Intro to Math Problem Solving
October 12

A "bumps" function

A few more function rules

Graphics functions

A triangle function

The WRAP function

The GAP Game

Random quadratic equations

Homework #6

A "bumps" function

As a reminder of how functions work, let's set up a MATLAB function for the mathematical function defined here:

$$z = \frac{2}{e^{(x-1/2)^2 + y^2}} + \frac{-2}{e^{(x+1/2)^2 + y^2}}$$

bumps.m

```
function z = bumps ( x, y )

%% BUMPS evaluates a function z(x,y) that has a bump up and one down.
%
%  X, Y, are the evaluation point.  X and Y can be vectors or arrays.
%
%  Z is the function value at (X,Y).
%
  z = 2.0 ./ exp ( ( x - 0.5 ).^2 + y.^2 ) ...
    - 2.0 ./ exp ( ( x + 0.5 ).^2 + y.^2 );

  return
end
```

Check with a plot?

Being able to see your work is huge help in catching errors. We can't call plot() because z is a function of two variables instead of 1.

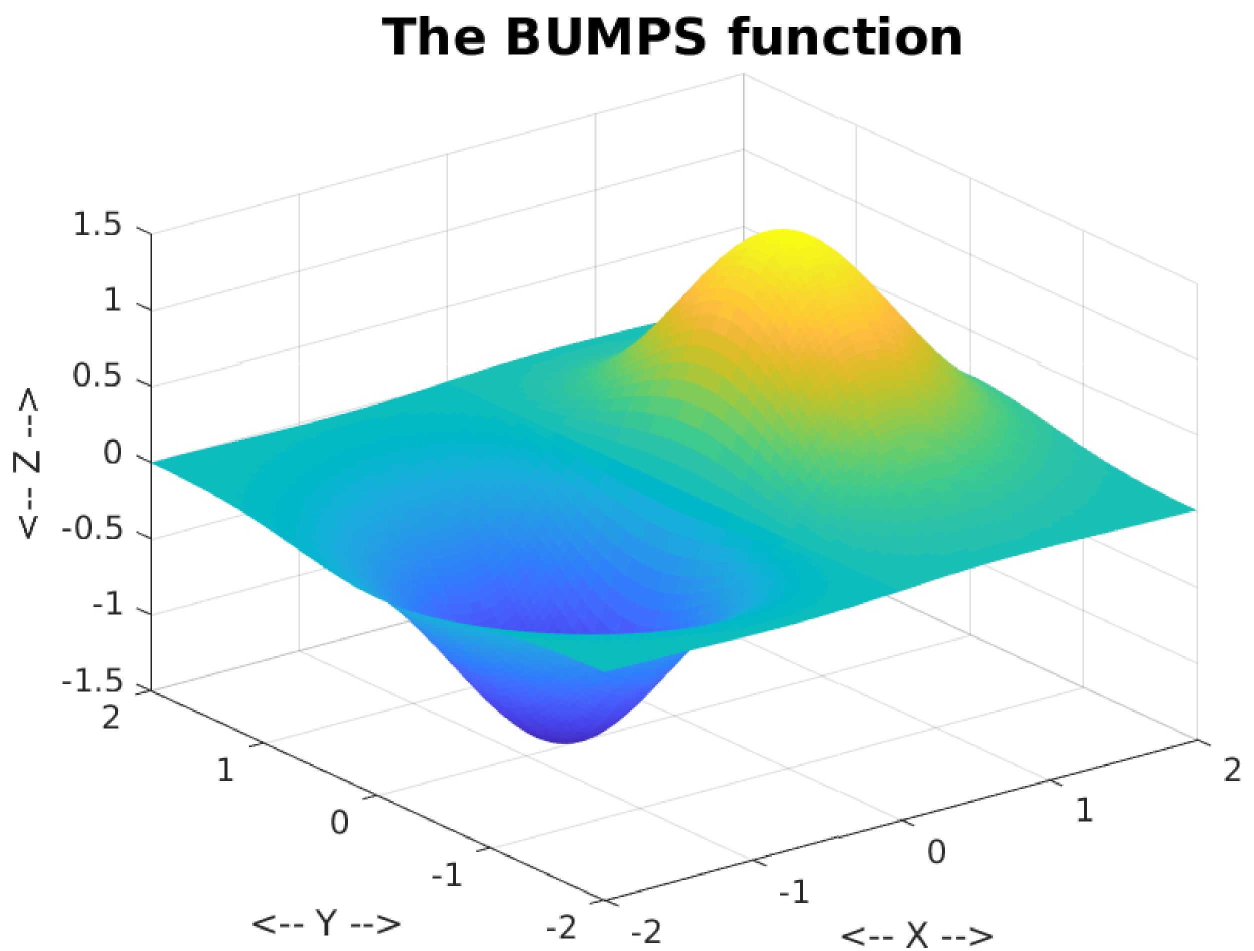
However, MATLAB has a surf() function which can display functions z(x,y).

