PHYS 512: Problem Set 7

Jade Ducharme

November 2021

1 Problem 1

First, I plotted the (x, y, z) triplets from *rand_points.txt* in a 3D plot and played around with the axes until you could sort of tell the numbers were aligned along a set of planes, as illustrated in Figure 1.

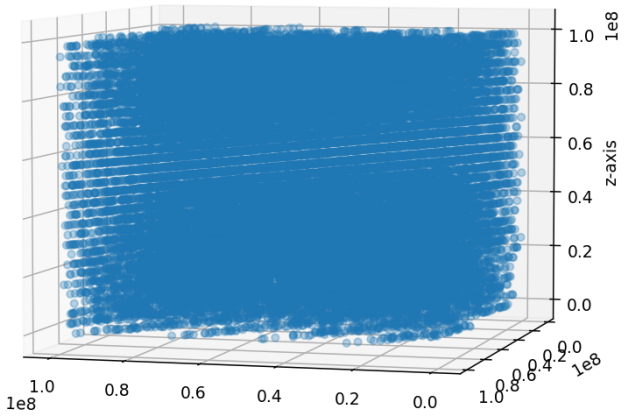

Figure 1: 3D plot of (x, y, z) triplets from *rand_points.txt*. You can definitely tell they aren't perfectly random, but it's hard to say whether they actually lie on a set of planes.

Next, I wanted to get a better picture of it, so using *scipy.optimize.curve_fit*, I fit x, y, and z to the following equation:

$$z = ax + by + c \tag{1}$$

I then plotted z as a function of $ax + by$, which showed much more clearly that the "random" points are all aligned on a set of planes. This is illustrated in Figure 2.

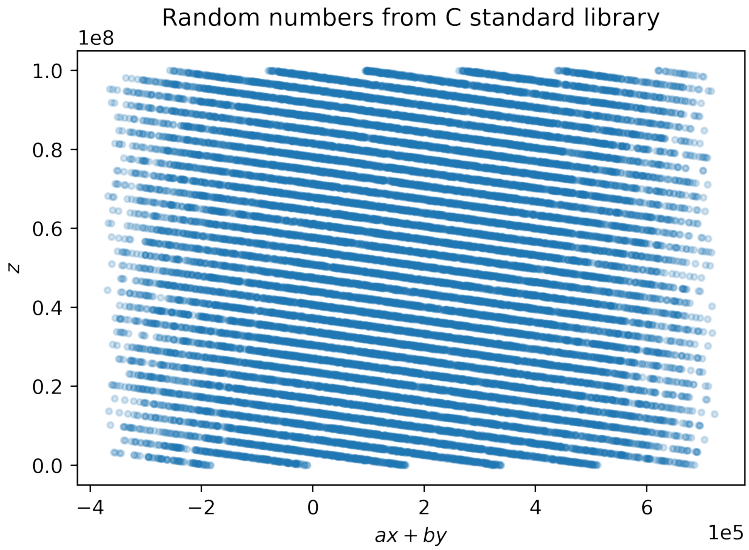

Figure 2: z as a function of $ax + by$ for a, b found using *scipy.optimize.curve_fit* for the random triplets in *rand_points.txt*. It is now very clear that these numbers are not random and lie along a set of planes.

Next, I simulated 3×10^8 random triplets between 0 and 2^{31} using Python's *random.randint* function. I plotted them on a 3d plot, played around with it, and couldn't see any planes along which the numbers could be distributed. They seemed perfectly random.

I tried fitting them to Equation 1, then plotted z as a function of $ax + by$, but they still seemed perfectly random, as shown in Figure 3.

