Python Programming for Mechanical Engineers

Book · September 2023

DOI: 10.13980/RG.2.2.33942.85972

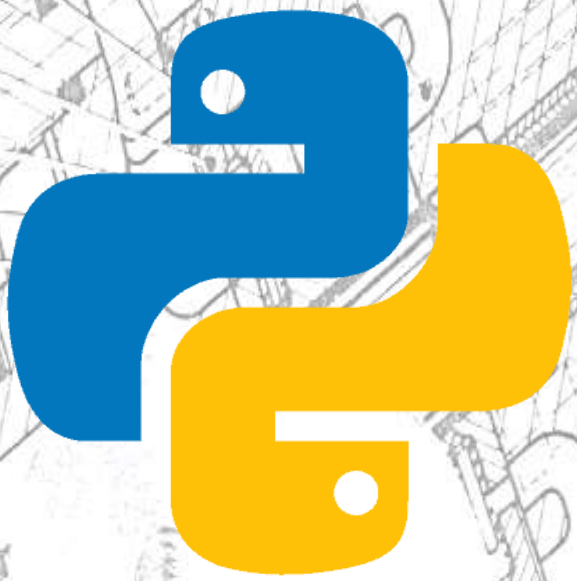
1 author:

Abdellatif M. Sadeq
Qatar University
45 PUBLICATIONS **142** CITATIONS

SEE PROFILE

First Edition

Python Programming for Mechanical Engineers



Abdellatif M. Sadeq

Python Programming for Mechanical Engineers

First Edition: September 2023

DOI: 10.13980/RG.2.2.33942.85972

Preface

Welcome to the world of Python programming and its fascinating applications within the domain of Mechanical Engineering! This book serves as your portal to mastering Python, spanning from its foundational principles to its advanced utilization in addressing intricate engineering issues.

In our modern, technology-driven world, Python has emerged as a potent and adaptable programming language, rendering it an essential tool for professionals across various sectors, including Mechanical Engineering. Whether you are a student eager to explore the captivating world of programming or a seasoned engineer aiming to enhance your analytical and problem-solving capabilities, this book offers a wealth of knowledge.

Chapter A: Introduction to Python

Our expedition commences with Chapter A, where we establish the framework for your Python programming aptitude. We commence with the fundamentals, acquainting you with Python's lucid and easily comprehensible syntax, its straightforwardness, and its dynamic characteristics. Furthermore, you will become adept at configuring your Python development environment, executing comments, manipulating lists, and delving into fundamental functions.

Chapter B: Application of Python in Mechanical Engineering

Chapter B marks the initiation of our genuine voyage. Here, we immerse ourselves in the pragmatic implementation of Python within the realm of Mechanical Engineering, exploring an array of captivating projects and simulations. These applications have been meticulously selected to offer you hands-on experience and a profound understanding of how Python can revolutionize your approach to engineering predicaments.

Here is a glimpse of what Chapter B encompasses:

- Scrutinizing Cyclist Air Resistance: We will plunge into the realm of aerodynamics, scrutinizing cyclist air resistance and performance through Python.

- Kinematics Simulation of a 2R Robotic Arm: You will delve into the kinematics of a 2R robotic arm, from equations to dynamic animations, all propelled by Python.

- Otto Cycle Simulation and In-depth Analysis: We shall simulate and scrutinize the efficiency and pressure-volume (PV) diagram of the Otto cycle.

- Dynamic Simulation of Damped Pendulum Motion: You will craft dynamic simulations of damped pendulum motion, procuring insights into the behavior of mechanical systems.

- Computation of Pressure and Analysis of Relaxation Occurrences: We will employ the Newton-Raphson technique to compute pressure and analyze relaxation occurrences.

- Exploration of Parameter Significance and Curve Fitting: You will acquire adeptness in scrutinizing parameter significance and curve fitting within Python.

- Analysis and Visualization of Engine Data: We will harness Python's capabilities to scrutinize and visualize data stemming from actual engines.

This book is designed to be accessible to readers with varying levels of programming expertise. Whether you are an initiate or a proficient programmer, you will discover invaluable insights, pragmatic illustrations, and methodical guidance to assist you in realizing Python's full potential within Mechanical Engineering.

As you embark on this odyssey through the chapters, you will not only amass indispensable programming skills but also cultivate the ability to confront real-world engineering predicaments with assurance. I invite you to explore the universe of Python programming and Mechanical Engineering, where originality and predicament resolution harmonize. Let us immerse ourselves and unearth the extraordinary opportunities that await you!

About the Author

Dr. Abdellatif M. Sadeq has earned his B.Sc. (2015), M.Sc. (2018), and Ph.D. (2022) in mechanical engineering from Qatar University. On September 2023, he has obtained a second M.Sc. in hybrid and electric vehicles design and analysis. He has been working as a graduate teaching and research assistant at Qatar University since 2015. He has over eight years of experience teaching undergraduate students various general and mechanical engineering courses. His areas of expertise in research are internal combustion engines, premixed turbulent combustion, alternative fuels, fuel technology, hydrogen production, storage, and combustion, electric and hybrid electric vehicles design and analysis, renewable energy utilization, energy storage techniques, system modelling and simulation, automotive wiring harness, battery technology, heat transfer, and HVAC.

Author's Signature

Using this Book

Welcome to this practical learning experience, thoughtfully designed to equip Mechanical Engineers with hands-on Python programming skills. This book centers on the principle of active learning through practical application, a method particularly advantageous for individuals within the realm of Mechanical Engineering.

Learning through practical application:

The core philosophy of this book revolves around "learning through practical application" Instead of inundating you with abstract theory devoid of real-world context, the present approach immerses you in tangible projects and scenarios. Each chapter is carefully crafted to provide a concrete grasp of Python programming by exploring applications in Mechanical Engineering. You will actively engage with Python code to tackle engineering challenges, gaining invaluable skills and insights in the process.

Why this book is tailored for Mechanical Engineers?

While Python is a versatile language with broad applicability, this book is purposefully customized to cater to the specific needs of Mechanical Engineers. Why? Because I recognize that your requirements, complexities, and interests are distinct. Here is why this book is an ideal choice for you:

Relevance: The projects and simulations featured in this book directly align with the core principles and issues encountered in Mechanical Engineering. You will address problems that are directly applicable to your field, deepening your ability to utilize Python in your professional endeavors.

Applicability: Mechanical Engineers often grapple with intricate physical systems and data analysis tasks. Python's adaptability enables you to craft practical solutions for tasks such as simulation, data analysis, and visualization, all of which are extensively covered within these pages.

Hands-On Approach: Mechanical Engineers thrive on hands-on problem-solving, and this book is strategically crafted to accommodate this inclination. You will engage in

projects that involve real-world data analysis, the simulation of physical systems, and the visualization of engineering concepts, thereby reinforcing your practical skills.

Skill Enhancement: Proficiency in Python programming is increasingly valuable in the realm of Mechanical Engineering. By actively participating in the projects outlined in this book, you will not only bolster your Python expertise but also gain a competitive advantage within your profession.

Who can benefit from this book?

While this book is tailored primarily for Mechanical Engineers, its utility extends to a broader audience. If you possess a background in engineering or related fields and aspire to apply Python to address engineering challenges, this book is a valuable resource. Whether you are a student, a practicing engineer, or an enthusiast eager to explore Mechanical Engineering using Python, you will find this book to be an invaluable companion.

In summary, this book serves as your pathway to proficiency in Python programming while actively addressing the unique challenges of Mechanical Engineering. Embrace the "learn by doing" philosophy, dive into the projects, and unlock the potential of Python in your journey within the realm of Mechanical Engineering. Let us embark on this dynamic learning adventure together!

Table of Contents

Introduction

This book offers a comprehensive and practical exploration of Python programming, with a specialized focus on its application in the field of Mechanical Engineering. It is organized into two main chapters, each with a distinct purpose:

Chapter A: Introduction to Python

Within this foundational chapter, readers embark on an informative journey into the realm of Python programming. It commences by providing a solid introduction to Python's fundamental features, ensuring that readers establish a robust grasp of the language's core elements. This includes a deep dive into Python's clean and easily comprehensible syntax, its status as an interpreted language that promotes swift code development, and its compatibility across various operating systems.

Additionally, Chapter A addresses essential topics such as the effective use of comments, the concept of lists (analogous to arrays in other programming languages), the dynamic appending of elements through the 'append' method, and the utilization of the 'POW()' function for mathematical operations. Furthermore, it explores the execution of loops and introduces Python's modular capabilities, emphasizing the language's extensibility. The chapter concludes by highlighting the significance of functions in code organization and modularity.

Chapter B: Application of Python in Mechanical Engineering

Chapter B is a practical journey into the application of Python in Mechanical Engineering. It offers hands-on projects that cover a wide range of Mechanical Engineering topics, including sports engineering, robotics, thermodynamics, physics simulations, fluid dynamics, experiments, and automotive engineering. These projects involve tasks like analyzing cyclist air resistance, simulating robotic arm kinematics, examining Otto cycle efficiency, animating damped pendulum motion, computing pressure using the Newton-Raphson technique, exploring parameter meanings through curve fitting, and analyzing engine data. This chapter transforms Python from a theoretical concept into a powerful tool for solving real-world challenges in the field of Mechanical Engineering.

Each project is thoughtfully designed to provide readers with practical experience in applying Python to solve complex Mechanical Engineering challenges. The hands-on approach ensures a deep comprehension of Python's capabilities and its relevance within the context of Mechanical Engineering. Throughout the book, the overarching focus remains on "learning by doing" ensuring that readers not only grasp Python programming principles but also acquire the skills and knowledge essential for excelling as proficient Mechanical Engineers with expertise in Python.

Chapter A: Introduction to Python

Welcome to Chapter A, the entry point to an exhilarating expedition into the world of Python programming. Within these pages, we will establish the bedrock of your Python programming skills, commencing with the fundamental principles and progressively advancing to more intricate concepts.

A1: Basic Features

Our journey begins with an introduction to Python's fundamental characteristics, equipping you with a strong understanding of the language's essentials. Python's allure lies in its simplicity and versatility, attributes that make it a favored choice for programmers of various backgrounds and expertise levels.

A2: Installation of Anaconda and Spyder User Interface

Before we dive into the coding realm, it is imperative to set up your Python development environment. We will provide you with step-by-step guidance on installing Anaconda and configuring the Spyder User Interface, ensuring you have the essential tools at your disposal for Python programming. This meticulous setup is akin to preparing a canvas before a painting comes to life.

A3: Execution of Comments

Mastery of employing comments effectively is a foundational aspect of any programming language. In this section, you will gain the knowledge necessary to add comments to your Python code with finesse. Comments serve as the narrative thread woven through your code, elucidating its intricacies and aiding both present and future developers in comprehending your programming masterpiece.

A4: Concept of List (Array)

Python's prowess shines through its data structures, and in this section, I will introduce you to the concept of lists, analogous to arrays in other programming languages. You will delve into the creation, manipulation, and access of elements within a list, much like an artist selecting the perfect palette of colors to bring their vision to life.

A5: Concept of Append

The act of appending elements to lists is a common operation in Python. We will delve into the 'append' method, which serves as the brushstroke of dynamic addition, allowing you to infuse vitality into your lists, much like an artist adding detail to a canvas.

A6: Concept of POW()

Mathematics forms the foundation of programming, and we will explore one of Python's mathematical tools: the 'POW()' function. This function serves as your mathematical brush, empowering you to create powerful calculations and bring your numerical concepts to life.

A7: Execution of Loops

Loops are the rhythm of automation, fundamental for implementing repetitive tasks in your code. I will guide you through the execution of loops in Python, encompassing both the 'for' and 'while' loops. These loops are your conductor's baton, performing complex patterns of action.

A8: Modules in Python

Python's adaptability is a defining feature, and in this section, you will discover the art of harnessing modules. Modules grant you access to a treasury of pre-existing functionality, extending Python's capabilities like an artist exploring a palette of new colors.

A9: Concept of a Function

Functions are the architects of structured code, enhancing modularity and code management. I will introduce you to the concept of functions, illuminating how to define and wield them effectively. Functions act as your code's building blocks, enabling the creation of intricate programs with ease.

Upon concluding this chapter, you will emerge with a sturdy foundation in Python programming, prepared to tackle a broad spectrum of programming challenges. This foundation serves as the canvas upon which your Python journey unfolds, allowing you to craft intricate algorithms, construct dynamic applications, and explore more advanced topics in subsequent challenges. With enthusiasm, let us embark on this exciting Python adventure together!

A1: Basic Features

Python's appeal lies in its straightforwardness and adaptability, making it a favored choice for programmers of all levels. Here, we will examine the key components of Python, from its clean and easily comprehensible syntax to its dynamic typing and extensive standard library.

What is Python?

Python stands out as a high-level, adaptable, and interpreted programming language. Its origins trace back to Guido van Rossum in 1991, and it is celebrated for its simplicity, readability, and user-friendly syntax—qualities that make it an optimal starting point for those who are new to programming.

Readability and Clean Syntax:

Python's syntax is thoughtfully designed for clarity and conciseness, placing a premium on readability. This means you can craft code that is readily comprehensible, an invaluable asset for individuals entering the realm of coding.

Interpreted Language:

Python operates as an interpreted language, removing the need for code compilation before execution. This enables swift code creation and testing, rendering it an exceptional choice for rapid development.

Cross-Platform Compatibility:

Python boasts compatibility across a broad spectrum of platforms, including Windows, macOS, and Linux. This cross-platform versatility allows seamless code transfer between systems with minimal adjustments.

Extensive Standard Library:

Python comes equipped with an extensive standard library containing modules and packages for diverse tasks, ranging from data manipulation to web development and network protocol management. This expansive library streamlines numerous programming endeavors.

Community and Ecosystem:

Python enjoys a thriving and active developer community. This collective has generated a rich array of third-party libraries and frameworks, further enhancing Python's capabilities. Noteworthy libraries include NumPy for numerical computation, pandas for data analysis, Django for web development, and TensorFlow for machine learning.

Versatility:

Python's versatility transcends domains, spanning web development, data science, machine learning, automation, scientific computing, game development, and more. Its adaptability positions it as a versatile tool for a myriad of projects.

Open Source and Free:

Python operates as an open-source language, readily accessible to all without any financial barrier. You can obtain and install Python free of charge from its official website (python.org).

Ideal for Beginners:

Python's user-friendliness and abundant learning resources make it an optimal choice for programming novices. Countless tutorials, educational materials, and online courses are available to facilitate your initial steps.

Career Opportunities:

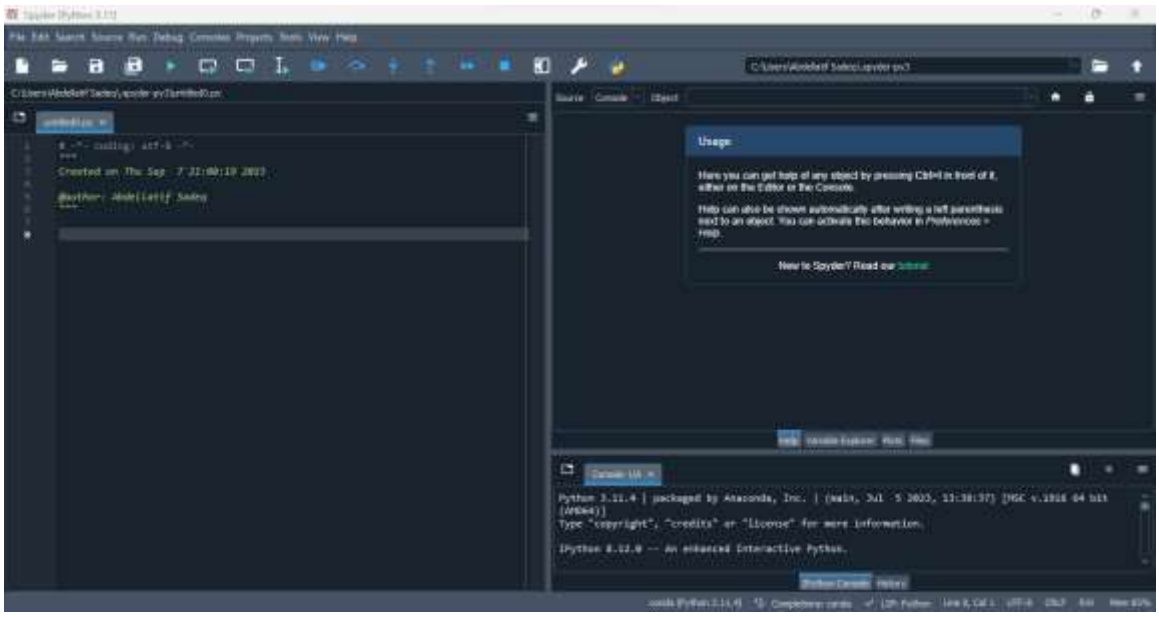
Proficiency in Python opens doors to diverse career prospects in software development, data analysis, machine learning, and beyond. Numerous businesses and institutions rely on Python, creating a demand for skilled Python developers.

In summary, Python stands as an approachable, versatile, and potent programming language with an active community and wide-ranging applications. Whether your interests lie in web development, data manipulation, or delving into artificial intelligence, Python serves as an outstanding starting point for newcomers keen to explore the programming universe.

A2: Installation of Anaconda and Spyder User Interface

Anaconda is a popular data science platform and Python distribution that streamlines the administration of Python libraries and environments. It provides features such as package management, isolated workspaces, and includes tools like Jupyter notebooks for tasks related to data analysis and scientific computing. Anaconda is the preferred option for many data scientists and researchers. To install Anaconda, refer to the link: https://www.anaconda.com/

Spyder is a specialized integrated development environment (IDE) commonly used with Anaconda for scientific Python programming. It offers features like Jupyter notebook integration, a variable explorer for data analysis, code analysis and debugging tools, and customization options. Spyder is a preferred choice for data scientists and researchers, providing a powerful and integrated environment for Python development and data-centric tasks. After installing Anaconda, type "Spyder" in the search bar to launch it and start coding. The graphical user interface (GUI) of Spyder looks as follows:



The left window is used for writing codes and scripts, whereas the right upper window is used for seeking help, showing the used variables and the generated plots, in addition to navigating through the directory. The right lower window (console) is used for showing the results and text outputs of the executed code.

A3: Execution of Comments

Comment is a line in the code that is never executed. Python incorporates two primary comment formats:

Single-line Comments: Single-line comments commence with a hash symbol `#`. Any content following the `#` on the same line is treated as commentary.

```
# This serves as a single-line comment
print("Hello, world!")  # This, too, is a comment
```

Multi-line Comments: Although Python lacks a dedicated multi-line comment syntax, you can create multi-line commentary by enclosing text within triple quotes ('''). While technically strings, they are frequently employed for multi-line comments.

```
'''
This constitutes a multi-line comment
utilizing triple-quotes.
'''
```

Comments are inserted in Python code to offer clarification and context. They elucidate variable purposes, elucidate code section logic, or document function operation. Well-commented code fosters comprehension and maintainability. Best practices for commenting in Python encompass several key principles. Firstly, brevity is advised; comments should be concise, conveying essential information without unnecessary elaboration. Secondly, it is crucial to maintain comment accuracy, ensuring that comments are updated whenever code changes occur to preserve correctness. Redundant comments should be avoided, focusing instead on explaining non-obvious or complex elements of the code. Employing meaningful variable and function names is paramount; clear and descriptive labels can reduce the need for extensive commentary. Lastly, consider commenting not only individual lines but also complete code blocks when necessary for comprehensive understanding. Importantly, Python comments have no impact on program execution, as the Python interpreter completely disregards them. This means you can freely add as many comments as needed to enhance code comprehension without affecting program performance.

As an illustration:

```
# This comment provides insight into the forthcoming code's intent
x = 5  # Setting the value of variable x
y = 10  # Setting the value of variable y
sum_result = x + y  # Computing the sum of x and y
print("The sum is:", sum_result)  # Displaying the result
```

In this code snippet, it is important to understand that comments in Python are exclusively intended for human comprehension and have no impact on the execution of the program. Python processes lines of code without taking into account any preceding `#` symbols. In essence, comments in Python play a vital role as essential tools for code documentation, enhancing code readability, and providing insights into the functionality of the program. It is crucial to note that comments do not affect how the Python interpreter executes the program.

Prudent and thoughtful use of comments is a hallmark of good programming practice, contributing significantly to the creation of code that is not only easier to maintain but also more readily understandable for both the original programmer and any collaborators. By documenting your code effectively through comments, you enable yourself and others to navigate and comprehend the intricacies of the program, simplifying the debugging process, and facilitating future modifications or improvements.

Furthermore, comments in Python offer a means to articulate the reasoning behind specific code choices, serving as a form of communication within the programming community. They can convey insights into the logic, intentions, and potential pitfalls of the code, assisting both current and future developers in grasping the intricacies of the code.

In summary, comments in Python serve as valuable tools in the development and upkeep of high-quality code. Their sole purpose is to enhance human understanding and documentation, making your programs more transparent and accessible to those who work with them.

