EEE554 – Random Signal Theory

Final Project Report

Implementation and analysis of rejection (Ziggurat) and baseline sampling algorithms

Mishel Jyothis Paul

Task 1: Proof why rejection sampling method could work

For sampling from $f_s(x)$, it is easier to sample from another distribution $h(x)$ such that $f_s(x)$ is bounded by

$$Mh(x) > f_s(x).$$

A sample is accepted if it lies under the envelope of f_s.

Let A be the event that a sample is accepted.

$P(A)$ = probability that a point under area $Mh(x) < f_s(x)$

$$= \int_{-\infty}^{\infty} \int_{0}^{\frac{f_s(x)}{Mh(x)}} h(x) \cdot dy \, dx = \int_{-\infty}^{\infty} \frac{f_s(x)}{Mh(x)} \cdot h(x) \, dx$$

$$= \frac{1}{M} \int_{-\infty}^{\infty} f_s(x) \, dx = \underline{\underline{\frac{1}{M}}}$$

For rejection sampling method to work, $f(x|A)$ should follow $f_s(x)$.

$$f(x|A) = \frac{P(A|x) \cdot f(x)}{P(A)} \quad \text{[Baye's rule]}$$

$$P(A|x) = \frac{f_s(x)}{Mh(x)}.$$

Substituting,

$$f(x|A) = \frac{\frac{f_s(x)}{Mh(x)} \cdot h(x)}{1/M} = \underline{\underline{f_s(x)}}$$

Task 2: Proof of correctness of Baseline sampling

For baseline method to work, for a uniform distribution $U \sim Unf(0,1)$, $F^{-1}(U)$ shout have $F(x)$ as its CDF.

Proof:

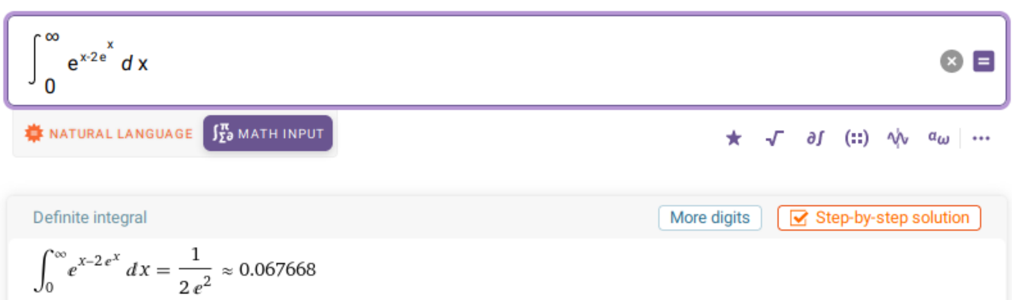
$$P(F^{-1}(u) \le x) \qquad [\text{CDF of } F^{-1}(u)]$$

$$= P[F(F^{-1}(u)) \le F(x)] \qquad [\text{applying } F \text{ on both sides of inequality}]$$

$$= P[u \le F(x)] \qquad [F(F^{-1}(x)) = x]$$

$$= \underline{\underline{F(x)}} \qquad [\text{For a uniform distribution, } P[u \le x] = x]$$

Task 3: Normalized PDF



$C = 2e^2 = 14.778$

This has been implemented in code as follows:

```
## PDF function (monotonically decreasing, not normalized
gx = lambda x: math.exp(x-2*math.exp(x))
func_limits = (0,3)

## Calculate normalized PDF
c = integrate.quad(gx, *func_limits)[0]
func = lambda x: gx(x)/c
```

Task 4: Function to compute CDF of distribution

Python library function scipy.integrate.quad was used to define the CDF of the given PDF.

```
def cdf(func, x, low_lim):
    return integrate.quad(func, low_lim, x)[0]
```

Task 5: Baseline sampling implementation

Baseline sampling was implemented using the bisection algorithm to find the inverse CDF as the given CDF has no closed form inverse. Numpy.random.uniform was used to generate uniformly distributed values for the CDF. A tolerance of 10^{-12} was used for accuracy of the calculated inverse CDF value to the generated random value.

