HDSRL QP-Based VC Controller Documentation

Randy Fawcett

Last Updated: June 9, 2023

Contents

1 Overview

The low-level controller discussed here has been used in many works produced by the HDSRL [1, 2, 3, 4, 5, 6, 7, 8]. In particular, the controller discussed here considers the use of partial feedback linearization in order to create a linear mapping between the torques and the movement of both the Cartesian position of the swing feet and the center of mass (linear and angular position). This is then used in tandem with a Quadratic Program (QP) to avoid numerical issues and singularities in the decoupling matrix. Furthermore, the QP formulation allows one to consider the friction cone condition and the torque limits.

The code itself is fairly well documented. Therefore, the purpose of this document is to provide an overview of the structure and organization of the code. It also provides some information and suggestions on use. However, the code provided with the controller will serve as a much better example for use cases. See also the brief documentation provided on the GitHub page.

2 Code Organization

The low-level code provided considers a *moderator pattern*. For a good overview of design patterns in C++, consider visiting this website. In essence, the code is set up such that there are a series of classes that all communicate through a "moderator", and the user only interacts with said moderator. In this case, the moderator is denoted by "LocomotionWrapper" (see Figure 1). All of the classes communicate through locomotion wrapper.

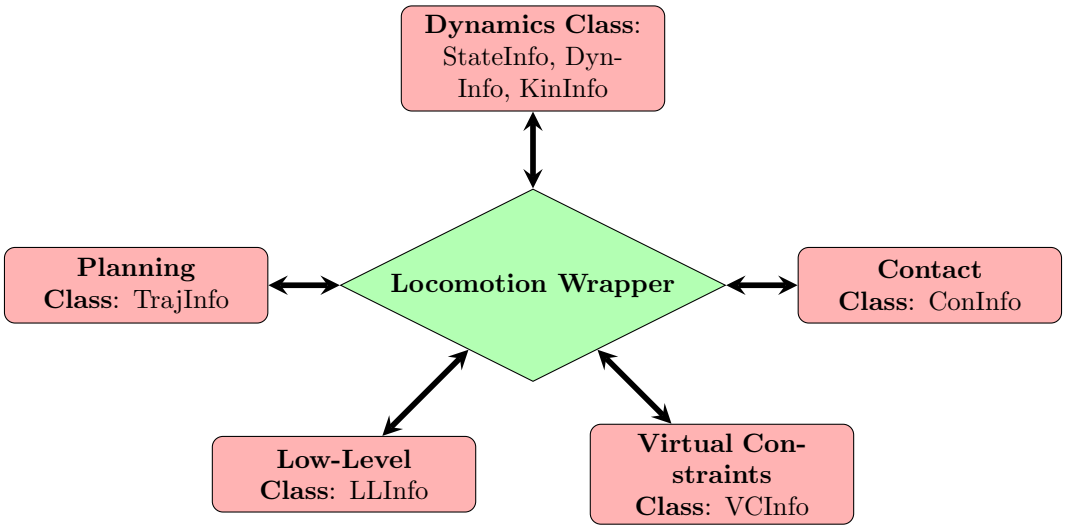

Figure 1: Basic overview of the code structure. All of the classes communicate through locomotion wrapper. Each class "owns" a data structure that can be shared with other classes through the moderator. For example, the planning class has a structure called TrajectoryInfo, which contains all of the information from the trajectory planner. Locomotion wrapper has a constant pointer to each of these structures. It is a constant pointer to encourage encapsulation. I.e., other classes should not be able to change the information in the structures. The data logging works similarly but has been left out here.

Each class "owns" a data structure that can be shared with other classes through the moderator. For example, the planning class has a structure called TrajectoryInfo, which contains all of the information from the trajectory planner. Locomotion wrapper has a constant pointer to each of these structures. It is a constant pointer to encourage encapsulation. I.e., other classes should not be able to change the information in the structures. For example, it makes sense that no class should be able to change the mass-inertia matrix except for the dynamics class.

Note: All of the structures can be found in the `global_loco_structs.hpp` header.

