2.4 Array Operations in Matlab

When we multiply a scalar with a vector or matrix, we simply multiply the scalar times each entry of the vector or matrix. Addition and subtraction behave in the same manner, that is, if we add or subtract two vectors or matrices, we obtain the answer by adding or subtracting the correponding entries.

Matrix multiplication, as we've seen, is quite different. When we multiply two matrices, for example, we don't simply multiply the corresponding entries. For example, enter the following matrices A and B.

```
>> A=[1 2;3 4], B=[2 5;-1 3]
A =
     1     2
     3     4
B =
     2     5
    -1     3
```

Compute the product of A and B.

```
>> A*B
ans =
     0    11
     2    27
```

Note that

$$\begin{bmatrix}1 & 2\\ 3 & 4\end{bmatrix}\begin{bmatrix}2 & 5\\ -1 & 3\end{bmatrix}=\begin{bmatrix}0 & 11\\ 2 & 27\end{bmatrix}.$$

Matrix multiplication is not performed by multiplying the corresponding entries of each matrix.

Array Multiplication. However, there are occasions where we need to "multiply" two matrices by finding the product of the corresponding entries, so that the resulting "product" looks as follows.

[1] Copyrighted material. See: http://msenux.redwoods.edu/Math4Textbook/

Warning 1. *This is not matrix multiplication. Rather, it is a special type of multiplication called* **array multiplication**.

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \begin{bmatrix} 2 & 5 \\ -1 & 3 \end{bmatrix} = \begin{bmatrix} 2 & 10 \\ -3 & 12 \end{bmatrix}$$

Note that the array multiplication is performed by simply multiplying the corresponding entries of each matrix.

Matlab provides an operator that allows us to perform *array multiplication*, namely the `.*` operator, sometimes pronounced “dot-times.” In the case of our previously entered matrices A and B, note how array multiplication performs.

```
>> A,B
A =
     1     2
     3     4
B =
     2     5
    -1     3
>> A.*B
ans =
     2    10
    -3    12
```

Thus, when you use `.*`, Matlab’s array multiplication operator, the product of two matrices or vectors is found by multiplying the corresponding entries in each matrix or vector.

Here is an example of using array multiplication with row vectors.

```
>> v=[1 2 3], w=[4 5 6]
v =
     1     2     3
w =
     4     5     6
>> v.*w
ans =
     4    10    18
```

Again, note that the array product of the row vectors **v** and **w** was found by mulitplying the corresponding entries of each vector.

Array multiplication works equally well for column vectors.

```
>> v=v.',w=w.'
v =
     1
     2
     3
w =
     4
     5
     6
>> v.*w
ans =
     4
    10
    18
```

Array Division. We will also need the ability to "divide" matrices or vectors by finding the quotient of the corresponding entries. The Matlab operator for *array division* is `./`, pronounced "dot-divided by." It works in exactly the same way as does array multiplication.

Warning 2. *This is not matrix division. Rather, it is a special type of division called* **array division**.

$$\begin{bmatrix} 1 & 3 \\ 5 & 7 \end{bmatrix} \Big/ \begin{bmatrix} 3 & 5 \\ 7 & 9 \end{bmatrix} = \begin{bmatrix} 1/3 & 3/5 \\ 5/7 & 7/9 \end{bmatrix}$$

Note that array division is performed by dividing each entry of the matrix on the left by the corresponding entry of the matrix on the right.

Matlab provides an operator that allows us to perform *array right division*, namely the `./` operator, sometimes pronounced "dot-divided by." It helps to first change the display to rational format.

```
>> format rat
```

Enter the matrices A and B that are used in **Warning 2**.

```
>> A=[1 3;5 7], B=[3 5;7 9]
A =
       1              3
       5              7
B =
       3              5
       7              9
```

Now, use array right division to obtain the result shown in **Warning 2**.

```
>> A./B
ans =
       1/3            3/5
       5/7            7/9
```

Matlab also has an array left division operator that is sometimes useful.

```
>> A.\B
ans =
       3              5/3
       7/5            9/7
```

Note that each entry of the result is found by dividing each entry of the matrix on the right by the corresponding entry of the matrix on the left. Hence, the term “left division.”

It’s often useful to divide a scalar by a matrix or vector. With array right division, the scalar is divided by each entry of the vector or matrix.

```
>> v=1:5
v =
       1              2              3              4
  5
>> 1./v
ans =
       1              1/2            1/3            1/4
  1/5
```

With array left division, each entry of the vector or matrix is divided by the scalar.

```
>> w=[4;5;6]
w =
       4
       5
       6
>> 7.\w
ans =
       4/7
       5/7
       6/7
```

Array Exponentiation. We will also need the ability to raise each entry of a vector or matrix to a power. The Matlab operator for *array exponentiation* is `.^`, pronounced "dot raised to." It works in exactly the same manner as does array multiplication and division.

Warning 3. *This is not the usual way to raise a matrix to a power. Rather, it is a special type of exponentiation called* **array exponentiation**.

