A black text on a white background

Description automatically generated

Department of Engineering

Name: Jack Henry Date: 17 October 2024

This student has a Student Support Document (SSD) from the Accessibility Resource Centre (ADRC) which gives the following recommendation regarding assessment:

In all assessed work, examinations and coursework, this student will not be penalised for minor errors in spelling and writing, or an over-emphasis on writing style over content, where these do not interfere with the communication.

Teaching Office 2024-25

# ENGINEERING TRIPOS PART II A

**EIETL MODULE EXPERIMENT 3F3 RANDOM VARIABLES and RANDOM NUMBER GENERATION**

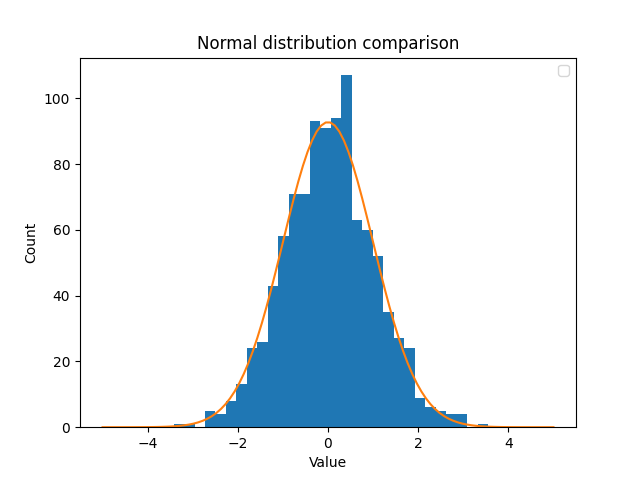
**Short Report Template Name: Jack Henry**

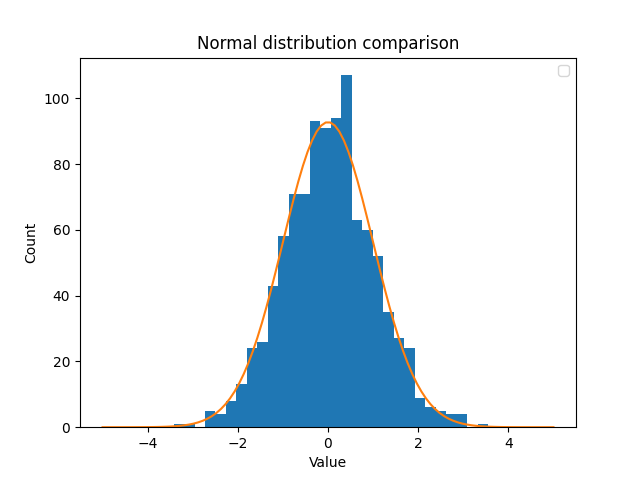
**College: Churchill**

**Lab Group Number: 15th November 2024 Student 10**

1. **Uniform and normal random variables.**

Histogram of Gaussian random numbers overlaid on exact Gaussian curve (scaled):





Histogram of Uniform random numbers overlaid on exact Uniform curve (scaled):

A graph of a tall building

Description automatically generated with medium confidence

Kernel density estimate for Gaussian random numbers overlaid on exact Gaussian curve:

A diagram of normal distribution

Description automatically generated

A diagram of normal distribution

Description automatically generated

Kernel density estimate for Uniform random numbers overlaid on exact Gaussian curve:

A graph of a normal distribution

Description automatically generated

Comment on the advantages and disadvantages of the kernel density method compared with the histogram method for estimation of a probability density from random samples:

The kernel density method shows a smooth curve that uses an average of the counts for each value. This gives a good overall shape when estimating a pdf of that is also smooth, however will also tend to smooth out any sharper features of the pdf. This can be seen from the slight smoothing of the peak of the normal distribution, and especially in the corners of the uniform distribution. The uniform distribution can still be identified from this method however, as the count goes up to 1.0, and the distribution centres on 0.5 and generally spans from 0 to 1. It also does not show the variation of the counts, which the histogram method does. The histogram method shows both the overall shape, and variation can be inferred through the visuals of each bin count. These bins however reduce the resolution of the estimation.

