AppliedW8

April 18, 2024

1 Question 1

Two possible algorithms:

Algorithm 1:

Find a box which bounds the function of interest. The x co-ordinates should be the same as the domain we want to integrate over. Zero can be taken as the lower bound for the y values. To find an upper bound, take a collection of x values between a and b and evaluate the function f at these points to find the largest value. As it is unlikely you found the true maximum, you may want to multiplicatively increase this amount to help ensure the true maximum of the function is bounded.

Next, randomly generate points, (x_i, y_i), inside this box. Count how many randomly generated points satisfy the condition $y_i < f(x_i)$. The area under the curve $f(x)$ between the x co-ordinates a and b can be approximated by the ratio of points satisfying the aforementioned conditioned to total number of randomly generated points multiplied by the area of the bounding box.

Algorithm 2:

This method utilises the Mean Value Theorem for Integrals. This states that the mean value for a function over the domain $[a, b]$, can be given by:

$$m = \frac{1}{b-a}\int_a^b f(x)dx$$

Rearranging this expression gives:

$$\int_a^b f(x)dx = m(b-a)$$

If we can estimate the mean, m, then we can estimate the integral. This can be done by randomly sampling points between a and b then taking the mean of function f evaluated at these points.

[1]:
```
import numpy as np
from matplotlib import pyplot as plt
```

1.1 Monte Carlo Integration

[2]:
```
def integrate(f, a, b, max_iter=1000):
    x = (b-a)*np.random.random(max_iter) + a
    fx = f(x)
    m = fx.mean()
    area = m*(b-a)
    return area
```

1.2 $\int_0^\pi sin(x) = 2$

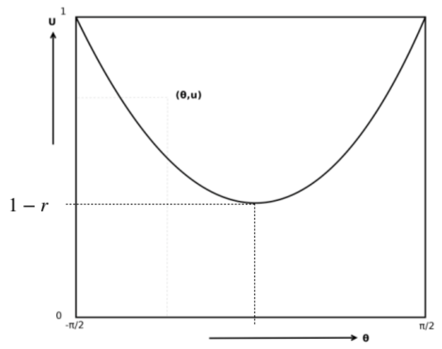
[3]:
```
f = lambda x: np.sin(x)
a=0
b=np.pi
for i in range(7):
    integral = integrate(f=f,a=a,b=b,max_iter=10**i)
    print('max_iter: {0}, integral:{1}, relative error: {2}'.format(10**i,␣
 ↪integral, np.abs((2-integral)/2)))
```

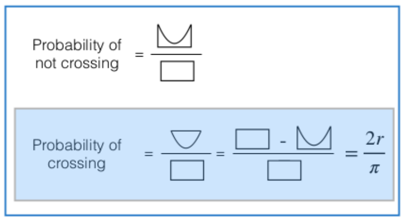
```
max_iter: 1, integral:0.3044102660547283, relative error: 0.8477948669726358
max_iter: 10, integral:2.4066146207634453, relative error: 0.20330731038172267
max_iter: 100, integral:1.9548523465190721, relative error: 0.022573826740463933
max_iter: 1000, integral:1.9481646533053811, relative error:
0.025917673347309433
max_iter: 10000, integral:2.0035546063745837, relative error:
0.0017773031872918565
max_iter: 100000, integral:1.9965879111811262, relative error:
0.001706044409436891
max_iter: 1000000, integral:1.9995262551647406, relative error:
0.00023687241762970768
```