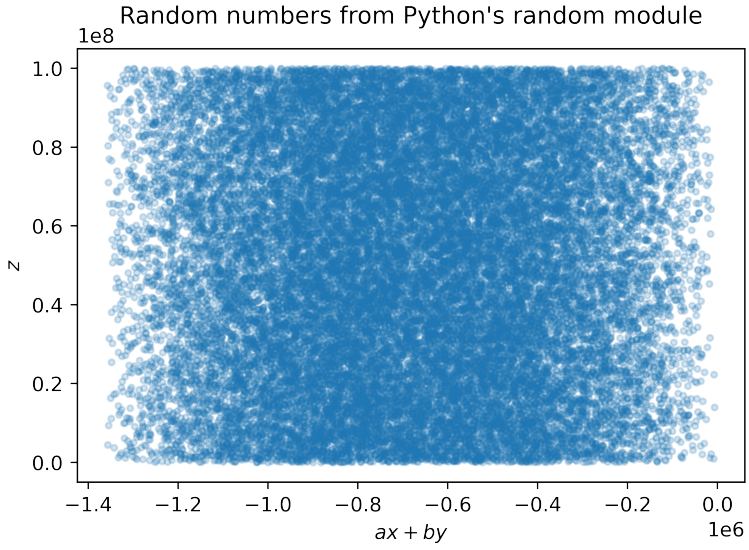

Figure 3: z as a function of $ax + by$ for a, b found using *scipy.optimize.curve_fit* for random triplets generated using Python's *random.randint* function. They appear perfectly random.

Finally, I tried loading the C standard library on my own device and simulating some random triplets, but when I went through the maneuver of plotting them on a 3D plot, fitting them to Equation 1, and plotting z vs. $ax + by$, they seemed perfectly random throughout, as shown in Figure 4.

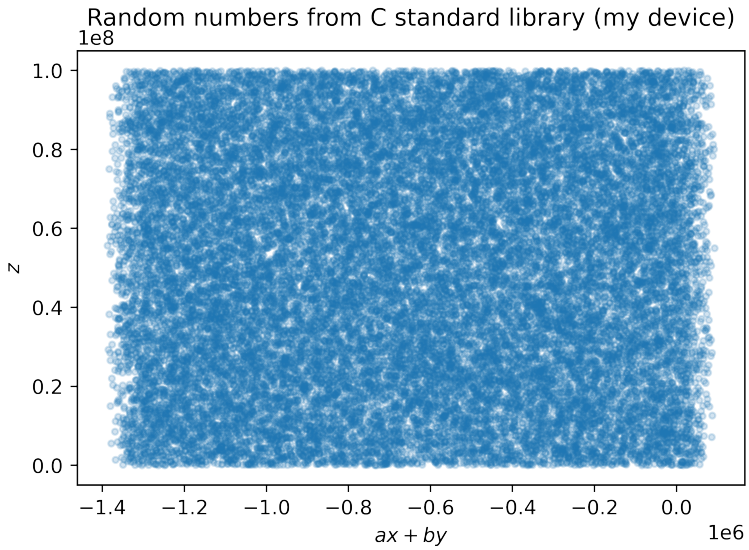

Figure 4: z as a function of $ax+by$ for a, b found using *scipy.optimize.curve_fit* for random triplets generated using the C standard library on my device. Surprisingly, they appear perfectly random.

My best guess as to why the random numbers generated on my device don't actually seem to lie on a set of planes is that I was using an updated version of the C library where they fixed that mistake in their random generator. I don't actually know if that's the case and I couldn't really find any info online, so I'm leaving it as it is.

2 Problem 2

2.1 Choosing a boundary curve

For this problem, the bounding curve could either be a Lorentzian on $[0, \infty)$ or a power-law of the form $f(x) = (1-a)x^{-a}$ on $[0, 1]$ for small values of a.

A Gaussian would be a bad option because it is only larger than an exponential on a very short range for small values of σ. The power law is also not ideal because it limits us to $[0, 1]$. The Lorentzian is therefore the obvious choice, since it is defined on exactly the same range as the exponential; that it, $x \in [0, \infty)$, and because it is larger than the exponential on the entirety of that range.

I present a visual comparison of different boundary curves in Figure 5.

2.2 Lorentzian boundary curve

I spent a lot of time trying to set up a Lorentzian PRNG. I thought there would be a *numpy* function for it, but apparently they only have a "standard Cauchy" function, which they define as:

$$f(x) = \frac{1}{\pi(x^2+1)} \tag{2}$$

Whereas I want the pure Lorentzian:

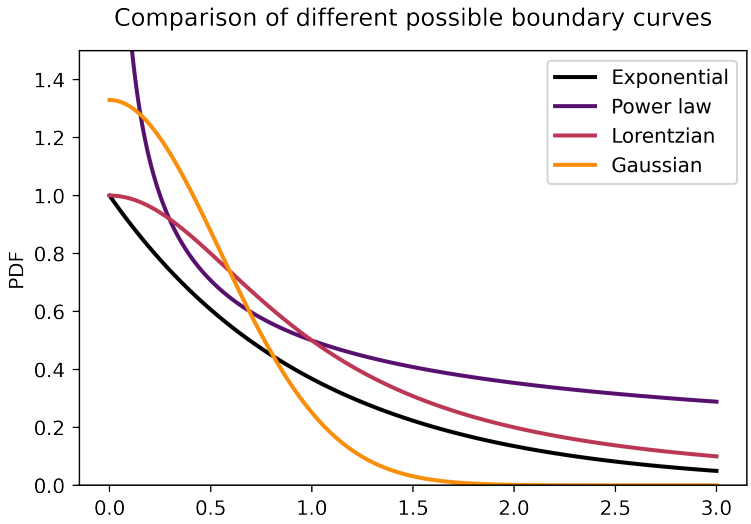

Figure 5: Different possible boundary curves for the rejection method as compared to the desired probability distribution function (exponential). The Lorentzian is closest to the exponential, making it the best boundary curve.

$$f(x) = \frac{1}{x^2 + 1} \tag{3}$$

without the annoying factor of $\frac{1}{\pi}$. That extra factor causes the standard Cauchy to only be larger than the exponential on $x > 3.95816$, at which point the exponential already lost 98% of its amplitude. So basically this standard Cauchy is completely useless.

I therefore had to set up my own Lorentzian PRNG using the transformation method. This gave rise to an annoying problem... the CDF of the Lorentzian is given by:

$$F(x) = \arctan(x) \tag{4}$$

Inverting that then gives me $y = \tan(x)$, where $x \sim U(0, 1)$ and $y \sim \text{Lorentzian}()$. This means y is always going to be between 0 and $\frac{\pi}{2}$, since that is the range of the tangent function for x between 0 and 1. At $x = \frac{\pi}{2}$, the exponential is at $\approx 21\%$ amplitude, so we can still sample on about 79% of the curve, but it is unfortunate to be forced to abandon the rest.

To make sure my Lorentzian PRNG works, I simulated a bunch of uniform random variables and transformed them into Lorentzian deviates using Equation 4. I plotted the histogram, which I present in Figure 6.

After confirming the Lorentzian PRNG worked, I focused on writing a rejection method using the steps outlined in *Numerical Recipes*:

1) Pick a uniform deviate between 0 and 1

2) Get the corresponding x value distributed according to the chosen boundary curve

3) Get the corresponding y value by plugging x in the equation of the PDF of the boundary curve

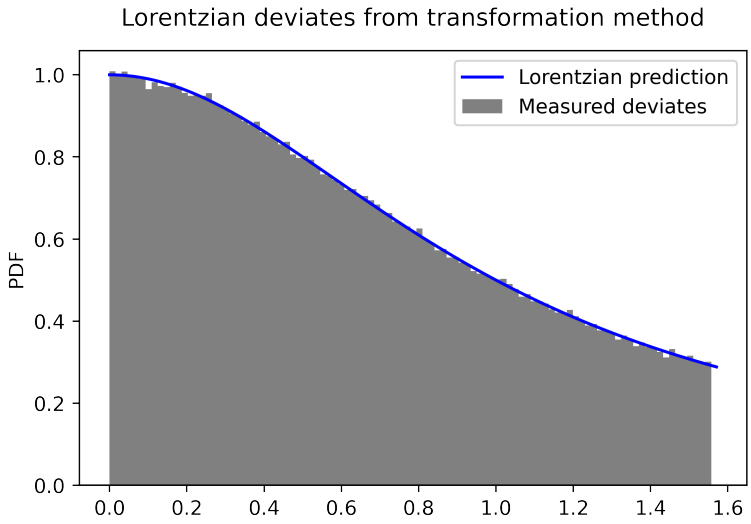

Figure 6: Lorentzian deviates generated from uniform random variables using the transformation method. Everything seems in order!

4) Pick a uniform deviate, r, between 0 and y

5) If r is smaller than e^{-x}, accept x

Using this method, I sampled $n = 10^6$ uniform deviates, turned them into Lorentzian deviates, and used the rejection method to select exponential deviates. This worked out quite nicely, as shown in Figure 7.

