PHY407
Lab #4: Solving linear and non-linear equations

Q1: Hayley Agler, Q2: Hayley Agler, Q3: Nicholas Pavanel

October 2020

Q1

a)

Nothing to submit.

b)

The time taken by LU decomposition is significantly shorter than the times taken by Gaussian and partial pivoting, as can be seen in 2. Partial pivoting takes slightly more time to compute than Gaussian but is very similar; this is expected as the two methods vary only by the addition of a partial pivoting program. The error of the Gaussian and partial pivoting methods is the same as can be seen from the plot in 1 , the error of LU decomposition is largely similar to that of Gaussian and partial pivoting, however at some points has slightly less error and is therefore more accurate.

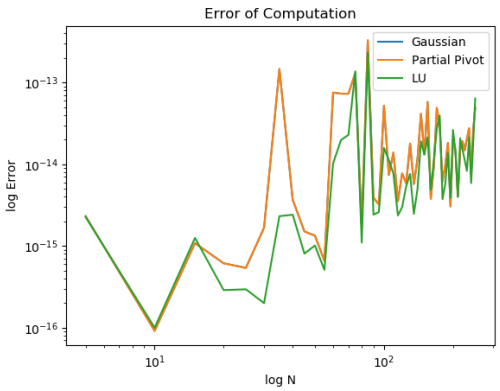

Figure 1: Error of each computation method.

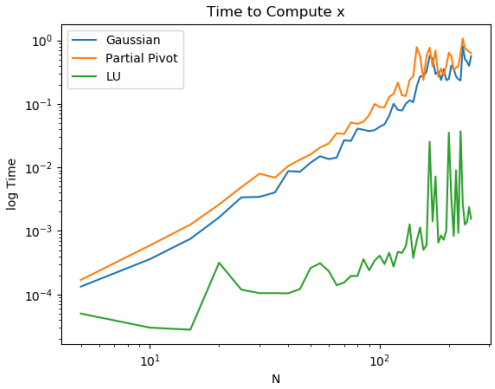

Figure 2: Time taken for each method to compute x.

c)

The real voltages were plotted in 3 and the voltages with the R6 resistor replaced with an inductor. The effect of replacing the resistor R6 by the imaginary impedance of the same magnitude can be seen in 4, where the amplitude of V3 has increased from about 1.5 to almost 3 Volts. The output for x_1, x_2, x_3 before adding the impedence is:

```
[1.69369369-0.16216216j, 1.45045045+0.2972973j,  1.85585586-0.13513514j]
```

The output for x_1, x_2, x_3 after adding the impedence is:

```
[1.69369369-0.16216216j, 1.45045045+0.2972973j,  1.85585586-0.13513514j]
```

Before adding the impedence, the amplitudes for the voltages are $|V_1| = 1.701439, |V_2| = 1.480605, |V_3| = 1.860769$ and the phases are $\theta_1 = -5.469095, \theta_2 = 11.5834186, \theta_3 = -4.164672$.

With the impedence, the amplitudes for the voltages are $|V_1| = 1.562118, |V_2| = 1.4994287, |V_3| = 2.8112764$ and the phases are $\theta_1 = -4.025909, \theta_2 = 21.6392826, \theta_3 = 14.352479$. From these numbers, we can tell that the impedence had a significant effect on the phase of each x, and although it had some effect on the voltages amplitudes, it had the greatest effect on the amplitude of V_3

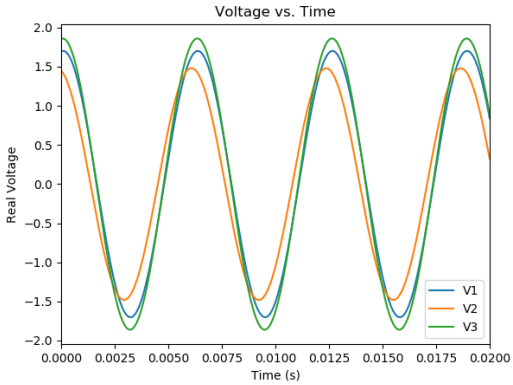

Figure 3: The real voltages are plotted over a range of 0.02 seconds.

