5_matrices

February 8, 2023

1 Matrices

variationalform `https://variationalform.github.io/`

Just Enough: progress at pace `https://variationalform.github.io/`

`https://github.com/variationalform`

Simon Shaw `https://www.brunel.ac.uk/people/simon-shaw`.

This document uses python

and also makes use of LaTeX

in Markdown

1.1 What this is about:

You will be introduced to …

- The **inner product** (or **dot product**, or **scalar product**) of vectors
- Matrices as a way to represent tables of numbers.
- Matrices as a way to *operate* on vectors (and other matrices)
- The arithmetic (adding and subtracting) of matrices.
- The `numpy` library (or package) for working with matrices in `python`

We'll then look at special types of matrices, and derived quantities.

As usual our emphasis will be on *doing* rather than *proving*: *just enough: progress at pace*

1.2 Assigned Reading

For this worksheet you should read Chapters 6 and 7 of [VMLS] for more introductory material on matrices, and also Appendix D of [DSML] if you want to read more about `python` and `numpy`.

- VMLS: Introduction to Applied Linear Algebra - Vectors, Matrices, and Least Squares, by Stephen Boyd and Lieven Vandenberghe, `https://web.stanford.edu/~boyd/vmls/`
- DSML: Data Science and Machine Learning, Mathematical and Statistical Methods by Dirk P. Kroese, Zdravko I. Botev, Thomas Taimre, Radislav Vaisman, `https://people.smp.uq.`

edu.au/DirkKroese/DSML and https://people.smp.uq.edu.au/DirkKroese/DSML/DSML.pdf

Further accessible material can be found in [FCLA], and advanced material is available in Chapters 2, 3 and 4 of [MML].

- MML: Mathematics for Machine Learning, by Marc Peter Deisenroth, A. Aldo Faisal, and Cheng Soon Ong. Cambridge University Press. https://mml-book.github.io.
- FCLA: A First Course in Linear Algebra, by Ken Kuttler, https://math.libretexts.org/Bookshelves/Linear_Algebra/A_First_Course_in_Linear_Algebra_(Kuttler)

All of the above can be accessed legally and without cost.

There are also these useful references for coding:

- PT: `python`: https://docs.python.org/3/tutorial
- NP: `numpy`: https://numpy.org/doc/stable/user/quickstart.html
- MPL: `matplotlib`: https://matplotlib.org

1.3 Matrices

A *matrix* is an n-row by m-column table of numbers enclosed in brackets. For example,

$$\boldsymbol{B} = \begin{pmatrix} 3 & 1 & -6.2 \\ 2 & -5 & \pi \end{pmatrix}, \qquad \boldsymbol{N} = \begin{pmatrix} -4 & 9 \\ 7/8 & 6 \\ 0 & 0 \end{pmatrix}, \qquad \boldsymbol{Z} = \begin{pmatrix} 1 & 2 & 0 \\ 1 & -2 & 1 \\ 0 & 1 & -4 \end{pmatrix},$$

are, in turn, a 2 by 3, a 3 by 2 and a 3 by 3 matrix. These are called the matrix *shape*, or *dimension*.

We will deal exclusively with matrices of **real numbers** from $\mathbb{R}$.

Complex matrices with entries from $\mathbb{C}$ can be introduced but we wont need them.

$$\boldsymbol{B} = \begin{pmatrix} 3 & 1 & -6.2 \\ 2 & -5 & \pi \end{pmatrix}, \qquad \boldsymbol{N} = \begin{pmatrix} -4 & 9 \\ 7/8 & 6 \\ 0 & 0 \end{pmatrix}, \qquad \boldsymbol{Z} = \begin{pmatrix} 1 & 2 & 0 \\ 1 & -2 & 1 \\ 0 & 1 & -4 \end{pmatrix},$$

Matrices are usually denoted by **bold** CAPITAL letters, and when $n = m$ we call the matrix *square*. For a square $n \times n$ matrix we will just use n as its dimension. So $\boldsymbol{Z}$ is square of dimension 3.

If we exchange the rows and columns of a matrix we get its *transpose*, denoted with a superscript T like so,

$$\boldsymbol{B}^T = \begin{pmatrix} 3 & 2 \\ 1 & -5 \\ -6.2 & \pi \end{pmatrix}, \qquad \boldsymbol{N}^T = \begin{pmatrix} -4 & \frac{7}{8} & 0 \\ 9 & 6 & 0 \end{pmatrix}, \qquad \boldsymbol{Z}^T = \begin{pmatrix} 1 & 1 & 0 \\ 2 & -2 & 1 \\ 0 & 1 & -4 \end{pmatrix}.$$

1.4 Using `numpy` to represent matrices

As we have seen, `numpy` is a key tool for scientific computing in `python`. See https://numpy.org

