

## Applied Engineering Problem Solving (CHEN.3170)

### Part I: Matlab Overview

#### Lesson 3: Programming in Matlab

After the first two lessons, you should now be well on your way to becoming comfortable with Matlab for a variety of common tasks -- such as evaluating and plotting 1-D and 2-D functions,  $f(x)$  and  $f(x,y)$ , for example. Certainly, however, there is a need for more capability than what has already been illustrated. In particular, all programming languages, including Matlab, have a number of features that allow more advanced and more challenging tasks to be performed, and the goal of this lesson is to highlight some of the most important of these programming features.

Your texts by Gilat and Chapra do a good job on this subject, so my role, as usual, will be to hit the high points and then to give some practical examples that use the features discussed. After finishing this lesson you should have carefully read Chapters 1-7 and 10 in Gilat's text and Chapter 1-3 in the book by Chapra. There is a lot of good stuff in these chapters, so I recommend strongly that you take some quality time to give this material a thorough review with particular attention to the following subjects for the current lesson:

- Processing input and output data within Matlab (including proper internal documentation),
- Controlling the flow of a program via conditional tests and looping structures, and
- The use of function subprograms for developing well-structured programs.

We will begin our discussion with the difference between a Matlab script file and a function file. We have already written several script files that simply contain a series of Matlab commands that are interpreted and executed sequentially. Any variables defined in a script file are stored in the Matlab workspace and these are available for manipulation and further processing within the command window or another script file.

A function file, like a script file, always has a .m extension (i.e. an m-file), and it also contains a series of Matlab commands -- and they allow for the development of modular well-structured programs. However, there are several important differences, including the fact that all variables within the function are local only to that function -- that is, when the function is complete, any variables that were defined within the function are no longer available. The function communicates with the command window and other functions via a list of input and output arguments. Global variables can also be used to exchange information between functions and the Matlab workspace.

The specific syntax that defines a function file is as follows:

**function [output arguments] = function\_name(input arguments)**  
(do something useful, being sure to define the output arguments)

where the **function...** line must be the first executable line within the function file.

As an example, consider the following simple function file named **fxym.m**:

```
function f = fxy(x,y)
x = x+2;  z = y*y;
f = x*z;
```

This function file simply evaluates the function

$$f(x, y) = (x + 2)y^2 \quad (1)$$

for scalar variables  $x$  and  $y$ .

If  $x = 1$  and  $y = 2$ , the result should be  $f(1, 2) = (1 + 2) \times 2^2 = 3 \times 4 = 12$ . We can execute the function in the Matlab command window as follows:

```
>> x = 1; y = 2;
>> f = fxy(x, y)
f =
    12
```

and the answer that is returned is as expected.

Now, what are the values of  $x$  and  $y$  within the Matlab workspace? Inside the function file, the input variable,  $x$ , was reassigned to be  $x = x + 2$ , so you might expect  $x = 3$ . However,  $y$  is not re-defined inside **fxy.m**, so you would expect it has not changed. However, if we type the values of  $x$  and  $y$  to the screen,

```
>> x, y
x =
     1
y =
     2
```

we see that neither  $x$  or  $y$  have changed. This is because any variable inside a function is local only to that function. Thus,  $x$  inside **fxy.m** is not the same as  $x$  in the Matlab workspace and therefore,  $x$  from the command line is unchanged by the function. In fact, inside the function, the variable  $x$  could have any other name we choose, since what is important is the position of the variable in the argument list. For example, consider the function **fst.m** that also evaluates eqn. (1):

```
function f = fst(s, t)
f = (s+2)*t^2;
```

And, executing this from the Matlab command window gives the same result as before, or

```
>> f = fst(x, y)
f =
    12
```

Understanding this relatively straightforward behavior is critical for working with functions in Matlab -- please make sure you have a good handle on what is happening here!!!

