Estimation of π using Monte Carlo Simulation

2202545E
School of Physics & Astronomy, University of Glasgow
15th of December

Abstract

π is one of the most widely spread mathematical constants used in several areas of both mathematics and physics. Its value, $(\pi = 3.1415926536)$, is defined by the ratio of a circle's circumference to its diameter. It has an infinite number of decimals and is therefore an irrational number and cannot be expressed by a common fraction. [1] The value of π can be approximated by fractions such as $\frac{22}{7}$ and $\frac{333}{106}$ or by a variety of methods invented by different mathematicians [2]. One of these methods is the Buffon's needle, proposed by the Comte de Buffon in 1777, which consists of randomly throwing a needle onto a board on which parallel, equally spaced lines are drawn and looking at the probability of the needle landing across one of the lines [3]. Using Monte Carlo in order to simulate the random throws, the probability was explored and the corresponding estimations of π calculated with the error that the Monte Carlo Method causes. The method overall consistently gives estimated π values that are close to actual value though is dependent on the choice of number of iterations. Each run of the method will give a different result due to the random number generation used in Monte Carlo but in order to continually achieve a result of $\pi \pm 0.1$ one will need $10,000$ iterations or above. Improvements can be made in order to increase efficiency of the method depending on the what it will be used for and the precision of π needed.

Contents

1 Introduction

1.1 Buffon's Needle Problem

A needle of length *l* is thrown at a board with lines drawn at equal spacing *t*. Let *x* be the distance from the centre of the needle to the closets line and θ the angle between the needle and the lines (here vertical). If equation (1) is satisfied we have that the needle crosses one of the lines since the horizontal distance from the center of the needle to a line is $d = \frac{l}{2}\sin\theta$.

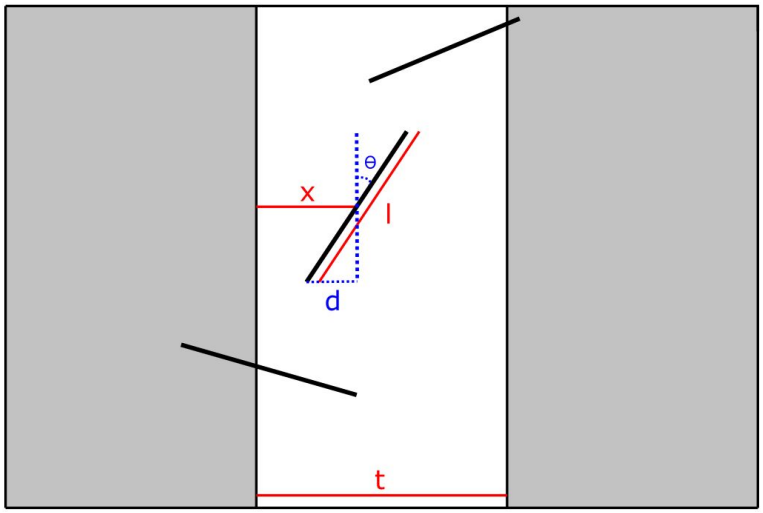


Figure 1: Buffon's Needle Problem [3]
(Created using InkScape.)

$$x \leq \frac{l}{2}\sin\theta \tag{1}$$

The probability of the needle landing on one of the lines can be found by equation (3) and is found by observing that $0 < x < \frac{t}{2}$ and $0 < \theta < \frac{\pi}{2}$ for all throws and $0 < x < \frac{l}{2}\sin\theta$ for the throws that satisfy the condition.

If we throw the needle *N* times and it lands *n* times on a line then we have that the value of π can be estimated using equation (2). [3]

$$\pi \approx \frac{2\,lN}{nt} \tag{2}$$

$$\frac{\int_0^{\frac{\pi}{2}}\int_0^{\frac{l}{2}\sin\theta} dx\,d\theta}{\int_0^{\frac{\pi}{2}}\int_0^{\frac{t}{2}} dx\,d\theta} = \frac{\int_0^{\frac{\pi}{2}} \frac{l}{2}\sin\theta\,d\theta}{\int_0^{\frac{\pi}{2}} \frac{t}{2}\,d\theta} = \frac{\frac{l}{2}}{\frac{\pi t}{4}} = \frac{2l}{\pi t} \tag{3}$$

We can in equation (3) see that the probability of the needle landing across one of the lines is directly proportional to the ratio $\frac{l}{t}$. This makes sense when considering a needle which is as long or longer than the spacing which would cause a probability of 1 since all throws would land across a line.

