Quadruped Robot Control and Optimization

Jiří Bláha Giulia Cortelazzo Semih Zaman

Abstract

This report presents the implementation and evaluation of quadruped robot control using `Python` and the `PyBullet` engine. The control of the robot gets progressively more advanced. Firstly, we go over forward jumping, before generalizing the control to lateral jumping and twist jumping to obtain an omnidirectional Cartesian proportional-derivative controller. Moreover, the effect of virtual model control on the system is observed. After, we move to Bayesian optimization using the `Optuna` library, to further optimize the variables controlling the jumping behaviour. Lastly, the appendix shows important plots and results, as well as our stance on Gen AI.

Background

A full, concise version of the theoretical background to this problem is available in [1]. We therefore kindly ask the reader to refer to this document for reference.

1 Introduction

The idea behind this project is to design, stress-test, and subsequently optimize a jumping controller for a given quadruped robot. More specifically, we are looking to create an omnidirectional controller, with which the quadruped can perform several jumping styles that should get progressively more difficult. Namely, these include forward jumping, lateral jumping, and twist jumping. For this specific task, a Cartesian PD controller is used, expecting us to find appropriate gains for the controller, such that all jumping styles are stable and thereby satisfactory. The controller design itself shall be covered within §2 of this report. After the controller is designed, using somewhat of a brute force approach, the task should continue with the introduction of optimization to the problem, resulting in us being able to perform the furthest forward, lateral, and twist jumps. At the outset, this should be achieved using only the force profile variables as optimization variables, though the assignment allows for further optimization endeavours among other variables of the controller. Overall, the optimization portion of the work is discussed within §3 of this report.

2 Controller

Let us start with the basic, so-called vanilla, version of the jumping controller. The reader should note that the code structure for this problem was somewhat modified to facilitate easier use, idea exploration, and debugging. Specifically, the script `quadruped_jump.py` was altered and accompanied with the newly created file `jump_profiles.py` to accommodate switching between all possible jump types. Therefore, it can now be ran with the following CLI command:[1]

```
python quadruped_jump.py <jt>
```

where the `jt` positional argument specifies the desired jumping behaviour and can take on the values

- `forward` for forward jumping,
- `lateral_left` or `lateral_right` for lateral jumping (left/right),

[1] Note that `python3` instead of `python` may need to be used, depending on the distribution.

- and `twist_cw` or `twist_ccw` for twist jumping (cw/c-cw),

covering all requested jump types, making the controller an omnidirectional one.

2.1 Architecture

The code architecture for the controller portion of the project essentially follows the assignment, apart from some of the aforementioned refactoring. In terms of imports, all dependencies used on top of the original assignment can be found in the `requirements.txt` file, and their installation follows the standard dependencies installation within a conda environment. The main entry point of the script is the `quadruped_jump()` function, defined in the following form:

```
def quadruped_jump(
    jt: str
) -> None:
```

and housing other core functions for simulating the quadruped. The end of the file is completed with a standard main guard, so that when the aforementioned CLI command is ran, the code block

```
if __name__ == "__main__":
<upper guard code>
    if not compare:
        quadruped_jump(jt)
<lower guard code>
```

is executed. Besides the main core, the script contains some utility and plotting code. Additionally two other files accompany the main script. The `profiles.py` file defines the class `FootForceProfile`, housing methods for creating force profiles at the feet. Additionally, to achieve the previously mentioned modularity, the `jump_params.py` dictionary was also created, housing all the local variables, unique to each specific jump type.

2.2 Global Variables

The global variables, i.e., variables that need to stay constant across all jump types, are the gains of the Cartesian controller. Surprisingly, their values have been determined almost exclusively by starting with forward jumping, and have not been altered much when generalizing to other jump types. They read[2,3]

```
KP_XY = 280.0
KP_Z = 1250.0
KD_FLIGHT_XY = 8.0
KD_FLIGHT_Z = 8.0
KD_STANCE_XY = 40.0
KD_STANCE_Z = 150.0
```

and, as was already mentioned, apply to all of the examined jump types.

2.3 Local Variables

As it has been disclosed previously, we term the jump-specific variables the local variables. Starting with forward jumping, we have found the best nominal foot position, in combination with the aforementioned global variables, to be[4]

```
"Z_OFFSET": -0.220,
"Y_OFFSET": 0.0838,
"X_OFFSET": 0.0011,
```

with the value of `Y_OFFSET` being determined directly from the robot's geometry, specifically the lateral distance between the body center and each hip joint in the `URDF` (unified robot description format) model used in the quadruped simulation. This fact in turn means that, for forward jumping, the legs are nor pointed inwards, nor outwards, and are found aligned with the body in the lateral direction instead. The offsets `X_OFFSET` and `Z_OFFSET` have been determined by virtue of trial and error, to achieve the best possible jumping behaviour when combined with the chosen force profile parameters. For completeness, we list all of the local variables for forward jumping (from `jump_params.py`) ahead.