Inverse CDF calculation:

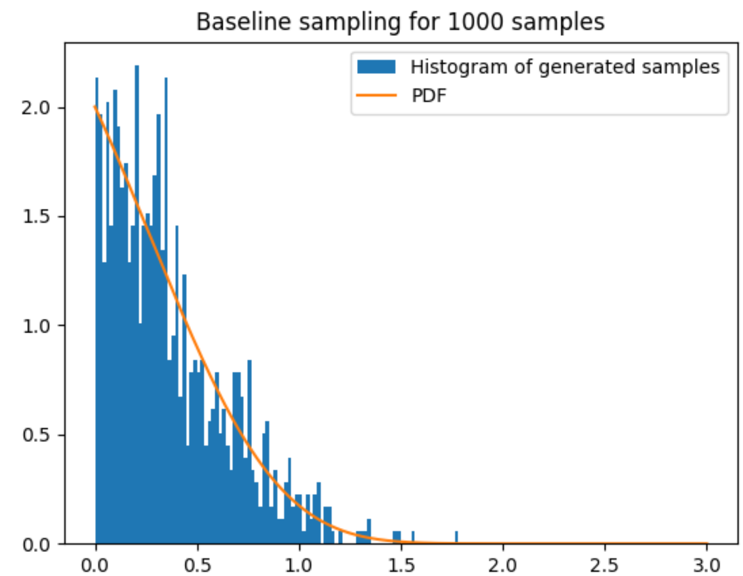
```
    def inv_cdf(func, u, limits):
        xmin, xmax = limits
        midpoint = (xmin+xmax)/2
        cdf_val = cdf(func, midpoint, limits[0])
        while abs(cdf_val-u)>tol:
            if cdf_val>u:
                xmax=midpoint
            else:
                xmin=midpoint
            midpoint = (xmin+xmax)/2
            cdf_val = cdf(func, midpoint, limits[0])
        return midpoint
```

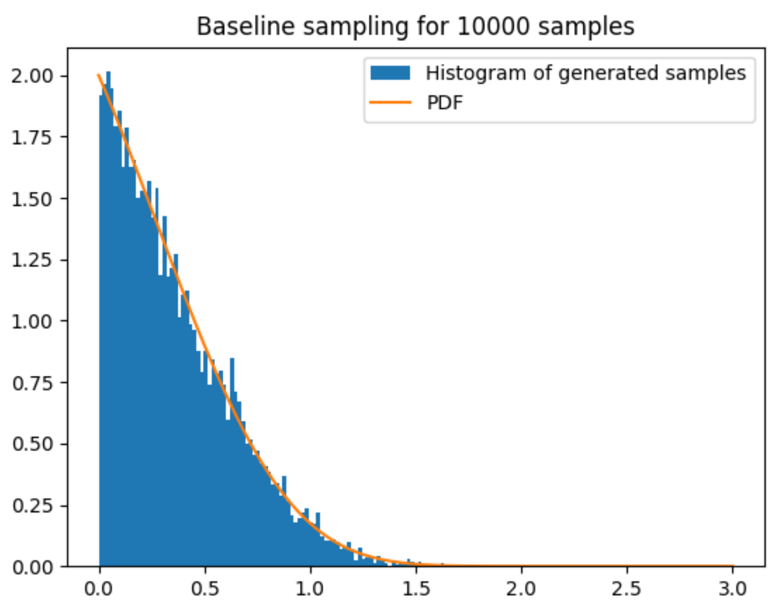
Sample generation:

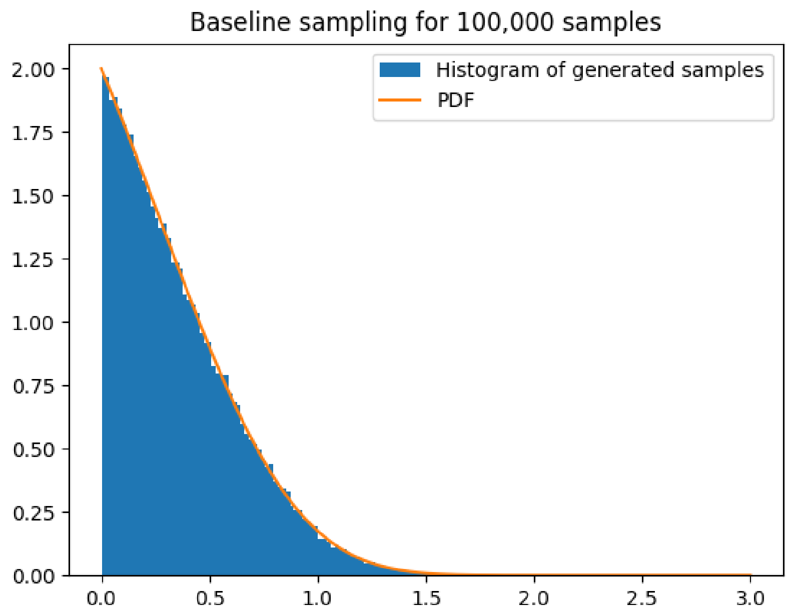
```
    while len(samples)<num_samples:
        u = unif(0,1)
        samples.append(inv_cdf(func, u, limits))
```

Task 6: Sampling using baseline function

Samples generated using baseline sampling were found to closely match the PDF. The accuracy of the algorithm was found to increase with an increase in the number of samples.







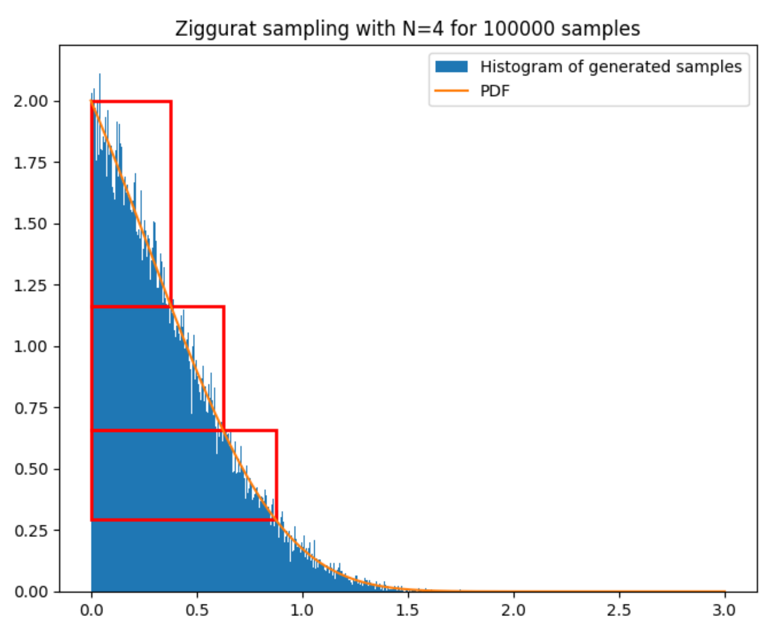
Task 7: Ziggurat setup

Bisection algorithm is used to find the X and Y coordinates corresponding to the guess area.

Calculated values for n=4:

X values: 0, 0.3789, 0.6262, 0.8745

Y values: 2.0, 1.1626, 0.6559, 0.2930



Code to generate X and Y coordinates for any N:

```
def get_points(func, limits, num_segments, area_tol=10e-12):
    def get_next_point(func, limits, start_x, ref_area, area_tol=10e-16):
        xmin, xmax = start_x, limits[1]
        midpoint = (xmin+xmax)/2
        mid_area = rect_area((limits[0], func(start_x)), (midpoint, func(midpoint)))
        while abs(ref_area-mid_area)>area_tol:
            midpoint = (xmin+xmax)/2
            mid_area = rect_area((limits[0], func(start_x)), (midpoint, func(midpoint)))
            if ref_area>mid_area: xmin = midpoint
            elif ref_area<mid_area: xmax = midpoint

            if midpoint == limits[1]: return None, None

        return (xmin+xmax)/2, mid_area

    total_area = integrate.quad(func, *limits)[0]
    ref_area = total_area/num_segments
    area_max = total_area
    area_min = 0
    nth_area = 0
    nth_rect_probability = 0

    x_list = [limits[0]]
    area_list = list()

    while abs(nth_area-ref_area)>area_tol:
        x_list = [limits[0]]
        area_list = []
        for _ in range(num_segments-1):
            x_point, area = get_next_point(func, limits, x_list[-1], ref_area)
            if not x_point: continue
            x_list.append(x_point)
            area_list.append(area)
        nth_area = rect_area((limits[0], 0), (x_list[-1], func(x_list[-1]))) + integrate.quad(func, x_list[-1], limits[1])[0]
        nth_rect_probability = rect_area((limits[0], 0), (x_list[-1], func(x_list[-1]))) / nth_area
        area_list.append(nth_area)

        if nth_area>ref_area:
            area_min = ref_area
        elif ref_area>nth_area:
            area_max = ref_area
        ref_area = (area_min+area_max)/2

    y_list = [func(x) for x in x_list]
    return x_list, y_list, nth_rect_probability
```

Task 8: Probability of point falling in tail of distribution

Probability of point falling in the rectangular region of tail distribution can be calculated as a ratio of the corresponding areas.

P(point is in rectangular part of nth region for N=4) = 0.8074

Task 9: Implementation of sampling from tail

Baseline sampling was used to obtain samples from the tail area of the distribution. No significant degradation in performance was observed. Implementation using Ziggurat algorithm is suggested from a performance perspective. However, it was found that the algorithm tends to go into an infinite loop due to issues with the random integer generator. This repeatedly calls the tail distribution until maximum recursion depth is reached. Using a baseline sampling algorithm instead was found to be more robust.

Code used:

```
samples.append(baseline.baseline(func, (x_list[-1], func_limits[1]), 1))
#samples.append(ziggurat(func, (x_list[-1], func_limits[1]), 1, num_segments))
```

Task 10: Ziggurat implementation for n=4, 32 and 256

Ziggurat implementation was done as two stages – a precompute stage that calculates the X, Y coordinates and a sampling stage that generates the required number of samples. During implementation, it was found that random integer generation built-ins in Python are inherently slow (mostly attributed to implementations in Python itself and not in C). To mitigate this behavior, random.randint was replaced with int(random.random()*num), which improved the performance drastically. There are no separate implementations for n=4,32 & 256. The implementation can handle any arbitrary value for number of segments.

Precompute stage:

```
## Precompute stage
x_list, y_list, nth_rect_probability = get_points(func, func_limits, num_segments, area_tolerance)
```

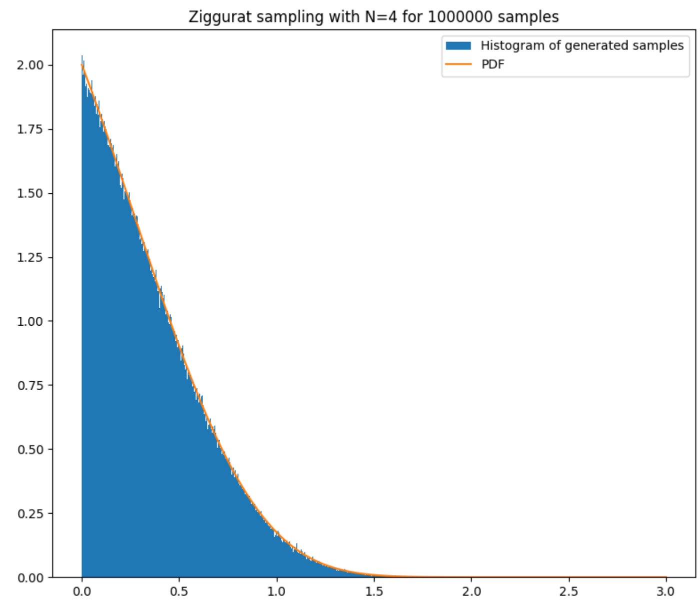
Sampling stage:

```
## Sampling stage
while(len(samples)<num_samples):
    time_s = time()
    k = int(random.random()*num_segments)
    time_random += (time()-time_s)
    if k<num_segments-1:
        time_s = time()
        x = unif(x_list[0], x_list[k+1])
        time_random += (time()-time_s)
        if x<x_list[k]:
            stats['X is accepted because X<xk'] += 1
            samples.append(x)
            continue
        else:
            time_s = time()
            y = unif(y_list[k], y_list[k+1])
            time_random += (time()-time_s)
            if y<func(x):
                stats['X>xk but X is accepted because Y<g(X)'] += 1
                samples.append(x)
                continue
            else:
                stats['Y>g(X) so X is rejected'] += 1
    else:
        rect_select = unif(0,1)
        if rect_select<nth_rect_probability:
            stats['Sample drawn from rectangular part of nth region'] += 1
            samples.append(unif(x_list[0], x_list[-1]))
        else:
            stats['Tail algorithm is run'] += 1
            samples.append(baseline.baseline(func, (x_list[-1], func_limits[1]), 1))
            #samples.append(ziggurat(func, (x_list[-1], func_limits[1]), 1, num_segments))
```

