Nair Homework 4

March 1, 2019

```
In [1]: %precision %g
        %matplotlib inline
        %config InlineBackend.figure_format = 'retina'

In [2]: from math import sqrt, pi, sin, cos, exp, inf, factorial, tan
        #from cmath import exp as cexp
        import numpy as np
        from numpy import linalg as LA
        from scipy import constants as C
        from scipy import integrate
        import matplotlib.pyplot as plt

        #from IPython.display import set_matplotlib_formats
        #set_matplotlib_formats('png', 'pdf')
```

1 CP 5.21 Electric field of a charge distribution

Electric potential due to a point charge is given by $\phi = \frac{q}{4\pi\epsilon_0 r}$.

The electric field can be found from this potential by taking partial derivatives, such that

$$\mathbf{E} = -\nabla\phi$$

I will define two partial derivative functions to find the field at respective points on the grid. The partial derivatives will be defined as according to the central difference

$$\frac{\partial f(x,y)}{\partial x} = \frac{f(x+\frac{h}{2},y) - f(x-\frac{h}{2},y)}{h}$$

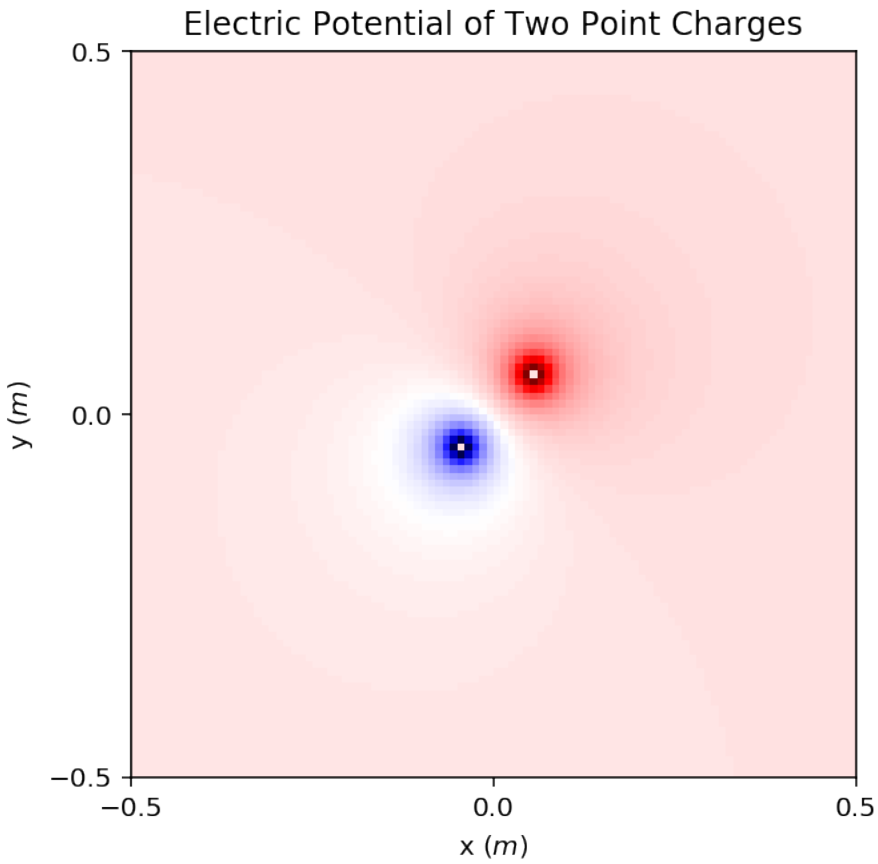
$$\frac{\partial f(x,y)}{\partial y} = \frac{f(x,y+\frac{h}{2}) - f(x,y-\frac{h}{2})}{h}$$

```
In [3]: q1 = 1 #charge 1 in coulombs
        q2 = -1 #charge 2 in coulombs
        e0 = C.epsilon_0

        def potential(q, r):
            return q / (4* pi* e0* r)
```

```
side = 1.0 #side length of the square (m)
points = 100 #number of grid points along each side
spacing = side / points #spacing of points (m)
separation = 0.1 #separation of charges (m)

x01 = side/2 + separation/2
y01 = side/2 + separation/2
x02 = side/2 - separation/2
y02 = side/2 - separation/2
```

In [4]:
```
#np.seterr(divide='ignore', invalid='ignore')
phi = np.empty([points,points],float)
#calculate the values in the array
for i in range(points):
    y = spacing * i
    for j in range(points):
        x = spacing * j
        r1 = sqrt((x-x01)**2 + (y-y01)**2)
        r2 = sqrt((x-x02)**2 + (y-y02)**2)
        if r1 != 0 and r2 != 0:
            phi[i,j] = potential(q1, r1) + potential(q2, r2)

fig1, ax1 = plt.subplots(1, 1, figsize = (5, 5))

#increases readability of plot
ax1.set_title("Electric Potential of Two Point Charges")
ax1.set_xlabel("x ($m$)")
ax1.set_ylabel("y ($m$)")
ax1.set_xticks([-0.5, 0, 0.5])
ax1.set_yticks([-0.5, 0, 0.5])

ax1.imshow(phi,origin="lower",extent=\
          [-0.5,0.5,-0.5,0.5],\
          cmap="seismic",vmax=7.5e11)
```