As before, we load in the numpy package and abbreviate it with `np`.

```
[3]: import numpy as np
```

Now we can set up the matrices above as `numpy` *arrays*, and print them out, as follows,

```
[4]: B = np.array([ [3,1, -6.2], [2,-5,np.pi] ])
     N = np.array([ [-4, 9],[7/8, 6],[0, 0 ] ])
     Z = np.array([ [1,2, 0],[1, -2, 1],[0, 1, -4] ])
     print('B = \n', B)
     print('N = \n', N)
     print('Z = \n', Z)
```

```
B =
 [[ 3.          1.          -6.2       ]
 [ 2.         -5.          3.14159265]]
N =
 [[-4.     9.   ]
 [ 0.875  6.   ]
 [ 0.     0.   ]]
Z =
 [[ 1  2  0]
 [ 1 -2  1]
 [ 0  1 -4]]
```

The matrix transpose just requires `.T` - like this

```
[5]: print('B.T = \n', B.T)
     print('N.T = \n', N.T)
     print('Z.T = \n', Z.T)
```

```
B.T =
 [[ 3.          2.        ]
 [ 1.         -5.        ]
 [-6.2         3.14159265]]
N.T =
 [[-4.     0.875  0.   ]
 [ 9.     6.     0.   ]]
Z.T =
 [[ 1  1  0]
 [ 2 -2  1]
 [ 0  1 -4]]
```

These agree with what we had earlier,

$$\boldsymbol{B}^T = \begin{pmatrix} 3 & 2 \\ 1 & -5 \\ -6.2 & \pi \end{pmatrix}, \qquad \boldsymbol{N}^T = \begin{pmatrix} -4 & \frac{7}{8} & 0 \\ 9 & 6 & 0 \end{pmatrix}, \qquad \boldsymbol{Z}^T = \begin{pmatrix} 1 & 1 & 0 \\ 2 & -2 & 1 \\ 0 & 1 & -4 \end{pmatrix}.$$

1.5 Addition and Subtraction

Matrices of the same dimension, or *shape*, can be added or subtracted as in,

$$\boldsymbol{B}-\boldsymbol{N}^T=\begin{pmatrix} 3 & 1 & -6.2 \\ 2 & -5 & \pi \end{pmatrix}-\begin{pmatrix} -4 & \frac{7}{8} & 0 \\ 9 & 6 & 0 \end{pmatrix}=\begin{pmatrix} 7 & \frac{1}{8} & -6.2 \\ -7 & -11 & \pi \end{pmatrix}$$

and

$$\boldsymbol{Y}=\boldsymbol{Z}+\boldsymbol{Z}^T=\begin{pmatrix} 1 & 2 & 0 \\ 1 & -2 & 1 \\ 0 & 1 & -4 \end{pmatrix}+\begin{pmatrix} 1 & 1 & 0 \\ 2 & -2 & 1 \\ 0 & 1 & -4 \end{pmatrix}=\begin{pmatrix} 2 & 3 & 0 \\ 3 & -4 & 2 \\ 0 & 2 & -8 \end{pmatrix}$$

This can be done in `numpy` exactly as expected…

[6]:
```python
print('B-N.T = \n', B-N.T)
Y = Z+Z.T
print('Y = \n', Y)
```

```
B-N.T =
 [[  7.            0.125        -6.2        ]
 [ -7.          -11.            3.14159265]]
Y =
 [[ 2  3  0]
 [ 3 -4  2]
 [ 0  2 -8]]
```

1.6 Symmetry

Notice that $\boldsymbol{Y}=\boldsymbol{Y}^T$. This means for example that $\boldsymbol{Y}-\boldsymbol{Y}^T=\boldsymbol{0}$ - the zero matrix.

[7]:
```python
print('Y = \n', Y)
print('Y.T = \n', Y.T)
print('Y-Y.T = \n', Y-Y.T)
```

```
Y =
 [[ 2  3  0]
 [ 3 -4  2]
 [ 0  2 -8]]
Y.T =
 [[ 2  3  0]
 [ 3 -4  2]
 [ 0  2 -8]]
Y-Y.T =
 [[0 0 0]
 [0 0 0]
 [0 0 0]]
```

Matrices which have this property are called *symmetric* - and this alludes to a reflection in the *leading diagonal*, that runs from top left to bottom right. We can also see that

$$\boldsymbol{Y}^T = (\boldsymbol{Z} + \boldsymbol{Z}^T)^T = \boldsymbol{Z}^T + (\boldsymbol{Z}^T)^T = \boldsymbol{Z}^T + \boldsymbol{Z} = \boldsymbol{Y}$$

and so adding a square matrix to its own transpose always produces a symmetric matrix. Some writers reserve vertically symmetric letters, such as $\boldsymbol{A}, \boldsymbol{H}, \dots$ for symmetric matrices, and non-symmetric letters, such as $\boldsymbol{B}, \boldsymbol{K}, \dots$, for non-symmetric matrices. We'll do this when it's practical.

