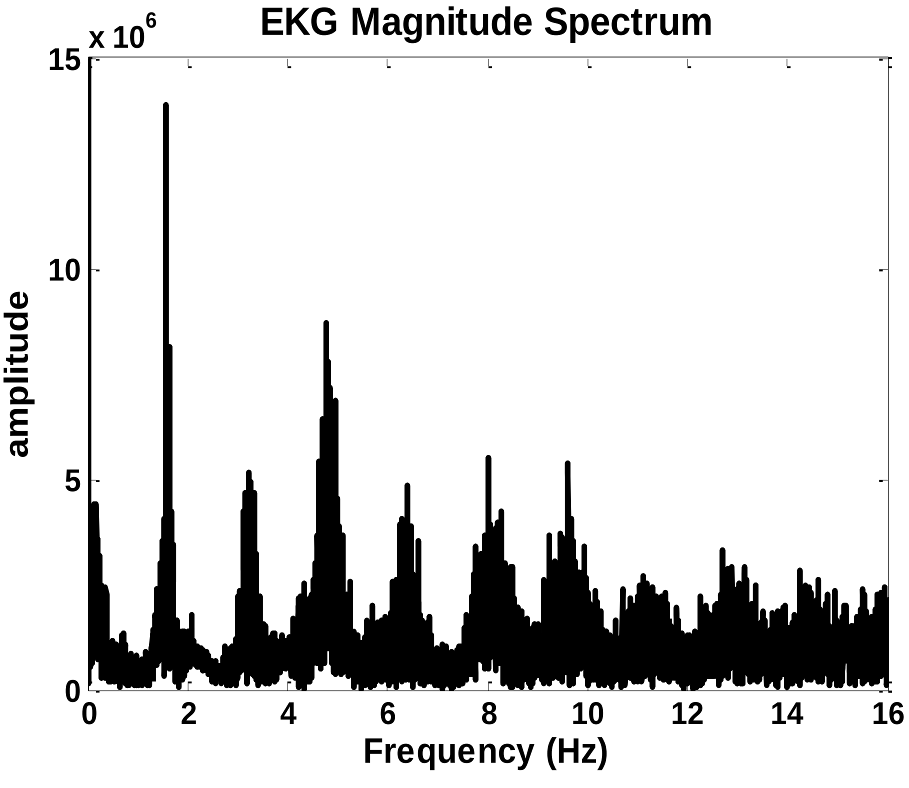
Digital Signal Processing
EE431

Virginia Military Institute

Student Laboratory Manual



James C. Squire, P.E., Ph.D. 1 August, 2014

Table of Contents

Note to students

I wrote these laboratories because I could not find an "ideal" laboratory manual in the roughly twenty currently published. Most were very abstract and involved neither programming nor practical application; although this purely mathematical treatment serves as a good preparation for graduate school it is nearly useless as a preparation for industry. Others stressed practical applications in a "cookbook" manner. These typically provided pre-written software that solved a particular problem. Although the programs worked as advertised, they did a poor job of teaching how and why the algorithms worked, and so did not prepare graduates how to modify them to solve different problems they may encounter. The treatment I follow is a mixture; the first part of the labs are typically theoretic and provide solid preparation should you decide to later attend graduate school. The last section of most of the laboratories provides a real-world, common problem that can be solved using the techniques you learned in the earlier sections. Problems are given throughout the laboratory in bolded boxes, immediately following the section that describes their solution method.

In addition to the standard problems in each laboratory, there are also a series of Challenge problems. These are typically much harder, include less detail, and further develop an aspect of a solution technique introduced in the laboratory. These problems are designed to provide a significant challenge in sections for which you feel completely comfortable. In a sense, this entire course as an upper level course synthesizing elements from advanced applied mathematics and practical engineering is a "challenge problem"; do not feel compelled to work the problems identified as "Challenge problems" unless you feel bored with the regular problems before them; however the highest possible grades will go to the students that can solve them all.

Writing laboratory reports is traditionally a bugbear of engineering students. They can easily take the same time as that required for the laboratory, but depending on the format can be very tedious to author. In my past laboratories you have had to complete reports in a standard Objectives / Procedure / Data / Discussion / Conclusions format that mirrors professional papers you will write in industry, journals in graduate school, and the somewhat arbitrary format it enforces is not unlike that of the Operations Order for those commissioning. You will be happy to know that you will not need to adhere to such a format for this course. Because of the amount of material you will cover in these labs, I have developed a fill-in-the-blanks report available on the course webpage for every lab except the individual project lab. You will find these tremendous time-savers. You will need them.

