Bruno Palomar Alsina, NIU 1528253

Lab 'Monte Carlo Method' report – 22_02_24

Monte Carlo example:

1) What is the main difference between the way the pi_MC.py and pi_markov_chain.py programs work?

The first program makes use of the statistical relation between the amount of droplets that randomly fall or appear inside a radius 1 circle contained within a square of side = 2r and the amount of them that fall outside the circle but inside the square. That way, since the droplets positions are completely random, the probability that they appear inside the circle is: [circle area(which equals PI) / square area (which equals 4)]. As the number of spawned droplets approaches infinite, the ratio between droplets within the circle and within the square will eventually be true to the aforementioned probability, and multiplying it by 4 will give us **PI**.

However, the second program is slightly different; instead of finding a circle/square relation by making droplets randomly appear, a single spawn is produced at 't=0' and each iteration of the execution consists of this original droplet making a random step in both *x* and *y* directions (with the exception case of the step being 'null' when it would mean exiting the square. A step of 0,0 is then registered). After a significant amount of steps/jumps, the covered area should resemble the distribution produced in the other program, and the final calculation remains the same: 4 times the relation of steps where r' is inside the circle and the total amount of steps.

I am still not sure whether one of these programs has an advantage over the other. If the advantage comes with execution time, the difference must reside in the numpy.random.uniform() method and a simple variable modification (x = x + delta) execution times, since eventually the plotted lists are the same in both scenarios.

2) What is the effect of delta in the Markov chain calculation of PI?

The size of the delta steps clearly affects the surface coverage distribution. However, the observed behavior does not seem to be linear. For a delta of 0.05 (very small) it is harder to uniformly cover the whole surface (an infinite number of iterations shall be used for that to happen) and that leads to 'non true randomness?' in the distribution. While in that sense, it seems like we want as big steps as possible to achieve randomness, I think that is only true for infinitely big surfaces and in this case a big enough delta will result in a lot of those mentioned 'null' steps even though the distribution is more uniform. Overall, a balance is needed. With enough time, a program that compares the value of PI as a function of both the number of iterations and delta size could be developed. As far as I have observed, the recommended value of 0.3 works best.

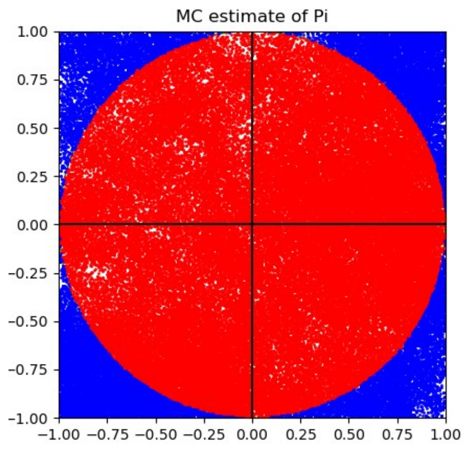


Figure 1: Resulting scatter plot for N = 100.000 and Delta = 0.05

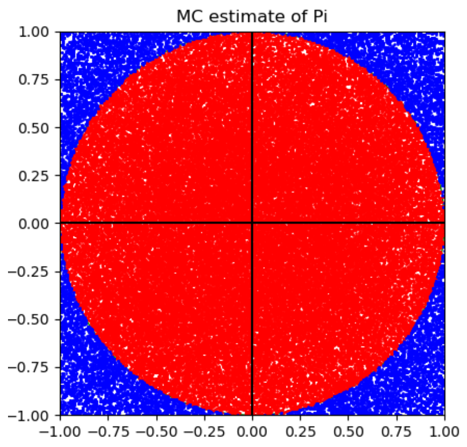
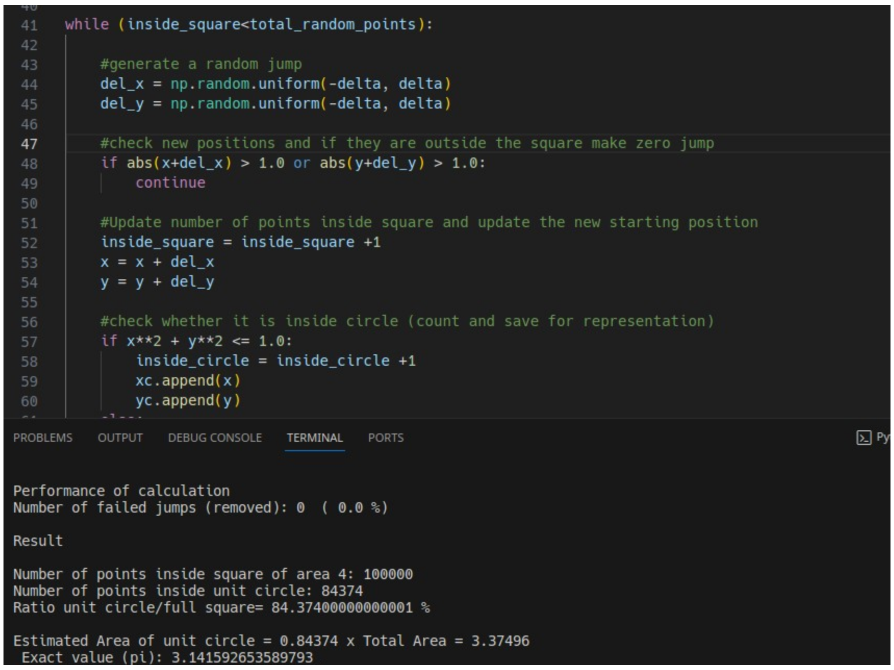


