AL-iLQR Tutorial

Brian Jackson

***Abstract*— Trajectory optimization is a powerful framework for controlling complex dynamical systems. While many algorithms for solving these difficult problems have been proposed, methods based on differential dynamic programming (DDP) have recently become very popular due to their straightforward implementation and computational efficiency. While the original DDP algorithm has no ability to deal with additional path constraints, the use of DDP within an augmented Lagrangian framework allows for a powerful and efficient framework for solving constrained trajectory optimization problems. This tutorial presents the algorithm in detail, based on experience derived from implementing state-of-the-art implementations of this algorithm, and aims to provide useful insight to help those unfamiliar with DDP methods quickly understand the algorithm and its extensions.**

I. Introduction

Trajectory optimization is a powerful framework for controlling complicated robotic systems. The value of trajectory optimization lies primarily in its generality, allowing it to be applied to a very broad class of dynamical systems. Importantly, trajectory optimization can be applied to any type of dynamical system whose dynamics are Markovian, i.e.

$$\dot{x} = f(x, u) \tag{1}$$

where $\dot{x} \in \mathbb{R}^n$ is the time derivative of the state $x \in \mathbb{R}^n$ and $u \in \mathbb{R}^m$ are the controls.

While very general and powerful, trajectory optimization, like all nearly all branches of optimal control, makes the important assumption that the states of the system are known exactly, i.e. that the system has full state feedback. Most formulations of trajectory optimization assume a given initial condition.

Trajectory optimization can be succinctly summarized by the underlying optimization problem being solved:

$$\begin{aligned} \underset{x(t),\, u(t)}{\text{minimize}} \quad & \ell(x(t_f)) + \int_0^{t_f} \ell(x(t), u(t)) dt \\ \text{subject to} \quad & \dot{x} = f(x, u), \\ & x(0) = x_0, \\ & g(x(t), u(t)) \leq 0, \\ & h(x(t), u(t)) = 0, \end{aligned} \tag{2}$$

where $x(t)$ and $u(t)$ are the state and control trajectories from time 0 to T. The dynamics constraint, which constrains derivatives of the optimization variables, is what sets trajectory optimization apart from other general nonlinear optimization problems. The field of trajectory optimization is dedicated to finding efficient ways to solve these differentially-constrained optimization problems.

The most common approach, and the one treated exclusively in the current tutorial, is to discretize the problem in time, dividing the time period of length T seconds into $N-1$ segments, typically of equal length of Δt seconds. This results in N discretization points, also referred to as "knot" points, including the initial and final times. There exist many methods for approximating the integrals in (2) with discrete sums, as well as transforming the ordinary differential equation for the dynamics (1) into a discrete difference equation of the form:

$$x_{k+1} = f(x_k, u_k, \Delta t). \tag{3}$$

The resulting discrete optimization problem can then be written as:

$$\begin{aligned} \underset{x_{0:N},\, u_{0:N-1}}{\text{minimize}} \quad & \ell_N(x_N) + \sum_{k=0}^{N-1} \ell_k(x_k, u_k, \Delta t) \\ \text{subject to} \quad & \\ & x_{k+1} = f(x_k, u_k, \Delta t), k = 1, ..., N\text{-}1, \\ & g_k(x_k, u_k)\{<= 0\}, \forall k, \\ & h_k(x_k, u_k) = 0, \forall k, \end{aligned} \tag{4}$$

There exist many methods for solving problems of the form (4). These methods are typically divided into two categories: "indirect" and "direct" methods. Direct methods treat both the states and controls as decision variables and use general-purpose nonlinear programming (NLP) solvers, such as SNOPT or IPOPT. These methods typically transcribe the optimization problem into something of the form given in (4), often with varying methods for approximating the continuous-time dynamics or unique formulations of problem constraints. The most common method, direct collocation (DIRCOL), uses Hermite-Simpson integration to integrate both the cost and the dynamics, which is essentially a 3rd order implicit Runge-Kutta integrator for the states and first-order hold (i.e. linear interpolation) for the controls. These methods benefit directly from the robustness and generality of the NLP solvers on which they depend. However, direct methods also tend to be fairly slow and require large optimization packages.

Alternatively, indirect methods exploit the Markov structure of (4) and often impose strict Markovianity across the entire problem, including the cost functions and constraints. The dynamics constraints are then implicitly enforced by simulating forward the system's dynamics. Differential Dynamic Programming (DDP) and iterative LQR (iLQR) are closely related indirect methods that solve (4) by breaking it into a sequence of smaller sub-problems. DDP methods improve on more naive "simple shooting" methods by incorporating a feedback policy during the forward simulation of the dynamics at each time step. Because of their strict enforcement of dynamic feasibility, it is often difficult to find a control sequence that produces a reasonable initialization for DDP methods. While they are fast and have a low memory footprint, making them amenable to embedded implementation, DDP methods have historically been considered less numerically robust and less well-suited to handling nonlinear state and input constraints.

This tutorial derives a method for solving constrained trajectory optimization problems using DDP or iLQR within an augmented Lagrangian framework. The result is a fast, efficient algorithm that allows nonlinear equality and inequality constraints on both the states and controls.

This tutorial is based on previous research [1]–[3] and experience implementing this algorithm in several programming languages.

II. Background

A. Augmented Lagrangian

Augmented Lagrangian is an optimization method for solving constrained optimization problems of the form

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & f(x) \\ \text{subject to} \quad & c(x)\{\leq,=\}0. \end{aligned} \tag{5}$$

One of the easiest methods for solving constrained optimization problems is to move the constraints into the cost function and iteratively increase the penalty for either getting close to or violating the constraint. However, penalty methods only converge to the optimal answer as the penalty terms are increased to infinity, which is impractical to implement in a numerical optimization routine with limited-precision arithmetic. Augmented Lagrangian methods improve on penalty methods by maintaining estimates of the Lagrange multipliers associated with the constraints. This is accomplished by forming the augmented Lagrangian

$$\mathcal{L}_A = f(x) + \lambda^T c(x) + \frac{1}{2} c(x)^T I_\mu c(x) \tag{6}$$

where λ are the Lagrange multipliers, μ are the penalty multipliers, and

$$I_\mu = \begin{cases} 0 & \text{if } c_i(x) < 0 \wedge \lambda_i = 0,\ i \in \mathcal{I} \\ \mu_i & \text{otherwise} \end{cases}. \tag{7}$$

The problem is then solved as follows:

1) solve $\min_x \mathcal{L}_A(x,\lambda,\mu)$, holding λ and μ constant
2) update Lagrange multipliers

$$\lambda_i^+ = \begin{cases} \lambda_i + \mu_i c_i(x^*) & i \in \mathcal{E} \\ \max(0, \lambda_i + \mu_i c_i(x^*) & i \in \mathcal{I}, \end{cases} \tag{8}$$

3) update penalty term: $\mu^+ = \phi\mu$, $\phi > 1$
4) check constraint convergence
5) if tolerance not met, go to 1

where $\mathcal{E}$ and $\mathcal{I}$ are the sets of equality and inequality constraint indices, respectively and ϕ is the penalty scaling parameter (typically $2 \leq \phi \leq 10$).