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}^2 = \begin{bmatrix} 1 & 4 \\ 9 & 16 \end{bmatrix}$$

Note that array exponentiation is performed by raising each entry of the matrix to the power of 2.

We can return to default formatting with the following command.

```
>> format
```

Now, suppose that we wish to raise each element of a vector to the third power. We use `.^` for this task.

```
>> v=3:7
v =
       3              4              5              6
  7
>> v.^3
ans =
      27             64            125            216
343
```

Note that raising the vector **v** to the third power is not possible because the dimensions will not allow $\mathbf{v}^2 = \mathbf{vvv}$.

```
>> v^3
??? Error using ==> mpower
Matrix must be square.
```

You can square each element of a matrix with the array exponentiation operator.

```
>> A=[1 2;3 4]
A =
       1              2
       3              4
>> A.^2
ans =
       1              4
       9             16
```

Note that array exponentiation completely differs from regular exponentiation.

```
>> A^2
ans =
       7             10
      15             22
```

Matlab Functions are Array Smart

Most of Matlab's elementary functions are what we like to call "array smart," that is, when wyou feed a Matlab function a vector or matrix, then that function is applied to each element of the vector or matrix.

Consider, for example, the behavior of the sine function. You can take the sine of a single number.

```
>> sin(pi/2)
ans =
       1
```

However, a powerful feature is the fact that matlab's sine function can be applied to a vector or matrix.

```
>> x=0:pi/2:2*pi
x =
         0    1.5708    3.1416    4.7124    6.2832
>> sin(x)
ans =
         0    1.0000    0.0000   -1.0000   -0.0000
```

Note that Matlab took the sine of each element of the vector (with a little round off error). You will see similar behavior if you take the sine of a matrix.

```
>> A=[0 pi/2; pi 3*pi/2]
A =
         0    1.5708
    3.1416    4.7124
>> sin(A)
ans =
         0    1.0000
    0.0000   -1.0000
```

Matlab took the sine of each entry of matrix A (to a little roundoff error).

You can take the natural logarithm[2] of a number with Matlab's **log** function.

```
>> log(1)
ans =
     0
```

Matlab's **log** function is array smart.

```
>> x=(1:5).'
x =
     1
     2
     3
     4
     5
>> log(x)
ans =
         0
    0.6931
    1.0986
    1.3863
    1.6094
```

Again, Matlab took the natural logarithm of each element of the vector. This is typical of the way most Matlab functions work.

You can access a list of Matlab's elementary functions with the command **help elfun**.

Basic Plotting

Because Matlab's elementary array functions are "array smart," it is a simple matter to obtain a plot of most elementary functions using Matlab's **plot** command.

In its simplest form, Matlab's **plot** command takes two vectors **x** and **y** which contain the x- and y-values of a collection of points (x, y) to be plotted. In its

[2] Students of mathematics are confused when they find that the Matlab command **log** is used to compute the natural logarithm of a number. Indeed, mathematicians usualy write $\ln x$ to denote the natural logarithm and log to denote the base ten logarithm. In Matlab **log** is use to compute the natural logarithm, while the command **log10** is used to compute the base ten logarithm.

default **plot(x,y)** form, Matlab's **plot** command plots the points in the order received and connects consecutive points with line segments.

The following commands store the numbers 0, $\pi/2$, π, $3\pi/2$ and 2π in the vector **x**, then evaluate the sine at each entry of the vector **x**, storing the results in the vector **y**.

```
>> x=0:pi/2:2*pi
x =
         0    1.5708    3.1416    4.7124    6.2832
>> y=sin(x)
y =
         0    1.0000    0.0000   -1.0000   -0.0000
```

The command **plot(x,y)** is used to produce the graph in **Figure 2.1**(a).

```
>> plot(x,y)
```

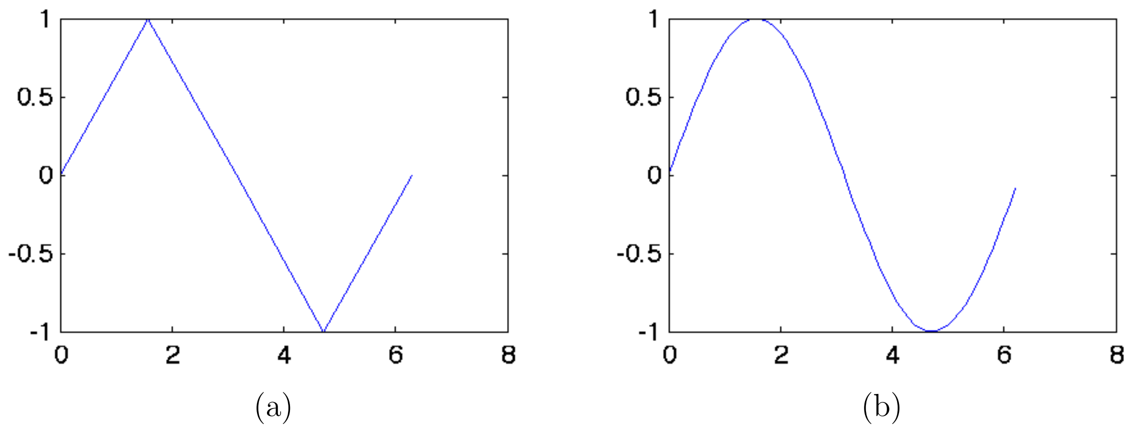


Figure 2.1. A plots of $y = \sin x$.

