1.7) $X \sim \mathcal{E}xp(\lambda)$

Claim: $\mathbb{P}(X > t+s \mid X > t) = \mathbb{P}(X > s) \quad \forall\, s, t \geq 0$

$$\mathbb{P}(X > t+s \mid X > t) = \mathbb{P}\left[\frac{(X > t+s) \cap (X > t)}{\mathbb{P}[X > t]}\right]$$

Since $s, t \geq 0$, $t+s \geq t \Rightarrow (X > t+s) \subseteq (X > t)$

$$\Rightarrow \mathbb{P}[(X > t+s) \cap (X > t)] = \mathbb{P}(X > t+s)$$

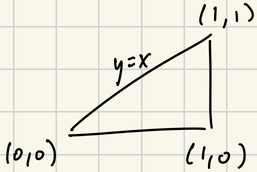
$$\Rightarrow \mathbb{P}(X > t+s \mid X > t) = \frac{\mathbb{P}(X > t+s)}{\mathbb{P}(X > t)} = \frac{\int_{t+s}^{+\infty} \lambda e^{-\lambda x}\,dx}{\int_{t}^{+\infty} \lambda e^{-\lambda x}\,dx}$$

$$= \frac{-e^{-\lambda x}\Big|_{x=t+s}^{+\infty}}{-e^{-\lambda x}\Big|_{x=t}^{+\infty}} = \frac{e^{-\lambda(t+s)}}{e^{-\lambda t}} = e^{-\lambda s}$$

$$\mathbb{P}(X > s) = \int_{s}^{+\infty} \lambda e^{-\lambda x}\,dx = -e^{-\lambda x}\Big|_{x=s}^{+\infty} = -(0 - e^{-\lambda s})$$

$$= e^{-\lambda s} \qquad \blacksquare$$

1.20)



$X \sim Unif(0,1)$

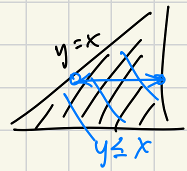
$(Y \mid X = x) \sim Unif(0, x)$

a) $f_{X,Y}(x,y) = f_{Y|X}(x,y)\, f_X(x)$

$= 1/x \quad , \quad x \in [0,1] \quad , \quad y \in [0,x]$

$\Updownarrow$

b) $f_Y(y) = \int_{\mathbb{R}} f_{X,Y}(x,y)\, dx = \int_y^1 1/x \, dx \qquad y \in [0,1] \; , \; x \in [y,1]$



$= \ln|x| \Big]_y^1 = \ln(1) - \ln(y)$

$= -\ln(y) \quad , \quad y \in [0,1]$

$\in \Delta$

c) $f_{X|Y}(x,y) = \dfrac{f_{X,Y}(x,y)}{f_Y(y)} = \dfrac{1/x}{-\ln(y)} \qquad \begin{array}{l} 0 \le y \le x \le 1 \\ 0 \le y \le 1 \end{array}$

$= \dfrac{1}{-x\ln(y)} \quad , \quad 0 \le y \le x \le 1 \qquad (\in \Delta)$

d) $\mathbb{E}(X \mid Y = y) = \int_{-\infty}^{+\infty} x\, f_{X|Y}(x,y)\, dx = \int_y^1 x \frac{1}{-x\ln(y)} = -\int_y^1 \frac{1}{\ln y}\, dx$

Assuming $y \in [0,1]$

$= \dfrac{x}{\ln y} \Big|_{x=y}^{1} \Rightarrow \mathbb{E}(X \mid Y = y) = \dfrac{1-y}{\ln y}$

e)

$$\mathbb{E}X = \int_{-\infty}^{+\infty} x f_X(x)\,dx = \int_0^1 x\,dx = \frac{1}{2}x^2\Big|_0^1 = \frac{1}{2}$$

$$\mathbb{E}Y = \int_{-\infty}^{+\infty} y f_Y(y)\,dy = -\int_0^1 y\ln(y)\,dy$$

$u = \ln y \qquad du = \frac{1}{y}dy$

$v = \frac{1}{2}y^2 \qquad dv = y\,dy$

$$= \frac{1}{2}y^2\ln y\Big]_0^1 - \int_0^1 \frac{1}{2}y^2\frac{1}{y}\,dy$$

$$= \frac{1}{2}y^2\ln y\Big|_0^1 - \frac{1}{2}\int_0^1 y\,dy = \frac{1}{2}y^2\ln y\Big|_0^1 - \frac{1}{4}y^2\Big|_0^1$$

$$= \left[\frac{1}{2}\left(-\lim_{y\to 0} y^2\ln y\right) - \frac{1}{4}\right] = -\frac{1}{2}\lim_{y\to 0}\frac{\ln y}{y^{-2}} - \frac{1}{4}$$

$$= -\frac{1}{2}\lim_{y\to 0}\frac{y^{-1}}{-2y^{-3}} - \frac{1}{4} = -\frac{1}{2}\lim_{y\to 0} -2y^2 - \frac{1}{4} = \frac{1}{4}$$