To use it, we need to create TABLES (or arrays or matrices) of X, Y, and Z data.

bumps_surf.m

```
x = linspace ( -2.0, + 2.0, 101 );     ← make x and y lists.  This is familiar.
y = linspace ( -2.0, + 2.0, 101 );

[ X, Y ] = meshgrid ( x, y );          ← this makes X and Y "tables".  This is new.

Z = bumps ( X, Y );                    ← "Z" will contain a "table" of Z values.

surf ( X, Y, Z, 'Edgecolor', 'None' );   ← Make a plot.

title ( 'The BUMPS function', 'Fontsize', 16 );
xlabel ( '<-- X -->' );
ylabel ( '<-- Y -->' );
zlabel ( '<-- Z -->' );

print ( '-djpeg', 'bumps.jpg' );
```

bumps.jpg, a plot made from "tables"



A few notes about functions

Now that we've been introduced to functions, it will be helpful to look at a few details and extra features that may come up from time to time.

Supply the right number of inputs!

```
function total = addem ( a, b, c )
  total = a + b + c;
  return
end

total = addem ( 1, 2, 3 )
total = addem ( 1, 2, 3, 4 )
total = addem ( 1, 2 )
total = addem ( )
total = addem
```

The function must set all outputs!

```
function [ big, small ] = maxmin ( a, b )

  if ( a < b )
    big = b;
    small = a;
  else
    big = a;
    little = b;     <- Oops!  Meant to say "small = ..."
  end

  return
end
```

This function will FAIL, but only in cases where a >= b!

You can RETURN early

```
function [ m, e ] = scientific ( x )
  m = x;
  e = 0;
  if ( 1 <= x && x < 10 )
    return
  end

  while ( m < 1 )
    m = m * 10;
    e = e - 1;
  end

  while ( 10 <= m )
    m = m / 10;
    e = e + 1;
  end

  return
end
```

Use ERROR() for Warnings

```
function ratio = dividem ( a, b )

  if ( b == 0 )
    error ( 'A/B undefined when B = 0!' )
  end

  ratio = a / b;

  return
end
```

A "PlotShape" function

If a triangle is described by xlist and ylist, we know that the command

```
fill ( xlist, ylist, 'r' );
```

draws a triangle filled with red; but if we want the <u>outline</u>, we have to repeat the first point:

```
plot ( [ xlist, xlist(1) ], [ ylist, ylist(1)], 'r-' );
```

Also, if we want to specify rgb color, we have to use a more complicated plot command.

And we usually draw a thicker line than the default.

Why don't we just write a function that looks like plot(), but takes care of these details for us?

plotshape.m

```
function plotshape ( xlist, ylist, color )

% plotshape will draw the polygon defined by xlist, ylist.
%
% color can be 'r', 'g', 'b', 'c', 'm', 'y', 'w', 'k'
% or it can be an RGB triple like [1.0, 0.4, 0.0].
%
  plot ( [ xlist, xlist(1)], [ ylist, ylist(1)], 'Color', ...
    color, 'LineWidth', 3 );

  return
end
```

Some Triangle Functions

This week's homework will be all about triangles. One question asks you to compute the perimeter, which involves summing the lengths of the sides:

```
perim = distance ( vertex 1 to vertex 2 )
      + distance ( vertex 2 to vertex 3 )
      + distance ( vertex 3 to vertex 1 )
```

Here "distance()" is NOT a MATLAB function, but just represents the fact that we need to compute that distance.

