Exploration and Comparison of Ballbot Controllers

1st Suzanna Jiwani
Electrical Engineering and Computer Science
Massachusetts Institute of Technology
Cambridge, USA
sjiwani@mit.edu

2nd Jorge Nin
Mechanical Engineering
Massachusetts Institute of Technology
Cambridge, USA
jorgenin@mit.edu

3rd Fischer Moseley
Physics
Massachusetts Institute of Technology
Cambridge, USA
fischerm@mit.edu

***Abstract*—We describe the ballbot problem and compare the relative performance of four different controllers: LQR, a neural network that approximates a cost-to-go function, trajectory optimization, and model predictive control (MPC). We use a standardized cost function to compare their performance, but also evaluate their behavior under permutations of model parameters. Finally, a qualitative analysis of the controllers is provided, and barries to practical application discussed.**

***Index Terms*—robotics, machine learning, nonlinear control theory, trajectory optimization, dynamic programming, robot kinematics**

I. Introduction

A ballbot is a type of robot with that balances on top of a ball, and moves by tilting its center of mass in some direction. Typically this is accomplished with either a set of omniwheels or an inverse mouse-ball drive, and to maintain controllability of the robot it is necessary to have the center of mass stay nearly centered on the ball. With this in mind, a controller to maintain the ballbot's position at the unstable fixed point is critical for operation, and poses an interesting dynamics problem that we solve here.

Although a controller is needed to stabilize a ballbot, they have a considerably smaller footprint than traditional wheeled robots, and have additional degrees of freedom in the tilt angles. These extra degrees of freedom make the ballbot trivially underactuated, and nontrivial to control. Given the exploratory nature of this project, we chose to limit our analysis to two-dimensional ballbots, where the robot on the ball has a degrees of freedom in both the x and θ directions, where θ indicates the angle of the robot from vertical, as diagrammed in. This is diagrammed in figure 2.

In order to control the ballbot, we developed four methods of control:

- A LQR controller utilizing linear feedback. This method is the most widely used across ballbots, and its exponential stability at the fixed point prompted us to use it as a baseline to reference the other controllers against.
- A dynamic programming based approach, using a neural network to approximate a cost-to-go function for any start and goal state. From this an optimal policy is easily obtained.
- A trajectory optimizer formulated as a nonlinear program with variable constraints, using Drake's `MathematicalProgram` wrapper for the SNOPT solver.
- A Model Predictive Controller, using the previously described finite-horizon trajectory optimization for motion planning.

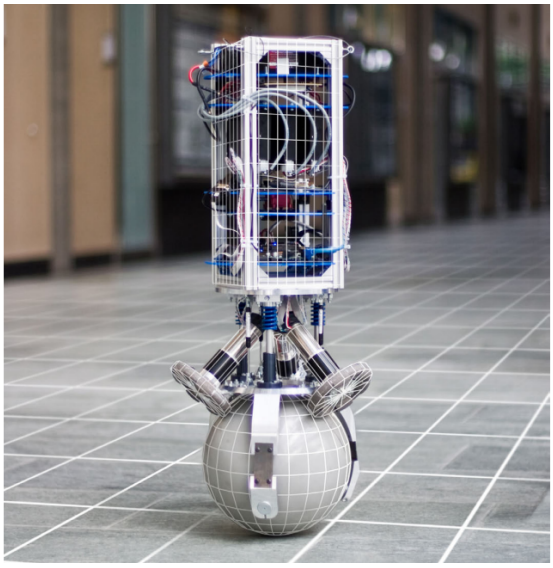


Fig. 1: Rezero, a ballbot built by students at ETH Zürich in 2010. The position is stabilized via a LQR controller. [6]

To evaluate the robustness of each controller, we varied the initial conditions and constraints to see under what conditions our controllers would be effective. Finally, we compared how well the four controllers adjusted to pertubations in model parameters, including changes in mass, inertia, and size.

II. Dynamics and Modelling

The ballbot exists in two-dimensional space and cannot leave the ground (ie. it must stay on the ground and has no ability to increase height), therefore only its distance along the x-axis and its tilt angle are necessary to fully describe its position. The resulting state space follows:

$$\boldsymbol{q}(t) = [x(t), \theta(t)]^\top \tag{1}$$

$$\dot{\boldsymbol{q}}(t) = [\dot{x}(t), \dot{\theta}(t)]^\top \tag{2}$$

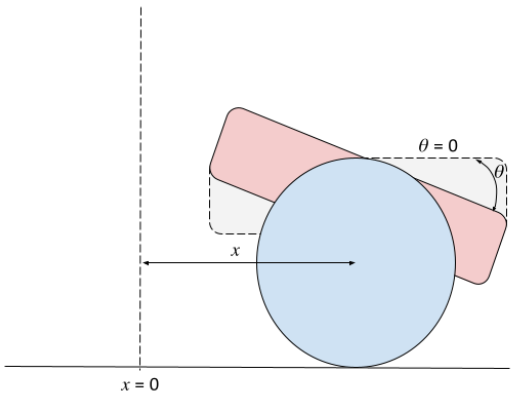


Fig. 2: A sketch of the model indicating the physical meaning of the state (x, θ).

