Quadruped Trajectory Optimization

Adarsh Kulkarni and Austin K. Chen

***Abstract*—Quadrupedal robots have become a notable area of study in recent years, as they have demonstrated significant potential across a wide variety of domains and tasks, like navigation over unstructured terrain. As such, there exist control strategies which can plan different kinds of gaits for such robots, and also recover from external disturbances. However, such control policies are typically reactive. In this work, we consider how quadruped robots may use a priori knowledge about the environment; specifically, we incorporate a height map around the leg of the robot. Using this height map, we parameterize obstacles and generate an offline collision-free swing trajectory. Finally, we present preliminary results on learning a neural-network based controller which imitates the output of the obstacle-aware trajectory optimization, towards being able to run the controller in real time.**

***Index Terms*—Robotics, Quadruped, Trajectory Optimization, Machine Learning**

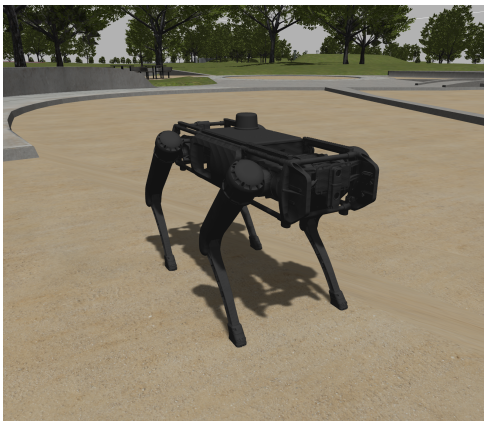


Fig. 1. Model of Ghost Robotics Vision 60 robot.

I. Introduction

QUADRUPEDAL robots have been widely studied in recent years, and the development of control strategies for such platforms has expanded accordingly. With a wide variety of applications across civilian and military fields, these robots have garnered significant attention from research institutions, industry, and the media. Of particular note is their ability to traverse irregular terrain. In contrast to wheeled robots, quadruped robots can often handle much more challenging environments. When compared with flying robots like quadrotors, quadruped robots can carry higher payload weights and operate for longer periods of time between battery charges.

Many robust behaviors have already been demonstrated on quadruped robots, and it is not difficult to find demonstrations of such robots recovering from a human-generated disturbance (e.g. a kick) [1]. However, such policies are typically reactive, and do not consider actively planning leg trajectories around obstacles in real time. Though many kinds of gaits already exist for the quadruped robot, in this work we seek to leverage a priori knowledge of the environment to generate leg trajectories on the fly.

We use a direct collocation approach in our trajectory optimization, where we solve for the position of the relevant knot points and the inputs for a single leg of the robot. Our cost is taken to be the distance from a reference trajectory along with the distance from the goal state and apex (highest point of step) states. Obstacle avoidance is encoded in the constraints, along with dynamic constraints and physical constraints (e.g. staying above ground).

We perform these experiments in the realm of single-leg optimization, and maintain that a full quadrupedal gait trajectory optimization pipeline could be a straightforward opportunity for future work using the single leg framework we propose in this paper.

This report is structured as follows: in Section II we detail the trajectory optimization problem that is solved, along with how obstacles are modeled and collisions are avoided. In Section III, we show how obstacles are automatically generated in accordance with a given height map. In Section IV we present an approach for learning a neural network that produces a trajectory given a heightmap, and in Section V we present our results and discussion. All experiments are performed on a model of the Ghost Robotics Vision 60 platform (Fig. 1) with the public URDF files from [1].

II. Trajectory Optimization

We use the direct collocation method for our trajectory optimization, where we optimize over the states x and control inputs u. All programs were solved with the `SNOPT` solver inside `Drake`. We take our state to be

$$x = \begin{bmatrix} \theta \\ \omega \end{bmatrix} \tag{1}$$

where θ and ω are the angles and angular velocities of the three leg joints, respectively. In total, since there are three leg joints, $x \in \mathbb{R}^6$.

