AMS 326 Numerical Analysis HW 2

By: Mehadi Chowdhury (115112722)

Link to GitHub where README, PDF, IPYNB and Seperate Python Files will be hosted: https://github.com/EmceeCiao/AMS_326_HW2

Background For Problem 2.1

Problem 2.1 (50 Points) The "kidney" equation $(x^2+y^2)^2 = x^3+y^3$ (red curve) can be graphed as

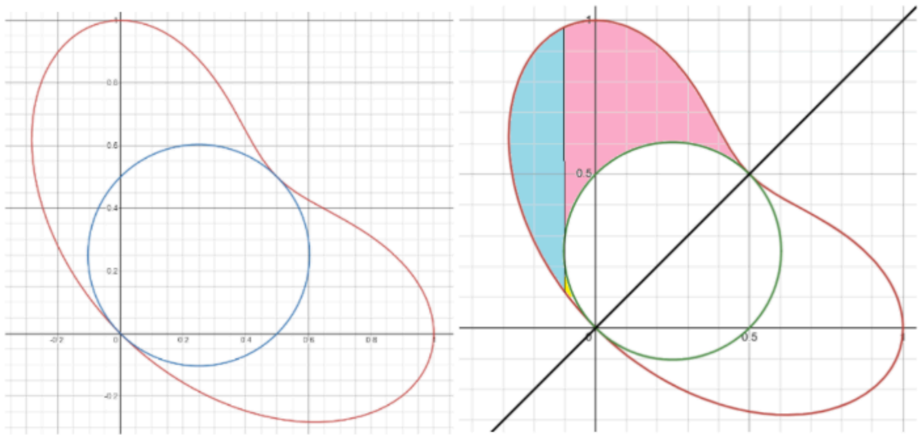


Figure 1. Two illustrations to assist your understanding of the problem. The right one, made by Elan S-K and posted with permission, does not suggest a particular method you may use.

Dig a disc from the kidney. The disc equation is $(x-0.25)^2+(y-0.25)^2 = 0.125$ (blue).

(1) Write a program to use the rectangle method to compute the area of the remaining kidney (4 significant digits).
(2) Write a computer program to use the trapezoidal method to compute the area of the remaining kidney (4 significant digits).

Problem 2.1 - Task 1

Problem Description

Given a kidney equation of $(x^2+y^2)^2 = x^3+y^3$ and digging a disc from the kidney with equation $(x-0.25)^2+(y-0.25)^2 = 0.125$
Write a program to use the rectangle method to compute the area of the remaining kidney to 4 siginficant digits.

Algorithm Description

There are three rectangle methods that can be used, but the most accurate one is the midpoint method, which is the one we will be using here! The midpoint method approximates the area underneath the curve by having us evaluate the function value at the midpoint of every interval and take the sum. The exact equation of the formula is as follows:

$\int_a^b f(x)dx \approx \Delta x[f(x_1) + f(x_2)+.....+f(x_n)]$

In this specific case however, we have to do a bit more thinking. Considering a disk is being cut out of the kidney and that the disk is in the equation form of a circle, we can easily compute it's area as a constant and subtract from the approximation we get using the rectangle method on the kidney.

To do this method on the kidney, we need a clear way to calculate a y-value, in this case this will require us to convert the equation into polar like so:

$(x^2 + y^2)^2 = x^3 + y^3 \rightarrow r^4 = r^3(sin^3\theta + cos^3\theta) \rightarrow r = sin^3\theta + cos^3\theta$

With this we can actually perform the rectangle method as we can treat r as our y-value and the bounds for theta or what is essentially x in this case is from 0 to 2 π. This has to be slightly adapted though as we are now in the polar plane, each "rectangle" instead of being f(x) * Δx it's now $1/2 * r^2 * \Delta\theta$ as it's a circular cut.

This is derived from the fact that a normal circle is πr^2 which when only taking a fraction of the circle we get $(\theta/(2\pi)) * (\pi r^2)$

Thus simplifying to what we had stated prior!

Now that we have our formula, all that's left to decide is the number of intervals we want for our accuracy, in our case I believe 10,000 intervals will be enough to get me close to an approximation within 4 significant digits.