Theoretical mean and standard deviation calculation for uniform density as a function of *N* :

The probability that a sample lies within a bin of the histogram is:

(1)

For a uniform distribution for values for :

(2)

The integral in equation (1) evaluated using equation (2) gives:

(3)

Each bin can be modelled as a binomial distribution as

Therefore, the mean of the count data for each bin is:

(4)

And the standard deviation of the count data for each bin is:

(5)

Explain behavior as *N* becomes large:

From equations (4) and (5):

and (6)

This means as N increases, the mean will increase in proportion by the same amount, whilst the standard deviation will only increase through a square root relationship.

Effectively, the relative value of the mean compared to the number of generated random variables remains constant, whilst the standard deviation decreases at a square root rate. This can be seen in the figures below.

Plot of histograms for *N* = 100, *N* = 1000 and *N* = 10000 with theoretical mean and *±*3 standard deviation lines:

A graph with blue lines and red lines

Description automatically generated

Red line = theoretical mean

Blue dotted line = theoretical mean ± 3 standard deviations

Are your histogram results consistent with the multinomial distribution theory?

Yes, the results are consistent. It can visually be seen from the histograms that the mean of the entire histogram is the theoretical mean, with some variation for each bin. This variation is contained within the ±3 standard deviation range, for all 3 subplots, which is where we would expect over 99% of results.

1. **Functions of random variables** For normally distributed *N* (*x|*0*,* 1) random variables, take *y* = *f* (*x*) = *ax* + *b*. Calculate *p*(*y*) using the Jacobian formula:

If and , we can calculate using

(7)

where for each possible function of .

In the case of ,

(8)

and

(9)

Therefore,

(10)

and as is a standard gaussian distribution, we can write

(11)

It is clear that the distribution for Y can be modelled as:

Explain how this is linked to the general normal density with non-zero mean and non-unity variance:

Doing the above calculations for a general distribution of , we get equation (12)

(12)

From this we can see that the transformation of f translates the mean and standard deviation for a general normal density by and respectively.

Verify this formula by transforming a large collection of random samples *x*(*i*) to give *y*(*i*) = *f* (*x*(*i*)), histogramming the resulting *y* samples, and overlaying a plot of your formula calculated using the Jacobian:

A graph of gaussian using a function

Description automatically generated

Now take *p*(*x*) = *N* (*x|*0*,* 1) and *f* (*x*) = *x*2. Calculate *p*(*y*) using the Jacobian formula:

Using the same method as before and equation (7), for and , it is now found that:

(13)

and

(14)

Therefore,

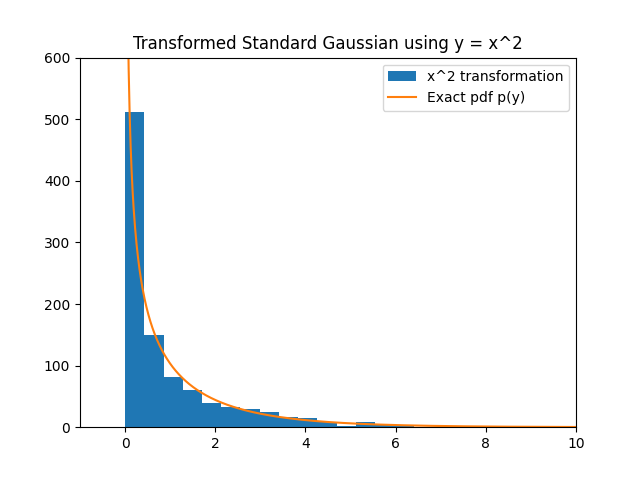
(15)

And so,

(16)

This is an exponentially distributed pdf.