Our input u follows a similar formulation, which is

$$u = \tau \tag{2}$$

where τ is the torque commanded at each of the three joints. Therefore, $u \in \mathbb{R}^3$.

A. Kulkarni and A.K. Chen are with the GRASP Lab at the University of Pennsylvania, Philadelphia PA 19104.

Emails: {akulkarn, akchen}@seas.upenn.edu

[1] See https://twitter.com/Ghost_Robotics/status/1285388598976618502

A. Program Constraints

First, we enforce a constraint on the initial state, making it equal to the initial position. Hard constraints are not enforced on the goal state (the targeted end of the trajectory) and the apex state (the highest point of the reference trajectory) in case obstacles make the constraints infeasible. The reason for creating a target apex height (naively achieved at a swing phase of 0.5) is to maintain some level of consistency with the blind, reactive gaits. While an opportunity for future work, we hypothesize that this can smooth out transitions between optimized and reactive gaits during locomotion.

Constraints enforcing joint limits and angular velocity limits are also enforced as bounding box constraints. We also impose a bounding box constraint on the efforts exerted by the actuators. To ensure that the leg stays above ground for the entire trajectory, we also calculate the position of the toe as a function of the state and enforce that it remains above the ground.

It should be noted that for the sake of this work, the actuators are simplified to be direct drive. Future work that aims to deploy a system such as this on a real platform would benefit from accounting for actuator and drivetrain characteristics.

Finally, we add constraints for obstacle avoidance. We model obstacles as spheres, and assume that the trajectory is collision-free if the minimum distance from the leg to the obstacle is always greater than the obstacle's radius combined with the leg's approximate collision radius. To do this, we model the link as a line and the obstacle as a point. Consider the endpoints p_{up} and p_{low} that define the position of the upper and lower parts of the link respectively in world coordinates (e.g. the knee and toe positions). Then, the link vector s_{link} can be defined as

$$s_{link} = p_{low} - p_{up} \tag{3}$$

and the distance from the upper link to the obstacle s_{obs} defined as

$$s_{obs} = p_{obs} - p_{up} \tag{4}$$

where p_{obs} is the position of the obstacle in world coordinates. Then, we may subtract the project of s_{obs} onto s_{link} from s_{obs} to get the perpendicular distance between the obstacle and link, provided the projection lies on the line segment itself. If the projection lies outside the line segment, the actual distance may be greater, so the constraint is more restrictive but still guarantees collision avoidance. Thus, the distance d may be defined as

$$d = s_{obs} - \left(\frac{s_{link}}{\|s_{link}\|_2}\right)\left(s_{obs} \cdot \frac{s_{link}}{\|s_{link}\|_2}\right) \tag{5}$$

where $(\cdot)$ and $\|x\|_2$ denote the dot product and L^2 norm, respectively. To ensure collision avoidance, we enforce the constraint that $d \geq r_{obs} + r_{link}$, where r_{obs} is the radius of the obstacle and r_{link} is the radius of the link. This process is repeated for both the upper link and the lower link.

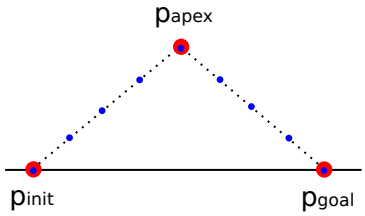


Fig. 2. Diagram of reference trajectory generation. Initial guesses for direct collocation (blue circles) are linearly interpolated between the initial point, apex point, and goal points (red circles).

B. Cost Function and Reference Trajectory

For our cost function, we impose a quadratic cost $\mathcal{C}$

$$\mathcal{C} = (x - \hat{x})^T I (x - \hat{x}) \tag{6}$$

where $\hat{x}$ is the reference trajectory. In other words, the cost is the deviation from the reference trajectory, which is a linear interpolation between the initial, apex, and goal points (specified as parameters). Half of the N collocation knot points interpolate from the initial to the apex point, and the remaining half interpolate from the apex point to the goal point. A visual depiction of the reference trajectory is shown in Figure 2. We set this reference trajectory as our initial guess when we are either not using an iterative refinement strategy, or in the first step of our iterative refinement strategy. It should be noted that a much smoother, polynomial spline could be used as the reference trajectory, but we hypothesized that the change in output trajectory would be minimal given the use of the reference in the cost term and not in the constraints.