Out[4]:
```
<matplotlib.image.AxesImage at 0x10f42ec18>
```

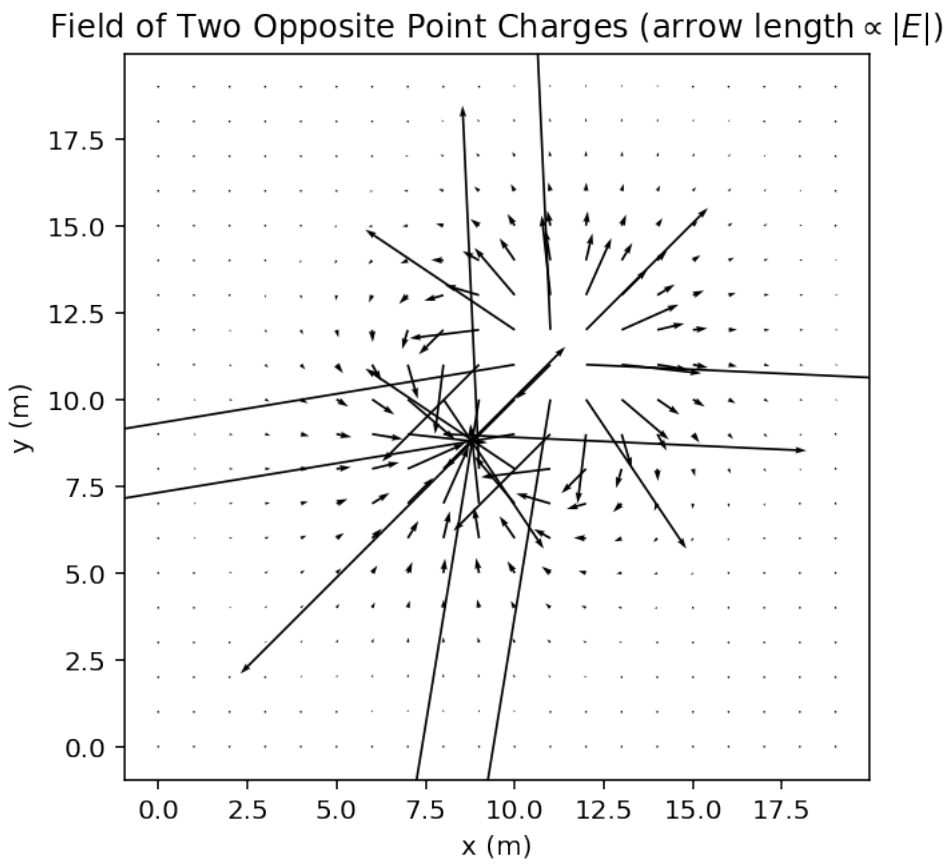


Because potential was only a function of distance from each point charge, it was easier to define it in terms of r. However, because we need to take the partial derivatives with respect to x and y separately, it is easier to redefine this potential function and then define partial derivative functions.

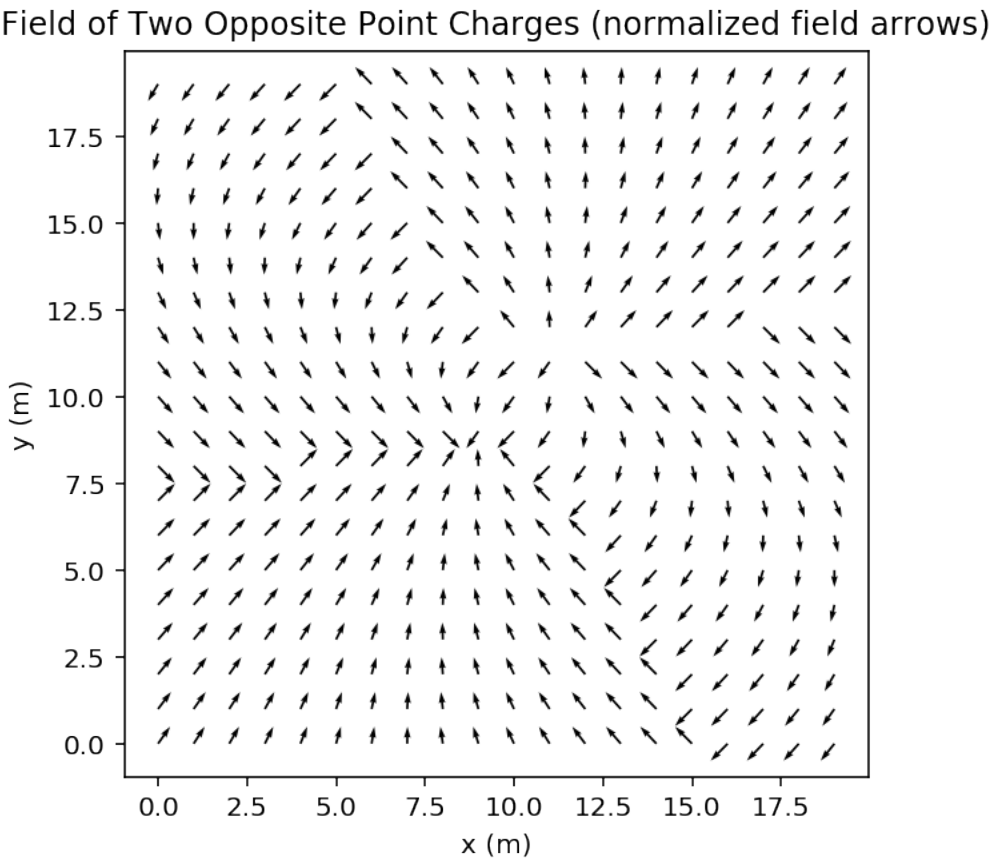
```
In [5]: def potential(q, x, y):
            r = sqrt(x**2 + y**2)
            return q / (4* pi* e0* r)

        """Creates partial derivative functions to
            find the field at points on the grid"""
        def Ex(q, x, y, h):

            partial = (potential(q,x+h/2,y) - potential(q,x-h/2,y)) / h
            return -partial

        def Ey(q, x, y, h):

            partial = (potential(q,x,y+h/2) - potential(q,x,y-h/2)) / h
            return -partial
```

In [6]:
```
h = 0.001
side = 1.0 #side length of the square (m)
points = 20 #number of grid points along each side
spacing = side / points #spacing of points (m)

x = np.linspace(-0.5,0.5,points)
y = np.linspace(-0.5,0.5,points)
E_x, E_y = np.meshgrid(x,y)

#calculate the values in the array
for i in range(points):
    y = spacing * i
    for j in range(points):
        x = spacing * j

        dx1 = x - x01
        dy1 = y - y01
        dx2 = x - x02
        dy2 = y - y02

        E_x[i,j] = Ex(q1, dx1, dy1, h) + Ex(q2, dx2, dy2, h)
        E_y[i,j] = Ey(q1, dx1, dy1, h) + Ey(q2, dx2, dy2, h)

#E_x /= np.sqrt(E_x**2 + E_y**2)
#E_y /= np.sqrt(E_x**2 + E_y**2)

fig2, ax2 = plt.subplots(1, 1, figsize = (5, 5))
ax2.set_xlabel("x (m)")
ax2.set_ylabel("y (m)")
ax2.set_title("Field of Two Opposite Point Charges\
 (arrow length$\propto |E|$)")
ax1.set_xticks([])
ax1.set_yticks([-0.5, 0, 0.5])
plt.quiver(E_x, E_y,headwidth=3)#, scale=1)
```

