Introduction to Scientific Computing

Function vs. Scripts,

Functions as Black Boxes,

Interacting with Functions and Scripts

Functions vs. scripts

- M-file: any set of instructions stored in a MATLAB file

Function

```
function d = myFunFunction(a,b,c)
% Takes scalar values a, b, and c,
modifies and sums the values

% Modifying the input values
var1 = 2*a;
var2 = b^2;
var3 = sqrt(c);

% Calculating the output
d = var1 + var2 + var3;
```

Script

```
% Define values of scalars a, b, and c
a = 1;
b = 2;
c = 15;

% Modifying the scalar values a, b, and c
var1 = 2*a;
var2 = b^2;
var3 = sqrt(c);

% Calculating the sum of the modified scalar
values
d = var1 + var2 + var3;
```

Commonalities between functions and scripts

- Both stored in M-files
- Both modified using the “Editor” window
- Standalone pieces of code
- Can be called from the command window or other functions

Why use scripts?

•What is a Matlab “script”

- MATLAB M-file where all variables, constants, etc. are called within the file

•Why develop code with a “script”

- You may want to develop multiple lines of code to solve a problem
- Consistently typing that code into the command window can become laborious / doesn’t “save” your code
- You can rapidly comment out and adjust syntax to debug your code

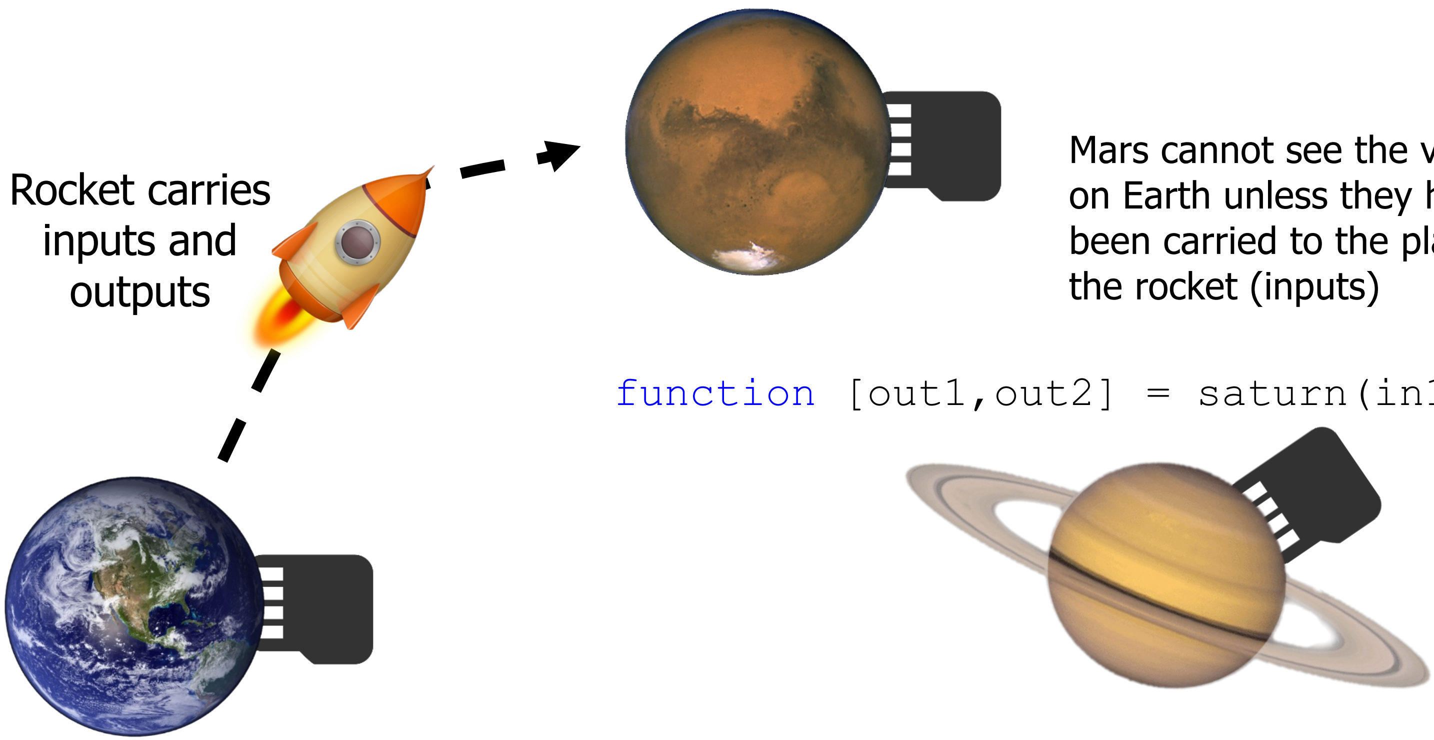
Why use functions?

