Trajectory Optimization and Model Predictive Control using Iterative Linear Quadratic Regulator (ILQR)

Tao Pang

***Abstract*— Based on the principles of dynamic programming, Iterative Linear Quadratic Regulator (ILQR) is a method to solve unconstrained, nonlinear trajectory optimization problems. Although ILQR does not allow constraints per se, it is possible to add additional terms to the cost function to "encourage" more interesting behaviors such as passing through way points. Moreover, it is claimed that ILQR can be solved at real-time rates (100Hz) and used as a controller. In this project, the feasibility of using ILQR as a controller is studied, and the finding is two-fold. Firstly, from a computational perspective, it is feasible to solve ILQR at 100Hz. However, ILQR suffers from the common difficulties faced by local methods: getting nice behavior out of the optimization involves a lot of gain tuning; and the lack of robustness against disturbances and measurement noise.**

I. Introduction

Differential dynamic programming (DDP) is a class of shooting methods for solving trajectory optimization problems. For a current trajectory of state $x(t)$ and control sequence $u(t)$, DDP iteratively optimizes the current trajectory by choosing a new control sequence $u^*(t)$ that minimizes a local, quadratic approximation of the cost-to-go.

Although DDP has existed in optimal control literature since the 1960s [1], it had not appeared much in robotics literature until 2003, when Todorov and Li evaluated the performance of several methods (DDP included) for generating locally optimal control policies for nonlinear robotic systems [2]. A year later, Li and Todorov noticed that the exact Hessian of the value function, which is required by DDP, is not necessary to generate good control policies. They proposed an algorithm which is similar to DDP, but uses a computationally-cheaper approximation of the Hessian, and named it iterative Linear Quadratic Regulator (ILQR) [3], probably due to the resemblance of the algorithm's equations to the Discrete-time Algebraic Riccati Equation (DARE). Yet a year later, ILQR was generalized to stochastic systems, and the resulting algorithm was aptly named iterative Linear Quadratic Gaussian (ILQG) [4]. In 2012, ILQR-based model predictive control (MPC) was applied to a simulated humanoid robot, which was able to synthesize remarkably complex behaviors such as getting up from prone [5].

Another line of work at ETH Zurich demonstrates that ILQR-based planning and control can be deployed on real hardware. Neunert *et al.* implemented ILQR-based Model Predictive Control (MPC) on smooth dynamical systems with 10-12 states (e.g. a hexcopter) [6]. Even for hybrid systems such as quadrupeds, complex trajectories such as trotting, galloping or even manipulating objects can emerge from unconstrained ILQR [7].

Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology, 77 Massachusetts Avenue, Cambridge, MA 02139. pangtao@mit.edu

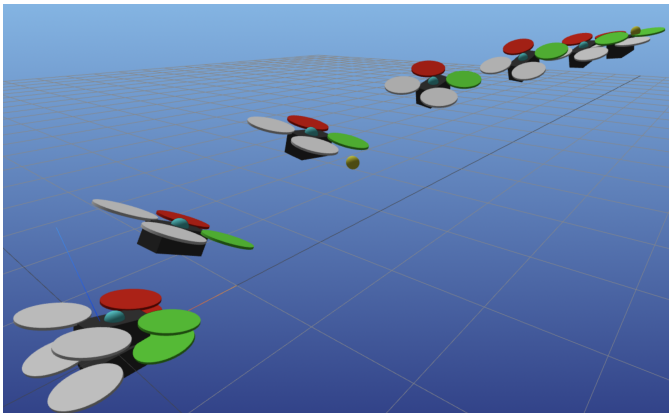

Fig. 1: Snapshots of quadrotor states along a trajectory generated by ILQR. The blue sphere represents the center of mass of the quadrotor. The trajectory starts at the origin (lower left corner) and ends at the goal point (yellow sphere at the upper right corner). This trajectory visualization is created with MeshCat [8].

