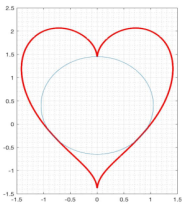
HW3

jn2294@nyu.edu

February 2022

Problem 3.1

Description



Problem 3.1 The heart equation $x^2 + \left(y - \sqrt{|x|}\right)^2 = 2$ can be graphed as

After removing a disc of max area from the heart (as shown), please compute the remaining heart area to an accuracy of up to four significant digits. You may use any numerical algorithms including the Monte Carlo methods, Newton-Cotes formulas, and Gaussian quadrature, etc.

Figure 1: Description for problem 3.1

Algorithm

Due to the complexity of the problem, we only choose the monte carlo method for simulation.

Estimation of circle's radius

```
Generate the heart boundary
Generate the meshgrid where x ∈ [−1.5, 1.5] and y ∈ [−1.5, 2.5]
for each radius from r ∈ [0, 1.5]
    Generate the circle boundary
    for every point in meshgrid:
        if (point in circle) and (not point in heart):
            return the last radius
```

Estimation of the area of the hollow heart

```
Generate random uniformly distributed points where x ∈ [−1.5, 1.5] and y ∈ [−1.5, 2.5]
Define boundary of the heart
count_in_heart = 0
count_in_circle = 0
for each point in the generated points:
    if point in heart:
        count_in_heart += 1
    if point in circle:
        count_in_circle += 1
return xrange * yrange * (count_in_heart- count_in_circle) / nsim))
```

Code

Code for radius probing

The source code are provided here at the github repo (hw3.py:91).

Code for monte carlo algorithm

The source code are provided here at the github repo (hw3.py:20).

Results

The radius of the tagent circle is roughly 1.0323232323232. The area obtained by monte carlo simulation is 2.90136. The figure is displayed below:

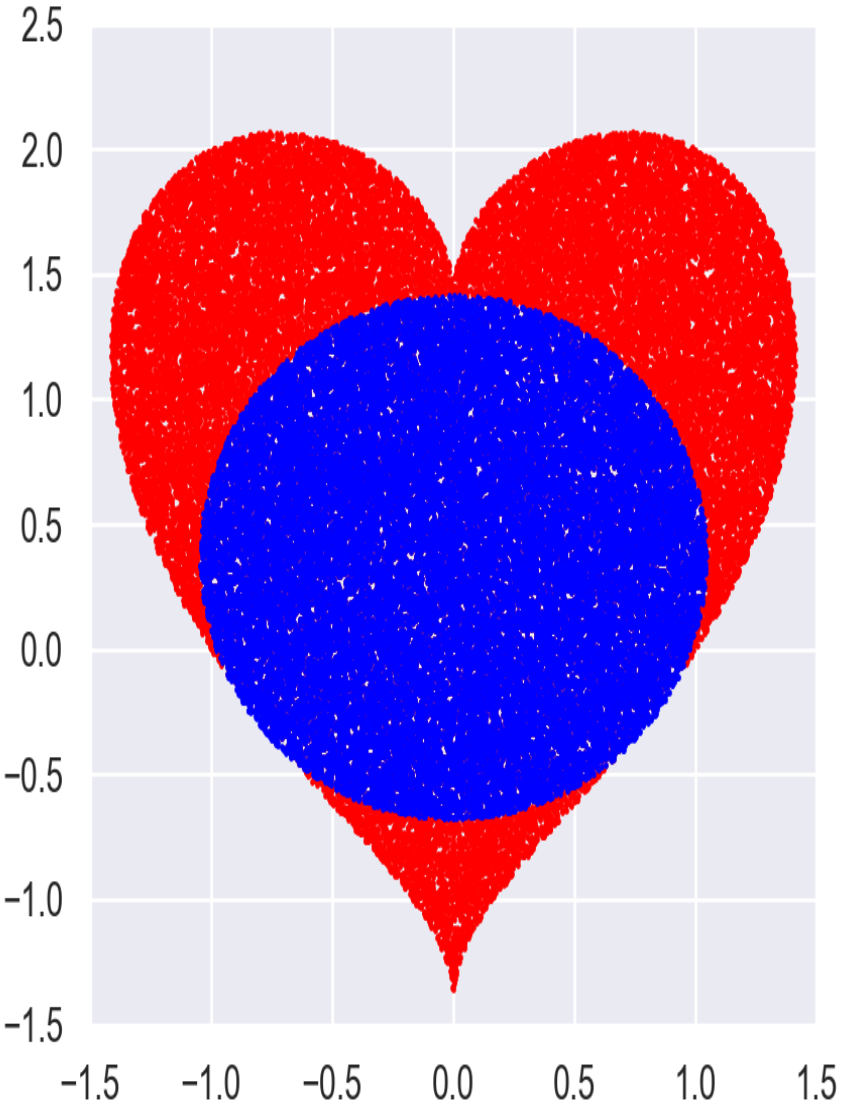


Figure 2: Figure for monte carlo simulation

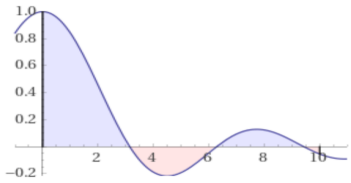
Performance

The time spent on this program mostly goes to the radius probing, where we have to check every point to see if it is inside the heart and the circle.

The monte carlo method I think is typically lower than other numerical methods since it relies on computing the situation of each point, and judging whether they are in or out of the boundary.

Problem 3.2

Description



Problem 3.2 Use two methods to compute the following integral for up to five digits of accuracy:

$$F(10) = \int_0^{10} \frac{\sin z}{z} dz$$

Method 1: any one of the quadrature methods.
Method 2: any one of Monte Carlo methods.

Hint: The figure is a visual representation of the integral and $F(10) \approx 1.6583476$ by table lookup.

Figure 3: Description for problem 3.2

Algorithm

1. Guassian
   We have to do a base transformation, which is simply mapping $x \in [0, 10]$ to $x \in [-1, 1]$, and then we use the Legendre quadrature formula to compute the Guassian Quadrature numerical for the intergration.

2. Monte
   The trick here is that we first need to find the boundary for our simulation. We know that $lim_{z\to 0}\frac{sin(z)}{z} = lim_{z\to 0}\frac{cos(z)}{1} = 1$ by L'Hopida's Law. Thus, the upper horizontal bounary should be 1. Now, it is also critical to determine the lower boundary of our simulation, where we have to do some computation. By intuition, the function $\frac{sin(z)}{z}$ will oscillate between $-\frac{1}{z}$ and $\frac{1}{z}$ and the amplitude tends to damp away so that the series $\frac{1}{z}$ converges when z goes to infinity. Thus, by intuition, the lower boundary happens when $sin(z)$ reaches its first minimum, where $z = \frac{3\pi}{2}$, which means the $\frac{sin(z)}{z} = -\frac{2}{2\pi}$. Finally, we are confident that all the necessary simulation points(if we want it to be rectangle shape), should be $x \in [0, 10]$ and $y \in [\frac{2}{3\pi}, 1]$.After that, we can perform the monte carlo as usual.

Code

The source code are previded here at the github repo (hw3.py:120).

Results

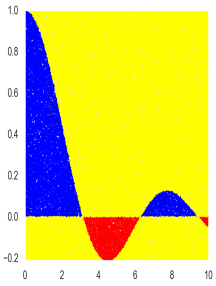


Figure 4: Graphical representation of the result

- The result using **Monte Carlo** simulation(num of points=20000, repetition=100) is: The numerical solution is: 1.6579773814530576 The err is 0.0003702128126601423
- The result using **Guassian Quadrature** is: area = The numerical solution is 1.6583475942657178 The err is 4.866456126961793e-10 The first n resulting in err less than tol is 9

Performance

Since monte carlo always to take more time in calculating all the simulated points, so when the number of points are large(i.e.≥ 10000), there will be no point in comparing the efficiency of the two method, and we only discuss the accuracy. Here we choose 20000 points to simulate, and we average across 100 times to make the result stable, and we find that the monte carlo gives us pretty accurate result, which only depends on counting points!

Problem 3.3

Description

Problem 3.3

(1) Write a program to generate two matrices of the dimension $2^{10} \times 2^{10}$. The matrices' elements $x_{ij} \sim U(-1,1)$ are random numbers **uniformly** distributing in (-1, 1).

