Armstrong Atlantic State University
Engineering Studies
MATLAB Marina – Debugging Primer

Prerequisites
The Debugging Primer assumes knowledge of the MATLAB IDE, MATLAB help, arithmetic operations, built in functions, scripts, variables, arrays, logic expressions, conditional structures, iteration, and functions. Material on these topics is covered in the MATLAB Marina Introduction to MATLAB module, MATLAB Marina Variables module, MATLAB Marina Arrays module, MATLAB Marina Logic Expressions module, MATLAB Marina Conditional Structures module, MATLAB Marina Iteration module, and MATLAB Marina Functions module.

Learning Objectives
1. Be able to identify the type of programming error from MATLAB messages.
2. Be able to use the MATLAB debugger to set breakpoints and debug programs

Terms
debug, syntax error, run time error, logic error, breakpoint, single step (step)

MATLAB Functions, Keywords, and Operators
None

Debugging
When a program or function doesn't operate correctly, what do you do? Debugging a program involves systematically verifying correct/incorrect operation and correcting incorrect portions of the program. This typically involves running the program for various test cases to identify potential errors and correcting each error as they are found. Always correct errors that appear earlier in the program first as correcting these may eliminate errors that occur later.

Programming Errors
There are three types of errors that can occur in programs: syntax errors, run-time errors, and logic errors.

Syntax errors are errors in the programming languages grammar (syntax) rules. Syntax errors are detected at compile time. MATLAB syntax errors are detected when the program is run and each line is compiled before being executed. Examples of syntax errors in MATLAB are misspelling a keyword such as `for` and not terminating an `if` statement with a corresponding `end` statement. MATLAB will display an error message including the line number where the error occurred in the Command Window for syntax errors. Figure 1a shows a program with a syntax error and Figure 1b shows the error message that MATLAB displays when the program is run.

Syntax errors can be corrected by finding the line number the error occurred at and correcting the syntax. Sometimes the cause of syntax errors is earlier than the indicated line. For example if a variable is used in a formula that was not previously defined, MATLAB will indicate the error occurs at the line number of the formula and the correction is assigning a value to that variable earlier in the program.

```
angle = pie/7;
result = cos(angle);
disp(result);
```

Figure 1a, MATLAB Program with Syntax Error

```
Error using pie (line 30)
Not enough input arguments.

Error in Program1 (line 1)
angle = pie/7;
```

Figure 1b, MATLAB Error Message for Syntax Error in Program of Figure 1a

The syntax error on line 5 in the program of Figure 1a is due to a misspelling of the built in π constant. The error message "Not enough input arguments" is misleading as it is an error message that occurs when a function is not invoked correctly. MATLAB has a built in pie chart function named `pie`, and this is the error that MATLAB identified rather than a misspelling of `pi`. The line number indicated by the error message is good place to look for the syntax error but the error message may not be appropriate for what the programmer intended.

Run-time errors are errors that occur while the program is executing and occur due to a program action that is not allowed. Runtime errors are detected only when the particular line where the error occurs is executed and the program ceases operation (crashes). Examples of MATLAB run-time errors are formulas using an undefined variable and invoking a function with the incorrect name or arguments. MATLAB will terminate the program execution and display an error message including the line number where the error occurred in the Command Window for run-time errors. Figure 2a shows a program with a run-time error and Figure 2b shows the error message that MATLAB displays when the program is run.

```
num = 5;
result = num/den;
disp(result);
```

Figure 2a, MATLAB Program with Run-time Error

Run-time errors can be corrected by finding the line number the error occurred at and correcting the condition that caused the action that is not allowed. This often requires the use of a debugger so one can see the specific action and values that caused the error. The correction may require modifying the program earlier than where the error occurred. The run-

time error on line 2 of the program of Figure 2a is due to the variable `den` not being defined before it is used. In this example, the MATLAB error message is meaningful and indicates the true error in the program.

```
Undefined function or variable 'den'.

Error in Program2 (line 2)
result = num/den;
```