> **THINK ABOUT:** Must a symmetric matrix be square? Can you add a non-square matrix to its own transpose? If $\boldsymbol{L} = \boldsymbol{K} + \boldsymbol{P}$ then what is $\boldsymbol{L} - \boldsymbol{P}$ equal to?

1.7 Matrix Multiplication by a Scalar

Matrices can be multiplied by a scalar: we just have to multiply every entry by that scalar. For example, for the matrix $\boldsymbol{B}$ given above, we could say $\boldsymbol{C} = 2\boldsymbol{B}$ so that,

$$\boldsymbol{C} = 2\boldsymbol{B} = 2\begin{pmatrix} 3 & 1 & -6.2 \\ 2 & -5 & \pi \end{pmatrix} = \begin{pmatrix} 6 & 2 & -12.4 \\ 4 & -10 & 2\pi \end{pmatrix}.$$

And, with `numpy` ...

```
[8]: C = 2*B
     print('C = \n', C)
```

```
C =
 [[  6.            2.          -12.4       ]
 [  4.          -10.            6.28318531]]
```

1.8 The vector dot, scalar or inner product

Before we discuss how to maultiply a matrix by a nmatrix we need to go back to vectors and introduce the *vector dot, scalar or inner product*.

Here it is:

$$\text{if } \boldsymbol{a} = \begin{pmatrix} 3 \\ -2 \\ 1 \end{pmatrix} \text{ and } \boldsymbol{p} = \begin{pmatrix} 5 \\ 2 \\ -10 \end{pmatrix}$$

then the inner product is formed like this,

$$(\boldsymbol{a}, \boldsymbol{p}) = \boldsymbol{a} \cdot \boldsymbol{p} = (3 \times 5) + (-2 \times 2) + (1 \times -10) = 15 - 4 - 10 = 1.$$

The pattern is: multiply each pair of components together and add all of the results up. The result only *exists* if the two vectors are of the same dimension.

Note that both of the notations $(\boldsymbol{a}, \boldsymbol{p})$ and $\boldsymbol{a} \cdot \boldsymbol{p}$ are in popular use.

We will use both of them. This is called the **inner product**, the **scalar product** or the ***dot product***. We'll use all of terms interchangably.

The inner, or scalar, or dot, product is going to be **very important** so let's look a bit more closely. If two vectors are the same size, $\boldsymbol{u}, \boldsymbol{v} \in \mathbb{R}^n$ for some positive integer n, then their scalar, or dot, product is the real number given by the following pattern,

$$(\boldsymbol{u}, \boldsymbol{v}) = \boldsymbol{u} \cdot \boldsymbol{v} = \boldsymbol{u}^T \boldsymbol{v} = \begin{pmatrix} u_1 \\ u_2 \\ \vdots \\ u_n \end{pmatrix}^T \begin{pmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{pmatrix} = \begin{pmatrix} u_1 & u_2 & \cdots & u_n \end{pmatrix} \begin{pmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{pmatrix} = u_1 v_2 + u_2 v_2 + \cdots + u_n v_n.$$

By using the tranpose, we see that we move horizontally along the left hand vector, and vertically down the right hand one, multiplying as we go, and adding up the results.

> **THINK ABOUT:** Is $\boldsymbol{u} \cdot \boldsymbol{v} = \boldsymbol{v} \cdot \boldsymbol{u}$? What is $\boldsymbol{u} \cdot \boldsymbol{u}$? How is $\boldsymbol{u} \cdot \boldsymbol{u}$ connected to $\|\boldsymbol{u}\|_2^2$? Write $\|\boldsymbol{u}\|_2$ in terms of $\boldsymbol{u}^T \boldsymbol{u}$.

If we think of each of two vectors $\boldsymbol{u}, \boldsymbol{v} \in \mathbb{R}^n$ as arrows pointing out from the origin, then we can think of the angle between those arrows.

Let this angle be denoted by θ, then it can be shown that

$$\boldsymbol{u} \cdot \boldsymbol{v} = \|\boldsymbol{u}\|_2 \, \|\boldsymbol{v}\|_2 \, \cos(\theta)$$

Remeber that $\boldsymbol{u} \cdot \boldsymbol{v}$, $\boldsymbol{u}^T \boldsymbol{v}$ and $(\boldsymbol{u}, \boldsymbol{v})$ all mean the same thing. **The last IS NOT a coordinate pair!**

EXERCISE: draw the vectors $\boldsymbol{u} = (5, 0)^T$ and $\boldsymbol{v} = (2, 2)^T$ emanating from the origin of $\mathbb{R}^2$. What is the angle between them? Verify the formula above.