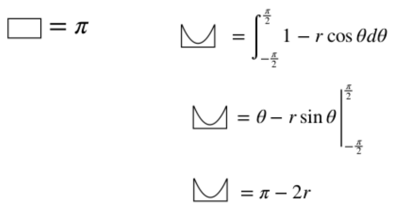
2 Question 2

Consider the case where $r < 1$ from the lecture notes shown below. Note that the distance the minimum of the curve is away from the x-axis is $1 - r$. In the case where $r > 1$, this minimum point extends past the x-axis and is more accurately displayed in the image below. The blue curve is still the curve $u = 1 - r\cos(\theta)$ and we're still interested in the case where $u > 1 - r\cos(\theta)$ as this is when the needle crosses a line but also when $u \geq 0$. When $u < 0$, the needle is crossing over two lines and u should be truncated to be zero as it is zero distance away from the line. This corresponds to Region 3 (R3) shown in green. To find the probability a needle crosses a line, we take the same ratio as the $r < 1$ case: the region above the curve (R3) divided by the total region which is still π. Instead of trying to calculate the size of R3, we will instead calculate the complement, the sum of R1 and R2 and subtract this from the total. Noting that the size of R1 and R2 are the same, we just need to find the size of R1 and subtract that twice from the total size.

As like before, we can do this using integration but first we need to find where our function intersects the x-axis. This will give us the upper bound for our integral, the lower bound is clearly




Probability of not crossing =
Probability of crossing = = - = 2r/π



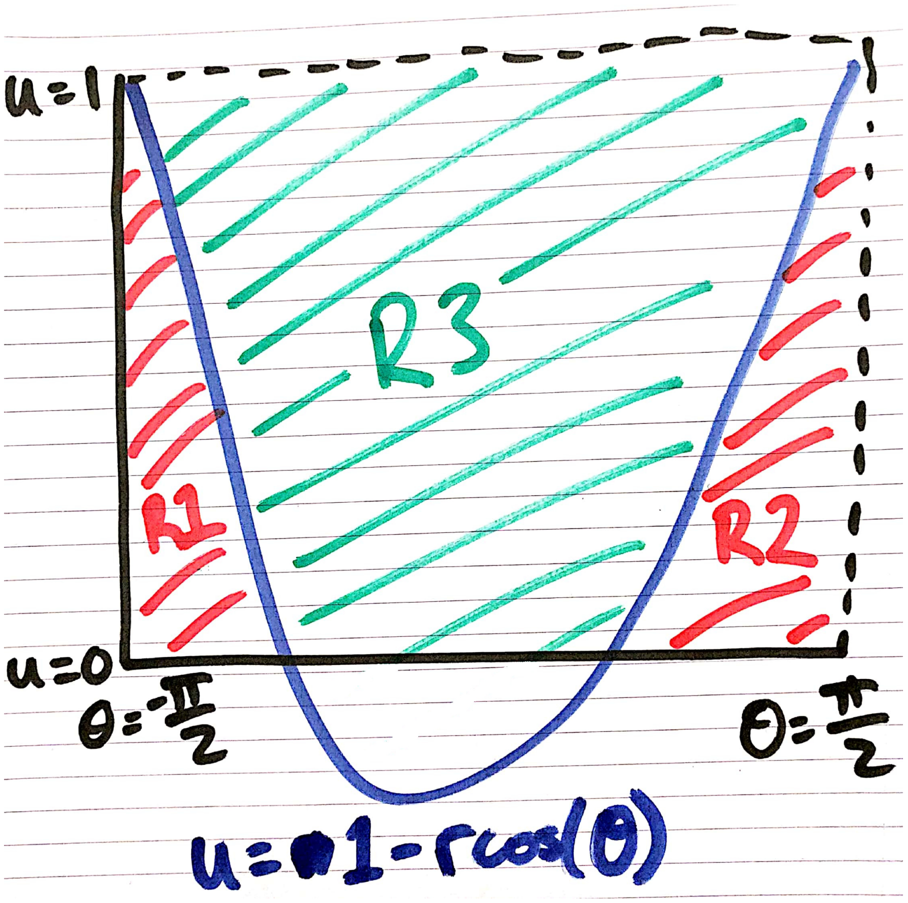
$\theta = -\pi/2$. Setting u equal to zero and solving for θ gives:

$$\begin{aligned}
1 - r\cos(\theta) &= 0 \\
\implies r\cos(\theta) &= 1 \\
\implies \cos(\theta) &= \frac{1}{r} \\
\implies \theta &= \cos^{-1}\left(\frac{1}{r}\right)
\end{aligned}$$