B. Linear Quadratic Regulator

The Linear Quadratic Regular (LQR) problem is a canonical problem in the theory of optimal control, partially due to the fact that it has analytical solutions that can be derived using a variety of methods, and from the fact that LQR is an extremely useful tool in practice. We present derivations for both continuous-time and discrete-time LQR. The discrete-time derivation is particularly useful in setting up the ideas that will used in deriving constrained DDP.

The continuous-time LQR problem is formulated as

$$\begin{aligned} \min_{u(t)} \quad & \frac{1}{2} x^T(t_f) Q(t_f) x(t_f) \\ & + \frac{1}{2}\int_0^{t_f} [x^T(t)Q(t)x(t) + u^T(t)R(t)u(t)]dt \\ \text{s.t.} \quad & \dot{x}(t) = A(t)x(t) + B(t)u(t) \end{aligned} \tag{9}$$

where R is a real, symmetric, positive-definite matrix and Q is a real, symmetric, positive semi-definite matrix.

and the discrete-time LQR problem is

$$\begin{aligned} \min_{u(t)} \quad & \frac{1}{2} x_N^T Q_N x_N \\ & + \frac{1}{2}\sum_{k=0}^{N-1} [x_k^T Q_k x_k + u_k^T R_k u_k]dt \\ \text{s.t.} \quad & x_{k+1} = A_k x_k + B_k u_k \end{aligned} \tag{10}$$

1) Hamilton-Jacobi-Bellman Derivation: The Hamilton-Jacobi-Bellman equation is an important equation in the theory of optimal control that states the necessary conditions for optimality for a continuous-time system. We define the cost function to be

$$J(x(t),u(t),t) = \ell(x(t_f),t) + \int_0^{t_f} \ell(x(t),u(t),t)dt \tag{11}$$

and the minimum cost function

$$J^*(x(t),t) = \min_{u(t)} J(x(t),u(t),t). \tag{12}$$

We now define the Hamiltonian as

$$\begin{aligned} \mathcal{H}(x(t),u(t),J_x^*,t) &= \ell(x(t),u(t),t) \\ &\quad + J_x^{*T}(x(t),t)[f(x(t),u(t),t] \end{aligned} \tag{13}$$

where $J_x^* = \frac{\partial J^*}{\partial x}$ and $f(x(t),u(t),t)$ are the dynamics.

The Hamilton-Jacobi-Bellman equation is then

$$0 = J_t^*(x(t),t) + \mathcal{H}(x(t),u^*(t),J_x^*,t) \tag{14}$$

We now use these to solve the continuous LQR problem. We form the Hamiltonian

$$\begin{aligned} \mathcal{H}(x(t),u(t),J_x^*,t) &= x^T(t)Q(t)x(t) + u^T(t)R(t)u(t) \\ &\quad + J_x^{*T}(x(t),t)[A(t)x(t) + B(t)u(t)] \end{aligned} \tag{15}$$

and minimize it with respect to $u(t)$ by setting $\partial \mathcal{H}/\partial u = 0$:

$$\frac{\partial \mathcal{H}}{\partial u} = Ru + B^T J_x^* = 0. \tag{16}$$

Solving with respect to u yields the optimal control

$$u^* = -R^{-1} B^T J_x^* \tag{17}$$

which is globally optimal since $\partial^2 \mathcal{H}/\partial u^2 = R(t)$ is positive definite.

Substituting the optimal control back into our Hamiltonian (15):

$$\begin{aligned}\mathcal{H}(x, u^*, J_x^*, t) =& \frac{1}{2} x^T Q x + \frac{1}{2} J_x^{*T} B R^{-1} B^T J_x^* \\ &+ J_x^{*T} [Ax - BR^{-1} B^T J_x^*] \\ =& \frac{1}{2} x^T Q x - \frac{1}{2} J_x^{*T} B R^{-1} B^T J_x^* \\ &+ J_x^{*T} A x\end{aligned} \tag{18}$$

We're now ready to use the Hamilton-Jacobi-Bellman equation (14). Since the HJB equation is a first-order partial differential equation of the minimum cost, we need to guess a solution, which we assume to be quadratic:

$$J^*(x(t), t) = \frac{1}{2} x^T(t) K(t) x(t) \tag{19}$$

where K is a symmetric positive-definite matrix.

Plugging these into the HJB equation we get:

$$0 = \frac{1}{2} x^T \big(\dot{K} + Q - K^T B R^{-1} B^T K + 2KA \big) x \tag{20}$$

Leveraging the symmetry of quadratic forms and the fact that (20) must be equal to zero for all $x(t)$ we arrive at the Riccati equation:

$$0 = \dot{K} + Q - K^T B R^{-1} B^T K + KA + A^T K. \tag{21}$$

When solved for $K(t)$ using a specialized Riccati solver, the optimal control law is given by

$$u*(t) = -R^{-1}(t) B^T(t) K(t) x(t). \tag{22}$$

2) Discrete LQR: We now shift our focus to solving the discrete-time LQR problem (10). We start by defining the value function

$$V_i(x) = \min_{u_k, \dots, u_{N-1}} \frac{1}{2} x_N^T Q x_N + \frac{1}{2} \sum_{k=i}^{N-1} x_k^T Q_k x_k + u_k^T R_k u_k \tag{23}$$

subject to the dynamics: $x_{k+1} = A_k x_k + B_k u_k$. This gives the cost of starting at a particular state and simulating the dynamics forward. The cost of the entire trajectory is therefore $V_0(x_0)$. From the Bellman equation and the principle of optimality we can re-define the value function in a more convenient, recursive form:

$$V_k(x) = \min_{u_k} \frac{1}{2} x_k^T Q_k x_k + \frac{1}{2} u_k^T R_k u_k + V_{k+1}(A_k x_k + B_k u_k) \tag{24}$$

which simple state that the cost-to-go is simply the cost incurred for the current decision, plus the cost-to-go of where our current decision takes us (by simulating our dynamics forward one time instance). For convenience, we define the action-value function $Q(x_k, u_k)$ to be the value being minimized:

$$V_k(x) = \min_{u_k} Q(x_k, u_k) \tag{25}$$

We also assume that $V_k(x)$ is a quadratic form, i.e.