The dynamics of the ballbot take the standard manipulator equation form:

$$\boldsymbol{M}(\boldsymbol{q})\ddot{\boldsymbol{q}} + \boldsymbol{C}(\boldsymbol{q}, \dot{\boldsymbol{q}})\dot{\boldsymbol{q}} = \tau_g(\boldsymbol{q}) + \boldsymbol{B}\boldsymbol{u} \tag{3}$$

Umashankar et al. analytically determined the system dynamics via Euler-Lagrange, and after transforming them into our coordinate system we find that they are:

$$\boldsymbol{M}(\boldsymbol{q}) = \begin{bmatrix} \frac{\alpha}{r} & 2\alpha + \beta\cos\theta \\ \frac{\alpha + \beta\cos\theta}{r} & 2\alpha + \gamma + 3\beta\cos\theta \end{bmatrix} \tag{4}$$

$$\boldsymbol{C}(\boldsymbol{q}, \dot{\boldsymbol{q}}) = \begin{bmatrix} -\beta\sin\theta\dot{\theta}^2 \\ -\beta\sin\theta\dot{\theta}^2 \end{bmatrix} \tag{5}$$

$$\boldsymbol{G}(\boldsymbol{q}) = \begin{bmatrix} 0 \\ -\frac{\beta g \sin\theta}{r} \end{bmatrix} \tag{6}$$

where α, β, and γ are constants given by the mass and rotational inertia of the ball. [3]

A. Representing the Ballbot as an URDF file

As we chose to use Drake for simulating the proposed controllers, we chose to represent the ballbot in the Unified Robot Description Format (URDF) to be easily imported. In doing so we model the ballbot as a rectangular base constrained to ride atop circular ball. The ball and the base are able to rotate freely, although for ease of simulation two actuators are modelled instead of one. These actuators provide force along the x-axis and torque around the θ-axis, however the gear reduction specified in the URDF file allows the two to be set equal to each other and treated as one combined actuator. This allows for the single actuator present in a physical system to be more easily represented, but means that the actuator vector $\boldsymbol{u}$ is technically two-dimensional. The exact URDF used is available in the source code on GitHub, and is based on a textbook example. [2]

Drake allows us to inspect the symbolic form of the dynamics after specifying the robot geometry and actuators in the URDF file. The results of this are shown below, and are of similar form to the analytic derivation done by Umashankar et al:

$$\boldsymbol{M}(\boldsymbol{q}) = \begin{bmatrix} 9 & 0.2\cos\theta \\ 0.2\cos\theta & 0.19 \end{bmatrix} \tag{7}$$

$$\boldsymbol{C}(\boldsymbol{q}, \dot{\boldsymbol{q}}) = \begin{bmatrix} -0.2\dot{\theta}^2\sin\theta & 0 \end{bmatrix} \tag{8}$$

$$\tau_g(\boldsymbol{q}) = \begin{bmatrix} 0 & 1.96\sin\theta \end{bmatrix} \tag{9}$$

$$\boldsymbol{B} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \tag{10}$$

$$\tau_{ext} = \begin{bmatrix} 0.1\dot{x} \\ 0.1\dot{\theta} \end{bmatrix} \tag{11}$$

Something to note is that the two equations of motion are not the same. There is a missing Coriolis effect and the mass matrix are not exactly the same. This is because the URDF assumes the ball isn't actually rotating and instead just slides along the floor. Because of the small moment of inertia of the ball however this won't really affect the results much. Also because we only care about the difference in controllers and not implementing them on a real life system, it doesn't matter if our dynamics are slightly off as long as they are consistent.

III. Controller Description

A. Linear Quadratic Regulator (LQR)

Given LQR's popularity in ballbot literature and relatively straightforward implementation, it was chosen as a benchmark to compare the remaining controllers against. The unconstrained nature of LQR also provides a natural point of comparison for trajectory optimization. Assuming both have the same cost function, a trajectory optimization problem formulated with nonlinear programming should yield the LQR solution in the unconstrained case. To that end, we provide the following cost function to LQR:

$$J = \int_0^\infty \left(\boldsymbol{q} \begin{bmatrix} 10 & 0 \\ 0 & 10 \end{bmatrix} \boldsymbol{q}^\top + \dot{\boldsymbol{q}} \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \dot{\boldsymbol{q}}^\top + \boldsymbol{u}^\top \boldsymbol{u} \right) dt \tag{12}$$

LQR relies on a linearization of the plant dynamics about a fixed point to determine the optimal policy. One issue that arises is when the plant is nonlinear, the difference between the linearized and truly nonlinear dynamics can diverge to the point where the LQR controller fails to stabilize the plant. In a ballbot this may happen at large tilt angles as the $\sin$ and $\cos$ terms diverge farther from their linear approximations, or this divergence may occur in actuator limits or friction.

