2.1 Vectors in Matlab

Matlab gets its name from the fact that it is a

MATrix **LAB**oratory.

The basic data structure in Matlab is the matrix. Even scalars (numbers) are considered to be 1×1 matrices (1 row and 1 column).

```
>> x=3
x =
     3
>> whos('x')
  Name      Size                    Bytes  Class

  x         1x1                         8  double array
```

Note that Matlab recognizes the **Size** of the variable x as a 1-by-1 matrix.

Let's now look at how vectors are stored in Matlab.

Row Vectors

A row vector has the form

$$\mathbf{v} = \begin{bmatrix} v_1 & v_2 & \dots & v_n \end{bmatrix}, \tag{2.1}$$

where v_1, v_2, ..., v_n are usually scalars (either real or complex numbers)[2].

When you enter a row vector in Matlab, you can separate the individual elements with commas.

```
>> v=[1, 2, 3, 4, 5]
v =
     1     2     3     4     5
```

You can also delimit the individual elements with spaces.

---

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

[2] We will see in later work that the entries in a vector can be of other data types, such as strings, for example.

```
>> v=[1 2 3 4 5]
v =
     1     2     3     4     5
```

You can determine the length of a row vector with Matlab's **length** command.

```
 >> length(v)
ans =
     5
```

You can access the element of **v** at position k with Matlab's indexing notation **v(k)**. For example, the element in the third position of vector **v** is found as follows.

```
 >> v(3)
ans =
     3
```

You can access the 1st, 3rd, and 4th entries of vector **v** as follows.

```
>> v([1,3,4])
ans =
     1     3     4
```

You can use indexing to change the entry in the 5th position of **v** as follows.

```
>> v(5)=500
v =
     1     2     3     4   500
```

You can change the 1st, 3rd, and 5th entries as follows. Note that the vector on the right must have the same length as the area to which it is assigned.

```
>> v([1,3,5])=[-10 -20 -30]
v =
   -10     2   -20     4   -30
```

If you break the equal length rule, Matlab will respond with an error message.

```
>> v([2,4])=[100 200 300]
???  In an assignment  A(I) = B, the number of elements in B and
 I must be the same.
```

There is one exception to this rule. You may assign a single value to a range of entries in the following manner.

```
>> v([1,3,5])=0
v =
     0     2     0     4     0
```

Column Vectors

A column vector has the form

$$\mathbf{v} = \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{bmatrix} \tag{2.2}$$

where v_1, v_2, ..., v_n are usually scalars (either real or complex numbers).

In Matlab, use the semicolon to end a row and begin a new row. This is useful in building column vectors.

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

The length of a column vector is determined in exactly the same way the length of a row vector is determined.

```
>> length(w)
ans =
     3
```

Indexing works the same with column vectors as it does with row vectors. You can access the second element of the vector **w** as follows.

```
>> w(2)
ans =
     2
```

You can use indexing notation to change the entry in the second position of **w** as follows.

```
>> w(2)=15
w =
     1
    15
     3
```

Matlab's Transpose Operator

The transpose of a row vector is a column vector, and vice-versa, the transpose of a column vector is a row vector. Mathematically, we use the following symbolism to denote the transpose of a vector.

$$\begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{bmatrix}^T = \begin{bmatrix} v_1 & v_2 & \dots & v_n \end{bmatrix}$$

In general, we have need of two differnt types of transpose operators: (1) a regular transpose operator, and (2) a conjugate transpose operator. To take a regular transpose, transposing a row vector to a column vector, or vice versa, a column vector to a row vector, Matlab uses the the operator .' (that's a period followed by a single apostrophe).

```
>> u=[1;3;5;7]
u =
     1
     3
     5
     7
>> u.'
ans =
     1     3     5     7
```

On the other hand, the conjugate transpose operator is a single apostrophe. Like a the regular transpose, the conjugate transpose also changes row vectors to column vectors, and vice-versa. But it also takes the complex conjugagte[3] of each entry.

```
>> c=[2+3i, i, 9, -1-2i]
c =
   2.0000 + 3.0000i   0 + 1.0000i  9.0000   -1.0000 - 2.0000i
>> c'
ans =
   2.0000 - 3.0000i
        0 - 1.0000i
   9.0000
  -1.0000 + 2.0000i
```

Increment Notation

One can easily create vectors with a constant increment between entries. You use Matlab's `start:increment:finish` construct for this purpose. In the case where no increment is supplied, the increment is understood to be 1.

```
>> x=1:10
x =
     1     2     3     4     5     6     7     8     9    10
```

As a second example, **0:0.5:10** will create a vector that contains entries starting at zero and finishing at 10, and the difference (the increment) between any two entries is 0.5.