I have yet to find a single definitive reference that clearly and succinctly describes both the theoretical and algorithmic aspect of ILQR. Almost every variation and trick of DDP/ILQR described in the papers published in the new millennium can be found in the book [1] by Jacobson and Mayne published in 1970. But as a book that talks about everything, it is hard to penetrate for people with limited knowledge in optimal control. Li and Todorov's early papers [2], [3] are too terse to follow. In [6], [7], the ILQR algorithm is squeezed into a quarter-page algorithm-formatted block, without clearly defining all the symbols used. Even worse, their description of the algorithm has a few obscure but crippling typos. Perhaps the best reference I have come across so far is [5]. It unambiguously describes the ILQR algorithm, and imparts as much theoretical background as possible within the page limit of conference papers.

This report is organized as follows: Section II gives a brief introduction to the ILQR algorithm; Section III and IV summarizes my observations when applying ILQR to double integrator and quadrotor, respectively.

II. ILQR algorithm

For a system with dynamics $\mathbf{x}_{i+1} = \mathbf{f}(\mathbf{x}_i, \mathbf{u}_i)$, the ILQR algorithm not only provides a (locally) op-

Algorithm 1: ILQR Algorithm

Given

- Discrete-time dynamics: $\mathbf{x}_{i+1} = \mathbf{f}(\mathbf{x}_i, \mathbf{u}_i)$
- Cost function: $J_0(\mathbf{x}_0, \mathbf{U}) = \sum_{i=0}^{N-1} l(\mathbf{x}_i, \mathbf{u_i}) + \frac{1}{2}(\mathbf{x}_N - \mathbf{x}_g)^T Q_N (\mathbf{x}_N - \mathbf{x}_g)$
- Initial and goal state: $\mathbf{x}_0$, $\mathbf{x}_g$
- Horizon: N

