# Systems of Equations

## 1 Matrices in Python

In Python, we can represent matrices as two dimensional numpy arrays. To create a $m \times n$ matrix, you can do
`a = numpy.zeros([m,n],float)`
to initialize the matrix with all zero elements , or
`a = numpy.empty([m,n],float)`
to create an empty array.

Or you can create an array with given starting values. For example, for a $2 \times 2$ array:
`a = numpy.array([[-4,2],[3,0]],float)`
where `[-4,2]` is the first row and `[3,0]` is the second row.

As we've seen before, you can do calculations with arrays, from simple arithmetic
`a+b`
to other mathematical functions
`numpy.sin(a)`
where `a` and `b` are arrays.

**Section 7, Exercise 1**

Another important thing to keep in mind when dealing with arrays is that
`b = a`
where `a` and `b` are both arrays may not work as you expect. This command just makes `b` another name for `a`. If you change `a`, `b` will be changed also. Using
`b = numpy.copy(a)`
will be make `b` an independent array from `b`, which will not change when `a` does.

**Example: Copying Arrays**

# 2 Simultaneous Linear Equations

## 2.1 Gaussian Elimination

Suppose you want to solve the following system of equations for variables $x$ and $y$:

$$\begin{aligned} 2x + 3y &= 4 \\ 4x - 6y &= 7 \end{aligned}$$

We can multiply the second equation by 1/2, and then subtract the second equation from the first to get equation to solve for $y$:

$$\begin{aligned} (2x + 3y) - (1/2)(4x - 6y) &= 4 - (1/2)7 \\ 3y + 3y &= 8/2 - 7/2 \\ 6y &= 1/2 \\ y &= 1/12 \end{aligned} \tag{1}$$

Plugging this value of $y$ back in to either equation will yield the value of $x$:

$$\begin{aligned} 2x + 3(1/12) &= 4 \\ 2x &= 16/4 - 1/4 \\ 2x &= 15/4 \\ x &= 15/8 \end{aligned} \tag{2}$$

Let's look at the same problem, writing it as a matrix equation:

$$\begin{pmatrix} 2 & 3 \\ 4 & -6 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 4 \\ 7 \end{pmatrix} \tag{3}$$

The method of *Gaussian elimination* is a method to turn this matrix into an upper triangular matrix (where all the elements below the diagonal are zero) and the diagonal elements are all one. For a 2x2 matrix, it should look like this:

$$\begin{pmatrix} 1 & a \\ 0 & 1 \end{pmatrix} \tag{4}$$

The first thing we do is divide the first equation by the upper left element (2), so that the matrix equation becomes

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

This system has the same solution as before; all we've done is multiply both sides of the first equation by a constant:

$$x + (3/2)y = 2 \tag{6}$$

Now, to make the lower left element zero, we first multiply it by 1/4:

$$\begin{pmatrix} 1 & 3/2 \\ 1 & -3/2 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 2 \\ 7/4 \end{pmatrix} \tag{7}$$

Now we can subtract the second row from the first (on both sides of the equation) and replace the second row with the result:

$$\begin{pmatrix} 1 & 3/2 \\ 0 & 3 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 2 \\ 1/4 \end{pmatrix} \tag{8}$$

This is equivalent to what we did above - combining the equations in a particular way to produce an equation that is only dependent on one of the variables. Now we multiply the second row by 1/3 on both sides to make the diagonal element 1:

$$\begin{pmatrix} 1 & 3/2 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 2 \\ 1/12 \end{pmatrix} \tag{9}$$

Now this system can be solved easily by backsubstitution. Multiplying out the second row gives

$$\begin{aligned} (0)x + (1)y &= 1/12 \\ y &= 1/12 = 0.083 \end{aligned} \tag{10}$$

Now $y$ can be plugged into the equation in the first row:

$$\begin{aligned} x + (3/2)y &= 2 \\ x &= 2 - (3/2)y \\ x &= 2 - (3/2)(1/12) \\ x &= 48/24 - 3/24 \\ x &= 45/24 \\ x &= 15/8 = 1.875 \end{aligned} \tag{11}$$

Now let's see how to write this in Python.

**Example: Gaussian Elimination for 2x2 Matrix**

Now let's write a general program to use Gaussian elimination for an $n \times n$ matrix. This is the general procedure:

- For the first row, divide by the diagonal element to make that element 1
  - For all the rows below the first row, divide each row by its first element to make the first column all 1's
  - Replace the second row with the first row minus the second row (first element in the second row will now be zero)

  - Replace the third row with the first row minus the third row (first element in the third row will now be zero)
  - ... and so on until you reach the last row. Now the first column is 1 0 0 ... 0.

- For the second row, divide by the diagonal element to make that element 1
  - For all the rows below the second row, divide each row by its second element to make the second column all 1's
  - Replace the third row with the second row minus the third row (second element in the third row will now be zero)
  - Replace the fourth row with the second row minus the fourth row (second element in the fourth row will now be zero)
  - ... and so on until you reach the last row. Now the second colum is $x$ 1 0 0 ... 0, where $x$ is some number.

- Keep repeating for each row until the matrix has zeroes below the diagonal and ones on the diagonal.

- Then apply backsubstitution, from the bottom up. Let the variables be represented by $\vec{x} = (x_0, x_1, x_2, ...x_{n-1})$.
  - $x_{n-1} = v_{n-1}$
  - $x_{n-2} = v_{n-2} - a_{n-2,n-1}x_{n-1}$
  - ...
  - $x_0 = v_0 - a_{0,1}x_1 + a_{0,2}x_2 + ... + a_{n-2,n-1}x_{n-1}$

We'll use general program to solve the $2 \times 2$ case above as well as this system with three equations with three unknowns:

$$\begin{aligned} -4x + y - z &= 8 \\ 2x - 5y + 2z &= 10 \\ 6x + 2y - 4z &= -3 \end{aligned}$$

**Example: Gaussian Elimination for nxn Matrix**

This method runs into a problem if any of the elements we need to divide by are zero. For example, this matrix equation:

$$\begin{pmatrix} 0 & 2 \\ -3 & 8 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 1 \\ 2 \end{pmatrix} \tag{12}$$

Our first step would normally be to divide the first row by the first diagonal, but that will lead us to divide by zero. This problem can be addressed using *pivoting*, which means to interchange the rows of the matrix to avoid the problem. Interchanging rows is equivalent to switching the order of the equations, which does not affect the result. One scheme to implement this is as follows: For the row $i$, check all lower rows to see which row has its $i$th element farthest from zero (positive or negative), then swap this row with row $i$. Note that there should not be an entire column of zero's; this would mean that the equations do not depend on one of the variables, and you have an ill-posed problem. So there is always a maximum non-zero element.

Furthermore, part of our scheme is to make the elements below the diagonal zero; if one of these elements is already zero, we can move on without doing anything. This also avoids the problem of dividing by zero.

