CE 209
Computation for Civil Engineers

2019–2020 Fall

Part II

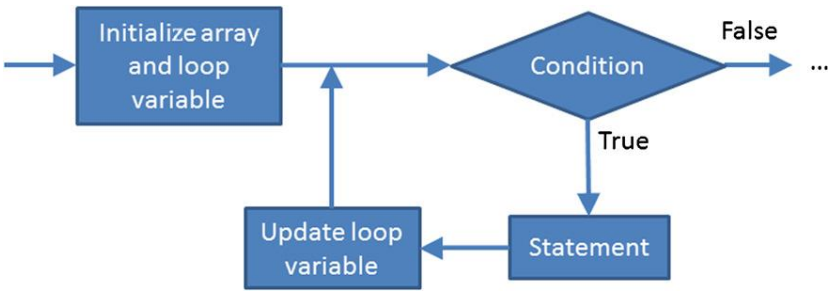
Textbooks

[Nagar 2018] Introduction to Octave for Engineers and Scientists.

[Farrell 2015] Programming Logic and Design - Comprehensive Version 8e.

For Loop

- In addition to conditional statements, the second fundamental type of programming construct that we will consider here is ***the iteration statement***. Iteration statements are intended for use in situations in which one or more operations need to be repeatedly performed a number of times. In computer programming, iteration statements are often known as ***loop*** statements.
- There are two types of iteration statements: ***for loops*** and ***while loops***:
- ***for loops*** can be useful when we want to execute some statements a fixed number of times.

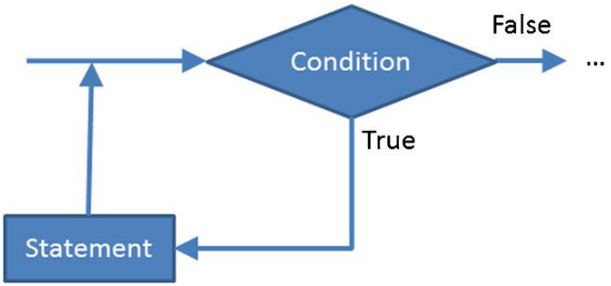


- Consider the following example. This code reads in a number from the user and computes its factorial.

```
% ask user for input
n = input('Enter number:');
f = 1;
if (n >= 0)
    for i = 2:n
          f = f * i;
    end
    disp(['The factorial of ' num2str(n) ' is ' num2str(f)]);
else
    disp(['Cannot compute factorial of the number']);
end
```

While Loop

- The second type of iteration statement available in Octave is the while loop. The control flow of a while loop is shown below. First, a condition is tested. If this condition evaluates to false then control immediately passes beyond the while statement. If it evaluates to true then a statement (or sequence of statements) is executed and the condition is retested. This *iteration* continues until the condition evaluates to false. Therefore, a while loop will execute continually until some condition is no longer met. *This type of behavior can be useful when continued loop execution depends upon some computation or input that cannot be known before the loop starts*.



- Consider the following example. This program first generates a random integer between 0 and 9. The Octave **randi** statement, used in this way, generates a random integer from a uniform distribution between 1 and its argument (in this case, 10), so by subtracting 1 from this we can get a random integer between 0 and 9.
- The following while loop will iterate whilst the condition **guess ~= i** is true. In other words, the loop will execute so long as guess is not equal to the random integer i. Since guess is initialized to −1 before the loop starts, the loop will always execute at least once.
- Inside the loop, a number is read from the user, and an if statement is used to display an appropriate message depending upon whether the guessed number is correct. The program will continue asking for guesses *until the correct answer is entered.*

```
i = randi(10) -1; % random integer between 0 and 9
guess = -1;
while (guess ~= i)
     guess = input('Guess a number:');
     if (guess == i)
         disp('Correct!');
     else
        disp('Wrong, try again ...');
     end
end
```

Break and Continue Statements

- Another way of altering the control flow of a program is to use jump statements. The effect of a jump statement is to unconditionally transfer control to another part of the program.
- Octave provides two different jump statements: break and continue. Both can only be used inside for or while loops.
- The effect of a break statement is to transfer control to the statement immediately following the enclosing control structure. Consider the following code.

```
total = 0;
while (true)
       n = input('Enter number: ');
       if (n < 0)
           disp('Finished!');
           break;
       end
       total = total + n;
end
disp(['Total = ' num2str(total)]);
```

- This piece of code reads in a sequence of numbers from the user. When the user types a negative number the loop is terminated by the break statement. Otherwise, the current number is added to the total variable. When the loop terminates (i.e. a negative number is entered), the value of the total variable, which is the sum of all of the numbers entered, is displayed.
- The continue statement is similar to the break statement, but instead of transferring control to the statement following the enclosing control structure, it only terminates the current iteration of the loop. *Program execution resumes with the next iteration of the loop*.

```
total = 0;
for i = 1:10
      n = input('Enter number: ');
      if (n < 0)
          disp('Ignoring!');
          continue;
      end
      total = total + n;
end
disp(['Total = ' num2str(total)]);
```

- A sequence of numbers is again read in and summed. However, this time there is a limit of 10 numbers, and negative numbers are ignored. If a negative number is entered, the continue statement causes execution to resume with the next iteration of the for loop.

