Michael VanDenburgh
SBU ID:

Homework 3 Report

Problem 3.1

1 Problem Description

A ferryboat tries to cross a river. We represent the river with a 2-dimensional Cartesian coordinate system such that the y-axis lies at the river's west bank and points north and the x-axis runs from west to east. The water is assumed to flow strictly northward at all time; it's speed is denoted by the following equation:

$$w(x) = 4v_0(\frac{x}{a} - \frac{x^2}{a^2})$$

Its speed varies depending on the x-coordinate its location. Assume the water flow drags the boat at full efficiency and ignore water resistance.

The boat tries to cross the river from a point $(a, 0)$ to dock at $(0, 0)$ moving at a constant speed v_0 relative to the water and keeping its head aimed toward the docking point $(0, 0)$ at all times. Compute the boat's trajectories for three cases of boat speeds: $v_B = 7, 14, 21$ using a river width $a = 7777$ and $v_0 = 14$.

2 Algorithm Description

2.1 Analytical Simplification

We know the boat's velocity components relative to the riverbanks are

$$\frac{dx}{dt} = -v_B \cos\alpha = -v_B \frac{x}{\sqrt{x^2+y^2}} \tag{1}$$

$$\frac{dx}{dt} = -v_B \sin\alpha + w(x) = -v_B \frac{y}{\sqrt{x^2+y^2}} + 4v_0(\frac{x}{a} - \frac{x^2}{a^2}) \tag{2}$$

This is a system of two differential equations with dependant variables x and y and independent variable t. Thus, the system can be manipulated analytically to eliminate the independent variable t, resulting in the following differential equation:

$$\frac{dy}{dx} = \frac{y}{x} - \frac{w(x)}{v_B(\frac{x}{\sqrt{x^2+y^2}})} \tag{3}$$

Since the boat has a speed of zero at position $x = a$, the following initial condition can be stated:

$$y(a) = 0$$

This initial condition combined with equation (3) establishes an initial value problem to be solved numerically.

2.2 Solving Initial Value Problem

To solve the initial value problem, I used the 4th order Runge-Kutta method (RK4); given the initial condition $y_0 = a$, we can approximate values at other points using

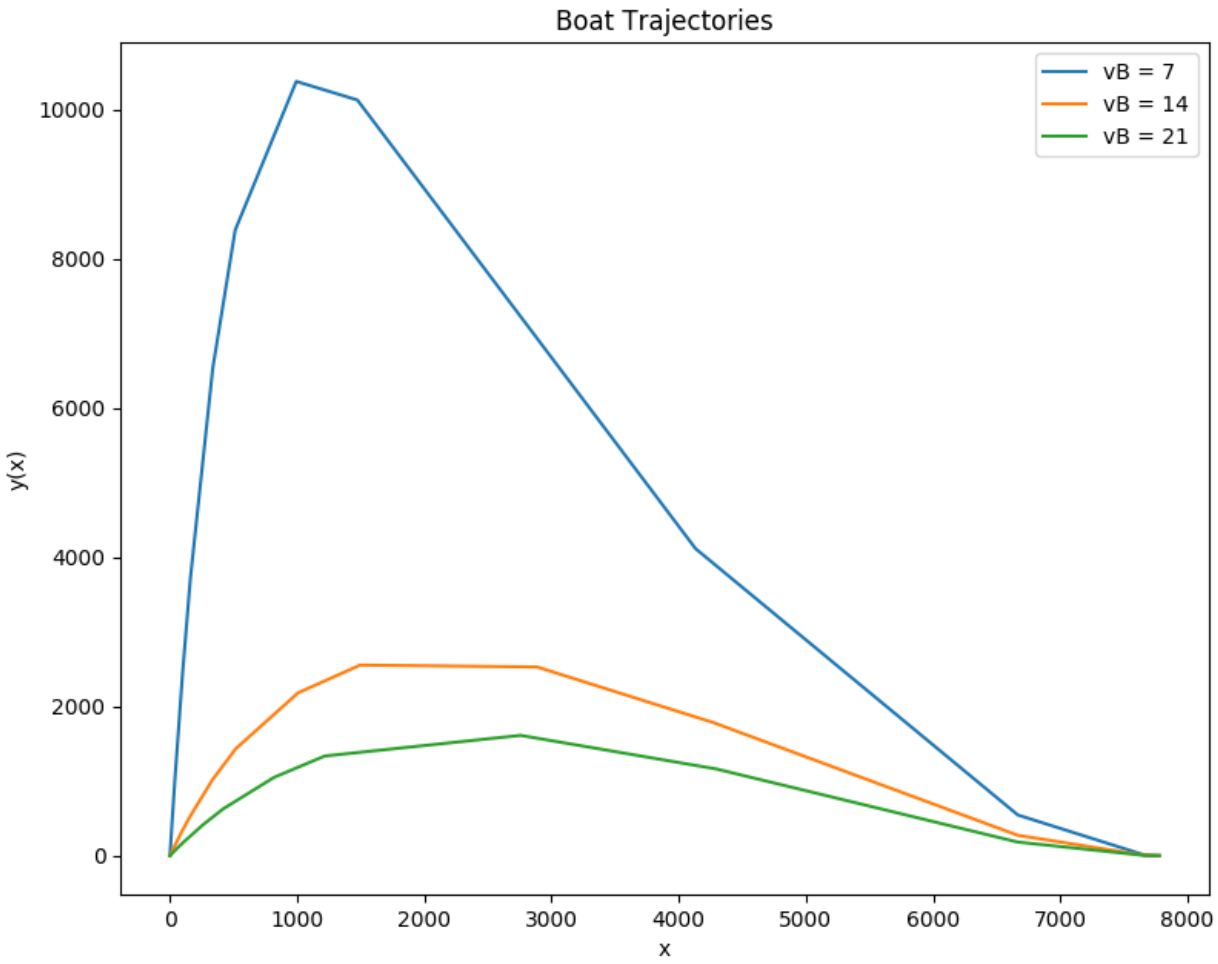
$$y_{n+1} = y_n + h \sum_{i=1}^{s} b_i k_i$$

Where the value of k is given by the following equations:

$k_1 = f(t_n, y_n)$,
$k_2 = f(t_n + c_2 h, y_n + a_{21} h k_1)$,
$k_3 = f(t_n + c_3 h, y_n + a_{31} h k_1 + a_{32} h k_2)$,
$k_i = f(t_n + c_i h, y_n + ai1 h k_1 + a_{i2} h k_2 + ... + a_{i,i-1} h l_{i-1})$

In this scenario, since the initial condition is given at $y(a = 7777)$, the RK4 algorithm starts at $x = 7777$ and iterates backwards until $x = 0$.

3 Results



4 Brief comments on performance & other questions

I chose to use the Runge-Kutta method instead of one of Euler's methods because it tends to be more accurate. The RK4 function runs in $O(\frac{x_n - x_0}{h})$ time; as the step size h increases, the longer the algorithm takes to run. Therefore there is a tradeoff between accuracy and time.

Problem 3.2

1 Problem Description

Perform numerical experiments for tossing a disk of diameter $d = \frac{3}{4}$ to parallel lines of distance $w = 1$. Estimate the probability when any part of the disk crosses a parallel line for 1,984,444,444 tosses.

2 Algorithm Description and Pseudo-Code

This variant of the Buffon Needle problem can be approximated with a Monte Carlo simulation. When tossing a disk there is one random variable to consider - the distance from the center of the disk to the nearest parallel line after it lands. This distance is at least 0 and at most $\frac{w}{2}$, where w is the distance between the parallel lines. If the distance from the center of the disk to the nearest parallel line is less than the disk's radius, then part of the disk landed on a parallel line.

To simulate a disk toss in my program I sampled a random number from a uniform distribution of range $(0, \frac{w}{2})$ using this algorithm:

$$x_{n+1} = (x_n + x_{n-1}) \; mod \; 1.0 = \begin{cases} x_n + x_{n-1} - 1.0 & if \; (x_n + x_{n-1}) > 1.0 \\ x_n + x_{n-1} & Otherwise \end{cases}$$

Source: Prof. Deng's Lecture 2 notes

This variable represents the distance from the center of the disk to the nearest parallel line after it lands. Then I determine whether or not it landed on a line and return the appropriate result. To estimate the probability of a disk landing on a line, I conducted 1,984,444,444 tosses and found the mean result.

Pseudo code:

```
w = 1
d = 3/4
sample_size = 1984444444

function toss_disk():
    distance_from_nearest_line = uniform(0, w/2)

    if distance < d/2
        return 1
    else
        return 0

sum = 0
for i = 0 to sample_size:
    sum = sum + toss_disk()
return mean(sum)
```

3 Results

The result I got from my program approaches 0.75 as more disks are tossed. Therefore I conclude that the probability of any part of the disk crossing a parallel line is 75%.

Extra credit probabilities (using same method with $w = 1$):

d	Probability
$\frac{1}{10}$	10%
$\frac{2}{10}$	20%
$\frac{3}{10}$	30%
$\frac{4}{10}$	40%
$\frac{5}{10}$	50%

d	Probability
$\frac{6}{10}$	60%
$\frac{7}{10}$	70%
$\frac{8}{10}$	80%
$\frac{9}{10}$	90%
$\frac{10}{10}$	100%

4 Brief comments on performance & other questions

The program runs in $O(N)$ time, where N is the number of disk tosses. This linear time complexity means that we can use sample sizes of more than a billion and still receive results in an acceptable amount of time.

Due to the large sample size, a high-level interpreted programming language like Python or MATLAB is not feasible to use for this Monte Carlo simulation. Instead I used C, which was able to perform the simulation in about 37 seconds. In addition, I had to make sure not to save any unnecessary values to prevent completely filling up available memory or slowing down the program.

Unfortunately the large sample size makes graphing infeasible, since it would require storing all 1,984,444,444 probabilities. I was able to produce a graph using a smaller sample size of 100,000; while it is not as accurate, it still illustrates the problem fairly well -