Psuedocode

Rectangle Method

```
disk_area ← radius^2 * pi
kidney_eq ← cos^3(x) + sin^3(x)
intervals ← 10,000

interval_vals ← [0, 2pi/intervals, 2* 2pi/intervals, .....]
$\Delta \leftarrow 2pi/intervals$

from i in range (0, intervals)
    midpoint ← (interval_vals[i] + interval_vals[i+1])/2
    area += (kidney_eq(midpoint))^2 * $\Delta$ * 0.5

return area - disk_area
```

Code

```
In [ ]: import numpy as np

        def disk_area(r_squared):
            # We are given r^2 in the formuala for the disk so let the user pass it in
            return r_squared * np.pi

        def kidney_equation(x):
            # Equation of Kidney after translating to Polar, x is the angle plugged in!
            return np.cos(x) ** 3 + np.sin(x) ** 3

        def rectangle_method_polar(kidney_eq, intervals, dug_area):
            """
            Function For Midpoint Method on Polar Equation with cut out area!

            Input:
            - kidney_eq: equation for the provided kidney
                         (Could be generalized to function of curve)
            - intervals: Whole number of intervals to be used in calculating
            - dug_area: Area of disk that's been cut out of the kidney

            Output:
            - result: The approximation of the area under the curve after subtracting out
                      the dug area using rectangle method
            """
            interval_vals = np.linspace(0, 2*np.pi, intervals+1)
            step = (2*np.pi)/intervals
            area = 0
            for i in range(len(interval_vals)-1):
                midpoint = (interval_vals[i] + interval_vals[i+1])/2
                area += 0.5 * step * (kidney_eq(midpoint) ** 2)
            return area-dug_area

        dug_area = disk_area(0.125)

        ans = rectangle_method_polar(kidney_equation, 10000, dug_area)
        print(f"Approximated Area: {ans}")
```

```
Approximated Area: 1.5707963267949003
```

Results

The results of my program were the following:

Area of Kidney with Cut Disk $\approx$ 1.570796

Performance

In terms of the performance of this rectangular method polar, the time complexity would be O(n) where n represents the number of intervals we decided to use as that's the number of operations performed. The space complexity of the program is also O(n) as we have to store all of these intervals. However, computationally, this is easier than computing the integral for the area and serves as a good estimate.

Problem 2.1 - Task 2

Problem Description

Given a kidney equation of $(x^2 + y^2)^2 = x^3 + y^3$ and digging a disc from the kidney with equation $(x - 0.25)^2 + (y - 0.25)^2 = 0.125$
Write a program to use the trapezoidal method to compute the area of the remaining kidney to 4 siginficant digits.

Algorithm Description

The trapezoidal method approximates the area underneath the curve by having us evaluate the function value at the left and right endpoints of every interval, divide it by 2 and multiply it by the step as the formula for a trapezoid is $(a + b)/2 * h$. After summing all these values together we get the approximate area under the curve.

The exact equation of the formula is as follows:

$$\int_a^b f(x)dx \approx \Delta x/2[f(x_0) + 2f(x_2) + 2f(x_3) + \ldots.. + f(x_n)]$$

In this specific case however, we have to do a bit more thinking just like for task 1! We will have to cut out the constant area of the disk which can be calculated easily as it's in the form of a circle.

Again to use the trapezoidal method on the kidney, we need a clear way to calculate a y-value, in this case this will require us to convert the equation into polar like so:

$$(x^2 + y^2)^2 = x^3 + y^3 \rightarrow r^4 = r^3(sin^3\theta + cos^3\theta) \rightarrow r = sin^3\theta + cos^3\theta$$

With this we can perform our trapezoidal method of integration, again with the caveat that we will have to do r^2 due to us working in polar coordinates and once again for accuracy I will be using 10,000 intervals.

Psuedocode

Trapezoidal Method

```
disk_area ⟵ radius^2 * pi
kidney_eq ⟵ cos^3(x) + sin^3(x)
intervals ⟵ 10,000

interval_vals ⟵ [0, 2pi/intervals, 2* 2pi/intervals, .....]
$\Delta \leftarrow 2pi/intervals$
```

```
from i in range (0, intervals)
    left ⟵ (kidney_eq(interval_vals[i]))^2
    right ⟵ (kidney_eq(interval_vals[i+1]))^2
    area += ((left + right)/2) * $\Delta$ * 0.5

return area - disk_area
```

