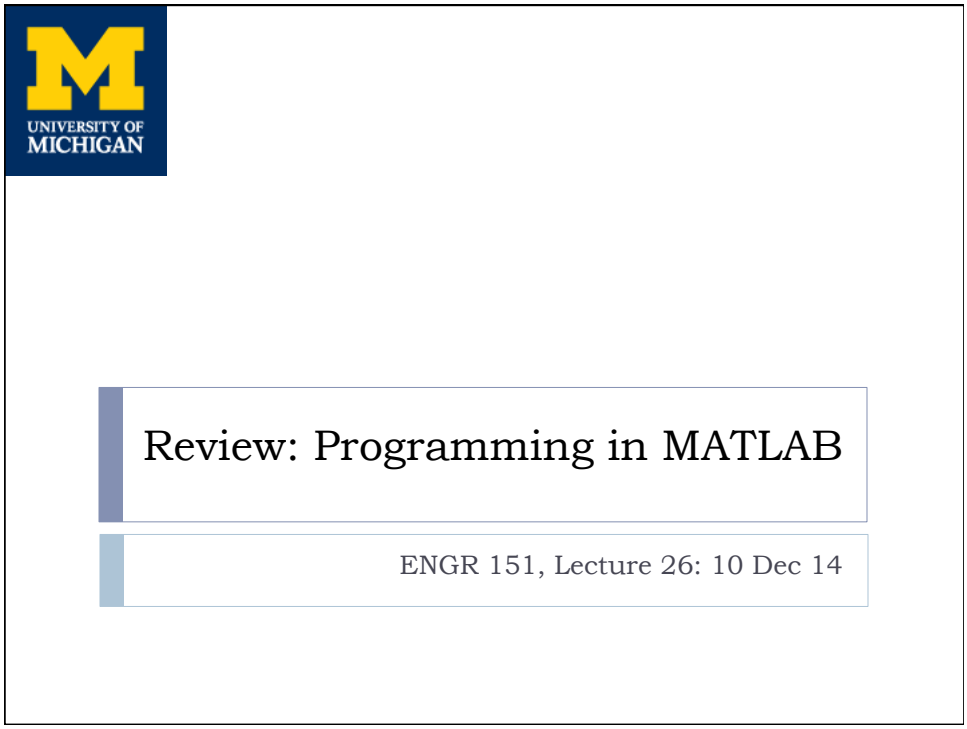


Announcements

- Project 8 due tonight 10 Dec 11pm
- Final exam: Wed 17 Dec 4pm
  - Style similar to midterms
  - Focuses on MATLAB section of course
    - General programming concepts from throughout
    - All programming in MATLAB
  - Closed book and notes (no electronics)
- Course evaluations

Room assignments, by last name:
- A-G: 1311 EECS
- H-K: 1008 EECS
- L-Y: 1500 EECS
- Z: 1311 EECS

Outline of Lectures Since Exam 2

17. Intro to MATLAB
18. Vectors and Matrices
19. Functions and I/O
20. Subarrays and Vectorization
21. Selection and Iteration
22. MATLAB Programs
23. Data Visualization (Plotting)
24. Fractals and the Mandelbrot Set
25. Mathematical Facilities

C++ vs. MATLAB

- Compiled
- Fast
- Strongly typed
- Predefined libraries
- Variety of data structures
- Graphing external
- Open standard

- Interpreted
- Slower
- Weakly typed
- More math libraries
- Matrix based
- Integrated graphing
- Proprietary (MathWorks)

Array as Fundamental Data Type

- To MATLAB all data is some kind of array.
- Scalars: arrays with one element (zero dimensions)
  - Elements typically a floating-point or character type
- Vectors: sequence of scalars (one dimension: row or column)
- Matrices: two or more dimensions (rows, columns, …)

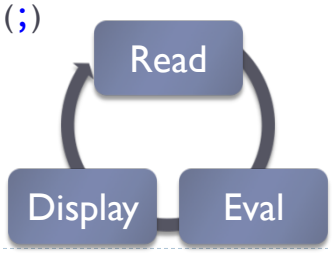
MATLAB Workspace

- "main" scope for MATLAB functions and variable names
- Names introduced (e.g., initial assignment) and referenced from command window or script file
- Lifetime of names: from introduction until `clear` command
- Use `who` command to display active variables in workspace