**Example: Gaussian Elimination with Pivoting**

**Section 7, Exercise 2**

## 2.2 LU Decomposition

Suppose we have a matrix:

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

The first step in Gaussian elimination for this matrix is equivalent to multiplying by this matrix:

$$\mathbf{L}_0 = \frac{1}{a_{00}} \begin{pmatrix} 1 & 0 & 0 & 0 \\ -a_{10} & a_{00} & 0 & 0 \\ -a_{20} & 0 & a_{00} & 0 \\ -a_{30} & 0 & 0 & a_{00} \end{pmatrix} \tag{14}$$

Let's show this explicitly:

$$\begin{aligned} \mathbf{L}_0\mathbf{A} &= \frac{1}{a_{00}} \begin{pmatrix} 1 & 0 & 0 & 0 \\ -a_{10} & a_{00} & 0 & 0 \\ -a_{20} & 0 & a_{00} & 0 \\ -a_{30} & 0 & 0 & a_{00} \end{pmatrix} \begin{pmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & a_{12} & a_{13} \\ a_{20} & a_{21} & a_{22} & a_{23} \\ a_{30} & a_{31} & a_{32} & a_{33} \end{pmatrix} \\ &= \frac{1}{a_{00}} \begin{pmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ 0 & -a_{10}a_{01} + a_{00}a_{11} & -a_{10}a_{02} + a_{00}a_{12} & -a_{10}a_{03} + a_{00}a_{13} \\ 0 & -a_{20}a_{01} + a_{00}a_{21} & -a_{20}a_{02} + a_{00}a_{22} & -a_{20}a_{03} + a_{00}a_{23} \\ 0 & -a_{30}a_{01} + a_{00}a_{31} & -a_{30}a_{02} + a_{00}a_{33} & -a_{30}a_{03} + a_{00}a_{33} \end{pmatrix} \\ &\equiv \begin{pmatrix} 1 & b_{01} & b_{02} & b_{03} \\ 0 & b_{11} & b_{12} & b_{13} \\ 0 & b_{21} & b_{22} & b_{23} \\ 0 & b_{31} & b_{32} & b_{33} \end{pmatrix} \end{aligned} \tag{15}$$

So this gives us the same result as the first step of Gaussian elimination: a matrix with the first column with a 1 on the diagonal and zeroes elsewhere.

The next step in Gaussian elimination can be accomplished by multiplying by this matrix:

$$\mathbf{L}_1 = \frac{1}{b_{11}} \begin{pmatrix} b_{11} & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ -b_{21} & b_{11} & 0 & 0 \\ 0 & -b_{31} & 0 & b_{11} \end{pmatrix} \tag{16}$$

which produces a matrix of the form:

$$\mathbf{L}_1\mathbf{L}_0\mathbf{A} = \frac{1}{a_{00}b_{11}} \begin{pmatrix} b_{11} & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ -b_{21} & b_{11} & 0 & 0 \\ 0 & -b_{31} & 0 & b_{11} \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 \\ -a_{10} & a_{00} & 0 & 0 \\ -a_{20} & 0 & a_{00} & 0 \\ -a_{30} & 0 & 0 & a_{00} \end{pmatrix} \begin{pmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & a_{12} & a_{13} \\ a_{20} & a_{21} & a_{22} & a_{23} \\ a_{30} & a_{31} & a_{32} & a_{33} \end{pmatrix}$$
$$\equiv \begin{pmatrix} 1 & c_{01} & c_{02} & c_{03} \\ 0 & 1 & c_{12} & c_{13} \\ 0 & 0 & c_{22} & c_{23} \\ 0 & 0 & c_{32} & c_{33} \end{pmatrix} \tag{17}$$

The next step can be accomplished by multiplying by $\mathbf{L}_2$,

$$\mathbf{L}_2 = \frac{1}{c_{22}} \begin{pmatrix} c_{22} & 0 & 0 & 0 \\ 0 & c_{22} & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & -c_{32} & c_{22} \end{pmatrix} \tag{18}$$

where

$$\mathbf{L}_2\mathbf{L}_1\mathbf{L}_0\mathbf{A} \equiv \begin{pmatrix} 1 & d_{01} & d_{02} & d_{03} \\ 0 & 1 & d_{12} & d_{13} \\ 0 & 0 & 1 & d_{23} \\ 0 & 0 & 0 & d_{33} \end{pmatrix} \tag{19}$$

Finally, the last step is accomplished by multiplying by $\mathbf{L}_3$:

$$\mathbf{L}_3 = \frac{1}{d_{33}} \begin{pmatrix} d_{33} & 0 & 0 & 0 \\ 0 & d_{33} & 0 & 0 \\ 0 & 0 & d_{33} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{20}$$

where

$$\mathbf{L}_3\mathbf{L}_2\mathbf{L}_1\mathbf{L}_0\mathbf{A} \equiv \begin{pmatrix} 1 & u_{01} & u_{02} & u_{03} \\ 0 & 1 & u_{12} & u_{13} \\ 0 & 0 & 1 & u_{23} \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{21}$$

This is the desired result.

Given a set of simultaneous linear equations, we can multiply both sides by the matrix $\mathbf{L}_3\mathbf{L}_2\mathbf{L}_1\mathbf{L}_0$:

$$\begin{aligned}\mathbf{A}\mathbf{x} &= \mathbf{v}\\ \mathbf{L}_3\mathbf{L}_2\mathbf{L}_1\mathbf{L}_0\mathbf{A}\mathbf{x} &= \mathbf{L}_3\mathbf{L}_2\mathbf{L}_1\mathbf{L}_0\mathbf{v}\end{aligned} \tag{22}$$

Everything on the right-hand side is known. On the left, we have an upper-triangular matrix times $\mathbf{x}$, which can easily be solved by backsubstitution. This is mathematically equivalent to the Gaussian elimination method. The advantage here is in the case where we need to solve many sets of equations with the same $\mathbf{A}$, but different $\mathbf{v}$. The left hand side in that case can be calculated once and need not be repeated. That tends to be the most time-consuming part of the Gaussian elimination process, and therefore using this method can speed up the program.

The typical way to write out this method is a little different from what's presented above. Let

$$\mathbf{L} \equiv \mathbf{L}_0^{-1}\mathbf{L}_1^{-1}\mathbf{L}_2^{-1}\mathbf{L}_3^{-1} \tag{23}$$

$$\mathbf{U} \equiv \mathbf{L}_3\mathbf{L}_2\mathbf{L}_1\mathbf{L}_0\mathbf{A} \tag{24}$$

then

$$\begin{aligned}\mathbf{L}\mathbf{U} &= \mathbf{L}_0^{-1}\mathbf{L}_1^{-1}\mathbf{L}_2^{-1}\mathbf{L}_3^{-1}\mathbf{L}_3\mathbf{L}_2\mathbf{L}_1\mathbf{L}_0\mathbf{A}\\ &= \mathbf{A}\end{aligned} \tag{25}$$

and

$$\begin{aligned}\mathbf{A}\mathbf{x} &= \mathbf{v}\\ \mathbf{L}\mathbf{U}\mathbf{x} &= \mathbf{v}\end{aligned} \tag{26}$$

Now $\mathbf{U}$ is an upper triangular matrix as we've already shown (Equation 21). $\mathbf{L}$ is lower triangular, given by

$$\mathbf{L} = \begin{pmatrix} a_{00} & 0 & 0 & 0\\ a_{10} & b_{11} & 0 & 0\\ a_{20} & b_{21} & c_{22} & 0\\ a_{30} & b_{31} & c_{32} & d_{33}\end{pmatrix} \equiv \begin{pmatrix} l_{00} & 0 & 0 & 0\\ l_{10} & l_{11} & 0 & 0\\ l_{20} & l_{21} & l_{22} & 0\\ l_{30} & l_{31} & l_{32} & l_{33}\end{pmatrix} \tag{27}$$

So we have

$$\mathbf{L}\mathbf{U}\mathbf{x} = \mathbf{v}$$

$$\begin{pmatrix} l_{00} & 0 & 0 & 0\\ l_{10} & l_{11} & 0 & 0\\ l_{20} & l_{21} & l_{22} & 0\\ l_{30} & l_{31} & l_{32} & l_{33}\end{pmatrix}\begin{pmatrix} 1 & u_{01} & u_{02} & u_{03}\\ 0 & 1 & u_{12} & u_{13}\\ 0 & 0 & 1 & u_{23}\\ 0 & 0 & 0 & 1\end{pmatrix}\begin{pmatrix} x_0\\ x_1\\ x_2\\ x_3\end{pmatrix} = \begin{pmatrix} v_0\\ v_1\\ v_2\\ v_3\end{pmatrix} \tag{28}$$

