

Basic ideas of Programming

(...and MATLAB, a friendly place to learn and use them)

Simon Kelly



A star in the upper left corner means that the material on that slide is not specific to Matlab – the concept applies to all programming languages, only the syntax might be different

Some slides selected and adapted from course “INTRODUCTION TO COMPUTER PROGRAMMING FOR SCIENTISTS AND ENGINEERS,” (UC Berkeley E77)

<http://jagger.me.berkeley.edu/~pack/e77>

<http://creativecommons.org/licenses/by-sa/2.0/>

Matlab Environment

When you start Matlab (double-click the Matlab icon, or type `matlab` and press return in a terminal window, depending on your operating system), the one thing you'll DEFINITELY see is:

The **Command Window** and the **Matlab prompt**:

>>

The rest of the environment will depend on your version and how it's customized.

Other useful windows (that you can easily open/locate) are:

command history – remembers commands recently run

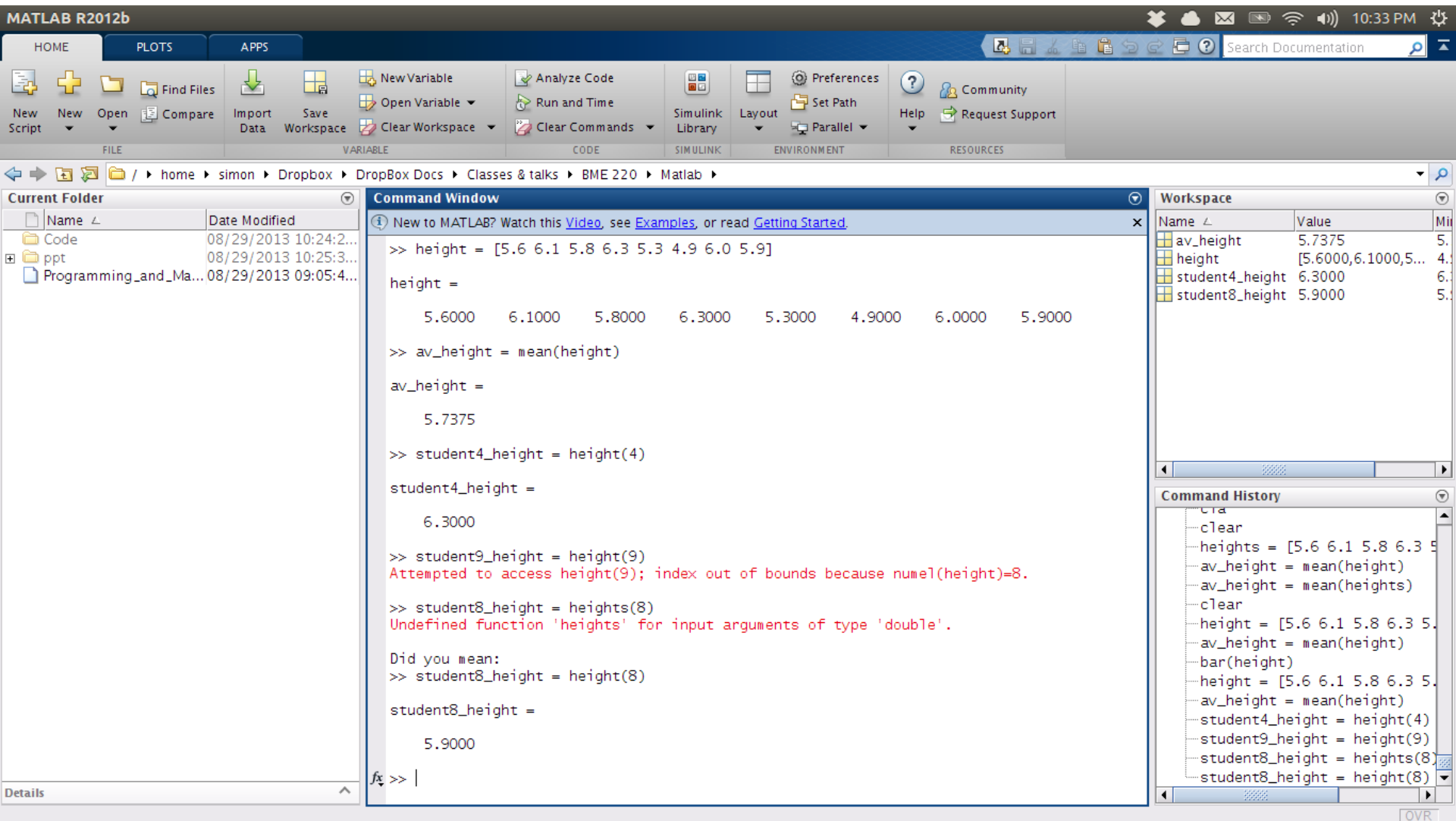
Workspace – variables you're currently working with

