MATLAB
User Defined Functions

Week 6

Loosely follows Chapter 7

User Defined Functions and Scripts

- MATLAB is built around commands and functions
  - Both can accept inputs and provide outputs
  - Functions do **not** save variables in the workspace
  - Functions **cannot** inherently see command line workspace variables
- Functions allow you to call a collection of commands multiple times
- It is necessary for you to understand how a function performs its task
- Code design is important (This is Software Engineering!)
  - Functions should be discrete and concise
  - Don’t let your function stick its hands in tasks it shouldn’t
  - Break it up when necessary

Concept of Function M-Files

- User defined functions are similar to MATLAB’s predefined functions
- They accept a number of inputs, perform some operation, and provide some number of outputs. (Any of these numbers could be zero)
- Just as with MATLAB functions, we need to know what they are supposed to do and if it does that correctly.
    - Verification - We just built a microwave, does it do what it should do?
    - Validation - We just built a microwave, is that what we were supposed to build?

Syntax for Functions

- Calling a user-defined function
  - my_function(...)
- Defining a user-defined function
  - function y = my_function(x)
  - x is a value passed in, commonly referred to as an argument
  - y is a variable which represents the output of the function
  - my_function is the name of the function
- Functions must be written in M-Files.
- Functions must share the same name as the file name.

Notes on Functions

- Naming convention for functions is the same as variables
    - camelCase
    - Underscore_Names
- As with variables, it's important for functions to have meaningful names
- Variables within a function should also have meaningful names
- Comments are, as always, extremely important.

More Notes on Functions

- Keep in mind whether or not you need element-wise calculations
- Remember to try help name_of_function to ensure you won’t be hiding one
- You will almost always want suppressed output in a function

Simple Function Examples

```
function volume = volumeSphere(radius)
	volume = (4/3)*pi*(radius^3);

function perimeter = perimeterSquare(side)
	perimeter = 4*side;

function root = squareRoot(x)
	root = x^(1/2);
```

Using Functions in the Command Window

- Accessed the same as MATLAB built-in functions
- The function file must be in the current directory
    - If not, you can supply a path or add it to your path/library
- To use a function in the command window

```
perimeterSquare(4)
ans =
       16
```

- Think about how this function could be modified for any polygon...

Functions with Multiple Inputs

- Functions can have multiple inputs
- Recall the constant acceleration equation of free fall
  - $x = x_o + v_o t + (1/2)at^2$

```
function x = displacement(xo, vo, a, t)
        x = xo + vo.*t + (1/2).*a.*t.^2
```

- NOTE: Obviously with free fall we *could* assume acceleration...

Functions with Multiple Outputs

- Acceleration of a particle and the force acting on it are as follows
  - $a = (v_2-v_1)/(t_2-t_1)$
  - $f = ma$
- A user function can be created to perform both calculations

```
function [a, F] = particleAcceleration(v2, v1, t2, t1, m)
        a = (v2-v1)/(t2-t1);
        f = m.*a;
```

Functions without outputs/inputs

- Functions can have no inputs

```
function massOfEarth = moe()
        massOfEarth = 5.976e24;
```

- Functions can have no outputs
  - tic is a simple example, there is no input, it simply starts a timer for use with toc
- Functions can have no inputs and no outputs
  - Very uncommon
  - Example might be a command that simply plots a constant function (plot is not an output)

Notes on Function Variables

- Sometimes referred to as parameters
- Variables created inside a function may **only** be accessed from within that function. (referred to as local variables)
- Local variables are deleted when a function finishes**
- The output will appear in the command window workspace
- Functions cannot access variables from the command window workspace**

Persistent Variables

- Persistent (static) variables retain their value at each function call

```
function findSum(inputvalue)
        persistent SUM_X

        if isempty(SUM_X)
                SUM_X = 0;
        end
        SUM_X = SUM_X + inputvalue;
```

Global Variables

- Variables from another workspace can be made available to the function

```
function h = falling(t)
        global GRAVITY
        h = 1/2*GRAVITY*t.^2;
```

And from the command window…

```
global GRAVITY
GRAVITY = 32;
y = falling([0:.1:5]');
```

The `type` Command

- type, followed by the name of a function M-File prints the contents of that M-File to the Command Window
  - A quick way of viewing the source of a function
- An alternative to using the help menu

- NOTE: help can be used to view comments in the header of a function

Local Functions

- Functions used by other functions within the same M-File
- Also known as Subfunctions
- Allow you to split tasks (keeping them discrete)
- Often used as “utility functions”
- Can be kept in the same M-File as a main function

Local Function Example

```
function [avg, med] = myStats(x)
    n = length(x);
    avg = myMean(x,n);
    med = myMedian(x,n);
end

function a = myMean(v,n)
% myMean Example of a local function.
    a = sum(v)/n;
end

function m = myMedian(v,n)
% myMedian Another example of a local function.

    w = sort(v);
    if rem(n,2) == 1
        m = w((n + 1)/2);
    else
        m = (w(n/2) + w(n/2 + 1))/2;
    end
end
```

