortcut for `tt(end-k:end,:)`.

Additionally, the number of subscripts you must specify depends on how many dimensions the array has:

- For tall column vectors, you can specify a single subscript such as `t(1:10)`.
- For tall row vectors, tall tables, and tall timetables, you must specify two subscripts.
- For tall arrays with two or more dimensions, you must specify two or more subscripts. For example, if the array has three dimensions, you can use an expression such as `tA(1:10,:,:)` or `tA(1:10,:)`, but not linear indexing expressions such as `tA(1:10)` or `tA(:)`.

Tip The `find` function locates nonzero elements in tall column vectors, and can be useful to generate a vector of indices for elements that meet particular conditions. For example, `k = find(X<0)` returns the linear indices for all negative elements in `X`.

For example, use parentheses indexing to retrieve the first ten rows of `tt`.

```
tt(1:10,:)
```

```
ans =

  10×3 tall table

      Var1        Var2        Var3
    ________    ________    ________

     0.53767      0.6737     0.29617
      1.8339    -0.66911      1.2008
     -2.2588    -0.40032      1.0902
     0.86217     -0.6718     -0.3587
     0.31877     0.57563    -0.12993
     -1.3077    -0.77809     0.73374
    -0.43359     -1.0636     0.12033
     0.34262     0.55298      1.1363
       :           :           :
       :           :           :
```

Retrieve the last 5 values of the table variable `Var1`.

```
tt(end-5:end,'Var1')
```

```
ans =

  6×1 tall table

      Var1
    ________

       1.769
      1.5106
     0.16401
    -0.28276
      1.1522
     -1.1465
```

Retrieve every 100th row from the tall table.

```
tt(1:100:end,:)
```

```
ans =

  10×3 tall table

      Var1          Var2          Var3
    _________    _________    ________

      0.53767       0.6737      0.29617
      0.84038    -0.041663     -0.52093
      0.18323       1.3419     0.052993
```

```
     0.079934     -0.40492     -1.6163
      0.26965      -1.5144     0.98399
    -0.079893      -1.6848    -0.91182
      0.47586      -2.1746      1.1754
       1.9085     -0.79383     0.18343
         :            :           :
         :            :           :
```

Extract Tall Table Variables

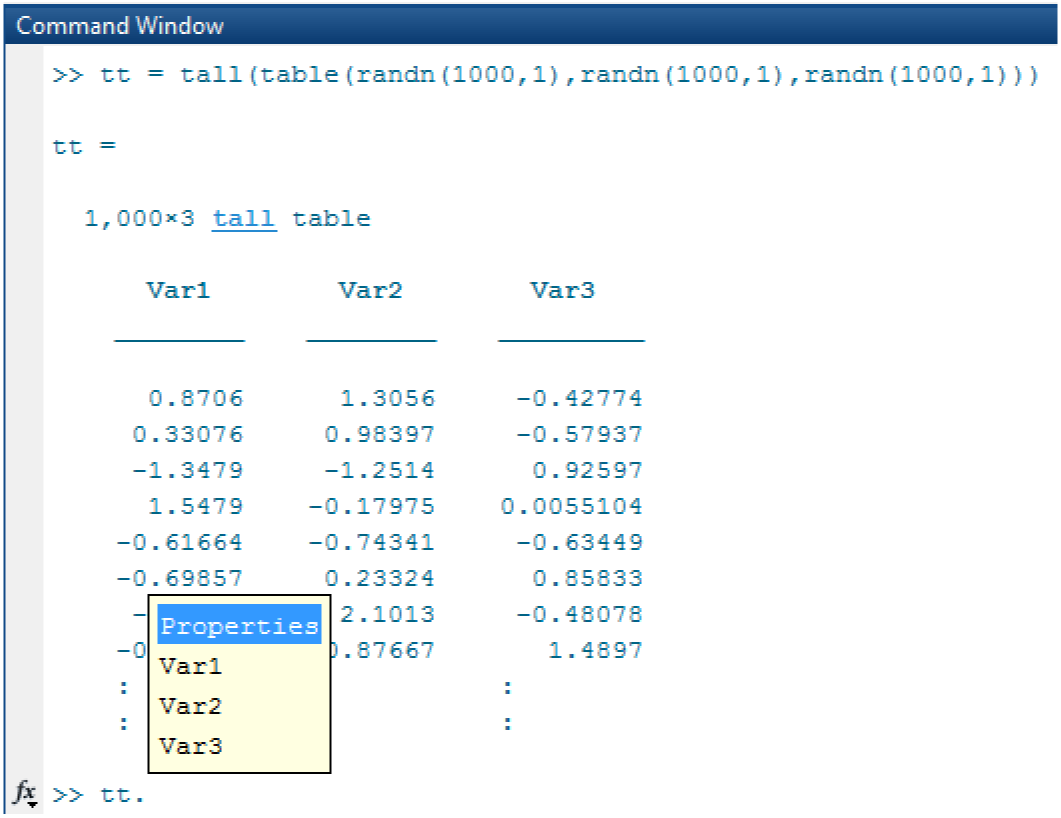
The variables in a tall table or tall timetable are each tall arrays of different underlying data types. Standard indexing methods of tables and timetables also apply to tall tables and tall timetables, including the use of `timerange`, `withtol`, and `vartype`.

For example, index a tall table using dot notation `T.VariableName` to retrieve a single variable of data as a tall array.

```
tt.Var1
```

```
ans =

  1,000×1 tall double column vector

    0.5377
    1.8339
   -2.2588
    0.8622
    0.3188
   -1.3077
   -0.4336
    0.3426
      :
      :
```

Use tab completion to look up the variables in a table if you cannot remember a precise variable name. For example, type `tt.` and then press **Tab**. A menu pops up:



You can also perform multiple levels of indexing. For example, extract the first 5 elements in the variable Var2. In this case you must use one of the supported forms of indexing for tall arrays in the parentheses.

```
tt.Var2(1:5)
```

```
ans =

  5×1 tall double column vector

    0.6737
   -0.6691
   -0.4003
   -0.6718
    0.5756
```

See “Access Data in Tables” or “Select Timetable Data by Row Time and Variable Type” for more indexing information.

Concatenation with Tall Arrays

In order to concatenate two or more tall arrays, as in `[A1 A2 A3 …]`, each of the tall arrays must be derived from a single tall array and must not have been indexed differently in the first dimension. Indexing operations include functions such as `vertcat`, `splitapply`, `sort`, `cell2mat`, `synchronize`, `retime`, and so on.

For example, concatenate a few columns from `tt` to create a new tall matrix.

```
[tt.Var1 tt.Var2]

ans =

  1,000×2 tall double matrix

    0.5377    0.6737
    1.8339   -0.6691
   -2.2588   -0.4003
    0.8622   -0.6718
    0.3188    0.5756
   -1.3077   -0.7781
   -0.4336   -1.0636
    0.3426    0.5530
      :         :
      :         :
```

To combine tall arrays with different underlying datastores, it is recommended that you use `write` to write the arrays (or calculation results) to disk, and then create a single new datastore referencing those locations:

```
files = {'folder/path/to/file1','folder/path/to/file2'};
ds = datastore(files);
```

Assignment and Deletion with Tall Arrays

The same subscripting rules apply whether you use indexing to assign or delete elements from a tall array. Deletion is accomplished by assigning one or more elements to the empty matrix, `[]`.

"( )" Assignment

You can assign elements into a tall array using the general syntax `A(m,n,...) = B`. The tall array `A` must exist and have a nonempty second dimension. The first subscript `m` must be either a colon `:` or a tall logical vector. With this syntax, `B` can be:

- Scalar
- A tall array derived from `A(m,…)` where `m` is the same subscript as above. For example, `A(m,1:10)`.
- An empty matrix, `[]` (for deletion)

"." Assignment

For table indexing using the syntax `A.Var1 = B`, the array `B` must be a tall array with the appropriate number of rows. Typically, `B` is derived from existing data in the tall table. `Var1` can be either a new or existing variable in the tall table.

You cannot assign tall arrays as variables in a regular table, even if the table is empty.

Extract Specified Number of Rows in Sorted Order

Sorting all of the data in a tall array can be an expensive calculation. Most often, only a subset of rows at the beginning or end of a tall array is required to answer questions like "What is the first row in this data by year?"

The `topkrows` function returns a specified number of rows in sorted order for this purpose. For example, use `topkrows` to extract the top 12 rows sorted in descending order by the second column.

```
t_top12 = gather(topkrows(tt,12,2))
```

```
Evaluating tall expression using the Local MATLAB Session:
Evaluation completed in 0.067 sec

t_top12 =

  12×3 table

      Var1        Var2        Var3   
    ________    ______    _________

     -1.0322    3.5699      -1.4689
      1.3312    3.4075      0.17694
    -0.27097    3.1585      0.50127
     0.55095    2.9745        1.382
     0.45168    2.9491      -0.8215
     -1.7115    2.7526      -0.3384
    -0.21317    2.7485       1.9033
    -0.43021    2.7335      0.77616
    -0.59003    2.7304      0.67702
     0.47163    2.7292      0.92099
    -0.47615     2.683     -0.26113
     0.72689    2.5383     -0.57588
```

Summarize Tall Array Contents

The `summary` function returns useful information about each variable in a tall table or timetable, such as the minimum and maximum values of numeric variables, and the number of occurrences of each category for categorical variables.

For example, create a tall table for the `outages.csv` data set and display the summary information. This data set contains numeric, datetime, and categorical variables.

```
fmts = {'%C' '%D' '%f' '%f' '%D' '%C'};
ds = tabularTextDatastore('outages.csv','TextscanFormats',fmts);
T = tall(ds);
summary(T)
```

```
Evaluating tall expression using the Local MATLAB Session:
- Pass 1 of 2: Completed in 0.16 sec
- Pass 2 of 2: Completed in 0.19 sec
Evaluation completed in 0.46 sec

Variables:

    Region: 1,468×1 categorical
        Values:

            MidWest        142
            NorthEast      557
            SouthEast      389
            SouthWest       26
            West           354
```

```
    OutageTime: 1,468×1 datetime
        Values:

            Min    2002-02-01 12:18
            Max    2014-01-15 02:41

    Loss: 1,468×1 double
        Values:

            Min           0
            Max           23418
            NumMissing    604

    Customers: 1,468×1 double
        Values:

            Min           0
            Max           5.9689e+06
            NumMissing    328

    RestorationTime: 1,468×1 datetime
        Values:

            Min           2002-02-07 16:50
            Max           2042-09-18 23:31
            NumMissing    29

    Cause: 1,468×1 categorical
        Values:

            attack               294
            earthquake             2
            energy emergency     188
            equipment fault      156
            fire                  25
            severe storm         338
            thunder storm        201
            unknown               24
            wind                  95
            winter storm         145
```

Return Subset of Calculation Results

Many of the examples on this page use `gather` to evaluate expressions and bring the results into memory. However, in these examples it is also trivial that the results fit in memory, since only a few rows are indexed at a time.

In cases where you are unsure if the result of an expression will fit in memory, it is recommended that you use `gather(head(X))` or `gather(tail(X))`. These commands still evaluate all of the queued calculations, but return only a small amount of the result that is guaranteed to fit in memory.

If you are certain that the result of a calculation will not fit in memory, use `write` to evaluate the tall array and write the results to disk instead.

See Also

`gather` | `head` | `table` | `tail` | `tall` | `topkrows`

More About

- "Tall Arrays for Out-of-Memory Data" on page 12-136

Histograms of Tall Arrays

This example shows how to use `histogram` and `histogram2` to analyze and visualize data contained in a tall array.

Create Tall Table

Create a datastore using the `airlinesmall.csv` data set. Treat `'NA'` values as missing data so that they are replaced with `NaN` values. Select a subset of the variables to work with. Convert the datastore into a tall table.

```
varnames = {'ArrDelay', 'DepDelay', 'Year', 'Month'};
ds = tabularTextDatastore('airlinesmall.csv', 'TreatAsMissing', 'NA', ...
    'SelectedVariableNames', varnames);
T = tall(ds)
```

```
T =

  Mx4 tall table

    ArrDelay    DepDelay    Year    Month
    ________    ________    ____    _____

        8          12       1987     10  
        8           1       1987     10  
       21          20       1987     10  
       13          12       1987     10  
        4          -1       1987     10  
       59          63       1987     10  
        3          -2       1987     10  
       11          -1       1987     10  
       :           :         :        :
       :           :         :        :
```

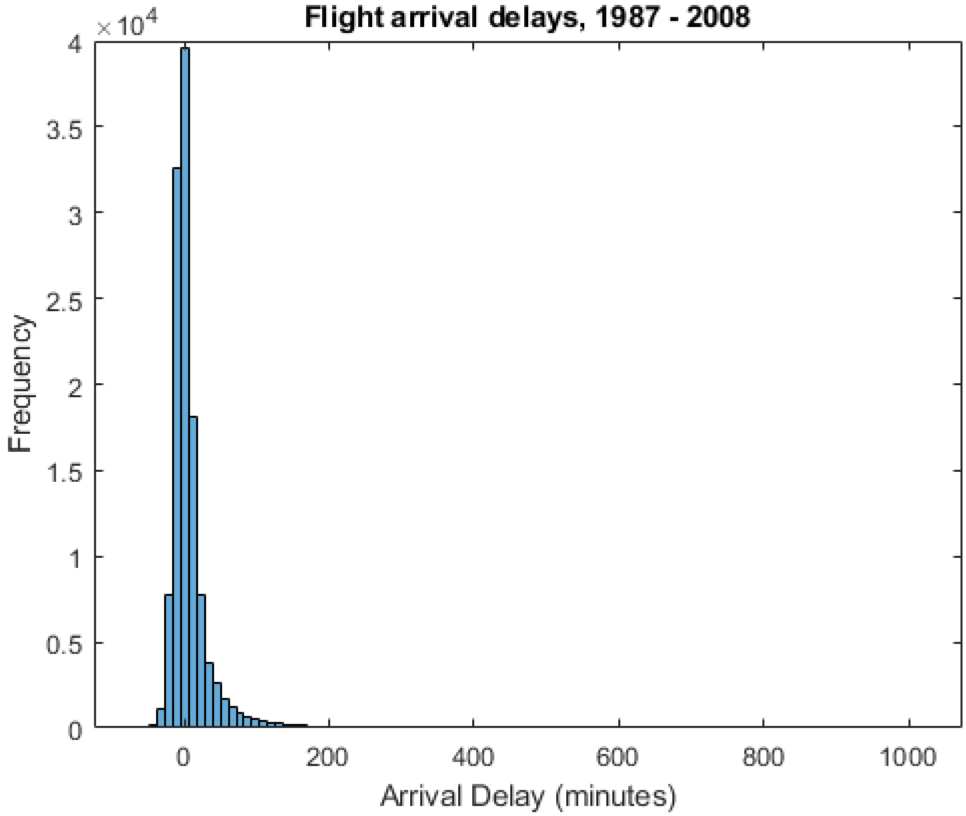
Plot Histogram of Arrival Delays

Plot a histogram of the `ArrDelay` variable to examine the frequency distribution of arrival delays.

```
h = histogram(T.ArrDelay);
```

```
Evaluating tall expression using the Local MATLAB Session:
- Pass 1 of 2: Completed in 1.9 sec
- Pass 2 of 2: Completed in 0.69 sec
Evaluation completed in 4.5 sec
```

```
title('Flight arrival delays, 1987 - 2008')
xlabel('Arrival Delay (minutes)')
ylabel('Frequency')
```



The arrival delay is most frequently a small number near 0, so these values dominate the plot and make it difficult to see other details.

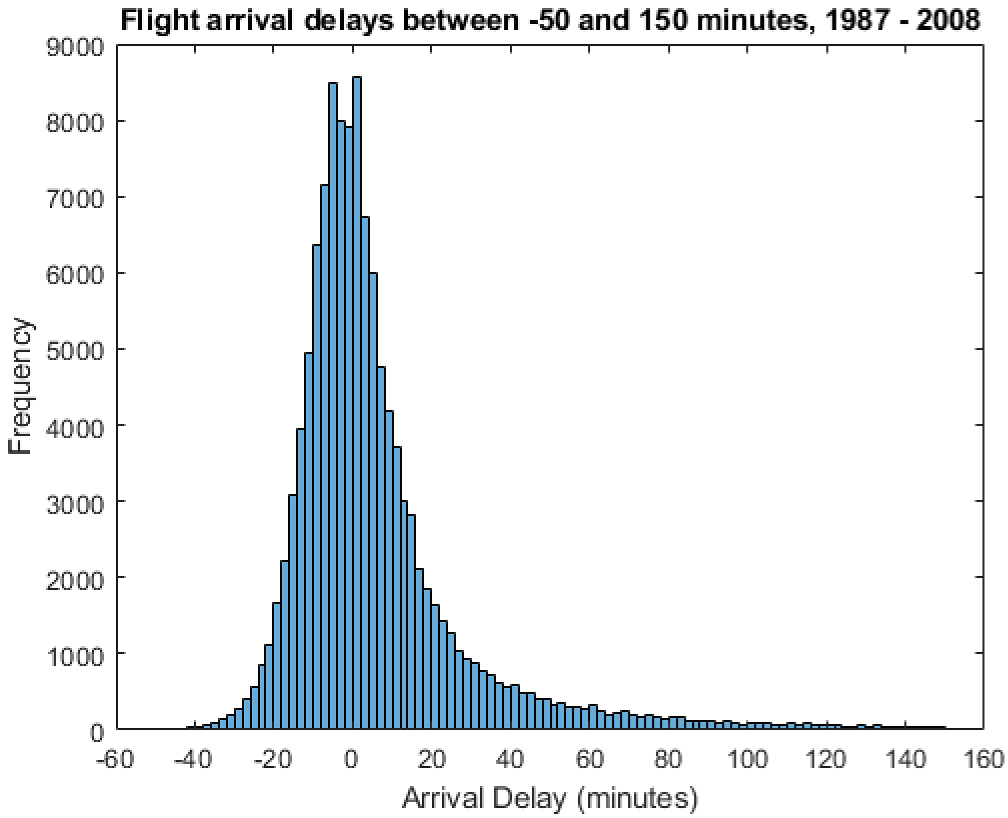
Adjust Bin Limits of Histogram

Restrict the histogram bin limits to plot only arrival delays between -50 and 150 minutes. After you create a histogram object from a tall array, you cannot change any properties that would require recomputing the bins, including `BinWidth` and `BinLimits`. Also, you cannot use `morebins` or `fewerbins` to adjust the number of bins. In these cases, use `histogram` to reconstruct the histogram from the raw data in the tall array.

```
figure
histogram(T.ArrDelay,'BinLimits',[-50,150])
```

```
Evaluating tall expression using the Local MATLAB Session:
- Pass 1 of 2: Completed in 0.84 sec
- Pass 2 of 2: Completed in 0.64 sec
Evaluation completed in 2 sec
```

```
title('Flight arrival delays between -50 and 150 minutes, 1987 - 2008')
xlabel('Arrival Delay (minutes)')
ylabel('Frequency')
```



From this plot, it appears that long delays might be more common than initially expected. To investigate further, find the probability of an arrival delay that is one hour or greater.

Probability of Delays One Hour or Greater

The original histogram returned an object `h` that contains the bin values in the `Values` property and the bin edges in the `BinEdges` property. You can use these properties to perform in-memory calculations.

Determine which bins contain arrival delays of one hour (60 minutes) or more. Remove the last bin edge from the logical index vector so that it is the same length as the vector of bin values.

```
idx = h.BinEdges >= 60;
idx(end) = [];
```

Use `idx` to retrieve the value associated with each selected bin. Add the bin values together, divide by the total number of samples, and multiply by 100 to determine the overall probability of a delay greater than or equal to one hour. Since the total number of samples is computed from the original data set, use `gather` to explicitly evaluate the calculation and return an in-memory scalar.

```
N = numel(T.ArrDelay);
P = gather(sum(h.Values(idx))*100/N)
```

```
P = 4.4809
```

Overall, the odds of an arrival delay one hour or longer are about 4.5%.