Out[6]:
```
<matplotlib.quiver.Quiver at 0x10762ec50>
```




```
In [7]: points = 10 #number of grid points along each side
        spacing = side / points #spacing of points (m)

        E_x /= np.sqrt(E_x**2 + E_y**2)
        E_y /= np.sqrt(E_x**2 + E_y**2)

        fig2, ax2 = plt.subplots(1, 1, figsize = (5, 5))
        ax2.set_xlabel("x (m)")
        ax2.set_ylabel("y (m)")
        ax2.set_title("Field of Two Opposite Point Charges\
         (normalized field arrows)")
        ax1.set_xticks([-0.5, 0, 0.5])
        ax1.set_yticks([-0.5, 0, 0.5])
        plt.quiver(E_x, E_y,headwidth=3,cmap='jet')
        #plt.scatter(x,y, color='c')

Out[7]: <matplotlib.quiver.Quiver at 0x113bfaf28>
```



While the quiver plot with varying arrow lengths contains more information, it's also difficult to discern the pattern of the field, which is clearly visible when all arrows are made the same length. I wasn't able to set the axes of the plots to reflect the distances as opposed to the number of points being used to create the vector field.

2 CP 6.1 Resistor circuit

Given the circuit diagram in the book, we can solve a system of equations to find the voltages at different points in the circuit. One equation given is

$$4V_1 - V_2 - V_3 - V_4 = V_+.$$

By writing out similar equations using Ohm's Law and Kirchoff's Junction Rule, we obtain the following equations.

$$3V_2 - V_1 - V_4 = 0$$

$$3V_3 - V_1 - V_4 = V_+$$

$$4V_4 - V_1 - V_2 - V_3 = 0.$$

We can write this in its vector-matrix form $A\mathbf{x} = \mathbf{v}$ with

$$A = \begin{pmatrix} 4 & -1 & -1 & -1 \\ -1 & 3 & 0 & -1 \\ -1 & 0 & 3 & -1 \\ -1 & -1 & -1 & 4 \end{pmatrix}, \mathbf{x} = \begin{pmatrix} V_1 \\ V_2 \\ V_3 \\ V_4 \end{pmatrix}, \text{and } \mathbf{v} = \begin{pmatrix} V_+ \\ 0 \\ V_+ \\ 0 \end{pmatrix}.$$

In [8]:
```
%%time

Vp = 5 #volts
A = np.array([[4,-1,-1,-1],
              [-1,3,0,-1],
              [-1,0,3,-1],
              [-1,-1,-1,4]], float)
v = np.array([Vp,0,Vp,0], float)
N = len(v)

#Gaussian elimination program w/out partial pivoting
for m in range(N):

    #Division by diagonal element
    div = A[m,m]
    A[m,:] /= div
    v[m] /= div

    #subtract from 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.empty(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("(V1, V2, V3, V4) = ({:2.2f},{:2.2f},{:2.2f},{:2.2f})"\
     .format(x[0],x[1],x[2],x[3]))
```

```
(V1, V2, V3, V4) = (3.00,1.67,3.33,2.00)
CPU times: user 306 µs, sys: 113 µs, total: 419 µs
Wall time: 345 µs
```

3 CP 6.2 Partial pivoting

Incorporating partial pivoting into the Gaussian elimination algorithm handles cases in which there are zeroes on the diagonal of our coefficient matrix. This would normally be a problem

because it would create a divide by zero error, however rows can be exchanged without effecting the solution, as long as they are changed consistently.

I will demonstrate that partial pivoting provides the same answers as standard Gaussian elimination when applied to the system in 6.1:

$$2w + x + 4y + z = -4$$
$$3w + 4x - y - z = 3$$
$$w - 4x + y + 5z = 9$$
$$2w - 2x + y + 3z = 7$$

In [9]:
```
"""This cell prints the result of using partial pivoting
    with Gaussian elimination to solve the above system"""
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 program w/ partial pivoting
for m in range(N):

    #Partial pivoting
    if A[m,m] == 0:
        A[[m,m+1]] = A[[m+1,m]]
        v[[m,m+1]] = v[[m+1,m]]

    #Division by diagonal element
    div = A[m,m]
    A[m,:] /= div
    v[m] /= div

    #subtract from 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.empty(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("(w, x, y, z) = ({:2.2f},{:2.2f},{:2.2f},{:2.2f})"\
              .format(x[0],x[1],x[2],x[3]))
```

```
(w, x, y, z) = (2.00,-1.00,-2.00,1.00)
```

```
In [10]: """This cell solves the system of equations
             without using partial pivoting"""
         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 program w/out partial pivoting
         for m in range(N):

             #Division by diagonal element
             div = A[m,m]
             A[m,:] /= div
             v[m] /= div

             #subtract from 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.empty(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("(w, x, y, z) = ({:2.2f},{:2.2f},{:2.2f},{:2.2f})"\
               .format(x[0],x[1],x[2],x[3]))
```

```
(w, x, y, z) = (2.00,-1.00,-2.00,1.00)
```

So, for that system, because there were no zeroes on the diagonal, the methods using partial pivoting and not performed equally well. Now, moving on to part (b), we attempt to solve the system in 6.17.

$$A = \begin{pmatrix} 0 & 1 & 4 & 1 \\ 3 & 4 & -1 & -1 \\ 1 & -4 & 1 & 5 \\ 2 & -2 & 1 & 3 \end{pmatrix} \text{ and } \mathbf{v} = \begin{pmatrix} -4 \\ 3 \\ 9 \\ 7 \end{pmatrix}.$$

Attempting to solve this, normal Gaussian elimination without partial pivoting fails.

```
In [11]: A = np.array([[0,1,4,1],
                       [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 program w/out partial pivoting
         for m in range(N):

             #Division by diagonal element
             div = A[m,m]
             A[m,:] /= div
             v[m] /= div

             #subtract from 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.empty(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("(w, x, y, z) = ({:2.2f},{:2.2f},{:2.2f},{:2.2f})"\
               .format(x[0],x[1],x[2],x[3]))
```

```
(w, x, y, z) = (nan,nan,nan,nan)


/Users/Varun/anaconda/lib/python3.6/site-packages/ipykernel_launcher.py:13: RuntimeWarning: divi
  del sys.path[0]
/Users/Varun/anaconda/lib/python3.6/site-packages/ipykernel_launcher.py:13: RuntimeWarning: inva
  del sys.path[0]
/Users/Varun/anaconda/lib/python3.6/site-packages/ipykernel_launcher.py:14: RuntimeWarning: divi

/Users/Varun/anaconda/lib/python3.6/site-packages/ipykernel_launcher.py:14: RuntimeWarning: inva
```

