Project

Abstract

This project showcases the importance and wide ranging capabilities of Python, in the fields of analysis, statistics and machine learning. The code snippets explore methodologies starting from concepts like hypothesis testing, probability calculations and simulations involving discrete distributions. By utilizing libraries such as NumPy, SciPy, Matplotlib and Seaborn these examples offer insights into analyses that can be comprehensible, to a diverse audience.

This research takes a dive into probability theory by employing techniques to address real world challenges. The introduction sets the stage by outlining the problem statement and emphasizing the significance of exploring probability theory. Subsequent chapters follow a structured narrative approach.

Chapter 1; Introduction. This chapter introduces the projects scope. Emphasizes the importance of probability theory, in addressing real world issues.

Chapter 2; Description of Data. Here we delve into the significance of data collection providing insights into data sources while addressing concerns such as outliers and missing values. We also offer solutions to overcome these challenges.

Chapter 3; Methodology. In this chapter we explore methods ranging from concepts to advanced topics like Markov Chains and factor analysis. It serves as a foundation for chapters.

Chapter 4; Results. The core of this project lies in analyzing both real world data. We employ techniques such as exploring variables conducting statistical analyses visualizing data and analyzing real world datasets.

Chapter 5; Conclusion. This final chapter summarizes our findings in a way that's easily understandable for technical readers. We highlight the implications and interpretations derived from our modeling and analysis processes. Additionally we present a comparison of models along with conclusions to audiences without a statistical background.

This collection strikes a balance between understanding and practical application making it a comprehensive guide for individuals navigating the field of data science. Whether you're interested in grasping concepts simulating systems or implementing machine learning models the provided code snippets offer a encompassing approach, to making informed data driven decisions.

This collection provides a foundation, for exploring statistical and machine learning principles while also helping readers grasp the capabilities of Python in the field of data science. Each piece of code is explained thoroughly making this resource extremely valuable for individuals seeking to expand their understanding and harness Pythons potential, for data analysis.

1 CHAPTER 1: INTRODUCTION

creating the world of probability is, like having a lot of things that helps us discover the miracles of chance and predicting real life outcomes. This project gives scope into probability theory aiming to understand how it applies in scenarios

Our exploration involves observing different methods and tools used for study of data examination and various events Through study of imitation of data we choose to demonstrate the occurrence of probabilities that govern our lives especially when dealing with variables. Additionally we understand's into Markov Chains to observe how one event will direct to other event.

For getting more accuracy we try to learn different techniques that decrease prediction errors and increase the correctness and dependent of our assumption This project isn't just abstract, it has practical applications. Understanding the concept of probability offers us a perspective to analyse uncertainties that impact aspects of life.

This project gives idea on how probability's applied in our day to day lives and enhancing the importance of its applications in real life.

2 Chapter 2: Data Description

The Iris dataset was used in R.A. Fisher's classic 1936 paper, The Use of Multiple Measurements in Taxonomic Problems, and can also be found on the UCI Machine Learning Repository.

It includes three iris species with 50 samples each as well as some properties about each flower. One flower species is linearly separable from the other two, but the other two are not linearly separable from each other.

The columns in this dataset are:

Id SepalLengthCm SepalWidthCm PetalLengthCm PetalWidthCm Species

3 Chapter 3: Methodologies

1. Mean:

- Definition: The mean represents the average value within a dataset, calculated by summing all values and dividing by the total number of values.

1. Variance:

- Definition: Variance measures the extent to which values in a dataset deviate from the mean. It quantifies the overall spread or dispersion of data points around the mean.

1. Standard Deviation:

- Definition: Standard deviation, the square root of the variance, provides insight into how much individual data points deviate from the mean, offering a measure of overall data dispersion.

1. Mode:

- Definition: The mode identifies the most frequently occurring value in a dataset, representing the number with the highest frequency.

1. First and Third Quantile:

- Definition: Quantiles divide a dataset into equal parts. The first quantile (25th percentile) denotes the value below which 25% of the data falls, while the third quantile (75th percentile) indicates the value below which 75% of the data falls.

1. Joint Distributions:

- Definition: Joint distributions deal with the probabilities associated with multiple random variables occurring simultaneously. They describe the likelihood of specific combinations of outcomes from different variables.

1. Conditional Expectations:

- Definition: Conditional expectations refer to the expected value of a random variable given certain information or conditions.

1. Bayes' Rule:

- Definition: Bayes' theorem is a mathematical observation used to change the probability of a predictions as more likely or information becomes useful.

1. Discrete and Continuous Random Variables:

- Definition: Random variables represent outcomes of a random phenomenon. Discrete random variables take on distinct values, while continuous random variables can assume any value within a range.

1. Order Statistics:

- Definition: Order statistics analyze the positions of values in a dataset when arranged in ascending or descending order.

1. Correlation:

- Definition: Correlation measures the relationship between two variables, indicating how changes in one variable are associated with changes in another.

1. Markov Chains:

- Definition: Markov Chains model sequences of events where the probability of each event depends only on the state attained in the previous event.

1. Simulation Techniques:

- Definition: Simulation techniques replicate real-world processes by generating artificial data, imitating the behavior of complex systems.

1. Factor Analysis:

- Definition: Factor analysis is a statistical study of recognising different bonds among variables by differentiating them into more useful, meaningful variantw

.

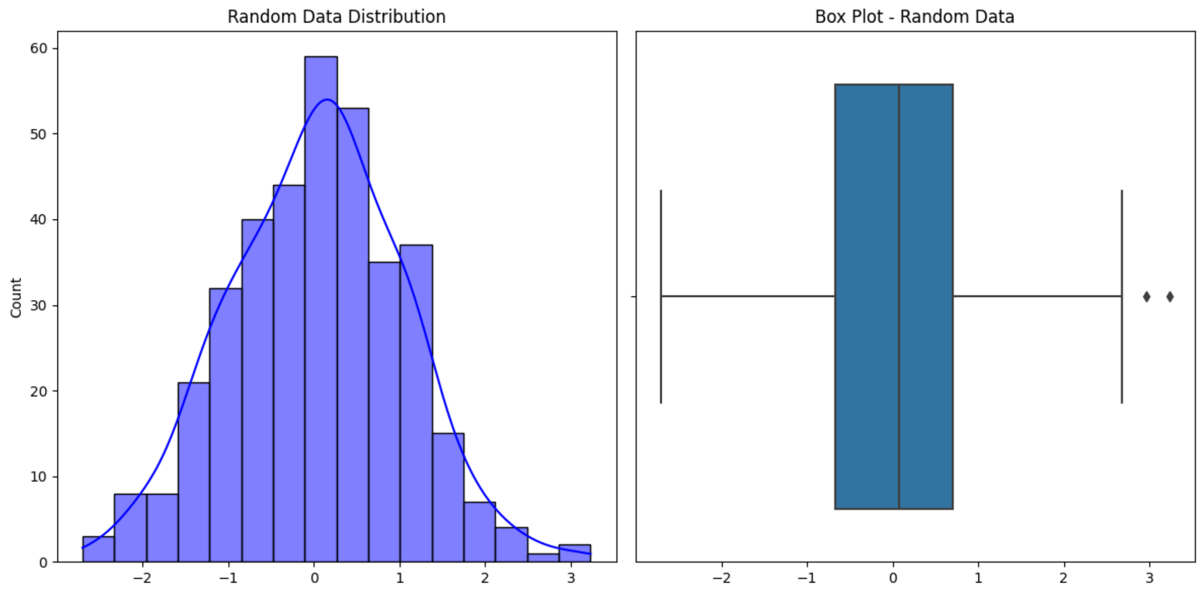
4 4 Chapter 4: Analysis and Results

4.1 SIMULATION DATA ANALYSIS

4.1.1 Simulating Continuous Random Variables:

In [1]:

```
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from scipy.stats import skew, kurtosis, norm
import pandas as pd


def generate_random_data(size=369):
    return np.random.randn(size)
def statistical_analysis_of(data):
    mean_of_data = np.mean(data)
    variance_of_data = np.var(data)
    std_dev_of_data = np.std(data)
    quantiles_of_data = np.percentile(data, [25, 75])
    modes_of_data = pd.Series(data).mode()
    if len(modes_of_data) > 0:
        mode_of_data = modes_of_data.tolist()
    else:
        mode_of_data = None

    order_of_data = np.sort(data)
    skewness_of_data = skew(data)
    kurt_of_data = kurtosis(data)

    return mean_of_data, variance_of_data, std_dev_of_data,
quantiles_of_data, mode_of_data, order_of_data, skewness_of_data,
kurt_of_data

random_data = generate_random_data()
mean_of_data, variance_of_data, std_dev_of_data, quantiles_of_data,
mode_of_data, order_of_data, skewness_of_data, kurt_of_data =
statistical_analysis_of(random_data)
plt.figure(figsize=(12, 6))
plt.subplot(1, 2, 1)
sns.histplot(random_data, kde=True, color='blue')
plt.title(f'Random Data Distribution')
plt.subplot(1, 2, 2)
sns.boxplot(x=random_data)
plt.title(f'Box Plot - Random Data')
plt.tight_layout()
plt.show()


print("\nStatistical Analysis for Random Data:\n")
print(f"Mean: {mean_of_data}\nVariance: {variance_of_data}\nStandard
Deviation: {std_dev_of_data}")
print(f"Quantiles (25th, 75th): {quantiles_of_data}\nMode:
{mode_of_data}\nSkewness: {skewness_of_data}\nKurtosis: {kurt_of_data}\n")
```



```
Statistical Analysis for Random Data:

Mean: 0.04053954213922376
Variance: 1.0038998698575017
Standard Deviation: 1.0019480375036929
Quantiles (25th, 75th): [-0.66634342  0.69571183]
Mode: [-2.698833528413543, -2.532381477327955, -2.406658042765403, -2.11421
2839555721, -2.0760222583596515, -2.0491920854070322, -2.0407952877350524,
-2.0358674633695553, -2.0271303233391107, -2.008991455052839, -1.9678197735
735679, -1.9113766151172973, -1.8040207508219674, -1.7778947031681742, -1.6
985649672712109, -1.6541629944930905, -1.6316908646600128, -1.6178700690123
105, -1.594806530829683, -1.544959547394251, -1.5415200153078237, -1.499839
0230600553, -1.4843330158321333, -1.4685513761227977, -1.4521195893099963,
-1.4488092461146493, -1.4297326434815147, -1.3847765650703137, -1.382379834
142602, -1.37540847779904, -1.3734636634440904, -1.3450650520225444, -1.320
2562719244921, -1.3081841189551036, -1.302769482494536, -1.2873526895370353
, -1.2859187879901368, -1.2661924602863204, -1.232928252883942, -1.21826489
1150138, -1.2150304320751892, -1.198693686119556, -1.1968687665872353, -1.1
88220248808917, -1.1856889887051896, -1.1819995059792165, -1.18083688872590
5, -1.1641232279710132, -1.1449194402617657, -1.1336179099445083, -1.123258
9575278582, -1.1150784848549096, -1.078337628970492, -1.0746051611361709, -
1.0687523135411252, -1.06108525684342, -1.055243245507662, -1.0502210695209
107, -1.0347389735389363, -1.034496709326832, -0.9925318459342426, -0.98598
13170991223, -0.9801720866690552, -0.9788637833515257, -0.9559707412985675,
-0.9274805266112413, -0.9130342410301847, -0.9104007675297581, -0.892572709
5553464, -0.8877206028958439, -0.8731440603703332, -0.8653681427959212, -0.
8327551855255912, -0.8187880202874805, -0.801865664352596, -0.7990305237707
083, -0.7969677575295497, -0.7941990399515818, -0.7882905612973905, -0.7661
210294050372, -0.7651209261458113, -0.7539668621653033, -0.7518767661664076
```

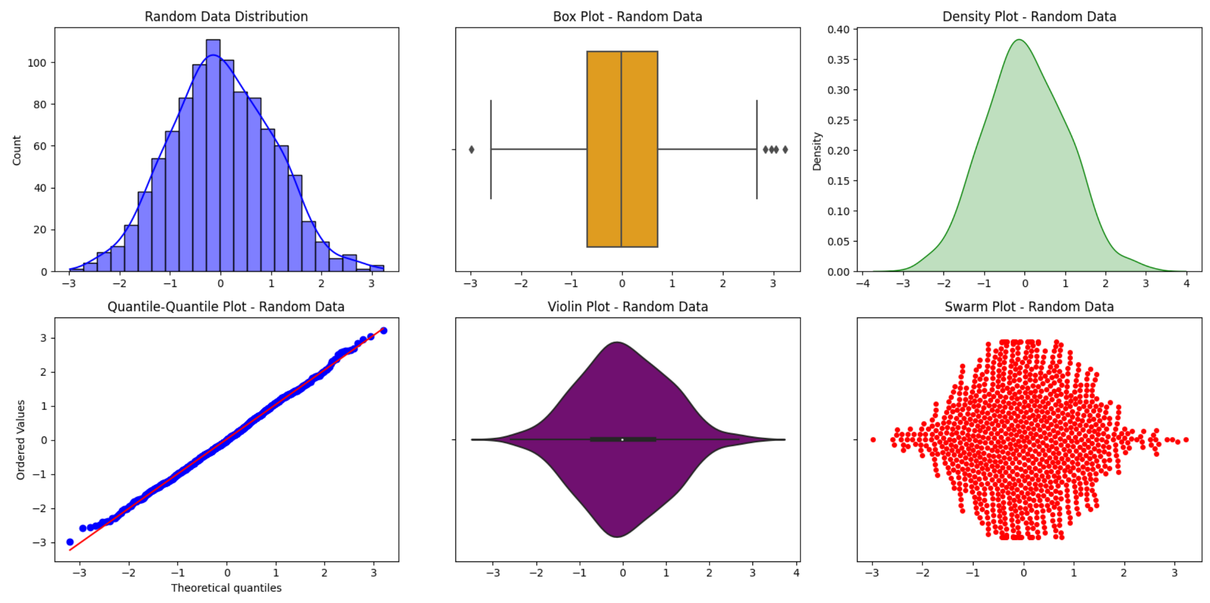
```
, -0.7415452662098166, -0.7365509014260951, -0.7271257889823058, -0.7239539
810347289, -0.7215019389623003, -0.7120631961938029, -0.6998672678820382, -
0.6786088402077701, -0.6731281024504779, -0.6663434194265893, -0.6641526400
852626, -0.6485367328309571, -0.6474642843309213, -0.6212515466872293, -0.6
21004273411703, -0.6204641346128704, -0.6104602223509257, -0.60827754900158
85, -0.594647580361068, -0.5803744919873978, -0.5721893688098062, -0.568707
4327106475, -0.5660460128201986, -0.5535965679417039, -0.5362669445634308,
-0.5283095310308831, -0.49803724720739134, -0.48972665642765995, -0.4828620
58238707, -0.4533113321382071, -0.41652411077971174, -0.4159412127925044, -
0.4154883092837749, -0.41337601225212295, -0.4126952494976641, -0.408111221
6280896, -0.39624831013382467, -0.3892318300523491, -0.36866489565315275, -
0.36594767627195396, -0.35621892953854156, -0.3540215383633587, -0.32257788
491015843, -0.31867111689169525, -0.3182564319223945, -0.31686407857505205,
-0.31670600574710933, -0.3160957646080266, -0.3120245297765165, -0.31089306
336830697, -0.3091410892879513, -0.2865562018128053, -0.2811913259389396, -
0.26873806865161487, -0.2676812248888217, -0.2504220829715864, -0.243746727
64685482, -0.24236593237595955, -0.22271452590064647, -0.20707549879399129,
-0.2055304984970831, -0.19889401393227796, -0.19392851067558695, -0.1865665
1520185902, -0.17640048638577344, -0.16889380048029573, -0.1623581004645946
, -0.1533049023337975, -0.14698781995882834, -0.13305280696867391, -0.12812
713091372946, -0.1219015806057406, -0.11928515202868004, -0.094439937807611
33, -0.09438335492748921, -0.08832746648072731, -0.0829427757534794, -0.081
62173527319981, -0.08118111294541774, -0.06797056349215914, -0.067332056303
44315, -0.04031013706111058, -0.03427746919070514, -0.03384296371835289, -0
.026096742350654824, -0.01825782010589742, -0.011865466975898086, -0.011580
718371668339, -0.0038299161891745016, -0.0032822450381474566, 0.00019192899
8191444, 0.0057903047591899015, 0.014530290186699952, 0.018893070239127046,
0.020138379458409, 0.03351285573044242, 0.04273363406518915, 0.064978473478
59062, 0.06576287012829804, 0.06589990980688043, 0.06700974192880157, 0.069
17241554649624, 0.0911792845525216, 0.09205075230869553, 0.0950446816553377
3, 0.09913128795849423, 0.1141654373468869, 0.12456743839019001, 0.12835999
276712476, 0.1319632521406831, 0.14342912863790006, 0.15555884557038954, 0.
15869797726092838, 0.16116351420208974, 0.16925775679738628, 0.169436232800
4859, 0.17323251443316354, 0.17857943053156283, 0.18081098158659636, 0.1814
6385079157823, 0.18405048187469153, 0.18473043593818972, 0.1921125353499823
5, 0.19313898088513307, 0.19522536369655938, 0.20012721417025792, 0.2114349
7290349472, 0.2238683305757191, 0.23244697433171477, 0.2454276977456891, 0.
26167904116821733, 0.2642800803688951, 0.26725980299896707, 0.2787602067135
316, 0.2836742246379142, 0.28420083392918877, 0.2910513116810514, 0.3001175
069823777, 0.3102095878178732, 0.31163986312240677, 0.31492462089244305, 0.
315884047632201, 0.3213109965822632, 0.33075625582862617, 0.346879847131476
24, 0.3500075462517146, 0.3610184577633333, 0.3610604602012387, 0.370409777
65256583, 0.3766816583334812, 0.37973302164154427, 0.38272108719561676, 0.3
836408984005951, 0.38499206915537587, 0.39376348249446314, 0.39416507540675
82, 0.395129075253213, 0.4124536229911784, 0.41305703865239, 0.422720056678
32274, 0.423263435341529, 0.4303486411089779, 0.43622034880583815, 0.437403
4453570189, 0.4375261290845861, 0.44263100144540946, 0.4474022373451951, 0.
45741363722030043, 0.45917013419234515, 0.49512137224229924, 0.499358642838
75636, 0.5189161600870323, 0.5330383187671874, 0.547868453936898, 0.5550488
673933269, 0.590923081180954, 0.592590539324467, 0.5946530281036272, 0.5984
089837594698, 0.6061521033790449, 0.6080634492874976, 0.6127710421991674, 0
.627390393333238, 0.6299843370672571, 0.634525676333745, 0.642410262492201,
0.6528378070406349, 0.6568254889805826, 0.6636486679488297, 0.6725243835969
174, 0.6919764161350278, 0.6920128062175018, 0.6928037810734293, 0.69571182
60816821, 0.7008484415500619, 0.7140134370093186, 0.7598867042976661, 0.765
7683252658243, 0.7662616934509694, 0.7664180605812777, 0.7842331737988547,
0.7853932906544631, 0.7888053814694609, 0.8229896545063715, 0.8338201232303
```

```
003, 0.8549832478566999, 0.8689447447852137, 0.8744638946574957, 0.87791762
27603502, 0.8974532587743589, 0.8997294397127108, 0.9014188038168907, 0.906
7994872470557, 0.9129583730767584, 0.9153277093181799, 0.9573332103373765,
0.9650663321301169, 0.9743521475364523, 0.9918996707475435, 0.9993930656461
72, 1.0082432707049507, 1.0108295741950168, 1.015183336142897, 1.0413612365
395675, 1.049086642855211, 1.0550198641961643, 1.0620438117855302, 1.071108
1084445264, 1.0847666663911186, 1.1038750002254683, 1.1102311891133296, 1.1
280371025186724, 1.1282439122684504, 1.1404232492877482, 1.1422688244238555
, 1.142860587045969, 1.1517271362826844, 1.1585979050865065, 1.160689395905
2091, 1.174540585046515, 1.1754952291340037, 1.1826810475936391, 1.18278968
65243386, 1.1884371812065206, 1.1956280282460152, 1.19642501271549, 1.20812
29121172563, 1.2110034762032973, 1.2162755198352952, 1.2299813857641566, 1.
2475534728543483, 1.251758931836333, 1.2627498330445486, 1.3220926835983449
, 1.327408527303071, 1.345299056028543, 1.37578707289111, 1.385893502467126
4, 1.3914670248824041, 1.406029774174746, 1.4469432239836753, 1.46978597759
2191, 1.5912958440670288, 1.601732319290018, 1.6099050656464051, 1.61959947
15356903, 1.625125302642313, 1.6326841979067939, 1.6625613324252644, 1.6774
17133051082, 1.6903598516776384, 1.7009577879531008, 1.807249985047504, 1.8
205334309426218, 1.9262063811544252, 1.9571255030174546, 1.9732004649385977
, 1.982450588413474, 2.098010283182572, 2.2157994446262497, 2.2710706046594
047, 2.390477847618545, 2.402277084791933, 2.6745616477557426, 2.9582893108
979564, 3.2310762910211337]
Skewness: 0.03064865321266709
Kurtosis: -0.050137519364255034
```

The code generates c random data from a normal distribution. The subsequent statistical_study of computes mean, variance, quantiles, mode, skewness, and kurtosis. We visualized the data distribution using a histogram and box plot, providing insights into its central tendency, spread, shape, and outliers, aiding in understanding the dataset's characteristics and potential deviations from a normal distribution.

