Chapter four
Programming in MATLAB

CONDITIONAL STATEMENTS(`if`):

A conditional statement is a command that allows MATLAB to make a decision of whether to execute a group of commands that follow the conditional statement, or to skip these commands. In a conditional statement a conditional expression is stated. If the expression is true, a group of commands that follow the statement are executed. If the expression is false, the computer skips the group. The basic form of a conditional statement is

`if` conditional expression consisting of relational and/or logical operators.

Examples:

```
if a < b
if c >= 5
if a == b
if a ~= 0
if (d<h)&(x>7)
if (x~=13)|(y<0)
```

All the variables must have assigned values.

• Conditional statements can be a part of a program written in a script file or a user-defined function
• As shown below, for every if statement there is an end statement.
The if statement is commonly used in three structures, `if-end`, `if-else-end`, and `if-elseif-else-end`, which are described next.

The `if-end` Structure:

The `if-end` conditional statement is shown schematically in Figure below. The figure shows how the commands are typed in the program, and a flowchart that symbolically shows the flow, or the sequence, in which the commands are executed. As the program executes, it reaches the **`if`** statement. If the conditional expression in the if statement is true (1), the program continues to execute the commands that follow the if statement all the way down to the end statement. If the conditional expression is false (0), the program skips the group of commands between the **`if`** and the **`end`**, and continues with the commands that follow the end.

The words **`if`** and **`end`** appear on the screen in blue, and the commands between the if statement and the end statement are automatically indented (they don't have to be), which makes the program easier to read. An example where the **`if-end`** statement is used in a script file is shown in Sample example.

Example: A worker is paid according to his hourly wage up to 40 hours, and 50% more for overtime. Write a program in a script file that calculates the pay to a worker. The program asks the user to enter the number of hours and the hourly wage. The program then displays the pay.

Solution

The program in a script file is shown below. The program first calculates the pay by multiplying the number of hours by the hourly wage. Then an `if` statement checks whether the number of hours is greater than 40. If so, the next line is executed and the extra pay for the hours above 40 is added. If not, the program skips to the end.

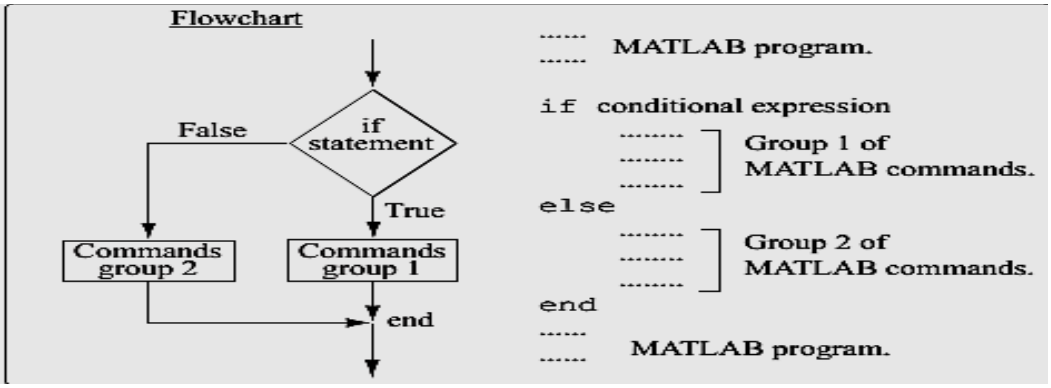
```
t=input('Please enter the number of hours worked  ');
h=input('Please enter the hourly wage in $  ');
Pay=t*h;
if t>40
    Pay=Pay+(t-40)*0.5*h;
end
fprintf('The worker''s pay is  $ %5.2f',Pay)
```

Application of the program (in the Command Window) for two cases is shown below (the file was saved as Workerpay):

```
>> Workerpay
Please enter the number of hours worked  35
Please enter the hourly wage in $  8
The worker's pay is  $ 280.00
>> Workerpay
Please enter the number of hours worked  50
Please enter the hourly wage in $  10
The worker's pay is  $ 550.00
```

The `if-else-end` Structure :

The if-else-end structure provides a means for choosing one group of commands, out of a possible two groups, for execution. The if-else-end structure is shown in Figure below. The figure shows how the commands are typed in the program, and a flowchart that illustrates the flow, or the sequence, in which the commands are executed.



