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 26, 2019

Aim

This week's session continues from last week's content on *vectors* and *matrices*, and how we can use Python to perform vector and matrix maths. This week we will look at eigenvalue equations, solving these using Python, and some examples of where these come up in computational chemistry.

0 Recap

Key points from last week:

0.1 Vectors

- *Vectors* are properties with both *magnitude* and *direction* (in contrast to *scalars*, which only have magnitude.)
- Vectors can be represented as a *linear combination* of basis vectors:

$$\mathbf{v} = \sum_i c_i \phi_i.$$

  e.g. for a two dimensional vector space with basis vectors $\mathbf{i}$ and $\mathbf{j}$, the column vector $\begin{bmatrix} 2 \\ 3 \end{bmatrix}$ describes a vector $2\mathbf{i} + 3\mathbf{j}$.

- In Python, we can represent vectors using one-dimensional numpy arrays, e.g.

```
>>> import numpy as np
>>> v = np.array([2, 3])
>>> print(v)
[2, 3]
```

- numpy arrays behave like vectors for addition and scalar multiplication:

```
>>> print(a*2)
[4, 6]
>>> u = np.array([3, 5])
>>> print(u + v)
[5, 8]
>>> print(u - v)
[1, 2]
```

- The dot product and cross product operations are provided as part of numpy:

```
>>> print(np.dot(u, v))
21
>>> print(np.cross(u, v))
-1
```

0.2 Matrices

- Matrices describe *linear transformations*: we generate a new set of basis vectors as a linear combination of the original basis vectors. e.g. in two dimensions, the matrix $\mathbf{M} = \begin{bmatrix} a & c \\ b & d \end{bmatrix}$ describes the basis transformation

$$\mathbf{i} \rightarrow \mathbf{i}' \tag{1}$$
$$\mathbf{j} \rightarrow \mathbf{j}'. \tag{2}$$

where

$$\mathbf{i}' = a \times \mathbf{i} + b \times \mathbf{j} \tag{3}$$
$$\mathbf{j}' = c \times \mathbf{i} + d \times \mathbf{j}. \tag{4}$$

i.e. the *columns* of the matrix $\mathbf{M}$ describe the transformation of the vectors $\begin{bmatrix} 1 \\ 0 \end{bmatrix}$ and $\begin{bmatrix} 0 \\ 1 \end{bmatrix}$.

- We can calculate the effect of a matrix $\mathbf{M}$ operating on a vector $\mathbf{v}$ by calculating the effect of the matrix operating on the basis vectors $\mathbf{i}$ and $\mathbf{j}$ and then multiplying these by the vector elements, e.g.

$$\mathbf{v}' = \mathbf{M} \cdot \mathbf{v} \tag{5}$$
$$= \begin{bmatrix} a & c \\ b & d \end{bmatrix} \cdot \begin{bmatrix} v_1 \\ v_2 \end{bmatrix} \tag{6}$$
$$= v_1 \begin{bmatrix} a \\ b \end{bmatrix} + v_2 \begin{bmatrix} c \\ d \end{bmatrix} \tag{7}$$
$$= v_1 \times \mathbf{i}' + v_2 \times \mathbf{j}'. \tag{8}$$

- In Python, we can represent vectors using one-dimensional numpy arrays, e.g.

```
>>> M = np.array([[1, 2],
                  [-1, 1]])
>>> print(M)
[[1 2]
 [-1 1]]
```

- Matix–vector multiplication can be performed using `np.matmul()`:

```
>>> print(np.matmul(M,v))
[8, 1]
```

which gives the same result as a sum over vector elements times transformed basis vectors (matrix columns), using array slicing (see week ?):

```
>>> print(v[0]*M[:,0] + v[1]*M[:,1])
[8, 1]
```

- The combined effect of one matrix operation followed by a second can be described by a single matrix. The new matrix is generated by multiplying the two original matrices together. In Python we can use the `np.matmul()` function for matrix multiplication:

$$\mathbf{M} = \begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix};$$

$$\mathbf{N} = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}.$$

$$\begin{aligned} \mathbf{P} &= \mathbf{M} \cdot \mathbf{N} \\ &= \begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix} \cdot \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} \\ &= \begin{bmatrix} 1 & 3 \\ 2 & 3 \end{bmatrix}. \end{aligned}$$

- The reverse or *inverse* of a matrix operation, $\mathbf{M}$ is described by the matrix inverse $\mathbf{M}^{-1}$. We can compute this in Python using `np.linalg.inv(M)`.

