Constrained Unscented Dynamic Programming

Brian Plancher, Zachary Manchester, and Scott Kuindersma

Abstract—Differential Dynamic Programming (DDP) has become a popular approach to performing trajectory optimization for complex, underactuated robots. However, DDP presents two practical challenges. First, the evaluation of dynamics derivatives during optimization creates a computational bottleneck, particularly in implementations that capture second-order dynamic effects. Second, constraints on the states (e.g., boundary conditions, collision constraints, etc.) require additional care since the state trajectory is implicitly defined from the inputs and dynamics. This paper addresses both of these problems by building on recent work on Unscented Dynamic Programming (UDP)—which eliminates dynamics derivative computations in DDP—to support general nonlinear state and input constraints using an augmented Lagrangian. The resulting algorithm has the same computational cost as first-order penalty-based DDP variants, but can achieve constraint satisfaction to high precision without the numerical ill-conditioning associated with penalty methods. We present results demonstrating its favorable performance on several simulated robot systems including a quadrotor and 7-DoF robot arm.

I. INTRODUCTION

Trajectory optimization algorithms [1] have become a powerful set of tools for synthesizing dynamic motions for complex robots [2], [3], [4], [5], [6]. Most robot tasks of interest are described by large sets of state constraints relating to, e.g., obstacle avoidance, reaching a desired goal state, and maintaining contact with the environment. Direct transcription methods for trajectory optimization parameterize both the state and input trajectories, allowing them to easily handle such constraints by formulating a large (and sparse) nonlinear program that can be solved using off-the-shelf Sequential Quadratic Programming (SQP) packages. In contrast, Differential Dynamic Programming (DDP) parameterizes only the input trajectory and uses Bellman’s optimality principle to iteratively solve a sequence of much smaller optimization problems [7], [8]. DDP and its variants have received increased attention due to growing empirical evidence that online planning is possible for high-dimensional robots [4], [9]. However, constraints are typically addressed approximately by augmenting the native cost with constraint penalty functions. This paper introduces a variant of DDP that captures nonlinear constraints on states and inputs with high accuracy while maintaining favorable convergence properties.

The classical DDP algorithm proceeds iteratively by simulating the dynamics forward from an initial state and input trajectory, then computing improved affine feedback terms backwards in time using local quadratic approximations of the cost-to-go. This procedure requires second derivatives of the dynamics (rank-three tensors) to be evaluated along the trajectory, which is the most expensive part of the algorithm. This computational bottleneck led to the development of the iterative Linear Quadratic Regulator (iLQR) algorithm [10] that uses only the first derivatives of the dynamics to reduce computation time at the expense of slower convergence. The relationship between DDP and iLQR is similar to the relationship between Newton’s method and the Gauss-Newton method in that the Hessian is approximated using only the Jacobian [11].

Recent work has proposed a completely derivative-free variant of DDP that uses a deterministic sampling scheme inspired by the unscented Kalman filter. The resulting algorithm, called Unscented Dynamic Programming (UDP) [12], has the same computational complexity per-iteration as iLQR with finite-difference derivatives, but provides empirical convergence approaching that of the full second-order DDP algorithm. The primary contribution of this paper is an extension of UDP that supports nonlinear constraints on states and inputs.

Several authors have proposed methods for adding constraints to DDP methods. For the special case of box constraints on input variables, quadratic programs (QPs) can be solved in the backwards pass [13], [14]. In this setting, bounds are placed on the inputs and enforced by projecting the feedback terms onto the constraint surface. This approach is able to enforce hard constraints on the inputs but does not support state constraints. Farshidian et al. [15] extend this from input constraints to equality constraints on the state and input through a similar projection framework while pure state constraints are still handled via penalty functions.

Designing effective penalty functions and continuation schedules can be difficult, often failing to satisfy constraints to high precision. Despite this, penalty methods have seen broad application in robotics. For example, van den Berg [16] uses exponential barrier cost terms in the LQR Smoothing algorithm to prevent a quadrotor from colliding with cylindrical obstacles. Barrier methods require particular care in cases where feasible initial guesses cannot be easily generated. It is common to increase penalty scaling coefficients on the constraint cost to eventually converge to a result that satisfies the constraints [15]. However, it is well known that these methods often lead to numerical ill-conditioning of the Hessian before reaching the desired constraint tolerance [17], [11]. In practice this can render trajectories infeasible or lead to collisions when run on hardware.

