Lecture 4 Week 4

Mohamad Nassereddine

Book Reference

The online textbook used in the Tutorials is: B. Hahn & D. Valentine, "Essential MATLAB for Engineers and Scientists", 7th Edition, Academic Press/Elsevier, 2019.

Chapters 3 & 4

Structure Plan

- This lecture is an introduction to the design of computer programs. The top-down design process is elaborated to help you think about good problem solving strategies as they relate to the design of procedures for using software like MATLAB
- In this lecture we look at one, the *structure plan*. Its development is the primary part of the software (or code) design process because it is the steps in it that are translated into a language the computer can understand—for example, into MATLAB commands.

Structure Plan

- When you run a script, the code is executed from top to bottom. Hence the second line of code will not be run until the first line has been run and so forth (seems obvious, but order is important!)
- Scripts can be saved. If you write the name of the saved script in the command window (or in another script), the code in the script will be run. The code is therefore re-useable, you don't have to type it over an over again.

Structure Plan: Sequencing

- When solving a problem with sequential flow, it is important to list the series of steps that need to be accomplished.
- The order of events can be extremely important
- Incorrect order can produce failure or errors

As an example, state the steps required for a person to complete from wake up to reaching the university

Structure Plan: Sequencing

- As an example, let us execute the following two examples:
- % Example 1
- a=6; % non-hypotenuse side of right angled triangle
- b=8; % another non-hypotenuse side of right angled triangle
- c=sqrt(a^2+b^2); % calculate hypotenuse
- a=2; % update a
- disp(c); % display answer to the user
- % Example 2
- a=6; % non-hypotenuse side of right angled triangle
- b=8; % another non-hypotenuse side of right angled triangle
- a=2; % update a
- c=sqrt(a^2+b^2); % calculate hypotenuse
- disp(c); % display answer to the user

Structure Plan: Sequencing

- The design process steps may be listed as follows:
- **Step 1** *Problem analysis.* The designer must fully recognize the need and must develop an understanding of the nature of the problem to be solved.
- **Step 2** *Problem statement.* Develop a detailed statement of the mathematical problem to be solved with a computer program.
- **Step 3** *Processing scheme.* Define the inputs required and the outputs to be produced by the program.
- **Step 4** *Algorithm.* Design the step-by-step procedure in a *top-down* process that decomposes the overall problem into subordinate problems. The subtasks to solve the latter are refined by designing an itemized list of steps to be programmed.
- **Step 5** *Program algorithm.* Translate or convert the algorithm into a computer language (e.g., MATLAB) and debug the syntax errors until the tool executes successfully.
- **Step 6** *Evaluation.* Test all of the options and conduct a validation study of the program.
- **Step 7** *Application.* Solve the problems the program was designed to solve. If the program is well designed and useful, it can be saved in your working directory (i.e., in your user-developed toolbox) for future use.

Structure Plan: Conditional

- Morning routine
- 1)Wake up and check day
  - 2)IF WORKDAY1)Turn off alarm
  - 2)Get out of bed
  - 3)Have a shower
  - 4)Get dressed
  - 5)Eat breakfast
  - 6)Brush teeth
  - 7)Go to uni/work

3)IF WEEKEND1)Go back to sleep…

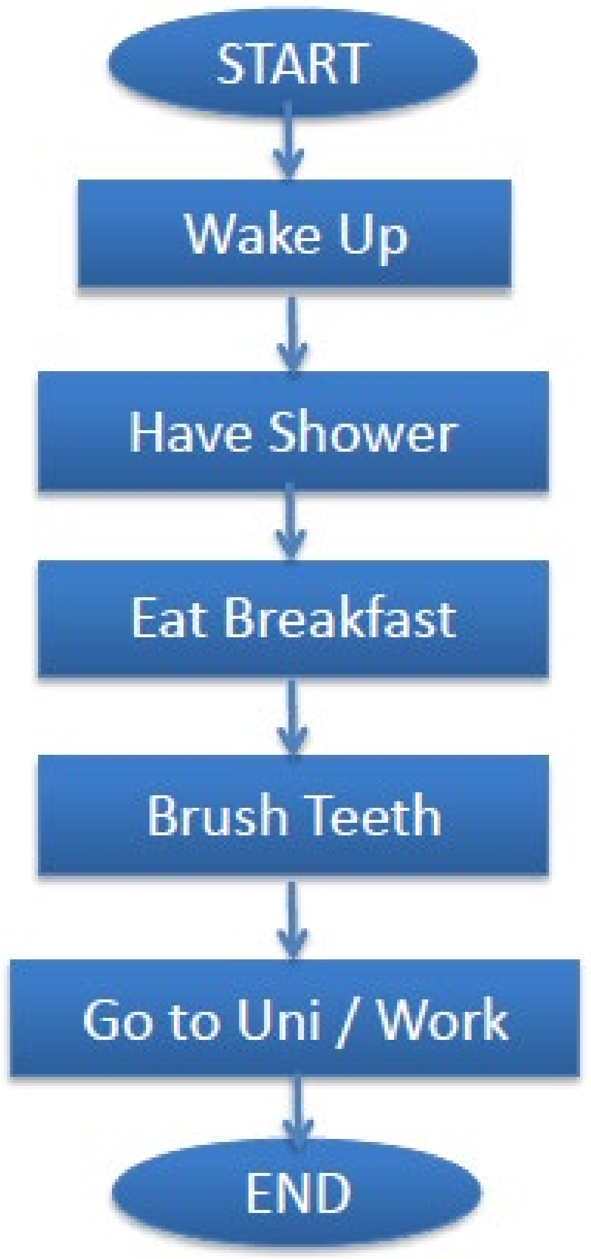
Structure Plan: Repetitive

- Gym routine
- 1)Warm up & stretch
- 2)Choose a workout
- 3)Complete a set
  - 4)Finished the workout? If no, go to 3)
  - If yes, continue
- 5)Take a small break
- 6)Next workout…