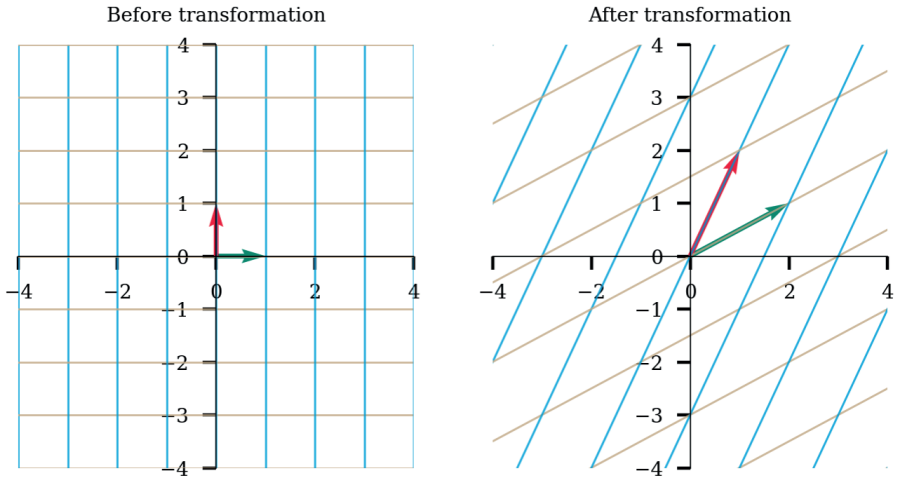

Figure 1: CAPTION

1 Eigenvalues and Eigenvectors

We have seen that a matrix $\mathbf{M}$ operating on a vector $\mathbf{v}$ produces a new vector $\mathbf{v'}$:

$$\mathbf{v}' = \mathbf{M} \cdot \mathbf{v}.$$

In general the matrix $\mathbf{M}$ can change both the magnitude and direction of the vectors $\mathbf{v} \rightarrow \mathbf{v'}$.

Consider the matrix $\mathbf{M} = \begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix}$. This gives the linear transformation shown in Figure 1. We can also visualise this transformation by considering the effect on a set of vectors with length 1 and different directions; shown in Figure 2. Now we see that the effect of the matrix transformation is to transform a unit circle into an ellipse.

The major and minor axes of this ellipse (given by the longest and shortest transformed vectors) lie along $\begin{bmatrix} 1 \\ 1 \end{bmatrix}$ and $\begin{bmatrix} -1 \\ 1 \end{bmatrix}$, respectively. If we compare these transformed vectors to the corresponding *original* vectors we see that the effect of $\mathbf{M}$ operating on these two vectors is only to scale them. Their directions have not changed (Figure 3). Mathematically, this means

$$\begin{aligned} \mathbf{v'} &= \mathbf{M} \cdot \mathbf{v} \\ s\mathbf{v} &= \mathbf{M} \cdot \mathbf{v}. \end{aligned}$$

i.e. operating on $\mathbf{v}$ by $\mathbf{M}$ gives the *same vector* $\mathbf{v}$ back, times a scalar, s. This only happens for these two "special" vectors, which we call the *eigenvectors* of the matrix $\mathbf{M}$. The scalar values s are the *eigenvalues* of the matrix $\mathbf{M}$.

We can calcuate the eigenvalues and eigenvectors of a matrix using texttt-tnp.linalg.eig()[1]:

[1] `https://docs.scipy.org/doc/numpy/reference/generated/numpy.linalg.eig.html`

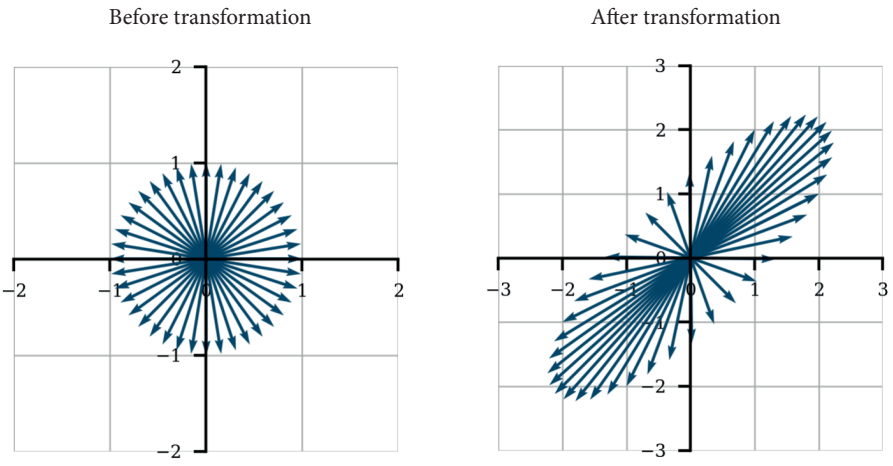

Figure 2: The effect of the matrix **M** is to transform the unit circle (left) into an ellipse (right).

```
>>> print(np.linalg.eig(M))
(array([3., 1.]), array([[ 0.70710678, -0.70710678],
       [ 0.70710678,  0.70710678]]))
```

This returns two arrays. The first array contains the eigenvalues of **M**, and the second array contains the eigenvectors of **M**.

```
>>> print(np.linalg.eig(M)[0])
[3., 1]
>>> print(np.linalg.eig(M)[1])
[[ 0.70710678 -0.70710678]
 [ 0.70710678  0.70710678]]
```

Note that the eigenvectors are returned as a matrix. Each of the *columns* of this matrix correspond to one of the eigenvectors. We can see that for the matrix $\mathbf{M} = \begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix}$ we get one eigenvector $\begin{bmatrix} 0.70710678 \\ 0.70710678 \end{bmatrix}$ with eigenvalue 3, and a second eigenvector $\begin{bmatrix} -0.70710678 \\ 0.70710678 \end{bmatrix}$ with eigenvalue 1.

Calculating eigenvalues and eigenvectors is useful in a number of chemistry applications, for example:

- Describing the rotational motions of molecules—finding principal axes of rotation and principal moments of inertia (eigenvectors and eigenvalues of the inertia matrix).
- Describing the vibrational motions of molecules—finding normal modes (eigenvectors of the dynamical matrix and their associated frequencies (eigenvectors and eigenvalues of the dynamical matrix).
- Solving the time-independent Schrödinger equation $H\Psi = E\Psi$, which is

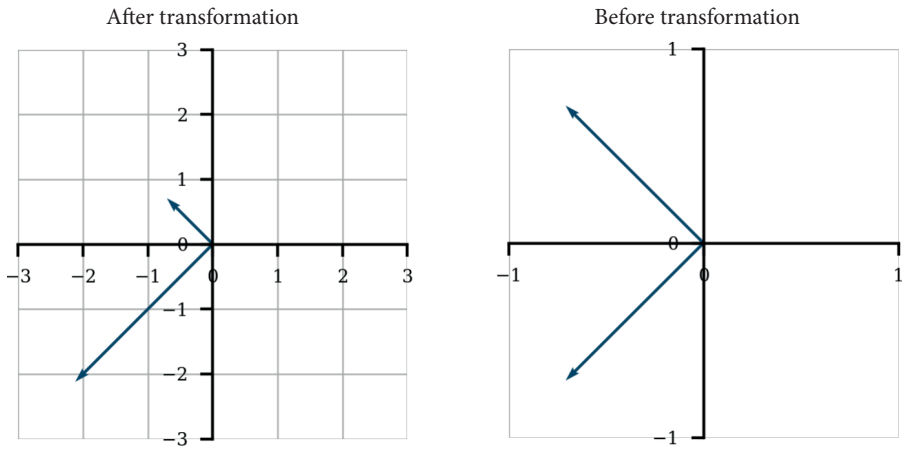

Figure 3: Left: The two vectors corresponding to the major and minor axes of the ellipse in Figure 2. Right: The same two vectors *before* the transformation. Both of these vectors have the same direction before and after the transformation by **M** and are *eigenvectors* of **M**.

an eigenvalue equation, to find molecular orbitals and their corresponding energies (eigenvectors and eigenvalues of the Hamiltonian matrix).

1.1 Key ideas

- A two-dimensional matrix transforms a unit circle into an ellipse.
- A vector that does not change direction under the operation **M** is an *eigenvector* of **M**. The scaling factor of this vector is the corresponding *eigenvalue*.