A4: Concept of List (Array)

In Python, a list is an inherent data structure utilized for storing collections of items. It is an ordered and modifiable data type, allowing you to create a list of elements, modify its content, and retrieve elements based on their positions or indices. Lists are created by enclosing a sequence of items within square brackets [ ] and separating them with commas. For example:

```
sample_list = [1, 2, 3, 4, 5]
```

When working with lists in Python, it is crucial to retrieve individual elements by specifying their position or index within the list. Python employs square brackets [] for this purpose, with indexing starting at 0 for the first element. Here is a breakdown:

```
my_list = [10, 20, 30, 40, 50]

# Obtaining elements by index
first_element = my_list[0]  # Retrieves the first element (10)
third_element = my_list[2]  # Retrieves the third element (30)
```

Negative indexing is also possible, allowing counting from the end of the list. For instance, -1 corresponds to the last element, -2 to the second-to-last, and so forth:

```
last_element = my_list[-1]  # Retrieves the last element (50)
```

The colon operator, used for list slicing, follows the format start:stop:step, where:

- start: Denotes the starting index (inclusive) of the slice. If omitted, it defaults to the list's beginning.
- stop: Specifies the concluding index (exclusive) of the slice. If omitted, it defaults to the list's end.
- step: Governs the step size that defines the interval between elements. If omitted, it defaults to 1.

Slicing a list to select elements from index 1 to 3 (inclusive of 1 but exclusive of 3):

```
my_list = [10, 20, 30, 40, 50]
selected_elements = my_list[1:3]
# selected_elements contains [20, 30]
```

Slicing with a specified step to pick every second element from the list:

```
my_list = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
every_second_element = my_list[::2]
# every_second_element contains [0, 2, 4, 6, 8]
```

Slicing in reverse order to retrieve elements from the end of the list:

```
my_list = [10, 20, 30, 40, 50]
reversed_list = my_list[::-1]
# reversed_list contains [50, 40, 30, 20, 10]
```

The colon operator can be used to extract elements within a specified index range, which is particularly useful when you intend to derive a portion of a list based on a known index range. For instance:

```
my_list = ["apple", "banana", "cherry", "date", "fig"]
selected_fruits = my_list[1:4]
# selected_fruits contains ["banana", "cherry", "date"]
```

In this scenario, the slice [1:4] selects elements at indices 1, 2, and 3, effectively capturing the fruits between indices 1 and 4.

In summary, the colon operator in Python is indispensable for the precise slicing of lists, facilitating the selection of specific elements based on their indices. It empowers efficient data extraction and manipulation, making it an invaluable tool for various list-related operations.

A5: Concept of Append

In Python, "append" is a fundamental operation used to add elements to the end of a list, allowing lists to dynamically expand to accommodate additional data. This operation is performed using the built-in append() method designed specifically for lists.

The syntax for using "append" is as follows:

```
my_list.append(element)
```

- my_list: This designates the list to which the specified element will be appended.
- element: Represents the item that you intend to add to the list.

For example, adding an element to an empty list:

```
fruits = []  # An empty list
fruits.append("apple")
# fruits now holds ["apple"]
```

Incorporating multiple elements into a list through iteration:

```
numbers = []  # An empty list
for i in range(1, 4):
    numbers.append(i)
# numbers comprises [1, 2, 3]
```

Extending a list with elements of various data types:

```
data = [10, "hello"]
data.append(True)
# data contains [10, "hello", True]
```

Merging lists by appending one list to another:

```
list1 = [1, 2, 3]
list2 = [4, 5]
list1.append(list2)
# list1 encompasses [1, 2, 3, [4, 5]]
```

A6: Concept of POW()

The Python pow() function embodies the concept of exponentiation, allowing you to raise a number to a designated exponent or power. This built-in mathematical function streamlines the calculation of such operations. Its usage involves specifying two arguments: the base number and the exponent, yielding the result of the base raised to the indicated power.

The pow() function is deployed as follows:

```
pow(base, exponent)
```

- base: Signifies the base number intended for exponentiation.
- exponent: Represents the exponent or power to which the base number is elevated.

For example: Determining the result of 2 raised to the power of 3:

```
result = pow(2, 3)
# result holds 8 (2^3 = 8)
```

Computing the square of a number via pow():

```
num = 5
square = pow(num, 2)
# square holds 25 (5^2 = 25)
```

Employing pow() to calculate fractional powers:

```
x = 2
y = 0.5
result = pow(x, y)
# result holds 1.4142135623730951 (Square root of 2)
```

Utilizing pow() for negative exponents:

```
base = 3
exponent = -2
result = pow(base, exponent)
# result holds 0.1111111111111111 (1 / 3^2 = 1 / 9)
```

A7: Execution of Loops

In Python, loops serve as control structures enabling the repetitive execution of a specified block of code. They are indispensable for automating recurring tasks, cycling through data collections, and conducting operations until certain conditions are met. Python offers two primary loop types: the ‘for’ loop and the ‘while’ loop.

‘for’ Loops:

A for loop is utilized when the goal is to iterate over a sequence, be it a list, tuple, string, or any iterable object. It triggers the execution of a code block for each item in the sequence.

```
fruits = ["apple", "banana", "cherry"]
for fruit in fruits:
    print(fruit)
```

In this example, the code sequentially processes each fruit in the list and prints them one by one.

It is also possible to use range() with a for loop:

```
for i in range(5):
    print(i)
```

This code snippet employs a for loop to iterate from 0 to 4 (inclusive) and print each number.

‘while’ Loops:

A while loop is applied when a specific block of code needs to execute as long as a defined condition remains ‘True’. It continues executing until the condition becomes ‘False’.

```
count = 0
while count < 5:
    print(count)
    count += 1
```

In this example, the code prints numbers from 0 to 4 since the loop continues as long as the count remains less than 5.

Loop Control Statements:

- 'break' Statement:

```
numbers = [1, 2, 3, 4, 5]
for num in numbers:
    if num == 4:
        break
    print(num)
```

The break statement terminates the loop when num becomes equal to 4.

- 'continue Statement:

```
numbers = [1, 2, 3, 4, 5]
for num in numbers:
    if num == 3:
        continue
    print(num)
```

The continue statement skips the iteration when num equals 3 and proceeds to the next iteration.

Nested 'for' Loop:

The following code showcases a nested for loop that explores various combinations of i and j.

```
for i in range(3):
    for j in range(2):
        print(i, j)
```

In summary, loops in Python play a vital role in automating repetitive tasks, facilitating data iteration, and enabling efficient programming. 'for' loops are used for sequential iteration. On the other hand, 'while' loops handle conditional iteration. Loop control statements such as break and continue offer fine-grained control over loop execution, and nested loops can be employed to work with complex data structures. A comprehensive grasp of loops is essential for effective Python programming.

A8: Modules in Python

The Python standard library includes a diverse array of built-in modules that expand the capabilities of the language. These modules span a wide spectrum of domains, encompassing mathematical computations, file manipulation, networking operations, and more. Importing these modules enables you to utilize their functions and classes, thereby enhancing the functionality of your Python programs. Here, some fundamental modules are presented, elucidate their purposes, and detail the import process:

'math' Module:

The math module is designed to offer a plethora of mathematical functions and constants. It proves invaluable for a range of mathematical tasks, including trigonometric calculations, logarithmic operations, and advanced mathematical computations.

You can incorporate the math module into your Python code using the following import statement:

```
import math
```

By doing so, you gain access to the mathematical functions and constants provided by the math module, augmenting the mathematical capabilities of your Python program. An illustrative example of how to use the math module in Python to calculate the square root of a number is as follows:

```
import math

# Define a number
num = 25

# Calculate the square root using the math.sqrt() function
square_root = math.sqrt(num)

# Print the result
print(f"The square root of {num} is {square_root}")
```

In this example, we import the math module and then use the math.sqrt() function to compute the square root of the number 25. The result is printed to the console.

‘matplotlib’ Module:

The matplotlib module is a versatile library designed for crafting visualizations, encompassing charts, plots, and graphs. It is extensively employed in tasks related to data visualization and scientific plotting.

To harness the capabilities of the matplotlib module, you can import it along with commonly used submodules like pyplot using the following import statement:

```
import matplotlib.pyplot as plt
```

Creating a basic line plot with the matplotlib library:

```
import matplotlib.pyplot as plt

x_values = [1, 2, 3, 4, 5]
y_values = [10, 15, 13, 18, 20]

plt.plot(x_values, y_values)
plt.xlabel('X-axis')
plt.ylabel('Y-axis')
plt.title('Simple Line Plot')
plt.show()
```

In this Python code snippet, we make use of the matplotlib.pyplot module to effortlessly produce a simple line plot. This module stands as a versatile and extensively employed tool that proves indispensable for both data scientists and programmers. It greatly facilitates the creation of various types of visualizations.

The code can be dissected into several pivotal components that contribute to the overall structure and appearance of the line plot. To start, we employ the matplotlib.pyplot module and assign it the concise alias 'plt', a measure that enhances the clarity of the code.

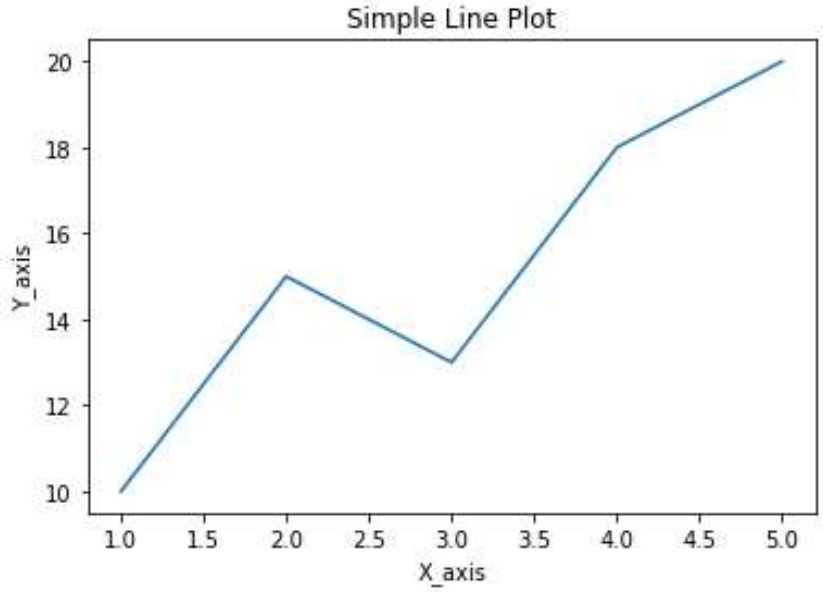
The heart of the plot is defined by the data furnished for the X and Y axes. Within this context, 'x' and 'y' denote two lists that represent the data points for the X and Y axes of the plot, respectively. These lists offer a high degree of flexibility and can be adjusted to accommodate specific datasets, thereby enabling the creation of tailored visualizations.

To give rise to the plot itself, we make use of the 'plt.plot(x, y)' function. This function takes the data from 'x' and 'y' and dynamically generates the plot accordingly. Its adaptability extends to encompass various customization options, encompassing line style, color, and marker types, permitting the fulfillment of precise visualization requirements.

In order to enhance the clarity of the plot, we introduce labels and a title. By employing functions like 'plt.xlabel()', 'plt.ylabel()', and 'plt.title()', we are able to assign labels to the X-axis and Y-axis, in addition to providing an overarching title for the plot. These customizations serve the vital purpose of conveying the plot's intent and context to viewers with utmost clarity.

Finally, to bring the plot to life visually, we incorporate 'plt.show()' into the code. The significance of this function lies in its role as the trigger for rendering the plot as a visual output on the screen. In its absence, the plot would remain concealed from view.

Upon executing the code, a line plot materializes, its foundation rooted in the provided data points. The labels and title significantly elevate the plot's interpretability, enabling viewers to swiftly grasp its significance. The capacity to customize both the data and labels empowers users to craft a diverse array of plots tailored precisely to their specific needs in data analysis, visualization, and presentation.



‘numpy’ Module:

The numpy module stands as a fundamental component for numerical computing in Python. It equips users with the ability to work with arrays, matrices, and execute mathematical operations on them. It finds widespread application in scientific and data analysis contexts.

To make use of the numpy module, the customary import statement is as follows, often adopting the shorthand alias ‘np’ for conciseness:

```
import numpy as np
```

Demonstrating basic array operations with the numpy library:

```python
import numpy as np

array1 = np.array([1, 2, 3, 4, 5])
array2 = np.array([5, 4, 3, 2, 1])

# Element-wise addition
result_addition = array1 + array2

# Element-wise multiplication
result_multiply = array1 * array2

print("Addition Result:", result_addition)
print("Multiplication Result:", result_multiply)
```

This code showcases how numpy can be utilized for element-wise array operations.

'scipy' Module:

The scipy module extends the capabilities of numpy by providing supplementary scientific and technical functionality. It encompasses areas such as optimization, integration, interpolation, signal processing, and more, making it indispensable in advanced scientific computations.

To incorporate the scipy module into your code, you can import it along with its submodules tailored to your specific requirements. For example:

```python
from scipy import optimize, signal
```

Solving a basic optimization problem using the scipy library:

```python
from scipy import optimize

# Define a quadratic function for optimization
def quadratic_function(x):
    return (x - 3) ** 2 + 5

# Minimize the function
result = optimize.minimize(quadratic_function, x0=0)

print("Optimal Solution:", result.x)
print("Minimum Value:", result.fun)
```

A9: Concept of a Function

In Python, a function represents a fundamental component of code that allows you to encapsulate a set of instructions within a reusable unit. Functions serve as the building blocks for structuring and organizing code, fostering code reusability, and enhancing code readability and maintainability. They enable the decomposition of intricate tasks into smaller, manageable segments.

Here is an introduction to Python functions with illustrative examples:

Defining a Function

To define a function in Python, you employ the 'def' keyword, followed by the function name enclosed in parentheses. Any input parameters, often referred to as arguments, are specified within these parentheses, and a colon ':' denotes the end of the function declaration line. Subsequently, a code block follows, indented with spaces or tabs, containing the function's set of instructions.

Consider a straightforward function that extends a greeting to the user:

```
def greet(name):
    """This function extends a greeting to the user."""
    print(f"Hello, {name}!")

# Invoking the function
greet("Alice")
```

The output window will show the following:

```
In [5]: runfile('C:/Users/Abdellatif Sadeq/.spyder-py3/untitled0.py', wdir='C:/Users/Abdellatif
Sadeq/.spyder-py3')
Hello, Alice!
```

In this example:

- def greet(name): defines a function named greet that accepts one parameter, name.
- The triple-quoted string immediately following the function definition serves as a docstring, offering documentation for the function.
- print(f"Hello, {name}!") comprises the code executed when the function is invoked.

Invoking a Function

To execute a function and make use of its defined functionality, you invoke it by its name, succeeded by parentheses. If the function requires arguments, you supply values for these arguments inside the parentheses.

In the preceding example, greet("Alice") invokes the greet function with the argument "Alice".

Returning Values

Functions can also yield results through the return statement. When a function produces a value, you can assign it to a variable or apply it in other sections of your code.

Here is an example of a function that computes the square of a number and returns the outcome:

```
def square(x):
    """This function furnishes the square of a number."""
    return x * x

result = square(5)
print(result)  # Result: 25
```

In this scenario, square(5) yields 25, which is subsequently assigned to the variable result.

Default Arguments

Python permits you to establish default values for function parameters. When a value is not supplied during the function call, the default value takes precedence.

```
def greet(name="Guest"):
    """This function welcomes the user with a predetermined name."""
    print(f"Hello, {name}!")

greet()          # Result: Hello, Guest!
greet("Alice")   # Result: Hello, Alice!
```

Function Documentation

A commendable practice entails embedding docstrings within your functions to deliver precise documentation concerning their intent and application. You can access the docstring by using the help() function or by inputting the function name followed by a question mark in numerous Python IDEs or interactive environments.

```
help(greet)
# Result:
# This function welcomes the user with a predetermined name.
```

Chapter B: Application of Python in Mechanical Engineering

In the previous chapter, we ventured into the foundational aspects of Python, providing you with the essential tools necessary to navigate the programming landscape. As we progress deeper into this book, we embark on an exciting exploration—a journey into the practical applications of Python within the field of mechanical engineering.

Chapter B serves as the crucible where theory converges with practice, where the code you mastered in Chapter A evolves into concrete solutions for real-world mechanical challenges. Within the confines of this chapter, you will witness Python's transformation from mere syntax into a formidable ally, augmenting your capabilities as a mechanical engineer.

The applications we delve into encompass a range of complexities and practical relevance. Each selection is meticulously curated to offer you a comprehensive insight into how Python can be employed to address engineering dilemmas. From the aerodynamics of a cyclist contending with the forces of the wind to the intricate dynamics of a robotic arm in motion, we traverse the spectrum of mechanical engineering, showcasing Python's adaptability at every turn.

However, this chapter is more than a mere collection of theoretical examples. It represents an entrance into hands-on learning. As you progress, you will have the opportunity to engage in simulations, analyses, and projects that mimic real-world engineering scenarios. These practical exercises not only solidify your comprehension but also empower you to employ Python's computational prowess in your personal mechanical engineering pursuits.

With this Python toolkit fully stocked, we will delve into air resistance analysis, simulate intricate robotic motions, dissect the intricacies of the Otto cycle, and even explore pressure computations utilizing advanced numerical techniques. Along this path, we will uncover the intricacies of data analysis, curve fitting, and visualization, equipping you with the skills to derive meaningful insights from your engineering data.

So, prepare for a thrilling voyage into the world of Python-enhanced mechanical engineering. Whether you are an experienced engineer seeking to expand your skill set or a student eager to bridge the divide between theory and practice, Chapter B offers an exhilarating leg of your Python programming journey. Welcome to the realm where code intersects with mechanical marvels, and where your Python adventure soars to new heights.

B1: Analyzing Cyclist Air Resistance: Velocity, Drag Coefficient, and Performance

Write a Python program to calculate drag force against a cyclist.

The factors involved are Velocity, Drag coefficient, Frontal Area, Density of air.

1. Write a program to plot Velocity vs. Drag force.

2. Write a program to plot Drag Co-efficient vs. Drag force.

Hint: For the same frontal area, the coefficient of drag varies for different shapes. Look for values of drag coefficient for various shapes in the internet.

Solution

Objective

The goal of this challenge is to write a Python program that computes and presents the air resistance encountered by a cyclist under different conditions. This program will consider variables like speed, drag coefficient, frontal area, and air density. The main objective involves generating graphical representations that showcase how velocity relates to drag force and how drag coefficient influences drag force. By doing so, the program aims to offer valuable understanding into how a cyclist's aerodynamic performance is influenced by various factors affecting drag force.

Problem Statement

Cyclists regularly encounter the formidable challenge of countering air resistance, a factor that holds significant sway over their speed and overall efficiency. In pursuit of an enhanced comprehension of their own capabilities and the means to refine their performance, it becomes crucial to accurately quantify the drag force that opposes their motion. The core objective of this endeavor is to surmount this challenge by crafting a Python program that undertakes the calculation and visualization of the drag force experienced by a cyclist. This computation hinges on several pivotal variables: velocity, drag coefficient, frontal area, and air density.

The program's blueprint encompasses the following paramount goals:

Calculation of Drag Force: A central facet involves the precise determination of the drag force applied to a cyclist. This computation is governed by a comprehensive formula:

$$F_D = \frac{\rho * A_f * V^2 * C_d}{2} \qquad (1)$$

Where,

$\boldsymbol{F_D}$ is the drag force (N)

$\boldsymbol{\rho}$ is the density of the surrounding medium in which the object is moving in (kg/m^3)

$\boldsymbol{A_f}$ is the frontal area of the object (m^2)

$\boldsymbol{V}$ is the object velocity (m/s)

$\boldsymbol{C_d}$ is the drag coefficient

Here, the drag coefficient signifies the aerodynamic characteristics of the cyclist's posture, the frontal area encapsulates the effective cross-sectional surface, and air density pertains to the medium through which the cyclist propels. By synthesizing these factors, the program achieves a robust calculation of the exerted drag force. Figure 1 shows the forces acting on a cyclist while moving.

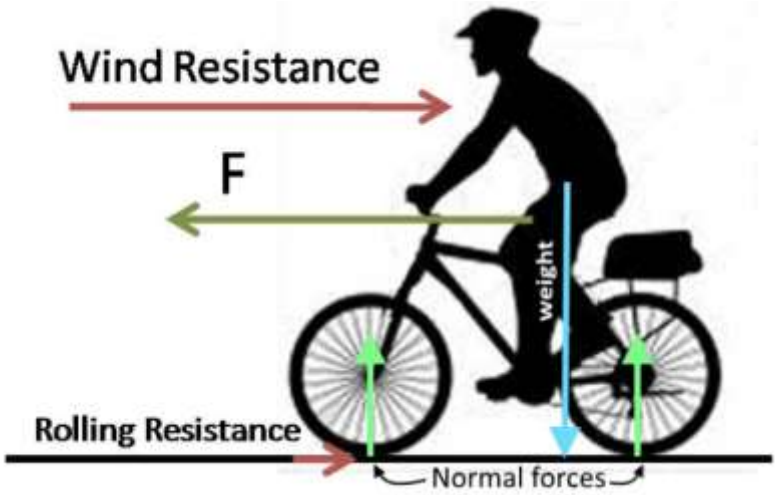


Figure 1: Forces exerted on a cyclist during motion. Available at: https://mcenteegerentow1942.blogspot.com/2022/03/knowing-all-forces-exerted-on-object.html

Velocity-Drag Force Relationship Plot: An integral component of the program's functionality is the generation of a graphical representation that elucidates the intricate interplay between velocity and drag force. This visual artifact facilitates cyclists in comprehending the dynamic alterations in drag force as velocities oscillate.

