Nikolas Poholik

Dr. Mitchell

ECE 3217 - Computer Architecture

March 22, 2024

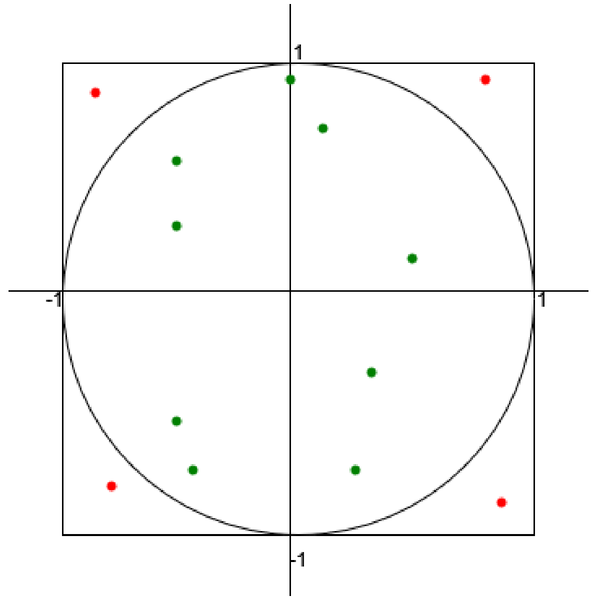
The Monte Carlo Method: Estimating Pi

The goal of this project was to take advantage of a Linux System call, which found 64 bit random numbers, and work towards establishing a Monte Carlo simulation to estimate Pi. Now, the assembler code to normalize the random numbers obtained between values of -1 and 1 was provided at the start of this project. However, once numbers are established on this range, it is possible to take the area of the square it describes and compare it to the ratio of the areas of each shape. Figure 1 showcases visually what this might look like, where the red dots represent points randomly selected that appear outside of the circle and green dots represent points randomly selected that appear inside the circle. If we simply define the areas of the square in equation 1, and the circle in equation 2, then it should be fair to say that given sufficient points generated within the defined domain and range, that the ratio of areas should be approximately equal to the ratio of the points inside the circle (area of circle) over the total points generated (area of the square). Equation 3 shows the simplification of this.

Equations

1. $Area\ (Square)\ =\ L\ \times H\ = 2 \times 2 = 4$
2. $Area\ (Unit\ Circle)\ =\ \pi$
3. $\frac{Area(Circle)}{Area(Square)} = \frac{\pi}{4} = \frac{Points\ Within\ Circle}{Points\ within\ Square\ (Total\ Points)} \Rightarrow \pi\ =\ 4\ \times \frac{Points\ within\ Circle}{Points\ within\ Square}$

Figure 1: Visualization of Monte Carlo Simulation



Now, with the basis of the method defined and the random numbers available, the remaining portion of this project involved programming it in assembler. This involved modifying the random number call slightly so that it could have a variable amount of iterations (100, 10,000, and 1,000,000) as well as call for two values, an x and y. Next, floating point operations were introduced to first calculate $x^2 + y^2$. If this value was less than or equal to 1, it is contained within the unit circle and a counter to track this is incremented. Regardless of if the point is within the circle or not, a different counter is incremented to all the points generated throughout the loop. Finally, the simplified version of equation 3 is handled utilizing floating point operations after the loop terminates, and a call to the C library is made to print the value found to the terminal.

This method was found to be fairly close to a decent approximation. It was never able to consistently get more than 2-3 digits of Pi, however. One of the main issues considered for this is that the random number generator likely has a fair amount of bias involved when selecting numbers, and this skews the simulation by a noticeable margin. Results printed to the terminal can be seen in table 1 of the appendix.

