Real Numbers on a Computer and Importance Sampling

Matthew Seguin

Preliminary Importing of Packages

```
import numpy as np
from scipy import stats
from bitstring import Bits
import matplotlib.pyplot as plt
```

1.

Recall that 64 bit floating point numbers are stored in the form $(-1)^S \times (1.d) \times 2^{e-1023} = (-1)^S \times 1.d_1 d_2 ... d_{52} \times 2^{e-1023}$ where S is the first bit representing the sign, e is the integer value coming from the next eleven bits used to adjust the magnitude of the number, and d is the remaining 52 bits used for representing the digits of that number. Note that d is a base 2 number not in base 10.

First defining a function for seeing the bit representation:

```
def bits(x, type='float', len=64):
    if type == 'float':
        obj = Bits(float = x, length = len)
    elif type == 'int':
        obj = Bits(int = x, length = len)
    else:
        return None
    return(obj.bin)
```

a.

Making our float64 variable for infinity and checking its bits.

```
x = np.float64(10)
y = x**309
```

```
RuntimeWarning: overflow encountered in scalar power y = x**309
```

```
y
```

```
inf
```

```
bits(y)
```

```
'0111111111110000000000000000000000000000000000000000000000000000'
```

This essentially shows that the exponent is as large as it can be for infinity. If these bits were used to store a finite number the exponent would be $1 \times 2^{10} + 1 \times 2^9 + ... + 1 \times 2^1 + 1 \times 2^0 = 2047$ which would mean the finite number this **would** represent is $(-1)^0 \times (1.0) \times 2^{2047-1023} = 2^{1024}$ but these bits are taken up to represent infinity. So we can't actually represent 2^{1024} since there are some bit patterns that are reserved for positive and negative infinity.

b.

```
bits(2**-1022)
```

```
'0000000000010000000000000000000000000000000000000000000000000000'
```

This essentially says that $(-1)^0 \times (1.0) \times 2^{1-1023} = 2^{-1022}$. Based on this the bits for 2^{-1023} should be represented by $(-1)^0 \times (1.0) \times 2^{0-1023} = 2^{-1023}$ which in bits would be just 64 zeros.

c.

Now we are not working with normalized numbers.

I see that denormalized numbers are stored as $0.B \times 2^{e-1022}$ where B is now the 52 bits at the end (still in base 2) and e is always just 11 zeros. Just based off of this the smallest positive number we could theoretically get would be represented by all zeros and a 1 in the final spot which would be $2^{-52} \times 2^{-1022} = 2^{-1074}$. This is consistent with what I got from testing.

Here is the progression of every positive number no greater than 2^{-1023} that can be represented exactly.

```
bits(2**-1023)
```

```
'0000000000001000000000000000000000000000000000000000000000000000'
```

This is not the same as what we predicted in part b, now what this says is $(-1)^0 \times (2^{-1}) \times 2^{-1022} = 2^{-1023}$.

Continuing we have:

```
bits(2**-1024)
```

```
'0000000000000100000000000000000000000000000000000000000000000000'
```

```
bits(2**-1025)
```

```
'0000000000000010000000000000000000000000000000000000000000000000'
```

```
bits(2**-1026)
```

```
'0000000000000001000000000000000000000000000000000000000000000000'
```

It continues onward, moving the 1 to the right each time until we get near the smallest positive number

```
bits(2**-1073)
```

```
'0000000000000000000000000000000000000000000000000000000000000010'
```

```
bits(2**-1074)
```

```
'0000000000000000000000000000000000000000000000000000000000000001'
```

```
bits(2**-1075)
```

```
'0000000000000000000000000000000000000000000000000000000000000001'
```

As we can see the bit representation does not change going from 2^{-1074} to 2^{-1075} showing that 2^{-1074} is indeed the smallest positive number we can represent. This is consistent with the theoretical result we saw before that $2^{-1074} \approx 4.940656 \times 10^{-324}$ should be the smallest positive number we can represent in this way.

2.

```
rng = np.random.default_rng(seed = 1)

def dg(x, form = '.20f'):
  print(format(x, form))

z = rng.normal(size = 100)
x = z + 1e12
## Calculate the empirical variances
dg(np.var(z))
```

```
0.72514887009499828796
```

```
dg(np.var(x))
```

```
0.72514631554484365594
```

Recall that for 64 bit floating point numbers there is only accuracy up to the 16th digit. This is not just for decimal places it is for any digits in a number. So while the relative error stays the same at every magnitude (just machine epsilon) the absolute error increases linearly in the magnitude of the number.

