Chapter 13 Control Flow

13.1 For Loops

'For' Loops are one of several ways to control the order of execution of MATLAB commands in an M-file. It (or something equivalent) is a basic construct of all high level programming languages. The 'For' Loop is used when a group of MATLAB commands are to be executed a predetermined number of times. The 'For' Loop commands are enclosed between the initial line which defines the loop variable and an array of values for it to assume and the 'end' statement which signals the end of the loop. The general form is

```
for x = array
  (commands)
end
```

Example 13.1.1

```
A=[1 2; 3 4] % Create square matrix A
for i=1:5  % Set array values for loop variable i as 1,2,...,5
B=A^i  % Compute matrix B from constant matrix A and loop variable i
end % for i
C=diag(diag(B)) % Create diagonal matrix C using diagonal from matrix B

A =  1     2
     3     4

B =  1     2
     3     4

B =  7    10
    15    22

B =  37    54
     81   118

B = 199   290
    435   634

B = 1069   1558
    2337   3406

C = 1069      0
       0   3406
```

Example 13.1.2

```
for x=linspace(0,pi/2,4)  % Set array values for loop variable x
x
y=sin(x)
z=cos(x)
end  % for x
```

```
x =  0
y =  0
z =  1

x = 0.5236
y = 0.5000
z = 0.8660

x = 1.0472
y = 0.8660
z = 0.5000

x = 1.5708
y = 1
z = 6.1232e-017
```

The loop variable in a '`For`' Loop is not required to be present in any of the loop commands. If its not, all the loop commands are simply executed for each iteration of the loop.

Example 13.1.3

```
x=0; % Initialize x
for k = 1:5
x=x+rand % Accumulate sum
end % for k
```

```
x =  0.9862
x =  1.8716
x =  2.2764
x =  2.9035
x =  3.2890
```

The loop variable in a '`For`' Loop should not appear by itself on the left side of an assignment statement, i.e. it should not be reassigned because the loop will execute a fixed number of times according to the array of values specified in the first line.

Example 13.1.4

```
y(4)=0 % Preallocate memory for array y
for n=1:4
n  % Display value of loop variable
y(n)=n*rand
n=4  % This will not terminate the loop
end
```

```
y =     0     0     0     0

n =  1
y =  0.4712    0         0         0
n =  4

n =  2
y =  0.4712    0.2986    0         0
n =  4

n =  3
y =  0.4712    0.2986    0.4076    0
n =  4

n =  4
y =  0.4712    0.2986    0.4076    2.1300
n =  4
```

The values assumed by the loop variable are not confined to a specific ordering, e.g. equally spaced numbers.

Example 13.1.5

```
index_values=[1 9 16 25];
for i=index_values
i,y=i^0.5
end
```

```
i =   1
y =   1

i =   9
y =   3

i =   16
y =    4

i =   25
y =    5
```

The loop variable array values can be based on results from commands issued prior to the loop. For example, it may consist of indices obtained by searching an array for specific conditions.

Example 13.1.6

```
clear all
X=10*rand(4)  % Create 4´4 array X of 16 random numbers from 0 to 10
indices=[find(X>7)]'  % Find indices of X corresponding to elements
                      % greater than 7 and convert to row vector
j=1;
for i=indices % Set loop index values
i,y(j)=X(i)  % Save elements of X which are greater than 7 in array y
j=j+1; % Increment index j
end
```

```
X =
    0.6932    7.3393    0.1289    5.6948
    8.5293    5.3652    8.8921    1.5926
    1.8033    2.7603    8.6602    5.9436
    0.3242    3.6846    2.5425    3.3110

indices =   2     5    10    11

i =   2
y =   8.5293

i =   5
y =   8.5293    7.3393

i =  10
y =   8.5293    7.3393    8.8921

i =  11
y =   8.5293    7.3393    8.8921    8.6602
```

In most cases, the array of values specified in the first line of the '`For`' Loop is a row vector i.e. a `1×n` array. However, it can be an `m ×n` array in which case the loop variable is initially assigned the first column of the array for the first iteration of the loop, the second column of the array for the second iteration, etc. As an illustration, Example 13.1.6 can be modified to leave the array of values '`indices`' as a single column vector instead of transposing it into a row vector.

