Integration and Sampling

Written by:

- Philip Ilten (University of Cincinnati)
- Steve Mrenna (University of Cincinnati)

In this notebook we first investigate sampling random numbers other than uniform, and then use random number sampling to calculate integrals.

Requirements

We need a random number generator. We could use one of the RNGs implemented in `rng.ipynb`, but instead we will use the default `numpy` RNG. We also need the `math` module and `matplotlib`.

```
# Import the `numpy` and `math` modules.
import numpy as np
import math

# Import the `matplotlib` module.
import matplotlib.pyplot as plt

# Create an RNG, with a seed of 10.
rng = np.random.default_rng(10)
```

Introduction

Typical events produced within the Large Hadron Collider (LHC) from colliding protons have $\mathcal{O}(100)$ or more particles produced. When calculating a cross-section for a two-to-two process we typically only need to integrate over two variables, θ and ϕ. A two-to-n process requires integrating over $3n-4$ variables, so a typical LHC event would require integrating over $\mathcal{O}(300)$ variables. This is numerically challenging, at best, and with current technology is just simply not possible. To calcululate LHC events, we can instead factorise the problem into more manageable parts using probabilistic methods. Even still, calculating a perturbative cross-section for a 4-body final state requires integrating over 8 variables which is a challenging numerical integration. The bottom line is that performing high dimension integrals quickly and efficiently is a core problem in particle physic and is very numerically challenging.

However, before we tackle integration with MC, we need to first discuss how we can efficiently sample distributions. In the `rng.ipynb` notebook we have hard to make a good generator for uniformly-

distributions. In the Fig1.1pynb notebook, we have had to make a good generator for uniformly distributed random variates. In practice, however, the probability distributions of interest are not uniform. Fortunately, uniform random variates can either be transformed into a different distribution or used as part of an accept/reject algorithm that converges to the desired probability distribution. Random variates -- uniform or not -- are also a primary part of the Monte Carlo integration method, so it is worthwhile to know how to transform uniform into complicated.

In this notebook, we only consider continous distributions, but everything that we say can be applied, with some modification, to discrete distributions.

Analytic Sampling

Analytic, or inverse cumulative distribution function (CDF) sampling allows us to transform a uniform distribution into our target distribution, $f(x)$. However, this is not possible for every $f(x)$. To sample $f(x)$ the following must generally be fulfilled.

1. The sampling of $f(x)$ is bounded, where over this range $f(x)$ is positive.

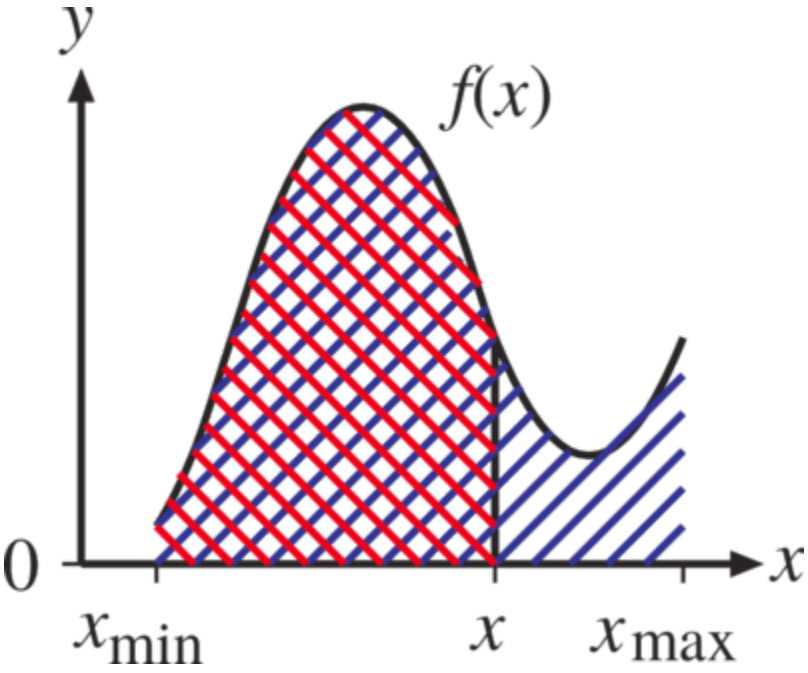
$$f(x) \geq 0 \text{ for } x_{\min} < x < x_{\max}$$

2. The integral of $f(x)$ can be calculated.

$$F(x) = \int \mathrm{d}x\, f(x)$$

3. The integral of $f(x)$ can be inverted, which we label $F^{-1}(x)$.

With these three conditions met we can then sample a distribution for $f(x)$ as follows. First, we can consider integrating a distribution from $x_{\min}$ to x, as shown in the figure below.



We then draw a uniform random number R which gives us the following relation.

$$\int_{x_{\min}}^{x} \mathrm{d}x' \, f(x') = R \int_{x_{\min}}^{x_{\max}} \mathrm{d}x' \, f(x')$$

We then perform the integration, where $F(x)$ is the indefinite integral of $f(x)$.

$$F(x) - F(x_{\min}) = R(F(x_{\max}) - F(x_{\min}))$$

We can then write $F(x_{\max}) - F(x_{\min})$ as A, the area under the integral.

$$F(x) - F(x_{\min}) = RA$$

We then solve for x.

$$x = F^{-1}(F(x_{\min}) + RA)$$

So, we can uniformly sample R and then use the final relation to transform this into x, as sampled from $f(x)$.

Exercise: generic sampler

Before we try to generate any specific distributions using this method, let us first set up a generic sampler class which uses the steps above.

```
class SampleAnalytic:
    """
    Base class to analytically sample a target distribution from a uniform
    distribution.
    """

    def __init__(self, rng, xmin, xmax):
        """
        Initialize the sampler, given the limits on f(x).

        rng:  uniform random number generator, should have method `uniform()`.
        xmin: lower bound of the sampling region.
        xmax: upper bound of the sampling region.
        """
        self.rng = rng
        self.xmin = xmin
        self.xmax = xmax
        self.F_xmin = self.F(xmin)
        self.area = self.F(xmax) - self.F(xmin)

    def f(self, x):
        """
        Return the function being sampled, f(x). This method is not necessary,
        but very useful for importance sampling and checking the distribution.

        x: value to calculate f(x) for.
        """
        return 0.0
```

```
    def F(self, x):
        """
        Returns F(x), the indefinite integral for f(x).

        x: value to calculate the indefinite integral for f(x).
        """
        return 0.0

    def F_inv(self, f):
        """
        Returns the inverse of the F(x).

        F: the value of F(x) to calculate the inverse.
        """
        return 0.0

    def __call__(self):
        """
        Return the sampled value.
        """
        # Sample the uniform random number.
        r = self.rng.uniform()
        return self.F_inv(self.F_xmin + r * self.area)
```

Exercise: linear function

Sample from a linear distribution with the following form.

$$f(x) = mx + b$$

```
class SampleLinear(SampleAnalytic):
    """
    Class to analytically sample a linear function.
    """

    def __init__(self, rng, xmin, xmax, m, b):
        """
        Initialize the sampler, given the limits on f(x) and the linear
        parameters.

        f(x) = mx + b

        rng:  uniform random number generator, should have method `uniform()`.
        xmin: lower bound of the sampling region.
        xmax: upper bound of the sampling region.
        m:    slope of the linear distribution.
        b:    intercept of the linear distribution.
        """
        # Set the linear parameters. This must be done before the base class
        # is initialized
```

```
        # is initialized.
        self.m = m
        self.b = b
        # Initialize the base class.
        super().__init__(rng, xmin, xmax)

    def f(self, x):
        """
        Return the function being sampled, f(x).

        x: value to calculate f(x) for.
        """
        return self.m * x + self.b

    def F(self, x):
        """
        Returns F(x), the indefinite integral for f(x).

        x: value to calculate the indefinite integral for f(x).
        """
        return self.m * x**2 / 2 + self.b * x

    def F_inv(self, f):
        """
        Returns the inverse of the F(x).

        F: the value of F(x) to calculate the inverse.
        """
        # Handle the special case of no slope.
        if self.m == 0:
            return f / self.b
        else:
            return abs(((self.b**2 + 2 * self.m * f) ** 0.5 - self.b) / self.m)
```

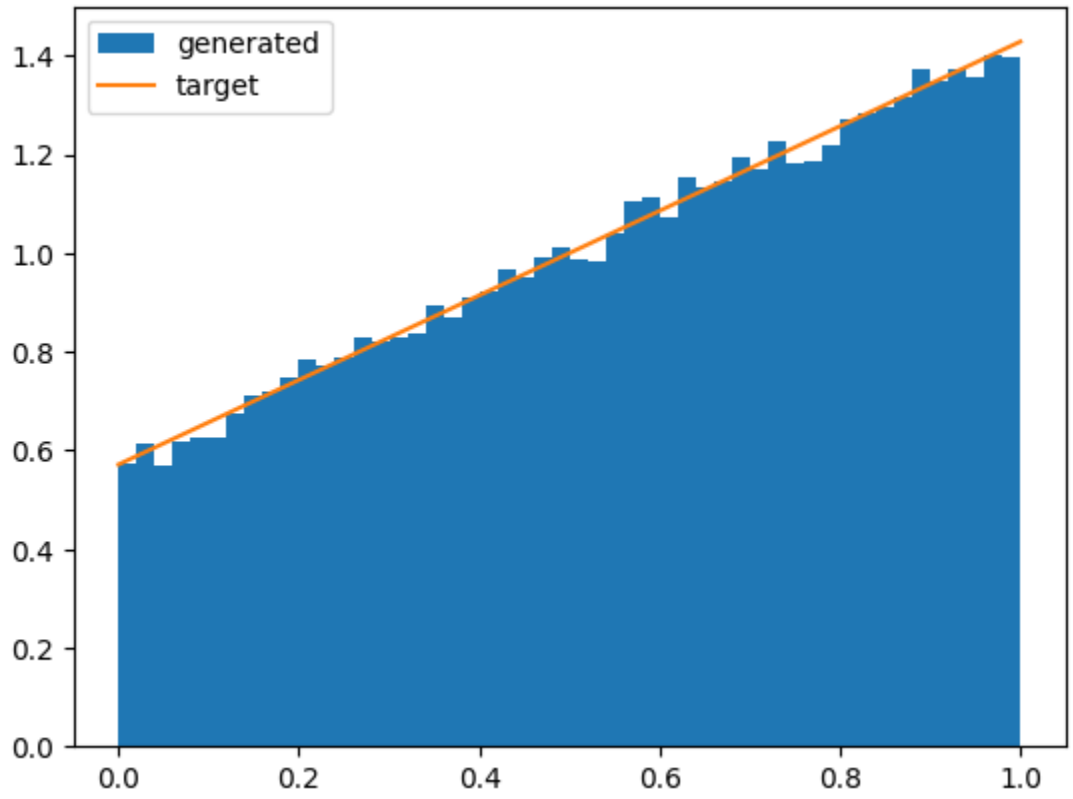
Now, let us test whether this sampler works for $m = 3$ and $b = 2$ between 0 and 1. We will want to test a number of distributions, so let us first write a little method that does just that. The following method plots the normalized sampled distribution and compares this to the normalized target function $f(x)$.

```
def plot_sampler(sampler, n=100000, bins=50, points=1000):
    """
    Plots the distribution from a sampler for a specific distribution.

    sampler: random number sampler.
    n:       number of points to sample.
    bins:    number of bins in the histogram.
    points:  number of points to evaluate for the function.
    """
    # Sample the distribution.
    rns = []
    for i in range(0, n):
```

```
            # Store the value.
            rns += [sampler()]

    # Calculate the target function.
    xs = np.linspace(sampler.xmin, sampler.xmax, points)
    fs = [sampler.f(x) / sampler.area for x in xs]

    # Create the plot.
    fig, ax = plt.subplots()

    # Draw the histogram.
    ax.hist(rns, bins=bins, density=True, label="generated")

    # Draw the target function, make sure to normalize.
    ax.plot(xs, fs, label="target")

    # Draw the legend.
    ax.legend()

    return fig, ax
```

With this method, test to see if the distribution being generated matches the target.

```
# Create the sampler.
sampler = SampleLinear(rng, 0, 1, 3, 2)

# Plot the comparison.
plot_sampler(sampler);
```



Exercise: Breit-Wigner

Also known as a Cauchy distribution, the relativistic Breit-Wigner is of particular importance in particle physics because it can describe the distribution of masses for a specific particle type, e.g., a Z boson. If we want to be able to efficiently sample a mass distribution, then we need to be able to sample a relativistic Breit-Wigner. The form of the function is as follows.

$$f(x) = \frac{1}{\pi}\left(\frac{\gamma}{(x - x_0)^2 + \gamma^2}\right)$$

Remember, the normalization of this function does not matter. In a particle physics context, γ is $M\Gamma$ where M is the mass of the particle and Γ is its width. Then, x_0 is M.

