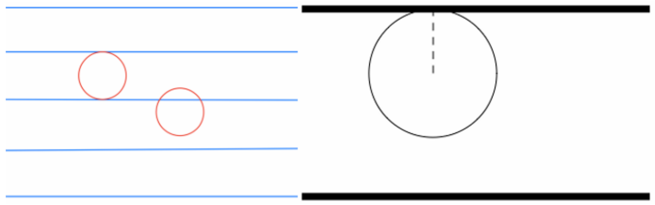
114847849
Kangning Fengwu
AMS326 Homework 3
Source Code Link: https://github.com/kfengwu/AMS326.git

## Problem Description:

**Problem 3-1 (35 points)**: Buffon's Needle refers to a simple Monte Carlo method for the estimation of the value of $\pi = 3.14159265$...among others. Here is in a nutshell of such experiment: Suppose you draw many parallel, and equally distanced (1 inch, for example), lines on a desktop and suppose you have many 1-inch-long needles. Dropping a needle randomly to the table, you see: (1) The needle crosses or touches one of the lines, or (2) the needle crosses no lines. Repeating such droppings $N$ times, you noticed $N_C$ times the needle crosses a line.



For our project, instead of dropping needles, you perform numerical experiments for tossing a disc of diameter $d$ (to be specified below) to parallel lines of distance $w = 1$. Estimate (numerically, of course) the probability when any part of the disc crosses a parallel line for $n = 4{,}444{,}444$ tosses with each of the following disc diameters $d = \frac{1}{10}, \frac{2}{10}, \frac{3}{10}, \frac{4}{10}, \frac{5}{10}, \frac{6}{10}, \frac{7}{10}, \frac{8}{10}, \frac{9}{10}, \frac{10}{10}, \frac{15}{10}, \frac{20}{10}, \frac{30}{10}$.

For $d < w$, you may cross no more than 1 line and the crossing probability depends on "d".
For $d > w$, you may cross more than 1 line and, as such, you must specify the probabilities for crossing, at least, 1-, 2-, 3-, 4-... lines. Obviously, for $d = \frac{20}{10}$ (and $w = 1$, as always),

$$P(1\text{ line}) = 100\%$$
$$P(2\text{ line}) = 100\%$$
$$P(3\text{ line}) = 000\%$$

In the case of $P(3\text{ line})$, we have a tiny exception the disc touches all three lines.

Please estimate, and make a plot for, the probabilities as a function of $d$.

**Note**: The above $n_{\text{realistic}} = 4{,}444{,}444$ is a realistic number and one can get crazy with $n_{\text{dream}} = 4{,}444{,}444{,}444$ for better accuracy. For me, seven "4"'s are enough.

