NMSM Homework Exercises

AUTHOR Guglielmo Bordin

LAST EDITED 11th November 2023

{·1} Sampling random points within *d*-dimensional domains by hit and miss

I skipped the integration on the rectangle, solving only the disk case. The source code is in `A01b_disk_hit_miss.c`; I implemented the main part of the algorithm like this:

```
for (n = 0; n < n_plot; ++n) {
    hits = 0; // reset hit counter
    throws = (1 + n) * dn;
    for (i = 0; i < throws; ++i) {
        x = RngStream_RandU01(rngs);
        y = RngStream_RandU01(rngs);
        if (x * x + y * y < 1)
            ++hits;
    }
    mc_pi = 4 * (double)hits / throws;

    // write to file: throws + \t + relative error + \n
    fprintf(file, "%d\t", throws);
    fprintf(file, "%g\n", fabs(1 - mc_pi * M_1_PI));
}
```

The error as a function of the number of throws is shown in Fig. 1. It is comfortably under 1 % with around 25 000–30 000 iterations.

{·2} Sampling random numbers from a given distribution

The idea is to sample from the probability distribution $\rho_n(x) = cx^n$ in $[0, 1]$. First, using the normalization condition we can find out what c should be:

$$1 = \int_0^1 cx^n \, dx = \frac{c}{n+1} \implies c = n + 1. \tag{2.1}$$

Then, we find the expression of the associated cumulative density function:

$$F_n(x) = (n+1) \int_0^x y^n \, dy = x^{n+1}, \tag{2.2}$$

and invert it:

$$u = x^{n+1} \implies x = u^{1/(n+1)}. \tag{2.3}$$

So, inside the code `A02a_inversion_method.c` I sample a random `double` from a uniform distribution between 0 and 1 using `drand48()`, and I raise it to the power of $1/(n+1)$ to get x:

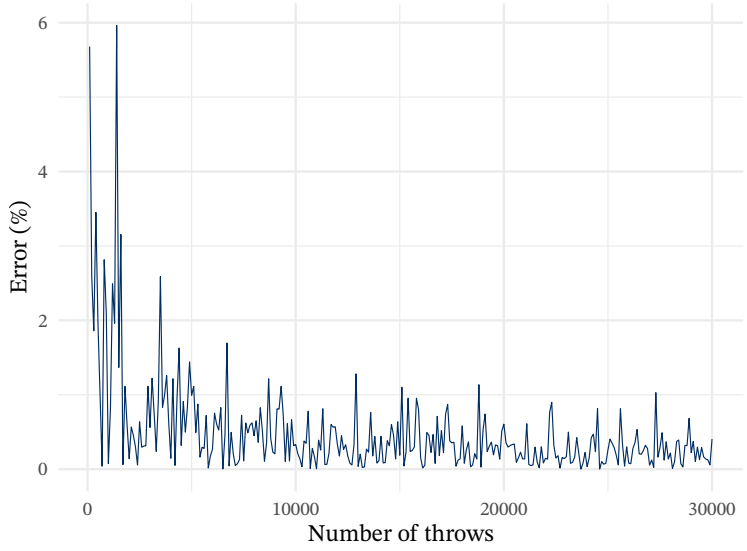

Figure 1: error in the Monte Carlo estimation of the area of a unit disk, as a function of the number of 'throws'.

```
double* x = malloc(n_smp * sizeof(*x));
for (int i = 0; i < n_smp; ++i)
    x[i] = pow(drand48(), 1.0 / (n + 1));
```

A histogram of 100 000 points sampled from ρ with $n = 3$ is displayed in Fig. 2.

Inside `A02b_inversion_method.c` I modified the code to sample from $\rho_2(x) = cx^2$ in $[0, 2]$. c is different this time, of course:

$$1 = \int_0^2 cx^2\,dx = \frac{8}{3}c \implies c = \frac{3}{8}. \tag{2.4}$$

The cumulative is then

$$F_2(x) = \frac{3}{8}\int_0^x y^2\,dy = \frac{x^3}{8} \implies x = 2u^{1/3}. \tag{2.5}$$

Once again, you can see the comparison between a 100 000-points histogram and the theoretical curve in Fig. 3.

{·3} Sampling via transformation of coordinates

If we want to sample points uniformly distributed over the unit disk, we cannot simply generate $r \sim \mathcal{U}_{[0,1]}$ and $\vartheta \sim \mathcal{U}_{[0,2\pi]}$. The result of doing that is shown in Fig. (4a), and the corresponding source code is `A03aa_disk_naive.c`. While the angular distribution poses no problem, there is an undesired higher density of points close to the centre of the disk.