[2] For reference, see `quadruped_jump.py`.

[3] Note that due to symmetry of the quadruped, we have been able to group the `X` and `Y` values.

[4] For reference, see `jump_params.py`.

```
JUMP_PARAMS = {
    "forward": {
        "IMPULSE_F0": 3.8,
        "IDLE_F1": 1.0,
        "FORCE_FX": 140.0,
        "FORCE_FY": 0.0,
        "FORCE_FZ": 370.0,
        "N_JUMPS": 15,
        "K_VMC": 0.0,
        "Z_OFFSET": -0.220,
        "Y_OFFSET": 0.0838,
        "X_OFFSET": 0.0011,
    },
    <other jump types>
```

It is important to note that while we address the offsets as being determined by the chosen force profile parameters, we have rather gone back and forth on all the parameters simultaneously until a satisfactory outcome was achieved. This, somewhat ad hoc, design process resulted in the center of mass being located quite low. Going even lower results in a considerable amount of drifting away from a straight line, yet this effect is also slightly present for the final chosen value. When raising the center of mass, this effect again becomes more and more obvious, up to a point where the quadruped almost does not move forward at all, again underlining why the chosen value gives the best compromise between performance and stability.

2.4 Omnidirectionality

Achieving omnidirectionality essentially entails finding such local variables for each jump type so that, in combination with the global gains, it performs the best while staying stable. For obvious reasons, we choose not to list the explicit local variables for all of the other jump types as they can be found in `jump_params.py`. As far as lateral jumping (in both directions) is concerned, the most important observation is that the forces excreted at the feet need to be considerably smaller compared to the forward jumping case. Additionally, both the impulse and idle frequencies are smaller, making the respective durations larger, and the nominal foot position also differs. Here, the quadruped sits slightly higher, with legs pointed outward in the lateral direction. Surprisingly, while the force profile for both the left and right cases is the same, the lateral offset of the legs needs to be different. We attribute this fact to suboptimal controller design, as the global variables need to cover all jump types. Hence, for any semi-stable combination of local variables for the lateral jump we could find, it was still struggling considerably. Overall, we have found lateral jumping to be the most sensitive to the global variables and thereby the most challenging to get right. This is why it falls short in speed and precision in comparison to forward jumping.[5]

For what concerns twist jumping (in both directions) the nominal foot position is almost identical to that in the forward jumping case. Here, the only difference is in the `X_OFFSET`, which is larger, i.e., the legs are pushed more forward. The frequencies driving the force profile generator are also identical, yet respective magnitudes are smaller and, exclusively for this case, all nonzero. As we are looking to generate yaw torque, and not simply translate again, we need opposite-sign tangential forces between diagonal leg pairs. This is achieved using

```
def per_leg_force(
    F_base: np.ndarray,
    leg_id: int,
    jt: str
) -> np.ndarray:
```

which we take liberty to define more rigorously in the following lines. Let

$$\mathbf{f}_{\text{base}} = \begin{bmatrix} f_x & f_y & f_z \end{bmatrix}^T$$

be the base (nominal) force profile applied by the controller. Each leg j of the

[5] And also twist jumping.

robot, with $j \in \{0,1,2,3\}$, is associated with sign coefficients of the form

$$\sigma_x^{(j)} \in \{+1,-1\}, \quad \sigma_y^{(j)} \in \{+1,-1\}$$

describing the leg position, i.e., left/right, front/back. Further, define

$$\sigma = \begin{cases} +1 & \text{for c-cw twist} \\ -1 & \text{for cw twist} \end{cases}$$

to be the twist coefficient. What follows is the conditional statement[6]

```
if jt in ("twist_ccw",
↪  "twist_cw"):
    $\sigma = +1.0$ if jump_type ==
    ↪  "twist_ccw" else $\sigma = -1.0$
    $f_x^{(j)} = \sigma\sigma_x^{(j)} f_x$
    $f_y^{(j)} = \sigma\sigma_y^{(j)} f_y$
    return np.array([$f_x^{(j)}$, $f_y^{(j)}$,
    ↪  $f_z$], dtype=float)
else:
    return $\mathbf{f}_{\text{base}}$
```

which returns the standard force profile across all legs for translational jump types, i.e., forward and lateral jumping. In terms of performance, the twist jump falls between the forward jump and the lateral jump, meaning it performs better than the lateral jump, while performing slightly worse than the forward jump.

