MATLAB Linear Algebra

Leigh J. Little SUNY Brockport January 30, 2003

Contents

1 Introduction

MATLAB is a powerful software program that allows you to do very complex numerical simulations and view the results using a variety of graphical tools. MATLAB is very easy to learn. It has its own programming language which allows you to create detailed functions and operates much like an interactive FORTRAN or C/C++ compiler.

MATLAB has its roots in linear algebra and as such, it interprets calculations in a linear algebra context. In order to make the best use of MATLAB, a cursory understanding of linear algebra concepts is required. Fortunately, you only need to know a little to do a lot.

This handout is divided into two major parts. The first half consists of a formal, mathematical description of the minimum amount of linear algebra needed to utilize MATLAB effectively. The second half concentrates on how to create vectors and matrices in MATLAB as well as how it interprets various linear algebra calculations.

2 Linear Algebra

2.1 Scalars

Scalars are constants. π and e are 2 well-known examples of scalars. Scalars can be added, subtracted, multiplied and (most of the time) divided. Other operations, such as the trigonometric, logarithmic and exponential functions can also be applied to scalars.

Scalars can be either real or complex.

2.2 Vectors

A vector is a set of scalars arranged into a row or column. Examples of row vectors are:

$$w = \begin{pmatrix} 1 & 2 & 3 \end{pmatrix}; \qquad v = \begin{pmatrix} -3 & 4 & 0 \end{pmatrix},$$

while examples of column vectors are

$$x = \begin{pmatrix} 4 \\ 5 \\ 6 \end{pmatrix}; \qquad y = \begin{pmatrix} 3 \\ 9 \\ -2 \end{pmatrix}.$$

By convention, vectors are indicated by lower case letters. Each scalar in the vector is called an *element* or a *component* of the vector and every vector has an associated dimension (or length). The dimension of a vector is equal to the number of components. In mathematics, vectors are assumed to be *column* vectors. In general, a column vector x of length n can be written as

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

The subscripts allow each element to be referenced individually if needed.

2.2.1 The Transpose Operation

A basic operation on vectors that is frequently performed is called a *transpose*. The transpose changes a row vector to a column vector (and vice-versa). The symbol for transposition is a superscript T. Thus, if $y = x^T$, then

$$y = x^T = \begin{pmatrix} x_1 & x_2 & \cdots & x_n \end{pmatrix}.$$

If the vector elements are also complex, the *Hermitian* transpose can also be formed. The Hermitian transpose, indicated by a superscript H, takes the complex conjugate of the elements in addition to performing a transpose, thus, if $y = x^H$, then

$$y = x^H = \begin{pmatrix} \bar{x}_1 & \bar{x}_2 & \cdots & \bar{x}_n \end{pmatrix}.$$

where $\bar{x}_i$ represents the complex conjugate of element i.

2.3 Matrices

A matrix is a collection of scalars arranged in a rectangular grid. Examples of matrices are:

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}; \qquad B = \begin{pmatrix} 3 & -2 \\ 0 & 5 \\ -1 & 9 \end{pmatrix}; \qquad C = \begin{pmatrix} 5 & 2 & -1 \\ 0 & 8 & -7 \end{pmatrix}.$$

By convention, matrix variables are indicated by capital letters. Just as with vectors, matrices have an associated `size` (or `dimension`). In the above example, A is called a 3×3 matrix because it has 3 rows and 3 columns. Similarly, B is 3×2 and C is 2×3. If a matrix has the same number of rows and columns it is called a square matrix, otherwise it is rectangular.

The elements of a matrix are indexed by a double subscript according to their row and column location. Thus, in the above example, $A_{2,3} = 6$ and $A_{3,1} = 7$. As with vectors, the first element of a matrix is indexed as $A_{1,1}$ (not $A_{0,0}$ as it would be in C). In general, if A is an $n \times m$ matrix, then

$$A = \begin{pmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,m} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,m} \\ \vdots & \cdots & \cdots & \vdots \\ a_{n,1} & a_{n,2} & \cdots & a_{n,m} \end{pmatrix}$$

where $a_{i,j}$ is the element in row i and column j.

Matrices can also be viewed as collections of row or column vectors. For example, the matrix B above can be viewed as the column vectors

$$\begin{pmatrix} 3 \\ 0 \\ -1 \end{pmatrix}; \quad \begin{pmatrix} -2 \\ 5 \\ 9 \end{pmatrix}.$$

placed side-by-side or the row vectors

$$\begin{pmatrix} 3 & 2 \end{pmatrix}$$
$$\begin{pmatrix} 0 & 5 \end{pmatrix}$$
$$\begin{pmatrix} -1 & 9 \end{pmatrix}$$

stacked on top of each other.

To refer to an entire row or column of a matrix, the $*$ notation can be used:

$$\begin{aligned} b_{i,*} &= \text{row } i \text{ of the matrix } B, \\ b_{*,j} &= \text{column } j \text{ of the matrix } B. \end{aligned}$$

For the above example,

$$b_{*,2} = \begin{pmatrix} -2 \\ 5 \\ 9 \end{pmatrix}$$

and

$$b_{1,*} = \begin{pmatrix} 3 & 2 \end{pmatrix}.$$

Using the $*$ notation, the matrix B could be written as

$$B = \begin{pmatrix} b_{1,*} \\ b_{2,*} \\ b_{3,*} \end{pmatrix}$$

or

$$B = \begin{pmatrix} b_{*,1} & b_{*,2} \end{pmatrix}.$$

When a matrix is written in this way, it is said to be *partitioned* by rows or columns

Matrices can also be transposed. For some matrix A, the transpose operation A^T interchanges the rows and columns of a matrix. In element notation, the transpose is written

$$a^T_{i,j} = a_{j,i}.$$

The transposes of the matrices A, B and C above are

$$A^T = \begin{pmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{pmatrix}; \qquad B^T = \begin{pmatrix} 3 & 0 & -1 \\ -2 & 5 & 9 \end{pmatrix}; \qquad C^T = \begin{pmatrix} 5 & 0 \\ 2 & 8 \\ -1 & -7 \end{pmatrix}.$$

The *diagonal* of a matrix consists of those elements whose row and column entries are the same.

$$\text{diag}(A) = \{a_{1,1}, a_{2,2}, \ldots, a_{k,k}\}$$

where $k = \min(n, m)$. For A, B and C,

$$\begin{aligned} \text{diag}(A) &= \{1, 5, 9\} \\ \text{diag}(B) &= \{3, 5\} \\ \text{diag}(C) &= \{5, 8\}. \end{aligned}$$

2.4 Vector-Vector Operations

This section describes the mathematical interpretation of operations between 2 vectors.