This almost looks like a perfect FOR loop:

```
perim = 0.0;
for i = 1 : 3
  perim = perim + distance ( vertex i to vertex i+1 )
end
```

but this "breaks" on the last step!

A simple fix

```
perim = 0.0;
for i = 1 : 3

  if ( i == 1 )
    vertex_old = vertex(3);
  else
    vertex_old = vertex(i-1);
  end

  perim = perim + distance ( vertex(i) – vertex_old );
end
```

Fix with extra variable

```
perim = 0.0;
im1 = 3;

for i = 1 : 3

  perim = perim + distance ( vertex(i) - vertex(im1) );
  im1 = i;

end
```

So im1 is 3, 1, 2, in loops 1, 2, and 3.

A clever fix

```
perim = 0.0;

for i = 1 : 3
  perim = perim + distance ( vertex(i) -  ...
    vertex( mod ( i+1, 3 ) + 1 );
end
```

because mod(i+1,3)+1 = 3, 1, 2 for i = 1, 2, 3.

Advantages to a FOR loop

We could have compute the perimeter by simply writing out the three terms of the sum.

The advantage of figuring out a way to use a FOR loop for that kind of computation is that you can easily adapt the computation to handle a square (4 sides), and you can see how to generalize it to handle a polygon with n sides.

A “wrap around” function

In the triangle perimeter case, we saw that while the first vertex was counting 1, 2, 3, the second vertex was going 2, 3, 1. That is, once we reached the maximum value of 3, the next value “wrapped around” to 1.

We tried three different ways to deal with this issue, with an IF statement, or an extra variable, or a MOD function.

What if we wrote a “wrap around” function that said, “I am counting between 1 and n, but if I say n+1, I must really mean 1.”

wrap.m

```
function i = wrap ( i, ilo, ihi )
%
% WRAP uses “wrap-around” counting.
%
  n = ihi + 1 – ilo;                ← How many values?
  i = ilo + mod ( i – ilo, n );   ← Where does I
   belong?

  return
end
```

Wrap Demo

wrap(-2,1,3) = 1	wrap(-2,3,6) = 6
wrap(-1,1,3) = 2	wrap(-1,3,6) = 3
wrap(0,1,3) = 3	wrap(0,3,6) = 4
wrap(1,1,3) = 1	wrap(1,3,6) = 5
wrap(2,1,3) = 2	wrap(2,3,6) = 6
wrap(3,1,3) = 3	wrap(3,3,6) = 3
wrap(4,1,3) = 1	wrap(4,3,6) = 4
wrap(5,1,3) = 2	wrap(5,3,6) = 5
wrap(6,1,3) = 3	wrap(6,3,6) = 6
wrap(7,2,3) = 1	wrap(7,3,6) = 3

Version 4

```
perim = 0.0;

for i = 1 : 3

   perim = perim + distance ( x(i),y(i) to  x(wrap(i+1)),y(wrap(i+1)) );

end
```

wrap(i+1) = 2, 3, 1 as i = 1, 2, 3.

We will find wrap.m useful for some other problems we will work on.

Generalize to Polygon

```
function perim = polygon_perimeter ( xlist, ylist )

  n = length ( xlist );
  perim = 0.0;
  im1 = n;

  for i = 1 : n

    perim = perim + distance ( ( xlist(i), ylist(i)) to (xlist(im1),ylist(im1) ) )
    im1 = i;

  end

  return
end
```

The "Gap N" Game

Keep tossing a fair coin until

$$| \text{Heads} - \text{Tails} | == N$$

Score = total number of tosses

Write a function Gap(N) that returns the score. Estimate the average score given N.

The Packaging...

```
function nTosses = Gap( N )
```

```
Heads = 0; Tails = 0; nTosses = 0;
while ( abs(Heads-Tails) < N )
   nTosses = nTosses + 1;
   if ( rand() < 0.5 )
       Heads = Heads + 1;
   else
       Tails = Tails + 1;
   end
end
```

The Header…

```
function nTosses = Gap(N)
```

output parameter list

input parameter list

The Body

```
Heads = 0; Tails = 0; nTosses = 0;
while ( abs(Heads-Tails) < N )
    nTosses = nTosses + 1;
    if ( rand ( ) < 0.5 )
        Heads = Heads + 1;
    else
        Tails = Tails + 1;
    end
end
```

The necessary output value is computed.

