UNIVERSITÀ DEGLI STUDI DI PADOVA

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

CORSO DI LAUREA IN INGEGNERIA INFORMATICA

Towards a Pythonic framework for control and analysis of magnetic levitation systems

Relatore: Prof. Damiano Varagnolo

Laureando: Alessandro Lincetto

Correlatori: PhD Candidate Hans A. Engmark
PhD Candidate Kiet Tuan Hoang

ANNO ACCADEMICO 2022 – 2023

Data di laurea: 21 Novembre 2023

Ai miei genitori

Paola e Luca

Abstract

Magnetic levitation represents a fascinating technology with applications in various fields, from scientific research to industry. This thesis focuses on the development of a Python software framework for the control and analysis of magnetic levitation systems. We will present the mathematical foundations governing magnetic levitation and provide an open-source Python implementation available on GitHub. We will discuss the major challenges in implementing the software in Python, highlighting the differences compared to MATLAB. Comparisons will be made between simulations performed using the software and results obtained with MATLAB, with an analysis of the speed differences between the two platforms. One of the key findings of this research is that Python is fundamentally slower than MATLAB in executing magnetic levitation simulations. This issue will be examined in detail, and solutions will be proposed to enhance the performance of the Python software. In particular, Just-in-time (JIT) compilation techniques will be considered to accelerate code execution. This thesis will analyze different approximations for the elliptic functions ellipk and ellipe used in the software, and compare their accuracy and performance. In the end, the LQR controller was simulated using MATLAB and Python.

The primary objective of this work is to construct a comprehensive Python framework for the control and analysis of magnetic levitation, with the ambition of contributing to the advancement of research in this field.

Abstract

La levitazione magnetica rappresenta una tecnologia affascinante con applicazioni in diversi settori, dalla ricerca scientifica all'industria. Questa tesi si concentra sulla creazione di un software in Python per il controllo e l'analisi della levitazione magnetica. Verranno presentati i fondamenti matematici che governano la levitazione magnetica e verrà fornita un'implementazione Python open-source disponibile su GitHub. Saranno discussi i principali ostacoli nell'implementazione del software in Python, evidenziando le differenze rispetto all'uso di MATLAB. Verranno eseguite comparazioni tra le simulazioni effettuate con il software e i risultati ottenuti con MATLAB, con un'analisi delle differenze di velocità tra le due piattaforme. Uno dei principali risultati emersi dalla ricerca è che Python risulta più lento rispetto a MATLAB nell'esecuzione delle simulazioni di levitazione magnetica. Questo problema sarà esaminato in dettaglio e saranno proposte soluzioni per migliorare le prestazioni del software Python. In particolare, verranno considerate le tecniche di compilazione Just-in-time (JIT) per accelerare l'esecuzione del codice. Questa tesi analizzerà diverse approssimazioni per le funzioni ellittiche ellipk ed ellipe utilizzate nel software e ne comparerà l'accuratezza e le prestazioni.

Infine sono state eseguite simulazioni del controllore LQR in MATLAB e in Python. L'obiettivo principale di questo lavoro è costruire un framework completo in Python per il controllo e l'analisi della levitazione magnetica, con l'ambizione di contribuire al progresso della ricerca in questo campo.

Contents

Listings

List of Figures

Chapter 1

Introduction

1.1 Overview

Magnetic levitation technology, also known as "maglev," stands as one of the most incredible innovations in the fields of transportation and technology. It relies on the fundamental principle of magnet interaction, specifically using magnetic fields to lift and guide objects without any physical contact with a supporting surface.

In the realm of transportation, magnetic levitation systems offer revolutionary advantages, including the complete absence of mechanical friction between the vehicle and the track, enabling incredible speeds and unprecedented efficiency. These systems are often employed for high-speed trains, such as the renowned maglev train, which travels at speeds exceeding 500 km/h. Magnetic levitation is also gaining interest for urban transport applications, with many cities worldwide exploring suspended magnetic public transit projects.

Beyond transportation, magnetic levitation technology finds applications in areas like scientific research, high-efficiency cooling device manufacturing, and the aerospace industry.

1.2 MagLev Project

In previous projects, a magnetic levitation platform has been created, utilizing a combination of permanent magnets and electromagnets under the control of a high-performance Teensy 4.1 microcontroller. The Teensy 4.1 uses an ARM Cortex-M7 processor running at 600 MHz, making it among the fastest microcontrollers available.

The platform is equipped with eight permanent magnets arranged in a circular configuration and four solenoids controlled by two A4950 motor drivers. These elements work together to stabilize a magnetic disk suspended in mid-air, referred to as "levmag." Feedback is provided through mangetic measurements from three TLE493D Hall-effect sensors, integrated in the system.

The platform's structure is enclosed within a transparent plexiglass frame. The control electronics are situated in the lower section, while the second section accommodates the permanent magnets that generate a static magnetic field to stabilize the object along the z-axis. The third section is dedicated to the solenoids, which collaboratively stabilize the object along the x and y axes. Positioned just below the point of levitation, the upper section contains the Hall-effect sensor, ensuring accurate position estimation.
The overarching aim of this project, originally conceived by a Norwegian student, is to transform this platform into an educational control engineering laboratory.

1.3 My contribution

In the context of this project, I had to tackle the challenge of translation the MATLAB code into Python, with the primary goal of optimizing the software, improving overall performance and efficiency. During this process, I noticed significant differences in simulation times between the two languages, with particular attention focused on two key functions: ”ellipk” and ”ellipe.”

The most noticeable divergences emerged during the conversion regarding the performance of the ”ellipk” and ”ellipe” functions. Initially, simulations in Python seemed to require considerably more time than their MATLAB counterparts. This prompted me to closely examine how these functions were handled in the two languages, revealing substantial differences in the underlying mathematical operations.

Once the code transposition process was completed, I proceeded with software testing, using a simulation based on a Linear Quadratic Regulator (LQR) controller to evaluate the effectiveness of the optimizations made to the Python code. This allowed me to verify how the changes had affected the overall performance of the control system and ensure that the software adequately met the specific project requirements.

