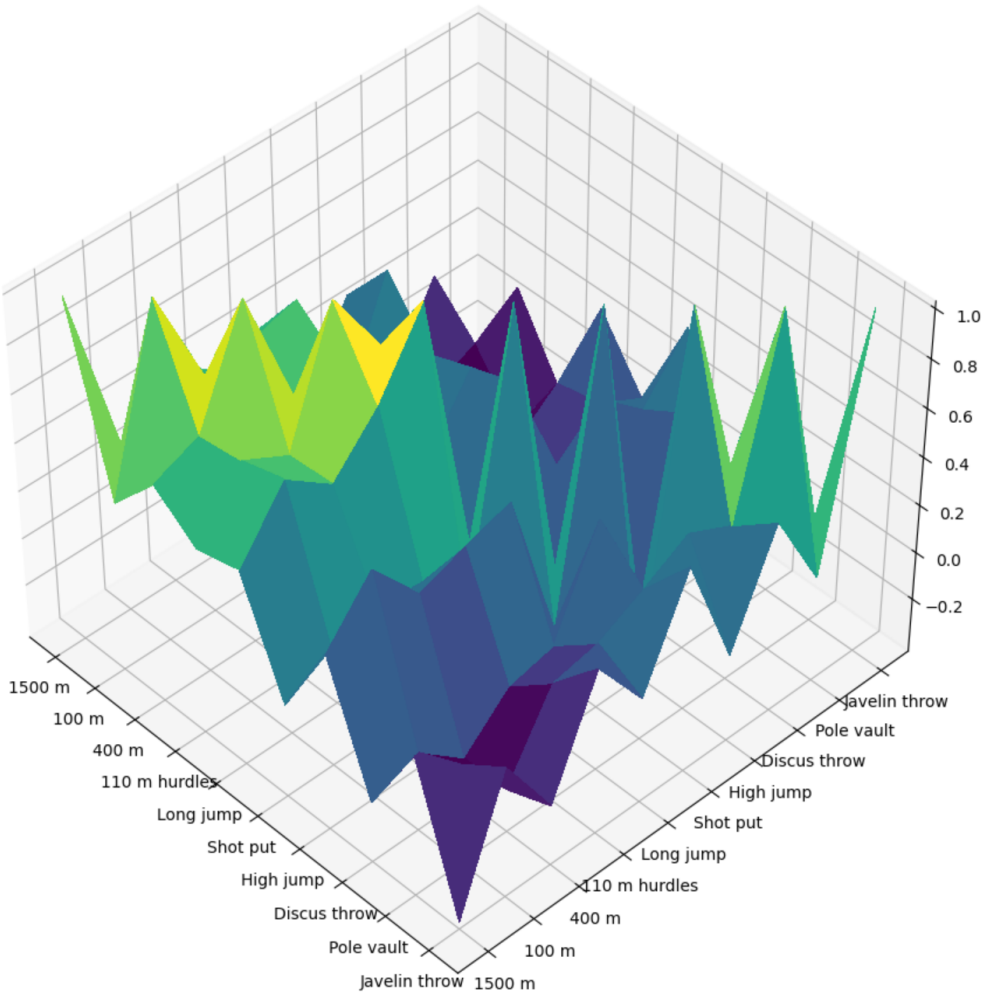
UFR-ST Besançon, Master CompuPhys year 2023-2024

Statistical exploitation of measurements

Practical exercises report

Léo BECHET



Disclaimer : Editor theme changed in between practicals, which explains the different colors in code blocks

Table

Topic 1 : Uncertainty of a political poll

Introduction

Political polls are tools used almost daily by journalists and media to get a global opinion on current world affairs, such as local policies being passed, opinions on world-wide events such as wars, scandals, and more commonly known, elections. However, such tools are based on a sample of the population and are therefore subject to uncertainties. In this subject, we aim to study such uncertainties and clear common misconceptions when exploiting polls.

Note : The outputs presented in this report are made using a code rerun multiple times. Due to the random nature of the code, the results may not be inadequate with other results presented. Interpretations are however exact. If you wish to run the code on your own hardware, you can find it alongside this report or refer to this link github.com/lele394/SEM/tree/main/SEM%20Topic1

First case : Political poll

This case studies the creation of a 1000 boolean sequence. The state has a 50% chance of being True or False. Let's take a look at how they are generated in the code :

```
def random_pull(proba):
    return random() < proba

def random_sequence(n=1000):
    return [random_pull(0.5) for i in range(n)]

seq = random_sequence()

print(f'len {len(seq)}; mean {sum(seq)/len(seq)}')
print(f'number of YES : {sum(seq)}')
```

We define a function that randomly return true or false based on a probability, and use it to construct a 1000 sequence of True and False items. Here's the output of the above code:

Console output :

```
====== 1.1.2 For 1 sequence =====
len 1000; mean 0.502
number of YES : 502
```

We can see that the number of True we get is 502, which is around 50%. This is alright knowing or starting conditions.

Phase 1 : sequence of 1000 booleans

Let's repeat the experiment 500 times and plot the histogram of the number of True per sequence. We use the following code to do so

```
number_of_seq = 500
x = [i for i in range(0,1000)]
y = [0 for i in range(0,1000)]
max, min = 0,1000

for i in range(number_of_seq):
    r = sum(random_sequence())
    y[r] += 1/number_of_seq
    if r > max: max = r
    if r < min: min = r

if True: #toggle
    plt.bar(x,y,  width=1, align='center')
    plt.xlim(min, max)
    plt.title("Distribution of YES for 500 sequenceS")
    plt.xlabel("Number of YES for a sequence of 1000 pulls")
    plt.ylabel("Amount of time it happend for 500 sequences")
    plt.show()
```

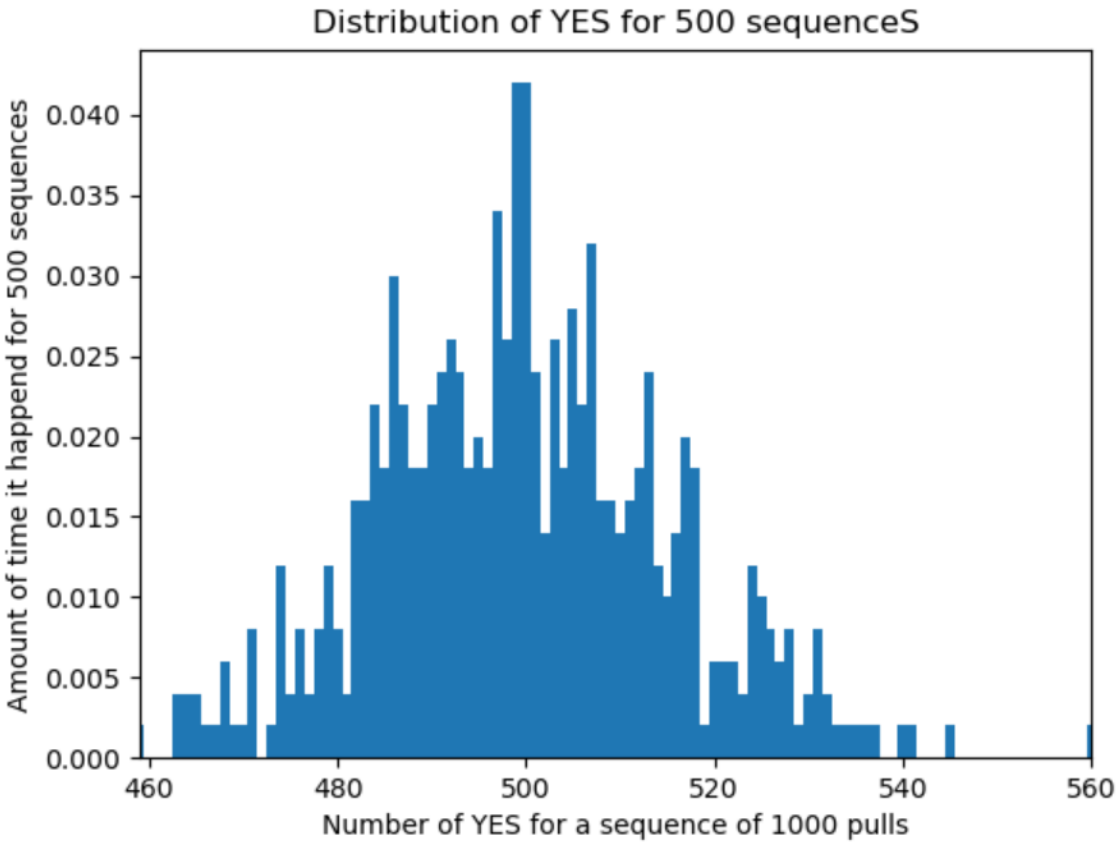


Fig T1.1 : Distribution of YES for 500 tests of 10000 pulls. We can spot an oddity at 560.