Since $1/r > 0$, this gives the value for the positive root, but due to symmetry, the negative root can be given by $-\cos^{-1}(1/r)$. Integrating $u = 1 - r\cos(\theta)$ over the bounds $-\pi/2 \leq \theta \leq -\cos^{-1}(1/r)$ we get:

$$\begin{aligned}
\int_{-\frac{\pi}{2}}^{-\cos^{-1}\left(\frac{1}{r}\right)} 1 - r\cos(\theta)d\theta &= [\theta - r\sin(\theta)]_{-\frac{\pi}{2}}^{-\cos^{-1}\left(\frac{1}{r}\right)} \\
&= \left[-\cos^{-1}\left(\frac{1}{r}\right) - r\sin\left(-\cos^{-1}\left(\frac{1}{r}\right)\right)\right] - \left[-\frac{\pi}{2} - r\sin\left(-\frac{\pi}{2}\right)\right] \\
&= -\cos^{-1}\left(\frac{1}{r}\right) + r\sin\left(\cos^{-1}\left(\frac{1}{r}\right)\right) + \frac{\pi}{2} - r \\
&= -\cos^{-1}\left(\frac{1}{r}\right) + r\sqrt{1 - \frac{1}{r^2}} + \frac{\pi}{2} - r \\
&= -\cos^{-1}\left(\frac{1}{r}\right) + \sqrt{r^2 - 1} + \frac{\pi}{2} - r
\end{aligned}$$

From line 3 to line 4 we have used the identity: $\sin(\cos^{-1}(x)) = \sqrt{1 - x^2}$. Doubling this and



subtracting it from the total area, π, we get an area of:

$$\begin{aligned}\text{R3}_{\text{Area}} &= \pi - 2\left(-\cos^{-1}\left(\frac{1}{r}\right) + \sqrt{r^2-1} + \frac{\pi}{2} - r\right)\\ &= 2\left(r + \cos^{-1}\left(\frac{1}{r}\right) - \sqrt{r^2-1}\right)\end{aligned}$$

Finally, dividing by the total area gives the probability of a needle of length $r > 1$ crossing a line.

$$\text{Probability of Crossing} = \frac{2}{\pi}\left(r + \cos^{-1}\left(\frac{1}{r}\right) - \sqrt{r^2-1}\right)$$

2.1 Buffon

[4]:
```
f = lambda r, theta: 1 - r*np.cos(theta)
r=0.5
def solve_buffon(f, r, max_iter=1000):
    theta = (np.pi)*np.random.random(max_iter) - np.pi/2
    u = np.random.random(max_iter)
    under = u > 1-r*np.cos(theta)
    prob = sum(under)/max_iter
    return prob
```

```
[5]: solve_buffon(f=f, r=r, max_iter=1000)
```

```
[5]: 0.291
```

2.2 r<1

```
[6]: 2*r/np.pi
```

```
[6]: 0.3183098861837907
```

2.3 r>1

```
[7]: (2/np.pi)*(r + np.arccos(1/r) - np.sqrt(r**2-1))
```

```
/tmp/ipykernel_332607/3816886798.py:1: RuntimeWarning: invalid value encountered
in arccos
  (2/np.pi)*(r + np.arccos(1/r) - np.sqrt(r**2-1))
/tmp/ipykernel_332607/3816886798.py:1: RuntimeWarning: invalid value encountered
in sqrt
  (2/np.pi)*(r + np.arccos(1/r) - np.sqrt(r**2-1))
```

```
[7]: nan
```

3 Question 3

Inverse sampling: Let $F(x)$ be the distribution function for a random variable, X. To obtain a sample, x, from X, obtain a sample, u, from a continuous uniform variable between 0 and 1 then x can be calculated by $x = F^{-1}(u)$.