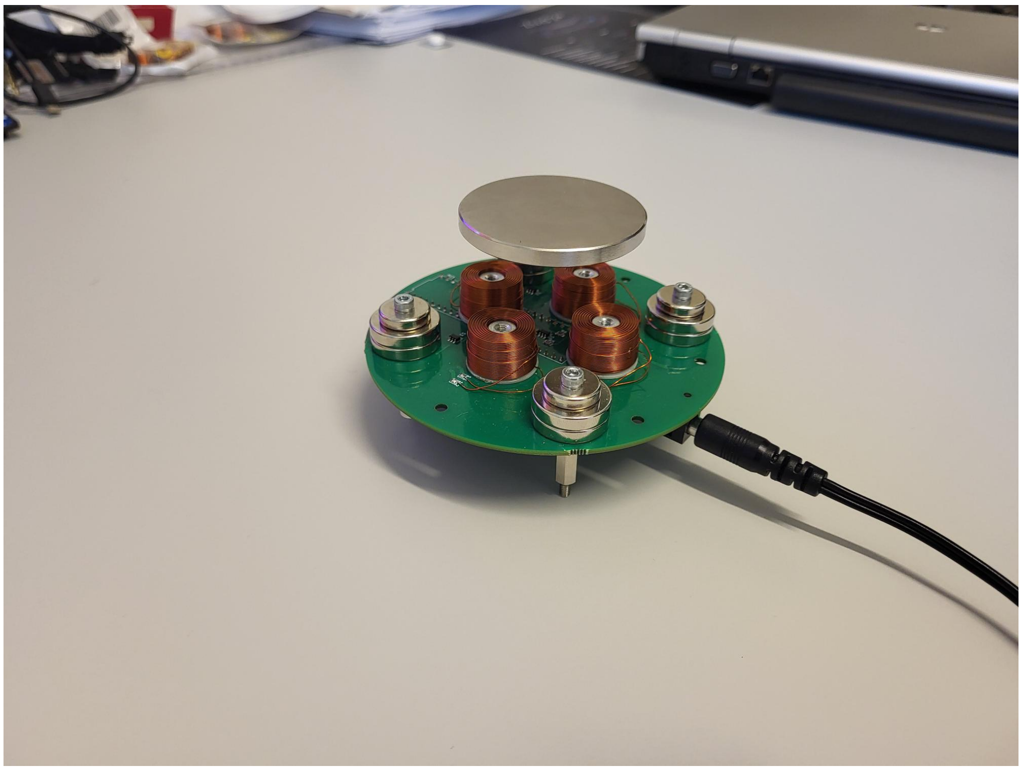

Figure 1.1: MagLev System

Chapter 2

Mathematical models

In this chapter, the focus will be on exploring fundamental calculations essential for the analysis of the magnetic field generated by a cylindrical coil.
First of all I need to compute k^2 that represents the nonlinearity of the magnetic field, providing insights into the geometry of magnetic field lines. The second parameter that I need is the scaling factor c that normalizes the involved physical quantities, adapting them to the system's dimensions.
Eliptic integrals are mathematical tools are used to calculate the complex magnetic quantities in the MagLev system. These integrals will be a topic of discussion several times in this thesis. In this case I have to calculate two types of elliptic integrals, the ellipe and the ellpik functions [8, 9].
The computation of the elliptic parameter k^2, the scaling factor c, complete elliptic integrals $K(k^2)$ and $E(k^2)$, and the utility functions will be discussed in detail. Furthermore the calculations related to the base magnetic field, force, and torque will be presented below.

2.1 Calculation of Elliptic Parameter k^2

The elliptic parameter k^2, used in magnetic field calculations, is determined by the Biot-Savart law[12]:

$$\mathbf{B}(\mathbf{r}) = \frac{\mu_0 I}{4\pi} \int \frac{R\, d\phi}{\sqrt{1 - k^2 \sin^2 \phi}} \tag{2.1}$$

We introduce k^2 as:

$$k^2 = \frac{4R\rho}{(R+\rho)^2 + (z - z')^2} \tag{2.2}$$

This k^2 represents the ellipticity in elliptic integrals and is used to compute magnetic fields in cylindrical coordinates.

2.2 Calculation of Scaling Factor c

The scailing factor c is calculated by Biot-Savart's law. For a coil in cylindrical coordinates, we define:

$$c = \frac{\mu_0 N I}{4\pi\sqrt{R\rho}} \tag{2.3}$$

Where N is the coil's number of windings. This c is the scaling factor used to compute the magnetic field in cylindrical coordinates.

2.3 Calculation of Elliptic Integrals

Considering again Biot-Savart's law:

$$\mathbf{B}(\mathbf{r}) = \frac{\mu_0}{4\pi} \int \frac{I\, d\mathbf{l} \times (\mathbf{r} - \mathbf{r}')}{|\mathbf{r} - \mathbf{r}'|^3} \tag{2.4}$$

For a cylindrical coil, we define the elliptic parameter k^2 as:

$$k^2 = \frac{4R\rho}{(R+\rho)^2 + (z - z')^2} \tag{2.5}$$

Utilizing k^2, we express the complete elliptic integrals of the first and second kind ($K(k^2)$ and $E(k^2)$) as:

$$K(k^2) = \int_0^{\frac{\pi}{2}} \frac{d\phi}{\sqrt{1 - k^2 \sin^2 \phi}} \tag{2.6}$$

$$E(k^2) = \int_0^{\frac{\pi}{2}} \sqrt{1 - k^2 \sin^2 \phi}\, d\phi \tag{2.7}$$

These are the formulas for the complete elliptic integrals used to compute the magnetic field generated by the cylindrical coil.