The augmented Lagrangian approach to solving constrained nonlinear optimization problems was conceived to
We define \( Q(\delta x, \delta u) \) to be the local change in the minimization argument in (3) under state and input perturbations, \( \delta x, \delta u \):

\[
Q(\delta x, \delta u) = \ell(x + \delta x, u + \delta u) + V(f(x + \delta x, u + \delta u)) - \ell(x, u) - V(f(x, u)).
\]

Taking the second-order Taylor expansion of \( Q \), we have:

\[
Q(\delta x, \delta u) \approx \frac{1}{2} \begin{bmatrix} 1
\delta x \\
\delta u \end{bmatrix}^T \begin{bmatrix}
0 & Q_x^T \\
Q_x & Q_{xx} & Q_{xu} \\
Q_u & Q_{ux} & Q_{uu} \end{bmatrix} \begin{bmatrix} 1 \\
\delta x \\
\delta u \end{bmatrix}, \tag{5}
\]

where the block matrices are computed as:

\[
\begin{align*}
Q_{xx} &= \ell_{xx} + f_x^T \nabla_{xx} f_x + V'_x \cdot f_{xx} \\
Q_{uu} &= \ell_{uu} + f_u^T \nabla_{uu} f_u + V'_u \cdot f_{uu} \\
Q_{xu} &= \ell_{xu} + f_x^T \nabla_{xu} f_u + V'_x \cdot f_{xu} \tag{6} \\
Q_x &= \ell_x + f_x^T V'_x \\
Q_u &= \ell_u + f_u^T V'_u.
\end{align*}
\]

Following the notation used elsewhere [4], we have dropped explicit time indices and used a prime to indicate the next timestep. Derivatives with respect to \( x \) and \( u \) are are denoted with subscripts. The rightmost terms in the equations for \( Q_{xx}, Q_{uu}, \) and \( Q_{xu} \) involve second derivatives of the dynamics, which are rank-three tensors. As mentioned previously, these tensor calculations are relatively expensive and are often omitted, resulting in the iLQR algorithm [10].

Minimizing equation (5) with respect to \( \delta u \) results in the following correction to the control trajectory:

\[
\delta u = -Q_{uu}^{-1} (Q_{ux} \delta x + Q_u) \equiv K\delta x + d, \tag{7}
\]

which consists of an affine term \( d \) and a linear feedback term \( K\delta x \). These terms can be substituted back into equation (5) to obtain a quadratic model of \( V \) at the previous timestep:

\[
\begin{align*}
\Delta V &= -\frac{1}{2} Q_{uu}^{-1} (Q_{ux} \delta x + Q_u) \\
V_x &= Q_x - Q_{u} Q_{uu}^{-1} Q_{ux} \\
V_{xx} &= Q_{xx} - Q_{xu} Q_{uu}^{-1} Q_{ux}.
\end{align*} \tag{8}
\]

Therefore, a backward update pass can be performed starting at the final state, \( x_N \), by setting \( V_N = \ell(f(x_N)) \) and iteratively applying the above computations. Once an update is computed a forward simulation pass is done to compute the new state trajectory using the updated controls. This forward-backward process is repeated until the algorithm converges within a specified tolerance.

DDP, like other variants of Newton’s method, can achieve quadratic convergence near a local optimum [8], [22]. However, care must be taken to ensure good convergence behavior from arbitrary initialization. First, a line search parameter, \( \alpha \), must be added to the forward pass to ensure a satisfactory decrease in cost. Second, a regularization term, \( \rho \), is added to \( Q_{uu} \) in equation (7) to ensure positive-definitiveness [23].

overcome the conditioning problems of penalty methods by adding a linear Lagrange multiplier term to the objective [11]. As pointed out by other researchers [18], these methods may be particularly well-suited to trajectory optimization problems as they allow the solver to temporarily traverse infeasible regions and aggressively move towards local optima before making incremental adjustments to satisfy constraints. A theoretical formulation of this method was proposed for use in the DDP context over two decades ago [19] and was later used to develop a hybrid-DDP algorithm [20], but this work appears to have received little attention in the robotics community. We compare our algorithm [20], but this work appears to have received little attention in the robotics community. We compare our constrained UDP algorithm against this method.

In the remainder of the paper, we review key concepts from DDP, augmented Lagrangian methods, and the unscented transform (Section II), introduce the constrained UDP algorithm (Section III), and describe our experimental results on an inverted pendulum, a quadrotor flying through a virtual forest, and a manipulator avoiding obstacles (Section IV). Several practical considerations are also discussed.

II. BACKGROUND

In the following subsections, we summarize key ideas from DDP, augmented Lagrangian methods, and the unscented transform, all of which form the basis for the constrained UDP algorithm described in Section III.

A. Differential Dynamic Programming

To begin we will assume a discrete-time nonlinear dynamical system of the form:

\[
x_{k+1} = f(x_k, u_k), \tag{1}
\]

where \( x \in \mathbb{R}^n \) is the system state and \( u \in \mathbb{R}^m \) is a control input. The goal is to find an input trajectory, \( U = \{u_0, \ldots, u_{N-1}\} \), that minimizes an additive cost function,

\[
J(x_0, U) = \ell(f(x_N)) + \sum_{k=0}^{N-1} \ell(x_k, u_k), \tag{2}
\]

where \( x_0 \) is the initial state and \( x_1, \ldots, x_N \) are computed by integrating forward the dynamics (1).