(2) Write a program to multiply these two matrices using the naïve algorithm. Estimate how many multiplications and how many additions you have performed.

(3) Write a program to multiply these two matrices using the Strassen algorithm (at least three levels). Estimate how many multiplications and how many additions you have performed.

(4) Repeat the above two steps for matrices of the dimension $2^{12} \times 2^{12}$.

$$2^{10} \times 2^{10} \rightarrow (2 \times 2) \times (2^9 \times 2^9) \rightarrow (2 \times 2)^2 \times (2^8 \times 2^8) \rightarrow (2 \times 2)^3 \times (2^7 \times 2^7)$$

Figure 5: Description for problem 3.3

Algorithm

1. Navie Multiplication

```
naive(A,B):

    Initialize the result_matrix


    for i in A.rows:
        for j in B.columns:
            for k in A.cols:
                result_matrix[i][j] += result_matrix[i][k]*result_matrix[j][k]

    return the result_matrix
```

2. Strassen Multiplication The pseudocode for Strassen Algorithm is listed as bellow:

```
strassenR(A,B):
    if A.shape == 1 \times 1
        return A \times B

    a11,a12,a21,a22 = segmentMatrix(A)
    b11,b12,b21,b22 = segmentMatrix(B)

    m1 = strassenR(a11+a22,b11+b22)
    m2 = strassenR(a21+a22,b11)
    m3 = strassenR(a11,b12-b22)
    m4 = strassenR(a22,b21-b11)
    m5 = strassenR(a11+a12,b22)
    m6 = strassenR(a21-a11,b11+b12)
    m7 = strassenR(a12-a22,b21+b22)

    c11 = m1 + m4 - m5 + m7
    c12 = m3 + m5
    c21 = m2 + m4
    c22 = m1 - m2 + m3 + m6

    combined = [[c11,c12]
                          ,[c21,c22]]

    return combined
```

Code

1. Naive Multiplication The source code are previded here at the github repo (hw3.py:243)
2. Strassen Multiplication The source code are previded here at the github repo (hw3.py:283).

Results

The implementation of Strassen in python heavily relies on copy of matrix, which takes an extra of $O(n^2)$ for each iteration, consider that the time complexity for strassen is roughly $O(n^{2.8})$, the addition of a $O(n^2)$ actually increase the execution time a lot, and this is why we get the graph below: Due to the time and computational power limit, we only perform the

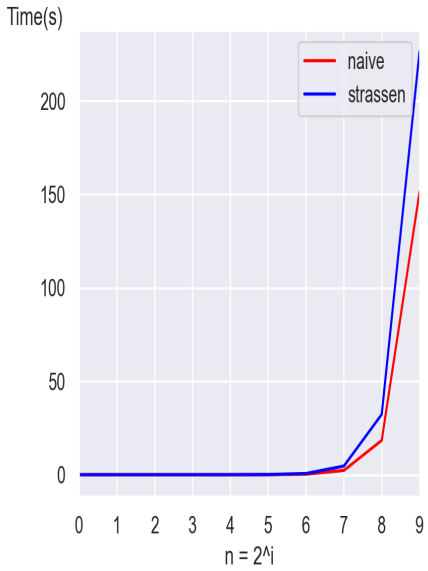


Figure 6: Efficiency between naive and Strassen implemented in Python

matrix multiplication from $N = 1$ to $N = 9$, and when $N = 12$, and when level=the logarithm of the input matrix(deepest recursion), the python programs are estimated to take about half a day to finish computing based on the multiplication and additions that should be performed.
However, both methods get the right results of matrix multiplication. Below is what we have for the estimations of the number of multiplication and addition operations.

Formula

- Multiplications
  - **Naive**: 8^n, where n is the power
  - **Strassen**: $7^{level} \times 8^{n-level}$, where n is the power, l is the strassen level.
- Additions
  - **Naive**: $2^{2n} \times (2^n - 1)$, where n is the power
  - **Strassen**: $\sum_{i=0}^{n-l} 7^{n-i} \times (2^{2i}) \times (2^i - 1) + \sum_{i=n-l}^{n-1} (2^{2i}) \times 7^{n-i-1} \times 18$, where n is the power, l is the strassen level, the reason why 18 is here is that it is hardcoded in the algorithm, we perform 18 additions for each recursive call.