More precisely for this example:

We start with z, a vector of size 100, containing data from a standard normal accurate up to the 16th digit (which will most of the time just be the 16th decimal place but even in bad scenarios will only be the 15th decimal place). Therefore when we calculate the variance using z our result will be accurate to about 16 decimal places (but again 15 in bad cases).

Then x is just the result of adding 1×10^{12} to all of these numbers. Mathematically this does not change the variance. However when doing computation on a computer there is a slight change in the results. This is because the number we are adding to each value (1×10^12) while it still has 16 **digits** of accuracy it does not have accuracy up to the 16th decimal place. Rather it has only accuracy up to the 3rd decimal place since 13 of those digits come before the decimal place. Therefore all of the values in the resulting vector x only have accuracy up to the 3rd decimal place which means calculating the variance using x is only accurate up to the 3rd decimal place.

To summarize: the variances should be the same but changing the scale of the numbers by adding 1×10^12 results in increased absolute error in calculation so calculating the variance using z is more accurate (to about the 15th-16th decimal place) while calculating the variance using x is less accurate (to about the 3rd decimal place).

Here is a demonstration that the numbers are only accurate to the 3rd decimal place after adding 1×10^12:

```
z = rng.normal(size = 10)
x = z + 1e12 - 1e12
_ = [dg(j) for j in z]
print("") # Just adding space
_ = [dg(j) for j in x]
```

```
-0.65128101244339398068
0.86244479631574677558
-0.12559208403432720047
0.66915324078945281894
1.21884360517122325440
```

```
0.38292958271347238286
-0.87572114342284546051
-1.51431863170463842927
1.75338411751637268665
-0.11129219318751944201

-0.651245117187500000000
0.86242675781250000000
-0.12561035156250000000
0.66918945312500000000
1.21887207031250000000
0.38293457031250000000
-0.87573242187500000000
-1.51428222656250000000
1.75341796875000000000
-0.11132812500000000000
```

We can see that while some are accurate up to the 4th decimal place or so these are lucky happenings and we can only truly guarantee accuracy up to the 3rd.

3.

The approximate expression to be calculated is given by:

$$f(y^*|y,x) \approx \frac{1}{m}\sum_{j=1}^{m}\prod_{i=1}^{n} f(y_i^*|y,x,\theta_j)$$

Where $\theta_j \sim \pi(\theta|y,x)$ for $j \in \{1,...,m\}$ is the posterior distribution of each θ_j.

a.

While densities don't necessarily need to be small in magnitude we are often working with data where they are (for example if we are working with normal data $f(x) = \frac{1}{\sigma\sqrt{2\pi}}e^{-\frac{(x-\mu)^2}{2\sigma^2}} \leq \frac{1}{\sigma\sqrt{2\pi}}$ can be relatively small in magnitude) so when we multiply a large number of small numbers we might get underflow so it is advised against directly using the product to find this approximation. However, if we do it on the log scale we will get only the exponents of the numbers and sum them which will be fine to do since if we are worried about underflow that means the exponents are negative numbers that aren't extremely small so summing over all of the log values (the exponents) of the density will not result in underflow.

b.

Now we are computing:

$$\frac{1}{m}\sum_{j=1}^{m}\exp\sum_{i=1}^{n}\log f(y_i^*|y,x,\theta_j)$$

Again if we were initially worried about underflow then the values from the sum of the logs will be very large negative numbers (i.e. far from 0 and negative). So when we try to take the exponent we will likely get underflow again. More specifically when we try to perform the calculation $\exp(v_j)$ we will get underflow because v_j is supposedly a very large negative number so $\exp(v_j)$ should be extremely close to 0 (so close that we can't represent it accurately on a computer and it will just get assigned the value 0). This is bad because then our final approximation will just be a sum of 0s which is just 0 and clearly not what we are looking for.

c.

Now we are computing:

$$\log f(y^*|y,x) \approx \log\Big(\frac{1}{m}\sum_{j=1}^{m}\exp(v_j)\Big)$$

Again there will likely be an issue with underflow when we try to compute this and we will get log(0) which does not exist so to be able to perform this calculation we first need to rescale all of the exponents. Let $z = \min\{|v_1|, ..., |v_m|\}$ (i.e. the $|v_j|$ closest to 0). Then we can rewrite this expression as:

$$\log f(y^*|y,x) \approx \log\Big((\exp(-z))\frac{1}{m}\sum_{j=1}^{m}\exp(z - v_j)\Big)$$

Now since all of these data come from the same distribution we expect each $z - v_j$ to be much much closer to 0 rather than very negative so all of those exponents should work fine and we won't get underflow. Now quickly note that we chose z to be the min because this makes it much less likely that we get underflow when computing $\exp(-z)$ at the end. So this equivalent expression should be much safer to use when computing your approximation.