[3] The complex conjugate of $a + bi$ is $a - bi$.

```
>> x=0:0.5:10
x =
  Columns 1 through 6
         0    0.5000    1.0000    1.5000    2.0000    2.5000
  Columns 7 through 12
    3.0000    3.5000    4.0000    4.5000    5.0000    5.5000
  Columns 13 through 18
    6.0000    6.5000    7.0000    7.5000    8.0000    8.5000
  Columns 19 through 21
    9.0000    9.5000   10.0000
```

In this case, not that the row vector is too long to be contained in the width of the screen. Thus, the result wraps around. We can keep track of the column we're in by noting the headers. For example, **Columns 1 through 8** indicates that we're looking at the entries 1 through eight of the row vector **x**.

Of course, if we'd rather store a column vector in **x**, we can use the transpose operator.

```
>> y=(0:0.2:1).'
y =
         0
    0.2000
    0.4000
    0.6000
    0.8000
    1.0000
```

Increment notation can be extremely useful in indexing. For example, consider the following vector **v**.

```
>> v=100:-10:10
v =
   100    90    80    70    60    50    40    30    20    10
```

We can access every entry, starting at the 4th entry and extending to the last entry in the vector with the following notation.

```
>> v(5:end)
ans =
    60    50    40    30    20    10
```

We can access the entry in the even positions of the vector as follows.

```
>> v(2:2:end)
ans =
    90    70    50    30    10
```

Suppressing Output. There are times we will want to suppress the output of a command, particularly if it is quite lengthy. For example, the output of the command **z=0:0.1:100** will have over 1000 entries. If we place a semicolon at the end of this command, output of the command to the screen is *suppressed*. That is, the vector is stored in the variable **z**, but nothing is printed to the screen.

```
>> z=0:0.1:100;
```

Initialization with Zeros

Matlab has a number of commands to build special vectors. Let's explore just a few.

Unlike most computer languages, Matlab does not require that you declare the type or the dimension of variable ahead before assigning it a value. However, the Mathworks recommends that in certain situtaions it is more efficient to initialize a variable before acessing and/or assigning values with a program.

► **Example 1.** *It is well known that the infinite series*

$$e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!} = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \cdots + \frac{x^n}{n!} + \cdots \tag{2.3}$$

converges to e^x. Use the first 20 terms of the series to approximate e.

We are presented with an immediate difficulty. Many mathematical notations start their indexing at zero, as we see in the summation notation for the infinite series (**2.3**). However, the first entry of every vector in Matlab has index 1, not index 0. Often, we can shift the index in a mathematical expression to alleviate this conflict. For example, we can raise the index in the summation notation in

(2.3) by 1 if we subsequently lower the index in the expression by 1. In this manner, we arrive at

$$e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!} = \sum_{n=1}^{\infty} \frac{x^{n-1}}{(n-1)!}.$$

Note that this latter expression produces the same series. That is,

$$e^x = \sum_{n=1}^{\infty} \frac{x^{n-1}}{(n-1)!} = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \cdots + \frac{x^n}{n!} + \cdots. \tag{2.4}$$

To find e, or equivalently, e^1, we must substitute $x = 1$ into the infinite series (2.4). We will also use the first 20 terms to find an approximation of e. Thus,

$$e^1 = \sum_{n=1}^{\infty} \frac{(1)^{n-1}}{(n-1)!} \approx \sum_{n=1}^{20} \frac{1}{(n-1)!} = 1 + 1 + \frac{1}{2!} + \frac{1}{3!} + \frac{1}{4!} + \cdot + \frac{1}{19!}. \tag{2.5}$$

We will first use Matlab's **zeros** command to initialize a column vector having 20 entries, all of which are initially zero.

```
>> S=zeros(20,1);
```

We've suppressed the output. You might want to remove the semicolon at the end of this command to see that you have a column vector with 20 entries, all of which are zero.

We'll now write a **for** loop[4] to compute the partial sums of series (2.5) and record them in the row vector **S**. In the first position of **S** we'll place the first term of the series (2.5). After the loop completes, the second entry of **S** will contain the sum of the first two terms of (2.5), the third entry of **S** will contain the sum of the first three terms of (2.5), etc. The twentieth entry of the vector **S** will contain the sum of the first twenty terms of (2.5).