C. Runtime and Iterative Refinement

The runtime to solve our program varies drastically based on the number of obstacles and the number of knot points. We used a Ryzen 5 3600 and a Threadripper 3970x as our primary compute. For a simple feasible case with less than 10 obstacles and 15 knot points, the program generates a trajectory within a few seconds. However, when we scale to over 100 obstacles with 35 knot points, the solves can take upwards of 30 minutes. It should be noted that this scale of obstacles is only achieved when the heightmap discretization strategy, described in the next section is used.

To alleviate the issue of long runtimes, we implemented an iterative refinement strategy. We would first solve the program with all obstacles and constraints, but with fewer knot points. Once this program was solved, we would then use the generated trajectory as an initial guess to another program with the final number of desired knot points, and with the cost function removed (turning it into a feasibility problem). Empirically, we found that to avoid obstacles in between knot points, we required $N = 35$. Our final pipeline first solves using $N = 15$, and then uses the calculated solution as an initial guess for $N = 35$. While improvement factors vary and we haven't conducted a full analysis, qualitatively we

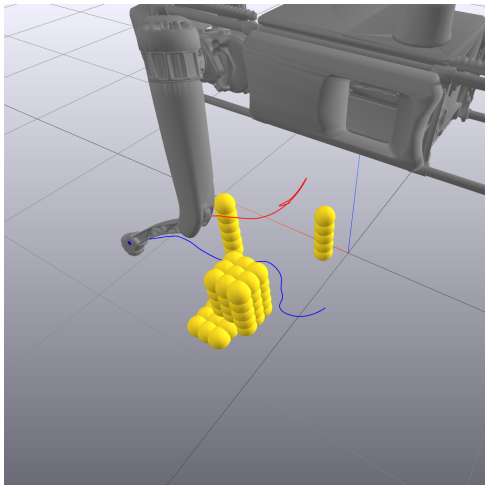


Fig. 3. Example of obstacle generation from heightmap (yellow spheres). Blue and red lines denote the planned trajectories for the toe and knee, respectively.

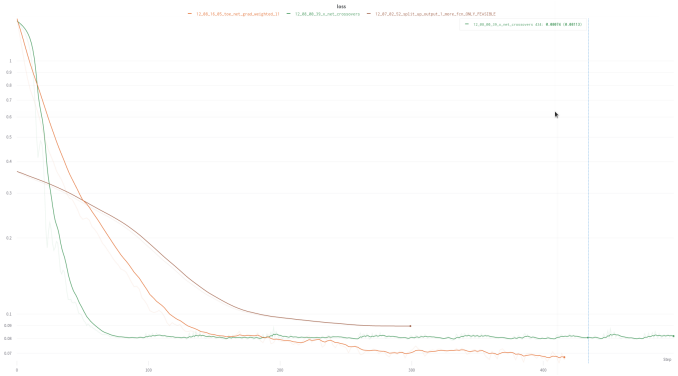


Fig. 4. Loss curves for the highest performing models of each architecture type: effort, joint angles and toe location.

generally see at least a 3x speedup when compared to outright solving for $N = 35$.

III. Heightmap Integration

In this section, we describe the automatic creation of obstacles from a heightmap of the terrain surrounding the robot leg. Such data is assumed to be available before the trajectory is planned, and may come from different types of sensors, e.g. depth cameras, sonar, or lidar.

We assume that the heightmap is given as a 2D array, and covers an area of .5m by .5m centered around the shoulder joint of the leg. The resolution is also given as a parameter; higher resolution will result in a better discretization, but longer solve times when the trajectory is generated through the process in Sec. II. For our tests, we consider each discretized cell to range from 2cm square to 4cm square.

