Lloyd Max quantization algorithm implementation code

Submitted in partial fulfilment of Digital Communication (ELL712) course

Name: Dibyajyoti Jena

Entry Number: 2022EEE2712

Language: Python

Environment: Jupyter Notebook

Advantages of Jupyter Notebook: Blockwise code execution, edit and make reports alongside code for explaination

Lloyd Max quantization algorithm

It is an iterative algorithm to find boundary points and representation points, for a given input distribution, so as to minimize mean squared error.

Implementation steps:

1. Create appropriate distributions(Gaussian, Uniform, Rayleigh).
2. For each distribution, define random representation points a_j.
3. Find out boundary points, mean of consecutive representation points $b_{j+1} = \frac{a_j + a_{j+1}}{2}$
4. Find out new representation point a_j by taking conditional mean of the input pdf between b_j and b_{j+1}.

$$a_{jnew} = \frac{\int_{b_j}^{b_{j+1}} f_U(u).du}{Q(u)}$$

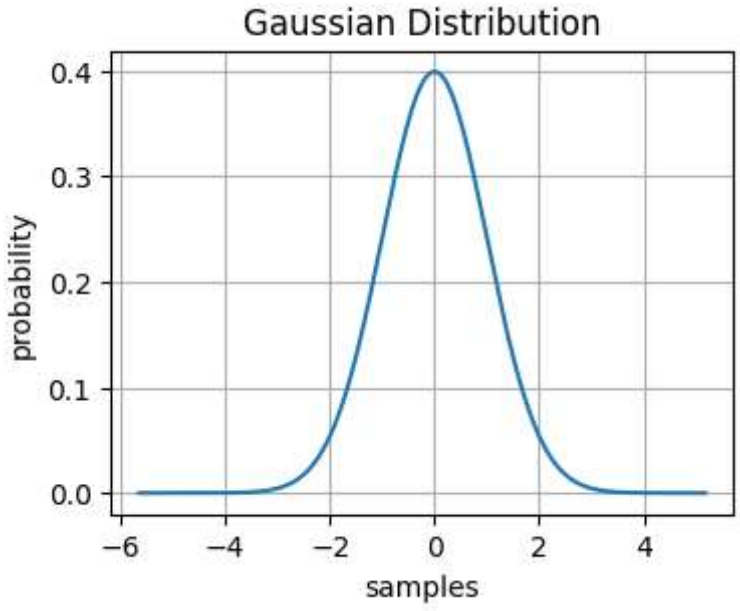
5. Find out and display MSE for a batch of inputs.
6. Repeat steps 3 and 4 until there is no decrement in MSE.

In [1]:
```
###.....................Actual Program Starts Here...................
##.....................................................................
#...................Import useful libraries to be used..............



import math           ## basic math operations
import numpy as np    ## numerical package for array operations
import tqdm as tqdm   ## loop timer; helps in keeping a track on loops
from scipy.integrate import quad ## definite integration function
import matplotlib.pyplot as plt  ## plotting
```

Defining Gaussian pdf

In [61]:
```
mean = 0
std_dev = 1
var = std_dev**2
gsample = np.random.normal(loc=mean, scale=std_dev, size=10000000)
## random.normal is a package that generates samples from normal distribution
## taking 10 million samples; loc = mean; scale = std dev; size = number of sampl
gsample.sort()

gaussian = (np.exp(-((gsample-mean)**2)/(2*var)))/(2*np.pi*var)**0.5
## GAUSSIAN pdf

plt.figure(figsize = (4,3))
plt.plot(gsample, gaussian)
plt.xlabel("samples")
plt.ylabel("probability")
plt.title("Gaussian Distribution")
plt.grid()
print("End points are: ",gsample[0], gsample[-1])
```

```
End points are:  -5.644475287043961 5.174982879663277
```



In [4]:
```
## check if probability integrates to 1
## only then I can be sure my Gaussian function is valid

def gaussian_func(x, mean = 0, var = 1):
    return (np.exp(-((x-mean)**2)/(2*var)))/(2*np.pi*var)**0.5

I = quad(gaussian_func,-np.inf,np.inf) ## calling integration function
I
```

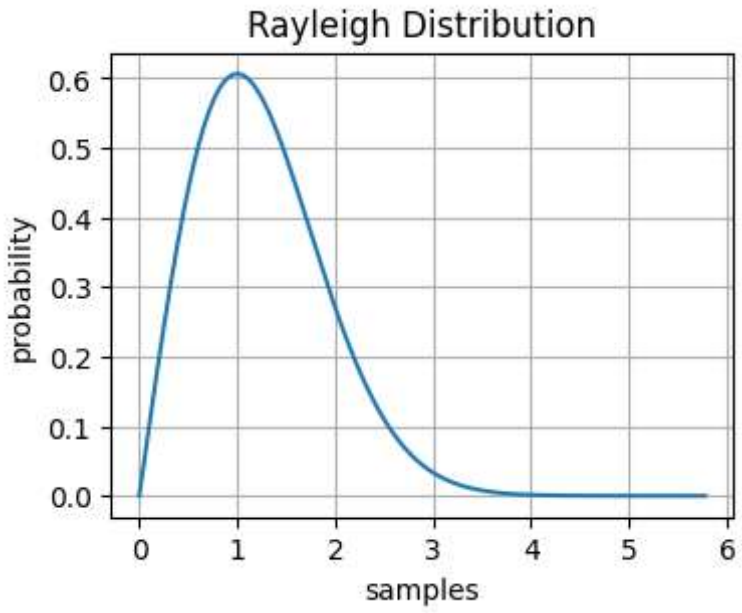
Out[4]:
```
(0.9999999999999998, 1.017819132089219e-08)
```

Integration function integrate.quad returns 2 values, integration answer and its error percentage. The above gaussian pdf is valid, since total probability is close to one.

