homework-02

July 1, 2025

1 Homework 2

1.1 References

- Lectures 4-8 (inclusive).

1.2 Instructions

- Type your name and email in the "Student details" section below.
- Develop the code and generate the figures you need to solve the problems using this notebook.
- For the answers that require a mathematical proof or derivation you should type them using latex. If you have never written latex before and you find it exceedingly difficult, we will likely accept handwritten solutions.
- The total homework points are 100. Please note that the problems are not weighed equally.

[ ]:
```
import matplotlib.pyplot as plt
%matplotlib inline
import matplotlib_inline
matplotlib_inline.backend_inline.set_matplotlib_formats('svg')
import seaborn as sns
sns.set_context("paper")
sns.set_style("ticks")

import numpy as np
import scipy
import scipy.stats as st
import urllib.request
import os

def download(
    url : str,
    local_filename : str = None
):
    """Download a file from a url.

    Arguments
    url            -- The url we want to download.
    local_filename -- The filemame to write on. If not
                      specified
```

```
    """
    if local_filename is None:
        local_filename = os.path.basename(url)
    urllib.request.urlretrieve(url, local_filename)
```

1.3 Student details

- **First Name:** Hannah
- **Last Name:** Moskios
- **Email:** hmoskios@purdue.edu
- **Used generative AI to complete this assignment (Yes/No):** Yes
- **Which generative AI tool did you use (if applicable)?:** ChatGPT 4o

1.4 Problem 1 - Joint probability mass function of two discrete random variables

Consider two random variables X and Y. X takes values $\{0, 1, \ldots, 4\}$ and Y takes values $\{0, 1, \ldots, 8\}$. Their joint probability mass function, can be described using a matrix:

[ ]:
```
P = np.array(
    [
        [0.03607908, 0.03760034, 0.00503184, 0.0205082 , 0.01051408,
         0.03776221, 0.00131325, 0.03760817, 0.01770659],
        [0.03750162, 0.04317351, 0.03869997, 0.03069872, 0.02176718,
         0.04778769, 0.01021053, 0.00324185, 0.02475319],
        [0.03770951, 0.01053285, 0.01227089, 0.0339596 , 0.02296711,
         0.02187814, 0.01925662, 0.0196836 , 0.01996279],
        [0.02845139, 0.01209429, 0.02450163, 0.00874645, 0.03612603,
         0.02352593, 0.00300314, 0.00103487, 0.04071951],
        [0.00940187, 0.04633153, 0.01094094, 0.00172007, 0.00092633,
         0.02032679, 0.02536328, 0.03552956, 0.01107725]
    ]
)
```

The rows of the matrix correspond to the values of X and the columns to the values of Y. So, if you wanted to find the probability of $p(X = 2, Y = 3)$ you would do:

[ ]:
```
print(f"p(X=2, Y=3) = {P[2, 3]:.3f}")
```

```
p(X=2, Y=3) = 0.034
```

A. Verify that all the elements of P sum to one, i.e., that $\sum_{x,y} p(X = x, Y = y) = 1$.

[ ]:
```
# Your code here
sum_of_elements = np.sum(P)
print(f"Sum of elements = {sum_of_elements:.3f}")
```

```
Sum of elements = 1.000
```

B. Find the marginal probability density of X:

$$p(x) = \sum_y p(x, y).$$

You can represent this as a 5-dimensional vector.

[ ]:
```
# Hint, you can do this in one line if you read this:
# help(np.sum)
p_x = np.sum(P, axis=1)
print("p(x) = ", np.round(p_x, 3))
```

```
p(x) =  [0.204 0.258 0.198 0.178 0.162]
```

C. Find the marginal probability density of Y. This is a 9-dimensional vector.

[ ]:
```
# Your code here
p_y = np.sum(P, axis=0)
print("p(y) = ", np.round(p_y, 3))
```

```
p(y) =  [0.149 0.15  0.091 0.096 0.092 0.151 0.059 0.097 0.114]
```

D. Find the expectation and variance of X and Y.

[ ]:
```
# Your code here
# Define the possible values that X and Y can take
x_vals = np.arange(5)
y_vals = np.arange(9)

# Compute the expectation and variance of X
E_X = np.sum(x_vals * p_x)
V_X = np.sum((x_vals ** 2) * p_x) - E_X ** 2

# Compute the expectation and variance of Y
E_Y = np.sum(y_vals * p_y)
V_Y = np.sum((y_vals ** 2) * p_y) - E_Y ** 2

# Print the results
print(f"E[X] = {E_X:.3f}")
print(f"V[X] = {V_X:.3f}\n")
print(f"E[Y] = {E_Y:.3f}")
print(f"V[Y] = {V_Y:.3f}")
```

```
E[X] = 1.835
V[X] = 1.872

E[Y] = 3.693
V[Y] = 7.191
```

E. Find the expectation of $E[X + Y]$.

[ ]:
```
# Your code here
# E[X + Y] = E[X] + E[Y]
E_XY = E_X + E_Y
print(f"E[X + Y] = {E_XY:.2f}")
```

```
E[X + Y] = 5.53
```

F. Find the covariance of X and Y. Are the two variable correlated? If yes, are they positively or negatively correlated?

[ ]:
```
# Your code here
# Covariance
C_XY = 0.0
for x in range(x_vals.shape[0]):
    for y in range(y_vals.shape[0]):
        C_XY += (x - E_X) * (y - E_Y) * P[x, y]

print(f"C[X,Y] = {C_XY:.2f}\n")
print("Yes, the two variables are correlated since C[X,Y] does not equal 0.")
print("Since C[X,Y] is greater than 0, X and Y are positively correlated.")
```

```
C[X,Y] = 0.32

Yes, the two variables are correlated since C[X,Y] does not equal 0.
Since C[X,Y] is greater than 0, X and Y are positively correlated.
```

[ ]:
```
# Correlation coefficient
rho_XY = C_XY / (np.sqrt(V_X) * np.sqrt(V_Y))
print(f"rho_XY = {rho_XY:.2f}\n")
```

```
rho_XY = 0.09
```

G. Find the variance of $X + Y$.

[ ]:
```
# Your code here
# V[X + Y] = V[X] + V[Y] + 2C[X,Y]
V_XY = V_X + V_Y + 2 * C_XY
print(f"V[X + Y] = {V_XY:.2f}")
```

```
V[X + Y] = 9.70
```

J. Find the probability that $X + Y$ is less than or equal to 5. That is, find $p(X + Y \leq 5)$. Hint: Use two for loops to go over all the combinations of X and Y values, check if $X + Y \leq 5$, and sum up the probabilities.

[ ]:
```
# Your code here
prob = 0.0
for x in range(x_vals.shape[0]):
    for y in range(y_vals.shape[0]):
```

```
        if x + y <= 5:
            prob += P[x, y]

print(f"p(X + Y <= 5) = {prob:.3f}")
```

```
p(X + Y <= 5) = 0.535
```

1.5 Problem 2 - Zero correlation does not imply independence

The purpose of this problem is to show that zero correlation does not imply independence. Consider the random variable X and Y following a standard normal distribution. Define the random variable as $Z = X^2 + 0.01 \cdot Y$. You will show that the correlation between X and Z is zero even though they are not independent.

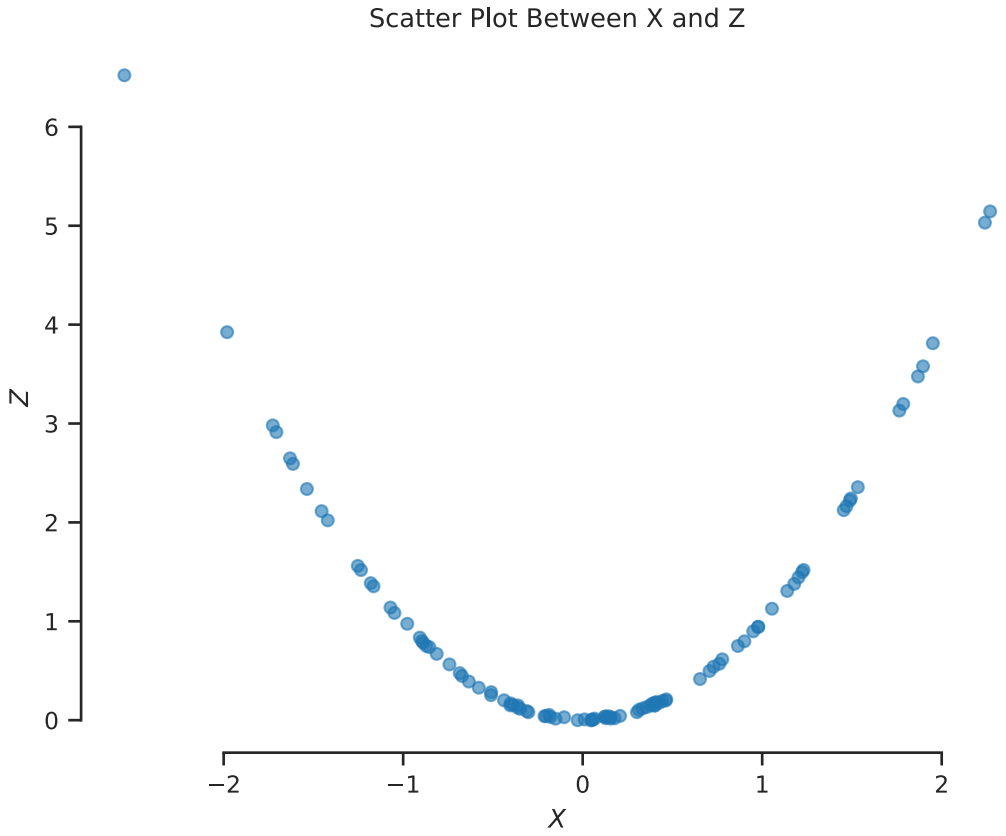
A. Take 100 samples of X and Z using numpy or scipy. Hint: First sample X and Y and use the samples to get Z.