However, if we use the method of partial pivoting, the system is solvable.

```
In [12]: A = np.array([[0,1,4,1],
                       [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 program w/ partial pivoting
for m in range(N):

    #Division by diagonal element
    if A[m,m] == 0:
        A[[m,m+1]] = A[[m+1,m]]
        v[[m,m+1]] = v[[m+1,m]]
    div = A[m,m]
    A[m,:] /= div
    v[m] /= div

    #subtract from 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.empty(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("(w, x, y, z) = ({:2.2f},{:2.2f},{:2.2f},{:2.2f})"\
     .format(x[0],x[1],x[2],x[3]))
```

```
(w, x, y, z) = (1.62,-0.43,-1.24,1.38)
```

4 CP 6.3 LU decomposition

In LU decomposition, the goal is to factor the original matrix A into lower and upper triangular matrices, respectively. This is beneficial if we see multiple systems of equations where the set of coefficients on the unknowns are the same, but result in different set of solutions. By decomposing A into L and U, we don't need to solve the system from scratch every time. In LU decomposition, intermediate matrices $L_0, L_1, ..., L_n$ are calculated. For the final relation, the lower and upper triangular matrices are respectively

$$L = L_0^{-1}L_1^{-1}L_2^{-1}L_3^{-1}\cdots \qquad \text{and} \qquad U = \cdots L_3L_2L_1L_0A.$$

In [13]:
```
#matrix and vector from Eq 6.32
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)
```

```
In [14]: def LU(A):
             """Defines a function to perform LU decomposition
                 on a matrix A based on gausslim.py from book"""

             N = len(A)
             U = np.copy(A)
             L = np.zeros([N,N],float)

             #Gaussian elimination
             for m in range(N):

                 for i in range(m, N):
                     L[i,m] = U[i,m]

                 #Division by diagonal element
                 div = U[m,m]
                 U[m,:] /= div

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

             return L,U

         L, U = LU(A)

         L@U
```

```
Out[14]: array([[ 2.,  1.,  4.,  1.],
                [ 3.,  4., -1., -1.],
                [ 1., -4.,  1.,  5.],
                [ 2., -2.,  1.,  3.]])
```

So, we recovered the original matrix A when the matrices L and U are multiplied together. Now, implementing double backsubstitution, the equation $A\mathbf{x} = \mathbf{v}$ can be solved as

$$LU\mathbf{x} = \mathbf{v}$$

in two steps such that

$$U\mathbf{x} = \mathbf{y} \text{ and } L\mathbf{y} = \mathbf{v}.$$

```
In [15]: #LU decomposition with double backsubstitution
         def LU_with_DB(A, v):
```

```
        """Defines a function to perform LU decomposition
            on a matrix A based on gausslim.py from book.
            Then uses double backsubstitution to solve
            system LUx = v"""

        N = len(A)
        U = np.copy(A)
        L = np.zeros([N,N],float)
        w = np.copy(v)

        #Gaussian elimination
        for m in range(N):

            for i in range(m, N):
                L[i,m] = U[i,m]

            #Division by diagonal element
            div = U[m,m]
            U[m,:] /= div
            w[m] /= div

            #subtract from lower rows
            for i in range(m+1, N):
                mult = U[i,m]
                U[i,:] -= mult * U[m,:]
                w[i] -= mult * w[m]

        #first backsubstitution
        y = np.empty(N, float)
        for m in range(N-1, -1, -1):
            y[m] = w[m]
            for i in range(m+1, N):
                y[m] -= U[m,i] * y[i]

        #second backsubstitution
        x = np.empty(N, float)
        for m in range(N-1, -1, -1):
            x[m] = y[m]
            for i in range(m+1, N):
                x[m] -= L[m,i] * x[i]

        return x

LU_with_DB(A,v)
```

```
Out[15]: array([ 2., -1., -2.,  1.])
```

So we arrive at the solution $\mathbf{x} = \begin{pmatrix} 2, & -1, & -2, & 1 \end{pmatrix}^{\mathrm{T}}$, which is confirmed by the solve function from the numpy library below.

```
In [16]: LA.solve(A,v)

Out[16]: array([ 2., -1., -2.,  1.])
```

5 CP 6.4 A circuit of resistors (revisited)

Now, I'll solve the same circuit from exercise 6.1 using the built-in functionality of numpy. The physical equations I derived still hold and will form the matrix and solution space we solve for. The system of equations is

$$4V_1 - V_2 - V_3 - V_4 = V_+$$
$$3V_2 - V_1 - V_4 = 0$$
$$3V_3 - V_1 - V_4 = V_+$$
$$4V_4 - V_1 - V_2 - V_3 = 0$$

where again we have it in matrix form given by $A\mathbf{x} = \mathbf{v}$ with

$$A = \begin{pmatrix} 4 & -1 & -1 & -1 \\ -1 & 3 & 0 & -1 \\ -1 & 0 & 3 & -1 \\ -1 & -1 & -1 & 4 \end{pmatrix}, \ \mathbf{x} = \begin{pmatrix} V_1 \\ V_2 \\ V_3 \\ V_4 \end{pmatrix}, \text{and } \ \mathbf{v} = \begin{pmatrix} V_+ \\ 0 \\ V_+ \\ 0 \end{pmatrix}.$$

```
In [17]: %%time

         Vp = 5 #volts
         A = np.array([[4,-1,-1,-1],
                       [-1,3,0,-1],
                       [-1,0,3,-1],
                       [-1,-1,-1,4]], float)
         v = np.array([Vp,0,Vp,0], float)

         x = LA.solve(A,v)
         x

CPU times: user 463 µs, sys: 270 µs, total: 733 µs
Wall time: 596 µs
```

This is the same answer given to us by the "hand made" Gaussian elimination function whereas in this case it took approximately a fourth to a fifth of the time to run, which was negligible for such a small system, but could prove importantly different for larger cases.