1.2 Monte Carlo

The Monte Carlo method can be used to estimate results of processes or operations which otherwise are very hard to solve. The main concept of Monte Carlo is to generate random numbers within specific intervals and then testing whether these variables satisfy a given set of requirements. As the number of iterations of this method increases the final results converge towards the actual answer. [3]

1.2.1 Integration

An example of using the Monte Carlo method is integration estimation as some integrals are very difficult or impossible to simplify and hence numerically solve. The method relies on the fact that the area under the curve can be split into *N* slits which area can be approximated to a rectangle of width $x_{i+1} - x_i$ and height $\langle f(x)\rangle = \frac{f(x_{i+1})-f(x_i)}{2}$ as seen in equation (4). Summing these areas over the entire integral interval gives us equation (5). Here the equation is slightly modified to take into account the importance of some x_i values over others by introducing w_i which is the *weight* of each x_i and $W = \sum_{i=1}^{N} w_i$.

For Monte Carlo, since the points are randomly generated and therefore are of the same importance, we have that $w_i = 1 \; \forall i \in [1, N]$ and hence $W = N$. [3]

$$I_i = \int_{x_i}^{x_{i+1}} f(x_i)dx = (x_{i+1} - x_i)\langle f(x_i)\rangle \tag{4}$$

$$I = \int_{x_1}^{x_N} f(x)dx \approx \frac{(x_N - x_1)}{W} \sum_{i=1}^{N} w_i f(x_i) \tag{5}$$

1.2.2 Error on Monte Carlo

The error on a Monte Carlo estimation can be found by equation(6) where σ is the standard deviation as seen in equation (7). It can be seen that the error on the method is proportional to $\frac{1}{\sqrt{N}}$ meaning that the precision of the result can be controlled by the choice of number of iterations. [3]

$$\sigma_{MC} = \frac{\sigma}{\sqrt{N}}(x_{max} - m_{min}) \tag{6}$$

$$\sigma^2 = \frac{1}{N}\sum_{i=1}^{N} \left[\, f(x_i) - \overline{f}\,\right]^{\,2} = \frac{1}{N}\left(\sum_{i=1}^{N} [\, f(x_i)]^{\,2}\right) - \overline{f}^2 \text{ where } \overline{f} = \frac{1}{N}\sum_{i=1}^{N} f(x_i) \tag{7}$$

1.3 Code

Monte Carlo Estimation: The code (found in appendix A.1) consists of a main function which asks for user input on the length of the needle and the spacing between the lines. An error will occur if the length of the needle is longer than the spacing since then all throws will satisfy the condition and we end up with $\pi \approx 2$ from equation (2). The user will be asked to enter a new needle length in order for the program to continue. The program then loops through each element in a list which contains the wanted number of iterations for the Monte Carlo simulation. The number of iterations together with the needle length and spacing is used to call a function defined outside the main function.

This function defines all the variables needed in order to estimate π together with the error on the Monte Carlo estimation and the absolute and relative error of the estimation compared to the actual value of π. A for loop is initiated for $i \in [0 : N]$ for which a random number *j* is generated in order to get *x*, θ and *y*. The c++ function $srand$ is used with the current time used as seed is used for the random number generation. The variables *x*, θ and *y* are used in order to check the two conditions $x \leq \frac{l}{2}\sin\theta$ and $x \leq \frac{1}{2}\sqrt{l^2 - 4y^2}$ respectively. Separate counters and sums are increased if any or both of these conditions are satisfied. After the loop finishes the counters together with the length of the needle and the spacing are used in order to calculate the π estimation with the corresponding absolute and realtive error compared to the actual value of π. The error on each of the two Monte Carlo methods is calculated using the sums found throughout the loop. All results are put into a file in order to save and plot them.

