Comparison of Optimal Control Frameworks in the Context of Bipedal Walking

Julian Eßer

March 30, 2020

Contents

Chapter 1

INTRODUCTION

Project Objectives

The primary goal of this two-month project is to gain insights about the bipedal walking capabilities of two open-source frameworks for optimal control of robots (Crocoddyl, Drake).

In particular, this involves

1. Installing and familiarizing with the libraries description of dynamical systems and optimization problems,
2. Working with the provided examples, especially those related to legged-locomotion,
3. Integrating a universal description of the RH5 robot and test the solvers
4. Comparing the frameworks in terms of suitability for performing bipedal walking tasks.

Organisation

All tasks related to this project are handled as issues and managed centrally from the **related repository** [2]. Even though most of the implementation is done in the forked repositories of the according frameworks, this seems to be a clear way of organizing.

Chapter 2

CROCODDYL - LAAS-CNRS

The first framework we were interested to analyze is named Crocoddyl and was first published within 2019. Crocoddyl turned out to provide a huge bandwith of examples related to legged robots (quadrupeds, bipeds) performing diverse movements under contact (simple walking, quadrupedal gait variations, jumping).

This chapter briefly introduces the library functionalities and summarizes the insights and results gained during the integration of a new robot model. Through the course if this work, a simplified model of the RH5 robot has been integrated into Crocoddyl and been used to generate simple bipedal walking trajectories.

2.1 Introduction

2.1.1 Motivation

Crocoddyl is an **optimal control library for robot control under contact sequence**. Its solver is based on an efficient Differential Dynamic Programming (DDP) algorithm. Crocoddyl computes optimal trajectories along with optimal feedback gains. It uses Pinocchio for fast computation of robot dynamics and its analytical derivatives [1].

Crocoddyl is focused on multi-contact optimal control problem (MCOP) which has the form:

$$\mathbf{X}^*, \mathbf{U}^* = \begin{Bmatrix} \mathbf{x}_0^*, \cdots, \mathbf{x}_N^* \\ \mathbf{u}_0^*, \cdots, \mathbf{u}_N^* \end{Bmatrix} = \arg\min_{\mathbf{X},\mathbf{U}} \sum_{k=1}^{N} \int_{t_k}^{t_k+\Delta t} l(\mathbf{x}, \mathbf{u}) dt$$

subject to

$$\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x}, \mathbf{u}),$$
$$\mathbf{x} \in \mathcal{X}, \mathbf{u} \in \mathcal{U}, \boldsymbol{\lambda} \in \mathcal{K}.$$

where

- the state $\mathbf{x} = (\mathbf{q}, \mathbf{v})$ lies in a manifold, e.g. Lie manifold $\mathbf{q} \in SE(3) \times \mathbb{R}^{n_j}$, n_j being the number of degrees of freedom of the robot.
- the system has underactuacted dynamics, i.e. $\mathbf{u} = (\mathbf{0}, \boldsymbol{\tau})$,
- $\mathcal{X}$, $\mathcal{U}$ are the state and control admissible sets, and
- $\mathcal{K}$ represents the contact constraints.

Note that $\boldsymbol{\lambda} = \mathbf{g}(\mathbf{x}, \mathbf{u})$ denotes the contact force, and is dependent on the state and control.

2.1.2 Features

According to the description in the Github repository [1], it comprises the following features:
Crocoddyl is **versatible**:

- various optimal control solvers (DDP, FDDP, BoxDDP, etc) - single and multi-shooting methods
- analytical and sparse derivatives via Pinocchio
- Euclidian and non-Euclidian geometry friendly via Pinocchio
- handle autonomous and nonautomous dynamical systems
- numerical differentiation support

Crocoddyl is **efficient** and **flexible**:

- cache friendly,
- multi-thread friendly
- Python bindings (including models and solvers abstractions)
- C++ 98/11/14/17/20 compliant
- extensively tested

2.2 How-To

2.2.1 Install

Two ways to go

Basically there are existing two ways of installing Crocoddyl:

- Option 1: Installation via the *robotpkg* package manager
- Option 2: Installation from source

I personally would recommend the installation through *robotpkg*, since it preserves you from dealing with the multiple dependencies of Crocoddyl and therefore seems to be the faster approach. Generally you should decide beforehand which python version you want to use. This effects the robotpkg version as well as the export of the PYTHONPATH variable.

Installation via robotpkg (preferred)

Steps for installing via robotpkg according to the installation section of [1]

1. Add robotpkg as source repository to apt:

```
sudo tee /etc/apt/sources.list.d/robotpkg.list <<EOF
deb [arch=amd64] http://robotpkg.openrobots.org/wip/packages/debian/pub $(lsb_release -sc) robotpkg
deb [arch=amd64] http://robotpkg.openrobots.org/packages/debian/pub $(lsb_release -sc) robotpkg
EOF
```

2. Register the authentication certificate of robotpkg:

   ```
   curl http://robotpkg.openrobots.org/packages/debian/robotpkg.key | sudo apt-key add -
   ```

3. You need to run at least once apt update to fetch the package descriptions:

   ```
   sudo apt-get update
   ```

4. The installation of Crocoddyl:

   ```
   sudo apt install robotpkg-py27-crocoddyl # for Python 2
   sudo apt install robotpkg-py36-crocoddyl # for Python 3
   ```

5. Finally you will need to configure your environment variables (watch out for the python version!), e.g.:

   ```
   export PATH=/opt/openrobots/bin:$PATH
   export PKG_CONFIG_PATH=/opt/openrobots/lib/pkgconfig:$PKG_CONFIG_PATH
   export LD_LIBRARY_PATH=/opt/openrobots/lib:$LD_LIBRARY_PATH
   export PYTHONPATH=/opt/openrobots/lib/python3.6/site-packages:$PYTHONPATH
   ```

(Installation from source)

If you prefer installing Crocoddyl from source, the following steps should do the work:

```
git clone https://github.com/loco-3d/crocoddyl.git
git submodule update --init
mkdir build && cd build
export PKG_CONFIG_PATH=/opt/openrobots/lib/pkgconfig
cmake -DCMAKE_INSTALL_PREFIX=/opt/openrobots  ..
make
sudo make install
```

Additionally you will have to install the dependent libraries (i.e. pinocchio, example-robot-data (optional for examples, install Python loaders), gepetto-viewer-corba (optional for display), jupyter (optional for notebooks) and matplotlib (optional for examples) and fix the incude paths.

For Displaying Results: Additionally Install Gepetto-Viewer

In order to see not just 2-dimensional plots, but also observe the 3D robot behaviour, you additionally have to install the gepetto-viewer.

```
sudo apt install robotpkg-py36-qt4-gepetto-viewer-corba
```

2.2.2 Running the Examples

Since the installation through robotpkg did not provide you with the examples from the git repository, you should clone the repo [1] for getting the data. You do not have to build the library, since it already is installed. In the cloned repository go to */examples*. For running e.g. the bipedal walking example, just type

```
python3 bipedal_walk.py
```

and you will see the calculations for optimal gait trajectories running in the console. The examples provide a *plot* and *display* argument. In order to display the 3D results and also plot some data, just do

```
gepetto-gui
```

for starting the 3D environment. Then, in another terminal, run the example:

```
python3 bipedal_walk.py display plot
```

2.3 Abstract Workflow

For each node (i.e. each timestep) of the optimal control problem,

1. Load robot data (URDF, SRDF, Meshes)
2. Define Action Models (Dynamics+Costs) for running and terminal states
   - Setup a cost model
   - Add the desired cost functions (state, control, frame-placement etc.)
   - Calculate Integrated & Differential Action Model (IAM/DAM) based on the model
3. Define the optimal control problem (knots+IAMs, initPose)
4. Solve the Shooting Problem

2.4 Issues and Insights

2.4.1 Issues Encountered

Since Crocoddyl currently is under active development, there frequently will occure smaller incompatibilities because of versioning issues.This is a brief overview of emerged difficulties:

- Python versioning errors in the examples.

  The examples most often are written for python2, which means that if you are under python3, you will have to adapt some commands (e.g. lists handling, matplotlib, print).

- Crocoddyl versioning errors in the examples.

  Since Crocoddyl depends on other libraries (i.e. Pinnochio, example_robot_data), there sometimes occured errors with the class because they were not updated.

- Confusions displaying the results via the Gepetto-Viewer

  The Gepetto-Viewer is used for displaying the robots and resulting trajectories from optimization. The examples only contain out of the box solutions. If one wants to simply display a robot in some specified pose (e.g. the initial pose) the following, quite unintuitive, commands have to be applied:

  ```
  display = crocoddyl.GepettoDisplay(rh5_legs, 4, 4, frameNames=[rightFoot, leftFoot])
  display.display(xs=[x0])
  ```

2.4.2 Limitations

- No possibility to model 4 contact points per foot since the solver can't properly handle redundant constraints.

2.4.3 Cost Functions

Notes:

- The cost function can contain multiple *cost items* (i.e state/control error, frame displacements or center of mass tracking).
- Weights are considered in the costs via scalar multiplication with the identity matrices (Ix, Ixx etc.) of the according cost item.
- These weighted matrices of cost items are simply summed up within a *costModelContainer*.

2.4.4 Joint Limits

- Input Data: Within the URDF file, for each joint there are specified the
  - torque limit (effort),
  - position limits (lower, upper),
  - velocity limit.
- These limits are not automatically taken into account in Crocoddyl when solving a shooting problem, but explicitly have to be adressed.
- **Torque Limits**: Require the use of specific solvers, standard ddp is not sufficient. Implemented solvers that can handle torque limits explicitly are:
  - BoxDDP (Compare Tassa method [4])
  - BoxFDDP (Novel solver that is under development at LAAS)

  State Limits (Pos/Vel): Position and Velocity limits can be handled via penalization, i.e. added as cost terms to the optimization problem.

2.4.5 Introduction: Bipedal Walking in Crocoddyl

- The desired foot step plan simply is an equally distributed line of knots specified via the desired stepheight and steplength.
- A long walk consists of multiple gaitphases, each phase is a single shooting problem.
- These problems are generated with *createWalkingProblem()* involving one left and one right foot step.
- Each shooting problem contains various locomotion phases
  - Double support at beginning (both legs on ground) via *createSwingFootModel()*
  - Right step (Swing-up and swing-down phase equally distributed) init via *createFootstepModels()*
  - Double support again

  - Left step
- In the end, all knots of all phases are basically one *swingFootModel*. They only vary in the adressed foot, and if a CoM task or a swingFootTask is set.
- The *swingFootModel* is an IAM containing a
  - 6D multi-contact model,
  - Cost model and
  - Differentiation (DAM) and Integration (IAM) routines.

2.4.6 Cost Function for Bipedal Walking

The main cost items (weights in brackets)utilized in the example are

- (1e6) CoM tracking
- (1e6) Foot tracking
- (1e1) Friction cone per contact point
- (1e1) State regression, i.e. penalize joint variance
- (1e-1) Control input regression, i.e. penalize control effort

2.4.7 Multi-Contact Model

Previous: Assume foot to have a point contact

- Each *swingFootModel*, meaning each knot of the OC, includes at least one individual contact model.
- The number of contact models depends is specified via *supportFootIds*, i.e. the number of foots that currently are not in the air (e.g. biped: 1 for swing-phase, 2 for double-support).
- For each point of one foot that actually does touch the ground, a contact has to be added to the according contact model (specified via frame ID)
- In the bipedal examples they assume a point-contact, i.e. adding only one contact per supporting foot.
- The point-contact is specified via the name of the according link from the URDF.

[aborted] Extension: Consider 4 point contact per foot

The key implementation idea:

- Each contact model for one supporting foot now contains four contact items instead of one
- The cost model contains now four cost items for the four friction cones instead of just one

The simulation did not run at all with this kind of model, neither did the solver showed any convergence. An explanation might be the following:
By providing 4/8 contact items (one support foot, double support while standing) with each 6 constraints (position, rotation) we overconstrain the dynamics of the system. The underlaying dynamics solver from the crocoddyl library does not seem to be able to handle these redundant constraints properly and fails.

2.5 Working with the Examples

This section contains a brief overview of some of the examples that have been modified. Please note that many **additional explanations have been added in the commentaries of the examples**. All figures and additionaly videos are contained within the /media directory of the repo [2].

As stated in the ReadMe of the Repo, Crocoddyl comes with some introductory examples that are written as Jupyter notebooks (.ipynb). While

`examples/notebooks/introduction_to_crocoddyl.ipynb`

offers a more conceptual overview about crocoddyl, other ones represent basic underactuated systems (e.g. Cartpole swing-up, Bipdeal Walking).

2.5.1 Manipulator: Multi-Point Trajectory

The task in the tutorial

`examples/notebooks/arm_manipulation.ipynb`

was to find an optimal trajectory for a manipulator from an initial configuration to a target point (red ball).

Extending the Example

I extended this example to a multi-point optimal control problem by considering four targets to reach in a row. This extended example can be found in

`/examples/arm_manipulation_trajectory.py`

Differences to the existing one-target example include:

- Defining an array of the four targets in space
- Setup individual cost functions for each of the sequences
- Setup running and terminal models for the sequences and finally
- Define the shooting problem as row of these sequences
- Optimize the weights of the cost functions successively for the four sequences.

Results: Multiple Targets

2.5.2 Talos Legs: Bipedal Walking

The Crocoddyl library offers an example for bipedal walking with the lower body of the Talos [3] Robot. The results of the solved trajectory can be found in Figure 2.2.

