Module Experiment 3F3
Random variables and random number generation

Tahmid Azam
Emmanuel College

November 2025

1 Uniform and normal random variables

1.1 Estimating a probability density function

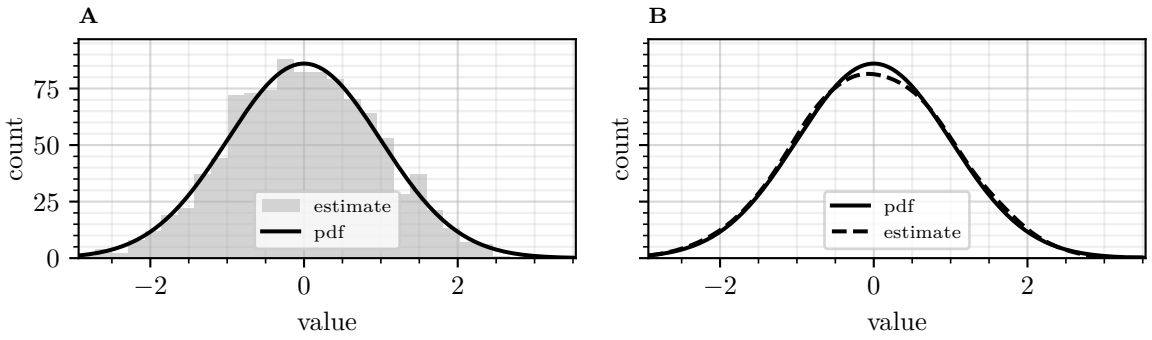

Figure 1: Histogram (subplot A, bin count, $k = 30$) and kernel smooth density function (subplot B, $\mathcal{K} = \mathcal{N}(x|0, 1)$, $\sigma = 0.3$) estimates of random samples from a normal distribution (`numpy.random.randn`, $n = 1000$) compared to the exact scaled probability density function (pdf) of a normal distribution (`scipy.stats.norm`).

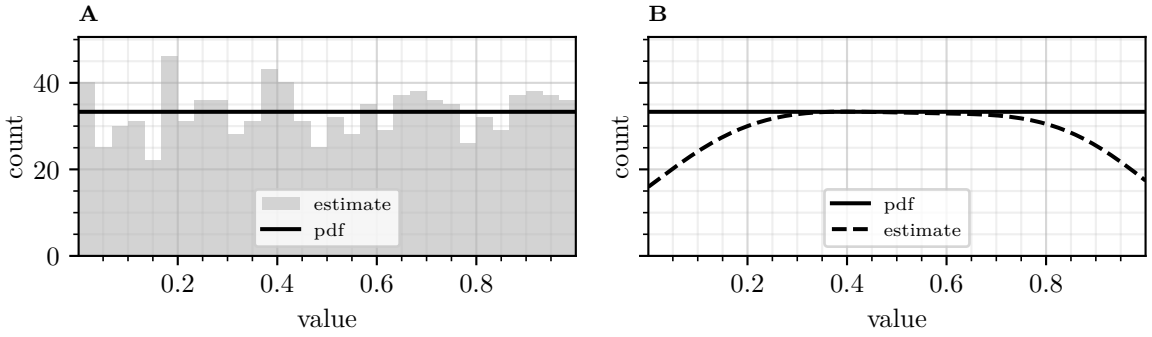

Figure 2: Histogram (subplot A, $k = 30$) and kernel smooth density function (subplot B, $\mathcal{K} = \mathcal{N}(x|0, 1)$, $\sigma = 0.15$) estimates of random samples from a uniform distribution (`numpy.random.rand`, $n = 1000$) compared to the exact scaled pdf of a uniform distribution (`scipy.stats.uniform`).