Plotting the cumulative probabilities and outputting the 95% confidence interval using the following code gives us :

```
plt.xlim(min, max)
s_proba = []
c=0
for i in y:
    c+= i
    s_proba.append(c)

conf_interval = [0,0]
_s_pass = False
for i in range(len(s_proba)):
    if s_proba[i] > 0.05 and not _s_pass:
        print(f'cumulative sum pass 5% at {i}')
        _s_pass = True
        conf_interval[0] = i

    if s_proba[i] > 0.95:
        print(f'cumulative sum pass 95% at {i}')
```

```
        conf_interval[1] = i
        break

print(f'Confidence interval is between {conf_interval[0]/1000 * 100}%
and {conf_interval[1]/1000 * 100}%. If probability is in this interval
the sequence can be considered random.')

if True: # toggle
    plt.plot(x,s_proba)
    plt.xlabel("Number of YES for a sequence of 1000 pulls")
    plt.ylabel("Cumulative sum of probabilites")
    plt.title("Cumulative probabilities")
    plt.show()
```

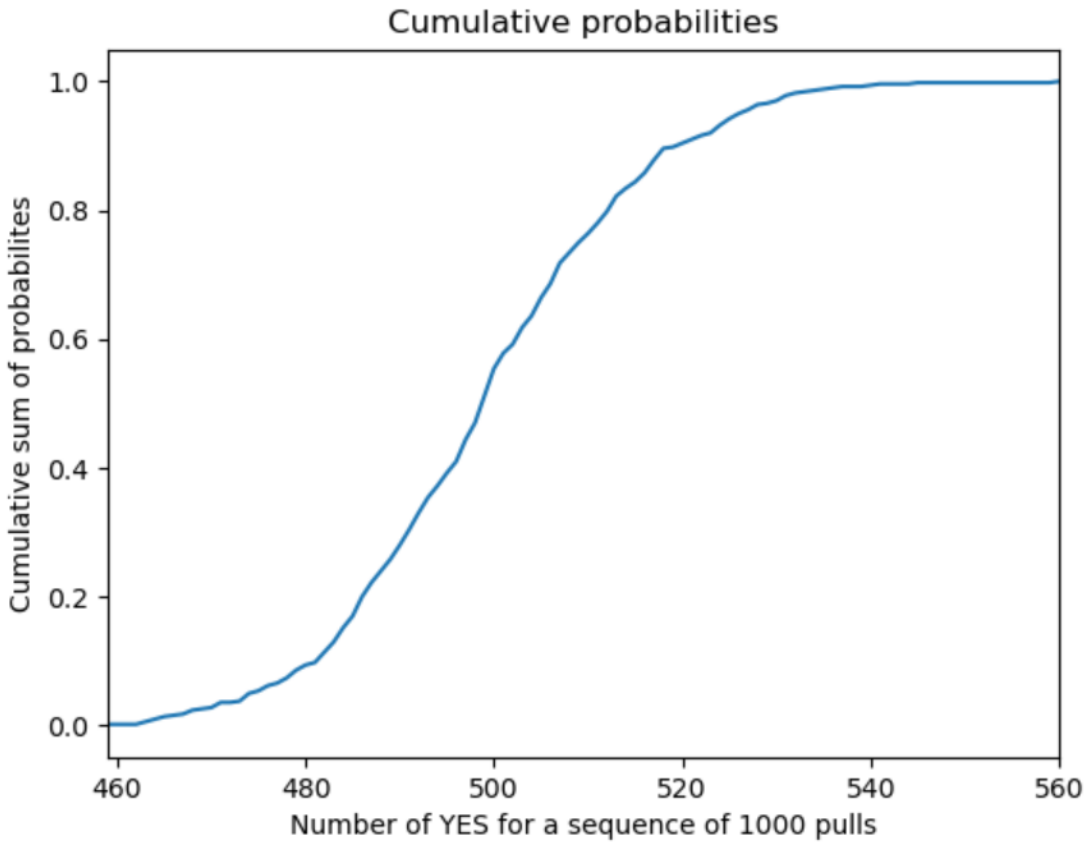


Fig T1.2 : Cumulative probabilities from the fig T1.1

Console output :

```
cumulative sum pass 5% at 475
cumulative sum pass 95% at 526
Confidence interval is between 47.5% and 52.6%. If probability is in this
interval the sequence can be considered random.
```

When comparing the theory, one will find, using the mean estimator, that the standard-mean is divided by around 10 compared to the previous one. This is due to the fact that we repeat 1000 measures. The division factor can be explained as the square root of 1000 is 10.

Phase 2 : sequence of 4000 booleans

Everything of the Phase 1 is redone here with a slight modification, switching from 1000 people per polls to 4000. If you have any question on how results are obtained, please refer to the corresponding part in phase 1.

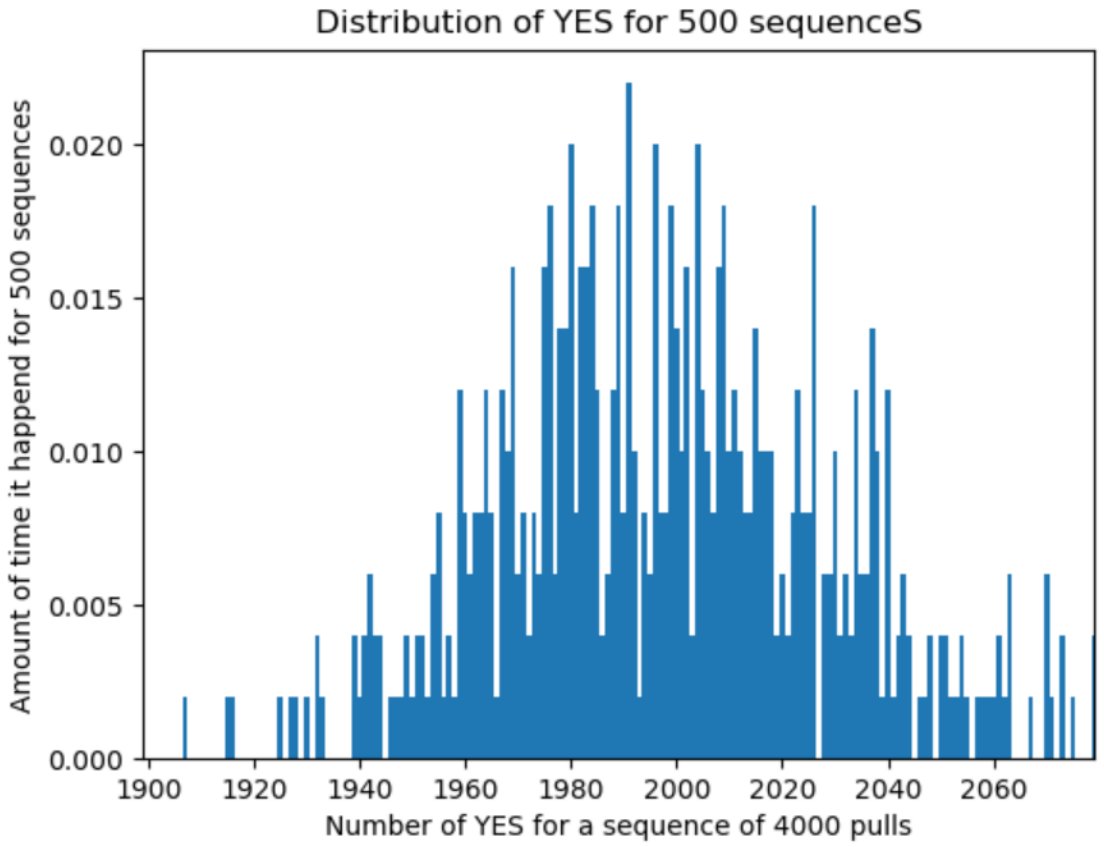


Fig T1.3 Distribution of YES for 500 sequences of 40000 pulls. It looks centered around 200