Read/Eval/Display Loop

- The MATLAB interpreter processes input (from keyboard or scripts) as follows:

1. read the next statement
   - a command, assignment, or expression
2. execute the statement, evaluating expressions as needed
3. if an assignment or expression stmt, display the result
   - unless statement ends in semicolon (;)



Scalar Operations

- all binary, infix:

Addition	a + b
Subtraction	a – b
Multiplication	a * b
Division	a / b
Left Division	a \ b
Exponentiation	a ^ b

Creating Matrices

- Matrices are specified in row order, separated by commas or spaces.
- Rows can be separated by semicolons (;) or new lines.

```
a = [1.0, 3.0, 5.0; 2.0, 4.0, 6.0]
b = [1.0  3.0  5.0
     2.0  4.0  6.0 ]
```

Accessing Array Elements

- Parentheses denote index operator ( )
  - unlike C++, MATLAB indices start at 1
  - Example:

```
b = [1.0  3.0  5.0
     2.0  4.0  6.0 ]
b(1,3) ➔ 5.0
b(2,2) = 12 ➔
              b = [1.0  3.0  5.0
                   2.0  12.0  6.0 ]
```

Vectors with Constant Increments

- The colon operator provides a shortcut for vectors with regular spacing

  first : increment : last

  ```
  x = 1:2:8;
  ```

  is equivalent to

  ```
  x = [ 1  3  5  7 ];
  ```

- If increment is omitted, treated as 1
  - for example `1:4` is the vector `[1 2 3 4]`

Vectors in Index Expressions

```
a = [0 1 2 3 4 5 6 7 8 9]

a(1) → 0
a(8) → 7
a(1:5) → [0 1 2 3 4]
a(4:10) → [3 4 5 6 7 8 9]
a(3:3:10) → [2 5 8]
```

Array Operations

- Array or Matrix operations (no difference)
  - Addition `a + b`
  - Subtraction `a - b`
- Array (element-by-element) operations
  - Multiplication `a .* b`
  - Right Division `a ./ b`
  - Left Division `a .\ b`
  - Exponentiation `a .^ b`

Matrix Operations

- Multiplication `a * b`
- Right Division `a / b`
- Left Division `a \ b`
- Exponentiation `a ^ b`

- Multiplication is standard matrix multiplication.
  - # columns of `a` must equal # rows of `b`
- Division (conceptually) inverts the denominator matrix and multiplies by the numerator.
- Exponentiation: `a` must be square, `b` a scalar

char Function

- converts ASCII codes (integer between 0 and 127) into characters
- applies elt-by-elt to an array
- Examples:
  - `char(65:70)` → ABCDEF
  - `char([65:70]+1)` → BCDEFG
  - `3+'a'` → 100
  - `char(3+'a')` → d

MATLAB Type Conversions

- Sometimes automatic (coercion):
  - `'a' + 3` → 100
  - `'a' + 'b'` → 195
  - `90 + true` → 91
  - `~[40 0 -3]` → [0 1 0] (logicals)
- Can also use explicit conversion (cast) functions:
  - `logical([40 0])` → [1 0] (logicals)
  - `char(98)` → b
  - `double('b')` → 98
  - `uint8([38 -5 2.8 666])` → [38 0 3 255]

Array Element Types

- In an array, all elements must be same type
- Element type determined on array initialization
  - `v = 1:5` a vector of numbers
  - `w = 'abcde'` a vector of characters
- Assigning different types to elements will cause a conversion
  - `v(3) = 'x'` → v = [1 2 120 4 5]
  - `w(3) = 120` → w = 'abxde'
- To assign an element to chosen type, initialize that way or perform explicit conversion
  - `v(1:numElts) = 'x';`
  - `v = ones(1,numElts); v = char(v * 120);`
  - `v = char('x' * ones(1,numElts));`

ENGR 151 Poll

- How should we change the language mix in this course?