In [2]:

```python
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from scipy.stats import skew, kurtosis, norm, probplot
import pandas as pd
import warnings
warnings.filterwarnings('ignore')


def generate_random_data(size=1000):
    return np.random.randn(size)


def perform_statistical_analysis_for(data):
    mean_of_data = np.mean(data)
    variance_of_data = np.var(data)
    std_dev_of_data = np.std(data)
    quantiles_of_data = np.percentile(data, [25, 75])

    modes_of_data = pd.Series(data).mode()
    if len(modes_of_data) > 0:
        mode_of_data = modes_of_data.tolist()
    else:
        mode_of_data = None
    order_of_data = np.sort(data)
    skewness_of_data = skew(data)
    kurt_of_data = kurtosis(data)

    return mean_of_data, variance_of_data, std_dev_of_data,
quantiles_of_data, mode_of_data, order_of_data, skewness_of_data,
kurt_of_data

random_data = generate_random_data()

mean_of_data, variance_of_data, std_dev_of_data, quantiles_of_data,
mode_of_data, order_of_data, skewness_of_data, kurt_of_data =
perform_statistical_analysis_for(random_data)

plt.figure(figsize=(16, 8))
plt.subplot(2, 3, 1)
sns.histplot(random_data, kde=True, color='blue')
plt.title(f'Random Data Distribution')


plt.subplot(2, 3, 2)
sns.boxplot(x=random_data, color='orange')
plt.title(f'Box Plot - Random Data')


plt.subplot(2, 3, 3)
sns.kdeplot(random_data, color='green', fill=True)
plt.title(f'Density Plot - Random Data')


plt.subplot(2, 3, 4)
probplot(random_data, dist="norm", plot=plt)
plt.title(f'Quantile-Quantile Plot - Random Data')


plt.subplot(2, 3, 5)
sns.violinplot(x=random_data, color='purple')
plt.title(f'Violin Plot - Random Data')


plt.subplot(2, 3, 6)
sns.swarmplot(x=random_data, color='red')
plt.title(f'Swarm Plot - Random Data')

plt.tight_layout()
plt.show()


print("\nStatistical Analysis for Random Data:\n")
print(f"Mean: {mean_of_data}\nVariance: {variance_of_data}\nStandard
Deviation: {std_dev_of_data}")
print(f"Quantiles (25th, 75th): {quantiles_of_data}\nMode:
{mode_of_data}\nSkewness: {skewness_of_data}\nKurtosis: {kurt_of_data}\n")
```



```
Statistical Analysis for Random Data:

Mean: 0.019750657602453116
Variance: 1.030741593691681
Standard Deviation: 1.015254447757645
Quantiles (25th, 75th): [-0.69075122  0.71335344]
Mode: [-2.9827452087600363, -2.589362300862882, -2.562902969620122, -2.5227
833928670096, -2.5026667224684305, -2.4187608115285464, -2.4053145318895055
, -2.3865310903700947, -2.3779172805173845, -2.3068959471924937, -2.2981274
98280471, -2.287656139947985, -2.2503020931549185, -2.2003020019291415, -2.
171901615775079, -2.1462198897617775, -2.0708014451005345, -2.0462528505720
257, -2.0404233258194835, -2.0402627446489285, -2.0203637768722795, -2.0112
613373599806, -1.9724002238430591, -1.94286610627302, -1.9392117632295243,
-1.936984425728893, -1.8906565447561448, -1.8631433266602286, -1.8217782390
148864, -1.8162570156919269, -1.810003188344754, -1.808703338595561, -1.807
6674726879278, -1.7704086021728656, -1.7593321211179078, -1.756887837746124
3, -1.7554133928653857, -1.750569320363105, -1.7406516324845402, -1.7378320
208019724, -1.7356154972082056, -1.734804928097935, -1.7249237882739805, -1
.7093720345489756, -1.703249235795456, -1.7014494902514201, -1.654530305871
3883, -1.6498286347547335, -1.6207691896645078, -1.6082867413742643, -1.603
2510407618399, -1.587520374715603, -1.5777939051327416, -1.5756782606021573
, -1.5634133983992382, -1.5506052285654783, -1.5472293421663277, -1.5311585
270192445, -1.5260497951136367, -1.5230187151073913, -1.5041130563480523, -
1.495929426267232, -1.488928488990406, -1.4850170999442611, -1.482152697958
483, -1.479410977329876, -1.4791757404318833, -1.468549209759334, -1.453806
2157975324, -1.4508928255602593, -1.440141315714867, -1.435869360473109, -1
.4325935305174224, -1.4238206597096574, -1.4167751815477072, -1.41168378517
0892, -1.4013719517730718, -1.3993760261592312, -1.3938130266590067, -1.391
6935689012007, -1.3881328100242012, -1.3880006896082429, -1.382098295524657
5, -1.3742543092533461, -1.3726719125567546, -1.367273470330649, -1.3592027
896560563, -1.3400487446841078, -1.3280919257369201, -1.3250113201888407, -
1.324772016612398, -1.3143824914267919, -1.3068422815775615, -1.30452908992
12196, -1.3040968326220599, -1.2937294177632017, -1.282844398084, -1.280013
1884002508, -1.2773646011437623, -1.2741672665239727, -1.2735514123733325,
-1.2709185632112263, -1.2671630351597003, -1.259808975793545, -1.2567752877
989178, -1.2545402051284142, -1.2531794742749283, -1.252757872053243, -1.24
82694183757992, -1.2457458105206993, -1.2421736063727484, -1.24185959980386
37, -1.2382937946984387, -1.2293312324295866, -1.2231845481821397, -1.21657
```

```
37641738772, -1.215423480917694, -1.2145902177867947, -1.2095914649412223,
-1.2060629572179256, -1.205430205256844, -1.1867326811480454, -1.1812683133
82949, -1.1781000974991154, -1.1726049980201114, -1.1695734646866607, -1.16
26145374019874, -1.1582724895316943, -1.1565157252621863, -1.15413568157406
29, -1.1532967843840907, -1.1510324577994697, -1.1499437231034895, -1.13911
53021279847, -1.137896544595353, -1.1374986860946386, -1.1297907380629308,
-1.1106415914278254, -1.1084222270240587, -1.0975894295911708, -1.092852988
603211, -1.084462948388932, -1.082467046679021, -1.0743343429657723, -1.071
689507820377, -1.0693810003250854, -1.0659080341781102, -1.062699482069114,
-1.048965102566403, -1.0481866818019994, -1.0377385507124919, -1.0361916225
05032, -1.0295252712741998, -1.0257355901610545, -1.0248656406256293, -1.01
7742638552844, -1.0160642973880403, -1.0060887393378473, -1.000812983288672
7, -0.9954453262337967, -0.9924662233622211, -0.9882756457164076, -0.986503
2321289516, -0.9760834969542421, -0.9753977464263675, -0.9744903766485485,
-0.9698906435597416, -0.9693171730507577, -0.9646911707189645, -0.959308054
0313992, -0.9583374798945019, -0.9534988891423497, -0.9527919011741663, -0.
9506844175876176, -0.9443356419386828, -0.9417987393288646, -0.940614628797
0819, -0.9358747250751188, -0.9294258151949012, -0.9256441331709994, -0.925
1802628715338, -0.9224048838936934, -0.9183735657196141, -0.910532583606256
1, -0.8994483394432224, -0.8987753587294792, -0.8960008273958726, -0.894349
3783852299, -0.8870886999457077, -0.8807880585576818, -0.8796827175377518,
-0.8775418432817119, -0.8764705450236289, -0.8739877302986241, -0.873253270
1726454, -0.8702788105574523, -0.869777869642562, -0.8686958246137985, -0.8
667128823563727, -0.8635047320470319, -0.861699641663399, -0.85922854388379
44, -0.854916790082214, -0.8483794119993254, -0.8453080214183514, -0.828054
0956699486, -0.8276299515515797, -0.8194245824799109, -0.8187872625922716,
-0.8180209220305725, -0.8169464449103894, -0.8132881574817332, -0.802835379
4172581, -0.7980535423049142, -0.7961708415568302, -0.7844281424523314, -0.
7828849555274188, -0.7802754195661014, -0.7789690428388205, -0.776312960274
5066, -0.7751798073493844, -0.7727519031747575, -0.7726423180581405, -0.770
5922767193241, -0.7674925505287274, -0.7659024192557511, -0.752891588880794
7, -0.7519900312092113, -0.7517315205193305, -0.7514529758082011, -0.744726
0707485521, -0.7363113736948063, -0.7334474445253194, -0.7326201856224348,
-0.7256506903986787, -0.7209438762412338, -0.7154415341092004, -0.713570133
9894955, -0.7134863775170979, -0.7131460731904334, -0.7111122671570071, -0.
707531676862265, -0.7049894024058281, -0.7044011237070025, -0.7022876229232
78, -0.7014988927116984, -0.7002801872456602, -0.6996221981322844, -0.69426
11324004141, -0.6918977685097761, -0.690369036331308, -0.6880105718197849,
-0.6841990956923634, -0.6708137396799438, -0.6676027772295461, -0.663592673
3025999, -0.6594800529779793, -0.6549859442003219, -0.6428341211827914, -0.
6358760691211612, -0.6308374195037939, -0.6302511522444788, -0.628547923096
7996, -0.6274944977799717, -0.6237551813548888, -0.6226573831106268, -0.617
7934835551728, -0.6158412140244119, -0.6155766925291596, -0.612670347740761
2, -0.6087285900059592, -0.6064379933403523, -0.6063608233390595, -0.603903
2012489292, -0.6011593683247829, -0.5989577920289656, -0.5964916314083047,
-0.5883333937277603, -0.5882099068690602, -0.5833160216250889, -0.582644002
8468421, -0.579705304209213, -0.5786501945278045, -0.5770965672970282, -0.5
696492554113496, -0.5668099604276047, -0.5656740356461647, -0.5615725456253
935, -0.5594686364235538, -0.5541986282366497, -0.5513581180545766, -0.5513
506805451641, -0.5444807938319691, -0.5218460426749548, -0.5198047906946925
, -0.5180694158406441, -0.517304236368197, -0.516165694418153, -0.513193140
9240715, -0.5115169177310017, -0.5105321359896529, -0.50833791657637, -0.50
43917021975478, -0.502836420074529, -0.49679464096065473, -0.49662460356807
436, -0.49462843072707113, -0.49369150649448706, -0.48987254635662597, -0.4
8785870282923255, -0.48153335975003253, -0.47869646883283934, -0.4770319408
2880157, -0.47654784560109326, -0.4689322457583864, -0.4684624661903925, -0
.46823265018688204, -0.4668111739728931, -0.46399023397769296, -0.460122312
```

```
1540022, -0.45898369098558467, -0.45640732425420644, -0.4537726007288539, -
0.45162330145779483, -0.4455674834526043, -0.43857223014243385, -0.43834352
881988387, -0.4357930296467508, -0.43396165342749105, -0.4330379666318479,
-0.43041399246199963, -0.43006816825605654, -0.42810866014730986, -0.422367
00031656765, -0.42084402332949405, -0.4200670166591578, -0.4188365429534044
, -0.414872302418051, -0.4142874867607288, -0.4140202919353415, -0.40951425
45607318, -0.40901091382403515, -0.40549716313325346, -0.4027297659188288,
-0.4005252154709231, -0.3975952254780433, -0.39381603107142177, -0.39242324
52874855, -0.3900182814639284, -0.38835357207422344, -0.38662660387083714,
-0.3854073811984859, -0.3853935169932932, -0.3824507354425134, -0.380384619
5408077, -0.3795601508487965, -0.3779447048446663, -0.37242990521490515, -0
.3696057119935939, -0.3666316142168159, -0.3661498926400364, -0.36547484099
49389, -0.36521624554002735, -0.3646489011567976, -0.363188463303421, -0.35
79712899783076, -0.3564682084176171, -0.35410031052209867, -0.3506458198154
971, -0.3488274920788745, -0.34778198936650556, -0.34497950559493956, -0.34
285183250623635, -0.3411920684603496, -0.3371694609868189, -0.3336878037194
512, -0.33287071720106665, -0.3322032997369657, -0.3316426218287916, -0.327
77774601738946, -0.3259880356105391, -0.3189448129019931, -0.31391508398222
58, -0.31317163124693037, -0.30816059291625636, -0.30079045270415333, -0.29
951959466354067, -0.29334950243810215, -0.29315911168413306, -0.28207681191
587647, -0.2796933380407038, -0.2779310972565784, -0.2770942188793804, -0.2
7697191992496806, -0.27537566328150764, -0.2747411953055142, -0.27145267575
166915, -0.26720825839222756, -0.26173317209553354, -0.2540076405811383, -0
.2508775740556845, -0.2505040869974144, -0.24941712755374496, -0.2462258774
6071373, -0.24149911420726353, -0.24046864785867159, -0.24000649365791477,
-0.2394952211651457, -0.23901369444089726, -0.23215288328709044, -0.2319388
7280187392, -0.22187573071296102, -0.2210488605986583, -0.21859136673722793
, -0.2180368070153506, -0.2162258105134832, -0.21476416122862915, -0.209995
55876243206, -0.2072050222206171, -0.20717472498467066, -0.2064363874854805
2, -0.20518417022805505, -0.2045722576592557, -0.20349784998945541, -0.2027
3914479937363, -0.20205896994432818, -0.19959244506060486, -0.1979117165673
0086, -0.1909329981631783, -0.18957993272657034, -0.1837388118590731, -0.18
332506512679012, -0.18329703516708323, -0.17723465502543473, -0.17285363703
586704, -0.16961289823458905, -0.16894789746369893, -0.16869999133881683, -
0.16702362235313437, -0.16591605421390115, -0.16309324204701567, -0.1629648
9187864044, -0.16226691110841254, -0.1603501090760094, -0.15843183392584034
, -0.1554747519029398, -0.154623413700289, -0.15421986948761704, -0.1506416
8582241913, -0.14976394782579594, -0.14584704801323997, -0.1385833792584932
6, -0.13317284079632946, -0.13124383370585607, -0.13056915294021246, -0.127
7078397980036, -0.12569491414735445, -0.12095745369479345, -0.1179218950494
6186, -0.11701524648039029, -0.11549523053292597, -0.11343548614594968, -0.
11074110982989568, -0.10821724341037156, -0.1075453630885765, -0.1066475327
9529232, -0.10544201918267125, -0.10436498625071812, -0.10331550103668474,
-0.10044344532710084, -0.09701476358581622, -0.08614686708001874, -0.081963
02197659595, -0.0814974491733973, -0.08028507454025748, -0.0802355278236553
3, -0.07747176297178497, -0.07683795637818484, -0.07619702327575534, -0.072
45362168314493, -0.07239154614493182, -0.07140802148952768, -0.071373399031
1528, -0.06666522141515242, -0.06386657522867567, -0.05278727002976925, -0.
05186493718797062, -0.05044294807628709, -0.050313131472528, -0.04998591310
375418, -0.04988552195769557, -0.04333380240931282, -0.03959527584830538, -
0.039332188164504366, -0.03873495639315048, -0.03351097429448678, -0.029826
61022595737, -0.019872088928278204, -0.016596484283469354, -0.0164440038992
55277, -0.010633545318607362, -0.00715879843580901, 0.0013785584592752875,
0.0023483149553989767, 0.003154954029947473, 0.006350254317285005, 0.006533
761401282986, 0.015731075353449694, 0.01623558938459376, 0.0231070684075672
63, 0.024368444129188552, 0.029705313514061134, 0.030347217156985285, 0.033
22170510338486, 0.03330449889021281, 0.0345679179206753, 0.0349137400266041
```

```
9, 0.03545055835725072, 0.03713658928988133, 0.03999421639590009, 0.0417076
1658793494, 0.045505190091167734, 0.050252361634504604, 0.05203967149048037
4, 0.05424061732008706, 0.06329918528737644, 0.06520916542758512, 0.0673068
8926160423, 0.0745187329745932, 0.07794958214052201, 0.07947467915916183, 0
.08118793075363426, 0.08123788217807096, 0.08257095359706548, 0.08707329466
20141, 0.08793596445936248, 0.08816999570717747, 0.09835394289462505, 0.099
50650427190938, 0.10335706068847773, 0.10470295459917897, 0.104820424716065
16, 0.10587541711429244, 0.10767646526846267, 0.11489021107156422, 0.117417
16394305614, 0.1194709928674875, 0.1207613681845075, 0.12389955278526613, 0
.1272595367809018, 0.12736075659071738, 0.13086589830591389, 0.131157253513
4371, 0.13220127158215922, 0.13483933691023114, 0.1350916395803685, 0.13635
867453360684, 0.13841323723384424, 0.13987518955536168, 0.1429077838979949,
0.1431607845305525, 0.14733784903540273, 0.15262999118744244, 0.15568450001
968512, 0.15697601219151244, 0.15916666139409016, 0.160480716176075, 0.1606
1952422218298, 0.16114239181509665, 0.161950088314343, 0.16305921456967595,
0.16458279499447748, 0.1658118049731377, 0.16724540379333816, 0.17349671303
02882, 0.17744805125653032, 0.17956964337801962, 0.17988843345649874, 0.185
46464359790418, 0.19103915943283215, 0.1936045986916307, 0.1942557294520065
3, 0.19428092535641336, 0.2048271814046292, 0.2052294891635873, 0.205689458
00790678, 0.2138241277018392, 0.2206705619808119, 0.22844380356577518, 0.23
043166016308364, 0.23058844428061373, 0.2331236381769873, 0.233271193598025
56, 0.2356563641273258, 0.23637298252629022, 0.2412217880541968, 0.24504629
866148678, 0.24982565178591398, 0.25107183082351814, 0.2532012145222722, 0.
25333331672016396, 0.2570726265104106, 0.2589018118051762, 0.26066886823213
55, 0.26221413274700683, 0.2623083277760132, 0.2625077488936187, 0.26580445
122147, 0.2681267883196285, 0.2712684624813136, 0.27537714383826906, 0.2769
2638493511235, 0.2881581653334144, 0.29014282728661683, 0.2913962153172162,
0.2938343581975014, 0.29659875926241147, 0.29669210786068356, 0.29757571190
813026, 0.29768080849802747, 0.2993146362470675, 0.30352570685513874, 0.313
62852060722374, 0.3151205820219284, 0.31720390377511315, 0.3186337245747183
, 0.32077985386167646, 0.32511723179553315, 0.326909919968972, 0.3296598934
5226815, 0.3300074058497535, 0.33411602776591365, 0.3416632923293612, 0.345
99323863947823, 0.35290952366848294, 0.35697107572355885, 0.358056698365699
7, 0.35810858566138754, 0.3602868925355878, 0.3635689416173394, 0.369815712
5156603, 0.3758878798957954, 0.37778906896044706, 0.37961649784066087, 0.38
104550384697294, 0.3816067085326987, 0.38260534087300996, 0.389522375555246
05, 0.3906667661200182, 0.39142189854400244, 0.393761526951727, 0.396086031
9170701, 0.40006142617609397, 0.4067556562815385, 0.4106589140067872, 0.412
40972860484476, 0.41836504064178737, 0.419862020519515, 0.4202884823301728,
0.42126210195131725, 0.42365158233032213, 0.43105556521376354, 0.4401547746
4056824, 0.44082340961197714, 0.44938750389252785, 0.4507858026908138, 0.45
087675697788093, 0.45210022770564495, 0.4566865101376185, 0.463976280597687
8, 0.4647262386948913, 0.4658449458468793, 0.46920060827297927, 0.471269617
56028235, 0.47387266320450394, 0.4827779361959991, 0.4904695482220737, 0.49
2129838231158, 0.498358141384247, 0.5034462880621449, 0.5040416209726837, 0
.5095667096830103, 0.518772637726897, 0.5227394093657247, 0.522966899789704
1, 0.5253011201355832, 0.5253274948024691, 0.5277850127139294, 0.5295734735
966323, 0.5445990125789485, 0.5476049341675896, 0.5484203571480284, 0.55002
26550900273, 0.5518220080483, 0.5539618359506492, 0.5553587279312261, 0.562
9809007526595, 0.5671834025079884, 0.56817541420235, 0.5702277288745232, 0.
575005118079826, 0.5778474314509322, 0.5792413206266093, 0.5815616732683625
, 0.5910615572534844, 0.592070023595102, 0.5925025309334665, 0.600743099635
7036, 0.6029415853203144, 0.6044326057043586, 0.6048560751808764, 0.6050197
43133535, 0.6052092441564807, 0.6110781289730605, 0.6200407181436018, 0.624
7738418093833, 0.6256997469346864, 0.63020819202168, 0.6312490614781776, 0.
6321093708385737, 0.6350226977245118, 0.6376846929343857, 0.641208547002993
2, 0.641568738839276, 0.6420549371509142, 0.6439839532219038, 0.64969716424
```

```
98753, 0.6506445282631083, 0.6532280872664986, 0.6556772230924163, 0.655891
0548118162, 0.6587000646302532, 0.6664324987736386, 0.6756435279134582, 0.6
765791752962923, 0.6843663789826353, 0.6869066252253597, 0.6887564000728579
, 0.6891262444687851, 0.6987509937748353, 0.7022384884060815, 0.70301710966
86977, 0.7034478651708561, 0.7055287029203832, 0.7071551402643786, 0.707214
6251922344, 0.7073358311764937, 0.7095225449113856, 0.709632992651454, 0.71
24627785574438, 0.716025431949382, 0.7269661668824416, 0.7285278853649637,
0.7301235492575104, 0.7312516711979727, 0.7323435688771215, 0.7352616602232
744, 0.738415739849822, 0.7398586735771887, 0.7406409482365951, 0.743128529
8711009, 0.7488758404430063, 0.7496715908171524, 0.7501121677201529, 0.7548
352632214356, 0.7608802211496055, 0.7649592116958653, 0.7760273506434778, 0
.7760694281373708, 0.790252115902819, 0.7980532439745892, 0.802863826127435
6, 0.8044357873544034, 0.8136203029031704, 0.8162292114228865, 0.8228809982
03601, 0.827614623848311, 0.8289077795590023, 0.8327241277986833, 0.8409229
778387679, 0.844451481575653, 0.847007731201975, 0.8473292460795396, 0.8489
194874116324, 0.8595918516988073, 0.8605782017590868, 0.8634780649459732, 0
.8672523261343031, 0.871331014967531, 0.8727586345724986, 0.876058656864995
, 0.8770790005746338, 0.8784178265300318, 0.8785981636765009, 0.88161879782
29301, 0.8845571155532614, 0.8897412219890252, 0.8942321620390763, 0.897028
1469987128, 0.9146572259877291, 0.9165180687212439, 0.9192051114746158, 0.9
196560691629266, 0.920314464914683, 0.9214951614212589, 0.9286217186490507,
0.9295426070146645, 0.9296491062553217, 0.9361787279888134, 0.9383473011271
499, 0.9404974958619619, 0.9566558729300657, 0.9581408997342086, 0.96951101
72306652, 0.9702843140421922, 0.9707420241107952, 0.9714347037468662, 0.983
5646315849972, 0.9898623322560655, 0.9973184276715628, 0.9980014587883456,
1.0135085613801258, 1.0153949310114827, 1.016027839857299, 1.02076525470972
26, 1.0242496969292403, 1.0253016037915972, 1.025337429763006, 1.0279733908
600968, 1.035972812693877, 1.0403274079704736, 1.0420612183761127, 1.042131
6389082618, 1.0468160877160761, 1.0542555218322054, 1.0546329845927152, 1.0
551693887526, 1.0622490166656533, 1.0678649066165549, 1.0790543816318603, 1
.0801870278538002, 1.081431955293906, 1.084403549207061, 1.0872397549098374
, 1.0874751515865617, 1.0887040489994566, 1.0983847369115152, 1.10551542810
82655, 1.110464548362276, 1.113788041834394, 1.119076641378699, 1.122337385
0949274, 1.122776887499169, 1.1235714010865085, 1.1237536341853784, 1.13204
3826773241, 1.136384060572724, 1.1376771456910795, 1.1527519906030208, 1.16
41191670680349, 1.1695923937614081, 1.1708219783769214, 1.1745022225973694,
1.1780584794558033, 1.1910992513931258, 1.19292707171586, 1.196000632855619
2, 1.2047485173469974, 1.207630536302557, 1.226496059322752, 1.234723697475
0648, 1.2401750805839653, 1.243886589152799, 1.2449216020092635, 1.24546684
21733956, 1.2483250947543265, 1.2485041287685403, 1.2498026345665787, 1.251
8145709315258, 1.2535272200653926, 1.2536828061263634, 1.256521788479392, 1
.2572391187430105, 1.261261111698065, 1.2639526168062116, 1.265600613604607
8, 1.2815402065160588, 1.289839977610563, 1.2931959108180597, 1.29384593055
26047, 1.3127027257449453, 1.318285781950461, 1.3189560843969048, 1.3193867
808517794, 1.3217289624748423, 1.3272220070157559, 1.3284940902087945, 1.33
01996323582665, 1.338014825035364, 1.3552760097908503, 1.3556257779523848,
1.3558615197153303, 1.3568643447901017, 1.3570670258884148, 1.3599274051494
543, 1.3617468169372702, 1.3645270629695152, 1.371289902480054, 1.375186238
1327769, 1.384415232944221, 1.3941909537355606, 1.3974793146748146, 1.40563
7273696578, 1.4056603225959843, 1.4099858200558373, 1.411015351364111, 1.42
69848516863635, 1.4302385543267129, 1.4319469939877199, 1.43315384033127, 1
.4364165137266889, 1.4415541250450277, 1.443962772836627, 1.444936677298199
7, 1.445418721050606, 1.4469428953283954, 1.4707775104530865, 1.47565922385
62802, 1.4757019864235794, 1.4838494788956094, 1.4887728639551263, 1.500449
3867930078, 1.5035797145265515, 1.5085355286825528, 1.5268113466747657, 1.5
365591299333081, 1.5375912110484151, 1.5464186921677163, 1.5668919551032683
, 1.5759992861915697, 1.5856570928595564, 1.59557439895527, 1.5969498835098
```