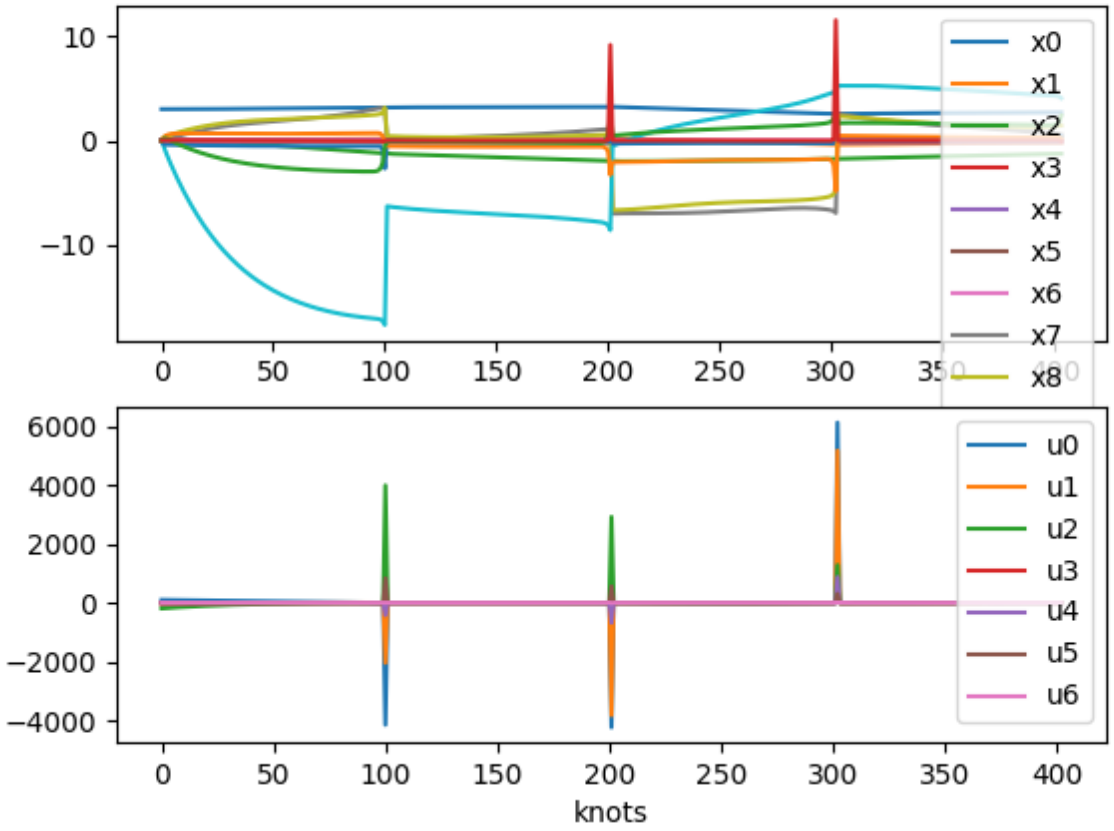


Figure 2.1: Multi-Point Optimal Control of the Manipulator for reaching four targets. The high-control peaks could be handled by a more advanced solver (e.g. box-ddp), but were not performed here.

2.6 Results: RH5 Legs

2.6.1 Navigating the Files

The main file for testing RH5 offers several options for setting up, analyzing and constraining the optimization problem of bipedal walking including

- Choosing a desired URDF
- Specifying gait length and variations in step length/height
- Vary the initial pose of the robot
- Constraining the torque input (Use different solver)
- Defining singular or multiple point contacts per foot (Use different class from biped utils)

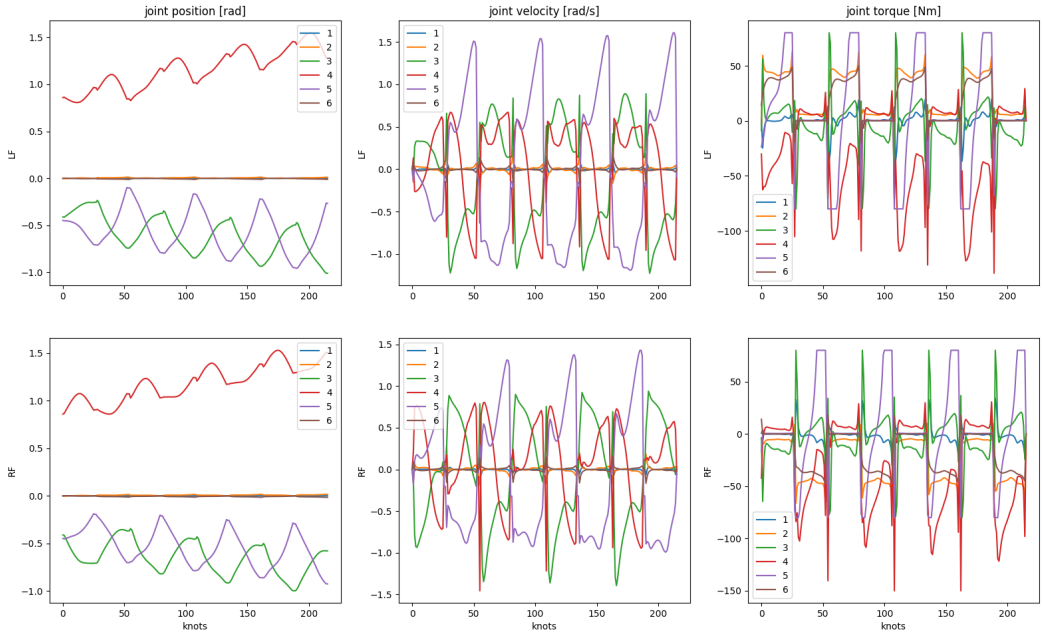


Figure 2.2: Results for the talos legs performing a walk with three gait-phases.

2.6.2 Integration of the RH5 Legs into Crocoddyl

The main issue that had to be solved was related to the underlying URDF file of the robot. In particular:

- Choosing one of our several files (abstract-smurf)
- Cutting out everything apart from the legs and the root joint
- Fixing the mesh file paths:
  We usually define the path relative to the URDF file location, e.g.

  `"../meshes/stl/RH5_Root_Link.001.stl".`

  The integrated URDF parser in Crocoddyl instead, expects the path specified via the package URI, e.g.

  `"package://abstract-smurf/meshes/stl/RH5_Root_Link.001.stl".`

- Adjust the contact frames for the Walking Problem.

2.6.3 Performing a Full Gait

Results for a full gait (6 consecutive steps) are shown in Figure 2.4.

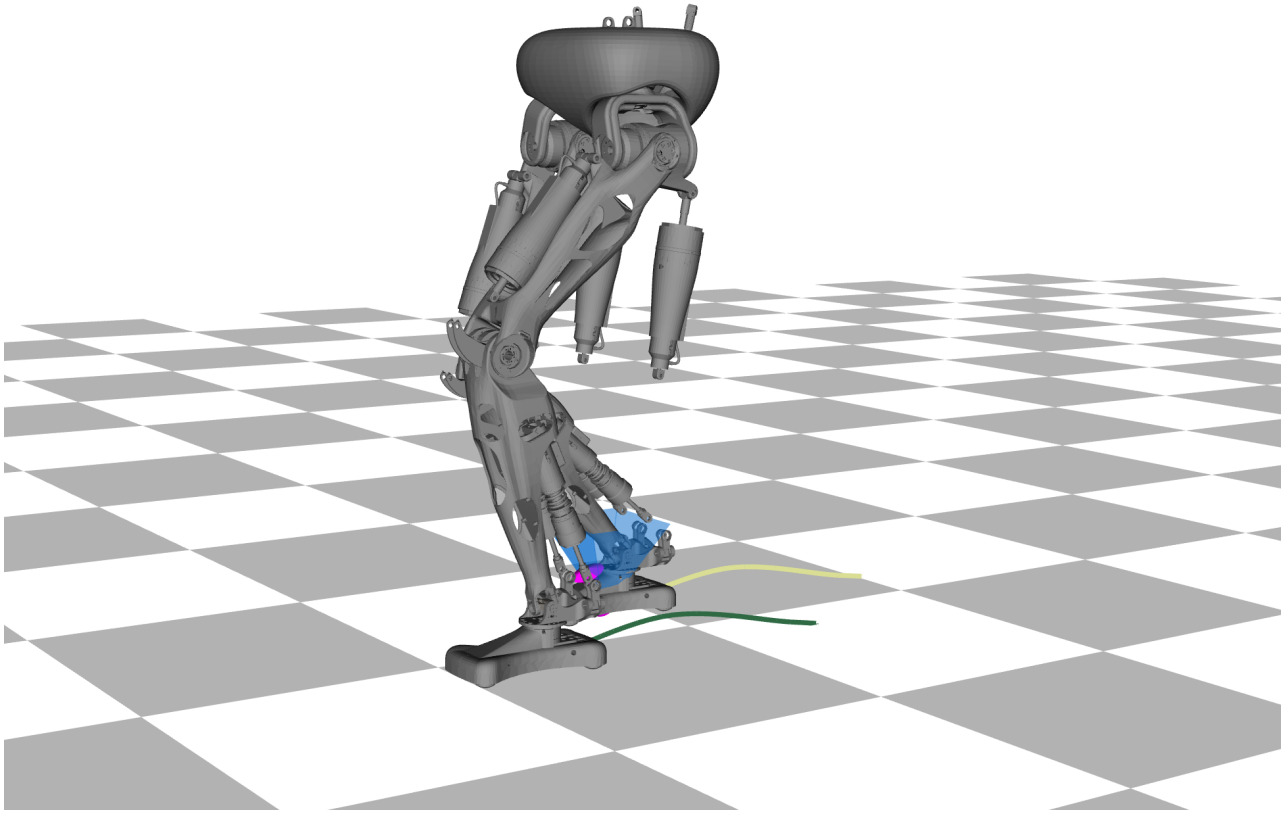


Figure 2.3: Visualization of the RH5 legs using the gepetto-viewer.

2.6.4 Torque-Constrained Full Gait

It is possible to constrain the input torques via the limits that are parsed from the URDF. However, the correct solver (box-ddp) needs to be applied to the OC problem for considering these limits. Results are shown in Figure 2.5

2.6.5 Initial Pose Variants: Starting Near the Zero Configuration

The full gait has been initialized according to the pose specifications from the RH5 DFKI smurf file. In this and the following section, two solutions for other initial poses are presented: One near the zero configuration and one at the zero configuration.

2.6.6 Initial Pose Variants: Starting At the Zero Configuration

Initializing at the zero configuration turns out to be more critical. Sometimes the solver converges to a feasable solution (Figure 2.7), sometimes it does not find a proper solution (Figure 2.8).

2.6.7 Towards Periodic Joint Trajectories

All previous results share a common pattern: While the trajectories for joint velocity and input torque are mostly periodic, the joint trajectories appear to be not. Especially, the absolute value of several joints increases over time. Investigations have shown that this pattern can be prevented by modifying the cost function. When it comes to joint movements, the state regression penalization is crucial.

Increasing the state regression cost item by one dimension (from 1e1 to 1e2) leads to

- Periodic joint trajectories and
- Bounded joint limits.
- Not reaching the desired step height.

Consequently, this approach is appeared not to be the right way to go. Alternative approaches could include setting up an equality constraint for the initial and terminal pose. The (preliminary) results are shown in Figure 2.9.

2.7 Results: RH5 Legs + Torso

The results from the last section give a first impression, of how optimal control for bipedal walking applied to RH5 can look like. The natural next step is to extend the analysed robot model with the torso. This an especially useful step towards first real-world experiments on the robot, since the computational unit is integrated within the torso and it therefore would be difficult to only control the RH5 legs.

Note on the walking speed of videos and related CoM shifting:

The real walking speed (0.75m/s) is slowed down within the videos for proper analysis of the systems behavior. Since the desired speed is this high, the resulting CoM nearly doesn't shift at all, which is one of the key properties of dynamic walking. The same parameters hold for the simulation results for just the RH5 legs.

2.7.1 Necessary Adjustments

Including the torso in our analysis, we in particular include three additional DoFs (Body-Roll, Body-Pitch, Body-Yaw) that need to be considered. The following adjustments were taken:

- Create an appropriate URDF version based on the abstract-urdf
- Consider the dimensional changes of the joint space within the implementation
- Adjust the initial pose of the robot
- Extend the plotting functionalities for further analysis

2.7.2 Adopted Findings from Previous Analysis of the Legs

Within the previous lection different aspects have been covered. For the continuing analysis we will cherry-pick and apply the most promising ones. In particular, this means to

- Performing a full gait (6 consecutive steps),

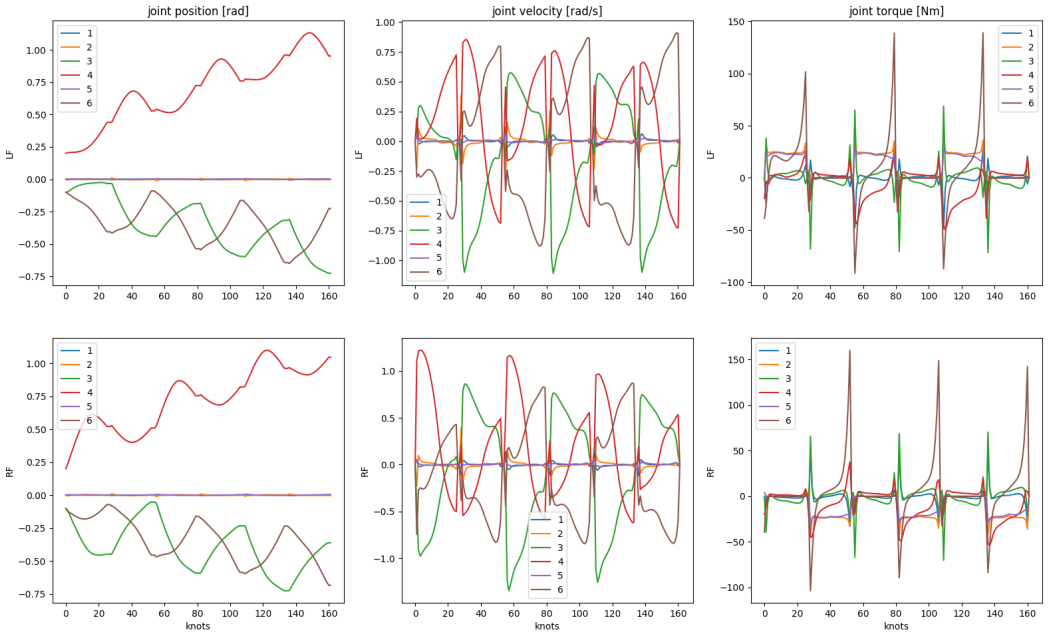
- Consider bounded input torques, but only to the extend defined in the URDF (no additional artificially reduction)
- Choose an initial pose that is as close as possible to the zero configuration
- Have special emphisis on the state weights in order to achieve periodic joint movements.