2.4 Calculation of Complete Elliptic Integrals

Initiating with Biot-Savart's law:

$$\mathbf{B}(\mathbf{r}) = \frac{\mu_0}{4\pi} \int \frac{I\, d\mathbf{l} \times (\mathbf{r} - \mathbf{r}')}{|\mathbf{r} - \mathbf{r}'|^3} \tag{2.8}$$

For a cylindrical coil, we define the elliptic parameter k^2 as:

$$k^2 = \frac{4R\rho}{(R+\rho)^2 + (z - z')^2} \tag{2.9}$$

Utilizing k^2, we express the complete elliptic integrals of the first and second kind ($K(k^2)$ and $E(k^2)$) as:

$$K(k^2) = \int_0^{\frac{\pi}{2}} \frac{d\phi}{\sqrt{1 - k^2 \sin^2 \phi}} \tag{2.10}$$

$$E(k^2) = \int_0^{\frac{\pi}{2}} \sqrt{1 - k^2 \sin^2 \phi}\, d\phi \tag{2.11}$$

These integrals are essential in calculating the magnetic field components (B_ρ and B_z) for a cylindrical coil. The final formulas for B_ρ and B_z in terms of elliptic integrals are:

$$B_\rho = -\frac{z}{\rho} \cdot c \cdot \sqrt{k^2} \left(K(k^2) - \frac{\rho^2 + R^2 + z^2}{(\rho - R)^2 + z^2} \cdot E(k^2) \right) \tag{2.12}$$

$$B_z = c \cdot \sqrt{k^2} \left(K(k^2) - \frac{\rho^2 - R^2 + z^2}{(\rho - R)^2 + z^2} \cdot E(k^2) \right) \tag{2.13}$$

2.5 Calculation of base magnetic field, force and torque

The base magnetic field acts on the MagLev. It is mathematically discretized into n points, and the total force and torque is the sum of the force and torque acting on all of these points. To calculate the force and torque the software use two function described below.

First of all the software implement a function `calculate_base_magnetic_field` that calculate the base magnetic field. In detail this function computes the magnetic field generated in a magnetic levitation system, taking into account two main components: permanent magnets and solenoids. For each permanent magnet and solenoid, the function calculates the local magnetic field based on their positions and associated current. These local magnetic fields are then summed to obtain the overall magnetic field. The final result is a representation of the three-dimensional magnetic field produced by the magnetic levitation system.

There is another function to compute the force and torque called `calculate_force_and_torque`. This function start with it extractiion of the physical dimensions of permanent magnets, that are neccessary on the magnetic field and subsequent calculations. After that this function use three-dimensional grid for angular calculations, allowing for a detailed analysis of the position and orientation of objects within the system. The conversion of polar coordinates to Cartesian coordinates simplifies this analysis.

The function also calculates the local magnetic field based on the objects' positions and the currents in solenoids. Furthermore, it determines the tangent vector to the objects' trajectories, providing the information for calculating the exerted forces.

In the end it returns two vectors: f, representing the total force acting on the objects, and t, representing the total torque applied.

2.6 Utility functions

There are three functions that are useful for calculating various things in the software. The first function, *cartesian_to_polar*, allows the conversion from cartesian coordinates to polar coordinates, returning the polar angle, radial distance, and the z-coordinate. The second function, *polar_to_cartesian*, performs the inverse operation, converting polar coordinates back to cartesian coordinates. Lastly, the third function, *calculate_rotation_matrix*, computes a rotation matrix based on specific rotation angles around the x, y, and z axes.

2.6.1 Cartesian to polar function

This function[13] converts cartesian coordinates (x, y, z) into polar coordinates (ϕ, ρ, z). The first step is the calculation of the polar angle:

$$\phi = \text{arctan2}(y, x)$$

After that this function have to calculate the radial distance:

$$\rho = \sqrt{x^2 + y^2}$$

The coordinate z remains unchanged during conversions between polar and Cartesian coordinates because it always represents the vertical height and does not change regardless of the coordinate system used.

2.6.2 Polar to cartesian

This function[15] converts polar coordinates (ϕ, ρ, z) into cartesian coordinates (x, y, z). First, the x coordinate needs to be calculated:

$$x = \rho \cdot \cos(\phi)$$

After computing x, the next step is to determine the y coordinate:

$$y = \rho \cdot \sin(\phi)$$

The coordinate z remains unchanged for the same reason in the previous paragraph.

2.6.3 Calculate rotation matrix

This function[6] calculates a rotation matrix R by combining three elementary rotations around the x, y, and z axes. This function that calculates the rotation matrix is used to describe how a three-dimensional object is rotated around its three principal axes: x, y, and z. This matrix represents the geometric transformation that alters the position and orientation of an object in space. The rotation matrix R is obtained by multiplying the rotation matrices R_x, R_y, R_z corresponding to rotations around the x, y, and z axes, respectively:

$$R = R_z(\gamma) \cdot R_y(\beta) \cdot R_x(\alpha)$$

Where: Rotation matrix $R_x(\alpha)$ for rotation around the x axis:

$$R_x(\alpha) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(\alpha) & -\sin(\alpha) \\ 0 & \sin(\alpha) & \cos(\alpha) \end{bmatrix}$$

Rotation matrix $R_y(\beta)$ for rotation around the y axis:

$$R_y(\beta) = \begin{bmatrix} \cos(\beta) & 0 & \sin(\beta) \\ 0 & 1 & 0 \\ -\sin(\beta) & 0 & \cos(\beta) \end{bmatrix}$$

Rotation matrix $R_z(\gamma)$ for rotation around the z axis:

$$R_z(\gamma) = \begin{bmatrix} \cos(\gamma) & -\sin(\gamma) & 0 \\ \sin(\gamma) & \cos(\gamma) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

The resulting matrix R represents the desired rotation transformation.

Chapter 3

Differences between Python and MATLAB focusing on compilation

In this chapter I describe the main defferences between Python and MATLAB, in detail I consider the syntax, code structure, data types and in particular the compilation about this two languages. Forthermore I'll go into more detail about the elliptic function (ellipk and ellipe) and the solution about how to solve the timing compilation about this functions.

3.1 Python vs MATLAB: code, structure and data types

3.1.1 Syntax

Python and MATLAB exhibit significant differences in their syntax and code structure. In Python, indentation is a crucial role in the syntax, serving to delimit code blocks, such as those within loops or functions. In contrast, MATLAB uses the "end" keyword to signify the end of a code block.
Another difference pertains to curly braces and square brackets. In Python, curly braces are primarily used to define dictionaries and sets, while MATLAB employs square brackets [] to define matrices. Regarding the semicolon, in MATLAB, it is common practice to terminate each statement with a semicolon (;) to suppress out-

put, whereas in Python, the semicolon is optional and is often omitted to enhance code readability.
In string manipulation, Python offers greater flexibility, allowing string concatenation using the + operator. In MATLAB, on the other hand, the strcat() function is used, or string concatenation can be performed implicitly.
Concerning comments, Python uses the # symbol to denote comment text, whereas MATLAB utilizes the % symbol for commenting code. Variable assignment differs between the two languages. In Python, variables can be assigned without explicitly declaring their type, thanks to dynamic typing. In MATLAB is common practice to explicitly declare the variable type. This is the main differences between Python and MATLAB in terms of syntax.

