



Matrix Representation, Operations and Vectorization

Recall that MATLAB is an abbreviation for Matrix Laboratory. Everything in MATLAB is a matrix, be it a scalar, variable or an image. The primary motivation behind building an entire piece of software for matrices was the fact that matrices are very important in engineering. Almost everyone has seen matrices in mathematics, where they are used for linear transformations, but matrices are also used in a wide range of applications in engineering where they represent a variety of different notions. In this section, we will see different representations of matrices, their applications, and learn different operations on them. Another important aspect is the trick known as vectorization. Since we will be dealing with big matrices, we will see that it is important to avoid loops over matrices using this trick.

Matrix Representation

In this section, we'll look at ways to represent matrices.

Conventional Sense: Matrices

Matrices are widely used in mathematics, especially in Linear Algebra. Recall that a matrix is a two-dimensional set of elements. The most common usage is to represent a transformation on some space. For example, consider a 2D vector

$$x = \begin{bmatrix} 2 \\ 3 \end{bmatrix}.$$

Let us take the rotation matrix A as

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

Now define y as

$$y = Ax = \begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix} \begin{bmatrix} 2 \\ 3 \end{bmatrix} = \begin{bmatrix} 3 \\ -2 \end{bmatrix}$$

which is the 90° rotated version of x .

Data Sense: Arrays

There is a whole new meaning of matrices in engineering where they can also be used to represent data. Such matrices are called arrays. For example, consider an input signal to an electric motor shown in Figure 2-1.

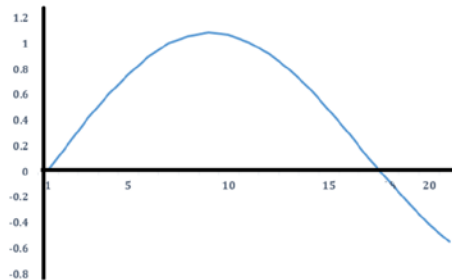


Figure 2-1. *Input signal to an electric motor*

The same signal can be represented as the following matrix:

```
x=[0.0048 0.21 0.41 0.597 0.76 0.89 1.00 1.059 1.085 1.07 1.01 0.92 0.80 0.65 0.48 0.29 0.10
-0.0773565 -0.25 -0.41 -0.55];
```

An image is another example of such data representation. A color image is a three dimensional matrix. For example, the small 4×4 pixel image in Figure 2-2 can be represented as the following matrix

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

where 0 1 2 3 denotes black, red, green and blue respectively.

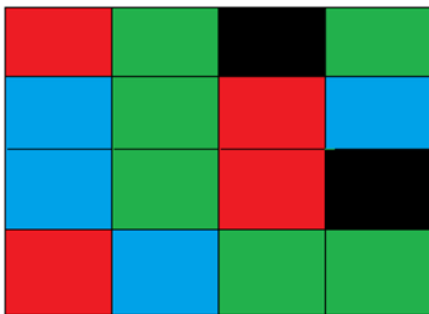


Figure 2-2. *A sample image*

A set of points representing a plot is another example of such a representation.

Model Representation

In engineering, we often model a physical system as a mathematical model where each such model consists of some parameters and design equations. These parameters can also be represented as matrices. The simplest example is a polynomial. Consider $p(x) = x^3 - 2x^2 + 6x + 4$. We can represent this as the matrix $p = [3 \ -2 \ 6 \ 4]$. Other examples are neural networks, transfer functions etc. Consider the transfer function

$$F(s) = \frac{s+3}{s^2+3s+2}$$

which is a ratio of two polynomials, so we can represent it by two vectors $n1 = [1 \ 3]$ and $d = [1 \ 3 \ 2]$. How we are going to use these vectors for model computations (for example, step response) is something we need to remember. Note that all these representations look the same. Since they are represented as matrices, there is no distinction between them from the perspective of MATLAB. When storing some p , MATLAB doesn't make any distinction as to whether p denotes a polynomial or a neural network's bias value. The programmer needs to remember what a vector/matrix represents.

Operations

MATLAB provides a wide range of operations and functions you can perform over vectors. In MATLAB, you can represent each operator as a function too. Therefore, in this discussion, we will be using functions and operators interchangeably. Since a matrix can represent various different notions, as we have seen in the previous section, there are different functions/operators suited for different representations, sometimes with almost the same name and/or purpose. This section will describe these operations and their important differences.

Matrix Operations

The operators that are performed over matrices in their conventional mathematical sense and which follow the laws of linear algebra are known as matrix operators. For example, to invert a transformation, we can use the `invert` operator or `pinv` function.