2.1 LocoWrapper

As mentioned previously, this class acts as a moderator between all other classes and is the interface through which users interact with the code. When actually using the controller, the user only interacts with locomotion wrapper by using setters and getters defined within the locomotion wrapper class. I.e., assume you have an instance of the locomotion wrapper called `loco_obj`. In order to set get the desired torque, one would use something analogous to the following:

```
double tau[18] = {0.0};
loco_obj->calcTau( <inputs to function...> );
tau = loco_obj->getTorque();
```

Therefore, in your main code, you do not update the dynamics, plan the trajectory, update the virtual constraints, run the optimization problem, etc. The locomotion wrapper does all of these things, and you simply interact with it. That is, at least, the intention. However, the locomotion wrapper may not be

suitable for all purposes. The intention is that each individual would write their own wrapper as needed since all of the blocks are modular.

This modularity also allows for easier changes in the future. For example, the dynamics class currently utilizes pre-made closed-form expressions for the dynamics (generated using FROST [9]) instead of using a numerical algorithm [10]. If you wanted to change the manner in which the dynamics were calculated, you would simply need to change the dynamics class. As long as the inputs and outputs of the class are the same, then they can be interchanged very easily.

The primary function in this class is `calcTau`, which updates the state, trajectory, contact, and virtual constraints, runs the low-level controller, and exports data. However, the user may also set the desired GRF, COM trajectory, command velocity, etc. Look through the header file for more information on the getters and setters currently available.

It is also worth noting that this class is a friend of the parameters class. In essence, this means that the variables of the parameters class are also accessible directly from the locomotion wrapper class. The parameters class is used to load parameters from a text file that determines how the low-level controller works. Look through `params/LL_joystick.txt` and `params/Walking_joystick.txt` for more information. These text files are passed into the constructor of the locomotion wrapper at run time, and therefore they are also passed to the constructor of the parameters class, such that parameters can be changed without needing to recompile the code.

DO NOT integrate your trajectory planner (for example, MPC) directly into this class. When moving to hardware, the MPC needs to run on a thread separate from the low-level controller. Therefore, you **CANNOT** have the high-level code mixed into the low-level code. This will cause issues when moving to experiments.

2.2 Dynamics Class

The dynamics class (RobotModel.cpp) contains all of the functions necessary for updating the state, the dynamics, and the kinematics. Therefore, this class owns the data structures `StateInfo`, `DynamicsInfo`, and `KinematicsInfo`. See `global_loco_structs.hpp` for more detailed information on the structures.

2.3 Motion Planner Class

This class owns the structure `TrajectoryInfo`, which will be denoted by `traj` in this section.

This class handles trajectory planning. Even when used in tandem with a high-level trajectory planner, this class is still necessary because it handles footstep planning. This class could potentially change considerably depending on the project. What I have done in the past is to use this class to create a desired trajectory even when using a higher-level planner (i.e., this class plans for the desired trajectory over an entire horizon, and for a 1 ms time step). The desired reduced-order trajectory can be stored in `traj->redDes`. The desired trajectory can then be passed to the high-level planner. However, the virtual constraints are based on a variable `comDes` which is structured as follows: `comDes` $= (x, y, z, \dot{x}, \dot{y}, \dot{z}, roll, pitch, \omega_x, \omega_y, \omega_z)$. So, after the high-level controller runs, you must set `comDes` by using `loco_obj->setDesired(...)`. This is a bit of a round-about way of doing things, but it works well for the time being.

2.3.1 Brief Note on the traj Structure

Given the manner in which the data is collected, it is important to note that `traj->desVel` is meant only to contain the user-specified desired velocity. Using this desired velocity, you may pass it directly

to the low-level controller by using it to pack `traj->comDes`, or pass it to a higher-level planner using `traj->redDes` (reduced-order desired trajectory). The result of the higher-level planner would then be used to pack `traj->comDes` for tracking. Although the MPC trajectory is not saved explicitly when collecting data, the command can be easily reconstructed by noting that the virtual constraints are collected (i.e., you have access to h^{des} and its derivatives).

2.4 Contact Class

The contact class is largely unused but remains here in case a sophisticated contact detection algorithm is ever implemented. For now, it essentially just keeps track of the current contacting feet and the number of contacts.

2.5 Virtual Constraints Class

The virtual constraints class calculates the error between the controlled variables (the COM position and orientation, and the position of the swing feet) and the desired evolution of said variables. All of this information is stored in the `VCInfo` structure, which belongs to the virtual constraints class. The desired evolution of the COM is defined either heuristically through the motion planner discussed previously or using an optimal trajectory from a planner. The desired swing foot location is defined using Raiberts Heuristic [11]. Using the desired swing foot position, we then create a time-varying bezièr polynomial [12] moving the foot from the previous location to the new location and track that using the virtual constraints.

2.6 Low-level Class

Finally, the low-level class actually creates the matrices for the optimization problem and solves for the torques required to track the virtual constraints defined above. See the code and [1, 2, 3] for more details regarding how the matrices are calculated. The torques that are calculated are stored in the `LLInfo` structure. In addition, using the torques and the desired swing leg trajectories, the low-level controller also provides estimated joint positions and velocities that are used on the hardware to act as a damping term. This is done using the dynamics and inverse kinematics.

3 Other Random Things

- There are many transformations and utilities that can be found in the `util_include` folder

References

[1] K. A. Hamed, J. Kim, and A. Pandala, "Quadrupedal locomotion via event-based predictive control and qp-based virtual constraints," *IEEE Robotics and Automation Letters*, vol. 5, no. 3, pp. 4463–4470, 2020.

[2] R. T. Fawcett, A. Pandala, A. D. Ames, and K. Akbari Hamed, "Robust stabilization of periodic gaits for quadrupedal locomotion via QP-based virtual constraint controllers," *IEEE Control Systems Letters*, vol. 6, pp. 1736–1741, 2021.

[3] J. Kim and K. Akbari Hamed, "Cooperative locomotion via supervisory predictive control and distributed nonlinear controllers," *ASME Journal of Dynamic Systems, Measurement, and Control*, vol. 144, no. 3, Dec 2021.

[4] R. T. Fawcett, L. Amanzadeh, J. Kim, A. D. Ames, and K. A. Hamed, "Distributed data-driven predictive control for multi-agent collaborative legged locomotion," *IEEE International Conference on Robotics and Automation*, June 2023.

[5] J. Kim, R. T. Fawcett, V. R. Kamidi, A. D. Ames, and K. A. Hamed, "Layered control for cooperative locomotion of two quadrupedal robots: Centralized and distributed approaches," *arXiv preprint arXiv:2211.06913*, In Review, March 2023.

[6] R. T. Fawcett, A. Pandala, J. Kim, and K. Akbari Hamed, "Real-Time Planning and Nonlinear Control for Quadrupedal Locomotion With Articulated Tails," *Journal of Dynamic Systems, Measurement, and Control*, vol. 143, no. 7, February 2021.

[7] R. T. Fawcett, K. Afsari, A. D. Ames, and K. Akbari Hamed, "Toward a data-driven template model for quadrupedal locomotion," *IEEE Robotics and Automation Letters*, vol. 7, no. 3, pp. 7636–7643, 2022.

[8] A. Pandala, R. T. Fawcett, U. Rosolia, A. D. Ames, and K. A. Hamed, "Robust predictive control for quadrupedal locomotion: Learning to close the gap between reduced-and full-order models," *IEEE Robotics and Automation Letters*, vol. 7, no. 3, pp. 6622–6629, 2022.

[9] A. Hereid and A. D. Ames, "Frost: Fast robot optimization and simulation toolkit," in *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. Vancouver, BC, Canada: IEEE/RSJ, Sep. 2017.

[10] J. Carpentier, G. Saurel, G. Buondonno, J. Mirabel, F. Lamiraux, O. Stasse, and N. Mansard, "The pinocchio c++ library : A fast and flexible implementation of rigid body dynamics algorithms and their analytical derivatives," in *2019 IEEE/SICE International Symposium on System Integration (SII)*, 2019, pp. 614–619.

[11] M. H. Raibert, *Legged robots that balance*. MIT press, 1986.

[12] R. T. Fawcett, *Real-Time Planning and Nonlinear Control for Robust Quadrupedal Locomotion with Tails*. MS thesis, Virginia Tech, Advisor: K. Akbari Hamed, 2021, Available Online: https://7c91a126-a82e-473d-af78-5fcdc5f4feae.filesusr.com/ugd/d9fe13_fba8bc68a95f451da349e63be2601bf1.pdf.