Mathematics of Time Series Forecasting

Build Robust Time Series Forecasting Systems in Python Using Mathematical Theory, Statistical Modeling, Machine Learning, and Deep Learning

Copyright © 2026 Orange Education Pvt Ltd, AVA®

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author nor **Orange Education Pvt Ltd** or its dealers and distributors, will be held liable for any damages caused or alleged to have been caused directly or indirectly by this book.

Orange Education Pvt Ltd has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capital. However, **Orange Education Pvt Ltd** cannot guarantee the accuracy of this information. The use of general descriptive names, registered names, trademarks, service marks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.

First Published: March 2026

Published by: Orange Education Pvt Ltd, AVA®

Address: 9, Daryaganj, Delhi, 110002, India

275 New North Road Islington Suite 1314 London,

N1 7AA, United Kingdom

ISBN (PBK): 978-93-49887-66-4

ISBN (E-BOOK): 978-93-49887-68-8

Scan the QR code to explore our entire catalogue

www.orangeava.com

Table of Contents

1. Introduction to Time Series and Mathematical Foundations
- Introduction
- Structure
- Overview of Stochastic Processes
- Discrete and Continuous Time
    - *Discrete-time Process*
    - *Continuous-time Process*
  - *Discrete and Continuous State*
    - *Discrete-State*
    - *Continuous State*
- Types of Stochastic Processes
  - *Markov Chains*
  - *Poisson Process*
  - *Brownian Motion*
  - *Random Walk*
- Comparison of Stochastic Processes
- Conclusion
- References

2. Preparing Time Series Data
- Introduction
- Structure
- Types of Stationarity
  - *Strong Stationarity*
  - *Weak Stationarity*
  - *Trend Stationarity*
  - *Difference Stationarity*
- Conclusion
- References

3. Tests for Stationarity – Part 1
- Introduction
- Structure

- Test for Stationarity
  - *Visual Inspection of Stationarity*
  - *Kolmogorov–Smirnov (K–S) Test*
- Dickey-Fuller Test (Original)
  - *Augmented Dickey-Fuller (ADF) Test*
- Conclusion
- References

4. Tests for Stationarity – Part 2

- Introduction
- Structure
- Test for Stationarity
  - *Phillips–Perron (PP) Test*
  - *Kwiatkowski-Phillips-Schmidt-Shin (KPSS) Test*
  - *Elliott-Rothenberg-Stock (DF-GLS) Test*
  - *Ng-Perron Test*
- Conclusion
- References

5. Tests for Stationarity – Part 3

- Introduction
- Structure
- Test for Stationarity
  - *Schmidt-Phillips Test*
  - *Variance Ratio Test*
- Zivot-Andrews Test
  - *Leybourne–McCabe Test*
- Conclusion
- References

6. Foundations of Time Series Preparation

- Introduction
- Structure
- Data Loading and Parsing
- Time Series Structure and Indexing
- Time Series Indexing and Resampling
- Handling Missing and Irregular Data

Detrending and Deseasonalizing in Time Series Analysis
Transformation Techniques
Conclusion
References

7. Statistical Models for Forecasting
Introduction
Structure
Autoregressive (AR) Models
Moving Average (MA) Models
Autoregressive Moving Average (ARMA) Model
Autoregressive Integrated Moving Average (ARIMA) Model
Seasonal ARIMA (SARIMA) Model
Exponential Smoothing Models
Conclusion
References

8. ML and DL for Timeseries
Introduction
Structure
Decision Tree Regressor
Random Forest
Gradient Boosted Trees (XGBoost)
Long Short-Term Memory (LSTM)
Conclusion
References

9. Multivariate Time Series Models
Introduction
Structure
Vector Autoregression (VAR)
Vector Error Correction Model (VECM)
Multivariate XGBoost
Multivariate Long Short-Term Memory (LSTM)
Conclusion
References

CHAPTER 1

Introduction to Time Series and Mathematical Foundations

Introduction

Time series analysis is a vibrant field of study focused on exploring how data points evolve over time. In this book, we delve into the mathematical foundations of time series analysis and forecasting. This book will focus on looking at practical examples and real-world applications of time series data, along with step-by-step instructions to apply them using Python.

Whether you are a data scientist, data analyst, data engineer, or Python programmer who works on data that deals with time, you will benefit from the chapters of this book. In this opening chapter, we lay the mathematical groundwork necessary to understand and apply advanced time series methods.

We will start this chapter by introducing the concept of a stochastic process, which models uncertainties and variations inherent in time-dependent data. By treating a time series as a realization of a stochastic process, we will gain powerful tools throughout this chapter to characterize and predict the behavior of time series data.

By the end of this chapter, readers will be able to understand the foundational role of stochastic processes in modeling time series data, distinguishing between discrete and continuous time/state formulations, and recognizing how randomness is systematically captured in dynamic systems. The readers will also be able to identify and differentiate key types of stochastic processes, including Markov Chains, Poisson Processes, Brownian Motion, and Random Walks, and grasp their mathematical formulations and real-world applications.

Structure

In this chapter, the following topics will be covered:

- Overview of Stochastic Processes
  - Discrete and Continuous Time
  - Discrete and Continuous State
- Types of Stochastic Processes
  - Markov Chains
  - Poisson Process
  - Brownian Motion
  - Random Walk
- Comparison of Stochastic Processes

Overview of Stochastic Processes

Stochastic processes are mathematical models that help in understanding or defining systems that evolve over time with randomness. These processes are fundamental to modeling time series data since time series data is often random, uncertain, and has a lot of variability. This section covers a detailed explanation of stochastic processes, their mathematical concepts, and their importance in time series analysis.

Discrete and Continuous Time

The simple definition of a stochastic process is that it is a collection of random variables indexed by time. It can be mathematically defined as follows.

$$X_t : t \in T$$

X_t is a random variable for each time point t, in the index set of T.

T represents continuous or discrete time.

Discrete-time Process

A discrete-time stochastic process is a type of stochastic process where the evolution of a system or phenomenon is observed at specific, separated

points in time, typically indexed by integers (for example, t = 0, 1, 2, …). At each time step, the process takes on a value from a set of possible outcomes, and this value is governed by some underlying probability distribution that may change over time or depend on past values. This kind of process is ideal for modeling phenomena where data is naturally collected at regular intervals, such as daily stock prices, monthly sales, or hourly temperature readings. The index set T in a discrete-time process can be counted. Example: t = 0, 2, 4, 6,…10

Continuous-time Process

The index set T in a continuous time process cannot be counted. For example: t can be between 0 and ∞.

Let us look at these concepts with a simple traffic story of a metropolis on a busy Monday morning. The streets of a metropolis are busy with cars, taxis, and buses, each hurrying to work or school. It is always a challenge for traffic engineers to model and predict ever changing congestion of these streets. Let us turn to stochastic processes and their mathematical frameworks to capture the randomness in how traffic evolves from moment to moment.

In this traffic story, let us consider X_t as the number of cars travelling past a traffic signal at time t. Over the course of a single day, the traffic increases and decreases, reflecting the unpredictable nature of the real-life road conditions. In this case, we can choose to either measure the number of cars in a discrete time interval, such as every 15 minutes, or on a continuous time basis, such as every second throughout the day.

To understand this further, let us simulate the traffic scenario by defining the traffic rate as a mathematical function.

Code:

```
def traffic_rate(time_minute):
  if ٠ <= time_minute < 360:
    return ٢
  elif 360 <= time_minute < 540:
    return ١٠
  elif 540 <= time_minute < 900:
    return ٥
```

```
  elif 900 <= time_minute < 1140:
   return ١٢
  else:
   return 3
```

In the preceding code, we have defined the code for the traffic rate as a function of time. This function, **`traffic_rate`**, determines the rate of traffic (how many cars are arriving per minute) at a given time of day. It divides the day into different time periods (such as morning, afternoon, evening, and so on) and assigns a specific traffic rate to each period. For example, if the time period is between 0 and 360, then the number of cars per minute is returned as 2. Thus, traffic rate is a simple function that returns the number of cars on the road at any point in time, depending on what time of the day it is.