`// Initial trajectory.`
$\mathbf{U} \leftarrow \{\mathbf{u_0}, \mathbf{u_1}, \ldots, \mathbf{u}_{N-1}\}$
$\mathbf{X} \leftarrow \{\mathbf{x}_0, \mathbf{x}_1, \ldots, \mathbf{x}_N\}$
$J \leftarrow J_0(\mathbf{x}_0, \mathbf{U})$
$n_i = 0$ `// iterations counter.`
do
  `// Backward pass.`
  $V_{\mathbf{xx}} \leftarrow Q_N$
  $V_{\mathbf{x}} \leftarrow Q_N(\mathbf{x}_N - \mathbf{x}_g)$
  for $i \leftarrow N-1$ **to** 0 **do**
    `// Partial derivatives.`
    $l_{\mathbf{x}} \leftarrow \frac{\partial l}{\partial \mathbf{x}}\big|_{\mathbf{x}_i, \mathbf{u}_i}$, $l_{\mathbf{u}} \leftarrow \frac{\partial l}{\partial \mathbf{x}}\big|_{\mathbf{x}_i, \mathbf{u}_i}$
    $\mathbf{f}_{\mathbf{x}} \leftarrow \frac{\partial \mathbf{f}}{\partial \mathbf{x}}\big|_{\mathbf{x}_i, \mathbf{u}_i}$, $\mathbf{f}_{\mathbf{u}} \leftarrow \frac{\partial \mathbf{f}}{\partial \mathbf{u}}\big|_{\mathbf{x}_i, \mathbf{u}_i}$
    $Q_{\mathbf{x}} \leftarrow l_{\mathbf{x}} + \mathbf{f}_{\mathbf{x}}^T V_{\mathbf{x}}$
    $Q_{\mathbf{u}} \leftarrow l_{\mathbf{u}} + \mathbf{f}_{\mathbf{u}}^T V_{\mathbf{x}}$
    $Q_{\mathbf{xx}} \leftarrow l_{\mathbf{xx}} + \mathbf{f}_{\mathbf{x}}^T V_{\mathbf{xx}} \mathbf{f}_{\mathbf{x}}$
    $Q_{\mathbf{uu}} \leftarrow l_{\mathbf{uu}} + \mathbf{f}_{\mathbf{u}}^T V_{\mathbf{xx}} \mathbf{f}_{\mathbf{u}}$
    $Q_{\mathbf{ux}} \leftarrow l_{\mathbf{ux}} + \mathbf{f}_{\mathbf{u}}^T V_{\mathbf{xx}} \mathbf{f}_{\mathbf{x}}$
    `// Controller.`
    $\mathbf{k}_i = -Q_{\mathbf{uu}}^{-1} Q_{\mathbf{u}}$
    $\mathbf{K}_i = -Q_{\mathbf{uu}}^{-1} Q_{\mathbf{ux}}$
    `// update Vxx and Vx.`
    $V_{\mathbf{x}} \leftarrow Q_{\mathbf{x}} + \mathbf{K}_i^T Q_{\mathbf{uu}} \mathbf{k}_i + \mathbf{K}_i^T Q_{\mathbf{u}} + Q_{\mathbf{ux}}^T \mathbf{k}_i$
    $V_{\mathbf{xx}} \leftarrow Q_{\mathbf{xx}} + \mathbf{K}_i^T Q_{\mathbf{uu}} \mathbf{K}_i + \mathbf{K}_i^T Q_{\mathbf{ux}} + Q_{\mathbf{ux}}^T \mathbf{K}_i$
  end
  `// Forward pass with line search.`
  $\alpha = 1$
  $n_\alpha = 0$ `// line search steps counter.`
  $\hat{\mathbf{x}}_0 = \mathbf{x}_0$
  do
    for $i \leftarrow 0$ **to** $N-1$ **do**
      $\hat{\mathbf{u}}_i \leftarrow \mathbf{u}_i + \alpha \mathbf{k}_i + \mathbf{K}_i(\hat{\mathbf{x}}_i - \mathbf{x}_i)$
      $\hat{\mathbf{x}}_{i+1} = \mathbf{f}(\hat{\mathbf{x}}_i, \hat{\mathbf{u}}_i)$
    end
    $\hat{J} \leftarrow J_0\left(\mathbf{x}_0, \hat{\mathbf{U}}\right)$
    $\alpha \leftarrow \alpha/2$
    $n_\alpha \leftarrow n_\alpha + 1$
  while $\hat{J} > J$ *and* $n_\alpha \leq 5$
  cost_reduction $\leftarrow (J - \hat{J})/J$
  $J \leftarrow \hat{J}$, $\mathbf{X} \leftarrow \hat{\mathbf{X}}$, $\mathbf{U} \leftarrow \hat{\mathbf{U}}$
  $n_i \leftarrow n_i + 1$
while cost_reduction > 0.01 *and* $n_i \leq 5$

timal trajectory $\mathbf{X} \equiv \{\mathbf{x}_0, \mathbf{x}_1, \ldots, \mathbf{x}_N\}$ and $\mathbf{U} \equiv \{\mathbf{u_0}, \mathbf{u_1}, \ldots, \mathbf{u}_{N-1}\}$, it also provides a feedback controller $\mathbf{K} \equiv \{\mathbf{K}_0, \mathbf{K}_1, \ldots, \mathbf{K}_{N-1}\}$, such that the control input at time step i is given by

$$\mathbf{u}(i) \coloneqq \mathbf{u}_i + \mathbf{K}_i(\mathbf{x}(i) - \mathbf{x}_i), \tag{1}$$

where $\mathbf{x}(i)$ is the actual system state at step i.

When ILQR is used as MPC, new optimal trajectories are generated at a high rate. For every new trajectory generated, only $\mathbf{u}_0$ (or the first couple of $\mathbf{u}_i$'s) is applied to the system before the next optimal trajectory and control policy becomes available.

