1207 Techniques for Debugging MATLAB M-files

Revision: 1.0 *Last Date Modified: 28-April-2003*

Introduction

Bug Types

Graphical Debugging Using the MATLAB Editor/Debugger

Debugging from the MATLAB Command Prompt

Common Errors and Warnings

Errors

Introduction

Section 1: Who Should Use This Technical Note?

This Tech Note is intended for anyone writing code in MATLAB who would like to learn how to use MATLABs tools to find and eliminate bugs within their programs. Although it is not required, before reading this Tech Note, it is recommended that you review the Debugging M-files section in Chapter 7 of the Using MATLAB (PDF) manual, which briefly discusses the MATLAB Editor's debugging capabilities.

Section 2: What Does This Tech Note Cover?

This Tech Note describes how to detect syntax errors in your MATLAB functions and scripts. It includes clear descriptions and illustrations of the tools you can use to detect and correct errors, along with examples that demonstrate some of the more common errors MATLAB programmers encounter. The Common Errors and Warnings section of this Tech Note, beginning with Section 21, describes *some* of the more commonly encountered error and warning messages.

This Tech Note cannot, of course, describe all the possible error and warning messages that functions in MATLAB may return, due to the infinite number of code sequences you might program. However, the techniques you will use in debugging the errors and warnings in this section will provide you with a foundation that you can use to help determine the source of errors not listed in this Tech Note.

This Tech Note will *not* cover debugging MEX-files, Simulink models, or Simulink S-functions; for information about troubleshooting these types of files please see one of the following:

Technical Note 1605: The MEX-Files Guide

Technical Note 1819: How Do I Debug C MEX S-Functions?

Section 3: Definitions of Terms

- A bug is a flaw in a program. The term became associated with computers as a result of a moth getting stuck in an early computer at Harvard University. The story is described in more detail here.
- The MATLAB Editor/Debugger is the window that appears by default when you open or create an M-file on a platform on which Java is enabled. Two ways to open the Editor/Debugger are to type
  ```
  edit
  ```
  at the MATLAB command prompt, or to select **New** from the **File** menu and choose to create an M-file.
- Debug mode is a special mode where the debugging commands and Editor/Debugger options can be used to locate bugs in a program. It is indicated by the MATLAB prompt changing from the normal >> to the keyboard prompt K>>, the Editor/Debugger opening (if it is not already open and the appropriate preference set), and an arrow appearing at the current line in the M-file being executed. You can control if the Editor opens when MATLAB enters debug mode by going to the **File -> Preferences** menu and selecting or deselecting **Automatically open files when debugging** in the main Editor/Debugger preferences.
- A workspace is the section of memory where the variables for a script or function are stored. The main workspace, used by the commands you type at the MATLAB command prompt and by script files, exists continuously while MATLAB is open. Function workspaces are created when the function is first called and are destroyed when the function exits.

Section 4: Conventions

Within this Technical Note, example MATLAB code is provided. This code appears in the following type style:

```
cos(pi)
```

If you would like to evaluate the code within MATLAB for yourself, you can either copy and paste the code directly into MATLAB, or, if you are using the MATLAB Help Browser to view this Tech Note, you can simply highlight the code you are investigating, right-click, and select **Evaluate selection**.

Words in all caps and in monospace font are the names of MATLAB functions. Often, these works will also be links to the corresponding MATLAB function documentation. For example, this sentence refers to the MATLAB function `SIN`, which computes the sine of an angle.

Section 5: For What Versions of MATLAB Are These Techniques Valid?

The majority of these techniques should work for MATLAB 5.3 (R11) and higher, although the syntax of certain commands may have changed between the time it was released and the present. The screen shots shown in this Tech Note are taken from the Windows version of MATLAB 6.5 (R13).

Bug Types

Section 6: What Are the Three Main Types of Bugs?

The three major types of bugs are typographic errors (typos), syntax errors, and algorithmic errors. This Tech Note will not address algorithmic errors in detail, due to the individuality of each algorithm. The definitions of each type of bug are as follows:

- A *typo*, or *typographic error*, is a simple typing error. If it occurs while you are typing a function name, it can be easy to find -- but if you mistype a variable name, it can lead to unexpected results that can be extremely difficult to track down and eliminate. For example, typing

```
cod(pi)
```

when you meant to type

```
cos(pi)
```

will return an error message that is easy to understand:

```
??? Undefined function or variable 'cod'.
```

However, if you type

```
x=cos(y);
```

when you meant to type

```
x=cos(t);
```

and `t` is a different size than `y`, you'll likely receive an error message about the size of the array when you try to manipulate it later in your code, long after the assignment statement has been executed. Tracing the error back to this line may be difficult if you have a significant amount of code between the assignment and the line where the error occurs.

- A *syntax error* occurs when the calling syntax you use for a function is incorrect, or when you provide the function with inputs that are the wrong shape, size, and/or type or are otherwise not valid for the function in question. For example, the typo in the previous bullet (typing `y` when you intended to type `t`) can lead to a syntax error (when you try to use the `x` variable in a later line of your code.) The error messages for these types of errors usually indicate the problem, but deciphering what they mean can sometimes be difficult, and tracking down where the problem originated can be even harder. This type of error message is what the techniques in this Tech Note are designed to catch and investigate, but the techniques also work to some extent for the other two main types of bugs. The Debugging M-Files section in Chapter 7 of the Using MATLAB (PDF) manual, which discusses using the Editor/Debugger and debugging M-files, groups typographical errors and syntax errors under the category of syntax errors.
- An *algorithmic error* occurs when the program executes perfectly, but the result you receive is not what you expect. For instance, if you wrote a program to add two numbers, passed it 2 and 3, and received 6 as a result (with no error or warning messages) that would be an algorithmic error. The main technique involved in debugging these types of bugs is to compute the answer you expect to the problem by some means other than MATLAB (or by using an example worked in a text on the subject) and stepping through the code using the Step button (which is described in the next section) or the `DBSTEP` function (described in Section 15) verifying that the algorithm computes the correct result at each stage of the calculation. This procedure can be streamlined by setting breakpoints after sections of the code and stopping the first time the answer at a breakpoint disagrees with the answer computed using the other method; then you can step through the code between that breakpoint and the previous one to locate the error precisely.

Graphical Debugging Using the MATLAB Editor/Debugger

Section 7: What Debugging Tools Are Available from the MATLAB Editor/Debugger?

As you might expect from the name, there are many tools in the MATLAB Editor/Debugger that you can use to isolate and correct errors. These tools include breakpoints, the program execution controls in the Debug menu, variable datatips, and the Editor context menu. For purposes of illustrating some of these tools and techniques, you should use the `why.m` file for editing purposes. To do this, type

```
edit why
```

at the MATLAB command prompt. The relevant section of the Editor window is reproduced in Figure 1:

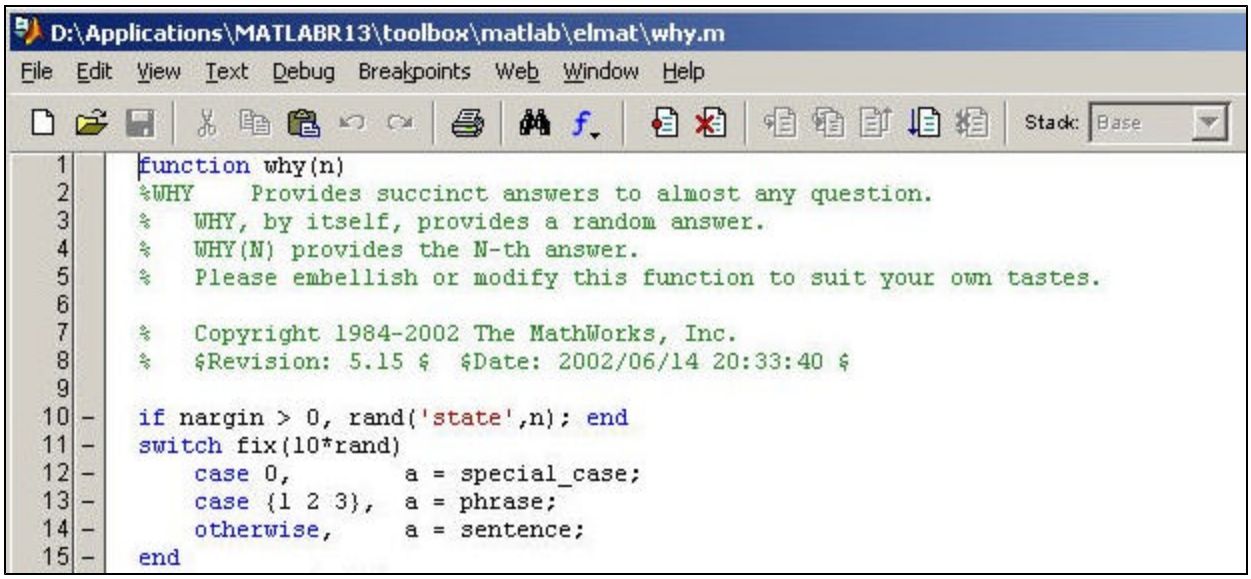

Figure 1: The main Editor window

Section 8: Breakpoint Menu and Buttons

Breakpoints stop the execution of your function and cause MATLAB to enter debug mode, so that you can verify that the function is doing what it should be doing. You can set breakpoints in the Editor in four different ways:

1. Clicking on the horizontal line next to a given line sets a breakpoint at that line. The column this line is located in is called the breakpoint alley. MATLAB will stop executing the function just before that line. In Figure 2, clicking on the line next to line 10 sets the breakpoint at that location. Now, if you type

   ```
   why
   ```

   at the MATLAB command prompt, execution will stop just before line 10 executes:

   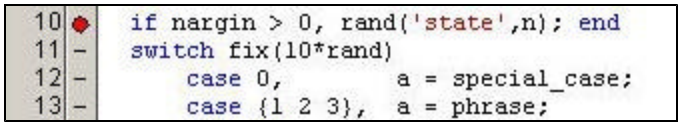

   Figure 2: Setting a breakpoint using the breakpoint alley

2. Alternatively, clicking on the first button shown in Figure 3 below (in the fourth group of buttons) when the cursor is on line 10 will accomplish the same task. The second button shown in Figure 3 will clear all breakpoints in the file:

   

   Figure 3: The set breakpoint and clear breakpoint buttons

3. Selecting the Breakpoints menu provides you with the opportunity to set and clear breakpoints and to choose certain conditions which, when satisfied, will stop execution immediately. These conditions are

discussed more in depth in the entry for DBSTOP in Section 14. Figure 4 shows the options the options available on the Breakpoints menu:

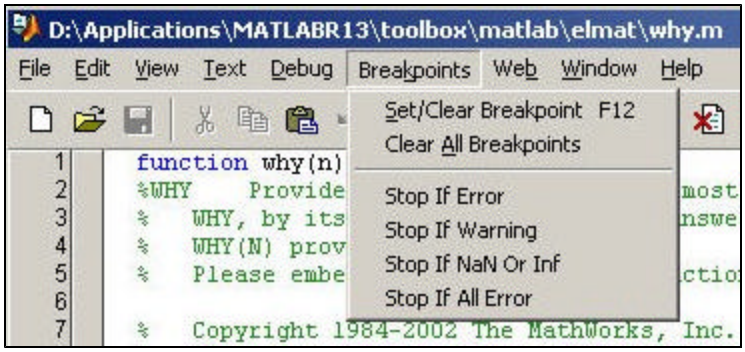

Figure 4: The Breakpoints menu

4. Right-click when the cursor is on the line on which you want to set a breakpoint. Select **Set/Clear Breakpoint** from the menu which appears:

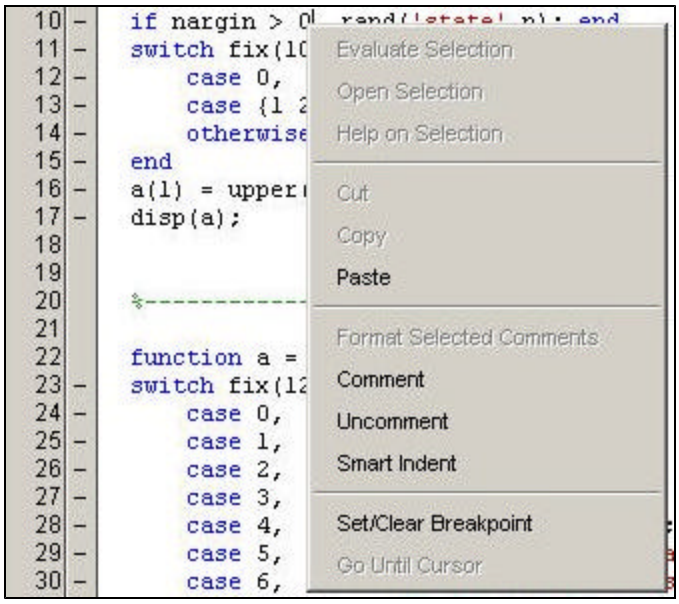

Figure 5: Setting a breakpoint with the context menu

Try using breakpoints now. Set a breakpoint at line 10, then run

```
why(37)
```

from the MATLAB command prompt. When MATLAB brings up the K>> prompt, type

```
n
```

and press **Enter**. At this point in the execution of the code, the variable n has the value 37. If you expected it to be something else, like 42, you would know that there is a bug before line 10 in the `why.m` function.

You may be wondering how to get MATLAB to return to its normal prompt and continue with its computations. The K>> prompt indicates that you're in debug mode; to exit debug mode, type one of the following three commands

```
return
```

or

```
dbcont
```

or

```
dbquit
```

at the MATLAB prompt. You can also exit debug mode using features of the MATLAB Editor; this is the topic of the next section. If you executed the call to WHY above and execute the RETURN function at this point, the result you see partially explains why MATLAB behaves the way it does.

The DBQUIT function causes MATLAB to exit debug mode and any scripts or functions that were running when you entered debug mode. The RETURN command also exits debug mode but does not exit the current script or function. However, any error messages that were interrupted by entering debug mode will appear. You can use the RETURN function inside an M-file to cause it to return to the calling function or to the command line (if it was called from the command line.) Like RETURN, DBCONT exits debug mode and proceeds with executing the current script or function, resuming at the next line to be executed. However, DBCONT cannot appear in a function outside of debug mode.

Section 9: Debug Menu and Buttons

Once you have reached debug mode (by reaching a breakpoint or by using either the KEYBOARD or DBSTOP functions, both described in in Section 14), you can control the execution of the program using the commands on the Debug menu and the debug buttons. In fact, you can even start the program from the Debug menu, wait for it to reach a breakpoint, and then proceed one command at a time, verifying the output as you go. To run the program from the menu, select **Run** from the Debug menu, or simply press F5.

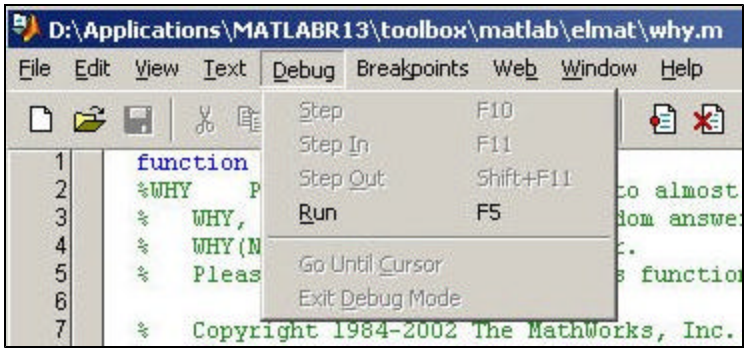

Figure 6: Running a program using the Debug menu

You can also press the **Run** button in the last set (the debug set) of buttons in the toolbar:

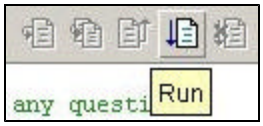

Figure 7: Running a program using the Run button

Once you have reached a breakpoint, you will enter debug mode. If there are no breakpoints in your function, the function will run to completion.

In debug mode, you have several options:

- **Step** This option will allow you to step through your function one line at a time, verifying that execution is proceeding smoothly. To step through, press F10 (for Windows machines) or F6 (for UNIX or Linux machines), or click on the first button in the debug set of buttons in the toolbar.

