Bayes Parameter Estimation

In the lecture, we have analyzed how the parameters of prior and posterior distributions relate analytically for Gaussian distributions. However, in the more general non-Gaussian case, the analysis is not possible. Instead, we need to use numerical methods to compute or estimate these distributions.

Let us consider data generated by a distribution of parameter θ. The prior and the data-generating probability functions are the following:

$$p(\theta) = \frac{1}{2}e^{-|\theta|} \qquad\qquad p(x|\theta) = \frac{1}{2}e^{-|x-\theta|}$$

Given an observation x, the posterior distribution for the unkown parameter θ can be obtained from the Bayes theorem:

$$p(\theta|x) = \frac{p(\theta)p(x|\theta)}{p(x)} \qquad \text{where} \quad p(x) = \int p(\theta)p(x|\theta)d\theta$$

Suppose that we have observed the following data point:

In [5]:

```
x = 3.0
```

We now study several techniques to compute/estimate this posterior distribution based either on explicit integration or sampling.

1. Explicit Integration (10 P)

Applying the Bayes formula requires the computation of an integral for $p(x)$. A numerical approximation of the integral is given by the Riemann sum:

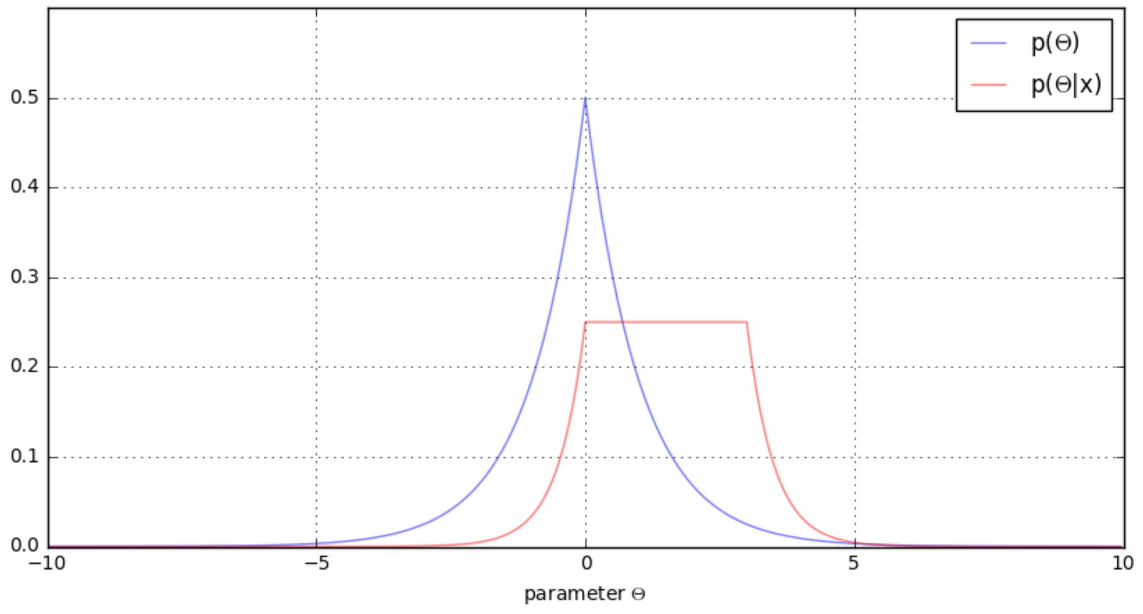
$$p(x) = \sum_{k=-\infty}^{\infty} p(k\cdot\Delta\theta)\cdot p(x|k\cdot\Delta\theta)\cdot\Delta\theta$$

where $\theta = k\cdot\Delta\theta$. In practice, we restrict the range of the integration where it has most of its support (i.e. choose k such that θ is between -10 to 10), and consider a small constant step size $\Delta\theta = 0.05$. In your code, you should use of numpy fast operations (e.g. `numpy.sum`) when possible, and avoid explicit loops in Python.

