National Technical University of Athens
MSc - Data Science and Machine Learning

Data Driven Models
Assignment 2

MSc student
Vasileios Depastas
A.M: 03400131
vasileiosdepastas@mail.ntua.gr

July 2023

1 Problem definition and deriving equations

In this assignment, a rectangular plate of 1m x 1m is heated by a candle residing under it. The candle's flame is positioned under the point $(0.55, 0.45)$, assuming that the plane spans from $(0, 0)$ to $(1, 1)$ in two axes x and y. We are looking to define the temperature field $T(x, y)$ at thermal equilibrium for every position (x, y) of the plane.
The steady state equation that describes the field $T(x, y)$ along the plate at thermal equlibrium is:

$$-\left(\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2}\right) = f(x, y),$$

with $f(x, y)$ being the external heat source due to the candle's presence. It's given by:

$$f(x, y) = 100 \cdot \exp - \frac{(x - 0.55)^2 + (y - 0.45)^2}{r},$$

with $r \sim N(0.05, 0.005^2)$ being a normal random variable. We also consider Dirichlet boundary conditions for the edges, assigning $T = 0$ there. We therefore have $\mu = 0.05$ and $\sigma = 0.005$.
Then, we discretize the plate to a 40 x 40 grid with $h = \Delta x = \Delta y = \frac{1}{40}$. Utilizing the following second order central difference approximation to 2nd derivatives:

$$\frac{\partial^2 T}{\partial x^2} \approx \frac{T(x + h, y) - 2T(x, y) + T(x - h, y)}{h^2}$$
$$\frac{\partial^2 T}{\partial y^2} \approx \frac{T(x, y + h) - 2T(x, y) + T(x, y - h)}{h^2}$$

you can get the finite difference scheme:

$$-T(x + h, y) - T(x, y + h) + 4T(x, y) - T(x - h, y) - T(x, y - h) \approx h^2 f(x, y).$$

Below is the grid designed using matplotlib in Python:

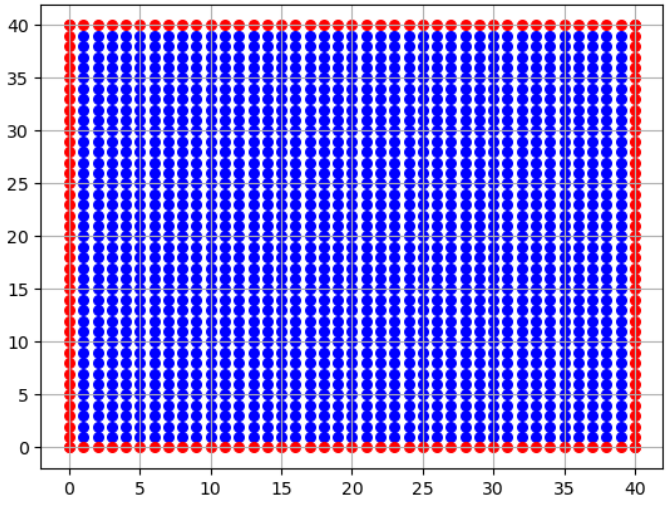

Figure 1: Grid with indexes

We are looking then to derive the linear system of equations

$$K \cdot t = b,$$

where t is a 39x39 vector with the values of the temperature field $T(x, y)$, b is a 39x39 vector with the values $h^2 f(x, y)$ and K is a deterministic (39x39,39x39) array that facilitates the coupling of one point with itself and its neighbors. By the finite difference scheme equation, we can see that for each point in the grid except the border ones where $T = 0$, the b or $h^2 f(x, y)$ value is essentially equal to the sum of -1 values for each neighboring point with non-zero T, i.e. for every neighbor not in the periphery of the grid plus the value of 4 for the interaction with the point itself.