Using Bellman’s principle of optimality [21], we can define the optimal cost-to-go, \( V_k(x) \), by the recurrence relation:

\[
\begin{align*}
V_N(x) &= \ell_f(x_N) \\
V_k(x) &= \min_u \ell(x, u) + V_{k+1}(f(x, u)). \tag{3}
\end{align*}
\]

When interpreted as an update procedure, this relationship leads to classical dynamic programming algorithms [21]. However, the curse of dimensionality prevents direct application of dynamic programming to most systems of interest to the robotics community. In addition, while \( V_N(x) = \ell_f(x_N) \) often has a simple analytical form, \( V_k(x) \) will typically have complex geometry that is difficult to represent due to the nonlinearity of the dynamics (1). DDP avoids these difficulties by settling for local approximations to the cost-to-go along a trajectory.
B. Unscented Dynamic Programming

UDP replaces the gradient and Hessian calculations in equation (6) with approximations computed from a set of sample points [12]. To apply the unscented transform in an estimation context, one samples the state distribution at a small number of strategically chosen points, known as sigma points, that are then propagated through the nonlinear dynamics. The transformed points are then used to calculate a new mean and covariance [24]. In the DDP context, we map sample points backward in time through the nonlinear to approximate the cost-to-go at earlier timesteps.

To compute the derivatives of \( V(f(x, u)) \) appearing in the Hessian (6), a set of \( 2(n+m) \) sample points is generated from the columns of the following matrix:

\[
L = \text{chol} \left( \begin{bmatrix} V'_{xx} & 0 \\ V'_{ux} & \ell_{uu} \end{bmatrix}^{-1} \right).
\]

(9)

Each column, \( L_i \), and its negative are scaled by a constant factor, \( \beta \), and added to the vector \( [x'; u] \) (again, using the shorthand \( x = x_k, u = u_k, x' = x_{k+1} \)):

\[
\begin{bmatrix} \tilde{x}'_i \\ \tilde{u}_i \end{bmatrix} = \begin{bmatrix} x' \\ u \end{bmatrix} + \beta L_i \begin{bmatrix} \tilde{x}'_{i+m+n} \\ \tilde{u}_{i+m+n} \end{bmatrix} = \begin{bmatrix} x' \\ u \end{bmatrix} - \beta L_i. \quad (10)
\]

The samples are then propagated backwards through the dynamics such that \( \tilde{x}_i = f^{-1}(\tilde{x}'_i, \tilde{u}_i) \). A backwards dynamics function can always be defined for a continuous-time dynamical system by simply integrating backwards in time using, for example, a Runge-Kutta method. Note that this problem is not well posed for dynamics that include rigid contact unless certain smoothing approximations are made [25].

Using these sample points, the Hessian in equation (6) becomes:

\[
\begin{bmatrix} Q_{xx} & Q_{xu} \\ Q_{ux} & Q_{uu} \end{bmatrix} = M^{-1} + \begin{bmatrix} \ell_{xx} & \ell_{ux} \\ \ell_{ux} & 0 \end{bmatrix},
\]

(11)

\[M = \frac{1}{2\beta^2} \sum_{i=1}^{2(n+m)} \begin{bmatrix} \tilde{x}'_i \\ \tilde{u}_i \end{bmatrix} - \begin{bmatrix} x' \\ u \end{bmatrix} \begin{bmatrix} \tilde{x}'_{i+m+n} \\ \tilde{u}_{i+m+n} \end{bmatrix} \quad \begin{bmatrix} x' \\ u \end{bmatrix} \begin{bmatrix} \tilde{x}_i \\ \tilde{u}_i \end{bmatrix} - \begin{bmatrix} x' \\ u \end{bmatrix} \begin{bmatrix} \tilde{x}'_{i} \\ \tilde{u}'_{i} \end{bmatrix} - \begin{bmatrix} x' \\ u \end{bmatrix} \begin{bmatrix} \tilde{x}'_{i} \\ \tilde{u}'_{i} \end{bmatrix} \begin{bmatrix} x' \\ u \end{bmatrix} \begin{bmatrix} \tilde{x}_i \\ \tilde{u}_i \end{bmatrix} - \begin{bmatrix} x' \\ u \end{bmatrix} \begin{bmatrix} \tilde{x}'_{i+m+n} \\ \tilde{u}'_{i+m+n} \end{bmatrix}, \]

where the matrix \( D \) is defined as follows:

\[
D = \begin{bmatrix} \tilde{x}_i - \tilde{x}_{i+m+n} \\ \tilde{u}_i - \tilde{u}_{i+m+n} \\ \tilde{x}_{m+n} - \tilde{x}_{2(m+n)} \\ \tilde{u}_{m+n} - \tilde{u}_{2(m+n)} \end{bmatrix}. \quad (13)
\]

This procedure is mathematically equivalent to a centered finite difference.

C. Augmented Lagrangian Methods

A natural approach to approximately enforcing constraints in optimization algorithms is to add a quadratic penalty to the cost function. Suppose for the sake of presentation that we wish to solve the generic minimization problem:

\[
\text{minimize} \quad g(z) \\
\text{subject to} \quad c_i(z) = 0 \quad i \in \mathcal{E}, \quad (14)
\]

where \( g(z) \) and \( c_i(z) \) are smooth nonlinear functions. We will return to the specific DDP context and describe how inequality constraints are handled in the next section.