A. Cover MATLAB first, C++ second
B. Keep C++ first, but switch to MATLAB earlier
C. Interleave C++ and MATLAB throughout course
D. No change

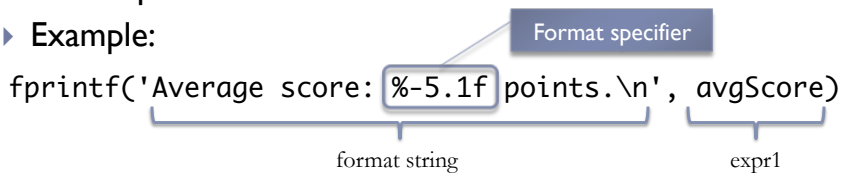
Script Files

- A list of MATLAB commands, saved in a file
- Use extension .m
  - hence called "M-files"
  - can run scriptname.m by typing scriptname as command
- Running script file equivalent to typing in commands, in sequence

Formatted Output

- The procedure `fprintf` provides fine control of how data is formatted for output to display or file
- General form for display output:

  `fprintf(`*format_string*`,` *expr1* `,` *expr2*`, ...)`
  - displays expression values as specified in the format string.
- Format string: literal string, with specifications to format each expression
- Example:



Function Parameters and Return Values

```
function [outputs] = function_name (inputs)
% H1 line: function name and short description
% comments documenting function behavior
%

Body
```

- inputs: comma-separated list of argument variable names
  - variables passed by value
- outputs: list (comma- or space-separated) of output variable names
  - variables must be assigned in function body
  - on termination these values returned
- All names in function have local scope
  - Function cannot reference or modify variables in workspace scope

Subfunctions

- A function (.m) file may contain multiple function definitions
  - The first is the primary function and should correspond to the name of the file
  - Subsequent functions are called subfunctions
- Primary and subfunctions may call each other
  - regardless of order of appearance in file
  - only primary function may be called from main scope
- Each (sub)function has its own scope for local variables

General File I/O

- MATLAB manages file input/output through file IDs
  - Numbers for tracking files used by a program
- To open a file:

```
file_id_var = fopen(filename);
file_id_var = fopen(filename,permission);
```

- Example:

```
fid = fopen('Myfile.txt', 'r');
```

returns -1 if file open fails

- To close this file:

```
close(fid)
```

Relational Operators in MATLAB

==	equal to
~=	not equal to
>	greater than
>=	greater than or equal to
<	less than
<=	less than or equal to

(exact equality, beware of rounding errors)

conditional expressions take a logical value:
1 for true, 0 for false

Relational Operators on Arrays

- Apply elt-by-elt between arrays

$$\begin{pmatrix} 1 & 5 \\ 3 & -1 \end{pmatrix} > \begin{pmatrix} 3 & 1 \\ -1 & 2 \end{pmatrix} \rightarrow \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$$

- Also apply elt-by-elt between a scalar and an array

$$5 > \begin{pmatrix} 8 & 2 \\ 10 & -1 \end{pmatrix} \rightarrow \begin{pmatrix} 0 & 1 \\ 0 & 1 \end{pmatrix}$$

Logical Expressions

operator	function	meaning
&	and	*and*
|	or	*or*
~	not	*not*
	xor	*exclusive or*

p	q	~p not(p)	p & q and(p,q)	p | q or(p,q)	xor(p,q)
false	false	1	0	0	0
false	true	1	0	1	1
true	false	0	0	1	1
true	true	0	1	1	0

Vectors as Indices

- Specifying a vector index selects a subarray with elements corresponding to the indices in the vector
- If v is a vector with n elts, then x(v) is equivalent to:

```
[x(v(1)), x(v(2)), …, x(v(n))]
```

- For example, given:

```
x = [ 1  7  4  -1  13 ]
x(3)        →   4
x([1 3 5]) →   [1 4 13]
```

Vectors as Matrix Indices

- Can also use vector indices on matrices
- For example, given:

```
x = [ 1  7  4  -1  13 ]
M = [ x; x+1; x+2 ]
M(1,[1 3 5])       → [1 4 13]
M([1 2],[1 3 5])   → ?
                     [1 4 13
                      2 5 14]
```

Assigning to Subarrays

```
arr = [ 1:4; 5:8; 9:12 ]
arr([1 3],1:2) = [ -4 -3; -2 -1 ]
arr(:,[2 4]) = 0
```

$$\begin{pmatrix} -4 & 0 & 3 & 0 \\ 5 & 0 & 7 & 0 \\ -2 & 0 & 11 & 0 \end{pmatrix}$$

Logical Arrays

- The result of applying a relational or logical operator to an array is a logical array (array of boolean values)

```
x = [ 1 7 10; 9 0 2 ];
x > 5        → [ 0 1 1; 1 0 0 ]
x > 5 | ~x   → [ 0 1 1; 1 1 0 ]
```

