MATLAB®

Data Import and Export
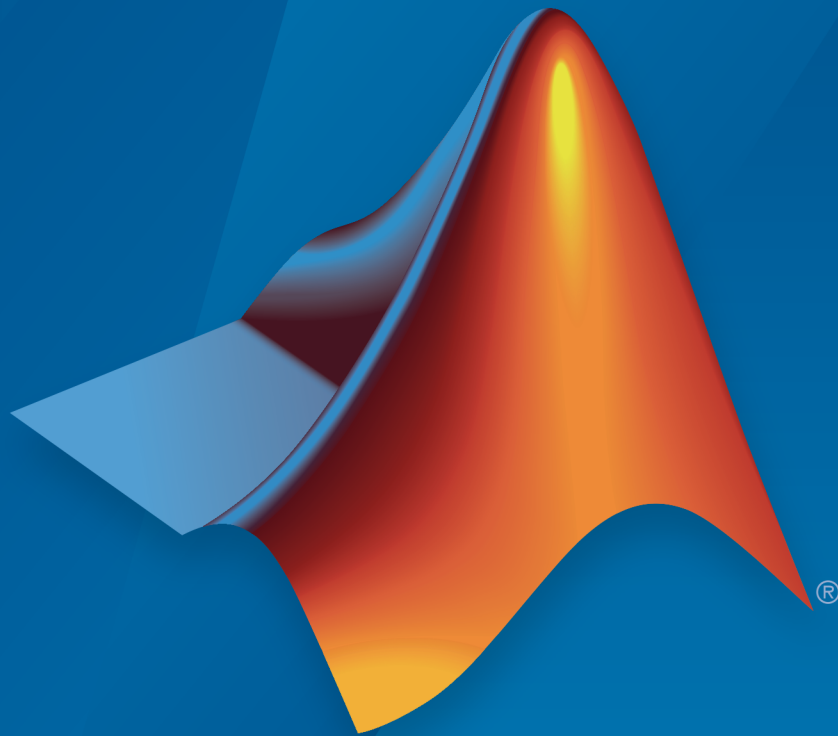
MATLAB®

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.
3 Apple Hill Drive
Natick, MA 01760-2098

Revision History

Contents

<div align="right">

Text Files

</div>

2

3

4

Low-Level File I/O

5

Images

Scientific Data

6

Audio and Video

7

8

<div align="right">

XML Documents

</div>

9

<div align="right">

Memory-Mapping Data Files

</div>

Internet File Access

10

11

TCP/IP Support in MATLAB

12

1

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 `importdata` function, or import these file formats interactively, with the following exceptions:

- `importdata` and interactive import do not support H5 and netCDF files.
- `importdata` does not support HDF files.

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	`csvread`	`csvwrite`
		Delimited numbers	`dlmread`	`dlmwrite`
		Delimited numbers, or a mix of strings and numbers	`textscan`	none
		Column-oriented delimited numbers or a mix of strings and numbers	`readtable`	`writetable`
Spreadsheet	XLS XLSX XLSM XLSB (Systems with Microsoft® Excel® for Windows® only)	Worksheet or range of spreadsheet	`xlsread`	`xlswrite`
		Column-oriented data in worksheet or range of spreadsheet	`readtable`	`writetable`

File Content	Extension	Description	Import Function	Export Function
	XLTM (import only) XLTX (import only) ODS (Systems with COM interface)			
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"
	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		

File Content	Extension	Description	Import Function	Export Function
	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		

File Content	Extension	Description	Import Function	Export Function
Audio (Windows)	M4A MP4	MPEG-4	`audioread`	`audiowrite`
	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`		

File Content	Extension	Description	Import Function	Export Function
Video (Linux)	any	Formats supported by your installed GStreamer plug-ins, including `.ogg`	`VideoReader`	none

You can use web services such as a RESTful API 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"
- "Call WSDL Web Service"

Methods for Importing Data

In this section...
"Tools that Import Multiple File Formats" on page 1-7
"Importing Specific File Formats" on page 1-7
"Importing Data with Low-Level I/O" on page 1-8

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

1-7

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 9-2. 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 4-10

Import Images, Audio, and Video Interactively

In this section...
"Viewing the Contents of a File" on page 1-9
"Specifying Variables" on page 1-10
"Generating Reusable MATLAB Code" on page 1-11

Note: For information on importing text files, see "Select Text File Data Using Import Tool" on page 2-4. For information on importing spreadsheets, see "Select Spreadsheet Data Using Import Tool" on page 3-4. For information on importing HDF4 files, see "Import HDF4 Files Interactively" on page 6-61.

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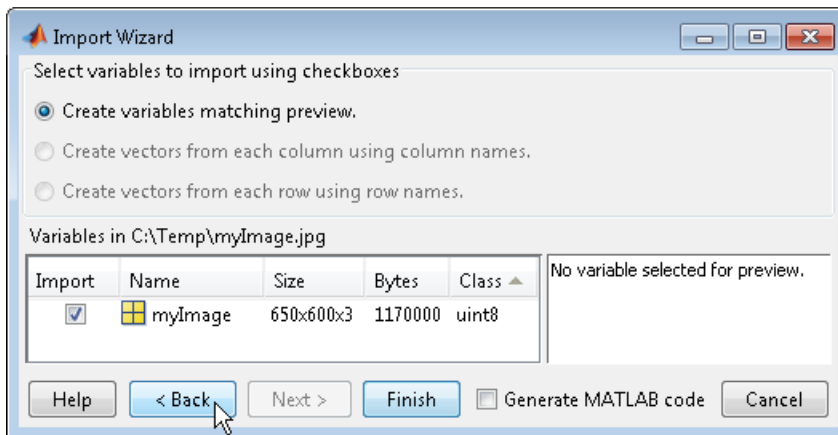
The right pane of the new window includes a preview tab. Click the button in the preview tab to show an image or to play audio or video.
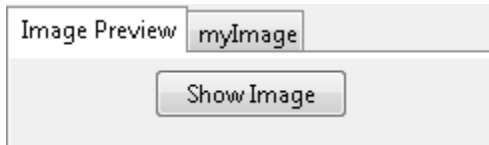
Specifying Variables

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:
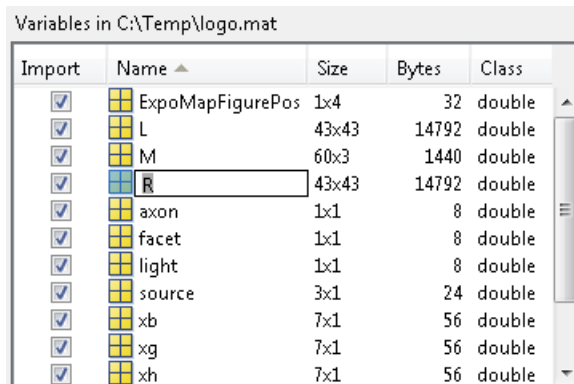
- "Renaming or Deselecting Variables" on page 1-10
- "Importing to a Structure Array" on page 1-11

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.

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 a 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.

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
```

View Contents of MAT-File

MAT-files are binary MATLAB format files that store workspace variables.

To see the variables in a MAT-file before loading the file into your workspace, click the file name in the Current Folder browser. Information about the variables appears in the Details Panel.

Alternatively, use the command `whos -file` *`filename`*. This function returns the name, dimensions, size, and class of all variables in the specified MAT-file.

For example, view the contents of the example file `durer.mat`:

```
whos -file durer.mat
```

MATLAB returns:

```
 Name          Size              Bytes  Class      Attributes

  X           648x509          2638656  double
  caption       2x28               112  char
  map         128x3              3072  double
```

The byte counts represent the number of bytes that the data occupies when loaded into the MATLAB workspace. Compressed data uses fewer bytes in a file than in the workspace. In Version 7 or higher MAT-files, MATLAB compresses data. For more information, see "MAT-File Versions" on page 1-23.

Load Parts of Variables from MAT-Files

In this section...

Load Using the matfile Function

This example shows how to load part of a variable from an existing MAT-file.

To run the code in this example, create a Version 7.3 MAT-file with two variables.

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

Construct a `matlab.io.MatFile` object that can load parts of variables from the file, `example.mat`.

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

example =

  matlab.io.MatFile

  Properties:
      Properties.Source: C:\Documents\MATLAB\example.mat
    Properties.Writable: false
                      A: [5x5   double]
                      B: [10x10 double]
```

The `matfile` function creates a `matlab.io.MatFile` object that corresponds to a MAT-file and displays the properties of the `matlab.io.MatFile` object.

Load the first column of `B` from `example.mat` into variable `firstColB`.

```
firstColB = example.B(:,1);
```

When you index into objects associated with Version 7.3 MAT-files, MATLAB loads only the part of the variable that you specify.

By default, `matfile` only allows loading from existing MAT-files. To enable saving, call `matfile` with the `Writable` parameter.

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

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

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

Load from Variables with Unknown Names

This example shows how to dynamically access variables, whose names are not always known. Consider the example MAT-file, `topography.mat`, that contains one or more arrays with unknown names.

Construct a `matlab.io.MatFile` object that corresponds to the file, `topography.mat`. Call `who` to get the variable names in the file.

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

varlist =

    '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_`. Access the data in each variable as properties of `matObj`. Because `varName` is a variable, enclose it in parentheses.

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

```
end
```

View the contents of the structure array, S.

```
S
```

```
S =

    elevationOf_topomap1: [64x1 double]
    elevationOf_topomap2: [128x1 double]
```

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

Avoid Repeated File Access

The primary advantage of `matfile` over the `load` function 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 negatively impacts the performance of your code.

For example, suppose a variable in your file contains many rows and columns, and loading a single row requires most of the available memory. To calculate the mean of the entire data set, calculate the mean of each row, and then find the overall mean.

```
example = matfile('example.mat');
[nrows, ncols] = size(example,'B');

avgs = zeros(1, nrows);
for idx = 1:nrows
    avgs(idx) = mean(example.B(idx,:));
end
overallAvg = mean(avgs);
```

Avoid Inadvertently Loading Entire Variables

When you do not know the size of a large variable in a MAT-file, and want to load parts of that variable at a time, do not use the `end` keyword. Rather, call the size method for `matlab.io.MatFile` objects. For example, this code

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

requires less memory than

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

which temporarily loads the entire contents of B. For very large variables, loading takes a long time or generates Out of Memory errors.

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

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

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

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

The difference in syntax is subtle, but significant.

Partial Loading Requires Version 7.3 MAT-Files

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

The matfile function creates 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**.

Save Parts of Variables to MAT-Files

In this section...
"Save Using the matfile Function" on page 1-19
"Partial Saving Requires Version 7.3 MAT-Files" on page 1-20

Save Using the matfile Function

This example shows how to change and save part of a variable in a MAT-file. To run the code in this example, create a Version 7.3 MAT-file with two variables.

```
A = rand(5);
B = ones(4,8);
save example.mat A B -v7.3;
clear A B
```

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

```
example = matfile('example.mat','Writable',true)
example.B(1,:) = 2 * example.B(1,:);
```

The `matfile` function creates a `matlab.io.MatFile` object that corresponds to a MAT-file:

```
matlab.io.MatFile

Properties:
      Properties.Source: C:\Documents\MATLAB\example.mat
    Properties.Writable: true
                      A: [5x5 double]
                      B: [4x8 double]
```

When you index into objects associated with Version 7.3 MAT-files, MATLAB loads and saves only the part of the variable that you specify. This partial loading or saving requires less memory than `load` or `save` commands, which always operate on entire variables.

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 your file contains many rows and columns, and loading

a single row requires most of the available memory. Rather than updating one element at a time, update each row.

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

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

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

```
example = matfile('example.mat','Writable',true);
example.B = 10 * example.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:

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

This method always requires that you load and save the entire variable.

Use either method to add a variable to the file. For example, this code

```
example = matfile('example.mat','Writable',true);
example.C = magic(8);
```

performs the same save operation as

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

Partial Saving Requires Version 7.3 MAT-Files

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

The `matfile` function creates 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**.

Save Structure Fields as Separate Variables

If any of the variables in your current workspace are structure arrays, the default behavior of the `save` function is to store the entire structure. To store fields of a scalar structure as individual variables, use the `-struct` option to the `save` function.

For example, consider structure `S`:

```
S.a = 12.7;  S.b = {'abc', [4 5; 6 7]};  S.c = 'Hello!';
```

Save the entire structure to `newstruct.mat` with the usual syntax:

```
save('newstruct.mat', 'S')
```

The file contains the variable `S`:

```
  Name      Size                   Bytes  Class
  S         1x1                      550  struct
```

Alternatively, save the fields individually with the `-struct` option:

```
save('newstruct.mat', '-struct', 'S')
```

The file contains variables `a`, `b`, and `c`, but not `S`:

```
  Name      Size                   Bytes  Class
  a         1x1                        8  double
  b         1x2                      158  cell
  c         1x6                       12  char
```

To save only selected fields, such as `a` and `c`:

```
save('newstruct.mat', '-struct', 'S', 'a', 'c')
```

MAT-File Versions

In this section...
"Overview of MAT-File Versions" on page 1-23
"Save to Nondefault MAT-File Version" on page 1-24
"Data Compression" on page 1-25
"Accelerate Save and Load Operations for Version 7.3 MAT-Files" on page 1-25

Overview of MAT-File Versions

MAT-files are binary MATLAB files that store workspace variables. By default, all save operations except new file creation with the `matfile` function create Version 7 MAT-files. When you create new MAT-files using the `matfile`, the default MAT-file version is 7.3.

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

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.

This table lists and compares all MAT-file versions.

Version	Loads in MATLAB Versions	Supported Features	Compression	Maximum Size of Each Variable	Value of `version` argument to `save` function	Preference Option
Version 7.3	7.3 (R2006b) 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	7.0 (R14) or later	Unicode® character	Yes	2^{31} bytes per variable	`'-v7'`	**MATLAB Version 7 or later**

Version	Loads in MATLAB Versions	Supported Features	Compression	Maximum Size of Each Variable	Value of `version` argument to `save` function	Preference Option
		encoding, which enables file sharing between systems that use different default character encoding schemes, and all Version 6 features.				(save -v7)
Version 6	5 (R8) 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 value of `version` as the last input argument 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 fact, loading compressed data is sometimes *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.

However, 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`

File Size Increases Unexpectedly When Growing Array

This example shows how to prevent an array from growing when writing one million double-precision values to a file, by assigning initial values to the array.

Construct a `matlab.io.MatFile` object for writing.

```
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');
  ```

`matFileOfDoubles.mat` is less than 5000 bytes. Assigning an initial value to the last element of the array does not create a large file.

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 now because the array is populated with data.

Loading Variables within a Function

If you define 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.

For example, consider a MAT-file with variables height, width, and length. If you load these variables using a function such as findVolume,

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

MATLAB interprets the reference to length as a call to the MATLAB length function, and returns an error:

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

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

- Load into a structure array. For example, define the findVolume function as follows:

  ```
  function vol = findVolume(myfile)
    dims = load(myfile);
    vol = dims.height * dims.width * dims.length;
  ```
- Explicitly include the names of variables in the call to the load function.
- Initialize variables (e.g., assign to an empty matrix or empty string) within the function before calling load.

To determine whether a particular name is associated with a MATLAB function, use the exist function.

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 string using the CPU counter and time, and this string 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.

2

Text Files

Ways to Import Text Files

You can import text files into MATLAB both interactively and programmatically.

To import data interactively, use the Import Tool. You can generate code to repeat the operation on multiple similar files. The Import Tool supports text files, including those with the extensions `.txt`, `.dat`, `.csv`, `.asc`, `.tab`, and `.dlm`. These text files can be nonrectangular, and can have row and column headers, as shown in the following figure. Data in these files can be a combination of numeric and nonnumeric text, and can be delimited by one or more characters.

To import data from text files programmatically, use an import function. Most of the import functions for text files require that each row of data has the same number of columns, and they allow you to specify a range of data to import.

```
Text header line ─────────────┐
                               │
               Class Grades for Spring Term
Column headers ──────────────┐
                             └─ Grade1  Grade2  Grade3
               John      85      90      95
Row headers ──── Ann       90      92      98
               Martin   100      95      97
               Rob       77      86      93
Tab-delimited data ──────────┘
```

This table compares the primary import options for text files.

Import Option	Description	For Examples, See…
Import Tool	Import a file or range of data to column vectors, a matrix, a cell array, or a table. You can generate code to repeat the operation on multiple similar files.	"Select Text File Data Using Import Tool" on page 2-4
`readtable`	Import column-oriented data into a table.	"Import Mixed Data from Text Files" on page 2-14
`csvread`	Import a file or range of comma-separated numeric data to a matrix.	"Import Comma-Separated Data" on page 2-11

Import Option	Description	For Examples, See...
`dlmread`	Import a file or a range of numeric data separated by any single delimiter to a matrix.	"Import Delimited Numeric Data" on page 2-12
`TabularTextDatastore` with `read` or `readall` functions	Import one or more column-oriented text files. Each file can be very large and does not need to fit in memory.	"Read and Analyze Large Tabular Text File" on page 11-102
`textscan`	Import a nonrectangular or arbitrarily formatted text file to a cell array.	"Import Data from a Nonrectangular Text File" on page 2-25

For information on importing files with more complex formats, see "Import Text Data Files with Low-Level I/O" on page 4-2.

Select Text File Data Using Import Tool

In this section...
"Select Data Interactively" on page 2-4
"Import Data from Multiple Text Files" on page 2-7

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 this example, `grades.txt`, contains the following data (to create the file, use any text editor, and copy and paste):

```
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
```

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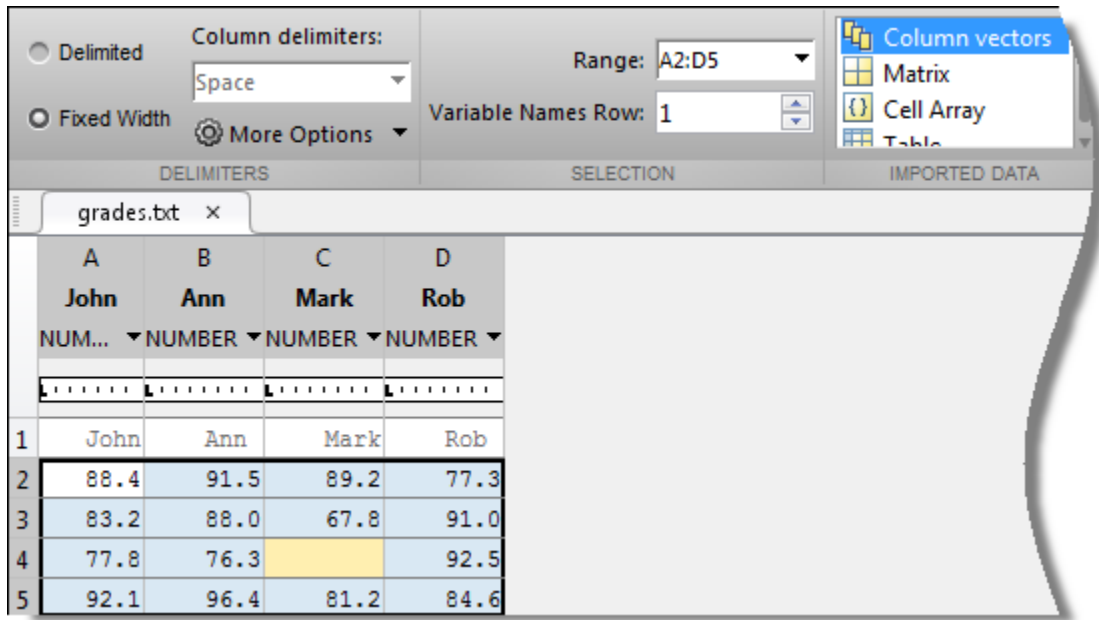
The Import Tool recognizes that `grades.txt` is a fixed width file. In the **Imported Data** section, select how you want the data to be imported. The following table indicates how data is imported depending on the option you select.

Option Selected	How Data is Imported
Column vectors	Import each column of the selected data as an individual m-by-1 vector.
Matrix	Import selected data as an m-by-n numeric 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.

Under **More Options**, you can specify whether the Import Tool should use a period or a comma as the decimal separator for numeric values.
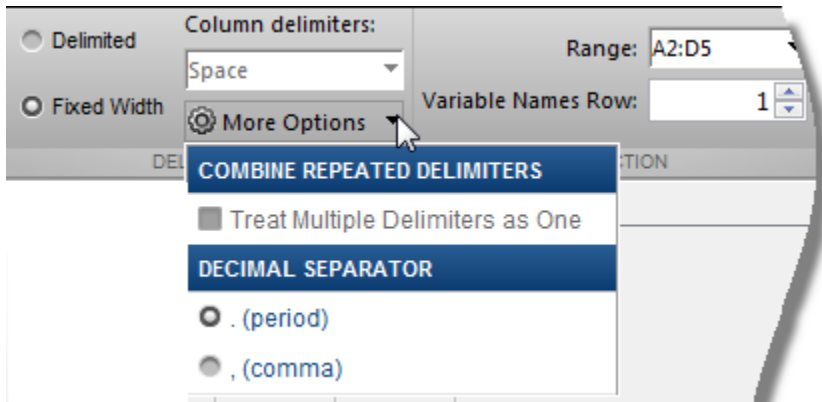
Double-click on a variable name to rename it.

You also can use the **Variable Names Row** box in the **Selection** section to select the row in the text file that the Import Tool uses 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.
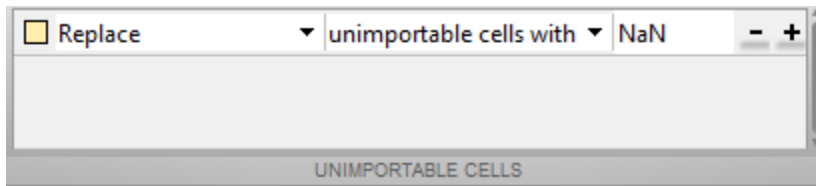
All rules apply to the imported data only, and do not change the data in the file. You must specify rules any time the range includes nonnumeric data and you are importing into a matrix or numeric column vectors.

You can see how your data will be imported when you 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

This example shows how to perform the same import operation on multiple files using the Import Tool. You can generate code from the Import Tool, making 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 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`.

More About

- "Ways to Import Text Files" on page 2-2

Import Dates and Times from Text Files

Formatted dates and times (such as `'01/01/01'` or `'12:30:45'`) are *not* numeric fields. MATLAB interprets dates and times in files as text strings unless you specify that they should be interpreted as date and time information. When reading a text file using `textscan` or `readtable`, indicate date and time data using the `%D` format specifier. Additionally, you can specify a particular date and time format using `%{`*fmt*`}D`, where *fmt* is the date and time format. For example, the format specifier, `%{dd/MMM/yyyy}D`, describes the datetime format, day/month/year.

You can use the Import Tool to import formatted dates and times as datetime values. 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.
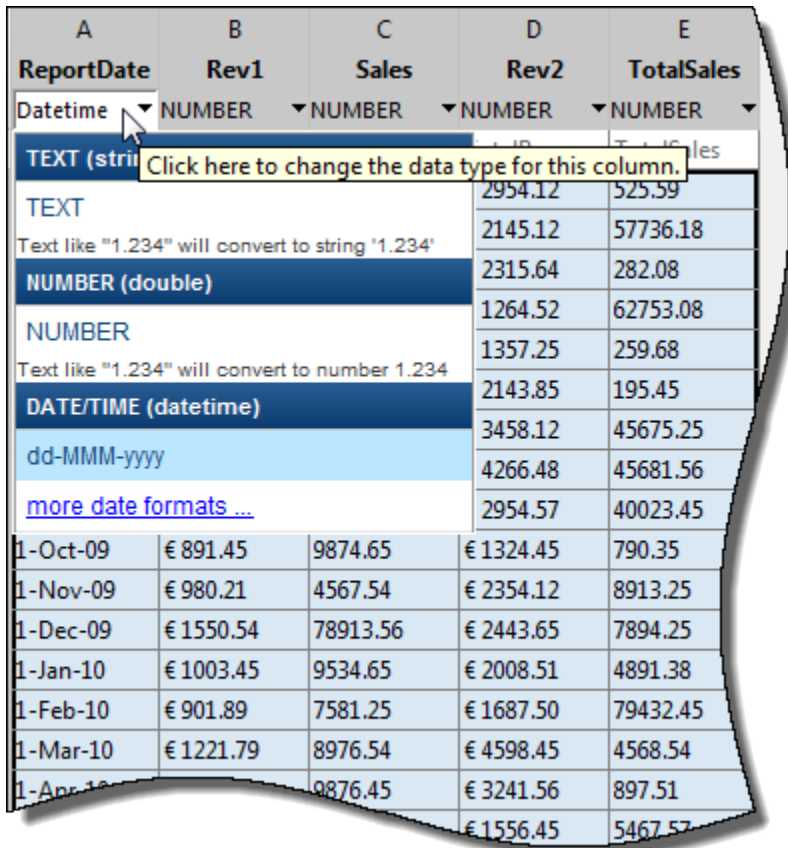
See Also

`readtable` | `textscan`

More About

- "Import Mixed Data from Text Files" on page 2-14

Import Numeric Data from Text Files

Import Comma-Separated Data

This example shows how to import comma-separated numeric data from a text file, using the `csvread` function.

Create a sample file named `ph.dat` that contains the following comma-separated data:

```
85.5, 54.0, 74.7, 34.2
63.0, 75.6, 46.8, 80.1
85.5, 39.6, 2.7, 38.7

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

The sample file, `ph.dat`, resides in your current folder.