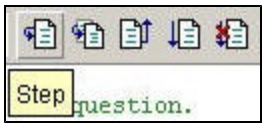

Figure 8: The Step button

- **Step In** This option will allow you to open the file containing the first function called on the line you are currently debugging and step through that function. This allows you to continue debugging without having to determine which file to open next and without having to set breakpoints in every function called by your program, every function called by those functions, etc. Note that Step In will ignore built-in functions, MEX-files, and P-code files without a corresponding M-file, as there are no M-files containing code for those functions. F11 (on Windows) or F7 (on UNIX/Linux) causes MATLAB to step in, as does the second button in the debug set.

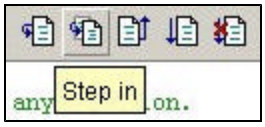

Figure 9: The Step In button

- **Step Out** This will allow you to return from a function you stepped in to its calling function without having to execute each of the remaining lines in the function individually. This is useful if you are absolutely certain that the last section of your code is correct and you only wanted to verify a certain section near the beginning. When you step out of a function, you will still be in debug mode and will be on the line of the calling function on which you called the function from which you stepped out.

  Shift-F11 (Windows) or Shift-F7 (UNIX/Linux), or the third button on the debug set of buttons causes you to step out.

Figure 10: The Step Out button

- **Continue** Once you are in debug mode, the **Run** button (as shown above) turns into the **Continue** button. Use this button if you want to continue executing code until the next time you reach a breakpoint, until you receive an error, or until the code is finished.

- **Go until cursor** If you are in debug mode and want to execute a section of code starting at the current point and proceeding until a certain point (without setting a breakpoint) you can position the cursor at the point where you want execution to end and select **Go until cursor** on the Debug menu. This will cause all the code between the location at which execution was last stopped and the current location of the cursor to be executed.

- **Exit Debug Mode** If you step through your code and find the error, you may simply want to stop execution so you can edit the function and correct the problem. To do this, press the last button in the Debug set of buttons to exit debug mode and execution of your program.

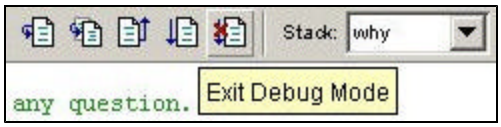

Figure 11: The Exit Debug Mode button

Section 10: Datatips for Variables

Another of the MATLAB Editor's features that makes debugging programs easier is the ability to display datatips containing the values of variables during execution of the program. These datatips allow you to verify that the calculations in your program return the result you expect and that other functions which call your function pass the correct input arguments.

Whether or not datatips appear is controlled by an option in the MATLAB preferences. In the Editor/Debugger Display preferences, shown in Figure 12, select **Enable datatips in edit mode** to enable datatips. Clear this checkbox to disable them.

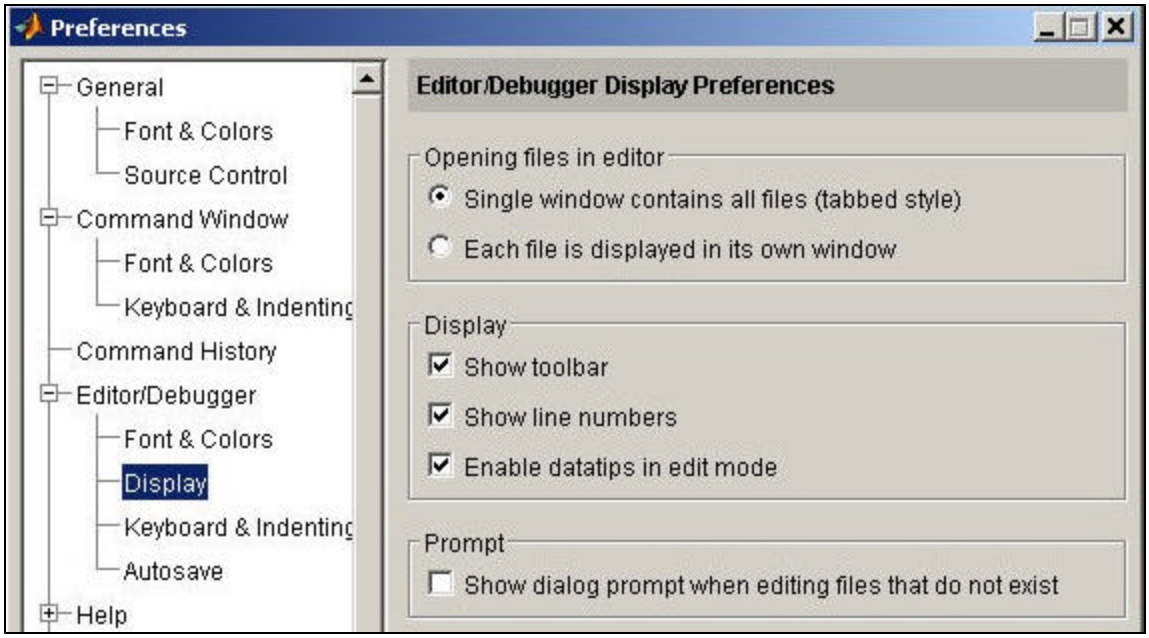

Figure 12: Enabling datatips using Preferences

The way to find variable values in the current function's workspace is to move the mouse over them; a datatip (like the ones labeling the buttons in the previous section) will appear. This datatip displays the value of the variable, or as much of its value as it can. For example, if you set a breakpoint on line 10 of `why.m` and call it by typing:

```
why(19)
```

and then move the mouse over the appearance of n in that line, you will see Figure 13:

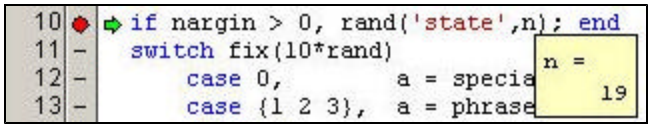

Figure 13: Verifying the value of a variable using a datatip

Note that, for clarity purposes, the cursor does not appear in the image above.

- **The Stack pull-down menu:** You can change from one function's workspace to a previous function's workspace using the **Stack** pull-down menu. This allows you to trace an error back to its source; for example, if one of the inputs to your function is incorrect, you can move to the workspace of the function that called your function and verify that the input was computed correctly.

  You can move to any workspace, starting with the base workspace and proceeding to the workspace of any function that has been called. Once you move to a different workspace, if it is associated with a function file, that function will be opened (if it is not already open) in the Editor/Debugger and you will be able to examine the values of variables in that new workspace. Figure 14 shows the **Stack** pull-down menu during a call to the why function; you can remain in why's workspace, or move up to the base workspace.

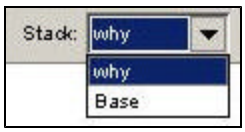

Figure 14: The Stack pull-down menu

Section 11: Evaluate Selection

If you have a section of code in your M-file that you want to execute individually, you should select that particular section, and then right-click and select the **Execute Selection** item in the menu that appears. This will cause MATLAB to execute that section in the current workspace with the current values for variables. This is useful in debug mode, as you can specify known values for certain variables that may be causing problems and execute the code after the problematic lines to test if they are in fact the cause of the problem. You can also evaluate the code using the F9 key or the **Evaluate Selection** option in the Text menu.

Section 12: Help on Selection

If you select the name of an M-file called by your script or function, and then right-click on it and select **Help on Selection** from the menu that appears, the documentation for that M-file will open in the Help browser. You can use this to verify the calling syntax for a function or to verify that it does what you think it does.

Debugging from the MATLAB Command Prompt

Section 13: What Debugging Tools Are Available at the MATLAB Command Prompt?

Many of the techniques discussed in Sections 7 through 12 of this Tech Note for use in the Editor/Debugger have their counterparts at the command prompt. This section describes how to make use of functions to debug programs from the command prompt.

You can download a quick reference sheet containing a brief description of each of the functions discussed in this section here.