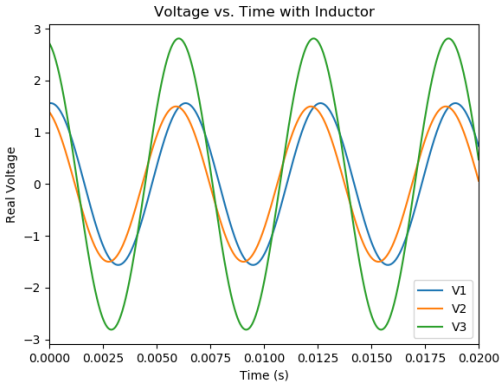

Figure 4: The real voltages are plotted again but with the resistor R6 replaced with an imaginary impedence of the same magnitude.

Q2

To compute the Hamiltonian matrix H_{mn} the following programs were used. Firstly, the components of the matrix were computed with:

```
#modelled from page 4 of lab handout
def H_mn(m,n):
    if m != n and m % 2 == 0 and n % 2 == 0:
        return 0
    elif m != n and m % 2 == 1 and n % 2 == 1:
        return 0
    elif m != n and m % 2 == 0 and n % 2 == 1 or m != n and m % 2 == 1 and n % 2 == 0:
        return - (8 * a * m * n) / (np.pi**2 * (m**2 - n**2)**2)
    elif m == n:
        return 0.5 * a + (np.pi**2 * hbar**2 * m**2) / (2 * M * L**2)
```

and secondly, the matrix was assembled with:

```
def H_matrix(size):
    output = np.zeros((size,size))
    for i in range(1, size + 1):
        for j in range(1, size + 1):
            output[i-1, j-1] = H_mn(i,j) #using lab handout method
    return output
```

. Using the above programs the first ten energy levels of the quantum well were computed from a 10x10 Hamiltonian matrix. These are shown below:

```
array([  5.83683216,  11.18196154,  18.66433505,  29.14644637,
```

```
        42.65836143,  59.1898144 ,  78.73541835, 101.29327501,
       126.86114119, 155.56729014])
```

. Note that the ground state energy is 5.84eV. The first ten energy levels of the quantum well were the computed from a 100x100 Hamiltonian matrix. These are shown below:

```
array([  5.83683176,  11.18196021,  18.66433318,  29.14643757,
        42.65835231,  59.18976181,  78.73536652, 101.29264403,
       126.86030879, 155.43765724])
```

Note the similarity between the energy values. This implies that the calculations are quite accurate.

The eigenvalues and eigenvectors were computed using the code from the previous parts of question 2. The wavefunctions were the computed and plotted in 5. The wavefunctions were normalized by taking

$$\int_0^L |\psi(x)|^2 \mathrm{d}x \tag{1}$$

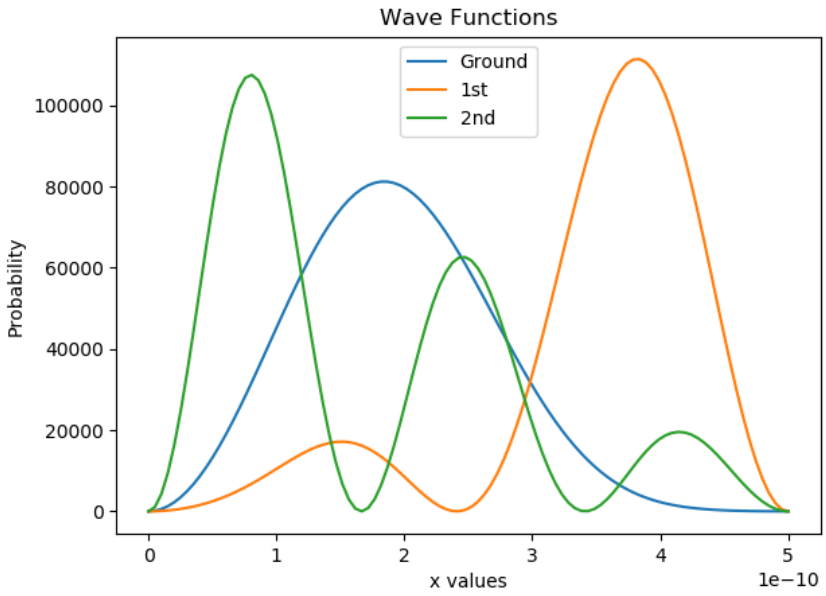

Figure 5: The first 3 wavefunctions plotted as a function of x.