Plotting of Results: The code (found in appendix A.3) used for plotting the data for a number of runs of the code consists of a max and min *x* value that can be chosen depending on the variation in number of iterations used for the estimations of π. The script will iterate through the command line arguments given to it when run from the terminal. The files should be non-zero and exist in the same folder as where the script is run and this is checked by the program. Separate Gnuplot jpeg files are created for each file given to the script where the estimated π value with its error-bars is plotted against the corresponding number of iterations used to get the estimate. A line at $y = \pi$ is plotted for comparison.

2 Estimating π

We will in this simulation approximate the value of π using Buffon's Needle where the integrals in equation (3) will be estimated using Monte Carlo. This will be done by generating *N* random *x* and θ values and seeing if they satisfy equation (1). The number of needles that cross a line, *n*, will then together with the total number of throws be used to find an estimated value of π. The c++ code used for this simulation can be found in appendix A.1.

The error on each iteration is calculated using equations (6) and (7) using $f(x) = \frac{l}{2}\sin\theta$ and gives the equations seen in (8).

$$\sigma^2 = \frac{1}{N}\left(\sum_{i=0}^{N}\left[\frac{l}{2}\sin\left(\theta_i\right)\right]^2\right) - \left(\frac{1}{N}\sum_{i=0}^{N}\left[\frac{l}{2}\sin\left(\theta_i\right)\right]\right)^2 \qquad \sigma_{MC} = \frac{\sigma}{\sqrt{N}}\frac{\pi}{2} \tag{8}$$

Since Monte Carlo relies on generation of random numbers, different results will be gotten every time the code is run, an example can be seen in table 1 (full output from code can be found in appendix A.2.) We can here see that even for low numbers of iterations the relative error is small, though this is dependent on the random *x* and θ values which in this case the throws were 'lucky'.

Number of iterations	**Estimated π**	ϵ_{rel}
100	3.13725 ± 0.0647956	0.00138073
500	3.07692 ± 0.0296079	0.0205849
1000	3.13112 ± 0.0209644	0.00722062
5000	3.10921 ± 0.00938921	0.0103074
10000	3.11466 ± 0.0066555	0.00857345

Table 1: Table of results for different number of iteration with $l = 4$ and $t = 5$

The code was run ten times ($l = 4$, $t = 5$) and the results including error bars can be seen in figure 2.

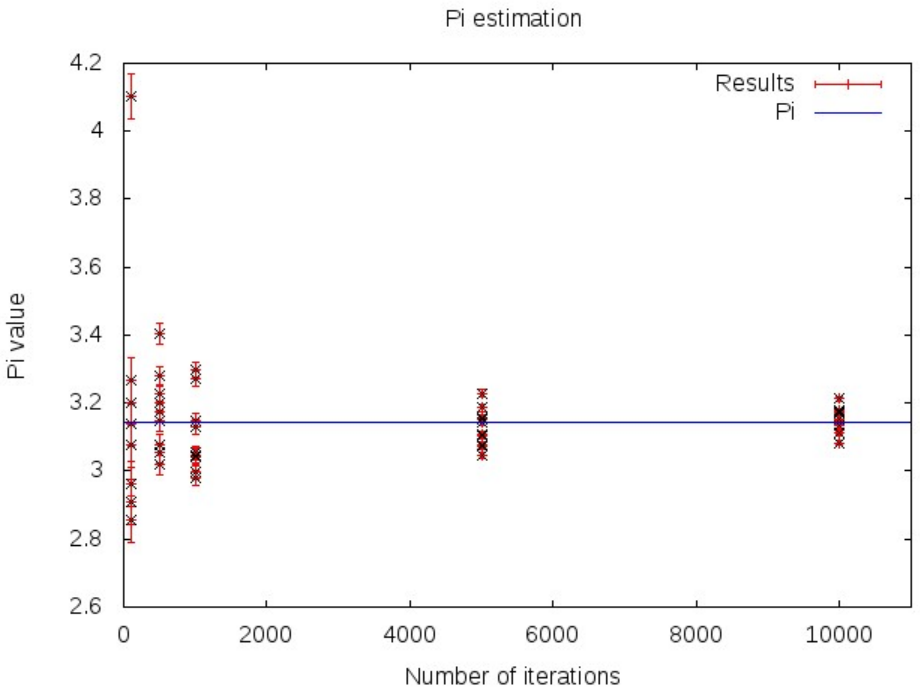
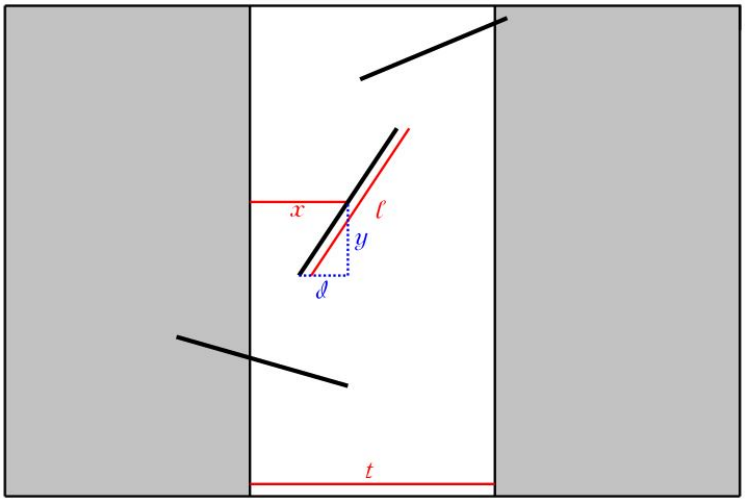