Example 13.1.7

```
clear all
X=10*rand(4)  % Create 4´4 array X of 16 random numbers from 0 to 10
indices=find(X>7)  % Find indices of X corresponding to elements
                   % greater than 7 and store as column vector
for i=indices  % Set loop variable i equal to column vector 'indices'
i,y=X(i)  % Address elements of X according to contents of column
          % vector i and store those elements in vector y
end

X =
    6.5861    7.9183    6.3899    4.4159
    8.6363    1.5259    6.6900    4.8306
    5.6762    8.3303    7.7209    6.0811
    9.8048    1.9186    3.7982    1.7600

indices =
     2
     4
     5
     7
    11
```

```
i =
     2
     4
     5
     7
    11

y =
    8.6363
    9.8048
    7.9183
    8.3303
    7.7209
```

'For' Loops can be nested when appropriate.

Example 13.1.8

```
A(4,4)=0 % Preallocate memory for array A
for i=1:4
   for j = 1:4
        A(i,j)=(-1)^(i+j);  % Create matrix 'A' of 1's and (-1)'s
   end %  for j
end   % for i
A
B=(A+1)/2  % Create matrix B of 0's and 1's
find(B) % Find indices of nonzero elements of array B

A =
     1    -1     1    -1
    -1     1    -1     1
     1    -1     1    -1
    -1     1    -1     1

B =
     1     0     1     0
     0     1     0     1
     1     0     1     0
     0     1     0     1

ans =
     1
     3
     6
     8
     9
    11
    14
    16
```

The power of MATLAB is its ability to treat arrays as a data type which can be manipulated without the need for one or more nested loops. For example, if 'A' and 'B' are equally sized arrays, then the matrix sum is simply 'A+B'. Other programming

languages necessitate one nested loop for each dimension. An array X can be treated as a single entity when evaluated in a function 'f(X)' without the use of nested loops.

Example 13.1.9

```
A=rand(4) % Create 4´4 array of random numbers from 0 to 1
B=rand(4) % Create 4´4 array of random numbers from 0 to 1
C=A+B  % Find sum of A and B and store in C
%%%%%  Find sum of A and B using nested loops  %%%%%
for i=1:4
   for j=1:4
      C(i,j)=A(i,j)+B(i,j);
   end  %  for j
end  % for i
C

A =
    0.1761    0.3762    0.6177    0.5995
    0.0679    0.9522    0.6492    0.8986
    0.3094    0.7193    0.7563    0.1719
    0.3348    0.7793    0.1478    0.8189

B =
    0.0693    0.7599    0.8459    0.7616
    0.9557    0.3087    0.7184    0.6695
    0.3173    0.7153    0.8704    0.9020
    0.0052    0.0809    0.8722    0.8215

C =
    0.2453    1.1361    1.4635    1.3610
    1.0236    1.2609    1.3676    1.5681
    0.6267    1.4346    1.6267    1.0740
    0.3400    0.8603    1.0200    1.6404

C =
    0.2453    1.1361    1.4635    1.3610
    1.0236    1.2609    1.3676    1.5681
    0.6267    1.4346    1.6267    1.0740
    0.3400    0.8603    1.0200    1.6404
```

Example 13.1.10

```
n=1000;
x=rand(1,n);  % Create row vector x of n random numbers
ave1=mean(x) % Find the average of values in vector x
%%%%% Find average value of n values in array x using a loop %%%%%
sum=0; % Initialize sum
for i=1:n
sum=sum+x(i);
end  % for i
ave2=sum/n % Compute the average of values in vector x

ave1 =  0.5016
ave2 =  0.5016
```