Nested Loops

- Any of the control structure statements we have covered so far can be nested. This simply means putting one statement inside another one.

```
n = input('Enter number: ');
while (n >= 0)
       f = 1;
       for i = 2:n
           f = f * i;
       end
       disp(f);
       n = input('Enter number: ');
end
```

- Here, we have a for loop nested inside a while loop. The while loop reads in a sequence of numbers from the user, terminated by a negative number. For each positive number entered, the for loop computes its factorial, which is then displayed.

Some Excersises

Question 1

A shop is having a five-day sale.
Each day, starting on Monday, the price will drop 10% of the previous day's price.

Example:
If the original price of a product is 20.00 TRY;

- On Monday the sale price would be 18.00 TRY (10% less than the original price).
- On Tuesday the sale price would be 16.20 TRY (10% less than Monday).
- On Wednesday the sale price would be 14.58 TRY (10% less than Tuesday).
- On Thursday the sale price would be 13.12 TRY (10% less than Wednesday).
- On Friday the sale price would be 11.81 TRY (10% less than Thursday).

Write the Octave code that will calculate and list the price of an item ***for each and every one of the five days***, given the original price.

The above list is an example only. Your code should work with any original price of a product that the user enters.

Solution:

Remember the solution to the factorial problem, which was given previously as follows:

```
n = input('Enter number:');
f = 1;
for i = 2:n
    f = f * i;
end
disp(['The factorial of ' num2str(n) ' is ' num2str(f)]);
```

The solution to the problem is similar, but much simpler: We always multiply with 0.9:

```
n = input('Enter the price:');
for i = 1:5
    n = n * 0.9 ;
    disp(['The price at day ' num2str(i) ' is ' num2str(n) ' TRY' ]);
end
```

The above solution is sufficient for getting full marks. But if you want to display the day name properly, you modify the code as follows:

```
n = input('Enter the price:');
for i = 1:5
    n = n * 0.9 ;
    if (i==1) dayName = 'Monday'    ; end
    if (i==2) dayName = 'Tuesday'   ; end
    if (i==3) dayName = 'Wednesday' ; end
    if (i==4) dayName = 'Thursday'  ; end
    if (i==5) dayName = 'Friday'    ; end
    disp(['The price on ' dayName ' is ' num2str(n) ' TRY' ]);
end
```

Question 2

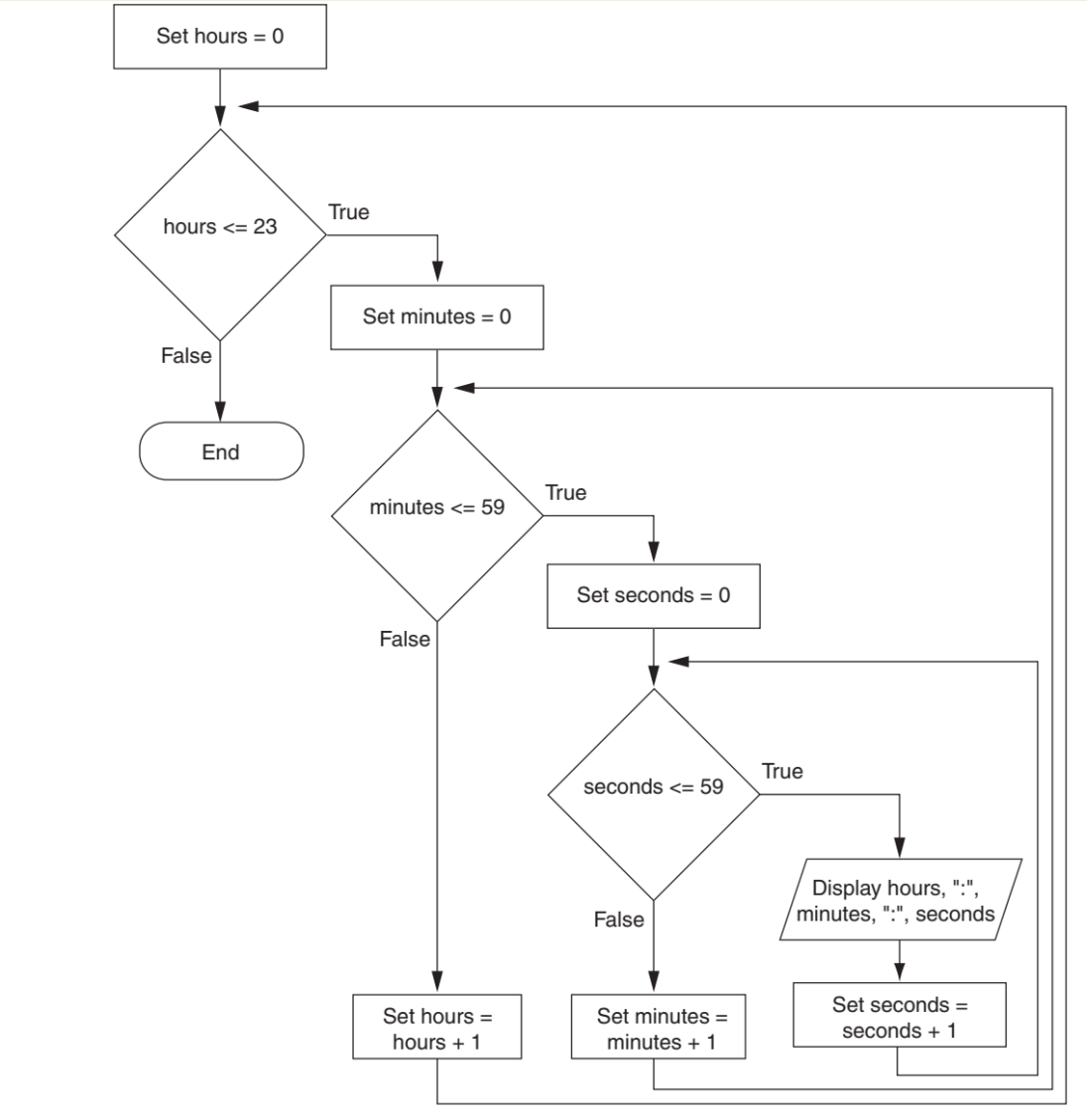
A clock simulator will be written. The program will display the following output:

```
0:0:0
0:0:1
0:0:2
.
.
23:59:59
```