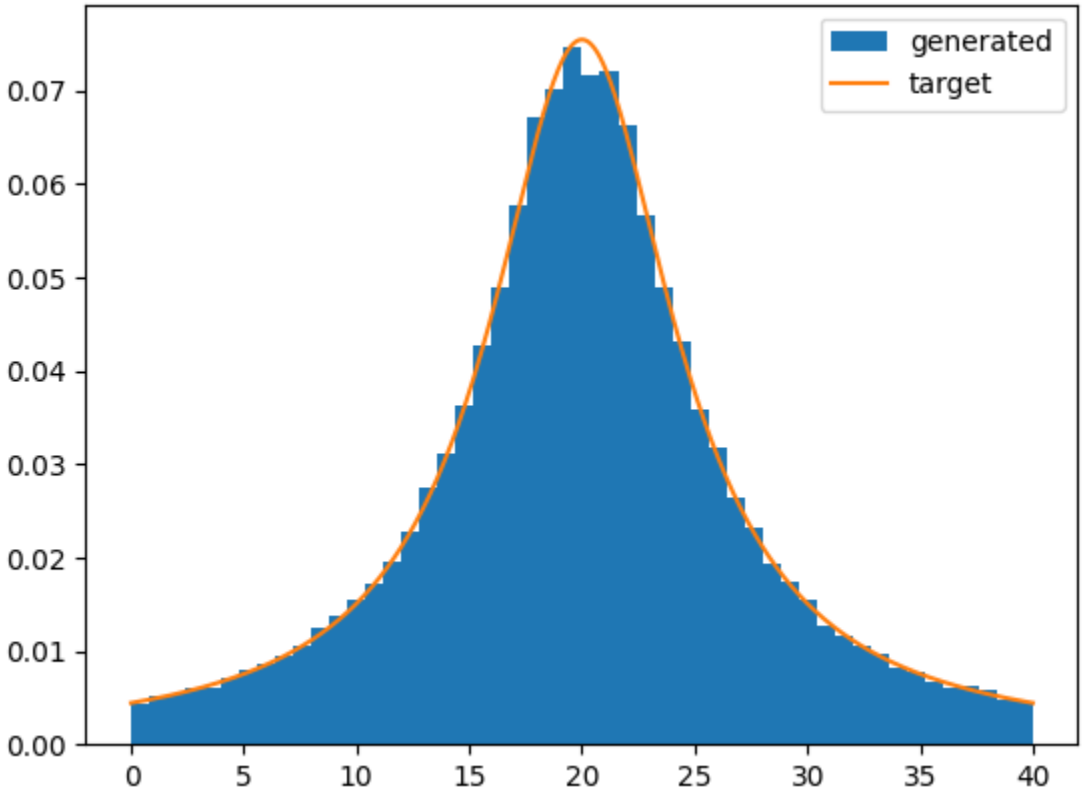
Implement a sampler for the Breit-Wigner distribution.

```
class SampleCauchy(SampleAnalytic):
    """
    Class to analytically sample a Cauchy function.
    """

    def __init__(self, rng, xmin, xmax, x0, gamma):
        """
        Initialize the sampler, given the limits on f(x) and the linear
        parameters.

        f(x) = 1/pi * (gamma/(x - x0)^2 + gamma^2)

        rng:   uniform random number generator, should have method `uniform()`.
        xmin:  lower bound of the sampling region.
        xmax:  upper bound of the sampling region.
        x0:    location parameter.
        gamma: scale parameter.
        """
        # Set the parameters.
        self.x0 = x0
        self.gamma = gamma
        # Initialize the base class.
        super().__init__(rng, xmin, xmax)

    def f(self, x):
        """
        Return the function being sampled, f(x).

        x: value to calculate f(x) for.
        """
        return 1 / math.pi * self.gamma / ((x - self.x0) ** 2 + self.gamma**2)

    def F(self, x):
        """
```

```
        Returns F(x), the indefinite integral for f(x).

        x: value to calculate the indefinite integral for f(x).
        """
        return 1 / math.pi * math.atan((x - self.x0) / self.gamma) + 1 / 2

    def F_inv(self, f):
        """
        Returns the inverse of the F(x).

        F: the value of F(x) to calculate the inverse.
        """
        return self.x0 + self.gamma * math.tan(math.pi * (f - 1 / 2))
```

Check this distribution for a $\gamma = 5$, $x_0 = 20$, and range of 0 to 40.

```
# Create the sampler.
sampler = SampleCauchy(rng, 0, 40, 20, 5)

# Plot the comparison.
plot_sampler(sampler);
```



Exercise: Gaussian

Perhaps one of the most sampled distributions out there is the Gaussian.

$$f(x) = \frac{1}{} e^{-(x-\mu)^2/(2\sigma^2)}$$

$$f(x) \quad \sqrt{2\pi\sigma^2}$$

As a matter of fact, the Gaussian is critical for many machine learning techniques. However, looking at the function above, it is clear that it is not possible to calculate a closed analytic form for either $F(x)$ or $F^{-1}(x)$. So, our analytic method from above fails. What do we do instead? One option is to numerically calculate the $F(x)$ and $F^{-1}(x)$, for which there are relatively efficient methods.

However, it turns out there is a very clever transform that you can do, commonly called the Box-Muller transform. We won't go through the derivation here, but it has to do with relating Cartesian and polar coordinates. Anyhow, the method is as follows.

1. Sample two random numbers R_1 and R_2.
2. Transform these into two independent Gaussian distributed numbers x_1 and x_2 with the following.

$$x_1 = \sqrt{-2\log(R_1)}\cos(2\pi R_2)$$
$$x_2 = \sqrt{-2\log(R_1)}\sin(2\pi R_2)$$

3. These x_i are for a Gaussian with $\mu = 0$ and $\sigma = 1$, so they need to be muplitied by σ with μ added on.

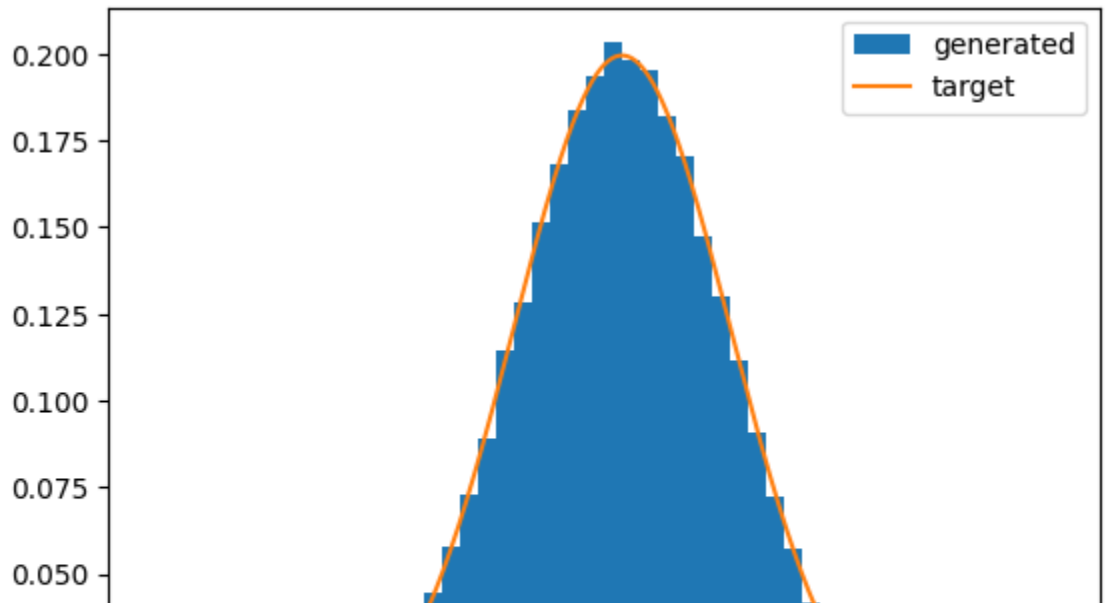
What is great about this method is that not only is it simple and fast, it also is not bounded, which is required of the method we used above! Using this transformation, define a Gaussian sampler.

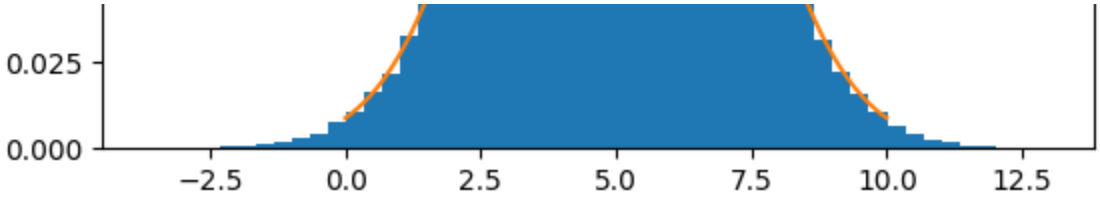
```
class SampleGaussian:
    """
    Class to sample a Gaussian distribution.
    """

    def __init__(self, rng, xmin, xmax, mu, sigma):
        """
        Initialize the sampler. Note, the limits `xmin` and `xmax` here only
        define the limits when used for drawing with the `plot_sampler` method.
        Sampling is performed without any limits.

        rng:   uniform random number generator, should have method `uniform()`.
        xmin:  minimum x for plotting (not sampling).
        xmax:  maximum x for plotting (not sampling).
        mu:    mean of Gaussian.
        sigma: width of Gaussian.
        """
        # Set the parameters.
        self.rng = rng
        self.xmin = xmin
        self.xmax = xmax
        self.mu = mu
        self.sigma = sigma

        # Set the area being sampled. This distribution is normalized
```

```
        # Set the area being sampled. This distribution is normalized.
        self.area = 1

    def f(self, x):
        """
        Return the function being sampled, f(x).

        x: value to calculate f(x) for.
        """
        return (
            1
            / (2 * math.pi * self.sigma**2) ** 0.5
            * math.exp(-((x - self.mu) ** 2) / (2 * self.sigma**2))
        )

    def __call__(self):
        """
        Return the sampled value.
        """
        # Sample the two uniform random numbers.
        r1 = self.rng.uniform()
        r2 = self.rng.uniform()
        # Return only one of the two transformed values.
        return (
            self.sigma * (-2 * math.log(r1)) ** 0.5 * math.cos(2 * math.pi * r2)
            + self.mu
        )
```

Check this distribution.

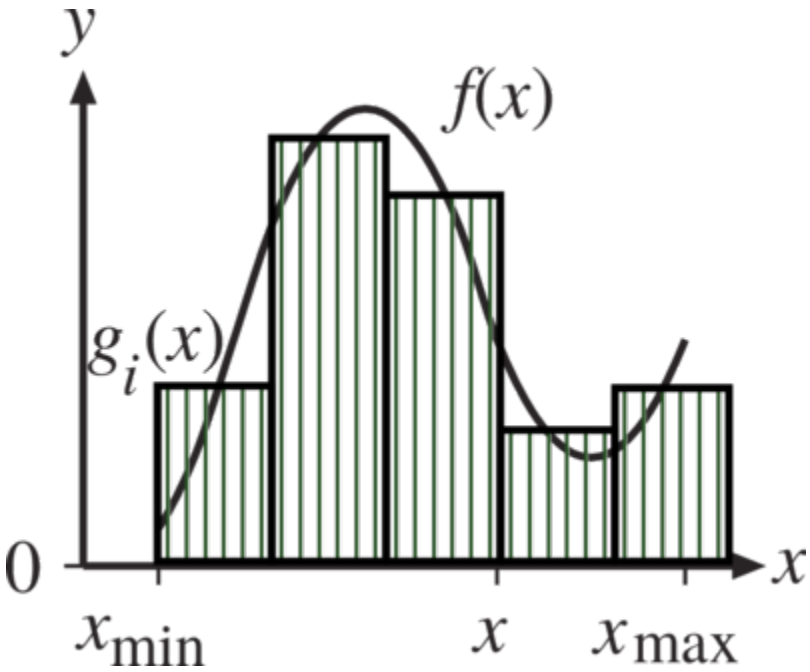
```
# Create the sampler.
sampler = SampleGaussian(rng, 0, 10, 5, 2)

# Plot the comparison.
plot_sampler(sampler);
```



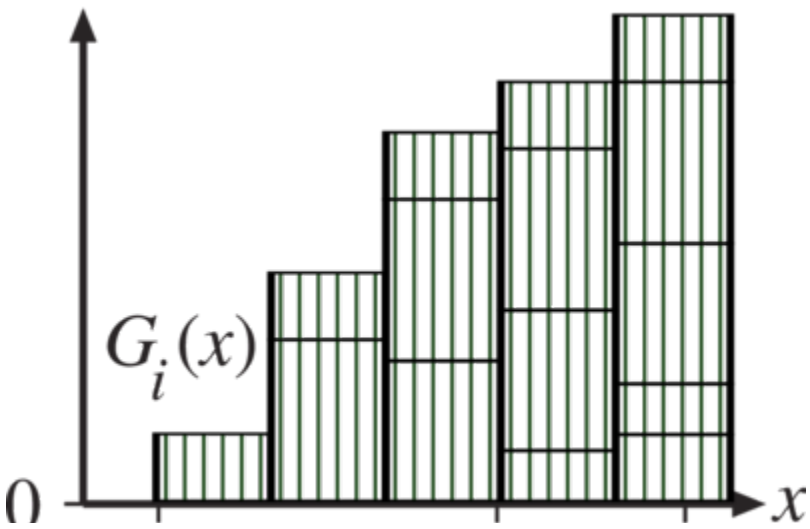


Binned Sampling

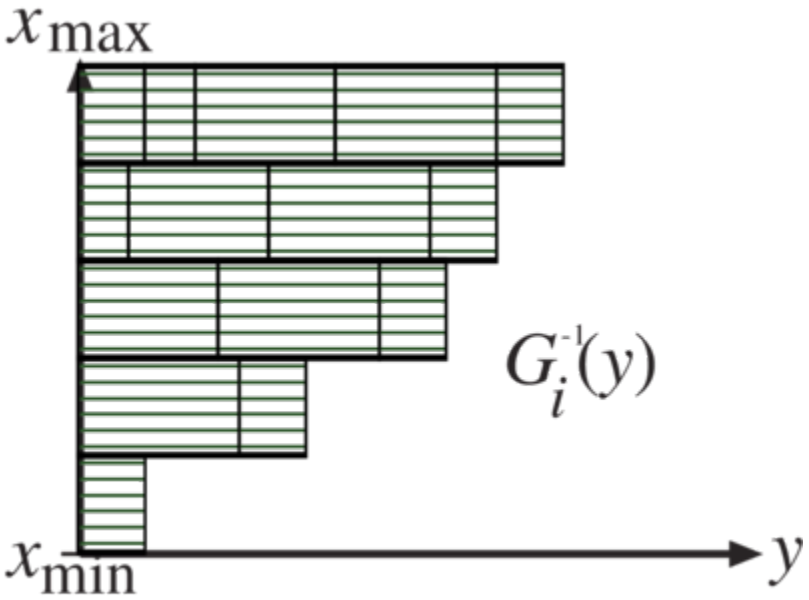
The idea behind binned sampling is exactly the same as for analytic sampling, but rather than inverting the CDF analytically, we do it numerically. The idea is as follows. First, we create a binned probability distribution function (PDF).



We can think of this as splitting $f(x)$ up into g_i divisions (bins). Next, we create the CDF for this distribution. We can do this by creating a new histogram, where each entry is given by the sum of bins, up to that point.



Finally, we invert this CDF by effectively swapping our bin edges x for our function values y.



Let's now try to implement this type of sampling. The algorithm is then as follows.

1. Sample a uniform random number R.
2. Find the corresponding bin i of $G_i^{-1}(x)$.
3. Uniformly sample an R between the bin edges, this is our transformed value.

Example: histograms

A key ingredient to binned sampling is the histogram, so we need to build a little histogram class. Below is an outline of how we can do this. Fill in the missing parts.