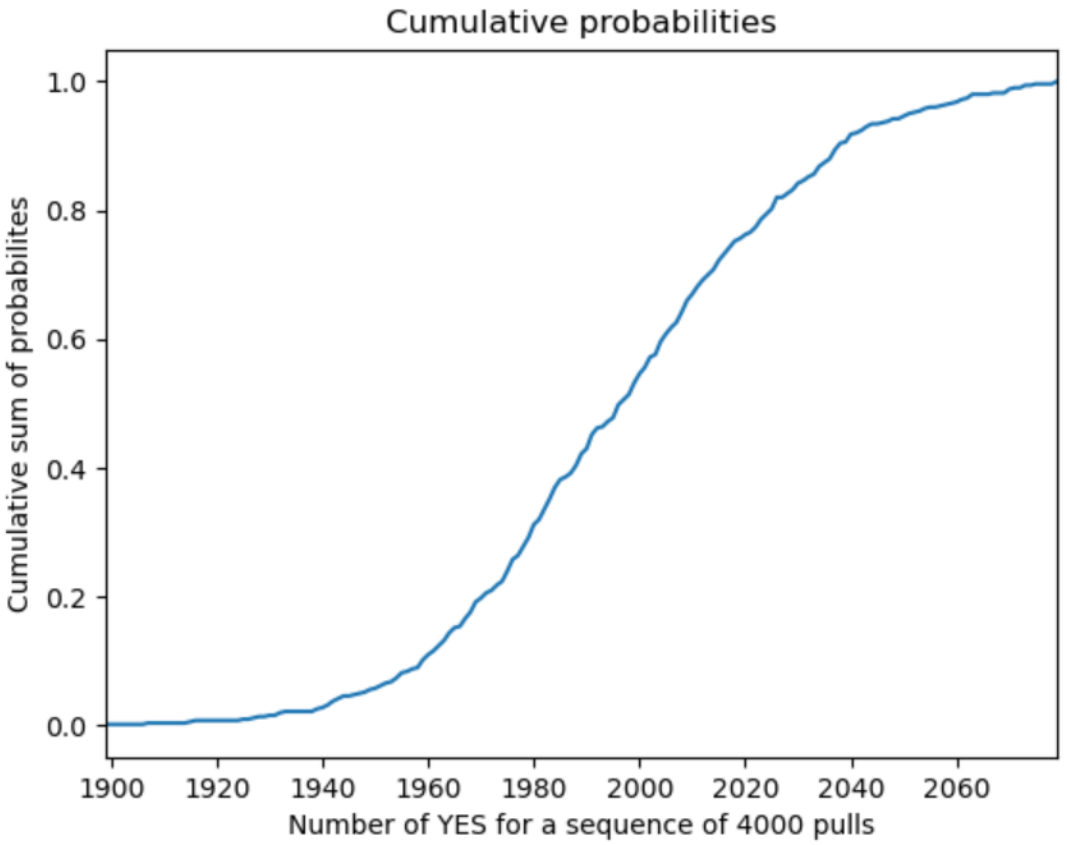


Fig T1.4 : Cumulative probabilities from Fig T1.3

Console output :

```
 ===== REDOES THE PREVIOUS TESTS BUT WITH POLLS OF 4000 PEOPLE =====
cumulative sum pass 5% at 1948
cumulative sum pass 95% at 2051
Confidence interval is between 48.699999999999996% and 51.275000000000006%.
If the probability is in this interval the sequence it can be considered
random.
```

We can compare the confidence interval with the one from phase 1. We can deduce that the higher the number of pull per sequence, the smaller the confidence interval. This is logical as when the number of tests increases, the more it is representative of the overall population. In this case the population has to be 50% True as per the random pull probability set. However this model.

We can draw a conclusion on political poll. If we consider a politician which popularity is of exactly 50%, and use polls with a 2% margin error, a poll made on a certain day returning 49% popularity and a second poll made a week later returning 51% popularity does not expose that the politician's popularity grew by 2% in a week. It is absolutely impossible to compute the exact popularity using polls. Media's use of poll can be extremely misleading to people not familiar with statistical tools. This can be used in multiple ways, including propaganda.

Second case : Measure of a second

Now let's take a look at the simulation of the measure of a second. In our case we consider that the uncertainty is around plus or minus 0.02 second. Considering that we take a 95% confidence interval, that gives us a standard deviation of 0.01 second. Let's now simulate 1000 measurements using the standard deviation given above. The following code will be used in this part.

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import norm

measures = 1000

def generate_data(mean, std_dev, num_values):
   data = norm.rvs(mean, std_dev, num_values)
   return data

pulls = generate_data(0, 0.01, measures)

# Create a histogram
hist, bins, __ =plt.hist(pulls, bins=30, alpha=0.7)
plt.xlabel('Value')
plt.ylabel('Frequency')
plt.title('Histogram of Pulls')
plt.grid(True)
```

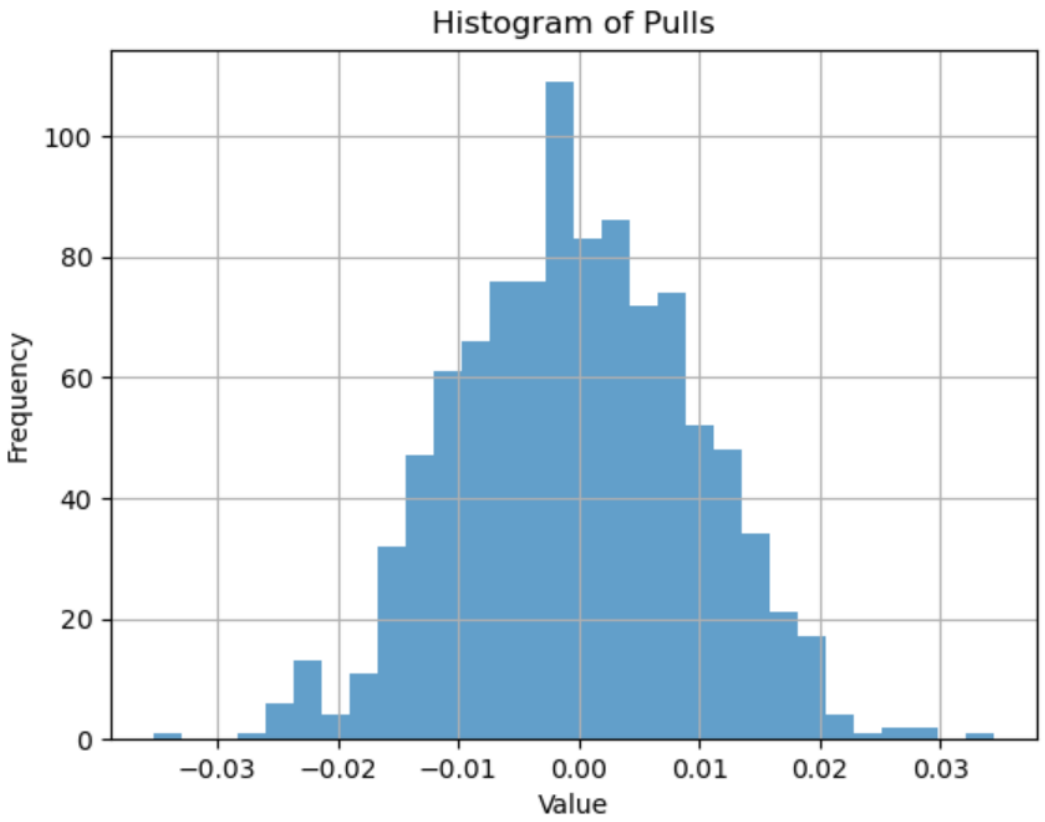


Fig T1.5 : Histogram of 1000 normally distributed tests

This histogram shows the deviation from the exact measurement. The shape of the histogram does give a gaussian curve. Now let's take a look at the cumulative sum in order to compute the standard deviation of the sequence. The following code will be used.

```
# Calculate the cumulative sum
cumulative_sum = [i/measures for i in np.cumsum(hist)]

# Create a plot for the cumulative sum
plt.figure()
plt.plot(bins[:-1], cumulative_sum, marker='o')
plt.xlabel('Value')
plt.ylabel('Cumulative Sum')
plt.title('Cumulative Sum of Histogram')

# get std-dev from the experimental pulls
std_dev = 0
for i in range(len(cumulative_sum)):
   if cumulative_sum[i]> 0.95:
       std_dev = bins[i]/2 #conf int = 2*sigma
       break
```

```
print(f'Std-dev for 1000 experimental pull is {std_dev}')
```

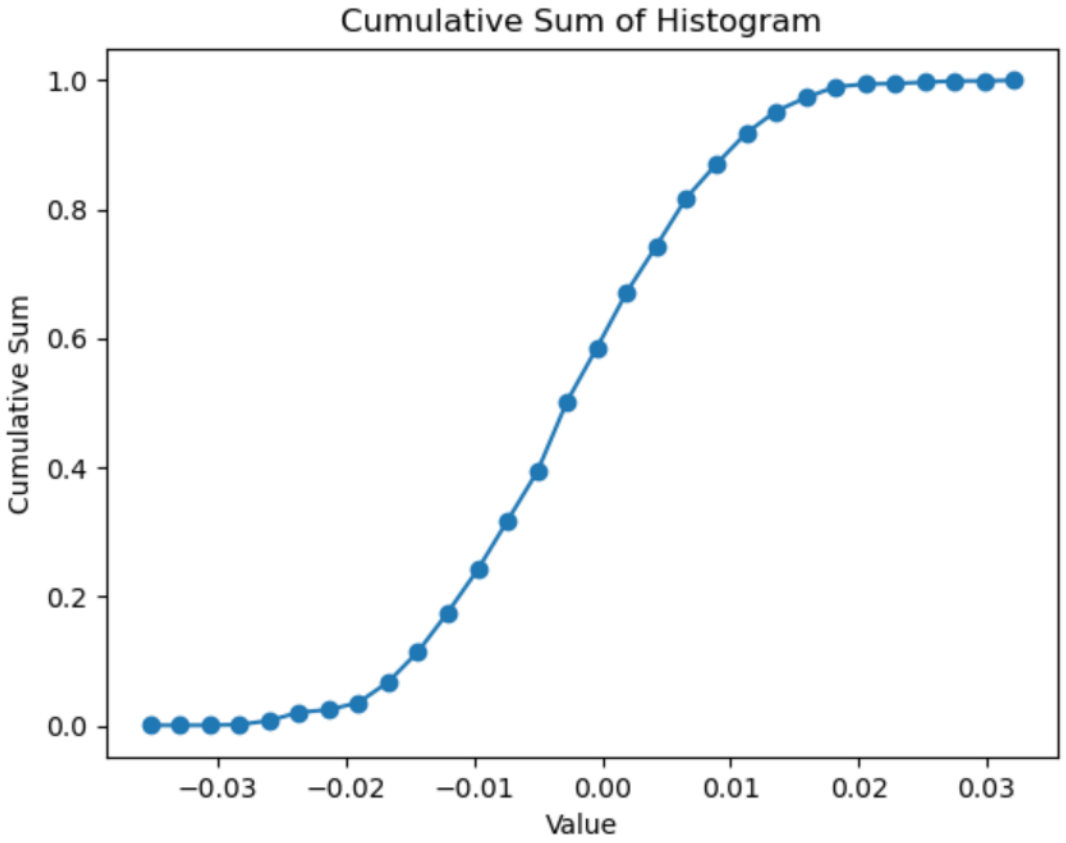


FIg T1.6 : Cumulative sum of Fig T1.5.

Console output :

```
Std-dev for 1000 experimental pull is 0.006769953655205423
```

The computed standard deviation is of around 0.007, which is close to the value computed above (0.01). We can conclude that the measurement does fit in the 95% confidence interval as 0.007 < 0.01.

Let's now take a look at the arithmetical average of two sequences of 100 measurements. The following code will be used for this case.

