9 Matrices and vectors

Vectors are special cases of *matrices*. A matrix is a rectangular array of numbers, the size of which is usually described as $m \times n$, meaning that it has m rows and n columns. For example, here is a 2×3 matrix:

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

To enter this matrix into MATLAB you use the same syntax as for vectors, typing it in row by row:

```
>> A=[5 7 9
     -1 3 -2]
A =
     5     7     9
    -1     3    -2
>>
```

alternatively, you can use semicolons to mark the end of rows, as in this example:

```
>> B=[2 0; 0 -1; 1 0]
B =
     2     0
     0    -1
     1     0
```

You can also use the colon notation, as before:

```
>> C = [1:3; 8:-2:4]
C =
     1     2     3
     8     6     4
```

A final alternative is to build up the matrix row-by-row (this is particularly good for building up tables of results in a `for` loop):

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

9.1 Matrix multiplication

With vectors and matrices, the `*` symbol represents matrix multiplication, as in these examples (using the matrices defined above)

```
>> A*B
ans =
    19    -7
    -4    -3
>> B*C
ans =
     2     4     6
    -8    -6    -4
     1     2     3
>> A*C
??? Error using ==> *
Inner matrix dimensions must agree.
```

You might like to work out these examples by hand to remind yourself what is going on. Note that you cannot do `A*C`, because the two matrices are incompatible shapes.[11]

When just dealing with vectors, there is little need to distinguish between row and column vectors. When multiplying vectors, however, it will only work one way round. A row vector is a $1 \times n$ matrix, but this can't post-multiply a $m \times n$ matrix:

```
>> x=[1 0 3]
x =
     1     0     3
>> A*x
??? Error using ==> *
Inner matrix dimensions must agree.
```

9.2 The transpose operator

Transposing a vector changes it from a row to a column vector and vice versa. The transpose of a matrix interchanges the rows with the columns. Mathematically, the transpose of `A` is represented as $\mathtt{A}^T$. In MATLAB an apostrophe performs this operation:

```
>> A
A =
     5     7     9
    -1     3    -2
>> A'
ans =
     5    -1
     7     3
     9    -2
```

[11] In general, in matrix multiplication, the matrix sizes are

$$(l \times m) * (m \times n) \rightarrow (l \times n)$$

When we try to do `A*C`, we are attempting to do $(2 \times 3) * (2 \times 3)$, which does not agree with the definition above. The middle pair of numbers are not the same, which explains the wording of the error message.

```
>> A*x'
ans =
    32
    -7
```

In this last example the `x'` command changes our row vector into a column vector, where it can be pre-multiplied by the matrix `A`.

9.3 Matrix creation functions

MATLAB provides some functions to help you build special matrices. We have already met `ones` and `zeros`, which create matrices of a given size filled with 1 or 0.

A very important matrix is the *identity matrix*. This is the matrix that, when multiplying any other matrix or vector, does not change anything. This is usually called `I` in formulæ, so the MATLAB function is called `eye`. This only takes one parameter, since an identity matrix must be square:

```
>> I = eye(4)
I =
     1     0     0     0
     0     1     0     0
     0     0     1     0
     0     0     0     1
```

we can check that this leaves any vector or matrix unchanged:

```
>> I * [5; 8; 2; 0]
ans =
     5
     8
     2
     0
```

The identity matrix is a special case of a *diagonal matrix*, which is zero apart from the diagonal entries:

$$D = \begin{bmatrix} -1 & 0 & 0 \\ 0 & 7 & 0 \\ 0 & 0 & 2 \end{bmatrix}$$

You could construct this explicitly, but the MATLAB provides the `diag` function which takes a vector and puts it along the diagonal of a matrix:

```
>> diag([-1 7 2])
ans =
    -1     0     0
     0     7     0
     0     0     2
```

The `diag` function is quite sophisticated, since if the function is called for a matrix, rather than a vector, it tells you the diagonal elements of that matrix. For the matrix `A` defined earlier:

```
>> diag(A)
ans =
     5
     3
```

Notice that the matrix does not have to be square for the diagonal elements to be defined, and for non-square matrices it still begins at the top left corner, stopping when it runs out of rows or columns.

9.4 Building composite matrices

It is often useful to be able to build matrices from smaller components, and this can easily be done using the basic matrix creation syntax:

```
>> comp = [eye(3) B;
           A    zeros(2,2)]
comp =
     1     0     0     2     0
     0     1     0     0    -1
     0     0     1     1     0
     5     7     9     0     0
    -1     3    -2     0     0
```

You just have to be careful that each sub-matrix is the right size and shape, so that the final composite matrix is rectangular. Of course, MATLAB will tell you if any of them have the wrong number of row or columns.

9.5 Matrices as tables

Matrices can also be used to simply tabulate data, and can provide a more natural way of storing data:

```
>> t=0:0.2:1;
>> freq=[sin(t)' sin(2*t)', sin(3*t)']
freq =
         0         0         0
    0.1987    0.3894    0.5646
    0.3894    0.7174    0.9320
    0.5646    0.9320    0.9738
    0.7174    0.9996    0.6755
    0.8415    0.9093    0.1411
```

Here the nth column of the matrix contains the (sampled) data for $\sin(nt)$. The alternative would be to store each series in its own vector, each with a different name. You would then need to know what the name of each vector was if you wanted to go on and use the data. Storing it in a matrix makes the data easier to access.

