Lab1_Sampling_TotallySolved

May 3, 2020

1 Lab1: Sampling from random variables

This is an exercise notebook on how to generate random numbers from a random variable given either its density or its cumulative distribution function (CDF).

We have seen - Direct methods - Acceptance-rejection method

Remember to revise the lecture on sampling before attempting to solve it!

1.1 Pseudo-random number generation

1.1.1 1. Sampling from a uniform distribution

1.1. Linear Congruential Generator (LCG) Define a function `sample_from_uniform()` that takes `N` as input and returns a sequence $x_1, \ldots, x_N$, where $x_i \sim \mathcal{U}(0,1)$ $\forall 1 \leq i \leq N$. Use the LCG algorithm with a proper choice for parameters `a`, `c` and `m`, which are going to be passed as input to the function.

Start with a seed $z_0 \in \mathbb{N}$ and define the sequence of integers $z_i = (a \cdot z_{i-1} + c)(mod \quad m)$. Then $u_i = \frac{z_i}{m} \in [0,1)$.

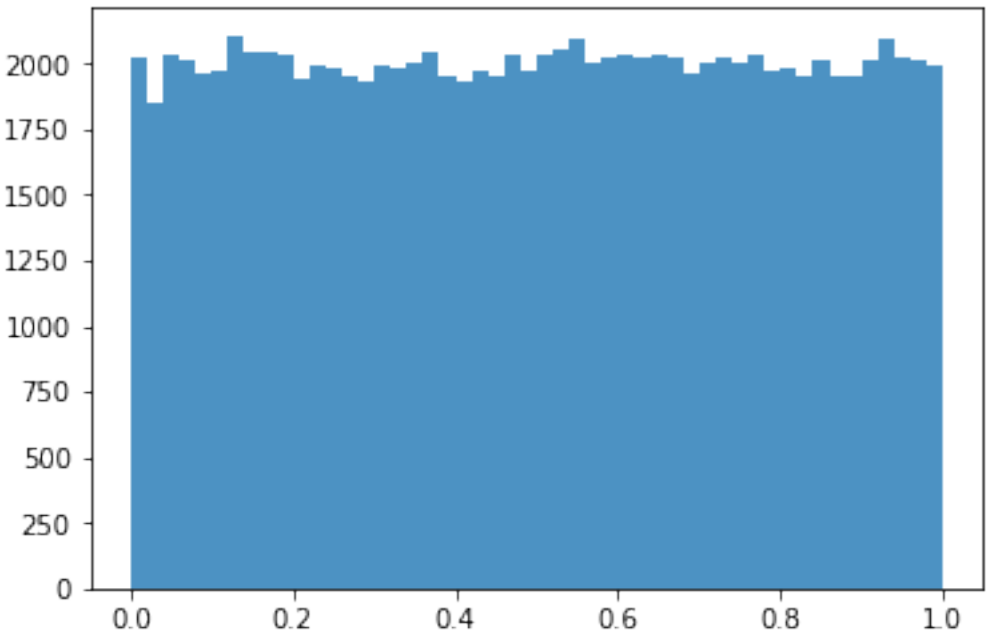
Remember to choose m very large and pick a and c in order to generate a *full-period* sequence.

```
In [1]: import numpy as np
        import matplotlib.pyplot as plt
        import time
```

```
In [2]: def sample_from_uniform(N, seed = None, a = 16807, c = 3, m = 2**31-1):

            x = np.empty(N)
            seed = int(time.time())

            x_tmp = seed
            for i in range(N):
                x[i] = (a*x_tmp+c) % m
                x_tmp = x[i]

            return x/m
```

```
In [3]: N = 100000

        U = sample_from_uniform(N)
```

```
In [4]: kwargs = dict(histtype='stepfilled', alpha=0.8, bins=50)

        plt.hist(U, **kwargs);
```



1.2. χ^2**-test** Once you have generated the `N` random variates $x_1, \ldots, x_N$, use the χ^2-test to find out whether H_0: *"*$x_1, \ldots, x_N \sim \mathcal{U}(0,1)$ *and indipendent"* can be accepted.

```
In [5]: from scipy.stats import chisquare
        n_bins= 10
        n_obs_per_bin, bin_edges = np.histogram(U, bins = n_bins)

In [6]: sign_level = 0.05
        chisq, p = chisquare(n_obs_per_bin,(N/n_bins)*np.ones(n_bins))

        print(chisq, p)
        if p > sign_level:
            print("accept the H0")
        else:
            print("reject H0")

20.729400000000002 0.01390765105417472
reject H0
```

1.3. Given a interval $I = [a, b]$, how can you sample from $\mathcal{U}(a, b)$?

```
In [7]: a = 1
        b = 10

        x = a+sample_from_uniform(10)*(b-a)
        print(x)
```

```
[3.36727406 7.77516699 3.23167358 5.73781609 6.47503428 3.90115912
 7.78134907 8.13376923 9.25942595 1.17196332]
```

1.2 2. Inversion methods

The CDF of a distribution maps a number in the domain to a probability between 0 and 1. Inversion methods apply to random variables whose CDF is invertible.