Defining Rayleigh pdf

In [62]:
```
mode = 1
rsample = np.random.rayleigh(scale=mode, size=10000000) # 10 million samples
## scale = mode, where the distribution peaks
## size = number of samples
rsample.sort()

rayleigh = rsample*np.exp(-((rsample/mode)**2)/2)/mode**2

plt.figure(figsize = (4,3))
plt.plot(rsample, rayleigh)
plt.xlabel("samples")
plt.ylabel("probability")
plt.title("Rayleigh Distribution")
plt.grid()
print("End points are: ",rsample[0], rsample[-1])
```

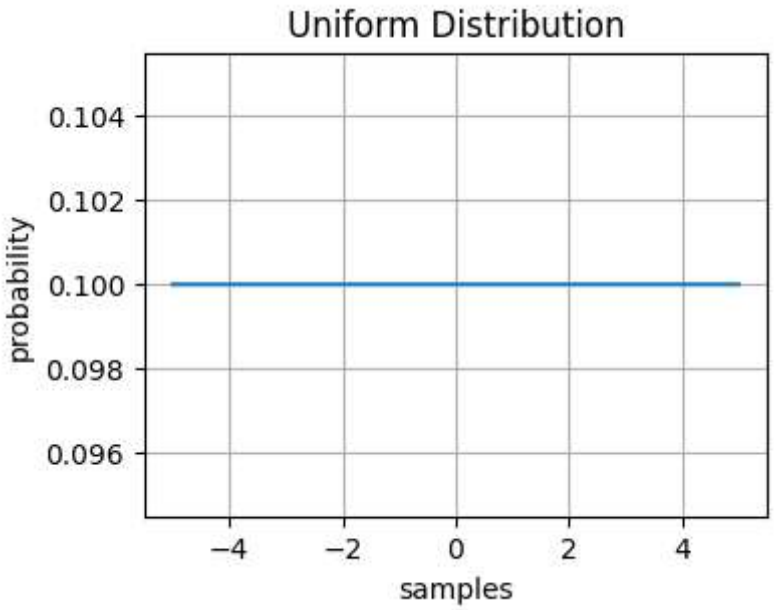
```
End points are:  0.0004923817390603047 5.777782664060156
```



Defining Unifrom pdf

In [63]:
```
start = -5 ## start point
end = 5  ## end point
usample = np.random.uniform(low = start,high = end, size=10000000)
## random.uniform generates random uniform samples
## 10 million samples
usample.sort()

uniform = np.ones(10000000)*1/(end-start)

plt.figure(figsize = (4,3))
plt.plot(usample, uniform)
plt.xlabel("samples")
plt.ylabel("probability")
plt.title("Uniform Distribution")
plt.grid()
print('End points are: ', usample[0], usample[-1])
```

```
End points are:  -4.999999988922573 4.9999976458205655
```



Next, we define representation points a_j and boundary points b_j for each given pdf.

In [7]:
```
## define representation points a
## define boundaries b
##.............................GAUSSIAN REPRESENTATION POINTS.................

def finding_a_gaussian(b):
    def gaussian_prob(x, mean = 0, var = 1):
        return (np.exp(-((x-mean)**2)/(2*var)))/(2*np.pi*var)**0.5
    ## function to find probability

    def gaussian_mean(x, mean = 0, var = 1):
        return  (x*(np.exp(-((x-mean)**2)/(2*var))))/(2*np.pi*var)**0.5
    ## function to find mean

    a = []
    for n in range(len(b)-1):
        prob = quad(gaussian_prob,b[n],b[n+1])[0] ##conditional probability
        cond_exp = quad(gaussian_mean,b[n],b[n+1])[0]  ## expectation
        cond_exp = cond_exp/prob                  ## conditional expectation
        a.append(cond_exp)

    return a

##.............................RAYLEIGH REPRESENTATION POINTS.................
def finding_a_rayleigh(b):
    def rayleigh_prob(x, mode = 1):
        return x*np.exp(-((x/mode)**2)/2)/mode**2
    def rayleigh_mean(x, mode = 1):
        return  (x**2)*np.exp(-((x/mode)**2)/2)/mode**2
    a = []
    for n in range(len(b)-1):
        prob = quad(rayleigh_prob,b[n],b[n+1])[0]
        cond_exp = quad(rayleigh_mean,b[n],b[n+1])[0]
        cond_exp = cond_exp/prob
        a.append(cond_exp)

    return a

##.............................UNIFORM REPRESENTATION POINTS.................
def finding_a_uniform(b):
    def uniform_prob(x, start = -5, end = 5):
        return (1/(end-start))
    def uniform_mean(x, mode = 1):
        return  (x/(end-start))
    a = []
    for n in range(len(b)-1):
        prob = quad(uniform_prob,b[n],b[n+1])[0]
        cond_exp = quad(uniform_mean,b[n],b[n+1])[0]
        cond_exp = cond_exp/prob
        a.append(cond_exp)

    return a


##.......................BOUNDARY POINTS.............................
## THIS FUNCTION REMAINS SAME FOR ALL
```

```
def finding_b(a, oldb):
    b = []
    b.append(oldb[0])
    for n,i in enumerate(a[:-1]):
        b.append((i+a[n+1])/2)
    b.append(oldb[-1])
    return b
```

Finding MSE: $MSE = \frac{\sum_M (u_j - a_j)^2}{M}$

where u_j is input and a_j is representation point, M is number of inputs in a batch.