6 CP 6.7 A chain of resistors

Ohm's law tells us that $V = IR$, and Kirchhoff's current junction law says that $\sum_i I_i = 0$ for a given junction in a circuit. So by Ohm's law, the currents flowing into the junction 1 are $\frac{V_1 - V_+}{R}$, $\frac{V_1 - V_2}{R}$, and $\frac{V_1 - V_3}{R}$. So the Kirchhoff's junction rule for 1 gives the equation

$$\frac{V_1 - V_+}{R} + \frac{V_1 - V_2}{R} + \frac{V_1 - V_3}{R} = 0.$$

Multiplying through by the common denominator R and collecting like terms and unknowns gives the first equation in the system provided in the book:

$$3V_1 - V_2 - V_3 = V_+.$$

The 1st and N^{th} junction are the edge cases where only three wires connect to the junction and hence only three currents flow in/out. Every other junction in the chain will have four currents flowing in/out that correspond to currents flowing from the $i^{\text{th}} - 2$, $i^{\text{th}} - 1$, $i^{\text{th}} + 1$, and $i^{\text{th}} + 2$ junctions. Similarly, V_+ will not be the potential at an adjacent junction for any other than the first and second, so the second also becomes an edge case. We can write the general equation as

$$\frac{V_i - V_{i-2}}{R} + \frac{V_i - V_{i-1}}{R} + \frac{V_i - V_{i+1}}{R} + \frac{V_i - V_{i+2}}{R} = 0.$$

Once we have this, the system provided in the book is clearly reached where

$$-V_{i-2} - V_{i-1} + 4V_i - V_{i+1} - V_{i+2} = 0.$$

We can rewrite this system of equations in matrix form that satisfies the relation $A\mathbf{v} = \mathbf{w}$. These are given by $A\mathbf{x} = \mathbf{v}$ with

$$A = \begin{pmatrix} 3 & -1 & -1 & 0 & 0 & 0 & \cdots & 0 \\ -1 & 4 & -1 & -1 & 0 & 0 & \cdots & 0 \\ -1 & -1 & 4 & -1 & -1 & 0 & \cdots & 0 \\ 0 & -1 & -1 & 4 & -1 & -1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & \cdots & -1 & -1 & 4 & -1 & -1 & 0 \\ 0 & \cdots & 0 & -1 & -1 & 4 & -1 & -1 \\ 0 & \cdots & 0 & 0 & -1 & -1 & 4 & -1 \\ 0 & \cdots & 0 & 0 & 0 & -1 & -1 & 3 \end{pmatrix}, \quad \mathbf{v} = \begin{pmatrix} V_1 \\ V_2 \\ V_3 \\ V_4 \\ \vdots \\ V_{N-3} \\ V_{N-2} \\ V_{N-1} \\ V_N \end{pmatrix}, \text{ and } \quad \mathbf{w} = \begin{pmatrix} V_+ \\ V_+ \\ 0 \\ 0 \\ 0 \\ \vdots \\ 0 \\ 0 \\ 0 \end{pmatrix}.$$

Because all the points in the circuit are connected to at most 4 other points, many of the off-diagonal elements are zero. We see that are matrix of coefficients A is banded and can thus be solved in a less computationally intensive manner than iterating through every element.

```
In [29]: """Solving a banded system of linear equations as defined

             by Mark Newman used for the following problems in the

             homework. I edited the numpy functions, adding the

             prefix 'np.' so it would run properly in my notebook'"""

         def banded(Aa,va,up,down):

             # Copy the inputs and determine the size of the system
             A = np.copy(Aa)
             v = np.copy(va)
             N = len(v)

             # Gaussian elimination
             for m in range(N):
```

```
            # Normalization factor
            div = A[up,m]

            # Update the vector first
            v[m] /= div
            for k in range(1,down+1):
                if m+k<N:
                    v[m+k] -= A[up+k,m]*v[m]

            # Now normalize the pivot row of A and subtract from lower ones
            for i in range(up):
                j = m + up - i
                if j<N:
                    A[i,j] /= div
                    for k in range(1,down+1):
                        A[i+k,j] -= A[up+k,m]*A[i,j]

        # Backsubstitution
        for m in range(N-2,-1,-1):
            for i in range(up):
                j = m + up - i
                if j<N:
                    v[m] -= A[i,j]*v[j]

        return v
```

In [30]:
```
#my banded system function solver
def banded_solve(Aa,va,bw):
    """Solves banded system of linear equations
        with bandwidth bw above and below diagonal.
        Takes an NxN matrix, vector, and number of
        elements above and below diagonal to look at
        as arguments."""

    # Copy the inputs and determine the size of the system
    A = np.copy(Aa)
    v = np.copy(va)
    N = len(v)
    s = 0 #variable to have something to do in except clause

    #Gaussian elimination program w/out partial pivoting
    for m in range(N):

        #Division by diagonal element
        div = A[m,m]
        A[m,m] /= div
```

```
            try:
                for j in range(m-bw,m+bw+1):
                    A[m, j] /= div
            except:
                s += 1

            v[m] /= div

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

                try:
                    for j in range(m-bw,m+bw+1):
                        A[i,j] -= mult * A[m,j]
                except:
                    s += 1

                v[i] -= mult * v[m]

        #backsubstitution
        x = np.empty(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]

        return x
```

In [31]:
```
#cell to define problem specific matrices and vectors
Vp = 5 #volts
N = 6

#matrix for my function
A = np.zeros([N,N],float)
for i in range(N-2):
    A[i,i] = 4
    A[i,i+1] = -1
    A[i,i+2] = -1
    A[i+1,i] = -1
    A[i+2,i] = -1

A[0,0] = 3
A[N-1,N-1] = 3
A[N-2,N-2] = 4
A[N-2,N-1] = -1
A[N-1,N-2] = -1
```

```
#matrix for banded.py function
B = np.empty([5,N],float)
B[0,:] = -1
B[1,:] = -1
B[2,:] = 4
B[3,:] = -1
B[4,:] = -1

B[2,0] = 3
B[2,N-1] = 3

w = np.zeros(N,float)
w[0] = Vp
w[1] = Vp

print(banded(B,w,2,2))
print(banded_solve(A,w,2))
print(LA.solve(A,w))
```

```
[ 3.7254902   3.43137255  2.74509804  2.25490196  1.56862745  1.2745098 ]
[ 3.7254902   3.43137255  2.74509804  2.25490196  1.56862745  1.2745098 ]
[ 3.7254902   3.43137255  2.74509804  2.25490196  1.56862745  1.2745098 ]
```

So for the circuit where $N = 6$, the results of the function I wrote to solve banded systems of linear equations match the results of Newman's function and the function in the numpy.linalg library. For the case where $N = 10000$, the results also match the results of the other functions. I'll graph the voltages at each point next.