Figure 2: Resulting scatter plot when N = 100.000 and Delta = 0.6

3) Controversy revolving rejected Monte Carlo moves.
I think the original Markov chain rejection rule is generally fine. I can only see it significantly affecting the result in cases where the rejected step happens at a position where the droplet needs multiple moves to re-enter the circle (when a low value of delta is used), meaning that the position of the next moves does not follow the general probability very well (like manipulating the odds). However, that would only be true for calculations with very few iterations or moves, and neglecting the rejected move and performing it again until accepted does affect the distribution a lot, since every time the droplet is outside the circle inside the square, there would be a higher chance that the next move directs it back into the circle. Thus, I expect the calculation resulting in a higher number than the actual PI.



```
while (inside_square<total_random_points):

    #generate a random jump
    del_x = np.random.uniform(-delta, delta)
    del_y = np.random.uniform(-delta, delta)

    #check new positions and if they are outside the square make zero jump
    if abs(x+del_x) > 1.0 or abs(y+del_y) > 1.0:
        continue

    #Update number of points inside square and update the new starting position
    inside_square = inside_square +1
    x = x + del_x
    y = y + del_y

    #check whether it is inside circle (count and save for representation)
    if x**2 + y**2 <= 1.0:
        inside_circle = inside_circle +1
        xc.append(x)
        yc.append(y)
```

```
Performance of calculation
Number of failed jumps (removed): 0  ( 0.0 %)

Result

Number of points inside square of area 4: 100000
Number of points inside unit circle: 84374
Ratio unit circle/full square= 84.37400000000001 %

Estimated Area of unit circle = 0.84374 x Total Area = 3.37496
 Exact value (pi): 3.141592653589793
```

As expected, after modifying the code to ignore rejected moves and making them anew, the calculated value of PI is way over the its actual value. Note: the same N and delta parameters were used as in previous calculations.

The Ising model:

4) ising_snapshots.

a) Monte Carlo or Markov chain?
The code makes use of a Markov chain, since the probability of acceptance of the spin state at each position of the grid is not independent and instead is affected by the previous modifications or non modifications. This happens because the acceptance is calculated in the mcmove() method and it takes the surrounding spin states into account.

b) Number of spin changes in every iteration.
Either none or one change is performed. The code checks one random location at a time (within the grid) and changes the spin or not based on the energetic viability of such configuration.

c) Where is E calculated? What does the %N operation do?

E is calculated in:

```
s = config[a, b]
#calculate energy cost of this new configuration (the % is for calculation of periodic boundary condition)
nb = config[(a+1)%N,b] + config[a,(b+1)%N] + config[(a-1)%N,b] + config[a,(b-1)%N]
cost = 2*s*nb
```

The %N operation gives the division residual of whatever is divided by N. In the context of this code, it is used to check whether the selected coordinate is the edge of the grid or not. Given that it is, then check the configuration of the opposite edge of the grid (this is done considering that the unit grid is repeated in space?).

d) Spin acceptance/rejection?

The program simply rejects spin configurations where the energy cost is negative, and it may also reject states following Boltzmann probability (whether a random number is over or under the boltzman factor `elif rand() < np.exp(-cost*beta)`).

5) ising_snapshot_v2.

The average energy and average magnetization are calculated.

The energy is calculated by adding up every -nb*S term (local and surrounding energy) and dividing it by 4 and the total number of positions.

The magnetization is simply calculated by adding up all the spin states and dividing by the total number of positions.