Read the entire file using `csvread`. The file name is the only required input argument to the `csvread` function.

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


M =

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

`M` is a 3-by-4 `double` array containing the data from the file.

Import only the rectangular portion of data starting from the first row and third column in the file. When using `csvread`, row and column indices are zero-based.

```
N = csvread('ph.dat',0,2)
```

```
N =

   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 `dlmread` function.

Create a tab-delimited file named `num.txt` that contains the following data:

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

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

The sample file, `num.txt`, resides in your current folder.

Read the entire file. The file name is the only required input argument to the `dlmread` function. `dlmread` determines the delimiter from the formatting of the file.

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

```
M =

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

`M` is a 4-by-4 `double` array containing the data from the file.

Read only the rectangular block of data beginning from the second row, third column, in the file. When using `dlmread`, row and column indices are zero-based. When you specify

a specific range to read, you must also specify the delimiter. Use `'\t'` to indicate a tab delimiter.

```
N = dlmread('num.txt','\t',1,2)


N =

    45    74
    61    18
    79    41

```

`dlmread` returns a 3-by-2 `double` array.

Read only the first two columns. You can use spreadsheet notation to indicate the range, in this case, `'A1..B4'`.

```
P = dlmread('num.txt','\t','A1..B4')


P =

    95    89
    23    76
    61    46
    49     2

```

See Also
`csvread` | `dlmread`

More About
- "Ways to Import Text Files" on page 2-2

Import Mixed Data from Text Files

This example shows how to use the `readtable` function to import mixed data from a text file into a table. Then, it shows how to modify and analyze the data in the table.

Sample File Overview

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

```
Region,OutageTime,Loss,Customers,RestorationTime,Cause
SouthWest,01/20/2002 11:49,672,2902379,01/24/2002 21:58,winter storm
SouthEast,01/30/2002 01:18,796,336436,02/04/2002 11:20,winter storm
SouthEast,02/03/2004 21:17,264.9,107083,02/20/2004 03:37,winter storm
West,06/19/2002 13:39,391.4,378990,06/19/2002 14:27,equipment fault
```

The file contains six columns. The first line in the file lists column titles for the data. These are the column titles, along with a description of the data in that column:

- `Region`: Text value for one of five regions where each electrical outage occured
- `OutageTime`: Date and time at which the outage started, formatted as month/day/year hour:minute
- `Loss`: Numeric value indicating the total power loss for the outage
- `Customers`: Integer value indicating the number of customers impacted
- `RestorationTime`: Date and time at which power was restored, formatted as month/day/year hour:minute
- `Cause`: Category for the cause of the power outage, provided as text.

Specify Format of Data Fields

Create a string of format specifiers to describe the data in the text file. You can then pass the format specifiers to the `readtable` function to import the data. Because `outages.csv` contains six columns of data, create a string of six format specifiers, such as `'%f'` for a floating-point number, `'%C'` for a categorical value, and `'%D'` for a date and time value.

```
formatSpec = '%C%{MM/dd/yyyy HH:mm}D%f%f%{MM/dd/yyyy HH:mm}D%C';
```

The `formatSpec` string tells `readtable` to read the first and last columns in the file as categorical data, the second and fifth columns as formatted date and time data, and the third and fourth columns as floating-point values. For the `%{MM/dd/yyyy HH:mm}D`

specifiers, the text between the curly braces describes the format of the date and time data.

Read Text File

Call `readtable` to read the file. Use the `Delimiter` name-value pair argument to specify the delimiter. The default delimiter is a comma. Use the `Format` name-value pair argument along with the `formatSpec` value to describe the format of the data fields in the file.

```
T = readtable('outages.csv','Delimiter',',', ...
    'Format',formatSpec);
```

`readtable` returns a table containing the outage data.

View the first five rows and first four variables of the table.

```
T(1:5,1:4)
```

```
ans =

    Region          OutageTime          Loss       Customers
    _____    _____    _____    _____

    SouthWest    02/01/2002 12:18     458.98     1.8202e+06
    SouthEast    01/23/2003 00:49     530.14     2.1204e+05
    SouthEast    02/07/2003 21:15     289.4      1.4294e+05
    West         04/06/2004 05:44     434.81     3.4037e+05
    MidWest      03/16/2002 06:18     186.44     2.1275e+05
```

The type of data contained in the table is mixed. The first and last variables are `categorical` arrays, the second and fifth variables are `datetime` arrays, and the remaining variables are numeric data.

Modify Imported Data

Modify the format of the datetime columns in `T`.

```
T.OutageTime.Format = 'dd-MMM-yyyy HH:mm:ss';
T.RestorationTime.Format = 'dd-MMM-yyyy HH:mm:ss';
```

View the first five rows and first four variables of the table.

```
T(1:5,1:4)
```

```
ans =

     Region            OutageTime          Loss     Customers

    _____    _____    _____    _____


    SouthWest     01-Feb-2002 12:18:00    458.98    1.8202e+06
    SouthEast     23-Jan-2003 00:49:00    530.14    2.1204e+05
    SouthEast     07-Feb-2003 21:15:00    289.4     1.4294e+05
    West          06-Apr-2004 05:44:00    434.81    3.4037e+05
    MidWest       16-Mar-2002 06:18:00    186.44    2.1275e+05
```

Append to Imported Data

Calculate the duration of each electrical outage and append the data to the table.

```
T.Duration = T.RestorationTime - T.OutageTime;
```

View the first five rows of the data in the `Duration` column of `T`.

```
T.Duration(1:5)
```

```
ans =

    148:32:00
          NaN
    226:59:00
     00:26:00
     65:05:00
```

Sort Imported Data

Sort the table by the `OutageTime` variable. Then, view the first five rows and first four variables of the sorted table.

```
T = sortrows(T,'OutageTime','ascend');
T(1:5,1:4)
```

```
ans =
```

Region	OutageTime	Loss	Customers
SouthWest	01-Feb-2002 12:18:00	458.98	1.8202e+06
MidWest	05-Mar-2002 17:53:00	96.563	2.8666e+05
MidWest	16-Mar-2002 06:18:00	186.44	2.1275e+05
MidWest	26-Mar-2002 01:59:00	388.04	5.6422e+05
MidWest	20-Apr-2002 16:46:00	23141	NaN

See Also
readtable

More About

- "Import Dates and Times from Text Files" on page 2-9
- "Access Data in a Table"

Import Large Text File Data in Blocks

This example shows how to read small blocks of data from an arbitrarily large delimited text file using the `textscan` function and avoid memory errors. The first part of the example shows how to specify a constant block size. The second part of the example shows how to read and process each block of data in a loop.

Specify Block Size

Specify a constant block size, and then process each block of data within a loop.

Copy and paste the following text into a text editor to create a tab-delimited text file, `bigfile.txt`, in your current folder.

```
## 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
```

This file has commented lines beginning with ## , throughout the file. The data is arranged in five columns: The first column contains strings indicating timestamps. The second, third, and fourth columns contain numeric data indicating temperature, humidity and wind speed. The last column contains descriptive strings.

Define the size of each block to read from the text file. You do not need to know the total number of blocks in advance, and the number of rows of data in the file do not have to divide evenly into the block size.

Specify a block size of 5.

```
N = 5;
```

Open the file to read using the fopen function.

```
fileID = fopen('bigfile.txt');
```

fopen returns a file identifier, fileID, that the textscan function calls to read from the file. fopen positions a pointer at the beginning of the file, and each read operation changes the location of that pointer.

Describe each data field using format specifiers, such as '%s' for a string, '%d' for an integer, or '%f' for a floating-point number.

```
formatSpec = '%s %f %f %f %s';
```

In a while loop, call textscan to read each block of data. The file identifier, format specifer string, and the segment size (N), are the first three inputs to textscan. Ignore the commented lines using the CommentStyle name-value pair argument. Specify the tab delimiter using the Delimiter name-value pair argument. Then, process the data in the block. In this example, call scatter to display a scatter plot of temperature and humidity values in the block. The commands within the loop execute while the file pointer is not at the end of the file.

```
k = 0;
while ~feof(fileID)
 k = k+1;
 C = textscan(fileID,formatSpec,N,'CommentStyle','##','Delimiter','\t');
 figure
 scatter(C{2},C{3})
 title(['Temperature and Humidity, Block ',num2str(k)])
end
```

Temperature and Humidity, Block 1

Temperature and Humidity, Block 2

Temperature and Humidity, Block 3
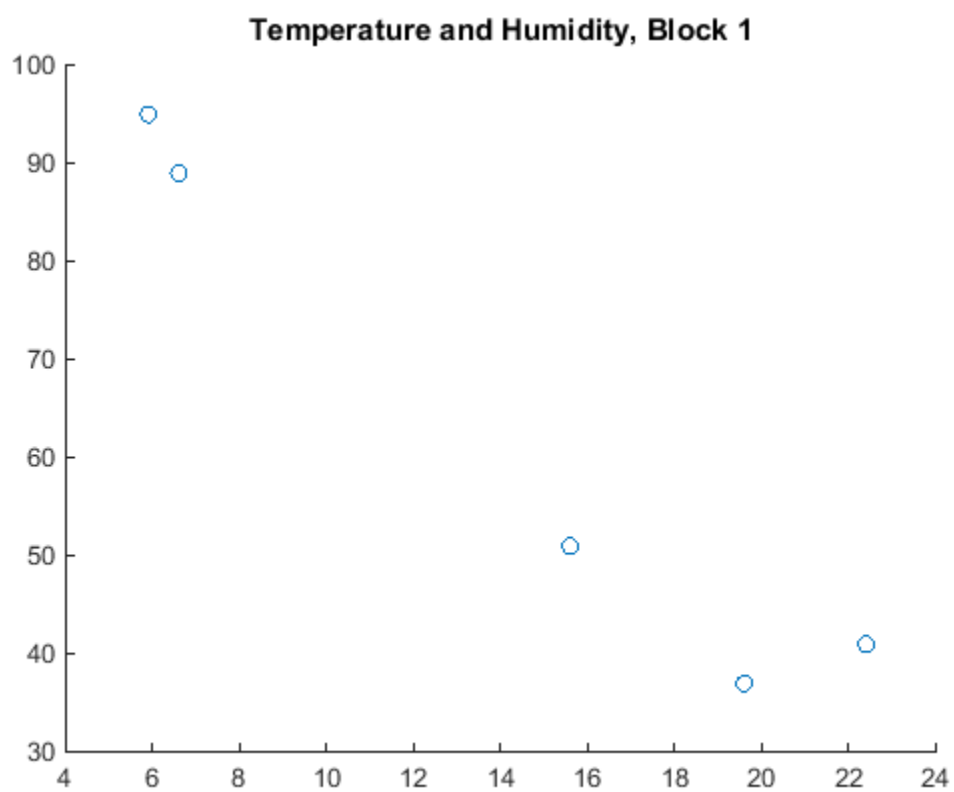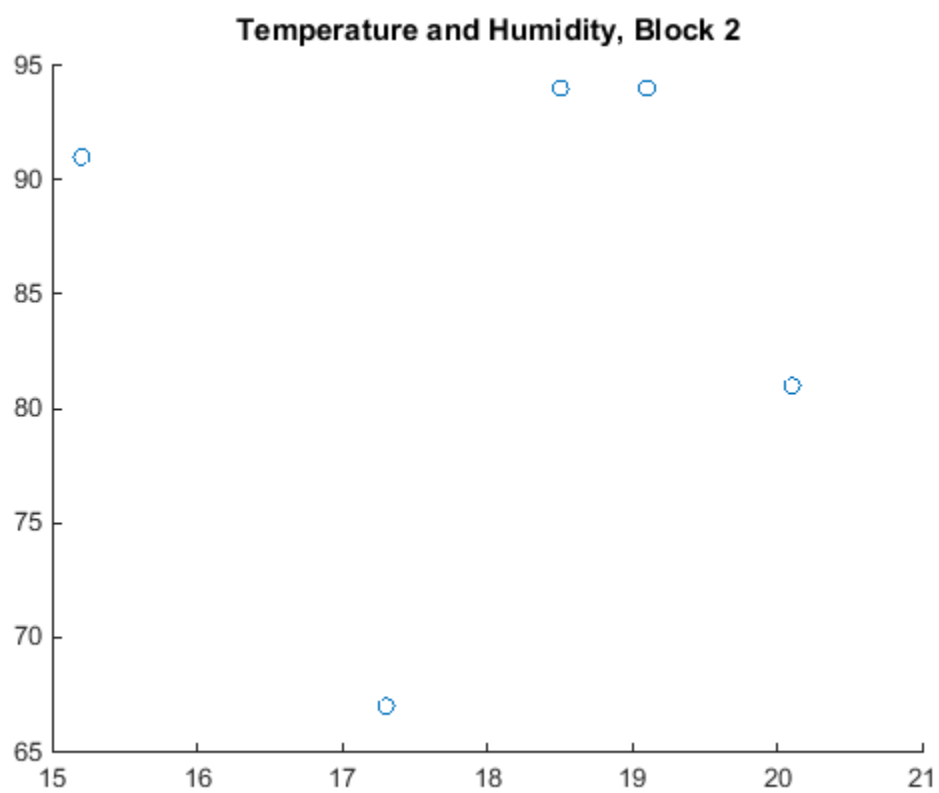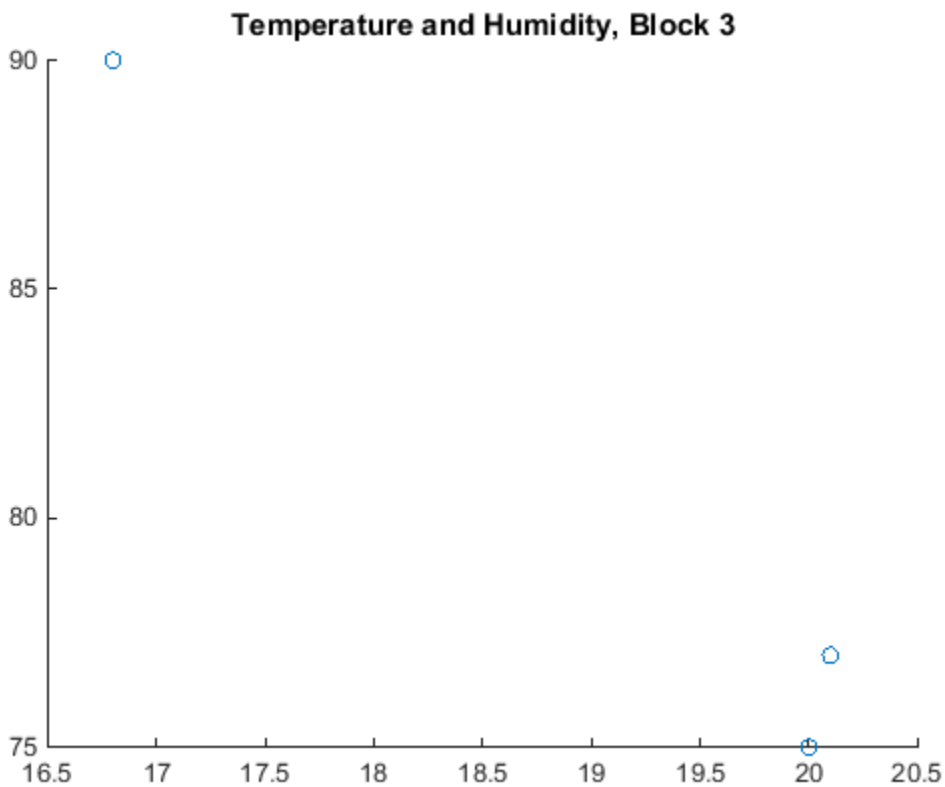
textscan reads data from `bigfile.txt` indefinitely, until it reaches the end of the file or until it cannot read a block of data in the format specified by `formatSpec`. For each complete block, `textscan` returns a 1-by-5 cell array. Because the sample file, `bigfile.txt`, contains 13 rows of data, `textscan` returns only 3 rows in the last block.

View the temperature values in the last block returned by `textscan`.

```
C{2}
```

```
ans =

    20.1000
    20.0000
    16.8000
```

Close the file.

```
fclose(fileID);
```

Read Data with Arbitrary Block Sizes

Read and process separately each block of data between commented lines in the file, `bigfile.txt`. The length of each block can be arbitrary. However, you must specify the number of lines to skip between blocks of data. In `bigfile.txt`, each block of data is preceded by two lines of comments.

Open the file for reading.

```
fileID = fopen('bigfile.txt');
```

Specify the format of the data you want to read. Tell `textscan` to ignore certain data fields by including `%*` in the format specifier string, `formatSpec`. In this example, skip the third and fourth columns of floating-point data using `'%*f'`.

```
formatSpec = '%s %f %*f %*f %s';
```

Read a block of data in the file. Use the `HeaderLines` name-value pair argument to instruct `textscan` to skip two lines before reading data.

```
D = textscan(fileID,formatSpec,'HeaderLines',2,'Delimiter','\t')

D =

    {7x1 cell}    [6x1 double]    {6x1 cell}
```

`textscan` returns a 1-by-3 cell array, `D`.

View the contents of the first cell in `D`.

```
D{1,1}

ans =

    '06-Sep-2013 01:00:00'
    '06-Sep-2013 05:00:00'
    '06-Sep-2013 09:00:00'
    '06-Sep-2013 13:00:00'
    '06-Sep-2013 17:00:00'
    '06-Sep-2013 21:00:00'
    '## B'
```

`textscan` stops reading after the text, `'## B'`, because it cannot read the subsequent text as a number, as specified by `formatSpec`. The file pointer remains at the position where `textscan` terminated.

Process the first block of data. In this example, find the maximum temperature value in the second cell of `D`.

```
maxTemp1 = max(D{1,2})

maxTemp1 =

    22.4000
```

Repeat the call to `textscan` to read the next block of data.

```
D = textscan(fileID,formatSpec,'HeaderLines',2,'Delimiter','\t')

D =

    {8x1 cell}    [7x1 double]    {7x1 cell}
```

Again, `textscan` returns a 1-by-3 cell array.

Find the maximum temperature value in this block of data.

```
maxTemp2 = max(D{1,2})

maxTemp2 =

    20.1000
```

Close the file.

```
fclose(fileID);
```

See Also
`fopen` | `textscan`

More About

- "Access Data in a Cell Array"
- "Moving within a File" on page 4-14

Import Data from a Nonrectangular Text File

This example shows how to import data from a nonrectangular file using the `textscan` function. When using `textscan`, your data does not have to be in a regular pattern of columns and rows, but it must be in a repeated pattern.

Create a file named `nonrect.dat` that contains the following (copy and paste into a text editor):

```
begin
v1=12.67
v2=3.14
v3=6.778
end
begin
v1=21.78
v2=5.24
v3=9.838
end
```

Open the file to read using the `fopen` function.

```
fileID = fopen('nonrect.dat');
```

`fopen` returns a file identifier, `fileID`, that `textscan` calls to read from the file.

Describe the pattern of the file data using format specifiers and delimiter parameters. Typical format specifiers include `'%s'` for a string, `'%d'` for an integer, or `'%f'` for a floating-point number. To import `nonrect.dat`, use the format specifier `'%*s'` to tell `textscan` to skip the strings `begin` and `end`. Include the literals `'v1='`, `'v2='`, and `'v3='` as part of the format specifiers, so that `textscan` ignores those strings as well.

```
formatSpec = '%*s v1=%f v2=%f v3=%f %*s';
```

Import the data using `textscan`. Pass the file identifier and `formatSpec` as inputs. Since each data field is on a new line, the delimiter is a newline character (`'\n'`). To combine all the floating-point data into a single array, set the `CollectOutput` name-value pair argument to `true`.

```
C = textscan(fileID,formatSpec,...
                'Delimiter', '\n', ...
                'CollectOutput', true)

C =
```

```
    [2x3 double]
```

textscan returns the cell array, C.

Close the file.

```
fclose(fileID);
```

View the contents of C.

```
celldisp(C)
```

```
C{1} =

    12.6700    3.1400    6.7780
    21.7800    5.2400    9.8380
```

See Also
textscan

More About
· "Access Data in a Cell Array"

Write to Delimited Data Files

In this section...
"Export Numeric Array to ASCII File" on page 2-27
"Export Table to Text File" on page 2-28
"Export Cell Array to Text File" on page 2-30

Export Numeric Array to ASCII File

- "Export Numeric Array to ASCII File Using `save`" on page 2-27
- "Export Numeric Array to ASCII File Using `dlmwrite`" on page 2-28

To export a numeric array as a delimited ASCII data file, you can use either the `save` function, specifying the `-ASCII` qualifier, or the `dlmwrite` function.

Both `save` and `dlmwrite` are easy to use. With `dlmwrite`, you can specify any character as a delimiter, and you can export subsets of an array by specifying a range of values.

However, `save -ascii` and `dlmwrite` do not accept cell arrays as input. To create a delimited ASCII file from the contents of a cell array, you can first convert the cell array to a matrix using the `cell2mat` function, and then call `save` or `dlmwrite`. Use this approach when your cell array contains only numeric data, and easily translates to a two-dimensional numeric array.

Export Numeric Array to ASCII File Using `save`

To export the array A, where

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

to a space-delimited ASCII data file, use the `save` function as follows:

```
save my_data.out A -ASCII
```

To view the file, use the `type` function:

```
type my_data.out
```

```
1.0000000e+000   2.0000000e+000   3.0000000e+000   4.0000000e+000
5.0000000e+000   6.0000000e+000   7.0000000e+000   8.0000000e+000
```

When you use `save` to write a character array to an ASCII file, it writes the ASCII equivalent of the characters to the file. For example, if you write the character string `'hello'` to a file, `save` writes the values

```
104 101 108 108 111
```

to the file in 8-digit ASCII format.

To write data in 16-digit format, use the `-double` option. To create a tab-delimited file instead of a space-delimited file, use the `-tabs` option.

Export Numeric Array to ASCII File Using `dlmwrite`

To export a numeric or character array to an ASCII file with a specified delimiter, use the `dlmwrite` function.

For example, to export the array `A`,

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

to an ASCII data file that uses semicolons as a delimiter, use this command:

```
dlmwrite('my_data.out',A, ';')
```

To view the file, use the `type` function:

```
type my_data.out

1;2;3;4
5;6;7;8
```

By default, `dlmwrite` uses a comma as a delimiter. You can specify a space (`' '`) or other character as a delimiter. To specify no delimiter, use empty quotation marks (`''`).