current directory – just like in Windows explorer, Mac Finder, you are always currently “in” a directory or folder

Editor – a text editor for writing scripts and functions

Note where the `Help` menu is – you'll need this!

The Matlab version I currently use looks like this:



Yours most likely looks different... but NO BIGGIE. The same windows, menu items, etc are there, it might just take you a few seconds to locate/identify them

Matlab as a **Calculator**

The command window of Matlab can be used simply as a **calculator**

- Type **expressions** at the `>>`, and press return
- Result is computed, and displayed and saved as a temporary variable called `ans`
 - Use numbers, `+`, `*`, `/`, `-`, `()`, `sin`, `cos`, `exp`, `abs`, `round`, ...
 - TRY IT!

```
>> sin(pi/4)
```

```
ans =
```

```
0.7071
```



Expressions have **Precedence** rules

In other words, certain things are computed before other things

- For example, how does matlab (or any language) know whether to perform the addition or the multiplication first in this expression:

```
>> 3 * 4 + 6
```

...because it has a precedence levels. Multiplication takes precedence over addition, so the answer to the above is 18, NOT 30.

Here are the levels (highest first):

1. Whatever is in parenthesis ()
2. Power (^)
3. Multiplication and division (*, /)
4. Addition and subtraction (+, -)

Within a level, Matlab goes left to right

Examples of expressions

Legal expressions

```
>> 4
>> 5 + pi
>> 6*sqrt(2)^4-12
>> 6 * sqrt(    2.0)  ^ 4 - 12
>> sin(pi/3)^2 + cos(pi/3)^2
>> 1.0/0.0
>> -4/inf
>> 0/0
```

Illegal expressions

```
>> 2 4
>> (2,4)
```

Error messages

Read them carefully – a large portion of the time you will quickly figure out what is wrong



Variables

Use names to assign result of an expression to a variable
Variables do not need to be declared before assignment

A single “equal” sign (=) is the assignment operator,

LHS = RHS

Read this as

evaluate expression on the right-hand side, and then...

assign the result to the variable named on the left-hand-side

Therefore

The right-hand-side needs to be a legal Matlab expression

The left-hand-side needs to be a single variable name

A semicolon at the end of the RHS expression suppresses the display, but the assignment still takes place.

Examples of Variables and Assignment

Legal

```
>> A = sqrt(13)
```


```
>> B = exp(2);
```

```
>> A = 2*B
```


```
>> A = A + 1
```

```
>> C = tan(pi/4)
```

Compute the square root of 13 and assign it to the variable named "A"



Make A equal to double the value of variable B. This overwrites whatever A was before



Increase the value of A by 1



Illegal (all for different reasons)

```
>> D = sqrt(E) + 1;
```

```
>> 3 = E
```

```
>> 3*A = 14
```

```
>> F = 2 3
```

Try them to see why they produce errors

The “workspace”

All variables that you create are stored in matlab's “working memory,” which is called the workspace. These are the variables that Matlab currently “knows about.”

Variables are accessed from the prompt (`>>`) or in scripts (introduced later) using their name as a reference

If you can't already see the workspace as a window in the Matlab environment, type either:

```
>> who
```

```
>> whos
```

You can clear (ie, erase) variables with the `clear` command

```
>> clear A
```

clears the variable `A` from the workspace. Just

```
>> clear
```

clears everything.



