Lab Report 4

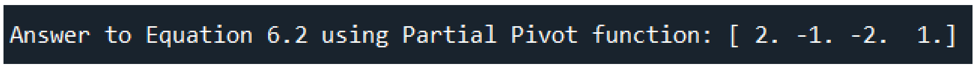
Seunghyun Park 1003105855

All questions and Lab report are done by Seunghyun Park

Questions

1.

(a)



```
Answer to Equation 6.2 using Partial Pivot function: [ 2. -1. -2.  1.]
```

Figure 1: The solution to Equation (6.2) using PartialPivot(A,v)

(b)

Pseudocode

```
import numpy
import matplotlib.pyplot
import time
from numpy.linalg import solve
from numpy.random import rand
from SolveLinear import GaussElim
from SolveLinear import PartialPivot

#Set N from 5 to a few hundred
NS = numpy.arange(5,300)
#Assign an empty list for time takes for Gaussian elimination
t_G = []
#Assign an empty list for time takes for Partialpivoting
t_P = []
#Assign an empty list for time takes for LU decomposition
t_L = []
#Assign an empty list for error of Gaussian elimination
error_G = []
```

```
#Assign an empty list for error of Partialpivoting
error_P = []
#Assign an empty list for error of LU decomposition
error_L = []

#Measures the time it takes to solve and the error for using each method
for N in NS:
        #Assign random values for elements of matrix
        v = rand(N)
        A = rand(N,N)

        t1 = Start time of Gaussian elimination
        x_G = GaussElim(A,v) #Solve the equation using Gaussian elimination
        t2 = End time of Gaussian elimination
        t_G.append(t2-t1) # Append the time takes for Gaussian elimination
        #Calculate and append the error of Gaussian elimination
        vsol_G = np.dot(A,x_G)
        error_G.append( np.mean(abs(v-vsol_G)))

        t3 = Start time of PartialPivoting
        x_P = PartialPivot(A,v) #Solve the equation using PartialPivoting
        t4 = End time of Gaussian elimination
        t_P.append(t4-t3) # Append the time takes for PartialPivoting
        #Calculate and append the error PartialPivoting
        vsol_P = np.dot(A,x_P)
        error_P.append( np.mean(abs(v-vsol_P)))

        t5 = Start time of LU decomposition
        x_L = solve(A,v) #Solve the equation using LU decomposition
        t6 = End time of Gaussian elimination
        t_L.append(t6-t5) # Append the time takes for LU decomposition
        #Calculate and append the error LU decomposition
        vsol_L = np.dot(A,x_L)
        error_L.append( np.mean(abs(v-vsol_L)))
```

```
#Plot the timings and errors in logarithmic scale
plt.loglog(timing of Gaussian elimination vs N)
plt.loglog(timing of PartialPivoting vs N)
plt.loglog(timing of LU decomposition vs N)
plt.loglog(error of Gaussian elimination vs N)
plt.loglog(error of PartialPivoting vs N)
plt.loglog(error of LU decomposition vs N)
```