```
data = generate_data(0, 0.01, 100)
print(f'mean of the first 100 pull sequence : {sum(data)/len(data)}')
data = generate_data(0, 0.01, 100)
print(f'mean of the 2nd 100 pull sequence : {sum(data)/len(data)}')
```

Console output :

```
mean of the first 100 pull sequence : -0.0021538705648827395
mean of the 2nd 100 pull sequence : 0.0016722558753093592
```

As expected, different sequences do not produce the same average. This is to be expected as pulls are done randomly, arithmetic averages can be considered as random variables as well. The lower the number of measurements, and the higher the average difference between 2 sequences will be high. Let's study it by characterizing its standard deviation and confidence interval. This will be done the same way we did it previously, using the code below.

```
measures = 100
means = []
for means_to_do in range(1000):
   data = generate_data(0, 0.01, measures)
   means.append(sum(data)/len(data))

# Create a histogram
hist, bins, __ =plt.hist(means, bins=30, alpha=0.7)
plt.xlabel('Value')
plt.ylabel('Frequency')
plt.title('Histogram of Means')
plt.grid(True)
plt.show()

cumulative_sum = [i/measures for i in np.cumsum(hist)]

# Create a plot for the cumulative sum
plt.figure()
plt.plot(bins[:-1], cumulative_sum, marker='o')
plt.xlabel('Value')
plt.ylabel('Cumulative Sum')
plt.title('Cumulative Sum of Histogram')
plt.show()

# get std-dev from the experimental pulls
std_dev = 0
for i in range(len(cumulative_sum)):
   if cumulative_sum[i]> 0.95:
       std_dev = bins[i]/2 #conf int = 2*sigma
       break

print(f'Std-dev of mean for 1000 sequence of 100 pull : {std_dev}')
print(f'Confidence interval is equal to std-dev*2 : {std_dev*2}')
```

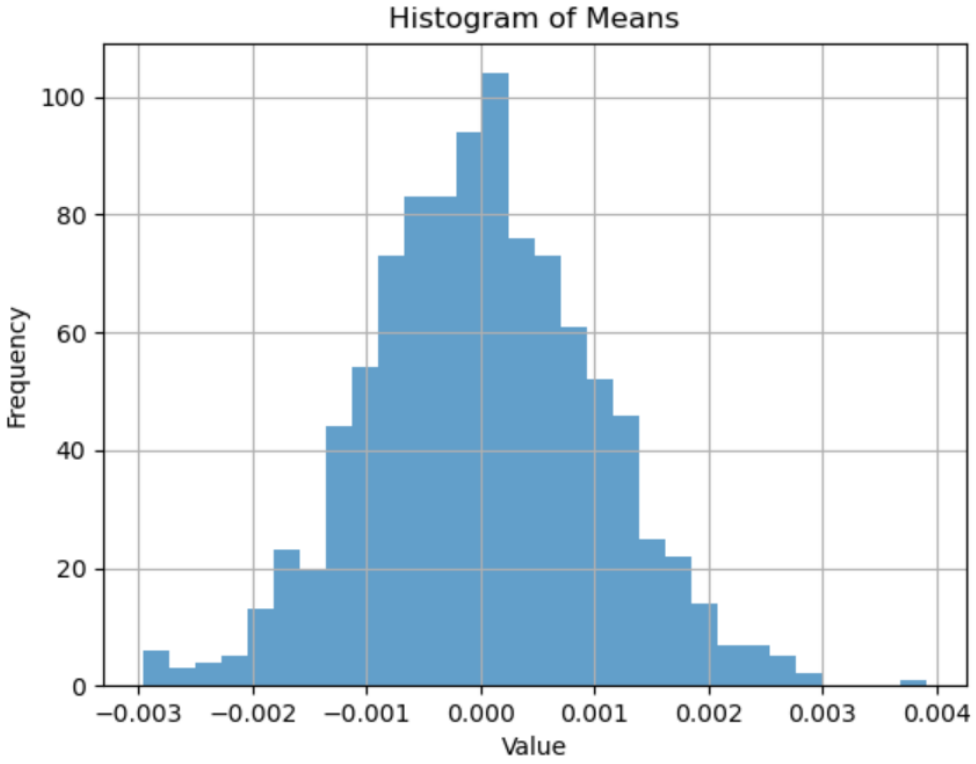


Fig T1.7 : Histogram of 1000 series of 100 measurements

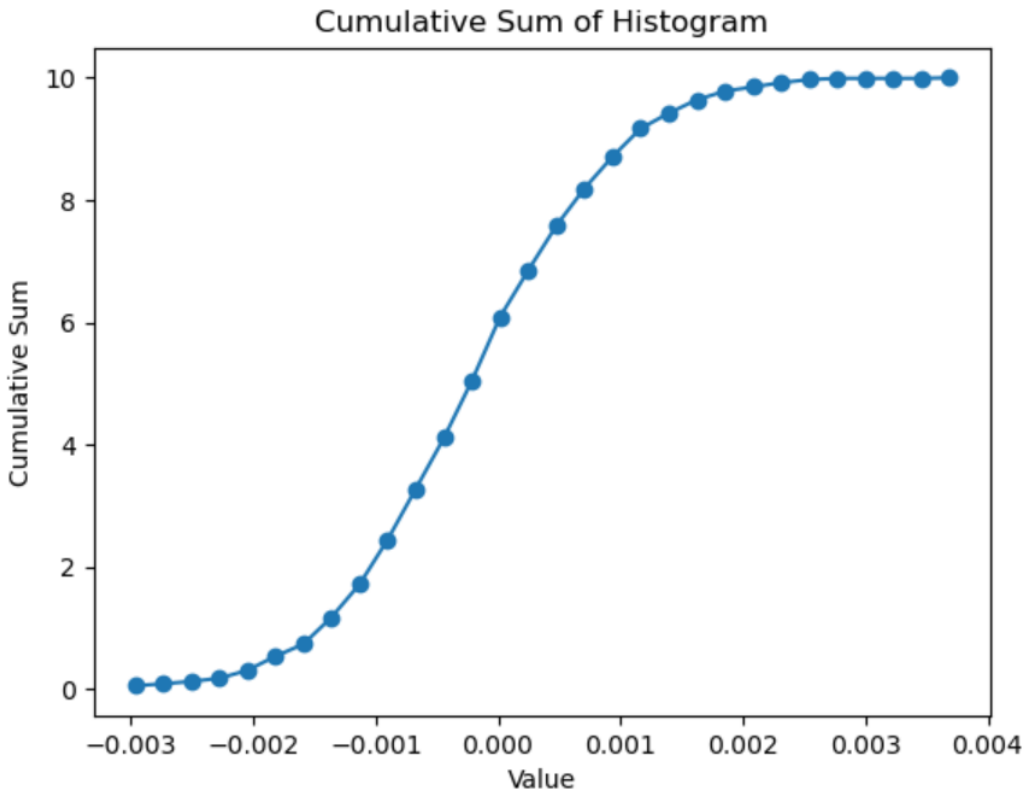


Fig T1.8 : Cumulative some of Fig T1.7

Console output :

```
Std-dev of mean for 1000 sequence of 100 pull : -0.0006799270521168299
```

```
Confidence interval is equal to std-dev*2 : -0.0013598541042336599
```

We plotted the usual graphs and made the usual calculations, we find that the standard deviation is around -0.0007 and that our confidence interval is around -0.001. One interpretation we can draw is that when measuring a second using a 100 measures sequence, we are 95% sure that the real measure is in plus or minus 0.001s away from what the average is. We can also say that on average, averages will be around 0.0006s away from the real measure. These informations gives us hindsights on what the real value is and allows us to build statistical models that can later be used in other studies.

Note :

Precision seems to be impacted by the number of bins (here 30) with this implementation. A higher number of bins may be advisable in other situations.

Topic 2: χ^2 test and non linear fitting ofParis-Alger fly data

Introduction

χ^2tests are statistical tests that are used to estimate the correlation between variables. For example, when measuring endurance and biathlon performance of an athlete, we will find that both variables are closely linked, but correlating those measures with the number of books the person reads will not show any links. Two main variant of the test exists, one from Pearson, published in 1900, focuses on a test for goodness of fit. The second one, named Fisher's exact test is used for data set with small number of points.

Modules importation

Here are defined the various modules used in this part of the practical.