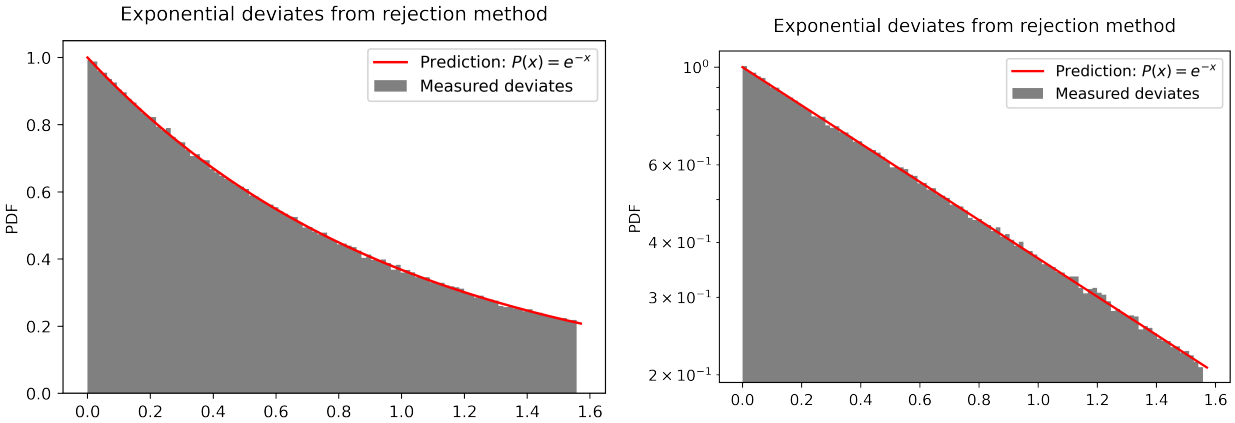

Figure 7: Exponential deviates obtained by rejection sampling from Lorentzian deviates. (Left) linear scale. (Right) log scale. It works!

Finally, I determined the efficiency by taking the ratio of the number of uniform deviates to the number of accepted deviates. The efficiency I found for the rejection method using the Lorentzian boundary curve is 78.97%.

2.3 Power law boundary curve

I also did everything but with a power law boundary curve. After feeling really depressed due to *numpy* only having the (useless) standard Cauchy function and not the Lorentzian, I completely gave up on the idea of using a Lorentzian boundary curve for a while and focused all my efforts on the power law. It's only after I wrote everything that I went back and realized I could just write my own Lorentzian PRNG and that using a Lorentzian boundary curve is (as expected) more efficient than using a power law.

Anyway, the power law I chose is given by:

$$\begin{aligned} f(x) &= (1-a)x^{-a} \\ f(x) &= 0.5x^{-0.5}, \end{aligned} \tag{5}$$

where I chose $a = 0.5$ because on $x \in [0, 1]$, this is both normalized and greater than e^{-x}. The CDF is given by:

$$F(x) = \sqrt{x} \tag{6}$$

Inverting this gives my $y = x^2$, where $x \sim U(0, 1)$ and $y \sim \text{PowerLaw}(0.5)$. Again, to make sure the PRNG works, I simulated $n = 10^6$ uniform deviates and applied Equation 6. The resulting histogram is shown in Figure 8.

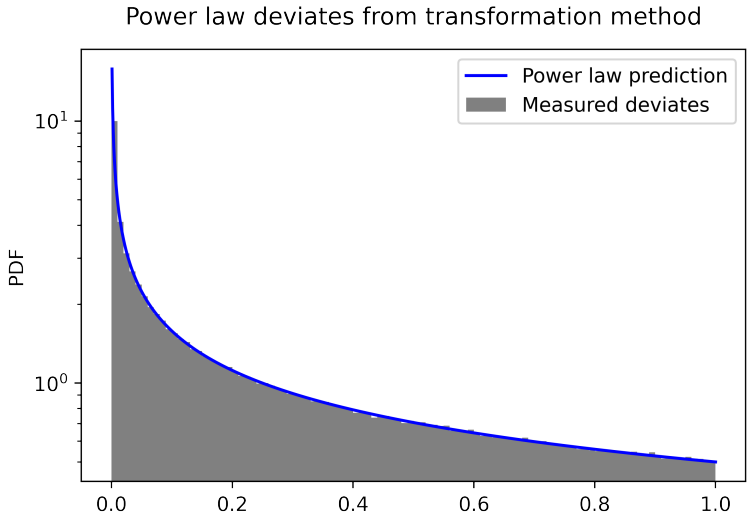

Figure 8: Power law deviates generated from uniform random variables using the transformation method. Everything seems in order! The y axis is on a log scale because it makes it easier to see that the histogram does in fact trace the power law curve.