In **Figure 2.1**(a), there are not enough plotted points to give a true picture of the graph of $y = \sin x$. In a second attempt, we use Matlab's **start:increment:finish** construct to examine the value of the sine at an increased number of values of x. We use semicolons to suppress the output to the display.

```
>> x=0:0.1:2*pi;
>> y=sin(x);
>> plot(x,y)
```

The result of these commands is the plot of the graph of $y = \sin x$ shown in **Figure 2.1**(b).

Matlab's **linspace** command is useful for creating a range of values for plotting. The syntax **linspace(a,b,n)** generates n points between the values of a and b, including the values at a and b. The following sequence of commands were used to create the graph of $y = \sin x$ in **Figure 2.2**.

```
>> x=linspace(0,2*pi,200);
>> y=sin(x);
>> plot(x,y)
>> axis tight
>> xlabel('x-axis')
>> ylabel('y-axis')
>> title('The graph of y=sin(x).')
```

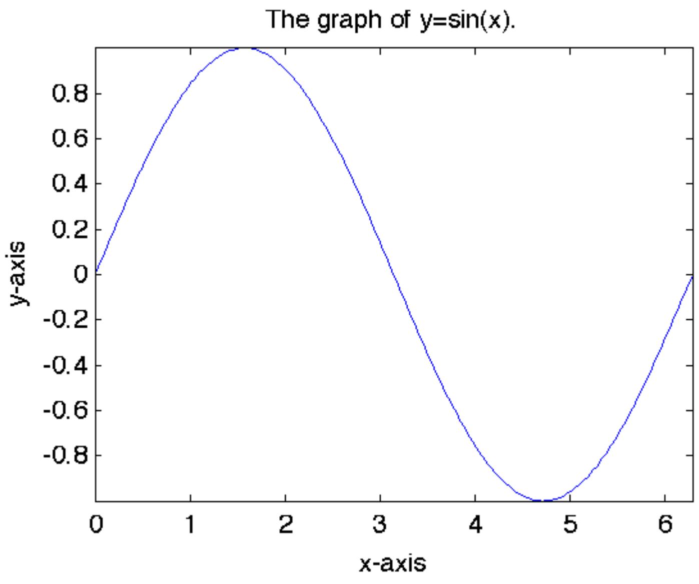


Figure 2.2. The graph of $y = \sin x$ on the interval $[0, 2\pi]$.

Some comments are in order.

1. The command **linspace(0,2*pi,200)** generates 200 equally spaced (linearly spaced) points between 0 and 2π, including the values of 0 and 2π.
2. The **axis tight** command "tightens" the axes window to the plot.
3. Matlab's **xlabel** accepts string input (delimited with single apostrophes) and uses the text to annotate the horizontal axis of the plot.

4. Similarly, the **ylabel** command is used to annotate the vertical axis.
5. Finally, Matlab's **title** command accepts string input and uses the text to annotate the plot with a title.

Script Files

One soon tires of typing commands interactively at the prompt in Matlab's command window. Commands that can be typed sequentially at the prompt in the Matlab command window can be placed into a file and executed en-mass.

As an example, type the following command at the Matlab prompt to open Matlab's editor.

```
>> edit
```

When the editor opens, type in the lines we used above to plot the graph of the sine function on the interval $[0, 2\pi]$.

```
x=linspace(0,2*pi,200);
y=sin(x);
plot(x,y)
axis tight
xlabel('x-axis')
ylabel('y-axis')
title('The graph of y=sin(x).')
```

Save the file in a directory of your choice as **sineplot.m**. You must always save a script file with the extension **.m** attached, as in **sineplot.m**. You are free to choose any filename you wish, but avoid naming the file with a name reserved by Matlab. For example, it would be unwise to name a script file **plot.m**, because you would no longer have access to Matlab's **plot** command.

To execute the script **sineplot.m**, you have two choices.

1. While in Matlab's Editor, press the F5 key to execute the script. *Note: If you save the file in a directory other than Matlab's current directory (the one shown in the navigator window in the toolbar), pressing F5 will bring up a dialog with choices: (1) Change the Matlab current directory, (2) Add directory to the top of Matlab path, and (3) Add directory to the bottom of*

the Matlab path. Select change Matlab current directory and click the OK button.

2. You can also return to the command window and enter the filename of the script at the Matlab prompt, as in **sineplot**. In this case, you need not include the extension.

The script should execute and display the graph of $y = \sin x$, as shown in **Figure 2.2**. If your script fails to execute, read further to determine the source of the problem.

Trouble Shooting. If your script won't execute, here are some things to check.

