Scientific Publications

Universal Journal of Computer Sciences and Communications, 2023, 1, 704
www.scipublications.org/journal/index.php/ujcsc
DOI: 10.31586/ujcsc.2023.704

Article

Some Software Application of the Monte Carlo Method

Ivan H. Krykun [1,*], Serhii A. Lukhverchyk [2]

[1] Department of Theory of Control Systems of Institute of Applied Mathematics and Mechanics of NAS of Ukraine, Sloviansk, Ukraine
[2] Department of Information Technologies, Vasyl' Stus Donetsk National University, Vinnytsia, Ukraine

*Correspondence: Ivan H. Krykun (iwanko@i.ua)

Abstract: We study the using the Monte Carlo method and its application. Below are several examples of software implementations of the Monte Carlo method for performing calculations that will allow us to determine the necessary information in cases where probability can be applied. Below is a software implementation of the examples in the C# programming language. The programs have a desktop interface and allow us to calculate such values as the number π and the time required to perform certain actions.

Keywords: Method Monte Carlo; C# program language; Random numbers generator; Simulation; The number π

How to cite this paper:
Krykun, I. H. ., & Lukhverchyk, S. A. (2023). Some Software Application of the Monte Carlo Method. *Universal Journal of Computer Sciences and Communications*, 1(1), 41–50. Retrieved from https://www.scipublications.com/journal/index.php/ujcsc/article/view/704

Academic Editor:
Max Robert Marinho

Received: May 21, 2023
Accepted: xx xx, 2023
Published: xx 26, 2023

1. Introduction

The Monte Carlo method is a numerical method used to solve mathematical problems by generating random numbers and statistically analyzing the results. This method is based on the idea that if a large number of random numbers are generated, it is possible to obtain an approximate value for a complex mathematical problem.

The Monte Carlo method is a simulation-based approach used in mathematics, computer science, engineering, finance, and other fields to estimate the probability of certain outcomes. It is based on running multiple simulations with different inputs to calculate the probability of certain outcomes. Simulations are typically run with random inputs and the output is derived from the average of all the simulations. The Monte Carlo method is typically used for estimating the probability of complex outcomes that have multiple variables. It can also be used to solve optimization problems with many variables [1, 2, 3, 4, 5, 6, 7, 8].

The principle of operation of the Monte Carlo method is that random numbers are generated from a known distribution, and then they are used to solve the problem. For example, if you want to calculate the integral of a function, you can generate random points in the region where the integration takes place and calculate the average value of the function at those points. This average value will be the approximate value of the integral.

The Monte Carlo method is a widely used simulation-based approach in various fields, including mathematics, computer science, engineering, finance, and others. It is especially useful for estimating the probability of complex outcomes with multiple variables, solving optimization problems, and assessing risks in financial decision-making [2, 3].

This method is based on running multiple simulations with different inputs, typically random, to calculate the probability of certain outcomes. By generating samples from a given distribution, it can approximate the expected value of a real-life situation [4]. However, some critics argue that the method's reliance on randomness makes it unreliable and unable to account for all sources of uncertainty, while others suggest that the programmer's bias can affect the results [2, 3, 4].

Modern applications of the Monte Carlo method include many different fields such as finance, materials science, biology, physics, computer graphics, and others. For example, in finance, the Monte Carlo method is used to assess risks and make investment decisions. In materials science, the Monte Carlo method is used to model the properties of materials and their interactions with other materials. In biology, the Monte Carlo method is used to model biochemical processes and interactions between biological molecules.

In addition to its applications in business, the Monte Carlo method is also used in physics and engineering to model particle interactions and understand complex systems [4]. Overall, the advantages of the Monte Carlo method outweigh its disadvantages, making it a valuable tool for risk analysis and decision-making in various fields [2, 3, 4].

In this work, we will use the Monte Carlo method to obtain an approximation of the well-known irrational number π. To do this, we will use a random number generator to generate the coordinates of random points (on a plane or in space) and, based on the geometric relationships between the geometric measures of some geometric figures (for example, areas of a square and a circle in the two-dimensional case), we will obtain an approximation of the number π.