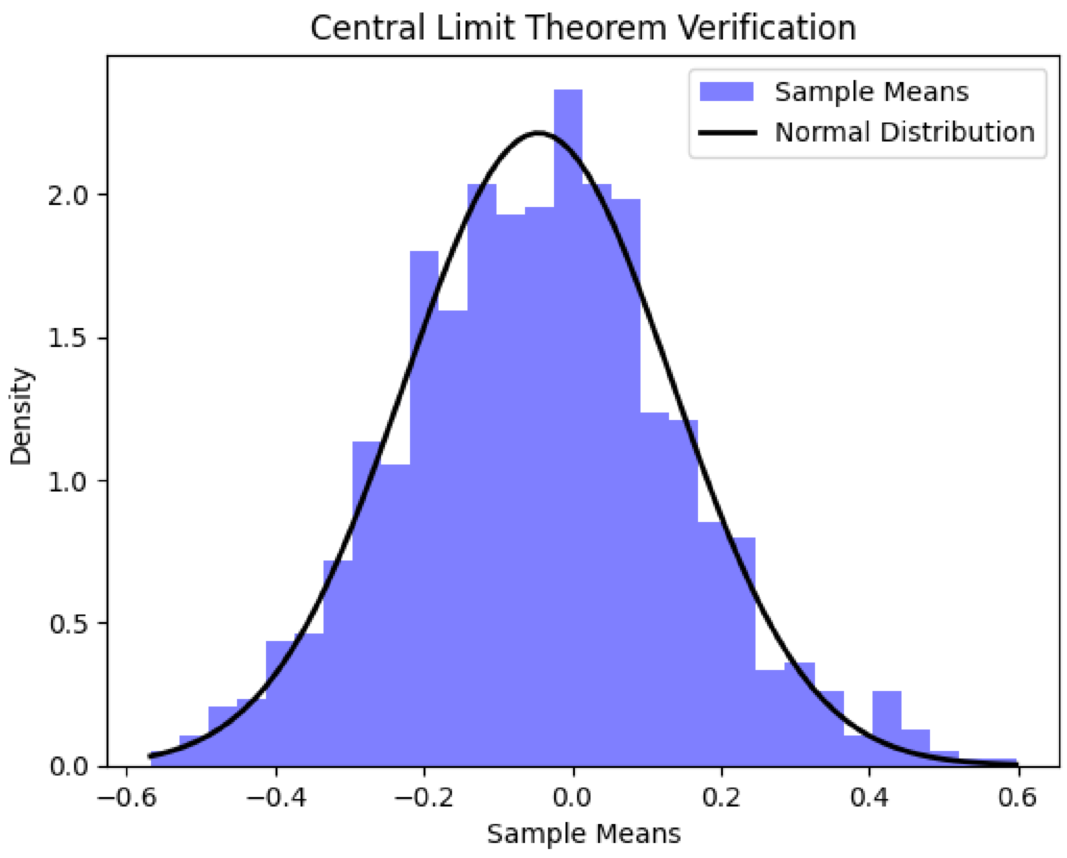
```
122, 1.60422536537862, 1.613417812139645, 1.6219892259924467, 1.62428969130
1258, 1.6420456250470805, 1.6463548744336565, 1.6486987947788925, 1.6706250
334319694, 1.682198205548185, 1.694579407465392, 1.6985538829969917, 1.7024
505111023438, 1.703633473956976, 1.7083485402124587, 1.7130570098189382, 1.
7792112822142216, 1.8006462469564608, 1.817226503231325, 1.8174020629705745
, 1.8177892172320573, 1.8373881942591692, 1.8490525215473295, 1.86231074870
95114, 1.8659891667418023, 1.8724869401217825, 1.877682525446926, 1.8865480
868491933, 1.8894004915689837, 1.891070314620199, 1.9391193244861336, 1.943
4099192504983, 1.9731407481513061, 1.99710063705634, 2.0027203078910834, 2.
0136304173115986, 2.0606608222204055, 2.067973016339695, 2.090939959147377,
2.1398594613829918, 2.1606388613164125, 2.2425072840330222, 2.3198587463467
94, 2.3420081229706082, 2.3716292966314128, 2.3727403403463754, 2.512449511
0518006, 2.5238657224686687, 2.567765785710375, 2.5969556502880695, 2.62695
31085818234, 2.6278782238055487, 2.63550312606235, 2.6804789010942787, 2.84
17629698810103, 2.9569548509121066, 3.0524497771817094, 3.2250173161371234]
Skewness: 0.0843116509495048
Kurtosis: -0.1531850406731352
```

We use different data and observed various different statistical studies by using various plots and graphs ,These visualizations helps in knowing the various math approaches like mean, skewness, and kurtosis provide further meaningful observances .

In [3]:

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

np.random.seed(0)

std_normal_distribution_data = np.random.normal(loc=0, scale=1, size=1000)
normally_distributed_data = std_normal_distribution_data


def clt_verification(data, sample_size, no_of_samples):
    means_of_samples = []

    for _ in range(no_of_samples):
        sample = np.random.choice(data, size=sample_size, replace=True)
        means_of_samples.append(np.mean(sample))


    plt.hist(means_of_samples, bins=30, density=True, alpha=0.5,
color='blue', label='Sample Means')


    mean_of_data, std_dev_of_data = np.mean(data), np.std(data)
    x = np.linspace(min(means_of_samples), max(means_of_samples), 100)
    pdf = norm.pdf(x, mean_of_data, std_dev_of_data / np.sqrt(sample_size))
    plt.plot(x, pdf, 'k-', linewidth=2, label='Normal Distribution')

    plt.title(f'Central Limit Theorem Verification')
    plt.xlabel('Sample Means')
    plt.ylabel('Density')
    plt.legend()
    plt.show()
```

```
sample_size = 30
no_of_samples = 1000
clt_verification(std_normal_distribution_data, sample_size, no_of_samples)
```



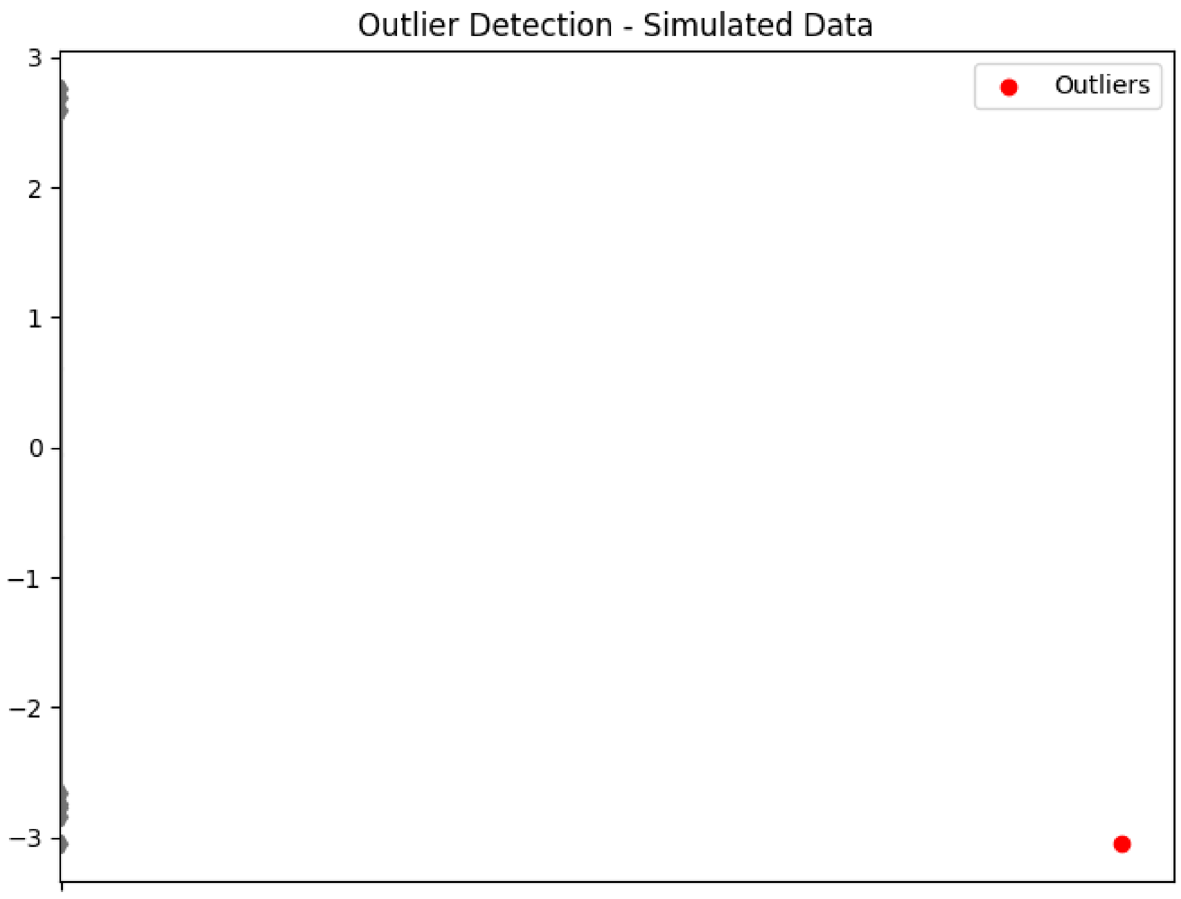
The clt_verification function uses simulations to illustrate the Central Limit Theorem (CLT. It creates values by taking samples of a specified size from the dataset. Through repeated sampling and averaging it shows how these averages tend to resemble a distribution. The function requires three inputs; data, sample size and number of samples. It randomly selects samples calculates their averages and creates a histogram that overlays a curve representing the distribution. This visualization demonstrates how the sample averages tend to follow a bell shaped curve, which aligns with what we expect from a distribution based on the Central Limit Theorem (CLT). By adjusting the values of sample_size and no_of_samples we can explore how different sample sizes and numbers affect the convergence, towards a distribution. This is a concept, in statistics as it shows that regardless of the datas distribution under certain conditions sample means tend to follow a normal distribution pattern.

In [4]:

```
import seaborn as sns
import matplotlib.pyplot as plt

def detect_outliers_in(data, threshold=3):
    z_scores_of_data = (data - np.mean(data)) / np.std(data)
    outliers_of_data = np.abs(z_scores_of_data) > threshold
    return outliers_of_data

outliers_of_data = detect_outliers_in(normally_distributed_data)


print(f'Outliers: {normally_distributed_data[outliers_of_data]}')

plt.figure(figsize=(8, 6))
```

```
sns.boxplot(y=normally_distributed_data, color='lightgreen')

outliers_of_data = detect_outliers_in(normally_distributed_data)

indices_of_outliers = np.where(outliers_of_data)[0]
plt.scatter(indices_of_outliers,
normally_distributed_data[outliers_of_data], c='red', label='Outliers')

plt.title('Outlier Detection - Simulated Data')
plt.legend()
plt.show()
```

```
Outliers: [-3.04614305]
```

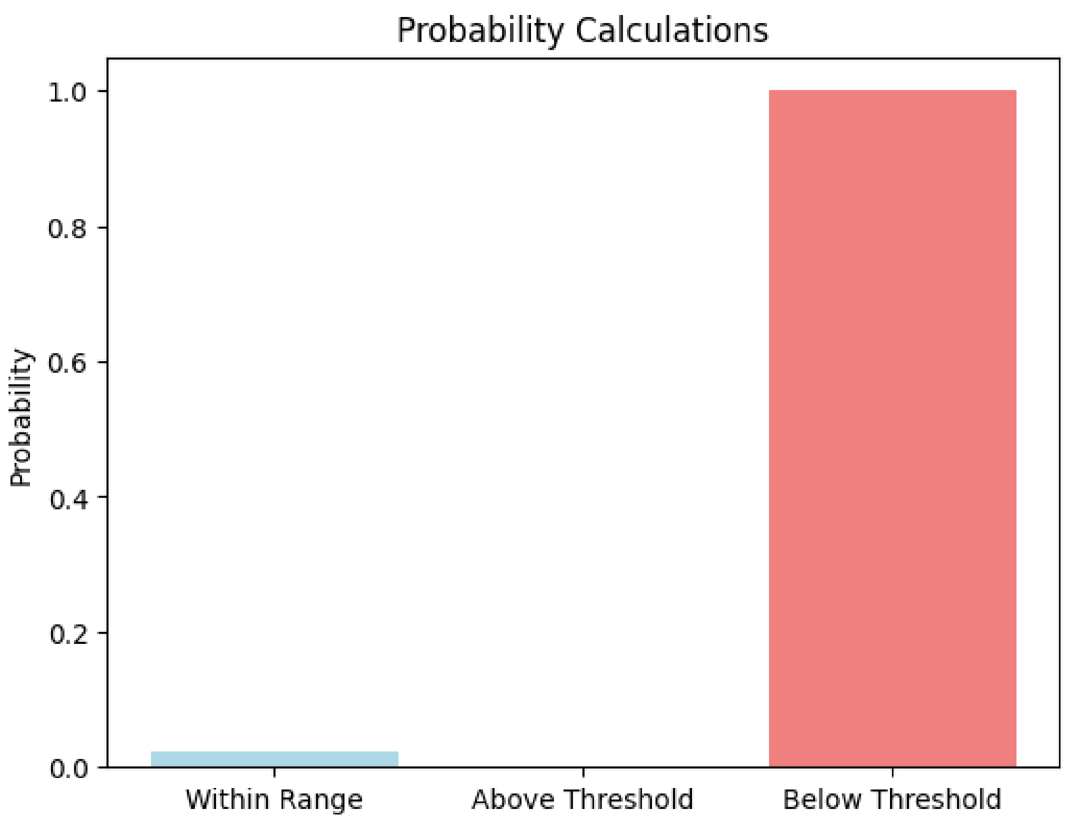


This code genarates a function called "detect_outliers_in," which it uses the Z scores to mark outliers that go more than the given threshold of 3 deviations. This function uses. Mark these outliers as it understand them to identify extreme values. The process to recollect the outliers and display them is essential, for various data and identifying patterns that might affect analysis. By varying the threshold or using display patterns we can gain insights, into outlier detection across different datasets.

In [5]:

```
import matplotlib.pyplot as plt

def calculate_probabilities_of(data, lower_bound, upper_bound, threshold):
    prob_of_data_within_range = np.mean((data >= lower_bound) & (data <=
upper_bound))
    prob_of_data_above_threshold = np.mean(data > threshold)
    prob_of_data_below_threshold = np.mean(data < threshold)
    return prob_of_data_within_range, prob_of_data_above_threshold,
prob_of_data_below_threshold

lower_bound = 2
upper_bound = 4
threshold = 3.5


prob_of_data_within_range, prob_of_data_above_threshold,
prob_of_data_below_threshold =
calculate_probabilities_of(normally_distributed_data, lower_bound,
upper_bound, threshold)


print(f'Probability within range: {prob_of_data_within_range}')
print(f'Probability above threshold: {prob_of_data_above_threshold}')
print(f'Probability below threshold: {prob_of_data_below_threshold}')


all_probabilities = [prob_of_data_within_range,
prob_of_data_above_threshold, prob_of_data_below_threshold]
labels_of_propbabilities = ['Within Range', 'Above Threshold', 'Below
Threshold']

plt.bar(labels_of_propbabilities, all_probabilities, color=['lightblue',
'lightgreen', 'lightcoral'])
plt.title('Probability Calculations')
plt.ylabel('Probability')
plt.show()
```

```
Probability within range: 0.023
Probability above threshold: 0.0
Probability below threshold: 1.0
```



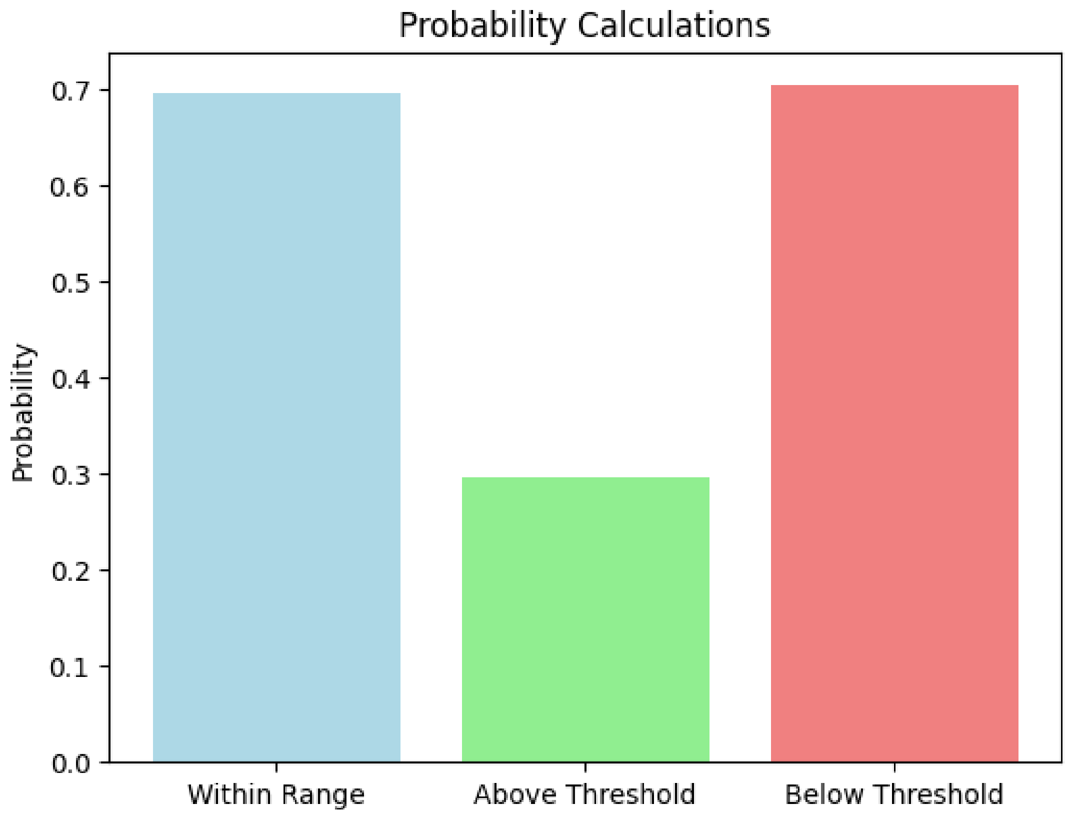
The code has of functions that generate probabilities, for instances like those within a range (i.e within range or out of) a given threshold. It then shows the probabilities with given raw data. By utilizing matplotlib it generates a bar chart that visualizes the probabilities assigning colors to each scenario. This graphical representation facilitates comparison and comprehension of the probabilities. The code offers flexibility to calculate and visualize probability scenarios in a dataset allowing adjustments of parameters and visual elements based on the datasets characteristics or research needs.

In [6]:

```
def calculate_probabilities_of(data, lower_bound, upper_bound, threshold):
    prob_of_data_within_range = np.mean((data >= lower_bound) & (data <=
upper_bound))
    prob_of_data_above_threshold = np.mean(data > threshold)
    prob_of_data_below_threshold = np.mean(data < threshold)

    return prob_of_data_within_range, prob_of_data_above_threshold,
prob_of_data_below_threshold

std_normal_distributed_data = np.random.normal(loc=3, scale=1, size=1000)


lower_bound = 2
upper_bound = 4
threshold = 3.5


prob_of_data_within_range, prob_of_data_above_threshold,
prob_of_data_below_threshold =
calculate_probabilities_of(std_normal_distributed_data, lower_bound,
upper_bound, threshold)
```

```
print(f'Probability within range: {prob_of_data_within_range}')
print(f'Probability above threshold: {prob_of_data_above_threshold}')
print(f'Probability below threshold: {prob_of_data_below_threshold}')

all_probabilities = [prob_of_data_within_range,
prob_of_data_above_threshold, prob_of_data_below_threshold]
labels_of_propbabilities = ['Within Range', 'Above Threshold', 'Below
Threshold']

plt.bar(labels_of_propbabilities, all_probabilities, color=['lightblue',
'lightgreen', 'lightcoral'])
plt.title('Probability Calculations')
plt.ylabel('Probability')
plt.show()
```

```
Probability within range: 0.695
Probability above threshold: 0.297
Probability below threshold: 0.703
```



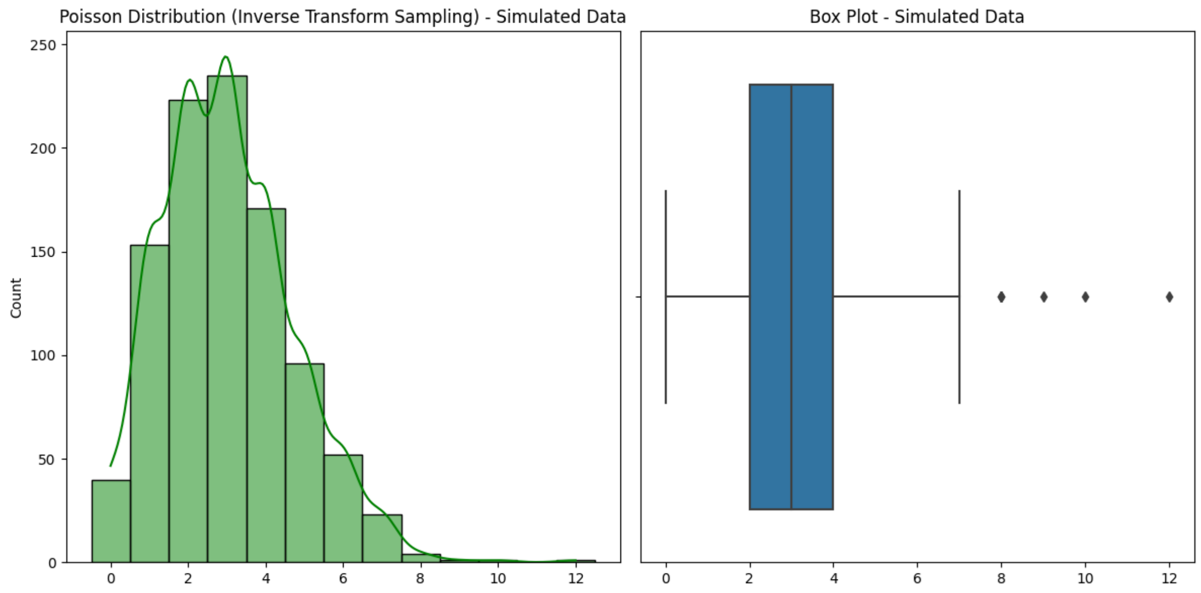
Here the function calculate_probabilities_of, which computes probabilities based on a given dataset. It determines the probability of data falling within a specified range, the probability of data being above a given threshold, and the probability of data being below that threshold. It then calculates and displays these probabilities using a bar plot, providing insights into the occurrence likelihood of data within a range or exceeding/falling below a certain threshold in the dataset generated from a normal distribution centered at 3 with a standard deviation of 1.

4.1.2 Simulating from Discrete Distributions:

In [7]:

```
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from scipy.stats import poisson, skew, kurtosis
import pandas as pd


def inverse_transform_discrete_sampling(distribution, size=1000):
    random_data = np.random.rand(size)
    return distribution.ppf(random_data).astype(int)


def perform_discrete_statistical_analysis_for(data):
    mean_of_data = np.mean(data)
    variance_of_data = np.var(data)
    std_dev_of_data = np.std(data)
    quantiles_of_data = np.percentile(data, [25, 75])

    modes_of_data = pd.Series(data).mode()
    if len(modes_of_data) > 0:
        mode_of_data = modes_of_data.tolist()
    else:
        mode_of_data = None

    order_of_data = np.sort(data)
    skewness_of_data = skew(data)
    kurt_of_data = kurtosis(data)

    return mean_of_data, variance_of_data, std_dev_of_data,
quantiles_of_data, mode_of_data, order_of_data, skewness_of_data,
kurt_of_data

lambda_value = 3
simulated_data_poisson =
inverse_transform_discrete_sampling(poisson(mu=lambda_value), size=1000)


mean_of_data, variance_of_data, std_dev_of_data, quantiles_of_data,
mode_of_data, order_of_data, skewness_of_data, kurt_of_data =
perform_discrete_statistical_analysis_for(simulated_data_poisson)


plt.figure(figsize=(12, 6))
plt.subplot(1, 2, 1)
sns.histplot(simulated_data_poisson, kde=True, color='green',
discrete=True)
plt.title('Poisson Distribution (Inverse Transform Sampling) - Simulated
Data')

plt.subplot(1, 2, 2)
sns.boxplot(x=simulated_data_poisson)
plt.title('Box Plot - Simulated Data')

plt.tight_layout()
plt.show()
```

```
print("\nStatistical Analysis for Poisson Distribution (Inverse Transform
Sampling) - Simulated Data:\n")
print(f"Mean: {mean_of_data}\nVariance: {variance_of_data}\nStandard
Deviation: {std_dev_of_data}")
print(f"Quantiles (25th, 75th): {quantiles_of_data}\nMode:
{mode_of_data}\nSkewness: {skewness_of_data}\nKurtosis: {kurt_of_data}\n")
```



```
Statistical Analysis for Poisson Distribution (Inverse Transform Sampling)
- Simulated Data:

Mean: 3.004
Variance: 2.851984
Standard Deviation: 1.688781809470957
Quantiles (25th, 75th): [2. 4.]
Mode: [3]
Skewness: 0.6315483722107328
Kurtosis: 0.740811201074886
```

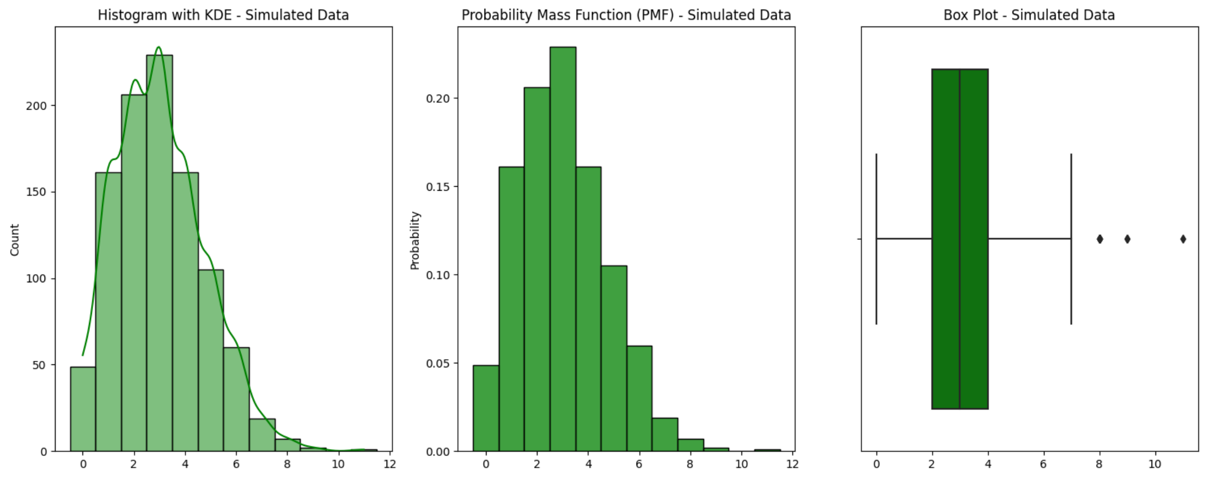
This piece of code performs an analysis. Creates visualizations, for a simulated dataset that follows a Poisson distribution using inverse transform sampling. The analysis includes metrics and graphical representations to gain insights into the properties and characteristics of the dataset.

1. Statistical Analysis Function; The perform_discrete_statistical_analysis_for function calculates statistics for the Poisson data. It computes metrics such as the variance, standard deviation quantiles (specifically the 25th and 75th percentiles) mode order statistics, skewness and kurtosis. These measures provide information about the tendency, spread, shape and tail behavior of the generated Poisson dataset.

2. Visualization; The code generates two plots to represent the Poisson data. The first plot shows a histogram with Kernel Density Estimation (KDE) which gives an understanding of how the data's distributed. The second plot presents a box plot that summarizes aspects like tendency spread and potential outliers in a concise manner. By performing analysis and creating visualizations together we aim to gain an understanding of the characteristics of this simulated Poisson dataset. These analyses help quantify aspects such, as shape, variability and skewness of the distribution while facilitating an exploration into its underlying statistical properties.

In [8]:

```
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from scipy.stats import poisson, skew, kurtosis
import pandas as pd

def inverse_transform_discrete_sampling(distribution, size=1000):
    random_data = np.random.rand(size)
    return distribution.ppf(random_data).astype(int)


def perform_discrete_statistical_analysis_for(data):
    mean_of_data = np.mean(data)
    variance_of_data = np.var(data)
    std_dev_of_data = np.std(data)
    quantiles_of_data = np.percentile(data, [25, 75])

    modes_of_data = pd.Series(data).mode()
    if len(modes_of_data) > 0:
        mode_of_data = modes_of_data.tolist()
    else:
        mode_of_data = None

    order_of_data = np.sort(data)
    skewness_of_data = skew(data)
    kurt_of_data = kurtosis(data)

    return mean_of_data, variance_of_data, std_dev_of_data,
quantiles_of_data, mode_of_data, order_of_data, skewness_of_data,
kurt_of_data


lambda_value = 3
simulated_data_poisson =
inverse_transform_discrete_sampling(poisson(mu=lambda_value), size=1000)


mean_of_data, variance_of_data, std_dev_of_data, quantiles_of_data,
mode_of_data, order_of_data, skewness_of_data, kurt_of_data =
perform_discrete_statistical_analysis_for(simulated_data_poisson)


plt.figure(figsize=(15, 6))


plt.subplot(1, 3, 1)
sns.histplot(simulated_data_poisson, kde=True, color='green',
discrete=True)
plt.title(f'Histogram with KDE - Simulated Data')


plt.subplot(1, 3, 2)
sns.histplot(simulated_data_poisson, stat="probability", discrete=True,
color='green')
plt.title(f'Probability Mass Function (PMF) - Simulated Data')
plt.subplot(1, 3, 3)
sns.boxplot(x=simulated_data_poisson, color='green')
plt.title(f'Box Plot - Simulated Data')

plt.tight_layout()
plt.show()


print("\nStatistical Analysis for Poisson Distribution (Inverse Transform
Sampling) - Simulated Data:\n")
print(f"Mean: {mean_of_data}\nVariance: {variance_of_data}\nStandard
Deviation: {std_dev_of_data}")
print(f"Quantiles (25th, 75th): {quantiles_of_data}\nMode:
{mode_of_data}\nSkewness: {skewness_of_data}\nKurtosis: {kurt_of_data}\n")
```



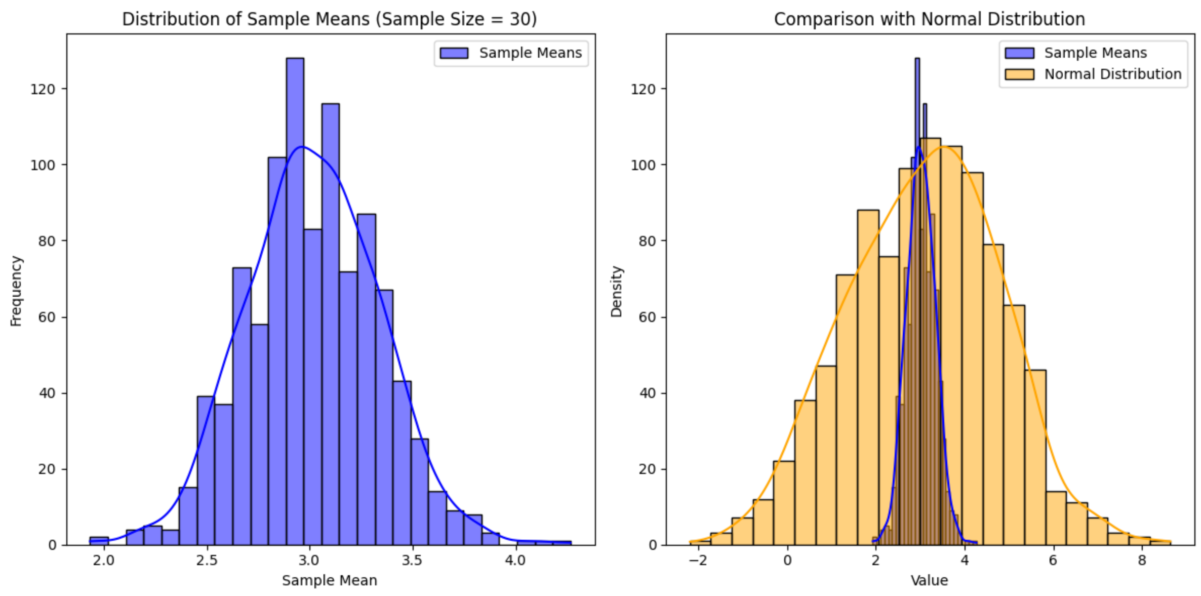
```
Statistical Analysis for Poisson Distribution (Inverse Transform Sampling)
- Simulated Data:

Mean: 3.007
Variance: 3.026951
Standard Deviation: 1.7398134957517717
Quantiles (25th, 75th): [2. 4.]
Mode: [3]
Skewness: 0.5270147484465152
Kurtosis: 0.19788422888899637
```

The code generates a dataset by using a method called inverse transform sampling. It follows a Poisson distribution with a value of 3. Then it performs a analysis that includes calculations, for mean, variance, percentiles, mode, skewness and kurtosis. These calculations give us insights into the tendency (spread (variation) and shape of the data. Additionally the code creates three visualizations; a histogram, with Kernel Density Estimation to show how the data is distributed a Probability Mass Function plot to highlight probabilities and a box plot to help identify any outliers or extreme values. This combined approach allows us to thoroughly explore and understand the characteristics and distribution of the datasets probability distribution.

In [9]:

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

```python
import seaborn as sns
from scipy.stats import poisson, norm


def generate_means_of_samples(data, sample_size, no_of_samples):
    means_of_samples = np.zeros(no_of_samples)
    for i in range(no_of_samples):
        sample = np.random.choice(data, size=sample_size, replace=True)
        means_of_samples[i] = np.mean(sample)
    return means_of_samples


def inverse_transform_discrete_sampling(distribution, size):
    random_data = np.random.rand(size)
    return distribution.ppf(random_data).astype(int)


lambda_value = 3
size_of_data = 10000
sample_data_poisson =
inverse_transform_discrete_sampling(poisson(mu=lambda_value), size_of_data)


sample_size = 30
no_of_samples = 1000
means_of_samples = generate_means_of_samples(sample_data_poisson,
sample_size, no_of_samples)


plt.figure(figsize=(12, 6))


plt.subplot(1, 2, 1)
sns.histplot(means_of_samples, kde=True, color='blue', label='Sample
Means')
plt.title(f'Distribution of Sample Means (Sample Size = {sample_size})')
plt.xlabel('Sample Mean')
plt.ylabel('Frequency')
plt.legend()


plt.subplot(1, 2, 2)
sns.histplot(means_of_samples, kde=True, color='blue', label='Sample
Means')
sns.histplot(norm.rvs(size=no_of_samples, loc=np.mean(sample_data_poisson),
scale=np.std(sample_data_poisson)),
             kde=True, color='orange', label='Normal Distribution')
plt.title(f'Comparison with Normal Distribution')
plt.xlabel('Value')
plt.ylabel('Density')
plt.legend()

plt.tight_layout()
plt.show()
```

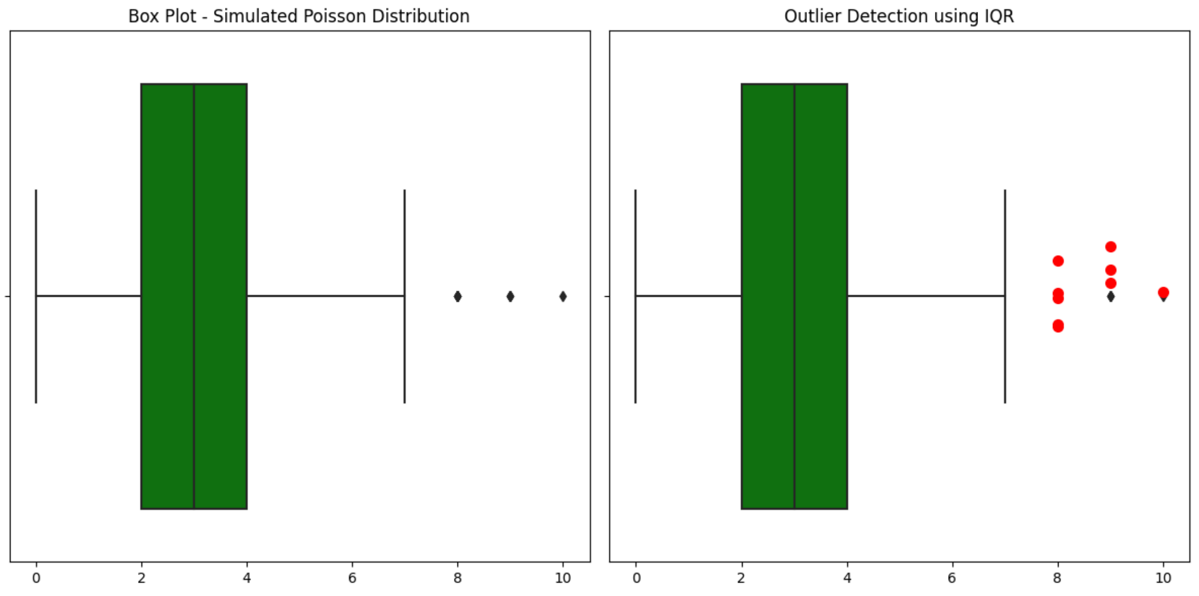


This piece of code demonstrates the Central Limit Theorem (CLT) by creating a dataset that consists of 10,000 values, from a Poisson distribution with a value of 3. The dataset is generated using inverse transform sampling. By selecting samples of size 30 from this dataset it calculates the sample means. Shows how these 24 means vary across the different samples. The histogram, in the subplot illustrates the distribution of these means, which approximates a distribution as predicted by the CLT. Additionally the right subplot compares the distribution of sample means with a distribution derived from statistics of the original dataset confirming that it converges towards normality.. Ultimately, this illustrates the CLT's fundamental premise: the sampling distribution of means tends towards normality regardless of the population's underlying distribution, providing a clear visual demonstration of this statistical principle.

In [10]:

```
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from scipy.stats import poisson


def inverse_transform_discrete_sampling(distribution, size=1000):
    random_data = np.random.rand(size)
    return distribution.ppf(random_data).astype(int)

lambda_value = 3
data_size = 1000
sample_data_poisson =
inverse_transform_discrete_sampling(poisson(mu=lambda_value),
size=data_size)


plt.figure(figsize=(12, 6))


plt.subplot(1, 2, 1)
sns.boxplot(x=sample_data_poisson, color='green')
plt.title('Box Plot - Simulated Poisson Distribution')


q1 = np.percentile(sample_data_poisson, 25)
q3 = np.percentile(sample_data_poisson, 75)
```

```
iqr = q3 - q1
lower_bound = q1 - 1.5 * iqr
upper_bound = q3 + 1.5 * iqr

outliers_of_data = (sample_data_poisson < lower_bound) |
(sample_data_poisson > upper_bound)


plt.subplot(1, 2, 2)
sns.boxplot(x=sample_data_poisson, color='green')
sns.stripplot(x=sample_data_poisson[outliers_of_data], color='red', size=8)
plt.title('Outlier Detection using IQR')

plt.tight_layout()
plt.show()


print("Identified Outliers:")
print(sample_data_poisson[outliers_of_data])
```



```
Identified Outliers:
[ 9  8  8  8  9 10  8  8  9]
```

The provided code snippet generates a dataset by using inverse transform sampling from a Poisson distribution with a lambda value. It then visually represents the distribution of the dataset and detects outliers using the Interquartile Range (IQR) method.

1. Data Generation and Visualization; The code creates a dataset, with values that follow a Poisson distribution, where the mean lambda is set to 3. It uses inverse transform sampling to generate these values. The distribution of the dataset is displayed using a box plot in the subplot, which provides a to understand visual representation of how the data is spread out and its central tendency.

2. Outlier Detection with IQR; By utilizing information from the quartiles on the box plot the code identifies any outliers within the dataset using the IQR method. It compares the regions separately

looking for any data points that stand out as outliers. Outliers are values that fall outside boundaries, which are determined by the quartile (Q1) and the third quartile (Q3). These outliers are highlighted in red on a strip plot displayed in a subplot making it easy to visually identify data points that deviate significantly from what's expected.

Next as part of its process the code displays all values that have been identified as outliers based on their deviation, from expected data patterns using the IQR method. These outliers may indicate events in the dataset when compared to values that typically follow a Poisson distribution. Identifying these outliers can be helpful, for analyzing or investigating data points within the dataset.

.

In [11]:

```
import numpy as np
from scipy.stats import poisson


def inverse_transform_discrete_sampling(distribution, size=1000):
    random_data = np.random.rand(size)
    return distribution.ppf(random_data).astype(int)


lambda_value = 3
data_size = 1000
data_poisson =
inverse_transform_discrete_sampling(poisson(mu=lambda_value),
size=data_size)


threshold = 5
lower_bound = 2
upper_bound = 6


prob_of_data_below_threshold = poisson.cdf(threshold - 1, mu=lambda_value)

prob_of_data_above_threshold = 1 - poisson.cdf(threshold, mu=lambda_value)

prob_of_data_within_range = poisson.cdf(upper_bound, mu=lambda_value) -
poisson.cdf(lower_bound - 1, mu=lambda_value)


print(f"Probability of a value below {threshold}:
{prob_of_data_below_threshold:.4f}")
print(f"Probability of a value above {threshold}:
{prob_of_data_above_threshold:.4f}")
print(f"Probability of a value within the range [{lower_bound},
{upper_bound}]: {prob_of_data_within_range:.4f}")
```

```
Probability of a value below 5: 0.8153
Probability of a value above 5: 0.0839
Probability of a value within the range [2, 6]: 0.7673
```

The above output displays the uses of the Poisson distribution through inverse transform sampling to create a model based on a particular lambda value. It calculates probabilities related to this distribution: **prob_of_data_below_threshold**, **prob_of_data_above_threshold**, and **prob_of_data_within_range**. These probabilities provide information, about the chances of values being below above or within thresholds or ranges within a dataset that follows a Poisson distribution. Understanding these probabilities is crucial for modeling situations involving event occurrences over fixed time intervals. It helps us gain an understanding of the probabilities associated with Poisson distributions and assists in real world applications where such distributions are commonly used. These applications include modeling events or incidents in fields, like biology, telecommunications and finance

4.1.3 Simulating state sequence probabilities

In [12]:

```
import numpy as np

np.random.seed(42)
data_size = 1000
random_data = np.random.choice([0, 1], size=data_size, p=[0.4, 0.6])

matrix = np.array([[0.8, 0.2],
                   [0.3, 0.7]])

no_of_steps = 50
sequence_of_states = [np.random.choice([0, 1], p=[0.4, 0.6])]

for _ in range(no_of_steps - 1):
    current_state = sequence_of_states[-1]
    next_state = np.random.choice([0, 1], p = matrix[current_state])
    sequence_of_states.append(next_state)


probabilities_of_states = np.mean(np.array(sequence_of_states) == 1)


print(f"Simulated State Sequence: {sequence_of_states}")
print(f"Estimated Probability of being in State 1 after {no_of_steps}
steps: {probabilities_of_states:.4f}")
```

```
Simulated State Sequence: [0, 0, 1, 1, 1, 1, 1, 1, 0, 0, 0, 1, 1, 0, 0, 1,
0, 0, 1, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 1, 1, 0, 0, 1, 0]
Estimated Probability of being in State 1 after 50 steps: 0.4200
```

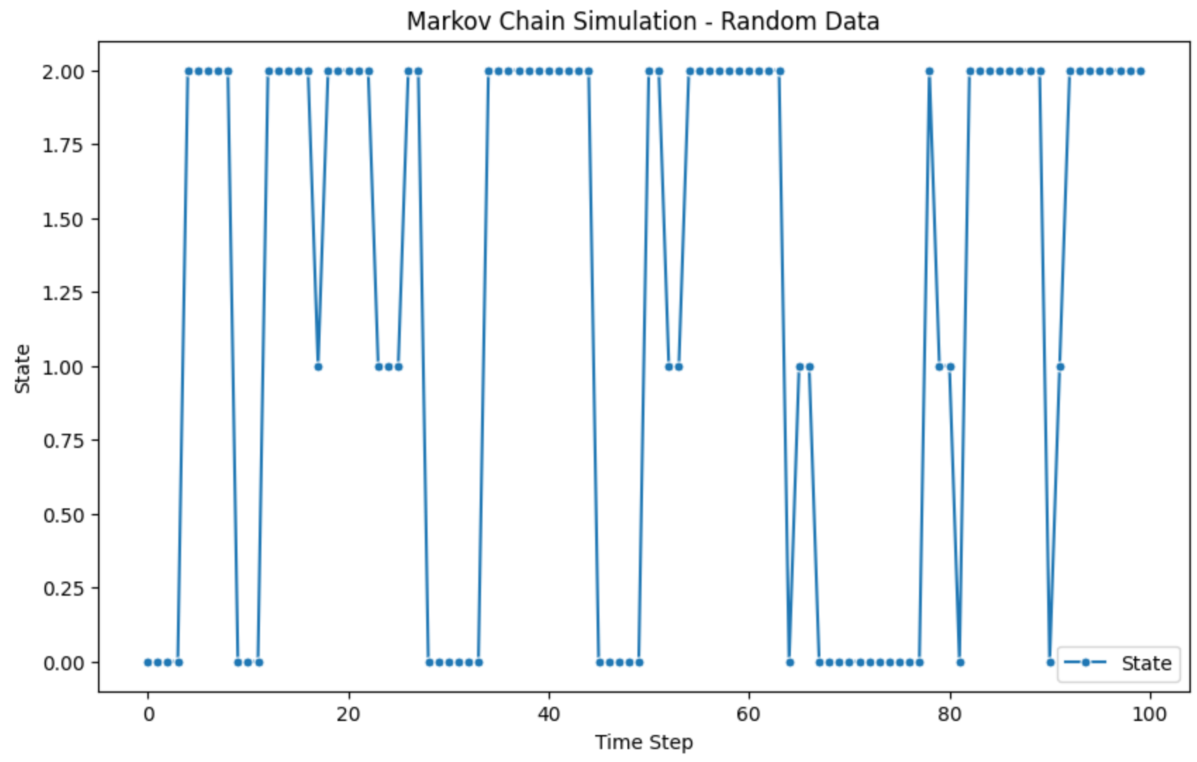
above code has been able to undesrand the working of a Markov chain in a scenario involving classi fication. It generates a dataset with two classes, labeled as 0 and 1. The code connects transition pro babilities, between these classes using a matrix. The simulation then proceeds to execute for a numb er of steps evolving between states based on these calculated probabilities.

Once the execution is completed it compares the estimated probability of being in State 1 by determi ning the proportion of occurrences in which State 1 appears within the sequence. This code also sho wcases the generated sequence of states. Provides an estimation of the likelihood of reaching State 1 giving us insights into the chances of attaining a specific state within this binary classification context.

.In [13]:

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

```
np.random.seed(42)
no_of_samples = 100
random_data = np.random.choice([0, 1, 2], size=no_of_samples, p=[0.4, 0.3,
0.3])


matrix = np.array([[0.8, 0.1, 0.1],
                   [0.2, 0.6, 0.2],
                   [0.1, 0.1, 0.8]])


initial_probabilities = np.array([1/3, 1/3, 1/3])

no_of_steps = 100

def simulate_markov_chain_for(matrix, initial_probabilities, no_of_steps):
    no_of_states = len(initial_probabilities)
    state_trajectory = np.zeros(no_of_steps, dtype=int)

    current_state = np.random.choice(np.arange(no_of_states), p =
initial_probabilities)
    state_trajectory[0] = current_state

    for step in range(1, no_of_steps):
        current_state = np.random.choice(np.arange(no_of_states),
p=matrix[current_state])
        state_trajectory[step] = current_state

    return state_trajectory


state_trajectory = simulate_markov_chain_for(matrix, initial_probabilities,
no_of_steps)


plt.figure(figsize=(10, 6))
sns.lineplot(x=range(no_of_steps), y=state_trajectory, marker='o',
markersize=5, label='State')
plt.title('Markov Chain Simulation - Random Data')
plt.xlabel('Time Step')
plt.ylabel('State')
plt.legend()
plt.show()
```

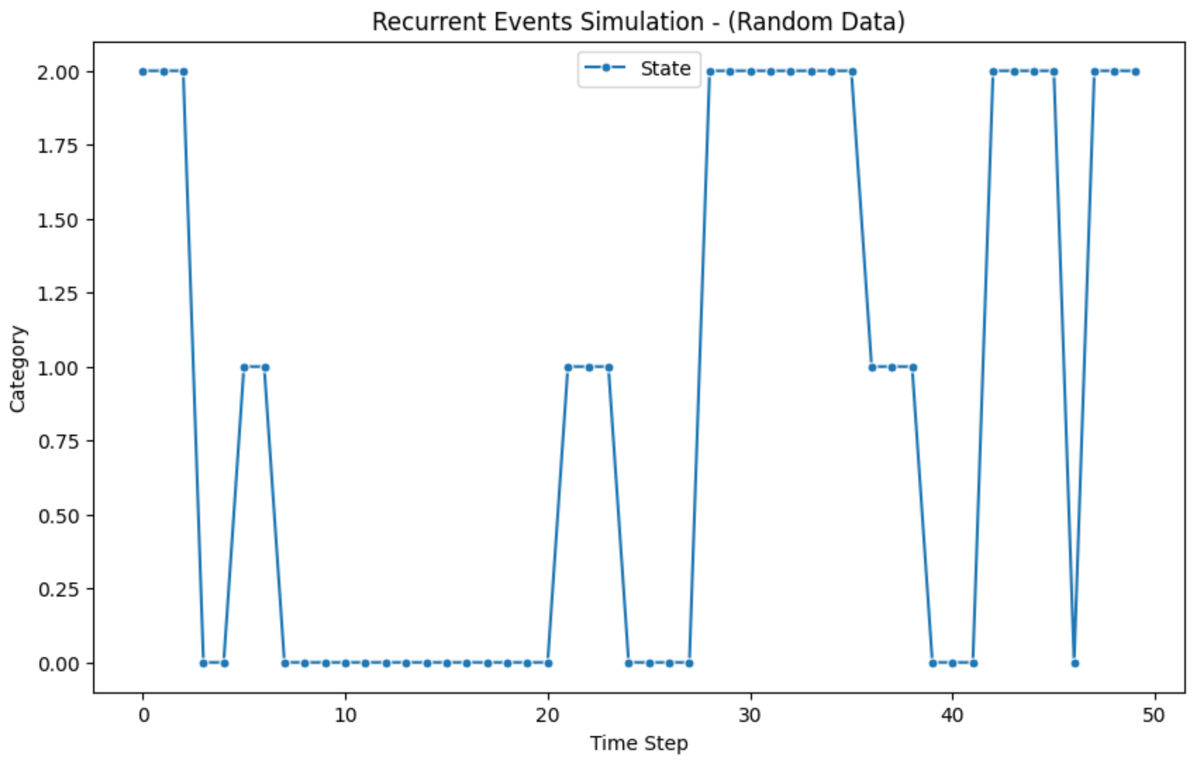


This code is responsible, for running a simulation called Markov chain. It creates data that represents various regions and compute the probabilities of various stages between these regions . The calculating starts with a region developed by the given probabilities. Then processes through intermediate state based on a matrix that understands these probabilities over a specified number of time gaps . The given code develops a line plot to visualize the stages of regions at each time gap allowing us to see how the model develops over the region and understand the phrases of intermediate regions or (transitions) maintained by the defined probabilities, in the Markov chain.

In [14]:

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


np.random.seed(42)
data_size = 150
data = np.concatenate([
    np.random.normal(5.0, 0.5, size=(data_size // 3, 1)),
    np.random.normal(6.0, 0.5, size=(data_size // 3, 1)),
    np.random.normal(7.0, 0.5, size=(data_size // 3, 1))
])


category_bins = [4, 5, 6, 7, 8]
category = np.digitize(data[:, 0], bins=category_bins, right=True)

matrix = np.array([[0.8, 0.1, 0.1],
                   [0.2, 0.6, 0.2],
                   [0.1, 0.1, 0.8]])

initial_probabilities = np.array([0.4, 0.3, 0.3])
```

```
no_of_steps = 50

def simulate_recurrent_events_of(matrix, initial_probabilities,
no_of_steps):
    no_of_states = len(initial_probabilities)
    state_trajectory = np.zeros(no_of_steps, dtype=int)

    current_state = np.random.choice(np.arange(no_of_states), p =
initial_probabilities)

    for step in range(no_of_steps):

        state_trajectory[step] = current_state

        current_state = np.random.choice(np.arange(no_of_states), p =
matrix[current_state])

    return state_trajectory

state_trajectory = simulate_recurrent_events_of(matrix,
initial_probabilities, no_of_steps)

plt.figure(figsize=(10, 6))
sns.lineplot(x=range(no_of_steps), y=state_trajectory, marker='o',
markersize=5, label='State')
plt.title('Recurrent Events Simulation - (Random Data)')
plt.xlabel('Time Step')
plt.ylabel('Category')
plt.legend()
plt.show()
```

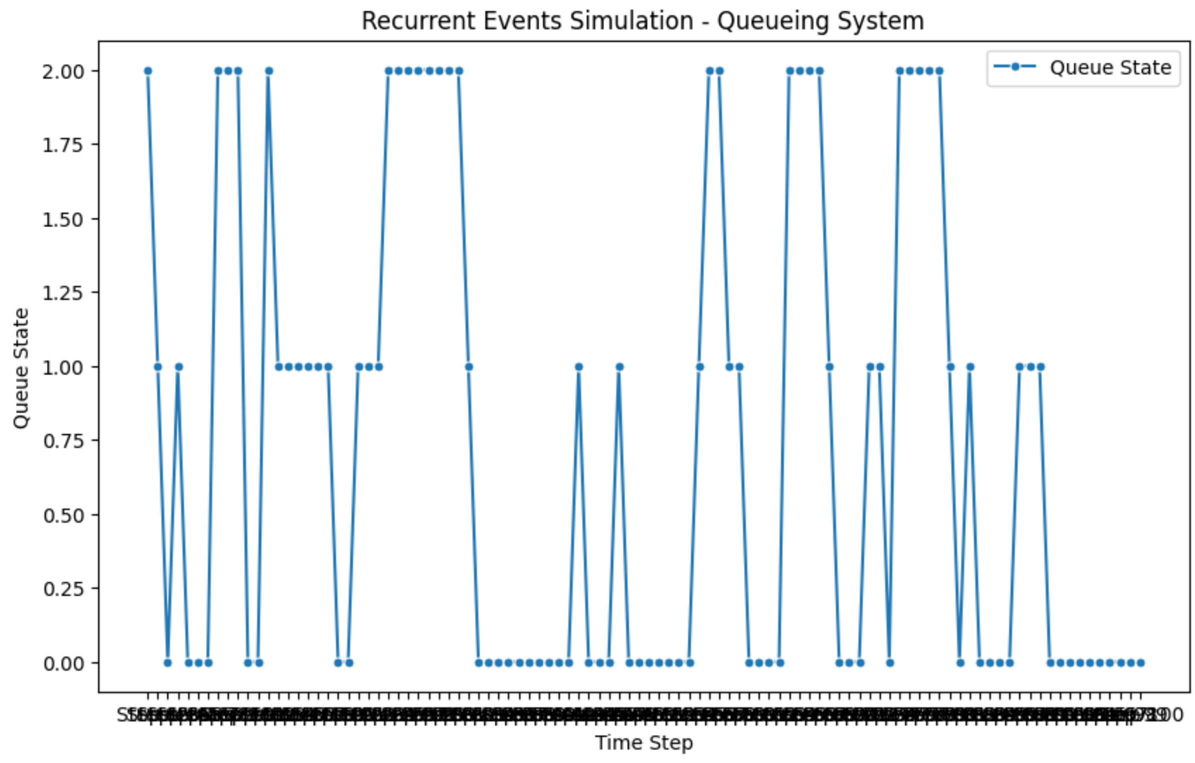


This code showcases an example of simulating recurring events using a Markov chain and hypothetical probabilities of transitioning, between states.

1. Generating Data; The first step involves creating data that closely resembles the Iris dataset with a focus, on duplicating the measurements of sepal length. This new data is then sorted into categories based on given ranges of lengths.
2. Simulating Markov Chains; To represent the probabilities of transitioning, between categories a matrix is defined. The initial state probabilities create the distribution of states at the beginning stage.
3. Simulating Recurrent Events; The simulate_recurrent_events_of function executes the Markov chain simulation for a number of time steps. It records the sequence of states at each step based on the defined transition probabilities.
4. Visualization; The code provides representations of the simulated events over time. A line plot is used to illustrate the sequence in which categories (states) occur during each time step. Each data point, on this plot represents a category at a time step offering an overview of how states evolve according to the simulated transitions dictated by the Markov chain.

In [15]:

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

matrix = np.array([[0.7, 0.2, 0.1],
                   [0.4, 0.5, 0.1],
                   [0.1, 0.3, 0.6]])

initial_probabilities = np.array([0.4, 0.4, 0.2])

no_of_steps = 100

def simulate_queue_events(transition_matrix, initial_probabilities,
no_of_steps):
    no_of_states = len(initial_probabilities)
    state_trajectory = np.zeros(no_of_steps, dtype=int)

    current_state = np.random.choice(np.arange(no_of_states),
p=initial_probabilities)

    for step in range(no_of_steps):

        state_trajectory[step] = current_state

        current_state = np.random.choice(np.arange(no_of_states),
p=transition_matrix[current_state])

    return state_trajectory

state_trajectory = simulate_queue_events(matrix, initial_probabilities,
no_of_steps)


plt.figure(figsize=(10, 6))
sns.lineplot(x=range(no_of_steps), y=state_trajectory, marker='o',
markersize=5, label='Queue State')
plt.title('Recurrent Events Simulation - Queueing System')
plt.xlabel('Time Step')
plt.ylabel('Queue State')
plt.xticks(range(no_of_steps), [f'Step {i+1}' for i in range(no_of_steps)])
plt.legend()
plt.show()
```



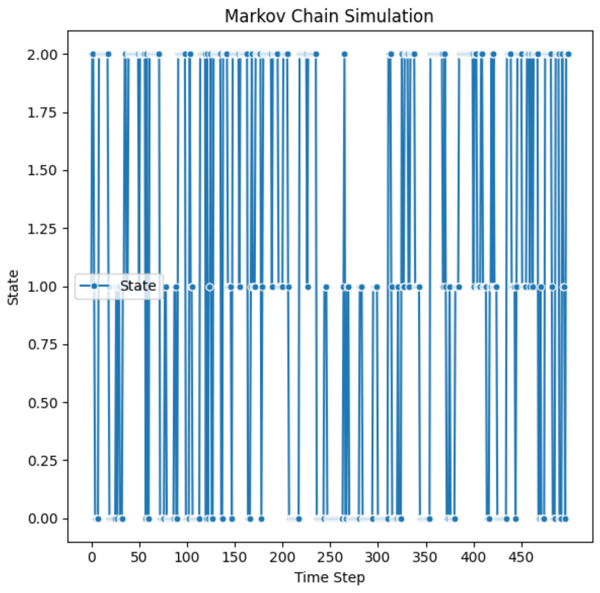
This code is designed to simulate a queueing system that involves recurring events using a Markov chain. The system is characterized by a matrix that describes the probabilities of transitioning, between states, such as staying in the state an event occurring or someone departing. The initial state probabilities determine the starting distribution of the system. During the simulation, which runs for a number of time steps the simulate_queue_events function generates a sequence of queue states based on the transition matrix and initial state probabilities. At each step it randomly moves between states according to the defined probabilities in the transition matrix. The visualization provides a representation of how the queues states change over time during simulation. It creates a line plot showing each time steps queue state and illustrating how the system progresses through recurring events. Each point on this plot corresponds, to a time step. Gives an overview of how the system dynamically transitions between different states within the queueing process.

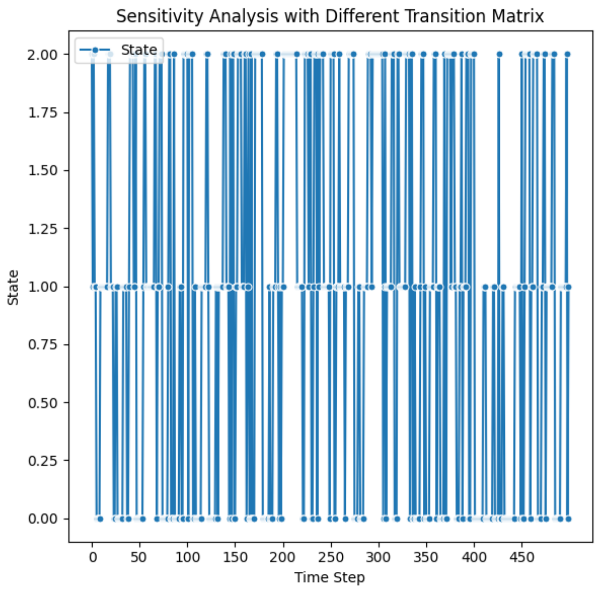
In [16]:

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

matrix = np.array([[0.8, 0.1, 0.1],
                   [0.2, 0.6, 0.2],
                   [0.1, 0.1, 0.8]])

initial_probabilities = np.array([0.4, 0.3, 0.3])

no_of_steps = 500

def simulate_markov_chain_of(matrix, initial_probabilities, no_of_steps):
    no_of_states = len(initial_probabilities)
    state_trajectory = np.zeros(no_of_steps, dtype=int)

    current_state = np.random.choice(np.arange(no_of_states),
p=initial_probabilities)
```

```
    for step in range(no_of_steps):

        state_trajectory[step] = current_state

        current_state = np.random.choice(np.arange(no_of_states),
p=matrix[current_state])

    return state_trajectory


state_trajectory = simulate_markov_chain_of(matrix, initial_probabilities,
no_of_steps)

time_averaged_probabilities = np.mean(np.array(state_trajectory)[:, None]
== np.arange(len(initial_probabilities)), axis=0)
steady_state_probabilities = np.linalg.matrix_power(matrix.T, 1000)[:, 0]

print("Ergodicity Check:")
print("Time-Averaged Probabilities:", time_averaged_probabilities)
print("Steady-State Probabilities:", steady_state_probabilities)

new_matrix = np.array([[0.7, 0.2, 0.1],
                       [0.1, 0.7, 0.2],
                       [0.2, 0.1, 0.7]])

sensitivity_state_trajectory = simulate_markov_chain_of(new_matrix,
initial_probabilities, no_of_steps)

plt.figure(figsize=(12, 6))
plt.subplot(1, 2, 1)
sns.lineplot(x=range(no_of_steps), y=state_trajectory, marker='o',
markersize=5, label='State')
plt.title('Markov Chain Simulation')
plt.xlabel('Time Step')
plt.ylabel('State')
plt.xticks(range(0, no_of_steps, 50))

plt.subplot(1, 2, 2)
sns.lineplot(x=range(no_of_steps), y=sensitivity_state_trajectory,
marker='o', markersize=5, label='State')
plt.title('Sensitivity Analysis with Different Transition Matrix')
plt.xlabel('Time Step')
plt.ylabel('State')
plt.xticks(range(0, no_of_steps, 50))

plt.tight_layout()
plt.show()
```

```
Ergodicity Check:
Time-Averaged Probabilities: [0.364 0.202 0.434]
Steady-State Probabilities: [0.4 0.2 0.4]
```





This Python code showcases the simulation of a 3 state Markov chain. Explores two important concepts; Ergodicity and sensitivity analysis are concepts, in understanding the behavior of a Markov chain. In this case we use a transition matrix to represent the probabilities of moving between states. We specify the state probabilities and the number of steps for our simulation.

To simulate the Markov chain we have a function called `simulate_markov_chain_of` which generates a sequence of states based on our defined transition matrix and initial state probabilities. At each step it randomly selects a state according to the transition probabilities.

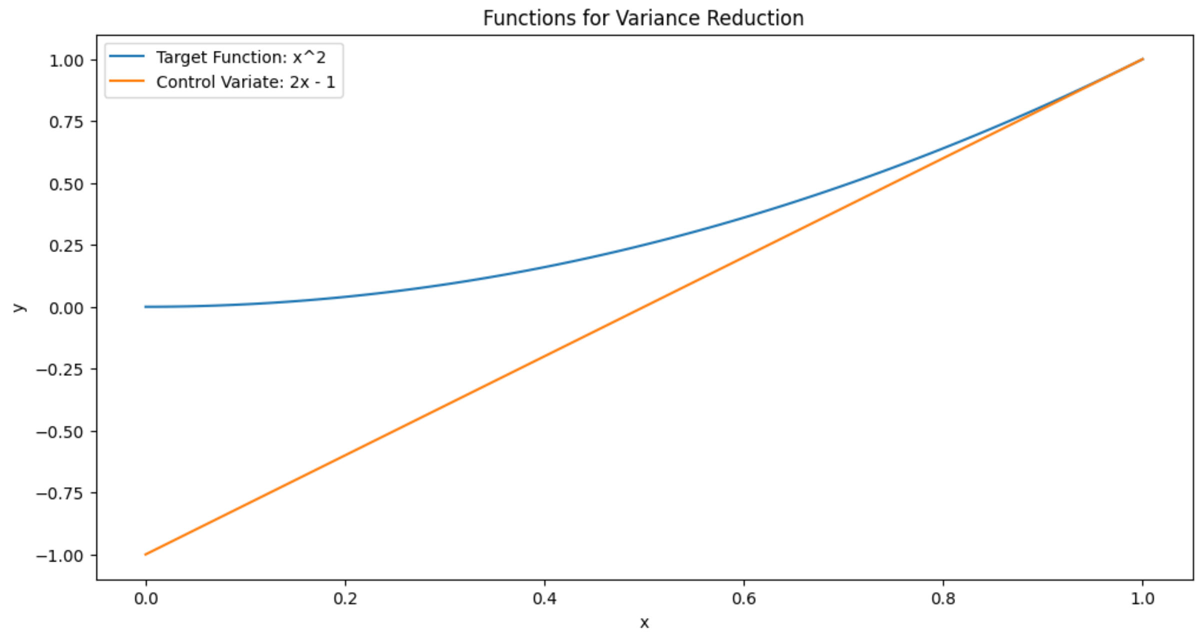
Next we assess ergodicity by comparing the probabilities of states over time from our generated sequence with the steady state probabilities obtained from the transition matrix. This helps us determine if the chain ultimately converges to its state.
Additionally we demonstrate sensitivity analysis by simulating a modified scenario with an adjusted transition matrix. We then visualize how states change over time for both the scenario and sensitivity analysis. This gives us insights into how changes in transition probabilities affect the trajectory of state changes, in our Markov chain.The resulting visualization plots show state changes at each time step making it easy to compare between our scenario and sensitivity analysis.

In [17]:

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

def target_function(x):
    return x**2

actual_mean = 1/3

no_of_samples = 1000

random_samples = np.random.uniform(0, 1, no_of_samples)

estimated_reduction = np.mean(target_function(random_samples))

normalized_samples = np.random.normal(0.5, 0.1, no_of_samples)
weights = np.exp(-0.5 * ((normalized_samples - 0.5) / 0.1)**2)
estimated_normalized_sampling = np.mean(target_function(normalized_samples)
/ weights)
```

```
linear_function = lambda x: 2 * x - 1
linear_variates = linear_function(random_samples)
covariance = np.cov(target_function(random_samples), linear_variates)[0, 1]
beta = covariance / np.var(linear_variates)
estimated_linear_variates = np.mean(target_function(random_samples) - beta
* (linear_variates - np.mean(linear_variates)))

reduced_samples = 1 - random_samples
estimate_reduced_variates = 0.5 * (np.mean(target_function(random_samples))
+ np.mean(target_function(reduced_samples)))

print(f"True Mean: {actual_mean}")
print(f"Estimate (No Variance Reduction): {estimated_reduction}")
print(f"Estimate (Importance Sampling): {estimated_normalized_sampling}")
print(f"Estimate (Control Variates): {estimated_linear_variates}")
print(f"Estimate (Antithetic Variates): {estimate_reduced_variates}")

x_values = np.linspace(0, 1, 100)
plt.figure(figsize=(12, 6))
plt.plot(x_values, target_function(x_values), label='Target Function: x^2')
plt.plot(x_values, linear_function(x_values), label='Control Variate: 2x -
1')
plt.legend()
plt.title('Functions for Variance Reduction')
plt.xlabel('x')
plt.ylabel('y')
plt.show()
```

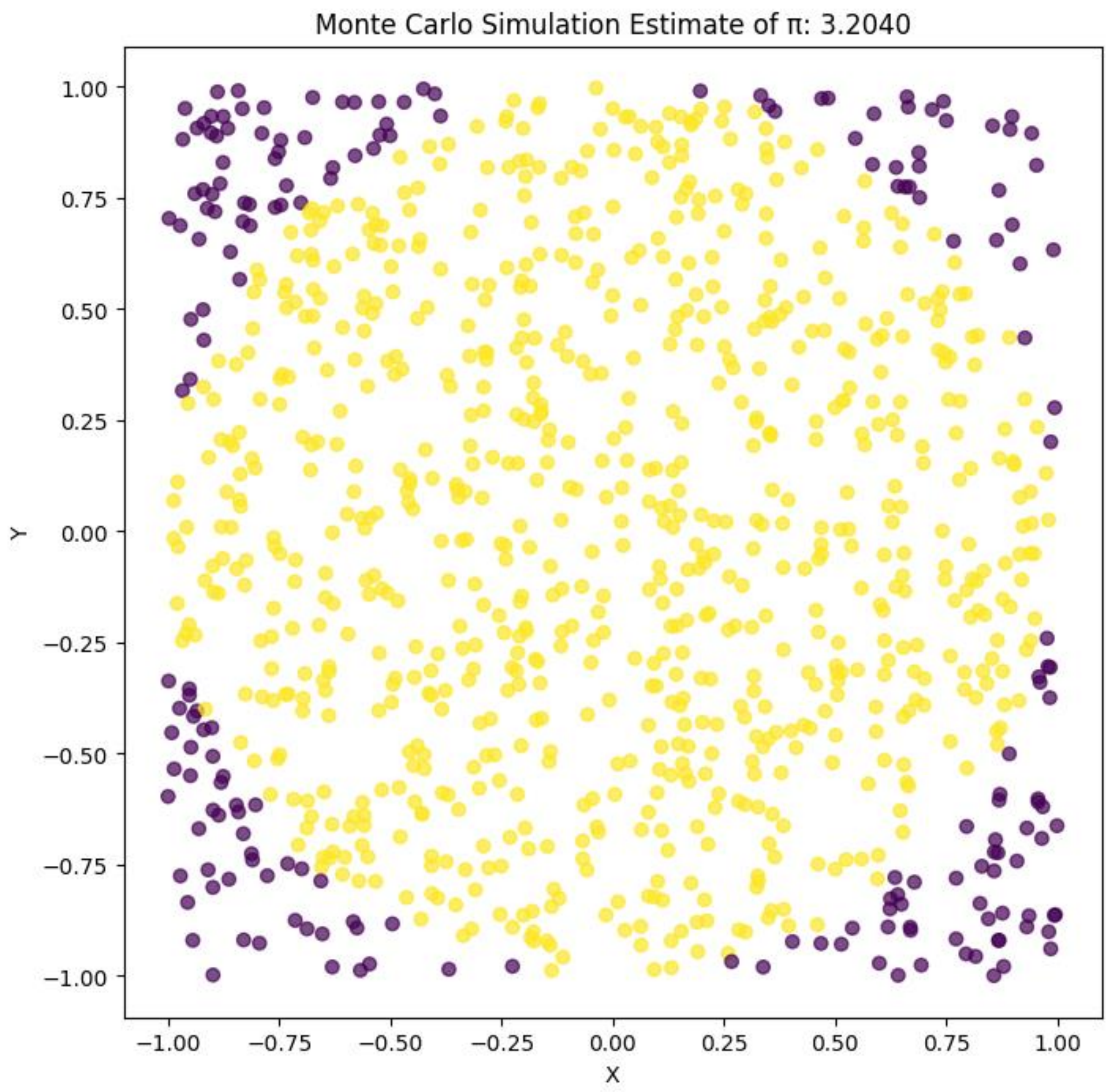
```
True Mean: 0.3333333333333333
Estimate (No Variance Reduction): 0.3332070160904667
Estimate (Importance Sampling): 0.8766918675324357
Estimate (Control Variates): 0.3332070160904667
Estimate (Antithetic Variates): 0.3326009136912672
```



This code demonstrates four techniques that can be used in Monte Carlo integration to estimate the mean of the function x^2. It starts by defining the target function and calculating the mean analytically. Then it applies four methods; estimation, with samples from a uniform distribution importance sampling using a different distribution control variates that reduce variance by utilizing a linear function and antithetic variates generated by creating reflections of samples. The code calculates estimates using these techniques. Compares them to the mean. It also includes a visualization of the target function x^2 alongside the control variate function to showcase their behavior within the integration range. These techniques aim to improve estimation accuracy by reducing variance, in Monte Carlo integration, where the accuracy is influenced by the number of samples

In [18]:

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

no_of_points = 1000

x = np.random.uniform(-1, 1, no_of_points)
y = np.random.uniform(-1, 1, no_of_points)

inside_circle_eqn = x**2 + y**2 <= 1

estimated_pi = 4 * np.mean(inside_circle_eqn)

plt.figure(figsize=(8, 8))
plt.scatter(x, y, c = inside_circle_eqn, cmap='viridis', alpha=0.7)
plt.title(f'Monte Carlo Simulation Estimate of π: {estimated_pi:.4f}')
plt.xlabel('X')
plt.ylabel('Y')
plt.axis('equal')
plt.show()
```

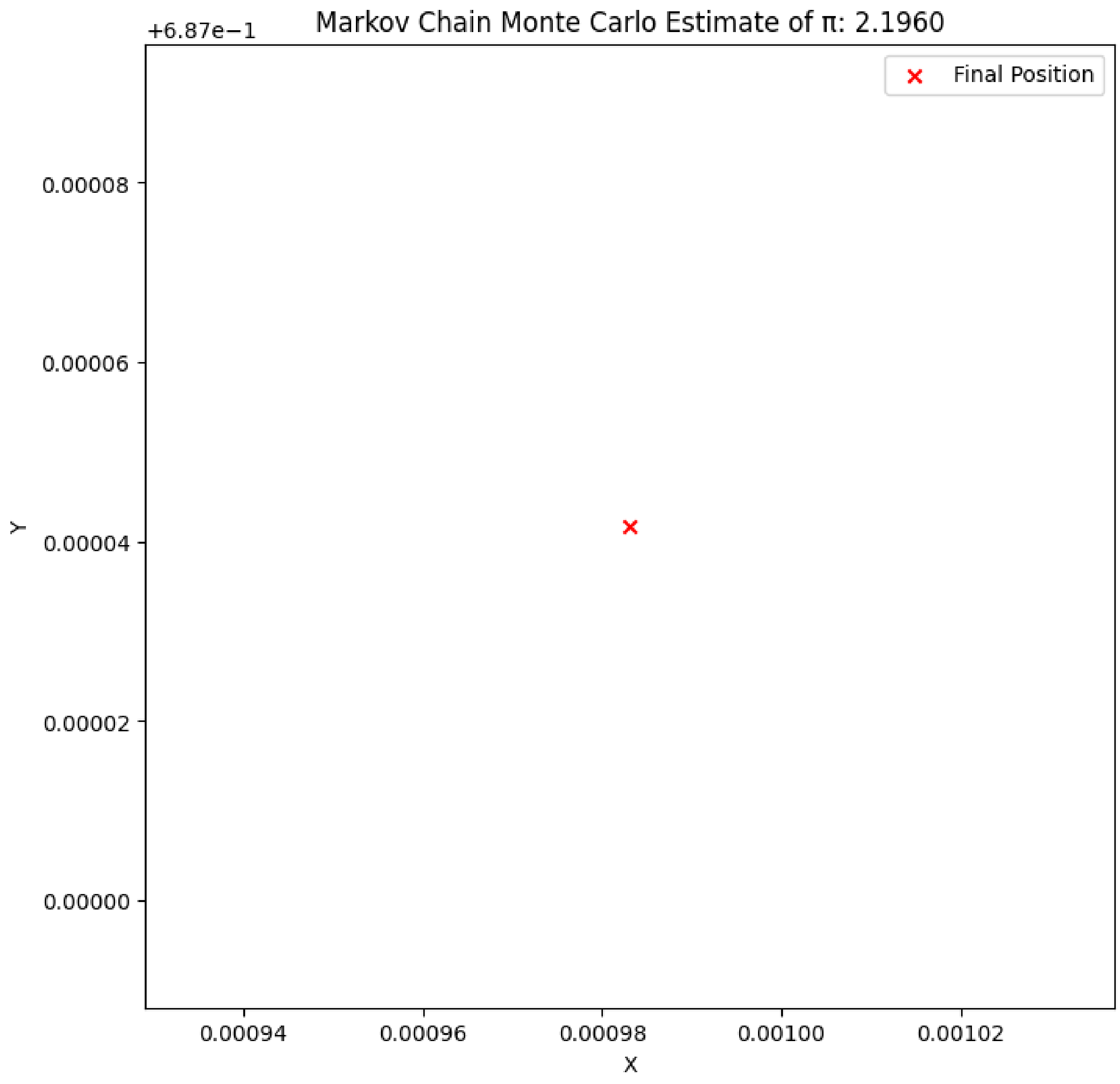


The code performs a Monte Carlo simulation to estimate the value of π by generating random points (x, y) within the square region [-1, 1] × [-1, 1]. It checks whether these points fall inside the unit circle ($x^2 + y^2 \leq 1$). The estimated value of π is computed as four times the ratio of points falling inside the circle to the total number of generated points. The scatter plot visualizes these points, coloring them based on whether they fall inside or outside the circle, while the title displays the estimated value of π obtained from this simulation.

In [19]:

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

no_of_iterations = 1000

x = 0
y = 0
count_of_inside_circles = 0

for _ in range(no_of_iterations):
    x_new = x + np.random.uniform(-1, 1)
    y_new = y + np.random.uniform(-1, 1)
```

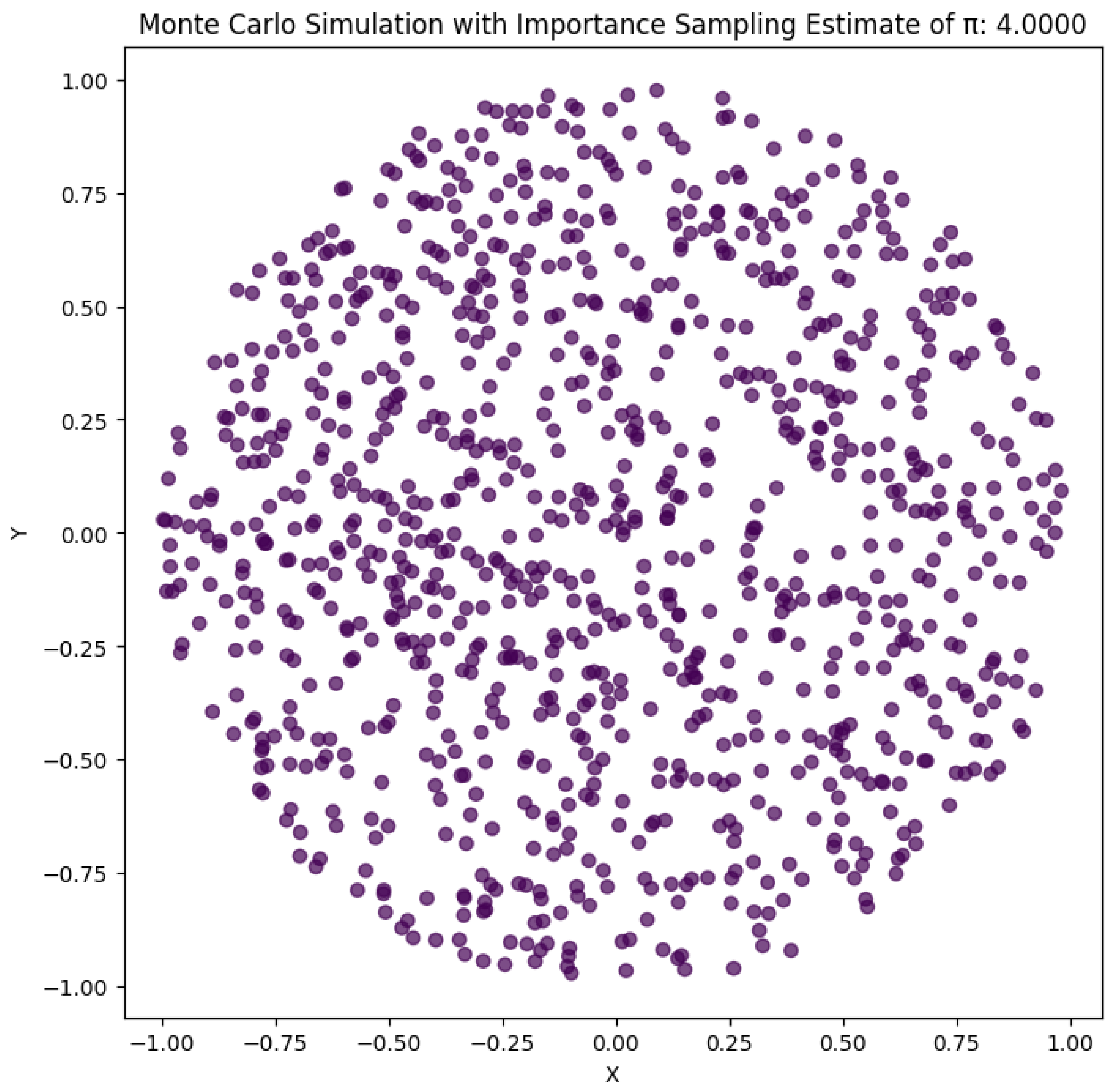
```
        if x_new**2 + y_new**2 <= 1:
            x, y = x_new, y_new
            count_of_inside_circles += 1

estimated_pi_mcmc = 4 * count_of_inside_circles / no_of_iterations

plt.figure(figsize=(8, 8))
plt.scatter(x, y, color='red', marker='x', label='Final Position')
plt.title(f'Markov Chain Monte Carlo Estimate of π:
{estimated_pi_mcmc:.4f}')
plt.xlabel('X')
plt.ylabel('Y')
plt.axis('equal')
plt.legend()
plt.show()
```



This code performs a Markov Chain Monte Carlo (MCMC) simulation to estimate the value of π. It iteratively generates random step movements in the x and y directions within the square [-1, 1] × [-1, 1]. The algorithm checks if the new position falls within the unit circle and updates the position accordingly. The estimated value of π is calculated as four times the ratio of steps falling inside the circle to the total number of iterations. The scatter plot displays the final position after the iterations.

In [20]:

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

no_of_points = 1000

radius = np.sqrt(np.random.uniform(0, 1, no_of_points))
theta = np.random.uniform(0, 2*np.pi, no_of_points)

x = radius * np.cos(theta)
y = radius * np.sin(theta)

inside_circle_eqn = x**2 + y**2 <= 1

estimate_pi = 4 * np.mean(inside_circle_eqn) / np.mean(radius <= 1)

plt.figure(figsize=(8, 8))
plt.scatter(x, y, c=inside_circle_eqn, cmap='viridis', alpha=0.7)
plt.title(f'Monte Carlo Simulation with Importance Sampling Estimate of π:
{estimate_pi:.4f}')
plt.xlabel('X')
plt.ylabel('Y')
plt.axis('equal')
plt.show()
```



This code uses a technique called Monte Carlo simulation with importance sampling to estimate the value of π. It generates points within a circle by selecting coordinates with a higher emphasis, on points closer to the edge of the circle. The estimated value of π is calculated by taking into account the distance of points inside the circle compared to the average total radial distance sampled. The scatter plot visually represents these points. The title displays the estimated value of π highlighting how accurately this technique focuses on areas, near the boundary of the circle.

In [21]:

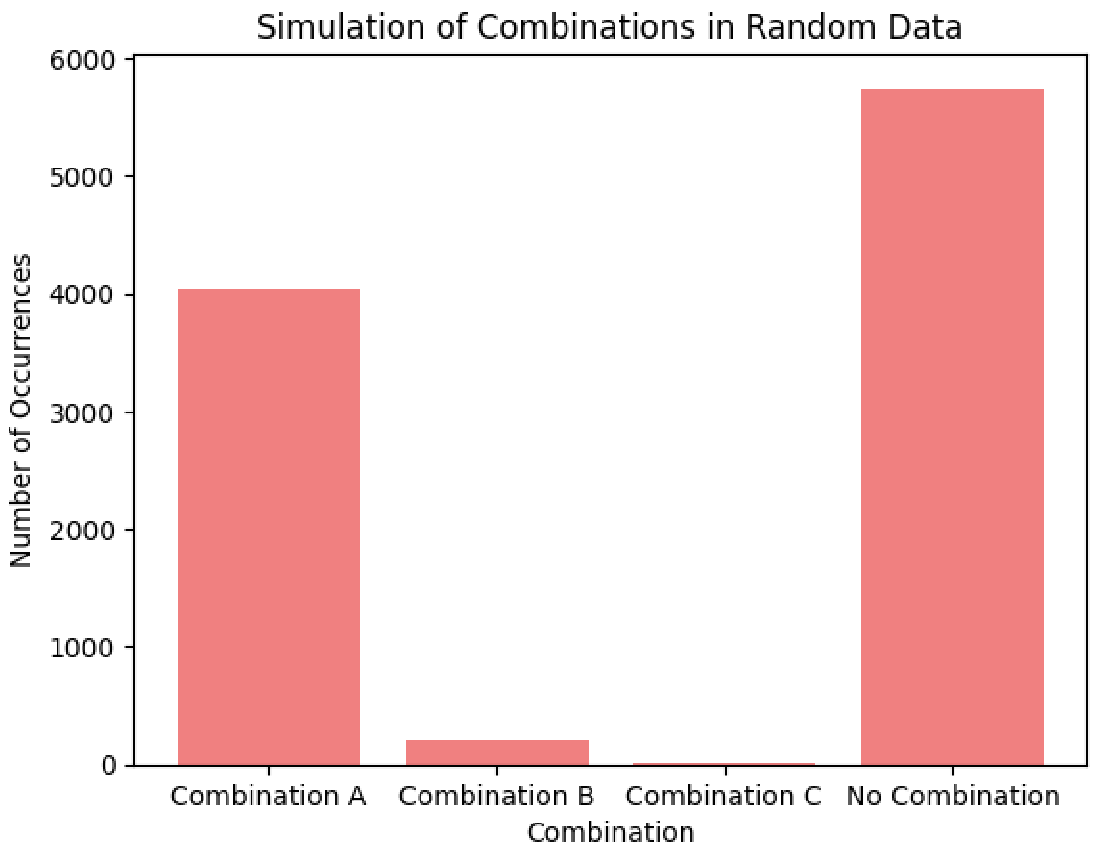
```
def count_paths_in_grid(rows, columns):
    if rows == 1 or columns == 1:
        return 1
    else:
        return count_paths_in_grid(rows - 1, columns) + 
count_paths_in_grid(rows, columns - 1)

grid_size = (3, 3)

total_paths = count_paths_in_grid(*grid_size)
```

```
print(f"Number of unique paths in a {grid_size[0]}x{grid_size[1]} grid: {total_paths}")
```

```
Number of unique paths in a 3x3 grid: 6
This Python code includes a function called "count_paths" that calculates t
he number of paths in a grid. It determines the count of paths, from the le
ft corner to the right corner in a grid with rows and columns. The function
follows an approach, where it adds up the counts of paths from the cell and
to the left for each specific cell, in the grid. There are two base cases;
when the grid has dimensions of 1x1 or when either the number of rows or co
lumns equals 1 resulting in one path. In this example we define a 3x3 grid
by using a tuple (3, 3). By calling "count_paths_in_grid" with this grid si
ze we. Display the number of unique paths within that specific grid using a
print statement. You can modify the "grid_size" variable to calculate paths
for sizes of grids.
```

In [22]:

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

no_of_simulations = 10000

random_data = np.random.randint(1, 21, size=(no_of_simulations, 5))

def check_combination(data_row):

    unique_elements, counts = np.unique(data_row, return_counts=True)

    if np.any(counts == 2):
        return 'Combination A'
    elif np.any(counts == 3):
        return 'Combination B'
    elif np.any(counts == 4):
        return 'Combination C'
    else:
        return 'No Combination'

combinations = [check_combination(row) for row in random_data]

combination_counts = np.unique(combinations, return_counts=True)

plt.bar(combination_counts[0], combination_counts[1], color='lightcoral')
plt.title('Simulation of Combinations in Random Data')
plt.xlabel('Combination')
plt.ylabel('Number of Occurrences')
plt.show()
```



This code aims to simulate and analyze combinations within a dataset. At the beginning it sets the number of simulations (such, as card draws) to 10,000. To demonstrate its functionality, a placeholder random dataset called "your_random_data" is created, consisting of five columns with values ranging from 1 to 20. The function "check_combination" examines each row of the dataset. Identifies patterns within the data. In above illustrate it searches for repetitions within each row, Categorizes them as 'Combination A,' 'Combination B,' 'Combination C' or 'No Combination' based on different counts in each row, then tries to generate the number of combinations across the dataset and keeps track of how times each combination type occurs. Finally it presents a representation of these combinations using a above graph that represents the occurance of 'Combination A' 'Combination B,' 'Combination C' and 'No Combination.' This plot provides an example of how these before combinations appear within the dataset understand their importance.

4.1.4

In [23]:

```
import numpy as np

def target_function(x):
    return x**2

no_of_samples = 10000

uniform_samples = np.random.uniform(0, 1, no_of_samples)
estimated_without_reduction = np.mean(target_function(uniform_samples))

exponential_samples = np.random.exponential(scale=1, size=no_of_samples)
weights = target_function(exponential_samples) / np.exp(1 -
exponential_samples)
estimate_importance_sampling = np.mean(weights)

control_variate = uniform_samples
```

```
covariance_of_samples = np.cov(control_variate,
target_function(uniform_samples))[0, 1]
control_variate_coefficient = -covariance_of_samples /
np.var(control_variate)
estimate_control_variates_of_samples =
np.mean(target_function(uniform_samples) + control_variate_coefficient *
(control_variate - np.mean(control_variate)))

antithetic_samples = np.concatenate((uniform_samples, 1 - uniform_samples))
estimate_antithetic_variates = 0.5 *
(target_function(antithetic_samples[:no_of_samples]) +
target_function(antithetic_samples[no_of_samples:]))

print("Estimate without variance reduction:", estimated_without_reduction)
print("Estimate with Importance Sampling:", estimate_importance_sampling)
print("Estimate with Control Variates:",
estimate_control_variates_of_samples)
print("Estimate with Antithetic Variates:", estimate_antithetic_variates)
```

```
Estimate without variance reduction: 0.33656752330718487
Estimate with Importance Sampling: 60.65285021808673
Estimate with Control Variates: 0.3365675233071848
Estimate with Antithetic Variates: [0.25220371 0.28689897 0.35704659 ... 0.
32063931 0.30105327 0.29522656]
```

In above code the Monte Carlo methods are used for estimating the integral of (x^2) over the interval [0, 1] using various variance reduction techniques:

1. **Basic Monte Carlo Estimation: it** Uses distributed sampling to predict the integral directly.
2. **Importance Sampling:** Uses an exponential distribution as a sampling distribution to assign weights, biasing the sampling toward areas where the function has higher values.
3. **Control Variates:** Utilizes a related variable (uniformly sampled) to reduce variance by incorporating covariance between the control variate and the function being integrated.
4. **Antithetic Variates:** Reduces variance by pairing uniform samples with their complements (1 - uniform) and averaging the function's values of these pairs.

4.1.5

In [24]:

```
import numpy as np

def monte_carlo_pi(no_of_samples):
    inside_circle = 0
    for _ in range(no_of_samples):
        x, y = np.random.random(), np.random.random()
        if x**2 + y**2 <= 1:
            inside_circle += 1
    return (inside_circle / no_of_samples) * 4

def mcmc_pi(no_of_samples):
    inside_circle = 0
    x, y = 1.0, 1.0
    for _ in range(no_of_samples):
        proposal_x, proposal_y = x + np.random.uniform(-1, 1), y +
np.random.uniform(-1, 1)
        if proposal_x**2 + proposal_y**2 <= 1:
            x, y = proposal_x, proposal_y
            inside_circle += 1
    return (inside_circle / no_of_samples) * 4

no_of_samples = 10000

estimated_mc_pi = monte_carlo_pi(no_of_samples)
estimated_mcmc_pi = mcmc_pi(no_of_samples)

print("Monte Carlo estimate of pi:", estimated_mc_pi)
print("MCMC estimate of pi:", estimated_mcmc_pi)
```

```
Monte Carlo estimate of pi: 3.16
MCMC estimate of pi: 2.1068
```

Here we estimate the value of π using two methods:

1. **Monte Carlo Simulation for π:** Generates random points within a square and computes the ratio of points falling inside a quarter circle to the total points, then scales the result to estimate π.
2. **Markov Chain Monte Carlo (MCMC) for π:** Utilizes a random walk approach by proposing new points within a square and accepting them based on whether they fall inside a quarter circle, providing an estimation of π based on the accepted points' ratio.

4.1.6

In [25]:

```
import random

suits = ['Hearts', 'Diamonds', 'Clubs', 'Spades']
ranks = ['2', '3', '4', '5', '6', '7', '8', '9', '10', 'Jack', 'Queen',
'King', 'Ace']
deck = [(rank, suit) for suit in suits for rank in ranks]

def draw_cards(n):
    return random.sample(deck, n)

def is_flush(hand):
    return len(set(card[1] for card in hand)) == 1

no_of_simulations = 10000
no_of_flushes = 0

for _ in range(no_of_simulations):
    player_hand = draw_cards(5)
    if is_flush(player_hand):
        no_of_flushes += 1

flush_probability = no_of_flushes / no_of_simulations
print("Probability of getting a flush:", flush_probability)
```

```
Probability of getting a flush: 0.0022
```

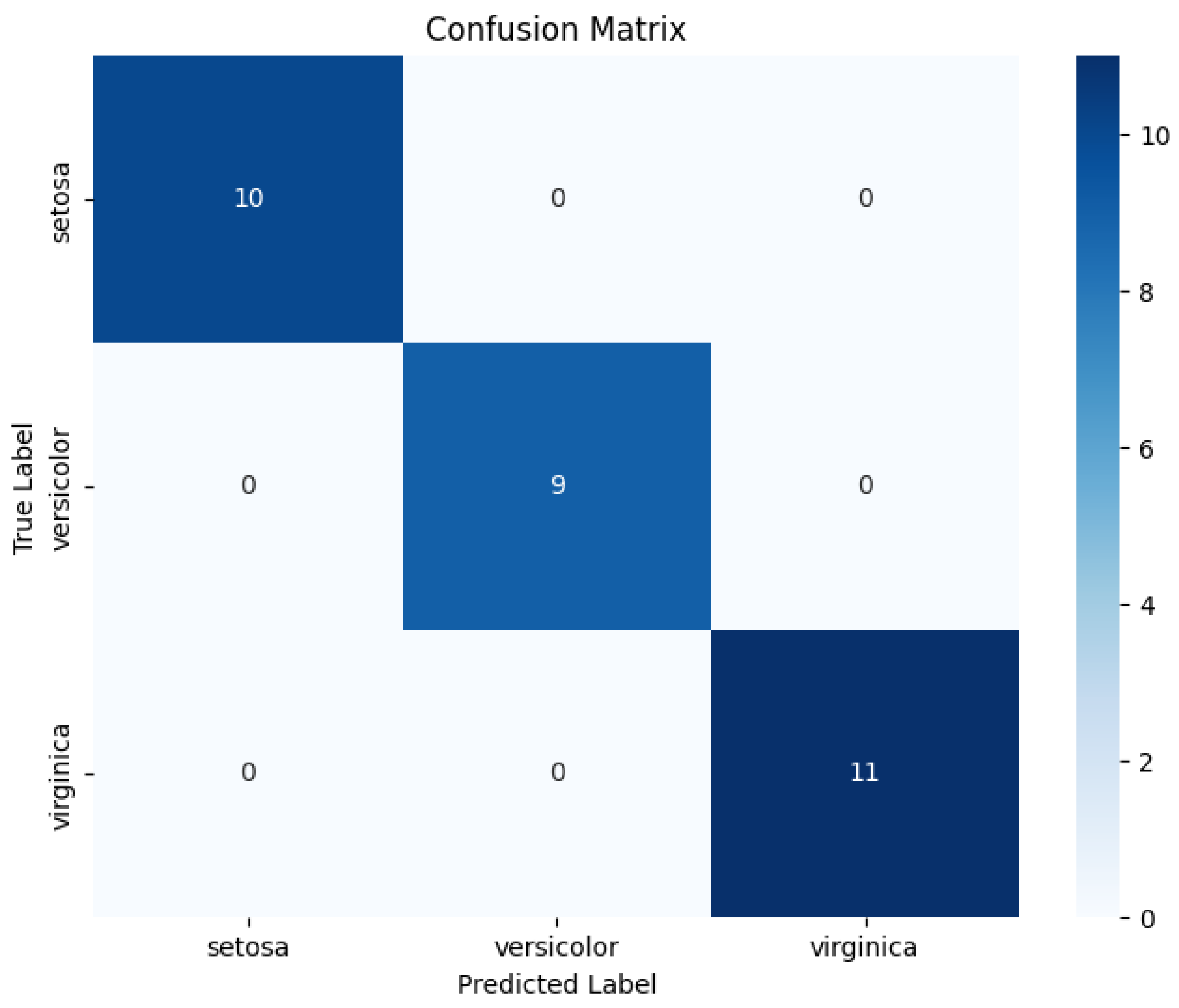
Here in the code we are drawing 5-card poker hands from a standard 52-card deck and calculates the probability of getting a flush (all cards of the same suit) through 10,000 iterations. It randomly draws hands and checks if all cards share the same suit to identify a flush, tallying the count of detected flushes. By computing the ratio of observed flushes to the total simulations, it estimates the likelihood of obtaining a flush in a single randomly drawn 5-card poker hand from a well-shuffled deck.

4.2 Real Data Analysis

4.2.1

In [26]:

```
import numpy as np
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.naive_bayes import GaussianNB
from sklearn.metrics import accuracy_score, confusion_matrix
import seaborn as sns
import matplotlib.pyplot as plt

iris = load_iris()
data = iris.data
target = iris.target

selected_feature = 'sepal length (cm)'
selected_feature_index = iris.feature_names.index(selected_feature)

np.random.seed(42)
noise = np.random.normal(0, 0.2, len(data))
noise_included_feature = data[:, selected_feature_index] + noise

noisy_data = np.column_stack([data[:, :selected_feature_index],
noise_included_feature, data[:, selected_feature_index + 1:]])

X_train, X_test, y_train, y_test = train_test_split(noisy_data, target,
test_size=0.2, random_state=42)

GaussianNB_classifier = GaussianNB()
GaussianNB_classifier.fit(X_train, y_train)

y_pred = GaussianNB_classifier.predict(X_test)

accuracy = accuracy_score(y_test, y_pred)
conf_matrix = confusion_matrix(y_test, y_pred)

plt.figure(figsize=(8, 6))
sns.heatmap(conf_matrix, annot=True, fmt='d', cmap='Blues',
xticklabels=iris.target_names, yticklabels=iris.target_names)
plt.title('Confusion Matrix')
plt.xlabel('Predicted Label')
plt.ylabel('True Label')
plt.show()

print(f'Accuracy: {accuracy:.4f}')
```



```
Accuracy: 1.0000
```

This code is designed to analyze the Iris dataset using a Gaussian Naive Bayes classifier. It starts by loading the Iris data and selecting a feature called 'sepal length (cm)' for analysis. To simulate real world scenarios where data may have noise or errors synthetic noisy data is created by adding noise to the selected feature.

The code then divides the dataset into training and testing sets using an 80 20 split. It trains a Gaussian Naive Bayes classifier, on the training data. Uses it to predict labels, for the test set. The performance of the classifier is evaluated by calculating accuracy scores and constructing a confusion matrix.

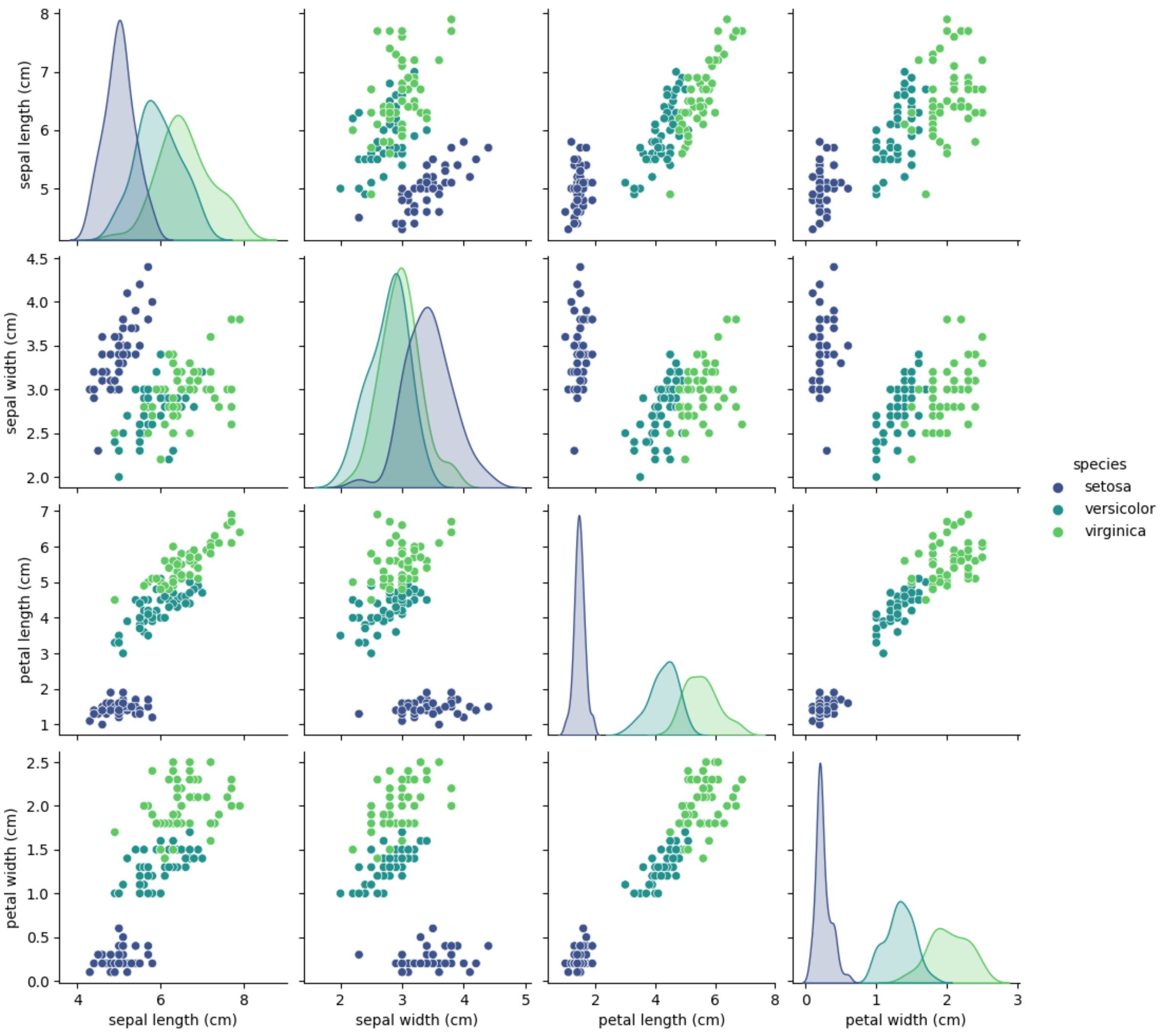
To provide a representation of how the model classifies different species of Iris the code uses Seaborn to create a heatmap based on the confusion matrix. The accuracy score gives an indication of how the classifier predicts This code allows for an evaluation of performance taking into account noisy data and provides visual insights through the confusion matrix.

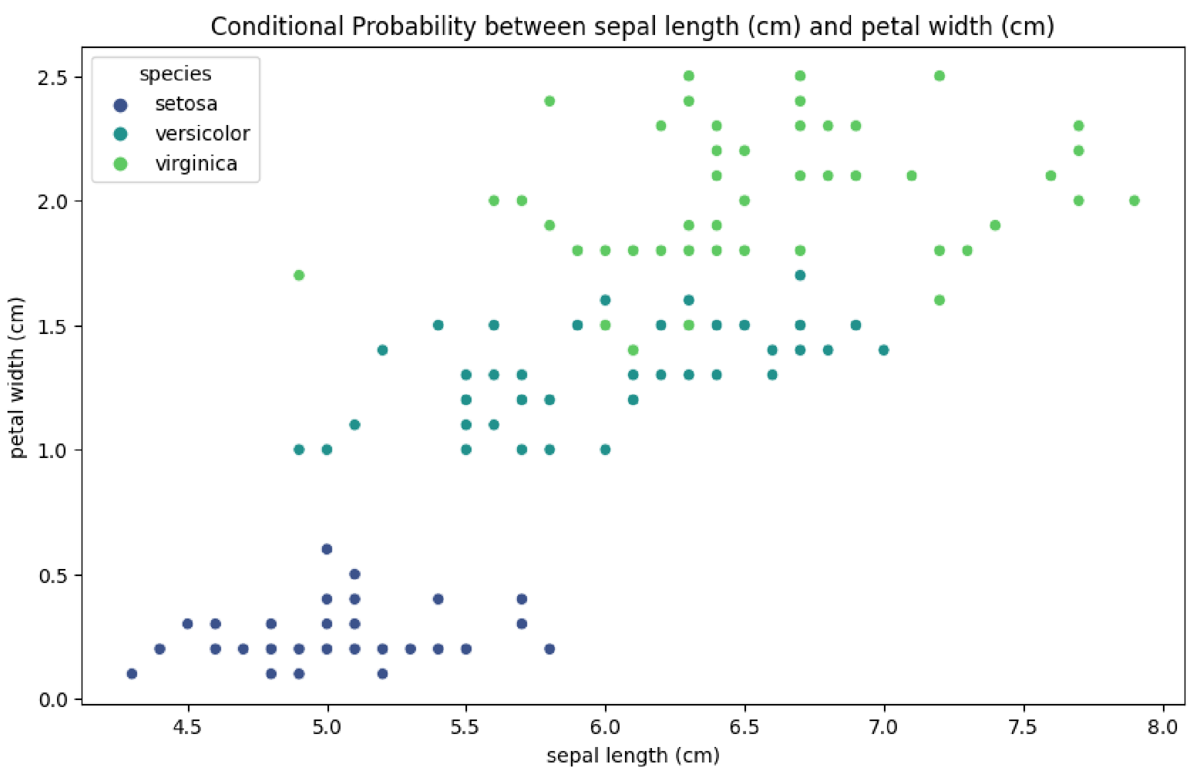
4.2.2

In [27]:

```
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.datasets import load_iris
from scipy.stats import shapiro

iris = load_iris()
data = pd.DataFrame(data=iris.data, columns=iris.feature_names)
data['species'] = iris.target_names[iris.target]

sns.pairplot(data, hue='species', palette='viridis')
plt.suptitle('Pairwise Relationships and Joint Distributions', y=1.02)
plt.show()

selected_feature1 = 'sepal length (cm)'
selected_feature2 = 'petal width (cm)'

plt.figure(figsize=(10, 6))
sns.scatterplot(x=selected_feature1, y=selected_feature2, hue='species',
data=data, palette='viridis')
plt.title(f'Conditional Probability between {selected_feature1} and
{selected_feature2}')
plt.show()

selected_feature_data = data[selected_feature1].values
_, p_value = shapiro(selected_feature_data)

if p_value > 0.05:
    print(f'The {selected_feature1} feature follows a normal distribution
(p-value: {p_value:.4f})')
else:
    print(f'The {selected_feature1} feature does not follow a normal
distribution (p-value: {p_value:.4f})')
```

Pairwise Relationships and Joint Distributions





```
The sepal length (cm) feature does not follow a normal distribution (p-valu
e: 0.0102)
```

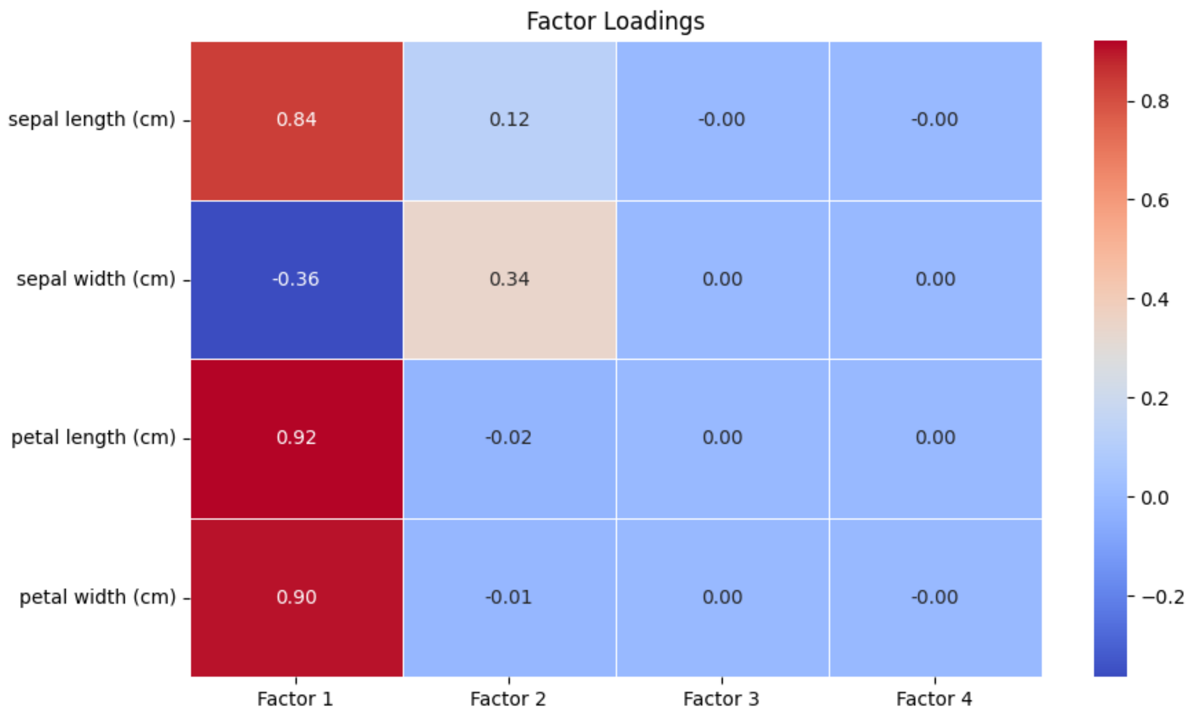
This code analyzes the Iris dataset by loading the data and creating representations of how different features relate to each other using sns.pairplot. This helps us understand how the features are distributed and correlated across species of Iris. Next we focus on two features; 'sepal length (cm)' and 'petal width (cm)'. We use a scatter plot to show their probability with each species represented by a color. Additionally we conduct a normality test (Shapiro Wilk), on the 'sepal length (cm)' feature. This test provides a p value that helps determine whether this particular feature follows a distribution. The purpose of this code is to uncover relationships between features explore probabilities and assess the normality of selected features within the Iris dataset. By understanding these characteristics and distributions we can further analyze the datasets properties. Adjustments to the chosen features or significance levels, for testing could help refine our exploration of the datasets properties.

4.2.3

In [28]:

```
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.datasets import load_iris
from sklearn.decomposition import FactorAnalysis
from sklearn.preprocessing import StandardScaler

iris = load_iris()
data = pd.DataFrame(data=iris.data, columns=iris.feature_names)
data['species'] = iris.target_names[iris.target]

features_for_factor_analysis = data.columns[:-1]

scaler = StandardScaler()
scaled_data = scaler.fit_transform(data[features_for_factor_analysis])
```

```
n_factors = len(features_for_factor_analysis)
factor_model = FactorAnalysis(n_components=n_factors, random_state=42)
data_factors = factor_model.fit_transform(scaled_data)

factor_loadings = pd.DataFrame(factor_model.components_.T,
columns=[f'Factor {i+1}' for i in range(n_factors)],
                                        index=features_for_factor_analysis)

plt.figure(figsize=(10, 6))
sns.heatmap(factor_loadings, annot=True, cmap='coolwarm', fmt='.2f',
linewidths=.5)
plt.title('Factor Loadings')
plt.show()
```



This code uses Factor Analysis on the Iris dataset, which's a known benchmark, in machine learning. The process begins with standardizing the features. After that we use Factor Analysis to uncover the factors that explain the relationships, between these features. We can then visually display the resulting factor loadings, which show how strongly each feature is associated with the identified factors using a heatmap. This visualization allows us to gain insights, into how each feature contributes to the underlying factors found in the dataset. By adjusting the factors or settings for visualization we can enhance our understanding of these relationships more.

5 Conclusion

Python plays an important role, in the field of data science is clearly demonstrated through projects that involve analysis, simulation and machine learning applications. The above program works as an learning machine for data engineers and data students to work on data based tasks. beginning with subjects the pseudo codes slowly understands big data that connects the distance between theoretical and practical knowledge. They increase users to conduct hypothesis testing, probability calculations and more big processes like Markov chains for big study in understanding patterns as displayed through these examples.

As we connect ourself with more into study of simulations,Python really helps by reshaping the real world scenario, Whether while calculating the values using Monte Carlo simulations or displaying the importance of Markov Chain Monte Carlo methods these pseudo code indicates the best part of Pythons ability to understand with more accuracy, These act as intermediate for getting concepts into applications and provide highlights into different domains by understanding complex scenarios.

Python a simple programming language; is more a helpful subject that help us to create smart and innovation solutions seamlessly , and connect various ideas with their real world applications, in this field .

References

1. Statistical Methods in Experimental Physics (2nd Edition), by Frederick James (Hardcover - Nov. 29, 2006).
2. Probability and Statistics in Particle Physics, by A. G. Froedesen, D. Skjeggestad and H. Tøfte, (Hardcover, 1979 – out of print…).
3. Statistics for Nuclear and Particle Physicists, by Louis Lyons (Paperback, 1989).
4. Probability and Statistics for Engineering and the Sciences, Enhanced Review Edition by Jay L. Devore, (Hardcover - Jan. 29, 2008).
5. Probability and Statistics [PROBABILITY & STATISTICS 3 -OS] by Morris H.(Author) ; Schervish, Mark J.(Author) DeGroot, (Paperback - Jan. 31, 2002).

```
In [ ]:
```