In our case we have kept to a simple model with no friction or actuator limits, so the only nonlinearities are the $\sin$ and $\cos$ terms in the manipulator equations, as well as the nonzero Coriolis term. All of these nonlinearities are soft and do not rapidly diverge from their linear approximations, and thus

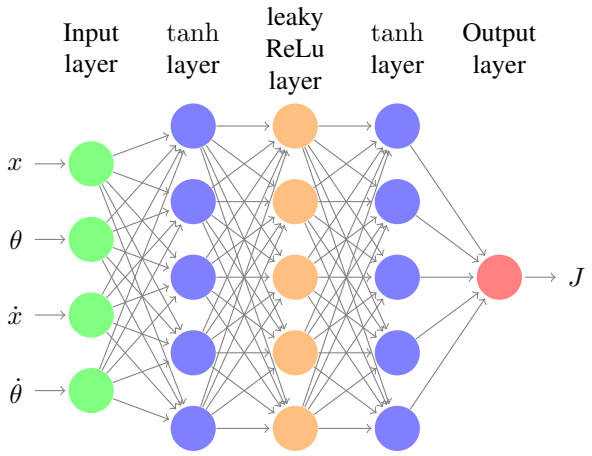


Fig. 3: The architecture of the neural network used to approximate a cost-to-go function. The input layer is four neurons wide, the first tanh layer is 80 neurons wide, the leaky ReLu layer is 80 layers wide, and the second tanh layer is 50 neurons wide. All layers are dense and fully connected.

should be reasonably approximated with a linearization of the plant.

B. Neural Network to Approximate Cost-To-Go

In an effort to evaluate the suitability of dynamic programming for controlling the ballbot, a cost-to-go function was developed using the architecture shown in Figure 3. The network was trained over a randomly selected subset of the state space bounded by $\{q, \dot{q}\} \in [-3, 3]$ using the same cost function passed used for LQR in Equation 12. The action space U was discritized between $\{u\} \in [-4, 4]$ with variable spacing. From each one of the starting states, every possible action was taken. The transition cost of the function was added to the predicted cost to go then the minimum of that cost to go was taken as the real cost to go. This value was then used as our loss function for the neural network. From the cost to go function that was derived we could at any state predict the new cost to go from taking any action by using the model of the dynamics to predict the next state and inputting that into the neural net. We could then select the action that would minimize the cost to go of our next state. It was hoped that through a generic network we would be able to find an optimal policy similar to LQR, but further exploit the nonlinear dynamics and provide more intelligent movement. The results of this learned cost-to-go function are shown in Figure 6.

C. Trajectory Optimization

In addition to the dynamic programming approach accomplished by the neural network, a method of control utilizing trajectory optimization was also implemented. Rather than trying to solve for the optimal controller over some subset of state space, we instead found an optimal trajectory through state space that is valid only from a single initial condition. Instead of representing this as a function, we can represent this solution as a trajectory $x(t), u(t)$ defined over a finite interval.

We formulate the trajectory optimization of the ballbot as a direct collocation problem, and construct the optimization using Drake's generic `MathematicalProgram`, which we solve with SNOPT. We constructed the problem to occur over $T = 50$ discrete points, for which we defined the following decision variables:

- $\boldsymbol{h}$, a vector of length T of the time intervals between points in the trajectory (ie, the physical time between the t^{th} and $(t+1)^{\text{th}}$ points)
- $\boldsymbol{q}$, a $(T+1) \times 2$ vector of generalized positions (see equation 1)
- $\dot{\boldsymbol{q}}$, a $(T+1) \times 2$ vector of generalized velocities (see equation 2)
- $\ddot{\boldsymbol{q}}$, a $(T+1) \times 2$ vector of generalized accelerations (see equation 3)
- $\boldsymbol{u}$, a $(T+1) \times 2$ vector of actuations (see equation 3)

When creating the program, a few necessary constraints were enforced to ensure the solution was physical:

- Bounding the time intervals $\boldsymbol{h}[t]$ between points to be within a given h_{min} and h_{max}, as required by the direct collocation formulation.
- Enforcing the system dynamics given in section II. This is done by ensuring that the manipulator equations are met at all points along the trajectory.

Although these constraints are necessary to properly formulate the problem, they do not provide any additional constraints that would not be present in a LQR optimization. The additional constraints that make a nonlinear programming approach appealing include limits on actuation and tilt angle - perhaps to represent the real-life constraint where the ballbot would drop something balancing atop it if it tilts too far. It is also possible to force the optimizer to bring the ballbot to the goal state at $t = t_{end}$, but in our experience we noticed this can often yield highly nonphysical trajectories.

