**Rust Programming Lab #2 9th August 2022**

**Rust by example** **- Learn by trying**

Since this lab will also precede any formal lecture, we will continue to learn by trying!

An interesting mathematical curiosity is that you can calculate **p** to arbitrary precision using a (good) random number generator.

The area of a circle with radius, ***r***, is ***A*** = **p*r***2. So a unit circle (***r***=1) has area, **p**, and one quadrant has area **p**/4.

If you generate ***n*** ***random*** points, within a unit square aligned with the ***x*** and ***y*** axes and based at (0,0), and count the number of points that lie ***inside*** the unit circle on (0,0), ***nin***, then

***nin*** / ***n*** = area inside the unit circle / area of the unit square = **p** / 4 / 1

Given the outline code, pi\_outline.rs, add code to count the number of points *inside* a unit circle. The outline will generate an (***x***,***y***) pair where ***x*** and ***y*** are random numbers in (0,1). The distance of this point from the origin, ***d*** = √(***x***2 + ***y***2). The observant among you will have realized that, if ***r* = 1**, then √***r*** = 1 too 😉 – to simplify your code a little. Using that count, you can estimate **p**. Rust has a library constant, **core::f64::consts::PI** ; you can use it to see whether this approach to evaluating **p** actually works!

## Random number generators

Random numbers have many uses in solving real problems, from initializing your neural net, to simulations, determining whether you can win at that new casino game *or* making sure that your virtual reality game doesn’t (boringly) start at the same point every time! An apparently simple problem has actually been the subject of intense research over many years: the problem is to find a ‘truly’ random number generator that never repeats the sequence of numbers generated. A common strategy generates a series, **X1, X2,** …., given a seed **X0** : **0** < **X0** < ***m***, frpm:

**Xn = (a \* Xn-1 + b) % m**

**where**  **m > 0** (the modulus is positive), **0 < a < m** (the multiplier is positive but less than the modulus) and **0** ≤ **b < m.** However, this is a **pseudo-random** generator: after a while, it will repeat itself, *i.e.* it is not truly random. For the purpose of this exercise, you can assume that the random number generator in the std Rust library is a reasonable one.

**Exercise 1**

When using a random number generator, it is usually necessary to generate a very large number of points before the calculation converges on the target. So you will need to adjust your program to generate a large number of points, before you see a reasonable answer. Print out the current estimate of **p** at suitable intervals (try some large numbers!) and the difference with the ‘true’ value, showing that the calculated value actually does approach **p**, but maybe not very fast 😉.

In this case, the value of **p** depends on the quality of the random number generator (as do many calculations relying on truly random behaviour!). Think how you might check that the numbers generated are truly random – or you could search the literature to find the many mathematical papers that discuss true random numbers!

**Exercise 2**

One simple test for ‘randomness’ would suggest the mean of numbers in (0,1) would approach 0.5 - for a sufficiently large number. Add this test to your code, *i.e.* sum the generated number(s) and compute the mean(s) of the sum(s) so generated. How far did the mean(s) deviate from 0.5? Print them out at suitable intervals too. Set a tolerance target, e.g. **EPS**, is the mean acceptable?

**Exercise 3**

There are other ways to write loops in Rust. For a problem that is expected to converge to an answer, you might use an infinite loop and exit when your result meets your target. Change your program to use the **loop { … }** and insert a **break** when your answer is acceptable. Hint: initially set your **EPS** value to be quite large, e.g. 0.01, to check that your program is running correctly, *i.e.* stopping in some reasonable time. Then change EPS to be a more interesting target, e.g. 0.001, and see how many iterations it takes to reach the target. The curious might try setting EPS much lower, e.g. 10-5, but be prepared to leave your computer running overnight 😊.

From the previous lab exercise, you saw:

1. Making a loop – the for command will execute the loop 100 times
2. Declaring variables with let - x. sqrt\_x, xx, dx.
3. Setting variable types – f64 is a 64 bit floating point value
4. Calling library functions – sqrt() - to calculate a square root.
5. Adding an if .. else .. statement
6. Code blocks in braces – { …. }
7. Basic print commands – print! and println!
8. Converting variables from one type to another .. as converts the integer n to an f64 type

Now you can add,

1. Library constants – there are similar constants, for ***e*** (as **E**), 1/**p** (as **FRAC\_1\_PI**), *etc.*
2. Declaring mutable (changeable) variables – **mut** keyword
3. Another looping command - **loop**
4. Adding library functions – here the random number generator and **abs**
5. Two ways to use a std library function – **abs** (compare with **sqrt** in lab 1)
6. Reminder that Rust does not allow ‘mixed mode’ expressions –
you must **cast** (‘**as**’ operator) to the same type as the remainder of the expression first
7. Constants for tolerances (acceptable variations in floating point numbers) - **EPS**

**Fill in a sample of your results in the attendance record and hand it in.**

|  |  |  |  |
| --- | --- | --- | --- |
| **Attendance**  | **01286120**  | **Elementary Systems Programming** | **9 Aug 2022** |

|  |  |  |
| --- | --- | --- |
| **Name (Thai script\*)** |  | **Student ID** |
| **(Transliteration as you enrolled)** |  |
| **\****Please write clearly: practice for one farang who is trying to improve* **😉** |

|  |  |  |
| --- | --- | --- |
| Exercise 1 | Iterations | Value of p |
|  |  |  |
|  |  |
|  |  |
| Exercise 2 | Iteration # | Mean | Tolerance |
|  |  |  |  |
|  |  |  |
|  |  |  |
| Exercise 3 | Tolerance set | # iterations to reach target |
|  |  |  |
|  |  |