The first line is an **`if`** statement with a conditional expression. If the conditional expression is true, the program executes group 1 of commands between the **`if`** and the **`else`** statements and then skips to the end. If the conditional expression is false, the program skips to the **`else`** and then executes group 2 of commands between the **`else`** and the **`end`**.

It should be pointed out here that several **`elseif`** statements and associated groups of commands can be added. In this way more conditions can be included. Also, the else statement is optional. This means that in the case of several **`elseif`** statements and no else statement, if any of the conditional statements is true the associated commands are executed; otherwise nothing is executed.

THE switch-case **STATEMENT:**

The switch-case statement is another method that can be used to direct the flow of a program. It provides a means for choosing one group of commands for execution out of several possible groups. The structure of the statement is shown in Figure below

• The first line is the switch command, which has the form:

```
switch  switch expression
```

he switch expression can be a scalar or a string. Usually it is a variable that has an assigned scalar or a string. It can also be, however, a mathematical expression that includes pre-assigned variables and can be evaluated

• Following the switch command are one or several case commands. Each has a value (can be a scalar or a string) next to it (value1, value2, etc.) and an associated group of commands below it.

• After the last case command there is an optional otherwise command followed by a group of commands.

• The last line must be an end statement.

```
.....
.....     MATLAB program.

switch switch expression
     case value1
     ........
     ........     ] Group 1 of commands.
     case value2
     ........
     ........     ] Group 2 of commands.
     case value3
     ........
     ........     ] Group 3 of commands.
     otherwise
     ........
     ........     ] Group 4 of commands.
end
.....
.....    MATLAB program.
```

The structure of a `switch-case` statement.

How does the switch-case statement work?

The value of the switch expression in the **`switch`** command is compared with the values that are next to each of the case statements. If a match is found, the group of commands that follow the **`case`** statement with the match are executed. (Only one group of commands-the one between the **`case`** that matches and either the **`case`, `otherwise`, or `end`** statement that is next—is executed).

• If there is more than one match, only the first matching case is executed.
• If no match is found and the **`otherwise`** statement (which is optional) is present, the group of commands between **`otherwise`** and **`end`** is executed.
• If no match is found and the otherwise statement is not present, none of the command groups is executed.
• A **`case`** statement can have more than one value. This is done by typing the values in the form: `{value1, value2, value3, ...}` . (This form, which is not covered in this book, is called a cell array.) The case is executed if at least one of the values matches the value of switch expression.

Example: Write a program in a script file that converts a quantity of energy (work) given in units of either joule, ft-lb, cal, or eV to the equivalent quantity in different units specified by the user. The program asks the user to enter the quantity of energy, it current units, and the desired new units. The output is the quantity of energy in the new units.

The conversion factors are: $1\,\text{J} = 0.738\,\text{ft-lb} = 0.239\,\text{cal} = 6.24 \times 10^{18}\,\text{eV}.$
Use the program to:
(*a*) Convert 150 J to ft-lb.
(*b*) Convert 2,800 cal to J.
(*c*) Convert 2.7 eV to cal.

Solution

The program includes two sets of `switch-case` statements and one `if-else-end` statement. The first switch-case statement is used to convert the input quantity from its initial units to units of joules. The second is used to convert the quantity from joules to the specified new units. The `if-else-end` statement is used to generate an error message if units are entered incorrectly.

```
Ein=input('Enter the value of the energy (work) to be converted: ');
EinUnits=input('Enter the current units (J, ft-lb, cal, or eV):  ','s');
EoutUnits=input('Enter the new units (J, ft-lb, cal, or eV):  ','s');
error=0;
switch EinUnits
case 'J'
    EJ=Ein;
case 'ft-lb'
    EJ=Ein/0.738;
case 'cal'
    EJ=Ein/0.239;
case 'eV'
    EJ=Ein/6.24e18;
otherwise
    error=1;
end
switch EoutUnits
case 'J'
    Eout=EJ;
case 'ft-lb'
    Eout=EJ*0.738;
case 'cal'
    Eout=EJ*0.239;
case 'eV'
    Eout=EJ*6.24e18;
```

Assign 0 to variable `error`.

First `switch` statement. Switch expression is a string with initial units.

Each of the four `case` statements has a value (string) that corresponds to one of the initial units, and a command that converts `Ein` to units of J. (Assign the value to EJ.)

Assign 1 to `error` if no match is found. Possible only if initial units were typed incorrectly.