Penalty methods for solving constrained optimization problems start by defining a new cost function,

\[
g_p(z; \mu) = g(z) + \frac{\mu}{2} \sum_{i \in \mathcal{E}} c_i(z)^T c_i(z), \quad (15)
\]

where \( \mu \) is a scalar weighting parameter. As \( \mu \to \infty \), the minimizing value of \( g_p(z; \mu) \) will converge toward satisfaction of the constraints [11]. While \( \mu \) often does not have to grow unbounded for a solution to be found within a given tolerance, numerical issues are still prevalent since the condition number of the Hessian of \( g_p \) grows with \( \mu \) (see [11] for additional details). It is important to note that, while we focus on quadratic penalties in this paper, there are a wide variety of other penalty functions that can be used. For example, \( L_1 \) loss functions have also been used for collision-free path planning in robotic arms and humanoids [26].

Augmented Lagrangian solvers add a term to \( g_p \) with the goal of improving constraint satisfaction without needing to aggressively increase \( \mu \). The augmented Lagrangian function,

\[
L_A(z; \mu, \lambda) = g(z) + \frac{\mu}{2} \sum_{i \in \mathcal{E}} c_i(z)^T c_i(z) + \sum_{i \in \mathcal{E}} \lambda_i^T c_i(z), \quad (16)
\]

Given initial values for \( \mu \) and \( \lambda \), an unconstrained minimization is performed after which \( \mu \) and \( \lambda \) are updated. As in penalty methods, \( \mu \) is systematically increased across these “major iterations” using a predefined schedule, but the update for \( \lambda \) has a specific form that we now describe. The update to \( \lambda \) at major iteration \( j \) can be derived by considering the first-order necessary conditions evaluated at an approximate minimizer, \( z_j \):

\[
0 \approx \nabla_z L_A(z_j; \mu_j, \lambda^j) = \nabla_z g(z_j) - \sum_{i \in \mathcal{E}} \left[ \lambda_i^j - \mu_j c_i(z_j) \right] \nabla_z c_i(z_j). \quad (17)
\]

Recall that the first-order necessary conditions for a local solution, \( z^* \), \( \lambda^* \), of the original constrained optimization problem is given by differentiating the (true) Lagrangian [11]:

\[
0 = \nabla_z g(z^*) - \sum_{i \in \mathcal{E}} \lambda_i^* \nabla_z c_i(z^*). \quad (18)
\]

Comparing (17) and (18), a natural update rule for \( \lambda \) arises:

\[
\lambda_i^{j+1} \leftarrow \lambda_i^j - \mu_j c_i(z_j) \quad \forall i \in \mathcal{E}. \quad (19)
\]
It can be shown that given $\lambda^*$, the solution, $z^*$, of (14) is a strict local minimizer of (15)–(6), which we denote evaluated. A modified version of when inequality constraints are satisfied. Including a Gauss-Newton approximation of the constraint terms:

$$I_k c(x_k, u_k) + \text{diag}(\lambda_k) c(x_k, u_k),$$

(20)

where $c(x, u)$ is the vertical concatenation of all equality and inequality constraints:

$$c_i(x, u) = 0, \quad i \in E,$$

$$c_i(x, u) \geq 0, \quad i \in I.$$  

(21)

Note that we are using separate $\mu^i_k$ and $\lambda^i_k$ for each constraint at each timestep.

To handle inequality constraints, we follow [18] and define $I_{\mu_k}$ as a diagonal matrix that encodes the active constraints:

$$I_{\mu_k}(i, i) = \begin{cases} 
\mu^i_k & \text{if } i \in E \text{ or } c_i(x_k, u_k) < 0 \text{ or } \lambda^i_k > 0 \\
0 & \text{otherwise}
\end{cases}.$$  

(22)

Including $I_{\mu_k}$ in (20) ensures that penalties are not incurred when inequality constraints are satisfied.

At each timestep during the backward pass of the UDP algorithm, the constraint functions and their gradients are evaluated. A modified version of $Q(\delta x, \delta u)$ from equations (13)–(16), which we denote $\hat{Q}$, is then defined to include a Gauss-Newton approximation of the constraint terms:

$$\hat{Q}_{xx} = Q_{xx} + \frac{\partial c(x, u)}{\partial x} I_{\mu} \frac{\partial c(x, u)}{\partial x},$$

$$\hat{Q}_{uu} = Q_{uu} + \frac{\partial c(x, u)}{\partial u} I_{\mu} \frac{\partial c(x, u)}{\partial u} + \rho I,$$

$$\hat{Q}_{xu} = Q_{xu} + \frac{\partial c(x, u)}{\partial x} I_{\mu} \frac{\partial c(x, u)}{\partial u},$$

(23)

$$\hat{Q}_x = Q_x + c(x, u) I_{\mu} \frac{\partial c(x, u)}{\partial x} + \text{diag}(\lambda) \frac{\partial c(x, u)}{\partial x},$$

$$\hat{Q}_u = Q_u + c(x, u) I_{\mu} \frac{\partial c(x, u)}{\partial u} + \text{diag}(\lambda) \frac{\partial c(x, u)}{\partial u},$$

where we have included a regularization parameter $\rho > 0$. Equations (17) and (18) are used to compute the feedback policy during the backwards pass as usual.

The algorithm proceeds by running the inner augmented UDP algorithm until convergence, and then updating $\mu$ and $\lambda$ according to a set schedule until the desired feasibility tolerance is achieved. The full constrained UDP procedure is summarized in Algorithm 1.