Logical Expressions and Logical Operators

Logical expressions are expressions that take the value of **true** or **false**.

For example, “the sky is blue” is a logical expression. So is $4 < 3$. The first is true, and (this might come as a shock...) the second, is false.

Logical operators are the symbols we use to form logical expressions – the ' $<$ ' sign above is one of them.

We already saw that the symbol “=” means **assign** the value of what's on the right, to the variable on the left. It's not a mathematical statement of equality like $(2+x)^2 = x^2 + 4x + 4$. Because a single = is already taken for assignment, Matlab (and many other languages) use double == to denote a logical comparison of equality.

Here's a list of logical operators:

== equals

> greater than,

< less than

>= greater than or equal

<= less than or equal

~= NOT equal to

Matlab also represents the number 1 as True and 0 as False. E.g., you could make an array of numbers representing whether or not the integers 1-6 are even:

```
>> is_even = [0 1 0 1 0 1]
```

Complex Numbers

All arithmetic in Matlab works on complex numbers as well.

When you start Matlab, two variables already exist, and are equal to $\sqrt{-1}$. They are `i` and `j`. It's common to overwrite them without even realizing, but you can always create the number with the expression `sqrt(-1)`

```
>> i
>> sqrt(-1)
>> j
>> 4 + 6j
>> 4 + 6*j
>> C = 1 - 2i;
>> real(C)
>> imag(C)
>> abs(C)
>> angle(C) * 180/pi
```

Saving the workspace

When you quit Matlab (by typing “exit” or closing the window), the variables in the workspace are erased from memory. If you need them for later use, you must save them. You can save all variables (or just some of them) to a file using the command

`save`

```
>> save
```

saves all of the variables in the workspace into a file called `matlab.mat` (it is saved in the current directory)

```
>> save Andy
```

saves all of the variables in the workspace into a file called `Andy.mat`

```
>> save Important A B C D*
```

saves the variables `A`, `B`, `C` and any variable beginning with `D` in the workspace into a file called `Important.mat`. You can load a `.mat` file later:

```
>> load Andy
```

loads all of the variables from the file `andy.mat`

Arrays

An array is a rectangular arrangement of numbers- also called a matrix. They can have more dimensions than 2, and can be just one column (a “column vector”) or just one row (a “row vector”).

For example, this is a “3 x 2” array which has 2 columns and 3 rows:

A =


2	3
5	6
3	1

>> A(2,1)

ans =

5

This is how we access the number in the first column and second row. Note round brackets and that the row number comes first – this is the “first dimension” in matlab



Creating and concatenating Arrays

Horizontal and Vertical Concatenation (ie., “stacking”)

- Square brackets, [, and] to define arrays
- Spaces (and/or commas) to separate columns
- Semi-colons to separate rows

Example

>> [3 4 5 ; 6 7 8] is the 2-by-3 array $\begin{bmatrix} 3 & 4 & 5 \\ 6 & 7 & 8 \end{bmatrix}$

If A and B are arrays with the same number of rows, then

>> C = [A B] is the array formed by stacking A “next to” B

Once constructed, C does not “know” that it came from two arrays stacked next to one another!

If A and B are arrays with the same number of columns, then

>> [A ; B] is the array formed by stacking A “on top of” B

So, [[3 ; 6] [4 5 ; 7 8]] is equal to [3 4 5 ; 6 7 8]

Creating special arrays

ones (n,m)

–a *n-by-m* `double` array, each entry is equal to 1

zeros (n,m)

–a *n-by-m* `double` array, each entry is equal to 0

rand (n,m)

–a *n-by-m* `double` array, each entry is a random number between 0 and 1.

Examples

```
>> A = ones (2,3) ;
```

```
>> B = zeros (3,4) ;
```

```
>> C = rand (2,5) ;
```

Recall: “double” just refers to a way that Matlab can store a number in memory (specifically, the precision, or how many decimal places), and is the default way – nothing mysterious!