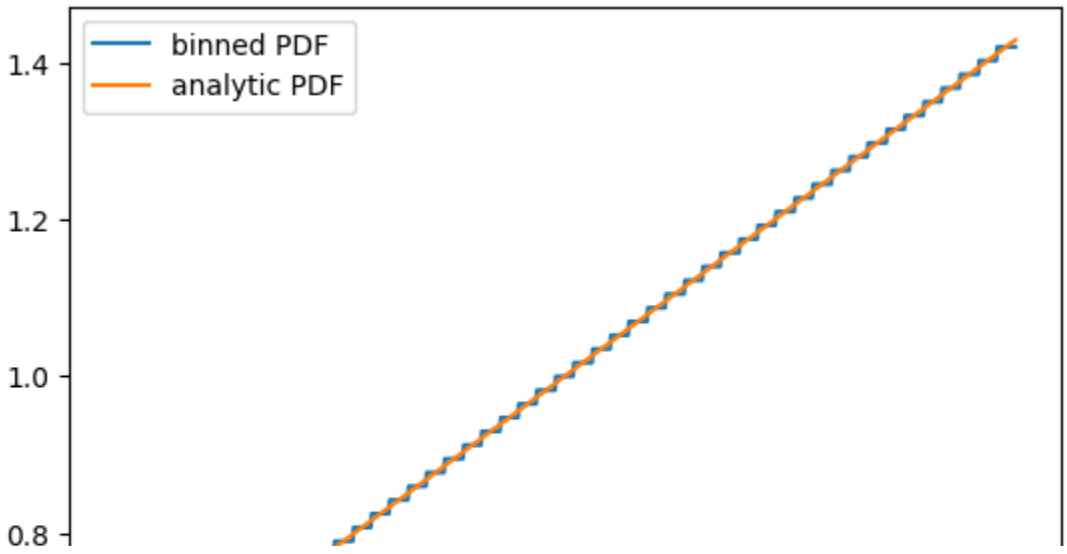
```
class Histogram:
    """
    Histogram for binned sampling.
    """

    def __init__(self, f, xmin, xmax, bins=50):
        """
        Initialize the histogram.

        f:     function to sample, should be callable, `f(x)`.
        xmin:  minimum x value.
        xman:  maximum x value.
        nbins: number of bins.
        """
        # Store the x minimum and maximum. Not necessary, but useful to
        # keep track of
```

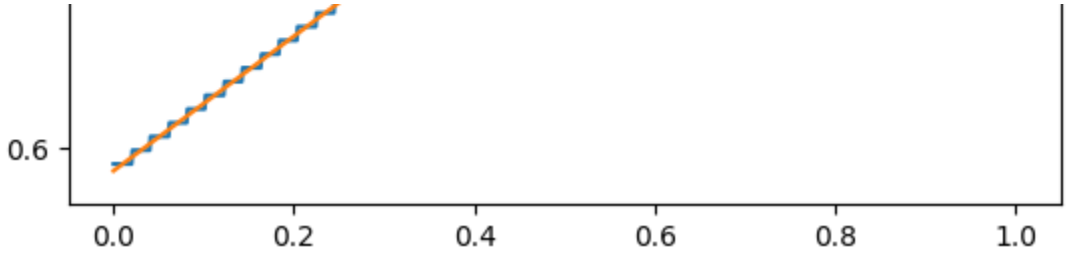
```
    # Keep track of.
    self.xmin = xmin
    self.xmax = xmax

    # Define the histogram edges, use `numpy.linspace`.
    self.edges = np.linspace(xmin, xmax, bins)

    # Define the PDF and CDF.
    self.pdf = []
    self.cdf = []
    pdf_sum = 0
    for i, xmax in enumerate(self.edges[1:]):
        # Calculate the PDF and append.
        xmin = self.edges[i]
        pdf = f((xmin + xmax) / 2)
        self.pdf += [pdf]

        # Calculate the CDF and append.
        pdf_sum += pdf
        self.cdf += [pdf_sum]

    # We store the normalization for the PDF. This is just `pdf_sum` times
    # bin width.
    self.norm = (self.edges[1] - self.edges[0]) * pdf_sum

    # The PDF is not yet a PDF, so we normalize it.
    self.pdf = [bin / self.norm for bin in self.pdf]

    # The CDF is also not yet a CDF, so we normalize it.
    self.cdf = [bin / pdf_sum for bin in self.cdf]

    # We now turn the CDF into edges for the inverse CDF, by prepending
    # the CDF by 0, since the CDF must start at 0.
    self.edges_icdf = [0] + self.cdf

def bin(self, x, edges=None):
    """
    Return the bin for a given x.

    x:     value to find the bin for.
    edges: optionally, edges of the histogram. The default is to use the
           edges of the histogram. This allows us to use this method when
           finding the bin for the inverse CDF.
    """
    # If no edges are provided, default to `self.edges`.
    if edges == None:
        edges = self.edges

    # Loop over the edges. Skip the
    for bin, edge in enumerate(edges[1:]):
        # Return if `x` is less than the edge.
        if x < edge:
            # For underflow, just return the first bin.
            return bin
```

```
            return bin

        # Return an overflow, just return the last bin.
        return bin

    def bin_icdf(self, r):
        """
        Return the bin from the inverse CDF.
        """
        # Set the `edges` argument to `self.edges_icdf` and use the `bin`
        # method.
        return self.bin(r, self.edges_icdf)
```

We would now like to test our histogram class. Let's use the function `f` from our linear sampler for the range 0 to 1, $m = 3$, and $b = 2$.

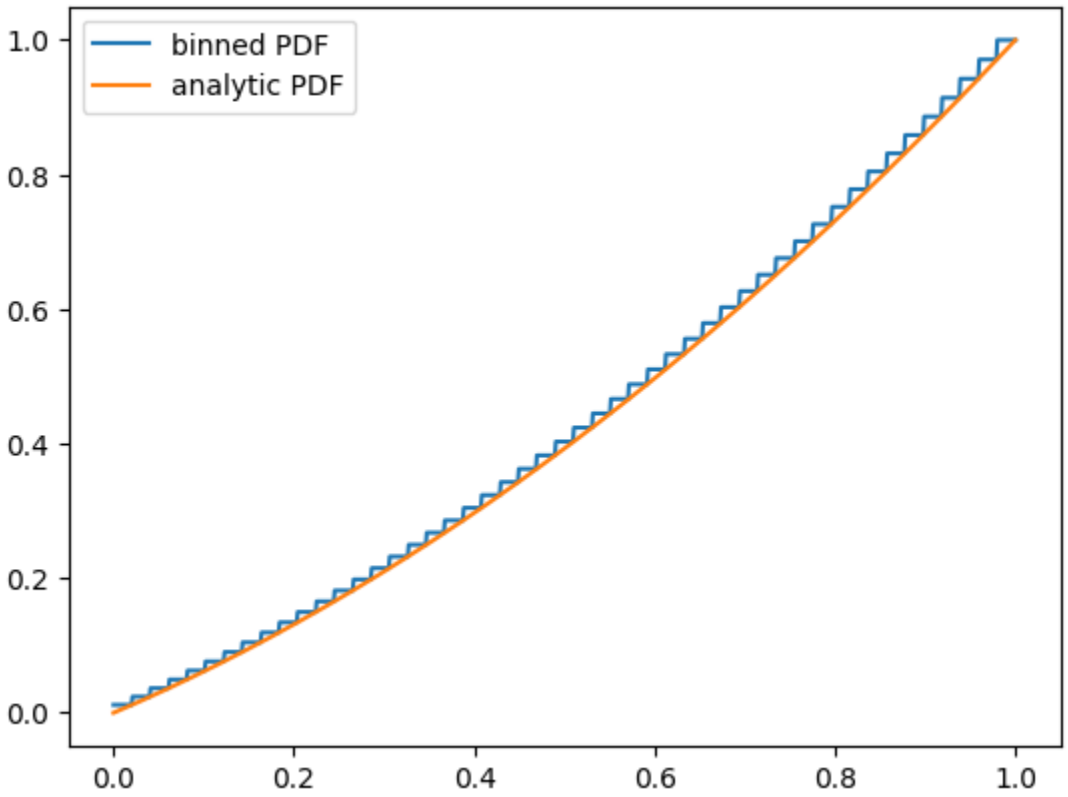
```
# Create the linear sampler for its function and integral.
line = SampleLinear(rng, 0, 1, 3, 2)

# Create a histogram for the function of this sampler.
hist = Histogram(line.f, line.xmin, line.xmax)

# Create a figure.
fig, ax = plt.subplots()

# Plot the binned PDF.
xs = np.linspace(hist.xmin, hist.xmax, 1000)
bys = [hist.pdf[hist.bin(x)] for x in xs]
ax.plot(xs, bys, label="binned PDF")

# Plot the analytic PDF. We need to make sure to normalize the integral here.
ays = [line.f(x) / line.area for x in xs]
ax.plot(xs, ays, label="analytic PDF")

# Create the legend.
ax.legend();
```





Test the CDF makes sense as well.

```
# Create a figure.
fig, ax = plt.subplots()

# Plot the binned CDF.
xs = np.linspace(hist.xmin, hist.xmax, 1000)
bys = [hist.cdf[hist.bin(x)] for x in xs]
ax.plot(xs, bys, label="binned PDF")

# Plot the analytic CDF.
ays = [line.F(x) / line.area for x in xs]
ax.plot(xs, ays, label="analytic PDF")

# Create the legend.
ax.legend();
```



Example: binned sampling

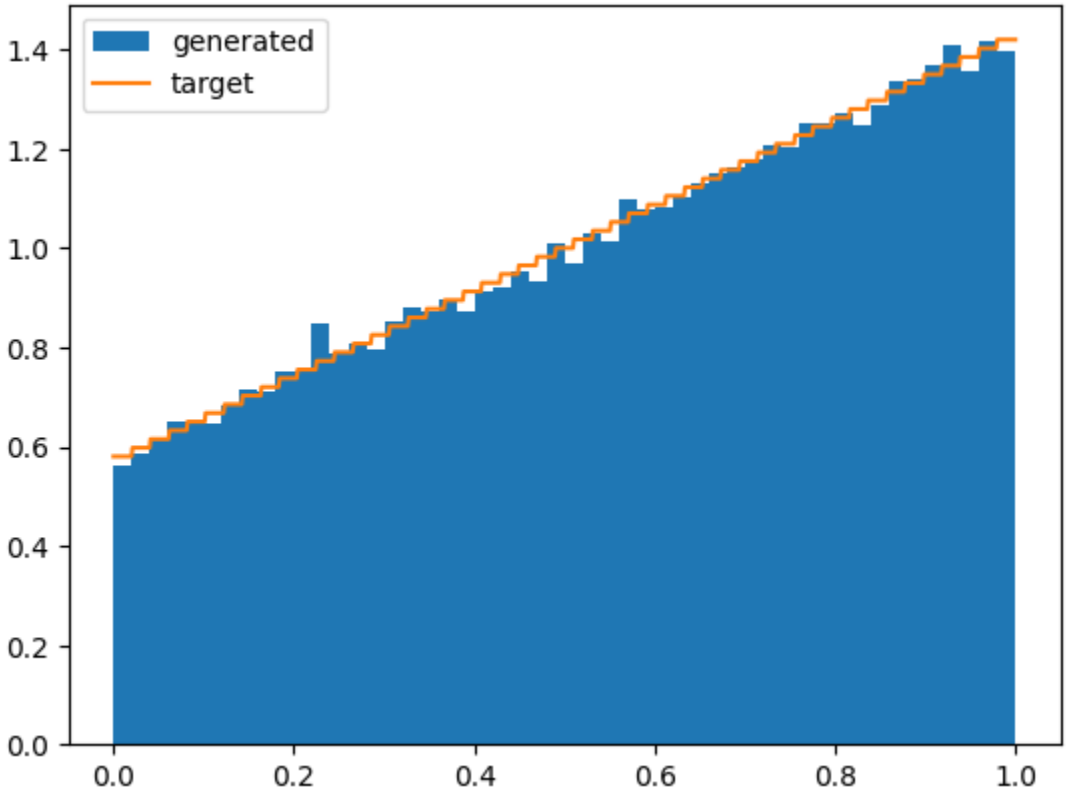
Create a binned sampler given the code skeleton below.

```
class SampleBinned:
    """
    Sampler using binned sampling.
    """

    def __init__(self, rng, hist):
        """
        Initialize the sampler.

        rng:  uniform random number generator, should have method `uniform()`.
        hist: histogram to sample from.
        """
        # Store the RNG, histogram, xmin, and xmax.
        self.rng = rng
        self.hist = hist
        self.xmin = hist.xmin
        self.xmax = hist.xmax

        # Set the area for `plot_sampler`. Since the binned PDF is normalized,
        # this is just 1.
        self.area = 1

    def f(self, x):
        """
        Return the sampled function. This is the binned PDF being sampled.
        Needed for `plot_sampler`.

        x: value to calculate f(x) for.
        """
        # Return 0 if outside the range.
        if x < self.xmin or x > self.xmax:
            return 0.0
        # Return the binned PDF otherwise.
        return self.hist.pdf[self.hist.bin(x)]

    def __call__(self):
        """
        Return the sampled value.
        """
        # Sample a uniform random number.
        r = self.rng.uniform()

        # Get the bin from the inverted CDF.
        i = hist.bin_icdf(r)

        # Get the edges for this bin.
        xmin = hist.edges[i]
        xmax = hist.edges[i + 1]

        # Uniformly sample between these values and return.
```

```
        r = self.rng.uniform()
        return r * (xmax - xmin) + xmin
```

Now that we have sampler, let's check that it generates what we want.

