Solving Optimal Control Problems with Interior Point Differential Dynamic Programming

Pranav Pativada
Supervised by Mingda Xu

October 2023

1 INTRODUCTION

Optimal control theory, commonly referred to as trajectory optimization in robotics, presents a framework for motion planning and synthesising complex behaviors in nonlinear dynamic systems [1]. The objective in optimal control is given a problem, to find an *optimal* or desirable set of states and controls that comply with the system's dynamics and any additional constraints. These trajectories are evaluated for optimality through a designated objective function. As such, optimal control problems can be formulated as optimisation problems in which a subset of constraints are governed by the system's dynamics [1, 2].

Optimal control finds practical application across diverse areas such as autonomous systems [3, 4], aerospace engineering [5] and economic policy formulation [6, 7]. These domains rely on the core principle of finding optimal solutions within the confines of respective system limitations.

The formulation of optimal control problems into optimisation problems has led to efficient and specialised numerical methods being developed to solve them, such as Differential Dynamic Programming (DDP) approach [8]. Details of these methods will be discussed later, with particular focus on the Interior Point Differential Dynamic Programming (IPDDP) method [2].

Our work involves porting the IPDDP implementation from MATLAB into a solver interface in Julia [2]. IPDDP is important as it addresses the limitations of previous DDP methods in mangaging additional constraints. The transition from MATLAB to Julia will enhance its accessability and applicability within the wider robotics and control community.

We structure our report as follows. Section 1 provides an introduction to optimal control problems, their formulations, approaches to solve them, and an example application with an inverted problem. Section 2 presents a variety of numerical methods, ranging from the state-of-the-art to IPDDP. Section 3 discusses our contributions. Section 4 discusses our results and analysis, and future work and extensions are listed in Section 5.

1.1 Optimal Control Problems

We define the variables and formulate the optimal control problem as follows. We specifically focus on optimal control problems with finite horizon, discrete-time systems with trajectory horizon T.

Decision variables. We denote the variables to be optimised as the *decision* variables. These are the state state trajectory $x = (x_0, ..., x_T)$ and control trajectory $u = (u_0, ..., u_{T-1})$, composed of state vectors $x_t \in \mathbb{R}^n$ and control vectors $u_t \in \mathbb{R}^m$ for a given time step t.

Dynamics. Given a known initial state x_{init}, we set $x_0 = x_{\text{init}}$. Each subsequent state is then given by time-varying dynamics functions f_t such that

$$x_{t+1} = f_t(x_t, u_t) \tag{1}$$

for $t \in \{0, ..., T-1\}$ and $f_t : \mathbb{R}^n \times \mathbb{R}^m \to \mathbb{R}^n$ are twice continuously differentiable.

Objective function. The objective function measures the optimality of our trajectories. It can be designed in many ways, for example - one approach would be to penalise deviations from a desried goal state.

Objective functions are additive across time and are usually separated into stage cost functions $\ell_t : \mathbb{R}^n \times \mathbb{R}^m \to \mathbb{R}$ for $t \in \{0, ..., T-1\}$ and a terminal cost function $\ell_T : \mathbb{R}^n \to \mathbb{R}$, both twice continuously differentiable.

Optimisation problem. We frame the optimal control problem as the following optimisation proble, where

the optimal trajectories are recovered from the solution:

$$\begin{aligned} \underset{x,u}{\text{minimise}} \quad & \sum_{t=0}^{T-1} \ell_t(x_t, u_t) + \ell_T(x_T) \\ \text{subject to} \quad & x_{t+1} = f_t(x_t, u_t) \\ & c_t(x_t, u_t) \leq 0 \\ & x_0 = x_{\text{init}}, \end{aligned} \tag{2}$$

where x_{init} is a known initial state.

Constraints. For the above problem, we can introduce additional supplementary constraints, such as actuation limits or state boundaries. These constraints are formulated as functions $c_t : \mathbb{R}^n \times \mathbb{R}^m \to \mathbb{R}^l$, which are twice continuously differentiable.