Now, as mentioned above, data exchange can also occur between global variables (type **help global** in Matlab). To illustrate this, let's evaluate the same expression given in eqn. (1) in function **fxyz.m**:

```
function f = fxyz(x, y)
global z
x = x+2; z = y*y;
f = x*z;
```

where the only difference between this function and **fxy.m** is the definition, within the function, that  $z$  is a global variable.

Now, to see how this works, let's execute the following commands from the interactive command line:

```
>> clear all
>> global z
>> x = 1; y = 2; z = 3;
>> f1 = fxy(x,y)
f1 =
    12
>> z
z =
     3
>> f2 = fxyz(x,y)
f2 =
    12
>> z
z =
     4
```

Is it clear what has happened here? The **global z** statement defines *z* as a global variable in the Matlab workspace. Since a similar command is contained within function **fxyz.m**, when this function is executed, the *z = y\*y* assignment redefines the variable *z* and, since it is global, the new value is passed back to the *z* variable in the Matlab workspace. Thus, there are only two ways to get information into or out of a function file -- via the input and output argument lists or by defining the appropriate variables as being global variables (in all files where they are needed).

One additional feature of functions that offers a bit of flexibility in the design of a function is the use of a variable number of input and output arguments. Many of Matlab's built-in functions use this feature and, if programmed properly, this can add significant flexibility to your function files. In particular, Matlab defines two special variables, **nargin** and **nargout**, that are assigned values equal to the number of input variables and the number of output variables in the calling program, respectively. The values of **nargin** and **nargout** can be different each time the function is called, depending upon how the function subprogram is being used. Of course, to support the use of **nargin** and **nargout**, the functions must have the logic to decide how to process the input and output depending upon the value of these variables -- and we will give a brief example shortly as part of our study of the **loadColData.m** function (see below).

Program flow control is another essential ingredient of a full-featured programming language. In addition, the ability to repeat mathematical evaluations or groups of commands many times is a necessary part of many algorithms. Within Matlab, flow control is treated primarily with the **if ... else ... end** structure, and repeated operation of various code segments is achieved with either the **for ... end** or the **while ... end** constructs. In most cases, the **for ... end** looping structure is used when some fixed number of loops is desired, whereas the **while ... end** syntax is used when the loop is repeated until some test or condition is no longer valid. Note also that the **switch ... end** structure is sometimes useful as an alternative to a long sequence of **elseif** blocks within an **if ... elseif ... else ... end** structure.

Many of these programming features -- the **if ... else ... end**, **switch ... end**, and **while ... end** structures, require some mechanism for determining whether a conditional test is true or false. If the test is true, the loop is continued or the code segment within the **if ... else ... end** structure is executed. If, however, the test is not satisfied, then program flow proceeds to the next **elseif** or **else** condition or to the **end** of the structure. The tests are performed with a series of relational and logical operators (see your texts or the Matlab **help** facility for a full list and for a discussion

of operator precedence), including an equality test, `==`, a less than or equal to test, `<=`, the not equal to test, `~=`, etc., etc..

As a simple example, consider the following Matlab code:

```
>> format compact
>> x = [1 2 3 4 5]
x =
    1    2    3    4    5
>> y = [1 -2 3 -4 5]
y =
    1   -2    3   -4    5
>> z = x == y
z =
    1    0    1    0    1
>> z2 = x ~= y
z2 =
    0    1    0    1    0
```

After defining two arrays, **x** and **y**, we test to see if **x** is equal to **y** (note the double equal sign within the Matlab code for the equality test but a single equal sign for the assignment operator) and set the result to variable **z**. Similarly, a second test is made to determine if **x** is not equal to **y**. The result of these two tests is a vector of ones and zeros (true and false values, respectively). In particular, the **z** and **z2** arrays are logical variables, which can be verified with the *whos* command,

```
>> whos
  Name      Size      Bytes  Class

  x         1x5         40  double array
  y         1x5         40  double array
  z         1x5          5  logical array
  z2        1x5          5  logical array
```

Grand total is 20 elements using 90 bytes

**It is important to note that a conditional test on a logical array is true only if all the elements are true.** For example, the following code displays the second statement since the **if z == 1** statement fails,

```
>> if z == 1
    disp('x and y are identical')
else
    disp('At least one corresponding value of x and y are not equal')
end
```

At least one corresponding value of x and y are not equal

This means that, in most cases, you may want to check on individual elements of a logical array within a looping structure. For example, testing **if z(i) == 1** within a loop over **i** would give three true results and two false answers when using the logical **z** array from the above illustration.