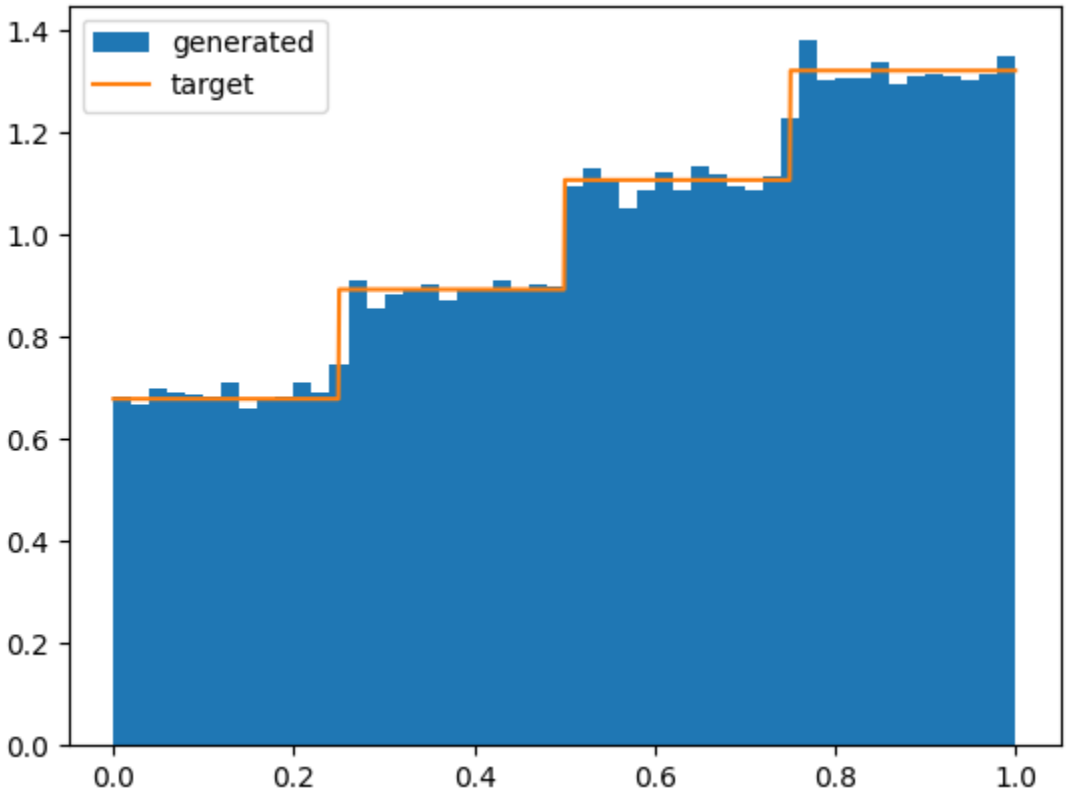
```
# Create the linear sampler for its function and integral.
line = SampleLinear(rng, 0, 1, 3, 2)

# Create the histogram.
hist = Histogram(line.f, line.xmin, line.xmax)

# Create the sampler.
sampler = SampleBinned(rng, hist)

# Plot the comparison.
plot_sampler(sampler);
```



Given the number of bins we have, and the number of points we are sampling, we don't see the discrete nature of the PDF. Make the comparison again, but now using only 5 bins in the discrete PDF.

```
# Create the linear sampler for its function and integral.
line = SampleLinear(rng, 0, 1, 3, 2)

# Create the histogram.
hist = Histogram(line.f, line.xmin, line.xmax, bins=5)

# Create the sampler.
```

```
sampler = SampleBinned(rng, hist)

# Plot the comparison.
plot_sampler(sampler);
```



Accept-or-Reject Sampling

Both the analytic and binned sampling use a fixed number of uniform random numbers per random number generated according to the target distribution. With analytic sampling, we require just one uniform random number which is then transformed to the target distribution. For binned sampling, we generate one uniform random number to select the bin, and another to select the location within that bin. For accept-or-reject (AOR) sampling, an indeterminate number of uniform random numbers may need to be sampled before the random number from the target distribution is obtained. This is the main short-coming of AOR. However, the simplicity and generality of AOR make it a go-to sampling method in MC.

For AOR to work, the target function $f(x)$ must satify two conditions.

1. The sampling of $f(x)$ is bounded, where over this range $f(x)$ is positive.

$$f(x) \geq 0 \text{ for } x_{\min} < x < x_{\max}$$

2. Within the range $x_{\min}$ to $x_{\max}$ $f(x)$ is finite.

The idea behind AOR is then as follows.

1. Find the maximum of $f(x)$ within the sampling range, $f_{\max}$.

2. Generate a uniform random number R_1 and map this to the range $x_{\min}$ to $x_{\max}$.

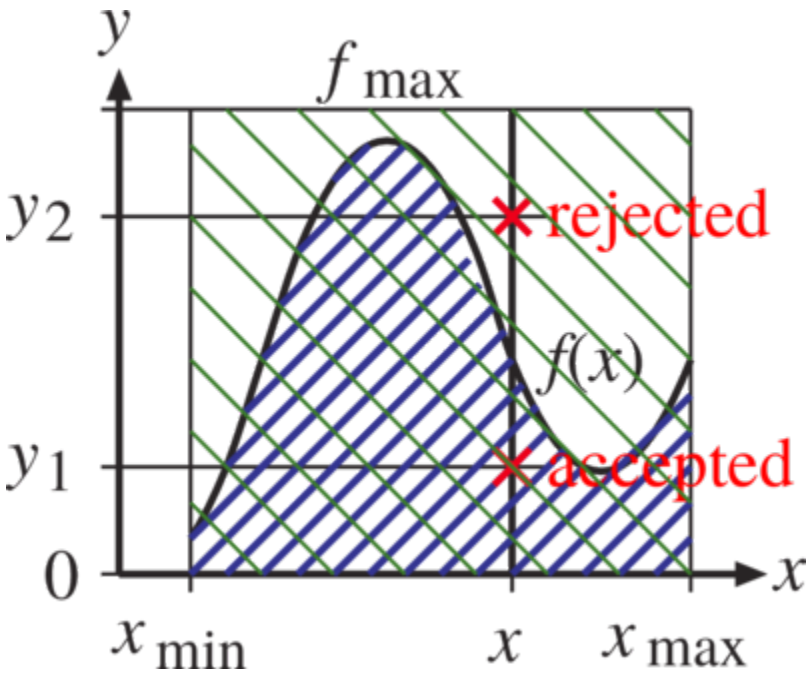
$$x = x_{\min} + R_1(x_{\max} - x_{\min})$$

3. Select a uniform random number R_2 and map this to the range 0 to $f_{\max}$.

$$y = R_2 f_{\max}$$

4. If $y > f(x)$ then reject the point and return to (2), otherwise accept the point and return the value x.

The following figure illustrates this process.



In this example, for a given x two possible y samplings are shown, y_1 and y_2. Here, y_1 falls below $f(x)$ so it is accepted, while y_2 falls above $f(x)$ so it is rejected. Note that this method can be easily expanded to n dimensions.

Exercise: general AOR

The beauty of AOR is that all we really need to know is how to evaluate the function, and the maximum of the function over the sampling range. That's it. Let us create a general class that performs AOR.

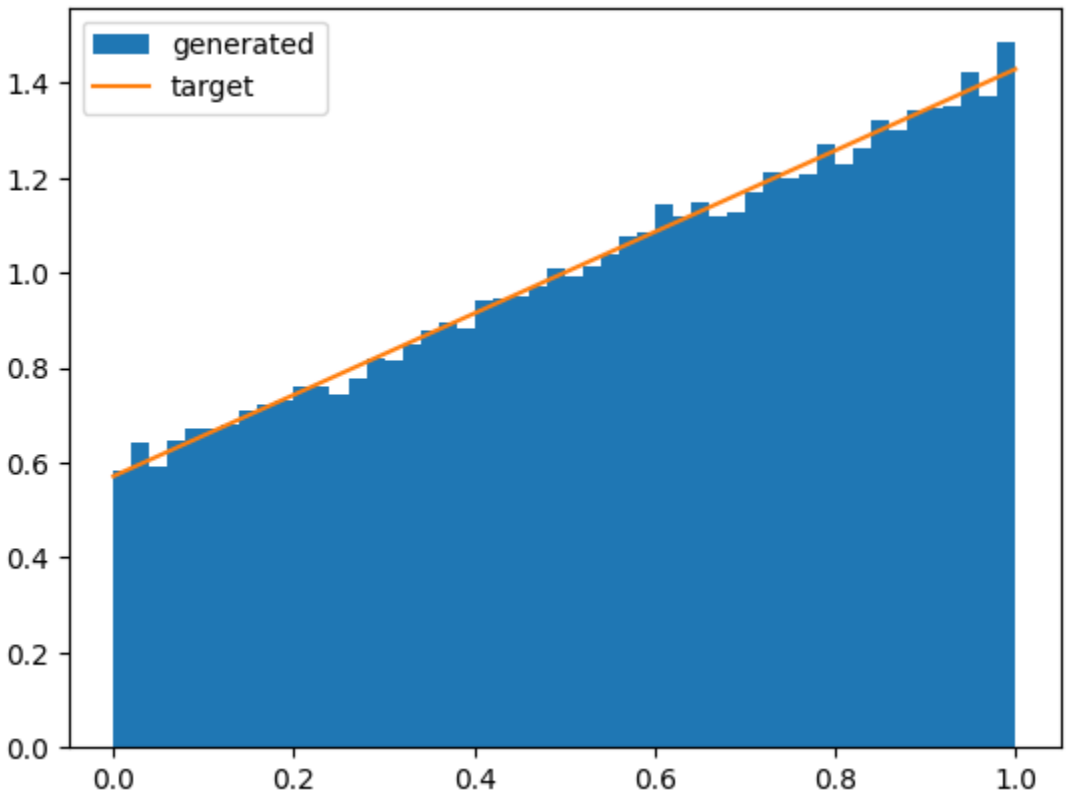
```
class SampleAOR:
    """
    Class to perform accept-or-reject sampling.
    """

    def __init__(self, rng, xmin, xmax, f, fmax=None, grid=100):
        """
        Initialize the sampler.

        rng:    uniform random number generator, should have method `uniform()`
```

```
    rng:    uniform random number generator, should have method `uniform()`.
    xmin:   minimum x value.
    xman:   maximum x value.
    f:      function to sample, should be callable, `f(x)`.
    fmax:   maximum of f(x), if not provided, then found numerically.
    grid:   number of points to numerically find maximum of f(x).
    """
    # Store the necessary members.
    self.rng = rng
    self.xmin = xmin
    self.xmax = xmax
    self.f = f

    # It is useful to track the efficiency of the generator. Store the
    # number of accept and reject attempts.
    self.accept = 0
    self.reject = 0

    # Find the maximum for f(x) with a simple grid search if not supplied.
    if fmax == None:
        self.fmax = 0
        for x in np.linspace(xmin, xmax, grid):
            self.fmax = max(self.fmax, f(x))
        # Add 5% head room on this.
        self.fmax *= 1.05
    else:
        self.fmax = fmax

def e(self):
    """
    Return the current efficiency of the sampler.
    """
    # This is defined as n_accept/n_total.
    return self.accept / (self.accept + self.reject)

def __call__(self, nmax=10000):
    """
    Sample from the target distribution.

    nmax: only sample this many times before giving up.
    """
    # Loop over the maximum number of attempts.
    for i in range(0, nmax):
        # Generate the two necessary uniform random numbers.
        r1 = self.rng.uniform()
        r2 = self.rng.uniform()
        # Transform `r1` to an x.
        x = self.xmin + r1 * (self.xmax - self.xmin)
        # Transform `r2` to y.
        y = r2 * self.fmax
        # Check if y < f(x).
        if y < self.f(x):
            self.accept += 1
            return x
```

```
            return x
        else:
            self.reject += 1
```

Now, let's see how well this works. Let us first try sampling the linear distribution from before. To make the comparison with `plot_sampler` we need the sampler object to have the member `area`, which is the integral of the function over the sampling range. This is used to normalize the generated distribution to the PDF, but is not actually required for the sampling method. Later on, we will see in the MC Integration how we can simultaneously calculate the integral while sampling. For now, we can simply set the area of the sampler object to that of the line sampler class.

```
# Create the linear sampler for its function and integral.
line = SampleLinear(rng, 0, 1, 3, 2)

# Create the sampler.
sampler = SampleAOR(rng, line.xmin, line.xmax, line.f)

# Set the area of the `sampler` to that of the `line`.
sampler.area = line.area

# Plot the comparison.
plot_sampler(sampler);
```



What is the efficiency of this sampler?

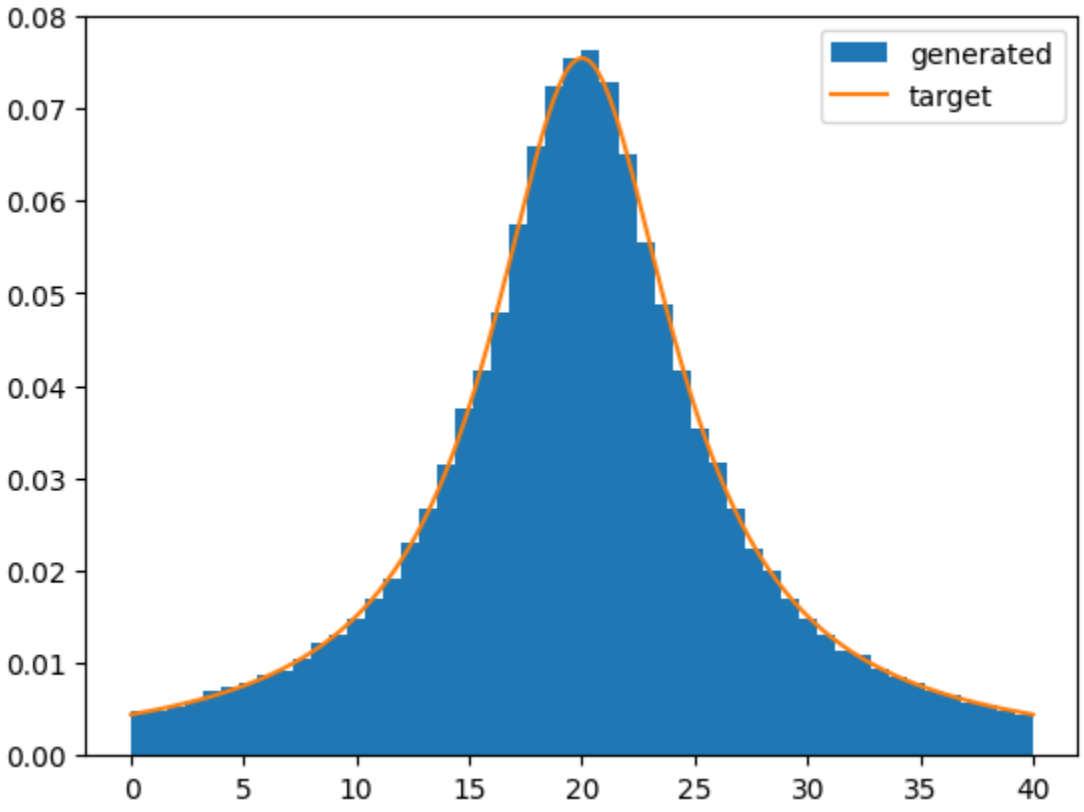
```
# Use the `e()` method of the `SampleAOR` class.
```

```
print(sampler.e())
```

```
0.6689276420969543
```

Can we sample a more complicated function? Try applying this sampler to the Cauchy distribution that we generated before.

```
# Create the Cauchy sampler for its function and integral.
cauchy = SampleCauchy(rng, 0, 40, 20, 5)

# Create the sampler.
sampler = SampleAOR(rng, cauchy.xmin, cauchy.xmax, cauchy.f)

# Set the area of the `sampler` to that of the `cauchy`.
sampler.area = cauchy.area

# Plot the comparison.
plot_sampler(sampler);
```



What is the efficiency for this sampling? Is this better or worse than for the line?

