

INTRODUCTORY TUTORIAL TO MATLAB

PREPARED FOR:

ME 352: MACHINE DESIGN I
Fall Semester 2019

School of Mechanical Engineering
Purdue University
West Lafayette, Indiana 47907-2088

August 19th, 2019

Table of Contents

	Pages
Running from the Command Window.....	2
Running from an M-File.....	2
Some Matlab Basics.....	3
Some Elementary Matlab Commands	5
M-File Example: A Slider-Crank Linkage	9
Debugging and Editing M-files	10

Your lab TA will provide web links for additional information on Matlab.
Additional information on Matlab can be downloaded from the website:

www.mathworks.com

Running from the Command Window

```
> x = 2
> y = 3
> z = x*y
> z = y*sin(1.5*x)

> x = [1,2,3]
> A = [2,1,0;1,2,1;0,1,1]
> y = A*x
> x = x'
> y = A*x

> b = [1;2;3]
> lookfor determinant
> help det
> y(1) = det([b(1),A(1,2);b(2),A(2,2)])
> y(1) = det([b,A(:,1)])

> whos
> clear
> whos

> x = (0:0.01:1)*2*pi
> x = (0:0.01:1)*2*pi;
> y = sin(x);
> plot(x,y)
> z = sin(2*x);
> plot(x,y,x,z)
> xlabel('this is the x-axis')
> ylabel('this is the y-axis')
> gtext('y=sin(x)')
> gtext('z=sin(2*x)')

> b = [x;y;z];
> fid = fopen('stuff.txt','w');
> fprintf(fid,'%12.0f,%12.3f,%12.3f\n',b);
> fclose(fid)
```

Running from an M-File

- Open a new file under the File menu (choose "M-file").
- Enter the Matlab commands in the same way that you would through the Command window. For example:

```
clear
for k = 1:10
    x(k) = k;
    y(k) = k^2;
end
```

plot(x,y)

- Save the file on your *tools* directory. Matlab will automatically append a “.m” extension at the end of whatever name that you give to the file. For example, if the file is to be named “main”, then Matlab will save the file as “main.m”.
- Return to the Command window.
- While in the Command window, move to the directory containing your saved file.
- To run the file, simply type the name of the file (without the “.m” extension).

Some Matlab Basics

The following are some basic things about Matlab that you will need to know before using the application. The list is by no means complete. You will continually learn new things every time that you use Matlab.

- (a) Command window. The Command window is the window that first appears when you first get onto Matlab. This is the window within which you will be doing all your Matlab operations. Think of this window as being a calculator, a calculator that will allow you to perform simple arithmetic operations, scientific calculations, plotting, exporting/ importing data to/from M-files, etc.
- (b) Script files (M-files). If the amount of instructions given to Matlab become long and involved and will be performed repeatedly, you will find it convenient to place your instructions in a script file (known as an M-file) that can be executed as a program.
 - to *create a new M-file*, open a new file under the File menu. Within the Matlab Editor, type your Matlab commands and then save the file on your tools directory.
 - To *modify an existing M-file*, use the Matlab Editor. Within the Matlab Editor window, make the desired changes to the file. Be sure to save this file before leaving the Matlab Editor window.
 - To *run an existing M-file*, simply type the name of the M-file (without the “.m” extension) while in the Command window. Be sure that you are in the directory containing your saved file.
- (c) Matlab variables. All variables in Matlab are *matrices*¹. That is, a single variable, say x , is made up of rows and columns. The symbol $x(i,j)$ represents an element on the *i*th row and *j*th column of x .
 - A number, or *scalar*, is a matrix having only a single row and column.
 - A *row vector* is a matrix having a single row (e.g., $x(1,n)$).
 - A *column vector* is a matrix having a single column (e.g., $x(n,1)$).

WARNING: Actually, all variables in Matlab are also *complex*. Unless otherwise defined, however, the variables will contain only real parts. Within Matlab, the

¹ A matrix is equivalent to an *array* in Fortran or C with the number of indices restricted to two.

symbols **i** and **j** are reserved for the imaginary number $\sqrt{-1}$. That is, if you want to define a complex number $x = 2 + 3i$, you would simply use the command:

$$\mathbf{x = 2 + 3*i}$$

You can use the symbols **i** and **j** as name for variables other than $\sqrt{-1}$; however, you must define them as such before they are used. Just be careful.

(d) Variable names. The names used for Matlab variables can be any number/combination of alphabetic characters, numeric characters and the underscore symbol ('_').

- The first character MUST be alphabetic.
- Characters beyond the 19th are IGNORED.
- Punctuation marks are NOT allowed in the variable names as they have special meaning in Matlab.
- The variable names are *case sensitive*. That is, the following variable names will correspond to different variables: myname, MyName, mYnaMe, etc.
- Blanks are NOT allowed in a variable name. Use the underscore for a blank if you feel that a blank would be useful helping to understand the meaning of a variable name. For example, the name "input_angle" may be easier to read than "inputangle".

(e) Basic arithmetic operations. Matlab supports all the basic arithmetic operations of addition (+), subtraction (-), multiplication (*), division(/) and exponentiation (^). Some details on these operations in Matlab:

- If **a** and **b** are both $(n \times m)$ matrices, then $a \pm b =$ an $(n \times m)$ matrix **c** for which $c(i,j) = a(i,j) \pm b(i,j)$.
- If **a** and **b** are $(n \times m)$ and (m,p) matrices², respectively, then $a * b$ is an $(n \times p)$ matrix **c** defined by the usual matrix algebra multiplication rules:

$$c(i,j) = \sum_{k=1}^m a(i,k) * b(k,j)$$

- In general, a/b is not a meaningful operation when **a** and **b** are non-scalar matrices. For work that you will be doing in this course, the division operation will be done only on *scalar variables*.
- If **a** is a *square* $(n \times n)$ matrix, then a^p is an $(n \times n)$ matrix **b** made up of **a** multiplied by itself **p** times (note that $b(i,j) \neq a(i,j)^p$ if **p** is an integer. If **a** is a scalar, then $b = a^p$).
- Preceding these operational symbols with a "." (such as: ".+", ".-", ".*", "./" or ".^") allows that operation to be performed on the *individual components* of

² This says that the number of *rows* in the matrix **a** must be the same as the number of *columns* in the matrix **b**. If this is not true and you have Matlab attempt this multiplication operation, it will complain very loudly!

the matrices. For example, if a and b are $(n \times m)$ and $(n \times m)$ matrices, respectively, then $a .* b$ is an $(n \times m)$ matrix c defined by:

$$c(i,j) = a(i,j) * b(i,j)$$

- (f) Elementary math functions. Matlab supports the elementary math functions of \sin , \cos , \tan , asin , acos , atan , atan2 , etc. The functions operate on each element of the matrix individually. For example, if a is an $(n \times m)$ matrix, then $b = \sin(a)$ is also an $(n \times m)$ matrix with $b(i,j) = \sin(a(i,j))$.

Some Elementary Matlab Commands

The following are some fundamental Matlab commands and symbols that you may need in your work in this course.

- (1) Getting help. The **help** command is likely to be the most helpful (pardon the redundancy) command that you will use in Matlab. Typing

help <topic>

while in the Command window will provide you with details on that topic (provided that it exists). These details will include a description of the function, the input list and (generally) an example of how it is used.

While the **help** command allows you to access help, it is NOT convenient to use if you do not know the name or the exact spelling of the name of the particular topic (which is often the case when you are first beginning). An alternate approach on obtaining help is the usage of the **lookfor** command. Typing

lookfor <keyword>

while in the Command window will search through all the first lines of the Matlab help topics and returning those that contain that keyword. This can be a lengthy process and generally takes much more time than using the **help** command.

- (2) Clearing variables. Variables used in Matlab remain in memory until they are removed by the **clear** command or until you exit from Matlab. The **clear** command can be used to remove *selected* variables with

clear name1 name2 name3

where name1, name2 and name3 are existing variables, or it can be used to remove *all* variables with

clear

It is a good practice to make the first line of a M-file a **clear** if this M-file is a main program. This eliminates all existing variables and will avoid conflict with variables used within main program M-file.

- (3) Print suppression. Matlab has the (often bothersome) characteristic of providing a screen echo of the result for all calculations. To suppress this echo, terminate each command line with a semi-colon “;”. However, having an echo of a calculation can be useful as a debugging tool. During the debugging phase, you may want to remove the semi-colon on select lines to help track your calculations.
- (4) Comment lines. All text found after a percent sign “%” is taken as a comment statement and is ignored by Matlab.
- (5) Continuation. If a Matlab statement is too long and needs to be continued onto the next line, break the statement with three periods (“...”) followed by an Enter. For example, the following is treated as a *single* command:

```
x=2*cos(t)+4*ln(y)+a*b+ ...
20*det(d);
```

- (6) Identifying existing variables. During the debugging phase of your programming, it is often useful to get information on the variables that currently exist in the memory for Matlab. Typing

```
whos
```

while in the Command window will provide a listing of all variables that currently exist along with size and type of variables.

- (7) Determining the size of matrices. The **whos** command is useful for determining the size of matrices while in the Command window. Often times it is necessary to determine the number of rows and columns of a variable during the execution of an M-file and to store these numbers as variables for later use. To find the number of rows and columns of a matrix “a”, use

```
[nr,nc] = size(a);
```

where nr and nc are the returned values for the number of rows and columns, respectively, for the matrix a.

- (8) Creating a matrix. The following three approaches all create the *same* (3 × 2) matrix a:

$$a = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$

- **a(1,1)=1;**
- a(1,2)=2;**
- a(1,3)=3;**
- a(2,1)=4;**
- a(2,2)=5;**

a(2,3)=6;

- **a = [1,2,3;4,5,6];**

From above we see that the delimiter “,” moves the next variable to the next *column*, whereas the delimiter “;” moves the next variable to the beginning of the next *row*.

- **a = [(1:3);(4:6)]**

(From above we see that the operation **(1:3)** creates a *row* vector of [1 2 3] and the operation **(4:6)** creates the *row* vector [4 5 6].)

- (9) Transpose of a matrix. The mathematical operation of “transposing” a matrix switches the rows and columns of the matrix. In Matlab, transpose operation is performed with a single quote (‘) following the matrix. Using the (3 × 2) matrix a above, the Matlab command of:

b = a’;

produces the (2 × 3) matrix b of:

$$b = \begin{bmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{bmatrix}$$

Note that if *c* is a *row* vector with *n* elements, **c’** creates a *column* vector with *n* elements. Similarly, if *c* is a *column* vector with *n* elements, **c’** creates a *row* vector with *n* elements.

