

# 1 Optional: Probability mass function (discrete)

This is a more general version of the previous bin variable, with the main difference being that each bin has a different likelihood (as specified by the input function). So it's a combination of the discrete variable (using a running sum to determine which bin you fall in) and the bins (chopping up a continuous variable into bins).

Technical note: In theory land, there is a difference between doing this as a continuous function (probability density) versus chopping it up into pieces (probability mass). You can actually do continuous functions, but it's a bit trickier and we don't need it (see for example <https://www.comsol.com/blogs/sampling-random-numbers-from-probability-distribution-functions/>)

For this example we're going to use a class instead of a method because (in order to make it efficient) you want to pre-calculate a running sum from the given probabilities. It would be very expensive to do this every time you asked for a sample, like you did in the discrete problem.

This is also a good time to do some fancy numpy array stuff, namely, using "where" to find the index (instead of writing your own for loop)

```
In [43]: class SampleProbabilityMassFunction:
def __init__(self, in_pdf, x_range=(0.0, 1.0), n_bins=100):
    """ Given a probability mass function, what range of x to use, and the number of samples
    sum/data needed to generate random samples from that pmf
    @param in_pdf - the function representing the probability distribution
    @param x_range - min and max x values as a tuple
    @param n_bins - number of bins """

    # TODO - Initialize correctly
    # Where the bins start and end
    # The amount of probability to put in each bin
    # The running sum
    # Where the center of each bin is (see sample_bin_variable above)
    self.bin_centers = np.zeros(n_bins)

    size_of_bin = (x_range[1] - x_range[0]) / n_bins
    self.bin_starts = np.zeros(n_bins)
    self.bin_ends = np.zeros(n_bins)
    self.bin_starts[0] = x_range[0]
    self.bin_ends[0] = x_range[0] + size_of_bin
    for i in range(1, n_bins):
        self.bin_starts[i] = self.bin_starts[i-1] + size_of_bin
        self.bin_ends[i] = self.bin_ends[i-1] + size_of_bin

    self.bin_centers = (self.bin_starts + self.bin_ends) / 2

    self.prob_per_bin = np.zeros(n_bins)
    for i in range(n_bins):
        self.prob_per_bin[i] = in_pdf(self.bin_centers[i])
```

```

# Create the pmf by evaluating in_pdf at the center of each bin
# Don't forget to normalize - the sum of self.bin_heights should be 1
self.bin_heights = np.zeros(n_bins)
self.bin_heights = self.prob_per_bin / np.sum(self.prob_per_bin)

# Running sum of probabilities - bin_sum[i] = sum(bin_heights[0:i])
# Note: It's a bit easier to generate_sample if you make this array n_bins+1, with the
# and the last value being 1
self.bin_sum = np.zeros(n_bins+1)
self.bin_sum[0] = 0
for i in range(1, n_bins+1):
    self.bin_sum[i] = self.bin_sum[i-1] + self.bin_heights[i-1]

def generate_sample(self):
    """ Draw one sample from the pmf
    Very similar to the discrete example above, for picking which bin, except you've pre-c
    Very similar to bin_sample for returning the bin center, except you've pre-calculated t
    @return bin center """
    zero_to_one = random.uniform()

    # You want the i where bin_sum[i] <= zero_to_one < bin_sum[i+1]
    # Not fancy version: Use a for loop
    # Fancy version: Use np.where
    # TODO - return correct bin center
    # return where zero_to_one is between bin_sum[i] and bin_sum[i+1]

    for i in range(len(self.bin_sum)-1):
        if self.bin_sum[i] <= zero_to_one < self.bin_sum[i+1]:
            return self.bin_centers[i]

def _generate_counts(self, n_samples):
    """ Generate n samples
    @param n_samples - number of samples
    @returns a numpy array with the counts for each bin, normalized"""

    # Counts
    counts = np.zeros(self.bin_centers.shape[0])

    # Make sure to take enough samples for all of the bins...
    bin_width = self.bin_centers[1] - self.bin_centers[0]
    for _ in range(0, self.bin_centers.shape[0] * 100):
        x_value = self.generate_sample()
        bin_index = np.ceil((x_value - self.bin_centers[0]) / bin_width)
        counts[int(bin_index)] += 1.0

    # Normalize
    counts = counts / sum(counts)
    return counts

def test_self(self, in_pdf):
    """ Check/test function
    @param in_pdf - the pdf function used to generate the values
    @returns True/False"""

```

```

        # Expected probability values
        expected_probs = in_pdf(self.bin_centers)
        # Normalize
        expected_probs /= np.sum(expected_probs)

        counts = self._generate_counts(100 * self.bin_centers.shape[0])

        for exp, c in zip(expected_probs, counts):
            print(f"pmf perc {c} expected {exp}")

            if np.abs(exp - c) > 0.1:
                print("Failed")
                return False

        print("Passed")
        return True

```

```

In [44]: def pdf(x):
        """ Made-up pdf (a quadratic). Can be anything, as long as it's not negative
        @param x
        @ return (x+1) * (x+1) + 0.1 """
        return (x+1) ** 2 + 0.1

```

```

In [45]: # Syntax check
        x_min = -2.0
        x_max = 1.0
        n_bins = 10

        # Make the class
        my_sample = SampleProbabilityMassFunction(pdf, (x_min, x_max), n_bins)
        # Generate a sample
        ret_value = my_sample.generate_sample()
        if x_min < ret_value < x_max:
            print("PMF: Passed syntax check")

```

PMF: Passed syntax check

```

In [46]: def test_pmf(b_do_plot=True):
        # Make the class
        x_min = -2.0
        x_max = 1.0
        n_bins = 10
        print("Sample pmf")
        my_sample = SampleProbabilityMassFunction(pdf, (x_min, x_max), n_bins)

        if not b_do_plot:
            return my_sample.test_self(pdf)

```

```

print(f"Passed test: {my_sample.test_self(pdf)}")

# Plot the results
_, axs = plt.subplots(1, 2)
xs = np.linspace(x_min, x_max, n_bins * 10)
ys = pdf(xs)
ys = ys / sum(pdf(my_sample.bin_centers))
axs[0].plot(xs, ys, '-k', label="pdf")
axs[0].plot(my_sample.bin_centers, my_sample.bin_heights, 'bX', label="pmf")
axs[0].legend()
axs[0].set_title("pdf to pmf")

# The more samples you take, the more it will look like the pmf
counts = my_sample._generate_counts(1000 * n_bins)
axs[1].plot(xs, ys, '-k', label="pdf")
axs[1].plot(my_sample.bin_centers, counts, 'bX', label="pmf samples")
axs[1].legend()
axs[1].set_title("Sampled pmf")

return True

```

In [47]: test\_pmf()

```

Sample pmf
pmf perc 0.077 expected 0.07528604118993135
pmf perc 0.039 expected 0.03684210526315787
pmf perc 0.013 expected 0.014874141876430203
pmf perc 0.009 expected 0.009382151029748284
pmf perc 0.016 expected 0.020366132723112138
pmf perc 0.04 expected 0.04782608695652175
pmf perc 0.096 expected 0.09176201372997714
pmf perc 0.17 expected 0.15217391304347824
pmf perc 0.198 expected 0.2290617848970252
pmf perc 0.342 expected 0.3224256292906178
Passed
Passed test: True

```

Out[47]: True