Now we define a new vector $\mathbf{y}$:

$$\begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \end{pmatrix} \equiv \begin{pmatrix} 1 & u_{01} & u_{02} & u_{03} \\ 0 & 1 & u_{12} & u_{13} \\ 0 & 0 & 1 & u_{23} \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{pmatrix} \tag{29}$$

Now we can split our backsubstitution into two parts. First we use this to solve for $\mathbf{y}$, using backsubstitution from top to bottom:

$$\begin{pmatrix} l_{00} & 0 & 0 & 0 \\ l_{10} & l_{11} & 0 & 0 \\ l_{20} & l_{21} & l_{22} & 0 \\ l_{30} & l_{31} & l_{32} & l_{33} \end{pmatrix} \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \end{pmatrix} = \begin{pmatrix} v_0 \\ v_1 \\ v_2 \\ v_3 \end{pmatrix} \tag{30}$$

which gives

$$\begin{aligned} l_{00}y_0 &= v_0 \\ y_0 &= \frac{v_0}{l_{00}} \end{aligned} \tag{31}$$

and then

$$\begin{aligned} l_{10}y_0 + l_{11}y_1 &= v_1 \\ y_1 &= \frac{v_1 - l_{10}y_0}{l_{11}} \end{aligned} \tag{32}$$

and so on.

Then we can use this to solve for $\mathbf{x}$, using backsubstitution from bottom to top as before:

$$\begin{pmatrix} 1 & u_{01} & u_{02} & u_{03} \\ 0 & 1 & u_{12} & u_{13} \\ 0 & 0 & 1 & u_{23} \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \end{pmatrix} \tag{33}$$

so we have

$$x_3 = y_3 \tag{34}$$

$$x_2 = y_2 - u_{23}x_3 \tag{35}$$

and so on.

Note that pivoting must be used with this method as well, to avoid dividing by zeroes.

This method, called *LU decomposition*, is the most commonly used method for solving simultaneous linear equations. In Python, the `linalg` module from numpy provides functions for soling simultaneous equations. In particular, `linalg.solve` uses LU decomposition and backsubstitution as we've discussed here.

**Example: LU Decomposition**

## 2.3 Tridiagonal Matrices

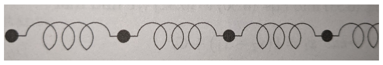
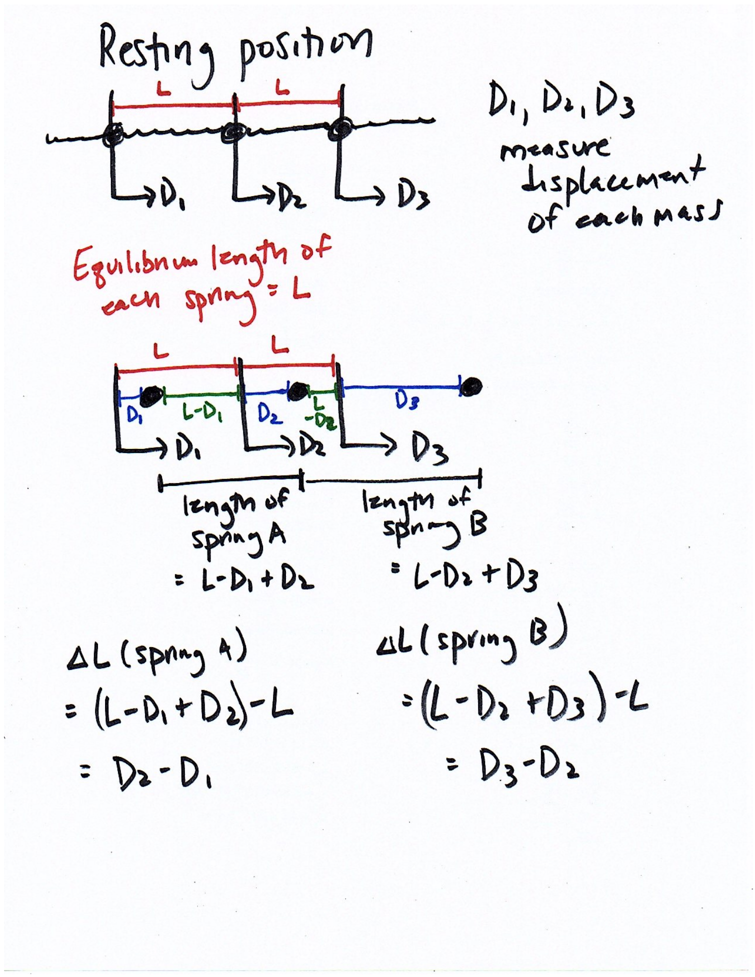


Figure 1:

Suppose we have a set of $N$ identical masses in a row, joined by identical linear springs, as shown in Figure 1. (This system can be used as a model of the vibration of atoms in a solid.) The displacement of the $i$th mass relative to its rest position is given by $D_i$. The spring constant for each spring is $k$, and the mass of each spring is $m$. To find the equation of motion for each mass, consider Figure 2. In the figure $D_1, D_2, D_3$ are the displacements of each mass from equilibrium, and $L$ is the length of each spring at equilibrium.



Resting position
L
L
$D_1$
$D_2$
$D_3$
$D_1, D_2, D_3$ measure displacement of each mass
Equilibrium length of each spring = L
L
L
$D_1$
$L-D_1$
$D_2$
$L-D_2$
$D_3$
$D_1$
$D_2$
$D_3$
length of spring A = $L-D_1+D_2$
length of spring B = $L-D_2+D_3$
$\Delta L$ (spring A) = $(L-D_1+D_2)-L$ = $D_2-D_1$
$\Delta L$ (spring B) = $(L-D_2+D_3)-L$ = $D_3-D_2$

Figure 2:

The two springs on either side of the middle mass exert forces on the middle mass. In the figure, the spring on the left (spring A) is stretched by a distance $D_2 - D_1$, and the spring on the right (spring B) is stretched by a distance $D_3 - D_2$. If spring A is stretched, the restoring force will compress the spring, which would pull the middle mass to the *left*. Therefore, the force on the middle mass from spring A is $F_A = -k(D_2 - D_1)$. If spring B is stretched the restoring force will compress that spring, which would pull the middle mass to the *right*. Therefore, the force on the middle mass from spring B is $F_B = k(D_3 - D_2)$. The total restoring force on the middle mass is therefore $F = -k(D_2 - D_1) + k(D_3 - D_2)$.

**Section 7, Exercise 3**

If we call the middle mass the $i$ mass, then the equation of motion of the $i$th mass is

$$\begin{aligned} m\frac{d^2 D_i}{dt^2} &= -k(D_i - D_{i-1}) + k(D_{i+1} - D_i) \\ m\frac{d^2 D_i}{dt^2} &= k(D_{i-1} - D_i) + k(D_{i+1} - D_i) \end{aligned} \tag{36}$$

(Plugging in $i = 2$ gives the result we found in the figure.)

If we include an external force imposed on each mass $i$, the equation of motion becomes:

$$m\frac{d^2 D_i}{dt^2} = k(D_{i-1} - D_i) + k(D_{i+1} - D_i) + F_i \tag{37}$$

The above equation of motion assumes that there are two springs for each mass, but that won't be true for the first and last mass in the line. For those masses, the equations of motion are:

$$m\frac{d^2 D_0}{dt^2} = k(D_1 - D_0) + F_0 \tag{38}$$

$$m\frac{d^2 D_{N-1}}{dt^2} = k(D_{N-2} - D_{N-1}) + F_{N-1} \tag{39}$$

and Equation 36 only applies for $i = 1$ to $i = N - 2$ (where we're using Python numbering from 0 to $N - 1$ for $N$ masses).