Two vectors, x and y can be added or subtracted only of they have the same length and orientation (*i.e.*, are both row or column vectors). The operation is performed *componentwise*, thus $z = x + y$ means

$$\begin{pmatrix} z_1 \\ z_2 \\ \vdots \\ z_n \end{pmatrix} = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} + \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix}$$

or more compactly,

$$z_i = x_i + y_i.$$

A similar definition holds for subtraction.

Multiplication of vectors is much more complicated. In order for two vectors to be multiplied, the "inner dimensions" must match. For x and y as defined above, the vector product

$$z = x * y$$

is not defined. This is because

$$\begin{aligned} x * y &= (n \times 1) * (n \times 1). \\ &\quad\ \uparrow \quad\ \uparrow \end{aligned}$$

Hence, the inner dimensions (1 and n) do not match.

2.4.1 The Dot Product

There are two basic vector product operations that can be performed. The first is the *dot* product. The dot product of two column vectors x and y is the quantity $y^T x$ and is defined as

$$y^T x = \sum_{i=1}^{n} x_i \, y_i.$$

In other words, the dot product multiplies the vector elements componentwise and sums the result. If x and y are allowed to be complex, then the inner product is defined as $y^H x$. Also note that x and y must have the same length in order to compute their dot product.

As an example, suppose that x and y are defined as

$$x = \begin{pmatrix} 3 \\ 0 \\ -1 \end{pmatrix}; \qquad y = \begin{pmatrix} 2 \\ 1 \\ -2 \end{pmatrix}.$$

Then

$$y^T x = \begin{pmatrix} 2 & 1 & -2 \end{pmatrix} \begin{pmatrix} 3 \\ 0 \\ -1 \end{pmatrix} = 2(3) + 1(0) + (-2)(-1) = 8.$$

It is instructive to examine the dot product in more detail:

$$\begin{aligned} y^T x &= (1 \times n) * (n \times 1) \\ &\quad\ \uparrow \quad\ \uparrow \end{aligned}$$

Here, the inner dimensions match, so the operation is permitted. Further, if we discard the inner dimensions, the remaining dimensions are 1×1 (*i.e.*, a scalar).

The dot product is also called the *inner product* or the *scalar product*.

2.4.2 The Outer Product

The second type of vector product, the *outer product,* is less common. If x is a vector of length n and y is a vector of length m, then the outer product is denoted as xy^T. Again, examine this product in more detail.

$$\begin{array}{rcl} xy^T & = & (n \times 1) * (1 \times m). \\ & & \uparrow \quad \uparrow \end{array}$$

The inner dimensions (1×1) match. The remaining dimensions are $n \times m$, hence, an outer product results in a matrix. Also, note that the length of the vectors x and y does not have to be the same.

Let $A = xy^T$. Then the elements of the outer product matrix A are denoted by

$$a_{i,j} = x_i \, y_j.$$

Finally, as with the dot product, if the vectors are complex then the outer product is defined as xy^H.

For x and y defined as above,

$$xy^T = \begin{pmatrix} 6 & 2 & -6 \\ 0 & 0 & 0 \\ -2 & -1 & 2 \end{pmatrix}.$$

There are two pictures that are helpful to keep in mind when forming dot and outer products.

$$\begin{array}{rcll} \begin{pmatrix} \quad \end{pmatrix} * \begin{pmatrix} \quad \\ \quad \end{pmatrix} & = & () & = \text{ dot product} \\ \begin{pmatrix} \quad \\ \quad \end{pmatrix} * \begin{pmatrix} \quad \end{pmatrix} & = & \begin{pmatrix} \quad & \quad \\ \quad & \quad \end{pmatrix} & = \text{ outer product.} \end{array}$$

2.5 Matrix-Vector Products

Because matrices and vectors have different dimensions, they cannot be added or subtracted. They can, however, be multiplied.

Suppose x is a column vector of length m and A is an $n \times m$ matrix. Then the matrix-vector product is denoted by Ax. Examining the quantities more closely,

$$\begin{array}{rcl} Ax & = & (n \times m) * (m \times 1). \\ & & \uparrow \quad \uparrow \end{array}$$

Thus, the result of a matrix-vector product is a column vector of length n. The elements of this vector (let $z = Ax$) are defined by

$$z_i = \sum_{k=1}^{m} a_{i,k} \, x_k.$$

This more easily remembered in terms of dot products as

$$z_i = \text{dot product of } A_{i,*} \text{ and } x.$$

where $A_{i,*} =$ row i of A.

Another version of a matrix-vector product can occur then the vector is multiplied from the left. If y is assumed to be a column vector of length n, then the only way a matrix vector product can be formed is $w^T = y^T A$ since

$$\begin{array}{rcl} y^T A & = & (1 \times n) * (n \times m). \\ & & \uparrow \quad \uparrow \end{array}$$

The result of this operation is another row vector of length m. Again,

$$w_j^T = \sum_{k=1}^{n} y_k \, a_{k,j}$$

or

$$w_i^T = \text{dot product of } y \text{ and } A_{*,j}$$

where $A_{*,j} =$ column j of A.

The final version of a matrix-vector product is called the *weighted inner product* or the A-inner product and is denoted as $v = y^T Ax$. This product is easiest to interpret in terms of dot products. It can be thought of as either the matrix vector product $z = Ax$ followed by the dot product $v = y^T z$ or the matrix-vector product $w^T = y^T A$ followed by the dot product $v = w^T x$.