13.2 While Loops

'`While`' Loops as the name suggests are a series of commands which are repeatedly executed as long as some condition remains True. The syntax for a '`While`' Loop is

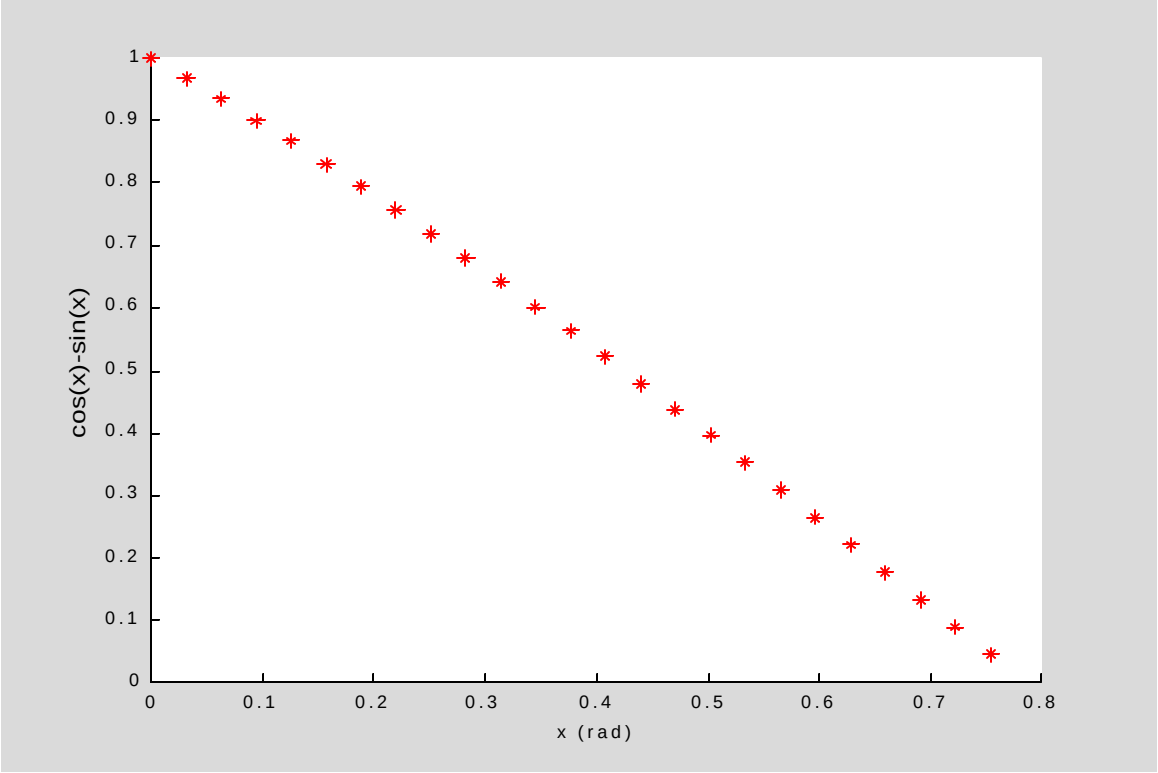
```
while expression
  (commands)
end
```

Assuming the initial evaluation of the expression is True entering the '`While`' Loop, execution of the commands inside the loop eventually causes the expression to become False. Otherwise, the loop would execute indefinitely. The expression can produce either a scalar or an array. In either case, the single element or all the array elements must be True for the loop commands to be executed. The primary difference between the '`While`' and '`For`' Loops is the uncertainity about the number of loop iterations in a '`While`' Loop in contrast to the '`For`' Loop which executes a fixed number of times (assuming no '`break`' command inside the '`For`' loop).