2.7.3 Performing a Full Gait

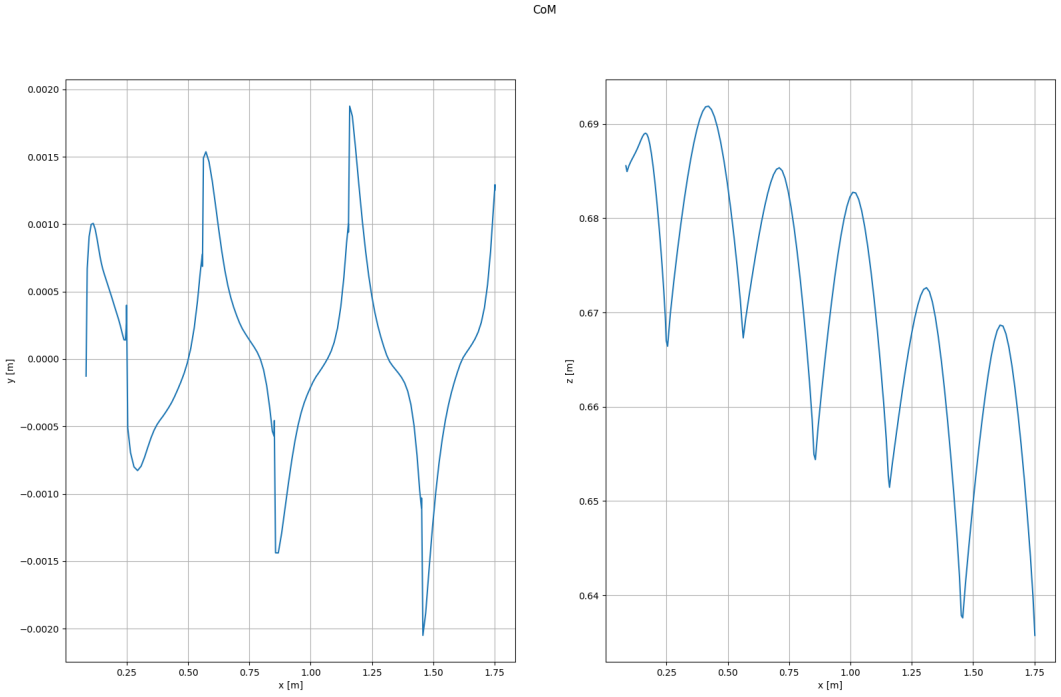
Results for a full gait (6 consecutive steps) are shown in Figure 2.12. This first shot without proper adjustment of the cost functions revealed an interesting pattern: Although stable, the body pitch (joint 2 in the upper plot) increases within each step and therefore is not periodically. As can be seen in the related video, this means the torso is going to 'fall' with each step more into the right direction.

2.7.4 Towards a Stable Torso

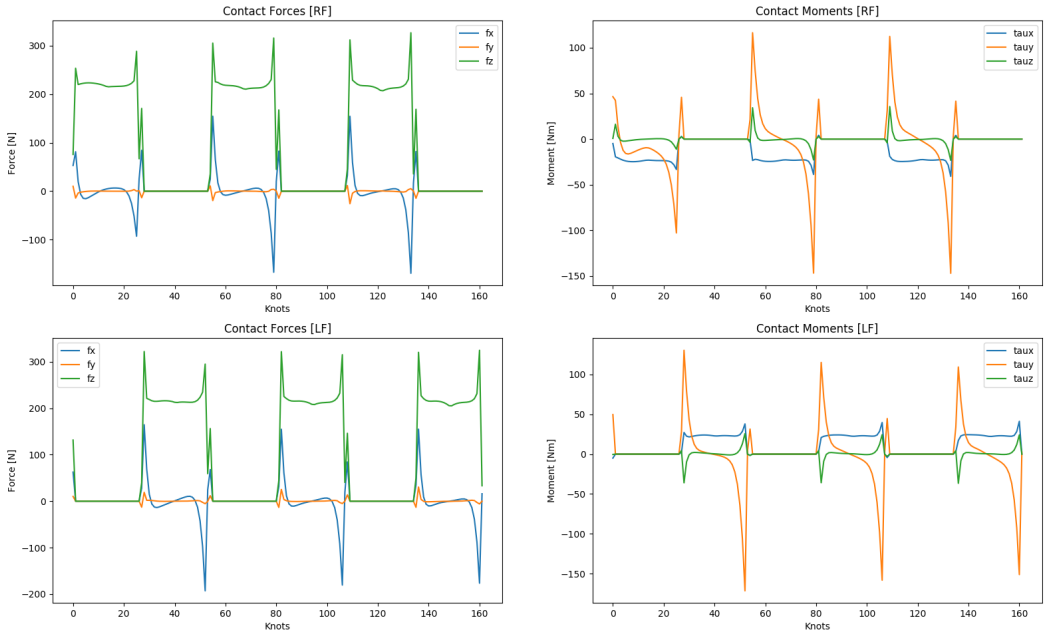
In order to handle the appeared problems with the torso drift during the walk, we have to adjust the cost functions. Further analysis revealed, that a modification of the individual weights for the state regression cost item can lead to a stabilised upper body.



(a) Solution for states and torques.

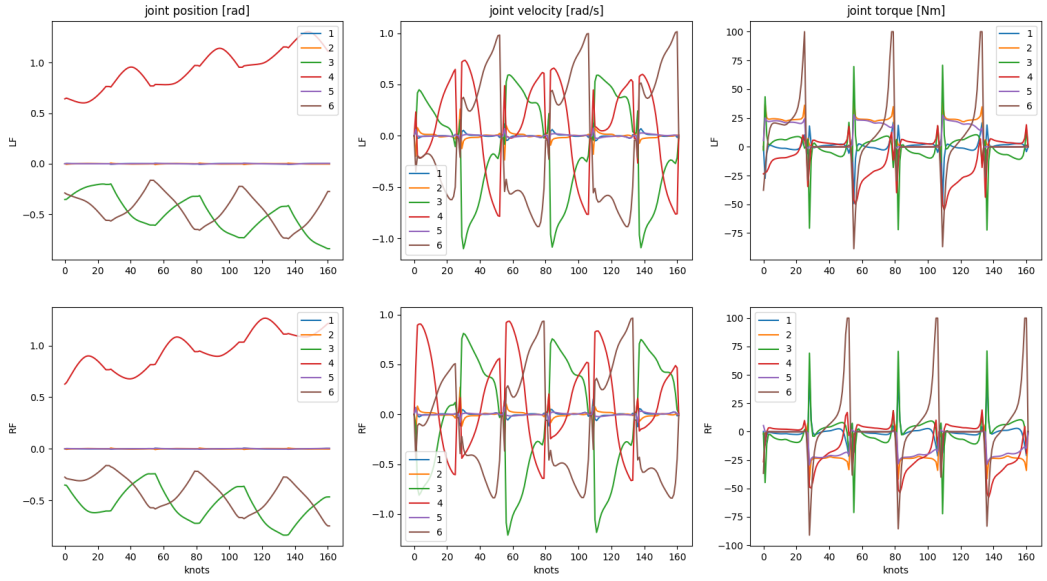


(b) CoM results.

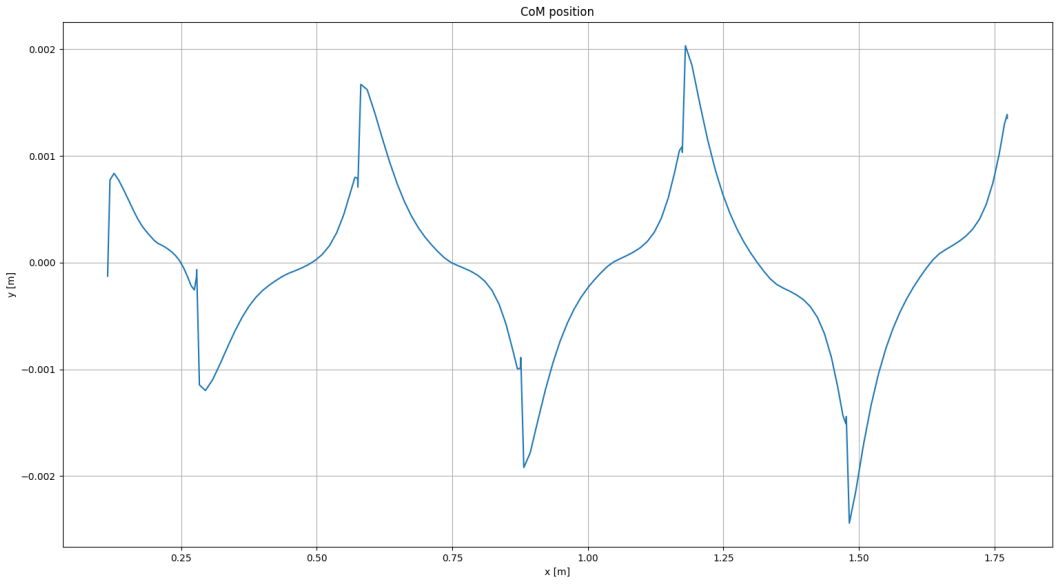


(c) Contact wrenches.

Figure 2.4: Results for a walk with three gait-phases. The walk is implemented as a sequence of three consecutive shooting problems similar to the 2 steps task.

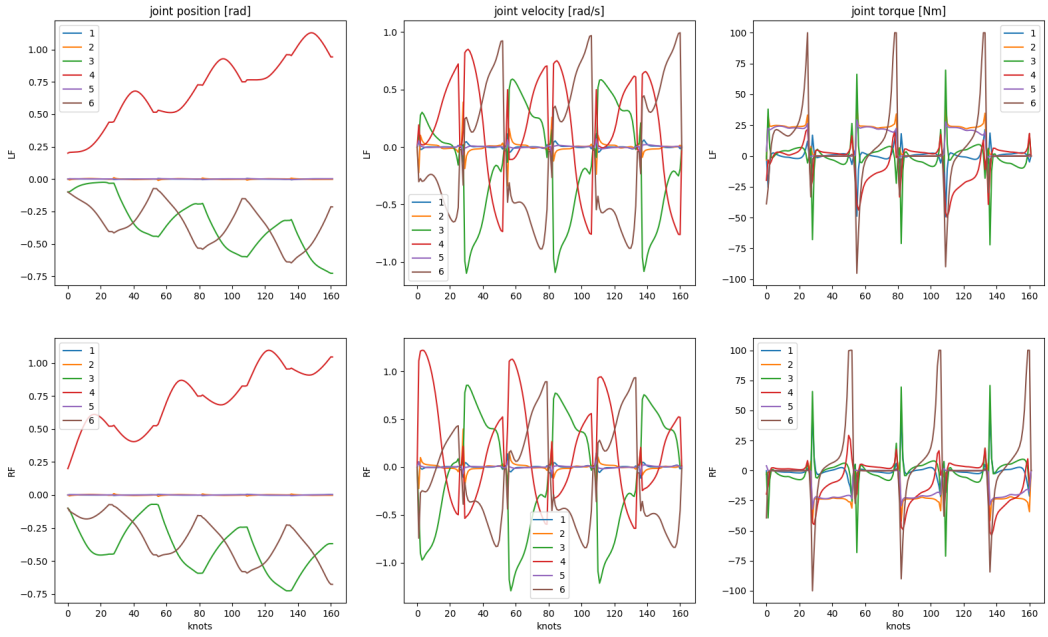


(a) Solution for states and torques.



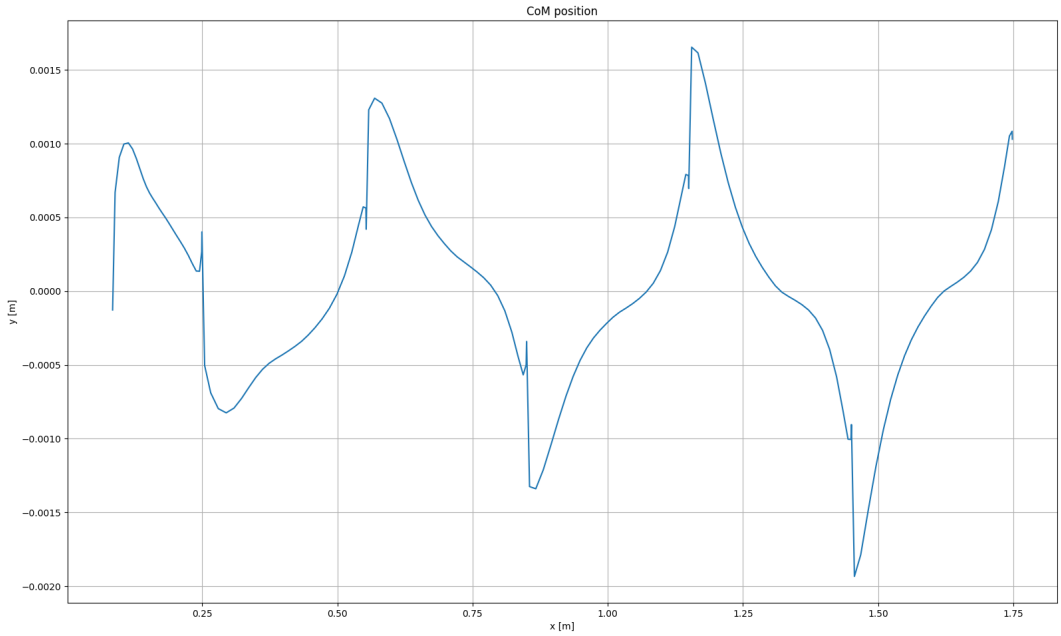
(b) CoM results.