Local Variables

```
Heads = 0; Tails = 0; nTosses = 0;
while ( abs(Heads-Tails) < N )
   nTosses = nTosses + 1;
   if ( rand ( ) < 0.5 )
       Heads = Heads + 1;
   else
       Tails = Tails + 1;
   end
end
```

A Helpful Style

```
Heads = 0; Tails = 0; n = 0;
while ( abs(Heads-Tails) < N )
   n = n + 1;
   if ( rand ( ) < 0.5 )
       Heads = Heads + 1;
   else
       Tails = Tails + 1;
   end
end
nTosses = n;
```

Explicitly assign output value at the end.

The Specification…

```
function nTosses = Gap(N)
```

```
% Simulates a game where you
% keep tossing a fair coin
% until |Heads - Tails| == N.
% N is a positive integer and
% nTosses is the number of
% tosses needed.
```

Compute an Expected Value

The gap() function puts the computation into a neat package. Now we can easily refer to that computation by name. Let's use it to estimate the average value of the score (number of tosses) for a given value of N (the gap size).

Strategy:

Play "Gap N" a large number of times, say "M".

Add each score to "total".

After M games, compute total/M to get a typical score for this value of N.

Solution…

```
N = input('Enter N:');
M = 10000;
s = 0;
for k=1:M
   s = s + Gap(N);
end
ave = s/M;
```

A very common methodology for the estimation of expected value.

Sample Outputs

```
N = 10  Expected Value =  98.67
```

```
N = 20  Expected Value = 395.64
```

```
N = 30  Expected Value = 889.11
```

Solution…

```
N = input('Enter N:');
M = 10000;
s = 0;
for k=1:M
   s = s + Gap(N);
end
ave = s/M;
```

Program development is made easier by having a function that handles a single game.

What if the Game Was Not "Packaged"?

```
s = 0;
for k=1:M
    score = Gap(N)
    s = s + score;
end
ave = s/M;
```

```
s = 0;
for k=1:M
    Heads = 0; Tails = 0; nTosses = 0;
    while ( abs(Heads-Tails) < N )
       nTosses = nTosses + 1;
       if ( rand() < 0.5 )
           Heads = Heads + 1;
       else
           Tails = Tails + 1;
       end
    end
    score = nTosses;
    s = s + score;
end
ave = s/M;
```

A more cumbersome implementation

Is there a Pattern?

```
N = 10  Expected Value =  98.67
```

```
N = 20  Expected Value = 395.64
```

```
N = 30  Expected Value = 889.11
```

Compute MANY Expected Values

We computed the expected value of Gap(N) for one value of N.

We would expect that the score (number of tosses), would increase as we increased N (the gap between Heads and Tails).

The interesting question is how this expected value increases with N.

We can estimate the expected value of Gap(N) for a range of N-values, say, N = 1:30

Pseudocode

```
for N=1:30

    Estimate expected value of Gap(N)

    Display the estimate.

end
```

Pseudocode

```
for N=1:30
```

Estimate expected value of Gap(N)

Display the estimate.

```
end
```

Refine this!

Done that..

```
M = 10000;
s = 0;
for k=1:M
    s = s + Gap(N);
end
ave = s/M;
```

Sol'n Involves a Nested Loop

```
for N = 1:30
% Estimate the expected value of Gap(N)
   s = 0;
   for k=1:M
      s = s + Gap(N);
   end
   ave = s/M;
   fprintf('%3d    %16.3f',N,ave)
end
```

Sol'n Involves a Nested Loop

```
for N = 1:30
% Estimate the expected value of Gap(N)
   s = 0;
   for k=1:M
      s = s + Gap(N);
   end
   ave = s/M;
   disp(sprintf('%3d   %16.3f',N,ave))
end
```

But during derivation, we never had to reason about more than one loop.

Output

N	Expected Value of Gap(N)
1	1.000
2	4.009
3	8.985
4	16.094
28	775.710
29	838.537
30	885.672

Looks like N^2.

Maybe increase M to solidify conjecture.