: convention

The “: (colon) convention” is used to create row vectors, whose entries are evenly spaced.

7:2:18 equals the row vector $[7 \ 9 \ 11 \ 13 \ 15 \ 17]$

If F , J and L are numbers with $J > 0$, $F \leq L$, then $F:J:L$ creates a row vector

$[F \ F+J \ F+2*J \ F+3*J \ \dots \ F+N*J]$

where $F+N*J \leq L$, and $F+(N+1)*J > L$

Many times, the increment is 1. Shorthand for $F:1:L$ is $F:L$

The SIZE command

If A is an array, then `size(A)` is a 1-by-2 array.

- The (1,1) entry is the number of rows of A
- The (1,2) entry is the number of columns of A

If A is an array, then

`size(A,1)` is the number of rows of A

`size(A,2)` is the number of columns of A

Example

```
>> A = rand(5,6);
```

```
>> B = size(A)
```

```
>> size(A,2)
```

Accessing single elements of a vector

If A is a vector (ie, a row or column vector), then

$A(1)$ is its first element,

$A(2)$ is its second element,...

Example

```
>> A = [ 3   4.2  -7  10.1  0.4  -3.5 ] ;
```

```
>> A(3)
```

```
>> Index = 5 ;
```

```
>> A(Index)
```

This syntax can be used to assign an entry of A . Recall *assignment*

```
>> VariableName = Expression
```

An entry of an array may also be assigned

```
>> VariableName(Index) = Expression
```

So, change the 4'th entry of A to the natural logarithm of 3.

```
>> A(4) = log(3) ;
```

Accessing multiple elements of a vector

Example: Make a 1-by-6 row vector, and access multiple elements, giving back row vectors of various dimensions.

```
>> A = [ 3    4.2   -7    10.1   0.4   -3.5 ];  
>> A([1 4 6]) % 1-by-3, 1st, 4th, 6th entry  
>> Index = [3 2 3 5];  
>> A(Index) % 1-by-4
```

The `Index` or indices should be integers. You can use the `:` for indexing as well – say you wanted the first 3 elements of A:

```
>> A(1:3) % first 3 elements
```

Arrays are indexed similarly.

If M is, say 4 x 3, you can access the first two rows using

```
>> M(1:2, :)
```

`:` on its own means “ALL of this dimension”

Make vectors and arrays and play around!

See what the functions “reshape” and “repmat” do...

Unary Numeric Operations on **double** Arrays

Unary operations involve one input argument. Examples are:

- Negation, using the “minus” sign
- Trig functions, `sin`, `cos`, `tan`, `asin`, `acos`, `atan`,
...
- General rounding functions, `floor`, `ceil`, `fix`,
`round`
- Exponential and logs, `exp`, `log`, `log10`, `sqrt`
- Complex, `abs`, `angle`, `real`, `imag`

Example: If **A** is an **N1-by-N2-by-N3-by-...** array, then

B = sin(A) ;

is an **N1-by-N2-by-N3-by-...** array. Every entry of **B** is the sin of the corresponding entry of **A**. The “for”-loop that cycles the calculation over all array entries is an example of the vectorized nature of many Matlab builtin functions

Binary (two arguments) operations on Arrays

Addition (and subtraction)

- If A and B are arrays of the same size, then $A+B$ is an array of the same size whose individual entries are the sum of the corresponding entries of A and B
- If A is an array and B is a scalar, then $A+B$ is an array of the same size as A , whose individual entries are the sum of the corresponding entries of A and the scalar B
- If A is a scalar, and B is an array, use same logic as above

Scalar-Array Multiplication

- If A is an array, and B is a scalar, then $A*B$ is an array of the same size as A , whose individual entries are the product of the corresponding entries of A and the scalar B .