Figure 2b, MATLAB Error Message for Run-time Error in Program of Figure 2a

Logic errors are errors in the program that cause the program to operate incorrectly. The program executes but produces an incorrect or unintended result. The program does not operate the way it is supposed to. Logic errors are detected by the programmer; no error messages are generated by MATLAB. Examples of logic errors are using an incorrect formula for a calculation, an incorrect logic expression in a conditional structure, an infinite while loop due to an incorrect sentinel value, and an incorrect algorithm. Figure 3a shows a program with a logic error and Figure 3b shows the resulting incorrect program output.

```
angle = 45;
result = cos(angle);
disp(result);
```

Figure 3a, MATLAB Program with Logic Error

```
0.5253
```

Figure 3b, Output of Program with Logic Error of Figure 3a

Logic errors are typically the most difficult errors to detect and correct. Since no error messages are generated, one does not have the line number where the error occurred. One must first identify the source of the error and then correct the problem. The logic error in the program of Figure 3a is due to the variable `angle` being assigned an angle in degrees rather than radians (or using the `cos` instead of `cosd` function). The only way to recognize this is to notice that the result 0.5253 is not the cosine of 45 degrees which should be 0.7071.

Techniques for identifying the source of logic errors include:

- Tracing the program by hand (or using a calculator) line by line to determine the result of each line of the program and whether or not it is correct.
- Execute each line one by one in the Command Window to determine the result of each line of the program and whether or not it is correct.
- Adding extra display statements to print the values of important variables before and after operations and in critical portions of control statements (`if`, `for`, `while`). Use the printed values to determine if the values are what they should be after each

operation. Remember to remove the extra display statements (or comment them out) form the final version of the program.

- Using an interactive debugger and set breakpoints to monitor the values of variables in the workspace while the program runs. Most modern IDEs including MATLAB have built in interactive debuggers.

MATLAB Debugger

MATLAB provides debugging capabilities via breakpoints and debug commands. Breakpoints are executable lines in the script or function that MATLAB can stop execution at while running the program. Breakpoints are set by opening the program or function in the MATLAB editor and either clicking on the dash (-) just to the right of the line number of the statement in or by placing the curser on the line you want a breakpoint at and from the `Debug` menu select `Set/Clear Breakpoint`. Breakpoints show up as grey or red dots by the line depending on whether the program has been saved or not. Breakpoints are cleared in a similar manner.

When a program with breakpoints is run, MATLAB will enter debug mode (command line cursor changes from >> to K>>) and halt execution at the breakpoints. With execution halted, one can examine the current workspace and perform operations at the command line. When in debug mode, the `debug` menu has options for how the program will be executed. The `Step` command executes the current statement and stops at the next statement. The `Continue` command resumes execution of the file until completion or until another breakpoint is encountered. Breakpoints can be set, cleared, and disabled via the `Set/Clear Breakpoint`, `Clear Breakpoints in All Files`, and `Enable/Disable Breakpoint` commands. Debug mode can be exited by selecting Exit Debug Mode from the Debug menu.

Debugging Example

Consider the program of Figure 4 that performs a running concatenation of positive numbers read from the user and creates a string.

```
% read in and save values until negative number entered
values = [];
newValue = input('Enter value (- number to stop): ');
while (newValue >= 0)
    values = [values, newValue];
    newValue = input('Enter value (- number to stop): ');
end

disp('Values are: '),disp(values);
```

Figure 4, Running Concatenation of Positive Numbers Read from user

For the running concatenation, the initial result is an empty vector and each new valid value is concatenated to the end of the current vector of values. Running concatenations can be used to

create vectors of the same data type, for examples vectors of numbers or vectors of characters (strings).