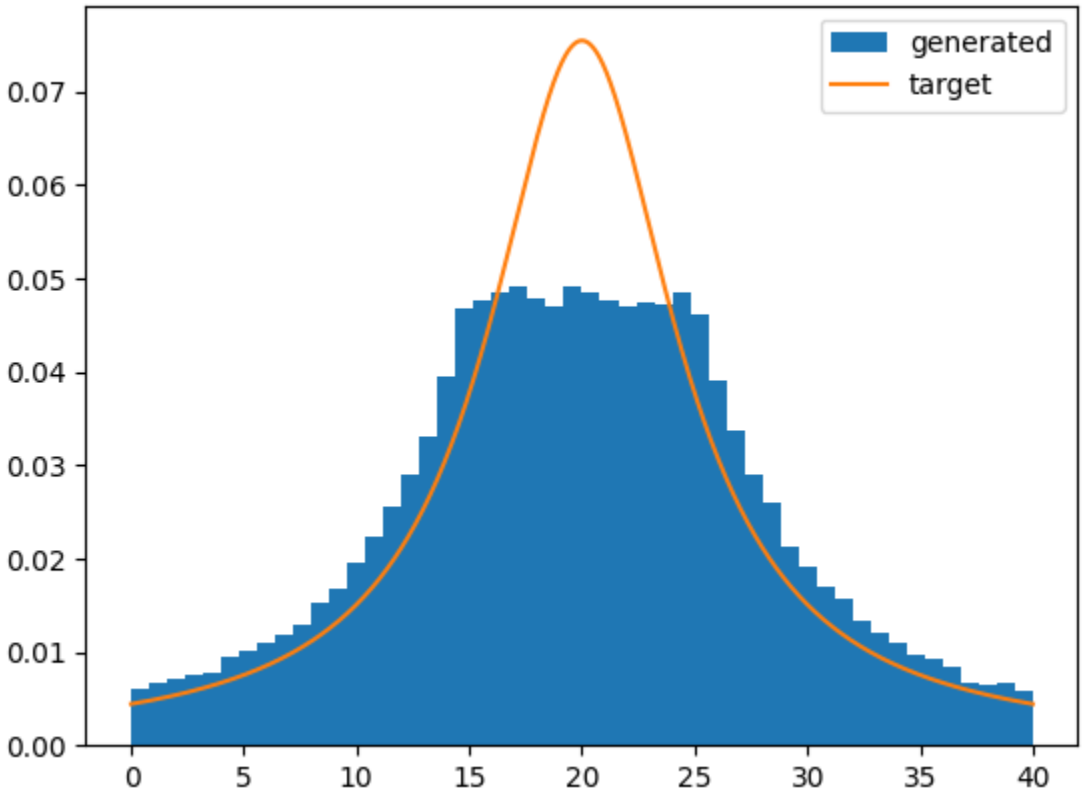
```
# Use the `e()` method of the `SampleAOR` class.
print(sampler.e())

# We get a value that is less than for the line sampling.
```

```
0.31542258741148455
```

Hopefully we see that this sampling is less efficient. We throw away around 70% of the points we sample.

Finally, let's see what happens if we don't choose our $f_{\max}$ correctly. We can do this by explicitly setting `fmax` to $f(x_0)$, which will be half our maximum for this particular function.

```
# Create the Cauchy sampler for its function and integral.
cauchy = SampleCauchy(rng, 0, 40, 20, 5)

# Incorrectly set an `fmax` that is half f(x0).
fmax = cauchy.f(cauchy.x0) / 2

# Create the sampler.
sampler = SampleAOR(rng, cauchy.xmin, cauchy.xmax, cauchy.f, fmax)

# Set the area of the `sampler` to that of the `cauchy`.
sampler.area = cauchy.area

# Plot the comparison.
plot_sampler(sampler);
```



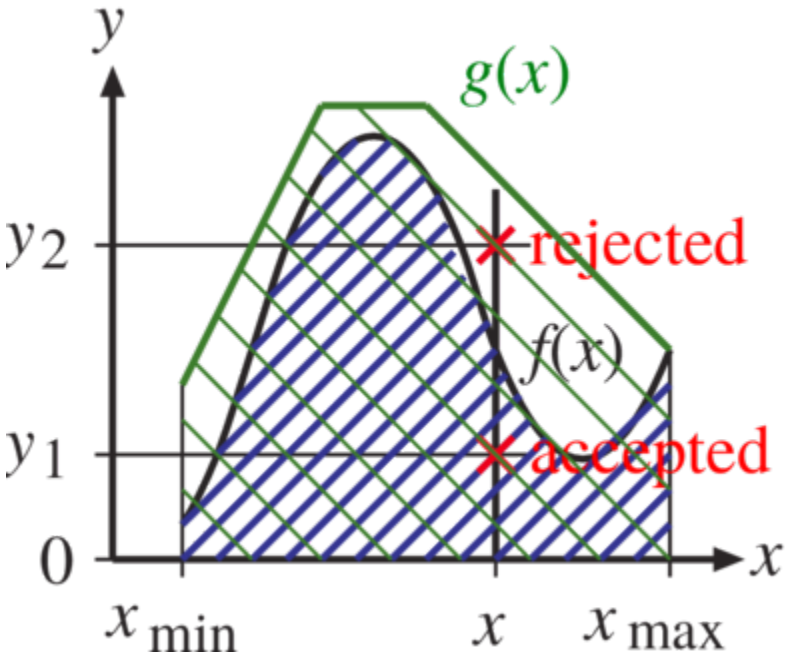
We should see that distribution does not look correct. This large enough value of $f_{\max}$ is critical for the AOR method. The maximum must truly be the maximum for that function over the sampling range, otherwise sculpting effects of the distribution will occur.

Importance Sampling

Importance sampling is still accept-or-reject, we just choose a distribution that better bounds the function $f(x)$. Spefically, we choose a function $g(x)$ that we can sample easily, whether analytic or some other method, that is already greater than $f(x)$.

$$g(x) > f(x) \text{ for } x_{\min} < x < x_{\max}$$

Using the same example given at the start of the Accept-or-Reject Sampling, we can imagine a bounding function like the green one given in the plot below.



1. Find $g(x)$ which bounds $f(x)$ within the sampling range.
2. Generate x distributed according to $g(x)$.
3. Select a uniform random number R and map this to the range 0 to $g(x)$.

$$y = Rg(x)$$

4. If $y > f(x)$ then reject the point and return to (2), otherwise accept the point and return the value x.

Exercise: general importance sampler

Let us try to implement a general importance sampler in the skeleton given below. The result should look relatively similar to the AOB sampler.

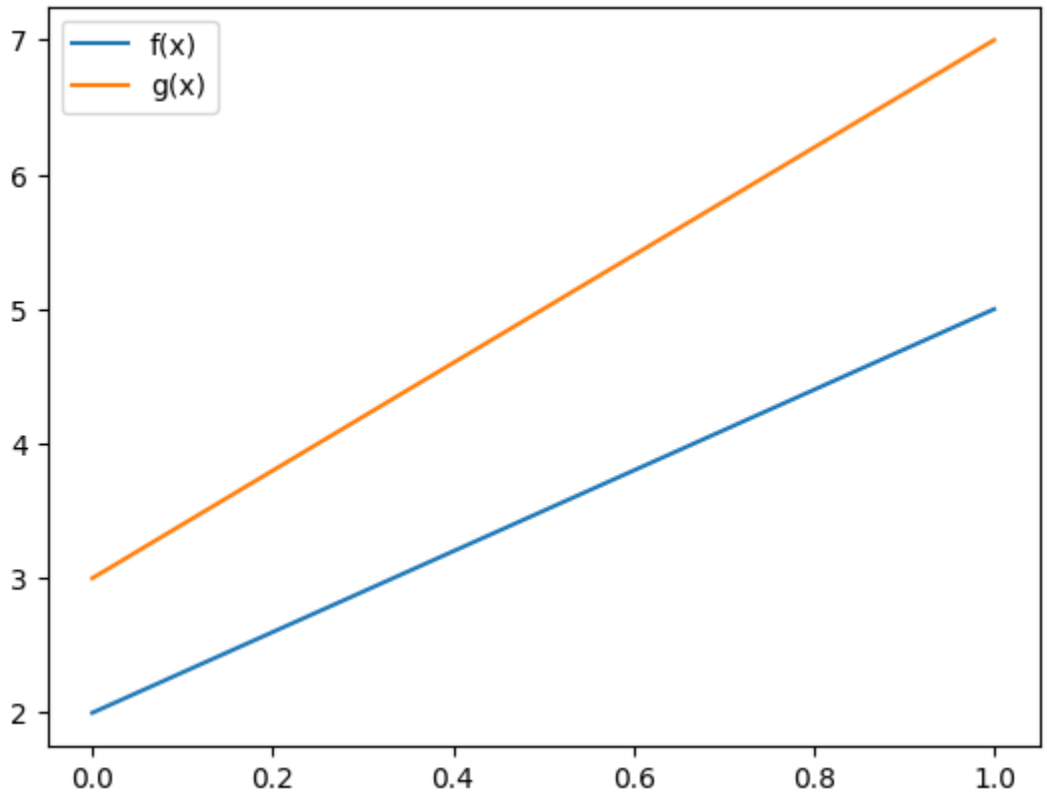
```
class SampleImportance:
    """
    Class to perform importance sampling.
    """
```

```
def __init__(self, rng, xmin, xmax, f, g, grng):
    """
    Initialize the sampler.

    rng:  uniform random number generator, with method `uniform()`.
    xmin: minimum x value.
    xman: maximum x value.
    f:    function to sample, should be callable, `f(x)`.
    g:    function that bounds f(x), should be callable `g(x)`.
    grng: samples `g(x)`, should be callable `gnrg()`.
    """
    # Store the necessary members.
    self.rng = rng
    self.xmin = xmin
    self.xmax = xmax
    self.f = f
    self.g = g
    self.grng = grng

    # It is useful to track the efficiency of the generator. Store the
    # number of accept and reject attempts.
    self.accept = 0
    self.reject = 0

def e(self):
    """
    Return the current efficiency of the sampler.
    """
    # This is defined as n_accept/n_total.
    return self.accept / (self.accept + self.reject)

def __call__(self, nmax=10000):
    """
    Sample from the target distribution.

    nmax: only sample this many times before giving up.
    """
    # Loop over the maximum number of attempts.
    for i in range(0, nmax):
        # Generate `x` from the g(x) sampler.
        x = self.grng()
        # Generate a uniform random number.
        r = self.rng.uniform()
        # Transform `r` to y.
        y = r * self.g(x)
        # Check if y < f(x).
        if y < self.f(x):
            self.accept += 1
            return x
        else:
            self.reject += 1
```

Exercise: even more linear sampling

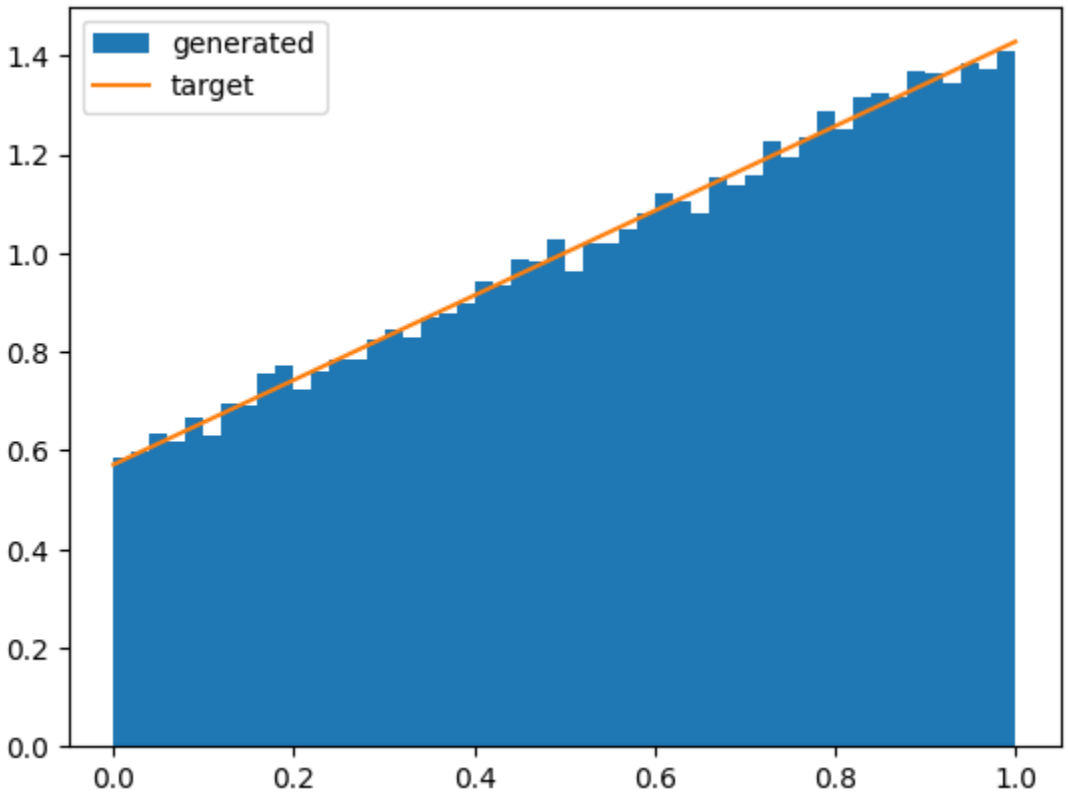
While there are any number of interesting functions we could try to importance sample, let us return to a linear function to allow us to cross-check that our importance sampler is working correctly. Let us target the distribution $m = 3$ and $b = 2$ between 0 and 1. We can choose an oversampling function that is implemented with the `SampleLinear` class to provide our $g(x)$. Here, let us choose the oversampling function with $m = 4$ and $b = 3$. First let us just plot these two functions. We can use the `SampleLinear` class, which is complete overkill.

```
# Create the two samplers.
f_sampler = SampleLinear(rng, 0, 1, 3, 2)
g_sampler = SampleLinear(rng, 0, 1, 4, 3)

# Create our x points.
xs = np.linspace(0, 1)

# Create our plot.
fig, ax = plt.subplots()

# Plot the two lines.
ax.plot(xs, [f_sampler.f(x) for x in xs], label="f(x)")
ax.plot(xs, [g_sampler.f(x) for x in xs], label="g(x)")

# Draw a legend.
ax.legend();
```



Clearly $g(x)$ bounds $f(x)$. Now, let us set up our sampler and compare the generated distribution to the expected.

```
# Create the f(x) linear sampler for its function and integral.
line = SampleLinear(rng, 0, 1, 3, 2)

# Create the g(x) linear sampler.
grng = SampleLinear(rng, 0, 1, 4, 3)

# Create the sampler.
sampler = SampleImportance(rng, line.xmin, line.xmax, line.f, grng.f, grng)

# Set the area of the `sampler` to that of the `line`.
sampler.area = line.area

# Plot the comparison.
plot_sampler(sampler);
```



What is the efficiency of this sampling? Can you calculate what this efficiency is exactly?

```
# Use the `e()` method of the `SampleImportance` class.
print(f"efficiency = {sampler.e():.2f}")

# The exact efficiency should just be the integral of f(x) over g(x).
print(f"exact efficiency = {line.area/grng.area:.2f}")
```

