**Supplemental material – PHYS 6260**
Prof. John Wise
*Newman Chapter 6: Solution of Linear and Non-linear Equations*

---

# 6.1: Simultaneous Linear Equations

Systems of linear equations can be found in many physics problems. They are well understood but tedious to solve, making it a prime candidate for a computational problem. Let's look at an example,

$$\begin{aligned} 2w + x + 4y + z &= -4 && (1)\\ 3w + 4x - y - z &= 3 && (2)\\ w - 4x + y + 5x &= 9 && (3)\\ 2w - 2x + y + 3z &= 7 && (4) \end{aligned}$$

which can be expressed in matrix form, $\mathbf{Ax} = \mathbf{v}$, or

$$\begin{pmatrix} 2 & 1 & 4 & 1 \\ 3 & 4 & -1 & -1 \\ 1 & -4 & 1 & 5 \\ 2 & -2 & 1 & 3 \end{pmatrix} \begin{pmatrix} w \\ x \\ y \\ z \end{pmatrix} = \begin{pmatrix} -4 \\ 3 \\ 9 \\ 7 \end{pmatrix} \quad (5)$$

In linear algebra, we learn how to solve this equation by inverting the matrix and multiplying both sides by it. However, matrix inversion is cumbersome and complicated, especially numerically, thus making it inefficient. Furthermore, the process does not well scale to very large matrices. The most straightforward process of solving linear systems is called **Gaussian elimination**.

## Gaussian elimination

This method alters the system in a repetitive fashion to reduce the matrix into an upper triangular one. It uses two main rules that do not change the solution:

1. We can multiply any of the equations (i.e. both left and right sides) by a constant.
2. We can add or subtract any multiple of one row to another row.

Let's apply these rules to the previous example and reduce it to have zeros belong the diagonal through the following steps.

1. Divide the first row by the top-left element so that it equals one.

$$\begin{pmatrix} 1 & 0.5 & 2 & 0.5 \\ 3 & 4 & -1 & -1 \\ 1 & -4 & 1 & 5 \\ 2 & -2 & 1 & 3 \end{pmatrix} \begin{pmatrix} w \\ x \\ y \\ z \end{pmatrix} = \begin{pmatrix} -2 \\ 3 \\ 9 \\ 7 \end{pmatrix} \tag{6}$$

2. Subtract a multiple of the first row from the second row so that the first column equals zero.

$$\begin{pmatrix} 1 & 0.5 & 2 & 0.5 \\ 0 & 2.5 & -7 & -2.5 \\ 1 & -4 & 1 & 5 \\ 2 & -2 & 1 & 3 \end{pmatrix} \begin{pmatrix} w \\ x \\ y \\ z \end{pmatrix} = \begin{pmatrix} -2 \\ 9 \\ 9 \\ 7 \end{pmatrix} \tag{7}$$

3. Repeat step #2 for the third and fourth rows.

$$\begin{pmatrix} 1 & 0.5 & 2 & 0.5 \\ 0 & 2.5 & -7 & -2.5 \\ 0 & -4.5 & -1 & 5 \\ 0 & -3 & -3 & 2 \end{pmatrix} \begin{pmatrix} w \\ x \\ y \\ z \end{pmatrix} = \begin{pmatrix} -2 \\ 9 \\ 11 \\ 11 \end{pmatrix} \tag{8}$$

4. Divide the second row by its second element so that it equals one.

$$\begin{pmatrix} 1 & 0.5 & 2 & 0.5 \\ 0 & 1 & -2.8 & -1 \\ 0 & -4.5 & -1 & 5 \\ 0 & -3 & -3 & 2 \end{pmatrix} \begin{pmatrix} w \\ x \\ y \\ z \end{pmatrix} = \begin{pmatrix} -2 \\ 3.6 \\ 11 \\ 11 \end{pmatrix} \tag{9}$$

5. Subtract a multiple of the second row from the third and fourth rows so that the second column is zero.

$$\begin{pmatrix} 1 & 0.5 & 2 & 0.5 \\ 0 & 1 & -2.8 & -1 \\ 0 & 0 & -13.6 & 0 \\ 0 & 0 & -11.4 & -1 \end{pmatrix} \begin{pmatrix} w \\ x \\ y \\ z \end{pmatrix} = \begin{pmatrix} -2 \\ 3.6 \\ 27.2 \\ 21.8 \end{pmatrix} \tag{10}$$

