End of Master's Project

Master's in Industrial Engineering

Automatics Specialisation

Implementation of a control algorithm for a self-balancing robot in Julia in a Jetson Nano

Author: Fernando Lazcano Alvarado

Tutor: Ignacio Alvarado Aldea

Department of Systems Engineering and Automatics
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla
Seville, 2020

Trabajo Fin de Máster
Máster en Ingeniería Industrial
Intensificación Automática

Implementation of a control algorithm for a self-balancing robot in Julia in a Jetson Nano

Autor:
Fernando Lazcano Alvarado

Tutor:
Ignacio Alvarado Aldea
Profesor titular

Dpto. de Ingeniería de Sistemas y Automática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla
Sevilla, 2020

Proyecto Fin de Carrera: Implementation of a control algorithm for a self-balancing robot in Julia in a Jetson Nano

Autor: Fernando Lazcano Alvarado

Tutor: Ignacio Alvarado Aldea

El tribunal nombrado para juzgar el Proyecto arriba indicado, compuesto por los siguientes miembros:

Presidente:

Vocales:

Secretario:

Acuerdan otorgarle la calificación de:

Sevilla, 2020

El Secretario del Tribunal

To my family

To my friends

ACKNOWLEDGEMENTS

This section of most of my Master's companions is usually less effusive and commonly seen more as an imposition than a place to really thank the ones who made possible not only the project itself, but also the path followed from the beginning to where we are. I will try to express, as good as my words can, my gratitude to the ones I realise and to say sorry in advance for the ones I forgot.

This project would not have been possible without the outstanding assistance, advices, understanding, quick replies, patience and, above all, insistence and availability of Ignacio, my tutor. He has been a very close tutor and a very nice person. Thank you.

To my family, particularly to my parents, siblings, and grandparents for being understanding people when nerves won the match of nerves on the edge and made me say and do things I did not want to. Thank you all, and specially to my sister, for always being close when I needed, uniquely when those bad times we know occurred. Thank you.

I would really like to thank to the Council of the Wise, my friends, not for supporting me, but just for being… well, them. Thank you for being part of my life from the beginning.

Finally, I would like to make a special mention to L.M.G. You cannot tell me I am not an Engineer anymore, but I am sure you will come up with a way to torture me for some more years with other 'theoretically-speaking' shenanigans.

I will try to never forget that I have reached what you I am only because of myself, but also thanks to the ones which are part of my reality, who are, indeed, part of me.

Fernando Lazcano Alvarado

Seville, 2020

SYNOPSIS

This text will drive the reader through the process of developing a programme which will control an inverted-pendulum-like robot in its steady stable position, which is facing the pendulum upwards.

In addition, this thesis intends to introduce the reader to Julia—a fast programming language specifically focused on mathematical development—and the Jetson Nano—a minicomputer, comparable to Arduino, but with much better characteristics in relation to what is being tested—.

Throughout the document, it will be learnt how to use and implement complex control algorithms in Julia and MATLAB. The programmes will be created from scratch so as to learn from the very base of the language. In addition, the mathematical development of the equations which will simulate the robot's model and the controller will be explained from zero.

Several tests will be performed in order to check the response of the algorithms and programmes to different casuistry and eventually, a graph for each corresponding test will be drawn to show the behaviour of the system.

INDEX

Acknowledgements **9**

Synopsis **11**

Index **13**

Index of tables **15**

Index of figures **17**

Notation **21**

1 Preface **23**

1.1 The cart-pole *23*

1.2 The Segway *24*

1.3 Our project *25*

2 Julia **29**

2.1 Introduction *29*

2.2 Features analysis *30*

2.2.1 Technical computing 30

2.2.2 Compiling 30

2.2.3 Type checking 30

2.2.4 Speed 31

2.2.5 Multiple dispatch 32

2.2.6 Composability 32

2.2.7 Parallelism 33

2.2.8 Package manager 33

3 NVIDIA Jetson Nano **35**

- 35

3.1 Introduction *35*

3.2 Getting started *38*

3.3 Configuration of the Jetson Nano *40*

4 The system **45**

4.1 Introduction *45*

4.2 Equations of the system *46*

5 Controller design **51**

5.1 Basic concepts *51*

5.2 Mathematical development *52*

5.2.1 Model transformation 52

5.2.2 Dealing with constraints 53
5.2.3 Change to incremental model 56
5.2.4 Model integration 57
5.3 Particularization 59

6 The coding 61
6.1 Mathematical clarifications 61
6.2 Optimizer in Julia 64
6.2.1 JuMP 64
6.2.2 Ipopt 66
6.2.3 Optimization problem 67
6.3 Code in Julia 68
6.3.1 Optimization 68
6.3.2 Plotting 73
6.4 Code in MATLAB 75
6.4.1 Optimization 75
6.4.2 Plotting 78

7 Results 81
7.1 PC specifications 81
7.2 Julia's programme 81
7.2.1 In the PC 81
7.2.2 In Jetson Nano 93
7.3 MATLAB's programme 104
7.3.1 Steady reference 105
7.3.2 Changing reference 107
7.3.3 Using linearised model 110

8 Conclusions 113
8.1 Contributions 113
8.2 What is next? 114

9 References 11

INDEX OF TABLES

Table 3-1. Jetson Nano’s technical specifications. 36
Table 3-2. Comparison between Nano and Nano 2GB development kits. 37
Table 4-1. Mathematical development to obtain system's equations. 47
Table 7-1. PC characteristics. 81

INDEX OF FIGURES

Figure 1-1. Cart-pole system. 23

Figure 1-2. Pendulum's stable position. 24

Figure 1-3. Pendulum's unstable position. 24

Figure 1-4. A Segway. 24

Figure 1-5. Segway's analogy. 25

Figure 1-6. Physical implementation of the previous project. 26

Figure 1-7. Electronics scheme of the previous project. 26

Figure 3-1. NVIDIA Jetson Nano Developer Kit. 35

Figure 3-2. NVIDIA Jetson Nano module with passive heatsink. 37

Figure 3-3. Developer kit module and carrier boards: front and rear views. 39

Figure 3-4. Developer kit carrier boards: rev A02 top view. 39

Figure 3-5. Developer kit module and carrier board: rev B01 top view. 39

Figure 3-6. MPU-6050 detailed view. 41

Figure 3-7. J41 header pin layout. 42

Figure 4-1. Inverted pendulum. 45

Figure 4-2. Two-wheeled inverted pendulum example. 46

Figure 4-3. Schematic of the robot. 46

Figure 6-1. JuMP's logo. 64

Figure 6-2. COIN-OR's logo. 66

Figure 7-1. Tilt (steady reference). 82

Figure 7-2.Tilt's speed (steady reference). 83

Figure 7-3. Wheels' speed (steady reference). 83

Figure 7-4. Control action (steady reference). 84

Figure 7-5. Julia's optimization time with steady reference. 84

Figure 7-6. Julia's compiler behaviour. 85
Figure 7-7. Tilt (distant starting point). 86
Figure 7-8. Tilt's speed (distant starting point). 86
Figure 7-9. Wheels' speed (distant starting point). 87
Figure 7-10. Control action (distant starting point). 87
Figure 7-11. Tilt (changing reference). 88
Figure 7-12. Tilt's speed (changing reference). 89
Figure 7-13. Wheels' speed (changing reference). 89
Figure 7-14. Control action (changing reference). 90
Figure 7-15. Optimization time. 91
Figure 7-16. Solving time with changing setpoints. 91
Figure 7-17. Solving time (linearised model). 91
Figure 7-18. Linear vs. nonlinear tilt. 92
Figure 7-19 Linear vs. nonlinear tilt's speed. 92
Figure 7-20 Linear vs. nonlinear wheels' speed. 93
Figure 7-21 Linear vs. nonlinear control action. 93
Figure 7-22. Tilt (steady reference). 94
Figure 7-23. Tilt's speed (steady reference). 95
Figure 7-24. Wheels' speed (steady reference). 95
Figure 7-25. Control action (steady reference). 96
Figure 7-26. Julia's optimization time with steady reference (Jetson Nano). 96
Figure 7-27. Julia's compiler behaviour (Jetson Nano). 97
Figure 7-28. Tilt (distant starting point). 97
Figure 7-29. Tilt's speed (distant starting point). 98
Figure 7-30. Wheels' speed (distant starting point). 98
Figure 7-31. Control action (distant starting point). 99
Figure 7-32. Optimization time (distant starting point). 99
Figure 7-33. Tilt (changing reference). 100
Figure 7-34. Tilt's speed (changing reference). 100
Figure 7-35. Wheels' speed (changing reference). 101
Figure 7-36. Control action (changing reference). 101
Figure 7-37. Optimization time. 102
Figure 7-38. Solving time with changing setpoints. 102
Figure 7-39. Linear vs. nonlinear tilt. 103
Figure 7-40. Linear vs. nonlinear tilt's speed. 103
Figure 7-41. Linear vs. nonlinear wheels' speed. 104
Figure 7-42. Linear vs. nonlinear control action. 104
Figure 7-43. Tilt (steady reference). 105
Figure 7-44. Tilt's speed (steady reference). 105
Figure 7-45. Wheels' speed (steady reference). 106

Figure 7-46. Control action (steady reference). 106
Figure 7-47. MATLAB's optimization time with steady reference. 107
Figure 7-48. Tilt (changing reference). 108
Figure 7-49. Tilt's speed (changing reference). 108
Figure 7-50. Wheels' speed (changing reference). 109
Figure 7-51. Control action (changing reference). 109
Figure 7-52. MATLAB's computation time. 110
Figure 7-53. Linear vs. nonlinear tilt. 110
Figure 7-54. Linear vs. nonlinear tilt's speed. 111
Figure 7-55. Linear vs. nonlinear wheels' speed. 111
Figure 7-56. Linear vs. nonlinear control action. 112

NOTATION

m_r	Wheel's mass
R	Wheel's radium
M	Mass of the vehicle's body (without wheels)
L	Distance from the wheel's axle and the centre of gravity
g	Gravity
θ	Wheels' rolling angle
ϕ	Tilt with respect to the vertical
ϕ_0	Centre of gravity deviation tilt
ϕ_c	Corrected tilt
L	Lagrangian
T_{tras}	Translational kinetic energy
T_{rot}	Rotational kinetic energy
V	Potential energy
x_r	Horizontal position of the wheels
$\dot{x}_r$	Horizontal velocity of the wheels
y_r	Vertical position of the wheels
$\dot{y}_r$	Vertical velocity of the wheels
x_m	Horizontal position of the centre of mass
$\dot{x}_m$	Horizontal velocity of the centre of mass
y_m	Vertical position of the centre of mass
$\dot{y}_m$	Vertical velocity of the centre of mass
x_k	State of the system at sample k.
u_k	Control action at sample k.
y_k	Output of the system at sample k.
A	State matrix.
B	Input matrix.
C	Output matrix.
D	Feedthrough matrix.
V	Cost function.
Q	States weighting matrix.
R	Inputs weighting matrix.
H	Quadratic objective term.

f	Linear objective term.
A_{qp}	Linear inequality constraints.
b_{qp}	Linear inequality constraints.
$\hat{x}_k$	State incremental variable.
$\hat{y}_k$	Output incremental variable.
$\hat{u}_k$	Input incremental variable.

1 PREFACE

The aim of this project is to create a controller capable of leading an inverted-pendulum-like robot, based on the popular urban transport Segway. Nevertheless, its initial purpose was not only to program a controller for the robot, but also to implement it in a real prototype which would have included several improvements with respect to its previous versions. Sadly, public health situation in Spain has prevented us from doing so due to the impossibility of 3D-printing the necessary parts of the robot.

1.1 The cart-pole

An inverted pendulum-like robot is one which includes, mainly, two wheels and a body. The body stands over the wheels, which share an axle, and it is supposed to hold all the electronics, such as the microcontroller, the engines, the batteries, etc. inside.

Its functioning principle is based on the cart-pole system, which is shown below:

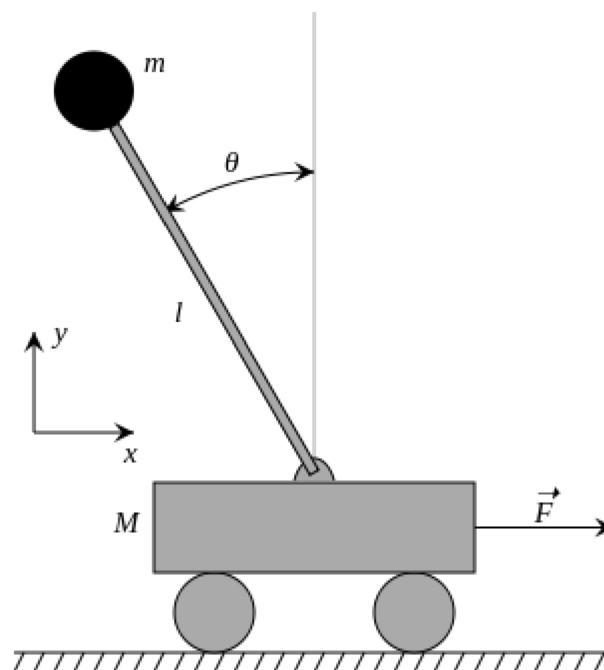


Figure 1-1. Cart-pole system.

The cart-pole is, as its name says, a pole (an inverted pendulum, where the mass is placed above (and a cart). The cart is allowed to move horizontally thanks to having a set of two wheels, which are themselves thrusted by a motor, whose exerted force is represented in the image as the force $\vec{F}$. Along with this, there is a pole fixed to the cart by a rotation joint, which allows it to freely oscillate around the joint.

This system is by itself highly unstable, and the two equilibrium points numerical analysis throws are the one

with the pole vertically aligned facing down and the other facing up:

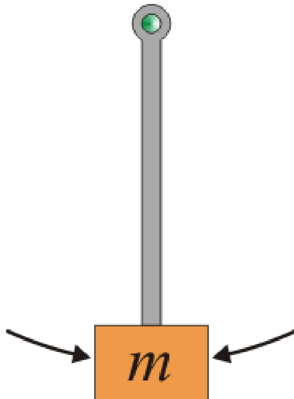


Figure 1-2. Pendulum's stable position.

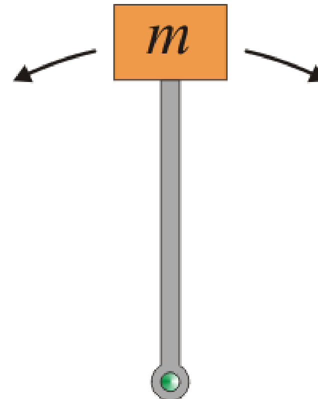


Figure 1-3. Pendulum's unstable position.

Being the first one not very interesting to analyse, the facing-up pendulum was the object of study of this project, i.e., it will be tried to drive an inverted pendulum-like robot to the unstable point

1.2 The Segway

A commercial robot based on the inverted pendulum is the Segway. It is a robot on which a person stands and is driven by the subject's inclination:

Figure 1-4. A Segway.

When a person stands on the Segway and leans forward, the robot will make the wheels roll to compensate that temporal imbalance and it will try to make the robot stand in a vertical position. Thus, if the controller keeps on leaning forward, the wheels will keep on rolling forward—and hence, the robot will go forward as well—until the person desires to stop being inclined.

The problem the Segway's controller must solve every time is analogue to balancing a vertically placed stick (or broom) with a finger:

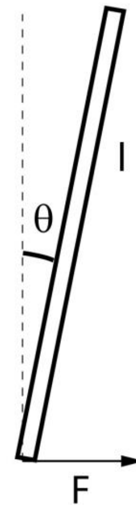


Figure 1-5. Segway's analogy.

It can be easily appreciated that Segway are robots based on the cart-pole, as most of the mass is distributed throughout the body, which is driven by the force created by the rolling wheels.

1.3 Our project

The case study of this work initially consisted of a self-made robot based on Segway’s structure being controlled by the new Jetson Nano, a minicomputer with very good specifications and community response throughout the net.

The aim was to build—by 3d printing—a robot from scratch and design the algorithm that would drive it to reach the balance point explained above.

For the algorithm, **Julia** language was chosen. It was elected due to being a language with very good timing specifications, i.e., a fast one, a very needed characteristic for driving the robot through the setpoints, as slow calculations would mean that the robot could reach instability in the calculation time.

Some pictures of the previous project are shown below to inspire the reader about what it was going to be performed throughout the project:

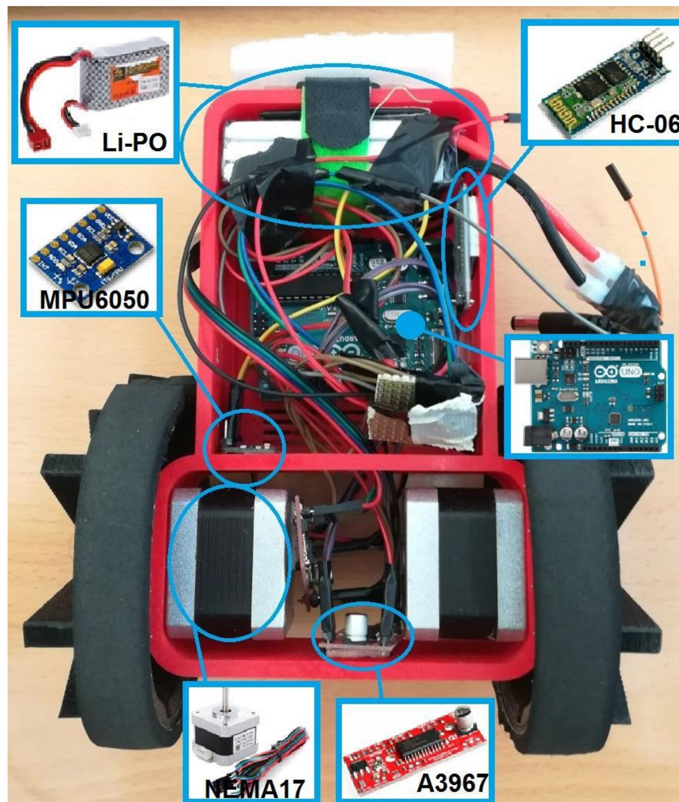


Figure 1-6. Physical implementation of the previous project.

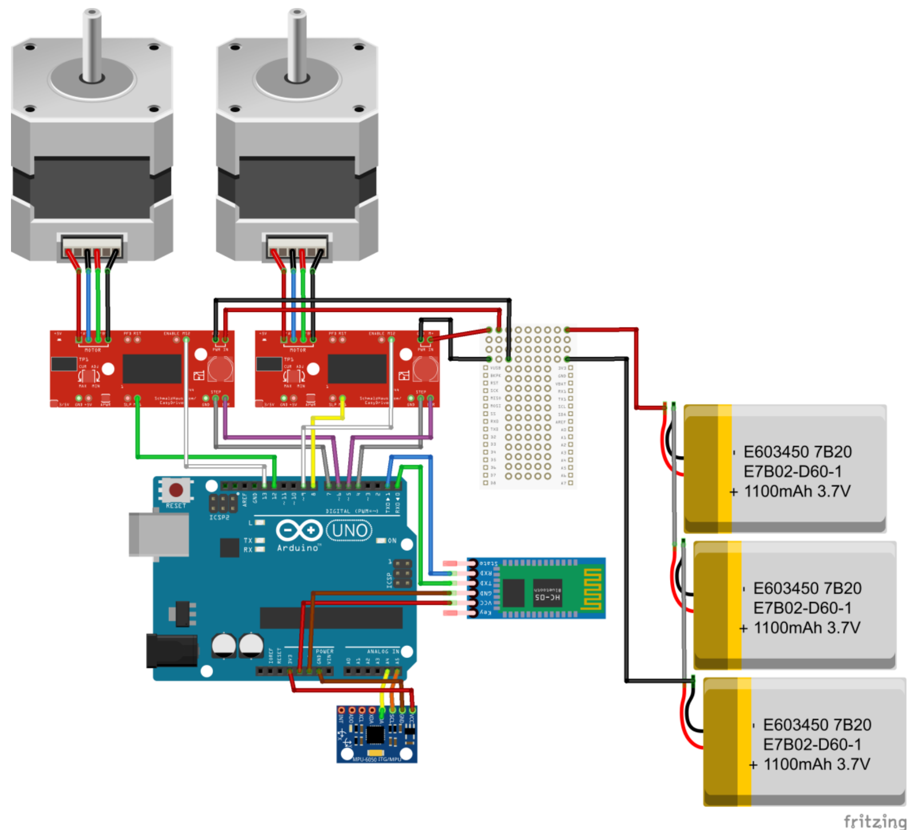


Figure 1-7. Electronics scheme of the previous project.

As it can be seen in the previous pictures, an Arduino UNO was the brain in charge of performing the calculations of control actions based in how inclined was the vehicle—measurements given by an IMU—and the setpoints, which could be stated in the programme.

Even so, it has been written in past because the improvements could not be implemented due to the restrictions imposed by the global health situation caused by COVID-19 and other external factors out of the control of the

author.

Instead of that, it was thought that using the Jetson Nano and Julia to create a non-previously developed programme would be challenging enough and it would also contribute to the current state of art of both Julia and the Jetson Nano, which are both recent technologies under development and with very scarce documentation to read about.

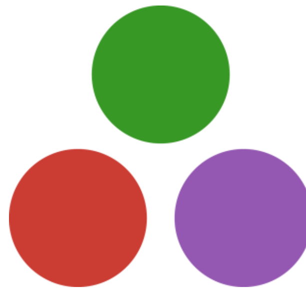
This document covers what Julia and the Jetson Nano are, telling about all their specifications, specific installation procedures and configuration issues. After that, some mathematical developments are shown to demonstrate where the equations come from, to subsequently—in the following chapter—demonstrate the equations the controller should have when programming. Then, the specific programmes used in MATLAB and Julia are shown and explained in detail and compared to eventually show the results, performed not only in the Jetson Nano, but also in the author's PC. Finally, the conclusions drawn from this work are summed up.

2 JULIA

(Did we mention it should be as fast as C?)

- julialang.org -

Julia is a modern, functional programming language, developed in 2009 by Jeff Bezanson, Stefan Karpinski, Viral B. Shah, and Alan Edelman, who had the idea of designing a language that was free, high-level, and fast. Its creators wanted to combine the readability and simplicity of Python with the speed of statically typed, compiled languages like C. The language was officially unveiled to the world in 2012 when the team launched a website with a blog post explaining the language's mission.



2.1 Introduction

As said, Julia is a relatively new language and is still under development. This means there are more bugs and fewer native packages than it would be expected from a more mature language.
Established languages like Python and Java also have much larger communities, making it easier to find tutorials, third-party packages, and answers to one's questions. On the other hand, Julia's speed, ease of use, and suitability for big-data applications (through its high-level support for parallelism and cloud computing) have helped it to grow quickly and it continues to attract new users. Nonetheless, Julia changes frequently due to being a relatively new programming language. In this project, **version 1.4.2** has been the chosen one.

A strong point about Julia is that is an interpreted language, just like MATLAB or Python—two languages to which Julia will be compared throughout this project—but when the time of compiling arrives, it behaves just as a compiled language—thanks to JIT—as C, for instance, gaining that comparable speed for which it is widely known. Even so, this celerity superior to interpreted languages shall be beheld after the first cycle.

Although Julia is designed as a technical language first, this does not mean that it cannot be used for other things. Just like Python, Julia can also be used for writing software in the widest variety of application domains. This is because Julia does not include language constructs designed to be used within a specific application domain. Julia lets you write UIs, statically compile your code, or even deploy it on a webserver. It also has powerful shell-like capabilities for managing other processes. It provides Lisp-like macros and other metaprogramming

facilities. The standard library also provides asynchronous I/O, process control, logging, profiling, and more.

However, this chapter is not intended as a guide to learn Julia, but a review of the aspects that lead us to choose it above other programming languages. These main features are the following, and some of them will be treated in detail subsequently:

- **Julia is compiled, not interpreted.** For faster runtime performance, Julia is just-in-time (JIT) compiled using the LLVM compiler framework. At its best, Julia can approach or match the speed of C.
- **Julia is interactive.** Julia includes a REPL (read-eval-print loop), or interactive command line, like what Python offers. Quick one-off scripts and commands can be punched right in.
- **Julia has a straightforward syntax.** Julia's syntax is similar to Python's—terse, but also expressive and powerful.
- **Julia combines the benefits of dynamic typing and static typing.** You can specify types for variables, like "unsigned 32-bit integer". Furthermore, it is one recommended good programming practice [1]. But you can also create hierarchies of types to allow general cases for handling variables of specific types—for instance, to write a function that accepts integers without specifying the length or signing of the integer. You can even do without typing entirely if it is not needed in a particular context.
- **Julia can call Python, C, and Fortran libraries.** Julia can interface directly with external libraries written in C and Fortran. It is also possible to interface with Python code by way of the PyCall library, and even share data between Python and Julia.
- **Julia supports metaprogramming.** Julia programs can generate other Julia programs, and even modify their own code, in a way that is reminiscent of languages like Lisp.
- **Julia has a full-featured debugger.** Julia 1.1 introduced a debugging suite, which executes code in a local REPL and allows you to step through the results, inspect variables, and add breakpoints in code. You can even perform fine-grained tasks like stepping through a function generated by code.

2.2 Features analysis

2.2.1 Technical computing

Designed with data science in mind, Julia excels at numerical computing with a syntax that is great for math, with support for many numeric data types, and providing parallelism out of the box, but more on that last bit later. Julia's multiple dispatch is a natural fit for defining number and array-like data types.

The Julia REPL (Read/Evaluate/Print/Loop) provides easy access to special characters, such as Greek alphabetic characters, subscripts, and special maths symbols. If the backslash is typed, a string can then be typed (usually the equivalent LATEX string) to insert the corresponding character. This is great as it allows developers to just derive some equation and directly type it in.

2.2.2 Compiling

Julia is a compiled language—that is one of the reasons that it performs faster than interpreted languages. However, unlike traditional compiled languages, Julia is not strictly statically typed. It uses JIT (Just In Time, i.e. every statement is run using compiled functions which are either compiled right before they are used, or cached compilations from before) compilation to infer the type of each individual variable in your code. The result is a dynamically typed language that can be run from the command line like Python, but that can achieve comparable speeds to compiled languages like C and Go.

2.2.3 Type checking

Other languages as Python are dynamically typed, meaning that a variable can be declared without specifying its type; the Python interpreter determines the type from the value provided (e.g. `m = 5` will be interpreted as an integer). Variables in Julia can be declared in this way as well; however, it is possible to specify types, or a range of possible types, for a variable. Specifying the expected types for a function helps the compiler optimize for better performance and can also prevent errors resulting from unexpected or incorrect input.

2.2.4 Speed

It is hard to talk about Julia without talking about speed. Julia prides itself on being very fast.

Unlike Python, Julia, which is interpreted, is a compiled language that is primarily written in its own base. Nevertheless, unlike other compiled languages like C, Julia is compiled at run-time, whereas traditional languages are compiled prior to execution. Julia, especially when written well, can be as fast and sometimes even faster than C. As mentioned before, Julia uses the Just In Time (JIT) compiler and compiles incredibly fast, though it compiles more like an interpreted language than a traditional low-level compiled language like C, or Fortran.

Julia is faster than other scripting languages, allowing you to have the rapid development of Python/MATLAB/R while producing code that is as fast as C/Fortran. Julia complies codes on the fly, reaching an incredible velocity. It combines the method of interpreting languages, such as Python, that transform the code into bytecode on the fly, with compiled languages (C/Fortran), that compile their code into machine code and then running the resulted executable. This approach allows Julia to produce similar speeds to C, while having a code like Python or MATLAB.

To demonstrate Julia's computational power, a demonstration section of code is shown, displaying a brief application whose intention is merely to sum an array of random numbers:

```
julia> x = rand(1000);

julia> function sum_global()
           s = 0.0
           for i in x
               s += i
           end
           return s
       end;

julia> @time sum_global()
  0.017705 seconds (15.28 k allocations: 694.484 KiB)
496.84883432553846

julia> @time sum_global()
  0.000140 seconds (3.49 k allocations: 70.313 KiB)

496.84883432553846
```

If slight changes are performed, such as passing `x` as an argument to the function, it no longer allocates memory (the allocation reported below is due to running the `@time` macro in global scope) and is significantly faster after the first call:

```
julia> x = rand(1000);

julia> function sum_arg(x)
           s = 0.0
           for i in x
               s += i
           end
           return s
       end;

julia> @time sum_arg(x)
  0.007701 seconds (821 allocations: 43.059 KiB)
496.84883432553846

julia> @time sum_arg(x)
  0.000006 seconds (5 allocations: 176 bytes)

496.84883432553846
```

The 5 allocations seen are from running the `@time` macro itself in global scope. If we instead run the timing in a function, we can see that indeed no allocations are performed:

```
julia> time_sum(x) = @time sum_arg(x);

julia> time_sum(x)
  0.000001 seconds

496.84883432553846
```

Julia's compiler does not have to know beforehand what type of variable it is being tried to use, but it cleverly plans whenever a function is called. When a function is called in Julia, the arguments are already known; Julia's compiler uses this knowledge to figure out the exact CPU instructions necessary for the arguments by peeking into the function. Once the exact instructions are mapped out, Julia can execute them very quickly—therefore, calling a function in Julia takes long the first time. During this period, Julia's compiler would be figuring out all the types of variables being used and compiles them all into fast and precise CPU instructions. This means that when the same function is called repeatedly, consequent calls run much, much faster.

2.2.5 Multiple dispatch

There exist two types of programming languages, static languages, where it is required to have a type computable before the execution of the program, and dynamic languages, where nothing is known about types until run time, when the actual values manipulated by the program are available.

Multiple dispatch refers to declaring different versions of the same function to better handle input of different types. For example, two different `reverse` functions might be written, one that accepts an array as an argument and one that accepts a string. The Julia interpreter will check the type of the argument whenever `reverse` is called and dispatch it to the version matching that type. A multiple dispatch environment ensures that, depending on the variables provided to the function, different results will be produced.

Type stability is the idea that there is only 1 possible type which can be output from a method. For example, the reasonable type to output from `*(::Float64,::Float64)` is a `Float64`. It does not matter what you give it, it will produce a `Float64`. This right here is multiple dispatch: the `*` operator calls a different method depending on the types that it sees. When it sees floats, it will generate floats.

Julia allows for unicode characters, and these can be used by tab completing Latex-like statements. Also, multiplication by a number is allowed without the `*` if followed by a variable. For example, the following is allowed Julia code:

```
α = 0.5
∇f(u) = α*u; ∇f(2)
sin(2π)
```

The default behaviour in Julia when types are omitted is to allow values to be of any type. Thus, one can write many useful Julia functions without ever explicitly using types. When additional expressiveness is needed, however, it is easy to gradually introduce explicit type annotations into previously "untyped" code. Adding annotations serves three primary purposes: to take advantage of Julia's powerful multiple-dispatch mechanism, to improve human readability, and to catch programmer errors.

2.2.6 Composability

Julia packages naturally work well together. This is thanks to the language's function composition which makes it easy to pass two or more functions as arguments. Julia has a dedicated function composition operator (∘) for achieving this.

For example, the sqrt() and + functions can be composed like this:

```
julia> (sqrt ∘ +)(3, 5)
(sqrt ∘ +)(3, 5)
```

which adds the numbers first, then finds the square root.

This example composes three functions.

```
julia> map(first ∘ reverse ∘ uppercase, split("you can compose functions like
this"))
6-element Array{Char,1}:
'U'
'N'
'E'
'S'
'E'
'S'
```

This was just a basic example, but Julia makes it easy for packages to communicate with each other. Matrices of unit quantities, or data table columns of currencies and colours, just work—and with good performance.

2.2.7 Parallelism

Math and scientific computing thrive when you can make use of the full resources available on a given machine, especially multiple cores. In fact, data has gotten so huge that it has become unpractical to run applications on a single computer. Nowadays, people are computing big data on multiple nodes in a cluster to decrease the execution time by running tasks in parallel, but unfortunately many languages were never designed for this.

For instance, both Python and Julia can run operations in parallel, but Julia was designed for parallelism from the ground-up and provides built-in primitives for parallel computing at every level: instruction level parallelism, multi-threading, and distributed computing. The Celeste.jl project achieved 1.5 PetaFLOP/s on the Cori supercomputer at NERSC using 650,000 cores [2].

It is possible to run code in parallel in Python with the purpose of taking advantage of all of the CPU cores on your system. However, Python's methods for parallelizing operations often require data to be serialized and deserialized between threads or nodes. This requires importing modules and involves some quirks that can make concurrency difficult to work with. Julia's parallelization is more refined. In contrast, it has top-level support for parallelism and a simple, intuitive syntax for declaring that a function should be run concurrently:

```
nheads = @parallel (+) for i = 1:100000000

  rand(Bool)

end
```

2.2.8 Package manager

Julia's Pkg package manager comes loaded with its own REPL from which you can build, add, remove, and instantiate packages. This is especially convenient because of Pkg's tie-in with Git.

Pkg is designed around "environments" instead of a single global set of packages like traditional package managers. In Julia, packages can either be local to an individual project, or shared and selected by name. Environments are maintained in a manifest file, containing the exact set of packages and versions which a particular application needs.

Thanks to such environments, each application maintains its own its own independent set of package versions—this greatly improves reproducibility, allowing developers to check out a project on a new system, simply materialize the environment described by its manifest file, and immediately be up and running with a known-good set of dependencies.

Due to its built-in package manager, Julia already has over 4000 registered packages (and the number keeps growing [3]). It also provides the possibility to resort to C, Fortran, and Python packages, making it easy to run existing code.

Even so, Julia is a relatively new language and is still under development. This means there are more bugs and fewer native packages than it would be expected from a more mature language. Established languages like Python and Java also have much larger communities, making it easier to find tutorials, third-party packages, and answers to one's questions. It is possible to run Python libraries in Julia (through the PyCall package), and C/Fortran libraries can be called and run directly from Julia code. This allows Julia users to access a wider range of external libraries than it would otherwise have, but Python still has the advantage of a large set of native packages and a vibrant community.

On the other hand, Julia's speed, ease of use, and suitability for big-data applications (through its high-level support for parallelism and cloud computing) have helped it to grow quickly and it continues to attract new users. Julia developers are already working at companies including Google, NASA, and Intel, and major projects like RStudio have announced plans to add support for Julia.

3 NVIDIA JETSON NANO

The new Jetson Nano is the ultimate starter AI computer that allows hands-on learning and experimentation at an incredibly affordable price.

- Deepu Talla, vice president and general manager of Edge Computing at NVIDIA -

The NVIDIA Jetson platform introduced six years ago revolutionized embedded computing by delivering the power of artificial intelligence to edge computing devices. NVIDIA Jetson today is widely used in diverse fields such as robotics, retail, industrial, agriculture, and AIoT—also called intelligent IoT, artificial intelligence meeting internet of things—.

Figure 3-1. NVIDIA Jetson Nano Developer Kit.

3.1 Introduction

The science of AI computing is changing fast, and new neural network architectures that deliver better accuracy and performance are being invented by researchers. AI practitioners today use a wide variety of AI models and frameworks in their projects, so the ideal platform for learning and creating AI projects is one flexible enough to run a diverse set of AI models and also powerful enough to deliver the performance required to create meaningful interactive AI experiences.

Hardware, low power consumption, high accuracy and performance are crucial factors for deep learning applications. High level graphics processing units (GPU) are commonly used in high performance deep learning applications. However, it is a lot in terms of cost and power consumption to build a high-performance platform.

NVIDIA has a specific product line with great computing capabilities for AI and Machine Learning such as the

Jetson **AGX XAVIER** or **Jetson TX2**, but its price skyrockets for those who just want to do a specific project or test the technology.

In March 2019, NVIDIA launched the original Jetson Nano Developer Kit and enabled developers, students, and enthusiasts alike to learn, explore, and build AI applications for edge devices. The Jetson Nano offers a much cheaper solution. In addition, it is itself the size of a SO-DIMM RAM module of only 69.6 x 45 millimetres and that makes it especially interesting to easily insert it into a final product by simply including a socket type connector of the same type together with its circuitry necessary for its operation. The adoption by the enthusiast community was quite rapid. The Jetson Nano Developer Kit not only supports all popular AI frameworks and networks, but also delivers a powerful AI performance.

NVIDIA® Jetson Nano™ Developer Kit is a small, powerful computer that lets running multiple neural networks in parallel for applications like image classification, object detection, segmentation, and speech processing. All in an easy-to-use platform that runs in as little as 5 watts. Useful for deploying computer vision and deep learning, Jetson Nano runs Linux and provides 472 GFLOPS of FP16 compute performance with 5-10W of power consumption. Nonetheless, even being created with those purposes in mind, it is also feasible for developing a myriad of different projects, hence being feasible for our project as well.