Once the heightmap is received, obstacles are generated for each grid cell in the heightmap. We stack obstacles with radius equal to the size of the grid cell until the height specified at that cell is reached. Once all the obstacles are generated, we can then proceed with the trajectory optimization from Section II to plan an obstacle-avoiding trajectory. An example is shown in Figure 3.

This is a critical aspect of our pipeline, since it results in a perception agnostic system that doesn't require complex obstacle recognition upstream. Many trajectory optimization pipelines rely on more high fidelity obstacle representations, but this in turn results in significant difficulty for the perception stack. Additionally, this 2D representation lends itself quite well to deep learning.

It should be noted that there are still situations where the optimization returns infeasible. This is desired, as qualitatively, we can verify that the given obstacle space is in fact infeasible given the kinematic and workspace constraints of the robot's leg.

IV. Learned Controller

Once we had a working optimization pipeline for offline obstacle avoidance, we build out a data creation pipeline that randomly generates obstacles in the leg's workspace, and then proceeds to solve the optimization problem in that environment. We also have the ability to randomly generate different start, end and apex points and solve for that case as well.

The structure of our network is relatively simple. It starts with 2 convolutional layers on the heightmap, followed by a flatten and linear layers that output the trajectory in different forms. At this time our network architectures only support a constant start, end and apex, but we detail some steps that could be taken to create a system that is able to perform on varied values for those parameters.

In particular, we ran numerous experiments on different parameterizations of the output trajectory. Initially, we attempted to learn a direct mapping of effort values, or U, from the heightmap. This lead to a large output vector of length Nxu where N is the number of knot points and u is the dimension of the effort variables. In our case, this was 35 knot points and 3 effort values per knot point.

In general, the output of this network was relatively poor, and while it learned general trends in the trajectories, such as a veer to one side, it was unable to learn the critically important "kinks" and swings in the trajectory that are necessary to avoid obstacles. We continued experiments by instead learning our x variable, or joint angles, at each nought point. We also switched to a branched network that has 3 distinct fully connected branches each of which predicts all the nought points for one of the output variables. Put another way, when outputting 3 joint angles, we implement 3 downstream fully connected layer sets with a couple of cross connection between the branches. This network performs better than learning the efforts directly, but is still not quite useable.

As a sanity check, we also train a "feasibility classifier" on the input obstacle heightmap. This classifier simply predicts whether there is a feasible trajectory for the leg through the given obstacle space. We currently do not tackle the problem of network behavior given an infeasible obstacle map, and train only on the samples that result in a feasible trajectory.

The final geometric parameterization we try is to learn the toe xyz locations. It should be noted that technically, for the joint angles and toe xyz configurations of output, we would also need to learn joint and toe velocities in order to actually

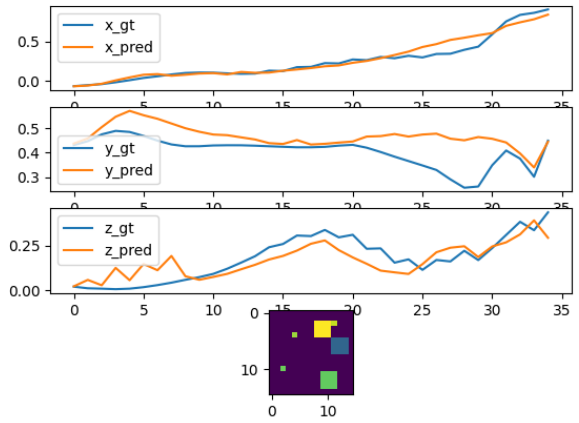


Fig. 5. Sample output from the current best performing toe xyz based network. Randomly generated obstacle map is shown at the bottom.