$\Rightarrow$ $\mathbb{E}X = \frac{1}{2}$, $\mathbb{E}Y = \frac{1}{4}$

In [102…

```
import numpy as np
import matplotlib.pyplot as plt
import math
import bisect
from collections import defaultdict
from typing import List
```

Rejection sampling

In this worksheet you'll be doing the following tasks:

1. Define a function to sample from a discrete distribution.
2. Define classes for continuous and discrete random variables (provided).
3. Define a function to sample from a mixture distribution.
4. Define a rejection sampling function.
5. Test your functions on some examples.

After this assignment, we'll extract this code into a separate module and keep importing it in the future assignments.

Discrete distribution

1. To start with, copy your `RNG` class from the previous assignment below. Make sure to delete the unimplemented `mixture` method if one still exists there.
2. Also, the method `np.random.random()` returns a single random number (and not a list with a single element). Make sure this is the same behavior in your `RNG` class for all the sampling methods. If you simply called the method `self.uniform` and used list comprehension you should be fine. If you wrote your own loop then make sure to return a single number and not a list with a single element if `size=None` (not when `size=1`). I ran into this issue when testing my code.
3. Then define a method `discrete` in the `RNG` class that takes a list of (unnormalized) probabilities and a list of values and returns a random sample from the corresponding discrete distribution. For example, if the input is `weights= [2, 8]` and `list=[3, 5]` then the output should be `3` with probability `0.2` and `5` with probability `0.8`.

In [102…

```
class RNG:
    def __init__(self, seed=None):
        self._rng = np.random.default_rng(seed)
        self.random = self._rng.random

    def seed(self, seed=None):
        self._rng = np.random.default_rng(seed)
```