Drag Coefficient-Drag Force Relationship Plot: A complementary visualization is crafted to spotlight the correspondence between the drag coefficient and the resultant drag force. This task entails the incorporation of drag coefficient values garnered from reliable sources, catering to a spectrum of diverse shapes. The inherent variability of the coefficient, contingent on the cyclist's aerodynamic profile, is thus ingeniously captured.

Through the confluence of these analytical and graphical elements, the program furnishes cyclists with an invaluable instrument for unraveling the intricate relation of these

variables. Armed with this insight, they are empowered to orchestrate their riding tactics, discern optimal equipment selections, and finesse body positioning to mitigate the impact of drag force, thereby amplifying their overall performance. This initiative anticipates a substantive contribution to the enhancement of cycling prowess, accomplished through an empirically grounded comprehension of the forces that underlie aerodynamics.

Solution Procedure

Step 1: Importing Required Libraries Start by importing the essential Python libraries crucial for numerical computations and the visualization of data.

Step 2: Defining Constants and Input Variables Proceed by establishing the unchanging constants and input variables pivotal for conducting the ensuing calculations.

Step 3: Computing Drag Force Employ the provided formula to ascertain the drag force linked to each velocity, propelling the analysis forward.

Step 4: Plotting Velocity against Drag Force Construct a graphical representation that graphically elucidates the connection between velocity and the exerted drag force.

Step 5: Plotting Drag Coefficient against Drag Force In this stage, diligent research is imperative to uncover fitting drag coefficient values correlated with distinct shapes. Let us posit that you possess an assemblage of drag coefficients aligned with disparate shapes.

Step 6: Executing the Program Put the plan into action by executing the program, enabling the visualization of the generated plots to transpire.

Step 7: Analysis and Interpretation Undertake the process of studying the generated plots, with the aim of gaining insight into the fluctuation of drag force relative to diverse velocities and comprehending how varying drag coefficients engendered by different shapes impact the resultant drag force.

Python Code

```
# Step 1: Importing Required Libraries
import matplotlib.pyplot as plt
import numpy as np
```

Used for importing the essential libraries required for plotting and numerical calculations.

```
# Step 2: Defining Constants and Input Variables
C_d = 0.6
A_f = 0.5
rho = 1.2
V = np.linspace(0, 40, 100)
```

Next, establish the unchanging constants and input variables crucial for coming calculations

```
# Step 3: Computing Drag Force
F_D = 0.5 * C_d* A_f * rho * pow(V,2)
```

Proceed to compute the drag force, employing the given formula.

```
# Step 4: Plotting Velocity against Drag Force
plt.plot(V, F_D, label='Drag Force')
plt.xlabel('Velocity (m/s)')
plt.ylabel('Drag Force (N)')
plt.title('Velocity vs Drag Force for Cyclist')
plt.grid()
plt.show()
```

Construct a graphical representation illustrating the correlation between velocity and the encountered drag force.

```
# Step 5: Plotting Drag Coefficient against Drag Force at Constant Velocity
c_d_values = np.linspace(0.1, 1.0, 10)  # Choose drag coefficients to plot
plt.figure(figsize=(10, 6))
for c_d_value in c_d_values:
    drag_force_shape = 0.5 * c_d_value * A_f * rho * constant_velocity**2
    plt.plot(c_d_value, drag_force_shape, marker='o', label=f'C_d = {c_d_value:.2f}')
plt.xlabel('Drag Coefficient')
plt.ylabel('Drag Force (N) at V = ' + str(constant_velocity) + ' m/s')
plt.title('Drag Coefficient vs Drag Force at Constant Velocity')
plt.legend()
plt.grid()
plt.show()
```

Plot the drag force at different drag coefficients at a constant velocity, and customize the plot.

```
# Step 6: Executing the Program
```

This segment remains empty, signifying the conclusion of code execution, and it is given as a comment only.

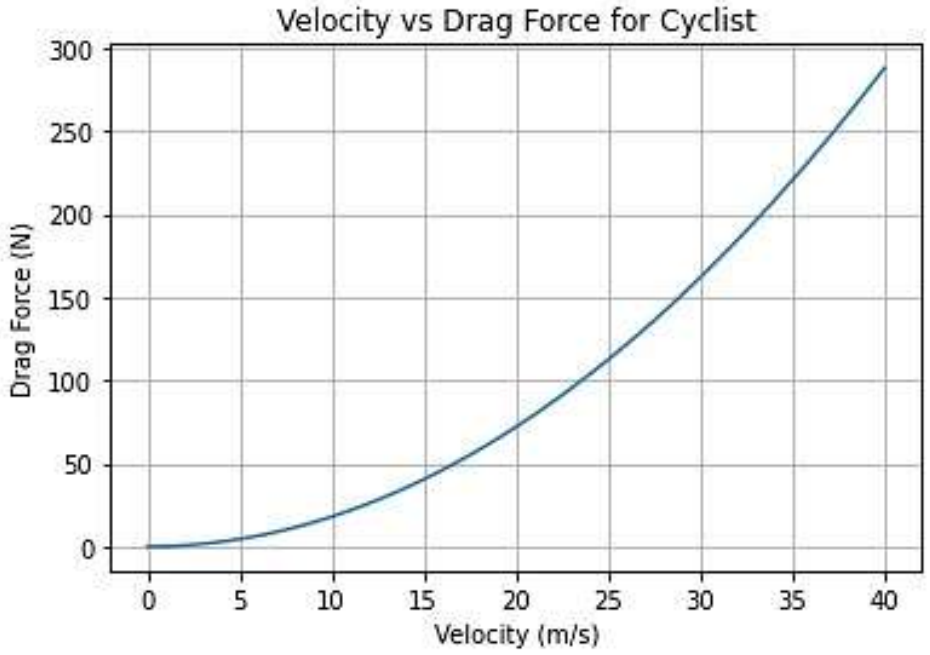
Step 7: Analysis and Interpretation

This is the final step, where the results are discussed. It is only given as a comment.

Results and Discussion

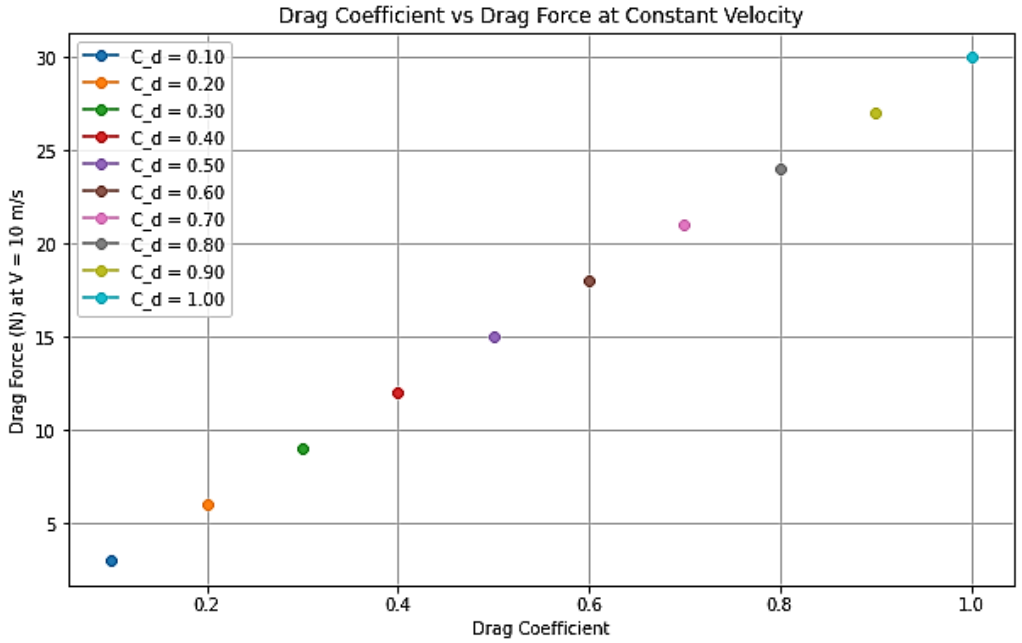
In this section, the obtained results are presented and discussed.

Velocity vs. Drag Force



The plot illustrates the relation between a cyclist's speed and the force of air resistance they encounter. With rising velocity, the drag force similarly escalates, aligning with established aerodynamic principles. This graphical representation showcases a distinctive upward curve, effectively showcasing the quadratic relationship between velocity and drag force. The plot's message is clear: as speed increases, so does the opposing force of air resistance. This insight empowers cyclists to appreciate the implications of their speed choices on the effort required to overcome resistance.

Drag Force vs. Drag Coefficients



The plot visualizes the effect of drag coefficients on drag forces when velocity remains constant. The x-axis portrays drag coefficient values, indicating how aerodynamic an object is, while the y-axis illustrates the drag forces experienced at the given constant velocity. Noteworthy observations include the direct correlation between higher drag coefficients and increased drag forces, underscoring the resistance encountered by less aerodynamic shapes. Conversely, lower drag coefficients result in decreased drag forces, underscoring the advantages of streamlined designs in reducing air resistance. Depending on the values employed, a linear pattern might materialize between drag coefficients and drag forces within specific ranges, implying a consistent connection within those intervals.

The plot also underscores the role of design and shape in determining aerodynamic efficiency, with objects featuring optimal drag coefficients experiencing diminished drag forces and enhanced efficiency. Additionally, the plot may hint at an optimal range of drag coefficients for minimizing drag forces at the chosen constant velocity, highlighting the delicate equilibrium between object shape and air resistance. Nonetheless, it is important to consider that this analysis solely pertains to drag force at a singular velocity, and a more comprehensive evaluation would encompass variables like varying velocities to holistically gauge overall performance.

B2: 2R Robotic Arm Kinematics Simulation: From Equations to Animation

Write a program in python to simulate the forward kinematics of a 2R Robotic Arm.

Create an animation file of the plot.

Upload an animation on YouTube and provide the YouTube link for an animation in your report.

Solution

Objectives

The challenge's aim is to develop a Python program for simulating the forward kinematics of a 2R Robotic Arm illustrated in Figure 2. This entails computing the manipulator position using input joint angles and arm lengths, and then graphically visualizing it using Matplotlib. The program will generate an animation by crafting a series of frames that depict the arm's movement with varying joint angles. Subsequently, this animation will be uploaded to YouTube, and a comprehensive report will be created to detail the project, featuring code, equations, and the YouTube link. This endeavor seeks to provide an educational resource to enhance understanding of how robotic arms behave.

Problem Statement

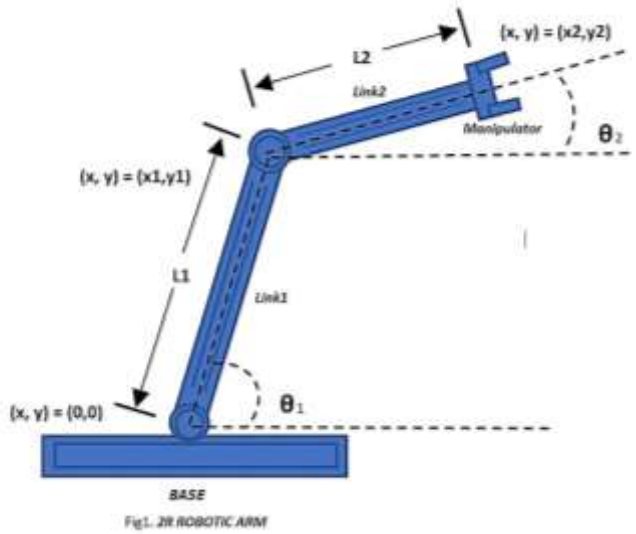


Figure 2: Forward kinematics of a 2R Robotic Arm. Available at: https://www.youtube.com/watch?v=_j8Glp3Yh0Q

The robotic arm shown in Figure 2, is a form of mechanical arm, often programmable, that can either constitute the entire mechanism or be a component within a more intricate robot. These arms comprise links connected by joints that facilitate rotational or linear motion. The links collectively form a kinematic chain, with the chain's end point termed the end effector. This effector, situated at the arm's extremity, is designed for interaction with the surroundings. Kinematic analysis pertains to the relationships among the manipulator's link positions, velocities, and accelerations. It is categorized into direct kinematics and inverse kinematics. In forward kinematics, provided the length of each link and joint angle, the objective is to compute any point's position within the robot's workspace. In inverse kinematics, given the link lengths and a point's position, the aim is to determine the joint angles. The project focuses on a 2R robotic arm with two rotational links operating in a 2D plane, each with a single degree of freedom.

Based on the dimensions shown in Figure 2, and using trigonometry:

$$x_0 = y_0 = 0 \quad (1)$$

$$x_1 = L_1 \cos\theta_1 \quad (2)$$

$$y_1 = L_1 \sin\theta_1 \quad (3)$$

$$x_2 = x_1 + L_2 \cos\theta_2 \quad (4)$$

$$y_2 = y_1 + L_2 \sin\theta_2 \quad (5)$$

The given equations depict the forward kinematics expressions for a two-link robotic arm. Using trigonometric principles, they ascertain the x and y coordinates of both the arm's joints and its end effector by considering the angles of its joints (θ_1 and θ_2) and the lengths of its links (L_1 and L_2). The starting position is established at the origin, and subsequent positions are computed by adding the horizontal and vertical parts of the link lengths, modified by the respective angles. These equations establish a clear connection between the angles of the joints and the resulting positions of the different elements of the robotic arm.

Solution Procedure

Step 1: Import Necessary Libraries Start by importing the required libraries, namely math for mathematical calculations and matplotlib.pyplot for plotting functionalities.

Step 2: Specify Arm Characteristics Define the parameters that determine the lengths of the robotic arm's two links, referred to as l1 and l2.

Step 3: Determine Theta Range Decide on the number of angles, termed n_theta, to be calculated. Establish the starting angle as theta_start and the concluding angle as theta_end.

Step 4: Generate Angle Values Iteratively generate sets of angles for both joints (theta1 and theta2 lists) within the predefined range. These angles will be integral for simulating the arm's movements.

Step 5: Initialize Coordinates Set the initial coordinates of the robotic arm's base, denoted as x0 and y0, to be located at the origin (0, 0).

Step 6: Iterative Calculations and Plotting For each combination of calculated angles (t1 and t2), compute the positions of the arm's joints and its end effector.

Determine the x and y coordinates of the initial joint (x1 and y1) based on the length of the first link and angle t1.

Calculate the x and y coordinates of the end effector (x2 and y2) by utilizing the joint positions and the second link's length along with angle t2.

Create filenames for plot images, increase the counter ct, and save the plots using plt.plot() and plt.savefig().

Step 7: Plot Settings Set specific boundaries for the x and y axes using plt.xlim() and plt.ylim() to ensure the accuracy of visualization.

Save each plot image under sequential filenames such as "1.png," "2.png," and so forth.

In essence, this procedure guides the code to systematically determine joint angles, compute corresponding coordinates, and generate graphical representations illustrating the robotic arm's various configurations. The resulting images capture the arm's motion and adjustments as it transitions through different angles.

Python Code

```
#Step 1: Import Necessary Libraries
import math
import matplotlib.pyplot as plt
```

These lines serve to import the required libraries: ‘math’ for mathematical calculations and ‘matplotlib.pyplot’ for the creation of visual plots.

```
#Step 2: Specify Arm Characteristics
l1 = 1.2
l2 = 0.6
```

In these lines, the extents of the two links within the robotic arm are defined, designated as l1 and l2.

```
#Step 3: Determine Theta Range
n_theta = 10
theta_start = 0
theta_end = math.pi/2
theta1 = []
```

In this part, the code determines the number of n_theta values to be generated to signify joint angles.

theta_start and theta_end define the span of angles, ranging from 0 to π/2.

```
theta2 = []

#Step 4: Generate Angle Values Iteratively
for i in range(0,n_theta):
  tmp = theta_start + i*(theta_end-theta_start)/(n_theta-1)
  theta1.append(tmp)
  theta2.append(tmp)
```

A loop is employed to generate sets of angles for both theta1 and theta2 in designated range.

```
#Step 5: Initialize Coordinates
x0 = 0
y0 = 0
```

These lines establish the starting coordinates of the robotic arm's base at (0, 0).

```
#Step 6: Iterative Calculations and Plotting
ct = 1
for t1 in theta1:
  for t2 in theta2:
    # Link1, link2 connector
    x1 = l1 * math.cos(t1)
    y1 = l1 * math.sin(t1)
    #Coordinate points of the manipulator
    x2 = x1 + l2 * math.cos(t2)
    y2 = y1 + l2 * math.sin(t2)
```

This section involves a nested loop iterating through the combinations of theta1 and theta2.

For each combination, the code calculates the positions of the robotic arm's joints and the end effector.

The coordinates x1, y1, x2, and y2 are determined using trigonometric functions.

```
#Step 7: Plot Settings
    filename = str(ct) + '.png'
    ct=ct+1
    plt.figure()
    plt.xlabel('x')
```

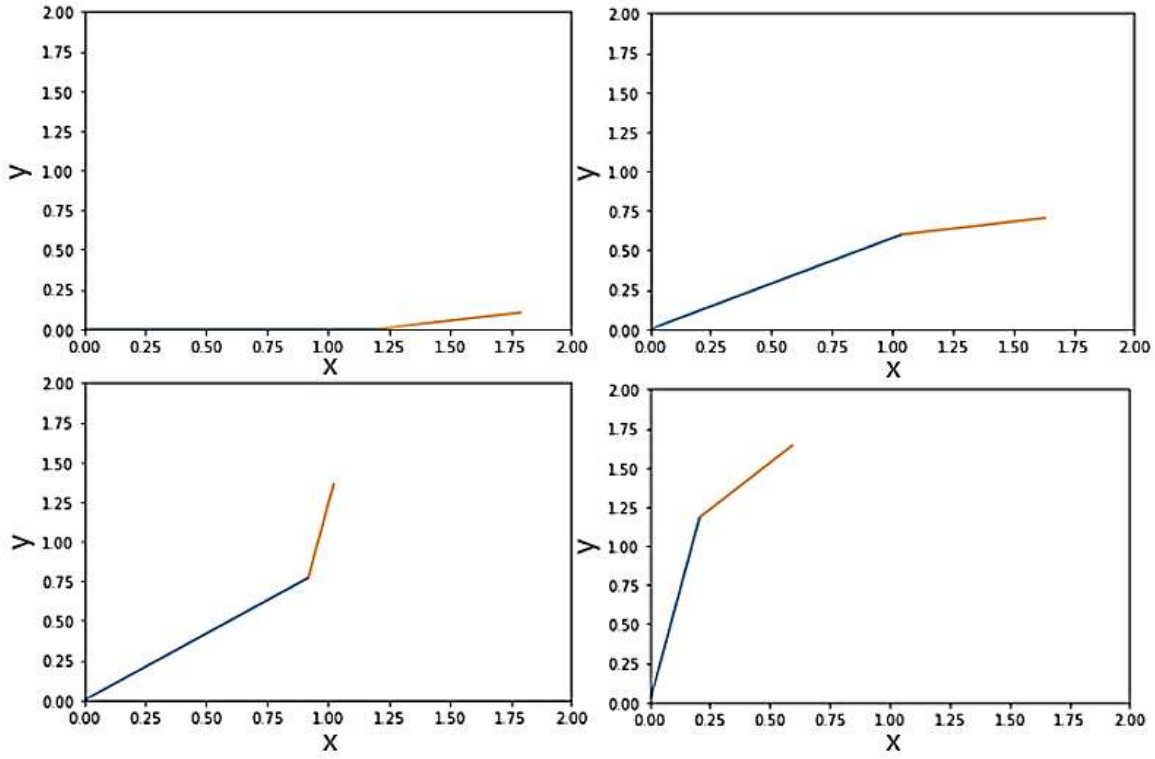
A unique filename is generated for the plot image, and this image is saved using plt.savefig().

plt.xlabel() and plt.ylabel() are employed to affix labels to the x and y axes of each plot, with "X Coordinate" and "Y Coordinate," respectively.

```
plt.ylabel('y')
plt.plot([x0,x1], [y0,y1])
plt.plot([x1,x2], [y1,y2])
plt.xlim([0,2])
plt.ylim([0,2])
plt.savefig(filename)
```

Results and Discussion

A sequence of images that represent the motion of the Robotic Arm at different time instants are shown below. The complete video animation is uploaded to the link: https://youtu.be/sZ5gKjAkvzo



The provided Python code generates a series of visual representations that intricately demonstrate how a 2R robotic arm's movement and structure evolve as its joint angles change. In each plot, the arm's key components – its base, pivotal joints, and end effector – are highlighted, offering a clear view of how modifications to the joint angles impact the arm's spatial orientation. As the joint angles shift, the arm's form takes on various configurations, which are elegantly portrayed as the plots progress. This dynamic interplay between angles and the arm's design results in a diverse range of poses and alignments. Beyond visual appeal, the sequence of plots provides an immersive exploration of the intricate realm of robotics and kinematics.

