PHY407, Fall 2022
Oct 5 Wednesday

Zachary Folan

Question 1

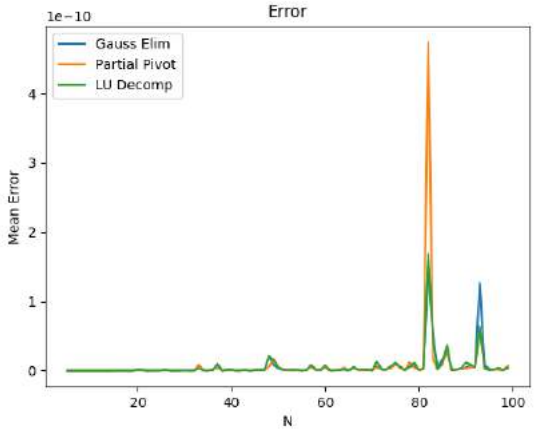
Part a)

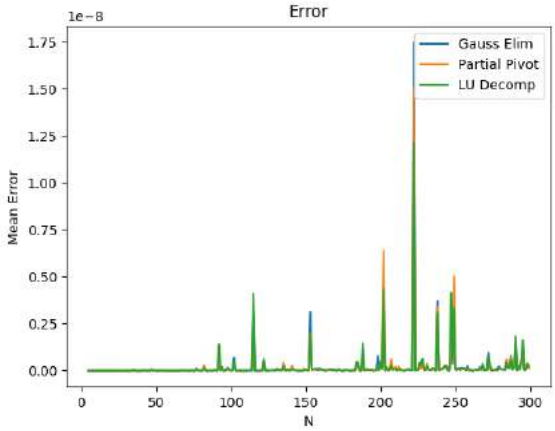
PartialPivot was completed and the output created is displayed below.

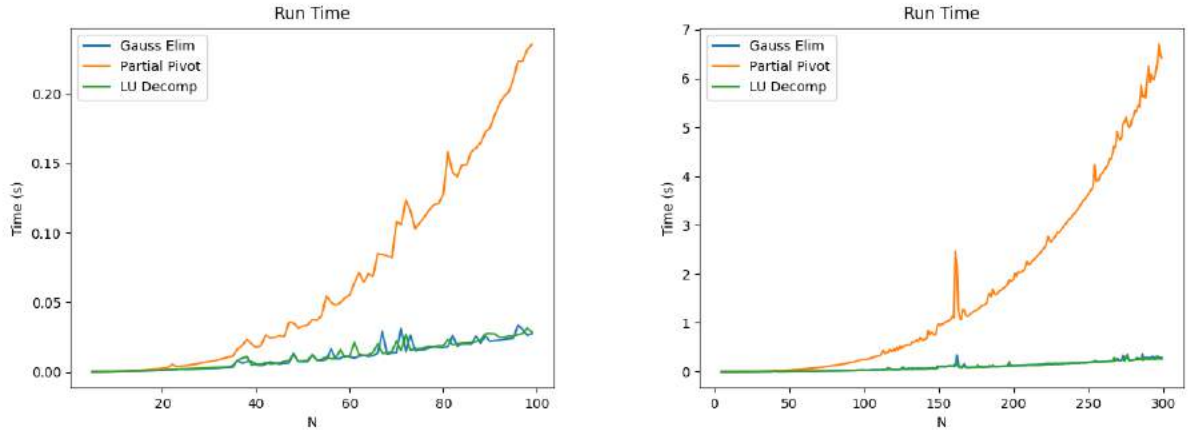
```
Input array, A:
 [[ 2.  1.  4.  1.]
 [ 3.  4. -1. -1.]
 [ 1. -4.  1.  5.]
 [ 2. -2.  1.  3.]]
v:
 [-4.  3.  9.  7.]
Gauss Elim Solution:
 [ 2. -1. -2.  1.]
Partial Pivot Solution:
 [ 2. -1. -2.  1.]
```

Part b)

PartialPivot, GaussElim and LU Decomposition were all tested for values of $N \in [5, 300]$ and $N \in [5, 100]$ to better display variances in all methods. The run time and error were recored and plotted for each value of N used.







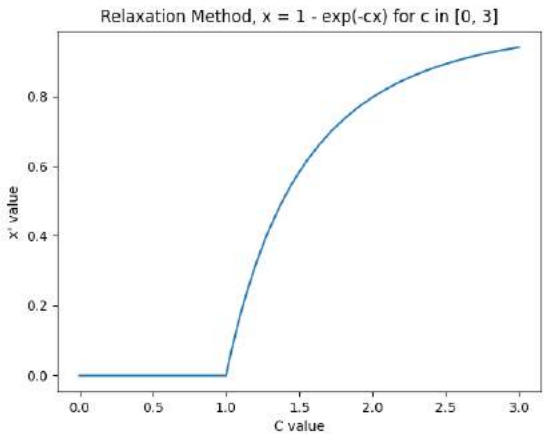
Displayed in the graphs above we can see that the mean error of all methods have no significant differences. The runtime of GaussElim and LU Decomposition are similar and have the same runtime complexity in practice; however, we can see that PartialPivot has a far greater asymptotic upper bound producing exponentially larger runtimes as N increases.