Addressing with Logical Arrays

- Logical array as index: selects elts corresponding to "true" values in the logical array
- For example:

```
x = [ 3 7 10 9 4 2 ]
x(x > 5)          →  [ 7 10 9 ]
x(mod(x,2)==0)    →  [ 10 4 2 ]
x(mod(x,2))       →  error
x([1 1 0 1 0 0])  →  error
```

Select based on condition rather than position

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

Selective Operations

```
x = [ 1 7 10; 9 4 2 ]
w = x > 5;
```

- Subtract 2 from all values of x greater than 5

```
x(w) = x(w)-2;
```

same selection on left and right side

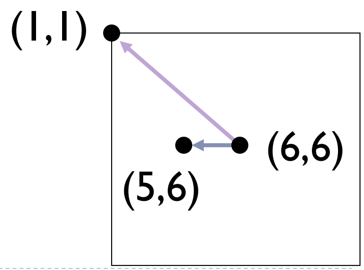
$$x = \begin{pmatrix} 1 & 7 & 10 \\ 9 & 4 & 2 \end{pmatrix}$$

$$w = \begin{pmatrix} 0 & 1 & 1 \\ 1 & 0 & 0 \end{pmatrix}$$

Vectorized Programming

- Write a short program with no loops that:
  - creates an 11 x 11 matrix,
  - where cell (r,c) contains the distance of point (r,c) from the center location (6,6),
  - unless the distance is less than 2, in which case it should be replaced by 2.

e.g., M(1,1) = sqrt(25+25);
M(5,6) = 1 → 2;



(programming)

```
col = ones(11,1) * (-5:5);
row = col';
M = sqrt(row.^2 + col.^2);
```

-5	-4	-3	-2	-1	0	1	2	3	4	5

col

-5	-4	-3	-2	-1	0	1	2	3	4	5
-5	-4	-3	-2	-1	0	1	2	3	4	5
-5	-4	-3	-2	-1	0	1	2	3	4	5
-5	-4	-3	-2	-1	0	1	2	3	4	5
-5	-4	-3	-2	-1	0	1	2	3	4	5
-5	-4	-3	-2	-1	0	1	2	3	4	5
-5	-4	-3	-2	-1	0	1	2	3	4	5
-5	-4	-3	-2	-1	0	1	2	3	4	5
-5	-4	-3	-2	-1	0	1	2	3	4	5
-5	-4	-3	-2	-1	0	1	2	3	4	5
-5	-4	-3	-2	-1	0	1	2	3	4	5

row

-5	-5	-5	-5	-5	-5	-5	-5	-5	-5	-5
-4	-4	-4	-4	-4	-4	-4	-4	-4	-4	-4
-3	-3	-3	-3	-3	-3	-3	-3	-3	-3	-3
-2	-2	-2	-2	-2	-2	-2	-2	-2	-2	-2
-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
0	0	0	0	0	0	0	0	0	0	0
1	1	1	1	1	1	1	1	1	1	1
2	2	2	2	2	2	2	2	2	2	2
3	3	3	3	3	3	3	3	3	3	3
4	4	4	4	4	4	4	4	4	4	4
5	5	5	5	5	5	5	5	5	5	5

(programming)

```
col = ones(11,1) * (-5:5);
row = col';
M = sqrt(row.^2 + col.^2);
```

```
M =
 [ 7.0711  6.4031  5.8310  5.3852  5.0990  5.0000  5.0990  5.3852  5.8310  6.4031  7.0711
   6.4031  5.6569  5.0000  4.4721  4.1231  4.0000  4.1231  4.4721  5.0000  5.6569  6.4031
   5.8310  5.0000  4.2426  3.6056  3.1623  3.0000  3.1623  3.6056  4.2426  5.0000  5.8310
   5.3852  4.4721  3.6056  2.8284  2.2361  2.0000  2.2361  2.8284  3.6056  4.4721  5.3852
   5.0990  4.1231  3.1623  2.2361  1.4142  1.0000  1.4142  2.2361  3.1623  4.1231  5.0990
   5.0000  4.0000  3.0000  2.0000  1.0000       0  1.0000  2.0000  3.0000  4.0000  5.0000
   5.0990  4.1231  3.1623  2.2361  1.4142  1.0000  1.4142  2.2361  3.1623  4.1231  5.0990
   5.3852  4.4721  3.6056  2.8284  2.2361  2.0000  2.2361  2.8284  3.6056  4.4721  5.3852
   5.8310  5.0000  4.2426  3.6056  3.1623  3.0000  3.1623  3.6056  4.2426  5.0000  5.8310
   6.4031  5.6569  5.0000  4.4721  4.1231  4.0000  4.1231  4.4721  5.0000  5.6569  6.4031
   7.0711  6.4031  5.8310  5.3852  5.0990  5.0000  5.0990  5.3852  5.8310  6.4031  7.0711 ]
```

