

**NLOptControl: A modeling language for solving optimal control problems**

Huckleberry Febbo, Paramsothy Jayakumar, Jeffrey L. Stein, and Tulga Ersal*

---

**Abstract**—Current direct-collocation-based optimal control software is either easy to use or fast, but not both. This is a major limitation for users that are trying to formulate complex optimal control problems (OCPs) for use in on-line applications. This paper introduces NLOptControl, an open-source modeling language that allows users to both easily formulate and quickly solve nonlinear OCPs using direct-collocation methods. To achieve these attributes, NLOptControl (1) is written in an efficient, dynamically-typed computing language called Julia, (2) extends an optimization modeling language called JuMP to provide a natural algebraic syntax for modeling nonlinear OCPs; and (3) uses reverse automatic differentiation with the acyclic-coloring method to exploit sparsity in the Hessian matrix. This work explores the novel design features of NLOptControl and compares its syntax and speed to those of PROPT. The comparison shows that NLOptControl models OCPs more concisely than PROPT. The speeds of various collocation methods within PROPT and NLOptControl are benchmarked over a range of collocation points using performance profiles; overall, NLOptControl's single, two, and four interval pseudospectral methods are roughly 14, 26, and 36 times faster than PROPT's, respectively. NLOptControl is well-suited to improve existing off-line and on-line control systems and to engender new ones.

---

**I. INTRODUCTION**

Optimal control software packages that implement direct-collocation methods are used in a number of off-line [1], [2], [3], [4], [5], [6] and on-line [7], [8] applications as summarized in Table I. The primary function of these packages is to directly transcribe a human modeler’s formulation of an optimal control problem (OCP) into a nonlinear programming problem (NLP). A key challenge with this process is enabling human modelers (i.e., users) to easily formulate new and complex problems while producing an NLP that can be quickly solved by an external NLP solver. However, current direct-collocation-based optimal control software packages are generally either fast or easy to use, but not both. Thus, these packages are not well suited for non-expert users trying to formulate complex problems for on-line applications, wherein speed is critical. Therefore, there is a need for a direct-collocation-based optimal control software package that is both fast and easy to use. In this paper, an approach to bridging this gap is presented and incorporated into a new, open-source optimal control modeling language called NLOptControl [9].

As seen in Table I, some of the most well-known optimal control software packages (GPOPS-II, PROPT, DIDO) are closed-source and often require a licensing-fee. These drawbacks limit their research value, since they are not freely available to the entire research community, results may be difficult to reproduce, and if the details of the underlying algorithms cannot both be seen and modified, then open validation and development of the these algorithms is not possible [10], [11]. Fortunately, several noteworthy open-source optimal control software packages exist. For completeness, this paper does not limit its discussions to these open-source packages.

Optimal control packages with an algebraic syntax that closely resembles the Bolza form of OCPs [12] are categorized as easy to use. It is noted that there are other design features that affect ease of use; for instance, not having a built-in initialization algorithm [13] reduces ease of use, but these aspects of ease of use are not addressed in this paper. Table I shows that this work categorizes the direct-collocation-based optimal control software packages GPOPS-II [1], PROPT [3] and GPOCS [14] as easy to use and CasADi [15] and DIDO [16] as not easy to use.

For ease of use, modeling languages should have a syntax that closely resembles the class of problems for which they have been designed. Modeling languages like AMPL and GAMs are not embedded in a pre-existing computational language, which allows for syntactical flexibility, when developing them. However, this approach (1) makes development of the modeling language difficult and time-consuming, and (2) does not directly expose users to the breath of features avail-

---

**TABLE I: Landscape of direct-collocation-based optimal control software focusing on their applications and properties. † indicates that the software is too slow for use on the on-line application.**

| Software   | Off-line | Applications | On-line | Properties |
|------------|----------|--------------|---------|------------|
| GPOPS-II   | [1]      | Chemical     | [17]    | [†]        |
| PROPT      | [3]      | [3]          | [18]    | [†]        |
| GPOCS      | [4]      | [5]          | [19]    |            |
| DIDO       | [6]      | [6]          | [7]     |            |
| ACADO      |          |              |         |            |
| CasADi     | [10]     |              |         |            |
| Custom     | [11]     |              |         |            |
| NLOptControl | [9]     | [11, 20]     | [12]    |            |

---

*Corresponding author (tersal@umich.edu)
able in a computational language such as C++ or MATLAB. For these reasons, modeling languages are often embedded in a pre-existing computational language.

It can be difficult to establish a syntax for the modeling within the syntactical confines of a pre-existing computational language. To overcome this issue, operator overloading can be used. For instance, a multiple-shooting method based optimal control software package called ACADO [21] uses operator overloading to allow its user to define an OCP using symbolic expressions that closely resemble the actual mathematical expressions of the problem. However, a naive implementation of operator overloading can lead to performance issues [22]. Additionally, Moritz Diehl, a researcher who developed ACADO and MUSCOD-II, later acknowledges that, ACADO Toolkit [21], DIRCOL [23], DyOS [24], and MUSCOD-II [25] restrict the problem formulations, particularly for users not involved with the development of these tools [15]. The above acknowledgment is included in a paper [15] that introduces CasADi. CasADi allows users to formulate OCPs with fewer restrictions that ACADO. However, CasADi requires that users write the code for the transcription methods. Transcription methods are a general class of numerical methods used to approximate continuous-time OCPs; a direct-collocation method is a type of transcription method. CasADi lets users to code their own transcription methods to avoid creating a "black box" OCP solver that is only capable of solving restrictive formulations, as with ACADO. While this approach may be pedagogically valuable for users, it can lead to bugs and long development time [26] and it makes CasADi’s syntax not closely resemble OCPs. For these reasons, this paper does not categorize CasADi as easy to use. On similar grounds, DIDO is not categorized as easy to use.

For safety in on-line applications, the trajectory needs to be provided to the plant in real-time. An on-line optimal control example is a nonlinear model predictive control (NMPC) problem. Real-time is achieved when the NLP solve-times are all less than the chosen execution horizon. Otherwise, the low-level controllers will not have a trajectory to follow. Despite the need for small solve-times (i.e., speed), several implementations of direct-collocation methods within the MATLAB computational language are not able to achieve solve-times that are less than the execution horizon for a number of NMPC applications. As seen in Table I, GPOCS, GPOPS-II, and custom MATLAB software are not fast enough for NMPC applications in aircraft [18], robot [17], and UGV [19], [20] systems, respectively. On the other hand, CasADi, which is written in C++, is fast enough for an NMPC application in a robot system [7]. Given this practical limitation, this paper will now discuss why some direct-collocation-based optimal control packages are fast while others are slow.

As seen in Table I, this work categorizes GPOPS-II, PROPT, GPOCS, and DIDO as slow and CasADi as fast. If a package uses sparse automatic differentiation methods implemented in a computation language that approaches the speeds of C, it is categorized as fast; the reasoning for this categorization is explained below.

The main algorithmic step in direct method based numerical optimal control is solving the NLP. The solve-time for this step consists of two major parts: (1) the time spent running optimization algorithms within the NLP solver, and (2) the time spent evaluating the nonlinear functions and their corresponding derivatives. Fortunately, low-level algorithms, which are available within several prominent NLP solvers, such as KNITRO [27], IPOPT [28], and SNOPT [29], can be used to reduce the time associated with running the optimization algorithms. The second component is discussed here in terms of current direct-collocation-based optimal control software packages.

The speed of direct-method-based optimal control software depends on the speed of the differentiation method within the computational language in which it is implemented. GPOPS-II uses a sparse finite difference method [30] to calculate the derivatives using the MATLAB computational language. However, finite difference methods, like the sparse finite difference method, are not only slow, but they are also inaccurate [31]. In addition to this, the dynamically-typed MATLAB computational language is typically slow in comparison to statically-compiled languages such as C and Fortran. Since GPOPS-II uses a slow differentiation method within a relatively slow computational language, it is categorized as slow. PROPT uses either symbolic- or forward-automatic differentiation to calculate the derivatives using MATLAB. While PROPT’s methods are more accurate and generally faster than finite difference methods, they do not exploit the sparse structure of the Hessian matrices that is born from a direct-collocation method, like the sparse finite difference method in GPOPS-II. Given this computational limitation and the slow speed of MATLAB, this paper considers PROPT to be slow as well. On the other hand, CasADi uses the star-coloring method [32] to exploit the sparse structure of the Hessian matrix and reverse automatic differentiation implemented in C++ [15]. Since CasADi employs a differentiation methods that is well suited for the sparse structure of the Hessian matrix and it is implemented in a fast computational language, CasADi is categorized as fast. On similar grounds, this paper identifies GPOCS and DIDO as slow.

In sum, there is no direct-collocation-based optimal control software package is both fast and easy to use. CasADi is fast, but not easy to use; and GPOPS-II, PROPT, and GPOCS are easy to use, but not fast. Thus, there is a need for a package that is both fast and easy to use.

This paper investigates an approach for improving both speed and ease of use of optimal control software. As described in detail in Section II, this approach uses recent advances in computational languages and differentiation methods in contrast to the computational languages and differentiation methods used by current direct-collocation-based optimal control software. Additionally, also unlike current direct-collocation-based optimal control software packages, this approach extends an optimization modeling language to include syntax for modeling OCPs. More specifically, this approach is as follows:

**Approach**

- For ease of use and speed, NLoptControl is embedded in the fast, dynamically-typed Julia programming lan-
guage [33].

- For increased ease of use, \textsc{NLOptControl} extends the \textit{JuMP} optimization modeling language [34], which is written in \textit{Julia}, to include a natural syntax for modeling OCPs in Bolza form.

- For increased speed, \textsc{NLOptControl} uses the acyclic-coloring method [35] to exploit sparsity in the Hessian matrix and reverse-automatic differentiation through the \textit{ReverseDiff} package [36], which is also written in \textit{Julia}.

Therefore, this work addresses the following research question: Can the above outlined approach improve speed and ease of use of direct-collocation-based optimal control software? This question is answered by comparing \textsc{NLOptControl}'s speed and ease of use to those of \textsc{PROPT}.

\textsc{NLOptControl} was released as a free, open-source software package in the summer of 2017 [9]. Since then, the literature has shown that \textsc{NLOptControl} is fast and easy to use. For speed, \textsc{NLOptControl} was leveraged to solve complex trajectory planning problems for an unmanned ground vehicle system in real-time — solving these types of problems in real-time using \textsc{MATLAB} was not feasible in prior work [19], [20]. For ease of use, \textsc{NLOptControl} was used to create a new optimal control based learning algorithm [37] without any help from the developers of \textsc{PROPT}.

The remainder of this paper is organized as follows. Section II further describes \textsc{NLOptControl}'s approach to bridging the research gap. Section III describes the classes of offline and on-line OCPs that can be solved using \textsc{NLOptControl}. Section IV provides a brief background on numerical optimal control and a mathematical description of the direct-collocation methods implemented within \textsc{NLOptControl}. Section VI provides an example that compares \textsc{NLOptControl}'s ease of use against \textsc{PROPT}'s and benchmarks \textsc{NLOptControl}'s speed against \textsc{PROPT}. Section VII answers the research question and discusses further implications. Finally, Section VIII summarizes the work and draws conclusions.

II. SOFTWARE ECOSYSTEM

Advances in computational languages, optimization modeling languages, and differentiation methods and tools made it possible to create \textsc{NLOptControl}. This section describes these software advances and shows how they can be leveraged to create a modeling language for a class of optimization problems.

A. Computational languages

Direct-collocation based optimal control software packages are embedded in either a statically- or a dynamically typed computational language. Dynamically typed languages enable users to quickly develop and explore new concepts, yet they are typically slow; statically typed languages sacrifice the user’s productivity for speed. Recently, however, a dynamically typed computing language called \textit{Julia} has become a popular alternative to the computing languages that the current optimal control software packages are embedded in. It has become popular, because it allows users to write high-level code that closely resembles their mathematical formulas, while producing low-level machine code that approaches the speed of \textsc{Fortran} [33]. The claim that \textit{Julia} is not only fast, but also easy to use, motivates the investigation presented in this paper. Specifically, this paper investigates the ability of the \textit{Julia} computational language to improve speed and ease of use for optimal control software.

B. Modeling optimization problems

In the late 1970’s, researchers using optimization software were more concerned with the need to improve the software’s ease of use than its speed [38]. Eventually, this concern led to the development a number of optimization modeling languages, such as \textsc{GAMS} [39] and \textsc{AMPL} [40]. The role of an optimization modeling language is to translate optimization problems from a human-friendly language to a solver-friendly language [41], [42]. In other words, optimization modeling languages do not solve optimization problems; they focus on modeling problems at a high-level and passing optimization problems to external low-level solvers, which are the NLP solvers and the differentiation tools in the context of this work. Similarly, in this work, the high-level problem is the NLP, given in Eqn. 1 - Eqn. 3 (i.e., the NLP model) as

\[
\text{minimize} \quad f(z) \quad (1) \\
\text{subject to} \quad g(z) \leq 0 \quad (2) \\
\quad h(z) = 0 \quad (3)
\]

where the objective function \( f : \mathbb{R}^n \rightarrow \mathbb{R} \), with \( n \) defined as the number of design variables; the inequality constraints \( g : \mathbb{R}^n \rightarrow \mathbb{R}^q \); and the equality constraints \( h : \mathbb{R}^n \rightarrow \mathbb{R}^q \), are all assumed to be twice-continuously differentiable functions [28], [43].

A number of standard optimization problem classes do not fit readily into the NLP model. In addition to this, translating these standard problem classes into the NLP model can require significant work. Thus, for users interested in simply modeling these standard problem classes, and not translating these problems into an NLP model, the NLP model should be extended to include higher-level modeling languages for these standard problem classes. However, most optimization modeling languages are not designed to be extended in this fashion [42]. Because of this limitation, both the speed and ease of use of optimal control packages have suffered. \textsc{GPOCS}, \textsc{GPOPS-II}, and \textsc{PROPT} are slow because the sparse-automatic differentiation methods — typically available through an optimization modeling language — are not available in \textsc{MATLAB}; so, these packages use less efficient differentiation methods. Additionally, since these packages are not built upon an existing NLP modeling language, the API tends to be overly flexible, which can lead to modeling errors [21].

\textit{JuMP} [22], a recent optimization modeling language that is embedded in the fast, dynamically-typed \textit{Julia} programming language [33], is designed to be extended to include new classes of optimization problems. \textit{JuMP} extensions include: parallel multistage stochastic programming [44], robust optimization [45], chance constraints [46], and sum of squares...
A. Modeling OCPs

(e.g., NMPC problems).

capabilities and salient design features for on-line applications can model, and demonstrates NLOptControl for either off-line or on-line applications. This control the plant for either an off-line or on-line tasks.


time in NMPC applications, horizon that is added to account for the non-negligible solve-

to provide a natural syntax for modeling OCPs; leverages ReverseDiff to exploit sparsity in the Hessian matrices. Research shows that the acyclic-coloring method is faster than the star-coloring method [48], which was used in the CasADi package [15]. These advances are leveraged to create an optimal control modeling language called NLOptControl.

