Homework 2

References

- Lectures 4-8 (inclusive).

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.

In [2]:
```
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")

from typing import Callable
import numpy as np
from numpy.typing import NDArray
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)
```

Student details

- **First Name:** Matthew
- **Last Name:** Hansen
- **Email:** hanse217@purdue.edu
- **Used generative AI to complete this assignment (Yes/No):** No
- **Which generative AI tool did you use (if applicable)?:**

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:

In [3]:
```
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:

In [4]:
```
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 $\sum_{x,y} p(X = x, Y = y) = 1$.

In [5]:
```
print(f"Sum of all probabilities: {P.sum():.3f}")
```

```
Sum of all probabilities: 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.

In [6]:
```
p_x = P.sum(axis=1)
p_x
```

Out[6]:
```
array([0.20412376, 0.25783426, 0.19822111, 0.17820324, 0.16161762])
```

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

In [7]:
```
p_y = P.sum(axis=0)
p_y
```

Out[7]:
```
array([0.14914347, 0.14973252, 0.09144527, 0.09563304, 0.09230073,
       0.15128076, 0.05914682, 0.09709805, 0.11421933])
```

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

In [8]:
```python
def expectation(p: np.ndarray, x: np.ndarray) -> float:
    return (p * x).sum()
```

In [9]:
```python
x_vals = np.arange(5)
y_vals = np.arange(9)

E_X = (x_vals * p_x).sum()
print(f"E[X] = {E_X:.3f}")
E_Y = (y_vals * p_y).sum()
print(f"E[Y] = {E_Y:.3f}")
E_X2 = (x_vals**2 * p_x).sum()
E_Y2 = (y_vals**2 * p_y).sum()
V_X = E_X2 - E_X**2
print(f"V[X] = {V_X:.3f}")
V_Y = E_Y2 - E_Y**2
print(f"V[Y] = {V_Y:.3f}")
```

```
E[X] = 1.835
E[Y] = 3.693
V[X] = 1.872
V[Y] = 7.191
```

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

In [10]:
```python
type Function = Callable[[float, float], float]


def expectation(fn: Function, P: NDArray) -> float:
    return sum(fn(i, j) * P[i, j] for i in x_vals for j in y_vals)
```

In [11]:
```python
fn = lambda x, y: x + y
print(f"E[X + Y] = {expectation(fn, P):.3f}")
```

```
E[X + Y] = 5.529
```

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

In [12]:
```python
C_XY = sum((x - E_X) * (y - E_Y) * P[x, y] for x in x_vals for y in y_vals)

print(f"C[X, Y] = {C_XY:.2f}")
if C_XY > 0:
    print("X and Y are positively correlated")
elif C_XY < 0:
    print("X and Y are negatively correlated")
else:
    print("X and Y are not correlated")
```

```
C[X, Y] = 0.32
X and Y are positively correlated
```

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

In [13]:
```
def variance(fn: Function, P: NDArray) -> float:
    square = lambda f: lambda *args: f(*args) ** 2
    return expectation(square(fn), P) - expectation(fn, P) ** 2
```

In [14]:
```
V_XY = variance(fn, P)
print(f"V[X + Y] = {V_XY:.3f}")
```

```
V[X + Y] = 9.700
```

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.

In [15]:
```
P_XY_leq_5 = sum(P[x, y] for x in x_vals for y in y_vals if (x + y <= 5))
print(f"P[X + Y <= 5] = {P_XY_leq_5:.3f}")
```

```
P[X + Y <= 5] = 0.535
```

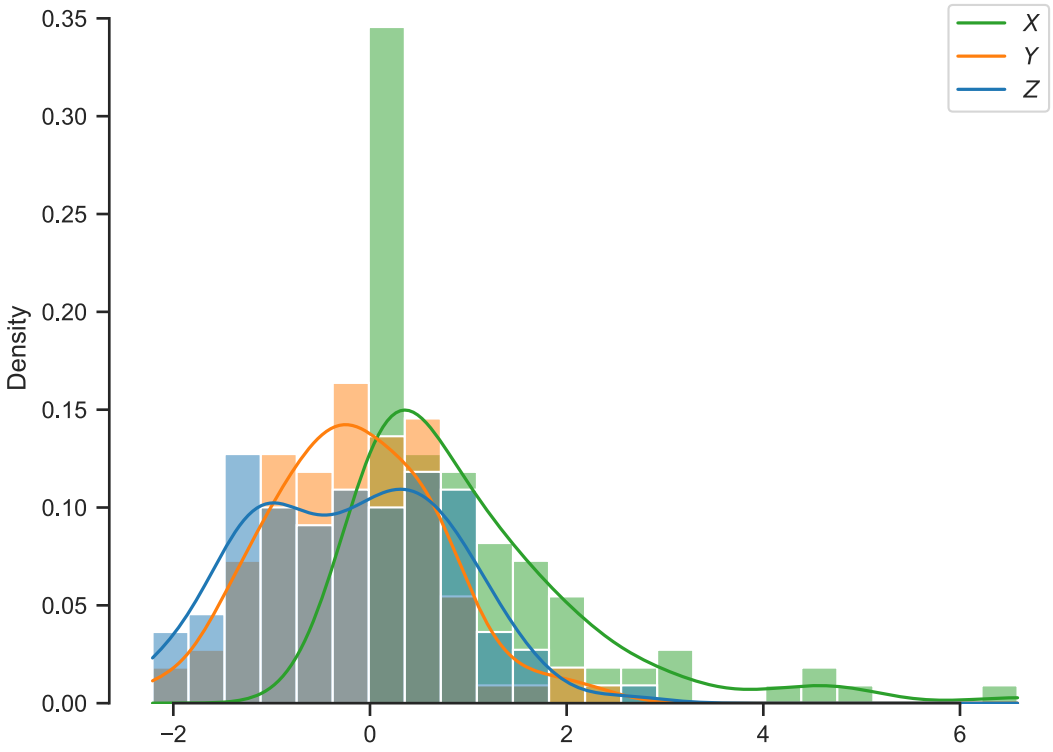
Problem 2 - Zero correlation does not imply independence

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.

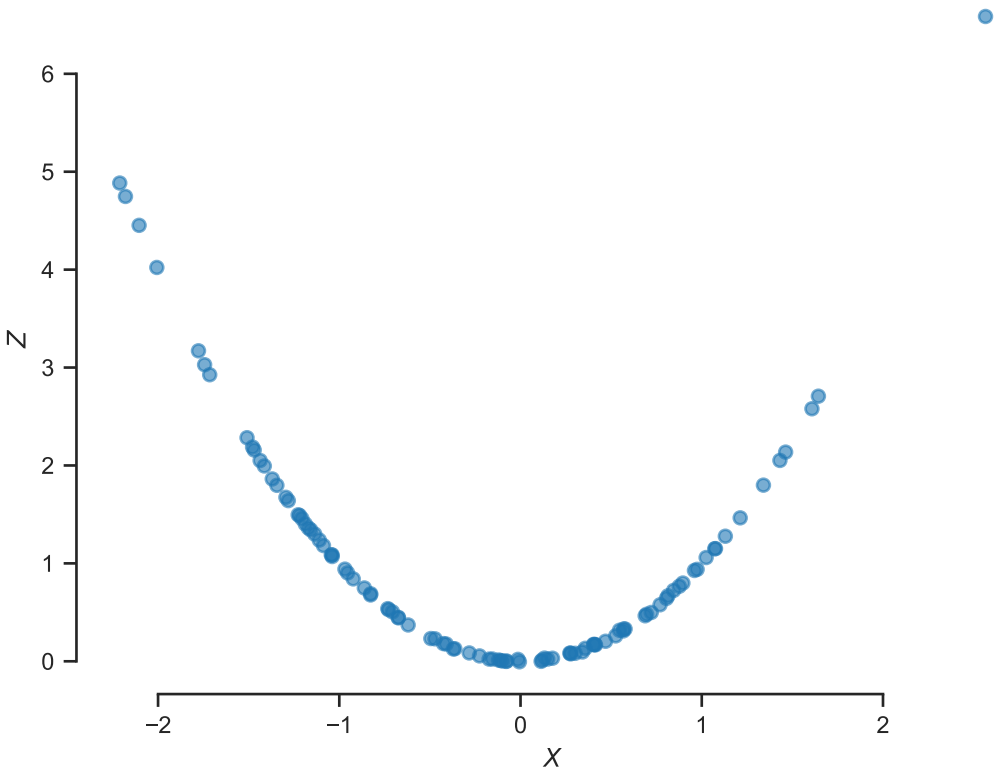
In [16]:
```
unif_norm = st.norm()
```

In [17]:
```
def get_samples(n: int, seed: int = 123456) -> tuple[NDArray]:
    x = unif_norm.rvs(n, seed)
    y = unif_norm.rvs(n, seed * 7)
    z = x**2 + 0.01 * y
    return x, y, z