$$V_k(x) = \frac{1}{2} x_k^T P_k x_k \tag{26}$$

so that the action-value function takes the following form:

$$\begin{aligned}Q(x_k, u_k) = & \frac{1}{2} x_k^T Q_k x_k + \frac{1}{2} u_k^T R_k u_k \\ & + \frac{1}{2} (A_k x_k + B_k u_k)^T P_{k+1} (A_k x_k + B_k u_k)\end{aligned} \tag{27}$$

Since there are no controls to optimize at the last time step, we note that

$$V_N(x) = \frac{1}{2} x_N^T Q_N x_N \tag{28}$$

Since we know the terminal cost-to-go, we can find the optimal cost-to-go for all time steps once we have a recurrence relation that gives V_k as a function of V_{k+1}. We start by optimizing the action-value function at time step k, which has the following first-order necessary condition:

$$\begin{aligned}\frac{\partial Q}{\partial u} =& 0 \\ =& R_k u_k + B^T P_{k+1} (A_k x_k + B_k u_k)\end{aligned} \tag{29}$$

Solving for u we find the optimal control trajectory:

$$\begin{aligned}u_k^* =& -(R_k + B_k^T P_{k+1} B_k)^{-1} B_k^T P_{k+1} A_k x_k \\ =& -Q_{uu}^{-1} Q_{ux} x_k \\ =& -K_k x_k\end{aligned} \tag{30}$$

where $Q_{uu} = \partial^2 Q/\partial u^2$ and $Q_{ux} = \partial^2 Q/\partial u \partial x$. We introduce this notation for easy comparison to the more complicated DDP derivation in the following sections.

With an optimal feedback policy, we now substitute (30) into (27) to get the cost-to-go:

$$\begin{aligned}V_x(x) =& Q(x_k, u_k^*) \\ =& \frac{1}{2} \big(x_k^T Q_k x_k + x_k^T K_k^T R_k K_k x_k \\ & + x_k^T (A_k - B_k K_k)^T P_{k+1} (A_k - B_k K_k) x_k \big) \\ =& \frac{1}{2} x^T \big(Q_k + A_k^T P_{k+1} A_k \\ & + K_k^T (R_k + B^T P_{k+1} B_k) K_k \\ & - K_k^T B_k^T P_{k+1} A_k - A_k P_{k+1} B_k K_k \big) x_k \\ =& \frac{1}{2} x_k^T P_k x_k\end{aligned} \tag{31}$$

where, after substituting in (30) and simplifying,

$$\begin{aligned}P_k =& Q_k + A_k^T P_{k+1} A_k \\ & - A_k^T P_{k+1}^T B_k (R_k + B_k^T P_{k-1} B_k)^{-1} B_k^T P_{k+1} A_k.\end{aligned} \tag{32}$$

This establishes the recurrence relation between V_k and V_{k+1}, which can be used to recursively calculate the cost-to-go for the entire trajectory, along with the optimal feedback control gains K_k.

III. AL-DDP

We now present the derivation for solving constrained trajectory optimization problems using differential dynamic programming and iterative LQR within an augmented Lagrangian framework (AL-DDP or AL-iLQR). The derivation proceeds very similarly to that of discrete LQR, as derived in Section II-B.2.

The key idea of DDP is that at each iteration, all nonlinear constraints and objectives are approximated using first or second order Taylor series expansions so that the approximate functions, now operating on deviations about the nominal trajectory, can be solved using discrete LQR. This optimal feedback policy is computed during the "backward pass" (Algorithm 1), since the dynamic programming step begins at the tail of the trajectory, as in LQR. The optimal deviations are then applied to the nominal trajectory during a "forward pass" (Algorithm 2), using the optimal feedback policy during the forward simulation—also known as rollout–of the dynamics.

To handle constraints, we simply "augment" the cost function with the multiplier and penalty terms of the augmented Lagrangian, treating λ and μ as constants. After several iterations DDP, the multipliers and penalty terms are updated, and the process is repeated. The algorithm is summarized in Algorithm 3. We now proceed with the formal derivation.

A. Backward Pass

We first form the augmented Lagrangian of (4):

$$\begin{aligned}\mathcal{L}_A =&\ell_N(x_N) + \big(\lambda_N + \frac{1}{2}c_N(x_N)I_{\mu,N}\big)^T c_N(x_N) \\ &+ \sum_{k=0}^{N-1} \big[\ell_k(x_k,u_k,\Delta t) \\ &+ \big(\lambda + \frac{1}{2}c_k(x_k,u_k)^T I_{\mu,k}\big)^T c_k(x_k,u_k)\big] \\ =&\mathcal{L}_N(x_N,\lambda_N,\mu_N) + \sum_{k=0}^{N-1}\mathcal{L}_k(x_k,u_k,\lambda_k,\mu_k)\end{aligned} \tag{33}$$

where $\lambda_k \in \mathbb{R}^{p_k}$ is a Lagrange multiplier, $\mu_k \in \mathbb{R}^{p_k}$ is a penalty weight, and $c_k = (g_k, h_k) \in \mathbb{R}^{p_k}$ is the concatenated set of inequality and equality constraints with index sets $\mathcal{I}_k$ and $\mathcal{E}_k$, respectively. $I_{\mu_k} \in \mathbb{R}^{p_k \times p_k}$ is the penalty matrix defined in (7).

We now define the cost-to-go and the action-value functions as before:

$$V_N(x_N)|_{\lambda,\mu} = \mathcal{L}_N(x_N,\lambda_N,\mu_N) \tag{34}$$
$$\begin{aligned}V_k(x_k)|_{\lambda,\mu} &= \min_{u_k}\{\mathcal{L}_k(x_k,u_k,\lambda_k,\mu_k) \\ &\quad + V_{k+1}(f(x_k,u_k,\Delta t))|_{\lambda,\mu}\}\end{aligned} \tag{35}$$
$$= \min_{u_k} Q(x_k,u_k)|_{\lambda,\mu}, \tag{36}$$

where $V_k(x)|_{\lambda,\mu}$ is the cost-to-go at time step k evaluated with the Lagrange multipliers λ and the penalty terms μ.