Plots

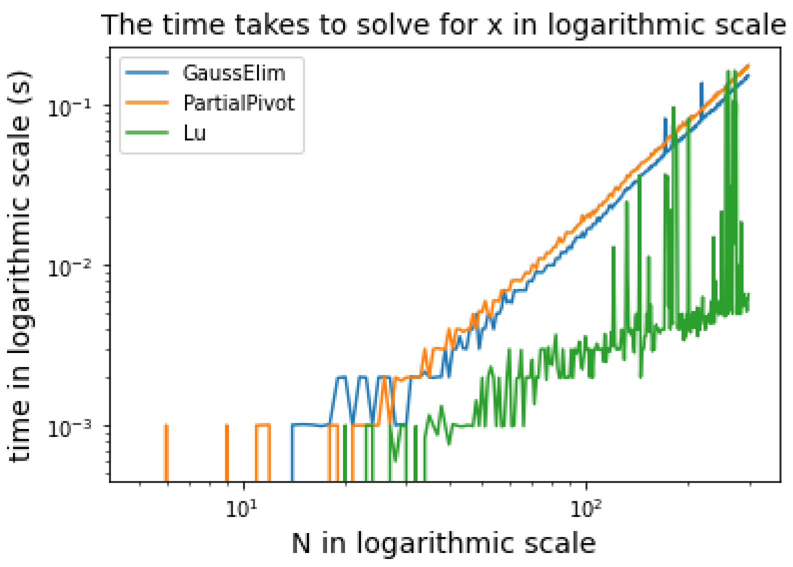


Figure 2: Timing of each method in logarithmic scale

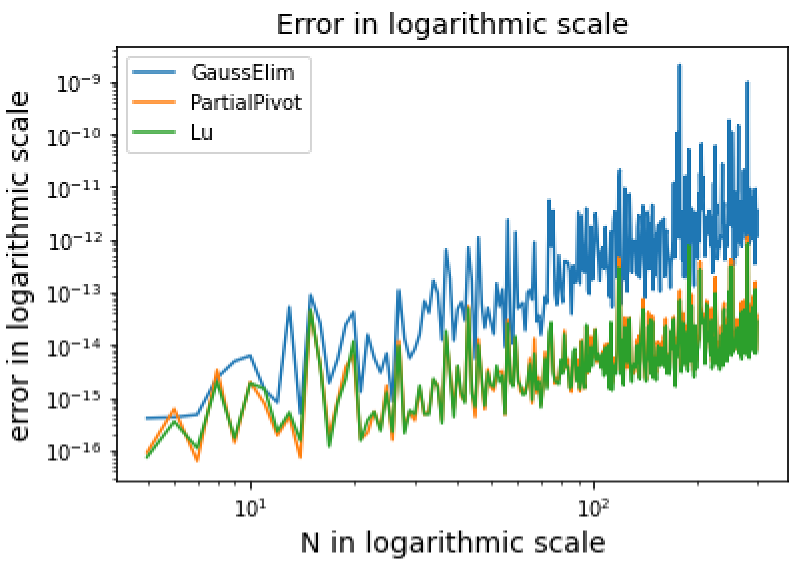


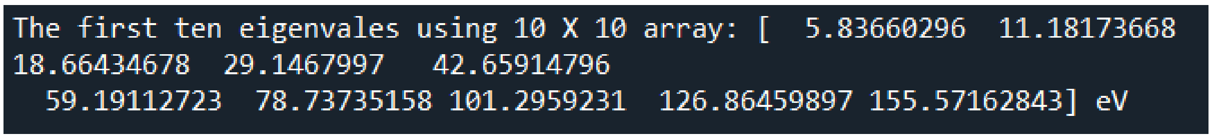
Figure 3: Error of each method in logarithmic scale

Written Answers

Figure 2 shows the time it takes to solve for x using each method. It is clear that LU decomposition is the fastest method. While the Partial Pivoting is just a modified version of the Gaussian elimination method, it is slower than the Gaussian elimination method because the Partial Pivoting method rearranges the rows at each stage. From figure 3, we can see that the error of the Gaussian elimination method is certainly larger than the error of other methods. It is because there are some special cases that Gaussian elimination does not work such as zeros on pivot position. Overall, the accuracy of LU decomposition method and Partial Pivoting method seem to be very similar while Gaussian elimination method is less accurate.

Question 2

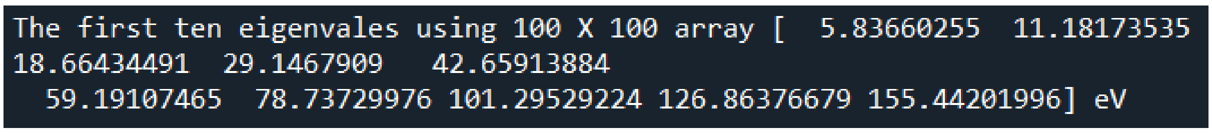
(c)



```
The first ten eigenvales using 10 X 10 array: [  5.83660296  11.18173668
18.66434678  29.1467997   42.65914796
  59.19112723  78.73735158 101.2959231  126.86459897 155.57162843] eV
```

Figure 4: The first ten eigenvalues using 10*10 array

(d)



```
The first ten eigenvales using 100 X 100 array [  5.83660255  11.18173535
18.66434491  29.1467909   42.65913884
  59.19107465  78.73729976 101.29529224 126.86376679 155.44201996] eV
```

Figure 5: The first ten eigenvalues using 100*100 array

Comparing the eigenvalues obtained in part (d) with the eigenvalues from part (c), we can see that the difference between them is very small in the lower energy. However, as the energy increases, the difference of the eigenvalues gets larger. This implies that to obtain an accurate result in a higher state of energy, a higher dimension of matrix is needed.