We employ several new ideas on which we should comment.

1. The command **format long**[5] displays more than three times as many digits than the default format.

2. Note that several commands can be put on a single command line. We need only separate the commands with commas. In the case where we want to suppress the output, we use a semicolon instead of a comma.

[4] We'll discuss **for** loops in detail in a later section.
[5] To return to default display, type **format**.

3. The command **for k=2:20** uses **start:increment:finish** notation. Because there is no increment in **2:20**, the increment is assumed to be 1. Hence, the command **for k=2:20** sets k equal to 2 on the first pass through the loop, sets $k = 3$ on the second pass through the loop, and so on, then finally sets $k = 20$ on the last pass through the loop.

4. Matlab's command for $k!$ is **factorial(k)**.

5. The entries of **S** contain the partial sums of the series (**2.5**). Hence, to get the kth entry of **S**, we must add the kth term of the series (**2.5**) to the $(k-1)$th term of **S**.

```
>> format long
>> S(1)=1; for k=2:20, S(k)=S(k-1)+1/factorial(k-1); end, S
S =
   1.00000000000000
   2.00000000000000
   2.50000000000000
   2.66666666666667
   2.70833333333333
   2.71666666666667
   2.71805555555556
   2.71825396825397
   2.71827876984127
   2.71828152557319
   2.71828180114638
   2.71828182619849
   2.71828182828617
   2.71828182844676
   2.71828182845823
   2.71828182845899
   2.71828182845904
   2.71828182845905
   2.71828182845905
   2.71828182845905
```

Note the apparent convergence to the number 2.71828182845905.

In Matlab, the expression **exp(x)** is equivalent to the mathematical expression e^x.

```
>> exp(1)
ans =
   2.71828182845905
```

Hence, it appears that the series (2.3) converges to e, at least as far as indicated by the sum of the first twenty terms in (2.5).

The following command will plot the entries in **S** versus their indicies (1:20).

```
>> plot(S,'*')
```

Note the rapid convergence to e in **Figure 2.1**.

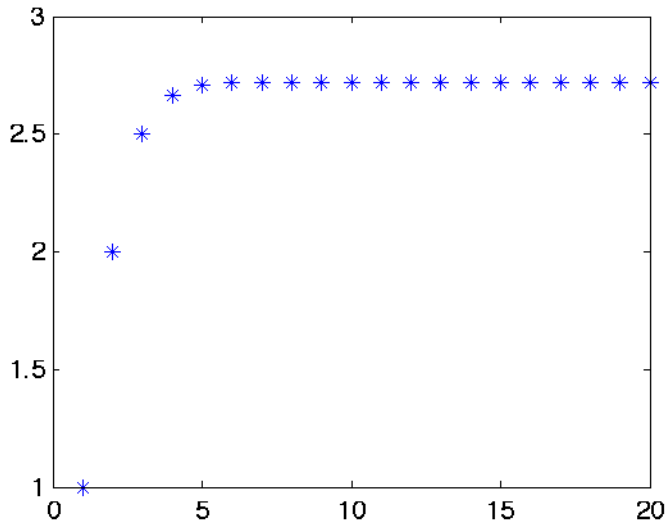


Figure 2.1. Plotting partial sums of series (2.5).

Scalar Multiplication

A scalar is a number, usually selected from the real or complex numbers. To multiply a scalar times a vector, simply multiply each entry of the vector by the scalar. In symbols,

$$\begin{aligned} \alpha\mathbf{v} &= \alpha\begin{bmatrix} v_1 & v_2 & \dots & v_n \end{bmatrix} \\ &= \begin{bmatrix} \alpha v_1 & \alpha v_2 \dots & \alpha v_n \end{bmatrix}. \end{aligned}$$

For example,

$$2\begin{bmatrix}1 & 2 & 3 & 4 & 5\end{bmatrix} = \begin{bmatrix}2 & 4 & 6 & 8 & 10\end{bmatrix}.$$

Matlab handles scalar multiplication easily.

```
>> v=1:5
v =
     1     2     3     4     5
>> w=2*v
w =
     2     4     6     8    10
```

To multiply a column vector by a scalar, multiply each entry of the column vector by the scalar. Note that scalar multiplication is handled identically for both row and column vectors.

```
>> v=(1:3)'
v =
     1
     2
     3
>> w=2*v
w =
     2
     4
     6
```

Let's look at another example.