The cost-to-go function assumes the additive form commonly used in optimal control:

$$J_0(\mathbf{x}_0, \mathbf{U}) = \sum_{i=0}^{N-1} l(\mathbf{x}_i, \mathbf{u}_i) + l_f(\mathbf{x}_N). \tag{2}$$

A natural choice for l and l_f is

$$\begin{aligned} l(\mathbf{x}_i, \mathbf{u}_i) &= \frac{1}{2}\left(\bar{\mathbf{x}}_i^T Q \bar{\mathbf{x}}_i + \bar{\mathbf{u}}_i^T R \bar{\mathbf{u}}_i\right) \\ l_f(\mathbf{x}_N) &= \frac{1}{2}(\mathbf{x}_N - \mathbf{x}_g)^T Q_N (\mathbf{x}_N - \mathbf{x}_g) \end{aligned} \tag{3}$$

where $\bar{\mathbf{x}}_i = \mathbf{x}_i - \mathbf{x}_g$ and $\bar{\mathbf{u}}_i = \mathbf{u}_i - \mathbf{u}_g$. $\mathbf{x}_g$ and $mathbfu_g$ are time-invariant goal/target points. However, as ILQR (at least the version implemented in this project) does not allow constraints, it is hard to get "interesting" behaviors from simple quadratic objective functions.

One of the desirable properties of a trajectory is to pass through way points. A *way point* is a pair $(\mathbf{x}_w, t_w)$ where $\mathbf{x}_w$ is a point in the system's state space, and t_w a time. The system "passes" a way point $(\mathbf{x}_w, t_w)$ means that the state of the system is $\mathbf{x}_w$ at time t. Way points are useful, for instance, when a trajectory needs to pass through a narrow passage. A couple of way points can be placed along the passage to ensure that the trajectory is collision-free.

One way to "encourage" the trajectory to pass a way point (proposed in [7]) is to add to $l(\mathbf{x}_i, \mathbf{u}_i)$ a time-discounted quadratic form centered around $(\mathbf{x}_w, t_w)$:

$$\frac{1}{2}(\mathbf{x}_i - \mathbf{x}_w)^T Q_w (\mathbf{x}_i - \mathbf{x}_w) \sqrt{\frac{\rho_w}{2\pi}} \exp\left(-\frac{\rho_w}{2}(t - t_w)^2\right). \tag{4}$$

A pseudo-code version of the ILQR algorithm as implemented by the author is provided on the left. It uses the same notations as [5]. The choice of iteration and convergence threshold is entirely empirical. For example, the number of iterations for quadrotor trajectories to converge is almost always less than 4, so the iteration threshold is set to 5.

III. Running ILQR on Double Integrator

The first thing I tried with ILQR is to make sure that it generates reasonable trajectories for simple, linear systems such as the double integrator. In fact, when using a quadratic cost-to-go function (without adding way points), the ILQR backward pass lead to the following iterative equations:

$$\begin{aligned} V_{\mathbf{xx}} &= Q + A^T V'_{\mathbf{xx}} A - A^T V'_{\mathbf{xx}} B \left(R + B^T V'_{\mathbf{xx}} B\right)^{-1} B^T V'_{\mathbf{xx}} A \\ K &= \left(R + B^T V'_{\mathbf{xx}} B\right)^{-1} B^T V'_{\mathbf{xx}} A \end{aligned} \tag{5}$$

where the prime symbol denotes the next time step, and A, B are given by the linear dynamics of the double integrator system ($\dot{\mathbf{x}}_{i+1} = A\mathbf{x}_i + B\mathbf{u}_i$). Not surprisingly, Equation (5) is the same as the dynamic programming solution to the discrete-time LQR problem [9].

Figure 2 shows two trajectories generated by ILQR. The trajectories have initial condition $\mathbf{x}_0 = [0, 0]$ and goal state $\mathbf{x}_g = [1, 0]$. Both trajectories are driven to $\mathbf{x}_g$ by ILQR. Moreover, when the way point cost in Equation (4) is added to the cost function, the trajectory passes through the way point almost perfectly.