In the next step, let us simulate the traffic at discrete time intervals.

Code:

```
def simulate_discrete_traffic(interval=15, total_minutes=1440,
seed=None):
  if seed is not None:
   np.random.seed(seed)

  times = list(range(0, total_minutes, interval))
  car_counts = []

  for t in times:
   lam = traffic_rate(t) * interval
   arrivals = np.random.poisson(lam)
   car_counts.append(arrivals)
  return times, car_counts
```

The preceding code simulates traffic flow over a day in discrete time intervals, for example, 15 minutes. It calculates how many cars arrive during each interval, if cars arrive randomly but follow a certain pattern at certain hours of the day. The peak hours of the day will have more cars, whereas regular hours will have fewer. The results are returned as two lists: **`times`** will consist of time intervals, and **`car_counts`** will consist of the corresponding number of cars.

In the preceding code, we have used the Poisson distribution to randomly generate the number of cars arriving at that interval. The Poisson distribution

is often used to model random events (such as car arrivals) that happen independently at a known average rate.

Now that we have defined the discrete traffic simulation, let us write a function to visualize it and understand what the traffic would look like.

Code:

```
def plot_traffic(times, car_counts, title="Traffic
Simulation"):
  plt.figure(figsize=(10, 5))
  plt.plot(times, car_counts, marker='o', linestyle='-',
  label='Cars Arriving')
  plt.title(title)
  plt.xlabel('Time (minutes)')
  plt.ylabel('Number of Cars')
  plt.grid(True)
  plt.legend()
  plt.show()
```

In the preceding code, **`plot_traffic`** is used to visualize the results of a traffic simulation. It takes data about when cars arrive (**`times`** and **`car_counts`**) and creates a line graph to show how the number of cars changes over time. It is like drawing a picture of the traffic flow throughout the day. It creates a line graph with time on the x-axis and the number of cars on the y-axis.

This graph would help us to see:

- When traffic is heavy (for example, during rush hour).
- When traffic is lightest (for example, late at night).
- How traffic changes over time.

With all the required functions defined, let us now bring the discrete traffic simulation to life by calling the functions that have been defined so far.

Code:

```
discrete_times, discrete_counts = simulate_discrete_traffic(
  interval=١٥,
  total_minutes=1440,
  seed=٤٢
)
```

```
print('Discrete Times', discrete_times)
print('Discrete Counts', discrete_counts)
```

The output of the simulated data is displayed as follows:

```
Discrete Times [0, 15, 30, 45, 60, 75, 90, 105, 120, 135, 150,
165, 180, 195, 210, 225, 240, 255, 270, 285, 300, 315, 330,
345, 360, 375, 390, 405, 420, 435, 450, 465, 480, 495, 510,
525, 540, 555, 570, 585, 600, 615, 630, 645, 660, 675, 690,
705, 720, 735, 750, 765, 780, 795, 810, 825, 840, 855, 870,
885, 900, 915, 930, 945, 960, 975, 990, 1005, 1020, 1035, 1050,
1065, 1080, 1095, 1110, 1125, 1140, 1155, 1170, 1185, 1200,
1215, 1230, 1245, 1260, 1275, 1290, 1305, 1320, 1335, 1350,
1365, 1380, 1395, 1410, 1425]
Discrete Counts [28, 34, 23, 32, 36, 24, 27, 29, 32, 26, 29,
25, 31, 32, 27, 33, 22, 26, 27, 31, 21, 17, 28, 36, 142, 135,
160, 158, 160, 145, 165, 144, 143, 159, 167, 133, 82, 82, 75,
70, 89, 73, 68, 69, 91, 78, 83, 88, 70, 68, 73, 63, 93, 75, 98,
75, 69, 78, 56, 89, 164, 187, 169, 175, 185, 158, 173, 187,
180, 186, 188, 206, 174, 204, 170, 194, 46, 55, 42, 48, 47, 35,
46, 48, 49, 41, 48, 48, 34, 40, 55, 51, 46, 38, 40, 48]
```

Let us further look at the overview and plot of the simulated data as follows:

```
print("=== Discrete-Time Simulation (every 15 min) ===")
print(f"Number of observations: {len(discrete_times)}")
print(f"Total cars over the day: {sum(discrete_counts)}")
print(f"Average cars per observation:
{np.mean(discrete_counts):.2f}")
print(f"Max cars in a single interval:
{max(discrete_counts)}\n")
plot_traffic(discrete_times, discrete_counts, "Discrete-Time
Traffic (Every 15 min)")
```

The preceding code simulates traffic over a full day (24 hours) and then analyzes and visualizes the results. It uses the **`simulate_discrete_traffic`** function to generate the data, calculates summary statistics such as total cars, average cars, and maximum cars in an interval, and finally creates a graph to show how traffic changes over time.

Output:

```
=== Discrete-Time Simulation (every 15 min) ===
```

```
Number of observations: 96
Total cars over the day: 8147
Average cars per observation: 84.86
Max cars in a single interval: 206
```

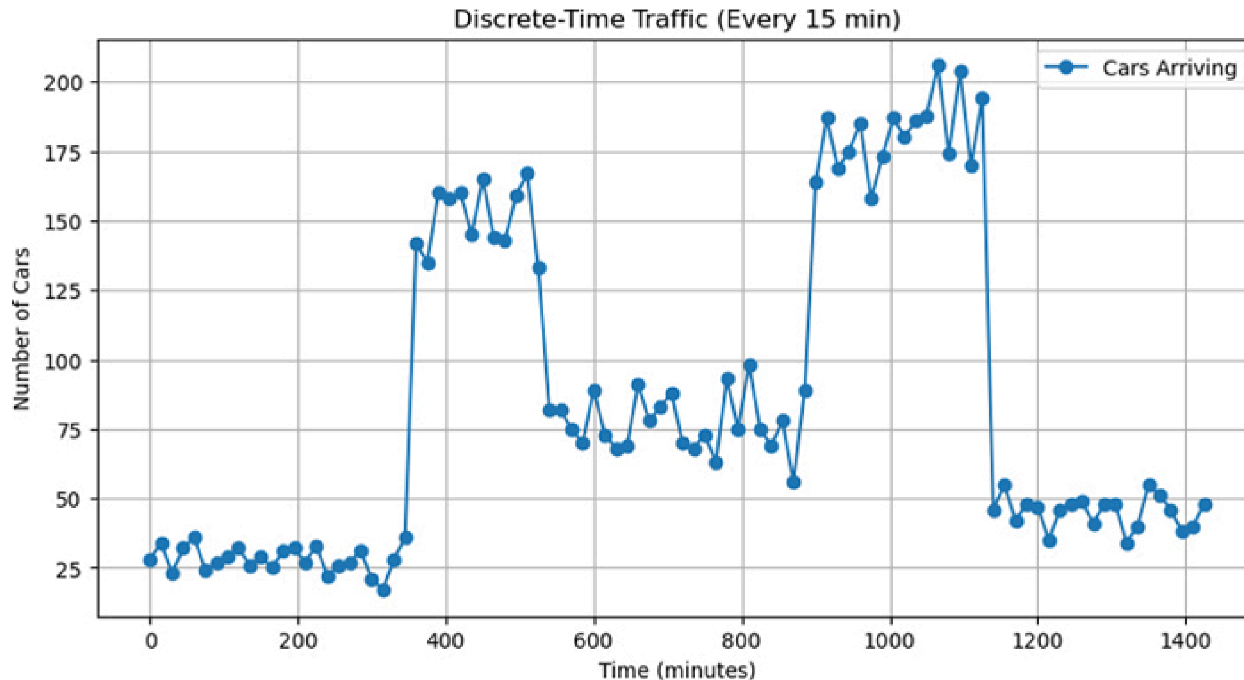

Figure 1.1: *Discrete Time Traffic*

Figure 1.1 depicts the traffic captured at every 15-minute interval over a day of 24 hours. *Figure 1.1* is a great example of a time series analysis for discrete data. The **`discrete_times`** that represent discrete time intervals of 15 minutes and **`discrete_counts`** that represent the number of cars arriving at each interval are a classic example of time series data, where each data point is tied to a specific time.

Let us now create a similar example for continuous time process and review the traffic application of time series further.

Code:

```
def simulate_continuous_traffic(delta=1, total_minutes=60,
seed=None):
  if seed is not None:
    np.random.seed(seed)

  steps = int(total_minutes / delta)
  times = [delta * i for i in range(steps)]
```

```
car_counts = []

for t in times:
    lam = traffic_rate(int(t)) * delta
    arrivals = np.random.poisson(lam)
    car_counts.append(arrivals)
return times, car_counts
```

In the preceding code, **`simulate continuous traffic`** simulates traffic flow over time in a more detailed way compared to the discrete time process simulation example. Instead of looking at traffic in fixed intervals (such as every 15 minutes), it simulates traffic in very small-time steps (for example, every 1 minute or even every second). This gives a more continuous view of how traffic changes over time.

Continuous traffic data is also generated using the Poisson distribution. It uses randomness to model car arrivals but follows a pattern defined by **`traffic_rate(t)`**. This function simulates traffic in very small time steps (for example, every minute), giving a more detailed view of traffic flow.

With all the required functions defined, let us now bring the continuous traffic simulation to life by calling the functions that have been defined so far.

Code:

```
continuous_times, continuous_counts =
simulate_continuous_traffic(
    delta=١,
    total_minutes=60,  # 1 hour
    seed=٤٢
)
print('continuous_times',continuous_times)
print('continuous_counts',continuous_counts)
```

The output of the simulated data is displayed as follows:

```
continuous_times [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13,
14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29,
30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45,
46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59]
continuous_counts [4, 1, 3, 3, 1, 2, 1, 1, 2, 2, 1, 0, 4, 2, 1,
3, 2, 6, 1, 1, 2, 2, 1, 3, 4, 1, 3, 1, 4, 4, 2, 0, 0, 2, 2, 2,
```

```
1, 5, 4, 0, 1, 2, 2, 0, 2, 2, 3, 1, 0, 2, 2, 1, 3, 2, 3, 2, 0,
2, 1, 1]
```

Let us further look at the overview and plot of the simulated data as follows.

```
print("=== Continuous-Time Simulation (every 1 min) ===")
print(f"Number of observations: {len(continuous_times)}")
print(f"Total cars over the hour: {sum(continuous_counts)}")
print(f"Average cars per minute:
{np.mean(continuous_counts):.2f}")
print(f"Max cars in a single minute:
{max(continuous_counts)}\n")

plot_traffic(continuous_times, continuous_counts, "Continuous-
Time Traffic (Every 1 min, 1 hour)")
```

The preceding code simulates traffic flow over 1 hour in 1-minute intervals (continuous time steps) and then analyzes and visualizes the results. It uses the **`simulate_continuous_traffic`** function to generate the data, calculates some summary statistics such as total cars, average cars, and maximum cars in an interval, and finally creates a graph to show how traffic changes over time.

Output:

```
=== Continuous-Time Simulation (every 1 min) ===
Number of observations: 60
Total cars over the hour: 116
Average cars per minute: 1.93
Max cars in a single minute: 6
```

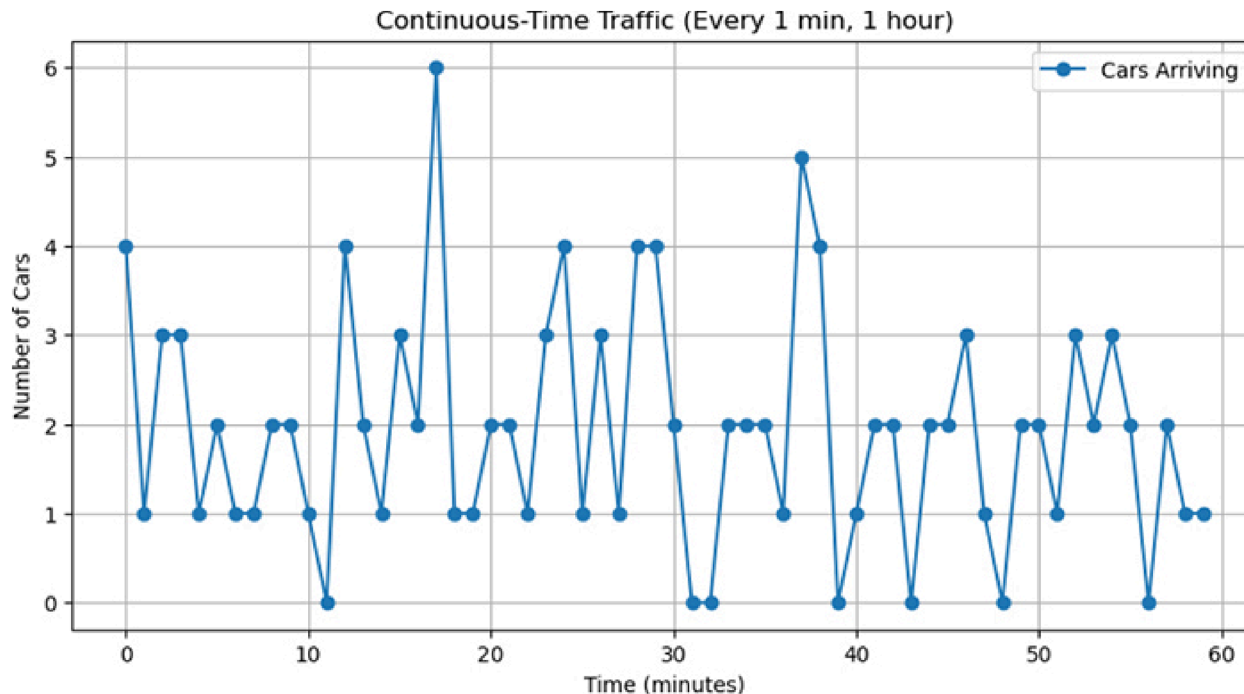

Figure 1.2: *Continuous Time Traffic*

Figure 1.2 depicts the traffic captured at every 1-minute interval over 1 hour. *Figure 1.2* is a great example of a time series analysis for continuous data. The **`continuous_times`** that represent continuous time intervals of every 1 minute and **`continuous_counts`** that represent the number of cars arriving at each interval are another classic example of time series data, where each data point is tied to a specific time.

Both the discrete time process and continuous time process examples covered in this section for traffic simulation can be used to:

- **Simulate traffic patterns:** Since it generates realistic data about how many cars arrive at different times of the hour or a day.
- **Analyze traffic:** Since it calculates useful statistics such as total cars, averages, and peak traffic.
- **Visualize traffic:** Since it creates a graph to make it easy to see trends, such as rush hours or quiet times.

This example lays the groundwork for more advanced time series analysis techniques, such as forecasting, decomposition, and statistical modeling.

Discrete and Continuous State

Now that we have looked at the time aspect of stochastic processes, let us further look at the State space concept of the stochastic processes.

For the same traffic stochastic processes, X_t, the state space is the set of all possible values for the number of cars that X_t can have at any time t.

How State impacts traffic modeling?