Example 13.2.1

```
x=0;
hold on
while cos(x)>sin(x)  % Loop executes as long as cos(x)>sin(x) is True
diff=cos(x)-sin(x);
plot(x,diff,'r*')  % Plot data point (x,diff) as a red * marker
xlabel('x (rad)')
ylabel('cos(x)-sin(x)')
x=x+pi/100;  % Increment x
end  % while cos(x)>sin(x)
x,cos(x),sin(x) % Display last values of x, cos(x) and sin(x)

x =  0.7854
ans =  0.7071
ans =  0.7071
```



The 'While' Loop may fail to execute even once if the initial evaluation of the test expression is not True. This can be avoided if necessary, i.e. there will be at least one pass through the 'While' Loop commands, by initializing certain variables to guarantee the initial evaluation of the test expression is True. In the following example, all elements of the array 'y' are set to 1 to force the loop to execute at least once.

Example 13.2.2

```
y=ones(1,5)  % Initialize row vector y to all 1's
sum=zeros(1,5);  % Initialize row vector 'sum' to all 0's
while y>0  %  Continue loop execution while all elements of array y are
           % greater than zero
x=randn(1,5)  % Generate row vector x of N(0,1) values
y=x+2 % Calculate array y values used in the 'while' expression
sum=sum+x;
end  % while
sum

y =    1         1         1         1         1

x =   -0.8468   -0.2463    0.6630   -0.8542   -1.2013
y =    1.1532    1.7537    2.6630    1.1458    0.7987

x =   -0.1199   -0.0653    0.4853   -0.5955   -0.1497
y =    1.8801    1.9347    2.4853    1.4045    1.8503
```

```
x =   -0.4348   -0.0793    1.5352   -0.6065   -1.3474
y =    1.5652    1.9207    3.5352    1.3935    0.6526

x =    0.4694   -0.9036    0.0359   -0.6275    0.5354
y =    2.4694    1.0964    2.0359    1.3725    2.5354

x =    0.5529   -0.2037   -2.0543    0.1326    1.5929
y =    2.5529    1.7963   -0.0543    2.1326    3.5929

sum = -0.3791   -1.4982    0.6650   -2.5511   -0.5700
```

As with 'For' Loops, 'While' Loops can be nested or a 'For' Loop can be nested inside a 'While' Loop.

Example 13.2.3

```
clear all
y=1 % Force outer while loop to execute at least once
num=0;
while y>0.25
    x=rand;
    while x+1>y
         num=num+1; % Number of times inner while loop executes
         w(num)=x*y;
         y=y+0.5;
    end  % while x+1>y
    y=rand
end  % while y>0.25
num,w
```

```
y =     1

y =    0.4303
y =    0.9137
y =    0.3879
y =    0.0667

num =  8

w = 0.4948   0.1487   0.3214   0.1463   0.3752   0.8588  1.3424   1.8260
```

Both 'For' Loops and 'While' Loops can be terminated with a 'break' command situated inside the loop. In the next example, a 'break' command terminates execution of a nested 'For' Loop and the program jumps to the next command outside the 'For' Loop.

Example 13.2.4

```
clear
winnings=0;
roster=char('Smith',    'Jones',    'Pierce',    'Edwards',    'Gonzalez',
'Bishop', 'Ball', 'Freeman', 'Lane', 'Price');
player=roster(1,:);
k=0;
while  player(1)~='P'

    k=k+1;
    index=ceil(10*rand)
    winnings(k)=0;
    for i=1:index
       winnings(k)=winnings(k)+10;
            if winnings(k)==50
                break
            end  % if winnings==50
    end  % for i
    player=roster(index,:)

end  % while player(1)~='P'
winnings

index =     2
player =  Jones

index =     8
player =  Freeman

index =     8
player =  Freeman

index =     3
player =  Pierce

winnings =    20    50    50    30
```

13.3 If-Else-End Constructions

There are times when a sequence of commands are to be executed only when a relational condition evaluates as True. When its not, the commands are skipped. The basic construct is

```
If expression
  (commands)
end
```

The expression may be a simple logical expression or may consist of several logical subexpressions.