Although I worked every problem in these labs there is still likely to be mistakes, misspellings, and areas that could be written more clearly. I welcome all your comments that will help make this laboratory manual a useful reference for the rest of your engineering careers.

Laboratory 1: Matlab Review

1. Objectives

- Review basic Matlab commands, both for scalars and vectors
- Review how to perform complex arithmetic in Matlab
- Review how to plot complex-valued results
- Review the differences between scripts and functions
- Improve programming skills

2. Introduction

A. Background

Matlab is the de facto standard tool in both academia and industry for developing and testing new DSP algorithms. It is only slightly less fast than highly-optimized C code, and much faster than C to design, debug, and view. Industry typically uses Matlab to develop and test new algorithms, and then selects a cheap microcontroller, dedicated DSP chip, or FPGA to implement the algorithms (a cellphone attached to a PC running Matlab would not sell well). A DSP chip is nearly identical to a microcontroller, but with additional features that lend it to specific DSP tasks (such as having a dedicated multiply-add instruction which are used in filtering signals, and having large dedicated input/output buffers to accept and output streaming signals).

Lately, industry has begun using Matlab in applications for *Hardware-In-The-Loop Development*. This refers to starting a design purely using simulation, and gradually adding real-hardware blocks to the simulation until the hardware has completely replaced the simulation blocks and the system is designed. An example is Sparta's recently developed Javelin II shoulder-launched self-guided anti-tank weapon. At first the weapon was simulated using one program that modeled the dynamics of the rocket engine (it took a stream of steering and thrust parameters and output a stream of location information), another that modeled the sensors (it took a stream of actual location information and output a stream of simulated GPS coordinates and simulated video images), and a third program that modeled the control algorithm (it took the stream of sensor information and output a stream of steering and thrust parameters for the rocket engine). Each was modeled in Matlab, and connected together. Next, the actual rocket engine was built and mounted on a test stand, and put in the place of the simulated rocket engine. Real sensors measuring engine thrust were next added, replacing the simulated sensors. Finally, the control algorithm was fine-tuned, placed within an FPGA, and connected to the completed system.

When you leave this course, you will be very proficient in Matlab, and ready to tackle any graduate-level DSP course (should you take that path), or industry development tasking.

B. Laboratory administration

1) Help received

All laboratories will be worked in teams of two students. There may be at most a single team of 3 or an individual working alone if there are an odd number of students. Except as noted below, help received outside the lab team is not authorized and will result in a 50% deduction to every section in which help received is indicated (help received not relevant to completing specific problems, such as how to start Matlab, is fine). You don't need to document help received by your lab partner, by the instructor, by the help system incorporated into Matlab, or the Mathworks website.

2) **Lab Report**
To make your work more efficient and allow you to complete more DSP learning in our allotted lab time, I have prepared an outline report for each of your labs. You may download it from my website at http://www.jimsquire.com. Most of the material can be simply cut-and-pasted in directly from your Matlab session. You don't need to add connecting text; if I want an essay type response I will specifically ask for it. Only one lab report per team needs to be completed. Share the work evenly; at the end of the course I will ask you to evaluate how well you and your lab partner shared the workload, and this will influence 15% of your lab grade.

3) **Attendance**
At this point in your career, I trust you to control your time schedule. You may do as much or as little of the work for each lab as you desire during the scheduled lab time. I have everything posted on the web, so should you desire to do them the night before, come in, hand me your report, and leave, that is fine. Similarly you may leave and do the lab later on your own. I discourage this, since you are not authorized to receive help from outside your lab team and I can assist your team during lab in ways that can save you substantial amounts of time. Still, I understand you juggle many responsibilities and that occasionally other issues will take priority, and I respect your judgment.

3. Laboratory

1. Login and start Matlab

2. Create a data folder structure

You should create a new data folder for this course and under it a new folder for this laboratory, and make it Matlab's current working directory. The current working directory is the directory in which Matlab expects to find the user's functions (i.e. m-files) and scripts.

a) Locate the Current Directory type-in box in the toolbar at the top of the Matlab main window
b) Navigate to your personal directory on the network.
c) Inside your personal directory, create a new subfolder called EE431 (don't use a space).
d) Inside that folder, create a subdirectory called Lab1 (with no space).
e) Note that the Current Directory type-in box in the toolbar now points to your Lab1 working directory.