A. Updating $\mu$ and $\lambda$

As is typically done in augmented Lagrangian methods, we update $\lambda$ only if the constraint violation of the local minimizer is less than a threshold value, $\phi$, and otherwise update $\mu$. The parameter $\phi$ is updated according to a predefined schedule to help guide the algorithm toward a solution while avoiding large increases in $\mu$ early on. While there are many different variations on this schedule in the literature, most variations suggest a monotonically decreasing schedule for $\phi$, which results in a monotonically increasing $\mu$ [27], [11].

In our initial experiments, we found that input constraints and state constraints were often in competition with each other. Therefore, a global $\mu$ performed poorly as it would not allow the various constraints to be balanced appropriately. In practice, we found that using a separate $\mu$ and $\lambda$ for each constraint at each timestep allows the algorithm more flexibility in adjusting the penalty parameters and improves convergence. Unfortunately, this flexibility sometimes comes at the price of additional major iterations, where adjacent timesteps pass large inputs back-and-forth until both $\mu$ values are increased sufficiently. Despite this drawback, we found that the additional flexibility was required for convergence on hard problems. We also found that early termination of major iterations (by using loose cost convergence criteria) often allowed the algorithm to converge to a final solution faster.

IV. EXAMPLES

In this section, three numerical examples are provided to demonstrate the performance of constrained UDP. We compare UDP and iLQR using both penalty and augmented Lagrangian formulations. All of the algorithms are implemented in MATLAB, and in each setting the same scheduling of updates to $\mu$ and $\lambda$ are used.

A. Inverted Pendulum

We first consider the classic inverted-pendulum system and swing-up task. We define the state vector to be $x = [y, \theta, \dot{y}, \dot{\theta}]^T$, where $y$ is the translation of the cart and $\theta$ is the angle of the pendulum measured from the downward equilibrium. The initial state is $x_0 = [0, 0, 0, 0]^T$ and the goal state is $x_g = [0, \pi, 0, 0]^T$. We use a quadratic cost function of the form:

$$J = \frac{1}{2} (x_N - x_g)^T Q (x_N - x_g) +$$

$$\frac{1}{2} \sum_{k=0}^{N-1} \frac{1}{2} (x_k - x_{g})^T Q (x_k - x_g) + \frac{1}{2} u_k^T R u_k,$$

(24)

where $Q = 0.1 \times I_{4 \times 4}$, $R = 0.01$, and $Q_N = 1000 \times I_{4 \times 4}$. We set the number of knot points to $N = 120$ with a time horizon of 4 seconds and use a 3rd-order Runge-Kutta
Algorithm 1 Constrained UDP
1: Initialize \( \mu, \lambda, \phi, \rho, \bar{U}, x_0 \)
2: Perform forward pass to compute \( X = \{x_0, x_1, \ldots, x_N\} \)
3: \( \mathbf{while} \ \max(c) > \epsilon_c \ \mathbf{do} \)
4: \( \forall \text{cost not converged do} \)
5: Compute \( V_N \) and derivatives
6: \( \text{for } k = N-1, \ldots, 0 \ \text{do} \)
7: \( \{11\}, \{12\}, \{23\} \rightarrow Q^k \)
8: if \( Q_{uu}^k \) is invertible then
9: \( \{7\} \rightarrow K_k, d_k \)
10: \( \{8\} \rightarrow V_k \) and derivatives
11: else
12: Increase \( \rho \) go to line 6
13: end if
14: \( \alpha = 1 \)
15: \( \tilde{x}_0 = x_0 \)
16: \( \text{for } k = 0, \ldots, N-1 \ \text{do} \)
17: \( \tilde{u}_k = u_k + \alpha d_k + K_k(\tilde{x}_k - x_k) \)
18: \( \tilde{x}_{k+1} = f(\tilde{x}_k, \tilde{u}_k) \)
19: end for
20: Compute \( J \) using \( (2) \) and \( \tilde{X}, \tilde{U} \)
21: if \( J \) satisfies line search criteria then
22: Update \( X \leftarrow \tilde{X}, U \leftarrow \tilde{U} \)
23: else
24: Reduce \( \alpha \) and go to line 17
25: end if
26: \( \forall \text{end while} \)
27: \( \text{for } k = 0, \ldots, N \ \text{do} \)
28: \( \text{for } i \in E \cup I \ \text{do} \)
29: if \( c_i^k < \phi_i^k \) then
30: Update \( \lambda_i^k \) using \( (19) \)
31: Reduce \( \phi_i^k \)
32: else
33: Increase \( \mu_i^k \)
34: end if
35: end for
36: end for
37: end while
38: end while

integration method. We initialized the algorithms with all of the states and controls set to 0 (which gives a very poor initial trajectory). We also enforced an input constraint of \( \pm 30 \) \( N \), and a final state constraint of \( x_N = x_g \). We ran optimizations at three different constraint tolerances: \( 1 e^{-2} \) (“low precision”), \( 1 e^{-4} \) (“medium precision”), and \( 5 e^{-7} \) (“high precision”). In all cases we set our intermediate cost convergence tolerance to \( 1 e^{-2} \) and our final iteration cost convergence tolerance to \( 1 e^{-6} \). We also set a maximum value of \( 1 e^{30} \) for \( \mu \). We found that allowing \( \mu \) to grow much larger than this degraded performance due to poor numerical conditioning. For both penalty-based and augmented UDP, \( \beta \) was set to \( 1 e^{-2} \).