Structure Plan: Sequential Flow

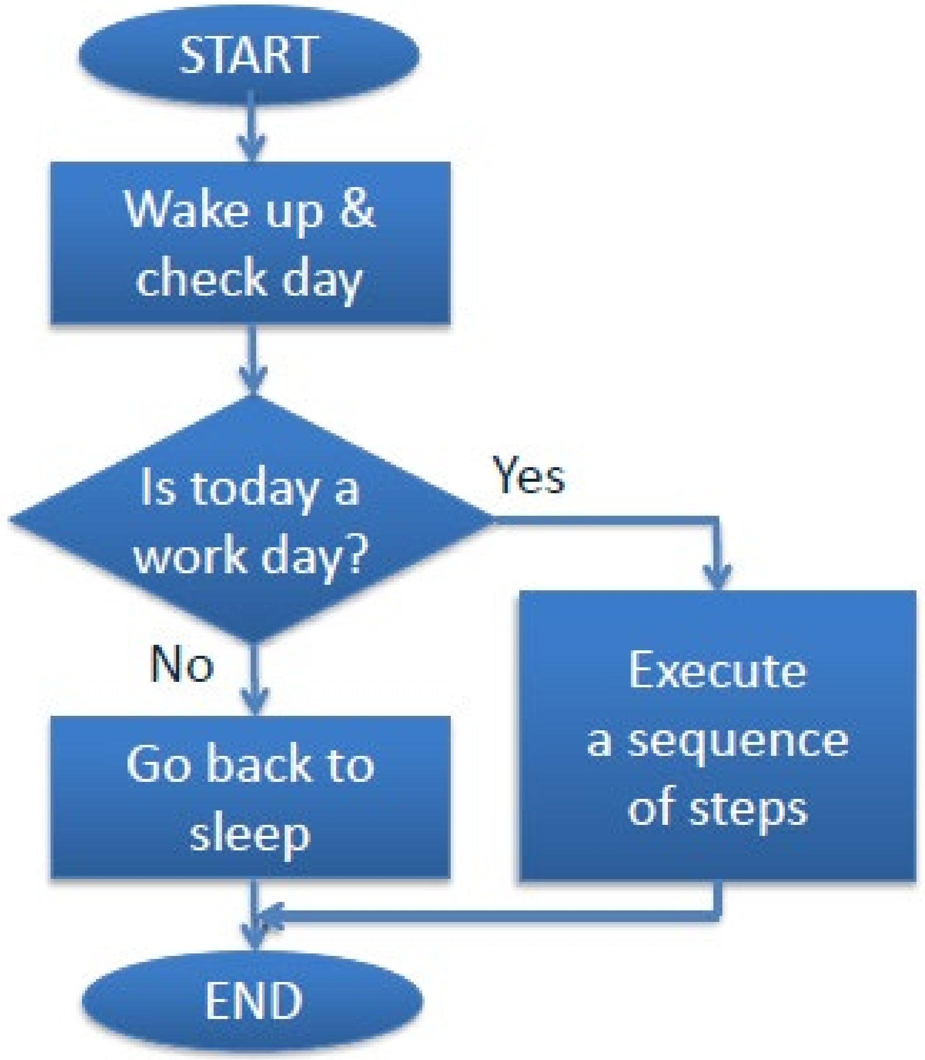
- Using a similar set of steps to the ‘morning routine’ example earlier, we can see this program is ***sequential*** in nature
  - It is a sequence
  - Only one direction

- •The flowchart clearly shows only one path from START to END



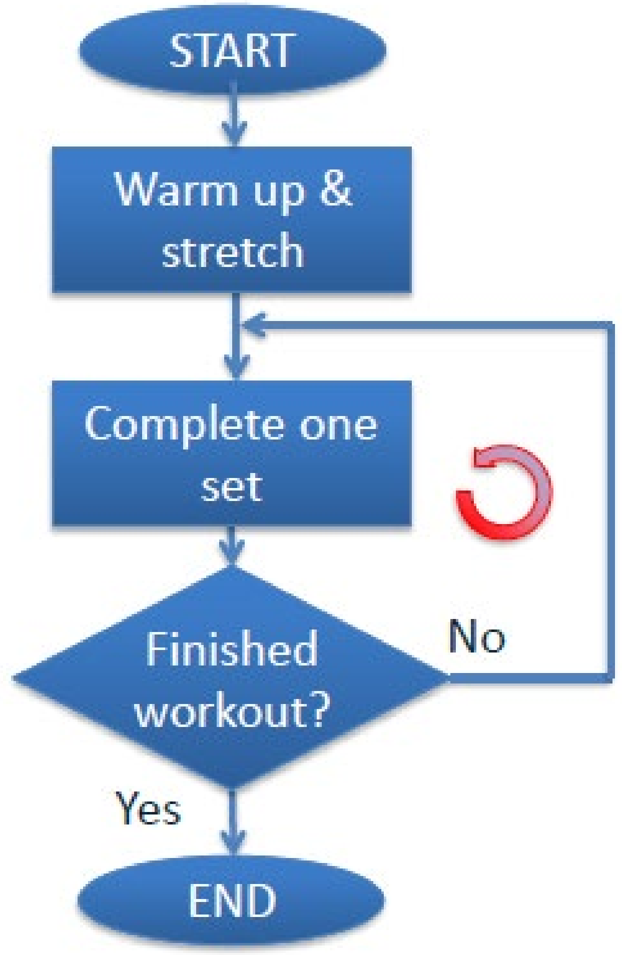
Structure Plan: Conditional Flow

- Where conditions are tested, there must be multiple paths
- The decision diamond allows for two paths a ‘Yes’ and ‘No’ path (true/false)