1. If a another script of the same name is being executed instead of yours, a helpful Matlab command is the command **which sineplot.m**. This will respond with the path to the script **sineplot.m** which will be executed when the user types **sineplot** at the command prompt. If the path is not as expected, you then know that Matlab is calling a different **sineplot.m**.
2. Another useful command is **type sineplot**. This will produce a listing of the file **sineplot.m** in the command window. You can then read the script and see if it is the one you expect to execute.

Matlab's Path. Matlab has a search path which contains a list of directories that it searches when you type a filename or command at the Matlab prompt. You can view the path by typing the following at the command prompt.

```
>> path
```

This will cause a long list of directories to be displayed on the screen. In searching for commands and files, Matlab obeys the following rules, in the following order.

1. Matlab searches the current directory first. You can determine the current directory by typing the command **pwd** (present working directory — a UNIX command that Matlab understands), or by viewing the navigation window on the toolbar atop of the command window.

2. After searching the current directory, Matlab will search directories according to the order shown in the output of the **path** command.

Changing the Current Directory. We've said that Matlab searches the current directory first. If you save your script in a location other than Matlab's current directory, Matlab will search for your script by looking in directories in the order dictated by Matlab's path. Thus, the usual cure for a script that won't run is to change Matlab's current directory to the directory in which your script resides. This can be accomplished in one of two ways.

1. In the toolbar below the menus, there is a navigation window that indicates the path to the current directory. To the right of this navigation window is a button with three dots. If you click this, you can browse the directory hierarchy of your machine and select the directory containing your script file. The navigation window will reflect your change of directories, indicating the new current directory. If this directory now matches where you saved your script, typing the name of your script at the prompt in the command window should now execute your script.
2. If you are more command-line oriented, then you can use the UNIX command `cd` ("change directory") to change the current directory to the directory containing your script file. On a Mac, if I know I saved the file in `~/tmp`, then this command will change Matlab's current directory to that directory.

   ```
   >> cd ~/tmp
   ```

   You can check the result of this command with the UNIX command `pwd` ("present working directory").

   ```
   >> pwd
   ans =
   /Users/darnold/tmp
   ```

   On a PC, I might save my script in the file `C:\homework`. Executing the following command will set Matlab's current directory to match the directory containing your script.

   ```
   >> cd C:\homework
   ```

If your current directory matches the directory containing your script, you should be able to execute your script by typing **sineplot** at the prompt in the command window.

Adding Directories or Folders to Matlab's Path. If you have a collection of important files that you use quite often, you won't want to be constantly changing the current directory to use them. In this case, you can add the directory in which these files reside to Matlab's path. Type the following command at the Matlab prompt.

```
>> pathtool
```

Click the button **Add Folder ...** then browse your directory hierarchy and select the folder or directory that you want to add to Matlab's path. Once a folder is selected, there are buttons to move it up or down in the path hierarchy (remember, Matlab searches for files in the order given in the path, top to bottom). You can **Close** the path tool, which means that your adjustment to Matlab's path is only temporary. Next time you start Matlab, changes made to the path will no longer exist. Alternatively, you can make changes to the path permanent (you can come back and make changes later) by using the **Save** button in the path tool. After saving, use the **Close** button to close the pathtool.

The pathtool should be used sparingly. If you are working on an assigment, it is better to have a folder dedicated to that assignment, then change the current directory to that folder and begin working. However, if you have a number of useful utility files to which you would like to maintain access at all times, then that is a folder of files that warrants being added to Matlab's path.

If you are working at school, there is a file named **pathdef.m** in your home directory (H:\ ) that is executed at startup to initialize Matlab's path. If you wish to make permanent changes to the path using the path tool, we recommend that you first change the current directory to your root home directory with this command.

```
>> cd ~/
```

Now you can execute the command **pathtool** and make additions, deletions, or edits to your path. Once you have saved your changes and closed the path tool, change the current directory back to where your were and continue working.

If you are working at home, the file **pathdef.m** is contained in Matlab's hierarchy, so it doesn't matter where you are when you open the **pathtool**.

2.4 Exercises

1. Enter **H=hilb(3)** and change the display output to rational format with the command **format rat**. Use the appropriate command to square each entry of matrix H.

2. Enter **P=pascal(3)** and change the display output to rational format with the command **format rat**. Use the appropriate command to square each entry of matrix P.

3. Create a diagonal matrix with the commands **v=1:3** and **D=diag(v)**. Take the exponetial of each element of the matrix D and explain the result.

4. Create the vector **v=1:5**, then take the exponential of each entry of the vector **v**. Create a diagnonal matrix with the resulting vector. How does this differ from the answer found in **Exercise 3**?

5. The sum of the squares of the integers from 1 to n is given by the formula

$$\frac{n(n+1)(2n+1)}{6}.$$

a) Use the formula to determine the sum of the squares of the integers from 1 to 20, inclusive.

b) The following **for** loop will sum the the squares of the integers from 1 to 20, inclusive.

```
s=1;
for k=2:20
    s=s+k^2;
end
s
```

Verify that this **for** loop produces the same result as in part (a).

c) The following code sums the squares of the integers from 1 to 20, inclusive.

```
s = sum((1:20).^2)
```

Explain why.