6. Repeat for the third and fourth rows to make their diagonal element equal to one and the values to the left zero.

$$\begin{pmatrix} 1 & 0.5 & 2 & 0.5 \\ 0 & 1 & -2.8 & -1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} w \\ x \\ y \\ z \end{pmatrix} = \begin{pmatrix} -2 \\ 3.6 \\ -2 \\ 1 \end{pmatrix} \tag{11}$$

## Backsubstitution

Now that we have an upper triangular matrix, it's straightforward to start with the lower row and work our way upwards in the matrix, solving for all of the unknowns. Suppose if

we have a matrix in the following form,

$$\begin{pmatrix} 1 & a_{01} & a_{02} & a_{03} \\ 0 & 1 & a_{12} & a_{13} \\ 0 & 0 & 1 & a_{23} \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} w \\ x \\ y \\ z \end{pmatrix} = \begin{pmatrix} v_0 \\ v_1 \\ v_2 \\ v_3 \end{pmatrix} \tag{12}$$

The solution to the $\mathbf{x}$ can be shown to be

$$z = v_3 \tag{13}$$
$$y = v_2 - a_{23}z \tag{14}$$
$$x = v_1 - a_{12}y - a_{13}z \tag{15}$$
$$w = v_0 - a_{01}x - a_{02}y - a_{03}z \tag{16}$$

## Example 6.1: Gaussian elimination with backsubstitution

Now that we have an algorithm for solving a linear system of equations, let's write a program for it.

```
import numpy as np

A = np.array([[2, 1, 4, 1],
              [3, 4, -1, -1],
              [1, -4, 1, 5],
              [2, -2, 1, 3]], float)
v = np.array([-4, 3, 9, 7], float)
N = len(v)

# Gaussian elimination
for m in range(N):
    # Divide by the diagonal element
    div = A[m,m]
    A[m,:] /= div
    v[m] /= div

    # Now subtract from the lower rows
    for i in range(m+1, N):
        mult = A[i,m]
        A[i,:] -= mult*A[m,:]
        v[i] -= mult*v[m]

# Backsubstitution
x = np.zeros(N, float)
```

```
for m in range(N-1, -1, -1):
    x[m] = v[m]
    for i in range(m+1, N):
        x[m] -= A[m,i] * x[i]

print(x)
```

## Pivoting

Suppose that the system that we are solving has $a_{00} = 0$, making the first step of dividing out the $a_{00}$ impossible. We can simply swap two rows of the matrix and the right-hand side because we are free to write the system in any order. For example,

$$\begin{pmatrix} 0 & 1 & 4 & 1 \\ 3 & 4 & -1 & -1 \\ 1 & -4 & 1 & 5 \\ 2 & -2 & 1 & 3 \end{pmatrix} \begin{pmatrix} w \\ x \\ y \\ z \end{pmatrix} = \begin{pmatrix} -4 \\ 3 \\ 9 \\ 7 \end{pmatrix} \tag{17}$$

turns into

$$\begin{pmatrix} 3 & 4 & -1 & -1 \\ 0 & 1 & 4 & 1 \\ 1 & -4 & 1 & 5 \\ 2 & -2 & 1 & 3 \end{pmatrix} \begin{pmatrix} w \\ x \\ y \\ z \end{pmatrix} = \begin{pmatrix} 3 \\ -4 \\ 9 \\ 7 \end{pmatrix} \tag{18}$$

Pivoting might introduce problems down the line where the following rows might have the same problem. There exists a method called *partial pivoting*. If $m^{\text{th}}$ element in the $m^{\text{th}}$ row is zero, we compare it to the upcoming rows and swap it with the one with the $m^{\text{th}}$ element that has the largest absolute value. This ensures that the element we divide by in our Gaussian elimination is always as far from zero as possible.

## LU Decomposition

In computational physics problems, it is common for the **A** matrix to remain constant but the right-hand side **v** change. Although Gaussian elimination is a reliable method to solve the system, it can be computationally inefficient to solve it every time. Instead of performing this operation many times, we can "store" the solution method inside a set of matrices. This method is called **LU decomposition**.