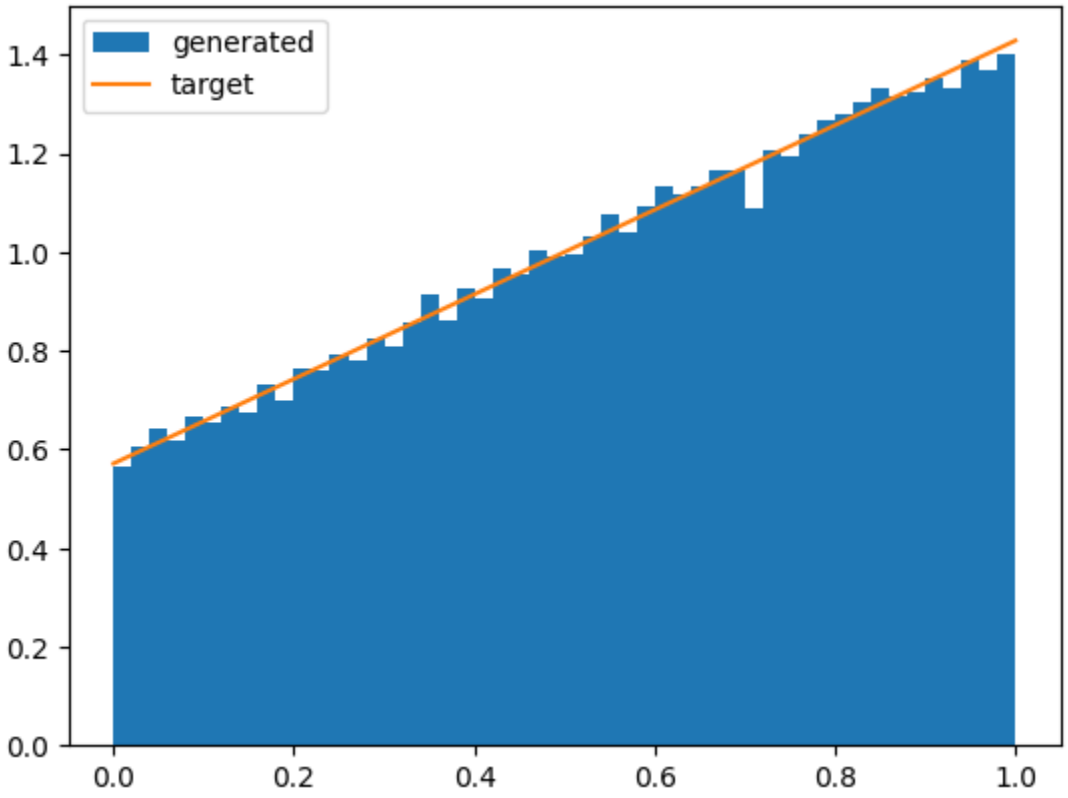
```
efficiency = 0.70
exact efficiency = 0.70
```

```
exact efficiency = 0.70
```

Could we make this efficiency 100%? Create an importance sampler that does this. Check the distribution and efficiency.

```
# Create the f(x) linear sampler for its function and integral.
line = SampleLinear(rng, 0, 1, 3, 2)

# If g(x) = f(x), we then we are completely efficient.
# Create the g(x) linear sampler.
grng = SampleLinear(rng, 0, 1, 3, 2)

# Create the sampler.
sampler = SampleImportance(rng, line.xmin, line.xmax, line.f, grng.f, grng)

# Set the area of the `sampler` to that of the `line`.
sampler.area = line.area

# Plot the comparison.
plot_sampler(sampler)

# Use the `e()` method of the `SampleImportance` class.
print(f"efficiency = {sampler.e()}")
```

```
efficiency = 1.0
```



Exercise: Lund fragmentation function

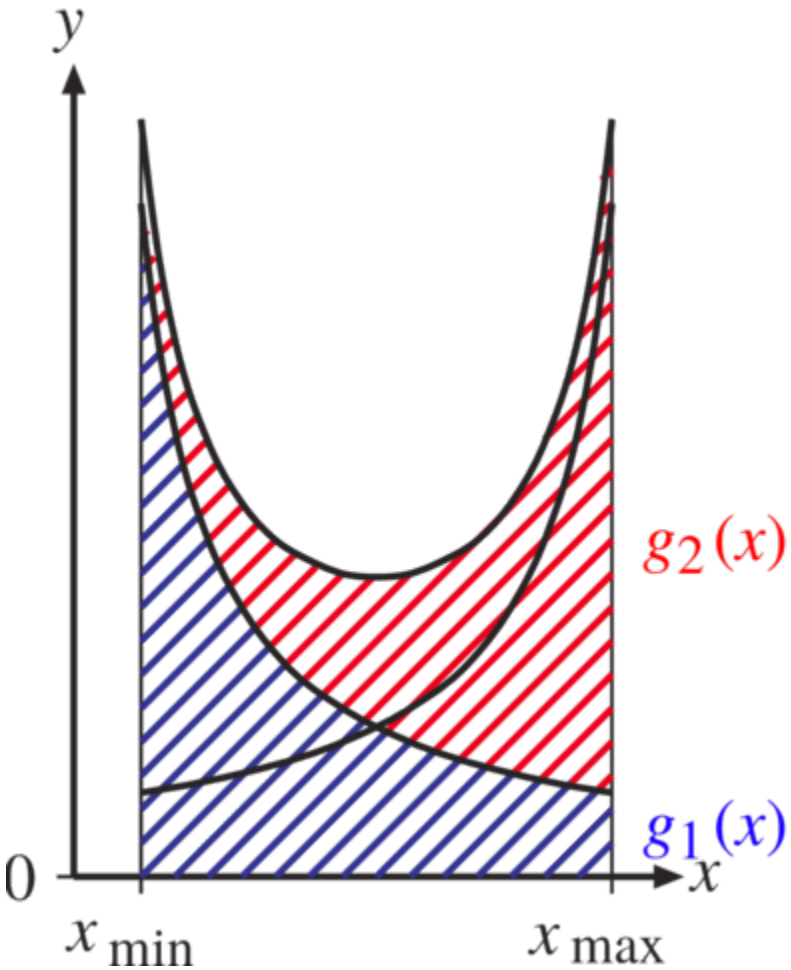
The example above is perhaps not particularly satisfying, so you are encouraged to try more complex functions. In the [hadronization.ipynb] notebook, we will introduce the Lund fragmentation function,

$$f(x) = \frac{(1-x)^a}{x}\exp\left(-b\frac{m^2}{x}\right),$$

where a, b, and m are parameters, and x runs from 0 to 1. It turns out that this function is much harder to sample, and in that notebook we perform a relatively simple importance sampling of it.

Multichannel Sampling

Sometimes, we would like to perform importance sampling, but we don't have a single $g(x)$ which bounds $f(x)$. Consider the example below.



Here, the sum of the two functions $g_1(x)$ and $g_2(x)$ bounds $f(x)$. The algorithm is as follows.

1. Select $g_i(x)$ from n channels with relative probability.

$$\frac{G_i(x_{\max}) - G_i(x_{\min})}{\sum_{j=1}^{n} G_j(x_{\max}) - G_j(x_{\min}}$$

2. Generate x distributed according to $g_i(x)$.

3. Select a uniform random number $\mathcal{K}$ and map this to the range 0 to $g_i(x)$.

$$y = \mathcal{R} g_i(x)$$

4. If $y > f(x)$ then reject the point and return to (2), otherwise accept the point and return the value x.

Exercise: general multichannel

Just as for the importance sampling, let us first implement a general multichannel sampler. When we implemented the importance sampling, we factorized the $g(x)$ function definition from the $g(x)$ sampler. Here, we also need the integral of $g(x)$, i.e., $G(x)$. This can be obtained numerically, but for simplicity in this example, let us limit ourselves to analytic sampling. This means that we can just require a list of samplers deriving from the `SampleAnalytic` class, where all these functions are already defined.

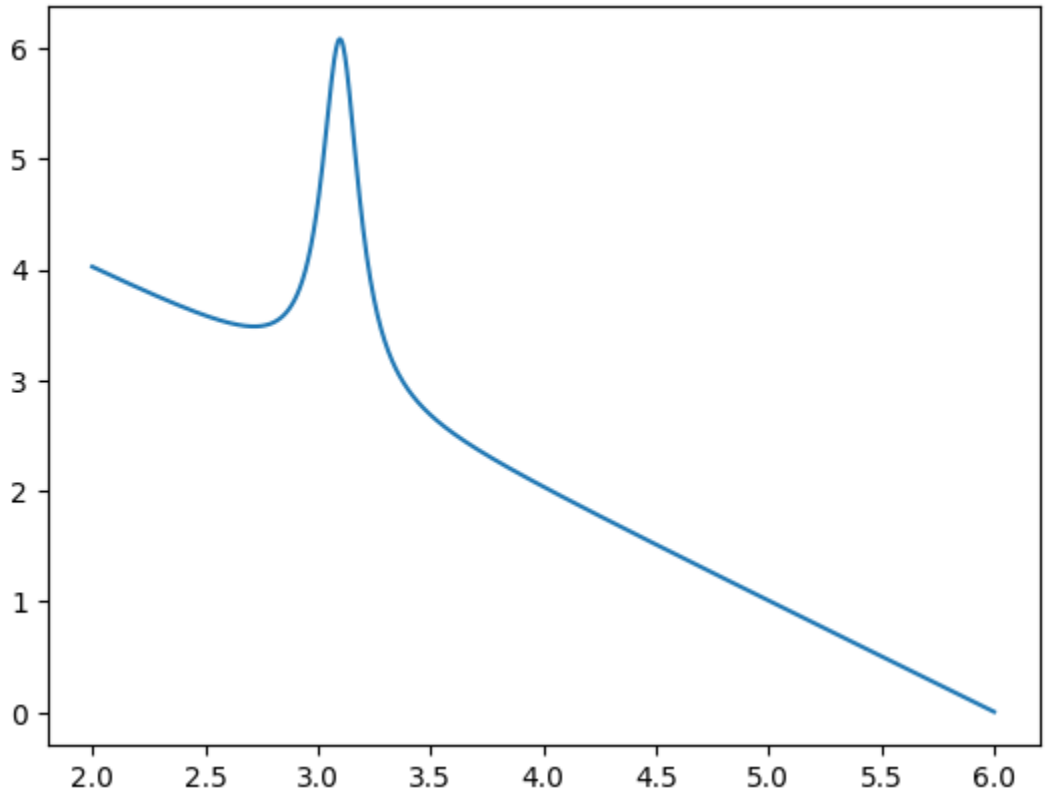
```
class SampleMulti:
    """
    Class to perform multichannel sampling.
    """

    def __init__(self, rng, xmin, xmax, f, grngs):
        """
        Initialize the sampler.

        rng:   uniform random number generator, with method `uniform()`.
        xmin:  minimum x value.
        xman:  maximum x value.
        f:     function to sample, should be callable, `f(x)`.
        grngs: samplers for g_i(x) which summed bound f(x). Should be derived
               from `SampleAnalytic` class.
        """
        # Store the necessary members.
        self.rng = rng
        self.xmin = xmin
        self.xmax = xmax
        self.f = f
        self.grngs = grngs

        # Calculate the channel weights. This is the integral over the sampling
        # range per channel. We could also just use the `area` member.
        areas = [grng.F(xmax) - grng.F(xmin) for grng in grngs]
        # Next, we calculate the total sum.
        area_sum = sum(areas)
        # Finally, we construct `wgts` to be the running cumulative weight.
        # We can then sample a uniform random number, and loop over the weights
        # to select the channel. This is implemented in the `channel` method.
        wgt_sum = 0
        self.wgts = []
```

```
        self.wgts = []
        for area in areas:
            wgt_sum += area / area_sum
            self.wgts += [wgt_sum]

        # It is useful to track the efficiency of the generator. Store the
        # number of accept and reject attempts.
        self.accept = 0
        self.reject = 0

    def e(self):
        """
        Return the current efficiency of the sampler.
        """
        # This is defined as n_accept/n_total.
        return self.accept / (self.accept + self.reject)

    def channel(self):
        """
        Randomly sample one of the channels according to their
        probabilities.
        """
        r = self.rng.uniform()
        for i, wgt in enumerate(self.wgts):
            if r < wgt:
                return i
        return i

    def __call__(self, nmax=10000):
        """
        Sample from the target distribution.

        nmax: only sample this many times before giving up.
        """
        # Loop over the maximum number of attempts.
        for i in range(0, nmax):
            # Choose the channel.
            i = self.channel()
            # Generate `x` from the g(x) sampler.
            x = self.grngs[i]()
            # Generate a uniform random number.
            r = self.rng.uniform()
            # Transform `r` to y.
            y = r * self.grngs[i].f(x)
            # Check if y < f(x).
            if y < self.f(x):
                self.accept += 1
                return x
            else:
                self.reject += 1
```

Exercise: signal on background

Let us consider an example which does not fully capture the power of multichannel sampling, but is of high energy physics interest. We oftentimes will have resonance signals on continuum backgrounds like the following.

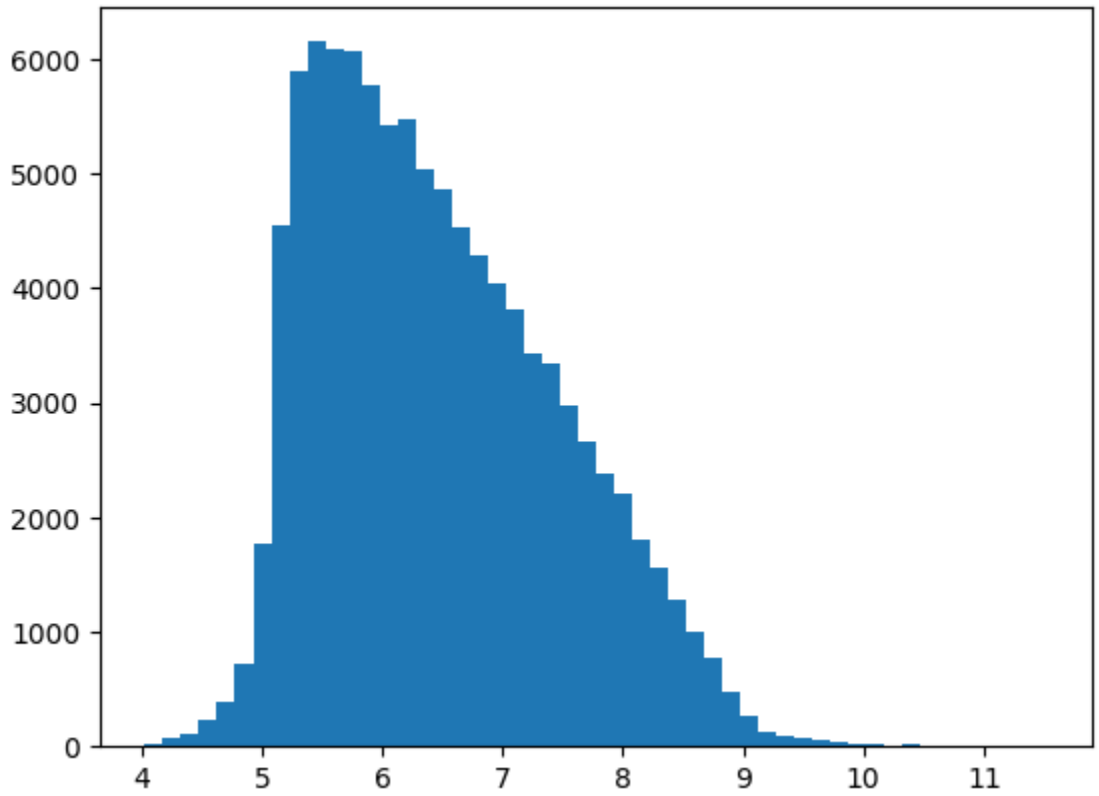
```
# Create our signal and background functions.
signal = SampleCauchy(rng, 2, 6, 3.1, 0.1)
background = SampleLinear(rng, signal.xmin, signal.xmax, -1, 6)

# Create our x points.
xs = np.linspace(signal.xmin, signal.xmax, 1000)

# Create our plot.
fig, ax = plt.subplots()

# Plot the two lines.
ax.plot(xs, [signal.f(x) + background.f(x) for x in xs]);
```



To sample this, we might be very tempted to just sample the signal distribution, x_1, and add this to a sample from the background disribution x_2. This will not work! Adding randomly sampled variables from PDFs is not equivalent to adding the PDFs themselves. Actually, this makes for an interesting distribution, but certainly not the one we want. Plot this distribution below.