3.1.2 Code structure

In terms of functions, Python and MATLAB have distinct approaches. In Python, functions are declared using the "def" keyword, offering a straightforward way to define and use them. Conversely, in MATLAB, functions are defined using the "function" keyword, following a more rigid structure.
Python offers a rich array of libraries for file handling, making tasks like reading, writing, and manipulating files straightforward. In contrast, MATLAB features built-in functions for reading and writing data to and from files, simplifying file-related operations within the MATLAB environment.
Regarding object orientation, Python stands out as an object-oriented language, where everything is treated as an object with associated attributes and methods. While MATLAB does support objects, it is less oriented toward object-oriented programming and generally leans more toward procedural programming.

3.1.3 Data types

In terms of numeric data types, Python and MATLAB follow distinct approaches. Python offers greater flexibility in managing numeric data types, allowing a variety of operations and conversions between different types. In contrast, MATLAB specializes in handling numeric arrays, with a strong emphasis on compatibility between matrix dimensions for arithmetic operations.
In the context of lists and arrays, Python features lists that can contain elements of different types, providing more flexibility in data management. MATLAB primarily relies on numeric matrices, which are homogeneous and require a uniform data structure.
Concerning dictionaries and similar data structures, Python offers the "dictionary" data type that allows for key-value associations, making the management of key-value data more straightforward. In MATLAB, data structures like "struct" or "map" are used for similar purposes, albeit with different syntax.

3.2 Compiling

In this section, I will describe the differences in compilation between Python and MATLAB. Python[5] is primarily an interpreted language, while MATLAB adopts ahead-of-time compilation. We will also discuss the use of Just-In-Time (JIT) compilation in both languages. In the end, I will analyze the compilation of the ellipk and ellipe functions and discuss some possible solutions to speed up the compilation.

3.2.1 Python compiling

Python is an interpreted language, it means that Python source code is executed directly, line by line, by the Python virtual machine, known as the "interpreter," as the program runs. This implies that there's no need for a separate early translation of the code into a standalone executable format before execution, as is the case with some other languages like C or C++. This approach offer two main advantages:

Development Flexibility: Since there's no separate compilation phase required, you can write, modify, and test Python code directly within your development environment. This makes Python ideal for rapid prototyping and iterative development.
Ease of Use: Interpretation significantly simplifies the process of writing and debugging code, as you can execute individual statements or blocks of code to immediately see results.
However, there is a disadvantage. Interpretation tends to be slower than running highly optimized native code because the interpreter has to analyze and translate the code on each execution. This can result in reduced performance in applications that demand intensive computations or high efficiency.

3.2.2 MATLAB compiling

MATLAB primarily uses ahead-of-time compilation so the code written in MATLAB is translated into an intermediate form of code or native code during the development phase before the program is executed. This process of ahead-of-time translation often results in an executable file or a library that can be run independently of the original source code. This comilation has two advantages:
Pre-execution Optimization: Thanks to ahead-of-time compilation, the code is optimized before the actual program execution. This means that the code can be made more efficient and faster in advance, tailored to the specifics of the computer's hardware architecture.
Independence from Source Code: Once the ahead-of-time compilation phase is completed, the result is an executable file or library that can be distributed and run on other systems without the need to share the original source code. This simplifies the distribution of MATLAB-based applications.
However before running a MATLAB program, you need to go through a compilation process. This can entail some additional planning and longer development times compared to the direct interpretation commonly found in Python.

3.2.3 Compilation Just in Time (JIT)

Just-In-Time (JIT) compilation[7] is an optimization technique used in programming languages to improve code performance. It involves translating and optimizing source code into native code during program execution. In other words, the code is not translated entirely in advance but rather during execution. In the JIT compilation there are several positive aspects:

- Dynamic Optimization: JIT compilation allows dynamic optimization of code based on real-time analysis of execution conditions. This means that the code can be optimized specifically for the hardware architecture and runtime conditions.
- Reduced Latency: Since JIT compilation occurs in real-time, it can reduce latency compared to ahead-of-time compilation, where the compilation phase must be completed before execution.
- Adaptability: JIT compilation can focus on optimizing parts of code that are frequently executed or resource-intensive, adapting to the program's needs.
- Performance Improvement: JIT compilation is used to enhance code performance, especially in applications that require intensive calculations or rapid execution.
- Portability: JIT compilation enables writing source code in a high-level language and executing it on different platforms without the need for recompilation.
- Flexibility: JIT compilation can be selectively applied only to critical parts of the code, allowing for development flexibility and reducing complexity.

3.3 ellipk and ellipe functions

This section presents the elliptic functions, in particular we will examine the ellipk function and the ellipe function. Will be presented mathematical formulas of this functions and the application of this in the MagLev Project. Furthermore will be presented the reasons why more time was spent on these functions.

3.3.1 Elliptic functions in this project

Let's thoroughly examine these functions because they are the cause of the Python code running slower compared to MATLAB. The elliptic fuction are implemented by the Python library `Scipy.special`.

In this project, the functions `ellipk` and `ellipe` are used within the functions: `calculate_magnetic_field`, `calculate_base_magnetic_field`, `calculate_force_and_torque`, and `calculate_base_magnetic_field`.

In particular the elliptic function in the `calculate_magnetic_field` function are applied for calculating essential parts of the magnetic field generated by an electric current system. These functions are necessary to obtain the radial component (`Brho`) and the component along the z-axis (`Bz`) of the magnetic field.

`ellipk` and `ellipe` are used in `calculate_base_magnetic_field` to calculate the magnetic field from permanent magnets and solenoids in the magnetic levitation system. These functions are use to determine the magnetic force generated by these components.

`ellipk` and `ellipe` are used in `calculate_force_and_torque` to calculate forces and torques due to the magnetic field acting on a thin wire in the magnetic levitation system. These functions help quantify the interaction between the thin wire and the magnetic field.