Code

In [ ]:
```python
import numpy as np

def disk_area(r_squared):
    # We are given r^2 in the formuala for the disk so let the user pass it in
    return r_squared * np.pi

def kidney_equation(x):
    # Equation of Kidney after translating to Polar, x is the angle plugged in!
    return np.cos(x) ** 3 + np.sin(x) ** 3

def trapezoidal_method_polar(kidney_eq, intervals, dug_area):
    """
    Function For Trapezoidal Method on Polar Equation with cut out area!

    Input:
    - kidney_eq: equation for the provided kidney
                 (Could be generalized to function of curve)
    - intervals: Whole number of intervals to be used in calculating
    - dug_area: Area of disk that's been cut out of the kidney

    Output:
    - result: The approximation of the area under the curve after subtracting
              out the dug area using trapezoidal method
    """
    interval_vals = np.linspace(0, 2*np.pi, intervals+1)
    step = (2*np.pi)/intervals
    area = 0
    for i in range(len(interval_vals)-1):
        left_value = (kidney_eq(interval_vals[i])) ** 2
        right_value = (kidney_eq(interval_vals[i+1])) **2
        area += 0.5 * step * ((left_value + right_value)/2)
    return area-dug_area

dug_area = disk_area(0.125)

ans = trapezoidal_method_polar(kidney_equation, 10000, dug_area)
print(f"Approximated Area: {ans}")
```

```
Approximated Area: 1.5707963267949026
```

Results

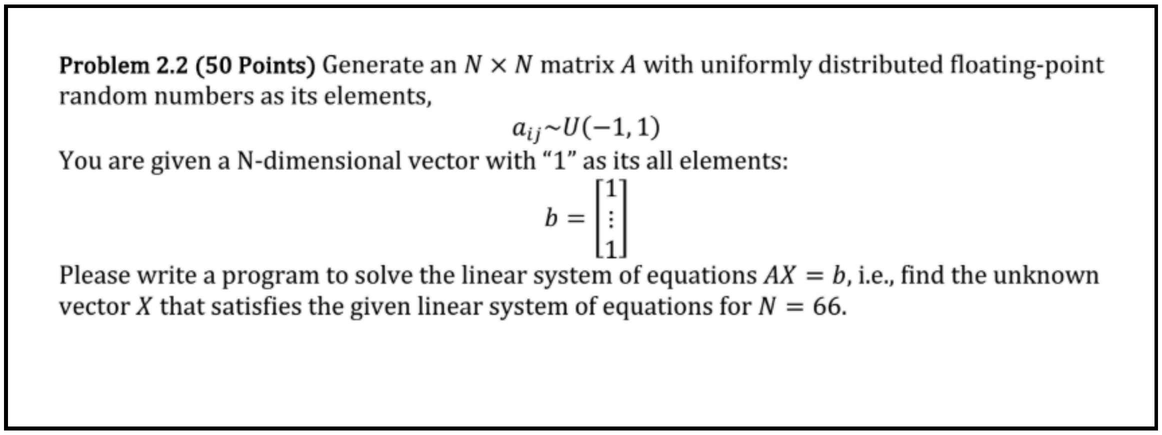
The results of my program were the following:

Area of Kidney with Cut Disk ≈ 1.570796

Performance

In terms of the performance of this trapezoidal method, the time complexity would be O(n) where n represents the number of intervals we decided to use as that's the number of computations performed. The space complexity of the program is also O(n) as we have to store all of these intervals. There is not much else to comment on this program's performance besides it being a good way to approximate the area under a curve as computing integrals is computationally expensive.

Background For Problem 2.2



Problem 2.2 (50 Points) Generate an $N \times N$ matrix A with uniformly distributed floating-point random numbers as its elements,

$$a_{ij} \sim U(-1, 1)$$

You are given a N-dimensional vector with "1" as its all elements:

$$b = \begin{bmatrix} 1 \\ \vdots \\ 1 \end{bmatrix}$$

Please write a program to solve the linear system of equations $AX = b$, i.e., find the unknown vector X that satisfies the given linear system of equations for $N = 66$.

Problem 2.2

Problem Description

Given some N x N matrix called A filled with uniformly distributed floating-point random numbers,
and a vector, b, of size n with 1 as all of it's elements, write a program to solve AX = b, if N = 66.

Algorithm Description

The algorithm I used to solve Ax=b in this case was gaussian-elimination. To explain the algorithm, it consists of two parts. Forward Elimination and Backward Substitution and this is all done on the augmented matrix A|b. The essential idea behind Forward Elimination is that we want to create an upper triangular matrix, meaning all the entries below the diagonal are 0.

To accomplish this we need a pivot row for each diagonal entry that will be scaled and subtracted from the other rows to 0 out the values below that entry. To find this row, I took the row with the maximum absolute value for that diagonal entry and swapped it with the row that was previously there. After doing this for every diagonal entry, the matrix is now in the form of an upper triangular matrix and Back Substitution can now be done.