Suppose that we have a matrix

$$\mathbf{A} = \begin{pmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & a_{12} & a_{13} \\ a_{20} & a_{21} & a_{22} & a_{23} \\ a_{30} & a_{31} & a_{32} & a_{33} \end{pmatrix} \tag{19}$$

The first step in Gaussian elimination can be nicely written as

$$\mathbf{L}_0 \begin{pmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & a_{12} & a_{13} \\ a_{20} & a_{21} & a_{22} & a_{23} \\ a_{30} & a_{31} & a_{32} & a_{33} \end{pmatrix} = \begin{pmatrix} 1 & b_{01} & b_{02} & b_{03} \\ 0 & b_{11} & b_{12} & b_{13} \\ 0 & b_{21} & b_{22} & b_{23} \\ 0 & b_{31} & b_{32} & b_{33} \end{pmatrix} \tag{20}$$

where

$$\mathbf{L}_0 = \frac{1}{a_{00}} \begin{pmatrix} 1 & 0 & 0 & 0 \\ -a_{10} & a_{00} & 0 & 0 \\ -a_{20} & 0 & a_{00} & 0 \\ -a_{30} & 0 & 0 & a_{00} \end{pmatrix} \tag{21}$$

The second step in Gaussian elimination can similarly be expressed as

$$\mathbf{L}_1 \begin{pmatrix} 1 & b_{01} & b_{02} & b_{03} \\ 0 & b_{11} & b_{12} & b_{13} \\ 0 & b_{21} & b_{22} & b_{23} \\ 0 & b_{31} & b_{32} & b_{33} \end{pmatrix} = \begin{pmatrix} 1 & c_{01} & c_{02} & c_{03} \\ 0 & 1 & c_{12} & c_{13} \\ 0 & 0 & c_{22} & c_{23} \\ 0 & 0 & c_{32} & c_{33} \end{pmatrix} \tag{22}$$

where

$$\mathbf{L}_1 = \frac{1}{b_{11}} \begin{pmatrix} b_{11} & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & -b_{21} & b_{11} & 0 \\ 0 & -b_{31} & 0 & b_{11} \end{pmatrix} \tag{23}$$

For a $4 \times 4$ matrix considered here, there are two more $\mathbf{L}$ matrices to complete Gaussian elimination,

$$\mathbf{L}_2 = \frac{1}{c_{22}} \begin{pmatrix} c_{22} & 0 & 0 & 0 \\ 0 & c_{22} & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & -c_{32} & c_{22} \end{pmatrix}, \quad \mathbf{L}_3 = \frac{1}{d_{33}} \begin{pmatrix} d_{33} & 0 & 0 & 0 \\ 0 & d_{33} & 0 & 0 \\ 0 & 0 & d_{33} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{24}$$

Now we have everything needed to solve the system, $\mathbf{A}\mathbf{x} = \mathbf{v}$,

$$\mathbf{L}_3\mathbf{L}_2\mathbf{L}_1\mathbf{L}_0\mathbf{A}\mathbf{x} = \mathbf{L}_3\mathbf{L}_2\mathbf{L}_1\mathbf{L}_0\mathbf{v} \tag{25}$$

which is equivalent to how we solved the previous example with Gaussian elimination. The advantage is that we have stored the solution in the product of these four matrices and can apply this to any $\mathbf{v}$, given the same left-hand side $\mathbf{A}$.

We can express LU decomposition as the product of two matrices: a (L)ower triangular matrix and an (U)pper triangular matrix,

$$\mathbf{L} \equiv \mathbf{L}_0^{-1}\mathbf{L}_1^{-1}\mathbf{L}_2^{-1}\mathbf{L}_3^{-1}, \qquad \mathbf{U} \equiv \mathbf{L}_3\mathbf{L}_2\mathbf{L}_1\mathbf{L}_0\mathbf{A}, \tag{26}$$

Multiplying these matrices together, we find

$$\mathbf{L}\mathbf{U} = \mathbf{A} \tag{27}$$

so that we can solve the system with

$$\mathbf{LUx} = \mathbf{v} \tag{28}$$

For completeness, here are the inverse of the $\mathbf{L}$ matrices even though it is straightforward to calculate them.