▶ **Example 2.** *Generate a vector that contains 1000 uniform random numbers on the interval* $[0, 10]$. *Draw a histogram of the data set contained in the vector.*

Matlab's **rand** command is used to generate uniform random numbers on the interval $[0, 1]$. By uniform, we mean any number in the interval $[0, 1]$ has an equally like chance of being selected.

```
>> v=rand(1000,1);
```

The histogram in **Figure 2.2** is created with Matlab's **hist(v)** command.[6]

```
>> hist(v)
```

In **Figure 2.2**, note that each of the bins contains approximately 100 numbers, lending credence that any real number from the interval $[0, 1]$ has an equally likely chance of being selected.

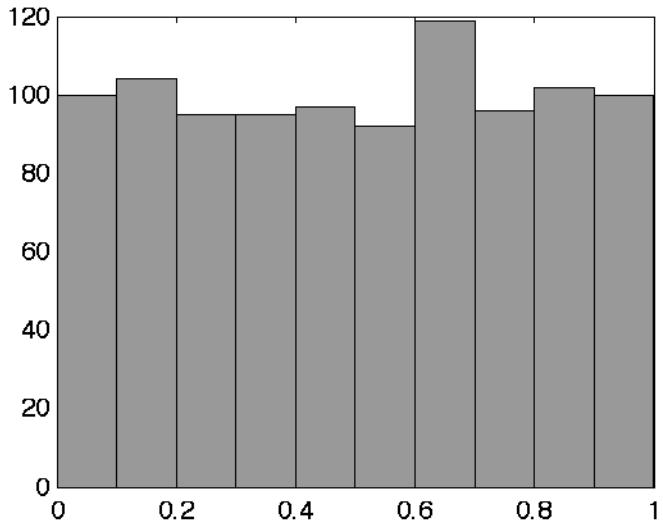


Figure 2.2. A histogram of 1000 uniform random numbers selected from the interval $[0, 1]$.

We can represent the fact that Matlab's **rand** command selects a random number from the interval $[0, 1]$ with the symbolism

$$0 \leq \text{rand} \leq 1.$$

If we want to generate uniform random numbers on the interval $[0, 10]$, multiply all three members of the last inequality by 10 to produce the result

$$0 \leq 10 \cdot \text{rand} \leq 10. \tag{2.6}$$

Hence, the expression $10 \cdot \text{rand}$ should generate uniform random numbers on the interval $[0, 10]$.

[6] Some readers might want to learn how to produce more granular control over the appearance of their histogram. If so, type **help hist** for more information.

To generate 1000 uniform random numbers from the interval $[0, 10]$, we start by generating a column vector of length 1000 with the expression **rand(1000,1)** (1000 rows and 1 column). This vector contains uniform random numbers selected from the interval $[0, 1]$. Multiply this vector by 10, which multiplies each entry by 10 to produce uniform random numbers from the interval $[0, 10]$. Matlab's **hist(v)** command produces the histogram in **Figure 2.3**

```
>> v=10*rand(1000,1); hist(v)
```

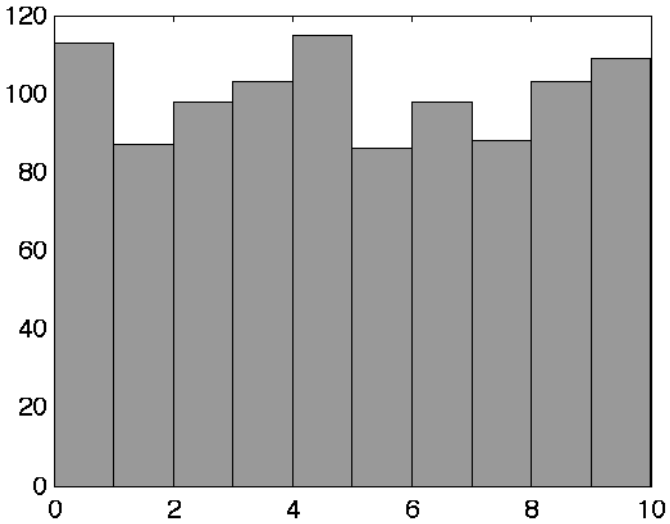


Figure 2.3. A histogram of 1000 uniform random numbers selected from the interval $[0, 10]$.