Intuition: Given a sample x from a random variable X, define $u := F_X(x)$. Sample u from $\mathcal{U}(0,1)$ and obtain a a sample from X by computing as $F_X^{-1}(u) = x$.

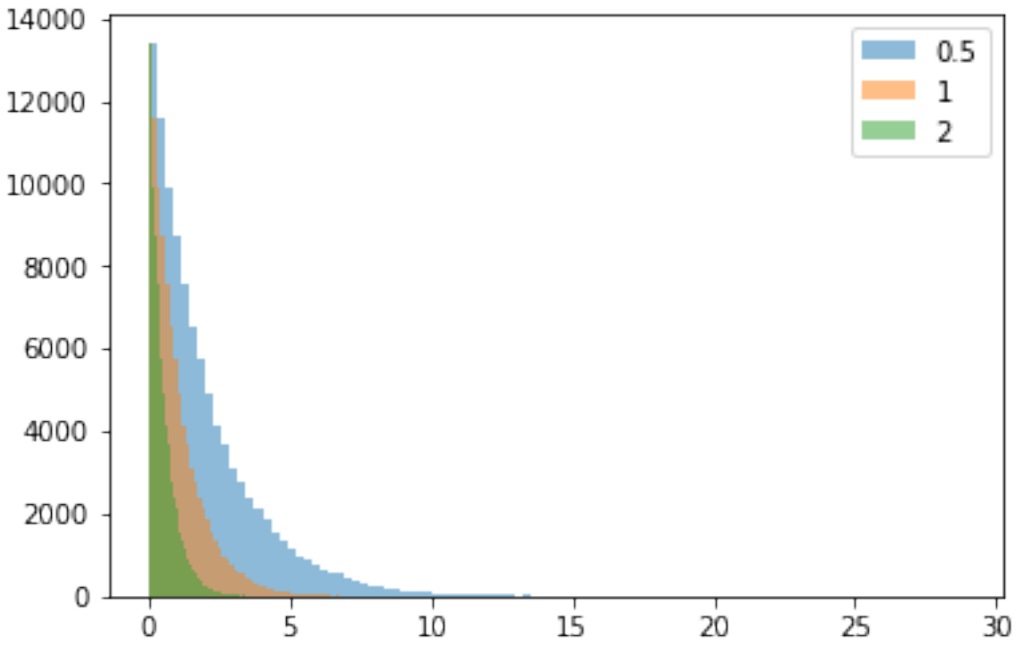
1.2.1 2.1 Exponential distribution:

Define a function `sample_from_exp()`, which takes as inputs the parameter `lambda` and the length of the ouput sequence `N`. The function should sample from the r.v. $X \sim Exp(\lambda)$ using the inversion method. Recall that $u = F_X(x) = 1 - \exp^{-\lambda x}$.

```
In [8]: def sample_from_exp(param, N, seed = 7):

            U = sample_from_uniform(N, seed)
            X = -np.log(U)/param

            return X
```

```
In [9]: l1 = 0.5
        l2 = 1
        l3 = 2

        x1 = sample_from_exp(l1,N = 100000)
        x2 = sample_from_exp(l2,N = 100000)
        x3 = sample_from_exp(l3,N = 100000)


        kwargs = dict(alpha = 0.5, bins = 100)

        plt.hist(x1, **kwargs);
        plt.hist(x2, **kwargs);
        plt.hist(x3, **kwargs);
        plt.legend(["0.5", "1", "2"])
```

```
Out[9]: <matplotlib.legend.Legend at 0x7f009d711d10>
```



2.1.1. Kolmogorov–Smirnov test KS is one of the most useful and general nonparametric methods for measuring the goodness of fit of a distribution. The Kolmogorov–Smirnov statistic quantifies a distance between the empirical distribution function of the sample and the cumulative distribution function of the reference distribution. The null distribution of this statistic is calculated under the *null hypothesis* that the sample is drawn from the reference distribution.