Clearly, these logical tests can get complicated -- and I suggest that you do your best to “keep it simple”. I will certainly try to follow my own advice in subsequent examples, since code that is easy to follow is always preferable to code that uses clever programming tricks, even though they may give the same results...

The last major topic to be addressed in this lesson on Programming in Matlab deals with input and output operations and the proper documentation of your programs and analysis results. In practice, this is a very broad subject, because input and output operations can involve communication between the user and the program as well as information exchange between

programs via data files -- and there are a lot of different options here. Thus, this discussion will certainly not be exhaustive. Instead, we will try to stay as focused as possible on only the primary techniques for information transfer to and from Matlab programs.

The first point to emphasize is the need for good documentation of your Matlab programs and any analysis results that are generated within these programs. As noted above, it is important that your programs are easy to follow, and proper internal documentation via comments should help in this area. A brief discussion of the purpose and basic program logic at the top of the program is always extremely helpful. Also, you should always define the key variables within the program -- and please include units so that one knows exactly what the numbers mean. You should also use meaningful variable names that match, as closely as possible, the external documentation for your project or design study.

Also, you should always work with the base design variables within your programs. By this I mean, don't do a bunch of simple intermediate calculations by hand and then enter a numerical value for some combination of variables into your programs. You should enter the primary variables directly (dimensions, material properties, environmental inputs, conversion factors, etc.) into the code and do the intermediate calculations in Matlab. Checking things by hand is important and I certainly encourage this, but let Matlab do the calculations anyway, even if you have done some prior hand computations. With this approach, you will have direct access to the base design variables for future parametric studies and for good internal documentation.

Breaking a long program into several smaller script files and function subprograms, where each does some part of the overall analysis, is also extremely useful. This keeps the main program clean and the overall program logic straightforward -- with the details of the particular calculations or input and output processing steps segregated within their own subprograms. This approach also helps in debugging longer programs, since you can focus on one program unit at a time -- remember the "divide and conquer" philosophy.

You should also extend your good internal documentation practices to any printed or plotted results from your programs. We have already demonstrated some tools for annotating your plots with the use of well labeled titles and legends (see the *gtext*, *legend*, *title*, *num2str*, ..., commands), and this practice should be routine for all your Matlab programs -- a picture is only worth a thousand words if it is well annotated!!!

For printed output, Matlab has two primary functions -- the *disp* command and the *fprintf* function -- and we have used both briefly in previous examples. The *disp* function is used for relatively simple display tasks and the *fprintf* command, which is much more powerful, is used for formatted print to the screen or to a file. Your texts should be consulted for further details and for several short examples that use the *fprintf* function. And, of course, you should consult the Matlab *help* facility for more information as needed. However, in my opinion, the easiest way to see how to use these commands is via illustrative examples for practical problems, and we will supply many such cases at the end of this section of notes as well as in most of the subsequent lessons.

For simple user interaction with Matlab programs, the *input* and *menu* commands are quite useful. These commands are discussed in your texts and in the Matlab help files. These two commands are only useful when a few input variables are to be defined or a few options are to be selected at run time. In all cases, however, when the number of user inputs exceeds 4 or 5 parameters, the program should directly access a data file that contains the information of

interest. Data files can take on several forms and be as simple as a separate Matlab script file that defines the data and computational options for a specific problem, a simple space- or comma-delimited ascii file with columns of data that can be read or written with *dlmread* or *dlmwrite*, respectively, or a fairly complex ascii or binary file with mixed character and numerical information. Typing *help iofun* at the Matlab prompt will give you a list of the many file formats handled directly with specialized Matlab functions, and the various high-level and low-level file handling utilities that are available -- don't be too intimidated with the long list of functions given, since most of these are only needed in special circumstances (but it is nice to know that they are available, if needed).