We can imagine discretizing the grid to a total of 41x41 points with distance between them $h = \frac{1}{40}$. Therefore, each point has coordinates (x_n, y_m), where $x_n = \frac{n}{40}$ and $y_m = \frac{m}{40}$, for $n,\ m = 0, 1, 2, ..., 40$. We are then looking to define the $T(x_n, y_m)$ values for all the $n,\ m$. For the periphery of the grid, we have:

$$T(x = 0, y) = 0, \quad T(x = 1, y) = 0, \quad y \in \{y_0, y_1, ..., y_{40}\}$$

$$T(x, y = 0) = 0, \quad T(x, y = 1) = 0, \quad x \in \{x_0, x_1, ..., x_{40}\}.$$

In total, $41 + 40 + 40 + 39 = 160$ points will have $T(x, y) = 0$. And therefore there is a remaining $41 \cdot 41 - 160 = 1521 = 39 \cdot 39$ points with temperature field $T(x_n, y_m)$ with $x_n \in \{x_1, x_2, ..., x_{39}\}$ and $y_m \in \{y_1, y_2, ..., y_{39}\}$.

By randomly sampling a value for r from the standard normal distribution, we can then calculate the b vector that depends from h^2 and $f(x, y)$ and then solve for t in $K \cdot t = b$, where K is a fixed coupling matrix. Below, we have randomly sampled a value for r, in this ocassion $r = 0.05398$ and we get the resulting heatmap for the temperature field along the grid/plate.