- **What is a function?**
  - A MATLAB M-file that starts with `function`
  - Generally, you need to pass information into the function and the function returns one to several results

- **Why use functions?**
  - The function serves as a black box of code
  - Its functionality has been tested / verified and it can be called at any time without questioning the output
  - Long, complicated computations can be broken into multiple, smaller pieces of code
  - Enables top-down program design

Visual interpretation of scripts vs. functions

```
function [out1,out2,out3] = mars(in1,in2)
```



How to specify / use a function

- **Establishing a function requires a specific syntax**
  - For a single output and input

```
function output = myFunc(input)
```

  - For multiple outputs and inputs

```
function [output1,output2,output3] =
                            myFunc(input1,input2,input3)
```

*Note: this would be on a single line

- **Naming / saving a function**
  - Must start with a letter from the alphabet (a through z)
  - The name used to save the function needs to be the function name

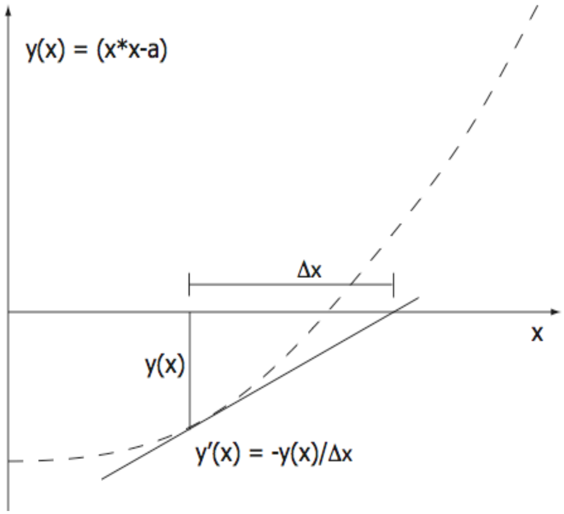
Newton's Method

Compute the square root of a given number, `a`, using only `+`, `−`, `*`, `/`

Consider the function `y(x)=(x`^{2}`−a)`.

We are looking for the root of this function, that is `x` such that `(x`^{2}`−a)=0`

Newton's Method: At any given value of `x` we approximate the function by it's tangent line and compute where the line crosses the `x` axis. Repeat this procedure to get closer and closer to the answer



$$y(x_i) = \left(x_i^{\,2} - a\right); \quad y'(x_i) = 2x_i$$

$$y'(x_i) \approx \left(\frac{-y(x_i)}{\Delta x_i}\right) \Rightarrow \Delta x_i = \left(\frac{-y(x_i)}{y'(x_i)}\right)$$

$$x_{i+1} = x_i + \Delta x_i$$

$$x_{i+1} = x_i + \left(\frac{-y(x_i)}{y'(x_i)}\right) = x_i + \left(\frac{-\left(x_i^{\,2} - a\right)}{2x_i}\right)$$

$$= \left(x_i + (a/x_i)\right)/2$$

Function: `my_sqrt(a)`

- **Initial estimate for square root:** `x = (a/2)`
  - Repeat as `abs(x^2 - a) > 1.0e-6`
    - Computer the function value at `x`: `y = (x^2 - a)`
    - Compute the gradient at `x`: `gradient = 2*x`
    - Compute the step: `step = -y/gradient`
    - Compute the new estimate: `x = x + step`

- **Matlab code**

  my_sqrt.m

Interacting with a function

- **How do I "call" a function**
  - Initiate function execution from the command line - make sure you include the inputs! (DEMO)
  - Functions can be called from other functions (DEMO)

Function H1 line

- **The first line of a function is referred to as the "H1 line" and is searchable by MATLAB using** `lookfor`

```matlab
function out1 = testFunction(in1)
% A test function
% This would report, in the help function, all of the functionality of the
% function
% in1 = function input (units: not specified)
% out2 = function output (units: not specified)
```

Command Window

```
>> lookfor 'a test function'
testFunction                              - A test function
fx
```

Function help lookup

- The first contiguous block of text is returned as the help query for the function

Command Window

```
>> help testFunction
  A test function
  This would report, in the help function, all of the functionality of the
  function
  in1 = function input (units: not specified)
  out2 = function output (units: not specified)
>>
```

Recursive functions

- Functions can call themselves - such functions are called "recursive" functions

- Ex.

```
function recursiveAdd(nToAdd)
% Adds a number to the input up to
% a value of 25

if nToAdd<25
    nToAdd = nToAdd+1;
    disp(nToAdd)
    recursiveAdd(nToAdd);
end
```

What happens when a function is called

1. Arguments are *copied* to the `input` parameters

2. Function body is run in its own *private* work space

- it can't see or use any variables in the calling context

3. When the function is done the `output` parameters are *copied* to the output parameters so that they are available to the calling context.

4. Paths

- Matlab finds functions by searching a set of directories specified in the path
- The current directory is included (first) in that list
- It is often convenient to store the function definitions in .m files in the current directory

Search paths

- MATLAB searches through a series of folders described as the "search path" when functions or scripts are invoked
- These paths are folders on your computer and include the current working folder (the folder from which your functions/scripts are currently running) and folders included during the MATLAB installation (often toolboxes)
- The working folder at startup can be changed using the `userpath` function, e.g. `userpath('C:\MyMATLABwork')`
- The order in which MATLAB searches along paths can be revealed by typing `path` at the command line
- Paths can also be "permanently" added to the search path using the `addpath()` function
- Paths can be removed using the `rmpath()` function

I called a function - where did my variables go?