[ ]:
```
# Your code here
N = 100
np.random.seed(0)
X = np.random.normal(loc=0, scale=1, size=N)
Y = np.random.normal(loc=0, scale=1, size=N)
Z = X ** 2 + 0.01 * Y

np.set_printoptions(suppress=True)  # Disable scientific notation
print(f"X = {np.round(X, 3)}\n")
print(f"Z = {np.round(Z, 3)}")
```

```
X = [ 1.764  0.4    0.979  2.241  1.868 -0.977  0.95  -0.151 -0.103  0.411
  0.144  1.454  0.761  0.122  0.444  0.334  1.494 -0.205  0.313 -0.854
 -2.553  0.654  0.864 -0.742  2.27  -1.454  0.046 -0.187  1.533  1.469
  0.155  0.378 -0.888 -1.981 -0.348  0.156  1.23   1.202 -0.387 -0.302
 -1.049 -1.42  -1.706  1.951 -0.51  -0.438 -1.253  0.777 -1.614 -0.213
 -0.895  0.387 -0.511 -1.181 -0.028  0.428  0.067  0.302 -0.634 -0.363
 -0.672 -0.36  -0.813 -1.726  0.177 -0.402 -1.63   0.463 -0.907  0.052
  0.729  0.129  1.139 -1.235  0.402 -0.685 -0.871 -0.579 -0.312  0.056
 -1.165  0.901  0.466 -1.536  1.488  1.896  1.179 -0.18  -1.071  1.054
 -0.403  1.222  0.208  0.977  0.356  0.707  0.011  1.786  0.127  0.402]

Z = [ 3.131  0.147  0.945  5.031  3.476  0.975  0.899  0.015  0.03   0.183
  0.039  2.124  0.571  0.034  0.194  0.119  2.242  0.041  0.104  0.739
  6.522  0.416  0.75   0.564  5.145  2.114 -0.002  0.054  2.356  2.163
  0.016  0.148  0.781  3.924  0.115  0.031  1.519  1.444  0.154  0.08
  1.085  2.021  2.913  3.812  0.284  0.201  1.56   0.616  2.592  0.041
  0.801  0.167  0.253  1.386 -0.     0.177  0.016  0.081  0.391  0.127
  0.447  0.149  0.671  2.981  0.019  0.17   2.648  0.199  0.835  0.006
  0.541  0.02   1.307  1.518  0.152  0.476  0.75   0.328  0.093  0.003
  1.354  0.798  0.21   2.338  2.221  3.578  1.378  0.033  1.139  1.127
  0.15   1.497  0.043  0.942  0.132  0.498  0.008  3.198  0.038  0.175]
```

B. Do the scatter plot between X and Z.

[ ]:
```
# Your code here
fig, ax = plt.subplots()
ax.scatter(X, Z, s=20, marker="o", alpha=0.6)
ax.set_xlabel(r"$X$")
ax.set_ylabel(r"$Z$")
ax.set_title("Scatter Plot Between X and Z")
sns.despine(trim=True);
```



C. Use the scatter plot to argue that X and Z are not independent.

Answer: The scatter plot shows a clear parabolic relation between X and Z. The points are clustered tightly around the curve $Z = X^2$. In other words, Z increases parabolically as X moves away from 0.

If X and Z were independent, we would expect the points to be diffuse and pattern-free. Instead, the data lies in a deterministic curve. The visible functional link between X and Z confirms that they are not independent.

D. Use the samples you took to estimate the variance of Z.

[ ]:
```
# Your code here
# For large sample sizes, the difference between the population variance and
# sample variance is insigificant. As per the professor's instructions, we
# will use the sample variance (unbiased) for this problem.

# Unbiased/sample variance (ddof=1 means divide by N-1)
V_Z = np.var(Z, ddof=1)
print(f"V(Z) = {V_Z:.4f}\n")
```

```
V(Z) = 1.7044
```

E. Use the samples you took to estimate the covariance between X and Z.

[ ]:
```
# Your code here
# Unbiased/sample estimate of covariance (np.cov uses ddof=1 by default)
C_XZ = np.cov(X, Z)[0, 1]
print(f"C[X,Z] = {C_XZ:.4f}\n")
```

```
C[X,Z] = 0.1293
```

F. Use the results above to find the correlation between X and Z.

[ ]:
```
# Your code here
# Corr[X,Z] = C[X,Z] / sqrt(V[X]*V[Z])
V_X = np.var(X, ddof=1)
corr_XZ = C_XZ / np.sqrt(V_X * V_Z)
print(f"Corr[X,Z] = {corr_XZ:.5f}")
```

```
Corr[X,Z] = 0.09774
```

G. The correlation coefficient you get may not be very close to zero. This is due to the fact that we estimate it with Monte Carlo averaging. To get a better estimate, we can increase the number of samples. Try increasing the number of samples to 1000 and see if the correlation coefficient gets closer to zero.

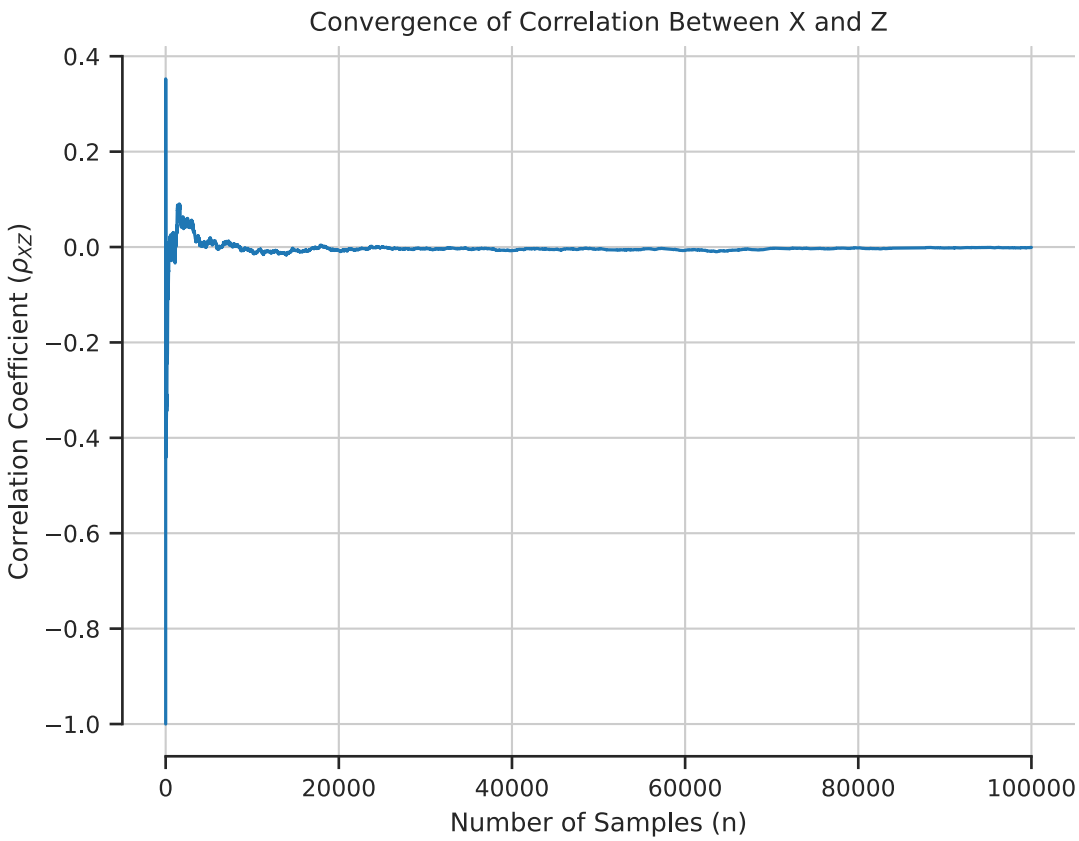
[ ]:
```
# Your code here
# Generate the random samples
N_G = 1000
np.random.seed(0)
X_G = np.random.normal(loc=0, scale=1, size=N_G)
Y_G = np.random.normal(loc=0, scale=1, size=N_G)
Z_G = X_G ** 2 + 0.01 * Y_G

# Compute the correlation coefficient
V_X_G = np.var(X_G, ddof=1)
V_Z_G = np.var(Z_G, ddof=1)
C_XZ_G = np.cov(X_G, Z_G)[0, 1]
corr_XZ_G = C_XZ_G / np.sqrt(V_X_G * V_Z_G)
```

```
print(f"Corr[X,Z] = {corr_XZ_G:.5f}")
print("The correlation coefficient gets closer to 0 when we increase the number␣
 ↪of samples.")
```

```
Corr[X,Z] = -0.04158
The correlation coefficient gets closer to 0 when we increase the number of
samples.
```