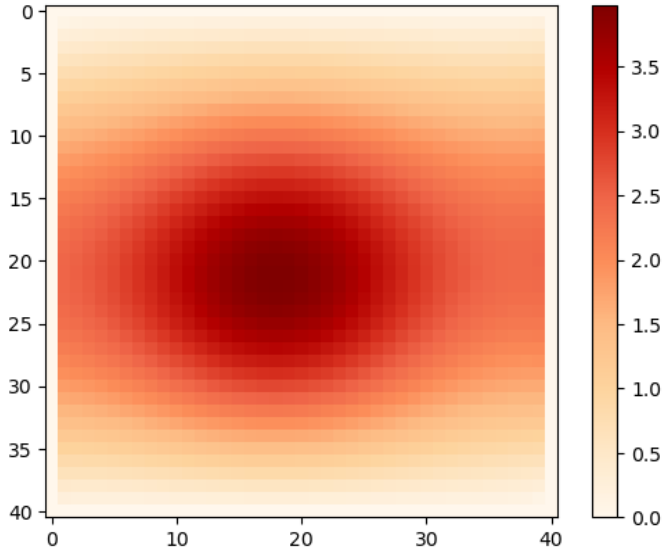

Figure 2: Temperature field heatmap for $r = 0.05398$

We can notice in the heatmap that the field is not centered in the grid due to the fact that the candle is also not centered but on point $(0.55, 0.45)$. Below, we present the code used for the results presented previously.

```
import numpy as np
import matplotlib.pyplot as plt
import itertools


def heatmap2d(arr: np.ndarray):
    plt.imshow(arr, cmap='OrRd')
    plt.colorbar()
    plt.show()


def K_matrix_generation(grid_dim):

    grid_dim_squared = grid_dim**2
    line_multiple = grid_dim - 1
    K = np.zeros((grid_dim_squared, grid_dim_squared))
    # iterate over the lines of the matrix
    for i in range(grid_dim_squared):
        np.fill_diagonal(K, 4)
        # if the left neighbor on same x axis then assign -1
        if (i-1>=0) and (i-1<=grid_dim_squared-1):
            K[i][i-1] = -1
        # if the right neighbor on same x axis then assign -1
        if (i+1>=0) and (i+1<=grid_dim_squared-1):
            K[i][i+1] = -1
        if (i-1-line_multiple>=0) and (i-1-line_multiple<=grid_dim_squared-1):
            K[i][i-1-line_multiple] = -1
        if (i+1+line_multiple>=0) and (i+1+line_multiple<=grid_dim_squared-1):
            K[i][i+1+line_multiple] = -1
    return K


def b_vector_generation(grid_dim, r):

    h = 1 / (grid_dim + 1)
    b = np.zeros((grid_dim, grid_dim))

    for x_minus_1 in range(grid_dim):
        x = (x_minus_1 + 1) * h
        for y_minus_1 in range(grid_dim):
            y = (y_minus_1 + 1) * h
            b[x_minus_1][y_minus_1] = 100 * np.exp(-((x-0.55)**2 + (y-0.45)**2) / r)

    return (h**2) * b


def solve_linear_system(K, b):
    grid_dim = b.shape[0]
    b_flattened = b.flatten()
    t = np.linalg.solve(K, b_flattened)
    return t


# plot the grid

# define the lower and upper limits for x and y
minX, maxX, minY, maxY = 1, 39, 1, 39
# create one-dimensional arrays for x and y
x = np.linspace(minX, maxX, (maxX-minX)+1)
y = np.linspace(minY, maxY, (maxY-minY)+1)

horizontal_peripheral_pts = itertools.product(range(41), (0, 40))
vertical_peripheral_pts = itertools.product((0, 40), range(41))
pts = itertools.product(x, y)
plt.scatter(*zip(*pts), marker='o', s=30, color='blue')
plt.scatter(*zip(*horizontal_peripheral_pts), marker='o', s=30, color='red')
```

```
plt.scatter(*zip(*vertical_peripheral_pts), marker='o', s=30, color='red')

X, Y = np.meshgrid(x, y)
plt.grid()
plt.show()


grid_dimension = 39

K = K_matrix_generation(grid_dimension) # generate K matrix

# solve equation for t, taking a randomly generated r
r_mean_value = 0.05
r_s_value = 0.005
r = np.random.normal(r_mean_value, r_s_value)
print(f'Sampled r={r}.')

b = b_vector_generation(grid_dimension, r=r)
t = solve_linear_system(K, b)

# add to T padding for where T(x,y)=0
t = t.reshape(grid_dimension, grid_dimension)
t = np.pad(t, 1, 'constant')

# ploy heatmap
heatmap2d(t)
```

2 Monte Carlo Simulation

In this section, we perform a Monte Carlo simulation in order to obtain the probability density function of the temperature at the midpoint $(0.5, 0.5)$ of the plate.

We select a large number of experiments, 20000, for the Monte Carlo simulation and we proceed with:

- Sampling an r value from the $N(0.05, 0.005^2)$ distribution.
- Calculate the b vector for the r value.
- Solve the linear system of equations to get the solution for the temperature field matrix t.
- Keep the t value on the point of interest $(0.5, 0.5)$.

The point $(0.5, 0.5)$ is essentially the point on the matrix t with indexes $(20, 20)$. Below is the code used in order to generate the simulated t values on the midpoint.

```
mc_sims = 20000  # number of simulations

t_center = np.array([])  # t values for point (0.5, 0.5) in the center of the plate (in grid has idxs (20, 20))

for sim in range(mc_sims):
    # sample r value
    r = np.random.normal(r_mean_value, r_s_value)

    # calculate t temperatures table
    b = b_vector_generation(grid_dimension, r=r)
    t = solve_linear_system(K, b)
    t = t.reshape(grid_dimension, grid_dimension)
    t = np.pad(t, 1, 'constant')   # add to T padding for where T(x,y)=0 - default padding value is 0 therefore omitted

    t_center = np.append(t_center, t[20, 20])
```

Then, we can plot the histogram of t values for the midpoint of the plate.

```
import seaborn as sns

sns.histplot(
    t_center, kde=True,
    stat="density", kde_kws=dict(cut=3)
)

plt.show()
```