2.5 Virtual Model Control

Finally, we conclude the controller portion of this work with virtual model control (VMC). The virtual model has varying effects for different jump types. For instance, in forward jumping, VMC is not being used at all as it does not bring any obvious benefit when evaluating the simulation performance, hence, the gain `K_VMC` can be seen as set to zero within the local variables for the forward jump. As for other jump types, we rely the heviest on VMC in both twist jumps and the lateral right jump, all with a gain of `10.0`. As for the lateral left jump, `K_VMC` is set to `8.0`. Coming back to the question why VMC is not so helpful, we provide the answer that during basic jumping, the body does not experience major oscillation. Hence, we suggest VMC would be more beneficiary in the case of walking or running. Nevertheless, the effect of VMC for our case is shown as part of the appendix, where we look at the roll and pitch of the base over time with VMC off, and with VMC on.[7]

3 Optimization

With the basic controller set up, let us move on to optimizing some of its parameters. The reader should note that, in similar fashion to the controller, the code structure for this problem was again somewhat modified for the same purposes. This essentially means that running the script now requires executing the following CLI command:[8]

```
python quadruped_jump_opt.py <jt>
```

where the `jt` positional argument again bares the same meaning as in the plain controller case, with the minor difference of it now being the jumping behavior we are looking to optimize. In terms of the optimization itself, we shall be using Bayesian optimization, specifically the `Optuna` library, easily importable with

```
import optuna
```

i.e., via the standard `import` syntax. The decision (optimization) variables for this problem are the force profile variables, namely the force values f_x, f_y, f_z, and the impulse/idle frequencies φ_0, φ_1. On the other hand, the global gains of the Cartesian controller and the feet offsets for each jump type are not considered among the decision variables given they were configured in the controller portion.

[6]Note that for illustrative purposes, the statement does not follow `Python` syntax exactly.

[7]For this comparison, we have extended the basic CLI command with the `-compare-vmc` flag.

[8]Note that `python3` instead of `python` may need to be used, depending on the distribution.

3.1 Architecture

The code architecture of the optimization portion is more complex than that of the controller one. For this reason, we shall not go into specifics as much and refer the reader to the code itself instead. On the other hand, we feel it is important to explain at least the high-level structure. The main optimization script, `quadruped_jump_opt.py`, comes with three other, newly created files. The `jump_params_opt.py` file follows a similar structure to `jump_params.py`, and houses the decision variables (and their ranges) for each jump type. For post optimization testing and visualization, the file `optimals_demo.py` file was created. When ran using the CLI command[9]

```
python optimals_demo.py <jt>
```

it demos the solution for a specific jump type `jt`, pulling the optimal values from `optimals.py`, concluding our high-level review of the code architecture.

3.2 Jumping Optimization

Achieving furthest jumps can ultimately be described globally for all jump types as a single optimization problem

$$\begin{aligned} &\max_{\boldsymbol{\zeta}\in[\boldsymbol{\alpha},\boldsymbol{\beta}]} \Delta^{(\mathtt{jt})}(\boldsymbol{\zeta}) \\ &\text{s.t. } g_1(\boldsymbol{\zeta}) = \bar{z} - z(\boldsymbol{\zeta}) \le 0 \\ &\qquad g_2(\boldsymbol{\zeta}) = \chi(\boldsymbol{\zeta}) - \bar{\chi} \le 0 \end{aligned}$$

where $\boldsymbol{\zeta}$ is the decision variables vector, box-bounded between some $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$,

$$\Delta^{(\mathtt{jt})}(\boldsymbol{\zeta}) = \begin{cases} +\Delta x(\boldsymbol{\zeta}) & \text{(forward)} \\ +\Delta y(\boldsymbol{\zeta}) & \text{(lateral left)} \\ -\Delta y(\boldsymbol{\zeta}) & \text{(lateral right)} \\ +\Delta \psi(\boldsymbol{\zeta}) & \text{(twist c-cw)} \\ -\Delta \psi(\boldsymbol{\zeta}) & \text{(twist cw)} \end{cases}$$

and the functions $g_1(\boldsymbol{\zeta})$ and $g_2(\boldsymbol{\zeta})$ constrain the problem to some specified minimum permitted body height (threshold of $\bar{z}$) and maximum allowed body tilt (threshold of $\bar{\chi}$) during the trial, respectively. When either of the inequality constraints $g_1(\boldsymbol{\zeta})$ or $g_2(\boldsymbol{\zeta})$ is violated, the optimization is handled through penalty-based relaxation. Instead of enforcing the constraints strictly, a penalty term $\Pi(\boldsymbol{\zeta})$ is computed, yielding a new objective

$$\max_{\boldsymbol{\zeta}\in[\boldsymbol{\alpha},\boldsymbol{\beta}]} \Delta^{(\mathtt{jt})}(\boldsymbol{\zeta}) - \Pi(\boldsymbol{\zeta}),$$

in which the penalty reads

$$\begin{aligned} \Pi(\boldsymbol{\zeta}) = \gamma_z\, &\mathbb{I}\{g_1(\boldsymbol{\zeta}) > 0\} \\ + \gamma_\chi\, &\mathbb{I}\{g_2(\boldsymbol{\zeta}) > 0\}, \end{aligned}$$

where $\mathbb{I}\{\cdot\}$ is the indicator function.