- (10) “For” loops³. If a set of Matlab operations are to be repeated a prescribed number of times, a “for” loop will prove to be useful. For example, the loop:

```
for k = 1:5  
    x(1,k) = k;  
end
```

creates a row vector of:

$$x = [1 \quad 2 \quad 3 \quad 4 \quad 5]$$

(In the above loop, the commands between the “for” and “end” are repeated until *k* exceeds the value of 5.)

- (11) “While” loops. If a set of Matlab operations are to be repeated an indefinite number of times, a “while” loop will prove to be useful. For example, the loop:

```
k = 1;
```

³ The Matlab “for” loop is equivalent to the Fortran “do” loop.

```

while k <= 5
    x(1,k) = k;
    k=k+1;
end

```

creates a row vector of:

$$x = [1 \quad 2 \quad 3 \quad 4 \quad 5]$$

In the above loop, the commands between the “while” and “end” are repeated until k exceeds the value of 5.

(12) Some special elementary matrices:

- **x=eye(n)** creates an $(n \times n)$ *identity* matrix. That is, $x(i,j) = 1$ for $i = j$ and $x(i,j) = 0$ for $i \neq j$.
- **x=zeros(n,m)** creates an $(n \times m)$ matrix of zeros. That is, $x(i,j) = 0$ for all i and j .
- **x=ones(n,m)** creates an $(n \times m)$ matrix of ones. That is, $x(i,j) = 1$ for all i and j .

(13) Plotting. Making 2-D plots with Matlab is a very easy operation. Say that we have two row vectors x and y . To make an x - y plot (x on the abscissa and y on the ordinate), simply use:

```
plot(x,y)
```

For example, to make a plot of the function $y = \sin(x)$ for $0 \leq x \leq 2\pi$, use:

```

x=(0:0.1:1)*2*pi;
y=sin(x);
plot(x,y)

```

(Note that the command **x=(0:0.1:1)*2*pi** above creates a row vector with $0 \leq x \leq 2\pi$ in increments of 0.1). There are many options available with the plot command. Do a **help plot** to find out what options are available. There are many Matlab commands that are related to plotting that you will find useful, among which are: **xlabel**, **ylabel**, **title**, **gtext**, **axis**. Note that these five commands act on the *current* plot; that is, they must be executed *after* the plot command.

(14) Changing plot windows. Subsequent usage of the **plot** command will *erase* the current plot window before replotting. If you desire to keep a plot window open without being erased, use the **figure** command. For example, say that you have a plot in the window “Figure(1)” that you want to keep for a while. Before the next **plot** command, use:

```
figure(2)
```

to open a new plot window named “Figure(2)”. The next usage of the **plot** command will place the plot in the window “Figure(2)”.

Also typing **figure(n)** will bring an existing “Figure(n)” window to the foreground of the screen.

- (15) Creating tabular output. The easiest way to produce tabular output that can be printed as a hard copy or that can be copied into a word processor for your reports is the **fprintf** command. The **fprintf** command creates a formatted output to a file.

Say that you have three row vectors x , y and z for which you want x , y and z to form columns of a table. This table is to be stored in a file named “results”. This can be done as follows:

```
x=(0:0.1:1)*pi;
y=sin(x);
z=cos(x);
a=[x;y;z];
fid=fopen('results.txt','w')
fprintf(fid,'%6.1f %9.3f %9.3f \n',a);
fclose(fid)
```

The formatting syntax used in **fprintf** is the same as is used in the C programming language.

For more details on this, please use the **help fprintf** command.

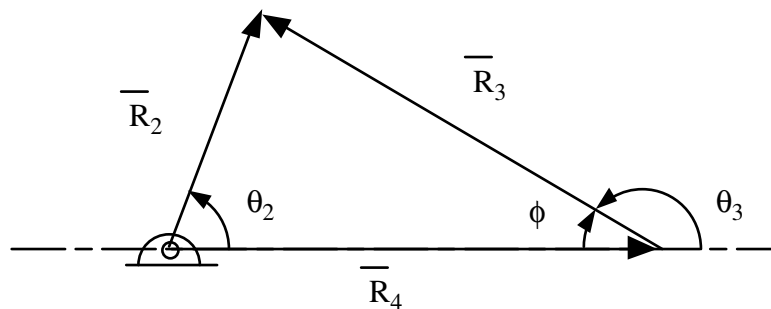
- (16) Making a hardcopy of a plot.

Say that you have made a plot which appears in the plot window named “Figure (2)”. Now you want to make a hardcopy of this plot.

With the Figure (2) window active, simply use the “Print” command under the File menu.

If you want to paste a copy of your plot into a Word document, use the “Copy Figure” command under the Edit menu while the Figure(2) window is active. Then simply paste the figure into your Word document.

M-File Example: A Slider-Crank Linkage



The Law of sines:

$$\frac{\sin \phi}{R_2} = \frac{\sin \theta_2}{R_3} \quad \text{and} \quad \theta_3 = \pi - \phi$$

The Law of cosines: $R_4^2 = R_2^2 + R_3^2 - 2R_2 R_3 \cos(\pi - \theta_2 - \phi)$

Matlab Code for the Position Solution of the Slider-Crank Mechanism.

```
clear
R2=1;
R3=3;
nt=360;
theta2=2*pi*(0:nt-1)/(nt-1);
for ii=1:nt
    phi(ii)=asin(R2*sin(theta2(ii))/R3);
    theta3(ii)=pi-phi(ii);
    R4(ii)=sqrt(R2^2+R3^2-2*R2*R3*cos(pi-theta2(ii)-phi(ii)));
end
plot(theta2*180/pi,R4)
```

Alternate Matlab Code (using ability of Matlab functions to evaluate matrices)

```
clear
R2=1;
R3=3;
nt=360;
theta2=2*pi*(0:nt-1)/(nt-1);
phi=asin(R2*sin(theta2)/R3);
theta3=pi-phi;
R4=sqrt(R2^2+R3^2-2*R2*R3*cos(pi-theta2-phi));
plot(theta2*180/pi,R4)
```

Debugging and Editing M-Files

This information is provided by Mathworks on their website and is provided here as a quick reference. We recommend that you visit http://www.mathworks.com/access/helpdesk/help/techdoc/matlab_env/edit_d25.html#9240 or http://www.mathworks.com/access/helpdesk/help/techdoc/matlab_env/matlab_env.html for various other Matlab tutorials. We also recommend that you read through the tutorials on your own before asking for help on debugging your code. There are several methods for creating, editing, and debugging files with MATLAB.

Creating and Editing Files-- Options	Instructions
MATLAB Editor/Debugger	See Starting, Customizing, and Closing the Editor/Debugger , and Creating, Editing, and Running Files . You can create, open, edit and save M-files as well as other file types in the MATLAB Editor/Debugger--see Creating and Editing Other Text File Types .
MATLAB stand-alone Editor (without running MATLAB)	See Opening the Editor Without Starting MATLAB .
Any text editor, such as Emacs or vi	To specify another editor as the default for use with MATLAB, select File -> Preferences -> Editor/Debugger , and for Editor , specify the Text editor . Click Help in the Preference dialog box for details. You can then debug M-files using the MATLAB Editor/Debugger or debugging functions.

Debugging M-Files-Options	Instructions
General debugging tips	See Finding Errors in M-Files .
MATLAB Editor/Debugger	See Debugging Process and Features .
MATLAB debugging functions (for use in the Command Window)	See Debugging Process and Features .

Use [preferences for the Editor/Debugger](#) to set up the editing and debugging environment to best meet your needs.

For information about the MATLAB language and writing M-files, see the [MATLAB Programming](#) documentation.

Debugging M-Files

This section introduces general techniques for finding errors in M-files. It then illustrates MATLAB debugger features found in the Editor/Debugger as well equivalent Command Window debugging functions, using a simple example. It includes these topics:

- I. [Finding Errors in M-Files](#)
- II. [Debugging Example--The Collatz Problem](#)
- III. [Debugging Process and Features](#)

I. Finding Errors in M-Files

Debugging is the process by which you isolate and fix problems with your code. Debugging helps to correct two kinds of errors:

Syntax errors--For example, misspelling a function name or omitting a parenthesis.

Run-time errors--These errors are usually algorithmic in nature. For example, you might modify the wrong variable or code a calculation incorrectly. Run-time errors are usually apparent when an M-file produces unexpected results. Run-time errors are difficult to track down because the function's local workspace is lost when the error forces a return to the MATLAB base workspace.

In addition to finding and fixing problems with your M-files, you might want to improve the performance and make other enhancements using MATLAB tools.

Use the following techniques to isolate the causes of errors and improve your M-files.

Technique or Tool	Description	For More Information
Syntax highlighting	Syntax highlighting helps you identify syntax errors in an M-file before you run the file.	Syntax Highlighting
Error messages	When you run an M-file with a syntax error, MATLAB will most likely detect it and display an error message in the Command Window describing the error and showing its line number in the M-file. Click the underlined portion of the error message, or position the cursor within the message and press Ctrl+Enter . The offending M-file opens in the Editor/Debugger, scrolled to the line containing the error. To check for syntax errors in an M-file without running the M-file, use the pcode function.	none
Editor/Debugger and Debugging Functions	The MATLAB Editor/Debugger and debugging functions are useful for correcting run-time errors because you can access function workspaces and examine or change the values they contain. You can set and clear <i>breakpoints</i> , indicators that temporarily halt execution in an M-file. While stopped at a breakpoint, you can change workspace contexts, view the function call stack, and execute the lines in an M-file one by one.	Debugging Example--The Collatz Problem and Debugging Process and Features
Cells	In the Editor/Debugger, isolate sections of an M-file, called cells, so you can easily make changes to and run a single section.	Rapid Code Iteration Using Cells
M-Lint	Use M-Lint to help you verify the integrity of your code and learn about potential improvements. Access M-Lint from the Editor/Debugger by	M-Lint Code Check Report

	selecting Tools -> Check Code with M-Lint .	
Profiler	Use the Profiler to help you improve performance and detect problems in your M-files. Access the Profiler from the Editor/Debugger by selecting Tools -> Open Profiler .	Profiling for Improving Performance
Visual Directory and M-File Reports	The Visual Directory tool and M-File Reports can help you polish and package M-files before providing them to others to use. Access all of these tools from the Current Directory browser. You can run one of these, the Dependency Report, for the current file directly from the Editor/Debugger--select Tools -> Show Dependency Report .	Visual Directory in Current Directory Browser and Directory Reports in Current Directory Browser

Other Useful Techniques for Finding and Correcting Errors

Add keyboard statements to the M-file--keyboard statements stop M-file execution at the point where they appear and allow you to examine and change the function's local workspace. This mode is indicated by a special prompt:

K>>

Resume function execution by typing `return` and pressing the **Enter** key. For more information, see the [keyboard](#) reference page.