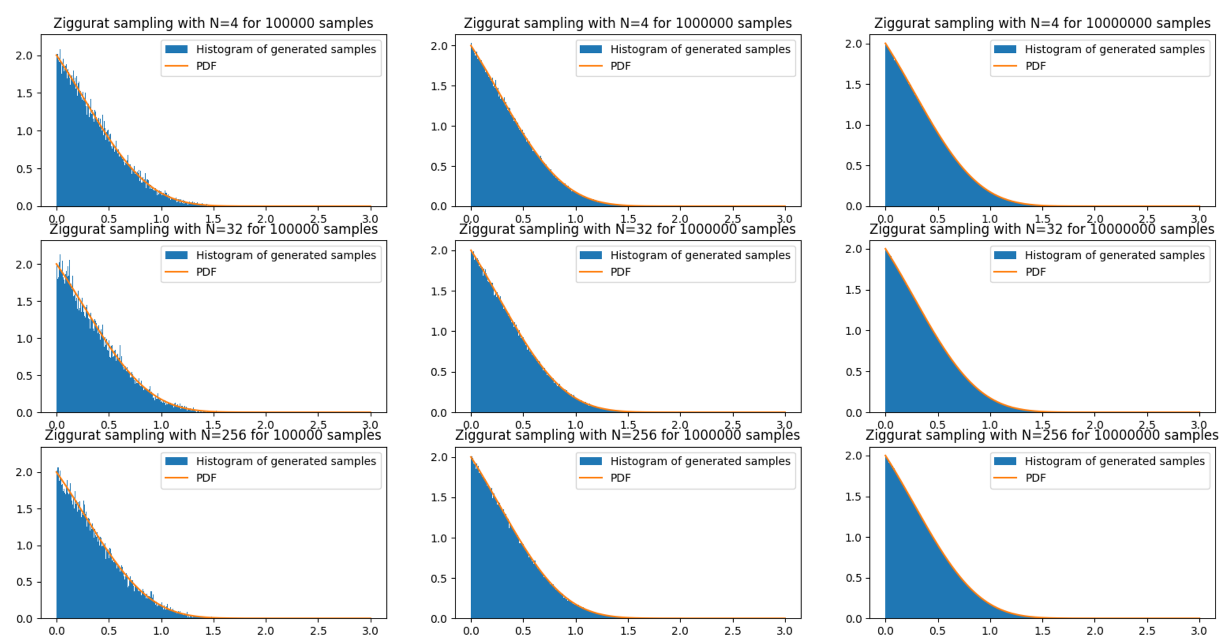
Task 11: Sampling using Ziggurat algorithm

Generated samples were found to be closely matching with the given PDF. Results were found to be more accurate for a higher number of generated samples.