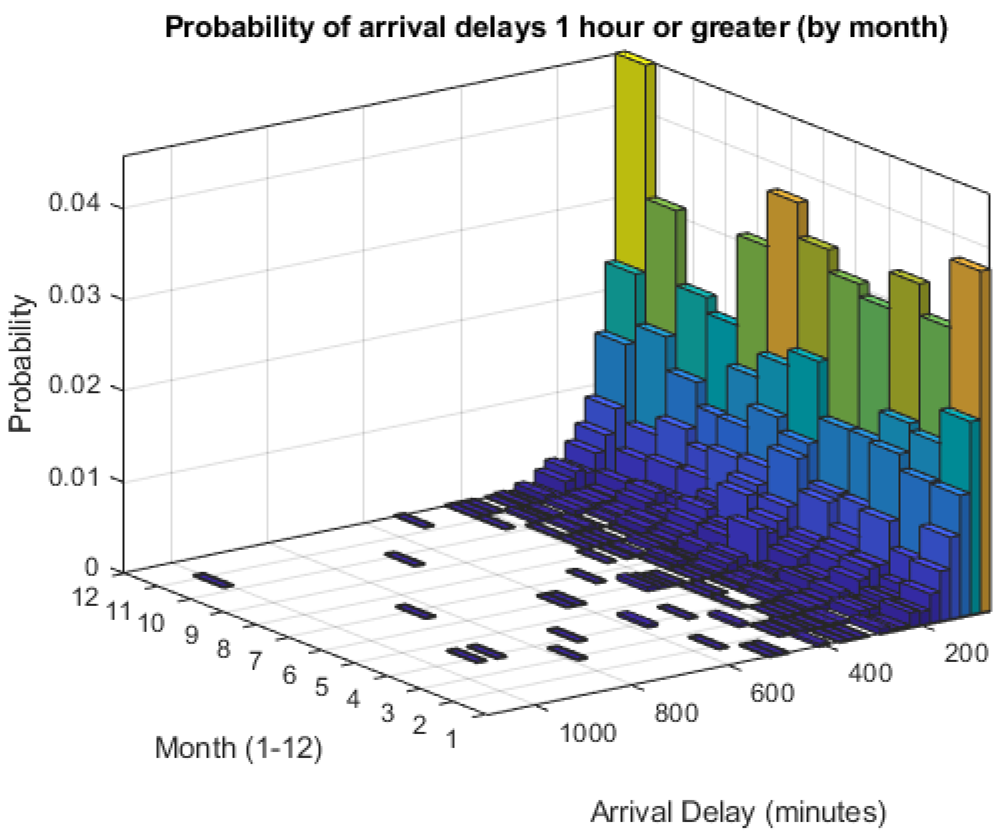
Plot Bivariate Histogram of Delays by Month

Plot a bivariate histogram of the arrival delays that are 60 minutes or longer by month. This plot examines how seasonality affects arrival delay.

```
figure
h2 = histogram2(T.Month,T.ArrDelay,[12 50],'YBinLimits',[60 1100],...
    'Normalization','probability','FaceColor','flat');
```

```
Evaluating tall expression using the Local MATLAB Session:
- Pass 1 of 1: Completed in 1.5 sec
Evaluation completed in 1.8 sec
Evaluating tall expression using the Local MATLAB Session:
- Pass 1 of 1: Completed in 0.78 sec
Evaluation completed in 0.9 sec
```

```
title('Probability of arrival delays 1 hour or greater (by month)')
xlabel('Month (1-12)')
ylabel('Arrival Delay (minutes)')
zlabel('Probability')
xticks(1:12)
view(-126,23)
```



Delay Statistics by Month

Use the bivariate histogram object to calculate the probability of having an arrival delay one hour or greater in each month, and the mean arrival delay for each month. Put the results in a table with the variable P containing the probability information and the variable MeanByMonth containing the mean arrival delay.

```
monthNames = {'Jan','Feb','Mar','Apr','May','Jun',...
    'Jul','Aug','Sep','Oct','Nov','Dec'}';
G = findgroups(T.Month);
M = splitapply(@(x) mean(x,'omitnan'),T.ArrDelay,G);
delayByMonth = table(monthNames, sum(h2.Values,2)*100, gather(M), ...
    'VariableNames',{'Month','P','MeanByMonth'})
```

```
Evaluating tall expression using the Local MATLAB Session:
- Pass 1 of 2: Completed in 0.81 sec
- Pass 2 of 2: Completed in 1.7 sec
Evaluation completed in 3.5 sec

delayByMonth=12×3 table
     Month       P       MeanByMonth
    _______    ______    ___________

    {'Jan'}    9.6497      8.5954
    {'Feb'}    7.7058      7.3275
    {'Mar'}    9.0543      7.5536
    {'Apr'}    7.2504      6.0081
    {'May'}    7.4256      5.2949
    {'Jun'}     10.35      10.264
    {'Jul'}    10.228      8.7797
    {'Aug'}    8.5989      7.4522
    {'Sep'}    5.4116      3.6308
    {'Oct'}     6.042      4.6059
    {'Nov'}    6.9002      5.2835
    {'Dec'}    11.384      10.571
```

The results indicate that flights in the holiday month of December have an 11.4% chance of being delayed longer than an hour, but are delayed by 10.5 minutes on average. This is closely followed by the summer months of June and July, where there is about a 10% chance of being delayed an hour or more and the average delay is roughly 9 or 10 minutes.

See Also

histogram | histogram2 | tall

More About

- “Tall Arrays for Out-of-Memory Data” on page 12-136

Visualization of Tall Arrays

Visualizing large data sets requires that the data is summarized, binned, or sampled in some way to reduce the number of points that are plotted on the screen. In some cases, functions such as `histogram` and `pie` bin the data to reduce the size, while other functions such as `plot` and `scatter` use a more complex approach that avoids plotting duplicate pixels on the screen. For problems where the pixel overlap is relevant to the analysis, the `binscatter` function also offers an efficient way to visualize density patterns.

Visualizing tall arrays does *not* require the use of `gather`. MATLAB immediately evaluates and displays visualizations of tall arrays. Currently, you can visualize tall arrays using the functions and methods in this table.

<table>
<tr><th>Function</th><th>Required Toolboxes</th><th>Notes</th></tr>
<tr><td><code>plot</code></td><td>—</td><td rowspan="3">These functions plot in iterations, progressively adding to the plot as more data is read. During the updates, a progress indicator shows the proportion of data that has been plotted. Zooming and panning is supported during the updating process, before the plot is complete. To stop the update process, press the pause button in the progress indicator.</td></tr>
<tr><td><code>scatter</code></td><td>—</td></tr>
<tr><td><code>binscatter</code></td><td>—</td></tr>
<tr><td><code>histogram</code></td><td>—</td><td></td></tr>
<tr><td><code>histogram2</code></td><td>—</td><td></td></tr>
<tr><td><code>pie</code></td><td>—</td><td>For visualizing categorical data only.</td></tr>
<tr><td><code>binScatterPlot</code></td><td>Statistics and Machine Learning Toolbox™</td><td>Figure contains a slider to control the brightness and color detail in the image. The slider adjusts the value of the <code>Gamma</code> image correction parameter.</td></tr>
<tr><td><code>ksdensity</code></td><td>Statistics and Machine Learning Toolbox</td><td>Produces a probability density estimate for the data, evaluated at 100 points for univariate data, or 900 points for bivariate data.</td></tr>
</table>

Function	Required Toolboxes	Notes
datasample	Statistics and Machine Learning Toolbox	datasample enables you to extract a subsample of a tall array in a statistically sound way compared to simple indexing. If the subset of data is small enough to fit in memory, then you can use plotting and fitting functions on the subset that do not directly support tall arrays.

Tall Array Plotting Examples

This example shows several different ways you can visualize tall arrays.

Create a datastore for the `airlinesmall.csv` data set, which contains rows of airline flight data. Select a subset of the table variables to work with and remove rows that contain missing values.

```
ds = tabularTextDatastore('airlinesmall.csv','TreatAsMissing','NA');
ds.SelectedVariableNames = {'Year','Month','ArrDelay','DepDelay','Origin','Dest'};
T = tall(ds);
T = rmmissing(T)
```

```
T =

  Mx6 tall table

    Year    Month    ArrDelay    DepDelay    Origin     Dest  
    ____    _____    ________    ________    _______    _______

    1987     10          8          12       {'LAX'}    {'SJC'}
    1987     10          8           1       {'SJC'}    {'BUR'}
    1987     10         21          20       {'SAN'}    {'SMF'}
    1987     10         13          12       {'BUR'}    {'SJC'}
    1987     10          4          -1       {'SMF'}    {'LAX'}
    1987     10         59          63       {'LAX'}    {'SJC'}
    1987     10          3          -2       {'SAN'}    {'SFO'}
    1987     10         11          -1       {'SEA'}    {'LAX'}
      :        :        :           :           :          :
      :        :        :           :           :          :
```

Pie Chart of Flights by Month

Convert the numeric `Month` variable into a categorical variable that reflects the name of the month. Then plot a pie chart showing how many flights are in the data for each month of the year.

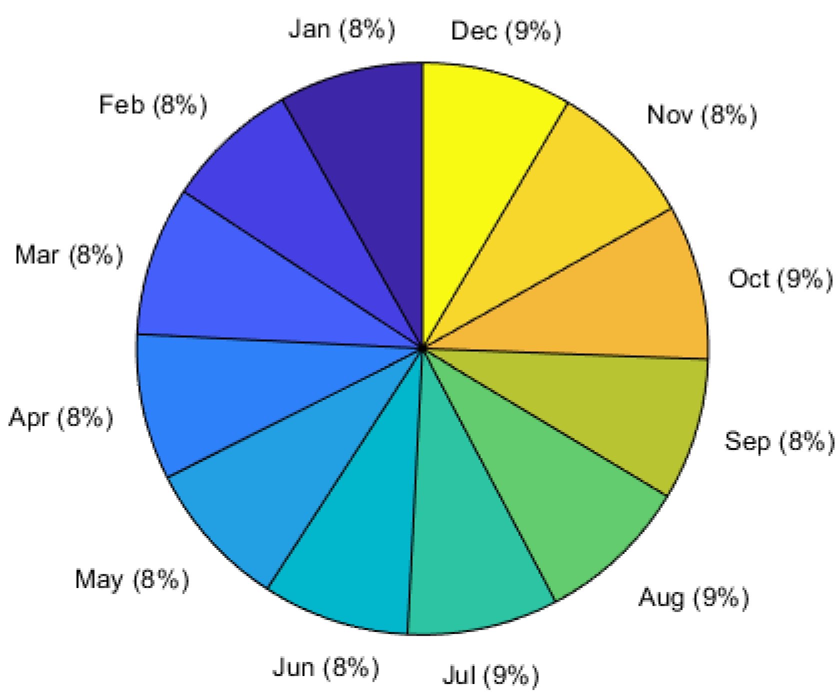
```
T.Month = categorical(T.Month,1:12,{'Jan','Feb','Mar','Apr','May','Jun','Jul','Aug','Sep','Oct',
```

```
T =

  Mx6 tall table

    Year    Month    ArrDelay    DepDelay    Origin     Dest  
    ____    _____    ________    ________    _______    _______
```

```
    1987      Oct          8          12        {'LAX'}    {'SJC'}
    1987      Oct          8           1        {'SJC'}    {'BUR'}
    1987      Oct         21          20        {'SAN'}    {'SMF'}
    1987      Oct         13          12        {'BUR'}    {'SJC'}
    1987      Oct          4          -1        {'SMF'}    {'LAX'}
    1987      Oct         59          63        {'LAX'}    {'SJC'}
    1987      Oct          3          -2        {'SAN'}    {'SFO'}
    1987      Oct         11          -1        {'SEA'}    {'LAX'}
     :         :          :           :            :          :
     :         :          :           :            :          :
```

```
pie(T.Month)
```

```
Evaluating tall expression using the Local MATLAB Session:
- Pass 1 of 2: Completed in 1.3 sec
- Pass 2 of 2: Completed in 1.1 sec
Evaluation completed in 3 sec
```



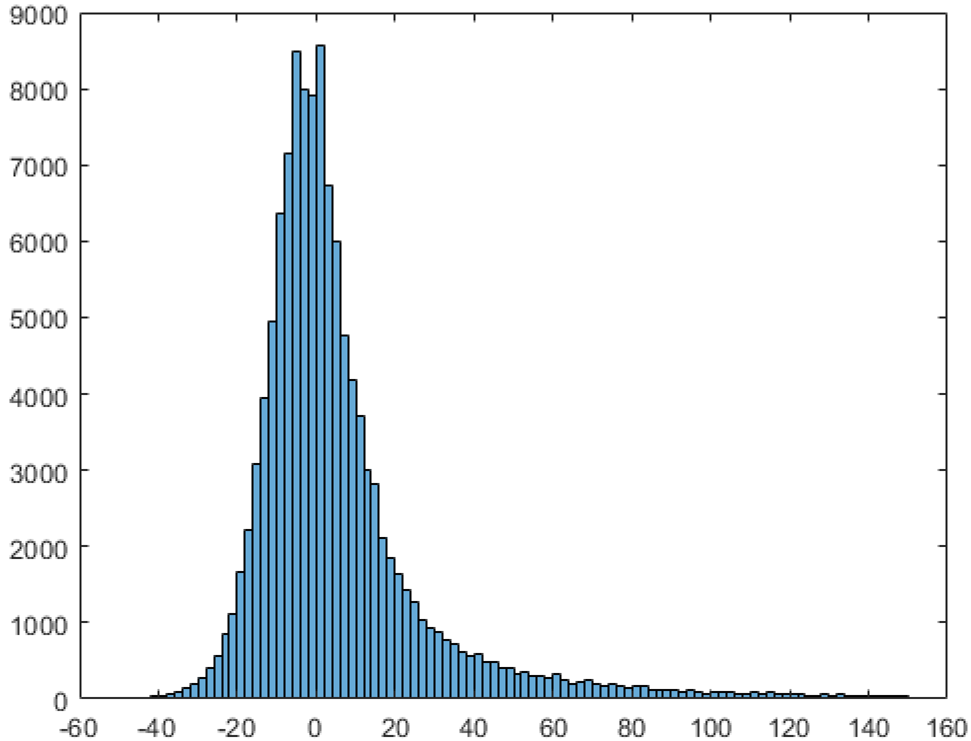
Histogram of Delays

Plot a histogram of the arrival delays for each flight in the data. Since the data has a long tail, limit the plotting area using the `BinLimits` name-value pair.

```
histogram(T.ArrDelay,'BinLimits',[-50 150])
```

```
Evaluating tall expression using the Local MATLAB Session:
- Pass 1 of 2: Completed in 2.4 sec
```

```
- Pass 2 of 2: Completed in 0.76 sec
Evaluation completed in 4 sec
```

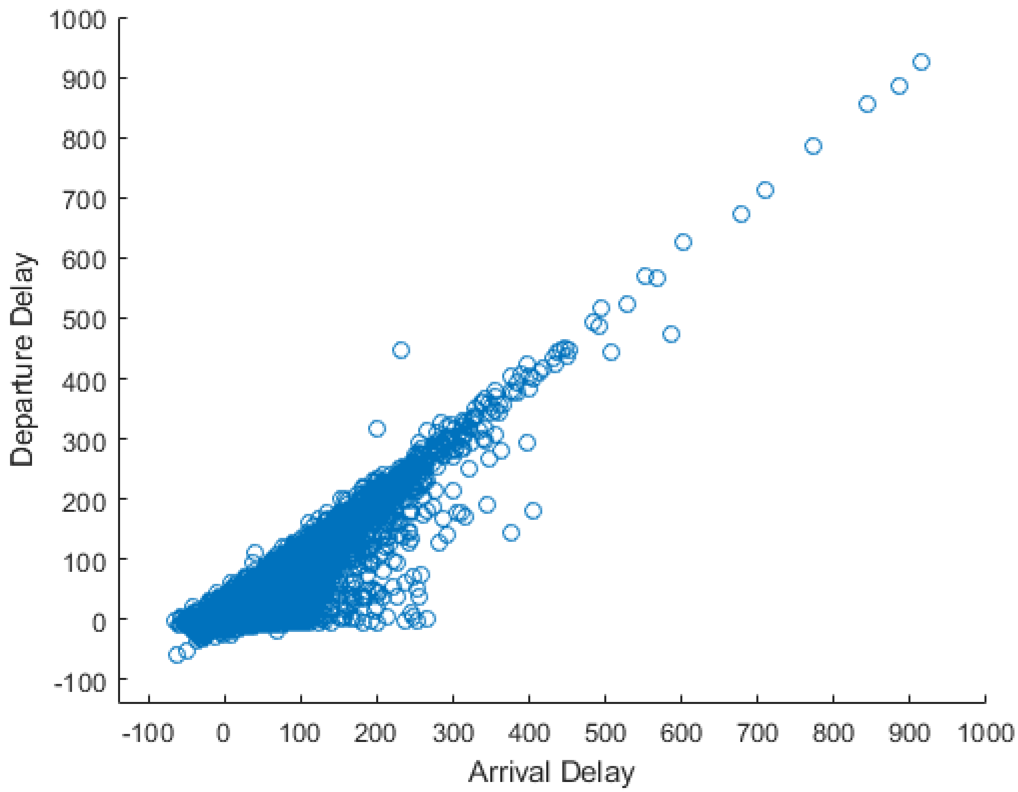


Scatter Plot of Delays

Plot a scatter plot of arrival and departure delays. You can expect a strong correlation between these variables since flights that leave late are also likely to arrive late.

When operating on tall arrays, the `plot`, `scatter`, and `binscatter` functions plot the data in iterations, progressively adding to the plot as more data is read. During the updates the top of the plot has a progress indicator showing how much data has been plotted. Zooming and panning is supported during the updates before the plot is complete.

```
scatter(T.ArrDelay,T.DepDelay)
xlabel('Arrival Delay')
ylabel('Departure Delay')
xlim([-140 1000])
ylim([-140 1000])
```

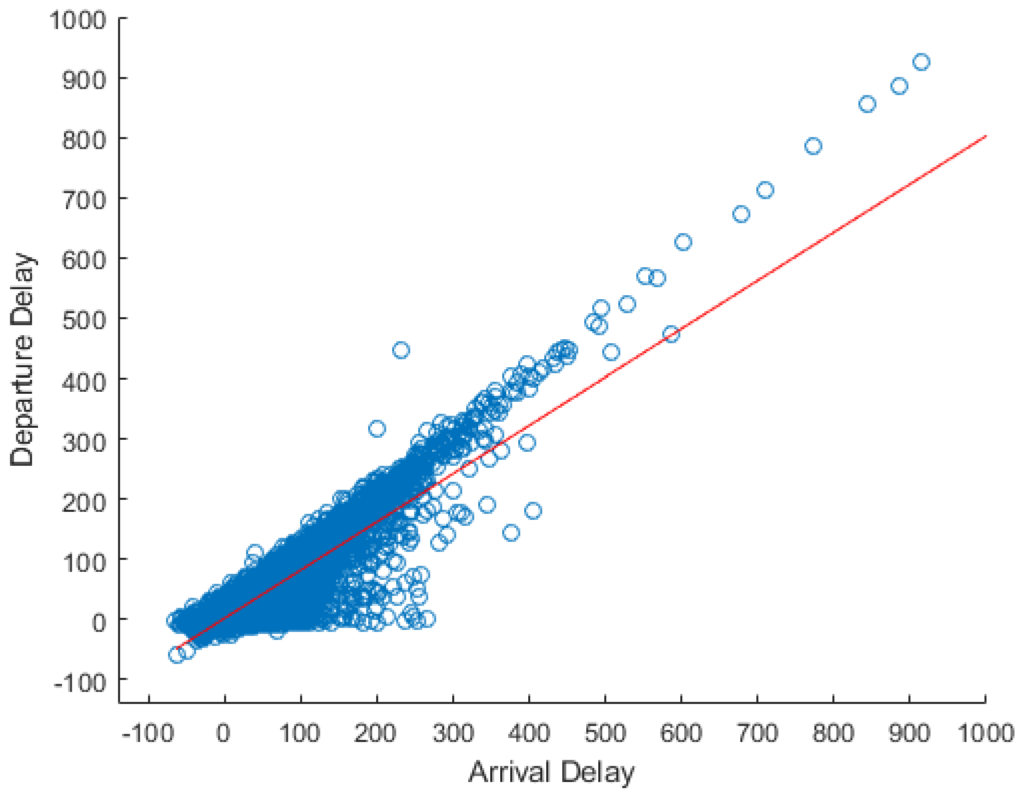


The progress bar also includes a **Pause/Resume** button. Use the button to stop the plot updates early once enough data is displayed.

Fit Trend Line

Use the `polyfit` and `polyval` functions to overlay a linear trend line on the plot of arrival and departure delays.

```
hold on
p = polyfit(T.ArrDelay,T.DepDelay,1);
x = sort(T.ArrDelay,1);
yp = polyval(p,x);
plot(x,yp,'r-')
hold off
```

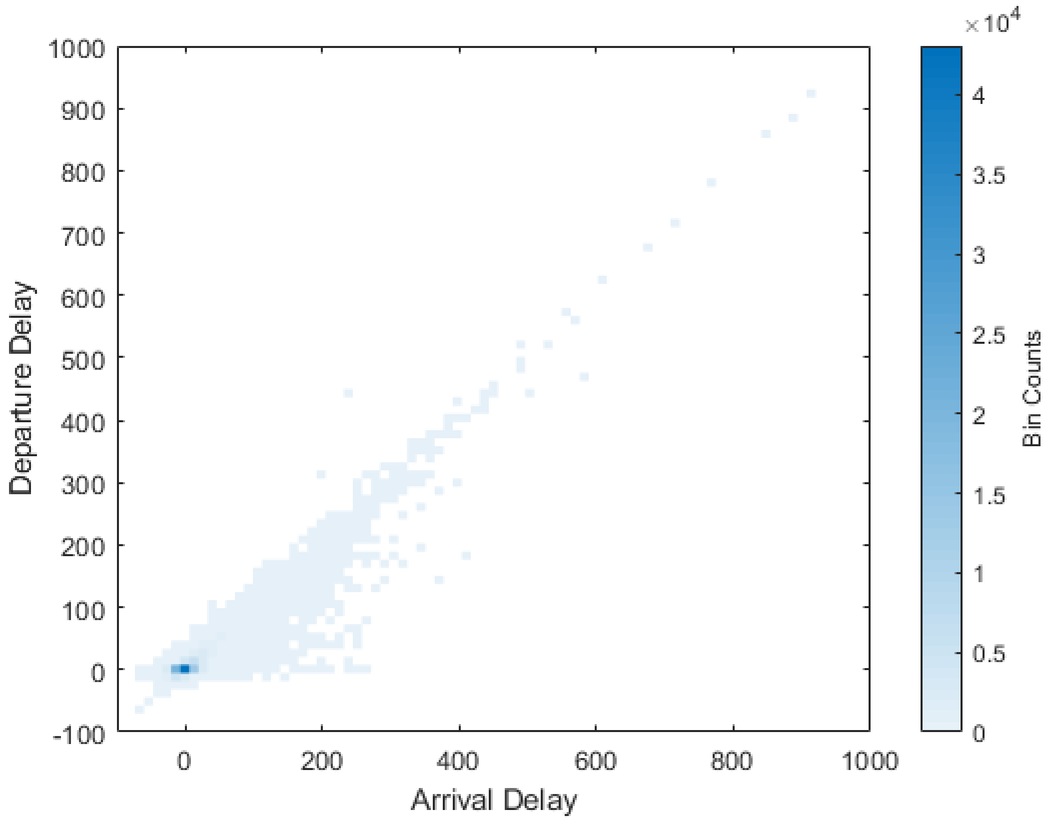


Visualize Density

The scatter plot of points is helpful up to a certain point, but it can be hard to decipher information from the plot if the points overlap extensively. In that case, it helps to visualize the density of points in the plot to spot trends.