In the end this elliptical function are used in `calculate_base_magnetic_field` to calculate parts of the magnetic field generated by permanent magnets and solenoids in the magnetic levitation system that serve for determining how permanent magnets and solenoids contribute to the total magnetic field.

3.3.2 What is an eliptic function?

Elliptic functions are mathematical tools used to understand and solve problems involving shapes similar to ellipses. They are like ”special formulas” that help calculate things like the lengths of curves or areas within ellipses. The complete elliptic functions of the first and second kind, denoted as K(m)[9] and E(m)[8] respectively, are examples of these special formulas and each serves a specific purpose.

3.3.3 ellipk

The complete elliptic function of the first kind, denoted as K(m), is defined as the complete elliptic integral of the first kind:

$$K(m) = \int_0^{2\pi} \frac{1}{\sqrt{1 - m\sin^2(\theta)}} d\theta$$

In this formula:

- ”m” is a parameter ranging from 0 to 1, which influences the shape of the ellipse.
- The symbol θ represents the integration angle, varying from 0 to 2π (a full circle).
- The function K(m) returns a value representing the length of one-quarter of the ellipse, which is a quarter of the ellipse’s perimeter.

3.3.4 ellipe

The complete elliptic function of the second kind, denoted as E(m), is defined as the complete elliptic integral of the second kind:

$$E(m) = \int_0^{2\pi} \sqrt{1 - m\sin^2(\theta)} d\theta$$

In this formula:

- ”m” is a parameter ranging from 0 to 1, which influences the shape of the ellipse.

- - The symbol θ represents the integration angle, varying from 0 to 2π (a full circle).
- The function E(m) returns a value representing the area of one-quarter of the ellipse, with a major semi-axis of 1 and a minor semi-axis of $\sqrt{1-m^2}$.

3.4 Timing simulation

To understand how long the code takes to calculate the various parameters, we calculated the actual execution time using the Python library `time`. Specifically we analized the execution time of three main parameters: force, torque for multiple etas and the magnetic field calculation.

```
# Calculate the force and torque for multiple etas
etas = np.random.rand(100, 12)
u = np.zeros(4)
start_time = time.time()
for eta in etas:
    deta = maglevSystem(eta, u, params)
end_time = time.time()

print(end_time - start_time)
```

Listing 3.1: Time for calculating force and torque for muliple etas

```
# Compute and print the execution time for the
    calculate_magnetic_field function
start_time = time.time()
b = calculate_magnetic_field(phi, rho, z, R, l, I, mu0, N
    )
end_time = time.time()

print(f"Execution time: {end_time - start_time} seconds")
```

Listing 3.2: Time for calculating magnetic field

The result of this time calculation is the seguent:

```
0.23845720291137695
Execution time: 0.3296694755554199 seconds
```

Listing 3.3: Time for calculating magnetic field

3.5 Solutions for python speed

In the following section, we will focus on various strategies to optimize Python code in order to enhance its performance. We will explore the utilization of tools like Numba[1] and Cython[10], the use of optimized libraries and code parallelization.

3.5.1 Numba

Numba[1] is a Python library that provides an efficient way to improve the performance of Python functions through Just-In-Time (JIT) compilation.
Numba works by translating Python code into a format that can be executed more quickly by the CPU or GPU. It uses a Just-In-Time (JIT) compiler to generate optimized code.
I will then present an example of possible implementation of Numba in this project.

```python
from numba import jit

@jit
def ellipk(x):
    #implementation of ellipk
    pass

@jit
def ellipe(x):
    #implementation of ellipe
    pass
```

Listing 3.4: Example of Numba implementation in MagLev Project

3.5.2 Cython

Cython[10] is a powerful tool for enhancing the performance of Python, similar to Numba but with a slightly different approach. It is used to optimize Python code by compiling it into C or C++ code and then creating native Python module extensions that can be executed more efficiently. Firstly, it's necessary to annotate the Python code with explicit data type declarations, specifying which types of variables and function arguments will be used. These annotations provide crucial information to Cython about the nature of data and allow the compiler to generate highly optimized equivalent C code.

After that, the Cython code is written in a *.pyx* file. This file incorporates both the original Python code, enriched with type declarations, and additional Cython code to handle the conversion between Python and C types. When the *.pyx* file is compiled, Cython generates a native Python module, which is a binary file containing the resulting C code.

In particular, Cython optimization is the elimination of Python's dynamic type overhead. In Python standard, variables can dynamically change type during program execution. However, Cython allows for the explicit declaration of variable types. This declaration enables the compiler to generate C code that directly works with static data types, thus avoiding dynamic type overhead. The result is faster code, as type checks during execution are no longer required, significantly accelerating performance.

3.5.3 Library optimized

For optimizing performance in Python, a fundamental strategy involves the use of specialized external libraries for calculating elliptic functions such as *mpmath*[3], *scipy*[4], and *sympy*[11]. These functions play a significant role in numerous scientific and engineering applications, often requiring high precision and, at times, optimal performance.
Adopting external libraries like *mpmath*, *scipy*, and *sympy* represents a practical approach to address these specific requirements. The *mpmath* library, in particular, offers the ability to perform mathematical calculations with arbitrary precision, ensuring extremely accurate results. *Scipy*, on the other hand, provides results with standard precision but with a highly efficient implementation. Meanwhile, *sympy* offers the possibility to obtain exact symbolic expressions for ellipk and ellipe, allowing for sophisticated mathematical manipulations.

3.5.4 Code parallelization

In the process of optimizing the performance of our Python program, one of the key strategies to consider is parallelization. This technique allows us to make the most of the available processing resources on our system, speeding up computations and reducing execution times. Parallelization is particularly useful when we need to perform independent calculations on a large amount of data, as is the case with calculations of the elliptic functions ellipk and ellipe.
Implementing parallelization in Python can be achieved using libraries such as *multiprocessing* or *concurrent.futures*. These libraries enable us to divide calculations into smaller tasks and execute them in parallel on multiple cores or processes of our computer.