The work is organized as follows: in following Section 2, we specify the algorithms of approximation of the number π using the Monte Carlo method. In Section 3, we present the code disclosure and explanation in a two-dimensional and in three-dimensional space. The simulation results and conclusions about the accuracy of the approximation of the number π using the obtained programs are presented in Section 4. Section 5 contains the discussion and prospects for further research. In Appendixes A and B, we give the program code in a two-dimensional and in three-dimensional space, respectively.

2. Materials and Example of the Problem

The Monte Carlo method is a statistical method used to calculate uncertain or complex mathematical problems, including the calculation of the number π.

The idea of the method is to generate random values corresponding to the input data of the problem and use them to estimate the resulting value. The more random values are generated, the more accurate the result can be.

2.1. Calculation of the number π using the Monte Carlo method

For example, to calculate π using the Monte Carlo method, you can consider a circle with a radius of 1, which is located in a square with a side of 2. The number π can be calculated as the ratio of the area of the circle to the area of the square.

To do this, it is necessary to generate random points within the square, and then check whether each point falls within the circle. Then the ratio of the number of points that fall within the circle to the total number of points will be an approximation of the ratio of the area of the circle to the area of the square. The more points are generated, the more accurate the approximation will be.

In other words, in two-dimensional space, you can calculate approximately the number π as follows:

1. Generate random coordinates of the points in a square with side 2R (where R is the radius of the circle inscribed in the square).
2. Count the number of points in a circle with radius R that lies inside the square.
3. Calculate the ratio between the number of points that fell into the circle and the total number of points generated. This ratio will be an approximation to $\pi/4$.
4. Multiply the obtained ratio by 4 to get the approximate value of the number π.

This idea of using the Monte Carlo method to calculate approximately the number π in two-dimensional space can be seen in Figure 1 below.

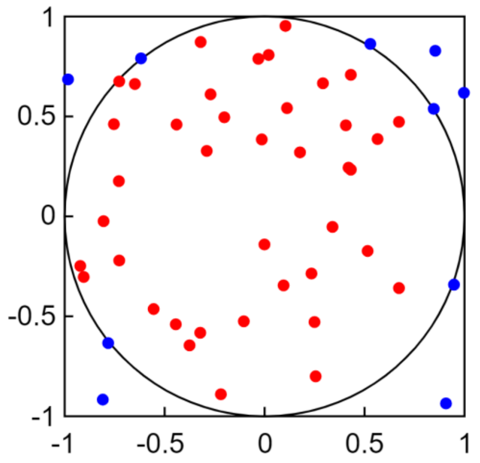


Figure 1. Points generated in a square in a two-dimensional space.

In three-dimensional space, the Monte Carlo method can be used to calculate the volume of a sphere. Here, the algorithm is as follows:

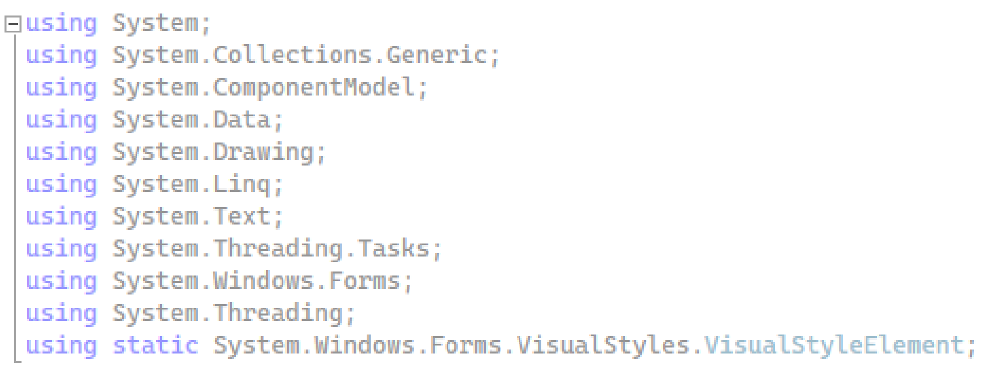
1. Generate random coordinates of points in a cube with side 2R (where R is the radius of the sphere inscribed in the cube).
2. Calculate the number of points inside the ball with radius R that lies inside the cube.
3. Calculate the ratio between the number of points that fell into the ball and the total number of points generated. This ratio is an approximation to the volume of the sphere divided by the volume of the cube, i.e. $\pi/6$.
4. Multiply the obtained ratio by 4 to get the approximate value of the number π.