- **Using the integration method, calculate the posterior function for the indicated range and step size.**
- **Plot the prior and posterior probability functions.**

In [13]:

```
%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np
stepsize = 0.05
samplesize = 100000
def CalculateEvidance(ptheta,pxtheta):
    return np.sum(ptheta * pxtheta * stepsize)

def GetUniformData(start,end,size):
    return np.random.uniform(start,end,size)

def CalculatePrior(data):
    return (1/2)*(np.e**-(np.abs(data)))

def CalculateLikelyhood(data):
    return (1/2)*(np.e**-(np.abs(x - data)))

def CalculatePosterior(ptheta,pxtheta,px):
    return (ptheta * pxtheta)/px

def GetInputData(start,end,size):
    return np.arange(-10,10,size)

inputdata = GetInputData(-10,10,stepsize)
priordata = CalculatePrior(inputdata)
likelyhood = CalculateLikelyhood(inputdata)
evidence = CalculateEvidance(priordata,likelyhood)
posteriordata = CalculatePosterior(priordata , likelyhood , evidence)
plt.figure(figsize=(10,5))
plt.plot(inputdata,priordata,alpha=0.5,color='blue',label='p($\Theta$)')
plt.xlabel('parameter $\Theta$')
plt.plot(inputdata,posteriordata,alpha=0.5,color='red',label='p($\Theta$|x)')
plt.ylim(0,0.6)
plt.legend(loc='upper right')
plt.grid(True)
```



2. Basic Rejection Sampling (10 P)

When the parameter space is large, analytic integration becomes untractable, and therefore, sampling algorithms are needed. In this exercise we test on the same one-dimensional example as in exercise 1, a simple rejection sampling algorithm to approximate the posterior distribution. Let $q(\theta)$ be the joint probability function defined as $q(\theta) = p(x|\theta) \cdot p(\theta)$. The rejection sampling algorithm seeks to sample uniformly from the surface under the function $q(\theta)$ by sampling uniformly from a larger box, and rejecting samples that are not under $q(\theta)$. The algorithm operates as follows:

repeat 100000 times:

1. $\theta \sim \mathcal{U}(-10, 10)$
2. $u \sim \mathcal{U}(0, 1)$
3. accept θ if $q(\theta) < u$

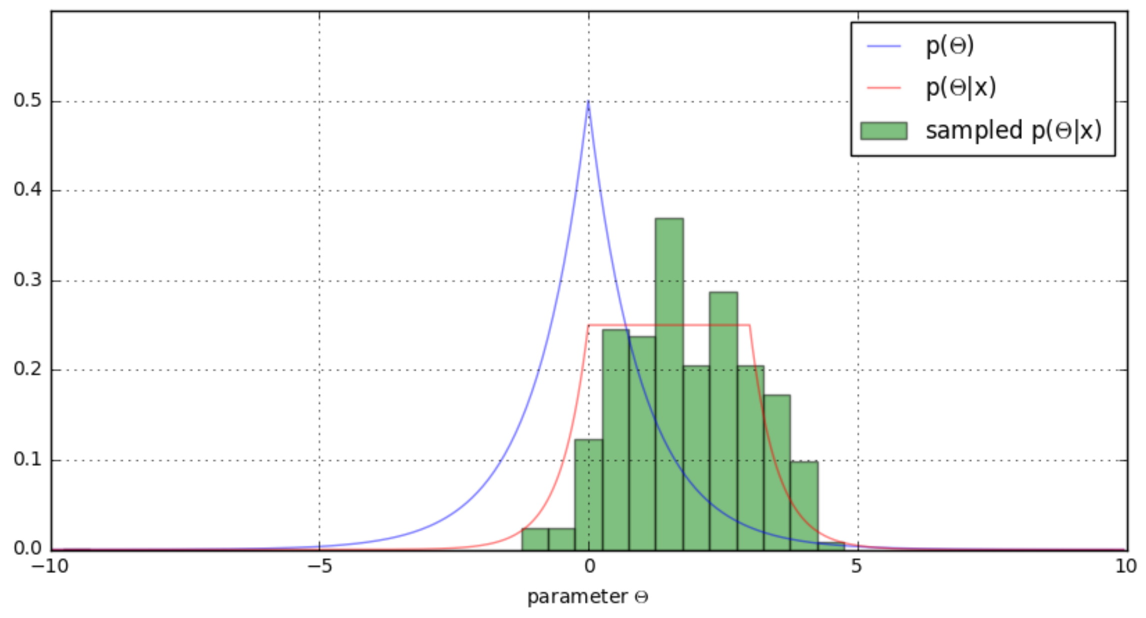
where $\mathcal{U}(a, b)$ denotes a uniform distribution on the interval $[a, b]$. The list of accepted θs forms an empirical posterior distribution in the parameter space, that can be viewed as a probability function by computing a histogram. We use a bin size of 0.5 for the histogram.