Q3

a)

See figure 6.

b)

A comparison between the methods of relaxation and over-relaxation for solving the non-linear equation $x = 1 - e^{-cx}$.

The Function

```
def f(c,x):
    return 1 - np.exp(-c * x)
```

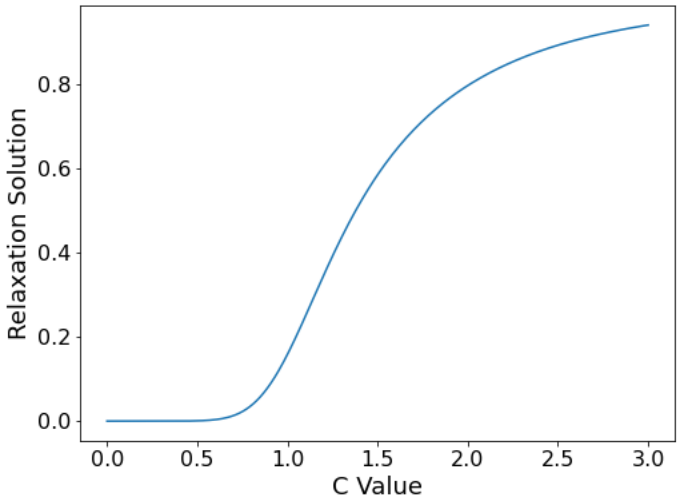

Figure 6: This plot shows solutions produced with the relaxation method to the equation $x = 1 - e^{-cx}$. The relaxation method for solving non-linear equations decrees that one iterates the equation $x' = f(x)$ until the solution converges to a desired level of accuracy.

The Relaxation Method

```
def relaxation_3(c,x0,steps):
    num_steps = 0
    for k in range(steps):
        if num_steps != 0 and np.round(x0,6) == np.round(f(c,x0),6):
            print('It took',num_steps,'steps to get a soln accurate to 10^(-6)')
            break
        else:
            num_steps += 1
            x0 = f(c,x0)
```

The Over-Relaxation Method

```
def over_relaxation(c, x0, w, steps):
    num_steps = 0
    for k in range(steps):
        if num_steps != 0 and np.round(x0,6) == np.round(f(c,x0),6):
            print('It took',num_steps,'steps to get a soln accurate to 10^(-6)')
            break
        else:
            num_steps += 1
            x0 = x0 + (1 + w) * (f(c,x0) - x0)
            print(np.round(x0,6))
```

To compare the relaxation and the over-relaxation programs, both programs were passed initial x-values of $x0 = 2$ and c-values of $c = 2$, corresponding to the non-linear equation $x = 1 - e^{-2x}$. The relaxation method took 14 iterations to reach a solution accurate to 10^{-6}. After trial and error optimization for the over-relaxation w-value, a w-value $w = 0.7$ caused over-relaxation method to take 6 iterations to reach a solution accurate to 10^{-6}. As predicted on page 260 from the text, the over-relaxation method can find a solution approximately twice as fast as the relaxation method. It should be noted that, also as predicted in the text, the over-relaxation method can be slower than the relaxation method with a poor choice of w.

For example, with a w-value $w = -0.2$ the over-relaxation method took 20 iterations to reach a solution accurate to 10^{-6}.

A negative w value would help to approximate a solution for diverging behaviour. This is because a negative w value takes the approximation to just a little bit off of where the program thinks the solution should be, not to just a little bit more towards where it should be. In this way it could approximate something that is not converging to a point, but to away from a point.

c)

6.13 b))

A comparison between the methods of relaxation, over-relaxation, binary, and Newton's for solving the non-linear equation $5e^{-x} + x - 5 = 0$.

The Relaxation Method

```
#define a function to be used for relaxation and over-relaxation
def g(x):
    return 5 - 5 * np.exp(-x)

#modify relaxation_3 so that it uses g(x)
def relaxation_4(x0,steps):
    num_steps = 0
    for k in range(steps):
        if num_steps != 0 and np.round(x0,6) == np.round(g(x0),6):
            print('It took relaxation',num_steps,'steps to get a soln accurate to 10^(-6):',
            np.round(x0,6))
            break
        else:
            num_steps += 1
            x0 = g(x0)
```