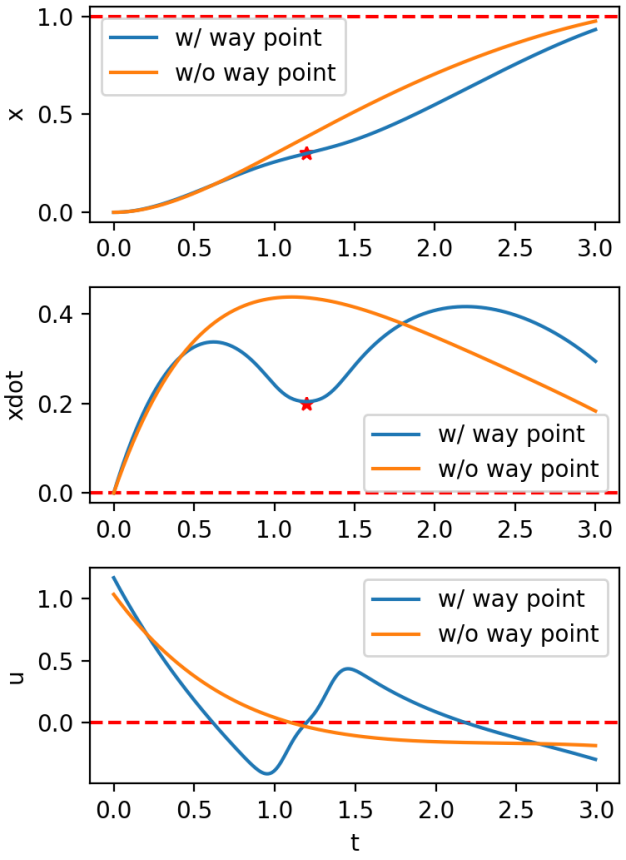

Fig. 2: Double integrator trajectories generated by ILQR. Red dotted lines denote $\mathbf{x}_g$. The red asterisk denotes way point $([0.3, 0.2], 1.2)$.

Figure 3 shows the different contributions to the total cost-to-go made by the LQR cost in Equation (3) and the way point cost in Equation (4). The total (J_{total}), LQR (J_{LQR}) and way point (J_{wpt}) cost-to-go are plotted along the trajectory itself, shown as the red dotted line in the phase plane. As expected, J_{LQR} decreases along the trajectory. In contrast, due to the time discount, J_{wpt} is flat before and after the waypoint, and only has non-zero gradient (and affects the trajectory) in the vicinity of the way point.

IV. Running ILQR on 3D Quadrotor

In this section, I will discuss how ILQR scales up to higher-dimensional systems. The system in question is the

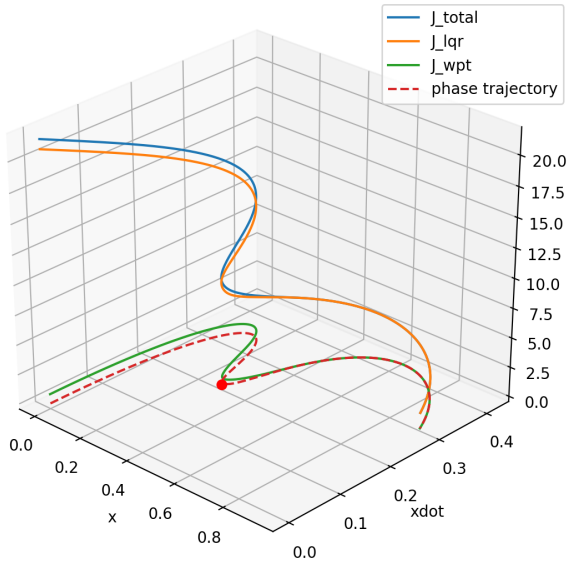

Fig. 3: Cost-to-go of states along the blue trajectory (which passes the way point) in Figure 2. The red dot denotes the way point.

3D quadrotor with 12 states. The state of the quadrotor is $\mathbf{x} = [q, \dot{q}]$, where $q = [x, y, z, \phi, \theta, \psi]$ is the configuration of the quadrotor. $[x, y, z]$ is the center of mass (CG) position of the quadrotor in the world reference frame. $[\phi, \theta, \psi]$ is the roll-pitch-yaw angles that represent the orientation of the quadrotor.