Appendix:

[1] Table 1:

	Value of Pi		
Trial Amount	**First Run**	**Second Run**	**Third Run**
100	3.040000	3.280000	3.320000
10,000	3.132400	3.146800	3.139600
100,000	3.140400	3.139080	3.141080
1,000,000	3.139136	3.145936	3.138492

[2] Monte Carlo Simulation Code:

```
// Nikolas Poholik
// 3/22/24
//-------------------------------------------------------------------------------
.text
.global _start
//--------------------------------------------------------------------------------
_start:
//------------------------------------------------
        // One Hundred Point Trial:
        // movz x0, #100, lsl #0
        //-------------------------
        // Ten Thousand Point Trial:
        // movz x0, #10000, lsl #0
        //-------------------------
        // One Hundred Thousand Point Trial:
        // movz X0, 0x86A0, lsl #0
        // movk X0, 0x0001, lsl #16
        //-------------------------
        // One Million Point Trial:
        movz X0, 0x4240, lsl #0
        movk X0, 0x000F, lsl #16
        //-------------------------
//------------------------------------------------
        ldr x8, =count
        str x0, [x8, 0]

        movz x0, #65535, lsl #0 // build a constant that is 7FFF FFFF FFFF FFFF
        movk x0, #65535, lsl #16
        movk x0, #65535, lsl #32
        movk x0, #32767, lsl #48
        scvtf d1, x0
        ldr x0, =divisor
        str d1, [x0, 0]   // Store this constant in the divisor variable
// Process given to generate random #'s
// Define the counter for total points (square) and points within circle (unit circle)
mov x14, #0 // x14 = # of total points
mov x15, #0 // x15 = # of points inside circle
repeat:
        mov x8, #278 // Setup for Syscall 278 - getrandom
        ldr x0, =var // buffer address
        mov x1, #8 // 8 bytes of randomness
        mov x2, #0 // flags ?
        svc #0

        ldr x8, =var // get address of variables
        ldr x9, [x8, 0] // load random number
        scvtf d0, x9 // convert to double percinsion
        str d0, [x8, 8] // store double percision

        ldr x0, =divisor // Load divisor from ram
        ldr d1, [x0, 0]
        fdiv d0, d0, d1 // make random number be between -1 and 1
```

```
        fmul d3, d0, d0 // get x^2

        mov x8, #278 // Setup for Syscall 278 - getrandom
        ldr x0, =var // buffer address
        mov x1, #8 // 8 bytes of randomness
        mov x2, #0 // flags ?
        svc #0

        ldr x8, =var // get address of variables
        ldr x9, [x8, 0] // load random number
        scvtf d0, x9 // convert to double percinsion
        str d0, [x8, 8] // store double percision

        ldr x0, =divisor // Load divisor from ram
        ldr d1, [x0, 0]
        fdiv d0, d0, d1 // make random number be between -1 and 1

        fmul d4, d0, d0 // get y^2

        fadd d5, d3, d4 // get x^2 + y^2

        mov x12, #1
        scvtf d6, x12 // convert 1 to floating point

        add x14, x14, x12 //increment total num of points by 1

        fcmp d5, d6
        b.gt skip

        add x15, x15, x12 // increment points within circle by 1
skip:
        ldr x8, =count // decrement the repeat counter
        ldr x10, [x8, 0]
        sub x10, x10, #1
        str x10, [x8, 0]
        cbz x10, _calculate
        b repeat
_calculate:
        mov x1, #4 // move 4 into a reg
        scvtf d0, x1 // convert 4 to floating point
        scvtf d2, x15 // convert # of points in circle to floating point
        scvtf d3, x14 // convert # of points outside circle to floating point
        fmul d0, d0, d2 // do 4 * # points in circle
        fdiv d0, d0, d3 // final calculations of pi ( 4 * # points in circle / # total points)

        // Print value
        ldr x0, =string
        ldr x8, =var
        str d0, [x8, 0]
        bl printf
// Exit code:
_exit:
        mov x8, #94
        mov x0, #0
        svc #0
//-----------------------------------------------------------------------------
.data
//string:
//.asciz "Num: %d: %lld %lf\n"
//.bss // variable
string:
.asciz "Pi: %lf\n"
var:
        .zero 8
        .zero 8
count:
        .zero 8
divisor:
        .zero 8
```