Second `switch` statement. Switch expression is a string with new units.

Each of the four `case` statements has a value (string) that corresponds to one of the new units, and a command that converts EJ to the new units. (Assign the value to `Eout`.)

```
otherwise
    error=1;
end
if error
    disp('ERROR current or new units are typed incorrectly.')
else
    fprintf('E = %g %s',Eout,EoutUnits)
end
```

Assign 1 to `error` if no match is found. Possible only if new units were typed incorrectly.

`If-else-end` statement.

If `error` is true (nonzero), display an error message.

If `error` is false (zero), display converted energy.

As an example, the script file (saved as Energy Conversion) is used next in the Command Window to make the conversion in part (*b*) of the problem statement.

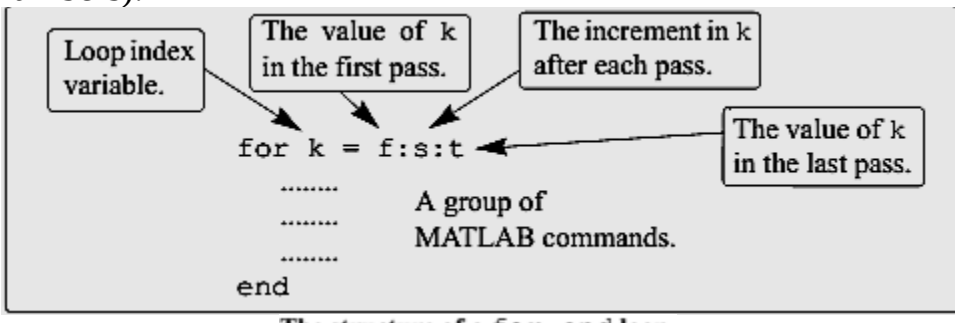
```
>> EnergyConversion
Enter the value of the energy (work) to be converted: 2800
Enter the current units (J, ft-lb, cal, or eV):  cal
Enter the new units (J, ft-lb, cal, or eV):  J
E = 11715.5 J
```

LOOPS:

for-end Loops:

In **`for-end`** loops the execution of a command, or a group of commands, is repeated a predetermined number of times. The form of a loop is shown in Figure below.

• The loop index variable can have any variable name (usually `i, j , k, m`, and `n` are used, however, `i` and `j` should not be used if MATLAB is used with complex numbers).



The structure of a `for-end` loop.

• In the first pass `k = f` and the computer executes the commands between the `for` and `end` commands. Then, the program goes back to the `for` command for the second pass. `k` obtains a new value equal to `k = f + s`, and the commands between the **`for`** and **`end`** commands are executed with the new value of `k`. The process repeats itself until the last pass, where `k = t`. Then the program does not

go back to the for, but continues with the commands that follow the end command. For example, if `k = 1:2:9`, there are five loops, and the corresponding values of `k` are 1, 3, 5, 7, and 9.

• The increment s can be negative (i.e.; k = 25:–5:10 produces four passes with k=25, 20, 15, 10).

• If the increment value s is omitted, the value is 1 (default) (i.e.; k = 3:7 produces five passes with k = 3, 4, 5, 6, 7).

• If f = t, the loop is executed once.

• If f > t and s > 0, or if f < t and s < 0, the loop is not executed.

• If the values of k, s, and t are such that `k` cannot be equal to `t`, then if `s` is positive, the last pass is the one where `k` has the largest value that is smaller than t (i.e., k = 8:10:50 produces five passes with k = 8, 18, 28, 38, 48). If `s` is negative, the last pass is the one where `k` has the smallest value that is larger than `t`.

• In the for command k can also be assigned a specific value (typed as a vector). Example: `for k = [7 9 -1 3 3 5].`

• The value of k should not be redefined within the loop.

• Each `for` command in a program *must* have an `end` command.

• The value of the loop index variable (k) is not displayed automatically. It is possible to display the value in each pass (which is sometimes useful for debugging) by typing `k` as one of the commands in the loop.

• When the loop ends, the loop index variable (k) has the value that was last assigned to it.

