homework-02_RohanDekate

September 18, 2023

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.

[1]:
```
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
np.random.seed(seed=1)
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:** Rohan
- **Last Name:** Dekate
- **Email:** dekate@purdue.edu
- **Used generative AI to complete this assignment (Yes/No):** No
- **Which generative AI tool did you use (if applicable)?:** NA

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

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

[2]:
```
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:

[3]:
```
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 $\sum_{x,y} p(X = x, Y = y) = 1$.

[4]:
```
# Your code here
print(f"Sum of P_ij = {np.sum(P):.2f}")
```

```
Sum of P_ij = 1.00
```

B. Find the marginal probability density of X:

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

You can represent this as a 5-dimensional vector.

[5]:
```python
p_x = np.sum(P, axis=1) # Axis = 1 tells sum to sum only the second axis
print(f"Marginal Probability Density of X: {p_x}")
```

```
Marginal Probability Density of X: [0.20412376 0.25783426 0.19822111 0.17820324
0.16161762]
```

[6]:
```python
# Verification
print(f"sum of p_x = {np.sum(p_x):.2f}")
```

```
sum of p_x = 1.00
```

[7]:
```python
# Hint, you can do this in one line if you read this:
help(np.sum)
```

```
Help on function sum in module numpy:

sum(a, axis=None, dtype=None, out=None, keepdims=<no value>, initial=<no value>,
where=<no value>)
    Sum of array elements over a given axis.

    Parameters
    ----------
    a : array_like
        Elements to sum.
    axis : None or int or tuple of ints, optional
        Axis or axes along which a sum is performed.  The default,
        axis=None, will sum all of the elements of the input array.  If
        axis is negative it counts from the last to the first axis.

        .. versionadded:: 1.7.0

        If axis is a tuple of ints, a sum is performed on all of the axes
        specified in the tuple instead of a single axis or all the axes as
        before.
    dtype : dtype, optional
        The type of the returned array and of the accumulator in which the
        elements are summed.  The dtype of `a` is used by default unless `a`
        has an integer dtype of less precision than the default platform
        integer.  In that case, if `a` is signed then the platform integer
        is used while if `a` is unsigned then an unsigned integer of the
        same precision as the platform integer is used.
    out : ndarray, optional
        Alternative output array in which to place the result. It must have
        the same shape as the expected output, but the type of the output
        values will be cast if necessary.
    keepdims : bool, optional
        If this is set to True, the axes which are reduced are left
        in the result as dimensions with size one. With this option,
        the result will broadcast correctly against the input array.

        If the default value is passed, then `keepdims` will not be
        passed through to the `sum` method of sub-classes of
        `ndarray`, however any non-default value will be.  If the
        sub-class' method does not implement `keepdims` any
        exceptions will be raised.
    initial : scalar, optional
        Starting value for the sum. See `~numpy.ufunc.reduce` for details.

        .. versionadded:: 1.15.0

    where : array_like of bool, optional
        Elements to include in the sum. See `~numpy.ufunc.reduce` for details.

        .. versionadded:: 1.17.0

    Returns
    -------
    sum_along_axis : ndarray
        An array with the same shape as `a`, with the specified
        axis removed.   If `a` is a 0-d array, or if `axis` is None, a scalar
        is returned.  If an output array is specified, a reference to
        `out` is returned.

    See Also
    --------
    ndarray.sum : Equivalent method.

    add.reduce : Equivalent functionality of `add`.

    cumsum : Cumulative sum of array elements.

    trapz : Integration of array values using the composite trapezoidal rule.

    mean, average

    Notes
    -----
    Arithmetic is modular when using integer types, and no error is
```

```
raised on overflow.

The sum of an empty array is the neutral element 0:

>>> np.sum([])
0.0

For floating point numbers the numerical precision of sum (and
``np.add.reduce``) is in general limited by directly adding each number
individually to the result causing rounding errors in every step.
However, often numpy will use a  numerically better approach (partial
pairwise summation) leading to improved precision in many use-cases.
This improved precision is always provided when no ``axis`` is given.
When ``axis`` is given, it will depend on which axis is summed.
Technically, to provide the best speed possible, the improved precision
is only used when the summation is along the fast axis in memory.
Note that the exact precision may vary depending on other parameters.
In contrast to NumPy, Python's ``math.fsum`` function uses a slower but
more precise approach to summation.
Especially when summing a large number of lower precision floating point
numbers, such as ``float32``, numerical errors can become significant.
In such cases it can be advisable to use `dtype="float64"` to use a higher
precision for the output.

Examples
--------
>>> np.sum([0.5, 1.5])
2.0
>>> np.sum([0.5, 0.7, 0.2, 1.5], dtype=np.int32)
1
>>> np.sum([[0, 1], [0, 5]])
6
>>> np.sum([[0, 1], [0, 5]], axis=0)
array([0, 6])
>>> np.sum([[0, 1], [0, 5]], axis=1)
array([1, 5])
>>> np.sum([[0, 1], [np.nan, 5]], where=[False, True], axis=1)
array([1., 5.])

If the accumulator is too small, overflow occurs:

>>> np.ones(128, dtype=np.int8).sum(dtype=np.int8)
-128

You can also start the sum with a value other than zero:

>>> np.sum([10], initial=5)
15
```

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

[8]:
```
# Your code here
p_y = np.sum(P, axis=0)
print(f"Marginal Probability Density of Y: {p_y}")
```

```
Marginal Probability Density of Y: [0.14914347 0.14973252 0.09144527 0.09563304
0.09230073 0.15128076
 0.05914682 0.09709805 0.11421933]
```

[9]:
```
# Verification
print(f"sum of p_y = {np.sum(p_y):.2f}")
```

```
sum of p_y = 1.00
```

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

[10]:
```
# Your code here
E_X = np.sum(np.arange(5) * p_x)
print(f"Expectation of X: E[X] = {E_X:.2f}")

E_X2 = np.sum(np.arange(5) ** 2 * p_x)
V_X = E_X2 - E_X ** 2
print(f"Variance of X: V[X] = {V_X:.2f}")

E_Y = np.sum(np.arange(9) * p_y)
print(f"Expectation of Y: E[Y] = {E_Y:.2f}")
E_Y2 = np.sum(np.arange(9) ** 2 * p_y)
V_Y = E_Y2 - E_Y ** 2
print(f"Variance of Y: V[Y] = {V_Y:.2f}")
```

```
Expectation of X: E[X] = 1.84
Variance of X: V[X] = 1.87
Expectation of Y: E[Y] = 3.69
Variance of Y: V[Y] = 7.19
```

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

$\mathbb{E}[X+Y] = \mathbb{E}[X] + \mathbb{E}[Y]$

[11]:
```
# Your code here
print(f"Expectation of E[X+Y] = {E_X+E_Y:.2f}")
```

```
Expectation of 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?

```
[12]: # Your code here
      C_XY = 0.0 # Keeping track of the sum
      for x in range(len(p_x)):
          for y in range(len(p_y)):
              C_XY += (x - E_X) * (y - E_Y) * P[x, y] # the += means add to the left␣
       ↪hand side
      print(f"C[X, Y] = {C_XY:.2f}")
```

```
C[X, Y] = 0.32
```

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

```
rho_XY = 0.09
```

- The product ($\mathbb{C}[X, Y]$) is positive, hence the two random variables move in the same direction.
- The two random variables are slightly positively correlated (Correlation coefficient is close to 0).