H. Let's do a more serious estimation of Monte Carlo convergence. Take 100,000 samples of X and Z. Write code that estimates the correlation between X and Z using the first n samples for $n = 1, 2, \ldots, 100,000$. Plot the estimates as a function of n. What do you observe? How many samples do you need to get a good estimate of the correlation?

[ ]:
```
# Your code here
# Generate 100,000 random samples of X and Z
N_H = 100000
np.random.seed(1234)
X_H = np.random.normal(loc=0, scale=1, size=N_H)
Y_H = np.random.normal(loc=0, scale=1, size=N_H)
Z_H = X_H ** 2 + 0.01 * Y_H

# Initialize the correlation array
corr_XZ_H = np.zeros(N_H)

# Estimate the correlation for different numbers of samples
for n in range(1, N_H + 1):
    X_vals = X_H[:n]
    Z_vals = Z_H[:n]
    corr_XZ_H[n - 1] = np.corrcoef(X_vals, Z_vals)[0, 1]

# Plot the correlation as a function of n
fig, ax = plt.subplots()
n_vals = np.arange(1, N_H + 1)
ax.plot(n_vals, corr_XZ_H)
ax.set_xlabel("Number of Samples (n)")
ax.set_ylabel(r"Correlation Coefficient ($\rho_{XZ}$)")
ax.set_title("Convergence of Correlation Between X and Z")
ax.grid(True)
sns.despine(trim=True);
```



The plots show that the correlation estimate fluctuates significantly for small sample sizes. As n increases, the estimate stabilizes and converges to 0. Since X and Z are uncorrelated, it makes sense that the correlation coefficient converges to 0.

The correlation estimate becomes relatively stable when the sample size reaches approximately $20,000$. Beyond this point, additional samples provide diminishing returns in terms of accuracy. Therefore, around $20,000$ samples are needed to obtain a reliable estimate of the correlation coefficient.

1.6 Problem 3 - Creating a stochastic model for the magnetic properties of steel

The magnetic properties of steel are captured in the so-called $B-H$ curve, which connects the magnetic field H to the magnetic flux density B. The $B-H$ curve is a nonlinear function typically measured in the lab. It appears in Maxwell's equations and is, therefore, crucial in the design of electrical machines.

The shape of the $B-H$ curve depends on the manufacturing process of the steel. As a result, the $B-H$ differs across different suppliers but also across time for the same supplier. The goal of this problem is to guide you through the process of creating a stochastic model for the $B-H$ curve using real data. Such a model is the first step when we do uncertainty quantification for the design of electrical machines. Once constructed, the stochastic model can generate random samples of the $B-H$ curve. We can then propagate the uncertainty in the $B-H$ curve through Maxwell's

equations to quantify the uncertainty in the performance of the electrical machine.

Let's use some actual manufacturer data to visualize the differences in the $B-H$ curve across different suppliers. The data are here. Explaining how to upload data on Google Colab will take a while. We will do it in the next homework set. You should know that the data file `B_data.csv` needs to be in the same working directory as this Jupyter Notebook. I have written some code that allows you to put the data file in the right place without too much trouble. Run the following:

[ ]:
```
url = "https://github.com/PredictiveScienceLab/data-analytics-se/raw/master/
 ↪lecturebook/data/B_data.csv"
download(url)
```

If everything worked well, then the following will work:

[ ]:
```
B_data = np.loadtxt('B_data.csv')
B_data
```

[ ]:
```
array([[0.        , 0.00490631, 0.01913362, …, 1.79321352, 1.79337681,
        1.79354006],
       [0.        , 0.00360282, 0.01426636, …, 1.8367998 , 1.83697627,
        1.83715271],
       [0.        , 0.00365133, 0.01433438, …, 1.77555287, 1.77570402,
        1.77585514],
       …,
       [0.        , 0.00289346, 0.01154411, …, 1.7668308 , 1.76697657,
        1.76712232],
       [0.        , 0.00809884, 0.03108513, …, 1.7774044 , 1.77756225,
        1.77772007],
       [0.        , 0.00349638, 0.0139246 , …, 1.76460358, 1.76474439,
        1.76488516]])
```

The shape of this dataset is:

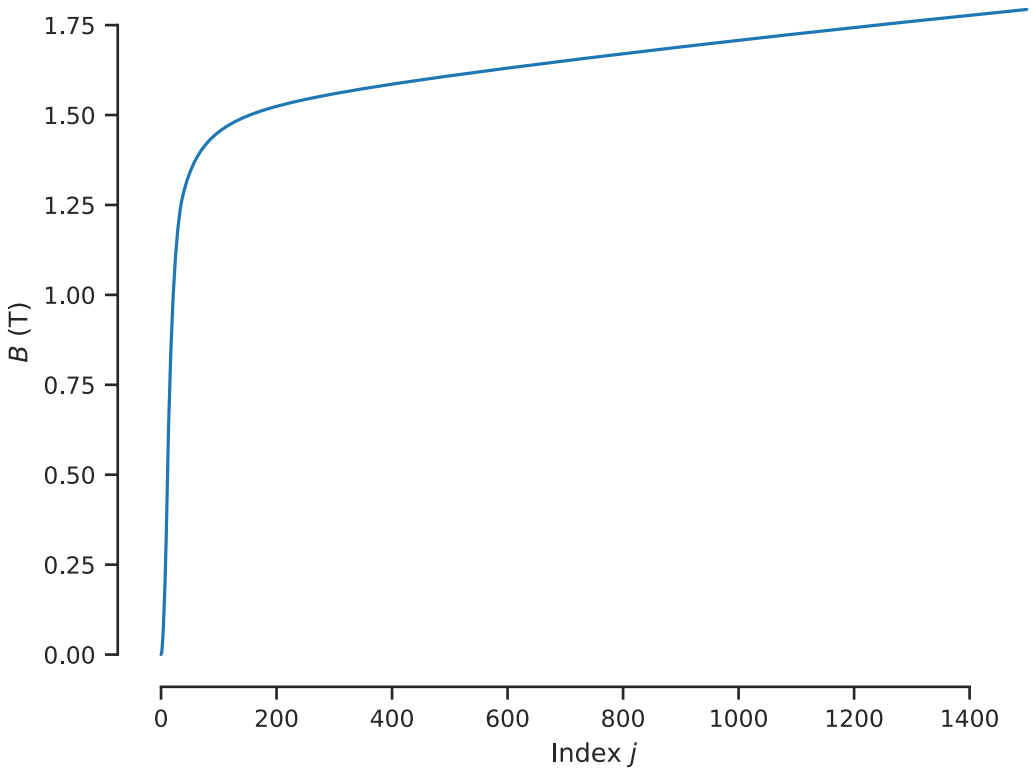
[ ]:
```
B_data.shape
```

[ ]:
```
(200, 1500)
```

The rows (200) correspond to different samples of the $B-H$ curves (suppliers and times). The columns (1500) correspond to different values of H. That is, the i,j element is the value of B at the specific value of H, say H_j. The values of H are equidistant and identical; we will ignore them in this analysis. Let's visualize some of the samples.