1.2 Solving Optimal Control Problems

The optimization problem defined in Eq. 2 inherently encompasses first-order optimality conditions, known as the Karush-Kuhn-Tucker (KKT) conditions [2]. The numerical methods that are a focal point of this project are designed to solve for trajectories that satisfy these KKT conditions [9]. This is done by iteratively refining a solution from an initial trajectory. Each iteration incrementally adjusts the trajectory, and so each iterate gets closer to satisfying the KKT conditions. In robotics terminology, these methods are categorized into two distinct types: direct and indirect.

Direct methods involve decision variables consisting of both the states x_t and controls u_t. These methods pass the problem formulated in Eq. 2 to general-purpose nonlinear programming (NLP) solvers like IPOPT [10] and SNOPT [11], which are well known for their versatility and robustness [1]. Most significantly, these methods explicitly incorporate and pass in the dynamics constraints as defined in Eq. 1.

Indirect methods leverage the dynamical systems structure of Eq. 2 for enhanced efficiency in comparison to direct methods [1]. Here, only the controls are considered as the decision variables to be optimised. This is because the dynamics constraints are used to determine the state trajectory when simulating the system dynamics. It is important to note that in these indirect methods, the dynamics constraints remain feasible throughout the process. This includes intermediate iterates that have not yet converged to an optimal solution satisfying the KKT conditions. Specialised indirect methods, such as Differential Dynamic Programming (DDP) and iterative Linear Quadratic Regulator (iLQR) – which is a DDP method that does not take into account second order dynamics – achieve rapid performance. These can be attributed to their unique theoretical derivations, alongside algorithmic considerations which improve their complexity to find optimal solutions faster.

We note that both direct and indirect methods have their advantages and disadvantages, which we explore in further detail in Section 2.

1.3 Difficult Problems in Optimal Control

We outline this section to discuss a difficult problem in optimal control - the inverted pendulum problem.

The inverted pendulum problem is a significant challenge in optimal control due to its inherently unstable system with highly non-linear dynamics [12, 13, 14]. The difficulty of this problem lies in it's equilibrium point - the upright position of the pendulum — which represents a non-linearly unstable equilibrium. In this state, any deviations to the pendulum can cause it to break equilibrium [12, 13]. Given that there is a constant need to counteract forces to maintain equilibrium, this makes the problem very non-trivial to solve. This is unlike stable systems, where deviations might correct themselves over time.

The non-linearity of the dynamics introduces complexity in designing control strategies that can stabilize the pendulum [12, 14]. This becomes more complex when additional constraints other than the dynamics are added, such as the mentioned actuator limits or state boundaries [12].

As such, problems like inverted pendulum who inherently have unstable and non-linear dynamics demand sophisticated numerical methods that are capable of handling non-linearities and additional constraints - while managing the complexity of the system and being reasonably fast for real-time use [14]. This gives rise to the combination of efficient and robust algorithms that can accurately model and control these difficult systems, such as IPDDP.

We discuss these methods, alongside their advantages, disadvantages, use-cases, and performance considerations in the following section.

2 NUMERICAL METHODS

2.1 IPOPT

The Interior Point OPTimizer (IPOPT) is a large scale non-linear programming solver that finds solutions of smooth, non-convex, constrained optimisation problems [10]. IPOPT uses a primal-dual interior point approach. It initialises the decision and dual variables and refines them till optimality is reached [10]. IPOPT solves a sequence of optimisation problems, in which each is characterised by a decaying perturbation parameter, $\mu > 0$ [9, 10]. The sequence of solutions is known as the central path, with the solution in the limit at $\mu = 0$ satisfying the KKT conditions for Eq. 2 [9]. IPOPT, alongside other trajectory optimisation methods, compute a search

direction taken from linearising the KKT conditions and solving a linear system called the KKT system [9, 10]. This search direction is then used to update the primal and dual variables [9].

To ensure robustness of convergence, IPOPT employs a filter line search strategy [10]. This is used to ensure that the successive iterates improve with respect to the objective function and constraints. The line search accepts steps with two criteria: sufficient decrease of the objective function and satisfaction of the constraints [9]. This enables the line search and filter to ensure that the iterates are moving towards optimality.