In [32]:
```
#cell to define problem specific matrices and vectors
Vp = 5 #volts
N = 10000

#matrix for my function
A = np.zeros([N,N],float)
for i in range(N-2):
    A[i,i] = 4
    A[i,i+1] = -1
    A[i,i+2] = -1
    A[i+1,i] = -1
    A[i+2,i] = -1

A[0,0] = 3
A[N-1,N-1] = 3
A[N-2,N-2] = 4
A[N-2,N-1] = -1
```

```
A[N-1,N-2] = -1

w = np.zeros(N,float)
w[0] = Vp
w[1] = Vp

x = np.arange(1,10001,1)
y = banded_solve(A,w,2)

fig3, ax3 = plt.subplots(1, 1, figsize = (10, 5))

#increases readability of plot
ax3.set_title("A Chain of Resistors")
ax3.set_xlabel("Internal Point")
ax3.set_ylabel("Internal Voltage (V)")
plt.plot(x,y)
plt.show()
```

7 CP 6.8 The QR algorithm

The problem introduces the following relations given the column vectors $\mathbf{a}_0, \mathbf{a}_1, ..., \mathbf{a}_{N-1}$ of the matrix A:

$$\mathbf{u}_i = \mathbf{a}_i - \sum_{j=0}^{i-1} (\mathbf{q}_j \cdot \mathbf{a}_i)\mathbf{q}_j \qquad \text{and} \qquad \mathbf{q}_i = \frac{\mathbf{u}_i}{|\mathbf{u}_i|}.$$

From this we can rearrange the equations to give us a system of equations that begins as

$$\mathbf{a}_0 = |\mathbf{u}_0|\,\mathbf{q}_0, \qquad \mathbf{a}_1 = |\mathbf{u}_1|\,\mathbf{q}_1 + (\mathbf{q}_0 \cdot \mathbf{a}_1)\mathbf{q}_0, \qquad \mathbf{a}_2 = |\mathbf{u}_2|\,\mathbf{q}_2 + (\mathbf{q}_0 \cdot \mathbf{a}_2)\mathbf{q}_0 + (\mathbf{q}_1 \cdot \mathbf{a}_2)\mathbf{q}_1.$$

These can be written in matrix form as

$$A = \begin{pmatrix} | & | & | & \cdots \\ \mathbf{a}_0 & \mathbf{a}_1 & \mathbf{a}_2 & \cdots \\ | & | & | & \cdots \end{pmatrix} = \begin{pmatrix} | & | & | & \cdots \\ \mathbf{q}_0 & \mathbf{q}_1 & \mathbf{q}_2 & \cdots \\ | & | & | & \cdots \end{pmatrix} \begin{pmatrix} |\mathbf{u}_0| & \mathbf{q}_0 \cdot \mathbf{a}_1 & \mathbf{q}_0 \cdot \mathbf{a}_2 & \cdots \\ 0 & |\mathbf{u}_1| & \mathbf{q}_1 \cdot \mathbf{a}_2 & \cdots \\ 0 & 0 & |\mathbf{u}_2| & \cdots \end{pmatrix} = QR.$$

Where Q is the orthonormal and R is upper triangular, so we have successfully completed the QR decomposition. We can test the QR decomposition on the matrix A to make sure we get back the same matrix.

$$A = \begin{pmatrix} 1 & 4 & 8 & 4 \\ 4 & 2 & 3 & 7 \\ 8 & 3 & 6 & 9 \\ 4 & 7 & 9 & 2 \end{pmatrix}.$$

```
In [22]: A = np.array([[1,4,8,4],
                       [4,2,3,7],
                       [8,3,6,9],
                       [4,7,9,2]], float)
```

```
        def QR(A):
            """With a matrix A as its argument, finds QR decomposition
                and returns those two matrices"""

            N = len(A)
            a = np.copy(A)
            u = np.zeros((N,N), float)
            q = np.zeros((N,N), float)

            #calculates orthonormal Q and intermediate U
            for i in range(N):
                s = 0
                for j in range(i):
                    s += (np.dot(q[:,j],a[:,i])) * q[:,j]

                u[:,i] = a[:,i] - s
                q[:,i] = u[:,i] / LA.norm(u[:,i])
            #q[:,0] *= -1

            r = np.zeros((N,N), float)

            #calculates upper triangular R
            for i in range(N):
                r[i,i] = LA.norm(u[:,i])
                for j in range(i):
                    r[j,i] = np.dot(q[:,j],a[:,i])
```

```
            return q,r

        q, r = QR(A)
        print("The orthonormal matrix Q is"), print(q)
        print("\n")
        print("The upper triangular matrix R is"), print(r)
        print("\n")
        print("Multiplied together, they return A.")
        print(q@r)
```

```
The orthonormal matrix Q is
[[ 0.10153462  0.558463    0.80981107  0.1483773 ]
 [ 0.40613847 -0.10686638 -0.14147555  0.8964462 ]
 [ 0.81227693 -0.38092692  0.22995024 -0.37712564]
 [ 0.40613847  0.72910447 -0.5208777  -0.17928924]]


The upper triangular matrix R is
[[  9.8488578    6.49821546  10.55960012  11.37187705]
 [  0.           5.98106979   8.4234836   -0.484346  ]
 [  0.           0.           2.74586406   3.27671222]
 [  0.           0.           0.           3.11592335]]


Multiplied together, they return A.
[[ 1.  4.  8.  4.]
 [ 4.  2.  3.  7.]
 [ 8.  3.  6.  9.]
 [ 4.  7.  9.  2.]]
```

```
In [23]: def eigenvalues(A):
            """This function returns the eigenvalues of a matrix
                through QR decomposition and then diagonalization.
                It's tolerance is 10^-6 for non-diagonal elements"""

            a = np.copy(A) #will be the diagonalized matrix
            N = len(a)
            epsilon = 10e-6
            V = np.identity(N)
            max_value = 1

            #runs until a certain accuracy has been achieved
            while max_value > epsilon:
                Q,R = QR(a)
                a = R@Q
                V = V@Q
```

```
            #finds largest off-diagonal element by masking diagonal
            mask = np.ones(a.shape, dtype=bool)
            np.fill_diagonal(mask, 0)
            max_value = a[mask].max()

        eigvals = np.zeros(N,float)
        for i in range(N):
            eigvals[i] = a[i,i]

        return eigvals

    print("The eigenvalues of A match the expected values.")
    print(eigenvalues(A))
```

```
The eigenvalues of A match the expected values.
[ 21.  -8.  -3.   1.]
```