Figure 2: Results with error-bars made in Gnuplot

We here see that the results vary depending on the run but overall the results converge towards the actual value of pi as the number of iterations increases. Depending on the wanted level of precision there is a minimum number of iterations needed in order to consistently get estimated π values that are within a chosen error band.

3 Monte Carlo Method without using π

As we can see previous method can be used to estimate π though π is used in the generation of the random angle θ. The process can easily be changed by changing from polar to Cartesian coordinates though this complicates the integrals which are used to calculate the $\frac{n}{N}$ ratio. The condition is now (9) which is found using Pythagoras's theorem and figure 3 shows a depiction of the scenario.



$$x \leq \frac{1}{2}\sqrt{l^2 - 4y^2} \tag{9}$$

The probability of the needle landing on one of the lines can be found by equation (11) and are found by observing that $0 < x < \frac{t}{2}$ and $0 < y < \frac{l}{2}$ for all throws and $0 < x < \frac{1}{2}\sqrt{l^2 - 4y^2}$ for the throws that satisfy the condition. (The intermediate steps of the upper integral can be found in appendix A.4.) [4]

If we throw the needle *N* times and it lands *n* times on a line then we have that

$$\pi \approx \frac{4\,tn}{lN} \tag{10}$$

Figure 3: Buffon's Needle Problem wihtout π
(Created using InkScape.)

$$\frac{\int_0^{\frac{l}{2}}\int_0^{\frac{1}{2}}\sqrt{l^2-4y^2}\,dx\,dy}{\int_0^{\frac{l}{2}}\int_0^{\frac{t}{2}}dx\,dy} = \frac{\int_0^{\frac{l}{2}}\frac{1}{2}\sqrt{l^2-4y^2}\,dy}{\int_0^{\frac{l}{2}}\frac{t}{2}\,dy} = \ldots = \frac{\frac{\pi l^2}{16}}{\frac{lt}{4}} = \frac{\pi l}{4t} \tag{11}$$

The error on the Monte Carlo method on the integral is found by equations (6) and (7) by using $f(x) = \frac{1}{2}\sqrt{l^2 - 4y^2}$ and the final equations can be seen in (12).