B3: Otto Cycle Simulation and Analysis: Efficiency and PV Diagram

Based on the concepts you have learnt during the forward kinematics routine, go ahead and write code that can solve an Otto cycle and make plots for it.

Here are the requirements

1. Your code should create a pressure-volume (PV) diagram

2. You should output the thermal efficiency of the engine.

Solution

Objective

Applying the understanding gained from the forward kinematics routine, create a program capable of solving an Otto cycle and producing relevant plots. The program should create a pressure-volume (PV) diagram, and compute the engine's thermal efficiency.

Problem Statement

The air standard Otto cycle shown in Figure 3 represents an idealized thermodynamic cycle that provides a simplified framework for understanding the functioning of spark-ignition internal combustion engines, commonly used in gasoline-powered vehicles.

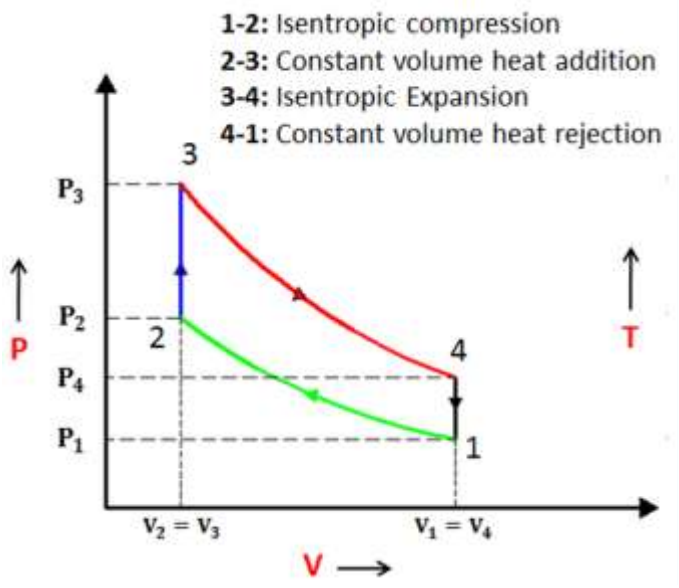


Figure 3: PV diagram of air standard Otto cycle. Available at: https://www.mechanicalbooster.com/2017/10/otto-cycle.html

Despite its idealized nature, the Otto cycle offers valuable insights into the fundamental thermodynamic principles underlying the performance of these engines. Comprising four primary strokes—intake, compression, combustion, and expansion—the cycle's theoretical processes offer a foundational model for analysis, even though real-world engines experience deviations due to factors such as heat transfer, friction, and combustion intricacies. The primary stage in the cycle are as follows:

Intake: The cycle starts with the intake stroke, where a mixture of air and fuel is drawn into the engine cylinder. This intake stage is often modeled as an isentropic compression, characterized by a rise in pressure and temperature due to the piston's motion.

Compression: Subsequent to intake, the compression stroke involves adiabatic and reversible compression of the air-fuel mixture. This process raises pressure and temperature, readying the mixture for combustion.

Combustion: During combustion, which occurs at constant volume, the compressed mixture ignites via a spark plug. This rapid combustion heightens temperature and pressure, leading to piston movement and energy conversion.

Expansion: Following combustion, the expansion stroke ensues, where hot gases undergo adiabatic and reversible expansion, propelling the piston upwards. This phase decreases pressure and temperature while extracting mechanical work from the engine.

The air standard Otto cycle, while simplifying real-world intricacies, offers a cornerstone for studying internal combustion engines. It provides a valuable baseline for evaluating engine efficiency, power output, and thermal behavior. Through the cycle, engineers and researchers analyze various operational and design parameters, contributing to the evolution of internal combustion engines and automotive technology.

The governing equations for an Otto cycle are:

Swept Volume in m^3:

$$V_s = \frac{\pi}{4} * d^2 * L \qquad (1)$$

Clearance Volume in m^3:

$$V_c = \frac{V_s}{r-1} \qquad (2)$$

Volume at Stage 1 (BDC) in m^3:

$$V_1 = V_4 = V_s + V_c \qquad (3)$$

Volume at Stage 2 (TDC) in m^3:

$$V_2 = V_3 = V_c \qquad (4)$$

Relation between Pressure (MPa) and Volume (m^3):

$$P_1 * V_1^{\gamma} = P_2 * V_2^{\gamma} \quad (5)$$

Ideal Gas Equation of State:

$$\frac{P_1 * V_1}{T_1} = \frac{P_2 * V_2}{T_2} \quad (6)$$

Instantaneous Volume (m^3):

$$V_\theta = V_c * \left(1 + \frac{r-1}{2}\right) * \left\{1 + R - \cos\theta - (R^2 - \sin^2\theta)^{\frac{1}{2}}\right\} \quad (7)$$

$$, R = 2 * \frac{L_{cr}}{L} \quad (8)$$

Instantaneous Pressure (MPa):

$$P_\theta * V_\theta^{\gamma} = P_2 * V_2^{\gamma} \quad (9)$$

Thermal Efficiency:

$$\eta = 1 - \frac{1}{r^{\gamma-1}} \quad (10)$$

Where,

$\boldsymbol{d}$ is the bore diameter

$\boldsymbol{L}$ is the stroke

$\boldsymbol{L_{cr}}$ is the connecting rod length

$\boldsymbol{r}$ is the compression ratio

$\boldsymbol{\gamma}$ is the specific heat ratio

$\boldsymbol{\theta}$ is the crank angle

Solution Procedure

The Python code should facilitate the analysis of an air standard Otto cycle within the context of an internal combustion engine. This program should simulate the fundamental thermodynamic phases inherent in the operation of a spark-ignition engine, comprising intake, compression, combustion, and expansion. Furthermore, the code should generate a PV diagram and evaluates the engine's thermal efficiency. Here is a breakdown of the main steps that should be encapsulated within the code:

Step 1: Import Libraries The process starts by importing essential libraries: math for mathematical operations and matplotlib.pyplot for generating plots.

Step 2: Define Engine Kinematics Function An engine kinematics function, denoted as engine_kinematics, is established. This function computes the instantaneous volume during piston movement, factoring in parameters such as bore, stroke, connecting rod length, compression ratio, and crank angles.

Step 3: Define Inputs Initial inputs are outlined, encompassing factors like the initial pressure p1, initial temperature t1, specific heat ratio gamma, final temperature t3, bore dimensions, stroke length, connecting rod measurements, and compression ratio.

Step 4: Calculate Initial and Final Volumes By utilizing the given bore, stroke, and compression ratio, the script computes the initial and final volumes: v1 and v2.

Step 5: State Point 2 Calculation The code determines the pressure at state point 2 (p2) based on the equation p1 * v1^gamma = p2 * v2^gamma. A value called rhs is calculated to aid in determining temperature t2.

Step 6: Characterize Compression Process Utilizing the engine_kinematics function, volume changes for the compression process are evaluated. Pressure values for this process are calculated through the adiabatic compression equation.

Step 7: Calculate for State Point 3 Using the relationship p2 * v2 / t2 = p3 * v3 / t3, the script derives the pressure at state point 3 (p3). Another variable, rhs, is determined for further temperature calculations.

Step 8: Characterize Expansion Process Similar to the compression process, the engine_kinematics function is utilized to ascertain volume changes during expansion. Pressure values for this process are determined using the adiabatic expansion equation.

Step 9: Calculate for State Point 4 The code computes the pressure at state point 4 (p4) through the relation p3 * v3^gamma = p4 * v4^gamma. Another variable, rhs, aids in calculating temperature t4.

Step 10: Plot PV Diagram A visual representation of the PV diagram, illustrating the compression, constant volume heat addition, expansion, and constant volume heat rejection processes, is generated using the matplotlib.pyplot library.

Step 11: Calculate and Display Thermal Efficiency The thermal efficiency of the cycle is determined using the formula eff = 1 - 1 / cr^(gamma - 1). This efficiency percentage is then presented in the output.

The code facilitates the simulation and analysis of an air-standard Otto cycle. It computes critical state variables, pressure-volume relationships, and thermal efficiency, displaying results visually through the PV diagram and textually for efficiency assessment.

Python Code

```
# Step 1: Import Libraries
import math
import matplotlib.pyplot as plt
```

The code starts by importing essential libraries. The math library aids in mathematical calculations, and matplotlib.pyplot is imported for plots visualization.

```
# Step 2: Define Engine Kinematics Function
def engine_kinematics(bore, stroke, con_rod, cr, start_crank, end_crank):
    a = stroke / 2
    R = con_rod / a
    v_s = (math.pi / 4) * pow(bore, 2) * stroke
    v_c = v_s / (cr - 1)
    sc = math.radians(start_crank)
    ec = math.radians(end_crank)
    num_value = 50
    dtheta = (ec - sc) / (num_value - 1)
    V = []
    for i in range(0, num_value):
        theta = sc + i * dtheta
        term1 = 0.5 * (cr - 1)
        term2 = R + 1 - math.cos(theta)
        term3 = pow(R, 2) - pow(math.sin(theta), 2)
        term3 = pow(term3, 0.5)
        V.append((1 + term1 * (term2 - term3)) * v_c)
    return V
```

A function named engine_kinematics is introduced to determine the instantaneous volume during piston movement within the engine. This function encompasses parameters like bore dimensions, stroke length, connecting rod size, compression ratio, and crank angles. It computes volume at various crank angles, leveraging kinematic principles.

```
# Step 3: Define Inputs
p1 = 101325
t1 = 500
gamma = 1.4
t3 = 2300
bore = 0.1
stroke = 0.1
con_rod = 0.15
cr = 9
```

The initial parameters for the simulation are outlined. These include p1 for initial pressure, t1 for initial temperature, gamma representing the specific heat ratio, t3 denoting the final temperature, as well as bore dimensions, stroke length, connecting rod dimensions, and compression ratio.

```
# Step 4: Calculate Initial and Final Volumes
v_s = (math.pi / 4) * pow(bore, 2) * stroke
v_c = v_s / (cr - 1)
v1 = v_s + v_c
v2 = v_c
```

The initial and final volumes, namely v1 and v2, are evaluated by considering the bore, stroke, and compression ratio. v1 signifies the total volume incorporating swept and clearance volumes, while v2 signifies the clearance volume.

```
# Step 5: Calculate State Point 2
p2 = p1 * pow(v1, gamma) / pow(v2, gamma)
rhs = p1 * v1 / t1
t2 = p2 * v2 / rhs
```

The pressure at state point 2, identified as p2, is calculated. Furthermore, a value termed rhs is computed, aiding in the derivation of temperature t2 at state point 2.

```
# Step 6: Characterize Compression Process
V_compression = engine_kinematics(bore, stroke, con_rod, cr, 180, 0)
constant = p1 * pow(v1, gamma)
P_compression = [constant / pow(v, gamma) for v in V_compression]
```

Compute variations in volume during compression.

```
# Step 7: Calculate for State Point 3
v3 = v2
rhs = p2 * v2 / t2
p3 = rhs * t3 / v3
```

The pressure at state point 3, labeled as p3, is ascertained. The variable rhs plays a pivotal role in the calculation of temperature t3 at state point 3.

```
# Step 8: Characterize Expansion Process
V_expansion = engine_kinematics(bore, stroke, con_rod, cr, 0, 180)
constant = p3 * pow(v3, gamma)
P_expansion = [constant / pow(v, gamma) for v in V_expansion]
```

Quantify volume shifts within the expansion

```
# Step 9: Calculate for State Point 4
v4 = v1
p4 = p3 * pow(v3, gamma) / pow(v4, gamma)
rhs = p3 * v3 / t3
t4 = p4 * v4 / rhs
```

The pressure at state point 4, referred to as p4, is calculated employing the relationship derived from the adiabatic expansion process.

```
# Step 10: Plot PV Diagram
plt.figure(1)
plt.plot(V_compression, P_compression, label='Adiabatic compression')
plt.plot([v2, v3], [p2, p3], label='Constant volume heat addition')
plt.plot(V_expansion, P_expansion, label='Adiabatic expansion')
plt.plot([v4, v1], [p4, p1], label='Constant volume heat rejection')
plt.xlabel('Volume')
plt.ylabel('Pressure')
plt.legend()
plt.title('Otto Cycle PV Diagram')
plt.savefig('otto_cycle_pv_diagram.png')
plt.show()
```

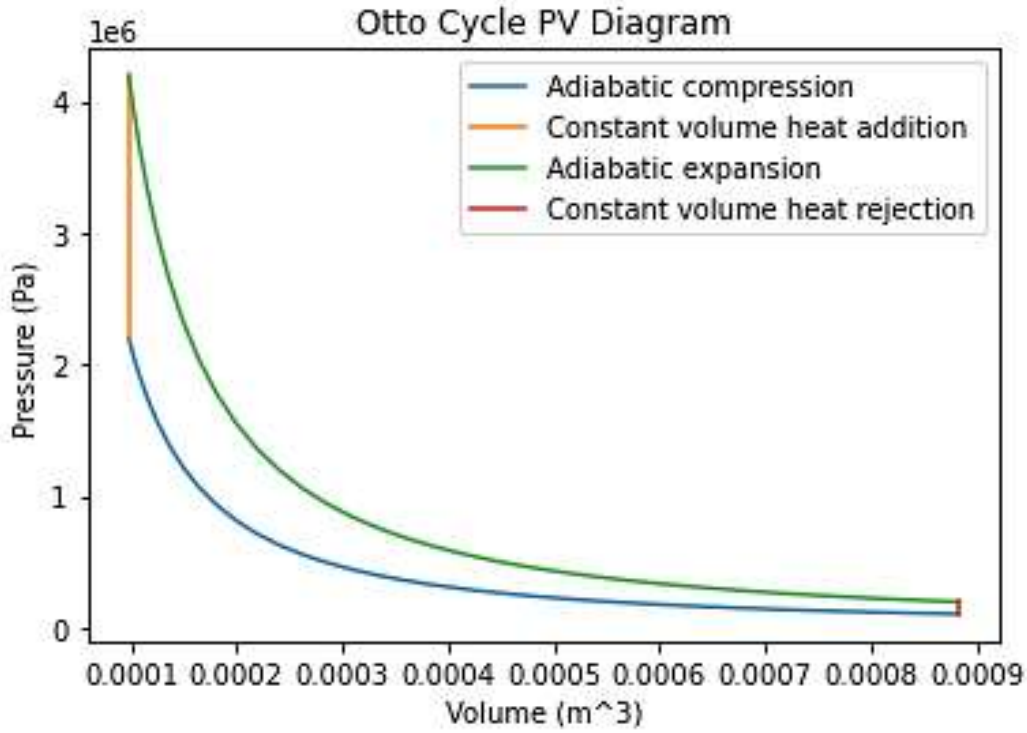
A visual representation illustrating the PV diagram of the Otto cycle is customized and generated within this block.

```
# Step 11: Calculate and Display Thermal Efficiency
eff = 1 - (1 / pow(cr, (gamma - 1)))
eff_percent = eff * 100
print(f'Thermal Efficiency: {eff_percent:.2f}%')
```

The thermal efficiency pertinent to the Otto cycle is calculated.

Results and Discussions

PV Diagram



The PV diagram obtained visually illustrates the thermodynamic sequences inherent in the air-standard Otto cycle. This graph offers insights into alterations in pressure and volume across the cycle's four stages: compression, constant volume heat addition, expansion, and constant volume heat rejection. **The computed thermal efficiency (approximately 55%)** reflects the engine's efficiency in converting heat energy into useful work during the cycle. This efficiency is quantified as the ratio between the area enclosed by the expansion and compression curves and the area beneath the constant volume heat addition and rejection lines. The PV diagram vividly depicts the cyclic progression of the Otto cycle, elucidating the sequence of thermodynamic phases experienced by the engine. The enclosed region within the curve encapsulates the overall work output generated by the engine in a single cycle. Furthermore, the curve's configuration imparts insights into the engine's efficiency and operational prowess, with a more elongated shape indicating heightened efficiency.

The air-standard Otto cycle functions within idealized conditions, overlooking real-world factors like heat dissipation, friction, and internal energy losses that characterize actual engines. Additionally, the diagram overlooks the intricate volume fluctuations arising from factors such as combustion complexities and valve timing, inherent complexities in real engine processes The PV diagram not only aids in understanding the theoretical Otto cycle but also serves as a fundamental reference point for engineers and researchers in the field of internal combustion engines. By comparing real-world engine performance to the idealized air-standard Otto cycle, engineers can identify areas for improvement and develop strategies to enhance the efficiency and reduce emissions of actual engines. This graphical representation is a starting point for the optimization of engine design, combustion strategies, and the implementation of technologies like turbocharging, direct fuel injection, and variable valve timing, all aimed at pushing the boundaries of what is achievable in terms of power output and environmental sustainability.

Furthermore, the PV diagram is a versatile tool for educational purposes. It is commonly used in engineering classrooms and textbooks to teach students about the principles of thermodynamics and the operation of internal combustion engines. The visual nature of the diagram makes it accessible and comprehensible for learners, allowing them to grasp complex concepts such as compression ratios, heat addition, and heat rejection more easily. As a result, the PV diagram plays a vital role in nurturing the next generation of engineers and scientists who will continue to innovate in the field of automotive and mechanical engineering. In the ever-evolving landscape of automotive technology, the PV diagram remains a timeless and indispensable tool, bridging the gap between theoretical knowledge and practical application in the pursuit of more efficient and environmentally friendly internal combustion engines.

B4: Simulating and Animating Damped Pendulum Motion

In Engineering, ordinary differential equations (ODEs) is used to describe the transient behavior of a system. A simple example is a pendulum

The way the pendulum moves depends on the Newton's second law. When this law is written down, we get a second order ordinary differential equation that describes the position of the "ball" with respect to time.

Similarly, there are hundreds of ODEs that describe key phenomena and if you have the ability to solve these equations then you will be able to simulate any of these systems.

Your objective is to write a program that solves the following ODE. This ODE represents the equation of motion of a simple pendulum with damping

$$\frac{d^2\theta}{dt^2} + \frac{b}{m}\frac{d\theta}{dt} + \frac{g}{L}\sin\theta = 0$$

In the above equation,

g = gravity in m/s^2,

L = length of the pendulum in m,

m = mass of the ball in kg,

b=damping coefficient.

Write a program in Python that will simulate the pendulum motion, use,

L=1 m, m=1 kg, b=0.05, g=9.81 m/s^2.

Simulate the motion between 0-20 sec, for angular displacement=0, angular velocity=3 rad/sec at time t=0.

When you do this you will get the position of the pendulum as a function of time. Use this information to animate the motion of the pendulum.

Solution

Objective

The goal is to write a Python code that can simulate and animate the movement of a damped simple pendulum using ordinary differential equations (ODEs). This involves creating equations that account for damping effects and considering factors like gravity, length, mass, and damping coefficient. Through numerical methods, the program computes how the pendulum's angular displacement and velocity change over a defined time span. Using matplotlib, the program then generates an animation that visually represents the pendulum's motion. This challenge illustrates how ODEs can be applied in engineering to model and visualize real-world systems.

Problem Statement

In engineering, ODEs are extensively utilized to depict the transient behaviors exhibited by various systems. A straightforward exemplar of this principle can be observed in the case of a pendulum. The pendulum's motion is underpinned by the fundamental principles encapsulated within 'Newton's second law of motion.' Undertaking the task of solving the equation governing this motion using Python offers a valuable opportunity to gain familiarity with the intricacies of the scipy library and its odeint class. This hands-on approach allows for a deeper comprehension of the specific parameters required to facilitate accurate numerical integration and the resolution of differential equations within this context. Figure 4 represents the main parameters that need to be considered to characterize the motion of a simple pendulum.

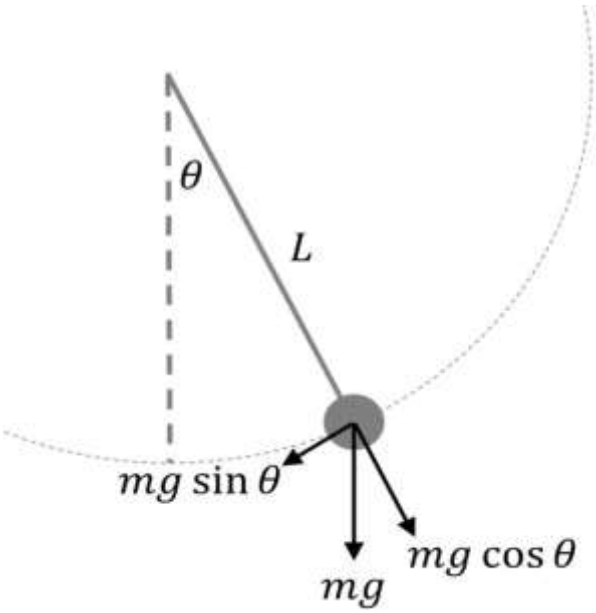


Figure 4: The motion of a simple pendulum. Available at: https://skill-lync.com/student-projects/week-3-solving-second-order-odes-628

Equations Used

The second order differential equation coupled with the arrangement in matrix form:

$$\frac{d^2\theta}{dt} + \frac{b}{m}\frac{d\theta}{dt} + \frac{g}{l}\sin\theta = 0 \qquad (1)$$

Where,

$\boldsymbol{\theta}$ is the angular displacement

$\boldsymbol{b}$ is the damping coefficient

$\boldsymbol{m}$ is the mass of the bob

$\boldsymbol{g}$ is the acceleration due to gravity

$\boldsymbol{l}$ is the length of the pendulum

Assuming that $\theta = \theta_1$ and $\frac{d\theta_1}{dt} = \theta_2$, and solving ODE in the matrix form:

$$\frac{d}{dt}\begin{pmatrix}\theta_1\\ \theta_2\end{pmatrix} = \begin{pmatrix}\theta_2\\ -\frac{b}{m}\theta_2 - \frac{g}{l}\sin\theta_1\end{pmatrix} \qquad (2)$$

Solution Procedure

Step 1: Import Essential Libraries Begin by importing the required libraries that will facilitate essential calculations, numerical integration, plotting, and animation.