Section 14: Setting, Clearing, and Querying Breakpoints

The `DBSTOP` and `DBCLEAR` functions serve to create and clear breakpoints, similar to the Breakpoints menu items discussed in Section 8. The `DBSTOP` function has seven different modes, each of which serves to set a different condition that MATLAB will check to determine if it needs to stop. The modes and a brief description of their behavior are as follows:

- **dbstop in [m-file]**
  This sets a breakpoint on the first executable line of the script or function [m-file]. The following command

  ```
  dbstop in why
  ```

  sets a breakpoint on the first line of `why.m` that can be executed. For `why.m`, this line is line 10.

- **dbstop in [m-file] at [line number]**
  This sets a breakpoint in the script or function [m-file] on line number [line number]. For instance

  ```
  dbstop in why at 10
  ```

  sets a breakpoint in `why.m` at line 10.

- **dbstop in [m-file] at [subfunction]**
  This sets a breakpoint on the first line of the subfunction named [subfunction] in the function [m-file]. For example

  ```
  dbstop in fminbnd at terminate
  ```

  sets a breakpoint on the first line of the `terminate` subfunction of the function `fminbnd.m`

- **dbstop if error**
  This syntax for the `DBSTOP` function is one of the most widely used debugging commands. When a file you execute encounters an error, normally the execution of the file stops and MATLAB displays the error message in the command window. However, if you execute

  ```
  dbstop if error
  ```

  before running the file, MATLAB will open that file in the Editor/Debugger (assuming you have set the preference on the main Editor/Debugger Preference window) to the line on which the error occurred and enter debugging mode instead. This will allow you to verify the status of the workspace at the time the error occurred, so you can verify that the computations you performed were the calculations you expected to perform.

  For example, if you modify line 10 of `why.m` to read

  ```
  if nargin > 0, rand('state',m);
  end
  ```

  instead of

  ```
  if nargin > 0, rand('state',n);
  end
  ```

  and then execute

  ```
  dbstop if error, why(5)
  ```

  MATLAB returns

  ```
  Undefined function or variable m
  ```

  and opens `why.m` (if it is not already open) and stops on line 10.

- **dbstop if all error**

This syntax for DBSTOP is similar to the syntax

```
dbstop if error
```

but it behaves differently in one specific case. When you execute code inside a TRY/CATCH block (which will be described in Section 18) and the code inside the TRY section of the block throws an error,

```
dbstop if error
```

does not cause MATLAB to enter debug mode, assuming that the CATCH section of the block will catch and handle the error. However,

```
dbstop if all error
```

does stop inside the TRY section of the block. This can be useful if you are writing a TRY/CATCH block to handle various types of errors and you want to test how a certain case in your TRY section will fail so you can CATCH it appropriately.

- **dbstop if warning**
  This syntax is similar to

  ```
  dbstop if error
  ```

  but causes MATLAB to stop when a warning message is thrown. Certain operations in MATLAB throw warnings (such as inverting a matrix that is almost singular) and continue processing the program. This DBSTOP syntax can detect those warnings and allow you (in the example above) to determine which matrix you are processing is close to singular.

- **dbstop if naninf or dbstop if infnan**
  These two syntaxes of DBSTOP perform the same function. They cause MATLAB to enter debug mode whenever it detects an Inf (Infinity) or a NaN (Not a Number) value. You can use this syntax to catch a situation where a program you are running experiences a problem and computes an infinite value. You can then stop the program before it proceeds through a long string of calculations on the erroneous Inf or NaN value. This is especially useful when performing a numerical simulation using the ODE solvers or optimization functions in MATLAB.

Each of these DBSTOP syntaxes have a corresponding syntax for the DBCLEAR function that will clear the breakpoint or stop condition (error, warning, naninf, or infnan) that was set for the call to DBSTOP.

Note: The dbstop if all error syntax has been omitted from the help and documentation for the DBCLEAR function, but this syntax still functions in the same manner as the other syntaxes. This documentation error will be corrected in a future version of MATLAB.

There are two additional syntaxes for DBCLEAR that also clear breakpoints and stop conditions but that do not directly correspond to a DBSTOP syntax. These are

```
dbclear all in [m-file]
```

which clears all breakpoints and stop conditions in the [m-file].m file, and

```
dbclear all
```

which clears all breakpoints and stop conditions in all M-files.

If you want to clear all the variables in your workspace as well as all breakpoints in your files, you do not need to

use two commands to do this. The CLEAR function will clear breakpoints when used with the `all, fun,` or `functions` arguments; calling it using the syntax

```
clear all
```

will clear the breakpoints in all files and the variables in the current workspace; however:

```
clear all
```

will not clear the special trap conditions, such as `dbstop if error`.

To query where your breakpoints are located, you can use the DBSTATUS function. If you call DBSTATUS with no input arguments, this command lists all the lines on which breakpoints are located for all files on the MATLAB path, and tells you if any of the special conditions (`DBSTOP IF ERROR`, `DBSTOP IF ALL ERROR`, `DBSTOP IF WARNING`, or `DBSTOP IF NANINF`) are set. If you call it with the name of an M-file as the input, DBSTATUS only lists lines in that function with breakpoints and the special condition breakpoints. To view the lines on which DBSTATUS indicates that breakpoints exist without opening up the Editor/Debugger, use the DBTYPE function.

If you want to stop execution of your M-file on a certain line but are worried about accidentally clearing it, you can add a call to the KEYBOARD function in your M-file. This function causes MATLAB to enter debug mode when it is executed, exactly as though you reached a breakpoint. However, unlike a breakpoint, it cannot be removed by CLEAR or DBCLEAR and will not appear on the DBSTATUS list.

Section 15: Moving from Workspace to Workspace

Once you have entered debug mode (by reaching a line with a breakpoint or a KEYBOARD statement on it or satisfying a trap condition set using DBSTOP) you can use the DBUP and DBDOWN functions to move up and down the calling stack. This allows you to trace an error back to the calling M-file if it involves incorrect data being passed from a calling function's workspace to the workspace of the function where the error occurs. This is not an issue with script files, as they use the workspace of their caller (or the base workspace if called from the command line or another script.) To display the entire calling stack, use the DBSTACK function. As an example, download the `testdbstack.m` file, and type

```
dbstop if error
```

Then run it by typing

```
testdbstack
```

Once you do this, you will see the following error message and the prompt will change to the debug mode prompt:

```
   This function is designed to demonstrate the use of the DBSTACK, DBUP, and
   DBDOWN functions.
   I will now pass a negative number to a function that expects a positive
   input.

??? Error using ==> testdbstack (mytestfun)
   I expected a positive input
```

The Editor/Debugger will open (if it is not already open) to line 13. Verify that it stopped on line 13 by typing

```
dbstack

   >In d:\applications\matlabr13\work\testdbstack.m (mytestfun) at line 13
    In d:\applications\matlabr13\work\testdbstack.m at line 8
```

This is the calling stack. First MATLAB started `testdbstack` and executed it up to line 8. Line 8 contains a call to a subfunction, `mytestfun`, so that subfunction call is added to the stack. The error occurred on line 13 of that function, which the stack reflects.

Check what value was passed to `mytestfun` by moving up the stack 1 level using

```
dbup
```

and verifying that it moved up with

```
dbstack
```

```
In d:\applications\matlabr13\work\testdbstack.m (mytestfun) at line 13
>In d:\applications\matlabr13\work\testdbstack.m at line 8
```

The green arrow in the Editor/Debugger will move from line 13 of the file to line 8, to show the position of the last command to be executed. Now, verify that the variable `MyInput` referred to on line 8 has the expected value (or confirm that the unexpected value of `MyInput` is the cause of the problem) by typing

```
MyInput
```

at the MATLAB command prompt. Since `MyInput` is passing an unexpected value to the function `mytestfun`, you have narrowed down the cause of the error message. Now, check how `MyInput` was computed and discover the problem – an unexpected minus sign.