Verify your result by histogramming of transformed random samples:



# Inverse CDF method

Calculate the CDF and the inverse CDF for the exponential distribution:

For an exponential distribution

(17)

The CDF is the integral of this,

(18)

The inverse CDF for is then

(19)

If we apply this to a uniform distribution of , we can generate an exponential distribution by transforming each uniformly distributed random variable. Due to the symmetry and range limit of the uniform distribution used we can also simply equation (19) to

(20)

Python code for inverse CDF method for generating samples from the exponential distribution:

def task\_3():

x = np.random.rand(1000) # generate uniform random variables

x\_values = np.linspace(0, 7, 1000)

exp\_pdf = np.exp(-x\_values)

y = -np.log(x) # Inverse CDF of x. x is uniformly distributed between 0 and 1 so can simplify 1-x to x

Area\_transformed\_exp = (1000\*(max(y)-min(y))/30)

plt.figure()

plt.hist(y, bins=30, label = 'Histogram of inverse CDF pdf')

plt.plot(x\_values, Area\_transformed\_exp\*exp\_pdf, label='exp pdf')

ks\_density = ksdensity(y, width=0.3)

plt.plot(x\_values, Area\_transformed\_exp\*ks\_density(x\_values), label='ksd', color='r')

plt.title('exponential distribution generated from inverse CDF')

plt.legend()

plt.xlim(-1,7)

plt.savefig('Inverse CDF Method')

plt.show()

plt.close()

Plot histograms/ kernel density estimates and overlay them on the desired expo- nential density:

*A graph of a function

Description automatically generated*

1. Simulation from a `difficult' density.

Python code to generate N random numbers drawn from the distribution of X:

def task\_5():

# a) choose parameters

alphas = [0.5, 1.5] # Different alpha values to test

betas = [-1, -0.5, 0, 0.5, 1] # Different beta values

# Function to generate X based on the given recipe

def calc\_X(alpha, beta, N):

b = (1 / alpha) \* np.arctan(beta \* np.tan(np.pi \* alpha / 2))

s = (1 + beta\*\*2 \* np.tan(np.pi \* alpha / 2)\*\*2)\*\*(1 / (2 \* alpha))

U = np.random.uniform(-np.pi / 2, np.pi / 2, N)

V = np.random.exponential(1, N)

X = (

s

\* (np.sin(alpha \* (U + b)) / (np.cos(U))\*\*(1 / alpha))

\* ((np.cos(U - alpha \* (U + b))) / V)\*\*((1 - alpha) / alpha)

)

return X

N = 10000 # Number of samples

bins = 100 # Higher resolution for histograms

for alpha in alphas:

fig, axs = plt.subplots(len(betas), figsize=(10, len(betas) \* 2), sharex=False)

for i, beta in enumerate(betas):

ax = axs[i] if len(betas) > 1 else axs

X = calc\_X(alpha, beta, N)

# Dynamically determine truncation range based on percentiles

x\_min, x\_max = np.percentile(X, [1, 99])

# Apply truncation only if outliers are few

X\_truncated = np.clip(X, x\_min, x\_max)

counts, bin\_edges = np.histogram(X\_truncated, bins=bins, range=(x\_min, x\_max), density=True)

# Adjust first and last bins for outliers

counts[0] += np.sum(X < x\_min) / (N \* (x\_max - x\_min) / bins)

counts[-1] += np.sum(X > x\_max) / (N \* (x\_max - x\_min) / bins)

# Plot histogram

ax.bar(bin\_edges[:-1], counts, width=(bin\_edges[1] - bin\_edges[0]), align='edge', label=f'Œ±={alpha}, Œ≤={beta}')

ax.set\_xlabel('X')

ax.set\_ylabel('Density')

ax.set\_xlim(x\_min, x\_max)

ax.legend()

ax.set\_xticks(np.linspace(x\_min, x\_max, 5)) # Add x-axis ticks

fig.suptitle(f'Difficult Densities for Œ±={alpha}', fontsize=16)

plt.tight\_layout()

plt.savefig(f'Difficult\_densities\_alpha\_{alpha}.png')

plt.show()

Plot some histogram density estimates with alpha= 0.5, 1.5 and several values of beta :

A graph of a number of numbers

Description automatically generated with medium confidence

A graph of different sizes and shapes

Description automatically generated with medium confidence

Zoomed in versions have also been provided.