In order to make the dynamic programming step feasible, we take a second-order Taylor series of the nonlinear cost-to-go

$$\delta V_k(x) \approx \frac{1}{2}\delta x_k^T P_k \delta x_k + p_k^T \delta x_k \tag{37}$$

where P_k and p_k are the Hessian and gradient of the cost-to-go at time step k, respectively. It is important to note that by taking Taylor series approximations, we have now switched to optimizing deviations about the nominal control trajectory.

Similar to (28), we can trivially calculate the cost-to-go at the terminal time step since there are no controls to optimize:

$$p_N = (\ell_N)_x + (c_N)_x^T(\lambda + I_{\mu_N} c_N) \tag{38}$$
$$P_N = (\ell_N)_{xx} + (c_N)_x^T I_{\mu_N} (c_N)_x. \tag{39}$$

which are simply the gradient and Hessian of (34) with respect to the states x_N.

The relationship between δV_k and δV_{k+1} is derived by taking the second-order Taylor expansion of Q_k with respect to the state and control,

$$\delta Q_k = \frac{1}{2}\begin{bmatrix}\delta x_k \\ \delta u_k\end{bmatrix}^T \begin{bmatrix}Q_{xx} & Q_{xu} \\ Q_{ux} & Q_{uu}\end{bmatrix}\begin{bmatrix}\delta x_k \\ \delta u_k\end{bmatrix} + \begin{bmatrix}Q_x \\ Q_u\end{bmatrix}^T \begin{bmatrix}\delta x_k \\ \delta u_k\end{bmatrix} \tag{40}$$

Dropping the time-step indices for notational clarity, the block matrices are,

$$Q_{xx} = \ell_{xx} + A^T P' A + c_x^T I_\mu c_x \tag{41}$$
$$Q_{uu} = \ell_{uu} + B^T P' B + c_u^T I_\mu c_u \tag{42}$$
$$Q_{ux} = \ell_{ux} + B^T P' A + c_u^T I_\mu c_x \tag{43}$$
$$Q_x = \ell_x + A^T p' + c_x^T(\lambda + I_\mu c) \tag{44}$$
$$Q_u = \ell_u + B^T p' + c_u^T(\lambda + I_\mu c), \tag{45}$$

where $A = \partial f/\partial x|_{x_k,u_k}$, $B = \partial f/\partial u|_{x_k,u_k}$, and $'$ indicates variables at the $k+1$ time step. It is in this step that we differentiate between DDP and iLQR. DDP computes the full second-order expansion, which includes computations with rank-3 tensors resulting from the vector-valued dynamics and general constraints. iLQR computes these expansions only to first order, such that the dynamics and constraints are both linear in the states and controls, resulting in a Gauss-Newton approximation of the true Hessian. While this lower accuracy expansion results in the need for more iterations, these iterations are considerably less expensive to compute, often resulting in a considerably faster overall convergence rate. The value of the full second-order information often depends on the type of problem being solved, and there exist a variety of methods for approximating the rank-3 tensor information without the need to explicitly compute it. The motivation for the name "iterative LQR" should now be clear, as we see that by linearizing the dynamics we arrive at problem nearly identical to (10).

Minimizing (40) with respect to δu_k gives a correction to the control trajectory. The result is a feedforward term d_k and a linear feedback term $K_k \delta x_k$. Regularization is added to ensure the invertibility of Q_{uu}:

$$\delta u_k^* = -(Q_{uu} + \rho I)^{-1}(Q_{ux}\delta x_k + Q_u) \equiv K_k \delta x_k + d_k. \tag{46}$$

Take a quick moment to note that this is almost exactly the same as (30), except we have now added some regularization to handle poorly conditioned Hessians (since these now depend on the expansion about the nominal trajectory) and now have a feedforward term, which results from the linear terms in the expansion. We would see similar terms appear in LQR if we included linear terms in the cost function.

Substituting δu_k^* back into (40), a closed-form expression for p_k, P_k, and the expected change in cost, ΔV_k, is found:

$$P_k = Q_{xx} + K_k^T Q_{uu} K_k + K_k^T Q_{ux} + Q_{xu} K_k \tag{47}$$
$$p_k = Q_x + K_k^T Q_{uu} d_k + K_k^T Q_u + Q_{xu} d_k \tag{48}$$
$$\Delta V_k = d_k^T Q_u + \frac{1}{2} d_k^T Q_{uu} d_k. \tag{49}$$

Algorithm 1 Backward Pass

```
1:  function BACKWARDPASS($X, U$)
2:      $p_N, P_N \leftarrow (38), (39)$
3:      for k=N-1:-1:0 do
4:          $\delta Q \leftarrow$ (40), (41)-(43)
5:          if $Q_{uu} \succ 0$ then
6:              $K, d, \Delta V \leftarrow (46), (49)$
7:          else
8:              Increase $\rho$ and go to line 3
9:          end if
10:     end for
11: return $K, d, \Delta V$
12: end function
```

B. Forward Pass

Now that we have computed the optimal feedback gains for each time step, we now update the nominal trajectories by simulating forward the dynamics. Since the initial state is fixed, the entire forward simulation can be summarized by the following updates:

$$\delta x_k = \bar{x}_k - x_k \tag{50}$$
$$\delta u_k = K_k \delta x_k + \alpha d_k \tag{51}$$
$$\bar{u}_k = u_k + \delta u_k \tag{52}$$
$$\bar{x}_{k+1} = f(\bar{x}_k, \bar{u}_k) \tag{53}$$

where $\bar{x}_k$ and $\bar{u}_k$ are the updated nominal trajectories and $0 \leq \alpha \leq 1$ is a scaling term.

1) Line Search: : As with all nonlinear optimization, a line search along the descent direction is needed to ensure an adequate reduction in cost. We employ a simple backtracking line search on the feedforward term using the parameter α. After applying equations (50)-(53) to get candidate state and control trajectories, we compute the ratio of the actual decrease to the expected decrease:

$$z = \frac{J(X, U) - J(\bar{X}, \bar{U})}{-\Delta V(\alpha)} \tag{54}$$

where

$$\Delta V(\alpha) = \sum_{k=0}^{N-1} \alpha d_k^T Q_u + \alpha^2 \frac{1}{2} d_k^T Q_{uu} d_k \tag{55}$$

is the expected decrease in cost, computed by simply using $\bar{d}_k = \alpha d_k$ to compute (49) at each time step. This can be computed very efficiently by simply storing the two terms in (49) separately and then scaling them by α and α^2 during the forward pass.