- **Implement this simple rejection sampling algorithm (make use of numpy parallelization when possible).**
- **Create a histogram from the accepted θs, and print the rejection rate.**
- **Plot the (normalized) histogram in superposition to the functions of the previous exercise.**

In [75]:

```
%matplotlib inline

populationdata = GetUniformData(-10,10,samplesize)
sampledata = GetUniformData(0,1,samplesize)

def GteRejectionSamplingResult():
    qpriordata = CalculatePrior(populationdata)
    qlikelyhood = CalculateLikelyhood(populationdata)
    qevidence = CalculateEvidance(qpriordata,qlikelyhood)
    qposteriordata = CalculatePosterior(qpriordata , qlikelyhood , qevidence)
    qtheta = qpriordata * qlikelyhood
    q=np.array(qtheta)
    con=q > sampledata
    acceptedTheta = np.extract(con,populationdata)
    return acceptedTheta

def PrintRejectionRate(acceptedData):
    acceptedLength=len(acceptedData)
    rej=1-(acceptedLength/100000)
    print ('Rejection rate:',rej)

def PlotGraph(acceptedData):
    plt.figure(figsize=(10,5))
    plt.plot(inputdata,priordata,alpha=0.5,color='blue',label='p($\Theta$)')
    plt.xlabel('parameter $\Theta$')
    plt.plot(inputdata,posteriordata,alpha=0.5,color='red',label='p($\Theta$|x)')
    plt.hist(acceptedData,bins=np.arange(-10,10,0.5),normed=1,align='right',alpha=0.5,c
olor='green',label='sampled p($\Theta$|x) ');
    plt.ylim(0,0.6)
    plt.legend(loc='upper right')
    plt.grid(True)
    plt.show()

acceptedDataIndex = GteRejectionSamplingResult()
PrintRejectionRate(acceptedDataIndex)
PlotGraph(acceptedDataIndex)
```

```
Rejection rate: 0.99756
```



3. Improving Rejection Sampling (10 P)

As it could be seen in exercise 2, the estimation of the posterior function is quite noisy, because the high rejection rate strongly reduces the number of samples available to build the statistics. A simple technique to reduce the number of rejections is to define a smaller box $[-10, 10] \times [0, h]$, where h is chosen as small as possible under the constraint that $h \geq q(\theta)$.

- **Show that $q(\theta)$ can be upper-bounded by $h = 0.25e^{-|x|}$, and that the bound is tight.**

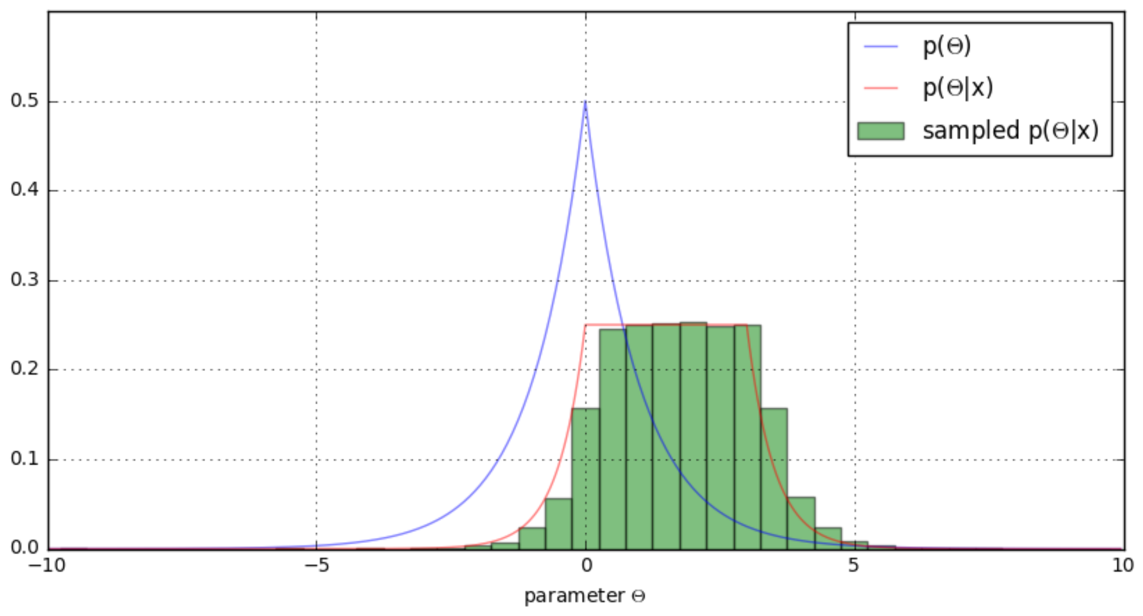
[REPLACE BY YOUR DEMONSTRATION]