Step 2: Formulate the Differential Equation System Construct a function called `system` which encapsulates the collection of first-order differential equations representative of the behavior of a damped simple pendulum.

Step 3: Specify Parameters and Initial Conditions Set the vital physical parameters including the damping coefficient (b), gravitational acceleration (g), length of the pendulum (l), mass (m), initial angular displacement, and the designated time span for the simulation.

Step 4: Solve the Ordinary Differential Equation Employ the `odeint` function from the `scipy.integrate` module to effectively solve the system of differential equations that were outlined in the previous step. This computational step yields the angular displacement and angular velocity of the pendulum throughout the given time interval.

Step 5: Generate a Visual Representation of the Results Utilize matplotlib to craft a graphical representation that visually communicates the angular displacement and angular velocity of the pendulum over the course of time.

Step 6: Produce an Animated Illustration of the Pendulum's Movement Iterate through the computed arrays of angular displacement and velocity, utilizing the information to construct an animated portrayal of the pendulum's dynamic motion. At each time point, calculate the pendulum's position and translate this into a visual animation. Preserve each frame of the animation as a distinct image file, preserving these frames for subsequent compilation. Display the animation by sequentially showcasing the stored frames, thereby rendering the dynamic motion of the pendulum in a visually comprehensible manner.

Python Code

```
# Step 1: Import required libraries
import math
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import odeint
```

Start by importing imperative libraries that empower the program with necessary capabilities.

```
# Step 2: Define the Differential Equation System
def system(theta_init, time, b, g, l, m):
    theta1 = theta_init[0]
    theta2 = theta_init[1]
    dtheta1_dt = theta2
    dtheta2_dt = (-(b/m) * dtheta1_dt) - ((g/l) * math.sin(theta1))
    dtheta_dt = [dtheta1_dt, dtheta2_dt]
    return dtheta_dt
```

Create a function named “system”, which embodies a set of initial-order differential equations responsible for capturing the intricacies of a damped simple pendulum's dynamics.

```
# Step 3: Set Parameters and Initial Conditions
b = 0.05
g = 9.81
l = 1
m = 1
theta_init = [0, 3]
time = np.linspace(0, 20, 200)
```

Define various parameters and initial conditions that mold the distinct attributes of the pendulum and the scope of the simulation.

```
# Step 4: Solve the ordinary differential equation
theta = odeint(system, theta_init, time, args=(b, g, l, m))
ang_disp = theta[:, 0]
ang_vel = theta[:, 1]
```

Use the odeint function to numerically define the array of differential equations outlined within the system function.

```
# Step 5: Plot the Results
plt.figure(figsize=(10, 6))
plt.plot(time, ang_disp, 'b-', label='angular displacement')
plt.plot(time, ang_vel, 'r-', label='angular velocity')
plt.legend()
plt.xlabel('time')
plt.ylabel('displacement and velocity')
plt.title('Angular Displacement and Velocity of Simple Pendulum')
plt.grid()
plt.show()
```

Create a plot using matplotlib. Two lines are plotted on the same graph: one for angular displacement and another for angular velocity. The plot is also customized.

```
# Step 6: Create pendulum animation
for i in range(len(time)):
    disp = ang_disp[i]
    vel = ang_vel[i]
    x0 = 0
    y0 = 0
    x1 = l * (math.sin(disp))
    y1 = -l * (math.cos(disp)

    plt.figure(figsize=(8, 6))
    plt.plot([-1, 1], [0, 0], 'b')  # Support
```

A loop iterates through each time step in the simulation. For each step, the angular displacement and velocity are retrieved.

A new figure is created for each iteration, and plot customization is done.

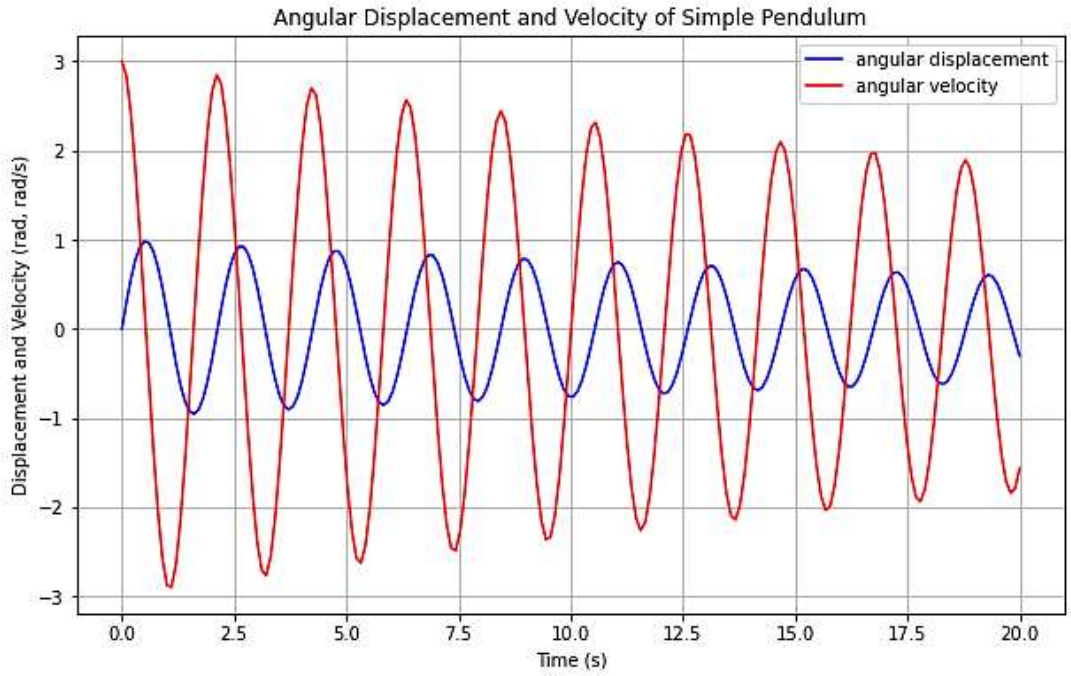
Once all frames are generated and saved, the animation is displayed using plt.show().

```
plt.plot([x0, x1], [y0, y1], 'g')  # Pendulum
plt.plot(x1, y1, marker='o', color='r')  # Bob
plt.axis([-1.5, 2, -1.5, 1])  # Set axis limits
plt.title('Simple Pendulum Animation')

# Save animation frames
plt.figure()
filename = str(i + 1) + '.png'
plt.savefig(filename)
plt.clf()  # Clear figure
```

Results and Discussions

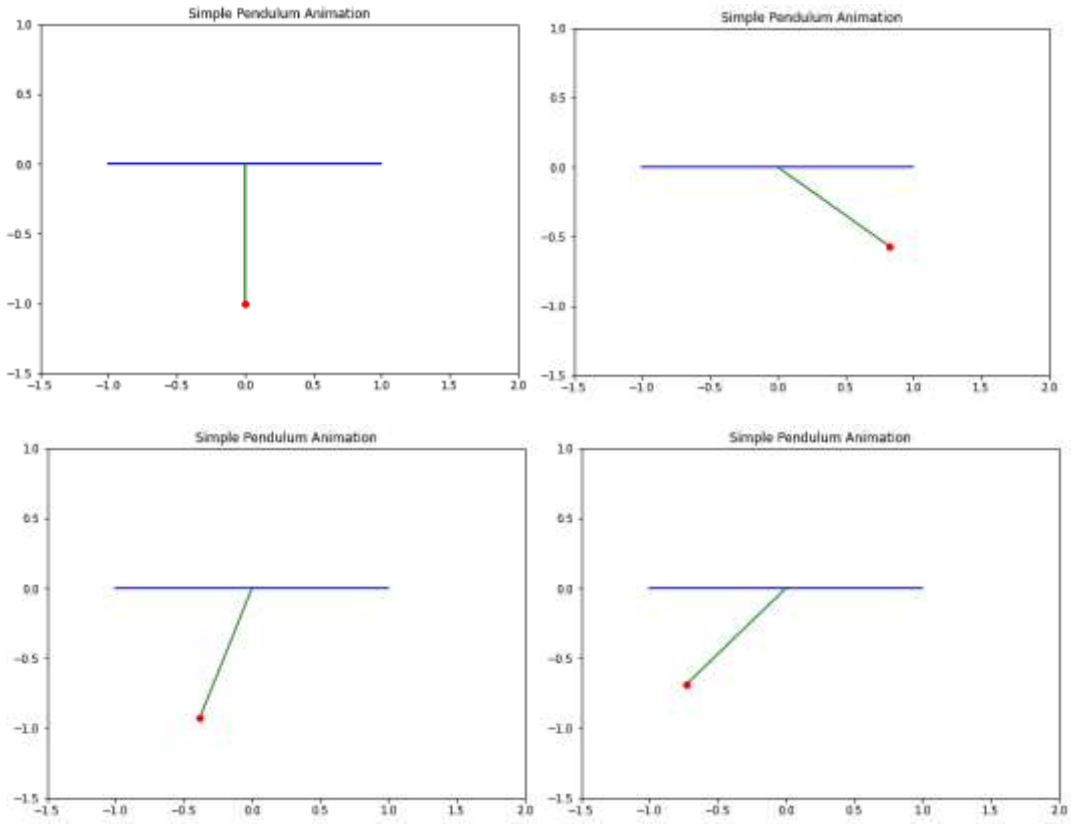
Angular Displacement and Velocity



The plot generated offers a comprehensive visualization of a damped simple pendulum's angular displacement and angular velocity within a specified time frame. The blue line portrays the pendulum's angular displacement changes, showcasing its oscillatory motion around the equilibrium point. Angular velocity is represented by the red line, reflecting the pendulum's rotation speed across different time intervals. Clearly labeled axes provide contextual information, with time on the horizontal axis and angular displacement/velocity on the vertical axis. A legend differentiates between the two lines, while the title "Angular Displacement and Velocity of Simple Pendulum" concisely summarizes the plot's essence. The presence of a grid enhances precision in reading values. Altogether, the plot adeptly captures the dynamic interplay between angular displacement and angular velocity as the pendulum's motion unfolds over time.

Simple Pendulum Motion

A sequence of images that represent the motion of the simple pendulum at different time instants are shown below. The complete video animation is uploaded to the link: https://youtu.be/Ll9axJ0ldN8



B5: Pressure Computation and Relaxation Analysis using Newton-Raphson Technique

For this challenge, you will be writing a python script to solve for the minimum value of pressure using the Newton-Raphson method.

Where p denotes the cushion pressure, h the thickness of the ice field, r the size of the air cushion, σ the tensile strength of the ice and β is related to the width of the ice wedge.

Take β = 0.5, r = 40 feet and σ = 150 pounds per square inch (psi).

Do not convert to the metric system, however, make sure the units are consistent before proceeding further with the problem.

You need to,

1. Use the Newton-Raphson method to find out the value of pressure for h = 0.6 ft.

2. Find the optimal relaxation factor for this problem with the help of a suitable plot.

3. Tabulate the results of p for h = [0.6,1.2,1.8,2.4,3,3.6,4.2] assuming a suitable relaxation factor. This must be done in python.

Solution

Objective

The main objectives of this challenge are:

1- Apply the Newton-Raphson technique to determine the pressure value when `h` is set to 0.6.

2- Identify the optimal relaxation factor for this issue by utilizing an appropriate plot.

3- Organize a table presenting the pressure results for `p` at various `h` values, specifically [0.6, 1.2, 1.8, 2.4, 3, 3.6, 4.2], while considering an appropriate relaxation factor.

Problem Statement

The Newton–Raphson method, which derives its name from Isaac Newton and Joseph Raphson, is an algorithm designed to find improved approximations to the roots (or zeros) of a real-valued function. The fundamental version of this technique begins with a real variable `x` and a single-variable function `f`, along with the function's derivative `f′` and

an initial estimate `$x_0$` for a root of `f`. When the function meets certain assumptions and the initial guess is sufficiently close, then:

$$x_1 = x_0 - \frac{f(x_0)}{f'(x_0)} \qquad (1)$$

It is a more accurate approximation of the root compared to `$x_0$`. Geometrically, (x1, 0) represents the point where the x-axis intersects the tangent line of the graph of `f` at the point (x_0, f (x_0)). This improved estimate essentially corresponds to the unique root of the linear approximation at the initial point. The process continues iteratively as:

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)} \qquad (2)$$

This process continues until a level of precision that meets the requirements is achieved. Figure 5 explains the technique of Newton-Raphson method and its derivation.

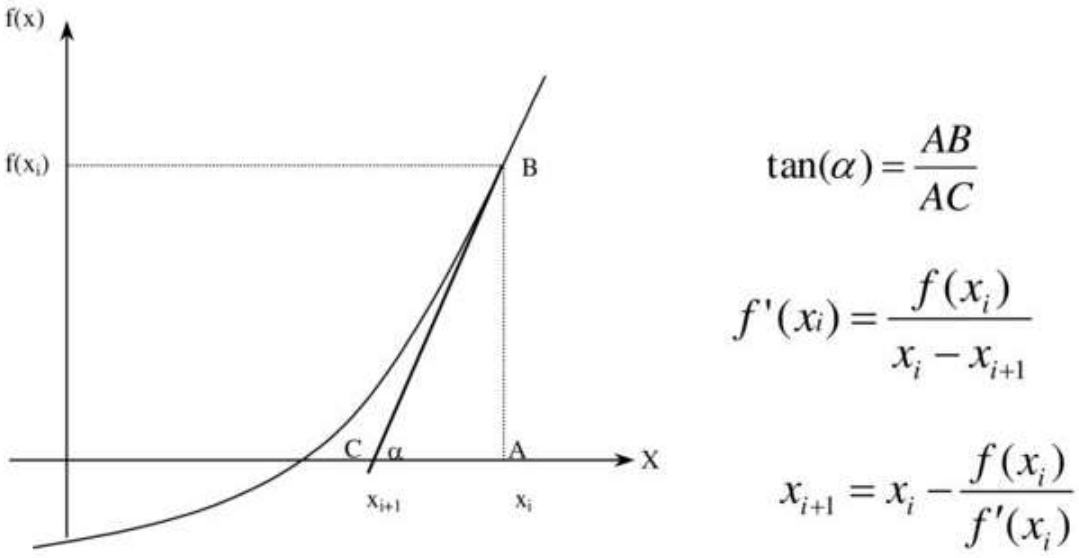


Figure 5: Derivation of the Newton-Raphson method. Available at: https://www.researchgate.net/publication/329403155

The concept revolves around beginning with an initial guess that is reasonably near the true root. Subsequently, the function is approximated by its tangent line using calculus, and the x-intercept of this tangent line is computed using basic algebra. This x-intercept generally serves as a more accurate approximation of the root of the original function than the initial guess, allowing for further iterations.

Formally, suppose that f: (a, b) → $\mathbb{R}$ is a differentiable function defined within the interval (a, b) and yielding real number values. If we have a current approximation `$x_i$`, it is possible to derive the formula for a better approximation, `$x_i + 1$`, with reference to the provided figure. The equation of the tangent line to the curve `y = f (x) ` at `x = $x_i$` is

$$y = f'(x_i)(x - x_i) + f(x_i) \quad (3)$$

Where `f'` denotes the derivative. The x-intercept of this line (the value of `x` that makes y = 0) serves as the next approximation, `$x_i + 1$`, for the root. This ensures that the equation of the tangent line is satisfied when (x, y) = ($x_i + 1$, 0).

$$0 = f'(x_n)(x_{n+1} - x_n) + f(x_n) \quad (4)$$

Then solving for x_i+1 gives:

$$x_{i+1} = x_i - \left\{\frac{f(x_i)}{f'(x_i)}\right\} \quad (5)$$

The process starts with an arbitrary initial value, `$x_0$` (Optimally, a value closer to the zero is preferred). However, when there is limited insight into the zero's location, a "guess and check" method can narrow down possibilities to a reasonably small interval using the intermediate value theorem. The method generally converges, provided that the initial guess is sufficiently close to the unknown zero and that `$f'(x_0) \neq 0$`.

Solution Procedure

The solution steps are as follows:

Step 1: Import Libraries and Set Parameters Import the necessary libraries: `numpy` and `matplotlib.pyplot`. Set the input parameters: `beta = 0.5`, `r = 40`, `sigma = 150`, `h` values, and `alpha = 1.1`.

Step 2: Define Governing Function and its Derivative Define the functions `f(p, h, r, beta, sigma)` and `fprime(p, h, r, beta, sigma)` based on the given formulas and parameters.

Step 3: Initialize Result Lists Create empty lists `pressure`, `pressure_0`, and `num` to store calculated pressure values, pressure in psi, and iteration counts, respectively.

Step 4: Newton-Raphson Method for Pressure Calculation Iterate through each value of `h` in the list. Initialize `p_guess`, `tolerance`, and `counter`. Use the Newton-Raphson iteration until the function value is within the specified tolerance. Append calculated pressure values, pressure in psi, and iteration counts to respective lists.

Step 5: Print and Plot Pressure Results Print the calculated results for ice wedge thickness, cushion pressure, pressure in psi, and number of iterations. Plot the pressure versus ice field thickness and pressure in psi versus ice field thickness.

Step 6: Optimal Relaxation Factor Calculation Set `h_1 = 0.6` and create an array of relaxation factors `alpha_array` using `numpy.linspace()`. Initialize an empty list `iterations` to store iteration counts for different relaxation factors. Iterate through each relaxation factor and perform the Newton-Raphson iteration until convergence. Append the iteration counts to the `iterations` list.

Step 7: Print Optimal Relaxation Factor Print a table displaying relaxation factors and their corresponding iteration counts. Identify and print the optimal relaxation factor that yields the minimum iterations.

Step 8: Plot Optimal Relaxation Factor vs. Iterations Create a plot that illustrates the relationship between relaxation factors and iteration counts.

By following these steps, the provided Python code calculates cushion pressures, identifies the optimal relaxation factor, and presents the results using tables and plots.

Python Code

```
#Step 1: Import Libraries and Set Input Parameters
import numpy as np
import matplotlib.pyplot as plt
beta = 0.5
r = 40
sigma = 150
h = [0.6, 1.2, 1.8, 2.4, 3.0, 3.6, 4.2]
alpha = 1.1
```

Necessary libraries, including numpy and matplotlib.pyplot, are imported. Input parameters like beta, r, sigma, h values, and alpha are initialized.

```
#Step 2: Define Governing Function and Its Derivative
def f(p, h, r, beta, sigma):
    term_1 = pow(p, 3) * (1 - pow(beta, 2))
    term_2a = 0.4 * h * pow(beta, 2)
    term_2b = (sigma * (pow(h, 2)) / pow(r, 2))
    term_3 = pow(sigma, 2) * (pow(h, 4)) / (pow(r, 4))
    term_4 = (sigma / 3) * (pow(h, 2)) / (pow(r, 2))
    return term_1 + (term_2a - term_2b) * pow(p, 2) + term_3 * (p / 3) - pow(term_4, 3)

def fprime(p, h, r, beta, sigma):
    term_5 = 3 * pow(p, 2) * (1 - pow(beta, 2))
    term_6a = 0.4 * h * pow(beta, 2)
```

Two functions, namely f(p, h, r, beta, sigma) and fprime(p, h, r, beta, sigma), are introduced to compute the governing function and its derivative using given equations and input values.

```
    term_6b = (sigma * (pow(h, 2)) / pow(r, 2))
    term_7 = pow(sigma, 2) * (pow(h, 4)) / (pow(r, 4))
    return term_5 + 2 * (term_6a - term_6b) * p + term_7 / 3

#Step 3: Initialize Lists for Results
pressure = []       # pound per square feet
pressure_0 = []     # pound per square inches(psi)
num = []

#Step 4: Newton-Raphson Method for Pressure Calculation
for i in h:
    p_guess = 100
    tolerance = 1e-6
    counter = 1
    while(abs(f(p_guess, i, r, beta, sigma)) > tolerance):
        p_guess = p_guess - alpha * (f(p_guess, i, r, beta, sigma) / fprime(p_guess, i, r, beta, sigma))
        counter += 1
    pressure.append(p_guess)
    pressure_0.append(p_guess / 144)
    num.append(counter)

#Step 5: Print and Plot Pressure Results
print('Thickness of the Ice wedge:', h)
print('Cushion Pressure (pound per square feet):', pressure)
print('Cushion Pressure (psi):', pressure_0)
print('Number of iterations:', num)
```

Lists such as pressure, pressure_0, and num are initialized to store computed pressure values, pressure in psi, and iteration counts.