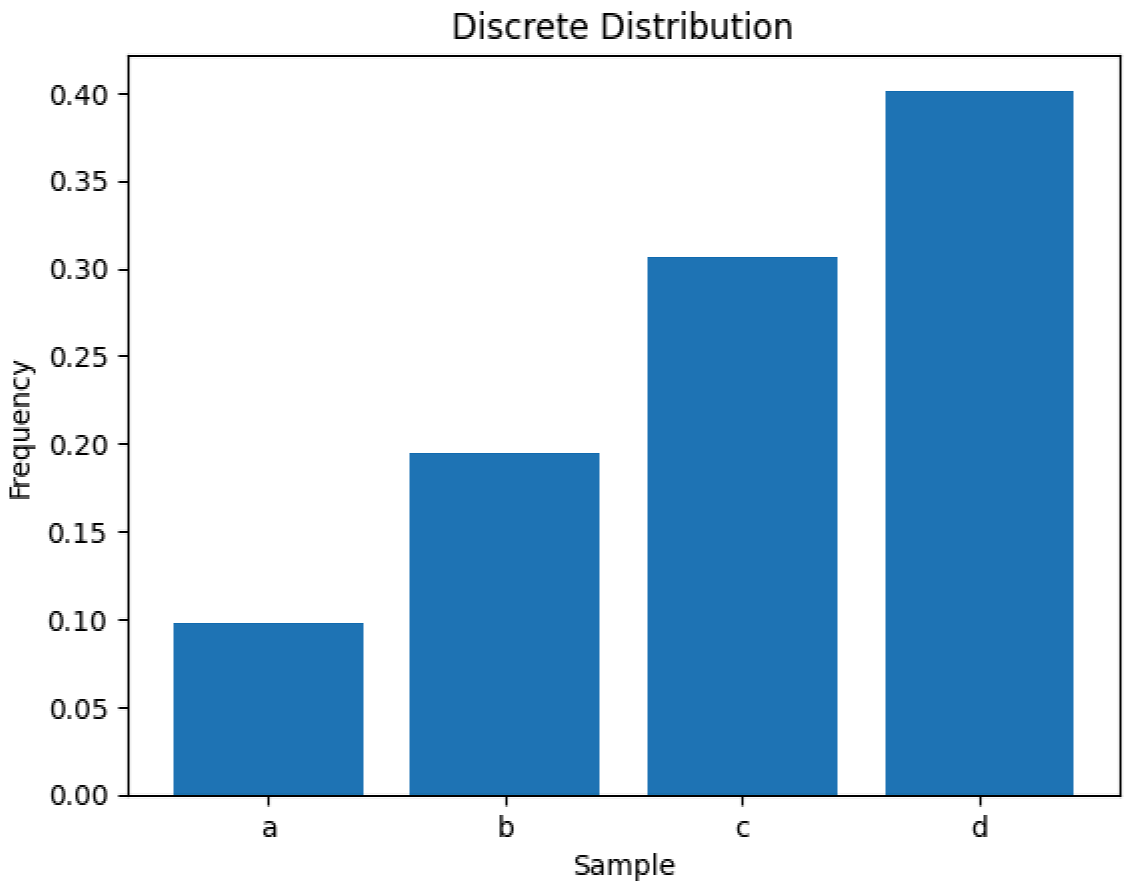
```
    def uniform(self, low=0.0, high=1.0, size=None):
        return low + (high - low) * self.random(size)

    # choose a random object from a list
    def choice(self, list_, size=None):
        u = self.random(size)
        return list_[np.floor(u * len(list_)).astype(int)]

    def discrete(self, weights, list_=None, size=None):
        total_weight = sum(weights)
        intervals = np.cumsum(weights) / total_weight
        u_samples = self.random(size)
        return [list_[bisect.bisect_left(intervals, s)] for s in u_samples]

    def bernoulli(self, p=0.5, size=None):
        assert 0 <= p <= 1, "p must be between 0 and 1"
        u = self.random(size)
        return (u < p).astype(int)

    def binomial(self, n=1, p=0.5, size=None):
        assert 0 <= p <= 1, "p must be between 0 and 1"
        sample = lambda x, y: np.sum(self.bernoulli(x, y))
        return np.array([sample(p, n) for _ in range(size)])

    def exponential(self, lambda_=1.0, size=None):
        assert lambda_ > 0, "lambda must be positive"
        return -1/lambda_ * np.log(1 - self.uniform(size=size))

    def normal(self, mean=0.0, std=1.0, size=None):
        R = np.sqrt(-2 * np.log(1 - self.uniform(size=size))) # u1
        Theta = 2 * np.pi * self.uniform(size=size) # u2
        return mean + std * R * np.cos(Theta) # element-wise multiplication

    def poisson(self, lambda_=1.0, size=None):
        assert lambda_ > 0, "lambda must be positive"
        samples = np.zeros(size)
        for i in range(size):
            N, sum = 0, 0
            while True:
                X = self.exponential(lambda_, size=1)
                sum += X
                if sum > 1:
                    break
                N += 1
            samples[i] = N
        return samples

    def beta(self, a=1.0, b=1.0, size=None):
        assert a > 0 and b > 0, "a and b must be positive"
        assert isinstance(a, int) and isinstance(b, int), "a and b must be i
        k, n = a, b + a - 1
        samples = np.array([self.uniform(size=n) for _ in range(size)])
        samples.sort(axis=1)
        return np.array([row[k-1] for row in samples])
```

```
    def triangular(self, low=0.0, high=1.0, mode=None, size=None):
        assert low <= mode <= high, "low <= mode <= high"
        samples = self.uniform(size=size)
        samples.sort()
        crit = (mode - low) / (high - low)
        return np.where(
            samples < crit,
            low + np.sqrt(samples * (high - low) * (mode - low)),
            high - np.sqrt((1 - samples) * (high - low) * (high - mode))
            )

    def weibull(self, shape=1.0, scale=1.0, size=None):
        assert shape > 0, "shape must be positive"
        assert scale > 0, "scale must be positive"
        return scale * (-np.log(1 - self.uniform(size=size)))**(1/shape)

rng = RNG()
```

Test your discrete sampling function

In [102…

```
# test discrete distribution
rng.seed(0)
size = 10000

samples = rng.discrete(weights=[0.1, 0.2, 0.3, 0.4], list_=['a', 'b', 'c', '
unique, counts = np.unique(samples, return_counts=True)
frequencies = counts / len(samples)

epsilon = 0.001 # allowance for floating point errors
assert abs(np.sum(frequencies) - 1) < epsilon, "Probabilities do not sum to

# check that the samples are generated correctly
plt.bar(unique, frequencies)
plt.xlabel('Sample')
plt.ylabel('Frequency')
plt.title('Discrete Distribution')
plt.show()
```



Continuous random variables

Below we define classes for continuous random variables that bundle the sampling methods together with the density functions. For now we only define the `pdf` method but for a full implementation we would need many more methods such as `cdf`, `ppf`, `mean`, `var`, etc. See the documentation of `scipy.stats.norm` class for an example. These classes are provided for you and you don't need to do anything with them. You can move on to the `Mixture` class.