The aforementioned optimization problem is run across 10 different seeds using a `for` loop in the main guard, with 40 trials per seed. Explicitly speaking, the main values of the problem read

```
N_TRIALS = 40
HEIGHT_THRESH = 0.08
HEIGHT_PENALTY = 2.0
TILT_THRESH_DEG = 30.0
TILT_PENALTY = 1.5
```

and, additionally,

```
seeds = [
    10 * i for i in range(10)
]
```

in the main guard. After optimization has finished, all scores for all seeds are plotted, with the best being highlighted and its values saved to `optimals.py` under the corresponding jump type.

Overall, the twist jump experienced the biggest improvement after optimization, effectively making the quadruped jump around 180 degrees in one jump. On the other hand, the lateral jump continued to struggle and was again the most sensitive jump type, mainly to the left. We again attribute this fact to suboptimal controller design, as discussed in §2.

[9] Note that `python3` instead of `python` may need to be used, depending on the distribution.

3.3 Hopping Controller

The continuous hopping controller optimization problem was formulated as a variation of the furthest jumps problem. Specifically, the overall problem remains essentially the same, but as for the objective function, we arrive at

$$\Delta^{(\mathtt{jt})}(\boldsymbol{\zeta}) = +\Delta x(\boldsymbol{\zeta})/T \quad \text{(hopping)},$$

where T is some total evaluation window. Notice that this expression is nothing but the average velocity over this time window, which we are trying to maximize. The evaluation window is a purely heuristic choice. For instance, we do

```
total_T = max(10 * jump_T, 10.0)
```

where `jump_T` the sum of the duration of both one force impulse and one idle state, i.e., T is either 10 times this duration or 10.0 seconds, whichever is larger. In terms of constraints, we again employ penalty-based relaxation but we make both the thresholds and penalties more strict.[10] This is due to bad performance when we use the initial penalty parameters. Otherwise, we keep the hopping optimization fair by again using 10 different seeds, with 40 trials per seed.

Altogether, the optimized hopping controller performs in a rather underwhelming manner. While the jumps became quicker and overall it seems like more distance was covered in a shorter time window, the force profiles found by the optimization algorithm make the quadruped stumble every few steps, slightly altering its trajectory, until a large amount of drift is present. This is happening regardless of how strict we make the constraints. As a matter of fact, we have been going back and forth on this problem for a long time, changing the penalties, decision variables bounds, etc., only to come away somewhat empty handed, concluding the initial forward jumping performs better over time, yet it is not quite as fast. For one, we again attribute this fact to suboptimal controller design, which is a topic we cover more in §2. On the other hand, we feel the controller is by all means the best compromise for all jump types. This ultimately means the optimization formulation for the jumping controller could have been more than just an extension to the furthest jumps problem, even if that only meant the introduction of more constraints on top of the existing ones. Yet, due to us being constrained by time, such avenues couldn't have been explored, making this the best hopping controller (after optimization) we were able to produce.

3.4 Discussion

Going from simulation to hardware, several physical considerations must be addressed. No model is perfect, e.g., the robot's real mass distribution and joint friction differ from simulation, real actuators have backlash, etc. Additionally, real sensors introduce noise and latency to the system, resulting in delays in the control. Therefore, on hardware, continuous trial and error optimization isn't feasible because of wear and risk. Instead, we could learn from simulation, and slightly tune with strong safety margins.

When designing a walking controller, we think body osicilattions during locomotion will be one of the main issues. That is due to the introduction of leg phasing, contact scheduling, etc., which naturally requires more coordination. As we have previously mentioned, we suggest—in contrast to jumping—VMC might need to be employed more heavily.

4 Conclusion

To not prolong this report further, we think every aspect of the project received its proper conclusion in its respective portion. Yet, to summarize our exploration of quadruped jumping, we can qualitatively conclude three main things.

[10] Specific values can be found in the source code.

The first main takeaway is that finding one set of universal parameters (gains) for an omnidirectional controller is difficult to do by hand. This statement brings us to the two remaining points. For one, some jump types therefore have to suffer (lateral), even when considering the easier controller case. Secondly, the aforementioned fact also carries over to optimization, as lateral jumping was by far the most difficult furthest jump to optimize, as well as hopping for that matter. On top of this qualitative conclusion, we provide a quantitative overview of the achieved results in the appendix.