While adding a cost function to the solver is not necessary to find a solution, it does provide a convenient way to communicate our intent to the optimizer. Some costs provided to the optimizer included:

- A linear distance cost, which linearly penalized the ballbot's distance from the goal state as $S = |x - x_d| + |\theta - \theta_d|$. In practice, we had significant difficulty in getting the optimizer to find a solution, which is to be expected as the laplacian of the cost ($\nabla^2 S$) is zero.
- A quadratic distance cost, which penalized the squared distance from the goal as $S = (x - x_d)^2 + (\theta - \theta_d)^2$. This didn't exhibit the same convergence failures as the linear distance cost, and produced physical trajectories.
- A quadratic cost on distance, velocity, and actuation. For the ease of comparison, this was chosen to be equal to the integrand of the cost function used in LQR and described in Equation 12. This was set as $S = 10(x - x_d)^2 + 10(\theta - \theta_d)^2 + (\dot{x} - \dot{x}_d)^2 + (\dot{\theta} - \dot{\theta}_d)^2 + u^2$. Using this cost also allowed us to compare our trajectory to the LQR trajectory. In theory this trajectory should be more be

more "optimal" than the LQR trajectory as it will not include linerization errors.

D. Model Predictive Control

Despite the flexibility of trajectory optimization in motion planning, the output of the algorithim is a trajectory, not a controller. While it is possible to directly feed the optimal actuator inputs $\boldsymbol{u}$ from the trajectory into the ballbot dynamics, the lack of feedback results in the ballbot failing to follow the optimal trajectory.

Although it was considered to use LQR to stabilize the ballbot around an optimal trajectory, we instead chose to investigate MPC. At every timestep, this controller computes an optimal trajectory and then commands the first actuation from the optimal $\boldsymbol{u}$ vector. The dynamics of the system then evolve the ballbot's state to the next timestep, at which point the process repeats. This process provides a natural extension of the motion planning algorithm used in trajectory optimization, and repackages it into a closed-loop controller. This stabilizes the ballbot against small perturbations around the optimal trajectory, but in the event of large disturbances, an entirely new optimal trajectory will be computed.

IV. Controller Performance

To evaluate the performance of each controller, we adopted running and total costs as metrics. We used LQR as a baseline, and were able to see how the different controllers attempted to minimize the cost of their trajectories with the same cost function as LQR. If any controller could drive the ballbot to the origin with a lower total cost than LQR, then we would consider it to be superior to LQR.

We also wanted to evaluate controller robustness, which was done by varying the initial conditions, as well as perturbing the model parameters while using controllers which assumed an unperturbed plant. Three different plants were used in our analysis:

- A standard ballbot.
- A heavy ballbot, with a ball that was 20% heavier and had 20% more moment of inertia than the standard ballbot.
- A small ballbot, with a ball that was 20% smaller than the standard ballbot.

We also did some additional analysis of each controller independently - most notably changing and adding optimization constraints to the trajectory optimizer, and MPC controllers.

A. Linear Quadratic Regulator (LQR)

LQR performed as expected. Its suitability as a baseline controller was confirmed when evaluating its response to initial conditions and model parameters. It had no problem driving the ballbot to the origin from reasonable initial conditions (ie, where the ballbot is not inverted), and was remarkably insensitive to model parameters. LQR's performance with respect to a changing model is seen in Figure 4 via the trajectories taken as the ballbot parameters are changed. The same general shape is found in all solutions, owing to the only weakly nonlinear dynamics of the ballbot around the origin.

The LQR controller was very easy to implement and worked straight "out of the box." No significant tweaking or tuning of parameters was needed, yet it performed spectacularly as a controller. This is probably due to the plant's soft nonlinearities, which are well-behaved around the fixed point and can be well approximated by linearized functions. Thus the LQR controller's approximation of the the optimal policy is very close to the actual optimal policy, giving us amazing control over our system.

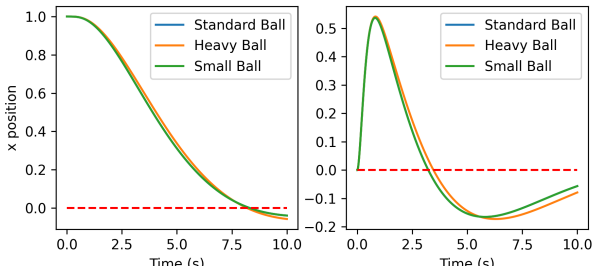


Fig. 4: The trajectory taken by the LQR controller as it drives variously sized ballbots to the origin from $\boldsymbol{q}(0) = \begin{bmatrix}1 & 0\end{bmatrix}$, $\dot{\boldsymbol{q}}(0) = \begin{bmatrix}0 & 0\end{bmatrix}$. The heavy ball has 20% more mass and moment of inertia than the standard ball, and the small ball is 20% smaller in radius.

