UCLouvain

epl

École polytechnique de Louvain

Innovating driving solution for numerical multibody systems

Oriented towards bikes

Author: **Max Opdecam**
Supervisor: **Nicolas Docquier, Paul Fisette**
Reader: **Gianluca Bianchin**
Academic year 2025–2026
Master [120] in Mechanical Engineering

Abstract

In the study of multibody system, it is sometime required to control the system in a certain aspect in order to see how it will behave. And this is the case for bikes, where we need to keep it in balance throughout the trajectory we desire to follow. We will thus create one of those controller, in our case it is a predictive control solver capable of handling all type of tree-like topologies multibody system. In a predictive control solver, we predict the movement of the system for a certain amount of time and optimise that prediction via a certain cost function. This solver requires knowing the equation of motion of the system and the control variable it can change in order to control the behaviour of the system. Those kinds of controller are very capable, robust and versatile controller with the main drawbacks of needing some computation time. We have created this controller by using two numerical tools, the first one is Casadi, a nonlinear solver develop at the KUL, and the second is Robotran, a multibody environment develops at the UCL. We will explain how the controller work and present the result and experience to show the capabilities of such controller. We will also talk about its advantages and drawbacks. Furthermore, we will go from a simple analytical model to a complete multibody system of a bike, and each time we will improve the controller. This work is oriented towards bike, but this solver could be used for every tree-like topologies multibody system.

Preliminary note

Utilisation of AI assisted tools

For this Thesis, two AI tools were used. ChatGPT from OpenAI and Grammarly from Grammarly.Inc. Grammarly was used for correcting the spelling and grammar mistakes. And ChatGPT was used to generate a picture of a ski. These were not used in order to create explanation nor are responsible for the content of this thesis.

Acknowledgement

This work has only been possible thanks to the efforts of other people. I want to thank them all for their help, advice and encouragement.

First of all, I want to thank the persons who help me throughout this master thesis : Nicolas Docquier, for being the supervisor of this master's thesis and for his valuable expertise, encouragement and availability.

Thanks to Paul Fisette for being part of the jury.

Thanks to Gianluca Bianchin for being part of the jury.

Notation

- X :is the SX array of the optimisation variables for our predictive solver
- lbx and ubx : are two SX arrays serving as the lower and upper bound for X that the solver must comply with.
- obj : is the SX scalar representing our cost function of the predictive solver
- g: is the SX array representing our constraint of the predictive solver
- lbg and ubg : are two SX arrays serving as the lower and upper bound for our constraints in g.
- P : is a SX array containing all the variables that the predictive solver cannot change.
- N: is the number of time step for the prediction
- T : is the time step of prediction

- δt : is the time step of integration
- nb_{frame} : s the number of time step for the integration
- x : is the position in the x-axis in [m]
- y : is the position in the y-axis in [m]
- z : is the position in the z-axis in [m]
- v_x : is the speed in the x-axis in [m/s]
- v_y : is the speed in the y-axis in [m/s]
- v_z : is the speed in the z-axis in [m/s]
- $\dot{v_x}$: is the acceleration in the x-axis in [m/s^2]
- $\dot{v_y}$: is the acceleration in the y-axis in [m/s^2]
- $\dot{v_z}$: is the acceleration in the z-axis in [m/s^2]
- θ : is the yaw angle in [degrees]
- ϕ: is the roll angle in [degrees]
- γ: is the pitch angle in [degrees]
- θ_p : is the yaw angular speed in [degrees/s]
- ϕ_p: is the roll angular speed in [degrees/s]
- γ_p: is the pitch angular speed in [degrees/s]
- $\dot{\theta_p}$: is the yaw angular acceleration in [degrees/s^2]
- $\dot{\phi_p}$: is the roll angular acceleration in [degrees/s^2]
- $\dot{\gamma_p}$: is the pitch angular acceleration in [degrees/s^2]

Contents

Chapter 1

Introduction

1.1 Context

Multibody dynamics is a well known research field where we study the dynamics of vehicle and their stability. In recent years, we are looking to use more environmentally friendly means of transport and logistic, and the bike could be a solution for the short urban transportation. This is one of the reason we have seen the rise in popularity of cargo-bike and electric bike. But with all those new ways of viewing the application of bike in our life, come the challenge to design bike that are easy to drive. And the stability of a bike is far from being a simple topic.

There has been a good deal of research on bike's stability, coming with all sort of conclusion and observation. But the easiest way to compare the stability of two bikes is to ride them and see which one behave the best. Sadly, producing and manually testing every single architecture of bike is not feasible. But we could test them in a numerical environment. However, this approach will require the creation of a controller for our bike that is robust, versatile, accurate, and fair.

This is the creation of such controller that we will explore in this thesis, from its logic to its results, going through its advantage and disadvantages and ending by its potential improvement.

1.2 Objectives

The aim of this master is thus the creation of a controller that will be able to control all kind of bike. We will thus first look at how we can control multibody systems and applied it on simple multibody model to serve a test case and conduct some experience in other to see what is possible and which direction to take.

After having succeed in controlling simple multibody systems on its own, the next goal is to incorporate the controller inside a multibody environment call Robotran. The idea behind this is to use the environment of Robotran to generate the multibody system. While doing so, we will also do experiments to optimise the incorporation and use of the environment of Robotran in the solver.

And the final goal is fully controlling a bike in Robotran and be able to study the submodels of that bike in a parametric study to show the robustest and fairness of the controller.

1.3 organization

The master thesis is organised in 6 chapter. Each chapter will build up the knowledge on the controller from the previous one to finally lead to the last multibody model. This chapter 1 give the context, objective of this thesis and end with the outline of the thesis.

The second chapter will briefly present the type of controller that have been considered. We will talk about their quality and drawbacks, their inner working, and finally why we chose the to use a predictive controller using the Casadi nonlinear solver.

Afterward, in chapter 3, we will start by presenting Casadi and its parameters. Then we will use our predictive solver in some simple analytical multibody model. The first is called "Roomba" and it will be created using the analytical equations of motion. In this first model, we will explain in detail on how we used Casadi, and we will be taking advantage of the fact that it is the only model fully readable by a human. Then we will study the parameters of our solver and see how do they impact the final trajectory. Afterward in the second analytical model, the "Inverse Pendulum" we will talk about all the integration and prediction scheme we can use and how do they interact with one another.

The models using both Robotran and Casadi will be explored in chapter 4. We will first present Robotran, our multibody formalism and how Casadi and Robotran interact with one another. Then we will recreate our "Inverse pendulum" in Robotran using two different approach. The idea is then to compare those two different approach to the analytical model to verify the accuracy of the controller. We will also compare the two approach and chose one for the rest of this thesis. Afterward, we will present the "Roomba Pendulum" model and with it explain how we decrease the time of computation by a significant margin.

In chapter 5 we will present the technical aspect of our final model, the "Ski-Bike". We will talk about its geometry, and we will present a simplified version of the ground contact model by Schwab applicable to ski.

And finally, in chapter 6, we will talk about the result of the Ski-Bike model. We will do a short parametric study to show the robustness and fairness of the controller. Then we will talk about its shortcomings, and finally we will do a conclusion talking about the reflection and potential improvement we could make to both the Ski-Bike and solver.

Chapter 2

Control types

In this chapter, we will present multiple ways we can control a multibody system on a more theoretical basis. We will present these control types relatively briefly because the exploring of multiple control type was not the goal of this thesis. The goal was to choose the most promising and focus on it.

2.1 Terminologies

In every type of controller, there is a common terminology we will use. The first is a distinction we will make between two type of variables in our multibody system.

- State variables : Those are the variables describing our system. It can be a position, a speed, an angle ... But most importantly, those variables will not be directly control by the controller. And due to the state variables being used to describe our multibody system, if one of them is missing, our system will no longer work.
- Control variables : Those are the variables that the controller will control. The system can exist without them, but if there is none of them, the system will just be uncontrolled.

Here is a good example to differentiate the two. The speed of a bike is a state variable, but the torque applied on the rear will is a control variable. A bike can exist without a torque on its rear wheel, but not without a speed. And of course the speed will change depending on the torque on the real wheel. Meaning, the state variables are controlled by the control variables.

The second terminology is the criteria we will use to judge a controller.

- Robustness: A slight change in the model will not lead to a completely different behaviour from the controller.
- Versatility : The controller can work with different models
- Accuracy : The constraint we put on the controller will be followed, and the controller would not do things that are impossible in real life. Plus, a small change in the controller will not lead to a completely different result.
- Fairness : The controller will be equally competent with all models.

2.2 Artificial intelligence

Since the arrival of AI, it has entered a lot of field in science and multibody dynamic is one of them. There are many algorithm that fall under the name AI, but of course here we are talking about a deep learning algorithm using a neural network. Those types of algorithms need to be trained on a certain problem with a certain cost function, and over time they will "learned" how to reduce that cost function. The method for them to learn, depends on the deep learning algorithm

That method of controlling a multibody system has been moved aside because it has an issue for us. The neural network resulting from that training is a "black box" meaning we have no way of knowing how it works and thus no way of granting the robustness the versatility and the fairness of such controller. Because it is logical that these kinds of controller would be more capable with the models it has been trained on. And thus the fairness of such controller is debatable. Plus, the only way to make it robust and versatile is to train with a countless number of models for countless number of hour and countless different scenario.

We have thus decided to go with a more arithmetic approach.

2.3 PID and pure pursuit

The most simple way of controlling a multibody system is a pure pursuit algorithm bind with a PID. That type of controller need a predetermined path and will try to follow it. The idea is simple, the models will look ahead on that predetermined path and will compare its current trajectory to that path. The difference of position, speed and acceleration will then be used inside a PID to manage one of our control variables. And each control variable will need its own PID.

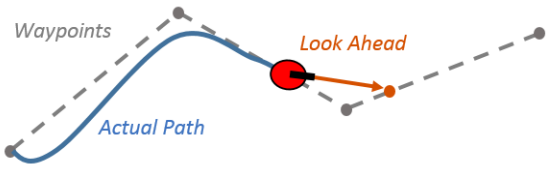


Figure 2.1: pure pursuit

Those kinds of controller are remarkably versatile, working with all kinds of models. But of course they are not robust, the slightest change in the model will require a complete revaluation of all the weight parameters of all the PID. Plus, the change of path will demand to change the distance at which the pure pursuit is looking ahead. With all those change require for each deviation, the fairness is inexistent. A model could have a better behaviour just because its PID has more adapted weights for that specific situation. And the accuracy of such model is also inexistent, it will do things that are physically correct but the path we gave them is more of a guideline that the actual path they will do. Meaning that with just a slight modification of the controller, we will obtain a vastly different result.

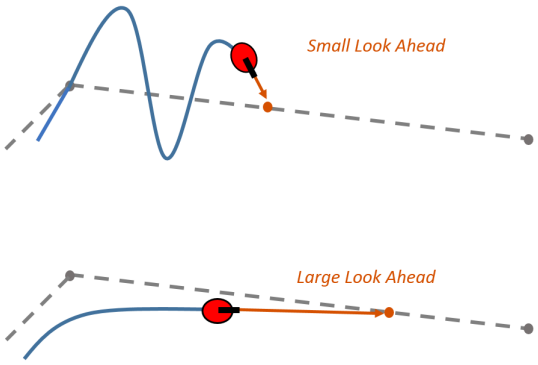


Figure 2.2: pure pursuit accuracy problem

2.4 Optimal control

This control type is the theoretically best control type. Its idea is very simple, we will plan the entire trajectory ahead and optimise that entire trajectory to reach a certain target. We can plan the entire trajectory ahead by knowing how the model will behave. For us, those equations are called the equation of motion

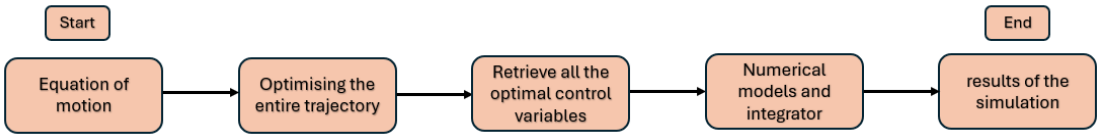


Figure 2.3: Optimal control

The robustness the versatility, the accuracy and fairness of such model is perfect. On the other hand, the downside is obvious the computation time for optimising the entire trajectory of a model is immense and for some average size model it becomes impossible due to the memory usage. Plus, if there is any error between the prediction and simulation, it will not work.

2.5 Predictive control

This controller is the one we will use. It uses the same idea as the optimal control, but only applied it for a predetermine horizon. The horizon is the distance the solver can anticipate the trajectory and thus the control variables. The anticipation of the trajectory is called the prediction. In a perfect world, the prediction will become the actual trajectory the model will follow. Here is a visual representation.

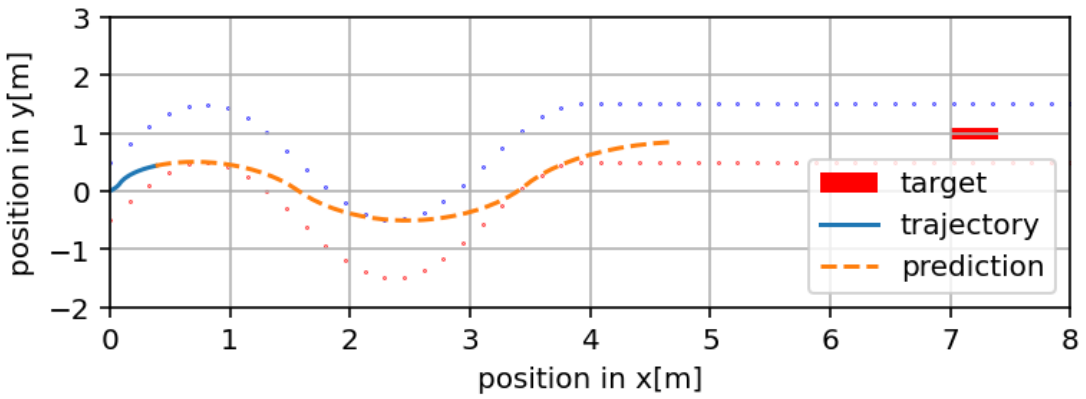


Figure 2.4: Predictive control

We have here the classic representation of a predictive solver in a 2D plane. We have the trajectory as a continued lined, the prediction has a dashed line, the dotted lines are the bounds that the model must not cross and the desired target that the predictive solver is trying to reach. The length of the prediction is call the horizon, and of course if that length is infinite we have an optimal controller.

The prediction will be recalculated many times during the simulation of the model because the prediction is optimal for a specific time step but will not be the case any more for the following time step of simulation. We thus have to keep the prediction update in a loop.

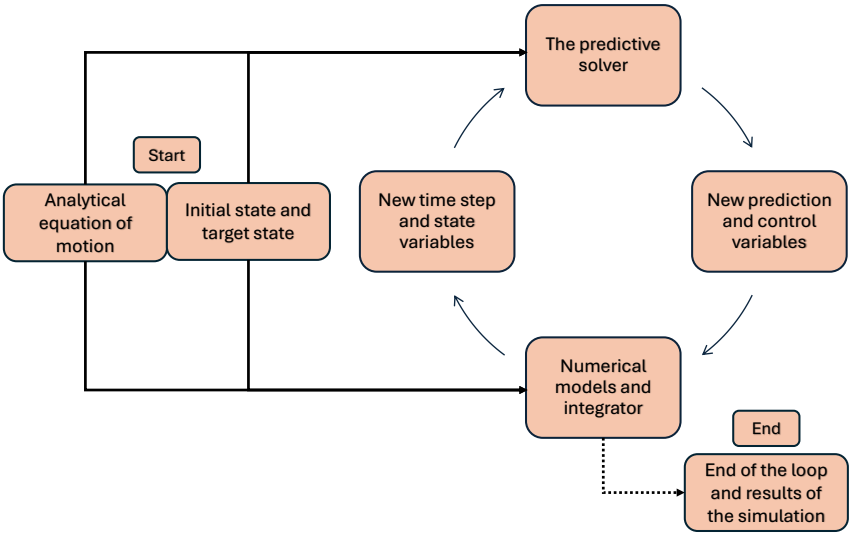


Figure 2.5: Predictive control organisation

Chapter 3

Intermediary Numerical models using analytical equation of motions

Now that we saw the different way, we could control the bike. We can now talk about the models. In this chapter we will present our solver Casadi, explain how it works and the solving methods it supports. Then we will explain how we implemented the solving methods to obtain our predictive solver. And afterward we will present all the models that use the analytical equations of motions and the difference between them, each model being closer to controlling a bike.

3.1 Casadi

Casadi is an optimiser for non-linear system of equation. It is being developed by the KULeuven in the Optimization in Engineering Centre. Casadi can thus optimise all kind of systems and is very generalised. But in our case, it will serve as our controller, being able to solve optimal control problems and thus predictive control problem.
To encode a problem in Casadi we must use symbolic variables and construct our system using Casadi's convention. That part will be referred to as the writing of the solver. After writing the solver, we will be able to call the solver to solve our system depending on the parameters we give in the call function.
The non-linear solver in Casadi can vary depending on the application, but for all our models, we will use the same called "iopt".

3.1.1 Symbolic variables

In Casadi there are 3 type of symbolic variables, all of which with their uses. But for our case we will only be needed the DM and SX variables

DM variables

The DM variables are the "INT" equivalent for Casadi. They are not symbolic, meaning they have a value and that value is known. The difference with "INT" is that there are compatible with the specific Casadi function.

SX variables

The SX variables are the scalars symbolic variables of Casadi. It is the type of variables that we will be using when constructing our non-linear system during the writing of the solver. By scalars symbolic variables, we mean a variable which its value is unknown, but we know that it is a scalar.
Those scalars symbolic variables can be arranged into a vector or even matrices. If it's the case, we then talk about SX array. But those array will be treated as a collection of scalars, not a tensor, by the solver.

MX variables

The MX variables are the matrices symbolic variables of Casadi. Those will be treated as tensors. But because we don't use them in our models, they will not be explained furthermore.

3.1.2 solver structures

Each solver in Casadi has the same convention and parameters. Some of those parameters need to be defined during the writing of the solver, others are only needed when we call the solver. But they are all necessary, and they convert all different roles so it is important to not mix them up.

Those parameters are:

- **X**: is the SX array of the optimisation variables. It contains all the variables that the solver will try to optimise. We give this information during the writing of the solver, but when we call the solver, we can give a first guess.
- **lbx and ubx** : are two SX arrays serving as the lower and upper bound for X that the solver must comply with. We give this information when we call the solver.
- **obj** : is the SX scalar representing our cost function. The solver will try to minimise this function. We give this information during the writing of the solver.
- **g**: is the SX array representing our constraint. The solver will try to make sure that all the constraints are true. We give this information during the writing of the solver
- **lbg and ubg** : are two SX arrays serving as the lower and upper bound for our constraints. We give this information when we call the solver.
- **P** : is a SX array containing all the variables that the solver cannot change. We give this information during the writing of the solver, but it is when we call the solver, we fix the value of those variables.

Meanwhiles, the conventions are:

- All SX arrays parameters must be of Dimension 1.
- All SX arrays are build vertically (the opposite of NumPy)
- All variables in obj must be listed in either P or X.
- All variables in g must be listed in either P or X.
- lbx and ubx has the same size as X

- lbg and ubg has the same size as g

These are all the principles to follow when using the Casadi solver. But there are different methods to implement the solver with different advantages and disadvantages. The difference between those methods are how do they handle the physical link between each prediction. We will refer to those links as physical constraints.

3.1.3 Single/Direct shooting methods

The first method is the most logical and direct way. The optimisation variables will only be our controls variables during the prediction because it is the only thing we really need to get out of our solver, it is what will control our system.

But that also mean that all the trajectory prediction optimisation will be lost in the solver because all we will receive are those controls variables. Furthermore, by not having the trajectory prediction in X, we can not put the physical constraint in g due to the fourth convention (3.1.2). We can not put constraint on the trajectory prediction if we don't know the trajectory prediction. The only way for us to respect those physical constraints in the solver is to agglomerate them in the cost function (meaning the "obj" parameter) to make the trajectory vanish. That will render our cost function extremely unreadable for a human and hard for our solver to converge on a solution.

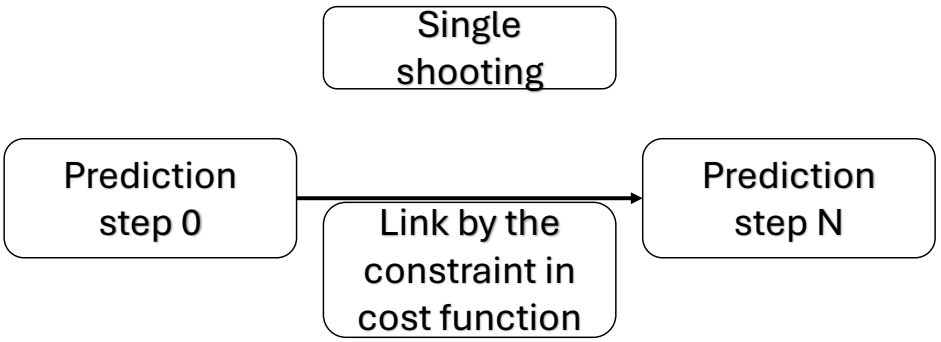


Figure 3.1: Single shooting logic

Its main quality is memory conservation, by only storing what is necessary, we of course use less memory. Its main drawback is its speed of computation due to the difficulty to converge to a good prediction. It is why we won't use that methods.

3.1.4 Multiples shooting methods

This is the method we will use for all our models. Our optimisation variables will be our control variables and also our state variables during the prediction. Meaning, we will have access to the prediction of every variable in ours models.

This enables the solver to directly optimise all the model at once, this is great for convergence. Furthermore, we know every thing there is to know about our model, this mean we can just

stack the physical constraints in the constraints array g. This will make that array very long, but at least the constraint will be simple enough that, for the most simple models, even a human can read them. The drawback is of course memory usage, but this is the recommended way of working with Casadi.

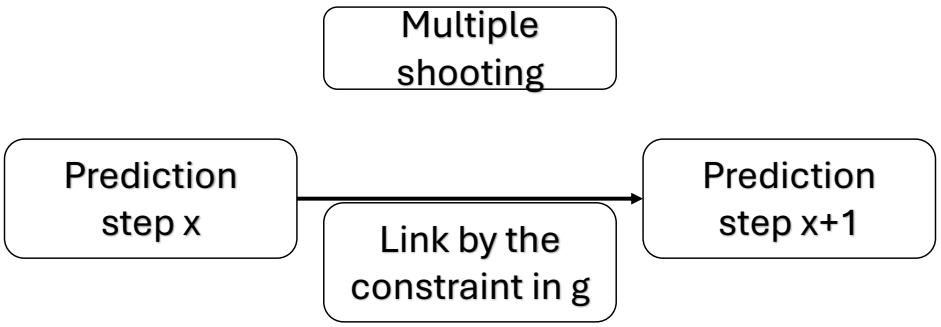


Figure 3.2: Multiples shooting logic

3.2 Numerical implementation

In this section, we will explain the organisation of the code for the analytical models. We will talk about the logic of computation and then also how and where Casadi will be used. We will present the parameters used in the writing of the solver. And talk about the equation of motions.

3.2.1 Overview and parameters

The computational logic of all analytical models work the same. The first thing to do is to manually implement the initial state, the target state and the equations of motion.

The second thing to do is to encode those equations into the Casadi solver in order to create the predicative control solver.

The third part is entering the integration loop, it works as follows. We call the solver with the state variables of the new time step (if it is the initial time step, we use the initial state). We obtain the prediction of our model behaviour but most importantly the control variables. From those control variables, we use our numerical model and time integrator to find the next time step with the new state variable, and we restart the loop.

The fourth and final part is collecting all the time step to have the complete movement of our model and being able to analyse those results.

For a better visualisation, the following schema 3.3 also explains it.

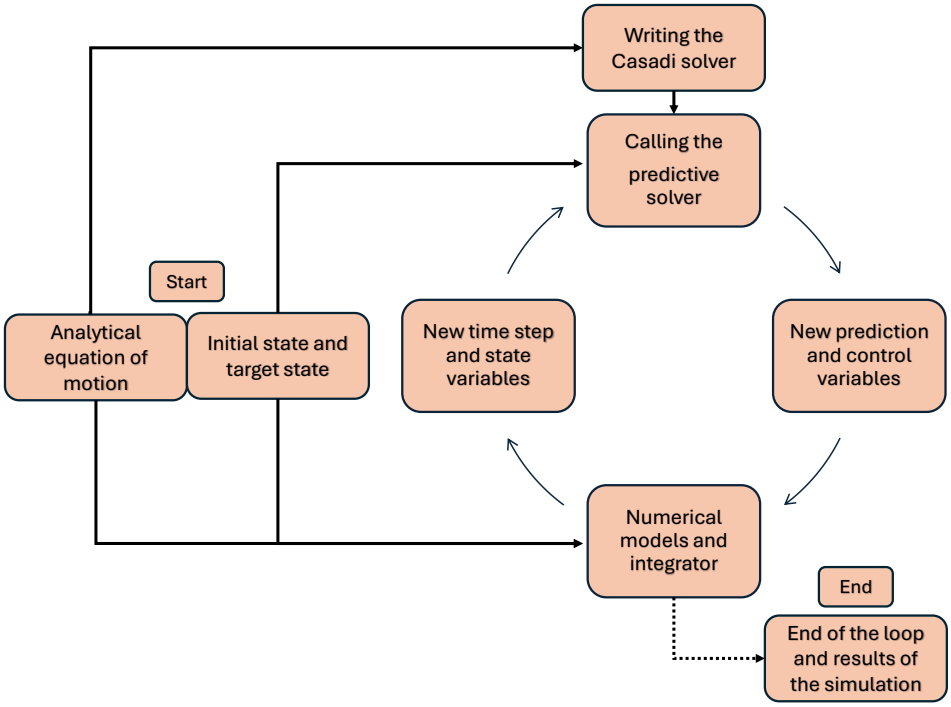


Figure 3.3: computational logic

We will now explain the parameters we can use to create different solver and numerical models during the writing.

