1 Purpose

This lab illustrates the use of looping structures by introducing a class of programming problems called numerical algorithms.

1. Practice the use of the C++ repetition constructs of `for`, `while`, and `do-while`.
2. Use computer-generated random numbers.
3. Explore the solution to several numerical problems.

2 Introduction

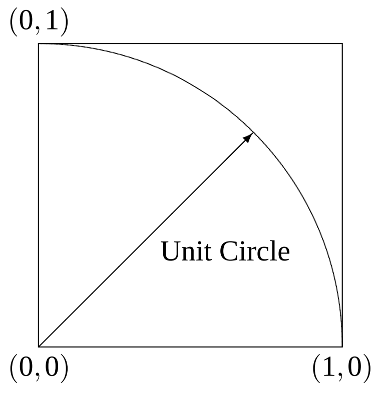
In this lab, you will write a program to estimate the value of π and a second program to the find the roots of an equation.

Create a `Lab9` directory and do your work there.

3 Monte Carlo Simulation

Your task in this part of the lab is to write a program that uses a monte carlo simulation (named after the famous casinos of Monte Carlo) to estimate the value of π. To do so you will use a probabilistic method that relies on random numbers.

This is how it works. Suppose that we have a square with sides of length 1.0, and the upper right quarter of a circle inside the square with a radius of 1. The area of a circle is πr^2. Since we have only a quarter of a circle and $r = 1$, the area inside the quarter circle is $\pi/4$.



The idea is to throw random darts at the square and then to count the darts that fall inside the circle, and the total number of darts. Given that the darts are produced in a truly random fashion, i.e., with uniform distribution, the probability of a dart falling in the circle is the ratio of the number of darts in the circle to the total number of darts:

$$\text{probability in circle} = \frac{\text{\# in circle}}{\text{total \#}}$$

This probability, however, is also equal to the ratio of the area of the circle to the area of the square. Since the square has an area of 1, this ratio is $\pi/4$. So solving for π we get the following:

$$\text{probability in circle} = \frac{\pi}{4}$$
$$\pi = 4 \times \text{probability in circle}$$

That is,

$$\pi = 4 \times \frac{\text{\# in circle}}{\text{total \#}}$$

So this is how we estimate π using random numbers.

Now how do we do this in a program? In the `~cs203/Labs/Lab9` directory you will find a file called `getRandom.h`. Include this file into your program. Look at the file `getRandom.h` and read the comments. To generate a random number simply call the function `initRandom()` *once* to initialize the random numbers and then call the function `getRandom()` each time you need a random number. For example,

```
x = getRandom();
```

This function will return a random double in $[0.0, 1.0)$ where the "`[`" means it includes 0.0 and the "`)`" means it does not include 1.0. Each point where a dart hits will require two random numbers: one for the x value of the point, and one for the y value of the point. We are generating points in the first quadrant of the circle only because x and y are always between 0.0 and 1.0.

To compute your estimate of π, you will need to count up the number of darts and the number of darts that fall within the circle. Points within the circle will have a distance from the origin that is less than 1; that is,

$$\sqrt{x^2 + y^2} < 1.0$$

How many darts to generate? Start by throwing 100 darts.

After you write your program, you should carry out three experimental test runs on your program with 100, 1000 and 10,000 darts. This will give you some idea of how much the estimates can vary. Put the results of three runs and a listing of your working program in your `handin.txt` file.

4 Finding the Roots of a Function

Many mathematical problems can be solved exactly by applying a procedure or algorithm. For example, there is an explicit formula for computing the roots of a quadratic equation. Likewise, a formula exists for computing the roots of cubic polynomials (those with degree 3). But for many problems (e.g., polynomials of degree 5 and higher), no such explicit formula exists, and methods to compute an approximate value must be used.