4.

If we have a random variable Z with density $f_Z(z)$ and CDF $F_Z(z)$ we can find the truncated distribution when we restrict it to $z < a$ for some fixed $a \in \mathbb{R}$. We know that the truncated density should be such that $f_{Z|Z<a}(z) \propto f_Z(z)\, I(z < a)$, therefore:

$$\int_{-\infty}^{\infty} f_{Z|Z<a}(z)\, dz \propto \int_{-\infty}^{\infty} f_Z(z)\, I(z < a)\, dz = \int_{-\infty}^{a} f_Z(z)\, dz = F(a)$$

So to normalize the truncated density we should divide by the CDF at a, namely $f_{Z|Z<a}(z) = \frac{f_Z(z)\, I(z<a)}{F(a)}$, to verify this is a density:

$$\int_{-\infty}^{\infty} f_{Z|Z<a}(z)\, dz = \int_{-\infty}^{\infty} \frac{f_Z(z)\, I(z < a)}{F(a)}\, dz = \frac{1}{F(a)} \int_{-\infty}^{a} f_Z(z)\, dz = \frac{F(a)}{F(a)} = 1$$

Now we can apply this to our specific problem.

a.

We can use packages in python to get the density and CDF of the t-distribution and normal distribution. Then here we are truncating at 4 for each.

To actually get the point we want to evaluate the densities at we can take a uniform random variable where the lower bound is 0 and the upper bound is whatever the CDF of a normal with variance 1 centered at -4 is at -4 then pass that through the inverse CDF of the normal, this will give us our desired value according to the truncated normal distribution and we can plug that in to each density.

Then recall that we want to find $\phi = \mathbb{E}_f[X] = \mathbb{E}_f[h(X)]$ so here $h(x) = x$.

```
def truncated_t(x):
  f = stats.t.pdf(df = 3, x = x)
  f = f/stats.t.cdf(df = 3, x = -4)
  return(f)

def truncated_norm(x):
  f = stats.norm.pdf(loc = -4, x = x)
  # The normal CDF part will just be 1/2
  # by symmetry so we would divide by 1/2 or
  # equivalently multiply by 2
  f = 2*f
  return(f)

def get_pt(trunc = -4):
  p = stats.norm.cdf(loc = -4, x = trunc)
  u = np.random.uniform(low = 0, high = p)
  z = stats.norm.ppf(loc = -4, q = u)
  return(z)

data = {"Point": [], "Ratio": [], "h Times Ratio": []}

m = 10000
for j in range(m):
  x = get_pt()
```