Section 16: Executing Your Code Using the DBSTEP Function

If you want to simply run your code, line by line, and verify the output after each step, you can use the `DBSTEP` function inside debug mode. Simply set a breakpoint (in the Editor/Debugger or use `DBSTOP`) at the line where you want to start your step-by-step investigation of the code and run the program. At the debug mode prompt, type

```
dbstep
```

This will cause MATLAB to execute one line of code and remain in debug mode. There are also three alternate syntaxes for `DBSTEP`, which will allow you to do more than simply step through one line:

- **dbstep N**
  This will cause MATLAB to execute the next N lines, where N is a positive integer.
- **dbstep in**
  If the next line to be executed is a call to another function, this syntax will cause the Editor/Debugger to open that function. MATLAB will stop at the first line in that function, as though there was a breakpoint at that line.
- **dbstep out**
  This syntax will cause MATLAB to execute the rest of the lines in the current function, then return to the calling function and stop on the line immediately after the function call.

Section 17: Displaying Status Messages Periodically

One of the simplest, yet sometimes most effective, debugging techniques is to display the results of certain key calculations or simply a string that indicates when a certain section of code has completed execution. The `DISP` function provides an easy way to do this. For example, in the sample code below if

```
Beginning program
```

and

```
Step 1 complete
```

are displayed but

```
Step 2 complete
```

is not, it is likely that there is a problem with the function `step2`:

```
disp('Beginning program')
step1
disp('Step 1 complete')
step2
disp('Step 2 complete')
```

Section 18: Using the TRY/CATCH Block to Capture Errors

Sometimes you want to execute a line of code that you know will fail under certain circumstances, but you don't want the failure of that line of code to cause your entire program to fail. The way to allow this line of code to execute but capture the error if it occurs is to use a `TRY/CATCH` block. The line of code to be executed will be the `TRY` section of the block, while the error handling code makes up the `CATCH` block. The code in this M-file demonstrates how to use a `TRY/CATCH` block. In this example, MATLAB will attempt to open a file, read the first few characters, and close the file. The `TRY/CATCH` block handles the case where the file you specify does not exist or there is a problem opening it. Call it with the name of a text file as the input argument, and then call it with a string that is not the filename of a text file on your system as input. If the `TRY/CATCH` block did not exist, the attempt to open the nonexistent file would return an error -- with `TRY/CATCH`, you can handle that scenario.

Another scenario that could cause problems is if the call to `FSCANF` function in the example file failed and terminated execution of the example. Without the `TRY/CATCH` block, the file you opened would remain open, potentially causing it to become corrupted or even causing your operating system to run out of file handles. By using the `TRY/CATCH` block, you can clean up open files and other resources you have used even if the code you are using causes an error during its execution.

Another situation in which the `TRY/CATCH` block is useful is when you are dealing with user input and interaction with your program. Since you cannot be certain what information your users will provide in all circumstances, `TRY/CATCH` will allow your code to handle unexpected input. For instance:

```
try
NumApples = input('How many apples do you wish to purchase?');
catch
disp('You entered an invalid number of apples!')
end
```

If you executed the `INPUT` statement alone, without including it in a `TRY/CATCH` block, any invalid input users have entered would cause the program to exit with an error. Using a `TRY/CATCH` block, though, you can warn users that they have done something unexpected and deal with it appropriately.

Section 19:Using the ERROR Function with the LASTERR and RETHROW Functions

Inside your `TRY/CATCH` block, you may want to proceed differently based on which specific error message you received; for instance, trying to open a nonexistent file would lead to a

```
File not found
```

error. You may want to allow the user to select a new filename to open rather than proceeding or terminating the program completely. You can determine the specific error that triggered the `CATCH` block using the `LASTERR` function.

The LASTERR function will return the error string and, if the error has one, the message identifier (that was introduced in MATLAB 6.5 (R13).) You can then compare the error string to a specific error message you want to trap. In MATLAB 6.5 (R13), there is a function similar to LASTERR that instead returns the error string and message identifier in a structure; this is the LASTERROR function.

Alternatively, after performing your error cleanup inside the CATCH block, you may want to trigger the error that forced you into the CATCH block, to alert the user that something has gone wrong. You can do this using the ERROR function (which explicitly throws an error) in combination with the LASTERR function or with the functions RETHROW and LASTERROR. Both ERROR and RETHROW will cause an error with the error string or structure returned by LASTERR or LASTERROR. If your error has a message identifier associated with it, RETHROW will allow you to avoid creating variables for the error string and message identifier separately. RETHROW also does not add the `??? Error using` line that ERROR adds to the beginning of each error message, so you will not see multiple lines starting with three question marks in the RETHROW error message.

Finally, there is a function LASTWARN that is similar to LASTERR but returns the last warning message. You can use this for similar purposes as LASTERR; you can use it to check if a specific warning occurred, to re-issue a warning whose display you suppressed, or to check if any warnings occurred. The WARNING function serves the same purpose for warnings as ERROR does for errors; you can use it to re-issue a warning.

You can download a sample function file that demonstrates the use of the LASTERR, LASTERROR, and RETHROW functions here.

Section 20: The WHICH Function

The WHICH function will display the location of the first file on the MATLAB path whose name matches the string that was passed to WHICH. This function will also tell you if the string is the name of a variable in the current workspace or is the name of a built-in function. You can also provide the WHICH function an additional argument, the -ALL flag, which will list all the variables, built-in functions, and files whose names match the string provided to WHICH. This allows you to verify that you are using the correct version of a file when there may be multiple files with the same name on the MATLAB path.

One of the most common cases where the WHICH function is useful in finding a problem with your code occurs when attempting to call a function. If another function with that name is located higher on the MATLAB search path or a variable with the function name exists, MATLAB will use the other function or the value of the variable instead of the function you intend. For example, if you define a variable why

```
why = 1:10;
```

then attempt to call the MATLAB built-in WHY function

```
why(37)
```

you will receive the following error:

```
??? Index exceeds matrix dimensions.
```

It is easy to use WHICH to find out whether why is a function or a variable. If you now type

```
which why
```

you now see

```
why is a variable.
```

rather than what is expected

```
D:\Applications\MATLABR13\toolbox\matlab\elmat\why.m
```

Common Errors and Warnings

Section 21: Download the Examples

To obtain the complete set of examples used in the following sections, please download the `.ZIP` file from the following location:

ftp://ftp.mathworks.com/pub/tech-support/tech-notes/1207Examples.zip

The corresponding example for each error or warning is listed at the end of each section. You should open the example listed at the end of each section in the Editor/Debugger before reading the description, so you can compare the code in the example file that produces the error with the explanation in the section.

Note: In the Solutions listed below, the sentence *Stop MATLAB on the line where the error [warning] occurs.* means

1. Set a breakpoint using `DBSTOP` or the Breakpoints menu in the Editor/Debugger if MATLAB detects an error (or set a breakpoint on the line where the error occurs, if you know the line number).
2. Execute your code.
3. Examine the line on which the debugger stops.

Errors

Section 22: Error Using ==> Reshape

```
??? Error using ==> reshape
Don't do this again!.

??? Error using ==> reshape
Cleve says you should be doing something more useful.

??? Error using ==> reshape
Seriously, size argument cannot be negative.
```

Explanation:
You are attempting to reshape a matrix with a negative dimension. This error is very specific, but it is included in this Tech Note because it often confuses people the first time they encounter it. The error message changes the second and third times you encounter this error. This is an Easter egg in MATLAB. Note that if you execute code that causes this error more than three times, the error message will not change after the third execution.

Common causes:
You have called the `RESHAPE` function with a negative value for one of the dimensions.

Solution:
Stop MATLAB on the line where the error occurs. Verify that the variables containing the dimensions to which you want to reshape the array are positive integers.

Example demonstrating this error:
`ReshapeError.m`

Section 23: Subscripted Assignment Dimension Mismatch

```
??? Subscripted assignment dimension mismatch.

??? In an assignment A(I) = B, the number of elements in B and I must be the
same.

??? In an assignment A(matrix,matrix) = B, the number of rows in B and the
number of elements in the A row index matrix must be the same.

??? In an assignment A(matrix,matrix) = B, the number of columns in B and the
number of elements in the A column index matrix must be the same.
```

Explanation:
You are attempting to assign more elements to a section of a matrix or vector than that section can hold. These four error messages occur when you

- Use a colon operator to specify an entire row or column in the destination matrix (the first error message)
- Use linear indexing with an incorrectly sized index array (the second error)
- Use row/column indexing with an incorrectly sized row index array (the third error)
- Use row/column indexing with an incorrectly sized column index array (the fourth error)

Common causes:
You are attempting to replace a section of one matrix with another and the size of the index vector or vectors used to specify the section of the destination matrix to be replaced does not match that of the source matrix. Alternatively, you are attempting to assign values to individual elements using row and column indices, which allocates a submatrix section of the original matrix.