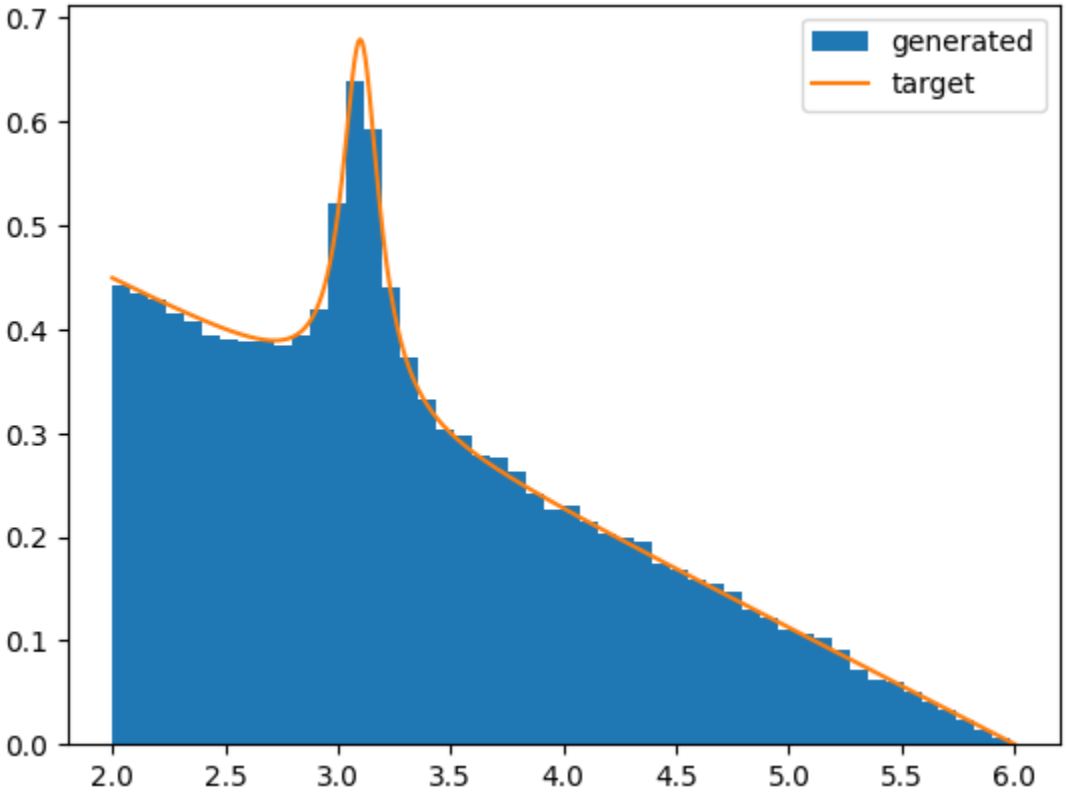
```
# Sample the distributions and add together.
rs = [signal() + background() for i in range(0, 100000)]
```

```
# Create the plot.
fig, ax = plt.subplots()

# Draw the histogram.
ax.hist(rs, bins=50);
```



Instead, we need to use multichannel sampling. Because we already have two functions that we can generate the distributions for, finding the summed $g_i(x)$ that bounds our $f(x)$ is straight forward, and also exact. Construct a multichannel sampler that does this and compare with the target distribution.

```
# Create our signal and background functions.
signal = SampleCauchy(rng, 2, 6, 3.1, 0.1)
background = SampleLinear(rng, signal.xmin, signal.xmax, -1, 6)

# Create a function that is the sum of the signal and background.
# We use a lambda function, which is an anonymous function (we don't
# use `def` to define it).
both = lambda x: signal.f(x) + background.f(x)

# Create the sampler.
sampler = SampleMulti(rng, signal.xmin, signal.xmax, both, [signal, background])

# Set the area of the `sampler` to that of the signal + background. This
# is not needed for the sampling, just the comparison.
sampler.area = signal.area + background.area

# Plot the comparison.
plot_sampler(sampler);
```



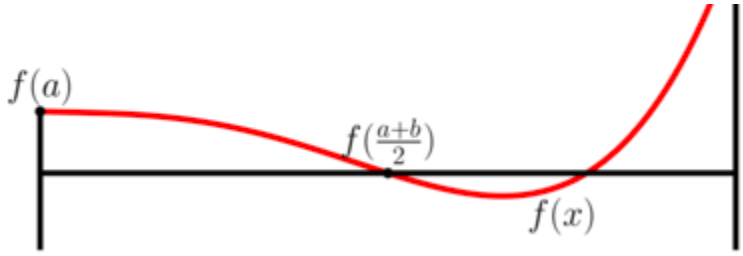
Quadrature Integration

Numerical integration is a critical tool for many scientific disciplines. Prior to computers, integrals that could not be calculated analytically were sometimes calculated physically. The function would be plotted on a piece of paper, the paper would then be cut along the line of the function, and the resulting paper was weighed. Dividing this weight by the weight of the paper before cutting would then give the relative integral. Computers make this process a little faster, but we need to make sure we use the right technique for the job.

One method of integration is called quadrature where we approximate the function we are trying to integrate with piecewise functions that we know the integrals for already. While there are any number of quadrature methods out there, we will only consider a few here.

Exercise: midpoint rule

Consider a function $f(x)$ and its integral over the range a to b. One of the simplest non-Monte Carlo methods for performing this integral is to evaluate the function at its midpoint between a and b, and then multiply this by $b - a$.



$$\int_a^b \mathrm{d}x\, f(x) \approx (b-a) f\left(\tfrac{a+b}{2}\right)$$

This method is called the midpoint rule. Note that the midpoint rule requires just a single function evaluation. If the function is computationally expensive to compute, it is important to reduce the number of function evaluations. Of course, the midpoint might not provide a good estimate of the function, so instead we can divide the integral into multiple peices and for each division evaluate the midpoint rule. This is the composite midpoint rule.

Implement the composite midpoint rule.

```
def integrate_midpoint(f, xmin, xmax, ndiv=1000):
    """
    Evaluate the definite integral of f(x) using the midpoint rule.

    f:    function to integrate, should be callable as `f(x)`.
    xmin: lower bound.
    xmax: upper bound.
    ndiv: number of divisions.
    """
    # Subdivide into division over the range of integration.
    xs = np.linspace(xmin, xmax, ndiv)
    # Calculate the division width.
    dx = xs[1] - xs[0]
    # Sum the integral for each midpoint.
    integral = 0
    for i, b in enumerate(xs[1:]):
        a = xs[i]
        integral += dx * f((a + b) / 2)
    return integral
```

Evaluate the integral of the Lund fragmentation function,

$$f(x) = \frac{(1-x)^a}{x} \exp\left(-b\frac{m^2}{x}\right)$$

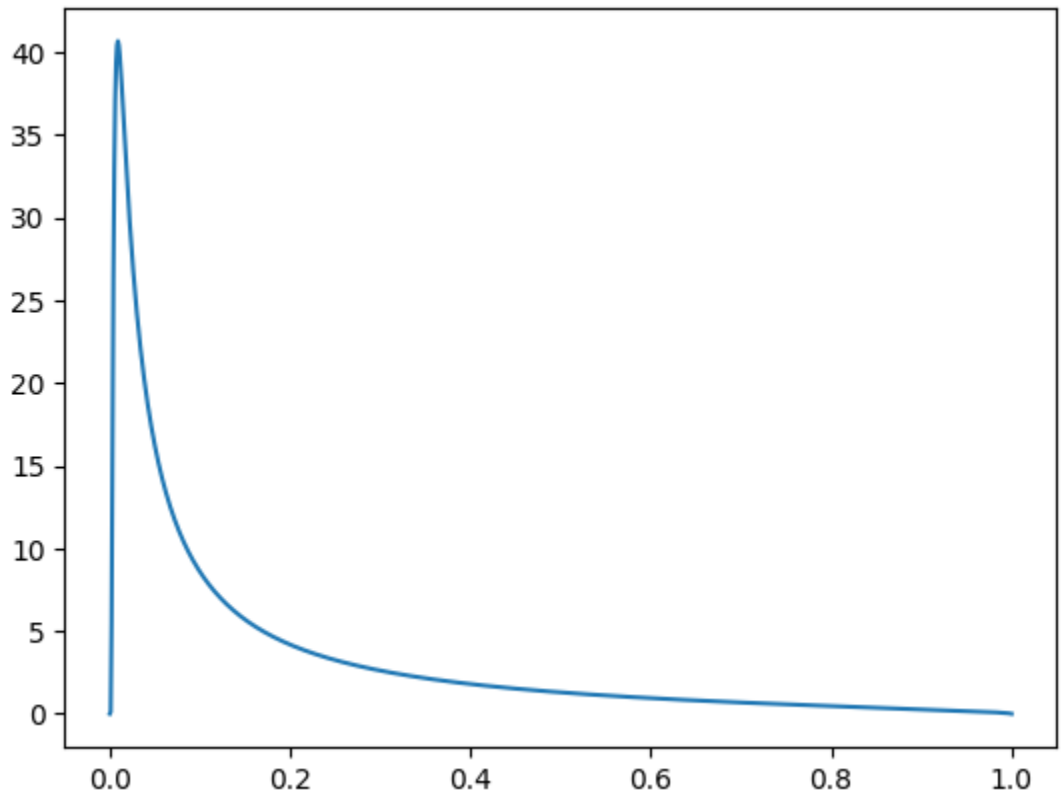
with $a = 0.6$, $b = 0.9$, and $m = 0.1$ over the interval 0 to 1 using the composite midpoint rule with 10 divisions. Just to clarify, the a and b of this function are parameters, and are not related to the a and b of the midpoint rule which define the integration limits. First, define the function.

```
def function lff(x, a, b, m):
```

```
    """
    Return the Lund fragmentation function.

    a: the power shape parameter.
    b: the exponential shape parameter.
    m: the transverse mass parameter.
    """
    # Return 0 if x is 0.
    if x == 0:
        return 0.0
    # Return the function.
    return ((1 - x) ** a / x) * math.exp(-(b * m**2) / x)
```

What does this function look like?

```
# Create the x values.
xs = np.linspace(0, 1, 1000)

# Define the parameters.
a, b, m = 0.6, 0.9, 0.1

# Calculate the function at these points.
ys = [function_lff(x, a, b, m) for x in xs]

# Create the plot and plot the curve.
fig, ax = plt.subplots()
ax.plot(xs, ys);
```



Next, integrate the function.

```
# The integrate method cannot pass arguments to f(x). So we use a lambda
# function here to pass the parameters that we want.
f = lambda x: function_lff(x, a, b, m)

# Perform the integration and print.
integral = integrate_midpoint(f, 0, 1, 10)
print(f"midpoint integral (n = 10) = {integral:.5e}")
```

```
midpoint integral (n = 10) = 3.11772e+00
```

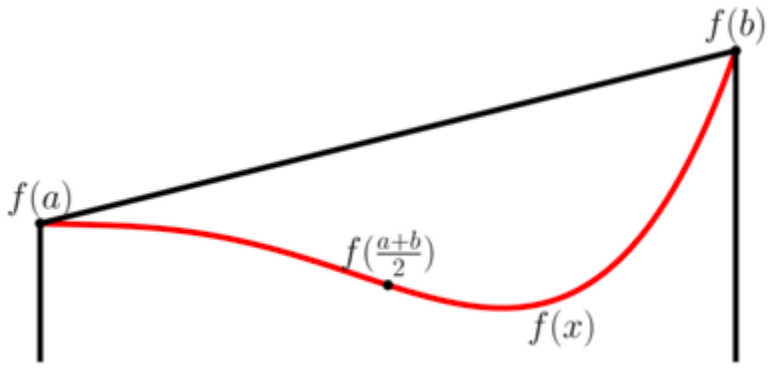
Compare with divisions of 10, 100, 1000, and 10000. Print to the integral to 6 significant digits. How many divisions are needed for convergence?

```
# The integrate method cannot pass arguments to f(x). So we use a lambda
# function here to pass the parameters that we want.
f = lambda x: function_lff(x, a, b, m)

# Perform the integration and print.
for ndiv in (10, 100, 1000, 10000):
    integral = integrate_midpoint(f, 0, 1, ndiv)
    print(f"midpoint integral (n = {ndiv:5d}) = {integral:.5e}")
```

```
midpoint integral (n =    10) = 3.11772e+00
midpoint integral (n =   100) = 3.53756e+00
midpoint integral (n =  1000) = 3.46835e+00
midpoint integral (n = 10000) = 3.46835e+00
```

Exercise: trapezoid rule

The midpoint rule approximates the function as a zero order polynomial, e.g., just a horizontal line, for which the integral is known. But, we can approximate $f(x)$ using other functions with simple analytic integrals, which may describe $f(x)$ better. With the trapezoidal rule, the function is approximated as a line from point a to point b, forming a trapezoid.



$$\int_a^b f(x)\,dx \approx \frac{b-a}{2}\left(f(a)+f(b)\right)$$

$$\int_a \mathrm{d}x\, f(x) \approx \frac{\tilde{b} - \tilde{a}}{2} \Big(f(a) + f(b) \Big)$$

Note that this method requires two function evaluations. Again, the integral can be divided into smaller intervals and the trapezoid rule can be performed for each division. Implement the trapezoid rule.

```
def integrate_trapezoid(f, xmin, xmax, ndiv=1000):
    """
    Evaluate the definite integral of f(x) using the trapezoid rule.

    f:    function to integrate, should be callable as `f(x)`.
    xmin: lower bound.
    xmax: upper bound.
    ndiv: number of divisions.
    """
    # Subdivide into division over the range of integration.
    xs = np.linspace(xmin, xmax, ndiv)
    # Calculate the division width.
    dx = xs[1] - xs[0]
    # Sum the integral for each trapezoid.
    integral = 0
    for i, b in enumerate(xs[1:]):
        a = xs[i]
        integral += dx / 2 * (f(a) + f(b))
    return integral
```

Calculate the integral for the Lund fragmentation function using the trapezoid rule for 10, 100, 1000, and 10000 divisions. How does this compare to the midpoint rule for the same number of divisions? Which of the two converges faster.