A. *Gain tuning is the name of the game.*

The quadrotor plant controlled by an ILQR MPC controller (running at 100Hz) is simulated in Drake [10]. The quadrotor starts at $\mathbf{x}_0 = [0, 0, 0, 0, 0, 0]$ and is commanded to reach $\mathbf{x}_g = [3, -1, 1, 0, 0, 0]$. A way point is defined at $([1, -0.3, 0.5, \pi/6, 0, 0], 1.0)$. Trajectories with and without way point terms in the cost function are shown in Figure 4 and Figure 5, respectively.

Unlike in the case of the double integrator system, the quadrotor trajectory does not perfectly pass through the given way point. Instead , it is only "nudged" towards the way point by the way point terms. In addition, divergence of ILQR is encountered quite frequently during gain tuning. I think these observations reflect the difficulties of using unconstrained, nonlinear optimizations in general: they *can* generate beautiful results when they converge, but convergence involves a lot of gain tuning.

B. *Towards real-time MPC*

Implemented in Python, the first version of ILQR MPC was only able to run at 2Hz, which is far from enough to control real hardware. Profiling using cProfile (Python's built-in profiling tool) reveals that the most expensive operations are

- $\frac{\partial \mathbf{f}}{\partial \mathbf{x}}$, $\frac{\partial \mathbf{f}}{\partial \mathbf{u}}$: taking the partial derivatives of the dynamics (with pydrake.forwarddiff), and
- $\mathbf{f}(\mathbf{x}, \mathbf{u})$: evaluating the dynamics.

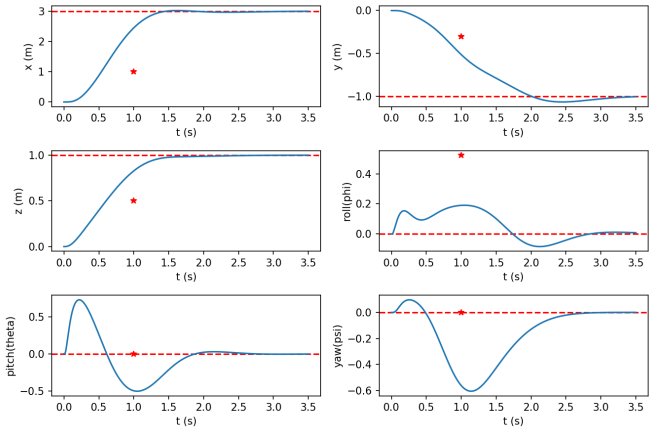

Fig. 4: Quadrotor trajectory using MPC with way point.

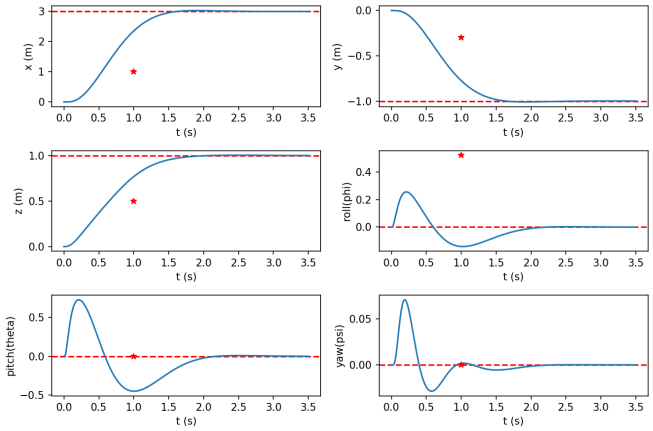

Fig. 5: Quadrotor trajectory using MPC without way point.

These expensive operations are re-implemented in C++, and the improvement is dramatic (Table I). The same operations are also implemented in Julia and the result is equally amazing. Unfortunately, calling Julia from Python is currently slow and buggy. People wishing to speed up their Python code with Julia will have to wait.

Note that the Julia and C++ implementations are not entirely the same, and the C++ time includes the overhead of calling C++ from Python. Therefore, this comparison should not be interpreted as a race between C++ and Julia. It is meant to show that running ILQR MPC at real time rates (100Hz) is feasible with some simple optimization.