In [8]:
```
## MSE
def find_nearest_element_index(arr,n):
    diff_array = np.absolute(arr-n)
    index = diff_array.argmin()
    return(index)
def finding_mse(u,aj):
    mse = 0
    for i in u:
        element = aj[find_nearest_element_index(aj,i)]
        error = (element-i)**2
        #print(element, error)
        mse = mse + error
    mse = mse/(len(u))
    return mse
```

Lloyd Max iterations

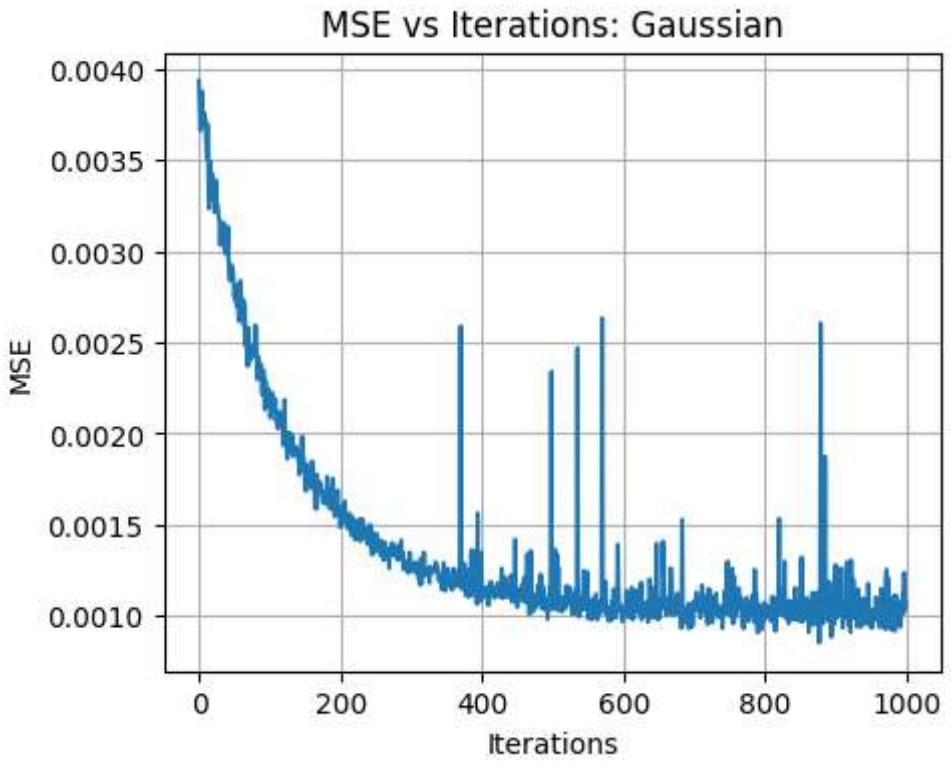
1. Gaussian pdf

In [64]:
```
## set boundary points be 51, representation points be 50
number = 51
## boundary points b can be random or uniform
## it does not really matter since in both cases they will converge on the same r
## uniformly set b

b = np.linspace(gsample[0], gsample[-1],number)
a = finding_a_gaussian(b)
## first setting boundary and then representation point for convenience

iters = 1000             ### 1000 iterations
MSE = []
for i in tqdm.tqdm(range(iters)):
    Uin = np.random.choice(gsample,1000)
    ## batch size 1000 taken from gaussian sample
    mse = finding_mse(Uin,a)
    MSE.append(mse)
    a = finding_a_gaussian(b)
    b = finding_b(a,b)
```

```
100%|██████████| 1000/1000 [00:26<00:00, 37.97it/s]
```

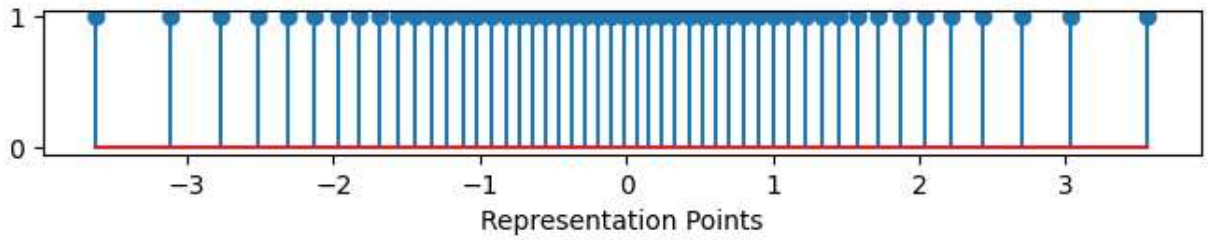
In [65]:
```
plt.figure(figsize = (5,4))
plt.plot(MSE)
plt.xlabel('Iterations')
plt.ylabel("MSE")
plt.title("MSE vs Iterations: Gaussian")
plt.grid()
print("Last MSE reached: ", MSE[-1])
```

```
Last MSE reached:  0.001078976509485901
```



In [66]:
```
## visualise representation points
one = np.ones(len(a))
plt.figure(figsize = (8,1))
plt.stem(a,one)
plt.xlabel('Representation Points')
```

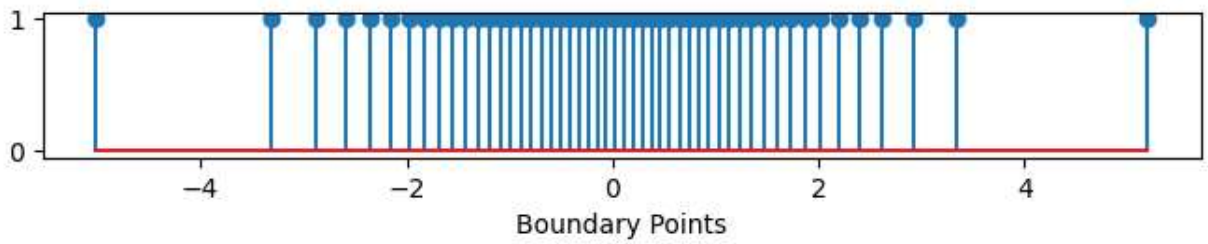
Out[66]:
```
Text(0.5, 0, 'Representation Points')
```



To be noticed: after running Lloyd Max, the optimal representation points have somewhat followed the distribution itself, i.e. densely packed representation points at places of high probability, sparse representation points in places of low probability.