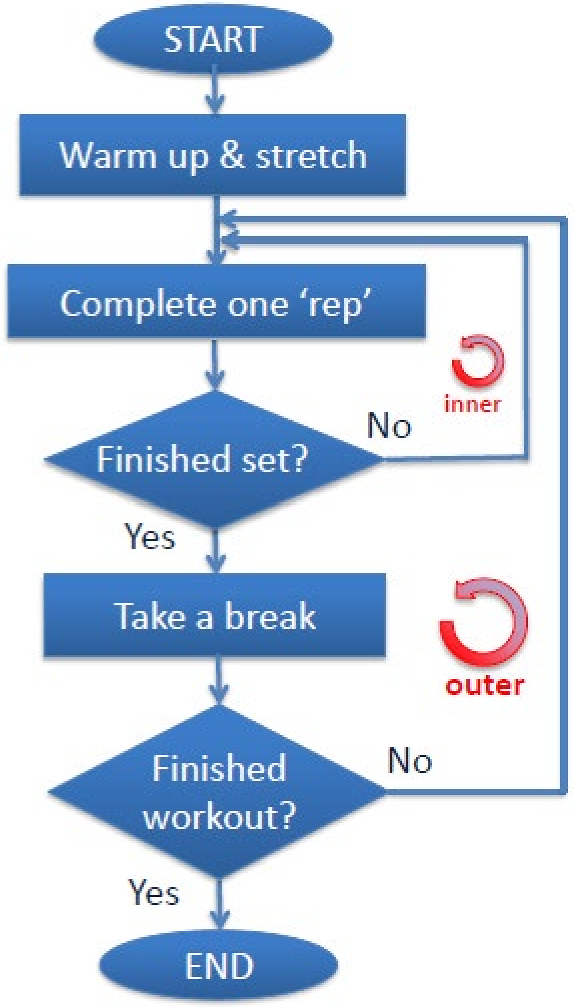
Structure Plan: Repetitive Flow

- The reason repetitive flow is usually called programming ***loops***, is because a circular pattern is created in the flow chart
- The program continues in the circular pattern until the condition is met to ***exit*** the ***loop***
- Next week we will look at **for** loops and **while** loops



Structure Plan: Nested Repetition

- We can 'nest' loops inside other loops
- Like the 'sets' and 'reps' in the gym example
- In this example, we call the 'reps' the ***inner loop***
- And the 'sets' are the ***outer loop***



Structure Plan: Boolean logic

- Boolean (logical) statements can only have one of two results: True (1) or False (0).
- The statements generally contain logical operators:
- == (True if equal to)
- > (True if greater than)
- < (True if less than)
- >= (True if greater than or equal to)
- <= (True if less than or equal to)
- ~= (True if not equal to)

Structure Plan: Boolean logic

- Example:
- What does each line do?
- a=1;
- b=6;
- c=a==b;
- disp(a~=b)
- disp(a>=b)
- True (1) or False (0)

Structure Plan: Conditional Statement

- The most popular condition statement is the ‘if’ statement. A Boolean statement must be used to define the condition.
- To create an if statement, use the following structure:
- **if condition**
  - **code that will run if the condition is true**
- **end**

- **This line will run irrespective of the condition being true/false.**

Structure Plan: Conditional Statement

- The most popular condition statement is the ‘if’ statement. A Boolean statement must be used to define the condition.
- To create an if statement, use the following structure:
- **if condition**
  - **code that will run if the condition is true**
- **end**

- **This line will run irrespective of the condition being true/false.**

Structure Plan: Conditional Statement

- These **if** statements can also include an **else** statement.
- The Code between **else** and **end** will only run if the condition is false.
- Indentation is useful to see which **else** and **end** corresponds with an **if**.
- Matlab Example

Structure Plan: Conditional Statement

- These **if** statements can also include an **else** statement.
- The Code between **else** and **end** will only run if the condition is false.
- Indentation is useful to see which **else** and **end** corresponds with an **if**.
- Matlab Example

Structure Plan: Conditional Statement

- More option:
- Matlab Example

Structure Plan: Conditional Statement

- More option:
- randi() -Uniformly distributed pseudorandom integers
- switch-Execute one of several groups of statements
- case-The switch block tests each case until one of the case expressions is true.
- otherwise-The otherwise block is optional. MATLAB executes the statements only when no case is true.

Example

- % What is the output? (notice the indentation)
- a=7;
- if a == 6
  - Disp ('Hey');
  - If a > 5
  - Disp ('It''sa trap!');
- else
- Disp ('Never mind');
- end
- else
- Disp ('This one?');
- if a <= 10
- Disp ('This one too?');
- Else if a ~= 4
- Disp ('This one three?');
- else
- disp('Time to give up.');
- end
- end

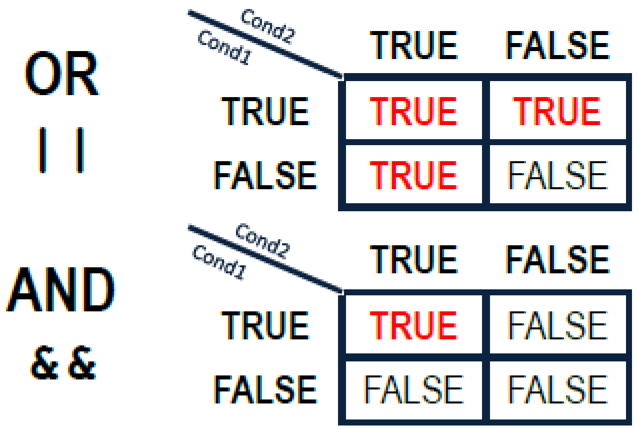
Structure Plan: Combining Condition

- Real world examples will usually have many more variables and conditions to test, so we need to have a way of combining these
  - We can use nested **if**s
- We can combine conditions using logical operators

- Let’s look at an example. A thermostat must trigger an alarm when the temperature of a machine part **either** drops below 10°C or rises above 40°C.