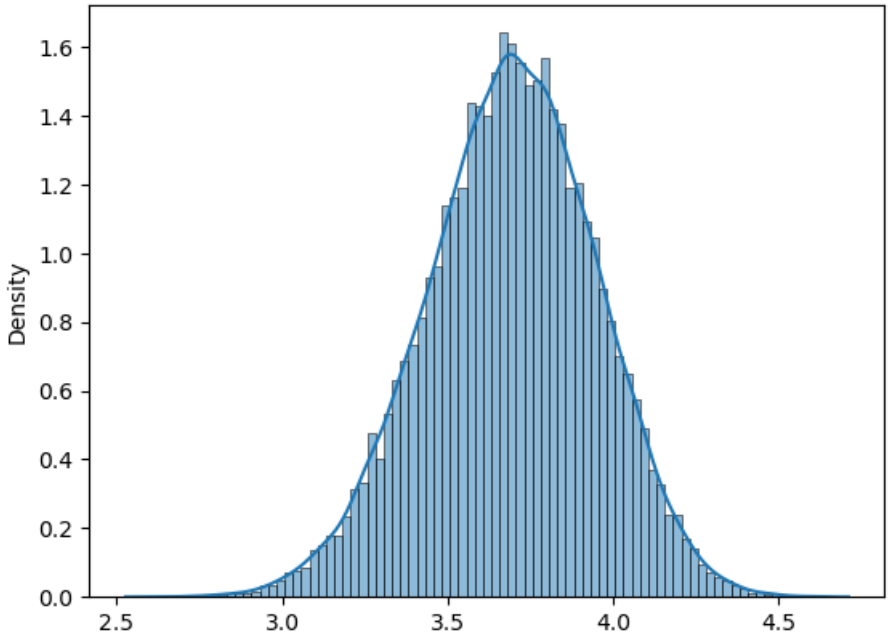

Figure 3: T value probability density for plate's midpoint

For the t values simulated, we calculate the mean and standard deviation.

```
print(f'Mean = {t_center.mean()}')
print(f'Variance = {t_center.var()}')
```

$$\text{Mean} = 3.692$$

$$\text{Variance} = 0.065.$$

3 PCA/POD dimensionality reduction method

In this final section, we perform a small number (here 200) of simulations for different r values and then use these solutions as an initial dataset.
From there we implement the PCA/POD method to reduce the dimensionality of the linear system that describes the problem and perform the Monte Carlo simulation (generating again 20000 values) on the reduced system.

The Monte Carlo simulation in the section 2, took approximately 1hr to complete. This is mainly due to the complexity of calculating the inverse matrix K^{-1}. Here a different strategy is followed in order to reduce the dimensionality of the system we are trying to solve. The reduced number of simulations (200) initially helps define the U reduced dataset with dimensions (39x39, 200), where each column is essentially a solution from the initial simulation. Then, PCA is applied on the dataset in order to reduce its dimension by finding the m first eigenvectors $\phi_1, \phi_2, ..., \phi_m$ that correspond tp the m highest eigenvalues (descending order) of the covariance matrix $U \cdot U^T$. We define

$$\phi = [\phi_1, \phi_2, ..., \phi_m]^T,$$

a vector of dimensions (m, 39x39), and then

$$t \approx \phi^T \cdot \hat{t},$$

where ϕ^T has dimensions (39x39, m) and $\hat{t}$ has dimensions (m, 39x39) and is the approximated solution with m dimensions. Therefore for the initial set of linear equations we get

$$K \cdot t = b$$

$$K\left(\phi^T \cdot \hat{t}\right) = b$$

$$K_{reduced}\hat{t} = b_{reduced},$$

where $K_{reduced} = \phi \cdot K \cdot \phi^T$ is a (m,m) matrix, instead of the initial K matrix that has (39x39, 39x39) dimensions and $b_{reduced} = \phi \cdot b$. Therefore finally

$$t \approx \phi^T \cdot (K^{-1}_{reduced} \cdot \phi \cdot b),$$

where now $K_{reduced}$ should have much smaller dimensions and therefore is faster to compute its inverse.