```
import statistics
from scipy.optimize import curve_fit
from scipy.stats import norm, chi2
from matplotlib import pyplot as plt
import numpy as np
```

χ2 test

Let's define the data vectors we will use to perform the test. We do so using python and the following program

```
T = [ 1.925 + 0.05*i for i in range(13)]
N = [19, 19, 39, 48, 87, 94, 104, 92, 57, 44, 28, 26, 13]

data = []
for i in range(len(N)):
   for j in range(N[i]):
       data.append(T[i])

mean_data = sum(data)/len(data)
print(f'mean of data ={mean_data}')
stdev_data = statistics.stdev(data)
print(f'stdev of data ={stdev_data}')
```

Running those lines allows us to print the mean and standard deviation of the data we will be using. Our vector has a mean of 2.216 and a standard deviation of 0.136. We are also

interested in plotting the cumulative distribution and the histogram of our data and compare them to a distribution and cumulative distribution of equivalent mean and standard deviation (i.e. with the values computed above.). We can do that using the following code.

```
# create N_sum
N_sum = np.cumsum(N)

# histogram
fig, axes = plt.subplots(1,2)
axes[0].bar(T,N, width=0.05, align='center', label="data")
axes[0].set_title("Number of light per flight-time")
axes[0].set_xlabel("Flight time")
axes[0].set_ylabel("Number of flights")

# cumulative distribution
axes[1].plot(T,N_sum, label='data') # just a curve
axes[1].set_title("Cumulative number of flight per flight-time")
axes[1].set_xlabel("Flight time")
axes[1].set_ylabel("Total number of flights")

# comparison to normal law
#plot that on the corresponding graphs
axes[0].bar(T, n_pdf, width=0.05, align='center', label="normal law",
alpha=0.6)
axes[1].plot(T, cdf*max(N_sum), label="normal law")

plt.tight_layout()
axes[1].legend()
plt.show()
```

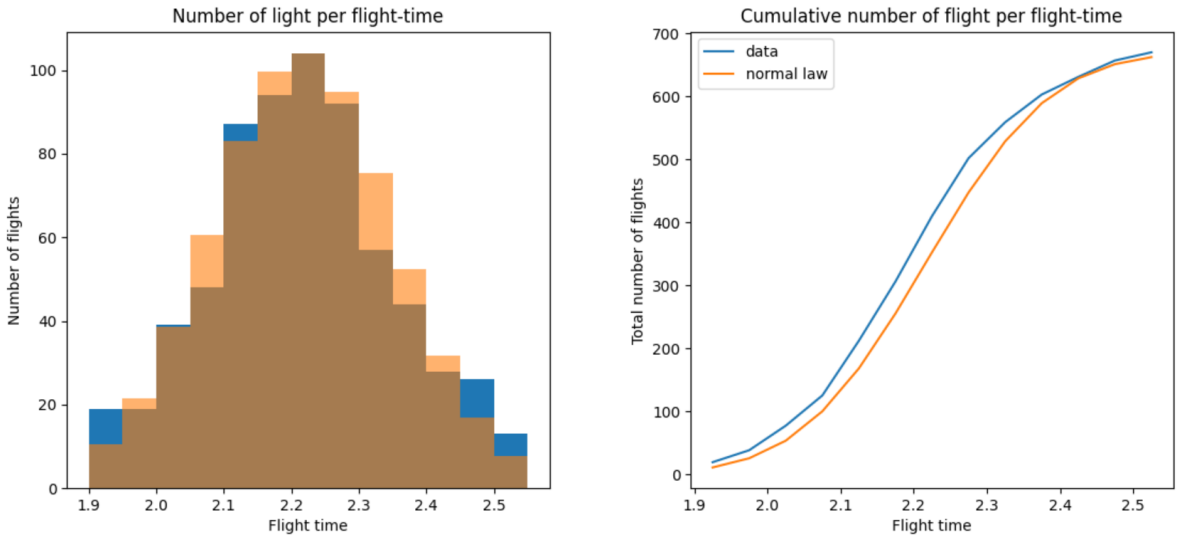


Fig T2.1 : Comparison of experimental data to a normal distribution.

We can suppose from those graphs that the distribution follows a gaussian curve.

Let's compute the D^2 of the data using python.

```
D = np.sum(([(a - b)**2/b for a, b in zip(N, n_pdf)]))
print("D^2:", D)
```

Console output :
```
D^2: 25.470562195114802
```

Finally, let's perform the chi test. We compute the degree of freedom of our data to be of 12, and we consider a confidence interval of 95%. We perform the test using the following code.

```
dof = 12 # degrees of freedom
p_value = 1-chi2.cdf(D, dof)

print("P-Value:", p_value)

if p_value<0.05: # 0.05 is the condition for the 95% interval, if under
that, we can reject the hypothesis
   print("p value is under 0.05, we can reject the hypothesis that it
is a gaussian.")
else:
   print("p value is above 0.05, we can't reject the hypothesis that it
is a gaussian.")
```

Console output:

```
P-Value: 0.012743855648706726
p value is under 0.05, we can reject the hypothesis that it is a gaussian.
```

The hypothesis of the flight distribution following a normal distribution is rejected by the chi test. We however wish to test the hypothesis further and verify the accuracy of a chi test.

Non linear fit of data

We would like to fit our data to a gaussian curve. To do so we will use the curve_fit function of the optimize scipy sub-module. We first define a gaussian function and pass it to curve_fit alongside its parameters to be fitted with. This is performed using the following code.

```
def gaussian(x, A, mu, sigma):
    return A * np.exp(-(x - mu)**2 / (2 * sigma**2))

initial_guess = [max(n_pdf), mean_data, stdev_data]
params, covariance = curve_fit(gaussian, T, N, p0=initial_guess)
A, mu, sigma = params
```

We wish to graph our fitted function to the data graphs. We can do that using the following program.

```
nbpts = 300
a = 1.8
b = 2.6
x = [a + i * (b - a) / (nbpts - 1) for i in range(nbpts)]

if True: # toggle
    plt.plot(T, N, label="given data", linestyle="", marker="x")
    plt.plot(x, [gaussian(i, initial_guess[0], initial_guess[1],
initial_guess[2]) for i in x], label="initial gaussian")
    plt.plot(x, [gaussian(i, A, mu, sigma) for i in x], label="fitted
curve")
    plt.text(1.8, 85, formatted_string, fontsize=9, color='black')
    plt.legend()
    plt.show()
```

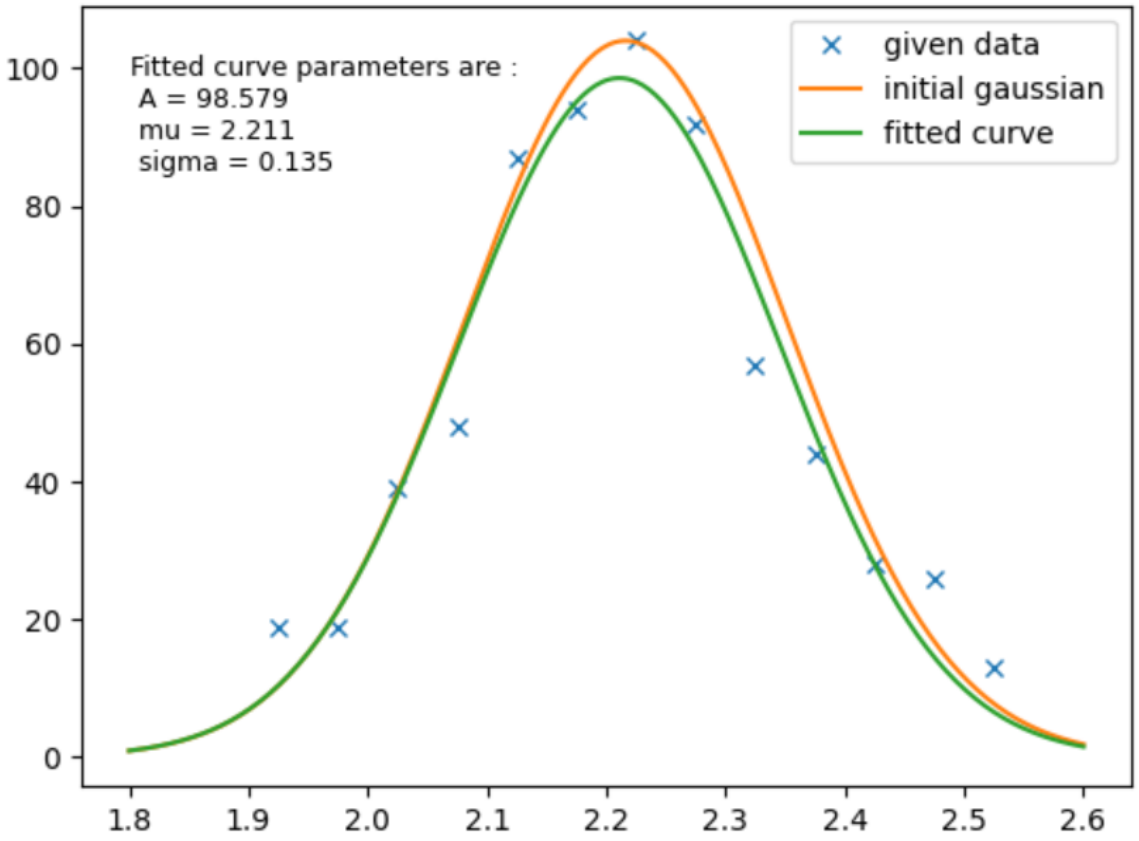


Fig T2.2 : Fitted curve plotted next to the initial estimated Gaussian and compared to data points.

Here we have the data points in blue cross, the initial gaussian in orange, and the final fitted gaussian in green. The points seems to fit the gaussian pretty well, we wish to compute the uncertainty using the covariance matrix returned by the fitting function.

