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].

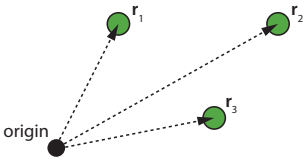
1 Vectors

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.

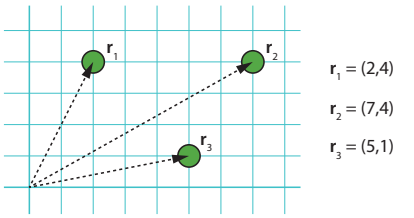
1.1 Example 1: Atomic positions

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

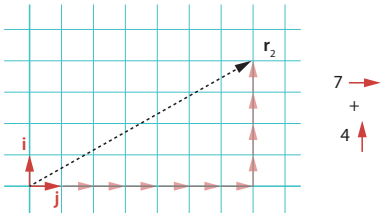
[1] `https://www.youtube.com/watch?v=fNk_zzaMoSs`
[2] `https://github.com/engineersCode/EngComp4_landlinear`
[3] `http://math.mit.edu/~gs/linearalgebra/`



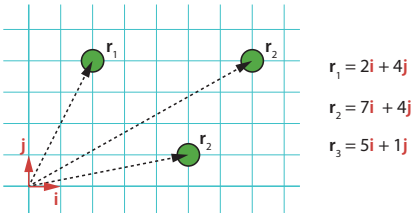
(a) Defining atomic positions requires knowing both distance and direction with respect to some reference *origin*.



(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**. **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).

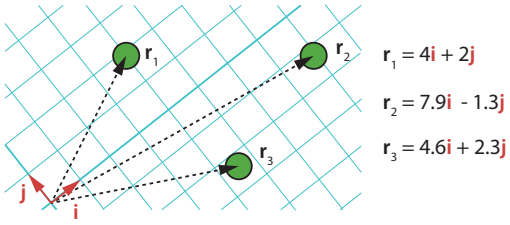


Figure 2: The three positions from Fig. 1, in a different basis.

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

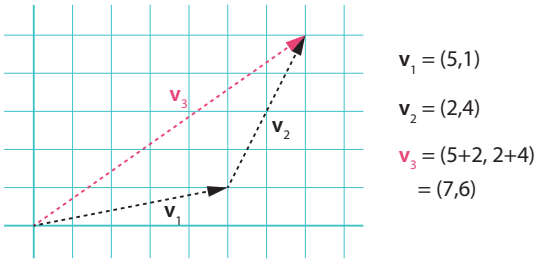


Figure 3: Vector addition,.

$(5\mathbf{i} + 1\mathbf{j})$ 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 involved multiplying by a *scalar* (hence the name). This changes the length of the vector, but not the direction, and involved 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

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

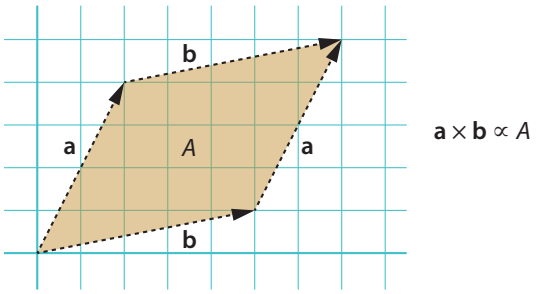


Figure 4: The cross product of **a** and **b** is proportional to the area A of the parallelogram with sides **a** and **b**.

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 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)
print(b)

# vector addition
c = a + b
print(c)
```

```
# vector subtraction
c = a - b
```

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

```
# vector multiplication?
d = a * b
print(d)
```

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)

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

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.