Pass by Reference or Value

- Typically programming languages default to value
- Some offer both
- MATLAB is slightly unique “Copy on Write”
    - Essentially, pass by reference

Key Takeaways

- Local Functions can help break up unruly code
- Can be kept “hidden” in a file with other functions

Key Takeaways

- Function Concepts and Syntax
  - Naming
  - Validation and Verification
  - Discrete and Concise
- Types of functions
  - No args, single, or multiple args
  - No outputs, single, or multiple outputs
- Variable Scope
  - Parameters/local
  - Persistent
  - Global

MATLAB Testing

Week 6

Not covered in the book

Common Types of Tests

- Unit Test
- Integration Test
- Functional Test
- Acceptance Test

We will only focus on unit testing

What is a Unit Test?

- Crucial part of software development
- Used to test a portion (unit) of code
- Ensures code meets requirements
- Regression testing
- Unfortunately, like comments, it’s often overlooked

How to Define Tests

- Find expected truths
- Find edge cases
- Find expected untruths
- Find expected values (scalar, vector, array, string, numeric)

- Test Driven Development

Example

The m-file begins with the following

```
function tests = testMontlyLoanPayment
clc;
tests = functiontests(localfunctions);
end
```

Each test is written as a simple function

```
function testNumPaymentsString(testCase)
expectedPmt=[];
actualPmt=getMonthlyLoanPayment(10000, 0.04, "ten");
verifyEqual(testCase, actualPmt, expectedPmt);
end
```

Assertions

Some assertions you can make are:

- verifyEqual()
- verifyTrue()
- verifyFalse()
- verifyNotEqual()
- verifyFail()
- verifyError()

Search “Table of Verifications” to see the complete list

MATLAB
Errors and Pitfalls

Week 6

Loosely follows chapter 11

Types of Errors

As we’ve discussed before, errors come in many forms

● Syntax Errors	Typically seen right away in the editor
● Runtime Errors	Typically seen while running the program
● Logic Errors	Almost never caught by MATLAB, no error in MATLAB

Syntax Errors

- The equivalent of a spelling or grammar error in a written document
  - Identified by MATLAB with the jagged underline much like a Word document
- lasterr can be used to recall the last error message generated
- Examples

```
2*(1+3

disp(‘The answer is ‘ num2str(2))
```

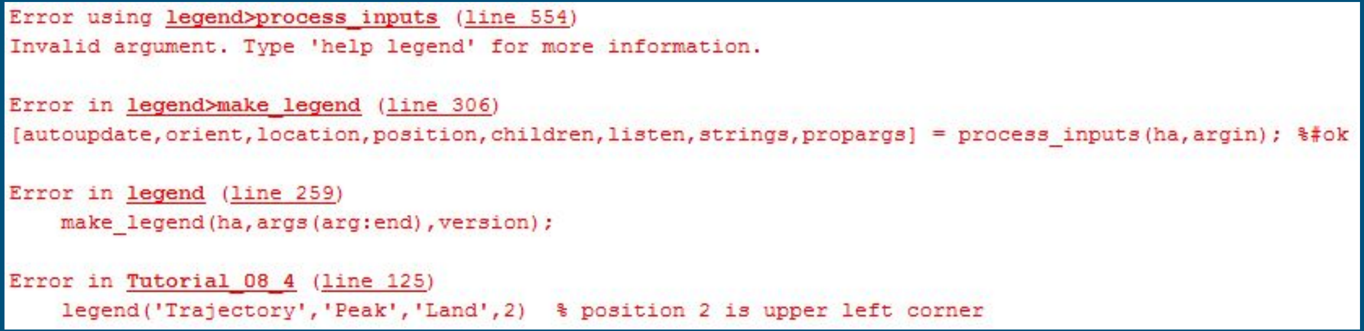
Name Hiding / Variable Scope

- Recall that pi=3 will overwrite MATLAB's internal pi variable
    - Naming a variable the same as a function will hide the function
    - When in doubt, use help nameOfVariable to see if that name already exists in MATLAB
- Scope refers to the visibility of a variable
    - A variable created inside of a loop or control structure may not be visible outside of that control structure.
    - A variable created inside of a function may not be visible outside of that function.