The accuracy of the Monte Carlo method depends on the number of generated points. The more points there are, the more accurate the approximation will be. In two-dimensional space, the Monte Carlo method can be more accurate because it is easier to implement in practice, and also for calculations in three-dimensional space, more random points need to be generated to obtain sufficient accuracy.

However, it should be noted that the accuracy of the Monte Carlo method depends not only on the number of points, but also on the quality of their generation, i.e., the random points must be evenly distributed in space. It is also important to know that the Monte Carlo method is a statistical method, so the results may have a certain error and repeatability in the calculations.

3. Code Disclosure and Explanation

Both programs were developed in the Visual Studio environment and written in the C# programming language. Within this interface, the user has the ability to specify the number of points that will be generated to calculate the number π. (The limit on the number of points that will be generated depends on the power of the hardware on which the experiment is performed). Let's look at Figure 2 below where the key part of the first program imports the namespace that contains the classes we need to work with (working with graphics, .net functions, working with form windows, etc.)

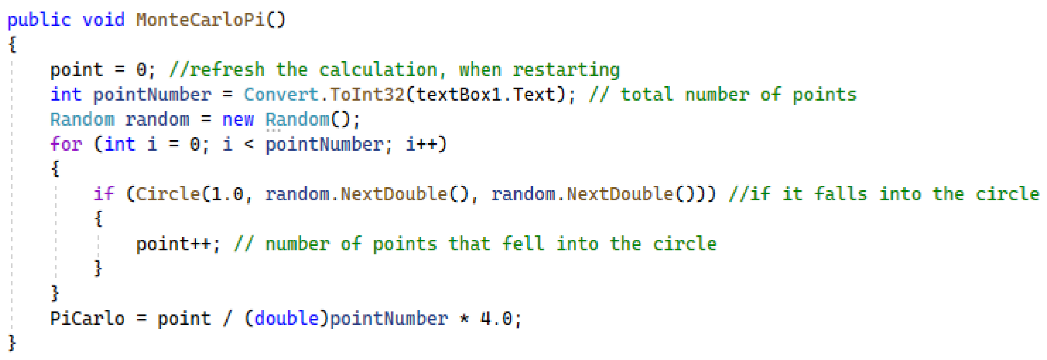


```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;
using System.Threading;
using static System.Windows.Forms.VisualStyles.VisualStyleElement;
```

Figure 2. Classes used in programs.

3.1. Code Disclosure and Explanation, two-dimensional case

Consider the basic logic of calculating π in two-dimensional space. For this purpose, a basic function called *MonteCarloPi()* was created. This function is called when user click on the "Calculate Pi" button and the code of this function in Figure 3 below.

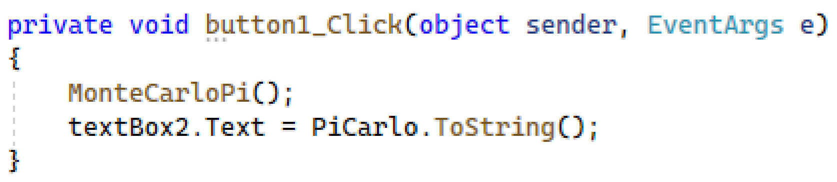


```
public void MonteCarloPi()
{
    point = 0; //refresh the calculation, when restarting
    int pointNumber = Convert.ToInt32(textBox1.Text); // total number of points
    Random random = new Random();
    for (int i = 0; i < pointNumber; i++)
    {
        if (Circle(1.0, random.NextDouble(), random.NextDouble())) //if it falls into the circle
        {
            point++; // number of points that fell into the circle
        }
    }
    PiCarlo = point / (double)pointNumber * 4.0;
}
```

Figure 3. The code of the *MonteCarloPi()* method.