We first perform the 200 simulations to get the reduced dataset U. Then we apply the PCA method from sklearn to get the first 200 components and the explained variance for including up to x components each time, where we increase x from the value 1 to 200. By plotting the explained variance vs used components, we can see that even 1 component is enough to explain more than 99.98% of the variance and therefore its adequate for our subsequent analysis.

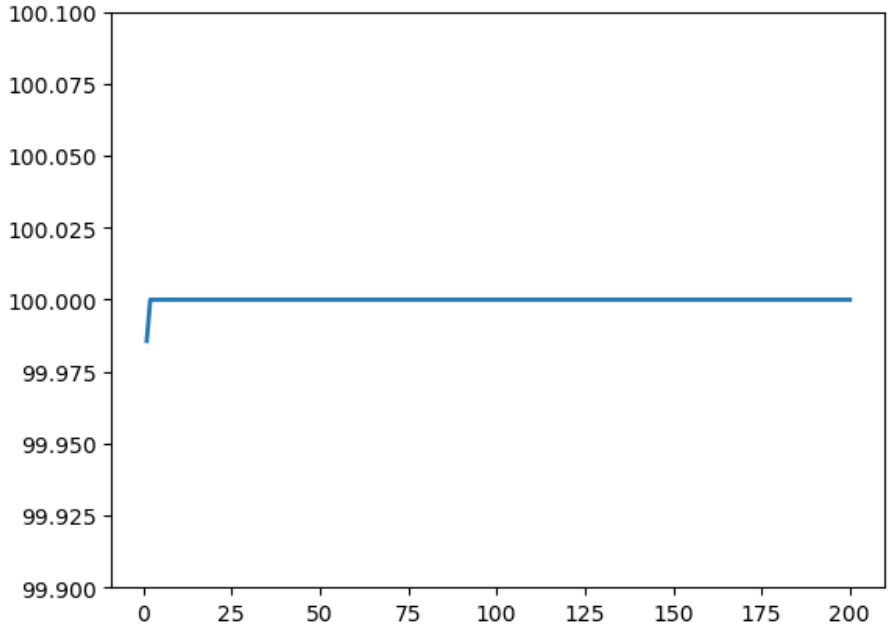

Figure 4: Explained variance vs number of PCA components

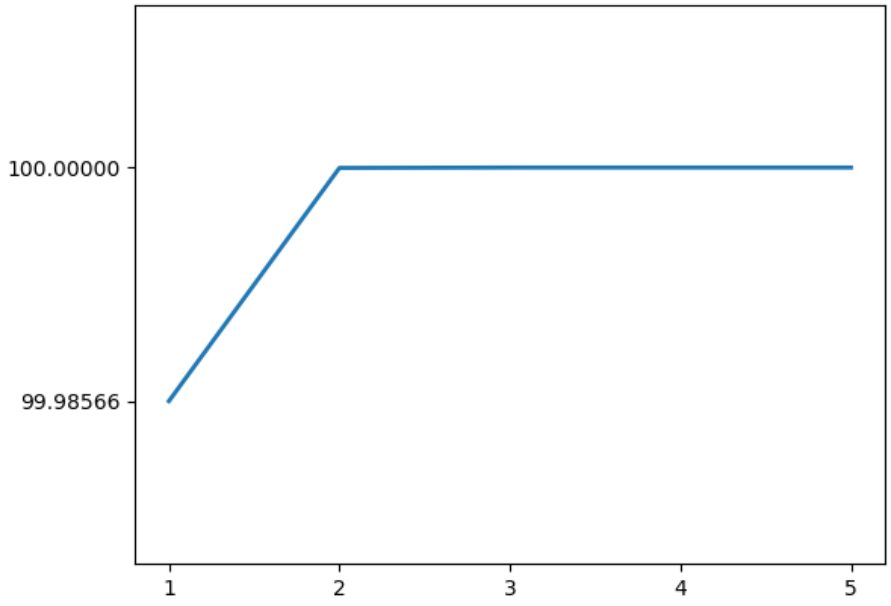

Figure 5: Explained variance vs number of PCA components - zoomed

The code for the above follows below.

```
from sklearn.decomposition import PCA
from matplotlib.ticker import MaxNLocator


pca_sims = 200  # number of simulations
# reduced_data = np.zeros((pca_sims, grid_dimension*grid_dimension))
reduced_data = np.zeros((pca_sims, grid_dimension*grid_dimension))

for sim in range(pca_sims):
    # sample r value
    r = np.random.normal(r_mean_value, r_s_value)

    # calculate t temperatures table
    b = b_vector_generation(grid_dimension, r=r)
    t = solve_linear_system(K, b)

    reduced_data[sim] = t


max_components = 200  # instead of 1521 that is the original length of t

explained_variances = np.zeros(max_components)

for components_kept in range(max_components):
    pca = PCA(n_components=components_kept+1)
    pca.fit(reduced_data)
    explained_variances[components_kept] = 100 * (pca.explained_variance_ratio_.sum())