2 Principal Rotation Axes and Principal Moments of Intertia

The rotational energy levels of a molecule depend upon the molecule's moments of inertia, making these important molecular quantities for interpreting rotational spectra, or for calculating the rotational partition function:

$$q_{\text{rot}} = \frac{\pi^{\frac{1}{2}}}{\sigma}\left(\frac{k_{\text{B}}T}{hcA}\right)^{\frac{1}{2}}\left(\frac{k_{\text{B}}T}{hcB}\right)^{\frac{1}{2}}\left(\frac{k_{\text{B}}T}{hcC}\right)^{\frac{1}{2}}, \tag{9}$$

where A, B, and C and the *principal moments of inertia* of a non-linear molecule.

Moments of inertia, described by **I**, describe how a rigid body responds to angular acceleration, and are analogous to intertial mass for linear acceleration:

$$\mathbf{F} = m\mathbf{a} \qquad \boldsymbol{\tau} = \mathbf{I}\boldsymbol{\alpha}$$
$$\mathbf{p} = m\mathbf{v} \qquad \mathbf{h} = \mathbf{I}\boldsymbol{\omega}$$

If we focus on angular momentum, then for a one-dimensional problem, where rotation occurs around a single axis (e.g. for a diatomic molecule), $\boldsymbol{\omega}$ and $\mathbf{h}$ are scalars and $\mathbf{I}$ is also a scalar:

$$h = I\omega$$

For a rigid diatomic molecule the moment of inertia, I, is given by

$$I = \sum_i m_i r_i^2$$

where r_i is the distance of each atom from the centre of mass. This is often written in terms of the *reduced mass* of the molecule, μ:

$$I = \mu r^2$$

where r is the molecular bond length.

In three dimensions, however, $\boldsymbol{\omega}$, and $\mathbf{h}$ are *vectors*, and are not required to have the same orientation. The quantity $\mathbf{I}$, which translates one vector into another vector, is therefore a *matrix*, called the *intertia matrix*:

$$\begin{bmatrix} l_x \\ l_y \\ l_z \end{bmatrix} = \begin{bmatrix} I_{xx} & I_{xy} & I_{xz} \\ I_{yx} & I_{yy} & I_{yz} \\ I_{zx} & I_{zy} & I_{zz} \end{bmatrix} \cdot \begin{bmatrix} \omega_x \\ \omega_y \\ \omega_z \end{bmatrix}.$$

where the *diagonal* elements (called "moments of inertia") are similar to the one-dimensional definition, e.g.

$$I_{xx} = \sum_i m_i(y_i^2 + z_i^2),$$
$$I_{yy} = \sum_i m_i(x_i^2 + z_i^2),$$
$$I_{zz} = \sum_i m_i(x_i^2 + y_i^2);$$

but the *off-diagonal* elements (called "products of inertia") must also be included:

$$I_{xy} = \sum_i m_i x_i y_i$$

etc.

Having non-zero off-diagonal elements in the intertia matrix makes working with angular motion in 3D more complicated than in 1D. It is often useful therefore to seek a simpler form for $\mathbf{I}$ that gives a more intuitive description of molecular rotational properties.

Each of the elements of $\mathbf{I}$ contains terms like x_i^2 or xy, where x, y, and z are distances of atoms along each direction from the centre of mass. The values of each element of $\mathbf{I}$ therefore depend on the relative orientation of the $\{x, y, z\}$ coordinate system and the molecule. Choosing a differently oriented coordinate system changes the x, y, and z coordinates of each atom, and changes the values of the elements of $\mathbf{I}$. We might therefore ask: is there a choice of

molecular orientation / coordinate orientation where the off-diagonal elements of $\mathbf{I}$ are all zero?

If this were the case, our angular momentum equation would now be:

$$\begin{bmatrix} l_x \\ l_y \\ l_z \end{bmatrix} = \begin{bmatrix} I_{xx} & 0 & 0 \\ 0 & I_{yy} & 0 \\ 0 & 0 & I_{zz} \end{bmatrix} \cdot \begin{bmatrix} \omega_x \\ \omega_y \\ \omega_z \end{bmatrix}.$$

and each component of $\mathbf{l}$ would be directly proportional to the corresponding component of $\mathbf{w}$. The "products of inertia" now do not appear, and we can describe the rotational properties of the molecule by specifying the three moments of inertia.