This method initializes the point variable as 0 and receives the number of points that the user has entered in *textBox1*. We create random x and y coordinates for each point using the *Random* class and check if the point falls inside a circle with radius 1 using the *Circle()* function. If the point falls inside the circle, we increment the value of point. Based on this, we calculate the value of π using the Monte Carlo method and save it to the *PiCarlo* variable.

In the *button1_Click()* method (see code in Figure 4 below) we call the *MonteCarloPi()* method to calculate the value of π and display it in the *textBox2*.



```
private void button1_Click(object sender, EventArgs e)
{
    MonteCarloPi();
    textBox2.Text = PiCarlo.ToString();
}
```

Figure 4. The code of the *button1_Click()* method.

So, the program generates random points inside a square with a side of 2 units and calculates how many of these points fall inside a circle with a radius of 1 unit. It then calculates an approximate value of π using the Monte Carlo method. The more points it generates, the more accurate the approximation.

The program's visual interface can be seen in Figure 5 below and the program’s code is given in **Appendix A**.

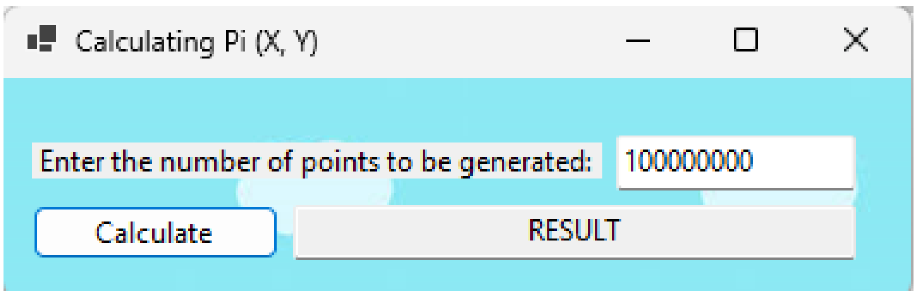


Figure 5. Screenshot of the program's visual interface.

3.2. Code Disclosure and Explanation, three-dimensional case

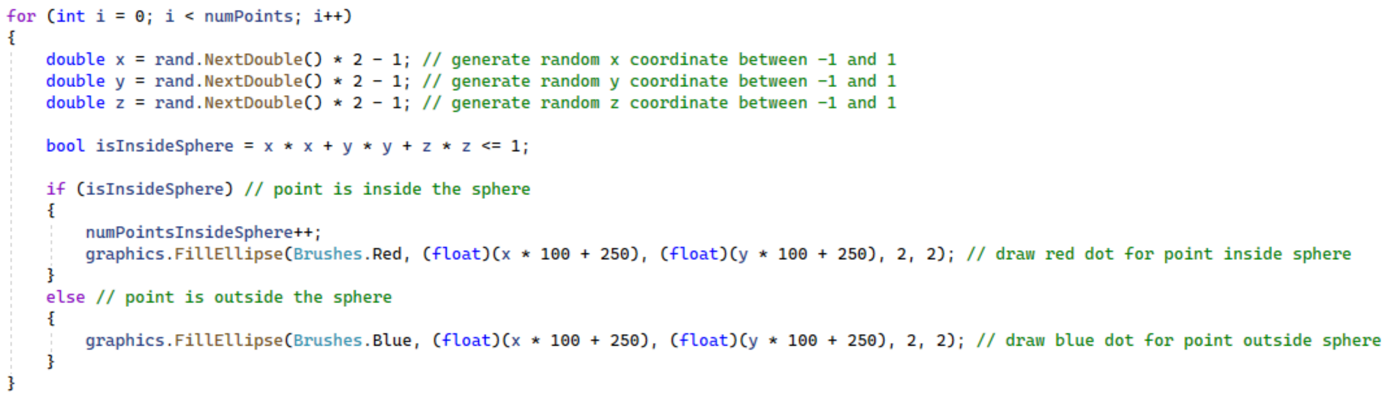
This code contains the definition of the *Form1* class, which is a Windows form. This class contains the methods and fields necessary to run a program to calculate the number π using the Monte Carlo method in three dimensions.

