MATLAB Project 2

Linear Systems & Vectors

1 Introduction

This project will be consist of two parts: the first will give you some experience using MATLAB to solve systems of linear equations, as well as showing you some different ways of finding the solution and creating augmented matrices (when necessary); the second will teach you the basics of working with vectors in MATLAB – ideas that will be generalized and expanded upon once we begin discussing matrices in general (in Chapter 2).

2 Obligatory Tasks

As with your last project, you will need to open your favorite word processor (like Microsoft Word) so that you can complete the Exercises listed in this project. Be sure to include your full name at the top of your document! Also, please note that this project will require you to do some work by hand. I recommend skipping some space in your document so you can handwrite the necessary solutions in later. Since there will be some handwritten components, if you choose to submit your document electronically you will need to print your document, complete the handwritten components, and then scan it back in (so it might be easier to hand in a paper copy!).

If you choose to submit your project electronically, please save your file as:

`Lastname_Firstname_MATLAB_Project2.docx`

(or whatever file extension you chose to use).

3 Exercise Key

The several icons will appear next to each Exercise you are asked to complete. Use the below table to determine what information should be included in your submitted document to fully complete the exercise.

Icon:	Item:	Comments:
	Commands entered in MATLAB & resulting output	You should copy relevant input and output from MATLAB and paste it into your document. You need only include commands that worked.
	Plots & Graphs	Include all graphs generated in an exercise unless the problem specifically tells you which/how many to include.
	Full sentence response	Each exercise contains a question that you should use at least one or two complete sentences to answer. Even if you're stuck, write down any reasoning or ideas you've had.
	Requires work by hand	Do scratch work by hand. Leave space in your document and write your scratch work directly on the assignment to turn in.

4 Systems of Linear Equations

We know that any system of linear equations can be transformed into a matrix equation which makes the system easier to solve. For example, the system

$$\begin{cases} 2x_1 - x_2 + x_3 = 8 \\ x_1 + 2x_2 + 3x_3 = 9 \\ 3x_1 - x_3 = 3 \end{cases}$$

can be written as

$$\begin{pmatrix} 2 & -1 & 1 \\ 1 & 2 & 3 \\ 3 & 0 & -1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 8 \\ 9 \\ 3 \end{pmatrix}.$$

We can also write the corresponding augmented matrix for this system

$$\left(\begin{array}{ccc|c} 2 & -1 & 1 & 8 \\ 1 & 2 & 3 & 9 \\ 3 & 0 & -1 & 3 \end{array}\right),$$

and perform Gaussian elimination to determine the solution.

While the idea of Gaussian elimination is easy to understand, in practice it might not be so easy to actually apply by hand. In the above augmented matrix we have nice integer-valued entries. What would happen if the our matrix looked like

$$\left(\begin{array}{ccc|c} 2.3928492398284 & -1.293821039 & 1.293848293848 & 8 \\ 1.384829482239 & 2.222134239 & 3.01993098210 & 9 \\ 3.294823493029 & -.11111219 & -1.02018302876 & 3 \end{array}\right)$$

instead? Gaussian elimination probably wouldn't be so easy anymore! Luckily for us, MATLAB can still perform Gaussian elimination on this matrix without any trouble at all.

Gaussian elimination in MATLAB is performed using the `rref( )` command (standing for "reduced row echelon form" as we are used to). So, to solve our original system, we would execute:

```
>> rref([2 -1 1 8;1 2 3 9;3 0 -1 3])

ans =
     1     0     0     2
     0     1     0    -1
     0     0     1     3
```

So we see our solution is given by $(2, -1, 3)$.

Alternatively, rather than entering the matrix directly into the `rref( )` function, we can define our augmented matrix, then pass that matrix along to the `rref( )` function:

```
>>  aug = [2 -1 1 8;1 2 3 9;3 0 -1 3]

aug =
     2    -1     1     8
     1     2     3     9
     3     0    -1     3

>> rref(aug)

ans =
     1     0     0     2
     0     1     0    -1
     0     0     1     3
```

This helps if we ever want to use the matrix `aug` again for other computations.

Even better yet, we could do the following:

```
>> A = [2 -1 1;1 2 3;3 0 -1]

A =
     2    -1     1
     1     2     3
     3     0    -1

>> b = [8;9;3]

b =
     8
     9
     3

>> aug = [A b]

aug =
     2    -1     1     8
     1     2     3     9
     3     0    -1     3
```

This way we can reuse the our matrix or vector (much more on these later) whenever we need without having to retype all of the entries.