The Over-Relaxation Method

```
#modify over_relaxation so that it uses g(x)
def over_relaxation_2(x0, w, steps):
    num_steps = 0
    for k in range(steps):
        if num_steps != 0 and np.round(x0,6) == np.round(g(x0),6):
            print('It took over-relaxation',num_steps,'steps to get a soln accurate to 10^(-6):',
            np.round(x0,6))
            break
        else:
            num_steps += 1
            x0 = x0 + (1 + w) * (g(x0) - x0)
            #print(np.round(x0,6))
```

The Binary Method

```
#define a function to be used for binary
def h(x):
    return 5 * np.exp(-x) + x - 5

#define binary search method
def binary(x1,x2,accuracy):
    num_steps = 0 #define varible to count
    if h(x1) < 0 and h(x2) > 0:
```

Method	Number of Steps
Relaxation	6
Over-Relaxation	4
Binary	22
Newton's	4

Table 1: A table displaying the number of iterations that different methods for solving non-linear equations took to produce a solution accurate to 10^{-6} *for the non-linear equation* $5e^{-x} + x - 5 = 0$.

```
        while abs(x1 - x2) > accuracy:
            x_mid = 0.5 * (x1 + x2)
            f_at_mid = h(x_mid)
            if f_at_mid < 0:
                x1 = x_mid
                num_steps += 1
            else:
                x2 = x_mid
                num_steps += 1
        print('It took binary',num_steps,'steps to get a soln accurate to 10^(-6):',
        np.round(0.5 * (x1 + x2),6))
    elif h(x1) > 0 and h(x2) < 0:
        while abs(x1 - x2) > accuracy:
            x_mid = 0.5 * (x1 + x2)
            f_at_mid = h(x_mid)
            if f_at_mid > 0:
                x1 = x_mid
                num_steps += 1
            else:
                x2 = x_mid
                num_steps += 1
        print('It took binary',num_steps,'steps to get a soln accurate to 10^(-6):',
        np.round(0.5 * (x1 + x2),6))
    else:
        print('Initial points do not enclose a zero.')
```

Newton's Method

```
#note that newton uses the same root method as binary, so h(x) will work here. Define h prime
def h_prime(x):
    return - 5 * np.exp(-x) + 1

#define a function with both h_prime and h
def newton_h(x):
    return x - h(x) / h_prime(x)

#define a function for the newton method
def newton(x0,accuracy):
    num_steps = 0
    while abs(x0 - newton_h(x0)) > accuracy:
        x0 = newton_h(x0)
        num_steps += 1
    print('It took Newton',num_steps,'steps to get a soln accurate to 10^(-6):',
    np.round(newton_h(x0),6))
```

Thus it seems that the over-relaxation method and Newton's method are the most efficient methods of solving the non-linear equation $5e^{-x} + x - 5 = 0$. The relaxation method isn't far behind, producing a solution accurate to 10^{-6} in only two more iterations. The binary method is by far the least efficient at solving the non-linear equation $5e^{-x} + x - 5 = 0$, taking almost four times as many iterations as the fastest method.

6.13 c))

We may solve for the temperature using the following lines of code:

```
#set initial conditions
x = 4.965114
lam = 5.02 * 10**(-7)
kb = 1.380649 * 10**(-23)
h = 6.6260701 * 10**(-34)
c = 2.9979246 * 10**8

#calculate temperature
T = (h * c) / (kb * x * lam)
```

The above conditions produce an estimated surface temperature of the sun at 5772.45 Kelvin. This is remarkably good. The value listed on Wikipedia is 5772 Kelvin.