(e)

Pseudocode

```
import numpy as np
from numpy.linalg import eigvalsh
from numpy.linalg import eigh
from scipy import constants
import matplotlib.pyplot as plt
from gaussxw import gaussxw

#Set mmax and nmax = 100
mmax = nmax = 100

#Define wavefunction
def wavefunction(x,state,mmax):
    #Compute eigenvalue and eigenvalue using numpy.linalg eigh
    eigenvalue, eigenvector = eigh(H(mmax,mmax))
    #Assign initial value of the wavefunction
    psi = 0
    for i in range(mmax):
        psi += eigenvector[i,state]* $sin\frac{\pi (i+1)x}{L}$
        return psi

#Define $|\psi(x)|^2$
def wave2(x,state,mmax):
        return $|wavefunction(x, state, mmax)|^2$

lower = 0 # lower bound of integral
upper = L # upper bound of integral
N = 100 # number of sample
x, w = gaussxw(N)
xp = 0.5 * (upper - lower) * x + 0.5 * (upper + lower) # x value
wp = 0.5 * (upper - lower) * w # Weight on integral
s = 0.0  # Initial value of integral

#Calculate the integral $\int_0^L |\psi(x)|^2 dx$:
for i in range(N):
        s += wp[i] * wave2(xp[i],0,3)

print(The value of integral = A=  s)

#Normalize the integral
$1 = \frac{1}{A}\int_0^L |\psi(x)|^2 dx$
#Assign x value from 0 to L
x = numpy.linspace(0,L,100)

#Plot the probability density of ground state and first two excited state
plt.plot( ground state normalized probability density )
plt.plot( first excited state normalized probability density )
plt.plot( second excited state normalized probability density )
```

Plot

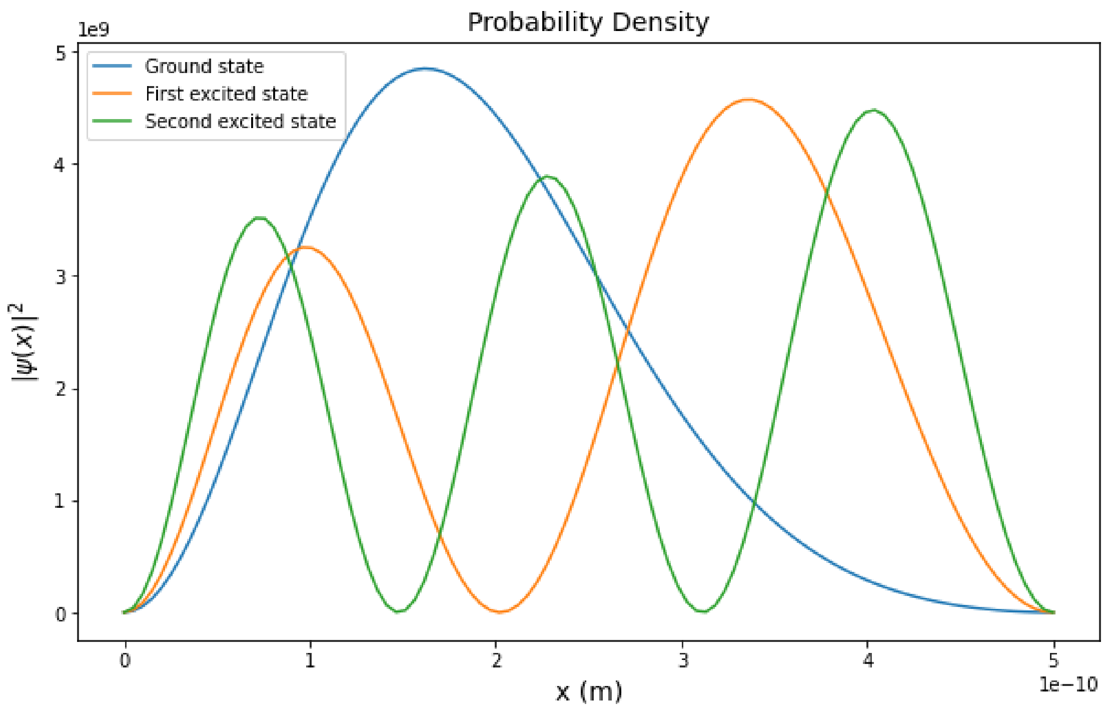


Figure 6: Probability density of ground state and the first two excited state

Question 3

(a)