Remove selected semicolons from the statements in your M-file--Semicolons suppress the display of output in the M-file. By removing the semicolons, you instruct MATLAB to display these results on your screen as the M-file executes.

List dependent functions--Use the [depfun](#) function to see the dependent functions. Similarly, use the dependency report in the Visual Directory tools.

II. Debugging Example--The Collatz Problem

The example debugging session requires you to create two M-files, `collatz.m` and `collatzplot.m`, that produce data for the Collatz problem.

For any given positive integer, n , the Collatz function produces a sequence of numbers that always resolves to 1. If n is even, divide it by 2 to get the next integer in the sequence. If n is odd, multiply it by 3 and add 1 to get the next integer in the sequence. Repeat the steps until the next integer is 1. The number of integers in the sequence varies, depending on the starting value, n .

The Collatz problem is to prove that the Collatz function will resolve to 1 for all positive integers. The M-files for this example are useful for studying the Collatz problem. The file `collatz.m` generates the sequence of integers for any given n . The file `collatzplot.m` calculates the number of integers in the sequence for all integers from 1 through m , and plots the results. The plot shows patterns that can be further studied.

Following are the results when n is 1, 2, or 3.

n	Sequence	Number of Integers in the Sequence
1	1	1
2	2 1	2
3	3 10 5 16 8 4 2 1	8

M-Files for the Collatz Problem

Following are the two M-files you use for the debugging example. To create these files on your system, open two new M-files. Select and copy the following code from the Help browser and paste it into the M-files. Save and name the files `collatz.m` and `collatzplot.m`. Save them to your current directory or add the directory where you save them to the search path. One of the files has an embedded error to illustrate the debugging features.

Code for `collatz.m`.

```

function sequence=collatz(n)
% Collatz problem. Generate a sequence of integers resolving to 1
% For any positive integer, n:
%   Divide n by 2 if n is even
%   Multiply n by 3 and add 1 if n is odd
%   Repeat for the result
%   Continue until the result is 1%

sequence = n;
next_value = n;
while next_value > 1
    if rem(next_value,2)==0
        next_value = next_value/2;
    else
        next_value = 3*next_value+1;
    end
    sequence = [sequence, next_value];
end

```

Code for collatzplot.m.

```

function collatzplot(m)
% Plot length of sequence for Collatz problem
% Prepare figure
clf
set(gcf,'DoubleBuffer','on')
set(gca,'XScale','linear')
%
% Determine and plot sequence and sequence length
for N = 1:m
    plot_seq = collatz(N);
    seq_length(N) = length(plot_seq);
    line(N,plot_seq,'Marker','.', 'MarkerSize',9, 'Color','blue')
    drawnow
end

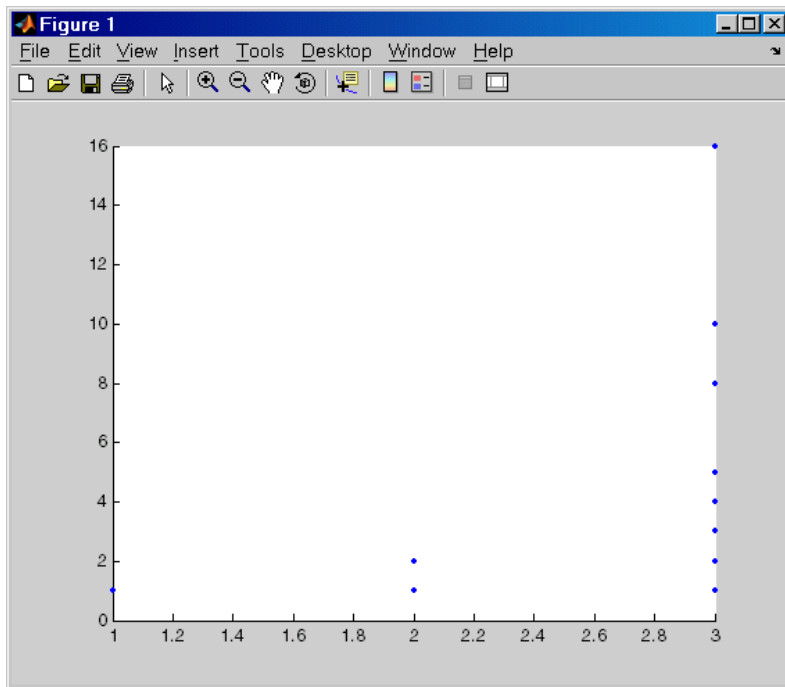
```

Trial Run for Example

Try out `collatzplot` to see if it works correctly. Use a simple input value, for example, 3, and compare the results to those shown in the preceding table. Typing

```
collatzplot(3)
```

produces the plot shown in the following figure.



The plot for $n = 1$ appears to be correct--for 1, the Collatz series is 1, and contains one integer. But for $n = 2$ and $n = 3$, it is wrong because there should be only one value plotted for each integer, the number of integers in the sequence, which the preceding table shows to be 2 (for $n = 2$) and 8 (for $n = 3$). Instead, multiple values are plotted. Use MATLAB debugging features to isolate the problem.

III. Debugging Process and Features

You can debug the M-files using the Editor/Debugger, which is a graphical user interface, as well as using debugging functions from the Command Window. You can use both methods interchangeably. The example describes both methods. The debugging process consists of

- [Preparing for Debugging](#)
- [Setting Breakpoints](#)
- [Running an M-File with Breakpoints](#)
- [Stepping Through an M-File](#)
- [Examining Values](#)
- [Correcting Problems and Ending Debugging](#)

Some additional debugging options include

- [Conditional Breakpoints](#)
- [Breakpoints in Anonymous Functions](#)
- [Error Breakpoints](#)

Preparing for Debugging

Do the following to prepare for debugging:

- Open the file--To use the Editor/Debugger for debugging, open it with the file to run.
- Save changes--If you are editing the file, save the changes before you begin debugging. If you try to run a file with unsaved changes from within the Editor/Debugger, the file is automatically saved before it runs. If you run a file with unsaved changes from the Command Window, MATLAB runs the saved version of the file, so you will not see the results of your changes.

- Add the files to a directory on the search path or put them in the current directory--Be sure the file you run and any files it calls are in directories that are on the search path. If all files to be used are in the same directory, you can instead make that directory be the current directory.

Example--Preparing for Debugging

Open the file `collatzplot.m`. Make sure the current directory is the directory in which you saved `collatzplot`.

Setting Breakpoints

Set breakpoints to pause execution of the function so you can examine values where you think the problem might be. You can set breakpoints in the Editor/Debugger, using functions in the Command Window, or both.

There are three basic types of breakpoints you can set in M-files:

- A standard breakpoint, which stops at a specified line in an M-file. For details, see [Setting Standard Breakpoints](#).
- A conditional breakpoint, which stops at a specified line in an M-file only under specified conditions. For details, see [Conditional Breakpoints](#).
- An error breakpoint that stops in any M-file when it produces the specified type of warning, error, or NaN or infinite value. For details, see [Error Breakpoints](#).


You can disable standard and conditional breakpoints so that MATLAB temporarily ignores them, or you can remove them. For details, see [Disabling and Enabling Breakpoints](#). Breakpoints are not maintained after you exit the MATLAB session.

You can only set valid standard and conditional breakpoints at executable lines in saved files that are in the current directory or in directories on the search path. When you add or remove a breakpoint in a file that is not in a directory on the search path or in the current directory, a dialog box appears, presenting you with options that allow you to add or remove the breakpoint. You can either change the current directory to the directory containing the file, or you can add the directory containing the file to the search path.

You cannot set breakpoints while MATLAB is busy, for example, running an M-file, unless that M-file is paused at a breakpoint.

Setting Standard Breakpoints

To set a standard breakpoint using the Editor/Debugger, click in the breakpoint alley at the line where you want to set the breakpoint. The breakpoint alley is the narrow column on the left side of the Editor/Debugger, just right of the line number. Set breakpoints at lines that are preceded by a - (dash). Lines not preceded by a dash, such as comments or blank lines, are not executable--if you try to set a breakpoint there, it is actually set at the next executable line.

Other ways to set a breakpoint are to position the cursor in the line and then click the Set/Clear Breakpoint button  on the toolbar, or select **Set/Clear Breakpoint** from the **Debug** menu or the context menu. A breakpoint icon appears.

Set Breakpoints for the Example. It is unclear whether the problem in the example is in `collatzplot` or `collatz`. To start, set breakpoints in `collatzplot.m` at lines 10, 11, and 12. The breakpoint at line 10 allows you to step into `collatz` to see if the problem might be there. The breakpoints at lines 11 and 12 stop the program where you can examine the interim results.

Click where there is a - (dash) to set a breakpoint at that line. The red icon indicates a valid breakpoint is set at that line.

```

1 function collatzplot(m)
2 % Plot length of sequence for Collatz problem
3 % Prepare figure
4 - clf
5 - set(gcf,'DoubleBuffer','on')
6 - set(gca,'XScale','linear')
7 %
8 % Determine and plot sequence and sequence length
9 - for N = 1:m
10 - plot_seq = collatz(N);
11 - seq_length(N) = length(plot_seq);
12 - line(N,plot_seq,'Marker','.', 'MarkerSize',9, 'Color','blue')
13 - drawnow
14 - end

```

Valid (Red) and Invalid (Gray) Breakpoints. Red breakpoints are valid standard breakpoints. If breakpoints are instead gray, they are not valid.

When breakpoints are gray, they are not valid. In this example, it is because the file has not been saved since changes were made to it. Save the file to make the breakpoints valid (red).

```

1 function collatzplot(m)
2 % Plot length of sequence for Collatz problem
3 % Prepare figure
4 - clf
5 - set(gcf,'DoubleBuffer','on')
6 - set(gca,'XScale','linear')
7 %
8 % Determine and plot sequence and sequence length
9 - for N = 1:m
10 - plot_seq = collatz(N);
11 - seq_length(N) = length(plot_seq);
12 - line(N,plot_seq,'Marker','.', 'MarkerSize',9, 'Color','blue')
13 - drawnow
14 - end

```

Breakpoints are gray for either of these reasons:

- The file has not been saved since changes were made to it. Save the file to make breakpoints valid. The gray breakpoints become red, indicating they are now valid. Any gray breakpoints that were entered at invalid breakpoint lines automatically move to the next valid breakpoint line with the successful file save.
- There is a syntax error in the file. When you set a breakpoint, an error message appears indicating where the syntax error is. Fix the syntax error and save the file to make breakpoints valid.