C. Proposed software ecosystem

Fig. 1 presents NLOptControl’s software ecosystem and its function as an optimal control software package. In terms of this ecosystem, it is: embedded in Julia; extends JuMP to provide a natural syntax for modeling OCPs; leverages ReverseDiff; and interfaces with KNITRO, IPOPT, and potentially other solvers to solve the automatically formulated NLP problem. To use NLOptControl, users need only formulate their OCP into a syntax-based model of the OCP. This model is then approximated using one of the direct-collocation methods implemented in NLOptControl, which at the time of this writing include: the Euler’s backwards, the trapezoidal, and the Radau collocation methods. After the model has been approximated, the software ecosystem solves this approximation to determine an optimal trajectory. This trajectory can then be followed using low-level controllers to control the plant for either an off-line or on-line tasks.

III. Scope of NLOptControl

NLOptControl is designed for modeling OCPs and solving them for either off-line or on-line applications. This section shows the types of problems that NLOptControl can model, and demonstrates NLOptControl’s visualization capabilities and salient design features for on-line applications (e.g., NMPC problems).

A. Modeling OCPs

An important class of optimization problems is the OCP. NLOptControl models single-phase, continuous-time, OCP

in a Bolza form [12] that is tailored for NMPC problems and adds slack constraints on the initial and terminal states as

\[
\begin{align*}
\text{minimize} & \quad M(x(t_0 + t_{ex}), t_0 + t_{ex}, x(t_f), t_f) \\
& + \int_{t_0 + t_{ex}}^{t_f} L(x(t), u(t), t) \, dt \\
& + w_s x_s + w_f x_f \\
\text{subject to} & \quad \frac{dx}{dt}(t) - F(x(t), u(t), t) = 0 \\
& \quad C(x(t), u(t), t) \leq 0 \\
& \quad x_0 - x_{0_{tol}} \leq x(t_0 + t_{ex}) \leq x_0 + x_{0_{tol}} \\
& \quad x_f - x_{f_{tol}} \leq x(t_f) \leq x_f + x_{f_{tol}} \\
& \quad x_{min} \leq x(t) \leq x_{max} \\
& \quad u_{min} \leq u(t) \leq u_{max} \\
& \quad t_{f_{min}} \leq t_f \leq t_{f_{max}} \\
& \quad x_0 - x_{0_{tol}} \leq x(t_0 + t_{ex}) \leq x_0 + x_{0_{tol}} \\
& \quad x_0 + x(t_0 + t_{ex}) \geq x_{0_{tol}} \\
& \quad x_f - x(t_f) \leq x_f \\
& \quad x_f + x(t_f) \geq x_f \\
\end{align*}
\]

where \( t_0 \) is the fixed initial time, \( t_{ex} \) is the fixed execution horizon that is added to account for the non-negligible solve-times in NMPC applications, \( t_f \) is the free final time, \( t \) is the time, \( x(t) \in \mathbb{R}^{n_{st}} \) is the state, with \( n_{st} \) defined as the number of states, and \( u(t) \in \mathbb{R}^{n_{ctr}} \) is the control, with \( n_{ctr} \) as the number of controls. \( x_{s_0} \in \mathbb{R}^{n_{st}} \) and \( x_{s_f} \in \mathbb{R}^{n_{st}} \) are optional slack variables for the initial and terminal states, respectively. The objective functional includes \( M : \mathbb{R}^{n_{st}} \times \mathbb{R} \times \mathbb{R}^{n_{st}} \times \mathbb{R} \rightarrow \mathbb{R} \) and \( L : \mathbb{R}^{n_{st}} \times \mathbb{R}^{n_{ctr}} \times \mathbb{R} \rightarrow \mathbb{R} \), \( C : \mathbb{R}^{n_{st}} \times \mathbb{R}^{n_{ctr}} \times \mathbb{R} \rightarrow \mathbb{R}^p \) denote the dynamic constraints and the path constraints, respectively; \( p \) is the number of path constraints. \( x_0 \in \mathbb{R}^{n_{st}} \) and \( x_f \in \mathbb{R}^{n_{st}} \) denote the desired initial and final states, respectively. \( x_{0_{tol}} \in \mathbb{R}^{n_{st}} \) and \( x_{f_{tol}} \in \mathbb{R}^{n_{st}} \) establish tolerances on the initial and final state, respectively. Constant upper and lower bounds on the state, control, and final time are included with Eqn. 9, Eqn. 10, and Eqn. 11, respectively. Finally, NLOptControl adds Eqn. III-A - Eqn.
III-A to the Bolza form for optional slack constraints on the initial and terminal states.

NLOptControl is embedded in the Julia language and specializes JuMP's syntax to better suit the domain of optimal control. JuMP leverages Julia’s syntactic macros [33] to enable a natural algebraic syntax for modeling optimization problems, without sacrificing performance or restricting problem formulations [22]. NLOptControl extends JuMP to include syntax for modeling OCPs in Bolza form in Eqn. 4 – Eqn. 11, with the option of including slack constraints on the initial and terminal states through Eqn. 8 – Eqn. 9.

For a basic example of this syntax, NLOptControl is now used to model the Bryson-Denham problem, which is given in mathematical form as

\[
\begin{align*}
\text{minimize} & \quad \frac{1}{2} \int_0^1 a(t)^2 \, dt \\
\text{subject to} & \quad \dot{v}(t) = a(t), \quad \dot{x}(t) = v(t), \quad x(t) \leq \frac{1}{12} \\
& \quad v(0) = -v(1), x(0) = 1, x(1) = 0
\end{align*}
\]

The `define()` function is used to create a model object and define Eqn. 7 - Eqn. 10 as

\[
n = \text{define}(\text{numStates} = 2, \text{numControls} = 1, X0 \leftrightarrow [0,1], \text{XF} = [0,-1], \text{XL} = [0,\text{NaN}], \text{XU} = [1/12, \text{NaN}], \text{CL} = [\text{NaN}, \text{NaN}], \text{CU} = [\text{NaN}, \text{NaN}])
\]

where \( n \) is an object that holds the entire optimal control model, `numStates` and `numControls` are the number of states and controls, \( X0 \) and \( XF \) are arrays of the initial and final state constraint, \( XL \) and \( XU \) are arrays of any lower and upper state bounds, \( NaN \) indicates that a particular constraint is not applied, and \( CL \) and \( CU \) are arrays of any lower and upper control bounds.

The dynamic constraints in Eqn. 5 are then added to the model through the `dynamics()` function as

```
dynamics!(n, [:(x2[j]), :(u1[j])])
```

where the `!` character indicates that the model object \( n \) is being modified by the function. The elements of the array \( :\{x2[j] \leftrightarrow j\} \) and \( :\{u1[j]\} \) represent \( v(t) \) and \( a(t) \); by default the state and control variables are \( x1,x2, \ldots \) and \( u1,u2, \ldots \) but they can be changed. Differential equations must be passed within an array of Julia expressions (i.e., \( \{():(),(),\ldots,:() \leftrightarrow j\} \)), and the index \( \{j\} \) must be appended to the state and control variables. \( j \) is used within NLOptControl to index particular time discretization points \( t \in [t_0 + t_{ex}, t_f] \).

The next step is to indicate whether or not the final time \( t_f \) is a design variable using the `configure()` function as

```
configure!(n; (:finalTimeDV => true))
```

where \( (:\{\text{finalTimeDV}\} \leftrightarrow \text{true}) \) indicates that the final time is a design variable, which is the case for the Bryson-Denham problem. Additional options can be passed to the `configure` function. However, this paper is not a tutorial; for a tutorial see NLOptControl’s documentation [9].

At this point, any path constraints in Eqn. 6 can be added to the model using JuMP’s `@NLconstraint` macro. However, these constraints are not needed for this example.

Next, the objective function in Eqn. 4 is added to the model. To accommodate for a Lagrangian term, NLOptControl provides the `integrate` function—similar to the `dynamics` function, an expression must be passed and the \( \{j\} \) syntax must be appended to all state and control variables. For the Bryson-Denham, the objective functional is modeled as

\[
\text{obj} = \int_0^1 a(t)^2 \, dt
\]

The JuMP macro `@NLobjective` is used to add the objective functional to the model as

```
@NLobjective(n, ocpi.mdl, Min, obj)
```

This problem is solved by passing the model \( n \) to the `optimize` function as

```
optimize!(n)
```

a) Visualization: NLOptControl allows users quickly plot the solutions to their problems. For plotting —by default— NLOptControl leverages GR as a backend, but it can be configured to utilize `matplotlib` instead. The command `allPlots(n)` plots the solution trajectories for the states, controls, and costates\(^1\). Invoking this command to visualize the solution to the Bryson-Denham problem that is modeled above produces Fig. 2.

B. Nonlinear model predictive control

Fig. 3 depicts two ways that NLOptControl can be used to solve OCPs for NMPC applications; Fig. 3a neglection control delays and Fig. 3b accounts for them. This section

\(^1\) if `n.s.ocp.evalCostates` is set to `true`
(a) Neglecting control delay $t_s$.

(b) Accounting for control delay $t_s$ using a fixed execution horizon $t_{ex}$ and a state prediction block.

Fig. 3: Nonlinear model predictive control framework available in NLOptControl.

The plant can be physical or virtual, but in either case is provided by the user. Because time can typically be allocated to initialize NMPC problems, the initialization block permits users to warm start their optimization problems so that the initial on-line solve time is much smaller. After initialization, at $t_0$, the first control signal $u(t)$ is sent to the plant and the first on-line OCP is solved. Each time an OCP-solve starts, $t_0$ is reset to the current time. An issue with this scheme is that it does not take into account the solve time (i.e., control delay $t_s$). That is, the initial state of the OCP is constrained to be the current state of the plant $x_0$ at the initial time $t_0$, so by the time the OCP has been solved $t_s$ has elapsed, and the plant will have evolved to a new state. If this control delay is small relative to the time scale of the dynamics, then neglecting it will not compromise the robustness. However, if the control delay is relatively large, then it cannot be neglected.

Fig. 3b illustrates an approach that accounts for these control delays. This approach adds a block that predicts the plant state at the current time plus a fixed execution horizon $t_0+t_{ex}$. The execution horizon $t_{ex}$ can be chosen based on the a heuristic upper limit on the solve times; often solve times do not change drastically when solved in a receding-horizon with varying parameters for the initial conditions and path constraints. This approach avoids having to predict individual solve times $t_s$.

NLOptControl provides various functionality tailored for solving NMPC problems. The remainder of this section simultaneously describes these features and provides an example that uses NLOptControl to formulate an OCP and solve it in a receding horizon. To this end, consider the moon lander OCP [51], which is given in without slack constraints in Eqn. 16 as

$$\text{minimize} \quad \int_0^{t_f} a(t) dt$$

subject to

$$\dot{x}(t) = v(t), \quad \dot{v}(t) = a(t) - g$$

$$x(t_0) = 10, \quad x(t_f) = 0$$

$$v(t_0) = -2, \quad v(t_f) = 0$$

$$0 \leq x(t) \leq 20, \quad -20 \leq v(t) \leq 20$$

$$0 \leq a(t) \leq 3, \quad 0.001 \leq t_f \leq 400$$

where the $x(t)$ is the altitude, $v(t)$ is the speed, $a(t)$ is the thrust, $g = 1.5$ is the local gravitational acceleration, $t_f$ is the final time. The objective is to minimize the thrust of the spaceship given the dynamic constraints, event constraints, control constraints, and final time constraints. Listing 1 shows the code needed to solve Eqn. 16 as an MPC problem. Line 1 creates a model $n$ of the OCP with the initial and terminal state constraints, and the constant upper and lower bounds on the state and control variables. As is, the model $n$ has low-tolerance hard constraints on the initial and terminal state conditions. However, these low-tolerance hard constraints can lead to infeasible problems and longer solve-times, especially when the loop is closed. That is, when the control drives the plant into an infeasible state space, an infeasible problem is engendered [52]; and typically, the solve-times increase as the problems become less-feasible. Therefore, an ability to easily adjust these low-tolerance constraints to high-tolerance hard constraints is desirable. As seen in Line 2, NLOptControl enables this feature through the \texttt{defineTolerances} function. \texttt{X0_tol} and \texttt{XF_tol} are arrays that set the tolerances on the initial $x_{0\text{tol}}$ and final states $x_{f\text{tol}}$, respectively.

When going from low- to high-tolerance hard constraints on the initial and terminal states, slack constraints should also be added. Because, when using these high-tolerance constraints without slack constraints, there is nothing pushing the initial and terminal states away from the edge of the infeasible region. Thus, infeasible problems are just as likely to occur. Adding slack constraints on the initial and terminal state constraints helps to mitigate these infeasible problems. Before slack constraints are added to the model, slack variables must...
be added. The size of a slack variable corresponds to the size of the respective constraint violation [53]. As seen in Line 4, NLOptControl allows such slack variable to be added using the configure function. (:xslackVariables => true) and (:x0slackVariables => true) adds slack variables on the initial and final state constraint, respectively. Both the objective of the moon lander problem and the slack constraints are added to model as on Line 6. n.ocep.x0s and n.xFs are arrays holding the slack variables on the initial and terminal states, respectively, and all of the terms in \( w_{s0} \) and \( w_{sf} \) (in Eqn. 4) are set to 100—these weights are set large enough such that the respective constraint violations are nearly zero. On Line 7, NLOptControl warm starts the optimization using the initOpt function; the initialization block in Fig. 3 captures this step.

The defineMPC function adds several basic settings to the model n. tex is the value of the fixed execution horizon and predictX0 is a bool, which, when set to true, indicates that the the framework in Fig. 3b is used. Thus, a prediction of the initial state needs to be made either by the user or using an internal model of the plant, which is added to n. In this simple example, the differential equations in Eqn. 16 govern the dynamics of the moon lander problem compared to the analytical solution.

Section B-A in the Appendices provides a plot NLOptControl’s closed-loop solution to the moon lander problem compared to the analytical solution.

IV. Numerical optimal control

This section provides an overview of numerical optimal control methods (i.e., transcription methods). The goal of this section is to motivate the choice of direct-collocation methods in NLOptControl, not to provide the reader with a complete description of numerical optimal control methods. Readers are referred to [55], [56], [43], for more comprehensive reviews on this subject. After these methods are discussed, the mathematics of the various direct-collocation methods as they are implemented in NLOptControl are provided.

A. Numerical optimal control overview

Tractable exact algorithms for solving OCPs suitable for practical applications do not exist; thus, numerical methods are used [57]. Numerical methods for solving OCPs (i.e., trajectory optimization problems) are generally broken into two categories: indirect and direct methods. Indirect methods seek the root of the necessary conditions for optimality [58] while direct methods seek the extrema of the cost functional [55]. Compared to direct methods, indirect methods produce better error estimates [12] and require less preliminary work to determine optimality [59]. However, indirect methods have several disadvantages: the necessary conditions must be derived [60]; the incorporation of path constraints requires an a priori estimation of the sequence of constrained/unconstrained singular arcs; and a guess needs to be made for the adjoint variables [43]. Due to these disadvantages, NLOptControl solves OCPs using direct methods.

---

Listing 1: NLOptControl code needed to formulate and solve the moon lander as an MPC problem.

```julia
n = define(numStates = 2, numControls = 1, X0 = [10., -2], XF = [0., 0.], CL = [0.], CU = [3.])
defineTolerances!(n; X0.tol = [0.01, 0.005], XF.tol = [0.01, 0.005])
dynamics!(n, :(x2[j]), :(u1[j]) - 1.5)
configure!(n; (:finalTimeDV => true), (:xslackVariables => true), (:x0slackVariables => true))
obj = integrate!(n; :(u1[j]))
@NLObjective(n.ocep.mdl, Min, obj + 100*(n.ocep.x0s[1] + n.ocep.x0s[2] + n.ocep.xFs[1] + n.ocep.xFs[2]))
initOpt!(n)
defineMPC!(n; tex = 0.2, predictX0 = true)
function IPplant(n, x0, t, U, t0, tf)
    spU = linearSpline(t, U[:,1])
    f = (dx, x, p, t) -> begin
        dx[1] = x[2]
        dx[2] = spU[t] - 1.5
    end
    return DiffEqBase.solve(ODEProblem(f, x0, (t0, tf)), Tsit5()), [spU]
end
defineIP!(n, IPplant)
simMPC!(n)
```
of the polynomial approximation types: h-methods (or local methods) [67], [68], [55], p-methods (or global methods) [69], [70], [71], and hp-methods (a hybrid of the h- and p-methods) [72], [1], [73]. In an h-method, the dynamic state constraints are satisfied using local approximations; e.g., Euler’s method or the trapezoidal method [67]. For h-methods, increasing the number and location of the collocation points [43], [1], [74] leads to convergence. However, a large number of points may be required for convergence, which can result in large solve-times [75]. p-methods can reduce the number of points needed for convergence, because they are more accurate than h-methods [76]. p-methods approximate OCPs using global polynomials constructed by collocating the dynamics at Gaussian quadrature points [76]. p-methods were originally developed to solve problems in computational fluid dynamics [77] and since have been used in practice in optimal control. For instance, p-methods were used to rotate the International Space Station 180 degrees without using any propellant\(^2\) [70]. A drawback with p-methods is that the Jacobian and Hessian matrices are much denser than with h-methods, which results in a larger NLP [78].

By construction, hp-methods help to mitigate the accuracy issues with h-methods and the NLP problem size with p-methods. Instead of using a single polynomial as with p-methods, hp-methods use multiple polynomials constrained to be connected to one another at the endpoints. This construction reduces the size of the NLP while maintaining accurate approximations [73].

### C. Direct-collocation methods in NLOptControl

At the time of this writing, three direct-collocation methods are implemented in NLOptControl: two h-methods and one p/hp-method. The remainder of this section illustrates how these methods are implemented in NLOptControl.

1) h-Methods: Euler’s backward method and the trapezoidal method are embedded in NLOptControl. However, before these h-methods are given, the h-discretization matrices used to approximate the continuous-time OCP are provided.

   a) h-Discretization Matrices: Consider that \( t \) is sampled at \( N \) evenly spaced discretization points \( t_i \in [t_0 + t_{ex}, t_f] \) and denote the result as the vector \( T = [T_1, \ldots, T_N] \). Then, for instance the \( t_0 + t_{ex} \) and \( t_f \) are defined as \( T_1 \) and \( T_N \), respectively. Denote the state and control discretization matrices as

\[
x(t) \bigg|_{t=T} = X
\]

and

\[
u(t) \bigg|_{t=T} = U,
\]

respectively. \( X[i] \) is the state at the \( i^{th} \) collocation point; thus, \( X[1] \) and \( X[N] \) index the values of the initial and final states, respectively. The control matrix is similarly defined; \( U[i] \) is the control at the \( i^{th} \) collocation point. Denote the minimum and maximum discretized state limit matrices as

\[
x_{min}(t) \bigg|_{t=T} = X_{min}
\]

\[
x_{max}(t) \bigg|_{t=T} = X_{max}
\]

\(^2\)The cost of the fuel saved was estimated at one million dollars and control of the space station orientation was accomplished using gyroscopes [5].
and 
\[ x_{\text{max}}(t) \bigg|_{t=T} = X_{\text{max}}, \]
respectively. Similarly, the minimum and maximum control limit matrices are denoted as 
\[ u_{\text{min}}(t) \bigg|_{t=T} = U_{\text{min}} \]
and 
\[ u_{\text{max}}(t) \bigg|_{t=T} = U_{\text{max}}, \]
respectively.

b) Euler’s Backward Method: The dynamic constraints in Eqn. 5 are locally approximated at \((N - 1)\) points defined by \(T[2 : N]\). To accomplish this, \((N - 1) \times n_{st}\) implicit constraints are added as shown in Eqn. 17
\[
\begin{align*}
0 &= X[i + 1] - X[i] - hF(X[i + 1], U[i + 1], T[i + 1]) \\
&= \eta_i, \text{ for } i \in (1 : N - 1)
\end{align*}
\]
where \(h\) is the time-step size, which is determined by dividing the time span \((t_f - t_0 - t_{xx})\) by \(N\).

The integral term in the cost functional in Eqn. 4 is approximated in Eqn. 18 as
\[
I = h \sum_{i=1}^{N} L(X[i], U[i], T_i)
\]
c) Trapezoidal Method: Similar to Euler’s backward method, the dynamic constraints in Eqn. 5 are locally approximated at \((N - 1)\) points defined by \(T[2 : N]\). To accomplish this, \((N - 1) \times n_{st}\) implicit constraints are added as shown in Eqn. 19
\[
\begin{align*}
0 &= X[i + 1] - X[i] - \frac{h}{2}(F(X[i], U[i], T[i]) + F(X[i + 1], U[i + 1], T[i + 1])) \\
&= \eta_i, \text{ for } i \in (1 : N - 1)
\end{align*}
\]
Next, the integral term in the cost functional in Eqn. 4 is approximated in Eqn. 20 as
\[
I = \frac{h}{2} \sum_{i=1}^{N} (L(X[i], U[i], T_i) + L(X[i + 1], U[i + 1], T_{i+1}))
\]
d) Discrete OCP: The h-method-based discrete OCP is given as
\[
\begin{align*}
\text{minimize} & \quad \mathcal{J}(X[1], T_1, X[N], T_N) + I \\
\text{subject to} & \quad \eta = 0 \\
& \quad C(X, U, T) \leq 0 \\
& \quad \phi(X[1], T_1, X[N], T_N) = 0 \\
& \quad X_{\text{min}} \leq X \leq X_{\text{max}} \\
& \quad U_{\text{min}} \leq U \leq U_{\text{max}} \\
& \quad t_{f_{\text{min}}} \leq T_{\text{N}} \leq t_{f_{\text{max}}}
\end{align*}
\]
where slack constraints can be included with the Mayer term in Eqn. 21 and Eqn. 23.

2) p-Methods: For generality, this paper only describes hp-methods, since the single interval method (i.e., p-method) is merely the case where the number of intervals is equal to one.

3) hp-Methods: The form of Eqn. 4 – Eqn. 11 must be modified to directly transcribe the OCP into an NLP using hp-methods. To apply Gaussian quadrature the interval of integration must be transformed from \([t_0 + t_{xx}, t_f]\) to \([-1, +1]\). To accomplish this, \(\tau \in [-1, +1]\) is introduced as a new independent variable and a change of variable, for \(t\) in terms of \(\tau\) using the affine transformation, \(t = \frac{t_f - t_0 + t_{xx}}{2} \tau + \frac{t_f + t_0 - t_{xx}}{2}\). Then, the interval \(\tau \in [-1, +1]\) is divided into a mesh of \(K\) intervals to accommodate for multiple intervals. With this, as in [75], an array of mesh points \((M_0, ..., M_K)\) for the boundaries of these intervals is defined, which satisfy
\[-1 = M_0 < M_1 < M_2 < \cdots < M_{K-1} < M_K = 1\]
Denote the continuous-time variables for the state and control are on each mesh interval, \(k \in (1, ..., K)\), by the arrays \(x^{(k)}(\tau)\) and \(u^{(k)}(\tau)\), respectively. Next, denote arrays of continuous-time variables for both the minimum and maximum state and control limits on each mesh interval, \(k \in (1, ..., K)\), as \(x_{\text{min}}^{(k)}, x_{\text{max}}^{(k)}, u_{\text{min}}^{(k)}, \text{ and } u_{\text{max}}^{(k)}\), respectively. The state continuity between the mesh intervals is ensured with the constraint \(x^{(k)}(M_k) = x^{(k+1)}(M_k)\) for \(k = (1, ..., K - 1)\) [75]. Similar to [1], this constraint is enforced programatically by making \(x^{(k)}(M_k)\) be the same variable as \(x^{(k+1)}(M_k)\). To continue to describe the hp-method implemented in NLoptControl, the hp-discretization matrices are defined, which hold the discrete-time values of the approximation to continuous-time problem.

a) hp-Discretization Matrices: First an array of time discretization vectors, \(\tau^{(k)} = [\tau^{(k)}_1, ..., \tau^{(k)}_{N_k}]\), is defined by evaluating the continuous functions at \(N_k\) specified \(\tau\)'s \(\in [M_{k-1}, M_k]\) for \(k \in [1, K]\), where \(N_k\) notates the number of collocation points in mesh interval \(k\); for instance, \(\tau^{(1)}_1 = -1\). Let
\[
N = [N_1, N_2, ..., N_k, ..., N_{K-1}, N_K]
\]
denote an array that holds the number of collocation points within each mesh interval, where \(N_k\) can be adjusted according to the desired level of fidelity for the \(k^{th}\) mesh interval. For \(k \in [1, ..., K]\), denote the state and control discretization matrix arrays as
\[
x^{(k)}(\tau) \bigg|_{\tau=\tau^{(k)}} = X^{(k)}
\]
and
\[
u^{(k)}(\tau) \bigg|_{\tau=\tau^{(k)}} = U^{(k)},
\]
respectively. Next, denote the minimum and maximum discretized state limit matrix arrays as
\[
x_{\text{min}}^{(k)}(\tau) \bigg|_{\tau=\tau^{(k)}} = X_{\text{min}}^{(k)}
\]
and
\[
x_{\text{max}}^{(k)}(\tau) \bigg|_{\tau=\tau^{(k)}} = X_{\text{max}}^{(k)},
\]
respectively. Similarly, the minimum and maximum control limit matrices are defined as
\[ U_{\text{min}}^{(k)}(\tau) \bigg|_{\tau=\tau^{(k)}} = U_{\text{min}}^{(k)} \]
and
\[ U_{\text{max}}^{(k)}(\tau) \bigg|_{\tau=\tau^{(k)}} = U_{\text{max}}^{(k)}, \]
respectively.

To approximate the modified OCP that is modified for hp-methods, \textsc{NLOptControl} builds on the work done in [69], [79], [30], which was implemented in \textsc{GPOPS-ii} [1]. Specifically, \textsc{NLOptControl} implements the Legendre-Gauss-Radau quadrature collocation method (Radau collocation method). For completeness, this section will briefly describe this method, but for a more thorough explanation, the reader is referred to the seminal work done in [1], [69], [79], [30].

b) Radau Collocation Method: In hp-methods, the states are approximated within each mesh interval with a Lagrange polynomial as
\[ x^{(k)}(\tau) = \sum_{j=1}^{N_k+1} X_j^{(k)} \mathcal{L}_j^{(k)}(\tau), \quad k \in [1, \ldots, K] \] (28)
with
\[ \mathcal{L}_j^{(k)}(\tau) = \prod_{i=1, i \neq j}^{N_k+1} \frac{\tau - \tau_i^{(k)}}{\tau_j^{(k)} - \tau_i^{(k)}}, \quad k \in [1, \ldots, K] \] (29)
\( \mathcal{L}_j^{(k)}(\tau) \) is the \((k^{th}, j^{th})\) Lagrange polynomial within a basis of Lagrange polynomials defined by \( j = (1, \ldots, N_k + 1) \) and \( k = (1, \ldots, K) \), \( \mathcal{L}_j^{(k)}(\tau) = [\tau_1^{(k)}, \ldots, \tau_{N_k}^{(k)}] \) and is the \( k^{th} \) set of the LGR collocation points (also, called LGR nodes [80]), which are defined on the \( k^{th} \) mesh interval (\( \tau \in [M_{k-1}, M_k] \)). Then to approximate the entire state, \( M_k \) is added as a noncollocated point [69] for \( k \in (1, \ldots, K) \).

The derivative of the state can then be approximated for each mesh interval as
\[ \frac{dx^{(k)}(\tau)}{d\tau} \approx \sum_{j=1}^{N_k+1} X_j^{(k)} \frac{d\mathcal{L}_j^{(k)}(\tau)}{d\tau}, \quad k \in [1, \ldots, K] \] (30)
with
\[ \frac{d\mathcal{L}_j^{(k)}(\tau)}{d\tau} \bigg|_{\tau=\tau^{(k)}} = D_{ij}^{(k)} \] (31)
where \( D_{ij}^{(k)} \) is an element of the \( N_k \times N_k+1 \) Legendre-Gauss-Radau differentiation matrix in the \( k^{th} \) mesh interval, as defined in [69].

Next, in order to approximate the integral of the Lagrange term in Eqn. 4, Gaussian-Legendre quadrature [81] is used as
\[ \int_{t_0+t_{ex}}^{t_f} L(x(t), u(t), t) dt \approx \frac{t_f - t_0 - t_{ex}}{2} \sum_{k=1}^{K} \frac{M_k - M_{k-1}}{2} \sum_{j=1}^{N_k} w_j^{(k)} L(X[j]^{(k)}, U[j]^{(k)}, \ldots, \tau_j^{(k)}, t_0 + t_{ex}, t_f) \] (32)
where \( u^{(k)} = [u_1^{(k)}, \ldots, u_{N_k}^{(k)}] \) is the \( k^{th} \) array of LGR weights.

Eqn. 32 is mathematically equivalent to the approximations made for the integral term in the cost functional in [73], but it is written in a slightly different form to reduce the computations needed within the NLP. Specifically, the \( \frac{M_k - M_{k-1}}{2} w_j^{(k)} \) term is calculated outside of the NLP, for \( j \in (1, \ldots, N_k) \) and \( k \in (1, \ldots, K) \). The result is stored in an array of vectors. Thus, the design variable \( t_f \) is removed from the summations in \textsc{NLOptControl}.

c) Discrete OCP: The p-method-based discrete OCP is shown in Eqn. 33 - Eqn. 39 as
\[ \text{minimize } \mathcal{M}(X[1]^{(1)}, t_0 + t_{ex}, X[N_{K+1}]^{(K)}, t_K) + I \]
subject to
\[ \sum_{j=1}^{N_k+1} X_j^{(k)} D_{ij}^{(k)} - \frac{t_f - t_0 - t_{ex}}{2} f(X_i^{(k)}, U_i^{(k)}, \tau_i^{(k)}, t_0 + t_{ex}, t_f) = 0 \] (34)
\[ C^{(k)}(X[i]^{(k)}, U[i]^{(k)}, \tau_i^{(k)}, t_0 + t_{ex}, t_f) \leq 0 \] (35)
\[ \phi(X[i]^{(1)}, t_0 + t_{ex}, X[N_{K+1}]^{(K)}, t_f) = 0 \] (36)
\[ X[i]^{(k)}_{\text{min}} \leq X[i]^{(k)} \leq X[i]^{(k)}_{\text{max}} \] (37)
\[ U[i]^{(k)}_{\text{min}} \leq U[i]^{(k)} \leq U[i]^{(k)}_{\text{max}} \] (38)
\[ t_f_{\text{min}} \leq t_f \leq t_f_{\text{max}} \] (39)
for \( (i = 1, \ldots, N_k) \) and \( (k = 1, \ldots, K) \).

4) Transforming to an NLP: Depending on the method, either the discrete OCP in Eqn. 21 - Eqn. 27 or the discrete OCP in Eqn. 33 - Eqn. 39 is then transformed into a large and sparse NLP given by Eqn. 1 - Eqn. 3.

Now that design and methods of \textsc{NLOptControl} have been provided, the following two sections compares its ease of use and speed to existing commonly used optimal control software.

V. EVALUATION DESCRIPTION

The next section compares \textsc{NLOptControl} and \textsc{PROPT} in terms of ease of use and speed. This section describes the conditions under which these comparisons are made.

A. Ease of use

Claiming that a software package is easy to use is subjective; even with the definition provided for ease of use, i.e., syntax that closely resembles the underlying OCP. Therefore, the respective syntax in \textsc{NLOptControl} and \textsc{PROPT} needed to model the moon lander OCP, as given in Eqn. 16, is compared.

B. Benchmark

The conditions under which \textsc{NLOptControl}’s speed is benchmarked against \textsc{PROPT} include the benchmark problem, methodology, and setup.

3To calculate both the LGR nodes and weights, \textsc{NLOptControl} leverages FastGaussQuadrature [82], [80], which uses methods developed in [83].
1) Benchmark problem: An OCP suitable for an NMPC-based ground vehicle application is used to benchmark NLOptControl against PROPT. The purpose of this problem is to find the steering and acceleration commands that drive a kinematic bicycle model \[84], \[85] to a goal location \((x_g = 0 \text{ m}, y_g = 100 \text{ m})\) as fast as possible (i.e., in minimum time) while avoiding crashing into a static obstacle. The cost functional is shown in Eqn. 40 as

\[
\text{minimize}_{ax(t), \alpha(t)} \ (x(t_f) - x_g)^2 + (y(t_f) - y_g)^2 + t_f \tag{40}
\]

The dynamic constraints are shown in Eqn. 41 as

\[
\begin{align*}
\dot{x}(t) &= u_x(t) \cos(\psi(t) + \beta(t)) \\
\dot{y}(t) &= u_x(t) \sin(\psi(t) + \beta(t)) \\
\dot{\psi}(t) &= \frac{u_x(t) \sin(\beta(t))}{l_b} \\
\dot{\alpha}_x(t) &= a_x(t)
\end{align*}
\]

where \(x(t)\) and \(y(t)\) are the position coordinates, \(\psi(t)\) is the yaw angle, \(u_x(t)\) is the longitudinal velocity, \(\alpha(t)\) is the steering angle, \(\beta(t) = \tan\left(\frac{\tan(\alpha(t))}{t_a}\right)^{-1}\), \(t_a = 1.58 \text{ m}\) and \(l_b = 1.72 \text{ m}\) are the distances from the center of gravity to the front and rear axles, respectively. The path constraints ensure that the vehicle avoids an obstacle, these constraints are shown in Eqn. 42 as

\[
1 < \frac{(x(t) - x_{obs})^2}{a_{obs} + m} + \left(\frac{y(t) - y_{obs}}{b_{obs} + m}\right)^2 \tag{42}
\]

where \(x_{obs} = 0 \text{ m}\) and \(y_{obs} = 50 \text{ m}\) denote the position of the center of the obstacle, \(a_{obs} = 5 \text{ m}\) and \(b_{obs} = 5 \text{ m}\) denote the semi-major and semi-minor axes, \(m = 2.5 \text{ m}\) is the safety margin that accounts for the footprint of the vehicle. The event constraints ensure that the vehicle starts at a particular initial condition, these constraints are given in Eqn. 43 as

\[
\begin{align*}
x(t_0) &= 0 \text{ m}, \quad y(t_0) = 0 \text{ m}, \quad \psi(t_0) = \frac{\pi}{2} \text{ rad} \\
\dot{x}(t_0) &= 15 \frac{\text{ m}}{\text{ s}}, \quad a_x(t_0) = 0 \frac{\text{ m}}{\text{ s}^2}, \quad \alpha(t_0) = 0 \text{ rad}
\end{align*} \tag{43}
\]

That is the vehicle is traveling straight ahead at a constant velocity of \(15 \frac{\text{ m}}{\text{ s}}\). The state and control bound constraints are given in Eqn. 44 as

\[
\begin{align*}
-100 \text{ m} &\leq x(t) \leq 100 \text{ m}, \quad -0.01 \text{ m} \leq y(t) \leq 120 \text{ m} \\
-2\pi \text{ rad} &\leq \psi(t) \leq 2\pi \text{ rad}, \quad \frac{5 \text{ m}}{\text{ s}} \leq u_x(t) \leq 29 \frac{\text{ m}}{\text{ s}} \\
-2 \frac{\text{ m}}{\text{ s}^2} &\leq a_x(t) \leq 2 \frac{\text{ m}}{\text{ s}^2}, \quad -30\frac{\pi}{180} \text{ rad} \leq \alpha(t) \leq 30\frac{\pi}{180} \text{ rad}
\end{align*} \tag{44}
\]

The final time is constrained to be \(0.001 \leq t_f \leq 50 \text{ s}\).

Solutions to Eqn. 40 - Eqn. 44 that are obtained in less than 0.5 s are deemed to be fast enough for real-time NMPC.

2) Benchmark methodology: Using the problem described above, a comprehensive benchmark is made between various solvers. A solver is defined by a particular combination of either NLOptControl or PROPT in conjunction with a particular direct-collocation method. The set of solvers \(S\) are listed in Table II as

| Legend label Description | |
|-------------------------|--|
| NLOptControl with LGR nodes with a single interval | NLOptControl with LGR nodes with two intervals |
| NLOptControl with LGR nodes with four intervals | NLOptControl using Euler’s method |
| NLOptControl using trapezoidal method | PROPT with Chebyshev nodes with a single phase |
| PROPT with Chebyshev nodes with two phases | PROPT with Chebyshev nodes with four phases |

Comparisons between the average solve-times of single interval/phase solvers (i.e., NLOptC, NLOptT, NLOptLGR, PROPT) and the multiple interval/phase solvers (i.e., NLOptLGR, NLOptLGR, PROPTC2, PROPTC4) must be considered in context. This is true because as the number of collocation points per interval/phase is increased, the two interval/phase solvers (i.e., NLOptLGR and PROPTC4) and the four interval/phase solvers (i.e., NLOptLGR and PROPTC4) are solving problems that roughly two and four times larger than the single interval/phase solvers, respectively. However, there are advantages of these multi interval/phase solvers, as discussed previously, that may be more important than the decreases in solve times. Thus, these solvers are included in the comparison here for a more comprehensive comparison.

Comparisons between the average solve-times of the multiple interval solvers in NLOptControl and the multiple phase solvers in PROPT also require consideration. Ideally, the benchmark between PROPT and NLOptControl would include the same direct-collocation methods. Unfortunately, PROPT and NLOptControl do not have the same direct-collocation methods. As such, comparisons are made between single/multiple phase Chebyshev pseudospectral methods in PROPT and multiple single/interval LGR pseudospectral methods in NLOptControl. Unlike a multiple interval method, in a multiple phase method, between phases, the constraints can change and the optimal transition time can be determined. In this work, the constraints do not change and the final time is divided evenly by the number of phases to determine the transition time. By doing this, the OCPs formulated by the multiple phase and multiple interval methods have roughly the same size and level of complexity. Thus, comparisons between the two software packages can be made with this issue in mind.