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

$\mathbb{V}[X + Y] = \mathbb{V}[X] + \mathbb{V}[Y] + 2\mathbb{C}[X, Y]$

```
[14]: # Your code here
      V_X_Y = V_X+V_Y+2*C_XY
      print(f"The Variance of X+Y = {V_X_Y:.2f}")
```

```
The Variance of 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.

```
[15]: # Your code here
      m, n = P.shape
      sum = 0
      for x in range(m):
        for y in range(n):
          if x+y<=5:
          # print(x,y)
          # print(P[x,y])
            sum = sum+P[x,y]
      print(f"p(X+Y<=5) = {sum}")
```

```
p(X+Y<=5) = 0.5345903099999999
```

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 $Z$.

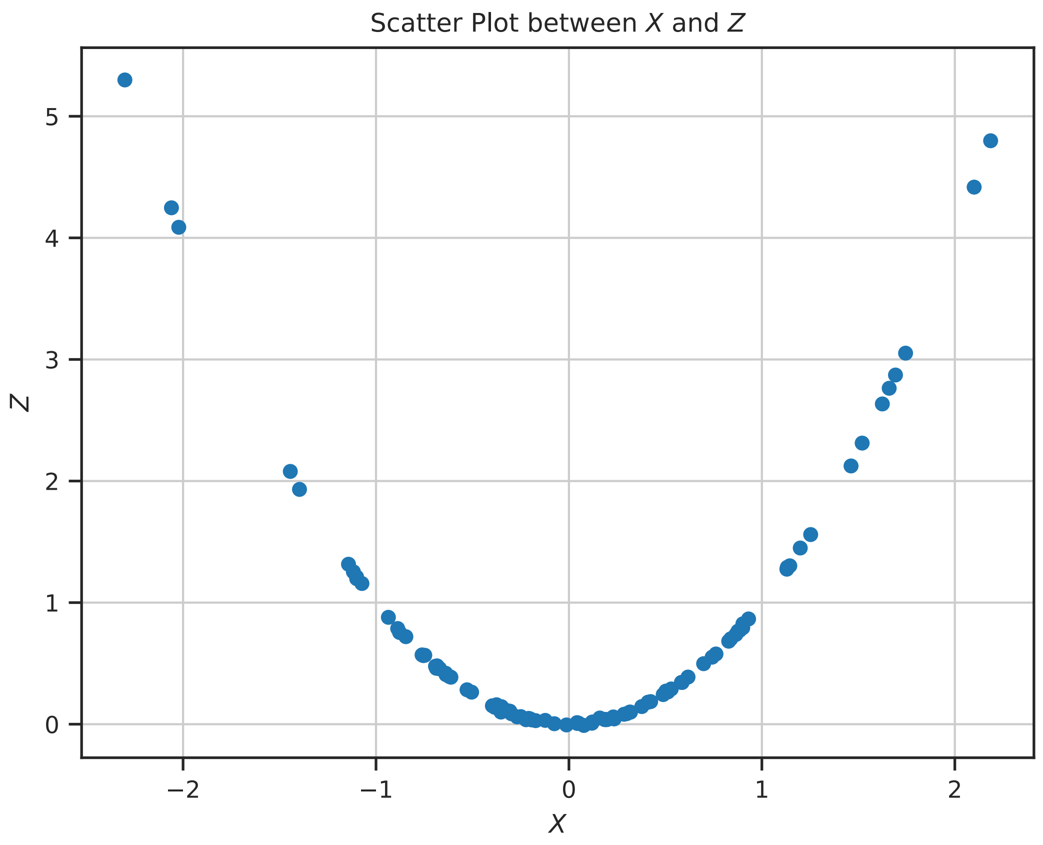
[16]:
```
# Your code here
mu = 0
sigma = 1
X = np.random.normal(loc=mu, scale=sigma, size=100)
Y = np.random.normal(loc=mu, scale=sigma, size=100)
```

[17]:
```
Z = X**2 + 0.01*Y
print(Z.shape)
```

```
(100,)
```

B. Do the scatter plot between $X$ and $Z$.

[18]:
```
# Your code here
plt.scatter(X,Z)
plt.grid()
plt.xlabel("$X$")
plt.ylabel("$Z$")
plt.title("Scatter Plot between $X$ and $Z$")
plt.show()
```



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

Answer: The shape of the curve in the plot is not linear. It is a U shape. This indicates there is no relation between the quantity on the x-axis and the one on the y-axis. Hence X and Z are independent.

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

```
[19]: # Your code here
      var_z = np.var(Z)
      print(f"Variance of Z = {var_z}")
```

```
Variance of Z = 1.2384745994018496
```

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

```
[20]: # Your code here
      cov_xz = np.cov(X,Z)[0][1]
      print(f"Covariance between X and Z = {cov_xz}")
```

```
Covariance between X and Z = 0.09350814777926905
```

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

[21]:
```python
# Your code here
corr_xz = cov_xz/np.sqrt(np.var(X)*np.var(Z))
print(f"Correlation between X and Z = {corr_xz}")
```

```
Correlation between X and Z = 0.09492617469123679
```

[22]:
```python
st.pearsonr(X, Z)[0]
```

[22]:
```
0.09397691294432443
```

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.

[23]:
```python
# Your code here
X1 = np.random.normal(loc=mu, scale=sigma, size=1000)
Y1 = np.random.normal(loc=mu, scale=sigma, size=1000)

Z1 = X1**2 + 0.01*Y1

cov_xz = np.cov(X1, Z1)[0][1]
corr_xz = st.pearsonr(X1, Z1)[0]
print(f"Correlation between X and Z = {corr_xz}")
```