As examples, define x and y as in section 2.4.1 and A as in Section 2.3. Then

$$\begin{aligned}
Ax &= \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} \begin{pmatrix} 3 \\ 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ 6 \\ 12 \end{pmatrix} \\
y^T A &= \begin{pmatrix} 2 & 1 & -2 \end{pmatrix} \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} = \begin{pmatrix} -8 & -7 & -6 \end{pmatrix} \\
y^T Ax &= \begin{pmatrix} -8 & -7 & -6 \end{pmatrix} \begin{pmatrix} 0 \\ 6 \\ 12 \end{pmatrix} = -114
\end{aligned}$$

2.6 Matrix-Matrix Operations

Matrices can be added or subtracted if they have the same dimensions. Like vectors, the sum is performed componentwise. For matrices A and B,

$$\begin{aligned}
C &= A + B \\
c_{i,j} &= a_{i,j} + b_{i,j}.
\end{aligned}$$

Matrix-matrix products are most simply understood in terms of dot products. Suppose A is an $n \times k$ matrix and B is an $k \times m$ matrix. Then the matrix product $C = AB$ is a matrix of size $n \times m$ whose elements are given by

$$c_{i,j} = \sum_{l=1}^{k} a_{i,l}\, b_{l,j}.$$

In terms of dot products,

$$c_{i,j} = A_{i,*}\, B_{*,j},$$

where where $A_{i,*} =$ row i of A and $B_{*,j} =$ column j of B. As an example, take A and B as defined in Section 2.3. Then

$$\begin{aligned}
C &= \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} \begin{pmatrix} 3 & -2 \\ 0 & 5 \\ -1 & 9 \end{pmatrix} \\
&= \begin{pmatrix} 0 & 35 \\ 6 & 71 \\ 12 & 107 \end{pmatrix}.
\end{aligned}$$

The elements $c_{2,1}$ and $c_{3,2}$ are computed from

$$\begin{aligned}
c_{2,1} &= \begin{pmatrix} 4 & 5 & 6 \end{pmatrix} * \begin{pmatrix} 3 \\ 0 \\ -1 \end{pmatrix} = 4(3) + 5(0) + 6(-1) = 6 \\
c_{3,2} &= \begin{pmatrix} 7 & 8 & 9 \end{pmatrix} * \begin{pmatrix} -2 \\ 5 \\ 9 \end{pmatrix} = 7(-2) + 8(5) + 9(9) = 107.
\end{aligned}$$

Matrix-matrix products can also be viewed as collections of matrix-vector products. For example, partition the matrix B into columns as

$$B = \begin{pmatrix} b_{*,1} & b_{*,2} \end{pmatrix}.$$

Then C can be viewed as a collection of column vectors, each of which is the result of the matrix-vector product of A with a column of B.

$$\begin{aligned} C &= \begin{pmatrix} c_{*,1} & c_{*,2} \end{pmatrix} \\ &= A \begin{pmatrix} b_{*,1} & b_{*,2} \end{pmatrix} \\ &= \begin{pmatrix} Ab_{*,1} & Ab_{*,2} \end{pmatrix} \end{aligned}$$

For example, the first column of C is given by

$$\begin{aligned} c_{*,1} &= A\, b_{*,1} \\ &= \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} \begin{pmatrix} 3 \\ 0 \\ -1 \end{pmatrix} \\ &= \begin{pmatrix} 0 \\ 6 \\ 12 \end{pmatrix}. \end{aligned}$$

A similar development is possible in terms of a row based partitioning of B.

Finally, an important fact regarding the multiplication of matrices. Matrix multiplication is generally, not a commutative operation, hence,

$$A * B \neq B * A.$$

3 MATLAB Linear Algebra

As mentioned before, MATLAB has its roots in linear algebra and interprets vector and matrix sums and products in a mathematical context. Frequently however there are other matrix-based computations that need to be done during the course of a simulation that do not have a standard linear algebra interpretation. As a result, simple calculations can be confusing to figure out. The section details how MATLAB interprets the basic operations and also gives examples of calculations that do not have a mathematical interpretation. In addition, the details of creating vectors and matrices of various types are explained.

3.1 MATLAB Scalars

Scalars in MATLAB are the same as in mathematics, with one important difference; scalars are always interpreted as 1×1 matrices.

From the command line, enter:

```
>> x = 6
```

The system will respond with

```
>>
x =
    6
```

If you type:

```
>> x = 6;
```

The system will respond with

```
>>
```

This is the most useful feature of MATLAB. Anytime you place a semicolon ; after a command, the action is carried out, but the result of the computation is not echoed to the display. This is very useful when learning how to operate MATLAB because you can see how it stores data. Also, it makes debugging a function easier because you can remove the ; from commands for which you want the results displayed during computation.

Now enter:

```
>> size(x)
```

The system responds with

```
ans =
     1     1
```

The system is telling you that `x` is a matrix with 1 row and 1 column (*i.e.*, a scalar).

All of the usual mathematical operations are available For example:

```
>> x = 1
x =
   1
>> y = 2
y =
   2
>> x + y
ans =
     3
>> x - y
ans =
     -1
>> x*y
ans =
     2
>> x/y
ans =
     0.5000
>> y^5
ans =
     32
>> sqrt(y)
ans =
     1.4142
```

The trigonometric functions (sin, cos, *etc.*) as well as their inverses are also available.

There are some special constants in MATLAB

`pi` - this is a double precision approximation of π.

`i,j` - these are the imaginary units, $i, j = \sqrt{-1}$.

`Inf` - infinity (for example, the result of `1/0`).

`NaN` - stands for Not a Number (for example, the result of `0*Inf`)

MATLAB will automatically detect the presence of complex numbers and use complex arithmetic when needed.

```
>> x = -1
x =
   -1
>> y = 2 + 3i
y =
   2.0000 + 3.0000i
>> z = 4 - j
z =
   4.0000 - 1.0000j
>> sqrt(x)
x =
   0 + 1.0000i
>> y*z
ans =
     11.0000 + 10.0000i
```

3.2 Relational Operators

Relational (or logical) operators are used to make comparisons between variables. The result of a logical comparison is always either TRUE (= 1) or FALSE (= 0). The relational operators available in MATLAB are shown in Table 1. Here are some simple examples of logical comparisons:

>	greater than
>=	greater than or equal to
<	less than
<=	less than or equal to
==	logical equivalence
&	logical AND
|	logical OR
~	logical NOT (negation)

Table 1: Relational Operators in MATLAB

```
>> x = -1
x =
   -1
>> y = 3
y =
   3
>> x < y
ans =
     1
>> x > y
ans =
     0
>> x == y
ans =
     0
>> (x < 0) & (y > 0)
ans =
     1
```

The first example returns 1 (TRUE) because `x < y` is true. Conversely, the next 2 examples return 0 (FALSE) because the comparisons `x > y` and `x == y` are false. Finally, the last example is called a compound relation. It is true because both `x < 0` AND `y > 0` are true.

3.3 MATLAB Vectors

MATLAB vectors can be either row or column vectors.

3.3.1 Creating Row Vectors

Row vectors can be created in many ways. The simplest is to specify each of the components. Enter the following commands:

```
>> x = [1 3 4 -1 6]
x =
    1     3     4    -1     6
```

When you create vector (and matrices) in this manner, the start and end of the vector is indicated by the `[` and `]` characters. The command

```
>> x = [1, 3, 4, -1, 6]
```

would give the same result.

What if you wanted to create a vector that contains a sequence of equally spaced variables? Suppose you wanted to have a vector whose first element is 0, last element is 1 and each element is 0.2 units apart? You can do this in the following way:

```
>> x = (0:0.2:1)
x =
   0  0.2000  0.4000  0.6000  0.8000  1.0000
```

Similarly, you could obtain the numbers from 1 to 10 by entering:

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

When the increment between elements is 1, you can leave out the middle number.

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

3.3.2 Creating column vectors

Column vectors can also be generated in several ways. To create a column vector vector element by element, enter:

```
>> y = [0; -1; 2; 9; 11]
y =

    0
   -1
    2
    9
   11
```

Notice that this is the same as creating a row vector, except that the elements are separated by semi-colons (;) and not spaces or commas.

How can equally spaced column vectors be created? To do this, you need to use the **transpose** operation. The transpose operation is used to convert a row vector to a column vector and vice versa. In MATLAB, you can obtain a transpose in 2 ways. The first is using the **transpose** command.

```
>> x = (1:5)
x =
   1  2  3  4  5
>> y = transpose(x)
y =
   1
   2
   3
   4
   5
```

If the elements of the vector are all real, a second alternative is to use the ' (backwards apostrophe) operator:

```
>> x = (1:5)
x =
   1  2  3  4  5
>> y = x'
y =
   1
   2
```

```
   3
   4
   5
```

If some of the elements of a vector are complex, then `transpose` and ' are not the same. The ' operator corresponds to the Hermitian transpose and thus takes the complex conjugate of the elements in addition to performing a transpose. For example:

```
>> x = [i 2+i 3-2i]
x =
   0 + 1.0000i   2 + 1.0000i   3 - 2.0000i
>> y = x'
y =
   0 - 1.0000i
   2 - 1.0000i
   3 + 2.0000i
```

The easiest way to create an equally spaced column vector is to create a row vector and transpose it.

3.3.3 Subvectors and Vector Subscripts

It is frequently necessary to perform operations on only a portion of a vector, hence, it is necessary to have a method of referring to only that portion.

A `subvector` is a contiguous part of a larger vector. Consider the row vector

```
>> x = (0:0.2:1)
x =
   0  0.2000  0.4000  0.6000  0.8000  1.0000
```

The vectors

$$\begin{aligned} v &= \begin{pmatrix} 0 & 0.2000 \end{pmatrix} \\ w &= \begin{pmatrix} 0.2000 & 0.4000 & 0.6000 \end{pmatrix} \\ y &= \begin{pmatrix} 0.6000 & 0.8000 & 1.0000 \end{pmatrix} \end{aligned}$$

are all subvectors of x. The vector v can be seen to consist of elements $1-2$ of x while w consists of elements $2-4$ and y contains elements $4-6$ of x.

MATLAB provides an easy method of referring to these subvectors using the notion of a vector subscript. Recall that individual elements of a vector can be extracted by using a scalar subscript (*e.g.*, `x(5) = 0.8000`, `x(3) = 0.4000`). The scalars 5 and 3 can be replaced by other vectors that extract only the portions of x that are needed.

For example, to obtain the vector v, you can enter:

```
>> v = x(1:2)
v =
   0  0.2000
```

Similarly, to obtain w and y, you can enter

```
>> w = x(2:4)
w =
   0.2000  0.4000  0.6000
>> y = x(4:6)
y =
   0.6000  0.8000  1.0000
```

All these examples use the idea of an `array index` to obtain the required subvectors. An example of an array indexed has already been seen several times. When x was created using `x = (0:0.2:1)`, an array index was used.

An array index consists of 3 (or sometimes 2) numbers separated by colons and enclosed by parentheses. The syntax for such an index is

```
array index = (start : step : end)
```

where `start` is the first value of the array, `end` is the last value of the array and `step` is the spacing of the vector elements. If the value of `step` is 1, it does not need to be explicitly given, hence, `(1:1:10)` is the same as `(1:10)`.

Here are some examples:

`x = (1:100)` => a vector from 1 to 100 with spacing 1.

`y = (1:2:100)` => a vector from 1 to 100 with spacing 2.

`z = (0:.1:10)` => a vector from 0 to 10 in with spacing 0.1.

Suppose you want the elements of the array index to go from 10 to 1. What happens if you enter `x = (10:1)`?

```
>> x = (10:1)
x =
   Empty matrix: 1-by-0
```

This operation failed. This is because the step has been implicitly given as 1, but a step of -1 is actually needed. The correct vector can be obtained by entering the step explicitly:

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

This example illustrates an important fact: If `step` is positive, then `start` must be smaller than `end`, while if `step` is negative, `start` must be larger than `end`.

We can combine these examples with extraction of elements from a vector. Let `x = (1:16)`. Then we have the following:

```
>> x = (1:16)
x =
   1   2   3   4   5   6   7   8   9   10   11   12   13   14   15   16

>> w = x(1:2:16)
w =
   1   3   5   7   9   11   13   15

>> y = x(2:2:11)
y =
   2   4   6   8   10

>> z = x(16:-4:3)
z =
   16   12   8   4
```

Note that `step` does not have to be chosen in a manner that exactly coincides with `end`. In the example for z above, the commands `z = x(16:-4:2)` and `z = x(16:-4:1)` would produce the same result.

The elements you select do not have to be consecutive or have an underlying pattern. For example, define the vectors `x`, `ii` and `jj` as:

```
>> x = (0:.2:1)
x =
   0   0.2000   0.4000   0.6000   0.8000   1.0000

>> ii = [6 3 2 5 4 1]
ii =
    6   3   2   5   4   1

>> jj = [5 5 5 1 1]
jj =
    5   5   5   1   1
```

The command `x(ii)` will extract elements of `x` according the the indices given in the vector `ii` (and similarly for `x(jj)`).

```
>> x(ii)
ans =
     1.0000  0.4000  0.2000 0.8000 0.6000 0

>> x(jj)
ans =
     0.8000 0.8000 0.8000  0  0
```

The operation of changing the ordering of the elements in a vector is called a *permutation*.

Finally, a special array index vector is available. For any vector, the array index `(:)` refers to the entire vector. This is used mainly in matrices.

3.3.4 The find command

The `find` command is a powerful command that can be used to locate elements in a vector that satisfy a certain criteria. This is best explained by an example.

```
>> x = [-2 0 3 5 -6 0 7 -9 0 10 -3 -2 4 5 ];

>> ii = find(x < 0)
ii =
    1  5  8  11  12
>> x(ii)
ans =
     -2  -6  -9  -3  -2
```

In this example, the `find` command has located the indices of all elements of `x` that satisfy the relation `x < 0`. Some other examples are:

```
>> jj = find(x == 0)
jj =
    2  6  9
>> x(jj)
ans =
     0  0  0
>> ll = find(abs(x) < 4)
ll =
    1  2  3  6  9  11  12
>> x(ll)
ans =
     -2  0  3  0  0  -3  -2
```

The first example returns the indices of all elements of `x` that are equal to 0 while the second returns the indices of those elements whose absolute value is less than 4.

3.4 MATLAB Matrices

As in mathematics, matrices in MATLAB are generalizations of vectors and can be viewed as a collection of scalars, row vectors or column vectors.

3.4.1 Creating Matrices

As with vectors, matrices can be created by explicitly listing the elements. You enter the elements row by row and start a new row using the `;` character.

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

```
   1  2  3
   4  5  6
   7  8  9
>> B = [ 3 -2; 0 5; -1 9]
B =
   3 -2
   0  5
  -1  9
>> C = [5 2 -1; 0 8 7]
C =
   5  2 -1
   0  8  7
```

In order to successfully create a matrix in this manner, the number of elements in each row must be the same. For example, the following command gives the error shown:

```
>> D = [1 2 3; 4 5]
??? Error using ==> vertcat
All rows in the bracketed expression must have the same
number of columns.
```

The more usual method of creating matrices in MATLAB is by building a large matrix from either vectors or smaller matrices.

3.4.2 Submatrices

A `submatrix` is a contiguous portion of a matrix. For example, the matrix

$$D = \begin{pmatrix} 5 & 6 \\ 8 & 9 \end{pmatrix}$$

is a 2×2 matrix that can be seen to correspond to the lower portion of the matrix A. Similarly, the matrix

$$E = \begin{pmatrix} 2 \\ 5 \\ 8 \end{pmatrix}$$

is a 3×1 matrix (or vector) corresponding to the second column of A.

You can easily extract portions of a matrix using the array index notation introduced for vectors. The matrix D is the same as rows $2-3$ and columns $2-3$ of matrix A, so we can obtain D by entering

```
>> D = A(2:3,2:3)
D =
    5  6
    8  9
```

Similarly, we can obtain E by entering

```
>> E = A(:,2)
E =
   2
   5
   8
```

From these examples, it can be seen that the same principles regarding vector subscripts and array index notation introduced for vectors also apply to matrices, although the number of possibilities is much greater since a matrix has 2 subscripts.

The second example is interesting in that it illustsrates the use of the whole vector array index operator (`:`). The command `A(:,2)` means 'all rows and column 2'. Similarly, the command `A(3,:)` means 'row 3 and all columns' and results in

```
>> F = A(3,:)
F =
   7  8  9
```

The methods for extracting submatrices can be fairly complex. For example:

```
>> ii = [1 3]
ii =
    1  3

>> jj = [2 3]
jj =
    2  3

>> A(ii,jj)
ans =
      2  3
      8  9
```

This example can be viewed as 2 separate steps. First, rows 1 and 3 are extracted (from the `ii` variable), then from these rows, columns 2 and 3 are extracted (from the `jj` variable). Of course, this is equivalent to extracting columns 2 and 3 then rows 1 and 3.

As with vectors, the extracted elements do not need to have a pattern to them.

```
>> ii = [ 3 1 ]
ii =
     3  1

>> jj = [3 1 2]
jj =
    3 1 2

>> A(ii,jj) =
ans =
      9  7  8
      3  1  2
```

3.4.3 Creating Matrices from Submatrices

The idea of submatrices can be used in reverse to create large matrices. Suppose you wanted to recreate the matrix A from the row vectors:

```
>> x1 = [1 2 3];
>> x2 = [4 5 6];
>> x3 = [7 8 9];
```

From the vectors `x1`,`x2` and `x3`, it can be seen that to obtain A, these vectors need to be stacked in the order they are given. This can be accomplished as follows.

```
>> A = [x1; x2; x3]
```

Recall that with in the `[` and `]`, a semicolon (`;`) starts a new row.

For the column oriented version of the same problem, we can recreate A from the column vectors

```
>> y1 = [1; 4; 7];
>> y2 = [2; 5; 8];
>> y3 = [3; 6; 9];
```

by typing

```
>> A = [y1 y2 y3]
```

Since the vectors are separated by spaces within the `[` and `]`, these vectors are placed next to each other.

As a more elaborate example, suppose you wanted to recreate A from the components:

$$D = \begin{pmatrix} 5 & 6 \\ 8 & 9 \end{pmatrix}; \quad w = \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix}; \quad y = \begin{pmatrix} 4 \\ 7 \end{pmatrix}.$$

Comparing the components with A, you can see that D forms the portion of lower right-hand portion of A, the transpose of w forms the first row of A and y makes up the remaining portion of column 1. To form A, all we need to keep in mind is that the matrix needs to be created row by row from top to bottom. The following sequence of commands will form A.

```
>> D  = [5 6; 8 9];
>> w = [1; 2; 3];
>> y = [4; 7];
>> A = [w'; y D];
```

The last expression within the square brackets reads 'the first row of `A` is the transpose of `w` and the next rows are the vector `y` appended with the matrix `D`.'

Whenever you create matrices using this technique, you must be sure that each row has the same number of elements, otherwise, you will get an error. For example, if you forgot the transpose operator on the `w`, you would see

```
>> A = [w; y D];
??? Error using ==> vertcat
All rows in the bracketed expression must have the same
number of columns.
```

3.4.4 Diagonal Matrices

Diagonal matrices are matrices that seem to have their elements aligned along the diagonals of the matrix. For example,

$$G = \begin{pmatrix} 1 & 3 & 0 & 0 & 0 \\ 2 & 1 & 3 & 0 & 0 \\ 0 & 2 & 1 & 3 & 0 \\ 0 & 0 & 2 & 1 & 3 \\ 0 & 0 & 0 & 2 & 1 \end{pmatrix}$$

is a *tridiagonal* matrix. The `main diagonal` of a matrix consists of those elements where the row and column are equal. For matrix G, the main diagonal corresponds to the locations of the 1's. There are 2 other diagonals in this matrix. The diagonal corresponding to the 2's is called the subdiagonal, since it is below and once-removed from the main diagonal. Similarly, the diagonal corresponding to the 3's is the superdiagonal.

This matrix can be easily created using the MATLAB command `diag`. For this example through, it is helpful to introduce the `ones` command. This command creates a matrix of all ones. For example:

```
>> x = ones(1,3)
x = 1  1  1

>> y = ones(3,1)
y =
    1
    1
    1
>> B = ones(2,3)
B =
   1  1  1
   1  1  1
```

The `diag` command takes a vector and places the components of the vector along a diagonal that you specify. The size of the resulting matrix depends on which diagonal the vector is placed. The matrix G can be created by entering the commands:

```
>> e4 = ones(4,1);
>> e5 = ones(5,1);
>> G = diag(e5) + 2*diag(e4,-1) + 3*diag(e4,1)
ans =

     1     3     0     0     0
     2     1     3     0     0
     0     2     1     3     0
     0     0     2     1     3
     0     0     0     2     1
```

The first part of the last command, `diag(e5)` creates an all zero 5×5 matrix with 1's on the main diagonal. The second part of the command, `2*diag(e4,-1)` takes the matrix just created and places 2's on the subdiagonal. The final part of the command `3*diag(e4,1)` takes the previous matrix and places 3's on the superdiagonal.

If the input to `diag` is a matrix instead of a vector, the output is a vector consisting of the matrix elements along the diagonal specified. For example,

```
>> diag(G)
ans =
     1
     1
     1
     1
     1

>> diag(G,1)
ans =
     3
     3
     3
     3

>> diag(G,-1)
ans =
     2
     2
     2
     2

>> diag(G,2)
ans =
     0
     0
     0
```

In the last example, all the elements on the diagonal twice removed above the main diagonal of G are all zero. This diagonal also has only 3 elements.

3.5 MATLAB Vector-Vector Operations

Just like with scalars, vectors can be added, subtracted and multiplied, however there are some restrictions that govern when and how these operations can be performed.

For this section, the following definitions are assumed:

```
>> x1 = [1 2 3]
x1 =
    1  2  3
```

```
>> x2 = [2 4 6]
x2 =
    2  4  6

>> y1 = [-1; 3; -5];
y1 =
    -1
     3
    -5

>> y2 = [2; -4; 6];
y2 =
     2
    -4
     6
```

We can add or subtract `x1` and `x2` in the usual way:

```
>> x3 = x1 + x2
x3 =
    3  6  9

>> x4 = x1 - x2
x4 =
    -1  -2  -3
```

The reason we can do this is because `x1` and `x2` have the same dimensions. If you try to add `x1` and `y1`, you get an error:

```
>> a1 = x1 + y1
??? Error using ==> +
Matrix dimensions must agree.
```

Even though `x1` and `y1` are vectors of length 3, they still have different dimensions because `x1` is a row vector and `y1` is a column vector. We can, however, add `y1` and `y2`:

```
>> y3 = y1 + y2
y3 =
     1
    -1
     1

>> y4 = y1 - y2
y4 =
     -3
      7
    -11
```

Also, we can add `x1` and `y1` if one of these is transposed prior to adding:

```
>> y5 = y1 + transpose(x1)
y5 =
     0
     5
    -2
```

Finally, since `x1` contains only real numbers, the above statement is equivalent to:

```
>> y5 = y1 + x1'
y5 =
     0
     5
    -2
```

Multiplication and division of vectors is much more complicated. The notion of vector (and matrix) multiplication usually introduced in a course in linear algebra states that 2 vectors (or matrices) can be multiplied only if their inner dimensions are the same. To see this, consider trying to form the product `y3 = y1 * y2`. If you try to do this, an error will be generated:

```
>> y3 = y1 * y2
??? Error using ==> *
Inner dimensions must agree.
```

To see why this will not work, write out the dimensions of the quantities being multiplied:

$$y1 * y2 => (3 \times 1) * (3 \times 1).$$

It can be seen that the 'inner dimensions' are 1 and 3. Since these are not equal, this product cannot be computed.

We can get a meaningful product by transposing either `y1` or `y2`. This leads to the notion of the dot and outer products.

3.5.1 The Dot Product

The `dot product` of 2 vectors may be familiar to you if you have taken Calculus or Linear Algebra. To perform a dot product, multiply corresponding vector elements and sum the results:

$$\text{dot}(y_1, y_2) = -1(2) + 3(-4) + (-5)(6) = -44.$$

Note that the result of a dot product is a scalar.

To get MATLAB to perform a dot product, `y1` needs to be transposed and then multiplied with `y2`:

```
>> y1' * y2
ans =
      -44
```

Let examine this in more detail. You can always tell the result of a vector product by looking at the dimensions. If the computation is examined, we see that

$$y1' * y2 => (1 \times 3) * (3 \times 1).$$

The inner dimensions 'cancel out' and the result is a 1×1 matrix (*i.e.*, a scalar).

3.5.2 The Outer Product

If we were to transpose `y2` instead of `y1`, a meaningful computation can be obtained, but the result would be a matrix and not a vector. To see this, look at the dimensions:

$$y1 * y2' => (3 \times 1) * (1 \times 3).$$

Here, the inner dimensions cancel out and the result is a 3×3 matrix. This is called an `outer product`. This type of vector product is rarely encountered, so we will only show the result for now:

```
>> y1 * y2'
ans =
      -2     4    -6
       6   -12    18
     -10    20   -30
```

3.5.3 Component Multiplication and Division

Component multiplication (also called a Haddamard product) is not usually encountered in a mathematical context, but is extremely useful in MATLAB. The result of a component multiplication of 2 vectors is another vector of the same length whose components are the product of corresponding vector elements. In order to perform this operation, MATLAB needs a special symbol to indicate that you intend for the multiplication to be done componentwise rather than interpreted as a dot or outer product. This is done with the . (period) operator. For example, we can perform component multiplication on `x1` and `x2` by entering:

```
>> x1 .* x2
ans =
     2  8  18
```

The resulting vector has the same orientation as `x1` and `x2` (*i.e.*, a row vector). Similarly, the component product of `y1` and `y2` can also be computed:

```
>> y1 .* y2
ans =
     -2
    -12
    -30
```

Note that if we try to compute the component product of `x1` and `y2` an error is still generated:

```
>> x1 .* y2
??? Error using ==> .*
Matrix dimensions must agree.
```

This can be computed by transposing either `x1` or `y2` during the multiplication:

```
>> x1 .* y2'
ans =
     2    -8    18

>> x1' .* y2
ans =
     2
    -8
    18
```

The same elements are obtained, but the orientation of the resulting vector is different in each case.