If z lies within a the interval $[\beta_1, \beta_2]$, usually $[1e\text{-}4, 10]$, we accept the candidate trajectories. If it does not, we update the scaling parameter $\alpha = \gamma\alpha$, where $0 < \gamma < 1$ is the backtracking scaling parameter. $\gamma = 0.5$ is typical. Increasing the lower bound on the acceptance interval will require that more significant progress is made during each backward-forward pass iteration. Decreasing the upper bound will keep the progress closer to the expected decrease. These values aren't changed much in practice.

2) Regularization: If the line search fails to make progress after a certain number of iterations, or the cost "blows up" to exceed some maximum threshold (can easily happen for highly nonlinear, unstable dynamics) the forward pass is abandoned and the regularization term ρ is increased prior to starting the backward pass. Increasing the regularization term effectively makes the partial Hessian in (46) more like the identity matrix, effectively "steering" the Newton (or Gauss-Newton) step direction towards the more naive gradient descent direction, which tends to be more reliable when the current iterate is far from the local optimum.

The regularization is also increased if the partial Hessian in (46) looses rank during the backward pass. In this case, the regularization term is increased and the backward pass is restarted from the beginning. The regularization is only decreased after a successful backward pass. The regularization scaling value is usually between 1.5 and 2.0.

Algorithm 2 Forward Pass

```
1:  function FORWARDPASS($X, U, K, d, \Delta V, J$)
2:      Initialize $\bar{x}_0 = x_0$, $\alpha = 1$, $J^- \leftarrow J$
3:      for k=0:1:N-1 do
4:          $\bar{u}_k = u_k + K_k(\bar{x}_k - x_k) + \alpha d_k$
5:          $\bar{x}_{k+1} \leftarrow$ Using $\bar{x}_k, \bar{u}_k$, (3)
6:      end for
7:      $J \leftarrow$ Using $X, U$
8:      if $J$ satisfies line search conditions then
9:          $X \leftarrow \bar{X}, U \leftarrow \bar{U}$
10:     else
11:         Reduce $\alpha$ and go to line 3
12:     end if
13: return $X, U, J$
14: end function
```

C. Termination Conditions

The "inner" DDP/iLQR solve is run until one of the following termination conditions is met

- The cost decrease between iterations $J_{\text{prev}} - J$ is less than some intermediate tolerance, $\epsilon_{\text{intermediate}}$. This is the typical exit criteria.

- The feedforward gains go to zero. We compute the average maximum of the normalized gains:

$$\nabla_{\text{ilqr}} = \frac{1}{N-1} \sum_{k=0}^{N-1} \frac{\|d_k\|_\infty}{|U_k|+1} \tag{56}$$

- The solver hits a maximum number of iterations, $i_{\text{ilqr}}^{\max}$

Algorithm 3 Iterative LQR

```
1: Initialize $x_0, U,$ tolerance
2: $X \leftarrow$ Simulate from $x_0$ using $U$, (3)
3: function iLQR($X, U$)
4:     $J \leftarrow$ Using $X, U$
5:     do
6:         $J^- \leftarrow J$
7:         $K, d, \Delta V \leftarrow$ BackwardPass($X, U$)
8:         $X, U, J \leftarrow$ ForwardPass($X, U, K, d, \Delta V, J^-$)
9:     while $|J - J^-| >$ tolerance
10: return $X, U, J$
11: end function
```

D. Augmented Lagrangian Update

One the inner solve hits one of the termination conditions listed in section III-C, the dual variables are updated according to,

$$\lambda_{k_i}^+ = \begin{cases} \lambda_{k_i} + \mu_{k_i} c_{k_i}(x_k^*, u_k^*) & i \in \mathcal{E}_k \\ \max(0, \lambda_{k_i} + \mu_{k_i} c_{k_i}(x_k^*, u_k^*)) & i \in \mathcal{I}_k, \end{cases} \tag{57}$$

and the penalty is increased monotonically according to the schedule,

$$\mu_{k_i}^+ = \phi \mu_{k_i}, \tag{58}$$

where $\phi > 1$ is a scaling factor.

After experimenting with various different heuristic schemes for updating the multipliers and the penalty parameters, we have found that the most naive approach, that is updating them every outer loop iteration, is by far the most reliable approach. It's possible that better performance may be achieved by skipping penalty updates, or only updating some of penalty parameters (e.g. the ones corresponding to active constraints), but we found that the most basic approach works the best on the largest variety of problems.

E. Hyperparameters

For convenience, we summarize all the hyperparameters in AL-iLQR, splitting them between those used for the inner unconstrained iLQR solve and those used by the outer Augmented Lagrangian solver.

IV. Improvements

Here we provide some improvements to the AL-iLQR algorithm derived in the previous section. These improvements are all incorporated into the ALTRO algorithm, summarized in Algorithm 4 and implemented in TrajectoryOptimization.jl.

Algorithm 4 ALTRO

```
1: procedure
2:     Initialize $x_0, U,$ tolerances; $\tilde{X}$
3:     if Infeasible Start then
4:         $X \leftarrow \tilde{X}$, $s_{0:N-1} \leftarrow$ from (74)
5:     else
6:         $X \leftarrow$ Simulate from $x_0$ using $U$
7:     end if
8:     $X, U, \lambda \leftarrow$ AL-iLQR($X, U,$ tol.)
9:     $(X, U, \lambda) \leftarrow$ Projection($(X, U, \lambda),$ tol.)
10: return $X, U$
11: end procedure
```

A. Square Root Backward Pass

Augmented Lagrangian methods make rapid convergence on constraint satisfaction, but only as long as the penalty terms are updated at every outer loop iteration. This can quickly lead to very large penalty parameters, resulting in severe numerical ill-conditioning. To help mitigate this issue and make AL-iLQR more numerically robust we derive a square-root backward pass inspired by the square-root Kalman filter.

1) Background: To begin, we provide some background on matrix square-roots. The Cholesky factorization of a square positive-definite matrix G factors the matrix into two upper-triangular matrices $G = U^T U$, where the upper-triangular matrix factor U can be considered a "square root" of the matrix G. We denote this matrix square root as $U = \sqrt{G}$.

Also in terms of background, the QR factorization $F = QR$ returns an upper-triangular matrix R and an orthogonal matrix Q.