Runtime Errors

- These errors are typically not caught until you run your program.
- Example

```
legend('Trajectory','Peak','Land',2)  % position 2 is upper left corner
```



```
Error using legend>process_inputs (line 554)
Invalid argument. Type 'help legend' for more information.

Error in legend>make_legend (line 306)
[autoupdate,orient,location,position,children,listen,strings,propargs] = process_inputs(ha,argin); %#ok

Error in legend (line 259)
    make_legend(ha,args(arg:end),version);

Error in Tutorial_08_4 (line 125)
    legend('Trajectory','Peak','Land',2)  % position 2 is upper left corner
```

The fix:

legend('Trajectory','Peak','Land',**'Location','northwest'**)

Logic Errors

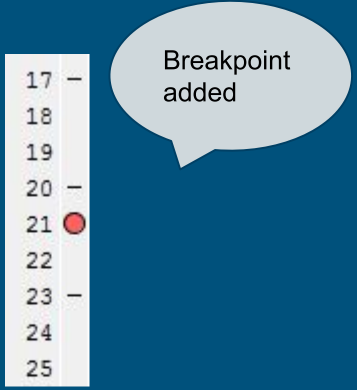
- Almost never found by MATLAB
- Programs will still run but will result in wrong answers
- Common problems
  - Order of magnitude
  - Order of operations
  - Code structure (multiple ifs)
  - Off by one
  - Rounding
  - Using = instead of ==

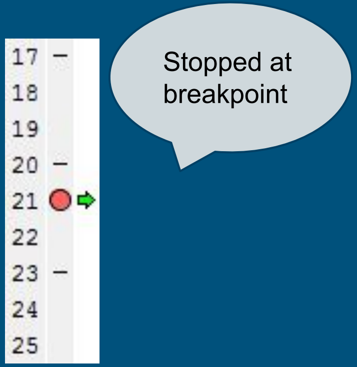
Error Handling

- Use the run drop down
  - Pause on Error
  - Pause on Warnings
  - Pause when NaN or Inf is returned
- Use the real-time debugger
- Try - Catch

Real-time Debugger

- Click the “gutter” anywhere you see a dash
    - A red “breakpoint” will appear
- When you run your program, MATLAB will stop at these breakpoints, at which point you can
    - View workspace variables
    - Enter commands in the command window
    - Click the “Continue” to progress the program
- To go even further, check out the step commands





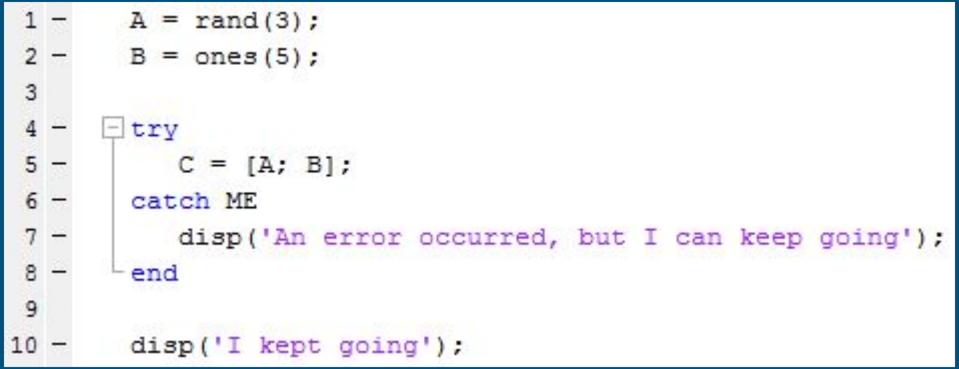
Try / Catch

- Code in the try block is executed and resulting errors are handled in the catch block.
- Only works for “caught” runtime errors (ex. not divide by zero)
- Standard format

```
try
   statements
catch exception
   statements
end
```

Simple Try / Catch Example

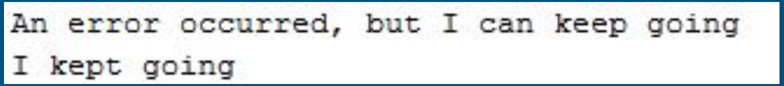
- An example for concatenating inconsistent matrices



```
A = rand(3);
B = ones(5);

try
    C = [A; B];
catch ME
    disp('An error occurred, but I can keep going');
end

disp('I kept going');
```