Consider modifying the running concatenation of numbers of Figure 4 to read in and save characters read from the user. Using a sentinel as in the example of Figure 4 for numbers would be nice; what would be a good sentinel for reading in characters? Possibly the empty set as other characters even the special ones might be desirable to use (there is a special end of file (eof) character when performing file input and output). The program of Figure 5a is a first attempt at the running concatenation of characters.

```
% read in and save characters until terminating character read
characters = [];
newChar = input('Enter character(empty set to stop): ','s');
while (characters ~= [])
    character = [characters, newChar];
    newChar = input('Enter character(empty set to stop): ','s')
end

disp('Characters are: '),disp(characters);
```

Figure 5a, Attempt 1, Running Concatenation of Characters Read from User

A sample run of the program of Figure 5a is shown in Figure 5b.

```
Enter character(empty set to stop): a
Characters are:
>>
```

Figure 5b, Result of Running Program of Figure 5a

There are no syntax errors, but the program does not do what was hoped. One character (an a) is read in but no characters are saved and the program terminates before receiving the terminating indicator of the empty set.

The MATLAB debugger can be used to set breakpoints as shown in Figure 5c. Running the program with the breakpoints, MATLAB enters debug mode. The program is executed until the first breakpoint is reached (line 4). At the breakpoints one can examine workspace variables and see if the program is doing what was expected. To run to the next breakpoint, select the Step button in the Editor Toolstrip (or hit F10).

Figure 5d shows what appears in in the MATLAB Command Window while stepping through the program. As the program is stepped through, execution stops at line 4, then line 5, then pauses for the user input, but never gets to line 7 or line 8. The while loop in the program is never executed. This is because the condition (characters ~= []) is never true. This condition always evaluates to the empty set (not true or false) and MATLAB does not treat the empty set as true.