The first line connects the namespaces that contain the classes and other objects necessary for the program.

The *MonteCarloSimulation* method contains the logic for calculating the number π. It checks that the *NumberOfPoints* field contains a value, and then determines the number of points to be generated. The *Random* object is used to generate random numbers.

Next, a new form is created to visualize the results. It will contain a *PictureBox* that will be used to display the points on the graphical plane. The form and the *PictureBox* are initialized and configured.

The loop from 17-27 is executed *numPoints* times. In each repetition, it generates random coordinates for x, y, and z in the range -1 to 1. After generating the coordinates, it checks if the point falls within the sphere of radius 1 located at the centre of the coordinates. If the point falls inside the sphere, the number of points inside the sphere increases. The points are displayed on the graphical interface, with red circles indicating points inside the sphere and blue circles indicating points outside the sphere. The code of this procedure one can see in Figure 6 below.

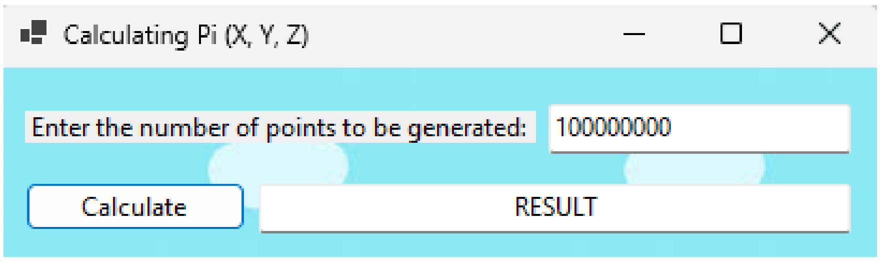


```
for (int i = 0; i < numPoints; i++)
{
    double x = rand.NextDouble() * 2 - 1; // generate random x coordinate between -1 and 1
    double y = rand.NextDouble() * 2 - 1; // generate random y coordinate between -1 and 1
    double z = rand.NextDouble() * 2 - 1; // generate random z coordinate between -1 and 1

    bool isInsideSphere = x * x + y * y + z * z <= 1;

    if (isInsideSphere) // point is inside the sphere
    {
        numPointsInsideSphere++;
        graphics.FillEllipse(Brushes.Red, (float)(x * 100 + 250), (float)(y * 100 + 250), 2, 2); // draw red dot for point inside sphere
    }
    else // point is outside the sphere
    {
        graphics.FillEllipse(Brushes.Blue, (float)(x * 100 + 250), (float)(y * 100 + 250), 2, 2); // draw blue dot for point outside sphere
    }
}
```

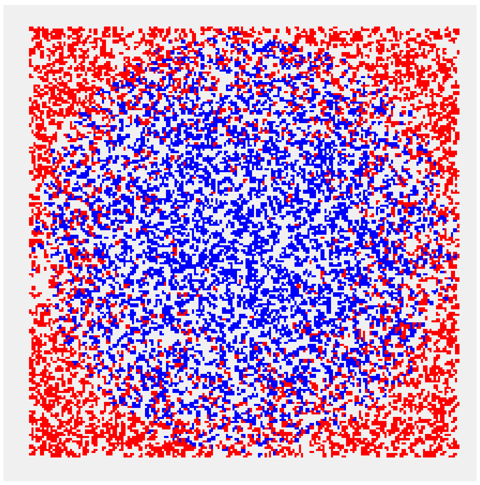
Figure 6. The point generation code in a three-dimensional case.

At the end of the cycle, the result is displayed in the *Result* field. This is the approximate value of π, calculated by the formula 6.0 * *numPointsInsideSphere*

The program code is given in **Appendix B**.


Calculating Pi (X, Y, Z)

Enter the number of points to be generated: 100000000



(a) (b)

Figure 7. Screenshot (a) shows the program interface with the ability to specify the number of points to be generated in the sphere and view the result. Screenshot (b) shows a visualization of the point generation in three-dimensional case.