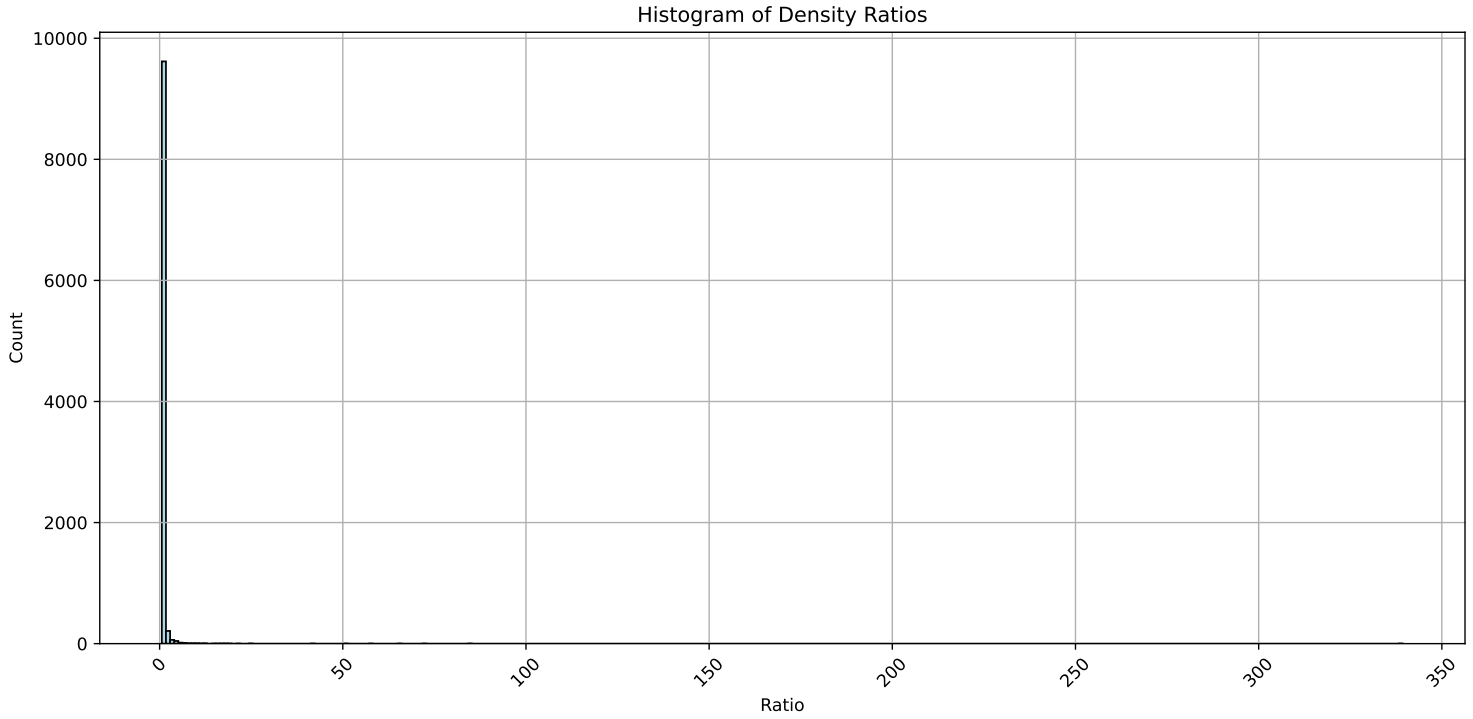
```
  ratio = truncated_t(x)/truncated_norm(x)
  data["Point"] += [x]
  data["Ratio"] += [ratio]
  data["h Times Ratio"] += [x*ratio]

np.mean(data["h Times Ratio"])
```

```
-4.804881608966896
```

This is just a quick sanity check that the expectation of our truncated distribution is less than -4 (and it is as it should be).

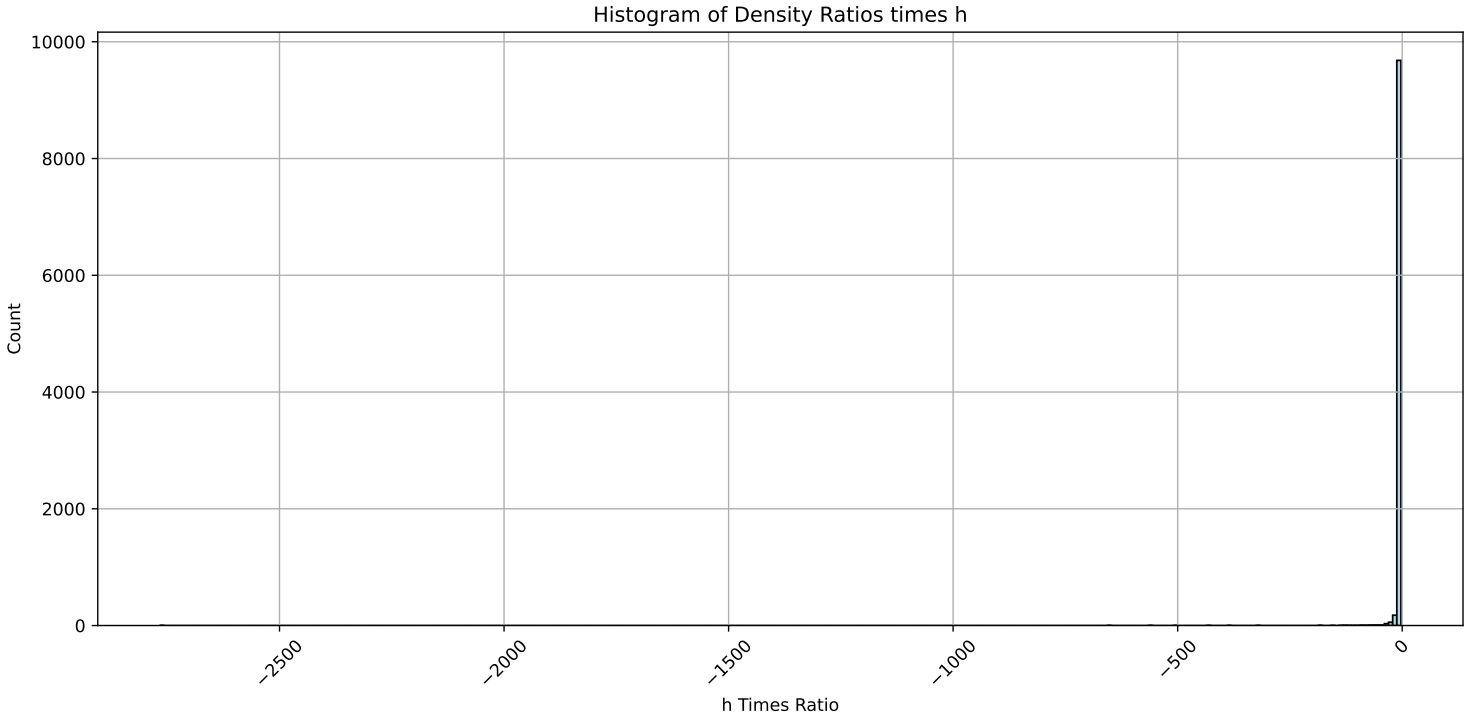
Now plotting:

```
# Graph results
# I assign some of these to a dummy variable because it was causing output I didn't want
_ = plt.figure(figsize=(12, 6))
_ = plt.hist(data["Ratio"],
             bins = 300,
             color = 'lightblue',
             edgecolor = 'black')
_ = plt.title("Histogram of Density Ratios")
_ = plt.xlabel("Ratio")
_ = plt.ylabel("Count")
plt.ticklabel_format(style = "plain", axis = "y")
_ = plt.xticks(rotation=45)
plt.grid()
plt.tight_layout()
plt.show()
```