Chapter 4

LQR simulation

4.1 Introduction to LQR Control

A LQR controller[14] is designed to optimally regulate linear dynamic systems, minimizing a quadratic cost function with the aim of bringing the system to the specified equilibrium point.
To apply the LQR controller, it is necessary to calculate some elements: a state equation, a cost function, a linearizzation around an equilibrium poin and the optimal matrix K.
This controller utilizes the state equation to descri.be how the system evolves over time without control actions. The cost function specifies control objectives by weighing state variables and control inputs. Linearization around an equilibrium point provides an approximate linear representation of the system. Once the optimal matrix K is calculated, the LQR controller generates optimal control signals, $u(t)$, by using the difference between the current system state and the desired state.

4.2 MagLev's LQR mathematical foundation

Let's start with the state equation:

$$\dot{x}(t) = Ax(t) + Bu(t)$$

This is the coditions when $t \to \infty$ and It is also where the model is linearized to obtain the linear model used to find a good LQR gain. Where:

- $x(t)$ represents the state vector of the system.

$$\boldsymbol{x}(t) = \begin{bmatrix} 0 \\ 0 \\ z_{\text{eq}} \\ \vdots \\ 0 \\ 0 \end{bmatrix}$$

- $u(t)$ represents the control input vector.

$$\boldsymbol{u}(t) = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix}$$

- A is the state dynamics matrix that represents how the system's state variables evolve over time in the absence of control inputs. It characterizes the natural behavior of the system.

$$A = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{bmatrix}$$

- B is the control input matrix that represents how control inputs affect the changes in state variables. It describes how the system responds to control commands.

$$B = \begin{bmatrix} b_{11} & b_{12} & \dots & b_{1m} \\ b_{21} & b_{22} & \dots & b_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \dots & b_{nm} \end{bmatrix}$$

Another important part of LQR controller is the cost function. This cost function is typically expressed as the sum of squares of state variables and control inputs weighted by appropriate weight matrices Q and R:

$$J = \int_0^\infty \left(\boldsymbol{x}(t)^T \boldsymbol{Q} \boldsymbol{x}(t) + \boldsymbol{u}(t)^T \boldsymbol{R} \boldsymbol{u}(t)\right) dt$$

1. $\boldsymbol{x}(t)$ represents the state vector of the system at time t. It is the same $\boldsymbol{x}(t)$ calculated previously.

2. $\boldsymbol{u}(t)$ represents the control input vector at time t. This one is also the same as $\boldsymbol{u}(t)$ in the state equation.

3. $\boldsymbol{Q}$ is the state error weighting matrix. In this project the matrix Q is the following:

$$Q = \begin{bmatrix} 1\times10^7 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1\times10^7 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1\times10^7 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1\times10^1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1\times10^1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1\times10^2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1\times10^2 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1\times10^3 \end{bmatrix}$$

4. ***R*** is the control signal weighting matrix. For this maglev system the matrix R is the following:

$$R = \begin{bmatrix} 1\times 10^0 & 0 & 0 & \dots & 0 \\ 0 & 1\times 10^0 & 0 & \dots & 0 \\ 0 & 0 & 1\times 10^0 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & 1\times 10^0 \end{bmatrix}$$

After having linearized the system to an equilibrium point we can move on to calculate optimal gain matrix K that used to calculate control inputs as follows:

$$u(t) = -K(x(t) - x_{\text{equilibrium}}) - u_{\text{equilibrium}}$$

- K is the gain matrix obtained from the LQR controller.
- $x_{\text{equilibrium}}$ represents the desired equilibrium conditions of the system.
- $u_{\text{equilibrium}}$ represents the desired equilibrium inputs.

4.3 Implementing LQR Control in Python

In this section there are some code snippets for calculating various elements necessary to simulate an LQR controller.
The parts of code that I implemented are the following:

1. The cost function
2. Linearization around an equilibrium point
3. The optimal K matrix
4. Control signals $u(t)$

4.3.1 Cost function

```
# Matrixes Q and R
Q = np.diag([1e7]*3 + [1e1]*2 + [1e2]*2 + [1e3, 1e2, 1e2
    ])
R = 1e-0 * np.eye(len(params['solenoids']['r']))

# Cost function
def cost_function(x, u):
    return np.dot(x.T, np.dot(Q, x)) + np.dot(u.T, np.dot
        (R, u))
```

Listing 4.1: Cost function

4.3.2 Linearization around an equilibrium point

```python
def linearizeODE(dx, xLp, uLp):
    n = len(xLp)
    m = len(uLp)

    A = np.zeros((n, n))
    B = np.zeros((n, m))

    delta = 1e-5
    for k in range(n):
        delta_vec = np.eye(n)[k] * delta
        A[:, k] = (dx(xLp + delta_vec, uLp) - dx(xLp -
           delta_vec, uLp)) / (2 * delta)
    A = np.round(A, 5)

    for k in range(m):
        delta_vec = np.eye(m)[k] * delta
        B[:, k] = (dx(xLp, uLp + delta_vec) - dx(xLp, uLp
            - delta_vec)) / (2 * delta)
    B = np.round(B, 5)

    return A, B
```

Listing 4.2: Linearization around an equilibrium point

4.3.3 Optimal matrix K

```
# LQR
Ared = A[np.ix_([0, 1, 2, 3, 4, 6, 7, 8, 9, 10], [0, 1,
    2, 3, 4, 6, 7, 8, 9, 10])]
Bred = B[[0, 1, 2, 3, 4, 6, 7, 8, 9, 10], :]

# Calculation of the optimal gain matrix K
Kred = np.round(lqr(Ared, Bred, Q, R), 3)
K = np.hstack([Kred[:, :5], np.zeros((4, 1)), Kred[:,
    5:], np.zeros((4, 1))])
```

Listing 4.3: Optimal matrix K

4.3.4 Control Signals u(t)

```
# Computing control inputs using the optimal gain matrix
    K
uLp = np.zeros(len(params['solenoids']['r']))
u = -K @ (eta - etaLp) + uLp
```

Listing 4.4: Control Signals u(t)

Chapter 5

Simulation results

In this chapter I present the simulation results of the MagLev system.
The Python code in section 5.1 simulate the LQR controller, while the MATLAB code in section 5.2 use the data from LQR simulation on Python to create a graphical rappresentation of the MagLev system and the LQR controller.
It will be possible to notice that the consistency observed in the results of the two simulations indicates that the MATLAB code has been accurately translated into Python code.
In the end of this chapter the table 5.1 represents the parameters that I used in the previous simulations.

