Lesson 2: Creating Arrays in MATLAB

☐ In Lesson 1, we explored many features of the MATLAB Desktop environment including the use of the command window as a basic calculator, how to write .m script files and a brief introduction to variables

☐ In all of the work we did in Lesson 1, we focused on storing, manipulating, and analyzing scalar-valued numeric data. In other words, all of the mathematical operations we performed and variables we saved in lesson one were scalars (individual numbers).

☐ In Lesson 2, we expand our perspective and explore how we can use MATLAB to create, store, and operate on arrays. As we will discuss, MATLAB stores all numerical data as arrays.

- One of the unique and very powerful features of the MATLAB computing environment is that MATLAB is an "array programming language",* meaning that

  1. all MATLAB variables are stored as arrays and thus each variable can contain multiple entries organized into rows or columns

     1A. The scalar variables we defined in lesson one to store single numbers are actually stored as numeric 1 x 1 arrays having 1 row and 1 column.

  2. MATLAB operations are designed to be generalized from scalars to vectors to matrices. In fact, MATLAB is designed to operate primarily on arrays by taking advantage of specialized hardware and software that is optimized to execute numerical computations on arrays (rather than on individual scalar

*Note: Most general purpose programming languages work mostly with variables as individual scalars unless the user imports special libraries to enable array-based operations

- A numeric array in MATLAB is an organized list of "numbers" specially arranged in rows or columns.

- In our study of arrays, we will focus on two types of arrays :

  Type 1 : one-dimensional array

  - These one-dimensional arrays are lists of numbers that are either stored as a single row or a single column.
  - We use one-dimensional numeric arrays to represent row or column vectors.

  Type 2: two-dimensional arrays

  - Two-dimensional arrays are collections of numbers arranged into <u>both</u> rows AND columns.
  - We use two-dimensional arrays to represent matrices.

Creating one-dimensional arrays to represent vectors

In the MATLAB Computing environment, one-dimensional arrays are often used to store row or column vectors. Just as the properties of real numbers form the building blocks of calculus, the properties of vectors set the foundations for applied linear algebra. We can use vectors and vector operations to do all of the following:

1. **Encode data:** Vectors encode multidimensional data in compact form.
2. **Combine data:** Vector addition and scalar-vector multiplication combine known data to form new, related information that gives insight into the nature of multidimensional data sets.
3. **Understand relationships:** The inner product of two vectors measures the angle between these vectors and indicates the degree to which one vector points in the same direction as another vector.
4. **Measure length and distance:** Vector norms measure the length of a vector and can be used to calculate distances between two vectors.
5. **Study algebraic properties:** By identifying the algebraic properties of vector operations, we enable a theoretic approach to constructing vectors and vector operations.

Vectors encode quantitative information. In this chapter, we define vectors as a list of real-valued data entries organized in a vertical or horizontal array. In linear algebra, vectors carry information as multiple dimensional data. The word vector originates from the latin word *advector*, which means carrier. There are two types of vectors we work with in this section: column vectors and row vectors. We begin with the definition of a column vector and illustrate the vast utility of this data structure.

Definition 2.1: Column vector

Let $n \in \mathbb{N}$. A **column vector** $\mathbf{x} \in \mathbb{R}^{n\times 1}$ has n real-valued entries, organized in n rows and one column, in the form

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}.$$

For each $i \in \{1, 2, ..., n\}$, the entry in the ith row of $\mathbf{x}$, labeled x_i, is a real number.

Column vectors are vertical arrays of real numbers. When writing $\mathbf{x} \in \mathbb{R}^{n\times 1}$, the superscript on $\mathbb{R}$ represent the dimensions of vector $\mathbf{x}$. The first argument, n, of the superscript indicates the number of rows of this vector. The second argument, 1, indicates the number of columns of $\mathbf{x}$. The times symbol, $\times$, separates the number of rows of the vector from the number of columns. We enumerate each entry of a vector using an index variable that we write as a subscript. We label the top most entry of a column vector as x_1 and increase the index variable by one as we move downward through the rows. This enumeration scheme continues until we reach the entry in the last row, which we call x_n.

© Jeffrey A. Anderson

EXAMPLE 2.1

Let's use column vectors to create a vertex model of a triangle in two dimensions. We define our three vertices as follows:

$$\mathbf{v}_1 = \begin{bmatrix} 0 \\ 0 \end{bmatrix}, \qquad \mathbf{v}_2 = \begin{bmatrix} -1 \\ 1 \end{bmatrix}, \qquad \mathbf{v}_3 = \begin{bmatrix} 2 \\ 1 \end{bmatrix}.$$

y

$\mathbf{v}_2$ $\mathbf{v}_3$

$\mathbf{v}_1$ x

In this model, we assume each vector represents a point in two-dimensional space. The entry in the first row of each vertex vector encodes distance along the horizontal axis and the entry in the second row encodes distance along the vertical axis. When we graph these vertices, we do not connect the dots because this model does not encode links between vertices. For now, our vertex model only defines points in the cartesian plane.

Now, let's see how to store these vectors in a one-dimensional, numeric array organized into a single column:

Command Window

```
>> v1 = [ 0; 0]

v1 =

     0
     0

>> v2 = [ -1; 1 ]

v2 =

    -1
     1

>> v3 = [ 2; 1 ]

v3 =

     2
     1

fx >>
```

In this example, we see two important features of storing column vectors as arrays in MATLAB which include:

1. The use of square brackets [] in defining an array
2. The use of a semicolon ; inside the array to delimit rows

Let's look at this in more detail.

© Jeffrey A. Anderson

Creating one-dimensional arrays

- To create one-dimensional arrays to represent vectors, we type all elements of our vectors inside the square brackets [ ] and mirror the following pattern to define arrays as variables:

variable_name = [type vector elements]

Let's start by exploring two options we have to create columns

Option 1: Using semi colons to delimit rows

- To create column vectors, we type left square bracket [, then the individual vector elements separated by a semicolon, and we finish our assignment by typing the right square bracket] after the last element.

- We also have a second option to create column vectors.

Option 2: Using Enter Key to delimit rows

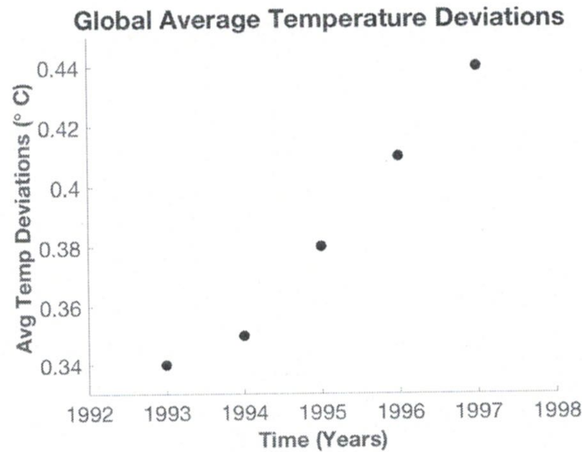
- To create a column vector, we type the left square bracket [, then the individual vector elements pressing the Enter key after each element, and we finish our assignment by typing the right square bracket] after the last entry of our vector.

- We can use either of these options to create column vectors from a list of known data that we would like to study.

EXAMPLE 2.2

As part of an effort to understand the effects of human activity on earth's climate, scientists study the changes in the average temperature around the globe. Below, we see a partial data set that presents global mean temperature deviations during the 1990's. The larger the deviations, the more likely it is that the climate is changing over time.

Year	Global Average Temperature Deviations (in °C)
1993	0.3400
1994	0.3500
1995	0.3800
1996	0.4100
1997	0.4400



We can use this data to create numerical columns arrays in MATLAB, as seen below:

```
Command Window
>> year = [1993; 1994; 1995; 1996; 1997]

year =

        1993
        1994
        1995
        1996
        1997

>> Avg_temp_dev = [
0.3400
0.3500
0.3800
0.4100
0.4400]

Avg_temp_dev =

    0.3400
    0.3500
    0.3800
    0.4100
    0.4400

fx >>
```

create a column vector using option 1 by delimiting rows using semicolon ;

Create a column vector using option 2 by delimiting rows using (Enter) Key.