a) Draw the flowchart for this program.
b) Write Octave code.

Solution:

The flowchart:



The code:

```
for hours = 0 : 23
    for minutes = 0 : 59
        for seconds = 0 : 59
            disp ([ num2str(hours) ":" num2str(minutes) ":" num2str(seconds)]) ;
        end
    end
end
```

Question 3

Write an Octave program that reads a sequence of numbers from the user.

- User may enter a negative value, or a positive value, or zero.
- When the user enters zero the loop is terminated.
- Otherwise, the negative and positive numbers are summed up separately.

When the loop terminates, the average of all negative numbers and the average of all positive numbers are displayed separately. For example:

```
Enter a number:   -5
Enter a number:   -15
Enter a number:   3
Enter a number:   12
Enter a number:   8
Enter a number:   4
Enter a number:   -10
Enter a number:   0
Average of negative numbers: -10                    % -30/3 = -10
Average of positive numbers: 6.75                   %  27/4 = 6.75
```

Solution:

We have solved half of this problem previously as follows:

```
total = 0;
while (true)
       n = input('Enter number: ');
       if (n < 0)
           disp('Finished!');
           break;
       end
       total = total + n;
end
disp(['Total = ' num2str(total)]);
```

The above solution to takes the sum of numbers entered and stops when a negative number is entered. For the exam question, we need to sum up negative and positive numbers separately. Also, we need to count how many positive and negative numbers are entered to be able to calculate the average.

The solution is as follows:

```
totalPositive = 0;
countPositive = 0;

totalNegative = 0;
countNegative = 0;

while (true)
       n = input('Enter number: ');
       if (n == 0)
           disp('Finished!');
           break;
       end

       if (n > 0)
           totalPositive = totalPositive + n;
           countPositive = countPositive + 1;
       end

       if (n < 0)
           totalNegative = totalNegative + n;
           countNegative = countNegative + 1;
       end
end
disp (['Average of Negative Numbers ' num2str(totalNegative/countNegative)]);
disp (['Average of Positive Numbers ' num2str(totalPositive/countPositive)]);
```

Question 4

Write a program to print a multiplication table.

- At the start, it should ask the user which table to print.
- After asking which table the user wants, ask her how high the table should go.

The output should look something like this, **if** the user enters 7 and 12: ***Note that this is only an example. Your code should also work with other numbers such as 12, 18 or 120, 9.***

```
Which multiplication table would you like? 7
How high do you want to go? 12
Here's your table:
7 x 1 = 7
7 x 2 = 14
7 x 3 = 21
7 x 4 = 28
7 x 5 = 35
7 x 6 = 42
7 x 7 = 49
7 x 8 = 56
7 x 9 = 63
7 x 10 = 70
7 x 11 = 77
7 x 12 = 84
```

Solution:

The solution is elementary, requiring only a single for loop:

```
n = input(' Which multiplication table would you like? ');
m = input(' How high do you want to go? ');
disp ('Here is your table: ') ;

for i=1:m
    disp([num2str(n) ' x '  num2str(i) ' = ' num2str(n*i)]);
end
```

Question 5

Read from keyboard the current population of a city, and a number representing the rate at which the population is increasing per year. Calculate the number of years it will take for **the population** of the city **to double**, assuming the present rate of growth remains constant.