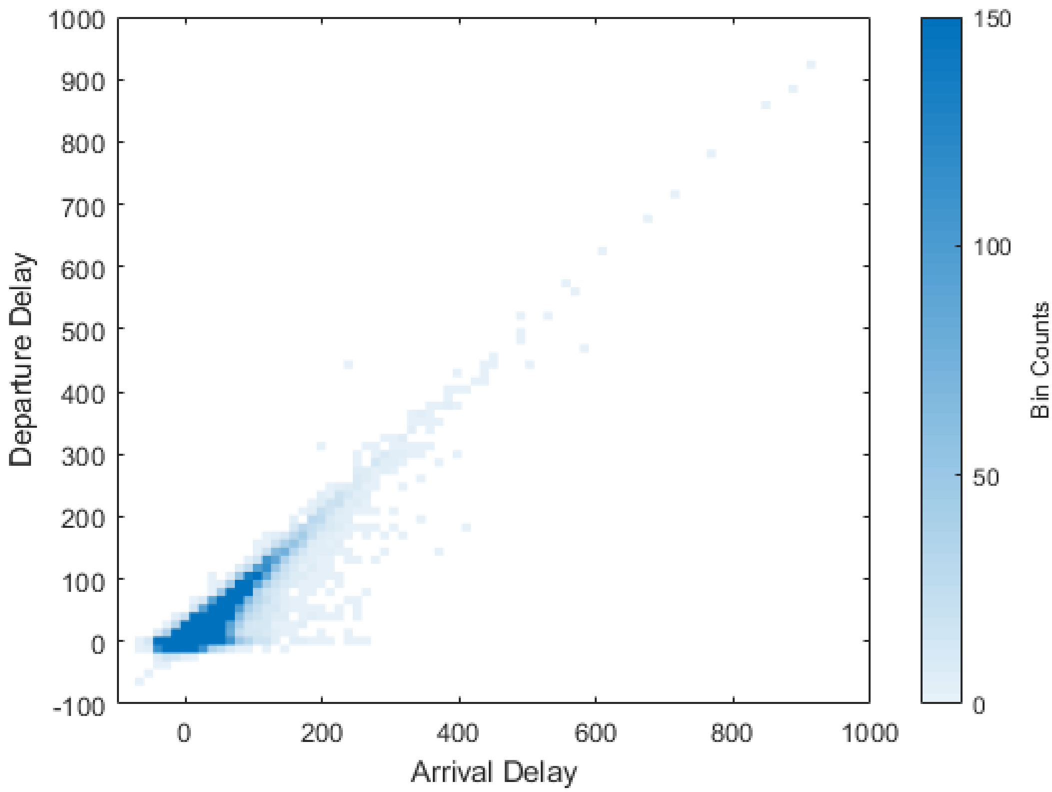
Use the `binscatter` function to visualize the density of points in the plot of arrival and departure delays.

```
binscatter(T.ArrDelay,T.DepDelay,'XLimits',[-100 1000],'YLimits',[-100 1000])
xlim([-100 1000])
ylim([-100 1000])
xlabel('Arrival Delay')
ylabel('Departure Delay')
```



Adjust the `CLim` property of the axes so that all bin values greater than 150 are colored the same. This prevents a few bins with very large values from dominating the plot.

```
ax = gca;
ax.CLim = [0 150];
```



See Also

`plot` | `polyfit` | `tall`

More About

- "Tall Arrays for Out-of-Memory Data" on page 12-136

Grouped Statistics Calculations with Tall Arrays

This example shows how to calculate grouped statistics of a tall timetable containing power outage data. The example uses the `grouptransform`, `groupsummary`, and `groupcounts` functions to calculate various quantities of interest, such as the most common power outage cause in each region. Even though the raw data in this example only has about 1500 rows, you can use the techniques presented here on much larger data sets because no assumptions are made about the size of the data.

Create Datastore and Tall Timetable

The sample file, `outages.csv`, contains data representing electric utility outages in the United States. The file contains six columns: `Region`, `OutageTime`, `Loss`, `Customers`, `RestorationTime`, and `Cause`.

Create a datastore for the `outages.csv` file. Use the `"TextScanFormats"` option to specify the kind of data each column contains: categorical (`"%C"`), floating-point numeric (`"%f"`), or datetime (`"%D"`).

```
data_formats = ["%C","%D","%f","%f","%D","%C"];
ds = tabularTextDatastore("outages.csv","TextscanFormats",data_formats);
```

Create a tall table on top of the datastore, and convert the tall table into a tall timetable. The `OutageTime` variable is used for the row times since it is the first datetime or duration variable in the table.

```
T = tall(ds);
T = table2timetable(T)
```

```
T =

  Mx5 tall timetable

    OutageTime    Region    Loss    Customers    RestorationTime    Cause
    __________    ______    ____    _________    _______________    _____

        ?           ?        ?          ?               ?             ?
        ?           ?        ?          ?               ?             ?
        ?           ?        ?          ?               ?             ?
        :           :        :          :               :             :
        :           :        :          :               :             :
```

Replace Missing Data

Some of the rows in the `RestorationTime` variable have missing times, represented by `NaT` values. Remove these rows from the table.

```
T = rmmissing(T,"DataVariables","RestorationTime");
```

For the numeric variables in the timetable, instead of removing rows with missing values, replace the missing values with the mean value for each region.

```
T = grouptransform(T,"Region","meanfill",["Loss","Customers"]);
```

Use `ismissing` to confirm that no pieces of missing data remain in the table.

```
tf = any(ismissing(T),"all");
gather(tf)
```

```
Evaluating tall expression using the Local MATLAB Session:
- Pass 1 of 4: Completed in 0.47 sec
- Pass 2 of 4: Completed in 1.6 sec
- Pass 3 of 4: Completed in 0.87 sec
- Pass 4 of 4: Completed in 1.1 sec
Evaluation completed in 6.2 sec

ans = logical
   0
```

Preview Data

Now that the data does not contain missing values, bring a small number of rows into memory to get an idea of what the data contains.

```
gather(head(T))
```

```
Evaluating tall expression using the Local MATLAB Session:
- Pass 1 of 1: Completed in 0.59 sec
Evaluation completed in 0.8 sec

ans=8×5 timetable
       OutageTime          Region        Loss      Customers      RestorationTime           Cause
    ________________    _________    ______    __________    ________________    ________________

    2002-02-01 12:18    SouthWest    458.98    1.8202e+06    2002-02-07 16:50    winter storm
    2003-02-07 21:15    SouthEast     289.4    1.4294e+05    2003-02-17 08:14    winter storm
    2004-04-06 05:44    West         434.81    3.4037e+05    2004-04-06 06:10    equipment fault
    2002-03-16 06:18    MidWest      186.44    2.1275e+05    2002-03-18 23:23    severe storm
    2003-06-18 02:49    West              0             0    2003-06-18 10:54    attack
    2004-06-20 14:39    West         231.29    1.5354e+05    2004-06-20 19:16    equipment fault
    2002-06-06 19:28    West         311.86    1.5354e+05    2002-06-07 00:51    equipment fault
    2003-07-16 16:23    NorthEast    239.93         49434    2003-07-17 01:12    fire
```

Mean Power Outage Duration by Region

Determine the mean power outage duration in each region using `groupsummary`. First, create a new variable `OutageDuration` in the table that contains the duration of each outage, found by subtracting the outage time from the restoration time. In the call to `groupsummary`, specify:

- `"Region"` as the grouping variable
- `"mean"` as the computation method
- `"OutageDuration"` as the variable to operate on.

```
T.OutageDuration = T.RestorationTime - T.OutageTime;
times = groupsummary(T,"Region","mean","OutageDuration")
```

```
times =

  Mx3 tall table

    Region    GroupCount    mean_OutageDuration
    ______    __________    ___________________

      ?           ?                  ?
      ?           ?                  ?
```

```
    ?          ?                  ?
    :          :                  :
    :          :                  :
```

Change the display format of the duration results to be in days, and gather the results into memory. The results show the mean outage duration in each region, as well as the number of reported outages in each region.

```
times.mean_OutageDuration.Format = "d";
times = gather(times)
```

```
Evaluating tall expression using the Local MATLAB Session:
- Pass 1 of 2: Completed in 2.1 sec
- Pass 2 of 2: Completed in 1.6 sec
Evaluation completed in 4.9 sec

times=5×3 table
     Region      GroupCount    mean_OutageDuration
    _________    __________    ___________________

    MidWest         138            34.135 days    
    NorthEast       548             24.21 days    
    SouthEast       379            1.7013 days    
    SouthWest        25            2.4799 days    
    West            349            28.061 days    
```

Most Common Power Outage Causes by Region

Determine how often each power outage cause occurs in each region. Use `groupcounts` with the `Cause` and `Region` variables as grouping variables. Gather the results into memory.

```
causes = groupcounts(T,["Cause","Region"]);
causes = gather(causes)
```

```
Evaluating tall expression using the Local MATLAB Session:
- Pass 1 of 2: Completed in 0.47 sec
- Pass 2 of 2: Completed in 0.83 sec
Evaluation completed in 1.8 sec

causes=43×3 table
          Cause            Region      GroupCount
    ________________    _________    __________

    attack              MidWest          12    
    attack              NorthEast       135    
    attack              SouthEast        19    
    attack              West            126    
    earthquake          NorthEast         1    
    earthquake          West              1    
    energy emergency    MidWest          19    
    energy emergency    NorthEast        29    
    energy emergency    SouthEast        79    
    energy emergency    SouthWest         7    
    energy emergency    West             46    
    equipment fault     MidWest           9    
    equipment fault     NorthEast        17    
    equipment fault     SouthEast        40    
    equipment fault     SouthWest         2    
```

```
    equipment fault     West             85
      ⋮
```

Each cause occurs several times in the table, so even though the table contains the correct data it is not in the proper format to see how often each cause occurs in each region. To improve the presentation of the data, unstack the GroupCount variable so that each column corresponds to a region and each row corresponds to an outage cause.

```
RegionCauses = gather(unstack(causes,"GroupCount","Region"))
```

```
RegionCauses=10×6 table
         Cause           MidWest    NorthEast    SouthEast    SouthWest    West
    ________________    _______    _________    _________    _________    ____

    attack                 12         135           19          NaN       126
    earthquake            NaN           1          NaN          NaN         1
    energy emergency       19          29           79            7        46
    equipment fault         9          17           40            2        85
    fire                  NaN           5            3          NaN        17
    severe storm           30         139          132            6        22
    thunder storm          31         102           54            6         7
    unknown                 4          10            3          NaN         4
    wind                   16          40           13            3        22
    winter storm           17          70           36            1        19
```

Not all combinations of outage causes and regions are represented in the data, so the resulting table contains some NaNs. Fill in the NaN values with zeros.

```
RegionCauses = fillmissing(RegionCauses,"constant",{"",0,0,0,0,0})
```

```
RegionCauses=10×6 table
         Cause           MidWest    NorthEast    SouthEast    SouthWest    West
    ________________    _______    _________    _________    _________    ____

    attack                12          135           19           0        126
    earthquake             0            1            0           0          1
    energy emergency      19           29           79           7         46
    equipment fault        9           17           40           2         85
    fire                   0            5            3           0         17
    severe storm          30          139          132           6         22
    thunder storm         31          102           54           6          7
    unknown                4           10            3           0          4
    wind                  16           40           13           3         22
    winter storm          17           70           36           1         19
```

Worst Power Outages in Each Region

Calculate the broadest customer impact for each power outage in each region.

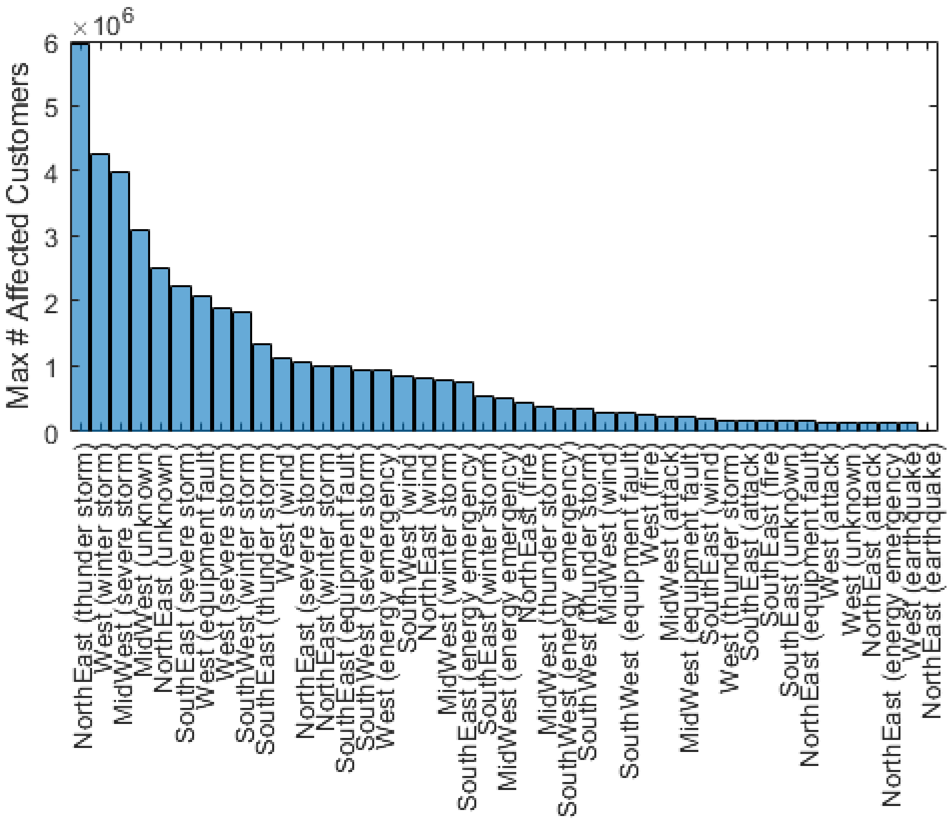
```
WorstOutages = groupsummary(T,["Region","Cause"],"max","Customers");
WorstOutages = gather(WorstOutages)
```

```
Evaluating tall expression using the Local MATLAB Session:
- Pass 1 of 2: Completed in 0.5 sec
- Pass 2 of 2: Completed in 0.89 sec
Evaluation completed in 1.9 sec
```

```
WorstOutages=43×4 table
     Region            Cause           GroupCount    max_Customers
    _________    _________________    __________    _____________

    MidWest      attack                   12          2.4403e+05
    MidWest      energy emergency         19          5.0376e+05
    MidWest      equipment fault           9          2.4403e+05
    MidWest      severe storm             30           3.972e+06
    MidWest      thunder storm            31          3.8233e+05
    MidWest      unknown                   4          3.0879e+06
    MidWest      wind                     16          2.8666e+05
    MidWest      winter storm             17          7.7697e+05
    NorthEast    attack                  135          1.5005e+05
    NorthEast    earthquake                1                   0
    NorthEast    energy emergency         29          1.5005e+05
    NorthEast    equipment fault          17           1.667e+05
    NorthEast    fire                      5          4.5139e+05
    NorthEast    severe storm            139          1.0735e+06
    NorthEast    thunder storm           102          5.9689e+06
    NorthEast    unknown                  10          2.4983e+06
      ⋮
```

Combine the data in the `Region` and `Cause` variables into a single categorical variable by briefly converting them into strings. Then, create a categorical histogram of the maximum number of affected customers for each cause in each region.

```
WorstOutages.RegionCause = categorical(string(WorstOutages.Region)+" ("+string(WorstOutages.Cause
histogram("Categories",WorstOutages.RegionCause,"BinCounts",WorstOutages.max_Customers,...
    "DisplayOrder","descend")
ylabel("Max # Affected Customers")
```



See Also

`findgroups` | `splitapply` | `tall`

More About

- "Grouping Variables To Split Data"
- "Split Data into Groups and Calculate Statistics"
- "Split Table Data Variables and Apply Functions"

Extend Tall Arrays with Other Products

Products Used: Statistics and Machine Learning Toolbox, Database Toolbox, Parallel Computing Toolbox, MATLAB Parallel Server, MATLAB Compiler

Several toolboxes enhance the capabilities of tall arrays. These enhancements include writing machine learning algorithms, integrating with big data systems, and deploying standalone apps.

Statistics and Machine Learning

Statistics and Machine Learning Toolbox enables you to perform advanced statistical calculations on tall arrays. Capabilities include:

- K-means clustering
- Linear regression fitting
- Grouped statistics
- Classification

See “Analysis of Big Data with Tall Arrays” (Statistics and Machine Learning Toolbox) for more information.

Control Where Your Code Runs

When you execute calculations on tall arrays, the default execution environment uses either the local MATLAB session, or a local parallel pool if you have Parallel Computing Toolbox. Use the `mapreducer` function to change the execution environment of tall arrays when using Parallel Computing Toolbox, MATLAB Parallel Server, or MATLAB Compiler:

- Parallel Computing Toolbox — Run calculations in parallel using local or cluster workers to speed up large tall array calculations. See “Use Tall Arrays on a Parallel Pool” (Parallel Computing Toolbox) or “Process Big Data in the Cloud” (Parallel Computing Toolbox) for more information.
- MATLAB Parallel Server — Run tall array calculations on a cluster, including Apache Spark™ enabled Hadoop clusters. This can significantly reduce the execution time of very large calculations. See “Use Tall Arrays on a Spark Enabled Hadoop Cluster” (Parallel Computing Toolbox) for more information.
- MATLAB Compiler — Deploy MATLAB applications containing tall arrays as standalone apps on Apache Spark. See “Spark Applications” (MATLAB Compiler) for more information.

One of the benefits of developing your algorithms with tall arrays is that you only need to write the code once. You can develop your code locally, then use `mapreducer` to scale up and take advantage of the capabilities offered by Parallel Computing Toolbox, MATLAB Parallel Server, or MATLAB Compiler, without needing to rewrite your algorithm.

Note Each tall array is bound to a single execution environment when it is constructed using `tall(ds)`. If that execution environment is later modified or deleted, then the tall array becomes invalid.

For this reason, each time you change the execution environment you must reconstruct the tall array.

Work with Databases

Database Toolbox enables you to create a tall table from a `DatabaseDatastore` that is backed by data in a database. For more information, see “Analyze Large Data in Database Using Tall Arrays” (Database Toolbox).

Note `DatabaseDatastore` has these limitations:

- `DatabaseDatastore` must use the local MATLAB session as the execution environment. Set this environment using the command `mapreducer(0)`.
- Standalone applications containing tall arrays that use `DatabaseDatastore` cannot be deployed against Apache Spark using MATLAB Compiler.

See Also
`gcmr` | `mapreducer` | `tall`

More About
- “Tall Arrays for Out-of-Memory Data” on page 12-136

Analyze Big Data in MATLAB Using Tall Arrays

This example shows how to use tall arrays to work with big data in MATLAB®. You can use tall arrays to perform a variety of calculations on different types of data that does not fit in memory. These include basic calculations, as well as machine learning algorithms within Statistics and Machine Learning Toolbox™.

This example operates on a small subset of data on a single computer, and then it then scales up to analyze all of the data set. However, this analysis technique can scale up even further to work on data sets that are so large they cannot be read into memory, or to work on systems like Apache Spark™.

Introduction to Tall Arrays

Tall arrays and tall tables are used to work with out-of-memory data that has any number of rows. Instead of writing specialized code that takes into account the huge size of the data, tall arrays and tables let you work with large data sets in a manner similar to in-memory MATLAB® arrays. The difference is that `tall` arrays typically remain unevaluated until you request that the calculations be performed.

This deferred evaluation enables MATLAB to combine the queued calculations where possible and take the minimum number of passes through the data. Since the number of passes through the data greatly affects execution time, it is recommended that you request output only when necessary.

Create datastore for Collection of Files

Creating a datastore enables you to access a collection of data. A datastore can process arbitrarily large amounts of data, and the data can even be spread across multiple files in multiple folders. You can create a datastore for most types of files, including a collection of tabular text files (demonstrated here), spreadsheets, images, a SQL database (Database Toolbox™ required), Hadoop® sequence files, and more.

Create a datastore for a `.csv` file containing airline data. Treat `'NA'` values as missing so that `tabularTextDatastore` replaces them with `NaN` values. Select the variables of interest, and specify a categorical data type for the `Origin` and `Dest` variables. Preview the contents.

```
ds = tabularTextDatastore('airlinesmall.csv');
ds.TreatAsMissing = 'NA';
ds.SelectedVariableNames = {'Year','Month','ArrDelay','DepDelay','Origin','Dest'};
ds.SelectedFormats(5:6) = {'%C','%C'};
pre = preview(ds)
```

```
pre=8×6 table
    Year    Month    ArrDelay    DepDelay    Origin    Dest
    ____    _____    ________    ________    ______    ____

    1987     10          8          12        LAX       SJC 
    1987     10          8           1        SJC       BUR 
    1987     10         21          20        SAN       SMF 
    1987     10         13          12        BUR       SJC 
    1987     10          4          -1        SMF       LAX 
    1987     10         59          63        LAX       SJC 
    1987     10          3          -2        SAN       SFO 
    1987     10         11          -1        SEA       LAX 
```

Create Tall Array

Tall arrays are similar to in-memory MATLAB arrays, except that they can have any number of rows. Tall arrays can contain data that is numeric, logical, datetime, duration, calendarDuration, categorical, or strings. Also, you can convert any in-memory array to a tall array. (The in-memory array A must be one of the supported data types.)

The underlying class of a tall array is based on the type of datastore that backs it. For example, if the datastore ds contains tabular data, then tall(ds) returns a tall table containing the data.

```
tt = tall(ds)
```

```
tt =

  Mx6 tall table

    Year    Month    ArrDelay    DepDelay    Origin    Dest
    ____    _____    ________    ________    ______    ____

     ?        ?         ?           ?          ?        ?
     ?        ?         ?           ?          ?        ?
     ?        ?         ?           ?          ?        ?
     :        :         :           :          :        :
     :        :         :           :          :        :
```

The display indicates the underlying data type and includes the first several rows of data. The size of the table displays as "Mx6" to indicate that MATLAB does not yet know how many rows of data there are.

Perform Calculations on Tall Arrays

You can work with tall arrays and tall tables in a similar manner in which you work with in-memory MATLAB arrays and tables.

One important aspect of tall arrays is that as you work with them, MATLAB does not perform most operations immediately. These operations appear to execute quickly, because the actual computation is deferred until you specifically request output. This deferred evaluation is important because even a simple command like size(X) executed on a tall array with a billion rows is not a quick calculation.

As you work with tall arrays, MATLAB keeps track of all of the operations to be carried out and optimizes the number of passes through the data. Thus, it is normal to work with unevaluated tall arrays and request output only when you require it. MATLAB does not know the contents or size of unevaluated tall arrays until you request that the array be evaluated and displayed.