References

[1] L. Gevers and A. Ijspeert. Design and optimization of a quadruped jumping controller. Mini-Project 1 Assignment, ME-412, EPFL, October 2025.

A Selected Results

At last, let us move to the quantitative results showcase. Note that we show only the most important results of our work. In fact, running the code through the aforementioned CLI commands produces a larger number of plots compared to what is shown here. Additionally, we take the liberty to show one figure group per page in order to make the viewing experience easier and the text within that specific figure more legible. All figures which we reference in the following text are to be found at the end of the report.

A.1 Controller

Starting from the controller results, we choose to include the most basic jump type, i.e., forward jumping. All used parameters for achieving this behaviour can be found in the `JUMP_PARAMS` dictionary code snippet in §2. The time evolution of the selected force magnitudes can be seen in Figure 1. Additionally, we include the comparison between VMC off and VMC on. Despite the introduction of drift we have discussed earlier, we can notice VMC helps reduce roll deviations over multiple jumps, while also slightly reducing pitch.

A.2 Optimization

Showing optimization results requires more detail. For such, we choose to show the most important graphics for every optimized jump type. Specifically, Table 1 shows all of the most important score, i.e., objective function, values across the optimized jump types. Note that we have condensed the lateral and twist jump types from four to two by selecting `lateral_right` as for lateral and `twist_ccw` as for twist jumping.

		Score	
Jump	Best	Mean	$\pm 1\sigma$
forward	1.027	0.854	0.130
lateral	0.330	0.290	0.023
twist	3.134	2.971	0.162
hopping	1.405	0.793	0.397

Table 1: Jumping optimization results. Lateral jumping and twist jumping are represented the by right and counter-clockwise jump directions, respectively.

The provided values are visualized in form of plots in Figures 2—5. Additionally, for forward and lateral jumping, we add the landing positions for all trials to better illustrate the behaviour. As can be seen from the results, and as we have discussed before, twist jumping wins overall in terms of optimization improvement. While results may show hopping to come in second place, we have already stated it does not perform as optimally. Rather, we would place forward jumping in second place, with lateral jumping and hopping coming joint third.

B AI Declaration

Hereby, we declare that generative AI was used for this project, yet we tried to keep the use of it limited. The two main aspects in which generative AI was used were code refactoring and sentence rephrasing, which we shall now explain.

B.1 Code Refactoring

Starting with code refactoring, we feel like this is the most obvious and most helpful use of generative AI for a project like this. As we have rebuilt the code from the ground up to suit our needs, the code got more intuitive for us on one hand, but on the other hand, it also got a lot bigger and harder to navigate. Because refactoring code is a tedious task for all software developers, using generative AI for such task really improved our productivity and allowed us to explore ideas more quickly, while keeping the functionality of the code we have written unchanged. It is because of the aforementioned reasons we take the liberty to say that generative AI was helpful in this case, even if we had to come in and refactor some code by hand.

Sample Prompt

Could you refactor this code block so it is more human readable and there is a clear separation of concerns while keeping the functionality exactly as is?

`<Some code>`

Sample Answer

Here's a cleaner, more structured version that preserves behavior exactly while clarifying responsibilities and general flow:

`<Some refactored code>`

B.2 Sentence Rephrasing

Getting our ideas across in text hasn't always been as smooth as we would've liked. At some points during work, we have stumbled upon reusing the same words multiple times in a single sentence, making it overcomplicated, and other pitfalls present in academic writing. To get to the point in a more concise way or to make some sentences more understandable to the reader, we have used generative AI on some occasions. Yet, we felt like, more often than not, it made sentences even more complex, and potentially misleading, hence why we have used the responses only as inspiration.

Sample Prompt

Could you rephrase this sentence to make it flow more naturally?

`<Some sentence>`

Sample Answer

Here are a few polished alternatives with slightly different tones:

`<Some alternatives>`

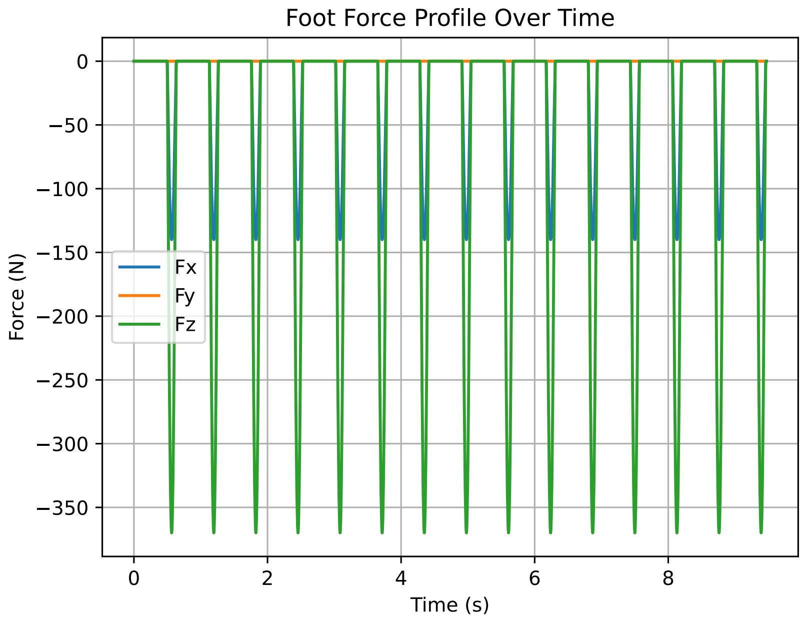
B.3 Other Use Cases

When it comes to other use cases of generative AI in this work, we have used it for a very small number of tasks besides the ones we have already mentioned, e.g., fixing minor bugs (good) and suggesting typical variable values (suboptimal).

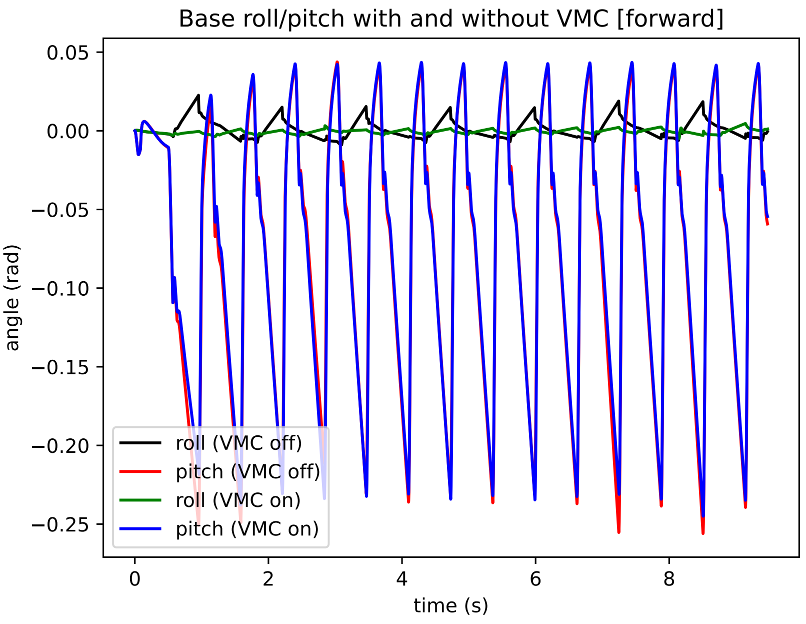
B.4 Summary

Overall, we are proud to state that, while AI was used, the majority of the code and the entirety of this report were written by hand. We have distanced ourselves from copy-pasting code or sentence suggestions as much as possible, hence why we think the learning experience has stayed the same as if no AI was used at all. This also comes down to the fact that AI was mostly used for repetitive tasks, i.e., code refactoring and sentence rephrasing. Finally, AI has made the project nor easier, nor harder to complete. We think it made it more enjoyable as one could spend less time with repetitive, tedious tasks, and focus on the core concepts.

What follows are figures referenced in Appendix A.

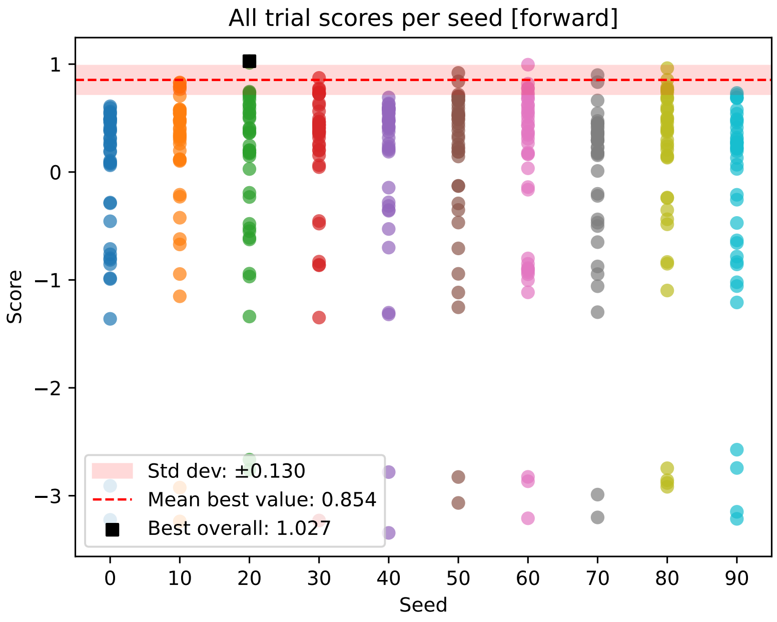


(a) Foot force profiles.

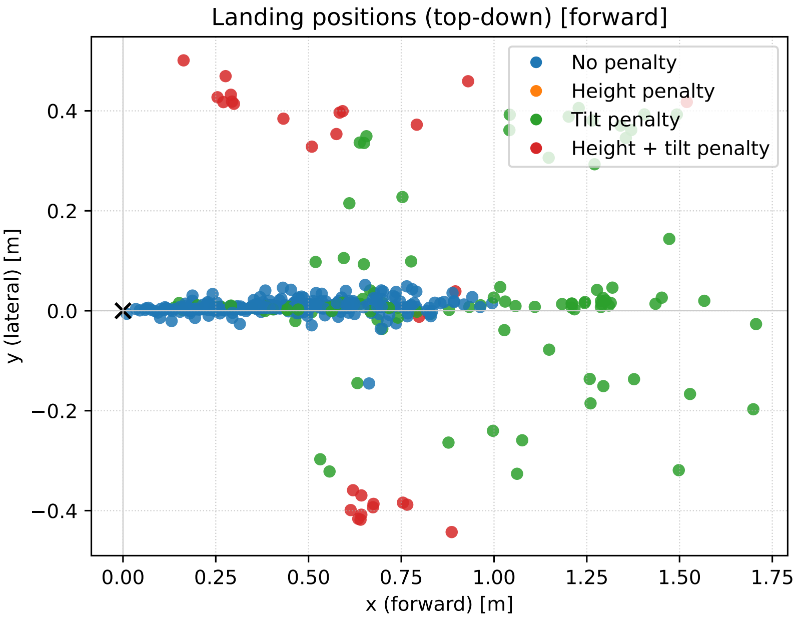


(b) Effect of VMC on base roll/pitch. `K_VMC` set to `20.0`.

Figure 1: Forward jumping controller results.

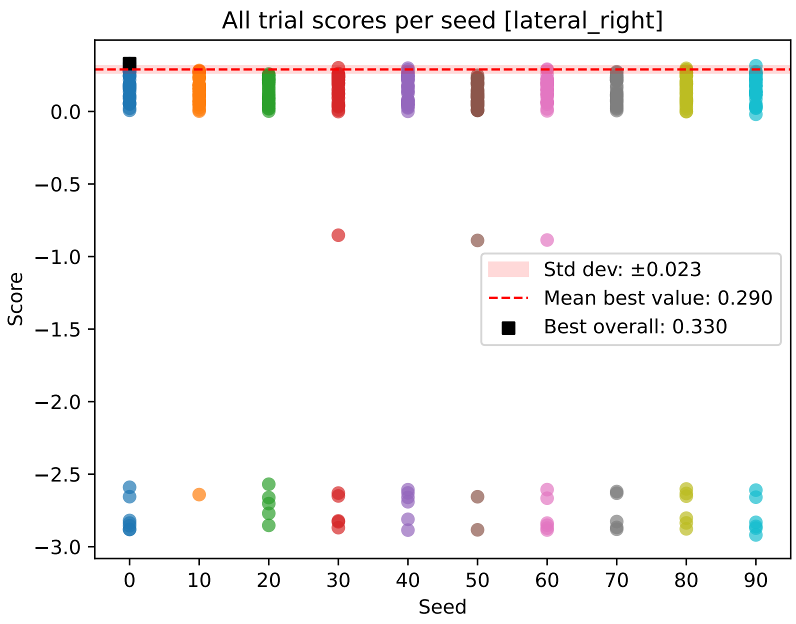


(a) All trial scores per seed.