In [102…

```
class RandomVariable():
    def __init__(self, rng):
        self.rng = rng

    def pdf(self, x):
        raise NotImplementedError

    def rsv(self, size=None):
        raise NotImplementedError

class Uniform(RandomVariable):
    def __init__(self, rng, low=0.0, high=1.0):
        super().__init__(rng)
        self.low = low
        self.high = high
```

```
    def rsv(self, size=None):
        return np.array(self.rng.uniform(low=self.low, high=self.high, size=

    def pdf(self, x):
        return np.array(np.where((x >= self.low) & (x <= self.high), 1 / (se

class Beta(RandomVariable):
    def __init__(self, rng, a=1.0, b=1.0):
        if not isinstance(a, int) or not isinstance(b, int):
            raise ValueError("a and b must be integers")
        super().__init__(rng)
        self.a = a
        self.b = b

    def rsv(self, size=None):
        return np.array(self.rng.beta(a=self.a, b=self.b, size=size))

    def pdf(self, x):
        return np.array(x**(self.a - 1) * (1 - x)**(self.b - 1) / self.beta(

    def beta(self, a, b):
        return math.factorial(a - 1) * math.factorial(b - 1) / math.factoria

class Exponential(RandomVariable):
    def __init__(self, rng, lambda_=1.0):
        super().__init__(rng)
        self.lambda_ = lambda_

    def rsv(self, size=None):
        return np.array(self.rng.exponential(lambda_=self.lambda_, size=size

    def pdf(self, x):
        return np.array(self.lambda_ * np.exp(-self.lambda_ * x))

class Normal(RandomVariable):
    def __init__(self, rng, mean=0.0, std=1.0):
        super().__init__(rng)
        self.mean = mean
        self.std = std

    def rsv(self, size=None):
        return np.array(self.rng.normal(mean=self.mean, std=self.std, size=s

    def pdf(self, x):
        return np.array(np.exp(-0.5 * ((x - self.mean) / self.std)**2) / np.

class Bernoulli(RandomVariable):
    def __init__(self, rng, p=0.5):
        super().__init__(rng)
        self.p = p

    def rsv(self, size=None):
        return np.array(self.rng.bernoulli(p=self.p, size=size))

    def pdf(self, x):
        return np.array(np.where(x == 1, self.p, 1 - self.p))
```

```
class Poisson(RandomVariable):
    def __init__(self, rng, lambda_=1.0):
        super().__init__(rng)
        self.lambda_ = lambda_

    def rsv(self, size=None):
        return np.array(self.rng.poisson(lambda_=self.lambda_, size=size))

    def pdf(self, x):
        return np.array(self.lambda_**x * np.exp(-self.lambda_) / math.facto

class Binomial(RandomVariable):
    def __init__(self, rng, n=1, p=0.5):
        super().__init__(rng)
        self.n = n
        self.p = p

    def rsv(self, size=None):
        return np.array(self.rng.binomial(n=self.n, p=self.p, size=size))

    def pdf(self, x):
        return np.array(math.factorial(self.n) / (math.factorial(x) * math.f

class Weibull(RandomVariable):
    def __init__(self, rng, shape=1.0, scale=1.0):
        super().__init__(rng)
        self.shape = shape
        self.scale = scale

    def rsv(self, size=None):
        return np.array(self.rng.weibull(shape=self.shape, scale=self.scale,

    def pdf(self, x):
        return np.array((self.shape / self.scale) * (x / self.scale)**(self.

class Triangular(RandomVariable):
    def __init__(self, rng, low=0.0, high=1.0, mode=None):
        super().__init__(rng)
        self.low = low
        self.high = high
        self.mode = mode if mode else (low + high) / 2

    def rsv(self, size=None):
        return np.array(self.rng.triangular(low=self.low, high=self.high, mo

    def pdf(self, x):
        return np.array(np.where((x >= self.low) & (x < self.mode),
                                 2 * (x - self.low) / ((self.high - self.low
                                 2 * (self.high - x) / ((self.high - self.lo

class Discrete(RandomVariable):
    def __init__(self, rng, weights, list_=None):
        super().__init__(rng)
        self.weights = weights
        self.list_ = list_
```

```
    def rsv(self, size=None):
        return np.array(self.rng.discrete(weights=self.weights, list_=self.l

    def pdf(self, x):
        return np.array(self.weights[self.list_.index(x)])
```

Mixture Distribution

Next, define a class `Mixture` that takes a list of weights and a list of component random variables. The class should have a method `rsv` that samples a random variable from the mixture distribution and a method `pdf` that evaluates the density function of the mixture distribution. Wrap your `rsv` output in a `np.array` (see above) to make sure it is a numpy array as `numpy` functions do not work well with lists.