Calculate the mean departure delay.

```
mDep = mean(tt.DepDelay,'omitnan')
```

```
mDep =

  tall double

    ?
```

Gather Results into Workspace

The benefit of deferred evaluation is that when the time comes for MATLAB to perform the calculations, it is often possible to combine the operations in such a way that the number of passes through the data is minimized. So, even if you perform many operations, MATLAB only makes extra passes through the data when absolutely necessary.

The `gather` function forces evaluation of all queued operations and brings the resulting output back into memory. Since `gather` returns the *entire* result in MATLAB, you should make sure that the result will fit in memory. For example, use `gather` on tall arrays that are the result of a function that reduces the size of the tall array, such as `sum`, `min`, `mean`, and so on.

Use `gather` to calculate the mean departure delay and bring the answer into memory. This calculation requires a single pass through the data, but other calculations might require several passes through the data. MATLAB determines the optimal number of passes for the calculation and displays this information at the command line.

```
mDep = gather(mDep)
```

```
Evaluating tall expression using the Local MATLAB Session:
- Pass 1 of 2: Completed in 1.3 sec
- Pass 2 of 2: Completed in 1.2 sec
Evaluation completed in 3.1 sec

mDep = 8.1860
```

Select Subset of Tall Array

You can extract values from a tall array by subscripting or indexing. You can index the array starting from the top or bottom, or by using a logical index. The functions `head` and `tail` are useful alternatives to indexing, enabling you to explore the first and last portions of a tall array. Gather both variables at the same time to avoid extra passes through the data.

```
h = head(tt);
tl = tail(tt);
[h,tl] = gather(h,tl)
```

```
Evaluating tall expression using the Local MATLAB Session:
- Pass 1 of 1: Completed in 0.81 sec
Evaluation completed in 0.99 sec

h=8×6 table
    Year    Month    ArrDelay    DepDelay    Origin    Dest
    ____    _____    ________    ________    ______    ____

    1987     10          8          12        LAX      SJC 
    1987     10          8           1        SJC      BUR 
    1987     10         21          20        SAN      SMF 
    1987     10         13          12        BUR      SJC 
    1987     10          4          -1        SMF      LAX 
    1987     10         59          63        LAX      SJC 
    1987     10          3          -2        SAN      SFO 
    1987     10         11          -1        SEA      LAX 


tl=8×6 table
    Year    Month    ArrDelay    DepDelay    Origin    Dest
    ____    _____    ________    ________    ______    ____

    2008     12         14           1        DAB      ATL 
    2008     12         -8          -1        ATL      TPA 
    2008     12          1           9        ATL      CLT 
```

```
    2008     12          -8           -4          ATL       CLT
    2008     12          15           -2          BOS       LGA
    2008     12         -15           -1          SFO       ATL
    2008     12         -12            1          DAB       ATL
    2008     12          -1           11          ATL       IAD
```

Use `head` to select a subset of 10,000 rows from the data for prototyping code before scaling to the full data set.

```
ttSubset = head(tt,10000);
```

Select Data by Condition

You can use typical logical operations on tall arrays, which are useful for selecting relevant data or removing outliers with logical indexing. The logical expression creates a tall logical vector, which then is used to subscript, identifying the rows where the condition is true.

Select only the flights out of Boston by comparing the elements of the categorical variable `Origin` to the value `'BOS'`.

```
idx = (ttSubset.Origin == 'BOS');
bosflights = ttSubset(idx,:)
```

```
bosflights =

  207x6 tall table

    Year    Month    ArrDelay    DepDelay    Origin    Dest
    ____    _____    ________    ________    ______    ____

    1987     10          -8            0      BOS       LGA
    1987     10         -13           -1      BOS       LGA
    1987     10          12           11      BOS       BWI
    1987     10          -3            0      BOS       EWR
    1987     10          -5            0      BOS       ORD
    1987     10          31           19      BOS       PHL
    1987     10          -3            0      BOS       CLE
    1987     11           5            5      BOS       STL
     :        :          :            :        :         :
     :        :          :            :        :         :
```

You can use the same indexing technique to remove rows with missing data or NaN values from the tall array.

```
idx = any(ismissing(ttSubset),2);
ttSubset(idx,:) = [];
```

Determine Largest Delays

Due to the nature of big data, sorting all of the data using traditional methods like `sort` or `sortrows` is inefficient. However, the `topkrows` function for tall arrays returns the top `k` rows in sorted order.

Calculate the top 10 greatest departure delays.

```
biggestDelays = topkrows(ttSubset,10,'DepDelay');
biggestDelays = gather(biggestDelays)
```

```
Evaluating tall expression using the Local MATLAB Session:
Evaluation completed in 0.074 sec

biggestDelays=10×6 table
    Year    Month    ArrDelay    DepDelay    Origin    Dest
    ____    _____    ________    ________    ______    ____

    1988      3        772         785        ORD      LEX
    1989      3        453         447        MDT      ORD
    1988     12        397         425        SJU      BWI
    1987     12        339         360        DEN      STL
    1988      3        261         273        PHL      ROC
    1988      7        261         268        BWI      PBI
    1988      2        257         253        ORD      BTV
    1988      3        236         240        EWR      FLL
    1989      2        263         227        BNA      MOB
    1989      6        224         225        DFW      JAX
```

Visualize Data in Tall Arrays

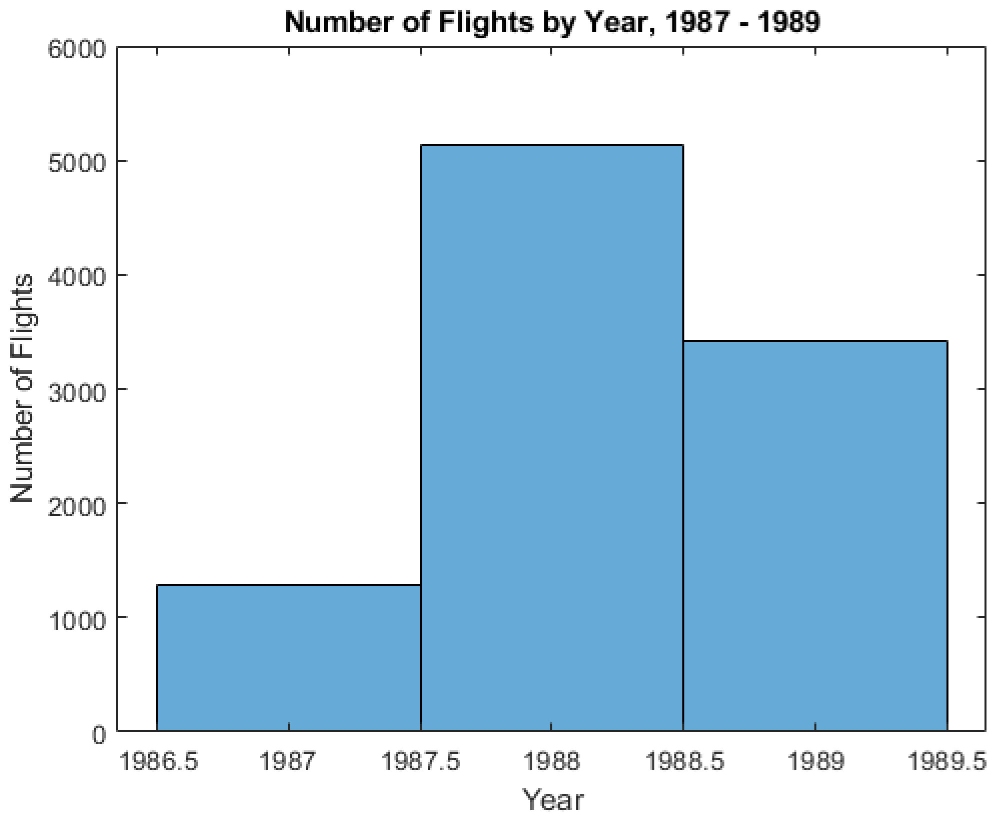
Plotting every point in a big data set is not feasible. For that reason, visualization of tall arrays involves reducing the number of data points using sampling or binning.

Visualize the number of flights per year with a histogram. The visualization functions pass through the data and immediately evaluate the solution when you call them, so `gather` is not required.

```
histogram(ttSubset.Year,'BinMethod','integers')
```

```
Evaluating tall expression using the Local MATLAB Session:
Evaluation completed in 0.46 sec
```

```
xlabel('Year')
ylabel('Number of Flights')
title('Number of Flights by Year, 1987 - 1989')
```



Scale to Entire Data Set

Instead of using the smaller data returned from `head`, you can scale up to perform the calculations on the entire data set by using the results from `tall(ds)`.

```
tt = tall(ds);
idx = any(ismissing(tt),2);
tt(idx,:) = [];
mnDelay = mean(tt.DepDelay,'omitnan');
biggestDelays = topkrows(tt,10,'DepDelay');
[mnDelay,biggestDelays] = gather(mnDelay,biggestDelays)
```

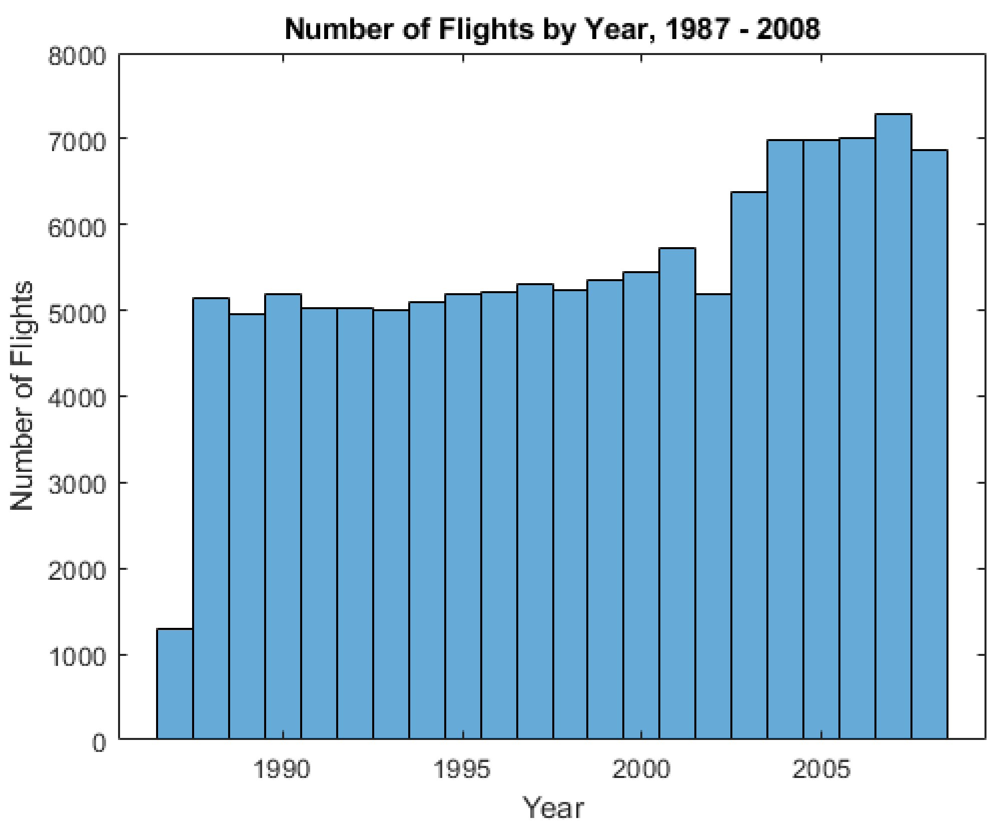
```
Evaluating tall expression using the Local MATLAB Session:
- Pass 1 of 2: Completed in 0.59 sec
- Pass 2 of 2: Completed in 1.2 sec
Evaluation completed in 2 sec

mnDelay = 8.1310

biggestDelays=10×6 table
    Year    Month    ArrDelay    DepDelay    Origin    Dest
    ____    _____    ________    ________    ______    ____

    1991      3          -8        1438       MCO      BWI
    1998     12         -12        1433       CVG      ORF
    1995     11        1014        1014       HNL      LAX
    2007      4         914         924       JFK      DTW
    2001      4         887         884       MCO      DTW
```

```
    2008       7          845          855        CMH       ORD
    1988       3          772          785        ORD       LEX
    2008       4          710          713        EWR       RDU
    1998      10          679          673        MCI       DFW
    2006       6          603          626        ABQ       PHX
```

```
histogram(tt.Year,'BinMethod','integers')
```

```
Evaluating tall expression using the Local MATLAB Session:
- Pass 1 of 2: Completed in 1.2 sec
- Pass 2 of 2: Completed in 0.8 sec
Evaluation completed in 2.3 sec
```

```
xlabel('Year')
ylabel('Number of Flights')
title('Number of Flights by Year, 1987 - 2008')
```

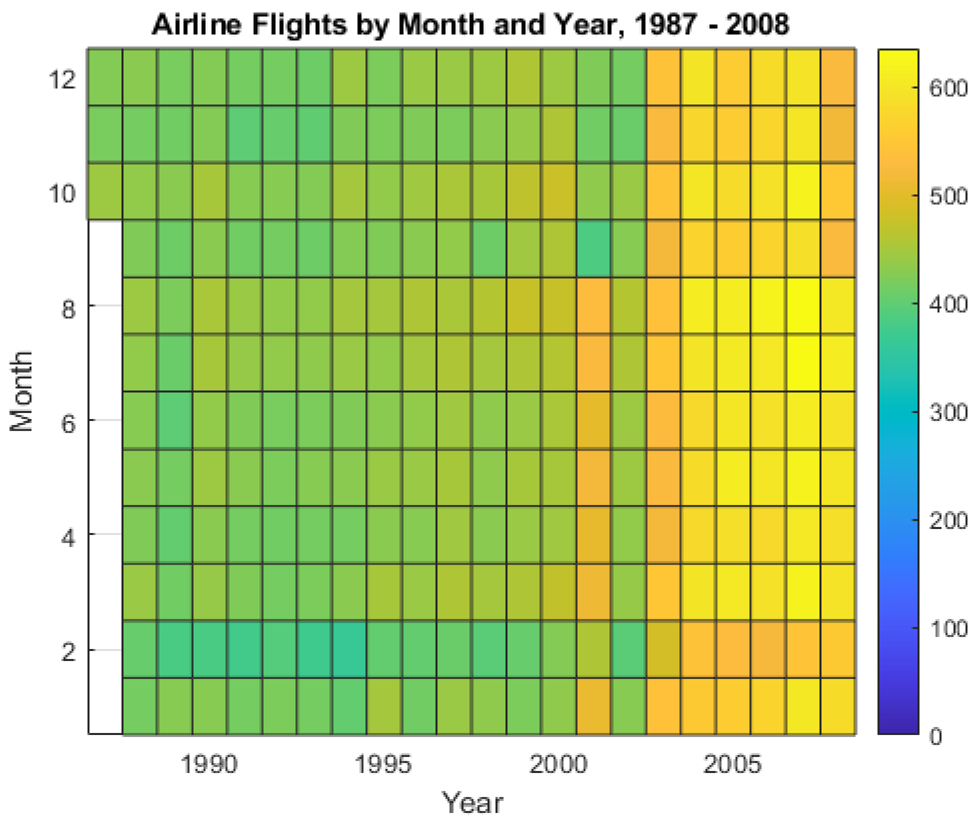


Use `histogram2` to further break down the number of flights by month for the whole data set. Since the bins for `Month` and `Year` are known ahead of time, specify the bin edges to avoid an extra pass through the data.

```
year_edges = 1986.5:2008.5;
month_edges = 0.5:12.5;
histogram2(tt.Year,tt.Month,year_edges,month_edges,'DisplayStyle','tile')
```

```
Evaluating tall expression using the Local MATLAB Session:
- Pass 1 of 1: Completed in 0.99 sec
Evaluation completed in 1.1 sec
```

```
colorbar
xlabel('Year')
ylabel('Month')
title('Airline Flights by Month and Year, 1987 - 2008')
```



Data Analytics and Machine Learning with Tall Arrays

You can perform more sophisticated statistical analysis on tall arrays, including calculating predictive analytics and performing machine learning, using the functions in Statistics and Machine Learning Toolbox™.

For more information, see "Analysis of Big Data with Tall Arrays" (Statistics and Machine Learning Toolbox).

Scale to Big Data Systems

A key capability of tall arrays in MATLAB is the connectivity to big data platforms, such as computing clusters and Apache Spark™.

This example only scratches the surface of what is possible with tall arrays for big data. See "Extend Tall Arrays with Other Products" on page 12-175 for more information about using:

- Statistics and Machine Learning Toolbox™
- Database Toolbox™
- Parallel Computing Toolbox™

- MATLAB® Parallel Server™
- MATLAB Compiler™

See Also

`tall`

More About

- “Tall Arrays for Out-of-Memory Data” on page 12-136

Develop Custom Tall Array Algorithms

Tall arrays are a powerful, intuitive way to work with large data sets using traditional MATLAB syntax. However, since tall arrays operate on blocks of the data, each of which individually fits into memory, the traditional algorithms of most functions need to be updated to use a parallelized approach to support tall arrays. This topic shows you how to develop your own parallelized algorithms to operate on tall arrays.

Currently available approaches for applying custom functions to tall arrays are:

- "Single-Step Transformation Operation" on page 12-188: Apply a function to the blocks of data in a tall array.
- "Two-Step Reduction Operation" on page 12-190: Apply a function to a tall array to transform the contents, and then apply another function to reduce the output to a single block.
- "Sliding-Window Operations" on page 12-193: Apply a moving-window function to a tall array to transform the contents.

Regardless of which operation you choose, there are options, performance considerations, and common issues that apply to all approaches.

Reasons to Implement Custom Algorithms

Most common mathematical functions and MATLAB operations already support tall arrays. If the functionality is already supported, then writing your own algorithm might not be necessary.

Here are some reasons why you might want to implement a custom algorithm for tall arrays:

- **Implement Currently Unsupported Functions** — If a particular function does not currently support tall arrays, then you can use the APIs outlined here to write a version of that function that supports tall arrays.
- **Leverage Existing Code** — If you have existing code that performs some operations on in-memory data, then with only minor modifications you can make it compatible to operate on tall arrays. This approach avoids the need to convert the code to fit the subset of the MATLAB language that supports tall arrays.
- **Gain Performance** — For example, you can rewrite a MATLAB function as a C++ MEX function, and then you can use the APIs outlined here to call the MEX function to operate on the data.
- **Use a Preferred External Library** — For compatibility within your organization it is sometimes required to use a specific external library for certain calculations. You can use the APIs outlined here to reimplement a function with those external libraries.

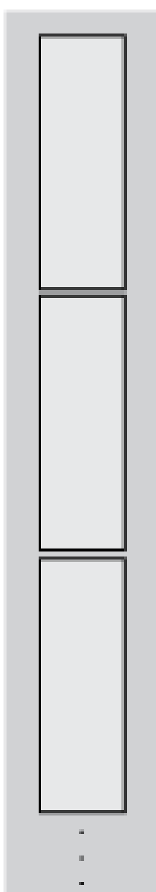
Supported APIs

The supported APIs are intended for advanced use and do not include extensive input checking. Expect to spend some time testing that the supplemental functions you implement satisfy all of the requirements and perform the calculations you expect. Currently supported APIs for authoring tall array algorithms are listed here.

Package Function Name	Description
`matlab.tall.transform`	Apply a specified function to each block of one or more tall arrays.
`matlab.tall.reduce`	Apply a specified function to each block of one or more tall arrays. Then feed the output of that function into a second reduction function.
`matlab.tall.movingWindow`	Apply moving window function to blocks of data.
`matlab.tall.blockMovingWindow`	Apply moving window function and block reduction to padded blocks of data.

Background: Tall Array Blocks

When you create a tall array from a datastore, the underlying datastore facilitates the movement of data during a calculation. The data moves in discrete pieces called *blocks*, where each block is a set of consecutive rows that can fit in memory. For example, one block of a 2-D array (such as a table) is `X(n:m,:)`. The size of each block is based on the value of the `ReadSize` property of the datastore, but the block is not always that exact size. For the purposes of developing tall array algorithms, a tall array is considered to be the vertical concatenation of many such blocks.