Structure Plan: Combining Condition

- MATLAB includes logical operators based on the English meanings of ‘OR’ and ‘AND’

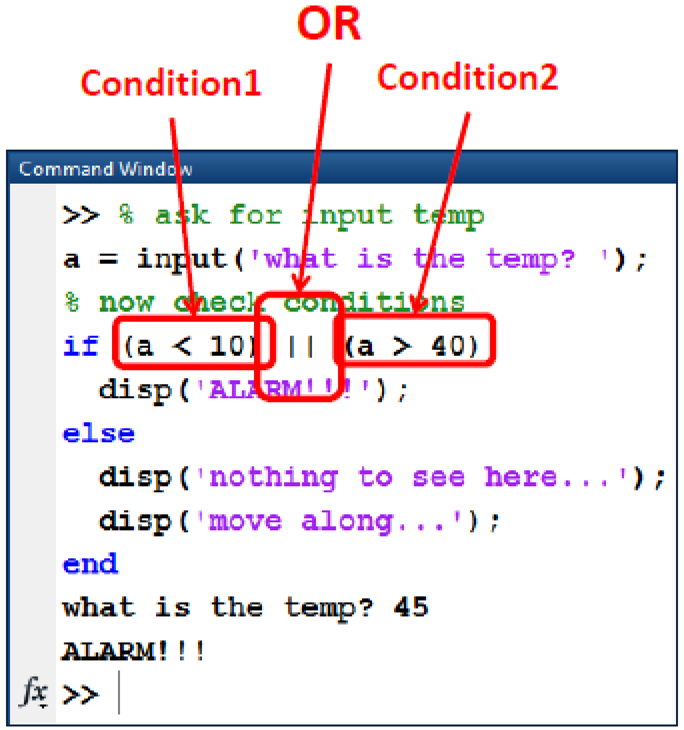


OR
| |

Cond1 \ Cond2	TRUE	FALSE
TRUE	TRUE	TRUE
FALSE	TRUE	FALSE

AND
& &

Cond1 \ Cond2	TRUE	FALSE
TRUE	TRUE	FALSE
FALSE	FALSE	FALSE

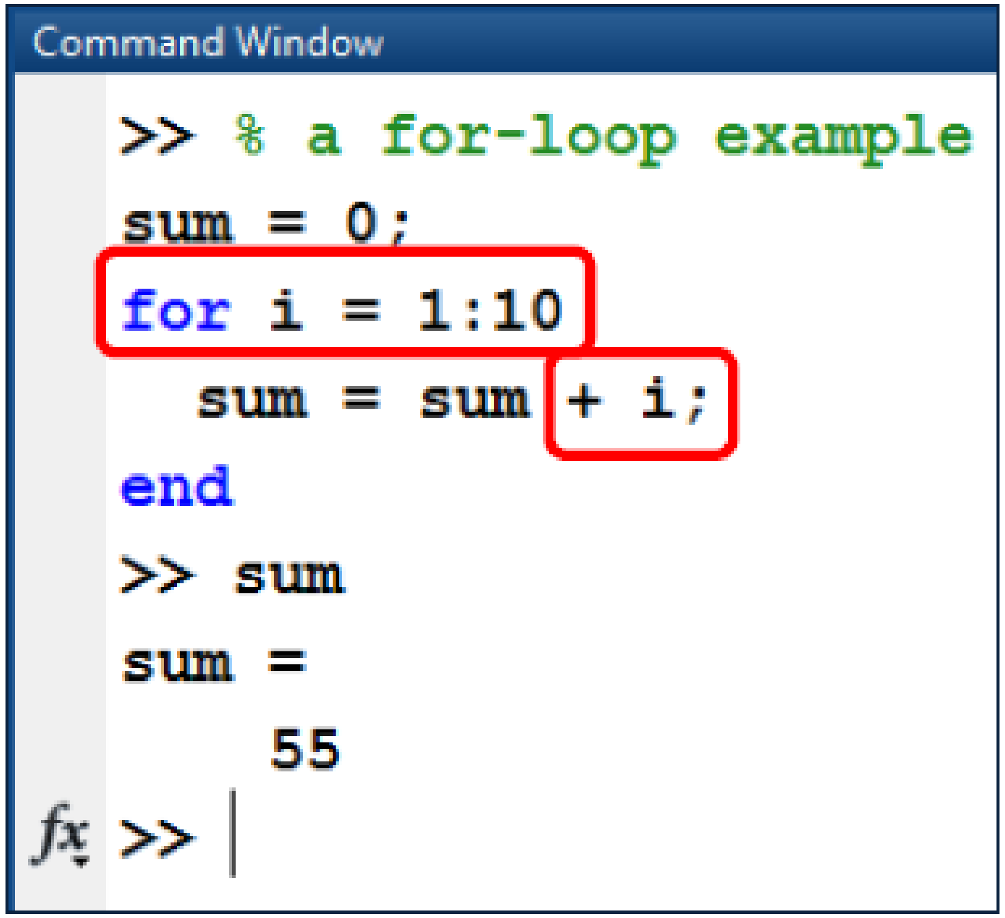


```
>> % ask for input temp
a = input('what is the temp? ');
% now check conditions
if (a < 10) || (a > 40)
  disp('ALARM!!!');
else
  disp('nothing to see here...');
  disp('move along...');
end
what is the temp? 45
ALARM!!!
>>
```

Structure Plan: Combining Condition

- The most common repetitive construct is the **for** loop
- The for loop sets out a determined number of loop iterations
- In this example, the index **I** will take on each of the integer values between 1 and 10 (inclusive)