[13]:
```python
u = np.array([5,0])
v = np.array([2,2])
costheta = u.dot(v) / ( np.linalg.norm(u)* np.linalg.norm(v) )
theta = np.arccos(costheta)
print('The angle theta = ', theta, ' radians')
print('The angle theta = ', 180*theta/np.pi, ' degrees')
```

```
The angle theta =  0.7853981633974484  radians
The angle theta =  45.00000000000001  degrees
```

$$\boldsymbol{u} \cdot \boldsymbol{v} = \|\boldsymbol{u}\|_2 \, \|\boldsymbol{v}\|_2 \, \cos(\theta)$$

> **THINK ABOUT:** Recalling that $-1 \le \cos\theta \le 1$, can you see that $-\|\boldsymbol{u}\|_2 \, \|\boldsymbol{v}\|_2 \le \boldsymbol{u} \cdot \boldsymbol{v} \le \|\boldsymbol{u}\|_2 \, \|\boldsymbol{v}\|_2$? Can you conclude that $|\boldsymbol{u} \cdot \boldsymbol{v}| \le \|\boldsymbol{u}\|_2 \, \|\boldsymbol{v}\|_2$? This result is called the **Cauchy-Schwarz inequality**.

If two vectors are at right angles to each other then $\cos\theta = 0$ and $\boldsymbol{u}\cdot\boldsymbol{v} = 0$. The vectors are then said to be **orthogonal**. If, in addition to being orthogonal the vectors have unit Pythagorean length, so that $\|\boldsymbol{u}\|_2 = \|\boldsymbol{v}\|_2 = 1$, then the vectors are said to be *orthonormal*.

For further and deeper information on vectors you can consult Chapter 4 of the introductory level text {FCLA}, `https://math.libretexts.org/Bookshelves/Linear_Algebra/A_First_Course_in_Linear_Algebra_(Kuttler)/04%3A_R`. A more sophisticated mathematical treatment of the subject can be found in Chapters 2 and 3 of {MML}, `https://mml-book.github.io`.

We can imagine stacking vectors next to each other side-by-side. Doing this produces tables of numbers - and they are what we are seeing as matrices.

The dot product is the key to matrix multiplication.

1.9 Matrix Multiplication

Matrices can be multiplied together provided that they are of compatible shape. The process uses the same pattern as the inner (or scalar, or dot) product of vectors. We go along the rows on the left and down the columns on the right. For example:

$$\text{if } \boldsymbol{J} = \begin{pmatrix} 3 & 4 \\ 2 & 1 \end{pmatrix} \text{ and } \boldsymbol{L} = \begin{pmatrix} 1 & 5 & -7 \\ -3 & -4 & 0 \end{pmatrix} \text{ then } \boldsymbol{JL} = \begin{pmatrix} -9 & -1 & -21 \\ -1 & 6 & -14 \end{pmatrix}$$

To see how this happens, lay the entire calculation out in detail like this,

$$\begin{pmatrix} 3 & 4 \\ 2 & 1 \end{pmatrix}\begin{pmatrix} 1 & 5 & -7 \\ -3 & -4 & 0 \end{pmatrix} = \begin{pmatrix} -9 & -1 & -21 \\ -1 & 6 & -14 \end{pmatrix}$$

and think about using the inner product for each row on the left of the product, $\boldsymbol{J}$, and each column on the right of the product, $\boldsymbol{L}$.

Your choice of row in $\boldsymbol{J}$ will give the results in that row on the right of the = sign, and your choice of column in $\boldsymbol{L}$ will give the results in that column on the right of the = sign.

$$\boldsymbol{JL} = \begin{pmatrix} 3 & 4 \\ 2 & 1 \end{pmatrix}\begin{pmatrix} 1 & 5 & -7 \\ -3 & -4 & 0 \end{pmatrix} = \begin{pmatrix} -9 & -1 & -21 \\ -1 & 6 & -14 \end{pmatrix}$$

For example, to get the -1 in row 1 and column 2 we see that it comes from the dot product pattern applied to the row 1 of $\boldsymbol{J}$ and column 2 of $\boldsymbol{L}$, like so

$$\begin{pmatrix} 3 & 4 \\ \cdot & \cdot \end{pmatrix}\begin{pmatrix} \cdot & 5 & \cdot \\ \cdot & -4 & \cdot \end{pmatrix} = \begin{pmatrix} \cdot & (3\times 5 + 4\times -4) & \cdot \\ \cdot & \cdot & \cdot \end{pmatrix} = \begin{pmatrix} \cdot & -1 & \cdot \\ \cdot & \cdot & \cdot \end{pmatrix}$$

Here is another way to see it,

$$\begin{pmatrix} \rightarrow & \rightarrow \\ \cdot & \cdot \end{pmatrix}\begin{pmatrix} \cdot & \downarrow & \cdot \\ \cdot & \downarrow & \cdot \end{pmatrix} = \begin{pmatrix} \cdot & \times & \cdot \\ \cdot & \cdot & \cdot \end{pmatrix}$$

Notice that we can only form the product $\boldsymbol{JL}$ if the number of columns in $\boldsymbol{J}$ equals the number of rows in $\boldsymbol{L}$. If this isn't the case then we say that $\boldsymbol{JL}$ **doesn't exist**.