# plot
fig, ax = plt.subplots()
ax.plot(range(1, max_components + 1), explained_variances, linewidth=2.0)
ax.ticklabel_format(useOffset=False)
ax.set(ylim=(99.9, 100.1))
plt.show()


# plot - zoom in
fig, ax = plt.subplots()
ax.plot([1,2,3,4,5], explained_variances[:5], linewidth=2.0)
ax.set(ylim=(explained_variances.min() - 0.01, 100 + 0.01), yticks=[explained_variances.min(), explained_variances.max()])
ax.ticklabel_format(useOffset=False)
ax.xaxis.set_major_locator(MaxNLocator(integer=True))
plt.show()
```

By selecting $m = 1$ component based on the above, we proceed to create the custom implementation of the PCA method. For 20000 simulated values for the temperature in the center of the plate, we now only need 5 minutes, which is $\frac{1}{20}$th of the time required before. The resulting pdf follows below.

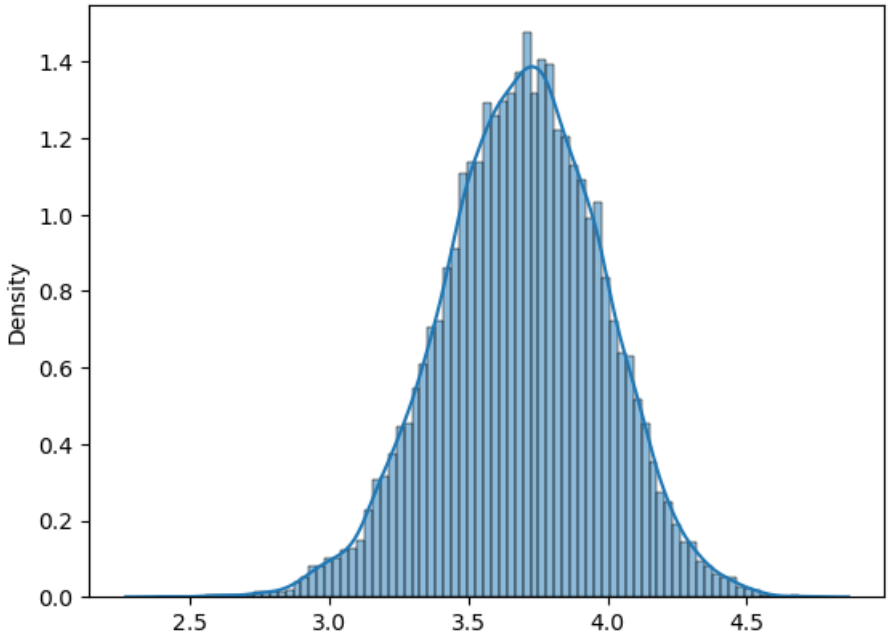

Figure 6: T value probability density for plate's midpoint - PCA method

For the t values simulated, we calculate the mean and standard deviation.

```
print(f'Mean = {t_center_pca.mean()}')
print(f'Variance = {t_center_pca.var()}')
```

$$\text{Mean} = 3.692$$

$$\text{Variance} = 0.083$$

The mean value is identical with the previous method whereas the variance is slightly higher. Next, we put together the two pdfs for comparison. With blue color is the Monte Carlo one with the full K matrix, wheareas with orange is the one with PCA first and the reduced $K_{reduced}$ matrix used. We can see that the two pdfs do not deviate much.