9.6 Extracting bits of matrices

Numbers may be extracted from a matrix using the same syntax as for vectors, using the `()` brackets. For a matrix, you specify the row co-ordinate and then the column co-ordinate (note that, in Cartesian terms, this is y and then x). Here are some examples:

```
>> J = [
     1     2     3     4
     5     6     7     8
    11    13    18    10];
>> J(1,1)
ans =
     1
>> J(2,3)
ans =
     7
>> J(1:2, 4)   %Rows 1-2, column 4
ans =
     4
     8
>> J(3,:)      %Row 3, all columns
ans =
    11    13    18    10
```

The `:` operator can be used to specify a range of elements, or if used just by itself then it refers to the entire row or column.

These forms of expressions can also be used on the left-hand side of an expression to write elements into a matrix:

```
>> J(3, 2:3) = [-1 0]
J =
     1     2     3     4
     5     6     7     8
    11    -1     0    10
```

10 Basic matrix functions

MATLAB allows all of the usual arithmetic to be performed on matrices. Matrix multiplication has already been discussed, and the other common operation is to add or subtract two matrices of the same size and shape. This can easily be performed using the `+` and `-` operators. As with vectors, MATLAB also defines the `.*` and `./` operators, which allow the corresponding elements of two matching matrices to be multiplied or divided. All the elements of a matrix may be raised to the same power using the `.^` operator.

Unsurprisingly, MATLAB also provides a large number of functions for dealing with matrices, and some of these will be covered later in this tutorial. Try typing

```
help matfun
```

for a taster. For the moment we'll consider some of the fundamental matrix functions.

The `size` function will tell you the dimensions of a matrix. This is a function which returns a vector, specifying the number of rows, and then columns:

```
>> size(J)
ans =
     3     4
```

The inverse of a matrix is the matrix which, when multiplied by the original matrix, gives the identity ($\mathtt{AA}^{-1} = \mathtt{A}^{-1}\mathtt{A} = \mathtt{I}$). It 'undoes' the effect of the original matrix. It is only defined for square matrices, and in MATLAB can be found using the `inv` function:

```
>> A = [
     3     0     4
     0     1    -1
     2     1    -3];
>> inv(A)
ans =
    0.1429   -0.2857    0.2857
    0.1429    1.2143   -0.2143
    0.1429    0.2143   -0.2143
>> A*inv(A)  %Check the answer
ans =
    1.0000    0.0000   -0.0000
         0    1.0000         0
         0    0.0000    1.0000
```

Again, note the few numerical errors which have crept in, which stops MATLAB from recognising some of the elements as exactly one or zero.

The *determinant* of a matrix is a very useful quantity to calculate. In particular, a zero determinant implies that a matrix does not have an inverse. The `det` function calculates the determinant:

```
>> det(A)
ans =
   -14
```

Summary

After reading through sections 9 and 10 of the tutorial guide and working through the examples you should be able to:

- ☐ Define matrices in MATLAB
- ☐ Use matrices and vectors in simple calculations
- ☐ Perform standard operations on matrices

Exercise 5: Euler angles

A. Theory

A rotation matrix in two dimensions is easy to define. There is only one axis of rotation, which is normal to the 2D plane, and a rotation of a positive angle θ (in a right-handed co-ordinate system) is given by the matrix

$$\begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}$$

Defining rotations in three dimensions is more difficult. One method of defining rotations is known as *Euler angles*, which are often used in mechanics.[12]

The method of Euler angles, specifies three angles. These are commonly called azimuth, elevation and roll (or in aeronautical terms yaw, pitch and roll). We shall label these three angles as ψ, θ and ϕ, and our object exists in a normal right-handed co-ordinate set, x, y and z.

The rotations must be applied in a specific order. A point (x, y, z) is first rotated from its original location by an angle ψ about the x axis (roll):

$$\begin{pmatrix} x' \\ y' \\ z' \end{pmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\psi & -\sin\psi \\ 0 & \sin\psi & \cos\psi \end{bmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix}$$

Then it is rotated by an angle θ about the original y axis (elevation):

$$\begin{pmatrix} x'' \\ y'' \\ z'' \end{pmatrix} = \begin{bmatrix} \cos\theta & 0 & -\sin\theta \\ 0 & 1 & 0 \\ \sin\theta & 0 & \cos\theta \end{bmatrix} \begin{pmatrix} x' \\ y' \\ z' \end{pmatrix}$$

The final rotation, ϕ is then about the original z axis (azimuth), defining the new location x''', y''' and z''' as

$$\begin{pmatrix} x''' \\ y''' \\ z''' \end{pmatrix} = \begin{bmatrix} \cos\phi & -\sin\phi & 0 \\ \sin\phi & \cos\phi & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{pmatrix} x'' \\ y'' \\ z'' \end{pmatrix}$$

B. Computing exercise

Write a MATLAB that function which takes three Euler angles and produces a single 3D rotation matrix, transforming points (x, y, z) to (x''', y''', z'''), as defined above.

Use this function to find the rotation matrix `R` which rotates a co-ordinate system by the Euler angles $(90°, 20°, 15°)$. Verify that this is a valid rotation matrix i.e. that it is orthogonal, and has a determinant of $+1$.

The point $\mathbf{p}$ is to be mapped to a new location $\mathbf{p'}$ using your matrix `R`. If $\mathbf{p} = (2, 3, 0)^T$, what is $\mathbf{p'}$? What matrix reverses this transformation?

Find the matrix `S` which represents the rotation given by $(-90°, -20°, -15°)$. To where does your transformed point $\mathbf{p'}$ map under this new rotation? Why is `S` is not the reverse of `R`?

C. Implementation notes

1. **Degrees and radians**
   Remember that MATLAB works in radians, not degrees. You might like to make use of the `sind` function you defined earlier, and also to define a `cosd` function.

D. Evaluation and marking

Write a MATLAB script which uses your Euler angle function to perform all the calculations necessary for the exercise. Show this working script, and your Euler angle function to a demonstrator for marking.