IPOPT exhibits both local quadratic convergence and global convergence properties [2,10]. Local quadratic convergence ensures that we converge quadratically to the solution given sufficient closeness to an optimal point (in which the KKT conditions are satisfied) [10]. Global convergences ensures that regardless of the initialisation of the decision and dual variables, convergence to a locally optimal point is guaranteed [10]. Arbitrary additional constraints are also easily handled by direct methods like IPOPT, as they can be added in and the solver will handle them to the best of its ability.

However, IPOPT does not take into account the dynamical systems nature of problems. In fields such as control and robotics, this is problematic as the optimisation problems are heavily intertwined with the dynamic processes. In particular, direct methods such as IPOPT are only dynamically feasible in the limit [10]. Most importantly, this means that extraction of an intermediate iterate solution can be potentially infeasible with respect to the dynamics. This motivates indirect methods that handle the dynamics of the system, such as DDP, which we will now describe.

2.2 Differential Dynamic Programming

Differential Dynamic Programming (DDP) methods are an indirect approach which take into account the dynamical system's structure [8]. They are thus well suited for optimisation problems in fields such as control and robotics. These methods utilise Bellman's Principle of Optimality, which defines that if an optimal sequence of controls

$$u(i), u(i+1), ...u(j), ..., u(k)$$

gives the solution

$$x(i), x(i+1), ...x(j), ..., x(k)$$

for $i > 0$, then the subpath from any state in that solution to the end must itself be an optimal path [8, 15]. Thus, we have that the truncated controls $u(j), ..., u(k)$ provides the optimal solution for the subpath from $x(j)$ onwards [15]. Invoking this principle allows us to define the cost-to-go function recursively

$$J_k^\star(x) = \min_{\substack{u \text{ s.t.} \\ c(x,u)\leq 0}} \left[\ell_t(x,u) + J_{k-1}^\star(f(x,u))\right], \qquad (3)$$

where $J_0^\star \coloneqq \ell_T(x)$ [2]. This allows the optimisation problem (Eq. 2) to be broken into a series of smaller subproblems, each of which contain an optimal substructure and are used to build the optimal solution.

DDP is usually comprised of two parts - the *backward* pass and the *forward* pass [8]. In the *backward* pass, we obtain a quadratic approximation for the cost-to-go function, obtained from linearising the dynamics and objective cost around the *nominal trajectory* [8]. This is then recursively updated backwards in time for the whole time horizon. The resulting approximation allows us to express an update direction to the nominal trajectory which minimizes the cost-to-go function [8]. Interestingly, the updates are provided in the form of a local feedback policy, otherwise known as *gains*, which are explicitly solved for in backward pass. In the *forward* pass, the nominal trajectory is updated by evaluating the local feedback policy [8]. This process is repeated until convergence or optimality is reached, at which point the solution is extracted. We note that a line search and filter is commonly applied to ensure that the updated nominal trajectory is better than the previous iterate [2]. This ensures that the sequence of iterates improve towards the optimality conditions defined by the KKT system.

An important feature of DDP is around the explicit evaluation of system dynamics is done in DDP. This ensures that the trajectories generated during the optimisation process are dynamically feasible. This is a key strength of indirect methods, as in contrast, direct methods suchas IPOPT are only feasible with regard to the dynamics in the limit [2, 10]. In scenarios where we only have bounded compute, this is a key advantage as we can take the sub-optimal solution so far knowing it is dynamically feasible.

From an algorithmic standpoint, DDP is also inherently much faster than other trajectory optimisation methods such as direct collocation (IPOPT) due to their efficient optimisation process. The explicit evaluation of the dynamics ensures they solve reduced KKT systems relative to direct methods. These systems scale *linearly* in size with complexity $\mathcal{O}(T)$ for indirect approaches, which is ensured by the forward and backward recursion in time [9]. However, it is important to note that this complexity does not consider the outer loop of the algorithm.

Direct methods without explicit dynamic constraints can solve the KKT system in $\mathcal{O}(T^3(n+m)^3)$, with state dimension n and control dimension m, without exploiting the sparsity or structure of the system [16]. Appropriate

variable re-ordering and solving a block-diagonal system can further reduce the complexity to $\mathcal{O}(T(n+m)^3)$ [16].