```
Correlation between X and Z = -0.012667222214150505
```

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 $n = 1, 2, \dots, 100,000$. 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?

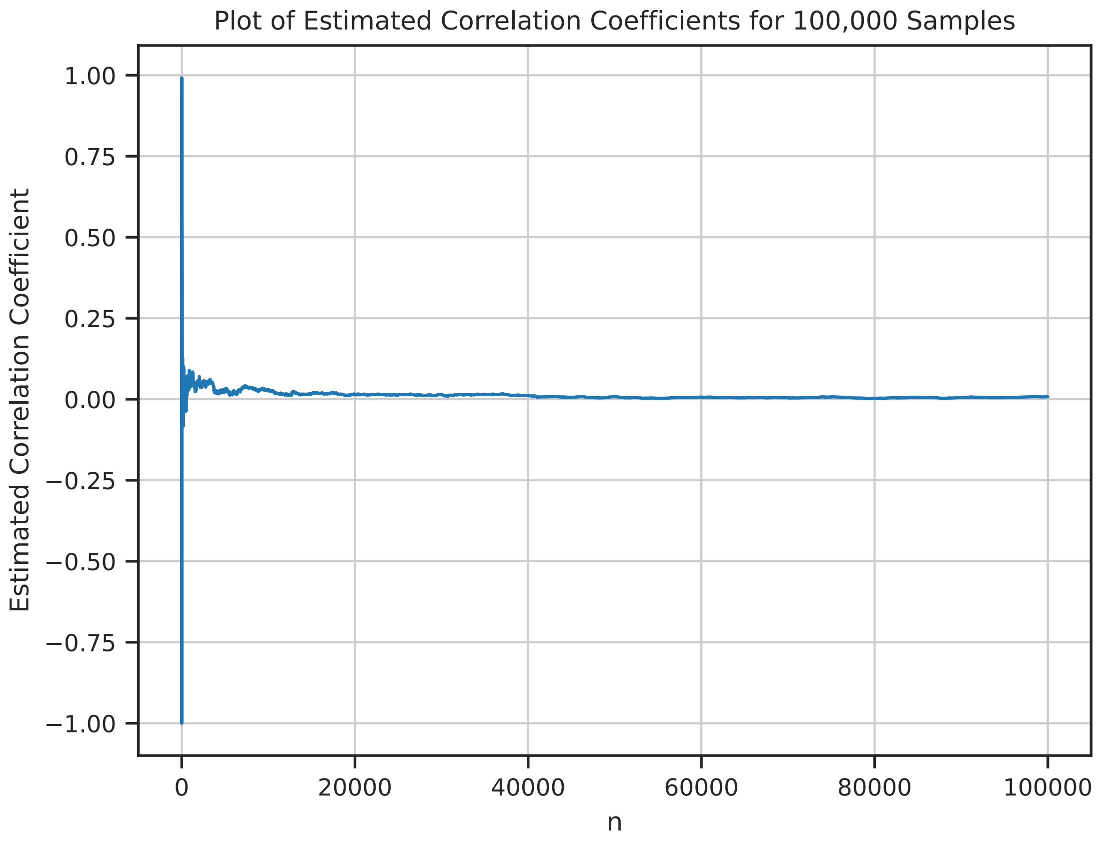
[24]:
```python
X2 = np.random.normal(loc=mu, scale=sigma, size=100000)
Y2 = np.random.normal(loc=mu, scale=sigma, size=100000)
Z2 = X2**2 + 0.01*Y2

corr = []
for n in range(2,100001):
  corr_xz = st.pearsonr(X2[:n], Z2[:n])[0]
  corr.append(corr_xz)
  if n%10000 == 0:
    print(f"For {n} samples, the Correlation between X and Z = {corr_xz}")
```

```
For 10000 samples, the Correlation between X and Z = 0.030081731552177703
For 20000 samples, the Correlation between X and Z = 0.0157922625544284
For 30000 samples, the Correlation between X and Z = 0.014230196131132512
For 40000 samples, the Correlation between X and Z = 0.009679263613508054
For 50000 samples, the Correlation between X and Z = 0.007772778580696004
For 60000 samples, the Correlation between X and Z = 0.006402013138222708
For 70000 samples, the Correlation between X and Z = 0.0050034827396739315
For 80000 samples, the Correlation between X and Z = 0.0029103886025358652
For 90000 samples, the Correlation between X and Z = 0.005825057791307222
For 100000 samples, the Correlation between X and Z = 0.00792607524801912
```

[25]:
```python
# Insert 0 at the beginning of corr list when the sample size is only 1
corr.insert(0,0)
```

[26]:
```python
n = range(len(corr))
plt.plot(n,corr)
plt.grid()
plt.title("Plot of Estimated Correlation Coefficients for 100,000 Samples")
plt.xlabel("n")
plt.ylabel("Estimated Correlation Coefficient")
```

[26]:
```
Text(0, 0.5, 'Estimated Correlation Coefficient')
```



- As the number of samples increases the correlation between X & Z approaches 0.
- Initially the correlation coefficent is unstable when we have less than 20000 samples.
- Around 40000 samples the correlation coefficient stabilizes to 0 and does not change much thereafter.

- Hence, around 40000 samples are needed for a good estimate of the correlation.

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:

```
[27]: 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:

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

```
[28]: 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:

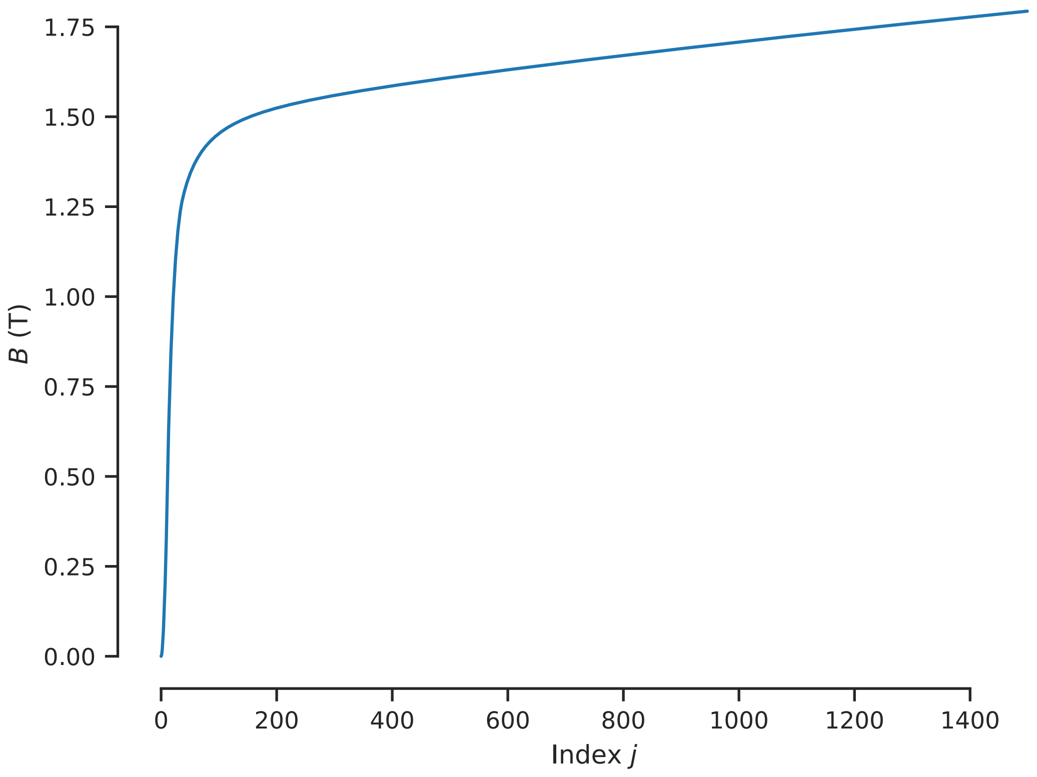
```
[29]: B_data.shape
```

```
[29]: (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.