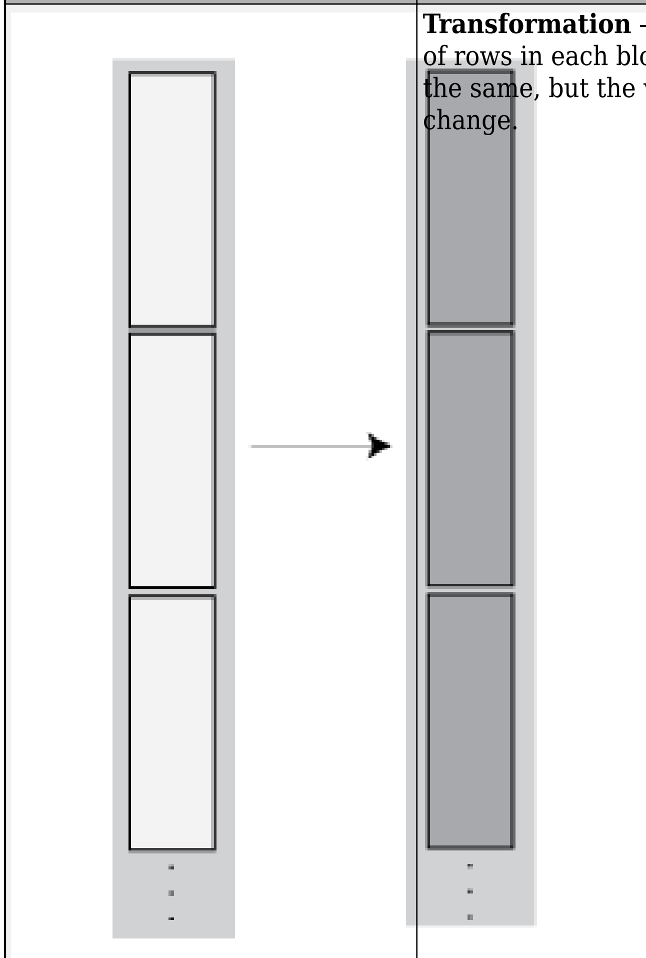


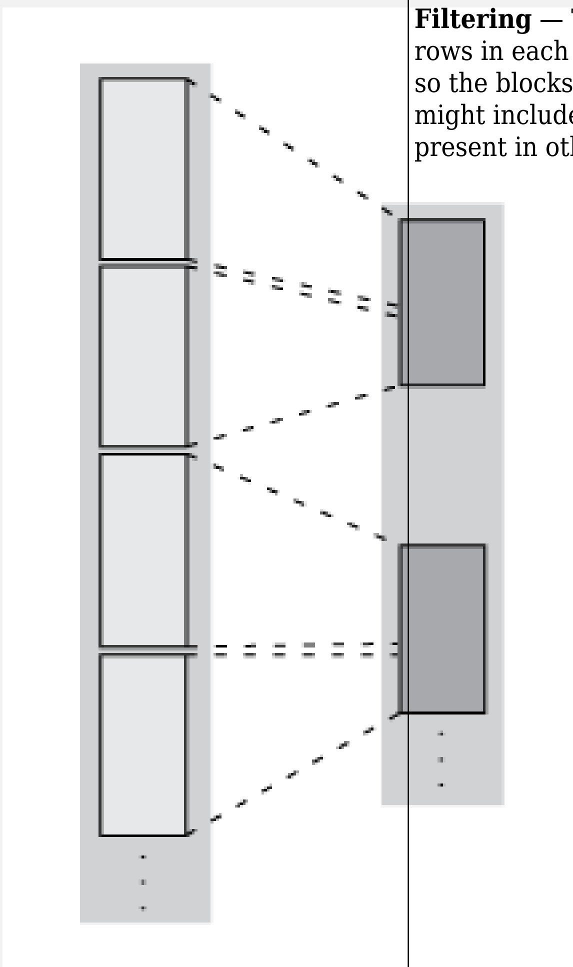
The blocks of a given array are chosen at run-time based on available memory, so they can be dynamic. Therefore, the blocks might not be *exactly* the same size between runs. If you have changes on your computer that affect the available memory, then that can impact the size of the blocks.

Although this page refers only to *blocks* and *rows* in a 2-D sense, these concepts extend to N-D tall arrays. The block size is only constrained in the first dimension, so the block includes all elements in other dimensions; for example, `X(n:m,:,:,...)`. Also, rather than rows, N-D arrays have *slices* such as `X(p,:,:,...)`.

Single-Step Transformation Operation

The `matlab.tall.transform` function applies a single function to each block of a tall array, so you can use it to apply a block-wise transformation, filtering, or reduction of the data. For example, you can remove rows with specific values, center and scale the data, or detect certain conditions and transform specific pieces of data. These figures show what happens to the blocks in an array when they are operated on by `matlab.tall.transform`.

Operation	Description	Examples
	**Transformation** — The number of rows in each block remains the same, but the values change.	• `A = matlab.tall.transform(@sin, tX)` calculates the sine of the elements in each block. • `A = matlab.tall.transform(@(X) X.^2, tX)` squares the elements in each block.

Operation	Description	Examples
	**Filtering** — The number of rows in each block are reduced, so the blocks in the new array might include rows originally present in other blocks.	• `A = matlab.tall.transform(@(X) topkrows(X,5), tX)` extracts only the top 5 rows from each block, filtering out the other rows. • `A = matlab.tall.transform(@sum, tX)` calculates the sum of the elements in each block, which reduces each block to a scalar. The number of elements in `A` is equal to the number of blocks.

Transform Syntax

The generic syntax to apply a single-step transform is

```
[tA, tB, tC, ...] = matlab.tall.transform(fcn, tX, tY, tZ, ...)
```

Functional Requirements for fcn

The general functional signature of `fcn` is

```
[a, b, c, ...] = fcn(x, y, z, ...)
```

`fcn` must satisfy these requirements:

1. **Input Arguments** — The inputs `[x, y, z, ...]` are blocks of data that fit in memory. The blocks are produced by extracting data from the respective tall array inputs `[tX, tY, tZ, ...]`. The inputs `[x, y, z, ...]` satisfy these properties:

   - All of `[x, y, z, ...]` have the same size in the first dimension after any allowed expansion.

- The blocks of data in `[x, y, z, ...]` come from the same index in the tall dimension, assuming the tall array is nonsingleton in the tall dimension. For example, if `tX` and `tY` are nonsingleton in the tall dimension, then the first set of blocks might be `x = tX(1:20000,:)` and `y = tY(1:20000,:)`.
- If the first dimension of any of `[tX, tY, tZ, ...]` has a size of `1`, then the corresponding block `[x, y, z, ...]` consists of all the data in that tall array.

2. **Output Arguments** — The outputs `[a, b, c, ...]` are blocks that fit in memory, to be sent to the respective outputs `[tA, tB, tC, ...]`. The outputs `[a, b, c, ...]` satisfy these properties:

   - All of `[a, b, c, ...]` must have the same size in the first dimension.
   - All of `[a, b, c, ...]` are vertically concatenated with the respective results of previous calls to `fcn`.
   - All of `[a, b, c, ...]` are sent to the same index in the first dimension in their respective destination output arrays.

3. **Functional Rules** — `fcn` must satisfy the functional rule:

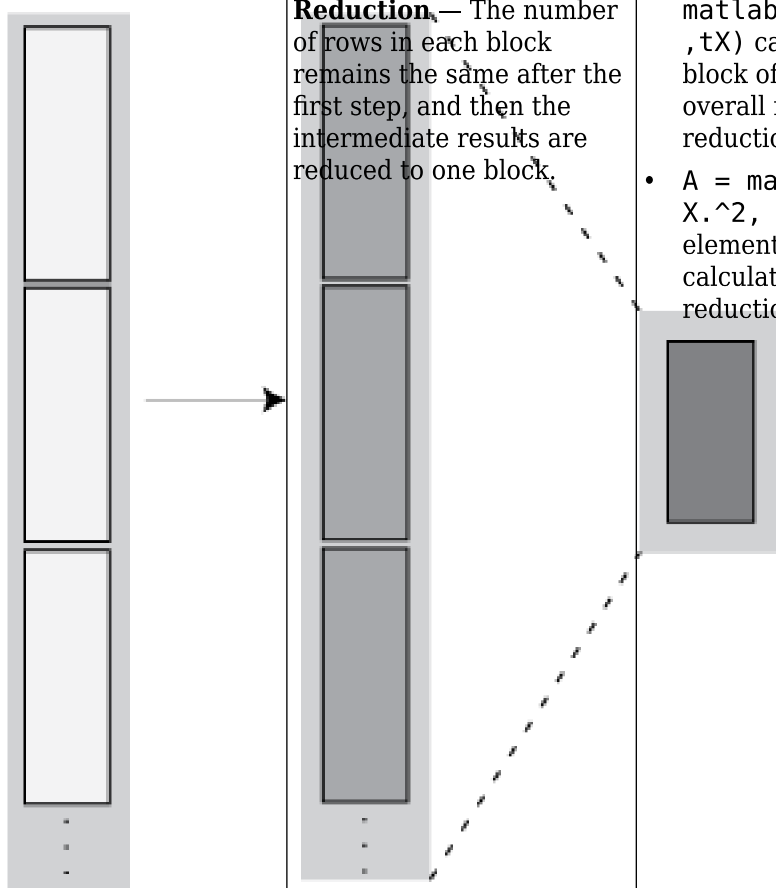
   - `F([inputs1; inputs2]) == [F(inputs1); F(inputs2)]`: Applying the function to the concatenation of the inputs should be the same as applying the function to the inputs separately and then concatenating the results.

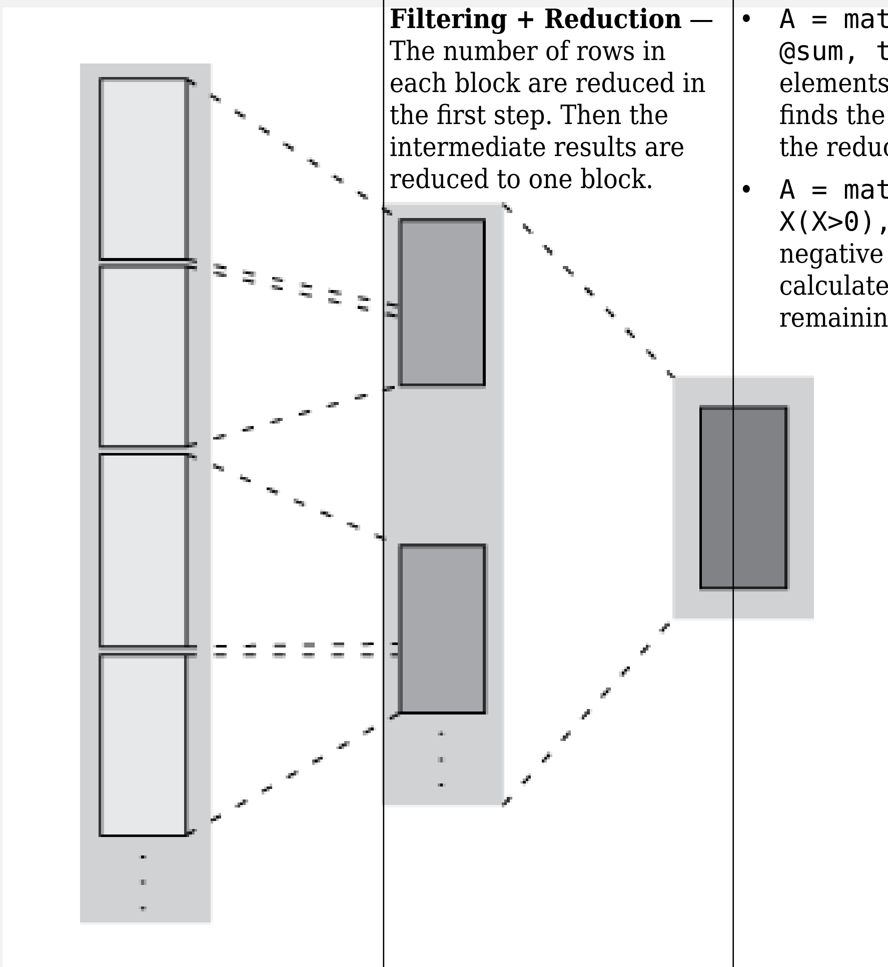
4. **Empty Inputs** — Ensure that `fcn` can handle an input that has a height of 0. Empty inputs can occur when a file is empty or if you have done a lot of filtering on the data.

Two-Step Reduction Operation

`matlab.tall.reduce` applies two functions to a tall array, with the result of the first step being fed in as input to a final reduction step. The reduction function is applied repeatedly to the intermediate results until a single final block that fits in memory is obtained. In the MapReduce paradigm, this process is similar to a "single key" MapReduce operation, where the intermediate results all have the same key and are combined in the reduction step.

The first step is similar to `matlab.tall.transform` and has the same requirements. However, the reduction step always reduces the intermediate results down to a single block that fits in memory. These figures show what happens to the blocks in an array when they are operated on by `matlab.tall.reduce`.

Operation	Description	Examples
	**Transformation + Reduction** — The number of rows in each block remains the same after the first step, and then the intermediate results are reduced to one block.	• `A = matlab.tall.reduce(@sin,@max,tX)` calculates the sine of each block of values, and then it finds the overall maximum value during the reduction step. • `A = matlab.tall.reduce(@(X) X.^2, @mean, tX)` squares the elements in each block, and then it calculates the overall mean in the reduction step.

Operation	Description	Examples
	**Filtering + Reduction** — The number of rows in each block are reduced in the first step. Then the intermediate results are reduced to one block.	• `A = matlab.tall.reduce(@sum, @sum, tX)` calculates the sum of the elements in each block, and then it finds the overall sum of elements in the reduction step. • `A = matlab.tall.reduce(@(X) X(X>0), @mean, tX)` filters out all negative values, and then it calculates the overall mean of the remaining values.

Reduce Syntax

The generic syntax to apply a two-step reduction is

```
[rA, rB, rC, ...] = matlab.tall.reduce(fcn, reducefcn, tX, tY, tZ, ...)
```

The functional signature of `fcn` is

```
[a, b, c, ...] = fcn(x, y, z, ...)
```

The functional signature of `reducefcn` is

```
[rA, rB, rC, ...] = reducefcn(a, b, c, ...)
```

That is, the input tall arrays `[tX, tY, tZ, ...]` are broken into blocks `[x, y, z, ...]` that are inputs to `fcn`. Then, `fcn` returns outputs `[a, b, c, ...]` that are inputs to `reducefcn`. Finally, `reducefcn` returns the final results `[rA, rB, rC]` that are returned by `matlab.tall.reduce`.

Functional Requirements for reducefcn

The requirements for `fcn` are the same as those that were outlined in “Functional Requirements for fcn” on page 12-189. However, the requirements for `reducefcn` are different.

The general functional signature of `reducefcn` is

```
[rA, rB, rC, ...] = reducefcn(a, b, c, ...)
```

`reducefcn` must satisfy these requirements:

1. **Input Arguments** — The inputs `[a, b, c, ...]` are blocks that fit in memory. The blocks of data are either outputs returned by `fcn`, or a partially reduced output from `reducefcn` that is being operated on again for further reduction. The inputs `[a, b, c, ...]` satisfy these properties:

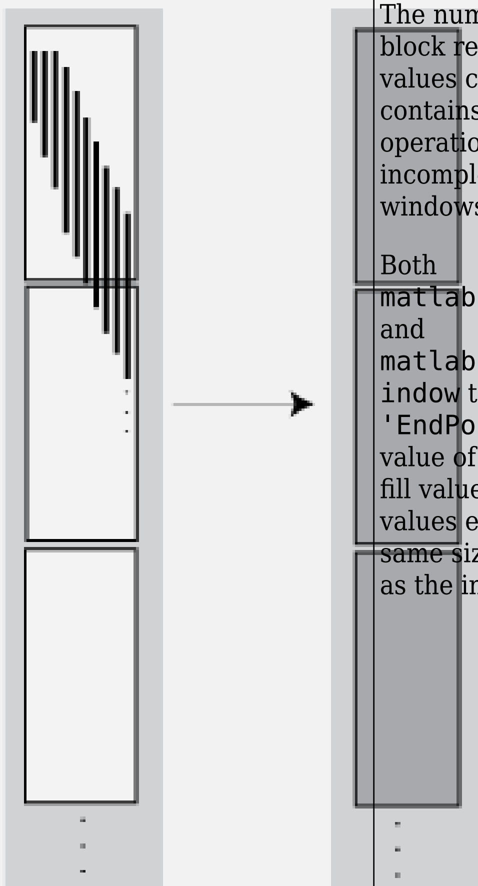
   - The inputs `[a, b, c, ...]` have the same size in the first dimension.
   - For a given index in the first dimension, every row of the blocks of data `[a, b, c, ...]` either originates from the input, or originates from the same previous call to `reducefcn`.
   - For a given index in the first dimension, every row of the inputs `[a, b, c, ...]` for that index originates from the same index in the first dimension.
2. **Output Arguments** — All outputs `[rA, rB, rC, ...]` must have the same size in the first dimension. Additionally, they must be vertically concatenable with the respective inputs `[a, b, c, ...]` to allow for repeated reductions when necessary.
3. **Functional Rules** — `reducefcn` must satisfy these functional rules (up to roundoff error):

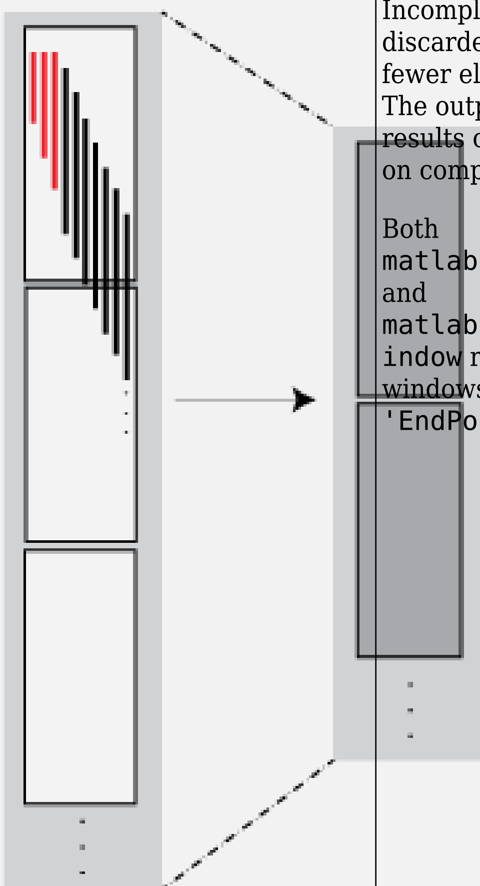
   - `F(input) == F(F(input))`: Applying the function repeatedly to the same inputs should not change the result.
   - `F([input1; input2]) == F([input2; input1])`: The result should not depend on the order of concatenation.
   - `F([input1; input2]) == F([F(input1); F(input2)])`: Applying the function once to the concatenation of some intermediate results should be the same as applying it separately, concatenating, and applying it again.
4. **Empty Inputs** — Ensure that `reducefcn` can handle an input that has a height of 0. Empty inputs can occur when a file is empty or if you have done a lot of filtering on the data. For this call, all input blocks are empty arrays of the correct type and size in dimensions beyond the first.

Sliding-Window Operations

The `matlab.tall.movingWindow` and `matlab.tall.blockMovingWindow` functions apply a function to windows of data within a tall array. While `matlab.tall.transform` and `matlab.tall.reduce` operate on entire blocks of data at a time, the moving-window functions operate on windows of data as a window moves from the beginning to the end of the array. The windows can span between the blocks of data being read from disk.

These figures show what happens to the blocks in an array when they are operated on by `matlab.tall.movingWindow` or `matlab.tall.blockMovingWindow`.

Operation	Description	Examples
	**Windowed Transformation** — The number of rows in each block remains the same, but the values change. The output contains the results of operations performed on both incomplete and complete windows of data. Both `matlab.tall.movingWindow` and `matlab.tall.blockMovingWindow` transform data when `'EndPoints'` is the default value of `'shrink'`, or when a fill value is specified. Both values ensure the output is the same size in the first dimension as the input.	• `A = matlab.tall.movingWindow(@mean, 100, tX)` calculates a moving mean using a window size of 100.

Operation	Description	Examples
	**Windowed Filtering** — Incomplete windows of data are discarded, so the output has fewer elements than the input. The output only contains the results of operations performed on complete windows of data. Both `matlab.tall.movingWindow` and `matlab.tall.blockMovingWindow` remove incomplete windows of data when `'EndPoints'` is `'discard'`.	• `A = matlab.tall.movingWindow(@mean, 100, tX, 'EndPoints', 'discard')` calculates a moving mean on complete windows of data, using a window size of 100.

You can use `matlab.tall.movingWindow` and `matlab.tall.blockMovingWindow` to apply windowed transformations or filters to data. For example, you can calculate a trailing average or a moving median, or you can apply several operations at once to the same window. The two functions differ in these ways:

- `matlab.tall.movingWindow` applies `fcn` to all windows of data, regardless of whether the windows are complete. `matlab.tall.blockMovingWindow` applies `windowfcn` to incomplete windows of data, and applies `blockfcn` to complete windows of data.
- `matlab.tall.movingWindow` operates on single windows of data at a time. `matlab.tall.blockMovingWindow` operates on entire blocks of data containing multiple complete windows, which reduces the number of function calls required in the calculation.

Moving Window Syntaxes

The syntax to apply a moving window operation to single windows of data is

```
[tA, tB, tC, ...] = matlab.tall.movingWindow(fcn, window, tX, tY, tZ, ...)
```

The functional signature of `fcn` must be

```
[a, b, c, ...] = fcn(x, y, z, ...)
```

Similarly, the syntax to apply a moving window operation to entire blocks of data is

```
[tA, tB, tC, ...] = matlab.tall.blockMovingWindow(windowfcn, blockfcn, window, tX, tY, tZ, ...)
```

The functional signatures of `windowfcn` and `blockfcn` must be

```
[a, b, c, ...] = windowfcn(info, x, y, z, ...)
[a, b, c, ...] = blockfcn(info, bX, bY, bZ, ...)
```

The `info` input is a structure that contains the fields `Window` and `Stride`. When you write the function. use these fields to pick out windows of data in each block.

For an outline of general rules that `fcn`, `windowfcn`, and `blockfcn` must follow, see “Functional Requirements for fcn” on page 12-189. Aside from the `info` input, `fcn` and `windowfcn` have the same requirements. However, the requirements for `blockfcn` are different since that function operates on entire blocks of data.

Functional Requirements for windowfcn

The general functional signature of `windowfcn` is

```
[a, b, c, ...] = windowfcn(info, x, y, ...)
```

The `info` input is a structure provided by `matlab.tall.blockMovingWindow` that includes these fields:

- `Stride` — Specified step size between windows (default: 1). Set this value with the `'Stride'` name-value pair.
- `Window` — Specified window size. Set this value with the `window` input argument.

`windowfcn` must satisfy these requirements:

1. **Input Arguments** — The inputs `[x, y, z, ...]` are blocks of data that fit in memory. The blocks are produced by extracting data from the respective tall array inputs `[tX, tY, tZ, ...]`. The inputs `[x, y, z, ...]` satisfy these properties:

   - All of the inputs `[x, y, z, ...]` have the same size in the first dimension.
   - The blocks of data in `[x, y, z, ...]` come from the same index in the tall dimension, assuming the tall array is nonsingleton in the tall dimension. For example, if `tX` and `tY` are nonsingleton in the tall dimension, then the first set of blocks might be `x = tX(1:20000,:)` and `y = tY(1:20000,:)`.
   - When the first dimension of any of `[tX, tY, tZ, ...]` has a size of `1`, the corresponding block `[x, y, z, ...]` consists of all the data in that tall array.
   - Applying `windowfcn` must result in a reduction of the input data to a scalar or a slice of an array of height 1.