Iterative methods are approximation methods which often lend themselves to implementation on computers. In an iterative method one begins with a guess of the solution, then using that guess, computes a better approximation of the solution. If the method is well-designed and the guess is reasonable, the new approximation is closer to the solution than the original guess. The method of computing a better approximation can be applied to our new approximation in turn, producing an even better approximation. This approach can be repeated (iterated) until the approximation is sufficiently close to the solution. Note that it is not always possible to get an exact solution using an iterative method; often the improvement method moves the approximation part way, but not all the way, to the answer. As a result, termination of the iteration is often based on getting an approximation of the error less than some threshold.

The method you will use today is known as *Newton's method* and is used to calculate the roots of functions. You will apply it to a quadratic equation; this will allow you to check the solutions, and will also prevent you from running into some pitfalls of the method that are beyond the scope of what we can cover today.

4.1 Basic Step

The basic step in Newton's method involves calculating a new value for the approximation. Let f be the function for which we are trying to find a root. We will use a simple quadratic equation as an example.

$$f(x) = x^2 + x - 2$$

Since this equation can be factored as $(x-1)(x+2)$, it has roots at 1 and -2. Let x be the variable holding the approximation, and let $f'(x) = 2x+1$ be the derivative of f. The new value of the approximation x_{i+1} is computed using the following:

$$x_{i+1} = g(x_i) = x_i - \frac{f(x_i)}{f'(x_i)}$$

Here is a table of values for f and g.

Iteration	$i = 1$	$i = 2$	$i = 3$
x_i	2	1.2	1.011765
$f(x_i)$	4	0.64	0.035433
$g(x_i)$	1.2	1.011765	1.000046

In the table above, the 2 in the first column is the initial guess for a root. This value is used to generate the next approximation (of a root), 1.2. Then 1.2 appears at the top of the next column and is used to produce the next approximation. This is the process your program should produce. Notice that as the new approximations are generated that the values computed for $f(x_i)$ get closer to 0.

Write C++ functions for g and f. You may either write f' as an function or just write its definition where it belongs in g. Make sure that both functions take the proper argument and return a value of the proper type. Test your program by printing the value of g applied to a few values. Note that $g(1) = 1$ and $g(-2) = -2$. Also check the value of $g(0)$; what should it be?

4.2 Input

The program also needs input from the user, who must supply both an initial guess x and an error bound e; both values are doubles. Write a function `getGuess` to read these two values and assign them to x and e. Use input validation to make sure that the value for e is between 0.0 and 0.1. **Hint:** You need to set two values from a single function. What sort of parameters should you use?

4.3 Stopping the Iteration

We stop the iteration when the approximation is close enough to the solution. Since we are looking for a root, we check for $f(x_{i+1})$ to be close to zero or less than the parameter e;

$$|f(x_{i+1})| < e$$

The bars mean to take the absolute value. You may use the absolute value function `fabs()` which is in the `cmath` library.

4.4 Writing the Program

Now you are ready to write the program to approximate roots. The steps are as follows:

- Read in the values for `x` and `e`.
- Iterate (i.e., **repeat**) the process of setting `x` to `g(x)` until the absolute value of `f(x)` is less than `e`. You should include a counter to keep track of how many iterations are required until the program terminates. **Optional:** It is also useful to include the counter in the loop condition: If something goes wrong and the number of iterations exceeds say 10000, then you may as well terminate the loop and print an error message.
- Once the loop terminates, print out the approximation of the root and the number of iterations required as part of a suitable message. (If you did the **optional** part above, this is the place that you can determine whether the threshold was exceeded or not, and print a suitable message.)

Once you get your program working, try the initial guess 2 with the error thresholds 0.1, 0.01, and 0.0001. Hand in the results from these three runs. Now try 3 different initial guesses with the error threshold 0.01; see which root each converges to. Hand in the results of these runs as well, and print your program too.

5 What To Hand In

- Monte Carlo Simulation: Program and results of 3 test runs.
- Finding Roots: Program and results of 6 test runs for the specified inputs.