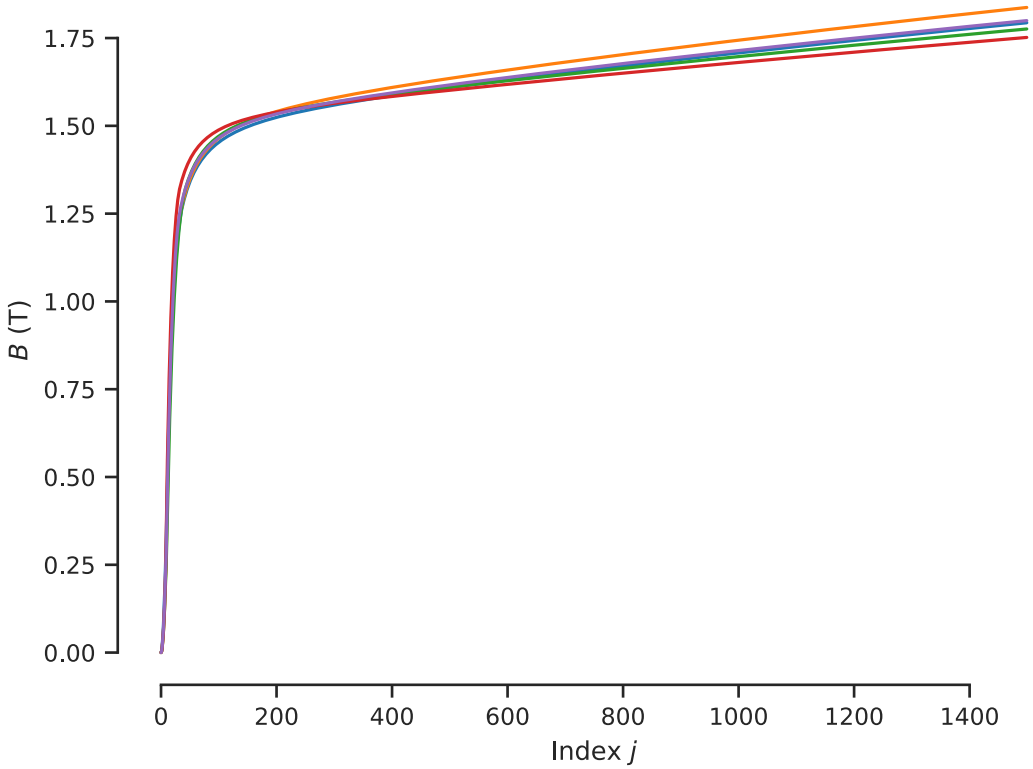
Here is one sample:

[ ]:
```
fig, ax = plt.subplots()
ax.plot(B_data[0, :])
ax.set_xlabel(r"Index $j$")
ax.set_ylabel(r"$B$ (T)")
sns.despine(trim=True);
```



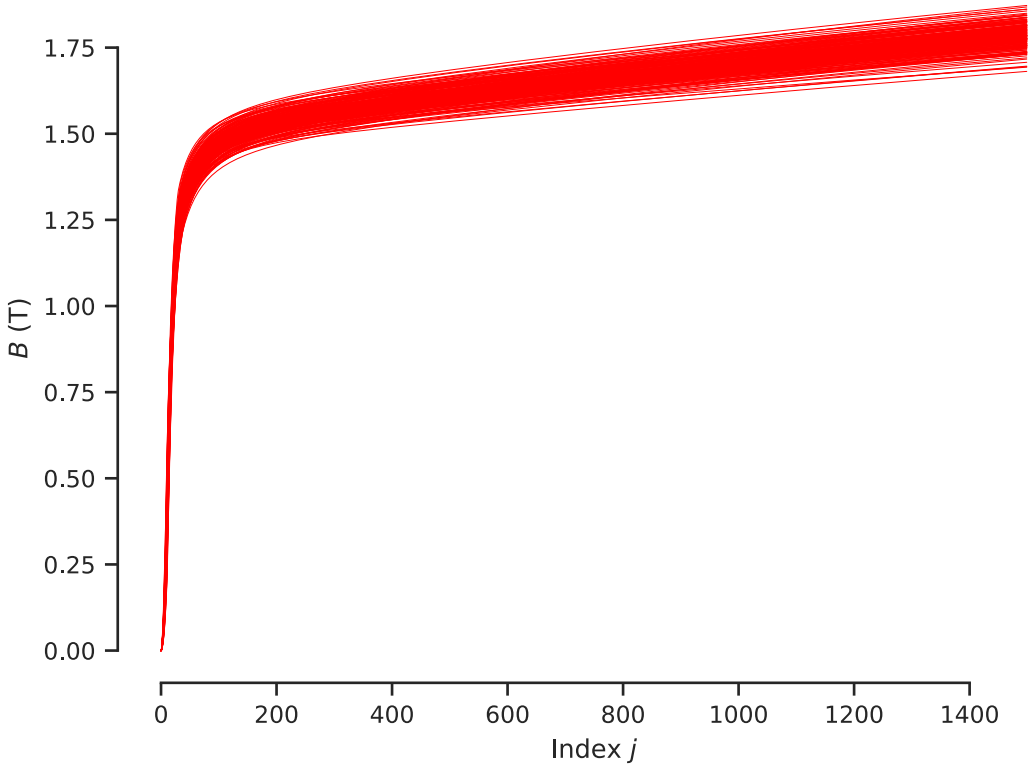
Here are five samples:

[ ]:
```
fig, ax = plt.subplots()
ax.plot(B_data[:5, :].T)
ax.set_xlabel(r"Index $j$")
ax.set_ylabel(r"$B$ (T)")
sns.despine(trim=True);
```



Here are all the samples:

[ ]:
```
fig, ax = plt.subplots()
ax.plot(B_data[:, :].T, 'r', lw=0.1)
ax.set_xlabel(r"Index $j$")
ax.set_ylabel(r"$B$ (T)")
sns.despine(trim=True);
```

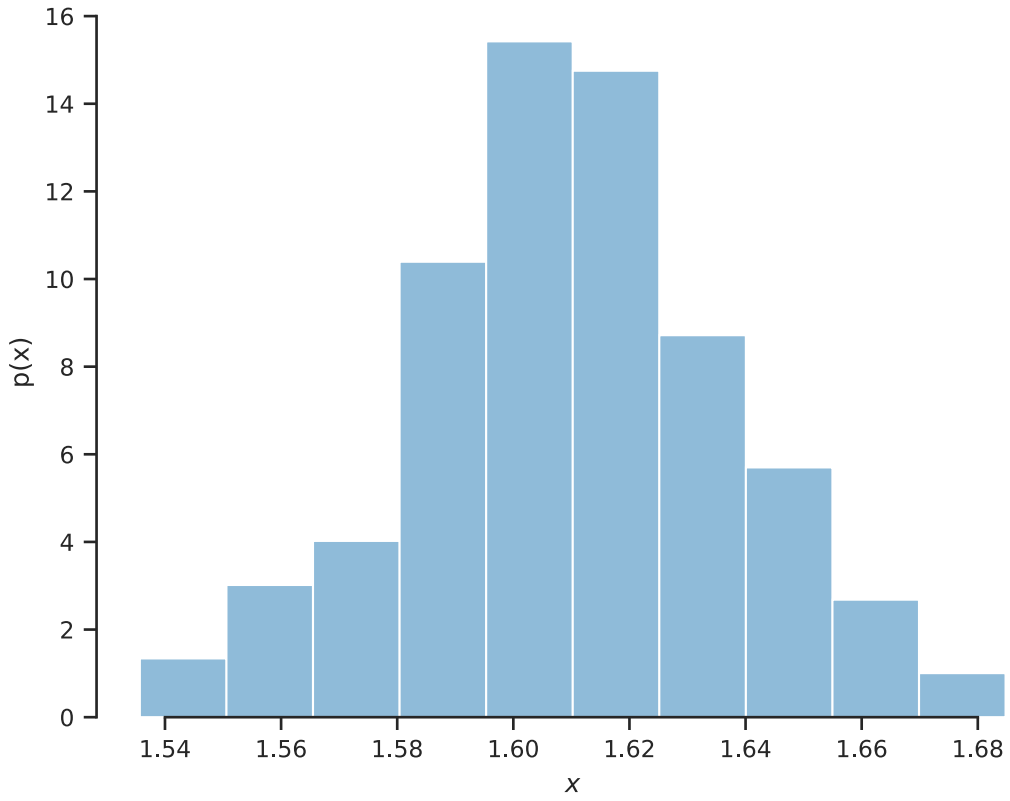


A. We are going to start by studying the data at only one index. Say index $j = 500$. Let's define a random variable

$$X = B(H_{500}),$$

for this reason. Extract and do a histogram of the data for X:

[ ]:
```
X_data = B_data[:, 500]
fig, ax = plt.subplots()
ax.hist(X_data, alpha=0.5, density=True)
ax.set_xlabel(r"$x$")
ax.set_ylabel(r"p(x)")
sns.despine(trim=True);
```



This looks like a Gaussian $N(\mu_{500}, \sigma^2_{500})$. Let's try to find a mean and variance for that Gaussian. A good choice for the mean is the empirical average of the data:

$$\mu_j = \frac{1}{N}\sum_{i=1}^{N} B_{ij}.$$

By the law of large numbers, this is a good approximation of the true mean as $N \to \infty$. Later we will learn that this is also the *maximum likelihood* estimate of the mean.