Export Table to Text File

This example shows how to export a table to a text file, using the `writetable` function.

Create a sample table, `T`, for exporting.

```
Name = {'M4';'M5';'M6';'M8';'M10'};
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,'RowNames',Name)
```

```
T =

              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
```

The table has both column headings and row names.

Export the table, `T`, to a text file named `tabledata.txt`.

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

View the file.

```
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
```

By default, `writetable` writes comma-separated data, includes table variable names as column headings, and does not write row names.

Export table `T` to a tab-delimited text file named `tabledata2.txt` and write the row names in the first column of the output. Use the `Delimiter` name-value pair argument to specify a tab delimiter, and the `WriteRowNames` name-value pair argument to include row names.

```
writetable(T,'tabledata2.txt','Delimiter','\t','WriteRowNames',true)
```

View the file.

```
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

Export Cell Array Using fprintf

This example shows how to export a cell array to a text file, using the `fprintf` function.

Create a sample cell array, `C`, for exporting.

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

```
C =

    'Atkins'    [32]    [77.3000]    'M'
    'Cheng'     [30]    [99.8000]    'F'
    'Lam'       [31]    [80.2000]    'M'
```

Open a file named `celldata.dat` for writing.

```
fileID = fopen('celldata.dat','w');
```

`fopen` returns a file identifier, `fileID`, that `fprintf` calls to write to the file.

Describe the pattern of the file data using format specifiers. Typical format specifiers include `'%s'` for a string, `'%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'`).

```
formatSpec = '%s %d %2.1f %s\n';
```

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.

Determine the size of C. Then, export one row of data at a time using the fprintf function.

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

fprintf writes a space-delimited file.

Close the file.

```
fclose(fileID);
```

View the file.

```
type celldata.dat
```

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

Convert Cell Array to Table for Export

This example shows how to convert a cell array of mixed text and numeric data to a table before writing the data to a text file. Tables are suitable for column-oriented or tabular data. You then can write the table to a text file using the writetable function.

Convert the cell array, C, from the previous example, to a table using the cell2table function. Add variable names to each column of data using the VariableNames name-value pair argument.

```
T = cell2table(C,'VariableNames',{'Name','Age','Result','Gender'});
```

Write table T to a text file.

```
writetable(T,'tabledata.dat')
```

View the file.

```
type tabledata.dat
```

```
Name,Age,Result,Gender
```

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

See Also
dlmwrite | fprintf | save | type | writetable

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.

3

Spreadsheets

Ways to Import Spreadsheets

In this section...
"Import Data from Spreadsheets" on page 3-2
"Paste Data from Clipboard" on page 3-2

Import Data from Spreadsheets

You can import data from spreadsheet files into MATLAB interactively, using the Import Tool, or programmatically, using an import function.

This table compares the primary import options for spreadsheet files.

Import Option	Description	For More Information, See...
Import Tool	Import a worksheet or range to column vectors, a matrix, a cell array, or a table. You can generate code to repeat the operation on multiple similar files.	"Select Spreadsheet Data Using Import Tool" on page 3-4
readtable	Import a worksheet or range to a table.	"Import a Worksheet or Range" on page 3-7
xlsread	Import a worksheet or range to numeric and cell arrays.	
importdata	Import one or more worksheets in a file to a structure array.	"Import All Worksheets from a File" on page 3-11

Some import options require that your system includes Excel for Windows. For more information, see "System Requirements for Importing Spreadsheets" on page 3-14.

Paste Data from Clipboard

Paste data from the clipboard into MATLAB using 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`.

Select Spreadsheet Data Using Import Tool

In this section...
"Select Data Interactively" on page 3-4
"Import Data from Multiple Spreadsheets" on page 3-6

Select Data Interactively

This example shows how to import data from a spreadsheet into the workspace with the Import Tool. The worksheet in this example includes three columns of data labeled `Station`, `Temp`, and `Date`:

```
Station   Temp    Date
12          98    9/22/2013
13           x    10/23/2013
14          97    12/1/2013
```

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. 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
Column vectors	Import each column of the selected data as an individual `m`-by-1 vector.
Matrix	Import selected data as an `m`-by-`n` numeric 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.

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.

If you choose to import as a matrix or 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. You can see how your data will be imported when you place the cursor over individual cells.
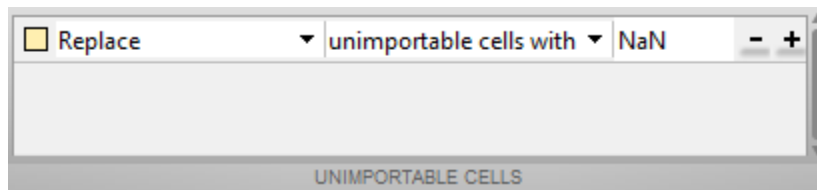
You can add, remove, reorder, or edit rules, such as changing the replacement value from NaN to another value. All rules apply to the imported data only, and do not change the data in the file. You must specify rules any time the range includes nonnumeric data and you are importing into a matrix or numeric 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 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** 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`.

Import a Worksheet or Range

In this section...

Read Column-Oriented Data into Table

This example shows how to import mixed numeric and text data from a spreadsheet into a table, using the `readtable` function. Tables are suitable for column-oriented or tabular data. You can store variable names or row names along with the data in a single container.

This example uses a sample spreadsheet file, `climate.xlsx`, that contains the following numeric and text data in a worksheet called `Temperatures`.

```
Time  Temp  Visibility
12  98  clear
13  99  clear
14  97  partly cloudy
```

Create the sample file for reading.

```
d = {'Time','Temp','Visibility';
     12 98 'clear';
     13 99 'clear';
     14 97 'partly cloudy'};

xlswrite('climate.xlsx',d,'Temperatures');
```

`xlswrite` warns that it has added a worksheet.

Call `readtable` to read all the data in the worksheet called `Temperatures`. 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('climate.xlsx','Sheet','Temperatures')

T =
```

```
Time     Temp        Visibility
____     ____    _____

12       98      'clear'
13       99      'clear'
14       97      'partly cloudy'
```

`readtable` returns a 3-by-3 table. By default, `readtable` reads the first row of the worksheet as variable names for the table.

Read only the first two columns of data by specifying a range, `'A1:B4'`.

```
cols = readtable('climate.xlsx','Sheet','Temperatures','Range','A1:B4')
```

```
cols =

   Time     Temp
   ____     ____

   12       98
   13       99
   14       97
```

`readtable` returns a 3-by-2 table.

Read Numeric and Text Data into Arrays

This example shows how to import mixed numeric and text data into separate arrays in MATLAB, using the `xlsread` function.

This example uses a sample spreadsheet file, `climate.xlsx`, that contains the following data in a worksheet called `Temperatures`.

```
Time     Temp
  12       98
  13       99
  14       97
```

Create the sample file for reading.

```
d = {'Time', 'Temp';
     12 98;
     13 99;
     14 97}
```

```
xlswrite('climate2.xlsx',d,'Temperatures');
```

`xlswrite` warns that it has added a worksheet.

Import only the numeric data into a matrix, using `xlsread` with a single output argument. `xlsread` ignores any leading row or column of text in the numeric result.

```
num = xlsread('climate2.xlsx','Temperatures')

num =
    12    98
    13    99
    14    97
```

`xlsread` returns the numeric array, `num`.

Alternatively, import both numeric data and text data, by specifying two output arguments in the call to `xlsread`.

```
[num,headertext] = xlsread('climate2.xlsx','Temperatures')

num =
    12    98
    13    99
    14    97

headertext =
    'Time'    'Temp'
```

`xlsread` returns the numeric data in the array, `num`, and the text data in the cell array, `headertext`.

Read only the first row of data by specifying a range, `'A2:B2'`.

```
row1 = xlsread('climate2.xlsx','Temperatures','A2:B2')

row1 =
    12    98
```

Get Information about a Spreadsheet

To determine whether a file contains a readable Excel spreadsheet, use the `xlsfinfo` function. For readable files, `xlsfinfo` returns a nonempty string, such as `'Microsoft Excel Spreadsheet'`. Otherwise, it returns an empty string (`''`).

You also can use `xlsfinfo` to identify the names of the worksheets in the file, and to obtain the file format reported by Excel. For example, retrieve information about the spreadsheet file, `climate2.xlsx`, created in the previous example:

```
[type, sheets] = xlsfinfo('climate2.xlsx')

type =
Microsoft Excel Spreadsheet
sheets =
    'Sheet1'    'Sheet2'    'Sheet3'    'Temperatures'
```

See Also
`readtable` | `xlsfinfo` | `xlsread`

More About

- "Import and Export Dates to Excel Files" on page 3-15
- "Access Data in a Table"

Import All Worksheets from a File

Import Numeric Data from All Worksheets

This example shows how to import worksheets in an Excel file that contains only numeric data (no row or column headers, and no inner cells with text) into a structure array, using the `importdata` function.

Create a sample spreadsheet file for importing by writing an array of numeric data to the first and second worksheets in a file called `numdata.xlsx`.

```
xlswrite('numdata.xlsx',rand(5,5),1);
xlswrite('numdata.xlsx',rand(5,6),2);
```

Import the data from all worksheets in `numdata.xlsx`.

```
S = importdata('numdata.xlsx')

S =

    Sheet1: [5x5 double]
    Sheet2: [5x6 double]
```

`importdata` returns a structure array, `S`, with one field for each worksheet with data.

Import Data and Headers from All Worksheets

This example shows how to read numeric data and text headers from all worksheets in an Excel file into a nested structure array, using the `importadata` function.

This example uses a sample spreadsheet file, `testdata.xlsx`, that contains the following data in the first worksheet, and similar data in the second worksheet.

```
   Time    Temp
     12      98
     13      99
```

```
    14     97
```

Write a sample file, `testdata.xlsx`, for reading. Write an array of sample data, `d1`, to the first worksheet in the file and a second array, `d2`, to the second worksheet.

```matlab
d1 = {'Time','Temp';
      12 98;
      13 99;
      14 97};

d2 = {'Time','Temp';
      12 78;
      13 77;
      14 78};

xlswrite('testdata.xlsx',d1,1);
xlswrite('testdata.xlsx',d2,2);
```

Read the data from all worksheets in `testdata.xlsx`.

```matlab
climate = importdata('testdata.xlsx')

climate =

         data: [1x1 struct]
     textdata: [1x1 struct]
    colheaders: [1x1 struct]
```

`importdata` returns a nested structure array, `climate`, with three fields, `data`, `textdata`, and `colheaders`. Structures created from Excel files with row headers include the field `rowheaders` instead of `colheaders`.

View the contents of the structure array named `data`.

```matlab
climate.data

ans =

    Sheet1: [3x2 double]
    Sheet2: [3x2 double]
```

`climate.data` contains one field for each worksheet with numeric data.

View the data in the worksheet named `Sheet1`.

```
climate.data.Sheet1

ans =

    12    98
    13    99
    14    97
```

The field, `Sheet1`, contains the numeric data from the first worksheet in the file.

View the column headers in each sheet.

```
headers = climate.colheaders

headers =

    Sheet1: {'Time'  'Temp'}
    Sheet2: {'Time'  'Temp'}
```

Both the worksheets named `Sheet1` and `Sheet2` have the column headers, `Time` and `Temp`.

See Also

`importdata`

More About

- "Import and Export Dates to Excel Files" on page 3-15

System Requirements for Importing Spreadsheets

In this section...
"Importing Spreadsheets with Excel for Windows" on page 3-14
"Importing Spreadsheets Without Excel for Windows" on page 3-14

Importing Spreadsheets with Excel for Windows

If your system has Excel for Windows installed, including the COM server (part of the typical installation of Excel):

- All MATLAB import options support XLS, XLSX, XLSB, XLSM, XLTM, and XLTX formats.
- `xlsread` also imports HTML-based formats.
- `xlsread` includes an option to open Excel and select the range of data interactively. To use this option, call `xlsread` with the following syntax:

  mydata = xlsread(*filename*, -1)
- If you have Excel 2003 installed, but want to read a 2007 format (such as XLSX, XLSB, or XLSM), install the Office 2007 Compatibility Pack.
- If you have Excel 2010, all MATLAB import options support ODS files.

Note: Large files in XLSX format sometimes load slowly. For better import and export performance, Microsoft recommends that you use the XLSB format.

Importing Spreadsheets Without Excel for Windows

If your system does not have Excel for Windows installed, or the COM server is not available:

- All MATLAB import options read XLS, XLSX, XLSM, XLTM, and XLTX files.

Import and Export Dates to Excel Files

In this section...
"MATLAB and Excel Dates" on page 3-15
"Import Dates on Systems with Excel for Windows" on page 3-15
"Import Dates on Systems Without Excel for Windows" on page 3-16
"Export Dates to Excel File" on page 3-17

MATLAB and Excel Dates

Microsoft Excel software can represent dates as character strings or numeric values. For example, in Excel for Windows, you can express April 1, 2012 as the character string `'04/01/12'` or as the numeric value `41000`. The best way to represent dates in MATLAB is to use datetime values. However, MATLAB functions import dates from Excel files as character strings or numeric values. Additionally, you might want to export text or numeric dates generated by existing code. Use the `datetime` function to convert date strings and serial date numbers to datetime values.

Both Excel and MATLAB represent numeric dates as a number of serial days elapsed from a specific reference date, but the applications use different reference dates.

This table lists the default reference dates for MATLAB and Excel. For more information about default reference dates in Excel, see the Excel help.

Application	Reference Date
MATLAB serial date number	January 0, 0000
Excel for Windows	January 1, 1900
Excel for the Macintosh	January 2, 1904

Import Dates on Systems with Excel for Windows

This example shows how to import an Excel file containing dates into a MATLAB table on a system with Excel for Windows.

Create the hypothetical file `weight.xls` that contains the following data.

```
Date      Weight
```

```
10/31/96    174.8
11/29/96    179.3
12/30/96    190.4
01/31/97    185.7
```

Import the data using `readtable`.

```
T = readtable('weight.xls')

T =

        Date          Weight
     _____    _____

     '10/31/1996'     174.8
     '11/29/1996'     179.3
     '12/30/1996'     190.4
     '1/31/1997'      185.7
```

On systems with Excel for Windows, the `Date` variable of the output table is a cell array of date strings.

Convert the date strings in `T` to datetime values, using the `datetime` function.

```
T.Date = datetime(T.Date,'InputFormat','MM/dd/yyyy')

T =

        Date         Weight
     _____    _____

     31-Oct-1996     174.8
     29-Nov-1996     179.3
     30-Dec-1996     190.4
     31-Jan-1997     185.7
```

Import Dates on Systems Without Excel for Windows

This example shows how to import an Excel file containing dates into a MATLAB table on a system without Excel for Windows.

Create the hypothetical file `weight.xls` that contains the following data.

```
Date        Weight
10/31/96    174.8
```

```
11/29/96    179.3
12/30/96    190.4
01/31/97    185.7
```

Import the data using `readtable`.

```
T = readtable('weight.xls')

T =

    Date      Weight
    _____     _____

    35369     174.8
    35398     179.3
    35429     190.4
    35461     185.7
```

The `Date` variable of the output table is an array of MATLAB serial date numbers.

Convert the numeric dates in `T` to MATLAB datetime values.

• If the file uses the 1900 date system (the default in Excel for Windows), type:

```
T.Date = datetime(T.Date,'ConvertFrom','excel')

T =

            Date            Weight
    _____    _____

    31-Oct-1996 00:00:00    174.8
    29-Nov-1996 00:00:00    179.3
    30-Dec-1996 00:00:00    190.4
    31-Jan-1997 00:00:00    185.7
```

• If the file uses the 1904 date system (the default in Excel for the Macintosh), type:

```
T.Date = datetime(T.Date,'ConvertFrom','excel1904');
```

Excel date numbers are rounded to the nearest microsecond.

Export Dates to Excel File

• "Export Datetime Values" on page 3-18

• "Convert Numeric Dates to Datetime Values Before Export" on page 3-18

Export Datetime Values

This example shows how to export datetime values to an Excel file using the `writetable` function.

Create a row vector of `datetime` values.

```
d = datetime({'11/04/1997','12/02/1997','01/05/1998','02/01/1998'},...
    'InputFormat','MM/dd/yyyy');
```

Create a row vector of sample data.

```
weights = [174.8 179.3 190.4 185.7];
```

Use the `table` function to create a table with columns that contain the data in `d` and `weights`. Use the `'VariableNames'` name-value pair argument to specify variable names in the table.

```
T = table(d',weights','VariableNames',{'Date','Weight'})

T =

      Date        Weight

    _____    _____

    04-Nov-1997    174.8
    02-Dec-1997    179.3
    05-Jan-1998    190.4
    01-Feb-1998    185.7
```

Export the table to a file named `myfile.xls` using the `writetable` function.

```
writetable(T,'myfile.xls');
```

The Excel file contains the dates represented as character strings.

Convert Numeric Dates to Datetime Values Before Export

This example shows how to export convert numeric dates to datetime values before exporting to an Excel file using the `writetable` function.

Create a matrix that contains dates represented as numeric values in the first column.

```
wt = [729698 174.8; ...
      729726 175.3; ...
      729760 190.4; ...
      729787 185.7];
```

Convert the matrix to a table. Use the `'VariableNames'` name-value pair argument to specify variable names in the table.

```
T = array2table(wt,'VariableNames',{'Date','Weight'})
```

```
T =

       Date        Weight
    _____      _____

     7.297e+05      174.8
    7.2973e+05      175.3
    7.2976e+05      190.4
    7.2979e+05      185.7
```

Convert the MATLAB serial date numbers in the `Weight` variable to datetime values.

```
T.Date = datetime(T.Date,'ConvertFrom','datenum')
```

```
T =

            Date            Weight
    _____    _____

    04-Nov-1997 00:00:00    174.8
    02-Dec-1997 00:00:00    175.3
    05-Jan-1998 00:00:00    190.4
    01-Feb-1998 00:00:00    185.7
```

Export the data using the `writetable` function.

```
writetable(T,'myfile.xls')
```

See Also
datetime | readtable | writetable

More About
- "Export to Excel Spreadsheets" on page 3-20

Export to Excel Spreadsheets

In this section...

Write Tabular Data to Spreadsheet File

This example shows how to export a table in the workspace to a Microsoft Excel spreadsheet file, using 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`.

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 =

    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`. Specify the portion of the worksheet to write to, using the `Range` name-value pair argument.

```
filename = 'patientdata.xlsx';
```

```
writetable(T,filename,'Sheet',1,'Range','D1')
```

By default, `writetable` writes the table variable names as column headings in the spreadsheet file.

Write table `T` to the second sheet in the file, but do not write the table variable names.

```
writetable(T,filename,'Sheet',2,'WriteVariableNames',false)
```

Write Numeric and Text Data to Spreadsheet File

This example shows how to export a numeric array and a cell array to a Microsoft Excel spreadsheet file, using the `xlswrite` function. 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, `xlswrite` writes your matrix data to the first worksheet in the file, starting at cell `A1`.

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';
```

```
xlswrite(filename,A,1,'E1:I5')
```

Write the cell array, C, to a rectangular region that starts at cell B2 on a worksheet named Temperatures in the file. When you specify the sheet, you can specify a range using only the first cell.

```
xlswrite(filename,C,'Temperatures','B2');
```

xlswrite displays a warning because the worksheet, Temperatures, does not already exist in the file.

Disable Warning When Adding New Worksheet

If the target worksheet does not already exist in the file, the writetable and xlswrite functions display the following warning:

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

You can disable these warnings with this command:

```
warning('off','MATLAB:xlswrite:AddSheet')
```

Supported Excel File Formats

writetable and xlswrite can write to any file format recognized by your version of Excel for Windows. If you have Excel 2003 installed, but want to write to a 2007 format (such as XLSX, XLSB, or XLSM), you must install the Office 2007 Compatibility Pack.

Note: If you are using a system that does not have Excel for Windows installed, writetable and xlswrite write data to a comma-separated value (CSV) file.

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 or xlswrite. 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 "Getting Started with COM".

See Also

writetable | xlswrite

More About

·   "Import and Export Dates to Excel Files" on page 3-15

4

Low-Level File I/O

Import Text Data Files with Low-Level I/O

In this section...
"Overview" on page 4-2
"Reading Data in a Formatted Pattern" on page 4-3
"Reading Data Line-by-Line" on page 4-5
"Testing for End of File (EOF)" on page 4-6
"Opening Files with Different Character Encodings" on page 4-9

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 "Ways to 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 4-3.

- `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 4-5.

- `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 4-10.

For additional information, see:

- "Testing for End of File (EOF)" on page 4-6
- "Opening Files with Different Character Encodings" on page 4-9

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 a string, `'%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 2 strings, 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 in string vectors: `fgetl` and `fgets`. The `fgets` function copies the newline character to the output string, but `fgetl` does not.

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

```
function y = litcount(filename, literal)
% Search for number of string matches per 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 the string `'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 string.

For example, the function `litcount` described in "Reading Data Line-by-Line" on page 4-5 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 string. 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, `fopen` opens files for processing using the default encoding for your system. 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" on page 4-10
"Reading Binary Data in a File" on page 4-11
"Reading Portions of a File" on page 4-13
"Reading Files Created on Other Systems" on page 4-16
"Opening Files with Different Character Encodings" on page 4-16

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 4-3.

- `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 4-5.

- `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 4-11.

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 4-11. Consider combining this method with "Testing for End of File" on page 4-13.
- Move to a specific location in a file to begin reading. For more information, see "Moving within a File" on page 4-14.
- Skip a certain number of bytes or bits after each element read. For an example, see "Write and Read Complex Numbers" on page 4-28.

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.

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.

The encoding scheme determines the number of bytes required to read or write `char` values. For example, US-ASCII characters always use 1 byte, but UTF-8 characters use

up to 4 bytes. MATLAB automatically processes the required number of bytes for each char value based on the specified encoding scheme. However, if you specify a uchar precision, MATLAB processes each byte as uint8, regardless of the specified encoding.

If you do not specify an encoding scheme, fopen opens files for processing using the default encoding for your system. 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.

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

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 strings. 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 string in the call to `fopen`. Possible permission strings 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 string 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);
```

4-21

```
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, `fopen` opens files for processing using the default encoding for your system. 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 Strings"
- "Write to Delimited Data Files" on page 2-27

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 4-18.

- fwrite, which writes a stream of binary data to a file. For more information, see "Write Binary Data to a File" on page 4-24.

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 string in the call to `fopen`. Possible permission strings 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 =

    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 =

    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

4-27

```
    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.

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.

The encoding scheme determines the number of bytes required to read or write `char` values. For example, US-ASCII characters always use 1 byte, but UTF-8 characters use up to 4 bytes. MATLAB automatically processes the required number of bytes for each `char` value based on the specified encoding scheme. However, if you specify a `uchar` precision, MATLAB processes each byte as `uint8`, regardless of the specified encoding.

If you do not specify an encoding scheme, `fopen` opens files for processing using the default encoding for your system. 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.

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 =

  Columns 1 through 4

   0.8147 + 0.7577i   0.0975 + 0.7060i   0.1576 + 0.8235i   0.1419 + 0.4387i
   0.9058 + 0.7431i   0.2785 + 0.0318i   0.9706 + 0.6948i   0.4218 + 0.3816i
   0.1270 + 0.3922i   0.5469 + 0.2769i   0.9572 + 0.3171i   0.9157 + 0.7655i
   0.9134 + 0.6555i   0.9575 + 0.0462i   0.4854 + 0.9502i   0.7922 + 0.7952i
   0.6324 + 0.1712i   0.9649 + 0.0971i   0.8003 + 0.0344i   0.9595 + 0.1869i

  Column 5

   0.6557 + 0.4898i
   0.0357 + 0.4456i
   0.8491 + 0.6463i
   0.9340 + 0.7094i
   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 Follwed 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 4-14

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 5-3 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.

The following section provides a step-by-step example of using Tiff object methods and properties to read subimages from a TIFF file. 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

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 can also 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 can then 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 Subdirectories in a TIFF File" on page 5-9, 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 returns an array of byte offsets that specify the location of subIFDs:

```
offsets = t.getTag('SubIFD')
```

3   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:

```
t.setSubDirectory(offsets(1));
```

4   Read the image data from the current IFD (the first subIFD) as you would with any other IFD in the file:

```
subimage_one = t.read();
```

5   View the first subimage:

```
imagesc(subimage_one)
```

6   To view the second subimage, call the `setSubDirectory` method again, specifying the byte offset of the second subIFD:

```
t.setSubDirectory(offsets(2));
```

7   Read the image data from the current IFD (the second subIFD) as you would with any other IFD in the file:

```
subimage_two = t.read();
```

8   View the second subimage:

```
imagesc(subimage_two)
```

9   Close the `Tiff` object.