Note that each of the 10 bins of the histogram in **Figure 2.3** appears to have approximately 100 elements, lending credence to a uniform distribution, one where each real number between 0 and 10 has an equal chance of being selected.

Vector Addition

You add two row vectors of equal length by adding the corresponding entries. That is,

$$\begin{bmatrix} u_1 & u_2 & \dots & u_n \end{bmatrix} + \begin{bmatrix} v_1 & v_2 & \dots & v_n \end{bmatrix} = \begin{bmatrix} u_1 + v_1 & u_2 + v_2 & \dots & u_n + v_n \end{bmatrix}.$$

Matlab handles vector addition flawlessly.

```
>> u=1:4
u =
     1     2     3     4
>> v=5:8
v =
     5     6     7     8
>> u+v
ans =
     6     8    10    12
```

Column vectors are added in the same manner as row vectors. That is, add the corresponding entries.

```
>> u=[1,2]', v=[3,4]'
u =
     1
     2
v =
     3
     4
>> u+v
ans =
     4
     6
```

You cannot add two vectors of different lengths.

```
>> u=1:3
u =
     1     2     3
>> v=1:4
v =
     1     2     3     4
>> u+v
??? Error using ==> plus
Matrix dimensions must agree.
```

It is not legal to add a scalar to a vector. That is, the expression

$$\begin{bmatrix}1 & 2 & 3\end{bmatrix} + 5$$

makes no sense whatsoever. However, it turns out that this is such a common requirement, Matlab allows the addition of a scalar and vector. To do so, Matlab interprets

$$\begin{bmatrix}1 & 2 & 3\end{bmatrix} + 3$$

to mean

$$\begin{bmatrix}1 & 2 & 3\end{bmatrix} + \begin{bmatrix}3 & 3 & 3\end{bmatrix},$$

which is easily seen in the following computation.

```
>> v=1:3
v =
     1     2     3
>> w=v+3
ans =
     4     5     6
```

Note that 3 was added to each entry of vector **v**.

Let's look at example where this feature is useful.

▶ **Example 3.** *Generate 1000 random numbers from a normal distribution with mean 100 and standard deviation 50. Produce a histogram of the data.*

Matlab's **randn** command is used to generate random numbers from the *standard* normal distribution having mean 0 and standard deviation 1.

```
>> v=randn(1000,1); hist(v)
```

The resulting histogram in **Figure 2.4** appears to possess the familiar bell-shape of the standard normal distribution. Note that the histogram appears to be centered about the number 0, lending credence to the fact that we drew numbers from the standard normal distribution, which is known to have mean zero. Further, note that most of the data (in this case all) is contained between -3 and 3. The standard deviation of the standard normal distribution is 1, and it is a a fact that most, if not all, of the data should fall within three standard deviations of the mean (which it does in **Figure 2.4**).

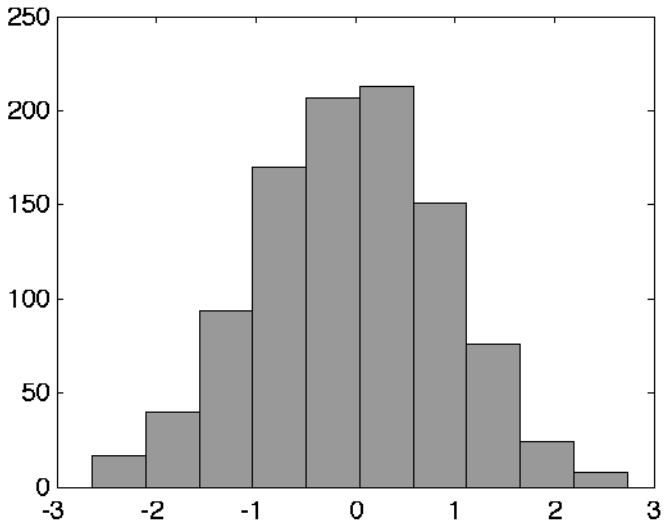


Figure 2.4. A histogram of 1000 random numbers selected from the standard normal distribution having mean 0 and standard deviation 1.

In creating **Figure 2.4**, the Matlab command **randn(1000,1)** produced numbers that fell between −3 and 3. It is reasonable to expect that if we we multiply each of these entries by 50, the range of numbers will now fall between −150 and 150. To center the new distribution about 1000, we need only shift the numbers 100 units to the right by adding 100 to every entry. The resulting histogram is shown in **Figure 2.5**.

```
>> v=100+50*randn(1000,1); hist(v)
```

Two attributes of the histogram in **Figure 2.5** seem reasonable.

