Computational Physics - Exercise 9

Maurice Donner Lukas Häffner

June 27, 2019

1 Random Numbers – Rolling Dice

We write a simple portable random number generator, using linear congruences:

$$I_{j+1} = aI_j + c \pmod{m} \tag{1}$$

For that, we create an initiation function, that creates a global Variable.

```
def init(initVal):
    global rand
    rand = initVal
```

This variable will now be rewritten over and over again by a number generated by (1):

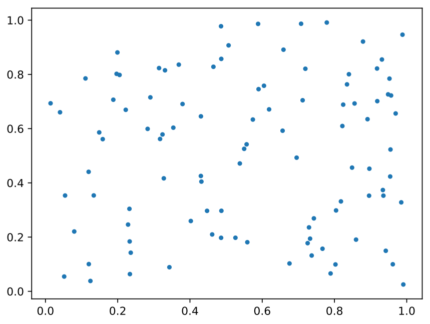
```
def generate_random(a,m,c,initVal):
    global rand
    rand = (a*rand+c)%m
    return rand
```

This function produces homogeneously distributed numbers between 0 and $(m-1)$. It can be normalized by $r_i = I_j/(m-1)$, to get homogeneously distributed real numbers between 0 and 1. We try it for example for $a = 106, m = 6075, c = 1283$:

```
Generating random number... a = 106, m = 6075, c = 1283
Random number sequence:
0: 1389
1: 2717
2: 3760
3: 4968
4: 5441
5: 904
6: 5982
7: 3575
8: 3583
9: 4431

Normalizing...
0: 0.22867961804412248
1: 0.4473164306881791
2: 0.6190319394138953
3: 0.8179124135660191
4: 0.8957853144550544
5: 0.14883108330589398
6: 0.9848534738228515
7: 0.5885742509054989
8: 0.5898913401382944
9: 0.7295027988146197
```

A simple way to check 'by eye' that the random number generator really does produce homogeneously distributed numbers, is to create two sequences with different initial values I_0 and J_0. Let $\mathbf{r}$ be a random normalized number sequence generated with $I_0 = 1$ and $\mathbf{s}$ generated with $J_0 = 2$. We plot all pairs (r_i, s_i) of generated numbers in a number plane:


(a) $n = 100$

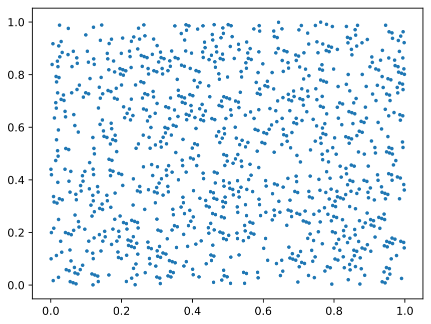

(b) $n = 1000$

Figure 1: Creating a set of random numbers

The eye is quite sensitive to see a good distribution. For $m = 100$ there is no visible pattern. Taking 1000 samples, will result in numbers, that appear random at first, however, there are small patterns forming, that look like little lines all pointing towards the same direction. To see the deterministic character of this random number sequence, we plot I_{j+1} against I_j for $I_0 = 1,$ and $n = 1000$:

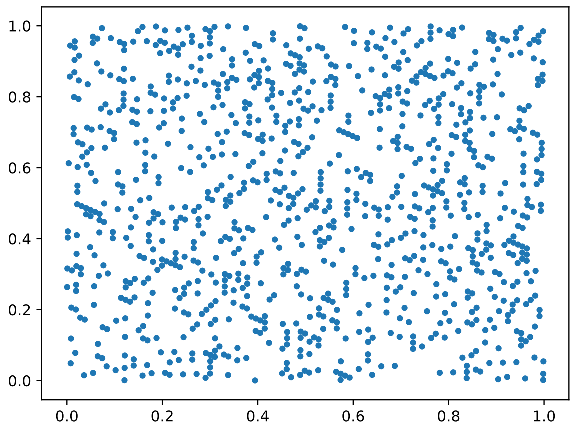


Figure 2: Deterministic Character of random number sequence

Again, overall the generator appears random, yet small patterns can still be seen. This effect can be midigated, by choosing larger values for a, m, and c. For example, for $a = 1060, m = 60751, c = 12835$, the small patterns, formerly visible in the deterministic character of our sequence vanishes. (Note, that choosing prime numbers for m further counteracts the formation of patterns)