Example:

Assume a population of 1,500,000, and 2 percent population increase rate are given for a city. In that case, the population of the city will be 1,530,000 next year. Then your code should calculate **after how many years** the population will be 3,000,000 in this city.

Solution:

The code requires a single while loop:

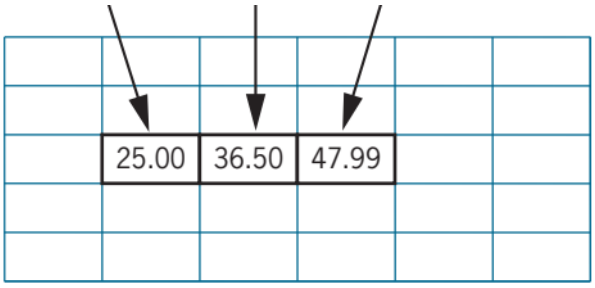
```
p = input(' What is the current population? ');
r = input(' What is the annual population increase rate? ');
y = 0 ;
k = p ;

while (k < 2*p)
   y = y + 1;
   k = k + k * r ;
   disp([' After '  num2str(y) ' years population will be: ' num2str(k)]);
end
```

❺ *ARRAYS*

- Until now, we have considered storing only one value as a variable. However, there can be situations when a set of elements require similar processing. Then it would be wise to store them as an ordered set instead of creating separate variables for each data point.
- Octave defines an object named **Array** that can store a ***sequential set of elements***.
- When you declare an array, you declare a structure that contains multiple data items; each data item is one **element** of the array.
- Each element has the same data type, and each element occupies an area in memory next to, or contiguous to, the others.
- You can indicate the number of elements an array will hold (the **size of the array)** when you declare the array along with your other variables and constants.
- Each array element is differentiated from the others with a unique **index**, which is a number that indicates the position of a particular item within an array. All array elements have the same group name, but each individual element also has a unique index.

```
prices[1] = 25.00
prices[2] = 36.50
prices[3] = 47.99
```



- Arrays can be defined by simply enclosing elements in square brackets and separating them by comma operators or whitespace. For example:

```
>> a1 = [1,2,3]
   a1 =
      1  2  3
>> a2 = [1 2 3]
   a2 =
      1  2  3
>> a = [1 0 0 3 2 1 3 0 1 5 3 2 1]    % defined an array
   a =
     1 0 0 3 2 1 3 0 1 5 3 2 1
>> a(1)                               % first element
   ans = 1
>> a(10)                              % tenth element
   ans = 5
```

- The semicolon ; operator sends the element in the next row instead of the next column. This way, a 2D or 3D array can be created.

```
>> a3 = [1;2;3]
   a3 =
       1
       2
       3
```

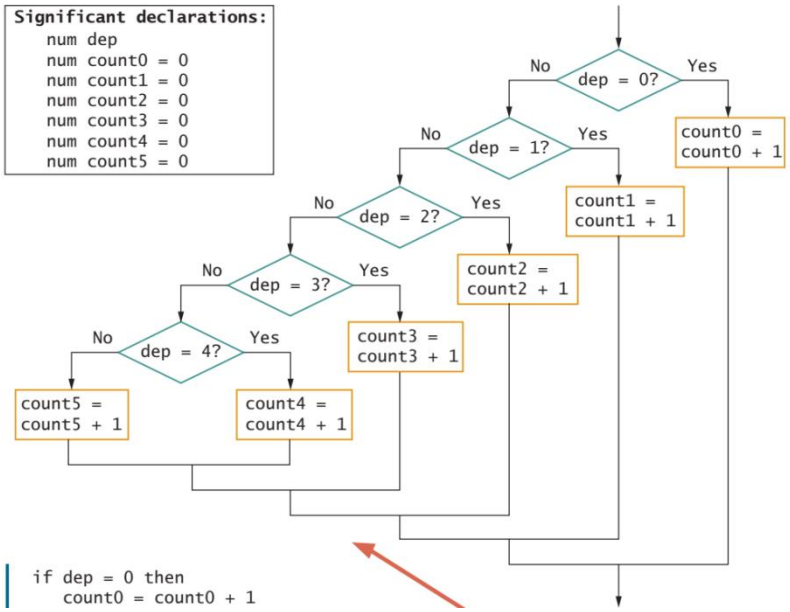
- The comma/whitespace operator will separate elements in **the same row** and ; will define the element in **the next column**.

```
>> a = [1 2 3;4 5 6]                % defining a two dimensional array
   a =
     1  2  3
     4  5  6
>> a(1,2)                           % element in row 1 and column 2
   ans = 2
>> a(2,1)                           % element in row 2 and column 1
   ans = 4
>> a(2,2)                           % element in row 2 and column 2
   ans = 5
>> a(2,4)                           % element in row 2 and column 4
   error: a(_,4): but a has size 2x3
```

- The elements of an array can have any data type.