Code

Q1

```
import numpy as np
from numpy import empty,copy
import matplotlib.pyplot as plt

#Lab_4_Q1_c
#define constants
R1=R3=R5=1000 # resistance in ohms
R2=R4=R6=2000 # ohms
C1=1e-6 # capacitance in F
C2=0.5e-6 # F
x_p=3 #volts
omega=1000 # rad/s
L=R6/omega
#define complex arrays
v=np.array([x_p/R1,x_p/R2,x_p/R3], complex)
#[x1,x2,x3]
A=np.array([[1/R1+1/R4+1j*omega*C1, -1j*omega*C1, 0],
         [-1j*omega*C1, 1/R2+1/R5+(1j*omega*C1)+1j*omega*C2, -1j*omega*C2],
            [0, -1j*omega*C2, 1/R3+1/(R6)+1j*omega*C2]], complex)


##from part 1

N=len(v)

def partialpivot(A,v):
    #want it to find the row with the largest 1st/diagonal element and switch that row with that one
```

```
    #for first row, first element, compare first element of each row, biggest one switches with first r
    #then the gaussian happens again
    #then second row first element should be 0, so find largest diagonal element in each row and swap w
    for i in range(N): #!= means not equal
        max_index=A[i:,i].argmax()+i #finds index of biggest element in ith column
        #if that index isnt the index of the diagonal element in the row i, switch the rows
        if max_index !=i: #if not equal
            #row i, max row = #max row, row i
            A[i, :], A[max_index, :] = copy(A[max_index, :]), copy(A[i, :])
            #A[[i,max_index]] =  A[[max_index, i]]
            #switch rows for the rhs too
            #v[[i,max_index]] =  v[[max_index, i]]
            v[i], v[max_index] = copy(v[max_index]), copy(v[i])



# Gaussian elimination
for m in range(N):
# Divide by the diagonal element

    partialpivot(A,v)
    div = A[m,m] #sets div=to the diagonal element of the matrix
    A [m, : ] /= div #divides each element in the mth row by the diagonal element
    v[m] /= div  #divides the mth element of the right hand side by the diagnoal element also

    # Now subtract from the lower rows
    for i in range(m+1,N):
        mult = A [i ,m] #mth element of next row down from mth row (row,col)
        #subtract second row by mult*first row
        A[i,:] -= mult*A[m,:] #c-=a -> c=c-a #the next row from m is equal to taht row minus the first
        v[i] -= mult*v[m] #same as above but for rhs


# Backsubstitution
x = empty(N,complex)
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)


#%%

#calculate voltage amplitudes
t=0
V1=np.abs(x[0]*np.exp(1j*omega*t)) #Volts
V2=np.abs(x[1]*np.exp(1j*omega*t))
V3=np.abs(x[2]*np.exp(1j*omega*t))

ph1=np.degrees(np.arctan(x[0].imag/x[0].real))#degrees
ph2=np.degrees(np.arctan(x[1].imag/x[1].real))
ph3=np.degrees(np.arctan(x[2].imag/x[2].real))
```

```
print("Amplitudes", V1,V2,V3, "Phases",ph1,ph2,ph3)

#%%
t2=np.linspace(0,0.02,1000)
#[0.0001,0.0002,0.0003,0.0004,0.0005,0.0006,0.0007,0.0008,0.0009,0.001,0.0011,0.0012,0.0013,0.0014,0.00

Vr1=[0]*len(t2)
Vr2=[0]*len(t2)
Vr3=[0]*len(t2)

for i in range(len(t2)):
    Vr1[i]=x[0]*np.exp(1j*(omega)*t2[i])
    Vr2[i]=x[1]*np.exp(1j*(omega)*t2[i])
    Vr3[i]=x[2]*np.exp(1j*(omega)*t2[i])

plt.figure(1)
plt.title("Voltage vs. Time with Inductor")
plt.plot(t2, np.real(Vr1), label='V1')
plt.plot(t2, np.real(Vr2), label='V2')
plt.plot(t2, np.real(Vr3), label='V3')
plt.xlabel("Time (s)")
plt.ylabel("Real Voltage")
plt.xlim(0,0.02)
plt.legend()
```