THINK ABOUT: Note that $\boldsymbol{JL}$ might exist when $\boldsymbol{LJ}$ doesn't. For this reason we always need to say whether we are **pre-multiplying** or **post-multiplying**. Then pre-multiplying $\boldsymbol{L}$ by $\boldsymbol{J}$ gives $\boldsymbol{JL}$ while post-multipying gives $\boldsymbol{LJ}$ (if these products exist).

THINK ABOUT: Given a matrix $\boldsymbol{C}$, can we always form $\boldsymbol{C}^2 = \boldsymbol{CC}$? What about $\boldsymbol{C}^3$, or $\boldsymbol{C}^4$ and so on? What about if $\boldsymbol{C}$ is square?

THINK ABOUT: Suppose that $\boldsymbol{JL}$ exists. If $\boldsymbol{J}$ has p rows and $\boldsymbol{L}$ has q columns, then what is the *dimension*, or *shape*, of $\boldsymbol{JL}$?

THINK ABOUT: Given non-square matrices $\boldsymbol{P}$ and $\boldsymbol{Q}$ for which $\boldsymbol{PQ}$ exists. Does $\boldsymbol{QP}$ exist? Create two such matrices and form $(\boldsymbol{PQ})^T$ and $\boldsymbol{Q}^T\boldsymbol{P}^T$. What do you find? Do you think this is always true?

1.10 Code for matrix multiplication

Recall that

$$\text{if } \boldsymbol{J} = \left(\begin{array}{cc} 3 & 4 \\ 2 & 1 \end{array}\right) \text{ and } \boldsymbol{L} = \left(\begin{array}{rrr} 1 & 5 & -7 \\ -3 & -4 & 0 \end{array}\right) \text{ then } \boldsymbol{JL} = \left(\begin{array}{rrr} -9 & -1 & -21 \\ -1 & 6 & -14 \end{array}\right)$$

To do this with `numpy` we use the `dot` method as follows:

```
[14]: J = np.array([[3, 4],[2, 1]])
      L = np.array([[1, 5, -7],[-3, -4, 0]])
      print('JL = \n', J.dot(L))
```

```
JL =
 [[ -9  -1 -21]
 [ -1   6 -14]]
```

An alternative is to use `np.dot(J,L)` like this,

```
[15]: print('JL = \n', np.dot(J,L))
```

```
JL =
 [[ -9  -1 -21]
 [ -1   6 -14]]
```

BUT: *what we must not do is this:* `J*L`. This will give an error Try it: use this `print('JL = \n', J*L)` for example.

To see what an expression like this does consider these matrices,

$$\text{if } \boldsymbol{J} = \left(\begin{array}{cc} 3 & 4 \\ 2 & 1 \end{array}\right) \text{ and } \boldsymbol{K} = \left(\begin{array}{rr} 1 & 5 \\ -3 & -4 \end{array}\right) \text{ then } \boldsymbol{JK} = \left(\begin{array}{rr} -9 & -1 \\ -1 & 6 \end{array}\right)$$

With `numpy` this is,

```
[16]: K = np.array([[1, 5],[-3, -4]])
      print('JK = \n', J.dot(K))
```

```
JK =
 [[-9 -1]
 [-1  6]]
```

which should not be a surprise. Now let's see what `J*K` is...

[18]:
```
print('JK = \n', J*K)
```

```
JK =
 [[ 3 20]
 [-6 -4]]
```

What just happened? Can you figure it out?

BE CAREFUL of this - it can cause errors (bugs) that are hard to spot.

> **THINK ABOUT:** is there a mathematical notation for this type of product? This is called *elementwise* multiplication - why? Look up *Hadamard Product* - is it related?

1.11 Notation

It is useful to be able to refer to the *elements* in a vector or a matrix in a generic way. We do this with subscripts.

Given an n-dimensional vector $\boldsymbol{v} \in \mathbb{R}^n$, we refer to the value in position j as v_j.

Similarly, given an $m \times n$-dimensional matrix $\boldsymbol{R} \in \mathbb{R}^{m,n}$, we refer to the value in row i, column j as r_{ij} (or when confusion might arise, as $r_{i,j}$).

For example,

$$\text{if } \boldsymbol{b} = \left(\begin{array}{r} 6 \\ -3 \\ 2.5 \\ -1 \\ 0 \end{array}\right) \text{ and } \boldsymbol{N}^T = \left(\begin{array}{rrr} -4 & 7/8 & 0 \\ 9 & 6 & 0 \end{array}\right),$$

Then $b_4 = -1$ and $N_{1,2} = 7/8$.