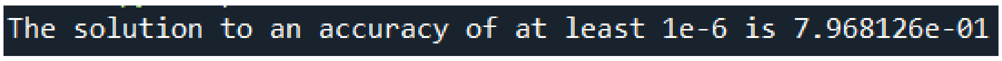
Pseudocode

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

def function $f(x) = 1 - e^{-cx}$:
    return $1 - e^{-cx}$

#Assign accuracy ϵ
accuracy = 1e-6
#Assign value of c
c = 2

#Find the solution of $f(x) = 1 - e^{-cx}$ using for loop:

for k in range(50):
    #Assign initial value of x
    x1 = 1.0
    #Assign initial value of error
    error = 1
    while error is greater than aimed accuracy:
        x1,x2 = f(x1),x1
        error = absolute value of ϵ', (ϵ' = $\frac{x_1-x_2}{1-1/f'(x_2)}$)
    #Print the solution of the equation
    print x1
    break

#Assign accuracy ϵ
accuracy = 1e-6
#Assign c value as cs
cs = numpy.arange(0,3,0.01)
#Assign empty list for x values
xs = []

#Calculate the solution for values of c from 0 to 3 in steps of 0.01 using for loop
for c in cs:
    #Assign initial value of x
    x1 = 1.0
```

```
        #Assign initial value of error
        error = 1
        while error is greater than aimed accuracy:
                x1,x2 = f(x1),x1
                error = absolute value of ϵ', (ϵ' = $\frac{x_1 - x_2}{1 - 1/f'(x_2)}$)
        #Append x1 values in xs list
        xs.append(x1)

#Print the solutions
print(xs)

#Plot x as a function of c
plt.plot(cs, xs)
```

Plots

Ex 6.10 part (a)



```
The solution to an accuracy of at least 1e-6 is 7.968126e-01
```

Figure 7: Solution of equation $x = 1 - e^{-cx}$ with c = 2

Ex 6.10 part (b)

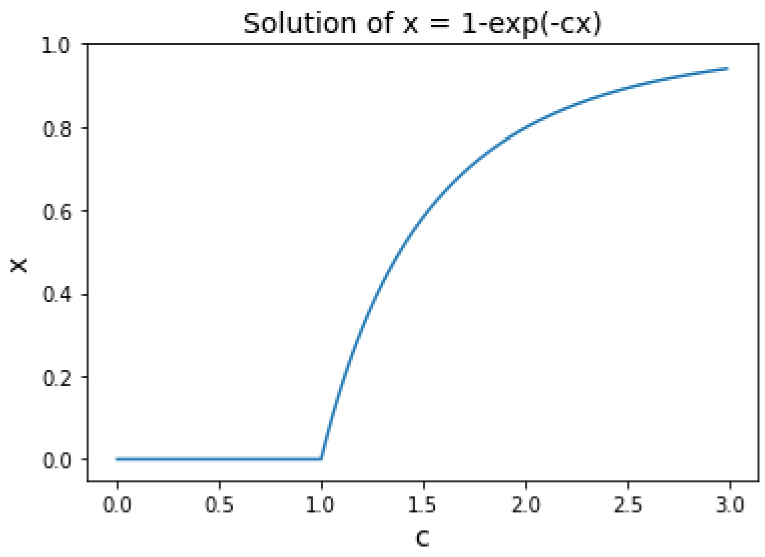


Figure 8 : Plot of $x = 1 - e^{-cx}$ as a function of c

(b)
Ex 6.11 part (b)

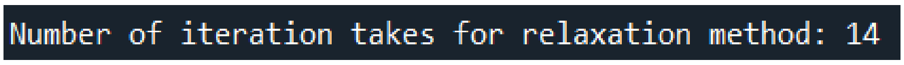
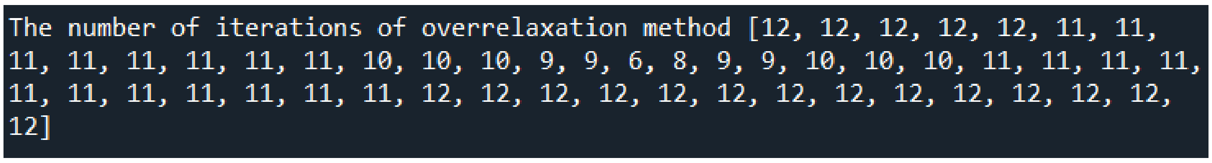

```
Number of iteration takes for relaxation method: 14
```

Figure 9: The number of iterations it takes to solve $x = 1 - e^{-cx}$ with relaxation method

Ex 6.11 part (c)

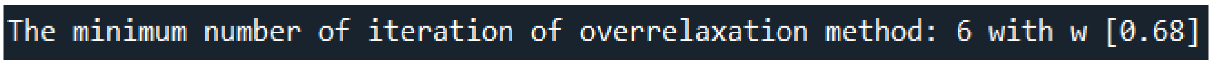
Pseudocode