Example 13.3.1

```
format bank
n=5;
mu=65; sigma=15;
student_ave=mu+sigma*randn(1,n);   % Generate n N(mu,sigma) averages

for k=1:n
   student_grade(k,1:4)='Fail'; % Initialize kth student grade to Fail

      if student_ave(k)>65  % Check if kth student ave greater than 65
        student_grade(k,1:4)='Pass'; % Change kth student grade to Pass
                                     % if ave greater than 65
      end  % if student_ave(k)>65

   student_ave(k),student_grade(k,1:4)
end  % for k
format short

ans =  58.51
ans =  Fail

ans =  40.02
ans =  Fail

ans =  66.88
ans =  Pass

ans =  69.32
ans =  Pass

ans =  47.80
ans =  Fail
```

Example 13.3.2

```
age=ceil(35+30*rand) % Generate random age
salary=ceil(50000+10000*randn) % Generate random salary
years_employed=ceil(10+10*rand) % Generate random employment years
bonus=2500;
   if (age>45 & salary>55000)|years_employed>15
     bonus=5000;
   end  % if  (age>45 & salary>55000)|years_employed>15
bonus
```

```
age =  61
salary =   37657
years_employed =  17
bonus =  5000
```

In the previous example, the expression '`age>45&salary>55000`' is evaluated first and if True, the second expression would not be evaluated because of the logical operator '`|`'. Conversely, if the first compound logical subexpression is False, then the second expression '`years_employed>15`' would be evaluated since the complete '`if`' expression '`(age>45&salary>55000)|(years_employed>15)`' could still be True.

The '`If`' construct is somewhat different when there are two alternatives, namely

```
if expression
  (commands evaluated if True)
else
  (commands evaluated if False)
end
```

In this case, the first set of commands are executed if the '`expression`' is True, otherwise the sequence of commands after the '`else`' are evaluated.

Example 13.3.3

```
n=1000;
mu=0;
sigma=5;
x=mu+sigma*randn(1,n); % Generate n N(mu,sigma) random deviates
num_expected=0; % Initialize number of expected outcomes
num_outliers=0; % Initialize number of outliers
   for i=1:n
      if abs(x(i))<mu+3*sigma  % Check if x(i) is within 3 sigma limits
        num_expected=num_expected+1;  % Increment num_expected if so
      else
        num_outliers=num_outliers+1;  % Increment num_outliers if not
      end  % if abs(x(i))<mu+3*sigma
   end  % for i
n,num_expected,num_outliers
```

```
n =  1000
num_expected = 998
num_outliers =   2
```

When there are more than two alternatives, the correct syntax is

```
if expression1
  (commands evaluated if expression1 is True)
elseif expression2
  (commands evaluated if expression2 is True)
elseif expression3
  (commands evaluated if expression3 is True)
elseif ...
.
.
else
  (commands evaluated if no other expression is True)
end
```

As soon as a True expression is encountered, the appropriate commands are executed and control passes to the first command after the '`end`' statement. Note, the final '`else`' is optional.

Example 13.3.4

```
n=1000; mu=0; sigma=5;
x=mu+sigma*randn(1,n); % Generate n N(mu,sigma) random deviates
num_1_sigma=0; % Initialize number of outcomes within 1 sigma limits
num_2_sigma=0; % Initialize number of outcomes within 2 sigma limits
num_3_sigma=0; % Initialize number of outcomes within 3 sigma limits
num_outliers=0; % Initialize number of outcomes outside 3 sigma limits
   for i=1:n
      if abs(x(i))<mu+1*sigma  % Check if x(i) is within 1 sigma limits
        num_1_sigma=num_1_sigma+1; % If so, Increment num_1_sigma
      elseif abs(x(i))<mu+2*sigma  %Check if x(i) within 2 sigma limits
        num_2_sigma=num_2_sigma+1; % If so, Increment num_2_sigma
      elseif abs(x(i))<mu+3*sigma  %Check if x(i) within 3 sigma limits
        num_3_sigma=num_3_sigma+1; % If so, Increment num_3_sigma
      else  % x(i) is an outlier
        num_outliers=num_outliers+1;  % Increment num_outliers
      end  % if abs(x(i))<mu+1*sigma
   end  % for i
n,num_1_sigma,num_2_sigma,num_3_sigma,num_outliers
```

```
n = 1000
num_1_sigma = 689
num_2_sigma = 257
num_3_sigma =  53
num_outliers =  1
```