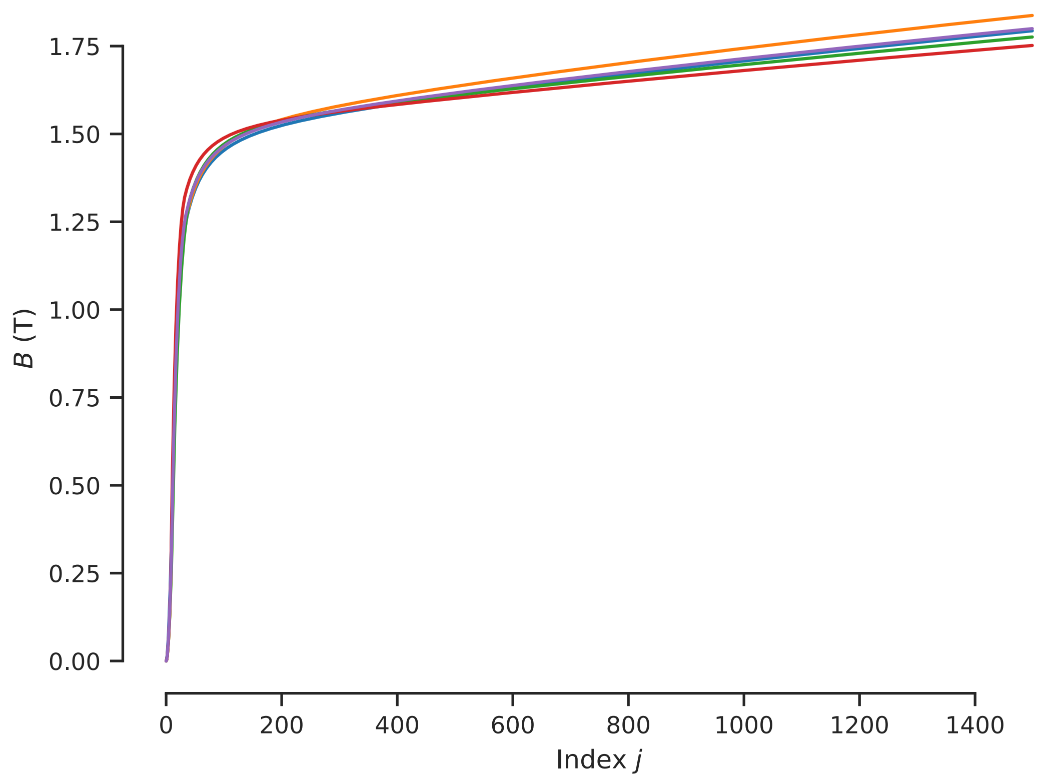
Here is one sample:

[30]:
```
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);
```



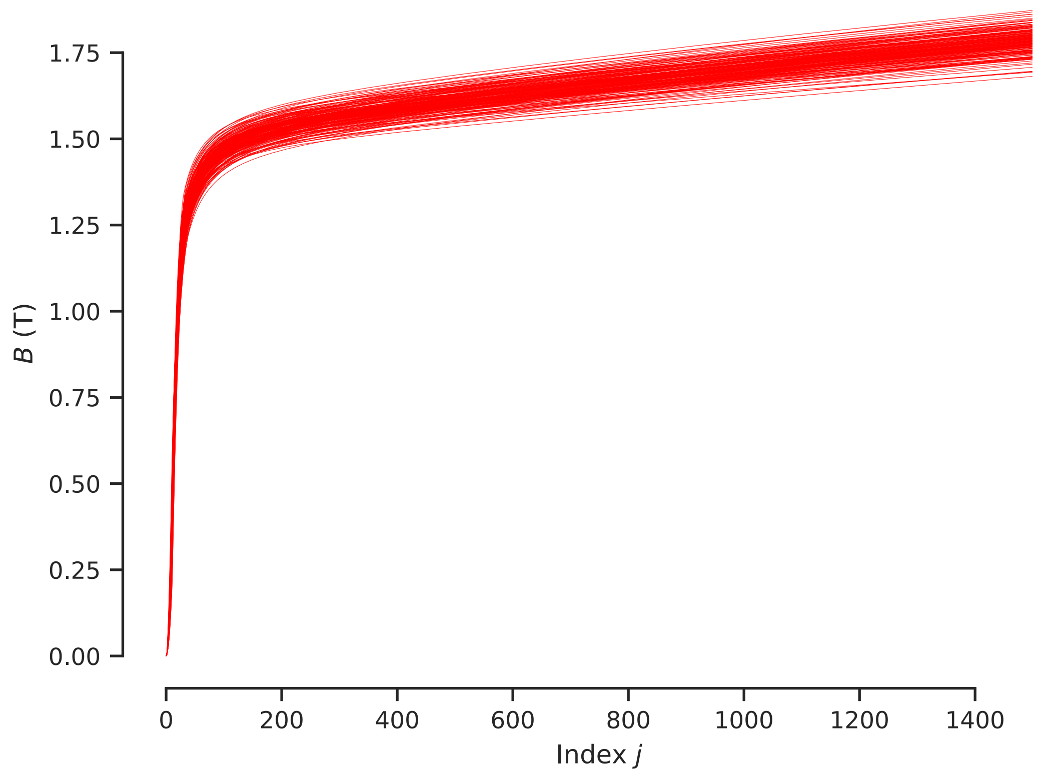
Here are five samples:

[31]:
```
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:

```
[32]: 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);
```

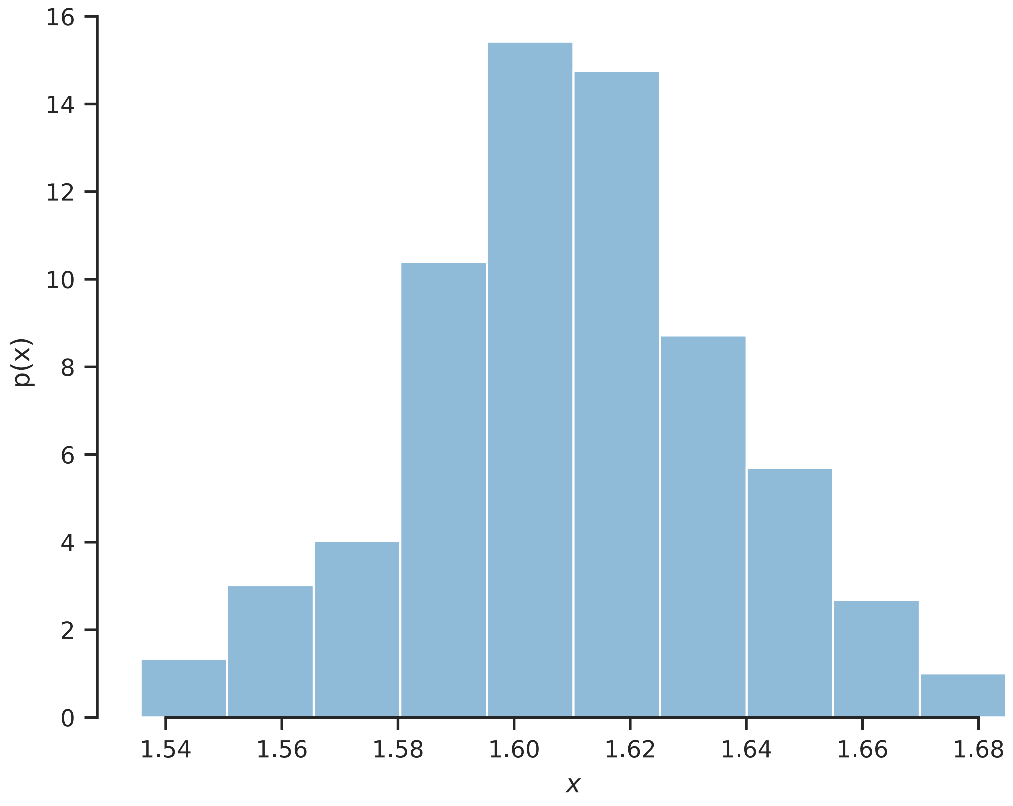


A. We are going to start by studying the data at only one index. Say index $j = 500$. Let's define a random variable

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

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

[33]:
```
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 $N(\mu_{500}, \sigma_{500}^2)$. 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:

[34]:
```
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:

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

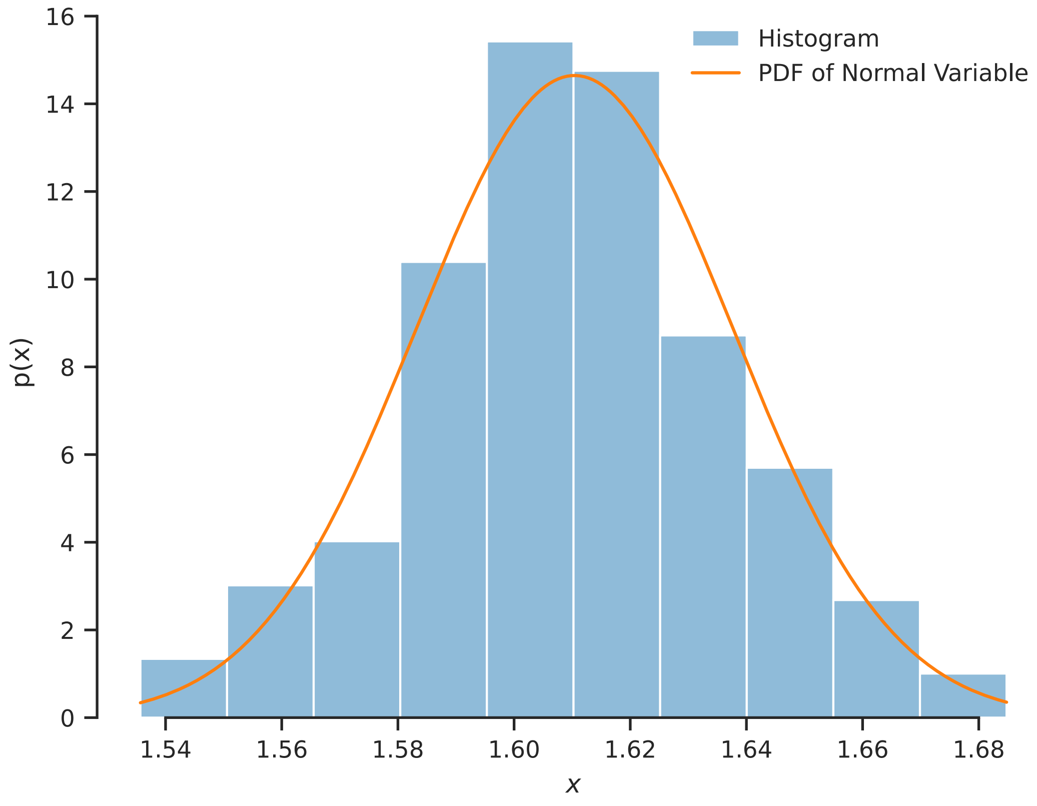
This also converges to the true variance as $N \to \infty$. Here it is:

[35]:
```
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`.

[36]:
```
# Your code here
X_NORM_500 = st.norm(loc=mu_500, scale=np.std(X_data))

X_data = B_data[:, 500]
fig, ax = plt.subplots()
ax.hist(X_data, alpha=0.5, density=True, label="Histogram")

x1s = np.linspace(
    X_data.min(),
    X_data.max(),
    200
)

ax.plot(
    x1s,
    X_NORM_500.pdf(x1s),
    label="PDF of Normal Variable"
)

plt.legend(loc="best", frameon=False)
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.

[37]:
```
# Your code here
print(f"P(X>1.66) = {1 - X_NORM_500.cdf(1.66)}")
```

```
P(X>1.66) = 0.034355748631309635
```

C. Let us now consider another random variable

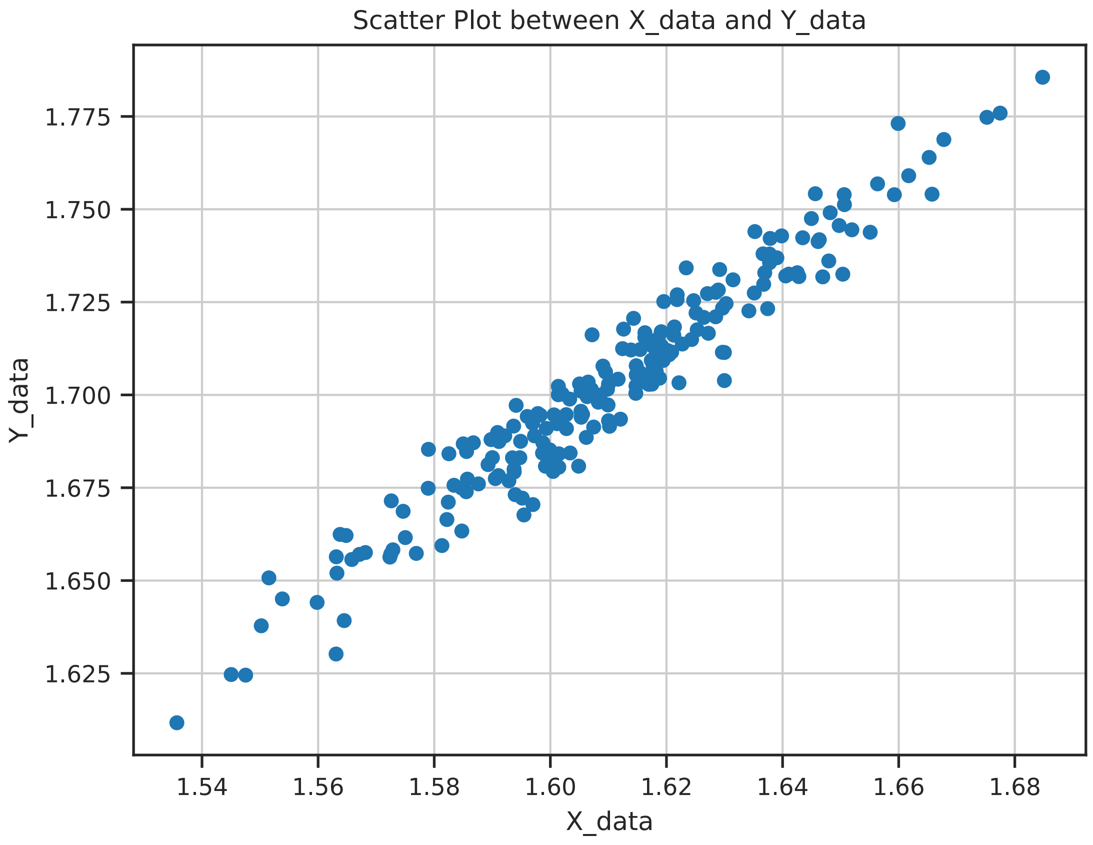
$$Y = B(H_{1000}).$$

Isolate the data for this as well:

[38]:
```
Y_data = B_data[:, 1000]
```

Do the `scatter` plot of X and Y:

[39]:
```
# Your code here
plt.scatter(X_data,Y_data)
plt.grid()
plt.xlabel("X_data")
plt.ylabel("Y_data")
plt.title("Scatter Plot between X_data and Y_data")
plt.show()
```



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).

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

```
[40]: (200, 2)
```

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:

```
[41]: 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:

```
[42]: # 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.

```
[43]: # Your code here
      def correlation_from_covariance(covariance):
          v = np.sqrt(np.diag(covariance))
          outer_v = np.outer(v, v)
          correlation = covariance / outer_v
          correlation[covariance == 0] = 0
          return correlation

      corr_xy = correlation_from_covariance(C_XY)
      print(f"The correlation coefficient between X & Y is : {corr_xy[0][1]}")
```

```
The correlation coefficient between X & Y is : 0.9706148920502284
```

Are the two variables X and Y positively or negatively correlated? **Answer:** X & Y are positively correlated as the correlation coefficient is very close to 1.

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

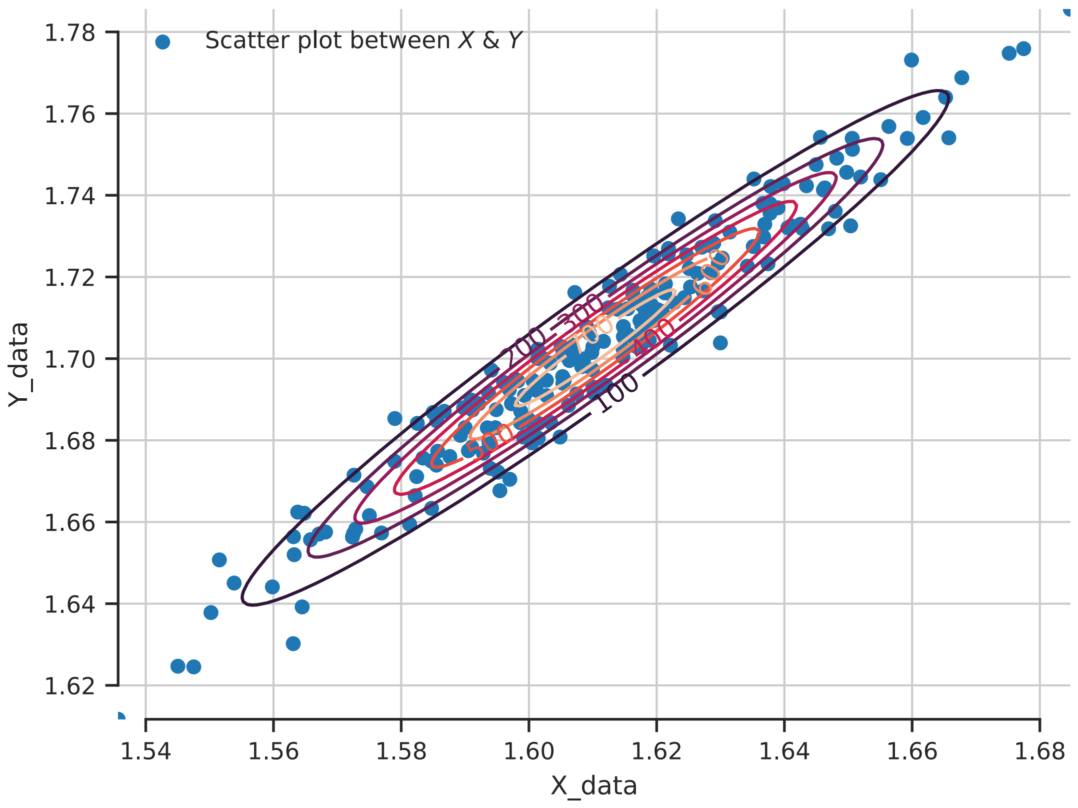
```
[44]: # Your code here
eigenvalues, eigenvectors = np.linalg.eigh(C_XY)
print(eigenvalues)
```

```
[2.47411589e-05 1.68827115e-03]
```

Since the eigenvalues are positive and > 0, the covariance matrix 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 $Y$.

```
[45]: # Your code here
# The multivariate normal random vector
XY = st.multivariate_normal(mean=mu_XY, cov=C_XY)
# CONTOURS
fig, ax = plt.subplots(dpi=150)
x1 = np.linspace(min(X_data), max(X_data), len(X_data))
x2 = np.linspace(min(Y_data), max(Y_data), len(Y_data))
X1, X2 = np.meshgrid(x1, x2)
X_flat = np.hstack(
    [
        X1.flatten()[:, None],
        X2.flatten()[:, None]
    ]
)
# PDF values
pdf_X = XY.pdf(X_flat).reshape(X1.shape)
c = ax.contour(X1, X2, pdf_X)
ax.clabel(c, inline=1, fontsize=10)
plt.scatter(X_data,Y_data, label="Scatter plot between $X$ & $Y$")
plt.legend(loc='best', frameon=False)
plt.xlabel("X_data")
plt.ylabel("Y_data")
plt.grid()
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:

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

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

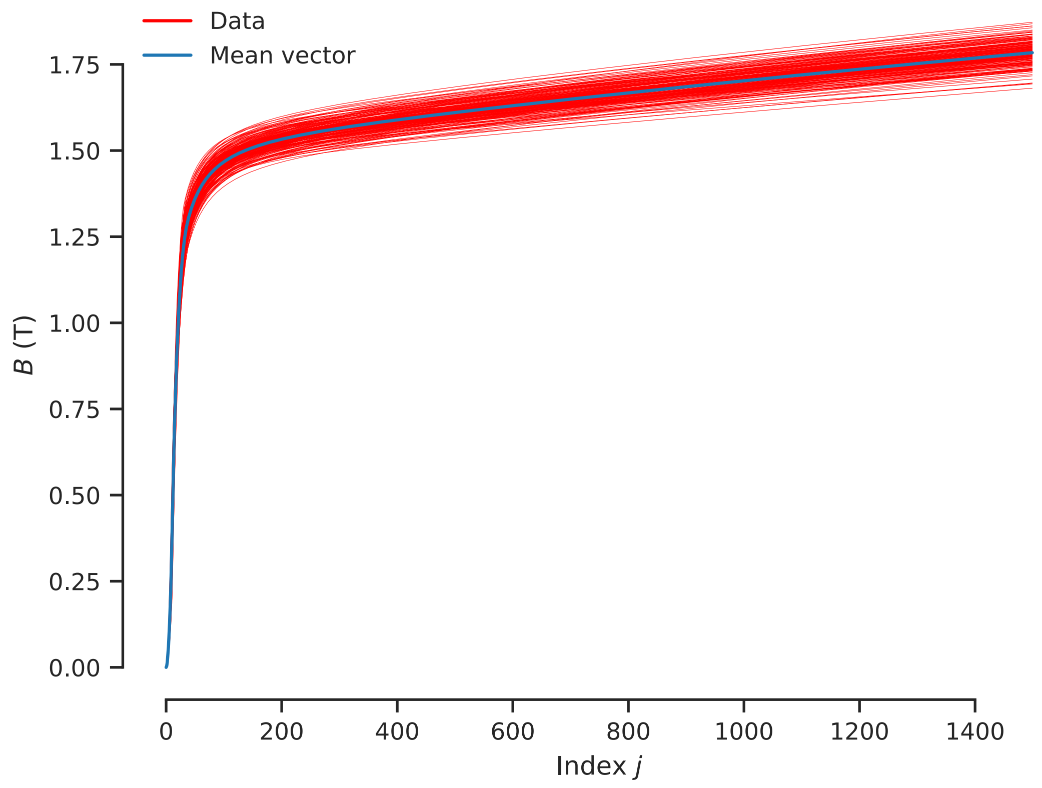
First, let's find the mean of this random vector:

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

```
[46]: 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:

```
[47]: 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);
```



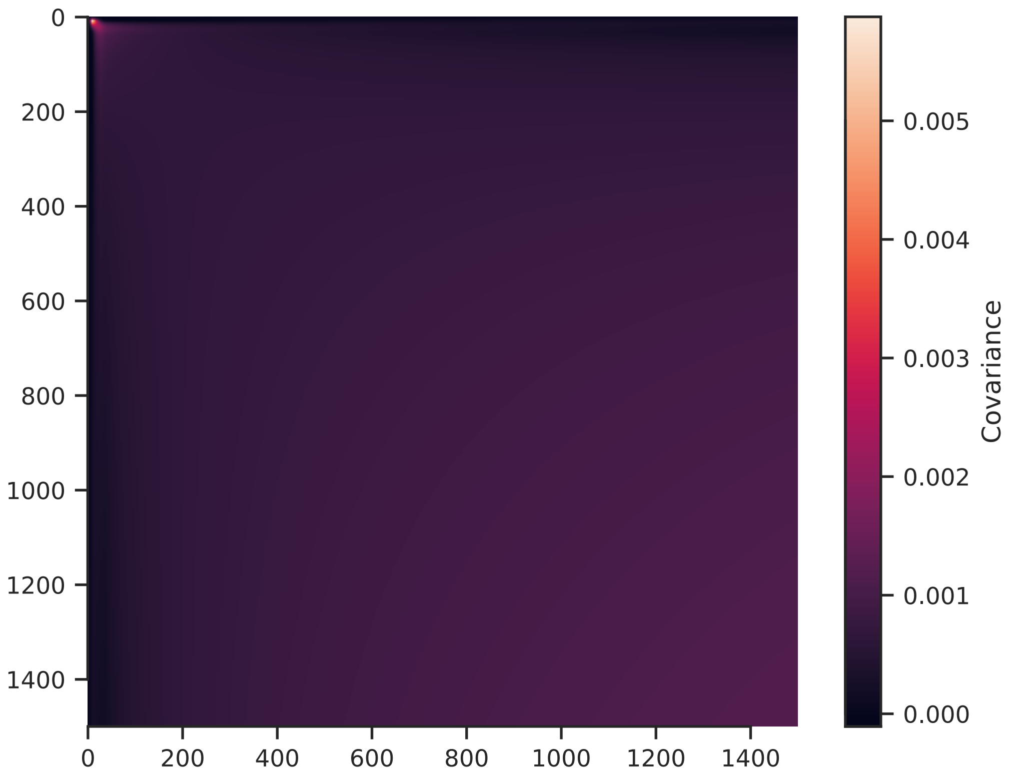
It looks good. Now, find the covariance matrix of **B**. This is going to be a 1500x1500 matrix.

```
[48]: B_cov = np.cov(B_data.T)
      B_cov
```

```
[48]: 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:

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



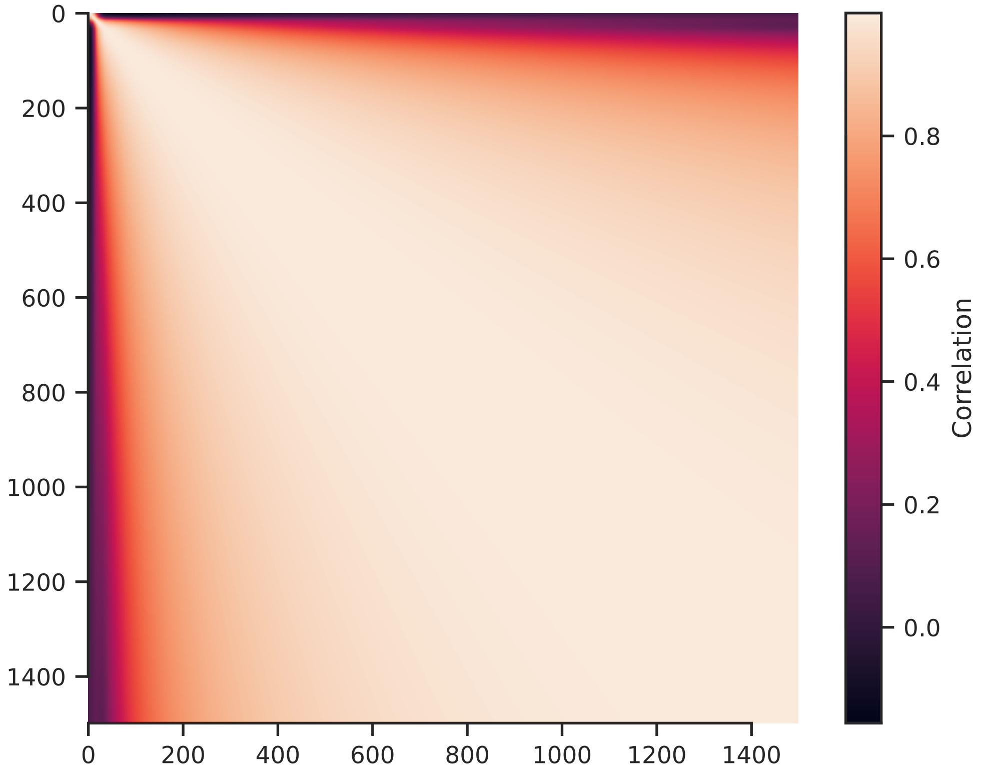
The numbers are very small. This is because the covariance depends on the units of the variables. We need to do the same thing we did with the correlation coefficient: divide by the standard deviations of the variables. Here is how you can get the correlation coefficients:

```
[50]: # 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
```

```
[50]: 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:

```
[51]: 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:

[52]:
```
print("Eigenvalues of B_cov:")
print(np.linalg.eigh(B_cov)[0])
```

```
Eigenvalues of B_cov:
[-3.85054191e-16 -3.39495371e-16 -2.71195607e-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:

[53]:
```
B = st.multivariate_normal(mean=B_mu, cov=B_cov)
```

```
---------------------------------------------------------------------------
LinAlgError                               Traceback (most recent call last)
<ipython-input-53-42ddfd48bf34> in <cell line: 1>()
----> 1 B = st.multivariate_normal(mean=B_mu, cov=B_cov)

/usr/local/lib/python3.10/dist-packages/scipy/stats/_multivariate.py in␣
 ↪__call__(self, mean, cov, allow_singular, seed)
    396         See `multivariate_normal_frozen` for more information.
    397         """
--> 398         return multivariate_normal_frozen(mean, cov,

    399                                           allow_singular=allow_singular
    400                                           seed=seed)

/usr/local/lib/python3.10/dist-packages/scipy/stats/_multivariate.py in␣
 ↪__init__(self, mean, cov, allow_singular, seed, maxpts, abseps, releps)
    837         self._dist = multivariate_normal_gen(seed)
    838         self.dim, self.mean, self.cov_object = (
--> 839             self._dist._process_parameters(mean, cov, allow_singular))

    840         self.allow_singular = allow_singular or self.cov_object.
 ↪_allow_singular
    841         if not maxpts:

/usr/local/lib/python3.10/dist-packages/scipy/stats/_multivariate.py in␣
 ↪_process_parameters(self, mean, cov, allow_singular)
    420             # array with `_PSD`, and then use wrapper that satisfies th
    421             # `Covariance` interface, `CovViaPSD`.
--> 422             psd = _PSD(cov, allow_singular=allow_singular)
    423             cov_object = _covariance.CovViaPSD(psd)
    424             return dim, mean, cov_object

/usr/local/lib/python3.10/dist-packages/scipy/stats/_multivariate.py in␣
 ↪__init__(self, M, cond, rcond, lower, check_finite, allow_singular)
```

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

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

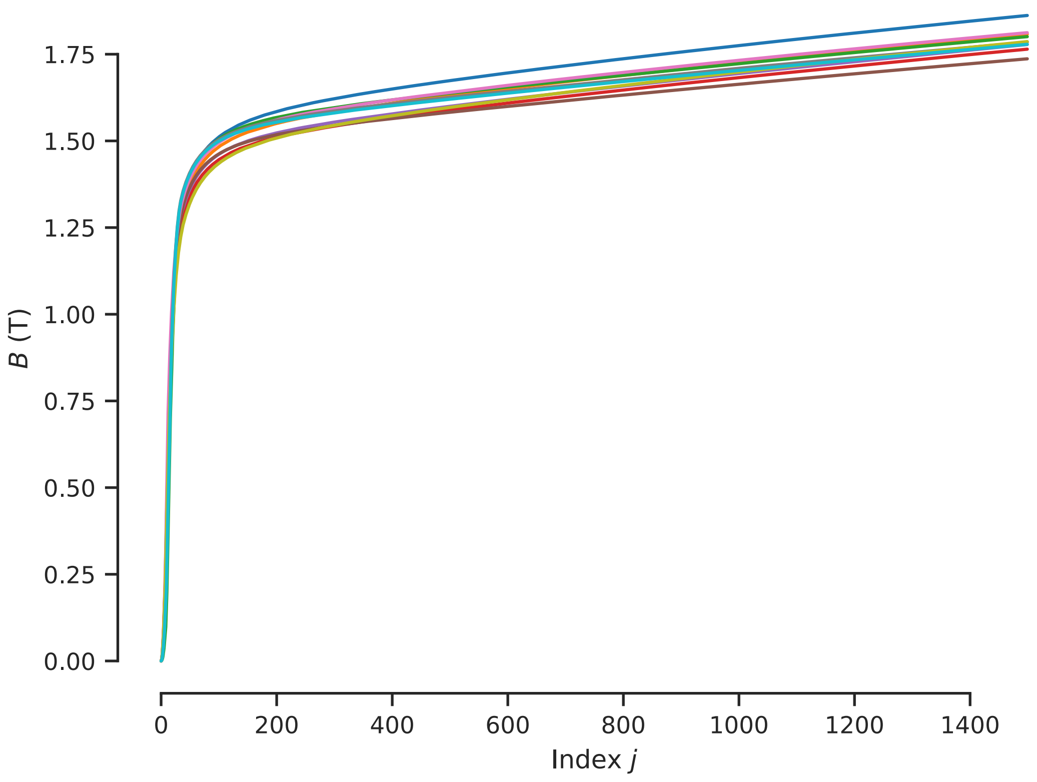
The way to overcome this problem is to add a small positive number to the diagonal. This needs to be very small so that the distribution stays mostly the same. It must be the smallest possible number that makes the covariance matrix behave well. This is known as the *jitter* or the *nugget*. Find the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

[54]:
```python
# Pick the nugget here
nugget = 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 **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.

[55]:
```python
# Your code here
B = st.multivariate_normal(mean=B_mu, cov=B_cov_w_nugget)
# SAMPLES
num_samples = 10
x_samples = B.rvs(size=num_samples)
x_samples_mu = np.mean(x_samples, axis=0)
fig, ax = plt.subplots()
ax.plot(x_samples[:, :].T)
# plt.plot([],[], 'r', label='Data')
# ax.plot(x_samples_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);
```



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`.

```
[ ]:
```