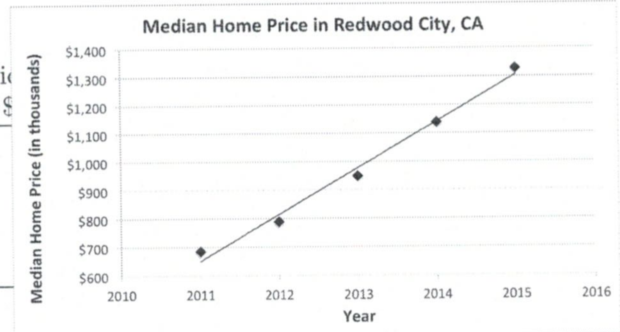
To avoid confusion and make our work easier to read, I recommend that we get in the habit of combining BOTH options 1 and 2 when storing column vectors, as we see in the example below.

© Jeffrey A. Anderson

EXAMPLE 2.3

The median sales price of existing homes in Redwood City has been steadily increasing from 2011 to 2015. If we set t to be the number of years after 2010, then we can list the median sales price (in thousands of dollars) for homes in Redwood City as:

Year	Median Home Pri (in thousands of $
2011	$684.0
2012	$787.5
2013	$949.0
2014	$1138.0
2015	$1328.0



Let's create numerical columns arrays in MATLAB using this data:

let's clear the "year" variable since we used same variable name in Example 2.2

Command Window

```
>> clear year
>> year = [
2011;
2012;
2013;
2014;
2015]

year =

        2011
        2012
        2013
        2014
        2015

>> RWC_home_price = [
684.0;
787.5;
949.0;
1138.0;
1328.0]

RWC_home_price =

   1.0e+03 *

    0.6840
    0.7875
    0.9490
    1.1380
    1.3280

fx >>
```

here we use both the semicolon and the enter key to specify a column vector from our data

This habit of using both semicolons and the enter key to delimit rows will come in particularly useful later in our work when we start to define two-dimensional arrays that represent matrices having both rows and columns. However, before we move onto matrices, let's first investigate how we can store row vectors in MATLAB.

© Jeffrey A. Anderson

Definition 2.2: Row Vector

Let $n \in \mathbb{N}$ be a positive integer. The vector $\mathbf{x} \in \mathbb{R}^{1\times n}$ given by

$$\mathbf{x} = \begin{bmatrix} x_1 & x_2 & \cdots & x_n \end{bmatrix}$$

is a **row vector** with n entries, $x_k \in \mathbb{R}$ for $k \in \{1, 2, ..., n\}$, organized in 1 row and n columns. When referring to a row vector $\mathbf{x} \in \mathbb{R}^{1\times n}$, the superscript on $\mathbb{R}$ represent the dimensions $1 \times n$ of the given vector. Just like when referring to column vectors, the first argument 1 indicates the number of rows and while the second argument n indicated the number of columns. Thus $\mathbb{R}^{1\times n}$ is the set of vectors that have 1 row and n columns.

It is less common to refer specifically to row vectors. Thus, it will always be necessary to write both dimensions when specifically referring to a row vector.

EXAMPLE 2.4

The population of the United States been growing for many years. Below we see evidence of this growth in data collected every ten years starting in 1930 and continuing through 2000.

Year	1930	1940	1950	1960	1970	1980	1990	2000
Population	123.2	132.2	151.3	179.3	203.3	226.5	248.7	271.4

We can store this data as one-dimensional arrays organized into rows, as seen below:

```
Command Window
>> clear year
>> year = [1930 1940 1950 1960 1970 1980 1990 2000]

year =

  Columns 1 through 5

        1930        1940        1950        1960        1970

  Columns 6 through 8

        1980        1990        2000

>> pop = [123.2, 132.2, 151.3, 179.3, 203.3, 26.5, 248.7, 271.4]

pop =

  Columns 1 through 6

  123.2000  132.2000  151.3000  179.3000  203.3000   26.5000

  Columns 7 through 8

  248.7000  271.4000

fx >>
```

© Jeffrey A. Anderson

- Just as we have two options to create column vectors, so two do we have two options to store vectors as rows in MATLAB.

Option 1: Using spaces to delimit columns

To create a row vector, type the left square bralket [, then enter each element with a space between them, and finish with the right square bracket]

Option 2: Using commas to delimit columns

Instead of spaces, we can also use commas to seperate elements. Again, my suggestion is to do both to make your data storage code easier to read.

Create a vector with constant spacing

- When we use MATLAB to model dynamics in the world around us, we will often want to create vectors with a known first and last term and a constant spacing between each element

- For example, suppose we want to count from 2 to 10 by twos to produce the row vector

$$t = [2 \quad 4 \quad 6 \quad 8 \quad 10]$$

MATLAB empowers us to create such vectors using special syntax.

To create a row vector with a known first and last element and constant spacing between each element, we will using the following syntax:

```
variable_name = [ m : q : n]
```

Each symbol in this statement has a very specific meaning:

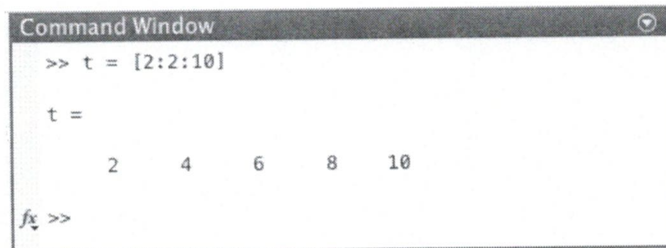
$$\texttt{m} = \text{the value of the first element in our vector} ,$$
$$\texttt{q} = \text{the desired spacing between each entry of our vector} ,$$
$$\texttt{n} = \text{the value of the last element in our vector} ,$$

EXAMPLE 2.5

Let's use this syntax to create the row vector t mentioned above and given by

$$t = \begin{bmatrix} 2 & 4 & 6 & 8 & 10 \end{bmatrix} .$$

To create and store this row vector in MATLAB, we can use the following code:


```
Command Window
>> t = [2:2:10]

t =

     2     4     6     8    10

fx >>
```

There are some special considerations to consider when using this syntax:

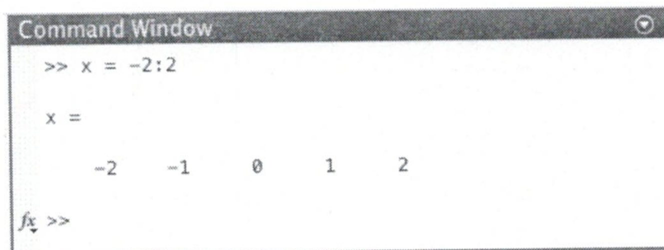
1. If only two numbers are types, MATLAB's default assumption is that we are specifying the first term `m` and the last term `n` while omitting a user-defined value for the spacing term `q`. In this case, MATLAB sets the default value for the space term `q` to be 1 as shown in Example 2.6 below.

EXAMPLE 2.6

Suppose we want to create a row vector x as given below

$$x = \begin{bmatrix} -2 & -1 & 0 & 1 & 2 \end{bmatrix} .$$

We notice that this vector that starts at $\texttt{m} = -2$, ends at $\texttt{n} = 2$, and has spacing $\texttt{q} = 1$. We can also omit the spacing `q`, as seen below.


```
Command Window
>> x = -2:2

x =

    -2    -1     0     1     2

fx >>
```

2. The spacing term q can be negative though in this case we need to make sure that $\mathtt{m} > \mathtt{n}$.

EXAMPLE 2.7

Suppose we want to create a row vector x as given below

$$x = \begin{bmatrix} -2 & -1 & 0 & 1 & 2 \end{bmatrix}.$$

This vector that starts at $\mathtt{m} = -2$, ends at $\mathtt{n} = 2$, and has spacing $\mathtt{q} = 1$. We can also omit the spacing q, as seen below.

```
Command Window
>> y = [100:-10:40]

y =

   100    90    80    70    60    50    40

>> error = [40:-10:100]

error =

   Empty matrix: 1-by-0

fx >> |
```

3. If the our three numbers m, n, and q are defined so that the value of the last term n is not a result of adding multiples of the spacing term q to the first term m, then MATLAB will automatically define the last element of the row vector to be the largest number possible that does not exceed n.