In [12]:
```
## visualise boundary points
one = np.ones(len(b))
plt.figure(figsize = (8,1))
plt.stem(b,one)
plt.xlabel('Boundary Points')
```

Out[12]:
```
Text(0.5, 0, 'Boundary Points')
```



One can see the representation points and boundary points follow the corresponding pdf.

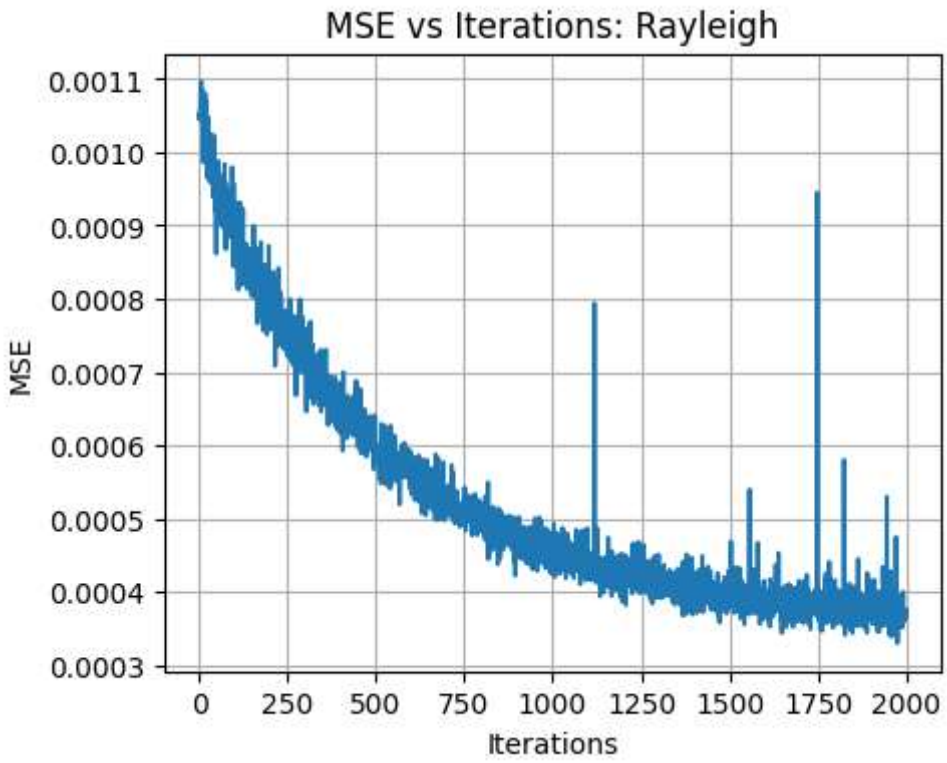
2. Rayleigh Pdf

In [67]:
```
## initialization
number = 51  ## number of boundary points
b = np.linspace(rsample[0],rsample[-1],number)
a = finding_a_rayleigh(b)

iters = 2000
batch_size = 1000
MSE = []
for i in tqdm.tqdm(range(iters)):
    Uin = np.random.choice(rsample,batch_size)
    mse = finding_mse(Uin,a)
    MSE.append(mse)
    a = finding_a_rayleigh(b)
    b = finding_b(a,b)
```

```
100%|██████████| 2000/2000 [00:54<00:00, 36.80it/s]
```

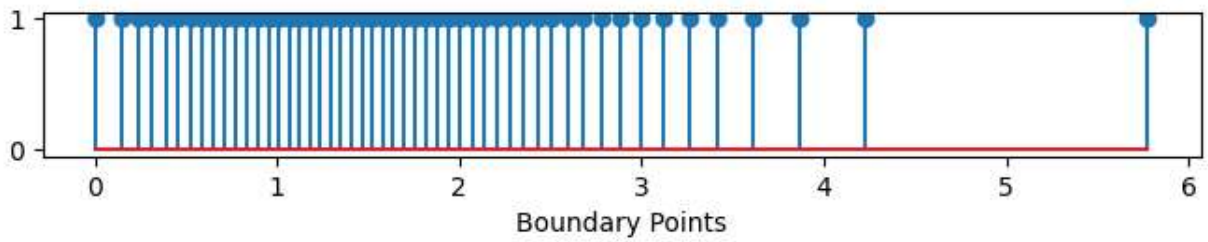
In [68]:
```
plt.figure(figsize = (5,4))
plt.plot(MSE)
plt.xlabel('Iterations')
plt.ylabel("MSE")
plt.title("MSE vs Iterations: Rayleigh")
plt.grid()
print("Last MSE reached: ", MSE[-1])
```

```
Last MSE reached:  0.00037494240888826587
```



In [69]:
```
## visualise boundary points
one = np.ones(len(b))
plt.figure(figsize = (8,1))
plt.stem(b,one)
plt.xlabel('Boundary Points')
```

Out[69]: Text(0.5, 0, 'Boundary Points')