## Algorithm:

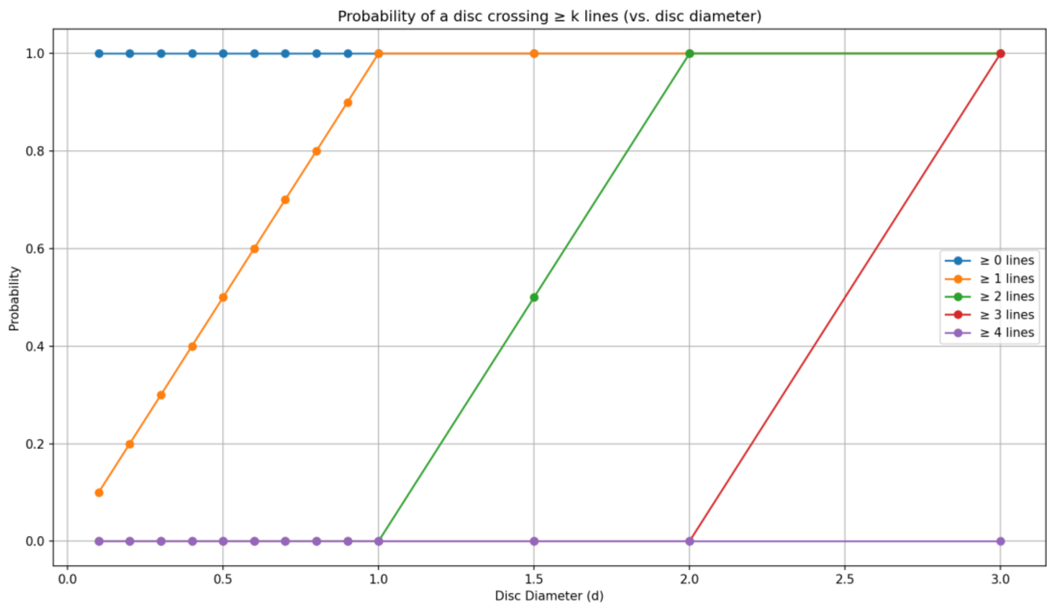
The simulation is performed using Monte Carlo to estimate the probability that a randomly dropped circular disc of a given diameter will cross at least k parallel lines spaced one unit apart. It tests various disc diameters and computes crossing probabilities for k=0 up to 4. For each diameter, the simulation runs 4,444,444 random tosses, generating vertical center positions and calculating the disc's vertical range. By comparing the floor values of the disc's top and bottom edges, it determines how many lines are crossed in each toss. The results are aggregated and normalized to estimate probabilities.

```
  n ← number of tosses
  w ← spacing between lines
  ds ← list of disc diameters
  max_lines ← maximum number of line crossings to consider
  results ← dictionary mapping each diameter d to a list of zeroes of length (max_lines + 1)
FOR each diameter d in ds DO:
  y_values ← generate n random numbers uniformly from [0, w]  // disc center positions

  FOR each y in y_values DO:
    low ← y - (d / 2)
    high ← y + (d / 2)
```

```
        line_low ← floor(low)
        line_high ← floor(high)
        num_crossed_lines ← line_high - line_low
        FOR k from 1 to max_lines DO:
            IF num_crossed_lines ≥ k THEN
                results[d][k] ← results[d][k] + 1

    FOR each diameter d in ds DO:
        FOR k from 1 to max_lines DO:
            results[d][k] ← results[d][k] / n   // convert counts to probabilities

    FOR k from 0 to 4 DO:
        Create a line plot of (d, results[d][k]) for each d in ds
Label axes, add title and legend, and display the plot
```

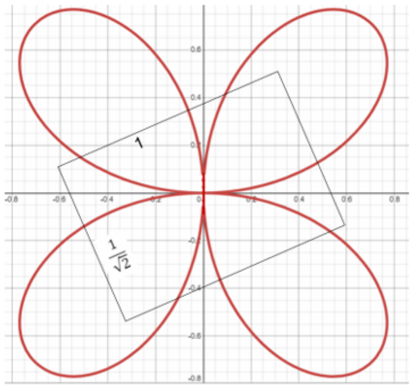
**Result:**

The following plot is generated to display the probabilities as a function of d. It shows that P(k>=0) for all discs is 1, P(k>=1) for discs with diameter < 1 are the same as their diameters, and for the discs with diameter >= 1 are 1, P(k>=2) for the disc with diameter = 1.5 is 0.5, and for the discs with diameter >= 2 are 1,  P(k>=3) for discs with diameter >= 3 is 1, P(k>=4) is 0 for no discs with diameter greater than 3.



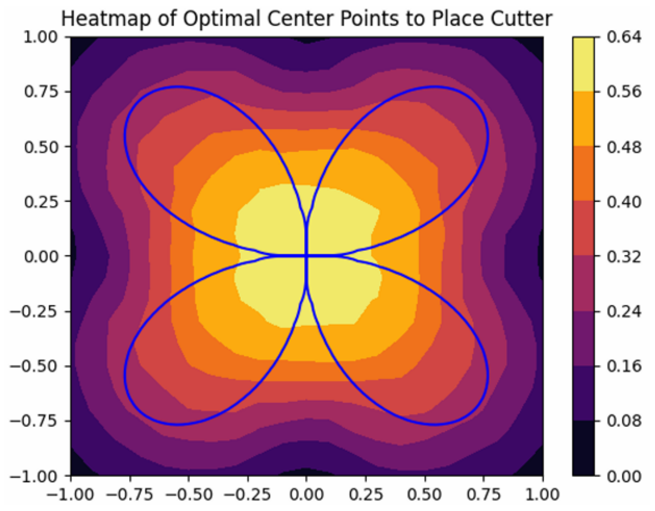
## Problem Description:

**Problem 3-2 (35 points):** The polar equation of a 4-leaf "rose" as graphed is $r = \sin 2\theta$ whose Cartesian coordinate equation is $(x^2 + y^2)^3 = 4x^2y^2$.



Please write a program to enable you to place a rectangular cutter (as shown) of sides $1 \times \frac{1}{\sqrt{2}}$ to cut the most area of the rose. You are suggested to start from a random spot for the cuts.