8 CP 6.9 Asymmetric quantum well

Unlike the previous qunatum well problem, the asymmetry of this problem prevents it from being solved analytically, however formatting it as a system of equations allows it to be solved numerically. The particle in the given well obeys the relation $\mathbf{H}\psi(x) = E\psi(x)$ where

$$\mathbf{H} = -\frac{\hbar^2}{2M}\frac{d^2}{dx^2} + V(x).$$

We can assume the wavefunction goes to zero outside the well because the potential is infinitely high for $x < 0$ and $x > L$. Thus we can Fourier transform the the wavefunction

$$\psi(x) = \sum_{n=1}^{\infty} \psi_n \sin\frac{\pi n x}{L}.$$

The Schrodinger equation $\mathbf{H}\psi = E\psi$ implies that

$$\sum_{n=1}^{\infty} \psi_n \int_0^L \sin\frac{\pi m x}{L}\mathbf{H}\sin\frac{\pi n x}{L}\,dx = \tfrac{1}{2}LE\psi_m.$$

This can be derived by expanding the Schrodinger equation so that it reads

$$\mathbf{H}\left(\sum_{n=1}^{\infty} \psi_n \sin\frac{\pi n x}{L}\right) = E\left(\sum_{m=1}^{\infty} \psi_m \sin\frac{\pi m x}{L}\right).$$

From here, we can multiply both sides by $\sin\frac{\pi m x}{L}$ and reorder the terms on each side so that we have

$$\left(\sum_{n=1}^{\infty} \psi_n \sin\frac{\pi m x}{L}\mathbf{H}\sin\frac{\pi n x}{L}\right) = E\left(\sum_{m=1}^{\infty} \psi_m \sin\frac{\pi m x}{L}\sin\frac{\pi m x}{L}\right).$$

We can integrate both sides from 0 to L with respect to x giving

$$\sum_{n=1}^{\infty} \psi_n \int_0^L \left(\sin \frac{\pi m x}{L} \mathbf{H} \sin \frac{\pi n x}{L} \right) dx = E \sum_{m=1}^{\infty} \psi_m \int_0^L \sin \frac{\pi m x}{L} \sin \frac{\pi m x}{L} \, dx = E \sum_{m=1}^{\infty} \psi_m \frac{L}{2}$$

by properties of Fourier Series. Thus we have our final result. The Schrodinger equation combined with the fact that the potential is infinite outside of the bounds of the quantum well together imply that

$$\sum_{n=1}^{\infty} \psi_n \int_0^L \sin \frac{\pi m x}{L} \mathbf{H} \sin \frac{\pi n x}{L} \, dx = \tfrac{1}{2} L E \psi_m.$$

If the Hamiltonian matrix $\mathbf{H}$ is defined according to

$$H_{mn} = \frac{2}{L} \int_0^L \sin \frac{\pi m x}{L} \mathbf{H} \sin \frac{\pi n x}{L} \, dx$$

which can be expanded to include the value of the Hamiltonian

$$H_{mn} = \frac{2}{L} \int_0^L \sin \frac{\pi m x}{L} \left[-\frac{\hbar^2}{2M} \frac{d^2}{dx^2} + V(x) \right] \sin \frac{\pi n x}{L} \, dx.$$

Taking the case where $V(x) = \frac{ax}{L}$ gives the expression

$$H_{mn} = \frac{2}{L} \int_0^L \sin \frac{\pi m x}{L} \left[-\frac{\hbar^2}{2M} \frac{d^2}{dx^2} + \frac{ax}{L} \right] \sin \frac{\pi n x}{L} \, dx.$$

This can be expanded so

$$H_{mn} = \frac{2}{L} \left[\int_0^L \frac{-\hbar^2}{2M} \sin \frac{\pi m x}{L} \left(\frac{d^2}{dx^2} \sin \frac{\pi n x}{L} \right) dx + \int_0^L \frac{ax}{L} \sin \frac{\pi m x}{L} \sin \frac{\pi n x}{L} \, dx \right].$$

$$H_{mn} = \frac{2}{L} \left[\frac{-\hbar^2}{2M} \int_0^L \sin \frac{\pi m x}{L} \left(-\left(\frac{\pi n}{L} \right)^2 \sin \frac{\pi n x}{L} \right) dx + \frac{a}{L} \int_0^L x \sin \frac{\pi m x}{L} \sin \frac{\pi n x}{L} \, dx \right].$$

$$H_{mn} = \frac{2}{L} \left[\frac{\hbar^2}{2M} \left(\frac{\pi n}{L} \right)^2 \int_0^L \sin \frac{\pi m x}{L} \sin \frac{\pi n x}{L} \, dx + \frac{a}{L} \int_0^L x \sin \frac{\pi m x}{L} \sin \frac{\pi n x}{L} \, dx \right].$$

This is the point where we create separate expressions for the cases where $m = n$ and where $m \neq n$ but one is even and one is odd. This is because the integrals take on different values based on the relationship between m and n. From the givens in the problem

$$\int_0^L x \sin \frac{\pi m x}{L} \sin \frac{\pi n x}{L} \, dx = \begin{cases} 0 & \text{if } m \neq n \text{ and both even or both odd,} \\ -\left(\frac{2L}{\pi} \right)^2 \frac{mn}{(m^2 - n^2)^2} & \text{if } m \neq n \text{ and one is even, one is odd,} \\ L^2/4 & \text{if } m = n \end{cases}$$

and

$$\int_0^L \sin\frac{\pi m x}{L}\,\sin\frac{\pi n x}{L}\,dx = \begin{cases} L/2 & \text{if } m = n, \\ 0 & \text{otherwise.} \end{cases}$$

So after doing the algebra for the cases, we are left with three expressions expressions for the three different cases (the second integral expression's "otherwise" case is split between two of the above cases.