I then followed the step-by-step method for the rejection method and produced a histogram of exponential deviates, which I present in Figure 9.

Finally, I determined the efficiency by taking the ratio of the number of uniform deviates to the number of accepted deviates. The efficiency I found for the rejection method using the power law boundary curve is 63.22%, which is a little bit worse than using the Lorentzian boundary curve.

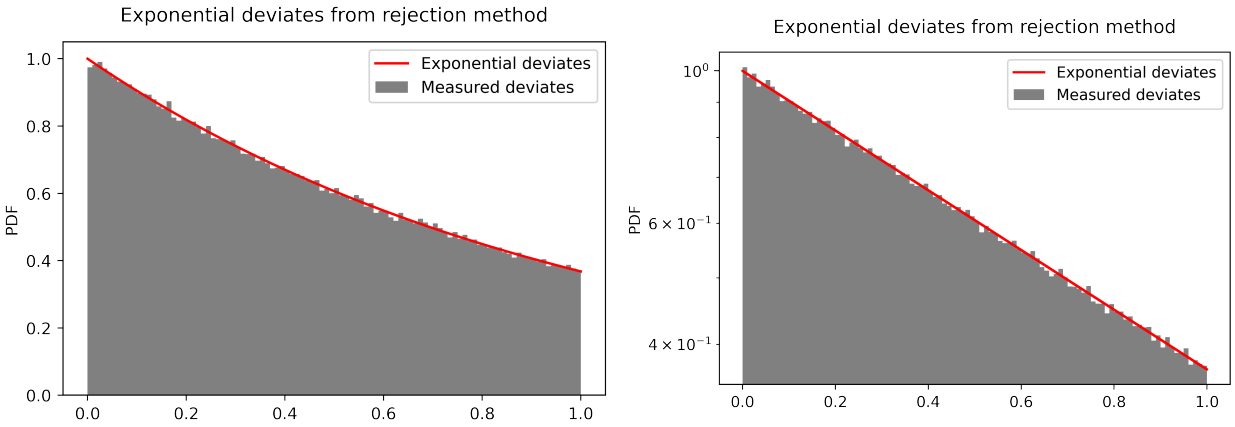

Figure 9: Exponential deviates obtained by rejection sampling from power law deviates. (Left) linear scale. (Right) log scale. It works!

3 Problem 3

I set up this problem by following the ratio-of-uniforms steps outlined in *Numerical Methods*:

1) Construct a teardrop-shaped region in the (u, v) plane bounded by $0 \leq u \leq \left[p\left(\frac{v}{u}\right)\right]^{\frac{1}{2}}$, where $p(x) = e^{-x}$ and $x = \frac{v}{u}$.

2) Sample uniformly in the rectangle bounding the teardrop region

3) Accept $\frac{v}{u}$ as a deviate if (u, v) is in the teardrop region

First, I solved for the equation of the boundary curve:

$$\begin{aligned} u &= \left[p\left(\frac{v}{u}\right)\right]^{\frac{1}{2}} \\ u &= \left[e^{-\frac{v}{u}}\right]^{\frac{1}{2}} \\ u^2 &= e^{-\frac{u}{v}} \\ \ln(u^2) &= -\frac{u}{v} \\ \therefore v &= -u\ln(u^2) \end{aligned} \tag{7}$$

This corresponds to the top part of the boundary. The bottom part would simply be given by:

$$v = u\ln(u^2) \tag{8}$$

I simulated 1000 values of u between 0 and 1 and found the corresponding v values using Equations 7 and 8. I plotted u vs. v and present it in Figure 10. I was also able to find, numerically, that for $u \in [0, 1]$, the limits on v are ± 0.7358.