> **THINK ABOUT** indices start at zero in `numpy` so A_{15} in maths becomes `N[0,4]` in code. **Be Careful!** this can be confusing.

1.12 The Matrix-Vector Product

This is a partcularly important form of matrix-matrix multiplication.

A column vector is no more than a matrix with one column. This means that we can multiply matrices and vectors providing they are of compatible dimension. A particularly important case is the *matrix-vector product* of the form $\boldsymbol{Bu} = \boldsymbol{f}$. For example,

$$\text{if } \boldsymbol{B} = \left(\begin{array}{rrr} 3 & -2 & 4 \\ -6 & 6 & -11 \\ 6 & 2 & 5 \end{array}\right) \text{ and } \boldsymbol{u} = \left(\begin{array}{r} 2 \\ 0 \\ -1 \end{array}\right) \text{ then } \boldsymbol{f} = \left(\begin{array}{r} 2 \\ -1 \\ 7 \end{array}\right)$$

because

$$\boldsymbol{B}\boldsymbol{u} = \begin{pmatrix} 3 & -2 & 4 \\ -6 & 6 & -11 \\ 6 & 2 & 5 \end{pmatrix} \begin{pmatrix} 2 \\ 0 \\ -1 \end{pmatrix} = \begin{pmatrix} 2 \\ -1 \\ 7 \end{pmatrix}.$$

Let's verify this with `numpy`...

```
[11]: B = np.array( [[3, -2, 4],[-6, 6, -11],[ 6, 2, 5 ]])
      u = np.array([[2], [0], [-1]])
      f = B.dot(u)
      print('f = \n', f)
```

```
f =
 [[ 2]
 [-1]
 [ 7]]
```

1.13 The Matrix Inverse

A way to build on the previous calculation is to remember that for 'ordinary' (scalar) variables b, u and f, if we have $bu = f$ then if $b \neq 0$ we can multiply both sides by b^{-1} and get $b^{-1}bu = b^{-1}f$. Of course, this is just $u = f/b$, and it works because $b^{-1}b = 1$.

For matrices we can imagine doing the same thing. If $\boldsymbol{B}\boldsymbol{u} = \boldsymbol{f}$ we might hope to be able to write $\boldsymbol{u} = \boldsymbol{B}^{-1}\boldsymbol{f}$. If we knew $\boldsymbol{B}$ and $\boldsymbol{f}$ this would then allow us to find $\boldsymbol{u} = \boldsymbol{B}^{-1}\boldsymbol{f}$.

The situation is quite complicated. First note that

$$\frac{1}{36}\begin{pmatrix} 52 & 18 & -2 \\ -36 & -9 & 9 \\ -48 & -18 & 6 \end{pmatrix} \begin{pmatrix} 3 & -2 & 4 \\ -6 & 6 & -11 \\ 6 & 2 & 5 \end{pmatrix} = \frac{1}{36}\begin{pmatrix} 36 & 0 & 0 \\ 0 & 36 & 0 \\ 0 & 0 & 36 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}.$$

This calculation also illustrates the general rule given above that if we multiply a matrix by a scalar then we should multiply every value in that matrix by that scalar.

The matrix on the right is called the identity matrix. Here it is three-by-three but such a square matrix exists for every dimension. It is always zero everywhere except for unit entries on the leading diagonal.

The identity matrix is, for us, always denoted by $\boldsymbol{I}$ and in matrix theory plays the role of 1. So, by analogy with $b^{-1}b = 1$ above, the above calculation suggests $\boldsymbol{B}^{-1}\boldsymbol{B} = \boldsymbol{I}$ where

$$\boldsymbol{B}^{-1} = \frac{1}{36}\begin{pmatrix} 52 & 18 & -2 \\ -36 & -9 & 9 \\ -48 & -18 & 6 \end{pmatrix}.$$

In this expression 36 is the value of the *determinant* of $\boldsymbol{B}$. If this value were zero then this *inverse matrix* would not be defined.

1.14 Determinant of a Square Matrix

The determinant is a difficult quantity to define and work with. It can only be defined for square matrices. If $\boldsymbol{K}$ is square its determinant is denoted by $\det(\boldsymbol{K})$.

Consider first the case of a 2×2 matrix:

$$\text{if } \boldsymbol{K} = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \text{ then } \det(\boldsymbol{K}) = ad - bc \text{ and } \boldsymbol{K}^{-1} = \frac{1}{\det(\boldsymbol{K})} \begin{pmatrix} d & -b \\ -c & a \end{pmatrix}$$

if, and only if, $\det(\boldsymbol{K}) \neq 0$. You can check that for yourself. The more general case is far more difficult to discuss so here we will merely accept this general result:

> **INVERSE MATRIX:** a square matrix $\boldsymbol{K}$ is invertible if and only if its determinant is non-zero. Its inverse satisfies $\boldsymbol{K}^{-1}\boldsymbol{K} = \boldsymbol{I}$.