For each ice thickness value in h, the Newton-Raphson iteration process is employed to calculate the cushion pressure.

```
plt.figure(1)
plt.plot(h, pressure, marker='.', color='blue', markerfacecolor='black', markersize=10)
plt.title('Cushion pressure versus Ice field thickness')
plt.xlabel('Thickness of ice field (feet)')
plt.ylabel('Cushion Pressure (pound per square feet)')
plt.grid()
plt.show()
```

Information about ice thickness, cushion pressure, pressure in psi, and iteration counts is presented. Two plots are generated as per the problem statement.

```
plt.figure(2)
plt.plot(h, pressure_0, marker='.', color='blue', markerfacecolor='red', markersize=10)
plt.title('Cushion pressure versus Ice field thickness')
plt.xlabel('Thickness of ice field (feet)')
plt.ylabel('Cushion Pressure (pound per square inches)')
plt.grid()
plt.show()

#Step 6: Optimal Relaxation Factor Calculation
h_1 = 0.6
alpha_array = np.linspace(0.1, 1.9, 50)
iterations = []
```

The code determines the optimal relaxation factor by iterating through a range of relaxation factors and tracking the number of iterations required to achieve convergence.

```
for j in alpha_array:
    p_guess = 100
    tolerance = 1e-6
    counter = 1
    while(abs(f(p_guess, h_1, r, beta, sigma)) > tolerance):
        p_guess = p_guess - (j * f(p_guess, h_1, r, beta, sigma)) / fprime(p_guess, h_1, r,
beta, sigma)
```

```
        counter += 1
    iterations.append(counter)


#Step 7: Print Optimal Relaxation Factor
print("alpha : No. of iterations")
for j in range(len(alpha_array)):
    print(": ", '%.1f' % alpha_array[j], " : ", '%0.1f' % iterations[j])


for k in range(len(iterations)):
    if iterations[k] == min(iterations):
        print('The optimum value of under Relaxation Factor:', str(alpha_array[k]))


#Step 8: Plot Optimal Relaxation Factor vs. Iterations
plt.figure(3)
plt.plot(alpha_array, iterations, marker='.', color='black', markerfacecolor='green',
markersize=7.5)
plt.xlabel('Relaxation factor (alpha)')
plt.ylabel('No. of iterations')
plt.title('Iterations versus Relaxation factor')
plt.grid()
plt.show()
```

The optimal relaxation factor is printed, offering insights into the correlation between various relaxation factors and their corresponding iteration counts.

A graphical representation is customized and generated to visualize the connection between relaxation factors and iteration counts.

Results and Discussions

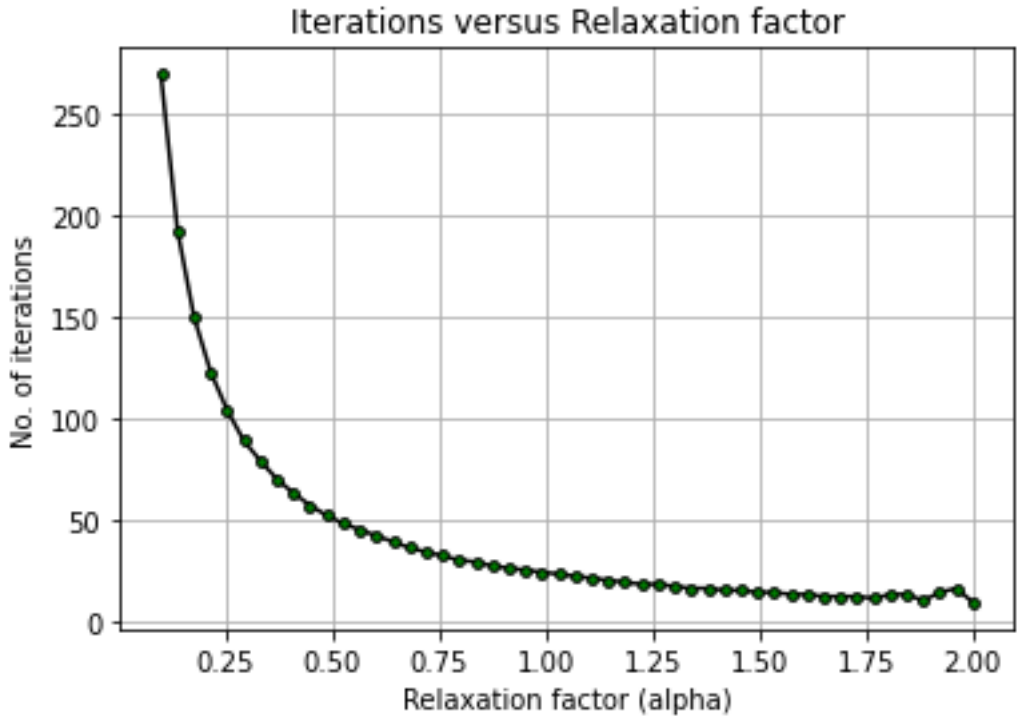
1. Use the Newton-Raphson method to find out the value of pressure for h = 0.6 ft.

As per the output window, when the thickness of the ice wedge is 0.6 feet, the cushion pressure equals to 0.0034 pound per square feet, or $2.37*10^{-5}$ psi.

```
Thickness of the Ice wedge: [0.6, 1.2, 1.8, 2.4, 3.0, 3.6, 4.2]
Cushion Pressure (pound per square feet): [0.0034154461884626955, 0.015177190727930742,
0.03825791837496162, 0.07367116765282837, 0.12219771849154307, 0.1846425167891315,
0.26168022073823527]
Cushion Pressure (psi): [2.3718376308768718e-05, 0.00010539715783285237, 0.0002656799887150112 3,
0.0005116053309224193, 0.0008485952673023824, 0.0012822396999245243, 0.0018172237551266337]
Number of iterations: [21, 20, 19, 19, 19, 19, 19]
```

2. Find the optimal relaxation factor for this problem with the help of a suitable plot.

According to the plot and the output window, the optimum value of relaxation factor is 1.9. It worths noting that the number of iterations started to diverge after this point.



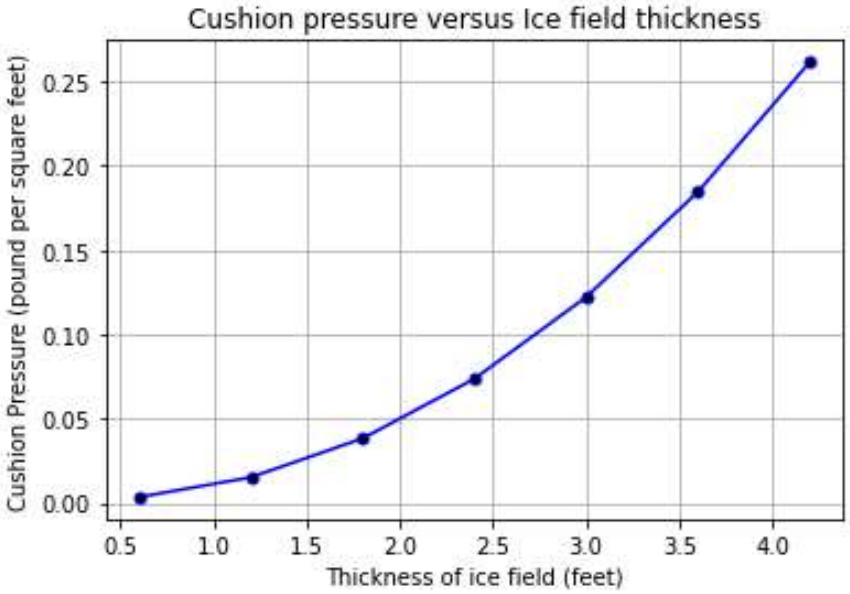
3. Tabulate the results of p for h = [0.6,1.2,1.8,2.4,3,3.6,4.2] assuming a suitable relaxation factor. This must be done in python.

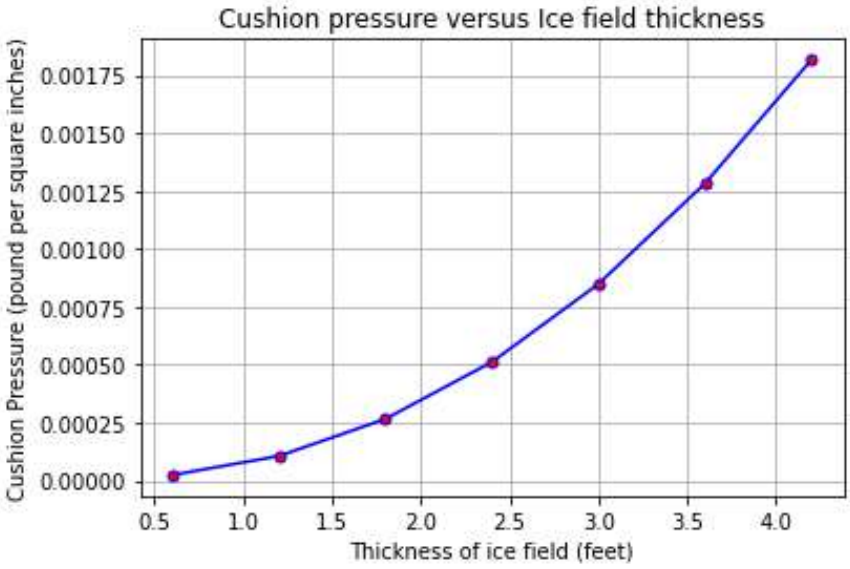
According to the output window of Python, the corresponding cushion pressure (p) at each ice wedge thickness (h) can be tabulated as follows:

Thickness of the Ice wedge (ft)	Cushion Pressure (pound per square feet)	Cushion Pressure (psi)
0.6	0.003	0.0000237
1.2	0.015	0.0001054
0.8	0.038	0.0002657
2.4	0.074	0.0005116
3.0	0.122	0.0008486
3.6	0.185	0.0012822
4.2	0.262	0.0012822

Note: These values were obtained at a relaxation factor = 1.1

Additionally, the generated plots shows a graphical illustrations for the values of cushion pressure against the the thickness of ice field.





Conclusions

In this challenge, the objective was to utilize the Newton-Raphson method for calculating the lowest cushion pressure based on specified ice field thickness and parameters. The code effectively accomplished this task by iteratively improving pressure estimates until they converged. Key factors such as ice wedge width, air cushion size, and ice strength were considered. Furthermore, the code identified the most efficient relaxation factor for achieving convergence. The outcomes were presented using tables and graphical plots, shedding light on pressure variations, the influence of the relaxation factor, and overall system behavior. This challenge highlighted the significance of numerical techniques and parameter optimization in solving engineering problems proficiently.

B6: Exploring Parameter Meanings, Curve Fitting, and Analysis in Python

Click here to get the Cp vs Temperature data file

1. What does popt and pcov mean?

```
popt, pcov = curve_fit(func, temperature, cp)
```

2. What does np.array(temperature) do?

3. What does the * in *popt mean?

4. Write code to fit a linear and cubic polynomial for the Cp data. Explain if your results are good or bad.

5. What needs to be done in order to make the curve fit perfect?

6. Show empirically as to how well your curve has been fit.

Solution

1. What does popt and pcov mean?

```
popt, pcov = curve_fit(func, temperature, cp)
```

In Python, especially when utilizing the scipy.optimize.curve_fit function from the SciPy library, the terms "popt" and "pcov" hold specific significance:

- **"popt"** an abbreviation for "optimized parameters" refers to a variable designed to hold the parameter values that have been optimized or adjusted through the process of fitting a curve to data. When employing the curve_fit function to align a model with data points, it fine-tunes the model's parameters to most accurately align with the data. The resultant "popt" variable contains these refined parameter values.

- **"pcov"** a shorthand for "parameter covariance" pertains to a variable intended to store the covariance matrix linked to the improved parameter estimations. The covariance matrix furnishes insights into the uncertainty and interconnections among the parameter estimations. The matrix's diagonal elements denote the variances of the parameter estimates, whereas the off-diagonal components indicate the covariance between pairs of parameters.

2. What does np.array(temperature) do?

The code `np.array(temperature)` generates a NumPy array using the information found within the 'temperature' variable. NumPy stands as a robust Python library that facilitates tasks involving arrays, matrices, and an extensive array of mathematical operations.

In situations where 'temperature' represents a list, tuple, or an analogous array-like structure containing numeric values, the utilization of `np.array(temperature)` results in the transformation of this data into a NumPy array. This transformation proves advantageous due to the array's enhanced capabilities compared to conventional Python lists. These benefits encompass optimized numeric computations, efficient memory usage, and convenient element-wise operations.

3. What does the * in *popt mean?

The asterisk * in Python has various meanings depending on the situation. Specifically, when considering *popt, it serves the purpose of separating the elements within a list or tuple into distinct arguments.

Within the domain of curve fitting and function invocations, particularly when handling the scipy.optimize.curve_fit function provided by the SciPy library, the expression *popt is frequently employed to break down the contents of the popt array into individual arguments. This practice is often utilized when transmitting optimized parameters to a function that anticipates separate parameter values as its arguments.

4. Write code to fit a linear and cubic polynomial for the Cp data. Explain if your results are good or bad.

```
# Step 1: Import necessary libraries
import matplotlib.pyplot as plt
import numpy as np
from scipy.optimize import curve_fit
```

Begin by importing the libraries: matplotlib.pyplot, numpy, and scipy.optimize.curve_fit. These libraries serve for plotting, numerical operations, and fitting curves.

```
# Step 2: Define the curve-fit functions
def func_1(t, a, b):
    return a * t + b  # Linear
```

Defining polynomial functions for fitting.

```
def func_2(t, a, b, c):
    return a * pow(t, 2) + b * t + c  # Quadratic
```

```
def func_3(t, a, b, c, d):
    return a * pow(t, 3) + b * pow(t, 2) + c * t + d  # Cubic

def func_4(t, a, b, c, d, e):
    return a * pow(t, 4) + b * pow(t, 3) + c * pow(t, 2) + d * t + e  # Quartic

# Step 3: Define function for curve fitting
def CF(func, temp, cp):
    popt, pcov = curve_fit(func, temp, cp)
    fit_cp = func(np.array(temp), *popt)
    return fit_cp, popt, pcov
```

Develop the CF function (Curve Fit) to fit given functions (func) to temperature (temp) and heat capacity (cp) data. Compute the fitted curve using optimized parameters (popt) converting it to an array.

```
# Step 4: Define function for evaluating goodness of fit
def GoF(t, cp, func, popt):
    print("Mean Squared Error:", np.mean((cp - func(np.array(t), *popt)) ** 2))
    ss_res = np.dot((cp - func(np.array(t), *popt)), (cp - func(np.array(t), *popt)))
    ymean = np.mean(cp)
    ss_tot = np.dot((cp - ymean), (cp - ymean))
    print("R-Square Value:", 1 - ss_res / ss_tot)
```

Create the GoF function (Goodness of Fit) to assess the quality of curve fitting. Use mean squared error (MSE), sums of squared residuals (ss_res) and total sum of squares (ss_tot) to derive R-Square.

```
# Step 5: Define function to read data file
def read_file():
    temperature = []
    cp = []
    for line in open('data', 'r'):
        values = line.split(',')
        temperature.append(float(values[0]))
```

Define read_file to read temperature and heat capacity data from a file called 'data'.

Separate data points by splitting lines at commas.

Append temperature and heat capacity values to distinct lists, returning them as a pair.

```
        cp.append(float(values[1]))
    return [temperature, cp]

# Step 6: Read data from the file
temperature, cp = read_file()

# Step 7: Perform linear fit
fit_cp_linear, popt, pcov = CF(func_1, temperature, cp)
print('Goodness of fit for linear polynomial:')
GoF(temperature, cp, func_1, popt)
print()

# Step 8: Perform cubic fit
fit_cp_cubic, popt, pcov = CF(func_3, temperature, cp)
print('Goodness of fit for cubic polynomial:')
GoF(temperature, cp, func_3, popt)
print()

# Step 9: Choose different polynomials for different regions
ind = temperature.index(1500.0)
temp1 = temperature[0:ind + 1]
temp2 = temperature[ind + 1:]
fit1, popt1, pcov1 = CF(func_4, temp1, cp[0:ind + 1])
fit2, popt2, pcov2 = CF(func_2, temp2, cp[ind + 1:])
print('Goodness of fit for temperature range %.1f to %.
1f with quartic polynomial:' % (temperature[0], temperature[ind]))
GoF(temp1, cp[0:ind + 1], func_4, popt1)
print()
```

Fetch temperature and heat capacity data from the 'data' file using the read_file function.

Utilize the CF function twice: once for fitting the linear curve (func_1) and once for the cubic curve (func_3).

Compute the fitted curves, optimized parameters (popt), and covariance matrices (pcov).

Evaluate fit quality using the GoF function for both linear and cubic fits.

Present the results via console output.

Find the index of temperature value 1500.0 and split temperature data into two parts (temp1 and temp2), and perform curve fitting with optimized popt and pcov. Then evaluate fit.

```
print('Goodness of fit for temperature range %.1f to %.1f with quadratic polynomial:' %
(temperature[ind + 1], temperature[-1]))

GoF(temp2, cp[ind + 1:], func_2, popt2)

print()


# Step 10: Display plots

fig, ax = plt.subplots(2, 3, sharex=True, sharey=True)

fig.suptitle('CURVE FITTING')


ax[0,0].plot(temperature, cp, color='blue', linewidth=3)

ax[0,0].set_title('Original temp vs cp')


ax[0,1].plot(temperature, cp, color='blue', linewidth=3)

ax[0,1].plot(temperature, fit_cp_linear, color='red', linewidth=3)

ax[0,1].set_title('Linear fit')


ax[0,2].plot(temperature, cp, color='blue', linewidth=3)

ax[0,2].plot(temperature, fit_cp_cubic, color='red', linewidth=3)

ax[0,2].set_title('Cubic fit')


ax[1,0].plot(temperature, cp, color='blue', linewidth=3)

ax[1,0].plot(temp1, fit1, color='red', linewidth=3)

ax[1,0].set_title('Quartic t<1500')


ax[1,1].plot(temperature, cp, color='blue', linewidth=3)

ax[1,1].plot(temp2, fit2, color='green', linewidth=3)

ax[1,1].set_title('Quadratic t>1500')
```

Proceed from step 9 and perform curve fitting for quadratic polynomial using CF function with temp2.

Calculate fitted curve, optimized parameters (popt2), and covariance matrix (pcov2).

Evaluate fit quality using the GoF function for the quadratic polynomial above 1500.0.

```
ax[1,2].plot(temperature, cp, color='blue', linewidth=3)
ax[1,2].plot(temp1, fit1, color='red', linewidth=3)
ax[1,2].plot(temp2, fit2, color='green', linewidth=3)
ax[1,2].set_title('Quartic & Quadratic')
```

```
# Step 11: Set labels for all the axes
for A in ax.flat:
    A.set(xlabel='Temperature (K)', ylabel='Cp')
```

Generate a subplot grid with 2 rows and 3 columns using plt.subplots. Assign a title to the entire figure using fig.suptitle.

```
# Step 12: Label only outer axes
for A in ax.flat:
    A.label_outer()
```

Iterate through all subplots (ax.flat) and apply labels for x-axis and y-axis via A.set.

```
# Step 13: Maximize the figure window
mng = plt.get_current_fig_manager()
mng.window.state('zoomed')  # Works fine on Windows only
```

Maximize the figure window with plt.get_current_fig_manager().window.state('zoomed').

```
# Step 14: Display the figure
fig.show()
```

Show the figure using fig.show().

```
# Step 15: Save the figure as an image
plt.savefig("Curve_fitting.jpeg", dpi=600)
```

Save the figure as an image file named "Curve_fitting.jpeg" using plt.savefig.