Function Alternative for Setting Breakpoints

To set a breakpoint using the debugging functions, use [dbstop](#). For the example, type

```
dbstop in collatzplot at 10
dbstop in collatzplot at 11
dbstop in collatzplot at 12
```

Some useful related functions are

- [dbtype](#)--Lists the M-file with line numbers in the Command Window.
- [dbstatus](#)--Lists breakpoints.

Running an M-File with Breakpoints

After setting breakpoints, run the M-file from the Command Window or the Editor/Debugger.

Running the Example

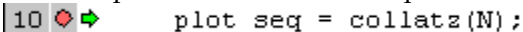
For the example, run `collatzplot` for the simple input value, 3, by typing in the Command Window

```
collatzplot(3)
```

The example, `collatzplot`, requires an input argument and therefore runs only from the Command Window and not from the Editor/Debugger.

Results of Running an M-File Containing Breakpoints

Running the M-file results in the following:

- The prompt in the Command Window changes to
 - `K>>`
- indicating that MATLAB is in debug mode.
- The program pauses at the first breakpoint. This means that line will be executed when you continue. The pause is indicated in the Editor/Debugger by the green arrow just to the right of the breakpoint, which in the example, is line 10 of `collatzplot` as shown here.






```
10 plot_seq = collatz(N);
```
- If you use debugging functions from the Command Window, the line at which you are paused is displayed in the Command Window. For the example, it would show
 - `10 plot_seq = collatz(N);`
- The function displayed in the **Stack** field on the toolbar changes to reflect the current function (sometimes referred to as the caller or calling workspace). The call stack includes subfunctions as well as called functions. If you use debugging functions from the Command Window, use [dbstack](#) to view the current call stack.
- If the file you are running is not in the current directory or a directory on the search path, you are prompted to either add the directory to the path or change the current directory.

In debug mode, you can set breakpoints, step through programs, examine variables, and run other functions.

Stepping Through an M-File

While debugging, you can step through an M-file, pausing at points where you want to examine values.

Use the step buttons or the step items in the **Debug** menu of the Editor/Debugger or desktop, or use the equivalent functions.

Toolbar Button	Debug Menu Item	Description	Function Alternative
	Continue o., Run o., Save and Run	Continue execution of M-file until completion or until another breakpoint is encountered. The menu item says Run or Save and Run if a file is not already running.	dbcont
None	Go Until Cursor	Continue execution of M-file until the line where the cursor is positioned. Also available on the context menu.	None
	Step	Execute the current line of the M-file.	dbstep
	Step In	Execute the current line of the M-file and, if the line is a call to another function, step into that function.	dbstep in
	Step Out	After stepping in, run the rest of the called function or subfunction, leave the called function, and pause.	dbstep out

Continue Running in the Example

In the example, `collatzplot` is paused at line 10. Because the problem results are correct for $N/n = 1$, we want to continue running until $N/n = 2$. Press the Continue button three times to move through the breakpoints at lines 10, 11, and 12. Now the program is again paused at the breakpoint at line 10.

Stepping In to Called Function in the Example

Now that `collatzplot` is paused at line 10 during the second iteration, use the Step In button or type `dbstep in` in the Command Window to step into `collatz` and walk through that M-file. Stepping into line 10 of `collatzplot` goes to line 9 of `collatz`. If `collatz` is not open in the Editor/Debugger, it automatically opens if you have selected **Debug -> Open M-Files When Debugging**.

The pause indicator at line 10 of `collatzplot` changes to a hollow arrow \curvearrowright , indicating that MATLAB control is now in a subfunction called from the main program. The call stack shows that the current function is now `collatz`.

In the called function, `collatz` in the example, you can do the same things you can do in the main (calling) function--set breakpoints, run, step through, and examine values.

Examining Values

While the program is paused, you can view the value of any variable currently in the workspace. Examine values when you want to see whether a line of code has produced the expected result or not. If the result is as expected, continue running or step to the next line. If the result is not as expected, then that line, or a previous line, contains an error. Use the following methods to examine values:

- [Selecting the Workspace](#)
- [Viewing Values as Datatips in the Editor/Debugger](#)
- [Viewing Values in the Command Window](#)
- [Viewing Values in the Workspace Browser and Array Editor](#)
- [Evaluating a Selection](#)

Many of these methods are used in [Examining Values in the Example](#).

Selecting the Workspace

Variables assigned through the Command Window and created using scripts are considered to be in the base workspace. Variables created in a function belong to their own function workspace. To examine a variable, you must first select its workspace. When you run a program, the current workspace is shown in the **Stack** field. To examine values that are part of another workspace for a currently running function or for the base workspace, first select that workspace from the list in the **Stack** field.

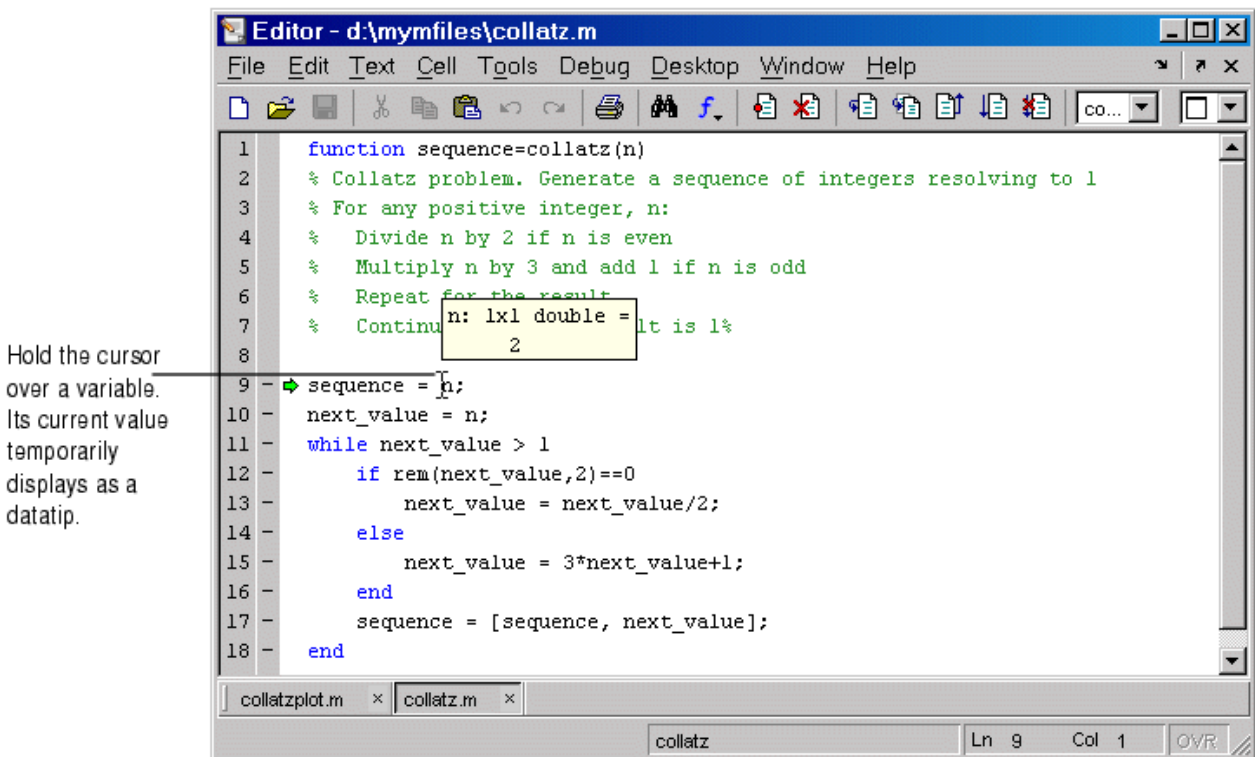
If you use debugging functions from the Command Window, use [dbstack](#) to display the call stack. Use [dbup](#) and [dbdown](#) to change to a different workspace. Use [who](#) or [whos](#) to list the variables in the current workspace.

Workspace in the Example. At line 10 of `collatzplot`, we stepped in, putting us at line 9 of `collatz`. The **Stack** shows that `collatz` is the current workspace.

Viewing Values as Datatips in the Editor/Debugger

In the Editor/Debugger, position the cursor to the left of a variable on that line. Its current value appears--this is called a datatip, which is like a tooltip for data. The datatip stays in view until you move the cursor. If you have trouble getting the datatip to appear, click in the line and then move the cursor next to the variable.

Datatips in the Example. Position the cursor over `n` in line 9 of `collatz`. The datatip shows that `n = 2`, as expected.



Viewing Values in the Command Window

You can examine values while in debug mode at the `K>>` prompt. To see the variables currently in the workspace, use [who](#). Type a variable name in the Command Window and MATLAB displays its current value. For the example, to see the value of `n`, type

```
n
```

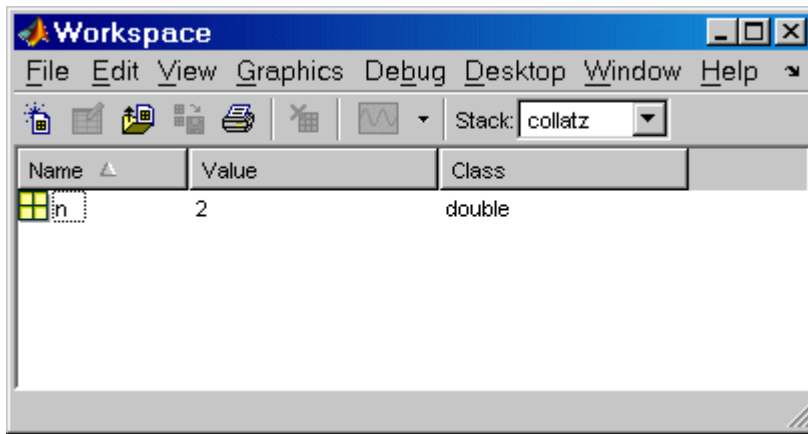
MATLAB returns the expected result

```
n = 2
```

and displays the debug prompt, `K>>`.