3. Evaluating Complex Variables and Expressions

Examples:

`z = 3 + j*6`	% creates complex variable z. Multiplication must be made explicit with a * symbol.
`z = j*exp(4)`	% replaces variable z with the number je^4.
`z = 2^8;`	% now z is set equal to 256.
`real(z)`	% returns the real part of z
`imag(z)`	% returns the imaginary part of z
`abs(z)`	% returns the magnitude of complex variable z
`angle(z)`	% returns the angle of complex variable z in radians. You must multiply by $180/\pi$ to get degrees.
`pi`	% returns 3.14159
`log(4)`	% returns the natural log of 4, ie what is usually typeset as ln(4)
`log10(4)`	% returns the base-10 log of 4, ie what is usually typeset as log(4)

Note: Frequently in the labs I give examples such as those above. You do not need to enter them; your lab reports will consist only of the boxed problems such as the one below. You may find it helpful to enter and play around with some of the more complicated commands.

Note: Know what matrix (including vector) multiplication means. [1 2] * [3 4] gives an error since matrix multiplication is only defined when the number of columns of the first matrix (here 2) equals the number of rows of the second (here 1). [1 2] * [3; 4] (where the semicolon means "start a new row") is defined (since 2 = 2) and gives a matrix answer with the number of rows of the first number (1) and the number of columns of the second number (also 1)…it gives a single number answer of 1*3 + 2*4 = 11. If you want the answer [1*3 2*4] = [3 8] then prefix the * sign with a period, like this: [1 2] .* [3 4] = [3 8].

Yet another note: In Word you may occasionally want to use a mathematical or Greek symbols like 2 ± 0.3 or 4 ∠ 34° or δ[n]. You can find them all in the Symbol font. Choose Insert, Symbol, and change the font to Symbol.

Problem 1
Express each of the following complex numbers in rectangular form using Matlab:
a) $j\,e^{j\,11\pi/4}$
b) $(1-j)^{10}$

Problem 2
Express each of the following complex numbers in magnitude/angle form in radians using Matlab:
a) $j\,e^{j\,11\pi/4}$
b) $3 + j6$

4. Creating and plotting vectors and matrices

Examples

```
v = 1:10;                    % shorthand for saying v = [1 2 3 4 5 6 7 8 9 10];
v = 0:0.1:5;                 % creates the vector [0  0.1  0.2  0.3…4.8  4.9  5]
v = linspace(0,2,50);  % creates a vector of 50 points, linearly spaced between 0 and 2
m = zeros(4,5);              % creates a matrix of zeros that is 4 rows by 5 columns
v = ones(1,10);              % creates a column vector of ones that is 10 long.
plot(x,y);                   % plots x vs. y using lines to join the {x,y} point locations
plot(x,y,'.');               % plots a single point at each {x,y} location
title, xlabel, ylabel % labels the plot
```

To insert a plot figure into a Word document, from the figure's menu choose Edit → Copy Figure; then cut and paste into Word.

To plot the equation y = x/2 * cos(x) between 0 and 5

a) First create the x vector with enough points to give a smooth plot
   `x = linspace(0,5,100);`

b) Next, find y at each point location x
   `y = x ./ 2 .* cos(x);`

   Note 1: when working with vectors you must use .+, .-, .*, ./, and .^ for element-by-element add, subtract, multiply, divide, and raise to a power.

   Note 2: cos(v), sin(v), tan(v), exp(v), log(v), and log10(v) work fine for scalar, vector, or matrix arguments.

c) Plot it using `plot(x,y)`

   Note 1: You can specify the color of the plot using plot (x,y,'r'), where r is red, b is blue, k is black, and g is green

Note 2: You can specify the line style of the plot using plot(x,y,'b-') for a solid blue line, 'r--' for a dashed red line, or 'g:' for a dotted green line. plot(x,y, 'b.') plots a single blue dot at the location of each x,y pair and does not join x,y pairs by a line of dots, unlike the other line styles.

Problem 3: Plot y = real part of ($e^{j x \pi}$) as x varies between 0 and 5. Neatly label the plot.

5. Saving and loading variables

First, copy the data files for this lab to your working directory.

a) Open a web browser and point it to my home page (www.jimsquire.com).
b) Navigate to the EE431 course page (Teaching → EE431)
c) Under the "Lab" section, right-click each data file and save it to the current working Matlab directory. This time there is only a single file called Lab1.mat; just download this one.