Dynamic constraints being explicitly included increases the size of the KKT system. For direct methods, this adds a complexity of $\mathcal{O}(Ta)$, due to the a additional number of dual and primal variables needed [17]. This is the case if the sparsity or problem structure of the KKT system can be exploited. In certain scenarios, such as if the KKT system is dense, there is no guarantee of sparsity or such a structure, in which an added worst-case complexity of $\mathcal{O}(T^3a^3)$ is needed to solve the system [16, 17]. When indirect and DDP methods solve these KKT systems, the sparsity is inherently guaranteed [9], leading to an $\mathcal{O}(T)$ complexity for solving [9].

Therefore, the optimal substructure property and significantly reduced complexity for KKT systems of indirect methods are implications from exploiting the dynamical nature of the system, and allow indirect methods to reach greater speeds. For this reason, DDP methods are the preferred choice in the field of control and robotics, where speed and feasibility of the solution are of high importance.

2.3 DDP With Constraints

A significant challenge within Differential Dynamic Programming (DDP) is its ability to address problems involving inequality constraints [2]. In contrast to direct methods, such as IPOPT, where arbitrary constraints can be seamlessly integrated, DDP cannot do so as easily. Though attempts have been made to generalise DDP methods to integrate constraints, such as incorporating box constraints on controls [18] and Augmented Lagrangian (AL) methods [19, 20], they are limited. These aforementioned methods, alongside other methods in literature have issues with numerical instability or can only handle restricted types of constraints. In the following sections we discuss the current state-of-the-art and a novel method, where both aim to seamlessly integrate constraints into the DDP framework.

2.4 ALTRO

Augmented Lagrangian TRajectory Optimizer (ALTRO) is the current state-of-the-art solver for constrained optimisation problems [1]. ALTRO integrates constraints into DDP by incorporating an Augmented Lagrangian objective with iLQR (AL-iLQR), as noted in Section 1.1 to address non-dynamic constraint violations by penalising them. The method extends its capability further by applying active-set projection, which projects solutions onto the manifold defined by active constraints [1]. This allows for rapid convergence. As such, ALTRO has a myriad of benefits taken from both direct and DDP methods. These include speed, handling of generalised state and input constraints, and initialisations that are infeasible with regard to additional constraints [1].

However, despite these strengths, penalty formulation methods like ALTRO have been previously observed to encounter ill-conditioning issues [9]. Additionally, ALTRO lacks both local and global convergence guarantees unlike IPOPT [1, 9]. IPOPT was also found to have tighter constraint violations by multiple orders of magnitude in comparison to a similar AL-iLQR method to ALTRO [17]. This has stemmed the exploration of more numerically robust interior point methods with convergence guarantees, such as local quadratic convergence, and tight constraint satisfaction [9, 17]. Furthermore, the notable speed of DDP has inspired its integration with interior point methods, which are anticipated to significantly exceed the performance of prior AL-based formulations.

2.5 IPDDP

The Interior Point Differential Dynamic Programming (IPDDP) method [2] presents an innovative solution for integrating constraints within the DDP framework. The method harnesses DDP's fast computation capabilities and speed, and augments it with the robust numerical stability and constraint handling of primal-dual interior point methods [2]. The IPDDP method is notable for circumventing the need to alter the objective function. It does not rely on expensive active-set routines and can also handle initialisations that are infeasible with regard to the inequality constraints [2]. A critical advantage of IPDDP is its demonstrated local quadratic convergence in scenarios involving nonlinear constraints. As ALTRO can be summarised as a combination of AL and iLQR, IPDDP can be conceptually represented as a synthesis of IPOPT and DDP. It takes IPOPT's comprehensive constraint handling and numerical stability with DDP's speed, positioning it as an advanced tool in optimal control problem-solving.

3 CONTRIBUTION

The current implementation of IPDDP is a codebase in MATLAB, which is inherently slow due to it being an interpreted language. We feel that the current codebase lacks intuitiveness and consideration for user experience, alongside being incompatible with a number of simulators such as Dojo[1]. This prohibits evaluation on difficult robotics planning problems, which hinders its applicability despite its desirable properties. This makes it difficult to test its effectiveness.