If $y = Ax$ then $x = A^*y$, where A^* is the Moore-Penrose inverse, which can be computed as follows:

```
x=[2 3 5];
A=[1 0 -1;1 1 0;1 2 3];
y=A*x;
x=pinv(A)*y
```

Suppose we apply a transformation operator twice:

```
y=A*A*x;
```

This is equivalent to computing the square of a matrix via the matrix power operator

```
Asq=A^2
```

and computing y by

```
y=Asq*x
```

Similarly you can compute the square root either by calling the function

```
y=sqrtm(A)*x
```

or via the power operator as

```
y=A^0.5*x
```

We can also compute the product of two matrices using the function `mtimes` or the `*` operator

```
B=mtimes(A,C);
B=A*C;
```

Both of the above lines return the same results. Remember that this is matrix multiplication according to the rules of linear algebra, so the number of columns in A should match the number of rows in C, otherwise the multiplication is not defined and MATLAB will return an error.

Note the letter m in `sqrtm` and `mtimes`, which denotes ‘matrix’. Since the square root and multiplication operations are defined differently for different representations of matrices, the matrix versions are `sqrtm` and `mtimes`. The former computes the matrix square root in the convention mathematical sense. Other important operations are `+`, `\` (`mldivide`), `/` (`mrdivide`), `'` (`ctranspose`), `^` (`mpower`). Readers can refer to the MATLAB documentation for a full list of operations at http://www.mathworks.com/help/matlab/matlab_prog/array-vs-matrix-operations.html. There are also some operators which can only be used in this representation, such as eigenvalue and determinant, which can be computed using the following commands

```
y=eig(A);
z=det(A);
```

Dot (array) Operators

Recall that matrices can also represent data and each element of such a matrix denotes a data value (e.g. time sampled data point, field value at a spatial point). When we perform an operation on such a matrix, it means that we are performing the same operation at each element independently. Such an elementwise operation is known as a dot operation or array operation and this type of operation must be used for data matrices. For example, consider a matrix V containing voltage values at time [0:0.1:1] given as

```
V=[0 .19 .38 .56 .71 .84 .93 .98 .99 .97 .90];
```

and the matrix I representing current values at the same time instances, given as

```
I=[.29 .47 .64 .78 .89 .96 .99 .99 .94 .86 .74];
```

Now we can compute the power values at each time instant by

```
P=V.*I
```

The `.*` operator is known as the times operator and is used to multiply two matrices elementwise. Compare the times operator with the `mtimes` operator discussed in the previous section. The letter ‘m’ is used to differentiate between the two types of multiplication. Here, the times operator is only valid if either the two matrices have the same size or if at least one of them is a scalar. Note that if one of the matrices is a scalar, the times and `mtimes` operators result in the same operation.

Unlike matrix operators, array operators support multi-dimension matrices and operations other than arithmetic operations. We discuss some of these operators here:

Arithmetic Operators

Most arithmetic operators are the array (elementwise) versions of matrix operations and result in numerical values. For example, `+`, `-`, `.`, `*`, `./` (`rdivide`), `.\` (`ldivide`), transpose (`'`), power (`.^`) are used to add, subtract, multiply, divide, left divide, transpose or compute powers of matrices. Most of these operators support scalar matrices or when one of the operands is a scalar. For example,

```
A=3;
B=[2 3; 4 5];
C=[1 2; 3 4];
G=B.^A
```

will result in

```
G=[8 27;64 125]
```

while

```
G=B.^C
```

will pair each corresponding elements ($B(i)$, $C(i)$) in B and C for each i and compute $B(i)^{C(i)}$, placing them at the i^{th} position in G to give

```
G=[2 9;64 625]
```

Other arithmetic operators are trigonometric, exponential, and logarithmic operators. For example, assume $X=[1 \ 4 \ 5]$. To compute $y(x) = x^2 + e^{0.4x} + \sin(x)\log(x)$ for each of the elements of X, we can write

```
Y=X.^2+exp(.4.*X)+log(x).*sin(X).
```

Note that since 0.4 is a scalar, we will occasionally use `*` instead of `.*`, keeping in the mind that `*` in fact represents the array operation and is being used only because it gives the same result as `.*`. For example, the following is equivalent to the previous computation

```
Y=X.^2+exp(.4*X)+log(x).*sin(X).
```

Note that since addition and subtraction are elementwise in both the matrix and array sense, there is no separate matrix and dot version of `+` and `-`.