The reason for this becomes clear when you consider the number of points falling inside a ring of given thickness δr. Consider a ring with inner radius r_1 and outer radius $r_1 + \delta r$, with $\delta r \ll r_1$, and another with radii r_2, $r_2 + \delta r$. Despite the ratio of their areas being r_2/r_1, the fraction of points within each ring remains the same. Thus, the number

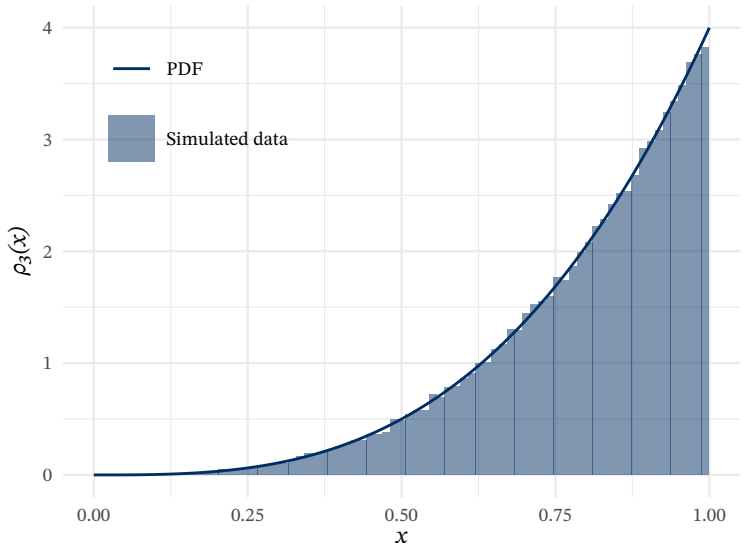

Figure 2: histogram of 100 000 points sampled from the probability distribution $4x^3$ in $[0, 1]$.

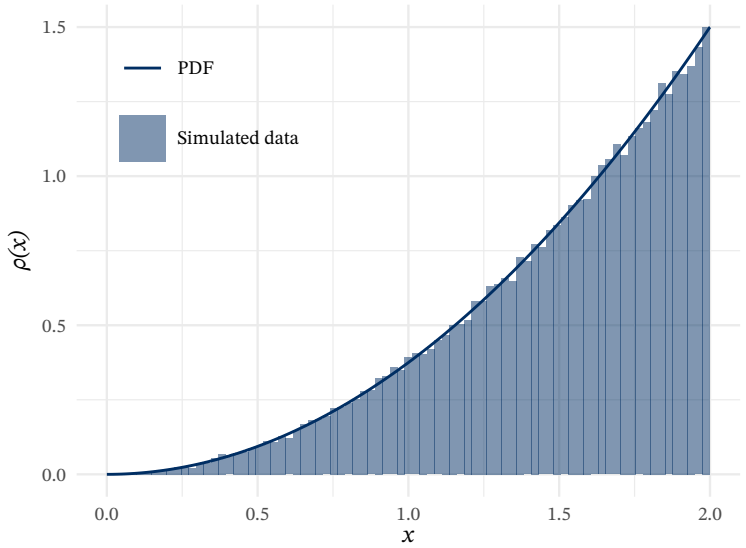

Figure 3: histogram of 100 000 points sampled from the probability distribution $3x^2/8$ in $[0, 2]$.