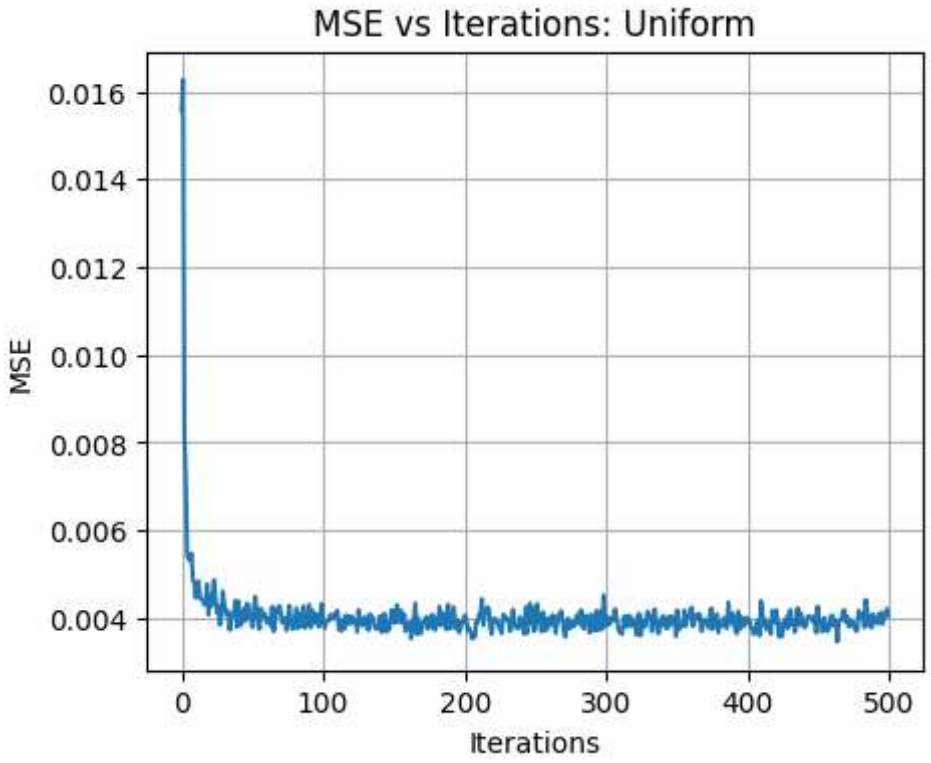
3. Unifrom PDF

In [70]:
```
## initialization
number = 51  ## number of boundary points
b = np.random.choice(usample,number)
b.sort()
a = finding_a_uniform(b)

iters = 500
batch_size = 1000
MSE = []
for i in tqdm.tqdm(range(iters)):
    Uin = np.random.choice(usample,batch_size)
    mse = finding_mse(Uin,a)
    MSE.append(mse)
    a = finding_a_uniform(b)
    b = finding_b(a,b)
```

100%| 500/500 [00:09<00:00, 53.45it/s]

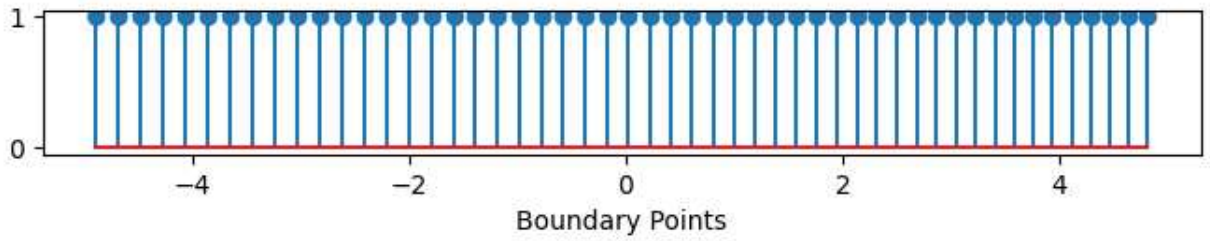
In [71]:
```
plt.figure(figsize = (5,4))
plt.plot(MSE)
plt.xlabel('Iterations')
plt.ylabel("MSE")
plt.title("MSE vs Iterations: Uniform")
plt.grid()
print("Last MSE reached: ", MSE[-1])
```

```
Last MSE reached:  0.004060733806337642
```



In [72]:
```
## visualise boundary points
one = np.ones(len(b))
plt.figure(figsize = (8,1))
plt.stem(b,one)
plt.xlabel('Boundary Points')
```

Out[72]:
```
Text(0.5, 0, 'Boundary Points')
```



Stochastic Gradient Descent

Stochatic Gradient Descent is a popular valley finding algrithm which works by minimizing the cost function for a given problem. The problem is optimized when cost function is minimized. Given this logic, and the cost function in our case is MSE, we set out to minimize mse, in other words, optimizing the representation points.

Stochastic gradient descent in this case works similar to Lloyd Max, but updating representation points is done by the following formula:

$$a_{new} = a_{old} - \alpha \frac{\partial[MSE]}{\partial a_j}$$

where gradient in the name suggests partial differentiation of MSE with respect to the corresponding a_j.

Here α is the learning rate, which can be fixed or variable. I have chosen to use a variable learning rate because it gives the same results in lesser number of iterations.

Learning Rate:

$$\alpha = 0.02exp(-0.0002index)$$

where index is iteration number Learning rate starts out at 0.02 and slowing decreases as the iterations increase.

We don't want the learning rate to stay large throughout the iterations, else the corrections become too high and chances are the algorithm will overshoot the minimum point.

Likewise the learning rate should not be too low else the algorithm will require too many iterations to reach a minima.