```
# we use the convariance outputted in the previous question
print(f'Covariance matrix of the fit \n{covariance}')
# Get the standard errors (uncertainties) for A, mu, and sigma
std_err_A, std_err_mu, std_err_sigma = np.sqrt(np.diag(covariance))
print(f'Uncertainties :\n U_A : {2*std_err_A}\n U_mu : {2*std_err_mu}\n
U_sigma : {2*std_err_sigma}')
```

```
Console output:
Covariance matrix of the fit
[[ 1.99850388e+01  1.36007645e-04 -1.88215862e-02]
 [ 1.36007645e-04  4.96463135e-05 -4.54543769e-07]
 [-1.88215862e-02 -4.54543769e-07  5.19082329e-05]]
Uncertainties :
 U_A : 8.940925852089707
 U_mu : 0.014092028023985988
 U_sigma : 0.014409473671551385
```

Our program finally outputs the uncertainties which are pretty low when compared to the order of magnitude of their associated variable, being always under 10%. A fit to a gaussian curve is considered to be quite good for this case, thus invalidating the chi test.

Conclusion

χ2 tests can be extremely powerful, however they should be used with caution. As demonstrated in this example, the tests revoke an hypothesis that when tested with a fit, appears to work well. Upon doing some research, it appears that χ2 tests are effective for larger sets of data. Considering the low amount of points that we used in this example, we can conclude that it is the reason for its failure. However this demonstrates the necessity of knowing the tools we use when performing any kind of tests. Cross verification using multiple methods also allows to mitigate errors and weaknesses of other techniques.

Topic 3: Application of PCA to the analysis of sports results

Introduction

Principal component analysis is a versatile technique with widespread applications in different domains, providing a powerful tool for analyzing and extracting meaningful information from high-dimensional datasets. It works by identifying and emphasizing the dominant patterns within the data, allowing for a more efficient representation of complex relationships. Through the reduction of dimensionality, PCA aids in mitigating issues like multicollinearity, enhancing computational efficiency, and facilitating clearer interpretation of results. Its utility spans various fields, including finance for portfolio optimization, biomedicine for gene expression analysis, and remote sensing for image feature extraction. In marketing, PCA assists in customer segmentation by revealing underlying patterns in behavior. Overall, PCA stands as a fundamental technique for exploratory data analysis and preprocessing, empowering researchers, analysts, and machine learning practitioners to uncover valuable insights in diverse datasets.

Data analysis

Module importation, helper functions

We import necessary libraries for data exploitations and computations. We will be using numpy and matplotlib.

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

The following 2 functions are used to print the matrix in the console or using the matplotlib library and built-in displays.

```
def printm(matrix):
    print(f'{len(matrix)}x{len(matrix[0])}')
    for i in matrix:
        print(i)

def showm(matrix_data):
    # Display the matrix using matshow
    plt.matshow(matrix_data, cmap='viridis')  # 'viridis' is a colormap,
you can choose others
    plt.colorbar()  # Display a color bar showing the scale
    plt.title('Matrix Visualization')  # Title for the plot
    plt.show()
```

Loading data

Our data is stored in a .CSV file called data.csv. We set the path to that file and load it in a python array using a custom made file parser.

```
file_path = 'data.csv'  # Replace 'your_file.csv' with the path to your
CSV file

# Initialize an empty list to hold the data
matrix = []
# Open the CSV file and read its contents
with open(file_path, 'r') as file:

   data = file.read()
   l = data.split("\n")
   l.pop(0)
   for line in l:
       matrix.append([int(i) for i in line.split(";")[2:-1]])
```

Useful matrices and values

We will now compute basic values and matrices that we will need later

```
M = np.array(matrix)
M_bar = np.tile(np.mean(M, axis=0), (19,1))
Mc = M - M_bar

C = np.cov(M, rowvar=False)

# eigenvlaues, eigenvectors
Dl, V = np.linalg.eigh(C)

D = np.zeros([len(Dl),len(Dl)])
for i in range(len(Dl)):
   for j in range(len(Dl)):
       if i ==j:
           D[i,j] = Dl[i]

R = np.corrcoef(M, rowvar=False)
Y = np.matmul(Mc, V)

# isolates the 3 most prominent values
Z = Y[:, -3:]
psi = np.transpose(V)[-3:, :]
```

Initial and truncated matrices

```
#  Question 1
sub = np.matmul(Z, psi)
print(f'sub {len(sub)}x{len(sub[0])}')

truncated_matrix =  M_bar + sub

print(f'M_trunc {len(truncated_matrix)}x{len(truncated_matrix[0])}')


#  Compare those 2 here
plt.matshow(truncated_matrix, cmap='viridis', label="M_trunc")
plt.matshow(M, cmap='viridis', label="M")
plt.show()
```

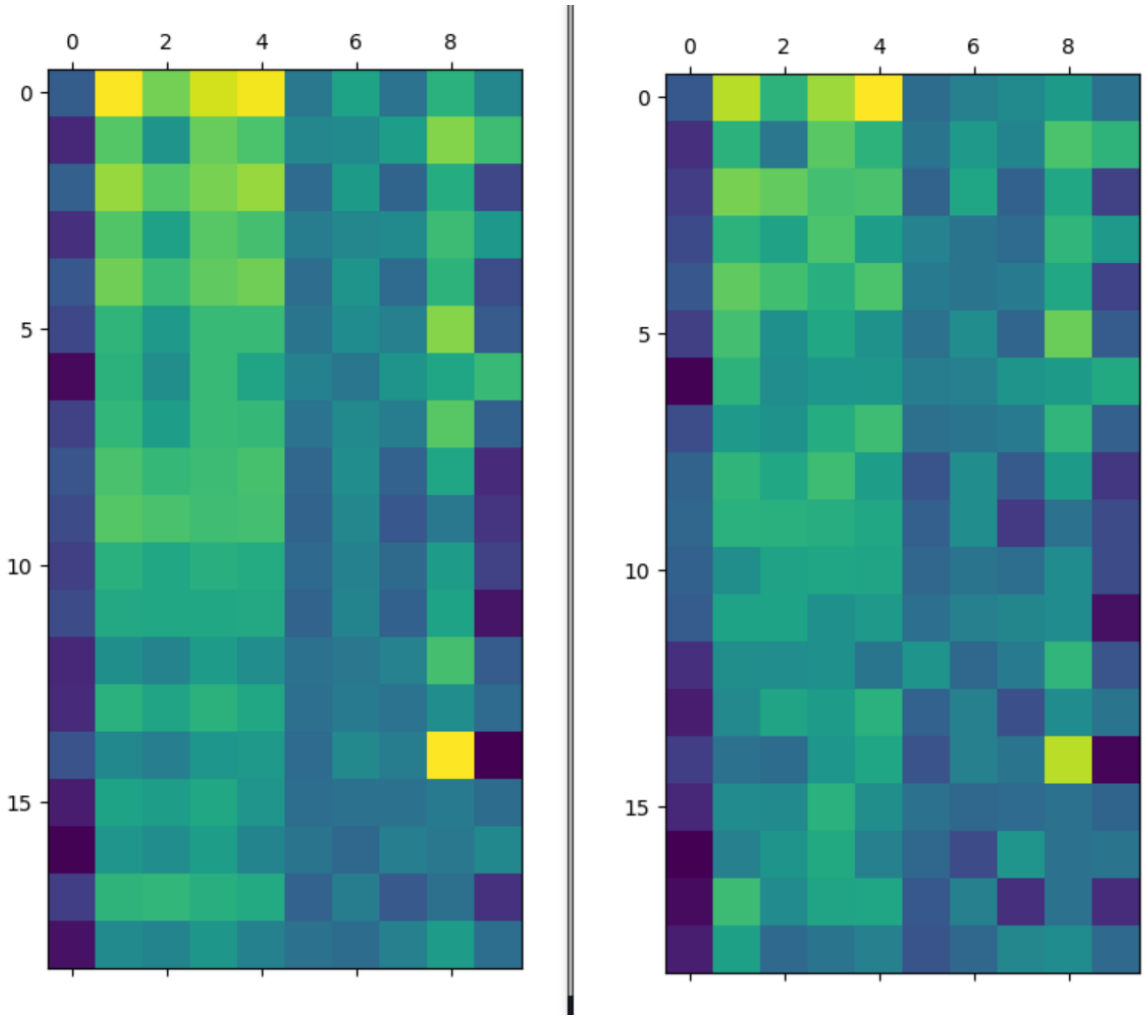


Fig T3.1 : In order, truncated matrix and original data matrix

When truncating the matrix using the 3 main components, we can spot a difference in the intensity of spots between each matrix. It can be mostly seen in the top band and last spot. The truncated matrix looks more homogenous than the original one, which comes from the loss of information when taking the 3 most prominent components.

Loss factor

The loss factor can be computed by summing the 3 components taken and dividing it by all the original components.