```
t.close();
```

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 5-5 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'
t.setTag(tagstruct)
```

For information about supported TIFF tags and how to set their values, see "Setting Tag Values" on page 5-11. 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.

```
t.write(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:

```
t.close();
```

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 5-6. 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 = t.getTag('ImageWidth');
height = t.getTag('RowsPerStrip');
numSamples = t.getTag('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.

```
t.writeEncodedStrip(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.

```
t.close();
```

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 5-6. 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 = t.getTag('Artist');
```

3   Add the `Artist` tag using the `setTag` method.

```
t.setTag('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.

```
t.rewriteDirectory();
```

5   Close your connection to the file by closing the `Tiff` object.

```
    t.close();
```
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');

t.getTag('Artist')

ans =

Pablo Picasso

t.close();
```

Creating Subdirectories in a TIFF File

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
t.setTag(tagstruct)
```

For information about supported TIFF tags and how to set their values, see "Setting Tag Values" on page 5-11. 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.

```
t.write(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.

```
t.writeDirectory();
```

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'
t.setTag(tagstruct2)
```

7 Write the image data and metadata to the subdirectory using the `write` method of the `Tiff` object.

```
t.write(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.

```
t.writeDirectory();
```

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'
t.setTag(tagstruct3)
```

10 Write the image data and metadata to the subdirectory using the `write` method of the `Tiff` object:

```
t.write(img_third);
```

11 Close your connection to the file by closing the `Tiff` object:

```
t.close();
```

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 `Tiff` object properties to specify the supported values for these tags. For example, use `Tiff.Compression.JPEG` to specify JPEG compression. See the Tiff class reference page for a full list of properties.

Table 1: Supported TIFF Tags

TIFF Tag	Class	Size	Supported Values	Notes
Artist	char	1xN		

TIFF Tag	Class	Size	Supported Values	Notes
BitsPerSample	double	1x1	1,8,16,32,64	See Table 2
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.
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 UnassociatedAlph 2	See Table 4.
FillOrder	double	1x1		
GeoAsciiParamsTag	char	1xN		
GeoDoubleParamsTag	double	1xN		
GeoKeyDirectoryTag	double	Nx4		

TIFF Tag	Class	Size	Supported Values	Notes
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		
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	

TIFF Tag	Class	Size	Supported Values	Notes
			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.
Photoshop	uint8	1xN		
PlanarConfiguration	double	1x1	Chunky: 1 Separate: 2	
PrimaryChromaticities	double	1x6		
ReferenceBlackWhite	double	1x6		
ResolutionUnit	double	1x1		
RICHTIFFIPTC	uint8	1xN		
RowsPerStrip	double	1x1		
SampleFormat	double	1x1	Uint: 1 Int: 2 IEEEFP: 3	See Table 2
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	

TIFF Tag	Class	Size	Supported Values	Notes
`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`

`XMP`	`char`	`1xn`		N>5
`XPostion`	`double`	`1x1`		
`XResolution`	`double`	`1x1`		

TIFF Tag	Class	Size	Supported Values	Notes
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` or `3x2^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

Photometric Values	BitsPerSample Values				
	1	8	16	32	64
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

Compression Values	BitsPerSample 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	Uint	Uint	IEEEFP

	BitsPerSample Values				
Compression Values	**1**	**8**	**16**	**32**	**64**
		Int	Int	Int 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+

[1] When you specify more than the expected number of samples per pixel (n+), you must set the `ExtraSamples` tag accordingly.

6

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 =

    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 =

         name: 'Time'
     datatype: 'cdf_epoch'
  numElements: 1
```

```
           dims: []
    recVariance: 1
    dimVariance: []


info  = cdflib.inquireVar(cdfid,1)


info =

          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 =

        2001
           1
           1
           0
           0
           0
```

Read Global Attribute From File

Determine which attributes in the CDF file are global.

```
info = cdflib.inquireAttr(cdfid,0)


info =

        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            25248  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 =

  01-Jan-2001 00:00:00
```

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 =

              Filename: 'example.cdf'
           FileModDate: '10-May-2010 21:35:00'
              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 =

  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            25248  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 as text strings in a cell array. 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                  232  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            25248  cell
  data_combined        1x6             8320  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              25248  cell
data_combined        1x6               8320  cell
data_datenums       24x6              21024  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.

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 =

     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 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 character string that specifies the 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`

NetCDF Data Type	MATLAB Class
`'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

`netcdf`

External Websites

·   NetCDF website

Import NetCDF Files and OPeNDAP Data

In this section...
"MATLAB NetCDF Capabilities" on page 6-15
"Read from NetCDF File Using High-Level Functions" on page 6-15
"Find All Unlimited Dimensions in NetCDF File" on page 6-18
"Read from NetCDF File Using Low-Level Functions" on page 6-19

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 6-2.

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 | netcdf

More About
·   "Map NetCDF API Syntax to MATLAB Syntax" on page 6-13

External Websites
·   NetCDF C Interface

Resolve Errors Reading OPeNDAP Data

If you have trouble reading OPeNDAP data, consider the following:

- 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

In this section...
"MATLAB NetCDF Capabilities" on page 6-25
"Create New NetCDF File From Existing File or Template" on page 6-25
"Merge Two NetCDF Files" on page 6-27
"Write Data to NetCDF File Using Low-Level Functions" on page 6-29

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 6-10.

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
netcdf

More About
·   "Map NetCDF API Syntax to MATLAB Syntax" on page 6-13

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, `http://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" on page 6-35
"Using the High-Level HDF5 Functions to Import Data" on page 6-35
"Using the Low-Level HDF5 Functions to Import Data" on page 6-41

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 (`http://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 Programatically" on page 6-51.

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.

- `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.
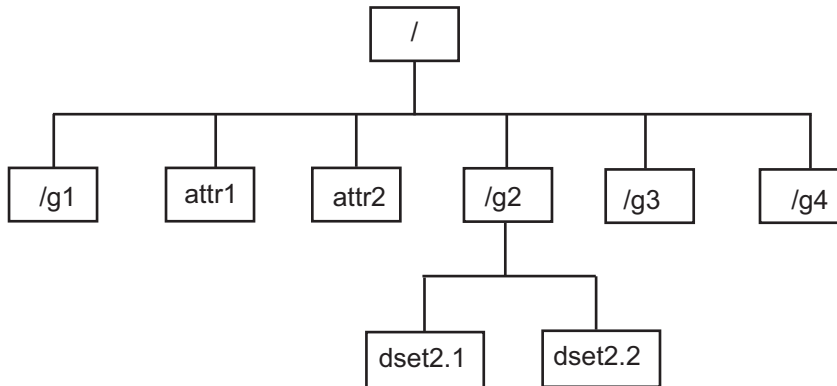
To get information about a data set, such as its name, dimensions, and data type, look at either of the structures returned in the `Datasets` field.

```
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

HDF5 Data Type	h5read Returns
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 strings
Enums	Cell array of strings, 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 strings.

```
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]
```

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 6-43.

Exporting to HDF5 Files

In this section...
"Overview" on page 6-42
"Using the MATLAB High-Level HDF5 Functions to Export Data" on page 6-42
"Using the MATLAB Low-Level HDF5 Functions to Export Data" on page 6-43

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 (`http://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 6-79.

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.

- `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 6-44
- "Map Between HDF5 Data Types and MATLAB Data Types" on page 6-46
- "Report Data Set Dimensions" on page 6-47
- "Write Data to HDF5 Data Set Using MATLAB Low-Level Functions" on page 6-47

- "Write a Large Data Set" on page 6-50
- "Preserve Correct Layout of Your Data" on page 6-50

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 http://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 text string for 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 text strings.

```
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 text strings, you must use a cell array to achieve the same result. The following code fragment provides an example of using a cell array 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 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 strings.

```
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 identifer 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 =

    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`

Import HDF4 Files Programatically

In this section...
"Overview" on page 6-51
"Using the MATLAB HDF4 High-Level Functions" on page 6-51

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 6-35.

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 6-57.

This section describes these high-level MATLAB HDF functions, including

- "Using hdfinfo to Get Information About an HDF4 File" on page 6-52
- "Using hdfread to Import Data from an HDF4 File" on page 6-52

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 6-54.

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 =

    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 =

    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 6-55

Import HDF4 Files Interactively

The HDF Import Tool is a graphical user interface that you can use to navigate through HDF4 or HDF-EOS files and import data from them. Importing data using the HDF Import Tool involves these steps:

The following sections provide more detail about each of these steps.

Step 1: Opening an HDF4 File in the HDF Import Tool

Open an HDF4 or HDF-EOS file in MATLAB using one of the following methods:

- On the **Home** tab, in the **Variable** section, click **Import Data**. If you select an HDF4 or HDF-EOS file, the MATLAB Import Wizard automatically starts the HDF Import Tool.

- Start the HDF Import Tool by entering the `hdftool` command at the MATLAB command line:

  ```
  hdftool
  ```

  This opens an empty HDF Import Tool. To open a file, click the **Open** option on the HDFTool **File** menu and select the file you want to open. You can open multiple files in the HDF Import Tool.

- Open an HDF or HDF-EOS file by specifying the file name with the `hdftool` command on the MATLAB command line:

  ```
  hdftool('example.hdf')
  ```

Viewing a File in the HDF Import Tool

When you open an HDF4 or HDF-EOS file in the HDF Import Tool, the tool displays the contents of the file in the Contents pane. You can use this pane to navigate within the file to see what data sets it contains. You can view the contents of HDF-EOS files as HDF data sets or as HDF-EOS files. The icon in the contents pane indicates the view, as illustrated in the following figure. Note that these are just two views of the same data.
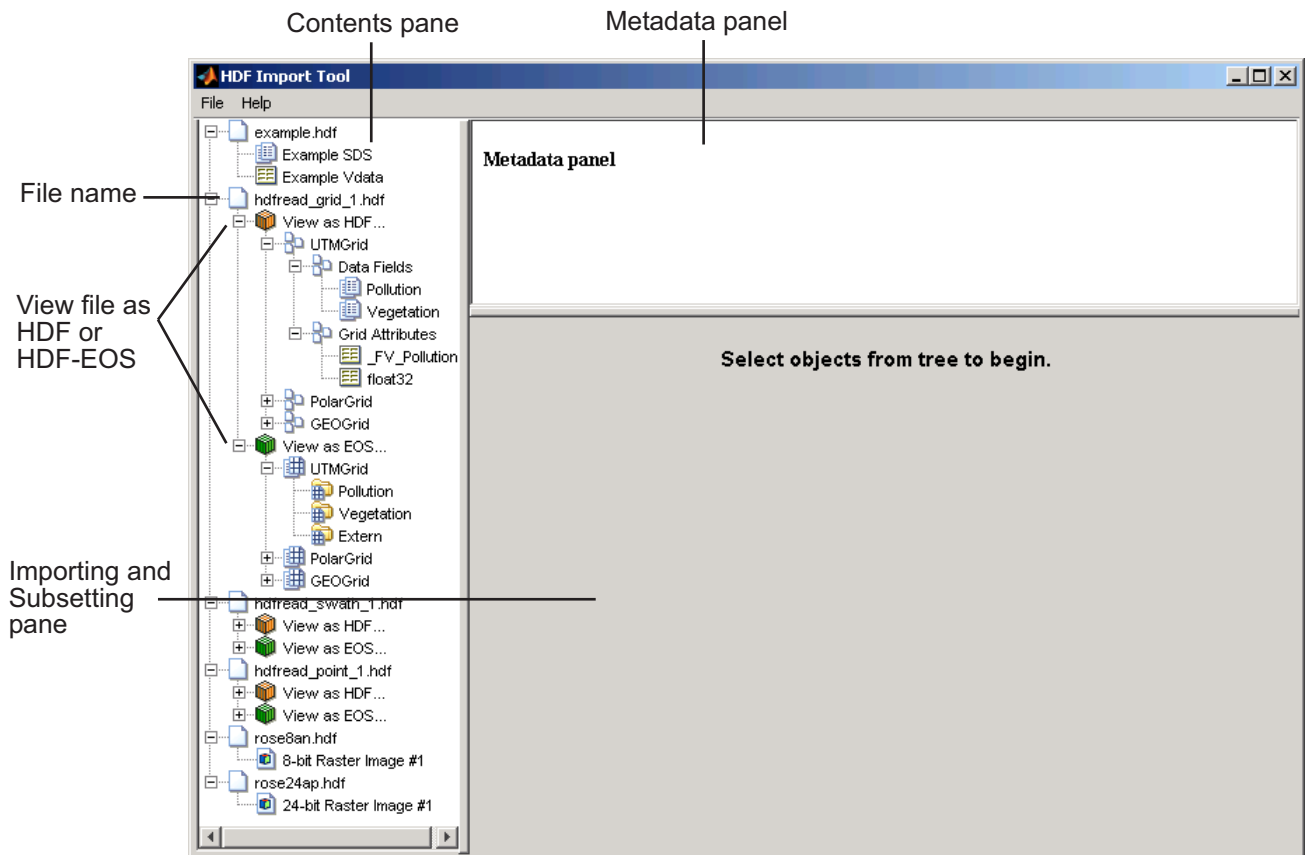
Step 2: Selecting a Data Set in an HDF File

To import a data set, you must first select the data set in the contents pane of the HDF Import Tool. Use the Contents pane to view the contents of the file and navigate to the data set you want to import.

For example, the following figure shows the data set Example SDS in the HDF file selected. Once you select a data set, the Metadata panel displays information about the data set and the importing and subsetting pane displays subsetting options available for this type of HDF object.

Step 3: Specifying a Subset of the Data (Optional)

When you select a data set in the contents pane, the importing and subsetting pane displays the subsetting options available for that type of HDF object. The subsetting options displayed vary depending on the type of HDF object. For more information, see "Using the HDF Import Tool Subsetting Options" on page 6-65.

Step 4: Importing Data and Metadata

To import the data set you have selected, click the **Import** button, bottom right corner of the Importing and Subsetting pane. Using the Importing and Subsetting pane, you can

- Specify the name of the workspace variable — By default, the HDF Import Tool uses the name of the HDF4 data set as the name of the MATLAB workspace variable. In the following figure, the variable name is `Example_SDS`. To specify another name, enter text in the **Workspace Variable** text box.

- Specify whether to import metadata associated with the data set — To import any metadata that might be associated with the data set, select the **Import Metadata** check box. To store the metadata, the HDF Import Tool creates a second variable in the workspace with the same name with "`_info`" appended to it. For example, if you select this check box, the name of the metadata variable for the data set in the figure would be `Example_SDS_info`.

- Save the data set import command syntax — The **Dataset import command** text window displays the MATLAB command used to import the data set. This text is not editable, but you can copy and paste it into the MATLAB Command Window or a text editor for reuse.

The following figure shows how to specify these options in the HDF Import Tool.

Import metadata with data set

Specify name of variable to store data set

MATLAB command used to import data

Click here to import data set
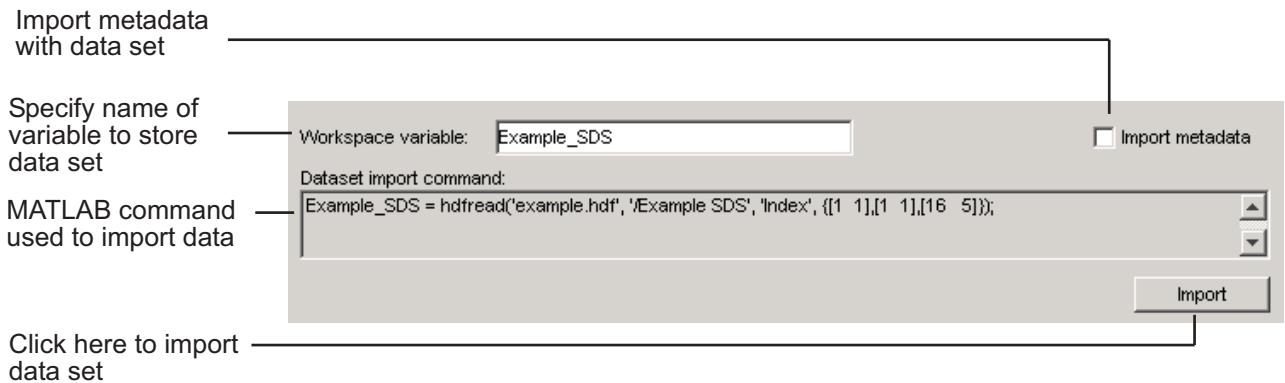
Step 5: Closing HDF Files and the HDF Import Tool

To close a file, select the file in the contents pane and click **Close File** on the HDF Import Tool **File** menu.

To close all the files open in the HDF Import Tool, click **Close All Files** on the HDF Import Tool **File** menu.

To close the tool, click **Close HDFTool** in the HDF Import Tool **File** menu or click the Close button in the upper right corner of the tool.

If you used the hdftool syntax that returns a handle to the tool,

```
h = hdftool('example.hdf')
```

you can use the close(h) command to close the tool from the MATLAB command line.

Using the HDF Import Tool Subsetting Options

Note: The HDF Import Tool will be removed in a future release.

When you select a data set, the importing and subsetting pane displays the subsetting options available for that type of data set. The following sections provide information about these subsetting options for all supported data set types. For general information about the HDF Import tool, see "Import HDF4 Files Interactively" on page 6-61.

Note To use these data subsetting options effectively, you must understand the HDF and HDF-EOS data formats. Therefore, use this documentation in conjunction with the HDF documentation (www.hdfgroup.org) and the HDF-EOS documentation (www.hdfeos.org).

HDF Scientific Data Sets (SD)

The HDF scientific data set (SD) is a group of data structures used to store and describe multidimensional arrays of scientific data. Using the HDF Import Tool subsetting parameters, you can import a subset of an HDF scientific data set by specifying the location, range, and number of values to be read along each dimension.
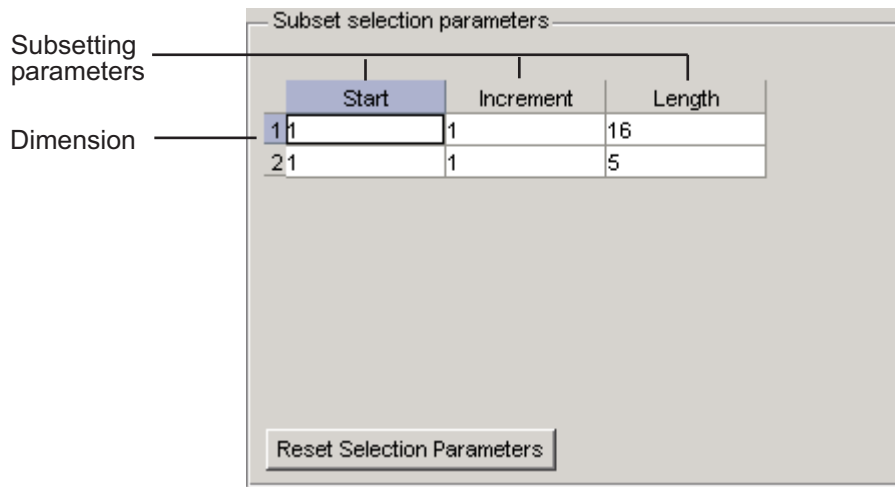
The subsetting parameters are:

- **Start** — Specifies the position on the dimension to begin reading. The default value is 1, which starts reading at the first element of each dimension. The values specified must not exceed the size of the relevant dimension of the data set.

- **Increment** — Specifies the interval between the values to read. The default value is 1, which reads every element of the data set.

- **Length** — Specifies how much data to read along each dimension. The default value is the length of the dimension, which causes all the data to be read.

HDF Vdata

HDF Vdata data sets provide a framework for storing customized tables. A Vdata table consists of a collection of records whose values are stored in fixed-length fields. All records have the same structure and all values in each field have the same data type. Each field is identified by a name. The following figure illustrates a Vdata table.

Fieldnames —	idx	Temp	Dewpt
	1	0	5
Records —	2	12	5
	3	3	7

Fields

You can import a subset of an HDF Vdata data set in the following ways:

- Specifying the name of the field that you want to import
- Specifying the range of records that you want to import

The following figure shows how you specify these subsetting parameters for Vdata.

Specify field to subset → (Subset selection parameters — Data fields: Idx, Temp, Dewpt)

Specify where to begin reading → First Record: 1

Specify how many records to read → Number of records: 10
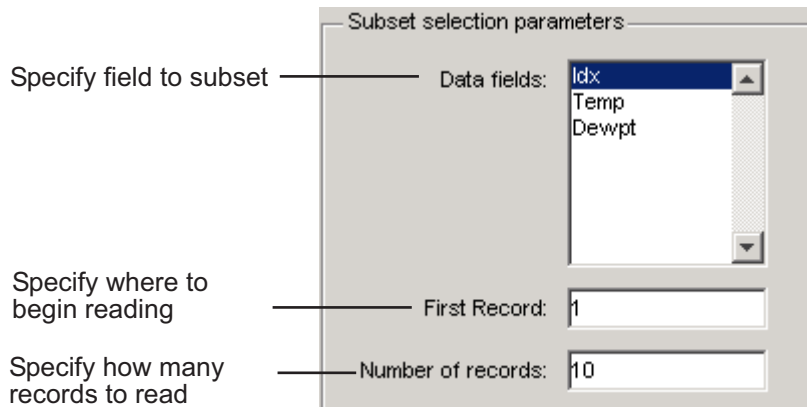
HDF-EOS Grid Data

In HDF-EOS Grid data, a rectilinear grid overlays a map. The map uses a known map projection. The HDF Import Tool supports the following mutually exclusive subsetting options for Grid data:

- "Direct Index" on page 6-69
- "Geographic Box" on page 6-69
- "Interpolation" on page 6-70
- "Pixels" on page 6-71
- "Tile" on page 6-71
- "Time" on page 6-72
- "User-Defined" on page 6-72

To access these options, click the Subsetting method menu in the importing and subsetting pane.

Click here to
see options

Direct Index

You can import a subset of an HDF-EOS Grid data set by specifying the location, range, and number of values to be read along each dimension.

Each row represents a dimension in the data set and each column represents these subsetting parameters:

- **Start** — Specifies the position on the dimension to begin reading. The default value is 1, which starts reading at the first element of each dimension. The values specified must not exceed the size of the relevant dimension of the data set.
- **Increment** — Specifies the interval between the values to read. The default value is 1, which reads every element of the data set.
- **Length** — Specifies how much data to read along each dimension. The default value is the length of the dimension, which causes all the data to be read.

Geographic Box

You can import a subset of an HDF-EOS Grid data set by specifying the rectangular area of the grid that you are interested in. To define this rectangular area, you must

specify two points, using longitude and latitude in decimal degrees. These points are two corners of the rectangular area. Typically, **Corner 1** is the upper-left corner of the box, and **Corner 2** is the lower-right corner of the box.

Optionally, you can further define the subset of data you are interested in by using Time parameters (see "Time" on page 6-72) or by specifying other User-Defined subsetting parameters (see "User-Defined" on page 6-72).

Interpolation

Interpolation is the process of estimating a pixel value at a location in between other pixels. In interpolation, the value of a particular pixel is determined by computing the weighted average of some set of pixels in the vicinity of the pixel.