It turns out that such a coordinate system can be found, and it corresponds to choosing the *eigenvectors* of $\mathbf{I}$ as our basis vectors. Let us suppose that we have found the eigenvalues, λ_i, and eigenvectors, $\mathbf{v}_i$, of $\mathbf{I}$. This gives us three eigenvalue equations:

$$\begin{aligned} \lambda_1 \mathbf{v}_1 &= \mathbf{I} \cdot \mathbf{v}_1 \\ \lambda_2 \mathbf{v}_2 &= \mathbf{I} \cdot \mathbf{v}_2 \\ \lambda_3 \mathbf{v}_3 &= \mathbf{I} \cdot \mathbf{v}_3. \end{aligned}$$

Let us now express our angular velocity vector $\boldsymbol{\omega}$ using these eigenvectors as a basis:

$$\boldsymbol{\omega} = c_1 \mathbf{v}_1 + c_2 \mathbf{v}_2 + c_3 \mathbf{v}_3.$$

And then the effect of the matrix $\mathbf{I}$ operating on $\boldsymbol{\omega}$:

$$\begin{aligned} \mathbf{I} \cdot \boldsymbol{\omega} &= c_1 \mathbf{I} \cdot \mathbf{v}_1 + c_2 \mathbf{I} \cdot \mathbf{v}_2 + c_3 \mathbf{I} \cdot \mathbf{v}_3 \\ &= c_1 \lambda_1 \mathbf{v}_1 + c_2 \lambda_1 \mathbf{v}_2 + c_3 \lambda_1 \mathbf{v}_3. \end{aligned}$$

which can be rewritten in matrix form as

$$\begin{aligned} \mathbf{I} \cdot \boldsymbol{\omega} &= \begin{bmatrix} \lambda_1 & 0 & 0 \\ 0 & \lambda_2 & 0 \\ 0 & 0 & \lambda_3 \end{bmatrix} \cdot \begin{bmatrix} c_1 \\ c_2 \\ c_3 \end{bmatrix} \\ &= \begin{bmatrix} \lambda_1 & 0 & 0 \\ 0 & \lambda_2 & 0 \\ 0 & 0 & \lambda_3 \end{bmatrix} \cdot \boldsymbol{\omega} \end{aligned}$$

This gives the simplified form that we wanted, *if* we use the eigenvectors of $\mathbf{I}$ as our basis vectors, i.e. as our coordinate system vectors. The eigenvectors of the inertia matrix are called the *principal rotation axes* of our molecule, and the corresponding eigenvalues are the *principal moments of inertia*.

For a molecular in an arbitrary orientation, we can therefore calculate the principal moments of inertia by finding the eigenvalues of the inertia matrix. The corresponding eigenvectors give us the principal rotation axes, which are often used to define a "natural" coordinate system for the molecule.

2.1 Rotating molecules onto their principal rotation axes

. If the principal rotation axes of a molecule define a natural coordinate system for that molecule, are we able to rotate an arbitrarily oriented molecule onto this

coordinate system? The answer is yes, and it combines the rotataion matrices from week 8 with the eigenvectors discussed this week.

For a given molecular rotation, the basis vectors **i**, **j**, and **k** point along x, y, and z respectively. But we would like to rotate our molecule (or equivalently to rotate the coordinate system) so that x, y, and z point along the principal rotation axes of the molecule. This is equivalent to *transforming* our coordinate system so that **i**, **j**, and **k** are aligned with the eigenvectors of the inertia matrix **I** (for the original coordinate system). Remember that a matrix transformation maps the original basis vectors onto the columns of the matrix. To map **i**, **j**, and **k** onto $\mathbf{v}_1$, $\mathbf{v}_2$, and $\mathbf{v}_3$ we operate on the vector coordinates of the atoms in the molecule with a matrix composed of **k** onto $\mathbf{v}_1$, $\mathbf{v}_2$, and $\mathbf{v}_3$ as columns.

Exercise

Today's exercise is to calculate principal moments of inertia and principal rotation axes of a series of molecules, and then to rotate these molecules so that they the principal axes are aligned with $\{x, y, z\}$.
The coordinates of each molecule can be downloaded from Moodle, where you will also find a `visualisation2.py` module that contains some code for visualising your molecules and their orientations.
In terms of planning your code, you want to be able to perform the following sequence of steps:

1. Each file contains data in the format $\{x, y, z, m\}$ for each atom, where m is the atomic mass of that atom. You will first want to read this in, and extract the atomic coordinates and masses.
2. Construct the inertia matrix **I**.
3. Calculate the eigenvalues and eigenvectors of **I**.
4. Operate on your atomic coordinates to generate new coordinates, with the molecular principal axes aligned with $\{x, y, z\}$.
5. Use the `visualisation.show()` function to look at your rotated molecules. Do these look how you would expect from what you know about molecular symmetry?