The compute performance, compact footprint, and flexibility of Jetson Nano brings endless possibilities to developers for creating AI-powered devices and embedded systems, which could be beneficial, if not crucial, in future enhancements of the vehicle, just as image recognition for self-guidance.

The Jetson Nano can be purchased either as a development kit or a single module. In either way, the specifications of it are listed in the table below:

Table 3-1. Jetson Nano's technical specifications.

GPU	128-core Maxwell
CPU	Quad-core ARM A57 @ 1.43 GHz
Memory	4 GB 64-bit LPDDR4 25.6 GB/s
Storage	microSD (not included)
Video Encode	4K @ 30 | 4x 1080p @ 30 | 9x 720p @ 30 (H.264/H.265)
Video Decode	4K @ 60 | 2x 4K @ 30 | 8x 1080p @ 30 | 18x 720p @ 30 (H.264/H.265)
Camera	2x MIPI CSI-2 DPHY lanes
Connectivity	Gigabit Ethernet, M.2 Key E
Display	HDMI and display port
USB	4x USB 3.0, USB 2.0 Micro-B
Others	GPIO, I2C, I2S, SPI, UART
Mechanical	69 mm x 45 mm, 260-pin edge connector

The Jetson Nano Developer Kit is an easy way to get started using Jetson Nano, including the module, carrier board, and software. What is included:

- 80x100mm Reference Carrier Board
- Jetson Nano Module with passive heatsink
- Pop-Up Stand
- Getting Started Guide

(the complete devkit with module and heatsink weighs 138 grams)

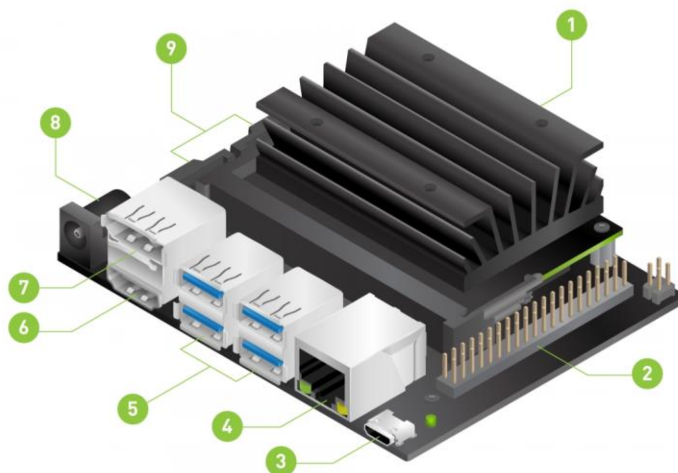


Figure 3-2. NVIDIA Jetson Nano module with passive heatsink.

1. microSD card slot for main storage
2. 40-pin expansion header
3. Micro-USB port for 5V power input, or for Device Mode
4. Gigabit Ethernet port
5. USB 3.0 ports (x4)
6. HDMI output port
7. DisplayPort connector
8. DC Barrel jack for 5V power input
9. MIPI CSI-2 camera connectors

NVIDIA **Jetson Nano** is an embedded system-on-module (SoM) and developer kit from the NVIDIA Jetson family, including an integrated 128-core Maxwell GPU, quad-core ARM A57 64-bit CPU, 4GB LPDDR4 memory, along with support for MIPI CSI-2 and PCIe Gen2 high-speed I/O. There is also the **Jetson Nano 2GB Developer Kit** with 2GB memory and the **same processing specs**.

Here, a table comparing the main aspects of Nano and Nano 2GB developer kits is show, not only with the purpose of reporting their main characteristics, but also to give the reader a quick reference to compare both of them, in the case the 2GB option was preferred:

Table 3-2. Comparison between Nano and Nano 2GB development kits.

	Jetson Nano Developer Kit	**Jetson Nano 2GB Developer Kit**
USB	(4x) USB 3.0 Type-A, USB 2.0 Micro-B	(1x) USB 3.0 Type-A, (2x) USB 2.0 Type-A, USB 2.0 Micro-B
Camera	(2x) MIPI CSI-2 x2 (15-position Camera Flex Connector)	(1x) MIPI CSI-2 x2 (15-position Camera Flex Connector)
Display	HDMI 2.0, DisplayPort	HDMI 2.0
Wireless	M.2 Key-E (PCIe x1)	802.11ac USB dongle*
Ethernet	Gigabit Ethernet (RJ45)	
Storage	MicroSD card slot	

Other	40-pin Header - (3x) I2C, (2x) SPI, UART, I2S, GPIOs	
Power	Micro-USB (5V⎓2.5A) or DC barrel jack (5V⎓4A)	USB-C (5V⎓3A)

3.2 Getting started

What will be needed:

- Power Supply
  - 5V⎓2.5A Micro-USB adapter.
  - 5V⎓4A DC barrel jack adapter, 5.5mm OD x 2.1mm ID x 9.5mm length, centre-positive. The one used in this project can be found in Amazon [4].
  - See the Jetson Nano Supported Component List and Power Supplies documents for more information. These documents can be found in the Jetson Download Center.
- MicroSD card (32GB UHS-1 recommended minimum).
- USB keyboard and mouse.
- Computer display (HDMI or DP).
- HDMI cable.
- USB Wi-Fi plug.

Regarding the power supply, it was initially chosen to use a micro-USB option. However, every time the Nano was turned on, a warning appeared stating that the computer could not run in maximum power mode. In addition, it also warned that the system was being throttled due to insufficient input voltage. Nevertheless, the micro-USB to USB cable and the AC/DC converter (both taken from a mobile phone) met the specifications suggested by NVIDIA in its website.

After this inconvenience, it was thought that a barrel jack power supply of 5 volts and 4 amperes would do the job. However, power supplies suggested by NVIDIA's official documentation were either expensive, American-plugged or, in the case of European versions, sold out. Thankfully, a 5V 4A barrel jack power supply found in Amazon worked, despite not having even one opinion. This power supply never gave a problem and the warnings about thermal throttling never appeared again.

To configure the Jetson Nano Developer Kit, the following steps will be followed:

- Follow the getting started guide [5] to setup the devkit and format the MicroSD card.
- Plug in an HDMI display into Jetson, attach a USB keyboard & mouse, and apply power to boot it up.
- Visit the Embedded Developer Zone [6] and Jetson Nano Developer Forum [7] to access the latest documentation and downloads.

To follow the guide above, it has been considered useful to provide schematics of the Nano:

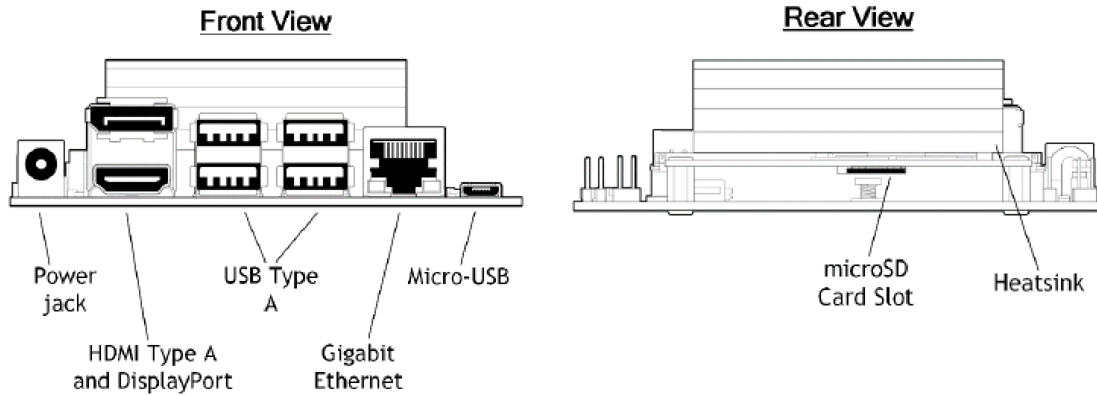


Figure 3-3. Developer kit module and carrier boards: front and rear views.

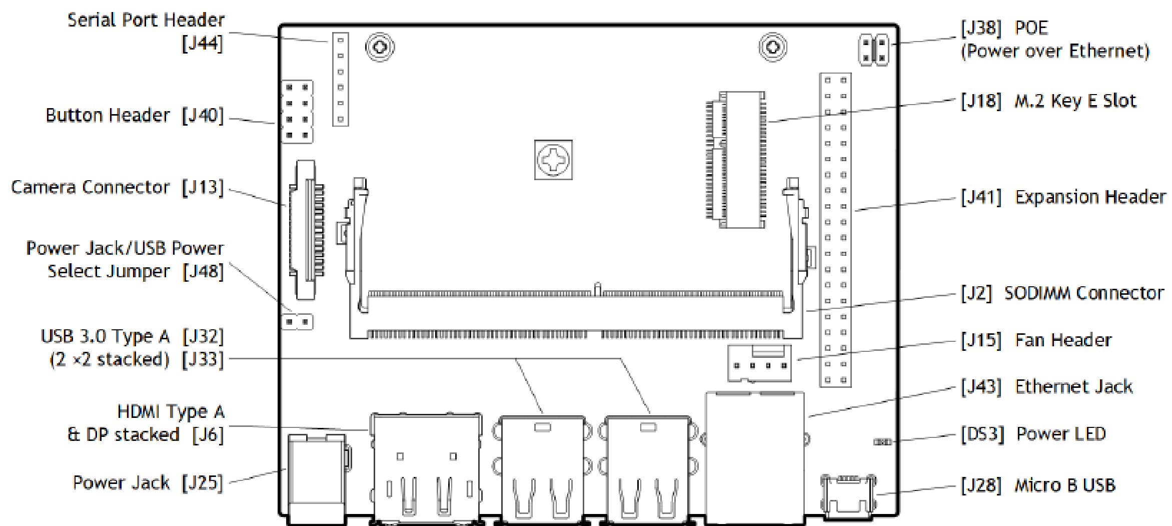


Figure 3-4. Developer kit carrier boards: rev A02 top view.

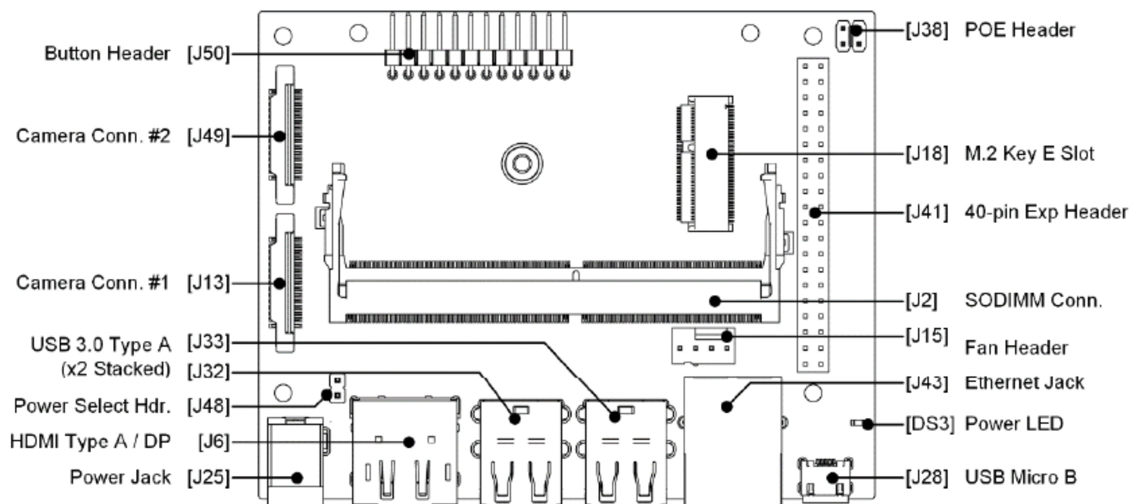


Figure 3-5. Developer kit module and carrier board: rev B01 top view.

The guide listed above is pretty straightforward and followed its instructions, the setup will be ready quite fast. It has been considered not necessary to insert the whole guide in the text of this project but the links to it, as it could change through time and leave the one written here useless.

All the latest documentation can be found in the Jetson Download Center [8]. There, there are included the development kit user guide, the SD card image, etc. From the latest versions to the oldest ones, provided a new development kit is released.

3.3 Configuration of the Jetson Nano

This project began with the idea of attaching the Nano to the Segway's body and to make it stabilise. Sadly, the global pandemic prevented us from doing so due to mobility restrictions and the impossibility of acquiring the 3d printing materials and going to the laboratories to make the printings.

Nonetheless, this was not known in the beginning of the project and some progresses were made right before the pandemic restrictions exploded. In this section, those achievements are listed, as it is supposed that they will be needed for further enhancements of the robot.

Firstly, **Visual Studio Code** will be installed to grant a quick and easy access to Julia, as it has been done in this project's development due to being more appealing to program from there instead of from Julia's interactive application.

Type the following commands in a terminal:

```
$ sudo apt-get update

$ sudo apt-get install curl

$ curl -L https://github.com/toolboc/vscode/releases/download/1.32.3/code-oss_1.32.3-arm64.deb -o code-oss_1.32.3-arm64.deb

$ sudo dpkg -I code-oss_1.32.3-arm64.deb
```

After these commands, Visual Studio Code will be installed in the Jetson Nano.

To **install Julia in the Nano**, the following steps will be followed:

1. Go to Julia Downloads page [9].
2. Download the Generic Linux Binaries for ARM aarch64.
3. Follow platform instructions for Linux [10].

From that, the Julia extension can be installed as well. Be sure to follow Visual Studio Code's instructions for installing Julia [11].

After having successfully turned the Nano on for the first time, it was tried to connect the IMU (inertial measurement unit) MPU6050 with the Jetson and make it read its values. The decision to use the MPU6050 was based on the fact that it was the device used in the previous versions of the project.

The MPU6050 is a 6-DoF Accelerometer and Gyroscope. It features three 16-bit analogue-to-digital converters (ADCs) for digitizing the gyroscope outputs and three 16-bit ADCs for digitizing the accelerometer outputs. For precision tracking of both fast and slow motions, the parts feature a user-programmable gyroscope full-scale range of ±250, ±500, ±1000, and ±2000°/sec (dps) and a user-programmable accelerometer full-scale range of ±2*g*, ±4*g*, ±8*g*, and ±16*g*. More information can be found inside its datasheet.

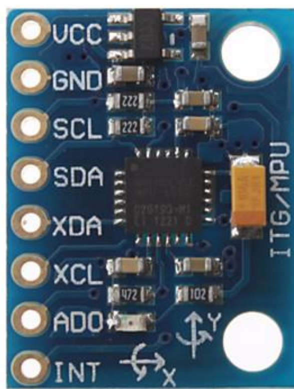


Figure 3-6. MPU-6050 detailed view.

It's easy to use the MPU6050 sensor with CircuitPython and the Adafruit CircuitPython MPU6050 library. This library allows to easily write Python code that reads the acceleration and adjust the measurement settings. This sensor can be used with any CircuitPython microcontroller board or with a Linux single board computer that has GPIO and Python thanks to Adafruit_Blinka, the CircuitPython-for-Python compatibility library. But to utilize it, Python must be installed on the Nano beforehand. To do this, open a terminal and execute the following command:

```
$ sudo apt-get install python3-dev
```

Next, go to Visual Studio Code and install the Python extension. It is utterly straightforward.

Afterwards, open a Python session in Visual Studio. From the terminal, type the following command to install the library:

```
$ sudo pip3 install adafruit-circuitpython-mpu6050
```

Additional information about Adafruit _CircuitPython_MPU6050 can be found in the Adafruit's website [12].

And that is all, at least for now, because anyone could run a programme based on this library, but first, it would be necessary to connect the MPU to the Jetson Nano. To do this, knowing the J41 header of the Jetson Nano will be essential. Information about the header can be found in the user guide, however, for the sake of ease, in [13] it has been developed a schematic of the pin arrangement, which is shown below:

Jetson Nano J41 Header					
Sysfs GPIO	Name	Pin	Pin	Name	Sysfs GPIO
	3.3 VDC *Power*	1	2	**5.0 VDC** *Power*	
	I2C_2_SDA *I2C Bus 1*	3	4	**5.0 VDC** *Power*	
	I2C_2_SCL *I2C Bus 1*	5	6	**GND**	
gpio216	**AUDIO_MCLK**	7	8	**UART_2_TX** */dev/ttyTHS1*	
	GND	9	10	**UART_2_RX** */dev/ttyTHS1*	
gpio50	**UART_2_RTS**	11	12	**I2S_4_SCLK**	gpio79
gpio14	**SPI_2_SCK**	13	14	**GND**	
gpio194	**LCD_TE**	15	16	**SPI_2_CS1**	gpio232
	3.3 VDC *Power*	17	18	**SPI_2_CS0**	gpio15
gpio16	**SPI_1_MOSI**	19	20	**GND**	
gpio17	**SPI_1_MISO**	21	22	**SPI_2_MISO**	gpio13
gpio18	**SPI_1_SCK**	23	24	**SPI_1_CS0**	gpio19
	GND	25	26	**SPI_1_CS1**	gpio20
	I2C_1_SDA *I2C Bus 0*	27	28	**I2C_1_SCL** *I2C Bus 0*	
gpio149	**CAM_AF_EN**	29	30	**GND**	
gpio200	**GPIO_PZ0**	31	32	**LCD_BL_PWM**	gpio168
gpio38	**GPIO_PE6**	33	34	**GND**	
gpio76	**I2S_4_LRCK**	35	36	**UART_2_CTS**	gpio51
gpio12	**SPI_2_MOSI**	37	38	**I2S_4_SDIN**	gpio77
	GND	39	40	**I2S_4_SDOUT**	gpio78

Figure 3-7. J41 header pin layout.

The Jetson Nano has two UARTS, one on the J44 Serial Port header, and one on the J41 Expansion Header.

There are also provisions for Serial Peripheral Interface (**SPI**) ports. In the default Nano Image configuration, the Jetson Nano does not have SPI port access. However, the device tree can be reconfigured for accessing SPI through the J41 Expansion Header. In the Jetson Nano J41 Pinout, NVIDIA recommends placement for two SPI ports.

You may also hear the J41 Expansion Header referred to as the GPIO Header. The two terms are interchangeable.

The remaining serial ports are Inter-integrated Circuit (**I2C**). The Inter-integrated Circuit protocol is intended for short distance communication within a single device. One of the reasons that it is popular with the maker crowd is that it only requires two wires (the "data" and the "clock") along with power and ground to get functioning.

Nano and the IMU are connected by I2C, so these pins—extracted from the J41 layout—will be useful to interconnect them:

- I2C Bus 1 SDA is on Pin 3.
- I2C Bus 1 SCL is on Pin 5.
- I2C Bus 0 SDA is on Pin 27.
- I2C Bus 0 SCL is on Pin 28.

Before wiring the Jetson, make sure that the power is disconnected. When the power is plugged in, the power and ground rails on the headers are always live, even if the processor itself is off.

For our case, Bus 1 will be wired:

- J41 Pin 3 (SDA) → MPU6050 SDA.
- J41 Pin 5 (SCL) → MPU6050 SCL.
- J41 Pin 1 (3.3V) → MPU6050 VCC.
- J41 Pin 6 (GND) → MPU6050 GND.

After wiring the board, plug the Jetson in. Once the Nano is up and running, open a terminal and execute:

```
$ i2cdetect -y -r 1
```

The default address of the MPU6050 is 0x68. It should be seen an entry of '68' in the addresses listed. If the entry is not displayed, then the wiring is probably incorrect. When the address does not show up, then it will not be able to use the device.

Subsequently, it will be possible to read values from the IMU in Python. To demonstrate the usage of the sensor, it will be initialized and the acceleration, rotation, and temperature measurements from the board's Python REPL will be read.

Run the following code to import the necessary modules and initialize the I2C connection with the sensor:

```
$ import time
$ import board
$ import busio
$ import adafruit_mpu6050

$ i2c = busio.I2C(board.SCL, board.SDA)
$ mpu = adafruit_mpu6050.MPU6050(i2c)
```

Now everything is ready to read values from the sensor using these properties:

- acceleration - The acceleration forces in the X, Y, and Z axes in m/s 2
- gyro - The rotation measurement on the X, Y, and Z axes in degrees/sec
- temperature - The temperature of the sensor in degrees C

For example, to print out the acceleration, gyro, and temperature values every second:

```
$ while True:
$     print("Acceleration: X:%.2f, Y: %.2f, Z: %.2f m/s^2"%(mpu.acceleration))
```

```
$     print("Gyro X:%.2f, Y: %.2f, Z: %.2f degrees/s"%(mpu.gyro))
$     print("Temperature: %.2f C"%mpu.temperature)
$     print("")
$     time.sleep(1)
```

Now, the values stated above should be displayed through the monitor.

4 THE SYSTEM

To successfully develop a working QP controller, a model of the inverted pendulum must be known and developed. In this chapter, the model of the system will be deduced based on the Lagrange equations.

4.1 Introduction

Our system is based on the inverted pendulum—or cart-pole—which is a system composed of a cart and an inverted pendulum placed above it. An illustrative picture of it is shown here:

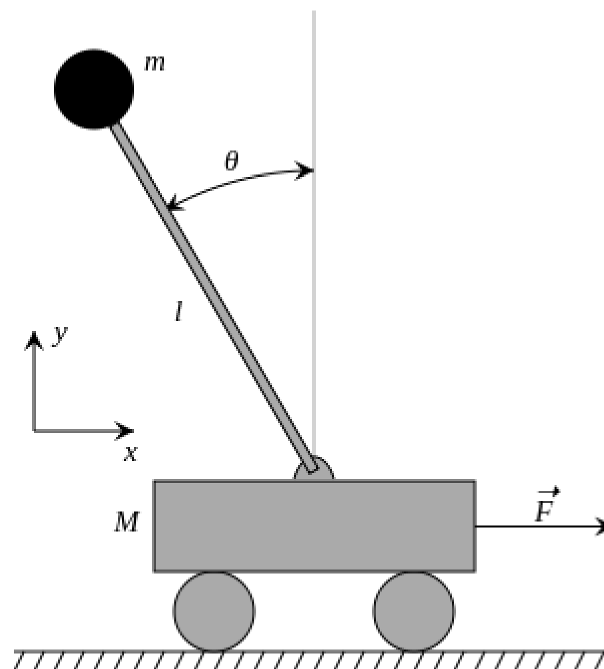


Figure 4-1. Inverted pendulum.

The whole system will move in the vertical plane, i.e. it is a two-dimensional problem, whose objective will be to drive the cart-pole to a reference equilibrium point, which is traditionally chosen as the one with zero horizontal speed and zero angle and angular speed.

Some other solutions can lead to different equilibrium points such as an analogue to the one above but with horizontal speed; it could be also possible to take the system to a trajectory where the pendulum is balancing constantly without falling, which would need the cart to follow an horizontally periodic trajectory, moving back and forth from left to right, and by choosing its top speed it would be also feasible to control how much the pole's angle varies.

Nevertheless, the self-balancing robot has a few differences concerning the inverted pendulum. First, there is no cart, but two wheels attached to the inverted pendulum sharing an axle; second, it is not possible to assume that

the mass is all concentrated in one point, but instead distributed throughout the whole body of the robot. In this picture, a schematic of the problem is shown:

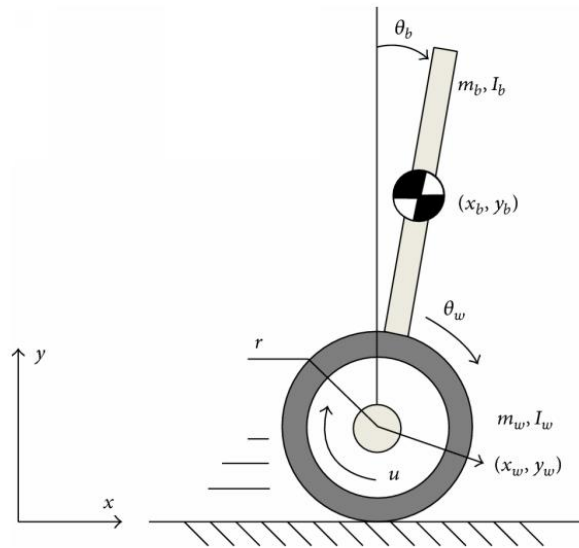


Figure 4-2. Two-wheeled inverted pendulum example.

Even so, the goal of its control is the same as the inverted pendulum: to reach an equilibrium point around vertical alignment of the body, being wheels speed a chosen parameter. Nevertheless, it must be taken into consideration that not every wheel speed can be chosen, as the dynamics of the system and the constraints imposed may result in an infeasible problem.

4.2 Equations of the system

This section is meant to clarify where do the system equations come from. The variables to be used in the following procedures are the ones that appear in the diagram below:

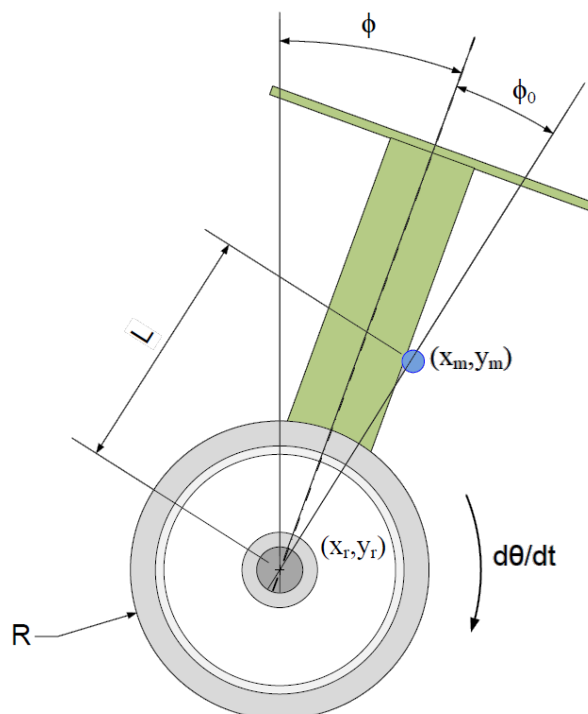


Figure 4-3. Schematic of the robot.

The notation of these variables that can be beheld in the scheme and their values are listed in the following table,

alongside a brief description of their meaning (under the name of 'parameter'):

Parameter	**Notation**	**Value**
Wheel mass	m_r	$0.044\ kg$
Wheel radium	R	$0.053\ m$
Mass of the vehicle without wheels	M	$0.745\ kg$
Distance from the wheel axis and the centre of gravity	L	$0.0805\ m$
Gravity	g	$9.81\frac{m}{s^2}$
Tilt with respect to the vertical	ϕ	
Centre of gravity deviation tilt	ϕ_0	n/a
Corrected tilt	ϕ_c	

To clarify, as it does not appear in the figure, ϕ_c is the sum of the tilt plus the deviation:

$$\phi_c = \phi + \phi_0$$

To deduce the relations that rule the system, the Lagrange equations, which are now recalled, will be used:

$$\frac{d}{dt}\left(\frac{\partial L}{\partial \dot{q}_k}\right) - \frac{\partial L}{\partial q_k} = Q_k^{nc} \qquad \text{Eq. 4-1}$$

Where L is the Lagrangian, q_k the generalised coordinates and Q_k^{nc} the non-conservative generalised forces. Having that in mind, the process to obtain the model equations is as follows:

Table 4-1. Mathematical development to obtain system's equations.

$L = T - V = T_{tras} + T_{rot} - V$	(1) Lagrangian
$T_{tras} = 2\frac{1}{2}m_r(\dot{x}_r^2 + \dot{y}_r^2) + \frac{1}{2}M(\dot{x}_m^2 + \dot{y}_m^2)$	(2) Translational kinetic energy
$x_r = R\theta$ $\dot{x}_r = R\dot{\theta}$	(3) Position and horizontal speed of the wheels
$y_r = R$ $\dot{y}_r = 0$	(4) Position and vertical speed of the wheels
$x_m = R\theta + Lsin(\phi + \phi_0)$ $\dot{x}_m = R\dot{\theta} + L\dot{\phi}\cos(\phi + \phi_0)$	(5) Position and horizontal speed of the vehicle's centre of gravity
$y_m = R + Lcos(\phi + \phi_0)$ $\dot{y}_m = -L\dot{\phi}\sin(\phi + \phi_0)$	(6) Position and vertical speed of the vehicle's centre of gravity
$T_{tras} = \left(m_r + \frac{1}{2}M\right)R^2\dot{\theta}^2 + \frac{1}{2}ML^2\dot{\phi}^2 + MRL\dot{\theta}\dot{\phi}\cos(\phi + \phi_0)$	(7) Translational kinetic energy

$$T_{rot} = \frac{1}{2} m_r R^2 \dot{\theta}^2 + \frac{1}{2} M L^2 \dot{\phi}^2$$

(8) Rotational kinetic energy (with respect to an axis coincident with the wheels one)

$$V = MgLcos(\phi + \phi_0)$$

(9) Potential energy (with respect to a height equal to the centre of the wheels)

$$L = \left(\frac{3}{2} m_r + \frac{1}{2} M\right) R^2 \dot{\theta}^2 + ML^2 \dot{\phi}^2 + MRL\dot{\theta}\dot{\phi}\cos(\phi + \phi_0) - MgLcos(\phi + \phi_0)$$

(10) Lagrangian with substitutions

$$\frac{\partial L}{\partial \theta} = 0$$

(11) Derivative of the lagrangian with respect to the wheels angle

$$\frac{\partial L}{\partial \dot{\theta}} = 2\dot{\theta}\left(\frac{3}{2} m_r + \frac{1}{2} M\right) R^2 + MRL\dot{\phi}\cos(\phi + \phi_0)$$

(12) Derivative of the lagrangian with respect to the wheels speed

$$\frac{d}{dt}\frac{\partial L}{\partial \dot{\theta}} = \ddot{\theta}(3m_r + M)R^2 + MRL\ddot{\phi}\cos(\phi + \phi_0) - MRL\dot{\phi}^2 \sin(\phi + \phi_0)$$

(13) Derivative with respect to time of (12)

$$\frac{\partial L}{\partial \phi_c} = -MRL\dot{\phi}\dot{\theta}\sin(\phi + \phi_0) + MgLsin(\phi + \phi_0)$$

(14) Derivative of the lagrangian with respect to the tilt

$$\frac{\partial L}{\partial \dot{\phi}_c} == 2ML^2\phi_{dot} + \dot{\theta}MRLcos(\phi + \phi_0)$$

(15) Derivative of the lagrangian with respect to the tilt derivative

$$\frac{d}{dt}\frac{\partial L}{\partial \dot{\phi}_c} = 2ML^2\ddot{\phi} + \ddot{\theta}MRLcos(\phi + \phi_0) - \dot{\theta}\dot{\phi}MRLsin(\phi + \phi_0)$$

(16) Derivative with respect to time of (15)

$$\frac{d}{dt}\frac{\partial L}{\partial \dot{\theta}} - \frac{\partial L}{\partial \theta} = \tau$$

$$\frac{\frac{d}{dt}\partial L}{\partial \dot{\phi}_c} - \frac{\partial L}{\partial \phi_c} = -\tau$$

(17) Lagrange equations particularized in the general coordinates based on the engine's torque.

$$\frac{d}{dt}\frac{\partial L}{\partial \dot{\theta}} - \frac{\partial L}{\partial \theta} + \frac{d}{dt}\frac{\partial L}{\partial \dot{\phi}_c} - -\frac{\partial L}{\partial \phi_c} = 0$$

(18) Result of summing equations from (17)

From the substitution of equations (11), (13), (14) and (16) into (18), the **system model** is obtained:

$$(2a + c \cdot cos(\phi + \phi_0))\ddot{\theta} + (c \cdot cos(\phi + \phi_0) + 2b)\ddot{\phi} - c\dot{\phi}^2 sin(\phi + \phi_0) - dsin(\phi + \phi_0) = 0 \quad \text{Eq. 4-2}$$

Being the constants:

$$a = \left(\frac{3}{2} m_r + \frac{1}{2} M\right) R^2$$
$$b = M \cdot L^2$$
$$c = MRL \quad \text{Eq. 4-3}$$

$$d = MgL$$

Namely, the system is defined by a second order nonlinear differential equation, which will be later transformed into a system of first order nonlinear differential equations to simulate the model behaviour.

5 CONTROLLER DESIGN

This chapter was written to show the reader how the controller is deducted and designed based on the system equations. It includes basic concepts of state-space models and a detailed deduction of how to transform the system equations in others more suitable for our case study.

5.1 Basic concepts

The most general state-space representation of a time-invariant linear system with is written in the following form:

$$\dot{x}(t) = Ax(t) + Bu(t)$$
$$y(t) = Cx(t) + Du(t) \quad \text{Eq. 5-1}$$

Despite this, the Jetson Nano is a discrete works in discrete time, so these equations will have to be discretized to make them work in the device.

The discretization will be done based on the Euler method, and the system discrete time-invariant variant will transform into:

$$x_{k+1} = Ax_k + Bu_k$$
$$y_k = Cx_k + Du_k \quad \text{Eq. 5-2}$$

Nonetheless, in our case, the matrix D will be zero as there is no dependence from the input in the case of the exit. This way, the model discrete time-invariant equations will eventually be:

$$x_{k+1} = Ax_k + Bu_k$$
$$y_k = Cx_k \quad \text{Eq. 5-3}$$

In addition, this project focuses on reaching the setpoint as quick as possible and with no steady state error. In other words, operating a dynamic system at minimum cost. A performance index of how near the system is from the setpoint must be defined, i.e. a cost function.

Amongst the myriad of possible performance indices to choose from, it has been decided to draw on the quadratic cost function commonly used in QR controllers:

$$V_N(x_F, u_F) = \sum_{i=0}^{N-1} \left[x_{k+i}^t Q x_{k+i} + u_{k+i}^t R u_{k+i}\right] + x_N^t P x_N \quad \text{Eq. 5-4}$$

Being Q, R and P matrices chosen by the designer of the controller.

5.2 Mathematical development

5.2.1 Model transformation

As stated before, the intention of this project was to reduce computing time from previous versions of it, trying to make a finer control. Due to that, there was the necessity of developing two versions of the controller: one in Julia, the new version; the other, in another programming language. MATLAB was chosen to be the face-off opponent to Julia, and to write the controller version in this language, the MATLAB function 'quadprog' was utilized.

'quadprog' is a solver for quadratic objective functions with linear constraints. It finds a minimum for a problem specified by:

$$\min_{x} \frac{1}{2} x^t H x + f^t x$$
$$s.t. \begin{cases} Ax \leq b, \\ A_{eq}x = b_{eq}, \\ lb \leq x \leq ub. \end{cases} \qquad \text{Eq. 5-5}$$

Being x the states and the rest of the matrices and vectors are deducted from the model equations.

The intent in this section is to transform the given cost function and model equations into others in the form of a quadratic programming problem. The mathematical development is shown along these lines.

Analysing the state equation:

$$\begin{aligned} x_{k+1} &= Ax_k + Bu_k \\ x_{k+2} &= Ax_{k+1} + Bu_{k+1} = A^2x_k + ABu_k + Bu_{k+1} \\ x_{k+3} &= Ax_{k+2} + Bu_{k+2} = A^3x_k + A^2Bu_k + ABu_{k+1} + Bu_{k+2} \\ x_{k+4} &= A^4x_k + \cdots \end{aligned} \qquad \text{Eq. 5-6}$$

And the output equation:

$$\begin{aligned} y_k &= Cx_k \\ y_{k+1} &= Cx_{k+1} = C(Ax_k + Bu_k) = CAx_k + CBu_k \\ y_{k+2} &= CA^2x_k + \cdots \end{aligned} \qquad \text{Eq. 5-7}$$

Followed this development, it would be easily recognised that the vector of future (or predicted) states x_F—made up of each state from the current one till the one at the end of the horizon N—can be expressed as:

$$x_F = \begin{bmatrix} x_k \\ x_{k+1} \\ \vdots \\ x_{k+N} \end{bmatrix} = \underbrace{\begin{bmatrix} I_{n_x} \\ A \\ A^2 \\ \vdots \\ A^N \end{bmatrix}}_{G_x} x_k + \underbrace{\begin{bmatrix} 0 & \cdots & \cdots & \cdots & \cdots & 0 \\ B & \cdots & \cdots & \cdots & \cdots & 0 \\ AB & B & 0 & \cdots & \cdots & \vdots \\ A^2B & AB & B & 0 & \cdots & \vdots \\ \vdots & \vdots & \vdots & \vdots & \ddots & 0 \\ A^{N-1}B & \cdots & \cdots & \cdots & \cdots & B \end{bmatrix}}_{G_u} \underbrace{\begin{bmatrix} u_k \\ \vdots \\ u_{k+N-1} \end{bmatrix}}_{u_F} \qquad \text{Eq. 5-8}$$

Note that the vector of predicted control actions u_F contains only till the $k + N - 1^{th}$ sample, as the effect of a control action is observed in the following sampling time. Therefore, there is no $(k + N)^{th}$ predicted control action.