As an example that uses many of the programming features discussed above, let's do a simulated Homework Problem (this problem was given as an actual student HW assignment in previous years). The problem focuses on analysis and plotting of data from an existing data file. The data describe the amount of rain that has fallen in Corvallis, Oregon for over 100 years. As part of the HW assignment, the students were asked to perform the following tasks:

1. Download the files **corvrain.dat** and **loadColData.m** from the course website.

Note that these files were originally obtained from [www.me.pdx.edu/~gerry/nmm/](http://www.me.pdx.edu/~gerry/nmm/), which is a website associated with a numerical methods textbook, *Numerical Methods with Matlab: Implementation and Application*, by Gerry Recktenwald. This website contains lots of supplementary information for this text, including the NMM toolbox (Numerical Methods with Matlab Toolbox).

2. Once you have the needed files, write a Matlab program to do the following:
  - a. Read the data using the **loadColData.m** function (see documentation within the file).
  - b. Compute and plot the yearly precipitation in inches.
  - c. Determine the average, minimum, and maximum annual precipitation for the period given. Also add a line on the above plot showing the average annual rainfall.
  - d. Finally, compute, plot, and tabulate the average rainfall by month.

The first part of the assignment asked the students to obtain two files from the course website. The two files are listed for reference purposes in Tables 1 and 2, where only a partial listing of the information in **corvrain.dat** is contained in Table 1 (the goal here is to show the file format and to fit it on one page for ease of visualization and presentation). Notice that the data file primarily contains columns of numerical data, with a single header line containing descriptive information about the file contents, and a single line of text data containing the column labels associated with each column of data in the file. This is a very common format used for storage of data.

Prof. Recktenwald wrote a generic Matlab function called **loadColData.m** to facilitate working with data files like that shown in Table 1. This function file is listed in Table 2. In addition to achieving its intended purpose, it is also a great example for illustrating many of the programming features mentioned above:

- It is a well-documented function file.
- It uses the **nargin** and **nargout** variables to handle variable input and output arguments.
- It has several *if ... end* conditional tests.

- It uses two nested *for ... end* structures, with an outer loop over all the rows containing column labels and an inner loop that processes each column label separately.

**Table 1 Partial listing of the corvrain.dat data file.**

Monthly Precipitation (hundredths of inches), Corvallis, Oregon												
year	Jan	Feb	Mar	Apr	May	Jun	Jul	Aug	Sep	Oct	Nov	Dec
1890	1062	888	567	182	29	123	50	14	0	165	22	380
1891	375	782	248	240	170	211	0	55	80	410	735	1183
1892	455	177	320	465	45	61	45	0	168	205	527	620
1893	235	540	455	518	366	60	0	9	324	567	828	402
1894	1238	529	812	281	190	294	10	5	120	445	210	535
1895	1165	155	445	306	436	28	39	0	215	0	464	1121
1896	835	324	313	698	571	98	0	116	41	330	1669	781
1897	284	698	571	173	109	209	9	38	157	238	1166	709
1898	382	548	234	244	226	133	23	12	315	159	863	362
1899	626	561	516	364	226	42	7	276	104	397	1093	757
1900	474	401	466	172	316	203	13	20	251	588	346	599
1901	847	628	415	309	186	109	0	18	272	223	621	508
1902	425	1029	696	393	286	27	144	0	116	171	1123	992
1903	724	164	268	267	103	109	34	44	37	173	1182	308
1904	556	1523	1170	182	76	32	85	11	59	274	748	905
1905	497	295	705	67	212	142	0	0	125	475	305	616
1906	722	676	242	238	307	282	0	0	224	270	935	675
1907	897	538	539	298	127	111	24	115	117	132	792	1333
1908	424	410	433	199	289	138	0	100	23	434	401	478
1909	1361	945	193	26	139	30	110	11	116	451	1153	496
1910	654	754	121	247	83	173	0	1	85	256	1045	433

a portion of the data was omitted here to save space (format is identical)