(programming)

```
col = ones(11,1) * (-5:5);
row = col';
M = sqrt(row.^2 + col.^2);
low = M < 2;
```

```
low =
  [ 0  0  0  0  0  0  0  0  0  0  0
    0  0  0  0  0  0  0  0  0  0  0
    0  0  0  0  0  0  0  0  0  0  0
    0  0  0  0  0  0  0  0  0  0  0
    0  0  0  0  1  1  1  0  0  0  0
    0  0  0  0  1  1  1  0  0  0  0
    0  0  0  0  1  1  1  0  0  0  0
    0  0  0  0  0  0  0  0  0  0  0
    0  0  0  0  0  0  0  0  0  0  0
    0  0  0  0  0  0  0  0  0  0  0
    0  0  0  0  0  0  0  0  0  0  0 ]
```

(programming)

```
col = ones(11,1) * (-5:5);
row = col';
M = sqrt(row.^2 + col.^2);
low = M < 2;
M(low) = 2;
```

```
M =
 [ 7.0711  6.4031  5.8310  5.3852  5.0990  5.0000  5.0990  5.3852  5.8310  6.4031  7.0711
   6.4031  5.6569  5.0000  4.4721  4.1231  4.0000  4.1231  4.4721  5.0000  5.6569  6.4031
   5.8310  5.0000  4.2426  3.6056  3.1623  3.0000  3.1623  3.6056  4.2426  5.0000  5.8310
   5.3852  4.4721  3.6056  2.8284  2.2361  2.0000  2.2361  2.8284  3.6056  4.4721  5.3852
   5.0990  4.1231  3.1623  2.2361  2.0000  2.0000  2.0000  2.2361  3.1623  4.1231  5.0990
   5.0000  4.0000  3.0000  2.0000  2.0000  2.0000  2.0000  2.0000  3.0000  4.0000  5.0000
   5.0990  4.1231  3.1623  2.2361  2.0000  2.0000  2.0000  2.2361  3.1623  4.1231  5.0990
   5.3852  4.4721  3.6056  2.8284  2.2361  2.0000  2.2361  2.8284  3.6056  4.4721  5.3852
   5.8310  5.0000  4.2426  3.6056  3.1623  3.0000  3.1623  3.6056  4.2426  5.0000  5.8310
   6.4031  5.6569  5.0000  4.4721  4.1231  4.0000  4.1231  4.4721  5.0000  5.6569  6.4031
   7.0711  6.4031  5.8310  5.3852  5.0990  5.0000  5.0990  5.3852  5.8310  6.4031  7.0711 ]
```

Setting up a Mesh

- Key trick: Set up two matrices X and Y to represent x coordinates and y coordinates, respectively
- The process of setting up a 2D grid is common when dealing with functions on the 2D plane.

$$X = \begin{pmatrix} -2 & -1 & 0 & 1 & 2 \\ -2 & -1 & 0 & 1 & 2 \\ -2 & -1 & 0 & 1 & 2 \\ -2 & -1 & 0 & 1 & 2 \\ -2 & -1 & 0 & 1 & 2 \end{pmatrix} \quad Y = \begin{pmatrix} -2 & -2 & -2 & -2 & -2 \\ -1 & -1 & -1 & -1 & -1 \\ 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 \\ 2 & 2 & 2 & 2 & 2 \end{pmatrix}$$

col row

meshgrid

- MATLAB provides a built-in function to do this
- `[X, Y] = meshgrid(xdom, ydom)`
  - X is the matrix of x coordinates
  - Y is the matrix of y coordinates
  - xdom is a vector defining the domain of x
  - ydom is a vector defining the domain of y
- Once the grid is set up then computing an arbitrary function as a function of the x and y value is easy

```
z = (X.^2 + Y.^2);
```

Simplifying the Last Program

```
[col, row] = meshgrid(-5:5, -5:5);
M = sqrt(row.^2 + col.^2);
M(M < 2) = 2;
```

ENGR 151 Poll

- How helpful did you find the zyBooks?