A simple example of a for-end loop (in a script file) is:

```
for k=1:3:10
     x = k^2
end
```

When this program is executed, the loop is executed four times. The value of `k` in the four passes is `k` = 1, 4, 7, and 10, which means that the values that are assigned to `x` in the passes are `x` = 1, 16, 49, and 100, respectively. Since a semicolon is not typed at the end of the second line, the value of `x` is displayed in the Command Window at each pass. When the script file is executed, the display in the Command Window is:

```
>> x =
     1

x =
    16

x =
    49

x =
   100
```

Sample Problem: Sum of a series

(*a*) Use a for-end loop in a script file to calculate the sum of the first *n* terms of the series: $\sum_{k=1}^{n} \frac{(-1)^k k}{2^k}$. Execute the script file for *n* = 4 and *n* = 20.

(*b*) The function sin(*x*) can be written as a Taylor series by:

$$\sin x = \sum_{k=0}^{\infty} \frac{(-1)^k x^{2k+1}}{(2k+1)!}$$

Write a user-defined function file that calculates sin(*x*) by using the Taylor series. For the function name and arguments use `y = Tsin(x,n)` . The input arguments are the angle x in degrees and n the number of terms in the series. Use the function to calculate sin(150) using three and seven terms.

Solution

(*a*) A script file that calculates the sum of the first `n` terms of the series is shown below. The summation is done with a loop. In each pass one term of the series is calculated

```
n=input('Enter the number of terms ' );
S=0;
for k=1:n
    S=S+(-1)^k*k/2^k;
end
fprintf('The sum of the series is: %f',S)
```

Setting the sum to zero.

for-end loop.

In each pass one element of the series is calculated and is added to the sum of the elements from the previous passes.

(in the first pass the first term, in the second pass the second term, and so on) and is added to the sum of the previous elements. The file is saved as Exp6_5a and then executed twice in the Command Window:

```
>> Exp6_5a
Enter the number of terms 4
The sum of the series is: -0.125000
>> Exp7_5a
Enter the number of terms 20
The sum of the series is: -0.222216
```

(*b*) A user-defined function file that calculates sin(*x*) by adding `n` terms of a Taylor series is shown below.

```
function y = Tsin(x,n)
% Tsin calculates the sin using Taylor formula.
% Input arguments:
% x The angle in degrees, n number of terms.

xr=x*pi/180;          Converting the angle from degrees to radians.
y=0;
for k=0:n-1
     y=y+(-1)^k*xr^(2*k+1)/factorial(2*k+1);   <- for-end loop.
end
```

The first element corresponds to $k = 0$, which means that in order to add n terms of the series, in the last loop $k = n - 1$. The function is used in the Command Window to calculate sin(150) using three and seven terms

```
>> Tsin(150,3)      Calculating sin(150°) with three terms of Taylor series.
ans =
      0.6523
>> Tsin(150,7)      Calculating sin(150°) with seven terms of Taylor series.
ans =
      0.5000                                  The exact value is 0.5.
```

Sample Problem: Modify vector elements

A vector is given by V = [5, 17, –3, 8, 0, –7, 12, 15, 20, –6, 6, 4, –7, 16]. Write a program as a script file that doubles the elements that are positive and are divisible by 3 or 5, and, raises to the power of 3 the elements that are negative but greater than –5.

Solution

The problem is solved by using a for-end loop that has an if-elseif-end conditional statement inside. The number of passes is equal to the number of elements in the vector. In each pass one element is checked by the conditional statement. The element is changed if it satisfies the conditions in the problem statement. A program in a script file that carries out the required operations is:

```
V=[5, 17, -3, 8, 0, -7, 12, 15, 20 -6, 6, 4, -2, 16];
n=length(V);
for k=1:n
   if V(k)>0 & (rem(V(k),3)==0 | rem(V(k),5)==0)
        V(k)=2*V(k);
   elseif V(k) < 0 & V(k) > -5
        V(k)=V(k)^3;
   end
end
V
```

Setting n to be equal to the number of elements in V.

for-end loop.

if-elseif-end statement.

The file is saved as Exp7_6 and then executed in the Command Window:

```
>> Exp7_6
V =
   10   17  -27    8    0   -7   24   30   40   -6   12    4
-8   16
```

`while-end` Loops:

`while-end` loops are used in situations when looping is needed but the number of passes is not known in advance. In `while-end` loops the number of passes is not specified when the looping process starts. Instead, the looping process continues until a stated condition is satisfied. The structure of a `while-end` loop is shown in figure below:

```
while conditional expression
   .....
   .....
   .....
end
```

A group of MATLAB commands.

The structure of a `while-end` loop.