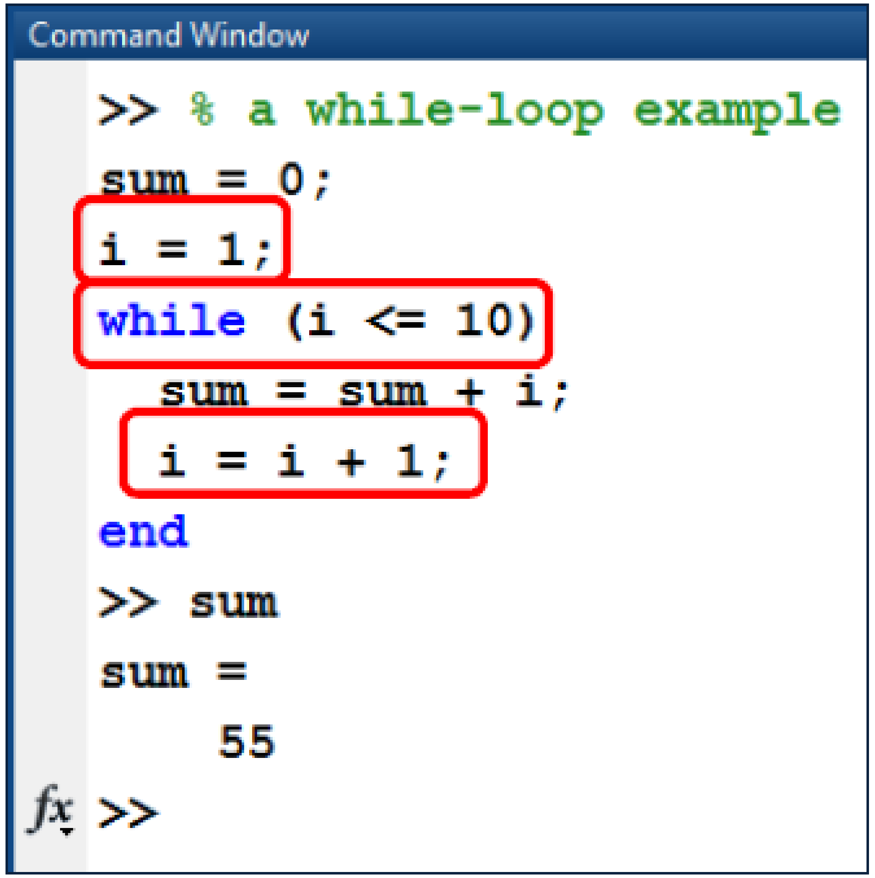
- The most common repetitive construct is the **for** loop
- The for loop sets out a determined number of loop iterations
- In this example, the index **I** will take on each of the integer values between 1 and 10 (inclusive)



```
>> % a for-loop example
sum = 0;
for i = 1:10
   sum = sum + i;
end
>> sum

sum =

    55

>> 
```

Structure Plan

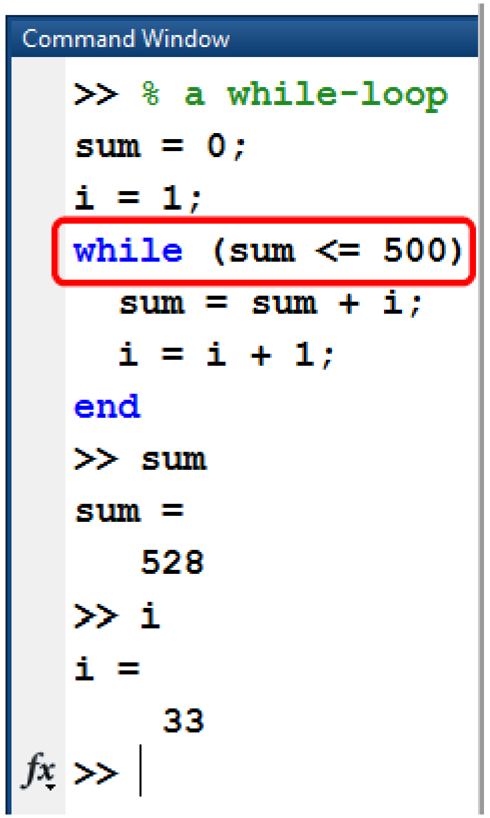
- The same task can be accomplished with a **while**-loop
- Simply keep looping **while**
  - i<= 10
- When using while, we need to manually increment the index I every iteration
- Every **for**-loop can be expressed as a **while**-loop



```
>> % a while-loop example
sum = 0;
i = 1;
while (i <= 10)
  sum = sum + i;
  i = i + 1;
end
>> sum

sum =

    55

>>
```

Structure Plan

- The true advantage of while loops is when looping for an *undetermined* number of iterations
- Keep performing an operation until some condition changes
- eg. continually check for when the sum exceeds the value 500
- Notice that the condition is checked at the start of the loop.
- The last iteration will increase **sum** to above 500.