EXAMPLE 2.8

Suppose we type in the following values for our special terms:

$$\mathtt{m} = 0.0, \qquad \mathtt{q} = 0.3, \qquad \mathtt{n} = 1.0,$$

Notice that if we add multiples of $\mathtt{q} = 0.3$ to the starting value 0.0, we will never produce the value $\mathtt{n} = 1.0$ exactly. However, if we type these values into MATLAB, we get a decent substitute as described above:

```
Command Window
>> z = [0.0:0.3:1.0]

z =

        0    0.3000    0.6000    0.9000

fx >>
```

© Jeffrey A. Anderson

The `linspace` function to create row vectors

In the last section of this lesson, we focused on creating row vectors using colons via the syntax

```
variable_name = [ m : q : n]
```

We can use this code to create a row vector by setting a constant spacing term `q`. This is really helpful is we know the size of the spacing term in advance. Sometimes however, we might not know (or care about) the exact value of our spacing term `q` in advance. Instead, we might want to specify the number of terms in the vector. MATLAB's `linspace` function is very handy for this purpose.

To create a vector with equal spacing by specifying the first term, the last term, and the number of desired terms (rather than the size of the spacing), we can use the following syntax:

```
variable_name = linspace(x1, x2, n)
```

Each symbol in this statement has a very specific meaning:

$$\begin{aligned} \texttt{x1} &= \text{the desired initial term in our vector },\\ \texttt{x2} &= \text{the desired last (or final) term in our vector },\\ \texttt{n} &= \text{the desired number of (equally spaced) elements in our vector.} \end{aligned}$$

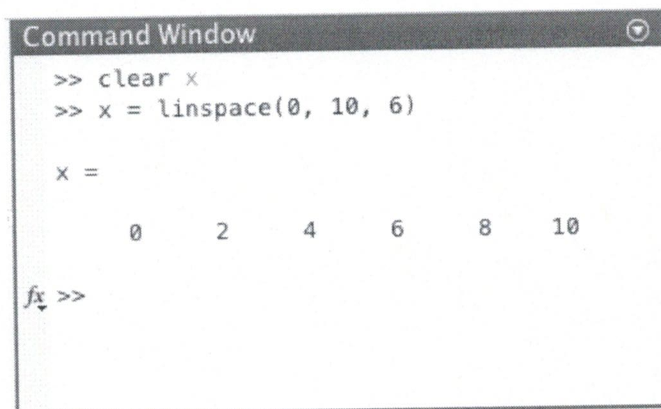
We can learn more about the `linspace` function by referring to MATLAB's documentation or by typing `help linspace` into the Command window.

EXAMPLE 2.9

Let's create a row vector x given by

$$x = \begin{bmatrix} 0 & 2 & 4 & 6 & 8 & 10 \end{bmatrix}.$$

that begins at $\texttt{x1} = 0$, ends at $\texttt{x2} = 10$ and has $\texttt{n} = 6$ elements.



Once again, we should be aware of some special features of the `linspace` function

1. We need to be sure to count both the first and last element of our vector when we are determining the desired number of elements.

EXAMPLE 2.10
Suppose we want to use `linspace` to create a vector with `x1` $= 0$, ends at `x2` $= 5$ whose elements are each successive integer. To do so, we need six elements, not five, as seen in the example below.

```
Command Window
>> clear all
>> t1 = linspace(0, 5, 5)

t1 =

         0    1.2500    2.5000    3.7500    5.0000

>> t2 = linspace(0, 5, 6)

t2 =

     0     1     2     3     4     5

fx >>
```

The lesson of this example is that if we want a vector with a predetermined spacing, we should either be prepared to think critically when using `linspace` or to use our previous `variable_name = [ m : q : n]` notation instead.

2. The first and last elements can be reverse ordered with `x1` $>$ `x2`

EXAMPLE 2.11
Suppose we want create am 1×11 row vector such that `x1` $= 60$ and `x2` $= 0$. To do so, we use the following code:

```
Command Window
>> y = linspace(60, 0, 11)

y =

    60    54    48    42    36    30    24    18    12     6     0

fx >> |
```

3. The default value for the number of elements `n` is 100. Thus, if we do omit the parameter `n` when calling the `linspace` function, MATLAB will automatically produce a 1×100 row vector.

© Jeffrey A. Anderson

EXAMPLE 2.12

If we do not explicitly specify the number of elements in `linspace`, MATLAB will use the default value and create a 1×100 row vector.

```
Command Window
>> long_vec = linspace(0, 10)

long_vec =

  Columns 1 through 7

         0    0.1010    0.2020    0.3030    0.4040    0.5051    0.6061

  Columns 8 through 14

    0.7071    0.8081    0.9091    1.0101    1.1111    1.2121    1.3131

  Columns 15 through 21

    1.4141    1.5152    1.6162    1.7172    1.8182    1.9192    2.0202

  Columns 22 through 28
```

As seen above, in this case we should probably use the semicolon ; to suppress the output since this vector is quite large.

In examples 2.10 and 2.12 above, we see our first example of MATLAB's tendency to overloaded functions. In computer science, *function overloading* is a programming concept that enables programmers to define two different functions that have the exact same name. In this case, we see the following two function calls

`linspace(0, 10, 6)` and `linspace(0, 10)`

produce very different output. The first function call provide three (input) arguments and produces a 1×6 row vector using these arguments. On the other hand, the second function call requires only two (input) arguments and automatically creates a 1×100 row vector.

© Jeffrey A. Anderson

Creating two-dimensional arrays to represent matrices

Matrices are used to model phenomenon requiring multiple dimensional data. Applications areas exist in almost every branch of physics including classical mechanics, statics, dynamics, electromagnetism, optics, and quantum mechanics. We also use matrices in computer graphics to create 2D and 3D models. In this section, we focus on some of the most accessible matrix modeling techniques. These will help us build intuition on how to construct matrices in later sections when we increase the complexity of our models.

Definition 2.3: Entry-by-entry definition of an $m \times n$ matrix

Let $m, n \in \mathbb{N}$. An $m \times n$ **matrix** A is a rectangular array of real numbers with m rows and n columns. We can write the general structure of an $m \times n$ matrix A as:

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix}$$

Each of the numbers in this array is called an **entry**, **element** or **coefficient** of the matrix. Each entry has a row and column index, which identifies where in the matrix that coefficient is stored.

Notice from our definition above, that all vectors are matrices but not all matrices are vectors.

Definition 2.4: Dimensions of a Matrix

The **size** of a matrix A is given in the form $m \times n$, where m is the number of rows and n gives the number of columns of the matrix. We call m, the number listed to the left of the $\times$ symbol, the **row dimension** of the matrix. On the other hand, we call n, the number listed to the right of the $\times$ symbol, the **column dimension** of the matrix.

© Jeffrey A. Anderson

One method to create a matrix in MATLAB uses the following syntax:

```
variable_name =[ 1st row elements;
                 2nd row elements;
                 3rd row elements;
                 ...;
                 last row elements;]
```

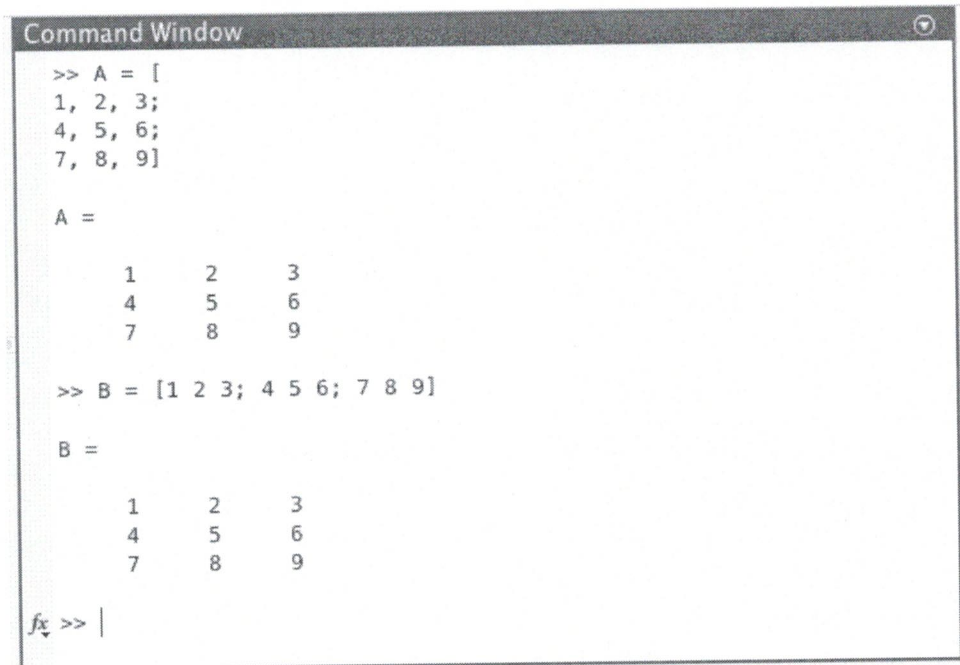
Notice that we create this matrix by typing each element, row by row, inside the square brackets [ ]. We begin this process by typing the left bracket [and then typing each element in the first row, separating the elements in this row by commas. Before starting on the next row, we type a semicolon and press Enter . We continue in this manner until we have typed all elements of the matrix. To finish our work, we type the right bracket] at the end of the last row.