- δt : is the time step of integration, it will be limited by the numerical stability of the models.
- nb_{frame} : is the number of time step the integration will do (so δt*nb_{frame} is the amount of time we will simulate)
- **T** : is the time step of prediction, it will be limited by the numerical stability of the models.
- **N**: is the number of time step the prediction will do (so T*N is the amount of time our prediction will actually see in advance, we will call it the horizon of the prediction)

3.2.2 Equation of motion

In regard to the equation of motion, there is a differentiation to be made between the two analytical models. For the first model, those equations will be the cinematic equation of motion, meaning we directly control the acceleration of the models. Those equations are easier for Casadi, but not very physical in reality.

And for the second models we will use the dynamic equation of motion, meaning we will control the forces acting on the model. This is the correct approach and will later make the binding easier with Robotran.

3.3 The Roomba models

This is the first model that have been done. It had served as a test case to understand how to use Casadi. This model consist of a little robot that can go straight and turn on itself, usually the kind of robots used to vacuum the floor, hence de the name "Roomba". The model simulate the movement of such a robot in a 2D plane.[1]

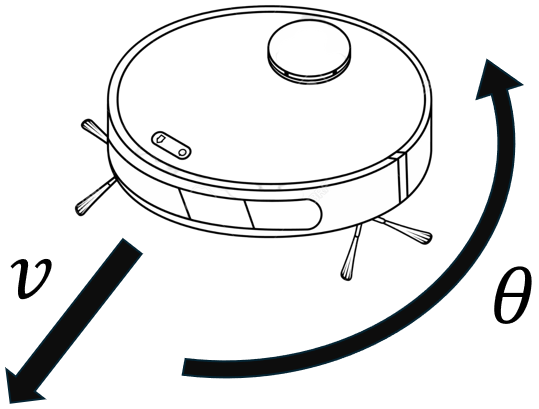


Figure 3.4: Roomba with its 2 degree of freedom

We will take our time to explain in explicit detail out that model work because every thing in that model can be read by a human. So it is a very good example to put into practice every thing explain so far. And the logic for the following models stay the same.

3.3.1 Physical model

As explain in the schema 3.3 the first part is the manual implementation of the equation of motion, the target state and the initial state. From the classical multibody theory, we can found the following cinematic equations of motion for our model.

$$\begin{aligned} \dot{x} &= v * cos(\theta) & \dot{\theta} &= \omega \\ \dot{y} &= v * sin(\theta) & \dot{v} &= a \end{aligned}$$

Table 3.1: equation of motion of the Roomba

Where x and y stand for the position in a 2D plane in [m]. θ described the yaw angle in [degrees]. v is the absolute speed in [m/s] of the Roomba. ω is the angular speed in [degrees/s] and finally a is the absolute acceleration in [m/s^2].

From those equations 3.1 we can derive two type of variable, the state variable and the control variable. This distinction is not necessary in order to use Casadi. It can handle that part by itself, but for the coding it is useful. We are using the cinematic equation of motion thus we are

[1]This model is inspired from this video https://www.youtube.com/watch?v=RrnkPrcpyEA but with tweaking to make it work with Python and additional constraint.

only able to control the acceleration of our system because we don't know anything about is mass or inertia.

State variables	control variables
x	ω
y	a
v	
θ	

Table 3.2: Type of variable of the Roomba

Now we will have to encode those equations into Casadi. To do so efficiently, we will use Casadi's function. This will have the quality of being a universal structure that will permit the standardisation of the solver.

It will require us to initiate all the variables as SX variable. For ease of storing, we will store the state variable in a SX array called *state* and the control variable in a SX array called *control*. From those two SX array, we will create a calculation template in a third SX array. and finally, create our Casadi's function out of those three SX array. That function will now provide us the derivative in time of all our state variables, if given as input the staking of *state* and *controle*. Meaning we have:

$$\begin{pmatrix} \dot{x} \\ \dot{y} \\ \dot{v} \\ \dot{\theta} \end{pmatrix} = function\_derive \begin{pmatrix} x \\ y \\ v \\ \theta \\ a \\ w \end{pmatrix} \tag{3.1}$$

For more information on how to create Casadi's function check A.1

3.3.2 Personalised constraints

This is also during this part that we will create our personalised constraint. In our case, those constraints regard the boundaries of the trajectory that the Roomba will have to follow. I decided to use Casadi's conditional functions. Here we will do a sinusoid, with a certain advancement k, that go into a straight after that c is higher than 4[m]. Of course, it is necessary that the boundaries are continuous and that the initial state is a valid solution.

$$f\_cont = \mathcal{F}(c,k) = \begin{cases} np.sin(2*k) & \text{if } c < 4 \\ 1 & \text{else} \end{cases} \tag{3.2}$$

For more information on how to create Casadi's conditional check A.2

3.3.3 Writing the solver

This is the second part of the programme that part as been standardised to require minimum manual adjustment (changing lbx, ubx, obj are the only requirement). That standardisation ask for some formalism in the parameters of the solver, seen in 3.1.2.

Parameter P

P is used to store the position of the Roomba for the current time step of simulation, the desired position of the Roomba and the time step of prediction. So the size of this parameter is $(2 \cdot \text{n\_state} + 1, 1)$ n_state being my number of state variable here 4.

$$P = \begin{bmatrix} P_0 & P_1 & P_2 & P_3 & P_4 & P_5 & P_6 & P_7 & P_8 \end{bmatrix} \tag{3.3}$$

And physically that array represented

$$P = \begin{bmatrix} x_i & y_i & \theta_i & v_i & x_f & y_f & \theta_f & v_f & T \end{bmatrix} \tag{3.4}$$

Where x_i is the initial position in x for the current time step of simulation and x_f is the desire position in x

Parameter X

For ease of programming, the parameter X of the solver has been separated in to two distinct parts. The first part is a SX array called X_state. That array is used to store the prediction of the solver for each state variable and for every step included step 0. Its dimension is thus (n_state, N+1) N being the number of prediction. Two things are worth noting. The first is that that array is not dimension 1 but 2 so we will have to change that when we reconstruct the parameter X of the solver to agree with Casadi's convention. Second thing, we clearly see that Casadi work vertically, so X_state with N = 3 is.

$$X\_state = \begin{bmatrix} X_0 & X_4 & X_8 & X_{12} \\ X_1 & X_5 & X_9 & X_{13} \\ X_2 & X_6 & X_{10} & X_{14} \\ X_3 & X_7 & X_{11} & X_{15} \end{bmatrix} \tag{3.5}$$

And physically that matrix represented

$$X\_state = \begin{bmatrix} x_0 & x_1 & x_2 & x_3 \\ y_0 & y_1 & y_2 & y_3 \\ \theta_0 & \theta_1 & \theta_2 & \theta_3 \\ v_0 & v_1 & v_2 & v_3 \end{bmatrix} \tag{3.6}$$

With x_0 being the state variable x at prediction step 0 so the initial position in x and x_1 being the state variable x at prediction step 1 so the first prediction for the state variable x.

The second part is a SX array called U. That array is used to store each prediction for each control variables, so its size is (n_control, N) n_control being the number of control variable here 2. U with N=3 is

$$U = \begin{bmatrix} U_0 & U_2 & U_4 \\ U_1 & U_3 & U_5 \end{bmatrix} \tag{3.7}$$

And physically that matrix represented

$$U = \begin{bmatrix} \omega_0 & \omega_1 & \omega_2 \\ a_0 & a_1 & a_2 \end{bmatrix} \tag{3.8}$$

At the end of the writing of the solver, X_state and U will be united under the named *opti_var* being a SX array of dimension 1 to satisfies the Casadi's convention. We will then give that SX array to the solver as the parameter X. This also mean that our first guess will have to be organised in the same way because it will be the signature of our solver. The organisation of *opti_var* when N=3 is

$$\begin{aligned} \mathrm{X} = opti\_var = [&X_0\ X_1\ X_2\ X_3\ X_4\ X_5\ X_6\ X_7\ X_8\ X_9\ X_{10}\ \dots \\ &X_{11}\ X_{12}\ X_{13}\ X_{14}\ X_{15}\ U_0\ U_1\ U_2\ U_3\ U_4\ U_5] \end{aligned} \tag{3.9}$$

Parameters lbx and ubx

lbx and ubx will apply the condition of the scalar they are given to the corresponding opti_var SX variable relative to the index (it is hard coded in Casadi). So lbx and ubx must have the same dimension as opti_var. They also need to be constructed manually for each model. lbx and ubx serve on putting condition like this:

$$lbx = \begin{bmatrix} -inf & -inf & -inf & -inf & -2 & .... \end{bmatrix} \tag{3.10}$$

$$ubx = \begin{bmatrix} inf & inf & inf & -inf & 2 & .... \end{bmatrix} \tag{3.11}$$

mean that the following condition must be true

$$-2 <= X_4 <= 2 \tag{3.12}$$

In this model lbx, ubx just put condition on ω and a giving them a maximum and minimum value.

Parameters g, lbg and ubg

The trio of SX arrays g, lbg and ubg all of dimetion 1. g will store the condition to satisfies. Meanwhile, lbg and ubg will be the bound of those conditions. In our multiple shooting methode, it is here that the work will be done. We have 3 type of constraint positional, cinematic and Personalised constraint in this model.

The positional constraint is straight forward to understand we want something to correspond exactly to something else, In our case we want our initial state (X_0, X_1, X_2, X_3) to be equal to our initial position(P_0, P_1, P_2, P_3) this translates to

$$g = \begin{bmatrix} X_0 - P_0 \\ X_1 - P_1 \\ X_2 - P_2 \\ X_3 - P_3 \\ \dots \end{bmatrix} \tag{3.13}$$

$$lbg = \begin{bmatrix} 0 & 0 & 0 & 0 & .... \end{bmatrix} \tag{3.14}$$

$$ubg = \begin{bmatrix} 0 & 0 & 0 & 0 & .... \end{bmatrix} \tag{3.15}$$

Physically the constraint mean

$$0 <= x_0 - x_i <= 0 \tag{3.16}$$
$$0 <= y_0 - y_i <= 0 \tag{3.17}$$
$$0 <= \theta_0 - \theta_i <= 0 \tag{3.18}$$
$$0 <= v_0 - v_i <= 0 \tag{3.19}$$
$$\tag{3.20}$$

Next we have the cinematic constraint. To establish those, we will use an Euler integration scheme and the function *function_derive* 3.1. The idea is that each new step must be exactly link to the last by its Euler integration.

$$g = \begin{bmatrix} ... \\ X_0 + \dot{X}_0 * P_8 - X_4 \\ X_1 + \dot{X}_1 * P_8 - X_5 \\ X_2 + \dot{X}_2 * P_8 - X_6 \\ X_3 + \dot{X}_3 * P_8 - X_7 \\ ... \end{bmatrix} \tag{3.21}$$

$$lbg = \begin{bmatrix} ... & 0 & 0 & 0 & 0 & ... \end{bmatrix} \tag{3.22}$$

$$ubg = \begin{bmatrix} ... & 0 & 0 & 0 & 0 & ... \end{bmatrix} \tag{3.23}$$

Physically, the constraint correspond to

$$0 <= x_0 + v_0 * cos(\theta_0) * T - x_1 <= 0 \tag{3.24}$$
$$0 <= y_0 + v_0 * sin(\theta_0) * T - y_1 <= 0 \tag{3.25}$$
$$0 <= \theta_0 + \omega_0 * T - \theta_1 <= 0 \tag{3.26}$$
$$0 <= v_0 + a_0 * T - v_1 <= 0 \tag{3.27}$$
$$\tag{3.28}$$

Finally the personalised constraint is our boundaries for the trajectory, so we will have to use *f_cont* 3.2. The objective is to constraint the value x and y such that they stay between an upper and a lower boundary.

$$g = \begin{bmatrix} ... \\ X_5 - f\_cont(X_4, X_4) \\ X_9 - f\_cont(X_8, X_8) \\ X_{13} - f\_cont(X_{12}, X_{12}) \end{bmatrix} \tag{3.29}$$

$$lbg = \begin{bmatrix} ... & -0.5 & -0.5 & -0.5 \end{bmatrix} \tag{3.30}$$

$$ubg = \begin{bmatrix} ... & 0.5 & 0.5 & 0.5 \end{bmatrix} \tag{3.31}$$

physically, those conditions correspond to

$$-0.5 <= y_1 - \begin{cases} np.sin(2 * x_1) & \text{if } x_1 < 4 \\ 1 & \text{else} \end{cases} <= 0.5 \tag{3.32}$$

$$-0.5 <= y_2 - \begin{cases} np.sin(2 * x_2) & \text{if } x_2 < 4 \\ 1 & \text{else} \end{cases} <= 0.5 \tag{3.33}$$

$$-0.5 <= y_3 - \begin{cases} np.sin(2 * x_3) & \text{if } x_3 < 4 \\ 1 & \text{else} \end{cases} <= 0.5 \tag{3.34}$$

Parameter obj

The cost function of that model is the following

$$obj = \sum_{i=1}^{N} \frac{i}{N} \sqrt{W_0(x_i - x_f)^2 + W_1(y_i - y_f)^2 + W_2(\theta_i - \theta_f)^2} \tag{3.35}$$

Where W_0, W_1 and W_2 are constant serving as weight parameters. That cost function tends to prioritise the end of the prediction, leading to a more aggressive controller.

Encoding the solver

After having defining everything, we can create the solver via the following commends

$$nlp\_struc = \{'x' : opti\_var, 'f' : obj, 'g' : g, 'p' : P\} \tag{3.36}$$

$$solver = cas.nlpsol('Solver', 'ipopt', nlp\_struc) \tag{3.37}$$

That solver will return us the prediction for all state variables and controls variables organised like in *opti_var*.

3.3.4 Integration loop

The next part of this model is the Integration loop. This part is responsible to make the Roomba advance in time. This model serving as a proof of concept, the integrator is a simple Euler explicit.

To start the loop, We will use the solver and all the bounds described in the previous chapter 3.3.3 and call it.

$$solution = solver(x0 = X0, p = P, lbx = lbx, ubx = ubx, lbg = lbg, ubg = ubg) \tag{3.38}$$

$X0$ being the prediction of the state variable and control variables from the previous time step. $X0$ serving us as first guess for the solver. *solution* will be the new prediction of the state variable and control variables, from which we can recuperate our new control variable.

Afterward, we can integrate as with any other multibody model for the current time step. And for the following time step, we update the starting position in P update $X0$ and repeat the process.

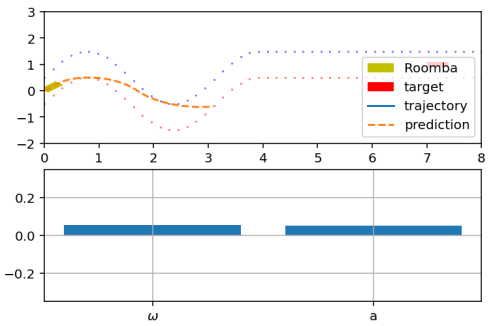
3.3.5 animation and results

Now that we have explained all the details of this model, we can have a look at the result and see in a more visual way how the solver work.

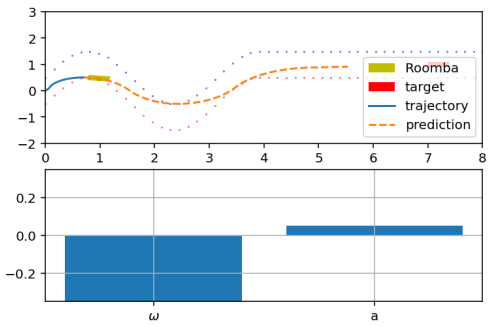
For the following simulation, we are using those parameters.

- N = 50
- T = δt = 0,25[s]
- The prediction scheme is Euler explicit
- The integrator scheme is Euler explicit
- The boundaries of the trajectory follow the equation 3.2
- a range is [-0,05,0,05][m/s^2]
- w range is [-20,20][degree/s] represented here by [rad/s]

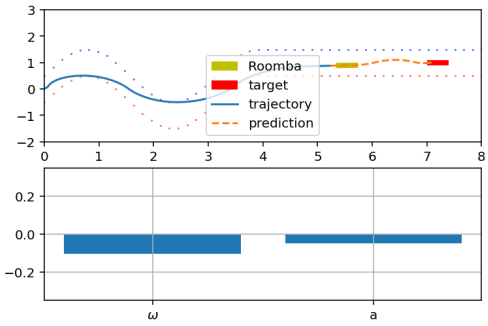
The following graphics are part of an animation where we collected the key frames. The upper part of each graphic represent the movement of the Roomba followed during the simulation. This movement take place in the 2D plane X-Y. In that part, the trajectory is represented by a plain line, meanwhile the prediction is a dashed line. And the boundaries ares the dotted lines. There is also the representation of the Roomba and the target position to achieve. As for the lower part, it represents the norm of our two control variables at the key frame.



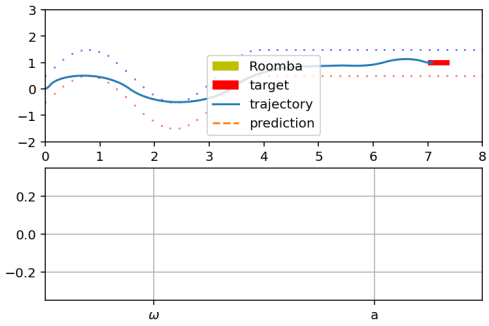
(a) Frame 0



(b) Frame 25



(c) Frame 75



(d) Frame 125

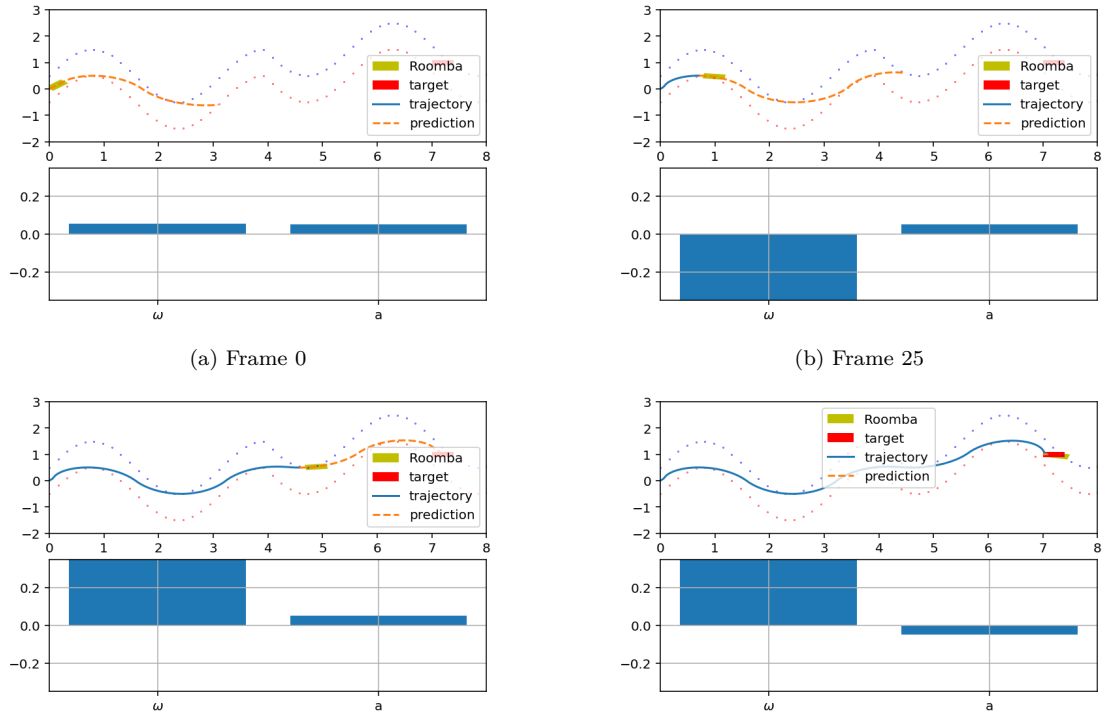
Figure 3.5: Animation Roomba

The first thing to observe is that the prediction correspond pretty well to the actual trajectory that the Roomba will follow. This is of course wanted and because we are using the same analytical equation of motion for both the prediction and integration we should obtain the same solution.
The second thing to note here is that the Roomba has done a light deport before reaching the target. This is because the Roomba is moving too fast and cannot stop in time with the range of acceleration we permit him to use. And because our cost function favouritise the end of the prediction, it seems that it was less costly to do that light deport than overshooting the target.

Because of the standardisation of the solver, it is very easy to change boundaries. We only need to modify the personalised constraints 3.2 and more precisely the function f_cont, and the solver will adjust the constraint accordingly. We will now use the following boundaries for the trajectory.

$$f\_cont = \mathcal{F}(c,k) = \begin{cases} np.sin(2*k) & \text{if } c < 4 \\ 1 + cos(2*k) & \text{else} \end{cases} \tag{3.39}$$

And as a result, we get the following graphics.



(a) Frame 0

(b) Frame 25

(c) Frame 75

(d) Frame 125

Figure 3.6: Animation Roomba

The solver manage to adapt to the new boundaries and give a satisfying solution. This is one of the quality of predictive controller, they will adapt to the boundaries they are given without

any manual change done to them. If we were using a PID with pure pursuit, we may have to modify the weight to make it work.

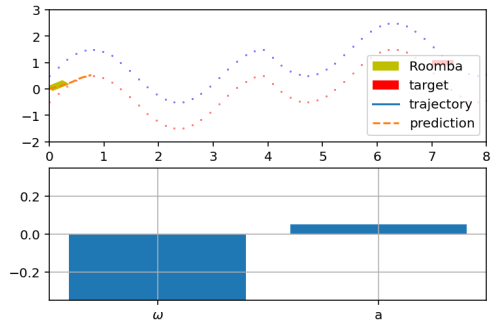
3.3.6 Experiment and observation

From that simple model, we have done some test to see how the solver would behave when we change some parameters inherent to its writing.

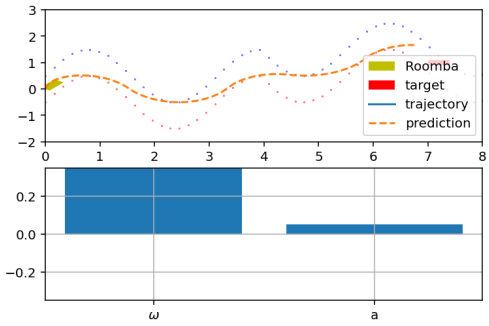
For now, we have been doing our simulation with 50 prediction steps(N), with a prediction time step of 0,25[sec](T) and an integration time step of 0,25[sec](δt)

The impact of the number of prediction step (N)

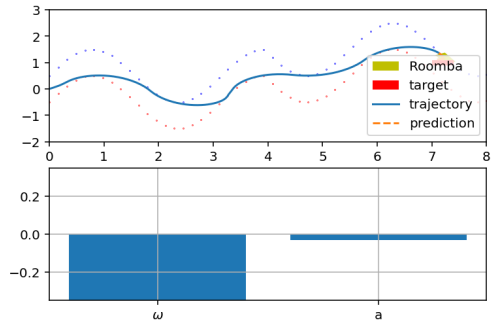
We will first try to change the number of prediction step. Because the more prediction step we have, the more time the solver will take to find a solution.



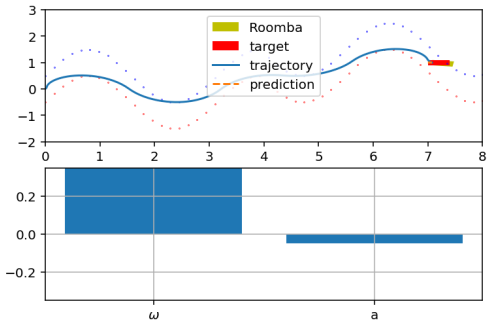
(a) Frame 0 and N = 25



(b) Frame 0 and N = 100



(c) Frame 125 N =25



(d) Frame 125 N = 100

Figure 3.7: Animation Roomba changing N

The two graphics on the left side are the graphic for 25 prediction steps. We can clearly see in the upper graphic that the horizon of the prediction is a lot shorter than previously. Meaning, the solver will not be able to anticipate as well the curves of the boundaries. And we see the result of that choice of in the lower graphic, with a more hesitant and less smooth trajectory. This has led the Roomba to not having yet reach his target after 125 time step.

Meanwhile, the two graphics on the right side use a prediction of 100 prediction steps. The prediction is thus logically longer, leading to a mush better trajectory. But the time of calculation is significatively increased.