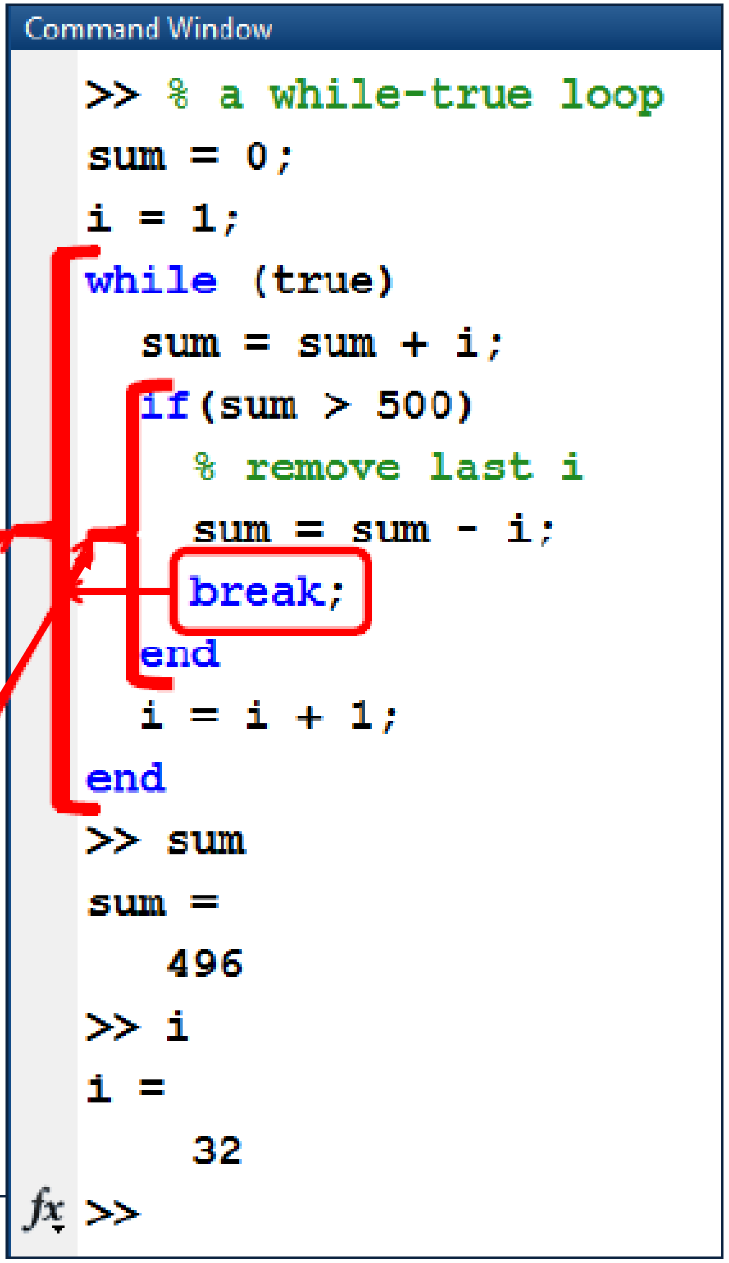
```
>> % a while-loop
sum = 0;
i = 1;
while (sum <= 500)
  sum = sum + i;
  i = i + 1;
end
>> sum
sum =
   528
>> i
i =
    33
>> 
```

Structure Plan

- In order to reverse the last **I** added to **sum**, we can create a condition in the middle of our loop to remove the last **i**
- Forever loops can be quite dangerous if exit conditions are not fully understood

this while(true) loop will never end by itself

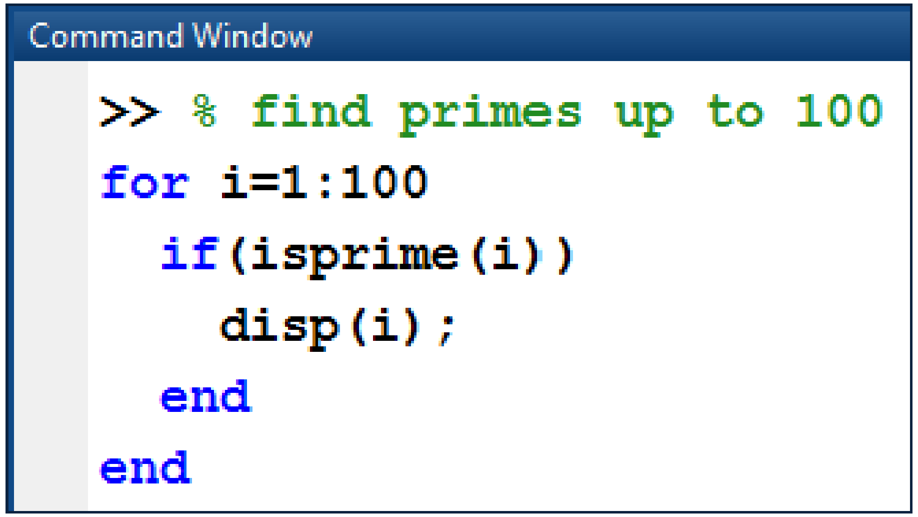
internal if-statement:
before sum>500, this code ignored
once sum>500, execute 2 lines



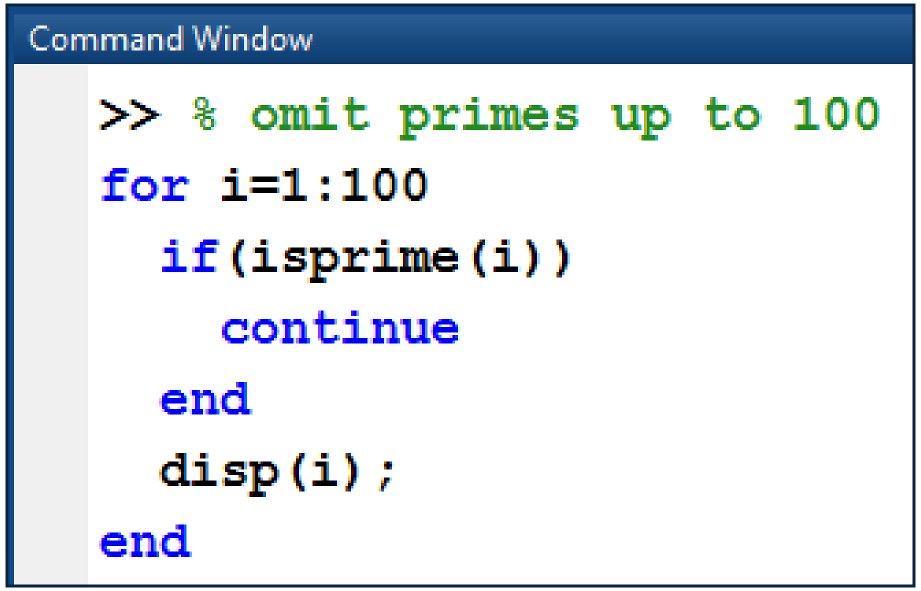
```
>> % a while-true loop
sum = 0;
i = 1;
while (true)
  sum = sum + i;
  if(sum > 500)
    % remove last i
    sum = sum - i;
    break;
  end
  i = i + 1;
end
>> sum

sum =

   496

>> i

i =

    32

>>
```