There is no mathematical interpretation of vector division, but the component idea can be used to divide each element of one vector by another. MATLAB provides 2 divide operations. The `./` operator is the usual division operation that you are familiar with (this is called right division). MATLAB also has a `.\` operator. This is called a left division and is far less frequently encountered. Here are 3 examples:

```
>> x2 ./ x1
ans =
     2  2  2

>> x2 .\ x1
ans =
     0.5000  0.5000  0.5000

>> x1 ./ x2
ans =
     0.5000  0.5000  0.5000
```

The first example divides each element in `x2` by the corresponding element of `x1`. The second case is an example of left division. In this case, the elements of `x1` are divided by those of `x2`. This is far less familiar and is likely to generate many errors or unknown results unless you are very comfortable with it. If you want to divide each element of `x1` by the corresponding element of `x2`, it is better to use the third example. The last two examples are equivalent, but the second is more clear.

3.6 MATLAB Matrix-Vector Operations

The only possible operation with a matrix and a vector that can be performed is multiplication. As with vectors, in order for a matrix and a vector to be multiplied, the inner dimensions must match. Define the following quantities:

```
>> A = [1 2 3; 4 5 6; 7 8 9];
>> x3 = [-1; 2; -3];
>> y3 = [4 -5 6];
```

Examine the product `A * x3`. This can be done because the inner dimensions match.

$$A * x3 => (3 \times 3) * (3 \times 1).$$

After canceling out the inner dimensions, the result will be a 3×1 matrix (*i.e.*, a column vector).

```
>> z = A * x3;
z =
   -6
  -12
  -18
```

Note that the operation `x3 * A` cannot be performed because the dimensions do not match.

This operation is very common, so some explanation of this result is needed. The first component of `z` is -6. This can be seen to be the dot product of `x3` with the first row of `A`. In equation form,

```
z(1) = dot( A(1,:), x3) = 1*(-1) + 2*2 + 3*(-3) = -6
```

where the whole array index has been used to extract the whole first row of `A`. Similarly,

```
z(2) = dot( A(2,:), x3) = 4*(-1) + 5*2 + 6*(-3) = -12
z(3) = dot( A(3,:), x3) = 7*(-1) + 8*2 + 9*(-3) = -18
```

Although less common, products of the form `y3 * A` also occur. Examining the dimensions, we see that

$$y3 * A => (1 \times 3) * (3 \times 3).$$

The inner dimensions match and the result is a 1×3 matrix (*i.e.*, a row vector).

```
>> v = y3 * A;
v =
   26  31  36
```

Here, you can see that the first component of `v` is the dot product of `y3` with the first column of `A`. In equation form,

```
v(1) = dot( y3, A(:,1)) = 4*1 + (-5)*4 + 6*7 = 26
```

where the whole array index has been used to extract the whole first column of `A`. Similarly,

```
v(2) = dot( y3, A(:,2)) = 4*2 + (-5)*5 + 6*6 = 31
v(3) = dot( y3, A(:,3)) = 4*3 + (-5)*6 + 6*9 = 36
```

3.7 MATLAB Matrix-Matrix Operations

As with vectors, 2 matrices can be added or subtracted only if they have the same number of rows or columns. Define the following quantities:

```
>> A = [1 2 3; 4 5 6; 7 8 9];
>> B = [1 -2 3; -4 5 -6; 7 -8 9];
>> A + B
ans =
     2   0   6
     0  10   0
    14   0  18
```

To add 2 matrices, simply add corresponding matrix elements.

Multiplying matrices is similar to forming matrix-vector products. As both of these are 3×3 matrices, we can form the matrix products `A * B` and `B * A`.

```
>> A * B
C =
   14   -16    18
   26   -31    36
   38   -46    54

>> D = B * A
D =
   14    16    18
  -26   -31   -36
   38    46    54
```

An important fact to note is that matrix multiplication is *not* commutative (*i.e.*, `A * B` is not the same as `B * A`).

There are several ways to view a matrix-matrix product. Perhaps the most clear way is to generalize the idea of a matrix-vector product introduced in the previous section. Consider the product `A * B`. The matrix `B` can be thought of as a collection of column vectors.

```
>> B = [b1 b2 b3];
```

Where `b1`, `b2` and `b3` are the column vectors

$$b1 = \begin{pmatrix} 1 \\ -4 \\ 7 \end{pmatrix}; \quad b2 = \begin{pmatrix} -2 \\ 5 \\ -8 \end{pmatrix}; \quad b3 = \begin{pmatrix} 3 \\ -6 \\ 9 \end{pmatrix}.$$

Thus, the first column of the matrix-matrix product `A * B` is, in equation form, `A * b1`. Similarly, the second and third columns of `A * B` are `A * b2` and `A * b3` respectively. This same idea can be applied to the matrix product `B * A`.

In general, to generate the element in row `i` and column `j` of a matrix-matrix product, form the dot product of row `i` of the first matrix with column `j` of the second matrix.

```
element(i,j) of A * B = dot ( row i of A, column j of B)
                      = dot ( A(i,:) , B(:,j) )
```

4 Lab Work

1) Define the following variables:

$$a = 6;\ \ b = 2;\ \ c = -3;\ \ d = -2;\ \ e = 1 + 2i;\ \ f = -1 + 2i.$$

Perform the following computations in MATLAB and verify the results by hand.

a) `a^2`

b) `b^a`

c) `c^2 - d^3`

d) `a < b`

e) `c == d`

f) `e * f`

g) `e / f`

2) For f as defined in Problem 1), answer the following

a) What is `f'`?

b) What is `conj(f)`?

c) Explain why a) and b) are the same.

d) Are `e` and `f` complex conjugates? Why or why not?