EXAMPLE 2.13

Let's create a matrix $A \in \mathbb{R}^{3\times3}$, given by

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

To do so, we can use the following code:



```
Command Window
>> A = [
1, 2, 3;
4, 5, 6;
7, 8, 9]

A =

     1     2     3
     4     5     6
     7     8     9

>> B = [1 2 3; 4 5 6; 7 8 9]

B =

     1     2     3
     4     5     6
     7     8     9

fx >>
```

I always recommend that you get in the habit of formatting your code so that debugging is as easy as possible. With this in mind, I like to enter my matrices using commas, spaces, semicolons, and the Enter button, as seen in the variable assignment for matrix A above. Notice that my variable definition looks almost identical to the original matrix. You can, of course, make the same definition using just spaces and semicolons, as seen in the assignment of matrix B above.

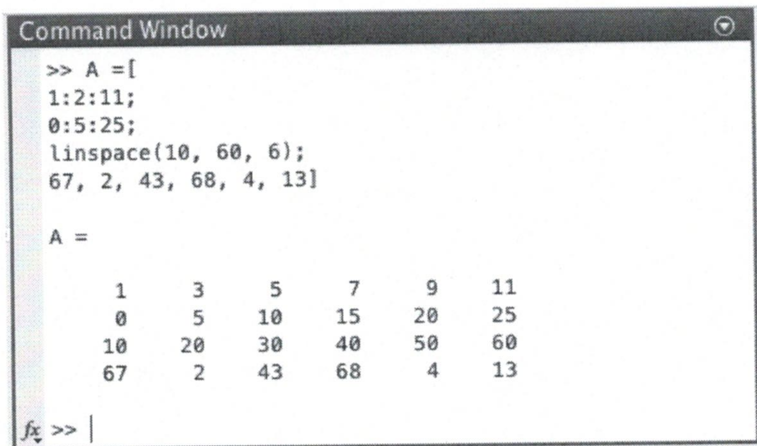
© Jeffrey A. Anderson

When defining a matrix entry by entry, we have numerous options determining how we create each entry. Below are some features to keep in mind as we work:

1. When typing row elements to define a matrix, the entered values can be numbers or vectors.

EXAMPLE 2.14

Let's take a look at how we can create a matrix by entering rows as vectors, functions with vector-valued output, and individual elements:

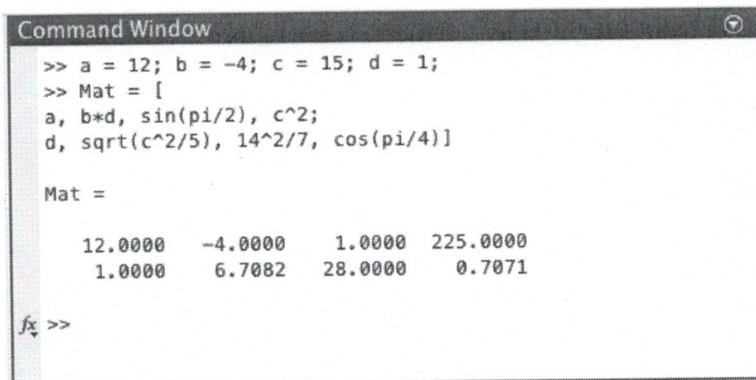


Notice that we used the `[ m : q : n]` notation from before to create row vectors but did so without explicitly type the left and right brackets. This alternative method of creating row vectors is handy when creating a so-called row-partitioned matrix, as seen in this example.

2. When typing row elements to define a matrix, the entered values can be numbers, predefine variables, or functions.

EXAMPLE 2.15

Let's take a look at how we can create a matrix by entering individual elements including stored variables, functions with scalar-valued output, and arithmetic operations on scalars:

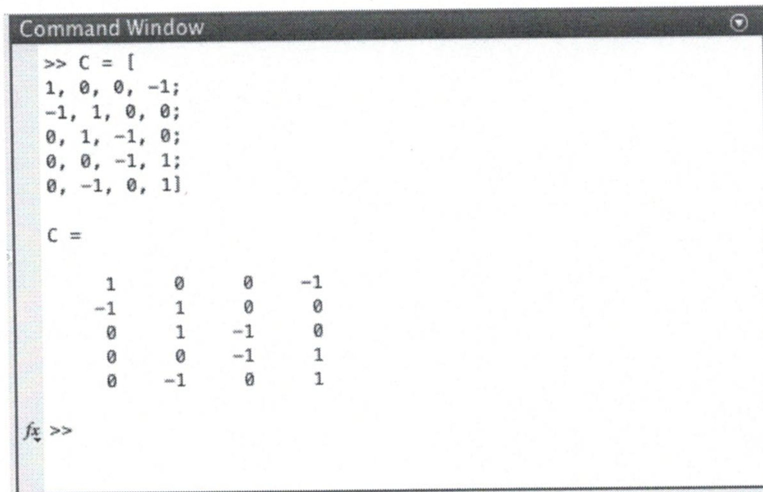


© Jeffrey A. Anderson

3. If a row contains a zero element, we must explicitly enter this value as a `0`.

EXAMPLE 2.16

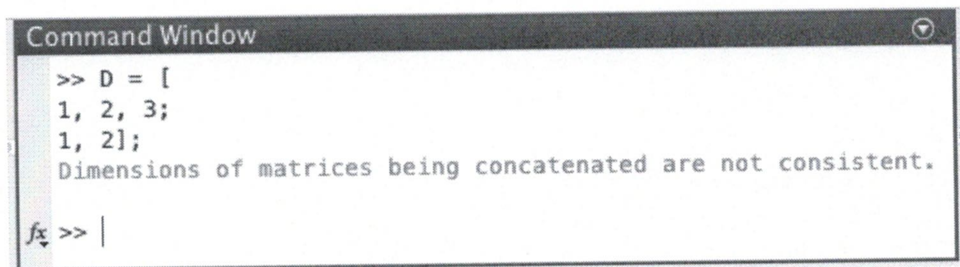
Below we see an example of a matrix $C \in \mathbb{R}^{5\times 4}$. Notice that we have to explicitly enter all zero entries by hand.



4. One very important rule is that **all rows must have the same number of elements**. MATLAB will display an error message if we attempt to define an incomplete matrix.

EXAMPLE 2.17

Let's try to create a matrix in which two different rows have unequal dimensions:



As we see, MATLAB will immediately produce an error message to indicate that our dimensions don't match. One of the reasons I recommend using both delimiters (comma and spaces for rows, semicolon and Enter for columns) is so that our code is easily checked to avoid these type of dimension errors.

© Jeffrey A. Anderson

Notes about variables in MATLAB

- MATLAB stores all numeric variables as arrays. Scalar variables are one-dimensional arrays with exactly 1 entry having one row and one column. This is also known as a 1×1 vector.
- The difference between a one-dimensional and two-dimensional array lies in the number of rows and columns allowed. One-dimensional arrays, which are used to store vectors, have either a single row or a single column. Two-dimensional arrays, which are used to store matrices) can have any number of rows or columns.
- In MATLAB, we do NOT have to explicitly define the size of our array in a variable assignment.
- Once we store a variable in MATLAB, we can edit this variable at any time. For example, we might change a scalar into a vector by adding row (or column entries) or we might change a vector into a scalar by deleting elements. We can also change a vector into a matrix by appending extra rows (or columns) or transform a matrix into a vector by deleting rows (or columns). We will get a much better handle on these options in our work below.
- I recommend reserving upper case variable names for matrices (two-dimensional arrays) and reserving lower case variable names for scalars and vectors. This practice follows conventions from your Linear Algebra courses and will come in useful when performing matrix-vector operations (as we will see later in the course).