So, the mean is:

[ ]:
```
mu_500 = X_data.mean()
print(f"mu_500 = {mu_500:.2f}")
```

```
mu_500 = 1.61
```

Similarly, for the variance a good choice is the empirical variance defined by:

$$\sigma_j^2 = \frac{1}{N}\sum_{i=1}^{N}(B_{ij} - \mu_j)^2.$$

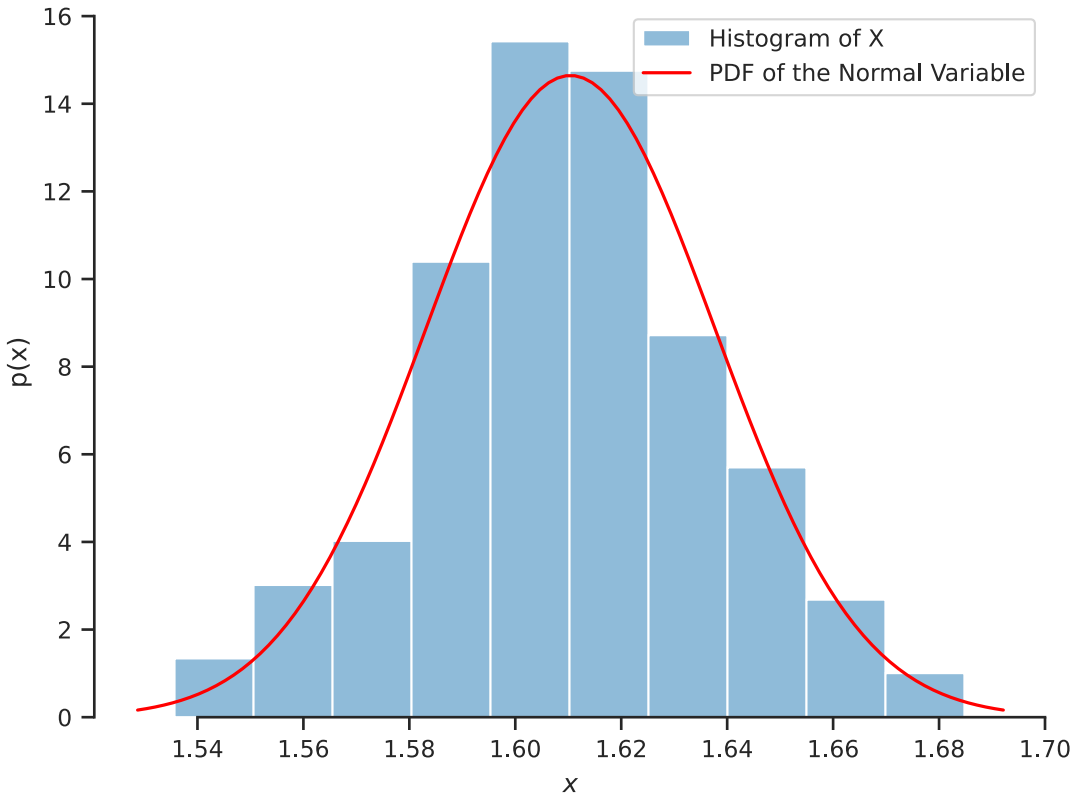
This also converges to the true variance as $N \to \infty$. Here it is:

[ ]:
```
sigma2_500 = np.var(X_data)
print(f"sigma_500 = {sigma2_500:.2e}")
```

```
sigma_500 = 7.42e-04
```

Repeat the plot of the histogram of X along with the PDF of the normal variable we have just identified using the functionality of `scipy.stats`.

[ ]:
```
# Your code here
# Use scipy.stats to generate the normal distribution
sigma_500 = np.sqrt(sigma2_500)
X = st.norm(loc=mu_500, scale=sigma_500)
x_vals = np.linspace(mu_500 - 3 * sigma_500, mu_500 + 3 * sigma_500, 100)

# Plot the histogram and PDF
fig, ax = plt.subplots()
ax.hist(X_data, alpha=0.5, density=True, label="Histogram of X")
ax.plot(x_vals, X.pdf(x_vals), 'r', label="PDF of the Normal Variable")
ax.legend(loc="best")
ax.set_xlabel(r"$x$")
ax.set_ylabel(r"p(x)")
sns.despine(trim=True);
```



B. Using your normal approximation to the PDF of X, find the probability that $X = B(H_{500})$ is greater than 1.66 T.

[ ]:
```
# Your code here
p_greater_than_166 = 1 - X.cdf(1.66)
print(f"p(X > 1.66 T) = {p_greater_than_166:.3f}")
```

```
p(X > 1.66 T) = 0.034
```

C. Let us now consider another random variable

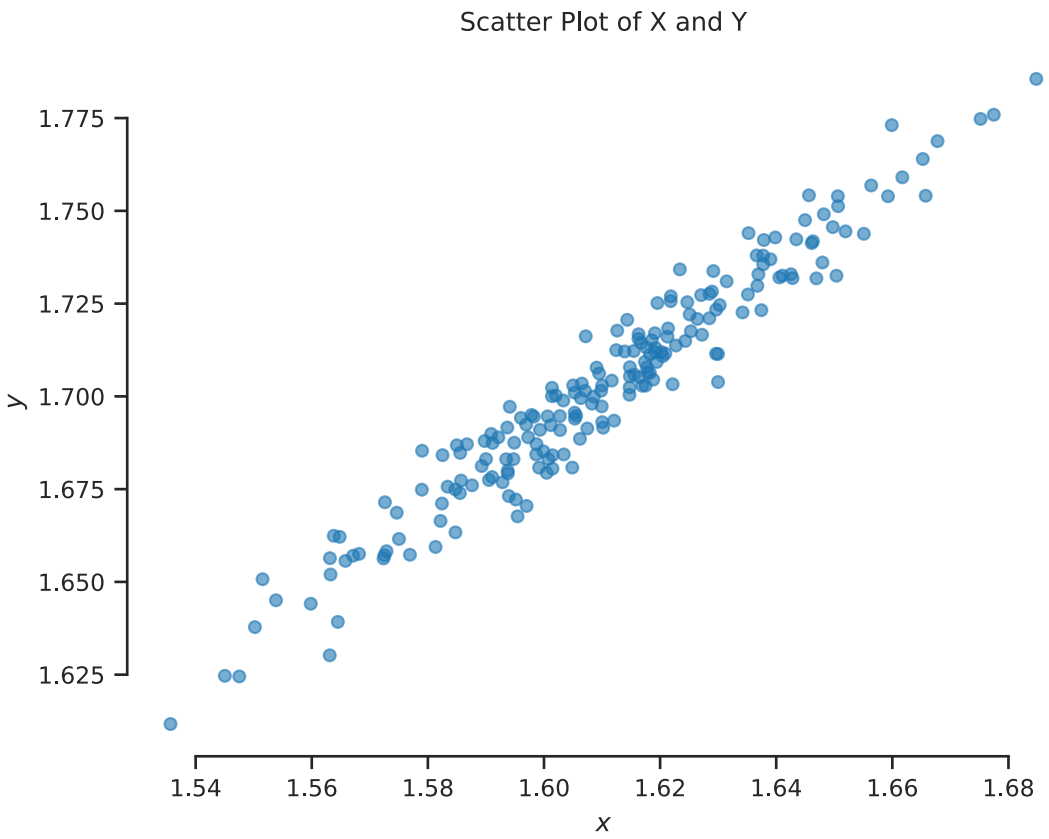
$$Y = B(H_{1000}).$$

Isolate the data for this as well:

[ ]:
```
Y_data = B_data[:, 1000]
```

Do the `scatter` plot of X and Y:

[ ]:
```
# Your code here
fig, ax = plt.subplots()
ax.scatter(X_data, Y_data, s=20, marker="o", alpha=0.6)
ax.set_xlabel(r"$x$")
ax.set_ylabel(r"$y$")
ax.set_title("Scatter Plot of X and Y")
sns.despine(trim=True);
```



D. From the scatter plot, it looks like the random vector

$$\mathbf{X} = (X, Y),$$

follows a multivariate normal distribution. What would be the mean and covariance of the distribution? First, organize the samples of X and Y in a matrix with the number of rows being the number of samples and two columns (one corresponding to X and one to Y).