Element-by-Element Multiplication

- If A and B are arrays of the same size, then $A.*B$ is an array of the same size whose individual entries are the product of the corresponding entries of A and B

Matrix multiplication

- If A and B are arrays, then $A*B$ is the matrix multiplication of the two arrays... More later

Intro to plotting with Matlab

If **X** is a 1-by-N (or N-by-1) vector, and **Y** is a 1-by-N (or N-by-1) vector, then

```
>> plot(X,Y)
```

creates a figure window, and plots the data in the axis. The points plotted are

$(X(1), Y(1))$, $(X(2), Y(2))$, ..., $(X(N), Y(N))$.

By default, Matlab will draw straight lines between the data points, and the points will not be explicitly marked. For more info, do **>> help plot**

Example:

```
>> X = linspace(0,3*pi,1000);
```

```
>> Y = sin(X);
```

```
>> plot(X,Y)
```

The function **linspace(F,L,N)** creates a vector of **N** values evenly spaced between **F** and **L**.

Matlab Scripts

You don't have to enter every expression and assignment one by one in the command window.

A script is a text file containing multiple lines of code (expressions, assignments, etc) that, when you click “run” or type its name in the command window, executes all lines of the script in order, one after another.

Try typing these lines in a new “m-file” in the editor, save the m-file as runthis.m, and run it:

```
X = linspace(0,4*pi,1000) ;
```

```
Y = sin(3*X) + 2*cos(5*X) ;
```

```
plot(X,Y)
```

```
maxy = max(abs(Y)) ;
```

```
title(['Peak of Y is ' num2str(maxy)]) ;
```

Can you figure out what each line is doing?



Functions

In mathematics, a function is a rule that assigns to each value of the input, a corresponding output value.

Consider the function f defined by the rule

$$f(x) = x^2 \text{ for all numbers } x.$$

Here, x is the input value, and $f(x)$ is the output value.

Equivalently, we could have written the rule as

$$f(y) = y^2 \text{ for all numbers } y.$$

Functions can have many inputs and produce (through multiple rules) many outputs, $f_1(a,b,c) = 2a+3b$, $f_2(a,b,c) = bc$.

In *programming*, a function is like a script with inputs and outputs – It's especially useful if you want to execute certain tasks (e.g. several lines of Matlab code) repeatedly. Every time you need to execute that task, you will only need to “call” the function.

This **modularity** helps break down a huge program task into a collection of smaller tasks, which individually are easier to design, write, debug and maintain.



The first line is the *function declaration line*.

```
function [dp, cp] = vecop(v, w)
```

The output variables. This function has two. The function's purpose is to compute these variables, based on the values of the input variables.

The input and output variables are also called the input and output arguments.

The input variables. This function has two. Within the function, the names of the input variables are **v** and **w**.

The function name, this function should be saved in a file called **vecop.m**

6-line function in `vecop.m`

Function declaration line

```
function [dp,cp] = vecop(v,w)
```

```
dp = sum(v.*w);
```

```
cp = zeros(3,1);
```

```
cp(1) = v(2)*w(3) - w(2)*v(3);
```

```
cp(2) = v(3)*w(1) - w(3)*v(1);
```

```
cp(3) = v(1)*w(2) - w(1)*v(2);
```

Logically correct expressions and assignments that compute the output variables using the values of the input variables.



Comments and blank lines add readability

```
function [dp,cp] = vecop2(v,w)  
% VECOP computes dot product and cross  
% product of two 3-by-1 vectors.
```

```
dp = sum(v.*w);
```

```
cp = zeros(3,1); % create 3-by-1
```

```
% Fill cp
```

```
cp(1) = v(2)*w(3) - w(2)*v(3);
```

```
cp(2) = v(3)*w(1) - w(3)*v(1);
```

```
cp(3) = v(1)*w(2) - w(1)*v(2);
```

comments



if, end

To conditionally control the execution of statements, you can use

```
if expression  
    statements  
end
```

expression should be a numeric or logical one.

If **expression** is **true**, or is nonzero, then the statements between the **if** and **end** will be executed. Otherwise they will not be.