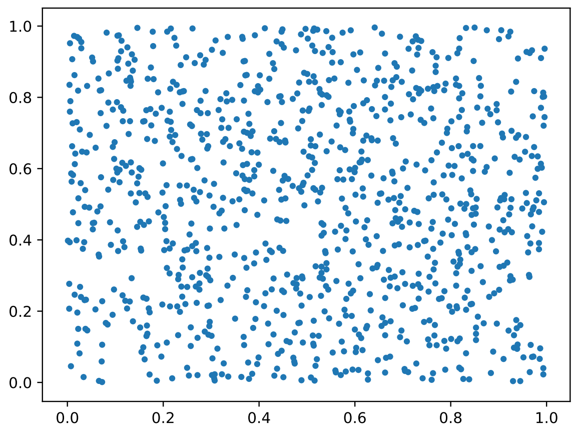


Figure 3: Choosing larger values → The plot becomes ”more random”

Next we create an experiment where the generator rolls dice. For that, we normalize our generator to 6, instead of 1 (This will be done in an extra `.py` file to avoid confusion). In this configuration, our random number generator generates numbers between 0 and 6. Because those numbers are equally distributed, all we need to do is an `int`-conversion. With that, python completely neglects any digits behind the comma. And our random numbers will hence be integer values between 0 and 5.
After adding up packets of 10 rolls each, we get a distribution:

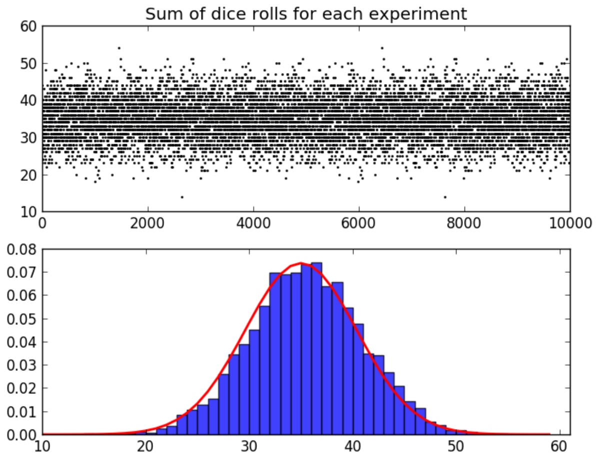


Figure 4: Adding 10 Dice rolls – Distribution

The Dice rolls follow a Gaussian Distribution according to the Central Limit Theorem.

2 Probability distribution functions

We consider a probablity distribution function given in the domain [0,a) by

$$p(x) = bx \tag{2}$$

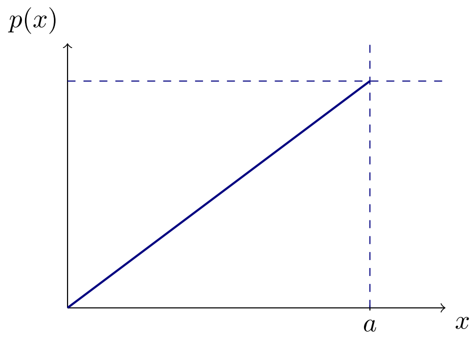


Figure 5: Probability disribution $p(x)$

To norm this distribution, we simply set the value of f(x) at x = a to 1:

$$\begin{aligned} f(a) &\overset{!}{=} 1 \\ &\Rightarrow b \cdot a = 1 \Rightarrow b = \frac{1}{a} \end{aligned} \tag{3}$$

Now we plot again our random number plane, but with the condition, that $s_i < f(x_i)$, with $x_i = r_i \cdot a$. Lastly, we plot a histogram with our generated values, to see if the histogram indeed follows equation (2). We experiment with different setsizes:

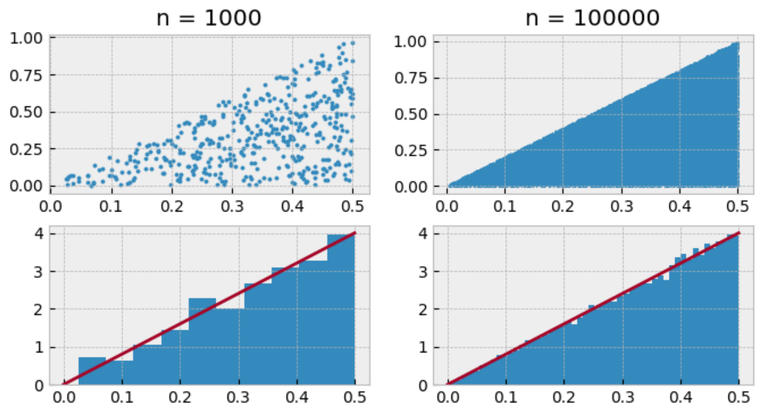


Figure 6: Distribution function $p(x)$

3 Determine π with random numbers RN

Lastly we use the same restriction method as before but with the function

$$f(x) = \sqrt{1 - x^2} \text{ for } 0 \le x \le 1. \tag{4}$$

Implementing this into code:

```python
def rejectionmethod(a,m,c,initVal,steps):
    # initialize with any starting value
    init(initVal)

    # Create Random number distribution
    r_i = ([])
    s_i = ([])
    acc = 0
    for i in range(steps):
        tmpr = generate_random(a,m,c)/(m-1) # r_i
        tmps = generate_random(a,m,c)/(m-1) # s_i
        f_x = np.sqrt(1-tmpr**2)            # f(x_i) = b*x_i
        if (tmps < f_x):
            r_i.append(tmpr)
            s_i.append(tmps)
            acc+=1                          # Count accepted values
    return r_i, s_i, acc
```

Using one quadrant of the coordinate system $(x, f(x) > 0)$, we generate numbers inside of a quarter of a circle. By dividing the accepted values by the total number of random numbers generated, and multiplying by 4, we can estimate the number π:

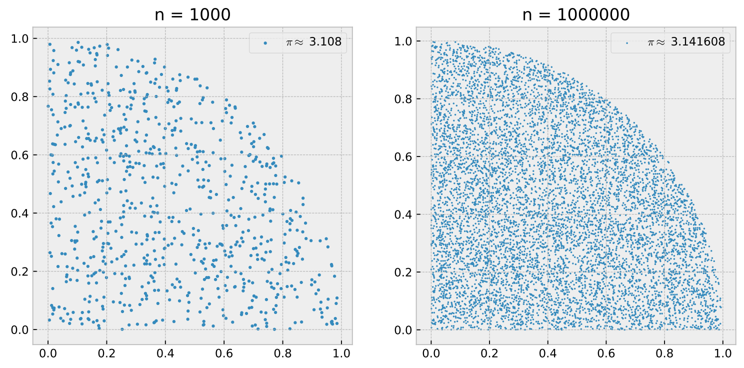


Figure 7: Approximating π with random numbers

Its obvious, that the accuracy of this procedure strongly depends on the amount of generated numbers, so we rewrite the former code, to iterate through different stepsizes (for different orders of magnitudes):

```python
def rejectionmethod(a,m,c,initVal,N):
    # initialize with any starting value
    init(initVal)
```

```
pi_estimations = np.ndarray((len(N)))
for setsize in enumerate(N):
    acc = 0
    print(r'Approximating␣$\pi$␣with␣', setsize[1], '␣random␣numbers...')
    for i in range(setsize[1]):
        tmpr = generate_random(a,m,c)/(m-1) # r_i
        tmps = generate_random(a,m,c)/(m-1) # s_i
        f_x = np.sqrt(1-tmpr**2)      # f(x_i) = b*x_i
        if (tmps < f_x):
            acc+=1
    pi_estimations[setsize[0]] = (acc/setsize[1]*4)
return pi_estimations
```

For this function, we choose an array of setsizes `N` that range between 100 and 10^n with $n > 2$ being our desired maximum order of magnitude. We then plot the Accuracy of our approximated π as a function of the setsize:

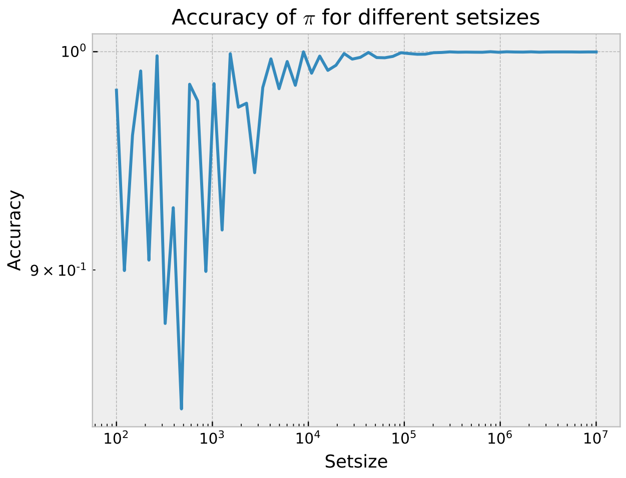


Figure 8: Accuracy of π as a function of the setsize n

Note: This method is not a really good method to approximate π since our random generator is slightly imperfect. Even for really high values of $a, m,$ and c, tiny patterns emerge and offset the value for π a little. However, with this method, π could be estimated with an accuracy of up to 3 decimals.