Eventually, the system equations can be expressed as:

$$x_F = G_x x_k + G_u u_F \qquad \text{Eq. 5-9}$$

For its side, the cost function can also be expressed as function of x_F and u_F, as shown:

$$V_N(x_F,u_F) = \underbrace{[x_k \quad x_{k+N}]}_{x_F^t} \underbrace{\begin{bmatrix} Q & 0 & & & \\ 0 & Q & 0 & & \\ & 0 & & & \\ & & & Q & 0 \\ & & & 0 & P \end{bmatrix}}_{QN} \underbrace{\begin{bmatrix} x_k \\ \\ x_{k+N} \end{bmatrix}}_{x_F} + \underbrace{[u_k \quad u_{k+N-1}]}_{u_F^t} \underbrace{\begin{bmatrix} R & 0 & & \\ 0 & R & 0 & \\ & 0 & & \\ & & & 0 \\ & & 0 & R \end{bmatrix}}_{RN} \underbrace{\begin{bmatrix} u_k \\ \\ u_{k+N-1} \end{bmatrix}}_{u_F} \quad \text{Eq. 5-10}$$

Finally, a proper cost function depending on future inputs and states has been obtained:

$$V_N(x_F,u_F) = x_F^t QN x_F + u_F^t RN u_F \quad \text{Eq. 5-11}$$

By substituting the system equations into the cost function, we obtain:

$$\begin{aligned} V_N &= (u_F^t G_u^t + x_k^t G_x^t) QN (G_x x_k + G_u u_F) + u_F^t RN u_F = \\ &= u_F^t [G_u^t QN (G_x x_k + G_u u_F) + RN u_F] + x_k^t G_x^t QN (G_x x_k + G_u u_F) = \\ &= u_F^t (G_u^t QN \cdot G_u + RN) u_F + u_F^t G_u^t QN \cdot G_x x_k + \underbrace{x_k^t G_x^t QN \cdot G_x x_k}_{constant\ w.r.t.\ u_F} + x_k^t G_x^t QN \cdot G_u u_F \end{aligned} \quad \text{Eq. 5-12}$$

As V_N will be optimized with respect to future outputs, the third term, independent on u_F can be omitted:

$$V_N = u_F^t \underbrace{(G_u^t \cdot QN \cdot G_u + RN)}_{\frac{1}{2}H} u_F + \underbrace{2x_k^t \cdot G_x^t \cdot QN \cdot G_u}_{f^t} u_F \quad \text{Eq. 5-13}$$

Above, terms of this equation with the one given to 'quadprog' have been compared and hence, it can be seen that:

$$H = 2u_F^t (G_u^t \cdot QN \cdot G_u + RN) u_F \quad \text{Eq. 5-14}$$

$$f = 2 \cdot G_u^t \cdot QN \cdot G_x \cdot x_k \quad \text{Eq. 5-15}$$

It can be noticed that the value of f^t depends on x_k, namely, the current state, so this dependence must be taken into consideration when developing the control algorithm, as it will vary each cicle.

With that, it has been finally achieved the transformation of the original cost function into a quadratic programming shape:

$$V_N = \frac{1}{2} u_F^t H u_F + f^t u_F \quad \text{Eq. 5-16}$$

5.2.2 Dealing with constraints

In this subsection, the process of achieving properly shaped restrictions from the given equations is shown, but for the sake of concision, this procedure will be exhibited by using a simplified example, where these considerations are taken into account:

- the state vector will be composed of two variables (instead of three, in our case):

$$x_k = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix},$$

- the output vector will be made up of one element,
- the input vector will be a scalar as well.

The specific values chosen in this project for the states and the control actions are the following. Take into consideration that constraints in the outputs have the same values as the ones in the state owing to C being an identity matrix:

$$\begin{bmatrix} -\frac{\pi}{4} \\ -4 \\ -60 \end{bmatrix} \leq \begin{bmatrix} \phi_c \\ \dot{\phi}_c \\ \dot{\theta} \end{bmatrix} \leq \begin{bmatrix} \frac{\pi}{4} \\ 4 \\ 60 \end{bmatrix} \begin{pmatrix} rad \\ rad \cdot s^{-1} \\ rad \cdot s^{-1} \end{pmatrix}$$
$$-90 \leq u \leq 90 \ (\frac{rad}{s^2}) \quad \text{Eq. 5-17}$$

5.2.2.1 Constraints in the states

$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \leq \begin{bmatrix} x_{1,\max} \\ x_{2,\max} \end{bmatrix}$$
$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \geq \begin{bmatrix} x_{1,\min} \\ x_{2,\min} \end{bmatrix} \rightarrow \begin{bmatrix} -1 & 0 \\ 0 & -1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \leq \begin{bmatrix} -x_{1,min} \\ -x_{2,min} \end{bmatrix} \quad \text{Eq. 5-18}$$

Therefore:

$$\underbrace{\begin{bmatrix} I_{n_x} \\ -I_{n_x} \end{bmatrix}}_{A_x} \underbrace{\begin{bmatrix} x_1 \\ x_2 \end{bmatrix}}_{x_k} \leq \underbrace{\begin{bmatrix} x_{1,\max} \\ x_{2,\max} \\ -x_{1,\min} \\ -x_{2,\min} \end{bmatrix}}_{b_x}$$
$$\underbrace{\begin{bmatrix} A_x & 0 & & & \\ 0 & A_x & 0 & & \\ & 0 & & & \\ & & & & 0 \\ & & & 0 & A_x \end{bmatrix}}_{AX} \underbrace{\begin{bmatrix} x_k \\ \\ x_{k+N} \end{bmatrix}}_{x_F} \leq \underbrace{\begin{bmatrix} b_x \\ \\ b_x \end{bmatrix}}_{bX} \quad \text{Eq. 5-19}$$

Thus:

$$AX \cdot x_F \leq bX \quad \text{Eq. 5-20}$$

By substituting the model's equations:

$$AX(G_u u_F + G_x x_k) \leq bX$$
$$AX \cdot G_u u_F + AX \cdot G_x x_k \leq bX$$
$$\underbrace{AX \cdot G_u}_{A_{qp}} \underbrace{u_F}_{x_{qp}} \leq \underbrace{\underbrace{bX}_{b1x} \underbrace{-AX \cdot G_x}_{b2x} x_k}_{b_{qp}} \quad \text{Eq. 5-21}$$

Therefore:

$$AX \cdot G_u \cdot u_F \leq b1x + b2x \cdot x_k \quad \text{Eq. 5-22}$$

As a result, we have obtained equations in the form of a quadratic programming problem:

$$A_{qp}x_{qp} \leq b_{qp}$$

Being the input u_F the variable to be optimized, namely, the x of the QP:

$$u_F = x_{qp}$$

This procedure shall be executed similarly with the outputs and inputs subsequently.

5.2.2.2 Constraints in the outputs

$$y \leq y_{\max}$$
$$y \geq y_{\min} \rightarrow -y \leq -y_{\min} \qquad \text{Eq. 5-23}$$

Hence:

$$\underbrace{\begin{bmatrix} I_{n_y} \\ -I_{n_y} \end{bmatrix}}_{Ay} y \leq \underbrace{\begin{bmatrix} y_{\max} \\ -y_{\min} \end{bmatrix}}_{b_y}$$

$$\underbrace{\begin{bmatrix} A_y & 0 & & & \\ 0 & A_y & 0 & & \\ & 0 & & & \\ & & & & 0 \\ & & & 0 & A_y \end{bmatrix}}_{AY} \underbrace{\begin{bmatrix} y_k \\ \\ y_{k+N} \end{bmatrix}}_{y_F} \leq \underbrace{\begin{bmatrix} b_y \\ \\ b_y \end{bmatrix}}_{bY} \qquad \text{Eq. 5-24}$$

To transform the output equations into equations in the states, we will use the second model equation:

$$y_k = Cx_k \qquad \text{Eq. 5-25}$$

Therefore:

$$\underbrace{\begin{bmatrix} y_k \\ \\ y_{k+N} \end{bmatrix}}_{y_F} = \underbrace{\begin{bmatrix} C & 0 & & & \\ 0 & C & 0 & & \\ & 0 & & & \\ & & & & 0 \\ & & & 0 & C \end{bmatrix}}_{T} \underbrace{\begin{bmatrix} x_k \\ \\ x_{k+N} \end{bmatrix}}_{x_F} \qquad \text{Eq. 5-26}$$

$$y_F = Tx_F$$

Using both equations the following is finally obtained:

$$AY \cdot T \cdot x_F \leq bY$$
$$AY \cdot T \cdot (G_u \cdot u_F + G_x \cdot x_k) \leq bY$$
$$\underbrace{AY \cdot T \cdot G_u}_{A_{qp}} \cdot \underbrace{u_F}_{x_{qp}} \leq \underbrace{\underbrace{bY}_{b1y} \underbrace{-AY \cdot T \cdot G_x}_{b2y} \cdot x_k}_{b_{qp}} \qquad \text{Eq. 5-27}$$

Eventually:

$$AY \cdot T \cdot G_u \cdot u_F \leq b1y + b2y \cdot x_k \qquad \text{Eq. 5-28}$$

5.2.2.3 Constraints in the inputs

$$\left.\begin{array}{c} u \leq u_{\max} \\ u \geq u_{\min} \rightarrow -u \leq -u_{\min} \end{array}\right\} \rightarrow \underbrace{\begin{bmatrix} I_{n_u} \\ -I_{n_u} \end{bmatrix}}_{A_u} u \leq \underbrace{\begin{bmatrix} u_{\max} \\ -u_{\min} \end{bmatrix}}_{b_u}$$

$$\underbrace{\underbrace{\begin{bmatrix} A_u & 0 & & & \\ 0 & A_u & 0 & & \\ & 0 & & & \\ & & & & 0 \\ & & & 0 & A_u \end{bmatrix}}_{AU}}_{A_{qp}} \underbrace{\underbrace{\begin{bmatrix} u_k \\ \\ u_{k+N-1} \end{bmatrix}}_{u_F}}_{x_{qp}} \leq \underbrace{\underbrace{\begin{bmatrix} b_u \\ \\ b_u \end{bmatrix}}_{bU=b1u}}_{b_{qp}} \quad \text{Eq. 5-29}$$

In the end:

$$AU \cdot u_F \leq b1u \quad \text{Eq. 5-30}$$

By adding all the equations of the restrictions, the matrices that will be given as an input to the QP solver can be created ('quadprog' in MATLAB and the optimizer in Julia):

$$\begin{aligned} A_{qp} &= [AX \cdot G_u; AY \cdot T \cdot G_u; AU] \\ b1_{qp} &= [bX; bY; bU] \\ b2_{qp} &= [-AX \cdot G_x; -AY \cdot T \cdot G_x; 0] \end{aligned} \quad \text{Eq. 5-31}$$

At last, the equation below is attained, in the intended QP shape:

$$A_{qp} u_F \leq \underbrace{b1_{qp} + b2_{qp} x_k}_{b_{qp}} \quad \text{Eq. 5-32}$$

The constraints vector (b_{qp}) has been split in two parts due to the dependence of the second term, $b2_{qp}x_k$, on the current state x_k, which will be crucial to the further development of the optimisation algorithm.

5.2.3 Change to incremental model

Even so, it will be necessary for the robot to change its speed, and that will mean changing its wheel's velocity, subsequently symbolizing a change in the reference. This change can be implemented by using incremental variables.

For the purpose of this project, only the reference of the wheels' speed will be changed. However, equations and code are also included in the case references in the inputs or the outputs (which would be the same changes as those done in the states, due to matrix of the model C being an identity matrix). In addition, changes in the state of reference different from the wheels' speed, such as the tilt or its velocity can be performed by simply typing the new reference state.

In this way, the following lines display the model and the QP equations transformed into an incremental model:

New variables:

$$\begin{aligned} \hat{x}_k &= x_k - x_r \\ \hat{y}_k &= y_k - y_r \\ \hat{u}_k &= u_k - u_r \end{aligned} \quad \text{Eq. 5-33}$$

Being x_r, y_r and u_r the reference values for the state, output, and input, respectively. In this particular case, where $\dot{\theta}_r$ is the reference for the speed of the wheels, these values will be:

$$x_r = \begin{bmatrix} 0 \\ 0 \\ \dot{\theta}_r \end{bmatrix}$$
$$y_r = Cx_r = x_r$$
$$u_r = 0$$
Eq. 5-34

Thus, the equations will be the following ones:

$$\hat{x}_F = \begin{bmatrix} \hat{x}_k \\ \hat{x}_{k+1} \\ \vdots \\ \hat{x}_{k+N} \end{bmatrix} = \underbrace{\begin{bmatrix} I_{n_x} \\ A \\ A^2 \\ \vdots \\ A^N \end{bmatrix}}_{G_x} \hat{x}_k + \underbrace{\begin{bmatrix} 0 & \cdots & \cdots & \cdots & \cdots & 0 \\ B & \cdots & \cdots & \cdots & \cdots & 0 \\ AB & B & 0 & \cdots & \cdots & \vdots \\ A^2B & AB & B & 0 & \cdots & \vdots \\ \vdots & \vdots & \vdots & \vdots & \ddots & 0 \\ A^{N-1}B & \cdots & \cdots & \cdots & \cdots & B \end{bmatrix}}_{G_u} \underbrace{\begin{bmatrix} \hat{u}_k \\ \vdots \\ \hat{u}_{k+N-1} \end{bmatrix}}_{u_F}$$
Eq. 5-35

Hence:

$$\hat{x}_F = G_x \hat{x}_k + G_u \hat{u}_F$$
Eq. 5-36

Similarly, it can be demonstrated that the same results apply in the objective function's case. Therefore:

$$V_N = \hat{x}_F^t \cdot QN \cdot \hat{x}_F + \hat{u}_F^t \cdot RN \cdot \hat{u}_F$$
Eq. 5-37

And, when making the development to adjust the cost function into a readable QP function, it can also be demonstrated that it turns out to eventually be:

$$V_N = \hat{u}_F' \underbrace{(G_u^t \cdot QN \cdot G_u + RN)}_{\frac{1}{2}H} \hat{u}_F + \underbrace{2x_k^t \cdot G_x^t \cdot QN \cdot G_u}_{f^t} \hat{u}_F$$
Eq. 5-38

However, when turning to the QP problem scope, a little care must be taken, in the sense that the constraints part of the equation turns slightly different, as shown here in the equations of the constraints in the states, specifically:

$$A_x \cdot x_k \leq b_x$$
$$A_x \cdot (x_k - x_r) \leq b_x - A_x \cdot x_r$$
$$A_x \cdot \hat{x}_k \leq b_x'$$
Eq. 5-39

The same procedure applies when developing the outputs' or the control action's (input) constraints.

Hence, restrictions are dependent on reference variation and, as such, they will be changed every time any of the references vary.

5.2.4 Model integration

This project is developed in simulation. As it has been considered not to implement it in the real world, a model of the self-balancing robot will be programmed, and tests will be performed using this model.

To do so, there are two options: either use the linearised model given by:

$$x_{k+1} = Ax_k + Bu_k$$
Eq. 5-40

Or to use the nonlinear one. The code given by this work permits the usage of either of them, while it has been considered better to use the nonlinear version.

To implement the nonlinear model, the Euler method for solving the ordinary differential equations given by the development of the model stated above is utilized.

The numerical procedure follows these steps. Let us define the variable:

$$\dot{\phi}_c = v \tag{Eq. 5-41}$$

Let us also recall the system model:

$$F = (2a + c \cdot cos(\phi + \phi_0))\ddot{\theta} + (c \cdot cos(\phi + \phi_0) + 2b)\ddot{\phi} - c\dot{\phi}^2 \, sin(\phi + \phi_0) \\ -dsin(\phi + \phi_0) = 0 \tag{Eq. 5-42}$$

Now, introducing the new variable:

$$F = (2a + c \cdot cos\phi_c)\ddot{\theta} + (c \cdot cos\phi_c + 2b)\dot{v} - cv^2 sin\phi_c - dsin\phi_c = 0 \\ \dot{v} = \left[dsin\phi_c + cv^2 sin\phi_c - (2a + c \cdot cos\phi_c)\ddot{\theta}\right](c \cdot cos\phi_c + 2b)^{-1} \tag{Eq. 5-43}$$

Therefore, a system of two first order nonlinear differential equations has been attained:

$$\begin{cases} \dot{\phi}_c = v \\ \dot{v} = \left[dsin\phi_c + cv^2 sin\phi_c - (2a + c \cdot cos\phi_c)\ddot{\theta}\right](c \cdot cos\phi_c + 2b)^{-1} \end{cases} \tag{Eq. 5-44}$$

On the other hand, the Euler method states that:

$$\dot{v} = \frac{v_{k+1} - v_k}{\delta} \tag{Eq. 5-45}$$

Being δ the step size.

Renaming ϕ_c to ϕ for the sake of concision and using the Euler method into the previous system of equations, the following is obtained:

$$\begin{cases} v_{k+1} = v_k + \delta\left[dsin\phi_c + cv^2 sin\phi_c - (2a + c \cdot cos\phi_c)\ddot{\theta}\right](c \cdot cos\phi_c + 2b)^{-1} \\ \phi_{k+1} = \phi_k + \delta v_k \end{cases} \tag{Eq. 5-46}$$

In addition:

$$\ddot{\theta}_k = \left.\frac{d\dot{\theta}}{dt}\right|_k \rightarrow \dot{\theta}_{k+1} = \dot{\theta}_k + \delta \underbrace{\ddot{\theta}_k}_{u_k} \tag{Eq. 5-47}$$

Thus, the prediction of the state based on the nonlinear model are the following:

$$x_{k+1} = \begin{bmatrix} \phi_{k+1} \\ \dot{\phi}_{k+1} \\ \dot{\theta}_{k+1} \end{bmatrix} = \\ = \begin{bmatrix} \phi_k + \delta\dot{\phi}_k \\ \dot{\phi}_k + \delta\left[dsin\phi_c + cv^2 sin\phi_c - (2a + c \cdot cos\phi_c)\ddot{\theta}\right](c \cdot cos\phi_c + 2b)^{-1} \\ \dot{\theta}_k + \delta u_k \end{bmatrix} \tag{Eq. 5-48}$$

These will be the preferred equations used when programming instead of those of the linearised model given by:

$$x_{k+1} = Ax_k + Bu_k$$ Eq. 5-49

5.3 Particularization

On the basis of the development done previously, this section will treat the specific equations that will be leading the self-balancing robot for its stabilization. The values here obtained for the matrices, vectors and scalars will be the ones used for creating the programme.

The model equation is recalled here:

$$F = (2a + c \cdot cos(\phi + \phi_0))\ddot{\theta} + (c \cdot cos(\phi + \phi_0) + 2b)\ddot{\phi}$$
$$-c\dot{\phi}^2 \, sin(\phi + \phi_0) - dsin(\phi + \phi_0) = 0$$ Eq. 5-50

Being the constants:

$$a = \left(\frac{3}{2}m_r + \frac{1}{2}M\right)R^2$$
$$b = M \cdot L^2$$
$$c = MRL$$
$$d = MgL$$ Eq. 5-51

The table of the system variables is recalled as well:

Parameter	Notation	Value
Wheel mass	m_r	$0.044\ kg$
Wheel radium	R	$0.053\ m$
Mass of the vehicle without wheels	M	$0.745\ kg$
Distance from the wheel axis and the centre of gravity	L	$0.0805\ m$
Gravity	g	$9.81 \frac{m}{s^2}$
Tilt with respect to the vertical	ϕ	
Centre of gravity deviation tilt	ϕ_0	n/a
Corrected tilt	ϕ_c	

It is desired to perform a linearisation around the equilibrium point whose conditions are:

$$\phi_{c_{eq}} = 0$$
$$\dot{\phi}_{c_{eq}} = 0$$
$$\ddot{\phi}_{c_{eq}} = 0$$
$$\cos\left(\phi_{c_{eq}}\right) \approx 1$$
$$\sin\left(\phi_{c_{eq}}\right) \approx \phi_{c_{eq}}$$ Eq. 5-52

To linearise, the derivatives of F with respect to each system variable are performed:

$\left.\frac{\partial F}{\partial \ddot{\theta}}\right\|_{eq} = 2a + c$	(1) Derivative of F with respect to the angular acceleration of the wheel
$\left.\frac{\partial F}{\partial \phi_c}\right\|_{eq} = -d$	(2) Derivative of F with respect to the corrected tilt
$\left.\frac{\partial F}{\partial \dot{\phi}_c}\right\|_{eq} = 0$	(3) Derivative of F with respect to the corrected tilt speed
$\left.\frac{\partial F}{\partial \ddot{\phi}_c}\right\|_{eq} = 2b + c$	(4) Derivative of F with respect to the corrected tilt acceleration

From the previous equations, the linearised model is obtained:

$$F_{lin} = (2a + c)\ddot{\theta} - d\phi_c + (2b + c)\ddot{\phi}_c = 0 \quad \text{Eq. 5-53}$$

Renaming:

$$e = 2a + c$$
$$f = 2b + c \quad \text{Eq. 5-54}$$

Then:

$$F_{lin} = e \cdot \ddot{\theta} + d \cdot \phi_c + f \cdot \ddot{\phi}_c = 0 \quad \text{Eq. 5-55}$$

The state-space of the system can be obtained:

$$\begin{bmatrix} \dot{\phi}_c \\ \ddot{\phi}_c \\ \ddot{\theta} \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 \\ \frac{d}{f} & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} \phi_c \\ \dot{\phi}_c \\ \dot{\theta} \end{bmatrix} + \begin{bmatrix} 0 \\ -\frac{e}{f} \\ 1 \end{bmatrix} \ddot{\theta} \equiv \dot{x} = Ax + Bu \quad \text{Eq. 5-56}$$

As the Jetson Nano works in discrete time and the previous system is continuous, it would be necessary to discretize it. The sampling time used in this project has been 20 milliseconds, and that is the figure used for discretisation, but it should be changed provided a different time is obtained when running the programme. The following is obtained:

$$x_{k+1} = A_d x_k + B_d u_k$$
$$y_k = C_d x_k \quad \text{Eq. 5-57}$$

The following state weighting matrix, Q, and the control action one, R, have been chosen according to experimental tests:

$$Q = \begin{bmatrix} 10 & 0 & 0 \\ 0 & 10 & 0 \\ 0 & 0 & 10 \end{bmatrix}$$
$$R = 1 \quad \text{Eq. 5-58}$$

6 THE CODING

In this chapter, the code algorithm and the code itself will be explained. But previously, a section of clarification about the mathematical developments shown in previous chapters is shown not only to make the reader comprehend the programme, but to also serve as a recall. In addition, some aspects considered necessary to the proper understanding of the code are told. Amongst these ones, JuMP language and Ipopt optimizer most prominent features are stated and some of their coding complications are remarked.

6.1 Mathematical clarifications

The algorithm developed in MATLAB and Julia follows the mathematical procedure explained in chapter 6. To sum it up, the pseudocode will be shown for easier comprehension. Later, the code itself will be shown, with annotations:

```
Initialize variables
Discretize model matrices
Create Gu and Gx matrices
Calculate P
Create QN and RN matrices
Create QP matrices: Aqp, b1qp and b2qp
FOR i=1:simulation steps
      IF there is a change in the reference
            Update b1qp
      Update f(x_k)
      Optimize the QP problem
      Extract the first component of u_F
      Update the state with the model equations
Plot the results
```

So, basically, what the program does is to run a QP problem based on the equations developed in previous chapters. Afterwards, it applies the control action, result of extracting the first component of the QP's solution, which will be the sequence of future outputs u_F. The rest of the planned outputs will be discarded as only the first one will be applied (i.e. u_k). Then, the new model state is integrated by using its equations, i.e., these in the case of choosing the nonlinear version:

$$x_{k+1} = \begin{bmatrix} \phi_{k+1} \\ \dot{\phi}_{k+1} \\ \dot{\theta}_{k+1} \end{bmatrix} = \left[\begin{matrix} \phi_k + \delta\dot{\phi}_k \\ \phi_k + \delta\left[dsin\phi_c + cv^2sin\phi_c - (2a + c \cdot cos\phi_c)\ddot{\theta}\right](c \cdot cos\phi_c + 2b)^{-1} \\ \dot{\theta}_k + \delta u_k \end{matrix} \right] \quad \text{Eq. 6-1}$$

Being δ the step size of the Euler method, which should be, ideally, smaller than the sampling time chosen to discretize the model's nonlinear equations.

Or, in the case the model wanted to be integrated by using the linearised equations, these should be the ones used in the programme:

$$x_{k+1} = A_d x_k + B_d u_k \qquad \text{Eq. 6-2}$$

Being A_d and B_d the discrete version of the original matrices. i.e., given the linear time-invariant state-space representation of the system:

$$\begin{bmatrix} \dot{\phi}_c \\ \ddot{\phi}_c \\ \ddot{\theta} \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 \\ \frac{d}{f} & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} \phi_c \\ \dot{\phi}_c \\ \dot{\theta} \end{bmatrix} + \begin{bmatrix} 0 \\ -\frac{e}{f} \\ 1 \end{bmatrix} \ddot{\theta} \equiv \dot{x} = Ax + Bu$$

$$A = \begin{bmatrix} 0 & 1 & 0 \\ \frac{d}{f} & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \qquad \text{Eq. 6-3}$$

$$B = \begin{bmatrix} 0 \\ -\frac{e}{f} \\ 1 \end{bmatrix}$$

Euler method is used to represent the derivative in a discrete-time form:

$$\frac{x_{k+1} - x_k}{\delta} = Ax_k + Bu_k$$

$$x_{k+1} = (\delta A + I)x_k + \delta B u_k \qquad \text{Eq. 6-4}$$

$$x_{k+1} = A_d x_k + B_d u_k$$

Being, eventually:

$$A_d = \delta \cdot A + I$$

$$B_d = \delta \cdot B \qquad \text{Eq. 6-5}$$

The discrete-time version of the continuous-time matrices of the model. The subscript d stands for discrete.

However, if we pay close attention to the chapter's six equations, it will be noticed that some of the variables calculated in each optimization problem depend on the current state. These variables and their formulae are developed subsequently.

Regarding the constraints, it was said previously that the right-side parts of the constraints' equations were divided into two: $b1qp$ and $b2qp$, as it can be seen here:

$$AX \cdot G_u \cdot u_F \leq b1x + b2x \cdot x_k \qquad \text{Eq. 6-6}$$

Where:

$$b1qp = b1x$$

$$b2qp = b2x \qquad \text{Eq. 6-7}$$

And the same applies to the constraints in the outputs and inputs (or control actions).

Therefore, it can be noticed that the right side of the inequality depends on x_k, i.e., the current state. Owing to this fact, constraints will be necessarily changed every iteration of the loop, as the state will inevitably change due to the integration of the model. Because of the nature of Julia's optimizer, the model shall be reconfigured

each iteration.

In addition, let us recall the cost function:

$$V_N = \frac{1}{2} u_F^t H u_F + f^t u_F \quad \text{Eq. 6-8}$$

It includes a vector, f, which depends on the current state:

$$f = 2 \cdot G_u^t \cdot QN \cdot G_x \cdot x_k \quad \text{Eq. 6-9}$$

Hence, this vector will be included inside the loop to change its value every time the state changes, which will occur mostly during the transient.

Apart from these two variables and the model's state update, nothing will change during execution time, and the remaining to be done will be to store the necessary variables into their respective vectors to be later plotted and running the optimizer time after time to obtain the optimum.

Before showing the full programme, it is also necessary to recall that whenever there is a change in reference, this—another change in the constraints—plays out:

$$\begin{gathered} A_x \cdot (x_k - x_r) \leq b_x - A_x \cdot x_r \\ A_x \cdot \hat{x}_k \leq b'_x \end{gathered} \quad \text{Eq. 6-10}$$

Hence:

$$\begin{bmatrix} A_x & \\ & A_x \end{bmatrix} \begin{bmatrix} \hat{x}_k \\ \hat{x}_{k+N} \end{bmatrix} \leq \begin{bmatrix} b'_x \\ b'_x \end{bmatrix} \quad \text{Eq. 6-11}$$

Therefore:

$$AX \cdot \hat{x}_F \leq bX' \quad \text{Eq. 6-12}$$

The same occurs in the case of outputs and inputs:

$$\begin{bmatrix} A_y & \\ & A_y \end{bmatrix} \begin{bmatrix} \hat{y}_k \\ \hat{y}_{k+N} \end{bmatrix} \leq \begin{bmatrix} b'_y \\ b'_y \end{bmatrix} \quad \text{Eq. 6-13}$$

Therefore:

$$AY \cdot \hat{y}_F \leq bY' \quad \text{Eq. 6-14}$$

And finally:

$$\begin{bmatrix} A_u & \\ & A_u \end{bmatrix} \begin{bmatrix} \hat{u}_k \\ \hat{u}_{k+N-1} \end{bmatrix} \leq \begin{bmatrix} b'_u \\ b'_u \end{bmatrix} \quad \text{Eq. 6-15}$$

Therefore:

$$AU \cdot \hat{u}_F \leq bU' \quad \text{Eq. 6-16}$$

Nonetheless, take into account that, for this project, only changes in the wheels' speed have been considered,

meaning so that only the states will change their references. The outputs, being a direct transformation of the states into themselves, will change their reference in the same way. Thus, it would be redundant to implement both the states and outputs changes in reference, so only the former has been activated in the code.

Regarding the inputs, no reference setpoint was considered, though its implementation is also present in the code, although neither activated. To do so, simply uncomment the respective lines.

It is important to take into consideration that not every setpoint combination of states, outputs—in the case matrix C was different from the identity matrix—and control actions shall give a feasible solution.

Nevertheless, in the program, the updating of bX', bY' and bU'matrices has been done in such a way that they are the non-prime ones with their components updated.

Thus, the components of $b1qp$ change:

$$b1qp = [bX'; bY'; bU'] \quad \text{Eq. 6-17}$$

Every time any of the references change.

Let us recall that the whole resources vector b_{qp} from a QP problem in the form:

$$A_{qp}u_F \leq b_{qp} = b1_{qp} + b2_{qp}x_k \quad \text{Eq. 6-18}$$

Already changed every iteration because of its dependence of the state x_k, which changes with each. It can now be seen that this vector will change twice—first, from the state dependence and secondly, from the reference change—whenever there is a change in any of the references.

6.2 Optimizer in Julia

6.2.1 JuMP

Figure 6-1. JuMP's logo.

JuMP is a package for Julia which depends on solvers to solve optimization problems. Most solvers are not written in Julia, and some require commercial licenses to use, so installation is often more complex.

This domain-specific modelling language for mathematical optimization embedded in Julia currently supports a number of open-source and commercial solvers—the term "solver" is used as a synonym for "optimizer". The convention in code, however, is to always use "optimizer", e.g., `Ipopt.Optimizer`.—for a variety of problem classes, including **linear programming**, **mixed-integer programming**, **second-order conic programming**, **semidefinite programming**, and **nonlinear programming**. JuMP's features include:

- User friendliness
    - Syntax that mimics natural mathematical expressions.
    - Complete documentation.
- Speed
    - Benchmarking has shown that JuMP can create problems at similar speeds to special-purpose modelling languages such as AMPL.
    - JuMP communicates with most solvers in memory, avoiding the need to write intermediary files.
- Solver independence
    - JuMP uses a generic solver-independent interface provided by the MathOptInterface package, making it easy to change between a number of open-source and commercial optimization software packages ('solvers').
    - Currently supported solvers include Artelys Knitro, Bonmin, Cbc, Clp, Couenne, CPLEX, ECOS, FICO Xpress, IPOPT, Gurobi, Ipopt, MOSEK, NLopt, and SCS.
- Access to advanced algorithmic techniques
    - Including efficient LP re-solves which previously required using solver-specific and/or low-level C++ libraries.
- Ease of embedding
    - JuMP itself is written purely in Julia. Solvers are the only binary dependencies.
    - Being embedded in a general-purpose programming language makes it easy to solve optimization problems as part of a larger workflow (e.g., inside a simulation, behind a web server, or as a subproblem in a decomposition algorithm).
        - As a trade-off, JuMP's syntax is constrained by the syntax available in Julia.
    - JuMP is MPL licensed, meaning that it can be embedded in commercial software that complies with the terms of the license.

Although JuMP has not reached version 1.0 yet, the releases are stable enough for everyday use and are being used in a number of research projects and neat applications by a growing community of users who are early adopters. JuMP remains under active development.

More information can be found in [14], where all the functions used by JuMP and related to each solver are explained in detail, but will not be covered inside this document.

6.2.2 Ipopt

Figure 6-2. COIN-OR's logo.

Ipopt (Interior Point OPTimizer, pronounced eye-pea-Opt) is a software package for large-scale nonlinear optimization. It is written in Fortran and C and is released under the EPL (formerly CPL).

IPOPT implements a primal-dual interior point method and uses line searches based on Filter methods (Fletcher and Leyffer). IPOPT can be called from various modelling environments and C.

IPOPT is part of the COIN-OR project, which is a project that aims to 'create for mathematical software what the open literature is for mathematical theory'.

It is designed to find (local) solutions of mathematical optimization problems of the form:

$$\min_{x \in R^n} f(x)$$

$$s.t. \begin{cases} g_L \leq g(x) \leq g_U \\ x_L \leq x \leq x_U \end{cases}$$

Where $f(x): R^n \rightarrow R$ is the objective function, and $g(x): R^n \rightarrow R^m$ are the constraint functions. The vectors g_L and g_U denote the lower and upper bounds on the constraints, and the vectors x_L and x_U are the bounds on the variables x. The functions $f(x)$ and $g(x)$ can be nonlinear and nonconvex but should be twice continuously differentiable. Note that equality constraints can be formulated in the above formulation by setting the corresponding components of g_L and g_U to the same value.

Ipopt solves QP, LP and NLP problems and, as such, it is valid to solve our problem. **Quadratic programming** (QP) is the process of solving a special type of mathematical optimization problem—specifically, a (linearly constrained) quadratic optimization problem, that is, the problem of optimizing (minimizing or maximizing) a quadratic function of several variables subject to linear constraints on these variables. Quadratic programming is a particular type of nonlinear programming.

To install and use Ipopt, type the following in a Julia's REPL:

```
import Pkg

Pkg.add("Ipopt")

using JuMP

using Ipopt

model=Model(Ipopt.Optimizer)
```

6.2.3 Optimization problem

Even though having stated before that no details will be given about JuMP's and Ipopt's programming, some key issues must be covered to fully understand the programme in Julia, as it is considered to be not only a recent language, but also a less-known than MATLAB one. Therefore, some explanation can be useful for the reader in case they did not know the language previously. On the contrary, if you know how to use Julia and JuMP, please skip this part. Follow these steps to create and solve a very simple optimization problem.

Models are created with the `Model` function. The optimizer can be set either in `Model()` or by calling `set_optimizer`:

```
julia> model = Model(IPOPT.Optimizer)

A JuMP Model

Feasibility problem with:

Variables: 0

Model mode: AUTOMATIC

CachingOptimizer state: NO_OPTIMIZER

Solver name: IPOPT
```

equivalently,

```
julia> model = Model();

julia> set_optimizer(model, IPOPT.Optimizer);

julia> model

A JuMP Model

Feasibility problem with:

Variables: 0

Model mode: AUTOMATIC

CachingOptimizer state: NO_OPTIMIZER

Solver name: IPOPT
```

The following commands will create two variables, `x`, and `y`, with both lower and upper bounds. Note the first argument is our model `model`. These variables (`x` and `y`) are associated with this model and cannot be used in another model.

```
julia> @variable(model, 0 <= x <= 2)

x

julia> @variable(model, 0 <= y <= 30)

y
```

It is also possible to specify different combinations of bounds, i.e., only lower bounds, only upper bounds, or no

bounds.

Next, the objective will be set. Note again the `model`, so it is known which model's objective is being set! The objective sense, `Max` or `Min`, should be provided as the second argument. Note also that there is no multiplication `*` symbol between `5` and the variable `x`—Julia is smart enough to not need it!—Feel free to use `*` if it makes you feel more comfortable, as it has been done with `3 * y`. (It has been intentionally inconsistent here to demonstrate different syntax; however, it is good practice to pick one way or the other consistently in the code.)

```
julia> @objective(model, Max, 5x + 3 * y)
```

```
5 x + 3 y
```

Adding constraints is a lot like setting the objective. Here a less-than-or-equal-to constraint is created using `<=`, but equality constraints can be created as well using `==` and greater-than-or-equal-to constraints with `>=`:

```
julia> @constraint(model, con, 1x + 5y <= 3)
```

```
con : x + 5 y <= 3.0
```

Note that in a similar manner to the `@variable` macro, the constraint has been named `con`. This will bind the constraint to the Julia variable con for later analysis.

Models are solved with the `JuMP.optimize!` function:

```
julia> optimize!(model)
```

After the call to `JuMP.optimize!` has finished, it is needed to query what happened. The solve could terminate for a number of reasons. First, the solver might have found the optimal solution or proved that the problem is infeasible. However, it might also have run into numerical difficulties or terminated due to a setting such as a time limit. It can be asked to the solver why it stopped using the `JuMP.termination_status` function:

```
julia> termination_status(model)
```

```
OPTIMAL::TerminationStatusCode = 1
```

In this case, IPOPT returned OPTIMAL, this means that it has found the optimal solution.

JuMP has support for general smooth nonlinear (convex and nonconvex) optimization problems. JuMP is able to provide exact, sparse second-order derivatives to solvers. This information can improve solver accuracy and performance.

Nonlinear objectives and constraints are specified by using the `@NLobjective` and `@NLconstraint` macros. Due to having chosen a QP problem as the one to solve in our project, `@NLobjective` will be needed.

6.3 Code in Julia

There is nothing else that needs to be commented to understand the programme, which is shown below, with inline comments to easier understanding of it. This is the Julia version against which the MATLAB's one will be compared. In addition, the plotting programme used—from where all the figures were drawn and which could be useful for the reader to own—is shown as well. The programmes themselves are shown here.

6.3.1 Optimization

```
################################################################################
################
# packages #

################################################################################
################
```

```
    using LinearAlgebra     # for identity matrices I
    using ControlSystems    # for 'dlqr'
    using JuMP
    using Ipopt
    using Plots
    using Printf


################################################################################
###############
# problem parameters #

################################################################################
###############

    # model parameters
    m_r=.044;
    M=.745;
    r=.053;
    L=.0805;
    g=9.81;

    # model auxiliary variables
    a=(1.5*m_r+.5*M)*r^2;
    b=M*L^2;
    c=r*L*M;
    d=M*g*L;
    e=2*a+c;
    f=2*b+c;

    # continuous time model matrices
    A=[0 1 0; d/f 0 0; 0 0 0];
    B=[0 -e/f 1]';
    nx,nu=size(B);
    C=I;             #  no observer will be needed
    ny=nx;

    # valores límite
    xmax=[90*pi/180;4;60];
    xmin=-xmax;
    umax=90;
    umin=-umax;
    ymax=[90*pi/180;4;60];
    # ymax=C*xmax;
    ymin=-ymax;

    # discretization
    deltaT=0.020;   #  sampling time
    A=A*deltaT+I;   #  these are de A_d and B_d, though named likewise to
simplify
    B=B*deltaT;
```

```
    # MPC variables
    N=3;            #  prediction horizon


#############################################################################
##############
# matrices of the model to make it look like this: x_F=Gu·u_F+G_x·x_k #

#############################################################################
##############
    Gx=zeros((N+1)*nx,nx);
    Gu=zeros((N+1)*nx,N*nu);
    for i=1:N+1
        Gx[(i-1)*nx+1:i*nx,:]=A^(i-1);
    end
    for i=2:N+1
        for j=1:i-1
            Gu[(i-1)*nx+1:(i)*nx,(j-1)*nu+1:j*nu]=A^(i-j-1)*B;
        end
    end


#############################################################################
###############
# matrices of the cost function to make it look like this: V = x_F^t·QN·x_F +
u_F^t·RN·u_F #

#############################################################################
###############
    Q=10*Matrix(I,nx,nx);
    R=Matrix(I,nu,nu);
    QN=zeros(nx*(N+1),nx*(N+1));
    RN=zeros((nu*N),(nu*N));

    K=-dlqr(A,B,Q,R);
    AK=A+B*K;
    QK=Q+K'*R*K;
    P=dlyap(AK',QK);

    for i=1:N
        QN[(i-1)*nx+1:i*nx,(i-1)*nx+1:i*nx]=Q;
    end
    QN[N*nx+1:(N+1)*nx,N*nx+1:(N+1)*nx]=P;
    for i=1:N
        RN[(i-1)*nu+1:i*nu,(i-1)*nu+1:i*nu]=R;
    end


#############################################################################
###############
```

```
# constraints

################################################################################
###############
    # initial references
    xr=[0; 0; 0];
    # setpoints in y and u deactivated because of no changes in them during
simulation
    # activate the corresponding in case there were changes
    # yr=C*xr;
    # ur=0;

    # constraints in the states
    Ax=[Matrix(I,nx,nx);Matrix(-I,nx,nx)];
    bx=[xmax;-xmin]-Ax*xr;
    nrx=length(Ax[:,1]);                            # number of constraints
in x
    AX=zeros((N+1)*nrx,(N+1)*nx);
    bX=zeros((N+1)*nrx,1);
    for i=1:N+1
        AX[(i-1)*nrx+1:i*nrx,(i-1)*nx+1:i*nx]=Ax;
        bX[(i-1)*nrx+1:i*nrx,:]=bx;
    end

    # constraints in the outputs
    Ay=[Matrix(I,ny,ny);Matrix(-I,ny,ny)];
    by=[ymax;-ymin];
    # uncomment if reference changes in y or if C is different from identity
    # by=[ymax;-ymin]-Ay*yr;
    nry=length(Ay[:,1]);
    AY=zeros((N+1)*nry,(N+1)*ny);
    bY=zeros((N+1)*nry,1);
    T=zeros((N+1)*nx,(N+1)*nx);
    for i=1:N+1
        AY[(i-1)*nry+1:i*nry,(i-1)*ny+1:i*ny]=Ay;
        bY[(i-1)*nry+1:i*nry,:]=by;
    end

    # constraints in the control action
    Au=[Matrix(I,nu,nu);Matrix(-I,nu,nu)];
    bu=[umax;-umin];
    # uncomment if there are changes in u's reference
    # bu=[umax;-umin]-Au*ur;
    nru=length(Au[:,1]);
    AU=zeros(N*nru,(N)*nu);
    bU=zeros(N*nru,1);
    for i=1:N
        AU[(i-1)*nru+1:i*nru,(i-1)*nu+1:i*nu]=Au;
        bU[(i-1)*nru+1:i*nru,:]=bu;
    end
```

```
    # creation of the matrices being used in the QP problem
    Aqp=[AX*Gu;AY*T*Gu;AU];
    b1qp=[bX;bY;bU];
    b2qp=[-AX*Gx;-AY*T*Gx;zeros(N*nru,nx)];

    H=2*(Gu'*QN*Gu+RN);

################################################################################
###############
# simulation #

################################################################################
###############

    pasoE=deltaT;            # Euler method's integration step
    Nsteps=200
    xSim=[1:Nsteps];

    ySim1=zeros(Nsteps);
    ySim2=zeros(Nsteps);
    ySim3=zeros(Nsteps);
    ySim4=zeros(Nsteps);
    elapsed=zeros(Nsteps);
    ref=zeros(Nsteps);

    function MPC()
        global ySim1,ySim2,ySim3,xr,b1qp,ref;
        x0=[pi/180*48; 0; 0];    # initial state
        xk=x0;

        for i=1:Nsteps
            if i==201              # loop for reference change in simulation
                xr=[0; 0; 50];
                bx=[xmax;-xmin]-Ax*xr;
                for j=1:N+1
                    bX[(j-1)*nrx+1:j*nrx,:]=bx;
                end
                b1qp=[bX;bY;bU];
            elseif i==401
                xr=[0; 0; 15];
                bx=[xmax;-xmin]-Ax*xr;
                for j=1:N+1
                    bX[(j-1)*nrx+1:j*nrx,:]=bx;
                end
                b1qp=[bX;bY;bU];
            end

            ref[i]=xr[3];

            # solving the optimization problem
            f=(2*(xk-xr)'*Gx'*QN*Gu)';
```

```
model=Model(optimizer_with_attributes(Ipopt.Optimizer,"print_level"=>0));
            @variable(model,uF[1:N]);
            @NLobjective(model,Min,sum(0.5*H[i,j]*uF[i]*uF[j] for i=1:N,
j=1:N)+sum(f[i]*uF[i] for i=1:N));
            @constraint(model,Aqp*uF .<= b1qp+b2qp*(xk-xr));
            optimize!(model);
            elapsed[i]=solve_time(model);
            uk=value.(uF[1]);

            # nonlinear model
            phiC=xk[1];
            vk=xk[2];
            thetaDot=xk[3];
            xk[1]=phiC+pasoE*vk;
            xk[2]=vk+pasoE*(d*sin(phiC)-c*sin(phiC)*vk^2-
(2*a+c*cos(phiC))*uk)*(c*cos(phiC)+2*b)^-1;
            xk[3]=thetaDot+pasoE*uk;
            ySim1[i]=xk[1];
            ySim2[i]=xk[2];
            ySim3[i]=xk[3];
            ySim4[i]=uk;

            # linearised model
            # xk=A*xk+B*uk;

            ySim1[i]=xk[1];
            ySim2[i]=xk[2];
            ySim3[i]=xk[3];
            ySim4[i]=uk;
        end
        ySim1*=180/pi;          # units change
        avg=0;
        # loop for measuring computing time
        for i=1:Nsteps
            avg=avg+elapsed[i]/Nsteps;
        end
        return @printf "%.6f seconds of average in solving the QP\n" avg
    end
```

6.3.2 Plotting

```
plotly()

xmaxRef=ones(length(xmax),Nsteps).*[xmax[1]*180/pi;xmax[2:end]];
umaxRef=ones(length(umax),Nsteps).*umax;

# steady/ changing reference
pSim1=plot(xSim,[xmaxRef[1,:] -xmaxRef[1,:] ySim1
zeros(Nsteps)],label=["Upper value" "Lower value" "Tilt"
"Reference"],title="Tilt",ylabel="°",xlabel="Simulation cycles")
```

```
pSim2=plot(xSim,[xmaxRef[2,:] -xmaxRef[2,:] ySim2
zeros(Nsteps)],label=["Upper value" "Lower value" "Tilt's speed"
"Reference"],title="Tilt's speed",ylabel="rad·s⁻¹",xlabel="Simulation
cycles")
pSim3=plot(xSim,[xmaxRef[3,:] -xmaxRef[3,:] ySim3 ref],label=["Upper value"
"Lower value" "Wheels' speed" "Reference"],title="Wheels'
speed",ylabel="rad·s⁻¹",xlabel="Simulation cycles")
pSim4=plot(xSim,[umaxRef[:] -umaxRef[:] ySim4 zeros(Nsteps)],label=["Upper
value" "Lower value" "Control action" "Reference"],title="Control
action",ylabel="rad·s⁻²",xlabel="Simulation cycles")
savefig(pSim1,"pSim1.png")
savefig(pSim2,"pSim2.png")
savefig(pSim3,"pSim3.png")
savefig(pSim4,"pSim4.png")

# linear vs. NL comparison
(ySim1NL,ySim2NL,ySim3NL,ySim4NL)=(ySim1,ySim2,ySim3,ySim4);
pSim1=plot(xSim,[xmaxRef[1,:] -xmaxRef[1,:] ySim1 ySim1NL
zeros(Nsteps)],label=["Upper value" "Lower value" "Linear" "Nonlinear"
"Reference"],title="Tilt",ylabel="°",xlabel="Simulation cycles")
pSim2=plot(xSim,[xmaxRef[2,:] -xmaxRef[2,:] ySim2 ySim2NL
zeros(Nsteps)],label=["Upper value" "Lower value" "Linear" "Nonlinear"
"Reference"],title="Tilt's speed",ylabel="rad·s⁻¹",xlabel="Simulation
cycles")
pSim3=plot(xSim,[xmaxRef[3,:] -xmaxRef[3,:] ySim3 ySim3NL ref],label=["Upper
value" "Lower value" "Linear" "Nonlinear" "Reference"],title="Wheels'
speed",ylabel="rad·s⁻¹",xlabel="Simulation cycles")
pSim4=plot(xSim,[umaxRef[:] -umaxRef[:] ySim4 ySim4NL
zeros(Nsteps)],label=["Upper value" "Lower value" "Linear" "Nonlinear"
"Reference"],title="Control action",ylabel="rad·s⁻²",xlabel="Simulation
cycles")
savefig(pSim1,"pSim1.png")
savefig(pSim2,"pSim2.png")
savefig(pSim3,"pSim3.png")
savefig(pSim4,"pSim4.png")

# optimization time
avg=0;
    for i=1:Nsteps
        global avg
        avg=avg+elapsed[i]/Nsteps;
    end
plot(xSim,[elapsed ones(Nsteps)*avg],title="Optimization
time",label=["Absolute time" "Average time"],
ylabel="Seconds",xlabel="Simulation cycles");
savefig("timing.png")
```

6.4 Code in MATLAB

In MATLAB, an analogue programme was written. The only difference one can find is that the solver of the QP problem is the MATLAB function ‘quadprog’.

6.4.1 Optimization

The programme is shown here, with inline explanatory comments:

```
%% variables initialisation
% model parameters
m_r=0.044;
M=0.745;
r=0.053;
L=0.0805;
g=9.81;

% model auxiliary variables
a=(1.5*m_r+.5*M)*r^2;
b=M*L^2;
c=r*L*M;
d=M*g*L;
e=(2*a+c);
f=(2*b+c);

% continuous time model matrices
A=[0 1 0; d/f 0 0; 0 0 0];
B=[0; -e/f; 1];
[nx,nu]=size(B);
C=eye(nx);  % no observer needed
ny=length(C(:,1));

% limit values
xmax=[90*2*pi/360;4;60];
xmin=-xmax;
umax=90;
umin=-umax;
ymax=[90*2*pi/360;4;60];
% ymax=C*xmax; % in the case C was different from identity
ymin=-ymax;

% discretisation
deltaT=0.020; % sampling time
A=A*deltaT+eye(nx);
B=B*deltaT;

% MPC variables
N=3;    % prediction horizon

% initial setpoint
xr=[0; 0; 0];
% yr=C*xr;
% ur=0;

%% model's matrices
Gx=zeros((N+1)*nx,nx);
Gu=zeros((N+1)*nx,N*nu);
for i=1:N+1
```

```
   Gx((i-1)*nx+1:i*nx,:)=A^(i-1);
end
for i=2:N+1
   for j=1:i-1
      Gu((i-1)*nx+1:(i)*nx,(j-1)*nu+1:j*nu)=A^(i-j-1)*B;
   end
end

%% cost function's matrices
Q=10*eye(nx);
R=1;
QN=zeros(nx*(N+1));
RN=zeros(nu*N);

K=-dlqr(A,B,Q,R);
AK=A+B*K;
QK=Q+K'*R*K;
P=dlyap(AK',QK);

for i=1:N
   QN((i-1)*nx+1:i*nx,(i-1)*nx+1:i*nx)=Q;
end
QN(N*nx+1:(N+1)*nx,N*nx+1:(N+1)*nx)=P;
for i=1:N
   RN((i-1)*nu+1:i*nu,(i-1)*nu+1:i*nu)=R;
end

%% constraints
% in the states
Ax=[eye(nx);-eye(nx)];
bx=[xmax;-xmin]-Ax*xr;
nrx=length(Ax(:,1));     % number of constraints in x
AX=zeros((N+1)*nrx,(N+1)*nx);
bX=zeros((N+1)*nrx,1);
for i=1:N+1
   AX((i-1)*nrx+1:i*nrx,(i-1)*nx+1:i*nx)=Ax;
   bX((i-1)*nrx+1:i*nrx,:)=bx;
end

% in the outputs
Ay=[eye(ny);-eye(ny)];
by=[ymax;-ymin];
% by=[ymax;-ymin]-Ay*yr;
nry=length(Ay(:,1));
AY=zeros((N+1)*nry,(N+1)*ny);
bY=zeros((N+1)*nry,1);
T=zeros((N+1)*nx);
for i=1:N+1
    AY((i-1)*nry+1:i*nry,(i-1)*ny+1:i*ny)=Ay;
    bY((i-1)*nry+1:i*nry,:)=by;
end

% in the control actions
Au=[eye(nu);-eye(nu)];
bu=[umax;-umin];
% bu=[umax;-umin]-Au*ur;
nru=length(Au(:,1));
AU=zeros(N*nru,(N)*nu);
```

```
bU=zeros(N*nru,1);
for i=1:N
    AU((i-1)*nru+1:i*nru,(i-1)*nu+1:i*nu)=Au;
    bU((i-1)*nru+1:i*nru,:)=bu;
end

% creation of the matrices used by quadprog
Aqp=[AX*Gu;AY*T*Gu;AU];
b1qp=[bX;bY;bU];
b2qp=[-AX*Gx;-AY*T*Gx;zeros(N*nru,nx)];

%% simulation
Nsteps=200;
% Nsteps=1/deltaT;       % simulation steps
H=2*(Gu'*QN*Gu+RN);
x0=[pi/180*10; 0; 0];        % initial state
xk=x0;
r1=ones(1,Nsteps)*xr(3);
xSim=(1:Nsteps);
ySim1=zeros(1,Nsteps);
ySim2=zeros(1,Nsteps);
ySim3=zeros(1,Nsteps);
ySim4=zeros(1,Nsteps);
fvalSim=zeros(1,Nsteps);
elapsed=zeros(1,Nsteps);
ref=zeros(1,Nsteps);
pasoE=deltaT;              % Euler method integration step
for i=1:Nsteps
    % setpoint change
    if i==201
        xr=[0; 0; 50];
        bx=[xmax;-xmin]-Ax*xr;
        for j=1:N+1
            bX((j-1)*nrx+1:j*nrx,:)=bx;
        end
        b1qp=[bX;bY;bU];
        r2=ones(1,Nsteps)*xr(3);
    elseif i==401
        xr=[0; 0; 15];
        bx=[xmax;-xmin]-Ax*xr;
        for j=1:N+1
           bX((j-1)*nrx+1:j*nrx,:)=bx;
        end
        b1qp=[bX;bY;bU];
        r3=ones(1,Nsteps)*xr(3);
    end

    ref(i)=xr(3);

    % QP constraints and f are updated
    f=(2*(xk-xr)'*Gx'*QN*Gu)';
    tic
    [uF,fval]=quadprog(H,f,Aqp,b1qp+b2qp*(xk-xr));
    elapsed(i)=toc;
    uk=uF(1);
%     xk=A*xk+B*uk;    % linearised model

    % nonlinear model
```

```
    phiC=xk(1);      % previous phi_c
    vk=xk(2);        % previous phi_cDot
    thetaDot=xk(3); % previous thetaDot
    xk(1)=phiC+pasoE*vk;
    xk(2)=vk+pasoE*(d*sin(phiC)-c*sin(phiC)*vk^2-
(2*a+c*cos(phiC))*uk)*(c*cos(phiC)+2*b)^-1;
    xk(3)=thetaDot+pasoE*uk;

    ySim1(i)=xk(1)*180/pi;
    ySim2(i)=xk(2);
    ySim3(i)=xk(3);
    ySim4(i)=uk;
    fvalSim(i)=fval;
end
```

6.4.2 Plotting

```
xmaxRef=ones(length(xmax),Nsteps).*[xmax(1)*180/pi;xmax(2:end)];
umaxRef=ones(length(umax),Nsteps).*umax;

%% steady/ changing reference
plot(xSim,xmaxRef(1,:),xSim,-
xmaxRef(1,:),xSim,ySim1,xSim,zeros(1,Nsteps))
title('Tilt')
xlabel('Simulation cycles')
ylabel('°')
legend('Upper value','Lower value','Tilt','Reference')
grid on
saveas(gcf,'pSim1.png')

plot(xSim,xmaxRef(2,:),xSim,-
xmaxRef(2,:),xSim,ySim2,xSim,zeros(1,Nsteps))
title("Tilt's speed")
xlabel('Simulation cycles')
ylabel('rad·s?¹')
legend('Upper value','Lower value',"Tilt's speed",'Reference')
grid on
saveas(gcf,'pSim2.png')

plot(xSim,xmaxRef(3,:),xSim,-xmaxRef(3,:),xSim,ySim3,xSim,ref)
title("Wheels' speed")
xlabel('Simulation cycles')
ylabel('rad·s?¹')
legend('Upper value','Lower value',"Wheels' speed",'Reference')
grid on
saveas(gcf,'pSim3.png')

plot(xSim,umaxRef(1,:),xSim,-
umaxRef(1,:),xSim,ySim4,xSim,zeros(1,Nsteps))
title('Control action')
xlabel('Simulation cycles')
ylabel('rad·s?²')
legend('Upper value','Lower value','Control action','Reference')
grid on
saveas(gcf,'pSim4.png')

%% linear vs. NL comparison
ySim1NL=ySim1;
```

```
ySim2NL=ySim2;
ySim3NL=ySim3;
ySim4NL=ySim4;

plot(xSim,xmaxRef(1,:),xSim,-
xmaxRef(1,:),xSim,ySim1,xSim,ySim1NL,xSim,zeros(1,Nsteps))
title('Tilt')
xlabel('Simulation cycles')
ylabel('°')
legend('Upper value','Lower value','Linear','Nonlinear','Reference')
grid on
saveas(gcf,'pSim1.png')

plot(xSim,xmaxRef(2,:),xSim,-
xmaxRef(2,:),xSim,ySim2,xSim,ySim2NL,xSim,zeros(1,Nsteps))
title("Tilt's speed")
xlabel('Simulation cycles')
ylabel('rad·s?¹')
legend('Upper value','Lower value','Linear','Nonlinear','Reference')
grid on
saveas(gcf,'pSim2.png')

plot(xSim,xmaxRef(3,:),xSim,-
xmaxRef(3,:),xSim,ySim3,xSim,ySim3NL,xSim,ref)
title("Wheels' speed")
xlabel('Simulation cycles')
ylabel('rad·s?¹')
legend('Upper value','Lower value','Linear','Nonlinear','Reference')
grid on
saveas(gcf,'pSim3.png')

plot(xSim,umaxRef(1,:),xSim,-
umaxRef(1,:),xSim,ySim4,xSim,ySim4NL,xSim,zeros(1,Nsteps))
title('Control action')
xlabel('Simulation cycles')
ylabel('rad·s?²')
legend('Upper value','Lower value','Linear','Nonlinear','Reference')
grid on
saveas(gcf,'pSim4.png')

%% computing time
avg=sum(elapsed)/Nsteps;

plot(xSim,elapsed,xSim,ones(1,Nsteps)*avg)
title("Optimization time")
ylabel('seconds')
xlabel('Simulation cycles')
grid on
```

7 RESULTS

This chapter includes the results obtained throughout the project. Even if the 'results' themselves could be considered only the plots explaining the programme's and code's performance, the configuration and information acquired about Julia and the Jetson Nano take part of the knowledge gained through the development of this whole project.

7.1 PC specifications

Here, the specifications of the PC where some of the experiments were performed are introduced:

Table 7-1. PC characteristics.

Model	Lenovo Y520-15IKBN
Processor	Intel Core i7-7700HQ CPU @ 2.80GHz 2.80GHz
RAM	8,00 GB

These characteristics will have to be considered when performing the experiments, as results obtained in the computer will be compared with the ones obtained in the Nano, which is far less powerful and, as so, higher computation time—in general, worst or slower results—will be obtained when testing the code in it. However, they are also compared to show the reader the full potential of this device and its limitations.

7.2 Julia's programme

In this section, the results presented in the PC and in the Jetson Nano running Julia's programme are presented.

7.2.1 In the PC

To present the results, the Julia's package `plots` will be used. To install it and make it run—as it could be seen in the 'packages' section of the programme—the following commands shall be typed:

```
import Pkg
Pkg.add("Plots")
using plots
```

Plots.jl is not actually a plotting package! Plots.jl is a plotting metapackage: it is an interface over many different plotting libraries. Thus, what Plots.jl is actually doing is interpreting commands and then generating the plots using another plotting library. This plotting library in the background is referred to as the **backend**. The nice thing about this is that this means many different plotting can be used libraries all with the Plots.jl syntax. In our case, **plotly** is the chosen backend, as it permits easy modification and consulting of the given graphs, allowing

us to zoom and acquire points' coordinates easily.

7.2.1.1 Steady reference

Firstly, some tests were performed without changing the reference in the speed of the wheels. The robot had the aim of reaching this reference:

$$x_r = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \begin{matrix} rad \\ rad \cdot s^{-1} \\ rad \cdot s^{-1} \end{matrix}$$

With a beginning state of:

$$x_0 = \begin{bmatrix} 10 \cdot \frac{\pi}{180} \\ 0 \\ 0 \end{bmatrix} \begin{matrix} rad \\ rad \cdot s^{-1} \\ rad \cdot s^{-1} \end{matrix}$$

i.e., 10 degrees far from the vertical equilibrium point.

In the following experiment, the system was driven from its starting state of x_0 to the reference setpoint x_r, which is, precisely, the upper equilibrium point. These were the resulting graphs:

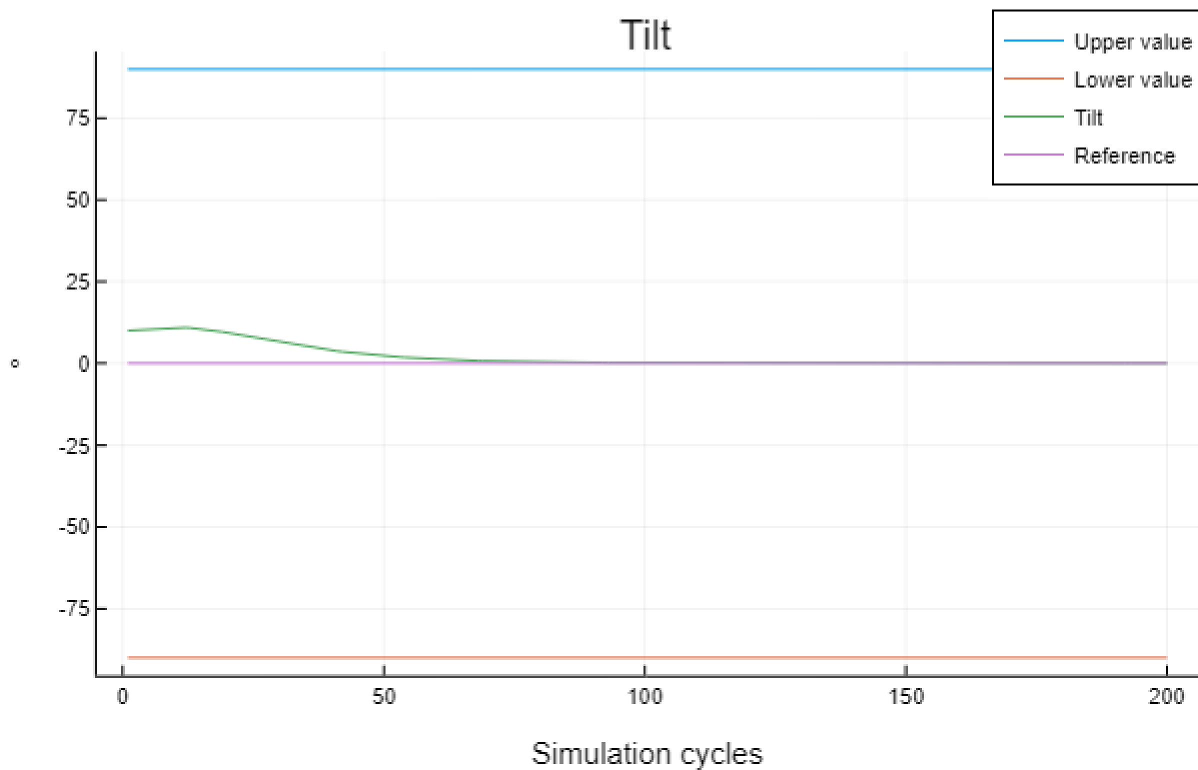


Figure 7-1. Tilt (steady reference).

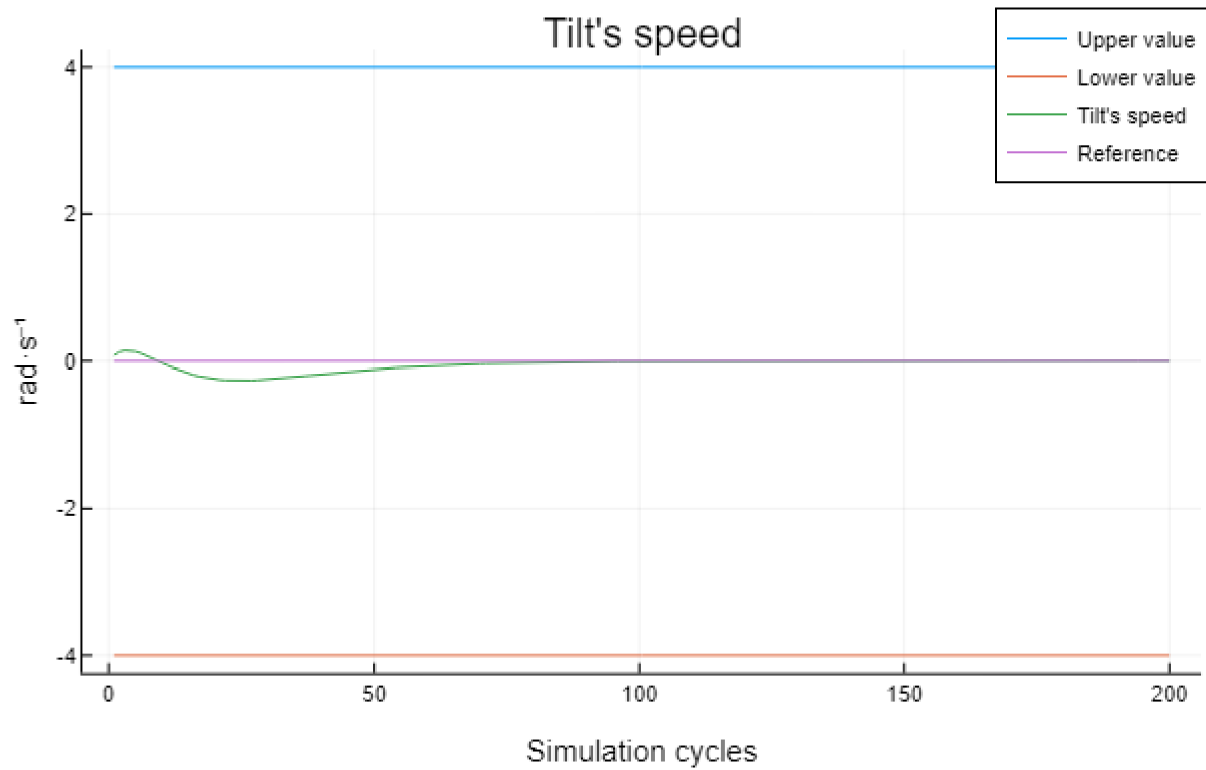


Figure 7-2.Tilt's speed (steady reference).

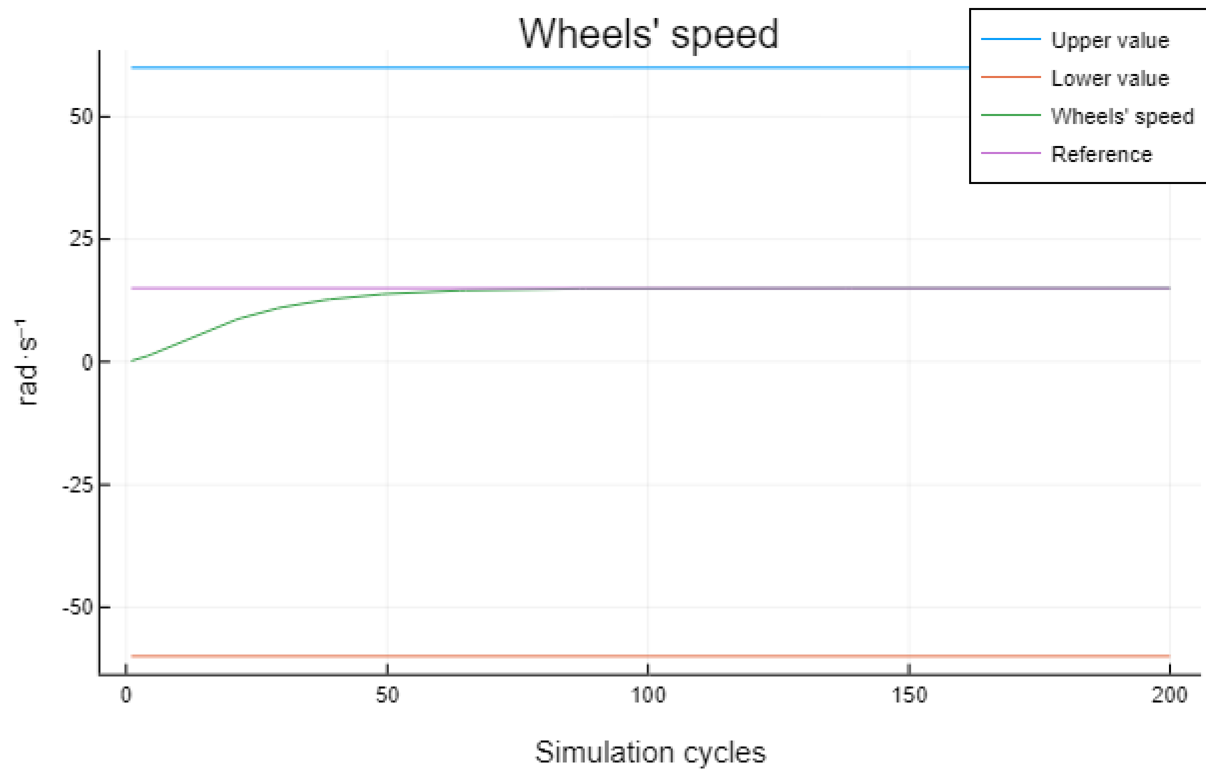


Figure 7-3. Wheels' speed (steady reference).

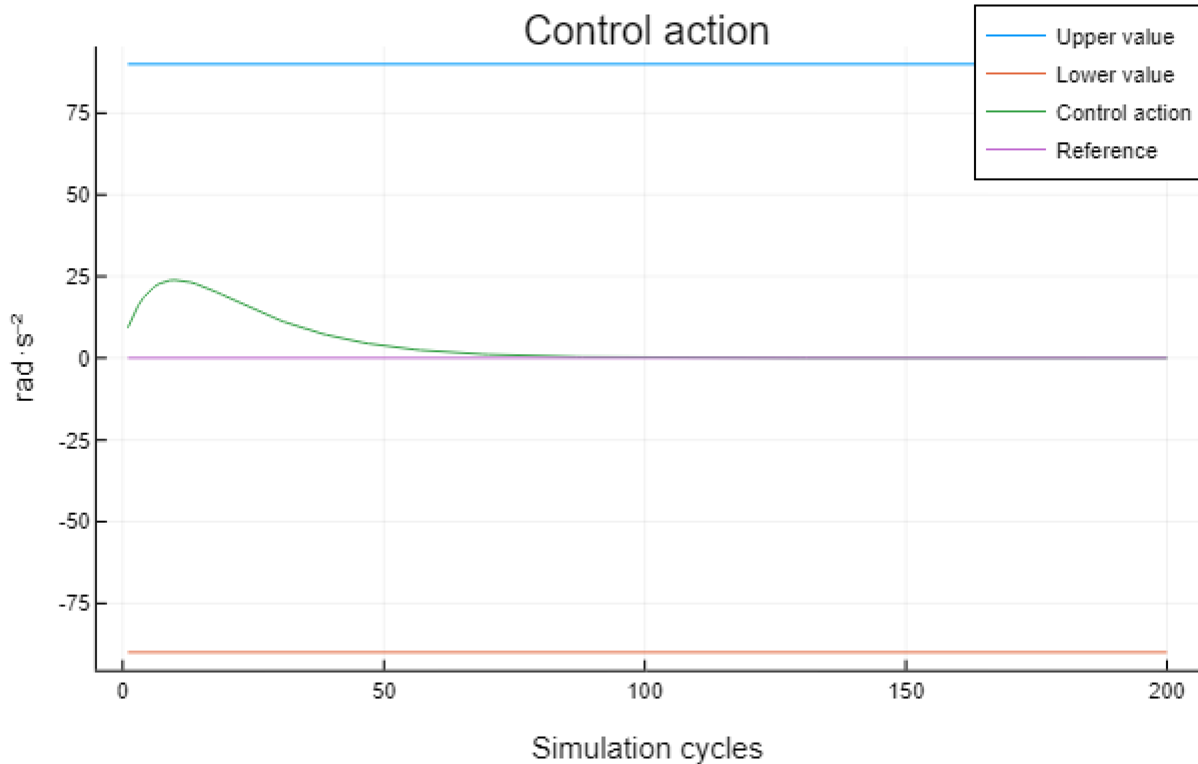


Figure 7-4. Control action (steady reference).

These plots show how the tilt, its speed and the wheels' velocity behave when the system is driven from its starting point to the reference. It can be seen that all the references are reached without exceeding the constraints, which are represented above and below the corresponding curve in blue and orange.

In addition, the computation time of solving the QP problem mas measured, and below a figure showing how it does change throughout the experiment is inserted:

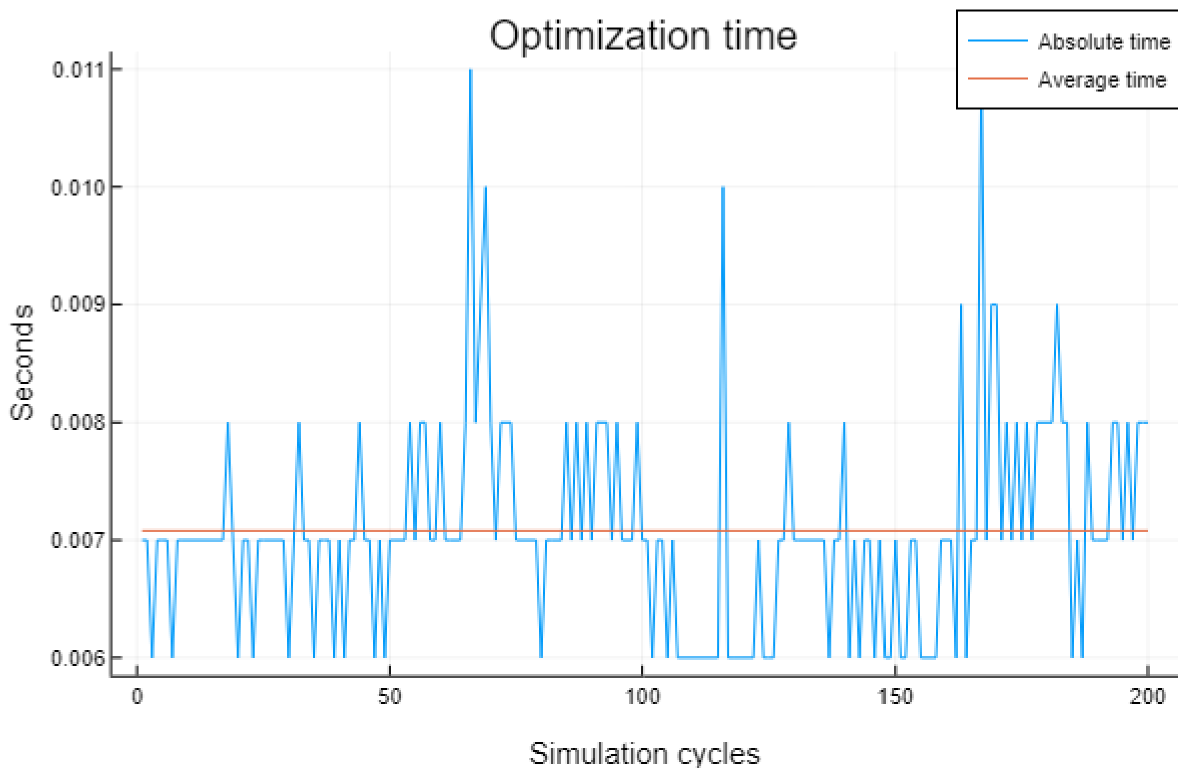


Figure 7-5. Julia's optimization time with steady reference.

With a measured average time of **0.00708 seconds**.

If the execution time of two or more consecutive runs of `MPC()` is measured, it can be appreciated that Julia's first slower iteration characteristic is true. Please, observe this screenshot taken from three consecutive executions of the `MPC()` function:

```
julia> @time MPC()
0.006670 seconds of average in solving the QP
  1.634533 seconds (769.24 k allocations: 51.738 MiB, 4.06% gc time)

julia> @time MPC()
0.006445 seconds of average in solving the QP
  1.427961 seconds (693.01 k allocations: 48.048 MiB)

julia> @time MPC()
0.006460 seconds of average in solving the QP
  1.441873 seconds (693.01 k allocations: 48.048 MiB)
```

Figure 7-6. Julia's compiler behaviour.

From second iteration on, the time the programme spends doing calculations shall be similar to the second, but always inferior to the first one. In addition, please note that the number of allocations in memory is reduced after performing the first iteration and remains steady for the remaining ones. Under the average solving time, the time of performing executing MPC wholly is shown for reference.

In the following experiment, while maintaining the reference of upper equilibrium point, the starting state was chosen:

$$x_0 = \begin{bmatrix} 45 \\ 0 \\ 0 \end{bmatrix} \begin{matrix} rad \\ rad \cdot s^{-1} \\ rad \cdot s^{-1} \end{matrix}$$

In case a more distant initial state is chosen, more abrupt results are obtained. Let us show the detailed view of these figures:

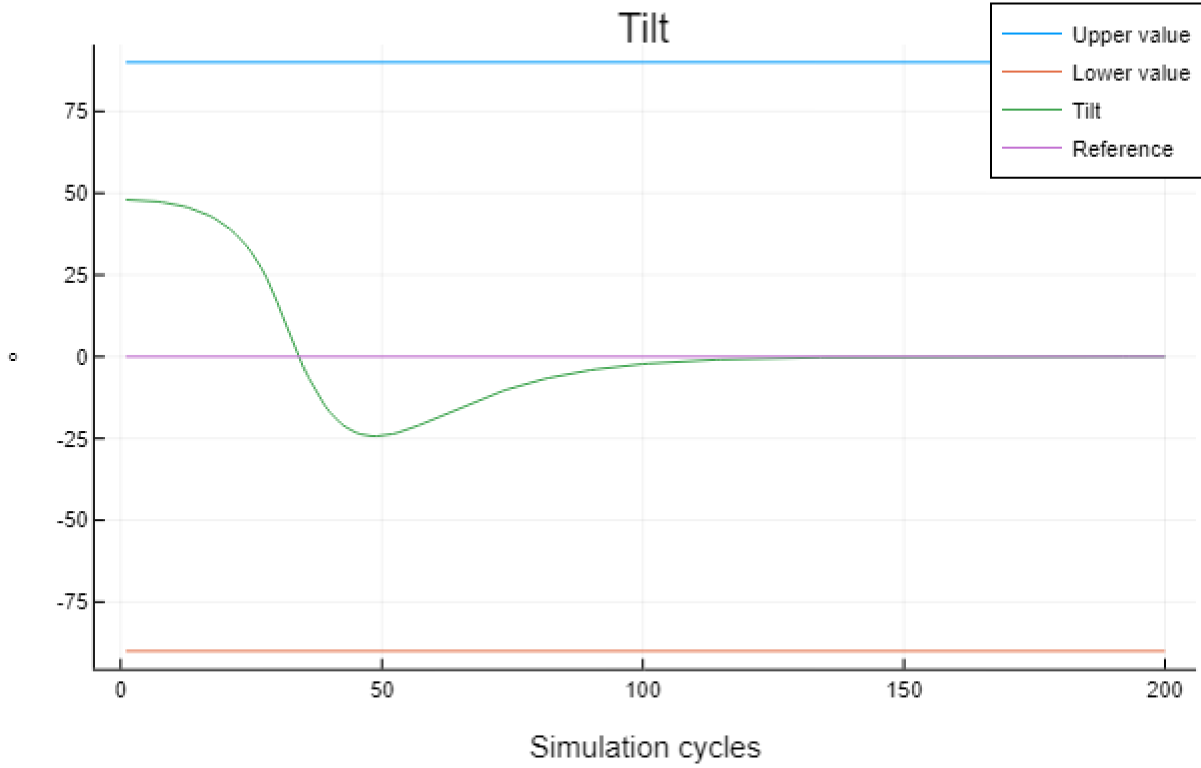


Figure 7-7. Tilt (distant starting point).

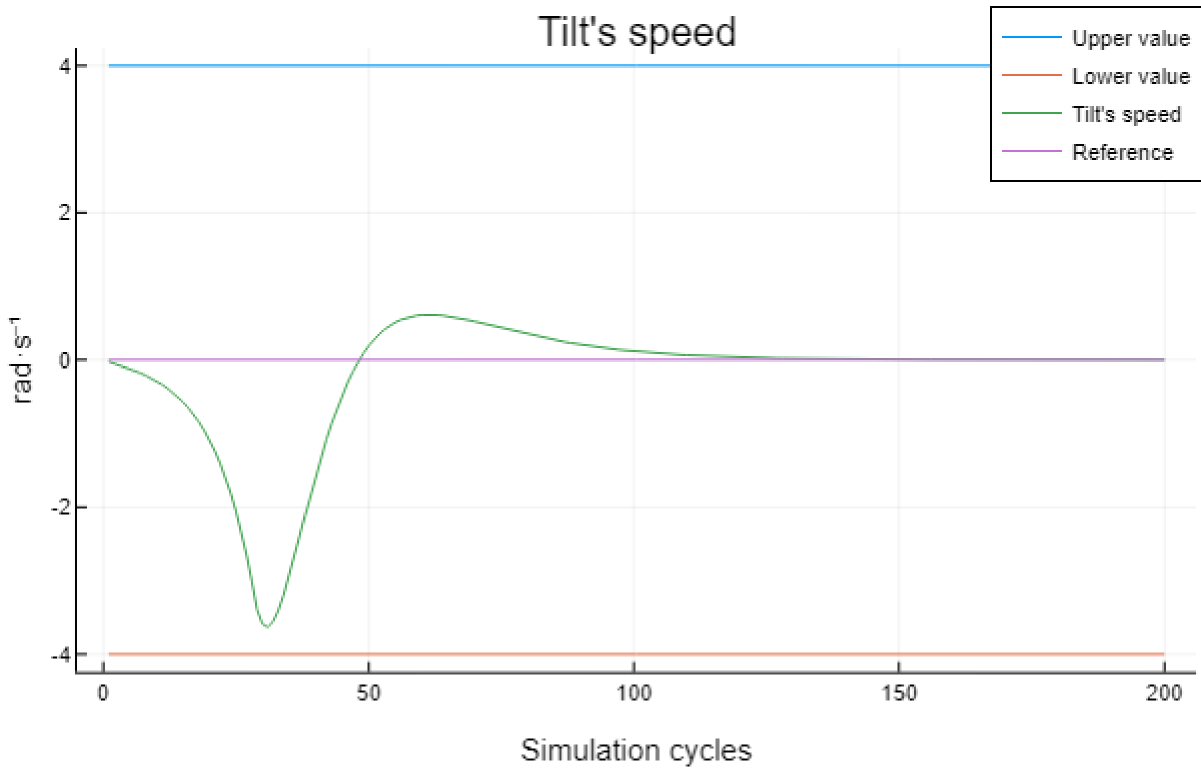


Figure 7-8. Tilt's speed (distant starting point).

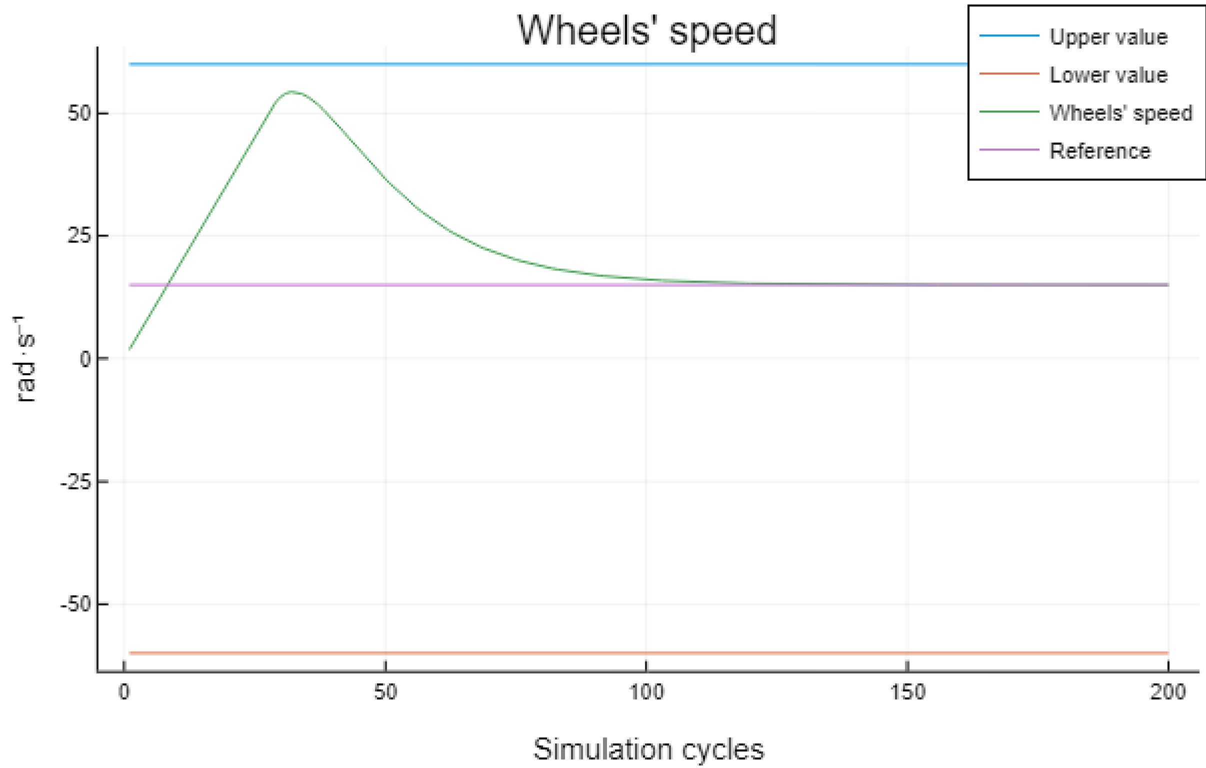


Figure 7-9. Wheels' speed (distant starting point).

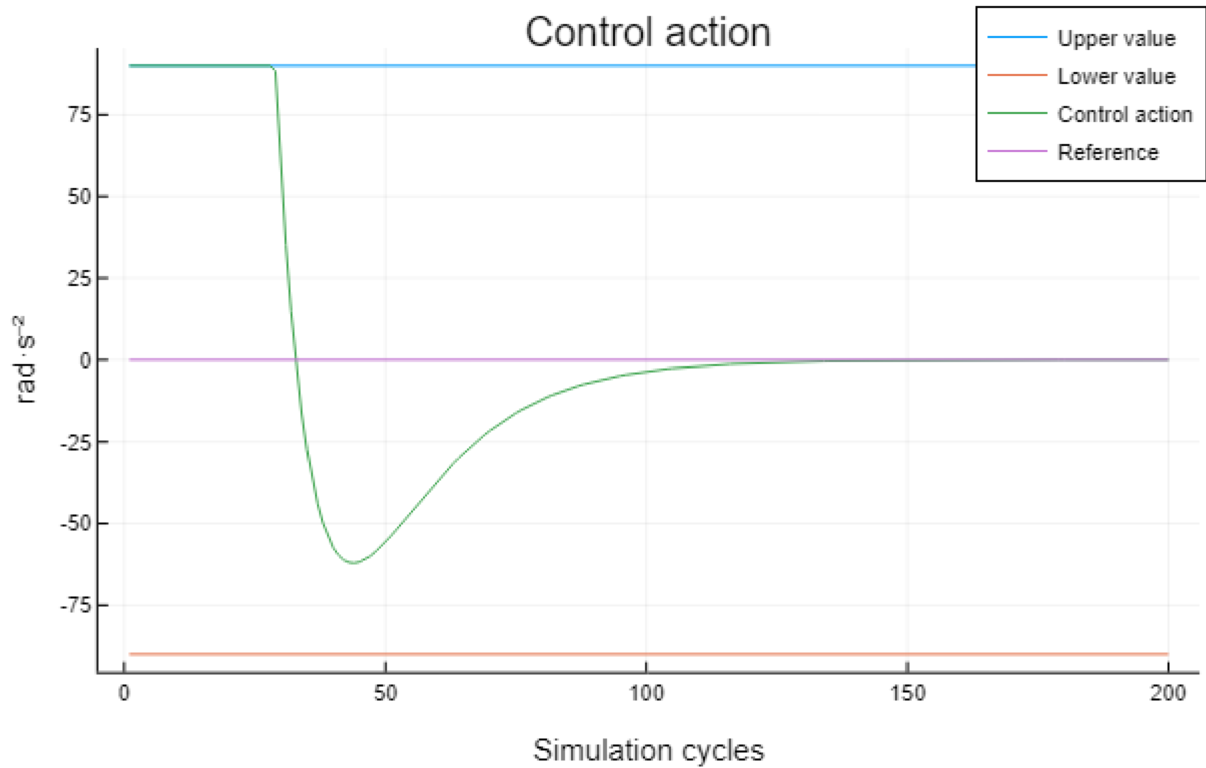


Figure 7-10. Control action (distant starting point).

Even so, it can be observed that constraints are neither surpassed. Please, pay attention to the control action’s plot. It can be beheld that its constraint is reached but not exceeded.

It has been now demonstrated that even with a relatively distant from the reference initial state, the setpoint is reached without overpassing the constraints.

7.2.1.2 Changing reference

On the contrary, this subsection shows how the controller behaves when adding changes in the references.

Beginning from a reference state of:

$$x_r = \begin{bmatrix} 0 \\ 0 \\ 45 \end{bmatrix} \begin{matrix} rad \\ rad \cdot s^{-1} \\ rad \cdot s^{-1} \end{matrix}$$

And an initial state of:

$$x_0 = \begin{bmatrix} 45 * \frac{\pi}{180} \\ 0 \\ 0 \end{bmatrix} \begin{matrix} rad \\ rad \cdot s^{-1} \\ rad \cdot s^{-1} \end{matrix}$$

e.g., 45° separated from the horizontal., two setpoint changes have been implemented:

- The first one, at the **200th** simulation cycle, where the setpoint of the wheels' speed moved to **50** $\frac{\boldsymbol{rad}}{\boldsymbol{s}}$:

$$x_r = \begin{bmatrix} 0 \\ 0 \\ 50 \end{bmatrix} \begin{matrix} rad \\ rad \cdot s^{-1} \\ rad \cdot s^{-1} \end{matrix}$$

- The second and last one, done at the **400th** simulation cycle, where the setpoint changed to **15** $\frac{\boldsymbol{rad}}{\boldsymbol{s}}$:

$$x_r = \begin{bmatrix} 0 \\ 0 \\ 15 \end{bmatrix} \begin{matrix} rad \\ rad \cdot s^{-1} \\ rad \cdot s^{-1} \end{matrix}$$

It has been waited **200** cycles to let the model reach the steady state, every time the reference changed plus the first time to reach the original reference, thus adding up to **600** simulation cycles.

Here, figures of the obtained results are shown to appreciate the detail:

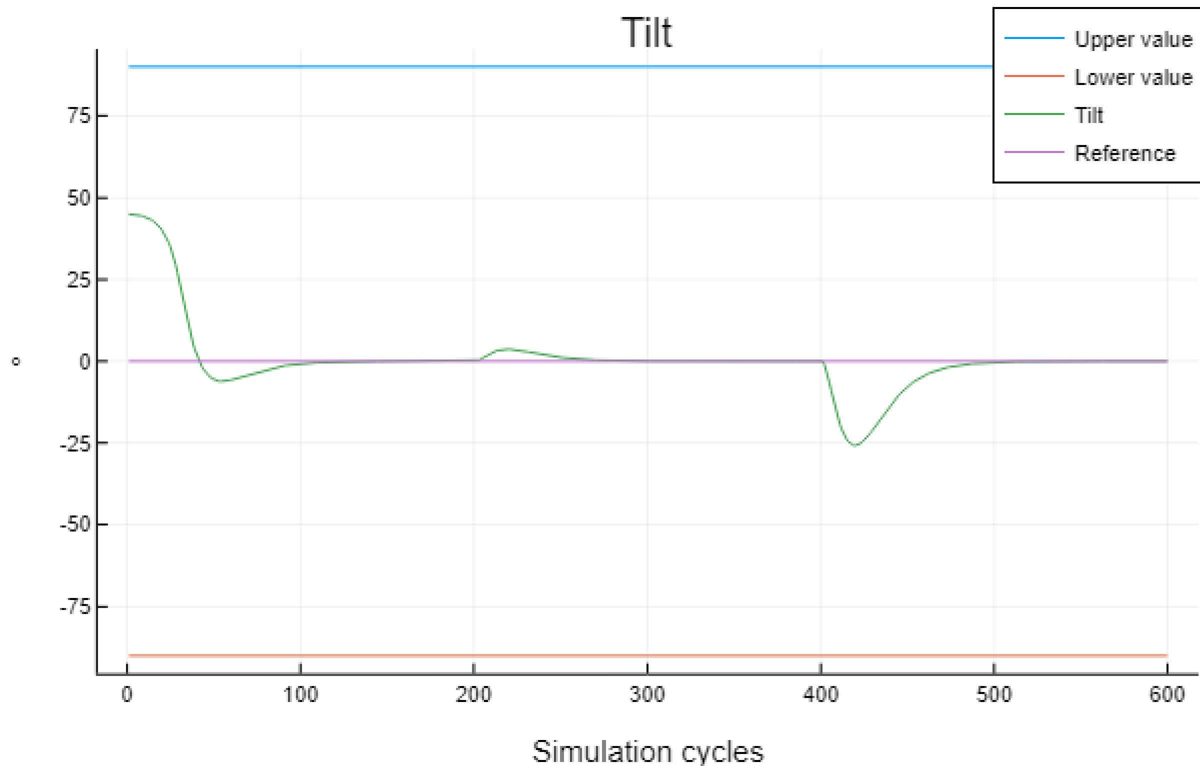


Figure 7-11. Tilt (changing reference).

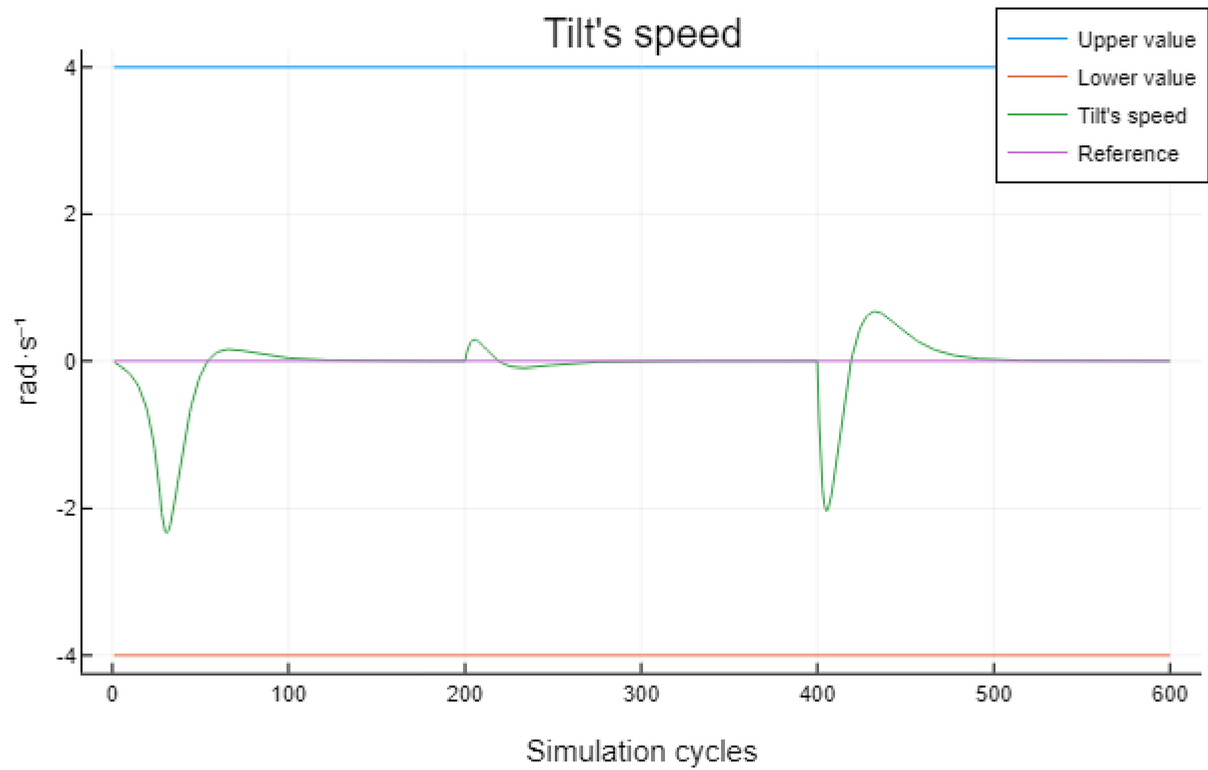


Figure 7-12. Tilt's speed (changing reference).

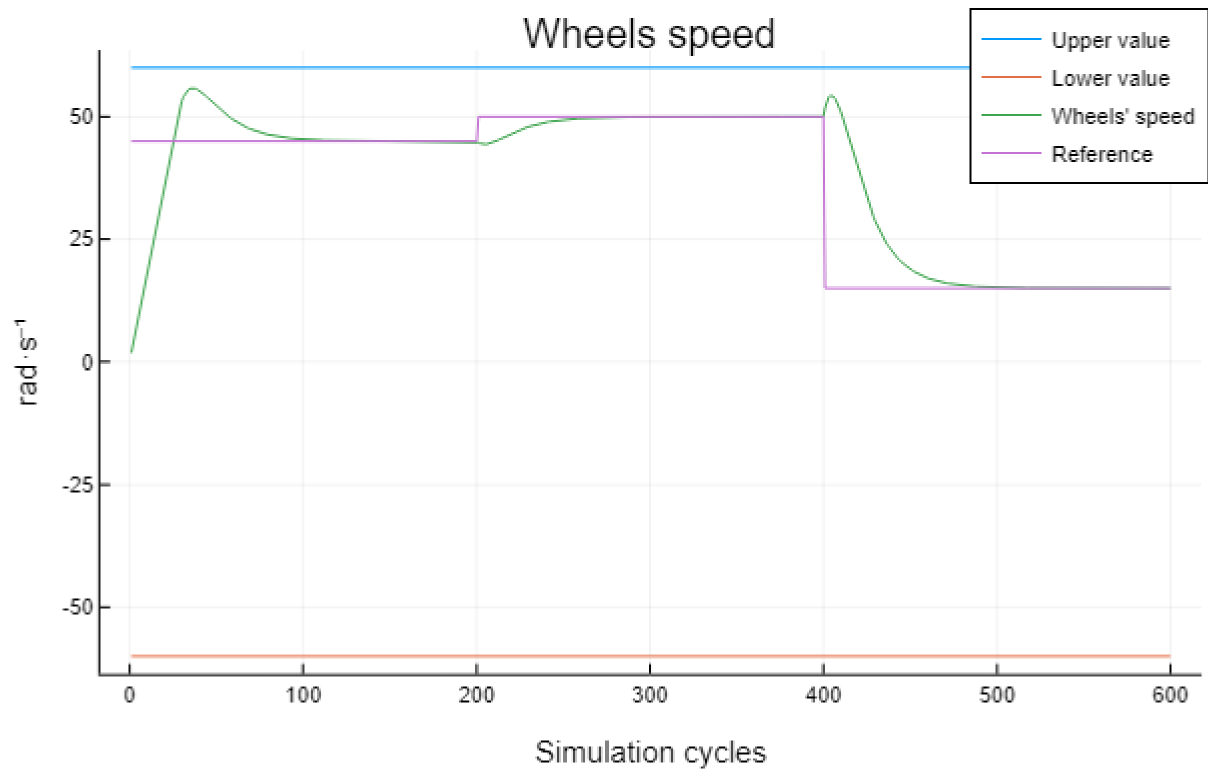


Figure 7-13. Wheels' speed (changing reference).

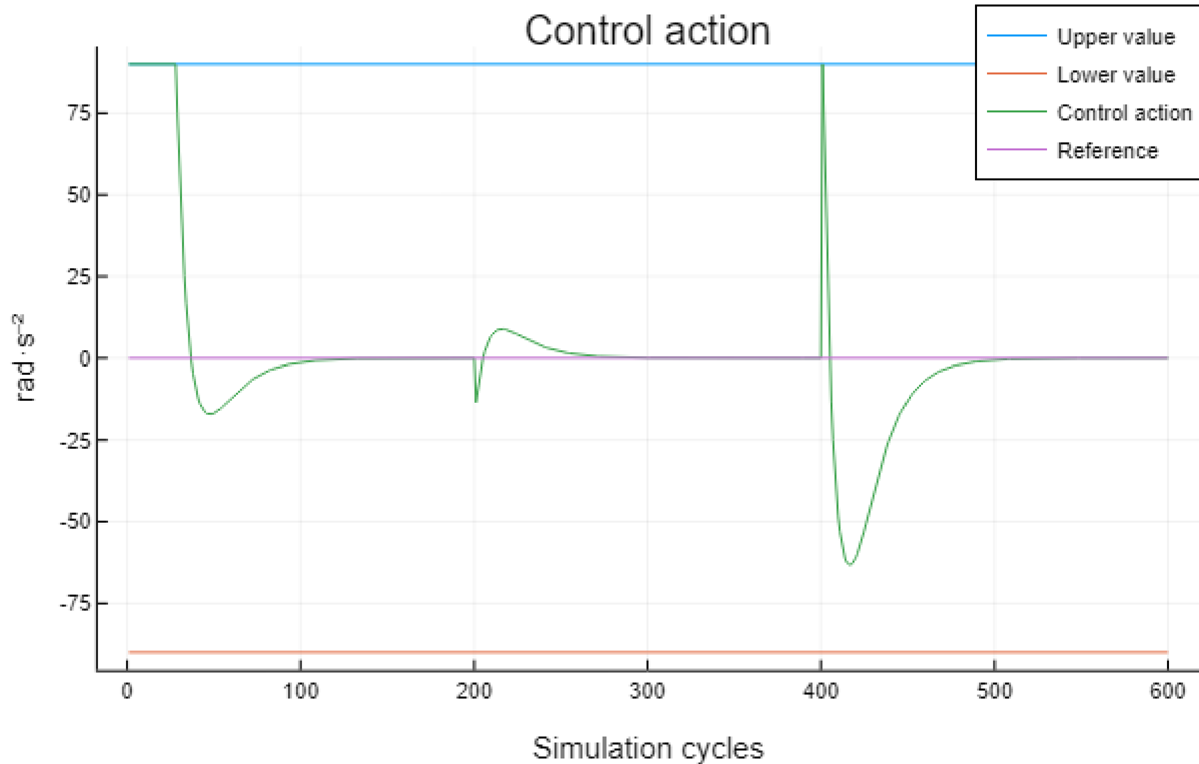


Figure 7-14. Control action (changing reference).

In the plots above, the blue and orange lines are the upper and lower constraints, respectively. In addition, the reference is shown in purple.

From the pictures, it can be observed that not only the reference is reached every time it changes, but also that the tilt and its speed remain zero. In addition, it can be beheld that none of the components of the state surpass their limits, meaning that the solver achieved to solve the QP problem each time.

Let us recall that a too sharp change in the wheels' speed or any of the other components of the reference state may lead to a non-feasible problem.

In addition, the solving time of the QP problem was measured and these results were obtained:

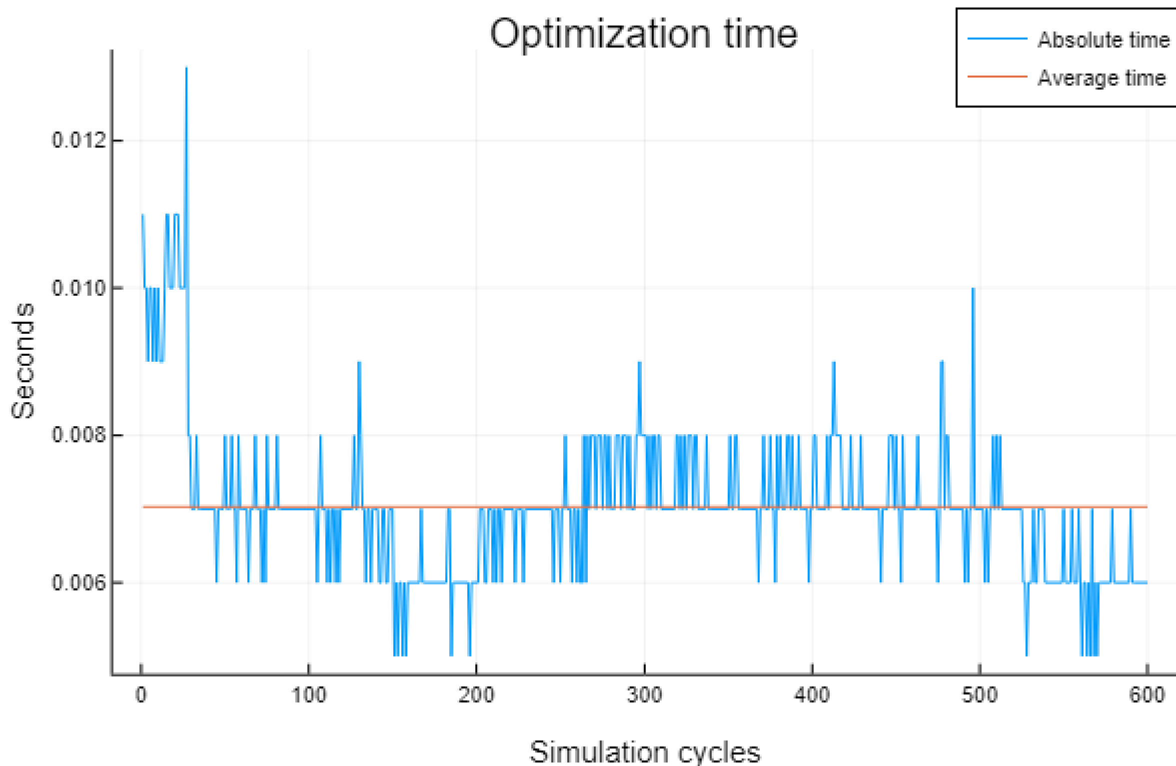


Figure 7-15. Optimization time.

With a measured average time of **0.00703 seconds**, very close to the result obtained with the steady reference.

```
julia> @time MPC()
0.006898 seconds of average in solving the QP
  4.828092 seconds (2.16 M allocations: 148.142 MiB, 1.44% gc time)

julia> @time MPC()
0.007338 seconds of average in solving the QP
  5.098910 seconds (2.09 M allocations: 144.497 MiB, 1.36% gc time)
```

Figure 7-16. Solving time with changing setpoints.

7.2.1.3 Using linearised model

Results using the linearised model were very similar, as it was expected. However, the linearised model seems to give a smoother response. The same experiments performed in the previous subsection were repeated to obtain these plots. These are the computation time when performing two consecutive instances of the MPC function:

```
julia> @time MPC()
0.006685 seconds of average in solving the QP
  7.205043 seconds (3.74 M allocations: 228.897 MiB, 1.60% gc time)

julia> @time MPC()
0.007025 seconds of average in solving the QP
  4.962195 seconds (1.97 M allocations: 142.349 MiB, 1.35% gc time)
```

Figure 7-17. Solving time (linearised model).

Finally, some detailed views of the plots comparing both linear (in green) and nonlinear (in purple) responses are shown below. Reference is shown in ochre:

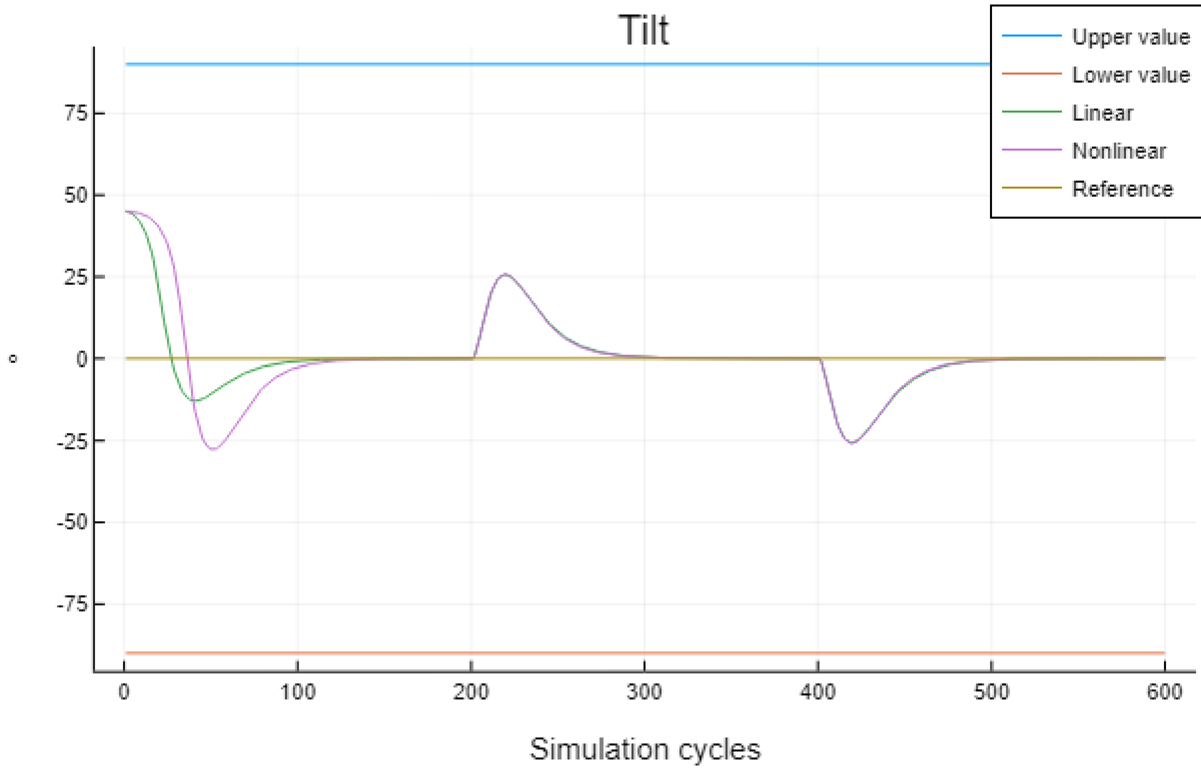


Figure 7-18. Linear vs. nonlinear tilt.

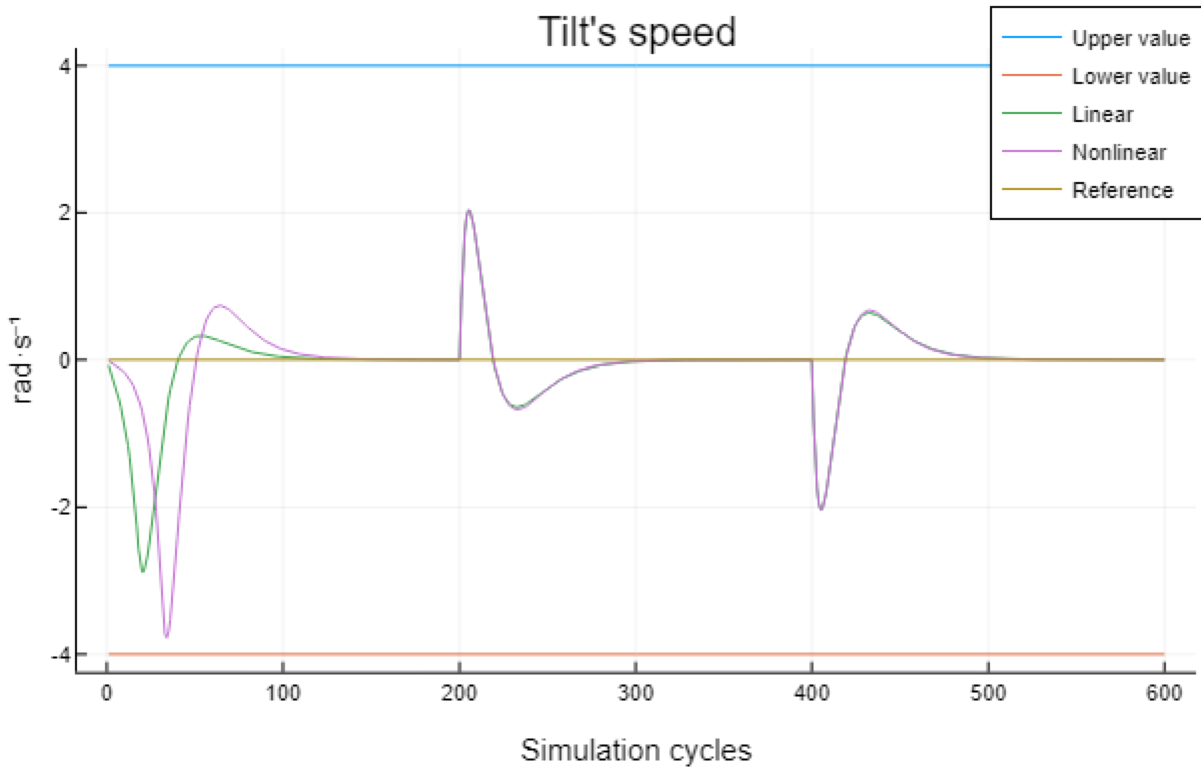


Figure 7-19 Linear vs. nonlinear tilt's speed.

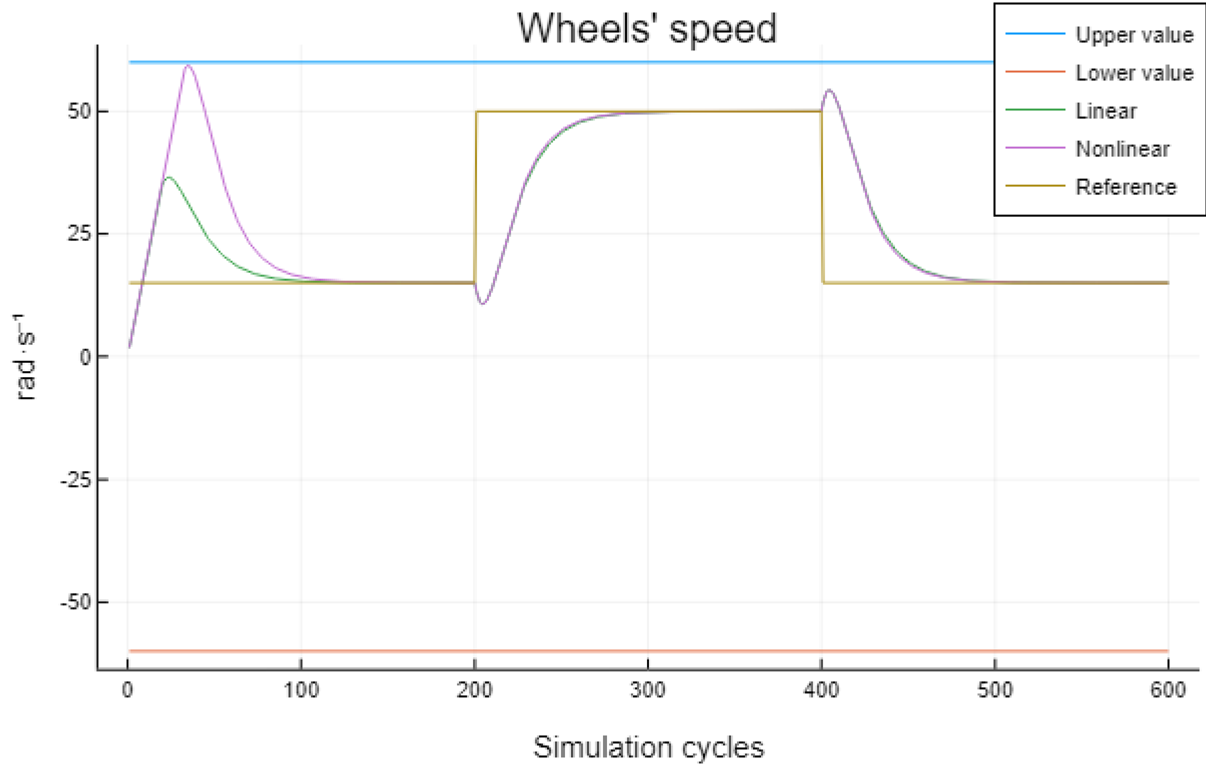


Figure 7-20 Linear vs. nonlinear wheels' speed.

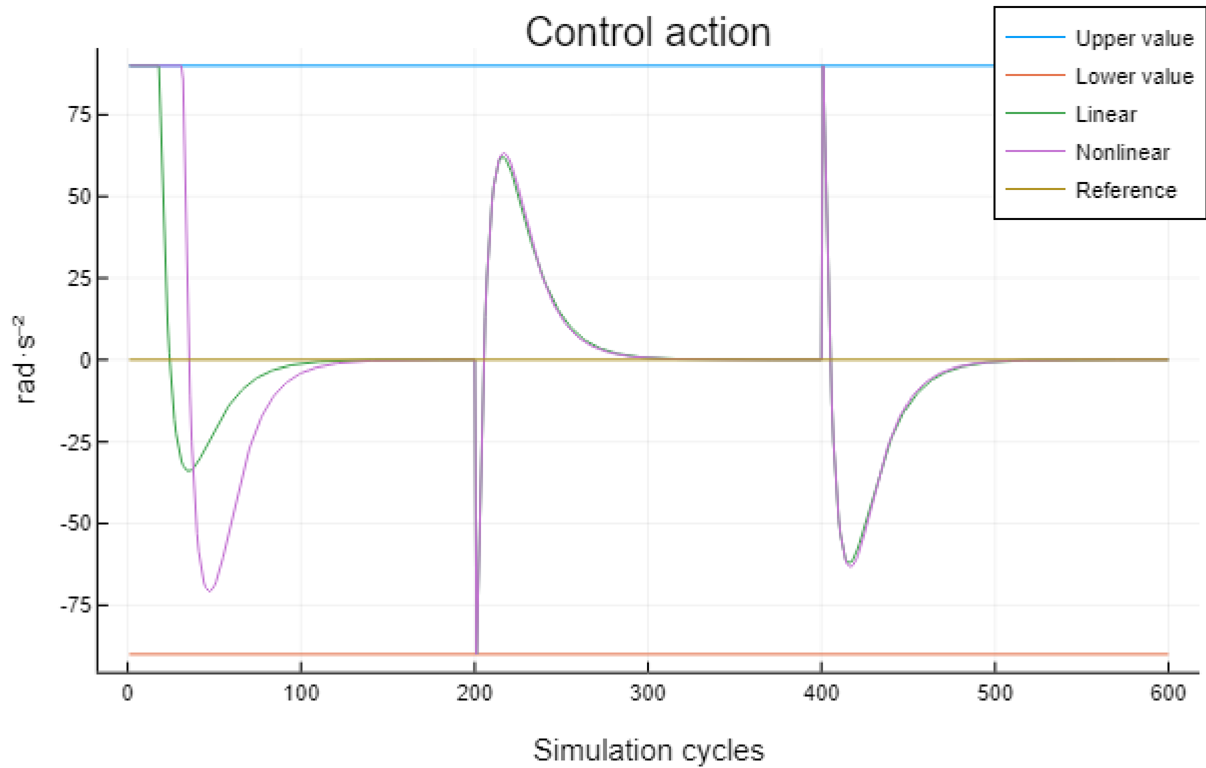


Figure 7-21 Linear vs. nonlinear control action.

As stated before, it can be appreciated that the linear response presents smoother results.

7.2.2 In Jetson Nano

The same tests performed in the PC were done in the Jetson Nano, obtaining the results that follow.

Appreciate that the average computation time in the Jetson Nano is increased in the order of two to three times that of the one obtained in the author's computer, but being faster than the 20 milliseconds obtained in the previous project, albeit suboptimal and hence, opening a new research direction to continue developing the following robots.

On the other hand, it will be noticed that the reference tracking results are very similar to the ones obtained on the PC. However, pay special attention to the performance when testing the models' behaviour when starting from a remote state; it can be observed that the response is sharper, and saturation is reached a larger number of simulation cycles.

7.2.2.1 Steady reference

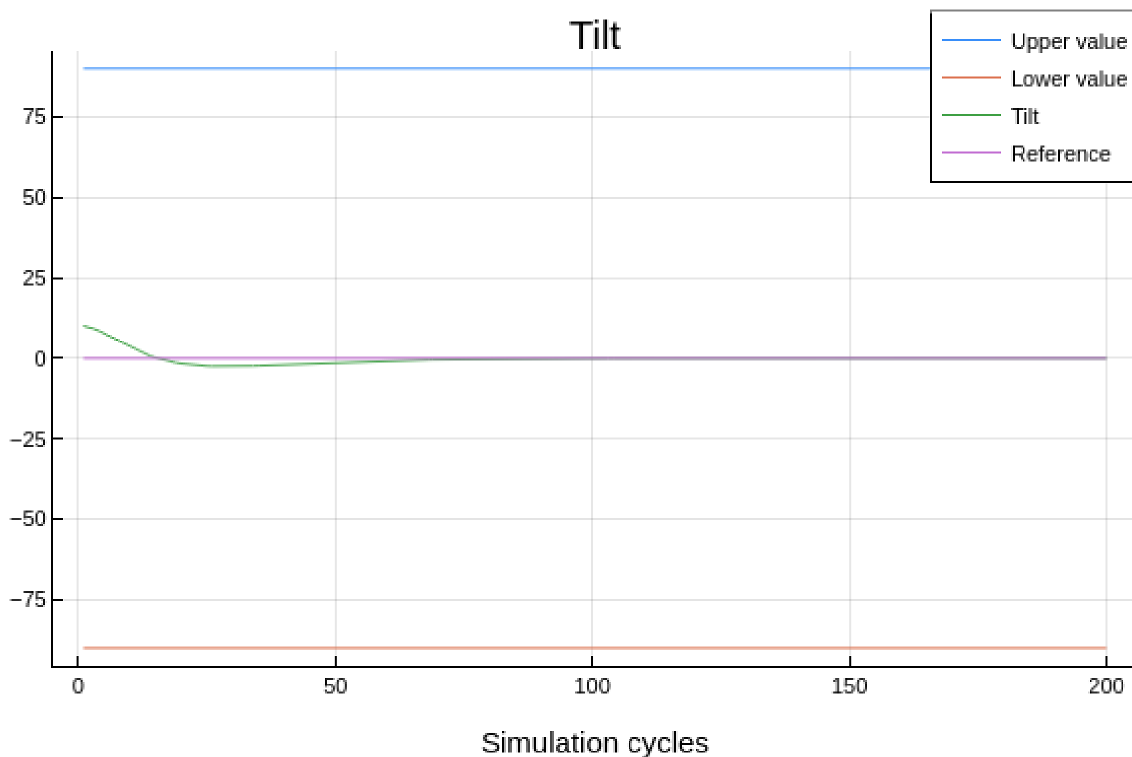


Figure 7-22. Tilt (steady reference).

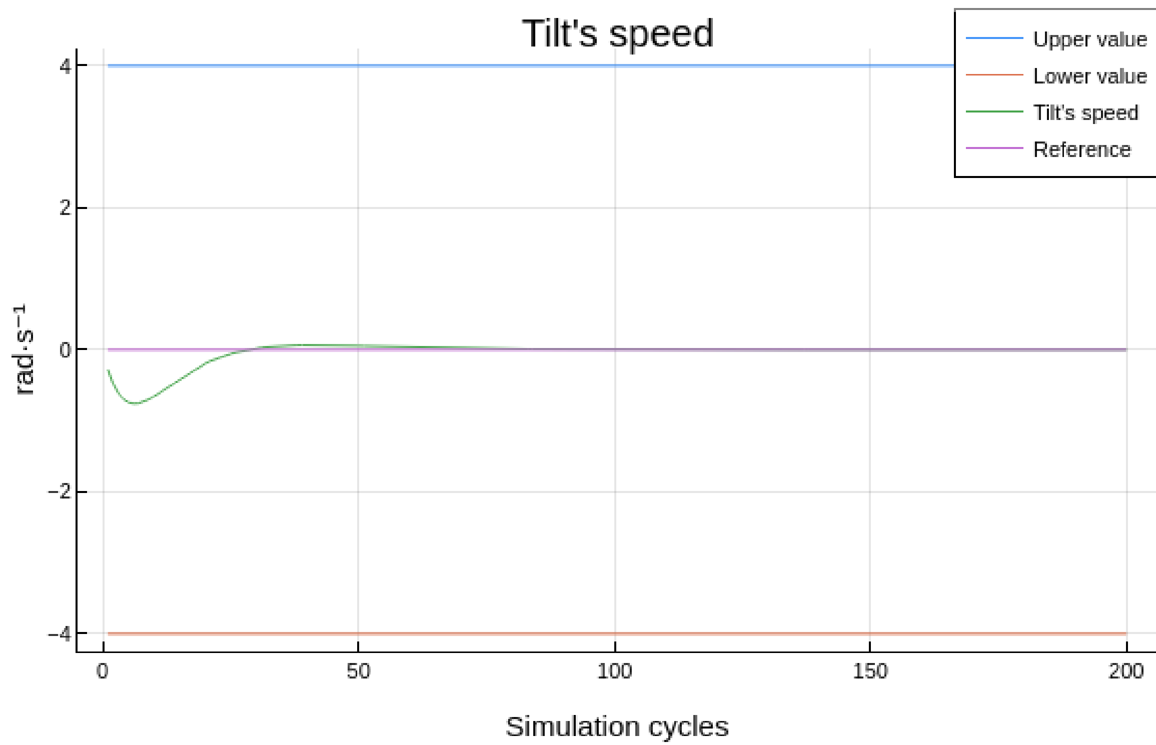


Figure 7-23. Tilt's speed (steady reference).

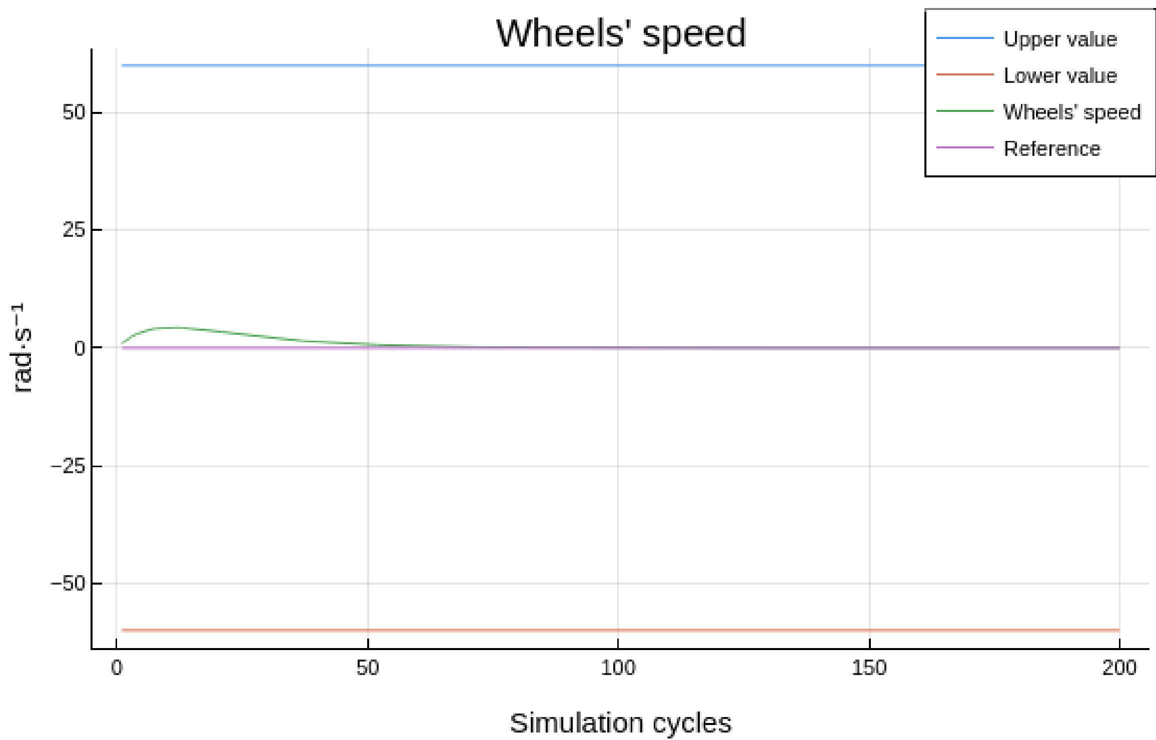


Figure 7-24. Wheels' speed (steady reference).

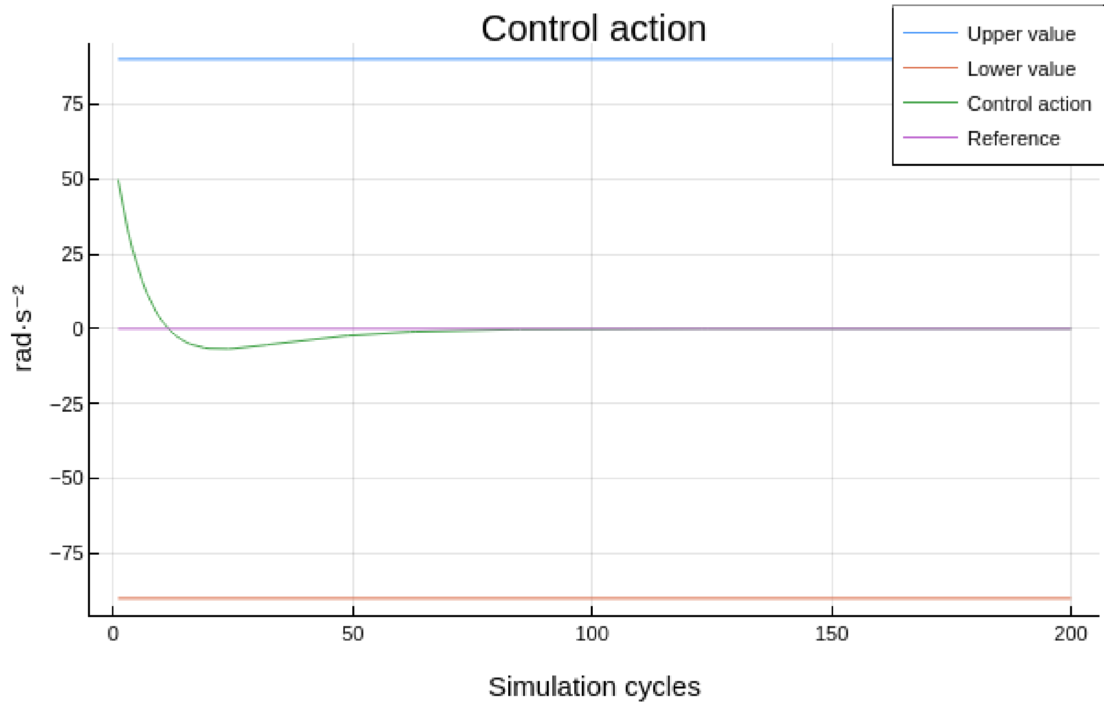


Figure 7-25. Control action (steady reference).

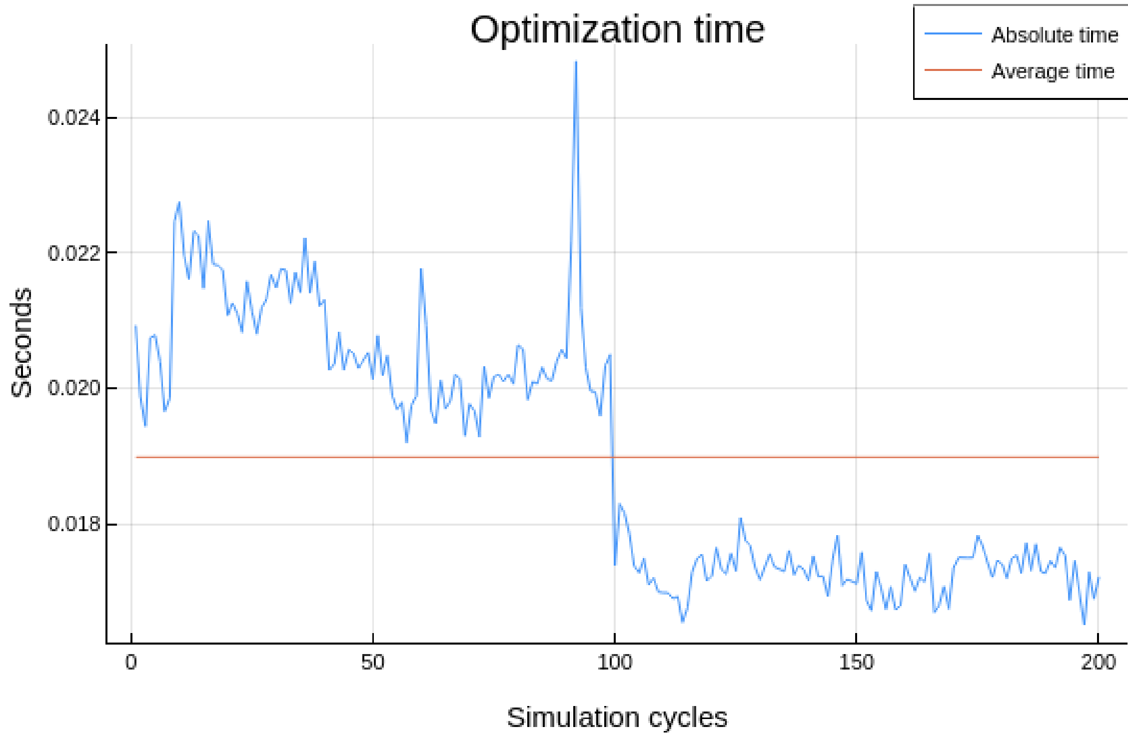


Figure 7-26. Julia's optimization time with steady reference (Jetson Nano).

```
julia> @time MPC()
0.018886 seconds of average in solving the QP
  4.515868 seconds (753.64 k allocations: 51.055 MiB)

julia> @time MPC()
0.018983 seconds of average in solving the QP
  4.895943 seconds (677.40 k allocations: 47.365 MiB, 3.07% gc time)
```

Figure 7-27. Julia's compiler behaviour (Jetson Nano).

When testing the programme with far-flung starting state, these results were obtained:

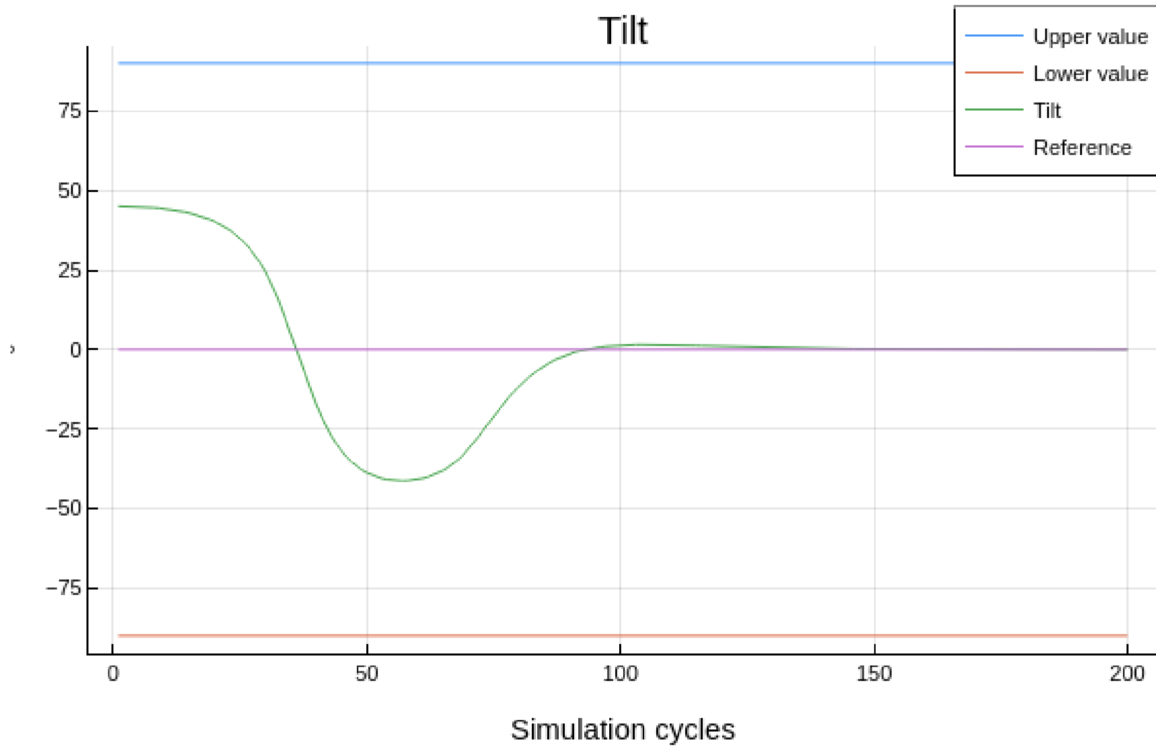


Figure 7-28. Tilt (distant starting point).

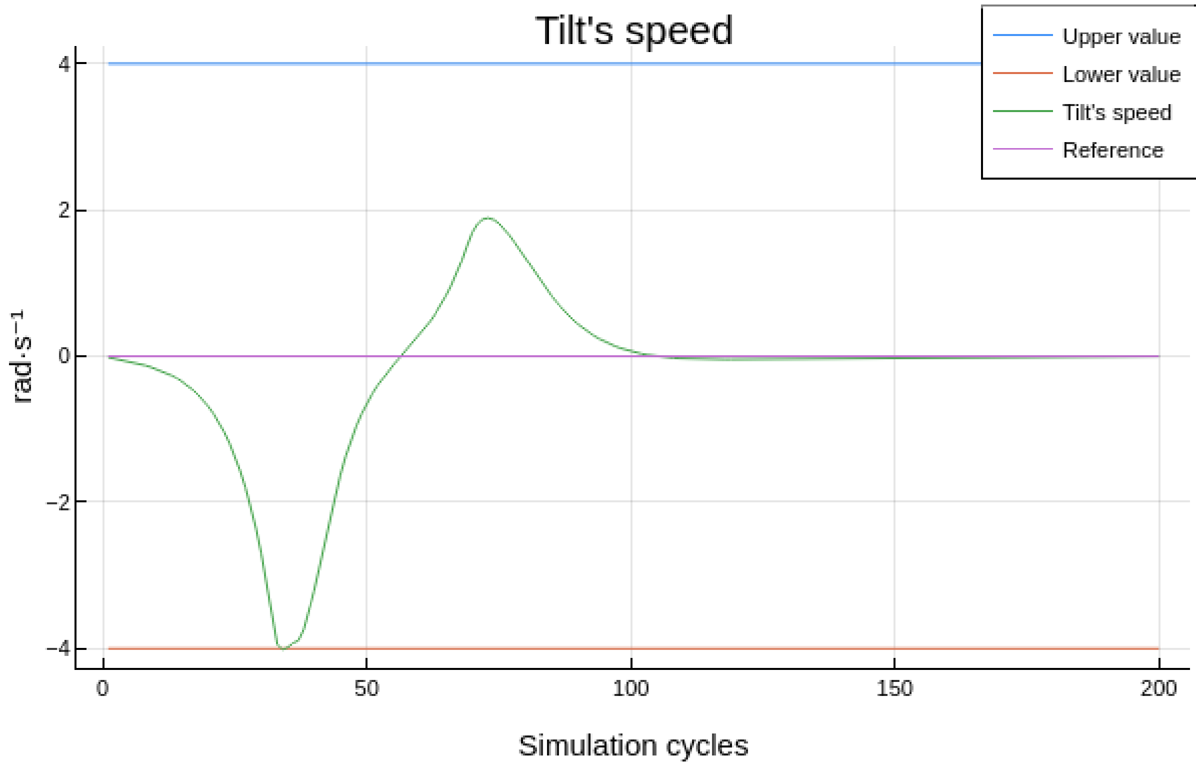


Figure 7-29. Tilt's speed (distant starting point).

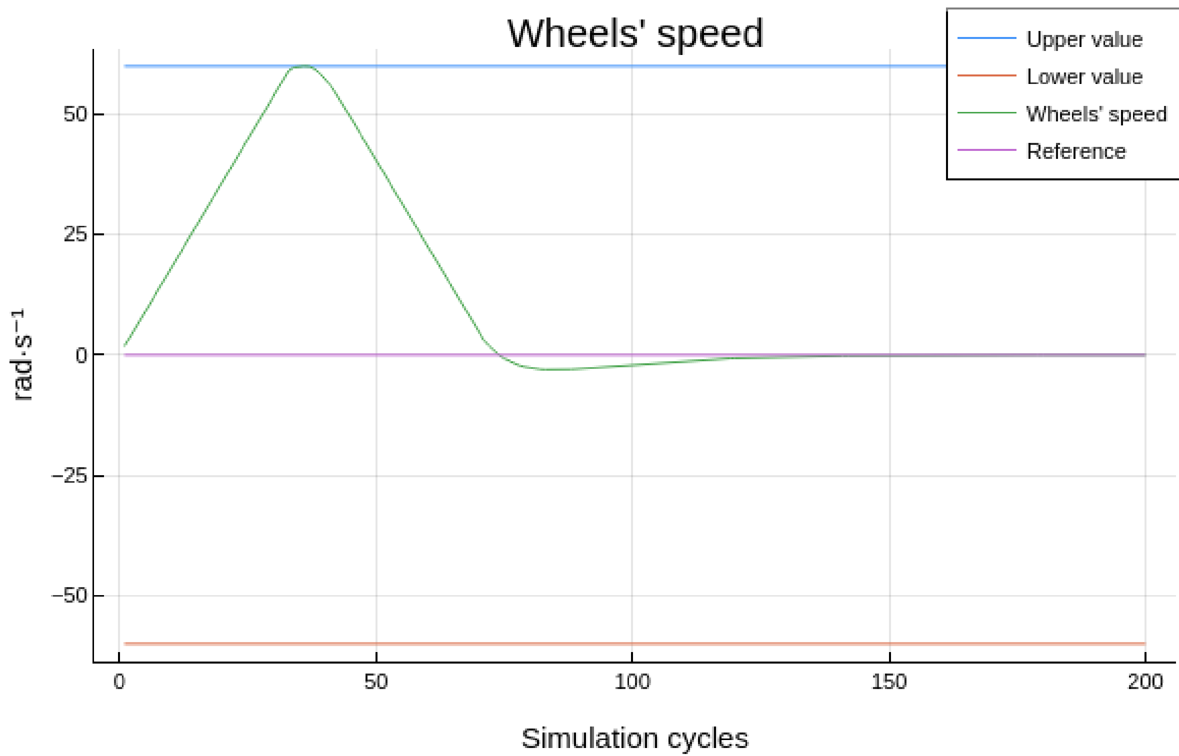


Figure 7-30. Wheels' speed (distant starting point).

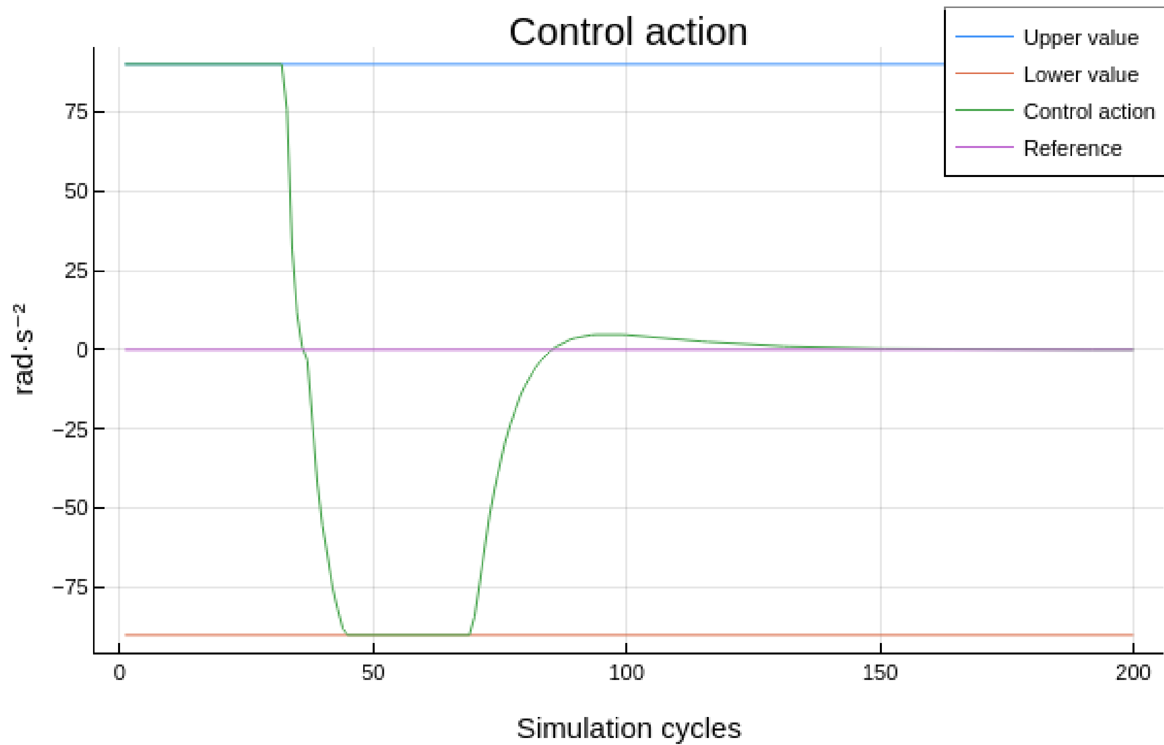


Figure 7-31. Control action (distant starting point).

It can be beheld that saturation is reached in every figure—specially in the case of the control action, something that should be regarded in the future—but the tilt's one.

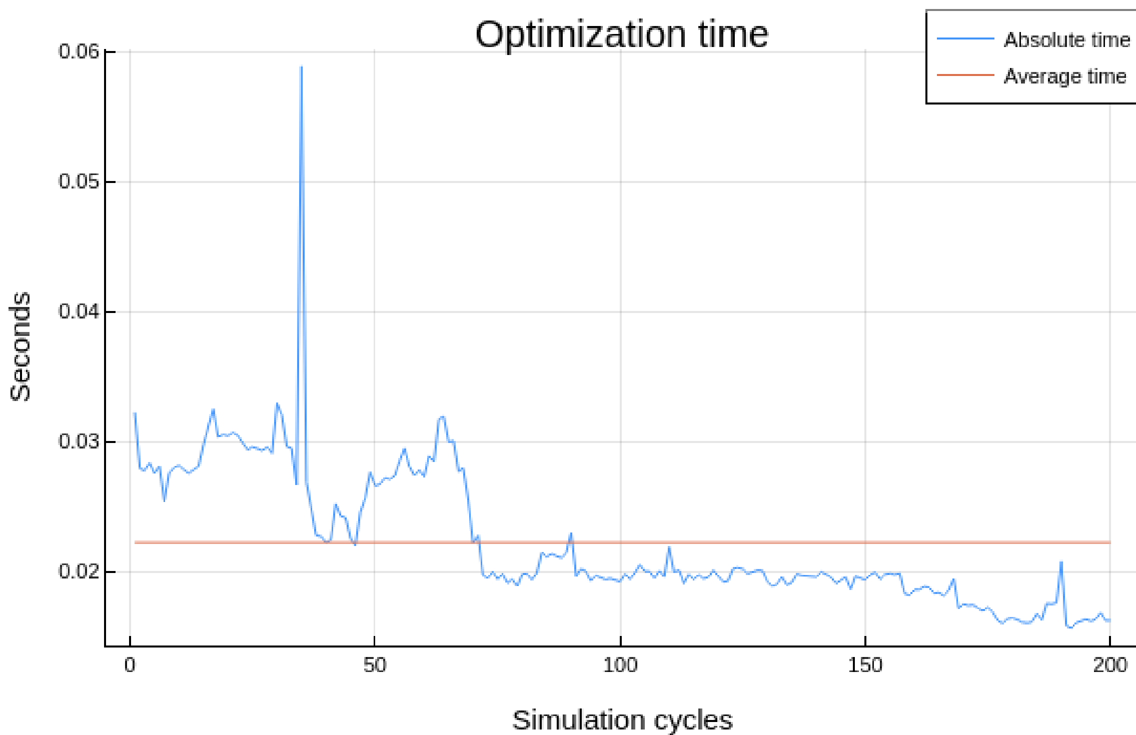


Figure 7-32. Optimization time (distant starting point).

7.2.2.2 Changing reference

The experiments when changing the wheels' speed reference are shown below:

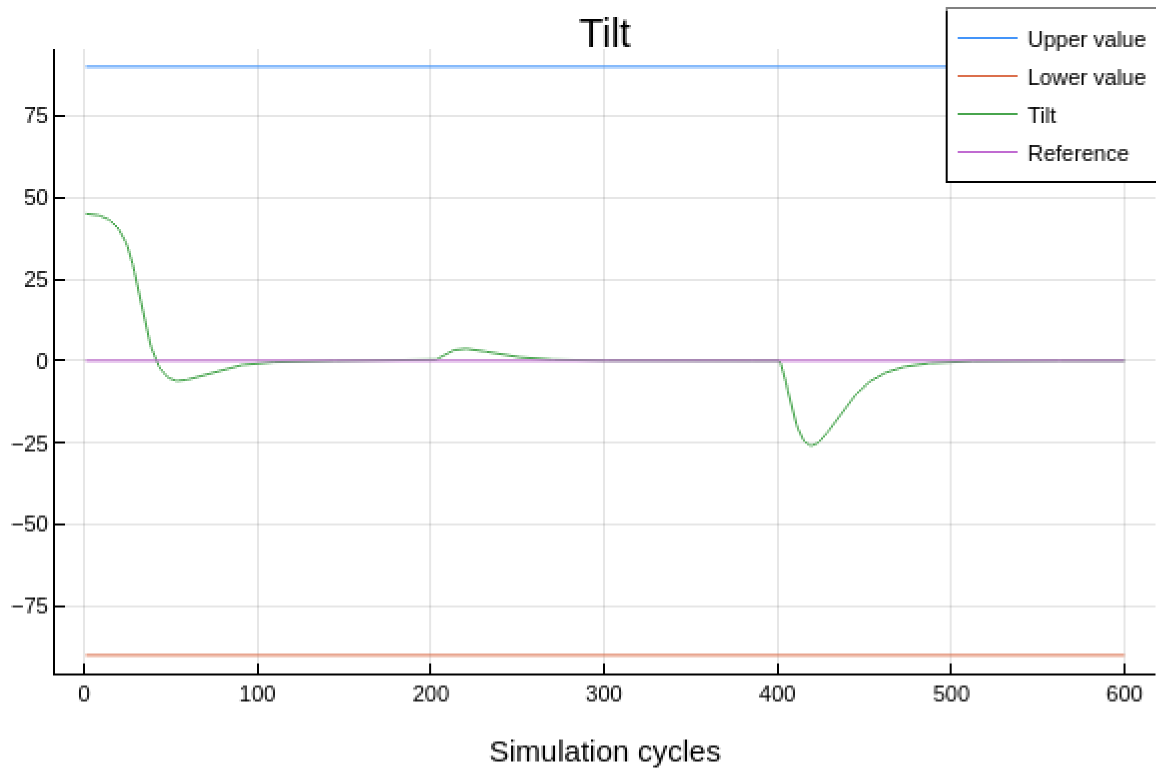


Figure 7-33. Tilt (changing reference).

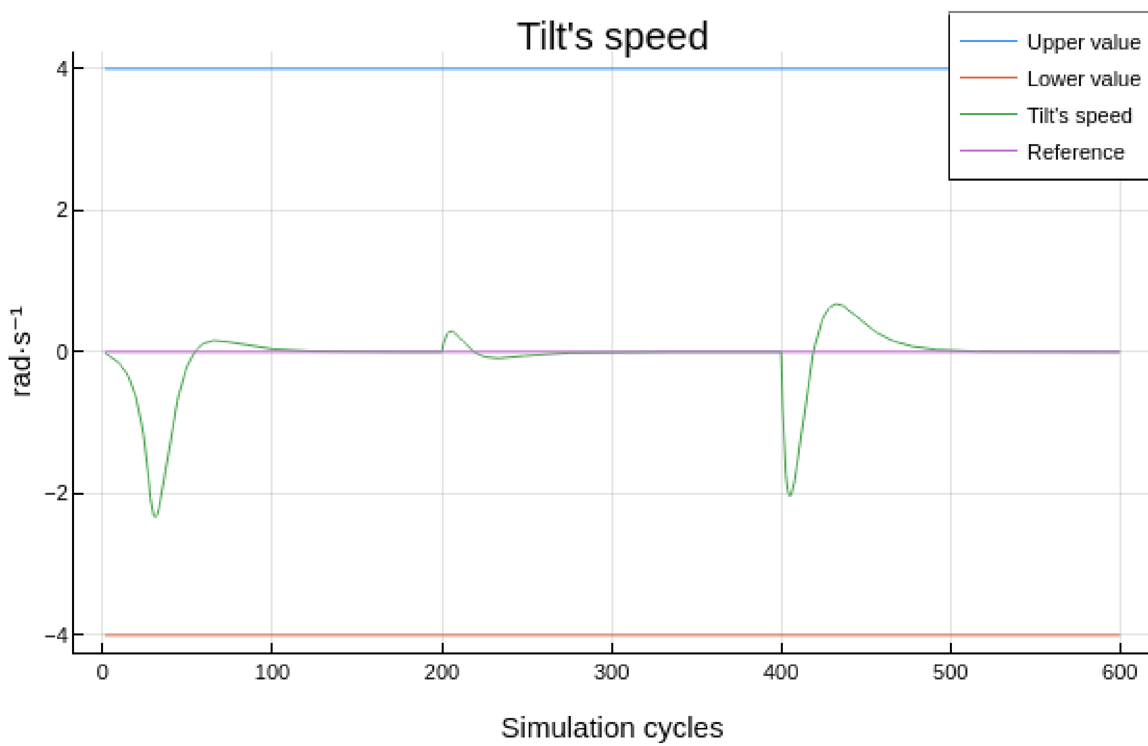


Figure 7-34. Tilt's speed (changing reference).

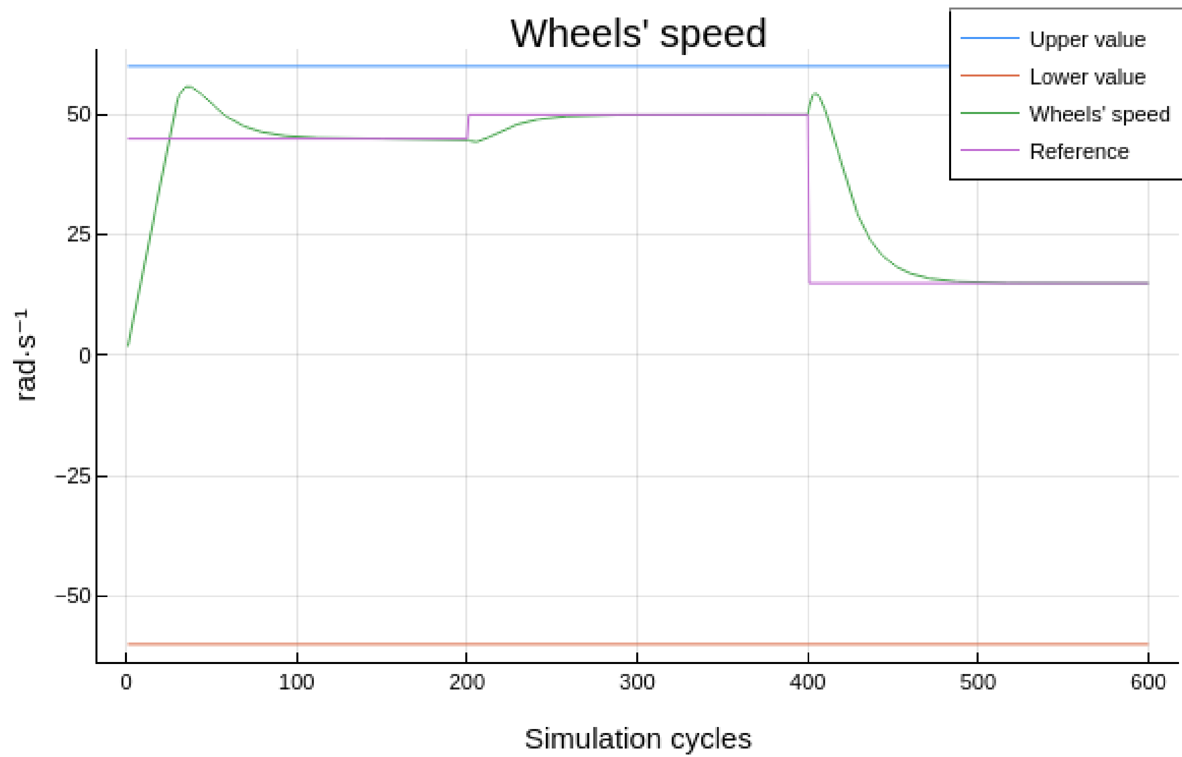


Figure 7-35. Wheels' speed (changing reference).

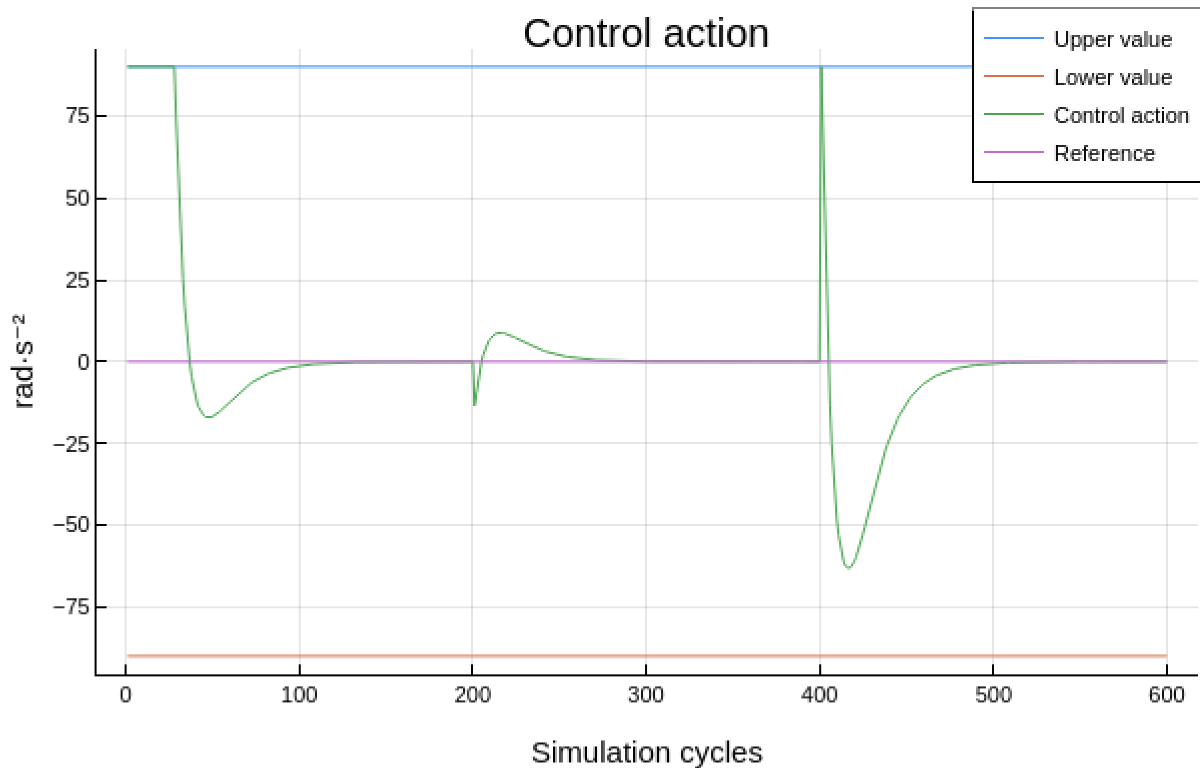


Figure 7-36. Control action (changing reference).

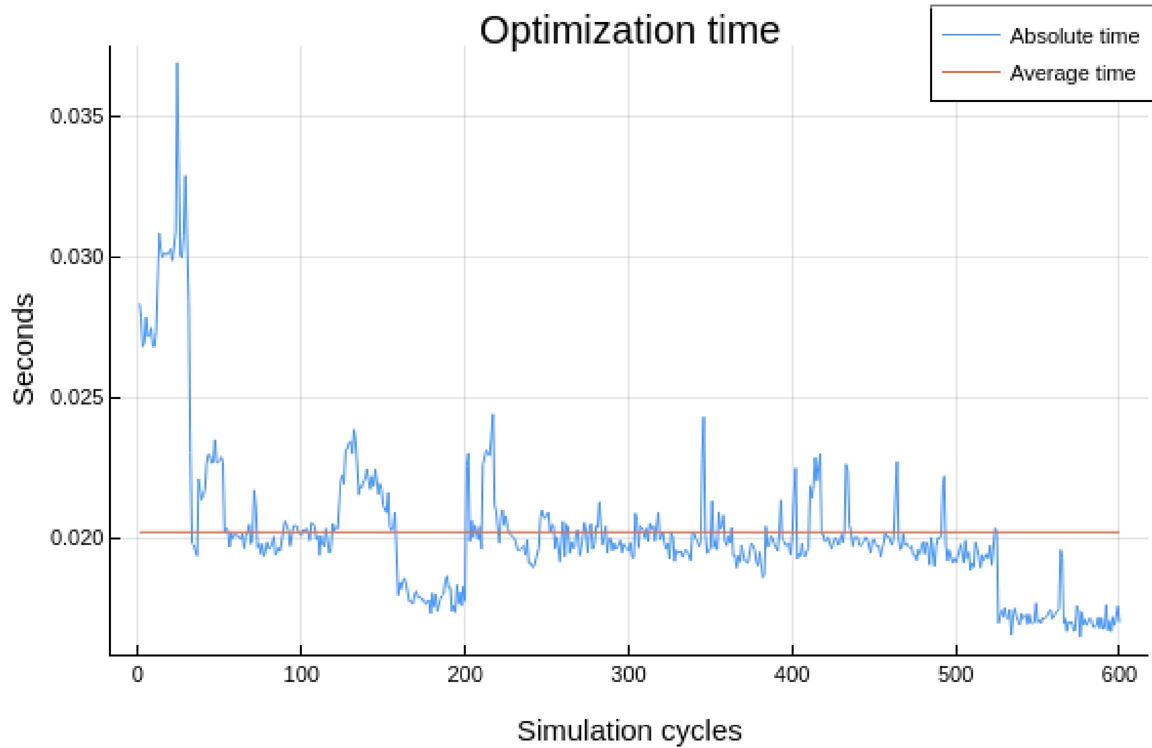


Figure 7-37. Optimization time.

```
julia> @time MPC()
0.019645 seconds of average in solving the QP
 14.366474 seconds (2.11 M allocations: 146.090 MiB, 1.45% gc time)

julia> @time MPC()
0.019894 seconds of average in solving the QP
 14.599150 seconds (2.04 M allocations: 142.446 MiB, 1.73% gc time)
```

Figure 7-38. Solving time with changing setpoints.

7.2.2.3 Using linearised model

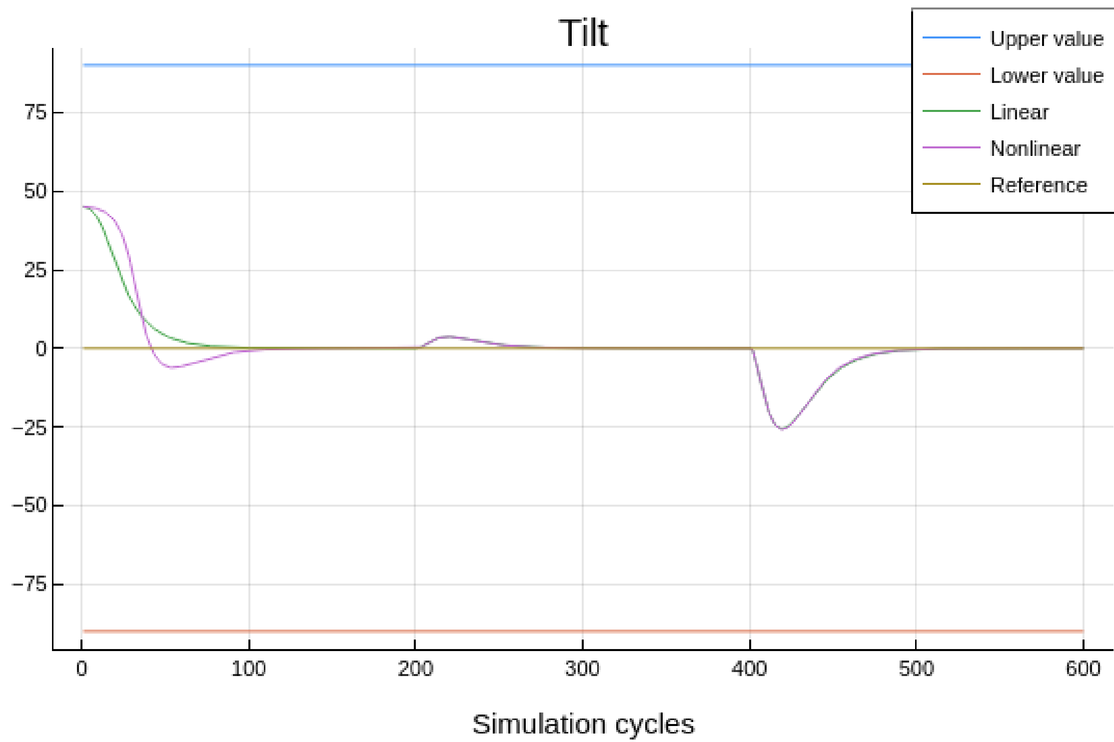


Figure 7-39. Linear vs. nonlinear tilt.

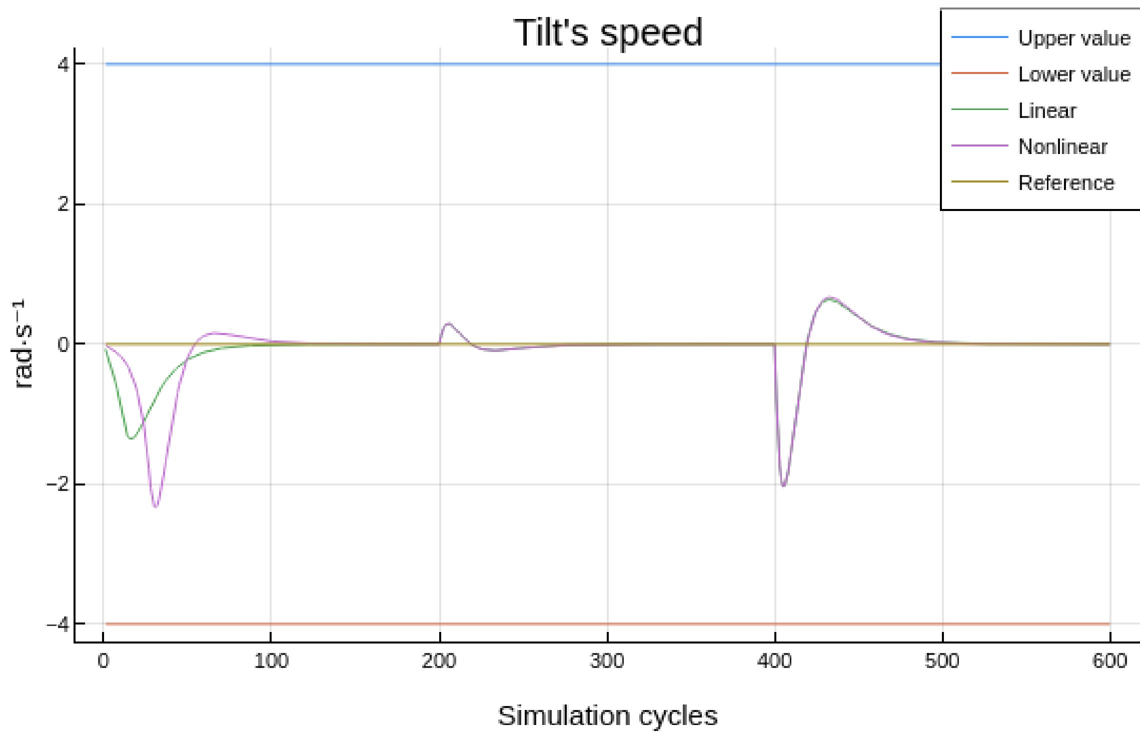


Figure 7-40. Linear vs. nonlinear tilt's speed.

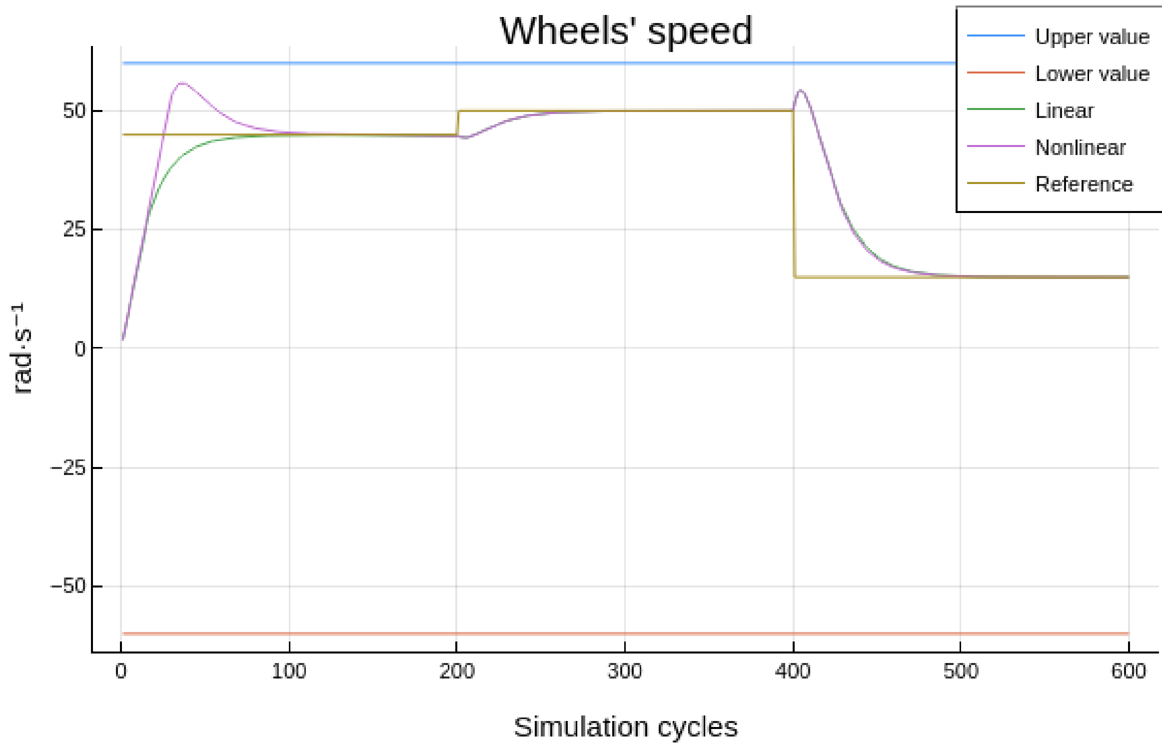


Figure 7-41. Linear vs. nonlinear wheels' speed.

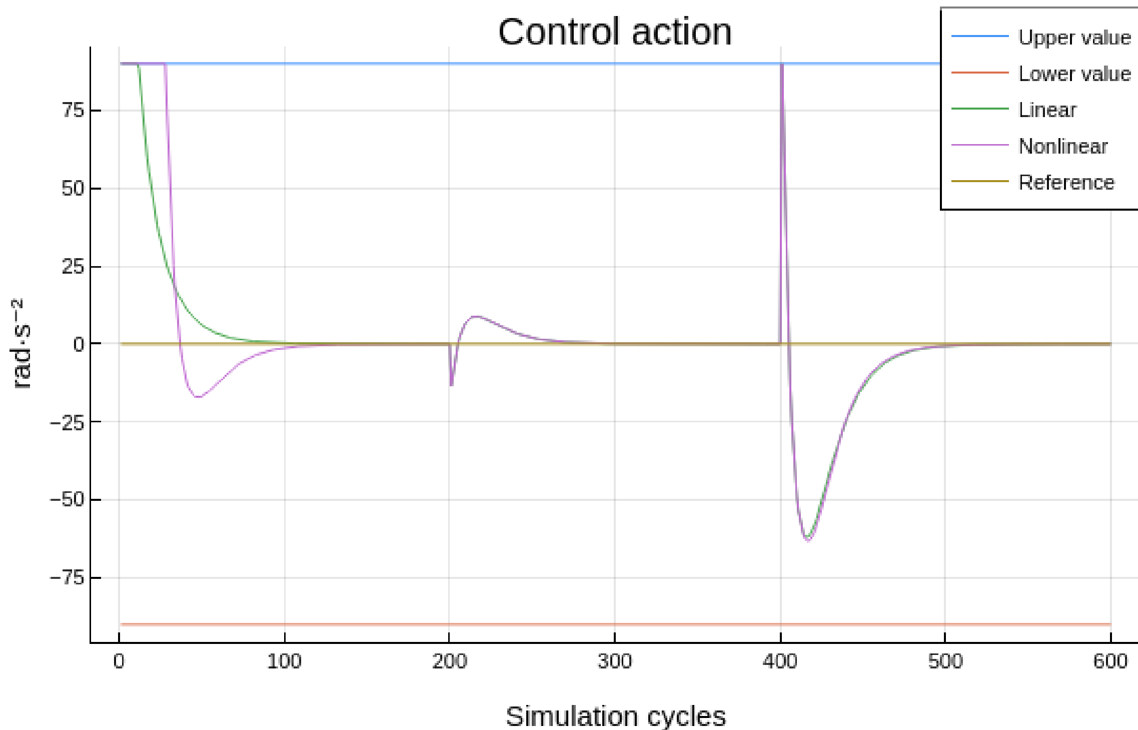


Figure 7-42. Linear vs. nonlinear control action.

7.3 MATLAB's programme

The results given in MATLAB were better than the ones in Julia, as the robot reached the setpoints almost in the same way but at a superior speed.

The experiments here performed are the same of those made in Julia, with the same changes in reference, same

initial states, etc.

7.3.1 Steady reference

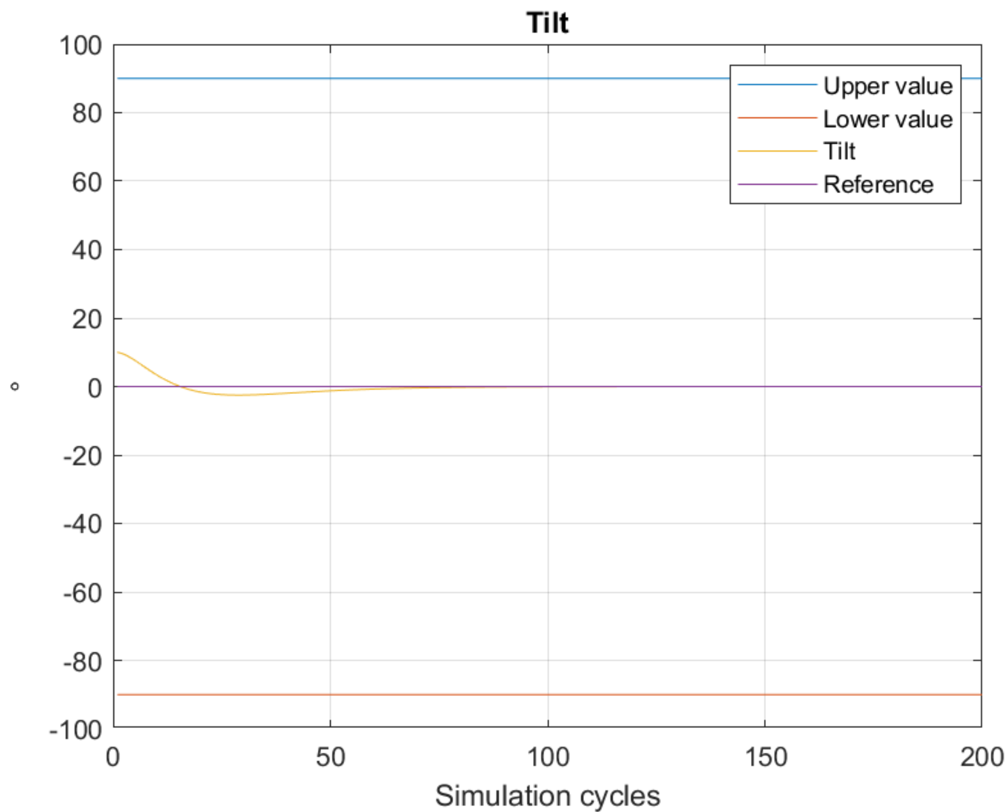


Figure 7-43. Tilt (steady reference).

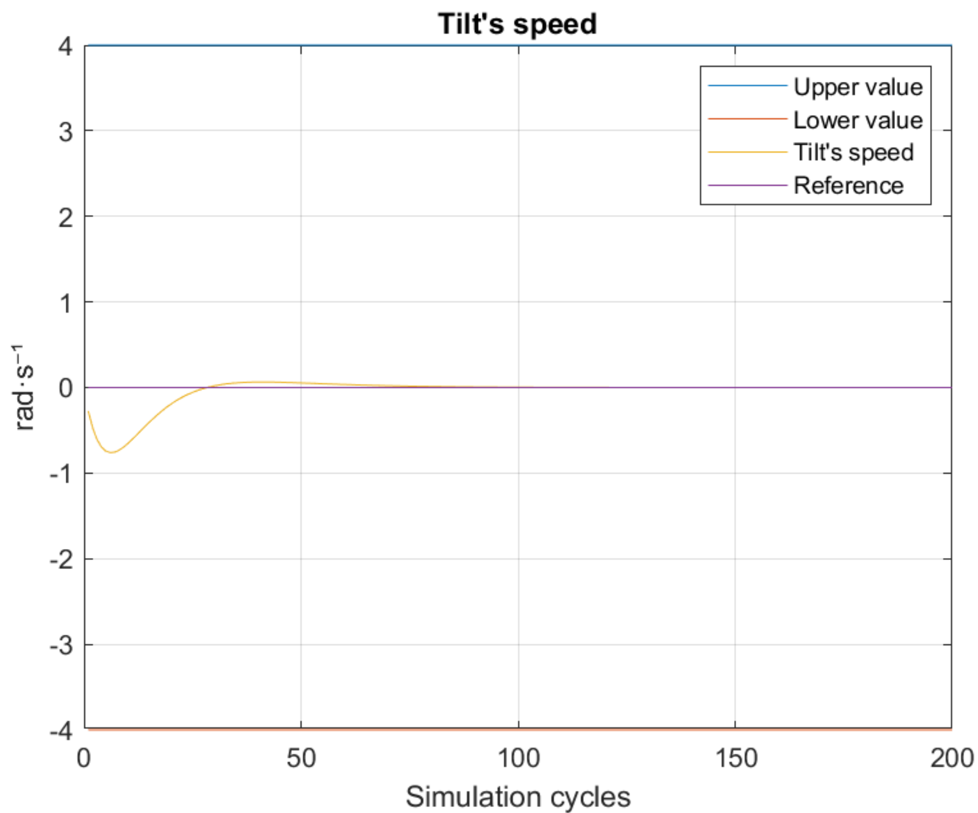


Figure 7-44. Tilt's speed (steady reference).

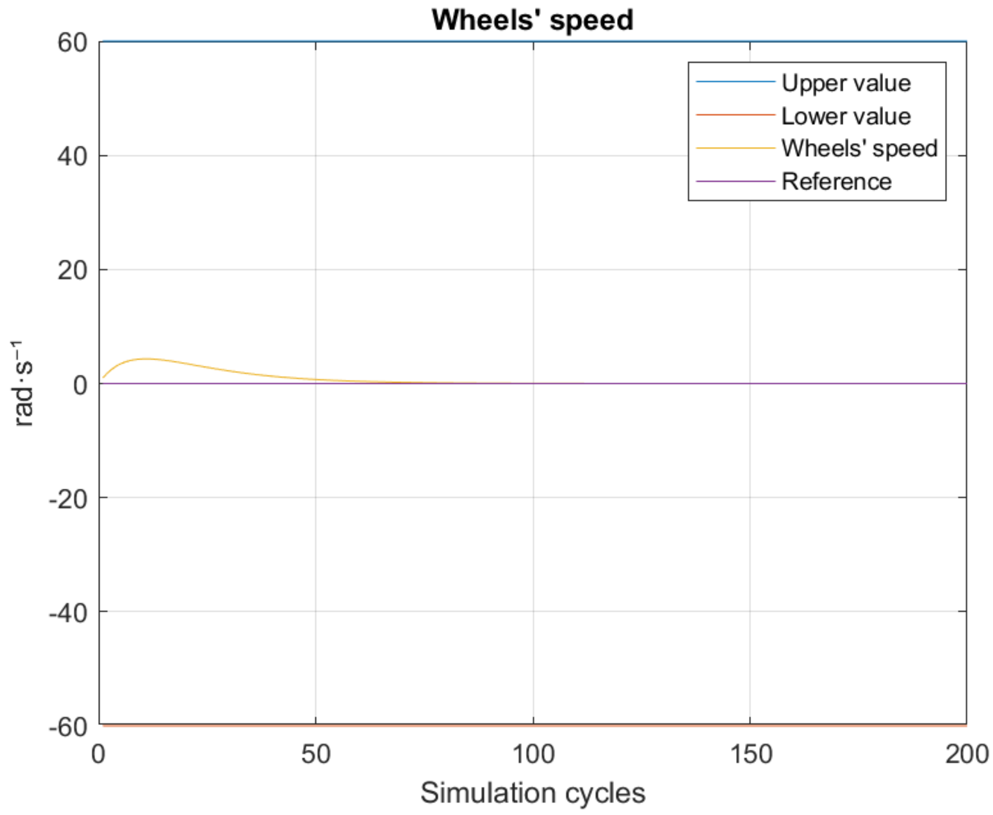


Figure 7-45. Wheels' speed (steady reference).

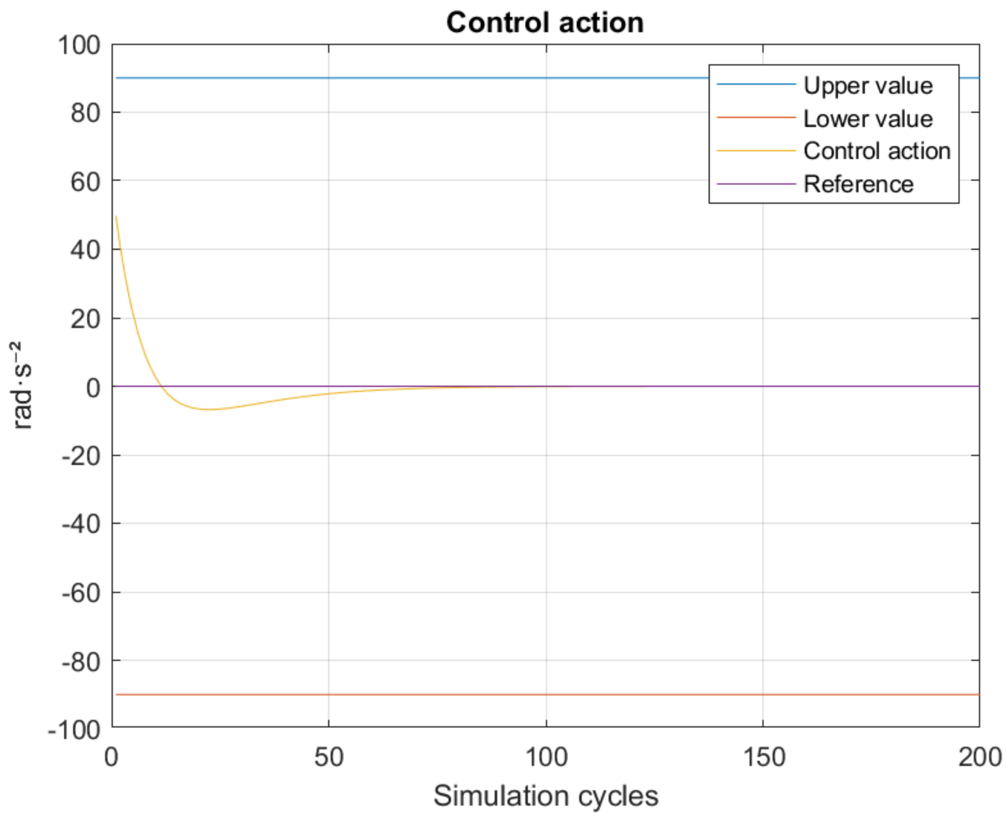


Figure 7-46. Control action (steady reference).

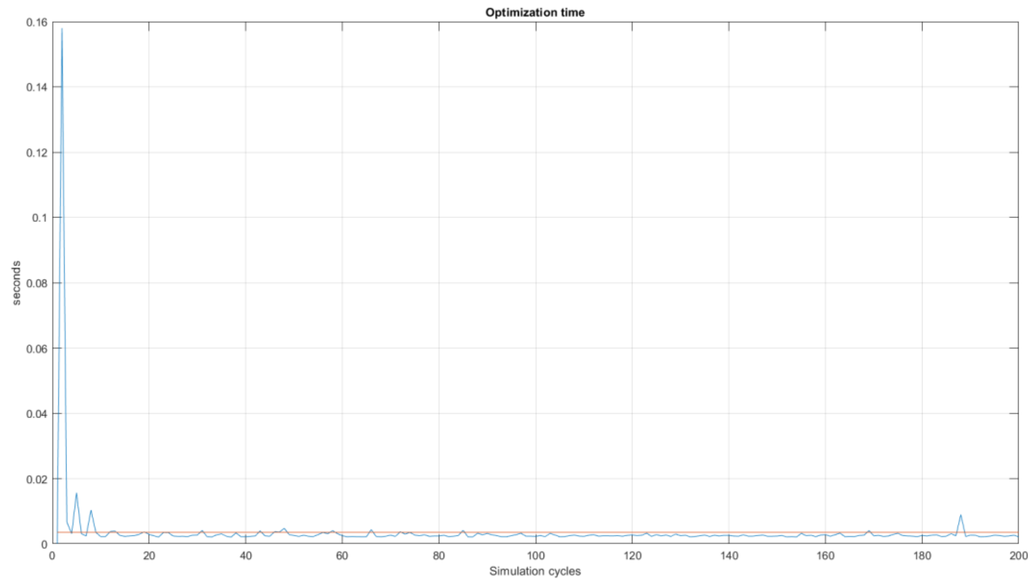


Figure 7-47. MATLAB's optimization time with steady reference.

In this case, the average computation time is of **0.0036 seconds**, approximately half of that obtained in Julia

7.3.2 Changing reference

When trying to replicate the experiment performed in Julia, with an initial state of:

$$x_r = \begin{bmatrix} 45 * \frac{\pi}{180} \\ 0 \\ 0 \end{bmatrix} \begin{matrix} rad \\ rad \cdot s^{-1} \\ rad \cdot s^{-1} \end{matrix}$$

The results were the following:

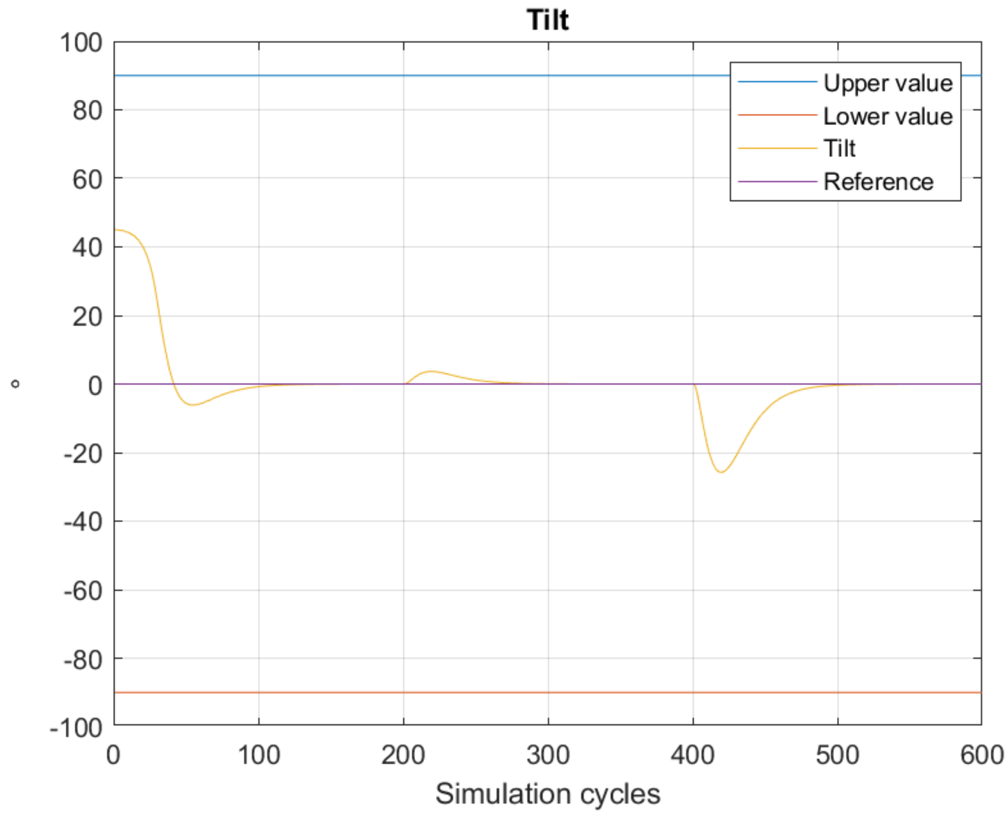


Figure 7-48. Tilt (changing reference).

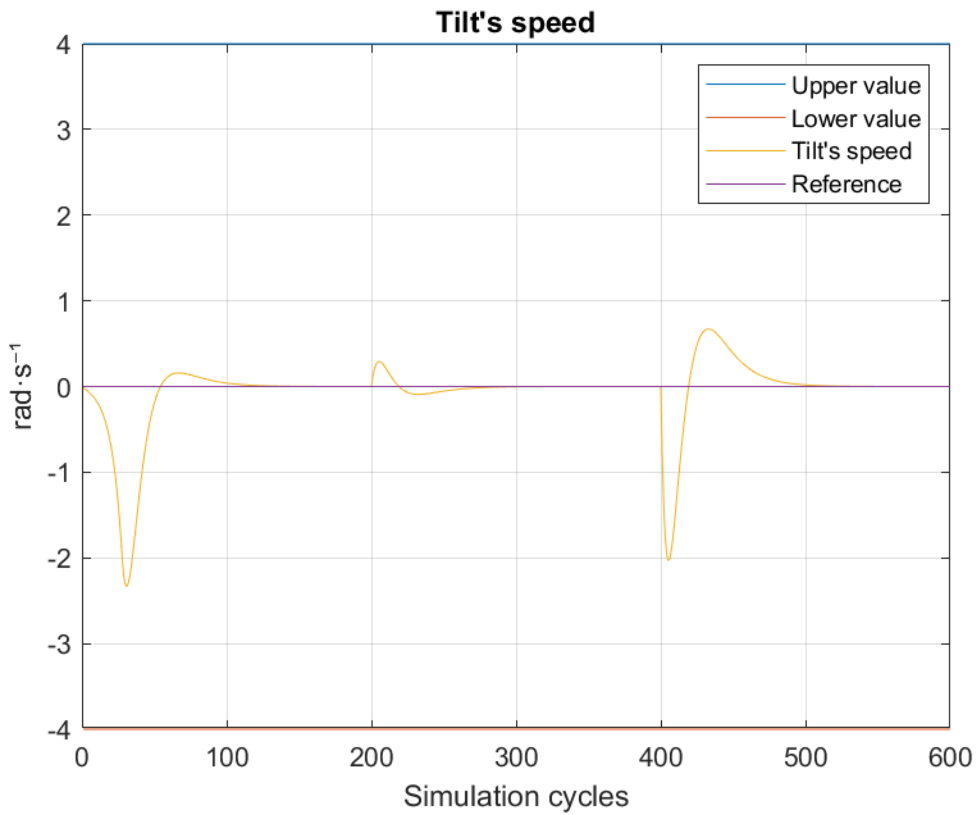


Figure 7-49. Tilt's speed (changing reference).

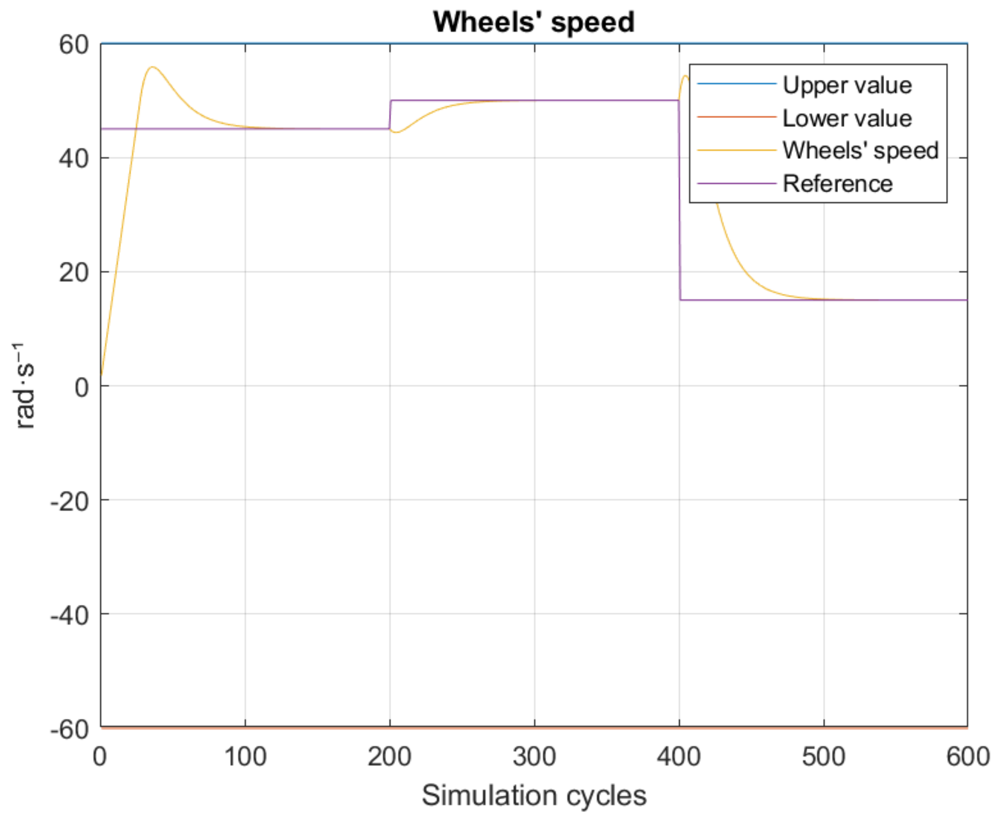


Figure 7-50. Wheels' speed (changing reference).

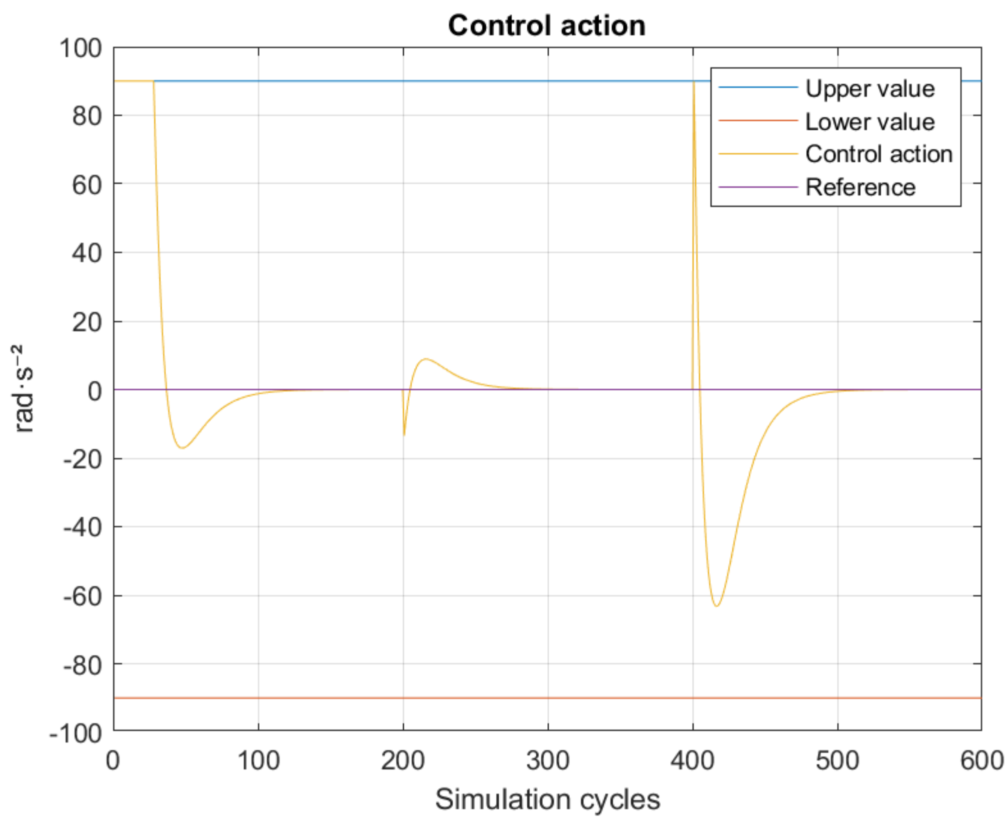


Figure 7-51. Control action (changing reference).

Here, a detailed image of the optimization time is shown:

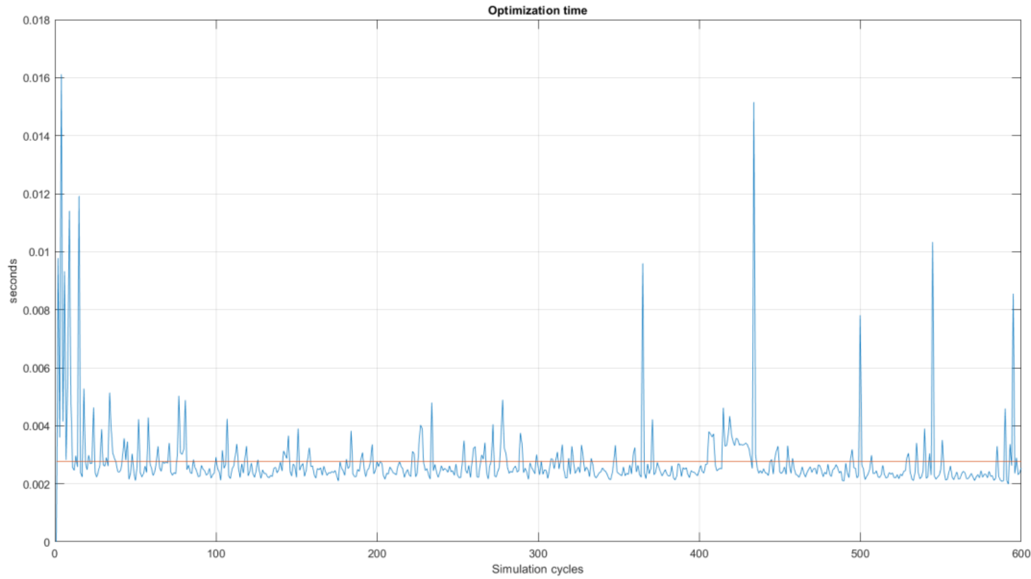


Figure 7-52. MATLAB's computation time.

With an average time of **0.0028seconds**, similar to the results obtained in Julia's version.

7.3.3 Using linearised model

This subsection shows the results obtained with the linearised model in MATLAB:

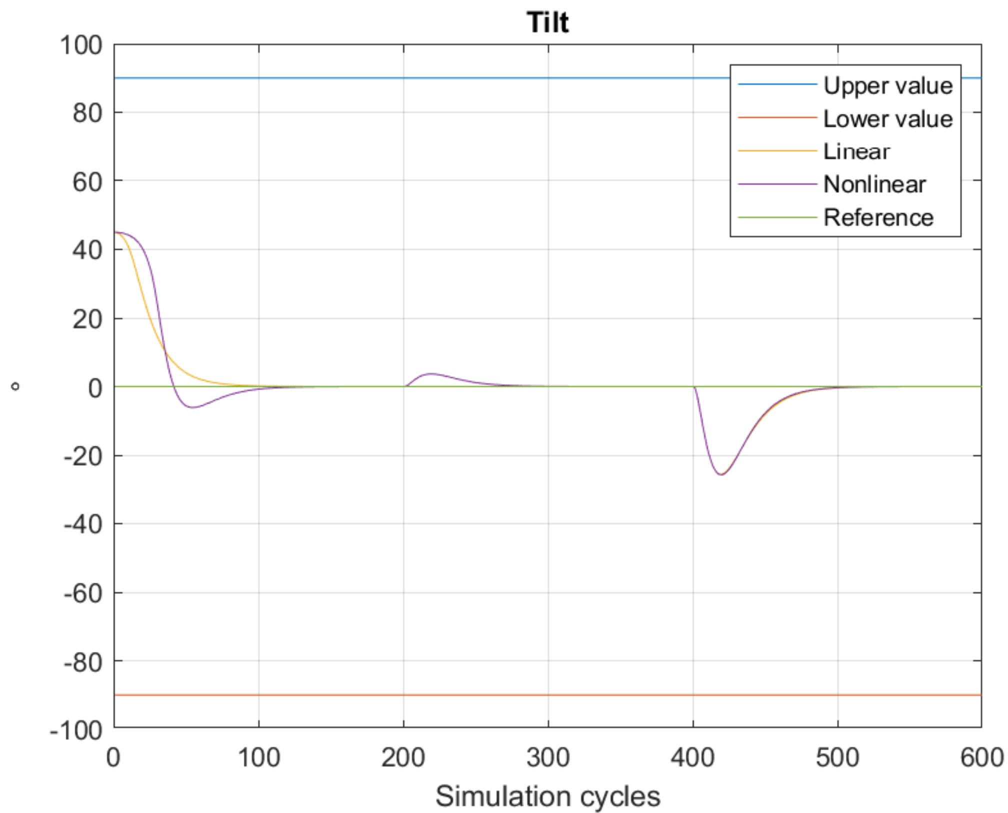


Figure 7-53. Linear vs. nonlinear tilt.

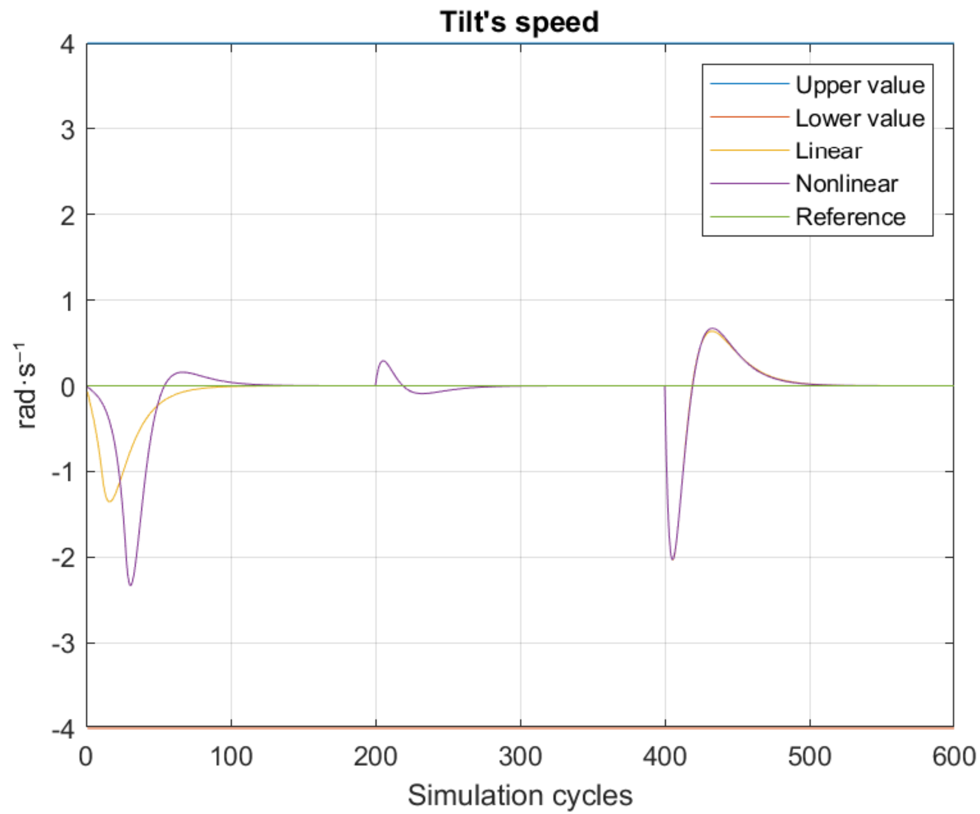


Figure 7-54. Linear vs. nonlinear tilt's speed.

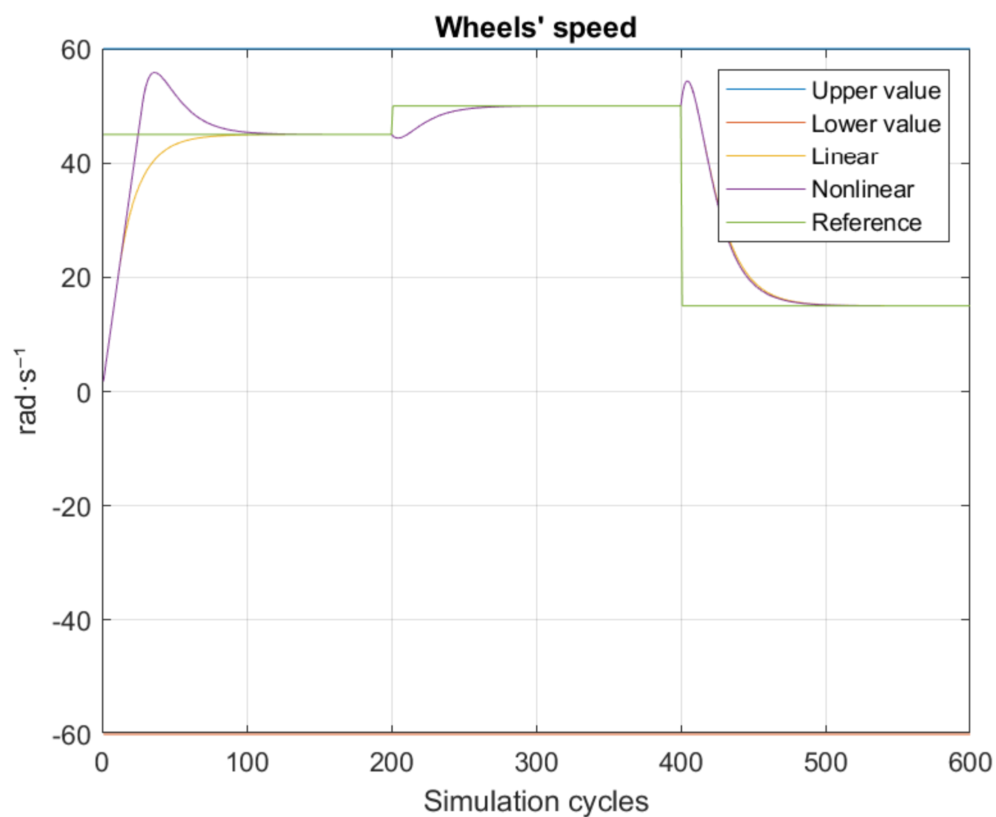


Figure 7-55. Linear vs. nonlinear wheels' speed.

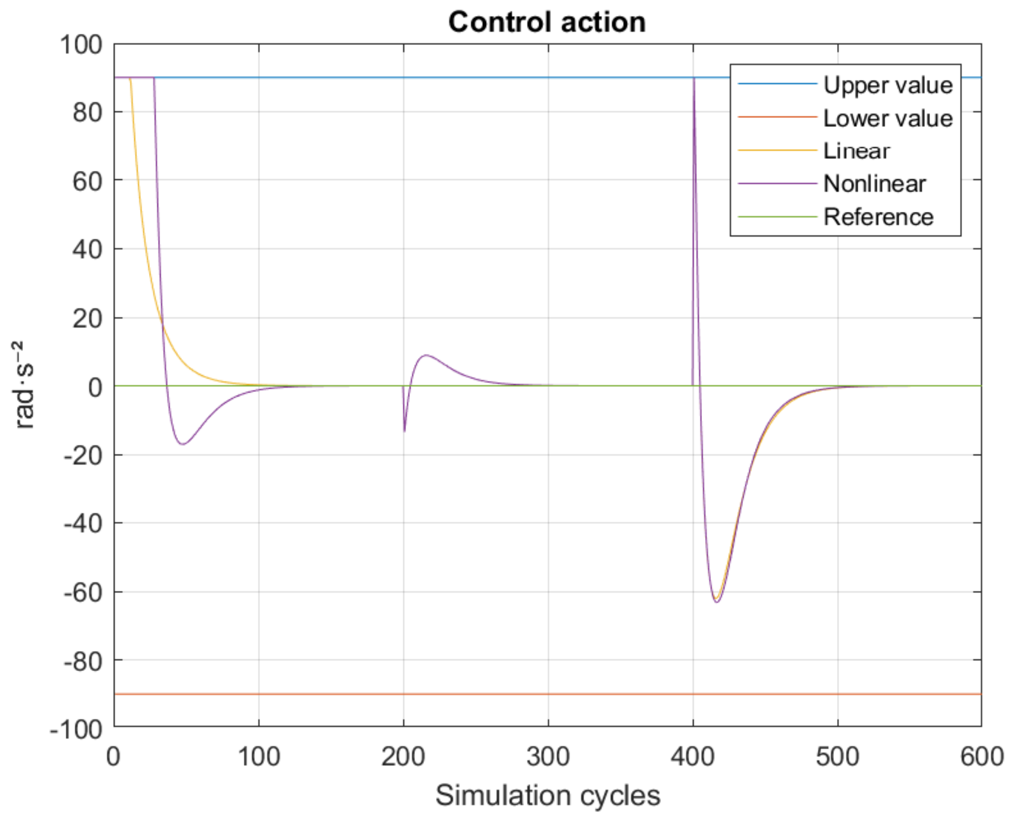


Figure 7-56. Linear vs. nonlinear control action.

Similar to Julia's version, linear approximation (yellow) is smoother than nonlinear one (purple).

8 CONCLUSIONS

*That's one **small step** for **man**, one **giant leap for mankind**.*

- Neil Armstrong -

This chapter is intended as the one including all the conclusions provided by the development of this project, i.e., an overview of what has been worked about throughout the time in which this document was written and a quick summary for the reader to extract the outputs easily.

8.1 Contributions

It has been said and demonstrated when talking about Julia that this language is faster than interpreted languages, When observing the results given by the programme developed throughout this project that is slower than MATLAB's version, but not so much and as fast as previous versions of the programme.

What could be the cause of this apparently contradictory behaviour? Apart from the fact that one is run in a more computationally powerful, it is thought that the programme has not been optimised as much as it could have been done, but instead it was made in order to provide a preliminary programme with sufficient options to be later modified with the goal of performance instead of versatility.

On the other side, it has been beheld that there is the possibility of implementing an MPC (model predictive control) strategy by using the Jetson Nano and Julia, and its further development and guidance to the programming optimization fields will be followed by the continuing projects. Julia and Jetson Nano have shown (in other areas) that each of them, independently, has the power to surpass its direct competitors—i.e. Python and Arduino, respectively—but it is indeed needed to explore more of the available coding options, such as other optimizers different from Ipopt, code optimization strategies in Julia, etc.

This project has achieved, then, to create a programme from scratch, capable of implementing an MPC based on the model equations of an inverted-pendulum-like robot, i.e., a Segway-type robot, whose model equations have also been developed from zero and, based on them, all the subsequent mathematical development, aimed to achieve a specific form given by the formulae used when solving a QP problem.

This programme can be easily modified by anyone to hold different types of systems and with a myriad of constraints, as it has been designed with adaptability in mind. It is fairly straightforward to perform changes in the programme and they will take effect automatically, as most of the code has been implemented in a referenced way, i.e. using the least possible specific information to perform its calculations. One could implement another model's equations by simply introducing the appropriate matrices and updating the constraints' relevant values. To sum up, a highly adaptable MPC programme is provided for its further development or to perform any kind of tests which are suitable in the MPC fashion.

In addition, the Jetson Nano suite and its possibilities have been explored and explained. It has been configured

appropriately to hold the programmes developed as well.

However, the reader must be rather careful with respect to Julia, as it is a language not only very recent, but under astonishingly quick development. Hence, the commands used in this project's programmes may remain useless when trying to improve what has been worked out in this text.

8.2 What is next?

Well, as commented throughout the text, the beginning goal for this project was to implement not only a programme capable of running and solving an MPC problem in record time, but to also implement it in a real robot.

Previous projects have been developed in this direction, but with Arduino, so it is a logical step to take the Jetson Nano and put it in the place of where the Arduino was, i.e. develop a fully-working prototype with the Jetson Nano attached to it and running the programme developed throughout this thesis or a newer—and possibly faster—version of it.

In addition, as mentioned, some improvements could be done to lighten the programme's computational burden. Some of the performance tips that could be tried to do so are:

- **Avoid global variables.** A global variable might have its value, and therefore its type, change at any point. This makes it difficult for the compiler to optimize code using global variables. Variables should be local, or passed as arguments to functions, whenever possible. Any code that is performance critical or being benchmarked should be inside a function.

  We find that global names are frequently constants and declaring them as such greatly improves performance. Uses of non-constant global variables can be optimized by annotating their types at the point of use.

  Passing arguments to functions is better style. It leads to more reusable code and clarifies what the inputs and outputs are.

- **Measure performance with `@time` and pay attention to memory allocation.** A useful tool for measuring performance is the `@time` macro. On the first call functions get compiled. (if `@time` has not yet been used in this session, it will also compile functions needed for timing.) The results of this run should not be taken seriously. For the second run, note that in addition to reporting the time, it will also be indicated that a significant amount of memory will be allocated.

  Unexpected memory allocation is almost always a sign of some problem with the code, usually a problem with type-stability or creating many small temporary arrays. Consequently, in addition to the allocation itself, it is very likely that the code generated for the function is far from optimal. Take such indications seriously.

- **Type declarations.** In many languages with optional type declarations, adding declarations is the principal way to make code run faster. This is not the case in Julia. In Julia, the compiler generally knows the types of all function arguments, local variables, and expressions.

- **Write type-stable functions.** When possible, it helps to ensure that a function always returns a value of the same type.

- **Avoid changing the type of a variable.**

Those are the main enhancements that could be applied to the current code, but there is a whole world about code optimisation that can be explored either in the Julia's official webpage, GitHub, forums, etc. It is highly encouraged to drive the following projects into that direction to eventually obtain the optimum code.

9 REFERENCES

[1] [Online]. Available: https://docs.julialang.org/en/v1/manual/performance-tips/.

[2] Regier et al., "Cataloging the visible universe through Bayesian inference in Julia at petascale," *Journal of Parallel and Distributed Computing,* vol. 127, pp. 89-104, 2019.

[3] [Online]. Available: https://julialang.org/packages/#julia_packages.

[4] [Online]. Available: https://www.amazon.es/dp/B01M0XSI6L?ref=ppx_pop_mob_ap_share.

[5] [Online]. Available: https://developer.nvidia.com/embedded/learn/get-started-jetson-Nano-devkit.

[6] [Online]. Available: https://developer.nvidia.com/embedded-computing.

[7] [Online]. Available: https://devtalk.nvidia.com/default/board/371/jetson-Nano/.

[8] [Online]. Available: https://developer.nvidia.com/embedded/downloads.

[9] [Online]. Available: https://julialang.org/downloads/.

[10] [Online]. Available: https://julialang.org/downloads/platform/.

[11] [Online]. Available: https://www.julia-vscode.org/.

[12] [Online]. Available: https://github.com/adafruit/Adafruit_CircuitPython_MPU6050.

[13] [Online]. Available: https://www.jetsonhacks.com/.

[14] [Online]. Available: https://jump.dev/JuMP.jl/dev/.