A screenshot of a graph

Description automatically generated

A group of blue graphs

Description automatically generated with medium confidence

Hence comment on the interpretation of the parameters alpha and beta:

The figures are set to truncate the graphs based on percentiles. From this we can see that the 98th percentile range for alpha = 0.5 is far greater than alpha = 1.5, suggesting a larger alpha reduces the overall range of the distribution. The magnitude of beta also seems to affect this range, with high magnitudes corresponding to higher ranges.

Despite the wider range of the alpha = 0.5 graphs, it seems that they also have a greatly concentrated centre of the distribution, with sharper peaks and what looks like a quickly decaying distribution, possibly suggesting a small range for around 80% of data points. This suggests a smaller alpha creates a higher concentrated centre. Beta seems to have minimal effect on this concentration.

Beta seems to mainly affect the skew of the distribution, with Beta values of 0 corresponding to a symmetrical distribution, and negative and positive values corresponding to the opposite direction skewness.

Both alpha and beta affect the mode value, and likely the mean as well. The position of this seems to be a balance between the two variables as the graphs for alpha = 0.5 are different in polarity of the mode compared to alpha = 1.5.

Appendix: full python code

import numpy as np

import matplotlib.pyplot as plt

from scipy.stats import uniform

def ksdensity(data, width=0.3):

"""Returns kernel smoothing function from data points in data"""

def ksd(x\_axis):

def n\_pdf(x, mu=0., sigma=1.): # normal pdf

u = (x - mu) / abs(sigma)

y = (1 / (np.sqrt(2 \* np.pi) \* abs(sigma)))

y \*= np.exp(-u \* u / 2)

return y

prob = [n\_pdf(x\_i, data, width) for x\_i in x\_axis]

pdf = [np.average(pr) for pr in prob] # each row is one x value

return np.array(pdf)

return ksd

def norm\_pdf(x, mu=0, sigma=1): # normal pdf

"""Returns normal distribution"""

u = (x - mu) / abs(sigma)

y = (1 / (np.sqrt(2 \* np.pi) \* abs(sigma)))

y \*= np.exp(-u \* u / 2)

return y

def uni\_pdf(x, b=1, a=0):

"""Returns uniform distribution"""

#y = np.linspace(1, 1, Length)

y = uniform.pdf(x, loc=a, scale=b-a)

return y

def task\_1():

def normal\_task():

#Plot histogram against normal

# np.linspace(start, stop, number of steps)

x\_values = np.linspace(-5., 5., 100)

# Plot normal distribution

x = np.random.randn(1000)

plt.figure(1)

plt.hist(x, bins=30) # number of bins

# Area of histogram = number of variables (N) times range divided by number of bins. This is the scale factor

Area\_norm = 1000\*(max(x)-min(x))/30 # This is a slight approximation but is sufficient

#plot exact normal distribution

plt.plot(x\_values, Area\_norm\*norm\_pdf(x\_values))

plt.legend()

plt.title('Normal distribution comparison')

plt.xlabel('Value')

plt.ylabel('Count')

plt.legend()

plt.savefig('Normal hist.png')

plt.show()

plt.close()

#Plot normal against kernel density

plt.figure(2)

plt.plot(x\_values, norm\_pdf(x\_values), label='normal')

ks\_density = ksdensity(x, width=0.4)

plt.plot(x\_values, ks\_density(x\_values), label='ksd')

plt.legend()

plt.title('Normal distribution comparison')

plt.xlabel('Value')

plt.ylabel('Count')

plt.legend()

plt.savefig('Normal kernel.png')

plt.show()

plt.close()

def uniform\_task():

# Plot uniform distribution against histogram

x = np.random.rand(1000)

plt.figure(1)

plt.hist(x, bins=20)

x\_values = np.linspace(-4, 4, 1000)

#plot exact uniform distribution

Area\_uni=1000\*1/20 #Area of histogram = number of variables (N) times range divided by number of bins. This is the scale factor

plt.plot(x\_values, Area\_uni\*uni\_pdf(x\_values), label='uniform')

plt.xlim(-0.5,1.5)

plt.legend()

plt.title('uniform distribution comparison')

plt.xlabel('Value')

plt.ylabel('Count')

plt.legend()

plt.savefig('Uniform hist.png')

plt.show()

plt.close()

#Plot normal against kernel density

plt.figure(2)

ks\_density = ksdensity(x, width=0.2)

plt.plot(x\_values, norm\_pdf(x\_values), label='uniform')

plt.plot(x\_values, ks\_density(x\_values), label='ksd')

plt.xlim(-4,4)

plt.legend()

plt.title('uniform distribution comparison')

plt.xlabel('Value')

plt.ylabel('Count')

plt.legend()

plt.savefig('Uniform kernel.png')

plt.show()

plt.close()

def N\_task():

N\_list=[100, 1000, 10000]

n\_bins = 20 # number of histogram bins

fig, ax = plt.subplots(3)

for N in N\_list:

# Generalising uniform histograms

mean = N/n\_bins # N is number of generated RVs. Mean for numbers only in range of interest.

sd = np.sqrt(N\*(1-1/n\_bins)/n\_bins)

x = np.random.rand(N)

p = ax[N\_list.index(N)]

p.hist(x, bins=n\_bins, label = 'N={}'.format(N))

p.axhline(y=mean, color='r', linestyle='-', linewidth=2)

p.axhline(y=mean+3\*sd, color='b', linestyle='--', linewidth=2)

p.axhline(y=mean-3\*sd, color='b', linestyle='--', linewidth=2)

p.set\_xlim(-0.5,1.5)

p.set\_xlabel('Value')

p.set\_ylabel('Count')

p.legend()

fig.suptitle('Histograms for an increasing N number of uniform random variables')

plt.savefig('Uniform histogram for N=X.png')

plt.show()

plt.close()

normal\_task()

uniform\_task()

N\_task()

def task\_2():

x = np.random.randn(1000) # generate gaussian random variable

def linear\_transformation():

a = 3

b = 6

x\_values = np.linspace(-5, 20, 1000)

y = a\*x+b #linear transform

y\_pdf = (1/(np.sqrt(2\*np.pi\*a\*\*2)))\*np.exp(-((x\_values-b)\*\*2)/(2\*a\*\*2))

Area\_transformed\_norm = (1000\*(max(y)-min(y))/30)

plt.hist(y, bins=30, label = 'ax+b transformation')

plt.plot(x\_values, Area\_transformed\_norm\*y\_pdf, label='Exact transformed Gaussian')

plt.title('Transformed Standard Gaussian using y = ax + b')

plt.legend()

plt.xlim(-5, 20)

plt.savefig('linear transformation')

plt.show()

plt.close()

def parabolic\_transformation():

x\_values = np.linspace(-1, 10, 1000)

y = x\*\*2

y\_pdf = (1/(np.sqrt(2\*np.pi\*x\_values)))\*np.exp(-x\_values/2)

Area\_transformed\_norm = (1000\*(max(y)-min(y))/30)

plt.figure()

plt.hist(y, bins=30, label = 'x^2 transformation')

plt.plot(x\_values, Area\_transformed\_norm\*y\_pdf, label='Exact pdf p(y)')

plt.title('Transformed Standard Gaussian using y = x^2')

plt.legend()

plt.xlim(-1, 10)

plt.ylim(0,600)

plt.savefig('parabolic transformation')

plt.show()

plt.close()

linear\_transformation()

parabolic\_transformation()

def task\_3():

x = np.random.rand(1000) # generate uniform random variables

x\_values = np.linspace(0, 7, 1000)

exp\_pdf = np.exp(-x\_values)

y = -np.log(x) # Inverse CDF of x. x is uniformly distributed between 0 and 1 so can simplify 1-x to x

Area\_transformed\_exp = (1000\*(max(y)-min(y))/30)

plt.figure()

plt.hist(y, bins=30, label = 'Histogram of inverse CDF pdf')

plt.plot(x\_values, Area\_transformed\_exp\*exp\_pdf, label='exp pdf')

ks\_density = ksdensity(y, width=0.3)

plt.plot(x\_values, Area\_transformed\_exp\*ks\_density(x\_values), label='ksd', color='r')

plt.title('exponential distribution generated from inverse CDF')

plt.legend()

plt.xlim(-1,7)

plt.savefig('Inverse CDF Method')

plt.show()

plt.close()

def task\_4():

# a) choose parameters

alphas = [0.5, 1.5] # Different alpha values to test

betas = [-1, -0.5, 0, 0.5, 1] # Different beta values

# Function to generate X based on the given recipe

def calc\_X(alpha, beta, N):

# Calculate constants

b = (1/alpha)\*np.arctan(beta\*np.tan(np.pi\*alpha/2))

s = (1+beta\*\*2\*np.tan(np.pi\*alpha/2)\*\*2)\*\*(1/(2\*alpha))

# b) Generate U and c) V

U = np.random.uniform(-np.pi/2, np.pi/2, N)

V = np.random.exponential(1, N)

# d) Calculate X

X = s\*(np.sin(alpha\*(U+b))/(np.cos(U))\*\*(1/alpha))\*((np.cos(U-alpha\*(U+b)))/V)\*\*((1-alpha)/alpha)

return X

N = 10000 # Number of samples

count = 0

# Simulate for different alphas and betas

for alpha in alphas:

fig, ax = plt.subplots(len(betas), sharex=False)

count = 0

for beta in betas:

p = ax[count]

X = calc\_X(alpha, beta, N)

outliers = 200

X\_2 = (np.sort(X, axis=1))[outliers/2:-outliers/2]

p.hist(X\_2, bins=500, label=f'Histogram (Œ±={alpha}, Œ≤={beta})')

p.set\_xlabel('X')

p.set\_ylabel('Density')

p.set\_xlim(np.min(X\_2),np.max(X\_2))

#p.set\_ylim(0,100)

p.legend()

count += 1

fig.suptitle('Histograms of a difficult desnity for Œ±={}'.format(alpha))

plt.savefig('Difficult densities for alpha = {}.png'.format(alpha))

plt.show()

plt.close()

def task\_5():

# a) choose parameters

alphas = [0.5, 1.5] # Different alpha values to test

betas = [-1, -0.5, 0, 0.5, 1] # Different beta values

# Function to generate X based on the given recipe

def calc\_X(alpha, beta, N):

b = (1 / alpha) \* np.arctan(beta \* np.tan(np.pi \* alpha / 2))

s = (1 + beta\*\*2 \* np.tan(np.pi \* alpha / 2)\*\*2)\*\*(1 / (2 \* alpha))

U = np.random.uniform(-np.pi / 2, np.pi / 2, N)

V = np.random.exponential(1, N)

X = (

s

\* (np.sin(alpha \* (U + b)) / (np.cos(U))\*\*(1 / alpha))

\* ((np.cos(U - alpha \* (U + b))) / V)\*\*((1 - alpha) / alpha)

)

return X

N = 10000 # Number of samples

bins = 100 # Higher resolution for histograms

for alpha in alphas:

fig, axs = plt.subplots(len(betas), figsize=(10, len(betas) \* 2), sharex=False)

for i, beta in enumerate(betas):

ax = axs[i] if len(betas) > 1 else axs

X = calc\_X(alpha, beta, N)

# Dynamically determine truncation range based on percentiles

x\_min, x\_max = np.percentile(X, [1, 99])