You define the region used for bilinear interpolation by specifying two points that are corners of the interpolation area:

• **Corner 1** – Specify longitude and latitude values in decimal degrees. Typically, **Corner 1** is the upper-left corner of the box.

• **Corner 2** — Specify longitude and latitude values in decimal degrees. Typically, **Corner 2** is the lower-right corner of the box

Pixels

You can import a subset of the pixels in a Grid data set by defining a rectangular area over the grid. You define the region used for bilinear interpolation by specifying two points that are corners of the interpolation area:

- **Corner 1** – Specify longitude and latitude values in decimal degrees. Typically, **Corner 1** is the upper-left corner of the box.

- **Corner 2** — Specify longitude and latitude values in decimal degrees. Typically, **Corner 2** is the lower-right corner of the box

Tile

In HDF-EOS Grid data, a rectilinear grid overlays a map. Each rectangle defined by the horizontal and vertical lines of the grid is referred to as a *tile*. If the HDF-EOS Grid data is stored as tiles, you can import a subset of the data by specifying the coordinates of the tile you are interested in. Tile coordinates are 1-based, with the upper-left corner of a two-dimensional data set identified as 1,1. In a three-dimensional data set, this tile would be referenced as 1,1,1.

Time

You can import a subset of the Grid data set by specifying a time period. You must specify both the start time and the stop time (the endpoint of the time span). The units (hours, minutes, seconds) used to specify the time are defined by the data set.

Along with these time parameters, you can optionally further define the subset of data to import by supplying user-defined parameters.

User-Defined

You can import a subset of the Grid data set by specifying user-defined subsetting parameters.

When specifying user-defined parameters, you must first specify whether you are subsetting along a dimension or by field. Select the dimension or field by name using the **Dimension or Field Name** menu. Dimension names are prefixed with the characters `DIM:`.

Once you specify the dimension or field, you use **Min** and **Max** to specify the range of values that you want to import. For dimensions, **Min** and **Max** represent a range of *elements*. For fields, **Min** and **Max** represent a range of *values*.

HDF-EOS Point Data

HDF-EOS Point data sets are tables. You can import a subset of an HDF-EOS Point data set by specifying field names and level. Optionally, you can refine the subsetting by specifying the range of records you want to import, by defining a rectangular area, or by specifying a time period. For information about specifying a rectangular area, see "Geographic Box" on page 6-69. For information about subsetting by time, see "Time" on page 6-72.

HDF-EOS Swath Data

HDF-EOS Swath data is data that is produced by a satellite as it traces a path over the earth. This path is called its ground track. The sensor aboard the satellite takes a series of scans perpendicular to the ground track. Swath data can also include a vertical measure as a third dimension. For example, this vertical dimension can represent the height above the Earth of the sensor.

The HDF Import Tool supports the following mutually exclusive subsetting options for Swath data:

• "Direct Index" on page 6-74

- "Geographic Box" on page 6-75
- "Time" on page 6-76
- "User-Defined" on page 6-76

To access these options, click the `Subsetting method` menu in the **Importing and Subsetting** pane.

Click here to
select a subsetting
option

Direct Index

You can import a subset of an HDF-EOS Swath data set by specifying the location, range, and number of values to be read along each dimension.

Each row represents a dimension in the data set and each column represents these subsetting parameters:

- **Start** — Specifies the position on the dimension to begin reading. The default value is 1, which starts reading at the first element of each dimension. The values specified must not exceed the size of the relevant dimension of the data set.

- **Increment** — Specifies the interval between the values to read. The default value is 1, which reads every element of the data set.

- **Length** — Specifies how much data to read along each dimension. The default value is the length of the dimension, which causes all the data to be read.

Geographic Box

You can import a subset of an HDF-EOS Swath data set by specifying the rectangular area of the grid that you are interested in and by specifying the selection Mode.

You define the rectangular area by specifying two points that specify two corners of the box:

- **Corner 1** — Specify longitude and latitude values in decimal degrees. Typically, **Corner 1** is the upper-left corner of the box.

- **Corner 2** — Specify longitude and latitude values in decimal degrees. Typically, **Corner 2** is the lower-right corner of the box.

You specify the selection mode by choosing the type of **Cross Track Inclusion** and the **Geolocation mode**. The **Cross Track Inclusion** value determines how much of the area of the geographic box that you define must fall within the boundaries of the swath.

Select from these values:

- **AnyPoint** — Any part of the box overlaps with the swath.
- **Midpoint** — At least half of the box overlaps with the swath.
- **Endpoint** — All of the area defined by the box overlaps with the swath.

The **Geolocation Mode** value specifies whether geolocation fields and data must be in the same swath.
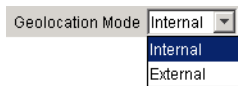
Select from these values:

- **Internal** — Geolocation fields and data fields must be in the same swath.
- **External** — Geolocation fields and data fields can be in different swaths.

Time

You can optionally also subset swath data by specifying a time period. The units used (hours, minutes, seconds) to specify the time are defined by the data set
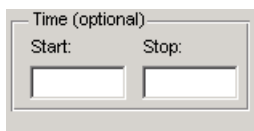
User-Defined

You can optionally also subset a swath data set by specifying user-defined parameters.
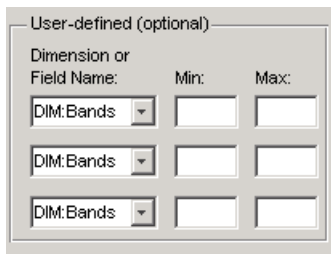
When specifying user-defined parameters, you must first specify whether you are subsetting along a dimension or by field. Select the dimension or field by name using the

Dimension or Field Name menu. Dimension names are prefixed with the characters `DIM:`.

Once you specify the dimension or field, you use **Min** and **Max** to specify the range of values that you want to import. For dimensions, **Min** and **Max** represent a range of *elements*. For fields, **Min** and **Max** represent a range of *values*.

HDF Raster Image Data

For 8-bit HDF raster image data, you can specify the colormap.

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

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 string specifying the MATLAB data type of the data set.

6-79

```
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 6-81
- "Close All Open HDF4 Identifiers" on page 6-82

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 6-55

7

Audio and Video

Read and Write Audio Files

This example shows how to write data to an audio file, get information about the file, and then read data from the audio file.

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')
```
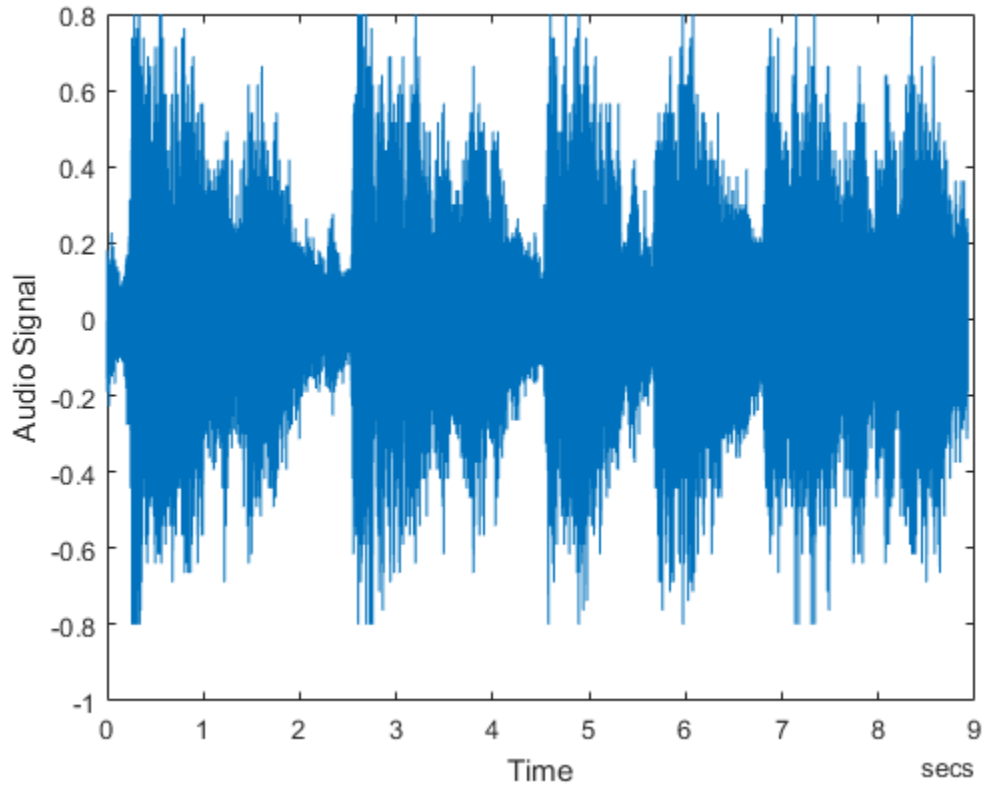
See Also
`audioinfo` | `audioread` | `audiowrite`

Related Examples
- "Import Images, Audio, and Video Interactively" on page 1-9

Record and Play Audio

In this section...
"Record Audio" on page 7-5
"Play Audio" on page 7-7
"Record or Play Audio within a Function" on page 7-8

Record Audio

To record data from an audio input device (such as a microphone connected to your system) for processing in MATLAB:

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
   NumberOfChannels: 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 7-8.

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 7-2

Get Information about Video Files

`VideoReader` creates an object that contains properties of the video file, including the duration, frame rate, format, height, and width. To view these properties, or store them in a structure, use the `get` method. For example, get the properties of the file `xylophone.mp4`:

```
xyloObj = VideoReader('xylophone.mp4');
info = get(xyloObj)
```

The `get` function returns:

```
info =

        Duration: 4.7000
            Name: 'xylophone.mp4'
            Path: 'matlabroot\toolbox\matlab\audiovideo'
             Tag: ''
        UserData: []
    BitsPerPixel: 24
       FrameRate: 30
          Height: 240
     VideoFormat: 'RGB24'
           Width: 320
     CurrentTime: 0
```

To access a specific property of the object, such as `Duration`, use dot notation as follows:

```
duration = xyloObj.Duration;
```

Read Video Files

Read All Frames in Video File

This example shows how to read and store data from all frames in a video file, display one frame, and then play all frames at the video's frame rate.

Construct a VideoReader object associated with the sample file, xylophone.mp4.

```
vidObj = VideoReader('xylophone.mp4');
```

Determine the height and width of the frames.

```
vidHeight = vidObj.Height;
vidWidth = vidObj.Width;
```

Create a MATLAB movie structure array, s.

```
s = struct('cdata',zeros(vidHeight,vidWidth,3,'uint8'),...
    'colormap',[]);
```

Read one frame at a time using readFrame until the end of the file is reached. Append data from each video frame to the structure array.

```
k = 1;
while hasFrame(vidObj)
    s(k).cdata = readFrame(vidObj);
    k = k+1;
end
```

Get information about the movie structure array, s.

```
whos s
```

```
  Name      Size                  Bytes  Class     Attributes
```

```
     s           1x141                32503552   struct
```

s is a 1-by-141 structure array, containing data from the 141 frames in the video file.

Display the fifth frame stored in s.

```
image(s(5).cdata)
```

Resize the current figure and axes based on the video's width and height. Then, play the movie once at the video's frame rate using the movie function.

```
set(gcf,'position',[150 150 vidObj.Width vidObj.Height]);
set(gca,'units','pixels');
set(gca,'position',[O O vidObj.Width vidObj.Height]);
movie(s,1,vidObj.FrameRate);
```

Close the figure.

```
close
```

Read All Frames Beginning at Specified Time

Read part of a video file starting 0.5 second from the beginning of the 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;
```

Create an axes to display the video. Then, read video frames until the end of the file is reached.

```
currAxes = axes;
while hasFrame(vidObj)
    vidFrame = readFrame(vidObj);
    image(vidFrame, 'Parent', currAxes);
    currAxes.Visible = 'off';
    pause(1/vidObj.FrameRate);
```

```
end
```


Read Video Frames Within Specified Time Interval

Read part of a video file from 0.6 to 0.9 second.

Construct a `VideoReader` object associated with the sample file, `'xylophone.mp4'`.

```
vidObj = VideoReader('xylophone.mp4');
```

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 the `CurrentTime` reaches 0.9 second. Append data from each video frame to the structure array, `s`.

```
k = 1;
while vidObj.CurrentTime <= 0.9
    s(k).cdata = readFrame(vidObj);
    k = k+1;
end
```

View the number of frames in `s`.

```
whos s
```

```
  Name      Size                  Bytes  Class     Attributes

  s         1x141              32503552  struct
```

`s` is a 1-by-141 structure array because `readFrame` read 141 frames.

View the `CurrentTime` property of the `VideoReader` object.

```
vidObj.CurrentTime
```

```
ans =

    0.9333
```

The `CurrentTime` property is now greater than 0.9.

Troubleshooting: Cannot Read Last Frame of Video File

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.

Avoid seeking to the last frame in a video file by setting the `CurrentTime` property to a value close to the `Duration` value. 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.

Do not use the `Duration` property to limit the reading of data from a video file. It is best to read data until the file reports that there are no more frames available to read. That is, use the `hasFrame` method to check whether there is a frame available to read.

See Also
`mmfileinfo` | `movie` | `VideoReader`

More About
*   "Supported Video File Formats" on page 7-16

Supported Video File Formats

In this section...
"What Are Video Files?" on page 7-16
"Formats That `VideoReader` Supports" on page 7-16
"View Codec Associated with Video File" on page 7-17
"Troubleshooting: Errors Reading Video File" on page 7-18

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`)

Platforms	File Formats
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
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
Linux	Any format supported by your installed plug-ins for GStreamer 0.10, as listed on http://gstreamer.freedesktop.org/documentation/plugins.html, including Ogg Theora (`.ogg`). **Note:** Support is provided for GStreamer 0.10 only, not later versions.

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`.

```
info = mmfileinfo('shuttle.avi');
```

View the `Format` field of the structure.

```
info.Video.Format

ans =

MJPG
```

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.

Convert Between Image Sequences and Video

This example shows how to 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.

```
f = figure;
f.Position = [150 150 shuttleAvi.Width shuttleAvi.Height];

ax = gca;
ax.Units = 'pixels';
ax.Position = [0 0 shuttleAvi.Width shuttleAvi.Height];

image(mov(1).cdata,'Parent',ax)
axis off
```


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.

Export to Video

To create an Audio/Video Interleaved (AVI) file from MATLAB graphics animations or from still images, follow these steps:

1 Create a `VideoWriter` object by calling the `VideoWriter` function. For example:

```
myVideo = VideoWriter('myfile.avi');
```

By default, `VideoWriter` prepares to create an AVI file using Motion JPEG compression. To create an uncompressed file, specify the `Uncompressed AVI` profile, as follows:

```
uncompressedVideo = VideoWriter('myfile.avi', 'Uncompressed AVI');
```

2 Optionally, adjust the frame rate (number of frames to display per second) or the quality setting (a percentage from 0 through 100). For example:

```
myVideo.FrameRate = 15;  % Default 30
myVideo.Quality = 50;    % Default 75
```

Note: Quality settings only apply to compressed files. Higher quality settings result in higher video quality, but also increase the file size. Lower quality settings decrease the file size and video quality.

---

3 Open the file:

```
open(myVideo);
```

Note: After you call `open`, you cannot change the frame rate or quality settings.

---

4 Write frames, still images, or an existing MATLAB movie to the file by calling `writeVideo`. For example, suppose that you have created a MATLAB movie called `myMovie`. Write your movie to a file:

```
writeVideo(myVideo, myMovie);
```

Alternatively, `writeVideo` accepts single frames or arrays of still images as the second input argument. For more information, see the writeVideo reference page.

5 Close the file:

```
close(myVideo);
```

See Also

`VideoWriter`

Characteristics of 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.

8

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 8-2
- "Example — Finding Text in an XML File" on page 8-3

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
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="http://www.mathworks.com/namespace/info/v1/info.xsd">

<!-- This is a sample info.xml file. -->

<list>

<listitem>
<label>Import Wizard</label>
<callback>uiimport</callback>
<icon>ApplicationIcon.GENERIC_GUI</icon>
</listitem>

<listitem>
<label>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, `xmlns:xsi` is the name of an attribute and `http://www.w3.org/2001/XMLSchema-instance` 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 8-2

For more information, see:

- "Creating an XML File" on page 8-5
- "Updating an Existing XML File" on page 8-7

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 8-3
- "Example — Creating an XML File with xmlwrite" on page 8-6
- `xmlread` and `xmlwrite`

9

Memory-Mapping Data Files

Overview of Memory-Mapping

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.
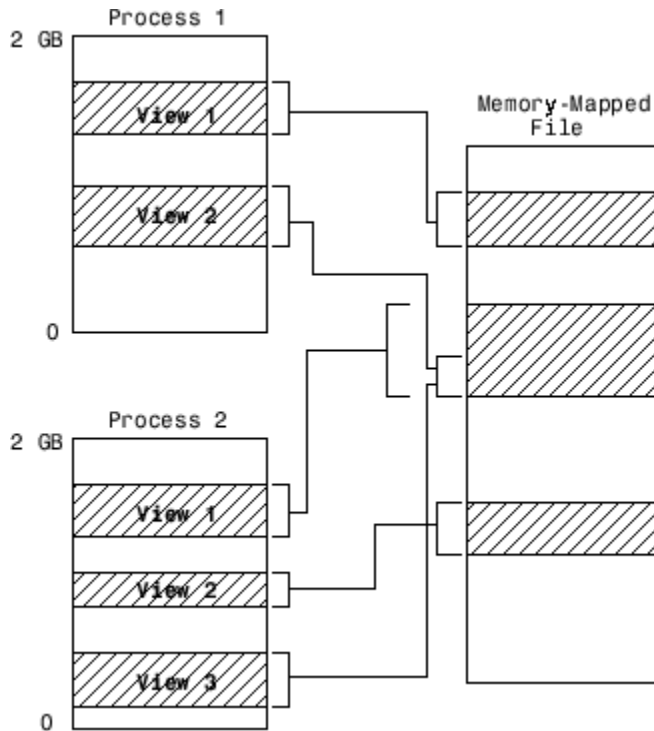
Also, within a single application, you can map the same segment of a file more than once.

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

In this section...
"Create a Simple Memory Map" on page 9-6
"Specify Format of Your Mapped Data" on page 9-7
"Map Multiple Data Types and Arrays" on page 9-8
"Select File to Map" on page 9-10

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 string 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 contains a string identifying 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 a string specifying 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.
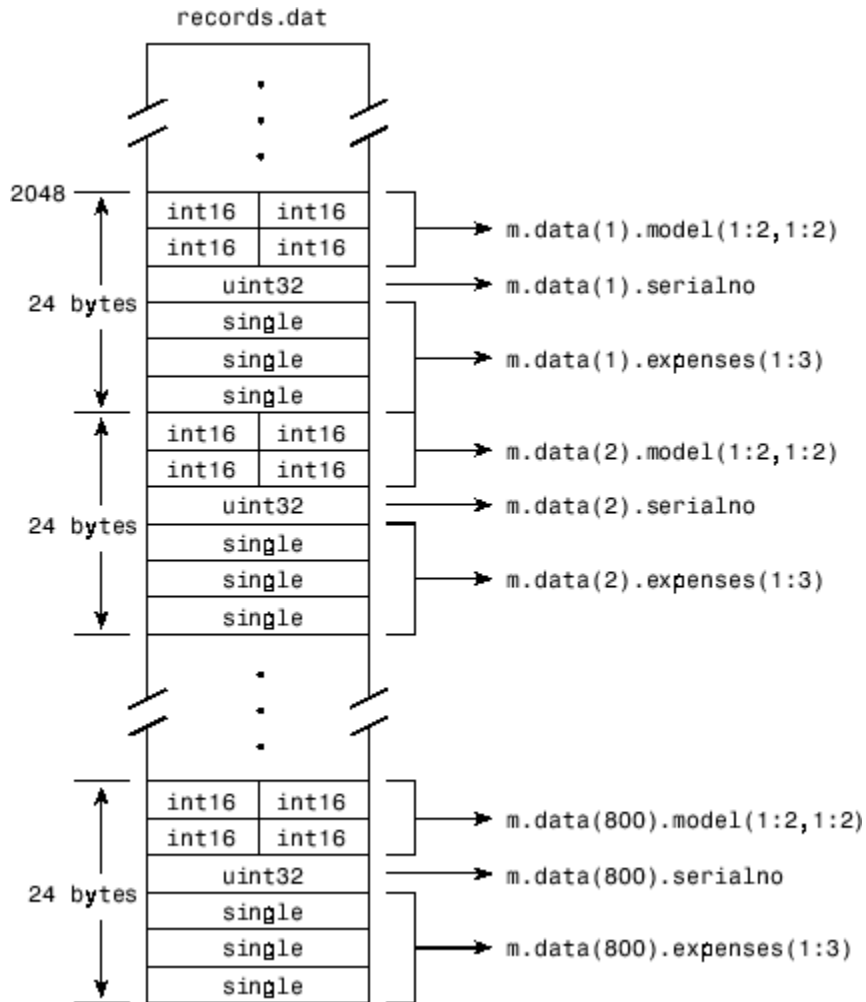
The next figure shows the ordering of the array elements more closely. In particular, it illustrates that MATLAB arrays are stored on the disk in column-major order. The sequence of array elements in the mapped file is row 1, column 1; row 2, column 1; row 1, column 2; and row 2, column 2.
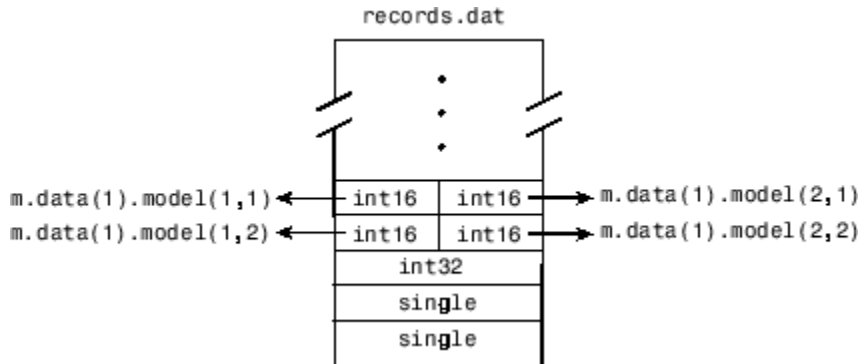
If the data in your file is not stored in this order, you might need to transpose or rearrange the order of array elements when reading or writing via a memory map.

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` string 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 9-11
- "Write to Mapped File" on page 9-17

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 =

    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 =

166x1 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 =

    x: [5x8 uint16]
    y: [4x5 double]
```

Read the `x` field of that structure from the file.

```
d(3).x


ans =

  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 =

    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))))
```
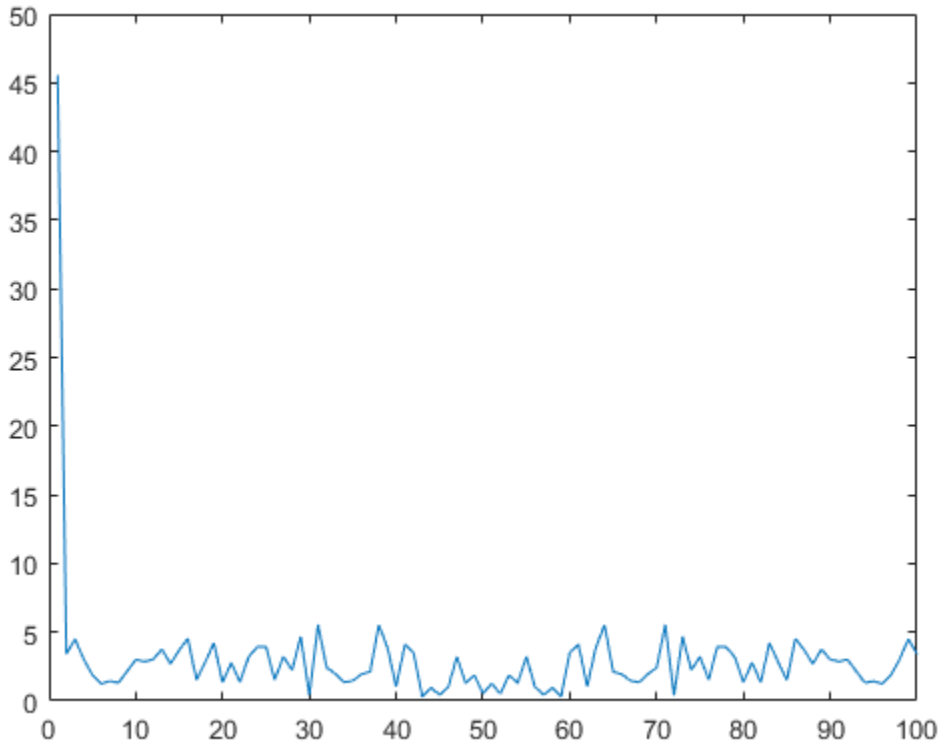
This is the first time that data is referenced and is when the actual mapping of the file to the MATLAB address space takes place.