$$H_{mn} = \frac{2}{L}\left[\frac{\hbar^2}{2M}\left(\frac{\pi n}{L}\right)^2 \int_0^L \sin\frac{\pi m x}{L}\sin\frac{\pi n x}{L}\,dx + \frac{a}{L}\int_0^L x\sin\frac{\pi m x}{L}\sin\frac{\pi n x}{L}\,dx\right].$$

The diagonal elements of the Hamiltonian (when $m = n$) are given by

$$H_{mn} = \frac{2}{L}\left[\frac{\hbar^2}{2M}\left(\frac{\pi n}{L}\right)^2\frac{L}{2} + \frac{aL^2}{4L}\right] \forall\, m = n$$

$$H_{mn} = \frac{\hbar^2}{2M}\left(\frac{\pi n}{L}\right)^2 + \frac{a}{2} \forall\, m = n. \tag{1}$$

The off-diagonal elements of the Hamiltonian (for the cases $m \neq n$ but only one is even) are gven by

$$H_{mn} = -\frac{2}{L}\left(\frac{a}{L}\right)\left(\frac{2L}{\pi}\right)^2 \frac{mn}{(m^2-n^2)^2} \forall\, m \neq n \text{ and one is even, one is odd}$$

$$H_{mn} = \frac{-8amn}{\pi^2(m^2-n^2)^2} \forall\, m \neq n \text{ and one is even, one is odd.} \tag{2}$$

Trivially, we have that

$$H_{mn} = 0 \,\forall\, m \neq n \text{ and both even, or both odd.} \tag{3}$$

And thus with equations (1), (2) and (3), the Hamiltonian can be populated. After, the diagonalized Hamiltonian of it yields the eigenvalues, which are the observable energy states we can find the particle in if exposed to this potential in a quantum well.

```
In [36]: #defines constants
         L = 5 #width of well (m) = 5 angstroms
         a = 10 #potential parameter (eV)
         m = C.m_e #mass of electron in the well (kg)
         h = C.hbar #Planck's reduced constant

In [37]: def Hmn(m,n):
             """Returns the mn element of the Hamiltonian H
                 based on the parameters specified above"""

             if m==n:
                 return ((h**2)/(2*m))*((pi*n)/(L))**2 + a/2
             elif m!=n:
                 if (m+n) % 2 == 1:
                     num = -8*a*m*n
```

```
                den = (pi**2) * ((m**2 - n**2)**2)
                return num / den
            else:
                return 0

    def Hamiltonian(N):
        """Creates the Hamiltonian for the given system
            with dimensions NxN"""

        H = np.zeros([N,N], float)
        for m in range(1,N+1):
            for n in range(1,N+1):
                H[m-1,n-1] = Hmn(m,n)

        return H

    LA.eigvals(Hamiltonian(10))

    print("The ground state eigenvalue is {:4.2f} eV."\
          .format(np.min(LA.eigvals(Hamiltonian(10)))))
```

```
The ground state eigenvalue is 5.84 eV.
```

9 CP 6.16 The Lagrange point

The distance between the Earth and the Moon where a satellite remains in synchronous orbit with the two is called the Lagrange point. At this point, the gravitational pull towards each of the Earth and the Moon creates the right balance of centripetal force for the orbit to be synchronous. If we assume circular orbits and that the mass of the Earth is much larger than either the Moon or the satellite, the Lagrange point L_1 must satisfy this equation for r

$$\frac{GM}{r^2} - \frac{Gm}{(R-r)^2} = \omega^2 r.$$

Say that the Earth and Moon have mass M and m, respectively. Say the satellite in question has mass m_1. Then, as can be seen in the diagram, the distance from the Earth to the satellite and from the Moon to the satellite are r and $R-r$, respectively. So we can write the two forces acting on the satellite as $F_{\text{Earth}} = \frac{GMm_1}{r^2}$ and $F_{\text{Moon}} = \frac{Gmm_1}{(R-r)^2}$. We can sum these forces to give the total force acting on the satellite as

$$\frac{GMm_1}{r^2} - \frac{Gmm_1}{(R-r)^2} = m_1 a_c = m_1\left(\frac{v^2}{r}\right) = m_1\omega^2 r$$

because $\omega = \frac{v}{r}$. Now, dividing by m_1 gives us the desired equation:

$$\frac{GM}{r^2} - \frac{Gm}{(R-r)^2} = \omega^2 r.$$

The general form of the secant method is to repeatedly solve the equation

$$x_3 = x_2 - f(x_2)\frac{x_2 - x_1}{f(x_2) - f(x_1)}$$

until the difference between x_3 and x_2 remains smaller than the desired accuracy for a few iterations. So in this case, we will be searching for r_3 and comparing to the known value of the Lagrange point $L_1 = 326045$ km. If we call our error δ, then $\delta = r_3 - r_2$. So we can write the error for the secant method as

$$\delta = -f(r_2)\frac{r_2 - r_1}{f(r_2) - f(r_1)}.$$

However, to solve this equation, it would be more useful if we write it as a function $f(r)$, so multiplying out the fraction on the left hand side leaves us with

$$GM(R-r)^2 - Gmr^2 = \omega^2 r^3 (R-r)^2.$$

We can then move everything to one side of the equation so that

$$0 = GM(R-r)^2 - Gmr^2 - \omega^2 r^3 (R-r)^2.$$

In [26]:
```
#defining constants
G = C.G #gravitational constant
M = 5.974e24 #earth mass (kg)
m = 7.348e22 #moon mass (kg)
R = 3.844e8 #distance to moon (m)
omega = 2.662e-6 #angular velocity (s^-1)

#secant method test function
def f(x):
    return sin(x)*tan(3*x) + exp(x) + x**3

#function for Lagrange point
def g(r):
    p1 = G*M*((R-r)**2)
    p2 = G*m*(r**2)
    p3 = (omega**2)*(r**3)*((R-r)**2)
    return p1 - p2 - p3
```

In [27]:
```
def secant(r1, r2):
    """Given two starting points x1 and x2, this
        approximates the root of a function to
        four significant digits using the secant
        method as described in the book"""

    accuracy = 1e-5
    delta = 1
    while abs(delta) > accuracy:
        delta = g(r2)*( (r2-r1) / (g(r2)-g(r1)) )
```

```
            r1 = r2
            r2 -= delta

        return r2
```

```
In [28]: print("The Lagrange point is {:4.4f} m from Earth."\
              .format(secant(100000,500)))
```

```
The Lagrange point is 326045420.2403 m from Earth.
```

This matches the known value of $r_{L_1} = 326045$ km.