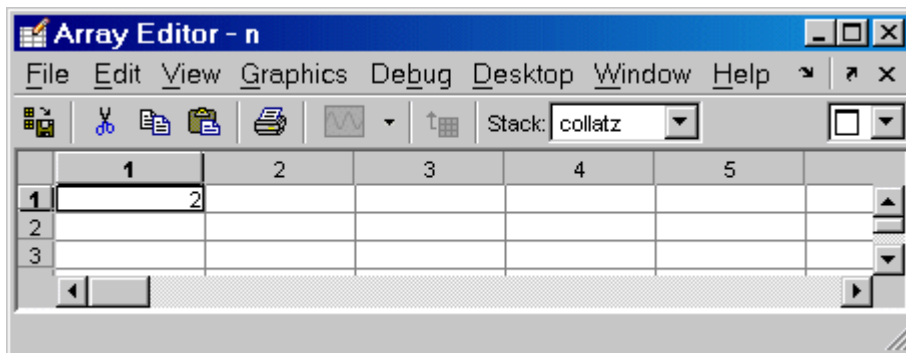
Viewing Values in the Workspace Browser and Array Editor

You can view the value of variables in the **Value** column of the Workspace browser. The Workspace browser displays all variables in the current workspace. Use the **Stack** in the Workspace browser to change to another workspace and view its variables.



The **Value** column does not show all details for all variables. To see details, double-click a variable in the Workspace browser. The Array Editor opens, displaying the content for that variable. You can open the Array Editor directly for a variable using [openvar](#).

To see the value of `n` in the Array Editor for the example, type
`openvar n`
 and the Array Editor opens, showing that `n = 2` as expected.



Evaluating a Selection

Select a variable or equation in an M-file in the Editor/Debugger. Right-click and select **Evaluate Selection** from the context menu. MATLAB displays the value of the variable or equation in the Command Window. You cannot evaluate a selection while MATLAB is busy, for example, running an M-file.

Examining Values in the Example

Step from line 9 through line 13 in `collatz`. Step again, and the pause indicator jumps to line 17, just after the `if` loop, as expected. Step again, to line 18, check the value of `sequence` in line 17 and see that the array is `2 1` as expected for `n = 2`. Step again, which moves the pause indicator from line 18 to line 11. At line 11, step again. Because `next_value` is now 1, the `while` loop ends. The pause indicator is at line 11 and appears as a green down arrow . This indicates that processing in the called function is complete and program control will return to the calling program. Step again from line 11 in `collatz` and execution is now paused at line 10 in `collatzplot`.

Note that instead of stepping through `collatz`, the called function, as was just done in this example, you can step out from a called function back to the calling function, which automatically runs the rest of the called function and returns to the next line in the calling function. To step out, use the Step Out button or type `dbstep out` in the Command Window.

In `collatzplot`, step again to advance to line 11, then line 12. The variable `seq_length` in line 11 is a vector with the elements

1 2

which is correct.

Finally, step again to advance to line 13. Examining the values in line 12, $N = 2$ as expected, but the second variable, `plot_seq`, has two values, where only one value is expected. While the value for `plot_seq` is as expected

2 1

it is the incorrect variable for plotting. Instead, `seq_length(N)` should be plotted.

Correcting Problems and Ending Debugging

These are some of the ways to correct problems and end the debugging session:

- [Changing Values and Checking Results](#)
- [Ending Debugging](#)
- [Disabling and Clearing Breakpoints](#)
- [Correcting an M-File](#)
- [Completing the Example](#)
- [Running Sections in M-Files That Have Unsaved Changes](#)

Many of these features are used in [Completing the Example](#).

Changing Values and Checking Results

While debugging, you can change the value of a variable in the current workspace to see if the new value produces expected results. While the program is paused, assign a new value to the variable in the Command Window, Workspace browser, or Array Editor. Then continue running or stepping through the program. If the new value does not produce the expected results, the program has a different or another problem.

Ending Debugging

After identifying a problem, end the debugging session. You must end a debugging session if you want to change and save an M-file to correct a problem, or if you want to run other functions in MATLAB.

Note It is best to quit debug mode before editing an M-file. If you edit an M-file while in debug mode, you can get unexpected results when you run the file. If you do edit an M-file while in debug mode, breakpoints turn gray, indicating that results might not be reliable. See [Valid \(Red\) and Invalid \(Gray\) Breakpoints](#) for details.

To end debugging, click the Exit Debug Mode button , or select **Exit Debug Mode** from the **Debug** menu.

You can instead use the function `dbquit` to end debugging.



After quitting debugging, the pause indicators in the Editor/Debugger display no longer appear, and the normal prompt `>>` appears in the Command Window instead of the debugging prompt, `K>>`. You can no longer access the call stack.

Disabling and Clearing Breakpoints

Disable a breakpoint to temporarily ignore it. Clear a breakpoint to remove it.

Disabling and Enabling Breakpoints. You can disable selected breakpoints so the program temporarily ignores them and runs uninterrupted, for example, after you think you identified and corrected a problem. This is especially useful for conditional breakpoints--see [Conditional Breakpoints](#).

To disable a breakpoint, right-click the breakpoint icon and select **Disable Breakpoint** from the context menu, or click anywhere in a line and select **Enable/Disable Breakpoint** from the **Debug** or context menu. You can also disable a conditional breakpoint by clicking the breakpoint icon. This puts an X through the breakpoint icon as shown here.

Disabled breakpoint  10  `plot_seq = collatz(N);`

After disabling a breakpoint, you can enable it to make it active again, or clear it. To enable it, right-click the breakpoint icon and select **Enable Breakpoint** from the context menu, or click anywhere in a line and select **Enable/Disable Breakpoint** from the **Breakpoints** or context menu. The X no longer appears on the breakpoint icon and program execution will pause at that line.

When you run `dbstatus`, the resulting message for a disabled breakpoint is
Breakpoint on line 10 has conditional expression 'false'.

Clearing (Removing) Breakpoints. All breakpoints remain in a file until you clear (remove) them or until they are automatically cleared. Clear a breakpoint after determining that a line of code is not causing a problem.

To clear a breakpoint in the Editor/Debugger, click anywhere in a line and select **Set/Clear Breakpoint** from the **Debug** or context menu. The breakpoint for that line is cleared. Another way to clear a breakpoint is to click a standard breakpoint icon, or a disabled conditional breakpoint icon.

To clear all breakpoints in all files, select **Debug -> Clear Breakpoints in All Files**, or click its equivalent button  on the toolbar.

The function that clears breakpoints is `dbclear`. To clear all breakpoints, use `dbclear all`. For the example, clear all of the breakpoints in `collatzplot` by typing

```
dbclear all in collatzplot
```

Breakpoints are automatically cleared when you

- End the MATLAB session
- Clear the M-file using `clear name` or `clear all`

Note When `clear name` or `clear all` is in a statement in an M-file that you are debugging, it clears the breakpoints.

Correcting an M-File

To correct a problem in an M-file,

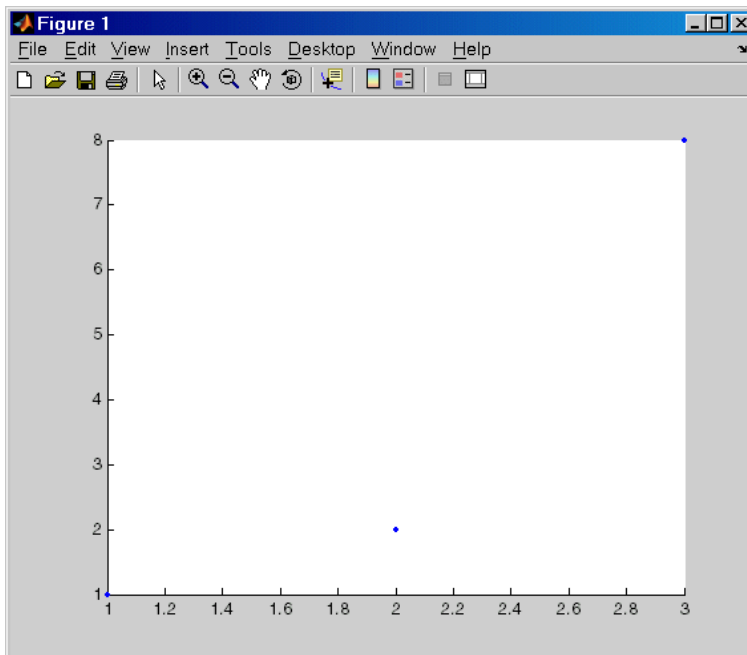
1. Quit debugging.
1. Do not make changes to an M-file while MATLAB is in debug mode. If you do edit an M-file while in debug mode, breakpoints turn gray, indicating that results might not be reliable. See [Valid \(Red\) and Invalid \(Gray\) Breakpoints](#) for details.
2. Make changes to the M-file.
3. Save the M-file.
4. Set, disable, or clear breakpoints, as appropriate.
5. Run the M-file again to be sure it produces the expected results.

Completing the Example

To correct the problem in the example, do the following:

1. End the debugging session. One way to do this is to select **Exit Debug Mode** from the **Debug** menu.
2. In `collatzplot.m` line 12, change the string `plot_seq` to `seq_length(N)` and save the file.
3. Clear the breakpoints in `collatzplot.m`. One way to do this is by typing
`dbclear all in collatzplot`
in the Command Window.
4. Run `collatzplot` for `m = 3` by typing
`collatzplot(3)`
in the Command Window.

- Verify the result. The figure shows that the length of the Collatz series is 1 when $n = 1$, 2 when $n = 2$, and 8 when $n = 3$, as expected.