This tells us that the truncated density of the t-distribution is usually much smaller than that of the truncated normal except for a small number of cases where the normal is much smaller. This makes sense because the truncated t-distribution is still supposed to be centered at 0 while the truncated normal is centered at -4 so in general the t-distribution's density will be smaller however the normal distribution decreases much quicker than the t-distribution does because it is an exponential versus a power so in a small number of extreme cases the normal density will be much less.

```
# Graph results
# I assign some of these to a dummy variable because it was causing output I didn't want
_ = plt.figure(figsize=(12, 6))
_ = plt.hist(data["h Times Ratio"],
             bins = 300,
             color = 'lightblue',
             edgecolor = 'black')
_ = plt.title("Histogram of Density Ratios times h")
_ = plt.xlabel("h Times Ratio")
_ = plt.ylabel("Count")
plt.ticklabel_format(style = "plain", axis = "y")
_ = plt.xticks(rotation=45)
plt.grid()
plt.tight_layout()
plt.show()
```



There are clealy some extreme weights that would impact $\hat{\phi}$ by looking at both histograms (however the impact won't be large because in the end we will still divide by $m = 10000$ which is much larger than any of these extreme points). This is because of the result I said before that in extreme cases the normal density will be much much smaller than the t-distribution density which will lead to a very large ratio as we are seeing. From each of these graphs it doesn't seem like the variance of $\hat{\phi} = \frac{1}{m}\sum_{i=1}^{m} \frac{x_i \times f(x_i)}{g(x_i)}$ is particularly large, everything is rather close together. Now we will estimate it.

The variance of $\hat{\phi}$ is given by:

$$\mathbb{V}_g[\hat{\phi}] = \mathbb{V}_g\Big[\frac{1}{m}\sum_{i=1}^{m}\frac{X_i \times f(X_i)}{g(X_i)}\Big] = \frac{1}{m^2}\mathbb{V}_g\Big[\sum_{i=1}^{m}\frac{X_i \times f(X_i)}{g(X_i)}\Big] = \frac{1}{m^2}\sum_{i=1}^{m}\mathbb{V}_g\Big[\frac{X_i \times f(X_i)}{g(X_i)}\Big]$$

$$= \frac{1}{m}\mathbb{V}_g\Big[\frac{X \times f(X)}{g(X)}\Big] \approx \frac{1}{m}\left(\frac{1}{m}\sum_{i=1}^{m}\Big(\frac{x_i\, f(x_i)}{g(x_i)} - (\frac{1}{m}\sum_{i=1}^{m}\frac{x_i\, f(x_i)}{g(x_i)})\Big)^2\right)$$

```
var_hat_h = np.mean((data["h Times Ratio"] - np.mean(data["h Times Ratio"]))**2)
var_phi_hat = var_hat_h/m
var_phi_hat
```

```
0.09446070469289973
```

The estimated variance is rather small. This is because $\hat{\phi}$ is a consistent estimator and m is very large.

b.