In practice it is usually very very difficult to check whether or not $\det(\boldsymbol{K}) \neq 0$. However, fortunately, the application area we are working on will often give us either a good clue, or a definite answer. We'll see a few examples of this as we go along.

> **THINK ABOUT:** Create a matrix $\boldsymbol{P}$ for which $\boldsymbol{P}^{-1}$ exists (or use the one above) and investigate how $\boldsymbol{P}\boldsymbol{P}^{-1}$ and $\boldsymbol{P}^{-1}\boldsymbol{P}$ are related. Do you think this is always true?

> **THINK ABOUT:** Create another matrix $\boldsymbol{Q}$ for which both $\boldsymbol{Q}\boldsymbol{P}$ and $(\boldsymbol{Q}\boldsymbol{P})^{-1}$ exist. How does this last quantity relate to $\boldsymbol{P}^{-1}\boldsymbol{Q}^{-1}$? Do you think this is always true?

1.15 Never do this - unless you know you need to

In practice we rarely if ever need to actually obtain the inverse or the determinant of a matrix.

If we really need them we can use `numpy` as follows...

[20]:
```
print('det(B) = ', np.linalg.det(B))
# multiply the inverse by 36 to tidy up the output...
print('inverse of 36B = \n', 36*np.linalg.inv(B))
```

```
---------------------------------------------------------------------------
LinAlgError                               Traceback (most recent call last)
<ipython-input-20-8ed45cdefcef> in <module>
----> 1 print('det(B) = ', np.linalg.det(B))
      2 # multiply the inverse by 36 to tidy up the output…
      3 print('inverse of 36B = \n', 36*np.linalg.inv(B))

<__array_function__ internals> in det(*args, **kwargs)

/anaconda3/lib/python3.7/site-packages/numpy/linalg/linalg.py in det(a)
   2154     a = asarray(a)
   2155     _assert_stacked_2d(a)
-> 2156     _assert_stacked_square(a)
   2157     t, result_t = _commonType(a)
   2158     signature = 'D->D' if isComplexType(t) else 'd->d'
```

```
/anaconda3/lib/python3.7/site-packages/numpy/linalg/linalg.py in␣
↪_assert_stacked_square(*arrays)
    202         m, n = a.shape[-2:]
    203         if m != n:
--> 204             raise LinAlgError('Last 2 dimensions of the array must be␣
↪square')
    205
    206 def _assert_finite(*arrays):

LinAlgError: Last 2 dimensions of the array must be square
```

We can verify the inverse by checking that $\boldsymbol{B}\boldsymbol{B}^{-1} = \boldsymbol{I}$ as follows,

```
[13]: print('B*B^(-1) = \n', B.dot(np.linalg.inv(B)))
```

```
B*B^(-1) =
 [[ 1.00000000e+00  0.00000000e+00  0.00000000e+00]
 [-2.22044605e-16  1.00000000e+00  2.77555756e-17]
 [-2.22044605e-16  0.00000000e+00  1.00000000e+00]]
```

Notice the *small* numbers - this isn't unusual in computing. The numerical precision in the processor is limited, and so the arithmetic is not exact. This is sometimes a problem, but usually not. If we enounter it as a problem, we'll discuss it.

We ought to check that $\boldsymbol{B}^{-1}\boldsymbol{B} = \boldsymbol{I}$ also...

```
[14]: print('B^(-1)*B = \n', np.linalg.inv(B).dot(B))
```

```
B^(-1)*B =
 [[ 1.00000000e+00  6.93889390e-17 -4.85722573e-17]
 [ 0.00000000e+00  1.00000000e+00  0.00000000e+00]
 [-5.55111512e-17 -1.66533454e-16  1.00000000e+00]]
```

1.16 Review

IN the above:

- we reviewed the mathematical notion of a *matrix*.
- we saw how using `numpy` in `python` we could
  - create matrices;
  - add and subtract them, and multiply by a scalar;
  - form matrix-vector products;
  - form element-wise products

We will be building extensively on these skills in the coming weeks.

Note again that we very rarely explicitly need the inverse of a matrix or a matrix determinant. These can be easy to determine for small matrices, as we saw above, but as the matrices become larger they take a very long time to compute. Avoid them at all costs.

1.17 Exercises

1. What does `L.shape` give? (Look at the dimensions of ***L***.)
2. What does `L.ndim` give? (***L*** is two-dimensional.)
3. Examine these statements. What do they do?