- Test the function for a slightly larger value of m , such as 6, to be sure the results are still accurate. To make it easier to verify `collatzplot` for $m = 6$ as well as the results for `collatz`, add this line at the end of `collatz.m`

```
sequence
```

- which displays the series in the Command Window. The results for when $n = 6$ are

```
sequence =
```

```
6      3      10      5      16      8      4      2      1
```

Then run `collatzplot` for $m = 6$ by typing

```
collatzplot(6)
```

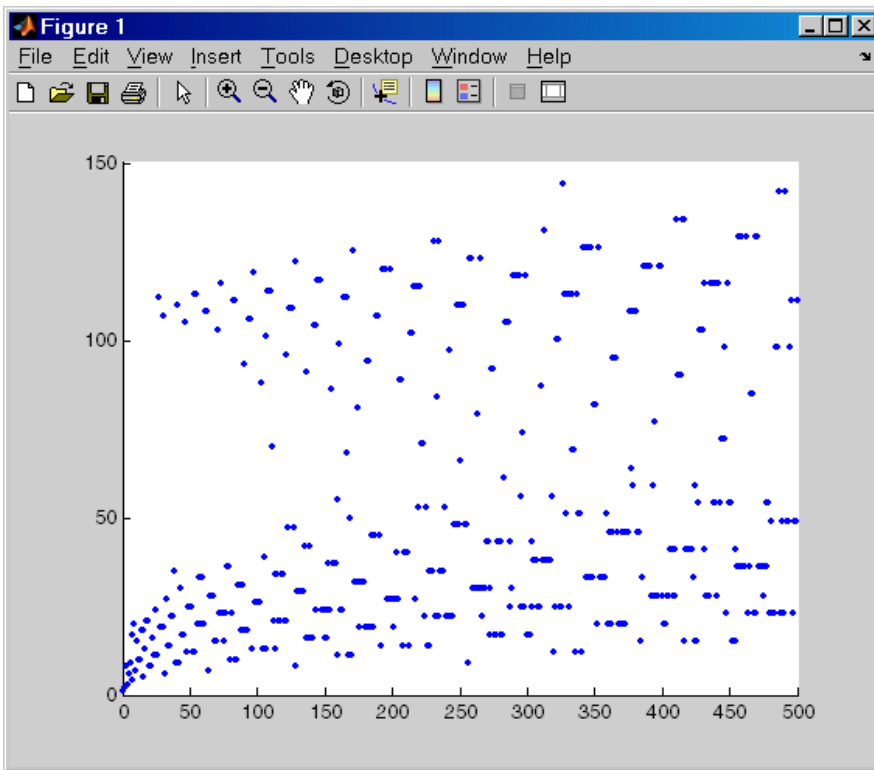
- To make debugging easier, you ran `collatzplot` for a small value of m . Now that you know it works correctly, run `collatzplot` for a larger value to produce more interesting results. Before doing so, you might want to suppress output for the line you just added in step 6, line 19 of `collatz.m`, by adding a semicolon to the end of the line so it appears as

```
sequence;
```

- Then run

```
collatzplot(500)
```

The following figure shows the lengths of the Collatz series for $n = 1$ through $n = 500$.



Running Sections in M-Files That Have Unsaved Changes

It is a good practice to make changes to an M-file after you quit debugging, and to save the changes and then run the file. Otherwise, you might get unexpected results. But there are situations where you might want to experiment during debugging, to make a change to a part of the file that has not yet run, and then run the remainder of the file without saving the change.

To do this, while stopped at a breakpoint, make a change to a part of the file that has not yet run. Breakpoints will turn gray, indicating they are invalid. Then select all of the code after the breakpoint, right-click, and choose **Evaluate Selection** from the context menu. You can also use cell mode to do this.

Conditional Breakpoints

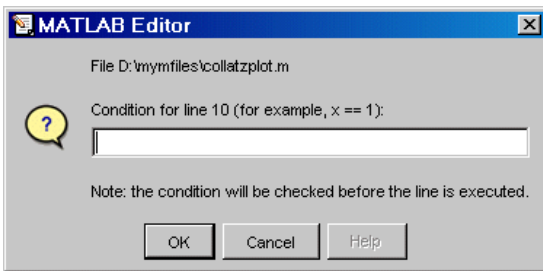
Set conditional breakpoints to cause MATLAB to stop at a specified line in a file only when the specified condition is met. One particularly good use for conditional breakpoints is when you want to examine results after a certain number of iterations in a loop. For example, set a breakpoint at line 10 in `collatzplot`, specifying that MATLAB should stop only if `N` is greater than or equal to 2. This section covers the following topics:

- [Setting Conditional Breakpoints](#)
- [Copying, Modifying, Disabling, and Clearing Conditional Breakpoints](#)
- [Function Alternative for Conditional Breakpoints](#)

Setting Conditional Breakpoints

To set a conditional breakpoint, follow these steps:

1. Click in the line where you want to set the conditional breakpoint. Then select **Set/Modify Conditional Breakpoint** from the **Debug** or context menu. If a standard breakpoint already exists at that line, use this same method to make it conditional.
1. The **MATLAB Editor** conditional breakpoint dialog box opens as shown in this example.



2. Type a condition in the dialog box, where a condition is any legal MATLAB expression that returns a logical scalar value. Click **OK**. As noted in the dialog box, the condition is evaluated before running the line. For the example, at line 10 in `collatzplot`, enter
 - o `N >= 2`
 - o
1. as the condition. A yellow breakpoint icon (indicating the breakpoint is conditional) appears in the breakpoint alley at that line.

Conditional breakpoint (yellow).

```

9 | for N = 1:m
10 |  plot_seq = collatz(N);
11 |  seq_length(N) = length(plot_seq);
12 |  line(N, plot_seq, 'Marker', '.', 'MarkerSize', 9, 'Color', 'blue')
13 |  drawnow
14 | end

```

When you run the file, MATLAB enters debug mode and pauses at the line only when the condition is met. In the `collatzplot` example, MATLAB runs through the `for` loop once and pauses on the second iteration at line 10 when `N` is 2. If you continue executing, MATLAB pauses again at line 10 on the third iteration when `N` is 3.

Copying, Modifying, Disabling, and Clearing Conditional Breakpoints

To copy a conditional breakpoint, right-click the icon in the breakpoint alley and select **Copy** from the context menu. Then right-click in the breakpoint alley at the line where you want to paste the conditional breakpoint and select **Paste** from the context menu.


Modify the condition for the breakpoint in the current line by selecting **Set/Modify Conditional Breakpoint** from the **Debug** or context menu.

Click a conditional breakpoint icon to disable it. Click a disabled conditional breakpoint to clear it.

Function Alternative for Conditional Breakpoints

Use the `dbstop` function with appropriate arguments to set conditional breakpoints from the Command Window, and use `dbclear` to clear them. Use `dbstatus` to view the breakpoints currently set, including any conditions, which are listed in the `expression` field. If no condition exists, the value in the `expression` field is `[]` (empty). For details, see the function reference pages: [dbstop](#), [dbclear](#), and [dbstatus](#).

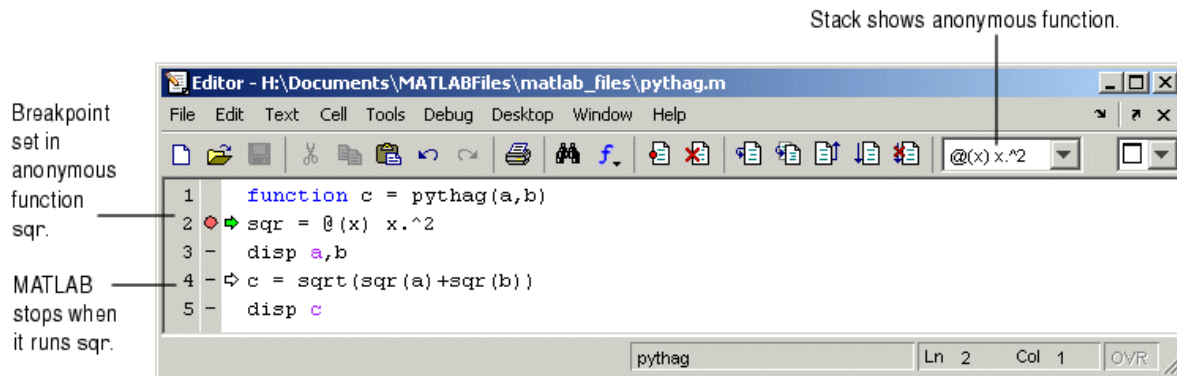
Breakpoints in Anonymous Functions

There can be multiple breakpoints in an M-file line that contains anonymous functions. There can be a breakpoint for the line itself (MATLAB stops at the start of the line), as well as a breakpoint for each anonymous function in that line. When you add a breakpoint to a line containing an anonymous function, the Editor/Debugger asks exactly where in the line you want to add the breakpoint. If there is more than one breakpoint in a line, the breakpoint icon is blue .

When there are multiple breakpoints set on a line, the icon is always blue, regardless of the status of any of the breakpoints on the line. Position the mouse on the icon and a tooltip displays information about all breakpoints in that line.

To perform a breakpoint action for a line that can contain multiple breakpoints, such as **Clear Breakpoint**, right-click in the breakpoint alley at that line and select the action. MATLAB prompts you to specify the exact breakpoint on which to act in that line.

When you set a breakpoint in an anonymous function, MATLAB stops when the anonymous function is called. The following illustration shows the Editor/Debugger when you set a breakpoint in the anonymous function `sqr` in line 2, and then run the file. MATLAB stops when it runs `sqr` in line 4. After you continue execution, MATLAB stops again when it runs `sqr` the second time in line 4. Note that the **Stack** display shows the anonymous function.



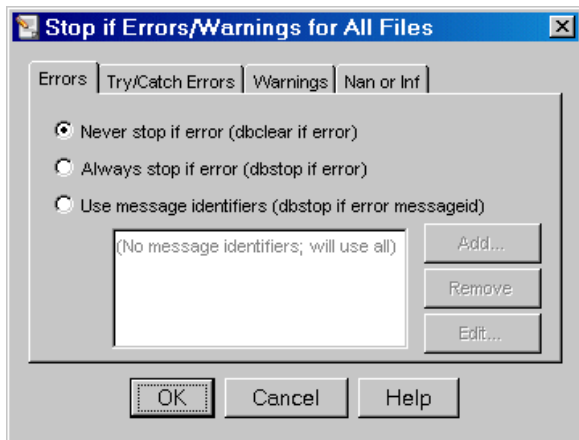
Error Breakpoints

Set error breakpoints to stop program execution and enter debug mode when MATLAB encounters a problem. Unlike standard and conditional breakpoints, you do not set these breakpoints at a specific line in a specific file. Rather, once set, MATLAB stops at any line in any file when the error condition specified via the error breakpoint occurs. MATLAB then enters debug mode and opens the file containing the error, with the pause indicator at the line containing the error. Files open only when you select **Debug -> Open M-Files When Debugging**. Error breakpoints remain in effect until you clear them or until you end the MATLAB session. You can set error breakpoints from the **Debug** menu in any desktop tool. This section covers the following topics:

- [Setting Error Breakpoints](#)
- [Error Breakpoint Types and Options](#) ([Errors](#), [Try/Catch Errors](#), [Warnings](#), [NaN or Inf](#), [Use Message Identifiers](#))
- [Function Alternative for Error Breakpoints](#)

Setting Error Breakpoints

To set error breakpoints, select **Debug -> Stop if Errors/Warnings**. In the resulting **Stop if Errors/Warnings for All Files** dialog box, specify error breakpoints on all appropriate tabs and click **OK**. To clear error breakpoints, select the **Never stop if ...** option for all appropriate tabs and click **OK**.