The kernel density estimate (KDE) produces a continuous and smooth function, unlike the discontinuous histogram. Both methods require selection of a resolution parameter (width, σ, for the KDE; bin count for the histogram). If these are too small, this leads to overfitting to the random samples; too large loses detail. The KDE also requires selection of a kernel function. The KDE performs poorly near the sharp edges of the uniform distribution (Figure 2) and creates artificial tails outside its domain of [0, 1] if evaluated there. The complexity of the histogram method for is $\mathcal{O}(n)$ (i.e., bin assignment for each n); for evaluating the KDE for m points, the complexity is $\mathcal{O}(n \times m)$ (i.e., sum over n kernels for each data point m) [1]. Optimised implementations of the KDE using the fast fourier transform achieve $\mathcal{O}(n + m \log m)$ [2]. Regardless, the KDE is more computationally expensive than the histogram method.

```python
from typing import Callable

import numpy as np
from matplotlib import pyplot as plt

from constants import PAGE_WIDTH_IN_INCHES
from ksdensity import ksdensity
from save_fig import save_fig


def plot_pdf_and_estimate(
        plot_id: str,
        gen_fn: Callable[[int], np.ndarray],
        pdf: Callable[[np.ndarray], np.ndarray],
        ks_density_width: float = 0.3,
        n: int = 1_000,
        bin_count: int = 30,
        linspace_num: int = 1_000,
):
    """
    Plots the histogram estimate and KDE of a given distribution.

    :param plot_id: The id of the plot used to name the exported file.
    :param gen_fn: The function used to generate samples.
    :param pdf: The probability density function.
    :param ks_density_width: The width used for the KDE.
    :param n: The number of samples.
    :param bin_count: The number of histogram bins.
    :param linspace_num: The number of points to evaluate the KDE at.
    """
    samples = gen_fn(n)
    samples_min = np.min(samples)
    samples_max = np.max(samples)
    x = np.linspace(samples_min, samples_max, linspace_num)
    pdf_values = pdf(x)
    ks_density = ksdensity(samples, width=ks_density_width)

    fig, (hist_ax, ksd_ax) = plt.subplots(
        figsize=(PAGE_WIDTH_IN_INCHES, 2),
        ncols=2,
        nrows=1,
        sharey=True
    )
    counts, bin_edges, _ = hist_ax.hist(
        samples,
        bins=bin_count,
        color="lightgrey",
        label="estimate"
    )

    bin_width = bin_edges[1] - bin_edges[0]
    scaled_pdf = pdf_values * bin_width * n
    scaled_ksd = ks_density(x) * bin_width * n

    hist_ax.plot(x, scaled_pdf, label="pdf", color="black")
    ksd_ax.plot(x, scaled_pdf, label="pdf", color="black")
    ksd_ax.plot(x, scaled_ksd, label="estimate", color="black", linestyle="dashed")

    ylim = np.max(counts) * 1.1

    for ax in [hist_ax, ksd_ax]:
        ax.set_ylim(bottom=0, top=ylim)
        ax.set_xlim(left=samples_min, right=samples_max)
        ax.set_xlabel("value")
        ax.set_ylabel("count")
        ax.legend(fontsize="x-small", loc="lower center")

    hist_ax.set_title("A", fontsize="small", loc="left", fontweight="bold")
    ksd_ax.set_title("B", fontsize="small", loc="left", fontweight="bold")

    save_fig(question=1, name=plot_id)
```

Figure 3: Python function definition for plotting the histogram estimate and KDE in Figure 1 and Figure 2.

```
plot_pdf_and_estimate(
    plot_id="normal",
    gen_fn=np.random.randn,
    pdf=lambda x: norm.pdf(x, 0, 1)
)

plot_pdf_and_estimate(
    plot_id="uniform",
    gen_fn=np.random.rand,
    pdf=lambda x: uniform.pdf(x, 0, 1),
    ks_density_width=0.15
)
```

Figure 4: Python function calls for plotting the histogram estimate and KDE in Figure 1 and Figure 2.

1.2 The multinomial distribution

For a continuous uniform distribution over $[a, b]$, we divide the interval into k equal-width bins. The probability of a random sample (of total N samples) falling into bin i is given by $p_j = \frac{1}{k}$, where $j = 1, 2, \ldots, k$. Using the multinomial theory, we can derive the theoretical mean and standard deviation of the histogram data as a function of N:

$$E[X_j] = Np_j = \frac{N}{k} \tag{1}$$

$$\mathrm{Var}(X_j) = Np_j(1 - p_j) = N\frac{1}{k}\left(1 - \frac{1}{k}\right) = \frac{N(k-1)}{k^2} \tag{2}$$

$$\sigma_{X_j} = \frac{\sqrt{N(k-1)}}{k} \tag{3}$$

Note the mean does not depend on N. To obtain the histogram estimate from the counts, we divide X_j by the number of samples N. Thus, we can derive expected value for the histogram estimate:

$$E\left[\frac{X_j}{N}\right] = \frac{1}{k} = p_j \tag{4}$$

Using the property $\mathrm{Var}(aX) = a^2\mathrm{Var}(X)$, we can derive the variance for the histogram estimate:

$$\mathrm{Var}\left(\frac{X_j}{N}\right) = \frac{1}{N^2}\mathrm{Var}(X_j) = \frac{1}{N^2} \cdot \frac{N(k-1)}{k^2} = \frac{k-1}{Nk^2} \tag{5}$$

$$\lim_{N\to\infty} \mathrm{Var}\left(\frac{X_j}{N}\right) = \lim_{N\to\infty} \frac{k-1}{Nk^2} = 0 \tag{6}$$

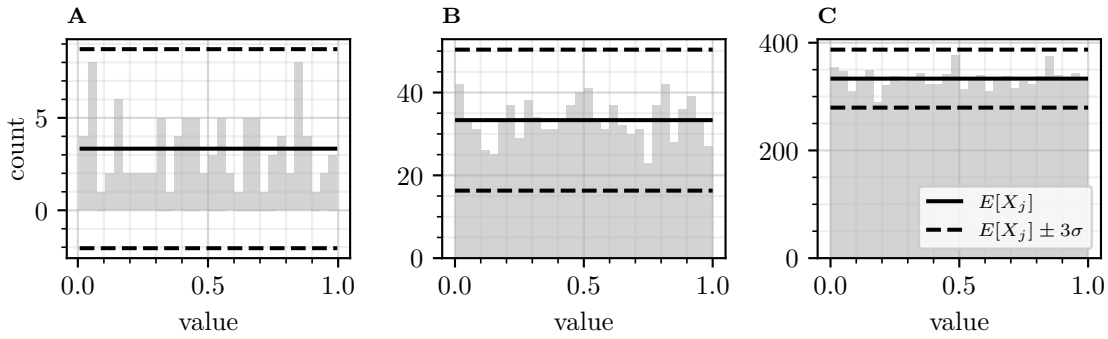

Figure 5: Data histograms ($k = 30$) for $N = 10^2$ (subplot A), $N = 10^3$ (subplot B), and $N = 10^4$ (subplot C) random samples from a uniform distribution (`numpy.random.rand`).

As N becomes large, the variance approaches 0 and the mean remains fixed at the true probability $p_j = \frac{1}{k}$. Consequently, the histogram estimate converges to the true pdf of the uniform distribution. In

Figure 5, the histogram counts fluctuate around the theoretical mean, and the majority of bins lie within $[E[X_j]-3\sigma, E[X_j]+3\sigma]$. As N increases, the fluctuations decrease and the histogram approaches the true pdf of the uniform distribution, consistent with the multinomial distribution theory. This consolidates the accuracy of Python's uniformly distributed random variates.

```python
import numpy as np
from matplotlib import pyplot as plt

from constants import PAGE_WIDTH_IN_INCHES
from save_fig import save_fig


def plot_multinomial_theory(
        n_values: list[int],
        subplot_titles: list[str],
        k: int = 30,
) -> None:
    """
    Plots histograms of the uniform distribution with overlaid horizontal
    lines for the theoretical mean and -3 and 3 times the standard deviation.

    :param n_values: The values for the sample count for each subplot.
    :param subplot_titles: The titles for each subplot.
    :param k: The number of histogram bins.
    """
    fig, axes = plt.subplots(figsize=(PAGE_WIDTH_IN_INCHES, 2), ncols=len(n_values), nrows=1)

    for i, (ax, n) in enumerate(zip(axes, n_values)):
        vector = np.random.rand(n)
        p = 1 / k
        theoretical_mean = n * p
        theoretical_std_dev = np.sqrt(n * p * (1 - p))
        counts, bin_edges, _ = ax.hist(vector, bins=k, color="lightgrey")

        ax.hlines(
            theoretical_mean, bin_edges[0], bin_edges[-1],
            color="black",
            label="$E[X_j]$",
        )

        three_std_dev_lines = [
            theoretical_mean + 3 * theoretical_std_dev,
            theoretical_mean - 3 * theoretical_std_dev
        ]
        ax.hlines(
            three_std_dev_lines,
            bin_edges[0], bin_edges[-1],
            color="black",
            linestyle="dashed",
            label=r"$E[X_j] \pm 3\sigma$"
        )

        ax.set_xlabel("value")
        ax.set_title(subplot_titles[i], fontsize="small", loc="left", fontweight="bold")

    axes[0].set_ylabel("count")
    axes[-1].legend(fontsize="x-small", loc="lower right")

    save_fig(question=1, name="uniform_multinomial")
```

Figure 6: Python function implementation for plotting the histograms in Figure 5.

```python
plot_multinomial_theory(
    n_values=[100, 1_000, 10_000],
    subplot_titles=["A", "B", "C"],
)
```

Figure 7: Python function calls for plotting the histograms in Figure 5.

2 Functions of random variables

For $p_X(x) = \mathcal{N}(x|0,1)$, and $y = f(x) = ax + b$ (Figure 8, subplot A), we can derive $p_Y(y)$ using the Jacobian formula:

$$p_X(x) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left[-\frac{(x-\mu)^2}{2\sigma^2}\right]\Bigg|_{\mu=0,\sigma^2=1} = \frac{1}{\sqrt{2\pi}} \exp\left[-\frac{x^2}{2}\right] \tag{7}$$

$$\frac{dy}{dx} = a \tag{8}$$

$$f^{-1}(y) = \frac{y-b}{a} \tag{9}$$

$$p_Y(y) = p_X(x)\frac{1}{\left|\frac{dy}{dx}\right|}\Bigg|_{x=f^{-1}(y)} = p_X\left(\frac{y-b}{a}\right)\cdot\frac{1}{|a|} = \frac{1}{|a|\sqrt{2\pi}} \exp\left[-\frac{1}{2}\left(\frac{y-b}{a}\right)^2\right] \tag{10}$$

$$= \frac{1}{\sqrt{2\pi a^2}} \exp\left[-\frac{(y-b)^2}{2a^2}\right] = \mathcal{N}(y|b,a^2) \tag{11}$$

For the case $y = f(x) = x^2$ (Figure 8, subplot B), we have to account for a 2-to-1 mapping:

$$\frac{dy}{dx} = 2x \tag{12}$$

$$f^{-1}(y) = \pm\sqrt{y}, \quad y \geq 0 \tag{13}$$

$$p_Y(y) = p_X(x)\frac{1}{\left|\frac{dy}{dx}\right|}\Bigg|_{x=\sqrt{y}} + p_X(x)\frac{1}{\left|\frac{dy}{dx}\right|}\Bigg|_{x=-\sqrt{y}} = \frac{p_X(\sqrt{y})}{2\sqrt{y}} + \frac{p_X(-\sqrt{y})}{2\sqrt{y}} \tag{14}$$

$$= \frac{1}{\sqrt{2\pi y}} \exp\left[-\frac{y}{2}\right], \quad y \geq 0 \tag{15}$$

Thus, squaring a standard normal random variable yields a distribution supported on $y \geq 0$, corresponding to a chi-squared distribution with one degree of freedom.

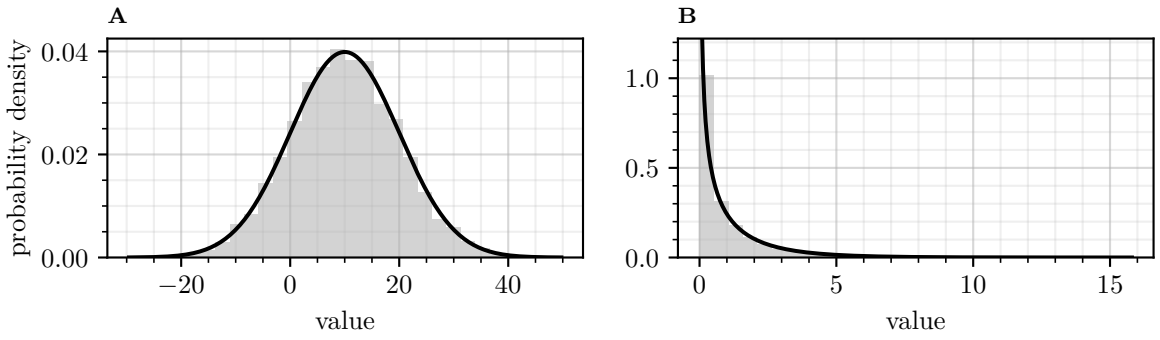

Figure 8: Histogram ($k = 30$) of $N = 10^5$ samples $x^{(i)}$ to give $y^{(i)} = f(x^{(i)})$ using $f(x) = ax + b$ (subplot A, $a = b = 10$) and $f(x) = x^2$ (subplot B) overlaid with the theoretical pdf calculated using the Jacobian.

```python
import numpy as np
from matplotlib import pyplot as plt
from scipy.stats import norm

from constants import PAGE_WIDTH_IN_INCHES
from save_fig import save_fig


def jacobian(
        a: float = 10,
        b: float = 10,
        n: int = 10_000,
        bin_count: int = 30,
        linspace_num: int = 1_000,
):
    """
    Plots histograms of linear [f(x) = ax + b] and square [f(x) = x^2] transformations of
    samples from the normal distribution, overlaid with the theoretical pdfs calculated
    using the Jacobian.

    :param a: The value of constant a.
    :param b: The value of constant b.
    :param n: The number of samples taken from the normal distribution.
    :param bin_count: The number of histogram bins.
    :param linspace_num: The number of points to evaluate the pdfs at.
    """
    x = np.random.randn(n)
    y_linear = a * x + b
    y_square = x ** 2
    y_linear_vals = np.linspace(min(y_linear), max(y_linear), num=linspace_num)
    y_square_vals = np.linspace(
        start=1e-6,  # Avoid division by 0
        stop=max(y_square),
        num=linspace_num
    )
    p_y_linear = (1 / abs(a)) * norm.pdf((y_linear_vals - b) / a)
    p_y_square = np.array([
        0.5 / np.sqrt(val) * (norm.pdf(np.sqrt(val)) + norm.pdf(-np.sqrt(val)))
        for val in y_square_vals
    ])

    fig, (linear_ax, square_ax) = plt.subplots(
        nrows=1,
        ncols=2,
        figsize=(PAGE_WIDTH_IN_INCHES, 2)
    )

    linear_ax.hist(y_linear, bins=bin_count, color="lightgrey", density=True)
    linear_ax.plot(y_linear_vals, p_y_linear, color="black")
    counts_square, _, _ = square_ax.hist(y_square, bins=bin_count, color="lightgrey", density=True)
    square_ax.plot(y_square_vals, p_y_square, color="black")

    max_bin_count = counts_square.max()
    square_ax.set_ylim(top=1.2 * max_bin_count)  # Limit y-axis due to large values close to 0.
    linear_ax.set_xlabel("value")
    square_ax.set_xlabel("value")
    linear_ax.set_ylabel("probability density")
    linear_ax.set_title("A", fontsize="small", loc="left", fontweight="bold")
    square_ax.set_title("B", fontsize="small", loc="left", fontweight="bold")

    save_fig(question=2, name="jacobian")
```

Figure 9: Python implementation for plotting the histogram estimates in Figure 8.

3 Inverse cumulative distribution function method

For the exponential distribution with mean 1 with pdf $p(y) = \exp(-y)$, $y \geq 0$, we can derive the cumulative distribution function (CDF) and its inverse:

$$F_Y(y) = P(Y \leq y) = \int_0^y e^{-t} dt = [-e^{-t}]_0^y = 1 - e^{-y}, \quad y \geq 0 \tag{16}$$

$$F_Y^{-1}(u) = -\ln(1-u), \quad u \in [0, 1) \tag{17}$$

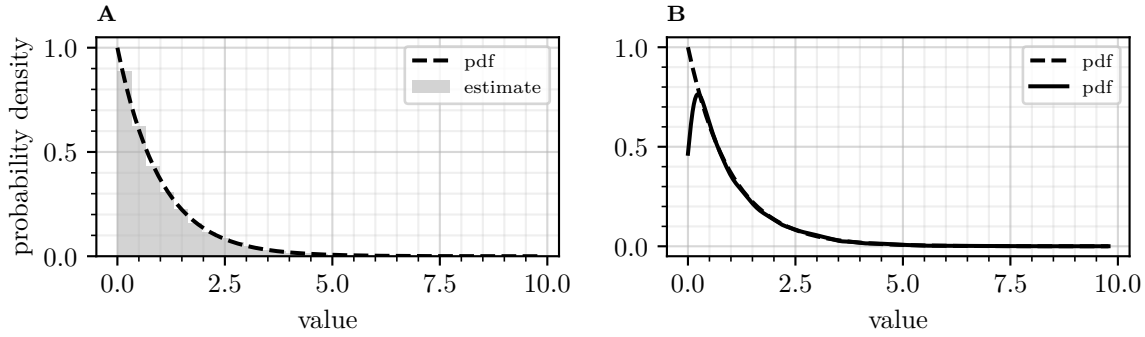

Figure 10: Histogram (subplot A, $k = 30$) and kernel density (subplot B, $\mathcal{K} = \mathcal{N}(x|0, 1)$, $\sigma = 0.15$) estimates from $N = 10^4$ samples generated using the inverse CDF method compared to the exact pdf of the exponential distribution with mean 1.

```python
import numpy as np
from matplotlib import pyplot as plt

from constants import PAGE_WIDTH_IN_INCHES
from ksdensity import ksdensity
from save_fig import save_fig


def inverse_cdf(
        ks_density_width: float = 0.15,
        n: int = 10_000,
        k: int = 30,
        linspace_num: int = 1_000,
):
    """
    Plots the histogram estimate and the KDE for the exponential
    distribution with mean 1 using the inverse cdf method.

    :param ks_density_width: The width used in the KDE.
    :param n: The number of samples from the uniform distribution.
    :param k: The number of histogram bins.
    :param linspace_num: The number of points to evaluate the KDE at.
    """
    u_samples = np.random.rand(n)
    exp_samples = -np.log(1 - u_samples)
    x = np.linspace(
        start=exp_samples.min(),
        stop=exp_samples.max(),
        num=linspace_num
    )
    pdf = np.exp(-x)
    ks_density = ksdensity(exp_samples, width=ks_density_width)

    fig, (hist_ax, ksd_ax) = plt.subplots(figsize=(PAGE_WIDTH_IN_INCHES, 2), nrows=1, ncols=2)

    hist_ax.plot(x, pdf, color="black", linestyle="dashed", label="pdf")
    ksd_ax.plot(x, pdf, color="black", linestyle="dashed", label="pdf")
    hist_ax.hist(exp_samples, bins=k, color="lightgrey", density=True, label="estimate")
    ksd_ax.plot(x, ks_density(x), color="black", label="pdf")

    hist_ax.set_ylabel("probability density")
    hist_ax.set_title("A", fontsize="small", loc="left", fontweight="bold")
    ksd_ax.set_title("B", fontsize="small", loc="left", fontweight="bold")

    for ax in [hist_ax, ksd_ax]:
        ax.legend(fontsize="x-small")
        ax.set_xlabel("value")

    save_fig(question=3, name="inverse_cdf")
```

Figure 11: Python implementation for plotting the histogram and KDE estimates in Figure 10.

4 Simulation from a 'difficult' density

For $\alpha \in (0, 2)$ $(\alpha \neq 1)$, $\beta \in [-1, 1]$, let:

$$b = \frac{1}{\alpha} \arctan\left[\beta \tan\left(\frac{\pi\alpha}{2}\right)\right] \tag{18}$$

$$s = \left[1 + \beta^2 \tan^2\left(\frac{\pi\alpha}{2}\right)\right]^{\frac{1}{2\alpha}} \tag{19}$$

$$U \sim \mathcal{U}(-\frac{\pi}{2}, \frac{\pi}{2}) \tag{20}$$

$$V \sim \mathcal{E}(V|1) \tag{21}$$

$$X = s\frac{\sin(\alpha(U+b))}{(\cos(U))^{\frac{1}{\alpha}}}\left(\frac{\cos(U - \alpha(U+b))}{V}\right)^{\frac{1-\alpha}{\alpha}} \tag{22}$$

In Figure 12, varying α changes the width of the density and varying β shifts the mass left or right. The parameter $\alpha \in (0, 2)$ controls the spread of the distribution. Small values of α concentrate most

of the probability mass near the center, producing a sharp peak, while larger values of α spread the distribution and increase the relative probability of extreme values. The parameter $\beta \in [-1, 1]$ controls the skewness: $\beta = 0$ gives a symmetric distribution, $\beta > 0$ skews it to the right, and $\beta < 0$ skews it to the left.

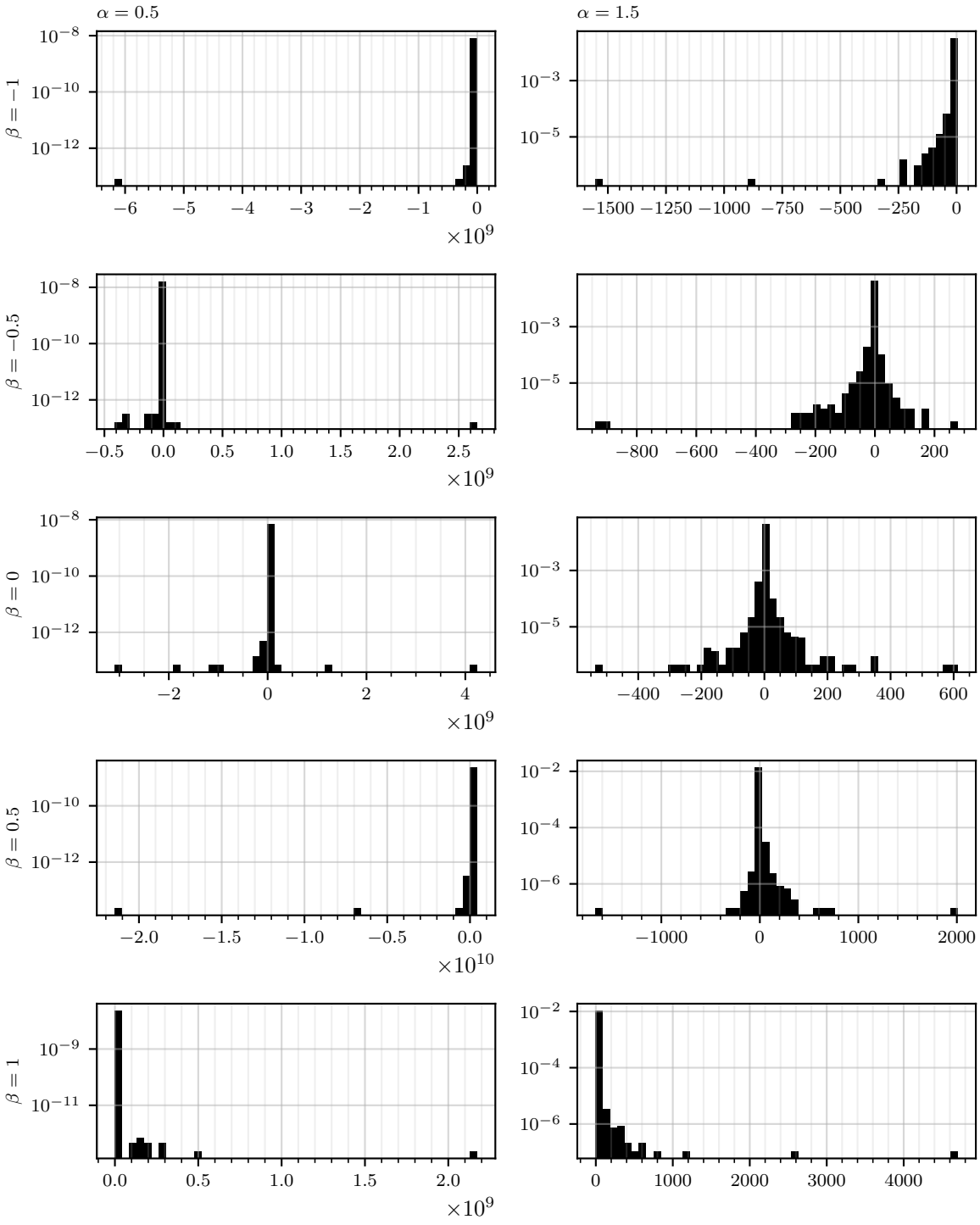

Figure 12: Histogram density estimates ($k = 30$, $\alpha = 0.5, 1.5$, $\beta = -1, -0.5, 0, 0.5, 1$) from $N = 10^5$ samples from X.

```python
import numpy as np


def calc_const_s(alpha: float, beta: float) -> float:
    return np.power(
        1 + np.power(beta, 2) * np.power(np.tan(np.pi * alpha / 2), 2),
        1 / (2 * alpha)
    )
```

Figure 13: Python implementation for calculating s.

```python
import numpy as np


def calc_const_b(alpha: float, beta: float) -> float:
    return (1 / alpha) * np.arctan(beta * np.tan(np.pi * alpha / 2))
```

Figure 14: Python implementation for calculating b.

```python
import numpy as np


def calc_rv_x(
        alpha: float,
        s: float,
        b: float,
        u: np.ndarray,
        v: np.ndarray
) -> np.ndarray:
    return s * ((np.sin(alpha * (u + b))) / (np.power(
        np.cos(u),
        1 / alpha
    ))) * np.power(
        ((np.cos(u - alpha * (u + b))) / (v)),
        (1 - alpha) / alpha
    )
```

Figure 15: Python implementation for calculating the random variable X.

```python
import numpy as np
from matplotlib import pyplot as plt

from calc_const_b import calc_const_b
from calc_const_s import calc_const_s
from calc_rv_x import calc_rv_x
from constants import PAGE_WIDTH_IN_INCHES, PAGE_HEIGHT_IN_INCHES
from save_fig import save_fig


def simulation(
        alphas: list[float],
        betas: list[float],
        n: int = 100_000,
        k: int = 50,
) -> None:
    """
    Plots histogram density estimates for a set of alpha and beta values.

    :param alphas: The list of alpha values, where 0 <= alpha <= 2, alpha != 1.
    :param betas: The list of beta values, where beta -1 <= beta <= 1.
    :param n: The number of samples of X to plot.
    :param k: The number of histogram bins.
    """
    fig, axes = plt.subplots(
        nrows=len(betas),
        ncols=len(alphas),
        figsize=(PAGE_WIDTH_IN_INCHES, PAGE_HEIGHT_IN_INCHES * 0.8),
    )

    for i, beta in enumerate(betas):
        for j, alpha in enumerate(alphas):
            b = calc_const_b(alpha, beta)
            s = calc_const_s(alpha, beta)
            u = np.random.uniform(-np.pi / 2, np.pi / 2, size=n)
            v = np.random.exponential(scale=1, size=n)
            x = calc_rv_x(alpha, s, b, u, v)

            ax = axes[i, j]
            ax.tick_params(axis='both', which='major', labelsize="small")
            ax.hist(x, bins=k, label="histogram", color="black", density=True, log=True)

            if j == 0:
                ax.set_ylabel(rf"$\beta = {beta}$", fontsize="small")

            if i == 0:
                ax.set_title(rf"$\alpha = {alpha}$", fontsize="small", loc="left")

    save_fig(question=4, name="simulation")
```

Figure 16: Python implementation for plotting the histogram estimates in Figure 12.

References

[1] Vikas C. Raykar, Ramani Duraiswami, and Linda H. Zhao. "Fast Computation of Kernel Estimators". In: *Journal of Computational and Graphical Statistics* 19.1 (Jan. 1, 2010), pp. 205–220. ISSN: 1061-8600. DOI: 10.1198/jcgs.2010.09046. URL: https://doi.org/10.1198/jcgs.2010.09046 (visited on 11/08/2025).

[2] Jeffrey Heer. "Fast & Accurate Gaussian Kernel Density Estimation". In: *2021 IEEE Visualization Conference (VIS)*. 2021 IEEE Visualization Conference (VIS). Oct. 2021, pp. 11–15. DOI: 10.1109/VIS49827.2021.9623323. URL: https://ieeexplore.ieee.org/document/9623323 (visited on 11/08/2025).