1. The histogram appears to be bell-shaped and centered about 100, making the mean approximately 100 as requested.

2. Three standard deviations of 50 is 150. If we subtract this from 100, then add it to 100, then the random numbers should range from −50 to 150, which is in approximate agreement with the histogram in **Figure 2.5**.

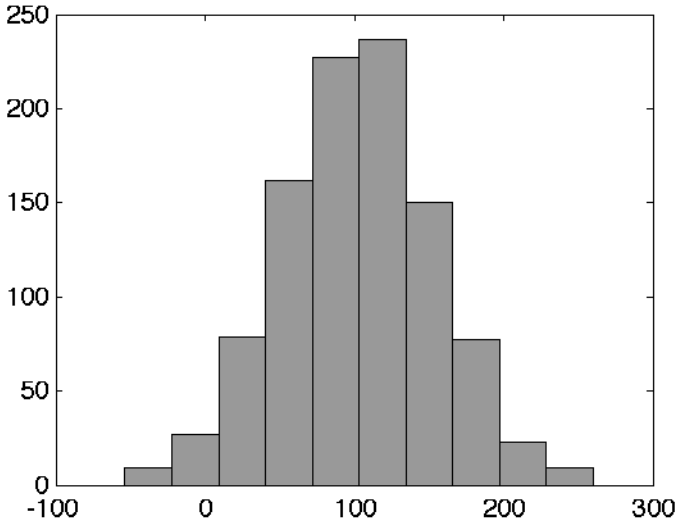


Figure 2.5. A histogram of 1000 random numbers selected from a normal distribution with mean 100 and standard deviation 50.

2.1 Exercises

In **Exercises 1-4**, use Matlab's **length** command to find the length of each of the given vectors.

1. **x=1:0.01:5**

2. **y=2:0.005:3**

3. **z=(100:0.5:200).'**

4. **x=(10:10:1000).'**

In **Exercises 5-6**, perform each of the following tasks.

i. Use Matlab's **linspace(a,b,n)** command to generate n equally spaced numbers between a and b for the given values of a, b, and n.
ii. Use Matlab's indexing notation to zero out every odd indexed entry.
iii. Use Matlab's **stem** command to plot the elements in the resulting vector versus their indices.

5. $a = 0$, $b = 10$, $n = 20$.

6. $a = -5$, $b = 5$, $n = 25$.

7. Use Matlab's **sum** command and **start:increment:finish** construct to find the sum of the integers from 1 to 1000.

8. Use Matlab's **sum** command and **start:increment:finish** construct to find the sum of the integers from 1000 to 10000.

9. Use Matlab's **sum** command and **start:increment:finish** construct to find the sum of the even integers from 1 to 1000.

10. Use Matlab's **sum** command and **start:increment:finish** construct to find the sum of the odd integers from 1 to 1000.

In **Exercises 9-14**, perform each of the following tasks for the given vector.

i. Use Matlab's **sum** command to find the sum of all entries in the given vector.
ii. Use Matlab's **sum** command and **start:increment:finish** construct to find the sum of of all the numbers with even indices in the given vector.
iii. Use Matlab's **sum** command and **start:increment:finish** construct to find the sum of of all the numbers with odd indices in the given vector.
iv. Verify that the results in part (ii) and (iii) sum to the result in part (i).

11. **x=0:0.05:10**

12. **x=10:3:120**

13. **x=(1000:10:2000).'**

14. **x=(150:5:350).'**

15. Let $\mathbf{u} = [1, 2, 3]$ and $\mathbf{v} = [4, 5, 6]$.

a) Use Matlab to compute $(\mathbf{u} + \mathbf{v})^T$.

b) Use Matlab to compute $\mathbf{u}^T + \mathbf{v}^T$ and compare to the result in part (a). Explain what you learned in this exercise.

16. Let $\mathbf{u} = [1, 2, 3]$ and $\alpha = 4$.

a) Use Matlab to compute $(\alpha\mathbf{u})^T$.

b) Use Matlab to compute $\alpha\mathbf{u}^T$ and compare to the result in part (a). Explain what you learned in this exercise.

In **Exercises 15-20**, write a simple **for** loop to populate a column vector with the first 24 terms of the given sequence. Use the **plot** command to plot the terms of the sequence versus it indices.

17. $a_n = (0.85)^n$

18. $a_n = (1.25)^n$

19. $a_n = \sin(\pi n/8)$

20. $a_n = -3\cos(\pi n/12)$

---

21. If r is a random number between 0 and 1, determine the range of the expression $(b - a)r + a$ where a and b are any real numbers with $a < b$.

In **Exercises 20-25**, perform each of the following tasks.

i. Fill a vector with 1000 uniformly distributed random numbers on the given interval. *Hint: For a helpful hint, see **Exercise 19**.*
ii. Use Matlab's **hist** command to draw a histogram of the random numbers stored in your vector.
iii. Explain why your histogram is a reasonable answer.

22. $[2, 6]$

23. $[1, 11]$

24. $[-5, 5]$

25. $[-10, 20]$

In **Exercises 24-29**, perform each of the following tasks.

i. Fill a vector with 1000 random numbers that are drawn from a normal distribution with the given mean μ and standard deviation σ.
ii. Use Matlab's **hist** command to draw a histogram of the random numbers stored in your vector.
iii. Explain why your histogram is a reasonable answer.

26. $\mu = 100$, $\sigma = 20$

27. $\mu = 50$, $\sigma = 10$

28. $\mu = 200$, $\sigma = 40$

29. $\mu = 150$, $\sigma = 30$

---

30. Read the documentation for Matlab's **primes** command, and use it to store the first 100 primes less than or equal to 1000 in a vector.

a) Find the sum of the first 100 primes.

b) Find the sum of the first, 20th, and 97th primes.

31. The **bar(v)** command generates a bar graph of the elements of the vector **v**. Assume that you are the CEO of a corporation that sells scented shoes on the Internet. You have just received data explaining that your company's profits between 1995 and 2004 were, respectively, \$2M, \$3M, \$7M, \$8M, \$14M, \$15M, \$6M, \$3M, \$2M, and \$1M. Meanwhile, your company spent a steady \$3M per year manufacturing the shoes. Use your knowledge of vectors and the **bar** command to generate a bar graph of your company's net profits[7] for 1995–2004, and decide whether or not you should keep your stock options in the company.

[7] Gross profits are the total amount earned in a given time period, while net profits take into account expenditures as well.

2.1 Answers

1.

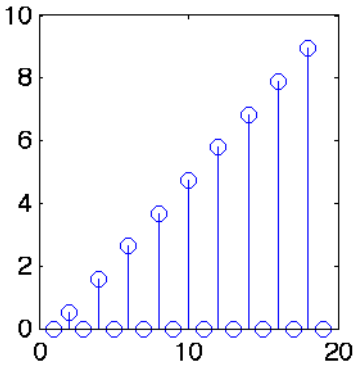
```
>> x=1:0.01:5;
>> length(x)
ans =
   401
```

3.

```
>> z=(100:0.5:200).';
>> length(z)
ans =
   201
```

5.

```
>> x=linspace(0,10,20);
>> x(1:2:end)=0;
>> stem(x)
```



7.

```
>> x=1:1000;
>> sum(x)
ans =
      500500
```

9.

```
>> x=2:2:1000;
>> sum(x)
ans =
      250500
```

11.

```
>> sum(x)
ans =
      250500
>> x=0:0.05:10;
>> sum(x(2:2:end))
ans =
   500
>> sum(x(1:2:end))
ans =
   505
>> sum(x)
ans =
   1005
```

13.

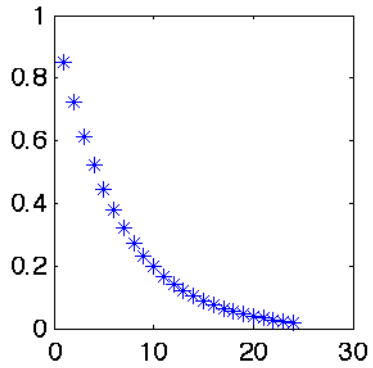
```
>> sum(x)
ans =
   1.0050e+03
>> x=(1000:20:2000).';
>> sum(x(2:2:end))
ans =
       37500
>> sum(x(1:2:end))
ans =
       39000
>> sum(x)
ans =
       76500
```

15. The transpose of a sum of two vectors is the sum of the transposes of the two vectors.

```
>> u=1:3; v=4:6;
>> (u+v).'
ans =
     5
     7
     9
>> u.'+v.'
ans =
     5
     7
     9
```

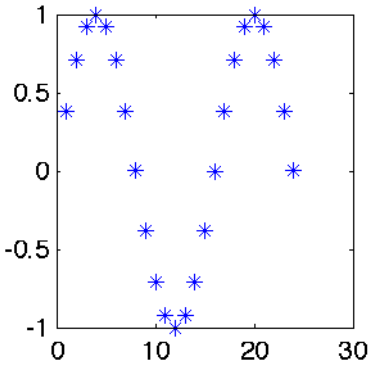
17.

```
>> a=zeros(24,1);
>> for n=1:24, a(n)=(0.85)^n;
end
>> plot(a,'*')
```



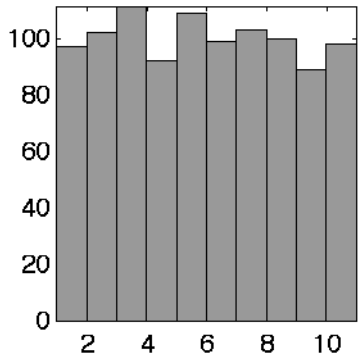
19.

```
>> a=zeros(24,1);
>> for n=1:24, a(n)=sin(pi*n/8);
end
>> plot(a,'*')
```



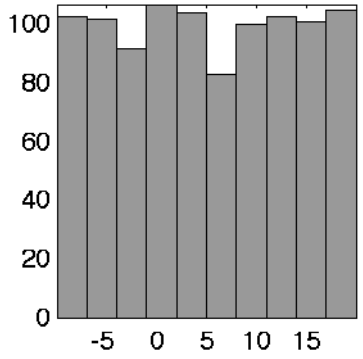
23. In the following histogram, there appears to be an equal amount of numbers in each bin over the range $[1, 11]$, lending evidence that the following set of commands select 1000 uniform random numbers from the interval $[1, 11]$.

```
>> a=1; b=11;
>> x=(b-a)*rand(1000,1)+a;
>> hist(x)
```



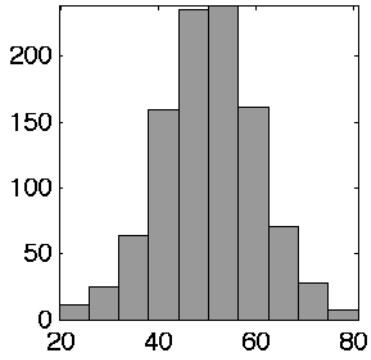
25. In the following histogram, there appears to be an equal amount of numbers in each bin over the range $[-10, 20]$, lending evidence that the following set of commands select 1000 uniform random numbers from the interval $[-10, 20]$.

```
>> a=-10; b=20;
>> x=(b-a)*rand(1000,1)+a;
>> hist(x)
```



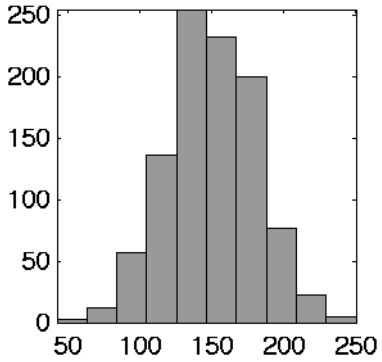
27. In the following histogram, the histogram appears to be centered around the mean $\nu = 50$. Further, if $\sigma = 10$, three standard deviations to the left of $\mu = 50$ is 10, and three standard deviations to the right of $\mu = 50$ is 80. In the histogram that follows, it would appear that the range is from 50 to 80, lending evidence that we've selected random numbers from a normal distribution with mean $\mu = 50$ and standard deviation $\sigma = 10$.

```
>> mu=50; sigma=10;
>> x=mu+sigma*randn(1000,1);
>> hist(x)
```



29. In the following histogram, the histogram appears to be centered around the mean $\nu = 150$. Further, if $\sigma = 30$, three standard deviations to the left of $\mu = 150$ is 60, and three standard deviations to the right of $\mu = 150$ is 240. In the histogram that follows, it would appear that the range is approximately from 50 to 250, lending evidence that we've selected random numbers from a normal distribution with mean $\mu = 150$ and standard deviation $\sigma = 30$.

```
>> mu=150; sigma=30;
>> x=mu+sigma*randn(1000,1);
>> hist(x)
```



31. The prognosis does not look good for your company's health.

```
>> v=[2 3 7 8 14 15 6 3 2 1]-
3;
>> bar(v)
```