     When the input is a matrix, N-D array, table, or timetable, applying `windowfcn` must result in a reduction of the input data in each of its columns or variables.
2. **Output Arguments** — The outputs `[a, b, c, ...]` are blocks that fit in memory to be sent to the respective outputs `[tA, tB, tC, ...]`. The outputs `[a, b, c, ...]` satisfy these properties:

   - All of the outputs `[a, b, c, ...]` must have the same size in the first dimension.

- All of the outputs `[a, b, c, ...]` are vertically concatenated with the respective results of previous calls to `windowfcn`.
- All of the outputs `[a, b, c, ...]` are sent to the same index in the first dimension in their respective destination output arrays.

3 **Functional Rules** — `windowfcn` must satisfy this functional rule:

- `F([inputs1; inputs2]) == [F(inputs1); F(inputs2)]`: Applying the function to the concatenation of the inputs should be the same as applying the function to the inputs separately and then concatenating the results.

Functional Requirements for blockfcn

The general functional signature of `blockfcn` is

```
[a, b, c, ...] = blockfcn(info, bX, bY, bZ, ...)
```

The `info` input is a structure provided by `matlab.tall.blockMovingWindow` that includes these fields:

- `Stride` — Specified step size between windows (default: 1). Set this value with the `'Stride'` name-value pair.
- `Window` — Specified window size. Set this value with the `window` input argument.

The blocks of data `bX, bY, bZ, ...` that `matlab.tall.blockMovingWindow` provides to `blockfcn` have these properties:

- The blocks contain only full-sized windows. `blockfcn` does not have to define a behavior for incomplete windows of data.
- The first window of data starts at the first element of the block. The last element of the last window is the last element of the block.

`blockfcn` must satisfy these requirements:

1 **Input Arguments** — The inputs `[bX, bY, bZ, ...]` are blocks of data that fit in memory. The blocks are produced by extracting data from the respective tall array inputs `[tX, tY, tZ, ...]`. The inputs `[bX, bY, bZ, ...]` satisfy these properties:

- All of the inputs `[bX, bY, bZ, ...]` have the same size in the first dimension after any allowed expansion.
- The blocks of data in `[bX, bY, bZ, ...]` come from the same index in the tall dimension, assuming the tall array is nonsingleton in the tall dimension. For example, if `tX` and `tY` are nonsingleton in the tall dimension, then the first set of blocks might be `bX = tX(1:20000,:)` and `bY = tY(1:20000,:)`.
- If the first dimension of any of the data inputs `[tX, tY, tZ, ...]` has a size of `1`, then the corresponding block `[bX, bY, bZ, ...]` consists of all the data in that tall array.
- Applying `blockfcn` must result in a reduction of the input data such that the result has height equal to the number of windows in the block. You can use `info.Window` and `info.Stride` to determine the number of windows in a block.

  If the input is a matrix, N-D array, table, or timetable, then applying `blockfcn` must result in a reduction of the input data in each of its columns or variables.

2 **Output Arguments** — The outputs [a, b, c, ...] are blocks that fit in memory, to be sent to the respective outputs [tA, tB, tC, ...]. The outputs [a, b, c, ...] satisfy these properties:

- All of the outputs [a, b, c, ...] must have the same size in the first dimension.
- All of the outputs [a, b, c, ...] are vertically concatenated with the respective results of previous calls to blockfcn.
- All of the outputs [a, b, c, ...] are sent to the same index in the first dimension in their respective destination output arrays.

3 **Functional Rules** — blockfcn must satisfy this functional rule:

- F([inputs1; inputs2]) == [F(inputs1); F(inputs2)]: Applying the function to the concatenation of the inputs should be the same as applying the function to the inputs separately and then concatenating the results.

Control Output Data Type

If the final output from any of the “Supported APIs” on page 12-186 has a different data type from the input, then you *must* specify the 'OutputsLike' name-value pair to provide one or more prototype arrays that have the same data type and attributes as the corresponding outputs. The value of 'OutputsLike' is always a cell array, with each cell containing a prototype array for the corresponding output argument.

For example, this call to matlab.tall.transform accepts one tall array tX as an input and returns two outputs with different types specified by the prototype arrays protoA and protoB. Output A has the same data type and attributes as protoA, and likewise for B and protoB.

```
C = {protoA protoB};
[A, B] = matlab.tall.transform(fcn, tX, 'OutputsLike', C)
```

A common way to supply the prototype arrays is to call fcn with trivial inputs of the proper data type, since the outputs returned by fcn have the correct data type. In this example, the transform function accepts a tall double, but returns a tall table. A prototype array is generated by calling fcn(0) and the prototype is specified as the value of 'OutputsLike'.

```
ds = tabularTextDatastore('airlinesmall.csv','TreatAsMissing','NA');
ds.SelectedVariableNames = {'ArrDelay', 'DepDelay'};
tt = tall(ds);
tX = tt.ArrDelay;

fcn = @(x) table(x,'VariableNames',{'MyVar'});
proto_A = fcn(0);
A = matlab.tall.transform(fcn,tX,'OutputsLike',{proto_A});
```

Coding and Performance Tips

- Put all analytics in a single function that you call to operate directly on the data, instead of using unnecessary nested functions.
- Experiment using a small subset of the data. Profile your code to find and fix bottlenecks before scaling up to the entire data set, where bottlenecks can be greatly amplified.
- Pay attention to the orientation of your data, since some functions return the outputs in different shapes depending on the input data. For example, unique can return either a row vector or a column vector depending on the orientation of the input data.

- Blocks are dynamically generated at run-time based on available computer memory. Make sure that any specified reduction function obeys the function rule `F([input1; input2]) == F([F(input1); F(input2)])`. If this rule is not obeyed, then the results can differ significantly between trials.
- Blocks can have any size in the first dimension, including 0 or 1. Size 0 or 1 can occur in intermediate calculations as a result of filtering or reduction operations. Make sure your function does the correct thing for both of these cases. One sign that the function does not handle these cases properly is when you receive an "Output is different size" error message.

See Also

`matlab.tall.blockMovingWindow` | `matlab.tall.movingWindow` | `matlab.tall.reduce` | `matlab.tall.transform`

More About

- “Index and View Tall Array Elements” on page 12-147
- “Visualization of Tall Arrays” on page 12-161
- “Extend Tall Arrays with Other Products” on page 12-175

TCP/IP Support in MATLAB

TCP/IP Communication Overview

Transmission Control Protocol (TCP) is a transport protocol layered on top of the Internet Protocol (IP) and is one of the most used networking protocols. The MATLAB TCP/IP client support uses raw socket communication and lets you connect to remote hosts from MATLAB for reading and writing data. For example, you could use it to acquire data from a remote weather station, and plot the data.

- **Connection based protocol** — The two ends of the communication link must be connected at all times during the communication.
- **Streaming protocol** — TCP/IP has a long stream of data that is transmitted from one end of the connection to the other end, and another long stream of data flowing in the opposite direction. The TCP/IP stack at one end is responsible for breaking the stream of data into packets and sending those packets, while the stack at the other end is responsible for reassembling the packets into a data stream using information in the packet headers.
- **Reliable protocol** — The packets sent by TCP/IP contain a unique sequence number. The starting sequence number is communicated to the other side at the beginning of communication. The receiver acknowledges each packet, and the acknowledgment contains the sequence number so that the sender knows which packet was acknowledged. This method implies that any packets lost on the way can be retransmitted because the sender would know that packets did not reach their destination because it had not received an acknowledgment. Also, packets that arrive out of sequence can be reassembled in the proper order by the receiver.

  Timeouts can be established because the sender knows (from the first few packets) how long it takes on average for a packet to be sent and its acknowledgment received.

You can create a TCP/IP connection to a server or hardware and perform read/write operations. Use the `tcpclient` function to create the connection, and the `write` and `read` functions for synchronously reading and writing data.

See “Create a TCP/IP Connection” on page 13-3 to get started, and “Write and Read Data over TCP/IP Interface” on page 13-7 for examples of reading and writing data.

Create a TCP/IP Connection

The MATLAB TCP/IP client support lets you connect to remote hosts or hardware from MATLAB for reading and writing data. The typical workflow is:

- Create a TCP/IP connection to a server or hardware.
- Configure the connection if necessary.
- Perform read and write operations.
- Clear and close the connection.

To communicate over the TCP/IP interface, you first create a TCP/IP object using the `tcpclient` function. The syntax is:

```
<objname> = tcpclient(Address, Port)
```

The address can be either a remote host name or a remote IP address. In both cases, the `Port` must be a positive integer between `1` and `65535`.

Create Object Using Host Name

This example creates the TCP/IP object `t` using the host address shown and `port` of `80`.

```
t = tcpclient('www.mathworks.com', 80)

t = 

   tcpclient with properties:

          Address: 'www.mathworks.com'
             Port: 80
          Timeout: 10
   BytesAvailable: 0
   ConnectTimeout: Inf
```

Note When connecting using a host name, such as a specified web address or `'localhost'`, the IP address will be resolved according to the configuration of your network interface. This may result in an IPv4 address or an IPv6 address. If your TCP/IP server expects the incoming connections to be of a certain type of address, for example IPv4 address only, you may be required to use the explicit IP address, instead of the host name, when creating the client.

Create Object Using IP Address

This example creates the TCP/IP object `t`using the IP address shown and `port` of `4012`.

```
t = tcpclient('172.28.154.231', 4012)

t = 

   tcpclient with properties:

          Address: '172.28.154.231'
             Port: 4012
          Timeout: 10
```

```
   BytesAvailable: 0
   ConnectTimeout: Inf
```

Set the Timeout Property

You can create the object using a name-value pair to set the `Timeout` value. The `Timeout` property specifies the waiting time to complete read and write operations in seconds, and the default is `10`. You can change the value either during object creation or after you create the object.

This example creates a TCP/IP object, but increases the `Timeout` to `20` seconds.

```
t = tcpclient('172.28.154.231', 4012, 'Timeout', 20)

t =

   tcpclient with properties:

          Address: '172.28.154.231'
             Port: 4012
          Timeout: 20
   BytesAvailable: 0
   ConnectTimeout: Inf
```

The output reflects the `Timeout` property change.

Set the Connect Timeout Property

You can create the object using a name-value pair to set the `ConnectTimeout` value. The `ConnectTimeout` property specifies the maximum time in seconds to wait for a connection request to the specified remote host to succeed or fail. The value must be greater than or equal to 1. If not specified, the default value of `ConnectionTimeout` is `Inf`. You can change the value only during object creation.

This example creates a TCP/IP object, but specifies the `ConnectTimeout` as `10` seconds.

```
t = tcpclient('172.28.154.231', 4012, 'ConnectTimeout', 10)

t =

   tcpclient with properties:

          Address: '172.28.154.231'
             Port: 4012
          Timeout: 10
   BytesAvailable: 0
   ConnectTimeout: 10
```

The output reflects the `ConnectTimeout` property change.

Note If an invalid address or port is specified or the connection to the server cannot be established, the object is not created.

Configure Properties for TCP/IP Communication

The `tcpclient` object has the following properties.

Property	Description
`Address`	Remote host name or IP address for connection. Specify address as the first argument when you create the `tcpclient` object. In this example `Address` is `'172.28.154.231'`. `t = tcpclient('172.28.154.231', 4012)`
`Port`	Remote host port for connection. Specify port number as the second argument when you create the `tcpclient` object. The `Port` must be a positive integer between `1` and `65535`. In this example `Port` is `4012`. `t = tcpclient('www.mathworks.com', 4012)`
`BytesAvailable`	Read-only property that returns the number of bytes available in the input buffer.
`Timeout`	Waiting time in seconds to complete read and write operations, specified as a positive value of type `double`. The default is `10`. You can change the value either during object creation, or after you create the object.
`ConnectTimeout`	Maximum time in seconds to wait for a connection request to the specified remote host to succeed or fail, specified as a positive value of type `double`. If not specified, the default value is `Inf`. You can change the value only during object creation.

Setting the Timeout

The default value for `Timeout` is `10` seconds. You can change the value either during object creation, or after you create the object.

You can optionally create the `tcpclient` object using a name-value pair to set the `Timeout` value.

This example creates the TCP/IP object and increases the `Timeout` to `20` seconds.

```
t = tcpclient('172.28.154.231', 4012, 'Timeout', 20)

t = 

   tcpclient with properties:

          Address: '172.28.154.231'
             Port: 4012
          Timeout: 20
   BytesAvailable: 0
   ConnectTimeout: Inf
```

The output reflects the `Timeout` property change from the default of `10` seconds to `20` seconds.

You can also change it anytime by setting the property value using this syntax.

```
<object_name>.<property_name> = <property_value>
```

This example using the same object named `t` increases the `Timeout` to 30 seconds.

```
t.Timeout = 30
```

Setting the Connect Timeout

You can create the `tcpclient` object using a name-value pair to set the `ConnectTimeout` value. The `ConnectTimeout` property specifies the maximum time in seconds to wait for a connection request to the specified remote host to succeed or fail. The value must be greater than or equal to 1. If not specified, the default value of `ConnectionTimeout` is `Inf`. You can change the value only during object creation.

This example creates a TCP/IP object, but changes the `ConnectTimeout` to `10` seconds.

```
t = tcpclient('172.28.154.231', 4012, 'ConnectTimeout', 10)

t =

   tcpclient with properties:

          Address: '172.28.154.231'
             Port: 4012
          Timeout: 10
   BytesAvailable: 0
   ConnectTimeout: 10
```

The output reflects the `ConnectTimeout` property change.

Write and Read Data over TCP/IP Interface

In this section...

Write Data

The `write` function synchronously writes data to the remote host connected to the `tcpclient` object. First specify the data, then write the data. The function waits until the specified number of values is written to the remote host.

In this example, a `tcpclient` object `t` already exists.

```
% Create a variable called data
data = 1:10;

% Write the data to the object t
write(t, data)
```

Note For any read or write operation, the data type is converted to `uint8` for the data transfer. It is then converted back to whatever data type you set if you specified another data type.

Read Data

The `read` function synchronously reads data from the remote host connected to the `tcpclient` object and returns the data. There are three read options:

- Read all bytes available (no arguments)
- Optionally specify the number of bytes to read
- Optionally specify the data type

If you do not specify a size, the default read uses the `BytesAvailable` property value, which is equal to the numbers of bytes available in the input buffer.

In these examples, a `tcpclient` object `t` already exists.

```
% Read all bytes available.
read(t)

% Specify the number of bytes to read, 5 in this case.
read(t, 5)

% Specify the number of bytes to read, 10, and the data type, double.
read(t, 10, 'double')
```

Note For any read or write operation, the data type is converted to `uint8` for the data transfer. It is then converted back to whatever data type you set if you specified another data type.

Acquire Data from a Weather Station Server

One of the primary uses of TCP/IP communication is to acquire data from a server. This example shows how to acquire and plot data from a remote weather station.

Note The IP address in this example is not a working IP address. The example shows how to connect to a remote server. You should substitute the address shown here with the IP address or host name of a server you want to communicate with.

---

1 Create the `tcpclient` object using the `Address` shown here and `Port` of `1045`.

```
t = tcpclient('172.28.154.231', 1045)

t =

   tcpclient with properties:

          Address: '172.28.154.231'
             Port: 1045
          Timeout: 10
   BytesAvailable: 0
```

See the note above step 1 about using a valid address.

2 Acquire data using the `read` function. Specify the number of bytes to read as `30`, for 10 samples from 3 sensors (temperature, pressure, and humidity). Specify the data type as `double`.

```
data = read(t, 30, 'double');
```

3 Reshape the 1x30 data into 10x3 data to show one column each for temperature, pressure, and humidity.

```
data = reshape(data, [3, 10]);
```

4 Plot the temperature.

```
subplot(311);
plot(data(:, 1));
```

5 Plot the pressure.

```
subplot(312);
plot(data(:, 2));
```

6 Plot the humidity.

```
subplot(313);
plot(data(:, 3));
```

7 Close the connection between the TCP/IP client object and the remote host by clearing the object.

```
clear t
```

Read and Write uint8 Data

This example shows how to read and write `uint8` data from an echo server.

1 Create the `tcpclient` object using a local host at `Port 7`.

```
t = tcpclient('localhost', 7)

t = 

   tcpclient with properties:

          Address: 'localhost'
             Port: 7
          Timeout: 10
   BytesAvailable: 0
```

2 Assign 10 bytes of `uint8` data to the variable `data`.

```
data = uint8(1:10)

data =

  1    2    3    4    5    6    7    8    9    10
```

3 Check the data.

```
whos data

Name     Size     Bytes     Class     Attributes

data     1x10        10     uint8
```

4 Write the data to the echoserver.

```
write(t, data)
```

5 Check that the data was written using the `BytesAvailable` property.

```
t.BytesAvailable

ans =

    10
```

6 Read the data from the server.

```
read(t)

ans =

  1    2    3    4    5    6    7    8    9    10
```

7 Close the connection by clearing the object.

```
clear t
```

Bluetooth Low Energy Communication

Bluetooth Low Energy Communication Overview

Bluetooth Low Energy is a wireless communication protocol for discovering and communicating with low-power peripheral devices. Bluetooth Low Energy support in MATLAB allows you to:

- Scan for nearby peripheral devices and view their advertisement data.
- Establish a connection between your computer and peripheral devices.
- Read and write characteristic and descriptor data.
- Subscribe to characteristics to enable notification or indication.

Bluetooth Low Energy Communication in MATLAB is different from "Bluetooth Communication" (Instrument Control Toolbox) in Instrument Control Toolbox™ and Bluetooth in Communications Toolbox™. To understand the use cases for each feature and to decide which one best fits your needs, refer to the documentation in their respective toolboxes.

Prerequisites and Setup

To use the Bluetooth Low Energy interface in MATLAB, you need the following:

- A peripheral device that supports Bluetooth 4.0 or higher
- A built-in or external Bluetooth 4.0 adapter on your computer
- MATLAB installed on your computer

Make sure your computer has Bluetooth turned on. The peripheral device that you are scanning for must be near your computer and disconnected from other devices and applications.

Bluetooth Low Energy Concepts

The Bluetooth Low Energy communication is based on the Generic Access Profile (GAP) protocol for device discovery and the Generic Attribute (GATT) protocol for device communication. Both of these protocols are documented in detail on the Bluetooth SIG website. Although specific knowledge about GAP and GATT is not necessary to communicate with Bluetooth Low Energy devices in MATLAB, there are a few important concepts to understand.

During discovery, devices are either central devices or peripheral devices.

- A central device scans for advertisement data from other devices. In the Bluetooth Low Energy interface in MATLAB, your computer is always the central device.
- A peripheral device transmits advertisement data. Peripheral devices are nearby devices that you are scanning for. Peripheral devices include fitness trackers, health monitors, and personal electronics such as smartwatches or wireless headphones.

After discovering the peripheral devices, you can connect to multiple peripheral devices from MATLAB at the same time. For more information about scanning for and connecting to peripheral devices, see "Find Your Bluetooth Low Energy Peripheral Devices" on page 14-4.

After establishing a connection between MATLAB on your computer and a Bluetooth Low Energy peripheral device, the two have a client-server relationship. The server is the device that contains data, while the client is the device that receives the data and performs a function.

Services, Characteristics, and Descriptors

The building blocks of Bluetooth Low Energy device communication are services, characteristics, and descriptors. All three of these play a role in reading from and writing to peripheral devices in MATLAB. In peripheral devices, their functions are as follows:

- Services are collections of related characteristics. Services are not readable or writable, but the characteristics that they contain can be. For example, the Heart Rate service includes both the Heart Rate Measurement and Body Sensor Location characteristics.
- Characteristics contain values for user data or device data. This is the primary attribute of the peripheral device that you want to read. For example, the Heart Rate Measurement characteristic is a standard characteristic commonly found on fitness trackers. You can read or write characteristic values.
- Descriptors describe characteristic values. Each characteristic can have one or more descriptors associated with it. A descriptor contains metadata about the characteristic value being measured. For example, the Client Characteristic Configuration Descriptor is a descriptor that determines whether the device is currently collecting characteristic data or not. You can read or write descriptor values.

All three of these can be standard or custom. Standard services, characteristics, and descriptors are defined by the Bluetooth SIG, while custom services, characteristics, and descriptors are usually specific to the device or device manufacturer.

In the Bluetooth Low Energy interface in MATLAB, you can use `read` and `write` on `characteristic` and `descriptor` objects. You can also use `subscribe` and `unsubscribe` with `characteristic` objects. For more information about how to perform these operations, see “Work with Device Characteristics and Descriptors” on page 14-7.

See Also

`ble` | `blelist` | `characteristic` | `descriptor` | `read` | `subscribe` | `unsubscribe` | `write`

More About

- “Find Your Bluetooth Low Energy Peripheral Devices” on page 14-4
- “Work with Device Characteristics and Descriptors” on page 14-7
- “Collect Data from Fitness Monitoring Devices” on page 14-12
- “Track Orientation of Bluetooth Low Energy Device” on page 14-18
- “Troubleshooting Bluetooth Low Energy” on page 14-23

External Websites

- Bluetooth SIG

Find Your Bluetooth Low Energy Peripheral Devices

You can see a list of nearby Bluetooth Low Energy peripheral devices using `blelist`. After you identify your device on the list, you can establish a connection to it from MATLAB using `ble`.

Scan for Devices

Scan for Bluetooth Low Energy peripheral devices from MATLAB using `blelist`. This function lists all nearby peripheral devices that are advertising data. You can view the advertisement data for each device in MATLAB to determine whether it is connectable. If your device does not appear in the list, make sure it is powered on, nearby, and disconnected from any other devices or applications. Then scan for it again.

```
list = blelist
```

```
list=5×5 table
    Index          Name               Address        RSSI    Advertisement
    _____    __________________    ______________    ____    _____________

      1      "Gear Fit2 (E16A)"    "8498663EE16A"    -54     [1×1 struct]
      2      ""                    "2C4D2724754D"    -69     [1×1 struct]
      3      ""                    "1B75E09FD18F"    -70     [1×1 struct]
      4      ""                    "4F7D6DAF9FCE"    -75     [1×1 struct]
      5      ""                    "7B8ADB5851BD"    -76     [1×1 struct]
```

Devices are sorted by RSSI, which represents signal strength. If there are several devices listed, use the `Name` field to identify your device.