13.4 Switch_Case Constructions

When the choice of which sequence of commands to execute depends on an equality test involving a common expression, a '`Switch-Case`' construct is sometimes easier to formulate. The syntax is

```
switch  expression
   case  test_expression1
     (commands1)
   case  test_expression2
     (commands2)
   .
   .
   case  test_expressionN
     (commandsN)
   otherwise
     (commands)
end
```

When '`expression`' on the header '`switch`' line is a scalar, it is compared with '`test_expression1`', i.e. Is '`expression==test_expression1`' True? If it is, the sequence of commands in '`commands1`' are executed. If its not, the process is continued until a '`case`' is encountered where it is True and the corresponding commands are executed. If there is no '`case`' where the '`expression'` and '`test_expression`' are equal, the commands after the optional `otherwise`' are executed. Note, without the optional '`otherwise`' present, its possible for none of the alternative sequence of commands to be executed.

Example 13.4.1

```
clear
n=4;  %   Car loan duration in years
m=12*n;  % Car loan duration in months
P=30000; % Loan amount

switch n

  case 3
      r=4.5;  % Annual interest rate for 3 year car loan
      i=r/1200;  % Monthly interest rate (as a decimal value)
      fact=(1+i)^m;
      A=P*(i*fact/(fact-1));  % Monthly payment for a 3 year car loan
```

```
  case 4
      r=5.5;  % Annual interest rate for 4 year car loan
      i=r/1200;  % Monthly interest rate (as a decimal value)
      fact=(1+i)^m;
      A=P*(i*fact/(fact-1));  % Monthly payment for a 4 year car loan

  case 5
      r=6.5;  % Annual interest rate for 5 year car loan
      i=r/1200;  % Monthly interest rate (as a decimal value)
      fact=(1+i)^m;
      A=P*(i*fact/(fact-1));  % Monthly payment for a 5 year car loan

end  % switch n

 if n==3|n==4|n==5  % Check if n equals 3, 4, or 5
     P,n,r,A
  end  % if n==3|n==4|n==5
```

```
P =   30000
n =   4
r =   5.5000
A =  697.6943
```

In the previous example, an '`otherwise`' should have been included to handle the case where `n` was other than 3,4, or 5. The command following the '`otherwise`' could be a string display so indicating.

If the '`expression`' in the '`Switch-Case`' construct is a character string, the equality tests use the '`strcmp`' command, e.g. the first equality test would be Is `strcmp(expression,test_expression1)` True? If the first comparison is not True, the remaining cases are checked until a True result occurs or the optional '`otherwise`' commands are executed.

Example 13.4.2

```
clear
league_payroll=0;
sport='Frisbees';
switch sport

  case  'Major League Baseball'
      ave_salary=1500000;
      num_teams=30;
      num_players=25;
      league_payroll=ave_salary*num_teams*num_players;

  case  'NFL Football'
      ave_salary=9500000;
      num_teams=20;
      num_players=45;
      league_payroll=ave_salary*num_teams*num_players;
```

```
  case  'NBA Basketball'
      ave_salary=2250000;
      num_teams=24;
      num_players=12;
      league_payroll=ave_salary*num_teams*num_players;

  case  'NHL Hockey'
      ave_salary=750000;
      num_teams=18;
      num_players=20;
      league_payroll=ave_salary*num_teams*num_players;

  otherwise
      disp('No data available for this sport')

end  % switch sport

if league_payroll>0
    sport,league_payroll
end  % if league_payroll>0
```

```
No data available for this sport
```