Build a function `ks_test()` that assess wheter or not your sample represent a good fit for the desired distribution. Use the *scipy.stats.kstest* library.

```
In [10]: import scipy.stats
         from scipy.stats import kstest

         '''
         alternative : {two-sided, less,greater}
         OUTPUTS:
         -KS test statistic (D = it compares the observed versus the expected cumulative relat
         -p-value: One-tailed or two-tailed p-value
         '''

         D, p = kstest(x2, 'expon', args= (0,1 / l2), alternative =  'less')
         print(D,p)
         # if p < sign-level, we don't believe that our variable follows a exp distribution in
         if p > sign_level:
             print("accept the H0")
         else:
             print("reject H0")
```

```
0.002631918413518841 0.2497855385736959
accept the H0
```

2.1.2. Generate a sample from a r.v. X such that its CDF is $F_X(x) = 1 - \exp(-\sqrt{x})$.

```
In [11]: inv_F = lambda U: (np.log(U))**2

         def sample_from_generic_F(inv_F, N, seed = 7):

             U = sample_from_uniform(N, seed)
             X = inv_F(U)

             return X

         sample_from_generic_F(inv_F, 10)
```

```
Out[11]: array([1.78344197e+00, 1.51094058e-02, 3.17583528e+01, 2.26152560e-04,
                3.83476414e+00, 3.76704753e-01, 4.94991131e-02, 5.65239324e-01,
                2.66071041e-01, 2.23741721e-03])
```

Remark: Computing the CDF for a discrete distribution is in general not too difficult: we simply add up the individual probabilities for the various points of the distribution. For a continuous distribution, however, we need to integrate the probability density function (PDF) of the distribution, which is impossible to do analytically for most distributions, making this method computationally inefficient for many distributions; however, it is a useful method for building more generally applicable samplers such as those based on rejection sampling.

1.2.2 2.3. Sampling from discrete random variable

Implement the algorithm `sample_discrete_rv()` which extract random samples x_i from a finite vector $p(\mathbf{x}) = [p(x_1), \ldots, p(x_n)]$ assuming $\sum_i p(x_i) = 1$.

Remember that sorting the values of $p(\mathbf{x})$ may make the algorithm more efficient. What about the sorting overhead?

```
In [12]: import random

         def sample_discrete(N,p):
             n = p.shape[0]
             S = np.cumsum(p)
             U = sample_from_uniform(N, seed)

             sampled_indexes = np.empty(N)
             for j in range(N):
                 for i in range(n):
                     if S[i] > U[j]:
                         sampled_indexes[j] = i
                         break

             return sampled_indexes
```

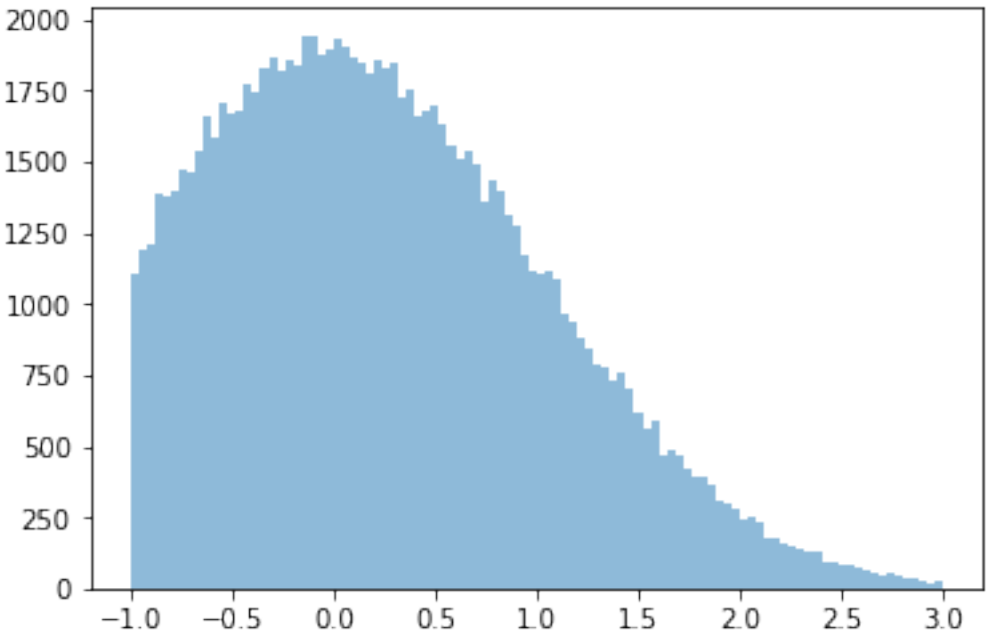
1.2.3 2.3. Sampling from a truncated distrubution

Generate a sample from a **normal distribution truncated** to $[a, b]$. For a distribution F, if you generate uniform random variates on the interval $[F(a), F(b)]$ and then apply the inverse CDF, the resulting values follow the F distribution.