1980	669	388	402	363	146	175	24	1	96	187	629	1133
1981	227	444	300	237	299	258	10	1	309	552	673	1398
1982	721	712	354	457	49	151	43	28	189	364	551	1056
1983	691	1031	878	301	151	139	255	221	53	105	993	735
1984	326	692	382	341	367	434	20	0	74	465	1355	401
1985	25	365	494	105	94	222	54	48	78	389	469	372
1986	653	990	304	184	250	31	115	0	356	280	862	350
1987	822	450	370	156	140	29	223	17	5	27	390	1142
1988	712	170	390	333	384	183	9	0	73	14	1087	397
1989	418	321	680	142	146	114	33	87	60	266	390	307
1990	950	579	221	238	143	153	45	172	83	456	487	354
1991	268	322	585	347	391	152	38	72	19	255	513	438
1992	452	454	104	408	0	118	118	44	55	352	499	738
1993	415	220	486	682	451	211	79	31	7	107	103	798
1994	389	559	346	194	113	189	1	0	89	384	903	626

**Table 2 Listing of Matlab file loadColData.m.**

```

function [x,Y,labout] = loadColData(fname,ncol,nhead,nrowl)
% loadColData Import a file containing header text, column titles and data
%
% Synopsis:  [x,Y] = loadColData(fname)
%            [x,Y] = loadColData(fname,ncol)
%            [x,Y] = loadColData(fname,ncol,nhead)
%            [x,Y] = loadColData(fname,ncol,nhead,nrowl)
%            [x,Y,labels] = loadColData(fname)
%            [x,Y,labels] = loadColData(fname,ncol)
%            [x,Y,labels] = loadColData(fname,ncol,nhead)
%            [x,Y,labels] = loadColData(fname,ncol,nhead,nrowl)
%
% loadColData can read plain text files with the following format
%
%      header line 1 ...
%      header line 2 ...
%      ...
%      col_1_label      col_2_label      col_3_label ...
%      col_1_label2     col_2_label2     col_3_label2 ...
%      ...
%      number           number           number ...
%      number           number           number ...
%      number           number           number ...
%      ...
%
%      where ... indicates any number of similar lines or columns
%
% Input:  fname = (string) name of the file containing the data (required)
%         ncol = total number of columns of data. Default: ncol = 2
%         nhead = number of lines of header information at the very top of
%               the file. Header text is read and discarded. Default = 0.
%         nrowl = number of rows of labels. Default: nrowl = 1
%
% Output: x = vector of values from the first column of data
%         Y = matrix of values from the second through ncol column of data
%         labels = (string) matrix of labels. To provide for labels of
%               arbitrary length, THE LABELS FOR EACH COLUMN OF DATA
%               ARE STORED IN SEPARATE ROWS OF THE labels MATRIX.
%               Thus, label(1,:) is the label for the first column,
%               label(2,:) is the label for the second column, etc.
%               More than one row of labels is allowed. In this case
%               the second row of the label for column one is
%               label(1+ncol,:). NOTE: Individual column headings
%               must not contain blanks.
%
% Gerald Recktenwald, gerry@me.pdx.edu
% Portland State University, Mechanical Engineering Department
% 24 August 1995, revised 29 April 1998, 27 Feb 1999
%
if nargin<2, ncol = 2; end
if nargin<3, nhead = 0; end
if nargin<4, nrowl = 1; end
%
% --- Open file for input, include error handling
fin = fopen(fname,'rt'); % read as plain text
if fin < 0
    error(['Could not open ',fname,' for input']);
end
%
% --- Read and discard header text one line at a time
for i=1:nhead, buffer = fgetl(fin); end
%
% --- Read column titles
labels = ''; % Initialize the labels matrix
for i=1:nrowl
    buffer = fgetl(fin); % Get next line as a string
    for j=1:ncol
        [next,buffer] = strtok(buffer); % Parse next column label
    end
end