- `np.ones(3)`
- `np.ones([3,1])`
- `np.ones([2,4])`
- `np.zeros([2,4])`
- `np.eye(4)`

2. What about these?

- `np.arange(4)`
- `np.arange(2,8)`
- `np.arange(2,9,2)`
- `np.linspace(4,9,num=6)`
- `np.linspace(4,9,num=11)`

3. Try these - explain the results:

- `D = L.reshape([3,2])`
- `E = L.reshape([6,])`

3. `numpy` is very powerful: see this introduction for more details https://numpy.org/doc/stable/user/absolute_beginners.html

[15]:
```
L.ndim
```

[15]:
```
2
```

[16]:
```
L.shape
```

[16]:
```
(2, 3)
```

[17]:
```
print(np.ones(3))
print(np.ones([3,1]))
print(np.ones([2,4]))
print(np.zeros([2,4]))
print(np.eye(4))
```

```
[1. 1. 1.]
[[1.]
 [1.]
 [1.]]
[[1. 1. 1. 1.]
 [1. 1. 1. 1.]]
[[0. 0. 0. 0.]
 [0. 0. 0. 0.]]
```

```
[[1. 0. 0. 0.]
 [0. 1. 0. 0.]
 [0. 0. 1. 0.]
 [0. 0. 0. 1.]]
```

[18]:
```
print(np.arange(4))
print(np.arange(2,8))
print(np.arange(2,9,2))
print(np.linspace(4,9,num=6))
print(np.linspace(4,9,num=11))
```

```
[0 1 2 3]
[2 3 4 5 6 7]
[2 4 6 8]
[4. 5. 6. 7. 8. 9.]
[4.  4.5 5.  5.5 6.  6.5 7.  7.5 8.  8.5 9. ]
```

[19]:
```
print('L = \n', L)
D = L.reshape([3,2])
print('D = \n', D)
E = L.reshape([6,])
print('E = \n', E)
```

```
L =
 [[ 1  5 -7]
 [-3 -4  0]]
D =
 [[ 1  5]
 [-7 -3]
 [-4  0]]
E =
 [ 1  5 -7 -3 -4  0]
```

1.18 Technical Notes, Production and Archiving

Ignore the material below. What follows is not relevant to the material being taught.

Production Workflow

- Finalise the notebook material above
- Clear and fresh run of entire notebook
- Create html slide show:
  - `jupyter nbconvert --to slides 5_matrices.ipynb`
- Set `OUTPUTTING=1` below
- Comment out the display of web-sourced diagrams
- Clear and fresh run of entire notebook
- Comment back in the display of web-sourced diagrams
- Clear all cell output
- Set `OUTPUTTING=0` below

- Save
- git add, commit and push to FML
- copy PDF, HTML etc to web site
    - git add, commit and push
- rebuild binder

Some of this originated from

https://stackoverflow.com/questions/38540326/save-html-of-a-jupyter-notebook-from-within-the-r

These lines create a back up of the notebook. They can be ignored.

At some point this is better as a bash script outside of the notebook

[20]:
```
%%bash
NBROOTNAME='5_matrices'
OUTPUTTING=1

if [ $OUTPUTTING -eq 1 ]; then
  jupyter nbconvert --to html $NBROOTNAME.ipynb
  cp $NBROOTNAME.html ../backups/$(date +"%m_%d_%Y-%H%M%S")_$NBROOTNAME.html
  mv -f $NBROOTNAME.html ./formats/html/

  jupyter nbconvert --to pdf $NBROOTNAME.ipynb
  cp $NBROOTNAME.pdf ../backups/$(date +"%m_%d_%Y-%H%M%S")_$NBROOTNAME.pdf
  mv -f $NBROOTNAME.pdf ./formats/pdf/

  jupyter nbconvert --to script $NBROOTNAME.ipynb
  cp $NBROOTNAME.py ../backups/$(date +"%m_%d_%Y-%H%M%S")_$NBROOTNAME.py
  mv -f $NBROOTNAME.py ./formats/py/
else
  echo 'Not Generating html, pdf and py output versions'
fi
```

```
[NbConvertApp] Converting notebook 5_matrices.ipynb to html
[NbConvertApp] Writing 631788 bytes to 5_matrices.html
[NbConvertApp] Converting notebook 5_matrices.ipynb to pdf
[NbConvertApp] Writing 61285 bytes to notebook.tex
[NbConvertApp] Building PDF
[NbConvertApp] Running xelatex 3 times: ['xelatex', 'notebook.tex', '-quiet']
[NbConvertApp] Running bibtex 1 time: ['bibtex', 'notebook']
[NbConvertApp] WARNING | bibtex had problems, most likely because there were no
citations
[NbConvertApp] PDF successfully created
[NbConvertApp] Writing 99530 bytes to 5_matrices.pdf
[NbConvertApp] Converting notebook 5_matrices.ipynb to script
[NbConvertApp] Writing 23293 bytes to 5_matrices.py
```

[ ]:
```

```