- **Unless cleared, variables will persist for “scripts”, but not for functions**
  - Whatever happens in a function, stays in a function
  - For instance, a variable, `myVar`, could be invoked in `myFunction1` and resulting in `myOut`, which is returned to another function, say `myFunction2` - this function could also use `myVar` without retaining “memory” of `myVar` as invoked in `myFunction1`
  - DEMO - retention of workspace variables for a “script” but not for a function

Cleaning up your MATLAB workspace and command line

- **Variables held in memory can be cleared using the `clear` command or by manually by selecting and deleting variables**
  - Do NOT invoke the clear command within a "`function`"
  - `clear` can be called within a "script" or from the command line

- **The command window may be cleared using the `clc` command**

Commenting functions / scripts

- **The general rule is at least one comment for every three lines of code**
  - Proper commenting helps others understand, add to, or debug your code
  - Proper commenting helps the instructors grade your assignments

- **Commenting / uncommenting large swaths of code**
  - Highlight a selection or place your cursor on a line and press “ctrl+r” to comment a section or line of code, respectively
  - “ctrl+t” uncomments the code

Example 1: Monthly Loan Payment

- Fixed rate mortgage (抵押贷款、按揭)

- Monthly payment (`output`) c
  - amount paid monthly ensuring that the loan is paid in full with interest at the end of its term

- Depends on (`input`) :
  - Loan amount P_0 (principal)
  - Interest rate r
  - Number of monthly payments N

$$c = \left[r + \frac{r}{(1+r)^N - 1} \right] P_0 = \frac{r(1+r)^N}{(1+r)^N - 1} P_0$$

loan.m

Example 2 Evaluating an Expression

$$f(x)=\frac{x^3\sqrt{3x+5}}{\left(x^2+1\right)^2}$$

- **Write a function file to evaluate allowing for `x` to be a vector**

Calculate

- `f(x) for x = 6`
- `f(x) for x = 1, 3, 5, 7, 9, 11`

[Ex10_2.m]

Example 3 Converting Temperatures

- **Write a function to convert degrees Fahrenheit (° F) to degrees Celsius (° C)**

  FtoC.m

- Use it to solve the following problem:
  - The change in length of an object ΔL due to temperature change ΔT is given by where

    $$\Delta L = \alpha * L * \Delta T$$

    where α is the coefficient of thermal expansion.

  - Find the change in area of a rectangular aluminum plate 4.5 m by 2.25 m with $\alpha = 23\text{x}10^{-6}/\ ^\circ\text{C}$ when the temperature changes from 40° F to 92° F

Anonymous Function

- Good for short (one-line) calculations used frequently in a longer program.

  Example: converting Fahrenheit to Celsius FtoC.m

- General form

```
function_name = @ (arguments) expression
FtoC = @ (F) 5*(F-32)./9
cube = @ (x) x^3
circle = @ (x,y) 16*x^2+9*y^2
```

Examples: Anonymous Function

$$f(x)=\frac{e^{x^2}}{\sqrt{x^2+5}}$$

```
FA = @ (x) exp(x^2)/sqrt(x^2+5)
FA(2)
FA = @ (x) exp(x.^2)./sqrt(x.^2+5)
FA([1 0.5 2])
```

$$f(x,y)=2x^2-4xy+y^2$$

```
HA = @ (x,y) 2*x^2-4*x*y+y^2
HA(2,3)
```

Subfunctions

- A function file can contain more than one user-defined function
- First function defined, the "*primary*," is how the function is known to the rest of the program
- Other functions, "*subfunctions*," are only known inside the function file and each has its own workspace (local variables)
- Subfunctions are used to implement “utility” calculations for primary function

Example 4 Mean and Standard Deviation

- Write a function that calculates mean (average) and standard deviation of a list of numbers.

stat.m

$$\bar{x} = \frac{x_1 + x_2 + \ldots + x_n}{n}$$

$$\sigma = \sqrt{\frac{\sum_{i=1}^{i=n} (x_i - \bar{x})^2}{n-1}}$$

Function Functions

- MATLAB built-in function `fzero` can find the zeros (values of x where f(x) = 0) of any function f(x). How do we describe f(x) to `fzero`?

- Function functions like `fzero` accept functions as arguments in two ways
  - Function handle
  - Using the name of the function in a string

Function Handles

- A data type holding a unique value associated with any function (user-defined, built-in, anonymous)
- Obtained by

`@function_name`

`@cos`

`@FtoC (as function file)`

`FtoC (as anonymous function)`

Example: `funplot` - evaluates and graphs a function over a specified range returning a matrix of values at each extreme and the midpoint.

Use user-defined function

Use anonymous function

funplot.m, **Fdemo.m**

Function Names in Strings

- Older method, less efficient.