N = 100
x_vals, y_vals, z_vals = get_samples(N)

sns.histplot([x_vals, y_vals, z_vals], kde=True, stat="density")
plt.legend(["$X$", "$Y$", "$Z$"])
sns.despine(trim=True)
```



B. Do the scatter plot between X and Z.

In [18]:
```
ax = plt.axes(xlabel="$X$", ylabel="$Z$")
ax.plot(x_vals, z_vals, "o", alpha=0.6)
sns.despine(trim=True)
```



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

Answer:

The plot shows a clear relationship between X and Z as they all fall on a parabolic shape, and therefore are not independent.

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

In [19]:
```
V_Z = z_vals.var()
print(f"V[Z] = {V_Z:.3f}")
```

```
V[Z] = 1.461
```

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

In [20]:
```
I_X = x_vals.mean()
I_Z = z_vals.mean()
C_XZ = ((x_vals - I_X)[:, np.newaxis] * (z_vals - I_Z)).sum()
print(f"C[X, Z] = {C_XZ:.5f}")
```

```
C[X, Z] = 0.00000
```

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

In [21]:
```
V_X = x_vals.var()
rho_XZ = C_XZ / np.sqrt(V_X * V_Z)
print(f"ρ[X, Z] = {rho_XZ:.5f}")
```

```
ρ[X, Z] = 0.00000
```

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.

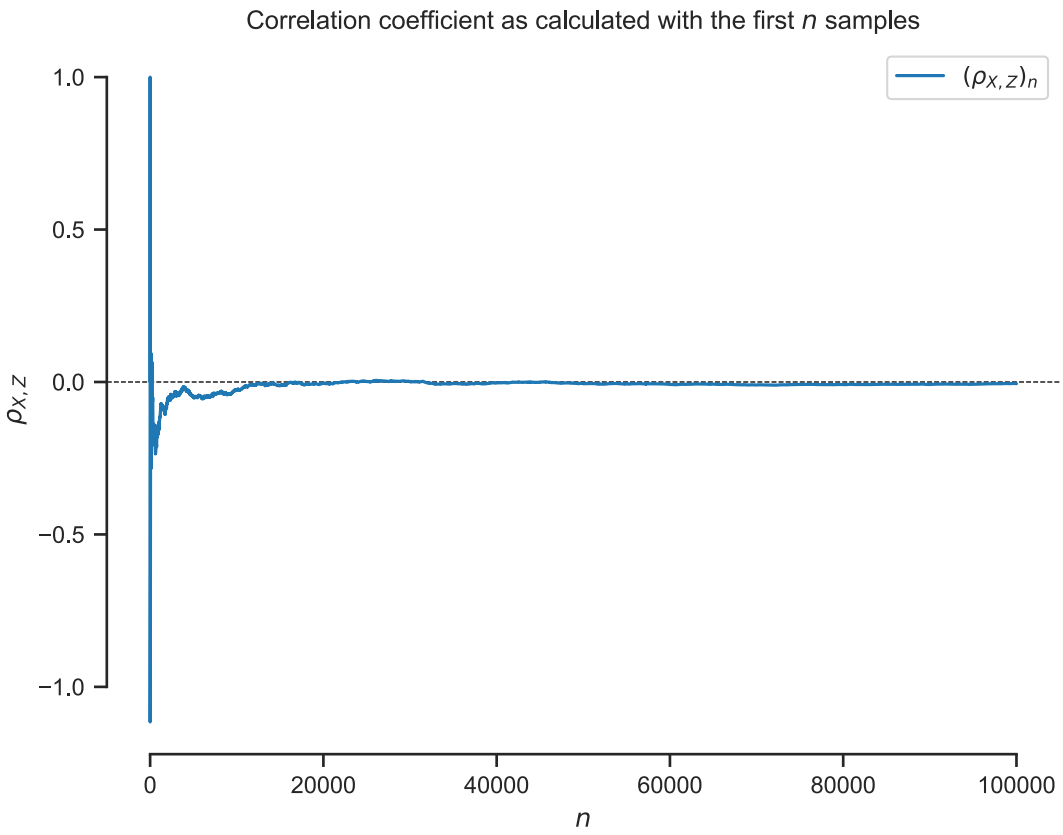
In [22]:
```
N = 1000
x_vals, _, z_vals = get_samples(1000)
E_X = x_vals.mean()
E_Z = z_vals.mean()
C_XZ = 1 / N * ((x_vals - E_X)[:, np.newaxis] * (z_vals - E_Z)).sum()