```



```

        labels = str2mat(labels,next);    % Add another row to the labels matrix
    end
end
if nrowl>0
    labels(1,:) = []; % delete first row created when labels matrix initialized
end

% --- Read in the x-y data
data = fscanf(fin,'%f');                % Load values into a column vector
fclose(fin);                            % Close file, release handle
nd = length(data);                      % Total number of data points
nr = nd/ncol;                           % Number of data rows after reshaping
if nr ~= round(nd/ncol)
    fprintf('Error in loadColData:\n');
    fprintf('\tnumber of data points = %d does not equal nrow*ncol\n',nd);
    fprintf('\tdata: nrow = %f\tncol = %d\n',nr,ncol);
    fprintf('\nHere are the column labels\n\t');
    for j=1:ncol, fprintf('%s ',labels(j,:)); end, fprintf('\n');
    error(sprintf('data matrix cannot be reshaped into %d columns',ncol))
end

data = reshape(data,ncol,nr)';          % Notice the transpose operator
x = data(:,1);
Y = data(:,2:ncol);
if nargout>2, labout = labels; end

```

- It uses several IO routines (*fopen*, *fgetl*, *strtok*, *fscanf*, and *fclose*) to read the ascii data file, and the *fprintf* and *sprintf* functions to write information to the screen and to a variable string (for use in the *error* function).
- It introduces several useful built-in functions (*str2mat*, *round*, *reshape*, and *error*) that we have not mentioned yet.

Thus, this relatively short program has two important roles; as a very useful function for reading ascii data files and as an excellent example of programming in Matlab. I suggest that you review this file carefully and use Matlab's *help* facility to determine the purpose and syntax of each new command or function whose use is not obvious.

Now, getting back to our simulated HW assignment, we want to use Prof. Recktenwald's **loadColData.m** function to read the **corvrain.dat** file for further processing. This was done in the **corvrain\_hwdemo.m** file as shown in Table 3. The tasks performed here directly follow the HW description given above. Once the data file is processed in Task 1, the remaining processing and plotting tasks are quite straightforward, using several Matlab commands which we have seen before (*sum*, *max*, *min*, *plot*, *bar*, *num2str*, etc.). In addition, the *fprintf* function is used to write some summary data to the screen (note that unit 1 defaults to the monitor screen).

The graphical output from our program is shown in Figs. 1 and 2 and the summary printed output is presented in Table 4. As apparent, the average rainfall in Corvallis, Oregon is about 40 inches, with yearly swings that can exceed  $\pm 15$  inches around this mean value. Also, as clearly indicated in Fig. 2, the late fall and winter months have much more precipitation than the late spring to late summer time frame. Thus, as a tourist visiting this part of the country, I suggest that the best chance for good vacation weather is from May through September!

The above HW demo is a good example of the type of programs you will be asked to generate as part of this course and in your various assignments as a working engineer. Performing data

processing and analyzing the performance of a particular process or product are common tasks that you will face -- and many of these will require that you perform a variety of pre or post processing manipulations of the original data.