The advertisement data for each device contains data defined in the Generic Access Profile (GAP) by Bluetooth SIG. The `Type` field in the advertisement data `Advertisement` shows the connection status. View advertisement data of the first device.

```
list.Advertisement(1)
```

```
ans = struct with fields:
                            Type: ["Connectable Undirected"    "Scan Response"]
                      Appearance: "Generic Watch"
              ShortenedLocalName: []
               CompleteLocalName: "Gear Fit2 (E16A)"
                    TxPowerLevel: []
    SlaveConnectionIntervalRange: []
        ManufacturerSpecificData: [0 117 1 0 2 0 1 3 0]
                     ServiceData: []
             CompleteServiceUUIDs: []
           IncompleteServiceUUIDs: []
        ServiceSolicitationUUIDs: []
```

Scan for Devices by Name

You can use name-value pair arguments to list only devices with a particular name.

For example, list all peripheral devices with names starting with the string `"UA"`.

```
list = blelist("Name","UA")
```

```
list=2×5 table
    Index          Name                  Address         RSSI    Advertisement
    _____    ___________________    ______________    ____    _____________

      1      "UA E39 MODULE"        "84DD20E39AB6"    -84     [1×1 struct]
      2      "UA Footpod 239AE2"    "0CF3EE239AE2"    -87     [1×1 struct]
```

You can use name-value pair arguments to scan for specific names when you have many peripheral devices nearby.

Scan for Devices by Service Name or UUID

For devices that are advertising services, you can use name-value pair arguments to list only devices with the specified service. Service name and service UUID are both valid.

For example, list all peripheral devices that advertise the `Heart Rate` service.

```
list = blelist("Services","Heart Rate")
```

```
list=1×5 table
    Index          Name                  Address         RSSI    Advertisement
    _____    ___________________    ______________    ____    _____________

      1      "UA E39 MODULE"        "84DD20E39AB6"    -84     [1×1 struct]
```

You can use name-value pair arguments to scan for specific services when you know which services you want to work with.

Scan for Longer Time

You can increase the amount of time MATLAB scans for nearby devices. This is useful if your device has a high advertising interval and does not appear in the list with the default timeout value of three seconds.

```
list = blelist("Timeout",10);
```

Connect to a Device

After you discover your device, establish a connection between MATLAB and the peripheral device using `ble`. Specify the device name or address from the `blelist` output.

```
b = ble("Gear Fit2 (E16A)")
```

```
b = 
  ble with properties:

               Name: "Gear Fit2 (E16A)"
            Address: "8498663EE16A"
          Connected: 1
           Services: [2×2 table]
    Characteristics: [3×5 table]

Show services and characteristics
```

After creating a connection to your device, you can work with the `Characteristics` listed in the `ble` properties. See “Work with Device Characteristics and Descriptors” on page 14-7 for more information and next steps.

See Also

`ble` | `blelist`

More About

- “Work with Device Characteristics and Descriptors” on page 14-7
- “Collect Data from Fitness Monitoring Devices” on page 14-12
- “Track Orientation of Bluetooth Low Energy Device” on page 14-18
- “Troubleshooting Bluetooth Low Energy” on page 14-23

Work with Device Characteristics and Descriptors

Before working with characteristics or descriptors, scan for and create a connection to your Bluetooth Low Energy peripheral device. See “Find Your Bluetooth Low Energy Peripheral Devices” on page 14-4 for more information and instructions. After connecting to your device, you can interface with it by reading or writing the device characteristics and descriptors.

```
b = ble("DemoDev");
```

Access Device Characteristics

View your device characteristics by looking at the `Characteristics` property of the `ble` object.

```
b.Characteristics
```

```
ans=11×5 table
        ServiceName                     ServiceUUID                              CharacteristicN
    ___________________    ______________________________________    ____________________________

    "Generic Access"       "1800"                                    "Device Name"
    "Generic Access"       "1800"                                    "Appearance"
    "Generic Access"       "1800"                                    "Peripheral Preferred Connec
    "Generic Access"       "1800"                                    "Central Address Resolution"
    "Generic Attribute"    "1801"                                    "Service Changed"
    "Heart Rate"           "180D"                                    "Heart Rate Measurement"
    "Heart Rate"           "180D"                                    "Body Sensor Location"
    "Battery Service"      "180F"                                    "Battery Level"
    "User Data"            "181C"                                    "Gender"
    "Custom"               "03B80E5A-EDE8-4B33-A751-6CE34EC4C700"    "Custom"
    "Custom"               "03B80E5A-EDE8-4B33-A751-6CE34EC4C700"    "Custom"
```

This table lists each characteristic and the service it is associated with. As the table shows, each service can contain multiple characteristics. If multiple characteristics have the same name, differentiate between them using the UUID. In this example, the device has both standard and custom characteristics. Standard characteristics are defined by the Bluetooth SIG, while custom characteristics are usually specific to the device or device manufacturer.

The `Attributes` field in this table tells you the read and write permissions for each characteristic. Select a characteristic you are interested in and view its properties using `characteristic`. For example, access the `"Gender"` characteristic using the service and characteristic names.

```
c = characteristic(b,"User Data","Gender")
```

```
c = 
  Characteristic with properties:

             Name: "Gender"
             UUID: "2A8C"
       Attributes: "Read" "Write"
      Descriptors: []
```

This characteristic is both readable and writable.

Read and Write Characteristic Data

Because the "Gender" characteristic is readable and writable, you can write data to it and verify the change in values.

Use read to get the current data. The full behavior of read for a characteristic depends on the Attributes property, as described in characteristicData.

```
data = read(c)
```

```
data = 0
```

Interpret the data by referring to the specification for this characteristic on the Bluetooth SIG website. 0 represents male and 1 represents female. Write 1 to the characteristic to indicate female using write.

```
write(c,1)
```

You can read from the characteristic again to observe the change in the data.

```
data = read(c)
```

```
data = 1
```

Subscribe to Characteristic

You can also subscribe to a characteristic to enable notification or indication on the characteristic. You can only subscribe to characteristics that contain "Notify", "Indicate", or both in the Attributes property. After you enable notification or indication for a characteristic, use read to get the updated data. See characteristicData for a description of the full behavior of read based on Attributes.

For this example, create a characteristic object that represents the "Heart Rate Measurement" characteristic.

```
c = characteristic(b,"Heart Rate","Heart Rate Measurement")
```

```
c = 
  Characteristic with properties:

             Name: "Heart Rate Measurement"
             UUID: "2A37"
       Attributes: "Notify"
      Descriptors: [1x3 table]
 DataAvailableFcn: []

Show descriptors
```

This characteristic supports "Notify".

Start receiving notifications by using subscribe. See type for a description of the full behavior of subscribe based on Attributes.

```
subscribe(c)
```

Read from the characteristic to check that you are receiving data.

```
read(c)
```

```
ans = 1×19

    23    14     1   187     1   186     1   185     1   184     1   183     1   182     1   181
```

Interpret the data by referring to the specification for this characteristic on the Bluetooth SIG website.

After you finish working with the characteristic, disable notifications using `unsubscribe`.

```
unsubscribe(c)
```

Use Callback Function to Read Characteristic Data

You can also create a callback function to read characteristic data as it updates with new data from the device.

Because the `"Heart Rate Measurement"` characteristic supports `"Notify"`, you can create a callback function called `displayCharacteristicData`. Specify the read mode as `'oldest'` instead of `'latest'`. Calling the `'latest'` data can lead to errors in the callback function caused by the flushing of previous data.

```
function displayCharacteristicData(src,evt)
    [data,timestamp] = read(src,'oldest');
    disp(data);
    disp(timestamp);
end
```

Use the `@` operator to assign the function handle to the `DataAvailableFcn` property of the characteristic. When a new notification is available, the callback is called.

```
c.DataAvailableFcn = @displayCharacteristicData
```

```
c = 
  Characteristic with properties:

             Name: "Heart Rate Measurement"
             UUID: "2A37"
       Attributes: "Notify"
      Descriptors: [1x3 table]
 DataAvailableFcn: displayCharacteristicData

Show descriptors
```

After you finish working with the characteristic, disable notifications and reset the callback using `unsubscribe`.

```
unsubscribe(c)
c.DataAvailableFcn = [];
```

Access Device Descriptors

If a characteristic has descriptors, you can access the descriptors to read from or write to them. View the descriptors for a characteristic by looking at the `Descriptors` property of the `characteristic` object.

For this example, show the `Descriptors` for the `"Heart Rate Measurement"` characteristic.

```
c.Descriptors
```

```
ans=1×3 table
               DescriptorName                 DescriptorUUID     Attributes
    ______________________________________    ______________    ____________

    "Client Characteristic Configuration"         "2902"        {1×2 string}
```

Access the `"Client Characteristic Configuration"` descriptor.

```
d = descriptor(c,"Client Characteristic Configuration")
```

```
d =
  Descriptor with properties:

          Name: "Client Characteristic Configuration"
          UUID: "2902"
    Attributes: ["Read"    "Write"]
```

This descriptor is both readable and writable.

Read and Write Descriptor Data

The `"Client Characteristic Configuration"` descriptor contains information about whether notification or indication is enabled or disabled. You can use `read` to get the current data.

```
data = read(d)
```

```
data = 1×2

     0     0
```

Interpret this data by referring to the specification for this descriptor on the Bluetooth SIG website.

This value changes when the notification or indication status changes. For example, write to this value to enable notification for the `"Heart Rate Measurement"` characteristic using `write`. Then, observe the change in values by reading the descriptor again.

```
write(d,[1 0])
data = read(d)
```

```
data = 1×2

     1     0
```

See Also

`characteristic` | `descriptor` | `read` | `subscribe` | `unsubscribe` | `write`

More About

- "Find Your Bluetooth Low Energy Peripheral Devices" on page 14-4
- "Collect Data from Fitness Monitoring Devices" on page 14-12
- "Track Orientation of Bluetooth Low Energy Device" on page 14-18
- "Troubleshooting Bluetooth Low Energy" on page 14-23

Collect Data from Fitness Monitoring Devices

This example shows how to collect and plot data from fitness monitoring devices using Bluetooth® Low Energy communication.

Hardware Setup

This example uses an Under Armour® heart rate monitor belt and a pair of Under Armour smart running shoes. Both devices support Bluetooth Low Energy communication.

Discover and Connect to Devices

First, check that the Bluetooth Low Energy devices support connections by finding them in MATLAB. The `blelist` function scans nearby Bluetooth Low Energy peripheral devices that are advertising.

```
blelist
```

```
ans=14×5 table
    Index          Name                          Address                      RSSI    Advertiseme
    _____    ___________________    ______________________________________    ____    ___________

      1      ""                     "21996E3A-8F31-496D-B332-79D03B759BC7"    -65     [1×1 struct
      2      ""                     "12377421-4EA3-4F77-9365-EFD580C34DE9"    -66     [1×1 struct
      3      "UA E39 MODULE"        "8206F662-BA4A-483F-A908-516FF506BFB0"    -67     [1×1 struct
      4      ""                     "CF65AD36-2146-4CA1-BCEA-FED47F6195CA"    -76     [1×1 struct
      5      "UA Footpod 239AE2"    "CF7B1A17-4104-4D7B-AE70-1837CAE9C9D0"    -78     [1×1 struct
      6      ""                     "67A1A92A-5F1C-4AF8-857A-99F194E9A5F8"    -82     [1×1 struct
      7      ""                     "5609D68D-0EED-41D7-BE19-F3ACAA119C7A"    -84     [1×1 struct
      8      ""                     "5A17DF46-6F71-4DAC-AF0D-0F28E3911187"    -85     [1×1 struct
      9      ""                     "9675A0FA-0394-468B-B908-040696E1C5BC"    -88     [1×1 struct
     10      ""                     "61540D17-C2DD-41D4-B107-E2E7374B11F4"    -88     [1×1 struct
     11      ""                     "C2A5CCC3-CA6C-4688-AAE9-A5BE039561F9"    -92     [1×1 struct
     12      ""                     "AF5E1195-4088-4A4B-ADA5-0BD3C91BFE62"    -93     [1×1 struct
     13      ""                     "83C69EFB-0FAD-4A35-B167-79C51A1F245D"    -94     [1×1 struct
     14      ""                     "993276FD-07EC-433D-85E1-E144B289B648"    -95     [1×1 struct
```

After the devices are found in MATLAB, connect to them by calling `ble.` Specify the name of the device if the name is unique, or specify the device address.

```
belt = ble("UA E39 MODULE")
```

```
belt = 
  ble with properties:

               Name: "UA E39 MODULE"
            Address: "8206F662-BA4A-483F-A908-516FF506BFB0"
          Connected: 1
           Services: [4×2 table]
    Characteristics: [22×5 table]

Show services and characteristics
```

```
shoe = ble("UA Footpod 239AE2")
```

```
shoe = 
  ble with properties:
```

```
             Name: "UA Footpod 239AE2"
          Address: "CF7B1A17-4104-4D7B-AE70-1837CAE9C9D0"
        Connected: 1
         Services: [8×2 table]
  Characteristics: [39×5 table]
```

Show services and characteristics

Access the Characteristics property of the `ble` object belt. This device has the "Heart Rate" service which contains the "Heart Rate Measurement" characteristic.

```
belt.Characteristics
```

```
ans=22×5 table
        ServiceName                    ServiceUUID                              Characteri
    ____________________    ______________________________________    __________________________

    "Heart Rate"            "180D"                                    "Heart Rate Measurement"
    "Heart Rate"            "180D"                                    "Body Sensor Location"
    "Heart Rate"            "180D"                                    "Heart Rate Control Point"
    "Battery Service"       "180F"                                    "Battery Level"
    "Device Information"    "180A"                                    "System ID"
    "Device Information"    "180A"                                    "Model Number String"
    "Device Information"    "180A"                                    "Serial Number String"
    "Device Information"    "180A"                                    "Firmware Revision String"
    "Device Information"    "180A"                                    "Hardware Revision String"
    "Device Information"    "180A"                                    "Software Revision String"
    "Device Information"    "180A"                                    "Manufacturer Name String"
    "Device Information"    "180A"                                    "IEEE 11073-20601 Regulato
    "Device Information"    "180A"                                    "PnP ID"
    "Custom"                "21A51000-4C86-11E2-BCFD-0800200C9A66"    "Custom"
    "Custom"                "21A51000-4C86-11E2-BCFD-0800200C9A66"    "Custom"
    "Custom"                "21A51000-4C86-11E2-BCFD-0800200C9A66"    "Custom"
      ⋮
```

Access the Characteristics property of the `ble` object shoe. This device has the "Running Speed and Cadence" service which contains the "RSC Measurement" characteristic.

```
shoe.Characteristics
```

```
ans=39×5 table
            ServiceName                      ServiceUUID                         Cha
    ___________________________    ______________________________________    ___________________

    "Device Information"           "180A"                                    "System ID"
    "Device Information"           "180A"                                    "Model Number Strin
    "Device Information"           "180A"                                    "Serial Number Stri
    "Device Information"           "180A"                                    "Firmware Revision 
    "Device Information"           "180A"                                    "Hardware Revision 
    "Device Information"           "180A"                                    "Software Revision 
    "Device Information"           "180A"                                    "Manufacturer Name 
    "Device Information"           "180A"                                    "IEEE 11073-20601 R
    "Device Information"           "180A"                                    "PnP ID"
    "Battery Service"              "180F"                                    "Battery Level"
    "Running Speed and Cadence"    "1814"                                    "RSC Measurement"
    "Running Speed and Cadence"    "1814"                                    "RSC Feature"
    "Running Speed and Cadence"    "1814"                                    "SC Control Point"
```

```
    "Custom"                        "21A54000-4C86-11E2-BCFD-0800200C9A66"    "Custom"
    "Custom"                        "21A54000-4C86-11E2-BCFD-0800200C9A66"    "Custom"
    "Custom"                        "21A54000-4C86-11E2-BCFD-0800200C9A66"    "Custom"
      ⋮
```

Read Heart Rate Data

Next, create an object for the "Heart Rate Measurement" characteristic by specifying its service and characteristic information.

```
hr = characteristic(belt, "heart rate", "heart rate measurement")
```

```
hr = 
  Characteristic with properties:

             Name: "Heart Rate Measurement"
             UUID: "2A37"
       Attributes: "Notify"
      Descriptors: [1x3 table]
 DataAvailableFcn: []

Show descriptors
```

Then read the current heart rate measurement from the device.

```
data = read(hr)
```

```
data = 1×4

    22    96    73     3
```

According to Bluetooth Low Energy Specification, the "Heart Rate Measurement" characteristic value contains a flag byte followed by one or many heart rate values. The format of the measurement value depends on the flag value. Convert the raw data to a heart rate in beats per minute (bpm).

```
flag = uint8(data(1));
% Get the first bit of the flag, which indicates the format of the heart rate value
heartRateValueFormat = bitget(flag, 1);
if heartRateValueFormat == 0
    % Heart rate format is uint8
    heartRate = data(2);
else
    % Heart rate format is uint16
    heartRate = double(typecast(uint8(data(2:3)), 'uint16'));
end
fprintf('Heart rate measurement: %d(bpm)\n', heartRate);
```

```
Heart rate measurement: 96(bpm)
```

Read Running Speed and Cadence Data

Similarly, create an object for the "RSC Measurement" characteristic by specifying its service and characteristic information.

```
rsc = characteristic(shoe, "running speed and cadence", "rsc measurement")
```

```
rsc = 
  Characteristic with properties:

             Name: "RSC Measurement"
             UUID: "2A53"
       Attributes: "Notify"
      Descriptors: [1x3 table]
 DataAvailableFcn: []

Show descriptors
```

Then read the current running speed and cadence value from the device.

```
data = read(rsc)
```

```
data = 1×10

     3     0     0     0     0     0    84    57     0     0
```

According to Bluetooth Low Energy Specification, the "RSC Measurement" characteristic value contains 2 bytes that represents the instantaneous speed and 1 byte that represents the instantaneous cadence. Convert the raw data to a running speed in meters per second (m/s) and to a cadence in steps per minute.

```
instantaneousSpeed = double(typecast(uint8(data(2:3)), 'uint16'))/256;
instantaneousCadence = data(4);
fprintf('Instantaneous speed: %.2f(m/s) and instantaneous cadence: %d(steps per minute)\n', insta
```

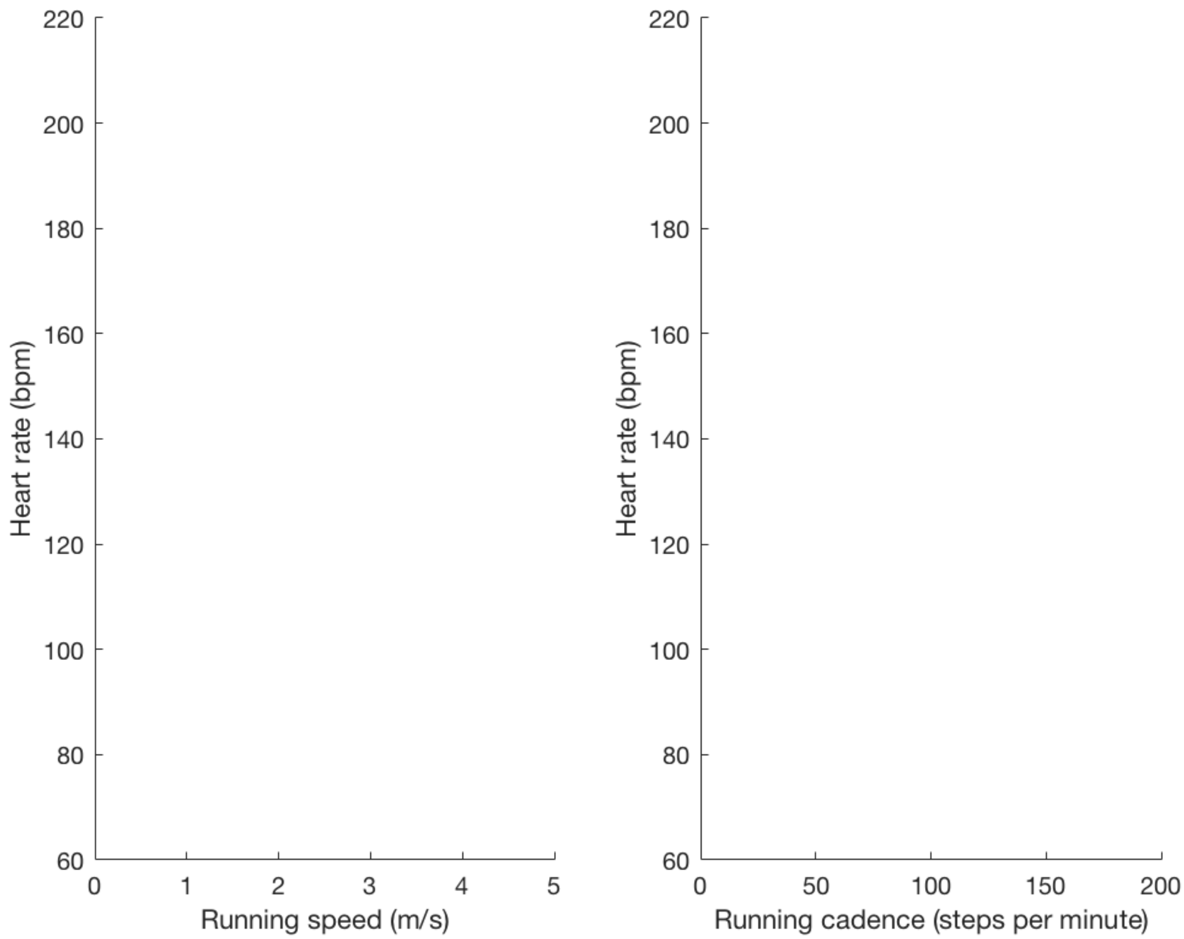
```
Instantaneous speed: 0.00(m/s) and instantaneous cadence: 0(steps per minute)
```

Correlate Heart Rate and Running Speed

After reading the heart rate and RSC data, track live data during a running session and correlate the two measurements to analyze fitness performance.

To show the correlation of fitness data, first create two plots to create an animation by adding data points in a loop.