You can see what variables are currently defined by typing

```
whos
```

You can also see the variables on the Workspace window that is usually to the left of the main Command window. Notice that not only the defined variables are listed, but also what size they are. Type it now.

You can load and save variables (for instance variables x and y) used by Matlab to a file (for instance, myfilename) using the command (note: no commas)

```
save myfilename x y
```

Matlab will create a single file called myfilename.mat that will hold x and y. Note the .mat extension is automatically added, and that the syntax is the same regardless of whether x and y are scalars, vectors, or matrices. Also note that the file created is not an ASCII file readable by, say, Notepad; the variables are stored in a compressed form. You can also store variables in ASCII form, perhaps to be read by an external program; see help save for details.

The word "workspace" refers to all the variables currently defined by Matlab. After working for a while it is easy to get confused by the number of variables you have defined and you may desire to clear some of them. To clear variables x and y, for instance, type

```
clear x y
```

Now type `whos` to see the effects.

To quickly clear all of the variables in the workspace, type

```
clear
```

by itself. Do this now, and then type whos to see the effects.

To load variables from a file, type

```
load myfilename
```

The .mat extension is optional. Do this now to load the file Lab1.mat that you earlier copied from my homepage. You can also read in an ASCII text file; see help load for details.

Problem 4

Using the above commands, determine the variable names I have stored in Lab1.mat.

6. Accessing vectors and matrices

To set scalar variable N to the length of vector v, type

```
N = length(v);
```

You could see if the vector was a row vector or column vector by typing

```
[nrow, ncol] = size(v)
```

For a row vector, ncol will be 1; for a column vector, nrow will be 1. The function length discussed above actually calls size and returns either nrow or ncol, whichever isn't 1.

Although one can multiply an entire vector v by 2 by saying v1 = v.*2; it is not always desirable to access an entire vector or matrix at once. To multiply the third element of vector v by 2, you instead type

```
v(3) = v(3) * 2;   % you don't need to type .* since both v(3) and 2 are scalars
```

To add one to elements 4 through 8 of vector v1, type

```
   v1([4:8]) = v1([4:8]) + 1;
```

To set elements 3, 5, and 9 of vector v2 to zero, type

```
   v2([3  5  9]) = 0;
```

To remove elements 3, 5, and 9, type

```
v2([3  5  9]) = [];        % this is probably the first time you've seen this trick!
```

To multiply all the elements of v3 by 5 except the first two, type

```
v3 = v3(3:end) * 5;        % note that here "end" means "end of the vector"
                           % also, you don't need .* since one variable is a scalar
```

You can combine the above ideas with what you know about creating vectors

```
v4(1:2:end) = []; % removes the 1, 3, 5, etc. entries in vector v4
```

Unlike vectors that just have a length, matrices have both a number of rows and columns. Use `size` to determine this:

```
[nrow, ncol] = size(m);
```

Accessing matrix elements is similar, but using row, column format

```
m(4,5) = m(4,5) + 1;       % adds 1 to row 4, column 5 of matrix m
m([1:2], [end-1:end]) = 0;       % sets the 4 elements of m that are in the first two
   rows and last two columns to zero
```

One trick with matrices is to use a colon to represent all the rows or all the columns

```
m(:,2) = [];         % this deletes the second column (and all the rows) of matrix m
m([1  3],: ) = 0     % sets the first and third rows of m (and all the columns) to zero.
```

You can transpose a matrix (i.e. make 3x5 matrix a 5x3 matrix) using the transpose operator '.

```
m'
```

This also works to convert a row vector into a column vector or a column vector into a row vector

```
v'
```

If you attempt to set part of a matrix or vector (e.g. m([1:2],[1:2]) for the upper-left 4 elements) to a different-sized matrix or vector (e.g. m([1:2],[1:2]) = [1 2 3 4 5]) then you will generate an error message

```
In an assignment A = B, the number of columns in B and number of
   elements in B must be the same.
```

Problem 5
Clear your workspace and reload the variables from the Lab1.mat file. Determine the command necessary to replace the third column of matrix m with the vector v5, and display the result.

You can also access vectors and matrices within a loop. For example, enter the following code at the Matlab prompt pressing the return key after each line:

```
>> x = 3 * 0:0.2:5 + 2;
>> for i = 1:length(x)-1
y(i) = x(i+2);
end
>>
```

Note 1: Loops are usually used within m-files. When it is entered at the command line, Matlab will wait to execute it until the `end` command is typed.