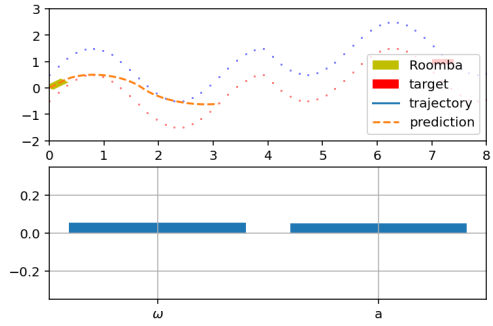
This lead to the conclusion that having more prediction step lead to better results and inevitably lead to a longer computations times. The number of prediction step, must thus be a compromise between the two. This why previously we settled for 50 prediction steps, and we can see that the difference between 50 prediction steps 3.6 and 100 prediction steps 3.7 is negligible.

Using a prediction time step (T) and a integration time step (δt) who are not equals

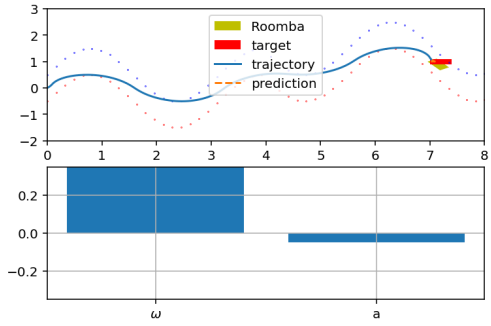
The idea behind this is that for the prediction, we want to have the longest horizon possible. Because as we saw in the previous subsection 3.3.6 this lead to better result. So if we can augment the horizon of the prediction without increasing the number of prediction step, we will obtain a better prediction without increasing the computation time. So we want our prediction time step to be as high as possible. We of course still need to be under the numerical stability of the model to calculate thing that are still physically correct. But the prediction doesn't need to be precise, it just needs to be correct enough.

Meanwhile, for the integration time step, we want it to be low because this lead to more precise integration. The numerical integration being imperfect by nature, the only way to keep the error of integration low is to have a low integration time step. This of course lead to a longer computation time, but this is a price we must pay to have a precise result at the end of the simulation.

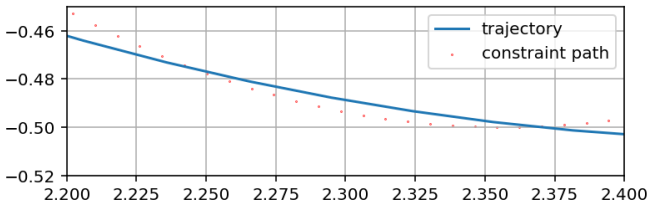
In an ideal world, we thus would want a high T and a low δt. But here is what happen when they are not equals. We have T = 0,25[sec] and δt = 0,1[sec]



(a) Frame 0



(b) Frame 300



(c) Closer zoom

Figure 3.8: Trajectory with T and δt who are not equals

In the two upper graphics, every thing seems to be has normal, but when we zoom on the curves we see a problem. It seems that the Roomba has cut has the corner. This is a small cut, but still it wasn't doing that before. That behaviour is due to the fact that the prediction and the integration will not have the exact same result with that trick. The prediction will still stay inside the boundaries because it has to, but the integration may cut the corner.
This is thus an effect to keep in mind when we are doing this trick to gain precision in our results. we may need to reduce our boundaries to allow our integrator to cut them a bit, because it is only a few centimetres. So depending on the application, this can be a worthwhile trick to use.

3.4 Inverse pendulum model

This model is the second model that have been done. This model consist of a chariot being able to move in a 2D plane and can accelerate in both horizontal directions independently. On top of this chariot, there is an inverse pendulum that can swing along the x-axis.

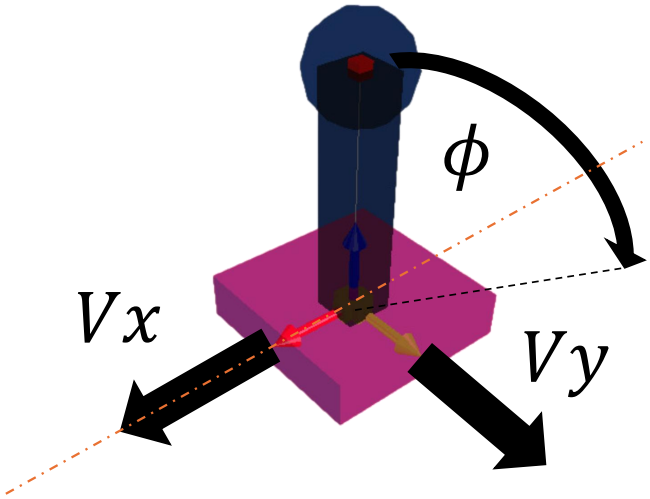


Figure 3.9: 3D representation of the inverse pendulum

The choice of an inverse pendulum was not done without reason.

- Being able to maintain in equilibrium an inverse pendulum, resembles maintaining a bike upright in corners.
- In addition, we will now be using the dynamic equation of motions because this is the type of equation Robotran will later give us.
- And finally for this model, we must use a better integrator scheme for the prediction and integration. Because an inverse pendulum is a periodic system, so using Euler explicit would not be physically correct any more.

This model use the same structure as the last one, so we will only talk about the changes.

3.4.1 physical model

The dynamic equations of motions that we will use are the followings:

$$\begin{aligned} \dot{x} &= v_x \\ \dot{y} &= v_y \\ \dot{\phi} &= \phi_p \end{aligned}$$

$$\dot{v_x} = F_x/(M+m)$$

$$\begin{pmatrix} \dot{v_y} \\ \dot{\phi_p} \end{pmatrix} = \begin{pmatrix} m+M & \frac{ML}{2}\cos\phi \\ \frac{ML}{2}\cos\phi & \frac{ML^2}{3} \end{pmatrix}^{-1} \left[\begin{pmatrix} F_y \\ 0 \end{pmatrix} - \begin{pmatrix} -\frac{ML}{2}\dot{\phi}^2\sin\phi \\ -\frac{ML}{2}g\sin\phi \end{pmatrix} \right]$$

Table 3.3: equation of motion Inverse pendulum

Where x and y stand for the position in a 2D plane in [m]. v_x is the speed in [m/s] in the x direction. v_y is the speed in [m/s] in the y direction. ϕ is the angle in [degrees] of the pendulum and the value 0 mean being straight-up. ϕ_p is the angular speed in [degrees/s]. F_x and F_y are both forces implied on the base either in the x or y direction in [N]. L is a constant and is the length of the pendulum in [m]. And finally, M and m are respectively the mass of the base and the masses of the pendulum in [kg].

As previously we can derive, from those equations 3.3, two type of variable, the state variable and the control variable.

State variables	**control variables**
x	F_x
y	F_y
ϕ	
v_x	
v_y	
ϕ_p	

Table 3.4: Type of variable Inverse pendulum

We don't need to have ours accelerations $(\dot{v_x}, \dot{v_y}, \dot{\phi_p})$ as state variables because we can find them back using our equation of motion 3.3. Plus, we are going to use more advance scheme like RK4 we will have 4 acceleration per prediction step. And finally because we do not have any constraint on them to satisfy there is no use storing their value for each prediction.

The equation of motion will be incorporate the same way as before, using a Casadi's function.

Writing the solver

That part like said preciously is standardised and so doesn't necessitate much adaptation for the new model. I will point out what need to be changed.

P will adjust itself without modifying the code

$$\begin{aligned} \mathrm{P} = [&P_0\ P_1\ P_2\ P_3\ P_4\ P_5\ P_6\ \dots \\ &P_7\ P_8\ P_9\ P_{10}\ P_{11}\ P_{12}] \end{aligned} \tag{3.40}$$

And physically that array represented

$$\begin{aligned} \mathrm{P} = [&x_i\ y_i\ \phi_i\ v_{xi}\ v_{yi}\ \phi_{pi}\ x_f\ \dots \\ &y_f\ \phi_f\ v_{xf}\ v_{yf}\ \phi_{pf}\ \Delta t] \end{aligned} \tag{3.41}$$

X_state will adjust itself without modifying the code .when N=3 X_state is:

$$X\_state = \begin{bmatrix} X_0 & X_6 & X_{12} & X_{18} \\ X_1 & X_7 & X_{13} & X_{19} \\ X_2 & X_8 & X_{14} & X_{20} \\ X_3 & X_9 & X_{15} & X_{21} \\ X_4 & X_{10} & X_{16} & X_{22} \\ X_5 & X_{11} & X_{17} & X_{23} \end{bmatrix} \tag{3.42}$$

And physically, that matrix represented

$$X\_state = \begin{bmatrix} x_0 & x_1 & x_2 & x_3 \\ y_0 & y_1 & y_2 & y_3 \\ \phi_0 & \phi_1 & \phi_2 & \phi_3 \\ v_{x0} & v_{x1} & v_{x2} & v_{x3} \\ v_{y0} & v_{y1} & v_{y2} & v_{y3} \\ \phi_{p0} & \phi_{p1} & \phi_{p2} & \phi_{p3} \end{bmatrix} \tag{3.43}$$

U will also adjust itself without modifying the code.

$$U = \begin{bmatrix} U_0 & U_2 & U_4 \\ U_1 & U_3 & U_5 \end{bmatrix} \tag{3.44}$$

And physically that matrix represented

$$U = \begin{bmatrix} F_{x0} & F_{x1} & F_{x2} \\ F_{y0} & F_{y1} & F_{y2} \end{bmatrix} \tag{3.45}$$

g will also adjust itself without modifying the code. As explain earlier, the prediction is made using a RK4 scheme, thus the dynamic constraint for g will be different and a lot less readable. The positional and personalised constraint regarding g, lbg and ubg are the same. Here is as an example of the constraints. Those constraints are linking step 1 to step 0 of the prediction.

$$g = \begin{bmatrix} \dots \\ (x_1 - (x_0 + RK4(v_x([x_0, y_0, \phi_0, v_{x0}, v_{y0}, \phi_{p0}, F_{x0}, F_{y0}])))) \\ (y_1 - (y_0 + RK4(v_y([x_0, y_0, \phi_0, v_{x0}, v_{y0}, \phi_{p0}, F_{x0}, F_{y0}])))) \\ (\phi_1 - (\phi_0 + RK4(\phi_p([x_0, y_0, \phi_0, v_{x0}, v_{y0}, \phi_{p0}, F_{x0}, F_{y0}])))) \\ (v_{x1} - (v_{x0} + RK4(\dot{v_x}([x_0, y_0, \phi_0, v_{x0}, v_{y0}, \phi_{p0}, F_{x0}, F_{y0}])))) \\ (v_{y0} - (v_{y0} + RK4(\dot{v_y}([x_0, y_0, \phi_0, v_{x0}, v_{y0}, \phi_{p0}, F_{x0}, F_{y0}])))) \\ (\phi_{p1} - (\phi_{p0} + RK4(\dot{\phi_p}([x_0, y_0, \phi_0, v_{x0}, v_{y0}, \phi_{p0}, F_{x0}, F_{y0}])))) \\ \dots \end{bmatrix} \tag{3.46}$$

opti_var will also adjust itself without modifying the code.

$$\begin{aligned} \mathrm{X} = opti\_var = [&X_0\ X_1\ X_2\ X_3\ X_4\ X_5\ X_6\ X_7\ X_8\ X_9\ X_{10}\ \dots \\ &X_{11}\ X_{12}\ X_{13}\ X_{14}\ X_{15}\ X_{16}\ X_{17}\ X_{18}\ X_{19}\ \dots \\ &X_{20}\ X_{21}\ X_{22}\ X_{23}\ U_0\ U_1\ U_2\ U_3\ U_4\ U_5] \end{aligned} \tag{3.47}$$

To implement the constraint that the inverse pendulum must stay upright. It is fairly simple, we must make sure that those constraints are met.

$$-20^\circ <= X_2 <= 20^\circ \tag{3.48}$$
$$-20^\circ <= X_8 <= 20^\circ \tag{3.49}$$
$$-20^\circ <= X_{14} <= 20^\circ \tag{3.50}$$
$$-20^\circ <= X_{20} <= 20^\circ \tag{3.51}$$
$$\tag{3.52}$$

This is typically the type of constraints store in lbx and ubx corresponding to the organisation of *opti_var* as explain in section3.3.3.

And lastly, the cost function of that model is the following

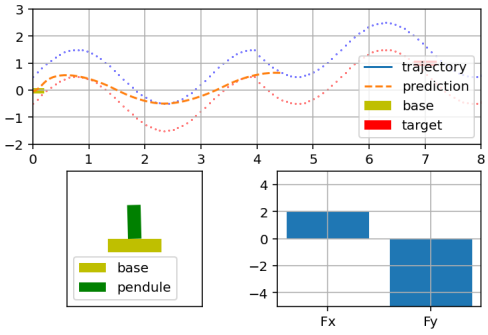
$$obj = \sum_{i=1}^{N} \frac{i}{N} \sqrt{W_0(x_i - x_f)^2 + W_1(y_i - y_f)^2} \tag{3.53}$$

It is very similar to the cost function 3.35 of the previous model.

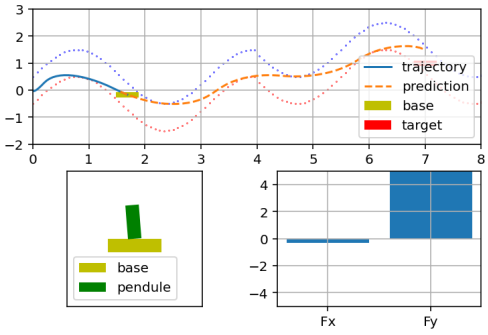
3.4.2 animation and results

We will now talk about the result of the simulation we made. For the following simulation, we are using the following parameters

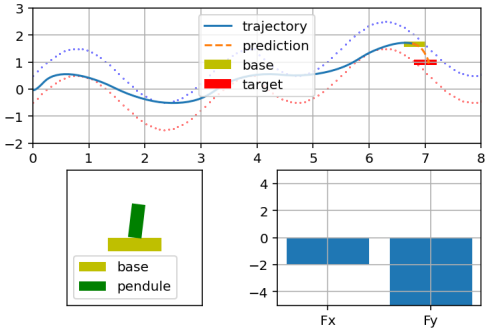
- N = 50
- T = δt = 0,1[s]
- The prediction scheme is RK4
- The integrator scheme is RK4
- The boundaries of the trajectory follow the equation 3.39
- F_x range is [-2,2][N]
- F_y range is [-4,4][N]
- ϕ range is [-20,20][degrees]



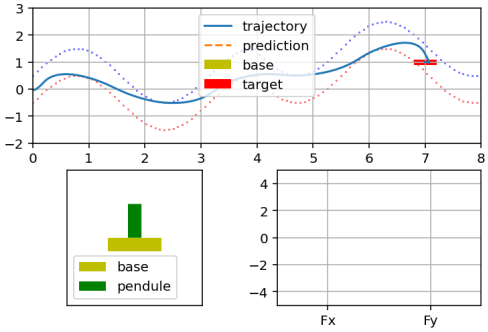
(a) Frame 0



(b) Frame 25



(c) Frame 75



(d) Frame 125

Figure 3.10: Animation Inverse pendulum

Those four graphics represents the key frame of our inverse pendulum. The upper part and lower right part of each graphic have already been explained here 3.5 but the lower left part is new. That new part of the graphic represent a view of the inverse pendulum, with its base and pendulum are seen from the back. Meaning that from this view, the vector X is going through this thesis. The view serve to check if the pendulum is still upright. But to be sure that the pendulum angle stay between the boundaries we gave it, we need a better graphic.

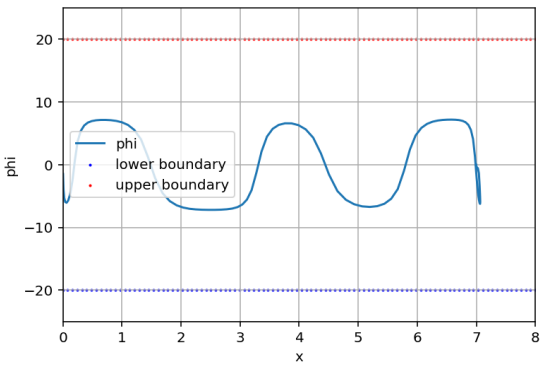


Figure 3.11: ϕ in function of x

This graphic 3.11 show us that ϕ is never near its boundaries, meaning that there is another factor limiting it. This is obviously F_y and or F_x because we bound their values fairly low. Meaning that the predictive solver is incapable of been too aggressive with the range of value we gave it.

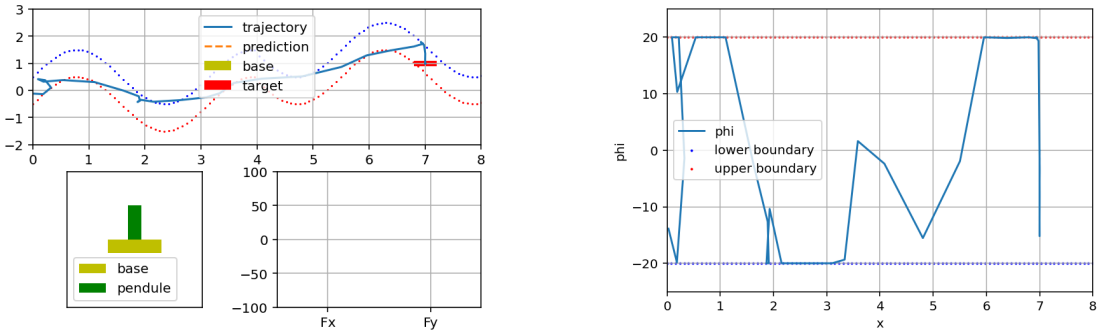
3.4.3 Experiment and observation

Now that we have seen that we have a working model for our inverse pendulum, we will talk about the different experiment we have done and the information we can extract from them.

Removing the limit on our control variables

In the previous section, we stated that the limiting factor for ϕ was F_x and F_y. So what happen when we don't limit our controls variables. The answer to that question is a bit disappointing, we teleport to the target and in 5 frame we stabilise on it. Because obviously the fastest way to reach a target is teleportation and if we allow infinite forces and thus infinite acceleration it is what happen. It was also the case in the Roomba models.

But now come the question if we limit our control variable to finite but enormous boundaries what happen. we will limit the forces between [-100,100][N] the rest of the simulation parameters 3.4.2 are the same.



(a) Frame 15

(b) ϕ in function of x

Figure 3.12: Trajectory with very loose boundaries on the control variables

As we can see, we still teleport, but this time it is not instant, it takes a few frames. At least we can check on the right graphic that the boundaries on ϕ work as intended, even when we are teleporting. We can also observe that by teleporting, the trajectory can cut through the boundaries. This is because we are only checking if the end of each time step end within the boundary. Because we work with time step, we are always doing jumps. When it is a small jump, this approximation is good. But when we are jumping up to two meters in one time step, this is no longer a good approximation.

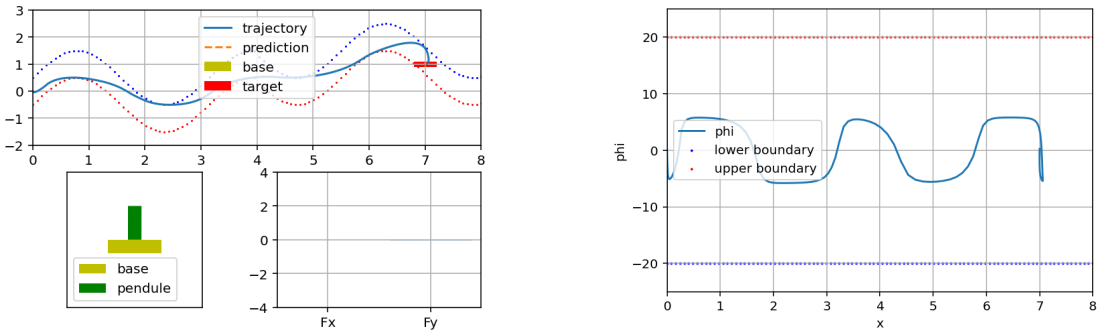
A solution to this would be to reduce T and δt in order to limit the range the solver could jump and thus having a more physically correct answer. But this would also diminish the horizon of the prediction, so not the best solution.

We could also show here that if we put boundaries on the control variables that are too low, it will become impossible for the predictive solver to find a prediction that will go to the target. Because it would be impossible to do the trajectory. Which is totally logical and expected. The solver should not be able to do impossible things.

The best solution is thus to have realistic boundaries on our control variables.

Working with different integrators schemes

So far we have always work with RK4 in this model because we know from the theory of numerical integrator that Euler explicit scheme is unstable for periodic systems. So it will not be physically correct to use, but we can always try. Because Euler explicit is of course a faster scheme than RK4, so if it works we could continue using it. The following simulation is done using the same parameters as in 3.4.2 but with Euler scheme for both the prediction and integration



(a) Frame 125

(b) ϕ in function of x

Figure 3.13: Animation Inverse pendulum with Euler explicit

That result is very interesting, it seems that the predicative solver is able to overcome the numerical instability of the Euler explicit. And manage still to find a solution. This mean that our solver is able to stabilise even numerically unstable models. But how wrong are we when we use Euler explicit scheme, let's compare the two.

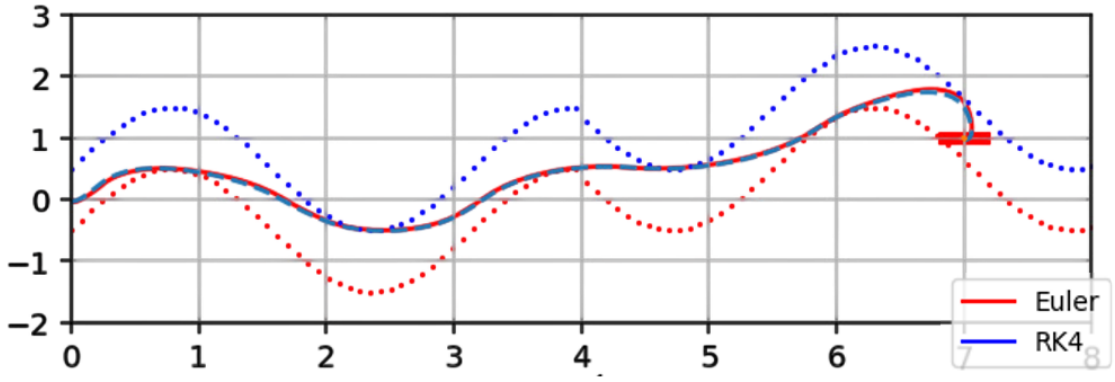
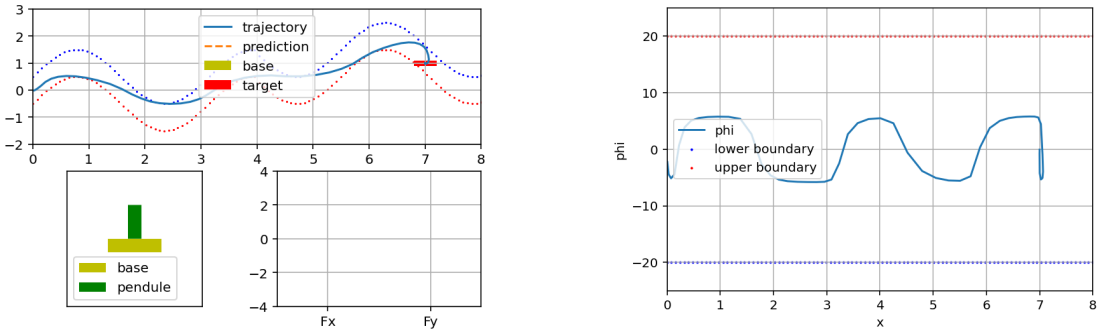


Figure 3.14: Comparison RH4-Euler eplicit

As we can see, the two result are somewhat similar. But the difference is still visible. And of course when we are doing simulations we want to have the more precise results so we will use RK4.

But we can try using Euler implicit. Euler implicit has the quality of being numerically stable whatever the problem, this could lead to better results.



(a) Frame 125

(b) ϕ in function of x

Figure 3.15: Animation Inverse pendulum with Euler implicit

We can see that the solver can also handle implicit constraints and converge roughly using the same trajectory as RK4. Plus there is something to note here, it is the fastest the predictive solver has ever converged, by a great margin. This is probably due to the over numerical stability that Euler implicit has, making it easy for Casadi to find the solution. But we must still compare the two.