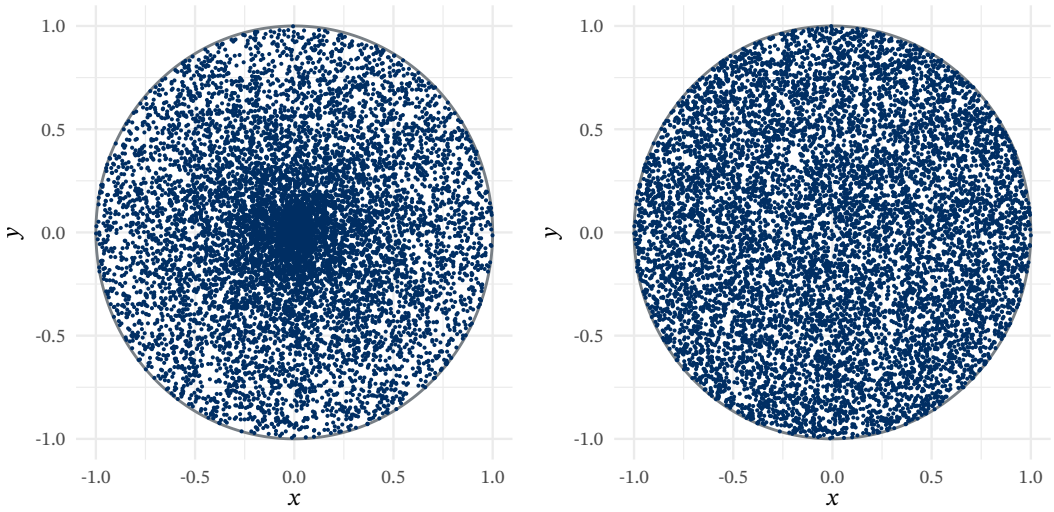

Figure 4: 10 000 points sampled on the unit disk with the wrong coordinate transformation (a) and with the correct one (b).

of points falling at a distance r from the centre should be proportional to r:

$$\rho_r(r) = 2r, \tag{3.1}$$

with the 2 in front to ensure normalization over $[0, 1]$. By computing the cumulative and inverting we get then

$$F_r(r) = 2\int_0^x x\,dx = x^2 \implies u = r^2 \implies r = \sqrt{u}. \tag{3.2}$$

So, I modified the code in `A03ab_disk_correct.c` to sample like this:

```
// allocate radius and theta arrays
double* r = malloc(n_smp * sizeof(*r));
double* t = malloc(n_smp * sizeof(*t));
for (i = 0; i < n_smp; ++i) {
    r[i] = sqrt(RngStream_RandU01(rngs));
    t[i] = 2 * M_PI * RngStream_RandU01(rngs);
}
```

The result is the correctly uniform sampling in Fig. (4b).

As regards the Box–Muller transform, I started from the factorization of $\rho_{x,y}(x, y)$:

$$\rho_{x,y}(x, y) = \frac{1}{2\pi}\exp\left(-\frac{x^2+y^2}{2}\right) = \rho_x(x)\rho_y(y), \quad \text{with } \rho_x(x) = \frac{e^{-x^2/2}}{\sqrt{2\pi}}. \tag{3.3}$$

If we switch to polar coordinates this becomes, remembering the factor r coming from the Jacobian,

$$\rho_{r,\vartheta}(r, \vartheta) = \frac{r}{2\pi}e^{-r^2/2}, \tag{3.4}$$

with $\rho_r(r) = re^{-r^2/2}$ and $\rho_\vartheta(\vartheta) = 1/2\pi$.

Then, we can marginalize over ϑ to get $\rho_r(r)$:

$$\rho_r(r) = \frac{1}{2\pi}\int_0^{2\pi} re^{-r^2/2}\,dr = re^{-r^2/2}. \tag{3.5}$$

As usual, we compute the cumulative and invert it to generate a random $r \sim \rho_r(r)$:

$$F_r(r) = \int_0^r xe^{-x^2/2}\,dx = 1 - e^{-r^2/2} \implies e^{-r^2/2} = 1 - u. \tag{3.6}$$

Since u is distributed uniformly, we can redefine it as $1 - u$ for simplicity:

$$-\frac{r^2}{2} = \log u \implies r = \sqrt{-2\log u}. \tag{3.7}$$

Then, to get the angle ϑ we recover the *conditional* distribution $\rho_{\vartheta\,|\,r}(\vartheta \mid r)$:

$$\rho_{\vartheta\,|\,r}(\vartheta \mid r) = \frac{\rho_{r,\vartheta}(r,\vartheta)}{\rho_r(r)} = \frac{1}{2\pi}. \tag{3.8}$$

Thus to get ϑ we have simply to sample from $\mathcal{U}_{[0,2\pi]}$.

The idea then is to sample two numbers u_1, u_2 from $\mathcal{U}_{[0,1]}$ at each iteration; at that point we can calculate

$$x = \sqrt{-2\log u_1}\cos(2\pi u_2), \quad y = \sqrt{-2\log u_1}\sin(2\pi u_2), \tag{3.9}$$

to get two numbers x, y distributed according to a standard Gaussian $\mathcal{N}(0,1)$. We can also get a $z \sim \mathcal{N}(\mu,\sigma)$ afterwards by multiplying x or y by σ and summing the mean: $z = \mu + \sigma x$.

The code is in `A03b_box_muller.c`; the relevant section is

```
double r, t;
double* x = malloc(n_smp * sizeof(*x));
double* y = malloc(n_smp * sizeof(*y));
for (int i = 0; i < n_smp; ++i) {
    r = sigma * sqrt(-2 * log(drand48()));
    t = 2 * M_PI * drand48();
    x[i] = mu + r * cos(t);
    y[i] = mu + r * sin(t);
}
```

You can see a histogram of 100 000 sampled points in Fig. 5.