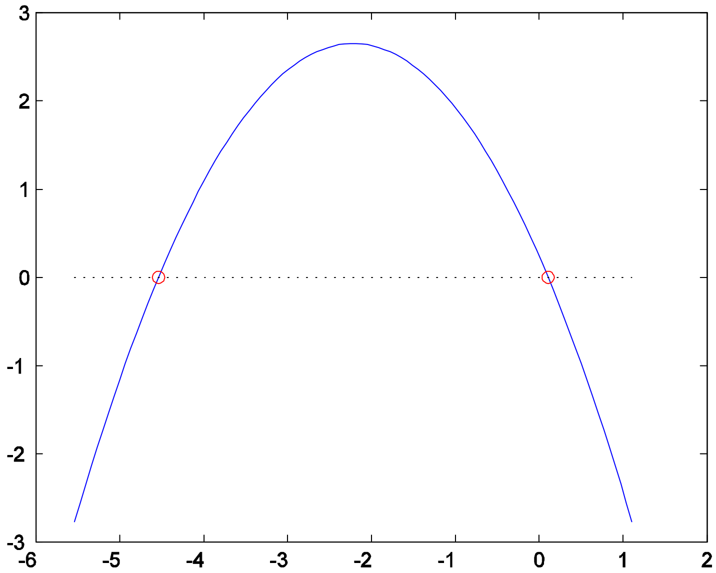
Random Quadratics

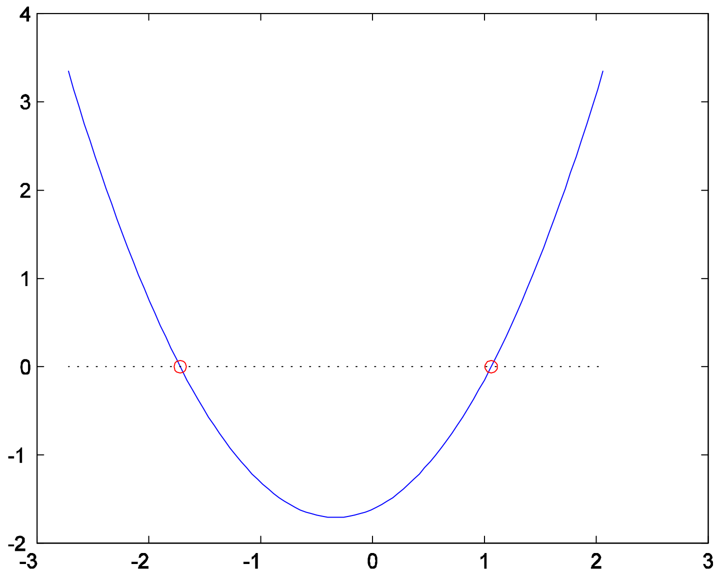
Generate a random quadratic

$$q(x) = ax^2 + bx + c$$

If it has two real roots, then plot $q(x)$ and highlight the roots.