Let's assume the system is driven by applying a sinusoidal driving force to the first mass (and only to the first mass, $F_i = 0$ for $i \neq 0$):

$$F_0 = Ce^{i\omega t} \tag{40}$$

where $C$ is a constant and

$$e^{i\omega t} = \cos(\omega t) + i\sin(\omega t) \tag{41}$$

(Sinusoidal functions are often represented in this complex form, with the assumption that we will take the real part of the number at the end of the calculation to represent the physical quantity.)

This applied force will make the atoms oscillate with angular frequency $\omega$, so that the solution for the position of each atom wil take the form

$$D_i(t) = x_i e^{i\omega t} \tag{42}$$

where $x_i$ is the amplitude of vibration of mass $i$. The second derivative of this function is

$$\begin{aligned}\frac{dD_i}{dt} &= (i\omega)x_i e^{i\omega t}\\ \frac{d^2D_i}{dt^2} &= (i\omega)^2 x_i e^{i\omega t}\\ \frac{d^2D_i}{dt^2} &= -\omega^2 x_i e^{i\omega t}\end{aligned} \tag{43}$$

If we substitute this solution into our three equations of motion, we have for mass 0:

$$\begin{aligned}m\frac{d^2D_0}{dt^2} &= k(D_1 - D_0) + F_0\\ -m\omega^2 x_0 e^{i\omega t} &= k(x_1 e^{i\omega t} - x_0 e^{i\omega t}) + Ce^{i\omega t}\\ -m\omega^2 x_0 &= k(x_1 - x_0) + C\\ -m\omega^2 x_0 &= kx_1 - kx_0 + C\\ (k - m\omega^2)x_0 - kx_1 &= C\end{aligned} \tag{44}$$

For mass $i$:

$$\begin{aligned}m\frac{d^2D_i}{dt^2} &= k(D_{i-1} - D_i) + k(D_{i+1} - D_i) + F_i\\ -m\omega^2 x_i e^{i\omega t} &= k(x_{i-1}e^{i\omega t} - x_i e^{i\omega t}) + k(x_{i+1}e^{i\omega t} - x_i e^{i\omega t}) + 0\\ -m\omega^2 x_i &= k(x_{i-1} - x_i) + k(x_{i+1} - x_i)\\ -m\omega^2 x_i &= kx_{i-1} - kx_i + kx_{i+1} - kx_i\\ -kx_{i-1} + (2k - m\omega^2)x_i - kx_{i+1} &= 0\end{aligned} \tag{45}$$

And for mass $N-1$:

$$\begin{aligned}m\frac{d^2D_{N-1}}{dt^2} &= k(D_{N-2} - D_{N-1}) + F_{N-1}\\ -m\omega^2 x_{N-1} e^{i\omega t} &= k(x_{N-2}e^{i\omega t} - x_{N-1}e^{i\omega t}) + 0\\ -m\omega^2 x_{N-1} &= k(x_{N-2} - x_{N-1})\\ -m\omega^2 x_{N-1} &= kx_{N-2} - kx_{N-1}\\ -kx_{N-2} + (k - m\omega^2)x_{N-1} &= 0\end{aligned} \tag{46}$$

If we define

$$\alpha \equiv 2k - m\omega^2 \tag{47}$$

then we can write these three equations as

$$(\alpha - k)x_0 - kx_1 = C \tag{48}$$
$$-kx_{i-1} + \alpha x_i - kx_{i+1} = 0 \tag{49}$$
$$-kx_{N-2} + (\alpha - k)x_{N-1} = 0 \tag{50}$$

In matrix form, this looks like

$$\begin{pmatrix} (\alpha-k) & -k & 0 & \dots & & & & \\ -k & \alpha & -k & 0 & \dots & & & \\ 0 & -k & \alpha & -k & 0 & \dots & & \\ \dots & 0 & -k & \alpha & -k & 0 & & \dots \\ & & & \ddots & & & & \\ & & \dots & 0 & -k & \alpha & & -k \\ & & & \dots & 0 & -k & & (\alpha-k) \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_{N-1} \\ x_{N-1} \end{pmatrix} = \begin{pmatrix} C \\ 0 \\ 0 \\ 0 \\ \vdots \\ 0 \\ 0 \end{pmatrix} \tag{51}$$

This is called a tridiagonal matrix, in which the matrix has nonzero elements only along the diagonal and immediately above and below the diagonal. We can solve this system using Gaussian elimination or LU decomposition, but a specialized algorithm is better, particularly for large matrices. This is because Gaussian elimination or LU decomposition is going to do a lot of calculations on zeros in that matrix, which are unnecessary; an algorithm specially designed for tridiagonal matrices will skip all operations on matrix elements that we know ahead of time are zeros. With that in mind, let's look at a program written specifically for this problem. We'll use Gaussian elimination, but only do operations on the tridiagonal elements. Let's assume $C = 1$, $m = 1$, $k = 6$, and $\omega = 2$, so $\alpha = 2(6) - (1)(2)^2 = 12 - 4 = 8$ and $\alpha - k = 2$. For the first case, we'll assume $N = 5$. The matrix looks like this:

$$\begin{pmatrix} 2 & -6 & 0 & 0 & 0 \\ -6 & 8 & -6 & 0 & 0 \\ 0 & -6 & 8 & -6 & 0 \\ 0 & 0 & -6 & 8 & -6 \\ 0 & 0 & 0 & -6 & 2 \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} \tag{52}$$

Then we'll assume $N = 26$ and plot the resulting values of $x_i$.

**Example: Tridiagonal Matrix**

## 3 Eigenvalues and Eigenvectors

Calculation of eigenvalues and eigenvectors of a matrix is a common matrix problem in physics. Most of these problems concern real symmetric matrices ($a_{ji} = a_{ij}$ or $\mathbf{A} = \mathbf{A}^T$) or Hermitian matrices if complex numbers are involved ($a_{ji} = a_{ij}*$).

For a symmetric matrix $\mathbf{A}$, and eigenvector $\mathbf{v}$ is a vector such that

$$\mathbf{A}\mathbf{v} = \lambda\mathbf{v} \tag{53}$$

where $\mathbf{v}$ is the eigenvector and $\lambda$ is the corresponding eigenvalue. For an $n \times n$ matrix, there are $n$ eigenvectors $\mathbf{v}_1$, $\mathbf{v}_2$, ..., $\mathbf{v}_n$ with eigenvalues $\lambda_1$, $\lambda_2$, ..., $\lambda_n$. The eigenvectors are all orthogonal to each other, meaning that $\mathbf{v}_i \cdot \mathbf{v}_j =$ if $i \neq j$. We will assume that they all have unit length, $|\mathbf{v}_i| = 1$.