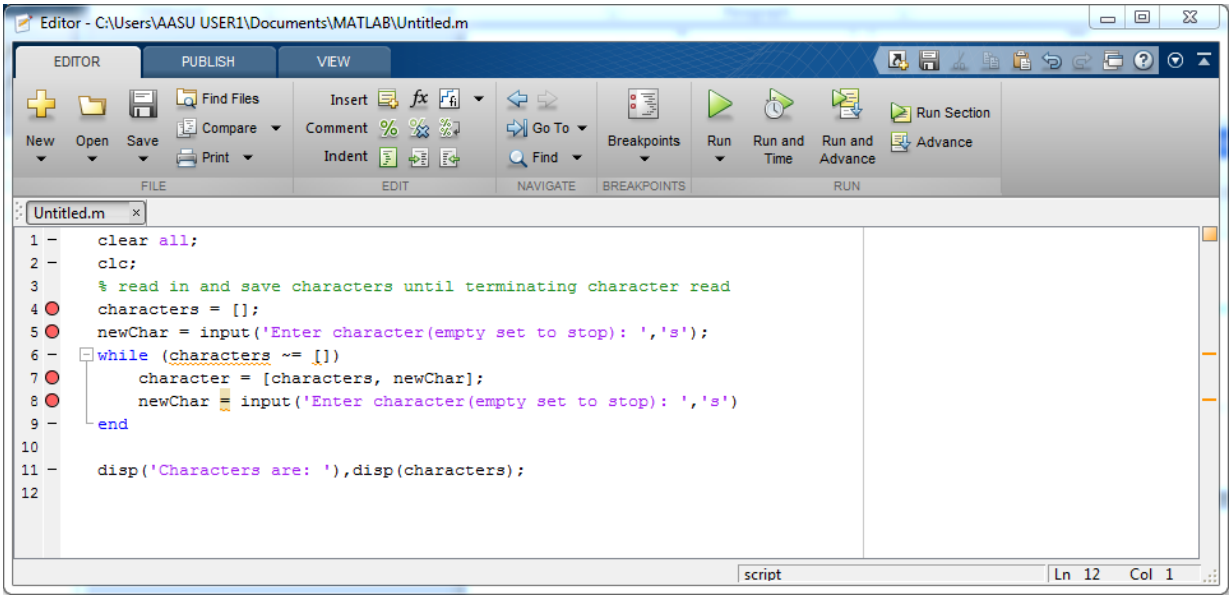

Figure 5c, Program with Breakpoints Set using MATLAB Editor

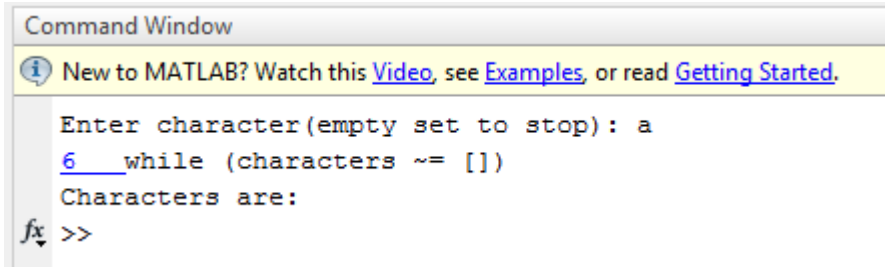

Figure 5d, MATLAB Command Window During Debugging

Figure 5e shows the next attempt with the terminating character changed to a ~.

```
% read in and save characters until terminating character read
characters = [];
newChar = input('Enter character(~ to stop): ','s');
while (characters ~= ~)
    character = [characters, newChar];
    newChar = input('Enter character(empty set to stop): ','s')
end

disp('Characters are: '),disp(characters);
```

Figure 5e, Attempt 2, Running Concatenation of Characters Read from User

The program of 5e does not execute as it has a syntax error. The error message is shown in Figure 5f. This error message is not very descriptive, but line 6 is the line `while (characters ~= ~)`. The syntax error is due to not enclosing the ~ character in single quotes to indicate that it is a character.

```
Error: File: Untitled.m Line: 6 Column: 23
Unbalanced or unexpected parenthesis or bracket.
```

Figure 5f, Error Message from Running Program of Figure 5e

The program of Figure 5e has two additional logic errors in addition to the syntax error. The logic expression in the while statement should compare `newChar` not `characters`, which is the result vector, to the terminating character. The concatenating operation uses the variable `character` instead of the variable `characters` on the left side of the assignment. This will mean that each iteration, the value of `newChar` and `characters` will be concatenated but the result is saved in the variable `character` which is then not used for the next concatenation, i.e. nothing is saved but the last character read in before the terminating character.

There is also a style issue, in that the input line inside the while loop body is not terminated with a semicolon and thus the user input will be echoed during the execution of the script. The fully corrected program is shown in Figure 5g.

```
% read in and save characters until terminating character read
characters = [];
newChar = input('Enter character(~ to stop): ','s');
while (newChar ~= '~')
    characters = [characters, newChar];
    newChar = input('Enter character(empty set to stop): ','s');
end

disp('Characters are: '),disp(characters);
```

Figure 5g, Final Version of Running Concatenation of Characters Read from User

Debugging Tips

Helpful tips for debugging:

- Syntax errors can be detected by running the program. MATLAB will identify the line number and type of error although the error message will not always clearly point to the error and some errors are actually caused by omitting statements prior to the line the error is indicated at (using a variable before it has been assigned a value).
- Logic errors can be identified using the MATLAB debugger and stepping through the program, examining the workspace variables at each step, and checking that the behavior is what it should be.
- Break the program/function up into sections. Implement and test each section separately. This helps to isolate errors to small sections of code.
- Make sure the sections are sequenced correctly. Sections that rely on operations in another section of the program must be placed after that section.

- When possible, test complex programs for simple cases with known solutions first. For example, when testing a program that approximates a cosine function using a Taylor series, test it first for known values of the cosine function (0, 45, 90 degrees).

Last modified Friday, August 09, 2013