There are other arithmetic operators which work on the matrix as a whole (which makes sense only for data representation) such as `sum`. The following example computes the average velocity of a system from a vector V containing the instantaneous velocities of it at uniform time samples:

```
avgV=sum(V)/length(V)
```

Note that since `sum(V)` and `length(V)` are both scalars, using the `./` and `/` operators result in the same answer, therefore we can use `/` here. Other similar operations are `prod`, `mean`, etc. Note that for higher dimensional matrices, these operations can be done in any dimension which results in different outputs. For example, consider the matrix

```
A=[2 3 5;1 2 4];
```

Note that the first dimension in MATLAB is taken to be the column dimension. Therefore the function `sum` will result in the columnwise sum

```
s=sum(A)
```

giving

```
s=[3 5 9].
```

However, if we need the operation to be performed in a different dimension, we can specify it. For example, the following will result in the rowwise sum

```
s=sum(A,2)
```

giving

```
s=[ 10
    7  ];
```

The operation `cumsum` computes the cumulative sum of any vector/matrix. Suppose (x, p) denotes the probability mass function of a random variable X

```
x=[0 2 5 7 10];
p=[.2 .4 .1 .05 .25];
```

We can compute the cumulative distribution function (`cdf`) by the following

```
f1=cumsum(p);
f=[.2 f1]
```

Note the use of appending the element 0.2 in `f`. Since `cumsum`'s output `f1` is one element shorter than `p` and the first element of `f1` corresponds to the second element of `x`, we need to concatenate 0.2 to `f1` to generate `f` such that the first element of `f` is equal to 0.2 and the second element of `f` (which is also the first element of `f1`) corresponds to the second element of `x`.

Logical Operators

Logical operators are applied over logical matrices. These operations again are elementwise and result in a matrix of the same size. The most common logical operators are `&` (AND), `|` (OR), `~` (NOT). The following example will compute the AND of two logical matrices `x` and `y`

```
x=[true false true]
y=[false true true]
z=x & y;
```

There are some logical operators which are performed over whole matrices. For example,

```
z=any(x)
```

will be true if any element of x is true.

Relational Operators

Relational operators can be used over numerical matrices to compare them and the result is typically a logical matrix. For example, to compute if each element of x is greater than the corresponding element of y, we can write

```
x=[3 4 5 1 3 5];
y=[3 3 2 5 6 3];
z=x>y;
```

Here z will be the same size as x and its i^{th} element will be 1 if the i^{th} element of x is greater than the i^{th} element of y. Similarly

```
z=x==y
```

will compare each element of x with the corresponding element of y, returning true if they are equal and false otherwise. You can also use `z=eq(x,y)` to achieve the same result.

Operations for Models

MATLAB also provides operations for various model representations. There are some operations which are valid for many models but may have different meanings and definitions.

For example, consider polynomials. Let $p_1(x) = x + 3$ and $p_2(x) = x + 1$, which can be represented as

```
p1=[1 3]
p2=[1 1]
```

To compute, $p_3(x) = p_1(x)p_2(x)$ we can write

```
p3=conv(p1,p2)
```

which will give

```
p3=[1 4 3]
```

denoting $p_3 = x^2 + 4x + 3$.

We can say that the multiplication operation has a new meaning here and carries a different name, convolution. We can perform polynomial specific operations, such as computing roots

```
f=root(p3)
```

with the result `f = [-1 -3]` containing the roots of $p_3(x)$ or compute the values of a polynomial at a vector x by

```
x=[1 2 3];
y=polyval(p3,x)
```

resulting in `y=[7 15 27]`.

Let us take another model: the transfer function. Consider two systems defined as

$$S_1 = \frac{1}{s+3}, S_2 = \frac{1}{s+1}$$

which can be represented as

```
S1num=1;S1den=[1 3];S1=tf(S1num,S1den);
S2num=1;S2den=[1 1];S2=tf(S2num,S2den);
```

The S_3 resulting from the serial connection of these two systems is given as

$$S_3 = S_1 S_2 = \frac{1}{s+3} \times \frac{1}{s+1} = \frac{1}{(s+3)(s+1)}$$

and can be computed in MATLAB as

```
S3=S1*S2;
```

We see that the `*` operation is overloaded here to give the cascaded output of two systems. Similarly, you can call the step function over a system to evaluate its step response.

```
y=step(S1);
```

All these operations are specific to models. When using these operations, the programmer must keep in mind which model a matrix represents, since MATLAB is not able to differentiate between these matrices and will return the output without throwing an error in most cases. For example, you can call a filter for a transfer function numerator polynomial, but it will not make sense.