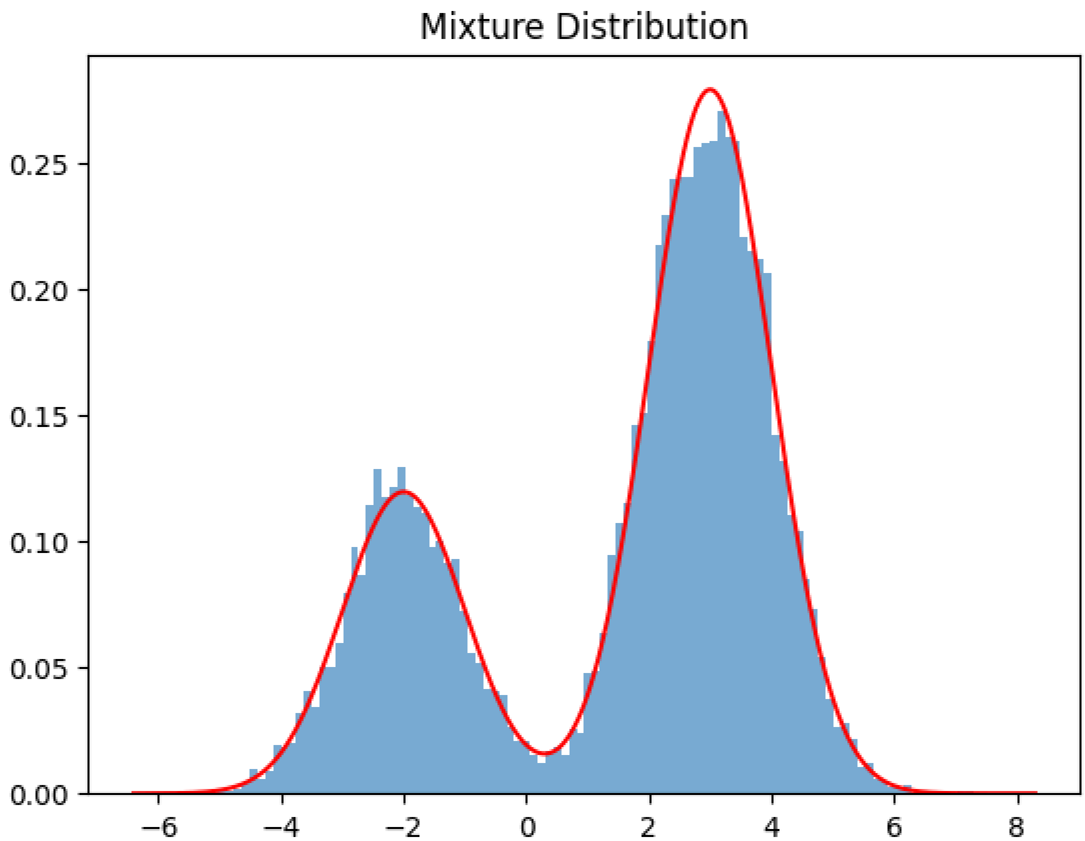
In [102…

```
class Mixture(RandomVariable):
    def __init__(self, rng, weights, components: List[RandomVariable]):
        assert len(weights) == len(components), "weights and components must
        super().__init__(rng)
        self.weights = weights
        self.components = components

    def rsv(self, size=None):
        sampled_components = Discrete(self.rng, self.weights, self.component
        return np.array([comp.rsv() for comp in sampled_components])

    def pdf(self, x):
        return np.array(sum([w * c.pdf(x) for w, c in zip(self.weights, self
```

Test your mixture distribution

In [102…

```
# test mixture of normals
rng.seed(0)
size = 10000

normal1 = Normal(rng, mean=-2, std=1)
normal2 = Normal(rng, mean=3, std=1)

mixture = Mixture(rng, weights=[0.3, 0.7], components=[normal1, normal2])

samples = mixture.rsv(size=size)

# histogram of mixture distribution
plt.hist(samples, bins=math.floor(np.sqrt(size)), density=True, alpha=0.6)
plt.title('Mixture Distribution')

# plot scaled pdf of mixture distribution
x = np.linspace(samples.min() - 1, samples.max() + 1, 1000)
plt.plot(x, mixture.pdf(x), "red")
plt.show()
```



Rejection sampler

In this section, you will implement and test a rejection sampling algorithm to generate samples from a target distribution using a proposal distribution. The rejection sampling algorithm will be implemented in the function `rejection_rsv`.

- Inputs:
  - `target_pdf`: The probability density function (pdf) of the target distribution (can be unnormalized).
  - `proposal_rv`: An instance of a continuous random variable class used as the proposal distribution. You will need to use both the `pdf` and `rsv` methods of this class.
  - `majorizing_constant`: A constant `M` such that `M * proposal_rv.pdf(x)` majorizes `target_pdf(x)` for all `x`.
  - `size`: The number of samples to generate.
- Outputs:
  - `samples`: An `ndarray` of samples from the target distribution.
  - `efficiency`: The proportion of samples that were accepted.