**Table 3 Listing of corvrain\_hwdemo.m.**

```
%
% CORVRAIN_HWDEMO.M    MATLAB file to process summary data concerning the
%                      amount of rainfall in Corvallis, Oregon from 1890 to 1994
%
% Several tasks will be performed here given the data file corvrain.dat:
% (file obtained from the NMM toolbox at www.me.pdx.edu/~gerry/nmm/ )
% 1. read the data file corvrain.dat using the loadColData.m file (also
%    from the NMM toolbox)
% 2. compute and plot the total yearly precipitation in inches
% 3. determine the average, maximum, and minimum annual precipitation
%    also add a line showing the average value to the plot from Task 2
% 4. compute, plot, and tabulate the average rainfall by month
%
% Reference: The idea for this problem was obtained from Prob. 3.28 on
% pgs. 145-146 of the text "Numerical Methods with Matlab: Implementation
% and Application," by G. Recktenwald, Prentice Hall, Inc., 2000.
%
% File prepared by J. R. White, UMass-Lowell (last update: Sept. 2017)
%

clear all, close all, nfig = 0;

% Task 1: use loadColData to read the desired data file
% --> resulting data: yr = vector containing the date (year)
%                  rain = matrix containing the rainfall (in hundreds of
%                  an inch) for each month and year
% [yr,rain] = loadColData('corvrain.dat',13,1,1);

% Task 2: compute and plot the yearly total precipitation (inches)
% rain = rain/100;           % converts hundreds of inch to inches
% yrrain = sum(rain');       % sums 12 months of data to get yearly total

% nfig = nfig+1; figure(nfig)
% plot(yr,yrrain,'r-','LineWidth',2), grid, hold on
% title('CorvRain\_HWDemo: Yearly Rainfall for Corvallis, Oregon')
% xlabel('year'),ylabel('amount of rain (inches)')

% Task 3: determine the average, maximum, and minimum annual precipitation
% also add a line showing the average value to the plot from above
% nyr = length(yr);           % number of years
% averain = sum(yrrain)/nyr;   % average value
% [minrain,imin] = min(yrrain); yrmin = yr(imin); % min rain and year it occurred
% [maxrain,imax] = max(yrrain); yrmax = yr(imax); % max rain and year it occurred

% fout = 1;
% fprintf(fout,'\n\n');
% fprintf(fout,'    CorvRain_HWDemo: Summary Rainfall Results (in inches) for Corvallis,
Oregon \n\n');
% fprintf(fout,'    average annual rainfall:           %8.1f \n',averain);
% fprintf(fout,'    minimum annual rainfall and year:      %8.1f   %5d \n',minrain, yrmin);
% fprintf(fout,'    maximum annual rainfall and year:      %8.1f   %5d \n',maxrain, yrmax);

% plot([1880 2000],[averain averain],'b-','LineWidth',2)
% gtext(['average rainfall = ',num2str(averain,'%4.1f'),' inches']), hold off

% Task 4: compute, plot, and tabulate the average rainfall by month
% monthrain = sum(rain);       % sums over all years to get monthly totals
% avemrain = monthrain/nyr;    % ave value by month

% nfig = nfig+1; figure(nfig)
% bar(avemrain), grid, range = axis; range(2) = 13; axis(range)
```

```

title('CorvRain\_HWDemo: Monthly Rainfall for Corvallis, Oregon')
xlabel('month of the year'),ylabel('amount of rain (inches)')

%
months = ['Jan. ','Feb. ','March','April','May ','June ','
          'July ','Aug. ','Sept.','Oct. ','Nov. ','Dec. '];
fprintf(fout,'\n');
fprintf(fout,'      Average Monthly Rainfall Values (in inches)\n');
fprintf(fout,'      Month          Amount  \n');
for n = 1:12
    fprintf(fout,'          %8s      %8.1f \n',months(n,:),avemrain(n));
end
%
% end of program

```

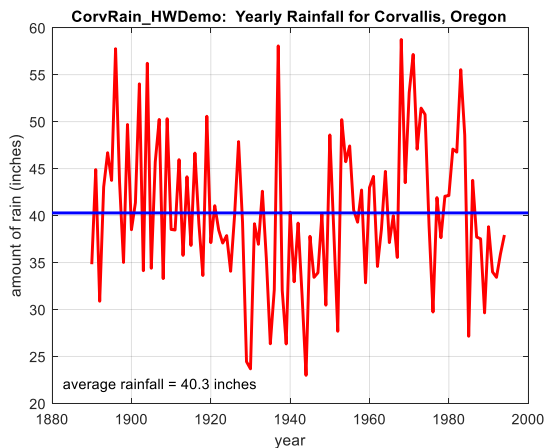


Fig. 1 Yearly rainfall for Corvallis, Oregon.

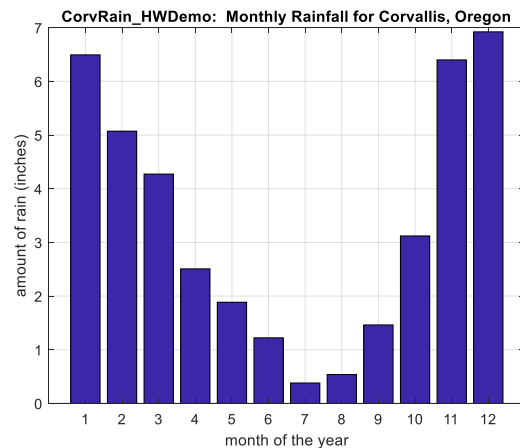


Fig. 2 Average monthly rainfall for Corvallis, Oregon.

**Table 4 Printed summary results from corvrain\_hwdemo.m.**

CorvRain\_HWDemo: Summary Rainfall Results (in inches) for Corvallis, Oregon

average annual rainfall:	40.3	
minimum annual rainfall and year:	23.0	1944
maximum annual rainfall and year:	58.7	1968

Average Monthly Rainfall Values (in inches)

Month	Amount
Jan.	6.5
Feb.	5.1
March	4.3
April	2.5
May	1.9
June	1.2
July	0.4
Aug.	0.5
Sept.	1.5
Oct.	3.1
Nov.	6.4
Dec.	6.9

As in previous lessons, three additional worked-out examples provide more illustrative applications where many of the Matlab programming features discussed in this lesson are used to advantage. These examples are available as separate pdf files, as follows:

**ed96.pdf -- Hurricane Edouard (Aug. 21 – Sept. 3, 1996)**

**rect1d\_fin\_1.pdf -- Heat Transfer in a Rectangular Fin**

**pin\_fin\_1.pdf -- Measurement Error in a Temperature Probe**

After you have finished your reading assignment for this lesson and after reviewing the above examples, you should be ready to attempt HW #3 (see **hw3xxx.pdf**). This homework usually involves 3 or 4 problems (sometimes separated into two homeworks, HW3a and HW3b) that require writing Matlab programs that use several of the programming features discussed in this lesson. Although the individual problems are not difficult, collectively they represent a challenging HW assignment that will take some quality time to complete -- so give yourself plenty of time to complete this assignment (or set of assignments, as appropriate).

As done for the previous homeworks, I prefer that you collect the Matlab m-files (don't forget the function files), the resultant plots and/or tabular results, any hand calculations, and a brief description of the results of each problem in a separate solution packet for each HW problem. Thus, for HW #3, you should prepare several separate solution packets, as appropriate, and put these together in a professional manner for submittal to me by the posted HW deadline.

Well, this marks the end of the first part of this course -- **Part I: Matlab Overview**. With the completion of this material, you should have a good understanding of programming in Matlab and a good foundation for solving engineering-related problems with this programming tool. We will see many more examples where Matlab can be utilized in addressing a variety of analysis issues, but the focus will now shift towards highlighting several different numerical methods that are needed as tools for solving realistic engineering design and analysis problems. As we shift to the second part of the course -- **Part II: Numerical Methods and Applications** -- we will continue to exploit the power of Matlab for a variety of problem-solving tasks. Unavoidably, however, we will need to introduce a fair amount of mathematics into our discussions, but the focus will always be on applied engineering problem solving, with Matlab as our primary programming tool...

I am ready to jump to Part II -- are you???