Figure 2.5: Results for full gait with bounded input torques. To be even more restrictive, the available torques form the URDF have been reduced artificially by 50 percent. The effect becomes clear for e.g. the AnkleFT.



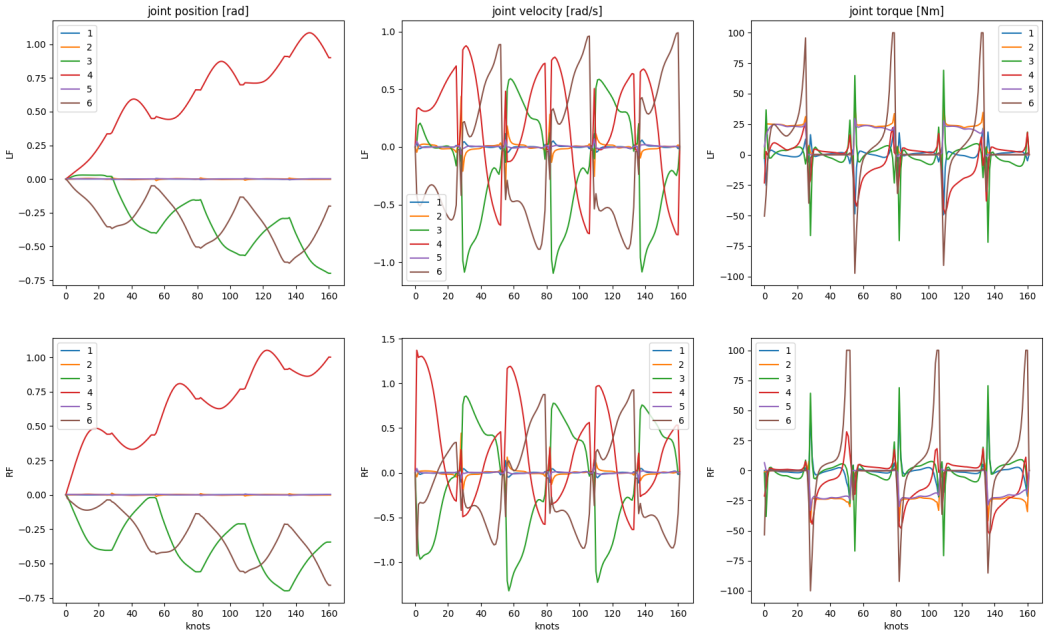
(a) Solution for states and torques

.



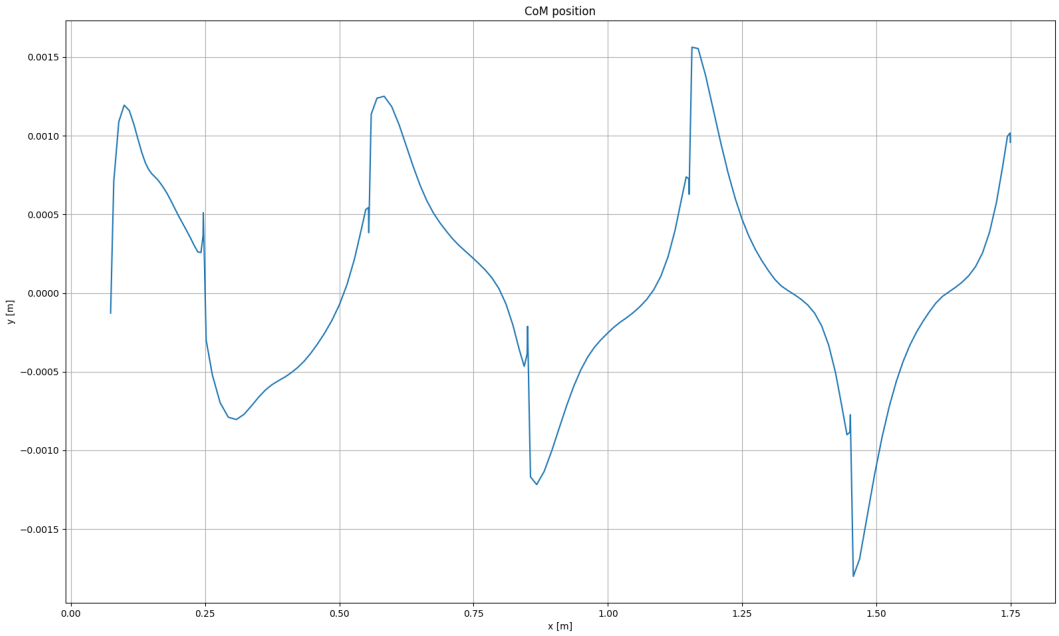
(b) CoM results.

Figure 2.6: Results for the full gait with initial pose **near** the zero configuration. q0=[0,0,-0.1,0.2,0,-0.1, 0,0,-0.1,0.2,0,-0.1] where n=12 and represents the joint angles [rad] for the left and right leg.



(a) Solution for states and torques

.



(b) CoM results.

Figure 2.7: Results for the full gait with initial pose **at** the zero configuration.

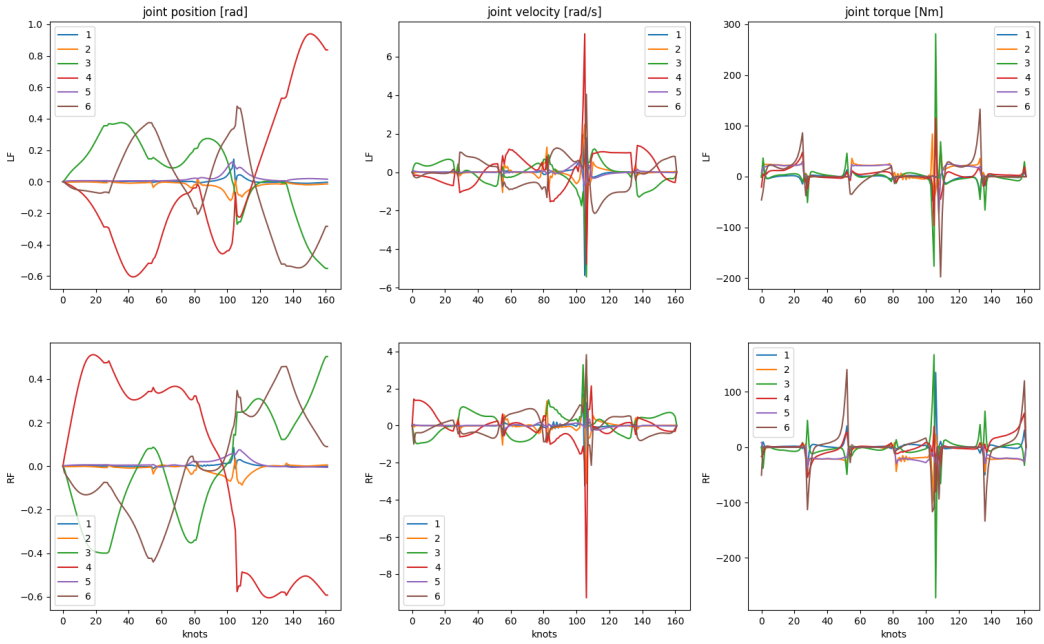
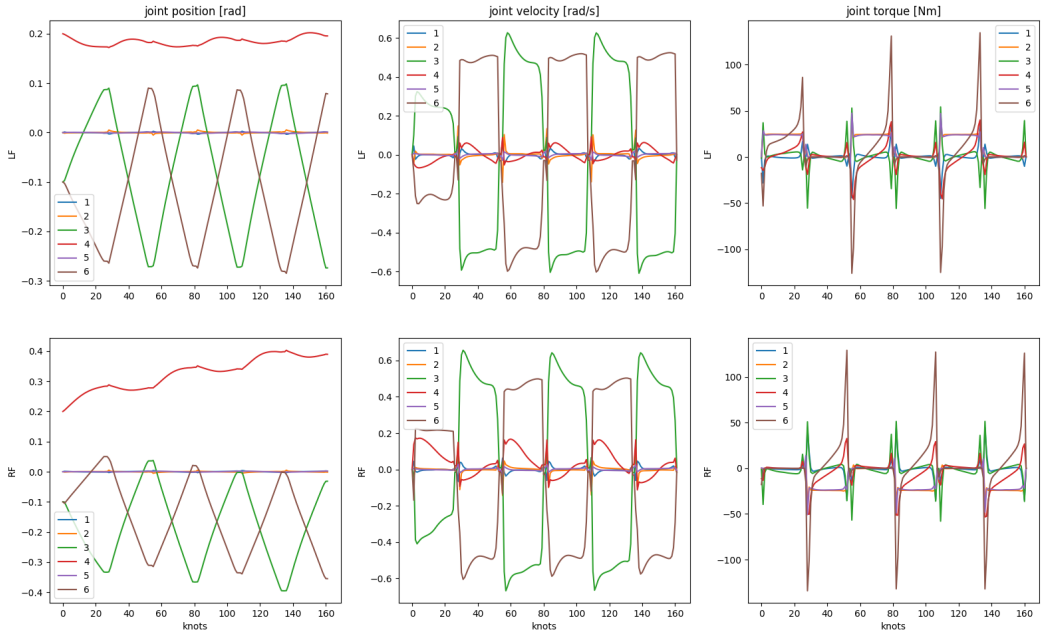
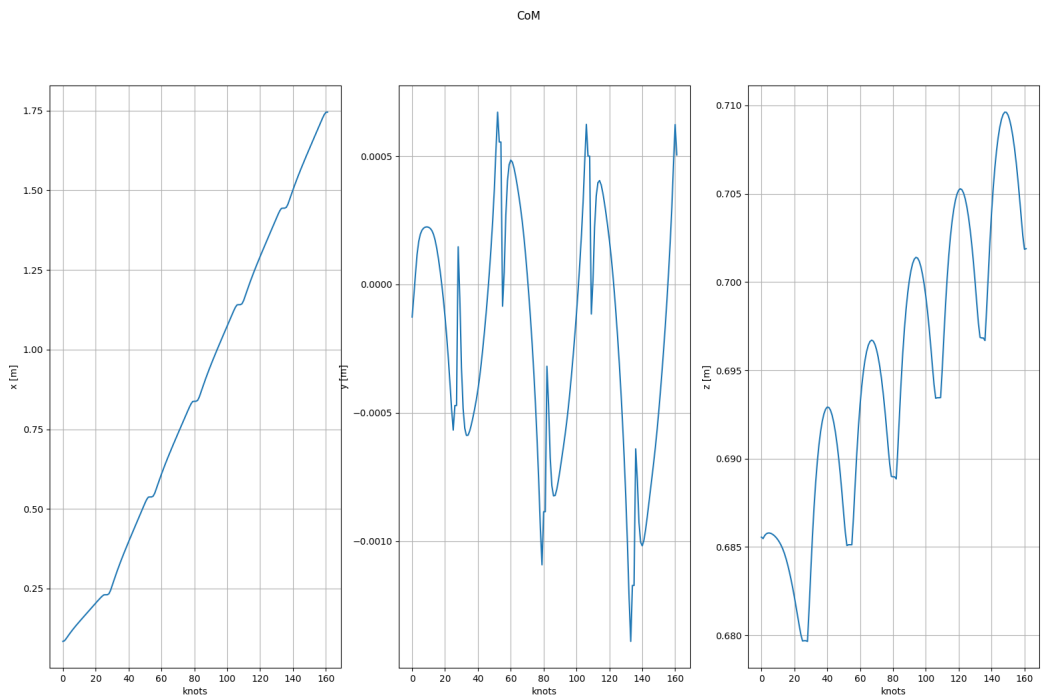


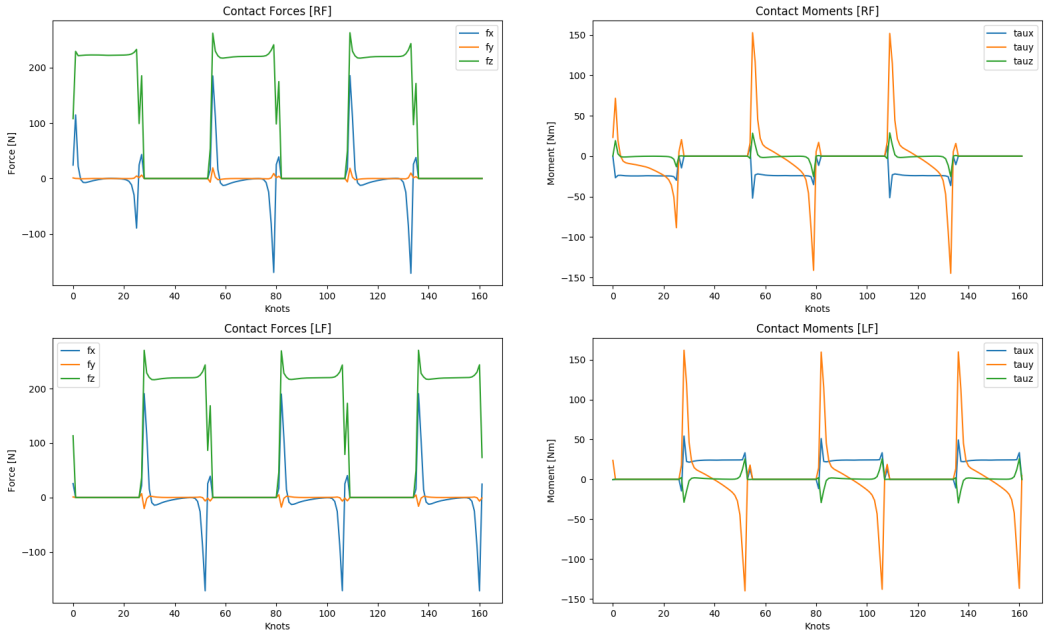
Figure 2.8: Another result for the full gait with initial pose **at** the zero configuration. The solution appears to be instable and would exceed the torque limits (see video).