Notes: (1) The following heatmap shows the area as a function of the center coordinates of the cutter for a given angle. (2) It's highly desirable to generate a 3D heatmap by varying the angle too. Now, we are quite confident that placing the cutter at the center of the rose will likely cut the most area. This depends, of course, on the size and orientation of the cutter. (3) This problem is much richer than a HW one and elucubrating it will likely result in a nice paper.



## Algorithm:

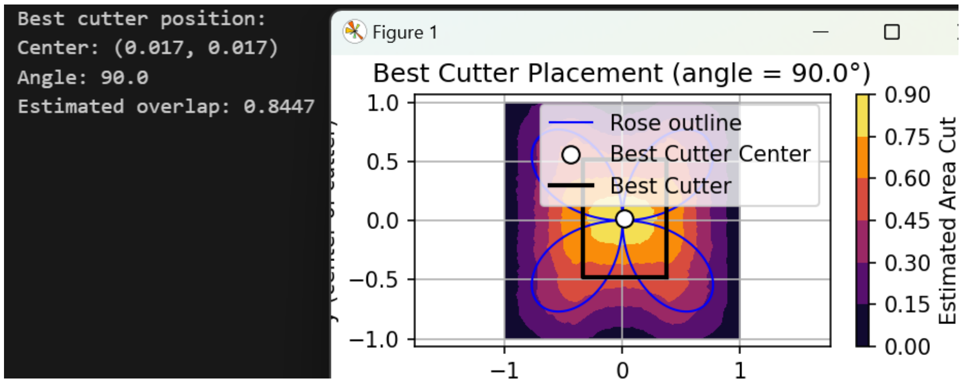
To estimate the amount of overlap between the cutter and the rose at a given configuration, the program uses a Monte Carlo simulation. It generates 3,600 random points uniformly distributed inside the rotated rectangle. These points are then tested against the rose's implicit equation to determine which of them lie within the rose. The fraction of points inside the rose gives a good approximation of the cutter's overlap with the shape. This approach is probabilistic, but with enough sample points, it provides a reliable estimate.

A brute-force grid search is used to evaluate all combinations of cutter center positions and rotation angles from 0 to 180 degree with 18 angles. For each combination, the estimated overlap is calculated, and the one with the highest overlap is recorded as the best configuration. This method guarantees finding the best result within the specified grid resolution, although it is computationally expensive.

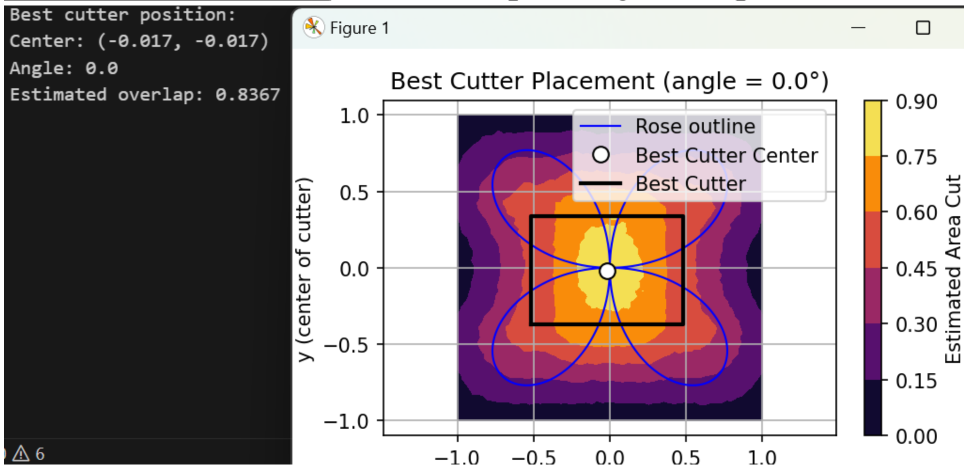
Once the optimal placement and angle are found, the program prints out and visualizes the results. It creates a heatmap showing the estimated overlap across the grid at the best-found angle.

```
Function in_rose(x, y):
    Return ((x^2 + y^2)^3 <= 4 * x^2 * y^2)

Function generate_cutter_points(center, width, height, angle, n = 3600):
    Convert angle from degrees to radians
    Generate n random (u, v) points uniformly in rectangle [-width/2, width/2] × [-height/2, height/2]
    Rotate each (u, v) point by angle
    Translate each rotated point by (x0, y0) to get global (x, y) coordinates
    Return x and y arrays

Function estimate_overlap(center, width, height, angle):
    (x, y) = generate_cutter_points(center, width, height, angle)
    Count number of (x, y) points that satisfy in_rose(x, y)
    Return (number inside rose) / total points

Initialize:
    xs = linspace from -1 to 1 with 60 points
    ys = linspace from -1 to 1 with 60 points
    angles = linspace from 0 to 90 with 18 values
    best_overlap = 0
    best_params = None

For each angle in angles:
    For each x0 in xs:
        For each y0 in ys:
            overlap = estimate_overlap((x0, y0), width=1.0, height=1/sqrt(2), angle)
            If overlap > best_overlap:
                best_overlap = overlap
                best_params = (x0, y0, angle)

Print:
    Best cutter center (x0, y0)
    Best rotation angle
    Estimated maximum overlap

Plot:
    Draw rose curve using r = sin(2θ)
    Plot best cutter center as a point
    Draw rectangle representing the best cutter placement
```

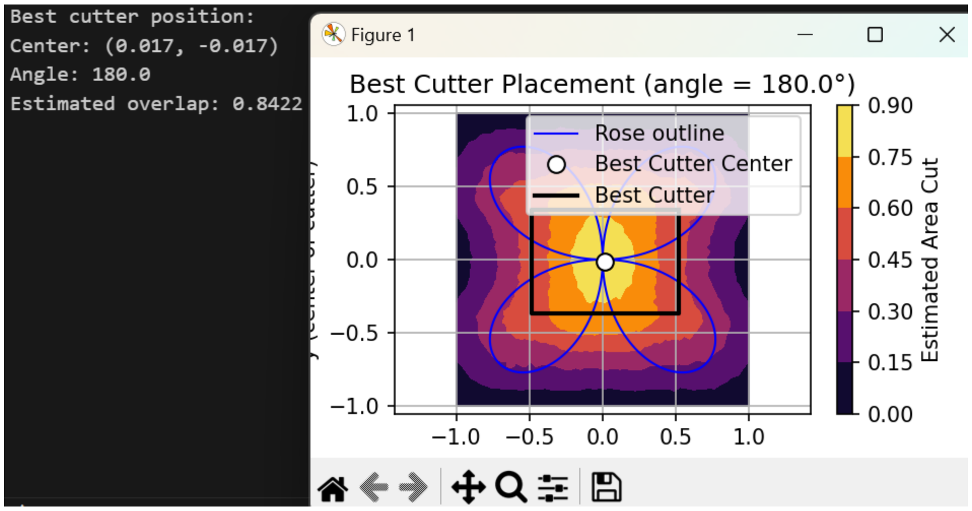
**Result:**
Because of the use of brute-force grid search, it takes about 15 seconds for the program to finish computing the results. After running the programs several times, it is confident to say that the best cutter position should be located at (0,0) with an angle of 0 or 90 or 180 degrees and an estimated overlap of about 84%.



```
Best cutter position:
Center: (0.017, 0.017)
Angle: 90.0
Estimated overlap: 0.8447
```



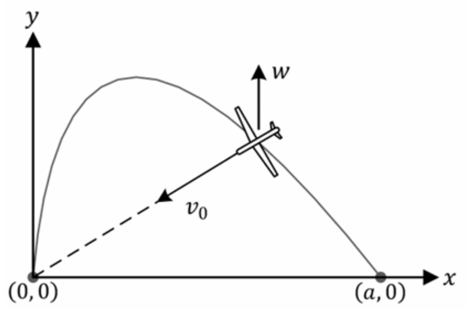
```
Best cutter position:
Center: (-0.017, -0.017)
Angle: 0.0
Estimated overlap: 0.8367
```



```
Best cutter position:
Center: (0.017, -0.017)
Angle: 180.0
Estimated overlap: 0.8422
```

**Problem Description:**

**Problem 3.3 (35 Points):** A flying plane at $(a, 0)$ approaches an airport whose coordinates are $(0, 0)$, which means the plan is $a$ miles east to the airport. A steady south wind of speed $w$ blows from south to north. While approaching the airport, the plane maintains a constant speed $v_0$ relative to the wind and always maintains its heading directly toward the airport. It is interesting to construct the DE and find its solution as the plane's trajectory.



The plane's velocity components relative to the airport tower are

$$\begin{cases} \dfrac{dx}{dt} = -v_0 \cos\alpha = -\, v_0 \dfrac{x}{\sqrt{x^2 + y^2}} \\ \dfrac{dy}{dt} = -v_0 \sin\alpha + w = -\, v_0 \dfrac{y}{\sqrt{x^2 + y^2}} + w \end{cases}$$

which is a system of two DEs with $x$ and $y$ are dependent variables and $t$ independent variable. We are not interested in time so we may eliminate explicit dependence on time and build the following DE between $x$ and $y$:

$$\begin{cases} \dfrac{dy}{dx} = \dfrac{y}{x} - k\sqrt{1 + \left(\dfrac{y}{x}\right)^2} \\ y(x = a) = 0 \end{cases}$$

where we have defined $k = \frac{w}{v_0}$.

Now, you are given the values $a = 100$, $w = 44$, and $v_0 = 88$. Please use any solution method including the Euler's methods and the Runge-Kutta method to compute the plane's trajectory until it lands at the airport.

Notes: (1) A trajectory is an assembly of points in the xy-plane, or a function $F(x, y) = 0$, or a curve (as shown above). (2) You may express the trajectory in a table or in a more-preferred curve. (3) The problem can be made more realistic and more sophisticated by adjusting the wind's direction, etc. I'm not going to do it for a course.

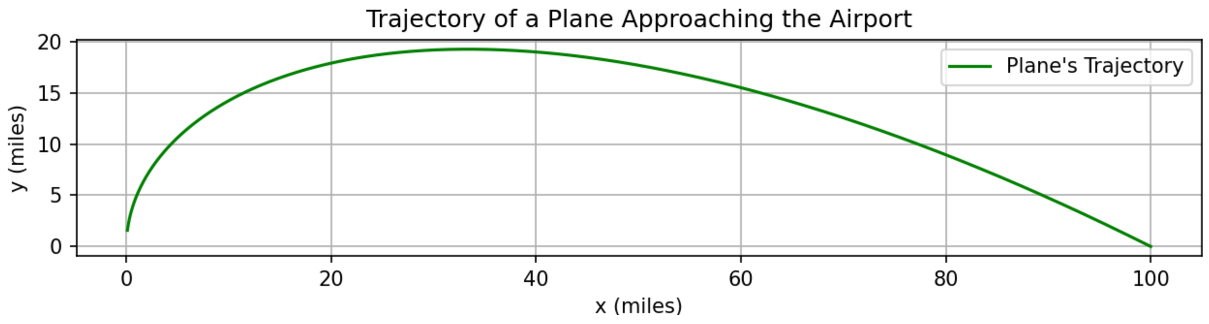
**Algorithm:**

To determine the flight path, the program sets up a nonlinear differential equation that models the change in the plane's vertical position y with respect to its horizontal position x. This equation is solved numerically using the 4th order Runge-Kutta method, which iteratively computes the trajectory from the initial point (x=100, y=0) back toward the airport. A small value is used to avoid division by zero as the plane nears the origin, and the step size is set to -0.01 for integrating from a to 0 and higher precision. Once the simulation completes, the program plots the trajectory, showing how the plane curves its path to compensate for the wind.

```
FUNCTION differential equation f(x, y):
    IF ABS(x) < epsilon THEN
        x ← epsilon            // Avoid division by zero
    ratio ← y / x
    RETURN ratio - k * sqrt(1 + ratio^2)
```

```
FUNCTION Runge-Kutta 4th Order Method:
    INPUT: function f, initial (x0, y0), step size h, number of steps n
    INITIALIZE lists xs ← [x0], ys ← [y0]
    FOR i FROM 1 TO n:
        x ← last value in xs
        y ← last value in ys
        COMPUTE:
            k1 ← f(x, y)
            k2 ← f(x + h/2, y + (h/2) * k1)
            k3 ← f(x + h/2, y + (h/2) * k2)
            k4 ← f(x + h, y + h * k3)
        y_next ← y + (h/6) * (k1 + 2*k2 + 2*k3 + k4)
        x_next ← x + h
        APPEND x_next to xs
        APPEND y_next to ys
    RETURN xs, ys

CALL Runge-Kutta method with f, x0, y0, h, and n_steps
STORE result as x_vals, y_vals

PLOT:
    Plot x_vals vs y_vals with labels
    Mark axes, set equal aspect ratio
    Display the plot
```

**Result:**


Trajectory of a Plane Approaching the Airport
Plane's Trajectory
y (miles)
x (miles)