B. Neural Network to Approximate a Cost-To-Go

The learned cost-to-go is sufficient for controlling the ballbot. The controller is able to drive the ballbot to the origin from reasonable initial conditions. As seen in Figure 5, at certain start states the dynamic programming approach can reach the goal state faster than LQR. However, due to discritization and fitting errors the controller is unable to stabilize the plant at this point. The ballbot then drifts from the origin and continues to accrue cost. Despite this, we observe in Figure 6 that dynamic programming's cost-to-go function closely matches LQR's near the fixed point. However, as we begin to traverse state space away from the center and its roughly linear dyanmics, we see that the controllers cost-to-go (and therefore policy) begins to diverge. This can give us an intuition as to why our dynamic programming approach can sometimes beat LQR in reaching the origin - the dynamic programming approach better plans for the non-linearities in the plant, of which LQR is not even aware.

In implementing the neural network we found a few issues. First of all is the aforementioned stability issue, where the controller would be unable to stop the ballbot from moving away from the origin, and it would slowly drift away from the origin if started at that point. Second was the sensitivity of the Network to its hyperparameters. Many combinations of hyper parameters needed to be brute-force tested by hand before a suitable set was found. Many times during training the network's performance would reach a local-minimum that would cause the resulting controller to perform terribly. The issue of having to try many hyperparameters is exacerbated by

the amount of time it took to train the network, often taking an hour or more of Satori supercomputer time before it was sufficiently trained. The convergence of the network was quite poor, and only after much tweaking to the batch size, state space domain, and the number of possible inputs, were we able to get the network to begin to converge.

Even with our best attempt, the dynamic programming approach still under-performed in many initial conditions compared to the LQR controller, accuring more cost on its way to the origin. It also took around ten times longer to run in Drake. These facts compounded with it's finicky nature, long training and implementation time, and non-scalabilty to large problems with more states and inputs limit the practicality of using such a controller on a real-world system.

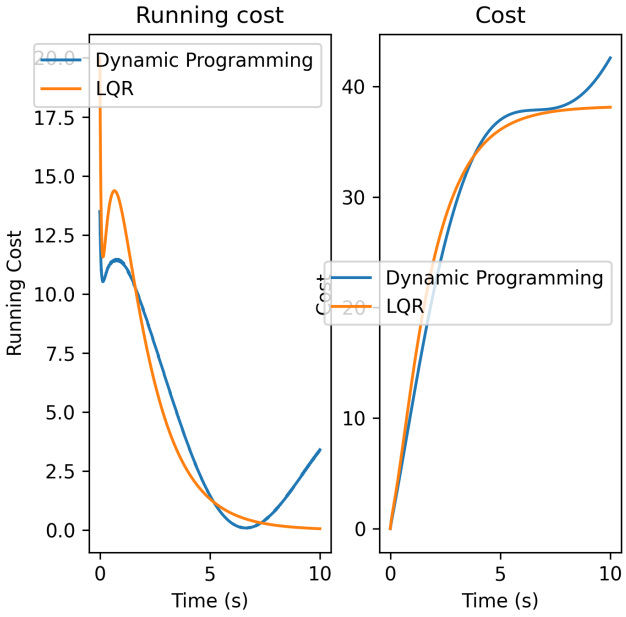
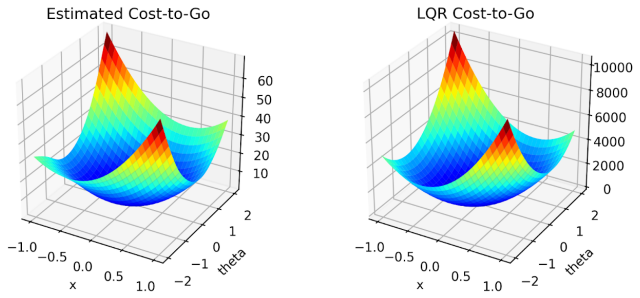


Fig. 5: The cost for the LQR controller vs the Dynamic programming controller starting from $\boldsymbol{q}(0) = \begin{bmatrix} 1 & 0 \end{bmatrix}$, $\dot{\boldsymbol{q}}(0) = \begin{bmatrix} 0 & 0 \end{bmatrix}$ and ending at the origin. The Dynamic programming approach begins by beating the LQR approach in time and cost, but overall loses because it is not able to stabilize at the origin.

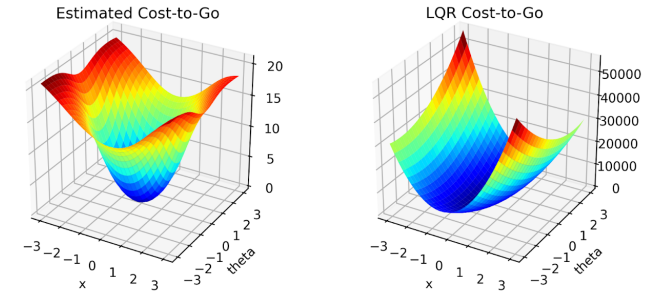
C. Trajectory Optimization

Trajectory optimization has a significant number of parameters that can be tweaked to gain intuition to the controller's performance. In our analysis, we chose to vary the constraints, initial conditions, model parameters, and cost functions.