Like time, the State space can also be classified into two types:

For instance, if we are tracking the length of the queue, such as the number of cars waiting at a traffic signal, then the discrete state space comes into the picture.

On the other hand, if we are tracking speed, position, or travel time, then the **continuous** state space comes into the picture.

Discrete-State

The discrete state space is countable. For example, {0, 1, 2, 3,…} represents the number of cars in a queue at a traffic signal.

The number of cars in a traffic signal can be written in a discrete state as:

$$X_{t+1} = X_t + A_{t+1} - D_{t+1}$$

X_t {0, 1, 2, …} is the discrete state or the length of the queue at a traffic signal during time t.

A_{t+1} is the number of cars arriving at the traffic signal during the interval [t, t+1].

D_{t+1} is the number of cars departing the traffic signal during the interval [t, t+1].

To understand this further, let us simulate the discrete state space of our traffic example further by defining its mathematical function.

Code:

```
def simulate_discrete_queue(num_steps=50, lambd=2.0,
capacity=3, seed=None):
    if seed is not None:
        np.random.seed(seed)

    X = np.zeros(num_steps+1, dtype=int)

    for t in range(num_steps):
```

```
    arrivals = np.random.poisson(lambd)
    departures = min(X[t], capacity)
    X[t+۱] = X[t] + arrivals - departures

  return X
```

In the preceding code, the function **`simulate_discrete_queue`** simulates a traffic signal system where cars arrive randomly at the signal. The traffic signal allows a fixed number of cars to pass during each time step. It also tracks how the number of cars waiting changes over time.

This function can be used in applications where we want to:

- Model traffic flows at intersections.
- Optimize traffic signal timing to reduce congestion.
- Predict how many cars will be waiting during peak hours.

Now that we have defined the discrete queue simulation, let us write a function to visualize it and understand what the traffic would look like.

Code:

```
def plot_queue(X, title="Discrete-State Queue Length"):
  plt.figure(figsize=(8, 4))
  plt.step(range(len(X)), X, where='post', marker='o')
  plt.title(title)
  plt.xlabel("Time Step")
  plt.ylabel("Number of Cars")
  plt.grid(True)
  plt.show()
X_discrete = simulate_discrete_queue(num_steps=30, lambd=2.0,
capacity=3, seed=42)
print('Discrete data',X_discrete)
```

The output of the simulated data is displayed as follows:

```
Discrete data [0 4 2 3 3 1 2 1 1 2 2 1 0 4 3 1 3 2 6 4 2 2 2 1
3 4 2 3 1 4 5]
```

Let us further look at the overview and plot of the simulated data as follows:

```
plot_queue(X_discrete, "Number of Cars in a Discrete-State
Queue")
```

In the preceding code, our goal is to visualize how a line of cars at a traffic signal changes over time. The **`plot_queue`** function does exactly that—it creates a graph to show the number of cars waiting at the signal during each time step. The function takes a list of numbers (X) representing the number of cars waiting at the signal over time and turns it into a step-by-step graph.

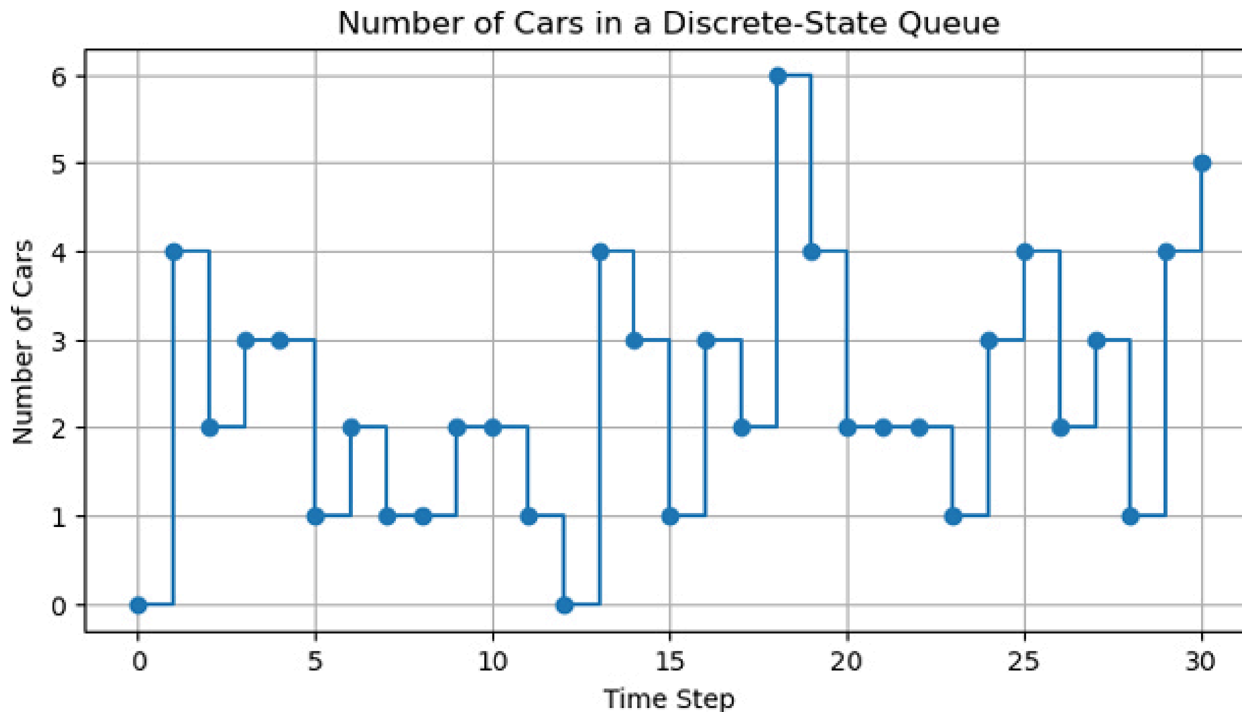

Figure 1.3: *Discrete State Queue*

Figure 1.3 shows:

- How the line of cars grows and shrinks.
- When the traffic signal lets cars through.
- How long the queue gets during peak times.

Let us now create a similar example for the continuous state process and review the traffic application of time series further.

Continuous State

The continuous space is an uncountable set, and it is usually referred to as R. For example, the exact location of the car on a highway can vary over a continuous range of real numbers.

The position of a car (in miles or kilometers) in continuous state space can be represented as:

$$Y_{t+1} = Y_t + Z_{t+1}$$

where:

$Y_t \in R$ the car can be in any real-valued position along the highway,

$Z_{t+1} \sim N(\mu, 2)$ is a normally distributed "step" that could capture acceleration, deceleration, or random fluctuations.

To understand this further, let us simulate the continuous state space of our traffic example further by defining its mathematical function.

Code:

```
def simulate_continuous_position(num_steps=50, mu=0.2,
sigma=0.5, seed=None):
  if seed is not None:
    np.random.seed(seed)

  Y = np.zeros(num_steps+1, dtype=float)

  for t in range(num_steps):
    step = np.random.normal(loc=mu, scale=sigma)
    Y[t+1] = Y[t] + step
  return Y
```

In the preceding code, the function **`simulate_continuous_position`** simulates real-world randomness in the movement of a car speeding up and slowing down unpredictably. The car's movement is random but follows a general trend (for example, it tends to move forward but sometimes slows down or even reverses slightly). The randomness is modeled using a normal distribution since it is a common distribution observed in most of the situations of daily traffic.

Now that we have created a function to track the position of the car on the highway over time, let us further create a function to visualize how far the car has traveled at each time step.

Code:

```
def plot_position(Y, title="Continuous-State Position"):
  plt.figure(figsize=(8, 4))
  plt.plot(range(len(Y)), Y, marker='o', linestyle='-')
```

```
  plt.title(title)
  plt.xlabel("Time Step")
  plt.ylabel("Position (miles)")
  plt.grid(True)
  plt.show()
Y_continuous = simulate_continuous_position(num_steps=30,
mu=0.2, sigma=0.5, seed=42)
print('Y_continuous', Y_continuous)
```

The output of the simulated data is displayed as follows:

```
Y_continuous [0.         0.44835708 0.57922493 1.10306919
2.06458412 2.14750744
2.23043896 3.22004537 3.80376273 3.76902554 4.24030556
4.20859671
4.17573184 4.49671297 3.74007285 3.07761393 2.99647017
2.69005461
3.04717827 2.79316624 2.28701439 3.21983877 3.30695062
3.54071472
3.02834063 2.95614927 3.21161056 2.83611377 3.22396278
3.12364344
3.17779656]
```

Let us further look at the overview and plot of the code as follows.

```
plot_position(Y_continuous, "Car Position in a Continuous-State
in a highway")
```

In the preceding code, the **`plot_position`** function takes a list of positions of the car at different time steps as input and turns it into a line graph.

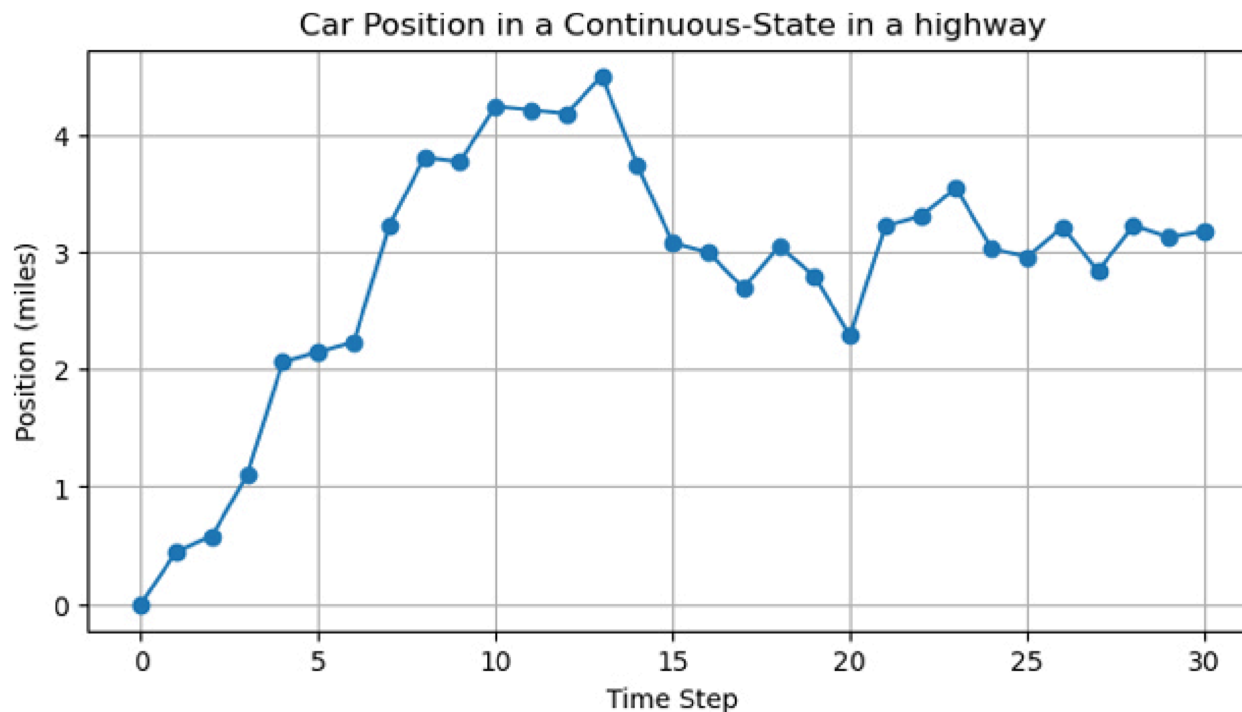

Figure 1.4: *Continuous State Position*

Figure 1.4 shows:

- How the car's position changes over time.
- Whether the car is moving forward, backward, or staying still.
- The randomness or smoothness of the car's movement.

Now that we have looked at the time and state aspects of stochastic processes, let us further look at the type of stochastic processes that help in understanding the randomness of time series data.

Types of Stochastic Processes

The common types of stochastic processes, along with their mathematical workings, are discussed in this section.

Let us look at the common types of stochastic processes as follows:

1. Markov Chains
2. Poisson Processes
3. Brownian Motion
4. Random Walks

Markov Chains

A Markov chain is a type of stochastic process that satisfies the Markov property:

```
PXn+1=j | Xn=i, Xn-1,…, X0)=P(Xn+1=j | Xn=i)
```

The probability of moving to a future state depends only on the current state and not on history.

From the traffic example,

`Xn` denotes the number of cars at a traffic signal at time step n.

The probability distribution of the number of cars at the next time step, Xn+1, depends only on `Xn` and not on the past states of the queue.

For simplicity of this example, let us make use of a simple set {0,1,2,3}. The model in this case has four states: state of 0 cars, 1 car, 2 cars, and 3 cars.

Let us define a transition matrix P, where each entry captures the probability of moving from state i to state j.

```
Pi, j=P(Xn+1=j | Xn=i)
```

A hypothetical transition matrix for this will look like:

P= 0.5 0.4 0.1 0.0 0.2 0.4 0.1 0.1 0.1 0.3 0.4 0.2 0.0 0.2 0.4 0.4

In this example P1,2=0.3, which means that there is a 30% chance of going from 1 car to 2 cars in the next time step.

To understand this further, let us simulate the Markov chains on our traffic example further by defining its mathematical function.

Code:

```
def simulate_markov_chain(transition_matrix, initial_state,
num_steps=20, seed=None):
  if seed is not None:
    np.random.seed(seed)

  num_states = transition_matrix.shape[0]
  states = [initial_state]

  for _ in range(num_steps):
    current_state = states[-1]
    probs = transition_matrix[current_state]
    next_state = np.random.choice(num_states, p=probs)
```

```
    states.append(next_state)

  return states
```

In the preceding code, **`simulate_markov_chain`** function simulates the Markov chain that moves between different states by taking the probabilities of state change in the transition matrix as input.

Let us define the transition matrix as follows:

Code:

```
transition_matrix = np.array([
  [٠.٠ ,٠.١ ,٠.٤ ,٠.٥],
  [٠.١ ,٠.٣ ,٠.٤ ,٠.٢],
  [٠.٢ ,٠.٤ ,٠.٣ ,٠.١],
  [٠.٤ ,٠.٤ ,٠.٢ ,٠.٠]
])
```

Let us further call the **`simulate_markov_chain`** function and review the results.

Code:

```
states = simulate_markov_chain(transition_matrix,
initial_state=0, num_steps=20, seed=42)
print('Simulated states', states)
print("=== Markov Chain Simulation for Traffic Queue ===")
print("Transition Matrix:\n", transition_matrix)
print("Simulated State Sequence:", states)
```

The preceding code generates the following output.

Code:

```
Simulated states [0, 0, 2, 2, 2, 1, 0, 0, 1, 2, 2, 0, 2, 3, 2,
1, 0, 0, 1, 1, 1]
=== Markov Chain Simulation for Traffic Queue ===
Transition Matrix:
  [[٠  ٠.١ ٠.٤ ٠.٥.  ]
  [٠.١ ٠.٣ ٠.٤ ٠.٢]
  [٠.٢ ٠.٤ ٠.٣ ٠.١]
  [٠.٤ ٠.٤ ٠.٢   .٠]]
Simulated State Sequence: [0, 0, 2, 2, 2, 1, 0, 0, 1, 2, 2, 0,
2, 3, 2, 1, 0, 0, 1, 1, 1]
```

Simulated state sequence is the simulated sequence of the number of cars transitioning from one state to another based on the probability matrix.

Let us define a function to visualize how the Markov chain states behave.

Code:

```
def plot_markov_chain_states(states, title="Markov Chain States
Over Time"):
  plt.figure(figsize=(8, 4))
  plt.step(range(len(states)), states, where='post', marker='o')
  plt.title(title)
  plt.xlabel('Time Step')
  plt.ylabel('Queue Length')
  plt.ylim(-0.5, max(states) + 0.5)
  plt.grid(True)
  plt.show()
plot_markov_chain_states(states, title="Number of Cars at a
Traffic Signal (Markov Chain)")
```

The preceding code takes a list of states from a Markov Chain simulation as inputs and creates a step plot.

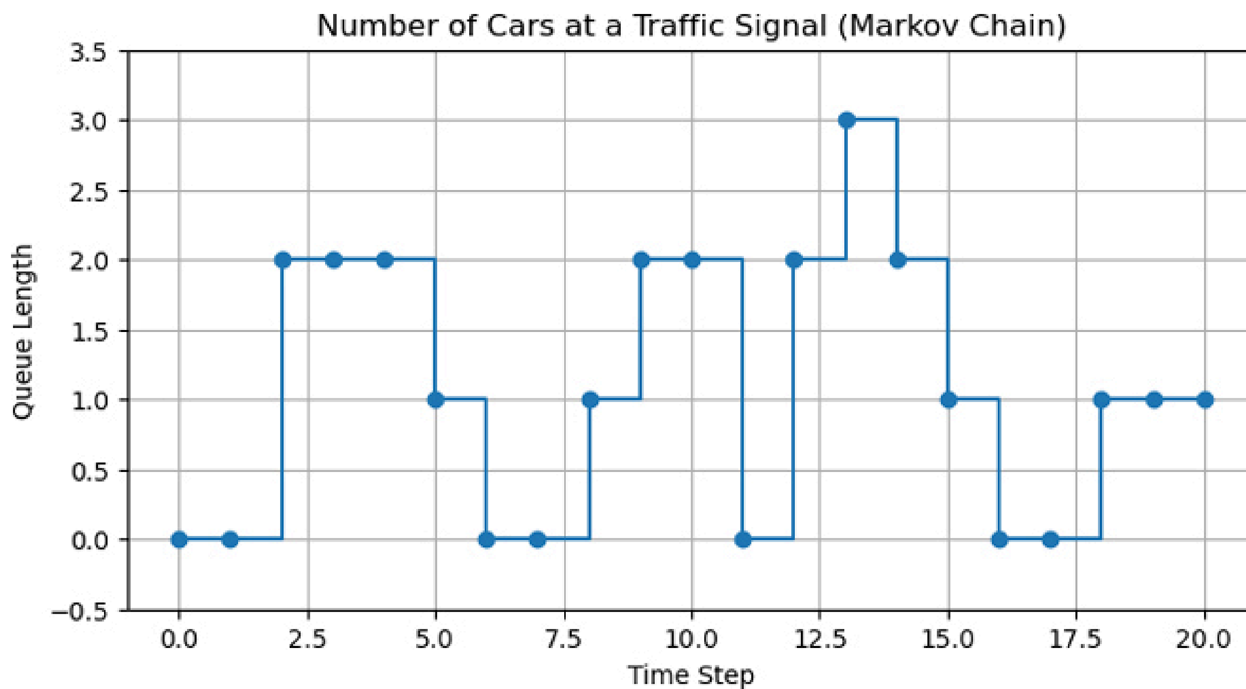

Figure 1.5: *Markov Chain State Space*

Figure 1.5 demonstrates the number of cars simulated by the Markov chain at each time step.

Having discussed Markov chains as one type of stochastic process, we now turn to another type—the Poisson process.

Poisson Process

A Poisson process is a stochastic process that models the number of cars arriving or events occurring at any point in time.

In a Poisson process, the rate is the rate at which events happen at any point in time. In this case, the number of cars arriving at the traffic signal at any unit of time is calculated at the rate of λ cars per unit time. It can be defined as follows.

$$Nt-Ns \sim Poisson(\lambda t-s)$$

where:

N(t) is the number of cars arriving at time t, and

N(s) is the number of cars arriving at time s

For $0 \leq s \leq t$.

Similarly, the number of arrivals in the interval (0,t) will be:

$$Nt \sim Poisson(\lambda t)$$

This can be rewritten into the Poisson probability function as follows:

$$PNt=k=(\lambda t)kk!\ e-\lambda t$$

where:

P(k) = the probability of k arrivals in an interval, and

λt = mean number of arrivals in an interval.

Properties of Poisson Process

The following are the two most important properties of the Poisson process that make it ideal for simulating the car arrivals at a traffic signal:

- **Memoryless Arrivals:** The car arrivals according to the Poisson process will occur randomly over time with no memory of how long it has been since the last arrival.

- **Stationary Increments:** During a stable period over the day, the arrival rate might roughly be constant over time.

To understand this further, let us simulate the Poisson Process on our traffic example further by defining its mathematical function.

Code:

```
def simulate_poisson_process(rate, max_time, seed=None):
 if seed is not None:
  np.random.seed(seed)

 arrival_times = []
 current_time = 0.0

 while True:
  interarrival = np.random.exponential(scale=1.0/rate)
  current_time += interarrival

  if current_time > max_time:
   break
  arrival_times.append(current_time)
 return arrival_times
```

In the preceding code, **`simulate_poisson_process`** function simulates car arrivals at a traffic signal over a period of time. It models the arrivals as a Poisson process, which is a common way to describe random events such as cars arriving at a certain average rate. It takes in **`rate`** as one of the inputs, which is the average number of cars arriving per unit of time. It also takes in **`max_time`** as input, which is the total time for this simulation. This function returns a list of arrival times for the cars. The time between arrivals follows an exponential distribution, which is a key property of Poisson processes.

Code:

```
rate = 2.0
max_time = 10.0
seed = 42

arrivals = simulate_poisson_process(rate, max_time, seed=seed)
print("Number of arrivals:", len(arrivals))
print("Arrival times:", arrivals)
```

This function can be called by providing the required inputs for **rate**, **max_time**, and **seed** (random seed value).

Output:

```
Number of arrivals: 23
Arrival times: [0.23463404498842955, 1.7396947604471902,
2.398067607219915, 2.8545388841078916, 2.939351319339065,
3.024149465296367, 3.0540688496007076, 4.059684281840678,
4.51922535865431, 5.134850389506606, 5.145250043506175,
6.897028781085331, 7.790243552762669, 7.909587365387142,
8.009926859761974, 8.111232571178101, 8.29260943590834,
8.664573351338767, 8.947341884728935, 9.119453381005906,
9.592638817955805, 9.667756082319475, 9.840513838320676]
```

The **simulate_poisson_process** function has generated the time of arrivals as output.

In the next step, let us define a function to plot the Poisson process.

Code:

```
def plot_poisson_arrivals(arrival_times, max_time):
  fig, ax = plt.subplots(figsize=(8, 1.5))

  ax.hlines(y=1, xmin=0, xmax=max_time, color='black')
  for t in arrival_times:
    ax.plot(t, 1, marker='o', color='red')

  ax.set_ylim(0.8, 1.2)
  ax.set_xlim(0, max_time)
  ax.set_xlabel("Time")
  ax.set_title("Poisson Process Arrivals")
  plt.show()
plot_poisson_arrivals(arrivals, max_time)
```

In the preceding code, the function **plot_poisson_arrivals** visualizes the car arrival times generated by the **simulate_poisson_process** function. It creates a timeline plot where each car arrival is marked as a red dot along a horizontal line. This makes it easy to see when cars arrive over time.

Figure 1.6: *Poisson Process Arrivals*

Figure 1.6 shows a **`timeline`** plot of car arrival times. It uses a horizontal line to represent time and red dots to mark arrivals. This plot helps to visualize the random events of car arrivals at a traffic signal.

Randomness in discrete time intervals is well captured by the Poisson distribution. The Poisson distribution is suitable for both discrete and continuous time series data. Let us further look at another stochastic process that is well-suited for continuous time intervals and continuous states.

Brownian Motion

Brownian motion or Wiener process is a stochastic process that can be used to understand the randomness in continuous-time and continuous-state time series data.

Standard Brownian motion is defined as follows:

Let ${W_t}_{t\geq 0}$ be a standard Brownian motion. Then, for each increment Δt:

$$W_{t+\Delta t} - W_t \sim N(0, \Delta t)$$

Over multiple increments:

$$W_t = {}_{k=1}^{n}(W_{t_k} - W_{t_{k-1}}),$$

where:

each $(W_{t_k} - W_{t_{k-1}})$ - increment is independent, and

$\sim N(0, t_k - t_{k-1})$ - increments are normally distributed.

The randomness in Brownian motion can be either positive or negative, and it can be unpredictable. For our traffic example, this randomness can go only in a positive direction since the movement of a car on a highway is in a forward direction. The Brownian motion, being the randomness in

continuous-state and continuous-time, can be applied to either the speed or the position of the car. Even though the movement of vehicles in traffic is strategic and is not totally uncertain and not a Brownian example, we will apply this concept to understand Brownian motion in general, since it is one of the stochastic processes that can impact time series data.

To understand this further, let us simulate the Brownian motion on our traffic example further by defining its mathematical function.

Code:

```
def simulate_brownian_motion(total_time=1.0, num_steps=1000,
seed=None):
  if seed is not None:
    np.random.seed(seed)

  dt = total_time / num_steps
  times = np.linspace(0, total_time, num_steps+1)

  increments = np.sqrt(dt) * np.random.randn(num_steps)

  W = np.zeros(num_steps+1)
  for i in range(1, num_steps+1):
    W[i] = W[i-1] + increments[i-1]

  return times, W
```

In the preceding code, the function **`simulate_brownian_motion`** simulates Brownian motion, also called a Wiener process. In the context of traffic, this code is for modeling the random fluctuations in the position of the car in traffic flow over time. The function returns two arrays: one for the time points and one for the corresponding values of the Brownian motion.

This function can be called by providing the required inputs for **`total_time`**, **`num_steps`**, and **`seed`** (random seed value).

Code:

```
times, W = simulate_brownian_motion(total_time=1.0,
num_steps=1000, seed=42)
print("=== Brownian Motion Simulation ===")
print(f"Final time: {times[-1]}")
print(f"Final position W(1.0) = {W[-1]:.4f}")
```

```
print(f"Mean of increments (should be close to 0):
{np.mean(np.diff(W)):.4f}")
print(f"Var of increments (should be close to dt):
{np.var(np.diff(W)):.4f}")
```

The **`simulate_brownian_motion`** function has generated the time and positions of the car as output.

```
=== Brownian Motion Simulation ===
Final time: 1.0
Final position W(1.0) = 0.6113
Mean of increments (should be close to 0): 0.0006
Var of increments (should be close to dt): 0.0010
```

In the next step, let us define a function to plot the Brownian motion.

Code:

```
def plot_brownian_motion(times, W, title="Brownian Motion
Simulation"):
  plt.figure(figsize=(10, 4))
  plt.plot(times, W, label="W(t)")
  plt.title(title)
  plt.xlabel("Time")
  plt.ylabel("Position (W(t))")
  plt.grid(True)
  plt.legend()
  plt.show()
plot_brownian_motion(times, W)
```

In the preceding code, the function **`plot_brownian_motion`** visualizes the results of a Brownian motion simulation. It takes the time points and the corresponding values of the Brownian motion as input and creates a line graph to show how the position of the car changes over time. This makes it easy to see the random fluctuations of the process.

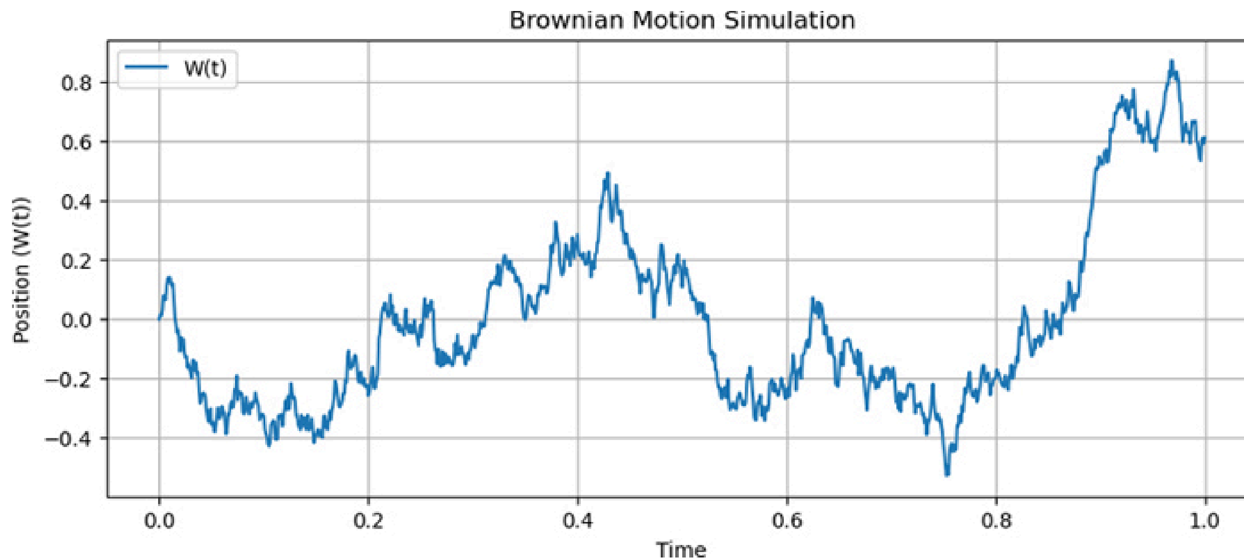

Figure 1.7: *Brownian Motion Arrivals*

Figure 1.7 shows a line graph of Brownian motion over time, with time on the x-axis and the position of the car due to Brownian motion on the y-axis.

Randomness in continuous time intervals and continuous state is well captured by Brownian motion. Let us further look at another slightly different stochastic process that is also well-suited for discrete time intervals, but can also be used for continuous time.

Random Walk

Random walk is one of the simplest stochastic processes and is suitable for discrete time intervals for capturing features such as speed and position of the car, rather than arrivals. In Random walk, each new state depends on the previous state plus some random increment.

Random walk can be defined as follows:

$$X_{t+1} = X_t + S_{t+1}$$

where, S_{t+1} is a random variable and each step in a state space can either move forward or backward.

Similar to the Brownian movement, the Random walk is also not a real-life scenario for the traffic example, but will be covered for conceptual understanding of one of the possible stochastic processes.

Let X_t be the position of a car at discrete time steps t = 0, 1, 2,…

$$X_{t+1} = X_t + S_{t+1}$$

where:

X_0 is the initial position of the car,

S_{t+1} is a random step where,

$$S_{t+1} = \{+1, \text{ with probability } p, -1, \text{ with probability } (1-p).$$

To understand this further, let us simulate the Random Walk on our traffic example further by defining its mathematical function.

Code:

```
def simulate_random_walk(num_steps=50, p=0.5, seed=None):
  if seed is not None:
   np.random.seed(seed)

  positions = np.zeros(num_steps + 1, dtype=int)

  for t in range(1, num_steps + 1):
   step = ١ if np.random.rand() < p else -1
   positions[t] = positions[t-١] + step

  return positions
```

In the preceding code, the function **simulate_random_walk** simulates a 1-dimensional random walk. For a car moving in traffic, at each step, it can either move forward (+1) or backward (-1). The direction it moves is assumed to be random, but the probability of moving forward is given by p. This function returns the positions of the car at each step.

This function can be called by providing the required inputs for **num_steps**, probability, and seed (random seed value).

Code:

```
positions = simulate_random_walk(num_steps=50, p=0.6, seed=42)
print("Final position:", positions[-1])
```

The **simulate_random_walk** function has generated the positions of the car as output.

```
Final position: 18
```

In the next step, let us define a function to plot the Random walk.

Code:

```
def plot_random_walk(positions, title="Random Walk of a
Vehicle"):
  plt.figure(figsize=(8, 4))
  plt.plot(positions, marker='o', label='Position')
  plt.axhline(y=0, color='gray', linestyle='--', alpha=0.5)
  plt.title(title)
  plt.xlabel("Time Step")
  plt.ylabel("Block Number")
  plt.grid(True)
  plt.legend()
  plt.show()
plot_random_walk(positions, "Car's Random Walk Along City
Blocks")
```

In the preceding code, the function **`plot_random_walk`** visualizes the results of the random walk simulation of a car in our traffic example. It takes the positions of the car at each time step as input and creates a line graph to show how the position changes over time. This makes it easy to see the random movement of the car.

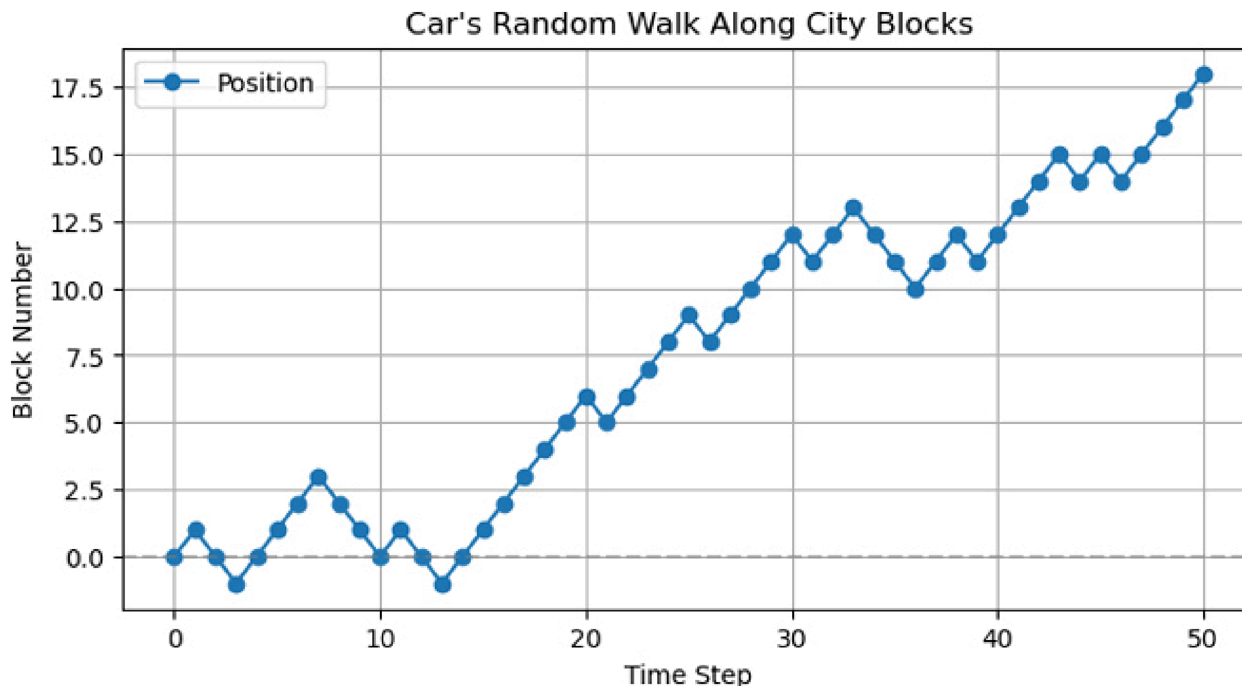

Figure 1.8: *Random Walk Positions*

Figure 1.8 shows a line graph of Random Walk over discrete time steps with time on the x-axis and the position of the car due to Random Walk on the y-

axis.

Comparison of Stochastic Processes

Having explored each stochastic process in detail, Markov chains, Poisson processes, Brownian motion, and random walks, this final section consolidates their key differences and unifies their theoretical foundations. These processes vary in their treatment of time (discrete versus continuous), state space (discrete versus continuous), and the nature of their randomness (memoryless transitions, event-driven jumps, Gaussian noise, or binomial steps).

Let us make a quick comparison of all the 4 types of stochastic processes covered in this chapter.

Characteristic	Markov Chain	Poisson Process	Brownian Motion	Random Walk
Time Domain	Discrete or Continuous	Discrete or Continuous	Continuous	Discrete
State Space	Discrete	Discrete (Counts)	Continuous	Discrete (typically integers)
Markov Property	Yes	Not required, but can be embedded	Not strictly, but can be embedded	Yes
Nature of Increments	Depends on transition probabilities	Stationarity and independent	Independent and normally distributed	Independent with probability p for +1 and 1-p for -1

Table 1.1: *Comparison of Stochastic Processes*

All these types of stochastic processes form the basis of measuring time series data and are further used in applications of forecasting time series and performing predictive analytics on the data. By understanding how data evolves, whether it arrives at random intervals as in Poisson, fluctuates continuously as in Brownian, or updates in discrete steps with memoryless transitions as in Markov, we can build models to predict future values or states.

Conclusion

In this chapter, we learned about an overview of the mathematical foundations of time series data, and we looked at the applications of stochastic processes with a traffic example.

We also looked at the concepts of discrete and continuous time and state in the domain of time series. We compared 4 different types of stochastic processes and analyzed them with examples as well as codes.

In the next chapter, we will look at the mathematical concepts behind data analysis, preparation, and visualization of time series in Python with examples and code.

References

- Komkov, V. and Dannon, V. (1991) Random walk simulation of chemical reactions represented by nonlinear reaction-diffusion equations, *ZAMM - Journal of Applied Mathematics and Mechanics / Zeitschrift für Angewandte Mathematik und Mechanik*, 71(3), pp. 103–110. Available at: https://doi.org/10.1002/zamm.19910710302.
- Trajstman, A.C. (2002) A Markov chain model for Newcastle disease and its relevance to the intracerebral pathogenicity index, *Biometrical Journal*, 44(1), pp. 43–57. Available at: https://doi.org/10.1002/1521-4036(200201)44:1<43::aid-bimj43>3.0.co;2-3.
- Addison, P.S., Qu, B., Nisbet, A. and Pender, G. (1997) A non-Fickian, particle-tracking diffusion model based on fractional Brownian motion, *International Journal for Numerical Methods in Fluids*, 25(12), pp. 1373–1384. Available at: https://doi.org/10.1002/(sici)1097-0363(19971230)25:12<1373::aid-fld620>3.0.co;2-6.
- Anderson, D.R., Sweeney, D.J., Williams, T.A., Camm, J.D., and Cochran, J.J. (2019) *Statistics for Business & Economics*. 14th edn. Boston: *Cengage Learning*, pp. 258–260.

You’ve Just Finished your Free Sample

Enjoyed the preview?

Buy: http://www.ebooks2go.com