We now seek a method for computing $\sqrt{A+B}$ in terms of $\sqrt{A}$ and $\sqrt{B}$, where $A, B \in \mathbb{R}^{n \times n}$ are square positive-definite matrices. We begin by noting that

$$A + B = \begin{bmatrix} \sqrt{A}^T & \sqrt{B}^T \end{bmatrix} \begin{bmatrix} \sqrt{A} \\ \sqrt{B} \end{bmatrix} = F^T F \tag{59}$$

where $F \in \mathbb{R}^{2n \times n}$. Taking the QR factorization of F we get

$$\begin{aligned} A + B =& F^T F \\ =& R^T Q^T Q R \\ =& R^T R \end{aligned} \tag{60}$$

since $Q^T Q = I$ given that Q is an orthogonal matrix. Therefore we have that $R = \sqrt{A+B}$ since R is upper-triangular. We use $\text{QR}_R(\cdot)$ to denote the operation of taking the QR factorization of the argument and returning the upper-triangular factor.

The related equation $\sqrt{A-B}$ can be computed using successive rank-one downdates of $\sqrt{A}$ using the rows of $\sqrt{B}$. We denote this operation as DownDate($\sqrt{A}, \sqrt{B}$).

2) Derivation: The ill-condition of the backward pass is most significant in the Hessian of the cost-to-go, P_k. Our objective is to find an algorithm that only stores the square root of this matrix and never calculates it explicitly.

TABLE I
iLQR Hyperparameters

Symbol	Name	Description	Typical Value(s)	Importance
ϵ_{cost}	Cost tolerance	Convergenced when difference in cost between iterations $< \epsilon_{\text{cost}}$	[1e-2, 1e-4, 1e-8]	High
$\epsilon^{\text{ilqr}}_{\text{grad}}$	Gradient tolerance	Converged when $\nabla_{\text{ilqr}} < \epsilon^{ilqr}_{\text{grad}}$	1e-5	Med
i^{Iqr}_{max}	Max iterations	Maximum number of backward/forward pass iterations	[50,500]	Med
β_1	Line search l.b.	Lower bound criteria for (54). ↑ requires more progress be made	[1e-10,1e-8,1e-1]	Low
β_2	Line search u.b.	Upper bound criteria for (54). ↓ requires progress match expected	[1,10,20]	Low
$i^{\text{ls}}_{\text{max}}$	Line search iterations	Maximum number of backtracking line search iterations	$[5, 10, 20]$	Low
ρ_{init}	Initial regularization	Initial value for regularization of Q_{zz} in backward pass	0	Low
ρ_{max}	Max regularization	Any further increases will saturate. ↓ allows for less aggressive regularization	1e-8	Low
ρ_{min}	Min regularization	Any regularization below will round to 0.	1e-8	Low
ϕ_ρ	Reg. scaling	How much regularization is increased/decreased	(1,1.6,10)	Low
J_{max}	Max cost	Maximum cost allowed during rollout	1e8	Med

TABLE II
Augmented Lagrangian Hyperparameters

Symbol	Name	Description	Typical Value(s)	Importance
ϵ_{cost}	Cost tolerance	Converged when difference in cost between iterations ¡ ϵ_{cost}	[1e2,1e-4,1e-8]	High
ϵ_{uncon}	iLQR cost tolerance	Cost tolerance for intermediate iLQR solves. ↓ can speed up overall convergence by requiring less optimality at each inner solve, resulting in more frequent dual updates.	[1e-1,1e-3,1e-8]	High
c_{max}	Constraint tolerance	Convergence when maximum constraint violation $< c_{\text{max}}$	[1e-2,1e-4,1e-8]	High
i^{outer}_{max}	Outer loop iterations	Maximum number of outer loop updates	[10,30,100]	Med
μ_{max}	Max penalty	↑ allows for more outer loop iterations with good convergence, but may result in poor conditioning. ↓ may avoid ill-conditioning	1e-8	Low
ϕ_μ	Penalty scaling	↑ Increases penalty faster, potentially converges faster, but will eventually fail to converge if too high	(1,10,100]	Med
μ_{init}	Initial penalty	↑ more likely to remain feasible and find a feasible solution faster. ↓ makes the initial problem appear unconstrained, which may be an ideal initial guess for constrained problem.	[1e-4,1,100]	Very High

We begin by calculating the square root of the terminal cost-to-go:

$$S_N = \sqrt{P}_N = \text{QR}_R\left(\begin{bmatrix}(\ell_N)_x x \\ I_{\mu,N} c_N\end{bmatrix}\right) \tag{61}$$

where we define $S \in \mathbb{R}^{n\times n} = \sqrt{P}$ to be the upper-triangular square root of the Hessian of the cost-to-go.

All that is left is to find $\sqrt{P}_k$ as an expression in terms of $\sqrt{P}_{k+1}$. We start by finding the square roots of Q_{xx} and Q_{uu}:

$$Z_{xx} = \sqrt{Q_{xx}} \leftarrow \text{QR}_R\left(\begin{bmatrix}\sqrt{\ell}_{xx} \\ S' A \\ \sqrt{I_\mu} c_x\end{bmatrix}\right) \tag{62}$$

$$Z_{uu} = \sqrt{Q_{uu}} \leftarrow \text{QR}_R\left(\begin{bmatrix}\sqrt{\ell}_{uu} \\ S' B \\ \sqrt{I_\mu} c_u \\ \sqrt{\rho} I\end{bmatrix}\right). \tag{63}$$

The optimal gains are then trivially computed as

$$K = -Z_{uu}^{-1} Z_{uu}^{-T} Q_{ux} \tag{64}$$

$$d = -Z_{uu}^{-1} Z_{uu}^{-T} Q_u, \tag{65}$$

Here it's important to note that these equations should be computed using a linear solver (e.g. the "\" operator in MATLAB or Julia) sequentially, since these equations are trivially computed using backward or forward substitution, given that the square root factors are triangular.