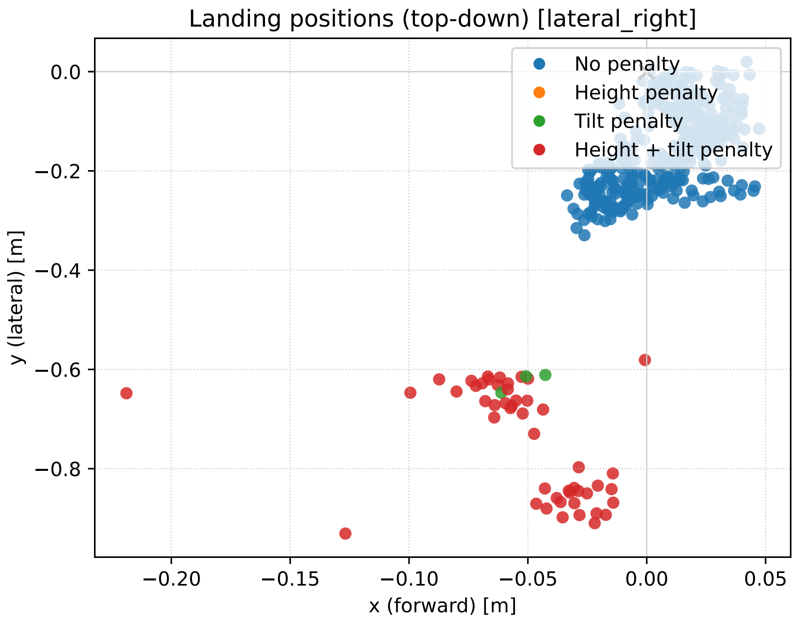


(b) Landing positions for all trials.

Figure 2: Forward jumping optimization results.



(a) All trial scores per seed.



(b) Landing positions for all trials.

Figure 3: Lateral jumping optimization results.

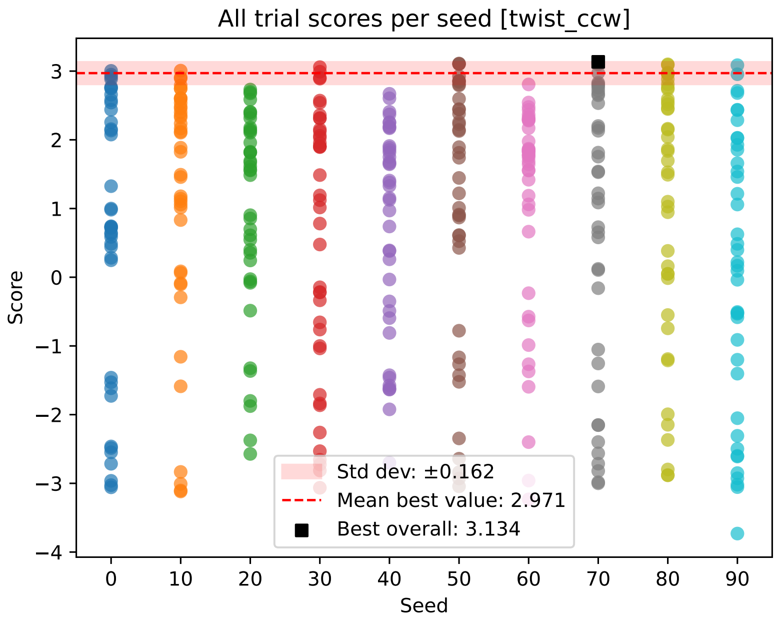


Figure 4: Twist jumping optimization results, all trial scores per seed.

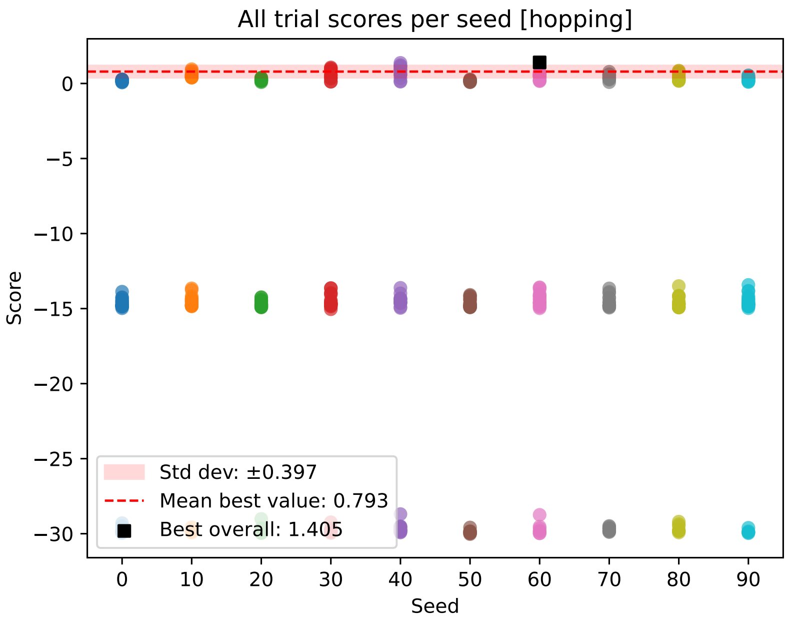


Figure 5: Hopping optimization results, all trial scores per seed.