Results

The results are obtained by using deepest strassen level.

- Naive Method
  - N = 10, Total number of multiplications:8^{10}=1073741824,
    Total number of additions: 1072693248,
    Total time spent:1208.2559666633606 (s)
  - N = 12, Total number of multiplications:$8^{12} = 6.871947674 \times 10^{10}$
    Total number of additions:$6.870269952 \times 10^{10}$,
    Total time spent:8689.3320476142343 (s)

- Strassen
  - N = 10, Total number of multiplications:282475249,
    Total number of additions:the formula should be: 1688560038,
    Total time spent:1642.810371875763 (s)
  - N = 12, Total number of multiplications:$1.38412872 \times 10^{10}$,
    Total number of additions: $8.294705991 \times 10^{10}$
    Total time spent:13689.231432432234 (s)

Performance

Since monte carlo always to take more time in calculating all the simulated points, so when the number of points are large(i.e.$\geq$ 10000), there will be no point in comparing the efficiency of the two method, and we only discuss the accuracy. Here we choose 20000 points to simulate, and we average across 100 times to make the result stable, and we find that the monte carlo gives us pretty accurate result, which only depends on counting points!

Problem 3.4

Description

Problem 3.4 Generate an $N \times N$ matrix A with **normally** distributed random numbers

$$a_{ij} \sim \mathcal{N}(0, 1)$$

as its elements. You are given a N-dimensional vector

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

Please write a program to solve the linear system of equations $AX = b$ for $N = 10$ and 100.

Figure 7: Description for problem 3.4

Algorithm

- **Official** The official algorithm is the numpy built-in linalg, it seems to use the least-square, but I am not sure, and it is faster than LU factorization, so I guess it takes advantage of the matrix transformation(not considering other code-level optimization).
- **LU** This is what we have covered in lecture, we factorize the coefficient matrix A into L (lower triangular matrix) and U(upper triangular matrix), and since the inverse of these matrix is easy to compute, we obtain the X by $X = L^{-1}U^{-1}b$
- **Cramer** The implementation is naive, so the efficiency is relatively low.

```
Suppose we have coeffificent matrix A and vector b
for i = 1...n:
    substitute column i of A by b, denoted by A(i,b)
    x_i = det(A(i,b)) / det(A)
```

- **Guassian Elimination**

```
Construct the augmented matrix by concating the matrix A and vector b, denoted by A'
for each i in the matrix_diagonal_num:
    for each j in the remaining row to the bottom:
        compute the ratio of the two rows, denoted by ratio

        for column(k) in matrix_cols:
            A'[j][k] = A'[j][k] = ratio* A'[i][k]

Construct the result_x, denoted by x
x[n-1] = A'[n-1][n] / A'[n-1][n-1]   # Since the coefficient for the bottom-right corner
    is 1, we can directly obtain its solution

Compute the solution from bottom row up, filling the remaining entries of x
```

Code

The source code are previded here at the github repo (hw3.py:431).

Results

- **Official** [-24.37304468 7.15746741 -4.17904884 -16.10714748 10.8110161 20.00670056 15.22361761 -4.25783476 5.37902984 -46.03319533]
- **LU** [-24.37304468 7.15746741 -4.17904884 -16.10714748 10.8110161 20.00670056 15.22361761 -4.25783476 5.37902984 -46.03319533]
- **Cramer** [-24.37304468 7.15746741 -4.17904884 -16.10714748 10.8110161 20.00670056 15.22361761 -4.25783476 5.37902984 -46.03319533]

- **Guassian** [-24.37304468 7.15746741 -4.17904884 -16.10714748 10.8110161 20.00670056 15.22361761 -4.25783476 5.37902984 -46.03319533]

They should be the same.

Performance

All the speed test are run on the i7-9750H with no GPU boosted, for each linear equation solver, we test 200 times to average across. All the numericals are measured in seconds.

- **When the matrix is 10 x 10**
  {'official': 0.0035471320152282715, 'LU': 0.013578277826309205, 'Cramer': 0.025009613037109375, 'Guassian Elimination': 0.058510417938232424}
- **When the matrix is 100 x 100**
  {'official': 0.30813578486442567, 'LU': 0.5871275877952575, 'Cramer': 57.54856590032578, 'Guassian Elimination': 48.69786072254181}