(a) Solution for states and torques



(b) CoM results.



(c) Contact wrenches.

Figure 2.9: Increasing the state regression cost item leads to a more periodic and bounded solution regarding the joint position trajectories.

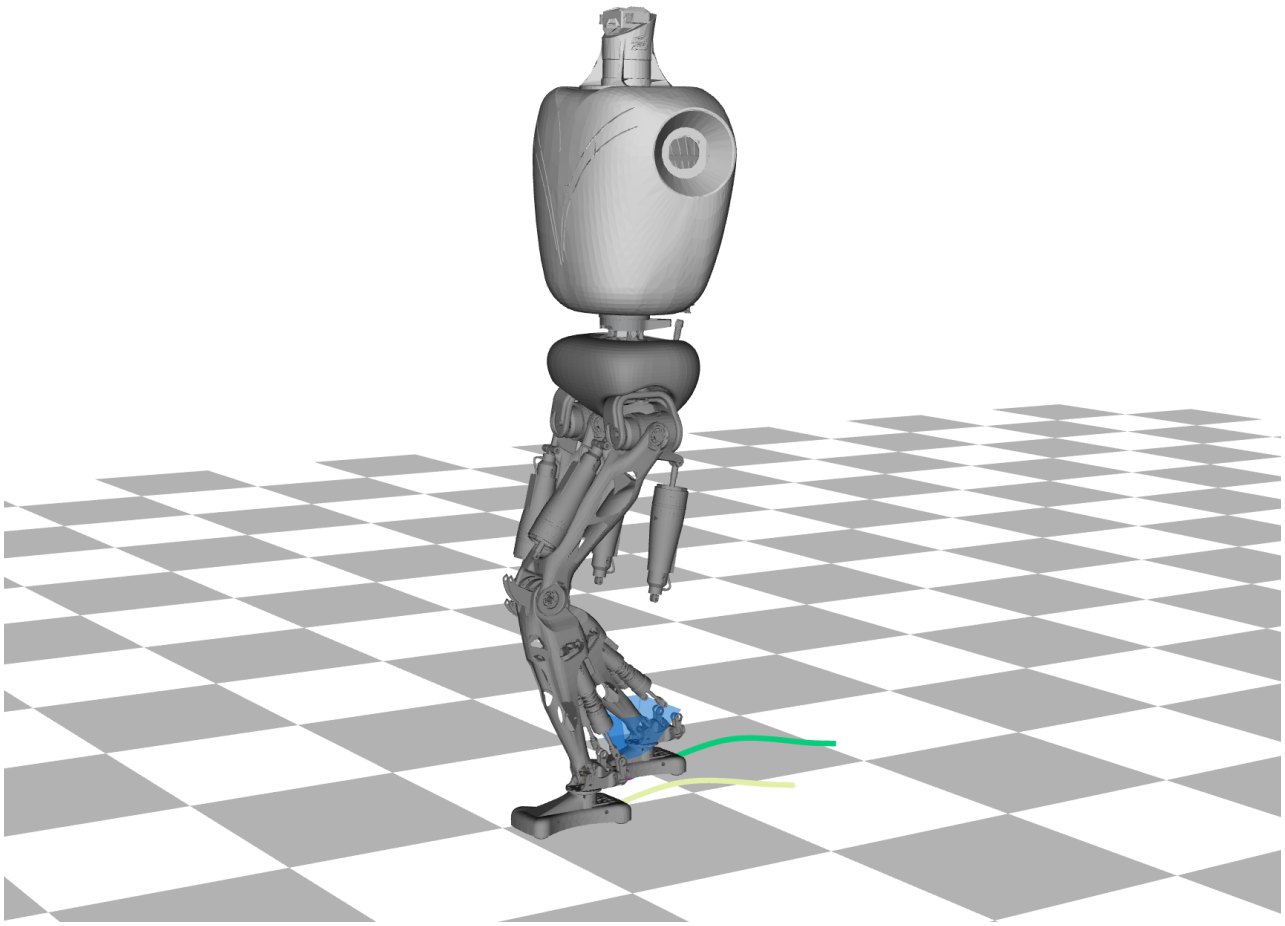
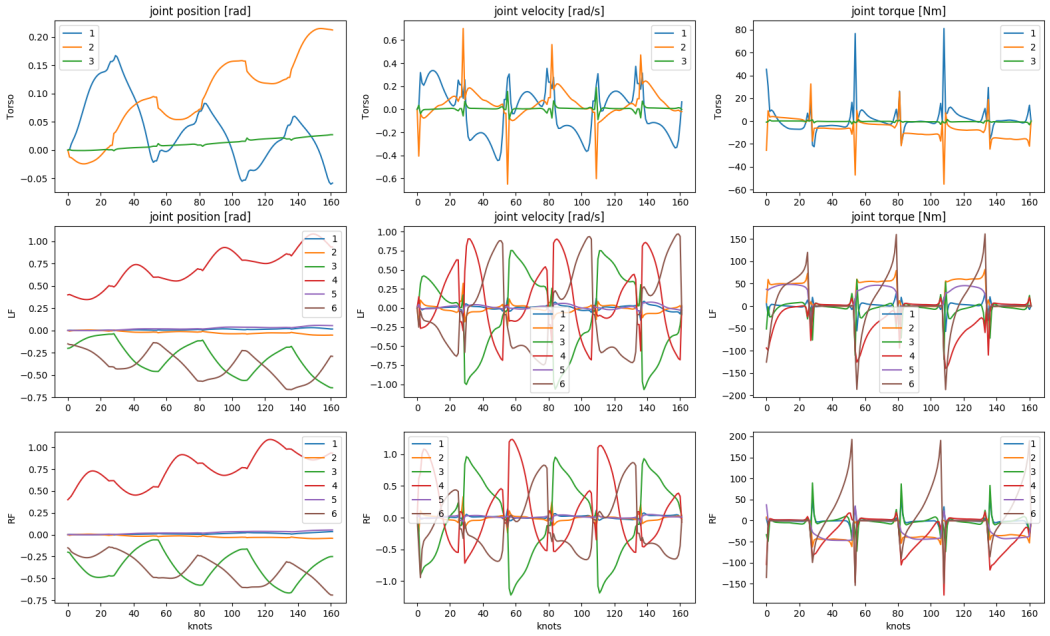
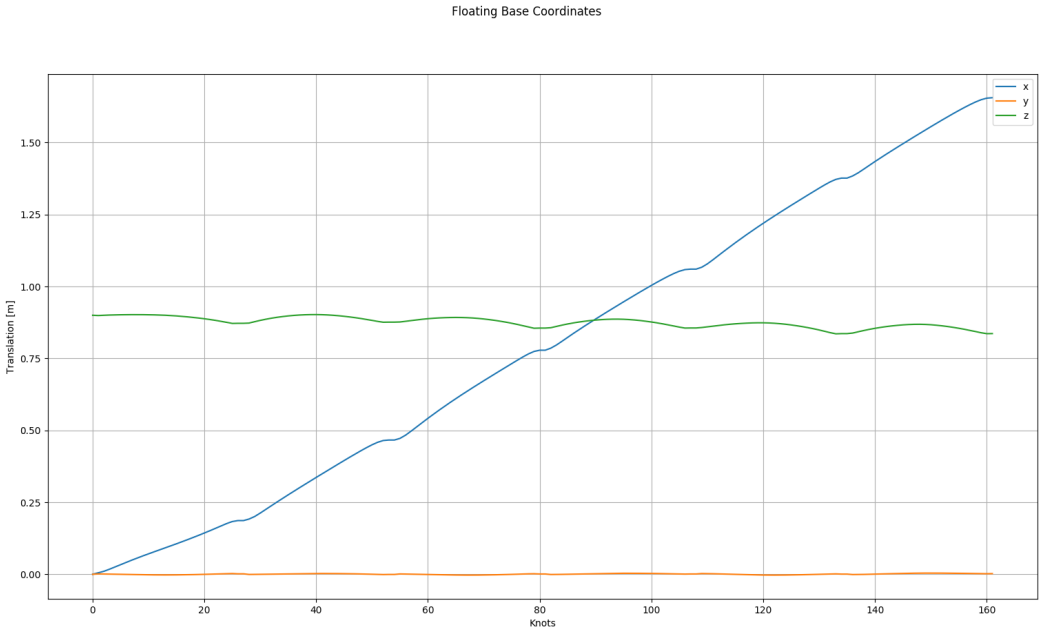


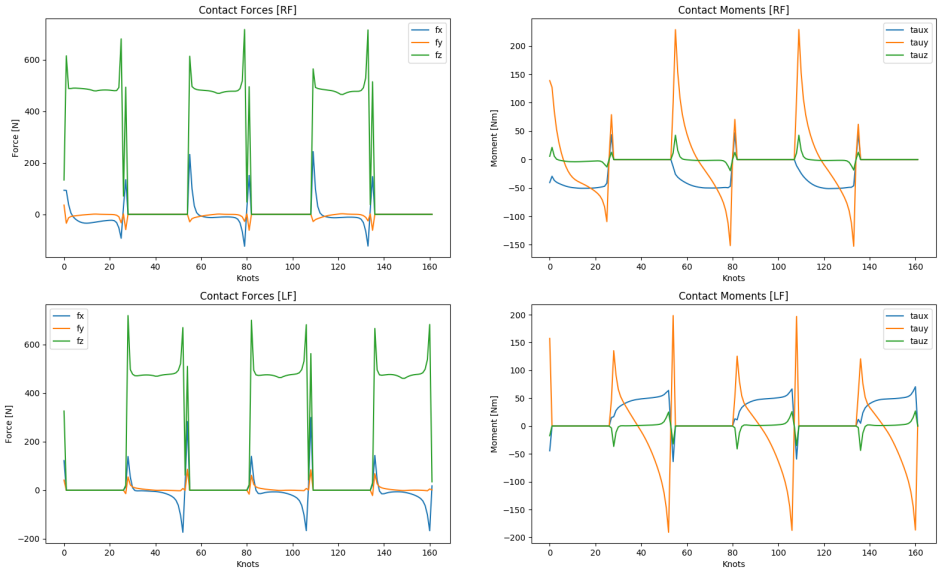
Figure 2.10: Visualization of the RH5 legs + torso using the gepetto-viewer.



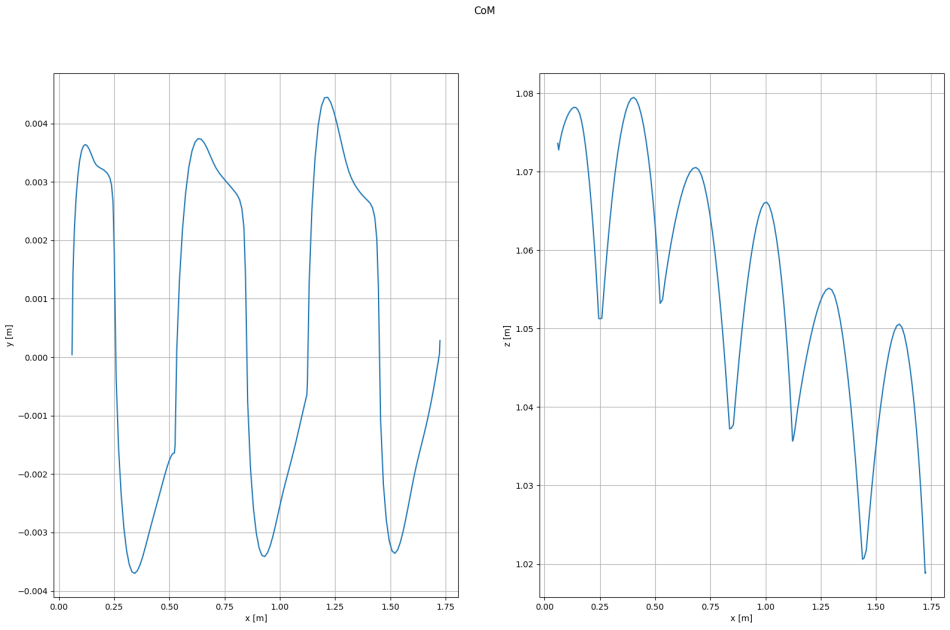
(a) Solution for states and torques.



(b) Floating base.