For example, to pause execution when a warning occurs, select the **Warnings** tab, and from it select **Always stop if warning**, then click **OK**. When you run an M-file and MATLAB produces a warning, execution pauses, MATLAB enters debug mode, and the file opens in the Editor/Debugger at the line that produced the warning. To remove the warning breakpoint, select **Never stop if warning** in the **Warnings** tab and click **OK**.

Error Breakpoint Types and Options

The four basic types of error breakpoints you can set are **Errors**, **Try/Catch Errors**, **Warnings**, and **NaN or Inf**. Select the **Always stop if ...** option for each tab to set that type of breakpoint. Select the **Use message identifiers ...** option to limit each type of error breakpoint (except Nan or Inf) so that execution stops only for specific errors.

Errors. When an error occurs, execution stops, unless the error is in a `try...catch` block. MATLAB enters debug mode and opens the M-file to the line that produced the error. You cannot resume execution.

Try/Catch Errors. When an error occurs in a `try...catch` block, execution pauses. MATLAB enters debug mode and opens the M-file to the line that produced the error. You can resume execution or use debugging features.

Warnings. When a warning is produced, MATLAB pauses, enters debug mode, and opens the M-file, paused at the line that produced the warning. You can resume execution or use debugging features.

NaN or Inf. When MATLAB encounters a NaN (not-a-number) or Inf (infinite) value, it pauses, enters debug mode, and opens the M-file, paused at the line that encountered the value. You can resume execution or use debugging features.

Use Message Identifiers. Execution stops only when MATLAB encounters one of the specified errors. This option is not available for the **Nan or Inf** type of error breakpoint. To use this feature:

1. Select the **Errors**, **Try/Catch Errors**, or **Warnings** tab.
2. Select the **Use Message Identifiers** option.
3. Click the **Add** button.
4. In the resulting **Add Message Identifier** dialog box, supply the message identifier of the specific error you want to stop for, where the identifier is of the form `component :message`, and click **OK**.
5. The message identifier you added appears in the **Stop If Errors/Warnings for All Files** dialog box, where you click **OK**.

You can add multiple message identifiers, and edit or remove them.

One way to obtain an error message identifier generated by a MATLAB function for example, is to produce the error, and then run the `lasterror` function. MATLAB returns the error message and identifier. Copy the identifier from the Command Window output and paste it into the **Add Message Identifier** dialog box. An example of an error message identifier is `MATLAB:UndefinedFunction`. Similarly, to obtain a warning message identifier, produce the warning and then run `[m,id] = lastwarn`; MATLAB returns the last warning identifier to `id`. An example of a warning message identifier is `MATLAB:divideByZero`.

Function Alternative for Error Breakpoints

The function equivalent for each option appears in the **Stop if Errors/Warnings for All Files** dialog box. For example, the function equivalent for **Always stop if error** is `dbstop if error`. Use the [dbstop](#) function with appropriate arguments to set error breakpoints from the Command Window, and use [dbclear](#) to clear them. Use [dbstatus](#) to view the error breakpoints currently set. Error breakpoints are listed in the `cond` field and message identifiers for breakpoints are listed in the `identifier` field of the `dbstatus` output.

Rapid Code Iteration Using Cells

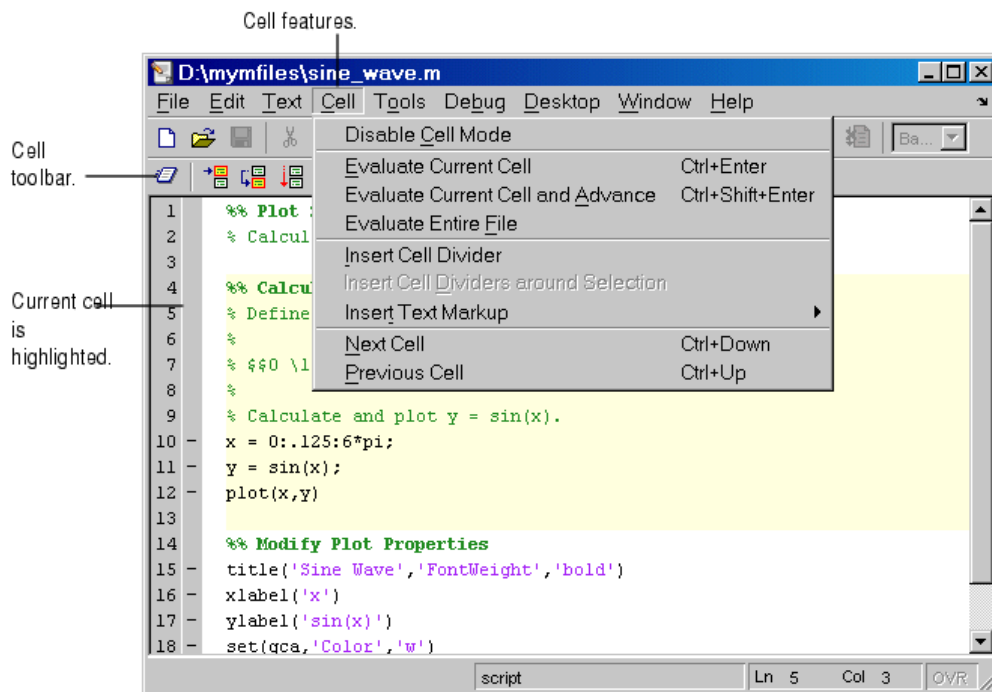
When working with MATLAB, you often experiment with your code--modifying it, testing it, and updating it--until you have an M-file that does what you want. Use the cell features in the MATLAB Editor/Debugger to make the experimental phase of your work with M-file scripts easier. You can also use cell features with function M-files, but there are some restrictions--see [Using Cells in Function M-Files](#).

[watch the Rapid Code Iteration Using Cells video demo](#)


The overall structure of many M-file scripts seems to naturally consist of multiple sections. Especially for larger files, you often focus efforts on a single section at a time, refining the code in just that section. To facilitate this process, use M-file *cells*, where a cell is a defined section of code.

This is the overall process of using cells for rapid code iteration:

1. In the MATLAB Editor/Debugger, enable cell mode. Select **Cell -> Enable Cell Mode**. Items in the **Cell** menu become selectable and the cell toolbar appears.
2. Define the boundaries of the cells in an M-file script using cell features. Cells are denoted by a specialized comment syntax. For details, see [Defining Cells](#).
3. Once you define the cells, use cell features to navigate quickly from cell to cell in your file, evaluate the code in a cell in the base workspace, and view the results. To facilitate experimentation, use cell features to modify values in cells and then reevaluate them, to see how different values impact the result. For details, see [Navigating and Evaluating with Cells](#).
4. Cells are also useful if you want to share your results by publishing your work in a presentation format, such as an HTML document. For details, see [Publishing to HTML, XML, LaTeX, Word, and PowerPoint Using Cells](#).



Defining Cells

Cell features operate on contiguous lines of code you want to evaluate as a whole in an M-file script, called cells. To define a cell, first be sure that cell mode is enabled (see step 1). Position the cursor just before the line you want to start the cell and then select **Cell -> Insert Cell Divider** or click the Insert Cell Divider button . After the cursor, MATLAB inserts a line containing two percent signs (%%), which is the "start new cell" indicator to MATLAB. A cell consists of the line starting with %% and the lines that follow, up to the start of the next cell, which is identified by %% at the start of a line.

You can also define a cell by entering two percent signs (%%) at the start of the line where you want to begin the new cell. Alternatively, select the lines of code to be in the cell and then select **Cell -> Insert Cell Dividers Around Selection**.

You can define a cell at the start of a new empty file, enter code for the cell, define the start of the next cell, enter its code, and so on. Redefine cells by defining new cells, removing existing cells, and moving lines of code.

MATLAB will not execute the code in lines beginning with %, so be sure to put any executable code for the cell on the following line. For [program control statements](#), such as `if ... end`, a cell must contain both the opening and closing statements, that is, it must contain both the `if` and the `end` statements.

Note that the first cell in a file does not have to begin with %. MATLAB automatically understands any lines above the first %% line to be a cell. If there are no cell dividers in an M-file, MATLAB understands the entire file to be a single cell.

Cell Titles and Highlighting

After the %, type a space followed by a description of the cell. The Editor/Debugger emphasizes the special meaning of the start of a cell by making any text following the percent signs and space bold. The text on the %% line is called the *cell title* (like a section title). Including cell titles is optional, however, they improve readability of the file and are used for cell publishing features.

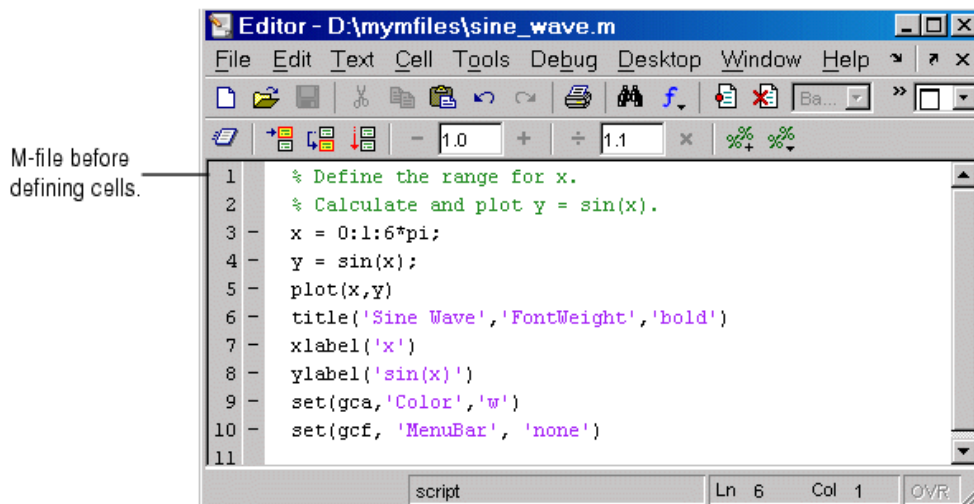
When the cursor is positioned in any line within a cell, the Editor/Debugger highlights the entire cell with a yellow background. This identifies it as the *current cell*. For example, it is used when you select the **Evaluate Current Cell** option on the **Cell** menu.