3) Give the commands that will create the following vectors:

$$w = \begin{pmatrix} 1 \\ 2 \\ -3 \end{pmatrix};\quad x = \begin{pmatrix} 0 & -1 & -4 \end{pmatrix};\quad y = \begin{pmatrix} i \\ 2i \\ -i \end{pmatrix};\quad z = \begin{pmatrix} 2 \\ -4 \\ 6 \end{pmatrix}.$$

4) Give the command that will create a row vector from -1 to 1 in steps of 0.1 and store it in the variable x. What are the dimensions of x (hint: use the `size` command)? Give the output from the following commands and explain the results. If you encounter an error, explain why the error occurs.

a) `x(2)`

b) `x(1:11)`

c) `x(0:22)`

d) `x(1:2:21)`

e) `x(21:-1:10)`

f) `x(1:.1:2)`

Is the last command safe to use?

5) These next 2 problems illustrate the use of the `find` command. Let x be as in Problem 4).

a) What is the output of `ii = find(x < 0)`?

b) What is the output of `x(ii)`?

c) Based on these results, what does this `find` command do?

6) Give the command that will create a row vector from -10 to 10 in steps of 1 and store it in the variable y.

a) What is the output of `ii = find(mod(y,2) == 0)`?

b) What is the output of `y(ii)`?

c) Based on these results, what does this `find` command do?

7) Create the matrix

$$A = \begin{pmatrix} 2 & 4 & 6 \\ 3 & 6 & 9 \\ 4 & 8 & 12 \end{pmatrix}.$$

Give the output of each command and explain the results. If you encounter an error, explain why the error occurs.

a) `A(3,1)`
b) `A(2,3)`
c) `A(4,2)`
d) `A(1,0)`
e) `A(:,2)`
f) `A(:,1:2:3)`
g) `A(1,:)`
h) `A(1:2,2:3)`
i) `A'`

Define `ii = [3 1]` and `jj = [3 2]`. Explain the output of

j) `A(:,jj)`
k) `A(ii,:)`
l) `A(ii,jj)`

8) Give the commands that will create the following quantities:

$$D = \begin{pmatrix} 2 & 4 \\ 3 & 6 \end{pmatrix}; \quad e = \begin{pmatrix} 6 & 9 \end{pmatrix}; \quad f = \begin{pmatrix} 4 \\ 8 \\ 12 \end{pmatrix}.$$

Now give the commands that will build the matrix A from Problem 7).

9) Give the commands that will create the matrix

$$\begin{pmatrix} 5 & -4 & -3 & -2 & -1 \\ 4 & 5 & -4 & -3 & -2 \\ 3 & 4 & 5 & -4 & -3 \\ 2 & 3 & 4 & 5 & -4 \\ 1 & 2 & 3 & 4 & 5 \end{pmatrix}.$$

10) The `find` command can also work for matrices. Let A be as defined in Problem 7).

a) What is the output from the command `ii = find(A == 4)`? Relate this output the the matrix `A`.
b) What is the output from the command `[ii,jj] = find(A == 4)`? Relate this output the the matrix `A`.
c) Although a) and b) find the same element, they refer to it in different manners. Which is more helpful?

For the remaining problems, explain the output from each of the commands below:

d) `[ii,jj] = find(A < 6)`
e) `[ii,jj] = find(A > 4)`
f) `ii = find(A <= 3)`

11) Create the following vectors in MATLAB:

$$w = \begin{pmatrix} 4 \\ 2 \\ 1 \end{pmatrix}; \quad x = \begin{pmatrix} 1 \\ 0 \\ 1 \end{pmatrix}; \quad y = \begin{pmatrix} 2 \\ -3 \\ 1 \end{pmatrix}; \quad z = \begin{pmatrix} i \\ -i \\ 3i \end{pmatrix}.$$

Can the following operations be performed? If yes, give the result and verify by hand computation. If not, completely explain why not.

a) `x * w`
b) `w * w`
c) `w' * w`
d) `y' * z`
e) `y * z`
f) `y * y'`
g) `y .* w`
h) `y .* w'`
i) `x .* z`
j) `x ./ y`
k) `w ./ y`
l) `z ./ w`

12) Create the following quantities in MATLAB:

$$A = \begin{pmatrix} -1 & 3 & 4 \\ 6 & -2 & -9 \\ 3 & 1 & 0 \end{pmatrix}; \quad x = \begin{pmatrix} 3 \\ -2 \\ 0 \end{pmatrix}; \quad y = \begin{pmatrix} -3 & 4 & 1 \end{pmatrix}.$$

Can the following operations be performed? If yes, give the result and verify by hand computation. If not, completely explain why not.

a) `A * x`
b) `x * A`
c) `A * y`
d) `y * A`
e) `A' * x`
f) `A * y'`
g) `y * A * x`

13) Create the following matrices in MATLAB:

$$A = \begin{pmatrix} 0 & 0 & 1 \\ 1 & -1 & -2 \\ -1 & 6 & 3 \end{pmatrix}; \quad B = \begin{pmatrix} 2 & -1 \\ 0 & 3 \\ -2 & 2 \end{pmatrix}; \quad C = \begin{pmatrix} 0 & 2 & -1 \\ -3 & 0 & 1 \end{pmatrix}.$$

Can the following operations be performed? If yes, give the result and verify by hand computation. If not, completely explain why not.

a) `A * B`
b) `A * C`
c) `B * A`
d) `C * A`
e) `B * A * C`
f) `C * A * B`

g) `A .* A`

h) `A .* C`

i) `A ./ B`

j) `A ./ A`

14) Let x be defined as in Problem 12). Perform the following operations and verify the results:

a) `y1 = x.^3`

b) `y2 = sqrt(x.*x)`

c) `y3 = cos(x)`

d) `y4 = tan(x)`

For the trigonometric functions, is x assumed to be in degrees or radians?

15) Let A be defined as in Problem 13). Perform the following calculations: and verify by hand:

a) `C1 = A.^2`

b) `C2 = 2*A.^2 - A`

c) `C3 = A - ones(3)`

d) `C4 = sqrt(A.^2)`

EXTRA CREDIT: Does the command

```
C5 = A + ones(size(A))
```

generate an error? If not, explain what this command does.