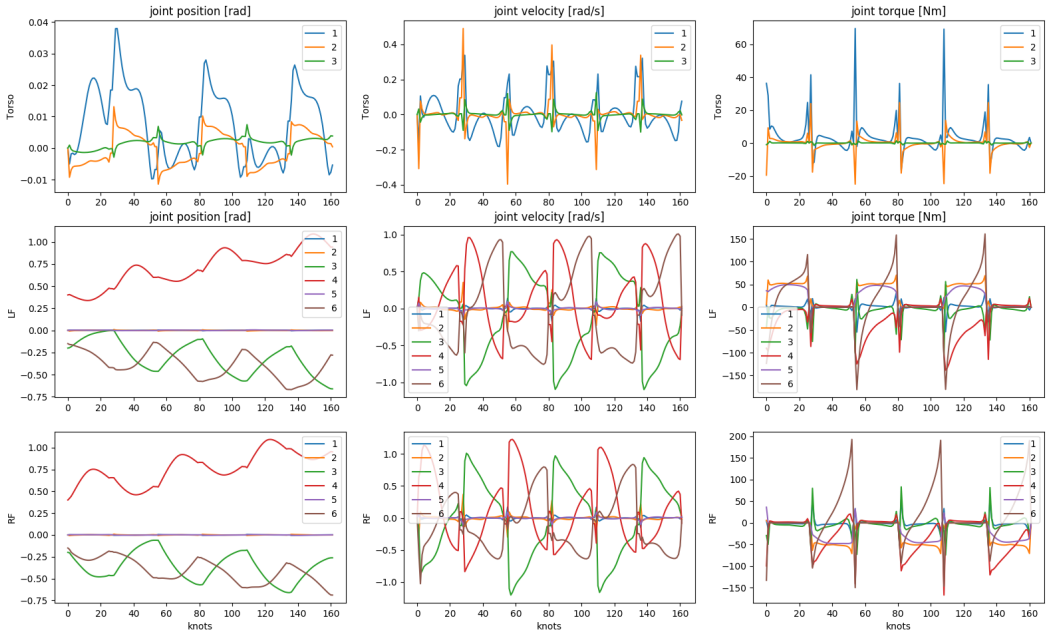


(c) Contact wrenches.

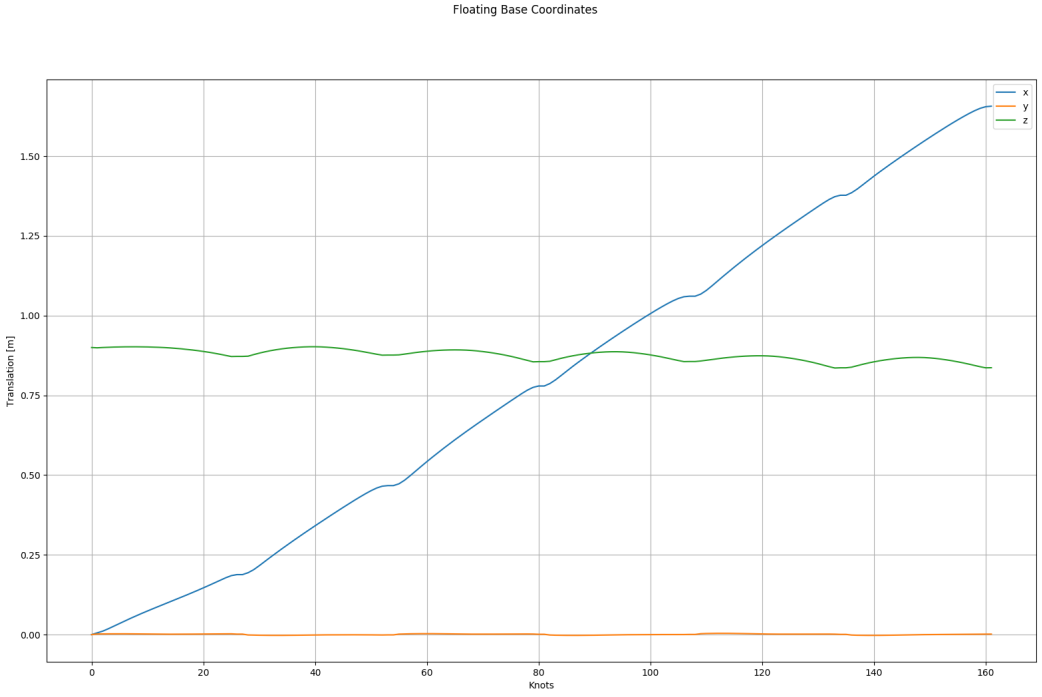


(d) Center of mass.

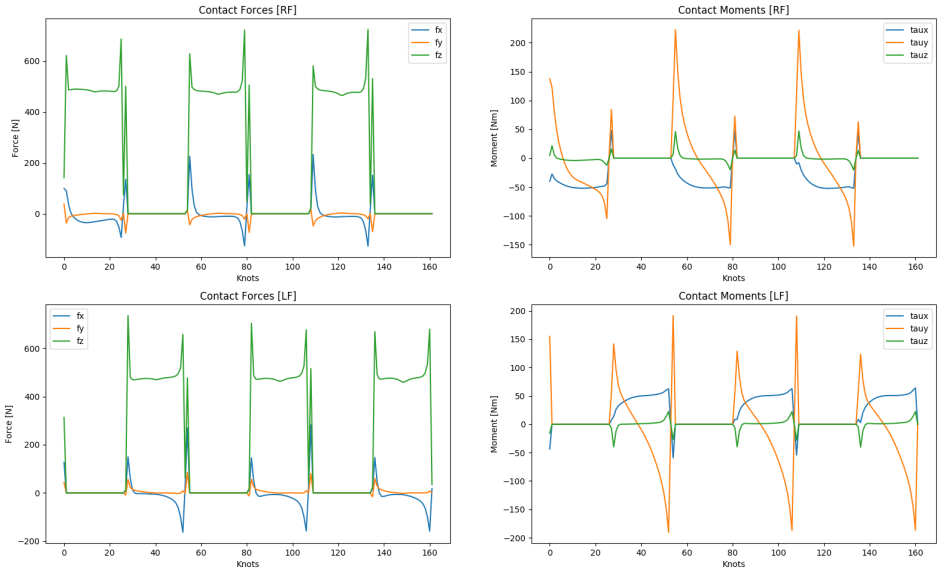
Figure 2.11: Results for the RH5 legs + torso for a full gait (6 steps).



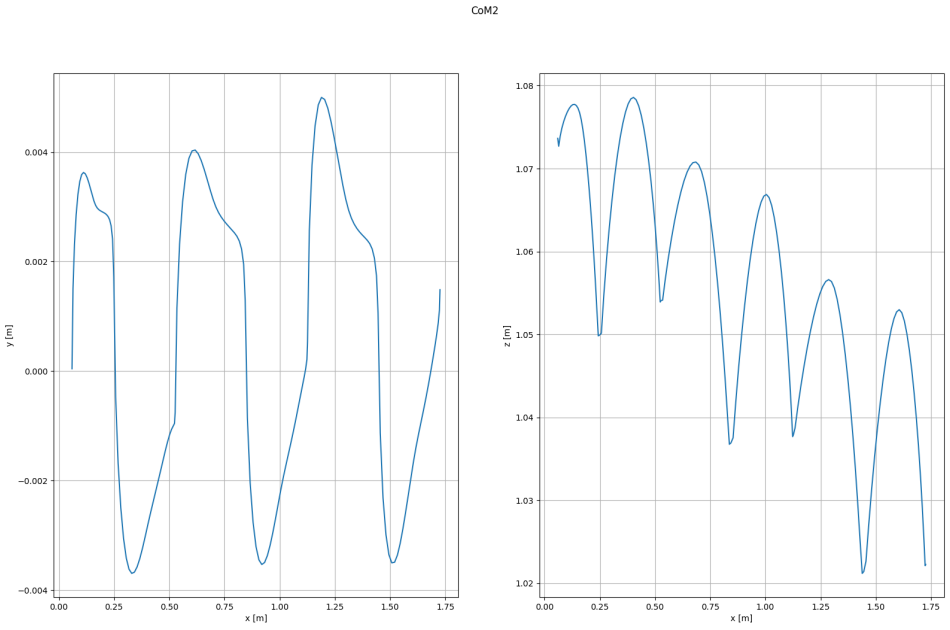
(a) Solution for states and torques.



(b) Floating base.



(c) Contact wrenches.



(d) Center of mass.

Figure 2.12: Results for the RH5 legs + a **stabilized torso** for a full gait (6 steps).

Chapter 3

DRAKE - MIT CSAIL

The second library we were interested in analyzing for its capabilites of generating bipedal walking patterns is Drake [6]. As it turned out, at the time of this work (spring 2020), Drake does not provide examples of legged locomotion anymore. Because of limited time, such an example has not been implemented.

Consequently, this chapter starts with a brief introduction to Drake and then summarizes the work with some simple examples, especially passive dynamic walkers. The focus of this work has been shifted to enhance the results gained with the Crocoddyl library , as described in chapter 2.

3.1 Introduction

3.1.1 Motivation

Drake is a **C++ toolbox for**

- Analyzing the dynamics of robots
- Building control systems for robots
- Heavy emphasis on optimization-based design/analysis

Drake aims to **simulate**

- Complex dynamics of robots (e.g. including friction, contact, aerodynamics etc.)
- Emphasis on exposing the structure in the governing equations (sparsity, analytical gradients, polynomial structure, uncertainty etc.)
- Making this information available for advanced planning, control, and analysis algorithms

Drake **provides**

- Python Interface
- Implementation of state-of-the-art algorithms
- Various examples

3.1.2 Core Modules

Drakes functionality is incorporated within several modules. This section gives a brief overview.

Modeling Dynamical Systems

Drake uses a Simulink-inspired description of dynamical systems.
Includes basic building blocks (adders, integrators, delays, etc), physics models of mechanical systems, and a growing list of sensors, actuators, controllers, planners, estimators.

Solving Mathematical Programs

Drake's MathematicalProgram class is used to solve the mathematical optimization problem in the following form

$$min_x f(x) \quad s.t. x \in S$$

Depending on the formulation of the objective function f, and the structure of the constraint set S, Drake can solve the following categories of optimization problems

- Linear programming
- Quadratic programming
- Nonlinear nonconvex programming
- Semidefinite programming
- Sum-of-squares programming
- Mixed-integer programming

Drake **automatically** calls suitable solvers for each category of optimization problem.

Multibody Kinematics and Dynamics

- Drake's **constraint system** helps solve computational dynamics problems with algebraic constraints
- Drake approximates real-world physical **contact** phenomena with a combination of geometric techniques and response models.

3.2 How-To

3.2.1 Install

Drake offers multiple ways of installation. This includes:

- Installation from Binaries
- Installation from Source (using bazel)
- Drake in Docker Containers

In this section some experiences for the installation from source and binaries are given. The choice mainly depends on if you want to use Drake via it's C++ or Python Interface.

Installation via Binaries

Propably the easiest way to access the Drake functionalities is via **pydrake** (python bindings). In this case it should be sufficient to install the binaries of Drake and and access them from a customized example directory, e.g. forked from the drake repo.

For Ubuntu 18.04 the installation boils down to

1. Download and extract the latest version to your /opt directory:

   ```
   curl -o drake.tar.gz https://drake-packages.csail.mit.edu/drake/ nightly/drake-latest-<platform>.tar
   rm -rf /opt/drake
   tar -xvzf drake.tar.gz -C /opt
   ```

2. Get the system dependencies:

   ```
   /opt/drake/share/drake/setup/install_prereqs
   ```

3. Add the python bindings to your PYTHONPATH environment variable:

   ```
   export PYTHONPATH=/opt/drake/lib/python3.6/site-packages:$ {PYTHONPATH}
   ```

4. Check if your installation was successful and that you can import pydrake:

   ```
   python3 -c 'import pydrake; print(pydrake.__file__)'
   ```

Installation from Source Using Bazel

If you instead prefer working directly on the **C++** examples or even want to contribute to the Drake library itself, an installation from source is recommended.

Detailed instructions can be found here: `https://drake.mit.edu/from_source.html`.

Please note that the installation of Drake from source can take you **several hours**.

Issues during build process

During installation from source I faced the following error message after a while:

```
Server terminated abruptly (error code: 14, error message: 'Socket closed'
```

Maybe, this crash was caused by to less RAM (8GB). A workaround adapted from `https://stackoverflow.com/a/34399184` was to

- Limit the number of parallel jobs and
- Limit the percentage of RAM Usage.

Finally, calling the build command with the following arguments was successfull:

```
CC=clang CXX=clang++ bazel build //... --ram_utilization_factor 30 --jobs=4
```

3.2.2 Run Python Examples

Basically there are two kind of python examples provided:

- Encapsulated in a jupyter notebook (.jpnb): Interactively run the blocks
- Pure python examples

The python examples can be found under the directory

`/drake/bindings/pydrake/examples/multibody`

Within this directory, to run a **2D example** (using planar-scenegraph-visualizer):

```
python3 run_planar_scenegraph_visualizer.py
```

If you instead want to run a **3D example** (using drake-visualizer): In a first console open the visualizer from the binaries

```
/opt/drake/bin$ ./drake-visualizer
```

Then, in a second console run (from your pydrake/examples/multibody directory again)

```
python3 cart_pole_passive_simulation.py
```

to see a 3D visualization of the simulated dynamics of a cart-pole model.

3.2.3 Additional Resources from MIT 6.832

Additional to the examples provided within the drake, there is existing other useful material. The MITs Underactuated Robotics class [5] offers a whole bunch of examples using pydrake. You can clone the repository with the course materials like this:

```
git clone https://github.com/RussTedrake/underactuated.git
sudo underactuated/scripts/setup/ubuntu/18.04/install_prereqs
export PYTHONPATH=‘pwd‘/underactuated:${PYTHONPATH}
```

3.3 Working with the Examples

This section contains some applications of the Drake library within several examples. Herein, the focus lies on understanding the implementation, but additionally some insights about the system dynamics will be shown.

Note that these examples are taken from Drake, but from the MIT Underactuated Robotics class. See section 3.2.3 for details on how to get the examples. The videos can all be found under `https://github.com/julesser/oc-frameworks/tree/master/OCFrameworks/Media/Drake`.

3.3.1 Cart-Pole

One classic example of a simple underactuated system is the famous cart-pole model. The system has 2DOF θ, x, where only the horizontal position x is acutated. For further details on the system visit `http://underactuated.csail.mit.edu/acrobot.html#cart_pole`.

Balancing around the upright using LQR

The Linear Quadratic Regulator (LQR) solves linear time-invariant system where the cost is described by a quadratic function. An exemplary task can be to stabilize the uprgith position of the pendulum from initial conditions.