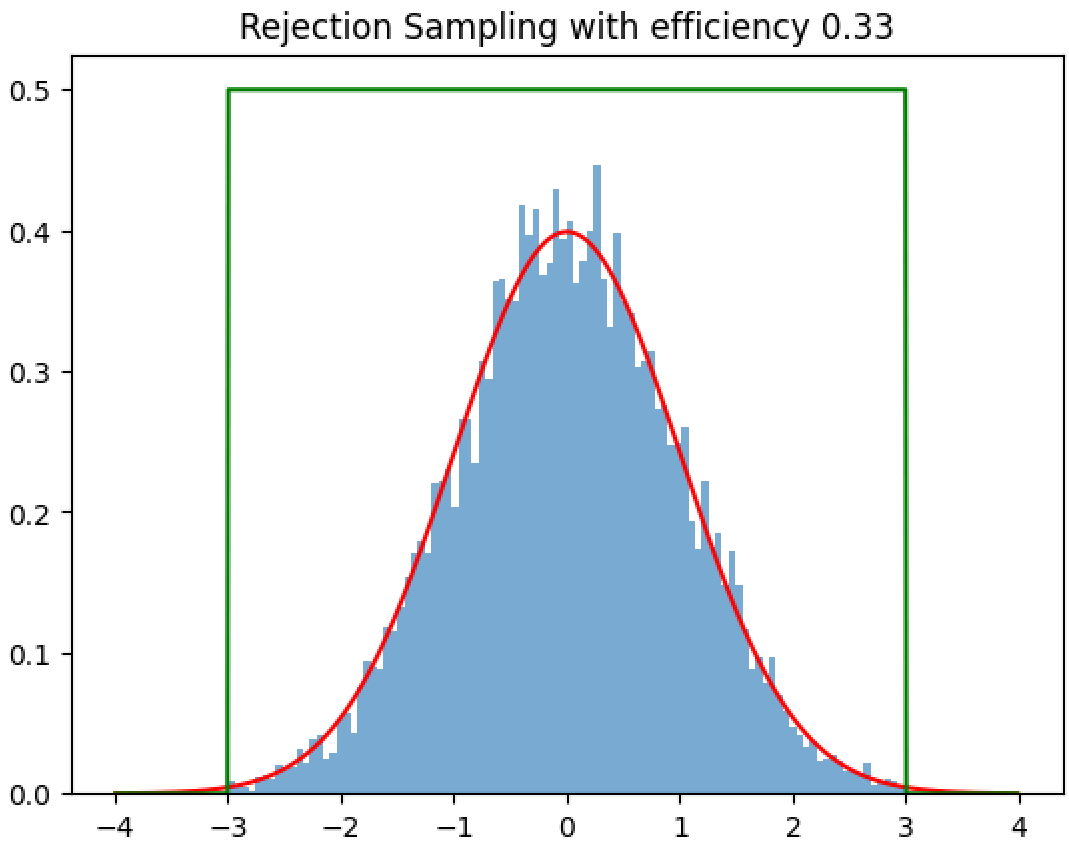
Note that `size` is the number of samples to accept, not the number of samples to generate. You will need to generate more samples than `size` to ensure that you accept `size` samples.

In [102…
```python
def rejection_rsv(target_pdf, proposal_rv: RandomVariable, majorizing_consta
  rng = RNG()
  accept = []
  trials = 0
  U = Uniform(rng, 0, 1)
  while len(accept) < size:
    x = proposal_rv.rsv(1)
    u = U.rsv(None)
    if u <= target_pdf(x) / (majorizing_constant * proposal_rv.pdf(x)):
      accept.append(x)
    trials += 1
  efficiency = size / trials
  return np.array(accept), efficiency
```

Test 1

Test your algorithm on a truncated standard normal distribution. Use a uniform distribution to majorize the target distribution. A sample code is provided below. Fill in the appropriate constants. Feel free to find the majorizing constant by trial and error.

In [102…
```python
# test rejection_rsv for normal distribution
# find appropriate values of low, high, and M
rng.seed(0)
size = 10000

target_pdf = Normal(rng, mean=0, std=1).pdf
proposal = Uniform(rng, low=-3, high=3)

M = 3

samples, efficiency = rejection_rsv(target_pdf, proposal, M, size)

assert len(samples) == size, "Incorrect number of samples"

# histogram of samples
plt.hist(samples, math.floor(math.sqrt(size)), density=True, alpha=0.6)
plt.title('Rejection Sampling with efficiency {:.2f}'.format(efficiency))

# plot scaled pdf of target distribution
x = np.linspace(samples.min() - 1, samples.max() + 1, 1000)
plt.plot(x, target_pdf(x), "red")
plt.plot(x, M*proposal.pdf(x), "green")

plt.show()
```