- **Perform the same experiment as in exercise 2, but using this tighter upper-bound for sampling.**

```
In [76]:
```

```
%matplotlib inline
h = 0.25 *np.exp(-np.absolute(x))
sampledata = GetUniformData(0,h,samplesize)


acceptedDataIndex = GteRejectionSamplingResult()
PrintRejectionRate(acceptedDataIndex)
PlotGraph(acceptedDataIndex)
```

```
Rejection rate: 0.79968
```

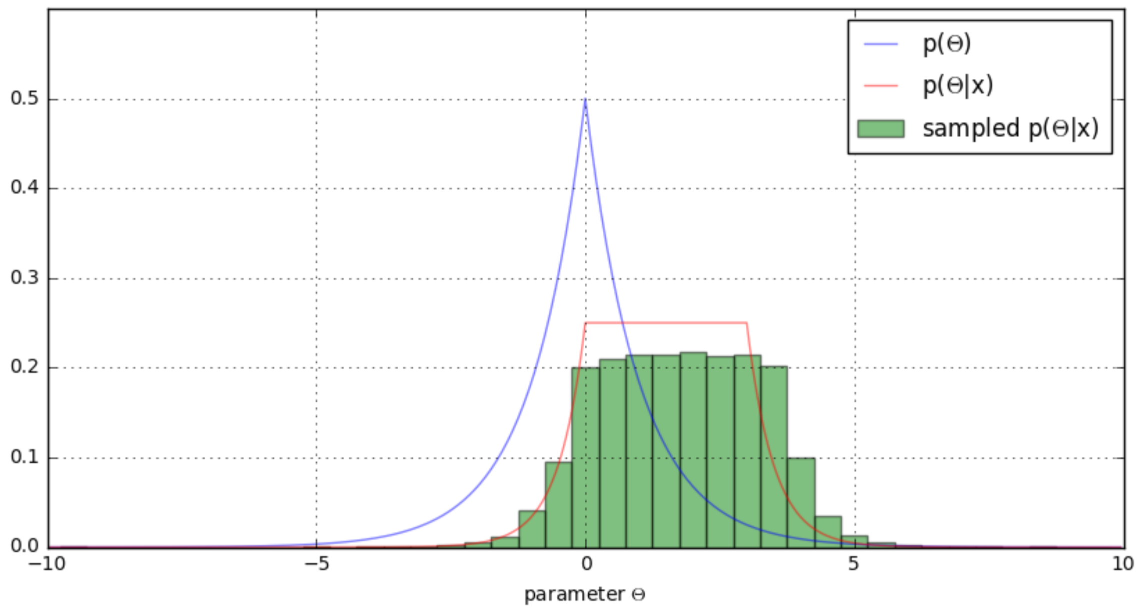


- **Show empirically, that if setting h smaller (e.g. $h = 0.125e^{-|x|}$), then the posterior is no longer sampled correctly.**

In [78]:

```
%matplotlib inline
h = 0.125 *np.exp(-np.absolute(x))
sampledata = GetUniformData(0,h,samplesize)

acceptedDataIndex = GteRejectionSamplingResult()
PrintRejectionRate(acceptedDataIndex)
PlotGraph(acceptedDataIndex)
```

```
Rejection rate: 0.76618
```



In [ ]: