

Intro Math Problem Solving

October 10

Question about Accuracy

Rewrite Square Root Script as a Function

Functions in MATLAB

Road Trip, Restaurant Examples

Writing Functions that Use Lists

Functions with Multiple Outputs

Control and Memory

Projects

Question

For homework problem **hw032**, we needed to check the child's age every year, to see when the expected weight reached 1,000 pounds. But the weight jumps to considerably more than 1,000.

"I tried taking tiny steps of 0.0001 year at a time, and I came very close to hitting 1,000 pounds exactly. But I could not figure out how to print the weights at the whole number years...I got no printout at all!"

10,000 steps of size 0.0001 \neq 1.0

Instead of advancing age 1 year at a time, let's take tiny steps to catch the jump to 1000 pounds better.

We still want some output, maybe once every year, but not at every tiny step! We see the 1,000 pound jump, but the weights at whole number ages **do not print out!**

```
age = 1.0;
next_age = 1.0;

while ( true )

    if ( age == next_age )    <- Print weight at whole number ages, 1, 2, 3, ...
        fprintf ( 'Weight at age %d = %g\n', age, weight );
        next_age = next_age + 1;
    end

    weight = 2.20462 * exp ( 0.175571 * age + 2.197099 );
    if ( weight >= 1000.0 )
        fprintf ( 'Reached 1,000 pounds at age %g\n', age );
        break;
    end

    age = age + 0.0001;    <- Take tiny steps in age, to see more detail.

end
```

Insight Through

Real Numbers Aren't Accurate

```
>> x = 1;
>> dx = 1/3;
>> x = x + dx;
>> x = x + dx;
>> x = x + dx    <- x = 1 + 1/3 + 1/3 + 1/3 so it should equal 2, right?
```

```
x = 2.0000    <- Well that looks right!
```

```
>> x == 2.0    <- Ask if x equals 2: 1 means yes and 0 means no!
```

```
ans = logical 0    <- What's going on?
```

```
>> format long    <-- Print x to lots of digits! It looks OK, but it's lying!
>> x
```

```
x = 2.0000000000000000
```

```
>> x - 2.0    <- Subtract 2 from x, and expect 0...
```

```
ans = -2.220446049250313e-16
```

```
So x is more like 1.999999999999999
```

Insight Through

Moral: Use Integer Fractions

If we base our age on whole numbers, like $i / 10000$, we can be sure not to have accuracy problems.

```
i = 10000; ← Start at age = 1.  
while ( true );  
    age = i / 10000;  
    weight = 2.20462 * exp ( 0.175571 * age + 2.197099 );  
    if ( mod ( i, 10000 ) == 0 )  
        fprintf ( 'X = %g, Weight = %g\n', age, weight );  
    end  
    if ( weight >= 1000.0 )  
        fprintf ( "Exceeded 1000 pounds at age %g\n", age );  
        break;  
    end  
    i = i + 1; ← ready for next step.  
end
```

Insight Through

MATLAB Functions

In Mathematics, we think of a function as a rule that takes an input value (or sometimes several input values) and returns an output value; we might write " $y(x) = \sin(x)$ " or " $y = \sin(x)$ ".

We've seen that MATLAB allows us to use many familiar mathematical functions, using a language similar to mathematics:

$y = \sin(x)$; $y = \text{abs}(x)$; $y = \text{max}(x1,x2)$; $y = \text{rand}()$;

In the expression " $y=\sin(x)$ " we say " x is the input" and " y is the output".

The most common pattern is 1 input, 1 output, but we also have:

$\text{big} = \text{max} (x1, x2) \leftarrow$ 2 inputs, 1 output;

$[x,y] = \text{ginput}()$; \leftarrow 0 inputs, 2 outputs;

$\text{plot} (\text{xlist}, \text{ylist})$; \leftarrow 2 inputs, 0 outputs;

Computing SQRT with a Script

MATLAB allows the user to create new functions to carry out computations that arise frequently.

We already know how to write scripts.

Let's look at the differences between a script and a function, to see the advantages of that approach.

We will suppose that we want to compute the square root of any positive number x , and that we will employ our simple averaging technique that we talked about in our very first class.

sqrt_script.m

```
% Estimate square root of number.  
% The number must be named A.  
% The answer will be stored in "RESULT".
```

```
L = A;  
for k = 1 : 10  
    W = A/L;  
    L = (L+W)/2;  
end  
RESULT = L;
```


Using sqrt_script.m

Assuming the file sqrt_script.m is in our MATLAB directory, we can use it by setting A, invoking the script, and then looking at the value in RESULT:

```
A= 2017.0;
```

```
sqrt_script;
```

```
fprintf ('sqrt(%f) is %f\n', A, RESULT);
```

Disadvantages:

Notice that:

- A) the input must be stored in the variable `A`. We can't ask for the square root of "`B`", and we can't ask for the square root of an expression like $(C+1)/2$ or the number 2017.
- B) the output must go into the variable `RESULT`;
- C) running the script will create or modify variables "`k`" and "`L`". If our program was already using variables of that name, now they've been changed.
- D) To understand all the effects of the script, we have to be able to see every line of the script, since each line is creating or setting variables, which might affect our calculation.

Side Effects of sqrt_script

```
k = 99;
```

```
W = 17.0;
```

```
A = 2017.0;
```

```
sqrt_script;
```

```
fprintf ( 'sqrt(%f) is %f\n', A, RESULT);
```

```
fprintf ( 'Is k still 99? k = %d\n', k );
```

```
fprintf ( 'Is W still 17? W = %g\n', W );
```

sqrt_function.m

```
function y = sqrt_function ( x )           ← The function declaration
%
%% SQRT_FUNCTION estimates the square root of number.
%
% X, input, is the number whose square root is desired.
% Y, output, is the estimated square root of X.
%
L = x;
for k = 1 : 10
    W = x/L;
    L = (L+W)/2;
end
y = L;

return
end
```

NO Side Effects of sqrt_function

```
k = 99;
```

```
W = 17.0;
```

```
A = 2017.0;
```

```
script_sqrt;
```

```
fprintf ( 'sqrt(%f) is %f\n', A, RESULT);
```

```
fprintf ( 'Is k still 99? k = %d\n', k );
```

```
fprintf ( 'Is W still 17? W = %g\n', W );
```

What has changed?

function RESULT = sqrt_function (A) ← Declaration

L = A;

for k = 1 : 10

W = A/L;

L = (L+W)/2;

end

RESULT = L;

return

← Exit

end

What's the difference?

A function promises to perform a specific calculation.

The header specifies input and output variables, but these names are just "pseudonyms"; the user can choose other names.

All other activities of the function are hidden, and have no effect on the user who calls it.

The Declaration Statement

function RESULT = sqrt_function (A)

The function declaration, or header statement says three things:

1) the name of the function:

"sqrt_function";

2) the "temporary" names of input: "A";

3) the "temporary" names of output:
"RESULT".

Names Don't Matter Now!

Instead of using variables with particular names, the function header just looks for input in the input parentheses. Inside the function, the input will be called "A" and the output "RESULT", but you are free to use any names you want.

```
you:      side = sqrt_function ( area )  
          ||                   ||
```

```
header: RESULT = sqrt_function ( A )
```

```
you:      value = sqrt_function ( 2017 )  
          ||                   ||
```

```
header: RESULT = sqrt_function ( A )
```

```
you:      sqrt_function ( 2017 ) ← RESULT will print, but is not stored.  
          ||                   ||
```

```
header: RESULT = sqrt_function ( A )
```

Some Rules for Functions

The name of the function should match the name of the file it is stored in.

```
function a = rectangle_area ( width, height )  
should be stored as "rectangle_area.m"
```

A function can have 0, 1 or several inputs. A zero-input function has empty parentheses:

```
x = rand ( );
```

A function can have 0, 1 or several output values. If there is more than one output value, square brackets must be used:

```
function [ s, c ] = trig_functions ( angle )
```

In MATLAB, some functions are specified in such a way that they can be called with 0, 1, or several arguments. Note, for example, the rand() and plot() functions:

```
x = rand; x = rand(); x = rand(1); x = rand(1,5);
```

```
plot ( xlist, ylist ); plot ( xlist, ylist, 'r-' ); plot ( xlist, ylist, 'LineWidth', 2 );
```

A Function Hides its Work

The function cannot "see" any values in the user program, except for the input values. It gives those values temporary names.

The function may set up more temporary variables as it computes.

When the function is done, the output value is returned to the user, and all the function's variable names and data "disappear".

We say that a function "doesn't have side effects": interaction is limited to the inputs and outputs only.

Define and Use a New Function

The cost of a road trip depends on:

- * your car's mileage (miles per gallon);
- * distance traveled (miles);
- * cost of a gallon of gas (\$ per gallon).

Create a function "road_trip_cost()" that calculates the cost in \$.

road_trip_cost.m

```
function cost = road_trip_cost ( mpg, miles, gas_price )  
  
    gallons = miles / mpg;  
    cost = gallons * gas_price;  
  
    return  
end
```

Note that the variable "**gallons**" is created inside this function, and disappears when the function is done.

Use the function:

header:

```
function cost = road_trip_cost ( mpg, miles, gas_price )
```

```
mpg = 25.5;
```

```
miles = 1230;
```

```
gas_price = 2.35;
```

```
cost = road_trip_cost ( mpg, miles, gas_price ); ← same names
```

```
rate = 25.5;
```

```
distance = 1230;
```

```
charge = 2.35;
```

```
price = road_trip_cost ( rate, distance, charge ); ← any old names
```

```
bucks = road_trip_cost ( 25.5, 1230.0, 2.35 ); ← numbers
```

Insight Through

A Restaurant Meal Cost Function

Your restaurant order has a certain cost.

Blacksburg adds 11% tax to the meal cost.

You want to tip, at the rate of 10%, 15% or 20%, of the meal plus tax, depending on poor, average, or good service.

You want to round up your cost to the dollar.

We have one input, the basic meal cost, and three outputs, the amount we pay based on the service we got.

Price of Restaurant Meal

```
function [ low_tab, medium_tab, high_tab ] = restaurant_tab ( meal )

    tax_rate = 0.11;      ← Blacksburg!
    tax = meal * tax_rate;
    charge = meal + tax;

    low_tip = charge * 0.10;
    medium_tip = charge * 0.15;
    high_tip = charge * 0.20;

    low_tab = ceil ( charge + low_tip );    ← The ceil() function rounds up.
    medium_tab = ceil ( charge + medium_tip );
    high_tab = ceil ( charge + high_tip );

    return
end
```


Multiple Outputs

If a function has multiple outputs, you supply corresponding variables in a list. If you shorten the list, only the first few variables will be set.

```
meal = 120.0;
```

```
[ lo, med, hi ] = restaurant_tab ( meal );
```

```
[ low, medium, high ] = restaurant_tab ( meal )
```

```
[ l, m ] = restaurant_tab ( 120.0 )
```

```
[ low_tip ] = restaurant_tab ( meal )
```

```
cheapo = restaurant_tab ( meal )
```

```
restaurant_tab ( meal )    ← only prints the first output.
```

You **CANNOT** ask for **only** the medium tip, the second output variable! To get the second output, you must get the first as well. To get the third, you must get the first and second.

The Theron Formula

Our formula for a child's weight at age n:

$$lb = 2.20462 e^{(0.175571 * n + 2.197099)}$$

This is easy to turn into a function:

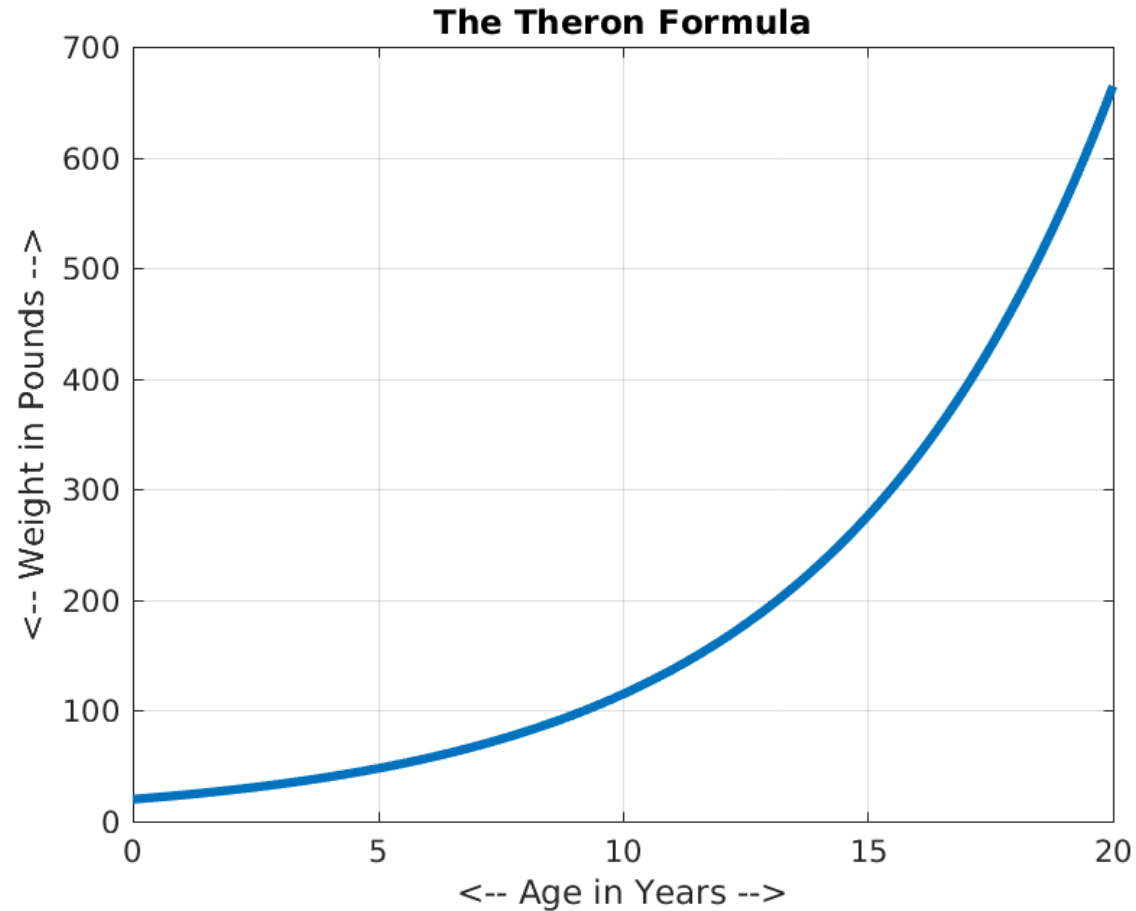
```
function lb = theron ( n )  
    lb = 2.20462 * exp( 0.175571 * n + 2.197099 );  
    return  
end
```

Plot A User Function

Can we plot our Theron function, just as we would plot the sine function?

```
n = linspace ( 0.0, 24.0, 101 );  
y = theron ( n );
```

Plot the Theron Weight Formula



Can Input Be a Vector?

For our Theron plot, the input "n" was a list of 101 values.
We were actually lucky that worked!

MATLAB can do arithmetic with a list in a way that is useful to us, as long as we use "dot operators" when needed:

List .* List

Anything ./ List

List .^ Anything

Anything .^ List

Even though "n" is a list, the Theron formula is "safe":

$$I_b = 2.20462 * \exp (0.175571 * n + 2.197099);$$

Plot Another Formula

Consider this formula for the weight of a trout, in kilograms, at age Y in years:

$$\text{kg} = 25.1 (1 - e^{(-1.19 y)})^3$$

We want to write a function "trout" that will accept the age Y , and return the weight KG . It will work fine.

Then, we decide we want to plot KG as a function of Y , which means we are going to give our "trout" function a list of values "Y", rather than just a single value.

What could go wrong?

trout.m

```
function kg = trout ( y )
```

```
    kg = 25.1 * ( 1.0 - exp ( - 1.19 * y ) ) ^ 3;
```

```
    return
```

```
end
```

Why will this function have trouble if we try to pass in a list of values in Y , rather than just one value?

Plot() uses Lists, Lists use Dot Operators

```
y = linspace ( 0, 6, 101 );
```

```
kg = trout ( y );
```

Error using ^

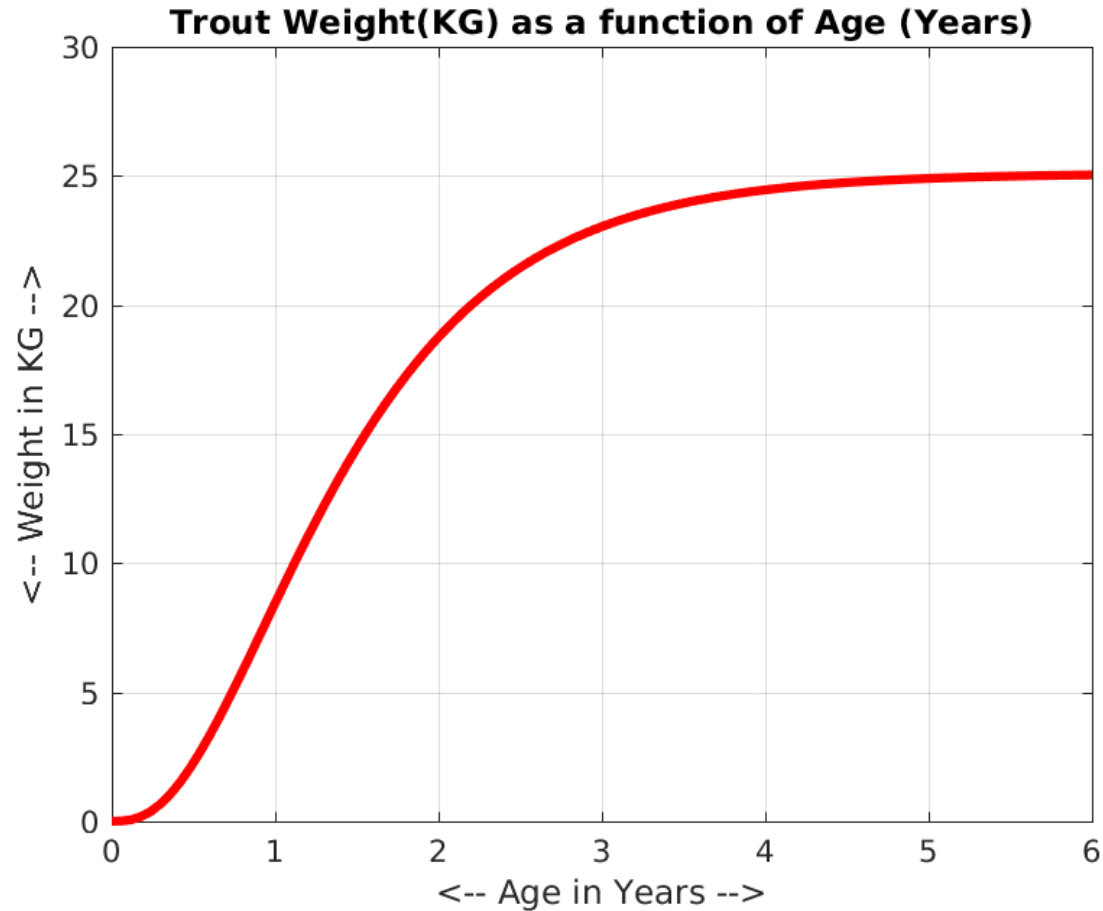
One argument must be a square matrix and the other must be a scalar. Use POWER (.^) for elementwise power.

Error in trout (line 2)

```
kg = 25.1 * ( 1.0 - exp ( - 1.19 * y ) ) ^ 3;
```

Since y is a list, we need to use ".^" instead of "^"

After we fix the function...



What if input is a vector?

A MATLAB function will allow input variables to be lists. If the input can be a list, then the calculations must be careful to use the "dot operators" as appropriate.

This is the most common thing to "break" when a function gets a list as input.

There are some other issues that can come up, and we will run into them as we go along.

Will sqrt_function work with List input?

```
function RESULT = sqrt_function ( A )
%
%% SQRT_FUNCTION2 estimates the square root of number.
%
% A, input, is the number whose square root is desired.
%
% RESULT, output, is the estimated square root of X.
%
L = A;
for k = 1 : 10
    W = A / L;
    L = ( L + W ) / 2;
end
RESULT = L;

return
end
```

No Error Message = No Error?

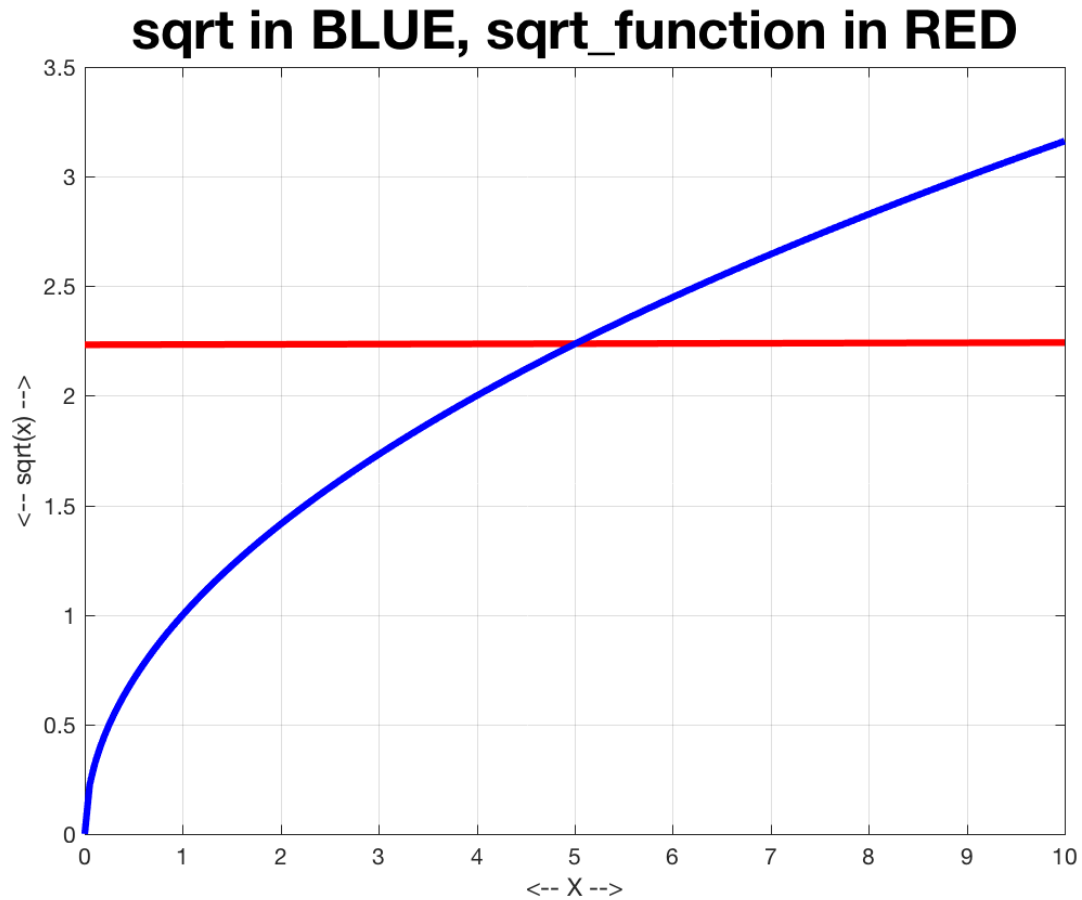
Suppose we wanted to plot the data returned by "sqrt_function()".

If we evaluate sqrt_function(x), we don't get an error message. But if we compare against MATLAB's sqrt() function, we see **something is wrong**.

What's worse than an error message? A silent error!

```
x = linspace ( 0.0, 10.0, 201 );  
y1 = sqrt_function ( x );  
y2 = sqrt ( x );  
plot ( x, y1, 'r', x, y2, 'b' );
```

The Vector Input Breaks Our Function!



A "tiny" error, a missing "dot"

```
function RESULT = sqrt_function2 ( A )
```

```
    L = A;
```

```
    for k = 1 : 10
```

```
        W = A ./ L;          <- A and L are lists. Use the DOT division!
```

```
        L = ( L + W ) / 2;
```

```
    end
```

```
    RESULT = L;
```

```
    return
```

```
end
```

Input that is also Output...

The functions we have seen so far take input, but return a completely new set of variables.

Sometimes it is convenient to have a variable go "through" a function, that is, be both an input and an output quantity.

One example checks numbers XLO and XHI , to ensure that $XLO \leq XHI$ and both numbers are in the range $[0,1]$.

xrange.m

```
function [ xlo, xhi ] = xrange ( xlo, xhi )
```

```
if ( xhi < xlo ) <- Make sure XLO <= XHI
```

```
    t = xlo;      <- You swap two variable values by using a helper variable "t".
```

```
    xlo = xhi;
```

```
    xhi = t;
```

```
end
```

```
xlo = max ( xlo, 0.0 ); <- Make sure XLO and XHI are in [0,1].
```

```
xlo = min ( xlo, 1.0 );
```

```
xhi = max ( xhi, 0.0 );
```

```
xhi = min ( xhi, 1.0 );
```

```
return
```

```
end
```

Notice that the inputs are modified, and become the outputs.

Insight Through

Real Roots of a Quadratic Equation

Task: Given the equation

$$a x^2 + b x + c = 0$$

return the real roots, if any.

The discriminant is:

$$D = b^2 - 4ac$$

If $D < 0$, no real roots

$D = 0$, 1 real root:

$$r = -b/(2a)$$

$D > 0$, 2 real roots:

$$r_1 = (-b + \sqrt{D})/(2a),$$

$$r_2 = (-b - \sqrt{D})/(2a)$$

real_roots.m

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

    d = b^2 - 4 * a * c;

    if ( d < 0.0 )
        r = [];
    elseif ( d == 0 )
        r = - b / ( 2 * a );
    else
        r = [ ( - b + sqrt(d) ) / ( 2 * a ), ...
              ( - b - sqrt(d) ) / ( 2 * a ) ];
    end

    return
end
```

Using real_roots.m

```
a = 1.0
```

```
b = 4.0;
```

```
c = 3.0;
```

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

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

<- Count how many results we got back.

```
if ( n == 0 )
```

```
    fprintf ( 'No real roots.\n' );
```

```
elseif ( n == 1 )
```

```
    fprintf ( 'One root: %g\n', r(1) );
```

```
else
```

```
    fprintf ( ' Roots: %g and %g\n', r(1), r(2) );
```

```
end
```

Control and Memory

Suppose a script contains a line that uses a function.

MATLAB reads the script line by line, and we say the script is "in control". When the function call is reached, the script "pauses", and control passes to the function, in order to compute the necessary value.

MATLAB copies input values from the script to the function. The function may set up other variables, but these are all local, and hidden from the script.

When the function has completed its computation, the output values are returned to the script, which then resumes control.

Script

```
a = 1  
b = f(2)  
c = 3
```

function

```
function y = f(x)  
z = 2*x  
y = z+1
```

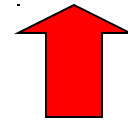
Let's execute the script line-by-line and see what happens during the call to `f`.

Script

```
a = 1  
b = f(2)  
c = 3
```

function

```
function y = f(x)  
z = 2*x  
y = z+1
```



x , y , z serve as local variables during the process. x is referred to as an input parameter.

● $a = 1$
 $b = f(2)$
 $c = 3$

```
function y = f(x)
z = 2*x
y = z+1
```

a: 1

Green dot tells us what the computer is currently doing.

Control passes to the function.

```
a = 1  
• b = f(2)  
c = 3
```

```
• function y = f(x)  
  z = 2*x  
  y = z+1
```

a: 1

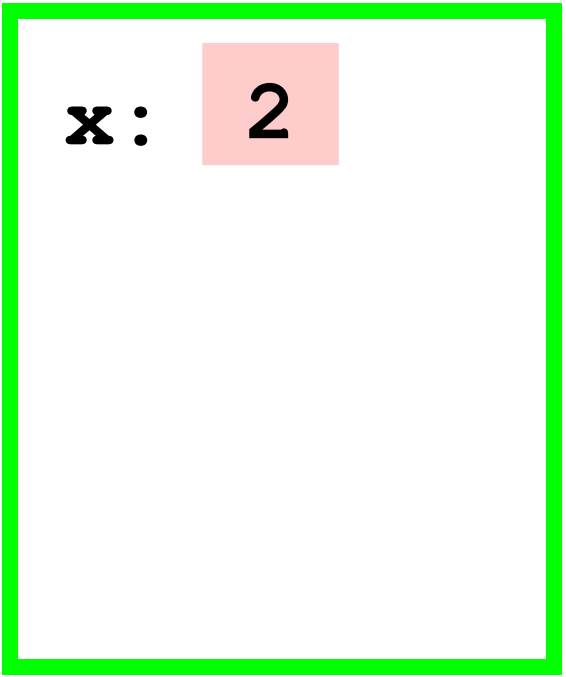
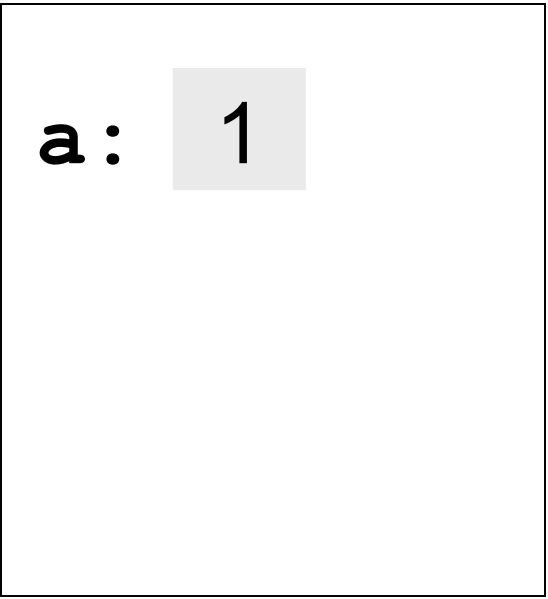
x: 2

The
input
value is
assigned
to x

Control passes to the function.

```
a = 1  
• b = f(2)  
c = 3
```

```
• function y = f(x)  
  z = 2*x  
  y = z+1
```



The input value is assigned to x

`a = 1`

● `b = f(2)`

`c = 3`

`function y = f(x)`

● `z = 2*x`


`y = z+1`

`a: 1`

`x: 2`

`z: 4`


`a = 1`

 `b = f(2)`

`c = 3`

`function y = f(x)`

`z = 2*x`

 `y = z+1`

`a: 1`

`x: 2`

`z: 4`

`y: 5`

The
last
command
is
executed

Control passes back to the calling program

```
a = 1  
● b = f(2)  
c = 3
```

```
function y = f(x)  
z = 2*x  
y = z+1
```


a: 1

b: 5

After the
the value is
passed back,
the call to the
function ends and
the local variables
disappear.

`a = 1`

`b = f(2)`

 `c = 3`

`function y = f(x)`

`z = 2*x`

`y = z+1`

`a: 1`

`b: 5`

`c: 3`

Insight Through

Repeat to Stress the
distinction between
local variables
and
variables in the calling program.

Script

```
z = 1
x = f(2)
y = 3
```

function

```
function y = f(x)
z = 2*x
y = z+1
```

Let's execute the script line-by-line and see what happens during the call to `f`.

● $z = 1$
 $x = f(2)$
 $y = 3$

```
function y = f(x)
z = 2*x
y = z+1
```

z: 1

Green dot tells us what the computer does next.

Control passes to the function.

z = 1
x = f(2)
y = 3

function y = f(x)
z = 2*x
y = z+1

z: 1

x: 2

The
input
value is
assigned
to x

$$z = 1$$

● $x = f(2)$

$$y = 3$$

function $y = f(x)$

● $z = 2 * x$

$$y = z + 1$$

z: 1

x: 2

z: 4

This does NOT change

$z = 1$
● $x = f(2)$
 $y = 3$

function $y = f(x)$
● $z = 2 * x$
 $y = z + 1$

$z : 1$

$x : 2$

$z : 4$

Because
this
is the
current
context

`z = 1`

● `x = f(2)`

`y = 3`

`function y = f(x)`

`z = 2*x`

● `y = z+1`

`z: 1`

`x: 2`

`z: 4`

`y: 5`

The
last
command
is
executed

Control passes back to the calling program

```
z = 1  
● x = f(2)  
y = 3
```

```
function y = f(x)  
z = 2*x  
y = z+1
```


```
z: 1
```

```
x: 5
```

After the
the value is
passed back,
the function
"shuts down"

z = 1

x = f(2)

 **y = 3**

function y = f(x)

z = 2*x

y = z+1

z: 1

x: 5

y: 3

Question Time

```
x = 1;  
x = f(x+1);  
y = x+1
```

```
function y = f(x)  
x = x+1;  
y = x+1;
```

What is the output?

A. 1 B. 2 C. 3 D. 4 E. 5

Question Time

```
x = 1;  
x = f(x+1);  
y = x+1
```

```
function y = f(x)  
x = x+1;  
y = x+1;
```

What is the output?



A. 1 B. 2 C. 3 D. 4 E. 5

Function Summary

The first line of a function file defines the output, the function name, and the input.

The function is stored in a file whose name matches the function name.

A script can call the function, as long as the script and function file are in the same directory.

The script must supply values for all required inputs, but the values don't have to have the same names as used in the function, and they can even be numbers or expressions.

Projects

You know that 15% of the course grade involves a final project.

One version of the project will simply be a long assignment of problems like the homework you have been doing, which I will hand out before Thanksgiving break.

You may instead propose a topic of interest to you. This would involve doing some reading, writing some programs and a report, and perhaps a short demo in class.

I have placed some documents in the new "projects" folder on Canvas. See if a topic interests you; then we can agree on a list of tasks for you.