V. Conclusion

For nonlinear systems, ILQR-based MPC *could* work, but not without a significant amount of gain tuning. With some optimization, it is also feasible to run ILQR at real-time rates (100Hz). However, there is no guarantee on the robustness or even the convergence of ILQR-based controller. Therefore, deploying such controller on real hardware could be risky.

Source Code

The python implementation of ILQR can be found at `https://github.mit.edu/pangtao/iLQR`.

Operation	Average time per call (μs)		
	Python	C++	Julia
$\frac{\partial \mathbf{f}}{\partial \mathbf{x}}, \frac{\partial \mathbf{f}}{\partial \mathbf{u}}$	1283	4.64	4.63
$\mathbf{f}(\mathbf{x}, \mathbf{u})$	547	1.83	0.36

TABLE I: Performance comparison between different implementations. Derivatives in C++ is evaluated using Eigen's autodiff with fixed-sized partials. Derivatives in Julia is evaluated with Julia's ForwardDiff package. Profiling of C++ functions is done in python using cProfile (python bindings created with Pybind11). Profiling of Julia code is done using Julia's benchmark tools.

The C++ implementation of quadrotor dynamics and partial derivatives can be found at `https://github.com/pangtao22/drake/blob/quadrotor_dynamics_autodiff/examples/quadrotor/quadrotor_dynamics.h`

Videos

Videos of quadrotor trajectories generated by ILQR can be found at `https://youtu.be/c5E7_0-b7es` and `https://youtu.be/22C3XeOvk2E`.

Acknowledgment

The author would like to thank Twan Koolen and Robin Deits for insightful advice and making the Julia code 300 times faster than the author's naive implementation.

References

[1] D. H. Jacobson and D. Q. Mayne, *Differential dynamic programming*. North-Holland, 1970.

[2] E. Todorov and W. Li, "Optimal control methods suitable for biomechanical systems," in *Engineering in Medicine and Biology Society, 2003. Proceedings of the 25th Annual International Conference of the IEEE*, vol. 2. IEEE, 2003, pp. 1758–1761.

[3] W. Li and E. Todorov, "Iterative linear quadratic regulator design for nonlinear biological movement systems." in *ICINCO (1)*, 2004, pp. 222–229.

[4] E. Todorov and W. Li, "A generalized iterative lqg method for locally-optimal feedback control of constrained nonlinear stochastic systems," in *American Control Conference, 2005. Proceedings of the 2005*. IEEE, 2005, pp. 300–306.

[5] Y. Tassa, T. Erez, and E. Todorov, "Synthesis and stabilization of complex behaviors through online trajectory optimization," in *Intelligent Robots and Systems (IROS), 2012 IEEE/RSJ International Conference on*. IEEE, 2012, pp. 4906–4913.

[6] M. Neunert, C. De Crousaz, F. Furrer, M. Kamel, F. Farshidian, R. Siegwart, and J. Buchli, "Fast nonlinear model predictive control for unified trajectory optimization and tracking," in *Robotics and Automation (ICRA), 2016 IEEE International Conference on*. IEEE, 2016, pp. 1398–1404.

[7] M. Neunert, F. Farshidian, A. W. Winkler, and J. Buchli, "Trajectory optimization through contacts and automatic gait discovery for quadrupeds," *IEEE Robotics and Automation Letters*, vol. 2, no. 3, pp. 1502–1509, 2017.

[8] R. Deits, "Meshcat-python: Webgl-based 3d visualizer for python," 2018. [Online]. Available: https://github.com/rdeits/meshcat-python

[9] S. Boyd, "Ee363 lecture notes, lecture 1," 2008. [Online]. Available: http://stanford.edu/class/ee363/lectures/dlqr.pdf

[10] R. Tedrake and the Drake Development Team, "Drake: A planning, control, and analysis toolbox for nonlinear dynamical systems," 2016. [Online]. Available: http://drake.mit.edu