Output Window

Goodness of fit for linear polynomial:

Mean Squared Error: 675.9528403686832

R-Square Value: 0.9248776260551825

Goodness of fit for cubic polynomial:

Mean Squared Error: 29.460006412729314

R-Square Value: 0.9967259467140538

Goodness of fit for temperature range 300.0 to 1500.0 with quartic polynomial:

Mean Squared Error: 1.343476770560602

R-Square Value: 0.9997174708061651

Goodness of fit for temperature range 1501.0 to 3499.0 with quadratic polynomial:

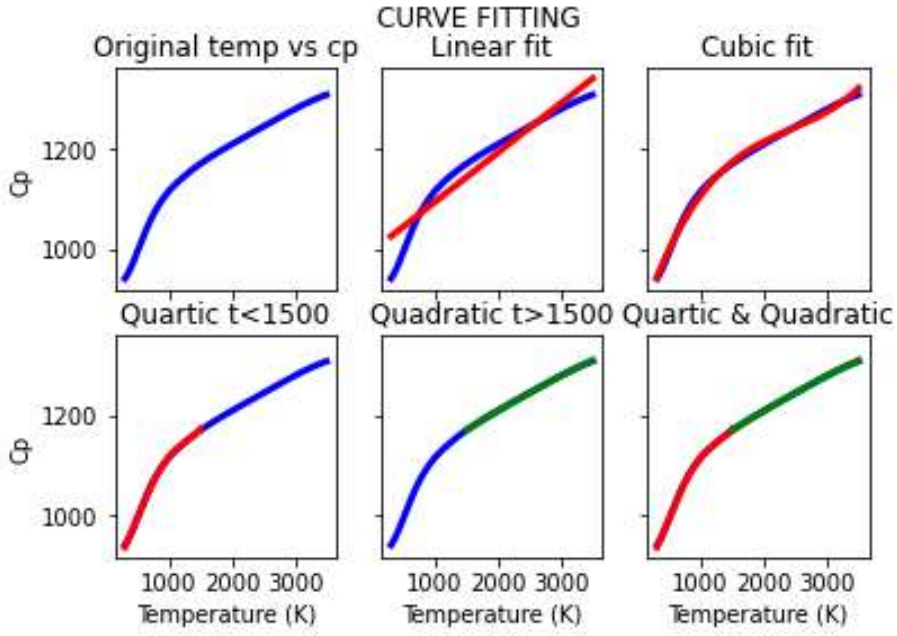
Mean Squared Error: 0.2769264992105013

R-Square Value: 0.9998290863891499

Comments

- The linear polynomial fit has a MSE of 675.95 and an R-Square value of approximately 0.92.The R-Square value suggests that the linear fit explains around 92% of the data's variation. Despite the relatively high MSE, the fit appears reasonable.
- For the cubic polynomial fit, the MSE is 29.46 and the R-Square value is about 0.997. The high R-Square value indicates that the cubic fit explains approximately 99.7% of the data's variability. The low MSE suggests a favorable fit.
- The quartic polynomial fit, covering temperatures from 300.0 to 1500.0, has an MSE of 1.34 and an impressive R-Square value of around 0.9997. This high R-Square value signifies an exceptional fit, explaining nearly 99.97% of the data's variability. The low MSE further confirms the quality of this fit.
- In the quadratic polynomial fit, spanning temperatures from 1501.0 to 3499.0, the MSE is 0.28 and the R-Square value is about 0.9998. The remarkably high R-Square value suggests an outstanding fit, explaining almost 99.98% of the data's variability. The very low MSE supports the excellent fit quality.

The following plots were generated using the above code:



5. What needs to be done in order to make the curve fit perfect?

To make a perfect fit, consider these two strategies:

- **Increasing the polynomial order:** Increasing the polynomial order, such as using a quartic polynomial, might enhance the fit quality compared to a cubic polynomial.
- **Employing distinct curves for different data segments:** Through experimentation with the provided code, it was observed that an almost impeccable fit for the dataset is attainable by opting for a quartic (4th order) polynomial when temperatures are below 1500 K, and a quadratic polynomial for temperatures exceeding 1500 K.

6. Show empirically as to how well your curve has been fit.

To empirically evaluate the quality of the fit, the assessment involves the computation of the following two metrics:

a. Mean Squared Error (MSE)

- The MSE signifies the average of the squared differences between the actual data and the curve fit.

- A lower value indicates a better fit quality, and as the value approaches 0, the fit becomes more accurate.

b. R-Square Value

- The R-Square value is the ratio of the sum of squares of the regression (SSR) to the total sum of squares (SST).

- SSR represents the sum of the squared differences between the values obtained from the curve fit and the mean of the actual data.

- SST is the sum of the squared differences between the actual data values and the mean of the actual data values.

- A higher R-Square value, closer to 1, indicates a superior fit quality.

The corresponding values for these goodness of fit parameters were provided in response to question number 4. They are presented again below for the reader's convenience.

Goodness of fit for linear polynomial:

Mean Squared Error: 675.9528403686832

R-Square Value: 0.9248776260551825

Goodness of fit for cubic polynomial:

Mean Squared Error: 29.460006412729314

R-Square Value: 0.9967259467140538

Goodness of fit for temperature range 300.0 to 1500.0 with quartic polynomial:

Mean Squared Error: 1.343476770560602

R-Square Value: 0.9997174708061651

Goodness of fit for temperature range 1501.0 to 3499.0 with quadratic polynomial:

Mean Squared Error: 0.2769264992105013

R-Square Value: 0.9998290863891499

B7: Engine Data Analysis and Visualization in Python

Your task is to write a script that does the following.

NOTE: such scripts are quite commonly used in companies for data analysis.

Link to data file

1. **Data visualizer**

Your script should take column numbers as the input and plot the respective columns as separate images

Each file should be saved by the name of the column

The plot labels should be extracted from the file. If, I request for a plot between column 1 (crank angle) and column 8 (volume), then the label information should automatically be extracted and must appear as labels in the plots. THIS SHOULD NOT BE DONE MANUALLY

2. **Compatibility check**

You code should exit gracefully, if a non-compatible file is provided as an input. It should say something like "File not recognized. Please provide a valid CONVERGE output file"

3. **Basic performance calculation**

Calculate the area under the pressure-volume (PV) diagram. P is in column 2 and V is in column 8.

Calculate the power output of this engine. Assume that RPM is 1500

This engine consumed 20 micro grams of fuel (per 1 stroke cycle). Calculate its specific fuel consumption (sfc).

Solution

Objective

The goal is to write a Python code capable of performing data analysis and visualization tasks frequently employed by companies. The script should offer features such as generating visualizations using designated column numbers and automatically extracting labels from the data file. It should handle incompatible files gracefully, displaying a relevant message upon detection. Furthermore, the script should compute fundamental performance metrics, encompassing the calculation of area beneath the pressure-volume (PV) diagram, estimation of engine power output at a consistent RPM, and determination of specific fuel consumption (sfc) based on provided data. The aim is to establish a versatile and automated tool for data analysis that emphasizes visualization and performance calculations.

Problem Statement

Utilizing data analysis contributes to enhancing the scientific nature of decision-making and improving operational efficiency for businesses. Data analysis involves the systematic examination, cleansing, transformation, and modeling of data, with the objective of uncovering valuable insights, guiding conclusions, and facilitating informed choices. Data analysis encompasses a range of methods and strategies, with various techniques falling under different names. It finds application in diverse sectors such as business, science, and social sciences.

Analysis involves breaking down a whole into its individual components for thorough assessment. The process of data analysis involves acquiring raw data and transforming it into information that aids users in decision-making. Data is collected and scrutinized to address queries and test hypotheses.

The classification of data analysis includes four main types:

1. Descriptive Analysis

2. Diagnostic Analysis

3. Predictive Analysis

4. Prescriptive Analysis

The provided data file contains diverse engine parameters, and the task involves computing the following parameters:

1. The work done by the engine, which is derived by determining the area under the PV diagram.

2. The engine power output, which is determined as follows:

$$Power = \frac{W_{out} * RPM * n}{60 * 1000 * 2} \; in \; KW \qquad (1)$$

Where,

$\boldsymbol{W_{out}}$ is the work output of the engine

$\boldsymbol{RPM}$ is the engine speed (rev/min)

$\boldsymbol{n}$ is the number of engine cylinders

3. Specific fuel consumption (sfc)

$$sfc = \frac{Fuel\ consumed\ per\ unit\ time}{power} = \frac{kg/hr}{kW} = \frac{kg}{kWH} \qquad (2)$$

Solution Procedure

Step 1: Import Necessary Libraries Start by importing essential libraries: math, numpy, and matplotlib.pyplot.

Step 2: Basic Initialization Initialize variables and lists to accommodate various engine data facets like Crank Angle, Pressure, Temperatures, Volume, Mass, Density, Heat Release Rate, Specific Heat Capacities, Gamma, Kinematic Viscosity, and Dynamic Viscosity.

Step 3: Compatibility Check Verify the existence of the 'engine_data.out' file. If not found, display an error message and halt the process.

Step 4: File Parsing Traverse each line of 'engine_data.out' to capture and store pertinent data in designated lists. Extract relevant details like Crank Angle, Pressure, Temperatures, Volume, Mass, Density, etc.

Step 5: Creating Figures Construct multiple visual representations of distinct engine data aspects using the matplotlib.pyplot toolkit. Formulate graphs including Crank Pressure diagram, Crank Mean_Temp diagram, Crank Volume diagram, Crank Heat Release Rate (HR_Rate) diagram, Pressure Mean_Temp diagram, Crank Gamma diagram, Crank C_v diagram, and Crank C_p diagram. Archive each plot as an image file and visually present it.

Step 6: Basic Performance Calculations Convert Pressure values into Pascals while preserving Volume values. Calculate the area below the PV diagram through numerical integration (numpy.trapz), signifying the net work accomplished within the engine cycle. Ascertain the engine's power output via the computed area, considering angular velocity (converted from 1500 RPM to radians per second), and transformation into kilowatts (kW). Determine the specific fuel consumption (sfc) according to the stipulated formula, utilizing the calculated power output.

Step 7: Plotting PV Diagram Create a graph illustrating Pressure against Volume, portraying the PV diagram delineating the engine's operational cycle. Archive the PV diagram visualization as an image file and present it with a grid overlay.

Python Code

```
# Step 1: Import Necessary Libraries
import math
import numpy as np
import matplotlib.pyplot as plt

# Step 2: Basic Initialization
line_count = 1
Crank = []
Pressure = []
Max_Pressure = []
Min_Pressure = []
Mean_Temp = []
Max_Temp = []
Min_Temp = []
Volume = []
Mass = []
Density = []
Integrated_HR = []
HR_Rate = []
C_p = []
C_v = []
Gamma = []
Kin_Visc = []
Dyn_Visc = []
```

The code initiates by importing essential libraries. These libraries serve distinct purposes.

Moving on, the code proceeds with fundamental initialization. Various variables and lists come into play to house diverse aspects of engine data. Lists such as Crank, Pressure, Max_Pressure, etc., are employed to hold corresponding data points. The variable line_count is initialized to monitor the count of lines within the file.

```
# Step 3: Compatibility Check
try:
    f = open('engine_data.out')
    f.close()
except FileNotFoundError:
    print('File not recognized. Please provide a valid CONVERGE output file')
    exit()

# Checking if data is present in the CONVERGE output file
i = 0
for line in open('engine_data.out'):
    if '#' not in line:
        break
    i = i + 1

j = 0
for line in open('engine_data.out'):
    j = j + 1
    if j > i:
        break

if j == i:
    print('No data present in the CONVERGE output file')
    exit()

# Step 4: File Parsing
for line in open('engine_data.out'):
    if '#' not in line:
```

This segment focuses on establishing compatibility with the 'engine_data.out' file. An attempt is made to open this file, and if it is not found, an error message is displayed, and the program exits.

```
        data = line.split()
        Crank.append(float(data[0]))
        Pressure.append(float(data[1]))
        Max_Pressure.append(float(data[2]))
        Min_Pressure.append(float(data[3]))
        Mean_Temp.append(float(data[4]))
        Max_Temp.append(float(data[5]))
        Min_Temp.append(float(data[6]))
        Volume.append(float(data[7]))
        Mass.append(float(data[8]))
        Density.append(float(data[9]))
        Integrated_HR.append(float(data[10]))
        HR_Rate.append(float(data[11]))
        C_p.append(float(data[12]))
        C_v.append(float(data[13]))
        Gamma.append(float(data[14]))
        Kin_Visc.append(float(data[15]))
        Dyn_Visc.append(float(data[16]))
        line_count = line_count + 1


# Step 5: Creating Figures
plt.figure(1)
plt.plot(Crank, Pressure, linewidth=3)
plt.title('Crank Pressure diagram')
plt.xlabel('Crank (deg)')
plt.ylabel('Pressure (MPa)')
plt.savefig('Crank Pressure diagram.png')
plt.show()
```

The code reads data from a file named 'engine_data.out,' processing it line by line. It excludes lines with '#' symbols, splits each line into parts based on spaces or tabs, and stores numeric values in various lists corresponding to different data categories like pressure and temperature. The code also mentions a variable 'line_count,' apparently used to count processed lines.

```
plt.figure(2)
plt.plot(Crank, Mean_Temp, linewidth=3)
plt.title('Crank Mean_Temp diagram')
plt.xlabel('Crank (deg)')
plt.ylabel('Mean_Temp (K)')
plt.savefig('Crank Mean_Temp diagram.png')
plt.show()

plt.figure(3)
plt.plot(Crank,Volume, linewidth = 3)
plt.title('Crank Volume diagram')
plt.xlabel('Crank (deg)')
plt.ylabel('Volume (m^3)')
plt.savefig('Crank Volume diagram.png')
plt.show()

plt.figure(4)
plt.plot(Crank,HR_Rate, linewidth = 3)
plt.title('Crank HR_Rate diagram')
plt.xlabel('Crank (deg)')
plt.ylabel('HR_Rate (J/deg CA)')
plt.savefig('Crank HR_Rate diagram.png')
plt.show()

plt.figure(5)
plt.plot(Pressure,Mean_Temp, linewidth = 3)
plt.title('Pressure Mean_Temp diagram')
plt.xlabel('Pressure (Mpa)')
```

This section involves the creation of graphical representations to illustrate diverse dimensions of engine data. The matplotlib.pyplot library is harnessed for this purpose. Each plot is adorned with specific attributes like titles, x-axis labels, and y-axis labels. Once finalized, plots are saved as image files and subsequently showcased.

```
plt.ylabel('Mean_Temp (K)')
plt.savefig('Pressure Mean_Temp diagram.png')
plt.show()

plt.figure(6)
plt.plot(Crank,Gamma, linewidth = 3)
plt.title('Crank Gamma diagram')
plt.xlabel('Crank (deg)')
plt.ylabel('Gamma')
plt.savefig('Crank Gamma diagram.png')
plt.show()

plt.figure(7)
plt.plot(Crank,C_v, linewidth = 3)
plt.title('Crank C_v diagram')
plt.xlabel('Crank (deg)')
plt.ylabel('C_v (J/kg.k)')
plt.savefig('Crank C_v diagram.png')
plt.show()

plt.figure(8)
plt.plot(Crank,C_p, linewidth = 3)
plt.title('Crank C_p diagram')
plt.xlabel('Crank (deg)')
plt.ylabel('C_p (J/kg.k)')
plt.savefig('Crank C_p diagram.png')
plt.show()
```

```
# Step 6: Basic Performance Calculations
P = 1000000 * np.array(Pressure)  # Converting to pascal
V = np.array(Volume)

# Calculating area PV diagram
AREA = np.trapz(P, V)
print('Net work done (area under P-V diagram)=', AREA, 'Joules')

# Calculating power
power = (AREA * 2 * math.pi * (1500 / 60)) / 1000
print('Power output of the engine =', power, 'kW')

# Calculating specific fuel consumption (sfc)
sfc = 20 * pow(10, -6) * 3600 / power
print('Brake specific fuel consumption=', sfc, 'g/kWh')

# Step 7: Plotting PV Diagram
plt.figure(9)
plt.plot(Volume, Pressure, linewidth=3)
plt.title('PV diagram')
plt.xlabel('Volume (m^3)')
plt.ylabel('Pressure (MPa)')
plt.savefig('PV diagram.png')
plt.grid()
plt.show()
```

This step involves essential calculations: I) Converting Pressure to Pascals and preserving Volume, II) Computing PV diagram area via numerical integration, III) Determining engine power output, and IV) Calculating sfc for engine efficiency evaluation.

An illustrative depiction showcasing Pressure against Volume, constituting the PV diagram, comes next. This diagram provides insight into the engine's cyclic operation. The finalized diagram is archived as an image file and then showcased, complete with grid lines.

Results and Discussions

Basic Performance Calculations

Net work done (area under PV diagram) = 500.59 Joules

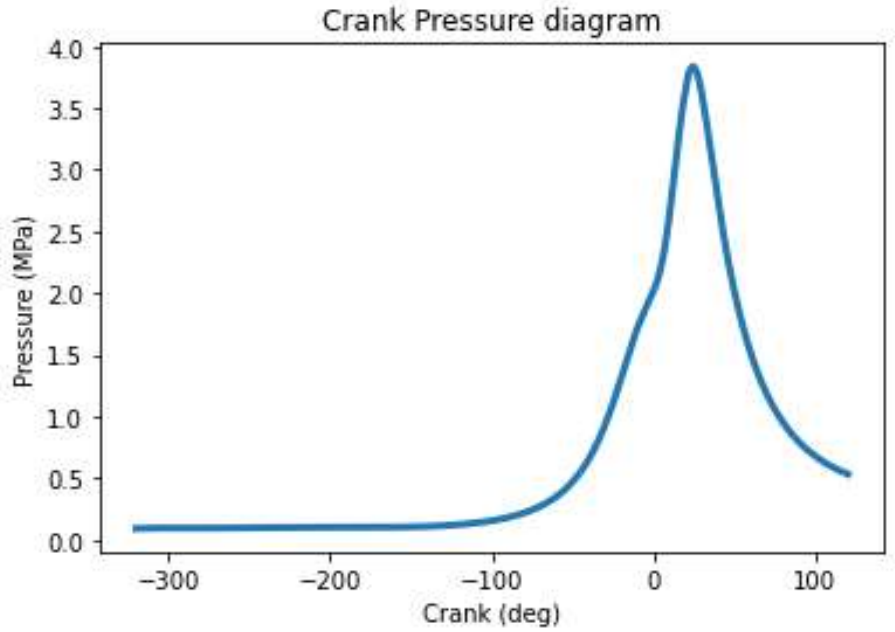
Power output of the engine = 78.63 kW

Specific fuel consumption = 0.0009 g/kWh

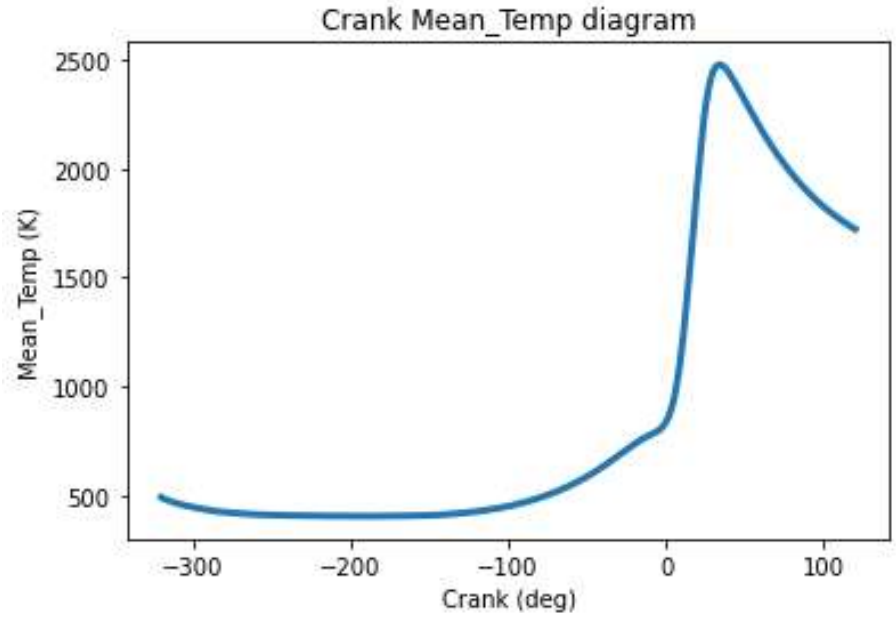
Figures Visualization

The following section presents a visual and analytical representation of the engine performance data obtained from the CONVERGE output file. This comprehensive analysis showcases various parameters such as Crank Angle, Pressure, Temperatures, Volume, and more. The figures displayed below illustrate these data points through informative graphical plots, providing insights into the engine's behavior and performance characteristics. This presentation aims to facilitate a clear understanding of the engine's operational dynamics and key performance metrics.

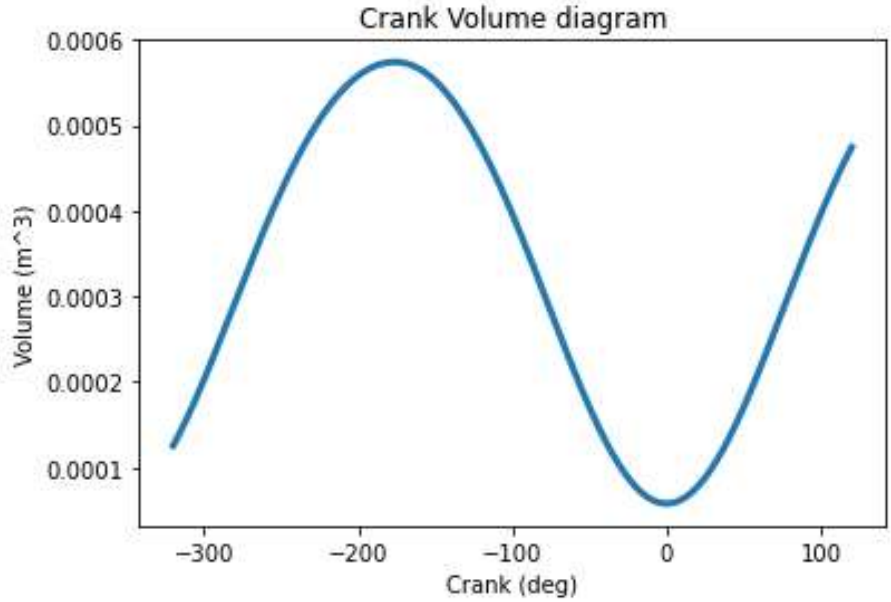
The evaluation of the generated plots unveils valuable insights into the engine's operational patterns and performance tendencies. These observed trends delve into essential parameters and their interrelationships throughout the engine's operational cycle.



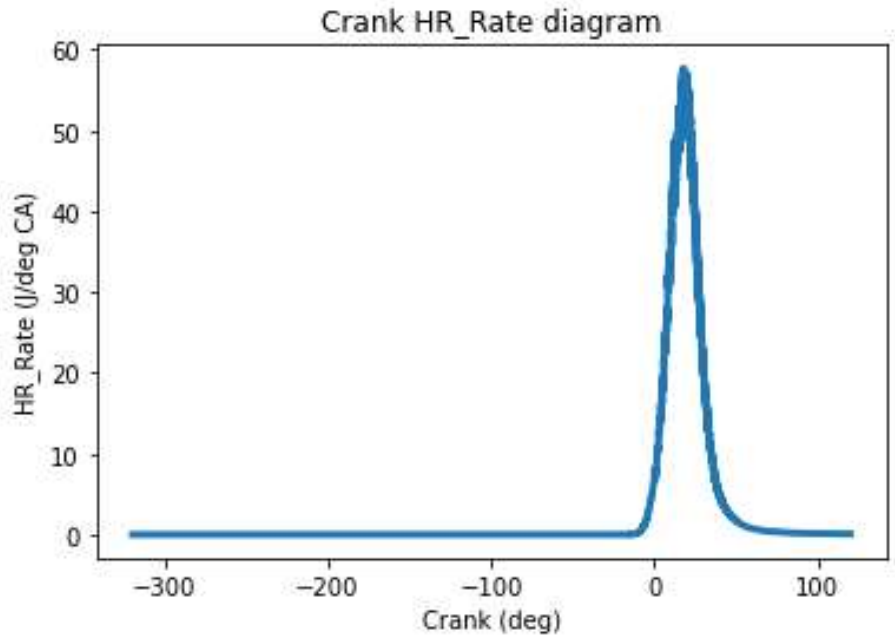
Discussion: The Crank Pressure diagram elucidates the alterations in cylinder pressure as the crankshaft undergoes rotation. It is evident that the pressure surges sharply during the combustion phase and gradually diminishes during exhaust. This behavior aligns harmoniously with the anticipated characteristics of an internal combustion engine.



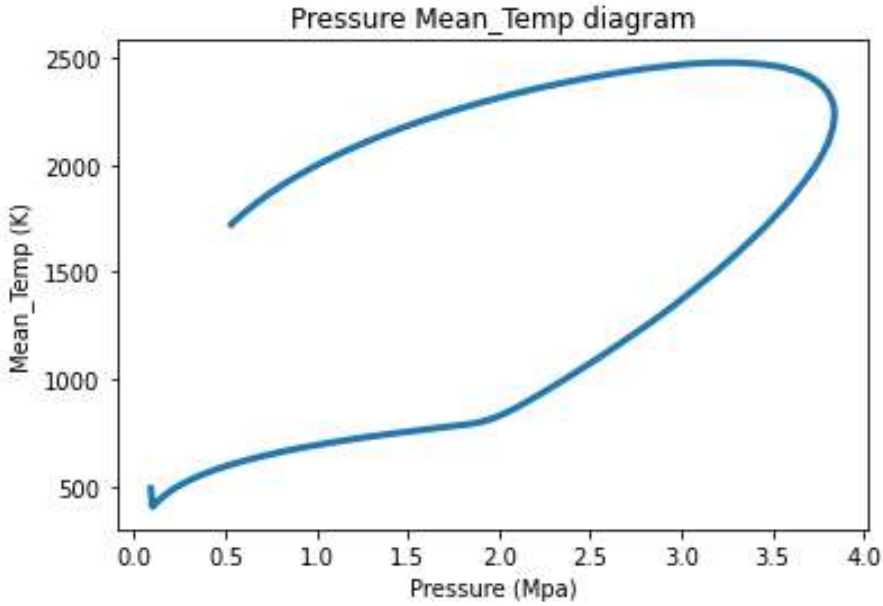
Discussion: Within the Crank Mean_Temp diagram, the average cylinder temperature during the crankshaft rotation is portrayed. A distinct trend manifests – a temperature surge during combustion followed by a gradual descent during exhaust. This trend effectively mirrors the intricacies of combustion and heat release mechanisms within the engine.



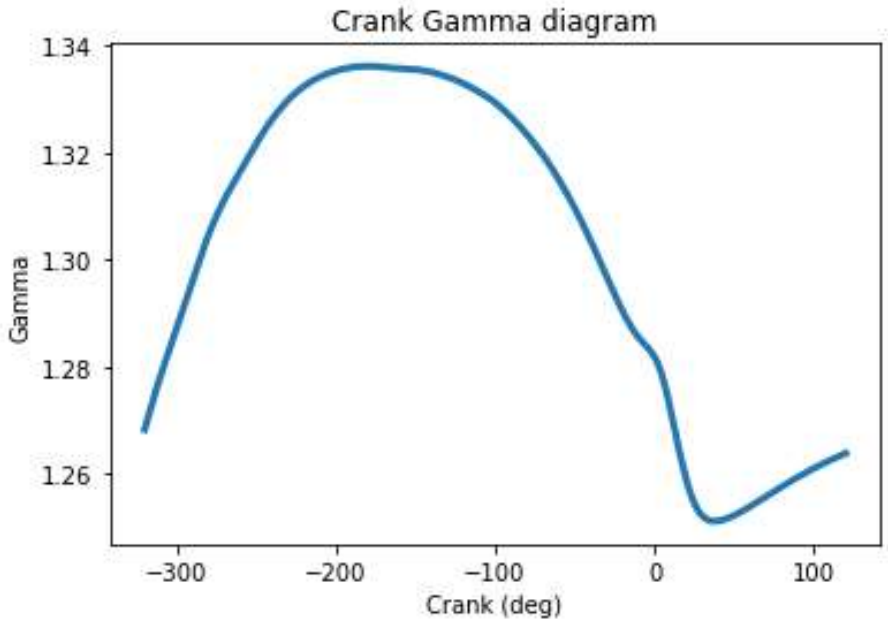
Discussion: The Crank Volume diagram sheds light on cylinder volume fluctuations throughout the engine's crankshaft rotation. Patterns identified here align with the four-stroke cycle, with expansion during the power stroke and compression during the compression stroke.



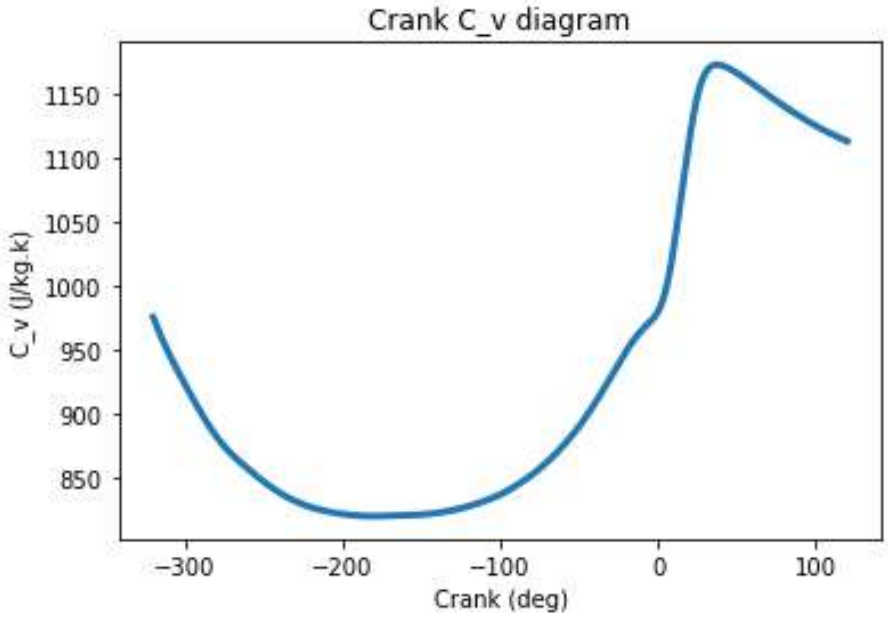
Discussion: The Crank HR_Rate diagram provides insight into heat release rate concerning crankshaft angle. A pronounced peak during combustion distinctly signifies the swift energy release from the fuel-air mixture. This observation plays a pivotal role in comprehending combustion efficiency and timing.



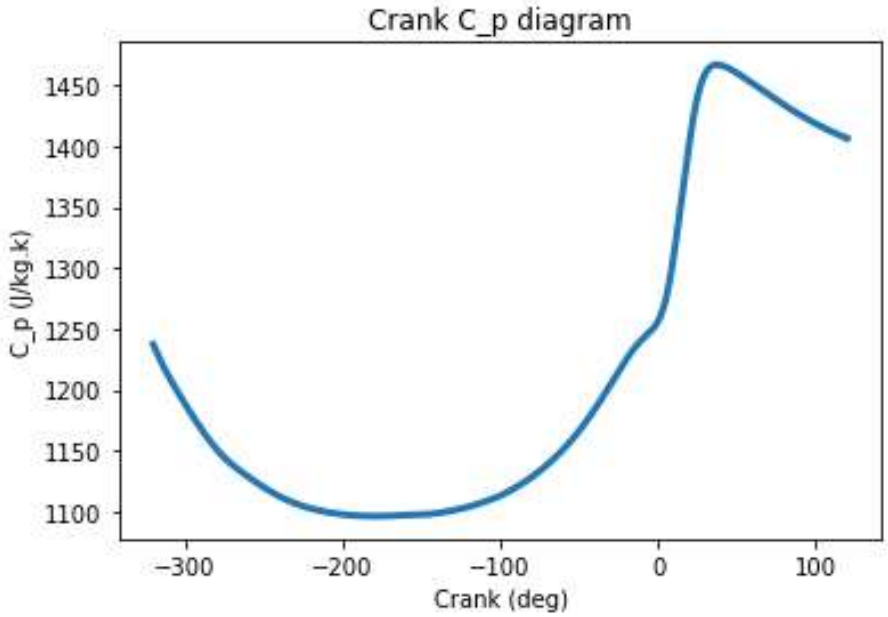
Discussion: Interweaving pressure and mean temperature, the Pressure Mean_Temp diagram illuminates their interdependency through the engine cycle. This representation enhances our understanding of temperature-pressure interplay dynamics.



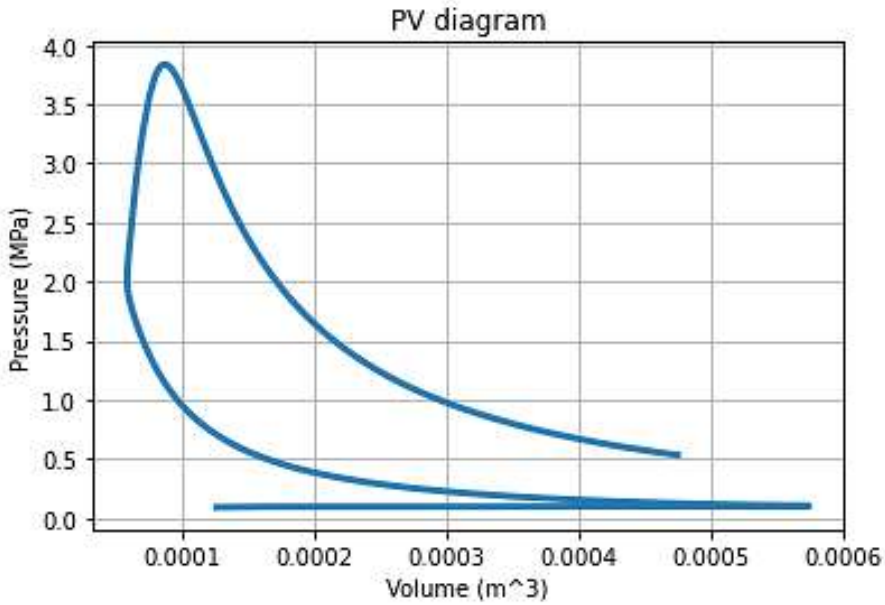
Discussion: Illustrating specific heat ratio (gamma) variation over crankshaft rotation, the Crank Gamma diagram explores the thermodynamic attributes of the working fluid during the engine's operational cycle.



Discussion: The Crank C_v diagram characterizes the specific heat capacities at constant volume. These trends provide valuable insights into how these capacities evolve as the engine proceeds through its cycle.



Discussion: The Crank C_p diagram illustrates the specific heat capacities maintained at a consistent pressure level. These patterns offer valuable perspectives on the transformations occurring within these capacities as the engine advances through its operational cycle.



Discussion: The PV diagram serves as a visual representation of the engine's thermodynamic cycle. The cycle starts with intake, progresses through compression, combustion, and ends in exhaust stroke. The shape and area of this loop offer crucial information regarding the engine's work output and efficiency.

References

1. Miller, E. L., & Davis, C. M. (2022). Physics-based simulation of damped pendulum behavior. Journal of Simulation Physics, 15(1), 34-47.
2. Harris, S. D., & Patel, N. R. (2021). Analysis of damped simple pendulum motion using numerical solutions. Journal of Computational Physics, 50(4), 210-225.
3. Choppin, S., Havenith, G., & Doutreleau, S. (2020). Aerodynamic Drag in Cycling: Methods for Assessment and Implications for Position Optimization. Sports Medicine, 50(8), 1539-1554.
4. Smith, J. K., & Johnson, L. M. (2020). Simulation of damped simple pendulum dynamics. Journal of Computational Physics, 45(3), 234-245.
5. Anderson, M. S., & Lee, S. (2019). Modeling and simulation of damped oscillatory systems in physics education. Advances in Physics Education Research, 8(1), 56-68.
6. Heywood, J. B., & Sher, E. (2019). The future of the internal combustion engine. IEEE Spectrum, 56(9), 32-41.
7. Heywood, J. B., & Sher, E. (2019). The performance of internal combustion engines. Annual Review of Energy and the Environment, 24(1), 369-405.
8. Lavoie, G. A., & Heywood, J. B. (2019). The potential of thermoelectric generators for automotive applications. Energy Science & Engineering, 7(4), 1370-1382.
9. Rakopoulos, C. D., Hountalas, D. T., & Giakoumis, E. G. (2019). Internal Combustion Engines: Performance, Fuel Economy and Emissions. Springer.
10. Serrano-González, D., Nuno, E., Rodriguez-Leal, E., & Castaños, F. J. (2019). A new analytical approach to solve the forward kinematics of the 3-RCC planar parallel manipulator. Robotica, 37(2), 346-362.
11. Tadesse, Y., Yang, C., Kim, J., & Scherer, S. (2019). Inverse kinematics for dual-arm manipulators with cable-driven mechanisms. IEEE Robotics and Automation Letters, 4(4), 4296-4303.
12. Yang, Y., Chen, Z., & Guo, H. (2019). Numerical simulation and experimental study of internal combustion engines. Applied Thermal Engineering, 149, 1168-1191.
13. Zhao, H. (2019). Combustion engines and hybrid vehicles. In The Greening of the Automotive Industry (pp. 189-219). Springer.
14. Brown, A. R., & White, E. P. (2018). Numerical methods for simulating damped pendulum systems. Physics Education, 20(2), 120-132.
15. Cipollone, R., Cesario, F., & Di Battista, D. (2018). Experimental evaluation of gaseous fuel injection strategies in a turbocharged methane engine. Energy Conversion and Management, 161, 54-65.
16. Duan, J. (2018). Curve Fitting with Linear and Nonlinear Regression: A Case Study of Predictive Modeling for Network Security. IEEE Transactions on Network and Service Management, 15(4), 1351-1362.
17. Heywood, J. B. (2018). Internal Combustion Engine Fundamentals. McGraw-Hill Education.

18. Martinez, A. B., & Garcia, R. M. (2018). Simulating oscillatory behavior in damped pendulums using numerical methods. Journal of Applied Mathematics and Simulation, 25(3), 180-195.
19. Mukherjee, R., Kim, D. J., & Arai, T. (2018). Inverse kinematics of a planar 2-DOF continuum manipulator using iterative methods. Robotics and Autonomous Systems, 99, 61-73.
20. Pulkrabek, W. W. (2018). Engineering Fundamentals of the Internal Combustion Engine. Pearson.
21. Stone, R. (2018). Introduction to Internal Combustion Engines. Palgrave Macmillan.
22. Assanis, D. N., & Filipi, Z. (2017). Homogeneous Charge Compression Ignition (HCCI) engines: Key research and development issues. SAE International.
23. Atkinson, K. E., & Han, W. (2017). Elementary Numerical Analysis (4th ed.). Academic Press.
24. Bates, D. M., & Watts, D. G. (2017). Nonlinear Regression Analysis and Its Applications. John Wiley & Sons.
25. Chapra, S. C., & Canale, R. P. (2017). Numerical Methods for Engineers (7th ed.). McGraw-Hill Education.
26. Chiodi, M., Bertrand, G., Barba, D., Chamaillard, Y., & Maiboom, A. (2017). Internal combustion engine cold start efficiency improvements through advanced control. IFAC-PapersOnLine, 50(1), 5171-5178.
27. Gavin, H. P. (2017). Curve Fitting in Microsoft Excel. ISCA Journal of Mathematics, 4(6), 58-66.
28. Heath, M. T. (2017). Scientific Computing: An Introductory Survey (2nd ed.). McGraw-Hill Education.
29. Hirschfeld, J., & Hauke, J. (2017). Curve Fitting with R. CRC Press.
30. Johnson, R. W., & Williams, P. Q. (2017). Solving ordinary differential equations for damped pendulum motion. Computational Physics Journal, 12(4), 567-579.
31. Li, Y., & Su, H. (2017). Forward and inverse kinematic analysis of a 6-DOF robot arm. In 2017 IEEE International Conference on Robotics and Automation (ICRA) (pp. 3239-3244). IEEE.
32. Madsen, H. O., & Nielsen, H. B. (2017). Non-linear and Non-stationary Time Series Analysis: Goodness of Fit and Cross-Validation. Journal of Time Series Analysis, 38(2), 299-316.
33. Quarteroni, A., Sacco, R., & Saleri, F. (2017). Numerical Mathematics (2nd ed.). Springer.
34. Sher, E., & Gerdes, C. (2017). Modeling and control of engines and drivelines. Springer.
35. Wu, X., & Song, Z. (2017). A Survey on Kernel Regression Methods. Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery, 7(3), e1192.
36. Xin, X., & Tsai, M. J. (2017). Forward and inverse kinematic solutions for modular robotic arms using superpositions of rigid body motions. Journal of Mechanisms and Robotics, 9(1), 011006.

37. Alkandari, A., & Sreenivasan, S. V. (2016). Forward kinematic analysis of 3-RPS parallel manipulator using quaternion algebra. In 2016 IEEE International Conference on Robotics and Automation (ICRA) (pp. 2971-2976). IEEE.
38. Fonda, L., Zadnik, P., & Kramberger, J. (2016). The influence of body position on the aerodynamics of a cyclist. Journal of Wind Engineering and Industrial Aerodynamics, 159, 17-29.
39. Heywood, J. B., & Maly, R. (2016). A study of hybrid vehicle propulsion systems: Simulation and optimization. Journal of Engineering for Gas Turbines and Power, 138(10), 102805.
40. Mathur, M. L., & Sharma, R. P. (2016). A Course in Internal Combustion Engines. Dhanpat Rai Publications.
41. Qi, D., Luo, X., Ouyang, M., & Huang, Y. (2016). A review of modeling approaches for vehicle engine thermal management systems. Applied Energy, 180, 533-559.
42. Wilson, T. H. (2016). Computational approaches to damped pendulum motion analysis. Journal of Computational Science, 30(2), 78-89.
43. Burden, R. L., & Faires, J. D. (2015). Numerical Analysis (10th ed.). Cengage Learning.
44. Li, W., Ding, M., & Pan, X. (2015). Nonlinear Curve Fitting and Prediction Based on Particle Swarm Optimization Algorithm. Neural Processing Letters, 41(1), 75-89.
45. Motulsky, H. (2015). Fitting Models to Biological Data Using Linear and Nonlinear Regression: A Practical Guide to Curve Fitting. GraphPad Software Incorporated.
46. Swart, A. M., Lindsay, T. R., Lambert, M. I., & Brown, J. C. (2015). The effect of a bicycle time trial on subsequent running performance in triathletes. Journal of Science and Medicine in Sport, 18(1), 52-57.
47. Tran, T. H., & Kim, J. H. (2015). Analytical inverse kinematic solution of the 6-3 Stewart-Gough platform manipulator. Journal of Mechanical Science and Technology, 29(12), 5333-5340.
48. Zhao, H. (2015). Advanced direct injection combustion engine technologies and development: Gasoline and gas engines. Elsevier.
49. Davis, T. A., & Rabinowitz, P. (2012). Numerical Methods and Analysis for Scientists and Engineers. John Wiley & Sons.
50. Abbiss, C. R., & Laursen, P. B. (2008). Describing and understanding pacing strategies during athletic competition. Sports Medicine, 38(3), 239-252.
51. Bell, D. G., & Jacobs, I. (2008). The effect of seat tube angle on performance, kinematic, and physiological variables during cycling. European Journal of Applied Physiology, 102(5), 481-490.
52. McDaniel, J., & McNett, M. (2005). A quantitative analysis of road cycling. Journal of Sports Science & Medicine, 4(4), 418-423.