Structure Plan

- Another irregular breaking mechanism is to use **continue** to break out of the *current iteration* In the first example, we list all numbers that are prime
- In the second example, we list all numbers that are NOT prime



```
>> % find primes up to 100
for i=1:100
  if(isprime(i))
    disp(i);
  end
end
```



```
>> % omit primes up to 100
for i=1:100
  if(isprime(i))
    continue
  end
  disp(i);
end
```

Functions and Data

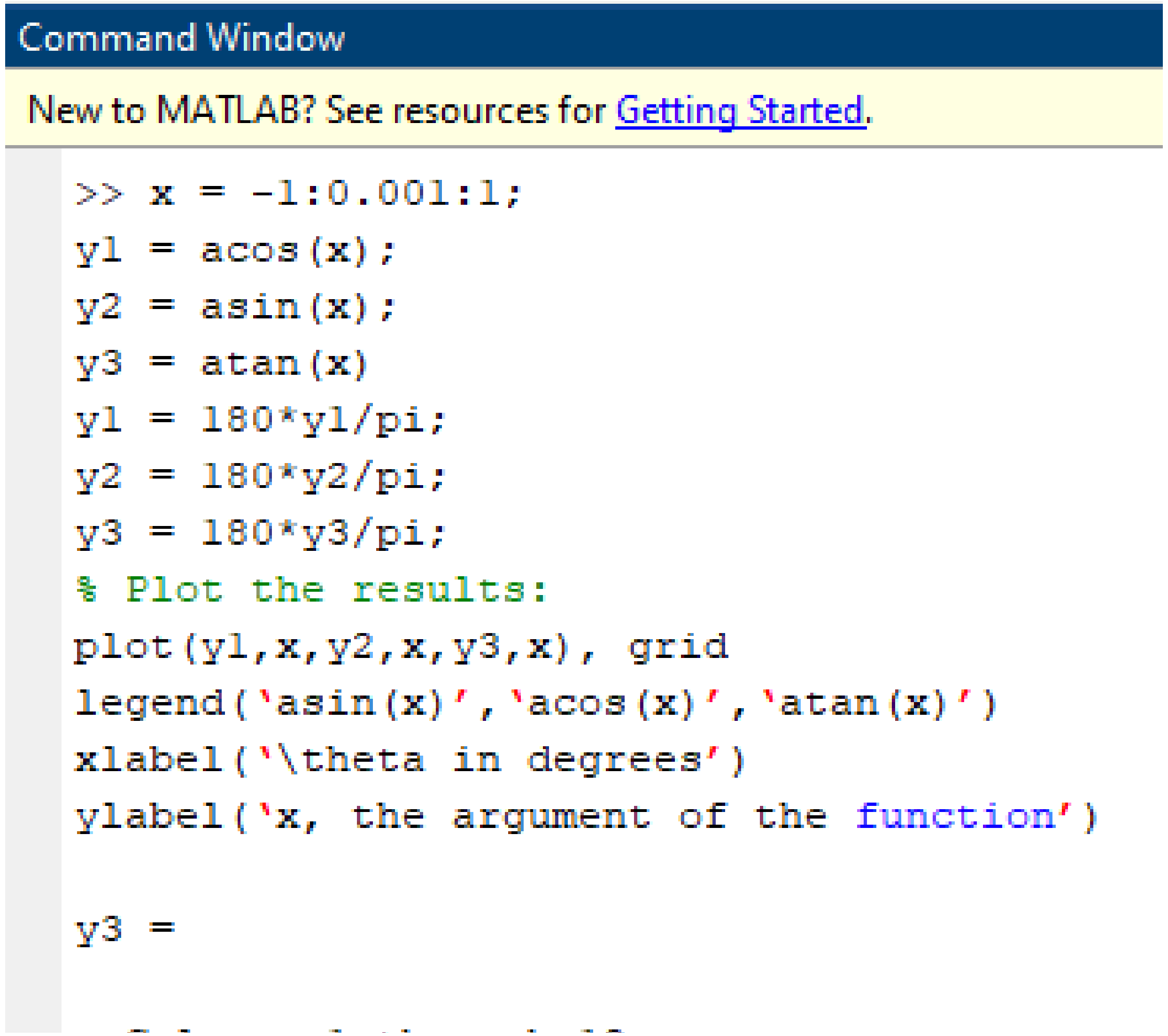
- Let us next examine an example that will hopefully inspire you to examine all of the functions listed as well as any other function that you may discover that MATLAB supports.
- Let us consider the arc-cosine, the arc-sine and the arc-tangent functions, i.e., acos(x), asin(x) and atan(x), respectively.
- If you specify x, i.e., the cosine, the sine and the tangent, respectively, between −1
- and 1, what quadrant of the circle are the output angles selected?
- To provide an answer, the following *M-file* script was created and executed. The graphical comparison of the computed results is illustrated in Fig. 4.2. The REMARKS at the end of the script provides an interpretation of the graphical results and, hence, an answer to the question raised.