[ ]:
```
XY_data = np.hstack([X_data[:, None], Y_data[:, None]])
```

In case you are wondering, the code above takes two 1D numpy arrays of the same size and puts them in a two-column numpy array. The first column is the first array, the second column is the second array. The result is a 2D numpy array. We take sampling averages over the first axis of the array.

The mean vector is:

[ ]:
```
mu_XY = np.mean(XY_data, axis=0)
print(f"mu_XY = {mu_XY}")
```

```
mu_XY = [1.61041566 1.70263681]
```

The covariance matrix is trickier. We have already discussed how to find the diagonals of the covariance matrix (it is simply the variance). For the off-diagonal terms, this is the formula that is being used:

$$C_{jk} = \frac{1}{N}\sum_{i=1}^{N}(B_{ij} - \mu_j)(B_{ik} - \mu_k).$$

This formula converges as $N \to \infty$. Here is the implementation:

[ ]:
```
# Careful with np.cov because it requires you to transpose the matrix we␣
 ↪defined in class
C_XY = np.cov(XY_data.T)
print(f"C_XY =")
print(C_XY)
```

```
C_XY =
[[0.00074572 0.00082435]
 [0.00082435 0.00096729]]
```

Use the covariance matrix `C_XY` to find the correlation coefficient between X and Y.

[ ]:
```
# Your code here
# Correlation coefficient = C[X,Y] / (sigma_X * sigma_Y)
corr_XY = C_XY[0, 1] / np.sqrt(C_XY[0, 0] * C_XY[1, 1])
print(f"Corr[X,Y] = {corr_XY:.3f}")
```

```
Corr[X,Y] = 0.971
```

Are the two variables X and Y positively or negatively correlated? **Answer:**

X and Y are positively correlated. This is because the correlation coefficient ($\rho_{X,Y} = 0.971$) is a positive value and close to 1.

E. Use `np.linalg.eigh` to check that the matrix `C_XY` is indeed positive definite.

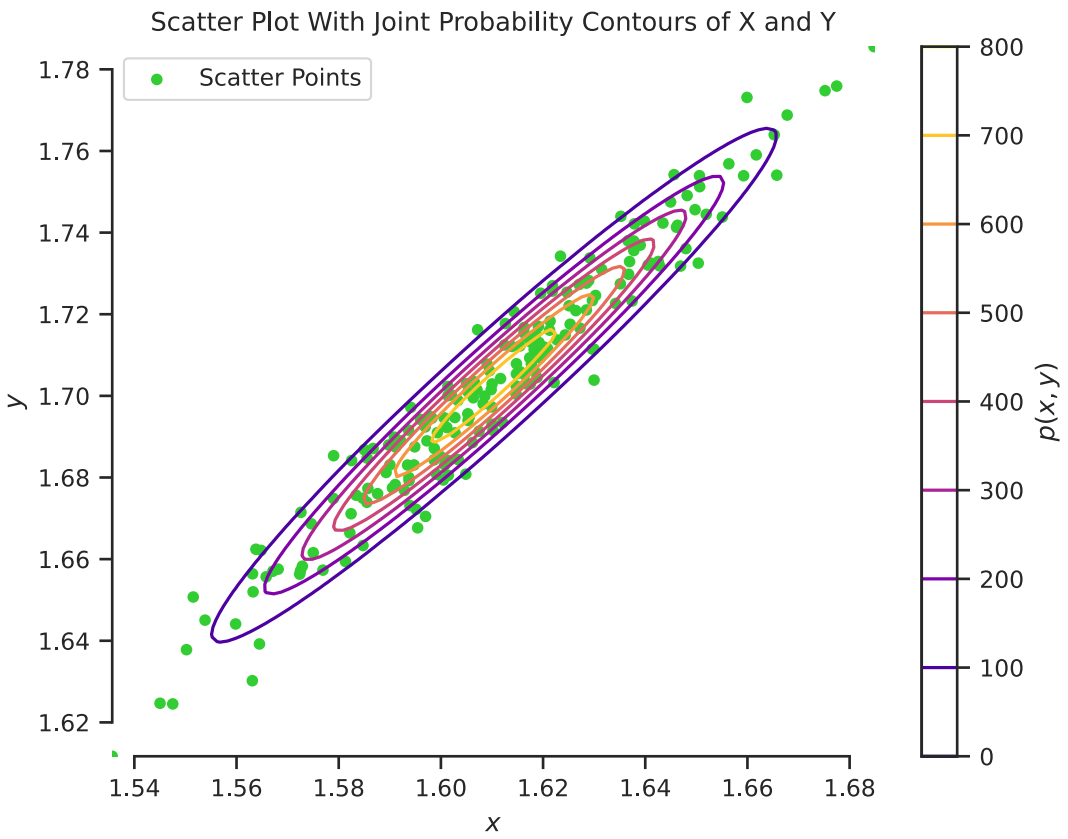
[ ]:
```
# Your code here
print("Eigenvalues of C_XY:")
print(np.linalg.eigh(C_XY)[0])
print("\nSince all eigenvalues are positive, C_XY is positive definite.")
```

```
Eigenvalues of C_XY:
[0.00002474 0.00168827]

Since all eigenvalues are positive, C_XY is positive definite.
```

F. Use the functionality of `scipy.stats.multivariate_normal` to plot the joint probability function of the samples of X and Y in the same plot as the scatter plot of X and Y.

[ ]:
```
# Your code here
# Use scipy.stats to generate the joint probability function
```

```
XY = st.multivariate_normal(mean=mu_XY, cov=C_XY)

# Create a grid of the X and Y points
x_vals = np.linspace(X_data.min(), X_data.max(), 100)
y_vals = np.linspace(Y_data.min(), Y_data.max(), 100)
X_vals, Y_vals = np.meshgrid(x_vals, y_vals)

# Flatten X and Y, then evaluate the PDF at those points
XY_flat = np.hstack([X_vals.flatten()[:, None], Y_vals.flatten()[:, None]])
Z_flat = XY.pdf(XY_flat)

# Reshape Z
Z_vals = Z_flat.reshape(X_vals.shape)

#Plot the scatter plot and contours
fig, ax = plt.subplots()
ax.scatter(X_data, Y_data, s=50, marker=".", c="limegreen", label="Scatter␣
 ↪Points")
c = ax.contour(X_vals, Y_vals, Z_vals, levels=8, cmap="plasma")
ax.set_xlabel(r"$x$")
ax.set_ylabel(r"$y$")
ax.set_title("Scatter Plot With Joint Probability Contours of X and Y")
ax.legend(loc="best")
plt.colorbar(c, label="$p(x,y)$")
sns.despine(trim=True);
```



G. Now, consider each $B-H$ curve a random vector. That is, the random vector $\mathbf{B}$ corresponds to the magnetic flux density values at a fixed number of H-values. It is:

$$\mathbf{B} = (B(H_1), \ldots, B(H_{1500})).$$

It is like $\mathbf{X} = (X, Y)$ only now we have 1,500 dimensions instead of 2.

First, let's find the mean of this random vector:

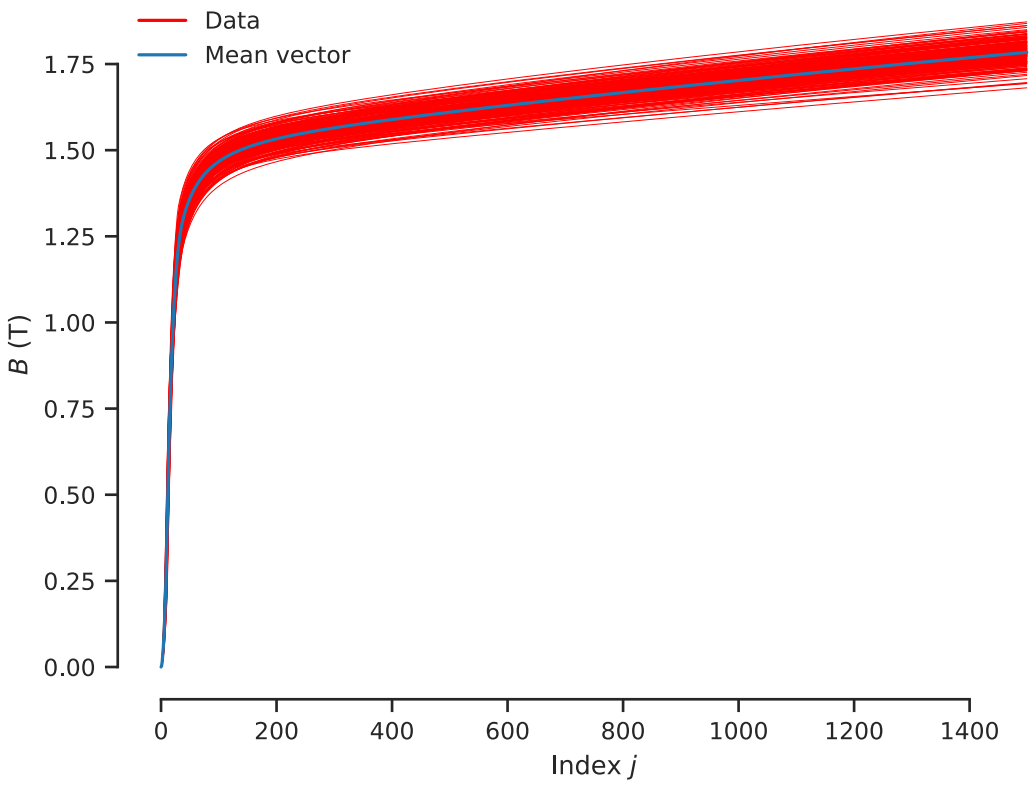
[ ]:
```
B_mu = np.mean(B_data, axis=0)
B_mu
```