We know that the distribution for a function is bounded between 0 and 1 and must be one-to-one as it is strictly increasing. We can uniquely map a randomly generated value between 0 and 1 to a sample by using the inverse of the cumulative function. The cumulative function grows most rapidly for the x values with the largest density, so the inverse will grow most slowly for the values between 0 and 1 that map to x values with largest density. Since these sections grow most slowly there is a greater amount of values between 0 and 1 mapping to the denser parts of the density, as we should expect.

We know that the integral of this function over the domain $0 \leq x \leq 3$ has to be one. We can use this to find the value of a.

$$\begin{aligned} \int_0^3 \frac{a}{x+1} dx &= 1 \\ a\left[\log(x+1)\right]_0^3 &= 1 \\ a\left[\log(4) - \log(1)\right] &= 1 \\ a\left[2\log(2)\right] &= 1 \\ \implies a &= \frac{1}{2\log(2)} \end{aligned}$$

To sample this distribution using inverse sampling, we need to find its CDF, $F(x)$. Recall:

$$F(x) = \int_{\infty}^{x} f(\bar{x}) d\bar{x}$$

Evaluating this integral gives (leaving a for simplicity):

$$\begin{aligned} F(x) &= \int_{\infty}^{x} \frac{a}{\bar{x}+1} d\bar{x} \\ &= a \int_{0}^{x} \frac{1}{\bar{x}+1} d\bar{x} \\ &= a \left[\log(\bar{x}+1)\right]_0^x \\ &= a \left[\log(x+1) - \log(1)\right] \\ F(x) &= a \log(x+1) \end{aligned}$$

From line one to line two, we use the fact that $f(x) = 0$ outside of the domain $0 \leq x \leq 3$. Inverting the CDF, we get:

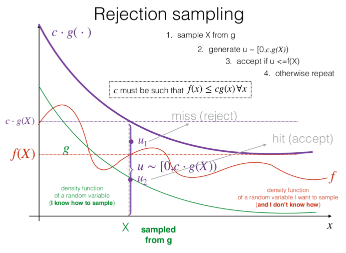
$$F^{-1}(x) = e^{\frac{x}{a}} - 1$$

To obtain a sample, x from our random variable, X, we simply sample u from a continuous uniform random variable over $0 \leq u \leq 1$ and evaluate $x = F^{-1}(u)$.

4 Question 4

Rejection sampling: Let $f(x)$ be the density function for the random variable, X, that we want to sample. Find a constant c and a random variable we know how to sample $g(x)$ such that $f(y) < c \cdot g(y)$, $\forall y$. Next, we sample a value u from a continuous uniform variable between 0 and 1 and a value x from the random variable with density $g(x)$. Then, we accept x as a sample from X if $f(x) \leq u \cdot c \cdot g(x)$. Repeat until we accept as many samples as required.

Intuitively, we can think of this as a Monte Carlo method. Observing the plot below, we want to generate points with an x component from the possible values our random variable X can take and y component between 0 and $c \cdot g(x)$, but only accept them if they're below . We can generate the x co-ordinate by sampling our known distribution and then use this to get a y co-ordinate value between 0 and $c \cdot g(x)$. If we generate a random number between 0 and 1 and multiple it by some number a, then this is equivalent to generating a random number between 0 and a. This is how we generate the y co-ordinate in this case - we obtain a random number u between 0 and 1 and multiply it by $c \cdot g(x)$. We only want to accept samples that fall below $f(x)$, that is, when $f(x) \leq u \cdot c \cdot g(x)$.


Rejection sampling
$c \cdot g(\cdot)$
1. sample X from g
2. generate u ~ [0,$c.g(X)$)
3. accept if u <=f(X)
4. otherwise repeat
c must be such that $f(x) \leq cg(x) \forall x$
$c \cdot g(X)$
miss (reject)
hit (accept)
$f(X)$
g
u_1
$u \sim [0, c \cdot g(X))$
u_2
f
density function of a random variable (I know how to sample)
density function of a random variable I want to sample (and I don't know how)
X
sampled from g
x

We can use rejection sampling.

We first choose the density function of a random variable we know how to sample on the same domain (i.e., $0 < x < 1$). We can generate a uniform random variable, so $g(x) = 1$ is suitable.

We find

$$c = \max \frac{f(x)}{g(x)} = \max(140x^3(1-x)^3)$$

.

$$f'(x) = 420(1-x)^3x^2 - 420(1-x)^2x^3 = 0$$

Factorising, we get

$$f'(x) = -420(x-1)^2x^2(2x-1) = 0$$

It is easy to verify the maximum occurs at $x = \frac{1}{2}$, with value $35/16$. Therefore:

$$\frac{f(x)}{cg(x)} = 64x^3(1-x)^3 = T(x)$$

To sample from the distribution, we generate a random numbers $u_1 = rand()$, which comes from $g(x)$; and a uniformly random number $u_2 = rand()$.

If $u_2 \leq T(u_1)$, then u_1 is your sample. Otherwise generate u_1 and u_2 again, and repeat the process.

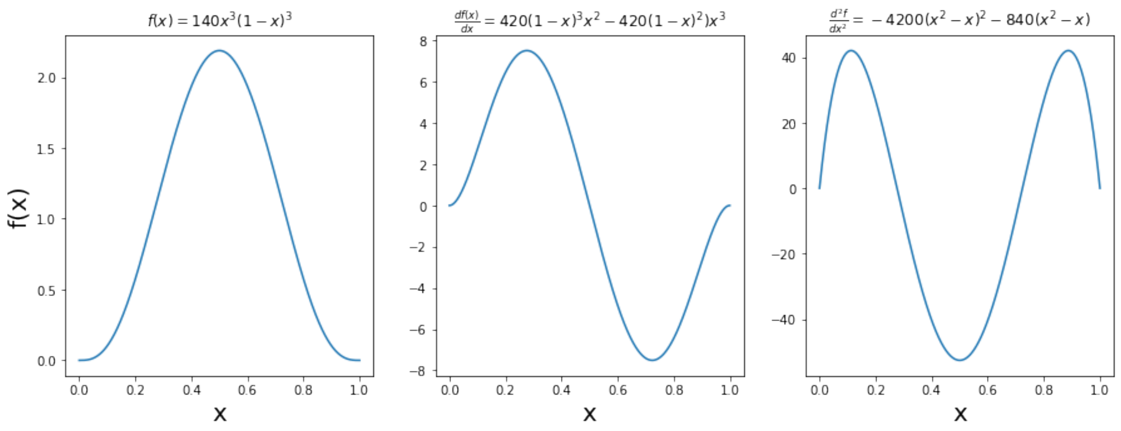
4.1 Rejection Sampling

[8]:
```
def newton(f, df, x, max_iter=1000, tol=1e-10, verbose=False):
    i = 0
    while (np.abs(f(x)) > tol) and (i<max_iter):
        if verbose: print('i: {0}, x: {1}, f(x): {2}'.format(i, x, f(x)))
        x = x - f(x)/df(x)
        i+=1
    return x
```

[86]:
```
f = lambda x: 140*(x**3)*((1-x)**3)
df = lambda x: 420*((1 - x)**3)*(x**2) - 420*((1 - x)**2)*x**3
d2f = lambda x: -4200*(x**2-x)**2 -840*(x**2-x)
xmin=0
xmax=1
x0=0.6

names = [r'$f(x)=140x^3(1-x)^3$',
         r'$\frac{df(x)}{dx}=420(1 - x)^3x^2 - 420(1 - x)^2)x^3$',
         r'$\frac{d^2f}{dx^2}=-4200(x^2-x)^2 -840(x^2-x)$']
X = np.linspace(xmin, xmax, 10000)
fig, axs = plt.subplots(1, 3, sharex=True, sharey=False)
fig.set_figwidth(15)
fig.set_figheight(5)
for i, fn in enumerate([f, df, d2f]):
    axs[i].plot(X, fn(X))
    axs[i].set_title(names[i])
    axs[i].set_xlabel('x', size=20)
axs[0].set_ylabel('f(x)', size=20)
# plt.legend()
```