6. Use the formula of **Exercise 1** to find the sum of the squares of the integers from 1 to 1000, inclusive.

a) Write a **for** loop to calculate the sum of the squares of the integers from 1 to 1000, inclusive.

b) Use arrays and Matlab's **sum** command to calculate the sum of the squares of the integers from 1 to 1000.

7. The sum of the cubes of the integers from 1 to n can be computed with the formula

$$\left[\frac{n(n+1)}{2}\right]^2.$$

a) Use the formula to determine the

sum of the cubes of the integers from 1 to 1000, inclusive.

b) Write a **for** loop to calculate the sum of the cubes of the integers from 1 to 1000, inclusive.

c) Use arrays and Matlab's **sum** command to calculate the sum of the cubes of the integers from 1 to 1000, inclusive.

8. Save the following in a scriptfile named **quicker.m**.

```
tic
s=1;
for k=2:100000
    s=s+k^3;
end
s
toc

tic
s=sum((1:100000).^3)
toc
```

The Matlab pair **tic** and **toc** record the time for the commands they enclose to execute. Run the script several times by typing **quicker** at the Matlab prompt. Is the **for** loop slower or faster than the arrary technique for computing the sum of the cubes of the integers between 1 and 100 000, inclusive?

9. To learn why it is important to initialize vectors, execute the following code.

```
clc
clear all
N=1000000
tic
a=zeros(N,1);
for k=1:N
    a(k)=1/k;
end
toc
```

Now try the same thing without initializing the vector **a**.

```
clc
clear all
N=1000000
tic
for k=1:N
    a(k)=1/k;
end
toc
```

Results will vary per machine. This causes my machine to hang and I have to break the program by typing **Ctrl+C** in the Matlab command window. Trying adding or deleting a zero from $N = 1000000$ to see how your machine reacts. This is an important lesson on the importance of initializing memory.

In **Exercises 10-19**, perform each of the following tasks for the given sequence.

i. Initialize a column vector of zeros with **a=zeros(10,1)**. Write a **for** loop to populate the vector with the first ten terms of the given sequence. *Note: You might find the* *display output* **format rat** *helpful.*

ii. Initialize a column vector **n** with **n=(1:10).'**. Use array operations to generate the first 10 terms of the sequence and store the results in the vector **a** (without the use of a for loop — array ops only).

iii. Plot the resulting vector **a** versus its index with **stem(a)**.

10. $a_n = (-3)^n$

11. $a_n = 2^n$

12. $a_n = \frac{1}{n}$

13. $a_n = \frac{1}{n^2}$

14. $a_n = \frac{(-1)^n}{n}$

15. $a_n = \frac{1}{3^n}$

16. $a_n = \frac{n}{n+1}$

17. $a_n = \frac{1-n}{n+2}$

18. $a_n = \sin \frac{n\pi}{5}$

19. $a_n = \cos \frac{2n\pi}{5}$

---

20. The *Fibonacci Sequence*,

$$1,\ 1,\ 2,\ 3,\ 5,\ 8,\ \ldots$$

is defined recursively by first setting $a_1 = 1$, $a_2 = 1$, and thereafter,

$$a_n = a_{n-1} + a_{n-2}.$$

Initialize a column vector **a** of length 100 with zeros, then write a **for** loop to populate the vector **a** with the first 100 terms of the Fibonacci Sequence. Use Matlab indexing to determine the 80th term of the sequence.

21. We define a sequence by first setting $a_1 = 1$. Thereafter,

$$a_n = \sqrt{2 + a_{n-1}}.$$

Initialize a column vector **a** of length 30 with zeros, then write a **for** loop to populate the vector **a** with the first 30 terms of the sequence. This sequence appears to converge to what number?

22. A sequence begins with the terms

$$\sqrt{2},\quad \sqrt{2\sqrt{2}},\quad \sqrt{2\sqrt{2\sqrt{2}}},\quad \ldots.$$

Write a **for** loop to generate the first 30 terms of this sequence. This sequence appears to converge to what number?

In **Exercises 22-30**, perform each of the following tasks.

i. Write a **for** loop to sum the given series.

ii. Use Matlab's **sum** command and array operations to perform the same task.

23. $\displaystyle\sum_{n=1}^{20} \frac{1}{n}$

24. $\displaystyle\sum_{n=1}^{20} \frac{1}{n^2}$

25. $\displaystyle\sum_{n=1}^{20} \frac{1}{2^n}$

26. $\displaystyle\sum_{n=1}^{20} \frac{n+1}{n}$

27. $\displaystyle\sum_{n=1}^{20} \frac{n}{n+1}$

28. $\displaystyle\sum_{n=1}^{20} 3^{-n}$

29. $\displaystyle\sum_{n=1}^{20} \frac{1}{n!}$

30. $\displaystyle\sum_{n=1}^{20} \frac{2^n}{n!}$

In **Exercises 30-38**, perform each of the following tasks.

i. In each case, create a script file to draw the graph of the given function. Include a printout of the resulting plot and the script file that produced it with your homework. *Hint: Type* **help elfun** *for help on using the given function in Matlab.*
ii. Use **x=linspace(a,b,n)** to produce enough domain values to produce a "smooth curve." Use the default form **plot(x,y)** to produce a solid blue curve.
iii. Label the horizontal and vertical axes with **xlabel** and **ylabel**.
iv. Use **title** to provide a table containing the equation of the given function and the requested domain.

31. Sketch $y = \cos x$ on the interval $[-2\pi, 2\pi]$.

32. Sketch $y = \sin^{-1} x$ on the interval $[-1, 2]$.

33. Sketch $y = |x|$ on the interval $[-2, 2]$.

34. Sketch $y = \ln x$ on the interval $[0.1, 10]$.

35. Sketch $y = e^x$ on the interval $[-2, 2]$.

36. Sketch $y = \cosh x$ on the interval $[-3, 3]$.

37. Sketch $y = \sinh^{-1} x$ on the interval $[-10, 10]$.

38. Sketch $y = \tan^{-1} x$ on the interval $[-10, 10]$.

2.4 Answers

1. Create the Hilbert matrix.

```
>> H=hilb(3)
H =
   1          1/2        1/3
   1/2        1/3        1/4
   1/3        1/4        1/5
```

Square each entry as follows.

```
>> H.^2
ans =
   1          1/4        1/9
   1/4        1/9        1/16
   1/9        1/16       1/25
```

Return to default display format.

```
>> format
```

3. Enter the vector **v**.

```
>> v=1:3
v =
     1     2     3
```

Create the diagonal matrix D.

```
>> D=diag(v)
D =
     1     0     0
     0     2     0
     0     0     3
```

Take the exponential of each entry of the matrix D.

```
>> E=exp(D)
E =
    2.7183    1.0000    1.0000
    1.0000    7.3891    1.0000
    1.0000    1.0000   20.0855
```

Off the diagonal, $e^0 = 1$, so each entry is a 1. On the diagonal $e^1 \approx 2.7183$, $e^2 \approx 7.3891$, and $e^3 \approx 20.0855$.

5. Set $n = 20$.

a) Then:

```
>> n=20;
>> n*(n+1)*(2*n+1)/6
ans =
        2870
```

b) Verify with **for** loop.

```
>> s=1;
>> for k=2:20
s=s+k^2;
end
>> s
s =
        2870
```

c) Array ops produce the same result.

```
>> s=sum((1:20).^2)
s =
        2870
```

The **(1:20)** produces a row vector with entries from 1 to 20. The array exponentiation **.^2** squares each entry of the vector. The **sum** command adds the entries of the resulting vector (the squares from 1 to 20).

7. Set $n = 1000$.

a) Then:

```
>> format long g
>> n=1000;
>> (n*(n+1)/2)^2
ans =
      250500250000
```

b) Sum with a **for** loop.

```
>> s=0;
>> for k=1:1000
s=s+k^3;
end
>> s
s =
    250500250000
```

c) Same result with array ops:

```
>> s=sum((1:1000).^3)
s =
    250500250000
```

11. A **for** loop.

```
>> a=zeros(10,1);
>> for k=1:10,
a(k)=2^k;
end
>> a
a =
           2
           4
           8
          16
          32
          64
         128
         256
         512
        1024
```

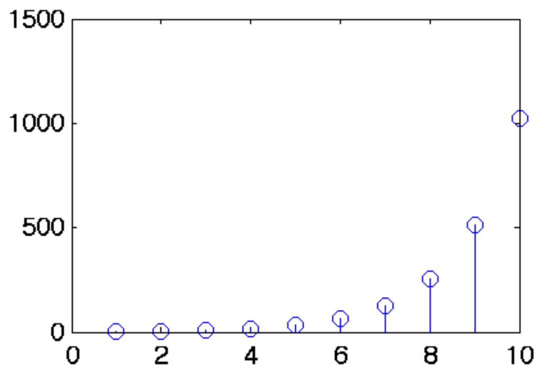
Same result with array ops.

```
>> a=2.^n
a =
           2
           4
           8
          16
          32
          64
         128
         256
         512
        1024
```

A stem plot.

```
>> stem(a)
```

The resulting stem plot.



13. A **for** loop.

```
>> a=zeros(10,1);
>> for k=1:10,
a(k)=1/k^2;
end
>> a
a =
       1
       1/4
       1/9
       1/16
       1/25
       1/36
       1/49
       1/64
       1/81
       1/100
```

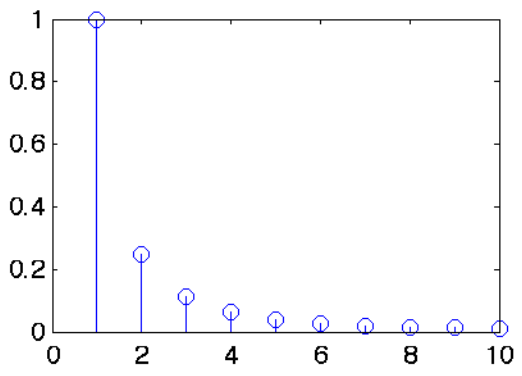
Same result with array ops.

```
>> n=(1:10).';
>> a=1./(n.^2)
a =
       1
       1/4
       1/9
       1/16
       1/25
       1/36
       1/49
       1/64
       1/81
       1/100
```

A stem plot.

```
>> stem(a)
```

The resulting stem plot.



15. A **for** loop.

```
>> a=zeros(10,1);
>> for k=1:10,
a(k)=1/(3^k);
end
>> a
a =
       1/3
       1/9
       1/27
       1/81
       1/243
       1/729
       1/2187
       1/6561
       1/19683
       1/59049
```

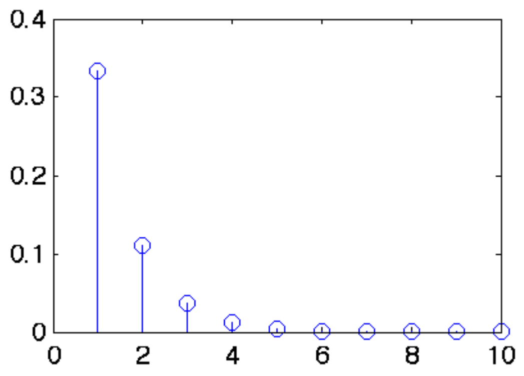
Same result with array ops.

```
>> a=1./(3.^n)
a =
       1/3
       1/9
       1/27
       1/81
       1/243
       1/729
       1/2187
       1/6561
       1/19683
       1/59049
```

A stem plot.

```
>> stem(a)
```

The resulting plot.



17. A **for** loop.

```
>> a=zeros(10,1);
>> for k=1:10,
a(k)=(1-k)/(k+2);
end
>> a
a =
        0
      -1/4
      -2/5
      -1/2
      -4/7
      -5/8
      -2/3
      -7/10
      -8/11
      -3/4
```

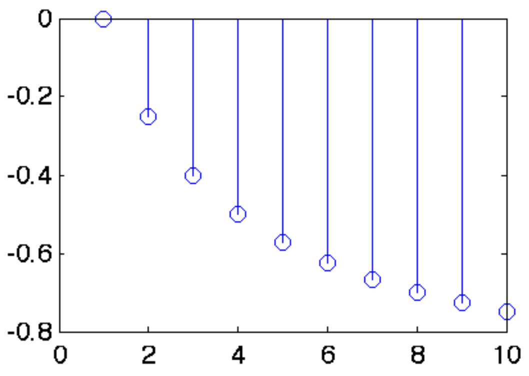
Same result with array ops.

```
>> a=(1-n)./(n+2)
a =
       0
      -1/4
      -2/5
      -1/2
      -4/7
      -5/8
      -2/3
      -7/10
      -8/11
      -3/4
```

A stem plot.

```
>> stem(a)
```

The resulting plot.



19. A **for** loop with default format.

```
>> format
>> a=zeros(10,1);
>> for k=1:10,
a(k)=cos(2*k*pi/5);
end
>> a
a =
    0.3090
   -0.8090
   -0.8090
    0.3090
    1.0000
    0.3090
   -0.8090
   -0.8090
    0.3090
    1.0000
```

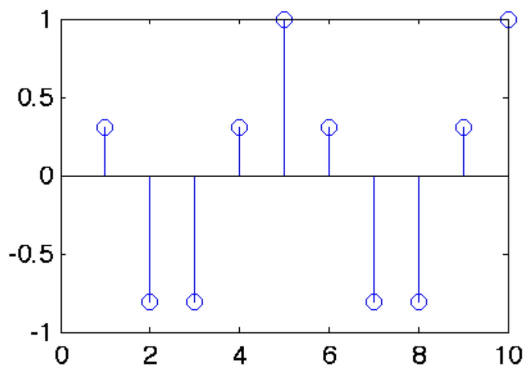
Same result with array ops.

```
>> n=(1:10).';
>> a=cos(2*n*pi/5)
a =
    0.3090
   -0.8090
   -0.8090
    0.3090
    1.0000
    0.3090
   -0.8090
   -0.8090
    0.3090
    1.0000
```

A stem plot.

```
>> stem(a)
```

The resulting plot.



21. Iniialization and **for** loop.

```
>> a=zeros(30,1);
>> a(1)=1;
>> for k=2:30
a(k)=sqrt(2+a(k-1));
end
```

Result.

```
>> format long
>> a
a =
   1.00000000000000
   1.73205080756888
   1.93185165257814
   1.98288972274762
   1.99571784647721
   1.99892917495273
   1.99973227581912
   1.99993306783480
   1.99998326688870
   1.99999581671780
   1.99999895417918
   1.99999973854478
   1.99999993463619
   1.99999998365905
   1.99999999591476
   1.99999999897869
   1.99999999974467
   1.99999999993617
   1.99999999998404
   1.99999999999601
   1.99999999999900
   1.99999999999975
   1.99999999999994
   1.99999999999998
   2.00000000000000
   2.00000000000000
   2.00000000000000
   2.00000000000000
   2.00000000000000
   2.00000000000000
```

It would appear that this sequence comverges to the number 2.

23. Summing with a **for** loop using default display format.

```
>> format
>> s=0;
>> for n=1:20,
s=s+1/n;
end
>> s
s =
   3.59773965714368
```

Using array ops.

```
>> n=1:20;
>> s=sum(1./n)
s =
   3.59773965714368
```

25. Summing with a **for** loop using default display format.

```
>> format
>> s=0;
>> for n=1:20,
s=s+1/(2^n);
end
>> s
s =
   0.99999904632568
```

Using array ops.

```
>> n=1:20;
>> s=sum(1./(2.^n))
s =
```

27. Summing with a **for** loop using default display format.

```
>> s=0;
>> for n=1:20,
s=s+n/(n+1);
end
>> s
s =
  17.35464129523727
```

Using array ops.

```
>> s=sum(n./(n+1))
s =
  17.35464129523727
```

29. Summing with a **for** loop using default display format.

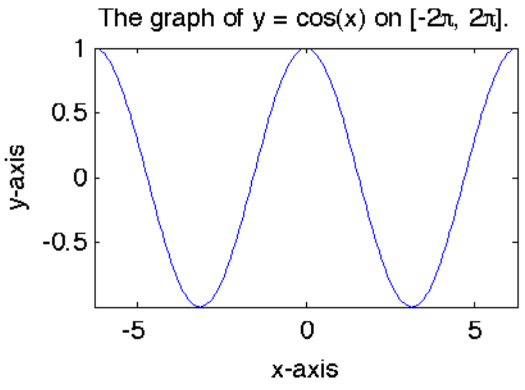
```
>> s=0;
>> for n=1:20,
s=s+1/factorial(n);
end
>> s
s =
   1.71828182845905
```

Using array ops.

```
>> s=sum(1./factorial(n))
s =
   1.71828182845905
```

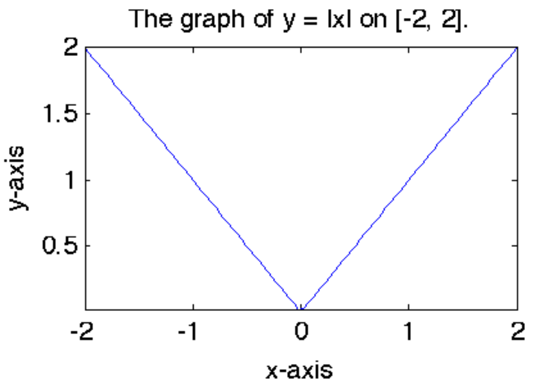
31. The following script file was used to produce that graph that follows.

```
clear all
close all
x=linspace(-2*pi,2*pi,200);
y=cos(x);
plot(x,y)
axis tight
xlabel('x-axis')
ylabel('y-axis')
title('The graph of y = cos(x)
on [-2\pi, 2\pi].')
```



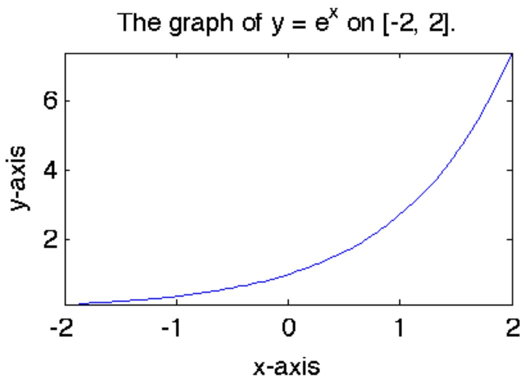
33. The following script file was used to produce that graph that follows.

```
clear all
close all
x=linspace(-2,2,100);
y=abs(x);
plot(x,y)
axis tight
xlabel('x-axis')
ylabel('y-axis')
title('The graph of y = |x| on
[-2, 2].')
```



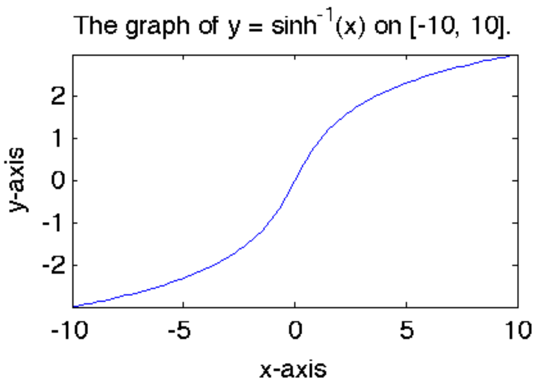
35. The following script file was used to produce that graph that follows.

```
clear all
close all
x=linspace(-2,2,100);
y=exp(x);
plot(x,y)
axis tight
xlabel('x-axis')
ylabel('y-axis')
title('The graph of y = e^x on
[-2, 2].')
```



37. The following script file was used to produce that graph that follows.

```
clear all
close all
x=linspace(-10,10,100);
y=asinh(x);
plot(x,y)
axis tight
xlabel('x-axis')
ylabel('y-axis')
title('The graph of y = sinh^{-
1}(x) on [-10, 10].')
```


The graph of y = sinh⁻¹(x) on [-10, 10].
y-axis
x-axis