The first line is a `while` statement that includes a conditional expression. When the program reaches this line the conditional expression is checked. If it is false (0), MATLAB skips to the end statement and continues with the program. If the conditional expression is true (1), MATLAB executes the group of commands that follow between the `while` and `end` commands. Then MATLAB jumps back to the while command and checks the conditional expression. This looping process continues until the conditional expression is false.

For a `while-end` loop to execute properly:

- The conditional expression in the `while` command must include at least one variable.
- The variables in the conditional expression must have assigned values when MATLAB executes the `while` command for the first time.
- At least one of the variables in the conditional expression must be assigned a new value in the commands that are between the `while` and the `end`. Otherwise, once

the looping starts it will never stop since the conditional expression will remain true.

An example of a simple while-end loop is shown in the following program. In this program a variable x with an initial value of 1 is doubled in each pass as long as its value is equal to or smaller than 15.

```
x=1                                  Initial value of x is 1.
while x<=15          The next command is executed only if x <= 15.
    x=2*x                            In each pass x doubles.
end
```

When this program is executed the display in the Command Window is:

```
x =                                  Initial value of x.
     1
x =
     2
x =                                  In each pass x doubles.
     4
x =
     8
x =          When x = 16, the conditional expression in the
    16       while command is false and the looping stops.
```

Important note:

When writing a `while-end` loop, the programmer has to be sure that the variable (or variables) that are in the conditional expression and are assigned new values during the looping process will eventually be assigned values that make the conditional expression in the `while` command false. Otherwise the looping will continue indefinitely (indefinite loop). In the example above if the conditional expression is changed to x >= 0.5, the looping will continue indefinitely. Such a situation can be avoided by counting the passes and stopping the looping if the number of passes exceeds some large value. This can be done by adding the maximum number of passes to the conditional expression, or by using the break command.

Since no one is free from making mistakes, a situation of indefinite looping can occur in spite of careful programming. If this happens, the user can stop the execution of an indefinite loop by pressing the **Ctrl + C** or **Ctrl + Break** keys.

Sample Problem : Taylor series representation of a function

The function can be represented in a Taylor series by $e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!}$.

Write a program in a script file that determines by using the Taylor series representation. The program calculates by adding terms of the series and stopping. When the absolute value of the term that was added last is smaller than 0.0001. Use a `while-end loop`, but limit the number of passes to 30. If in the 30th pass the value of the term that is added is not smaller than 0.0001, the program stops and displays a message that more than 30 terms are needed.

Use the program to calculate: e^{2}, e^{-4}, and e^{21}.

Solution

The first few terms of the Taylor series are:

$$e^{x} = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \ldots$$

A program that uses the series to calculate the function is shown next. The program asks the user to enter the value of x. Then the first term, **an**, is assigned the number 1, and **an** is assigned to the sum S. Then, from the second term on, the program uses a **while** loop to calculate the *n*th term of the series and add it to the sum. The program also counts the number of terms `n`. The conditional expression in the **while** command is true as long as the absolute value of the *n*th **an** term is larger than 0.0001, and the number of passes `n` is smaller than 30. This means that if the 30th term is not smaller than 0.0001, the looping stops.

```
x=input('Enter x ' );
n=1; an=1; S=an;
while abs(an) >= 0.0001 & n <= 30
    an=x^n/factorial(n);
    S=S+an;
    n=n+1;
end
if n >= 30
    disp('More than 30 terms are needed')
else
fprintf('exp(%f) = %f',x,S)
fprintf('\nThe number of terms used is: %i',n)
end
```

Start of the `while` loop.

Calculating the *n*th term.

Adding the *n*th term to the sum.

Counting the number of passes.

End of the `while` loop.

`if-else-end` loop.

The program uses an **if-else-end** statement to display the results. If the looping stopped because the 30th term is not smaller than 0.0001, it displays a message indicating this. If the value of the function is calculated successfully, it displays the value of the function and the number of terms used. When the program executes, the number of passes depends on the value of x. The program (saved as **expox**) is used to calculate e^{2}, e^{-4}, and e^{21}:

```
>> expox
```

```
Enter x 2                                          Calculating exp(2).
exp(2.000000) = 7.389046
The number of terms used is: 12                    12 terms used.
>> expox
Enter x -4                                         Calculating exp(–4).
exp(-4.000000) = 0.018307
The number of terms used is: 18                    18 terms used.
>> expox
Enter x 21                                         Trying to calculate exp(21).
More than 30 terms are needed
```

THE break ***AND*** continue ***COMMANDS***

The `break` command:

• When inside a loop (`for or while`), the **`break`** command terminates the execution of the loop (the whole loop, not just the last pass). When the **`break`** command appears in a loop, MATLAB jumps to the `end` command of the loop and continues with the next command (it does not go back to the `for` command of that loop).

• If the **`break`** command is inside a nested loop, only the nested loop is terminated.

• When a **`break`** command appears outside a loop in a script or function file, it terminates the execution of the file.

• The **`break`** command is usually used within a conditional statement. In loops it provides a method to terminate the looping process if some condition is met —for example, if the number of loops exceeds a predetermined value, or an error in some numerical procedure is smaller than a predetermined value. When typed outside a loop, the **`break`** command provides a means to terminate the execution of a file, such as when data transferred into a function file is not consistent with what is expected.

The `continue` command:

• The **`continue`** command can be used inside a loop (`for` or `while`) to stop the present pass and start the next pass in the looping process.

• The **`continue`** command is usually a part of a conditional statement. When MATLAB reaches the continue command, it does not execute the remaining commands in the loop, but skips to the `end` command of the loop and then starts a new pass.

Example: **Flight of a model rocke**

The flight of a model rocket can be modeled as follows. During the first 0.15s the rocket is propelled upward by the rocket engine with a force of 16 N. The rocket then flies up while slowing down under the force of gravity. After it reaches the

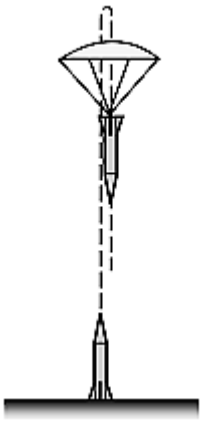
apex, the rocket starts to fall back down. When its downward velocity reaches 20 m/s a parachute opens (assumed to open instantly), and the rocket continues to drop at a constant speed of 20 m/s until it hits the ground. Write a program that calculates and plots the speed and altitude of the rocket as a function of time during the flight.

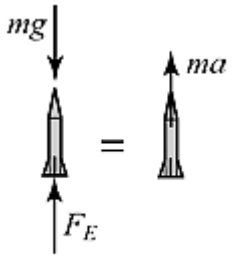
Solution

The rocket is assumed to be a particle that moves along a straight line in the vertical plane. For motion with constant acceleration along a straight line, the velocity and position as a function of time are given by

$$v(t) = v_0 + at \quad \text{and} \quad s(t) = s_0 + v_0 t + \frac{1}{2}at^2$$

where and are the initial velocity and position, respectively. In the computer program the flight of the rocket is divided into three segments. Each segment is calculated in a while loop. In every pass the time increases by an increment.





Segment 1: The first 0.15s when the rocket engine is on. During this period, the rocket moves up with a constant acceleration. The acceleration is determined by drawing a free body and a mass acceleration diagram (shown on the right). From Newton's second law, the sum of the forces in the vertical direction is equal to the mass times the acceleration (equilibrium equation)

$$+\uparrow \Sigma F = F_E - mg = ma$$

Solving the equation for the acceleration gives:

$$a = \frac{F_E - mg}{m}$$

The velocity and height as a function of time are:

$$v(t) = 0 + at \quad \text{and} \quad h(t) = 0 + 0 + \frac{1}{2}at^2$$

where the initial velocity and initial position are both zero. In the computer program this segment starts at $t = 0$, and the looping continues as long as $t < 0.15$ s. The time, velocity, and height at the end of this segment are , t_1, v_1, , and .

Segment 2: The motion from when the engine stops until the parachute opens. In this segment the rocket moves with a constant deceleration *g*. The speed and height of the rocket as functions of time are given by:

$$v(t) = v_1 - g(t - t_1) \quad \text{and} \quad h(t) = h_1 + v_1(t - t_1) - \frac{1}{2}g(t - t_1)^2$$

In this segment the looping continues until the velocity of the rocket is –20 m/s (negative since the rocket moves down). The time and height at the end of this segment are and .