After confirming that this worked and observing a nice teardrop region, I followed steps 2) and 3) outlined above for $n = 10^6$ uniform deviates and present the resulting histogram in Figure 11.

Finally, I determined the efficiency by taking the ratio of the number of uniform deviates to the number of accepted deviates. The efficiency I found for the ratio-of-uniforms method is 34.01%, so it is definitely a

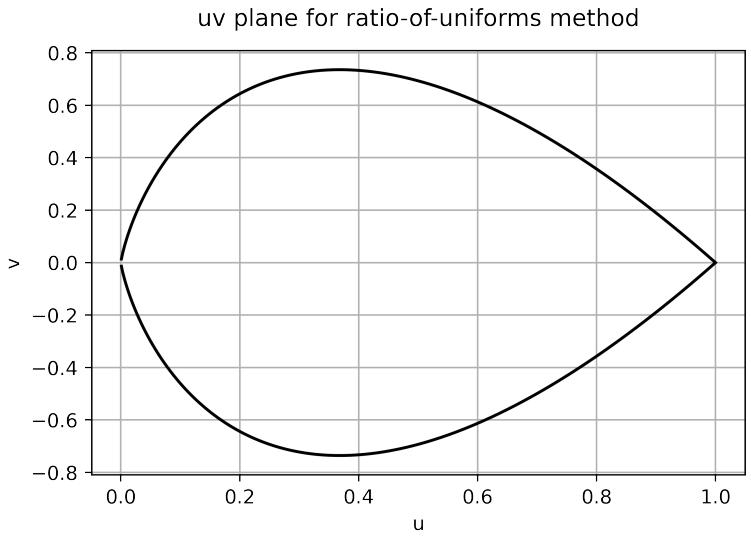

Figure 10: Teardrop-shaped boundary region in the (u, v) plane for the exponential PDF.

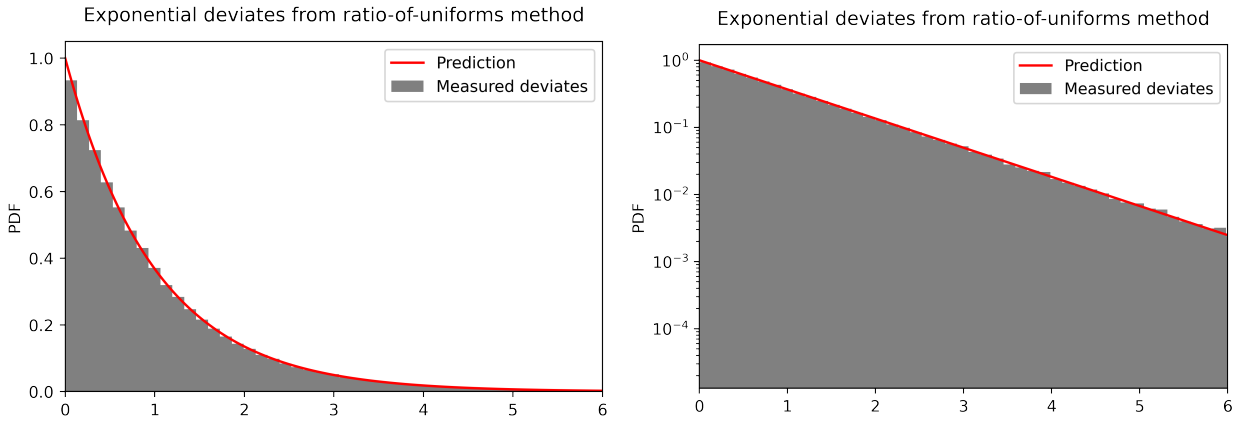

Figure 11: Exponential deviates obtained using the ratio-of-uniforms method. (Left) linear scale. (Right) log scale. It works!

lot worse in terms of efficiency than the rejection method.

However, this method allows sampling beyond $x > 1$ (which was our upper limit for the rejection method using the power law boundary curve) and even beyond $x > \frac{\pi}{2}$ (which was our upper limit for the rejection method using a Lorentzian boundary curve). So it may be less efficient, but we can actually explore and sample from the entire exponential. Depending on our purposes, the trade-off just might be worth it.