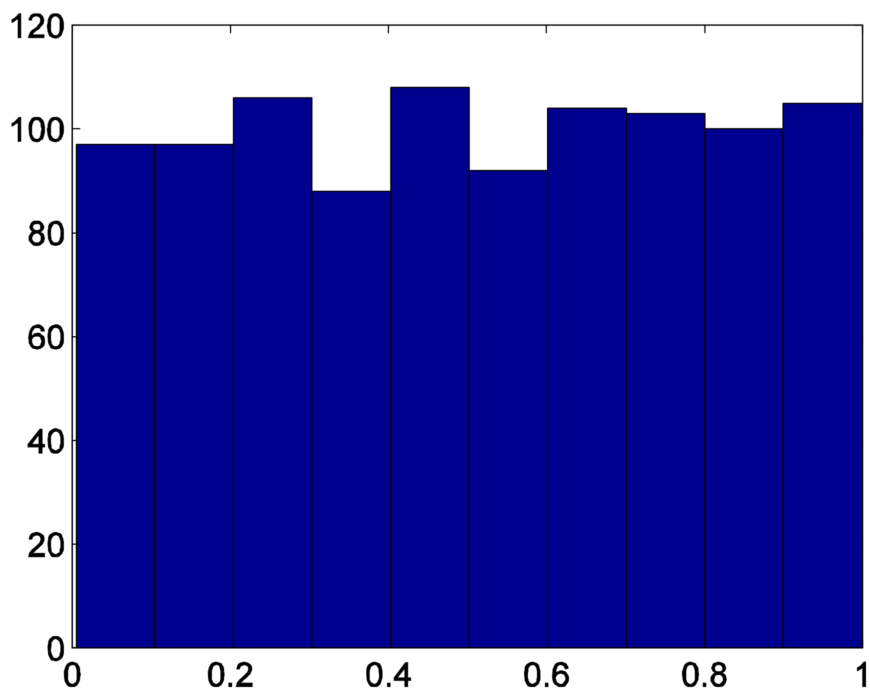
Sample Output





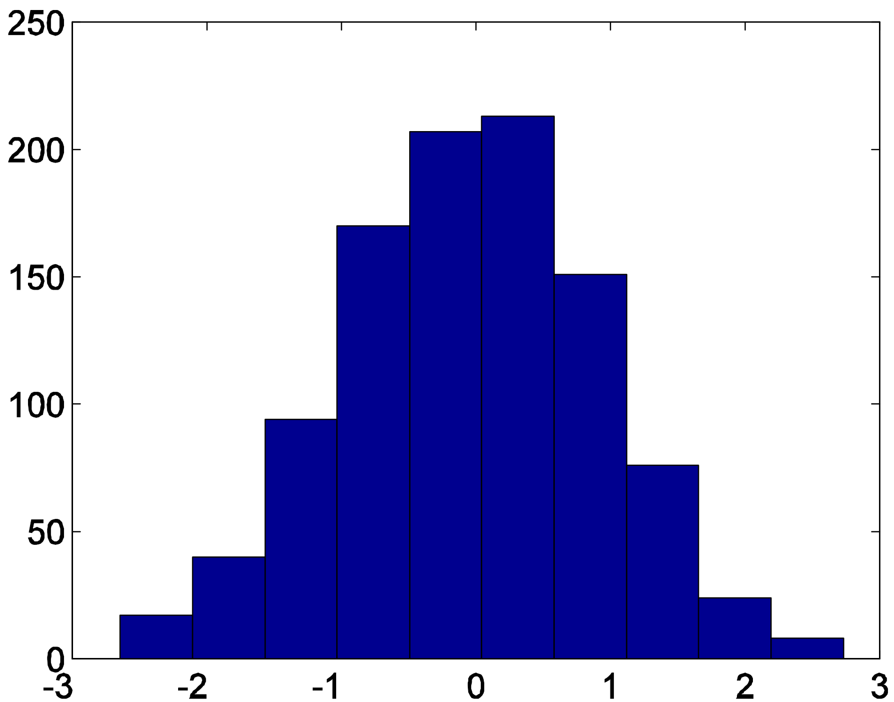
Uniform Random Numbers

rand() gives us a random value in [0,1], and picks values “uniformly”. Here is a histogram of a selection of 1000 such values.



Normal Random Numbers

randn() gives random values in (-oo,+oo), with average value 0, and a strong tendency to be close to 0. Negative values are as likely as positive ones.



Set random coefficients

```
function [a,b,c] = quadratic_random()

% To make our random coefficients more
% interesting, we generate them with randn().

  a = randn();
  b = randn();
  c = randn();

  return
end
```

Input & Output Parameters

```
function [a,b,c] = quadratic_random()
```

A function can have more than one output parameter.

Syntax: [v1,v2,...]

A function can have no input parameters.

Syntax: Nothing

Computing the Roots

```
function r = quadratic_roots_real ( a, b, c )

  d = b^2 - 4.0 * a * c;

  if ( d < 0.0 )
    r = [];
  elseif ( d == 0.0 )
    r = - b / ( 2.0 * a );
  else
    r = [ ( - b + sqrt ( d ) ) / ( 2.0 * a ), ...
          ( - b - sqrt ( d ) ) / ( 2.0 * a ) ];
  end

  return
end
```

Script Pseudocode

```
for k = 1:10
    Generate a random quadratic;
    Compute its real roots;
    If there are two real roots:
        plot the quadratic and roots.
end
```

Script Pseudocode

```
for k = 1:10
    Generate a random quadratic;
    Compute its real roots;
    If there are two real roots:
        plot the quadratic and roots.
end
```

```
[a,b,c] = quadratic_random();
```