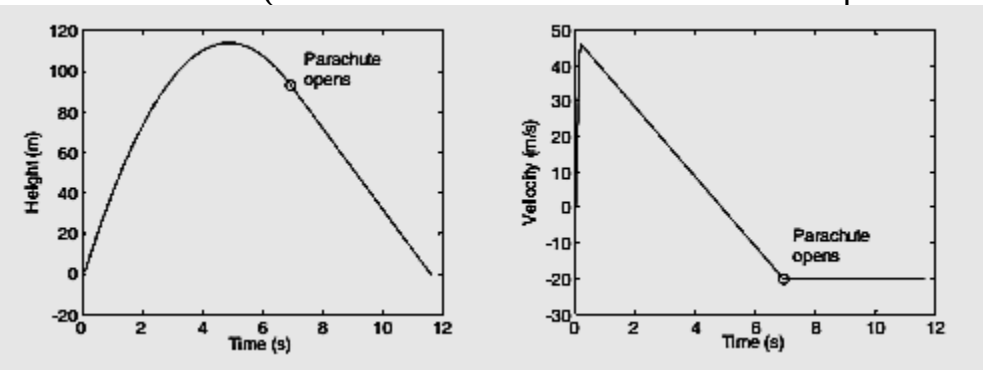
Segment 3: The motion from when the parachute opens until the rocket hits the ground. In this segment the rocket moves with constant velocity (zero acceleration). The height as a function of time is given by $h(t) = h_2 - v_{chute}(t - t_2),$ where v_{chute} is the constant velocity after the parachute opens. In this segment the looping continues as long as the height is greater than zero. A program in a script file that carries out the calculations is shown below.

```
m=0.05; g=9.81; tEngine=0.15; Force=16; vChute=-20; Dt=0.01;
clear t v h
n=1;
t(n)=0; v(n)=0; h(n)=0;
% Segment 1
a1=(Force-m*g)/m;
while t(n) < tEngine & n < 50000          The first while loop.
    n=n+1;
    t(n)=t(n-1)+Dt;
    v(n)=a1*t(n);
    h(n)=0.5*a1*t(n)^2;
end
v1=v(n); h1=h(n); t1=t(n);
% Segment 2
while v(n) >= vChute & n < 50000          The second while loop.
    n=n+1;
    t(n)=t(n-1)+Dt;
    v(n)=v1-g*(t(n)-t1);
```

```
    h(n)=h1+v1*(t(n)-t1)-0.5*g*(t(n)-t1)^2;
end
v2=v(n); h2=h(n); t2=t(n);
% Segment 3
while h(n) > 0 & n < 50000
   n=n+1;
    t(n)=t(n-1)+Dt;
    v(n)=vChute;
    h(n)=h2+vChute*(t(n)-t2);
end
subplot(1,2,1)
plot(t,h,t2,h2,'o')
subplot(1,2,2)
plot(t,v,t2,v2,'o')
```

The third `while` loop.

The accuracy of the results depends on the magnitude of the time increment `Dt`. An increment of 0.01 s appears to give good results. The conditional expression in the while commands also includes a condition for n (if n is larger than 50,000 the loop stops). This is done as a precaution to avoid an infinite loop in case there is an error in an of the statements inside the loop. The plots generated by the program are shown below (axis labels and text were added to the plots using the Plot Editor)



Example: **AC to DC converter**

A half-wave diode rectifier is an electrical circuit that converts AC voltage to DC voltage. A rectifier circuit that consists of an AC voltage source, a diode, a capacitor, and a load (resistor) is shown in the figure. The voltage of the source is $v_s = v_0 \sin(\omega t)$, where $\omega = 2\pi f$, in which f is the frequency. The operation of the circuit is illustrated in the lower diagram where the dashed line shows the source voltage and the solid line shows the voltage across the resistor. In the first cycle, the diode is on (conducting current) from $t = 0$ until $t = t_A$. At this time the diode turns off and the power to the resistor is supplied by the discharging capacitor. At $t = t_B$ the diode turns on again and continues to conduct current until $t = t_D$. The cycle continues as long as the voltage source is on. In this simplified analysis of this circuit, the diode is assumed to be ideal and the capacitor is assumed to have no charge initially (at $t = 0$). When the diode is on, the resistor's voltage and current are given by:

$$v_R = v_0 \sin(\omega t) \text{ and } i_R = v_0 \sin(\omega t)/R$$

The current in the capacitor is:

$$i_C = \omega C v_0 \cos(\omega t)$$

When the diode is off, the voltage across the resistor is given by:

$$v_R = v_0 \sin(\omega t_A) e^{(-(t-t_A))/(RC)}$$

The times when the diode switches off (t_A, t_D, and so on) are calculated from the condition $i_R = -i_C$. The diode switches on again when the voltage of the source reaches the voltage across the resistor (time t_B in the figure).

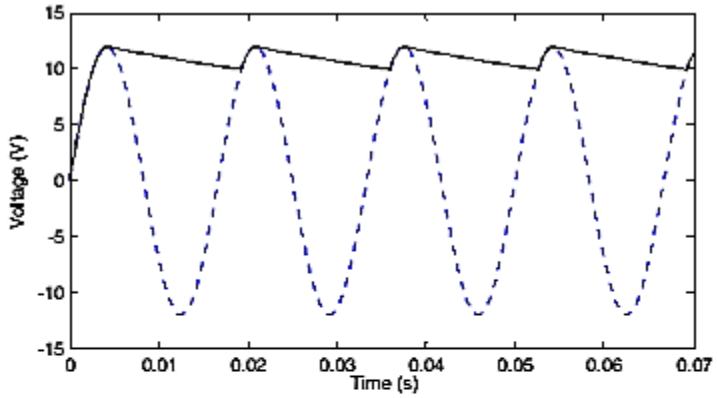
Write a MATLAB program that plots the voltage across the resistor and the voltage of the source as a function of time for $0 \le t \le 70$ ms. The resistance of the load is 1,800 Ω, the voltage source $v_0 = 12$ V, and $f = 60$ Hz. To examine the effect of capacitor size on the voltage across the load, execute the program twice, once with $C = 45$ μF and once with $C = 10$ μF.

Solution

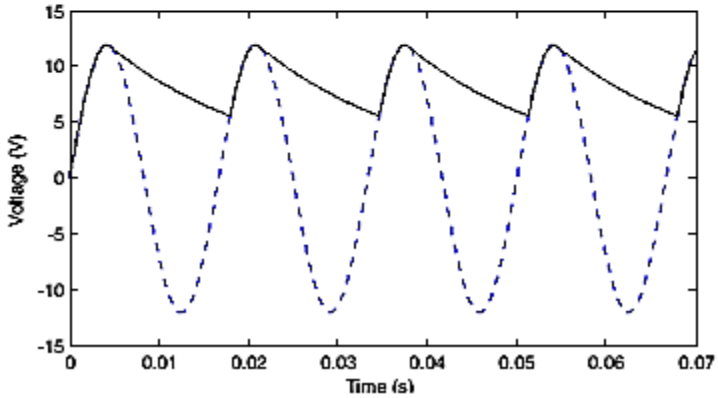
A program that solves the problem is presented below. The program has two parts—one that calculates the voltage v_R when the diode is on, and the other when the diode is off. The `switch` command is used for switching between the two parts. The calculations start with the diode on (the variable `state='on'`), and when $i_R - i_C \le 0$ the value of `state` is changed to '`off`', and the program switches to the commands that calculate v_R for this state. These calculations continue until $v_s \ge v_R$, when the program switches back to the equations that are valid when the diode is on.

```
V0=12; C=45e-6; R=1800; f=60;
Tf=70e-3; w=2*pi*f;
clear t VR Vs
t=0:0.05e-3:Tf;
n=length(t);
state='on'                          Assign 'on' to the variable state.
for i=1:n
   Vs(i)=V0*sin(w*t(i));            Calculate the voltage of the source at time t.
   switch state
      case 'on'                     Diode is on.
      VR(i)=Vs(i);
      iR=Vs(i)/R;
      iC=w*C*V0*cos(w*t(i));
      sumI=iR+iC;
      if sumI <= 0                  Check if iR - iC ≤ 0.
          state='off';              If true, assign 'off' to state.
          tA=t(i);                  Assign a value to tA.
      end
      case 'off'                    Diode is off.
      VR(i)=V0*sin(w*tA)*exp(-(t(i)-tA)/(R*C));
      if Vs(i) >= VR(i)             Check if vs ≥ vR.
         state='on';                If true, assign 'on' to the variable state.
      end
   end
end
plot(t,Vs,':',t,VR,'k','linewidth',1)
xlabel('Time (s)'); ylabel('Voltage (V)')
```

The two plots generated by the program are shown below. One plot shows the result with $C = 45\ \mu\text{F}$ and the other with $C = 10\ \mu\text{F}$. It can be observed that with a larger capacitor the DC voltage is smoother (smaller ripple in the wave).



$C = 45\ \mu F$



$C = 10\ \mu F$