In [24]:
```
def grad_descent_a(u,a,index):
    lrate = 0.025*np.exp(-0.0002*index)          ## Learning rate
    for i in u:
        index = find_nearest_element_index(a,i)
        ## Finding the nearest representation point's index

        element = a[index]
        error = 2*(element-i)
        a[index] = a[index] - lrate*error
        ## Updating representation point

    return a
```

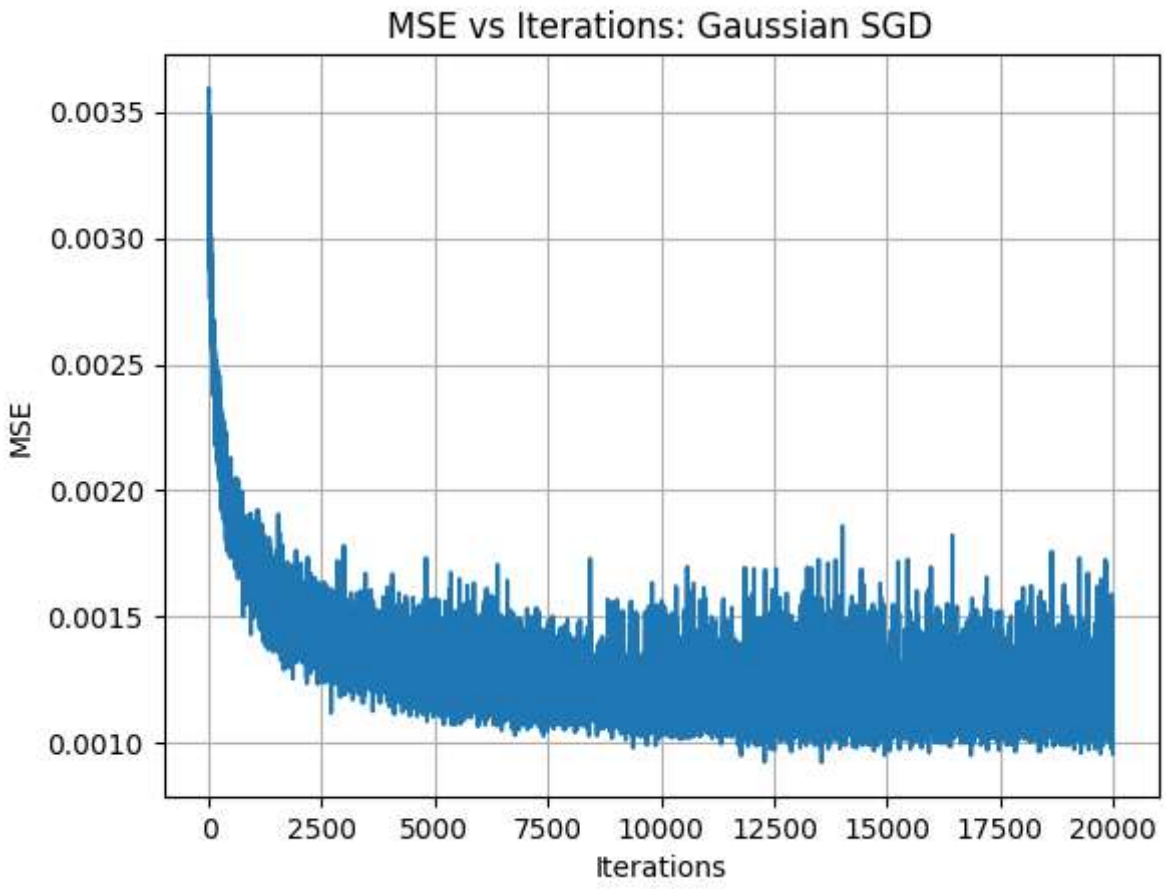
1. Gaussian pdf

In [54]:
```
number = 51
## INITIALIZING a and b
b = np.linspace(gsample[0], gsample[-1], number)
a = finding_a_gaussian(b)
#a = []
#for i in range(len(b)-1):
#    temp = (b[i]+b[i+1])/2
#    a.append(temp)

iters = 20000
MSE = []

for i in tqdm.tqdm(range(iters)):
    Uin = np.random.choice(gsample,1000)  ## batch size 1000
    mse = finding_mse(Uin,a)
    MSE.append(mse)
    a = grad_descent_a(Uin,a,i)
    b = finding_b(a,b)
```

```
100%|████████████████████████████████████████████████████████████| 20000/20000 [11:24<00:00, 29.21it/s]
```

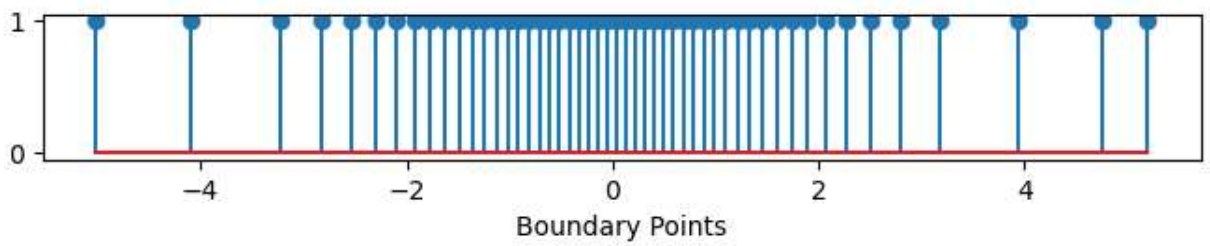
In [55]:
```
plt.plot(MSE)
plt.xlabel('Iterations')
plt.ylabel("MSE")
plt.title("MSE vs Iterations: Gaussian SGD")
plt.grid()
print("Last MSE reached: ", MSE[-1])
```

```
Last MSE reached:  0.0009559541101245054
```



In [56]:
```
## visualise boundary points
one = np.ones(len(b))
plt.figure(figsize = (8,1))
plt.stem(b,one)
plt.xlabel('Boundary Points')
```