[86]:
```
Text(0, 0.5, 'f(x)')
```



[10]:
```
x_local_max = newton(f=df, df=d2f, x=x0, verbose=True)
```

```
i: 0, x: 0.6, f(x): -4.8384
i: 1, x: 0.48, f(x): 1.0466426880000022
i: 2, x: 0.5001290322580645, f(x): -0.006774192646100019
i: 3, x: 0.49999999996562716, f(x): 1.804572491437284e-09
```

[11]:
```
x_local_max
```

[11]:
```
0.49999999999999994
```

[12]:
```
d2f(x_local_max)
```

[12]:
```
-52.5
```

[13]:
```
df(x_local_max)
```

[13]:
```
0.0
```

[14]:
```
f(x_local_max)
```

[14]:
```
2.1874999999999999
```

[111]:
```
class rejection_sampler:# assumes finite domain
    def __init__(self, f, df, d2f, a, b):
        self.f=f
        self.df=df
        self.d2f=d2f
        self.a=a
        self.b=b
        self.c = self.make_c()

    def get_max(self, max_iter=1000, verbose=False):
        xmax = self.a if self.f(self.a) > self.f(self.b) else self.b
        for i in range(max_iter):
            x = (self.b-self.a)*np.random.random()+self.a
            x_star = newton(f=self.df, df=self.d2f, x=x, max_iter=1000,␣
↪tol=1e-10, verbose=False)
            if x_star >= self.a and x_star <= self.b:
                fx = self.f(x_star)
                xmax = xmax if self.f(xmax) > self.f(x_star) else x_star
        return xmax

    def make_c(self, g=lambda x: 1, max_iter=10): # assumes uniform bounding␣
↪function
        xmax = self.get_max(max_iter=max_iter)
        self.c = f(xmax)
        self.xmax = xmax
        return self.c

    def sample_single(self, verbose=False):
        u1 = np.random.random()
        u2 = (self.b - self.a)*np.random.random() + self.a
        if self.f(u2)/self.c > u1:
            if verbose: print('x={0}, accepted'.format(u2))
            return u2
        else:
            if verbose: print('x={0}, rejected'.format(u2))
            return self.sample_single(verbose=verbose)

    def sample(self, n, verbose=False):
        samples = np.zeros(n)
        for i in range(n):
            samples[i] = self.sample_single(verbose=verbose)
        return samples
```

[112]:
```
sampler = rejection_sampler(f=f, df=df, d2f=d2f, a=0, b=1)
sampler.c, sampler.xmax
```

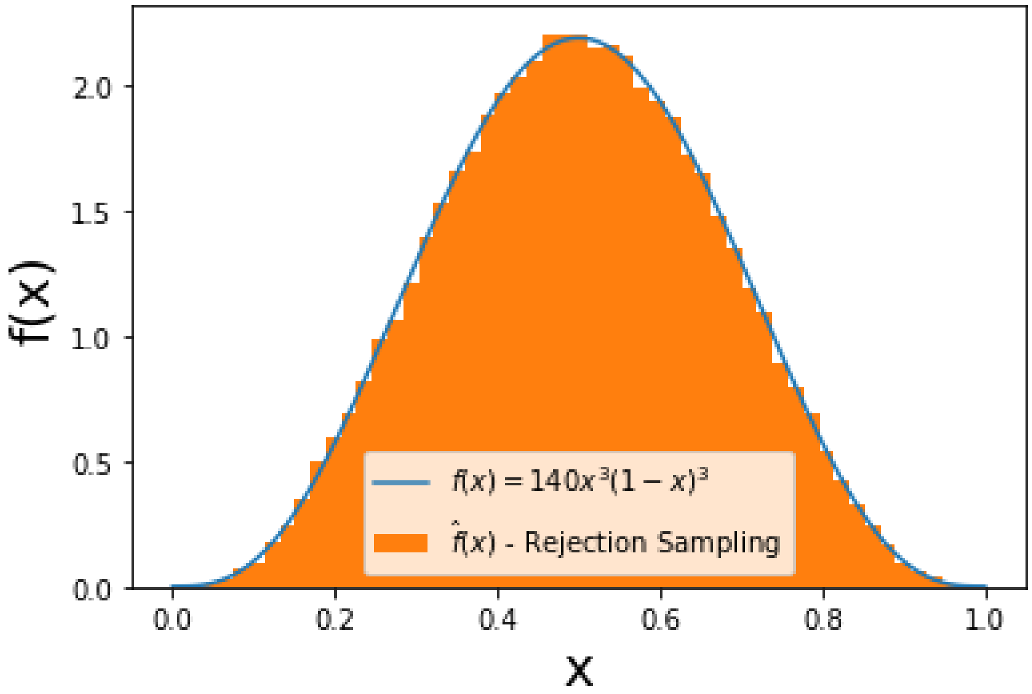
[112]:
```
(2.1875, 0.500000000000007)
```

[113]:
```
x = sampler.sample_single()
x, f(x)
```

```
[113]: (0.5361234602367603, 2.1534247403856113)
```

```
[119]: X = np.linspace(xmin, xmax, 1000)
       X_sample = sampler.sample(100000)
       # f_sample = f(X_sample)
       plt.plot(X, f(X), label=r'$f(x)=140x^3(1-x)^3$')
       plt.hist(X_sample, density=True, bins=50, label=r'$\hat{f}(x)$ - Rejection␣
        ↪Sampling')
       plt.xlabel('x', size=20)
       plt.ylabel('f(x)', size=20)
       plt.legend()
```

```
[119]: <matplotlib.legend.Legend at 0x7f9dbfd87940>
```



```
[ ]:
```

5 Question 5

- The PDF for a continuous uniform distribution can be given by:

$$g(x) = \frac{1}{b-a}, \quad a \leq x \leq b$$

Our bounding distribution needs to be selected such that it can be sampled across all of the same values our given distribution can be sampled over. Since our given distribution is defined for $x \geq 0$, our continuous uniform distribution would have parameters $a = 0$ and $b = \infty$ which gives a nonsensical $g(x)$. And so our issue for using a continuous random variable is that our given random variable has an infinite domain but a continuous random variable required a finite domain.

- If we consider using a normal distribution, we can note that we do not have the same issue as previous, the normal distribution is defined over $-\infty < x < \infty$, and so it covers the same values as our given distribution. Is the issue that our distribution covers x values outside the domain of our given distribution? No, if we sample the normal distribution and get a value less than 0, we can just ignore it and keep sampling till we get a value greater than zero.

  The issue for trying to use a normal distribution is that regardless of the value for c, there is eventually some x value where the constraint

  $$f(x) \leq cg(x),\ \forall x$$

  will no longer be valid. This is because our normal distribution decays at a rate of $\exp(-x^2)$ as x increases while our given distribution decays at a rate of $\exp(-x)$ as x increases.

- Let's look at the probability that we will sample a value greater than 20 from our given distribution:

  $$\int_{20}^{\infty} 2e^{-2x} dx = 4.2 \times 10^{-18}$$

  The probability of sampling a number greater than 20 is tiny. Let's assume that we will never sample a value greater than 20 and essentially truncate our probability distribution to have the domain $0 \leq x \leq 20$.

  We can now use a simple continuous uniform distribution over $0 \leq x \leq 20$ (giving $g(x) = \frac{1}{20}$) and $c = 40$ to bound our given distribution. Using this, you can now write an algorithm that utilises rejection sampling to sample the given distribution.