25) A blizzard is a massive snowstorm. Definitions vary, but for our purposes we will assume that a blizzard is characterized by both winds of 30 mph or higher and blowing snow that leads to visibility of 0.5 miles or less, sustained for at least four hours. Data from a storm one day has been stored in a file *stormtrack.dat*. There are 24 lines in the file, one for each hour of the day. Each line in the file has the wind speed and visibility at a location. Create a sample data file. Read this data from the file and determine whether blizzard conditions were met during this day or not.

Ch5Ex25.m

```
% Reads wind and visibility data hourly from a file and
% determines whether or not blizzard conditions were met

load stormtrack.dat

winds = stormtrack(:,1);
visibs = stormtrack(:,2);

len = length(winds);
count = 0;
i = 0;
% Loop until blizzard condition found or all data
% has been read

while count < 4 && i < len
   i = i + 1;
   if winds(i) >= 30 && visibs(i) <= .5
       count = count + 1;
   else
       count = 0;
   end
end
if count == 4
    fprintf('Blizzard conditions met\n')
else
    fprintf('No blizzard this time!\n')
end
```

45) Write your own code to perform matrix multiplication. Recall that to multiply two matrices, the inner dimensions must be the same.

$[A]_{m \times n}\ [B]_{n \times p} = [C]_{m \times p}$

Every element in the resulting matrix C is obtained by:

c_{ij} =

$$\sum_{k=1}^{n} a_{ik} b_{kj}$$

.

So, three nested loops are required.

mymatmult.m

```
function C = mymatmult(A,B)
% mymatmult performs matrix multiplication
% It returns an empty vector if the matrix
%  multiplication cannot be performed
% Format: mymatmult(matA, matB)

[m, n] = size(A);
[nb, p] = size(B);
if n ~= nb
    C = [];
else
    % Preallocate C
    C = zeros(m,p);
    % Outer 2 loops iterate through the elements in C
    %   which has dimensions m by p
    for i=1:m
        for j = 1:p
            % Inner loop performs the sum for each
            %  element in C
            mysum = 0;
            for k = 1:n
                mysum = mysum + A(i,k) * B(k,j);
            end
            C(i,j) = mysum;
        end
    end
end
```

18) The lump sum S to be paid when interest on a loan is compounded annually is given by $S = P(1 + i)^n$ where *P* is the principal invested, i is the interest rate, and *n* is the number of years. Write a program that will plot the amount S as it increases through the years from 1 to *n*. The main script will call a function to prompt the user for the number of years (and error-check to make sure that the user enters a positive integer). The script will then call a function that will plot S for years 1 through n. It will use 0.05 for the interest rate and $10,000 for P.

Ch6Ex18.m

```
% Plots the amount of money in an account
% after n years at interest rate i with a
% principal p invested

% Call a function to prompt for n
n = promptYear;

% Call a function to plot
plotS(n, .05, 10000)
```

promptYear.m

```
function outn = promptYear
% This function prompts the user for # of years n
% It error-checks to make sure n is a positive integer
% Format of call: promptYear or promptYear()
% Returns the integer # of years

inputnum = input('Enter a positive integer for n: ');
num2 = int32(inputnum);
while num2 ~= inputnum || num2 < 0
    inputnum = input('Invalid! Enter a positive integer: ');
    num2 = int32(inputnum);
end
outn = inputnum;
end
```

plotS.m

```
function plotS(n, i, p)
% Plots the lump sum S for years 1:n
% Format of call: plotS(n,i,p)
% Does not return any values

vec = 1:n;
s = p * (1+i).^ vec;
plot(vec,s,'k*')
xlabel('n (years)')
ylabel('S')
end
```

26) The following script *land* calls functions to:

- prompt the user for a land area in acres
- calculate and return the area in hectares and in square miles
- print the results

One acre is 0.4047 hectares. One square mile is 640 acres. Assume that the last function, that prints, exists – you do not have to do anything for that function. You are to write the entire function that calculates and returns the area in hectares and in square miles, and write just a function stub for the function that prompts the user and reads. Do not write the actual contents of this function; just write a stub!

land.m

```
inacres = askacres;
[sqmil, hectares] = convacres(inacres);
dispareas(inacres, sqmil, hectares) % Assume this exists
```

askacres.m

```
function acres = askacres
acres = 33;
end
```

convacres.m

```
function [sqmil, hectares] = convacres(inacres)
sqmil = inacres/640;
hectares = inacres*.4047;
end
```