All code for Q1 is located in solvelinear.py

Question 3

Part a)

The relaxation method was used to find a solution to the given function for varying values of c. The value found was 0.796813

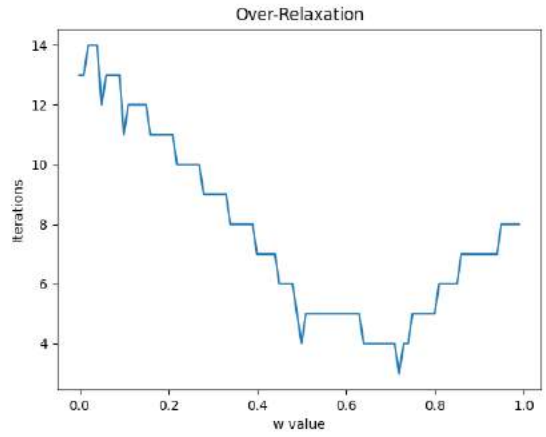



```python
import numpy as np
from numpy import exp
import matplotlib.pyplot as plt


def relaxation_method(rhs, initial_val, iters, c):
    x = initial_val
    x1 = 0
    for i in range(iters):
        x1 = rhs(x, c)
        if round(x1, 6) == round(x, 6):
            return round(x1, 6)
        x = x1
    return 0


def func(x, c):
    return 1 - exp(-c*x)


a = relaxation_method(func, 1, 50, 2)
print(a)

x = [c/100 for c in range(301)]
vals = [relaxation_method(func, 1, 1000, c) for c in x]

plt.plot(x, vals)
plt.title("Relaxation Method, x = 1 - exp(-cx) for c in [0, 3]")
plt.xlabel("C value")
plt.ylabel("x' value")
plt.show()
```

Part b)

b)
13 iterations were required to find a solution within $10e^{-6}$ which is 0.796813.

c)
Result: 0.796813 , Iterations: 13
Result: 0.796813 , Iterations: 11
Result: 0.796812 , Iterations: 11
Result: 0.796812 , Iterations: 9
Result: 0.796812 , Iterations: 7
Result: 0.796812 , Iterations: 4
Result: 0.796812 , Iterations: 5
Result: 0.796812 , Iterations: 4
Result: 0.796812 , Iterations: 5
Result: 0.796812 , Iterations: 7




```python
import numpy as np
from numpy import exp
import matplotlib.pyplot as plt


def relaxation_method(rhs, initial_val, iters, c):
    x = initial_val
    x1 = 0
    for i in range(iters):
        x1 = rhs(x, c)
        if round(x1, 6) == round(x, 6):
            return round(x1, 6), i
        x = x1
    return 0


def over_relaxation_method(rhs, initial_val, iters, c, w):
    x = initial_val
    x1 = 0
    for i in range(iters):
        x1 = (1 + w) * rhs(x, c) - w * x
        if round(x1, 6) == round(x, 6):
            print("Result: ", round(x1, 6), ", Iterations: ", i, "\\\\")
            return round(x1, 6), i
        x = x1
    return 0


def func(x, c):
    return 1 - exp(-c*x)
```

Part b and Part c Comparison)
In part b we can see that it took 13 iterations to find a solution where as in part c we can see on the graph that using the over-relaxation method we achieved the solution in 3 iterations for an ω value of roughly $\omega = 0.75$ taking less than a quarter of the iterations required with the basic relaxation method.

d)
It would be adventatgeous to use an $\omega < 0$ in cases where our value of $|x' - x|$ is too large resulting in too large of steps when using the relaxation method to find a solution.

Part c)

b)
The result achieved by the binary search method was 4.965114. The real solution is 4.965 which confirms our solution acheived by the binary search method is correct.

c)
Using $\lambda = 502$ and our achieved value of $x = 4.965$ then pluging that into the equation

$$x = \frac{hc}{\lambda k_B T}$$

we see that T = 5772.454227781519 Kelvin. This shows that in our case the temperature of the surface of the sun is 5772 K.

```python
def is_same_sign(x1, x2):
    return  x1 * x2 > 0


def binary_search(func, x1, x2):
    if is_same_sign(func(x1), func(x2)):
        return 0, 0

    i = 0
    while (x2 - x1) / 2.0 > 10**(-6):
        midpoint = (x1 + x2) / 2.0
        print(midpoint)
        if func(x1) * func(midpoint) < 0:
            x2 = midpoint
        else:
            x1 = midpoint

        i += 1

    return midpoint, i


def part_c_function(x):
    return 5*np.exp(-x) + x - 5


result, iterations = binary_search(part_c_function, 2, 8)
print(result, iterations)


def part_c(x):
    h = 6.62607015 * (10 ** (-34))
    c = 299792458
    k = 1.380649 * (10 ** (-23))
    l = 502 * (10 ** (-9))

    t = (h * c) / (k * l * x)

    print(t)
```