In SAS the QUANTILE function , but for many distributions it has to numerically solve for the root of the equation F(x) = u.

Hint: in scipy the function norm.ppf(p) implements the inverse CDF function, i.e. it solves numerically the following root finding problem: find x such that $F(x) = p$, where F is the CDF of a standard gaussian distribution

```
In [13]: from scipy.stats import norm

         def sample_truncated_norm(N, a, b):
             Fa = norm.cdf(a)
             Fb = norm.cdf(b)
             P = Fa + (Fb-Fa)*sample_from_uniform(N)
             X = norm.ppf(P)

             return X

         a, b = -1, 3
         N = 100000
         X = sample_truncated_norm(N, a, b)
         plt.hist(X, **kwargs);
```



1.3 3. Acceptance-rejection method

Rejection sampling is a basic technique to generate observations from a distribution. The method works for any distribution in $\mathbb{R}^m$ with a density. It's used where direct methods fail.

Define a funtion `rejection_sampling()` with takes as input density `f(x)` of the target distribution. In the basic implementation a uniform $g(x)$ is sufficient.

Sketch:

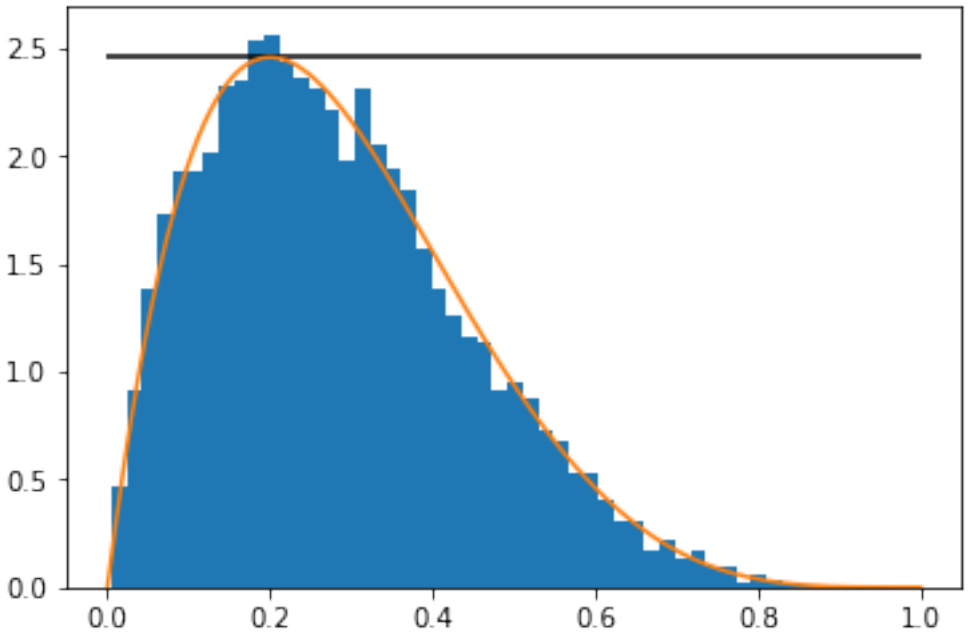
1. generate y with density $r = \frac{g(x)}{c}$, where $g(x)$ is a function majorizing the density $f(x)$ and $c = \int_{-\infty}^{\infty} g(x)dx$;
2. generate $u \in \mathcal{U}(0,1)$;
3. if $u \leq \frac{f(y)}{g(y)}$ return $x = y$, otherwise reject and go to step 1.

```
In [14]: def rejection_sampling(pdf,n=1000,xmin=0,xmax=1):

             # Calculates the minimal and maximum values of the PDF in the desired
             # interval. The rejection method needs these values in order to work
             # properly.

             x = np.linspace(xmin,xmax,1000)
             y = pdf(x)
             pmin = 0.
             pmax = y.max()

             # Counters
             naccept = 0
             ntrial = 0

             # Keeps generating numbers until we achieve the desired n
             ran = [] # output list of random numbers
             while naccept<n:
                 x = np.random.uniform(xmin,xmax) # x'
                 y = np.random.uniform(pmin,pmax) # y'

                 if y<pdf(x):
                     ran.append(x)
                     naccept=naccept+1
                 ntrial=ntrial+1

             ran=np.asarray(ran)

             return ran,ntrial, pmax
```