We can consider each eigenvector to be the columns of a $n \times n$ matrix $\mathbf{V}$ and combine all the equations given by

$$\mathbf{A}\mathbf{v}_i = \lambda_i\mathbf{v}_i; \quad i = 0, ..., n-1 \tag{54}$$

into one equation

$$\mathbf{A}\mathbf{V} = \mathbf{V}\mathbf{D} \tag{55}$$

where $\mathbf{D}$ is a diagonal matrix with the eigenvalues $\lambda_i$ as its diagonal entries. For example, for a $3 \times 3$ matrix $\mathbf{A}$:

$$\begin{gathered}
\mathbf{A}\mathbf{v}_i = \lambda_i\mathbf{v}_i \\
\begin{pmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \end{pmatrix} \begin{pmatrix} (v_i)_0 \\ (v_i)_1 \\ (v_i)_2 \end{pmatrix} = \lambda_i \begin{pmatrix} (v_i)_0 \\ (v_i)_1 \\ (v_i)_2 \end{pmatrix} \\
\begin{pmatrix} a_{00}(v_i)_0 + a_{01}(v_i)_1 + a_{02}(v_i)_2 \\ a_{10}(v_i)_0 + a_{11}(v_i)_1 + a_{12}(v_i)_2 \\ a_{20}(v_i)_0 + a_{21}(v_i)_1 + a_{22}(v_i)_2 \end{pmatrix} = \begin{pmatrix} \lambda_i(v_i)_0 \\ \lambda_i(v_i)_1 \\ \lambda_i(v_i)_2 \end{pmatrix}
\end{gathered} \tag{56}$$

$$\begin{aligned}
\mathbf{A}\mathbf{V} &= \begin{pmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \end{pmatrix} \begin{pmatrix} (v_0)_0 & (v_1)_0 & (v_2)_0 \\ (v_0)_1 & (v_1)_1 & (v_2)_1 \\ (v_0)_2 & (v_1)_2 & (v_2)_2 \end{pmatrix} \\
&= \begin{pmatrix} a_{00}(v_0)_0 + a_{01}(v_0)_1 + a_{02}(v_0)_2 & a_{00}(v_1)_0 + a_{01}(v_1)_1 + a_{02}(v_1)_2 & a_{00}(v_2)_0 + a_{01}(v_2)_1 + a_{02}(v_2)_2 \\ a_{10}(v_0)_0 + a_{11}(v_0)_1 + a_{12}(v_0)_2 & a_{10}(v_1)_0 + a_{11}(v_1)_1 + a_{12}(v_1)_2 & a_{10}(v_2)_0 + a_{11}(v_2)_1 + a_{12}(v_2)_2 \\ a_{20}(v_0)_0 + a_{21}(v_0)_1 + a_{22}(v_0)_2 & a_{20}(v_1)_0 + a_{21}(v_1)_1 + a_{22}(v_1)_2 & a_{20}(v_2)_0 + a_{21}(v_2)_1 + a_{22}(v_2)_2 \end{pmatrix} \\
&= \begin{pmatrix} \lambda_0(v_0)_0 & \lambda_1(v_1)_0 & \lambda_2(v_2)_0 \\ \lambda_0(v_0)_1 & \lambda_1(v_1)_1 & \lambda_2(v_2)_1 \\ \lambda_0(v_0)_2 & \lambda_1(v_1)_2 & \lambda_2(v_2)_2 \end{pmatrix} \\
&= \begin{pmatrix} (v_0)_0 & (v_1)_0 & (v_2)_0 \\ (v_0)_1 & (v_1)_1 & (v_2)_1 \\ (v_0)_2 & (v_1)_2 & (v_2)_2 \end{pmatrix} \begin{pmatrix} \lambda_0 & 0 & 0 \\ 0 & \lambda_1 & 0 \\ 0 & 0 & \lambda_2 \end{pmatrix} \\
&= \mathbf{V}\mathbf{D}
\end{aligned} \tag{57}$$

The most commonly used algorithm for finding eigenvalues and eigenvectors of real symmetric or Hermitian matrices is the *QR algorithm*. We're only going to consider the case of real and symmetric matrices, but applying the algorithm to Hermitian matrices is a straightforward extension of what we do here.

In the QR algorithm, the first step is to find the so-called *QR decomposition* of the matrix $\mathbf{A}$, which breaks $\mathbf{A}$ into an orthogonal matrix $\mathbf{Q}$ and an upper-triangular matrix $\mathbf{R}$, $\mathbf{A} = \mathbf{QR}$. Any square matrix can be written in this form.

An orthogonal matrix is a real square matrix whose coumns and rows are orthogonal unit vectors, which means that $\mathbf{Q}^T\mathbf{Q} = \mathbf{Q}\mathbf{Q}^T = \mathbf{I}$, where $\mathbf{I}$ is the identity matrix.

Let's write the matrix $\mathbf{A}$ as a set of $n$ column vectors $\mathbf{a}_0$, $\mathbf{a}_1$, ..., $\mathbf{a}_{n-1}$:

$$\mathbf{A} = \begin{pmatrix} \mathbf{a}_0 & \mathbf{a}_1 & \dots & \mathbf{a}_{n-1} \end{pmatrix} \tag{58}$$

We define two new sets of vectors, $\mathbf{u}_i$ and $\mathbf{q}_i$:

$$\mathbf{u}_i = \mathbf{a}_i - \sum_{j=0}^{i-1} (\mathbf{q}_j \cdot \mathbf{a}_i)\mathbf{q}_j \tag{59}$$

$$\mathbf{q}_i = \frac{\mathbf{u}_i}{|\mathbf{u}_i|} \tag{60}$$

With these vectors, the matrices $\mathbf{Q}$ and $\mathbf{R}$ are given by

$$\mathbf{Q} = \begin{pmatrix} \mathbf{q}_0 & \mathbf{q}_1 & \dots & \mathbf{q}_{n-1} \end{pmatrix} \tag{61}$$

$$\mathbf{R} = \begin{pmatrix} |\mathbf{u}_0| & \mathbf{q}_0 \cdot \mathbf{a}_1 & \mathbf{q}_0 \cdot \mathbf{a}_2 & \dots & \mathbf{q}_0 \cdot \mathbf{a}_{n-1} \\ 0 & |\mathbf{u}_1| & \mathbf{q}_1 \cdot \mathbf{a}_2 & \dots & \mathbf{q}_1 \cdot \mathbf{a}_{n-1} \\ 0 & 0 & |\mathbf{u}_2| & \dots & \mathbf{q}_2 \cdot \mathbf{a}_{n-1} \\ & & \ddots & & \end{pmatrix} \tag{62}$$

**Section 7, Exercise 4**

Let's show that this decomposition works out for a $3 \times 3$ matrix:

$$\mathbf{u}_0 = \mathbf{a}_0 \tag{63}$$

$$\mathbf{q}_0 = \frac{\mathbf{u}_0}{|\mathbf{u}_0|} \tag{64}$$

$$\mathbf{u}_1 = \mathbf{a}_1 - (\mathbf{q}_0 \cdot \mathbf{a}_1)\mathbf{q}_0 \tag{65}$$

$$\mathbf{q}_1 = \frac{\mathbf{u}_1}{|\mathbf{u}_1|} \tag{66}$$

$$\mathbf{u}_2 = \mathbf{a}_2 - (\mathbf{q}_0 \cdot \mathbf{a}_2)\mathbf{q}_0 - (\mathbf{q}_1 \cdot \mathbf{a}_2)\mathbf{q}_1 \tag{67}$$