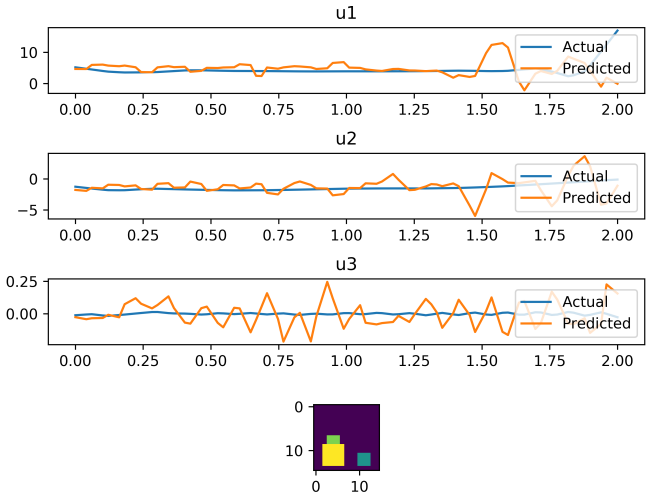


Fig. 6. Ground truth (blue) vs predicted (orange) splines for commanded torques.

compute control from state and close the loop. At this time we don't make enough progress to make that change, but it is a necessary step once the positions are learned to a sufficient degree of success.

The toe xyz models perform the best of all of the different configurations. We also make a change to the loss function for this set of experiments. Instead of a uniform L1 loss on the entire trajectory, as was used for the rest of the experiments, we upweight the L1 loss for knot points where the difference, or velocity, is high. This incentivizes the network to learn the kinks in the trajectory, while before the relatively low error in the smooth trajectory sections may have been drowning out the error in the sharp movements.

We also began a set of experiments to predict the coefficients of the splines instead of the knot points themselves. Unfortunately, due to time constraints the dataset used in this case was much smaller (only around 100 points). Besides the usage of an L2 loss, much of the structure that is mentioned previously with regards to models and training is preserved. As can be seen in Figure 6, the results are relatively poor. It is not yet clear if this is due to insufficient data, or a problem with the approach itself.

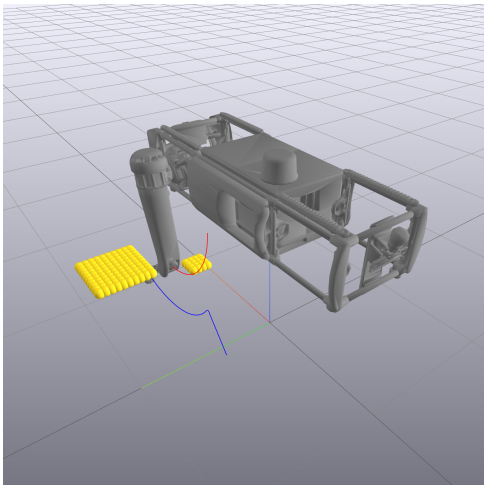
V. Results

To validate our approach, we randomly generated and manually inspected a series of test cases. As stated before, the number of knot points heavily influences the compute time and potential for collisions in between knot points. An example of a collision free trajectory is shown in Figure 7a, while an example of a colliding trajectory is shown in Figure 7b. As can be seen, the collisions observed are not usually particularly egregious, and seem to mostly occur when the trajectory slightly clips an obstacle. We discovered qualitatively that using around $N = 35$ knot points seemed to mostly resolve these issues, but at the cost of drastically increased computation time. It is because the trajectory planned with fewer knot points seems to be reasonable that we decided to implement the iterative refinement described in Section II-C, which provides feasible trajectories with substantially reduced computational cost. We also note that these collisions with lower N mostly occurred in more challenging environments, where the obstacles were placed close to the reference trajectory.

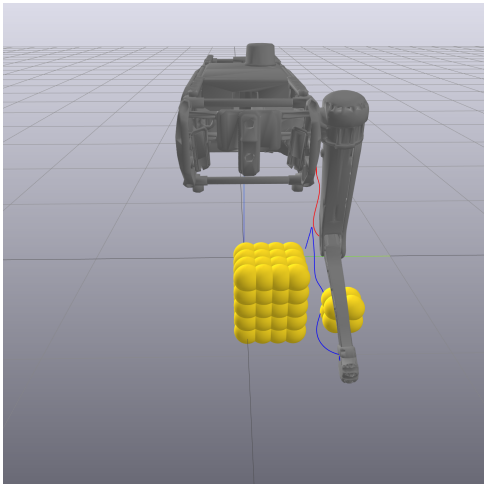
Due to our initialization guess and the presence of the cost penalizing the deviation from the reference trajectory, the result of the optimization sometimes provides trajectories that can be seen as suboptimal. For example, in Figure 8, the apex position of the toe (whose position is traced as the blue line) is very high. While the apex point may help with stepping over obstacles, it may be seen as unnecessary to raise the toe very high when there are no obstacles to step over.

