bioqtm385-inclass-07-monte-carlo-2024

November 12, 2024

##BIO/QTM 385: In class exercise for Monday, November 4th

(answers will be the part of Assignment #4, Due 11/11)

Carol Zhou Collaborated with Aanya Vusirikala and Sweta Balaji. Generative AI is used for question 1, 8, and 10. AI is used to generate the codes for deciding which N converges for question 1. For question 8 and 10, I just copied the question into AI.

As always, all questions to be answered will be in blue and places to write your answers will be in green.

[32]:
```
#import useful libraries
import numpy as np
import matplotlib.pyplot as plt
import numpy.random as random
import numpy.linalg as linalg
from scipy.stats import multivariate_normal
from scipy.stats import cauchy
import pandas as pd
```

[33]:
```
#@title Figure Settings
import ipywidgets as widgets       # interactive display
%config InlineBackend.figure_format = 'retina'
plt.style.use("https://raw.githubusercontent.com/NeuromatchAcademy/
 ↪course-content/NMA2020/nma.mplstyle")
```

0.1 Monte Carlo Estimation

As described in class, in Bayesian inference and Bayesian model selection, we are stuck evaluating complicated integrals in high numbers of dimensions that are not ammenable to analytical treatments or brute-force numerical calculations. Thus, we need to be smarter - building-up our representations through random sampling (hence, Monte Carlo). These methods, while more effective than brute force integration, can still take a while to converge, so we will explore how we might be able to tell if our samling has converged and some ideas for speeding-up the process. While we won't see those advantages in the relatively simple functions you will study today, the same methods allow for the computation of difficult integrals in high-dimensional spaces.

###A Review: Bayesian Inference and Bayesian Model Selection

As described in the lectures, if we have a model, M, with unknown parameters, θ, and a data set, X, then in Bayesian inference, we can compute the *posterior* distribution via:

$$
\begin{aligned}
p(\theta|X,M) &= \frac{p(X|\theta,M)p(\theta|M)}{p(X)} && (1)\\
&= \frac{p(X|\theta,M)p(\theta|M)}{\int p(X|\theta',M)p(\theta'|M)d\theta'}. && (2)
\end{aligned}
$$

While the numerator (the *likelihood* times the *prior*) is typically straight-forward to compute, the denominator is typically impossible (or at least really, really hard!) to compute exactly. While this difficulty does not present a huge problem in a small number of dimensions (we can numerically integrate to calculate the normalization) or when $p(X|\theta,M)p(\theta|M)$ is relatively smooth (we can find an approximate solution using fancy math tricks like the BIC), in most real-world applications, we can't compute the integral without resorting to sampling methods.

Why do we care that we can't compute the integral? After all, it's just a proportionality constant that is applied equally to all values of θ! One of the key advantages of Bayesian inference is that the output is a posterior probability distribution that we can sample from – thus we have a sense of the uncertainty of our parameter estimates given the data and our priors. Without the normalization constant, though, we can't compute the posterior distribution in full, thus limiting out knowledge about the uncertainty.

Similarly, in Bayesian model selection, when we compare two models, M_1 and M_2, we are interested in computing the ratio that we would pick one model over the other given the data:

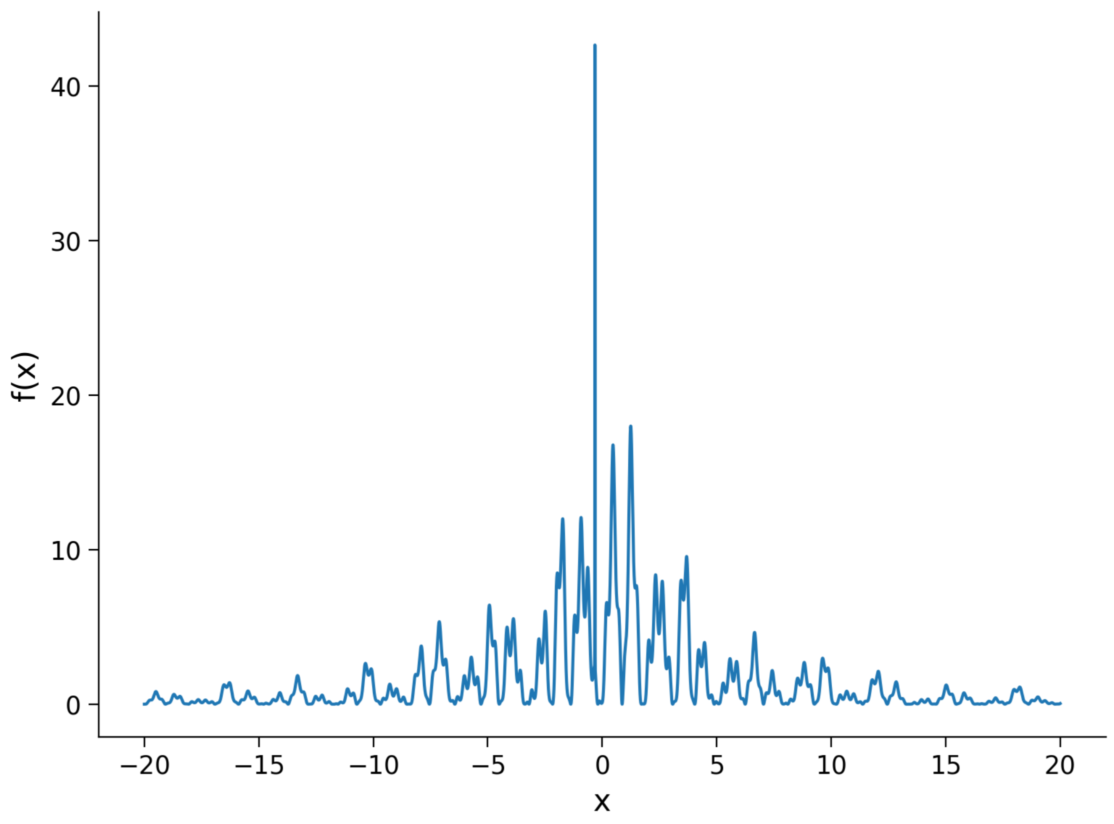
$$
\frac{p(M_1)}{p(M_2)} = \frac{p(M_1)}{p(M_2)}\frac{p(X|M_1)}{p(X|M_2)} = \frac{\int p(X|M_1,\theta)p(\theta|M_1)d\theta}{\int p(X|M_2,\theta)p(\theta|M_2)d\theta}. \quad (3)
$$

Thus, we are still tasked with computing an integral of the form $\int p(X|M,\theta)p(\theta|M)d\theta$, and we will try to figure-out how to achieve this aim in the sections to follow.

###A Made-up Example Function

Here, we will be exploring the function defined and plotted in the code below (`exampleFunction(x)`), which is defined on the interval $x \in [20, 20]$. This function can be treated as an unnormalized probability distribution (e.g., $p(x) = \frac{f(x)}{Z}$, where Z is unknown).

[34]:
```
#Run this code to define exampleFunction
def exampleFunction(x):
  z = .5*np.cos(x+1) - .4*np.cos(2.3*x) - 3*np.cos(3*x-1) + .2*np.cos(7.3*x-2)␣
↪+ 1*np.cos(5.2*x) + .5*np.cos(20.3*x) + 5*np.exp(-.5*(x+.32)**2/.005**2)
  z =  z**2/(1+(x/4)**2)
  z *= np.heaviside(x+20,0)*np.heaviside(20-x,0) + 1e-20
  return z

x = np.arange(-20,20,.0001)
plt.plot(x,exampleFunction(x))
plt.xlabel('x')
plt.ylabel('f(x)')
plt.show()
```



As you can see, this is a rather complicated function, and our hopes of integrating it analytically to find Z are rather small. Thus, we need to try to integrate the function numerically or through sampling.

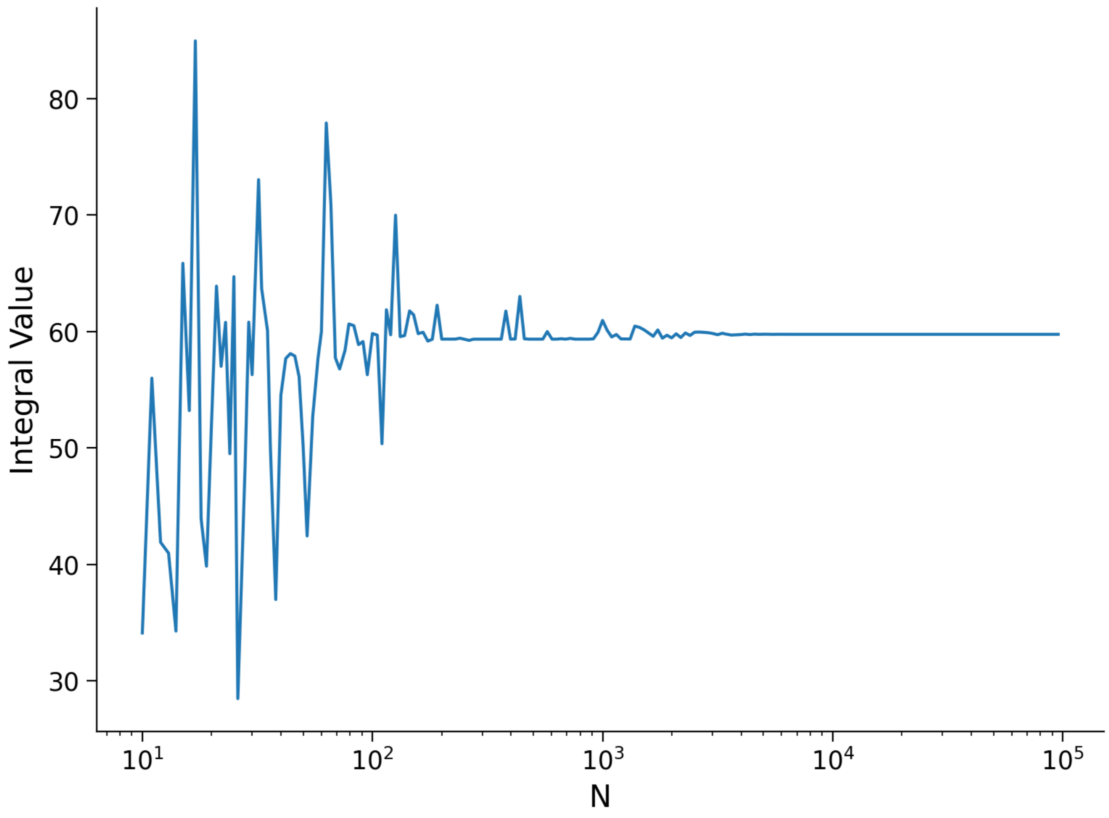
####Numerical Integration

One tactic that might work is to simply break-up the integral into chunks of size Δx and turn sum the function at values $x = -20, -20 + \Delta x, -20 + 2\Delta x, \ldots, 20 - \Delta x$. Thus,

$$\int_{-20}^{20} f(x)dx \approx \sum_{i=0}^{N=40/\Delta x-1} f(-20 + i\Delta x)\Delta x. \tag{4}$$

Question #1: Numerically evaluate the above sum for `exampleFunction` between -20 and 20 (evaluate the sum - don't use any built-in python `integrate` functions) for N = `10**np.arange(1,5,.02)`, and plot the results on a `plt.semilogx()` plot of N vs. the integral value. How large does N need to be before it converges within 1% of the result for $N = 10^5$ for all larger N?

[35]:
```
N = np.round(10**np.arange(1,5,.02))

result = np.empty(len(N))
for i in range(len(N)):
    delta_x = 40/N[i]
    x = np.arange(-20,20,delta_x)
    result[i] = np.sum(exampleFunction(x)*delta_x)

plt.semilogx(N, result)
plt.xlabel('N')
plt.ylabel('Integral Value')
plt.show()
```



[36]:
```
final_result = np.sum(exampleFunction(np.arange(-20,20,40/(10**5)))*(40/
 ↪(10**5)))
margin_of_error = 0.01 * final_result
convergence_index = None

for i in range(1, len(result)):
    if abs(result[i] - final_result) < margin_of_error:
        # Check if all subsequent values also satisfy the 1% error margin
        if np.all(np.abs(result[i:] - final_result) < margin_of_error):
            convergence_index = i
            break
```

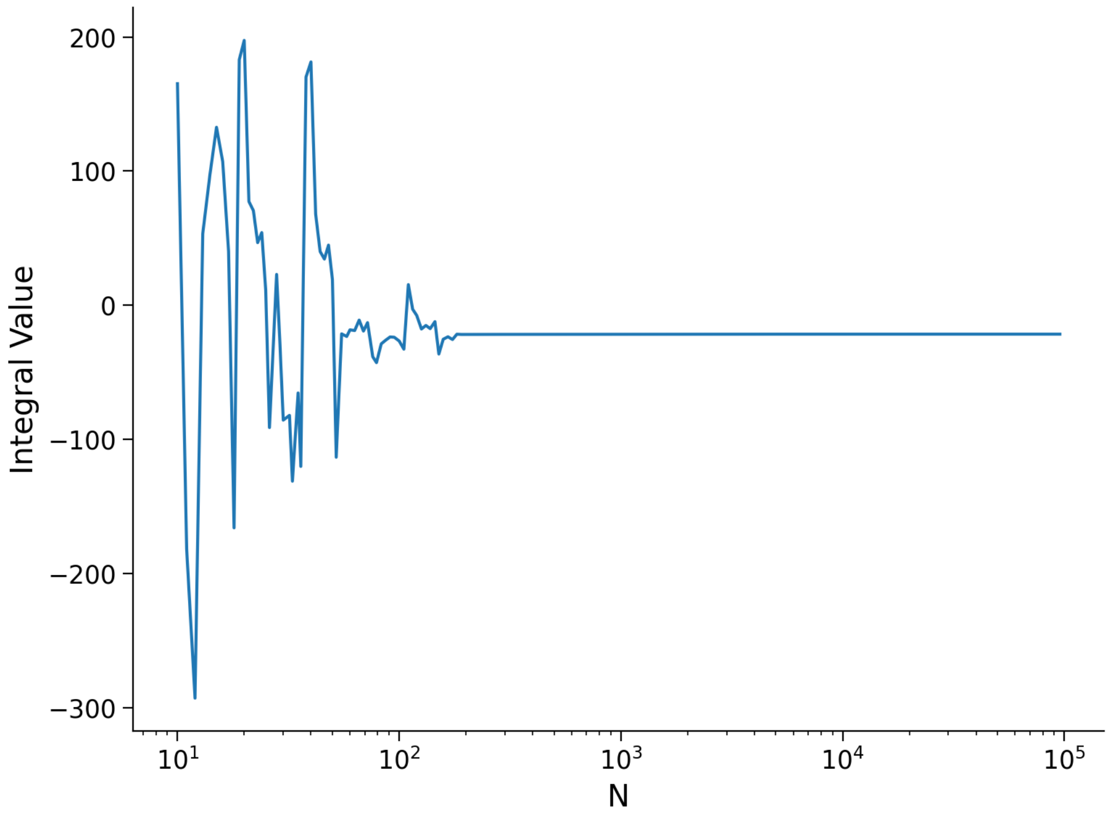
```
N[convergence_index]
```

[36]:
```
1514.0
```

How large does N need to be before it converges within 1% of the result for $N = 10^5$ for all larger N? N needs to be 1514.

Question #2: Let $\phi(x) = x^3$. Use your normalization constants from above to estimate and plot $\int_{-20}^{20} \phi(x)p(x)dx \equiv \int_{-20}^{20} \phi(x)\frac{f(x)}{Z}dx$ as a function of N (use the same range as above). How long until this expectation value converges to be within 1% of the $N = 10^5$ result for all larger N?

[37]:
```
result2 = np.empty(len(N))
for i in range(len(N)):
  delta_x = 40/N[i]
  x = np.arange(-20,20,delta_x)
  result2[i] = np.sum(x**3*(exampleFunction(x)/final_result)*delta_x)

plt.semilogx(N, result2)
plt.xlabel('N')
plt.ylabel('Integral Value')
plt.show()
```



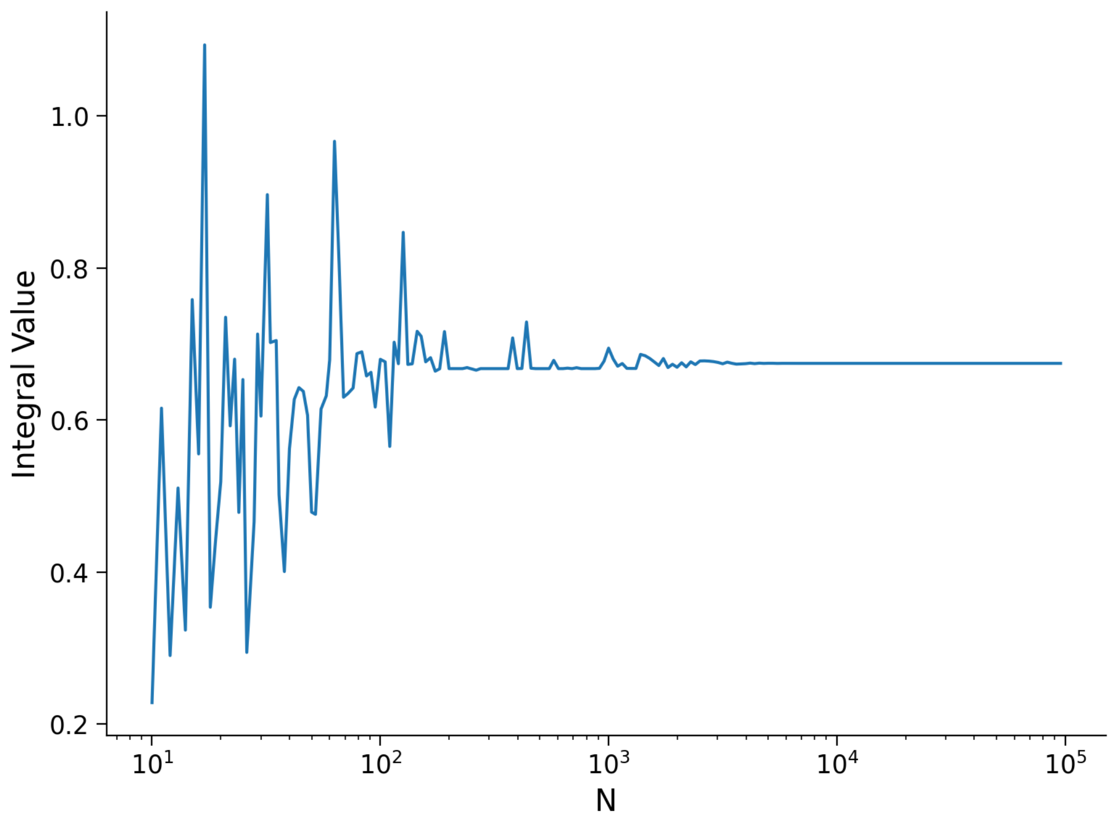
[38]:
```
final_result2 = np.sum(np.arange(-20,20,40/(10**5))**3*(exampleFunction(np.
↪arange(-20,20,40/(10**5)))/final_result)*(40/(10**5)))
margin_of_error = np.abs(0.01 * final_result2)
convergence_index = None

for i in range(1, len(result2)):
    if abs(result2[i] - final_result2) < margin_of_error:
        # Check if all subsequent values also satisfy the 1% error margin
        if np.all(np.abs(result2[i:] - final_result2) < margin_of_error):
            convergence_index = i
            break

N[convergence_index]
```

[38]:
```
182.0
```

How long until this expectation value converges to be within 1% of the $N = 10^5$ result for all larger N? It only takes N = 182 to converge.

Question #3: Now let $\phi(x) = e^{\frac{-x^2}{2\sigma^2}}$, with $\sigma = 5$. Repeat question #2. Why do you think that this did better/worse/the same than your answer to the previous question?

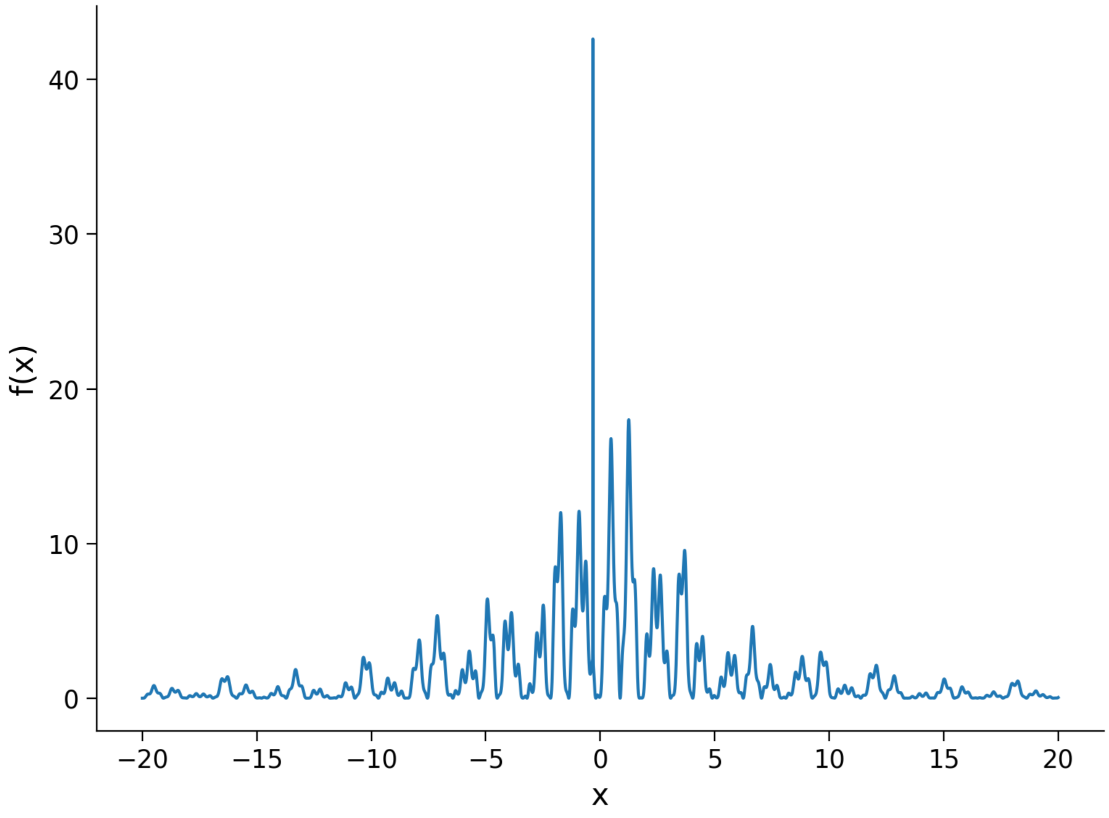
[39]:
```
result3 = np.empty(len(N))
for i in range(len(N)):
  delta_x = 40/N[i]
  x = np.arange(-20,20,delta_x)
  result3[i] = np.sum(np.exp(-x**2/(2*5**2))*(exampleFunction(x)/
↪final_result)*delta_x)

plt.semilogx(N, result3)
plt.xlabel('N')
plt.ylabel('Integral Value')
plt.show()
```

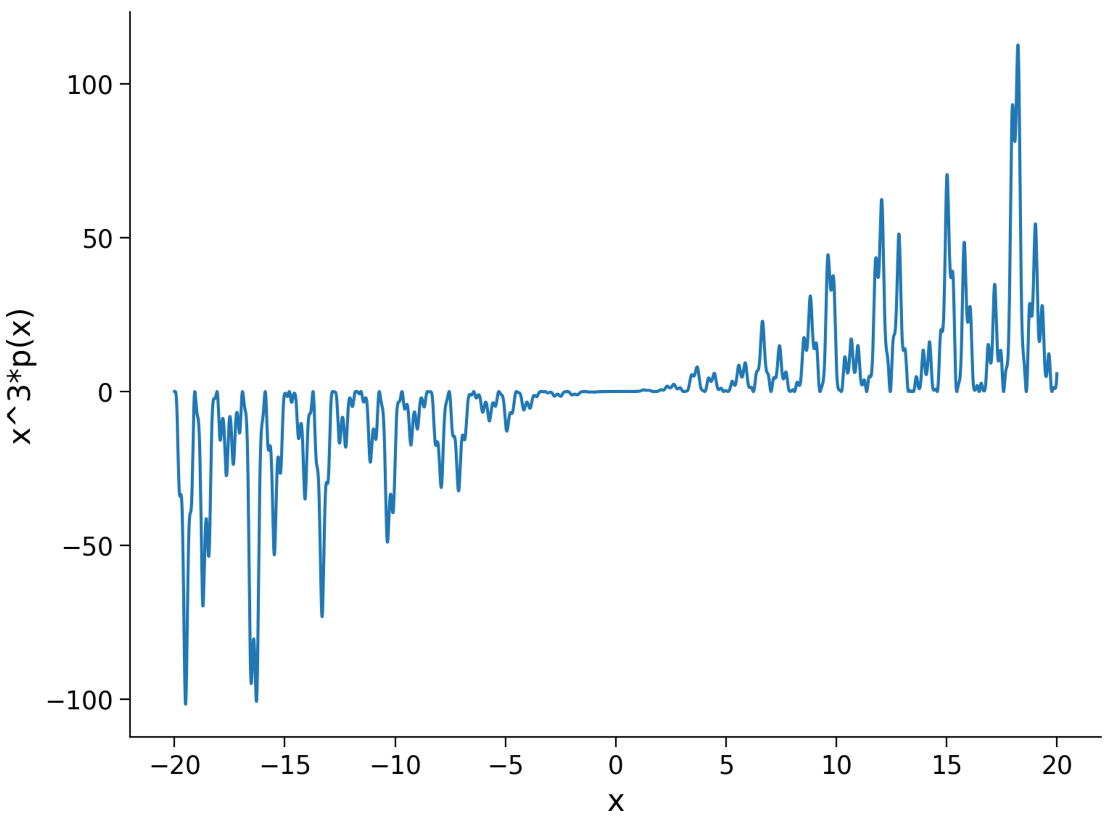


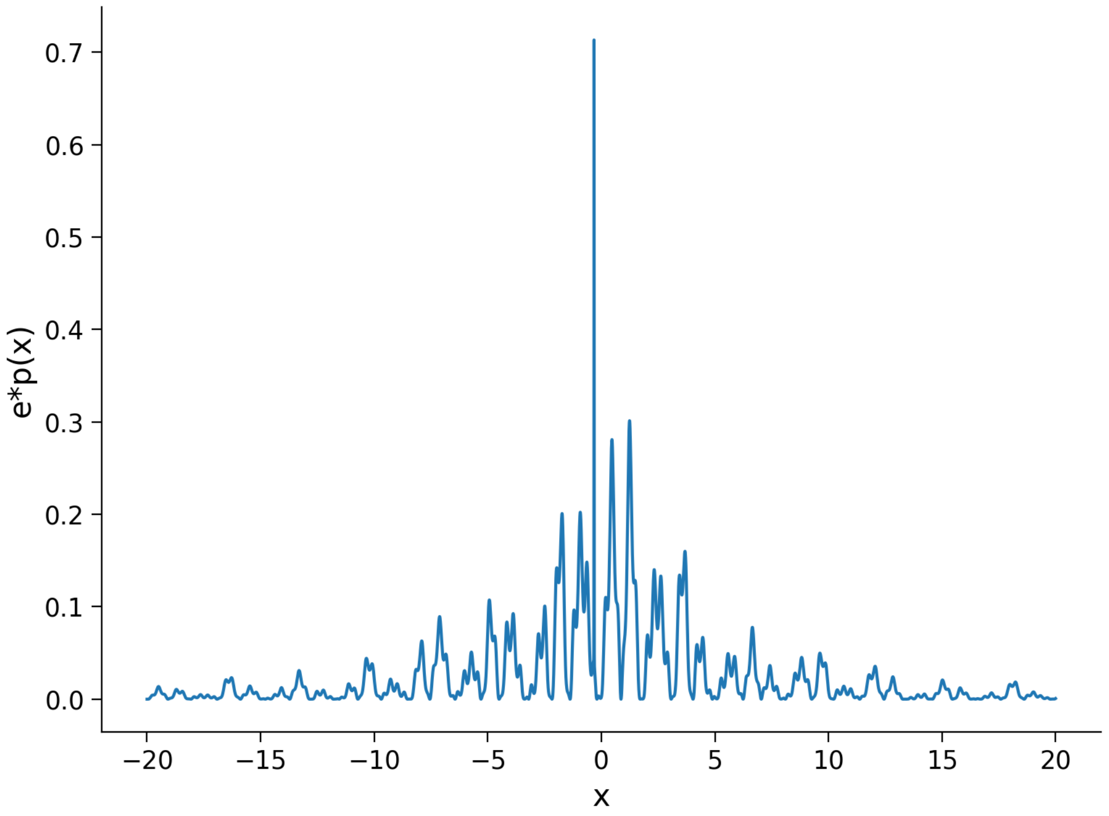
```
[40]: final_result3 = np.sum(np.exp(-np.arange(-20,20,40/(10**5))**2/
        ↪(2*5**2))*(exampleFunction(np.arange(-20,20,40/(10**5)))/final_result)*(40/
        ↪(10**5)))
      margin_of_error = np.abs(0.01 * final_result3)
      convergence_index = None

      for i in range(1, len(result3)):
          if abs(result3[i] - final_result3) < margin_of_error:
              # Check if all subsequent values also satisfy the 1% error margin
              if np.all(np.abs(result3[i:] - final_result3) < margin_of_error):
                  convergence_index = i
                  break

      N[convergence_index]
```

```
[40]: 1514.0
```

```
[41]: plt.plot(x,exampleFunction(x))
      plt.xlabel('x')
      plt.ylabel('f(x)')
      plt.show()
      plt.plot(x,x**3*exampleFunction(x)/final_result)
      plt.xlabel('x')
      plt.ylabel('x^3*p(x)')
      plt.show()

      plt.plot(x,exampleFunction(x)/final_result)
      plt.xlabel('x')
      plt.ylabel('e*p(x)')
      plt.show()
```



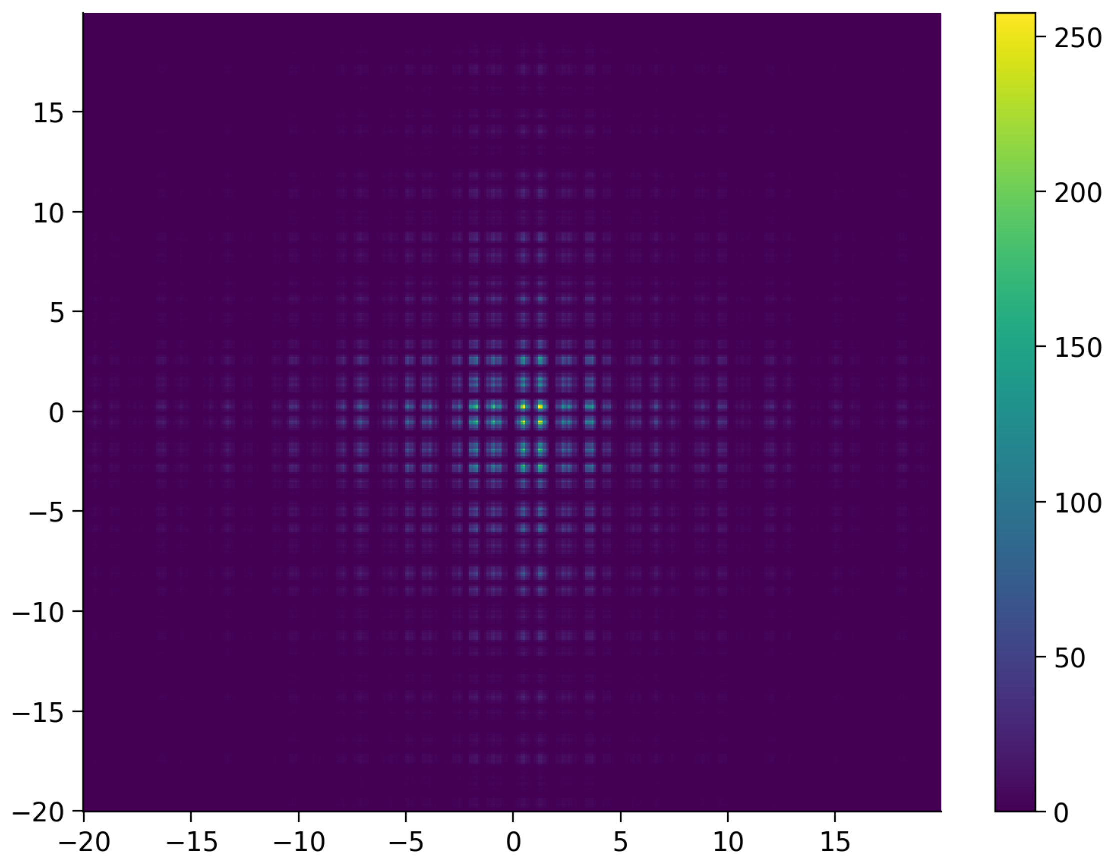

x^3*p(x)
100
50
0
−50
−100
−20
−15
−10
−5
0
5
10
15
20
x



Why do you think that this did better/worse/the same than your answer to the previous question? This did worse than the previous question because we multiply our example function with an exponetial function that basically just scale down our original function. The overall shape is the same, and thus, the value of N for convergence is the same. On the other hand, in Question 2, we multiply the example function with x^3, which makes the y values near x = 0 zeros. Thus, this is why it is easier to approximate (since the middle part is just flat) and quicker to converge.

Now, let's imagine we have a 2-dimensional function, $g(x, y) = f(x)g(y + 1)$, as shown and plotted in the cell below.

[42]:
```
def exampleFunction2d(x,y):
  return exampleFunction(x)*exampleFunction(y+1)

x = np.arange(-20,20,.1)
y = np.arange(-20,20,.1)
X,Y = np.meshgrid(x,y)
Z = exampleFunction2d(X,Y)
im = plt.pcolormesh(x,y,Z)
plt.colorbar()
plt.show()
```



To calculation the integral of $g(x, y)$ numerically, we could perform the same approximation as before, now just with an additional summation:

$$\int_{-20}^{20}\int_{-20}^{20} g(x,y)dxdy \approx \sum_{i=0}^{N_x=40/\Delta x-1}\sum_{j=0}^{N_y=40/\Delta y-1} g(-20+i\Delta x, -20+j\Delta y)\Delta x\Delta y. \tag{5}$$

Question #4: Using numerical integration, how many times do you anticipate having to evaluate $g(x, y)$ in order to obtain an accurate (within 1%) estimate of the integral? (Note: no need to actually simulate - just make your best guess of the answer from evidence obtained in your answer to Question #1)

Write your answer to Question #4 here. Since you are multiplying the example function by itself, using my answer to Question #1, the maximum number of times for it to converge will be 1514^2 = 2.29 x 10^6. This is because we have two independent dimensions and the total number of evaluations required will scale quadratically.

Question #5: The same as above in Question #5, but now for estimating $\int_{-20}^{20}\int_{-20}^{20} \zeta(x,y)g(x,y)dxdy$, with $\zeta(x,y) \equiv x^3y^3$?

Write your answer to Question #5 here. Since you are multipling the same function from Question #2 by itself, the maximum number of times for it to converge will be 182^2 = 33124. This is because, again, we have two independent dimensions and the total number of evaluations required will scale quadratically.

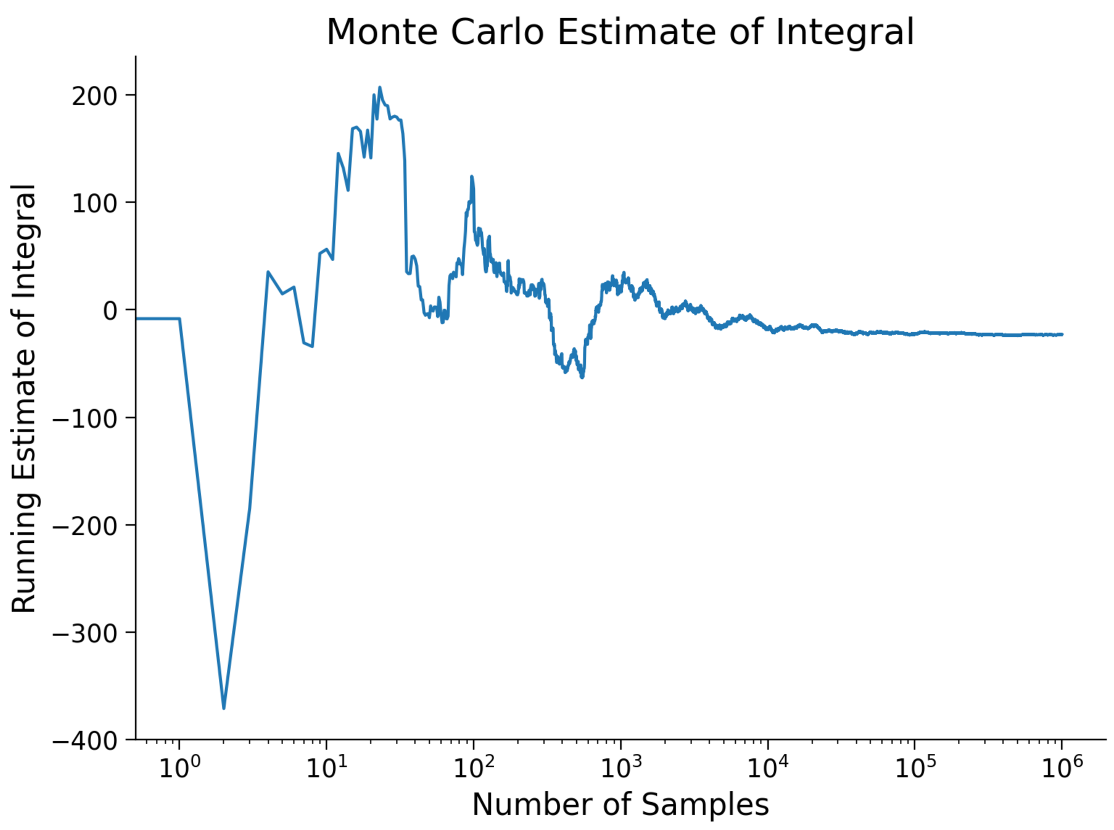
####Random Sampling

Returning back to our 1D function, `exampleFunction()`, we could also try to compute the integral through random sampling, uniformly, on the interval $x \in [-20, 20]$.

Question #6: Select 1,000,000 random samples out of a uniform distribution between -20 and 20 (`random.uniform()` is the function of interest). Plot a running talley of your estimate of $\int_{-20}^{20} \phi(x) f(x) dx$ as a function of the number of samples using `np.cumsum()`. By the end, has your solution converged to within 1% of your value for $N = 10^5$ in Question #2? If so, did the convergence take more, fewer, or the same number of function evaluations compared to your answers to Question #2 and Question #3?

(Hint: you can compute the estimate by taking the mean value of your $f(x_i)\phi(x_i)$ draws and divide by the mean value of your $f(x_i)$ draws to create a weighted average)

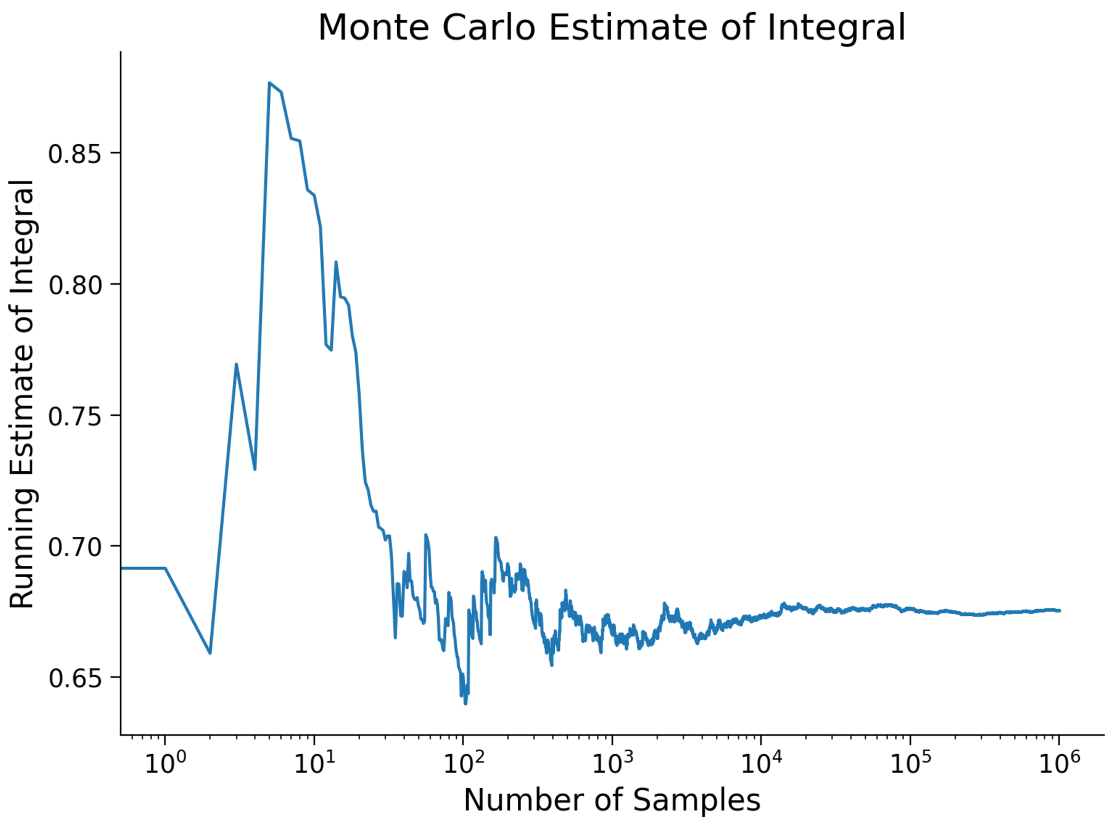
[43]:
```
x_samples = np.random.uniform(-20, 20, 1000000)

fx = exampleFunction(x_samples)

phi = x_samples**3

running_tally = np.cumsum(phi*fx) / np.cumsum(fx)

plt.semilogx(running_tally)
plt.xlabel('Number of Samples')
plt.ylabel('Running Estimate of Integral')
plt.title('Monte Carlo Estimate of Integral')
plt.show()
```



[44]:
```
margin_of_error = np.abs(0.01 * final_result2)
convergence_index = None

for i in range(1, len(running_tally)):
    if abs(running_tally[i] - final_result2) < margin_of_error:
        # Check if all subsequent values also satisfy the 1% error margin
        if np.all(np.abs(running_tally[i:] - final_result2) < margin_of_error):
            convergence_index = i
            break

print(convergence_index)
```

```
None
```

[45]:
```
phi = np.exp(-x_samples**2/(2*5**2))

running_tally_e = np.cumsum(phi*fx) / np.cumsum(fx)

plt.semilogx(running_tally_e)
plt.xlabel('Number of Samples')
plt.ylabel('Running Estimate of Integral')
plt.title('Monte Carlo Estimate of Integral')
plt.show()
```



[48]:
```
margin_of_error = np.abs(0.01 * final_result3)
convergence_index = None

for i in range(1, len(running_tally_e)):
    if abs(running_tally_e[i] - final_result3) < margin_of_error:
        # Check if all subsequent values also satisfy the 1% error margin
        if np.all(np.abs(running_tally_e[i:] - final_result3) <␣
 ↪margin_of_error):
            convergence_index = i
            break

print(convergence_index)
```

```
5515
```

By the end, has your solution converged to within 1% of your value for $N = 10^5$ in Question #2? If so, did the convergence take more, fewer, or the same number of function evaluations compared to your answers to Question #2 and Question #3? When phi_x = x^3, my solution does not converge within 1% of my value for N = 10^5 in Question 2. But when phi_x = e, my solution has converged to within 1% of your value for N=10^5 in Question #3. But it took more times (5515 times) compared to my both answers to Question 2 and 3.

####Importance Sampling

As you should have seen in the previous question, uniform sampling usually won't help us much, so we need to find a new strategy. The next potential idea is to make a guess for $p(x)$, the underlying distribution. Let's call that guess $q(x)$. The idea is that we can easily sample from $q(x)$, and if that if $q(x)$ allows for a better sampling of $p(x)$ than a uniform distribution, we can create a weighted average, where samples from q from areas where $f(x) = p(x) * Z$ is large are enhanced and samples from q where $f(x)$ is relatively small are diminished.

More precisely, we define:

$$w_i = \frac{f(x_i)}{q(x_i)}, \tag{6}$$

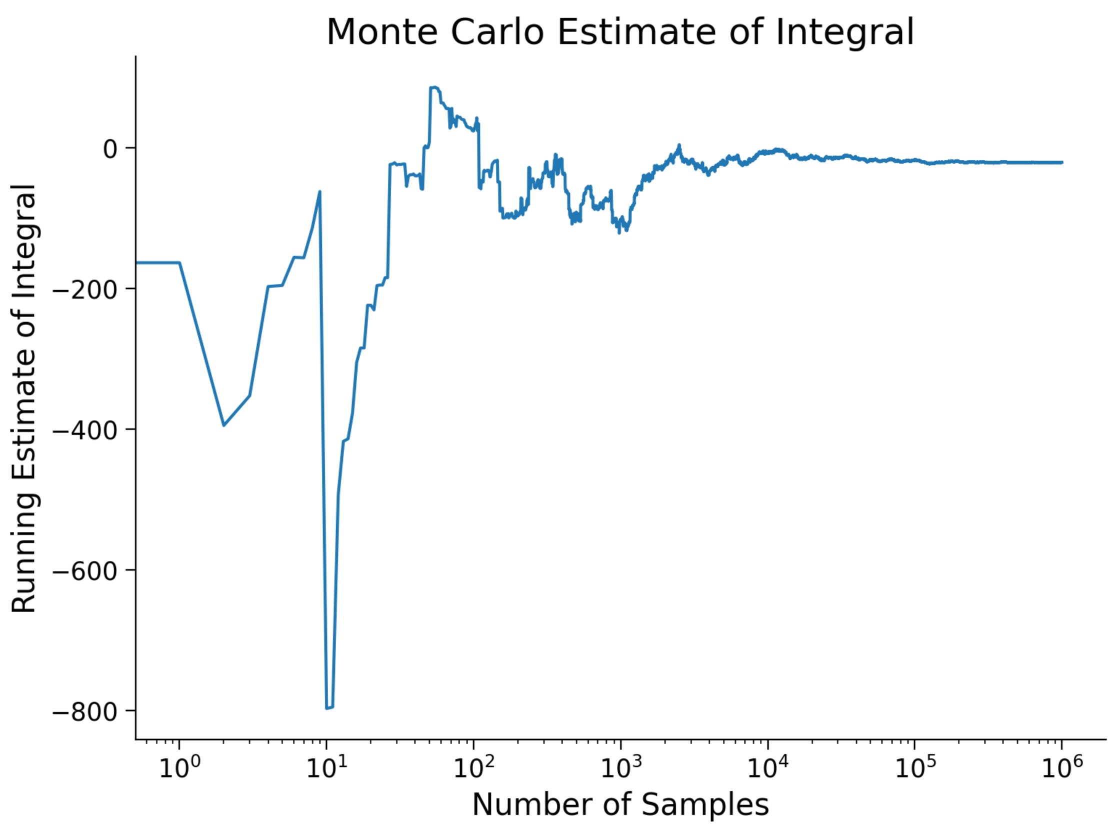
where x_i is the i^{th} draw from $q(x)$. Thus, to compute an expectation value, we can compute the weighted sum:

$$\hat{\Phi} = \frac{\sum_{i=1}^{N} w_i \phi(x_i)}{\sum_{i=1}^{N} w_i}. \tag{7}$$

Question #7: Let $q(x) = N(0, 100)$, a normal distribution with mean of zero and variance (σ^2) equal to 100. Generate $N=$1,000,000 samples from q and plot a running tally of your estimate for $\int_{-20}^{20} \phi(x)p(x)dx$. Has your estimate or your convergence time improved over uniform sampling?

[49]:
```python
from scipy.stats import norm

x_samples = np.random.normal(0, 10, 1000000)

fx = exampleFunction(x_samples)
qx = norm.pdf(x_samples, 0, 10)

phi = x_samples**3

w = fx / qx

running_tally2 = np.cumsum(w*phi) / np.cumsum(w)

plt.semilogx(running_tally2)
plt.xlabel('Number of Samples')
plt.ylabel('Running Estimate of Integral')
plt.title('Monte Carlo Estimate of Integral')
plt.show()
```

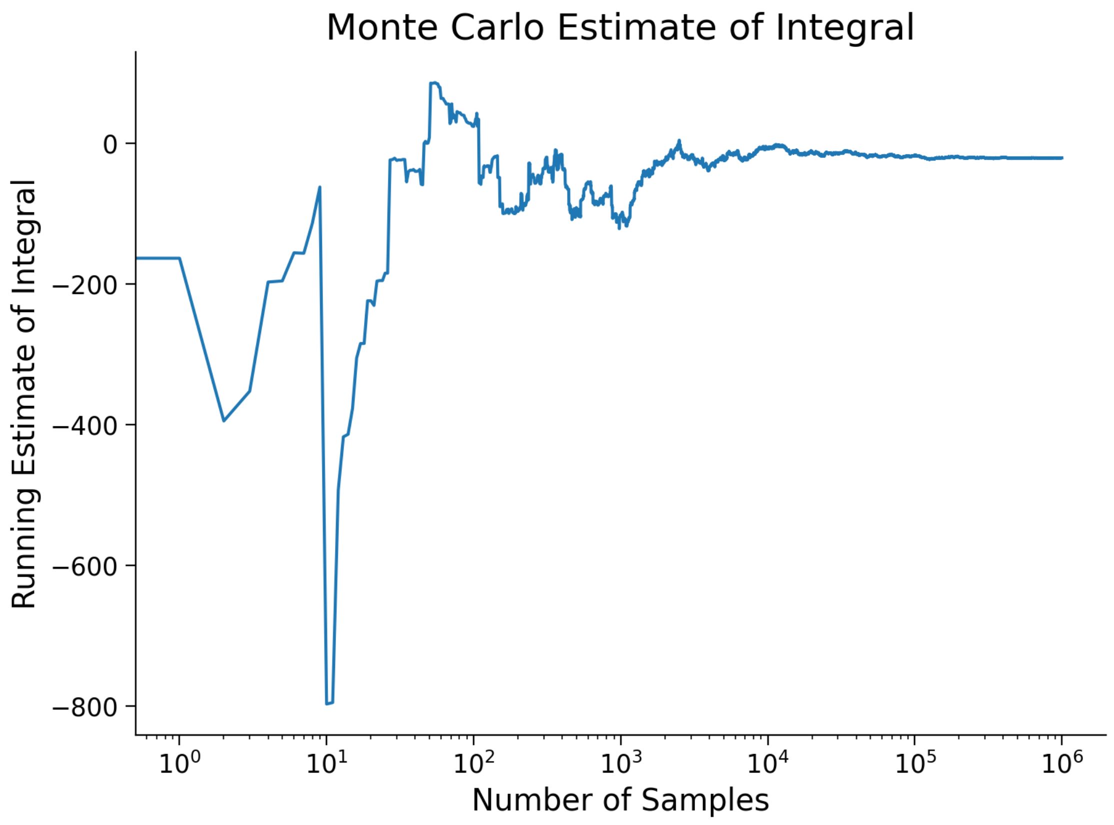
```
xs = random.normal(0,10,size=(1000000,1))
fs = exampleFunction(xs)
phis = xs**3
qs = np.exp(-.5*xs**2/100**2)/np.sqrt(2*np.pi*100)
ws = fs / qs
estimates_importance = np.cumsum(ws*phis)/np.cumsum(ws)
converged_val = estimates_importance[-1]
```



[51]:
```
margin_of_error = np.abs(0.01 * final_result2)
convergence_index = None

for i in range(1, len(running_tally2)):
    if abs(running_tally2[i] - final_result2) < margin_of_error:
        # Check if all subsequent values also satisfy the 1% error margin
        if np.all(np.abs(running_tally2[i:] - final_result2) < margin_of_error):
            convergence_index = i
            break

print(convergence_index)
```

```
None
```

[52]:
```
phi = np.exp(-x_samples**2/(2*5**2))

running_tally2_e = np.cumsum(w*phi) / np.cumsum(w)

plt.semilogx(running_tally2)
plt.xlabel('Number of Samples')
plt.ylabel('Running Estimate of Integral')
plt.title('Monte Carlo Estimate of Integral')
plt.show()
```



[53]:
```
margin_of_error = np.abs(0.01 * final_result3)
onvergence_index = None

for i in range(1, len(running_tally2_e)):
    if abs(running_tally2_e[i] - final_result3) < margin_of_error:
        # Check if all subsequent values also satisfy the 1% error margin
        if np.all(np.abs(running_tally2_e[i:] - final_result3) <␣
  ↪margin_of_error):
            convergence_index = i
            break

print(convergence_index)
```

```
13979
```

Has your estimate or your convergence time improved over uniform sampling? When phi_x = x^3, my solution does not converge within 1% of my value for N = 10^5 in Question 2. Thus, my convergence time has not improved over uniform sampling. Though when phi_x = e my solution has converged to within 1% of my value for N=10^5 in Question #3, my convergence time has not improved over uniform sampling either.

##Metropolis-Hastings Sampling

Importance sampling generates new samples out of the global distribution $q(\vec{x})$, but often it is beneficial to sample locally - spending more time around areas of particularly high probability. This is where Markov Chain Monte Carlo (MCMC) methods can be particularly useful. For now, we will start with Metropolis-Hastings sampling.

In Metropolis-Hastings sampling, we start from a randomly-chosen starting point (usually randomly-chosen from our prior), $\vec{x}^{(0)}$. From here, we iterate through the dimensions of $\vec{x}^{(0)}$, selecting a new point each time or sticking with the old one. There are many ways of choosing the next point to test, but for now, let's say that we choose a new point, $\tilde{x}_i$, out of a Gaussian distribution with mean $x_i^{(0)}$ and variance σ^2 ($\mathcal{N}(x_i^{(0)},\sigma^2)$). We then decide whether to keep this suggested new point or not in our sampled data within the context of the algorithm below:

0) Initialize σ
1) Initialize $\vec{x}^{(t)}$ and calculate $f(\vec{x}^{(t)})$ ($\vec{x}^{(t)} \in R^p$)
2) Let $\vec{x}^{(t)} \to \vec{x}^{(t+1)}$
3) For $i = 1, 2, \ldots, p$:
    (i) $\vec{x}^{(t+1)} \to \tilde{x}$
    (ii) $x_i^{(t)} + \eta \to \tilde{x}_i$ (η drawn from $N(0,\sigma^2)$)
    (iii) Calculate $a \equiv \frac{f(\tilde{x})}{f(\vec{x}^{(t+1)})}$
    (iv) If $a \geq 1$, then $\tilde{x} \to \vec{x}^{(t+1)}$
    (v) If $a < 1$, then $\tilde{x} \to \vec{x}^{(t+1)}$ with probability a (no change otherwise)
4) $t + 1 \to t$
5) Repeat Steps 2-4 until you decide to stop

Once a set of $\{\vec{x}^{(t)}\}_{t=1,\ldots,N}$ are obtained, then we usually assume that there is a "burn-in" time, m, that represents the effect of our initial value on the simulation, and we can calculate an expectation value of the function $\phi(\vec{x})$ via:

$$\hat{\Phi} = \frac{1}{N-m}\sum_{t=m+1}^{N} \phi(\vec{x}^{(t)}) \tag{8}$$

Question #8: Write a function, `runMetropolisHastingsExample(N,sigma)`, that implements the algorithm above for the 1d `exampleFunction(x)`, running for `N` times through the loop (Steps 2-5) with σ = `sigma`. The code should pick an initial value $x^{(0)}$ at random from [-1,1] and should return `xs`, an (`N`+1) $\times$ 1 array of the simulated $x^{(t)}$. (Note: since the example function is 1-d ($p = 1$) you don't need to implement the `for` loop in Step 3 above)

[83]:
```python
def runMetropolisHastingsExample(N,sigma):
    #step 1
    x = np.random.uniform(-1,1)
    xs = np.zeros(N+1)
```

```
  xs[0] = x

  #step 2
  for t in range(N):
    x_tilde = np.random.normal(x,sigma)
    f_x = exampleFunction(x)
    f_x_tilde = exampleFunction(x_tilde)

    #step 3
    a = f_x_tilde / f_x
    if a >= 1:
      x = x_tilde
    else:
      u = np.random.uniform(0,1)
      if u <= a:
        x = x_tilde

    #step 4
    xs[t+1] = x

  return xs
```

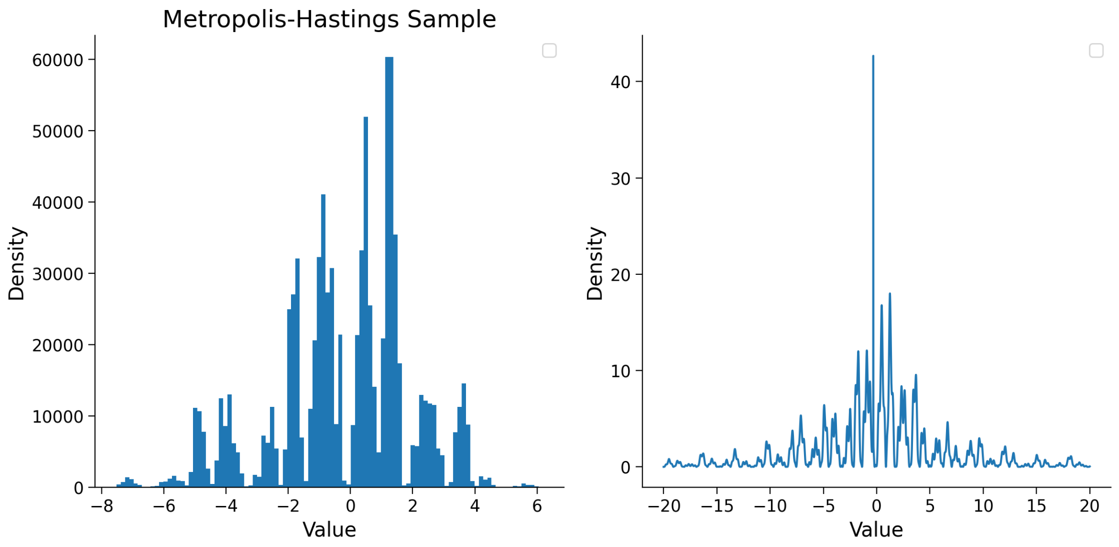
Question #9: Run your simulation code above for σ =.05 and $N=$1,000,000, and plot a histogram of the resulting values (assume a burn-in time of $m=$100,000). Do your results look simlar to the original function? Would you prefer Monte Carlo simulation or numerical integration in a single dimension? Why?

[88]:
```
resulting_values = runMetropolisHastingsExample(1000000, 0.05)
```

[89]:
```
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 6))

ax1.hist(resulting_values[100000:], bins=100)
ax1.set_xlabel('Value')
ax1.set_ylabel('Density')
ax1.set_title('Metropolis-Hastings Sample')
ax1.legend()

x = np.arange(-20, 20, 0.0001)
ax2.plot(x, exampleFunction(x))
ax2.set_xlabel('Value')
ax2.set_ylabel('Density')
ax2.legend()

plt.tight_layout()
plt.show()
```

```
WARNING:matplotlib.legend:No artists with labels found to put in legend.  Note
that artists whose label start with an underscore are ignored when legend() is
```

```
called with no argument.
WARNING:matplotlib.legend:No artists with labels found to put in legend.  Note
that artists whose label start with an underscore are ignored when legend() is
called with no argument.
```



Do your results look simlar to the original function? Would you prefer Monte Carlo simulation or numerical integration in a single dimension? Why? Yes, my result look similar to the original function since they both have a peak near the 0 (though the peak of my result is at the positive end while the peak of the original function is at the negative end). I would prefer Monte Carlo simulation in a single dimension for complex problems (which is the case here) when direct integration is infeasible because numerical integration could be computationally expensive or impossible. Monte Carlo, in this case, provide a good estimate of the underlying structure of the original function. However, I would prefer just using numerical integration for smooth functions since it would be fasters and more accurate than Monte Carlo.

###Testing for convergence

In real-world examples, we typically don't know what the answer is (this is why we do the numerical calculation!), so we lack a ground-truth to convince us that our results have converged. The general idea is that if we were to run the simulation many times, we would like the variation between chains to be comparable to the variation within a single chain. The most common statistical test for this type of convergence is the Gelman-Rubin convergence diagnostic.

To be more precise, let's imagine that we make M independent Monte Carlo simulations, each run for $n + N$ time steps (n is our burn-in time, or the number of steps we first take to make sure that the chain has converged onto the distribution from the random initial condition). Let $\vec{\mu}^{(i)}$ and $\sigma^{(i)}$ be the mean and standard deviation of the post burn-in data for simulation i. In addition, let's define $\bar{\mu}$ as the mean of all of the $\vec{\mu}_i$ values.

Thus, if we focus on one of the dimensions, i, that we are interested in, we can define the **between**

simulation variance (B) via the variance in the simulations mean values:

$$B = \frac{N}{M-1}\sum_{m=1}^{M}(\mu_i^{(m)} - \bar{\mu}_i)^2 \tag{9}$$

and the **within simulation** variance (W) is given by the average of the post-burn-in variances within each simulation:

$$W = \frac{1}{M}\sum_{m=1}^{M}(\sigma_i^{(m)})^2. \tag{10}$$

Ideally, we would compute these quantities for all dimensions, but we will focus on the 1-d problem today for the sake of simplicity.

Unfortunately, we can't just look at the ratios of these quantities for a variety of complicated reasons that are beyond the scope of this class (see here for details). We can, however, write-down an unbiased estimator of the posterior variance of our estimated parameter as the pooled variance, V:

$$V = \frac{N-1}{N}W + \frac{M+1}{MN}B. \tag{11}$$

V serves as an effective between simulation variance measure.

Given these quantities (and correcting for the number of dimensions, p, in our estimate, a statistical comparison between the within and between simulation variance is given via the **Gelman-Rubin Convergence Diagnostic**:

$$R_c = \sqrt{\frac{p+3}{p+1}\frac{V}{W}} \tag{12}$$

While somewhat difficult to derive, this diagnostic is relatively easy to compute, and we generally say that a simulation has converged if $R_c < 1.1$.

While it might seem numerically expensive to run the simulation M times, the simulations are all completely independent from each other. Accordingly, they can be run in parallel, so on a large system, running multiple instantiations often takes a similar amount of real-world time as running a single chain (although not a similar amount of total CPU time, naturally). If you want to try parallelizing your code for the question below, a tutorial is here.

Question #10: Using your function from Question #8, run $M = 5$ simulations of length $N =$ 220,000 ($\sigma =$.05). Calculate R_c. Would you say that your simulation has converged? Assume a burn-in time of $n = $ 20,000 time steps. (If you're curious, you can try to increase N to see if you can lower R_c)

[90]:
```python
# Parameters
N = 220000  # Number of steps for each chain
sigma = 0.05  # Proposal distribution std deviation
burn_in = 20000  # Burn-in steps
M = 5  # Number of independent simulations

# Run M simulations
chains = [runMetropolisHastingsExample(N, sigma) for _ in range(M)]

# Remove burn-in from chains
```

```
chains_burned = [chain[burn_in:] for chain in chains]

# Means and standard deviations for each chain
means = np.array([np.mean(chain) for chain in chains_burned])
std_devs = np.array([np.std(chain) for chain in chains_burned])

# Mean of means
mean_of_means = np.mean(means)

# Between-chain variance B
B = (N / (M - 1)) * np.sum((means - mean_of_means)**2)

# Within-chain variance W
W = np.mean(std_devs**2)

# Pooled variance V
V = ((N - 1) / N) * W + (M + 1)/ (M * N) * B

# Gelman-Rubin convergence diagnostic Rc
p = 1  # Dimension is 1 for this example
Rc = np.sqrt(((p + 3) / (p + 1)) * V / W)

# Output the result
print("Gelman-Rubin Convergence Diagnostic (Rc):", Rc)
```

```
Gelman-Rubin Convergence Diagnostic (Rc): 2.429755825787166
```

Would you say that your simulation has converged? Since the value of Rc is not smaller than 1.1, I would say that my simulation has not converged.

Instructions on how to save PDF files: (repeated from last time)

To make nice looking files to hand-in (which makes grading much easier!), please follow the instructions below.

First, run this code to install texlive and to link your google drive (it will ask your permission to do so)

[91]:
```
!sudo apt-get install texlive-xetex texlive-fonts-recommended␣
↪texlive-plain-generic pandoc

from google.colab import drive
drive.mount('/content/drive')
```

```
Reading package lists… Done
Building dependency tree… Done
Reading state information… Done
The following additional packages will be installed:
  dvisvgm fonts-droid-fallback fonts-lato fonts-lmodern fonts-noto-mono
  fonts-texgyre fonts-urw-base35 libapache-pom-java
```