[ ]:
```
array([0.        , 0.00385192, 0.01517452, …, 1.78373703, 1.78389267,
       1.78404828])
```

Let's plot the mean on top of all the data we have:

[ ]:
```
fig, ax = plt.subplots()
ax.plot(B_data[:, :].T, 'r', lw=0.1)
plt.plot([],[], 'r', label='Data')
ax.plot(B_mu, label="Mean vector")
```

```
ax.set_xlabel(r"Index $j$")
ax.set_ylabel(r"$B$ (T)")
plt.legend(loc="best", frameon=False)
sns.despine(trim=True);
```



It looks good. Now, find the covariance matrix of $\mathbf{B}$. This is going to be a 1500x1500 matrix.

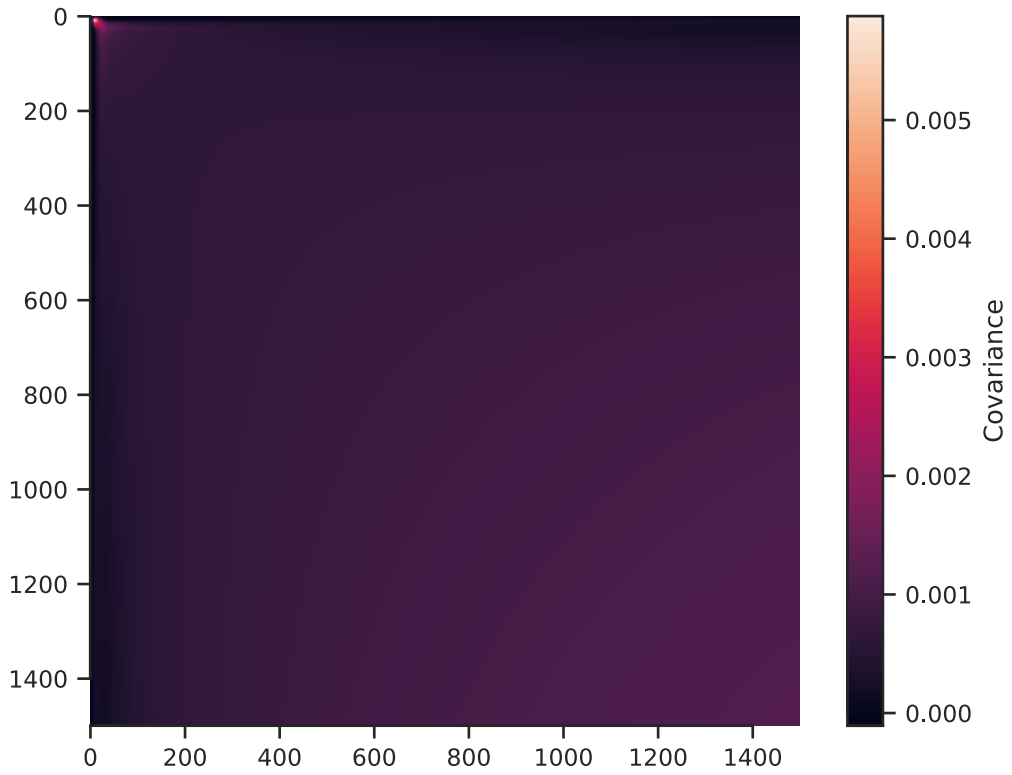
[ ]:
```
np.set_printoptions(suppress=False) # Enable scientific notation
B_cov = np.cov(B_data.T)
B_cov
```

[ ]:
```
array([[0.00000000e+00, 0.00000000e+00, 0.00000000e+00, …,
        0.00000000e+00, 0.00000000e+00, 0.00000000e+00],
       [0.00000000e+00, 1.16277948e-06, 4.41977479e-06, …,
        3.18233676e-06, 3.18391580e-06, 3.18549316e-06],
       [0.00000000e+00, 4.41977479e-06, 1.68041482e-05, …,
        1.22832828e-05, 1.22890907e-05, 1.22948922e-05],
       …,
       [0.00000000e+00, 3.18233676e-06, 1.22832828e-05, …,
        1.20268920e-03, 1.20293022e-03, 1.20317114e-03],
       [0.00000000e+00, 3.18391580e-06, 1.22890907e-05, …,
```

```
 1.20293022e-03, 1.20317134e-03, 1.20341237e-03],
[0.00000000e+00, 3.18549316e-06, 1.22948922e-05, …,
 1.20317114e-03, 1.20341237e-03, 1.20365351e-03]])
```

Let's plot this matrix:

[ ]:
```
fig, ax = plt.subplots()
c = ax.imshow(B_cov, interpolation='nearest')
plt.colorbar(c, label="Covariance")
sns.despine(trim=True);
```



The numbers are very small. This is because the covariance depends on the units of the variables. We need to do the same thing we did with the correlation coefficient: divide by the standard deviations of the variables. Here is how you can get the correlation coefficients:

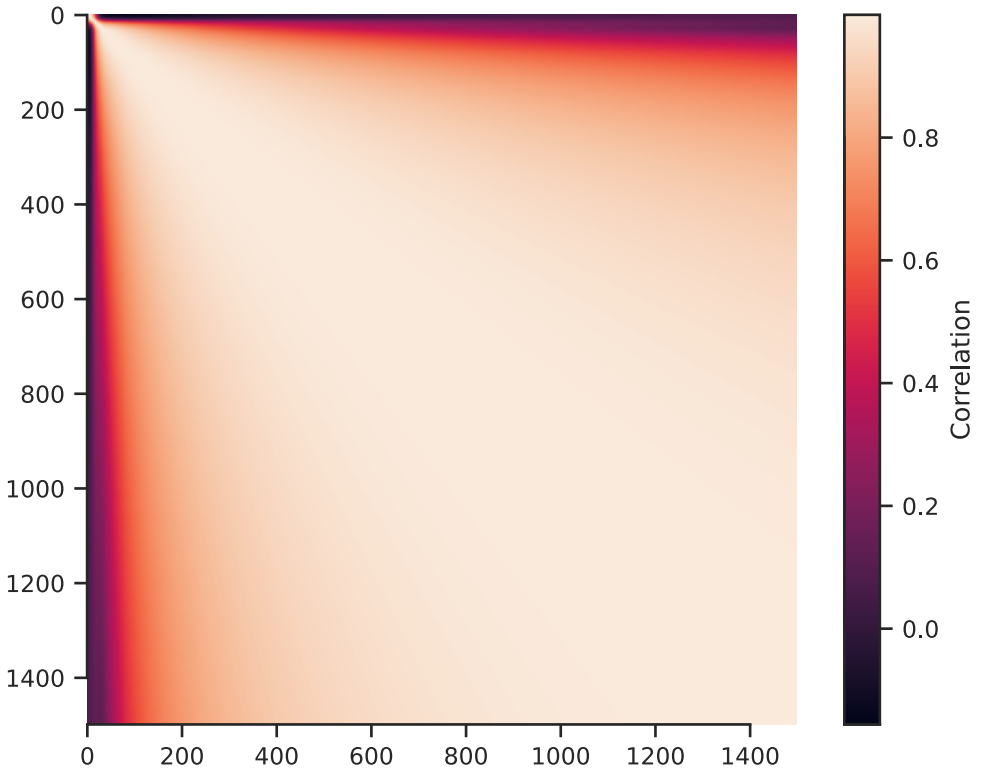
[ ]:
```
# Note that I have to remove the first point because it is always zero
# and it has zero variance.
B_corr = np.corrcoef(B_data[:,1:].T)
B_corr
```

[ ]:
```
array([[1.        , 0.99986924, 0.99941799, …, 0.08509827, 0.08512344,
        0.08514855],
```

```
[0.99986924, 1.        , 0.99983894, …, 0.08640313, 0.08642667,
 0.08645015],
[0.99941799, 0.99983894, 1.        , …, 0.08782484, 0.08784655,
 0.08786822],
…,
[0.08509827, 0.08640313, 0.08782484, …, 1.        , 0.99999998,
 0.9999999 ],
[0.08512344, 0.08642667, 0.08784655, …, 0.99999998, 1.        ,
 0.99999998],
[0.08514855, 0.08645015, 0.08786822, …, 0.9999999 , 0.99999998,
 1.        ]])
```

Here is the correlation visualized:

[ ]:
```
fig, ax = plt.subplots()
c = ax.imshow(B_corr, interpolation='nearest')
plt.colorbar(c, label="Correlation")
sns.despine(trim=True);
```



The values are quite a bit correlated. This makes sense because the curves are all very smooth and look very much alike.