Due to our lack of a penalty on control effort, the trajectory optimization will prioritize staying close to the reference trajectory, as seen in Fig. 8. This makes the guesses for the apex and goal state particularly important, and are parameters that should be tuned. Though we tried to incorporate a cost on the effort expended, we ran into issues with the arm going limp to minimize effort. This may be due to an improperly balanced cost function, which we would like to explore further in the future.

Overall, our approach performed well in terms of generating smooth collision-free trajectories. More work needs to be done to improve the runtime (for real-time planning) and trajectory initialization, but the initial results seen are promising. While we believe there is potential for runtime improvement in the optimization pipeline, we remain somewhat skeptical of possibility of compressing a 90 second optimization into ¡0.3s. This would be the maximum solve time given the nominal stepping frequency of the Ghost platform. This, along with clear CPU compute bottlenecks on board the Vision60's Nvidia Jetson Xavier, were main motivations for pursuing a learned approach that utilizes the trajectory optimization pipeline as supervision. For a network approximately the size of the architectures we were using as part of this work, it wouldn't be surprising to see 40-50 Hz inference times on the Xavier, which is much



(a) Collision free trajectory



(b) Colliding trajectory

Fig. 7. Examples of collision-free and colliding trajectories.

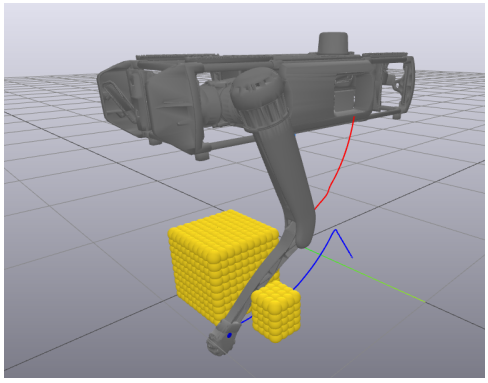


Fig. 8. Example of trajectory where the effect of the reference trajectory is particularly pronounced. Note the high position of the toe (blue line) without obstacles close by.

more useable in a realtime scenario.

VI. Conclusion

In this report, we have presented an approach to obstacle-aware trajectory optimization for quadruped robots. Through demonstrations in simulation, we have verified the efficacy of our approach, and generated a pipeline for converting heightmap data (from a depth camera or other sensor) into a series of obstacles which can then be incorporated into a trajectory optimization problem. While we were able to obtain collision-free trajectories in a wide variety of scenarios, computational complexity eventually became an issue, and for many of the tested parameters our approach is not able to run in real time. Future work could possibly address this, as we would like to explore more efficient formulations of the trajectory optimization program, as well as suitable parameters (e.g. heightmap resolution) for operation in the real world on the Vision 60. We also present initial ideas and results in addressing this realtime challenge by pursuing a learned approach to trajectory optimization that uses the nonlinear solver as supervision. We show experiments using a convolutional network that predicts multiple different parameterizations of the output trajectory. Given the heavy dependence of the optimizer on the initial guess, we would also like to explore how we can obtain better reference trajectories, and perhaps optimize over the control effort to obtain more efficient solutions. Finally, given that the optimization is performed on only a single leg, a natural extension would be to plan full stable gaits with all four legs.

Acknowledgment

The authors would like to thank Professor Michael Posa and the MEAM517 course staff for their constant support, and for putting together a fantastic course.

References

[1] G. Robotics, “Ghost description,” https://gitlab.com/ghostrobotics/ghost_description/-/tree/master, 2020.