4. Simulation results and conclusions

Using these two programs, we were able to conduct the following series of experiments. We perform 3 sets of data for each two-dimensional and three-dimensional case and calculate an approximate value of the number π according to algorithms in Section 2.

Remark 1. *The computational times in the tables below depend on the configuration of the used computer and the computational times for datasets, described in Tables 2 and 3 are similar to those shown in Table 1.*

Table 1. Table with the results of the first set of experiments.

Number of experiments	Two-dimensional space		Three-dimensional space	
	Approximate value of the number π	Computational time	Approximate value of the number π	Computational time
10`000	3.1196	< 5 seconds	3.1464	< 5 seconds
100`000	3.14028	~ 10 seconds	3.11538	~ 10 seconds
1`000`000	3.140748	~ 20 seconds	3.138192	~ 20 seconds
10`000`000	3.1412468	~ 1 minute	3.1419486	~ 1,5 minute
100`000`000	3.14185140	~ 2 minutes	3.14164384	~ 4 minutes
1`000`000`000	3.141632568	~ 12 minutes	3.141545876	~ 20 minutes

Table 2. Table with the results of the second set of experiments.

Number of experiments	Two-dimensional space	Three-dimensional space
10`000	3.1244	3.1350
100`000	3.14476	3.12768
1`000`000	3.141912	3.138702
10`000`000	3.1425212	3.1416354
100`000`000	3.14161776	3.14132576
1`000`000`000	3.141614468	3.141560344

Table 3. Table with the results of the third set of experiments.

Number of experiments	Two-dimensional space	Three-dimensional space
10`000	3.1348	3.1488
100`000	3.13688	3.13818
1`000`000	3.140808	3.137226
10`000`000	3.1425692	3.1410996
100`000`000	3.14172180	3.14166448
1`000`000`000	3.141525860	3.141594820

In addition, for this experiment, it is important to calculate the relative errors based on the results. To calculate the relative errors, we need to subtract the exact value of the number π (i.e. 3.141592653589793...) from the obtained experimental result and divide it by the exact value of the number π.

Table 4. Table of relative errors, two-dimensional calculations.

Number of experiments	First set	Second set	Third set
10`000	0.70%	0.55%	0.22%
100`000	0.04%	0.10%	0.15%
1`000`000	0.027%	0.01%	0.02%
10`000`000	0.011%	0.03%	0.03%
100`000`000	0.008%	0.001%	0.004%
1`000`000`000	0.0013%	0.0007%	0.002%

Table 5. Table of relative errors, three-dimensional calculations.

Number of experiments	First set	Second set	Third set
10`000	0.15%	0.21%	0.23%
100`000	0.83%	0.44%	0.11%
1`000`000	0.11%	0.09%	0.14%
10`000`000	0.011%	0.001%	0.02%
100`000`000	0.0016%	0.008%	0.002%
1`000`000`000	0.0015%	0.001%	0.00007%

Based on the results of the experiments presented in Tables 1-3 it appears that the two programs used for calculating π provide consistent results with increasing accuracy as the number of experiments increases. The relative errors for both the two-dimensional and three-dimensional calculations decrease as the number of experiments increases, with the third series of experiments yielding the most accurate results.

However, it is important to note that even the most accurate results still have some degree of error when compared to the exact value of π. The relative errors are generally quite small, but still significant when precision is critical. Therefore, it is necessary to consider the accuracy required for a particular application and to use appropriate methods for calculating the number π to achieve the desired level of precision.

5. Discussion and prospects for further research

The Monte Carlo method has become a crucial tool for analyzing complex probabilistic phenomena in science and technology, allowing for estimation of various characteristics of random processes. Further research on the method can improve its accuracy through the use of new methods for generating random numbers and computer system advancements.

Additionally, applying the Monte Carlo method to fields such as finance, biomedicine, statistics, and computer science holds great promise. Despite limitations, the method's potential for development and application is significant, and its capabilities are expanding with the increasing power of computer systems.

The question of using the results mentioned in the paper, in particular, in the modelling of random and natural processes will be enhanced using results and methods of recent studies of applications of random and natural processes [9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19] and computer technologies [20, 21].