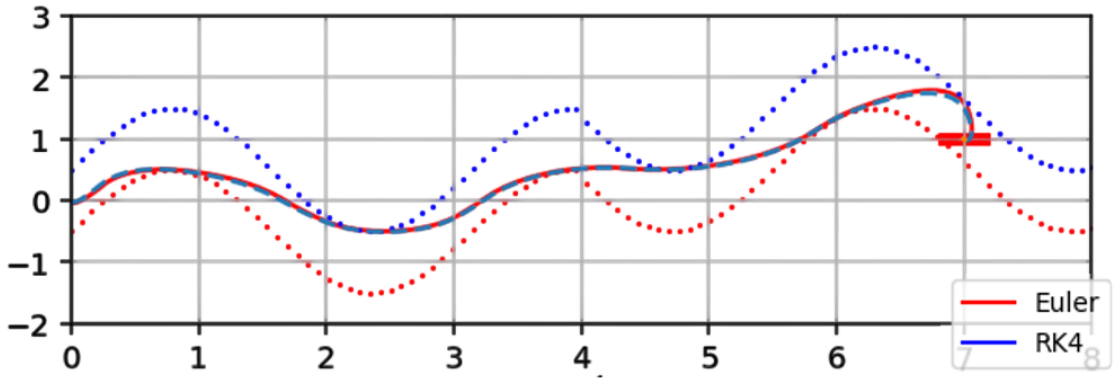


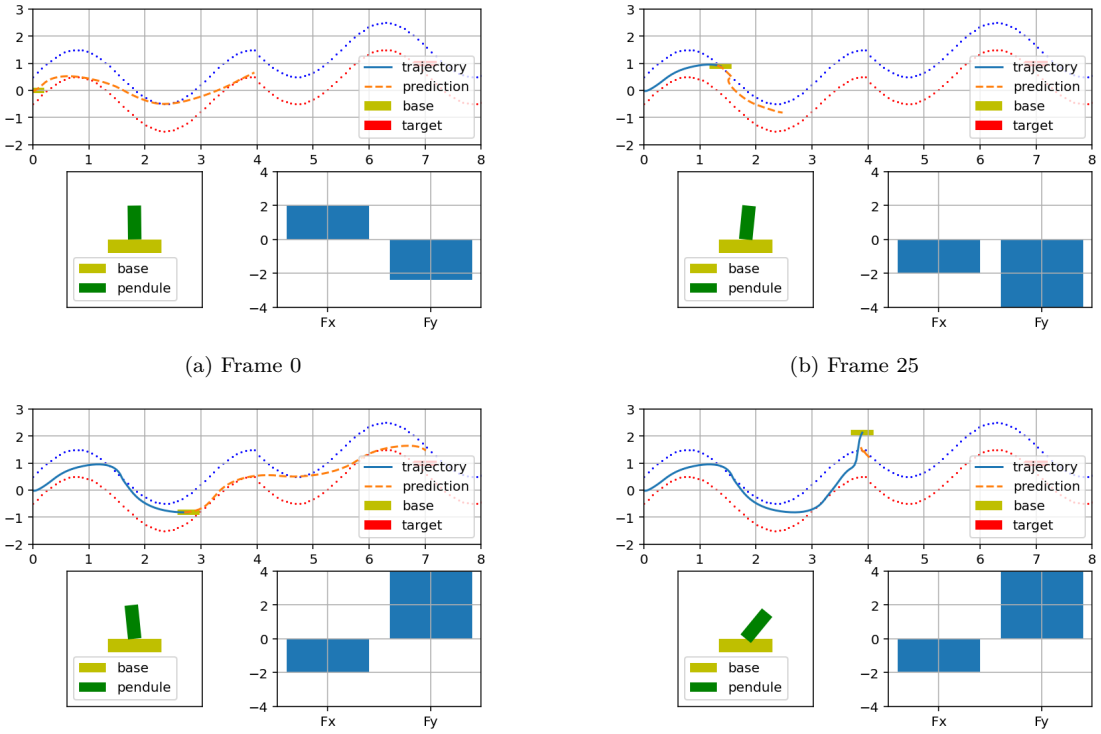
Figure 3.16: Comparison RH4-Euler implicit

The error between the two is still visible, of course we could reduce that error by making δt lower but Rk4 is a better scheme and if we want our model to be precise we must pay the price in computation time.

Prediction using Euler implicit and integrator using RK4

With everything, we experiment to this point, we will make something a little bit creative. So far we have been using the same scheme for the predicative solver and the integrator because this will make both of them perfectly agree with one another. But our prediction doesn't need to be precise, so maybe we could use a less precise scheme than the integration. And maybe our controls variables could still be correct enough. The idea behind this is to lower the computation time of the predicative solver while keeping the precision of the integrator intact.

For the following simulation, we will use RK4 for the integrator to keep the precision high and the predicative solver will use Euler implicit because as we said in the last subsection it was extremely fast. The rest of the parameters are the same as in section 3.4.2.



(a) Frame 0

(b) Frame 25

(c) Frame 50

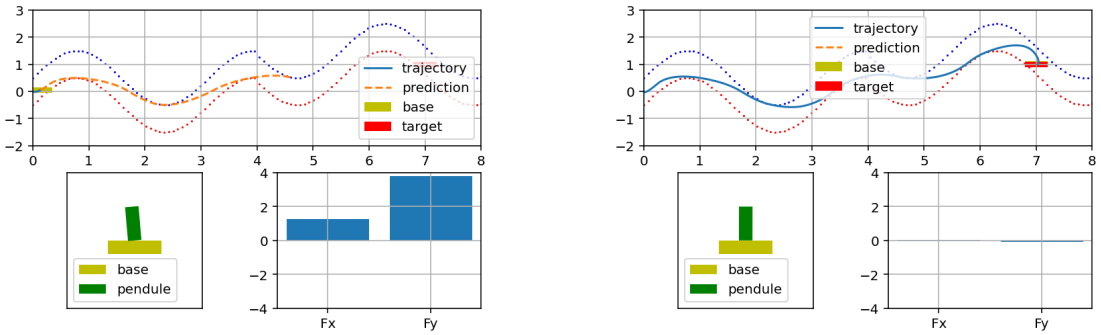
(d) Frame 75

Figure 3.17: Animation Inverse pendulum prediction using Euler implicit and integration using RK4

As we can see, this doesn't seem to work. We can see that the prediction that the base was supposed to follow in frame 0 is clearly not the trajectory the base followed in frame 50. The error between the prediction and the integration seem to be too great, and the predictive solver don't seem to recuperate those errors. We can conclude that in that case, the prediction and integration disagreed to mush with one another for the controller to work. At least this failure enable us to show that even when predictive solver failed, the simulation stays physically correct and don't explode.

But we could degrease the error between the two by using a lower δ and T but to keep a good horizon we will increase N. For the following simulation we will use those parameters.

- N = 500
- T = δt = 0,01[s]
- The prediction scheme is Euler implicit
- The integrator scheme is RK4
- The boundaries of the trajectory follow the equation 3.2
- F_x range is [-2,2][N]
- F_y range is [-4,4][N]
- ϕ range is [-20,20][degrees]



(a) Frame 75

(b) frame 1250

Figure 3.18: Animation Inverse pendulum experimentation

As you can see the idea work, sadly by multiplying N by 10 we have greatly augmented the time of computation and in the end the final result is similar to our first simulation 3.10. So it is not worth it. Plus, by having the predictive solver disagree with the integrator led to the first guess of prediction being bad, leading in return to more time being taken by the solver to re-converge to a new prediction. In other word, it is a good idea on paper, but it don't seem to work as intended.

This is why, knowing that Robotran use a RK4 scheme for its integrator, we will only be using that scheme for the rest of this thesis.

Chapter 4

Intermediary Numerical models using Robotran symbolic generation

4.1 Robotran

Robotran is an environment developed at UCLouvain capable of generating and computing multibody system. That environment is separated in 3 main blocs. Each one of them having its purpose.

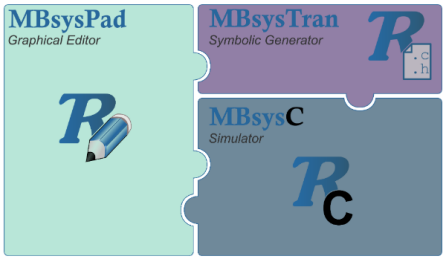
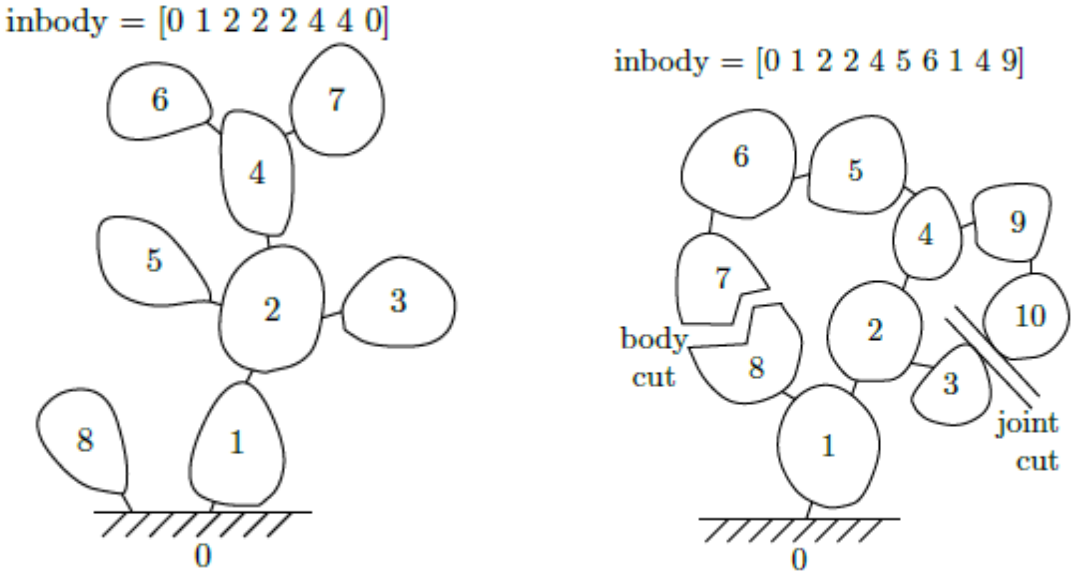


Figure 4.1: Robotran organisation

4.1.1 MBsysPad and multibody formalism

To start any project in Robotran we must first draw it in MBsysPad. All multibody systems in Robotran are described as an assembly of rigid bodies which are connected together by rotoid or prismatic joints. Those joints will then describe the possible motion between the bodies. There are two main type of topology, the tree-like topology or the close-loop topology.



(a) tree-like topology

(b) close loop topology

Figure 4.2: Comparison between the two topology

In our case, all our models will not have any cinematic loop, our topology will thus be tree-like. This choice has been made in others to simplify the binding of the predictive solver and Robotran. Because tree-like system have an easier equation motion and are easier to solve. But not all multibody system can be express in a tree-like topology. That representation is thus limited.

For the following of this thesis, We will only thus talk about tree-like topology, and what is explained here won't apply to close-loop topology. If you desire more information on close-loop system, you can check this website[1].

After having created our multibody system, We can generate the equations of motion4.1 describing the dynamics of our system in the following semi-explicit form.

$$M(q)\ddot{q} + c(q, \dot{q}, g) = Q(q, \dot{q}) \tag{4.1}$$

Where:

- M is the symmetric-generalized mass matrix of the system.
- c is the nonlinear dynamic vector that contains the gyroscopic, centrifugal, and gravity terms as well as the contribution of external resultant forces and torques, trq.
- q is a vector denoting the relative generalized coordinates.
- $\dot{q}$ is a vector denoting the relative generalised velocities.
- $\ddot{q}$ is a vector denoting the relative generalised accelerations.
- Q is a vector representing the generalised force and/or torques in the joints

[1] https://www.robotran.be/docs/documentation/ Robotran basic

This set of differential algebraic equation (DAE) is the most generalised way of portraying multibody systems. But those are not the most usable equations. To obtain those more usable equation, we need to introduce MBsysTran.

4.1.2 Robotran MBsysTran

MBsysTran is responsible for generating the symbolic equations of motions base on our multibody system we drew in MBsysPad. The generation of the symbolic equations will be a key component of our future models because depending on the 2 following representations, the solving logic of our controller will be differently. The controller having to obtain, in both case, the acceleration of the joints but will have to obtain them differently.

Implicit equations of motions

The implicit equations of motions is the first representation we will use. It is also the most compact representation of the equations of motion.

$$\Phi(q, \dot{q}, \ddot{q}) = Q \tag{4.2}$$

The implicit form can be used when we don't explicitly need to know the mass matrix. This is also the fastest equation to generate using MBsysTran. But will require, from our controller, to solve these DAE to obtain each acceleration.

Explicit equation of motions

This second way of representing the equations of motions require more calculation to obtain. Because we need to convert the DAE into an ordinary differential equation (ODE).

$$M(q)\ddot{q} + F_r(q, \dot{q}, Q) = 0 \tag{4.3}$$

This will demand MBsysTran to inverse a mass matrix symbolically to isolate the $\ddot{q}$ and obtain the explicit equation of motions.

$$\ddot{q} = -F_r(q, \dot{q}, Q) * M^{-1}(q) \tag{4.4}$$

For a better understanding of that form, we can recall that it was the form we were using in 3.3 for the analytical inverse pendulum model.

4.1.3 Robotran MBsysPy

The last part of the Robotran environment is MBsysPy (or MBsysC When programmed in C). This part is responsible for the simulation of our multibody system in time. It as the same role as what we previously referred to as "the numerical model and integrator"in the shema 3.3. Our predicative solver will thus need to agree with MBsysPy on how our models behave. We will thus now have a look at how MBsysPy normally work with tree-like topology.

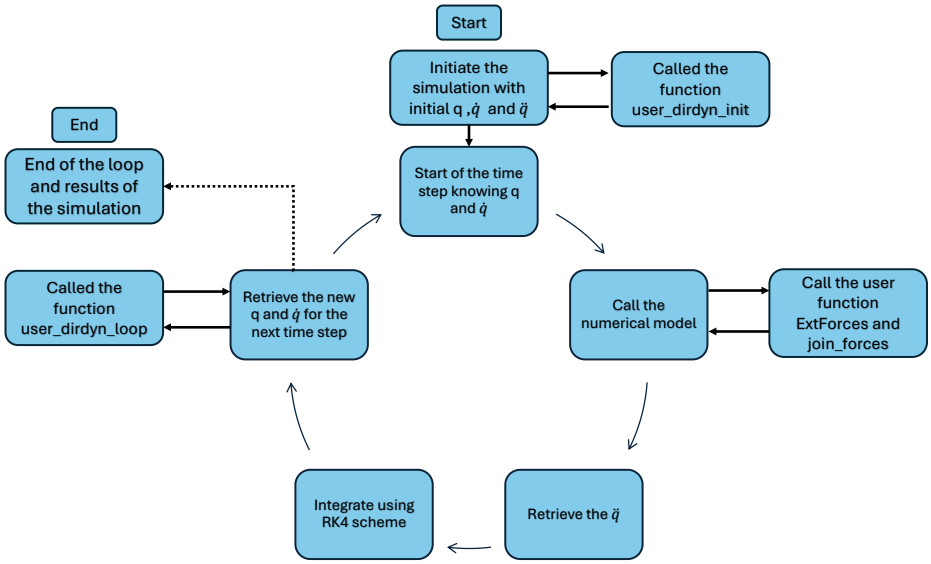


Figure 4.3: Inner working of MBsysPy

The simulation start with us knowing the initial state $(q, \dot{q}, \ddot{q})$. When the simulation start, MBsysPy will call a function name *user_dirdyn_init* that function will have a use for us later, but for now it is useless.

Now we enter the MBsysPy loop with our initial time step. MBsysPy call the numerical model created by MBsysTran and during the computation of the model it will call the function *user_ExtForces* and *user_joint_forces* where the user will programme the forces acting on the model.

Then MBsysPy retrieve from the numerical model the generalised acceleration($\ddot{q}$) of the current time step and will thus integrate it in others to obtain the generalized coordinates(q) and the generalised velocities($\dot{q}$) of the following time step. And just before MBsysPy continues the loop with the following time step, it will call a function name *user_dirdyn_loop*. And again, that function will have a use for us later, but for now it is useless.

The cycle repeat until we have simulated all the desire time step. After that it break the loop, and we obtain the result of the simulation, in other for us to analyse them.

4.2 Implementation of Casadi in Robotran

In this section, we will talk about how we implemented both Robotran and Casadi in the same time and how do they interact with one another.

4.2.1 Overview from a user point of view

In this section, we will have a look at how Casadi interact with Robotran. This look will explain how a user will perceive the binding of Casadi and Robotran. It will thus not go into detail but will explain how we can create a project binding the two.

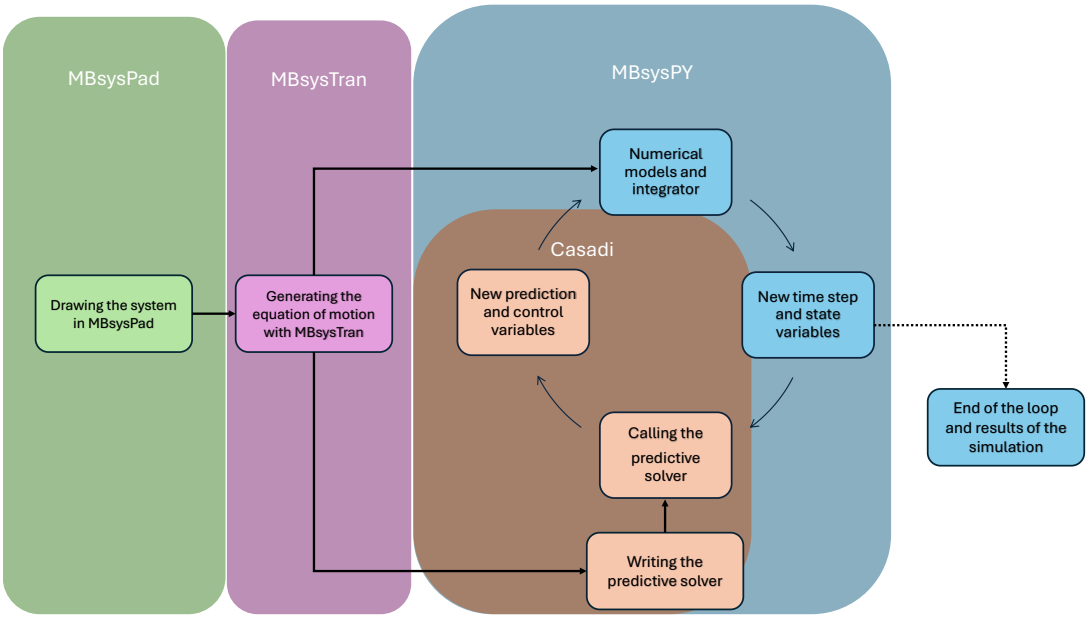


Figure 4.4: user visualisation of the process

As we can see, the start of creating a project in Robotran stays exactly the same. we draw our model in MBsysPad and generate the equation of motion using MBsysTran. The moment it starts to deviate is inside MBsysPy. The first thing the user will have to do is to generate the equation of motion twice, once for MBsysPy and once for Casadi. This is done using an inbuilt option in MBsysTran.

The other thing to notice is that all of Casadi computation will happen inside MBsysPy. Casadi will be programme inside the already existing user functions of MBsysPy and will thus not need an external programme nor script to function.

4.2.2 Writing the solver using MBSdata plus what will the solver control

Before in our analytical models we had to define every parameter manually, because we had to construct our multibody system ourselves. But now, by having access to the MBSdata structure, we can know every thing there is to know on our multibody model. We can thus take advantage of this and use that data structure to standardise even more.

Per example before in our analytical models, the organisation of X_state 3.3.3 depended on $function\_derive$ 3.1 who was done manually. But now we have a specific order for each of our joint (state variables) and our $function\_derive$ will be created by MBsysTran following the same

order. So X_state will also now have the same order as in MBSdata. For more clarification, see A.3 Of course those standardisations does not change the computational logic of our solver, they just reduce the manual input that is required to change models.

But the control variables can not been standardised, we still need to explicitly state what are we controlling in our model. In our case, we are going to control joint forces and/or external forces.

The joint forces are forces or torque that will be applied on a joint. Per example, to guide our bike, we need a torque on the joint of the handlebar to make it turn. Those are simple to introduce to Casadi and don't require mush work to implement.
Meanwhile, the external forces are forces and/or torque applied on a certain point of a body. Per example, when a bike ride the ground exerts force on the wheels of the bike those forces are applied on the contact point of the wheel. Those are harder to implement in Casadi for reasons we will explain later, but we will use them in our final model.

4.2.3 Using the predictive solver in MBsysPy loop

Now that we saw how MBsysPy work normally, where Casadi will apply in the user process and what the solver will control, It is time to have a closer look on how Casadi and MBsysPy interact together. And for that we will use the following schema as illustration.

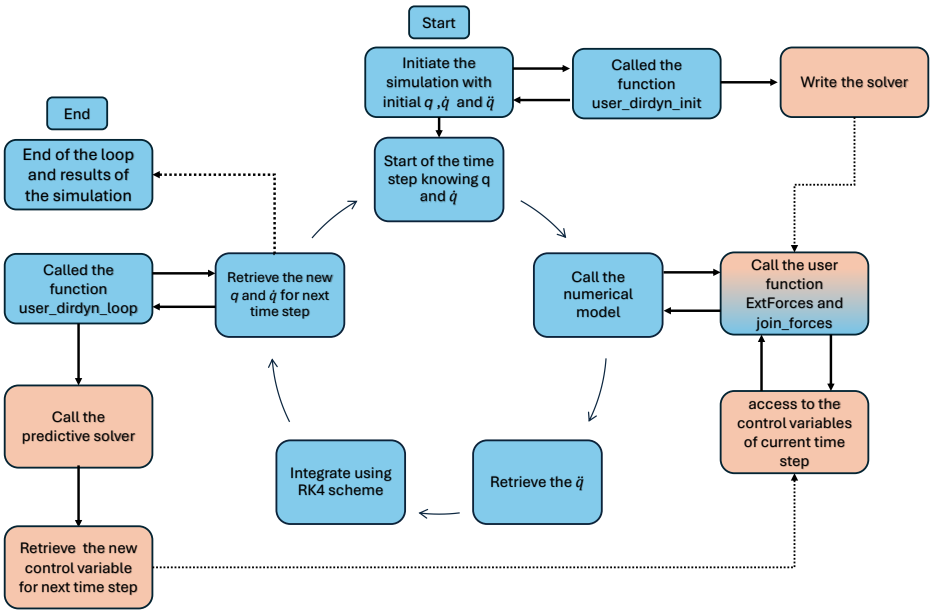


Figure 4.5: Inner working of MBsysPy with Casadi

This schema is a direct call back from 4.3, and thus we will only talk about the changes.

At the start of the simulation, MBsysPy will call $user\_dirdyn\_init$ that function will be where we write our predictive solver using the equations of motion generated by MBsysTran. Those equations of motion will have to call the function $user\_ExtForces$ and $user\_joint\_forces$ to fully compile. We must thus make those function in a way that work both with SX variables

for Casadi and "INT" for MBsysPy. The solution is to temporary transform the "INT" into DM variables 3.1.1 to make them compatible with Casadi's function, like the SX variables, and then to transform the DM variables back into "INT" for MBsysPy. This process is a bit tedious but necessary.

After writing the predictive solver, we enter in MBsysPy loop. Here, the numerical model will call the $user\_ExtForces$ and $user\_joint\_forces$ and because this could be our control variable, they also need to be able to access the result of our predictive solver when needed.

And finally at the end of each time step in the function $user\_dirdyn\_loop$ we call the predictive solver, using our last solution as first guess, to find new control variables. And afterward we must store them to be able to use them for the following time step when $user\_ExtForces$ and $user\_joint\_forces$ are called.

If you correctly understand the flow of computation, you would have noticed that for the initial time step the model is not control. And this is totally the case during the first time step we didn't have called the predictive solver yet and thus all control variables still have their initial value "0". Obviously, after the first time step, the model is controlled, but this is a small limitation to keep in mind.

4.3 Generating equation of motions for Casadi using MBsysTran

In this section, we will explain in further details the link between MBsysTran and Casadi. Because for now, we have only be saying that MBsysTran is generating the equation of motions for our predictive solver, but what does that implied and what does that change.

4.3.1 Accelred/explicit formulation of the equation of motion

Accelred is the name of the option in MBsysTran that giver us the explicit form of the equation of motion 4.4. That form give us directly the acceleration of our model if we know the generalized coordinates(q) and the generalised velocities($\dot{q}$) which are our state variables in Casadi.

This mean that Casadi can easily calculate the acceleration we need in order to put our dynamic constraint in the vector of constraint(g) like explain in equation 3.21. Our predictive solver works thus exactly the same as in our analytical model of the inverse pendulum section 3.4. And this is expected because in that model we were already using the explicit form of the equations 3.3.

The only thing we need to do when using the option Accelred is to manually make the equation of motions compatible for SX variables. But this is something that only need to be done once for each model, so totally manageable by hand.

4.3.2 Impdynared/implicit formulation of the equation of motion

Impdynared is the name of the option in MBsysTran that giver us the explicit form of the equation of motion 4.2. That form don't give us the acceleration directly but give a set of differential algebraic equation(DAE) to solve instead. This mean that Casadi will have to solve first that set

of DAE and then put the dynamic constraint in the vector of constraint (g). This will ask us to add a new set of constraint in g called the algebraic constraint that Casadi will have to satisfy. Sadly, this also mean that with this technique we are putting constraint on our acceleration meaning we have to store them in X 3.1.2 and because we are using RK4 as a predictor scheme we must store 4 acceleration for each joint and for each time step. For a more numerical explanation, see appendix A.4

4.4 Inverse pendulum model

As a first model using both Robotran and Casadi we decided to recreate the inverse pendulum model in section 3.4. This choice was made to verify that our model using Robotran would have the same result as with our analytical model. Plus, we will use two different equations of motion and thus two different solving logics, so having an already existing model to compare is a good idea.