For Loops for 1D Arrays (Vectors)

- Consider an application requested by a company's human resources department to produce statistics on employees' **claimed dependents**.
- The department wants a report that lists the number of employees who have claimed 0, 1, 2, 3, 4, or 5 dependents. (Assume that you know that no employees have more than five dependents.)
- Without using an array, you could write this application that produces counts for the six categories of dependents (0 through 5) by using a series of decisions. Figure below shows the flowchart for the decisionmaking part of such an application.



- In the flowchart, the variable dep is compared to 0. If it is 0, 1 is added to count0. If it is not 0, then dep is compared to 1. Based on the result, 1 is added to count1 or dep is compared to 2, and so on.
- Each time the application executes this decision-making process, 1 ultimately is added to one of the six variables that acts as a counter.
- The dependent-counting logic works, but even with only six categories of dependents, the decision-making process is unwieldy. What if the number of dependents might be any value from 0 to 10, or 0 to 20? The basic logic of the program would remain the same; however, you would need to declare many additional variables to hold the counts, and you would need many additional decisions.
- Although this logic works, its length and complexity are unnecessary once you understand how to use an array.
- Using an array provides an alternate approach to this programming problem and greatly reduces the number of statements you need. When you declare an array, you provide a group name for a number of associated variables in memory. For example, the six dependent count accumulators can be redefined as a single array named **counts**. The

individual elements become counts[1], counts[2], counts[3], counts[4], counts[5], and counts[6].

- The true benefit of using an array lies in your ability to use a variable as an index to the array, instead of using a literal constant such as 0 or 5.
- Within each decision, the value compared to **dep** and the constant that is the index in the resulting Yes process are always related. That is, when **dep** is 0, the index used to add 1 to the counts array is 1; when **dep** is 1, the index used for the counts array is 2, and so on. Therefore, you can just use dep as an index to the array. You can rewrite the decisionmaking process as simply:

```
counts(dep+1) = counts(dep+1) + 1 ;
```

- Note that, for loop can also count **backwards**. For example:

```
a = [ 1,2,3,4,5] ;
for i=5:-1:1
    disp([ 'a(' num2str(i) ') = ' num2str(a(i)) ]) ;
end
```

The output will be;

```
a(5) = 5
a(4) = 4
a(3) = 3
a(2) = 2
a(1) = 1
```

Operations on Arrays and Vectors

- Operating on arrays involves two aspects:
  - Operating on ***two or more arrays***
  - ***Element-wise*** operations
- All arithmetic operators, such as +, -, *, /, %, ^, etc., can be used in both cases.
- When you need to do element-wise operations, then a dot . is placed before the operator. The element-wise operators therefore become .+, .-, .*, ./, .%, and .^.
- For example, we write 2.+a to add 2 to each element individually. This can be done regardless of size and is implemented uniformly on all the elements of the matrix or vector. This will become clearer in the following example.

```
>> a = [1,2;3,4]
   a =
     1 2
     3 4
>> b = [5,6;7,8]
   b =
     5 6
     7 8
>> a + b
   ans =
       6  8
       10 12
>> 2 .+  a
   ans =
       3 4
       5 6
>> -10 .+  b
   ans =
       -5 -4
       -3 -2
```

- Other mathematical functions— like sin(), cos(), asin(), etc.—are already **vectorized,** which means they perform the operation on each element of the given array. When a and b are matrices to be added/subtracted, their elements are added/subtracted *with elements in the same position*. For this reason, the size of the two matrices added or subtracted must be the same.
- There are many operations that can be performed on a matrix or any array. For example:

```
>> a = rand(2,3)
   a =
     0.6787  0.7431  0.6555
     0.7577  0.3922  0.1712
>> b = rand(2,3)
   b =
     0.7060  0.2769  0.0971
     0.0318  0.0462  0.8235
>> c = (a < b)
   c =
     1  0  0
     0  0  1
```

- Here, two random matrices are created using random number generator function **rand(x,y)**.
- Then, these two matrices are compared to each other element-wise.
- The matrix c has elements, either 1 or 0, which are assigned by determining whether the corresponding elements of a are smaller than b.
- The matrix c contains logical data types, i.e., 1 and 0 represent the Boolean quantities True and False.
- A square matrix has an equal number of rows in each dimension. The built-in function issquare() can be used to check if the given matrix (represented by an array) is a square matrix and whether an appropriate function should be used. The result is 1 if the matrix is a square matrix and 0 otherwise. Its usage is illustrated in the following code:

```
>> a = [1,2;2,3];
>> issquare(a)
   ans = 1
>> b = [1,2;2,3;3,4];
>> issquare(b)
   ans = 0
```