Execution continues with any statements after the **end**.



if, else, end

```
if exp_1  
    statements1  
else  
    statements2  
end
```

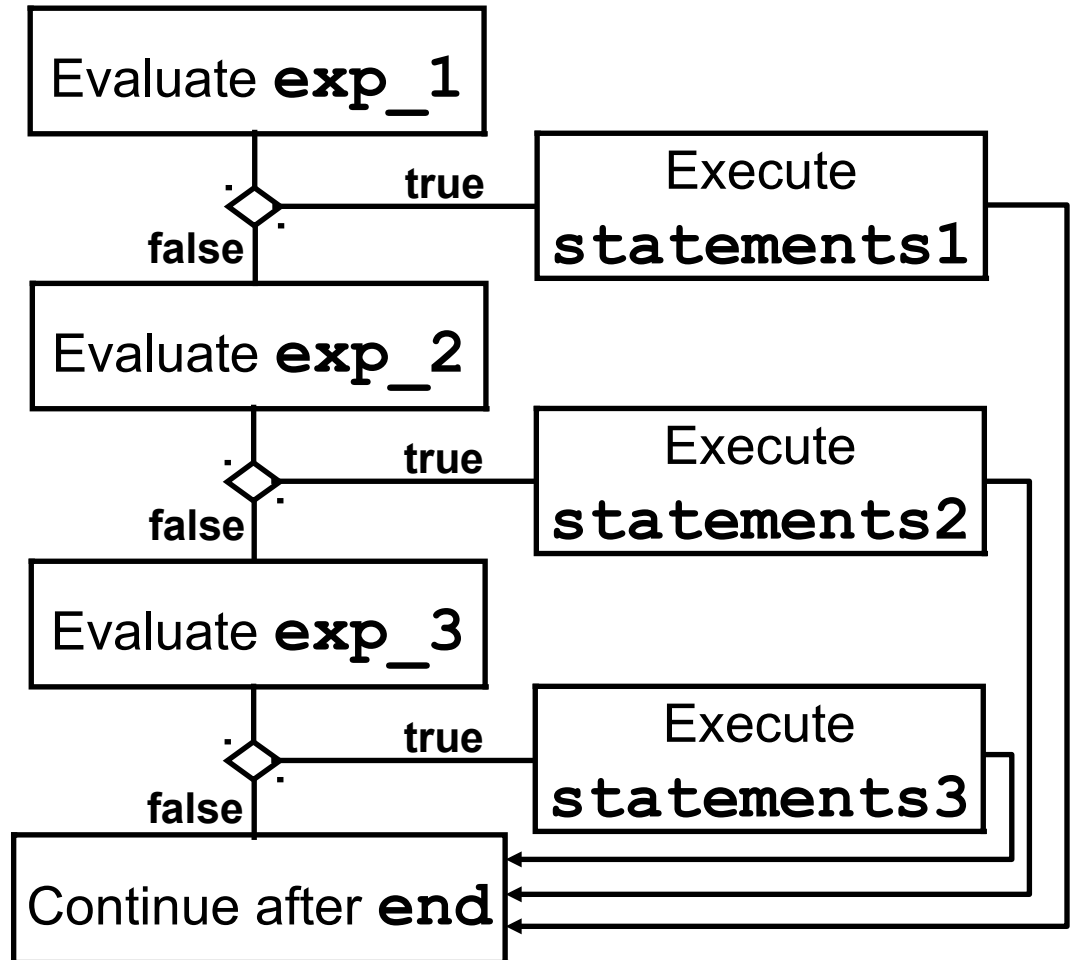
One of the sets of statements will be executed