$$\mathbf{q}_2 = \frac{\mathbf{u}_2}{|\mathbf{u}_2|} \tag{68}$$

$$
\begin{aligned}
\mathbf{QR} &= \begin{pmatrix} (q_0)_0 & (q_1)_0 & (q_2)_0 \\ (q_0)_1 & (q_1)_1 & (q_2)_1 \\ (q_0)_2 & (q_1)_2 & (q_2)_2 \end{pmatrix} \begin{pmatrix} |\mathbf{u}_0| & \mathbf{q}_0\cdot\mathbf{a}_1 & \mathbf{q}_0\cdot\mathbf{a}_2 \\ 0 & |\mathbf{u}_1| & \mathbf{q}_1\cdot\mathbf{a}_2 \\ 0 & 0 & |\mathbf{u}_2| \end{pmatrix} \\
&= \begin{pmatrix} (q_0)_0|\mathbf{u}_0| & (q_0)_0\mathbf{q}_0\cdot\mathbf{a}_1 + (q_1)_0|\mathbf{u}_1| & (q_0)_0\mathbf{q}_0\cdot\mathbf{a}_2 + (q_1)_0\mathbf{q}_1\cdot\mathbf{a}_2 + (q_2)_0|\mathbf{u}_2| \\ (q_0)_1|\mathbf{u}_0| & (q_0)_1\mathbf{q}_0\cdot\mathbf{a}_1 + (q_1)_1|\mathbf{u}_1| & (q_0)_1\mathbf{q}_0\cdot\mathbf{a}_2 + (q_1)_1\mathbf{q}_1\cdot\mathbf{a}_2 + (q_2)_1|\mathbf{u}_2| \\ (q_0)_2|\mathbf{u}_0| & (q_0)_2\mathbf{q}_0\cdot\mathbf{a}_1 + (q_1)_2|\mathbf{u}_1| & (q_0)_2\mathbf{q}_0\cdot\mathbf{a}_2 + (q_1)_2\mathbf{q}_1\cdot\mathbf{a}_2 + (q_2)_2|\mathbf{u}_2| \end{pmatrix} \\
&= \begin{pmatrix} \mathbf{q}_0|\mathbf{u}_0| & \mathbf{q}_0(\mathbf{q}_0\cdot\mathbf{a}_1) + \mathbf{q}_1|\mathbf{u}_1| & \mathbf{q}_0(\mathbf{q}_0\cdot\mathbf{a}_2) + \mathbf{q}_1(\mathbf{q}_1\cdot\mathbf{a}_2) + \mathbf{q}_2|\mathbf{u}_2| \end{pmatrix} \\
&= \begin{pmatrix} \mathbf{u}_0 & \mathbf{q}_0(\mathbf{q}_0\cdot\mathbf{a}_1) + \mathbf{u}_1 & \mathbf{q}_0(\mathbf{q}_0\cdot\mathbf{a}_2) + \mathbf{q}_1(\mathbf{q}_1\cdot\mathbf{a}_2) + \mathbf{u}_2 \end{pmatrix} \\
&= \begin{pmatrix} \mathbf{a}_0 & \mathbf{a}_1 & \mathbf{a}_2 \end{pmatrix} \\
&= \mathbf{A} \qquad (69)
\end{aligned}
$$

Now let's look at program that performs the QR decomposition.

**Example: QR Decomposition**

Now let's see how this decomposition is used to find eigenvalues and eigenvectors. We have a real, square, symmetric matrix $\mathbf{A}$, which is broken down into its QR decomposition. Let's multiply $\mathbf{A}$ by $\mathbf{Q}_1^T$ on the left and use the fact that $\mathbf{Q}_1$ is orthogonal:

$$
\begin{aligned}
\mathbf{A} &= \mathbf{Q}_1\mathbf{R}_1 \\
\mathbf{Q}_1^T\mathbf{A} &= \mathbf{Q}_1^T\mathbf{Q}_1\mathbf{R}_1 \\
\mathbf{Q}_1^T\mathbf{A} &= \mathbf{R}_1 \qquad (70)
\end{aligned}
$$

Now we define a new matrix $\mathbf{A}_1$ and combine with the equation above:

$$
\begin{aligned}
\mathbf{A}_1 &= \mathbf{R}_1\mathbf{Q}_1 \\
\mathbf{A}_1 &= \mathbf{Q}_1^T\mathbf{A}\mathbf{Q}_1 \qquad (71)
\end{aligned}
$$

Now we repeat the process for the matrix $\mathbf{A}_1$, getting the QR decomposition

$$\mathbf{A}_1 = \mathbf{Q}_2\mathbf{R}_2 \qquad (72)$$

then definining a new matrix

$$
\begin{aligned}
\mathbf{A}_2 &= \mathbf{R}_2\mathbf{Q}_2 \\
&= \mathbf{Q}_2^T\mathbf{A}_1\mathbf{Q}_2 \\
&= \mathbf{Q}_2^T\mathbf{Q}_1^T\mathbf{A}\mathbf{Q}_1\mathbf{Q}_2 \qquad (73)
\end{aligned}
$$

We keep repeating this process:

$$
\begin{aligned}
\mathbf{A}_1 &= \mathbf{Q}_1^T\mathbf{A}\mathbf{Q}_1 && (74) \\
\mathbf{A}_2 &= \mathbf{Q}_2^T\mathbf{Q}_1^T\mathbf{A}\mathbf{Q}_1\mathbf{Q}_2 && (75) \\
\mathbf{A}_3 &= \mathbf{Q}_3^T\mathbf{Q}_2^T\mathbf{Q}_1^T\mathbf{A}\mathbf{Q}_1\mathbf{Q}_2\mathbf{Q}_3 && (76) \\
&\dots \\
\mathbf{A}_k &= (\mathbf{Q}_k^T\dots\mathbf{Q}_1^T)\mathbf{A}(\mathbf{Q}_1\dots\mathbf{Q}_k) && (77)
\end{aligned}
$$

If this process is repeated enough, the off-diagonal elements get smaller and smaller until the matrix $\mathbf{A}_k$ eventually becomes diagonal. Since we can't repeat this process infinitely, we will define some threshold $\epsilon$ such that if the off-diagonal elements are $< \epsilon$, then we will say that $\mathbf{A}_k \approx \mathbf{D}$, where $\mathbf{D}$ is a diagonal matrix.

Let

$$\mathbf{V} = \mathbf{Q}_1\mathbf{Q}_2 \ldots \mathbf{Q}_k = \prod_{i=1}^{k} \mathbf{Q}_i \tag{78}$$

Since $\mathbf{V}$ is a product of orthogonal matrices, it is also an orthogonal matrix so that $\mathbf{V}\mathbf{V}^T = \mathbf{V}^T\mathbf{V} = \mathbf{I}$. With this definition we have

$$\begin{aligned} \mathbf{A}_k &\approx \mathbf{D} \\ \mathbf{V}^T\mathbf{A}\mathbf{V} &\approx \mathbf{D} \\ \mathbf{A}\mathbf{V} &\approx \mathbf{V}\mathbf{D} \end{aligned} \tag{79}$$

This means that for the matrix $\mathbf{A}$, the columns for matrix $\mathbf{V}$ are the eigenvectors of $\mathbf{A}$ and the diagonal elements of the diagonal matrix $\mathbf{D} \approx \mathbf{A}_k$ are the corresponding eigenvalues. This is the *QR algorithm* for calculating eigenvalues and eigenvectors. Let's write a program to do this.

Here's the procedure:

- Initialize the program: define matrix $\mathbf{A}$, define threshold $\epsilon$, create matrix $\mathbf{V}$ and initialize it to the identity matrix
- Compute QR decomposition (matrices $\mathbf{Q}$ and $\mathbf{R}$) for $\mathbf{A}$
- Update the value of $\mathbf{A}$ to be $\mathbf{A} = \mathbf{R}\mathbf{Q}$
- Update the value of $\mathbf{V}$ by multiplying the previous value of $\mathbf{V}$ by the new $\mathbf{Q}$.
- Check the off-diagonal elements of $\mathbf{A}$; if they are all less than $\epsilon$, then the process is done. If not, repeat the steps, starting the QR decomposition.

**Example: QR algorithm**

Of course there are built-in Python functions that do this same calculation; let's see how to use those. The functions for computing eigenvalues and eigenvectors for symmetric matrices are `numpy.linalg.eigh` (eigenvalues and eigenvectors) and `numpy.linalg.eigvalsh` (eigenvalues only), where the 'h' is for Hermitian. For a non-symmetric matrix, the functions are `numpy.linalg.eig` and `numpy.linalg.eigvals`.