Functions and Data

- % The question raised is: What range of angles, i.e.,
- % which of the four quadrents of the circle from 0 to
- % 2*pi are the angular outputs of each of the functions?
- % Assign the values of x to be examined:
- x = -1:0.001:1;
- % Compute the arc-functions:
- y1 = acos(x);
- y2 = asin(x);
- y3 = atan(x);

Functions and Data

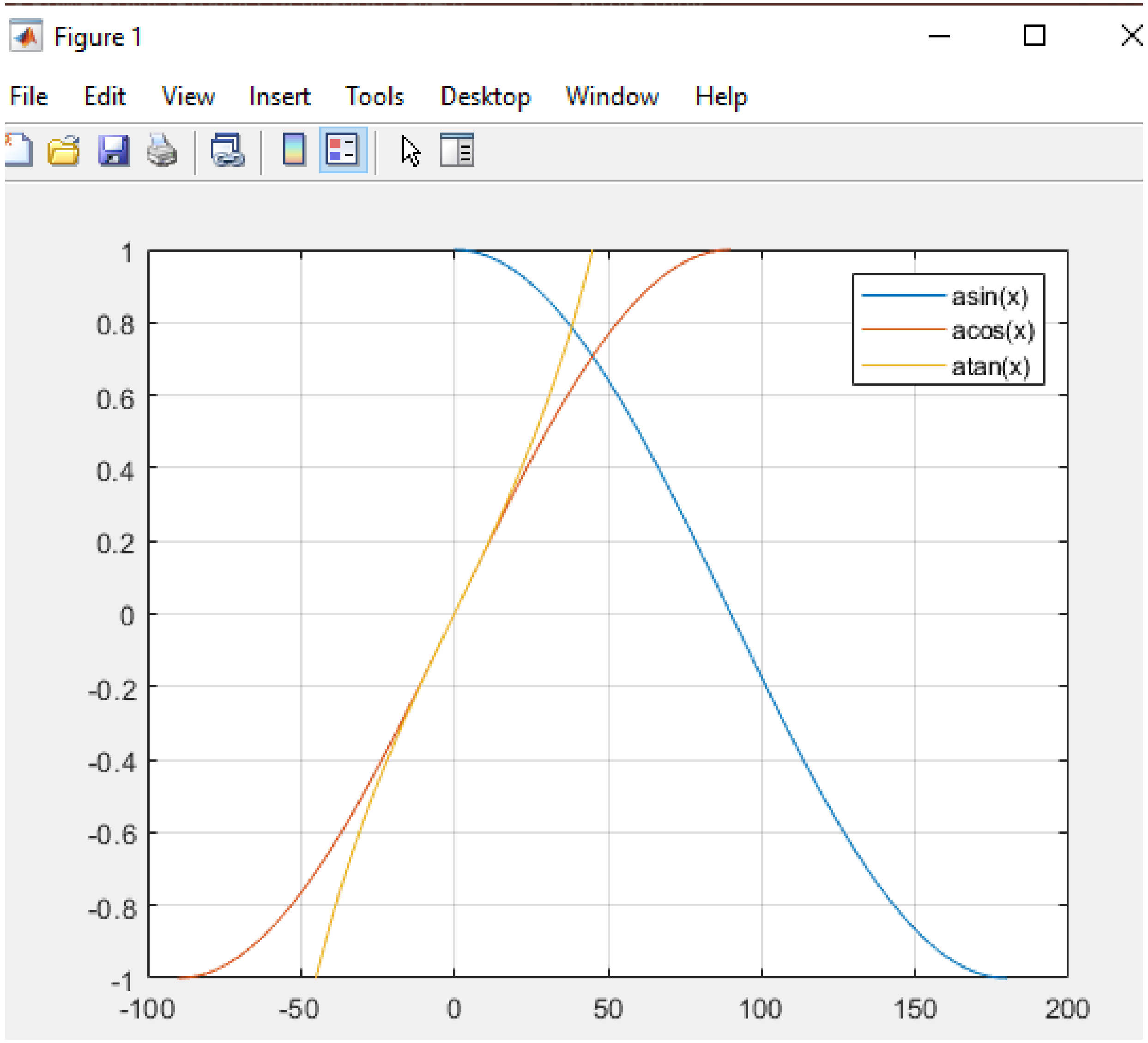
- % Convert the angles from radians to degrees:
- y1 = 180*y1/pi;
- y2 = 180*y2/pi;
- y3 = 180*y3/pi;
- % Plot the results:
- plot(y1,x,y2,x,y3,x),grid
- legend('asin(x)', 'acos(x)', 'atan(x)')
- xlabel('\theta in degrees')
- ylabel('x, the argument of the function')
- % REMARKS: Note the following:
- % (1) The acos(x) varies from 0 to 90 to 180 degrees.
- % (2) The asin(x) varies from -90 to 0 to 90 degrees.
- % (3) The atan(x) varies from -90 to 0 to 90 degrees.
- % To check remark (3) try atan(10000000) *180/pi.
- % Stop



Command Window

New to MATLAB? See resources for Getting Started.

```
>> x = -1:0.001:1;
y1 = acos(x);
y2 = asin(x);
y3 = atan(x)
y1 = 180*y1/pi;
y2 = 180*y2/pi;
y3 = 180*y3/pi;
% Plot the results:
plot(y1,x,y2,x,y3,x), grid
legend('asin(x)','acos(x)','atan(x)')
xlabel('\theta in degrees')
ylabel('x, the argument of the function')

y3 =
```



Import and Export data

- When you get into serious programming you will often need to store data on a disk. The process of moving data between MATLAB and disk files is called importing (from disk files) and exporting (to disk files).
- Data are saved in disk files in one of two formats: text or binary. In text format, data values are ASCII codes, and can be viewed in any text editor. In binary format, data values are not ASCII codes and cannot be viewed in a text editor. Binary format is more efficient in terms of storage space required.