Figure [1] provides some intuition for how the constrained UDP algorithm solves the inverted pendulum swing-up prob-

![Fig. 1. Cost and constraint violation per iteration for the constrained UDP algorithm for the inverted pendulum.](image-url)
methods may be sufficient, but UDP-A is superior when high-precision is desired.

### Low precision: max(c) < 1e-2, initial φ = 1e-1

<table>
<thead>
<tr>
<th>Method</th>
<th>Iters</th>
<th>Cost</th>
<th>c_x</th>
<th>c_u</th>
<th>μ_x</th>
<th>μ_u</th>
</tr>
</thead>
<tbody>
<tr>
<td>iLQR-P</td>
<td>254</td>
<td>1000.1</td>
<td>2.3e-6</td>
<td>5.8e-7</td>
<td>1e^-1</td>
<td>1e^-1</td>
</tr>
<tr>
<td>UDP-P</td>
<td>229</td>
<td>1022.4</td>
<td>1.2e-4</td>
<td>1.1e-6</td>
<td>1e^-8</td>
<td>1e^-8</td>
</tr>
<tr>
<td>iLQR-A</td>
<td>93</td>
<td>1001.0</td>
<td>8.2e-8</td>
<td>1.8e-9</td>
<td>1e^-6</td>
<td>1e^-6</td>
</tr>
<tr>
<td>UDP-A</td>
<td>140</td>
<td>999.6</td>
<td>8.8e-4</td>
<td>4.8e-4</td>
<td>1e^-6</td>
<td>1e^-6</td>
</tr>
</tbody>
</table>

### Medium precision: max(c) < 1e-4, initial φ = 1e-2

<table>
<thead>
<tr>
<th>Method</th>
<th>Iters</th>
<th>Cost</th>
<th>c_x</th>
<th>c_u</th>
<th>μ_x</th>
<th>μ_u</th>
</tr>
</thead>
<tbody>
<tr>
<td>iLQR-P</td>
<td>231</td>
<td>1111.9</td>
<td>1.7e-7</td>
<td>1.8e-8</td>
<td>1e^-1</td>
<td>1e^-1</td>
</tr>
<tr>
<td>UDP-P</td>
<td>258</td>
<td>1084.2</td>
<td>4.1e-5</td>
<td>3.4e-8</td>
<td>1e^-9</td>
<td>1e^-9</td>
</tr>
<tr>
<td>iLQR-A</td>
<td>98</td>
<td>1001.4</td>
<td>2.4e-9</td>
<td>7.7e-10</td>
<td>1e^-8</td>
<td>1e^-8</td>
</tr>
<tr>
<td>UDP-A</td>
<td>126</td>
<td>999.8</td>
<td>9.0e-8</td>
<td>3.7e-8</td>
<td>1e^-6</td>
<td>1e^-6</td>
</tr>
</tbody>
</table>

### High precision: max(c) < 5e-7, initial φ = 5e-3

<table>
<thead>
<tr>
<th>Method</th>
<th>Iters</th>
<th>Cost</th>
<th>c_x</th>
<th>c_u</th>
<th>μ_x</th>
<th>μ_u</th>
</tr>
</thead>
<tbody>
<tr>
<td>iLQR-P</td>
<td>231</td>
<td>1111.9</td>
<td>1.7e-7</td>
<td>1.8e-8</td>
<td>1e^-1</td>
<td>1e^-1</td>
</tr>
<tr>
<td>UDP-P</td>
<td>258</td>
<td>1084.2</td>
<td>4.1e-5</td>
<td>3.4e-8</td>
<td>1e^-9</td>
<td>1e^-9</td>
</tr>
<tr>
<td>iLQR-A</td>
<td>98</td>
<td>1001.4</td>
<td>2.4e-9</td>
<td>7.7e-10</td>
<td>1e^-8</td>
<td>1e^-8</td>
</tr>
<tr>
<td>UDP-A</td>
<td>117</td>
<td>999.8</td>
<td>4.6e-8</td>
<td>1.3e-7</td>
<td>1e^-6</td>
<td>1e^-6</td>
</tr>
</tbody>
</table>

**TABLE I**

INVERTED PENDULUM OPTIMIZATION RESULTS. SEE TEXT FOR DETAILS.

### B. Quadrotor

The objective for this example is to compute a trajectory that flies a quadrotor through a forest while avoiding collisions with trees. Our quadrotor plant has four inputs representing the thrust for each of the rotors and twelve states representing the x, y, z position, roll, pitch, and yaw angles, and their corresponding first derivatives. We again used a quadratic cost function with $Q = 0.1 \times I_{12 \times 12}$, $R = 0.01 \times I_{4x4}$, and $Q_N = 1000 \times I_{12 \times 12}$ and a 3rd-order Runge-Kutta method with $N = 120$ and a time horizon of 4 seconds. The algorithms were initialized with the controls set to 0, and an input constraint of $-10 \leq u \leq 10$, and a final state constraint of $x_N = x_f$. The algorithms were again tested at three different constraint tolerance values. For the unscented variants, $\beta$ was set to $1e^-4$. A screen shot of the final trajectory computed by the constrained UDP algorithm is shown in Figure 3.