- If **exp_1** is TRUE, then **statements1** are executed
- If **exp_1** is FALSE, then **statements2** are executed



if, elseif, end

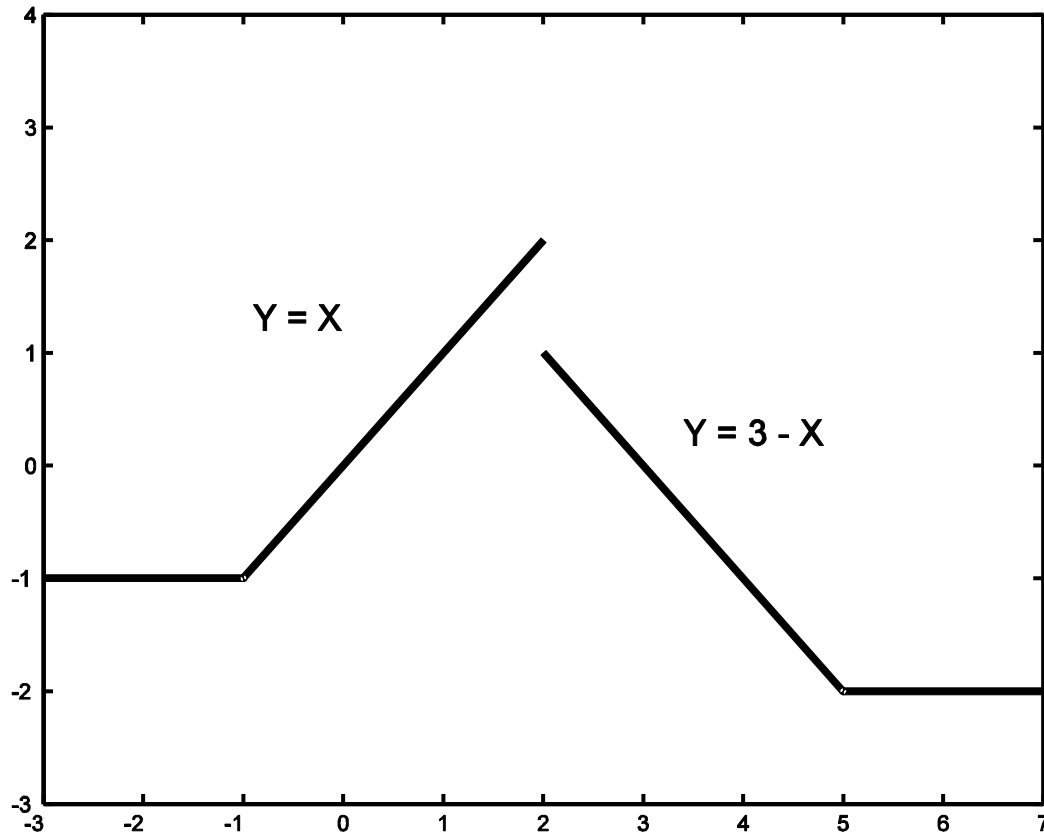
```
if exp_1
    statements1
elseif exp_2
    statements2
elseif exp_3
    statements3
end
```



Could also have an **else** before the **end**

Piecewise linear function

TASK: Create an m-file function for the mathematical function $Y = F(X)$ shown below.



```
function y =  
plinear(x)  
  
    if x < -1  
        y = -1;  
    elseif x < 2  
        y = x;  
    elseif x < 5  
        y = 3 - x;  
    else  
        y = -2;  
    end
```



for, end

We use a **for** loop to execute collection of statements a fixed number of times.

loopvariable → **for** **n=1:12** *controlvalue*
 statements
end

This is a loop that executes the **statements** as many times as there are elements in the controlvalue.

Before each “execution pass,” the *loopvariable* (**n**) is assigned to the corresponding element of *controlvalue*. The first time through, n is 1, the second time, n is 2, and so on.

The loop variable doesn't have to step in unit increments – you can have

```
for m=[5 3 8 6]  
end
```




While loop

Executing commands an undetermined number of times.

```
while expression
    statements
end
```

```
while expression
    statements
end
```

Evaluate **expression**

If TRUE, execute **statements**

If FALSE, jump past end



Example using **tic/toc**

tic is a built-in Matlab function that starts a timer.

Every subsequent call to **toc** (also Matlab built-in) returns the elapsed time (in seconds) since the originating call to **tic**.

The code below will cause the program to “pause” for one second before proceeding.

```
tic  
while toc<1  
end
```

It would be clumsy to do this without a **while**-loop