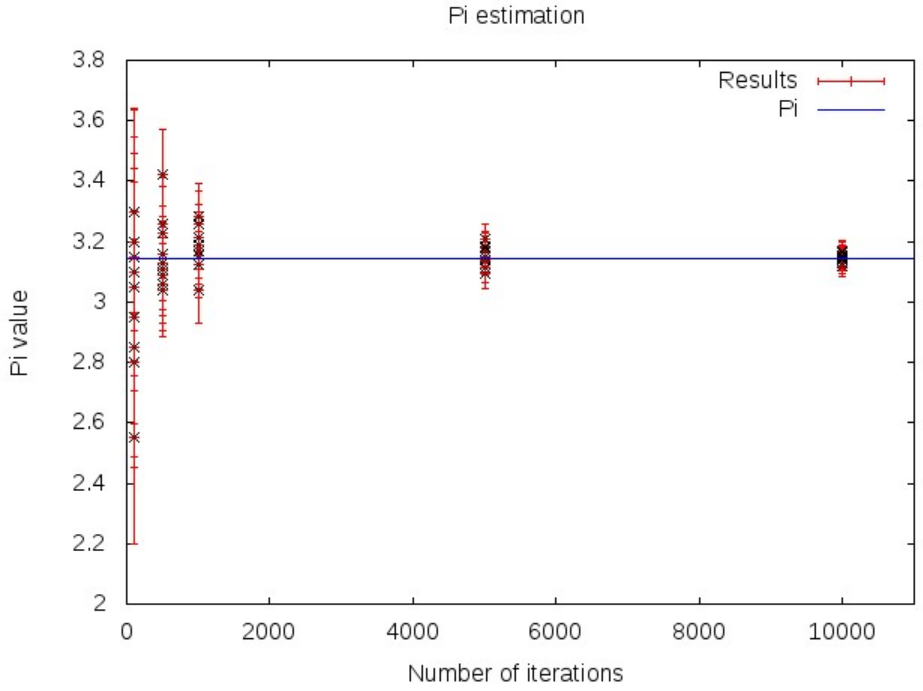
$$\sigma^2 = \frac{1}{N}\left(\sum_{i=0}^{N}\left[\frac{1}{2}\sqrt{l^2-4y^2}\right]^2\right) - \left(\frac{1}{N}\sum_{i=0}^{N}\left[\frac{1}{2}\sqrt{l^2-4y^2}\right]\right)^2 \quad \sigma_{MC} = \frac{\sigma}{\sqrt{N}}\frac{l}{2} \tag{12}$$

Running the code from appendix A.1, we as an example get the results which can be seen in table 2. We again see that for all number of iterations the estimation is relatively good which can be expected since the only difference to the previous method is a change of basis of coordinates.

Number of iterations	Estimated π	ϵ_{rel}
100	3.15 ± 0.341822	0.00267617
500	3.03 ± 0.155711	0.0355211
1000	3.085 ± 0.110132	0.018014
5000	3.141 ± 0.0489414	0.000188637
10000	3.1315 ± 0.034648	0.00321259

Table 2: Table of results for different number of iterations.

The code was run ten times ($l = 4$, $t = 5$) and the results including error bars can be seen in figure 4.



We again see that the results vary depending on the run but overall the results converge towards the actual value of pi as the number of iterations increases. Compared to the previous method is seems like this method converges faster. We again see that a given number of iterations is needed in order to consistently be within a given error band.

Figure 4: Results with error-bars (created using Gnuplot)

4 Discussion & Conclusion

Using the Monte Carlo method in order to simulate Buffon's needle gives a fairly good estimation of π depending on the chosen number of iterations. Looking at figure 2 and 4, we see that in order to get within an error band of ± 0.1 for all runs we need to chose a number of iterations of more than $10,000$. On the figures we also see that it might be possible to find function which describes the relationship between the number of iterations and the estimated π value range. It looks to be some function proportional to either $\frac{1}{x}$, $\frac{1}{\sqrt{x}}$ or e^{-x}. If the code had been run for more values a regression or interpolation could be made and an approximate function could then be used in order to chose the number of iterations needed in order to achieve a certain precision.

Another way to improve the method could be looking at the necessary number of iterations needed to consistently get within a chosen error-band. By this, one could optimize the program to only run for one (or a few different) number of iterations. This can be done by altering the code to have a for loop that iterates until the estimated π value is within the chosen error band. Repeating this several times, we will get a set of number of iterations for which the error band-condition is satisfied and we can chose a number of iterations more wisely and with an eye on effectiveness.This can be used in order to explore the relationship between the $\frac{l}{t}$ ratio and the estimated π value so that process is not to strenuous to calculate for multiple number of iterations.