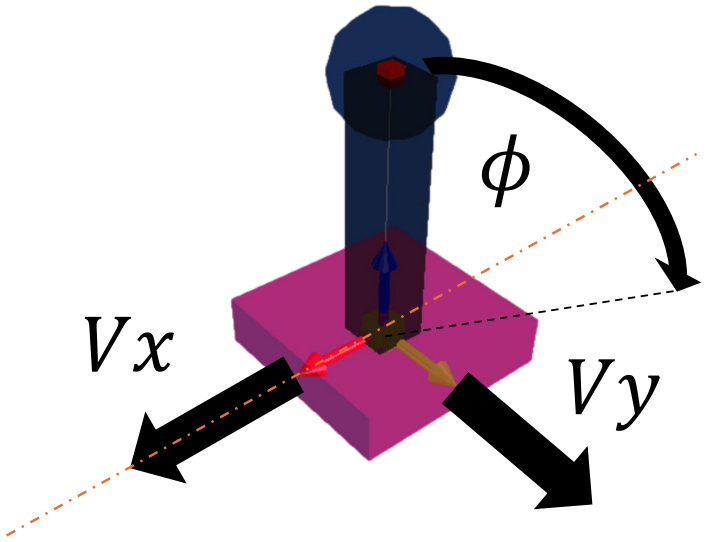


Figure 4.6: 3D representation of the inverse pendulum

Of course, by being the same model as preciously we will control the same forces and thus the vector u will be also the same as in section3.4. And those type of forces in Robotran are referred as $joint\_forces$ as explain in subsection4.2.2. Meaning that in this model, we are going to control 2 $joint\_forces$.

4.4.1 Accelred model

In this section, we will talk about the model using the equation generated by the option Accelred. When we call MBsysTran using the option Accelred the equation we are getting is the following.

$$\dot{x} = v_x$$
$$\dot{y} = v_y$$
$$\dot{\phi} = \phi_p$$

$$\dot{v_x} = F_x/(M+m)$$

$$\begin{pmatrix} \dot{v_y} \\ \dot{\phi_p} \end{pmatrix} = \begin{pmatrix} m+M & \frac{ML}{2}\cos\phi \\ \frac{ML}{2}\cos\phi & \frac{ML^2}{3} \end{pmatrix}^{-1} \left[\begin{pmatrix} F_y \\ 0 \end{pmatrix} - \begin{pmatrix} -\frac{ML}{2}\dot{\phi}^2\sin\phi \\ -\frac{ML}{2}g\sin\phi \end{pmatrix} \right]$$

Table 4.1: equation of motion Inverse pendulum by Accelred

Where x and y stand for the position in a 2D plane in [m]. v_x is the speed in [m/s] in the x direction. v_y is the speed in [m/s] in the y direction. ϕ is the angle in [degrees] of the pendulum and the value 0 mean being straight-up. ϕ_p is the angular speed in [degrees/s]. F_x and F_y are both forces implied on the base either in the x or y direction in [N]. L is a constant and is the length of the pendulum in [m]. And finally, M and m are respectively the mass of the base and the masses of the pendulum in [kg].

As explain in subsection4.1.2, when we use the option Accelred we are getting the explicit form of the equation of motion, and thus we obtain the same equation of motion as in our analytical model3.4. And this is totally expected because the equation in our analytical model were also the explicit equation of motion. This also implied that the solving logic will be the exact same as before. The only difference is that now our equation are generated by MBsysTran and the integration is also done by MBsysPy as explained in schemas 4.4 and 4.5.

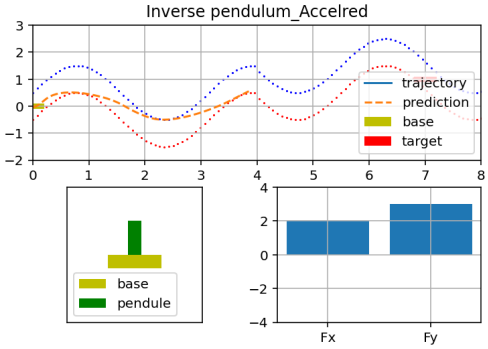
Graphics and results

We can thus directly talk about the results of our simulation, using the same representation as previously for our trajectory.

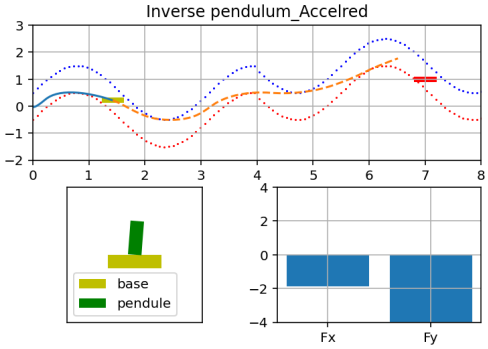
For the following simulation, we use those parameters.

- N = 50
- T = δt = 0,1[s]
- The prediction scheme is RK4
- The boundaries of the trajectory follow the equation 3.2
- F_x range is [-2,2][N]
- F_y range is [-4,4][N]
- ϕ range is [-20,20][degrees]

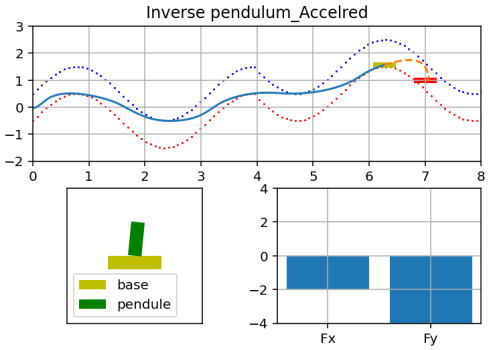
We can note that because we are using MBsysPy for the integration in time, our integration scheme will always be RK4. So there is no need to specify it any more.



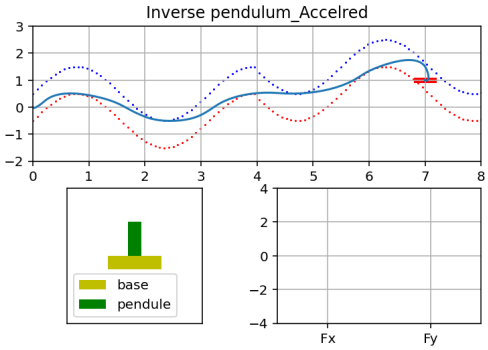
(a) Frame 1



(b) Frame 25



(c) Frame 75



(d) Frame 125

Figure 4.7: Animation Inverse pendulum using Accelred

With this simulation, we can see that we manage to implement our previous predicative solver into Robotran.

4.4.2 Impdynared model

In this section, we will interest ourselves with a new way of seeing the equation of motion and as state in section 4.3.2 it will have a major impact on how we write our solver. When we call MBsysTran with the option Impdynared we will obtain the implicit form of the equation of motion.

$$\begin{aligned} \dot{x} &= v_x \\ \dot{y} &= v_y \\ \dot{\phi} &= \phi_p \end{aligned}$$

$$\begin{pmatrix} (M+m) & 0 & 0 \\ 0 & m+M & \frac{ML}{2}\cos\phi \\ 0 & \frac{ML}{2}\cos\phi & \frac{ML^2}{3} \end{pmatrix} \begin{pmatrix} \dot{v_x} \\ \dot{v_y} \\ \dot{\phi_p} \end{pmatrix} + \begin{pmatrix} 0 \\ -\frac{ML}{2}\dot{\phi}^2\sin\phi \\ -\frac{ML}{2}g\sin\phi \end{pmatrix} = \begin{pmatrix} F_x \\ F_y \\ 0 \end{pmatrix}$$

Table 4.2: equation of motion Inverse pendulum by Impdynared

Where x and y stand for the position in a 2D plane in [m]. v_x is the speed in [m/s] in the x direction. v_y is the speed in [m/s] in the y direction. ϕ is the angle in [degrees] of the pendulum and the value 0 mean being straight-up. ϕ_p is the angular speed in [degrees/s]. F_x and F_y are both forces implied on the base either in the x or y direction in [N]. L is a constant and is the length of the pendulum in [m]. And finally, M and m are respectively the mass of the base and the masses of the pendulum in [kg].

As you can see, we don't have access directly to the acceleration of our system, to obtain them we must first solve a system of differential algebraic equation(DAE). The quality of such approach is that, the symbolic expression of those DAE is smaller than the symbolic equation for the ODE of the explicit form. We could thus have smaller constrain to satisfy in Casadi.

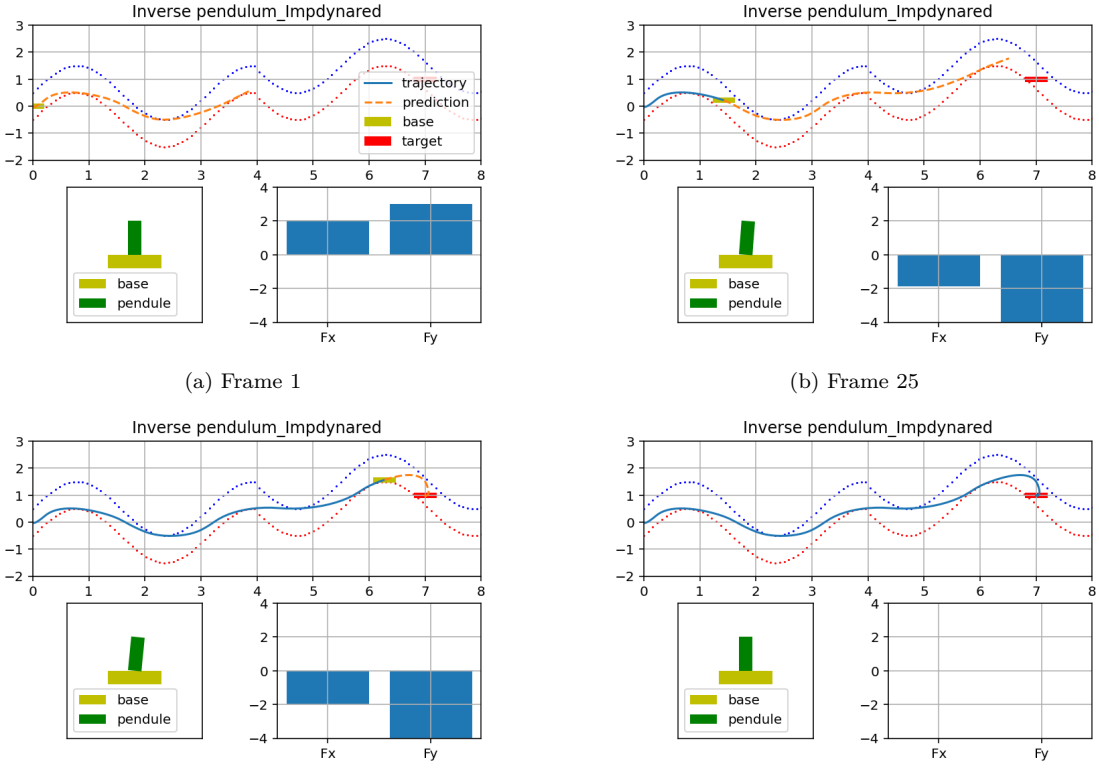
But in the meantime, this approach will ask Casadi to solve a lot DAE because by using RK4 as an integrator scheme we will need to solve 4 times these set of DAE for each prediction step. This will ask us to create a new type of constraint in the vector of constraint "g" 3.1.2, those constraints will be called algebraic constraint. Plus, in order to solve those DAE, we need to store the result somewhere for Casadi the only place is the vector "X" 3.1.2. If you wish to see what exactly that implied, you can find the explanation in the appendix A.4

Graphics and results

Now that we have explained every thing, lest see the results of our simulations.

For the following simulation, we use those parameters.

- N = 50
- T = δt = 0,1[s]
- The prediction scheme is RK4
- The boundaries of the trajectory follow the equation 3.2
- F_x range is [-2,2][N]
- F_y range is [-4,4][N]
- ϕ range is [-20,20][degrees]



(a) Frame 1

(b) Frame 25

(c) Frame 75

(d) Frame 125

Figure 4.8: Animation Inverse pendulum using Impdynared

4.4.3 Experiment and observation (Accelred and Impdynared comparison)

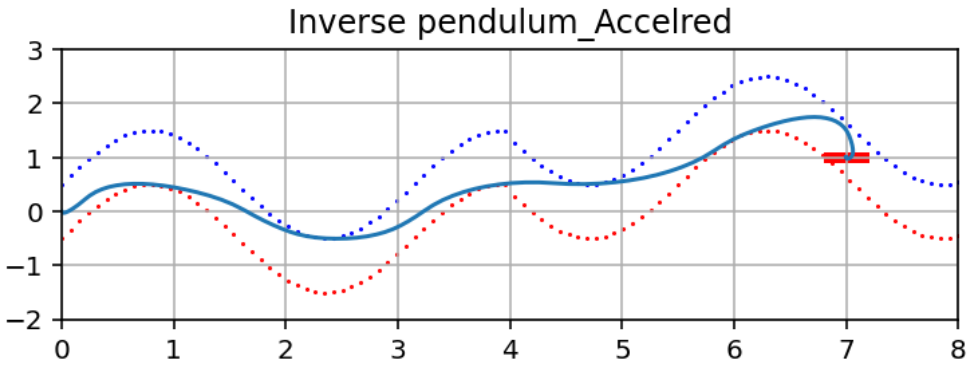
In this section, we will talk about the two kind of models and compare the two to each other and also to the analytical model.

Comparison of the trajectory between analytical, Accelred and Impdynared

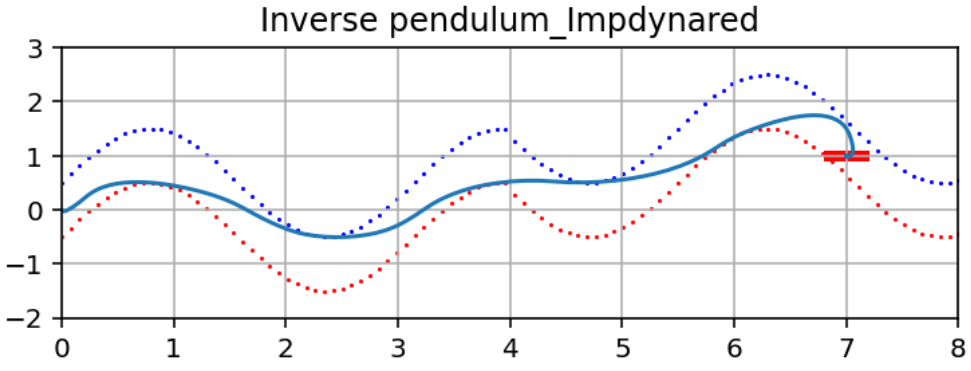
We will first need to compare are the trajectories. Because of course being the same problem with the same cost function and initial parameters, we want to obtain the exact same trajectories. We want to obtain the same result using different techniques to show that all those techniques have been correctly implemented in Robotran. For the following simulation, we use those parameters.

- N = 50
- T = δt = 0,1[s]
- The prediction scheme is RK4
- The boundaries of the trajectory follow the equation 3.2
- F_x range is [-2,2][N]

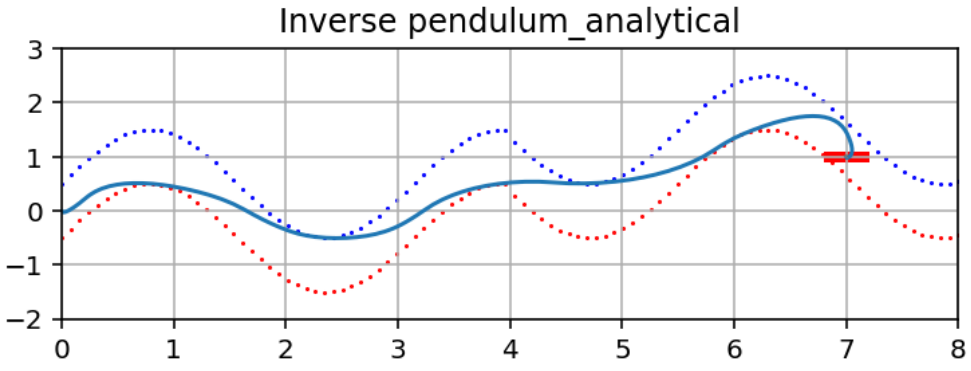
- F_y range is [-4,4][N]
- ϕ range is [-20,20][degrees]



(a) Accelred



(b) Impdynared



(c) Analytical

Figure 4.9: Trajectory comparison between analytical, Accelred and Impdynared

As we can see, all three methods have the exact same results, meaning that our implementation of Casadi inside the MBsysPy loop has been successful.

Comparison of the time between Accelred and Impdynared

Now that we showed that our models using Accelred and our model using Impdynared have the same result, it is this time to see which one work best with Casadi. And this is not a simple question to answer. Because Accelred give us a more direct way to obtain the acceleration we need to make our dynamics constraint in the vector "g" 3.1.2, and thus we have fewer constraint to satisfy. Meanwhile, Impdynared give us a set of DAE that is symbolically simpler and thus easier to use for the solver, but that solver will have to solve that system multiple times per prediction step.

To compare their computation time, I will use the same parameters as before

- The prediction scheme is RK4
- The boundaries of the trajectory follow the equation 3.2
- F_x range is [-2,2][N]
- F_y range is [-4,4][N]
- ϕ range is [-20,20][degrees]

But we will change the number of prediction step(N) the prediction time step(T) and the integration time step(δt). We will change them in a way to keep a constant prediction horizon(T*N) and so that T=δt. We will also keep a time of simulation constant, we are thus always simulating 10 second of real life time. we should thus have the same trajectory at the end, but we will refine it more each time. The time is in second[s].

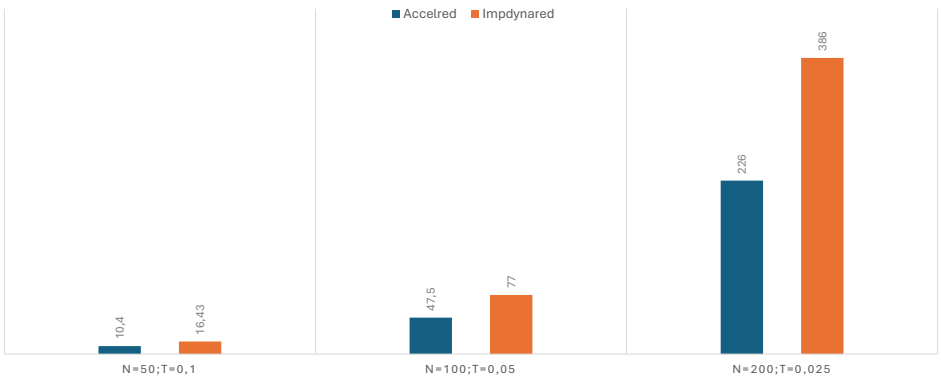


Figure 4.10: Comparison of the time (in second) between Accelred and Impdynared

As we can see, Accelred is faster everywhere. It is faster for all amount of prediction step.

Comparison of the convergence between Accelred and Impdynared

We will now talk about the convergence of the two models. When we are calling the predictive solver, it will try to converge to an optimal prediction for our cost function. And it is the time converging to a prediction that take the longest in our programme, so lest see how do they

compare to each other.

When we talk about convergence, there are two things to keep in mind, the amount of step it takes to converge(C_n) and the time taken by each step(C_t[ms]). We will thus examine both.

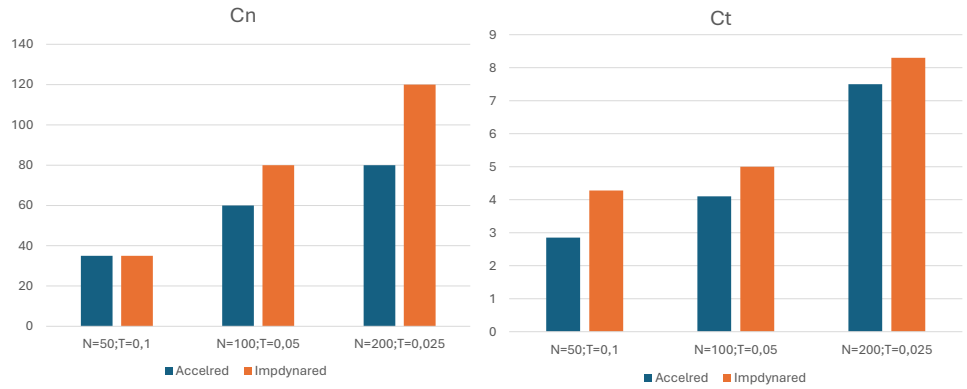


Figure 4.11: Comparison of the convergence between Accelred and Impdynared

We can observe two interesting tendencies. Accelred and Impdynared models take the same amount of step to converge at the start, but the more the model become refined, the more we see a difference. It seems that the smaller amount of constraint use by the Accelred models enable Casadi to converge using less step.

The other tendencies to notice is that the Accelred models compute each convergence step faster. But the more we refined the model, the less the difference is noticeable between the two models. It seems that because we have to solve the DAE with Impdynared there is an initial cost, but the more we refined the model, the less that initial cost impact the performances.

But still, even if the Impdynared model could compute each convergence step faster, when dealing with refined models, it seems that the simplicity of the Accelred models work best with Casadi to keep a low number of convergence steps. This why for the rest of the rapport we will use Accelred.

4.5 Roomba with Inverse pendulum model

This is the second model using both Robotran and Casadi in the same time. The goal of this model was to create a model without knowing about the analytical expression of the equation of motion. We could manually find them, but it would be tedious and not the goal here. The objective here is to fully trust that our implementation of Robotran and Casadi is robust and trust that MBsysTran will generate the correct symbolic expression of motion for Casadi.

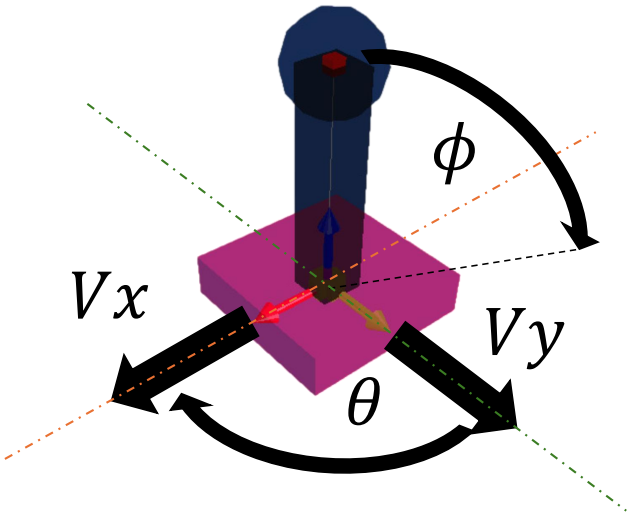


Figure 4.12: 3D representation of the Roomba pendulum

This model is the combination of the inverse pendulum and a Roomba. It can turn on himself and advance, while in the meantime it must keep its pendulum upright. This model is very close to the behaviour of a bike because it must keep its balance and also have a rolling without slipping constraint on its movement. The rolling without slipping constraint make sure that the Roomba can only go in the direction it is heading, this will thus link the speed in x and y with the angle θ.

4.5.1 The physical model

In this section, we will rapidly explain the physical model of the Roomba pendulum. And the special constraint it will follow.

We will first show our equation of motions.

$$
\begin{aligned}
\dot{x} &= v_x \\
\dot{y} &= v_y \\
\dot{\theta} &= \theta_p \\
\dot{\phi} &= \phi_p \\
\dot{v_x} &= \mathcal{F}_x(x, y, \theta, \phi, v_x, v_y, \theta_p, \phi_p, F_x, F_y, L_\theta) \\
\dot{v_y} &= \mathcal{F}_y(x, y, \theta, \phi, v_x, v_y, \theta_p, \phi_p, F_x, F_y, L_\theta) \\
\dot{\theta_p} &= \mathcal{F}_\theta(x, y, \theta, \phi, v_x, v_y, \theta_p, \phi_p, F_x, F_y, L_\theta) \\
\dot{\phi_p} &= \mathcal{F}_\phi(x, y, \theta, \phi, v_x, v_y, \theta_p, \phi_p, F_x, F_y, L_\theta)
\end{aligned}
$$

with $\mathcal{F}$ being an unknown symbolic function given by MBsysTran

Table 4.3: equation of motion Roomba pendulum by Accelred

Where x and y stand for the position in a 2D plane in [m]. v_x is the speed in [m/s] in the x direction. v_y is the speed in [m/s] in the y direction. θ is the angle [degrees] that permit the Roomba to turn on itself and thus change its heading, and the value 0 means that the heading is allong the axe x. θ_p is the angular speed in [degrees/s]. ϕ is the angle in [degrees] of the

pendulum and the value 0 means being straight-up. ϕ_p is the angular speed in [degrees/s]. F_x and F_y are both forces implied on the Roomba either in the x or y direction in [N]. And finally, L_θ is a torque [Nm] on the z axes of the Roomba.

As previously explained, for this model we don't have the analytical solution for the equation of motion. We get those equations from MBsysTran with the option Accelred to receive the explicit form of the equation of motion. We thus don't have any way to verified them. Because the symbolic generation use by MBsysTran is not very readable for a human. If you wish to see for yourself their formulation are in appendix A.5

From those equations, we can derive our states variables and controls variables.

