Exercise sheet 10

by Robin Heinemann (group 4), Paul Rosendahl (group 4)

February 1, 2019

1 Rejection sampling

A probability distribution $p(x)$ on the domain $[0, a)$ is given by

$$p(x) = bx$$

To be a proper probability distribution is has to be normalized meaning

$$\int_0^a p(x)\mathrm{d}x = 1$$

Using this we obtain

$$\int_0^a bx\mathrm{d}x = \frac{a^2 b}{2} = 1$$

This gives a equation for the parameter b:

$$b = \frac{2}{a^2}$$

Let $\{r_i\}$ be a set of uniformly distributed random numbers between 0 and 1. Using rejection sampling we can obtain from this set a set $\{x_i\}$ of random numbers obeying $p(x)$. We sample two numbers from $\{r_i\} : u$ and v. Then v is multiplied by a and u is multiplied by $f(a) = 2/a$. We add v to $\{x_i\}$ if $u < p(v)$. For different numbers of elements in $\{r_i\}$ this algorithm is used to obtain a set of random numbers that obey $p(x)$ and their histogram is plotted.

Figure 1: histogram for 100 elements in r_i

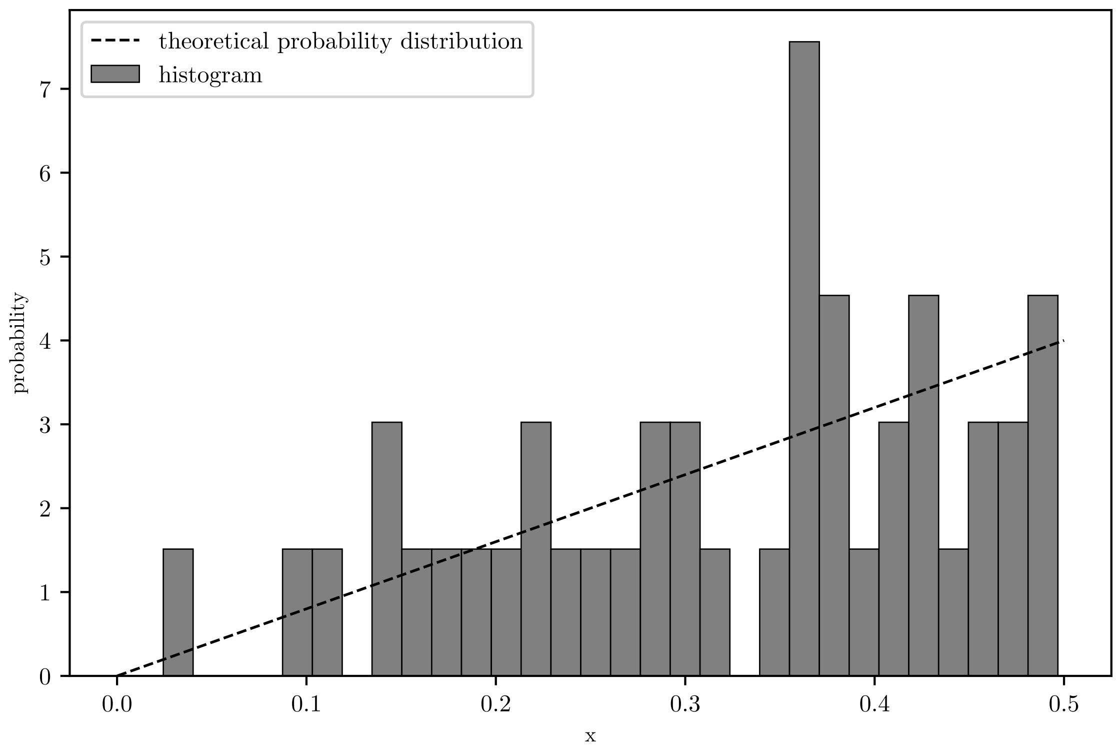


Figure 2: histogram for 100 elements in $\{r_i\}$

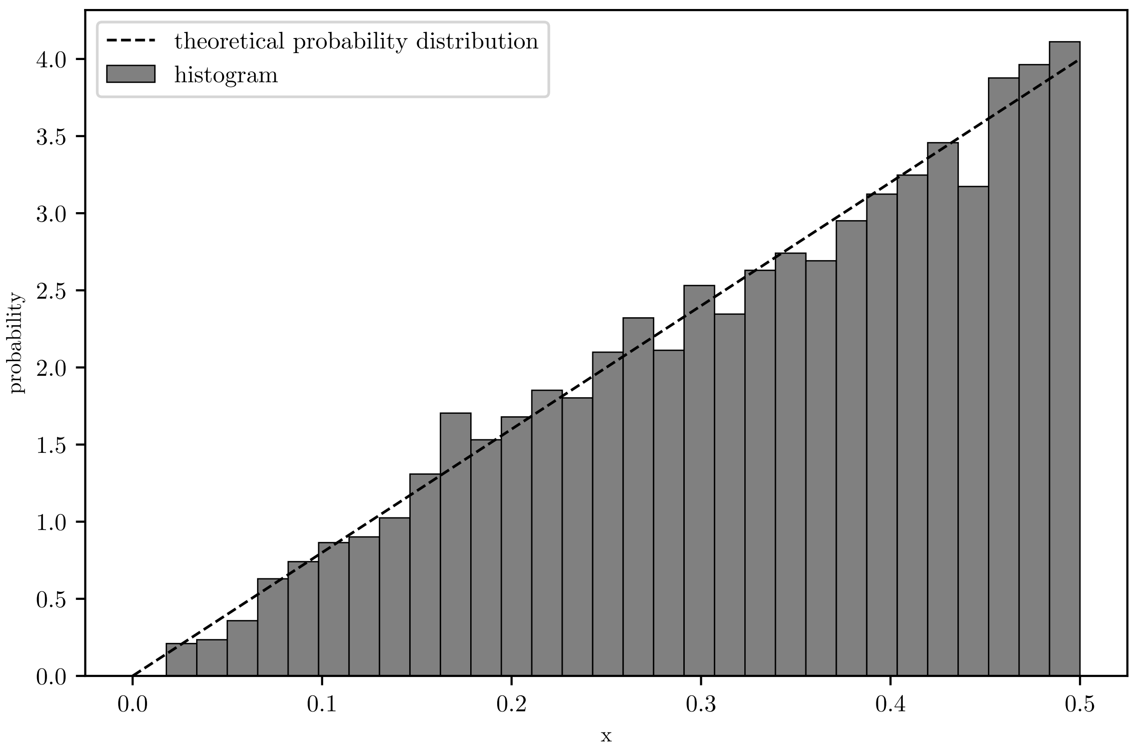


Figure 3: histogram for 100 elements in $\{r_i\}$

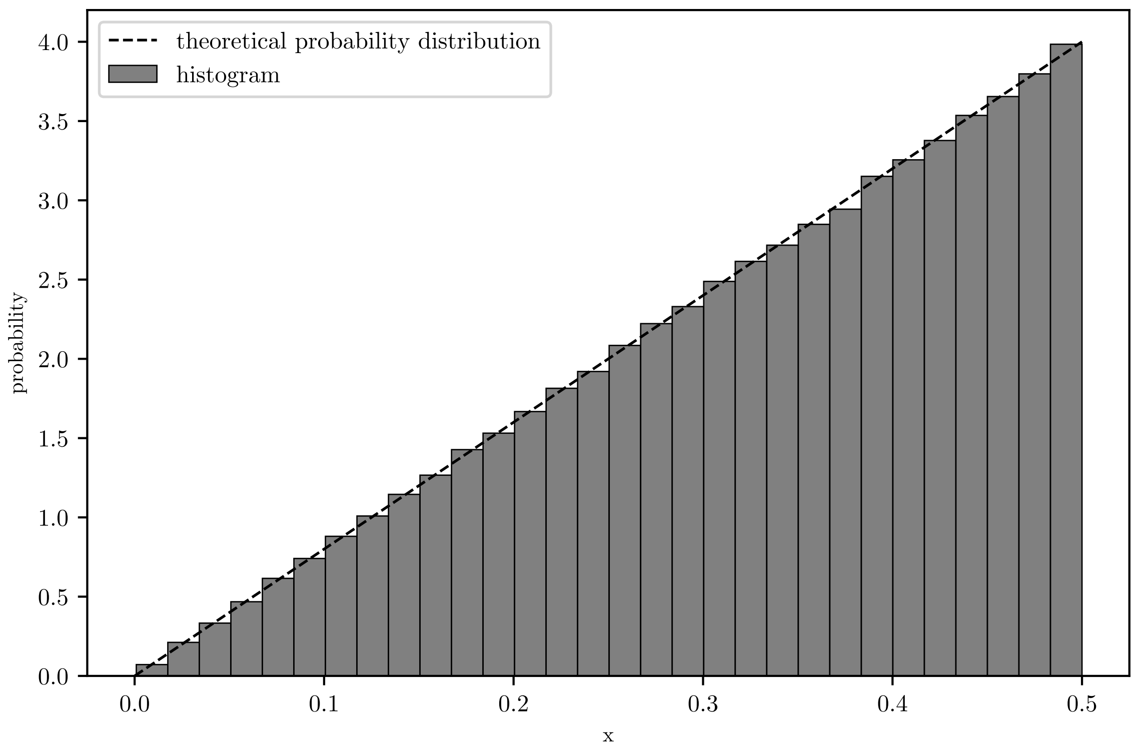
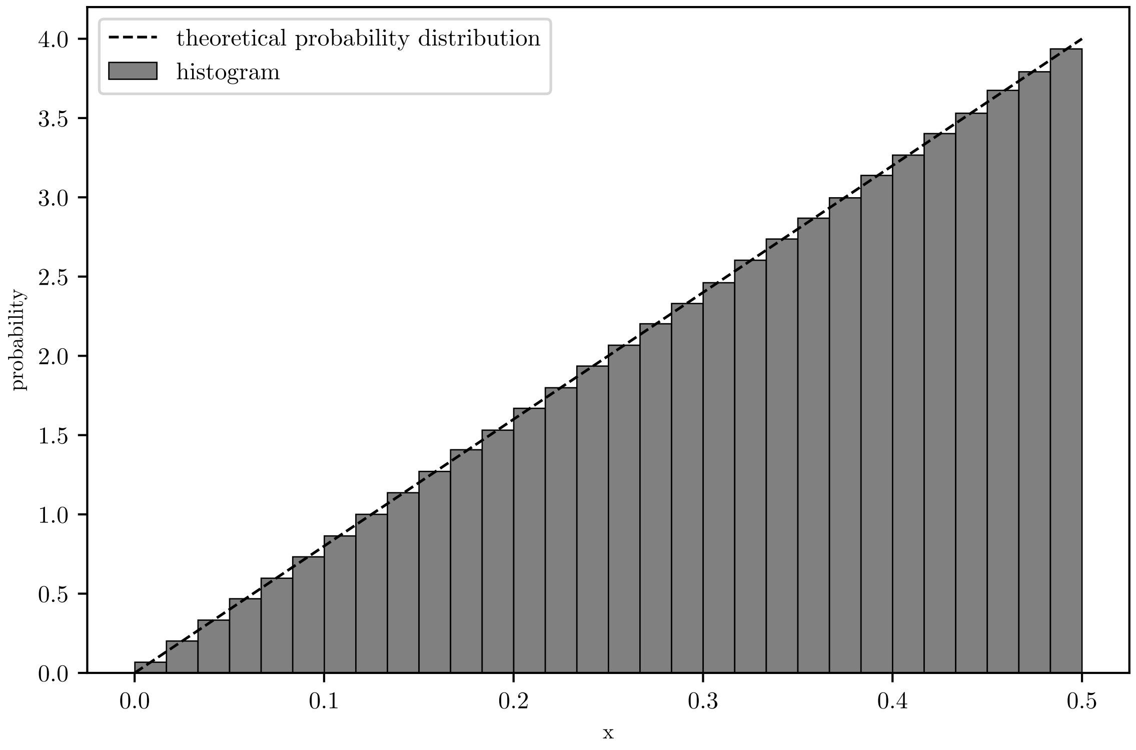


Figure 4: histogram for 100 elements in $\{r_i\}$



Using just 100 samples from $\{r_i\}$ the histogram follows $p(x)$ only very roughly. With 10 000 samples this drasticly improves and for 1 000 000 samples improves further. Finally with 10 000 000 samples the histogram is nearly indistinguishable from $p(x)$.
We can use this method of rejection sampling to easily calculate π. Observe the unit circle around the zero point. Taking two uniform random numbers x and y between 0 and 1 the probability p that they lie inside the quarter circle is simply given by the fraction of area coverd by the quarter circle in comparison to the total area coverd by the two numbers - the unit square. We obtain

$$p = \frac{A_{\text{quarter circle}}}{A_{\text{square}}} = \frac{\pi}{4}$$

p can be determined using rejection sampling of $f(x) = \sqrt{1-x^2}$ on the domain $0 \leq x \leq 1$ and counting the total number of samples N and the number of accepted samples n. We obtain $p = \frac{n}{N} \implies \pi = \frac{4n}{N}$. To analyse the accuray of this method for a fixed N the algorithm is repeated 20 times and each time the relative error is calculated. Finally the mean relative error as well as the standard deviation of the realtive error is determined. This is done for a number of different N.

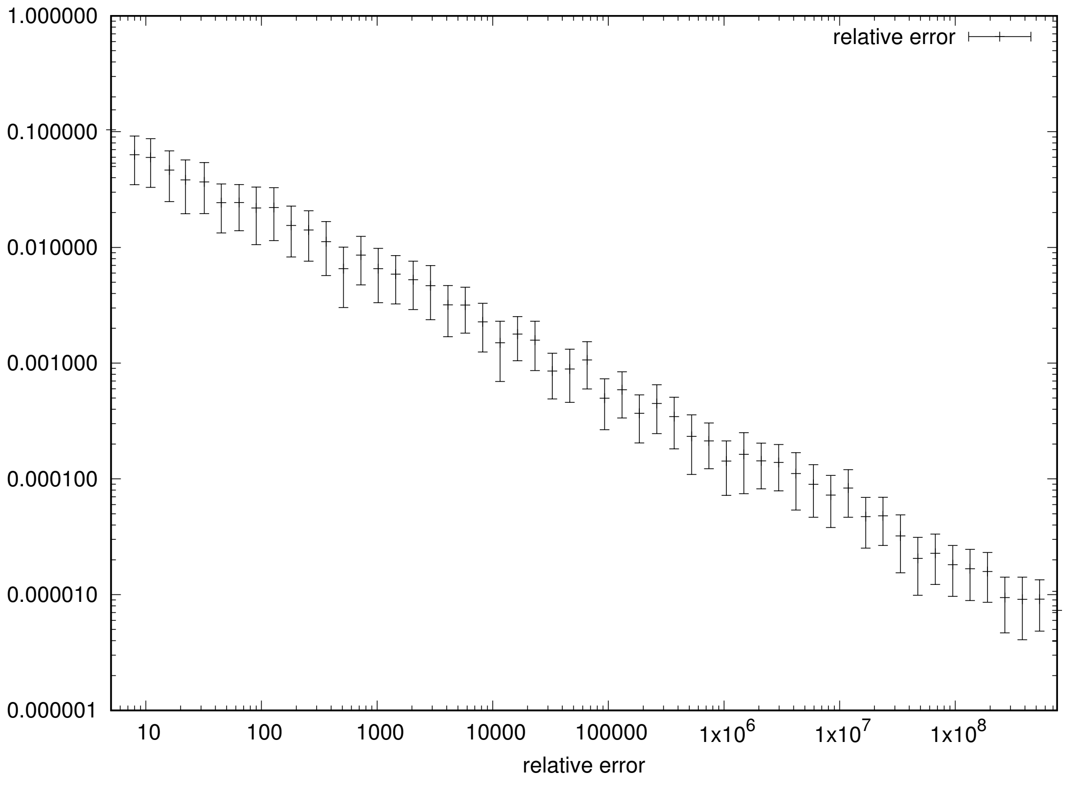


Figure 5: mean relative error for different N

This is nearly linear (in double-logarithmic representation) but slows down for big N.

2 Rust implementation

```
// cargo-deps: rand
extern crate rand;
use std::env;

fn main() {
    let pi = 3.14159265358979323846;

    for i in 5..40 {
    let n = 20;

    let mut vs = vec![0.0; n];

    let mut N: i64 = (2.0_f64.powf((i as f64) / 2.0)) as i64;
    let total = N;

    for j in 0..n {
        let mut N: i64 = (2.0_f64.powf((i as f64) / 2.0)) as i64;
        let total = N;
        let mut accepted = 0;

        while(N > 0) {
        N -= 1;

        let x = rand::random::<f64>();
        let y = rand::random::<f64>();

        if x * x + y * y < 1.0 {
            accepted += 1;
        }
        }

        let estimate = 4.0 * (accepted as f64) / (total as f64);

        vs.push((estimate - pi).abs() / pi);

    }

    let mut m = 0.0;
    for v in &vs {
        m += v;
    }

    m /= vs.len() as f64;

    let mut s = 0.0;

    for v in &vs {
        s += (m - v).powf(2.0);
    }
    s /= (n as f64) * ((n as f64) - 1.0);
    s = s.sqrt();

    // println!("{}, {}, {}", total, m, s);
    }
}
```

Listing 1: rust implementation of rejection sampling to determine π

```
// cargo-deps: rand
extern crate rand;
use std::env;

fn p(x : f64, a : f64) -> f64 {
    let b = 2.0 / (a * a);

    b * x
}

struct RejectionSample<'a> {
    f: &'a Fn(f64) -> f64,
```

```rust
    min: f64,
    max: f64,
    k: f64,
    N: usize
}

impl<'a> Iterator for RejectionSample<'a> {
    type Item = f64;

    fn next(&mut self) -> Option<f64> {
    while self.N > 0 {
        self.N -= 1;

        let u = rand::random::<f64>();

        let mut v = rand::random::<f64>();

        v *= (self.max - self.min);
        v += self.min;


        if self.k * u < (self.f)(v) {
        return Some(v);
        }
    }

    None
    }
}

fn main() {
    let args: Vec<String> = env::args().collect();

    let N = args[1].parse().unwrap();


    let a = 0.5;

    let rs = RejectionSample {
    f: &|x| p(x, a),
    min: 0.0,
    max: a,
    k: 2.0 / a,
    N: N
    };

    for i in rs {
    println!("{}", i);
    }
}
```

Listing 2: rust implementation of rejection sampling