```
loss_factor = sum(Dl[:7])/sum(Dl)
print(f'Loss factor is of about {loss_factor}')
```

Console output:
Loss factor is of about 0.25371654407507127

Using the above code we can conclude that using the 3 main components we lose about 25% of the information when performing this reduction.

Plotting of correlation matrix

We would like to plot the correlation matrix in 3D. To do so we use the matplotlib library and its pyplot sub module in order to generate a 3D view using the coefficients as heights. The following code block is used to generate the plot.

```
#3D plot
import matplotlib.pyplot as plt
import numpy as np

columns = ["1500 m", "100 m", "400 m", "110 m hurdles", "Long jump",
"Shot put", "High jump", "Discus throw", "Pole vault", "Javelin throw"]
X=np.array([i for i in range(len(R))])
Y=np.array([j for j in range(len(R[0]))])
Z=[]
X, Y = np.meshgrid(X, Y)

for i in range(len(R)):
    sub = []
    for j in range(len(R[0])):
        sub.append(R[i][j])
    Z.append(sub)
Z = np.array(Z)

fig = plt.figure()
```

```
ax = fig.add_subplot(projection = '3d')
surf = ax.plot_surface(X, Y, Z, rstride=1, cstride=1, cmap='viridis',
linewidth=0, antialiased=False)
# ax.set_zlim(0, 1.01)
plt.xticks(range(len(columns)), columns, rotation = 0)
plt.yticks(range(len(columns)), columns, rotation = 0)
plt.show()
```

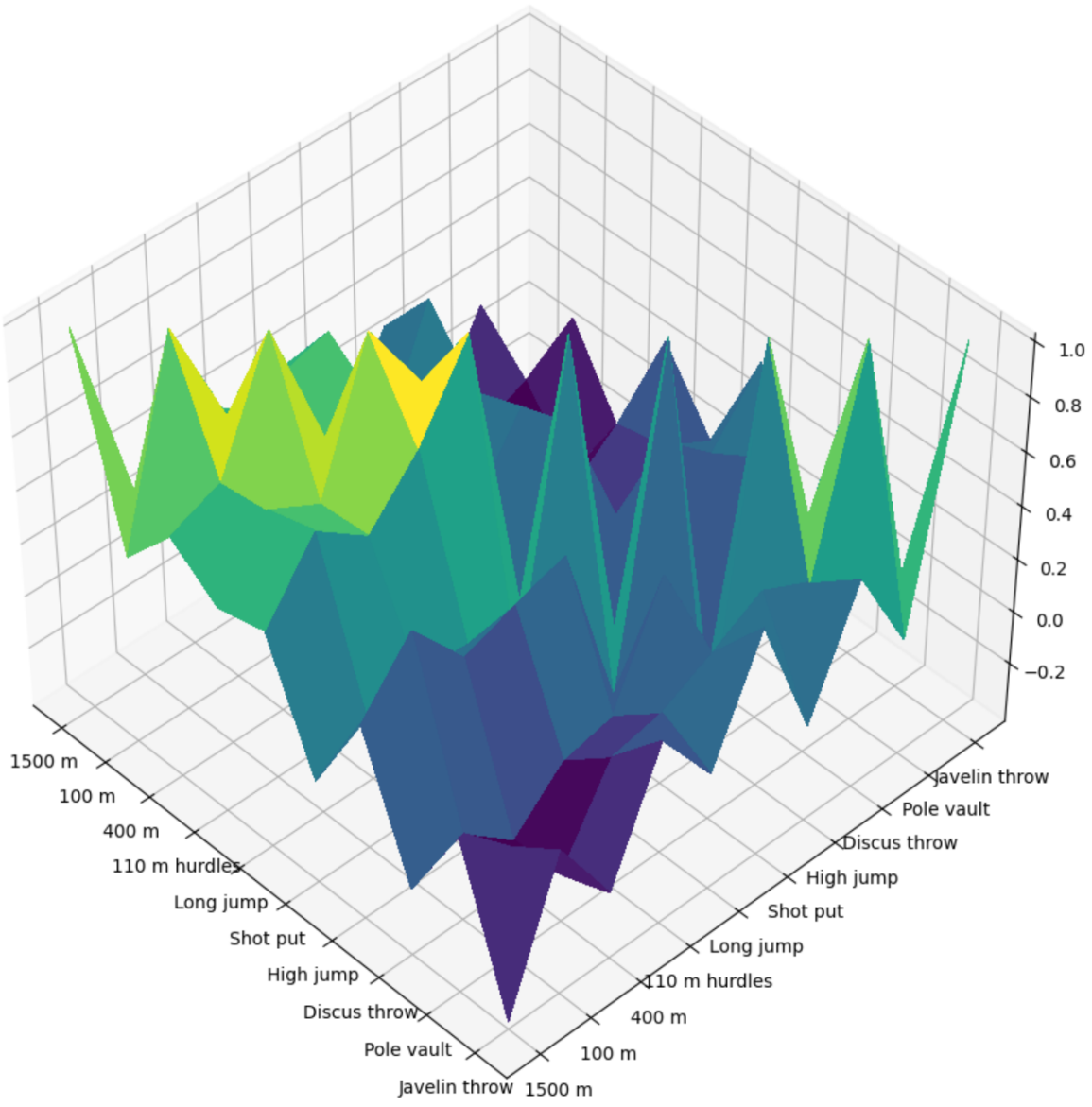


Fig T3.2 : 3D representation of the correlation matrix

Unfortunately this representation looks to be unfit for the interpretations we would like to make of it. Thus we will use a traditional 2D plot using a grayscale heatmap.

```
#2D plot
```

```
plt.matshow(R, cmap='viridis')  # 'viridis' is a colormap, you can
choose others
plt.colorbar()  # Display a color bar showing the scale
plt.title('Correlation matrix')  # Title for the plot
plt.xticks(range(len(columns)), columns, rotation = 90)
plt.yticks(range(len(columns)), columns, rotation = 0)
plt.show()
```

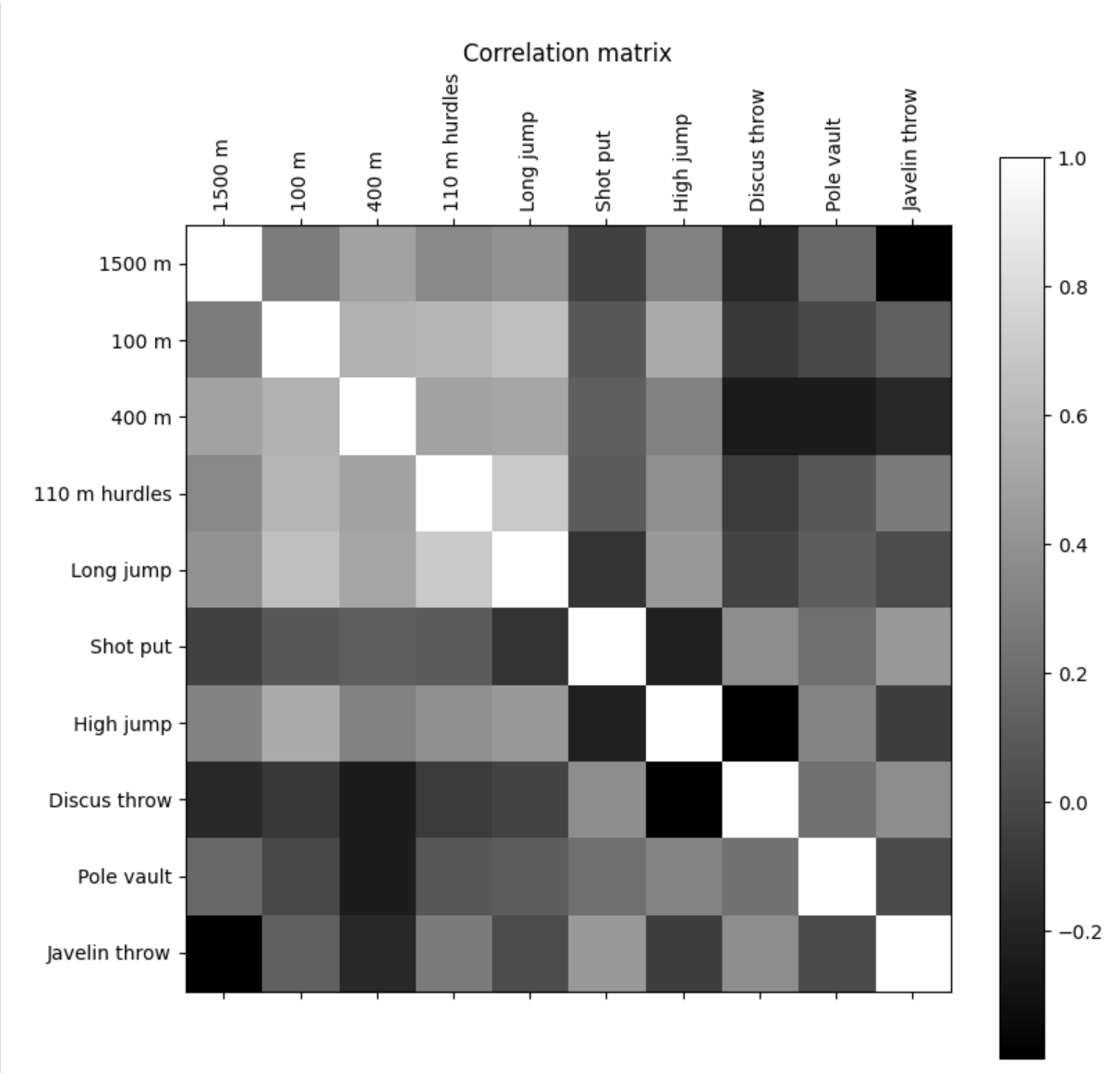


Fig T3.3 : 2D Inverse grayscale representation of the correlation matrix.

Using the grayscale heatmap scale, highly correlated variables will appear whiter. One can easily spot that the top corner of the map is way lighter. While Pole vault, Javelin throw, Discus throw and Shot put do not seem to be very correlated, High jump, long jump and running disciplines all seem to have a high correlation between each other. For a further analysis, this data could be passed to health and sports specialists. Based on our limited knowledge of anatomy, we speculate that this is due to the muscle groups used being different for non correlated sports.

3D Plotting of NxN rotation matrix and Karhunen-Loève matrix

We wish to plot a 3D representation of the rotation matrix using the 3 main components used for the previous analysis. We perform such a plot using the following code.