Draft version October 9, 2022

PHY407 Lab-02 Report

Raina Irons[1] and Zachary Folan[2]

[1] *Wrote and solved Q2*
[2] *Wrote and solved Q1, Q3*

QUESTION 2 - ASYMMETRIC QUANTUM WELL

PART C

We will utilize our H matrix we created in part b (not shown) in order to calculate the eigenvalues of it for a 10x10 matrix. The output statements of the eigenvalues are as follows.

$$5.8363688, 11.18106982, 18.66283939, 29.14410444, 42.65492877,$$
$$59.18504732, 78.72907359, 101.28510945, 126.85091187, 155.55474429$$

Which were calculated using the *numpy.linalg.eigvalsh*() function. We can check the validity of this value by noting that the ground state energy is around 5.84eV, which is what we get in the first entry.

PART D

The onyl difference between this section and Part C is changing the size of our H matrix now to be a 100x100 matrix. I will include the first few and last few values of our eigenvalues, the complete set will be in the *.py* file.

$$5.83636839 \tag{1}$$
$$11.1810685 \tag{2}$$
$$18.6628375 \tag{3}$$
$$29.1440956 \tag{4}$$
$$42.6549196 \tag{5}$$
$$59.1849947 \tag{6}$$
$$78.7290218 \tag{7}$$
$$101.284479 \tag{8}$$
$$126.850080 \tag{9}$$
$$155.425121 \tag{10}$$
$$187.009195 \tag{11}$$
$$221.602049 \tag{12}$$
$$... \tag{13}$$
$$1.19190935e+04 \tag{14}$$
$$1.21883300e+04 \tag{15}$$
$$1.24605748e+04 \tag{16}$$
$$1.27358278e+04 \tag{17}$$
$$1.30140891e+04 \tag{18}$$
$$1.32953585e+04 \tag{19}$$
$$1.35796362e+04 \tag{20}$$
$$1.38669221e+04 \tag{21}$$
$$1.41572163e+04 \tag{22}$$
$$1.44505187e+04 \tag{23}$$
$$1.47468294e+04 \tag{24}$$
$$1.50461618e+04 \tag{25}$$

Where we see the terms in Part C increase slightly faster than in Part D.

PART E

Using the *numpy.linalg.eigh*() function, we can find the eigenvectors of the H matrix for size 100x100 and use them to find the wave function. For example, we can find the ground state wave function through

```
for x in domain:
    tmp = 0
    for n in range(1,len(V0)-1):
        tmp += V0[n-1]*np.sin((n*np.pi*x)/L) # follows the summation equation
    psi0.append(tmp)
psi0 = np.array(psi0) # wave function
psi0_prob = np.abs(psi0)**2 # probability density
```

in which the domain is the well space, $V0$ is the ground state eigenvector, and L is the length of the well. We can normalize *psi_prob* by integrating over

$$A = \int_0^L |\Psi(x)|^2 dx \tag{26}$$

and dividing the whole wave function by $\sqrt{A}$. Doing this procedure for the ground and first two excited states gives us:

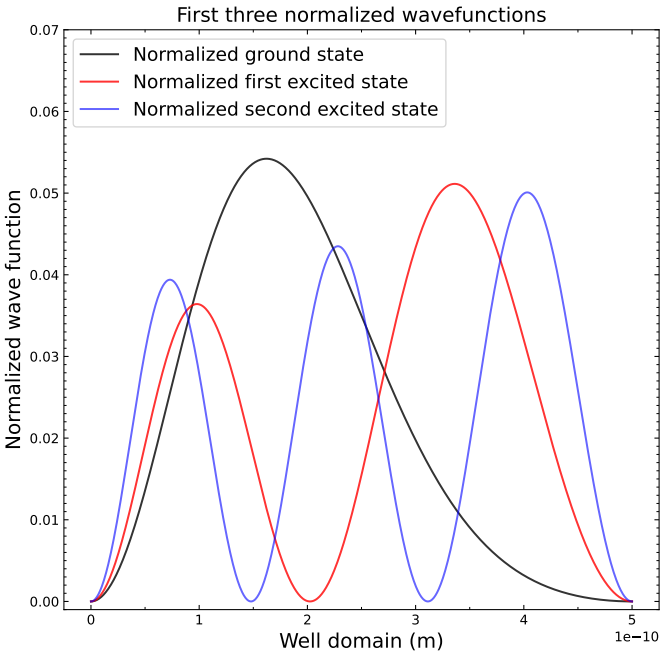


Figure 1: The above demonstrates the first three normalized states of the wave function for the asymmetric well for a 100x100 H matrix.