Q2

```
import numpy as np
import matplotlib.pyplot as plt
from time import time
import astropy.constants as c
import astropy.units as u
from scipy.linalg import eigh

#6.9 b)

#define constants
hbar = 1.0545718 * 10**(-34) # in J * s
L = 5 * 10**(-10) # in m
M = 9.1094 * 10**(-31) # in kg
a = 1.602176634 * 10**(-18) # in J

#define a function that compute H_{mn} for a given m and n
#below is modelled from page 4 of lab handout
def H_mn(m,n):
    if m != n and m % 2 == 0 and n % 2 == 0:
        return 0
    elif m != n and m % 2 == 1 and n % 2 == 1:
        return 0
```

```
    elif m != n and m % 2 == 0 and n % 2 == 1 or m != n and m % 2 == 1 and n % 2 == 0:
        return - (8 * a * m * n) / (np.pi**2 * (m**2 - n**2)**2)
    elif m == n:
        return 0.5 * a + (np.pi**2 * hbar**2 * m**2) / (2 * M * L**2)

#6.9 c)

#modify H_mn to return an array of size nxn, with m,n up to n
def H_matrix(size):
    output = np.zeros((size,size))
    for i in range(1, size + 1):
        for j in range(1, size + 1):
            output[i-1, j-1] = H_mn(i,j) #using lab handout method
    return output

#compute H_mn for the 10x10 case
H_10_10 = H_matrix(10)

#compute eigenvalues and eigenvectors in KMS units? for 10x10 case
evals_H1010, evecs_H1010 = eigh(H_10_10)
evals_H1010=evals_H1010* 6.242e18

#6.9 d)
#compute H_mn for the 100x100 case
H_100_100 = H_matrix(100)

#compute eigenvalues and eigenvectors in KMS units? for 100x100 case
evals_H100100, evecs_H100100 = eigh(H_100_100)
#evals_H100100=evals_H100100#* 6.242e18
#evecs_H100100=evecs_H100100#* 6.242e18

#6.9 e

x=np.linspace(0, L, 100)
psi_0=[0]*100#np.zeros(100)
psi_1=np.zeros(100)
psi_2=np.zeros(100)
psi_n=np.zeros((100,100))

#%%

def psi_n(n, x):
    psi = 0
    for m in range(100):
        psi +=  evecs_H100100[n][m] * np.sin(np.pi * (m+1) * x / L)
    return psi



def psi0(x):
    return psi_n(0, x)

def psi1(x):
    return psi_n(1, x)
```

```
def psi2(x):
    return psi_n(2, x)


#%%
#normalizing
def psi0_2(x):
    return psi0(x)**2

def psi1_2(x):
    return psi1(x)**2
def psi2_2(x):
    return psi2(x)**2

N1=8
a=0
b=L

h1=(b-a)/N1

s_trap1 = 0.5*psi0_2(a) + 0.5*psi0_2(b)
s_trap2 = 0.5*psi1_2(a) + 0.5*psi1_2(b)
s_trap3 = 0.5*psi2_2(a) + 0.5*psi2_2(b)

for k in range(1,N1):
    s_trap1 += psi0_2(a+k*h1)
    s_trap2 += psi1_2(a+k*h1)
    s_trap3 += psi2_2(a+k*h1)

    I1_trap1=s_trap1*h1
    I1_trap2=s_trap2*h1
    I1_trap3=s_trap3*h1


#%% Plotting

plt.title("Wave Functions")
plt.plot(x, psi0_2(x)/(np.sqrt(I1_trap1)), label='Ground')
plt.plot(x, psi1_2(x)/(np.sqrt(I1_trap1)), label='1st')
plt.plot(x, psi2_2(x)/(np.sqrt(I1_trap1)), label='2nd')
plt.xlabel("x values")
plt.ylabel('Probability')
plt.legend()
```