If you want cell titles to appear in plain rather than bold text, or if you do not want yellow highlighting for the current cell, change these preferences. Select **File -> Preferences -> Editor/Debugger -> Display** and change the appropriate **Cell display options**.

Example--Define Cells

This examples defines two cells for a simple M-file called `sine_wave`, shown in the following code and figure. The first cell creates the basic results, while the second label the plot. The two cells in this example allow you to experiment with the plot of the data first, and then when that is final, change the plot properties to affect the style of presentation.

```
% Define the range for x.
% Calculate and plot y = sin(x).
x = 0:1:6*pi;
y = sin(x);
plot(x,y)
title('Sine Wave','FontWeight','bold')
xlabel('x')
ylabel('sin(x)')
set(gca,'Color','w')
set(gcf, 'MenuBar', 'none')
```



1. Select **Cell -> Enable Cell Mode**, if it is not already enabled.
2. Position the cursor at the start of the first line. Select **Cell -> Insert Cell Divider**. The Editor/Debugger inserts %% as the first line and moves the rest of the file down one line. All lines are highlighted in yellow, indicating that the entire file is a single cell.
3. Enter a cell title following the %% . Type a space first, followed by the description.
Calculate and Plot Sine Wave
4. Position the cursor at the start of line 7, `title...` Select **Cell -> Insert Cell Divider**. The Editor/Debugger inserts a line containing only %% at line 7 and moves the remaining lines down by one line. Lines 7 through 12 are highlighted in yellow, indicating they comprise the current cell.
5. Enter a cell title for the new cell. On line 7, type a space after the %% , followed by the description
Modify Plot Properties
Save the file. The file appears as shown in this figure.

M-file after defining cells.

```

1  %% Calculate and Plot Sine Wave
2  % Define the range for x.
3  % Calculate and plot y = sin(x).
4  x = 0:1:6*pi;
5  y = sin(x);
6  plot(x,y)
7  %% Modify Plot Properties
8  title('Sine Wave','FontWeight','bold')
9  xlabel('x')
10 ylabel('sin(x)')
11 set(gca,'Color','w')
12 set(gcf, 'MenuBar', 'none')
13

```

Removing Cells

To remove a cell, delete one of the percent signs (%) from the line that starts the cell. This changes the line from a cell to a standard comment and merges the cell with the preceding cell. You can also just delete the entire line that contains the %%.

Navigating and Evaluating with Cells

While you develop an M-file, you can use these Editor/Debugger cell features:

- [Navigating Among Cells in an M-File](#)
- [Evaluating Cells in an M-File](#)
- [Modifying Values in a Cell](#)
- [Example--Evaluate Cells](#)

Navigating Among Cells in an M-File

To move to the next cell, select **Cell -> Next Cell**. To move to the previous cell, select **Cell -> Previous Cell**. To move to a specific cell, click the Show Cell Titles button  and from it, select the cell title to which you want to move. You can also go to cells by selecting **Edit -> Go To**.


Evaluating Cells in an M-File


To evaluate the code in a cell, use the **Cell** menu evaluation items or buttons in the cell toolbar. When you evaluate a cell, the results display in the Command Window, figure window, or otherwise, depending on the code evaluated.

The cell evaluation features run the code currently shown in the Editor/Debugger, even if the file contains unsaved changes. The file does not have to be on the search path. To evaluate a cell, it must contain all the values it requires, or the values must already exist in the MATLAB workspace.

Note While you can set breakpoints and debug a file containing cells, when you evaluate a file from the **Cell** menu or Cell Toolbar, breakpoints are ignored. To run the file and stop at breakpoints, use **Run/Continue** in the **Debug** menu. This means you cannot debug while running a single cell.

Evaluate Current Cell. Select **Cell -> Evaluate Current Cell** or click the Evaluate Cell button  to run the code in the current cell.

Evaluate and Advance. Select **Cell -> Evaluate Current Cell and Advance** or click the Evaluate Cell and Advance button  to run the code in the current cell and move to the next cell.

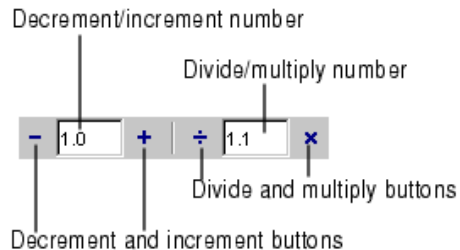
Evaluate File. Select **Cell -> Evaluate Entire File** or click the Evaluate Entire File button  to run all of the code in the file.

Note A beep means there is an error. See the Command Window for the error message.

Modifying Values in a Cell

You can use cell features to modify numbers in a cell, which also automatically reevaluates the cell.

To modify a number in a cell, select the number (or place the cursor near it) and use the value modification tool in the cell toolbar. Using this tool, you can specify a number and press the appropriate math operator to add (increment), subtract (decrement), multiply, or divide the number. The cell then automatically reevaluates.



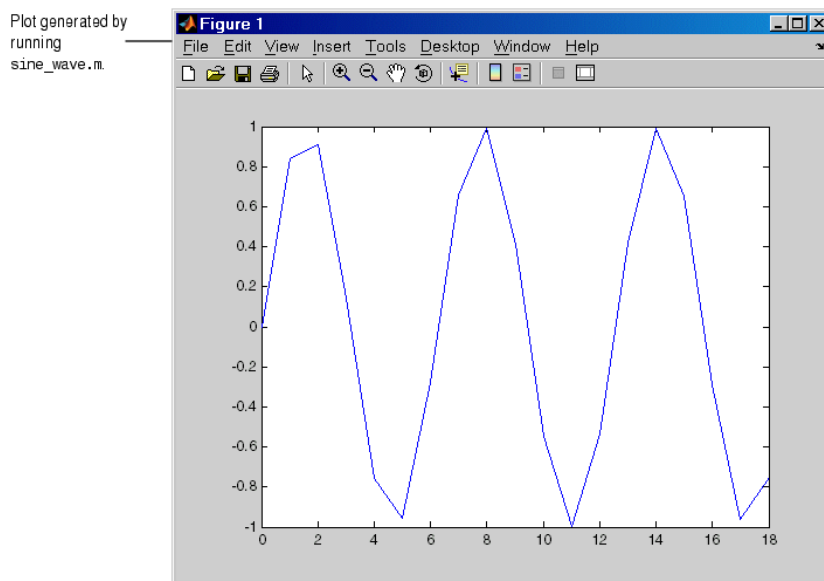
You can use the numeric keypad operator keys (-, +, /, and *) instead of the operator buttons on the toolbar.

Note MATLAB does not automatically save changes you make to values using the cell toolbar. To save changes, select **File -> Save**.

Example--Evaluate Cells

In this example, modify the values for `x` in `sine_wave.m`:

1. Run the first cell in `sine_wav.m`. Click somewhere in the first cell, that is, between lines 1 and 6. Select **Cell -> Evaluate Current Cell**. The following figure appears.



2. Assume you want to produce a smoother curve. Use more values for `x` in `0:1:6*pi`. Position the cursor in line 4, next to the 1. In the cell toolbar, change the 1.1 default multiply/divide by value to 2. Click the Divide button \div .

Line 4 becomes


```
4 - x = 0:0.5:6*pi;
```

and the length of x doubles. The plot automatically updates. The curve still has some rough edges.

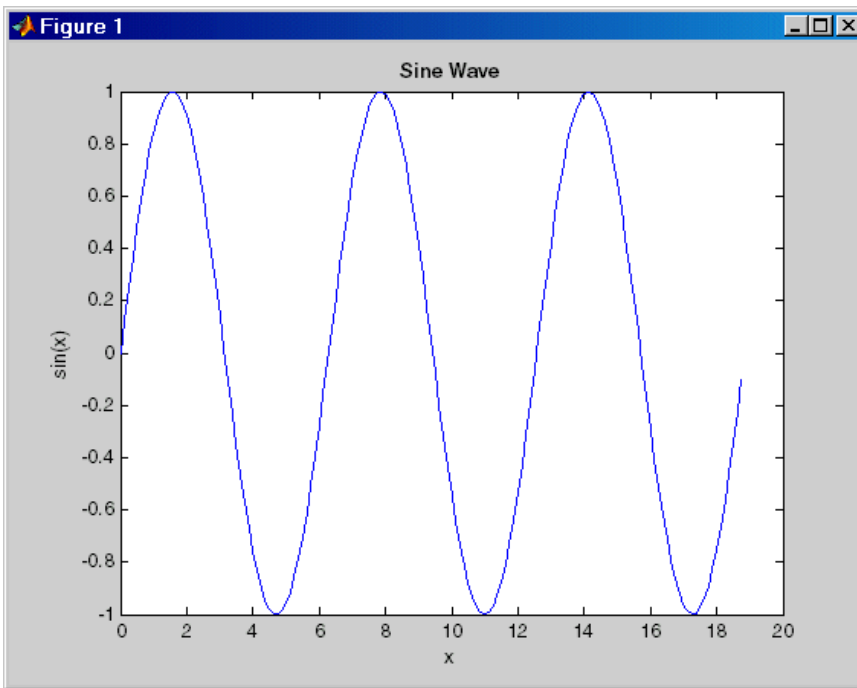
3. To add more values for x , click the divide button three more times. Line 4 becomes

```
4 - x = 0:0.0625:6*pi;
```

The curve is smooth, but because there are more values, processing time is slower. It would be better to find a smaller x that still produces a smooth curve.

4. In the cell toolbar, click the multiply button once. The increment for x as shown in line 4 changes from 0.0625 to 0.125.
The resulting curve is still smooth.
5. Save these changes. Select **File -> Save**.
6. You do not need to evaluate the entire file to modify the plot properties. Instead, evaluate the second cell, that is, lines 7 through 12. You can use the shortcut **Ctrl+Enter** to evaluate the current cell. (The shortcut appears with the menu item, **Cell -> Evaluate Current Cell**).

MATLAB updates the figure.



Using Cells in Function M-Files

You can define and evaluate cells in function M-files as long as the variables referred to in the cell are in your workspace. For example, this can be useful during debugging. If execution is stopped at a breakpoint, you can define cells and execute them without saving the file. If you are not debugging, add the necessary variables to the base workspace and then execute the cells. Cell publishing is not supported for function M-files.