The gradient (48) and change of the cost-to-go (49) are also computed with simple substitution:

$$p \;=\; Q_x \;+\; (Z_{uu}K)^T(Z_{uu}d) \;+\; K^T Q_u \;+\; Q_{xu} d \tag{66}$$

$$\Delta V = d^T Q_u + \frac{1}{2}(Z_{uu}d)^T(Z_{uu}d). \tag{67}$$

We note that (47) is a quadratic form that can be expressed as

$$\begin{aligned} P &= \begin{bmatrix} I \\ K\end{bmatrix}^T \begin{bmatrix} Q_{xx} & Q_{ux}^T \\ Q_{ux} & Q_{uu}\end{bmatrix}\begin{bmatrix} I \\ K\end{bmatrix} \\ P &= \begin{bmatrix} I \\ K\end{bmatrix}^T \begin{bmatrix} Z_{xx}^T & 0 \\ C^T & D^T\end{bmatrix}\begin{bmatrix} Z_{xx} & C \\ 0 & D\end{bmatrix}\begin{bmatrix} I \\ K\end{bmatrix} \\ &= \begin{bmatrix} Z_{xx} + CK \\ DK\end{bmatrix}^T \begin{bmatrix} Z_{xx} + CK \\ DK\end{bmatrix} \end{aligned} \tag{68}$$

where,

$$C = Z_{xx}^{-T} Q_{xu} \tag{69}$$

$$D = \sqrt{Z_{uu}^T Z_{uu} - Q_{ux} Z_{xx}^{-1} Z_{xx}^{-T} Q_{xu}} \tag{70}$$

$$= \text{DownDate}\left(Z_{uu}, Z_{xx}^{-T} Q_{xu}\right). \tag{71}$$

The square root of P_k is then

$$\sqrt{P}_k = \text{QR}_R\left(\begin{bmatrix} Z_{uu} + Z_{xx}^{-T} Q_{xu} K \\ \text{DownDate}\left(Z_{uu}, Z_{xx}^{-T} Q_{xu}\right) \cdot K\end{bmatrix}\right) \tag{72}$$

B. Infeasible State Trajectory Initialization

Desired state trajectories can often be identified (e.g., from sampling-based planners or expert knowledge), whereas finding a control trajectory that will produce this result is usually challenging. Dynamically infeasible state trajectory initialization is enabled by introducing additional inputs to the dynamics with slack controls $s \in \mathbb{R}^n$,

$$x_{k+1} = f(x_k, u_k) + s_k, \tag{73}$$

to make the system artificially fully actuated.

Given initial state and control trajectories, $\tilde{X}$ and U, the initial infeasible controls $s_{0:N-1}$ are computed as the difference between the dynamics and desired state trajectory at each time step:

$$s_k = \tilde{x}_{k+1} - f(\tilde{x}_k, u_k) \tag{74}$$

The optimization problem (4) is modified by replacing the dynamics with (73). An additional cost term,

$$\sum_{k=0}^{N-1} \frac{1}{2} s_k^T R_s s_k, \tag{75}$$

and constraints $s_k = 0,\ k{=}0, \dots, N{-}1$ are also added to the problem. Since $s_k = 0$ at convergence, a dynamically feasible solution to (4) is still obtained.

C. Optimizing Quaternions

When dealing with robotic systems with arbitrary orientation, such as satellites, airplanes, quadrotors, etc., the 3D orientation must be optimized. Rotations present a unique challenge for optimization since all vector parameterizations of rotations are either underparameterized (e.g. Euler angles) or subject to constraints (e.g. rotation matrix or quaternions). Quaternions are often considered the best parameterization of rotation, since they only use four parameters and have a number of desireable qualities that make them attractive for numerical computation. We begin with some background on quaternions, along with our notation conventions for quaternion operations, and then present a method for handling quaternions within AL-iLQR.

1) Background and Notation: The unit quaternion is a four-parameter non-singular representation of rotations comprising a vector part $z \in \mathbb{R}^3$ and a scalar part $s \in \mathbb{R}$. To represent quaternion operations as linear-algebraic expressions, the following conventions are used: A quaternion q encodes the rotation from a vehicle's body frame to the world frame. The equivalent rotation matrix is given by,

$$R(q) = I + 2z^{\times}(z^{\times} + sI), \tag{76}$$

where I is the 3×3 identity matrix and $z^{\times}$ denotes the 3×3 skew-symmetric cross-product matrix:

$$z^{\times} = \begin{bmatrix} 0 & -z_3 & z_2 \\ z_3 & 0 & -z_1 \\ -z_2 & z_1 & 0 \end{bmatrix}. \tag{77}$$

Quaternion multiplication is denoted $q_2 \otimes q_1$, and the quaternion conjugate, representing the opposite rotation, is denoted q^*. For $r \in \mathbb{R}^3, \hat{r}$ is a quaternion with $s = 0$ and $z = r$. $H \in \mathbb{R}^{4\times3}$ is the matrix "hat" operator, such that $\hat{r} = Hr$ and $r = H^T\hat{r}$. The matrix $M(q) \in \mathbb{R}^{4\times4}$ is defined such that $M(q_2)q_1 = q_2 \otimes q_1$.

Quaternion rotations require special consideration in optimization problems to properly account for the unit-norm constraint. We define $\phi \in \mathbb{R}^3$ to be a three-dimensional differential rotation, such that $\hat{\phi} = \delta q$, where $q = q_0 \otimes \delta q$, when δq is a small rotation. We seek the Jacobian $G(q) \in \mathbb{R}^{3\times4}$ that maps from the quaternion to our three-dimensional differential, ϕ. From above, we have $q = q_0 \otimes \delta q = q_0 \otimes \hat{\phi} = M(q_0)H\phi$, so $G(q_0)^T = M(q_0)H$.

If $h(q) : q \mapsto \mathbb{R}^p$ is a function that maps a quaternion to a real-valued vector, the linearization of $h(q)$ gives

$$\delta h \approx G(q)\frac{\partial h(q)}{\partial q}G(q)^T\phi. \tag{78}$$

Similarly, for a real-valued function $g(q) : q \mapsto \mathbb{R}$, the second-order Taylor series expansion is

$$\delta g \approx \frac{\partial g(q)}{\partial q}G(q)^T\phi + \frac{1}{2}\phi^T G(q)\frac{\partial^2 g(q)}{\partial q^2}G(q)^T\phi \tag{79}$$

2) Quaterion AL-iLQR: We now present modifications to AL-iLQR in order to optimize states that contain quaternions. With standard commercial solvers, like those used in direct trajectory optimization methods, quaternions can only be treated as vectors in $\mathbb{R}^4$, often resulting in poor performance. With AL-iLQR we can modify the algorithm to correctly optimize with respect to a unit quaternion. Extensions of this method to states with multiple quaternions are easily obtained through simple concatenation.

Let $x \in \mathbb{R}^n$ $(n \geq 4)$ be a state vector containing a unit quaternion of the form,

$$x = \begin{bmatrix} y \\ q \end{bmatrix} \tag{80}$$

where $y \in \mathbb{R}^{n-4}$ is a standard vector and q is the quaternion. Using the results from above, the difference between two states, x and x_0, is

$$\delta x = \begin{bmatrix} \delta y \\ \phi \end{bmatrix} = \begin{bmatrix} y - y_0 \\ H^T(q_0^* \otimes q) \end{bmatrix} \in \mathbb{R}^{n-1}, \tag{81}$$

from which we get the Jacobian,

$$G(x) = \begin{bmatrix} I_{n-4} & 0 \\ 0 & G(q) \end{bmatrix} \tag{82}$$

where $G(x) \in \mathbb{R}^{n-1\times n}$. The next sections detail how to modify the AL-iLQR backward and forward passes to correctly take expansions with respect to quaternion states.