Q3

```
#import modules
import numpy as np
import matplotlib.pyplot as plt

#pre-a) test an example from the book for understanding
```

```
#from page 251 of the text
def test(x):
    return 2 - np.exp(-x)

#define relaxation as per pg 251 of the text
def relaxation_test(x0,steps):
    for k in range(steps):
        x0 = test(x0)
    print(x0)

#test with in text example
relaxation_test(1,50)

#a) - 6.10 a)

#define function for 6.10
def f(c,x):
    return 1 - np.exp(-c * x)

#modify relaxation_test to relax with f(c,x)
def relaxation_1(c,x0,steps):
    for k in range(steps):
        x0 = f(c,x0)
        print(np.round(x0,6))

#run relaxation with printed values
relaxation_1(2,2,25)

#a) - 6.10 b)

#modify relaxation_1 to relax over a list of c values
def relaxation_2(c,x0,steps):
    solns = []
    for i in c:
        x = x0
        for k in range(steps):
            x = f(i,x)
        solns.append(x)
    return solns

#define the domain of c values
cs = np.arange(0,3.01,0.01)
#check solutions
solns = relaxation_2(cs,1,10)

#plot the relaxed solutions vs the cs
plt.figure(figsize=[8,6])
plt.plot(cs,solns)
plt.xticks(size=16)
plt.yticks(size=16)
plt.xlabel('C Value',size=18)
plt.ylabel('Relaxation Solution',size=18)
#plt.savefig('3')
```

```
#b) - 6.11 b)

#modify relaxation_1 so that it checks how many steps to get to the soln
def relaxation_3(c,x0,steps):
    num_steps = 0
    for k in range(steps):
        if num_steps != 0 and np.round(x0,6) == np.round(f(c,x0),6):
            print('It took',num_steps,'steps to get a soln accurate to
            10^(-6)')
            break
        else:
            num_steps += 1
            x0 = f(c,x0)

#run relaxation 3
relaxation_3(2,2,25)

#b) - 6.11 c)

#modify relaxation_3 for to employ overrelaxation
def over_relaxation(c, x0, w, steps):
    num_steps = 0
    for k in range(steps):
        if num_steps != 0 and np.round(x0,6) == np.round(f(c,x0),6):
            print('It took',num_steps,'steps to get a soln accurate to
            10^(-6)')
            break
        else:
            num_steps += 1
            x0 = x0 + (1 + w) * (f(c,x0) - x0)
            print(np.round(x0,6))

#run over_relaxation
over_relaxation(2,2,0.5,25)

#b) - 6.11 d)

#find answer in lab hand in sheet

#c) - 6.13 b)

#Relaxation

#define a function to be used for relaxation
def g(x):
    return 5 - 5 * np.exp(-x)

#modify relaxation_3 so that it uses g(x)
def relaxation_4(x0,steps):
    num_steps = 0
    for k in range(steps):
        if num_steps != 0 and np.round(x0,6) == np.round(g(x0),6):
            print('It took relaxation',num_steps,'steps to get a soln accurate to
            10^(-6):',np.round(x0,6))
```

```
            break
        else:
            num_steps += 1
            x0 = g(x0)

#modify over_relaxation so that it uses g(x)
def over_relaxation_2(x0, w, steps):
    num_steps = 0
    for k in range(steps):
        if num_steps != 0 and np.round(x0,6) == np.round(g(x0),6):
            print('It took over-relaxation',num_steps,'steps to get a soln accurate to
            10^(-6):',np.round(x0,6))
            break
        else:
            num_steps += 1
            x0 = x0 + (1 + w) * (g(x0) - x0)
            #print(np.round(x0,6))

#compute relaxation and over-relaxation methods
relaxation_4(2,50)
over_relaxation_2(2,0.035,50)

#Binary

#define a function to be used for binary
def h(x):
    return 5 * np.exp(-x) + x - 5

#define binary search method
def binary(x1,x2,accuracy):
    num_steps = 0 #define varible to count
    if h(x1) < 0 and h(x2) > 0:
        while abs(x1 - x2) > accuracy:
            x_mid = 0.5 * (x1 + x2)
            f_at_mid = h(x_mid)
            if f_at_mid < 0:
                x1 = x_mid
                num_steps += 1
            else:
                x2 = x_mid
                num_steps += 1
        print('It took binary',num_steps,'steps to get a soln accurate to
        10^(-6):',np.round(0.5 * (x1 + x2),6))
    elif h(x1) > 0 and h(x2) < 0:
        while abs(x1 - x2) > accuracy:
            x_mid = 0.5 * (x1 + x2)
            f_at_mid = h(x_mid)
            if f_at_mid > 0:
                x1 = x_mid
                num_steps += 1
            else:
                x2 = x_mid
                num_steps += 1
        print('It took binary',num_steps,'steps to get a soln accurate to
```

```
        10^(-6):',np.round(0.5 * (x1 + x2),6))
    else:
        print('Initial points do not enclose a zero.')

#run binary method
binary(2, 6, 10**(-6))

#Newton's Method

#note that newton uses the same root method as binary, so h(x) will work here.
#define h prime
def h_prime(x):
    return - 5 * np.exp(-x) + 1

#define a function with both h_prime and h
def newton_h(x):
    return x - h(x) / h_prime(x)

#define a function for the newton method
def newton(x0,accuracy):
    num_steps = 0
    while abs(x0 - newton_h(x0)) > accuracy:
        x0 = newton_h(x0)
        num_steps += 1
    print('It took Newton',num_steps,'steps to get a soln accurate to
    10^(-6):',np.round(newton_h(x0),6))

#run newton's method
newton(2,10**(-6))

#c) - 6.13 c)

#set initial conditions
x = 4.965114
lam = 5.02 * 10**(-7)
kb = 1.380649 * 10**(-23)
c = 2.9979246 * 10**8

#calculate temperature
T = (h * c) / (kb * x * lam)

#print temp
T
```