The code itself is pretty simple and would be difficult to improve. Minor changes are to exclude the calculation of each standard deviation in order to calculate the error on the Monte Carlo method. One could have the equation for the error to include the definition of the standard deviation which would decrease the number of calculations and variables but would make the program harder to read. Since we calculate everything twice (with or without using π to get the results) the code can easily be shortened by only using one of the two methods. In order to plot and save the results they are put into files which can be excluded if

this is not necessary which will also simplify the code.

Overall, the Monte Carlo method can be used in order to simulate Buffon's needle for estimation of π and will above a certain number of iterations (chosen depending on error band) give consistent results. The error on the method is proportional to $\frac{1}{\sqrt{N}}$ and we can therefore chose the precision of our results by varying the number of iterations.

References

[1] Wikipedia.org: *Pi*
https://en.wikipedia.org/wiki/Pi
Visited 13/12/2017

[2] Wolfram MathWorld: *Pi Approximations*
http://mathworld.wolfram.com/PiApproximations.html
Visited 12/12/2017

[3] Dr. Christoph Englert & Dr. David Miller:
Labscript for PHYS4008 Scientific Computing Laboratory
SUPA, School of Physics and Astronomy, University of Glasgow September 2017

[4] Symbolab.com: *Definite Integral Calculator*
https://www.symbolab.com/solver/step-by-step/%5Cint_%7B0%7D%5E%7B%5Cfrac%7Bl%7D%7B2%7D%20%7D%5Cint_%7B0%7D%5E%7B%5Cfrac%7B1%7D%7B2%7D%5Ccdot%5Csqrt%7Bl%5E%7B2%7D-4y%5E%7B2%7D%7D%20%7D1%20dxdy
Visited 07/12/2017

A Appendices

A.1 Appendix 1

C++ code used for Monte Carlo Simulation

```
#define _USE_MATH_DEFINES

#include <iostream>
#include <fstream>;
#include <cmath>
#include <stdlib.h>

using namespace std;

/*
This program will calculate the probablility of an event.
The setup is a board with lines of equal spacing t. A needle of length
    l is thrown at the board and it's position is measured by it's angle
     to horizontal (angle = theta).
This angle is used to calculate the distance from the centre of the
    needle to any line on the board.
We need to calculate the probability of the needle landing on one of
    the lines.

The distance x from the centre of the needle to the closest line is
    given by x <= l/2 * sin(theta) or x <= 1/2 * sqrt(l^2 - 4y^2).

We have the following boundary conditions;
0 < x < t/2, 0 < theta < pi/2, 0 < y < l/2

The probability is equal to 2l/(pi*t) or by (pi*l)/(4t).

All of this can therefore be used to estimate pi.
If we throw the needle N times and n of them land on a line, then pi
    can be estimated by;
pi ~ (N/n) * (2l)/(t) or pi ~ (4t*n)(l*N).

We can therefore use Monte Carlo to estimate pi by generating random x,
     theta and y N times and seeing how many of them satisfy the
    equation x <= (l/2)*sin(theta) or x <= 1/2 * sqrt(l^2 - 4y^2).

 ------ Variables ------
theta - angle between horizontal and needle
x - distance from centre of needle to nearest line
y - vertical diplacement of needle end
l - length of needle
t - spacing of lines
N - number of iterations
```

```
n,m - number of iterations satisfying the equation
S_1, S_2, S1, S2 sums used for calculation of error
sigma_1, sigma_2 - standard deviation squared
eMS_1, eMS_2 - error on Monte Carlo
e_abs1, e_abs2, e_rel1, e_rel2 - absolute and relative error on
    estimations
 */

// ------ Functions -------
void monte_carlo(float l,float t,float N) {
  ofstream results;
  results.open("results.txt", std::ios_base::app); // writing file to
     save output, second term for appending to file

  srand(time(NULL)); // to get random numbers

  float n=0,m=0; // used to contain the values of how many of the
     iterations satisfy the criterium.
  float theta=0, x=0, y=0, j=0;// j is used to generate random numbers
     in the separate intervals
  float S_1=0, S_2=0, S1=0, S2=0; // sums for calculating probability
     and error.

  results << "N=" << N << "\n";

  for (int i=0; i < N; i++){
    j = (double) rand() / (RAND_MAX); // random number between 0 and 1.
    // random distance
    x = (j)*(t/2); // random value between 0 and t/2.

    j = (double) rand() / (RAND_MAX);// re-defining j in order to
       generate random values for theta and y.
    // random angle
    theta = (j)*(M_PI/2); // random value between 0 and pi/2
    y = (j)*(l/2); // random value between 0 and l/2

    if ( x <= (l/2)*sin(theta) ){ // for estimating pi using pi
      n += 1;
      S_1 += sin(theta); S1 += pow(sin(theta),2); // sum for error
         calculations
    }
    if ( x <= 0.5*sqrt(pow(l,2)-4*pow(y,2)) ){
      // for estimating pi without using pi
      m += 1;
      S_2 += sqrt(pow(l,2)-4*pow(y,2)); S2 += pow(l,2)-4*pow(y,2); //
         sum for error calculations
    }
  }
```

```
  // ---Pi calculations---
  float pi_1 = (2*l*N)/(n*t);
  float pi_2 = (4*t*m)/(N*l);
  results << "pi_1= " << pi_1 << "\n" << "pi_2= " << pi_2 << "\n";

  // ---Error---
  // Using pi
  float sigma_1 = (1/N) * S1 - pow( ((1/N) * S_1) ,2); // standard
     deviation squared
  float eMS_1 = (sqrt(sigma_1)/sqrt(N)) * (M_PI/2); // error on
     MonteCarlo

  // absolute and relative error
  float e_abs1 = abs(M_PI - pi_1);
  float e_rel1 = e_abs1 / M_PI;

  // Without using pi
  float sigma_2 = (1/N) * S2 - pow( ((1/N) * S_2) ,2); // standard
     deviation squared
  float eMS_2 = (sqrt(sigma_2)/sqrt(N)) * (1/2); // error on Monte
     Carlo

  // absolute and relative error
  float e_abs2 = abs(M_PI - pi_2);
  float e_rel2 = e_abs2 / M_PI;

  results << "sigma_1= " << sigma_1 << "\n" << "eMS_1= " << eMS_1 << "\
     n" << "e_rel1= " << e_rel1 << "\n";
    results << "sigma_2= " << sigma_2 << "\n" << "eMS2_2= " << eMS_2 <<
        "\n" << "e_rel2= " << e_rel2 << "\n";
  results << "\n"; results.close(); // closing file

  // ---Files for plooting---
  ofstream data1, data2;
  data1.open("data1.txt", std::ios_base::app); // writing file to save
     results for plotting
  data1 << N << " " << pi_1 << " " << eMS_1 << "\n"; data1.close();

  data2.open("data2.txt", std::ios_base::app);
  data2 << N << " " << pi_2 << " " << eMS_2 << "\n"; data2.close();
}


// ----- Main Function -----
int main(){
  // Clear file for results
  ofstream results; results.open("results.txt"); results.close();
```

```
  // ---Getting variables---
  float l;
  cout << "Enter length of neeedle: ";
  cin >> l;

  float t;
  cout << "Enter spacing between lines: ";
  cin >> t;

  if ( l >= t ){
    cout << "The length is longer than or equal to the spacing. \n";
    cout << "Please enter a length shorter than " << t << ". \n";
    cin >> l;
  }

  // ---Loop to look at different iteration numbers---
  float N[]={100,500,1000,5000,10000};
  for (int i=0; i<(sizeof(N)/sizeof(N[0])); i++ ){
  // loop through the list of iteration numbers
    monte_carlo(l,t,N[i]);
    // call the function which will output the relevant estimation of
       pi together with the error on this.
  }
  return 0;
}
```

