5 Mathematical operations with matrices

There are different types of mathematical functions that operate on matrices. Here, we will describe three. First, there are functions that operate element-by-element in a way similar to ordinary algebra. Second, there are functions that, while they operate on the whole matrix or a subset of it, are very similar to ordinary algebra, for example functions for summing rows or columns. Third, there are functions that operate on the whole matrix, for example matrix multiplication.

5.1 Functions that operate element-by-element

Most ordinary mathematical functions in Matlab operate element-by-element when used on matrices. The main exceptions are the operators for multiplication, division, and exponentiation. These reason why these do not operate element-by-element is that the standard in Matlab is matrix algebra. And matrix multiplication, exponentiation, and division (or, rather, inversion) do not operate element-by-element. Addition and subtraction, however, operate element-by-element in matrix algebra as well, so in those cases there are no differences. All four basic arithmetic operations are described in Section 5.3, where the topic is matrix algebra.

Informally, ordinary mathematical functions that take the arguments within parentheses, such as `sqrt()` and `log()`, operate element-by-element, whereas functions that do not use parentheses, such as `*` or `/`, operate according to matrix algebra.

To multiply or divide matrices element-by-element using the ordinary operators `*` and `/`, you have to use so called dot-notation. This means that you put a dot in front of the operators. For example

```
>> X = [2 4 ; 3 4]; Y = [2 2 ; 3 1]; X.*Y, X./Y
ans =
      4    8
      9    4

ans =
      1    2
      1    4
```

Each element in the first answer is the product of the corresponding elements in `X` and `Y`, and each element in the second answer is the corresponding element in `X` divided by its counterpart in `Y`. Note that, the matrices must have exactly the same dimensions. The exception is if either `X` or `Y` is a scalar.

There are actually alternatives to these functions, and to plus and minus as well, that operate element-by-element as well. These are `plus()`, `minus()`, `times()`, and `rdivide()` for, in turn, element-by-element addition, subtraction, multiplication, and division. rdivide stands for "right division" (i.e., divide by the argument to the right). There is a "left division" command as well: `ldivide()`.

Other mathematical functions we have seen that use parentheses, such as the square root, also operate element-by-element.

```
>> sqrt(X)
ans =
      1.4142       2
      1.7321       2
```

In Section 4.2.3 we saw that relational operators also work element-by-element when comparing a matrix to a scalar. An example comparing two full matrices instead is

```
>> X > Y
ans =
      0    1
      0    1
```

Each element in `X` is compared to the corresponding element in Y. Two of them are not greater than the ones they are compared to; two are.

5.2 Elementary mathematical functions that operate columnwise

Matrices often contain data observations and it is practical to calculate ordinary summary statistics directly on them. The following are some elementary mathematical and statistical functions that operate on whole columns. For the statistical functions, think of `X` as a matrix containing many observations of a few variables, where each column represents a certain variable, for example weight or length, and each row contains one observation of each variable.

`min(X)`	The minimum value of `X` (columnwise). `[val,row]=min(X)` produces two vectors. `val` contains the maximum values for each column and `row` contains the row numbers where each corresponding maximum is located.
`max(X)`	The maximum value of `X` (columnwise). `[val,row]=max(X)` works similarly to `min(X)`.
`mean(X)`	The mean value of each column of `X`.
`median(X)`	The median value of each column of `X`.
`std(X)`	The standard deviation of each column of `X`.
`var(X)`	The variance of each column of `X`.
`sum(X)`	The columnwise sum of all values in `X`.
`cumsum(X)`	The columnwise cumulative sum of the values in `X`.
`prod(X)`	The columnwise product of all values in `X`.
`cumprod(X)`	The columnwise cumulative product of the values in `X`.
`diff(X)`	The difference between each consecutive element in `X`, columnwise.

In addition, the following calculate covariances and correlation coefficients.

`cov(X)`	Calculates the covariance matrix, assuming that each column in `X` represents outcomes of a variable.
`corrcoef(X)`	Calculates a matrix of correlation coefficients, assuming that each column in `X` represents outcomes of a variable.

If, for example, we want to calculate the mean amount of rain for the two remaining locations in `testMatrix`, we enter

```
>> mean(testMatrix(:,2:end))
ans =
        3.6667           7
```

Here, we pick out all rows and columns 2 and 3 and calculate the means of those two columns. Since the first column only contains day numbers, it makes little sense to calculate the mean of that column.

If you want the commands to operate rowwise, instead of columnwise, a convenient way of doing that is to use the transpose operator twice, as in

```
>> mean(testMatrix(:,2:end)')'
ans =

   7.5
     4
   4.5
```

This calculates the daily mean amount of rain in the two locations.