Trajectory Optimization using Direct Collocation

From a methodology point of view, this example is interesting for our research and therefore will be explained in a bit more detail. The resulting dynamic up-swinging and and stabilization can best be seen in the videos, while the optimal force solution presented in Figure 3.1.

1. Load and build the dynamic model (*plant+context*)

```
plant = MultibodyPlant(time_step=0.0)
scene_graph = SceneGraph()
plant.RegisterAsSourceForSceneGraph(scene_graph)
file_name = FindResource("models/cartpole.urdf")
Parser(plant).AddModelFromFile(file_name)
plant.Finalize()
context = plant.CreateDefaultContext()
```

2. Setup the optimization problem with discrete, uniform time steps (i.e. knots)

```
dircol = DirectCollocation(
    plant,
    context,
    num_time_samples=21,
    minimum_timestep=0.1,
    maximum_timestep=0.4,
    input_port_index=plant.get_actuation_input_port().get_index())
dircol.AddEqualTimeIntervalsConstraints()
```

3. Specify the initial and terminal state via constraints

```
initial_state = (0., 0., 0., 0.)
dircol.AddLinearConstraint(dircol.initial_state() == initial_state)
final_state = (0., math.pi, 0., 0.)
dircol.AddLinearConstraint(dircol.final_state() == final_state)
```

4. For each running knot, penalize actuator effort. For the terminal state, penalize the total time consumped.

```
R = 10  # Cost on input "effort".
u = dircol.input()
dircol.AddRunningCost(R * u[0]**2)
dircol.AddFinalCost(dircol.time())
```

5. Generate a piecewise-polynomial function and use it as initial guess

```
initial_x_trajectory = PiecewisePolynomial.FirstOrderHold(
    [0., 4.], np.column_stack((initial_state, final_state)))
dircol.SetInitialTrajectory(PiecewisePolynomial(), initial_x_trajectory)
```

6. Solve the trajectory optimizaton problem

```
result = Solve(dircol)
```

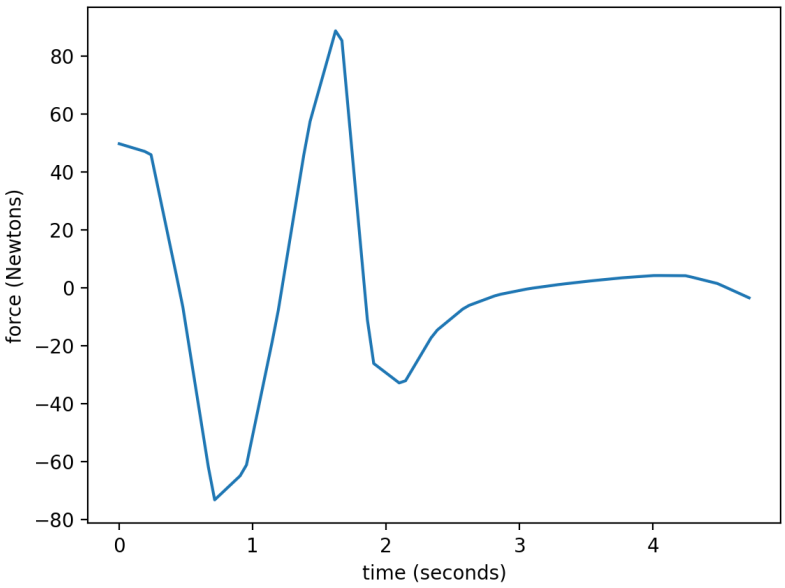


Figure 3.1: Resulting optimal horizontal force from trajectory optimization of the up-swining and stabilization for the cart-pole.

3.3.2 Passive Dynamic Walkers

Although there currently are no highly-articulated robots found in the Drake examples, it does provide some classic examples of simplified locomotion models.

Rimless Wheel

Simulating the simple dynamics of the rimless wheel model rolling down a declining slope, an continous limit cycle emerges (see Figure 3.2).

Compass Gait

The compass gait is a simple bipedal walking model containing two links and three pointmasses. For further details on the system visit `http://underactuated.csail.mit.edu/simple_legs.html#section2`.

Simulating the system dynamics for a declining slope results in periodic motions (see Figure 3.3).

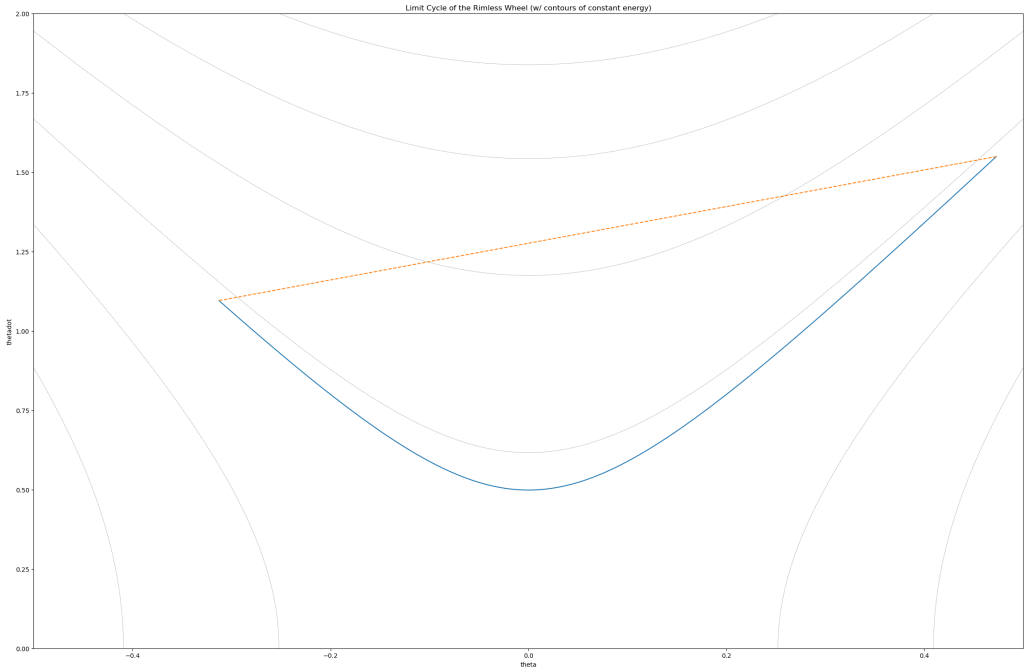


Figure 3.2: Results for an energy-optimal limit cycle for the rimless wheel using direct collocation trajectory optimization.

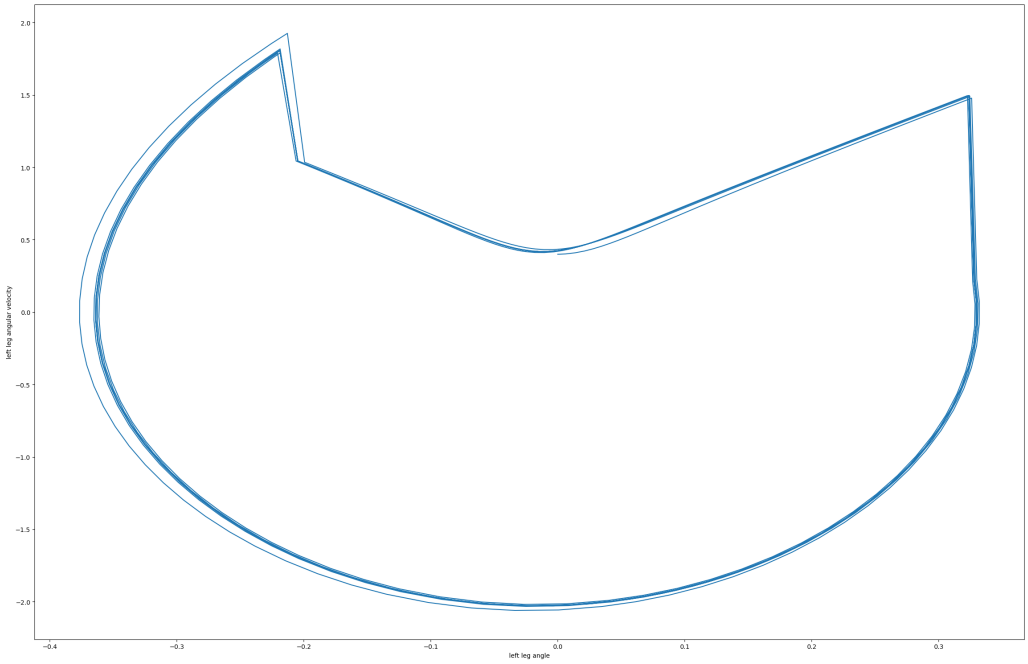


Figure 3.3: Resulting phase plot of the compass gait reveals periodic orbits.

Chapter 4

COMPARISON

This chapter offers a brief comparison of both frameworks in terms of generality, usability and bipedal walking capabilities. This comparison is from a very practical point of view and may be biased since the author investigated Crocoddyl more strongly than Drake. Furthermore, this work is context-specific to generating optimal trajectories for a simple bipedal walking pattern.

4.1 Generality vs Specificity

Both frameworks differ considerably in the variety of problems that they are trying to takle and the amount of implemented functionalities.

For solving optimal control problems, each of the frameworks follows its own path.

Overall Scope

Drake to a certain extend is a 'All in One' library since it contains modules for solving multibody dynamics, mathematical programs as well as a whole bunch of tools for modeling and analyzing systems to gain a deeper understanding of the underlying dynamics. On the opposite, Crocoddyl is designed for a very specific range of problems, namely to solve optimization problems for robots with multiple point-contacts. It depends on the Gepetto-Viewer for visualizing and the Pinnoccio library for kinematics formulation and dynamics calculations.

Optimal Control Solvers

Also the way the optimization problems are solved differ quite strong. Whereas Drake offers a wide range of available solvers, Crocoddyls solvers list in comparison is rather small and solely based on versions of the Differential Dynamic Programming (DDP) algorithm.

Formulation of Trajectory Optimization

Finally, the way of discretising the continous time-problem for the trajectory optimization methods differs. Crocoddyl on the one hand focuses on shooting methods, while Drake on the other hand utilizes direct transcription/collocation.

4.2 Usability

In general, the author considers Drake as well as Crocoddyl to be quite user-friendly, but Drake stands out with its excellent code documentation.

Python Bindings

The main reason for this assessment propably is the provision of python bindings within both libraries, which allows for rapid prototyping of examples.

Installation

To this end, the installation procedure could be simplified a lot since it was sufficient to install the binaries under /opt and use a cloned version of the examples directory. Following this approach, the Drake binaries can be installed as usual, while Crocoddyl utilizes the *robotpkg* manager.

Code Documentation

Drake offers a beautiful descripiton of it's C++ API that also holds for most of the functionalities within pydrake. Crocoddyl on the other hand provides only a very sparse documentation, e.g. on solver functionalities or implemented contact models.

Getting Started

What Crocoddyl lacks in code documentation is definitely made up for by an excellent collection of beginner-friendly tutorials! Drake also comes along with some beginner tutorials, but in the authors opinion the tend to be more abstract.

Example Functionalities

Also in the example domain Croccodyl can shine. Whereas Drake solely offers examples related containing simple systems, Crocoddyl provides the user with readily accessible examples of complex legged robots (e.g. quadrupeds, humanoids) performing challenging tasks (e.g. walking, jumping).

4.3 Bipedal Walking Capabilities

The example of bipedal walking provided by the Crocoddyl library served as a comfortable starting point for integrating and testing the RH5 robot (see section 2.6 for the results).

On the contrary, Drake currently does not offer this luxury. The goal of this section is to estimate the effort necessary to produce equivalent bipedal walking results within the Drake library that already have been achieved within Crocoddyl.

4.3.1 RH5 Example in Drake: Key Components

In the following, the key steps towards a functional bipedal walking example shall be determined and cross-checked with the functionalities Drake provides.

High-Level Formulation:

1. Build and visualize the robot via a URDF-based description
2. Implement basic locomotion logic (Double support, foot step)
3. Generate reference trajectory, i.e. initial guess
4. Formulate an optimization problem with discrete time-steps
    - Constrain the problem (e.g. bounded torques)
    - Time-step specific costs
    - Time-step specific point-contacts
    - Model impulse dynamics
5. Solve the problem with sophisticated method

Step 1-3: Basic Implementation Work

These steps do not have special requirements since it is soley python coding for setting up a general frame of the example.

Step 4: Formulate an Optimization Problem

`https://github.com/DAIRLab/dairlib/blob/master/examples/Cassie/run_dircon_walking.cc` can give a first impression on the dimensions of formulating a full optimization problem for a complex system within Drake.
Drake offers multiple ways of formulating an optimization problem; the **MultipleShooting Class** seems to be a good starting point. The following functions seem to be of relevance:

- AddRunningCost()
- AddFinalCost()
- AddConstraintToAllKnotPoints()
- SetInitialTrajectory()

Step 5: Sophisticated Solvers

Drake automatically selects the most suitable solver. No effort required on this matter.

4.3.2 RH5 Example in Drake: Estimated Effort

From the above inspection it arises the general impression, that Drake already offers most of the required functionalities required for creating a simple walk with RH5.
The estimated effort for producing a **simple walking example** is about **40-60 hours**, depending on the desired performance. The following key tasks could be determined:

- Implement the frame of the example (Load and visualize RH5, basic locomotion logic, reference trajectory)
- Build high-level interface for conveniently using costs and constraints in the optimization context (frame placement, CoM trajectory etc.)
- Appropriate multi-contact and impulse modeling

4.4 Summary

The purpose of this chapter was to compare both frameworks in the context of optimal control for bipedal walking. Table 4.1 offers a densed overview of this comparison.