**Example: QR Using Python Functions**

# 4 Nonlinear Equations

So far we have only considered systems of linear equations, simple linear combinations of the unknown variables. Now we will consider nonlinear equations, which are in general much harder to solve. We'll start out with just a single nonlinear equation, instead of a system of equations.

## 4.1 The Relaxation Method

Consider this equation:

$$x = 2 - e^{-x} \tag{80}$$

There's no analytical method for solving this equation. We'll use what's called the *relaxation method*. We'll start with a guess at the solution for $x$, plug this guess in on the right-hand side, and calculate a new value of $x$. Then we'll use this new value on the right-hand side, producing a new value of $x$, and continue this procedure until the solution converges. Our initial guess will $x = 1$.

**Example: Relaxation Method**

To use the relaxation method, your equation must be in the form of $x = f(x)$, where $f(x)$ is a known function. If the equation is not given in this format, you may have to rearrange to get it in that format. In addition, an equation could have more than one solution. In that case, the relaxation method will converge to only one of the solutions, dependent upon what starting value you give it.

There are some solutions to some equations that simply cannot be found using this method. Consider the following equation:

$$\log x + x^2 - 1 = 0 \tag{81}$$

The first thing to do is to rearrange to get this in the required format:

$$\begin{aligned} \log x + x^2 - 1 &= 0 \\ \log x &= 1 - x^2 \\ x &= e^{1-x^2} \end{aligned} \tag{82}$$

Clearly $x = 1$ is a solution. Let's try to get that using the relaxation method, with a starting value of $x = 0.5$.

**Example: Non-convergence**

When this happens, we can try again to rearrange the equation:

$$\begin{aligned} \log x + x^2 - 1 &= 0 \\ x^2 &= 1 - \log x \\ x &= \sqrt{1 - \log x} \end{aligned} \tag{83}$$

and try again with a starting value of $x = 0.5$.

**Example: Convergence**

(We do have to be a little careful with the starting value, since some values of $x$ will make $\log x > 1$, resulting in a negative number inside the square root.)

What determines whether or not the relaxation method will converge? Assume we have an equation of the form $x = f(x)$ that has a solution at $x = x^*$, meaning that $x^* = f(x^*)$. Let's make a Taylor expansion of $f(x)$ around $x^*$, and consider the value $x'$ which results when you plug some arbitrary value of $x$ into $f(x)$:

$$\begin{aligned}
x' &= f(x) \\
x' &= f(x^*) + (x - x^*)f'(x^*) + ... \\
x' &= x^* + (x - x^*)f'(x^*) + ... \\
x' - x^* &\approx (x - x^*)f'(x^*) \\
\frac{x' - x^*}{x - x^*} &\approx f'(x^*)
\end{aligned} \tag{84}$$

where we neglected higher order terms.

So $x$ is our initial guess, and $x'$ is our next guess. If $x'$ is closer to the solution $x^*$ than $x$ was, then the absolute value of the ratio on the left hand side above should be less than one. That requires that the absolute value of the first derivative of $f$ evaluated at $x^*$ is less than 1: $|f'(x^*)| < 1$. This is the condition for convergence.

Considering our example above, where $x^* = 1$:

$$\begin{aligned}
x &= e^{1-x^2} \\
f(x) &\equiv e^{1-x^2} \\
f'(x) &= -2xe^{1-x^2} \\
f'(1) &= -2(1)e^{1-1} = -2 \\
|f'(1)| &= 2 > 1
\end{aligned} \tag{85}$$

This is why the answer didn't converge. After we rearranged we had

$$\begin{aligned}
x &= \sqrt{1 - \log x} \\
f(x) &\equiv \sqrt{1 - \log x} \\
f'(x) &= \frac{1}{2}(1 - \log x)^{-1/2}\left(-\frac{1}{x}\right) \\
f'(1) &= \frac{1}{2}(1 - \log 1)^{-1/2}\left(-\frac{1}{1}\right) = -\frac{1}{2} \\
|f'(1)| &= \frac{1}{2} < 1
\end{aligned} \tag{86}$$

which is why the answer converged in this case.

Note that if $x = f(x)$, then $x = f^{-1}(x)$, where $f^{-1}$ denotes the functional inverse defined by $f^{-1}(f(x)) = x$:

$$\begin{aligned} x &= f(x) \\ f^{-1}(x) &= f^{-1}(f(x)) \\ f^{-1}(x) &= x \end{aligned} \tag{87}$$

It can be shown that if the solution to $x = f(x)$ doesn't converge with the relaxation method, then the solution to $x = f^{-1}(x)$ will converge. This is because the first derivative of $f$ is the reciprocal of the first derivative of its inverse; therefore if the first derivative of $f$ is greater than one (meaning no convergence), the first derivative of the inverse will be less than one (allowing convergence). In the example above, $\sqrt{1 - \log x}$ is the functional inverse of $e^{1-x^2}$:

$$\sqrt{1 - \log(e^{1-x^2})} = \sqrt{1 - (1 - x^2)} = \sqrt{x^2} = x \tag{88}$$

$$e^{1-(\sqrt{1-\log x})^2} = e^{1-(1-\log x)} = e^{\log x} = x \tag{89}$$

If the relaxation method fails for a particular equation, it will work if you invert the equation. However, not all functions can be inverted, so the relaxation method, while widely applicable, does not work for all equations.

**Section 7, Exercise 5**

Let's consider how to calculate the error on the solution in the relaxation method. Let $\epsilon$ be the error on our current estimate of the solution, so that

$$\epsilon = x^* - x \tag{90}$$

Let $\epsilon'$ be the error on the next estimate

$$\epsilon' = x^* - x' \tag{91}$$

where $x' = f(x)$ and $x^*$ is the true solution.

Using Equation 84, we have

$$\begin{aligned} \frac{x' - x^*}{x - x^*} &\approx f'(x^*) \\ \frac{\epsilon'}{\epsilon} &\approx f'(x^*) \end{aligned} \tag{92}$$

Then

$$x^* = x + \epsilon = x + \frac{\epsilon'}{f'(x^*)} \tag{93}$$

and

$$x^* = x' + \epsilon' \tag{94}$$

so that

$$\begin{aligned}
x + \frac{\epsilon'}{f'(x^*)} &= x' + \epsilon' \\
x - x' &= \epsilon' - \frac{\epsilon'}{f'(x^*)} \\
x - x' &= \epsilon' \left(1 - \frac{1}{f'(x^*)}\right) \\
\epsilon' &= \frac{x - x'}{1 - \frac{1}{f'(x^*)}} \qquad (95)
\end{aligned}$$

Assuming $f'(x) \approx f'(x^*)$, we can estimate the error as

$$\epsilon' \approx \frac{x - x'}{1 - \frac{1}{f'(x)}} \tag{96}$$

With this, we can set a target error and run the relaxation method until we've achieved the target.

**Example: Relaxation Method Error**

Let's consider ferromagnetism again. In a system with $N$ spins, the magnetization is given (up to a constant) by

$$M = \sum_{i=1}^{N} \langle s_i \rangle \tag{97}$$

where $\langle s_i \rangle$ is the average value of spin $i$. We can write this as

$$M = N \langle s_i \rangle \tag{98}$$

if the selection of spin $i$ is arbitrary and all spins are equivalent, i.e. assuming all the spins behave similarly.

Recall that the energy of the magnetic system can be written as

$$E = -J \sum_{(ij)} s_i s_j - \mu H \sum_i s_i \tag{99}$$

where $J$ is the exchange constant, $\mu$ is the dipole moment, $H$ is an external magnetic field, and the sum in the first term is over neighboring spins.