Backward pass: The backward pass of iLQR linearizes the dynamics at each time step and approximates the cost-to-go as a quadratic function by taking the second-order Taylor series expansion of the cost-to-go. $P_N \in \mathbb{S}^n_{++}$, $p_N \in \mathbb{R}^n$ are the Hessian and gradient of the terminal cost-to-go and the corrected terms are,

$$\tilde{P}_N = G(x_N)P_N G(x_N)^T \tag{83}$$

$$\tilde{p}_N = G(x_N)p_N \tag{84}$$

Q_{xx}, Q_{ux}, and Q_x are the terms from the approximate quadratic expansion of the action-value function at time step k that must be modified. The corrected terms from the expansion are:

$$\tilde{Q}_{xx} = G(x_k) Q_{xx} G(x_k)^T \tag{85}$$
$$\tilde{Q}_{ux} = Q_{ux} G(x_k)^T \tag{86}$$
$$\tilde{Q}_{x} = G(x_k) Q_{x}. \tag{87}$$

The rest of the backward pass proceeds as normal, using the corrected expansion terms.

Forward pass: The only required modification to the forward pass is the correct calculation of the difference between states x and x_0 using (81). The modified change in control, where $K \in \mathbb{R}^{m\times(n-1)}$ and $d \in \mathbb{R}^m$ are the feedback and feedforward gains from the backward pass, is:

$$\delta u_k = K_k \delta x_k + d_k. \tag{88}$$

Objective: Directly subtracting states that contain quaternions (or any rotational representation) is incorrect. A good alternative is to again use the state difference Jacobian, as demonstrated in the following objective,

$$\begin{aligned}&\tilde{J}(X,U) = \\ &(x_N - x_f)^T G(x_N)^T \tilde{W}_N G(x_N)(x_N - x_f) \\ &+ \Delta t \sum_{k=0}^{N-1} (x_k - x_f)^T G(x_k)^T \tilde{W}_k G(x_k)(x_k - x_f) \\ &+ (u - u_r)^T V_k (u - u_r)\end{aligned} \tag{89}$$

where $\tilde{W} \in \mathbb{S}_+^{n-1}, V \in \mathbb{S}_{++}^m$ are cost matrices. Here the three-parameter angular representation ϕ is penalized.

D. Solution Polishing

Augmented Lagrangian methods only make good progress on constraints as long as the penalty terms are updated. However, the penalties can only be updated a finite number of times before the penalties result in severe ill-conditioning, as discussed previously. As a result, augmented Lagrangian methods suffer from slow "tail" convergence, or convergence near the optimal solution. Active-set methods, on the other hand, exhibit quadratic convergence near the optimal solution. We present an active-set projection method to rapidly converge on the constraints once the convergence of AL-iLQR slows down.

This final solution polishing method solves the following optimization problem:

$$\begin{aligned}\underset{\delta z}{\text{minimize}} \quad & \delta z^T H \delta z \\ \text{subject to} \quad & D\delta z = d\end{aligned} \tag{90}$$

where $z \in \mathbb{R}^{Nn\times(N-1)m}$ is the concatenated vector of the states and controls at all time steps, H is the Hessian of the cost, and D, d are the linearized active constraints. Constraints are considered active if the constraint violation is less than some small positive value $\epsilon_{\text{constraint}}$.

This problem essentially projects the solution from AL-iLQR onto the constraint manifold, while minimizing impact to the cost. Algorithm 5 takes successive Newton steps δz, only updating the constraint Jacobian D when the convergence rate r drops below a threshold, allowing re-use of the same matrix factorization S for inexpensive linear system solutions. Further, this algorithm can be implemented in a sequential manner [4] that does not require building large matrices, making it amenable to embedded systems.

Algorithm 5 Projection
```
 1: function PROJECTION($Y$, tol.)
 2:     $H^{-1} \leftarrow$ invert Hessian of objective
 3:     while $\|d\|_\infty >$ tol. do
 4:         $d, D \leftarrow$ linearize active constraints
 5:         $S \leftarrow \sqrt{DH^{-1}D^T}$
 6:         $v \leftarrow \|d\|_\infty$
 7:         $r \leftarrow \infty$
 8:         while $v >$ tol. and $r >$ conv. rate tol. do
 9:             $Y, v^+ \leftarrow$ LINESEARCH($Y, S, H^{-1}, D, d, v$)
10:             $r \leftarrow \log v^+ / \log v$
11:         end while
12:     end while
13: return $Y$
14: end function
```

Algorithm 6 Projection Line Search
```
 1: function LINESEARCH($Y, S, H^{-1}, D, d, v_0$)
 2:     Initialize $\alpha, \gamma$
 3:     while $v > v_0$ do
 4:         $\delta Y_p \leftarrow H^{-1}D^T(S^{-1}S^{-T}d)$
 5:         $\bar{Y}_p \leftarrow Y_p + \alpha \delta Y_p$
 6:         $d \leftarrow$ UPDATECONSTRAINTS($\bar{Y}_p$)
 7:         $v \leftarrow \|d\|_\infty$
 8:         $\alpha \leftarrow \gamma\alpha$
 9:     end while
10: return $\bar{Y}, v$
11: end function
```

Appendix

References

[1] T. A. Howell, B. E. Jackson, and Z. Manchester, "ALTRO: A Fast Solver for Constrained Trajectory Optimization," in *IEEE/RSJ International Conference on Intelligent Robots and Systems*, Macau, China, Nov. 2019.

[2] B. E. Jackson, T. Punnoose, D. Neamati, K. Tracy, and R. Jitosho, "ALTRO-C: A Fast Solver for Conic Model-Predictive Control," in *2021 IEEE International Conference on Robotics and Automation (ICRA)*, Xi'an, China, Jun. 2021, p. 8.

[3] B. E. Jackson, K. Tracy, and Z. Manchester, "Planning With Attitude," *IEEE Robotics and Automation Letters*, pp. 1–1, 2021.

[4] C. V. Rao, S. J. Wright, and J. B. Rawlings, “Application of Interior-Point Methods to Model Predictive Control,” en, *Journal of Optimization Theory and Applications*, vol. 99, no. 3, pp. 723–757, Dec. 1998.