![Quadrotor trajectory](image)

**Fig. 3.** A collision-free quadrotor trajectory computed using constrained UDP.

The full results are shown below in Table II. As in the previous example, all of the algorithms converged to the low-precision constraint tolerance, with the augmented Lagrangian variants requiring fewer iterations. For the medium-precision case, both iLQR methods failed to achieve the required tolerance. We hypothesize that the higher-order information provided by the UDP backup procedure is responsible for its improved convergence [12]. For tight constraint tolerances, only UDP-A is able to find a feasible trajectory.

### Low precision: max(c) < 1e-2, initial φ = 1e-1

<table>
<thead>
<tr>
<th>Method</th>
<th>Iters</th>
<th>Cost</th>
<th>c_x</th>
<th>c_u</th>
<th>μ_x</th>
<th>μ_u</th>
</tr>
</thead>
<tbody>
<tr>
<td>iLQR-P</td>
<td>272</td>
<td>758.6</td>
<td>2.8e-3</td>
<td>2.9e-6</td>
<td>1e^-6</td>
<td>1e^-6</td>
</tr>
<tr>
<td>UDP-P</td>
<td>125</td>
<td>727.4</td>
<td>2.0e-3</td>
<td>7.8e-6</td>
<td>1e^-7</td>
<td>1e^-7</td>
</tr>
<tr>
<td>iLQR-A</td>
<td>109</td>
<td>712.4</td>
<td>3.7e-3</td>
<td>3.2e-6</td>
<td>1e^-6</td>
<td>1e^-6</td>
</tr>
<tr>
<td>UDP-A</td>
<td>115</td>
<td>707.3</td>
<td>7.6e-3</td>
<td>2.3e-6</td>
<td>1e^-6</td>
<td>1e^-6</td>
</tr>
</tbody>
</table>

### Medium precision: max(c) < 1e-4, initial φ = 1e-2

<table>
<thead>
<tr>
<th>Method</th>
<th>Iters</th>
<th>Cost</th>
<th>c_x</th>
<th>c_u</th>
<th>μ_x</th>
<th>μ_u</th>
</tr>
</thead>
<tbody>
<tr>
<td>iLQR-P</td>
<td>223</td>
<td>764.2</td>
<td>1.4e-4</td>
<td>3.2e-6</td>
<td>1e^-6</td>
<td>1e^-6</td>
</tr>
<tr>
<td>UDP-P</td>
<td>114</td>
<td>729.6</td>
<td>2.9e-5</td>
<td>2.2e-8</td>
<td>1e^-6</td>
<td>1e^-6</td>
</tr>
<tr>
<td>iLQR-A</td>
<td>253</td>
<td>712.6</td>
<td>3.3e-4</td>
<td>2.1e-6</td>
<td>1e^-6</td>
<td>1e^-6</td>
</tr>
<tr>
<td>UDP-A</td>
<td>158</td>
<td>708.8</td>
<td>9.1e-6</td>
<td>1.8e-6</td>
<td>1e^-6</td>
<td>1e^-6</td>
</tr>
</tbody>
</table>

### High precision: max(c) < 5e-6, initial φ = 5e-3

<table>
<thead>
<tr>
<th>Method</th>
<th>Iters</th>
<th>Cost</th>
<th>c_x</th>
<th>c_u</th>
<th>μ_x</th>
<th>μ_u</th>
</tr>
</thead>
<tbody>
<tr>
<td>iLQR-P</td>
<td>223</td>
<td>764.2</td>
<td>1.4e-4</td>
<td>3.2e-6</td>
<td>1e^-6</td>
<td>1e^-6</td>
</tr>
<tr>
<td>UDP-P</td>
<td>201</td>
<td>729.6</td>
<td>2.9e-5</td>
<td>2.2e-8</td>
<td>1e^-6</td>
<td>1e^-6</td>
</tr>
<tr>
<td>iLQR-A</td>
<td>254</td>
<td>744.9</td>
<td>9.1e-6</td>
<td>8.8e-6</td>
<td>1e^-6</td>
<td>1e^-6</td>
</tr>
<tr>
<td>UDP-A</td>
<td>149</td>
<td>708.8</td>
<td>5.3e-6</td>
<td>1.7e-6</td>
<td>1e^-6</td>
<td>1e^-6</td>
</tr>
</tbody>
</table>

**TABLE II**

QUADROTOR OPTIMIZATION RESULTS. SEE TEXT FOR DETAILS.

### C. Robotic Arm

The objective for this example is to compute a trajectory for a Kuka LBR iiwa 14 robotic arm to place a rigid object
onto a shelf while avoiding an obstacle in its workspace. The state vector is comprised of the 7 joint positions and velocities. We again used a quadratic cost function with $Q = I_{14} \times I_{14}$, $R = 10^{-4} \times I_{14}$, and $Q_N = 1000 \times I_{14}$. The dynamics were integrated with a 3rd-order Runge-Kutta method with $N = 400$ timesteps and a horizon of 4 seconds.

The algorithms were initialized with all control inputs set to 0, and input constraints of $-200 \leq u \leq 200$ Nm per joint, and a final state constraint of $x_N(x) = x_f$. The algorithms were again tested at three different constraint tolerance values. In all cases, we set our intermediate cost convergence tolerance to 10 and our final iteration cost tolerance to $10^{-2}$. Finally, the maximum $\mu$ was set to $10^{-30}$ and for the unscented variants, $\beta$ was set to $10^{-4}$. A screen shot of the initial state and a the trajectory computed by the constrained UDP algorithm in the high precision setting is shown in Figure 4.

Fig. 4. Initial state of the Kuka arm and a valid trajectory computed by the constrained UDP algorithm avoiding the obstacle showing the final state.

Table III contains the results of all trials. As before, all of the algorithms handled the low-precision case, producing similar overall costs. The penalty methods converged faster, with lower constraint violation, and larger $\mu$ values. Despite their success in the low precision case, however, both penalty methods failed in the medium precision case. Once again, in the high precision case, only the constrained UDP method succeeded. This example further demonstrates how penalty and iLQR based methods may succeed in with loose constraint tolerances, but the constrained UDP algorithm can support much more precise constraint satisfaction.

### V. CONCLUSION AND FUTURE WORK

We have presented the constrained UDP algorithm, a DDP variant capable of satisfying nonlinear state and input constraints with high accuracy through the use of an augmented Lagrangian. Several directions for future research remain. Combining multiple constraint-handling approaches may prove beneficial. For example, box constraints on inputs were captured using cost terms in our experiments. It would be straightforward to instead use existing QP techniques in the backward pass to compute input constraints [13], [14], while using augmented Lagrangian terms for state constraints. An empirical comparison including barrier methods in the DDP setting would also be interesting.

As highlighted in prior work on UDP [12], the sigma point scaling parameter, $\beta$, must be chosen ahead of time for each example. Automatic approaches to setting $\beta$ remain an interesting open problem. Similarly, more work is needed to determine optimal—or even very good—schedules for $\phi$ and $\mu$. We believe our results could be significantly improved if more time was spent exploring the space of update schemes. Finally, we have not yet optimized our implementation of constrained UDP to minimize computation time per iteration. In future work, we will evaluate the suitability of this algorithm for real-time model predictive control (MPC).

### ACKNOWLEDGMENTS

This work was supported by an Internal Research and Development grant from Draper, Inc. The authors would like to thank Patrick Varin and the members of the Harvard Agile Robotics Lab for their useful feedback and insights.

### REFERENCES


![Fig. 4](image-url)

**TABLE III**

*KUKA ARM OPTIMIZATION RESULTS. SEE TEXT FOR DETAILS.*

<table>
<thead>
<tr>
<th>Method</th>
<th>Precision</th>
<th>ITERS</th>
<th>Cost</th>
<th>$c_x$</th>
<th>$c_u$</th>
<th>$\mu_x$</th>
<th>$\mu_u$</th>
</tr>
</thead>
<tbody>
<tr>
<td>iLQR-P</td>
<td>Low</td>
<td>50</td>
<td>2161.4</td>
<td>9.6e-3</td>
<td>6.5e-8</td>
<td>1e-6</td>
<td>1e-0</td>
</tr>
<tr>
<td>UDP-P</td>
<td>Low</td>
<td>53</td>
<td>2155.4</td>
<td>8.6e-3</td>
<td>5.1e-8</td>
<td>1e-6</td>
<td>1e-0</td>
</tr>
<tr>
<td>iLQR-A</td>
<td>Medium</td>
<td>89</td>
<td>2171.3</td>
<td>7.1e-3</td>
<td>6.1e-3</td>
<td>1e-6</td>
<td>1e-2</td>
</tr>
<tr>
<td>UDP-A</td>
<td>Medium</td>
<td>88</td>
<td>2174.9</td>
<td>6.3e-3</td>
<td>5.4e-3</td>
<td>1e-6</td>
<td>1e-1</td>
</tr>
<tr>
<td>iLQR-P</td>
<td>High</td>
<td>155</td>
<td>2161.3</td>
<td>9.6e-3</td>
<td>6.2e-3</td>
<td>1e-6</td>
<td>1e-7</td>
</tr>
<tr>
<td>UDP-P</td>
<td>High</td>
<td>146</td>
<td>2226.8</td>
<td>7.0e-3</td>
<td>5.3e-3</td>
<td>1e-6</td>
<td>1e-5</td>
</tr>
<tr>
<td>iLQR-A</td>
<td>High</td>
<td>84</td>
<td>2688.4</td>
<td>5.1e-3</td>
<td>1.3e-3</td>
<td>1e-6</td>
<td>1e-4</td>
</tr>
<tr>
<td>UDP-A</td>
<td>High</td>
<td>82</td>
<td>2674.4</td>
<td>4.2e-3</td>
<td>1.8e-3</td>
<td>1e-6</td>
<td>1e-4</td>
</tr>
</tbody>
</table>


