BFGS WHAM

Cameron Kopp

June 2019

1 SampleConstructor Class Examples

1.1 Initialization Method

The initialization method builds the arrays needed for the sample construction methods as soon as a SampleConstructor object is created. The object requires the following arguments: minimum value of x, maximum value of x, simulation number , bin size, and the spring constant for the biasing potential.

```
xmin = 0
xmax = 5
simnum = 10
binsize = 1/4
k = 1
exampleSample = SampleConstructor(xmin,xmax,simnum,binsize,k)
```

From these inputs, the initialization method creates an array for both the index and the value of the range of bins, binindex and binx respectively.

```
print('binx, ', exampleSample.binx)
print('binindex, ', exampleSample.binindex)
```

```
In [32]: runfile('/Users/cameronkopp/Desktop/python/combineclasses.py',
wdir='/Users/cameronkopp/Desktop/python')
binx,  [ 0.    0.25  0.5   0.75  1.    1.25  1.5   1.75  2.    2.25  2.5
2.75
  3.    3.25  3.5   3.75  4.    4.25  4.5   4.75]
binindex,  [ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19]
```

The initialization method also creates similar arrays simx and simindex, where simx stores the location of the biasing potential for a given sample($x = x_{min} + j\left(\frac{x_{max}-x_{min}}{S}\right)$), and simindex stores the corresponding index for simulation $j = 0, 1, ..., S-1$ where S is equal to the simulation number argument.

```
print('simx, ', exampleSample.simx)
print('simindex, ', exampleSample.simindex)
```

```
In [33]: runfile('/Users/cameronkopp/Desktop/python/combineclasses.py',
wdir='/Users/cameronkopp/Desktop/python')
simx,  [ 0.   0.5  1.   1.5  2.   2.5  3.   3.5  4.   4.5]
simindex,  [0 1 2 3 4 5 6 7 8 9]
```

The method initializes two other arrays of zeros, hist and m, to store the histogram data from a sample constructor function. hist has dimensions (simindex,binindex), and m is a vector with size(binindex), and will store the total number of histogram points in each bin summed over all of the simulations.

```
print('hist shape ', shape(exampleSample.hist))
print('m shape, ', shape(exampleSample.m))
```

```
In [34]: runfile('/Users/cameronkopp/Desktop/python/combineclasses.py',
wdir='/Users/cameronkopp/Desktop/python')
hist shape  (10, 20)
m shape,  (20,)
```

1.2 Method sconXSquaredSinSquaredX(n) examples

The size of the sample needs to be larger to get an idea of how this method works.

```
xmin = 0
xmax = 5
simnum = 100
binsize = 1/100
k = 1
exampleSample = SampleConstructor(xmin,xmax,simnum,binsize,k)
```

The idea behind this method is to generate, for each simulation j, a set of n random numbers from a probability distribution that includes the biasing potential. It is fairly easy to construct these methods for any function, but $\rho(x) = x^2 \sin^2(x)$ seems to work well for the purpose of illustrating the method. This is function we expect to see, then, after we run the output of this method through the WHAM equations to remove the affects of the biasing potential. For each simulation j, this method will return n points independently chosen from the distribution

$$\rho_j(x) = Cx^2 \sin^2(x) * e^{-k(x-\mu_j)^2} \tag{1}$$

where

$$\mu_j = x_{min} + j\left(\frac{x_{max} - x_{min}}{S}\right) \tag{2}$$

and

$$C = \frac{1}{(x_{max})^2} \tag{3}$$

The algorithm used to generate random points from this distribution is called the Acceptance-Rejection method, which relies on the ability to define a function, $f_{max}(x)$, which satisfies $f_{max}(x) \geq \rho_j(x)$ for $x \in (x_{min}, x_{max})$, and which has a known CDF. In this segment of code, I have simply chosen $f_{max}(x) = 1$, and multiplied $\rho_j(x)$ by a constant to ensure that it less than one on $x \in (x_{min}, x_{max})$. Because $\sin^2(x)$

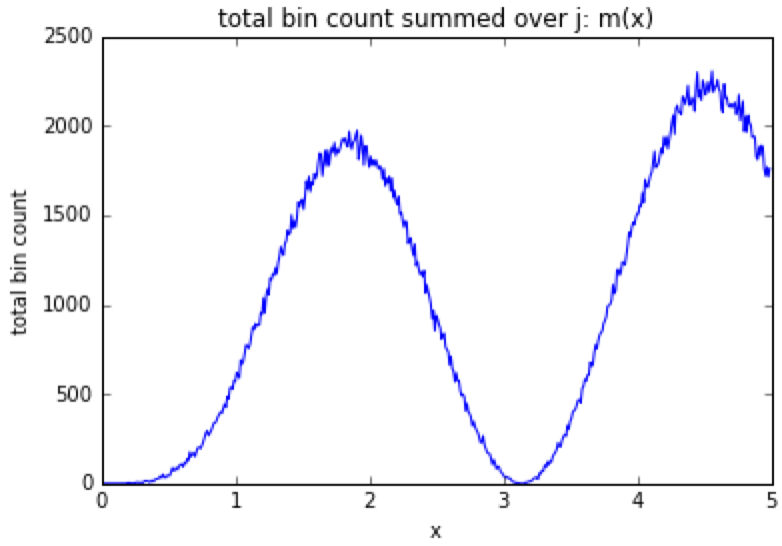
and the Gaussian biasing potential are both less than 1 for all x, $C = 1/(x_{max})^2$ will satisfy the inequality over $x \in (x_{min}, x_{max})$. This is definitely not the fastest possible algorithm, but it has been able to generate samples with n = 10000 within ten minutes on my laptop. Below is the segment of code which sets up the histogram.

```
    for j in self.simindex:
        print(j)
        # reset sample counter for each sim
        i = 0
        z = 0
        # center gaussian at simx[j]
        mu = self.simx[j]
        #WHILE LOOP GENERATES SAMPLES NI FOR SIM J
        while(i < self.ni): #and (z < 100000):
             z = z + 1
            # u distributed test sample point
            xu = r.uniform(0,1)
            xt = (ap*xu)**(1/3)
            # MAKE SURE TEST SAMPLE POINT IN RANGE XMIN,XMAX
            pxt = (3/ap)*xt**2
            #value of p(x)*bias
            pgxt = (3/ap)*ag * exp(-self.springk*((xt-mu)**2))*(xt**2)
                      g = exp(-self.springk*((xt-mu))**2)
                      print(g)
            # uniform random 0 to gxt
            y = r.uniform(0,pxt)
            # test if pbxt < y, if it is save sample, if not back to
            # top of the loop
            t = y - pgxt
            if t < 0:
                if (xt<self.xmax) and (xt>self.xmin):
                    self.xlist[j][i] = xt
                    histval = (xt * (1/self.binsize))-histfix
                    h = floor(histval)
                    ih = int(h)
                    self.hist[j][ih] = self.hist[j][ih] + 1
                    self.nj[j] = self.nj[j] + 1
                i = i + 1
```

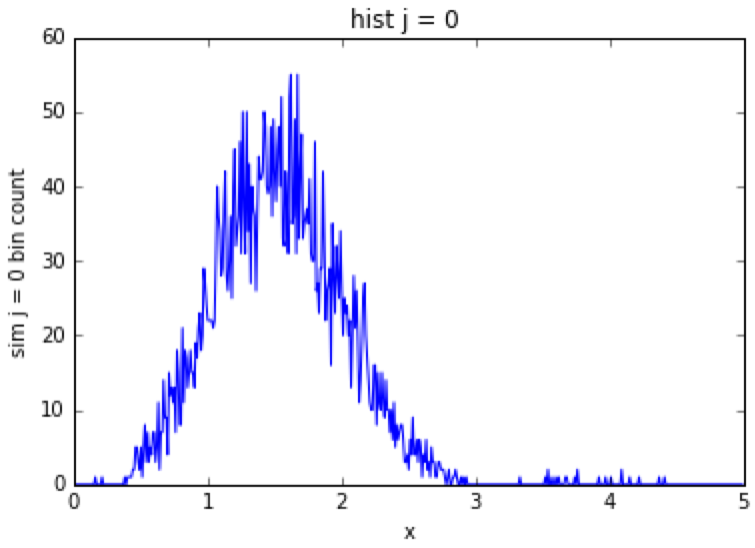
Calling this method returns the hist variable, which stores the histogram of simulation j in row j, with the columns representing each bin. It does not return the m vector, but it does create the vector and save it to the object instance itself. It is retrievable and readable with "instanceName".m, as shown in the argument for the first plot below with exampleSample.m.

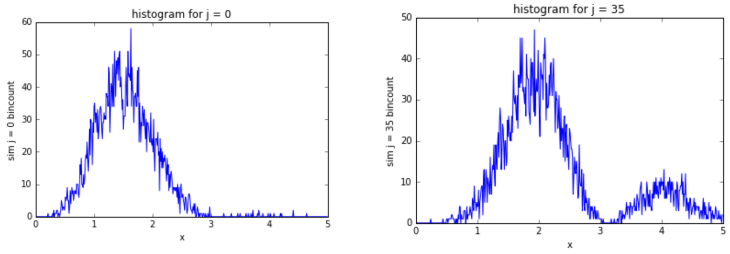
```
n = 5000
exampleHistogram = exampleSample.sconXSquaredSinSquaredX(n)

figure()
plot(exampleSample.binx,exampleSample.m)
xlabel('x')
ylabel('total bin count')
title('total bin count summed over j: m(x)')
show()
```

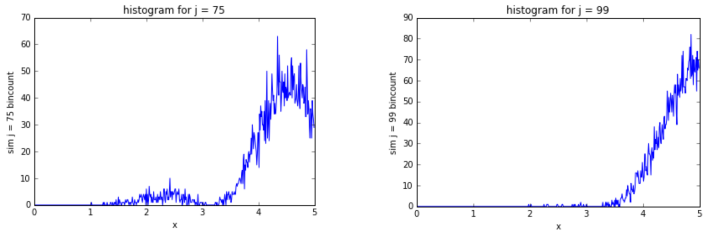



```
figure()
plot(exampleSample.binx,exampleHistogram[0])
xlabel('x')
ylabel('sim j = 0 bin count')
title('hist j = 0 ')
show()
```





(a) $j = 0$ (b) $j = 35$

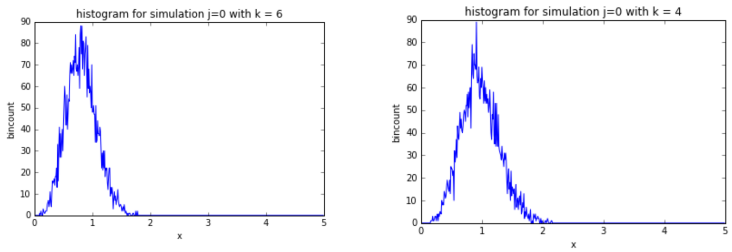


(c) $j = 75$ (d) $j = 99$

Figure 1: Plot of exampleHistogram[j] for simulation number: (a) $j = 0$, (b) $j = 35$, (c) $j = 75$, (d) $j = 99$

Changing the spring constant, k, effects the method by changing the standard deviation of the biasing potential, whose shape is Gaussian and thus has a standard deviation: $\sigma = \frac{1}{\sqrt{2k}}$. The plots generated below are the histogram of the same simulation number j, generated by the same method, as the plots above, but with the spring constant set to 4 instead of 1.

```
xmin = 0
xmax = 5
simnum = 100
binsize = 1/100
k = 4
exampleSample = sampleConstructor(xmin,xmax,simnum,binsize,k)
n = 5000
exampleHistogram = exampleSample.sconXSquaredSinSquaredX(n)
```



(a) $k = 6$ (b) $k = 4$

(c) $k = 1$ (d) $k = 1/2$

Figure 2: Histogram at simulation number $j = 0$ for: (a) $k = 6$, (b) $k = 4$, (c) $k = 1$, (d) $k = 1/2$

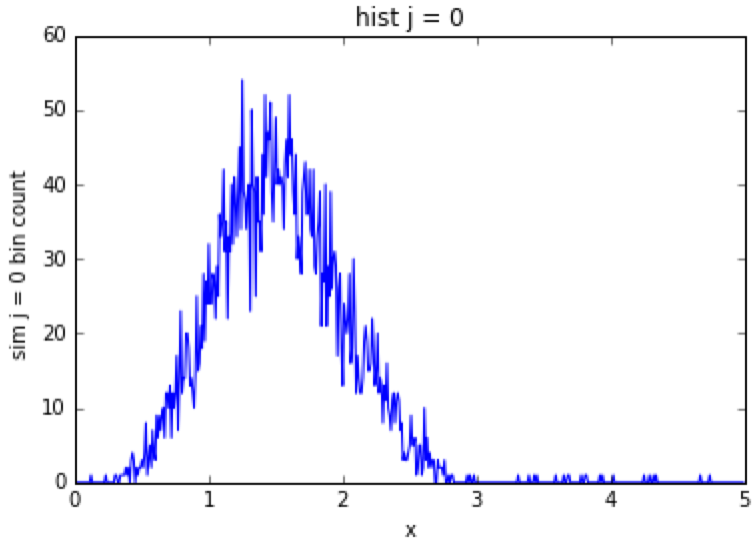
2 Subclass WhamEqs1D: Examples

The class WhamEqs1D is structured as a subclass of SampleConstructor, which means that all of variables and functionality of the class SampleConstructor can be accessed by an instance of class WhamEqs1D. For example the following code creates an instance of an object with an identical sample distribution to the plots for $k = 1$ above:

```
xmin = 0
xmax = 5
simnum = 100
binsize = 1/100
k = 1
exampleSample = WhamEqs1D(xmin,xmax,simnum,binsize,k)
n = 5000
exampleHistogram = exampleSample.sconXSquaredSinSquaredX(n)
```

Instead of using the returned variable example histogram to verify that the distributions are the same, the instance of the object itself can be used to show that the WhamEq1D object does have access to the same variables and functions that a SampleConstructor object would.

```
j = 0
figure()
plot(exampleSample.binx,exampleSample.hist[j])
xlabel('x')
ylabel('sim j = {} bin count '.format(j))
title('hist j = {}'.format(j))
show()
```

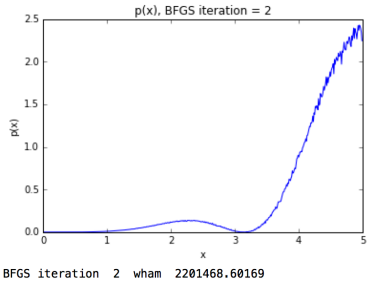


Now that the sample data is constructed the methods within the wham class are ready to minimize the wham equations. First call the initialGCalc() method, which returns the initial approximation of the normalization constants f_j, in the form cited from the 2012 Hummer and Zhu paper. Then the BFGS algorithm can be called with the argument: iteration limit for BFGS, error tolerance for BFGS, α, τ, and β which are constraints for the line search method within BFGS, the iteration limit for the line search algorithm, and a list of iterations i for the method to print a plot of $\rho(x)$. The entire code for setting up for and calling the BFGS method should look something like:

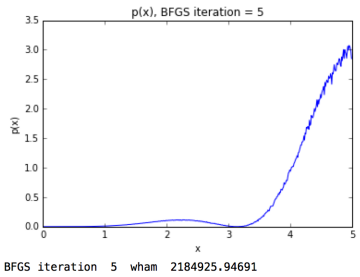
```
#SET UP SAMPLE DATA
xmin = 0
xmax = 5
simnum = 100
binsize = 1/100
k = 1
exampleSample = WhamEqs1D(xmin,xmax,simnum,binsize,k)
n = 5000
exampleHistogram = exampleSample.sconXSquaredSinSquaredX(n)
#SET UP FOR AND CALL BFGS METHOD
a0c = 2
betac = 0.01
taoc = 0.5
il = 150
lsl = 200
etol = 0
plotcheck = [2,5,10,25,50,100,150]
#CALL initialGCALC FIRST, the gjBFGSCalc()
g0 = exampleSample.initialGCalc()
exampleG = exampleSample.gjBFGSCalc(il,etol,a0c,taoc,betac,lsl,plotcheck)
```

Running this script will call the gjBFGSCalc(args) method, which will print the number of line search iterations, BFGS iteration number, and the value of the WHAM equation for the new g_i for that iteration i. It will also print a plot of the value of $\rho(x)$ for the iterations given by the argument plotcheck. Here I will just

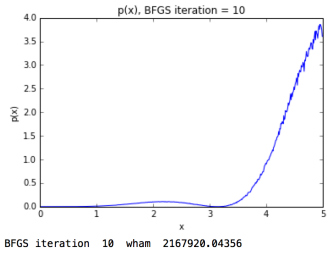
show the plots of $\rho(x)$ and the value of the WHAM equation for the given iteration.



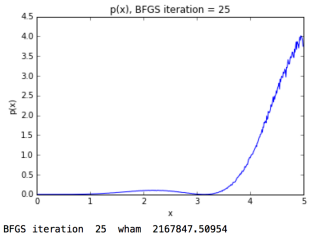
(a) $i = 2$



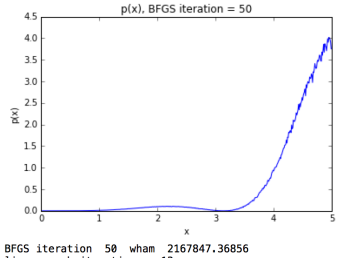
(b) $i = 5$



(c) $i = 10$



(d) $i = 25$



(e) $i = 50$

Figure 3: $\rho(x)$ at iteration number for: (a) $i = 2$, (b) $i = 5$ (c) $i = 10$, (d) $i = 25$, (e) $i = 50$

We know that by the fiftieth BFGS iteration the WHAM equation has converged, as the the value of the WHAM equation does not change and the graph matches $\rho(x) = x^2 \sin^2(x)$. I am still working on figuring out the best way to set up an error tolerance, seeing as there are times were the steps to convergence are smaller than others.

2.1 BFGS for a different $\rho(x)$

To see how this works on another function, create an instance using a different function in the SampleConstructor class. The sconXCubedSineSquared2PiX generates S simulations of random numbers chosen from the distribution

$$\rho_j(x) = C(x+2)^3 \sin^2(2\pi x) * e^{-k(x-\mu_j)^2}. \tag{4}$$

with

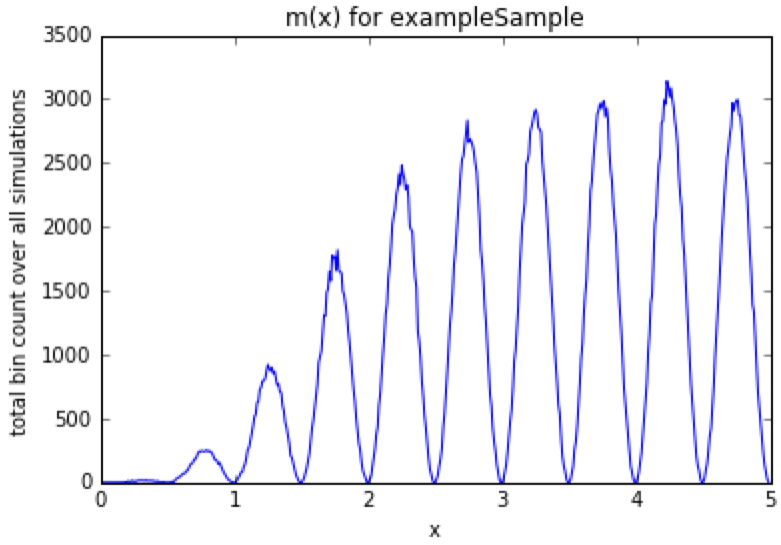
$$C = \frac{1}{(x_{max}+2)^3}, \quad \mu_j = x_{min} + j\left(\frac{x_{max} - x_{min}}{S}\right) \tag{5}$$

The code to set this simulation up should look something like:

```
#SET UP SAMPLE DATA
xmin = 0
xmax = 5
simnum = 100
binsize = 1/100
k = 1
exampleSample = WhamEqs1D(xmin,xmax,simnum,binsize,k)
n = 5000
exampleHistogram = exampleSample.sconXCubedSinSquared2PiX(n)
```

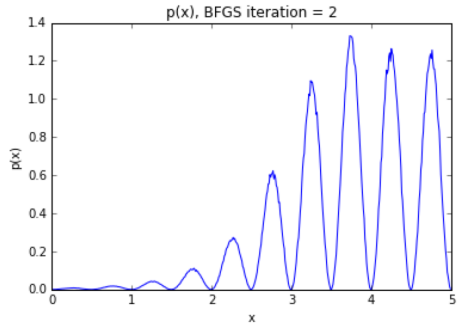
To check the total bin count over all simulations, we can check the object exampleSample's variable, m.

```
figure()
plot(exampleSample.binx,exampleSample.m)
xlabel('x')
ylabel('total bin count over all simulations')
title('m(x) for exampleSample')
show()
```

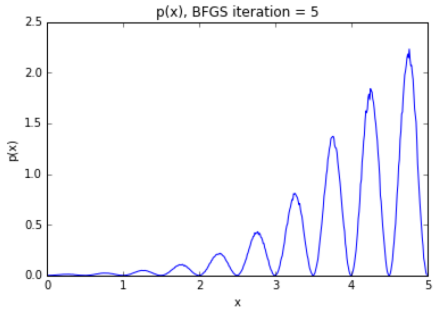


To run the BFGS method, we set up and call the method in the same way as the previous example.

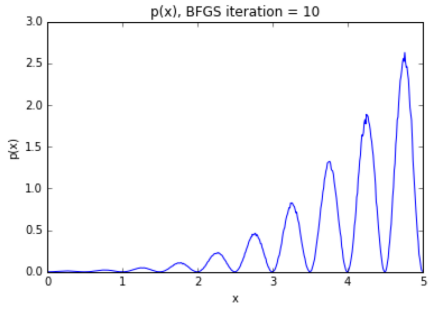
```
##SET UP FOR AND CALL BFGS METHOD
a0c = 2
betac = 0.01
taoc = 0.5
il = 150
lsl = 200
etol = 0
plotcheck = [2,5,10,25,50,100,150]
#CALL initialGCALC FIRST, the gjBFGSCalc()
g0 = exampleSample.initialGCalc()
exampleG = exampleSample.gjBFGSCalc(il,etol,a0c,taoc,betac,lsl,plotcheck)
```



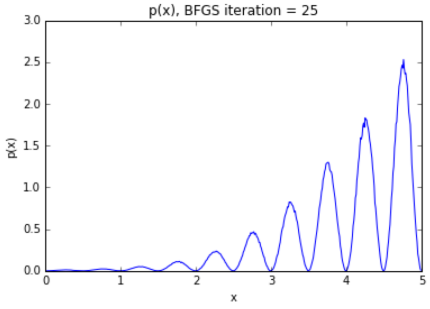
(a) $i = 2$



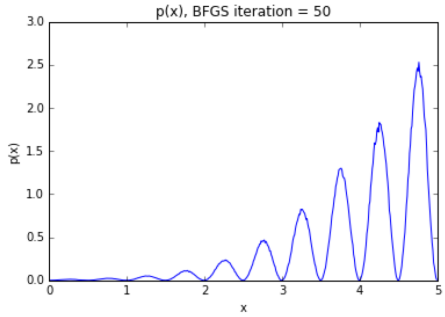
(b) $i = 5$



(c) $i = 10$



(d) $i = 25$



(e) $i = 50$

Figure 4: $\rho(x)$ at iteration number for: (a) $i = 2$, (b) $i = 5$ (c) $i = 10$, (d) $i = 25$, (e) $i = 50$