- Read about **rank()**, **trace()**, **norm()**, **eye()** functions in the documentation.
- You can automatically generate an array by defining a rule using the **colon : operator** or by using the **linspace()** arguments. These methods are widely used, as they are convenient ways to generate large matrices. It is important to remember that you can suppress the output being printed on the terminal by ending the command with the **semicolon ;** operator, since it can be quite annoying to see a large set of numbers on the terminal.
- You can generate a series of numbers and store them as arrays by using the command **start:step:stop**, where the numbers representing start, step, and stop are real numbers. The result is an **array**.
- Defining the brackets (**[]**) is optional. If the step is not defined, then it is taken as 1.

```
>> x = 1:2:10                    % without brackets, start=1, step=2, stop=10
   x =
     1 3 5 7 9
>> x = [1:1:10]                  % with brackets, start=1, step=1, stop =10
   x =
     1 2 3 4 5 6 7 8 9 10
>> y = 2.2:3.8                   % start=2.2, stop=3.8, step=1
   y =
     2.2000 3.2000
>> y = 2.2:0.2:3.8               % start=2.2, step=0.2, stop=3.8
   y =
     2.2000 2.4000 2.6000 2.8000 3.0000 3.2000 3.4000 3.6000 3.8000
```

- The **linspace**(**start**,**stop**,**n**) command produces an array starting with the first number and stopping at the second one, with a total of n numbers. Hence, they are linearly spaced.

```
>> a = linspace(1,2,5)                    % start=1, stop=2, number of items=5
   a =
     1.0000 1.2500 1.5000 1.7500 2.0000

>> a = linspace(1,2,10)                   % start=1, stop=2, number of items=10
   a =
     1.0000 1.1111 1.2222 1.3333 1.4444 1.5556 1.6667 1.7778 1.8889 2.0000
```

- A matrix that contains all 1s or all 0s is a **ones matrix** and **zeros matrix**, respectively. These are generally used for initialization of matrices of desired dimensions. The initialized matrix is then used for manipulations. These matrices can be created as follows:

```
>> ones(3,3)
   ans =
       1 1 1
       1 1 1
       1 1 1

>> zeros(3,3)
   ans =
       0 0 0
       0 0 0
       0 0 0
```

Matrix Multiplication

- Matrix multiplication and division are not straightforward tasks.
- An m×n matrix can only be multiplied by a n×t matrix, which results in m×t matrix. This is performed by multiplying elements of rows with elements of columns to get new elements.
- Note that, for matrices a and c,
  - **a*c** performs ***matrix multiplication***,
  - **a.*c** performs ***elementwise multiplication***.
- The requirements are as follows:
  - For matrix multiplication, the inner dimensions must match.
  - For element-wise multiplication, all dimensions must match.
- When the inverse of a matrix is multiplied by itself, you get an identity matrix, i.e., a matrix with 1 as its elements in the diagonal direction and 0 everywhere else. This can be used to determine if the functions **inv()** and **pinv()** are working fine.

```
>> a = [2,-2;4,2]
   a =
     2 -2
     4  2

>> inv(a)
   ans =
       0.16667   0.16667
      -0.33333   0.16667

>> a*inv(a)
   ans =
       1.00000  0.00000
       0.00000  1.00000
```

Double- and Triple-Nested Loops for 2D and 3D Arrays

- We will implement matrix addition without the support of matrix addition capabilities of Octave. Matrix addition is formulated as follows for any two m×n matrices A and B:

$$c_{ij} = a_{ij} + b_{ij} \qquad 1 \le i \le m,\ 1 \le j \le n \quad 1$$

- Therefore, the implementation requires two nested for-loops:

```
for i = 1:m
  for j = 1:n
      C(i,j) = A(i,j) + B(i,j) ;
  end
end
```

- There are many ways for flowcharting these two nested for-loops:

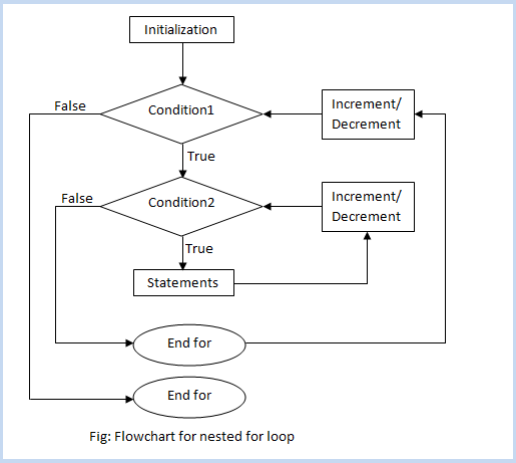


Fig: Flowchart for nested for loop

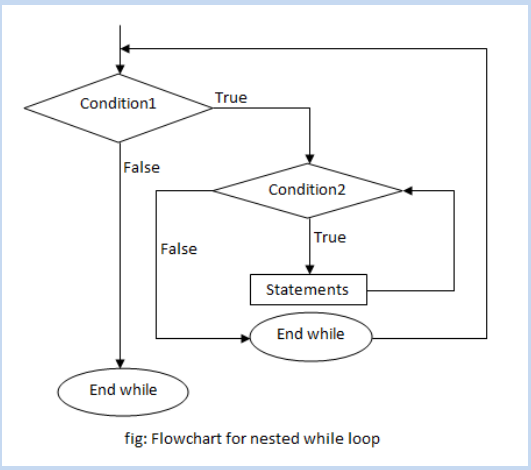
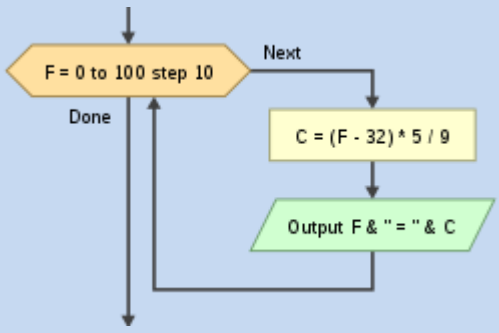
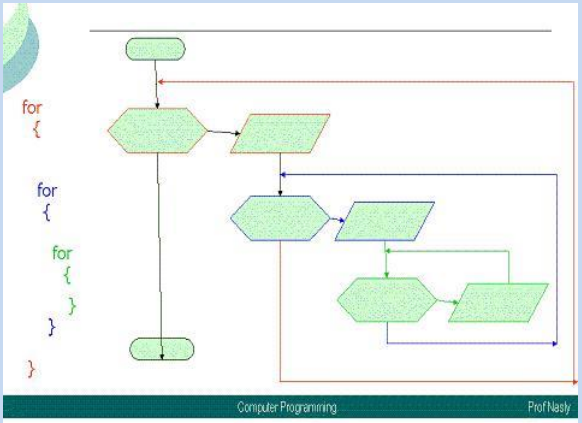


fig: Flowchart for nested while loop

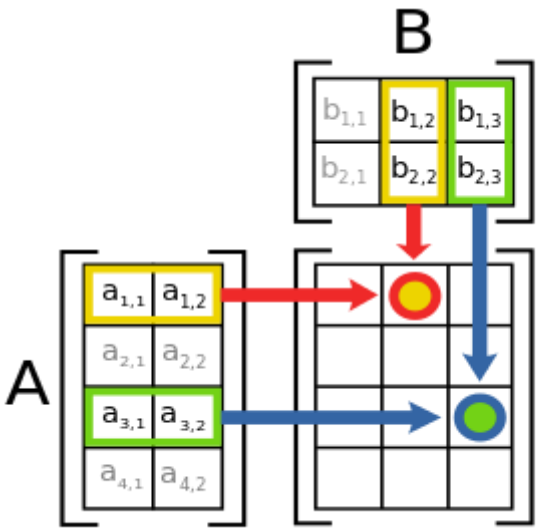
- For nested for-loops, we will prefer to use the following notation. Note that this version is more compact and has a one-to-one correspondance with loop statements in most programming languages.





- Matrix multiplication is formulated as follows for two matrices A and B:

$$c_{ij} = \sum_{k=1}^{m} a_{ik}\, b_{kj} \qquad 1 \le i \le n,\ 1 \le j \le p \ \text{ for } A^{n\times m}, B^{m\times p}, C^{n\times p} \quad 1$$



- Therefore, the implementation requires three nested for-loops:

```
for i = 1:n
  for j = 1:p
      sum = 0 ;
      for k = 1:m
          sum = sum + A(i,k) + B(k,j) ;
      end
      C(i,j) = sum ;
  end
end
```

- Strictly speaking, the variable **sum** is not really necessary. However we use it to the proper place for instructions that must be executed between the iterations of nested loops.
- Flowchart the matrix multiplication algorithm.