V_Z = z_vals.var()
V_X = x_vals.var()
rho_XZ = C_XZ / np.sqrt(V_X * V_Z)
print(f"ρ[X, Z] = {rho_XZ:.5f}")
```

```
ρ[X, Z] = -0.00000
```

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?

In [23]:
```
N = 100_000
x_vals, _, z_vals = get_samples(N)
I_running_x = np.cumsum(x_vals) / np.arange(1, N + 1)
I2_running_x = np.cumsum(x_vals**2) / np.arange(1, N + 1)
V_X_running = I2_running_x - I_running_x**2

I_running_z = np.cumsum(z_vals) / np.arange(1, N + 1)
I2_running_z = np.cumsum(z_vals**2) / np.arange(1, N + 1)
V_Z_running = I2_running_z - I_running_z**2

C_XZ_running = np.zeros(N)
rho_XZ_running = np.zeros(N)
for n in range(1, N):
    C_XZ_running[n] = (
        1 / n * ((x_vals[:n] - I_running_x[n]) * (z_vals[:n] - I_running_z[r
    )
    rho_XZ_running[n] = C_XZ_running[n] / np.sqrt(V_X_running[n] * V_Z_runni

ax = plt.axes(
    xlabel="$n$",
    ylabel=r"$\rho_{X,Z}$",
    title=r"Correlation coefficient as calculated with the first $n$ samples
)
ax.axhline(0, color="black", linestyle="--", linewidth=0.5)
ns = np.arange(1, N + 1)
ax.plot(ns, rho_XZ_running, label=r"$(\rho_{X,Z})_n$")
ax.legend()
sns.despine(trim=True)
```



It takes around 15,000 samples for the estimation to be very zero, disregarding the few near zero estimations we observed by chance with very low values of n.

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:

In [24]:
```
url = "https://github.com/PredictiveScienceLab/data-analytics-se/raw/master/
download(url)
```

If everything worked well, then the following will work:

In [25]:
```
B_data = np.loadtxt("B_data.csv")
```

The shape of this dataset is:

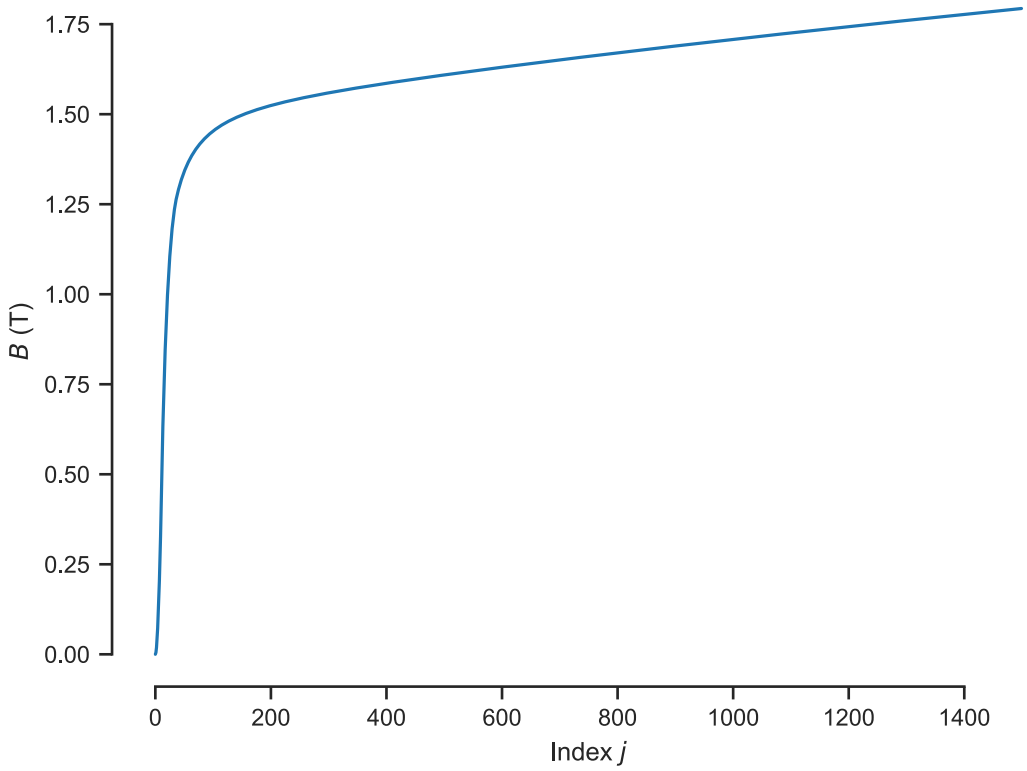
In [26]:
```
B_data.shape
```

Out[26]:
```
(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.