Now we do the same process again but using a t_1 distribution instead of a normal distribution.

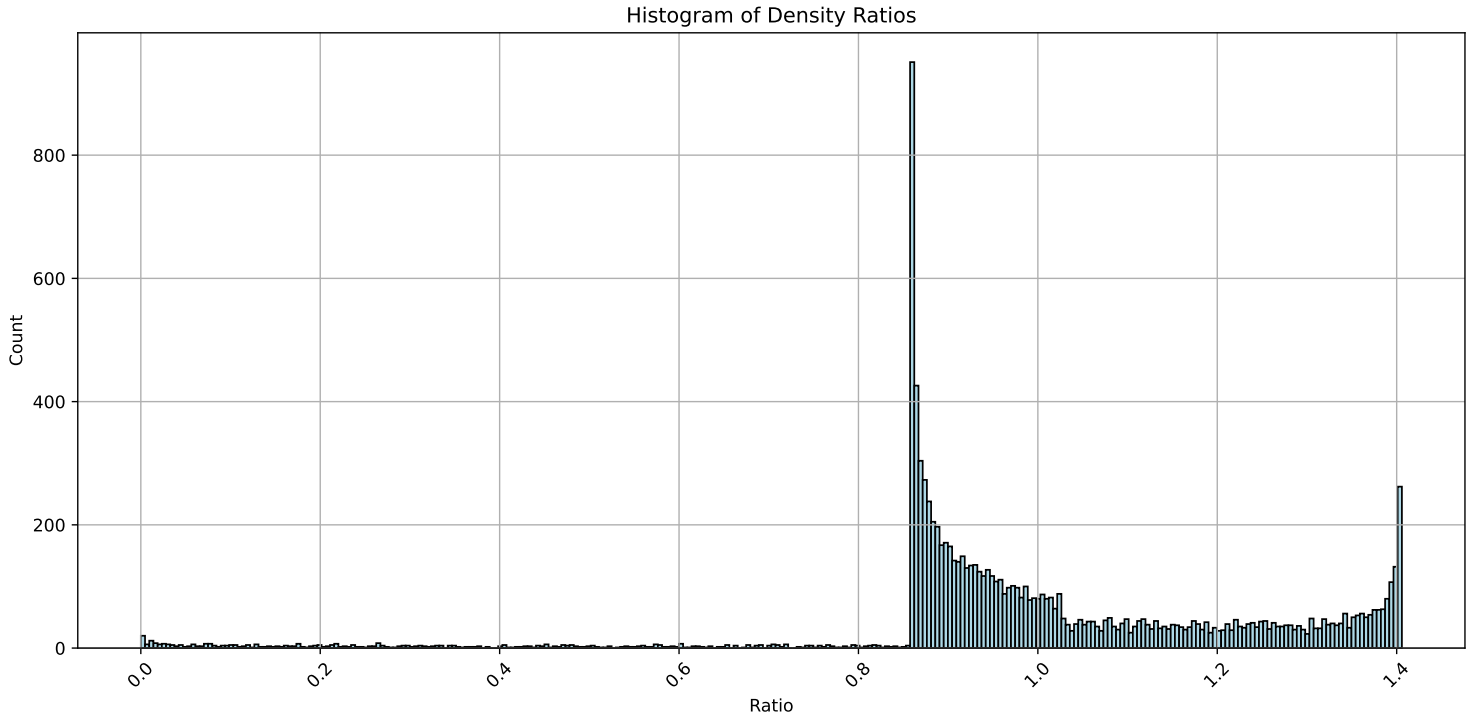
```
def truncated_t_3(x):
  f = stats.t.pdf(df = 3, x = x)
  f = f/stats.t.cdf(df = 3, x = -4)
  return(f)

def truncated_t_1(x):
  f = stats.t.pdf(loc = -4, df = 1, x = x)
  # The t CDF part will just be 1/2
  # by symmetry so we would divide by 1/2 or
  # equivalently multiply by 2
  f = 2*f
  return(f)

def get_pt(trunc = -4):
  p = stats.t.cdf(loc = -4, df = 1, x = trunc)
  u = np.random.uniform(low = 0, high = p)
  z = stats.t.ppf(loc = -4, df = 1, q = u)
  return(z)

data = {"Point": [], "Ratio": [], "h Times Ratio": []}

m = 10000
for j in range(m):
  x = get_pt()
  ratio = truncated_t_3(x)/truncated_t_1(x)
  data["Point"] += [x]
  data["Ratio"] += [ratio]
  data["h Times Ratio"] += [x*ratio]

np.mean(data["h Times Ratio"])
```

```
-6.265867878768089
```

We see quite a noticeable difference this time in $\hat{\phi}$ but still it is less than -4 and not too far away from the previous estimate so we can continue.

Now plotting:

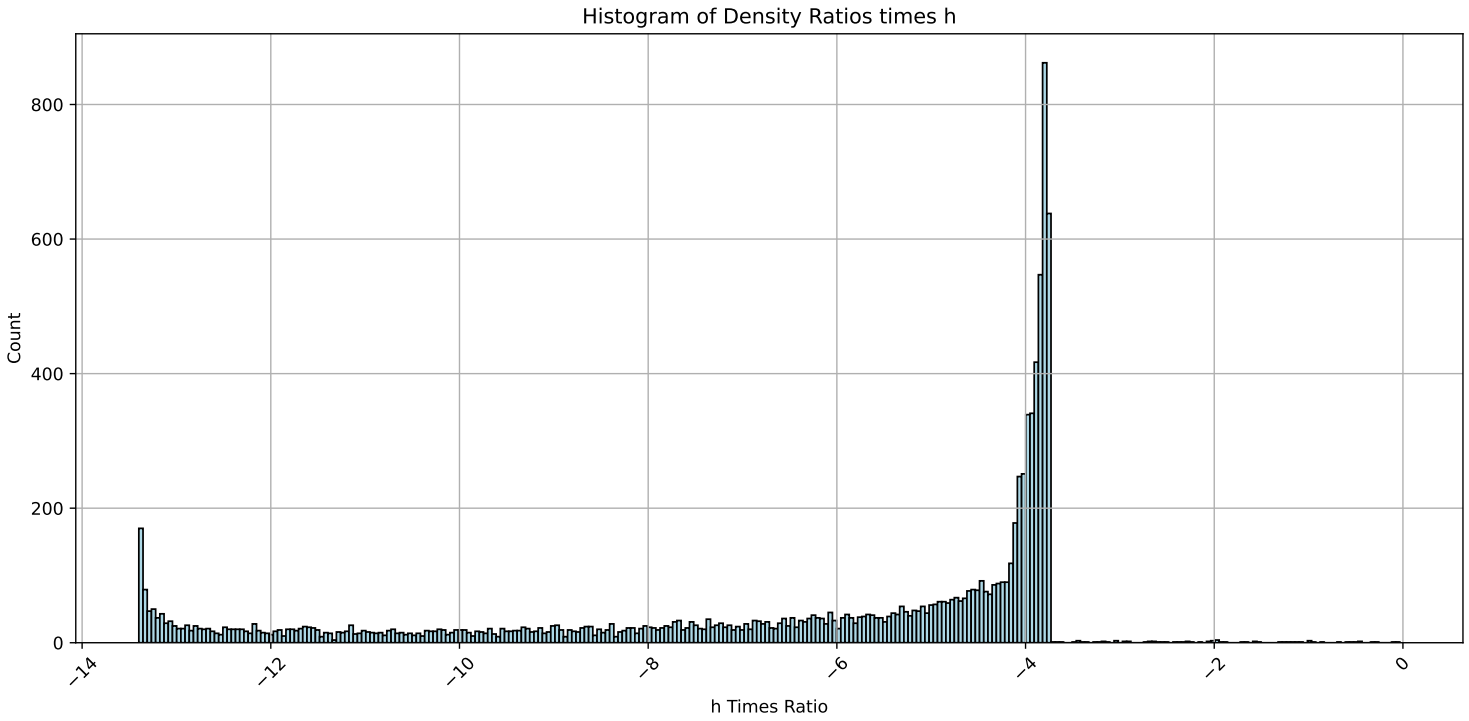
```
# Graph results
# I assign some of these to a dummy variable because it was causing output I didn't want
_ = plt.figure(figsize=(12, 6))
_ = plt.hist(data["Ratio"],
             bins = 300,
             color = 'lightblue',
             edgecolor = 'black')
_ = plt.title("Histogram of Density Ratios")
_ = plt.xlabel("Ratio")
_ = plt.ylabel("Count")
plt.ticklabel_format(style = "plain", axis = "y")
_ = plt.xticks(rotation=45)
plt.grid()
plt.tight_layout()
plt.show()
```