$$\mathbf{L}_0^{-1} = \begin{pmatrix} a_{00} & 0 & 0 & 0 \\ a_{10} & 1 & 0 & 0 \\ a_{20} & 0 & 1 & 0 \\ a_{30} & 0 & 0 & 1 \end{pmatrix}, \quad \mathbf{L}_1^{-1} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & b_{11} & 0 & 0 \\ 0 & b_{21} & 1 & 0 \\ 0 & b_{31} & 0 & 1 \end{pmatrix} \tag{29}$$

$$\mathbf{L}_2^{-1} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & c_{22} & 0 \\ 0 & 0 & c_{32} & 1 \end{pmatrix}, \quad \mathbf{L}_3^{-1} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & d_{33} \end{pmatrix}. \tag{30}$$

Multiplying these matrices together, we find $\mathbf{L}$ a lower triangular matrix, which is easy to calculate from the coefficients that we have calculated previously.

$$\mathbf{L} \equiv \mathbf{L}_0^{-1}\mathbf{L}_1^{-1}\mathbf{L}_2^{-1}\mathbf{L}_3^{-1} = \begin{pmatrix} a_{00} & 0 & 0 & 0 \\ a_{10} & b_{11} & 0 & 0 \\ a_{20} & b_{21} & c_{22} & 0 \\ a_{30} & b_{31} & c_{32} & d_{33} \end{pmatrix}, \tag{31}$$

LU decomposition is the most common method to solve a system of linear equations, and the user must incorporate (partial) pivoting when arranging the system. Because it's a common operation, most numerical Python modules, including `numpy`, has routines to solve systems of equations with LU decomposition. In `numpy`, this would be accomplished with one function call,

```
from numpy import linalg
x = linalg.solve(A,v)
```

where `A` and `v` are defined in Example 6.1 as arrays. Additionally, the `linalg` module has routines for other matrix operations, including taking the inverse, which is called `linalg.inv`.

There are two special cases of linear systems of equations that require less work than a typical LU decomposition. They are

**Tridiagonal matrices** that have a diagonal component and two adjacent diagonals with the form

$$\mathbf{A} = \begin{pmatrix} a_{00} & a_{01} & & & \\ a_{10} & a_{11} & a_{12} & & \\ & a_{21} & a_{22} & a_{23} & \\ & & a_{32} & a_{33} & a_{34} \\ & & & a_{43} & a_{44} \end{pmatrix} \tag{32}$$

**Banded matrices** that have more than three diagonals with zeros everywhere else.

## Example 6.2: Vibration in a 1D system

Consider a set of $N$ identical masses in a row, joined by identical linear systems. For simplicity, let's ignore gravity; consider that they are floating in outer space. The motion of the masses can be used as a simple model of atomic vibrations in a solid. Here we examine the modes of horizontal vibration of the system.

Let's denote the displacement of the $i^{\text{th}}$ mass relative to its rest position by $\zeta_1$. Then the equation of motion are given by Newton's second law:

$$m\frac{d^2\zeta_i}{dt^2} = k(\zeta_{i+1} - \zeta_i) + k(\zeta_{i-1} - \zeta_i) + F_i, \tag{33}$$

where $m$ is the mass and $k$ is the spring constant. The extra term $F_i$ denotes any external force imposed on mass $i$. The only exceptions to the equation are the masses at the ends:

$$m\frac{d^2\zeta_1}{dt^2} = k(\zeta_2 - \zeta_1) + F_1 \tag{34}$$

$$m\frac{d^2\zeta_N}{dt^2} = k(\zeta_{N-1} - \zeta_N) + F_N \tag{35}$$

Now suppose that we apply an external driving force $F_1 = Ce^{i\omega t}$ to the first mass only, where $C$ is a constant[1].

The net result of the applied force is that all of the masses oscillate with some frequency at some amplitude that can be expressed as the following:

$$\zeta_i(t) = x_i e^{i\omega t} \tag{36}$$

Substituting this solution into the equations of motion, we find that

$$\begin{aligned} -m\omega^2 x_1 &= k(x_2 - x_1) + C, & (37)\\ -m\omega^2 x_i &= k(x_{i+1} - x_i) + k(x_{i-1} - x_i), & (38)\\ -m\omega^2 x_N &= k(x_{N-1} - x_N) & (39) \end{aligned}$$

which can be rearranged, using $\alpha = 2k - m\omega^2$, as

$$\begin{aligned} (\alpha - k)x_1 - kx_2 &= C, & (40)\\ \alpha x_i - kx_{i-1} - kx_{x+1} &= 0, & (41)\\ (\alpha - k)x_N - kx_{N-1} &= 0. & (42) \end{aligned}$$