Out[56]:
```
Text(0.5, 0, 'Boundary Points')
```



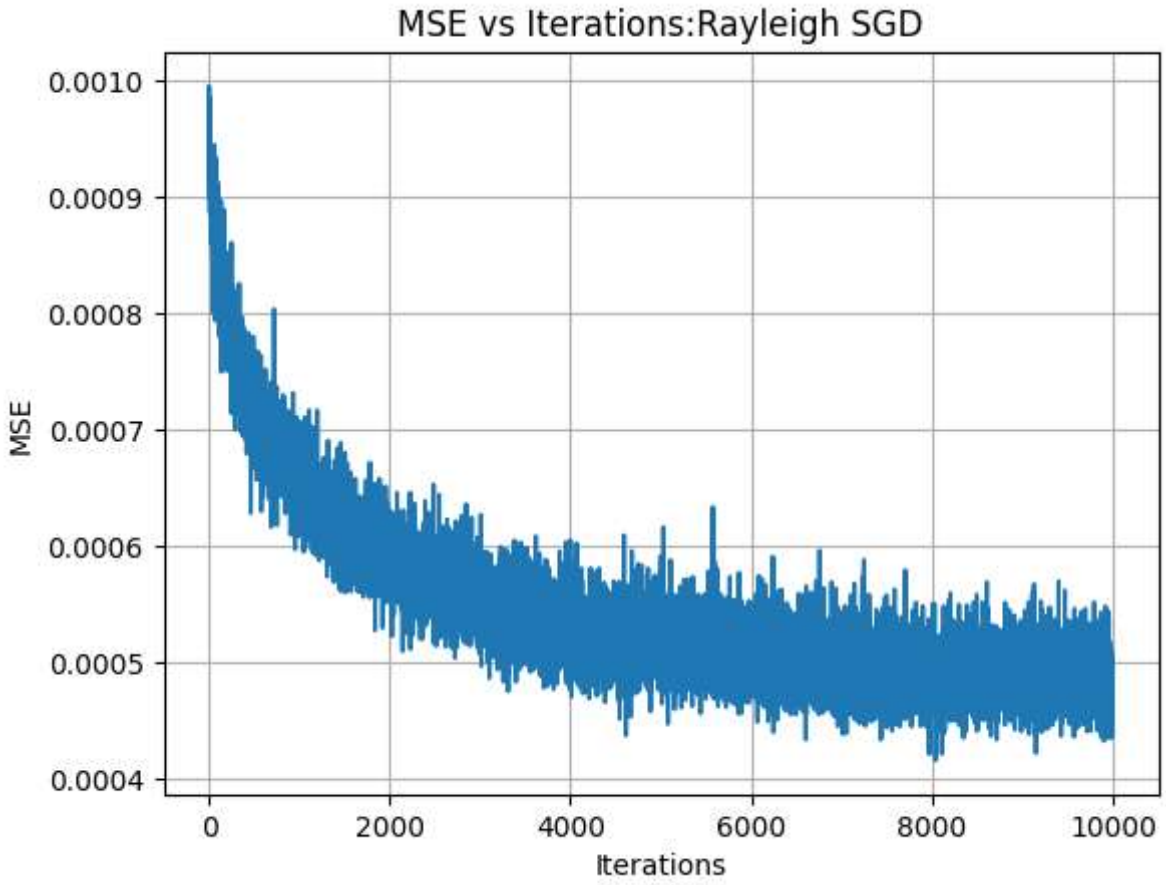
2. Rayleigh pdf

In [48]:
```
number = 51
## INITIALIZING a and b
b = np.linspace(rsample[0], rsample[-1], number)
a = finding_a_rayleigh(b)

iters = 10000
MSE = []

for i in tqdm.tqdm(range(iters)):
    Uin = np.random.choice(rsample,1000)  ## batch size 1000
    mse = finding_mse(Uin,a)
    MSE.append(mse)
    a = grad_descent_a(Uin,a,i)
    b = finding_b(a,b)
```

```
100%|██████████| 10000/10000 [02:41<00:00, 61.92it/s]
```

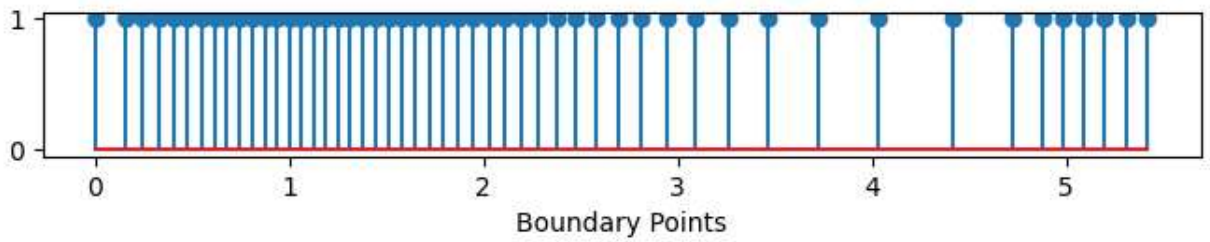
In [49]:
```
plt.plot(MSE)
plt.xlabel('Iterations')
plt.ylabel("MSE")
plt.title("MSE vs Iterations:Rayleigh SGD")
plt.grid()
print("Last MSE reached: ", MSE[-1])
```

```
Last MSE reached:  0.0004941244521086678
```



In [50]:
```
## visualise boundary points
one = np.ones(len(b))
plt.figure(figsize = (8,1))
plt.stem(b,one)
plt.xlabel('Boundary Points')
```

Out[50]: Text(0.5, 0, 'Boundary Points')