There is a wide range of such model representations and model-specific operations. Interested readers should refer to toolbox-specific documentation on the MATLAB website.

Indexing

Indexing of a matrix can be done in several ways. This section will describe different ways of indexing and their pros and cons. For the sake of the following examples, let us assume A is a 4×5 matrix.

Normal Indexing

Normal indexing is the standard indexing of any MATLAB matrix where we need to specify indices in each dimension (for example row and column index for a two dimensional matrix). The following example will extract the element located at the second row and third column of the matrix A :

```
y=A(2,3);
```

Similarly, to change the (3,4)th element of A to 5 we can write

```
A(3,4)=5
```


We can extract multiple columns and rows by specifying the indices as vectors. The following will extract a square matrix containing the second and third rows and third and first columns of the matrix A

```
y=A([2 3],[3 1]);
```

The following will extract all rows and the 5th and 4th columns of the matrix A

```
y=A(:,[5 4])
```

where `:` denotes all. The following will reverse the column order

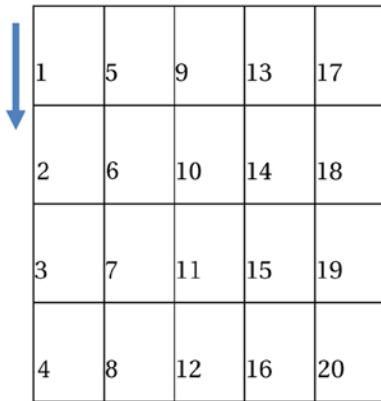
```
B=A(:,[end:-1:1])
```

where `end` is a MATLAB keyword which denotes the length in the dimension it is used.

Linear Indexing

We can also index a matrix by using only one index. The idea is to stack all the elements in one direction, proceeding columnwise as shown in Figure 2-3, indexing the entries by the numbers indicated. For example, we can linearly index A as in Figure 2-3 and we can index (2,3) by 10 using

```
y=A(10).
```



1	5	9	13	17
2	6	10	14	18
3	7	11	15	19
4	8	12	16	20

Figure 2-3. Linear indexing order in a 2D matrix

As for normal indexing, we can also use vectors to index multiple elements. For example, the following code will extract the diagonal of A

```
y=A(1:5:20)
```

Logical Indexing

Logical indexing or indexing by a logical variable is among the best features of MATLAB and this combined with relational operators can make programs very concise and elegant. Suppose I is matrix of the same size as A with true and false entries then

```
B=A(I)
```

will give a matrix containing only those elements of A for which the corresponding elements of I are true. The matrix I can be formed by using a relational operator. For example, suppose we want to extract all those elements from A which are bigger than 5. We first compute I by

```
I=A>5
```

then we index A by I

```
B=A(I)
```

where B will contain only those elements of A which are bigger than 5. Similarly, to replace each element of A which is bigger than 10 with 5, we can write

```
A(A>10)=5
```

Let us see some examples which use the tricks we have learnt so far.

Clipping a Signal

Suppose a signal $x(t)$ is given as

```
t=0:.01:2*pi  
x=sin(2*pi*t);
```

Suppose we apply this signal to a clipper circuit which can only pass a signal between -0.9 and 0.7. The following operation will generate the equivalent output $y(t)$

```
y=x;  
y(y>0.7) = 0.7;  
y(y<-0.9) = -0.9;
```

Halving a Matrix

```
A=rand(100,100);  
B=A(1:2:end,1:2:end);
```

Vectorization

It should be clear by now that MATLAB is designed for vector and matrix operations. You can often speed up your M-file code by using vectorizing algorithms that take advantage of this design. Vectorization means converting for and while loops to equivalent vector or matrix operations to speed up computations. We will learn more about this in this section.

Recall that we can compute the cube of each element of a matrix via

```
y=x.^3.
```

We can achieve the same via a traditional programming loop

```
for i=1:length(x)
    y(i)=x(i)^3
end
```

It is easy to see that the first method is short and elegant. But as well as being short, the first one is also efficient. Using the first method, MATLAB needs to call the Java function only once with the whole vector while in the second method, MATLAB calls the Java function multiple times, once for each element. This difference in fact becomes more prominent when working with large matrices. The use of functions/operators over whole matrices to avoid loops is known as vectorization. In this section we will learn some vectorization tricks via a few examples.

Example 1. Creating C such that $C(i,j)=A(i)^B(j)$

Suppose A ($1 \times m$) and B ($n \times 1$) are vectors and we are interested in generating a matrix C ($n \times m$) with elements $C(i,j) = A(i)^{B(j)}$.