Some Self-Study Excersises

Question 1

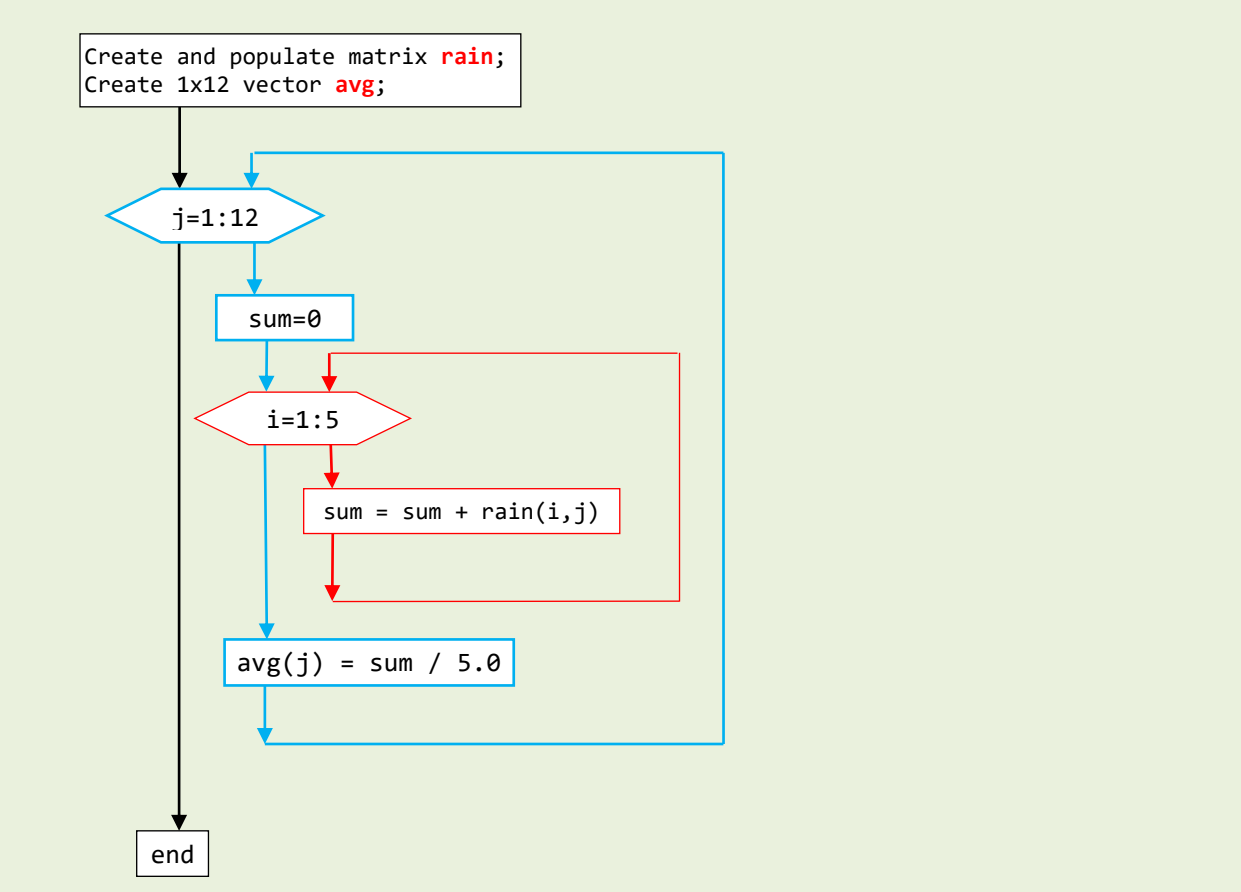
For Gaziantep, monthly rainfall data for 5 years (in Kg/m^2) is provided in a table such as below (Not all numbers are shown):

	Jan	Feb	March	April	May	June	July	Aug	Sept	Oct	Nov	Dec
1985	10	15	20	12	10	7	5	2	7	10	11	14
1986	8											
1987	12											
1988	9											
1989	11											

Assume that these values are given in a 5×12 matrix called **rain**. From this matrix, calculate the ***average rainfall for each month*** and write them into an 1×12 array called **avg**. (For the above table, the first element of **avg** should have the value of 10 (**avg(1)** = **10** since 10+8+12+9+11/5 = 50/5 = 10).

Solution:

The flowchart:



The Octave code is as follows. Be careful about the order of the indices of the matrix rain.

```
rain = randi(50,5,12);          % Create a rainfall matrix with random entries
avg  = zeros(1,12);

for j = 1:12
  sum = 0 ;
  for i = 1:5
    sum = sum + rain(i,j) ;
  end
  avg (j) = sum / 5.0;
end
```

■ Question 2

Find the greatest number in any array of numbers.

■ Question 3

Multiply negative numbers in an array of numbers with 10.

■ Question 4

Copy an ordered array of numbers in reverse order into another array.

■ Question 5

Sort the numbers in an array in **ascending** (increasing) order.

■ Question 6

Sort the numbers in an array in **descending** (decreasing) order.

■ Question 7

Select the positive numbers from an array of numbers and place them into a new array in descending order.

■ Question 8

Insert a number into an array of numbers in ascending order, so that the order is not disturbed.