Author Contributions: All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Acknowledgments: The authors express their heartfelt gratitude to the brave soldiers of the Ukrainian Armed Forces who protect the lives of the authors and their families from Russian bloody murderers since 2014.

Conflicts of Interest: The authors declare no conflict of interest.

Appendix A. Code of the program, two-dimensional case

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;
using System.Threading;
using static System.Windows.Forms.VisualStyles.VisualStyleElement;

namespace MC_Number_Pi
{
    public partial class Form1 : Form
    {
        public double PiCarlo; //pi
        private int point; //the number of points that fell into the circle
        public double PiCarlo1;

        public Form1()
        {
            InitializeComponent();
        }

        public void MonteCarloPi()
        {
            point = 0; //refresh the calculation, when restarting
            int pointNumber = Convert.ToInt32(textBox1.Text); // total number of points
            Random random = new Random();
            for (int i = 0; i < pointNumber; i++)
            {
                if (Circle(1.0, random.NextDouble(), random.NextDouble())) //if it falls into the circle
                {
                    point++; // number of points that fell into the circle
                }
            }
            PiCarlo = point / (double)pointNumber * 4.0;
        }
        static bool Circle(double radius, double x, double y) //checking whether the point falls within the circle
        {
            return (Math.Sqrt(x * x + y * y) < radius);
        }

        private void button1_Click(object sender, EventArgs e)
        {
            MonteCarloPi();
            textBox2.Text = PiCarlo.ToString();
        }
    }
}
```

Appendix B. Code of the program, three-dimensional case

```
namespace MonteCarloPi2
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }

        public void MonteCarloSimulation()
        {
            // checking that the NumberOfPoints field has a value
            if (!string.IsNullOrEmpty(NumberOfPoints.Text))
            {
                int numPoints = Convert.ToInt32(NumberOfPoints.Text);
                Random rand = new Random();
                int numPointsInsideSphere = 0;

                // create form to display visualization
                Form visualizationForm = new Form();
                visualizationForm.ClientSize = new Size(500, 500);

                // create drawing surface to display points
                PictureBox pictureBox = new PictureBox();
                pictureBox.Dock = DockStyle.Fill;
                visualizationForm.Controls.Add(pictureBox);

                // create bitmap to draw points on
                Bitmap bitmap = new Bitmap(pictureBox.Width, pictureBox.Height);
                Graphics graphics = Graphics.FromImage(bitmap);

                for (int i = 0; i < numPoints; i++)
                {
                    double x = rand.NextDouble() * 2 - 1; // generate random x coordinate between -1 and 1
                    double y = rand.NextDouble() * 2 - 1; // generate random y coordinate between -1 and 1
                    double z = rand.NextDouble() * 2 - 1; // generate random z coordinate between -1 and 1

                    bool isInsideSphere = x * x + y * y + z * z <= 1;

                    if (isInsideSphere) // point is inside the sphere
                    {
                        numPointsInsideSphere++;
                        graphics.FillEllipse(Brushes.Red, (float)(x * 100 + 250), (float)(y * 100 + 250), 2, 2); // draw
red dot for point inside sphere
                    }
                    else // point is outside the sphere
                    {
                        graphics.FillEllipse(Brushes.Blue, (float)(x * 100 + 250), (float)(y * 100 + 250), 2, 2); // draw
blue dot for point outside sphere
                    }
                }

                pictureBox.Image = bitmap; // display bitmap on picture box
```

```
                    double piEstimate = 6.0 * numPointsInsideSphere / numPoints;

                    Result.Text = ($"Estimated value of Pi: {piEstimate}");
                    //visualizationForm.FormClosed += (s, args) => Application.Exit();

                    visualizationForm.ShowDialog(); // display form as modal dialog
                }
            }