5.1 Python simulation

The simulation in the figure 5.1 demonstrates the MagLev system in closed loop with the LQR controller in Python. In particular, the three lines represents the response of the three first states of eta, x (blue), y (orange) and z (green).

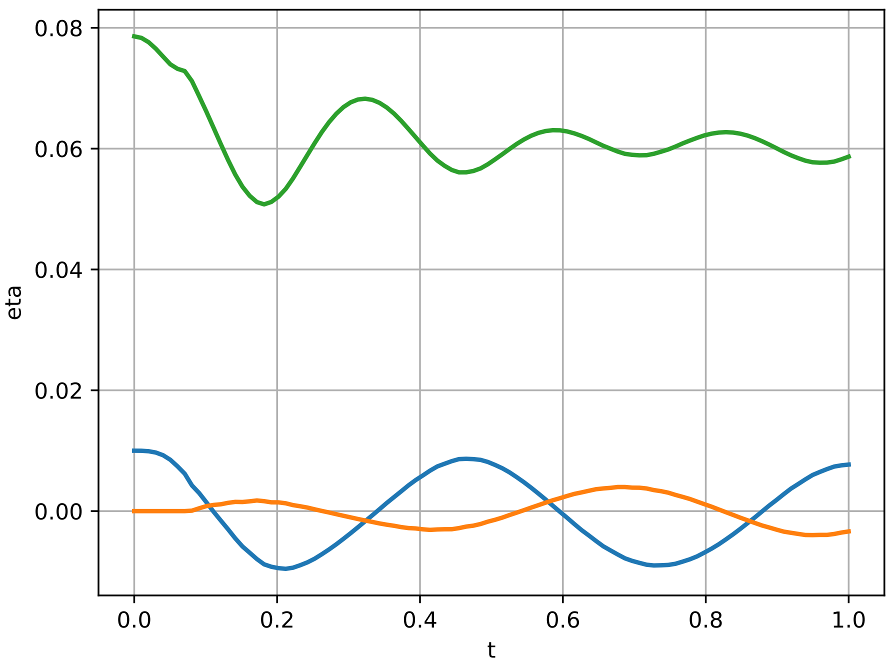

Figure 5.1: LQR controller on Python simulation

5.2 MATLAB simulation

In this section is showed a MATLAB simulation about the LQR controller of MagLev system using the data from previous simulation.
First of all it showed the MagLev system in the three-dimensional space figure 5.2. The figure 5.3 represents the simulation about the Z coordinate of my system, the figure 5.4 represents the coordinates about X and Y and the final figure 5.5 is showed the parameters α and β that are the tilt angles in the X and Y directions.

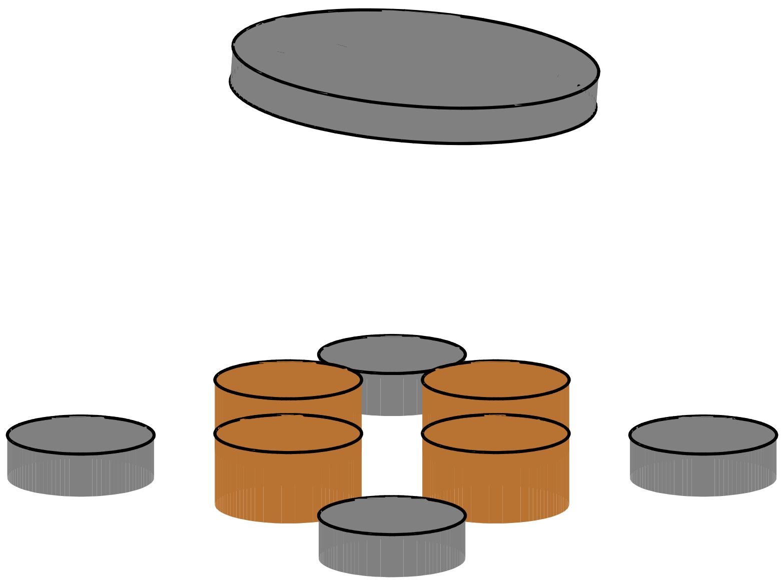
Figure 5.2: MagLev system in three-dimensional space

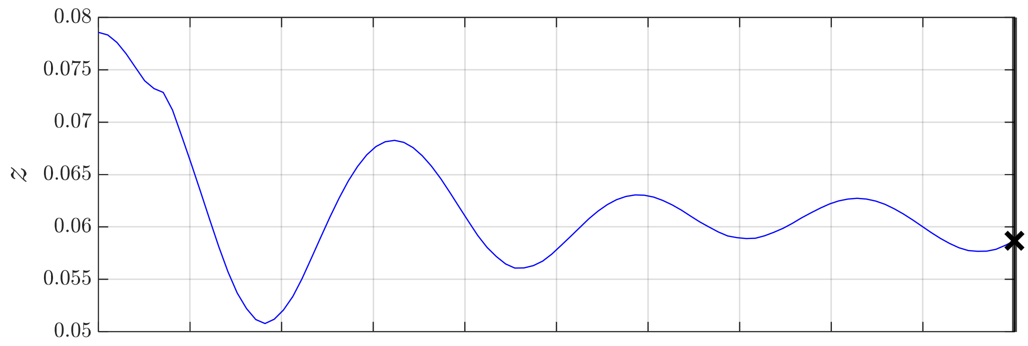

Figure 5.3: Z coordinate on MATLAB simulation

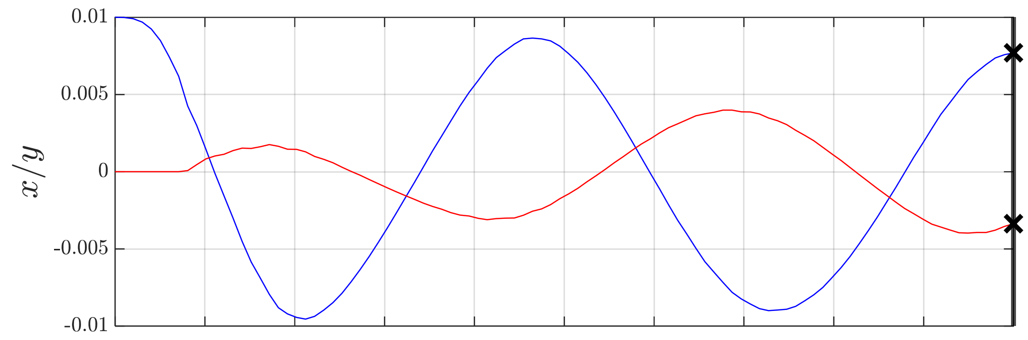

Figure 5.4: X(blue) and Y(red) coordinates on MATLAB simulation

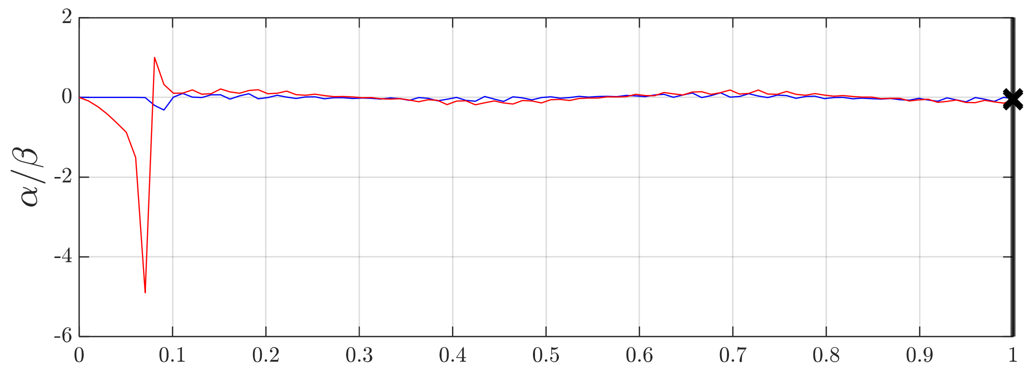

Figure 5.5: α/β parameters on MATLAB simulation

Table 5.1: Simulations parameters

Category	Parameter	Value			
		1	2	3	4
Solenoid	x	0.02	0	-0.02	0
	y	0	0.02	0	-0.02
	z	0	0	0	0
	r	0.01	0.01	0.01	0.01
	l	0.02	0.02	0.02	0.02
	nw	625			
Permanent	x	0.03	0	-0.03	0
	y	0	0.03	0	-0.03
	z	0	0	0	0
	r	0.01	0.01	0.01	0.01
	l	0.012	0.012	0.012	0.012
	J	1.22			
Magnet	coordinate	x	y	z	
	I	$0.195 * 0.025^2 *$ 0.25	$0.195 * 0.025^2 *$ 0.25	$0.195 * 0.025^2 *$ 0.5	
	r	0.025			
	l	0.005			
	J	1.22			
	m	0.195			
	n	50			
Physical	g	9.81			
	μ_0	$4\pi \times 10^{-7}$			

Chapter 6

Conclusion

6.1 Future applications

Looking in to the future, the framework of Maglev system control tool looks promising. I think to improve this framework it must be implementing (in addition to the LQR controller) a Model Predictive Control (MPC), PID (Proportional-Integral-Derivative) and other control system for the MagLev Project.
Furthermore this framework must be optimized applying the techniques mentioned in section 3.5 for a better user experience. I think also it can be considered other language to implement this software, for example Julia[2] that is popular scientific programming language that uses JIT. The long-term goal is to have a working and optimized framework for the analysis of magnetic levitation control that implements various control systems and that it is capable of simulating more complex maglev systems.

6.2 Final considerations

In this experience I had the opportunity to learn more about a magnetic levitation system and its control. In particular I learned how the magnetic levitation system available was apparently simple but in reality if you study it thoroughly you discover that it is an extremely complicated and complex system.
I also had a possibility to study in detail both the control system part (through the Python implementation of the LQR controller) and the entire compilation process with the aim of optimizing the simulation software.
I had the opportunity to work with smart people from another country, so I had to learn to work and communicate constantly in another language.
I also want to thank Professor Damiano Varagnolo, Hans A. Engmark and Kiet Tuan Hoang for their constant availability which allowed me to contribute to this project.

Bibliography

[1] Inc. Anaconda et al. *Compiling Python code with @jit.* URL: `https://numba.readthedocs.io/en/stable/user/jit.html`.

[2] JuliaLang.org contributors. *Julia 1.9 Documentation.* URL: `https://docs.julialang.org/en/v1/`.

[3] mpmath.org contributors. *Python librari for arbitrary precision floating-point arithmetic.* URL: `https://mpmath.org/`.

[4] scipy.org contributors. *Scipy v1.11.3 Manual.* URL: `https://docs.scipy.org/doc/scipy/`.

[5] Python Geeks. *Python Compilers.* URL: `https://pythongeeks.org/python-compilers/`.

[6] C. Poole Goldstein H. and J. Safko. "Classical Mechanics, 3rd Edition". In: (2002), pp. 142–144.

[7] Java Technology Edition IBM SDK. *The JIT compiler.* URL: `https://www.ibm.com/docs/en/sdk-java-technology/8?topic=reference-jit-compiler`.

[8] The SciPy. *scipy.special.ellipe.* URL: `https://docs.scipy.org/doc/scipy/reference/generated/scipy.special.ellipe.html`.

[9] The SciPy. *scipy.special.ellipk.* URL: `https://docs.scipy.org/doc/scipy/reference/generated/scipy.special.ellipk.html`.

[10] Cython Development Team. *Cython: C-Extensions for Python.* URL: `https://cython.org/#documentation`.

[11] SymPy Development Team. *How to Guides - SymPy 1.12 documentation.* URL: `https://docs.sympy.org/latest/guides/index.html`.

[12] The University of Texas. *Biot-Savart Law.* URL: `https://farside.ph.utexas.edu/teaching/355/Surveyhtml/node93.html`.

[13] Inc. The MathWorks. *Cartesian coordinates to polar or cylindrical.* URL: `https://it.mathworks.com/help/matlab/ref/cart2pol.html`.

[14] Inc. The MathWorks. *Linear-Quadratic Regulator (LQR) design.* URL: `https://it.mathworks.com/help/control/ref/lti.lqr.html`.

[15] Inc. The MathWorks. *Polar or cylindrical coordinates to Cartesian.* URL: `https://it.mathworks.com/help/matlab/ref/pol2cart.html`.