As we can see this is a much different story than before. There are no extreme cases because the densities are both power functions of similar power so their ratios are very close. Now we see that normally the t-distribution centered at 0 with 3 degrees of freedom normally has a density greater than that of the other. This is because the power of the density for the t-distribution with 3 degrees of freedom is greater and so the density will always be larger with the exception of points that are in a relatively small range (similar to how when $k > 0$ we know $x^{n+k} > x^n$ for all $x > 1$ and $x^{n+k} < x^n$ for all $0 < x < 1$).

```
# Graph results
# I assign some of these to a dummy variable because it was causing output I didn't want
_ = plt.figure(figsize=(12, 6))
_ = plt.hist(data["h Times Ratio"],
             bins = 300,
```

```
            color = 'lightblue',
            edgecolor = 'black')
_ = plt.title("Histogram of Density Ratios times h")
_ = plt.xlabel("h Times Ratio")
_ = plt.ylabel("Count")
plt.ticklabel_format(style = "plain", axis = "y")
_ = plt.xticks(rotation=45)
plt.grid()
plt.tight_layout()
plt.show()
```



Now there clealy aren't any extreme weights that would impact $\hat{\phi}$ by looking at both histograms. Again this is because of the result I said before that both densities are now power functions with similar powers. Initially looking at each of these graphs it seems like the variance of $\hat{\phi} = \frac{1}{m}\sum_{i=1}^{m} \frac{x_i \times f(x_i)}{g(x_i)}$ is larger than before but that would not be taking into account that before we had extreme cases so in fact now the variance should be less than before due to the lack of extreme cases. Now we will estimate it using the same formula as before just with our new values:

```
var_hat_h = np.mean((data["h Times Ratio"] - np.mean(data["h Times Ratio"]))**2)
var_phi_hat = var_hat_h/m
var_phi_hat
```

```
0.0009406758182550721
```

This is consistent with what we said before, the variance has decreased. This is again due to the fact that $\hat{\phi}$ is a consistent estimator and m is very large. Now we will continue by finding a simulation interval.

We can find a 95% simulation interval for $\hat{\phi}$ via the following:

First note that $\hat{\phi} = \frac{1}{m}\sum_{i=1}^{m} \frac{X_i \times f(X_i)}{g(X_i)}$ is a sum of iid random variables and so by the central limit theorem we know it is approximately normal for large m.

Then notice:

$$\mathbb{E}_g[\hat{\phi}] = \mathbb{E}_g\Big[\frac{1}{m}\sum_{i=1}^{m}\frac{X_i \times f(X_i)}{g(X_i)}\Big] = \frac{1}{m}\sum_{i=1}^{m}\mathbb{E}_g\Big[\frac{X_i \times f(X_i)}{g(X_i)}\Big] = \mathbb{E}_g\Big[\frac{X \times f(X)}{g(X)}\Big]$$

$$= \int_{-\infty}^{\infty}\frac{x\, f(x)}{g(x)}g(x)\, dx = \int_{-\infty}^{\infty} x\, f(x)\, dx = \mathbb{E}_f[X] = \phi$$

Therefore we can use the standard 95% normal confidence interval for an unbiased estimator. First we know:

$$0.95 = \mathbb{P}\Big[\phi \in (\hat{\phi} - z_{0.025}\sqrt{\mathbb{V}[\hat{\phi}]}, \hat{\phi} + z_{0.025}\sqrt{\mathbb{V}[\hat{\phi}]})\Big] \approx \mathbb{P}\Big[\phi \in (\hat{\phi} - 1.96\sqrt{\hat{\mathbb{V}}[\hat{\phi}]}, \hat{\phi} + 1.96\sqrt{\hat{\mathbb{V}}[\hat{\phi}]})\Big]$$

So the simulation interval is $\left(\hat{\phi} - 1.96\sqrt{\hat{\mathbb{V}}[\hat{\phi}]}, \hat{\phi} + 1.96\sqrt{\hat{\mathbb{V}}[\hat{\phi}]}\right)$ which is calculated below.

```
phi_hat = np.mean(data["h Times Ratio"])
[phi_hat - 1.96*(var_phi_hat**0.5), phi_hat + 1.96*(var_phi_hat**0.5)]
```

```
[-6.325981938882247, -6.20575381865393]
```