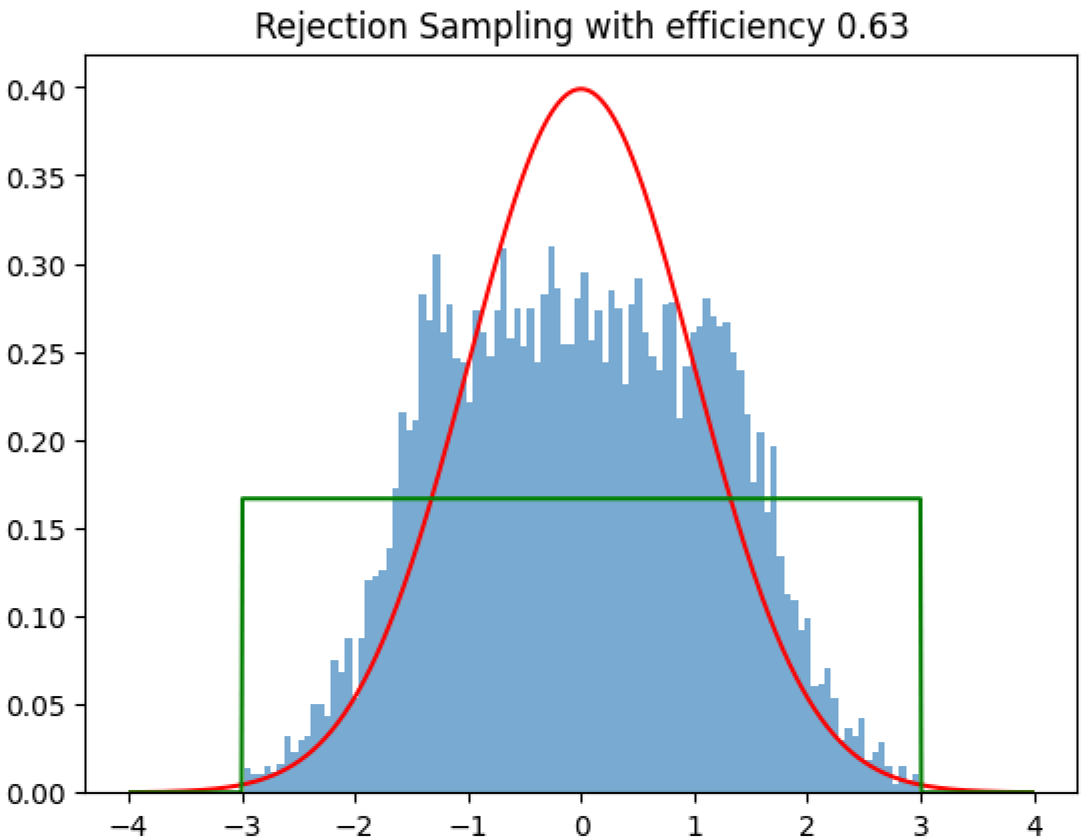
Test 2

Next see what happens if you choose a value of `M` that is too small, say less than half the height of peak of the target distribution.

In [102…

```
# test rejection_rsv for normal distribution
# find appropriate values of low, high, and M
rng.seed(0)
size = 10000

target_pdf = Normal(rng, mean=0, std=1).pdf
proposal = Uniform(rng, low=-3, high=3)

M = 1

samples, efficiency = rejection_rsv(target_pdf, proposal, M, size)

assert len(samples) == size, "Incorrect number of samples"

# histogram of samples
plt.hist(samples, math.floor(math.sqrt(size)), density=True, alpha=0.6)
plt.title('Rejection Sampling with efficiency {:.2f}'.format(efficiency))

# plot scaled pdf of target distribution
x = np.linspace(samples.min() - 1, samples.max() + 1, 1000)
plt.plot(x, target_pdf(x), "red")
plt.plot(x, M*proposal.pdf(x), "green")
```

```
plt.show()

# our distribution flattens out where the majorizing function is too low
```