- Pass name of function as a string

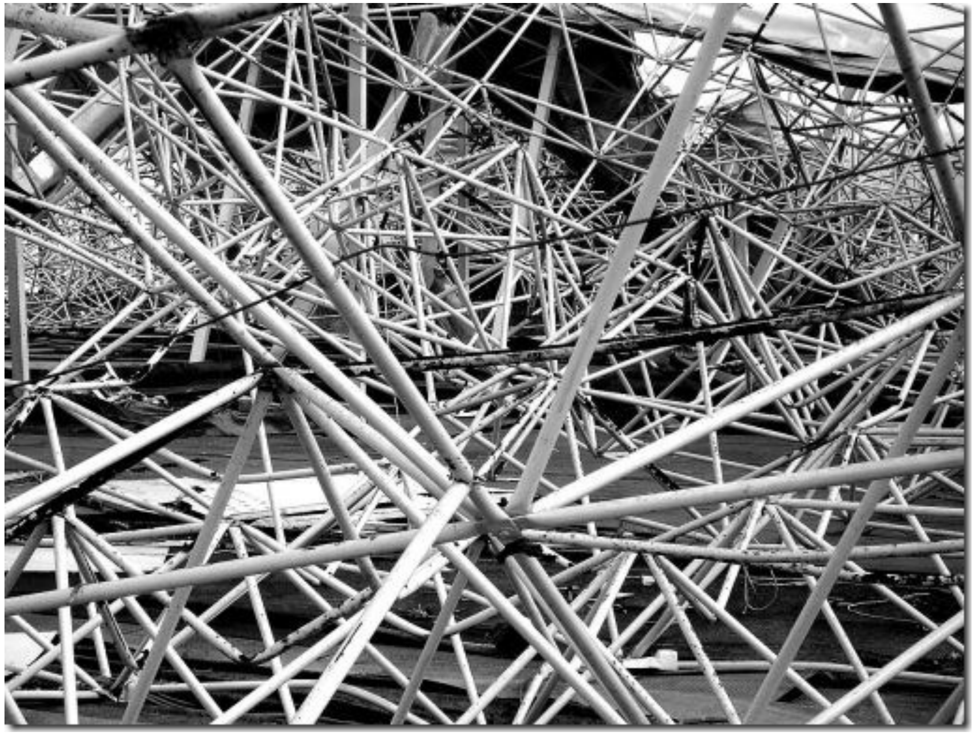
```
'cos'
'FtoC'
```

- Evaluate

```
var = feval('function_name', arguments)
```

funplotS.m

Solving real problems



Start simple and build to complexity

Accessing Vector Elements,

Iteration,

Relational Operators

Initializing vectors in MATLAB

•You now know of several ways to input a vector into MATLAB:

- Discrete method

```
x = [0.5, 1.5, 2, 3.2, 1.5, 0.4];
```

- Colon operator - establishes vectors with user-defined ranges and regularly-spaced intervals

```
x = 1:1:5; or x = 1:5;
x = -5:2:5;
```

- Linspace and logspace

```
x = linspace(1,5,5)
x = logspace(1,5,5)
```

Accessing vector elements

•What about accessing vector elements?

- For the following assume we have defined the following vector in memory, which we reset before each operation

```
x = [5, 9, 4, 1, 7, 3, 4, 8];
```

- Access the first vector index

```
y = x(1)          → y = 5
```

- Access a range of vector indices

```
y = x(1:3)        → y = [5, 9, 4]
y = x(6:8)        → y = [3, 4, 8]
```

- Access the last vector index

```
y = x(end)        → y = 8
```

Accessing vector elements

- **Assume** `x = [5, 9, 4, 1, 7, 3, 4, 8];`

- Access an offset vector index

  `y = x(1+4)` → `y = 7`

- Access the antepenultimate vector index

  `y = x(end-2)` → `y = 3`

- Access discrete indices

  `y = x([1 3 6])` → `y = [5, 4, 3]`

Overwriting / adding to vector elements

- **The existing contents of a vector may be overwritten or addended**
  - For the following assume we have defined the following vector in memory, which we will continuously modify

```
x = [1, 2, 3];
```

  - Overwrite the first index

```
x(1) = 5                → x = [5, 2, 3];
```

  - Overwrite the last index

```
x(end) = 1              → x = [5, 2, 1];
```

  - Add ending indices

```
x(end+1) = 8            → x = [5, 2, 1, 8];
```

Overwriting / adding to vector elements

- **Assume** `x = [5, 2, 1, 8];`

  - Add ending indices

    ```
    x(end+1) = [6, 9];
    ```

    → `assignment error (dimensions don't match)`

  - Add ending indices

    ```
    x(end+1:end+2) = [6, 9];
    ```

    → `x = [5, 2, 1, 8, 6, 9];`

Removing vector elements

- **Vector elements can be removed by specifying an empty set "[ ]"**
  - Say we establish the following vector

```
x = [8, 4, 5, 9, 3, 2, 5, 6];
```

  - Removing the first element of the vector

```
x(1) = [];
→ x = [4, 5, 9, 3, 2, 5, 6];
```

  - Removing several elements from the vector

```
x(end-3:end) = [];
→ x = [8, 4, 5, 9];
```

Iteration / `for` loops

- **Formula vectorization is preferred for speed, but sometimes iteration / loops are necessary**
  - A basic `for` loop that displays 1 through 20, by 1, at the command window

```
for jj = 1:20
    disp(jj)
end
```

  - `for` loops are preferred over `while` loops (shown later) when the number of iterations are known a priori / fixed

Iteration / `for` loops

- `for` **loops can take in colon operators of any viable form**

```
for jj = 1:3:19
    disp(jj)
end
```

- `for` **loops can also take discrete vectors**

```
for jj = [1 9 5]
    disp(jj)
end
```

Note: the for loop needs a closing "end" statement to execute properly

Iteration / `for` loops

- `for` **loops can be nested**

```
for jj = 1:10
    for kk = 1:10
        disp([jj kk])
    end
end
```

Note: each instance of the for loop requires an "end" statement

An example of a `for` loop

- Section 2.7.1 shows the use of a `for` loop for determining the square root of a number using Newton's method. Why is a `for` loop necessary here?

```
% Newton's method - as shown in section 2.7.1 of Essential Matlab, ed. 5,
% evaluated using for loops

clear; clc; close all

% We want to know the square root of scalar a
a = 2;

% Initial guess
x = a/2;

% For loop to converge to a solution
for jj = 1:10
    x = (x+a/x)/2;
end

disp(['The approximate square root is ',num2str(x,10),...
    ' and the actual solution is ',num2str(sqrt(a),10)])
```

Relational operators

- Loops can also be invoked using the `while` command – however, we first need to understand relational operators since `while` loops depend heavily on them

- Relational operators compare values (or vectors, matrices) and yield a true or false result

Relational operators

Relational Operator Syntax in MATLAB

Relational Operator	Meaning
<	Less than
<=	Less than or equal to
==	Equal to
~=	Not equal
>	Larger than
>=	Larger than or equal to

Iteration / `while` loops

- The `while` command executes a piece of code until a desired condition is satisfied

```
while <relationalExpresion>
    <expressionToEvaluate>
End
```

- A simple example

```
a = 1;
while a < 10
    a = a+1;
end
```

Iteration / `while` loops

`while` **loops are most appropriate when the extent (number of) iterations is <u>unknown</u> a priori**

(DEMO) Newton's method using a while loop to estimate the square root of a to a finite precision

Invoking a `for` and `while` loop for `n` iterations

- **Say you want to evaluate some expression *n* number of times and the value of the expression is not known a priori**

  - You have two options: a `for` loop or a `while` loop
  - `for` loops have compact syntax since they don't require a counter

Invoking a `for` and `while` loop for `n` iterations

- **Example: Newton's Method for evaluating the square root of a `for` loop**

```
n = 100;
a = 2;
x = a/2;
for ii = 1:n
  x = (x+a/x)/2;
end
```

Invoking a `for` and `while` loop for `n` iterations

- `while` **loop**

```
n = 100;
a = 2;
x = a/2;
ctr = 0;
while ctr < n
    ctr = ctr+1;
    x = (x+a/x)/2;
end
```

- **Neither method presents a clear time savings - what are you more comfortable with / what looks cleaner?**

Looping vs. Vectorization/Array Operations,

Decisions (`if`),

Boolean Expressions

Operations that Require while Loops,

Decisions Involving `if` / `elseif` / `else`,

Logical Operators

Array operations increase computation speed

•Consider the vector

`a = rand(1,N)` where `N` is the total number of array elements

Say we wish to calculate $b_i = a_i \times a_i$

We have three options

- Array operations / vectorization

  a.^2

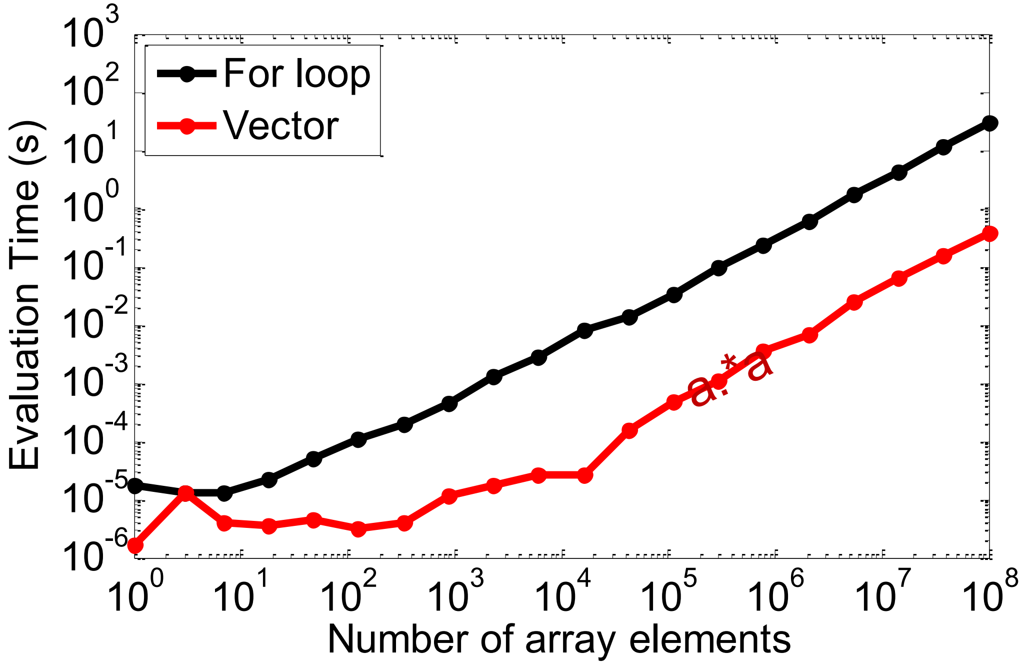
  a.*a

- Looping

  Using a `for` loop

Array operations increase computation speed

Time to Compute a*a using a for Loop vs. Vectorization



Time savings = orders of magnitude

Array operations increase computation speed

The Code Used to Generate the Previous Result (for your reference)

```
clear; clc; close all

n = round(logspace(0,8,20));

for jj = 1:length(n)

    a = rand(1,n(jj));

    clear b
    tic
    for kk = 1:length(a)
        b(kk) = a(kk)*a(kk);
    end
    t_for(jj) = toc;

    clear b
    tic
    b = a.*a;
    t_vect(jj) = toc;

end
```