1.3.1 3.1. Beta distribution:

Consider $X \sim Beta(\alpha, \beta)$. Define a function `sample_from_beta()` which takes as input the parameters `alpha`, `beta` and a integer `N` and returns a sample of length N. As before, check the fitness with the KS test.

```
In [15]: a, b = 2 , 5
         beta = lambda x: scipy.stats.beta.pdf(x, a, b)

         ran, ntrials,pmax = rejection_sampling(beta, n = 10000)

In [16]: xx = np.linspace(0,1,1000)
         plt.hist(ran, density = True, bins = 50)
         plt.plot(xx, beta(xx))
         plt.hlines(pmax, xmin = 0, xmax = 1)
         plt.show()

         print(ntrials)
```




```
24418
```

3.2. Improve computational efficiency: In order to reduce the overall number of trials required to obtain a sample of size N, find a more refined function that majorize the Beta distribution of exercise 3.1.

Optional: implement you refined solution and show the improvement in time w.r.t. N: i.e. time to the solution for increasing values of N for the simple case and the refined one, both in terms of reduced rejection rate and overall sampling time.

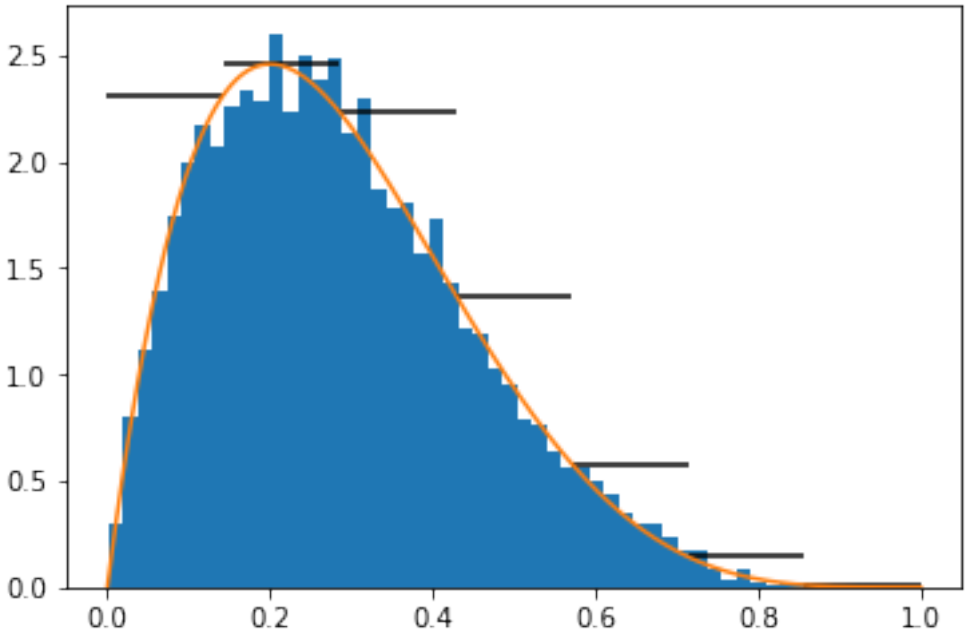
```
In [17]: def improved_rejection_sampling(pdf,n=1000, k = 10, xmin=0,xmax=1):
             edges = np.linspace(xmin,xmax,k+1)
             naccept = 0
```

```
    ntrial = 0
    prob = []
    pmin = 0.
    for i in range(k):
        x = np.linspace(edges[i],edges[i+1],100)
        y = pdf(x)
        pmax = y.max()

        prob.append(pmax)
        S = np.cumsum(prob)
        # Keeps generating numbers until we achieve the desired n
        ran = [] # output list of random numbers
    while naccept<n:
        u = np.random.uniform(xmin,xmax)
        for i in range(k):
            if S[i] > u:
                sampled_index = i
                break

        y = np.random.uniform(pmin,prob[sampled_index]) # y'

        if y<pdf(u):
            ran.append(u)
            naccept=naccept+1
        ntrial=ntrial+1

    ran=np.asarray(ran)

    return ran,ntrial, prob
```

In [18]:
```
k = 7
ran, ntrials,prob = improved_rejection_sampling(beta, k = k,  n = 10000)
print(ntrials)
```

```
23169
```

In [19]:
```
edges = np.linspace(0,1,k+1)
plt.hist(ran, density = True, bins = 50)
plt.plot(xx, beta(xx))
for i in range(k):
    plt.hlines(prob[i], xmin = edges[i], xmax = edges[i+1])
plt.show()
```



In [ ]:

In [ ]: