# llm_feedback_for_control

lm_ctrl_team

August 2025

## 1 Introduction

Welcome back! This semester, we will transition from exploration to implementation. Over the summer, you gained a valuable, first-hand look at the complexities of using Large Language Models (LLMs) to generate robotic behaviors. Now, we will build on that experience with a more focused and grounded approach. Our goal is to channel our efforts into a concrete project focused on quadrupedal robots.

**Quadrupeds** We will focus our work on quadrupeds for two strategic reasons. First, quadruped locomotion via a model predictive controller (MPC) has been well-studied , with a rich body of literature on how the MPCs should be formulated. This provides a strong foundation that both we and the LLMs can draw upon. Second, our work in simulation will directly map to the Unitree Go2 robot we have in the lab, creating a clear and exciting path to future hardware experiments.

**Trajectory Generation vs. MPC** To start, we will simplify the control problem. Instead of designing a full MPC that replans at every step, we will focus on solving a single trajectory optimization problem. This will give us a sequence of actions that we can execute in an open loop. While we will eventually need to introduce feedback to be able to execute the planned trajectory robustly, this simplified setup allows for faster iteration and reduces the computational burden, which is ideal for the initial development phase.

**The "Backflip" Challenge** Our central goal is to build a system that can translate the command, "Do a backflip," into the precise torque commands that make it a reality. A backflip is a highly dynamic and challenging maneuver, making it an excellent benchmark for our system. Is it possible to formulate a trajectory optimization problem to generate a viable backflip trajectory? The answer is yes. Trajectory optimization was used to generate a backflip on the MIT Cheetah robot six years ago (Katz et al., 2019) (Video)[1]. While many papers have followed to use trajectory optimization to generate dynamic behaviors for quadrupeds, the optimization problem has always involved careful hand design. The question we want to answer is whether we can automate this process by leveraging the powers of LLMs.

[1] Our Go2 robot should also be more than sufficient for the task (See this video).

| Description | Symbol | Dim. | Frame |
|---|---|---|---|
| Position of Base CoM | $\mathbf{p}$ | $3 \times 1$ | World ($\mathcal{W}$) |
| Linear Velocity of Base | $\dot{\mathbf{p}}$ | $3 \times 1$ | World ($\mathcal{W}$) |
| Orientation of Base | $\mathfrak{q}$ | $4 \times 1$ | N/A |
| Angular Velocity of Base | $\boldsymbol{\omega}$ | $3 \times 1$ | Body ($\mathcal{B}$) |
| Joint Angles | $\boldsymbol{\theta}$ | $12 \times 1$ | N/A |
| Joint Velocities | $\dot{\boldsymbol{\theta}}$ | $12 \times 1$ | N/A |
| Joint Torques | $\boldsymbol{\tau}$ | $12 \times 1$ | N/A |
| Contact Forces (per foot) | $\mathbf{f}_c$ | $3 \times 1$ | World ($\mathcal{W}$) |

Table 1: Table of relevant quantities.

## 2 Background: The Language of Quadruped Locomotion

We now introduce some basic concepts when it comes to modeling quadruped locomotion, which will be useful in formulating our trajectory optimization problems.

### 2.1 Modeling the Dynamics of Quadrupeds

The quadruped is typically modeled as a *floating-base system*. This model consists of an unactuated body (the floating base or torso) connected to four actuated legs, as illustrated in Figure 1. Each leg has three actuated joints: hip abduction/adduction (HAA), hip flexion/extension (HFE), and knee flexion/extension (KFE). This gives the system a total of 6 (for the base) + $4 \times 3 = 18$ degrees of freedom (DoF). To describe the state of the quadruped, we need the variables in Table 1.

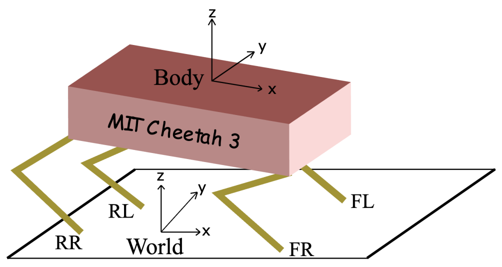

Figure 1: Simplified Model of Quadrupeds. Image taken from Di Carlo et al. (2018).

Note that we represent the body's orientation using a quaternion $\mathfrak{q} \in \mathbb{R}^4$ to avoid the singularities associated with Euler angles. The angular velocity $\boldsymbol{\omega}$ is conventionally expressed in the body frame $\mathcal{B}$. The *generalized coordinates* $\mathbf{q} \in \mathbb{R}^{19}$ combine the base pose and joint angles, $\mathbf{q} = (\mathbf{p}, \mathfrak{q}, \boldsymbol{\theta})$. The *generalized velocities* $\dot{\mathbf{q}} \in \mathbb{R}^{18}$ are $\dot{\mathbf{q}} = (\dot{\mathbf{p}}, \boldsymbol{\omega}, \dot{\boldsymbol{\theta}})$. The complete state of the robot is therefore $\mathbf{x} = (\mathbf{q}, \dot{\mathbf{q}})$.

We note that $\mathbf{x}$ fully determines the kinematic state of its feet. The Cartesian position of any foot $i$, denoted $\mathbf{p}_{f,i}$, can be computed from the generalized coordinates $\mathbf{q}$ via what's called the *forward kinematics* map, $\mathbf{p}_{f,i} = \text{FK}_i(\mathbf{q})$. Likewise, the foot's Cartesian velocity, $\dot{\mathbf{p}}_{f,i}$, is obtained by applying the configuration-dependent *Jacobian matrix* $\mathbf{J}_{p,i}(\mathbf{q})$ to the generalized velocities $\dot{\mathbf{q}}$, such that $\dot{\mathbf{p}}_{f,i} = \mathbf{J}_{p,i}(\mathbf{q})\dot{\mathbf{q}}$. Therefore, full knowledge of the robot's state provides a complete picture of the position and velocity of all its contacts. (For convenience, foot position and velocity are usually expressed in the body frame.)

The dynamics of this floating-base system are governed by the manipulator equations:

$$\mathbf{M}(\mathbf{q})\ddot{\mathbf{q}} + \mathbf{C}(\mathbf{q}, \dot{\mathbf{q}})\dot{\mathbf{q}} + \mathbf{G}(\mathbf{q}) = \mathbf{B}\boldsymbol{\tau} + \mathbf{J}_c^\top \mathbf{f}_c. \tag{1}$$

While you do not need to fully understand this equation[2], we would like to highlight the two quantities on the right-hand side of the equation. First, $\boldsymbol{\tau}$, the joint torques, is what we get to control. In other words, our job is to figure out what torques we send to the joints to drive the robot to perform our desired behavior. Secondly, the term $\mathbf{J}_c^\top \mathbf{f}_c$ represents the generalized forces resulting from external contacts (i.e., the *ground reaction forces* $\mathbf{f}_c$ at the feet), where $\mathbf{J}_c$ is the contact Jacobian. Usually, the complexity of legged locomotion arises from this term, as the robot must intelligently make and break contact with the environment.

## 2.2 Quadruped Trajectory Optimization

In this section, we introduce some key components in formulating a trajectory optimization problem for quadrupeds that can be written as:

$$\begin{aligned}
\min_{\substack{\{\bar{\mathbf{x}}_k\}_{k=0}^{N} \\ \{\bar{\mathbf{u}}_k\}_{k=0}^{N-1}}} \quad & \sum_{k=0}^{N-1} J(\bar{\mathbf{x}}_k, \bar{\mathbf{u}}_k) && \text{(2a)} \\
\text{subject to:} \quad & \bar{\mathbf{x}}_{k+1} = \bar{f}(\bar{\mathbf{x}}_k, \bar{\mathbf{u}}_k) \quad \cdots \quad \text{(Dynamics)} && \text{(2b)} \\
& \text{Friction Cone Constraints} && \text{(2c)} \\
& \text{No-Slip Constraints} && \text{(2d)} \\
& \text{Foot Height Constraints} && \text{(2e)}
\end{aligned}$$

Here, $J(\bar{\mathbf{x}}_k, \bar{\mathbf{u}}_k)$ is a cost function, which could be designed to minimize control effort (e.g., $\|\mathbf{f}_{c,k}\|^2$), to track a reference trajectory, or, in the case that any feasible trajectory, simple 0.

**Key Assumption** We will assume that *the contact schedule is given*, i.e., before the optimization begins, we already know which feet are supposed to be on the ground and which are in the air at every point in time. Working to relax this assumption is key in our future steps. This lets us define two modes for each leg:

- *Stance Phase:* The foot is on the ground, providing support and generating force.
- *Swing Phase:* The foot is in the air, repositioning for its next foothold.

Importantly, the friction cone, no-slip, and foot height constraints all apply only to the stance foot. Let us examine each constraint in detail.

**Simplified dynamics:** While the full and complex manipulator equations are great for modeling and simulation, they are often too complex to be used as the dynamics model in optimization. We Instead consider a simplified "kinodynamic" model with the state $\bar{\mathbf{x}} := (\mathbf{p}\ \dot{\mathbf{p}}\ \mathfrak{q}\ \boldsymbol{\omega}\ \boldsymbol{\theta}) \in \mathbb{R}^{24}$ and control input $\bar{\mathbf{u}} := (\dot{\boldsymbol{\theta}}\ \mathbf{f}) \in \mathbb{R}^{24}$. Here $\mathbf{f}_{c,k}$ stands for the ground reaction forces (GRFs) for each foot. By choosing GRFs as a direct control input, we sidestep having to model the intricate relationship between joint torques, mass distribution, and contact forces. We are essentially saying, "assume we can command these forces directly," and we let the optimizer find the forces needed to move the CoM as desired. After the GRFs are found, we can recover the torques needed to generated them via what's called *inverse dynamics.*

[2]If you want to, we encourage you to read more about it here, or take MEAM 520 for a deeper dive.)

**Friction Cone Constraints** As you probably learned in highschool physics, a surface can only provide a limited amount of tangential (frictional) force before an object begins to slip. This limit is proportional to the normal force pushing the foot onto the surface. This relationship defines a "friction cone". To prevent the robot's feet from slipping, we enforce that the GRF vector $\mathbf{f}_{c,i}$ for each stance foot must lie within this cone. Let $\mathbf{f}_{c,i} = (f_x, f_y, f_z)$, where $f_z$ is the normal force. The constraint is:

$$\sqrt{f_x^2 + f_y^2} \leq \mu f_z$$

Here, $\mu$ is the coefficient of friction between the foot and the ground. This nonlinear constraint is often linearized into a "pyramid" shape for computational efficiency.

**No-Slip Constraint** A foot on the ground must not be moving. More formally, for any foot $i$ that is in a stance phase at time $k$, its velocity must be zero. We use the Jacobian to relate the robot's generalized velocities to the foot's Cartesian velocity and set the result to zero.

$$\dot{\mathbf{p}}_{f,i} = \mathbf{J}_{p,i}(\mathbf{q}_k)\dot{\mathbf{q}}_k = \mathbf{0}.$$

This constraint ensures that the robot's body and leg motions are coordinated in a way that keeps the stance feet stationary on the ground.

**Foot Height Constraint** This constraint also enforces the definition of the stance phase by ensuring the foot is actually on the ground, i.e., its vertical position must be zero. We use forward kinematics to find the foot's position and constrain its z-component.

$$[\mathbf{p}_{f,i}]_z = [\mathrm{FK}_i(\mathbf{q}_k)]_z = 0.$$

This ensures that the robot does not float above or penetrate the ground. For swing feet, we would typically add a constraint that $[\mathbf{p}_{f,i}]_z \geq z_{\mathrm{min\_clearance}}$ to avoid scuffing the ground.