Script Pseudocode

```
for k = 1:10
   [a,b,c] = quadratic_random();
   Compute its real roots;
   If there are two real roots:
        plot the quadratic and roots.
end
```

```
r = quadratic_roots_real(a,b,c);
```

Script Pseudocode

```
for k = 1:10
  [a,b,c] = quadratic_random();
  r = quadratic_roots_real(a,b,c);
  If two real roots:
        plot the quadratic and roots.
end
```

```
n = length ( r );  if ( n == 2 )
```

Script Pseudocode

```
for k = 1:10
  [a,b,c] = quadratic_random();
  r = quadratic_roots_real(a,b,c);
  n = length ( r );
  if ( n == 2 )
      plot the quadratic and roots.
  end
end
```

Plot the Quadratic and Roots

```
r_min = min(r);
r_max = max(r);
x = linspace(r_min-1,r_max+1,100);
y = quadratic_evaluate ( a, b, c, x );
plot(x,y, ...
     x,0*y,':k', ...
     r_min,0,'or', ...
     r_max,0,'or')
```

Plot the Quadratic and Roots

```
r_min = min(r);
r_max = max(r);
x = linspace(r_min-1,r_max+1,100);
y = quadratic_evaluate ( a, b, c, x );
plot(x,y,x,0*y,':k',r_min,0,'or',r_max,0,'or
  ')
```

This determines a nice range of x-values.

Plot the Quadratic and Roots

```
r_min = min(r);
r_max = max(r);
x = linspace(r_min-1,r_max+1,100);
y = quadratic_evaluate ( a, b, c, x );
plot(x,y,x,0*y,':k',r_min,0,'or',r_max,0,'or
  ')
```

Get the y-values.

Evaluate a quadratic polynomial

```
function y = quadratic_evaluate ( a, b, c, x )

%% QUADRATIC_EVALUATE evaluates a quadratic polynomial.
%
%  A, B, C are the coefficients of the polynomial.
%
%  X is the number, list, or table of evaluation points.
%
%  Y is the number, list or table of values.
%
  y = a * x.^2 + b * x + c;

  return
end
```

Plot the Quadratic and Roots

```
r_min = min(r);
r_max = max(r);
x = linspace(r_min-1,r_max+1,100);
y = quadratic_evaluate ( a, b, c, x );
plot(x,y,x,0*y,':k',r_min,0,'or',r_max,0,'or
  ')
```

Graphs the quadratic.

Plot the Quadratic and Roots

```
r_min = min(r);
r_max = max(r);
x = linspace(r_min-1,r_max+1,100);
y = quadratic_evaluate ( a, b, c, x );
plot(x,y,x,0*y,':k',r_min,0,'or',r_max,0,'or
  ')
```

A black, dashed line x-axis.

Plot the Quadratic and Roots

```
r_min = min(r);
r_max = max(r);
x = linspace(r_min-1,r_max+1,100);
y = quadratic_evaluate ( a, b, c, x );
plot(x,y,x,0*y,':k',r_min,0,'or',r_max,0,'or
  ')
```

Highlight root r_min with red circle.

Plot the Quadratic and Roots

```
r_min = min(r);
r_max = max(r);
x = linspace(r_min-1,r_max+1,100);
y = quadratic_evaluate ( a, b, c, x );
plot(x,y,x,0*y,':k',r_min,0,'or',r_max,0,'or
  ')
```

Highlight root r_max with red circle.

Complete Solution with 3 User Functions

```
for k=1:10
    [a,b,c] = quadratic_random();
    r = quadratic_roots_real ( a, b, c );
    n = length ( r );
    if ( n == 2 )
      r_min = min(r); r_max = max(r);
      x = linspace(r_min-1,r_max+1,100);
      y = quadratic_evaluate ( a, b, c, x );
      plot(x,y,x,0*y,':k',r_min,0,'or',r_max,0,'or')
      shg        <- Bring graphics window to front!
      pause(2)   <-Wait a few seconds.
    end
end
```

Homework #6
Due October 20th

hw038: write a function which computes the perimeter of a triangle. (The 'wrap.m' function file might help you.)

hw039: write a function which shrinks a triangle.

hw040: write a function which computes the area of a quadrilateral, using a function for the area of a triangle.

(Homework #5 is due tomorrow midnight!)