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 =

  7.5449e+37
```

See Also
memmapfile

More About
- "Map File to Memory" on page 9-6
- "Write to Mapped File" on page 9-17

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" on page 9-17
"Write to Memory Mapped as Scalar Structure" on page 9-18
"Write to Memory Mapped as Nonscalar Structure" on page 9-19
"Syntaxes for Writing to Mapped File" on page 9-19
"Work with Copies of Your Mapped Data" on page 9-20

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)'
```

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
```

`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 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
```

`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
```

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	m.Data = X;

Format of the `Data` Property	Syntax for Writing to Mapped File
Example: `15x1 uint16` `array`	

Scalar (1-by-1) structure array
 	
Example:
 	
`1x1 struct array with field`
`                x`
`                y`
	`m.Data.`*`fieldname`*` = X;`
	
	`fieldname` is the name of a field.

Nonscalar (n-by-1) structure array
 	
Example:
 	
`20x1 struct array with fiel`
`                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
```

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
```

Verify that the data in the mapped file is not changed.

```
m.Data(2).x
```

See Also
memmapfile

More About
- "Map File to Memory" on page 9-6
- "Read from Mapped File" on page 9-11

Delete Memory Map

In this section...
"Ways to Delete a Memory Map" on page 9-22
"The Effect of Shared Data Copies On Performance" on page 9-22

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 a string and then, using memory-mapping, passes the string 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
```

```matlab
% Set first byte to zero, indicating a message is not
% yet ready.
m.Data(1) = 0;

str = input('Enter send string (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.

```matlab
function answer
```

```matlab
% 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 send string (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 send string (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 send string (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.
```

Internet File Access

MATLAB software provides functions for exchanging files over the Internet. You can exchange files using common protocols, such as File Transfer Protocol (FTP), Simple Mail Transport Protocol (SMTP), and HyperText Transfer Protocol (HTTP). In addition, you can create zip archives to minimize the transmitted file size, and also save and work with Web pages.

Download Data from Web Service

This example shows how to download data from a web service with the `webread` function. The World Bank provides a variety of 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 = datenum(1899,1,1);
xmax = datenum(2014,1,1);
xlim([xmin xmax])
```
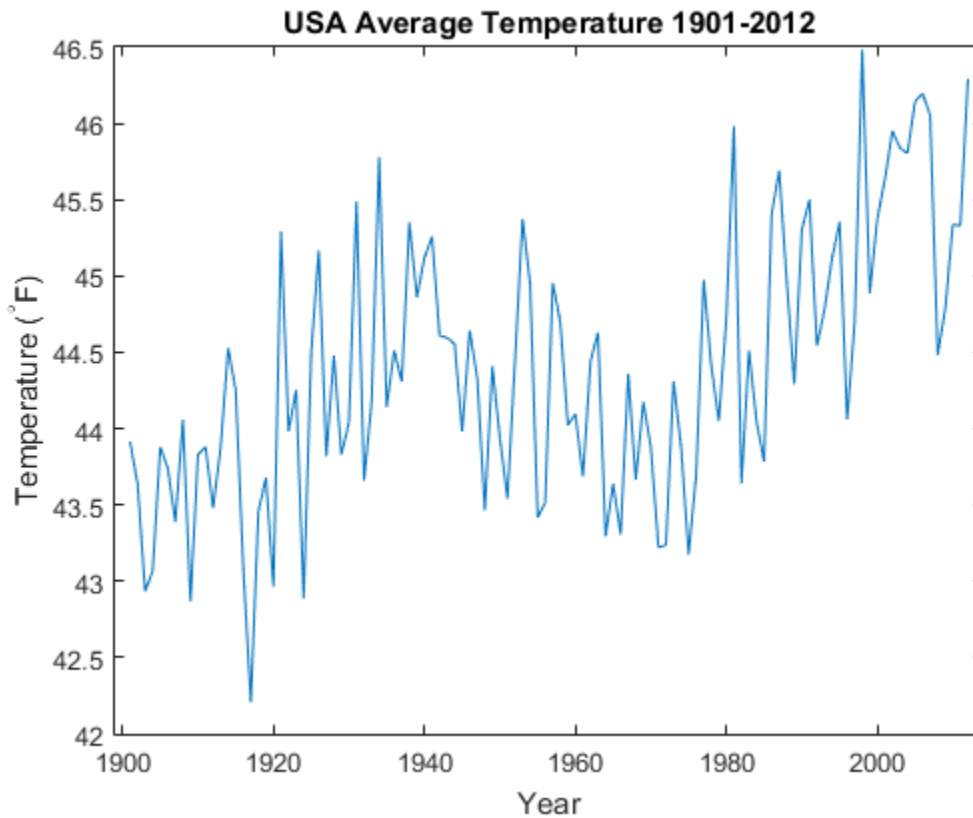

Overplot a least-squares fit of a line to the temperatures.

```
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
```
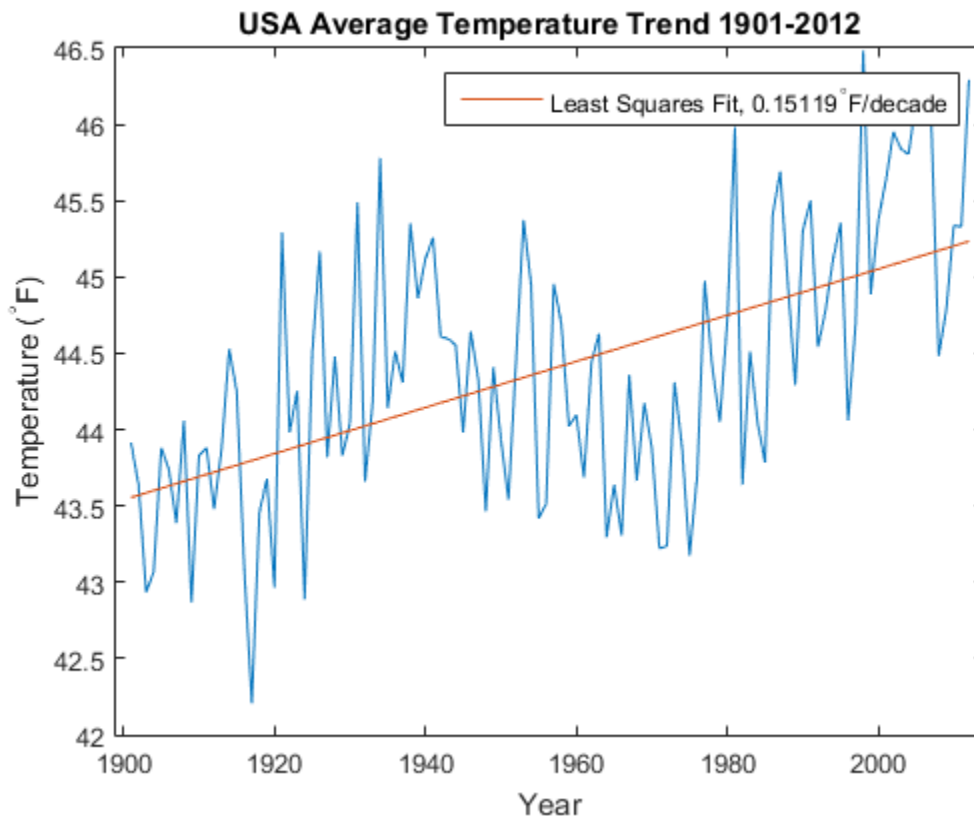

API and data courtesy of the World Bank: Climate Data API.

- See World Bank: Climate Data API for more information about the API.

- See World Bank: Terms of Use for terms of use for data and API.

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 a variety of 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])
```
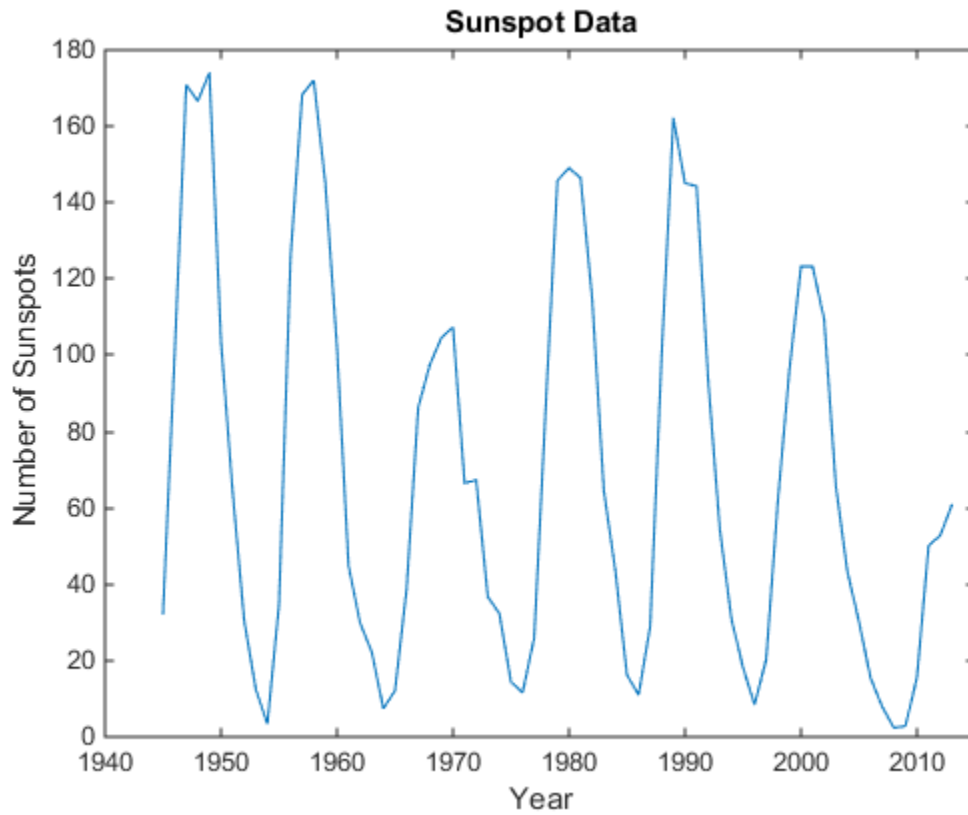
Aggregated data and web service courtesy of the NGDC. Sunspot data courtesy of the AAVSO, originally published in AAVSO Sunspot Counts: 1943-2013, AAVSO Solar Section (R. Howe, Chair).

- 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 a Web page's 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, `http://www.mathworks.com/matlabcentral/fileexchange/`. It assigns the results to a character array, `fullList`:

```
filex = 'http://www.mathworks.com/matlabcentral/fileexchange/';
fullList = webread(filex);
```

Retrieve a list of only those files uploaded to the File Exchange within the past 7 days that contain the word Simulink®. Set `duration` and `term` as parameters that `webread` passes to the Web page.

```
filex = 'http://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 = 'http://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/c
```

```
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 strings.

```
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

From MATLAB, you can connect to an FTP server to perform remote file operations. The following procedure uses a public FTP server at the National Geophysical Data Center (NGDC). To perform any file operation on an FTP server, follow these steps:

1 Connect to the server using the `ftp` function.
2 Perform file operations using appropriate MATLAB FTP functions. For all operations, specify the server object.
3 When you finish working on the server, close the connection object using the `close` function.

Example — Retrieve a File from an FTP Server

List the contents of the anonymous FTP service at the NGDC.

```
ngdc = ftp('ftp.ngdc.noaa.gov');
dir(ngdc)
```

```
DMSP                        Solid_Earth                 international            wdc
INDEX.txt                   ftp.html                    ionosonde
README.txt                  geomag                      mgg
STP                         hazards                     pub
Snow_Ice                    index.html                  tmp
```

Retrieve a file named `INDEX.txt`. To view the file, use the `type` function.

```
mget(ngdc,'INDEX.txt');
type INDEX.txt
```

```
                    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, NGDC
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 NGDC. Direct E-mail inquiries to ngdc.info@noaa.gov

Also see our web site: http://www.ngdc.noaa.gov/ngdc.html

NGDC is part of the:
U.S. Department of Commerce, National Oceanic and Atmospheric Administration (NOAA),
National Environmental Satellite, Data and Information Service (NESDIS)
```

View the contents of the **pub** folder:

```
cd(ngdc,'pub')
dir(ngdc)
```

Close the FTP connection.

```
close(ngdc)
```

FTP service courtesy of the NGDC. See NGDC Privacy Policy, Disclaimer, and Copyright for NGDC terms of service.

See Also
FTP

Display Hyperlinks in the Command Window

In this section...
"Create Hyperlinks to Web Pages" on page 10-17
"Transfer Files Using FTP" on page 10-17

Create Hyperlinks to Web Pages

When you create a hyperlink to a Web page, append a full hypertext string on a single line as input to the `disp` or `fprintf` command. For example, the following command:

```
disp('<a href = "http://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.

11

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?" on page 11-2
"MapReduce Algorithm Phases" on page 11-3
"Example MapReduce Calculation" on page 11-4

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 chunks that individually fit into memory. Each chunk goes through a Map phase, which formats the data to be processed. Then the intermediate data chunks 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 11-21.

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 chunk 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`.
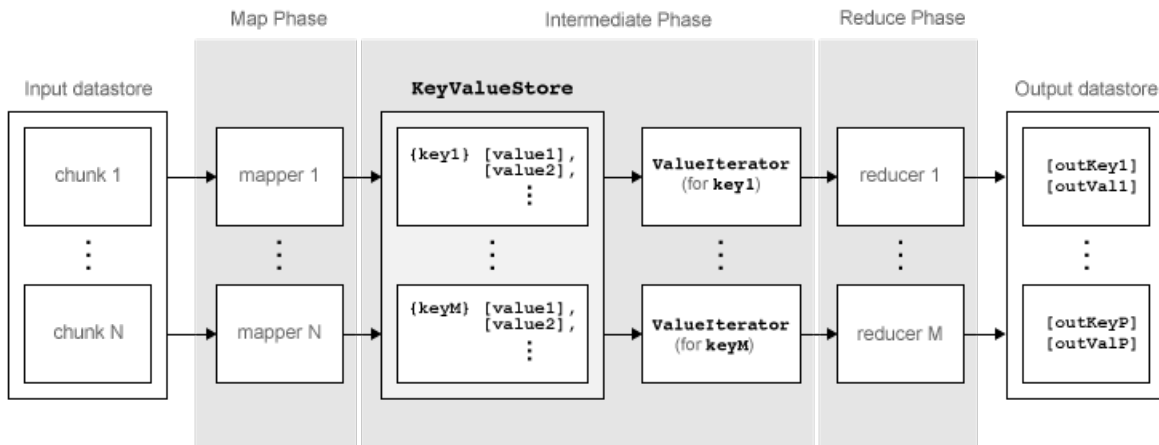
The algorithm has the following steps:

1 `mapreduce` reads a chunk of data from the input datastore using `[data,info] = read(ds)`, and then calls the map function to work on that chunk.

2 The map function receives the chunk 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 chunks in the input datastore.

3 After the map function works on all of the chunks 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 chunks.

`mapreduce` works with `TabularTextDatastore`, `ImageDatastore`, or `KeyValueDatastore`. `TabularTextDatastore` is a datastore for text files, `ImageDatastore` is a datastore for images, and `KeyValueDatastore` is a datastore for key-value pairs. The `datastore` function automatically determines which type is appropriate for the data set. For example, if you create a datastore for the `airlinesmall.csv` data set, then `datastore` determines that it should be a `TabularTextDatastore`:

```
ds = datastore('airlinesmall.csv','TreatAsMissing','NA')
```

```
ds =

  TabularTextDatastore with properties:

                        Files: {
                               ' ...\matlab\toolbox\matlab\demos\airlinesmall.csv'
                               }
            ReadVariableNames: true
                VariableNames: {'Year', 'Month', 'DayofMonth' ... and 26 more}

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

  Advanced Text Format Properties:
                TextscanFormats: {'%f', '%f', '%f' ... and 26 more}
            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
```

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 chunk 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 =

    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 the `datastore` reference page.

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 chunk of distance data. The second line creates a two-element vector with the total distance and count for the chunk, and the third line adds that vector of values to intermKVStore with the key, 'sumAndLength'. After this map function runs on all of the chunks of data in ds, the intermKVStore object contains the total distance and count for each chunk 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 11-10 and "Write a Reduce Function" on page 11-15.

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.m`, and `MeanDistReduceFun.m`.

```
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 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 =

    Key         Value
    _____     _____
```

```
'Mean'    [702.1630]
```

See Also
```
datastore | mapreduce
```

Related Examples
· "Build Effective Algorithms with MapReduce" on page 11-22

Write a Map Function

Role of Map Function in MapReduce

mapreduce requires both an input map function that receives chunks 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 chunk 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.
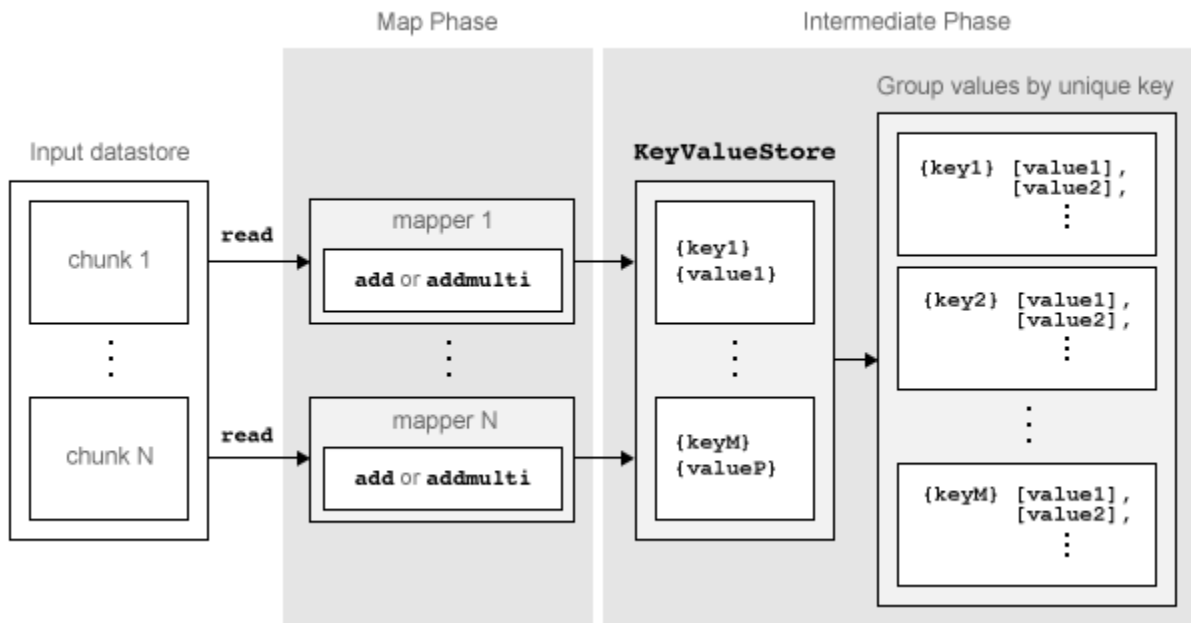
The Map phase of the `mapreduce` algorithm has the following steps:

1   `mapreduce` reads a single chunk of data using the `read` function on the input datastore, then calls the map function to work on the chunk.

2   The map function then works on the individual chunk 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 chunks of data in the input datastore, so that the total number of calls to the map function is equal to the number of chunks of data. The `ReadSize` property of the datastore determines the number of data chunks.

The Map phase of the `mapreduce` algorithm is complete when the map function processes each of the chunks 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 chunk 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 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

These examples contain some map functions used by the `mapreduce` examples in the `toolbox/matlab/demos` folder.

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.m` mapper file is one of the mappers used in the example file `TSQRMapReduceExample.m`.

```
type identityMapper.m
```

```
function identityMapper(data, info, intermKVStore)
% Mapper function for the MapReduce TSQR example.
%
% This mapper function simply copies the data and add them to the
% intermKVStore as intermediate values.

% Copyright 2014 The MathWorks, Inc.

x = data.Value{:,:};
add(intermKVStore,'Identity', x);
```

Simple Map Function

One of the simplest examples of a nonidentity mapper is `maxArrivalDelayMapper.m`, which is the mapper for the example file `MaxMapReduceExample.m`. For each chunk of input data, this mapper calculates the maximum arrival delay and adds a key-value pair to the intermediate `KeyValueStore`.

```
type maxArrivalDelayMapper.m
```

```
function maxArrivalDelayMapper (data, info, intermKVStore)
% Mapper function for the MaxMapreduceExample.

% Copyright 1984-2014 The MathWorks, Inc.

% Data is an n-by-1 table of the ArrDelay. As the data source is tabular,
% the return of read is a table object.
partMax = max(data.ArrDelay);
add(intermKVStore, 'PartialMaxArrivalDelay',partMax);
```

Advanced Map Function

A more advanced example of a mapper is `statsByGroupMapper.m`, which is the mapper for the example file `StatisticsByGroupMapReduceExample.m`. 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.

type statsByGroupMapper.m


```
function statsByGroupMapper(data, ~, intermKVStore, groupVarName)
% Mapper function for the StatisticsByGroupMapReduceExample.

% Copyright 2014 The MathWorks, Inc.

% 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]};
```

More Map Functions

For more information about common programming patterns in map or reduce functions, see "Build Effective Algorithms with MapReduce" on page 11-22.

See Also

add | addmulti | datastore | mapreduce

More About

- Using KeyValueStore Objects
- "Write a Reduce Function" on page 11-15
- "Getting Started with MapReduce" on page 11-2

Write a Reduce Function

Role of the Reduce Function in MapReduce

mapreduce requires both an input map function that receives chunks 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 chunk 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.
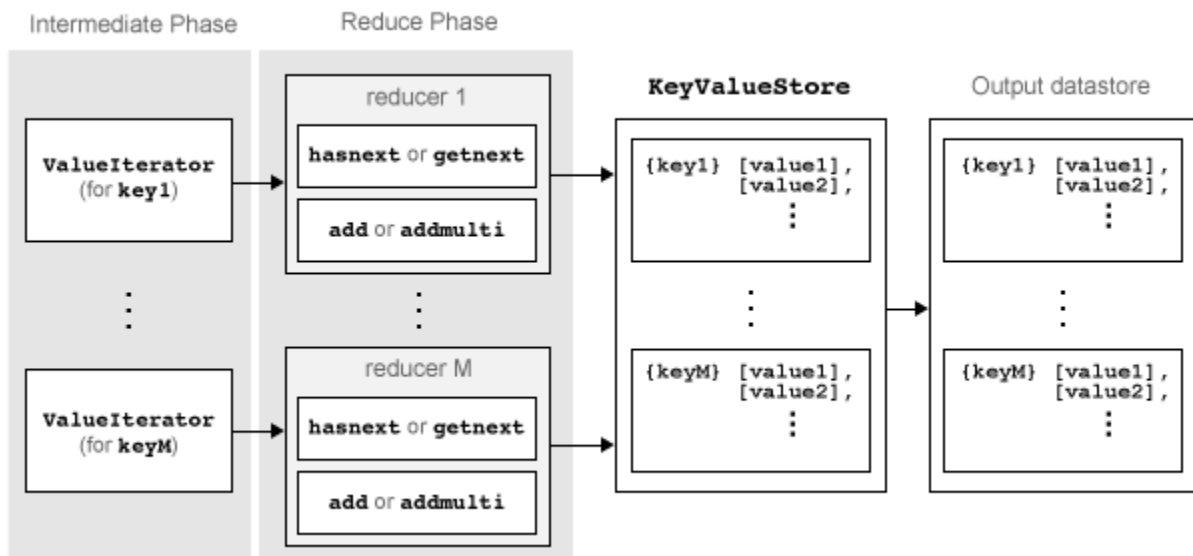
The Reduce phase of the mapreduce algorithm has the following steps:

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 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 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

These examples contain some reduce functions used by the `mapreduce` examples in the `toolbox/matlab/demos` folder.