Yet another shortcut is to bypass specifically defining the `aug` matrix at all. We could just as easily have executed the command

```
>> rref([A b])

ans =
     1     0     0     2
     0     1     0    -1
     0     0     1     3
```

Exercise 1. Consider the following homogeneous system of equations:

$$\begin{cases} x_1 - 3x_2 + 2x_3 = 0 \\ -2x_1 + 6x_2 - 4x_3 = 0 \\ 4x_1 - 12x_2 + 8x_3 = 0 \end{cases}$$

Use the `rref( )` command to determine the solution set of this system. Write your solution set in parametric form.

MATLAB actually has another method of solving a linear system of equations without relying on Gaussian elimination. Going back to our matrix description of the system,

$$\begin{pmatrix} 2 & -1 & 1 \\ 1 & 2 & 3 \\ 3 & 0 & -1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 8 \\ 9 \\ 3 \end{pmatrix},$$

we see the system is of the form $A\mathbf{x} = \mathbf{b}$. MATLAB can now solve this system using something that might look like division (be careful though, as we will see you cannot "divide by a matrix"!)

Since we have already entered A and $\mathbf{b}$ into MATLAB above so we don't need to do it again here, we may just evaluate the following:

```
>> x = A\b     %Note, we have used a backslash here "\" and NOT a forward slash "/"

x =
    2.0000
   -1.0000
    3.0000
```

Notice here that, while this solution and the solution we obtain using `rref( )` are equal they look a bit different (namely, the decimals on the latter solution). This is simply a consequence of MATLAB trying to "divide" by the matrix A where it must interpret the entries of A as floating-point values (numbers with decimals) rather than integers.

Exercise 2. (a) Consider the system of equations:

$$\begin{cases} 2x_1 + x_2 + 5x_3 = -1 \\ x_1 + 6x_3 = 2 \\ -6x_1 + 2x_2 + 4x_3 = 3 \end{cases}$$

On paper, convert this system of equations into a matrix equation of the form $A\mathbf{x} = \mathbf{b}$.

(b) Enter the matrix A and the vector $\mathbf{b}$ into MATLAB and execute the command

```
>> x = A\b
```

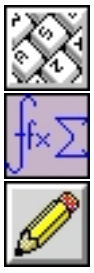
(c) In part (b) we found the vector $\mathbf{x}$ such that $A\mathbf{x}=\mathbf{b}$, so we would expect to obtain the vector $\mathbf{b}$ in MATLAB if we executed the command `A*x`. In other words, this means that, in MATLAB, `A*x - b` should equal the zero vector $\mathbf{0}$. Execute the following:

```
>> A*x - b
```

What do you get?

Note: The discrepancy you see in part (c) of Exercise 2 is simply due to rounding error. You'll notice that the error is a vector multiplied by a very small number, one on the order of 10^{-15}. The reason for this error is the same as the reason for why MATLAB introduced decimals in the previous solution – it all lies in the way that MATLAB store numbers. In this calculation MATLAB represents numbers in floating-point form to 10^{-14} accuracy. Thus, when you see 10^{-14} printed in calculations, it is equivalent to 0!

There is a drawback to solving systems of equations using the command `x=A\b` as seen in the following exercise:



Exercise 3. Consider the system of equations:

$$\begin{cases} -10x_1 + 4x_2 = 0 \\ 15x_1 - 6x_2 = 0 \end{cases}$$

As you did in Exercise 2, enter the corresponding matrix A and vector **b** into MATLAB and then type

```
>> x = A\b
```

Note the strange output (include it in your document!). Now, solve this system by hand. How many free variables are there? Based on your answer, can you explain why you get an error message when trying to use `x=A\b`?

5 Vectors

Seeing as how we already know how to input matrices into MATLAB, vectors should be no sweat – remember, after all, a vector is simply a $n \times 1$ matrix!

Let's begin by entering a simple column vector:

```
>> v=[1;3;5;7]

v =
     1
     3
     5
     7
```

Recall that the semicolon designates the end of a row. Since each row in a column vector contains only one element (there is just one column) we must enter a semicolon after every entry.