A. Very helpful
B. Generally helpful, but considerable room for improvement
C. Mixed: usefulness varied a lot across sections
D. Only slightly helpful
E. Not at all helpful (waste of time)

Two Key Elements of Programming

- Flow of Control
  - Sequence / Procedure
  - Selection
  - Iteration
- Data Representation
  - How do we structure the data that we are acting on?

Selection in MATLAB

- Two primary selection constructs:
  - `if`
  - `switch`
- Both enable selection of computational steps based on condition expressions

Selection using `if`

- General form of the `if` construct in MATLAB:

```
if cond_exp
    stmt*
elseif cond_exp
    stmt*
...
else
    stmt*
end
```

body comprises zero or more statements

optionally may include any number of `elseif` clauses

optional stmt(s) executed iff all previous conditions false

Nested `if`

- The body of an `if` may itself include `if` statements

```
d = b^2 -4*a*c;
if a ~= 0
  if d < 0
     disp('complex roots');
  else
     x1 = (-b + sqrt(d))/(2*a);
     x2 = (-b - sqrt(d))/(2*a);
  end
end
```

Switch

```
switch expr
  case case_expr
      stmt*
  case case_expr
      stmt*
  …
  otherwise
      stmt*
  end
```

Scalar or string expression

Execute stmt(s) after case with label corresponding to switch expr

body for each case comprises zero or more statements

optional
stmt(s) executed iff all previous conditions false

While in MATLAB

```
while cond_expr
      stmt*
end
```

For in MATLAB

```
for counter = vector_expr
      stmt*
end
```

Execute the loop body once for each elt of the vector, with counter bound to that elt value

For Example

```
val = 1;
for n = 1:10
  val = val * n;
end
```

val = 10!

```
val = 1;
for n = 1:2:10
  val = val * n;
end
```

val = 1 × 3 × 5 × 7 × 9

For with Vector Counters

```
for counter = array_expr
      stmt*
end
```

- The counter can also be assigned to an array with more than one dimension
- In this case the counter takes on the value of an entire column one column at a time

Iteration and Vectorization

- Explicit iteration constructs:
  - `while`
  - `for`
- Elt-by-elt operations on arrays can also effectively accomplish iterative tasks without explicit iteration programming constructs
- Vectorizing calculation in this way is generally
  - more efficient
  - more natural (once you get used to it)

than the corresponding for or while construct

Nested Function Definitions

- can also define a function *within* another function definition

```
function [a, b] = SomeFunction (x, y)
…
    function c = NestOne(z)
    …
    end
…
end
```

`NestOne` is a nested function of `SomeFunction`
`SomeFunction` is the nesting function of `NestOne`

Multilevel Function Nesting

```
function y = A (a1,a2)
…
    function z = B(b1,b2)
    …
        function w = C(c1,c2)
        …
        end
    end
    function u = D(d1,d2)
    …
        function h = E(e1,e2)
        …
        end
    end
…
end
```

variable scope

- Which fns have access to a1, a2, and y?
  - all of them
- other formal parameters?
- other variables (e.g., if B introduces b3)?
  - Accessible to A and C, and also D and E but only if A refers to b3.

Example: Newton's Method

- Solve for *x*:

$$f(x)=0$$

```
double squareRoot(double s, double eps)
{
  double guess = 1.0;
  double residual = abs(guess * guess - s);
  while ( residual > eps ) {
    guess = newGuess(guess,s);
    residual = abs(guess * guess - s);
    }
  return guess;
}
```

- Approach:
  - Guess a value, g
  - If f(g) close enough to zero, return g
  - Otherwise, generate an improved guess, repeat…
- Guess improvement formula:

$$g \leftarrow g - \frac{f(g)}{f'(g)}$$

Newton's Method in MATLAB

```
function root = newton(guess)
% newton help line goes here

  function y = f(x)
    y = x^3 + x - 3;
  end

  function y = df(x)
    y = 3*x^2 + 1;
  end
…
```