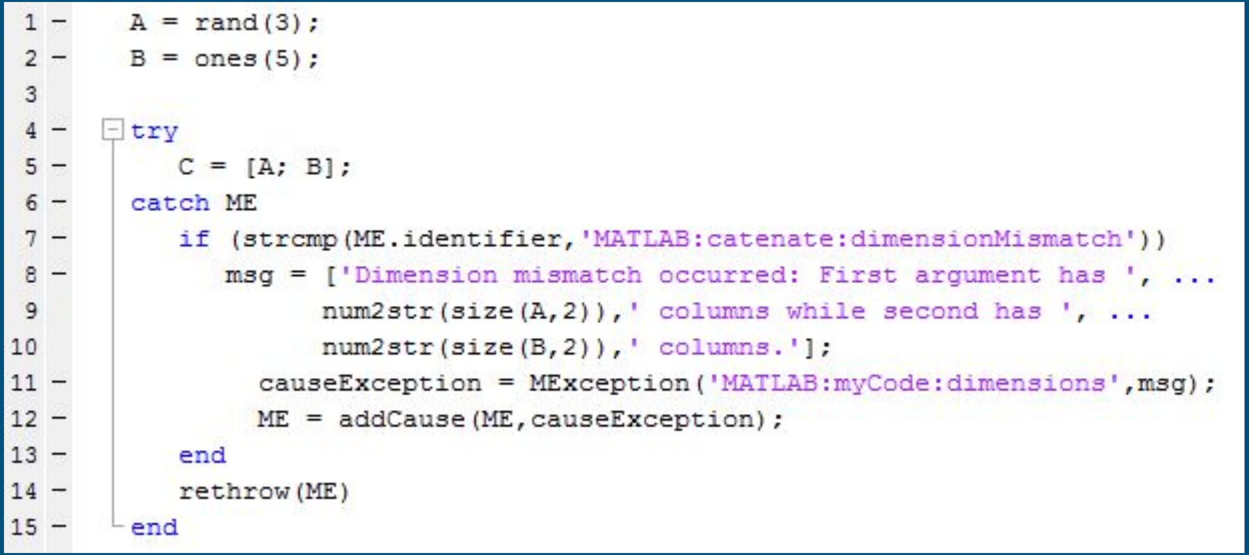


```
An error occurred, but I can keep going
I kept going
```

```
>> c=[A;B]
Error using vertcat
Dimensions of arrays being concatenated
are not consistent.
```

Complex Try / Catch Example

- Messages can be modified and “rethrown”

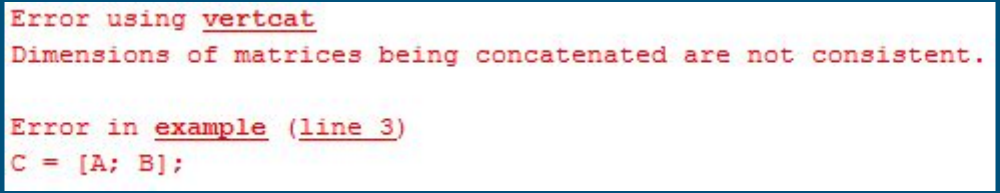


```
A = rand(3);
B = ones(5);

try
    C = [A; B];
catch ME
    if (strcmp(ME.identifier,'MATLAB:catenate:dimensionMismatch'))
        msg = ['Dimension mismatch occurred: First argument has ', ...
                num2str(size(A,2)),' columns while second has ', ...
                num2str(size(B,2)),' columns.'];
            causeException = MException('MATLAB:myCode:dimensions',msg);
            ME = addCause(ME,causeException);
    end
    rethrow(ME)
end
```

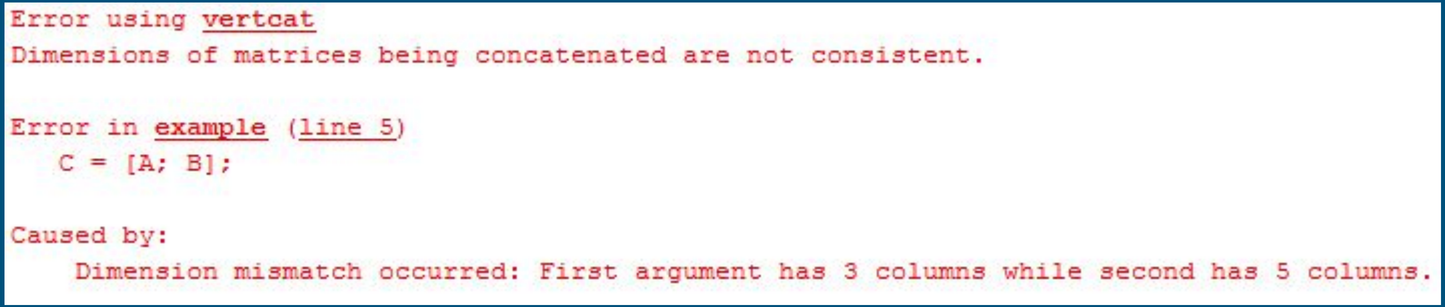
Try / Catch Example Output

- Before and after output of previous example



```
Error using vertcat
Dimensions of matrices being concatenated are not consistent.

Error in example (line 3)
C = [A; B];
```



```
Error using vertcat
Dimensions of matrices being concatenated are not consistent.

Error in example (line 5)
   C = [A; B];

Caused by:
    Dimension mismatch occurred: First argument has 3 columns while second has 5 columns.
```

Key Takeaways

- Three types of errors
  - Syntax
  - Runtime
  - Logic
- Ways of handling/debugging errors
  - Run drop down
  - Debugger
  - try/catch