Note that, if `X` is a row vector, the commands here do not operate on the columns of `X`, but on the single row. For example

```
>> max([1  2  3  4  5  6])
ans =
      6
```

5.3 Matrix algebra

If you have not seen matrix algebra before, it probably seems counterintuitive. It has many uses, though. For instance to solve systems of linear equations and to estimate regression coefficients, as we will see in Section 5.4 and Section 5.5. Here, we only present definitions of basic algebraic operators and how to use these operators in Matlab.

5.3.1 Matrix addition and subtraction

To add or subtract two matrices, they have to be of the same dimensions and all operations are element-by-element. The definition for matrix addition is[6]

$$\boldsymbol{X}+\boldsymbol{Y}=\begin{bmatrix} x_{1,1} & x_{1,2} & \cdots & x_{1,n} \\ x_{2,1} & x_{2,2} & \cdots & x_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ x_{T,1} & x_{T,2} & \cdots & x_{T,n} \end{bmatrix}+\begin{bmatrix} y_{1,1} & y_{1,2} & \cdots & y_{1,n} \\ y_{2,1} & y_{2,2} & \cdots & y_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ y_{T,1} & y_{T,2} & \cdots & y_{T,n} \end{bmatrix}=\begin{bmatrix} x_{1,1}+y_{1,1} & x_{1,2}+y_{1,2} & \cdots & x_{1,n}+y_{1,n} \\ x_{2,1}+y_{2,1} & x_{2,2}+y_{2,2} & \cdots & x_{2,n}+y_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ x_{T,1}+y_{T,1} & x_{T,2}+y_{T,2} & \cdots & x_{T,n}+y_{T,n} \end{bmatrix},$$

where both $\boldsymbol{X}$ and $\boldsymbol{Y}$ are Txn matrices (i.e., they have T rows and n columns). Subtraction is defined analogously. In Matlab, you issue the ordinary addition or subtraction operators, + or –. For example,

```
>> X=[1 2 ; 3 4]; Y = [4 3 ; 2 1]; X+Y, Y-X
ans =

      5   5
      5   5

ans =

       3   1
      -1  -3
```

Note that you can add or subtract a scalar from a matrix of any dimensions. The scalar is then added to, or subtracted from, each element in the matrix.

5.3.2 Matrix multiplication

To multiply two matrices, the number of columns of the first matrix must equal the number of rows of the second. If $\boldsymbol{X}$ is a Txk matrix and $\boldsymbol{Y}$ is a kxn matrix, then $\boldsymbol{X}^*\boldsymbol{Y}$ is a Txn matrix (i.e., the product has the same number or rows as the first matrix and the same number of columns as the second). The definition is

$$\boldsymbol{X} * \boldsymbol{Y} = \begin{bmatrix} x_{1,1} & x_{1,2} & \cdots & x_{1,k} \\ x_{2,1} & x_{2,2} & \cdots & x_{2,k} \\ \vdots & \vdots & \ddots & \vdots \\ x_{T,1} & x_{T,2} & \cdots & x_{T,k} \end{bmatrix} * \begin{bmatrix} y_{1,1} & y_{1,2} & \cdots & y_{1,n} \\ y_{2,1} & y_{2,2} & \cdots & y_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ y_{k,1} & y_{k,2} & \cdots & y_{k,n} \end{bmatrix}$$

$$= \begin{bmatrix} x_{1,1} * y_{1,1} + x_{1,2} * y_{2,1} + \cdots + x_{1,k} * y_{k,1} & x_{1,1} * y_{1,2} + x_{1,2} * y_{2,2} + \cdots + x_{1,k} * y_{k,2} & \cdots & x_{1,1} * y_{1,n} + x_{1,2} * y_{2,n} + \cdots + x_{1,k} * y_{k,n} \\ x_{2,1} * y_{1,1} + x_{2,2} * y_{2,1} + \cdots + x_{2,k} * y_{k,1} & x_{2,1} * y_{1,2} + x_{2,2} * y_{2,2} + \cdots + x_{2,k} * y_{k,2} & \cdots & x_{2,1} * y_{1,n} + x_{2,2} * y_{2,n} + \cdots + x_{2,k} * y_{k,n} \\ \vdots & \vdots & \ddots & \vdots \\ x_{T,1} * y_{1,1} + x_{T,2} * y_{2,1} + \cdots + x_{T,k} * y_{k,1} & x_{T,1} * y_{1,2} + x_{T,2} * y_{2,2} + \cdots + x_{T,k} * y_{k,2} & \cdots & x_{T,1} * y_{1,n} + x_{T,2} * y_{2,n} + \cdots + x_{T,k} * y_{k,n} \end{bmatrix}$$

Note that the definition implies that, in general, $\boldsymbol{X}^*\boldsymbol{Y} \neq \boldsymbol{Y}^*\boldsymbol{X}$, and that for it to be possible to calculate both $\boldsymbol{X}^*\boldsymbol{Y}$ and $\boldsymbol{Y}^*\boldsymbol{X}$, both matrices have to be square matrices.