## 2.3 Constraints for Backflipping

The constraints discussed so far—dynamics, friction, and contact state—are universal to any feasible quadruped motion. To elicit a specific, complex behavior like a backflip, we must augment this universal set with *task-specific constraints.*

In the original backflip paper (Katz et al., 2019) (see §IV.B.1), the authors also presumed a known contact schedule. While their optimization used a 2D model with full rigid-body dynamics and joint torques as direct inputs, the high-level constraint structure is relevant to our simplified 3D kinodynamic model. The key task-specific constraints they introduced were:

- **Initial and Terminal State:** The robot must begin and end in a stable, standing posture. The terminal body pitch is constrained to be approximately $2\pi$ radians greater than the initial pitch, enforcing a full rotation.
- **Ground Clearance:** During the flight phase, all points on the robot's body must remain above the ground plane to prevent collision.
- **Self-Collision Avoidance:** The geometry of the legs must be constrained to prevent them from intersecting with each other or the robot's body, especially during the tuck phase of the flip.

- **Actuator Limits:** The commanded forces and motions must be achievable by the robot's motors. In a full dynamics model, this corresponds to ensuring joint torques do not exceed their physical limits.

The central thesis of our project is to automate the creation of these task-specific constraints. While the universal dynamics constraints will form the basis of every optimization problem, we want to leverage an LLM to interpret a high-level command like "do a backflip" and automatically formulate the specific constraints (like the four listed above) required to produce the desired motion.

## 3 Getting Started: Introduction to the Codebase

As our starting point, I have provided a sample trajectory optimization pipeline, which you can access at this GitHub repository. This section will guide you through setting up the environment and understanding the core components of the code.

### 3.1 Running the Simulation

**Docker Setup (Optional)** The necessary dependencies for this project are pre-packaged in the Docker image `pympc` on the trebuchet server. To set up the environment on a local machine, download this dockerfile to a desired directory and run `docker build -t pympc .`. Note that this will use approximately 10GB of disk space.

**Environment Setup** First, clone the project repository. Then, launch the Docker container using the following command, replacing `PATH_TO/CLONED_REPO` with the actual path to the repository on your machine:

```
docker run -it -v PATH_TO/CLONED_REPO:/home --net=host pympc bash
```

This command starts the container and maps the project directory into the `/home` folder within the container. To verify your setup, run the following script from inside the container:

```
python examples/env.py
```

If successful, this will run an open-loop simulation of a Go2 robot and save the resulting trajectory as a video in `results/env_test.mp4`.

**The Simulation Environment** The script above also prints a description of the simulation environment's observation and action spaces. We use the `gym_quadruped` environment, which provides observations such as the robot's base pose and velocity, foot positions, and joint states. You can see in `examples/env.py` how the environment is instantiated and used in a standard Gym-like simulation loop.

### 3.2 Solving a Trajectory Optimization Problem

Now we move to the core of the project. To see an example of trajectory optimization, run:

```
    python main.py
```

This script solves an optimization problem and creates two videos in the `/home/results/` folder. The `planned_traj.mp4` video visualizes the state trajectory solved for by the optimizer. The `trajectory.mp4` video shows the result of applying the calculated control inputs in the simulation, with the planned trajectory overlaid for comparison. Ideally, these two trajectories should match. The current discrepancy is a key issue we will work to resolve.

In `main.py`, the optimization problem (the `HoppingMPC` class) is created as an object of class `KinoDynamic_Model` and then solved by calling the `mpc.solve_trajectory(...)` method. Because we use a simplified kinodynamic model for planning, we must also use inverse dynamics to translate the optimized ground reaction forces and joint velocities into the joint torques required by the simulator.

## 3.3 Unpacking the MPC Implementation

The MPC implementation is split into two main files: `examples/model.py` and `examples/mpc.py`.

- `model.py`: Defines the robot's kinodynamic model. It uses the `ADAM` library to compute kinematic properties like foot positions from the robot's state and uses simplified Newton-Euler equations to define the system dynamics, as shown in Equation (2b).
- `mpc.py`: Implements the trajectory optimization problem using the model from `model.py`. It uses the `Acados` framework, which formulates problems with box constraints of the form $h_l^k \leq h^k(\mathbf{x}, \mathbf{u}) \leq h_u^k$. This rigid structure enables efficient solving, but makes the implementation much less readable.

Below is a breakdown of how the key physical constraints are implemented within this framework. In general, the constraints are implemented by two components: (1) a `xxx_constraints_expr` method that defines the expression $h^{\text{xxx}}(\mathbf{x}, \mathbf{u})$, and (2) a `compute_xxx_constraints` method that finds and sets the appropriate bounds $h_l^{\text{xxx}}$ and $h_u^{\text{xxx}}$ based on the contact schedule.

**Friction Cone Constraints** This constraint prevents the feet from slipping. Instead of modeling the true (nonlinear) friction cone, the `_friction_cone_constraints_expr` method implements a common **linear approximation**, often called a "friction pyramid." For each foot, five linear inequalities are created. Four of these define the sides of a pyramid that contains the ground reaction force vector, ensuring that the tangential force is proportional to the normal force. The fifth inequality enforces minimum and maximum limits on the normal force itself. These constraints are only active for feet in the stance phase. This is done by setting $h_l = -\infty$ and $h_u = \infty$ in the `_compute_friction_cone_constraints` method.

**Foot Height Constraints** This constraint ensures that stance feet are on the ground and swing feet are in the air. The method `_foot_height_constraints_expr` uses the forward kinematics functions from `model.py` to get the vertical ($z$) position of each foot. The actual constraint bounds are then set dynamically in `_compute_foot_height_constraints`. For each time step, if a foot is scheduled to be in stance, its vertical position is constrained to be zero (e.g., within $[0, 0]$). If the foot is in swing, its height is constrained to be non-negative ($[0.0, 10^6]$), preventing it from penetrating the ground.

**Foot Velocity (No-Slip) Constraints** This constraint enforces the no-slip condition for stance feet. The implementation mirrors the foot height constraint's logic. First, `_foot_velocity_constraints_expr` uses the Jacobian matrix from `model.py` to create a symbolic expression for each foot's Cartesian velocity. Then, the `_compute_foot_velocity_constraints` function sets the bounds for this expression at each time step. If a foot is in stance, its velocity is constrained to be zero. If the foot is in swing, the velocity bounds are made very large, placing no restriction on its motion.

### 3.4 The Current State and Our Goal

While the code implements the constraints we need, the solver struggles to find a solution without significant relaxations (we need to relax the foot height bounds to be $[0, 0.1]$ and no-slip bounds to be $[-1.0, 1.0]$ while both should be $[0., 0.]$). As a result, the current "solution" is not physically realistic: the robot appears to bounce on landing (foot height is not strictly zero), and the stance feet slide (the no-slip condition is not strictly enforced). Our immediate goal is to diagnose these numerical issues and adjust the problem formulation to generate physically plausible trajectories.

## 4 The Next Step: Building a Robust Optimization Foundation

Our immediate goal is to address the numerical issues in the current pipeline to ensure we can generate physically realistic trajectories.