By changing the initial conditions and time allotted, we were able to see that typically, even when the solver isn't given enough time to find a smooth solution (ie. one that would make sense if a human were controlling the ballbot), it is still able to find some feasible solution. Figure 7 shows the results of the trajectory optimizer when actuation, velocity, and distance



(a) The cost-to-go near the origin, as approximated by the neural network (left) and analytically determined by the by the LQR cost function (right).



(b) The cost-to-go far from the origin, as approximated by the neural network (left) and analytically determined by the by the LQR cost function (right).

Fig. 6: The cost-to-go as determined by the neural network and LQR linearized about the origin. Near the origin there is remarkable agreement between the two, but this agreement is not maintained across the remainder of state space. Note the differences in the absolute cost values, this is because of the LQR cost to go being discrete.

from the goal are all penalized. We can see from part (c) that if given enough time, the optimizer will bring the ballbot to the origin naturally - even if there is no formal constraint to do so. These curves are smooth and clearly show how the controller is bringing each value to zero. When the end constraint is strictly imposed, as in part (b), the motion is still fairly smooth and clearly indicates the ballbot reaching the origin with zero velocity. However, when the time allotted is shortened but the controller is still mandated to bring the ballbot to the origin, as in part (a), the trajectory becomes significantly less smooth. In this case, the controller accrues additional cost by using more actuation, causing large shifts in theta in order to meet the constraint that it must meet the goal state in the time allotted. The risk of generating non-smooth trajectories is not intrinsically concerning, as discontinuous control is appropriate for some underactuated systems. However, the fact that the solver shows behavior similar to a least-time controller without being given a least-time problem is worth noting. The

end user must have the intuition to recognize this issue, and compensate by either increasing the time steps or relaxing the constraint at the goal.

The primary issue with trajectory optimization was guaranteeing convergence of the solver, which was highly dependent on finding a good initial guess. We provided the optimizer with a very simple initial guess, where $\boldsymbol{q}$ was set to be linear for all timesteps, which was differentiated once to obtain $\dot{\boldsymbol{q}}$ and twice to obtain $\ddot{\boldsymbol{q}}$. No initial guess was provided for the actuator input. Despite the work needed to properly initialize the solver and ensure it gave reasonable output, the controller is able to beat LQR. It approaches the origin faster than LQR, with a lower running cost at all times during the trajectory. This is probably due to the controller actually having a better understanding of the non-linear dynamics vs LQR. While as previously stated the non-linear dynamics are weak they do add up over time, and being able to better predict them would allow for a lower cost to be accrued. This can be see in particular right at the start, where the LQR controller starts with a very strong actuation and overshoots before the controller needs to correct costing the LQR controller a lot of time and thus accuring extra cost. On the other hand the MPC controller uses a much smaller input and does not really overshoot, allowing it reach the goal in almost half the time and with almost 1/3 less cost.

D. Model Predictive Control

Surprisingly, MPC performed very similarly to LQR, albeit with a marginally higher total cost as seen in Figure 9. To bound the runtime on the controller, we set the horizon to be 50 points as was used in the normal trajectory optimizer, but ran 25 steps of the optimizer to approach the goal. This was chosen as longer horizons proved computationally intractable, as the optimization sometimes failed at some points during the simulation. This could potentially be alleviated by providing the previous trajectory as an initial guess to the current timestep, but we didn't have time to try that here.

Related Work

Our findings that LQR provides a practical and performant controller comes at no surprise. The first self-balancing ballbot built in 2005 employed a LQR controller [4], and most subsequent work has used a combination of LQR and hand-tuned linear control. [6] [5] These approaches failed to consider actuator limits in the controller, which led to instability. [6] In response, MPC has been implemented on a few ballbots, with iterative Linear Quadratic Gaussian (iLQR) optimization used for local trajectory optimization in 2014 [7], and Sequential Quadratic Programming (SQP) in 2019. [8]

The relatively recent appearance of nonlinear controllers on physical ballbots helps to confirm the sensitivity we observed when working with them. In practice, LQR is seen to be sufficient for control, especially when state estimation is employed.

V. Further Work

Although the comparison here was primarily conducted in terms of running cost, there are a few other metrics we would like to continue testing. First, the region of attraction of the dynamic programming controller is not well determined from our analysis. Estimating the quality of fit of the optimal cost-to-go away from the fixed point still needs to be done, but the dynamic programming approach might have a larger region of attraction than LQR thanks to the large subset of state space it was trained on.

Second, a possible compromise may exist between LQR and dynamic programming. Dynamic programming is not particularly stable near the fixed point due to numerical errors, but it has learned the dynamics better. It may be possible to use dynamic programming to bring the ballbot near the fixed point, then switch to a LQR controller to stabilize it there. This would hopefully expand the region of attraction of LQR, but also provide the exponential stability of LQR near the fixed point.