This is exactly a tridiagonal matrix which we can solve with LU decomposition or Gaussian elimination. A code that solves this system can be written as

[1] $e^{i\phi} = \cos\phi + i\sin\phi$ is known as Euler notation.

```
import numpy as np
from numpy import linalg
import matplotlib.pyplot as plt

# Constants
N = 26
C = 1.0
m = 1.0
k = 6.0
omega = 2.0
alpha = 2*k-m*omega*omega

# Set up the initial values of the arrays
A = np.zeros([N,N],float)
for i in range(N-1):
    A[i,i] = alpha
    A[i,i+1] = -k
    A[i+1,i] = -k
A[0,0] = alpha - k
A[N-1,N-1] = alpha - k

v = np.zeros(N,float)
v[0] = C

x = linalg.solve(A,v)

# Make a plot using both dots and lines
plt.plot(x)
plt.plot(x,"ko")
plt.xlabel("Mass Number")
plt.ylabel("Amplitude $x_i$")
plt.savefig("12_spring.png")
```

producing the following plot of amplitudes of the 26 masses.

## 6.2: Eigenvalues and Eigenvectors

Similar to solving linear systems of equations is the determination of eigenvalues and eigenfunctions, which are common quantities in classical mechanics, quantum mechanics, and electromagnetism. Consider a symmetric matrix $\mathbf{A}$. A corresponding eigenvector $\mathbf{v}$ is one that satisfies

$$\mathbf{A}\mathbf{v} = \lambda\mathbf{v} \tag{43}$$

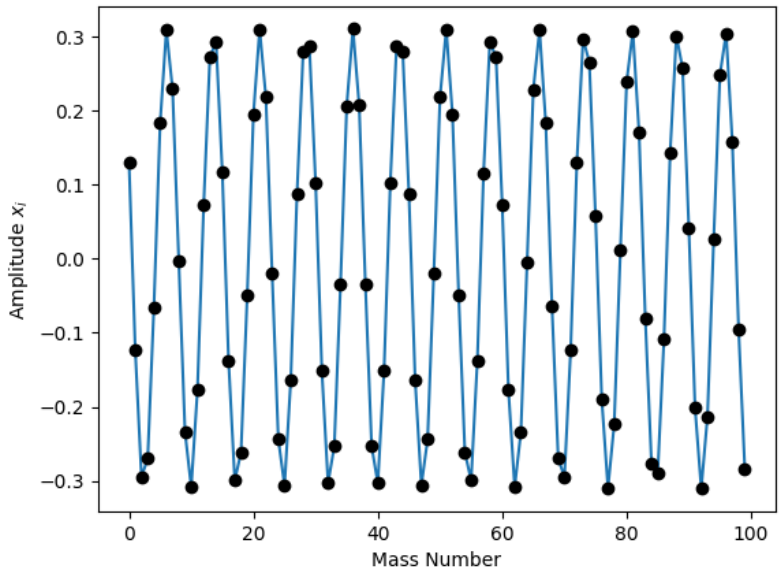


Figure 1: Amplitudes of vibration in a chain of identical masses coupled by springs

with $\lambda$ as the eigenvalue. For a $N \times N$ matrix, there are $N$ eigenvectors and $N$ eigenvalues. The eigenvectors are orthogonal to each other and are usually normalized to be unit vectors.

To calculate these quantities, we can put the $N$ eigenvectors as columns in a single matrix $\mathbf{V}$ and combine all of the equations $\mathbf{A}\mathbf{v}_i = \lambda_i \mathbf{v}_i$ into a single matrix equation,

$$\mathbf{AV} = \mathbf{VD}, \tag{44}$$

where $\mathbf{D}$ is a diagonal matrix with the eigenvalues as the elements. Because $\mathbf{V}$ is orthogonal, then $\mathbf{V}^T\mathbf{V}$ equals the identity matrix, $\mathbf{I}$. Here the superscript $T$ denotes the transpose of the matrix. To solve this system, we use the method called **QR decomposition**, that solves for the matrices $\mathbf{V}$ and $\mathbf{D}$. To do so, we decompose $\mathbf{A}$ into a matrix $\mathbf{Q}$ that is orthonormal and another matrix $\mathbf{R}$ that is an upper triangular matrix,

$$\mathbf{A} = \mathbf{Q}_1\mathbf{R}_1. \tag{45}$$

Because $\mathbf{Q}$ is orthonormal, we can multiply by its transpose to obtain $\mathbf{R}_1$,

$$\mathbf{Q}_1^T\mathbf{A} = \mathbf{Q}_1^T\mathbf{Q}\mathbf{R}_1 = \mathbf{R}_1 \tag{46}$$

