Midterm Computer Assignment

Zack M. Davis

13 November 2024

1.

$$\int_0^1 (3 - x^7)^{\frac{3}{2}} \, dx$$

Simulating in Rust using the Monte Carlo method with a uniform proposal distribution:

```
fn integral_1a(n: usize) -> f64 {
    let mut rng = rand::thread_rng();
    let integrand = |x: f64| -> f64 { (3. - x.powi(7)).powf(1.5) };
    (0..n).map(|_| integrand(rng.gen())).sum::<f64>() / n as f64
}
```

For $n := 100,\ 1000,\ 10000$, we get 4.8564, 4.9158, 4.894423, respectively, which approximately agrees with WolframAlpha's evaluation as 4.88643.

$$\int_0^\infty x(2 + x^5)^{-4} \, dx$$

Let's use ... say, Pareto($x_m = 1$, $\alpha = 5$) shifted by 1? (Eyeballing the graph from WolframAlpha, I think I want my proposal distribution to be a little thicker than the actual tail.)

$$(y-1)(2 + (y-1)^5)^{-4} \cdot \frac{\frac{5}{y^6}}{\frac{5}{y^6}} = \underbrace{\frac{1}{5} y^6 (y-1)(2 + (y-1)^5)^{-4}}_{\text{importance weight}} \times \underbrace{\frac{5}{y^6}}_{\text{proposal PDF}}$$

We have the code:

```
fn integral_1b(n: usize) -> f64 {
    let mut rng = rand::thread_rng();
    let proposal_distribution = Pareto::new(1., 5.).expect("valid Pareto distribution");
    let importance_weight =
        |y: f64| -> f64 { 0.2 * y.powi(6) * (y - 1.) * (2. + (y - 1.).powi(5)).powi(-4) };
    (0..n)
        .map(|_| importance_weight(proposal_distribution.sample(&mut rng)))
        .sum::<f64>()
        / n as f64
}
```

Our $n := 100,\ 1000,\ 10000$ estimates are 0.03287, 0.0204, 0.022726, respectively.

Oddly, WolframAlpha claims "(integral does not converge)", but I think that might be a bug on their part. (It also says "Standard computation time exceeded", so maybe it's a timeout rather than the integral really being crazy.) Let's switch to SciPy:

```
 In [1]: import scipy.integrate
   ...:
   ...: def f(x):
   ...:     return x * (2 + x**5)**-4
   ...:
   ...: scipy.integrate.quad(f, 0, 5)
Out[1]: (0.02266516521006867, 1.803236715980025e-09)
```

So SciPy thinks I'm right.

$$\int_{0.5}^{\infty} 4x(2+2x^3)^{-4}\,dx$$

Let's go with Pareto($x_m = 0.5$, $\alpha = 5$) for our proposal distribution. (It's nice that the lower limit of integration isn't zero, so that we don't have to transform-shift to use the polynomial-tailed Pareto distribution, which demands a positive shape parameter.) That's $\frac{\alpha x_m^\alpha}{x^{\alpha+1}} = \frac{5\cdot(0.5)^5}{x^6} = \frac{5}{32}x^{-6}$.

$$4x(2+2x^3)^{-4}\cdot\frac{\frac{5}{32}x^{-6}}{\frac{5}{32}x^{-6}} = \frac{32\cdot 4}{5}x^7(2+2x^3)^{-4} = \underbrace{\frac{128}{5}x^7(2+2x^3)^{-4}}_{\text{importance weight}}\times\underbrace{\frac{5}{32}x^{-6}}_{\text{proposal PDF}}$$

And then the inevitable code (between homework #10 and this, I'm starting to get bored, and I bet the grader is, too; hi, Daniil) is:

```
fn integral_1c(n: usize) -> f64 {
    let mut rng = rand::thread_rng();
    let proposal_distribution = Pareto::new(0.5, 5.).expect("valid Pareto distribution");
    let importance_weight = |x: f64| -> f64 {
        (128./5.) * x.powi(7) * (2. + 2. * x.powi(3)).powi(-4)
    };
    (0..n)
        .map(|_| importance_weight(proposal_distribution.sample(&mut rng)))
        .sum::<f64>()
        / n as f64
}
```

which gives us 0.02606, 0.02596, 0.026225 for our $n := 100,\ 1000,\ 10000$, respectively. And SciPy says it's 0.02621, so we're good.

2. **a**. X has a binomial distribution with $p = 0.03$ and $n = 100$, $f_X(k) = \binom{100}{k}(0.03)^k(0.97)^{100-k}$.
Y has a binomial distribution with $p = 0.05$ and $n = 100$, $f_Y(k) = \binom{100}{k}(0.05)^k(0.95)^{100-k}$.
b. The following Rust function generates 1000 samples from each distribution.

```
use rand::distributions::Distribution;
use rand_distr::Binomial;

fn question_2b() -> (Vec<u64>, Vec<u64>) {
    let mut rng = rand::thread_rng();
    let x = Binomial::new(100, 0.03).expect("valid binomial distribution");
    let y = Binomial::new(100, 0.05).expect("valid binomial distribution");
    let xs = (0..1000).map(|_| x.sample(&mut rng)).collect::<Vec<_>>();
    let ys = (0..1000).map(|_| y.sample(&mut rng)).collect::<Vec<_>>();
    (xs, ys)
}
```

c. The following Rust function (with the same 'use' statements as above) empirically estimates the probability that a batch of 100 parts from each supplier has less than 10 defectives. A typical run of this program might report that, say, $P(X < 10) \approx 0.998$ and $P(Y < 10) \approx 0.978$. (Exact values vary due to the pseudorandom nature of the query.)

```
fn question_2c() -> (f64, f64) {
    let mut rng = rand::thread_rng();
    let x = Binomial::new(100, 0.03).expect("valid binomial distribution");
    let y = Binomial::new(100, 0.05).expect("valid binomial distribution");
    let xs_less_than_10 = (0..1000).map(|_| x.sample(&mut rng)).filter(|n| *n < 10).count() as f64;
    let ys_less_than_10 = (0..1000).map(|_| y.sample(&mut rng)).filter(|n| *n < 10).count() as f64;
    (xs_less_than_10 / 1000., ys_less_than_10 / 1000.)
}
```

d. The following Rust function estimates that $P(X < Y) \approx 0.186$.

```
fn question_2d() -> f64 {
    let mut rng = rand::thread_rng();
    let x = Binomial::new(100, 0.03).expect("valid binomial distribution");
    let y = Binomial::new(100, 0.05).expect("valid binomial distribution");
    let a_more_defective = (0..1000)
        .map(|_| x.sample(&mut rng) > y.sample(&mut rng))
        .filter(|p| *p)
        .count() as f64;
    a_more_defective / 1000.
}
```