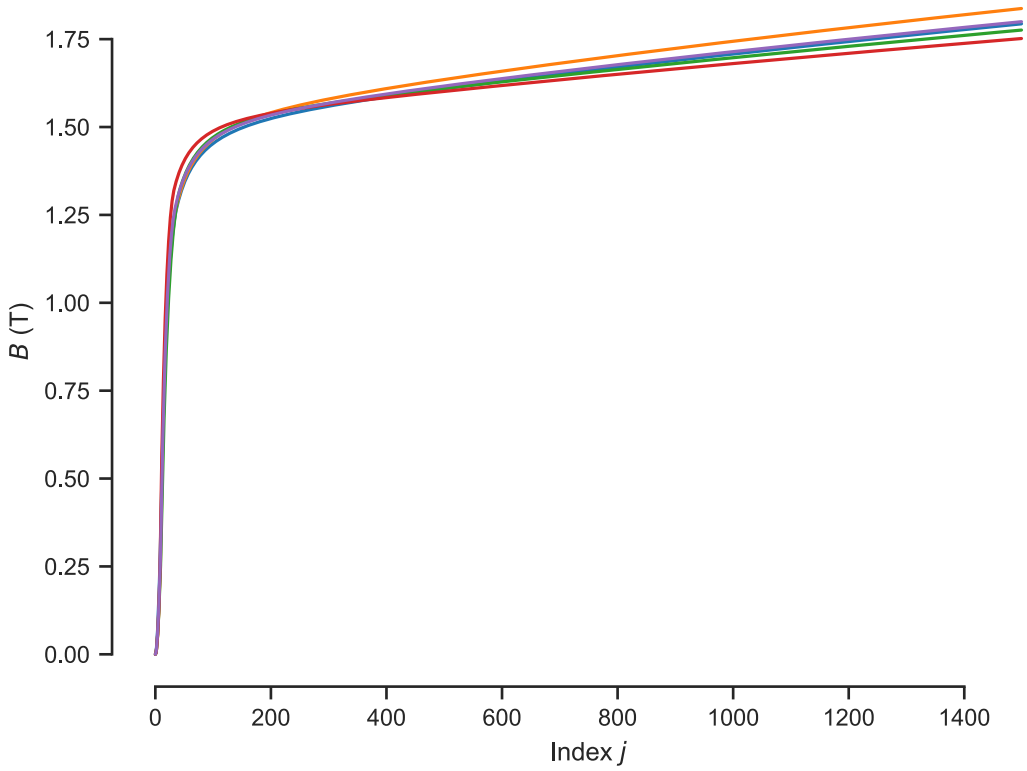
Here is one sample:

In [27]:
```
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)
```



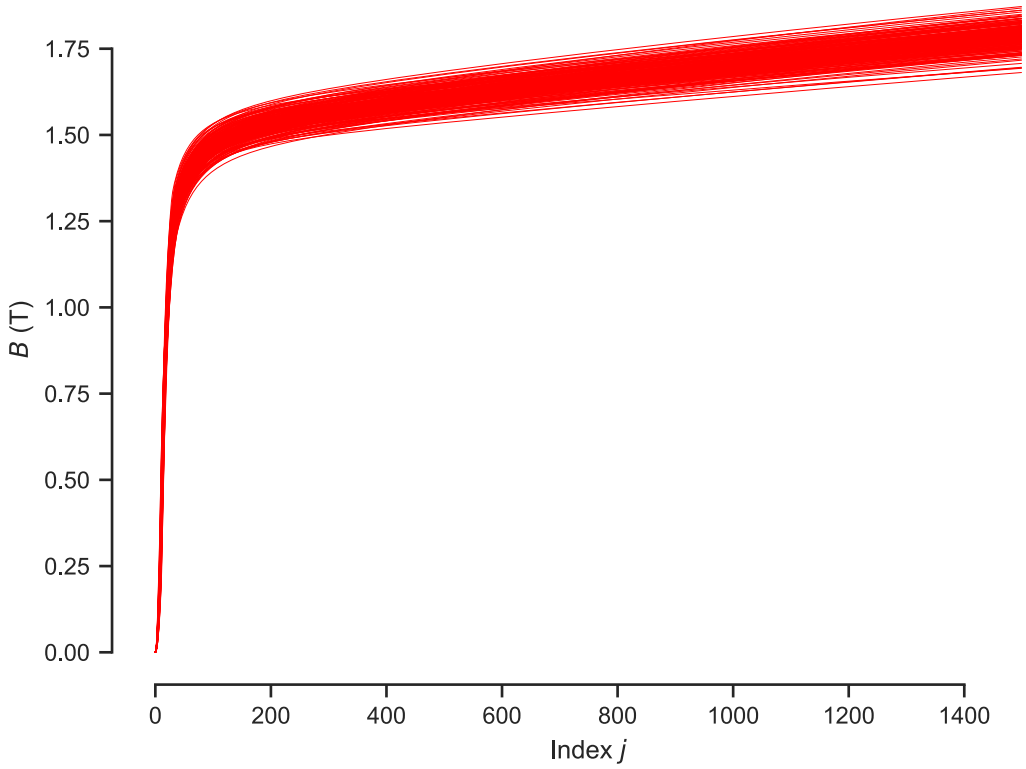
Here are five samples:

In [28]:
```
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:

In [29]:
```
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)
```

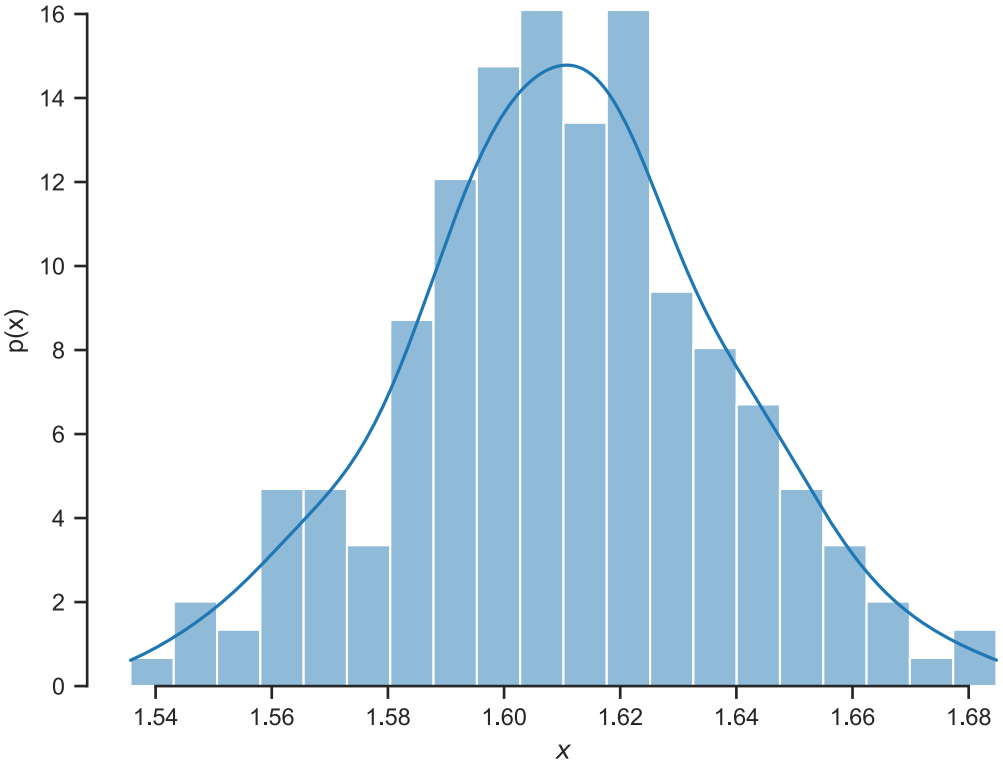


A. We are going to start by studying the data at only one index. Say index $j = 500$. Let's define a random variable

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

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

In [30]:
```
X_data = B_data[:, 500]
ax = plt.axes(xlabel="$x$", ylabel="p(x)")
sns.histplot(X_data, kde=True, stat="density", bins=20, ax=ax)
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:

In [31]:
```
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:

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

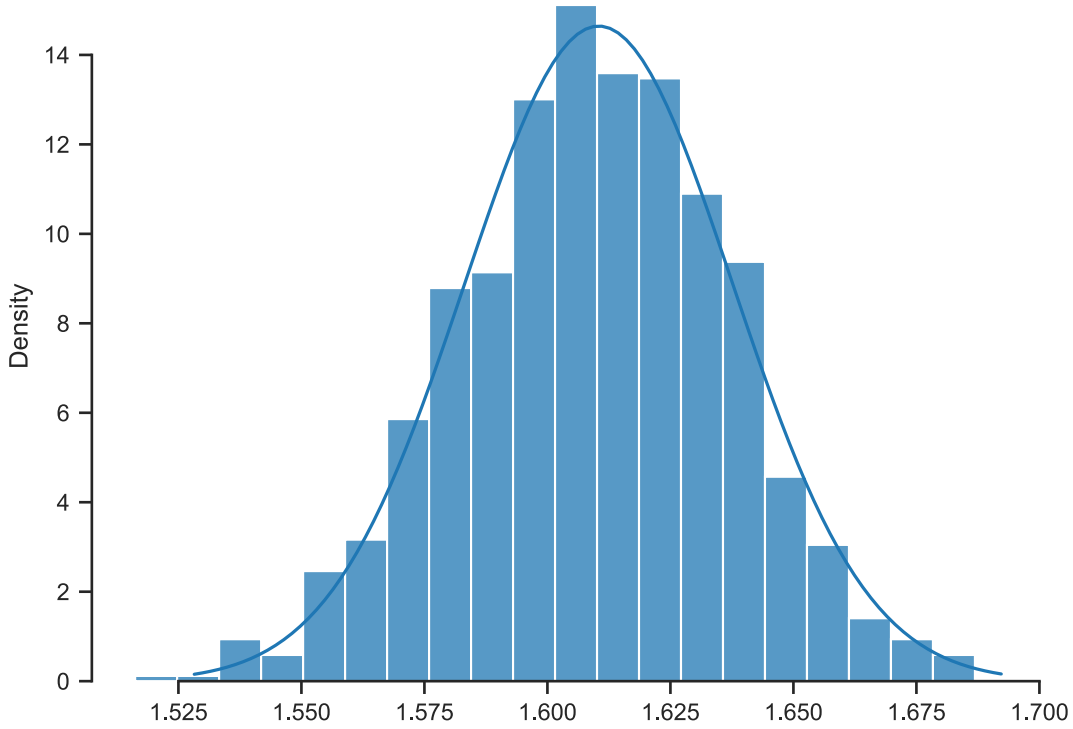
This also converges to the true variance as $N \to \infty$. Here it is:

In [32]:
```
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`.

In [33]:
```
std_500 = np.sqrt(sigma2_500)
X_est = st.norm(mu_500, std_500)

lims = mu_500 + std_500 * 4 * np.array([-1, 1])
x = np.linspace(*ax.get_xlim(), 100)
ax = sns.lineplot(x=x, y=X_est.pdf(x))

seed = 753574
x_samples = X_est.rvs(1000, seed)
sns.histplot(x_samples, stat="density", bins=20, ax=ax)
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.

In [34]:
```
print(f"P(B(H_500) > 1.66) = {1 - X_est.cdf(1.66):.3f}")
```

```
P(B(H_500) > 1.66) = 0.034
```

C. Let us now consider another random variable

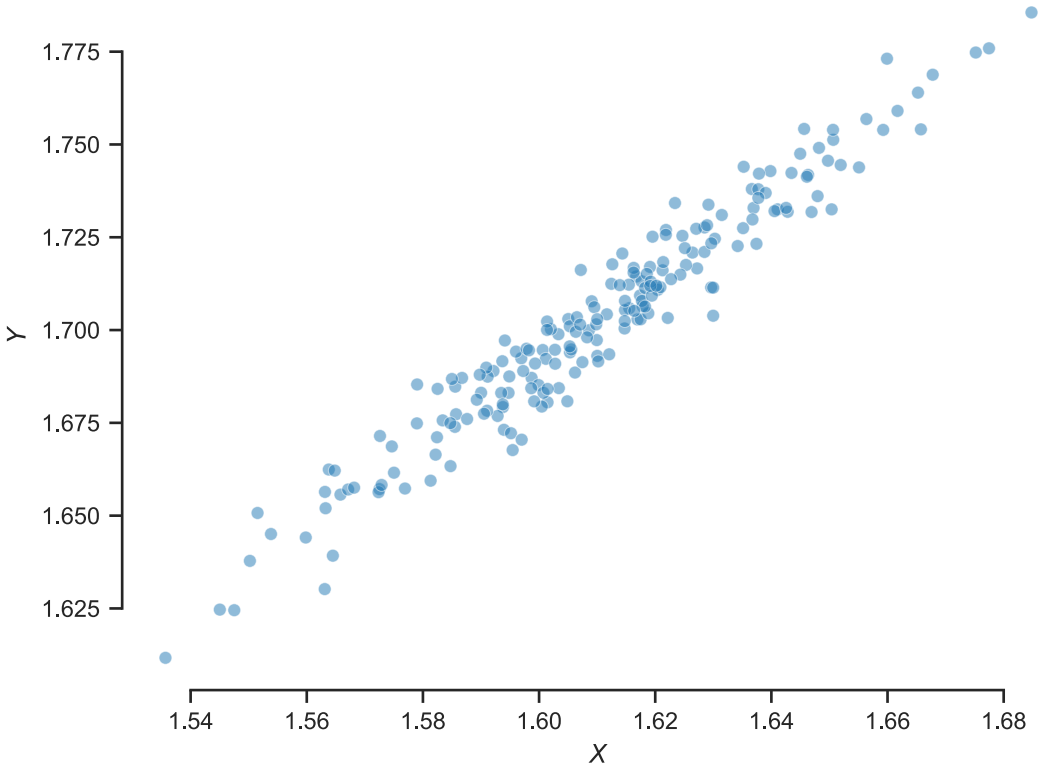
$$Y = B(H_{1000}).$$

Isolate the data for this as well:

In [35]:
```
Y_data = B_data[:, 1000]
```

Do the `scatter` plot of X and Y:

In [36]:
```
ax = plt.axes(xlabel="$X$", ylabel="$Y$")
sns.scatterplot(x=X_data, y=Y_data, alpha=0.5, ax=ax)
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).

In [37]:
```
XY_data = np.vstack([X_data, Y_data]).T
```

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:

In [38]:
```
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:

In [39]:
```
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.

In [40]:
```
V_X = C_XY[0, 0]
V_Y = C_XY[1, 1]
rho_XY = C_XY[0, 1] / np.sqrt(V_X * V_Y)
print(f"ρ[X, Y] = {rho_XY:.5f}")
```

```
ρ[X, Y] = 0.97061
```

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

Answer:

In [41]:
```
if rho_XY > 0:
    print("X and Y are positively correlated")
elif rho_XY < 0:
    print("X and Y are negatively correlated")
else:
    print("X and Y are not correlated")
```

```
X and Y are positively correlated
```

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

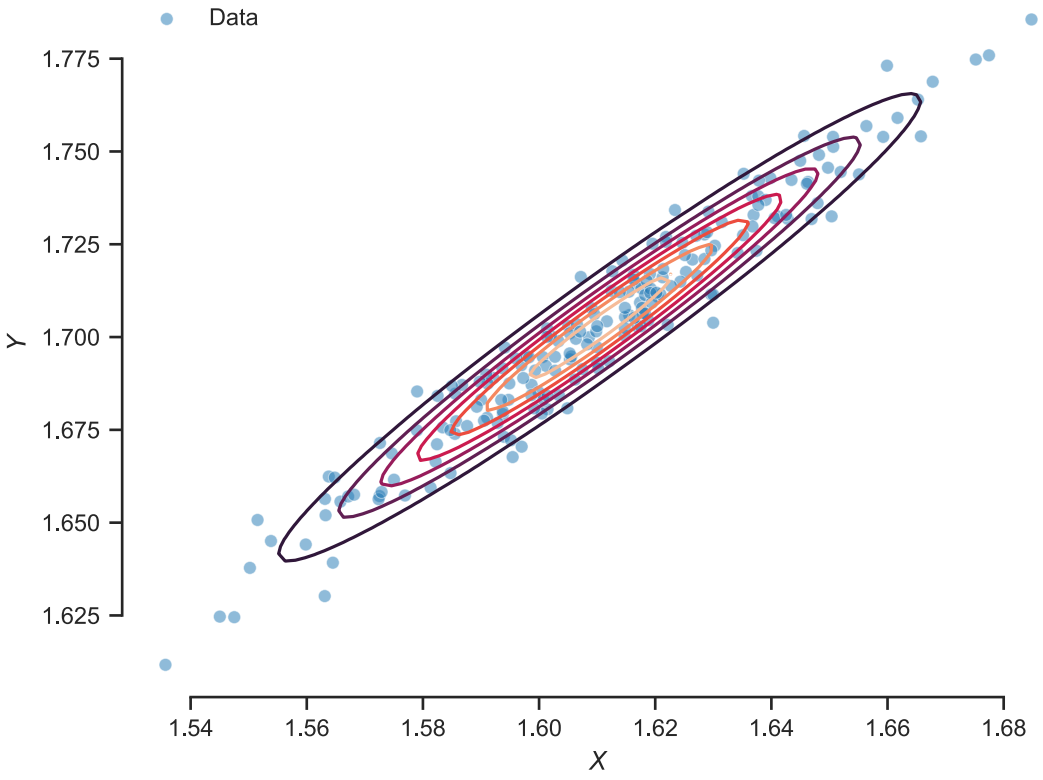
In [42]:
```
def is_symmetric(M: NDArray) -> bool:
    return np.allclose(M, M.T)


def has_positive_eigenvalues(M: NDArray) -> bool:
    return np.all(np.linalg.eigvals(M) > 0)


def is_positive_definite(M: NDArray) -> bool:
    return is_symmetric(M) and has_positive_eigenvalues(M)
```

```
print(is_positive_definite(C_XY))
```

```
True
```

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.

In [43]:
```
XY = st.multivariate_normal(mu_XY, C_XY)
ax = plt.axes(xlabel="$X$", ylabel="$Y$")
sns.scatterplot(x=X_data, y=Y_data, alpha=0.5, ax=ax, label="Data")

N = 100
x = np.linspace(*ax.get_xlim(), N)
y = np.linspace(*ax.get_ylim(), N)
X, Y = np.meshgrid(x, y)
Z = XY.pdf(np.dstack((X, Y))).reshape(N, N)
c = ax.contour(X, Y, Z)
ax.legend(frameon=False)
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.

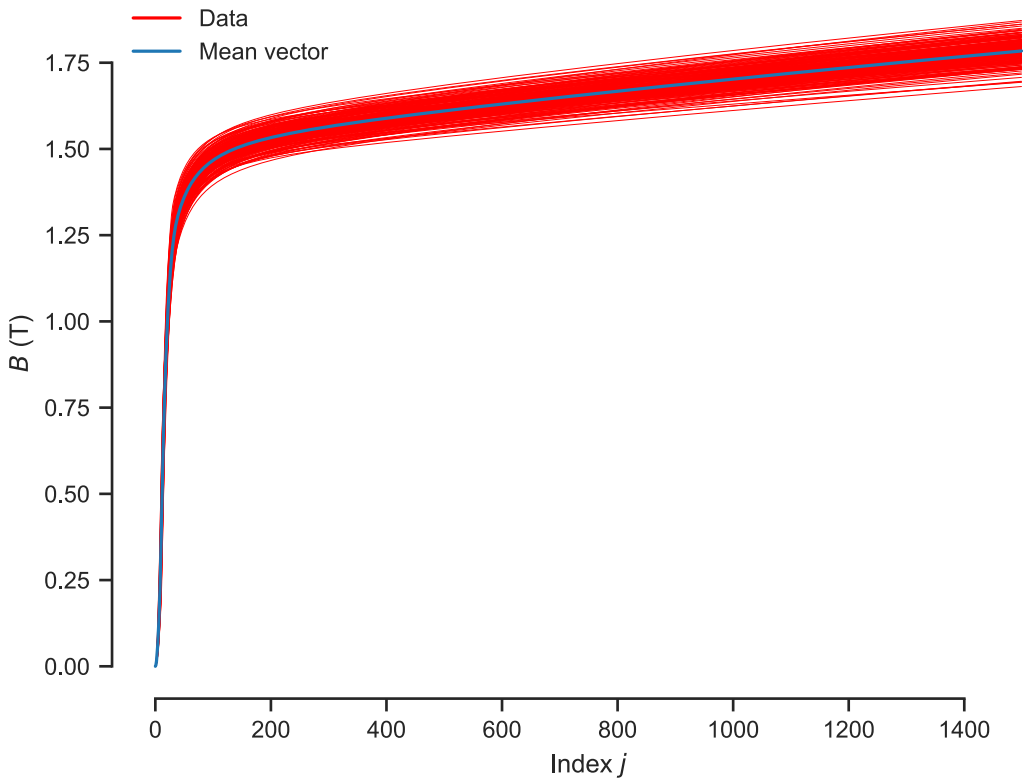
First, let's find the mean of this random vector:

In [44]:
```
B_mu = np.mean(B_data, axis=0)
B_mu
```

Out[44]:
```
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:

In [45]:
```
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)
```



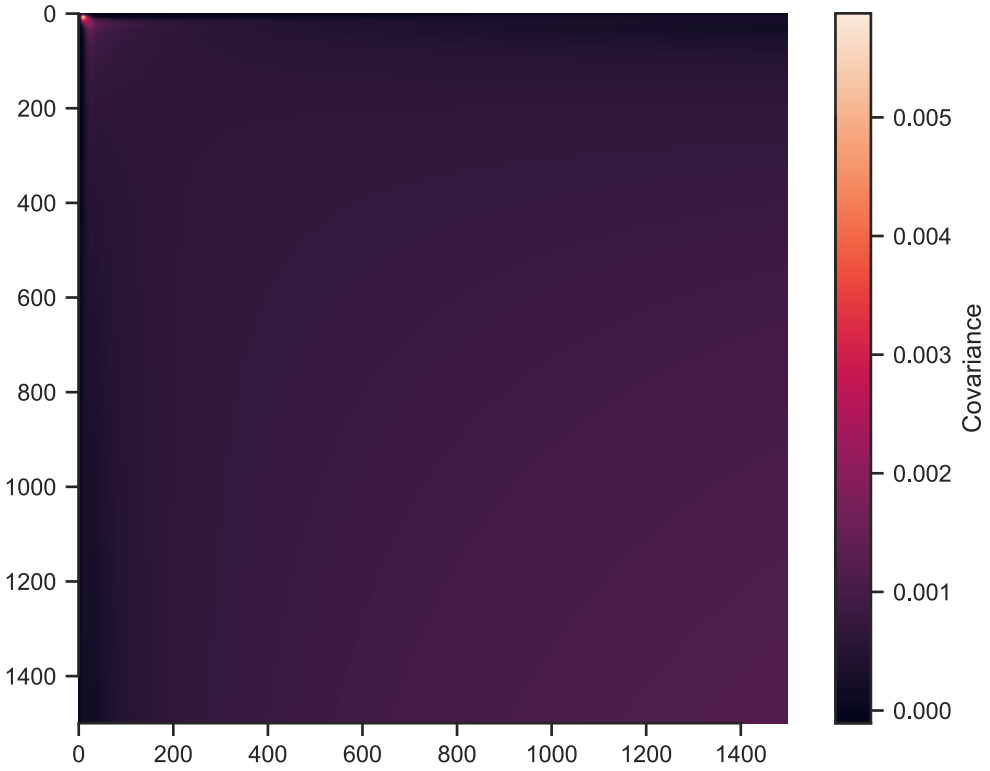
It looks good. Now, find the covariance matrix of $\mathbf{B}$. This is going to be a 1500x1500 matrix.

In [46]:
```
B_cov = np.cov(B_data.T)
B_cov
```

Out[46]:
```
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:

In [47]:
```
fig, ax = plt.subplots()
c = ax.imshow(B_cov, interpolation="nearest")
plt.colorbar(c, label="Covariance")
sns.despine(trim=True)
```



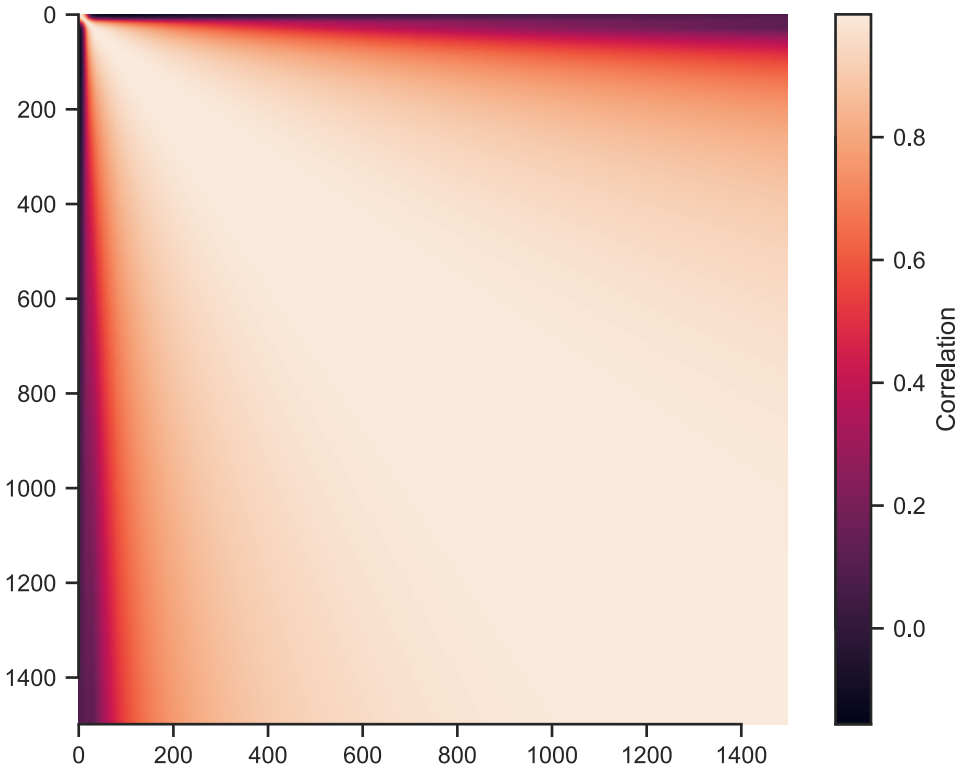
The numbers are very small. This is because the covariance depends on the units of the variables. We need to do the same thing we did with the correlation coefficient: divide by the standard deviations of the variables. Here is how you can get the correlation coefficients:

In [48]:
```
# 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
```

Out[48]:
```
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:

In [49]:
```
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:

In [50]:
```
print("Eigenvalues of B_cov:")
print(np.linalg.eigh(B_cov)[0])
```

```
Eigenvalues of B_cov:
[-5.55723825e-16 -2.80872435e-16 -1.93474497e-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:

In [51]:
```
# B = st.multivariate_normal(mean=B_mu, cov=B_cov)
```

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.

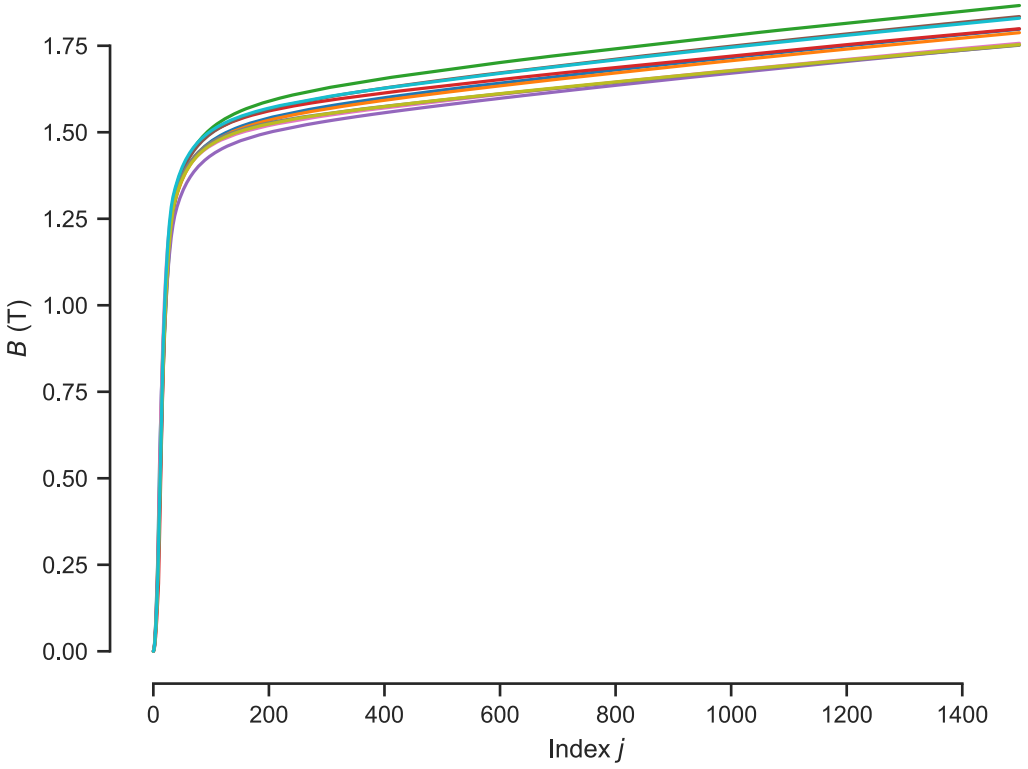
In [52]:
```
def build_B():
    order = -12
    while order < 0:
        nugget = 10**order
        try:
            B_cov = np.cov(B_data.T)
            B_cov_w_nugget = B_cov + nugget * np.eye(B_cov.shape[0])
            B = st.multivariate_normal(mean=B_mu, cov=B_cov_w_nugget)
            print(f"It worked with {nugget = }!")
            return B
        except:
            print(f"It did not work with {nugget = }. Increase nugget by 10x
            order += 1


B = build_B()
```

```
It did not work with nugget = 1e-12. Increase nugget by 10x.
It did not work with nugget = 1e-11. Increase nugget by 10x.
It did not work with nugget = 1e-10. Increase nugget by 10x.
It worked with nugget = 1e-09!
```

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.

In [53]:
```
samples = B.rvs(10)
ax = plt.axes(xlabel="Index $j$", ylabel="$B$ (T)")
```

```
ax.plot(samples.T)
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` .