Solution:
Stop MATLAB on the line where the error occurs. Verify that the size of the index array or arrays agrees with the size of the array you are attempting to assign to the destination matrix. The specific error message you receive will tell you which dimensions of the index array and the vector to be assigned are mismatched. Also, you should verify that the computations that created the index array and the vector or matrix to be assigned are correct. An unexpected size may indicate that the computations that created the matrix were incorrect.

Finally, you may want to use the `REPMAT` function to expand a smaller matrix to fill a large block, make use of scalar expansion to fill the region to which you are trying to assign, or use the `SUB2IND` or `IND2SUB` function to generate an appropriate index vector or set of index vectors.

Example demonstrating this error:
`AssignmentSizeMismatch.m`

Section 24: ")" Expected, "End of Line" Found

```
")" expected, "end of line" found.

Error: Missing operator, comma, or semicolon.
```

Explanation:
You have too many left parentheses (generally, this causes the first error message) or not enough right parentheses (the common cause of the second error message) in this line of code.

Common causes:

You added in a left parenthesis or removed a right parenthesis from the line of code.

Solution:
Count the number of left parentheses and right parentheses on the line of code. Verify that the quantity of the two types of parentheses are equal. Add in an appropriate number of right parentheses or remove extraneous left parentheses. You can also enable a setting in the **Keyboard & Indenting** window in the Editor preferences that will automatically detect mismatched parentheses when you type a new parenthesis or move the cursor past a parenthesis.

Example demonstrating this error:
`ParenthesisExpected.m`

Section 25: ‹ Function › Not Defined For Variables of Class ‹ Class ›

```
‹function› not defined for variables of class ‹class›
```

Explanation:
You are attempting to use a function that has not been defined to accept data of the class of the input argument.

Common causes:
You are attempting to perform arithmetic on variables of a class other than double arrays.

Solution:
One solution is to overload the function listed in the error message for use on variables of the class listed in the error message. For more information on overloading functions and MATLAB object-oriented programming, please see Chapter 21 of the Using MATLAB (PDF) manual, which discusses MATLAB classes and objects. As another solution, you can convert the variables of the listed class into double arrays or some other data type for which the appropriate function has been overloaded using the `DOUBLE` function (for doubles) or the appropriate conversion function for the other data types. Finally, if you have access to the Image Processing Toolbox 3.0 (R12+) or higher, you can perform arithmetic on integer arrays (`uint8`, `int8`, etc.) using the `IMADD`, `IMSUBTRACT`, `IMMULTIPLY`, and `IMDIVIDE` functions. These functions will perform arithmetic on arrays regardless of data type, as long as the types are consistent.

Example demonstrating this error:
`FunctionNotDefinedForClass.m`

Section 26: All Rows in the Bracketed Expression Must Have the Same Number of Columns

```
All rows in the bracketed expression must have the same number of columns.

All matrices on a row in the bracketed expression must have the same number of
rows.
```

Explanation:
You are attempting to concatenate two arrays using `[]` but the appropriate dimensions along which to concatenate are not the same size.

Common causes:
You have forgotten a semicolon (`;`) or a comma (`,`) when attempting to concatenate two character arrays, or you are using the wrong separator. This error commonly occurs when attempting to concatenate character arrays to try to create a multi-line string.

Solution:

Examine the line listed in the error message and insert the appropriate matrix element separator. Also verify that the dimensions along which you are trying to concatenate the matrices agree. The help files for the HORZCAT and VERTCAT functions describe the relationship that needs to exist for this concatenation to work correctly.

```
horzcat(A,B)
```

is the equivalent of `[A B]` while

```
vertcat(A,B)
```

is the equivalent of `[A;B]`. If you are attempting to concatenate strings, you should use the STRCAT and STRVCAT functions, which handle adjusting the appropriate dimensions automatically if it is necessary.

Example demonstrating this error:
`BracketedExpressionDimensions.m`

Section 27: Attempt to Execute SCRIPT ‹ Filename › as a Function

```
Attempt to execute SCRIPT as a function.
```

Explanation:
You are attempting to call a script file as though it were a function file by passing arguments to it.

Common causes:
You have created a script M-file with the same name as a function on the MATLAB path but higher on the path than that function, or are attempting to index into an array that does not exist when a script file with that name exists.

Solution:
Execute

```
which -all ‹filename›
```

for the file specified in the error message. Verify that the file you expect to execute is being used instead of any other file with the same name. If it is not, rename or delete the files higher on the MATLAB path than the file you expect to use, or reorder the directories on your MATLAB path to place the version of the file you want to use higher than any other version. For more information on the MATLAB path and file precedence, please see the information on How the Search Path Works.

Example demonstrating this error:
`AttemptExecuteScriptAsFunction.m` and `TestScriptAsFunction.m`

Section 28: Error When Evaluating Uicontrol Callback

```
Error when evaluating uicontrol Callback.
```

Explanation:
A callback function for a uicontrol in your figure or GUI has thrown an error.

Common causes:
There are many possible causes for this error. The section of the error message that follows the line above will explain the cause of the error; the line above simply identifies that the error is located in a uicontrol's callback function.

Solution:
Debug the callback function as normal based on the second portion of the error message returned from MATLAB. Search for that portion of the error message in this Tech Note.

Example demonstrating this error:
N/A

Section 29: Index Exceeds Matrix Dimensions

```
Index exceeds matrix dimensions.
```

Explanation:
You are attempting to reference a nonexistent element of an array.

Common causes:
Changing the size of an array unexpectedly, or encountering an empty array when you expected a nonempty array.

Solution:
Stop MATLAB on the line where the error occurs. Verify the size of the array into which you are referencing is large enough that the index you are using falls before the last element of the array -- one way to do this is to check the Workspace Browser. Verify that the variable was correctly created at each of the previous lines, starting with the most recent, where it appeared on the left side of an assignment statement.

Example demonstrating this error:
`IndexExceedsMatrixDimensions.m`

Section 30: Matrix Dimensions Must Agree

```
(Inner) matrix dimensions must agree.
```

Explanation:
You are attempting to perform a matrix operation, which requires certain matrix dimensions to agree, on matrices that do not satisfy this requirement.

Common causes:
You are attempting to multiply or divide two matrices where the number of columns in the first is not equal to the number of rows in the second (for `*`) or the number of rows do not match (for `\`). This often indicates that you are performing matrix operations when you instead intended to perform array operations.

Solution:
Stop MATLAB on the line where the error occurs. Verify that you are not performing an extra transpose operation or omitting one where necessary. Also verify the sizes of the matrices, which you are multiplying or dividing, agree in the corresponding dimensions. You can do this using the Workspace browser or the `SIZE` function. If you intended to perform array operations instead of matrix operations, replace the *, /, \, or ^ matrix operators with the .*, ./, .\, or .^ array operators instead. If you pass your formula as a string to the `VECTORIZE` function, `VECTorIZE` will return the formula with the matrix operators (*, /, and ^) replaced by the array operators (.*, ./, .^).

Example demonstrating this error:
`MatrixDimensionsMustAgree.m`

Section 31: Invalid ‹ Object › Property: ‹ Property ›

```
Invalid ‹object› property: ‹property›.
```

Explanation:
You are attempting to set a property of a Handle Graphics object, but that property is not a property of that Handle Graphics object. Alternatively, that name does not correspond to a property of any Handle Graphics object.

Common causes:
You are attempting to set a property, such as `FontName` or `Callback` for a figure, but figure objects have neither a `FontName` nor `Callback` property. If you know that the object has that property, there may be a typographical error in the name of the property.

Solution:
Verify that the name in the `SET` or `GET` statement on the line listed in the error message is spelled correctly and that the property you are attempting to access with the `SET` or `GET` function is a valid property for that type of Handle Graphics object using the Handle Graphics Object Properties browser.

Example demonstrating this error:
`InvalidObjectProperty.m`

Section 32: Invalid Handle

```
Invalid handle.
```

Explanation:
You are using Handle Graphics commands to try to manipulate a handle to an object that no longer exists.