Note 2: This could be better done without a loop by typing `y = x(2:end)`

7. Writing scripts and m-files

a) Scripts

Scripts are not programs, but just collections of commands that are run in a batch, just as if they were executed at the command window, but saved as a file with a .m extension. Scripts are evil, but m-files (also known as functions) are good. Here is why: say you saved the following script as MyScript.m:

```
degF = input ('What is the temperature in degrees Fahrenheit?');
degC = 5/9*(degF-32);
disp(sprintf('The temperature is %g degrees Celsius', degC));
```

The commands `disp` and `sprintf` are new. `Disp` simply displays a value to the screen without assigning it to default variable `ans`; type `disp`(4) and compare this to what happens when you just type 4 (and press the return key). `sprintf` creates a string that has embedded variables replace each instance of %g.

Now, if you type `MyScript` at the Matlab command prompt, the script will prompt you to enter a number, and will then display a different number on the screen. If you check your variables kept in memory after this by typing `whos` at the command prompt, you'll discover that both degF and degC are still taking up memory space. Worse, you can't call `MyScript` as you could a function such as y = sin(4) where the input argument 4 is passed to function sin, which then returns a number to y -- i.e. you can't say y = `MyScript`(70). Don't use scripts; m-files (i.e. functions) can do everything scripts can, and more.

b) M-file example 1

Best illustrated with an example, this is an m-file that extracts the last L elements from a vector x

```
function y = GetLast( x, L )
%GetLast returns the last L elements of vector x
% usage:
%  y = GetLast( x, L)
% where:
%  x = input vector
%  L = number of last elements to retrieve
%  y = output vector
N = length(x);
if (L > N)
   error('input vector too short!')
end
y = x( (N-L+1) : N );
```

This file should be saved as GetLast.m and may be invoked from the Matlab command prompt by typing
`test = GetLast(1:2:19, 3);`
then variable `test` will hold the vector [15 17 19]
Now if you check out the variables that are currently defined using `whos` you will not see the variables x, L, y, or N. These were all temporary variables created when the function ran and were deleted when the function finished. Only the value test remains. This both clears up memory and more importantly prevents a function from accidentally overwriting a variable you have set.

c) M-file example 2

This m-file takes a data vector, and checks each value to make sure it is not less than zero. If it is, it sets it to zero. This simulates how a microcontroller, unlike a PC, usually cannot represent numbers less than 0. The m-file returns two vectors, the data output vector that has negative values set to zero and a sample number vector called index to make plotting easier.

```
function [y,index] = MicroSim(x)
% MicroSim replaces negative entries in a vector to zero
% usage:
%  [y,index] = MicroSim(x)
% where
%  x = input vector
%  y = output vector
%  index hold the sample number.  E.g. plot result using plot(index, y)

% setup
[nrows, ncols] = size(x);
y = x;              % create the output vector

% error checking
if (ncols ~= 1 & nrows ~= 1)
   error('input to MicroSim must be a vector!')
end
L = max([nrows ncols]);    % length of input vector; same as length(x)

for n=1:L           % loop over the entire vector
   if y(n) < 0      % if its negative, make it equal to zero
         y(n) = 0;
   end
end

index = 1:L;        % make the index vector
```

This file should be saved as MicroSim.m. If called using the vector `data` = [-6 -1 0 5.2 845.9]; as follows:
`[y, index] = MicroSim(data);`
then `y` will equal [0 0 0 5 846], and `index` will equal [1 2 3 4 5].

Problem 6

A more accurate simulation of a microcontroller's limitations would include the fact that microcontrollers typically can only work with integers (not real numbers) and furthermore that the integers must be between 0 and 255. Modify the above program to do this, e.g. so that the given example above will return a y of [0 0 0 5 255]. To speed development, you can download the original MicroSim.m file from the course website. Test it using the test vector [-6 -1 0 5.2 845.9] to make sure it returns both the proper data and index.

8. Comprehension problems

Problem 7

Use Matlab to compute $\frac{(2+j\,5)(1-j\,5)}{(1+j\,7)(3+j\,2)}$ and express your answer in polar (magnitude and angle in degrees) format. Include your commands to do so.