Results for 1,000,000 samples with N=4:



Improvement in accuracy with increase in number of samples:



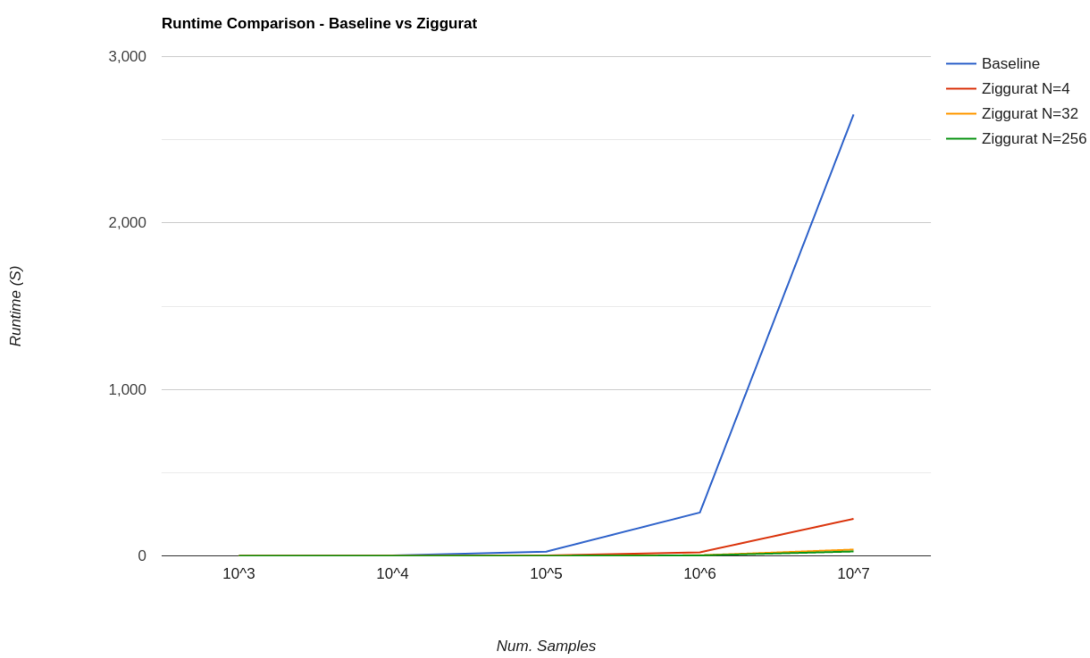
Task 12: Analysis

In all cases, Ziggurat algorithm was found to be faster than baseline sampling method. Performance was found to improve significantly with an increase in the number of segments used, especially for a higher number of samples.

Runtime Comparison

All runtimes measured in seconds.

Num. Samples	Baseline	Ziggurat (N=4)	Ziggurat (N=32)	Ziggurat (N=256)
1000	0.26	0.021	0.004	0.003
10,000	2.58	0.022	0.036	0.026
100,000	25.83	2.20	0.38	0.27
1000,000	260.89	22.10	3.74	2.66
10,000,000	2651.53	223.12	37.81	26.70



Time per sample:

Algorithm	Time (µS)
Baseline	260
Ziggurat (N=4)	22.31
Ziggurat (N=32)	3.78
Ziggurat (N=256)	2.67

Occurrences of outcomes in rejection loop for 10M samples

Outcome	N = 4	N = 32	N = 256
X is accepted because X < xk	418714	914295	987341
X > xk but X is accepted because Y < g(X)	263546	52567	8717
Y > g(X) so X is rejected	269066	52999	8810
k = n and sample is drawn from rectangular part of nth region	256522	30425	3722
k= n and the tail algorithm is run	61218	2713	220

Task 13: Performance improvements

Runtime of the Ziggurat algorithm can be further improved by the following methods:

- Removing normalization and any other constant float operations from the PDF function: Ziggurat algorithm works irrespective of the area under the graph of the given PDF. The runtime can be improved by removing these operations.
- The random integer generation for picking the segments is consuming majority (70%+) of the runtime in the performed experiments in python. This can be brought outside the sampling loop. Since the uniform sampling algorithm used does not have any dependency on the previously sampled value, this is an ideal candidate for parallel execution. The code can be adapted to work across multiple cores.