A.2 Appendix 2

Results from running code

```
N=100
pi_1= 3.13725
pi_2= 3.15
sigma_1= 0.170158
eMS_1= 0.0647956
e_rel1= 0.00138073
sigma_2= 2.93148
eMS_2= 0.342432
e_rel2= 0.00267617

N=500
pi_1= 3.07692
pi_2= 3.13
sigma_1= 0.177642
eMS_1= 0.0296079
e_rel1= 0.0205849
sigma_2= 2.99581
eMS_2= 0.154811
e_rel2= 0.00369002

N=1000
pi_1= 3.13112
pi_2= 3.155
sigma_1= 0.178124
eMS_1= 0.0209644
e_rel1= 0.003335
sigma_2= 2.98499
eMS_2= 0.10927
e_rel2= 0.00426768

N=5000
pi_1= 3.10921
pi_2= 3.141
sigma_1= 0.178644
eMS_1= 0.00938921
e_rel1= 0.0103074
sigma_2= 2.98329
eMS_2= 0.0488532
e_rel2= 0.000188637

N=10000
pi_1= 3.11466
pi_2= 3.154
sigma_1= 0.179523
eMS_1= 0.0066555
e_rel1= 0.00857345
sigma_2= 2.98038
eMS_2= 0.0345275
e_rel2= 0.00394939
```

A.3 Appendix 3

BASH code used for plotting

```
# !/bin/bash
# ------ Description -------
# This script will plot pi estimations including errorbars
# The script takes command line arguments and will create plots for
    each
#
# There will be created as many jpeg files as filenames inputted.
#
# It is assumed that the files are of the structure
# N pi error
# where N is the number of iterations
# pi is the estimated pi value for the number of iterations
# error is the error on Monte Carlo on the given number of iterations.
#
# -------------------------

# These can be changed depending on the number of iterations used.
x_min=0
x_max=11000

# For loop, looping through the command line arguments.
for var in "$@"
do
    file=$var

    # This statement checks whether the file entered exists and is non-
        zero.
    if [ -e "$file.txt" ] && [ -s "$file.txt" ]; then

        gnuplot <<EOF

set terminal jpeg
set output '${file}.jpeg'

set title "Pi estimation"
set xlabel "Number of iterations"
set xrange [${x_min}:${x_max}]
set ylabel "Pi value"

f(x) = 3.14159265359

plot '${file}.txt' using 1:2:3 lc rgb "red" title "Results" with
    yerrorbars,\
'${file}.txt' using 1:2 lc rgb "black" notitle,\
f(x) lc rgb "blue" title "Pi"
```

```
EOF

    fi
done
```

A.4 Appendix 4

Workings for calculation of probability without using π

$$\int_0^{\frac{l}{2}} \int_0^{\frac{1}{2}} \sqrt{l^2 - 4y^2}\, dx\, dy = \int_0^{\frac{l}{2}} \sqrt{l^2 - 4y^2}\, dy$$

This can be simplified at first without integration limits by the following substitutions.

$$y = \frac{l}{2}\sin(u) \Leftrightarrow u = \arcsin\left(\frac{2y}{l}\right) \Rightarrow \frac{\partial y}{\partial u} = \frac{l}{2}\cos(u) \Leftrightarrow \partial y = \frac{l}{2}\cos(u)\,\partial u$$

We can therefore write the previous integral as follows using the definition $\cos^2(u) = \frac{1}{2}(1 + \cos(2u)$.

$$\frac{l^2}{4}\int \cos^2(u)\, du = \frac{l^2}{4}\int \frac{1}{2}(1 + \cos(2u))\, du = \frac{l^2}{8}\left(u + \frac{1}{2}\sin(2u)\right)$$

We can now insert the definition of *u* in terms of *y* into this indefinite integral and apply the integration limits that were dropped before.

$$\frac{l^2}{16}\left[2\arcsin\left(\frac{2y}{l}\right) + \sin\left(2\arcsin\left(\frac{2y}{l}\right)\right)\right]_0^{\frac{l}{2}} = \frac{l^2}{16}\left[\pi + \sin(\pi) - 0\right] = \frac{\pi l^2}{16}$$