Common causes:
You have removed the Handle Graphics object from the figure or GUI before attempting to manipulate it using `SET` or `GET`. Another common cause is creating objects inside a `For-loop` without turning `HOLD` on; this frequently occurs using the `PLOT` function.

Solution:
Stop MATLAB on the line where the error occurs. Verify that the reference you are using is to an existing Handle Graphics object by backtracking in the code to the statement that created the Handle Graphics object. Also verify that the line, which was intended to create the object, did so by setting a breakpoint on it and executing the function again. The `FINDALL`, `FINDOBJ`, `ALLCHILD`, and `ISHANDLE` functions allow you to find the handles of objects and verify that the objects exist.

Example demonstrating this error:
`InvalidHandle.m`

Section 33: Matrix Must Be Square

```
Matrix must be square.
```

Explanation:
You are attempting to perform an operation that is only defined for square matrices on a matrix with an unequal number of rows and columns.

Common causes:
You are attempting to use an element-wise operator on a matrix that is nonsquare, but you are using the linear algebra operator instead. For example, you use the `^` operator (taking the power of a matrix) rather than `.^` (which takes the power of each element of the matrix).

Solution:
Examine the line listed in the error message and verify the matrix whose power you want to take is square, or that you are using the appropriate operator.

Example demonstrating this error:
`MatrixMustBeSquare.m`

Section 34: Product of Dimensions Is Greater Than Maximum Integer

```
Product of dimensions is greater than maximum integer.
```

Explanation:
You are attempting to create a full matrix with more elements than the maximum number of elements allowed in MATLAB.

Common causes:
MATLAB will attempt to create a matrix with a large number of elements, as long as that number of elements is less than the maximum number of elements allowed in a matrix. You can determine this maximum using the `COMPUTER` function.

Solution:
If the matrix you are attempting to create has relatively few nonzero elements, you may be able to create it as a sparse matrix. You can use the `SPARSE` function and the other sparse matrix manipulation functions to create and manage this matrix. Type

```
help sparfun
```

for a list of the sparse matrix manipulation functions. Note, however, that the limitation on the maximum number of elements still exists; now it only applies to the nonzero elements of the sparse matrix. If your matrix is not sparse, however, you will need to break it into sections with a number of elements less than the maximum returned by the `COMPUTER` function.

Example demonstrating this error:
`ProductDimensionsTooLarge.m`

Section 35: Too Many Input/Output Arguments

```
Too many output arguments.

Too many input arguments.

Not enough input arguments.
```

Explanation:
A function you are trying to call expects fewer input/output arguments, or more input arguments, than you have provided it

Common causes:
You have passed a function more input arguments than it expected to receive, perhaps by passing a list of inputs rather than a vector of inputs, or have tried to obtain two outputs from a function that only returns one.

Solution:
Stop MATLAB on the line where the error occurs. Verify that you have specified the correct number of input and/or output arguments, using `WHICH` to determine which version of the function you are using if necessary. If

this occurs immediately when you try to run your main function, verify that it can accept and/or return sufficient arguments. Another way to debug this problem is to use the NARGIN and NARGOUT functions. When you pass NARGIN or NARGOUT the name of the function, it returns the number of input or output arguments a function can accept, respectively. When used inside a function, it returns how many inputs or outputs are actually specified in the call to the function. Finally, the DBTYPE function, when used with the name of a file and the number 1 as inputs, will display the first line of the M-file (which shows the input and output arguments the function expects).

Example demonstrating this error:
`TooManyArguments.m`

Section 36: Undefined Function or Variable

```
Undefined function or variable.
```

Explanation:
MATLAB does not recognize the specified string as the name of a function on the MATLAB path or a variable visible to that function.

Common causes:

- You have made a typographical error when typing a command or variable name (such as typing a 1 [one] when you intended to type an l [L]).
- You have changed directories so that a function you used is no longer on MATLABs search path.
- You have used the wrong case for a function or variable name.
- You are trying to use a function for which you are not licensed.

Solution:
Stop MATLAB on the line where the error occurs. Verify that the undefined function or variable is visible to the function at that point (it is on the path or in the current workspace) and that it has been defined before this line of code executes. Check if a datatip appears when you move the mouse pointer over the variable name on that line. (Point at the following arrow for an example: >>)

If you are certain the variable or function exists, verify the case of the function or variable name. If the undefined identifier is a function, the WHICH function can help you verify that it is visible to the function where the error occurs.

Example demonstrating this error:
`UndefinedFunctionorVariable.m`

Warnings

Section 37: Suppressing Warning Messages

You can enable the display of line numbers in MATLAB warnings using the BACKTRACE option of the WARNING function. In addition, you can suppress all warning messages using the OFF state option. Once you have suppressed all warnings, you can enable the warning messages again using the ON state option. As a new feature in MATLAB 6.5 (R13), certain warnings can be selectively suppressed, as long as they have been given a message identifier. To do this, call WARNING using the OFF option and provide the message identifier as a second argument.

Section 38: Divide by Zero

```
Warning: Divide by zero.
```

Explanation:
Your calculations involve dividing by zero.

Common causes:
You are performing a division calculation (especially element-wise division, `./`) and one of the elements in the denominator of your calculation is 0.

Solution:
Stop MATLAB on the line where the warning occurs. Examine your calculations on that line and perform them step-by-step, searching for a situation where you divide by 0. If this is not expected behavior, verify that the variables and numbers involved in the calculation of the quantities in the denominator of your expression are being computed correctly. Alternatively, if this is expected behavior (and you are prepared to accept any Inf or NaN values you receive in the result), you can disable this specific warning in MATLAB 6.5 (R13) using the following command

```
warning off MATLAB:divideByZero
```

Example demonstrating this error:
`DivideByZero.m`

Section 39: Missing Operator, Comma, Semicolon, or White Space

For Versions of MATLAB up to MATLAB 6.1 (R12.1):

```
Future versions of MATLAB will require that whitespace, a comma,
or a semicolon separate elements of a matrix.
```

For MATLAB 6.5 (R13):

```
Missing operator, comma, semicolon, or white space.
```

Explanation:
You have tried to create a matrix with elements that are not separated by a comma, semicolon, or whitespace character (space, line break, etc.) This causes MATLAB 6.1 (R12.1) and earlier versions to return a warning; this warning was changed to an error in MATLAB 6.5 (R13).

Common causes:
You have forgotten to include a space or comma when you tried to enter a matrix. Alternatively, you are trying to perform an operation and forgot to include the operator. For example, the syntax

```
(1)(2)
```

will not multiply 1 and 2, but will give a warning or error.

Solution:
Examine the line listed in the error message and insert the appropriate matrix separator or operator where needed. Try typing the following:

```
help matrix_element_separators
```

for more information.

Example demonstrating this error:
`MissingElementSeparators.m`

Section 40: Imaginary Parts of Complex X and/or Y Arguments Ignored

```
Warning: Imaginary parts of complex X and/or Y arguments ignored.
Warning: Imaginary parts of complex X, Y, and/or Z arguments ignored.
```

Explanation:
You are attempting to plot using two complex inputs to a plotting function, like PLOT or PLOT3. In this case, MATLAB will plot using the real part of the first input as the independent variable X and the real part of the second input as the dependent variable Y.

Common causes:
You have performed a square root or FFT operation on the vectors you are attempting to plot, and those operations resulted in a complex vector. Alternatively, you used the variables I or J in the computation of your input vectors but those variables did not exist when you performed your computation. In this case, MATLAB will treat these as the imaginary unit.

Solution:
Stop MATLAB on the line where the warning occurs. Verify that the two vectors or matrices you pass to the PLOT function, or the three you pass to PLOT3, are not complex. If you want to plot the real part of a vector versus the complex part, pass the vector as a single complex vector to the PLOT function. If you want to plot the magnitude of the elements, use the ABS function on the vector before passing it to the PLOT function.

Example demonstrating this error:
`ImaginaryPartIgnored.m`

Section 41: Integer Operands Are Required for Colon Operator When Used as Index

```
Warning: Integer operands are required for colon operator when used as index.
```

Explanation:
You have used a noninteger value as one of the parameters (starting value, increment, or stopping value) for the colon (:) operator when using it to create a vector of indices to reference into a function.