[1] https://github.com/dojo-sim/Dojo.jl

Our main contribution involves porting the MATLAB implementation over into Julia. We provide a faster, cleaner implementation that is integrated with `IterativeLQR.jl` - a software package that performs trajectory optimisation using ALTRO[2]. This integration provides compatibility with robotic simulators, and allows us to properly evaluate the method against the existing numerical methods for optimal control problems. We exploit the inherent high performance language features of Julia such as automatic differentiation with `Symbolics.jl`, just-in-time (JIT) compilation, and static typing. This results in significant speed-ups in comparison to the MATLAB code. The integration into `IterativeLQR` also allows for a user-friendly and intuitive codebase that acts as a much more extendable API in the future. This allows the wider community to use, build, and improve the method. We note that we implement the code with a very different structure to the original MATLAB code, but reconcile it with the same results.

3.1 The Interface

The interface is clean and simple, and encapsulates the different parts of IPDDP. We first define the dynamics, costs, and constraints as functions that characterise the problem.

```
function dynamics(x, u)
    # Defines the dynamics
end
functions cost(x, u)
    # Define the cost
end
functions constraints(x, u)
    # Define the constraints
end
```

We note that one difference with our implementation in comparison to the original IPDDP algorithm is that each of the dynamics, constraints, and costs are time varying. As such, we use the above functions to initialise the dynamics, costs, and constraint objects. These are subsequently passed into the solver data structure which then extracts runs the IPDDP algorithm and extracts the solution. The solver also has access to a list of hyper-parameters which are defined in an options data structure. The code sample below shows the setup for the solver.

```
# Define the solver
solver = Solver(dynamics, cost,
            constraints)
# Solve the problem
solve!(solver)
# Extract the solution
x_sol, u_sol = get_trajectory(solver)
```

In comparison to the MATLAB code, this interface is more readable and intuitive for the user to follow along with. The encapsulation of dynamics, cost, and constraints makes each component very extendable in the future, while having little impact on the solver and the rest of the codebase.

We use this interface and implement `car`, `concar`, `invpend`, and `arm` from the IPDDP paper [2,18], where each is defined with their own set of dynamics, constraints, and costs similarly to above. At this stage, we have reconciled our results with the `car` example, and plan to reconcile the rest shortly later.

3.2 The Backward Pass

The backward pass implementation follows closely to the MATLAB code. We note that in our implementation, we do in-place matrix multiplication. This is done by pre-allocating matrices in the data structure within the solver, and utilising the overloaded `mul!` function. This results in faster compute times as the data structures do not need to be dynamically allocated. This is a key difference from the MATLAB implementation that we have changed to gain a speed-up.

We also differ in the computation of the gains. In the current MATLAB implementation, a Cholesky decomposition is used to compute the gains with a set regularisation. In our implementation, we do the same, but force the input matrix to the Cholesky decomposition to be symmetric prior to performing it. This is done by setting

$$X' = \frac{1}{2}(X + X^T),$$

where X is the input matrix, and performing the decomposition on X'. This is due to minor floating-point discrepancies we noticed in the matrix values, causing it to diverge from symmetry. This deviation from its symmetric nature leads to decomposition failure, resulting in excessively high regularisation that frequently prevents the solver from extracting a solution. As such, we do this to maintain the consistency of the algorithm and to prevent errors later on.

3.3 The Forward Pass

The forward pass implementation in Julia mirrors that of MATLAB. Our approach involves maintaining a similar execution process, with added optimizations. Notably, we store nominal trajectories and variables from preceding iterations. This allows improvement to be tracked over time via comparison to these nominal trajectories.

[2] https://github.com/thowell/IterativeLQR.jl

One particular optimisation is the immediate termination of the line search if the condition checks fail at any step. This applies to dual variables and constraints in feasible IPDDP, and the dual variables and slack variables in infeasible IPDDP. Such early termination prevents rollout from occuring as they do not improve the objective, saving wasted computation. This pre-emptive failure stops unnecessary computation from happening.

We note that during the evaluation of constraints, dual variables, and slack variables in the line search phase, Julia's approach showed certain unintended interactions with their nominal counterparts. These interactions stemmed from pointer manipulation errors. We mitigated these by employing deep copying techniques, but we note that this aspect of the Julia implementation offers a potential area for enhancement, particularly in optimizing memory usage and runtime efficiency.

4 EXPERIMENTAL RESULTS AND ANALYSIS

We conduct our experiments on an Apple MacBook Pro with an M2 Pro processor with 10 CPU cores.