(to be continued)

Nested "helper" functions, defining equation to be solved and its derivative.

adapted from Hahn & Valentine, *Essential MATLAB*

Newton's Method (continued)

```
…
steps = 0;
myrel = 1;
re = 1e-8;    %  relative error threshold

while (myrel > re) & (steps < 20)
  oldguess = guess;
  guess = guess - f(guess)/df(guess);
  steps = steps + 1;
  fprintf('guess: %8.4g; f(guess): %8.4g\n',...
           guess, f(guess));
  myrel = abs((guess-oldguess)/guess);
end

end
```

Function Handles

- Create a function handle by prepending @ before a function name.
- For example, suppose we have defined:

```
function y = myfunc(x)
  y = x^3 + x - 3;
```

  - then `@myfunc` is a function handle, and:
  - `feval(@myfunc,expr)` behaves just like `myfunc(expr)`
  - if we assign `myf2 = @myfunc` , then `myf2(expr)` also behaves just like `myfunc(expr)`

Newton's Method with Function Inputs

```
function root = newton2(guess,f,df)
% newton2 help line goes here

steps = 0; myrel = 1;
re = 1e-8;    %  relative error threshold

while (myrel > re) & (steps < 20)
  oldguess = guess;
  guess = guess - f(guess)/df(guess);
  steps = steps + 1;
  fprintf('guess: %8.4g; f(guess): %8.4g\n',...
           guess, f(guess));
  myrel = abs((guess-oldguess)/guess);
end

end
```

Function that takes function inputs called a higher-order function, or sometimes a "function function"

Anonymous Functions

- Function objects *without* names
- Form:

```
@ (paramList) expr
```

  - paramList: comma-separated list of arguments (just as in function definitions)
  - expr: a MATLAB expression
- To associate with a function handle variable:

```
handlevar = @ (paramList) expr
```

Using Anonymous Functions

- Example:

```
f = @(x) x.^3 + x - 3
df = @(x) 3*x.^2 + 1
```

- Use in expressions:

```
plot(1:20,f(1:20))
```

- Pass into functions:

```
newton2(1.0,f,df)
```

- Assigning handle to name not actually needed:

```
newton2(1.0, @(x) x.^3 + x - 3, ...
             @(x) 3*x.^2 + 1 )
```

Anonymous Functions: Scope

@ (paramList) expr

- Variables in paramList have scope local to anonymous function
- Body expr may also refer to any variables or functions available in scope where anonymous function defined
  - External references evaluated at *definition* time

Try this in MATLAB

```
x = 999;
z = -111;
myFunc = @(z) sqrt(x+z);
y = myFunc(601)
x = 24;
y2 = myFunc(601)
myFunc2 = @(y) 2*myFunc(y);
y3 = myFunc2(601)
myFunc = @(x) x^2;
y4 = myFunc2(601)
```

Plotting Data in MATLAB

- `plot`: general plotting function
  - Many forms, depending on number and content of arguments
  - Generates a new figure (e.g., for a graph) in the Figure window
    - Opens new window if necessary
- Basic form: `plot(x, y)`
  - x is a vector of x values
  - y is a vector of y values

Plot draws corresponding (x,y) points on graph, connecting adjacent points in vectors by lines

```
x = 1:0.1:10;
y = x .^ 2 - 10 * x + 15;
plot(x, y);
```

Multiple Plots on Same Figure Graph

- Provide additional pairs of x/y vectors as `plot` arguments

```
x = 0:0.1:10;
y1 = x .^ 2 – 10 * x + 15;
y2 = - x .^ 2 + 10 * x;
plot(x, y1, x, y2);
```

Time-Dependent Data

- Suppose the location of an object at time *t* is described by the equations:

$$x(t) = e^{-0.2t}\cos(2t)$$

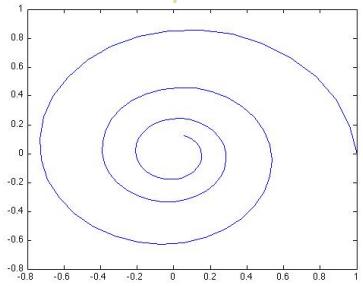
$$y(t) = e^{-0.2t}\sin(2t)$$

- We know how to plot this in two dimensions

```
t = 0:0.1:10;
x = exp(-0.2*t) .* cos(2*t);
y = exp(-0.2*t) .* sin(2*t);
plot(x,y);
```

