Lecture 8: Working with vectors and matrices

Dr Benjamin J. Morgan[1] and Dr Andrew R. McCluskey[1,2]

[1] *Department of Chemistry, University of Bath, email: b.j.morgan@bath.ac.uk*
[2] *Diamond Light Source, email: andrew.mccluskey@diamond.ac.uk*

November 19, 2019

Aim

This week gives a brief introduction to *vectors* and *matrices*, and using these to perform mathematical manipulations in Python.

Working with vectors and matrics is common in computational chemistry, and understanding the underlying mathematical concepts helps with understanding the algorithms that are used, and the code that implements them. Working with vectors and matrices falls within the branch of mathematics called *linear algebra*. The next two weeks aim to give you a taste of some of the mathematical properties of vectors and matrics, and why using these can be useful in computationally solving chemical problems.

If you would like to learn more about vectors, matrices, and linear algebra, I recommend the *Essence of Linear Algebra* 3Blue1Brown YouTube series[1]. Another useful resource is the *Land on Vector Spaces* series of Jupyter notebooks by Lorena Barba's group[2], which provides Jupyter-friendly tools for visualising vectors and matrix operations. If you prefer a more formal textbook, I recomment *Introduction to Linear Algebra* by Gilbert Strang[3].

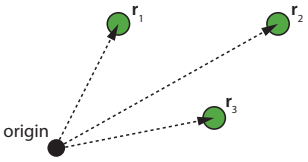
1 Vectors

Many problems in chemistry and physics involve working with *vector* quantities. A common definition of a vector in a physics context is a quantity with both *magnitude* and *direction*; for example, the positions or velocities of atoms, or the forces acting on atoms within a molecule.

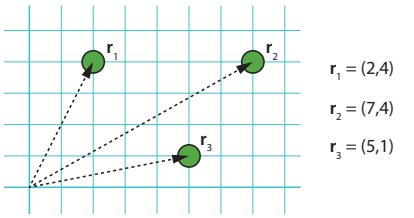
1.1 Example 1: Atomic positions

Defining atomic positions requires three pieces of information: the location of a reference position, called the *origin*, the distance of each atom from the origin, and the direction we move from the origin to reach each atom. Positions are therefore *vector* quantities (Fig. 1). A common choice for describing atomic

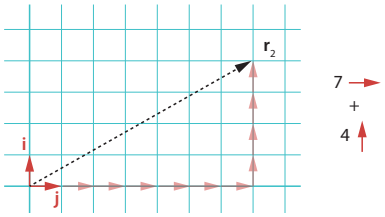
[1] `https://www.youtube.com/watch?v=fNk_zzaMoSs`
[2] `https://github.com/engineersCode/EngComp4_landlinear`
[3] `http://math.mit.edu/~gs/linearalgebra/`



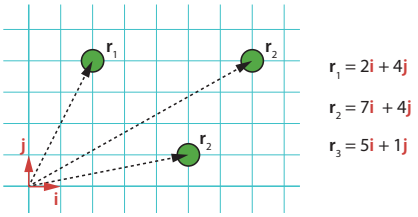
(a) Defining atomic positions requires knowing both distance and direction with respect to some reference *origin*.



(b) Describing positions using Cartesian coordinates, i.e. (x, y) coordinates in two dimensions, or (x, y, z) in three dimensions.



(c) The choice of coordinate system implicitly defines *basis* vectors, **i** and **j**. Any position can be expressed as a linear combination of these basis vectors.



(d) The (x, y) coordinates given in 1b are the coefficients used to define each position in the basis $(\mathbf{i}, \mathbf{j})$.

Figure 1

positions is to use Cartesian coordinates, i.e. (x, y) in two dimensions, or (x, y, z) in three dimensions [4]; this will be familiar from the interatomic distances (week 2) and molecular rotation (week 4) problems. In two dimensions, any position can be described by giving both the x coordinate and the y coordinate. Using the language of vectors, this choice of x and y coordinates defines a pair of *basis* vectors, which we will denote **i** and **j** [5]. **i** is a vector of length 1 pointing along x and **j** is a vector of length 1 pointing along y. Any position vector $\mathbf{r} = (x, y)$ can be expressed as a *linear combination* of these basis vectors, i.e. $\mathbf{r} = x \times \mathbf{i} + y \times \mathbf{j}$. This means one way to think about the usual notation (x, y) is that the two numbers describe the coefficients of **i** and **j** for a given linear combination. This might seem an overly complicated way of thinking about coordinates in Cartesian space, but it highlights that writing down a position vector such as $(3, 4)$ is only meaningful if the basis vectors are defined. If we had chosen a different set of basis vectors, the *same* positions would be described with *different* vectors.