Back Substituion starts in the last row, as we can easily solve for the x value in that row because that was our last diagonal entry, meaning no other variables are in the equation represented by that row. With this known value we can now go one up from the last row and solve. This can happen because in an upper triangular matrix, the rows above can only contain variables/unknowns from the rows below besides the unknown
already in that row. So, if we start solving repeatedly from the bottom, we can solve each row one at a time as they're will only be one unknown each time. Eventually, we will have solved for all the unknowns in our system of equations, providing us the values for vector x which is the solution to our problem of Ax=b.

To set up this augmented matrix using numpy, I found numpy.random.uniform() to be useful in generating the N x N matrix of uniformly distributed random floating point numbers, np.ones() to generate b, and np.hstack(), to create A|b. For the sake of validation numpy.linalg.solve() was used on A and b and the result was printed alongside the result I got to double check the validity of the gaussian-elimination I coded.

Psuedocode

Gaussian Elimination

```
N ⟵ 66
A ⟵ N x N Matrix of Floating Point Values between -1 and 1
b ⟵ N x 1 Matrix of 1's
Ab ⟵ A|b

for i in range (0, N):
    max_row ⟵ row with absolute max value found
    Ab[ max_row ] = Ab[ i ], Ab[ i ] = Ab[ max_row ]
    for row in range (i+1, N):
        Ab[ row ] - scalar * Ab[ i ]
x ⟵ [0, 0, 0, ...., 0]
for i in range (N-1, -1, -1):
    x[i] = Ab[i, -1] - (Ab[i+1 : N] * x[i+1 : N]) # Subtracting values to solve for x

return x
```

Code

In [ ]:
```
import numpy as np

def gaussian_elimination(N):
    """
    Function to perform gaussian elimination on Ax=b, where A is N x N and
    b is N x 1 and is all ones.

    Inputs:
    - N: Size of N * N matrix to be randomly generated
    Outputs:
    - x: A vector of size N that is the solution to Ax = b

    """
    A = np.random.uniform(low=-1, high=1, size=(N, N))
    b = np.ones((N,1))
    Ab = np.hstack((A,b)) # Concatenates A and b

    for i in range(N):
        max_val = 0
        max_row = i
        for row in range(i, N):
            # We only care to change this if it's a higher absolute max value
            if(abs(Ab[row][i])) > max_val:
                max_val = abs(Ab[row][i])
                max_row = row
        if i != max_row:
            # Numpy Row Swapping Below!
            Ab[[i, max_row]] = Ab[[max_row, i]]

        # Now we can actually start eliminating below our max row
        # which starts at i+1 and goes to N
        for row in range(i+1, N):
            factor = Ab[row, i]/Ab[i, i]
            Ab[row, i:] -= factor * Ab[i, i:]

    # Now Ab is in upper triangular form, so we backsubstitute to
    # solve our system of equations

    x = np.zeros(N) # initalize x to 0's
    for i in range(N-1, -1, -1):  # N - 1 is our last row!
        # Sum of all other known-values plugged in
        non_target_x_vals = np.sum(Ab[i, i+1:N] * x[i+1 : N])
        # Solving the unknown x by solving coeff * x + knowns = b
        x[i] = (Ab[i, -1] - non_target_x_vals)/(Ab[i, i])

    # Validation Check and Answer Printed Below!
    x_np = np.linalg.solve(A,b)
    print(f"Answer from Numpy's to Double Check:")
    print(x_np.flatten())
    print(f"\n")
    print(f"Answer from our implementation:\n")
    print(x)
    return x

x = gaussian_elimination(66)
print("")
```

```
Answer from Numpy's to Double Check:
[-0.65513875  2.27164969 -2.3277355   0.15036795  3.06738052  7.52839867
 -8.50888075  4.0354993  -0.10327575 -1.67601588 -3.11281321 -0.04507417
  0.84457057 -2.85250063 -4.56608997 -3.41111675 -2.72202623  0.48883397
  1.13116448  1.69344259 -2.46553667  4.72071861 -1.94698359 -2.18278142
 -3.78870778 -1.53519058 -0.94963729 -5.18143123 -1.45846812 -2.94596834
 -3.29382593 -0.48875594 -2.62714544 -4.03980592 -2.10168009  0.23553221
 -0.97238499  0.31820809 -4.23293801 -0.60784635  4.08196916  1.45878978
  4.70033964  1.72935184 -1.91493405  0.98023094 -1.09086277  0.35736359
  1.31866606  0.96334999 -0.72893173  0.68905128  1.27615133 -0.15247564
  1.19940675 -4.71790385 -2.26661036  0.78556461  1.59009989  1.93249598
  4.82538883  0.88612512  2.34454569 -4.09314132 -0.04540868  3.66184694]


Answer from our implementation:

[-0.65513875  2.27164969 -2.3277355   0.15036795  3.06738052  7.52839867
 -8.50888075  4.0354993  -0.10327575 -1.67601588 -3.11281321 -0.04507417
  0.84457057 -2.85250063 -4.56608997 -3.41111675 -2.72202623  0.48883397
  1.13116448  1.69344259 -2.46553667  4.72071861 -1.94698359 -2.18278142
 -3.78870778 -1.53519058 -0.94963729 -5.18143123 -1.45846812 -2.94596834
 -3.29382593 -0.48875594 -2.62714544 -4.03980592 -2.10168009  0.23553221
 -0.97238499  0.31820809 -4.23293801 -0.60784635  4.08196916  1.45878978
  4.70033964  1.72935184 -1.91493405  0.98023094 -1.09086277  0.35736359
  1.31866606  0.96334999 -0.72893173  0.68905128  1.27615133 -0.15247564
  1.19940675 -4.71790385 -2.26661036  0.78556461  1.59009989  1.93249598
  4.82538883  0.88612512  2.34454569 -4.09314132 -0.04540868  3.66184694]
```