If we wanted to create a larger and more interesting vector, entering the values one-by-one might be tedious. If the entries correspond to a distinct pattern (or sequence) MATLAB can do the hard work for you. For instance, if we want to create a vector containing the elements 1 through 100 we could execute the following:

```
>> v =[1:1:100]'

v =
     1
     2
     .
     .
     .
   100
```

The first value tells MATLAB the starting value you would like in your vector; the second instructs MATLAB to increase that value by 1 until you arrive at the last value. You might notice that we needed to include an apostrophe `'` after the square brackets – this is called the *transpose operator* in MATLAB. Transposition is something we will discuss much further in Chapter 2, but for us now, you only need to know that it turns a $1 \times n$ matrix into a $n \times 1$ matrix (or vector).

Rather than having a vector whose entries are increasing, we could just as easily create one with decreasing values:

```
>> u = [5:-1:0]'

u =
     5
     4
     3
     2
     1
     0
```

Exercise 4. Using the method above for constructing vectors whose entries follow a pattern, create a vector whose entries decrease from 15 down to -6 by 3.

There are a two vectors that are used so often that MATLAB has a built in function for creating them quickly, namely, the zero vector and a vector with all entries 1. To create the zero vector in $\mathbb{R}^n$ you simply need to execute the command `zeros(n,1)`, and to create a vector with all 1's, `ones(n,1)`:

```
>> zeros(4,1)

ans =
     0
     0
     0
     0

>> ones(3,1)

ans =
     1
     1
     1
```

These functions are much more powerful than we are giving them credit for here – they actually create special matrices (the zero-matrix and the *identity* matrix). We will see these matrices more in Chapter 2, but for now, if you want to see them simply change the 1 in the second entry of each of the above functions (the second entry represents the number of columns – we wanted column vectors, so we set that entry to 1!).

Addition, subtraction and scaling of vectors works just how you would expect:

```
>> u = [1;3;5;7]

u =
     1
     3
     5
     7

>> v = [0;1;2;4]

v =
     0
     1
     2
     4
```

```
>> u+v

ans =
     1
     4
     7
    11

>> u-v

ans =
     1
     2
     3
     3

>> 3*u

ans =
     3
     9
    15
    21
```

We can also extract specific data from each of our vectors in a very natural way. If we want the first entry in our vector `u` we simply execute the command `u(1)`. If we want the third entry in `v`, then we execute `v(3)`.

Using this, if we wanted to create a vector in $\mathbb{R}^3$ from the vector `u` in $\mathbb{R}^4$ by using everything except the third entry, we can execute

```
>> u2 = [u(1); u(2); u(4)]

u2 =
     1
     3
     7
```

We could also have executed the command `u2 = u([1 2 4])` to obtain the same vector. This command tells MATLAB to take only the parts of the vector `u` in position (or index) 1, 2 and 4.

This method works great for small vectors, but might not be so helpful for large ones. Luckily for us, we can use the colon notation we saw above:

```
>> u2 = u([1 3:4])

ans =
     1
     5
     7
```

This command tells MATLAB to take the first entry of `u` as well as all of the entries from 3 to 4. Entering `3:4` here is the same as entering `3:1:4`, where we are telling MATLAB to begin at 3 and end at 4 incrementing by 1 (if we want to increase by only 1, the 1 can be omitted!).

Thus, if we wanted to create a new vector consisting of the third, second and fourth entries of `u` (in that order!) we could use any of the following commands: `[u(3);u(2);u(4)]`, `u([3 2 4])`, or `u([3 2:2:4])` (where the last command tells MATLAB to include entry 3, and then all entries starting at 2 and ending at 4 when we increment by 2).

The method using colons might seem a bit cumbersome above, but it should come in handy in the following exercise!

Exercise 5. Create a vector **r** whose entries begin at 3, end at 99, and are incremented by 3 (you do not need to display/copy/paste this vector – use a semicolon (;) to surpress output in MATLAB). Then create a new vector **r1** that contains: the first 3 entries of **r**, the 15th entry, the 18th entry, and the last 4 entries (they would be entries 30 through 33). Note: You must use the colon notation to receive credit here – do not simply enter all of the values/locations by hand!

As a final note, just as we were able to easily create smaller vectors from larger ones, we can easily extend the length of already defined vectors:

```
>> s = [1;2;3]

s =
     1
     2
     3

>> t = [s;4;5;6]

t =
     1
     2
     3
     4
     5
     6
```