Problem 8

Create a vector t that contains 100 values linearly varying from 0 to 8π. Let $z=[1-\cos(2.25*t)]*e^{j*t}$. Plot the real and imaginary parts of z as the x and y coordinates, respectively. Use axis equal after plotting to make circles look like circles rather than ellipses. Comment on why the graph looks like hinged lines with slopes that suddenly change (surprising since the functions that compose it all have continuous derivatives), and what you can do to make the plot appear as smooth as it actually should be. (Problem with errors? Re-read the Laboratory Part 3 section, subparts 4 and 5).

Problem 9

Write a function **shorten** that removes every other element from an arbitrary length vector, creating a shorter vector made of only the odd-numbered elements of the original vector, and a function **lengthen** that creates a longer vector by adding an additional element between neighboring elements in the original vector. Each new element should equal the average of its neighboring elements. Test your solution by applying shorten and then lengthen to the vector x = [0 2 3 2 1]. What happens when you execute lengthen(shorten(x))? What happens when you execute shorten(lengthen(x))?
Note 1: although both of these are tough, lengthen in particular is quite challenging.
Note 2: partial credit is awarded if you complete the programs using loops. Full credit is given if you can use the faster (and that uses less code) vector accessing notation. For more information, see the information in tutorial part 6, e.g. looping

```
for i=1:2:length(x)
   x(i)=4;
end
```

to set every other value of a vector to 4, vs. the faster one-line equivalent of

```
x[1:2:length(x)] = 4.
```

4. Matlab Commands Used in Laboratory 1

Workspace commands

clear x y z	clears the definitions of variables x, y, and z
save test x y	saves the variables x and y in a file called test.mat
load test	loads all the variables stored in the test.mat file
whos	determines what variables are defined

Mathematical functions

real(z), imag(z)	returns the real and imaginary parts of z
abs(z)	returns the magnitude of (possibly complex) z
angle(z)	returns the angle of complex z in radians
round(z)	returns z rounded to the nearest integer

Vector functions

x = linspace(0,1,100)	returns a vector x of 100 numbers evenly-spaced from 0 to 1
x = [0:0.01:1]	returns a vector starting at 0 and incrementing in 0.01 steps up to 1
x = zeros(3,5)	creates a matrix of 3 rows and 5 columns filled with zeros
x = ones(3,5)	like the above but filled with ones
[rows,cols] = size(m)	returns the number of rows and columns in vector v
rows=length(v)	if v is a vector, returns its length, i.e. the number of rows if it is a row vector or the number of columns if it is a column vector.
max(v), min(v)	returns the maximum or minimum value in vector v

Matrix element access commands

v(4) = v2(3)	sets the 4th element of v to the value contained in the 3rd element of v2.
v([1 4]) = [5 10]	sets the first element of vector v to 5 and the fourth to 10
v([1:4]) = [10:-1:7];	sets the first four elements of vector v to 10, 9, 8, and 7 respectively
v([5:7]) = [];	makes v smaller by removing the elements at positions 5, 6, and 7
v(end) = 3;	sets the last element of v to 3.
m([1:2],[end-1:end])=0;	sets the four elements of matrix m that are in rows 1 and 2, and the second to the last and final columns, to the value 0.
m(:,3) = []	makes m smaller by removing the 3rd column

Testing

if(x==y)	returns a 1 if scalar x equals scalar y, otherwise returns a 0 If x and y are vectors, will compare respective elements and return a vector of 0's and 1's.
any, all	If working with vectors, use if(any(x==y)) or if(all(x==y).
if (x>=y)	Returns 1 if x is greater or equal to y. Similar for if (x<=y).
if (x~=y)	Returns 1 if x is not equal to y.

Flow control

```
for i=1:10
   v[i] = i/2;
end
```

fills vector v with 10 values from 1/2 to 5.

```
for i=1:2:10
  disp(v[i])
end
```

skips over every other value in vector v

Plotting/display commands

disp(m)	displays vector m on the screen without assigning it to default variable ans
sprintf('Hi %g',n)	if n is 4, returns the string 'Hi 4'. Usually used inside the disp command.
plot(x,y)	plots the x vector against the y vector

plot(x,y,'r-')	makes the plot red with straight lines. Can use g,b,k for green, blue, black. For lines can use – (straight line), -- (dashed line), or . (puts dots at each xy location but does not connect them into lines)
axis equal	makes the xy aspect ratio of the plot equal, so that circles look like circles and not ellipses
title('My title')	Places a title at the top of a plot window
xlabel('x axis')	Places the text 'x axis' below the horizontal axis of the plot
ylabel('y axis')	Places the text 'y axis' to the left of the plot's vertical axis