**1. Transcription to the Opti Framework** Your first and most important task is to transcribe the current Acados-based implementation into one built using CasADi's `Opti` framework. Since `Opti` is built on CasADi, much of the existing symbolic math from `model.py` can be reused. This transcription is crucial for several reasons:

- *Intuitive Formulation:* The `Opti` interface more closely mirrors the mathematical structure of an optimization problem, which will make it easier for us to debug and for an LLM to eventually generate.
- *Faster Iteration:* Acados is optimized for repeatedly solving the same problem (MPC), by first compiling the optimization problem into C code. For our single-shot trajectory optimization, this compilation creates unnecessary overhead. The `Opti` interface allows for faster development iterations.
- *Support for Complementarity Constraints:* Our long-term goal is to avoid specifying contact schedules by hand. This requires modeling contact with complementarity constraints, which are natively supported by `Opti` but not by Acados.

This task is designed to familiarize you with the core tools of our project. To get started, please take a look at the CasADi documentation, focusing on sections 1-4 (Basics) and 9 (The Opti stack).

**2. Generating Warmstart Trajectories** Nonlinear solvers are highly sensitive to the initial guess provided to them. A primary reason our current solver fails with tight constraints is its poor initialization (starting from $\mathbf{x} = \mathbf{0}$ and $\mathbf{u} = \mathbf{0}$). After transcribing the problem to `Opti`, your next task is to implement a function that generates a "warm-start" trajectory. Experiment with providing a more intelligent initial guess, such as a simple parabolic arc for the CoM, and observe how it affects solution quality and convergence.

**3. Improving Simulation and Visualization (Optional)** The current simulation and rendering loop is slow. Furthermore, visualizing additional data such as ground reaction force vectors or the body's acceleration could be highly beneficial for debugging. If you have time, contributions to optimize the rendering process or add new visualization features would be very valuable.

## 5 Outlook: Integration with Language Models

As mentioned before, the ultimate goal of this project is to have a language model, rather than a human, generate the task-specific constraints for a given command. This is where your experience from the summer becomes directly applicable. As a first step, we will build a pipeline that follows the iterative refinement logic shown in Algorithm 1. Implementing this loop requires three key components:

- A system prompt that instructs an LLM to generate optimization constraints (as CasADi code) from a user command and the results of the previous trajectory.
- A robust parser to safely execute the generated code, construct the optimization problem, and run the simulation.
- A feedback mechanism to package the simulation results into a new prompt for the next iteration of refinement.

We will discuss the design of these components in more detail in our upcoming meetings.

## References

Jared Di Carlo, Patrick M Wensing, Benjamin Katz, Gerardo Bledt, and Sangbae Kim. Dynamic locomotion in the mit cheetah 3 through convex model-predictive control. In *2018 IEEE/RSJ international conference on intelligent robots and systems (IROS)*, pages 1–9. IEEE, 2018.

Benjamin Katz, Jared Di Carlo, and Sangbae Kim. Mini cheetah: A platform for pushing the limits of dynamic quadruped control. In *2019 international conference on robotics and automation (ICRA)*, pages 6295–6301. IEEE, 2019.

**Algorithm 1:** Iterative Refinement Pipeline

```
Input: Language Command $\ell$, System Prompt $P$,
       Oracle $O$, Dynamics Model $\dot{\mathbf{x}} = \bar{f}(\bar{\mathbf{x}}, \bar{\mathbf{u}})$
Output: A set of $k$ constraints $h(\mathbf{x}, \mathbf{u}) \preceq 0$
1 LM Context $\mathcal{C}^0 \leftarrow P + \ell$;
2 for $i = 1, 2, \dots$ do
3     $h^i(\cdot, \cdot) \leftarrow LM(\mathcal{C}^i)$;
4     Find reference trajectory $\mathbf{x}^i_{\text{ref}}, \mathbf{u}^i_{ref} \leftarrow$ Solve the
        MPC problem (2) with additional constraints
        $h^i(\mathbf{x}, \mathbf{u}) \preceq 0$;
5     $\mathbf{x}^i, \mathbf{u}^i \leftarrow$ Execute the reference trajectory;
6     $\mathcal{C}^{i+1} \leftarrow$ Improve$(LM, \mathcal{C}^i, \mathbf{x}^i, \mathbf{u}^i, h^i)$;
7 return $h$
```