# Apply truncation only if outliers are few

X\_truncated = np.clip(X, x\_min, x\_max)

counts, bin\_edges = np.histogram(X\_truncated, bins=bins, range=(x\_min, x\_max), density=True)

# Adjust first and last bins for outliers

counts[0] += np.sum(X < x\_min) / (N \* (x\_max - x\_min) / bins)

counts[-1] += np.sum(X > x\_max) / (N \* (x\_max - x\_min) / bins)

# Plot histogram

ax.bar(bin\_edges[:-1], counts, width=(bin\_edges[1] - bin\_edges[0]), align='edge', label=f'Œ±={alpha}, Œ≤={beta}')

ax.set\_xlabel('X')

ax.set\_ylabel('Density')

ax.set\_xlim(x\_min, x\_max)

ax.legend()

ax.set\_xticks(np.linspace(x\_min, x\_max, 5)) # Add x-axis ticks

fig.suptitle(f'Difficult Densities for Œ±={alpha}', fontsize=16)

plt.tight\_layout()

plt.savefig(f'Difficult\_densities\_alpha\_{alpha}.png')

plt.show()

def task\_4\_peak\_focus():

# a) choose parameters

alphas = [0.5, 1.5] # Different alpha values to test

betas = [-1, -0.5, 0, 0.5, 1] # Different beta values

# Function to generate X based on the given recipe

def calc\_X(alpha, beta, N):

b = (1 / alpha) \* np.arctan(beta \* np.tan(np.pi \* alpha / 2))

s = (1 + beta\*\*2 \* np.tan(np.pi \* alpha / 2)\*\*2)\*\*(1 / (2 \* alpha))

U = np.random.uniform(-np.pi / 2, np.pi / 2, N)

V = np.random.exponential(1, N)

X = (

s

\* (np.sin(alpha \* (U + b)) / (np.cos(U))\*\*(1 / alpha))

\* ((np.cos(U - alpha \* (U + b))) / V)\*\*((1 - alpha) / alpha)

)

return X

N = 10000 # Number of samples

bins = 100 # Higher resolution for histograms

for alpha in alphas:

fig, axs = plt.subplots(len(betas), figsize=(10, len(betas) \* 4), sharex=False)

for i, beta in enumerate(betas):

ax = axs[i] if len(betas) > 1 else axs

X = calc\_X(alpha, beta, N)

# Main range for histogram

x\_min, x\_max = np.percentile(X, [5, 95]) # Focus closer on the peak

X\_main = X[(X >= x\_min) & (X <= x\_max)]

# Compute histogram for the main range

counts, bin\_edges = np.histogram(X\_main, bins=bins, range=(x\_min, x\_max), density=True)

# Plot zoomed histogram

ax.bar(bin\_edges[:-1], counts, width=(bin\_edges[1] - bin\_edges[0]), align='edge', label=f'Zoomed: Œ±={alpha}, Œ≤={beta}')

# Annotations for the peak

peak\_bin = np.argmax(counts)

peak\_value = bin\_edges[peak\_bin]

ax.annotate(f'Peak: {peak\_value:.2f}', xy=(peak\_value, counts[peak\_bin]),

xytext=(peak\_value + (x\_max - x\_min) \* 0.05, counts[peak\_bin] \* 1.1),

arrowprops=dict(facecolor='black', arrowstyle='->'), fontsize=10)

# Set axis labels and limits

ax.set\_xlim(x\_min, x\_max)

ax.set\_xlabel('X')

ax.set\_ylabel('Density')

ax.legend()

ax.set\_title(f'Zoomed Peak Analysis: Œ±={alpha}, Œ≤={beta}')

fig.suptitle(f'Zoomed Densities for Œ±={alpha}', fontsize=16)

plt.tight\_layout()

plt.savefig(f'Zoomed\_densities\_alpha\_{alpha}.png')

plt.show()

#task\_1()

#task\_2()

#task\_3()

#task\_4()

#task\_5()

#task\_4\_peak\_focus()