State variables	**control variables**
x	F_x
y	F_y
θ	L_θ
ϕ	
v_x	
v_y	
θ_p	
ϕ_p	

Table 4.4: Type of variable Roomba pendulum

And the rolling without slipping constraint is simply this constraint. It was made that way to avoid division by zero.

$$v_y = v_x * tan(\theta) \tag{4.5}$$

4.5.2 Writing the solver

The writing of the solver is pretty straits forward, we present here when N=3

We start with our state variables.

$$X\_state = \begin{bmatrix} x_0 & x_1 & x_2 & x_3 \\ y_0 & y_1 & y_2 & y_3 \\ \theta_0 & \theta_1 & \theta_2 & \theta_3 \\ \phi_0 & \phi_1 & \phi_2 & \phi_3 \\ v_{x0} & v_{x1} & v_{x2} & v_{x3} \\ v_{y0} & v_{y1} & v_{y2} & v_{y3} \\ \theta_{p0} & \theta_{p1} & \theta_{p2} & \theta_{p3} \\ \phi_{p0} & \phi_{p1} & \phi_{p2} & \phi_{p3} \end{bmatrix} \tag{4.6}$$

Then the control variables.

$$U = \begin{bmatrix} F_{x0} & F_{x1} & F_{x2} \\ F_{y0} & F_{y1} & F_{y2} \\ L_{\theta 0} & L_{\theta 1} & L_{\theta 2} \end{bmatrix} \tag{4.7}$$

The constraints in the vector 'g' that are presenting here serve on linking step 1 to step 0 of the prediction.

$$g = \begin{bmatrix} ... \\ (x_1 - (x_0 + RK4(v_x([x, y, \theta, \phi, v_x, v_y, \theta_p, \phi_p, F_x, F_y, L_\theta])))) \\ (y_1 - (y_0 + RK4(v_y([x, y, \theta, \phi, v_x, v_y, \theta_p, \phi_p, F_x, F_y, L_\theta])))) \\ (\theta_1 - (\theta_0 + RK4(\theta p([x, y, \theta, \phi, v_x, v_y, \theta_p, \phi_p, F_x, F_y, L_\theta])))) \\ (\phi_1 - (\phi_0 + RK4(\phi_p([x, y, \theta, \phi, v_x, v_y, \theta_p, \phi_p, F_x, F_y, L_\theta])))) \\ (v_{x1} - (v_{x0} + RK4(\dot{v_x}([x, y, \theta, \phi, v_x, v_y, \theta_p, \phi_p, F_x, F_y, L_\theta])))) \\ (v_{y1} - (v_{y0} + RK4(\dot{v_y}([x, y, \theta, \phi, v_x, v_y, \theta_p, \phi_p, F_x, F_y, L_\theta])))) \\ (\theta_{p1} - (\theta_{p0} + RK4(\dot{\theta p}([x, y, \theta, \phi, v_x, v_y, \theta_p, \phi_p, F_x, F_y, L_\theta])))) \\ (\phi_{p1} - (\phi_{p0} + RK4(\dot{\phi_p}([x, y, \theta, \phi, v_x, v_y, \theta_p, \phi_p, F_x, F_y, L_\theta])))) \\ ... \end{bmatrix} \tag{4.8}$$

Personalised constraint of the rolling without slipping.

$$g = \begin{bmatrix} ... \\ v_{y1} = v_{x1} * tan(\theta_1) \\ v_{y2} = v_{x2} * tan(\theta_2) \\ v_{y3} = v_{x3} * tan(\theta_3) \\ ... \end{bmatrix} \tag{4.9}$$

The following constraints are the one responsible to ensure that the pendulum stay upright. This is typically the type of constraints store in lbx and ubx.

$$-20^\circ <= \phi_1 <= 20^\circ \tag{4.10}$$
$$-20^\circ <= \phi_2 <= 20^\circ \tag{4.11}$$
$$-20^\circ <= \phi_3 <= 20^\circ \tag{4.12}$$

And finally, the cost function of that model is the following (the same as in the Roomba models 3.35)

$$obj = \sum_{i=1}^{N} \frac{i}{N} \sqrt{W_0(x_i - x_f)^2 + W_1(y_i - y_f)^2 + W_2(\theta_i - \theta_f)^2} \tag{4.13}$$

Where W_0, W_1 and W_2 are constant serving as weight parameters.

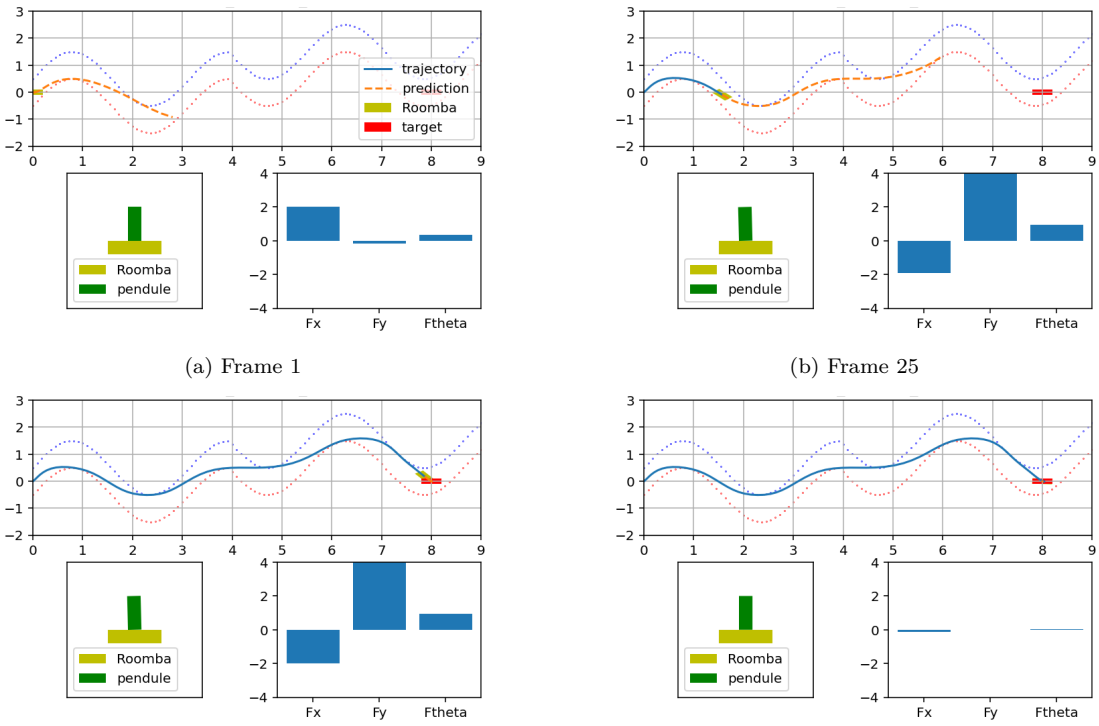
4.5.3 Graphics and results

We will now have a look at the result of this model//

We are using the following parameters

- N = 30
- T = δt = 0,1[s]
- The prediction scheme is RK4
- The boundaries of the trajectory follow the equation 3.2
- F_x range is [-2,2][N]
- F_y range is [-4,4][N]

- L_θ range is [-1,1][Nm]
- ϕ range is [-20,20][degrees]



(a) Frame 1

(b) Frame 25

(c) Frame 75

(d) Frame 125

Figure 4.13: Animation Roomba pendulum using Accelred

As we can see, even without knowing the analytical equation of motion, we manage to a working predictive solver.

4.6 Experiment and observation (making the predictive solver faster)

In this section, we will talk about the experiment that have been done with this model. All the experiment done with this model had all the same objective, making the predictive solver faster.

This is because the next and final model, the ski-bike, was extremely slow, taking one hour for only 0.25 seconds of simulation. This why this model has served as a test bench to see what could be done to improve the predictive solver speed of computation.

Improving the symbolic of the constraint vector "g"

The first thing we have done, was improving the symbolic of the constraint vector g. Casadi works using a recursive approach of the symbolic, see the appendix A.5 for a visual representation.

But this mean that if multiple equation have the same symbolic expression, we will only calculate one time that expression and use the result on those equations.

In earlier model, we were letting Casadi do its thing without supervision. But now we add the constraint in the vector 'g' in such a way that it maximise the recursivity of 'g'. Sadly, there is a tradeoff, it takes more times to write the solver that way. But this is a price that we pay one time at the writing of the solver. Meanwhile, we call the predictive solver multiple time during the simulation, and thus making the solver faster at the expense of its writing will save us time.

Tolerance over the cost function(new parameter : Tol)

Another way to make the predictive solver faster, is simply to converge earlier with a less optimal solution, by putting looser tolerances on what we defined as an optimal solution for Casadi (by default, Casadi is using a tolerance of $Tol = 10^-6$). Of course this mean that we will not a have the best theoretical solution possible, but a good solution in a few second is preferable that a perfect solution in an hour, in our case.

Using more than one prediction step at a time(new parrameter : N_skip)

For now, we have been calling the solver for each time step of the simulation. But our prediction is valid for multiple time step of the simulation, we could thus use multiple prediction time step before recalculating the prediction. This mean that by only using 2 prediction times step each time ($N\_skip = 2$), we will call the predictive solver 2 times less. This can be explained in a more graphic way:

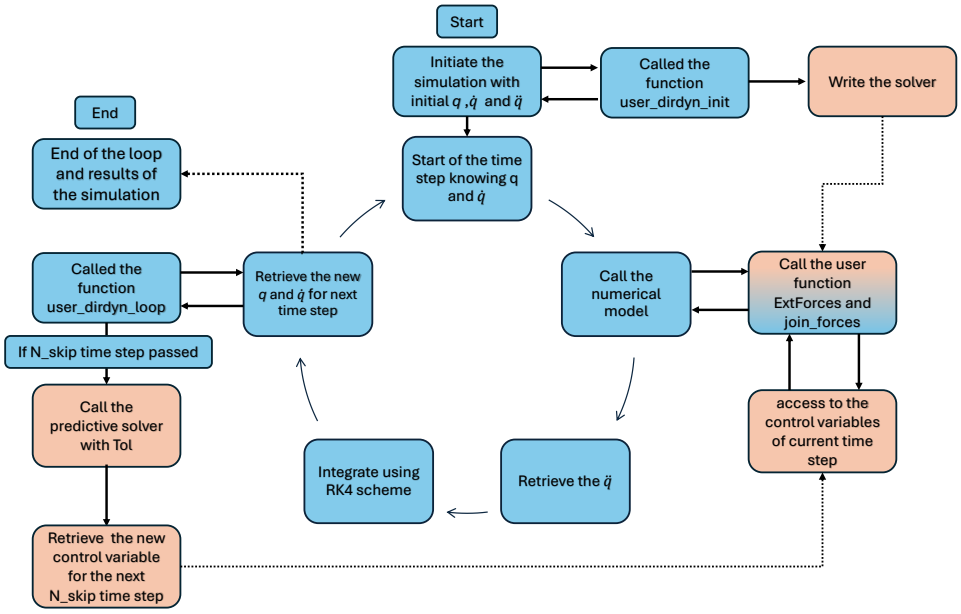


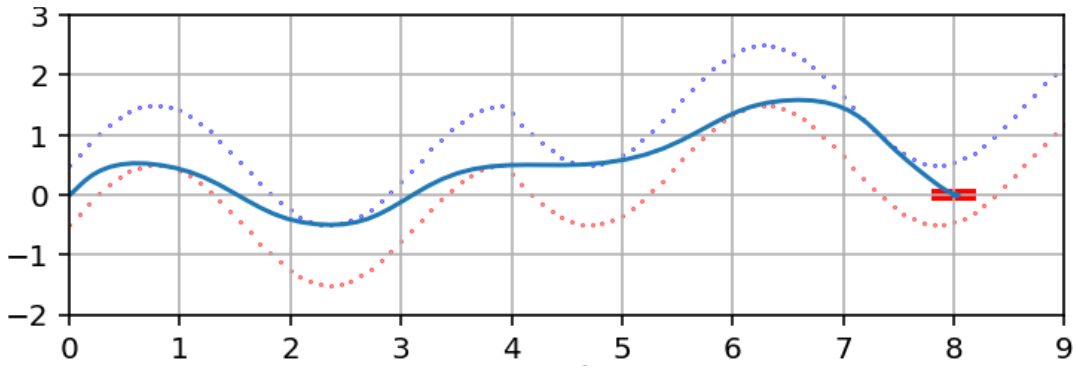
Figure 4.14: Inner working of MBsysPy with Casadi fast solver

Of course, doing that also comes with drawbacks. Our predictive solver, being called less often, mean that it won't be as reactive to abrupt change in the boundaries. And that also mean

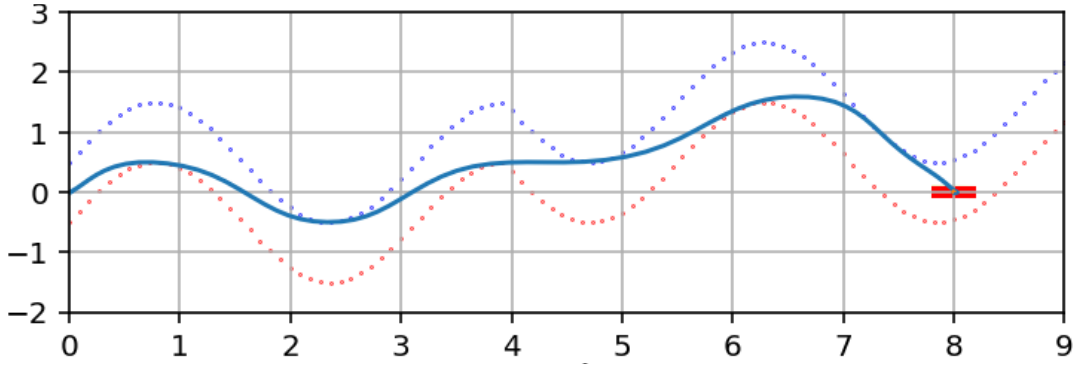
that the prediction and the integration need to agree on multiple time step at a time, and thus using a prediction scheme less precise than Rk4 won't work.

As a comparison between our old slow solver and the new fast solver, we will do a simulation using the following parameters. Of course, the slow solver doesn't use the parameters *Tol* an *N_skip*.

- N = 50
- T = δt = 0,1[s]
- Tol = 0,1*N
- N_skip = 0,1*N
- The prediction scheme is RK4
- The boundaries of the trajectory follow the equation 3.2
- F_x range is [-2,2][N]
- F_y range is [-4,4][N]
- L_θ range is [-1,1][Nm]
- ϕ range is [-20,20][degrees]



(a) slow solver taking 63 seconds



(b) fast solver taking 9 seconds

Figure 4.15: Comparison animation Roomba pendulum between the fast and slow solver

The difference of the trajectory between the two solvers is barely noticeable. But we compute the solution a lot quicker(9[s] instead of 63[s]) and for our final model the Ski-Bike we will be force to use the fast predictive solver because the model would take hours to compute otherwise.

Plus, we could also prove here that the more we refined the model, the more time we save. But this would be obvious, we are saving time each time we don't call the solver because of N_skip and we are saving time each time we call the solver because of Tol and the improvement we made in the recursivity of the vector constraint 'g' making the solver faster.

Chapter 5

Ski Bike models

In this chapter, we will present our final model, the Ski-Bike.

(a) Real Ski-Bike example

(b) Robotran equivalent

Figure 5.1: Picture showing a real Ski-Bike and its Robotran equivalent

We decided to create the Ski-Bike model for multiple reason. First, it is a very good approximation of a real bike while still being a simple model. This will enable us to keep a computation time acceptable while giving us result close to directly controlling a bike.

Second, this will simplify our ground contact model. Because if we were dealing with a wheel, we would have to create a model capable of working in a rotating assembly. Meaning we would have to recalculate for every time step, the contact point. But in our case with a ski, the contact point is fixed, making it easier for the symbolic generation of the equations used by Casadi.

Lastly, there are no suspension, for once again simplifying the model.

5.1 Robotran implementation

In this section, we will look at the Robotran model we will control. We will thus go over its geometry and the different links composing that geometry. We will also talk about the ground contact model and final show the behaviour of the untrolled Ski-Bike.

5.1.1 The geometry

The geometry of the model is fairly simple, being composed of only three bodies. The base being the ground, the frame of the Ski-Bike and the ski of the Ski-Bike. Of course the frame and the ski have their one mass, centre of mass and inertia.

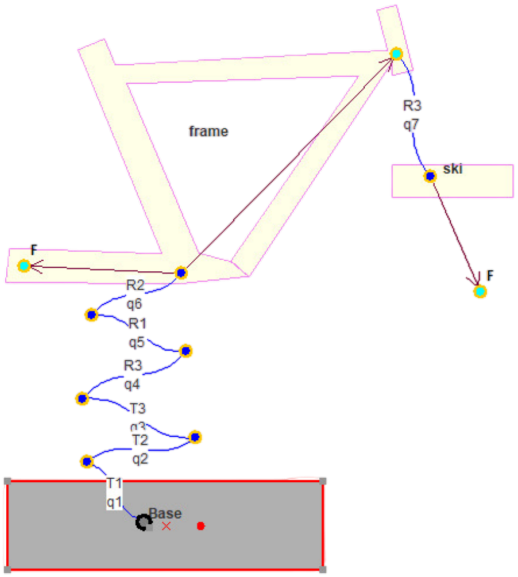


Figure 5.2: Geometry in Robotran of the Ski-Bike

We can also observe that the geometry is composed of 7 joints, 3 prismatic and 4 rotoide. Something noteworthy here is that the joint q6 doesn't start at 0 degrees, but instead start at the value of the rake angle. Because otherwise we should have to put a driven-joint before q7 to create that rake angle. This thus save us one joint and thus make the computation faster. But of course then every anchor point of the frame will be express in that rotated base. In insight this was not necessary, it is possible for Casadi to take account of the driven joint, and it wouldn't have change mush in computation time.

Lastly, we can see that there are two external-forces. Those forces will be the forces acting on the Ski-Bike by the ground, and will thus come from our ground contact model.

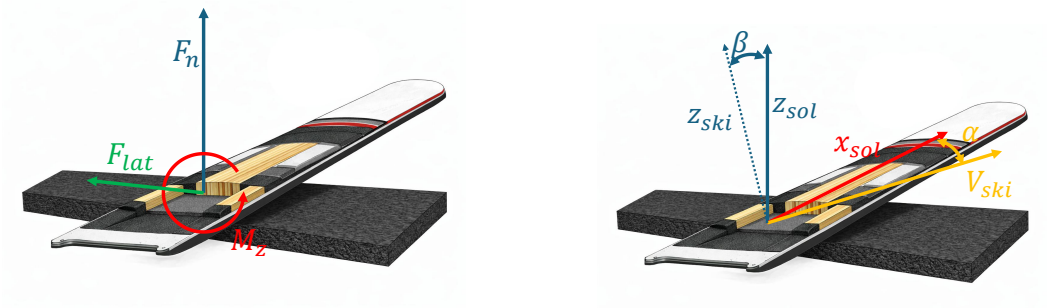
With all of this explained, we will now make a list of every joint name and role.

- q1 or x ,v_x,$\dot{v_x}$ is the position, the speed and the acceleration of the Ski-Bike in the generalised x direction in [m], [m/s], [m/s^2]
- q2 or y ,v_y,$\dot{v_y}$ is the position, the speed and the acceleration of the Ski-Bike in the generalised y direction in [m], [m/s], [m/s^2]
- q3 or z ,v_z,$\dot{v_z}$ is the position, the speed and the acceleration of the Ski-Bike in the generalised z direction in [m], [m/s], [m/s^2]

- q4 or θ, θ_p ,$\dot{\theta_p}$ is the angle, angular speed and angular acceleration of the yaw angle of the Ski-Bike in [degrees], [degrees/s], [degrees/s^2]
- q5 or ϕ, ϕ_p ,$\dot{\phi_p}$ is the angle, angular speed and angular acceleration of the roll angle of the Ski-Bike in [degrees], [degrees/s], [degrees/s^2]
- q6 or γ, γ_p ,$\dot{\gamma_p}$ is the angle, angular speed and angular acceleration of the pitch angle of the Ski-Bike in [degrees], [degrees/s], [degrees/s^2]
- q7 or δ, δ_p ,$\dot{\delta_p}$ is the angle, angular speed and angular acceleration of the handlebars of the Ski-Bike in [degrees], [degrees/s], [degrees/s^2]

5.1.2 The ground contact model (model of Sharp)

We will now explain briefly the ground contact model. This model is the model of Schwab [1] but applied to a ski.



(a) Forces and torque applied on the ski

(b) Vector of the ground contact model

Figure 5.3: Visualisation of the ground model

We will thus introduce two new angles, the slip angle α and the inclination angle β. The value of those angle can be determined by using the value of our joints. And from those angle we can compute the forces acting on the skis.

$$F_n = C_n * (z_{ski} - z_{sol}) + C_{\dot{n}} * (\dot{z_{ski}} - \dot{z_{sol}})$$
$$F_{lat} = C_\alpha * \alpha + C_\beta * \beta$$
$$M_z = C_{M\alpha} * \alpha$$

Table 5.1: Equation of the ground forces

In the equation of the ground model 5.1, we can see that F_n is a spring with damping. While F_{lat} and M_z are functions of the two angles, we discuss earlier.

This set of equation is far from being perfect. Firstly, we don't take into account that the ski could not touch the ground all the forces are always applied. Plus, F_n can even be negative if

[1] more information on `https://www.researchgate.net/profile/Arend-Schwab/publication/239293889_Linearized_dynamics_equations_for_the_balance_and_steer_of_a_bicycle_A_benchmark_and_review/links/0c960522a652fa18b8000000/Linearized-dynamics-equations-for-the-balance-and-steer-of-a-bicycle-A-benchmark-and-review.pdf`

the ski is above ground, which is physically incorrect. This is why we only interest ourselves on trajectories happening on completely flat ground with no holes or bump.

The second issue is that our F_{lat} is unbounded, meaning that we can never lose grip. It is thus impossible for this model to slide or to study that aspects.

The last thing is that in order to work, the contact model need a v_x not equal to 0. Meaning, we will no longer be ables to stop at the target.

5.1.3 The behaviour without controls

The geometry of a bike is not randomly made. This geometry has a range of speed at which the bike is capable of self-stabilisation. Sadly, our Ski-bike is not capable of such feats. Our model is natively unstable at all speeds.

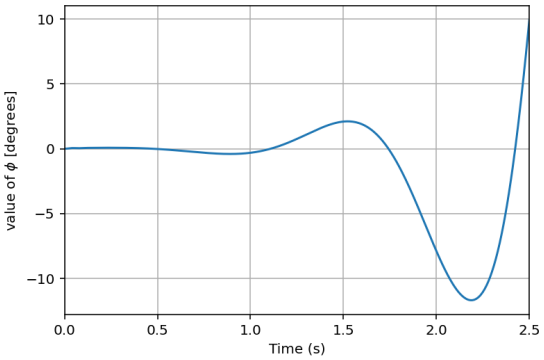


Figure 5.4: Uncontrolled behaviour of the Ski-Bike, showing the roll angle ϕ as function of time

Here is the evolution of the roll angle as a function of time of our uncontrolled Ski-Bike. And as we can see, the slitless perturbation will cause the Ski-Bike to fall on is side.

5.2 Casadi implementation

In this section, we will present our Casadi implementation. We will talk about the two implementation and why we chose the Accelred implementation. We will explain our two control variables and show the result of the simulation.

5.2.1 Accelred or Impdynared

For this model we have done both the Accelred an Impdynared implementation. To verify if our observation in the inverse pendulum model using Robotran section 4.4.3 still hold for a more complex model.

The first model that was completed was the Impdynared model. And this model had some big issue converging to a prediction, taking up to 3000 convergence steps for each prediction. Because as we saw in the graph 4.11, the more a model is refined the more convergence step is

needed for the Impdynared model to converge and our Ski-Bike model need to be very refined.

Our previous models had an integration time step of 0.1[s], but Ski-bike must have an integration time(δt) step of 5[ms] due to the numerical stability of the contact model. And because our prediction must also comply to the rule of numerical stability, the maximum prediction time step(T) is also 5[ms]. Meaning that if we want to keep a good horizon for our prediction, we must augment the number of prediction step (N). We can thus comprehend that Ski-Bike will be a very refined model compare to the previous ones. So in the end, this model using the Impdynared implementation was taking 1h for each 0.25 seconds of real time simulation. This is the reason why in section 4.6, we presented a way to make the solver faster, it was a necessity.

With all of this said, this why we use the Accelred implementation with the fast solver presented in 4.6. And with that we manage to cut down the time to 5 min of computation for 10 seconds of real time simulation. That is still far from perfect, but at least it is a time of computation we can work with.

It would be hard to make it faster without sacrificing the quality of the prediction. Because with this model, we are already converging with only 10 to 15 convergence steps. The problem is that each convergence step take a lot of time to compute on their on due to the high value of N.