continued...plotting commands

```
[ph] = plot(n,t_for,'k.-',n,t_vect,'r.-');
set(ph(1),'LineWidth',4,'MarkerSize',25)
set(ph(2),'LineWidth',4,'MarkerSize',25)
set(gca,'YTick',logspace(-10,10,21))
set(gca,'XTick',logspace(0,20,21))
set(gca,'XScale','log','YScale','log')
set(gca,'FontSize',20)
set(gca,'FontName','Arial')
lg = legend('For loop','Vector');
set(lg,'Location','NorthWest')
xlabel('Number of array elements')
ylabel('Evaluation Time (s)')
```

Decisions - the `if` statement

- **What if you want to evaluate a variable, vector, etc. based on certain criteria?**
  - The `if` statement evaluates a condition and executes a statement or statements if that condition is met
  - A simple example

```
a = 1;
if a == 1;
   disp('a is equal to 1! Yay!')
end
```

  - Like a `for` loop, the `if` statement is closed with an `end` statement

Decisions - nested `if` statements

- **What if several criteria need to be met?**
  - The `if` statement can be nested

```
a = 1;
b = 2;

if a == 1;
   if b == 2;
        disp(['a is equal to 1',...
              'and b is equal to 2.'...
              'Double yay!'])
   end
end
```

Boolean expressions / logical operators

Boolean expressions can be used to circumvent nested `if` statements

- The Boolean statements "&&" and "||" evaluate several expressions to determine if both or any of the statements are true

Logical Operator (Vectors / Scalars)	Logical Operator (Scalars)	Meaning
&	&&	and
|	||	or

Avoiding nested `if` statements using Boolean expressions

- The two if criteria can be combined into one if statement to clean up the code

```
a = 1;
b = 2;

if a == 1 && b == 2
     disp(['a is equal to 1',...
           'and b is equal to 2.'...
           'Double yay!'])
end
```

Decisions: `if`, `elseif`, and `else`

- **We learned about `if` decisions, which can be invoked to evaluate relational/logical criteria**
  - `if` statements are not limited to one set of criteria
  - Multiple criteria may be evaluated using the `if`, `elseif`, `else` formulation
  - Example
    - FEM shape function

A more complete list of logical operators

Table of Logical Operators

Logical Operator (Scalars)	**Logical Operator (Vectors)**	**Function call**	**Meaning**
&&	&	and(a,b)	And
||	|	or(a,b)	Or, inclusive
		xor(a,b)	Or, exclusive
~	~	not(a)	Not / compliment

The difference between `or` and `xor`

Truth table for `or` and `xor`

statement a	statement b	or(a,b)	xor(a,b)
F	F	F	F
F	T	T	T
T	F	T	T
T	T	T	F

`or` = truth for both statements leads to positive output
`xor` = truth for both statements leads to a negative output

Logical NOT

Logical Vectors

Logical NOT

- **We have discussed OR and AND ad nauseam - little has been said about NOT**
- **NOT behaves a little differently - it only requires one input - and returns the logical opposite**

  `not(0)` or `~0` returns `1`

  `not(1)` returns `0`

  `not(3)` returns `0`

- **Note: in the description of logical operators - nonzero values are treated as a truth / 1**

From logical scalars to vectors

- **We have covered logical scalars - most often in conjunction with decisions or while loops**
- **A logical scalar value can be invoked by making a relational comparison between one (in the case of not / ~) or two scalar values (in the case of >, < >=, ==, ~=, etc.)**
- **A true result is represented by 1 while a false result is represented by 0**

- Examples of logical scalars. What would be the result of each of the following?

```
a = 3>2            a = 2~=2
a = 2>3            a = 2==2
a = 2>=2           a = ~1
```

From logical scalars to vectors

- **Logical vectors are vectors of numbers (0 and 1) representing a `true (1)` or `false (0)` condition based on some relational expression**
  - Recall, a logical scalar

    `a = 3<=4` gives, `a = 1`

    This is of a "logical" class and is 1 byte in size (compared to 8 bytes for double precision)

  - Logical vector

    `a = [0 4 9 7 3 5] <= 4`

How are logical vectors "made"

- **Logical vectors occur as a result of a statement on a vector involving a relational operator -or- may be specified through discrete input**
  - Relational operator operating on a vector

    `a = [1 6 5] < 2`

    yields `a = [1 0 0]`

  - Directly specifying the logical operator

    `a = logical([1 0 0])`

    Call to logical function required otherwise a double precision number will be created

Logical vector rules

- **Say we define the following vectors in MATLAB:**

  `x = [5 9 2 4 3];` <- vector of class "double"

  `v = logical([1 0 1 0 1]);` <- logical vector

  `xp = x(v);` <- returns elements of `x` corresponding to `true` (`1`) indices of the logical vector `v`

- **Logical vector rules dictate:**
  1) **`v` may be used as a subscript for `x` given that it is of class "logical"**
  2) **only the elements of `x` corresponding to `1`'s (`true`) in `v` are returned**
  3) **`x` and `v` must be the same size**

Examples of logical vectors

What are the results of the following?

```
x = [1 4 3 6 2 8];
v = x>3;
```

yields `v`

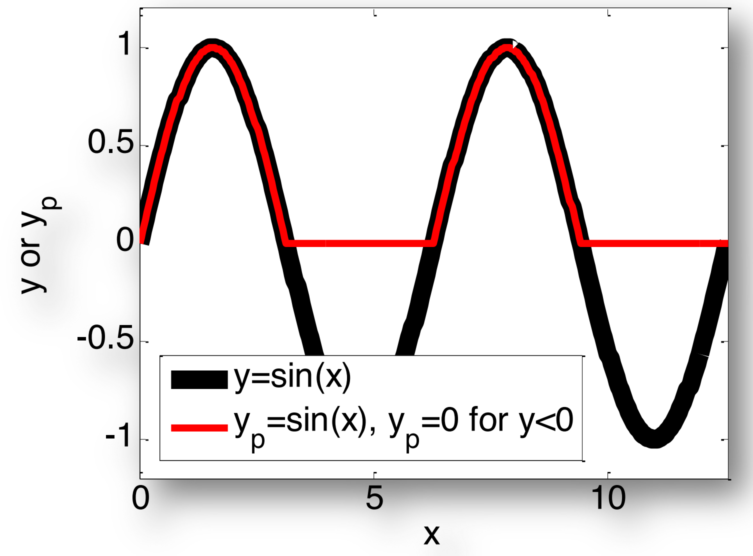
```
x = [1 4 3 6 2 8];
v = x<-3;
```

yields `v`

```
x = [1 4 3 6 2 8];
v = ~(~x);
```

Example of logical vector use: `sin(x)`

Produce the following plot: $y(x)=\begin{cases}\sin(x) & (\sin(x)>0)\\ 0 & (\sin(x)\le 0)\end{cases}$



Example of logical vector use: `sin(x)`

Method 1 - `for` and `if` expressions (not logical vectors)

```
% Domain vector and sin(x)
x = 0:pi/1000:3*pi;
y = sin(x);

% For loop and if condition used to
% set negative y values to 0
for jj = 1:length(y)
    if y(jj) < 0
        y(jj) = 0;
    end
end

% Plotting the result
plot(x,y,'k.')
```

Example of logical vector use: `sin(x)`

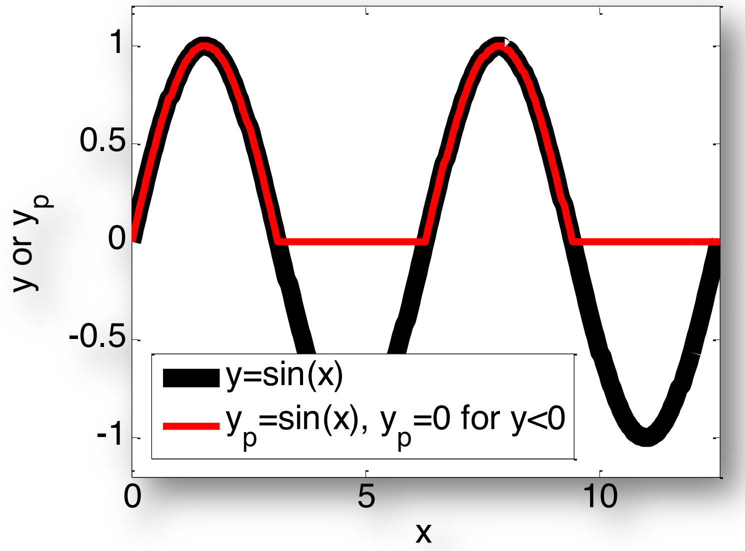
Method 2 - logical vectors

```
% Domain vector and sin(x)
x = 0:pi/1000:3*pi;
y = sin(x);

% Set all negative y values to 0
y = y.*(y>0)

% Plotting the result
plot(x,y,'k.')
```

This syntax is much more efficient!

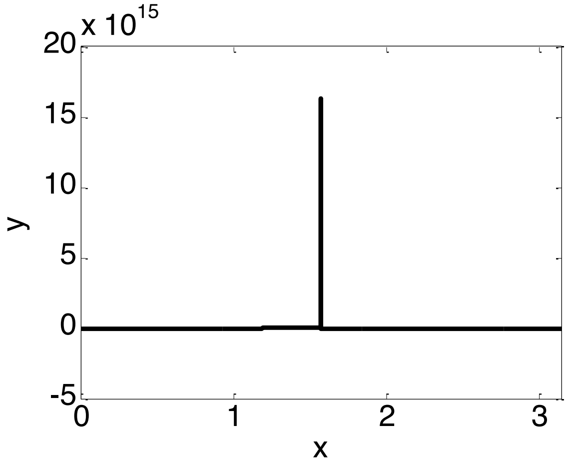
Code used for `plot`

```
% Domain and y = sin(x)
x = 0:pi/100:4*pi;
y = sin(x);

% Set y<0 to 0
yp = y.*(y>0);

% Plot the result
pl = plot(x,y,'k-',x,yp,'r-');
set(gca,'FontSize',18)
set(pl(1),'LineWidth',10)
set(pl(2),'LineWidth',4)
axis([0 4*pi -1.2 1.2])
xlabel('x'), ylabel('y or y_p')
lg = legend('y=sin(x)',...
    'y_p=sin(x), y_p=0 for y<0');
```