Third, it may be possible to improve the solution time of the trajectory optimizer. We found that SNOPT was incredibly sensitive to the initial guess provided, and in some cases its ability to converge depended on it. It may be possible to provide the solution of the N^{th} timestep in the MPC algorithim as an initial guess for the optimal trajectory at the $(N+1)^{\text{th}}$ timestep. This may provide better convergence, and perhaps decrease the time needed to solve the program.

VI. Conclusion

After working through all of these controllers, we learned a few things. First, LQR is amazing. It converges quickly and finds a policy that beats dynamic programming, even if it is not able to minimize the running cost nearly as quickly. Second, dynamic programming is kludgy. it had a hard time converging on the optimal cost-to-go function, especially at the edges of state space. Third, we learned that trajectory optimization requires a solid initial guess of the optimal trajectory, and significant manual tuning of horizons. In the end, our efforts were rewarded with a solid trajectory optimizer that could beat LQR, and a good idea of why our MPC formulation was behaving strangely.

VII. Source Code

The code used for this analysis can be found at `github.com/FischerMoseley/ball_boi`

References

[1] J. C. Butcher (2003), Numerical Methods for Ordinary Differential Equations, New York: John Wiley Sons, ISBN 978-0-471-96758-3.

[2] R. Tedrake. *Underactuated Robotics: Algorithms for Walking, Running, Swimming, Flying, and Manipulation (Course Notes for MIT 6.832)*. Downloaded on [date] from http://underactuated.mit.edu/

[3] U. Nagarajan et al., "The ballbot: An omnidirectional balancing mobile robot," The International Journal of Robotics Research, vol. 33, no. 6, pp. 917-930, 2013.

[4] T. Lauwers, G. Kantor, and R. Hollis, "12th International Symposium on Robotics Research," in One is Enough!, 2005.