5.2.2 Control variables and overview

Control variables

In this model, we will have to control variables. The first one is an external force called F_{long}

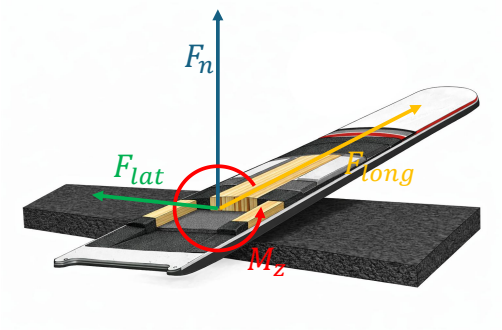


Figure 5.5: Representation of the ski with all the external forces

This force will only be applied in the longitudinal direction of the rear ski. This is the first time we are controlling an external force, and it has demanded some adjustment to make it work both in Casadi and Robotran, as we explained in section 4.2.3. That force enable the solver to accelerate or decelerate the Ski-Bike.

The second control variable is a joint force L_δ which is a torque directly applied on the handelbars and thus the joint q7. That torque is obviously responsible to turne the handelbards in the requiere position for the trajectories.

Overview

We can thus now make a quick summary of the model.

As we are using the option Accelred in MBsysTran we obtain the following equations of motions.

$$\begin{gathered}
\dot{x} = v_x \ ; \ \dot{\theta} = \theta_p \\
\dot{y} = v_y \ ; \ \dot{\phi} = \phi_p \\
\dot{z} = v_z \ ; \ \dot{\gamma} = \gamma_p \\
\dot{\delta} = \delta_p \\
\dot{v_x} = \mathcal{F}_x(x, y, z, \theta, \phi, \gamma, \delta, v_x, v_y, v_z, \theta_p, \phi_p, \gamma_p, \delta_p, F_{long}, L_\delta) \\
\dot{v_y} = \mathcal{F}_y(x, y, z, \theta, \phi, \gamma, \delta, v_x, v_y, v_z, \theta_p, \phi_p, \gamma_p, \delta_p, F_{long}, L_\delta) \\
\dot{v_z} = \mathcal{F}_z(x, y, z, \theta, \phi, \gamma, \delta, v_x, v_y, v_z, \theta_p, \phi_p, \gamma_p, \delta_p, F_{long}, L_\delta) \\
\dot{\theta_p} = \mathcal{F}_\theta(x, y, z, \theta, \phi, \gamma, \delta, v_x, v_y, v_z, \theta_p, \phi_p, \gamma_p, \delta_p, F_{long}, L_\delta) \\
\dot{\phi_p} = \mathcal{F}_\phi(x, y, z, \theta, \phi, \gamma, \delta, v_x, v_y, v_z, \theta_p, \phi_p, \gamma_p, \delta_p, F_{long}, L_\delta) \\
\dot{\gamma_p} = \mathcal{F}_\gamma(x, y, z, \theta, \phi, \gamma, \delta, v_x, v_y, v_z, \theta_p, \phi_p, \gamma_p, \delta_p, F_{long}, L_\delta) \\
\dot{\delta_p} = \mathcal{F}_\delta(x, y, z, \theta, \phi, \gamma, \delta, v_x, v_y, v_z, \theta_p, \phi_p, \gamma_p, \delta_p, F_{long}, L_\delta)
\end{gathered}$$

with $\mathcal{F}$ being an unknown symbolic function given by MBsysTran

Table 5.2: equation of motion Ski-Bike pendulum by Accelred

From those equations, we can derive our states variables and controls variables.

State variables	**control variables**
x ; y	F_{long}
z ; θ	L_δ
ϕ ; γ	
δ ; v_x	
v_y ; v_z	
θ_p ; ϕ_p	
γ_p ; δ_p	

Table 5.3: Type of variable Roomba pendulum

And finally, our cost function is the following.

$$obj = \sum_{i=1}^{N} \frac{i}{N} \sqrt{W_0(x_i - x_f)^2 + W_1(y_i - y_f)^2} \tag{5.1}$$

Where W_0, W_1 and are constant serving as weight parameters.

The rest of the parameters $(X\_state, U, g, lbg, ubg, lbx, ubx)$ have adapted as normal.

5.2.3 Graphic and results

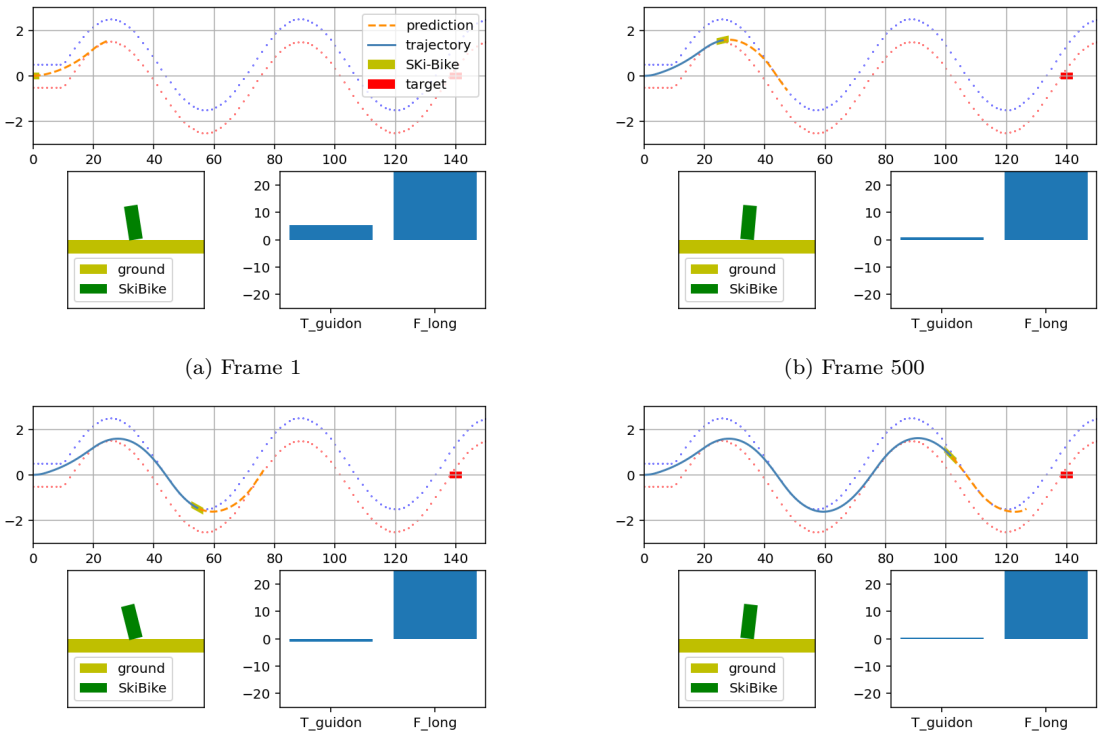
We will use the following boundary function

$$f\_cont = \mathcal{F}(c, k) = \begin{cases} 1 & \text{if } c < 10 \\ 2 * np.sin(0.1 * (k - 10)) & \text{else} \end{cases} \tag{5.2}$$

The idea being this boundary function is to give 10[m] to the model to stabilise itself before entering a sinus. Because due to the contact model, the Ski-bike can bounce on the z axis and need some time to stabilise.

For the following simulation, we are using these parameters.

- N = 400
- T = δt = 0,005[s]
- Tol = 1*N
- N_skip = 0,1*N
- The prediction scheme is RK4
- The boundaries of the trajectory follow the equation 5.2
- F_{long} range is [-25,25][N]
- L_{δ} range is [-10,10][Nm]
- ϕ range is [-25,25][degrees]
- v_x start at 10[m/s]



(a) Frame 1

(b) Frame 500

(c) Frame 1000

(d) Frame 1750

Figure 5.6: Animation Ski-Bike using Accelred

There is multiple thing we can note here. As stayed earlier, the integration and prediction time step (δ and T) is only 5[ms] meaning that to have a correct horizon the number of prediction step (N) has been augmented to 400. This enviably lead to longer computation time, hence why we are using the fast predictive solver.

Plus we can also observe that despise the fact that our Ski-Bike is dynamically unstable, as seen in section5.1.3, the predictive solver as no issue stabilising it. Meaning that during the parametric studies, we can study configuration of the Ski-Bike model that are dynamically unstable.

And lastly is that our control variable F_{long} seems to always be at its maximum. Meaning, we can take that exact same boundaries faster than our starting speed of 10[m/s]. Because the solver is trying to make the Ski-Bike move faster to reach its target earlier.

Chapter 6

Parametrical studies, observation, reflection and perpective

In this chapter, we will talk about all the conclusion we can make from the Ski-Bike model. Of course all the conclusion done in previous models still hold here so we won't do them again. We will also present a rather funny behaviour that can happen when we don't put certain constraint. And finally, we will talk about what could be done in the future if we wanted to continue this implementation of a predictive solver using Casadi in Robotran.

6.1 Parametrical studies and observation

Two of the great quality of using a predictive controller has a mean to control our multibody system is its versatility and robustness. Meaning, we can change certain aspect of our multibody system without having to change anything in the code of the predictive solver. We can change the mass, the inertias, the rake angle, the centre of mass, ... Mainly, as long as we don't alter the structure of the multibody system, the predictive solver will adjust itself to the change we have made. That enable us to test different submodels from the Ski-Bike and compare their behaviour using the exact same controller, and thus we have a fair way to potently determine which submodel is best for which purpose.

If we were using a most simple controller like a pure pursuit with a PID as presented in section 2.3 we would have to change the weigh parameters of the PID every time and thus this would not be fair to compare submodels. Because a model could be better only due to the weight parameters being more adapted for this submodel.

We will thus have a look at some submodel from the Ski-Bike.

6.1.1 Changing the mass of the frame

This is a rather simple first test, but what happen when the frame gets heavier or lighter. Our original Ski-Bike frame had a mass of 65 [kg] in other to take into account the mass of the frame and the cyclist. We will thus try to have the mass equal to 10 [kg] and 100 [kg] to see what happen when the Ski-Bike doesn't have a cyclist or a heavy one.

In other, to push the model, we will use a more aggressive boundary to follow, and we will block the force F_{long} to the value 0 to only compare the trajectories having the exact same speeds.

$$f\_cont = \mathcal{F}(c,k) = \begin{cases} 1 & \text{if } c < 20 \\ 2 * np.sin(0.2 * (k-20)) & \text{else} \end{cases} \tag{6.1}$$

We will use the following parameters for the simulations.

- N = 400
- T = δt = 0,005[s]
- Tol = 1*N
- N_skip = 0,1*N
- The prediction scheme is RK4
- The boundaries of the trajectory follow the equation 6.1
- F_{long} range is [-25,0][N]
- L_δ range is [-25,25][Nm]
- ϕ range is [-25,25][degrees]
- v_x start at 10[m/s]
- m_{frame} = 10 or 65 or 100 [kg]

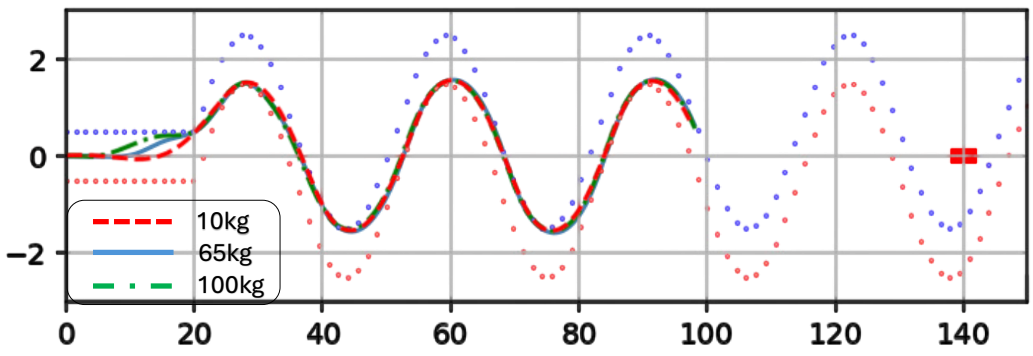


Figure 6.1: Changing the mass of the frame

As we can see, the only difference seems to be at the beginning of the trajectory. But the rest of the trajectory is extremely similar. The difference between each simulation is how smooth the solution find by the solver is.

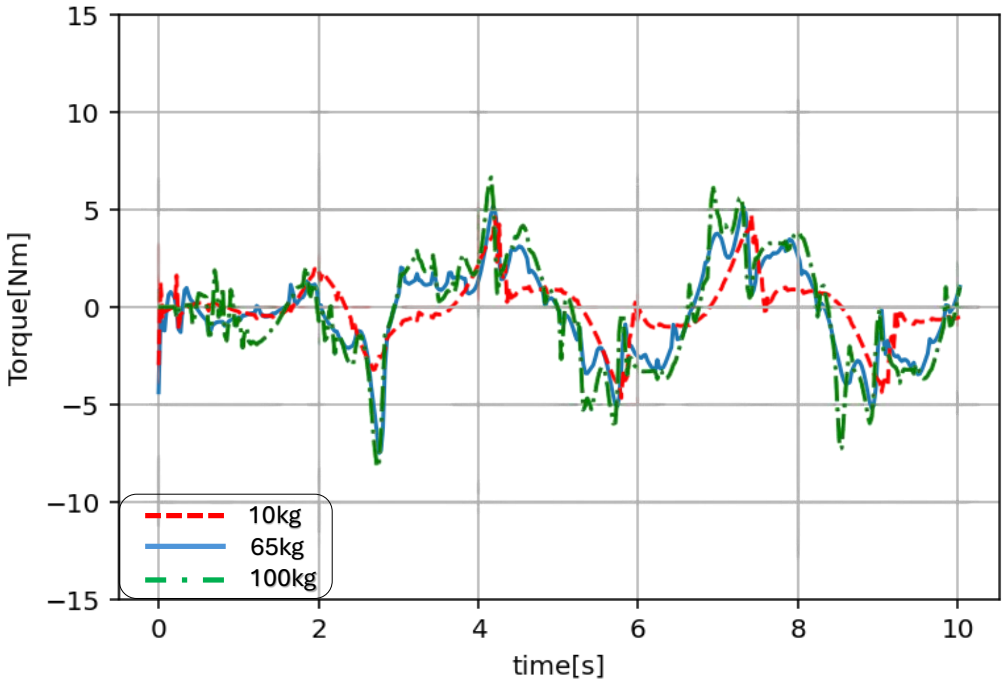


Figure 6.2: Torque on the handlebars while Changing the mass of the frame

And has we can see the lighter the frame is, the smother the torque is. And that result is easy to understand, it is easier to control a light bike than a heavy one. But we can also see that even if the value of the torque is different between each submodel, the trajectory they all follow in 6.1 was the same and thus the controller adapted by itself to the changes, as we wanted to prove.

6.1.2 Changing the rake angle

We will now make a more inserting parametric study. The rake angle of a bike is an important factor when discussing bike stability, thus being able to test different rake angle is also important. For the following simulations, we will use those parameters.

- N = 400
- T = δt = 0,005[s]
- Tol = 1*N
- N_skip = 0,1*N
- The prediction scheme is RK4
- The boundaries of the trajectory follow the equation 6.1
- F_{long} range is [-25,0][N]
- L_δ range is [-25,25][Nm]

- ϕ range is [-25,25][degrees]
- v_x start at 10[m/s]
- m_{frame} = 10 [kg]
- $rakeangle$ = 15 or 20 or 30 [degrees]

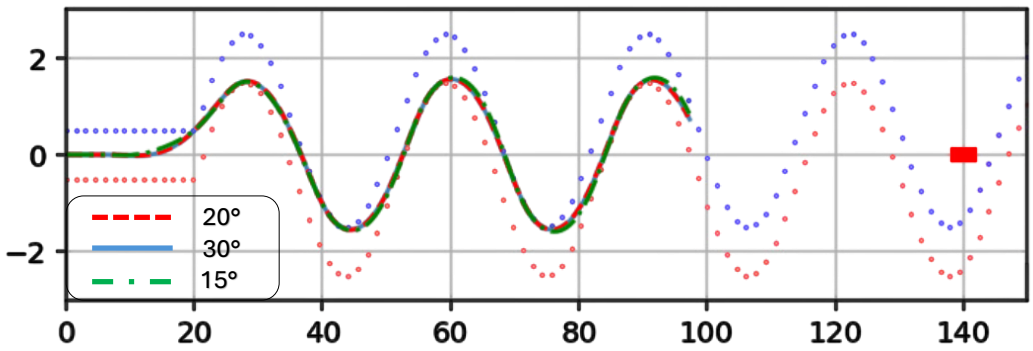


Figure 6.3: Trajectory of the Ski-Bike while Changing the rake angle

As we can see, whatever the rake angle is, the solver manages to adapt and deliver the exact same trajectory. However, there is something important to know here. When we were doing that parametric study, we wanted to test for 10 degrees and not 15 degrees. But the solver did not manage to find any solution for the 10 degrees of rake angle. And the following graphic help us understand why.

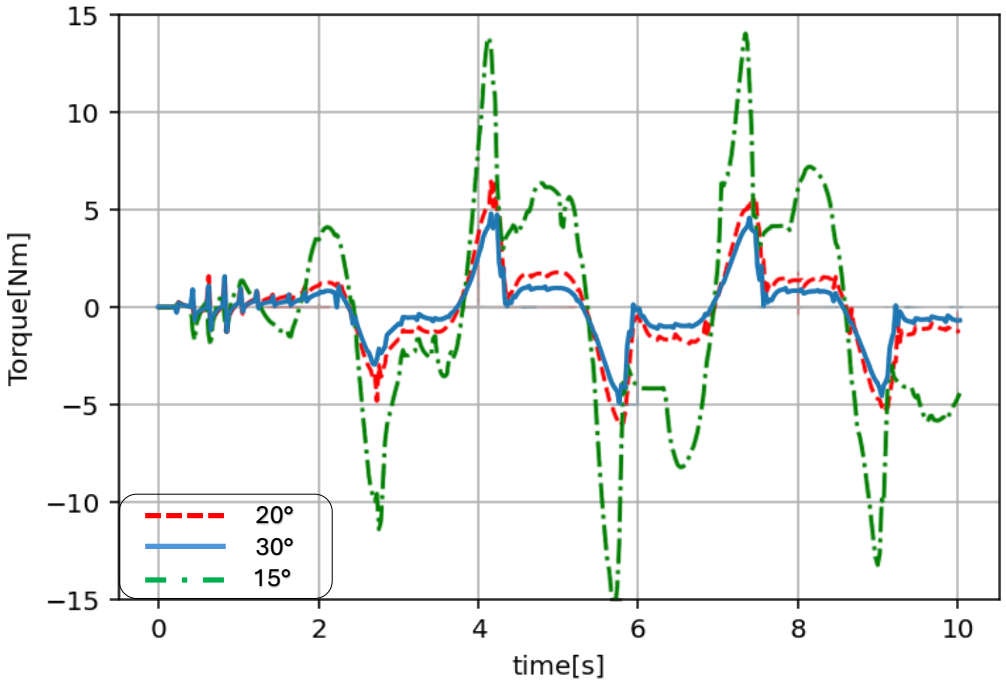


Figure 6.4: Torque on the handlebars while Changing the rake angle

This graphic show the torque applied on the handlebars over time for each rake angles. As we can see, the difference between 20 and 30 is minimal compare to 20 and 15. We can thus understand that the rake angle of 15 degrees is already quite unstable, and thus the rake angle of 10 degrees is so unstable that the predictive solver couldn't stabilise it. That observation is reassuring because it shows that bad geometries are identifiable by the result or the incapacity of the solver to control them. We can finally understand, that for this type of bike a rake angle between 20 and 30 is a good choice.

6.2 Awkward behaviour and shortcomings of the model

When we deal with controllers, sometimes the controller find ways to control our model we did not anticipate, and this controller is not different. One of them we already talk about was the teleportation in section 3.4.3. But this is a behaviour that all model can have if the control variables are not bounded. We will see now a behaviour that the Ski-Bike can have and how to avoid it.

6.2.1 Handlebar Rowing

This behaviour, like its name would suggest, is the solver deciding to turn the handlebars left for one time step and then turn the handlebar right for one other time step and then left again in some kind of rowing motion that keep the bike going forward. That behaviour happen in a very early stage of the development of the Ski-Bike, where there wasn't the control variable F_{long} yet.

The only control variable was L_δ and we were trying to make the bike go in a straight line. But what ever we tried, the handlebars kept rowing until we find this.

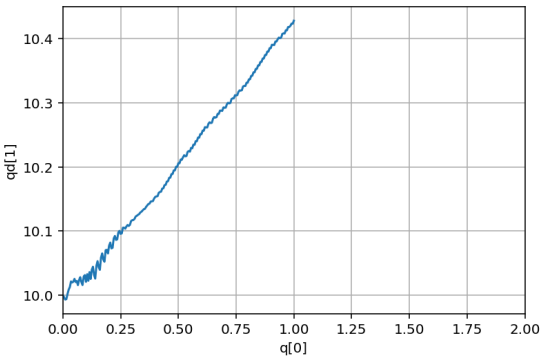


Figure 6.5: The issue

As we see in the picture, the velocity in x is increasing over time. It seems that by rowing the handlebars, the predictive solver founded a way to accelerate without using F_{long}. This wasn't expected nor wanted, but in the solver eyes it was the quickest way to reach its target by rowing to accelerate. The solution to this was simple, constraining the amount the control variable can change per prediction step. Because this behaviour was taking advantage of the fact that the solver could inverse the torque L_δ each 5[ms] and thus it was able to row. So by limiting the change of the control variables to 5% of their maximum value per prediction step, that behaviour was eliminated.

6.2.2 Speed limitation

The solver is not ables to work for absolute speed under 10[m/s] this could be the dynamically unstable nature of the Ski-Bike, like explained in section 5.1.3, but this could also be due to the horizon of the prediction. Because when we have small speed we travel less distance in the same amount of time and thus the horizon is smaller. Leading to the solver being 'surprise' that a boundary was in front of the Ski-Bike and it maybe too late to react.

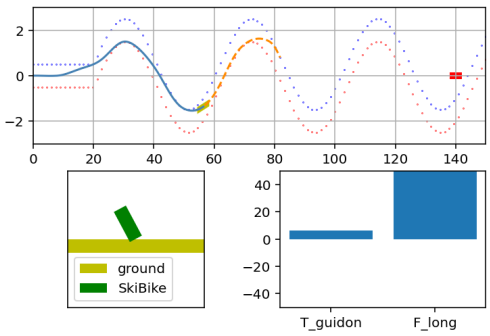
6.2.3 F_{long} limitation

The control variable F_{long} is working as intended, putting an external force in the longitudinal direction of the rear ski. Sadly, this external force will also create a torque that will impact the pitch of the Ski-Bike, like in real life. But this creates a problem, when F_{long} change is value to negative to decelerate there is a torque that will make the front of the Ski-Bike dive. And if that effect is too great, the solver doesn't seem to be able to recuperate.

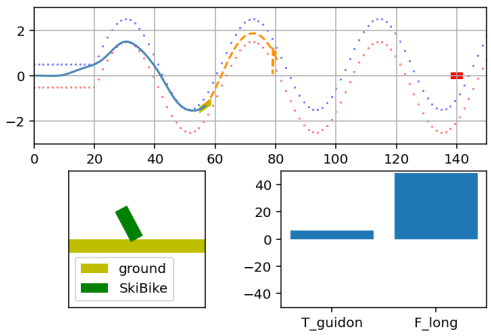
This is because our model doesn't have any suspension to account for that. The only thing that will account for that is the ground model with the force normal F_n, which is basically a damped spring. But that doesn't seem to be sufficient for big decelerations leading to the controller to failed staying in the boundaries.

A good example of this is the following simulation using those parameters.

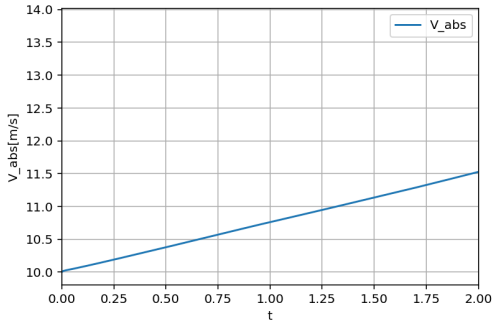
- N = 400
- T = δt = 0,005[s]
- Tol = 1*N
- N_skip = 0,1*N
- The prediction scheme is RK4
- The boundaries of the trajectory follow the equation 6.1
- F_{long} range is [-50,50][N]
- L_{δ} range is [-25,25][Nm]
- ϕ range is [-25,25][degrees]
- v_x start at 10[m/s]
- m_{frame} = 65[kg]



(a) Frame 960



(b) Frame 965



(c) Evolution of the absolute speed overtime

Figure 6.6: Animation Ski-Bike short komings F_{long}

As you can see, the start of the trajectory look normal but when the speed start to increase suddenly the prediction of the solver get broken, this is because we had to decelerate to pass the turn and the weight shift was too much to handle for the controller.

6.2.4 Convergence issue with certain cost function

This problem is not really one, but we noticed that Casadi has an easier time to converge when all prediction step are in the cost function of the predictive solver. This is why for the most part we have been using a cost function like this one.

$$obj = \sum_{i=1}^{N} \frac{i}{N} \sqrt{W_0(x_i - x_f)^2 + W_1(y_i - y_f)^2} \tag{6.2}$$

This cost function takes into account every prediction step, but it gives them an importance factor ($\frac{i}{N}$) because we are not interested that the start of the prediction is close to the target or not. What we want is to optimise the last position, like so.

$$obj = \sqrt{W_0(x_{i=N} - x_f)^2 + W_1(y_{i=N} - y_f)^2} \tag{6.3}$$

But without any indication on what is a good direction for every other prediction step, Casadi doesn't converge fast. It converges, but it can take 2000 convergence step to get approximately the same prediction as the last one 6.2, who converge in 10–30 steps. it is thus important to keep in mind that a good cost function can greatly improve computation time.

6.3 Reflection and perspective

Now that we have seen everything, it is time for a conclusion on our reflection and the perspective of what could be interesting to explore next with this kind of predictive controller in Robotran.//

6.3.1 Reflection on using a predictive controller

Like we saw during the entire rapport, a predictive controller is a very versatile tool. We have been changing between a lot of different models, but the logic behind the solver stayed the same. We have also seen that Robotran and Casadi can work together in a well organise fashion without needed to mush change between each model, by taking advantage of the already existing MBSdata structure to write the solver. There is of course still work that need to be done by the user (defining the cost function, creating the personalised constraints, putting the constraint in lbx and ubx vectors, creating the usual user function in Robotran) but most of the solver can adapt itself. If we were using a pure pursuit with PID, the entire PID should have been redone for each model.

Plus, another great advantage of predictive solver is that we can easily control more than one control variables. During this entire rapport we have been using at least 2 controls variables and even 3 for the Roomba pendulum in section 4.5 and that is possible without modifying mush in the code. The user only needs to make sure that the equation of motion generated by MBsysTran take the control variables that the user wanted to control. The rest is handled by the writing of the solver itself and then the inner workings of Casadi. Once again, if we were using a pure pursuit with PID, we would have to create one PID for each control variable. And of course the optimisation of each weight parameter of each PID would be depending on one another, leading to very difficult optimisation of those parameters.

The fairness of such solver is remarkable, as we saw during the parametric study section 6.1. Even when we were changing the rake angle, wish is a big aspect of bike handling, the predictive

solver was able to adapt and deliver us almost the exact same trajectory for all submodels. This enables us to make comparison between the submodel, knowing that a submodels handle better because it is a better geometry and not because the controller is just more capable with that submodel.

On the other hand, we have already talked about the disadvantage of such controllers, the computation time. We have presented in section 4.6 how we made the solver faster, but even with that, a simulation of the Ski-Bike can take 5 to 10 min depending on the difficulty.

In conclusion, a predictive solver is a very versatile, robust, accurate, and fair solver (definition of those word in section 2.1). But you will pay for it in computation time.

6.3.2 The Ski-Bike model

The first thing we could do is improving the model. We have stated some flaws and shortcomings of the Ski-Bike model, like its dynamic instability in section 5.1.3 or its lake of suspension in previous section 6.2. Those change would permit us to study low speed manoeuvres and enable the solver to accelerate and decelerate at its will. Leading of course to a model capable of studying more aspect of the bike handling.

Once we have sorted all those shortcomings, the second thing that would be interesting to do is to see if the controller of the Ski-Bike could potentially control a normal bike. By that, we mean having the equation of motion of the Ski-Bike being use by the Casadi predictive solver, but Robotran is simulating a normal bike. The idea behind this is to save on computation time, the solver having a simpler model to handle, but a model still relatively close to the model we are simulating. Of course, this will lead to the prediction and integrator not fully agreeing with one another, but the error could be small enough that the predictive solver can compensate like in figure 3.18.

6.3.3 Improving the solver

The predictive solver can be improved in many ways. Manly regarding is capabilities and speed of computation.

Capabilities

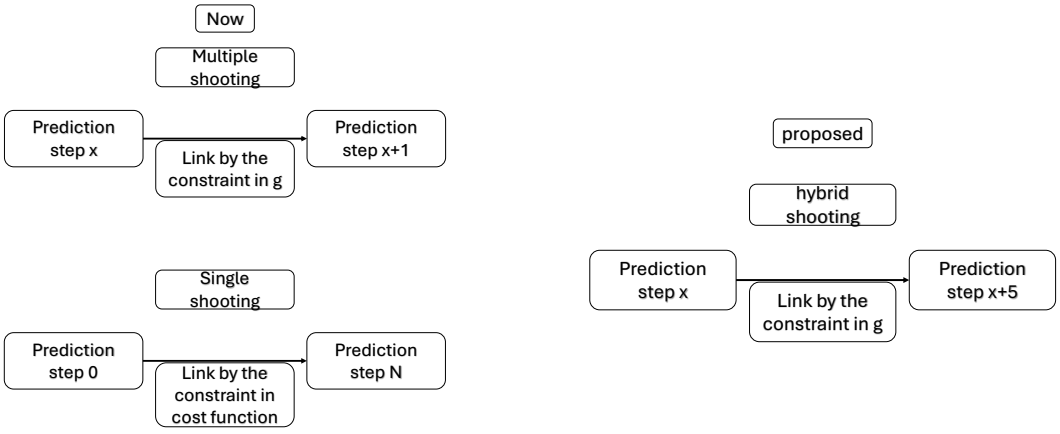
For now, the solver can handle joint forces and external forces, but these are not all the tools that Robotran can support. We have the driven joint that would be easy to added, we have the link forces that will demand more work and Constitutive differential equations that will also demand work. Those are all tools that could be added.

The other obvious thing would be to extend the solver to handle closed loop multibody systems.

The speed of computation

We have already tried to improve the speed of computation of the solver, but there is even more way we could do this. For now, the biggest issue is that each convergence step trying to optimise the prediction take almost 1 second to compute. We should thus work on bringing that time down.

One interesting thing we could do is mixing the multiple shooting approach with a direct shooting approach explain in section 3.1.3 in order to reduce the number of optimisation parameters. As we recall in a multiple shooting approach, each state variable is also an optimisation variable, leading in the Ski-bike model to have 400*(7+2) optimisation variables (N=400)(n_joint = 7)(n_controle = 2). Meanwhile, the single shooting don't stock the trajectory in its prediction, and thus we would have only 400*2 optimisation variables. The idea is the following



(a) the 2 kind of implementation

(b) The proposed implementation

Figure 6.7: Proposed implementation

The idea is, instead of linking each prediction step with its following prediction step, we will link them 5 by 5 and not storing the intermediate prediction step. The condition in g would thus be different.

Now we have this kind of dynamic constraint.

$$g = \begin{bmatrix} \dots \\ (x_1 - (x_0 + RK4(v_x(x_0, y_0, z_0, \theta_0, \phi_0, \gamma_0, \delta_0, v_{x0}, v_{y0}, v_{z0}, \theta_{p0}, \phi_{p0}, \gamma_{p0}, \delta_{p0}, F_{long0}, L_{\delta 0}) \\ (x_2 - (x_1 + RK4(v_x(x_1, y_1, z_1, \theta_1, \phi_1, \gamma_1, \delta_1, v_{x1}, v_{y1}, v_{z1}, \theta_{p1}, \phi_{p1}, \gamma_{p1}, \delta_{p1}, F_{long1}, L_{\delta 1}) \\ (x_3 - (x_2 + RK4(v_x(x_2, y_2, z_2, \theta_2, \phi_2, \gamma_2, \delta_2, v_{x2}, v_{y2}, v_{z2}, \theta_{p2}, \phi_{p2}, \gamma_{p2}, \delta_{p2}, F_{long2}, L_{\delta 2}) \\ (x_4 - (x_3 + RK4(v_x(x_3, y_3, z_3, \theta_3, \phi_3, \gamma_3, \delta_3, v_{x3}, v_{y3}, v_{z3}, \theta_{p3}, \phi_{p3}, \gamma_{p3}, \delta_{p3}, F_{long3}, L_{\delta 3}) \\ (x_5 - (x_4 + RK4(v_x(x_4, y_4, z_4, \theta_4, \phi_4, \gamma_4, \delta_4, v_{x4}, v_{y4}, v_{z4}, \theta_{p4}, \phi_{p4}, \gamma_{p4}, \delta_{p4}, F_{long4}, L_{\delta 4}) \\ \dots \end{bmatrix} \tag{6.4}$$

And if we use the hybrid implementation, we will obtain a single constraint liking x_0 to x_5 directly.

$$g = \begin{bmatrix} \dots \\ (x_5 - (x_0 + RK4(v_x(x_0, y_0, z_0, \theta_0, \phi_0, \gamma_0, \delta_0, v_{x0}, v_{y0}, v_{z0}, \theta_{p0}, \phi_{p0}, \gamma_{p0}, \delta_{p0}, F_{long0}, L_{\delta 0})) \\ +RK4(v_x(x_1, y_1, z_1, \theta_1, \phi_1, \gamma_1, \delta_1, v_{x1}, v_{y1}, v_{z1}, \theta_{p1}, \phi_{p1}, \gamma_{p1}, \delta_{p1}, F_{long1}, L_{\delta 1})) \\ +RK4(v_x(x_2, y_2, z_2, \theta_2, \phi_2, \gamma_2, \delta_2, v_{x2}, v_{y2}, v_{z2}, \theta_{p2}, \phi_{p2}, \gamma_{p2}, \delta_{p2}, F_{long2}, L_{\delta 2})) \\ +RK4(v_x(x_3, y_3, z_3, \theta_3, \phi_3, \gamma_3, \delta_3, v_{x3}, v_{y3}, v_{z3}, \theta_{p3}, \phi_{p3}, \gamma_{p3}, \delta_{p3}, F_{long3}, L_{\delta 3})) \\ +RK4(v_x(x_4, y_4, z_4, \theta_4, \phi_4, \gamma_4, \delta_4, v_{x4}, v_{y4}, v_{z4}, \theta_{p4}, \phi_{p4}, \gamma_{p4}, \delta_{p4}, F_{long4}, L_{\delta 4}))) \\ \dots \end{bmatrix} \tag{6.5}$$

In this implementation all the state 1,2,3,4 will not be stock as optimisation variable and will be lost inside the solver computation. We will thus not be able to put any constraint on those state variables. But we will save memory usage and probably decrease the computation time because we will have fewer optimisation variables and constraints, even if those constraints would be more complexes. Because we can recall that the constraint in g are recursive, meaning that if we are using the value x_i on multiple constraint, we will calculate it only one time. That hybrid implementation would thus be a trade-off between ease of computation (single shooting) and ease of convergence (multiple shooting) and of course we are not oblige to link them 5 by 5 you could link them however you want depending on the situation.

And we could push the idea further by having less control variables, because we have don't need to optimise our L_δ and F_{long} for every 5[ms], we could make do with every 25[ms]. We will thus obtain this

$$g = \begin{bmatrix} \dots \\ (x_5 - (x_0 + RK4(v_x(x_0, y_0, z_0, \theta_0, \phi_0, \gamma_0, \delta_0, v_{x0}, v_{y0}, v_{z0}, \theta_{p0}, \phi_{p0}, \gamma_{p0}, \delta_{p0}, F_{long0}, L_{\delta 0})) \\ +RK4(v_x(x_1, y_1, z_1, \theta_1, \phi_1, \gamma_1, \delta_1, v_{x1}, v_{y1}, v_{z1}, \theta_{p1}, \phi_{p1}, \gamma_{p1}, \delta_{p1}, F_{long0}, L_{\delta 0})) \\ +RK4(v_x(x_2, y_2, z_2, \theta_2, \phi_2, \gamma_2, \delta_2, v_{x2}, v_{y2}, v_{z2}, \theta_{p2}, \phi_{p2}, \gamma_{p2}, \delta_{p2}, F_{long0}, L_{\delta 0})) \\ +RK4(v_x(x_3, y_3, z_3, \theta_3, \phi_3, \gamma_3, \delta_3, v_{x3}, v_{y3}, v_{z3}, \theta_{p3}, \phi_{p3}, \gamma_{p3}, \delta_{p3}, F_{long0}, L_{\delta 0})) \\ +RK4(v_x(x_4, y_4, z_4, \theta_4, \phi_4, \gamma_4, \delta_4, v_{x4}, v_{y4}, v_{z4}, \theta_{p4}, \phi_{p4}, \gamma_{p4}, \delta_{p4}, F_{long0}, L_{\delta 0}))) \\ \dots \end{bmatrix} \tag{6.6}$$

And thus we would have only 400/5*(7+2) optimisation variable and 5 time less constraint to satisfy.

6.3.4 Using the solver for other application

One of the greatest quality of this predictive solver implementation is that it can control every multibody system as long as it is a tree like multibody system. We could thus utilise this exact same predictive solver to control all kind of different multibody system like a car or a train. The only limit here is computation time.

But a study that would be interesting to do if we still want to study a bike would be to simulate the impact of having a cyclist that would actively participate in the driving of the bike. By that, we mean a cyclist that would lean or counterlean in the corners depending on the situation

(a) Leaning in conrenrs

(b) Straight leaning

(c) Counter leaning

Figure 6.8: All type of leaning with a motorbike

Because, as we know from MotoGP, leaning in corners is better for fast cornering and taking the turn just a bit faster. Meanwhile, counterleaning is better to quickly change direction like avoiding a collision or pass through a series of low speeds turns. It would thus be interesting to study such active participation from the cyclist, because this controller can do that with relative ease. Plus, it seems that this topic isn't mush study

For the solver, it is just one more control variable, nothing more. We thus would add a third body representing the cyclist on top of the Ski-Bike connected by a rotoide joint, and we would control the torque on that joint. We have just added one state variable and one control variable, meaning the solver would be perfectly capable of studying such cases.

Appendix A

Casadi

A.1 Casadi's function

In this is example we will recreate the equation of motion of the Roomba model3.1
First thing first, we will have to initialise our symbols. So each state variables and each control variables have to be initialised, and we are using the SX variable of Casadi.

$$x_{sym} = cas.SX.sym('x') \tag{A.1}$$

After initialising them all, they will be stored in an array call either *state* or *controle*. *State* and *control* will thus be SX array. We will now use a third SX array to serve us as a calculation template.

$$dynamic\_equation\_derive = [v_{sym} * cas.cos(\theta_{sym}), v_{sym} * cas.sin(\theta_{sym}), \omega_{sym}, a_{sym}] \tag{A.2}$$

We can directly recall that the equation use in *dynamic_equation_derive* are the equations of motion 3.1. We now have all the symbols store in arrays and a calculation template also in array, we can finally create our Casadi's function.

$$function\_derive = cas.Function('f', cas.vercat(state, controle), dynamic\ equation\ derive) \tag{A.3}$$

That function will now provide us the derivative in time of all our state variables in a SX array, if given an SX array corresponding to $cas.vercat(state, controle)$. Meaning we have:

$$\dot{state} = function\_derive(cas.vercat(state, controle)) \tag{A.4}$$

A.2 Casadi's conditional

As an example, we want to create the following condition in Casadi.

$$\mathcal{F}(c, k) = \begin{cases} np.sin(2 * k) & \text{if } c < 4 \\ 1 & \text{else} \end{cases} \tag{A.5}$$

To do so, we will first have to initiate 2 new symbols like we did in A.1 one for the condition named c and one for the actual calculation named k. We can now define our condition using Casadi

$$z1 = cas.if\_else(c < 4, np.sin(2 * k), 1) \tag{A.6}$$

Now that we have our condition it is time to put it into a Casadi's function,

$$f\_cont = cas.Function("f\_cont", [c, k], [z1]) \tag{A.7}$$

The result will be a function like in A.3 but with an additional condition to verify during the calculation. It is important to note that we can make nested if_else condition in Casadi enabling us the capacity to make complex trajectories.

A.3 Convention of writing X_state using MBSdata

with a system that has only two joints and N=3

$$X\_state = \begin{bmatrix} X_0 & X_4 & X_8 & X_{12} \\ X_1 & X_5 & X_9 & X_{13} \\ X_2 & X_6 & X_{10} & X_{14} \\ X_3 & X_7 & X_{11} & X_{15} \end{bmatrix} \tag{A.8}$$

And that correspond to Robotran convention as

$$X\_state = \begin{bmatrix} q[1]_0 & q[1]_1 & q[1]_2 & q[1]_3 \\ q[2]_0 & q[2]_1 & q[2]_2 & q[2]_3 \\ qd[1]_0 & qd[1]_1 & qd[1]_2 & qd[1]_3 \\ qd[2]_0 & qd[2]_1 & qd[2]_2 & qd[2]_3 \end{bmatrix} \tag{A.9}$$

With $q[1]_0$ being the joint q[1] at prediction step 0 so the initial position of q[1] and $q[1]_1$ being the joint q[1] at prediction step 1 so the first prediction for the joint q[1].

A.4 Impdynared models inner working

We will write here, as an example, the inner working of the inverse pendulum model 4.4.2 working with Impdynared to let you see what the so-called algebraic constraint are. We will thus be using the following implicit equation of motion.

$$\begin{gathered} \dot{x} = v_x \\ \dot{y} = v_y \\ \dot{\phi} = \phi_p \end{gathered}$$

$$\begin{pmatrix} (M+m) & 0 & 0 \\ 0 & m+M & \frac{ML}{2}\cos\phi \\ 0 & \frac{ML}{2}\cos\phi & \frac{ML^2}{3} \end{pmatrix} \begin{pmatrix} \dot{v_x} \\ \dot{v_y} \\ \dot{\phi_p} \end{pmatrix} + \begin{pmatrix} 0 \\ -\frac{ML}{2}\dot{\phi}^2\sin\phi \\ -\frac{ML}{2}g\sin\phi \end{pmatrix} = \begin{pmatrix} F_x \\ F_y \\ 0 \end{pmatrix}$$

Table A.1: equation of motion Inverse pendulum by Impdynared

The writing of the following X_state followed the convention explained in A.3
With N = 3 and using RK4 for the prediction, we have the following X_state and U

$$X\_state = \begin{bmatrix} x_0 & x_1 & x_2 & x_3 \\ y_0 & y_1 & y_2 & y_3 \\ \phi_0 & \phi_1 & \phi_2 & \phi_3 \\ v_{x0} & v_{x1} & v_{x2} & v_{x3} \\ v_{y0} & v_{y1} & v_{y2} & v_{y3} \\ \phi_{p0} & \phi_{p1} & \phi_{p2} & \phi_{p3} \\ a_{x00} & a_{x10} & a_{x20} & a_{x30} \\ a_{y00} & v_{y10} & a_{y20} & a_{y30} \\ \phi_{pp00} & \phi_{pp10} & \phi_{pp20} & \phi_{pp30} \\ a_{x01} & a_{x11} & a_{x21} & a_{x31} \\ a_{y01} & a_{y11} & a_{y21} & a_{y31} \\ \phi_{pp01} & \phi_{pp11} & \phi_{pp21} & \phi_{pp31} \\ a_{x02} & a_{x12} & a_{x22} & a_{x32} \\ a_{y02} & a_{y12} & a_{y22} & a_{y32} \\ \phi_{pp02} & \phi_{pp12} & \phi_{pp22} & \phi_{pp32} \\ a_{x03} & a_{x13} & a_{x23} & a_{x33} \\ a_{y03} & a_{y13} & a_{y23} & a_{y33} \\ \phi_{pp03} & \phi_{pp13} & \phi_{pp23} & \phi_{pp33} \end{bmatrix} \tag{A.10}$$

$$U = \begin{bmatrix} F_{x0} & F_{x1} & F_{x2} \\ F_{y0} & F_{y1} & F_{y2} \end{bmatrix} \tag{A.11}$$

As we can see already, there are a lot more signs than previously when we were using the explicit equation of motion 4.6. This is due to the set of DAE we must solve and the fact that because we are using RK4 we must find 4 different acceleration for each prediction step and for each joint.

The positional and personalised constraint are similar than in 3.4, but we must now add dynamic constraint in g.

$$0 <= \begin{bmatrix} \dots \\ (x_1 - (x_0 + RK4(v_x([x_0, y_0, \phi_0, v_{x0}, v_{y0}, \phi_{p0}, a_{x0i}, a_{y0i}, \phi_{pp0i}, F_{x0}, F_{y0}])))) \\ (y_1 - (y_0 + RK4(v_y([x_0, y_0, \phi_0, v_{x0}, v_{y0}, \phi_{p0}, a_{x0i}, a_{y0i}, \phi_{pp0i}, F_{x0}, F_{y0}])))) \\ (\phi_1 - (\phi_0 + RK4(\phi_{p0}([x_0, y_0, \phi_0, v_{x0}, v_{y0}, \phi_{p0}, a_{x0i}, a_{y0i}, \phi_{pp0i}, F_{x0}, F_{y0}])))) \\ (v_{x1} - ((((v_{x0} + (1/6 * (T * a_{x00}))) + (1/3 * (T * (a_{x01}))))) + \\ (1/3 * (T * (a_{x02})))) + (1/6 * (T * (a_{x03})))) \\ (v_{y1} - ((((v_{y0} + (1/6 * (T * a_{y00}))) + (1/3 * (T * (a_{y01}))))) + \\ (1/3 * (T * (a_{y02})))) + (1/6 * (T * (a_{y03})))) \\ (\phi_{p1} - ((((\phi_{p0} + (1/6 * (T * \phi_{pp00}))) + (1/3 * (T * (\phi_{pp01}))))) + \\ (1/3 * (T * (\phi_{pp02})))) + (1/6 * (T * (\phi_{pp03})))) \\ \dots \end{bmatrix} <= 0 \tag{A.12}$$

And now comes the algebraic corresponding to those dynamic constraint.

$$0 <= \begin{bmatrix} \dots \\ DAE(x_0, y_0, \phi_0, v_{x0}, v_{y0}, \phi_{p0}, a_{x00}, a_{y00}, \phi_{pp00}, F_{x0}, F_{y0}) \\ DAE(RK4\_step(x_0, y_0, \phi_0, v_{x0}, v_{y0}, \phi_{p0}, 1), a_{x01}, a_{y01}, \phi_{pp01}, F_{x0}, F_{y0}) \\ DAE(RK4\_step(x_0, y_0, \phi_0, v_{x0}, v_{y0}, \phi_{p0}, 2), a_{x02}, a_{y02}, \phi_{pp02}, F_{x0}, F_{y0}) \\ DAE(RK4\_step(x_0, y_0, \phi_0, v_{x0}, v_{y0}, \phi_{p0}, 3), \phi_{p0.75}, a_{x03}, a_{y03}, \phi_{pp03}, F_{x0}, F_{y0}) \\ \dots \end{bmatrix} <= 0 \tag{A.13}$$

The constraints present here are only the constraint linking prediction step 0 to prediction step 1, we still need to link every other prediction with one another.

The function $RK4\_step$ mean the individual step in the RK4 integration while the DAE is the equation implicit equation of motion discribe in equation A.1

A.5 Symbolic formulation of Roomba pendulum

Here is the symbolic formulation of the equation of motion for the Roomba pendulum from section 4.5. This symbolic expression is the expression for all joints express in such way:

$$\begin{pmatrix} v_x \\ v_y \\ \theta_p \\ \phi_p \\ \dot{v_x} \\ \dot{v_y} \\ \dot{\theta_p} \\ \dot{\phi_p} \end{pmatrix} = \mathcal{F}([x, y, \theta, \phi, v_x, v_y, \theta_p, \phi_p, F_x, F_y, L_\theta])$$

f:(i0[11])->(o0[8]) SXFunction
$@1 = 0.5, @2 = cos(q_4), @3 = (qd_3 * @2), @4 = (@1 * ((qd_4 * @3) + ((qd_3 * qd_4) * @2))), @5 = cos(q_3), @6 = 9.81, @7 = sin(q_4), @8 = (qd_3 * @7), @9 = ((@6 * @7) + (@1 * (@8 * @3))), @10 = ((@9 * @2) - (((@6 * @2) - (@1 * (sq(qd_4) + sq(@8)))) * @7)), @11 = sin(q_3), @12 = (((@4 * @5) - (@10 * @11)) - F_1), @13 = (@11 * @2), @14 = (@1 * @13), @15 = (@5 + @5), @16 = ((@11+(@13*@2))+((@11*@7)*@7)), @17 = ((@15*@5)+(@16*@11)), @18 = (@14/@17), @19 = (@5 * @2), @20 = (@1 * @19), @21 = ((@15 * @11) - (@16 * @5)), @22 = (@21/@17), @23 = ((((@11+@11)*@11)+(((@5+(@19*@2))+((@5*@7)*@7))*@5))-(@21*@22)), @24 = ((@20+(@21 * @18))/@23), @25 = ((@12 * @22) - (((@4 * @11) + (@10 * @5)) - F_2)), @26 = ((@1 * @11) * @7), @27 = ((@1*@5)*@7), @28 = (@26-(@27*@22)), @29 = (@27/@17), @30 = ((@26-(@21 * @29))/@23), @31 = (((0.2+((@1*(@1*@7))*@7))-(@27*@29))-(@28*@30)), @32 = (((@28 * @24) - (@27 * @18))/@31), @33 = (((@12 * @29) - (((@1 * @4) * @7) - F_3)) - (@30 * @25)), @34 = (@20+(@14*@22)), @35 = ((@34*@30)-(@14*@29)), @36 = (((((@1*@9)+(@12*@18))+(@24*@25))-(@32*@33))/(((0.25-(@14*@18))-(@34*@24))-(@35*@32))), @37 = ((@33-(@35*@36))/@31), @38 = (((@25-(@28*@37))+(@34*@36))/@23), [qd_1, qd_2, qd_3, qd_4, (-((((@12 + (@14 * @36)) + (@27 * @37)) + (@21 * @38))/@17)), @38, @37, @36]$

UNIVERSITÉ CATHOLIQUE DE LOUVAIN
École polytechnique de Louvain
Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | **www.uclouvain.be/epl**