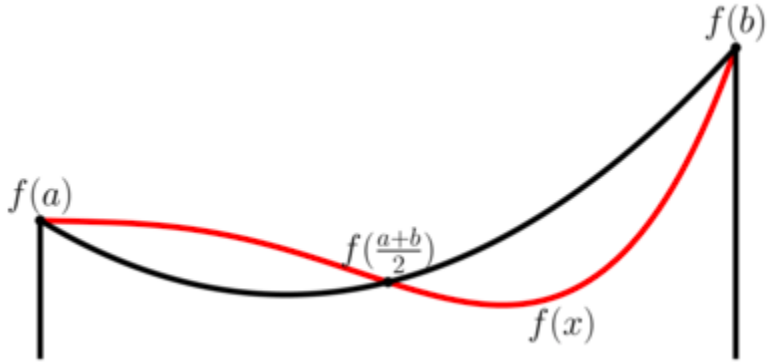
```
# The integrate method cannot pass arguments to f(x). So we use a lambda
# function here to pass the parameters that we want.
f = lambda x: function_lff(x, a, b, m)

# Perform the integration and print.
for ndiv in (10, 100, 1000, 10000):
    integral = integrate_midpoint(f, 0, 1, ndiv)
    print(f" midpoint integral (n = {ndiv:5d}) = {integral:.5e}")
    integral = integrate_trapezoid(f, 0, 1, ndiv)
    print(f"trapezoid integral (n = {ndiv:5d}) = {integral:.5e}")
```

```
 midpoint integral (n =    10) = 3.11772e+00
trapezoid integral (n =    10) = 1.99520e+00
 midpoint integral (n =   100) = 3.53756e+00
trapezoid integral (n =   100) = 3.41425e+00
 midpoint integral (n =  1000) = 3.46835e+00
trapezoid integral (n =  1000) = 3.46834e+00
 midpoint integral (n = 10000) = 3.46835e+00
trapezoid integral (n = 10000) = 3.46835e+00
```

Now calculate with 1000. How does this compare to the midpoint rule and the 10 division calculation?

Now calculate with 1000. How does this compare to the midpoint rule and the 10 division calculation?

We can use even higher order polynomials for the approximating function, which really is just used to interpolate $f(x)$. For Simpson's rule, a second order polynomial is used.



$$\int_a^b \mathrm{d}x\, f(x) \approx \frac{b-a}{6}\left(f(a) + 4f\left(\tfrac{a+b}{2}\right) + f(b)\right)$$

This approximation requires three function evaluations. In general, these types of approximations are called the Newton-Cotes formulas. However, as the degree of the approximating polynomial becomes very high, the approximation can perform very poorly. This is because high order interpolating polynomials suffer from Runge's phenomenon, where the polynomial begins to fluctuate wildly. Despite this, any Newton-Cotes method can be used for composite integration. In the example above with the Lund fragmentation function, we already see that a higher order interpolation does not necessarily mean a faster convergence of the integral.

MC Integration

Another way to integrate a function is Monte Carlo (MC) integration. Given a function $f(x)$, sample x from a uniform distribution and then calculate $f(x)$. If we calculate the mean for $f(x)$ and multiply this by the integration range, we will then converge to the integral for a large enough number of sampled x.

$$\int_a^b \mathrm{d}x\, f(x) \approx \langle f(x) \rangle (b-a)$$

The nice thing about MC integration is that extending it to higher dimensions is very straightforward.

$$\int_V \mathrm{d}\,\vec{x} f(\vec{x}) \approx \langle f(\vec{x}) \rangle V$$

Here, V is just the volume of the hypercube which defines the limits of the integral. For the quadrature methods, extending to higher dimensions is also possible, but correctly implementing the interpolating function can be non-trivial.

Exercise: 1D MC integration

Implement 1D MC integration.

```
def integrate_mc(rng, f, xmin, xmax, nsample=1000):
    """
    Evaluate the definite integral of f(x) using MC integration.

    rng:     random number generator with `uniform()` method.
    f:       function to integrate, should be callable as `f(x)`.
    xmin:    lower bound.
    xmax:    upper bound.
    nsample: number of points to sample.
    """
    # Sample the points while tracking the sum of the function.
    f_sum = 0
    for i in range(0, nsample):
        # Sample a uniform random number between xmin and xmax.
        r = rng.uniform() * (xmax - xmin) + xmin
        # Calculate the function and sum.
        f_sum += f(r)
    # Return the mean times the volume.
    return f_sum / nsample * (xmax - xmin)
```

Compare the MC integration method with the quadrature methods for n is 10, 100, 1000, and 10000, where n is either the number of divisions or sampling points. How quickly is the MC integral converging?

```
# The integrate method cannot pass arguments to f(x). So we use a lambda
# function here to pass the parameters that we want.
f = lambda x: function_lff(x, a, b, m)

# Perform the integration and print.
for n in (10, 100, 1000, 10000):
    integral = integrate_midpoint(f, 0, 1, n)
    print(f" midpoint integral (n = {n:5d}) = {integral:.5e}")
    integral = integrate_trapezoid(f, 0, 1, n)
    print(f"trapezoid integral (n = {n:5d}) = {integral:.5e}")
    integral = integrate_mc(rng, f, 0, 1, n)
    print(f"MC integral        (n = {n:5d}) = {integral:.5e}")
```

```
 midpoint integral (n =    10) = 3.11772e+00
trapezoid integral (n =    10) = 1.99520e+00
MC integral        (n =    10) = 2.21091e+00
 midpoint integral (n =   100) = 3.53756e+00
trapezoid integral (n =   100) = 3.41425e+00
MC integral        (n =   100) = 3.08495e+00
 midpoint integral (n =  1000) = 3.46835e+00
trapezoid integral (n =  1000) = 3.46834e+00
```

```
MC integral        (n =  1000) = 3.85865e+00
 midpoint integral (n = 10000) = 3.46835e+00
trapezoid integral (n = 10000) = 3.46835e+00
MC integral        (n = 10000) = 3.40502e+00
```

Exercise: stratified integration

As we can see from the previous example for a 1D function, MC integration does not converge quickly in comparison to quadrature methods. One way that this convergence can be enhanced is stratified sampling of the function. Just like for quadrature methods, where the function is subdivided, the same can be done for MC integration.

$$\int_a^b \mathrm{d}x\, f(x) = \sum \int_{a_i}^{b_i} \mathrm{d}x\, f(x) \approx \sum_i \langle f(x) \rangle_i (b_i - a_i)$$

Here, i indicates each subdivision of the integration limits. Again, the extension of this method to higher dimensions is relatively straight forward, where we now have V_i indiciating each hypercube subdivision.

$$\int_V \mathrm{d}\vec{x}\, f(\vec{x}) = \sum \int_{V_i} \mathrm{d}\vec{x}\, f(\vec{x}) \approx \sum_i \langle f(\vec{x}) \rangle_i V_i$$

If we were to use regular subdivisions for stratified MC integration, and sample the same number of points from each subdivision, we would just recover our initial MC integration algorithm. So, how do we make this useful? We haven't talked about integration uncertainty in this notebook, but it is critical for setting up stratified integration. Without deriviation, the following can be shown.

> The integration uncertainty is minimized if each subdivision is sampled proportional to the relative variance of that subdivision.

So, how do we set up stratified integration?

1. Create m regular subdivisions of the integration limits.
2. Track σ_i, the variance of $f(x)$ for each subdivision. In practice, we actually store the mean of $f(x)$ and the mean of $f(x)^2$.
3. Sample points from each subdivision proportional to the following probability.

$$p_i = \frac{\sigma_i}{\sum_j \sigma_j}$$

4. Track the MC integral for each subdivision, F_i.
5. Return the sum of these integrals.

Let us now implement a general stratified MC integration method with the skeleton below.

```
def integrate_mc_strat(rng, f, xmin, xmax, nsample=1000, mdiv=4, nwarm=100):
    """
```

```
Evaluate the definite integral of f(x) using stratified MC integration.
Returns a tuple of `integral`, `[counts]` so that the division of counts
can be analyzed.

rng:     random number generator with `uniform()` method.
f:       function to integrate, should be callable as `f(x)`.
xmin:    lower bound.
xmax:    upper bound.
nsample: number of points to sample.
mdiv:    number of subdivisions.
nwarm:   number of warmup points used to calculate the variance.
"""
# Set up the edges for the subdivisions. Since we sample x from a uniform
# distribution, these edges should just run from 0 to 1, we can then
# transform to xmin to max.
edges = np.linspace(0, 1, mdiv)
dx = edges[1] - edges[0]

# Set up the sum of f(x) for each subdivision. We use this for the
# variance.
f_sums = [0] * mdiv
# Set up the number of sampled points per subdivision.
ns = [0] * mdiv

# We now need to perform a warmup integration to determine our variances.
# There are other ways to do this, but for simplicity, we sample `nwarm`
# points per subdivision.
var_total = 0
var_sums = []
for j in range(0, mdiv - 1):
    # Calculate the variance for this subdivision.
    f_sum = 0
    f2_sum = 0
    for i in range(0, nwarm):
        r = rng.uniform() * (edges[j + 1] - edges[j]) + edges[j]
        fj = f(r)
        f_sum += fj
        f2_sum += fj**2
    # Update the sum of variances.
    var_total += (f2_sum - f_sum**2) / nwarm
    var_sums += [var_total]

# Sample the points while tracking the sums of the function.
for i in range(0, nsample):
    # Choose which subdivision `j` to sample from based on the relative
    # variance.
    r1 = rng.uniform()
    for j, var_sum in enumerate(var_sums):
        if r1 < var_sum / var_total:
            break

    # Sample a uniform random number between xmin and xmax for the chosen
    # subdivision
```

```
        # subdivision.
        r = rng.uniform() * dx + edges[j]
        # Calculate the function and update the sums and counts.
        f_sums[j] += f(r)
        ns[j] += 1

    # Return the summed subdivision integrals.
    total = 0
    for j, (f_sum, n) in enumerate(zip(f_sums, ns)):
        if n != 0:
            total += f_sums[j] / n * dx
    return total, ns
```

First, let us just check that our sratified integration is behaving sensibly. Our method returns both the integral and the number of counts per subdivision. Integrate the Lund fragmentation function we have been using with 1000 sampling points (not including the warm up points) and 10 subdivisions. The resulting number of counts per subdivision should roughly mirror the structure of the distribution, i.e., there should be more counts toward the peaking structure.

```
# The integrate method cannot pass arguments to f(x). So we use a lambda
# function here to pass the parameters that we want.
f = lambda x: function_lff(x, a, b, m)

# We perform the integration and print the returned number of counts list.
integral, ns = integrate_mc_strat(rng, f, 0, 1, 1000, 10)
print(f"sampled counts = {ns}")
```

```
sampled counts = [881, 73, 24, 11, 3, 4, 4, 0, 0, 0]
```

Now, let us compare how quickly this converges with respect to the other methods. Note, this is not an entirely fair comparison, because we require 1000 points for our "warmup" of sampling the variances.

```
# The integrate method cannot pass arguments to f(x). So we use a lambda
# function here to pass the parameters that we want.
f = lambda x: function_lff(x, a, b, m)

# Perform the integration and print.
for n in (10, 100, 1000, 10000):
    integral = integrate_midpoint(f, 0, 1, n)
    print(f" midpoint integral (n = {n:5d}) = {integral:.5e}")
    integral = integrate_trapezoid(f, 0, 1, n)
    print(f"trapezoid integral (n = {n:5d}) = {integral:.5e}")
    integral = integrate_mc(rng, f, 0, 1, n)
    print(f"MC integral        (n = {n:5d}) = {integral:.5e}")
    integral, ns = integrate_mc_strat(rng, f, 0, 1, n, 10)
    print(f"MC strat integral  (n = {n:5d}) = {integral:.5e}")
```

```
 midpoint integral (n =    10) = 3.11772e+00
trapezoid integral (n =    10) = 1.99520e+00
```

```
MC integral        (n =    10) = 4.82362e+00
MC strat integral  (n =    10) = 2.12969e+00
 midpoint integral (n =   100) = 3.53756e+00
trapezoid integral (n =   100) = 3.41425e+00
MC integral        (n =   100) = 4.15957e+00
MC strat integral  (n =   100) = 2.93452e+00
 midpoint integral (n =  1000) = 3.46835e+00
trapezoid integral (n =  1000) = 3.46834e+00
MC integral        (n =  1000) = 3.41272e+00
MC strat integral  (n =  1000) = 3.33968e+00
 midpoint integral (n = 10000) = 3.46835e+00
trapezoid integral (n = 10000) = 3.46835e+00
MC integral        (n = 10000) = 3.50915e+00
MC strat integral  (n = 10000) = 3.47434e+00
```

Exercise: integration with AOR

One of the nice things about sampling with AOR, is that we can also perform an integral at the same time. The integral is just

$$\int_a^b \mathrm{d}x\, f(x) \approx \varepsilon f_{\max}(b-a),$$

where ε is the sampling efficiency that we defined before. Again, this generalizes nicely to higher dimensions.

$$\int_V \mathrm{d}\vec{x}\, f(\vec{x}) \approx \varepsilon V$$

Use the `SampleAOB` class to integrate the Lund fragmentation function which we have been using as our test function. Sample 10000 points. Do you expect this will converge faster or slower than the standard MC integration?

```
# The integrate method cannot pass arguments to f(x). So we use a lambda
# function here to pass the parameters that we want.
f = lambda x: function_lff(x, a, b, m)

# Create the sampler.
sampler = SampleAOR(rng, 0, 1, line.f)

# Sample 10000 points.
for i in range(0, 10000):
    sampler()

# Calculate the integral from the efficiency and print.
integral = sampler.e() * sampler.fmax * (sampler.xmax - sampler.xmin)
print(f"MC AOJ integral  (n = {n:5d}) = {integral:.5e}")
```

```
MC AOJ integral  (n = 10000) = 3.48120e+00
```