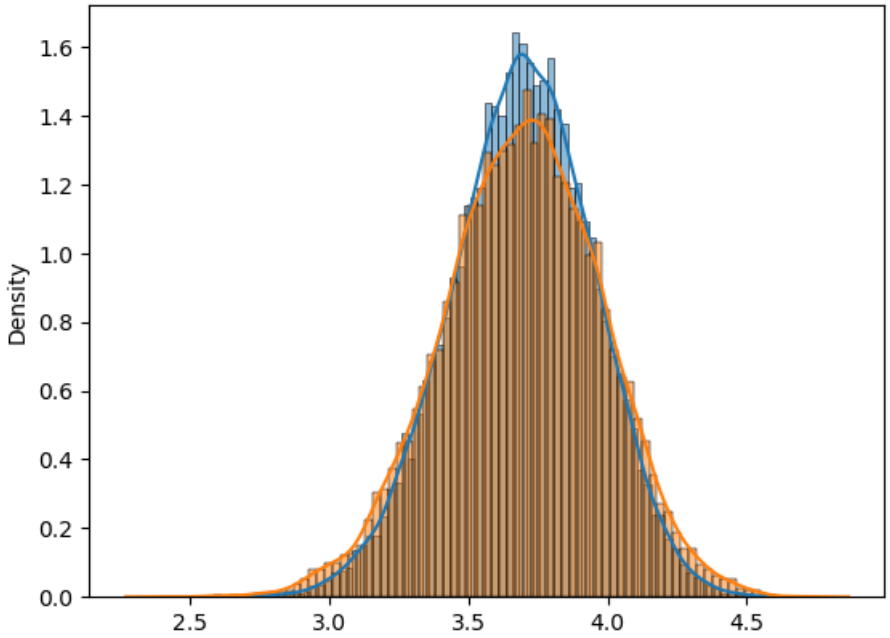

Figure 7: T value probability density for plate's midpoint - methods comparison

```
# find eigenvectors and eigenvalues of reduced data V matrix

covariance_matrix = np.matmul(reduced_data.T, reduced_data)
eigenvalues, eigenvectors = np.linalg.eig(covariance_matrix)

# keep only 1 term and create the Phi vector
phi = np.real(eigenvectors[:, :1].T)

# Create the reduced K matrix
K_reduced = np.matmul(phi, np.matmul(K, phi.T))
K_reduced_inversed = np.linalg.inv(K_reduced)

t_center_pca = np.array([])  # t values for point (0.5, 0.5) in the center of the plate (in grid has idxs (20, 20))

# LOOP HERE FOR MONTE CARLO SIMULATION
for sim in range(mc_sims):

    # sample r value
    r = np.random.normal(r_mean_value, r_s_value)

    # calculate b reduced
    b = b_vector_generation(grid_dimension, r=r)
    b_reduced = np.matmul(phi, np.resize(b, (1521, 1)))

    # t approximated
    t_approximated = np.matmul(K_reduced_inversed, b_reduced)
    t_real = np.matmul(phi.T, t_approximated)

    t_center_pca = np.append(t_center_pca, t_real[39*19+19])

sns.histplot(
    t_center_pca, kde=True,
    stat="density", kde_kws=dict(cut=3)
)
```

```
plt.show()

#compare the two pdfs


sns.histplot(
    t_center, kde=True,
    stat="density", kde_kws=dict(cut=3)
)

sns.histplot(
    t_center_pca, kde=True,
    stat="density", kde_kws=dict(cut=3)
)

plt.show()
```