Common causes:
You performed computations to obtain the starting or stopping values for the indexing but the result of those computations was not exactly an integer.

Solution:
Modify the index computations using the FIX, FLOOR, CEIL, or ROUND functions to ensure that the indices are integers. You can test if a variable contains an integer by comparing the variable to the output of the ROUND function operating on that variable when MATLAB is in debug mode on the line containing the variable.

Example demonstrating this error:
`IntegerOperandsRequired.m`

Section 42: Matrix is Close to Singular or Badly Scaled

```
Warning: Matrix is close to singular or badly scaled.
```

Explanation:
A matrix used in a computation is singular or is very close to being singular (illconditioned). Trying to solve a

system of linear equations whose coefficient matrix is singular can lead to incorrect answers.

Common causes:
A program you called is using a matrix that is badly conditioned as one of the arguments to the backslash (\) or forward slash (/) operators, which are commonly used to solve systems of linear equations.

Solution:
Stop MATLAB on the line where the warning occurs. Examine the line for instances of the two slash operators. If you find one, determine the condition number of the coefficient matrix (the matrix on the left side of the backslash (\) or on the right side of the forward slash (/) using the COND function. Large results for the condition number indicate the matrix is extremely illconditioned. You should verify (using the matrix multiplication operator (*)) that the result of solving the system is numerically reliable. For more information about condition numbers of matrices, please see Chapter 10 of the Using MATLAB (PDF) manual, which discusses conditioning of matrices used in solving systems of linear equations.

Example demonstrating this error:
`CloseToSingular.m`

Section 43: One or More Output Arguments Not Assigned During Call to ‹Function›

For Versions of MATLAB up to MATLAB 6.1 (R12.1):

```
Warning: One or more output arguments not assigned during call to ‹function›.
```

For MATLAB 6.5 (R13):

```
??? One or more output arguments not assigned during call to ‹function›.
```

Explanation:
One of the functions you have called is defined to return an output argument but that output argument does not exist in that function when it tries to return.

Common causes:
You have misspelled the name of one of your output arguments inside your function, or you have forgotten to assign a value to one of the output arguments of your function. Alternatively, the function was originally written with one or more output arguments, but the section of the function that computed the output argument was removed or modified in such a way that the output argument is now extraneous.

Solution:
Stop MATLAB on the last line of the function listed in the warning or error message. Verify that each of the output arguments listed in the function declaration line at the beginning of the function exists after that last line is executed (using the DBSTEP function or the Step button in the Editor). If the arguments do not, examine the function to determine where you intended the arguments to be declared. Verify that those lines of code are being executed and have no typographical errors.

Example demonstrating this error:
`OutputNotAssigned.m`

Section 44: OpenGL Not Available. Using ZBuffer.

```
Warning: OpenGL not available. Using ZBuffer.
```

Explanation:
You are attempting to display a plot that MATLAB tries to render using OpenGL, but OpenGL is not available or

is not functioning correctly on your computer. Therefore, MATLAB will use the ZBuffer renderer instead.

Common causes:
The common causes for OpenGL being unavailable or functioning incorrectly on your machine are listed in Tech Note 1201: The Graphics Rendering and Troubleshooting Guide.

Solution:
Please see Tech Note 1201: The Graphics Rendering and Troubleshooting Guide for information on how to correct this problem.

Example demonstrating this error:
N/A

Section 45: Subscript Indices Must Be Real Positive Integers or Logicals

For Versions of MATLAB up to MATLAB 6.1 (R12.1):

```
??? Index into matrix is negative or zero. See release notes on changes to
logical indices.
Warning: Subscript indices must be integer values.
```

For MATLAB 6.5 (R13):

```
?? Subscript indices must either be real positive integers or logicals.
```

Explanation:
You are attempting to index into a matrix or vector using a set of indices that include a number that is not a positive integer or a logical value. Indexing using a nonintegral positive value causes MATLAB 6.1 (R12.1) and earlier versions to return a warning; this warning was changed to an error in MATLAB 6.5 (R13). Both MATLAB 6.1 (R12.1) and MATLAB 6.5 (R13) return an error when you attempt to index into a matrix using a negative number or 0.

Common causes:
The most common cause of this warning or error is that you are indexing into one matrix with another matrix which contains 0's and 1's (this is called logical indexing) but those 0's and 1's do not have the logical attribute in MATLAB 6.1 (R12.1) and before or the index matrix is not a logical array in MATLAB 6.5 (R13). Alternatively, if you compute indices using floating-point arithmetic, the indices may not be exactly an integer. If the indices are not an integer, they will be rounded to the nearest integer and used to index into the matrix.

Solution:
Stop MATLAB on the line where the warning or error occurs. Verify that the indices you are using to index into the matrix are either positive integers or logical 0 values. To determine if the index vector X contains only integers, you can use the syntax

```
X==round(X)
```

or examine the values in the vector X-round(X) using the HEX format. See the documentation on the `FORMAT` function for more information on the HEX format.

Example demonstrating this error:
`InvalidIndices.m`

Section 46: Conclusion and Further Information

You should now have a better grasp at how to use the debugging tools from MATLAB, both from inside the

MATLAB Editor/Debugger and from the command line. You should also be able to diagnose and correct some of the most frequently encountered errors experienced by authors of MATLAB programs. The techniques you have learned will serve you well in tracking down and eliminating less frequent errors that you may experience in the future when you write your own M-files.

If you are receiving an error message that is not listed in this Tech Note and for which you do not know how to debug and correct, there are several ways you can obtain more information to help you deal with that error. One location from which you can obtain more information is the MathWorks Technical Support Web site, http://www.mathworks.com/support ; specifically the Solutions and Technical Notes. It is very likely that another MATLAB user has requested information about a similar error you may be encountering, and this information has been recorded in an online solution or a Tech Note like this one. You can search for information in the entire database or look for a solution specific to a toolbox you're using on the product page for that toolbox.

If the error message is not listed in the solution database, there are three other means by which you can request help.

- You can search the archives of the MATLAB newsgroup `comp.soft-sys.matlab` via the MATLAB Central Newsgroup Access, or within the Google newsgroup archive. Often, users post questions about error messages they are receiving and receive information, suggestions, corrections, and explanations from other users.
- If you cannot find an answer in the MathWorks Technical Support Web site or within the `comp.soft-sys.matlab` archives, you can post a question to the newsgroup yourself. You can use either of the archive links in the previous paragraph to post your question, or you can post via your Internet provider's news server. Generally, a message describing what you have already tried to do to correct the problem along with the error message will produce a response from one of the experienced users who read the newsgroup.
- If neither the solution database nor the newsgroup archives provides a solution for the problem you're experiencing and you do not receive a response from `comp.soft-sys.matlab`, you can contact the MathWorks Technical Support department, which you can reach by phone, e-mail, or by filling out the Web form located at the following URL:

  http://www.mathworks.com/support/contactus.shtml

If you frequently experience an error related to M-file programming not listed in this Tech Note and want to see it listed in the future, please fill out the form shown at the bottom of this Tech Note. Please include a copy of the exact error message, as well as a small (no more than 10 line) section of code that reproduces the error, the techniques you have used to determine the cause of the error, and your contact information. If you have not determined the cause of the error, please check the Technical Support Knowledge Base and/or contact our Technical Support department before submitting the form below.

Once you have written and debugged your MATLAB program, you may be interested in optimizing its performance. Resources you can consult to assist you in optimizing your program are the Tech Note 1109: The Code Vectorization Guide, and both Chapter 22, Maximizing MATLAB Performance and Chapter 23, MATLAB Programming Tips, of the Using MATLAB (PDF) manual.

Did this information help?	Yes ○	No ○	Didn't Apply ○
Is the level of technical detail appropriate?	Yes ○	Too Much ○	Not Enough ○

If you require further assistance, please submit a request.
A member of our Technical Support Staff will then contact you.

We would like your opinion on the information presented above.

What did you expect to find on this page that you want us to consider adding?

Additional Comments:

SUBMIT

related topics:

Demos | Search | Contact Support | Consulting | Press Room | Usability