{·4} Rejection method

In `A03ca_rejection_sampling.c` I implemented the sampling of random numbers from the probability distribution

$$f(x) = \frac{2}{\sqrt{pi}}e^{-x^2} \tag{4.1}$$

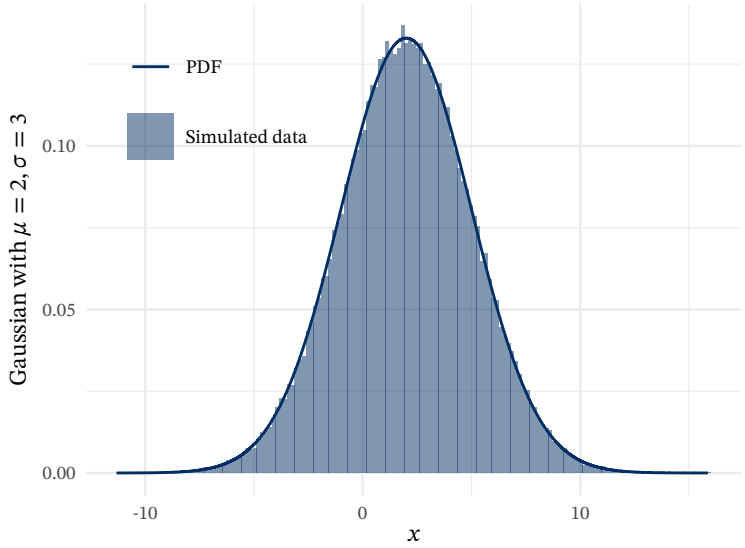

Figure 5: histogram of 100 000 points sampled with the Box–Muller algorithm from the normal distribution with $\mu = 2$ and $\sigma = 3$.

over $[0, \infty)$. I started from a function $g(x)$ that could serve as an upper limit to $f(x)$,

$$g(x) = \begin{cases} A & \text{for } 0 \leqslant x \leqslant p, \\ \dfrac{A}{p} x e^{p^2 - x^2} & \text{for } x > p. \end{cases} \tag{4.2}$$

First we normalize it to turn it into a proper probability distribution:

$$1 = \int_0^\infty g(x)\,dx = Ap + \frac{A}{p} e^{p^2} \int_p^\infty x e^{-x^2}\,dx = A\left(p + \frac{1}{2p}\right) \implies A = \frac{2p}{1 + 2p^2}. \tag{4.3}$$

Then we need a constant c such that $cg(x) \geqslant f(x)$ everywhere. The maximum of $f(x)$ is in $x = 0$, where $f(0) = 2/\sqrt{\pi}$, so we set $cA = 2/\sqrt{\pi}$. Another thing to consider is that $g(x)$ can have a maximum larger than A in $[p, \infty)$:

$$g'(x) = \begin{cases} 0 & \text{for } 0 \leqslant x \leqslant p, \\ \dfrac{A}{p}(1 - 2x^2) e^{p^2 - x^2} & \text{for } x > p. \end{cases} \tag{4.4}$$

Thus in the region $[p, \infty)$ the derivative is zero in $x = 1/\sqrt{2}$. To avoid this we have to choose $p > 1/\sqrt{2} \approx 0.707$.

Now, to generate numbers distributed according to $g(x)$ we compute, as usual, the cumulative and invert it:

$$G(x) = \begin{cases} Ax & \text{for } 0 \leqslant x \leqslant p, \\ 1 - \dfrac{A}{2p} e^{p^2 - x^2} & \text{for } x > p. \end{cases} \tag{4.5}$$

Since it is piecewise defined, we have to pay attention to the limits too:

$$\begin{cases} x = u/A & \text{for } u \leqslant Ap, \\ u = 1 - \dfrac{A}{2p} e^{p^2 - x^2} & \text{for } u > Ap, \end{cases} \tag{4.6}$$

which implies

$$x = \begin{cases} u/A & \text{for } u \leqslant Ap, \\ \sqrt{p^2 - \log\left[\frac{2p}{A}(1-u)\right]} & \text{for } u > Ap. \end{cases} \tag{4.7}$$