Now, let's define a new matrix that is just the reverse product of Equation (45)

$$\mathbf{A}_1 = \mathbf{R}_1\mathbf{Q}_1. \tag{47}$$

Combining these two equations, we can calculate $\mathbf{A}_1$,

$$\mathbf{A}_1 = \mathbf{Q}_1^T\mathbf{A}\mathbf{Q}_1, \tag{48}$$

which reduces the magnitude of the off-diagonal elements. We repeat this process until the maximum of these elements is less than some user-defined tolerance. Given $k$ iterations, we would perform the following operations,

$$\begin{aligned}
\mathbf{A}_1 &= \mathbf{Q}_1^T \mathbf{A} \mathbf{Q}_1 && (49)\\
\mathbf{A}_2 &= \mathbf{Q}_2^T \mathbf{Q}_1^T \mathbf{A} \mathbf{Q}_1 \mathbf{Q}_2 && (50)\\
\mathbf{A}_3 &= \mathbf{Q}_3^T \mathbf{Q}_2^T \mathbf{Q}_1^T \mathbf{A} \mathbf{Q}_1 \mathbf{Q}_2 \mathbf{Q}_3 && (51)\\
&\vdots && (52)\\
\mathbf{A}_k &= (\mathbf{Q}_k^T \cdots \mathbf{Q}_1^T) \mathbf{A} (\mathbf{Q}_1 \cdots \mathbf{Q}_k) && (53)\\
\mathbf{A}_k &\simeq \mathbf{D}, && (54)
\end{aligned}$$

where recall that $\mathbf{D}$ is a diagonal matrix. We can simplify the notation by defining

$$\mathbf{V} = \prod_{i=1}^{k} \mathbf{Q}_i, \tag{55}$$

which gives

$$\mathbf{D} = \mathbf{A}_k = \mathbf{V}^T \mathbf{A} \mathbf{V}. \tag{56}$$

We can finally multiply (from the left) the above equation by $\mathbf{V}$ to obtain

$$\mathbf{A}\mathbf{V} = \mathbf{V}\mathbf{D}, \tag{57}$$

which is exactly the *definition* of eigenvectors (columns of $\mathbf{V}$) and eigenvalues (diagonal values of $\mathbf{D}$).

In summary, a QR decomposition can be accomplished by iterating through the following steps:

1. Create an $N \times N$ matrix $\mathbf{V}$ to hold the eigenvectors and initially set it equal to $\mathbf{I}$. Choose a target accuracy for the off-diagonal elements.
2. Calculate the QR decomposition $\mathbf{A} = \mathbf{Q}\mathbf{R}$.
3. Update $\mathbf{A}$ to the new value $\mathbf{A} = \mathbf{R}\mathbf{Q}$.
4. Multiple $\mathbf{V}$ on the right by $\mathbf{Q}$.
5. If all of the off-diagonal elements are less than the tolerance, we are finished; otherwise, return to step #2.

The only tricky portion of the method is calculating the QR decomposition, which is described in Exercise 6.8 of the book and can be done in a few lines of Python.

Just like the LU decomposition, this is a very common operation in any computational science, so many software packages have optimized functions to calculate eigenvectors and eigenvalues. The `numpy.linalg` module indeed has these functions. The following sample code calculates both the eigenvectors and eigenvalues,

```
import numpy as np
from numpy import linalg
A = np.array([[1,2], [2,1]], float)
x,V = linalg.eigh(A)
print(x)
print(V)
```

However sometimes we only require the eigenvalues. In `numpy`, there exists a function `eigvalsh` only returning them, as calculating the eigenvectors is relatively time consuming compared to calculating the eigenvectors.

```
import numpy as np
from numpy import linalg
A = np.array([[1,2], [2,1]], float)
x = linalg.eigvalsh(A)
print(x)
```