4.1 Execution Time

We assess the execution times by running the `car` example in MATLAB and Julia, each tested fifty times under both feasible and infeasible starting conditions, utilizing a horizon of 51 steps.

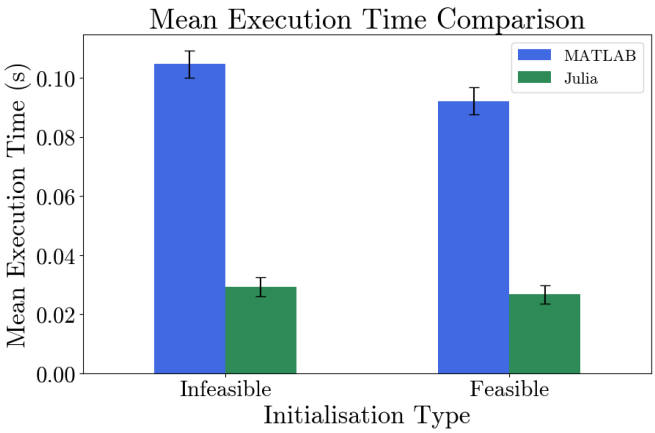

Figure 1: Mean execution time comparison for feasible and infeasible initialisations of IPDDP for the `car` example. Measurements were evaluated on the MATLAB and Julia implementation using a horizon of 51.

Figure 1 shows that for feasible initialisations, we observed mean execution times of 0.092 ± 0.005 and 0.027 ± 0.003 for MATLAB and Julia respectively. Infeasible initialisations resulted in mean execution times of 0.105 ± 0.005 for MATLAB and 0.029 ± 0.003 seconds for Julia. As such, we see a consistent performance increase with Julia over MATLAB, with a close to fourfold improvement in performance on average. We note that Julia's first initial run was slower in regard to execution time due to it's compilation process. However, subsequent runs still outperformed the MATLAB code over time with significant speed-ups.

4.2 Numerical Differences

We reconcile the numerical differences and outputs between the MATLAB and Julia code in this section. We note that the `car` example was used in testing with a horizon of 51.

a)

Iteration	Elapsed time	mu	Cost	Opt.error	Reg.power	Stepsize
41	4.23496e-02	4.06013e-07	4.70078e+00	6.95710e-07	0.000e+00	1.00000e+00
42	4.29591e-02	4.06013e-07	4.70078e+00	1.55099e-07	0.000e+00	1.00000e+00
43	4.41639e-02	4.06013e-07	4.70078e+00	1.18691e-07	0.000e+00	1.00000e+00
44	4.50094e-02	4.06013e-07	4.70078e+00	3.33525e-08	0.000e+00	1.00000e+00
45	4.57358e-02	2.13918e-08	4.70075e+00	1.40637e-02	0.000e+00	1.00000e+00
46	4.62955e-02	2.13918e-08	4.70075e+00	1.51200e-04	0.000e+00	1.00000e+00
47	4.71565e-02	2.13918e-08	4.70075e+00	2.41018e-05	0.000e+00	1.00000e+00
48	4.77907e-02	2.13918e-08	4.70075e+00	1.69531e-06	0.000e+00	1.00000e+00
49	4.91341e-02	2.13918e-08	4.70075e+00	6.93267e-07	0.000e+00	1.00000e+00
50	4.98433e-02	2.13918e-08	4.70075e+00	7.01301e-08	0.000e+00	1.00000e+00

Optimality reached!

b)

Iteration	Time	mu	Cost	Opt. error	Reg. power	Stepsize
41	0.06587	4.06e-07	4.7008	6.957e-07	0	1.000
42	0.06724	4.06e-07	4.7008	1.551e-07	0	1.000
43	0.06848	4.06e-07	4.7008	1.187e-07	0	1.000
44	0.06965	4.06e-07	4.7008	3.335e-08	0	1.000
45	0.07084	2.139e-08	4.7008	0.01406	0	1.000
46	0.07221	2.139e-08	4.7008	0.0001512	0	1.000
47	0.07367	2.139e-08	4.7008	2.41e-05	0	1.000
48	0.07511	2.139e-08	4.7008	1.695e-06	0	1.000
49	0.07634	2.139e-08	4.7008	6.933e-07	0	1.000
50	0.07768	2.139e-08	4.7008	7.013e-08	0	1.000