With the choice $cA = 2/\sqrt{\pi}$ the test $cg(x)\xi < f(x)$, with $\xi \sim \mathcal{U}_{[0,1]}$, becomes

$$\frac{\cancel{2}}{\cancel{\sqrt{\pi}}}\xi < \frac{\cancel{2}}{\cancel{\sqrt{\pi}}}e^{-x^2} \implies \xi < e^{-x^2}, \tag{4.8}$$

if we generated x with the uniform part of $g(x)$, or

$$\xi\frac{\cancel{2}}{\cancel{\sqrt{\pi}}}\frac{x}{p}e^{p^2-x^2} < \frac{\cancel{2}}{\cancel{\sqrt{\pi}}}e^{-x^2} \implies \xi x < pe^{-p^2} \tag{4.9}$$

otherwise. Let's take a look at the main loop in the code to make it clear:

```
int acc = 0;
while (acc < n_smp) {
    u = drand48();
    if (u < A * P) {
        // sample from unif g(x) = A
        x = u / A;
        if (drand48() < exp(-x * x))
            smp[acc++] = x;
    } else {
        // sample from exp g(x) = (A/p) x e^(p^2 - x^2)
        x = sqrt(p2 - log_2pA - log(1 - u));
        if (drand48() * x < P * exp(-p2))
            smp[acc++] = x;
    }
}
```

As you can see from the example histogram in Fig. 6, the sampling is correct.

In `A03cb_rejection_analysis.c` I modified slightly the code to analyse the acceptance rate of the algorithm as a function of p – subject to the condition I was mentioning earlier, $p > 1/\sqrt{2}$. You can see the results in Fig. 7: the best performance, unsurprisingly, is obtained with $p = 1/\sqrt{2}$, since for larger values $g(x)$ tends to move away from $f(x)$.

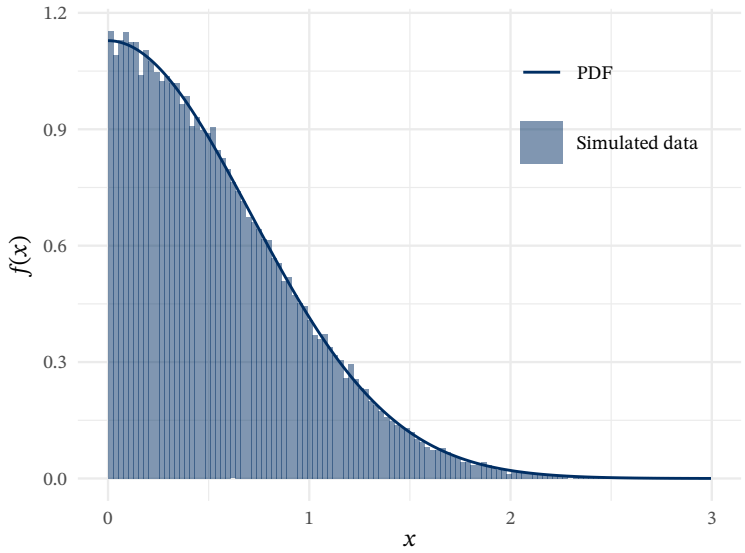

Figure 6: histogram of 100 000 points sampled from $f(x) = 2e^{-x^2}/\sqrt{\pi}$ with the rejection method implemented in `A03ca_rejection_sampling.c`.

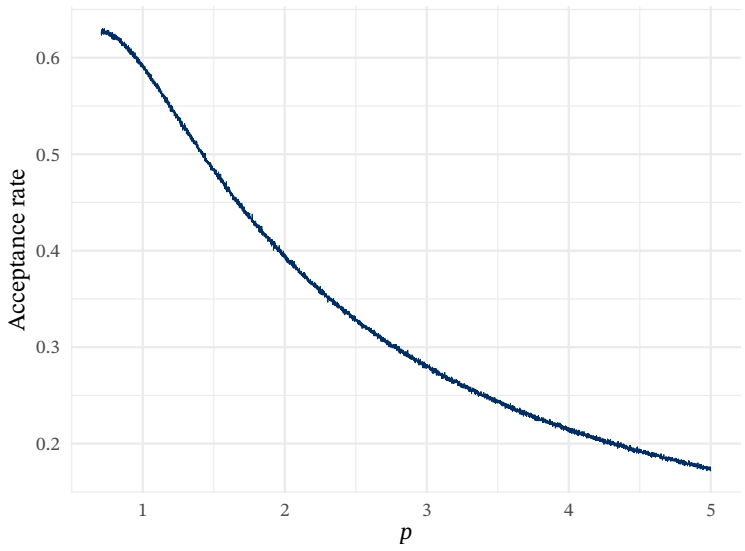

Figure 7: acceptance rate of 100 000 samples generated with the rejection method algorithm in `A03cb_rejection_analysis.c`., as a function of the breakpoint of $g(x)$.