Let's check if the covariance is indeed positive definite:

[ ]:
```
print("Eigenvalues of B_cov:")
print(np.linalg.eigh(B_cov)[0])
```

```
Eigenvalues of B_cov:
[-4.43233557e-16 -2.43956627e-16 -2.29458376e-16 …  4.66244763e-02
  1.16644070e-01  1.20726782e+00]
```

Notice that several eigenvalues are negative, but they are too small. Very close to zero. This happens often in practice when you are finding the covariance of large random vectors. It arises from the fact that we use floating-point arithmetic instead of real numbers. It is a numerical artifact. If you tried to use this covariance to make a multivariate average random vector using `scipy.stats` it would fail. Try this:

[ ]:
```
B = st.multivariate_normal(mean=B_mu, cov=B_cov)
```

```
---------------------------------------------------------------------------
LinAlgError                               Traceback (most recent call last)
/tmp/ipython-input-101-3953023254.py in <cell line: 0>()
----> 1 B = st.multivariate_normal(mean=B_mu, cov=B_cov)

/usr/local/lib/python3.11/dist-packages/scipy/stats/_multivariate.py in␣
 ↪__call__(self, mean, cov, allow_singular, seed)
    399         See `multivariate_normal_frozen` for more information.
    400         """
--> 401         return multivariate_normal_frozen(mean, cov,
    402                                           allow_singular=allow_singular
    403                                           seed=seed)

/usr/local/lib/python3.11/dist-packages/scipy/stats/_multivariate.py in␣
 ↪__init__(self, mean, cov, allow_singular, seed, maxpts, abseps, releps)
    906         self._dist = multivariate_normal_gen(seed)
    907         self.dim, self.mean, self.cov_object = (
--> 908             self._dist._process_parameters(mean, cov, allow_singular))
    909         self.allow_singular = allow_singular or self.cov_object.
 ↪_allow_singular
    910         if not maxpts:

/usr/local/lib/python3.11/dist-packages/scipy/stats/_multivariate.py in␣
 ↪_process_parameters(self, mean, cov, allow_singular)
    423             # array with `_PSD`, and then use wrapper that satisfies th
    424             # `Covariance` interface, `CovViaPSD`.
--> 425             psd = _PSD(cov, allow_singular=allow_singular)
    426             cov_object = _covariance.CovViaPSD(psd)
    427             return dim, mean, cov_object

/usr/local/lib/python3.11/dist-packages/scipy/stats/_multivariate.py in␣
 ↪__init__(self, M, cond, rcond, lower, check_finite, allow_singular)
```

```
    176             msg = ("When `allow_singular is False`, the input matrix␣
 ↪must be "
    177                    "symmetric positive definite.")
--> 178             raise np.linalg.LinAlgError(msg)
    179         s_pinv = _pinv_1d(s, eps)
    180         U = np.multiply(u, np.sqrt(s_pinv))

LinAlgError: When `allow_singular is False`, the input matrix must be symmetric,
 ↪positive definite.
```

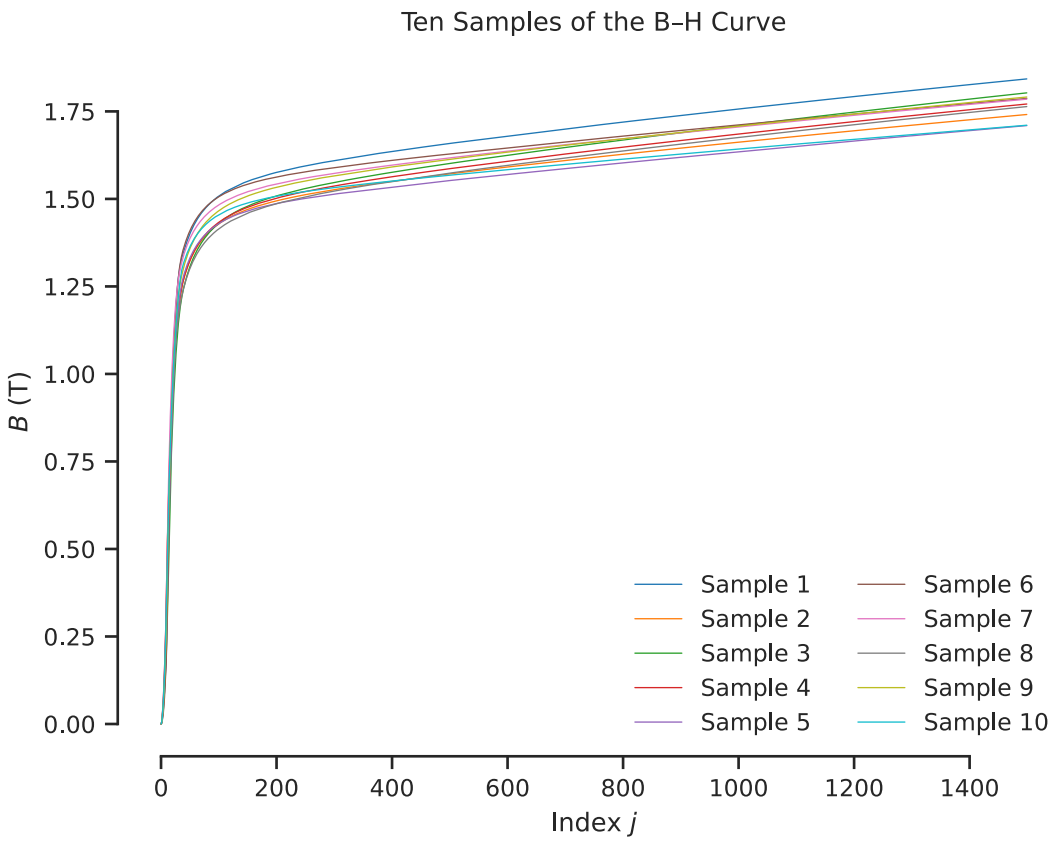
The way to overcome this problem is to add a small positive number to the diagonal. This needs to be very small so that the distribution stays mostly the same. It must be the smallest possible number that makes the covariance matrix behave well. This is known as the *jitter* or the *nugget*. Find the nugget playing with the code below. Every time you try, multiply the nugget by ten.

[ ]:
```
# Pick the nugget here
nugget = 1e-9  # The nugget was increased from 1e-12 to 1e-9
# This is the modified covariance matrix
B_cov_w_nugget = B_cov + nugget * np.eye(B_cov.shape[0])
# Try building the distribution:
try:
    B = st.multivariate_normal(mean=B_mu, cov=B_cov_w_nugget)
    print('It worked! Move on.')
except:
    print('It did not work. Increase nugget by 10.')
```

```
It worked! Move on.
```

H. Now, you have created your first stochastic model of a complicated physical quantity. By sampling from your newly constructed random vector $\mathbf{B}$, you have essentially quantified your uncertainty about the $B - H$ curve as induced by the inability to control steel production perfectly. Take ten samples of this random vector and plot them.

[ ]:
```
# Your code here
B_samples = B.rvs(size=10)
fig, ax = plt.subplots()
for k, curve in enumerate(B_samples, 1):
    ax.plot(curve, lw=0.5, label=f"Sample {k}")
ax.set_xlabel(r"Index $j$")
ax.set_ylabel(r"$B$ (T)")
ax.set_title("Ten Samples of the B-H Curve")
ax.legend(loc="best", ncol=2, frameon=False)
sns.despine(trim=True);
```



Congratulations! You have made your first stochastic model of a physical field quantity. You can now sample $B - H$ curves in a way that honors the manufacturing uncertainties. This is the first step in uncertainty quantification studies. The next step would be to propagate these samples through Maxwell's equations to characterize the effect on the performance of an electric machine. If you want to see how that looks, look at {cite}`sahu2020` and {cite}`beltran2020`.