Let's consider a system with only one spin in an external magnetic field $H$. There are only two possible states of the system (spin up or spin down, $s_1 = \pm 1$). In this case, the first term in the energy is zero (one spin has no neighbors to interact with), and only the second term contributes.

$$E_{\pm} = \mu H(\pm 1) \tag{100}$$

The probability of each state is

$$P_{\pm} = \frac{e^{\pm \mu H/(k_B T)}}{e^{\mu H/(k_B T)} + e^{-\mu H/(k_B T)}} \tag{101}$$

Given these probabilities, the average $\langle s_1 \rangle$ is given by

$$\begin{aligned}
\langle s_1 \rangle &= \frac{(1)P_+ + (-1)P_-}{P_+ + P_-} = \frac{P_+ - P_-}{1} = P_+ - P_- \\
\langle s_1 \rangle &= \frac{e^{\mu H/(k_B T)} - e^{-\mu H/(k_B T)}}{e^{\mu H/(k_B T)} + e^{-\mu H/(k_B T)}} \\
\langle s_1 \rangle &= \frac{\sinh(\mu H/(k_B T))}{\cosh(\mu H/(k_B T))} \\
\langle s_1 \rangle &= \tanh(\mu H/(k_B T)) \qquad (102)
\end{aligned}$$

where $\sinh, \cosh, \tanh$ are hyperbolic sine, cosine, and tangent.

This one spin system is then used to obtain an approximate solution for a system of $N$ spins. The assumption is that the interaction of $N$ spins is equivalent to one spin interacting with an *effective* magnetic field. This is called the "mean-field" approximation. For a system in which the external magnetic field is zero, the true energy is due to the spin interaction (the first term in the energy equation, $E_{\text{true}} = -J \sum_{(ij)} s_i s_j$). In the mean-field approximation, we will model the $N$-spin system as one spin interacting with a field $H_{\text{eff}}$ that is caused by the other $N-1$ spins, so the energy can be approximated by

$$E \approx -\mu H_{\text{eff}} s = \pm \mu H_{\text{eff}} \tag{103}$$

where the field $H_{\text{eff}}$ is due to all the other spins. If we can calculate $H_{\text{eff}}$, then the magnetization will be proportional to

$$M \sim \langle s \rangle = \tanh(\mu H_{\text{eff}}/(k_B T)) \tag{104}$$

Let's examine the energy equation again, and consider the contribution to the energy from one spin $i$:

$$\Delta E = -\left( J \sum_{\text{neighbors}} s_j \right) s_i - (\mu H)\, s_i \tag{105}$$

where the sum is over spin $i$'s neighbors. From this, it appears that the first term could be represented in terms of an effective field if

$$\mu H_{\text{eff}} = J \sum_{\text{neighbors}} s_j = Jz\langle s \rangle \tag{106}$$

We have

$$H_{\text{eff}} = \frac{J}{\mu} \sum_{\text{neighbors}} s_j = \frac{Jz}{\mu} \langle s \rangle \tag{107}$$

where $\langle s \rangle$ is the average neighboring spin and $z$ is the number of neighbors considered. Plugging this into Equation 104, we have

$$\langle s \rangle = \tanh \left( \frac{\mu}{k_B T} \frac{Jz}{\mu} \langle s \rangle \right)$$
$$\langle s \rangle = \tanh \left( \frac{Jz \langle s \rangle}{k_B T} \right) \tag{108}$$

Since the magnetization $M$ is proportional to the average spin $\langle s \rangle$ up to a constant, let's write this as

$$M = \tanh \left( \frac{JzM}{k_B T} \right) \tag{109}$$

To study the behavior of the magnetization as a function of temperature, let $J = 1$ and solve the equation

$$M = \tanh \frac{zM}{k_B T} \tag{110}$$

with

$$f(M) = \tanh \frac{zM}{k_B T} \tag{111}$$
$$f'(M) = \frac{z}{k_B T \cosh^2(zM/(k_B T))} \tag{112}$$

Clearly $M = 0$ is a solution, but let's find other solutions, as a function of $T$, measured in units of $1/k_B$. We have to assume some value for $z$, the number of neighbors, so let's use $z = 4$, the same number we considered in our simulations in the previous section.

**Example: Magnetization**

This is exactly the second order phase transition with a critical temperature that we explored with the Ising Model simulation. While $M = 0$ is always a solution, at low temperatures, there is another solution. Past the critical point, the $M = 0$ solution is the only one.

Figure 3 shows the same plot made based on our simlations. You may notice that we don't get the same value of the critical temperature with this method than we did with our simulations. The reason is the correlations that we investigated. Mean-field theory assumes that all spins are equivalent, when in reality, there is a high amount of correlation among near neighbors. Mean-field theory gives a good qualitative picture of the phase transition, but ignoring these local fluctuations causes mean-field theory to be off quantitatively, predicting

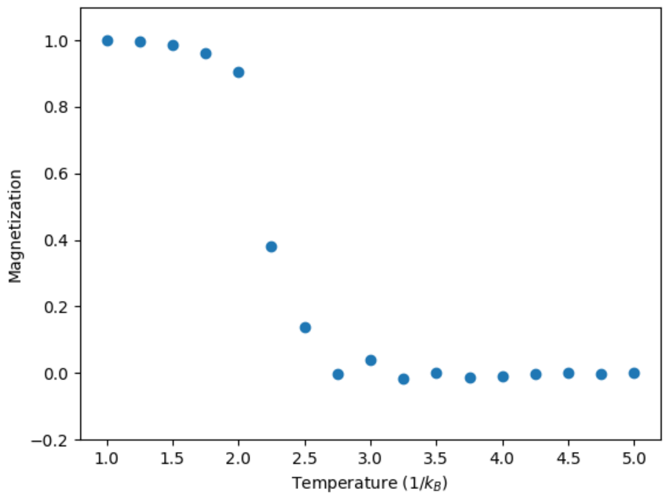


Figure 3:

the wrong critical value.

A nice feature of the relaxation method is that it extends easily to the solution of simultaneous nonlinear equations in two or more variables. Suppose we are given two equations to solve for two variables $x$ and $y$. We rearrange the equations to take the form:

$$x = f(x, y) \tag{113}$$

$$y = g(x, y) \tag{114}$$

Guess the initial starting values for both $x$ and $y$, substitude those values in the right hand side of both equations to get new values, and repeat until convergence. Though as before, convergence is not guaranteed.

This can be extended to any number of $N$ simultaneous equations in $N$ unknown variables.

## 4.2 Other Methods

A nonlinear equation for a single variable can always be arrange to put all the terms on one side of the equation, giving an equation of the form $f(x) = 0$. For our previous example, instead of

$$x = 2 - e^{-x} \tag{115}$$

we can have

$$2 - e^{-x} - x = 0 \tag{116}$$

Then finding the solution is equivalent to finding the roots (or zeroes) of $f(x)$.

Two common methods for root-finding are *binary search*, in which you start out with two values of $f$, one positive and one negative, and search in between until you find the value of the root, and *Newton's method*, in which you use the derivative of the function to guess at where the function crosses the axis and iterate.

A closely related problem is the problem of finding the maximum or minimum of a function. One method is to calculate the derivatives of the function, set them to zero, and then the techniques we've already discussed (matrix techniques if they are linear, or the relaxation method or Newton's method if they are not) to find the roots. For a function $f(x_1, x_2, ...x_n)$,

$$\frac{\partial f}{\partial x_i} = 0 \tag{117}$$

for all $i$.

If the derivatives cannot be calculated (because we don't have a given function for example), then the method of the *golden ratio search* can be used, which is similar to the binary search method for find roots.