[4] A good choice of coordinate system depends on the problem at hand, and Cartesian coordinated may not always be easiest to work with. For example, problems with spherical symmetry, such as describing atomic orbitals, will often be simpler when expressed in spherical coordinates (r, ϕ, θ).

[5] Vectors are usually distinguished from scalar variables by using upright bold text, e.g. **r** (used here) or an accenting arrow, e.g. $\vec{r}$. The components of a vector can be given as a list, e.g. $(1, 2)$, or as a column vector, e.g. $\begin{bmatrix} 1 \\ 2 \end{bmatrix}$. These two representations correspond to the same vector $1\vec{i} + 2\vec{j}$.

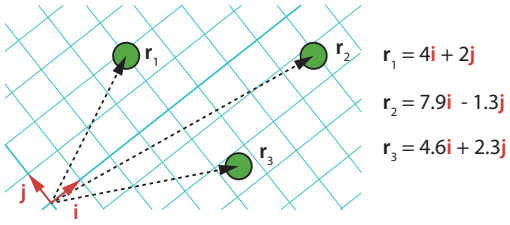


Figure 2: The three positions from Fig. 1, in a different basis.

1.2 Vector addition, subtraction, scaling, and "multiplication"

Vectors can be added together by adding the coefficients of each basis vector (Fig. 3). For example, adding together the vectors $\mathbf{v}_1 = (5, 1)$ and $\mathbf{v}_2 = (2, 4)$ gives us a new vector $\mathbf{v}_3 = (7, 6)$. This rule is explained by writing $\mathbf{v}_1$ as

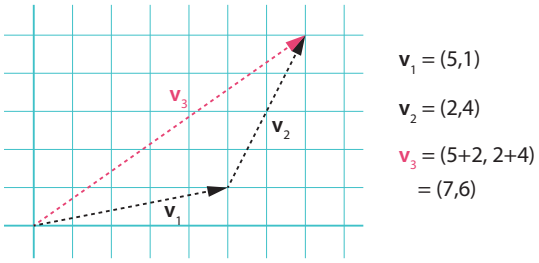


Figure 3: Vector addition,.

$(5\mathbf{i} + 1\mathbf{j})$ and $\mathbf{v}_2$ as $(2\mathbf{i} + 4\mathbf{j})$. Adding **v1** and **v2** then gives $[(5\mathbf{i} + 1\mathbf{j}) + (2\mathbf{i} + 4\mathbf{j})]$ and common terms can be collected together to give $[(5 + 2)\mathbf{i} + (1 + 4)\mathbf{j}] = (7\mathbf{i} + 6\mathbf{j})$. Subtracting one vector from another follows the same rules, but with the coefficients of each basis vector subtracted; e.g. $\mathbf{v}_1 - \mathbf{v}_2 = (3, -3)$.

Scaling a vector involves multiplying by a *scalar* (hence the name). This changes the length of the vector, but not the direction, and involves multiplying each of the basis vector coefficients by the scalar value. i.e. $\mathbf{v}_1 = (2, 3)$. $\mathbf{v}_1 \times 2 = (4, 6)$. Again, we can understand this by expanding out the original vector in terms of the basis vectors, $\mathbf{i}$ and $\mathbf{j}$: $(2\mathbf{i} + 3\mathbf{j}) \times 2 = (4\mathbf{i} + 6\mathbf{j})$.

1.2.1 The dot-product and the cross-product

We can also "multiply" two vectors together, although this is more complex. In fact there are *two* standard ways to define "multiplication" of vectors.

The *dot product* is also known as the "scalar product". This operation takes two vectors and returns a scalar quantity. The dot product of two vectors is denoted $\mathbf{a} \cdot \mathbf{b}$, and is defined as

$$\mathbf{a} \cdot \mathbf{b} = \sum_i a_i b_i = a_1 b_1 + a_2 b_2 + \ldots + a_n b_n. \quad (1)$$

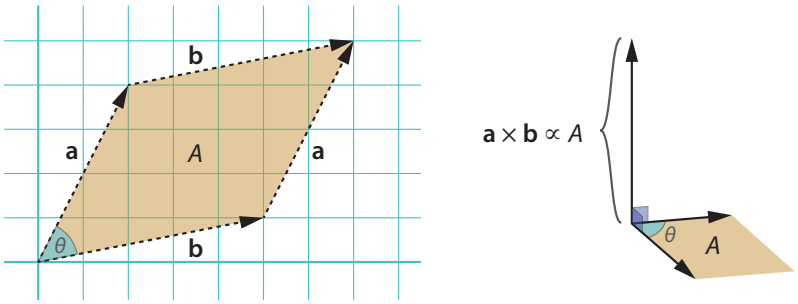


Figure 4: The cross product of **a** and **b** is proportional to the area A of the parallelogram with sides **a** and **b**.

For example, if $\mathbf{a} = (2, 1)$ and $\mathbf{b} = (3, 2)$ then $\mathbf{a} \cdot \mathbf{b}$ is given by

$$\begin{aligned} \mathbf{a} \cdot \mathbf{b} &= (x_\mathrm{a} \times x_\mathrm{b}) + (y_\mathrm{a} \times y_\mathrm{b}) && (2) \\ &= (2 \times 3 + 1 \times 2) && (3) \\ &= 8 && (4) \end{aligned}$$

An equivalent definition of the dot product is

$$\mathbf{a} \cdot \mathbf{b} = \|\mathbf{a}\| \, \|\mathbf{b}\| \cos\theta, \tag{5}$$

where $\|\mathbf{a}\|$ is the *length* of vector **a**, and θ is the angle between **a** and **b**.

The *cross-product* is also known as the "vector product". This operation takes two vectors and returns a vector quantity, with both magnitude and direction. The cross product between vectors **a** and **b** is denoted $\mathbf{a} \times \mathbf{b}$ and is defined as a vector *perpendicular* to the plane containing **a** and **b** with a length given by the parallelogram with **a** and **b** as sides (Fig. 4). This can be computed as

$$\mathbf{a} \times \mathbf{b} = \|\mathbf{a}\| \, \|\mathbf{b}\| \sin\theta \, \mathbf{n}, \tag{6}$$

where **n** is the *unit vector* (a vector with length 1) perpendicular to the plane containing **a** and **b**.

2 Working with vectors in Python

Working with vectors in Python is made simple by using `numpy` arrays.

```
import numpy as np

# define two 2D vectors using numpy arrays
>>> a = np.array([5,1])
>>> b = np.array([2,4])
>>> print(a)
[5, 1]
>>> print(b)
[2, 4]

# vector addition
```

```
>>> c = a + b
>>> print(c)
[6, 5]

# vector subtraction
>>> d = a - b
>>> print(d)
[3, -3]
```

Remember that multiplication of `numpy` arrays involves element-wise multiplication and returns a new array.

```
# vector multiplication?
>>> e = a * b
>>> print(e)
[10, 4]
```

This is *not* the same as $\mathbf{a} \cdot \mathbf{b}$ or $\mathbf{a} \times \mathbf{b}$. Instead we can use the `numpy.dot()` and `numpy.cross()` functions:

```
# dot product
>>> dot = np.dot(a,b)
>>> print(dot)
14

# cross product
>>> cross = np.cross(a,b)
>>> print(cross)
18
```

Exercise

In week 2 you wrote some code to calculate interatomic distances (and angles) between pairs of atoms in example molecules, using the expression

$$r_{ij} = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2 + (z_i - z_j)^2}. \quad (7)$$

Starting from your week 2 code, or from scratch, write a new version of this code that solves the problem using `numpy` arrays and `np.dot`.
The `molecule1.txt` and `molecule2.txt` files can be downloaded from Moodle.

3 Matrices as linear transformations

We saw previously that the elements of a vector can be thought of as the coefficients in a linear combination of basis vectors, i.e. $\begin{bmatrix} a \\ b \end{bmatrix} = (a, b) = a \times \mathbf{i} + b \times \mathbf{j}$. This means that a specific vector is only meaninful if the corresponding basis

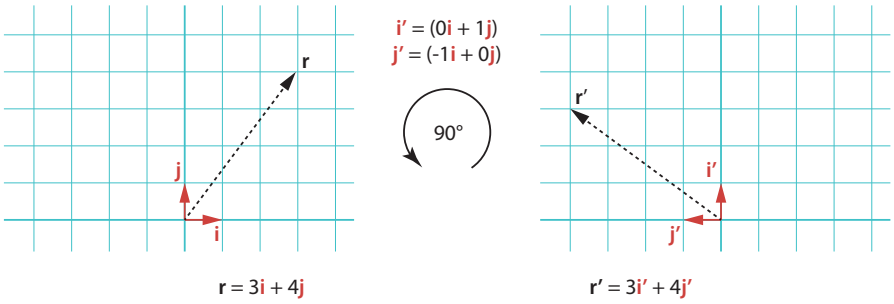


Figure 5: An example of a linear transformation: rotation by 90° anticlockwise.

is defined. A suitable choice of basis vectors depends on the problem we are interested in solving, and there are many situations where it can be useful to change from one set of basis vectors to another: for example a molecular dynamics simulation might use atomic positions, velocities, and accelerations in Cartesian coordinates, corresponding to a Cartesian basis, and usually called the *lab* frame of reference. The dynamics of individual molecules, however, might be easier to describe within a *molecular* frame of reference, with basis vectors aligned with specific bonds or along rotational axes. In this case, modelling our system involves *transforming* between these two sets of basis vectors.

As an example consider Fig. 5. This shows an initial basis with $\mathbf{i} = \begin{bmatrix}1\\0\end{bmatrix}$ and $\mathbf{j} = \begin{bmatrix}0\\1\end{bmatrix}$, and a vector $\mathbf{r}$:

$$\mathbf{r} = \begin{bmatrix}3\\4\end{bmatrix} = 3\mathbf{i} + 4\mathbf{j}.$$

We now rotate our basis by 90° anti-clockwise, which gives us a new vector $\mathbf{r}'$, which has elements $(3, 4)$ in this new, rotated, basis;

$$\mathbf{r}' = \begin{bmatrix}3\\4\end{bmatrix}' = 3\mathbf{i}' + 4\mathbf{j}',$$

where a prime symbol indicates we are in the new basis. What is the vector $\mathbf{r}'$ in the *old* basis? We can see from Fig. 5 that this should be $(-4, 3)$, but can we calculate this?

We start by expressing the *new* basis vectors, $\mathbf{i}'$ and $\mathbf{j}'$, in terms of the *old* basis vectors, $\mathbf{i}$ and $\mathbf{j}$:

$$\mathbf{i}' = (0\mathbf{i} + 1\mathbf{j}) = \begin{bmatrix}0\\1\end{bmatrix};$$

$$\mathbf{j}' = (-1\mathbf{i} + 0\mathbf{j}) = \begin{bmatrix}-1\\0\end{bmatrix}.$$

We can now expand the vector $\mathbf{r}'$ in terms of $\mathbf{i}$ and $\mathbf{j}$:

$$\begin{aligned}\mathbf{r}' &= (3\mathbf{i}' + 4\mathbf{j}')\\ &= 3(0\mathbf{i} + 1\mathbf{j}) + 4(-1\mathbf{i} + 0\mathbf{j})\\ &= -4\mathbf{i} + 3\mathbf{j}.\end{aligned}$$

We can also write this as a single *matrix* equation, where we multiply each element of our original vector $\mathbf{r}$ by the corresponding *column* of our matrix,

$$\mathbf{r}' = \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix} \mathbf{r}.$$

which looks like

$$\begin{aligned} \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 3 \\ 4 \end{bmatrix} &= 3 \begin{bmatrix} 0 \\ 1 \end{bmatrix} + 4 \begin{bmatrix} -1 \\ 0 \end{bmatrix} \\ &= \begin{bmatrix} 0 \\ 3 \end{bmatrix} + \begin{bmatrix} -4 \\ 0 \end{bmatrix} \\ &= \begin{bmatrix} -4 \\ 3 \end{bmatrix}. \end{aligned}$$

If we denote our matrix as $\mathbf{M}$, this can be written concisely as

$$\mathbf{r}' = \mathbf{M}\,\mathbf{r}.$$

3.1 Inverse matrix operations

We now have a matrix that describes a 90° anti-clockwise rotation. What if we want to *invert* this rotation, and rotate by 90° in a clockwise direction? If we follow the same procedure as above, we end up with a matrix $\mathbf{N} = \begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix}$, where

$$\mathbf{r} = \mathbf{M}\,\mathbf{r}'.$$

The matrix $\mathbf{N}$ is called the *inverse* of the matrix $\mathbf{M}$.

3.2 Matrix–matrix multiplication

What happens if we perform two matrix operations, one after the other? For example we perform *two* successive 90° rotations—giving a 180° rotation?

Mathematically this looks like

$$\begin{aligned} \mathbf{r}'' &= \mathbf{M}\,\mathbf{r}' \\ &= \mathbf{M}\,(\mathbf{M}\,\mathbf{r}) \\ &= \mathbf{M}\,\mathbf{M}\,\mathbf{r}. \end{aligned}$$

We first operate on $\mathbf{r}$ with M, to get $\mathbf{r}'$, and then operate on this vector again with $\mathbf{M}$. What is the result of this double operation, and can find a matrix that describes a general rotation by 180°?

Written out, $\mathbf{M}\,\mathbf{M}$ looks like

$$\mathbf{M}\,\mathbf{M} = \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix}.$$

We can determine the corresponsing matrix for a 180° rotation by considering what happens to the basis vectors $\mathbf{i}$ and $\mathbf{j}$ when we operate on them twice with $\mathbf{M}$.

$$\begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix} \mathbf{i} = \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

As we saw above, the first operation (working from right to left) left converts $\mathbf{i}$ to $\mathbf{i}' = \begin{bmatrix}0\\1\end{bmatrix}$. We now calculate the effect of the second operation

$$\begin{bmatrix}0 & -1\\1 & 0\end{bmatrix}\begin{bmatrix}0\\1\end{bmatrix} = 0\begin{bmatrix}0\\1\end{bmatrix} + 1\begin{bmatrix}-1\\0\end{bmatrix} = \begin{bmatrix}-1\\0\end{bmatrix}. \tag{8}$$

Doing the same for $\mathbf{j}'$ gives

$$\begin{bmatrix}0 & -1\\1 & 0\end{bmatrix}\begin{bmatrix}-1\\0\end{bmatrix} = -1\begin{bmatrix}0\\1\end{bmatrix} + 0\begin{bmatrix}-1\\0\end{bmatrix} = \begin{bmatrix}0\\-1\end{bmatrix}. \tag{9}$$

And we remember that the columns of a matrix describe the new basis functions, in this case $\mathbf{i}''$ and $\mathbf{j}''$. The matrix that describes a 180° rotation (or two successive 90° rotations) is therefore $\begin{bmatrix}-1 & 0\\0 & -1\end{bmatrix}$, i.e. both x and y are inverted.

4 Matrices in Python

Matrices can be described by multidimensional `numpy` arrays, and we can perform matrix multiplication with the `numpy.matmul()` function[6].

```
>>> r = np.array(3,4)
>>> M = np.array([[0,-1],
                  [1, 0]])
>>> r_prime = np.matmul(M,r)
>>> print(r_prime)
[-4 3]
```

The inverse of a matrix can be calculated (if it exists) using `numpy.linalg.inv()`.

```
>>> N = np.linalg.inv(M)
>>> print(N)
[[ 0.  1.]
 [-1. -0.]]
>>> np.matmul(N, r_prime)
[3, 4]
```

Matrix–matrix multiplication can also be performed with `numpy.matmul()`:

```
>>> MM = np.matmul(M,M)
>>> print(MM)
[[-1  0]
 [ 0 -1]]
```

[6] You can get the same results using `numpy.dot()`, but we encourage `numpy.matmul()` because it makes the final code easier to read and follow.

Exercise

In week 4 you wrote some code to rotate a molecule in two-dimensions. The equations for the rotation of coordinates (x, y) by β are

$$\begin{aligned} x' &= x\cos\beta - y\sin\beta \\ y' &= y\cos\beta + x\sin\beta. \end{aligned}$$

By considering the effect of this rotation on the basis vectors **i** and **j** derive the rotation matrix for a rotation around the origin by β.
Starting from your previous code, or from scratch, create a module named `matrix_transform` that includes a function `rotation`. that will take three variables `x`, `y`, and `angle`. This function will perform a rotation of `angle` on `x` and `y` to produce `x_new` and `y_new`, which are returned from the function. Use the `visualisation.show()` function to observe the rotation of the water molecule.
The `water.txt` file and the `visualisation.py` module can be downloaded from Moodle.