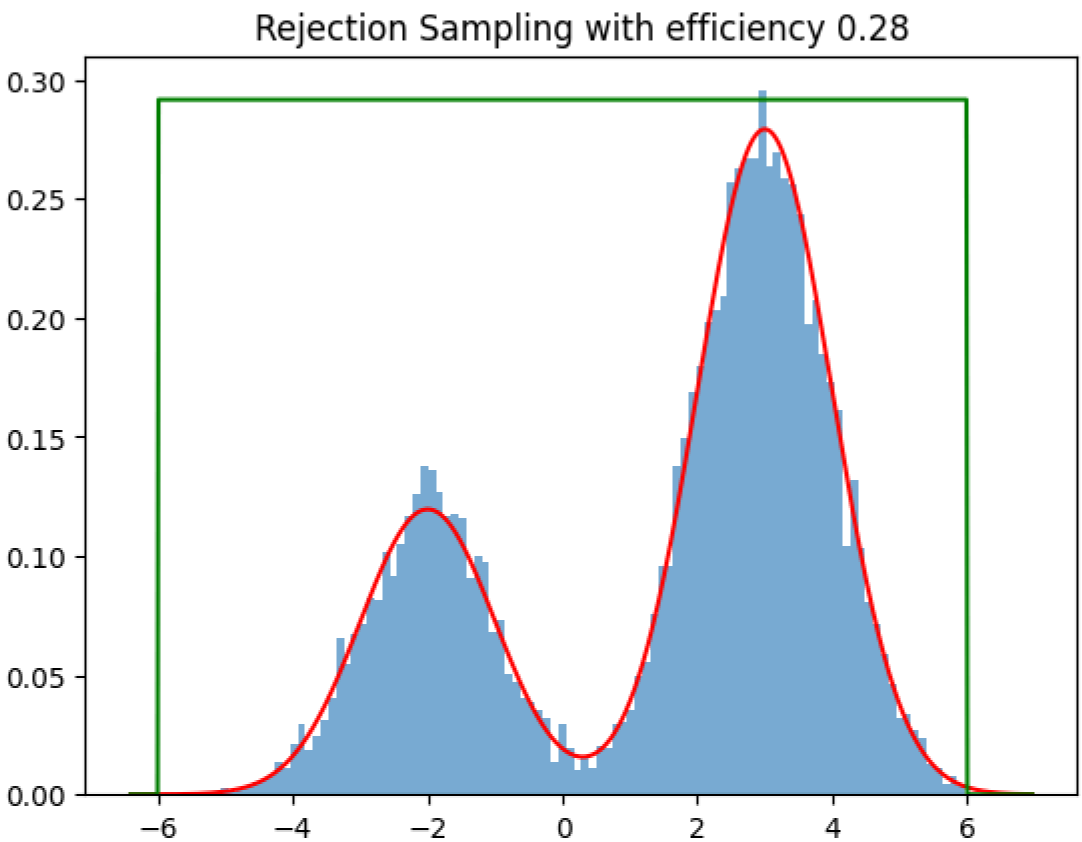
Test 3

Test your algorithm on a (truncated) mixture of a normal distributions. Use a uniform distribution to majorize the target distribution. This time write your own code. The final graph should show the target distribution, the scaled proposal distribution, and a histogram of the samples.

In [103…

```
# test rejection rsv
rng.seed(0)
size = 10000

normal1 = Normal(rng, mean=-2, std=1)
normal2 = Normal(rng, mean=3, std=1)

mixture = Mixture(rng, weights=[0.3, 0.7], components=[normal1, normal2])

target_pdf = mixture.pdf
proposal = Uniform(rng, low=-6, high=6)

M = 3.5

samples, efficiency = rejection_rsv(target_pdf, proposal, M, size)
```

```python
assert len(samples) == size, "Incorrect number of samples"

# histogram of samples
plt.hist(samples, math.floor(math.sqrt(size)), density=True, alpha=0.6)
plt.title('Rejection Sampling with efficiency {:.2f}'.format(efficiency))

# plot scaled pdf of target distribution
x = np.linspace(samples.min() - 1, samples.max() + 1, 1000)
plt.plot(x, target_pdf(x), "red")
plt.plot(x, M*proposal.pdf(x), "green")

plt.show()
```



Test 4

Finally, test your algorithm on a (truncated) mixture of a normal distributions. Use a mixture of two or more uniform distributions to majorize the target distribution. Compare the efficiency of the algorithm to the one above.

In [103…

```python
# test rejection rsv
rng.seed(0)
size = 10000

normal1 = Normal(rng, mean=-2, std=1)
normal2 = Normal(rng, mean=3, std=1)
```

```python
mixture = Mixture(rng, weights=[0.3, 0.7], components=[normal1, normal2])

target_pdf = mixture.pdf

uniform1 = Uniform(rng, low=-4.5, high=1)
uniform2 = Uniform(rng, low=1, high=5)
proposal = Mixture(rng, weights=[1.2, 2], components=[uniform1, uniform2])

M = 0.6

samples, efficiency = rejection_rsv(target_pdf, proposal, M, size)

assert len(samples) == size, "Incorrect number of samples"

# histogram of samples
plt.hist(samples, math.floor(math.sqrt(size)), density=True, alpha=0.6)
plt.title('Rejection Sampling with efficiency {:.2f}'.format(efficiency))

# plot scaled pdf of target distribution
x = np.linspace(samples.min() - 1, samples.max() + 1, 1000)
plt.plot(x, target_pdf(x), "red")
plt.plot(x, M*proposal.pdf(x), "green")

plt.show()

# we have a higher efficiency here
```