Example of logical vector use: `tan(x)`

Produce the following plot: $y(x) = \tan(x)$

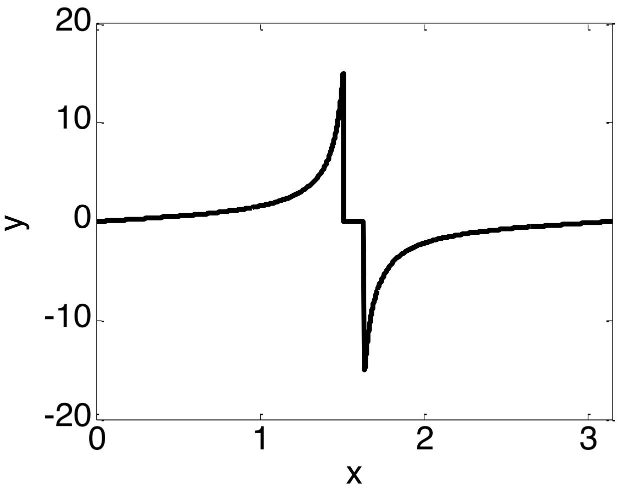



```
x = 0*pi:pi/10000:pi;
y = tan(x);
plot(x,y)
xlim([0 pi])
xlabel('x')
ylabel('y')
```

Can we use logical vectors to change the visualization of this function such that the details of y are not dominated by the asymptote?

Example of logical vector use: `tan(x)`

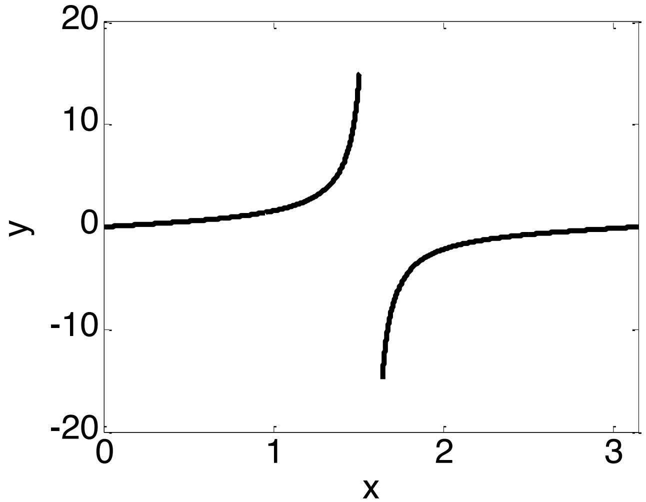
Set large (positive and negative) values to 0




```
x = 0*pi:pi/10000:pi;
y = tan(x);
y = y.*(abs(y)<15);
plot(x,y)
```

Example of logical vector use: `tan(x)`

Set large (positive and negative) values to NaN

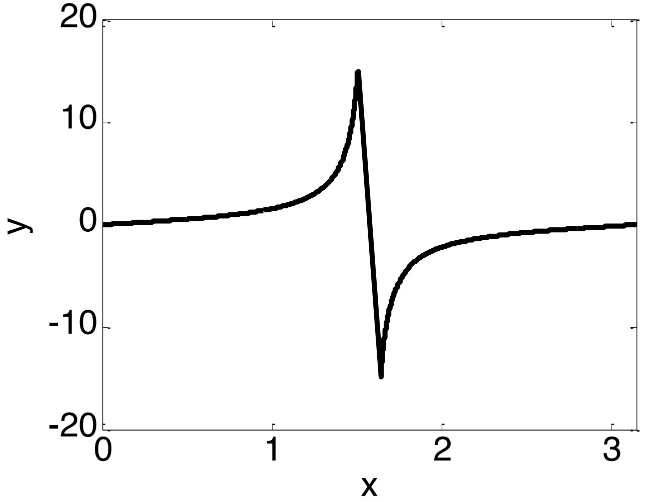



```
x = 0*pi:pi/10000:pi;
y = tan(x);
y(abs(y)>15) = NaN;
plot(x,y)
```

Note: a logical vector has been used to directly index and assign vector values

Example of logical vector use: `tan(x)`

Remove large (positive and negative) values




```
x = 0*pi:pi/10000:pi;
y = tan(x);
x = x(abs(y)<15);
y = y(abs(y)<15);
plot(x,y)
```

Example Sieve of Eratosthenes

- **Finds all of the primes between 1 and n**
  - A prime number can be divided evenly only by 1 and itself, and it must be a whole number greater than 1.
  - An algorithm that uses logical expressions and logical indexing

- **Algorithm**

1) List the numbers from 2 to n
2) The first number in this list must be a prime, add it to the list of primes
3) Strike off all multiples of this number from the list
4) Repeat steps 2 through 4 until the prime being considered is greater than or equal to the square root of n
5) All remaining numbers in the list must also be primes