29) Write a menu-driven program to investigate the constant π. Model it after the program that explores the constant *e*. Pi (π) is the ratio of a circle's circumference to its diameter. Many mathematicians have found ways to approximate π. For example, Machin's formula is:

$$\frac{\pi}{4} = 4 \arctan\left(\frac{1}{5}\right) - \arctan\left(\frac{1}{239}\right)$$

Leibniz found that π can be approximated by:

$$\pi = \frac{4}{1} - \frac{4}{3} + \frac{4}{5} - \frac{4}{7} + \frac{4}{9} - \frac{4}{11} + \ldots$$

This is called a sum of a series. There are six terms shown in this series. The first term is 4, the second term is –4/3, the third term is 4/5, and so forth. For example, the menu-driven program might have the following options:

- Print the result from Machin's formula.
- Print the approximation using Leibniz' formula, allowing the user to specify how many terms to use.
- Print the approximation using Leibniz' formula, looping until a "good" approximation is found.
- Exit the program.

Ch6Ex29.m

```
% This script explores pi

% Call a function to display a menu and get a choice
choice = pioption;

% Choice 4 is to exit the program
while choice ~= 4
   switch choice
       case 1
           % Print result from Machin's formula
           pimachin
       case 2
           % Approximate pi using Leibniz,
           % allowing user to specify # of terms
           pileibnizn
       case 3
           % Approximate pi using Leibniz,
           %  until a "good" approximation is found
           pileibnizgood
   end
   % Display menu again and get user's choice
   choice = pioption;
end
```

pioption.m

```
function choice = pioption
% Print the menu of options and error-check
%  until the user pushes one of the buttons
% Format of call: pioption or pioption()
% Returns integer of user's choice, 1-4

choice = menu('Choose a pi option', 'Machin', ...
    'Leibniz w/ n', 'Leibniz good', 'Exit Program');
% If the user closes the menu box rather than
%  pushing one of the buttons, choice will be 0
while choice == 0
    disp('Error - please choose one of the options.')
    choice = menu('Choose a pi option', 'Machin', ...
    'Leibniz w/ n', 'Leibniz good', 'Exit Program');

end
end
```

pimachin.m

```
function pimachin
```

```
% Approximates pi using Machin's formula and prints it
% Format of call: pimachin or pimachin()
% Does not return any values

machinform = 4 * atan(1/5) - atan(1/239);

fprintf('Using the MATLAB constant, pi = %.6f\n', pi)
fprintf('Using Machin''s formula, pi = %.6f\n',4*machinform)
end
```

pileibnizn.m

```
function pileibnizn
% Approximates and prints pi using Leibniz' formula
% Prompt user for number of terms n
% Format of call: pileibnizn or pileibnizn()
% Does not return any values

fprintf('Approximate pi using Leibiz'' formula\n')

% Call a subfunction to prompt user for n
n = askforn;

approxpi = 0;
denom = -1;
termsign = -1;
for i = 1:n
    denom = denom + 2;
    termsign = -termsign;
    approxpi = approxpi + termsign * (4/denom);
end
fprintf('An approximation of pi with n = %d is %.2f\n', ...
    n, approxpi)
end

function outn = askforn
% This function prompts the user for n
% It error-checks to make sure n is a positive integer
% Format of call: askforn or askforn()
% Returns positive integer n

inputnum = input('Enter a positive integer for n: ');
num2 = int32(inputnum);
while num2 ~= inputnum | num2 < 0
   inputnum = input('Invalid! Enter a positive integer: ');
   num2 = int32(inputnum);
```

```
end
outn = inputnum;
end
```

27) A team of engineers is designing a bridge to span the Podunk River. As part of the design process, the local flooding data must be analyzed. The following information on each storm that has been recorded in the last 40 years is stored in a file: a code for the location of the source of the data, the amount of rainfall (in inches), and the duration of the storm (in hours), in that order. For example, the file might look like this:

```
     321   2.4   1.5
     111   3.3   12.1
          etc.
```

Create a data file. Write the first part of the program: design a data structure to store the storm data from the file, and also the intensity of each storm. The intensity is the rainfall amount divided by the duration. Write a function to read the data from the file (use **load**), copy from the matrix into a vector of structs, and then calculate the intensities. Write another function to print all of the information in a neatly organized table. Add a function to the program to calculate the average intensity of the storms. Add a function to the program to print all of the information given on the most intense storm. Use a subfunction for this function which will return the index of the most intense storm.

flooding.m

```
% Process flood data

flooddata = floodInfo;
printflood(flooddata)
calcavg(flooddata)
printIntense(flooddata)
```

floodInfo.m

```
function flood = floodInfo
% load flood information and store in vector
% Format of call: floodInfo or floodInfo()
% Returns vector of structures

load floodData.dat
[r c] = size(floodData);

for i=1:r
    flood(i) = struct('source',floodData(i,1),'inches',...
        floodData(i,2),'duration',floodData(i,3),...
        'intensity', floodData(i,2)/floodData(i,3));
end
```

```
end
```

calcavg.m

```
function calcavg(flooddata)
% Calculates the ave storm intensity
% Format of call: calcavg(flooddata)
% Returns average storm intensity

avginten = sum([flooddata.intensity])/length(flooddata);
fprintf('The average intensity of the storms is %.4f',...
   avginten);
end
```

printflood.m

```
function printflood(flooddata)
% Prints flood info
% Format of call: printflood(flooddata)
% Does not return any values

for i = 1:length(flooddata)
    fprintf('Flood Source: %d\n', flooddata(i).source)
    fprintf('Total Rainfall (in inches): %.2f\n', ...
        flooddata(i).inches)
    fprintf('Duration of Storm: %.2f\n', ...
        flooddata(i).duration)
    fprintf('Intensity: %.3f\n\n', ...
        flooddata(i).intensity)
end
end
```

printIntense.m

```
function printIntense(flooddata)
% Prints info on most intense storm
% Format of call: printIntense(flooddata)
% Does not return any values

ind = findind(flooddata);

fprintf('\nThe most intense recorded storm began')
fprintf(' flooding at location %d.\n', ...
    flooddata(ind).source)
fprintf('%.2f inches of rain fell in %.2f hours\n\n',...
   flooddata(ind).inches, ...
   flooddata(ind).duration)
end
```

```
function ind = findind(flooddata)
% Determines most intense storm
% Format of call: findind(flooddata)
% Returns index of most intense storm

intensity = [flooddata.intensity];
mostintense = intensity(1);
ind = 1;

%search for the highest intensity value
for i=1:length(intensity)
    if intensity(i) > mostintense
       %if higher intensity is found, save value and index
        mostintense = intensity(i);
        ind = i;
    end
end
end
```

32) Write a function *mydsort* that sorts a vector in descending order (using a loop, not the built-in sort function).

mydsort.m

```
function outv = mydsort(vec)
% This function sorts a vector using the selection sort
% Format of call: mydsort(vector)
% Returns the vector sorted in descending order

for i = 1:length(vec)-1
    highind = i;
    for j=i+1:length(vec)
        if vec(j) > vec(highind)
            highind = j;
        end
    end
    % Exchange elements
    temp = vec(i);
    vec(i) = vec(highind);
    vec(highind) = temp;
end
outv = vec;
end
```

3) The overall electrical resistance of *n* resistors in parallel is given as:

$$R_T = \left(\frac{1}{R_1} + \frac{1}{R_2} + \frac{1}{R_3} + \ldots + \frac{1}{R_n} \right)^{-1}$$

Write a function *Req* that will receive a variable number of resistance values and will return the equivalent electrical resistance of the resistor network.

Req.m

```
function resis = Req(varargin)
% Calculates the resistance of n resistors in parallel
% Format of call: Req(res1, res2, ... , resn)
% Returns the overall resistance

resis = 0;
for i = 1:nargin
    resis = resis + 1/varargin{i};
end
resis = resis ^ -1;
end
```

17) The velocity of sound in air is 49.02 $\sqrt{T}$ feet per second where *T* is the air temperature in degrees Rankine. Write an anonymous function that will calculate this. One argument, the air temperature in degrees R, will be passed to the function and it will return the velocity of sound.

```
>> soundvel = @ (Rtemp) 49.02 * sqrt(Rtemp);
```

28) The Fibonacci numbers is a sequence of numbers F_i:

0 1 1 2 3 5 8 13 21 34 …

where F_0 is 0, F_1 is 1, F_2 is 1, F_3 is 2, and so on. A recursive definition is:

$F_0 = 0$

$F_1 = 1$

$F_n = F_{n-2} + F_{n-1}$ if n > 1

Write a recursive function to implement this definition. The function will receive one integer argument n, and it will return one integer value that is the n^{th} Fibonacci number. Note that in this definition there is one general case but two base cases. Then, test the function by printing the first 20 Fibonacci numbers.

fib.m

```
function outval = fib(n)
% Recursively calculates the nth Fibonacci number
% Format of call: fib(n)
% Returns the nth Fibonacci number
```

```
if n == 0
    outval = 0;
elseif n == 1
    outval = 1;
else
    outval = fib(n-2) + fib(n-1);
end
end
```

```
>> for i = 1:20
   fprintf('%d\n', fib(i))
end
```