```
sub = V.T[7:10, :]
fig = plt.figure()
ax = fig.add_subplot(projection = '3d')
for i in range(len(columns)):
    x = sub[0, i]
    y = sub[1, i]
    z = sub[2, i]
    ax.plot([0, x], [0, y], [0, z], label=columns[i])

plt.legend()
plt.show()
```

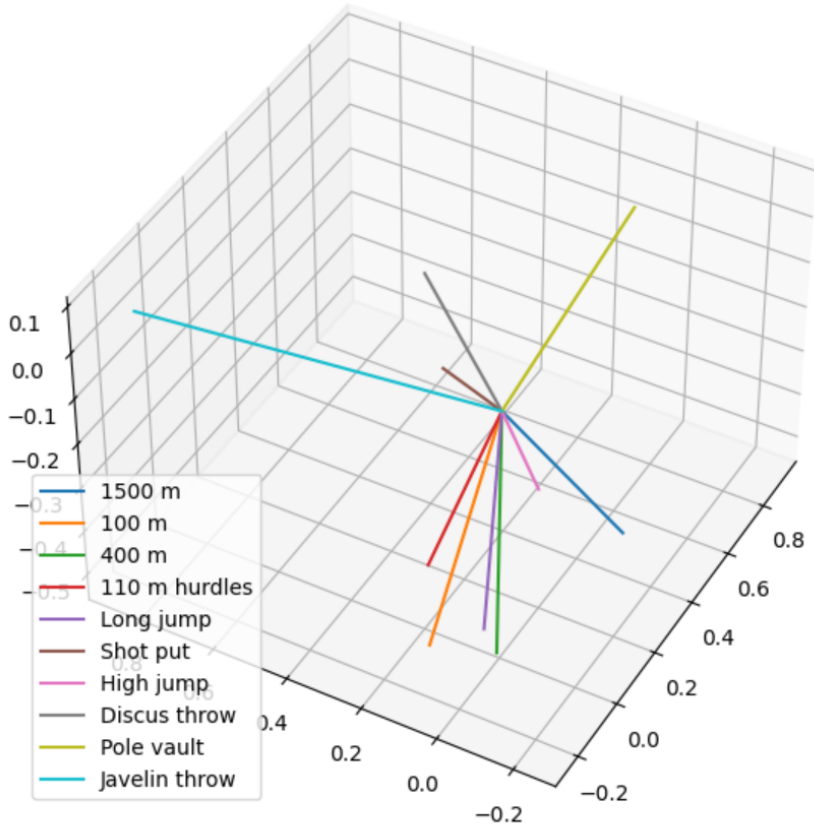


Fig T3.4 : 3D plot of the rotation matrix using the main 3 principal components and sports

Lines in the same direction and close to each other indicate a strong dependency on the same components. We can use the above 3D graph to conclude that long jump, 100m and 110m hurdles are quite related. Following the same reasoning as before, we can extrapolate that one of our principal components is “running”. The same way we can say that the other two are “jumping” and “throwing”.

We can also spot that some values end up in the negative. In this case, it would indicate that a certain component has a negative impact on the scores in the category. In our study, however, it seems that values are reversed. Thus vector positive on a certain component implies a negative contribution.

Let’s apply the same analysis to plot the Karhunen-Loève matrix. We use the same technique than for the previous matrix.

```
lines = np.array(["Warner","Mayer","Moloney","Scantling","Lepage",
"Ziemek","Victor","Shkurenyov","Urena","Bastien","Erm","Wiesiolek","Zhu
k","Kazmirek","Uibo","Helcelet","Sykora","Dos Santos","Roe"])

sub = Y[:, 7:10]
fig = plt.figure()
ax = fig.add_subplot(projection = '3d')
for i in range(len(lines)):
   x = sub[i, 0]
   y = sub[i, 1]
   z = sub[i, 2]
   ax.plot([0, x], [0, y], [0, z], label=lines[i])

plt.legend()
plt.show()
```

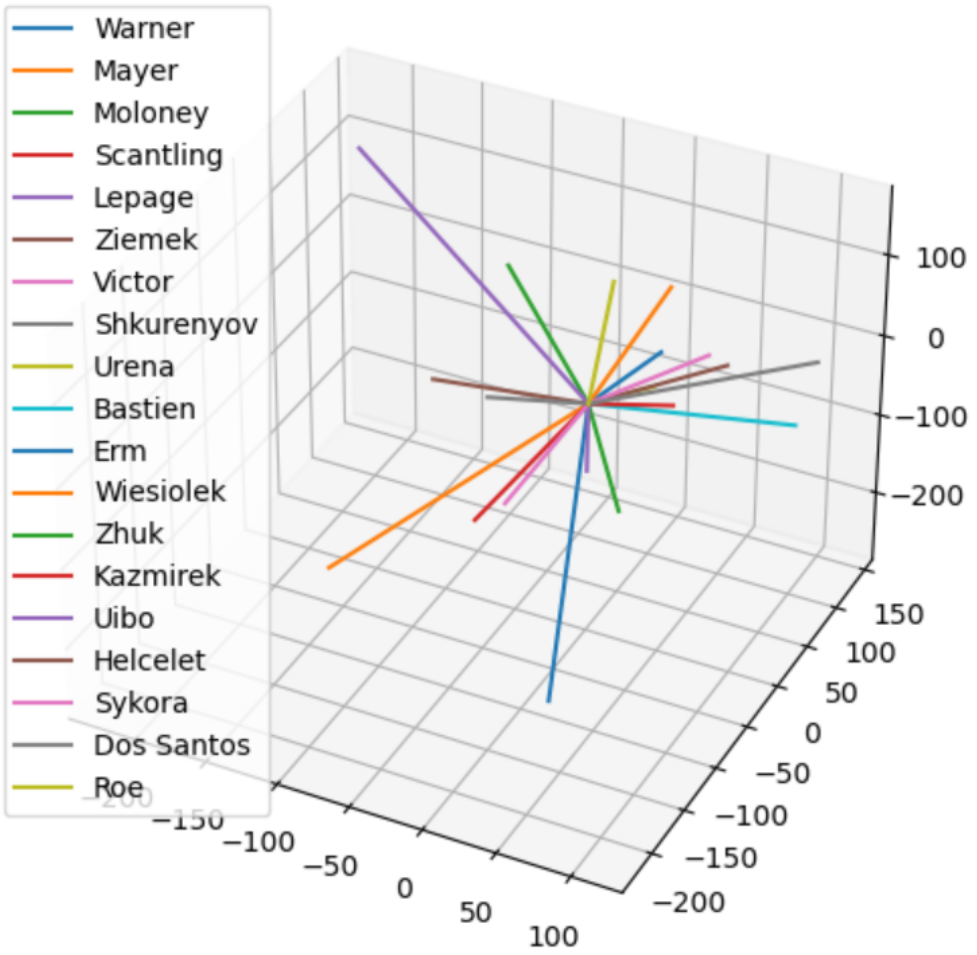


Fig T3.5 : 3D plot of the Karhunen-Loève matrix using the athletes

In order to compare the results of the graphs to the table we will check 2 athletes with close vectors and compare their results. Let's take Ziemek and Shkurenyov, who's vectors are almost collinear. We find that their results are always quite similar with respect to each discipline. Let's now take a look at Sykora and Uibo, who's vectors are almost opposite. We can see that they perform well, but in different disciplines.

Following some personal research, it appears that the length of a vector corresponds to the amount of variance of one athlete. This would mean that athletes with smaller vectors are more consistent across the board. Let's take 2 examples and see how they perform. Let's first take a look at Erm, which vector is one of the smallest. His scores range from 714 to 905 and are mostly spread around 800. Now let's check Uibo's result. His scores range from 598 to 1067. The values are also spread a lot in this interval. Those examples seem to support the theory. Athletes with a long vector are glass cannons, they will perform very well in some discipline but heavily underperform in others.

Following the analysis done above, we can say that the values found are to be adequate with the score table given as data.