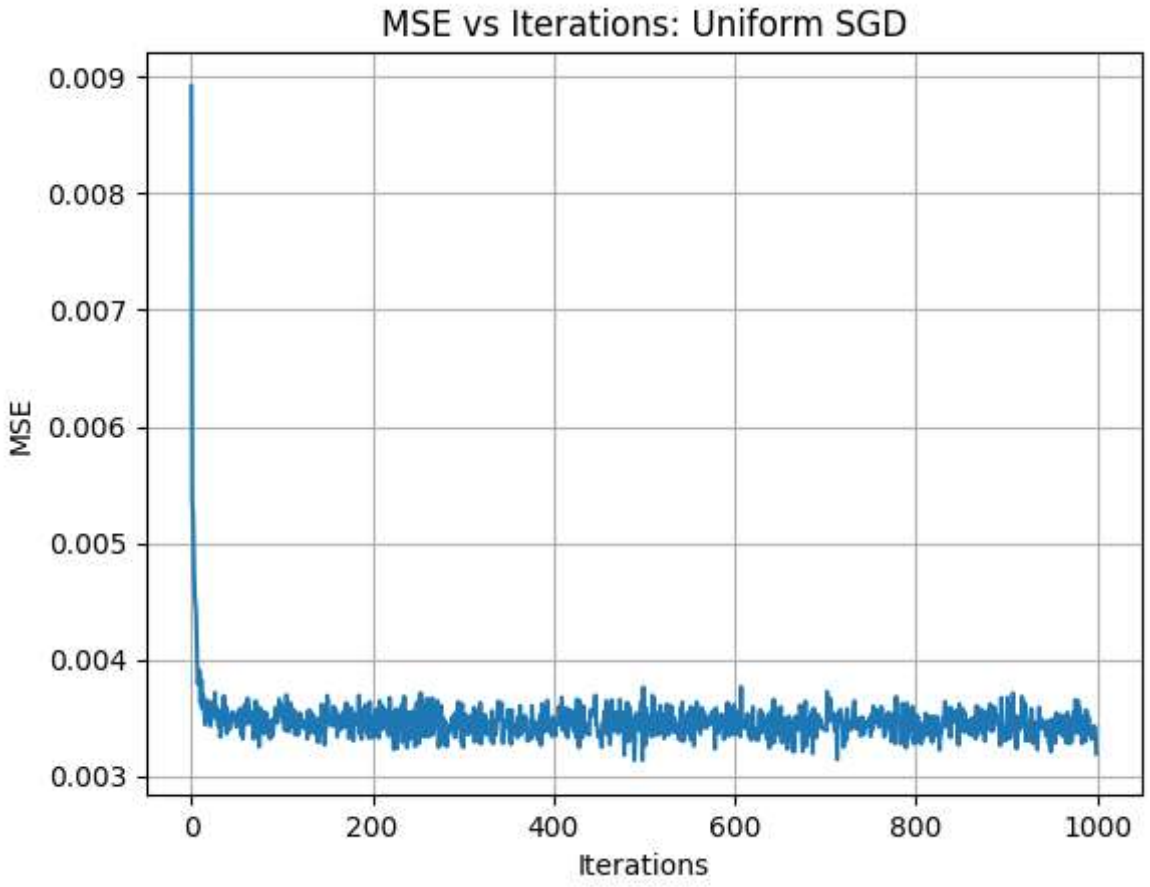
3. Uniform pdf

In [51]:
```
number = 51
## INITIALIZING a and b
b = np.random.choice(usample,number)
b.sort()
a = finding_a_uniform(b)

iters = 1000
MSE = []

for i in tqdm.tqdm(range(iters)):
    Uin = np.random.choice(usample,1000)  ## batch size 25
    mse = finding_mse(Uin,a)
    MSE.append(mse)
    a = grad_descent_a(Uin,a,i)
    b = finding_b(a,b)
```

```
100%|██████████| 1000/1000 [00:16<00:00, 59.34it/s]
```

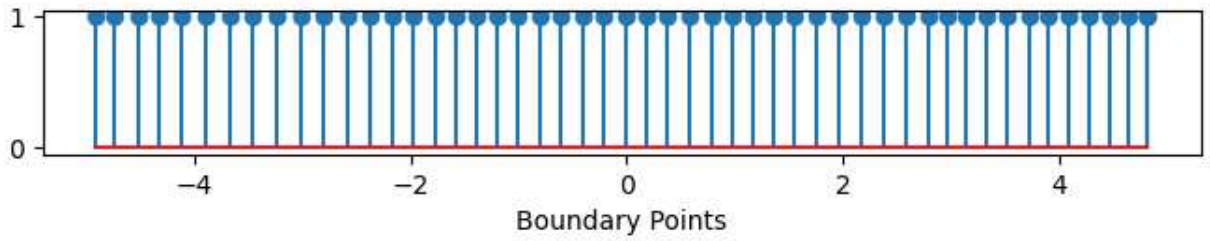
In [52]:
```
plt.plot(MSE)
plt.xlabel('Iterations')
plt.ylabel("MSE")
plt.title("MSE vs Iterations: Uniform SGD")
plt.grid()
print("Last MSE reached: ", MSE[-1])
```

```
Last MSE reached:  0.0031897244038577837
```



In [53]:
```
## visualise boundary points
one = np.ones(len(b))
plt.figure(figsize = (8,1))
plt.stem(b,one)
plt.xlabel('Boundary Points')
```

Out[53]:
```
Text(0.5, 0, 'Boundary Points')
```



Inference:

From the above results, we see that stochastic gradient descent performed equally well as did Lloyd Max.

To be noted: Stochastic Gradient Descent requires no prior knowledge of the pdf while Lloyd Max is dependent on the input pdf to calculate representation points. This makes SGD advantageous when pdf of input is unknown.

References

1. Robert Gallager, course materials for 6.450 Principles of Digital Communications I, Fall 2006. MIT OpenCourseWare(http://ocw.mit.edu/ (http://ocw.mit.edu/)), Massachusetts Institute of Technology.
2. https://numpy.org/doc/stable/ (https://numpy.org/doc/stable/)
3. https://docs.scipy.org/doc/numpy-1.15.0/reference/generated/numpy.random.random.html (https://docs.scipy.org/doc/numpy-1.15.0/reference/generated/numpy.random.random.html)
4. https://numpy.org/doc/1.16/reference/routines.random.html (https://numpy.org/doc/1.16/reference/routines.random.html)