Table 4.1: High-level comparison of the Drake and Crocoddyl Framework in the context of optimal control of bipedal walking tasks.

Metric	**Crocoddyl**	**Drake**
Generality	Optimal Control library	'All in One' library
OC Solvers	DDP-based	Various
Python Bindings	Y	Y
Code Documentation	Sparse	Extensive
Tutorials	Many	Some
Basic Examples	Y	Y
Bipedal Walking Examples	Y	N

Chapter 5

CONCLUSION

Within this two-month student project it was possible to integrate the RH5 robot in the **Crocoddyl** framework and generate first successfull trajectories for a simple walk (see chapter 2.6). These preliminary results offer much space for improvement and future research opportunities.

The investigation of the **Drake** library was limited to simple passive dynamic walking models (see 3.3). An estimation of producing comparable bipedal walking results like in Crocoddyl within the Drake framework, revealed a workload of approximitive 40-60 hours (see section 4.3 for details).

In the end, the author considers **both framworks suitable for further research** and application to the RH5 robot. Whereas Drake offers a more general framework for optimization-based analysis (various solvers, planner, controller etc.) and does not provide targeted examples for bipedal walking of highly-articulated robots, Crocoddyl focuses exactly on these kind of problems, offers great example functionalities, but also limits the available solvers to the special branch of DDP-based solvers.

Appendix A

Background: Crocoddyl Workflow

This chapter contains details on the workflow in Crocoddyl and presents some of the underlying math. This information can be found in [1] within examples/notebooks/introduction_to_crocoddyl.ipynb.

A.1 Define an Action Model (Dynamics+Costs)

In crocoddyl, an action model combines dynamics and cost models. Each node, in our optimal control problem, is described through an action model. In order to describe a problem, we need to provide ways of computing the dynamics, the cost functions and their derivatives. All these are described inside the action model.

To understand the mathematical aspects behind an action model, let's first get a locally linearize version of our optimal control problem as:

$$\mathbf{X}^*(\mathbf{x}_0), \mathbf{U}^*(\mathbf{x}_0) = \arg\max_{\mathbf{X},\mathbf{U}} = cost_T(\delta\mathbf{x}_N) + \sum_{k=1}^{N} cost_t(\delta\mathbf{x}_k, \delta\mathbf{u}_k)$$

subject to

$$dynamics(\delta\mathbf{x}_{k+1}, \delta\mathbf{x}_k, \delta\mathbf{u}_k) = \mathbf{0},$$

where

$$cost_T(\delta\mathbf{x}) = \frac{1}{2}\begin{bmatrix}1\\ \delta\mathbf{x}\end{bmatrix}^\top \begin{bmatrix}0 & \mathbf{l_x}^\top\\ \mathbf{l_x} & \mathbf{l_{xx}}\end{bmatrix}\begin{bmatrix}1\\ \delta\mathbf{x}\end{bmatrix}$$

$$cost_t(\delta\mathbf{x}, \delta\mathbf{u}) = \frac{1}{2}\begin{bmatrix}1\\ \delta\mathbf{x}\\ \delta\mathbf{u}\end{bmatrix}^\top \begin{bmatrix}0 & \mathbf{l_x}^\top & \mathbf{l_u}^\top\\ \mathbf{l_x} & \mathbf{l_{xx}} & \mathbf{l_{ux}}^\top\\ \mathbf{l_u} & \mathbf{l_{ux}} & \mathbf{l_{uu}}\end{bmatrix}\begin{bmatrix}1\\ \delta\mathbf{x}\\ \delta\mathbf{u}\end{bmatrix}$$

$$dynamics(\delta\mathbf{x}_{k+1}, \delta\mathbf{x}_k, \delta\mathbf{u}_k) = \delta\mathbf{x}_{k+1} - (\mathbf{f_x}\delta\mathbf{x}_k + \mathbf{f_u}\delta\mathbf{u}_k)$$

where an action model defines a **time interval** of this problem:

- $actions = dynamics + cost$

Important notes:

- An action model describes the dynamics and cost functions for a node in our optimal control problem.

- Action models lie in the discrete time space.
- For debugging and prototyping, we have also implemented numerical differentiation (NumDiff) abstractions.

These computations depend only on the definition of the dynamics equation and cost functions. However to asses efficiency, crocoddyl uses **analytical derivatives** computed from Pinocchio.

A.2 Differential Action Model

Optimal control solvers require the time-discrete model of the cost and the dynamics. However, it's often convenient to implement them in continuous time (e.g. to combine with abstract integration rules). In crocoddyl, this continuous-time action models are called "Differential Action Model (DAM)". And together with predefined "Integrated Action Models (IAM)", it possible to retrieve the time-discrete action model.
At the moment, we have:

- a simpletic Euler and
- a Runge-Kutte 4 integration rules.

An optimal control problem can be written from a set of DAMs as:

$$\mathbf{X}^*(\mathbf{x}_0), \mathbf{U}^*(\mathbf{x}_0) = \arg\max_{\mathbf{X},\mathbf{U}} = cost_T(\delta\mathbf{x}_N) + \sum_{k=1}^{N} \int_{t_k}^{t_k+\Delta t} cost_t(\delta\mathbf{x}_k, \delta\mathbf{u}_k)dt$$

subject to

$$dynamics(\delta\mathbf{x}_{k+1}, \delta\mathbf{x}_k, \delta\mathbf{u}_k) = \mathbf{0},$$

where

$$cost_T(\delta\mathbf{x}) = \frac{1}{2}\begin{bmatrix}1\\ \delta\mathbf{x}\end{bmatrix}^\top \begin{bmatrix}0 & \mathbf{l_x}^\top\\ \mathbf{l_x} & \mathbf{l_{xx}}\end{bmatrix}\begin{bmatrix}1\\ \delta\mathbf{x}\end{bmatrix}$$

$$cost_t(\delta\mathbf{x}, \delta\mathbf{u}) = \frac{1}{2}\begin{bmatrix}1\\ \delta\mathbf{x}\\ \delta\mathbf{u}\end{bmatrix}^\top \begin{bmatrix}0 & \mathbf{l_x}^\top & \mathbf{l_u}^\top\\ \mathbf{l_x} & \mathbf{l_{xx}} & \mathbf{l_{ux}}^\top\\ \mathbf{l_u} & \mathbf{l_{ux}} & \mathbf{l_{uu}}\end{bmatrix}\begin{bmatrix}1\\ \delta\mathbf{x}\\ \delta\mathbf{u}\end{bmatrix}$$

$$dynamics(\delta\dot{\mathbf{x}}, \delta\mathbf{x}, \delta\mathbf{u}) = \delta\dot{\mathbf{x}} - (\mathbf{f_x}\delta\mathbf{x} + \mathbf{f_u}\delta\mathbf{u})$$

Optimal control solvers often need to compute a quadratic approximation of the action model (as previously described); this provides a search direction (computeDirection). Then it's needed to try the step along this direction (tryStep).
Typically calc and calcDiff do the precomputations that are required before computeDirection and tryStep respectively (inside the solver). These functions update the information of:

- **calc**: update the next state and its cost value

$$\delta\dot{\mathbf{x}}_{k+1} = \mathbf{f}(\delta\mathbf{x}_k, \mathbf{u}_k)$$

- **calcDiff**: update the derivatives of the dynamics and cost (quadratic approximation)

$$\mathbf{f_x}, \mathbf{f_u} \quad (dynamics)$$

$$\mathbf{l_x}, \mathbf{l_u}, \mathbf{l_{xx}}, \mathbf{l_{ux}}, \mathbf{l_{uu}} \quad (cost)$$

A.3 Integrated Action Model

General speaking, the system's state can lie in a manifold M where the state rate of change lies in its tangent space $T_{\mathbf{x}}M$. There are few **operators that needs to be defined** for different rutines inside our solvers:

- $\mathbf{x}_{k+1} = integrate(\mathbf{x}_k, \delta\mathbf{x}_k) = \mathbf{x}_k \oplus \delta\mathbf{x}_k$
- $\delta\mathbf{x}_k = difference(\mathbf{x}_{k+1}, \mathbf{x}_k) = \mathbf{x}_{k+1} \ominus \mathbf{x}_k$

where $\mathbf{x} \in M$ and $\delta\mathbf{x} \in T_{\mathbf{x}}M$.
And we also need to defined the **Jacobians** of these operators with respect to the first and second arguments:

- $\frac{\partial \mathbf{x} \oplus \delta\mathbf{x}}{\partial \mathbf{x}}, \frac{\partial \mathbf{x} \oplus \delta\mathbf{x}}{\partial \delta\mathbf{x}} = Jintegrante(\mathbf{x}, \delta\mathbf{x})$
- $\frac{\partial \mathbf{x}_2 \ominus \mathbf{x}_2}{\partial \mathbf{x}_1}, \frac{\partial \mathbf{x}_2 \ominus \mathbf{x}_1}{\partial \mathbf{x}_1} = Jdifference(\mathbf{x}_2, \mathbf{x}_1)$

For instance, a state that lies in the Euclidean space will have the typical operators:

- $integrate(\mathbf{x}, \delta\mathbf{x}) = \mathbf{x} + \delta\mathbf{x}$
- $difference(\mathbf{x}_2, \mathbf{x}_1) = \mathbf{x}_2 - \mathbf{x}_1$
- $Jintegrate(\cdot, \cdot) = Jdifference(\cdot, \cdot) = \mathbf{I}$

These defines are encapsulated inside the State class. **For Pinocchio models, we have implemented the StatePinocchio class which can be used for any robot model.**

A.4 Solving the Optimal Control Problem

Our optimal control solver interacts with a defined ShootingProblem. A **shooting problem** represents a **stack of action models** in which an action model defines a specific node along the OC problem.
First we need to create an action model from DifferentialFwdDynamics. We use it for building terminal and running action models. In this example, we employ an simpletic Euler integration rule.
Next we define the set of cost functions for this problem. One could formulate

- Running costs (related to individual states)
- Terminal costs (related to the final state)

in order to penalize, for example, the state error, control error, or end-effector pose error.
Onces we have defined our shooting problem, we create a DDP solver object and pass some callback functions for analysing its performance.

A.5 Application to Bipedal Walking

In crocoddyl, we can describe the multi-contact dynamics through holonomic constraints for the support legs. From the Gauss principle, we have derived the model as:

$$\begin{bmatrix} \mathbf{M} & \mathbf{J}_c^\top \\ \mathbf{J}_c & \mathbf{0} \end{bmatrix} \begin{bmatrix} \dot{\mathbf{v}} \\ -\boldsymbol{\lambda} \end{bmatrix} = \begin{bmatrix} \boldsymbol{\tau} - \mathbf{h} \\ -\dot{\mathbf{J}}_c\mathbf{v} \end{bmatrix}$$

.

This DAM is defined in "DifferentialActionModelFloatingInContact" class.

Given a predefined contact sequence and timings, we build per each phase a specific multi-contact dynamics. Indeed we need to describe **multi-phase optimal control problem**. One can formulate the multi-contact optimal control problem (MCOP) as follows:

$$\mathbf{X}^*, \mathbf{U}^* = \begin{Bmatrix} \mathbf{x}_0^*, \cdots, \mathbf{x}_N^* \\ \mathbf{u}_0^*, \cdots, \mathbf{u}_N^* \end{Bmatrix} = \arg\min_{\mathbf{X},\mathbf{U}} \sum_{p=0}^{P} \sum_{k=1}^{N(p)} \int_{t_k}^{t_k+\Delta t} l_p(\mathbf{x}, \mathbf{u}) dt$$

subject to

$$\dot{\mathbf{x}} = \mathbf{f}_p(\mathbf{x}, \mathbf{u}), \text{for } t \in [\tau_p, \tau_{p+1}]$$

$$\mathbf{g}(\mathbf{v}^{p+1}, \mathbf{v}^p) = \mathbf{0}$$

$$\mathbf{x} \in \mathcal{X}_p, \mathbf{u} \in \mathcal{U}_p, \boldsymbol{\lambda} \in \mathcal{K}_p.$$

where $\mathbf{g}(\cdot, \cdot, \cdot)$ describes the contact dynamics, and they represents terminal constraints in each walking phase. In this example we use the following **impact model**:

$$\mathbf{M}(\mathbf{v}_{next} - \mathbf{v}) = \mathbf{J}_{impulse}^T$$

$$\mathbf{J}_{impulse}\mathbf{v}_{next} = \mathbf{0}$$

$$\mathbf{J}_c\mathbf{v}_{next} = \mathbf{J}_c\mathbf{v}$$

Bibliography

[1] Rohan Budhiraja Carlos Mastalli, Nicolas Mansard, et al. Crocoddyl: a fast and flexible optimal control library for robot control under contact sequence. https://gepgitlab.laas.fr/loco-3d/crocoddyl/wikis/home, 2019.

[2] Julian Eßer. Optimal control frameworks for bipedal walking. https://github.com/julesser/oc-frameworkse, 2020.

[3] Olivier Stasse, Thomas Flayols, Rohan Budhiraja, Kevin Giraud-Esclasse, Justin Carpentier, Joseph Mirabel, Andrea Del Prete, Philippe Souères, Nicolas Mansard, Florent Lamiraux, et al. Talos: A new humanoid research platform targeted for industrial applications. In *2017 IEEE-RAS 17th International Conference on Humanoid Robotics (Humanoids)*, pages 689–695. IEEE, 2017.

[4] Yuval Tassa, Nicolas Mansard, and Emo Todorov. Control-limited differential dynamic programming. In *2014 IEEE International Conference on Robotics and Automation (ICRA)*, pages 1168–1175. IEEE, 2014.

[5] Russ Tedrake. Mitx6.832x: Official lecture website. http://underactuated.csail.mit.edu/Spring2019/index.htmltextbook/ass 2020.

[6] Russ Tedrake and the Drake Development Team. Drake: Model-based design and verification for robotics, 2019.