We will first replicate A n times in rows to make it an $n \times m$ matrix Ar . Then we replicate B m times in columns so it also becomes an $n \times m$ matrix Br . Now we can perform $Ar.^Br$ to get C . We can see that $C(i,j) = Ar(i,j)^{Br(i,j)} = A(i)^{B(j)}$ as $Ar(i,j) = A(i)$ and $Br(i,j) = B(j)$.

```
Ar=repmat(A,n,1);
Br=repmat(B,1,m);
C=Ar.^Br;
```

Example 2. Calculating the Sum of Harmonics

Suppose we want to compute the signal $y(t)$ given as

$$y(t) = \sum_{N=1}^{10} \frac{1}{2N-1} \sin[2\pi(2N-1)50t]$$

for the range $t=0$ to 0.6 .

First we try to solve this problem using for loops

```
t = 0:0.001:0.6;
n=1:2:19;
y=0;
for i=n
    y=y+(1/i)*sin(2*pi*i*50*t)
end
```

Now let us try using vectorization. We need to understand that we need two dimensions because there is a time vector (column or horizontal dimension) and the frequency is also varying (row or vertical dimension).

```
n=[1:2:19]';
t=0:.001:0.6;
```

Now we perform a matrix multiplication (not elementwise multiplication) to generate a grid

```
h=n*t;
```

Here h has 10 rows: one row for each frequency and each column represents a time instant. The following

```
g=sin(2*pi*50*h);
```

will compute the signal for each possible frequency and time instant. We want to divide each row by the corresponding frequency from the vector n but we cannot use elementwise division of g by n because n has only one column while g has multiple columns. We observe that each row needs to be divided by the same element n(i) so we replicate n(i) so that the size of n becomes equal to the size of g and all elements of each row of n are the same as the first element of that row. The command

```
n_new=repmat(n,1,size(g,2));
```

will produce the desired matrix. Finally, we can sum each row of the division output columnwise to sum all the frequencies for each time instant.

```
x=sum(g./n_new);
```

Example 3. Conversion to Matrix Operations

Suppose we need to compute a vector A such that $A_i = \sum_m w_k B_{km}$ where w is a vector and B is a matrix. This can equivalently be computed as

```
A=w*B;
```

Example 4. Selective Inversion

Consider the following example. Let A equal [1 2 0 6 4 0 2] and suppose we are interested in constructing the vector B such that

$$B(i) = \begin{cases} \frac{1}{A(i)} & \text{if } A(i) \neq 0 \\ 0 & \text{if } A(i) = 0 \end{cases}$$

We will first make B equal to A. Then we will select those elements of B which are not 0 to obtain C. Then we will invert these elements to obtain D. Then we will reassign only these values (i.e. elements in D) to their original indices (the indices corresponding to non-zero elements of B).

```
A=[1 2 0 6 4 0 2];
B=A;
C=B(B~=0);
D=1./C;
B(B~=0)=D;
```

The answer is $B = [1 \ 1/2 \ 0 \ 1/6 \ 1/4 \ 0 \ 1/2]$.

We see that we sometimes need to apply logical indexing multiple times.

Tips for Performance Improvement

Let's review some tips for performance improvement.

Vectorization

As discussed, vectorization is very helpful in reducing computational time. It also makes code elegant and short. Sometimes, when there are multiple loops and some loops cannot be avoided, we should still try to vectorize as many loops as we can. Consider the following example where we want to compute the factorial of each element of the vector $x = [2 \ 3 \ 7 \ 4]$.

First Attempt: Two Loops

```
for i=1:4
    s=1;
    for j=1:x(i)
        s=s*j;
    end
    A(i)=s;
end
```

Second Attempt: One Loop

```
for i=1:4
    A(i)=prod(1:x(i));
end
```

Third Attempt: No Loop

```
mx=max(x);
S=1:mx;
r=[1 cumprod(S)];
A=r(x);
```

Preallocating Arrays

There are cases when loops cannot be avoided. In such cases, we should still make an effort to reduce the computational time by other means. For example, when a variable changes its size at each iteration, it is suggested to pre-allocate it a fixed size to improve the performance and computational time.

Fixed Type Variables

Although MATLAB lets you change the type of a variable within a program, for best performance it is recommended that you should not do it frequently. Changing the class or array shape of an existing variable slows MATLAB down as it must take extra time to process it. When you need to store data of a different type, it is advisable to create a new variable.