Slice Sampling Multimodal

March 21, 2019

1 Multimodal Slice Sampling Example

1.0.1 Import Libraries

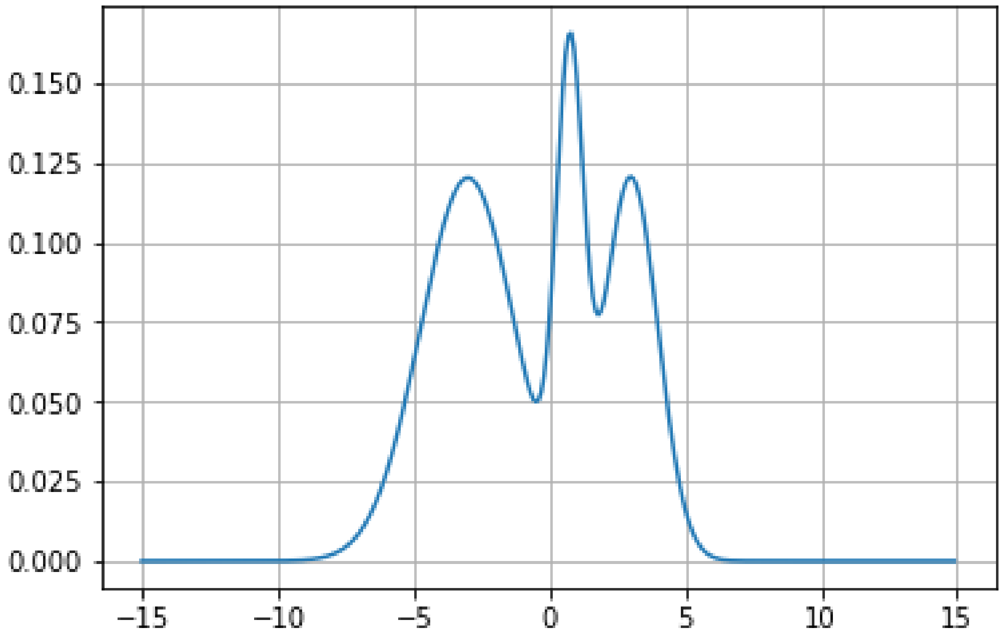
```
In [157]: import numpy as np
          import scipy.stats as stats
          import matplotlib.pyplot as plt
          import math
          from sklearn.preprocessing import normalize
          %matplotlib inline
```

1.0.2 Generate Unimodal Gaussian

```
In [176]: # Parameters used for Gaussian
          x_low = -15
          x_high = 15
          x_step = 0.1

          # Generate pdf
          X = np.arange(x_low, x_high, x_step)
          Y = 1.75*stats.norm.pdf(X,-3,1.75) + .6*stats.norm.pdf(X,.75,.5) + stats.norm.pdf(X,
          Y = normalize(Y[:,np.newaxis], axis=0).ravel()

          plt.plot(X, Y)
          plt.grid(True)
          plt.show()
```



1.0.3 Define function for initial sample for x_0

```
In [180]: low = X[0]
          high = X[-1]

          def sample_x(low=low, high=high):
              x0 = np.random.uniform(low=low, high=high)
              return x0

          x0 = sample_x(low, high)

          print(x0)
```

```
-5.102557107298681
```

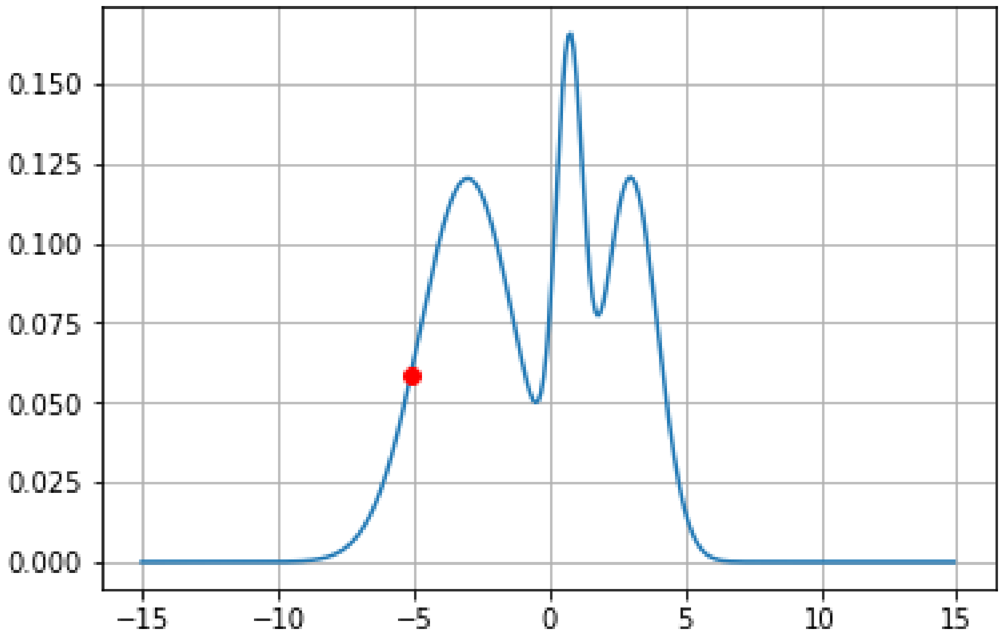
1.0.4 Define function for calculating f(x_0)

```
In [181]: def f_x(x):
              ind = np.argmin(np.abs(X - x))
              f_x0 = Y[ind]
              return f_x0

          f_x0 = f_x(x0)

          plt.plot(X, Y)
```

```
plt.plot([x0], [f_x0], 'ro')
plt.grid(True)
plt.show()
```

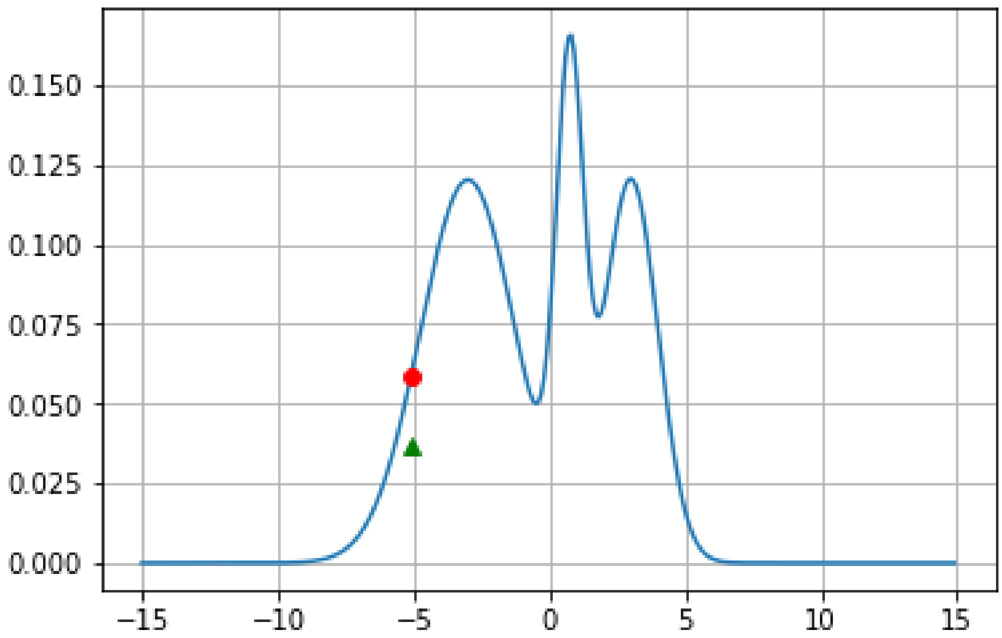


1.0.5 Define function for sampling in the interval (0, f(x_0)), calculate y

```
In [182]: def sample_y(x0, f_x0):
              y = np.random.uniform(low=0, high=f_x0)
              return y

          y = sample_y(x0, f_x0)

          plt.plot(X, Y)
          plt.plot([x0], [f_x0], 'ro')
          plt.plot([x0], [y], 'g^')
          plt.grid(True)
          plt.show()
          print(y)
```



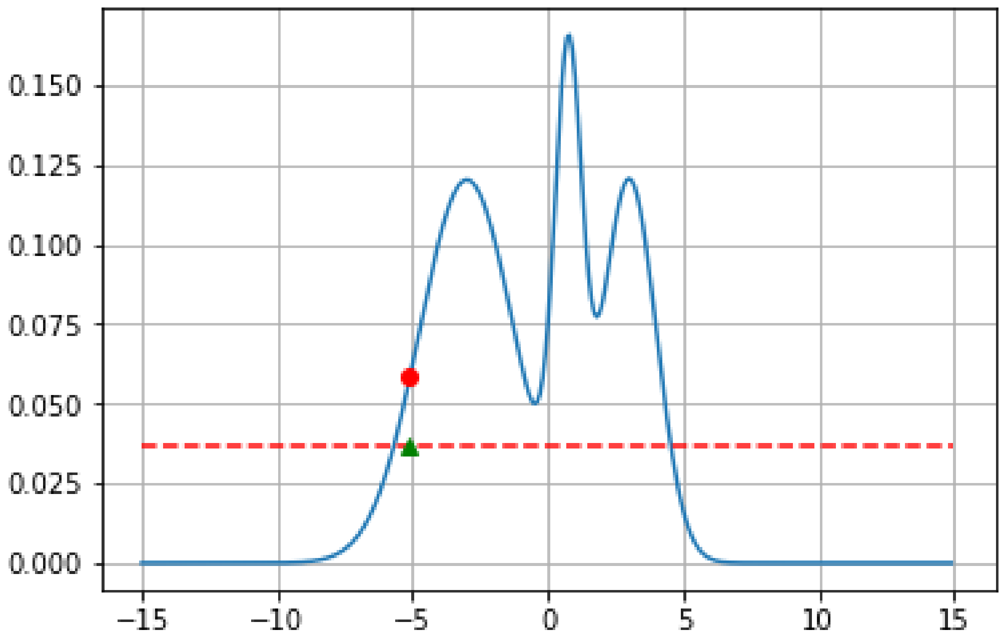

```
0.03687739857530197
```

1.0.6 Define horizontal slice using y

```
In [183]: def slice_y(y):
              _x = np.linspace(-15, 15, 50)
              horizontal_line = np.array([y for i in range(len(_x))])
              return _x, horizontal_line

          line = slice_y(y)

          plt.plot(X, Y)
          plt.plot(line[0], line[1], 'r--')
          plt.plot([x0], [f_x0], 'ro')
          plt.plot([x0], [y], 'g^')
          plt.grid(True)
          plt.show()
```



1.0.7 Define function for estimating the sampling interval using doubling update

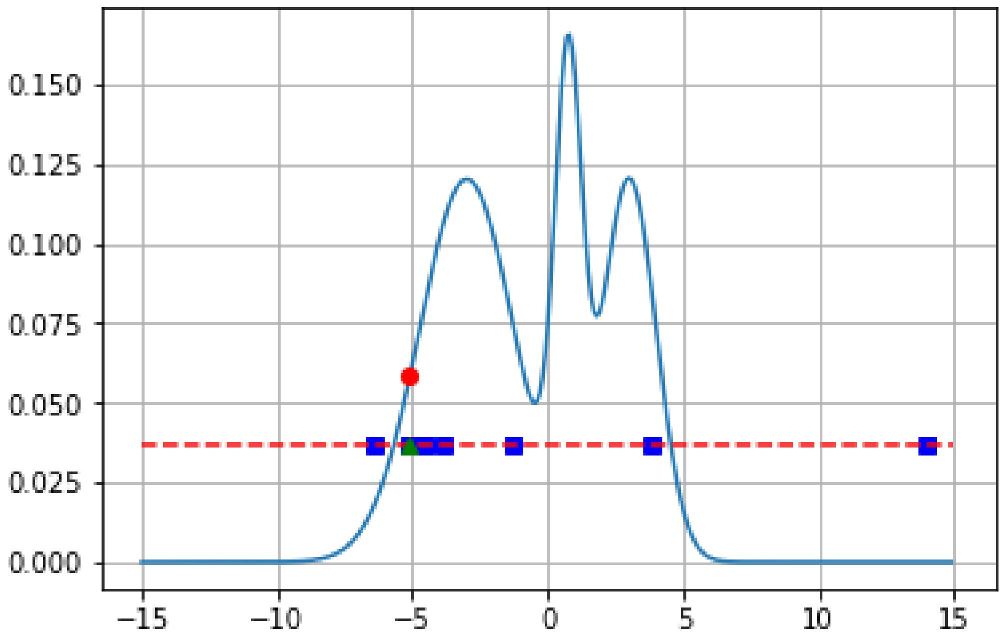
```
In [184]: def doubling_update(x0, f_x0, y, w=0.01, p=100):
              u = np.random.uniform()
              l = x0 - w*u
              r = l + w
              k = p
              left = []
              right = []

              while k > 0 and (y < f_x(l) or y < f_x(r)):
                  v = np.random.uniform()
                  if v < .5:
                      l = l - (r-l)
                      left.append(l)
                  else:
                      r = r + (r-l)
                      right.append(r)
                  k = k-1
              patches = np.concatenate((left,right), axis=None)
              return l, r, patches

          double = doubling_update(x0, f_x0, y)

          ascisse = np.array([y for i in range(len(double[2]))])
```

```
plt.plot(double[2],np.array([y for i in range(len(double[2]))]) , 'bs')
plt.plot(X, Y)
plt.plot(line[0], line[1], 'r--')
plt.plot([x0], [f_x0], 'ro')
plt.plot([x0], [y], 'g^')
plt.grid(True)
plt.show()
```



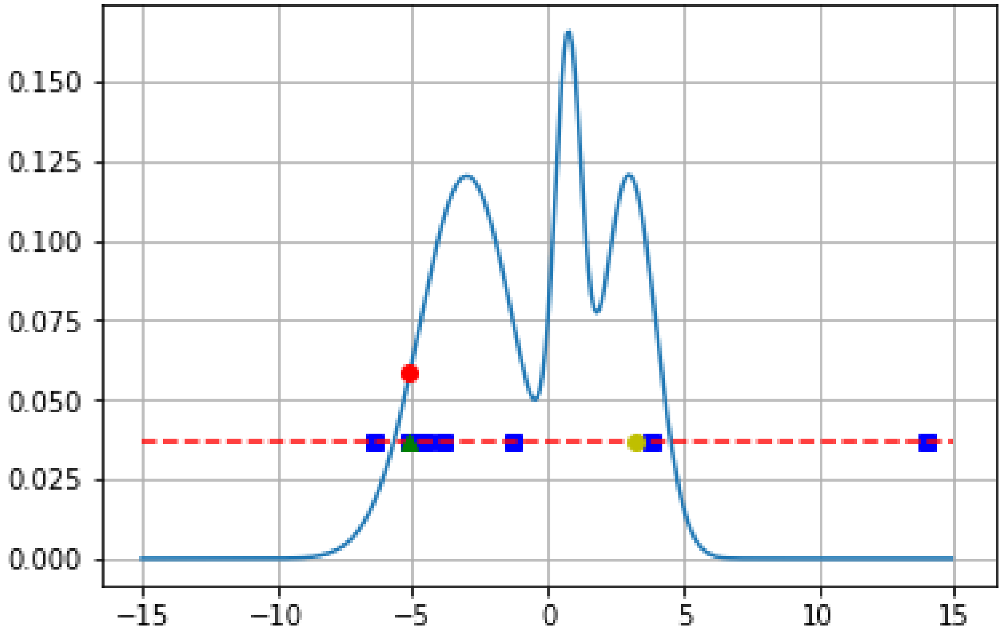
1.0.8 Define updating rule for sampling x_n from new interval

In [185]:
```
def new_x_double(y, l, r):
    new_x0 = sample_x(l, r)
    while y > f_x(new_x0):
        new_x0 = sample_x(l, r)
    return new_x0

new_double = new_x_double(y, double[0], double[1])

ascisse = np.array([y for i in range(len(double[2]))])
plt.plot(double[2], ascisse, 'bs')
plt.plot(X, Y)
plt.plot(line[0], line[1], 'r--')
plt.plot([x0], [f_x0], 'ro')
plt.plot([x0], [y], 'g^')
plt.plot([new_double], [y], 'yo')
```

```
plt.grid(True)
plt.show()
```

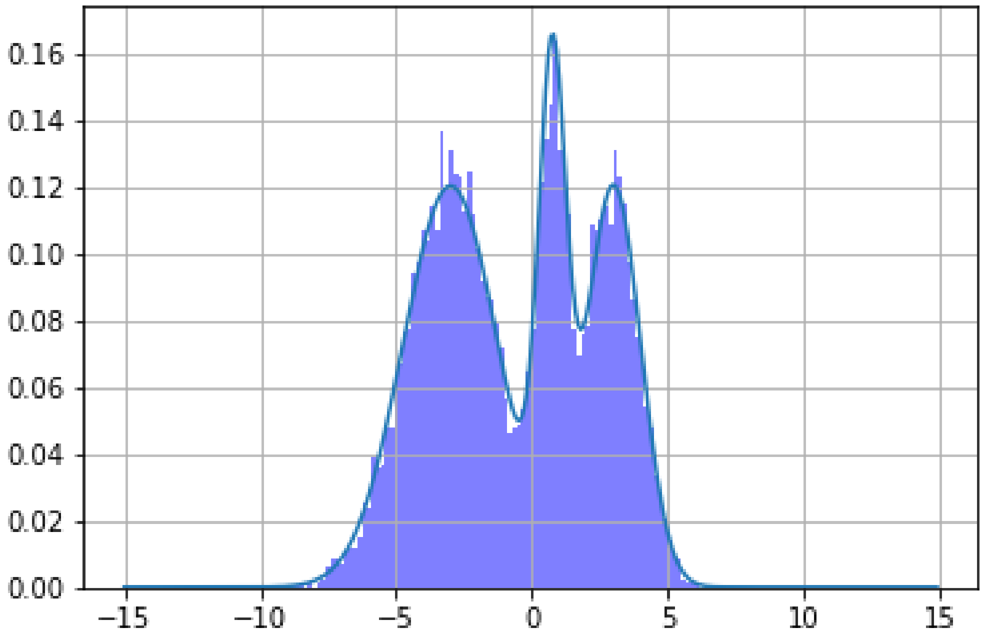


1.0.9 Sample and plot results

In [186]:
```
samples_double = []
i = 0
l = double[0]
r = double[1]

while i <10000:
    new_x = new_x_double(y, l, r)
    new_fx = f_x(new_x)
    new_sampled_y = sample_y(new_x, new_fx)
    new_double = doubling_update(new_x, new_fx, new_sampled_y)
    samples_double.append(round(new_x,2))
    y = new_sampled_y
    l = new_double[0]
    r = new_double[1]
    i = i+1

num_bins = 100
plt.plot(X, Y)
plt.grid(True)
n_d, bins_d, patches_d = plt.hist(samples_double, num_bins, facecolor='blue', alpha=(
plt.show()
```



1.0.10 Define Stepout Update function

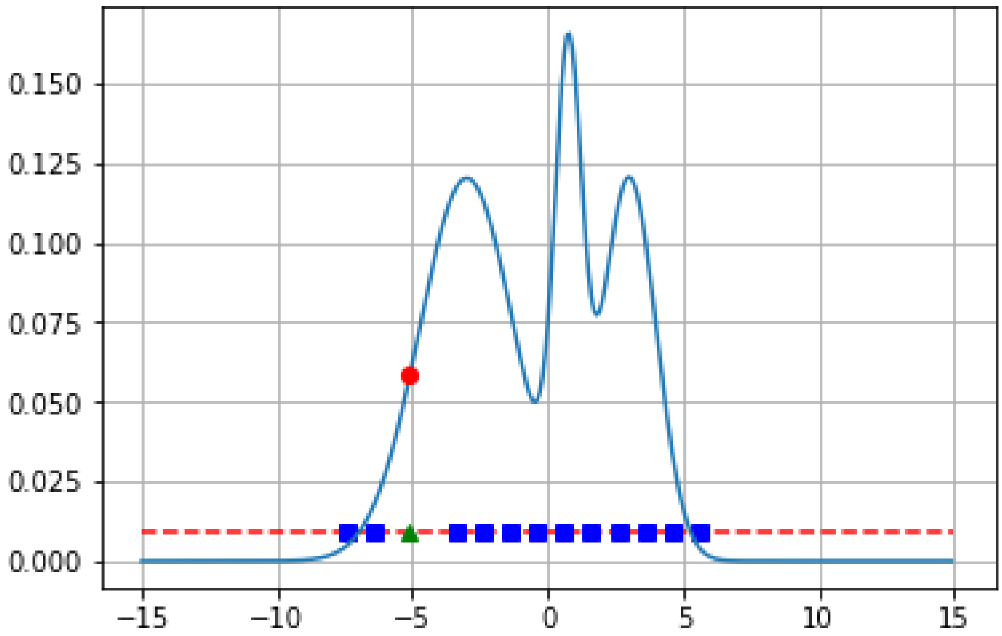
```
In [187]: def stepout_update(x0, f_x0, y, w=1, m=40):
              u = np.random.uniform()
              l = x0 - w*u
              r = l + w
              v = np.random.uniform()
              j = math.floor(m*v)
              k = (m-1) - j
              patches = []

              while j>0 and y<f_x(l):
                  l -= w
                  j -= 1
                  patches.append(l)
              while k>0 and y<f_x(r):
                  r += w
                  k -= 1
                  patches.append(r)

              return l, r, patches


          stepout = stepout_update(x0, f_x0, y)
```

```
line = slice_y(y)
plt.plot(line[0], line[1], 'r--')
plt.plot(stepout[2], np.array([y for i in range(len(stepout[2]))]), 'bs')
plt.plot(X, Y)
plt.plot([x0], [f_x0], 'ro')
plt.plot([x0], [y], 'g^')
plt.grid(True)
plt.show()
```



1.0.11 Define updating rule for sampling x_n from new interval

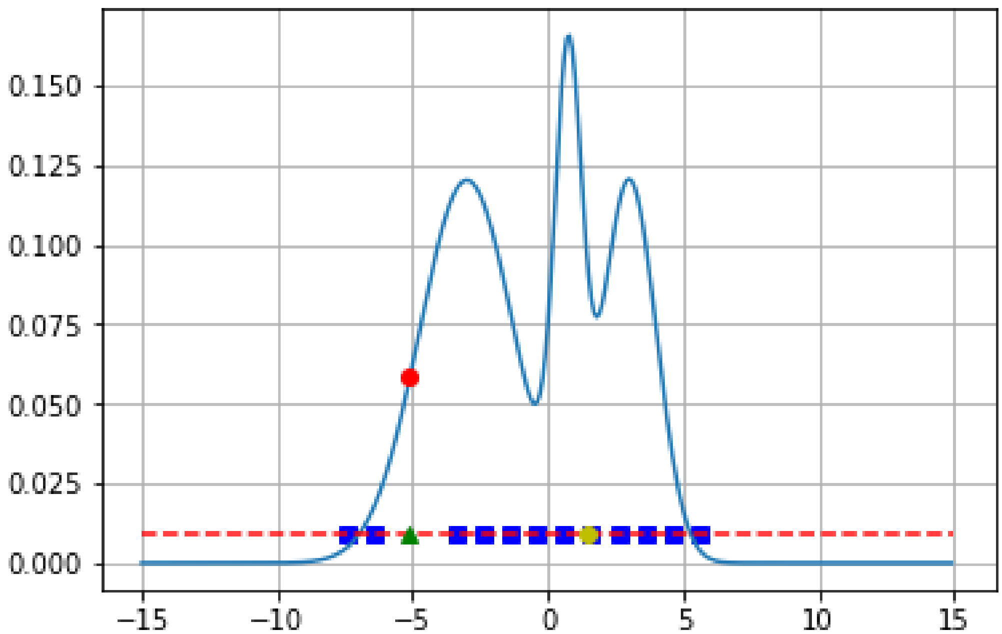
```
In [188]: def update_x_stepout(y, l, r):
              new_x = sample_x(l, r)
              while y > f_x(new_x):
                  new_x = sample_x(l, r)
              return new_x

          new_x_stepout = update_x_stepout(y, stepout[0], stepout[1])
          print(new_x_stepout)

          plt.plot(stepout[2], np.array([y for i in range(len(stepout[2]))]), 'bs')
          plt.plot(X, Y)
          plt.plot(line[0], line[1], 'r--')
          plt.plot([x0], [f_x0], 'ro')
          plt.plot([x0], [y], 'g^')
          plt.plot([new_x_stepout], [y], 'yo')
```

```
plt.grid(True)
plt.show()
```

```
1.4659574728678395
```

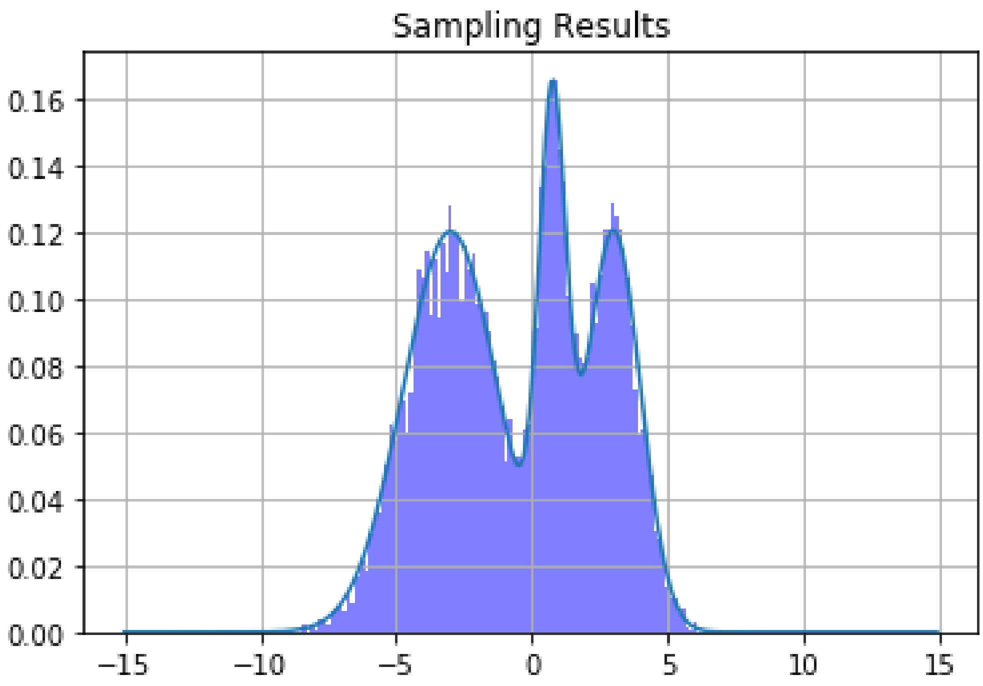


1.0.12 Sample and plot results

```
In [189]: samples_stepout = []
          iterations_s = []
          kl_values_s = []
          i = 0
          l = stepout[0]
          r = stepout[1]

          while i <10000:
              new_x = update_x_stepout(y, l, r)
              new_fx = f_x(new_x)
              new_sampled_y = sample_y(new_x, new_fx)
              new_stepout = stepout_update(new_x, new_fx, new_sampled_y)
              samples_stepout.append(round(new_x,2))
              y = new_sampled_y
              l = new_stepout[0]
              r = new_stepout[1]
              i = i+1
```

```
plt.title(r'Sampling Results')
plt.plot(X, Y)
num_bins = 100
n_s, bins_s, patches_s = plt.hist(samples_stepout, num_bins, facecolor='blue', alpha=
plt.grid(True)
plt.show()
```



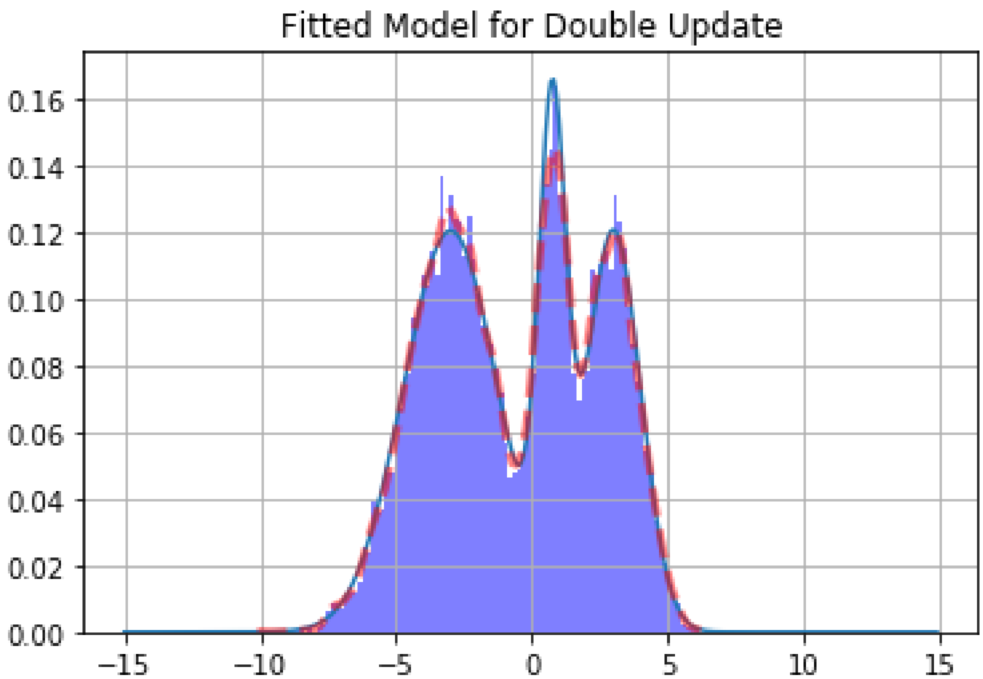
1.1 Fix approximate kernel model to sample distribution

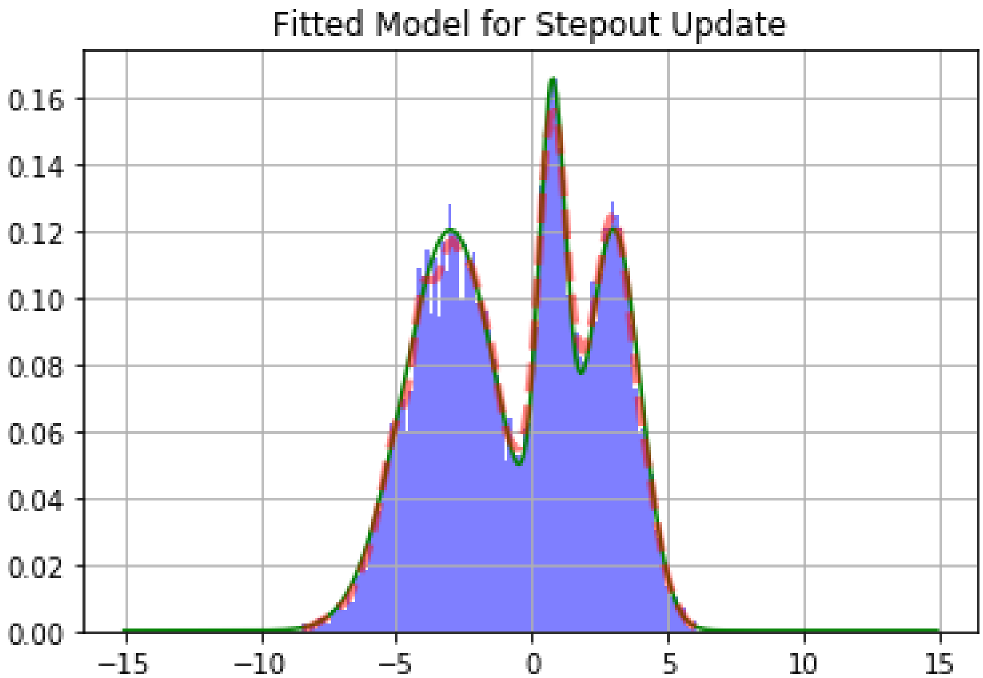
```
In [190]: from statsmodels.nonparametric.kde import KDEUnivariate

          def kde_statsmodels_u(x, x_grid, bandwidth=0.2, **kwargs):
              """Univariate Kernel Density Estimation with Statsmodels"""
              kde = KDEUnivariate(x)
              kde.fit(bw=bandwidth, **kwargs)
              return kde.evaluate(x_grid)

          plt.title(r'Fitted Model for Double Update')
          plt.plot(X, Y)
          pdf_d = kde_statsmodels_u(samples_double, bins_d)
          plt.plot(bins_d, pdf_d,'--', color='red', alpha=0.5, lw=3)
          n_d, bins_d, patches_d = plt.hist(samples_double, num_bins, facecolor='blue', alpha=(
          plt.grid(True)
          plt.show()
```

```
plt.title(r'Fitted Model for Stepout Update')
plt.plot(X, Y, color='green')
pdf_s = kde_statsmodels_u(samples_stepout, bins_s)
plt.plot(bins_s, pdf_s,'--', color='red', alpha=0.5, lw=3)
n_s, bins_s, patches_s = plt.hist(samples_stepout, num_bins, facecolor='blue', alpha=
plt.grid(True)
plt.show()
```




Fitted Model for Stepout Update
0.16
0.14
0.12
0.10
0.08
0.06
0.04
0.02
0.00
−15
−10
−5
0
5
10
15