© Jeffrey A. Anderson

The `zeros`, `ones`, and `eye` Commands

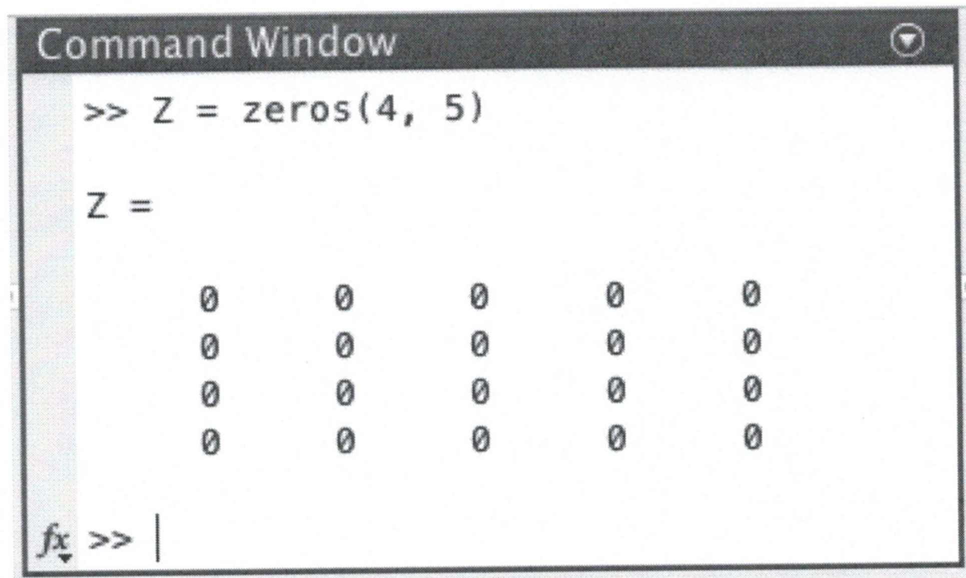
In Linear Algebra, there are a long list of special matrices with special known (predefined) entries. In this brief introduction, we focus on three special matrices: the zero matrix, the ones matrix, and the identity matrix.

EXAMPLE 2.18

The $m \times n$ zero matrix $Z \in \mathbb{R}^{m \times n}$ is a rectangular two-dimensional array with m rows and n columns in which every single entry is equal to zero. For example, let's consider a zero matrix with 4 rows and 5 columns, given by:

$$Z = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

One way to create such a matrix is to enter each zero, element by element, as shown above. However, MATLAB also has a **`zeros`** function that allows us to create the exact same two-dimensional array by simply specifying the number of rows and columns filled with zeros that we desire. Below is the code to produce our desired 4×5 zero matrix:



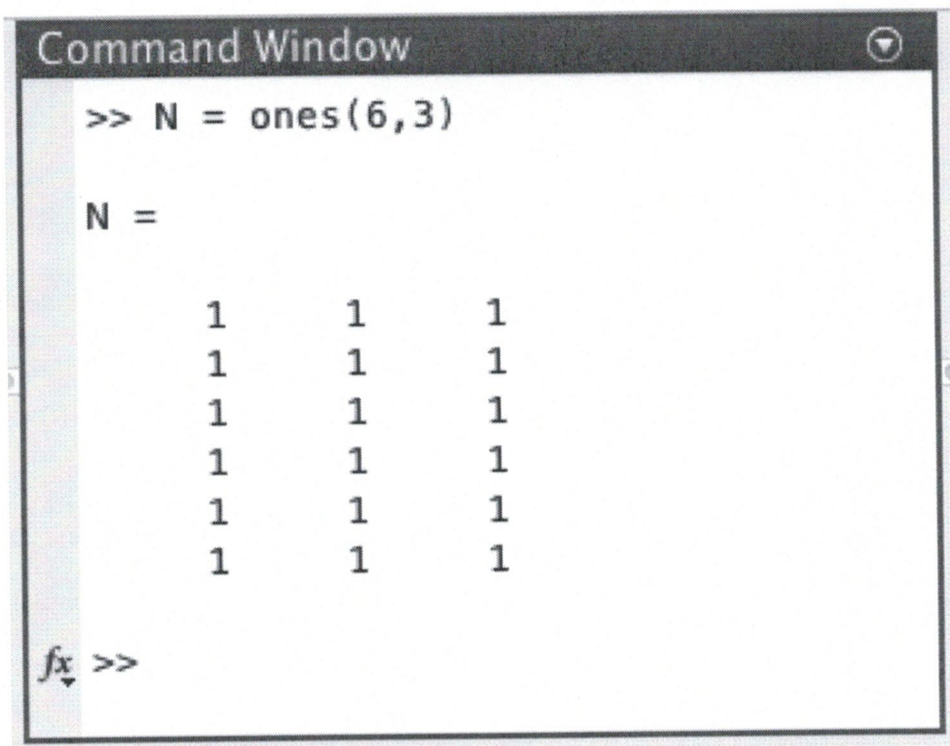
We can learn more about the **`zeros`** function by typing the command `help zeros` into the Command window.

EXAMPLE 2.19

The $m \times n$ ones matrix $N \in \mathbb{R}^{m \times n}$ is a rectangular two-dimensional array with m rows and n columns in which every single entry is equal to one. For example, let's consider a zero matrix with 6 rows and 3 columns, given by:

$$N = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

Again, MATLAB provides a special **ones** function that allows us to create the exact same two-dimensional array by simply specifying the number of rows and columns filled with ones. To produce our desired 6×3 matrix, we type:



We can learn more about the **ones** function by typing the command `help ones` into the Command window.

Before we define our last special matrix in this introduction, we need to speak a little about the anatomy of matrices. To do so, we say that the **main diagonal entries** of a square $n \times n$ matrix A are elements with equal row and column indices. In other words, we say that a_{ik} is on the main diagonal of A if $i = k$. We also state that the **main diagonal** of A is the set of all diagonal entries of A.

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} & a_{15} & a_{16} \\ a_{21} & a_{22} & a_{23} & a_{24} & a_{25} & a_{26} \\ a_{31} & a_{32} & a_{33} & a_{34} & a_{35} & a_{36} \\ a_{41} & a_{42} & a_{43} & a_{44} & a_{45} & a_{46} \\ a_{51} & a_{52} & a_{53} & a_{54} & a_{55} & a_{56} \\ a_{61} & a_{62} & a_{63} & a_{64} & a_{65} & a_{66} \end{bmatrix}$$

On the other hand, the **non-diagonal entries** of A are entries that are not on the main diagonal. Thus, a_{ik} is a non-diagonal entry if $i \neq k$.

Definition 2.5: Diagonal Matrix

Let $D \in \mathbb{R}^{n \times n}$ be a given, square matrix. We say that D is diagonal if $d_{ik} = 0$ for all $i \neq k$. Diagonal matrices take the form

$$D = \begin{bmatrix} d_{11} & 0 & \cdots & 0 \\ 0 & d_{22} & \ddots & \vdots \\ \vdots & \ddots & \ddots & 0 \\ 0 & \cdots & 0 & d_{nn} \end{bmatrix}$$

In this case, the entries on the main diagonal d_{ii} are any real numbers and are not necessarily zero.

Definition 2.6: Identity Matrix

Let $I_n \in \mathbb{R}^{n \times n}$ be $n \times n$ identity matrix given by

$$I_n = \begin{bmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & \ddots & \vdots \\ \vdots & \ddots & \ddots & 0 \\ 0 & \cdots & 0 & 1 \end{bmatrix}$$

In this case, we can define the identity matrix using the individual coefficients as follows

$$I_n(i,k) = \begin{cases} 1 & \text{if } i = k, \\ 0 & \text{if } i \neq k. \end{cases}$$

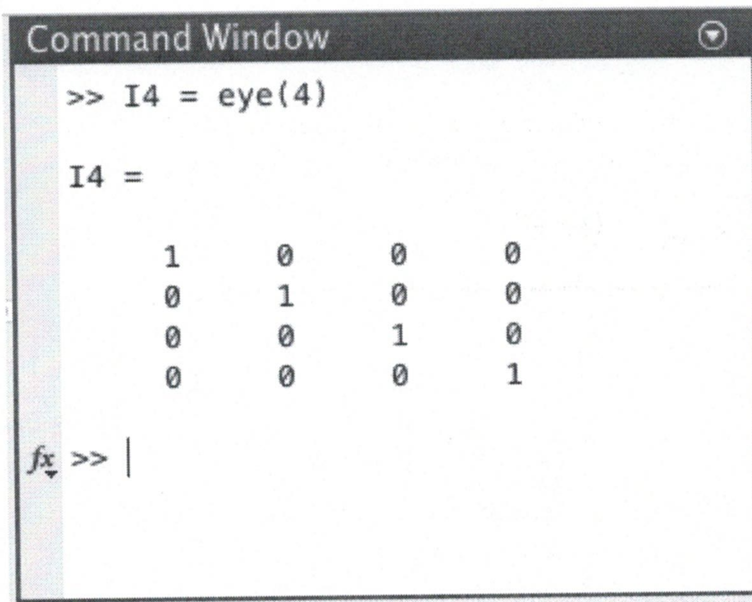
© Jeffrey A. Anderson

EXAMPLE 2.20

The $n \times n$ identity matrix $I_n \in \mathbb{R}^{n \times n}$ is a square, two-dimensional array with n rows and n columns in which the main diagonal entries are equal to one and all nondiagonal entries are zero. For example, let's consider the 4×4 identity matrix

$$I_4 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

To quickly store this matrix in MATLAB, we use the **eye** function that allows us to create identity matrices. To produce our desired 4×4 matrix, we type:



We can learn more about the **eye** function by typing the command `help eye` into the Command window.

The main reason that we work with one- and two-dimensional arrays lies in the wide array of modeling applications that lead to vectors and matrices. In fact, modern day microprocessors are designed to implement vector and matrix operations very quickly. Later in this course, we will explore how to define and execute arithmetic operations on arrays. For now, we limit our focus on how to store arrays as variables. The main take aways from this lesson is that all variable are stored as array and that we do not have to specify the size of an array before storing said array in memory.

© Jeffrey A. Anderson

The Transpose Operator

In many cases, we will want the option to switch the rows and columns of an array. This is known as *transposing* the elements in our array and can be accomplished with the transpose operator.

Definition 2.7: The Transpose of a vector

Let $\mathbf{x} \in \mathbb{R}^n$ be a column vector. We define the **transpose** of $\mathbf{x}$ as the $1 \times n$ row vector

$$\mathbf{x}^T = \begin{bmatrix} x_1 & x_2 & \cdots & x_n \end{bmatrix}$$

where the entry in the ith row of $\mathbf{x}$ becomes the entry in the ith column of $\mathbf{x}^T$.

On the other hand, let $\mathbf{y} \in \mathbb{R}^{1\times n}$ be a row vector. The **transpose** of $\mathbf{y}$ is the $n \times 1$ column vector

$$\mathbf{y}^T = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}$$

where the coefficient in the kth column of $\mathbf{y}$ becomes the entry in the kth row of $\mathbf{y}^T$.

When taking the transpose of a vector, we switch the row and column indices. This implies that the transpose of row vectors are column vectors while the transpose of column vectors are row vectors.

© Jeffrey A. Anderson

EXAMPLE 2.21

Let's create the vectors

$$\mathbf{x} = \begin{bmatrix} 0.0 & 0.5 & 1.0 & 1.5 & 2.0 \end{bmatrix}, \qquad \mathbf{y} = \begin{bmatrix} 0.0 \\ 0.5 \\ 1.0 \\ 1.5 \\ 2.0 \end{bmatrix}$$

Notice that $\mathbf{y} = \mathbf{x}^T$. In other words, to create vector $\mathbf{y}$ we simply switch the rows and columns of vector $\mathbf{x}$. To accomplish this in MATLAB, we can use the transpose operator, which is applied by typing a single quote ' immediately following the variable we want to transpose. The code below produces our desired vectors:

Command Window

```
>> x = [0:0.5:2]

x =

         0    0.5000    1.0000    1.5000    2.0000

>> y = x'

y =

         0
    0.5000
    1.0000
    1.5000
    2.0000

>> y = linspace(0, 2, 5)'

y =

         0
    0.5000
    1.0000
    1.5000
    2.0000

fx >>
```

We can learn more about the transpose operator by typing the command `help transpose` into the Command window.

© Jeffrey A. Anderson

> **Definition 2.8: Transpose of a Matrix**
>
> Let $A \in \mathbb{R}^{m \times n}$ be a matrix with real-valued coefficients. The **transpose** of A, denoted by A^T, is defined to be the matrix $A^T \in \mathbb{R}^{n \times m}$ such that the ith row of A^T is the row vector formed by transposing the ith column vector of A.

EXAMPLE 2.22

Let's create the matrix

$$A = \begin{bmatrix} 1 & 0 & -3 \\ 0 & 1 & 2 \\ 0 & 0 & 1 \end{bmatrix}, \qquad B = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -3 & 2 & 1 \end{bmatrix}$$

Notice that $B = A^T$. To accomplish this in MATLAB, we again use the transpose operator as follows:

Command Window

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

A =

     1     0    -3
     0     1     2
     0     0     1

>> B = A'

B =

     1     0     0
     0     1     0
    -3     2     1

fx >> |
```

© Jeffrey A. Anderson

Array Addressing

The fundamental building block of all arrays (whether we're storing vectors or matrices) is the individual entry. Each entry, also referred to as an **element**, of an array is defined by three unique pieces of information including

i. a row index $i \in \{1, 2, ..., m\}$

ii. a column index $k \in \{1, 2, ..., n\}$

iii. a real number given by $a_{ik} \in \mathbb{R}$

When specifying the individual entries of an array, we will identify all three of these pieces of information. We will always write the row index first and the column index second.

Definition 2.9: Entry Operator

Let $A \in \mathbb{R}^{m \times n}$. Define the map

$$\text{Entry}_{ik}(A) = A(i, k)$$

for $i, k \in \mathbb{N}$ with $1 \leq i \leq m$ and $1 \leq k \leq n$.

The row and column indices of an array element are known as the *address* of that element. In MATLAB, we can address elements individually in case we need to redefine individual elements or use individual entries to execute operations.

If $A \in \mathbb{R}^{6 \times 6}$, then the value of the element with row index 5 and column index 3 is

$$\text{Entry}_{53}(A) = A(5, 3)$$

This is equivalent to viewing the full matrix and picking out the element in the fifth row and third column

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} & a_{15} & a_{16} \\ a_{21} & a_{22} & a_{23} & a_{24} & a_{25} & a_{26} \\ a_{31} & a_{32} & a_{33} & a_{34} & a_{35} & a_{36} \\ a_{41} & a_{42} & a_{43} & a_{44} & a_{45} & a_{46} \\ a_{51} & a_{52} & \boxed{a_{53}} & a_{54} & a_{55} & a_{56} \\ a_{61} & a_{62} & a_{63} & a_{64} & a_{65} & a_{66} \end{bmatrix}$$

© Jeffrey A. Anderson

EXAMPLE 2.23

Let's create the 1×10 vector $\mathbf{x}$ given by

$$\mathbf{x} = \begin{bmatrix} 0 & 1 & 4 & 9 & 16 & 25 & 36 & 49 & 64 & 81 \end{bmatrix}.$$

In this matrix, we see that

$$\mathbf{x}(1,1) = 0, \qquad \mathbf{x}(1,4) = 9, \qquad \mathbf{x}(1,10) = 81.$$

Once we store this vector in MATLAB, each of the individual elements can be displayed, used, or redefined just like a scalar-valued variable. Let's take a look at how we can use array indexing to accomplish various tasks:

```
Command Window
>> x = [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]

x =

     0     1     4     9    16    25    36    49    64    81

>> x(1,4)

ans =

     9

>> x(1, 6) = 125

x =

     0     1     4     9    16   125    36    49    64    81

>> x(1,7) + x(1,9)

ans =

   100

>> x(1,3)^x(1,1) + sqrt(x(1,8))

ans =

     8

fx >> |
```

When dealing with one-dimensional arrays in MATLAB, we have the option of using a single scalar-valued index to address elements in our array. In instance, in the example above we find that `x(1,4)` and `x(4)` produce the same values if we type them in the command window. However, I will recommend against this practice. Instead, I will always recommend that you type both the row and column index of the specific element you are trying to access. This practice leads to an implicit reference to the size of a given one-dimensional array and helps us remember if we are working with row or column vectors.

© Jeffrey A. Anderson

EXAMPLE 2.24

Let's create the 3×4 matrix

$$A = \begin{bmatrix} 5 & 1 & 8 & 0 \\ 10 & -1 & 9 & 14 \\ 2 & 3 & -3 & 7 \end{bmatrix}.$$

In this matrix, we see that

$$A(3,1) = 2, \qquad A(1,3) = 8, \qquad A(2,4) = 14.$$

Once again, each entry can be displayed, used, or redefined just like a scalar-valued variable using our entry notation:

Command Window

```
>> A = [
5, 1, 8, 0;
10, -1, 9, 14;
2, 3, -3, 7]

A =

     5     1     8     0
    10    -1     9    14
     2     3    -3     7

>> A(2,4)

ans =

    14

>> A(1,4) = 13

A =

     5     1     8    13
    10    -1     9    14
     2     3    -3     7

>> A(2,4) - A(1,4)

ans =

     1

fx >>
```

© Jeffrey A. Anderson

Colon : notation for array addressing

Sometimes we want to refer to a range of elements in an array rather than just a single entry. We can do so using the colon : for array indexing.

Definition 2.10: Colon notation for column vectors

If $\mathbf{x} \in \mathbb{R}^{m\times 1}$, then $x(:,1) \in R^{m\times 1}$ refers to all the elements in the vector $\mathbf{x}$. If $1 \leq i \leq j \leq m$, then $\mathbf{x}(i:j,1)$ accesses elements i to j of the column vector $\mathbf{x}$. Here, we have

$$x(:,1) = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_{m-1} \\ x_m \end{bmatrix}, \qquad x(i:j,1) = \begin{bmatrix} x_i \\ \vdots \\ x_j \end{bmatrix}$$

EXAMPLE 2.25

Let's create the 7×1 column vector

$$\mathbf{x} = \begin{bmatrix} 30 & 25 & 20 & 15 & 10 & 5 & 0 \end{bmatrix}^T$$

We can use colon notation to address our array as follows:

```
Command Window
>> x = [30:-5:0]';
>> x(:,1)

ans =

    30
    25
    20
    15
    10
     5
     0

>> x(3:6,1)

ans =

    20
    15
    10
     5

fx >> |
```

© Jeffrey A. Anderson

Definition 2.11: Colon notation for row vectors

If $\mathbf{y} \in \mathbb{R}^{1\times n}$, then $y(1,:) \in \mathbb{R}^{1\times n}$ refers to all entries of the vector $\mathbf{y}$. If $1 \le k \le \ell \le n$, then $y(1,k:\ell)$ refers to entries k through ℓ of the row vector $\mathbf{y}$. We can visualize this as follows

$$y(1,:) = \begin{bmatrix} y_1 & y_2 & \cdots & y_{n-1} & y_n \end{bmatrix}, \qquad y(1,k:\ell) = \begin{bmatrix} y_k & \cdots & y_\ell \end{bmatrix}$$

EXAMPLE 2.26

Let's create the 7×1 column vector

$$\mathbf{y} = \begin{bmatrix} -800 & -600 & -400 & -200 & 0 & 200 & 400 & 600 \end{bmatrix}$$

We can use colon notation to address our array as follows:

```
Command Window
>> y = linspace(-800, 600, 8)

y =

  -800  -600  -400  -200     0   200   400   600

>> y(1,:)

ans =

  -800  -600  -400  -200     0   200   400   600

>> y(1,4:7)

ans =

  -200     0   200   400

fx >> |
```

© Jeffrey A. Anderson

Definition 2.12: Colon notation for columns of matrix

A handy way to specify individual columns of a matrix is to use **colon notation**. If $A \in \mathbb{R}^{m \times n}$ and $k \in \{1, 2, ..., n\}$ with $1 \leq i \leq j \leq m$, then

- $A(:, k) \in R^{m \times 1}$ designates all the rows in the kth column of A.
- $A(i:j, k)$ specifies rows i through j of column k.
- $A(i:j, :)$ refers to rows i through j in all columns of A

We can visualize the two vector-based options as follows:

$$A(:, k) = \begin{bmatrix} a_{1k} \\ a_{2k} \\ \vdots \\ a_{m-1,k} \\ a_{mk} \end{bmatrix}, \qquad A(i:j, k) = \begin{bmatrix} a_{ik} \\ \vdots \\ a_{jk} \end{bmatrix}$$

As for the submatrix of A, we have

$$A(i:j, :) = \begin{bmatrix} a_{i1} & a_{i2} & \cdots & a_{in} \\ \vdots & \vdots & \ddots & \vdots \\ a_{j1} & a_{j2} & \cdots & a_{jn} \end{bmatrix}$$

EXAMPLE 2.27

Let's create the 5×6 matrix given by

$$A = \begin{bmatrix} 1 & 3 & 5 & 7 & 9 & 11 \\ 2 & 4 & 6 & 8 & 10 & 12 \\ 3 & 6 & 9 & 12 & 15 & 18 \\ 4 & 8 & 12 & 16 & 20 & 24 \\ 5 & 10 & 15 & 20 & 25 & 30 \end{bmatrix}$$

Notice in this example we have $m = 5$ and $n = 6$. To explore how colon notation works with the columns of a matrix, let's set $i = 2, j = 3$, and $k = 4$. The commands and output to view the matrices associated with our last definition is shown on the next page.

© Jeffrey A. Anderson

Command Window

```
>> A =[1:2:11;
linspace(2,12,6);
3:3:18;
linspace(4,24,6);
linspace(5,25,6)]

A =

     1     3     5     7     9    11
     2     4     6     8    10    12
     3     6     9    12    15    18
     4     8    12    16    20    24
     5     9    13    17    21    25

>> A(:,4)

ans =

     7
     8
    12
    16
    17

>> A(2:3,4)

ans =

     8
    12

>> A(2:3,:)

ans =

     2     4     6     8    10    12
     3     6     9    12    15    18
```

Note: in the work above we simply displayed the results of each of the elements in our desired range of addresses. However, if we chose, we can also create new variables using these results. In other words, we can create vectors and matrices from other matrices using this colon notation.

© Jeffrey A. Anderson

Definition 2.13: Colon notation for rows of matrix

A handy way to specify individual columns of a matrix is to use **colon notation**. If $A \in \mathbb{R}^{m \times n}$ and $i \in \{1, 2, ..., m\}$ with $1 \leq k \leq \ell \leq n$, then

- $A(i, :) \in R^{1 \times n}$ designates all the columns in the ith row of A.
- $A(i, k : \ell)$ specifies columns k through ℓ of row i.
- $A(:, k : \ell)$ refers to columns k through ℓ in all rows of A

We can visualize the two vector-based options as follows:

$$A(:, k) = \begin{bmatrix} a_{i1} & a_{i2} & \cdots & a_{i,n-1} & a_{in} \end{bmatrix}, \quad A(i, k : \ell) = \begin{bmatrix} a_{ik} & \cdots & a_{i\ell} \end{bmatrix}.$$

As for the submatrix of A, we have

$$A(:, k : \ell) = \begin{bmatrix} a_{1k} & \cdots & a_{1\ell} \\ a_{2k} & \cdots & a_{2\ell} \\ \vdots & \ddots & \vdots \\ a_{mk} & \cdots & a_{m\ell} \end{bmatrix}$$

EXAMPLE 2.28

Let's return to our 5×6 matrix from Example 2.27 above. To explore how colon notation works with the rows of this matrix, let's set $i = 4, k = 2$, and $\ell = 5$. The commands and output to view the matrices associated with our last definition is shown on the next page.

© Jeffrey A. Anderson

```
Command Window
>> A

A =

     1     3     5     7     9    11
     2     4     6     8    10    12
     3     6     9    12    15    18
     4     8    12    16    20    24
     5     9    13    17    21    25

>> A(4,:)

ans =

     4     8    12    16    20    24

>> A(4,2:5)

ans =

     8    12    16    20

>> A(:,2:5)

ans =

     3     5     7     9
     4     6     8    10
     6     9    12    15
     8    12    16    20
     9    13    17    21

fx >>
```

Now guess the meaning of the code `A(2:3, 2:5)`? If so, extrapolate to define the following submatrix $A(i : j, k : \ell)$.

© Jeffrey A. Anderson

In the work above, we relied heavily on the colon to select and store a range of elements in a given array. Sometimes we may want to select only specific elements, specific rows, or specific columns to create new variables. We can do this by typing our desired element, row, or column addresses inside brackets, as shown in the example below:

```
Command Window
>> x = [66:-6:11]

x =

    66    60    54    48    42    36    30    24    18    12

>> v = x(1,[2, 5, 8:10])

v =

    60    42    24    18    12

>> A = [10:-1:4;ones(1,7); 2:2:14; zeros(1,7)]

A =

    10     9     8     7     6     5     4
     1     1     1     1     1     1     1
     2     4     6     8    10    12    14
     0     0     0     0     0     0     0

>> B = A([1, 3:4], [1,3,5:7])

B =

    10     8     6     5     4
     2     6    10    12    14
     0     0     0     0     0

fx >> |
```

© Jeffrey A. Anderson

Adding elements to existing vector-valued variables

One of the unique features of MATLAB is the ease with which we can edit and append variables. For example, we can change a vector-valued variable by adding more elements to create a longer vector or said vector can be transformed into a two-dimensional array.

EXAMPLE 2.29

Let's create a 1×5 row vector given by

$$\mathbf{x} = \begin{bmatrix} 0 & 1 & 8 & 27 & 64 \end{bmatrix}$$

Suppose we want to continue the pattern and form a longer row vector

$$\mathbf{x} = \begin{bmatrix} 0 & 1 & 8 & 27 & 64 & 125 & 216 & 343 & 512 & 729 \end{bmatrix}$$

Notice, to achieve this next vector we simply append our next highest integer cube powers onto the end of our original vector. Below we see the code that produces the same result:

```
Command Window
>> x = [0, 1, 8, 27, 64]

x =

     0     1     8    27    64

>> x(1, 6:10) = [125, 216, 343, 512, 729]

x =

     0     1     8    27    64   125   216   343   512   729

>> x = [0, 1, 8, 27, 64]; y = [125, 216, 343, 512, 729];
>> z = [x, y]

z =

     0     1     8    27    64   125   216   343   512   729

fx >> |
```

© Jeffrey A. Anderson

One feature of MATLAB related to appending vector that is useful to keep in mind is that if we append a value to a vector that is outside the predefined range of the vector, MATLAB assigns zero values to all elements between the last original element and the first new element assigned. We can see two examples of this feature in the code below:

```
Command Window
>> twos = linspace(2,2,5)

twos =

     2     2     2     2     2

>> twos(1,9) = 2

twos =

     2     2     2     2     2     0     0     0     2

>> threes(8,1)=3

threes =

     0
     0
     0
     0
     0
     0
     0
     3

fx >>
```

We can also add elements to vectors via concatenation.

```
Command Window
>> v1 = [10, -10, 8, -8]; v2 = [4:-2:-4];
>> v3 = [v1, v2]

v3 =

    10   -10     8    -8     4     2     0    -2    -4

>> v4 = [v2'; v1']

v4 =

     4
     2
     0
    -2
    -4
    10
   -10
     8
    -8

fx >>
```

In this case, we need to be very careful to properly track the use of the comma or semicolon depending on what type of vector we want to create.

© Jeffrey A. Anderson

Adding elements to existing matrix-valued variables

We can also add rows or columns to existing matrix-valued variables. Just like we did with vectors, we can 'grow' matrices by assigning new values or by appending existing variables. When adding elements to existing matrix-valued variables, we need to be very careful to check that our dimensions agree. Remember that all rows and columns of a matrix must have the same number of entries. This is especially important when adding rows and columns (since we should check that our new information matches the format of our existing variable).

```
Command Window
>> B = [0, 0, 1; 0, 1, 0; 1, 0, 0];
>> A = [eye(3), ones(3,3), B]

A =

     1     0     0     1     1     1     0     0     1
     0     1     0     1     1     1     0     1     0
     0     0     1     1     1     1     1     0     0

>> C = [eye(3); B]

C =

     1     0     0
     0     1     0
     0     0     1
     0     0     1
     0     1     0
     1     0     0

>> D = [3:3:9; 12:-4:4]

D =

     3     6     9
    12     8     4

>> D(3:4,:) = [zeros(1,3); ones(1,3)]

D =

     3     6     9
    12     8     4
     0     0     0
     1     1     1

fx >>
```

© Jeffrey A. Anderson

As always, we need to be keenly aware of the size of stored matrices when appending values. If we store a matrix A with dimensions $m \times n$ and we assign a new element with an address beyond the size of the matrix, MATLAB will automatically increase the size of the matrix to include the new element and assign all other entries the value of zero, as seen in the example below.

```
Command Window
>> clear
>> x = [1:4:17]

x =

     1     5     9    13    17

>> A = diag(x)

A =

     1     0     0     0     0
     0     5     0     0     0
     0     0     9     0     0
     0     0     0    13     0
     0     0     0     0    17

>> A(7, 8) = -4

A =

     1     0     0     0     0     0     0     0
     0     5     0     0     0     0     0     0
     0     0     9     0     0     0     0     0
     0     0     0    13     0     0     0     0
     0     0     0     0    17     0     0     0
     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0    -4

>> B(3,2) = 1

B =

     0     0
     0     0
     0     1

fx
```

© Jeffrey A. Anderson

Deleting elements from variables

We can delete individual elements or an entire range of elements in an existing vector-valued variable by eliminating these specific stored values. This reassignment is executed by using the square brackets with nothing typed between them `[]`. Using this notation, we can make vectors shorter as shown below:

```
Command Window
>> time = [0:0.5:3]

time =

         0    0.5000    1.0000    1.5000    2.0000    2.5000    3.0000

>> time(:,[2, 4, 6]) = []

time =

     0     1     2     3

>> time(1,1) = []
A null assignment can have only one non-colon index.

>> time(:,1) = []

time =

     1     2     3

fx >>
```

We can also delete ranges of elements in existing matrix-valued variables using the null `[]` assignment, as shown below:

```
Command Window
>> Fun = randi(100, 4, 5)

Fun =

    76     8    94     2    32
    76     6    13    34    53
    39    54    57    17    17
    57    78    47    80    61

>> Fun(:, 3) = []

Fun =

    76     8     2    32
    76     6    34    53
    39    54    17    17
    57    78    80    61

>> Fun(4,:) = []

Fun =

    76     8     2    32
    76     6    34    53
    39    54    17    17

fx >>
```

© Jeffrey A. Anderson

Built-in functions for handling arrays

MATLAB contains a rich library of built-in functions for managing and handling numerical arrays. Below we provide a table of some such functions that prove to be quite popular when working with arrays.

TABLE 2-2:

Built-in functions for handling arrays

Function	Description	Example

`length(A)`	Returns the number of elements in the vector A.	`>> A=[5 9 2 4];`
		`>> length(A)`
		`ans =`
		`4`

`size(A)`	Returns a row vector [m,n], where m and n are the size $m \times n$ of the array A.	`>> A=[6 1 4 0 12; 5 19 6 8 2]`
		`A =`
		`6 1 4 0 12`
		`5 19 6 8 2`
		`>> size(A)`
		`ans =`
		`2 5`

`reshape(A, m,n)`	Creates a m by n matrix from the elements of matrix A. The elements are taken column after column. Matrix A must have m times n elements.	`>> A=[5 1 6; 8 0 2]`
		`A =`
		`5 1 6`
		`8 0 2`
		`>> B = reshape(A,3,2)`
		`B =`
		`5 0`
		`8 6`
		`1 2`

`diag(v)`	When v is a vector, creates a square matrix with the elements of v in the diagonal.	`>> v=[7 4 2];`
		`>> A=diag(v)`
		`A =`
		`7 0 0`
		`0 4 0`
		`0 0 2`

`diag(A)`	When A is a matrix, creates a vector from the diagonal elements of A.	`>> A=[1 2 3; 4 5 6; 7 8 9]`
		`A =`
		`1 2 3`
		`4 5 6`
		`7 8 9`
		`>> vec=diag(A)`
		`vec =`
		`1`
		`5`
		`9`

© Jeffrey A. Anderson