~~Optimality reached~~

Figure 2: Feasible IPDDP output of the `car` problem with $T = 51$. a) shows the Julia code output. b) shows the MATLAB code output.

a)

Iteration	Elapsed time	mu	Cost	Opt.error	Reg.power	Stepsize
51	4.11726e-02	2.13918e-08	4.70075e+00	2.41018e-05	0.000e+00	1.00000e+00
52	4.20489e-02	2.13918e-08	4.70075e+00	1.69531e-06	0.000e+00	1.00000e+00
53	4.29337e-02	2.13918e-08	4.70075e+00	6.93268e-07	0.000e+00	1.00000e+00
54	4.35032e-02	2.13918e-08	4.70075e+00	7.01301e-08	0.000e+00	1.00000e+00

Optimality reached!

b)

Iteration	Time	mu	Cost	Opt. error	Reg. power	Stepsize
51	1.017	2.139e-08	4.7008	2.41e-05	0	1.000
52	1.021	2.139e-08	4.7008	1.695e-06	0	1.000
53	1.024	2.139e-08	4.7008	6.933e-07	0	1.000
54	1.029	2.139e-08	4.7008	7.013e-08	0	1.000

~~Optimality reached~~

Figure 3: Infeasible IPDDP output of the `car` problem with $T = 51$. a) shows the Julia code output. b) shows the MATLAB code output.

For a feasible initialisation, we find that the Julia output (Figure 2 a) is identical to that of the MATLAB output (Figure 2 b). Optimal solutions were both found in 50 iterations, and the other variables such as the perturbation, objective cost, optimality error were identical to machine precision.

The infeasible initialisation with `car` was also similar. Optimal solutions were found in 54 iterations for both Julia output (Figure 3 a) and MATLAB output (Figure 3 b).

Perturbation, objective cost, and optimality error variables were also identical to machine precision.

However, we observed numerical differences in intermediate computations between our implementation and the MATLAB code. In particular, values in Julia often had at least one extra digit of precision compared to the MATLAB code. This caused issues when performing the Cholesky decomposition, which we resolved as mentioned in Section 3.2. Highlighted in red is the difference in numerical precision. The values below in Table 1 are the matrix entries of a 3×3 input matrix to the Cholesky decomposition.

Table 1: Matrix entry comparison between Julia and MATLAB implementations.

Matrix Entry	Julia Values	MATLAB Values
1,1	27.55692438875954	27.5569243887595
2,1	4.273521188308174	4.27352118830817
3,1	1.5739007016011186	1.57390070160112
1,2	4.253521188308174	4.25352118830817
2,2	23.02296281269114	23.0229628126911
3,2	1.1613986839920225	1.16139868399202
3,1	1.5739007016011182	1.57390070160112
3,2	1.1613986839920225	1.16139868399202
3,3	20.782968239015794	20.7829682390158

5 EXTENSIONS

We detail additional extensions from IPOPT [10] to implement into IPDDP, with a focus on enhancing robustness and computational performance. These extensions include:

- Implementing symmetric indefinite solvers for inertia correction, replacing current block-inverse computations. This would give us additional eigenvalue information, which would tell us whether or not to add regularisation.
- Introducing advanced filters during the forward pass line search to optimise convergence.
- Incorporating second-order corrections using Newton-type steps to improve solution feasibility.
- Establishing a feasibility restoration phase.

6 CONCLUSION

Optimal control theory finds extensive application in various domains, especially in dynamic robotic systems. The Interior Point Differential Dynamic Programming (IPDDP) approach combines the efficiency of Differential Dynamic Programming with interior point methods' robustness, convergence assurance, and rigorous constraint satisfaction characteristics.The successful migration of the IPDDP implementation from MATLAB to a more efficient, user-friendly, and simulator-compatible interface in Julia was our principal contribution. This transition enhanced performance (up to fourfold for the `car` example), and added features such as symbolic auto-differentiation, in-place matrix multiplication, and pre-emptive failure during line search. In the future, we aim to extend the capabilities of this implementation by incorporating additional functionalities from IPOPT, which would further increase its performance and robustness. We note that our effort underscores our commitment to the field of optimal control and it's use in the applications of robotics and beyond.