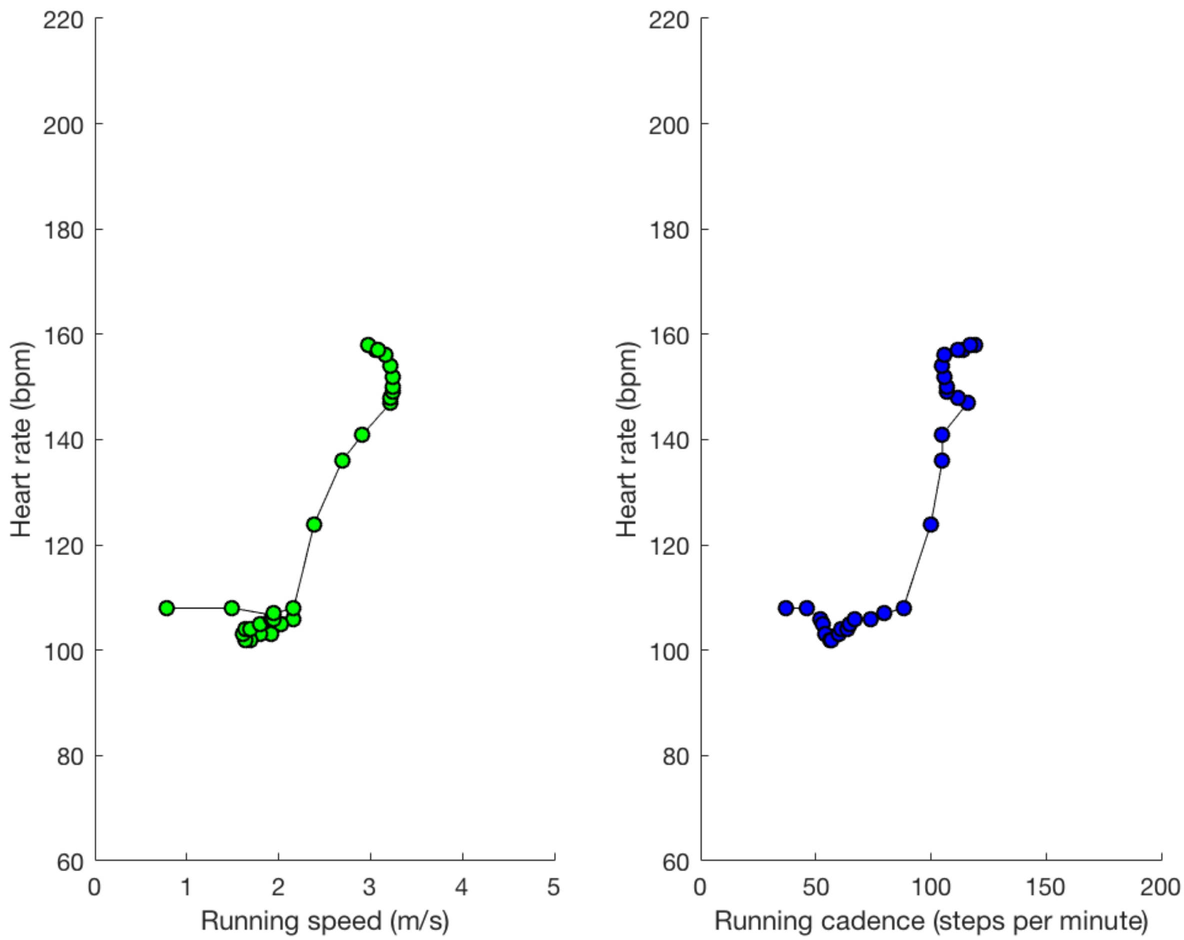
```
% Create a plot for running speed against heart rate
axSpeed = axes('XLim', [0, 5], 'YLim', [60, 220]);
xlabel(axSpeed, 'Running speed (m/s)');
ylabel(axSpeed, 'Heart rate (bpm)');
subplot(1, 2, 1, axSpeed);
hSpeed = animatedline(axSpeed, 'Marker', 'o', 'MarkerFaceColor', 'green');

% Create a plot for running cadence against heart rate
axCadence = axes('XLim', [0, 200], 'YLim', [60 220]);
xlabel(axCadence, 'Running cadence (steps per minute)');
ylabel(axCadence, 'Heart rate (bpm)');
subplot(1, 2, 2, axCadence);
hCadence = animatedline(axCadence, 'Marker', 'o', 'MarkerFaceColor', 'blue');
```



Next, read device data in a loop and update the plot as the user progresses from walking to jogging to running.

```
for loop = 1:30
    % Get heart rate data
    data = read(hr);
    flag = uint8(data(1));
    heartRateValueFormat = bitget(flag, 1);
    if heartRateValueFormat == 0
        heartRate = data(2);
    else
        heartRate = double(typecast(uint8(data(2:3)), 'uint16'));
    end

    % Get running speed data
    data = read(rsc);
    instantaneousSpeed = double(typecast(uint8(data(2:3)), 'uint16'))/256;
    instantaneousCadence = data(4);

    % Update plot with new data
    addpoints(hSpeed, instantaneousSpeed, heartRate);
```

```
    addpoints(hCadence, instantaneousCadence, heartRate);
    drawnow;
end
```



These plots generally show that higher running speed and cadence values correspond to an increased heart rate.

Disconnect from Device

Clear the device objects when you are finished working with them.

```
clear belt shoe
```

Track Orientation of Bluetooth Low Energy Device

This example shows how to track device orientation with device motion sensor data using Bluetooth® Low Energy communication.

Hardware Setup

This example uses the Nordic Thingy:52™ device. Nordic Thingy:52 is a Bluetooth Low Energy device with a 9-axis motion sensor. This device provides a rich set of sensor data including raw accelerometer, gyroscope, compass, and fused data. This example uses the device calculated rotation matrix to track the device orientation.

Discover and Connect to Device

First, check that the Bluetooth Low Energy device supports connections by finding it in MATLAB. The `blelist` function scans nearby Bluetooth Low Energy peripheral devices that are advertising.

```
blelist
```

```
ans=20×5 table
    Index      Name         Address        RSSI    Advertisement
    _____    ________    ______________    ____    _____________

      1      "Thingy"    "F2DF635320F6"    -54     [1×1 struct]
      2      ""          "5AE98748DC34"    -73     [1×1 struct]
      3      ""          "7A9762B423E0"    -76     [1×1 struct]
      4      ""          "5E0EAEF93E78"    -76     [1×1 struct]
      5      ""          "08534F9CC17B"    -77     [1×1 struct]
      6      ""          "4323693660AC"    -79     [1×1 struct]
      7      ""          "5386B1B9FCEC"    -82     [1×1 struct]
      8      ""          "2D132D3ACD33"    -83     [1×1 struct]
      9      ""          "537E555A0188"    -84     [1×1 struct]
     10      ""          "237E6384E9BF"    -87     [1×1 struct]
     11      ""          "2C0CA5F4793C"    -88     [1×1 struct]
     12      ""          "55D810EF7331"    -89     [1×1 struct]
     13      ""          "3A01FA8D3D18"    -89     [1×1 struct]
     14      ""          "2084C6A7DA4D"    -90     [1×1 struct]
     15      ""          "52DBAB89F58F"    -91     [1×1 struct]
     16      ""          "528E12038BD6"    -91     [1×1 struct]
      ⋮
```

After the device is found in MATLAB, connect to it by calling `ble.` Specify the name of the device if it has a unique name, or specify the device address.

```
b = ble("Thingy")
```

```
b = 
  ble with properties:

               Name: "Thingy"
            Address: "F2DF635320F6"
          Connected: 1
           Services: [9×2 table]
    Characteristics: [38×5 table]

Show services and characteristics
```

Access the Characteristics property of the `ble` object. The device has the "Motion Service" service, which contains the "Rotation Matrix" characteristic.

```
b.Characteristics
```

```
ans=38×5 table
           ServiceName                                 ServiceUUID                              Char
    ________________________________    ________________________________________    _____________

    "Generic Access"                    "1800"                                      "Device Name"
    "Generic Access"                    "1800"                                      "Appearance"
    "Generic Access"                    "1800"                                      "Peripheral Prefe
    "Generic Access"                    "1800"                                      "Central Address
    "Generic Attribute"                 "1801"                                      "Service Changed'
    "Thingy Configuration Service"      "EF680100-9B35-4933-9B10-52FFA9740042"      "Device Name"
    "Thingy Configuration Service"      "EF680100-9B35-4933-9B10-52FFA9740042"      "Advertising Para
    "Thingy Configuration Service"      "EF680100-9B35-4933-9B10-52FFA9740042"      "Connection Param
    "Thingy Configuration Service"      "EF680100-9B35-4933-9B10-52FFA9740042"      "Eddystone URL"
    "Thingy Configuration Service"      "EF680100-9B35-4933-9B10-52FFA9740042"      "Cloud Token"
    "Thingy Configuration Service"      "EF680100-9B35-4933-9B10-52FFA9740042"      "Firmware Versior
    "Thingy Configuration Service"      "EF680100-9B35-4933-9B10-52FFA9740042"      "MTU Request"
    "Thingy Configuration Service"      "EF680100-9B35-4933-9B10-52FFA9740042"      "NFC Tag Content'
    "Weather Station Service"           "EF680200-9B35-4933-9B10-52FFA9740042"      "Temperature"
    "Weather Station Service"           "EF680200-9B35-4933-9B10-52FFA9740042"      "Pressure"
    "Weather Station Service"           "EF680200-9B35-4933-9B10-52FFA9740042"      "Humidity"
      ⋮
```

Read Sensor Data

Next, create an object for the "Rotation Matrix" characteristic by specifying its service and characteristic information.

```
c = characteristic(b, "motion service", "rotation matrix")
```

```
c =
  Characteristic with properties:

             Name: "Rotation Matrix"
             UUID: "EF680408-9B35-4933-9B10-52FFA9740042"
       Attributes: "Notify"
      Descriptors: [1x3 table]
 DataAvailableFcn: []

Show descriptors
```

Then read the current rotation matrix value from the device.

```
data = read(c)
```

```
data = 1×18

   253    63   255   255    15     1   255   255     0    64     8     0   240   254   247   255
```

According to the Nordic Thingy:52 documentation, this raw data contains a 3-by-3 matrix where each element is a 16-bit integer sent in 2 bytes. Each element represents a signed floating point number, composed of 2 sign-and-exponent bits and 14 fraction bits. Interpret the raw data as a rotation matrix.

```
% Prepare 4-by-4 transform matrix to plot later (assume the device has no
% translation and only rotation)
transformMatrix = eye(4);
% Populate the transform matrix with 9 rotation matrix elements
for row = 1:3
    for column = 1:3
        % Extract the 2 bytes representing the current element in the rotation matrix
        beginIndex = (row-1)*3 + (column-1);
        element = data(2*beginIndex + (1:2));
        transformMatrix(row, column) = double(typecast(uint8(element), 'int16')) / (2^14);
    end
end
% Display the transform matrix
disp(transformMatrix);
```

```
    0.9998   -0.0001    0.0165         0
   -0.0001    1.0000    0.0005         0
   -0.0166   -0.0005    0.9998         0
         0         0         0    1.0000
```

Track Device Orientation

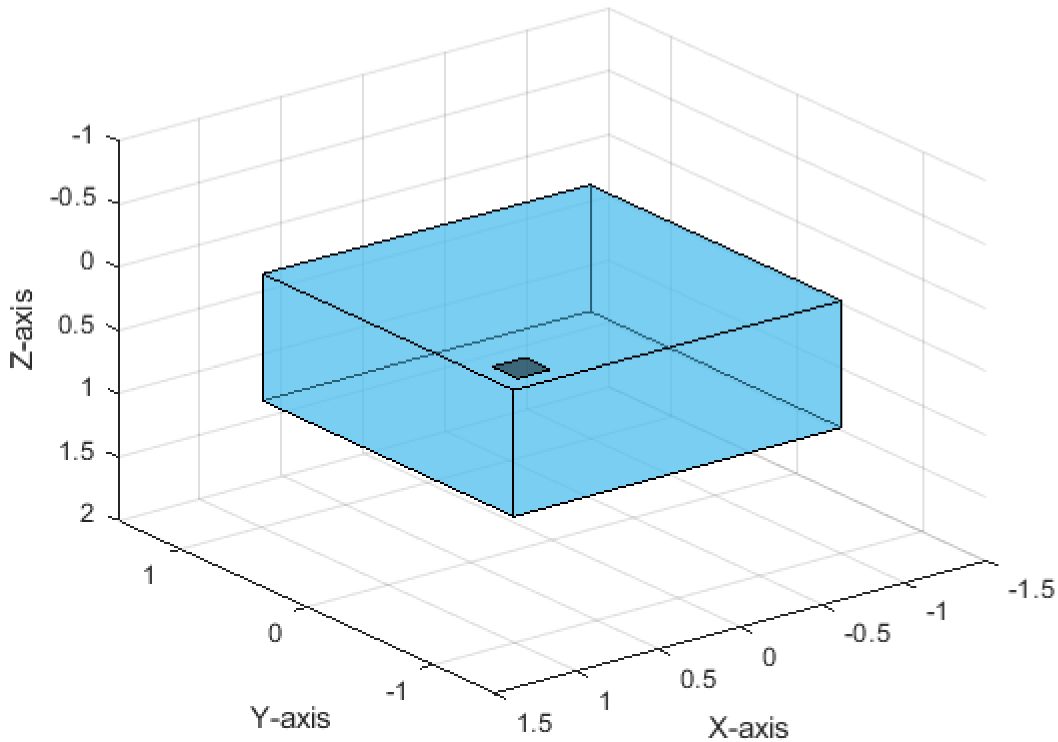
To show live tracking of the device orientation, first plot a 3-D object representing the Nordic Thingy:52 device.

```
% Create a 3-D plot
ax = axes('XLim', [-1.5 1.5], 'YLim', [-1.5 1.5], 'ZLim', [-1 2]);
xlabel(ax, 'X-axis');
ylabel(ax, 'Y-axis');
zlabel(ax, 'Z-axis');
% Reverse the 2 axis directions to match the device coordinate system
set(ax, 'Zdir', 'reverse');
set(ax, 'Xdir', 'reverse');
grid on; view(3);

% Define the surface color
color = [0.3010 0.7450 0.9330];

% Create patches for all cube surfaces by specifying the four corners of each surface
top = [-1 -1 1; 1 -1 1; 1 1 1; -1 1 1];
p(1) = patch(top(:,1), top(:,2), top(:,3), color);

bottom = [-1 -1 0; 1 -1 0; 1 1 0; -1 1 0];
p(2) = patch(bottom(:,1), bottom(:,2), bottom(:,3), color);

front = [1 -1 0; 1 1 0; 1 1 1; 1 -1 1];
p(3) = patch(front(:,1), front(:,2), front(:,3), color);

back = [-1 -1 0; -1 1 0; -1 1 1; -1 -1 1];
p(4) = patch(back(:,1), back(:,2), back(:,3), color);

left = [1 -1 0; -1 -1 0; -1 -1 1; 1 -1 1];
p(5) = patch(left(:,1), left(:,2), left(:,3), color);

right = [1 1 0; -1 1 0; -1 1 1; 1 1 1];
```

```
p(6) = patch(right(:,1), right(:,2), right(:,3), color);

mark = [0.9 -0.7 -0.01; 0.7 -0.7 -0.01; 0.7 -0.9 -0.01; 0.9 -0.9 -0.01];
p(7) = patch(mark(:,1), mark(:,2), mark(:,3), 'black');

% Set the object transparency
alpha(0.5)
```

After the 3-D object is created, link the rotation matrix data acquired from the device to the plot by using `hgtransform`.

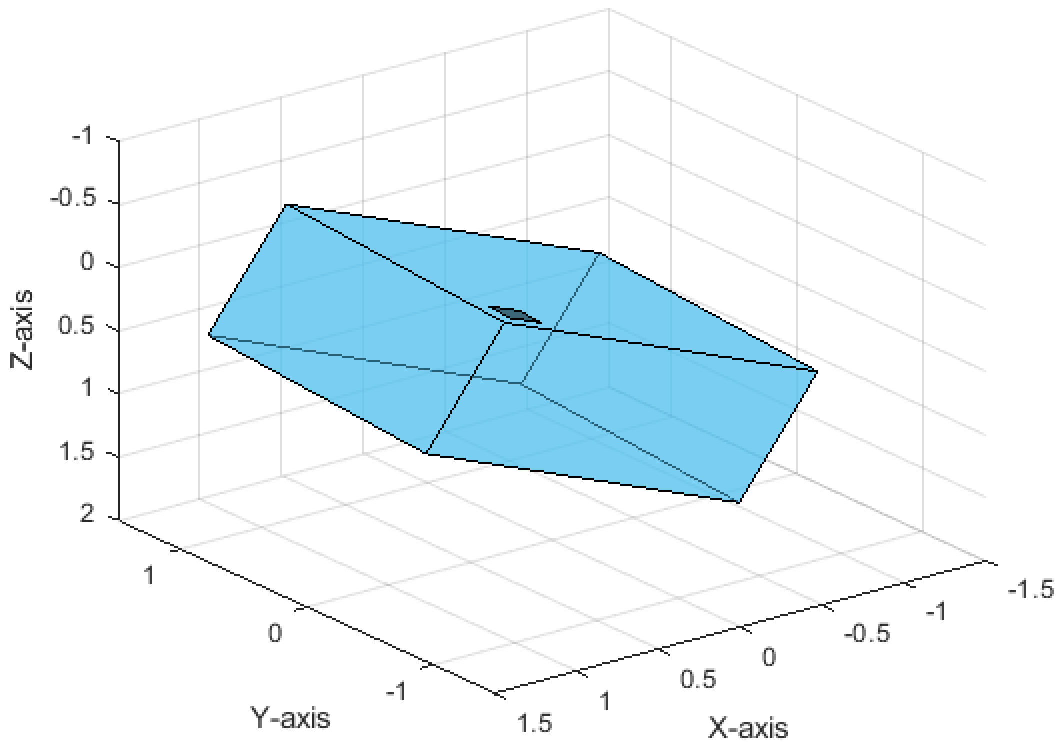
```
tfObject = hgtransform('Parent', ax);
set(p, 'Parent', tfObject);
```



With the `Transform` object, pull device data in a loop and use the data to update the object orientation. The rotation matrix data sent from the device has precision loss, which can cause matrix transformation warnings. For this example, ignore the warning by suppressing it. For greater accuracy, you can use "Euler" or "Quaternion" characteristic data and convert it to a rotation matrix using Robotics System Toolbox™.

```
warning('off', 'MATLAB:hg:DiceyTransformMatrix');
for loop = 1:100
    % Acquire device data
    data = read(c);
    % Prepare 4-by-4 transform matrix to plot later
    transformMatrix = eye(4);
    % Populate the transform matrix with 9 rotation matrix elements
```

```
    for row = 1:3
        for column = 1:3
            % Extract the 2 bytes representing the current element in the rotation matrix
            beginIndex = (row-1)*3 + (column-1);
            element = data(2*beginIndex + (1:2));
            transformMatrix(row, column) = double(typecast(uint8(element), 'int16')) / (2^14);
        end
    end

    % Update plot
    set(tfObject, 'Matrix', transformMatrix);
    pause(0.1);
end
```




```
warning('on', 'MATLAB:hg:DiceyTransformMatrix');
```

Close Device Connection

Clear the device object when you are finished working with it.

```
clear b
```

Troubleshooting Bluetooth Low Energy

If you are having issues connecting to your Bluetooth Low Energy peripheral device from MATLAB or are unable to read or write data, you can try some of the following troubleshooting tips.

For more information about the Bluetooth Low Energy interface, see:

- “Bluetooth Low Energy Communication Overview” on page 14-2
- “Find Your Bluetooth Low Energy Peripheral Devices” on page 14-4
- “Work with Device Characteristics and Descriptors” on page 14-7

Supported Platforms

The Bluetooth Low Energy interface is supported on these platforms:

- macOS 10.13 High Sierra or later
- Windows 10, version 1709 or later

Before trying other troubleshooting steps, make sure your computer is running one of these platforms.

Device Discovery and Connection

If MATLAB does not detect your built-in or external Bluetooth adapter when you call `blelist`, try the following:

- Make sure the adapter supports Bluetooth 4.0 and higher.
- Restart Bluetooth services on your computer.
- Update to the latest device drivers for your adapter.
- Reboot your computer.

If your device does not appear in the `blelist` output, make sure that you have done the following:

- Power on your peripheral device.
- Bring your peripheral device within range of your computer.
- Disconnect your peripheral device from any other devices or applications first. The output from `blelist` shows you only nearby devices that are currently advertising data. If your device is already connected in another application or in MATLAB, it might not appear in the output.
- Try a larger value for the `Timeout` parameter in `blelist`. This increases the amount of time MATLAB scans for nearby devices. For example, `blelist("Timeout",20)` searches for nearby peripheral devices for 20 seconds. The default timeout value is three seconds. If your device transmits advertisement data less often than once every three seconds, MATLAB might not capture it.

If your peripheral device powers off or disconnects, the UUID might change when it powers on again or reconnects. However, the name remains the same in the `blelist` output. If you are trying to create a new `ble` object for the same device, specify the new UUID instead of the name.

On Windows, if a peripheral device has already been paired but the firmware that defines characteristics and descriptors is changed, `ble` might fail to connect the device to your computer. To fix this, disconnect and then reconnect your device to Windows.

If you are unable to create a `characteristic` object for your Bluetooth Low Energy device on Windows 10, try pairing your device on Windows before using `ble` to connect to it in MATLAB. You can pair to a device in **Windows Settings** > **Devices** > **Add Bluetooth or other device**.

Read and Write Data

Using `read(c,'latest')` or `read(c)` inside a callback function for a characteristic with a high rate can throw an error or block MATLAB for a long time. Instead, use `read(c,'oldest')` in a callback function. For an example, see “Read Characteristic Data from a Bluetooth Low Energy Peripheral Device Using a Callback Function”.

On macOS, some device characteristics require authentication to read or write for the first time. After you create a `ble` object, run `read` or `write`. Follow the prompts that appear on your computer to pair your peripheral device.

See Also

`ble` | `blelist` | `characteristic` | `descriptor`

More About

- “Bluetooth Low Energy Communication Overview” on page 14-2
- “Find Your Bluetooth Low Energy Peripheral Devices” on page 14-4
- “Work with Device Characteristics and Descriptors” on page 14-7