In Matlab, you calculate matrix multiplication by using the ordinary multiplication operator, `*`. For example, using the previously defined `X` and `Y`

```
>> X*Y, Y*X
ans =
     8    5
    20   13
ans =
    13   20
     5    8
```

As you see from the example, multiplying `X` by `Y` from the left or from the right, premultiplying or postmultiplying, yields different answers. As an alternative to using the operator `*`, you can multiply matrices using the command `mtimes(X,Y)`.

5.3.3 Inverting a matrix

Division is not defined in matrix algebra. However, there is a corresponding concept that applies to square matrices. Remember that, dividing two scalars is equivalent to multiplying the first scalar with the inverse of the second: $\frac{X}{Y} = X * \frac{1}{Y}$. As inversion is defined in matrix algebra, multiplying a matrix with the inverse of another is similar to dividing the two matrices.

To understand the definition of matrix inversion, note that for scalars $X * \frac{1}{X} = 1$. An informal implicit definition of scalar inversion could then be "the number you multiply the scalar with to get the answer 1". What is the matrix equivalent of the number 1? The so-called identity matrix, usually denoted $\boldsymbol{I}$, is a square matrix with ones on the diagonal, and zeros elsewhere. This, you might recall, is the matrix that you create with the command `eye()`.[7] If you multiply any square matrix with $\boldsymbol{I}$, the result is the matrix you started with, so it is similar to multiplying a scalar with 1.

The identity matrix is used in the definition of matrix inverse. If there exist two matrices ***A*** and ***B*** such that ***A********B*** = ***I***, then ***B*** is the inverse of ***A*** and is denoted A^{-1}. It can be shown that if ***A********B*** = ***I*** then it is also the case that ***B********A*** = ***I***, so in this case it does not matter if you multiply from the right or from the left. Note, however, that not all square matrices are invertible, just as it is not possible to invert the scalar 0.

In Matlab, you calculate matrix inverse with `inv()` or by raising the matrix to the power of `-1`.

```
>> inv(X), X^-1
ans =
              -2       1
             1.5    -0.5
ans =

              -2       1
             1.5    -0.5
```

Matlab, however, also allows for matrix inversion using the division operators, `/` and `\`, although this notation is nonstandard. In these cases, the matrix that is *above* the division sign (i.e., to the left of `/` or to the right of `\`) is multiplied with the inverse of the other matrix. For example,

```
>> eye(2)/X, X\eye(2)
ans =
              -2       1
             1.5    -0.5
ans =
              -2       1
             1.5    -0.5
```

give the same results as above, and consequently also invert X. Furthermore

```
>> Y/X, Y*inv(X)
ans =
           -3.5     2.5
           -2.5     1.5
ans =

           -3.5     2.5
           -2.5     1.5
```

are two different ways of calculating $Y*X^{-1}$. As alternatives to using the operators `/` and `\`, you can use the commands `mrdivide(X,Y)` and `mldivide(X,Y)`, where the former is matrix right division and the latter is matrix left division.

5.3.4 Commands for linear algebra

Commonly used commands for linear algebra include

`norm(X)`	Matrix or vector norm.
`rank(X)`	Matrix rank.
`det(X)`	Determinant.
`trace(X)`	The sum of the diagonal elements.
`chol(X)`	Cholesky factorization.
`eig(X)`	Eigenvalues and eigenvectors.
`expm(X)`	Matrix exponential.
`logm(X)`	Matrix logarithm.
`sqrtm(X)`	Matrix square root.

5.4 Solving systems of linear equations

Matrix algebra provides an easy way to solve systems of linear equations. Suppose we have the following system of three equations

$$\begin{cases} x_1 + 2x_2 + 3x_3 = 15 \\ x_1 + 2x_2 + x_3 = 13 \\ -2x_1 + 5x_2 - 2x_3 = 19 \end{cases}$$

Such a system can be written in matrix notation as $\boldsymbol{A}^{*}\boldsymbol{X} = \boldsymbol{Y}$, where

$$\mathbf{A} = \begin{bmatrix} 1 & 2 & 3 \\ 1 & 2 & 1 \\ -2 & 5 & -2 \end{bmatrix}, \mathbf{X} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} \text{ and } \mathbf{Y} = \begin{bmatrix} 15 \\ 13 \\ 19 \end{bmatrix}$$

Now, if we premultiply each side of the equality by the inverse of A, we get

$$\boldsymbol{A}^{-1*}\boldsymbol{A}^{*}\boldsymbol{X} = \boldsymbol{A}^{-1*}\boldsymbol{Y}$$
$$\boldsymbol{I}^{*}\boldsymbol{X} = \boldsymbol{A}^{-1*}\boldsymbol{Y}$$
$$\boldsymbol{X} = \boldsymbol{A}^{-1*}\boldsymbol{Y}$$

To calculate the solution in Matlab, we use either the `inv()` command or `\`.

```
>> A=[1 2 3 ; 1 2 1 ; -2 5 -2]; Y=[15 13 19]'; X=A\Y, X=inv(A)*Y

X =
      2
      5
      1
X =
      2
      5
      1
```

Consequently, $x_1 = 2$, $x_2 = 5$, and $x_3 = 1$. To check this, issue

```
>> A*X
ans =
    15
    13
    19
```

which equals $\boldsymbol{Y}$.

5.5 Finding linear regression coefficients

You can find linear regression coefficients in a way similar to how we solved the system of linear equations. Suppose we have a large number of observations, say n observations, for one or several independent variables, say p variables, and equally many observations of a dependent variable, and that the model is

$$y_i = \beta_0 + \beta_1 x_{i,1} + \cdots + \beta_p x_{i,p} + \varepsilon_i, i = 1, \ldots, n.$$

Then we can write this in matrix form as $\boldsymbol{y} = \boldsymbol{X\beta} + \boldsymbol{\varepsilon}$, where

$$\boldsymbol{y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}, \boldsymbol{X} = \begin{bmatrix} 1 & x_{1,1} & \cdots & x_{1,p} \\ 1 & x_{2,1} & \cdots & x_{2,p} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & x_{n,1} & \cdots & x_{n,p} \end{bmatrix}, \boldsymbol{\beta} = \begin{bmatrix} \beta_0 \\ \beta_1 \\ \vdots \\ \beta_p \end{bmatrix}, \boldsymbol{\varepsilon} = \begin{bmatrix} \varepsilon_1 \\ \varepsilon_2 \\ \vdots \\ \varepsilon_n \end{bmatrix}.$$

In other words, $\boldsymbol{y}$ contains all observations of the dependent variable, $\boldsymbol{X}$ contains all observations of the p independent variables preceded by a column of ones to match the constant term (β_0), $\boldsymbol{\beta}$ contains the $p+1$ coefficients, and $\boldsymbol{\varepsilon}$ contains the error terms. Note that, if you perform the matrix multiplication, you get the same as in the preceding expression.

There are several ways to estimate $\boldsymbol{\beta}$, but the most frequently used is the OLS estimate (Ordinary Least Squares). In matrix algebra, the estimator can be written[1] $\widehat{\boldsymbol{\beta}} = (\boldsymbol{X}'\boldsymbol{X})^{-1}\boldsymbol{X}'\boldsymbol{y}$, where the "hat" on $\boldsymbol{\beta}$ indicates that it is an estimate.

You can use the OLS formula in Matlab by issuing `inv(X'*X)*X'*y` or `(X'*X)\X'*y`, but there is actually a better way. Looking at the matrix version of the model again, we see that if we could have premultiplied $\boldsymbol{y}$ with the inverse of $\boldsymbol{X}$, and ignore the error terms, this would have solved for $\boldsymbol{\beta}$. Mathematically, this is not correct, since $\boldsymbol{X}$ is generally not invertible. However, Matlab supports this notation for solving linear regressions, so you can calculate the OLS estimate of $\boldsymbol{\beta}$ as `X\y`. Internally, Matlab uses a completely different method to estimate $\boldsymbol{\beta}$ in this latter case, though. One that is more efficient. So using `X\y` is the preferred method in Matlab. To illustrate this, let us create sample variables and perform the regression.

```
>> X=[ones(1000,1) randn(1000,3) ]; e=randn(1000,1)*0.2;
>> beta=[0.1 0.2 0.3 0.4]'; y=X*beta+e;
>> betahat_1=inv(X'*X)*X'*y, betahat_2=X\y
betahat_1 =
      0.10063
      0.2065
      0.30098
      0.41276
betahat_2 =
      0.10063
      0.2065
      0.30098
      0.41276
```

The first two lines create the $\boldsymbol{X}$, $\boldsymbol{\varepsilon}$, $\boldsymbol{\beta}$, and $\boldsymbol{y}$ matrices. `X` is a matrix of random numbers and a column of ones, `e` is a vector of random numbers that are scaled to reduce the noise, `beta` is a column vector of coefficients, and `y` is a vector created using the other variables. The estimators, `betahat_1` and `betahat_2`, both produce the same estimates. The estimates, in turn, are close to the original $\boldsymbol{\beta}$.