            private void CalculateButton_Click(object sender, EventArgse)
            {
                MonteCarloSimulation();
            }
        }
    }
```

References

[1] Gamerman, D. *Monte Carlo Statistical Methods*. Springer, 2006.

[2] McKinney, W. *Python for Data Analysis.* O'Reilly Media: Sebastopol, CA, USA, 2012.

[3] Zivot, E. *Introduction to Computational Finance and Financial Econometrics*. Chapman and Hall/CRC, 2018.

[4] Pang, T. *An Introduction to Computational Physics*. Cambridge University Press, 2015, Chapter 10 "Monte Carlo Method in Quantum Systems Computing."

[5] Glasserman, P. *Monte Carlo Methods in Finance*. Springer, 2004.

[6] Newman, M.E.J. & Barkema, G.T. *Monte Carlo Simulation Methods in Statistical Physics*. Oxford University Press, 1999.

[7] Fishman, G. *Monte Carlo: Concepts, Algorithms and Applications*. Springer, 1996.

[8] Keller, A. & Heinrich, S. (Eds.). *Monte Carlo and Quasi-Monte Carlo Methods*. Springer, 2008.

[9] Krykun, I.H. Limit theorem for solution of stochastic equations with local time. *Proceedings of Institute of Applied Mathematics and Mechanics of NAS of Ukraine* **2005**, *Vol. 10,* pp. 110–120. (*in Ukrainian)*

[10] Krykun, I.H. Large deviation principle for stochastic equations with local time. *Theory of Stochastic Processes* **2009**, *Vol. 15(31), No. 2.* pp. 140–155.

[11] Krykun, I.H.; Makhno, S.Ya. The Peano phenomenon for Itô equations. *Journal of Mathematical Sciences* **2013**, *Vol. 192, Issue 4*, pp. 441–458. DOI: 10.1007/s10958-013-1407-5

[12] Krykun, I.H. Functional law of the iterated logarithm type for a skew Brownian motion. *Theory of Probability and Mathematical Statistics* **2013**, *Vol. 87*, pp. 79–98. DOI: 10.1090/S0094-9000-2014-00906-0

[13] Krykun, I.H. Convergence of skew Brownian motions with local times at several points that are contracted into a single one. *Journal of Mathematical Sciences* **2017**, *Vol. 221, Issue 5*, pp. 671–678. DOI: 10.1007/s10958-017-3258-y

[14] Krykun, I.H. The Arc-Sine Laws for the Skew Brownian Motion and Their Interpretation. *Journal of Applied Mathematics and Physics* **2018**, *Vol. 6, No. 2*, pp. 347–357. DOI: 10.4236/jamp.2018.62033

[15] Krykun, I.H. The Arctangent Regression and the Estimation of Parameters of the Cauchy Distribution. *Journal of Mathematical Sciences* **2020**, *Vol. 249, Issue 5*, pp. 739–753. DOI: 10.1007/s10958-020-04970-3

[16] Krykun, I.H. New Approach to Statistical Analysis of Election Results. *International Journal of Mathematical, Engineering, Biological and Applied Computing* **2022**, *1, No. 2*, pp. 68–76. DOI: 10.31586/ijmebac.2022.466

[17] Krykun, I.H. The Black-Scholes Exotic Barrier Option Pricing Formula. *Journal of Mathematics Letters* **2023**, *Vol. 1, No. 1*, pp. 10–18. DOI: 10.31586/jml.2023.604

[18] Krykun, I.H.; Pavlov, M.S. Statistics of Electoral Systems and Methods of Election Manipulation. *Journal of Social Mathematical & Human Engineering Sciences* **2023**, Vol. 1, No. 1, pp. 11–21. DOI: 10.31586/jsmhes.2023.610

[19] Krykun, I.H. On weak convergence of stochastic differential equations with irregular coefficients. *Journal of Mathematical Sciences* **2023**, *Vol. 273, Issue 3*, pp. 398–413. DOI: 10.1007/ s10958-023-06506-x

[20] Cherniichuk, H.P.; Krykun, I.H. Notes about Winning Strategies for Some Combinatorial Games. *Journal of Mathematics Letters* **2022**, *Vol. 1, No. 1*, pp. 1–9. DOI: 10.31586/jml.2022.496

[21] Bilous, R.V.; Krykun, I.H. A Problem of Accuracy of Computer Calculations. *Universal Journal of Computer Sciences and Communications* **2022**, *Vol. 1, No. 1*, pp. 35–40. DOI: 10.31586/ujcsc.2022.531