Plot of (x,y) over Time

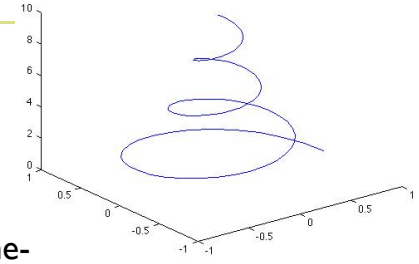
```
t = 0:0.1:10;
x = exp(-0.2*t) .* cos(2*t);
y = exp(-0.2*t) .* sin(2*t);
plot(x,y);
```



- Useful, but does not really show us where object is at particular times

A 3-Dimensional Plot

```
t = 0:0.1:10;
x = exp(-0.2*t) .* cos(2*t);
y = exp(-0.2*t) .* sin(2*t);
plot3(x,y,t);
```



- Now we can really see the time-space trajectory of the object

Visualizing Three-Dimensional Surfaces

- What does the following function look like?

$$z = \left(x^2 - y^2\right)e^{-\left(x^2+y^2\right)}$$

Cannot use line plot, because there is a z for every (x,y) pair

- Set up a mesh of (x,y) coordinates:
  `[x,y]= meshgrid(xstart:xinc:xend, ystart:yinc:yend)`

- Once the grid is set up, can compute z values:

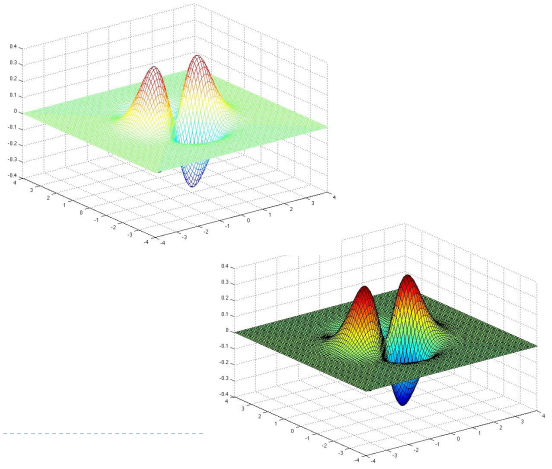
```
z = (x.^2 - y.^2) .* exp(-(x.^2 + y.^2));
```

Mesh and Surface Plots

```
[x, y] = meshgrid(-4:0.1:4);
z = (x.^2-y.^2).*exp(-(x.^2+y.^2));

mesh(x, y, z);

surf(x, y, z);
```



Mandelbrot Computation

```
function res = mandelbrotIterate (c, niters)

z = zeros(size(c));
res = ones(size(c));
active = (z==0);

for m = 1:niters
  z(active) = z(active).^2 + c(active);
  newactive = abs(z) <= 2;
  res(active & ~newactive) = m/niters;
  active = newactive;
end
```

drawMandelbrot (cont.)

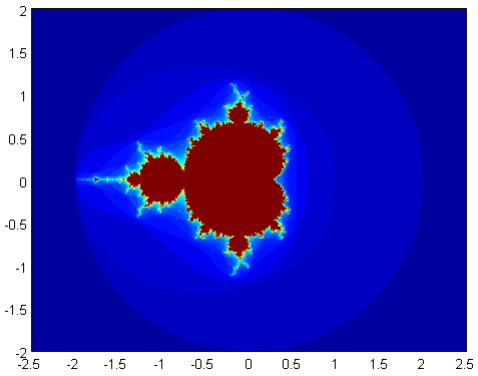
```
function drawMandelbrot(range,resolution,niter)
% drawMandelbrot generates a Mandelbrot set plot

xvals = linspace(range(1),range(2),resolution);
yvals = linspace(range(3),range(4),resolution);
[re, im] = meshgrid(xvals, yvals);
c = re + i*im;

res = mandelbrotIterate(c,niter);
pcolor(res);
shading flat;
colormap jet;
end
```

Generating the Mandelbrot Set Plot

- `drawMandelbrot([-2.5 2.5 -2 2],400,40)` produces the image:



More Mathematical Facilities

- Trend analysis using `diff` and `find`
- Polynomials
- Interpolation (linear and spline)
- Curve fitting