ACKNOWLEDGEMENTS

I would like to extend my gratitude to Mingda Xu for his invaluable mentorship and insights throughout the development of this work. His expertise and guidance have been instrumental in the successful completion of this project.

References

[1] Taylor A Howell, Brian E Jackson, and Zachary Manchester. Altro: A fast solver for constrained trajectory optimization. In *2019 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 7674–7679. IEEE, 2019.

[2] Andrei Pavlov, Iman Shames, and Chris Manzie. Interior point differential dynamic programming. *IEEE Transactions on Control Systems Technology*, 29(6):2720–2727, 2021.

[3] Patrick M. Wensing, Michael Posa, Yue Hu, Adrien Escande, Nicolas Mansard, and Andrea Del Prete. Optimization-based control for dynamic legged robots, 2022.

[4] Joseph Moore and Russ Tedrake. Control synthesis and verification for a perching uav using lqr-trees. In *2012 IEEE 51st IEEE Conference on Decision and Control (CDC)*, pages 3707–3714, 2012.

[5] Behçet Açıkmeşe, John M. Carson, and Lars Blackmore. Lossless convexification of nonconvex control bound and pointing constraints of the soft landing optimal control problem. *IEEE Transactions on Control Systems Technology*, 21(6):2104–2113, 2013.

[6] Robert Dorfman. An economic interpretation of optimal control theory. *The American Economic Review*, 59(5):817–831, 1969.

[7] Daniel Leonard and Ngo Van Long. *Optimal control theory and static optimization in economics*. Cambridge University Press, 1992.

[8] Ignat Georgiev. Deriving differential dynamic programming, February 2023.

[9] Jorge Nocedal and Stephen J. Wright. *Numerical Optimization*. Springer, New York, NY, USA, 2e edition, 2006.

[10] Andreas Wächter and Lorenz T. Biegler. On the implementation of an interior-point filter line-search algorithm for large-scale nonlinear programming. *Mathematical Programming*, 106(1):25–57, April 2005.

[11] Philip E Gill, Walter Murray, and Michael A Saunders. Snopt: An sqp algorithm for large-scale constrained optimization. *SIAM review*, 47(1):99–131, 2005.

[12] Lal Bahadur Prasad, Barjeev Tyagi, and Hari Om Gupta. Optimal control of nonlinear inverted pendulum system using pid controller and lqr: Performance analysis without and with disturbance input. *International Journal of Automation and Computing*, 11(6):661–670, December 2014.

[13] Lal Bahadur Prasad, Hari Om Gupta, and Barjeev Tyagi. Adaptive optimal control of nonlinear inverted pendulum system using policy iteration technique. *IFAC Proceedings Volumes*, 47(1):1138–1145, 2014.

[14] Elisa Sara Varghese, Anju K Vincent, and V Bagyaveereswaran. Optimal control of inverted pendulum system using pid controller, lqr and mpc. *IOP Conference Series: Materials Science and Engineering*, 263:052007, November 2017.

[15] Karl Henrik Johansson. Dynamic programming.

[16] Yang Wang and Stephen Boyd. Fast model predictive control using online optimization. *IEEE Transactions on Control Systems Technology*, 18(2):267–278, March 2010.

[17] Taylor A. Howell, Simon Le Cleac'h, Sumeet Singh, Pete Florence, Zachary Manchester, and Vikas Sindhwani. Trajectory optimization with optimization-based dynamics, 2023.

[18] Yuval Tassa, Nicolas Mansard, and Emo Todorov. Control-limited differential dynamic programming. In *2014 IEEE International Conference on Robotics and Automation (ICRA)*, pages 1168–1175. IEEE, 2014.

[19] Brian Plancher, Zachary Manchester, and Scott Kuindersma. Constrained unscented dynamic programming. In *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 5674–5680. IEEE, 2017.

[20] Gregory Lantoine and Ryan P Russell. A hybrid differential dynamic programming algorithm for constrained optimal control problems. part 1: Theory. *Journal of Optimization Theory and Applications*, 154:382–417, 2012.