Simple Reduce Function

One of the simplest examples of a reducer is `maxArrivalDelayReducer.m`, which is the reducer for the example file `MaxMapReduceExample.m`. 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.

type maxArrivalDelayReducer.m

```
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);
```

Advanced Reduce Function

A more advanced example of a reducer is `statsByGroupReducer.m`, which is the reducer for the example file `StatisticsByGroupMapReduceExample.m`. 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.

type statsByGroupReducer.m

```
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 11-22.

See Also

add | addmulti | datastore | getnext | hasnext | mapreduce

More About

· Using KeyValueStore Objects
· Using ValueIterator Objects

Speed Up and Deploy MapReduce Using Other Products

In this section...
"Execution Environment" on page 11-21
"Running in Parallel" on page 11-21
"Application Deployment" on page 11-21

Execution Environment

To use `mapreduce` with Parallel Computing Toolbox™, MATLAB Distributed Computing 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".

MATLAB Distributed Computing Server enables you to run the same applications on a remote computer cluster. For more information, including how to configure MATLAB Distributed Computing Server to support Hadoop clusters, see "Big Data".

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 "Hadoop Integration".

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 11-32	MaxMapReduceExamp	Find maximum arrival delay	One intermediate key and minimal computation.
"Compute Mean Value with MapReduce" on page 11-35	MeanMapReduceExam	Find mean arrival delay	One intermediate key with intermediate state (accumulating intermediate sum and count).
"Create Histograms Using MapReduce" on page 11-43	VisualizationMapR	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 11-38	MeanByGroupMapRed	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 11-88	HueSaturationValu	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

Example Link	Primary File	Description	Notable Programming Techniques
			be used to view the images.
"Simple Data Subsetting Using MapReduce" on page 11-52	SubsettingMapRedu	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.
"Using MapReduce to Compute Covariance and Related Quantities" on page 11-61	CovarianceMapRedu	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 11-67	StatisticsByGroup	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.

Example Link	Primary File	Description	Notable Programming Techniques
"Using MapReduce to Fit a Logistic Regression Model" on page 11-75	`LogitMapReduceExa`	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 11-82	`TSQRMapReduceExam`	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.

If you have Parallel Computing Toolbox and Bioinformatics Toolbox™, then see the `FastqMapReduceExample.m` example file in the `matlab/toolbox/bioinfo/biodemos/` folder. This example develops a `mapreduce` algorithm to analyze Next Generation Sequencing Data.

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" on page 11-25
"Execute mapreduce" on page 11-25
"Step Through Map Function" on page 11-26
"Step Through Reduce Function" on page 11-28

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`.

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 [icon] **Step** or [icon] **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 [icon] **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.

```
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'   [339]

Use add or addmulti to add more key-value pairs.
```

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.

```
1       function maxArrivalDelayReducer(intermKey, intermValIter, outKVStore)
2         % Reducer function for the MaxMapreduceExample.
3
4         % Copyright 2014 The MathWorks, Inc.
5
6         % intermKey is 'PartialMaxArrivalDelay'. intermValIter is an iterator of
7         % all values that has the key 'PartialMaxArrivalDelay'.
8  -     maxVal = -inf;
9  -     while hasnext(intermValIter)
10 ●⇒       maxVal = max(getnext(intermValIter), maxVal);
11 -     end
12        % The key-value pair add
13 ●     add(outKVStore,'MaxArriv
```
```
intermValIter: 1x1 matlab.mapreduce.ValueIterator =

    ValueIterator with properties:

      Key: 'PartialMaxArrivalDelay'

One or more values are available.
Use hasnext to check if more values are available. Use getnext to get the next value.
```

5   Now, remove the breakpoint on line 10 and use `dbcont` (or click ![Continue icon] **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 icon] **Step**) to execute line 13 only. The display for `outKVStore` shows the global maximum value that `mapreduce` will return as the final answer.

```
 1    ⊟ function maxArrivalDelayReducer(intermKey, intermValIter, outKVStore)
 2        % Reducer function for the MaxMapreduceExample.
 3
 4        % Copyright 2014 The MathWorks, Inc.
 5
 6        % intermKey is 'PartialMaxArrivalDelay'. intermValIter is an iterator of
 7        % all values that has the key 'PartialMaxArrivalDelay'.
 8  −     maxVal = -inf;
 9  −   ⊟ while hasnext(intermValIter)
10  −         maxVal = max(getnext(intermValIter), maxVal);
11  −     end
12        % The key-value pair added to outKVStore will become the output of mapreduce
13  ●     └ 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".

See Also
mapreduce

More About
- Using KeyValueStore Objects
- Using ValueIterator Objects
- "Getting Started with MapReduce" on page 11-2

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 = datastore('airlinesmall.csv', 'TreatAsMissing', 'NA');
ds.SelectedVariableNames = 'ArrDelay';
```

`datastore` returns a `TabularTextDatastore` object for the data. This datastore treats `'NA'` strings 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 =

    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 chunks 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 chunk 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)
% Mapper function for the MaxMapreduceExample.

% Copyright 1984-2014 The MathWorks, Inc.

% Data is an n-by-1 table of the ArrDelay. As the data source is tabular,
% the return of read is a table object.
partMax = max(data.ArrDelay);
add(intermKVStore, 'PartialMaxArrivalDelay',partMax);
```

The reducer receives a list of the maximum arrival delays for each chunk 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)
% 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);
```

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 =

         Key          Value
    _____   _____

    'MaxArrivalDelay'  [1014]
```

See Also
datastore | mapreduce

More About
- "Getting Started with MapReduce" on page 11-2
- "Build Effective Algorithms with MapReduce" on page 11-22

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 = datastore('airlinesmall.csv', 'TreatAsMissing', 'NA');
ds.SelectedVariableNames = 'ArrDelay';
```

`datastore` returns a `TabularTextDatastore` object for the data. This datastore treats `'NA'` strings 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 =

    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 chunks 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 chunk 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)
% Mapper function for the MeanMapReduceExample.

% Copyright 2014 The MathWorks, Inc.

% Data is an n-by-1 table of the ArrDelay. Remove missing value 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);
```

The reducer accepts the count and sum for each chunk 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)
% Reducer function for the MeanMapReduceExample.

% Copyright 2014 The MathWorks, Inc.

% intermKey is 'PartialCountSumDelay'
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);
```

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 =

          Key              Value
    _____      _____

    'MeanArrivalDelay'    [7.1201]
```

See Also
`datastore` | `mapreduce`

More About
- "Getting Started with MapReduce" on page 11-2
- "Build Effective Algorithms with MapReduce" on page 11-22

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 = datastore('airlinesmall.csv', 'TreatAsMissing', 'NA');
ds.SelectedVariableNames = {'ArrDelay', 'DayOfWeek'};
```

`datastore` returns a `TabularTextDatastore` object for the data. This datastore treats `'NA'` strings 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 =

    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 chunks 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 chunk 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)
% Mapper function for the MeanByGroupMapReduceExample.

% Copyright 2014 The MathWorks, Inc.

% 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]};
```

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.

```matlab
function meanArrivalDelayByDayReducer(intermKey, intermValIter, outKVStore)
% Reducer function for the MeanByGroupMapReduceExample.

% Copyright 2014 The MathWorks, Inc.

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);
```

Use `mapreduce` to apply the map and reduce functions to the datastore, `ds`.

```matlab
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 =

    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 =

    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 =

    DayOfWeek     MeanArrDelay
    _____     _____

    Sat           4.2095
    Tue           5.8569
    Sun           6.5241
    Wed           7.0038
    Mon           7.0833
    Thu           9.3185
    Fri           9.4193
```

See Also
datastore | mapreduce

More About

- "Getting Started with MapReduce" on page 11-2
- "Build Effective Algorithms with MapReduce" on page 11-22

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 chunks of the data. Then `mapreduce` sums the counts from all chunks. 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 = datastore('airlinesmall.csv', 'TreatAsMissing', 'NA');
ds.SelectedVariableNames = 'ArrDelay';
```

`datastore` returns a `TabularTextDatastore` object for the data. This datastore treats `'NA'` strings 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 =

    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 chunks 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 have arrival delay that in each interval
% specified by the EDGES vector, and add these counts to INTERMKVSTORE.
%

counts = histc( data.ArrDelay, edges );

add( intermKVStore, 'Null', counts );
```

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)
% get all intermediate results from the intermediate store

if hasnext(intermValList)
    outVal = getnext(intermValList);
else
    outVal = [];
end

while hasnext(intermValList)
    outVal = outVal + getnext(intermValList);
end

add(outKVStore, 'Null', outVal);
```

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')
```
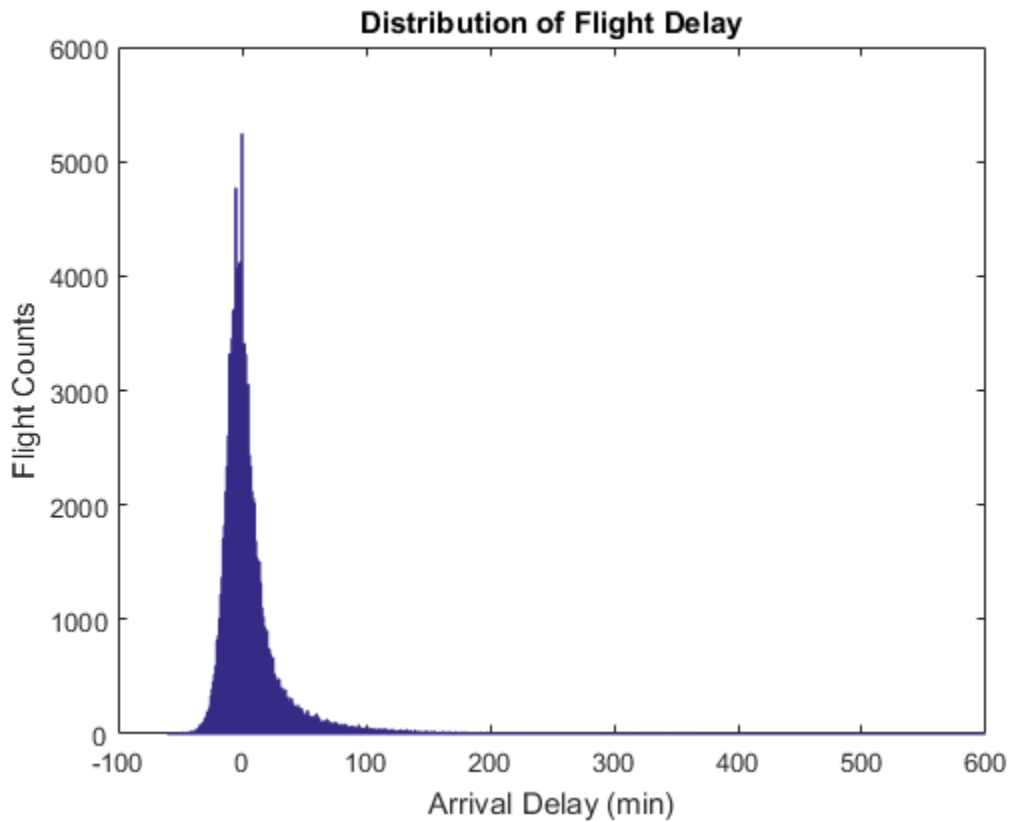
The histogram has long tails. Look at a restricted bin range to better visualize the delay distribution of the majority of flights. Zooming in a bit reveals there is a reporting artifact; it is common to round delays to 5-minute increments.

```
xlim([-50,50]);
grid on
grid minor
```
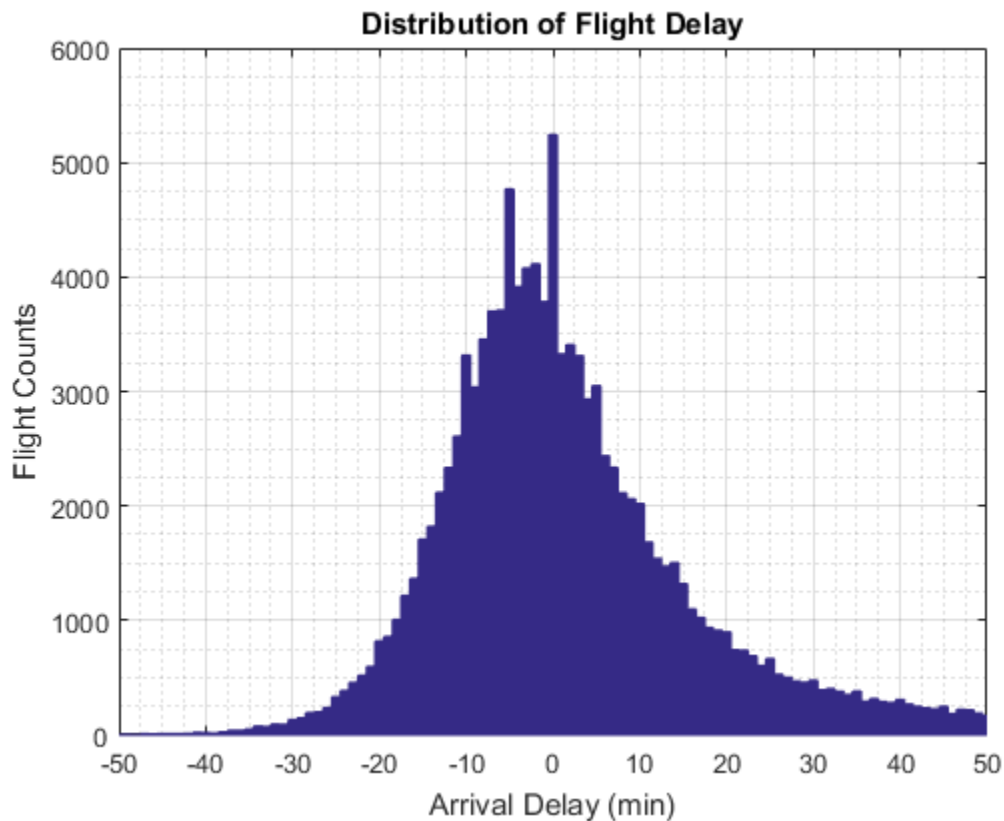
Smooth the counts with a moving average filter to remove the 5-minute recording artifact.

```
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
```
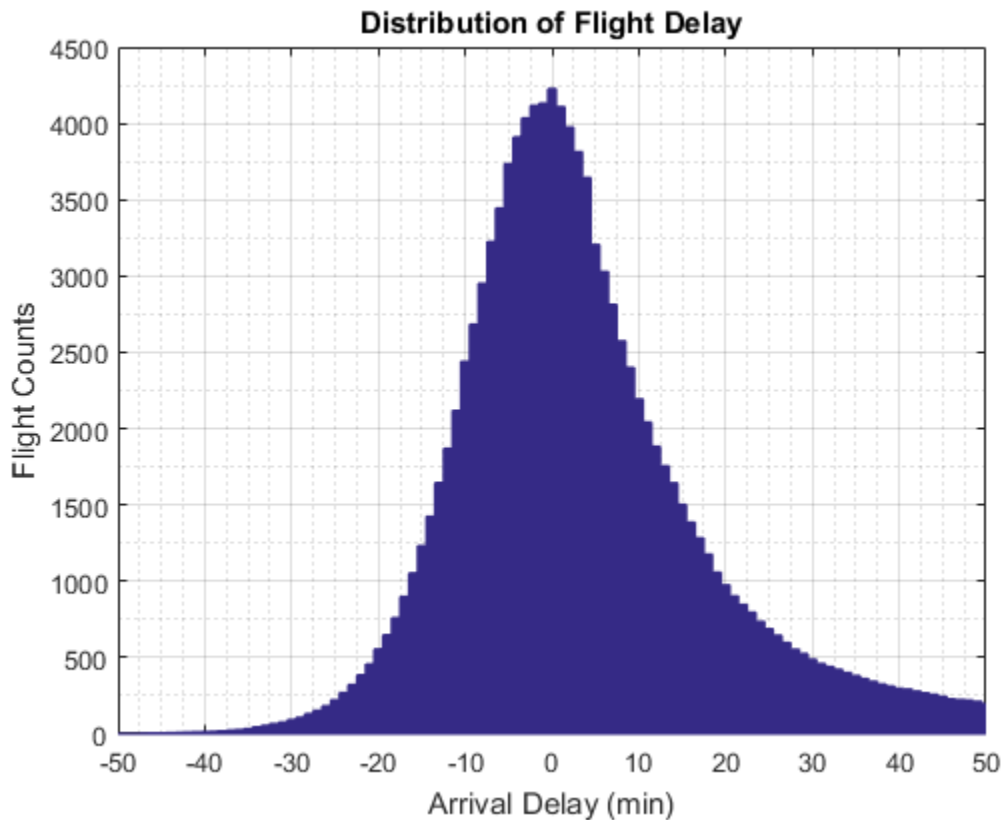
To give the graphic a better balance, do not display the top 1% of most-delayed flights. You can tailor the visualization in many ways without reprocessing the complete data set, assuming that you collected the appropriate information during the full pass through the data.

```
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')
```
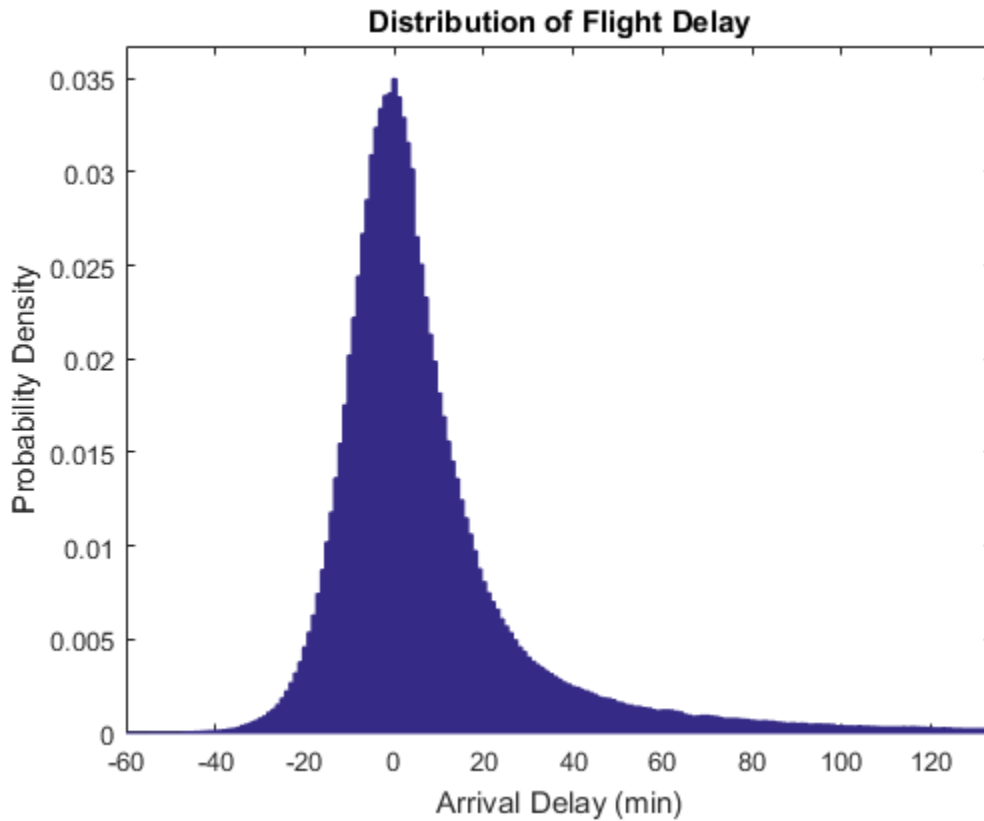

See Also

datastore | mapreduce

More About

· "Getting Started with MapReduce" on page 11-2

- "Build Effective Algorithms with MapReduce" on page 11-22

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 = datastore('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 =

  Columns 1 through 5

    'Year'    'Month'    'DepTime'    'UniqueCarrier'    'ActualElapsedTime'

  Columns 6 through 10

    'CRSElapsedTime'    'ArrDelay'    'DepDelay'    'Origin'    'Dest'

  Columns 11 through 15

    'TaxiIn'    'TaxiOut'    'CarrierDelay'    'WeatherDelay'    'NASDelay'
```

datastore returns a `TabularTextDatastore` object for the data. This datastore treats `'NA'` strings 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 =

    Year    Month    DepTime    UniqueCarrier    ActualElapsedTime    CRSElapsedTime
    ____    _____    _____    _____    _____    _____

    1987    10        642       'PS'             53                   57
    1987    10       1021       'PS'             63                   56
    1987    10       2055       'PS'             83                   82
    1987    10       1332       'PS'             59                   58
    1987    10        629       'PS'             77                   72
    1987    10       1446       'PS'             61                   65
    1987    10        928       'PS'             84                   79
    1987    10        859       'PS'            155                  143
```

Run MapReduce

The `mapreduce` function requires a map function and a reduce function as inputs. The mapper receives chunks 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).
```

11-53

```
% Copyright 2014 The MathWorks, Inc.

idx = data.Year > 1994 & (data.ActualElapsedTime - data.CRSElapsedTime)...
    > 1.50 * data.CRSElapsedTime;
intermVal = data(idx,:);

add(intermKVStore,'Null',intermVal);
```

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)
% Reducer function for the SubsettingMapReduceExample

% Copyright 2014 The MathWorks, Inc.

% 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);
```

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 =
```

Year	Month	DepTime	UniqueCarrier	ActualElapsedTime	CRSElapsedTime
1995	6	1601	'US'	162	58
1996	6	1834	'CO'	241	75
1997	1	730	'DL'	110	43
1997	4	1715	'UA'	152	57
1997	9	2232	'NW'	143	50
1997	10	1419	'CO'	196	58
1998	3	2156	'DL'	139	49
1998	10	1803	'NW'	291	81
2000	5	830	'WN'	140	55
2000	8	1630	'CO'	357	123
2002	6	1759	'US'	260	67
2003	3	1214	'XE'	214	84
2003	3	604	'XE'	175	60
2003	4	1556	'MQ'	142	52
2003	5	1954	'US'	127	48
2003	7	1250	'FL'	261	95
2003	8	2010	'AA'	339	115
2004	3	1238	'MQ'	184	69
2004	7	1730	'DL'	241	68
2004	8	1330	'XE'	204	80
2005	7	1951	'MQ'	251	97
2005	10	916	'MQ'	343	77

```
    2006    2     324    'B6'         1650              199
    2006    5    1444    'CO'          167               60
    2006    5    1250    'DL'          148               59
    2006    7    1030    'WN'          211               80
    2006    7    1424    'MQ'          254               69
    2006   11    2147    'UA'          222               77
    2006   11    1307    'AA'          175               60
    2007   10    1141    'OO'          137               54
    2008    1    1027    'MQ'          139               55
    2008    1    2049    'MQ'          151               60
    2008    2     818    'WN'          280               95
    2008    4    1014    'CO'          151               58
    2008    6    2000    'OH'          263              104
    2008    6    1715    'AA'          271               90
    2008   11    1603    'XE'          183               73
```

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 =

    Year    ArrDelay    DepDelay    TaxiIn    TaxiOut    CarrierDelay    WeatherDelay
```

____	_____	_____	_____	_____	_____	_____
1995	118	14	7	101	NaN	NaN
1996	220	54	12	180	NaN	NaN
1997	137	70	2	12	NaN	NaN
1997	243	148	4	38	NaN	NaN
1997	115	22	4	98	NaN	NaN
1997	157	19	6	95	NaN	NaN
1998	146	56	9	47	NaN	NaN
1998	213	3	11	205	NaN	NaN
2000	85	0	5	51	NaN	NaN
2000	244	10	4	273	NaN	NaN
2002	192	-1	6	217	NaN	NaN
2003	124	-6	13	131	NaN	NaN
2003	114	-1	8	106	NaN	NaN
2003	182	92	9	106	NaN	NaN
2003	78	-1	5	90	NaN	NaN
2003	166	0	11	170	0	0
2003	406	182	242	10	0	0
2004	115	0	6	61	0	0
2004	173	0	5	161	0	0
2004	124	0	9	102	0	0
2005	345	191	54	125	0	0
2005	266	0	13	183	0	0
2006	415	-1036	4	12	14	0
2006	131	24	7	118	0	6
2006	109	20	4	105	20	0
2006	226	95	5	130	0	0
2006	259	74	6	208	39	0
2006	160	15	3	158	15	0
2006	132	17	4	127	0	17
2007	107	24	7	100	0	0
2008	96	12	25	72	0	0
2008	175	84	12	107	0	0
2008	198	13	4	190	0	0
2008	92	-1	9	93	0	0
2008	204	45	12	212	0	45
2008	201	20	4	193	0	0
2008	124	14	12	93	0	0

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 http://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);
```

Create an anonymous function that performs the same selection of rows that is hard-coded in `subsettingMapper.m`.