```
import numpy as np

#Assign the value of ω from 0.5 to 1.
ws = arange(0.5,1)
#Assign accuracy ϵ
accuracy = 1e-6
#Create empty list for iteration
its = []
#Assign value of c
c = 2
def function f(x) = $1 - e^{-cx}$
    return $1 - e^{-cx}$
#Find the answers and iterations of $x = 1 - e^{-cx}$ with different value of ω using for loop
for ω in ws:
    Assign initial iteration value
    it = 0
    Assign initial value of x
    x1= 1.0
    Assign initial value of error
    while error is greater than accuracy:
        x1,x2 = f(x1),x1
        error = absolute value of ϵ', (ϵ' = $\frac{x_1 - x_2}{1 - 1/[(1+\omega)f'(x_2) - \omega]}$)
        it += 1
        #append iterations in list
        its.append(it)

#Find minimum iteration from the list
minimum iteration = numpy.where( its == numpy.amin(its))
#Find ω value that is corresponded to minimum iteration
minimum ω  = ws[index of minimum iteration]

print(The number of iteration of overrelaxation method)
print(The minimum number of iteration of overrelaxation method)
```



```
The number of iterations of overrelaxation method [12, 12, 12, 12, 12, 11, 11,
11, 11, 11, 11, 11, 11, 10, 10, 10, 9, 9, 6, 8, 9, 9, 10, 10, 10, 11, 11, 11, 11,
11, 11, 11, 11, 11, 11, 11, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12,
12]
```

Figure 10: The number of iterations of overrelaxation method
with ω = numpy.arange(0.5 , 1 , 0.1)



```
The minimum number of iteration of overrelaxation method: 6 with w [0.68]
```

Figure 11: The minimum number of iteration of overrelaxation method

The number of overrelaxation method iterations is less than the number of relaxation method iterations. The minimum number of overrelaxation method iteration occurs when ω = 0.68 and has a value of six, which is less than half of relaxation method iterations.

Ex 6.11 part (d)

The overrelaxation method is the same as the ordinary relaxation method when ω = 0, but for ω > 0 the method changes x' by ω *Δx more. If we set the estimated initial x value to be larger than the solution of the system and take ω as a negative, it could be faster than the ordinary relaxation method.

(c)

Pseudocode

```
import numpy as np
from scipy import constants

# Assign values of constants
h = Planck's Constant in $m^2 kg/s$
c = speed of light in $m/s$
kb = Boltzmann's Constant $J/K$

# Define f(x)= $5e^{-x} + x - 5$
def f(x):
        return $5e^{-x} + x - 5$

def binary search method(x1,x2):
        # x1,x2  = initial pair of points
        #Assign accuracy ϵ
        accuracy = 1e-6
        #Assign initial error
        error = 1
        signs of $f(x_1)$
        signs of $f(x_1)$
        if $f(x_1)$ and  $f(x_2)$ have opposite signs:
                while error is bigger than aimed accuracy:
                        midpoint $x'$ = $\frac{1}{2}(x_1 + x_2)$
                        evaluate f(x')
                        if $f(x')$  has same sign as $f(x_1)$:
                                $x_1$ = midpoint $x'$
                        else:
```

```
                x2 = midpoint $x'$
            error = absolute value of $(x_2 - x_1)$

    **else** (in case of $f(x_1)$ and $f(x_2)$ does not have opposite signs):
        print "Signs of $f(x_1)$ and $f(x_2)$ are same"
    **return** midpoint $x'$


#Find the solution of given function f(x)= $5e^{-x} + x - 5$
x = binary search method(x1,x2)

#Calculate the Wien displacement constant
Wien displacement constant = Plank's constant * speed of light / (boltzmann's constant * x)

#Calculate the surface temperature of the Sun
peak wavelength = 502e-9
temperature = Wien displacement constant / peak wavelength

**print** solution of the given function f(x)
**print** Wien displacement constant
**print** temperature of the Sun
```

Written answers

Ex 6.13 part (b)

The solution of given function f(x)= $5e^{-x} + x - 5$ calculated with the binary search method is 4.965114 and the value of the Wein displacement constant is 2.897772e-03 (m•K).
Compared to the theoretical Wein displacement constant value of 2.898e-3 (m•K), we can see the calculated value with the binary search method is very accurate.

Ex 6.13 part (c)

The estimated surface temperature of the Sun is 5772.455 K with the wavelength peak = 502nm.