Results

The results for this program won't be consistent because a new N x N matrix is generated each time. However, to show one of the outputs I recieved and it's comparison to the output provided by numpy's built in solver I've provided the output below.

My Output:

[-0.65513875 2.27164969 -2.3277355 0.15036795 3.06738052 7.52839867 -8.50888075 4.0354993 -0.10327575 -1.67601588 -3.11281321 -0.04507417 0.84457057 -2.85250063 -4.56608997 -3.41111675 -2.72202623 0.48883397 1.13116448 1.69344259 -2.46553667 4.72071861 -1.94698359 -2.18278142 -3.78870778 -1.53519058 -0.94963729 -5.18143123 -1.45846812 -2.94596834 -3.29382593 -0.48875594 -2.62714544 -4.03980592 -2.10168009 0.23553221 -0.97238499 0.31820809 -4.23293801 -0.60784635 4.08196916 1.45878978 4.70033964 1.72935184 -1.91493405 0.98023094 -1.09086277 0.35736359 1.31866606 0.96334999 -0.72893173 0.68905128 1.27615133 -0.15247564 1.19940675 -4.71790385 -2.26661036 0.78556461 1.59009989 1.93249598 4.82538883 0.88612512 2.34454569 -4.09314132 -0.04540868 3.66184694]

Numpy's Output: [-0.65513875 2.27164969 -2.3277355 0.15036795 3.06738052 7.52839867 -8.50888075 4.0354993 -0.10327575 -1.67601588 -3.11281321 -0.04507417 0.84457057 -2.85250063 -4.56608997 -3.41111675 -2.72202623 0.48883397 1.13116448 1.69344259

```
-2.46553667 4.72071861 -1.94698359 -2.18278142 -3.78870778 -1.53519058 -0.94963729
-5.18143123 -1.45846812 -2.94596834 -3.29382593 -0.48875594 -2.62714544 -4.03980592
-2.10168009 0.23553221 -0.97238499 0.31820809 -4.23293801 -0.60784635 4.08196916
1.45878978 4.70033964 1.72935184 -1.91493405 0.98023094 -1.09086277 0.35736359
1.31866606 0.96334999 -0.72893173 0.68905128 1.27615133 -0.15247564 1.19940675
-4.71790385 -2.26661036 0.78556461 1.59009989 1.93249598 4.82538883 0.88612512
2.34454569 -4.09314132 -0.04540868 3.66184694]
```

As you can see above the output found by my program performing Gaussian Elimination, provides results similar to numpy's linear algebra solver which I believe uses LU decomposition.

Performance

In terms of time complexity, Gaussian Elimination is O(n^3) where n is the number of rows and columns for the matrix being solved as doing Forward Elimination takes around n^3 operations and is the dominating term. Backward Substitution afterall, only takes O(n^2) operations and does not dominate.

For space complexity, Gaussian Elimination is O(n^2) as we must store the n x n matrix and the augmented matrix of size n x n+1, which in terms of complexity simplfies to O(n^2).