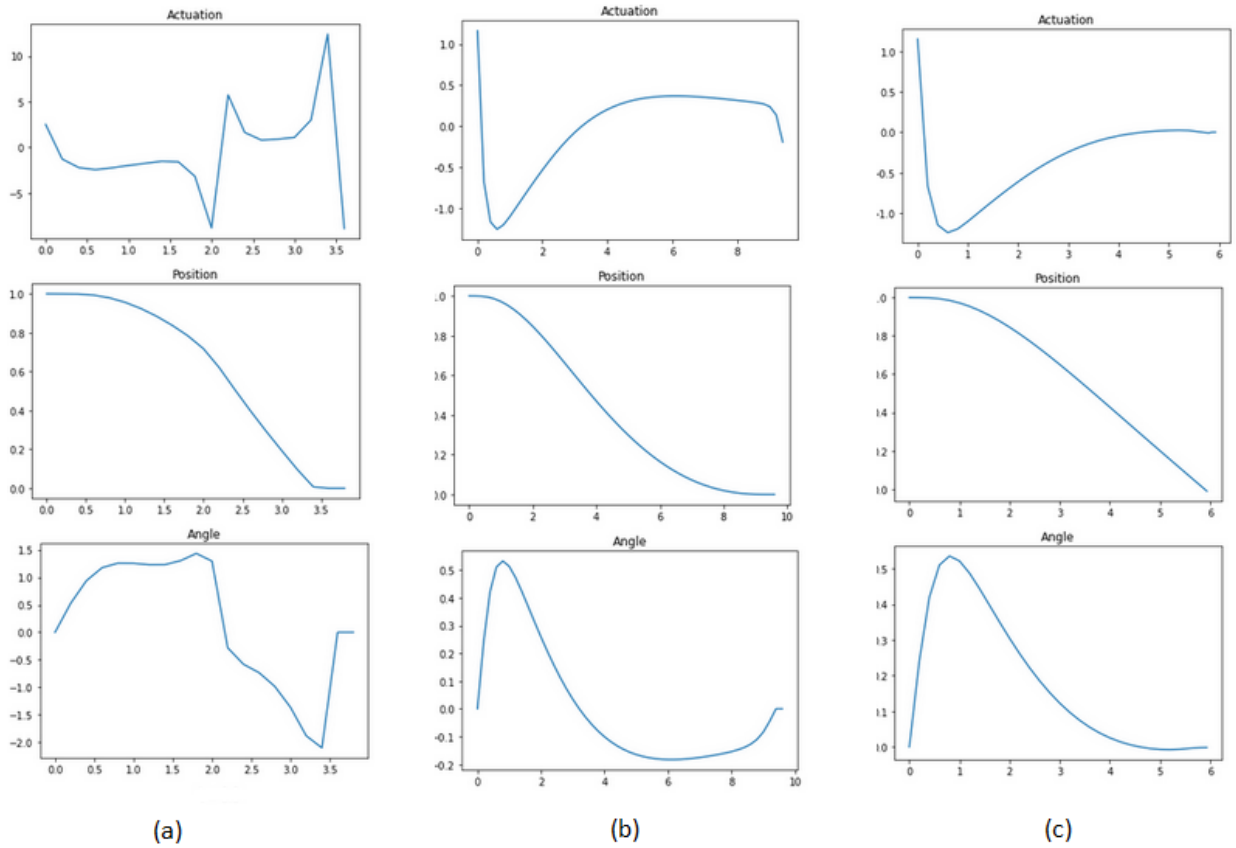


Fig. 7: The results of the trajectory optimization after penalizing in an LQR style. (a) shows the results when the optimizer is allowed 2s total and must get to the origin by the end, (b) shows the results when the optimizer is allowed 5s total and must get to the origin by the end, and (c) shows the results when the optimizer is allowed 5s total without an end constraint. The start state in all cases is $\boldsymbol{q}(0) = \begin{bmatrix} 1 & 0 \end{bmatrix}$, $\dot{\boldsymbol{q}}(0) = \begin{bmatrix} 0 & 0 \end{bmatrix}$.

[5] M. Kumagai and T. Ochiai, "Development of a robot balancing on a ball," 2008 International Conference on Control, Automation and Systems, 2008.

[6] P. Fankhauser and C. Gwerder, "Modeling and Control of a Ballbot," thesis, Eidgenössische Technische Hochschule Zürich, 2010.

[7] M. Neunert, F. Farshidian, and J. Buchli, "IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)," in Adaptive Real-time Nonlinear Model Predictive Motion Control, 2014.

[8] T. K. Jespersen, "Kugle: Modelling and Control of a Ball-Balancing Robot," thesis, Aalborg University, 2019.

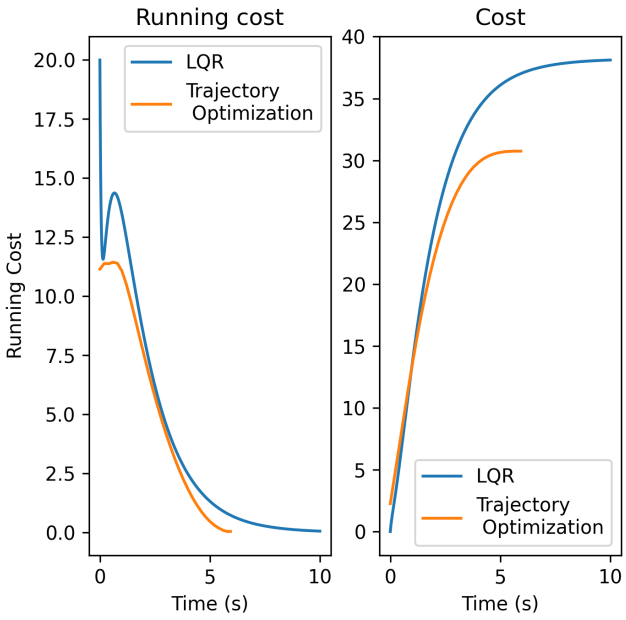


Fig. 8: The cost for the LQR controller vs the trajectory optimization starting from $\boldsymbol{q}(0) = \begin{bmatrix} 1 & 0 \end{bmatrix}$, $\dot{\boldsymbol{q}}(0) = \begin{bmatrix} 0 & 0 \end{bmatrix}$ and ending at the origin. We can see that the values for the cost of the trajectory optimization are lower overall and that it arrives at the origin quicker.

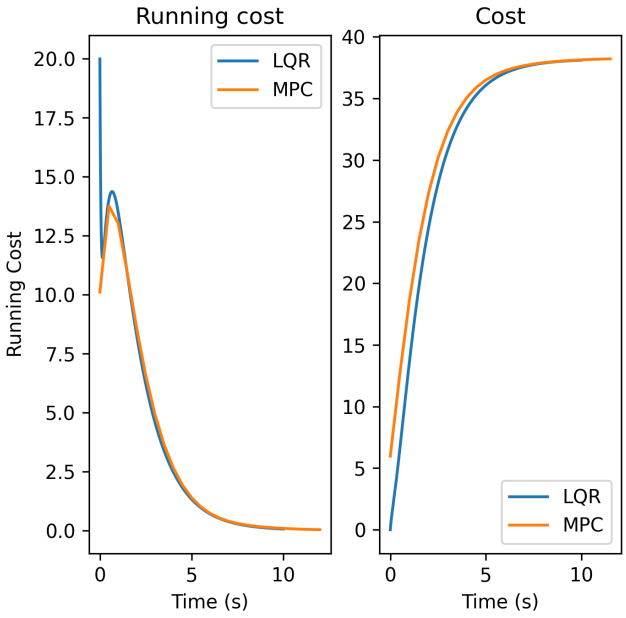


Fig. 9: The cost for the MPC controller vs LQRn starting from $\boldsymbol{q}(0) = \begin{bmatrix} 1 & 0 \end{bmatrix}$, $\dot{\boldsymbol{q}}(0) = \begin{bmatrix} 0 & 0 \end{bmatrix}$ and ending at the origin. It is seen that the running cost of trajectory taken by MPC is similar to that of LQR, however the total cost is slightly higher.