```
inFlightDelay15Opercent = ...
    @(data) data.Year > 1994 & ...
    (data.ActualElapsedTime-data.CRSElapsedTime) > 1.5O*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.m`. Subsequently, you can use this anonymous function to call `subsettingMapperGeneric.m` using only three arguments (the fourth is implicit).

```
configuredMapper = ...
    @(data, info, intermKVStore) subsettingMapperGeneric(data, info, ...
    intermKVStore, inFlightDelay15Opercent);
```

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.
```

See Also
datastore | mapreduce

More About
- "Getting Started with MapReduce" on page 11-2

- "Build Effective Algorithms with MapReduce" on page 11-22

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 = datastore('airlinesmall.csv', 'TreatAsMissing', 'NA');
ds.SelectedVariableNames = {'ActualElapsedTime', 'Distance', ...
                            'DepDelay', 'ArrDelay'};
```

`datastore` returns a `TabularTextDatastore` object for the data. This datastore treats `'NA'` strings 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 =

    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 chunks 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 chunk of data in the datastore, `ds`. Then, the mapper stores the computed values for each chunk 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)
%covarianceMapper Mapper function for mapreduce to compute covariance

% Copyright 2014 The MathWorks, Inc.

% 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 these 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 chunk 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)
%covarianceReducer Reducer function for mapreduce to compute covariance
```

```
% Copyright 2014 The MathWorks, Inc.

% 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)
Count = results.Value{1};
MeanVal = results.Value{2};
Covariance = results.Value{3};


results =

     Key          Value
    _____     _____

    'count'     [     120664]
    'mean'      [1x4 double]
    'cov'       [4x4 double]
```

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 =

    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 =

                         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 =

    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 =

   -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
```

See Also
datastore | mapreduce

More About
- "Getting Started with MapReduce" on page 11-2
- "Build Effective Algorithms with MapReduce" on page 11-22

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 = datastore('airlinesmall.csv', 'TreatAsMissing', 'NA');
ds.SelectedVariableNames = {'Month', 'UniqueCarrier', 'ArrDelay'};
```

`datastore` returns a `TabularTextDatastore` object for the data. This datastore treats `'NA'` strings 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 =

    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 chunks 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 chunk 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)
% Mapper function for the StatisticsByGroupMapReduceExample.

% Copyright 2014 The MathWorks, Inc.

% 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]};
```

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)
% 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);
```

```matlab
outValue = struct('Count',count, 'Mean',meanVal, 'Variance',variance,...
                  'Skewness',skewnessVal, 'Kurtosis',kurtosisVal);

% add results to the output datastore
add(outKVStore,intermKey,outValue);
```

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.

```matlab
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.

```matlab
r1 = readall(outds1)
```

```
r1 =
```

```
    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]
  'MQ'          [1x1 struct]
  'OO'          [1x1 struct]
  'XE'          [1x1 struct]
  'TZ'          [1x1 struct]
  'EV'          [1x1 struct]
  'FL'          [1x1 struct]
  'B6'          [1x1 struct]
  'DH'          [1x1 struct]
  'HA'          [1x1 struct]
  'OH'          [1x1 struct]
  'F9'          [1x1 struct]
  'YV'          [1x1 struct]
  '9E'          [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 =

              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
    NW        10097     5.4265      977.64       8.616      172.87
    OH         1414     7.7617        1224        3.57       24.52
    OO         3010     5.8618      1010.4      4.4263      32.783
    PA  (1)     313     5.3738      692.19      3.2061      20.747
    PI          861     11.252      1121.1      14.751      315.59
    PS           82     5.3902      454.51      2.9682      14.383
    TW         3718      7.411      830.76       4.139       30.67
    TZ          215      1.907      814.63      2.8269      13.758
    UA        12955     8.3939      1046.6      3.9742      28.187
    US        13666     6.8027      760.83      4.6905      47.975
    WN        15749     5.4581      562.49      4.0439      30.403
    XE         2294     8.8082      1410.1      3.7114      23.235
    YV          827     12.376      2192.6      3.9315      26.446
```

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 =

           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
```

See Also
datastore | mapreduce

More About
- "Getting Started with MapReduce" on page 11-2
- "Build Effective Algorithms with MapReduce" on page 11-22

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 = datastore('airlinesmall.csv', 'TreatAsMissing', 'NA');
ds.SelectedVariableNames = {'ArrDelay', 'Distance'};
```

`datastore` returns a `TabularTextDatastore` object for the data. This datastore treats `'NA'` strings 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 =

    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 chunk of input data.

Display the map function file.

```matlab
function logitMapper(b,t,~,intermKVStore)
%logitMapper Mapper function for mapreduce to perform logistic regression.

% Copyright 2014 The MathWorks, Inc.

% 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

% We want to perform weighted least squares. We do this by computing 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].
```

```
%
% Here in the mapper we receive one chunk at a time, so we compute one of
% the terms on the right hand side. The reducer will add them up to get the
% quantity on the left hand side, and then peform 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);
```

The reducer computes the regression coefficient estimates from the sums of squares and cross products.

Display the reduce function file.

```
function logitReducer(~,intermValIter,outKVStore)
%logitReducer Reducer function for mapreduce to perform logistic regression

% Copyright 2014 The MathWorks, Inc.

% 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 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);
```

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 =

   -1.7674
    0.1209


iteration =

     2


b =

   -1.8327
    0.1807


iteration =

     3


b =

   -1.8331
    0.1806


iteration =

     4


b =

   -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]')
```
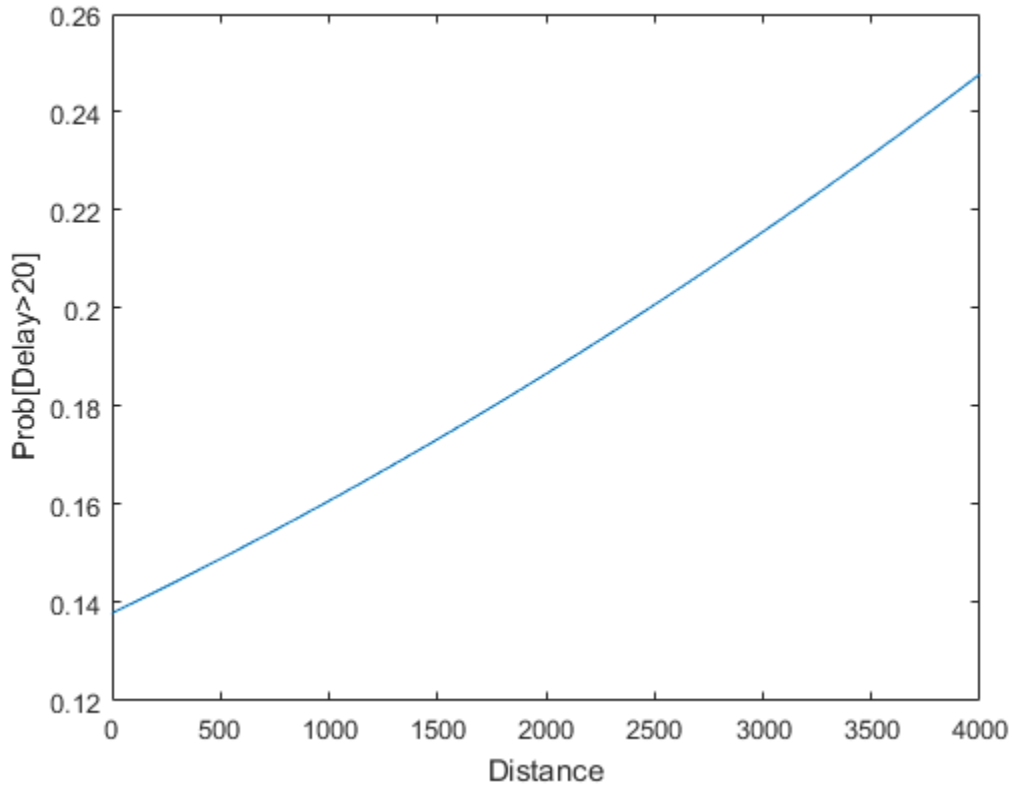
See Also
datastore | mapreduce

More About
*   "Getting Started with MapReduce" on page 11-2
*   "Build Effective Algorithms with MapReduce" on page 11-22

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 = datastore('airlinesmall.csv', 'TreatAsMissing', 'NA');
ds.ReadSize = 1000;
ds.SelectedVariableNames = {'ArrDelay', 'DepDelay', 'Distance'};
```

`datastore` returns a `TabularTextDatastore` object for the data. This datastore treats `'NA'` strings as missing and replaces the missing values with `NaN` values by default. The `RowsPerRead` property lets you specify how to partition the data into chunks. Additionally, the `SelectedVariableNames` property allows you to work with only the specified variables of interest, which you can verify using `preview`.

```
preview(ds)
```

```
ans =

    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 chunk (the ith) of data, which is a table of size $N_i \times 3$. The mapper computes the R matrix of this chunk 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 chunks 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)
% Mapper function for the TSQRMapReduceExample.

% Copyright 2014 The MathWorks, Inc.

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)
```

```
% Helper function to generate a key for the tsqrMapper function.

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);
```

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)
% Reducer function for the TSQRMapReduceExample.

% Copyright 2014 The MathWorks, Inc.

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);

outKVStore.add(intermKey,r);
```

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)
% Mapper function for the MapReduce TSQR example.
%
% This mapper function simply copies the data and add them to the
% intermKVStore as intermediate values.

% Copyright 2014 The MathWorks, Inc.

x = data.Value{:,:};
add(intermKVStore,'Identity', x);
```

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.

Display the reduce function file.

```
function tsqrReducer(intermKey, intermValIter, outKVStore)
% Reducer function for the TSQRMapReduceExample.

% Copyright 2014 The MathWorks, Inc.

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);

outKVStore.add(intermKey,r);
```

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  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 100%
```

View Results

Read the final results from the output datastore.

```
r = readall(outds2);
r.Value{:}
```

```
ans =

   1.0e+05 *

    0.1091    0.0893    0.5564
         0   -0.0478   -0.4890
         0         0    3.0130
```

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 http://doi.acm.org/10.1145/1996092.1996103

See Also
datastore | mapreduce

More About

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 = datastore({demoFolder, imsciFolder}, 'Type', 'image', ...
    'FileExtensions', {'.jpg', '.tif', '.png'})


ds =

  ImageDatastore with properties:

      Files: {
              'B:\matlab\toolbox\matlab\demos\cloudCombined.jpg';
              'B:\matlab\toolbox\matlab\demos\example.tif';
              'B:\matlab\toolbox\matlab\demos\landOcean.jpg'
               ... and 5 more
             }
    ReadFcn: @readDatastoreImage
```

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".

While the for-loop method is reasonable for small collections of images, it does not scale well to a large collection of images. This example uses `mapreduce` to accomplish the same task, however, the `mapreduce` method is highly scalable to larger collections of images.

Scale to MapReduce

- The `mapreduce` function requires a map function and a reduce function as inputs.
- The map function receives chunks 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 assoicated with 3 keys, `'Average Hue'`, `'Average Saturation'` and `'Average Brightness'`.

```matlab
function hueSaturationValueMapper(data, info, intermKVStore)
% Map function for the Hue Saturation Value MapReduce example.

% Copyright 1984-2015 The MathWorks, Inc.
    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
        add(intermKVStore, 'Average Brightness', struct('Filename', info.Filename, 'Avg
    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 reducefunction 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)
% Reduce function for the Hue Saturation Value MapReduce example.

% Copyright 1984-2015 The MathWorks, Inc.

    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 Hue
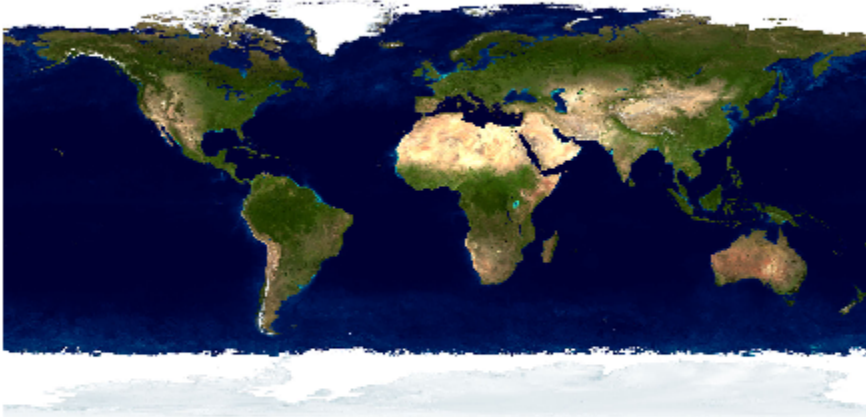
Maximum Average Brightness

See Also
datastore | mapreduce

More About
- "Getting Started with MapReduce" on page 11-2
- "Build Effective Algorithms with MapReduce" on page 11-22

Getting Started with Datastore

In this section...
"What Is a Datastore?" on page 11-95
"Create and Read from a Datastore" on page 11-96

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	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	
1	Year	Month	DayofMon	DayOfWe	DepTime	CRSDepTi	ArrTime	CRSArrTin	UniqueCa	FlightNun	TailNum	ActualEla	CRSElapse	AirTime	ArrDelay	DepDelay	Origin	Dest	Dista
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	
21	1987	10	15	4	1709	1710	1752	1749	TW	251	NA	103	99	NA	3	-1	PIT	STL	
22	1987	10	24	6	1515	1515	1544	1545	TW	283	NA	29	30	NA	-1	0	SRQ	RSW	
23	1987	10	25	7	2017	2017	2347	2329	TW	318	NA	150	132	NA	18	0	STL	BDL	
					2218	2220	2335	2322	TW						13	-2	STL	PHX	

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 for the types of data in this table. Each type of data is supported by a different type of datastore. The different types of datastores contain properties pertinent to the type of data that they support.

Type of File or Data	Datastore Type
Text files containing column-oriented data, including CSV files.	`TabularTextDatastore`
Image files, including formats that are supported by `imread` such as JPEG and PNG.	`ImageDatastore`
Key-value pair data that are inputs to or outputs of `mapreduce`.	`KeyValueDatastore`
Collections of data in a relational database. Requires Database Toolbox.	`DatabaseDatastore`

Create and Read from a Datastore

Use the `datastore` function to create a datastore. For example, create a datastore from the sample file, `airlinesmall.csv`. This file includes departure and arrival information about individual airline flights.

```
ds = datastore('airlinesmall.csv')

ds =
  TabularTextDatastore with properties:

                       Files: {
                              ' ...\matlab\toolbox\matlab\demos\airlinesmall.csv'
                              }
            ReadVariableNames: true
                VariableNames: {'Year', 'Month', 'DayofMonth' ... and 26 more}

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

  Advanced Text Format Properties:
              TextscanFormats: {'%f', '%f', '%f' ... and 26 more}
```

```
        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
```

The `datastore` function automatically determines the appropriate type of datastore to create. In this example, `datastore` creates a `TabularTextDatastore`.

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	Dis
1	Year	Month	DayofMor	DayOfWe	DepTime	CRSDepTi	ArrTime	CRSArrTin	UniqueCa	FlightNun	TailNum	ActualEla	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	
						1710	1752	1749	TW							3	-1	PIT	STL	

```
ds.SelectedVariableNames = {'DepTime','DepDelay'};
preview(ds)

ans =
    DepTime     DepDelay
    _____     _____

      642          12
     1021           1
     2055          20
     1332          12
```

11-97

```
    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` 20000 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`. For more information about MapReduce in MATLAB, see "Getting Started with MapReduce" on page 11-2.

See Also
datastore | ImageDatastore | KeyValueDatastore | TabularTextDatastore

Related Examples
- "Read and Analyze Large Tabular Text File" on page 11-102
- "Read and Analyze Image Files" on page 11-105
- "Read and Analyze MAT-File with Key-Value Data" on page 11-110

Read from HDFS

In this section...
"Specify Location of Data" on page 11-100
"Set Hadoop Environment Variable" on page 11-100
"Prevent Clearing Code from Memory" on page 11-101

Specify Location of Data

You can create a datastore for a collection of text files or sequence files that reside on the Hadoop Distributed File System (HDFS) using the `datastore` function. When you specify the location of the data, you must specify the full path to the files or folders using an internationalized resource identifier (IRI) of the form

```
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. For example, the command,

```
ds = datastore('hdfs://myserver/data/file1.txt')
```
creates a datastore for the file, `file1.txt`, in a folder named `data` located at a host named `myserver`.

hostname 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, the location

```
'hdfs://myserver:7867/data/file1.txt'
```
specifies a host named `myserver` with port `7867`, containing the file `file1.txt` in a folder named `data`. 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.

- If you work with Hadoop v1 only, set the `HADOOP_HOME` environment variable.

- If you work with 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, set the MATLAB_HADOOP_INSTALL environment variable.

For example, to set the HADOOP_HOME environment variable, type:

```
setenv('HADOOP_HOME','/mypath/hadoop-folder');
```
where *hadoop-folder* is the folder where Hadoop is installed, and */mypath/* is the path to that folder.

Prevent Clearing Code from Memory

When reading from HDFS or when reading Sequence files locally, the datastore function calls the javaaddpath command. This command:

- 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.

See Also
datastore | javaaddpath | mlock | setenv

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 chunk at a time or one file at a time.

Create a Datastore

Create a datastore from the sample file `airlinesmall.csv` using the `datastore` function. When you create the datastore, you can specify that the string, NA, in the data is treated as missing data.

```
ds = datastore('airlinesmall.csv','TreatAsMissing','NA');
```

`datastore` returns a `TabularTextDatastore`. The `datastore` function automatically determines the appropriate type of datastore to create based on the file extension.

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 =

    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 mutliple 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

datastore | TabularTextDatastore

Read and Analyze Image Files

This example shows how to create a datastore for a collection of image data, 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".

Identify two MATLAB® directories and create a datastore containing images with .jpg, .tif, and .png extensions in those directories.

```matlab
location1 = fullfile(matlabroot,'toolbox','matlab','demos');
location2 = fullfile(matlabroot,'toolbox','matlab','imagesci');

ds = datastore({location1,location2},'Type','image','FileExtensions',{'.jpg','.tif','.p
```

Initialize the maximum average HSV values and the corresponding image data.

```matlab
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).

```matlab
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 maxium average hue
            maxAvgH = avgH;
            dataH = data;
        end

        if avgS > maxAvgS        % Check for new maxium average saturation
            maxAvgS = avgS;
            dataS = data;
        end

        if avgV > maxAvgV        % Check for new maxium 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')
```

Maximum Average Hue


```
figure
imshow(dataS,'InitialMagnification','fit');
title('Maximum Average Saturation');
```

Maximum Average Saturation
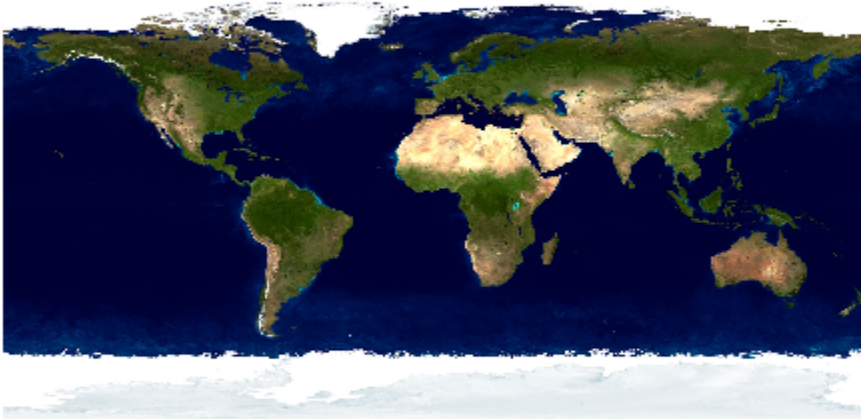
```
figure
imshow(dataV,'InitialMagnification','fit');
title('Maxium Average Brightness');
```

Maxium Average Brightness

See Also

datastore | ImageDatastore

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 =

    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 =

     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]
'MQ'        [ 3962]
'B6'        [  806]
'DH'        [  696]
'EV'        [ 1699]
'F9'        [  335]
'FL'        [ 1263]
'HA'        [  273]
'OH'        [ 1457]
'OO'        [ 3090]
'TZ'        [  216]
'XE'        [ 2357]
'9E'        [  521]
'YV'        [  849]
```

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 =

    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
'MQ'            3962
'B6'             806
'DH'             696
'EV'            1699
'F9'             335
'FL'            1263
'HA'             273
'OH'            1457
'OO'            3090
'TZ'             216
'XE'            2357
'9E'             521
'YV'             849
```

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 =

    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
'EA'         920
'PI'         871
'YV'         849
'B6'         806
'DH'         696
'9E'         521
'F9'         335
'PA (1)'     318
'HA'         273
'TZ'         216
'AQ'         154
'PS'          83
'ML (1)'      69
```

View a summary of the sorted table.

```
summary(T)


Variables:

    Airline: 29x1 cell string

    NumFlights: 29x1 double
        Values:

            min          69
            median     1457
            max       16578
```

Reset the datastore to allow rereading of the data.

```
reset(ds)
```

See Also
datastore | KeyValueDatastore

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 chunk 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
datastore | KeyValueDatastore

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 highly-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 12-3 to get started, and "Write and Read Data over TCP/IP Interface" on page 12-7 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 a TCP/IP object called `t`, using the host address shown, and `port` of `4012`.

```
t = tcpclient('www.mathworks.com', 4012)

t =

   tcpclient with properties:

          Address: 'www.mathworks.com'
             Port: 4012
          Timeout: 10
    BytesAvailable: 0
```

Create Object Using IP Address

This example creates a TCP/IP object called `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
```

Set the Timeout Property

You can optionally 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
```

The output reflects the Timeout 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 four properties.

Property	Description
Address	Specify the 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	Specify the 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	Specify 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. Must be a positive value of type `double`.

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
```

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
```

Write and Read Data over TCP/IP Interface

In this section...
"Write Data" on page 12-7
"Read Data" on page 12-7
"Acquire Data from a Weather Station Server" on page 12-8
"Read and Write uint8 Data" on page 12-9

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.

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
```

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
```