Each solver \(s\) is used to solve a set of problems \(P\). The benchmark problem is discretized over the range of collocation points \(p = 2, 3, \ldots, 102\) per interval or phase to realize the set of problems \(P\) tested for each solver; a total of 101 different values of \(p\) (i.e., levels-of-fidelity or problems) are tested. Each test is performed three times to provide the data needed to calculate the average solve-time \(t_{s,p}\) for the benchmark problem with a level-of-fidelity \(p\) using solver \(s\). A polynomial is interpolated through the \((x, y)\) solution points and sampled at 200 points to determine if the solution drives the vehicle through the obstacle. If a collision is determined for a particular combination of solver \(s^*\) and level-of-fidelity \(p^*\), then \(t_{s^*,p^*}\) is set to \(\text{NaN}\); such solutions are not practically feasible.

Conducting many benchmark tests helps accurately rank the solvers. However, analyzing large sets of benchmark data
can be overwhelming and the conclusions drawn from such analyses can be subjective. To help eliminate these issues, this work uses an optimization software benchmarking tool called performance profiles [86].

Performance profiles show the distribution function for a particular performance metric. Here, the performance metric is the ratio of the solver’s average solve-time to the best average solver solve-time given as

$$r_{s,p} = \frac{t_{s,p}}{\min(t_{s,p} : s \in S)}$$

where this performance metric is calculated for each solver $s$ at each level-of-fidelity $p = 2, 3, \ldots, 102$. If a solver does not solve a particular problem, then $r_{s,p}$ is set to $r_M$. $r_M$ is chosen to be a large positive number; the choice of $r_M$ does not affect the evaluation [86].

To assess a solver’s overall performance on the set of problems, the cumulative distribution function for the performance ratio is defined as

$$P_s(\Gamma) = \frac{1}{101} \sum_{p \in P \setminus \Delta p} \min(r_{s,p} : s \in S)$$

where $P_s(\Gamma)$ is the probability that solver $s$ can solve problem $p$ within a factor $\Gamma$ of the best ratio.

3) Setup: The setup is defined by the hardware platform and software stack. The results in this paper are produced using a single machine running Ubuntu 16.04 with the following hardware characteristics: an Intel Core i7 - 4910MQ CPU @ 2.90GHz × 8, and 16GB of RAM. For software, both NLOptControl 0.1.5 and PROPT use KNITRO 10.3 for the NLP solver with the default settings, except the maximum solve-time, which is set to 300 s.

VI. RESULTS

A. Ease of use

Listing 2 and Listing 3 show the respective syntaxis in NLOptControl and PROPT needed to model the moon lander OCP in Eqn. 16. Section B in the Appendices shows NLOptControl’s and PROPT’s solutions compared to the analytical solution.

Listing 2: NLOptControl code needed to formulate and solve the moon lander problem. The ! character indicates that the function is modifying the model.

```python
n = define(numStates = 2, numControls = 1, X0 = [10, -2], XF = [0., 0.], XL = [0., -20], XU = [20, 20], CL = [0., 3.], CU=[3.])
config = ![n,:(x2[j]), :(u1[j] - 1.5))]
configure!(n;{:(finalTimeDV => true)});
obj = integrate!(n; :(u1[j]));
@NLOptimize!(n. ocp.mdl, Min, obj);
```

NLOptControl can model OCPs more succinctly than PROPT. NLOptControl models Eqn. 16 with 5 lines of code, while it takes PROPT 12 lines — there are two main reasons for this: (1) it takes PROPT 4 lines of code to include the initial and final state conditions, and the upper and lower limits of the states and controls, while this is accomplished with a single line of code, with NLOptControl, and (2) several of PROPT’s features are required, while in NLOptControl they are optional; these features include an initial guess, an options structure, and the naming of the state and control variables. Additionally, PROPT has more verbose syntax than NLOptControl — PROPT’s collocate(), initial(), and final() functions require many characters per line of code.

B. Speed

The performances of the solvers in Table II are now examined on the set of problems realized by various discretizations of Eqn. 40 – Eqn. 44, as described in Sec. V. The results for these examinations are in Fig. 5a and Fig. 5b. Fig. 5a shows the performance, or average solve-times $t_{s,p}$, for each solver $s$ on each problem $p$. Fig. 5b shows the performance profiles for all of the solvers in four ranges of interest for $\Gamma$. Each range is on a separate plot. The purpose of this section is to (1) show the raw benchmark data in Fig. 5a and (2) provide an objective analysis of this data in Fig. 5b. In the following section, this information will be used to draw conclusions regarding the speed of NLOptControl and the best solver for the benchmark problem.

Fig. 5a shows that NLOptControl’s solvers are faster than PROPT’s. At a high-level, NLOptControl solves 88% of the problems in real-time using h-methods (i.e., NLOptE and NLOptT) and 46% of the time using p/h-methods (i.e., NLOptLGR2, NLOptLGR4, and NLOptLGR6). PROPT only solves 0.05% of the problems in real-time using p/h-methods (i.e., PROPTC1, PROPTC2, and PROPTC4). At a lower-level, the zoomed-in subplot in the the bottom graph of Fig. 5a shows that NLOptControl solves the benchmark problem in real-time when the number of collocation points per interval is less than: 80 for the single-interval case; 45 for the two-interval case; and 25 for the four-interval interval case. PROPT obtains real-time solutions when the number of collocation points per phase is less than 27 for the single-phase case and less than 4 for the two-phase case. For the four-phase case, PROPT cannot solve any of the problems in real-time.

Fig. 5a also shows that as the number of intervals/phases increase from NLOptLGR1 to NLOptLGR4 and PROPTC1 to PROPTC4, the solve-times increase exponentially. Due to the large solve-times with PROPT’s solvers, these trends can only be seen in the top graph of Fig. 5a — the bottom graph shows the trends for NLOptControl’s solvers. As discussed in the previous section, this increase in solve time is largely due to the fact that with an increase in the intervals/phases larger problems are created and they take longer to solve. Even though the NLOptLGR4 solver is solving a problem that is roughly four times larger than the PROPTC1 solver, NLOptLGR4, results in smaller solve-times.

Fig. 5a also shows that h-methods in NLOptControl are faster than the p-method for the benchmark problem. As the level-of-fidelity increases, the solve-times increase linearly with h-methods and exponentially with the p-method. Additionally, the number of collocation points needs to be greater than about 20 for the h-methods to ensure collision avoidance,
while the p-methods need 23 and 27 for NLoptControl and PROPT, respectively.

The four plots in Fig. 5b show the ranges of $\Gamma$ wherein certain solvers dominate. Each profile in this figure shows the probability $P$ that a given solver $s$ will solve the set of problems $P$ the fastest within a factor of $\Gamma$. At $\Gamma = 1$, the solver that has the highest probability of being the fastest is NLopt$_E$, with a probability of 0.881. NLopt$_E$ dominates until about $\Gamma = 1.8$, at which point NLopt$_T$ has the highest probability of being the fastest, with a probability of 0.891. NLopt$_T$ dominates until about $\Gamma = 80$. The remaining approximate ranges of domination are as follows: NLopt$_{LGR_2}$ from 80 to 160, NLopt$_{LGR_4}$ from 160 to 5,000, PROPT$_{C_4}$ from 5,000 onwards. Given enough time, PROPT$_{C_4}$ solves 100% of the problems. For this benchmark problem, while the NLopt$_T$ and NLopt$_E$ solvers are much faster than the NLopt$_{LGR_2}$, NLopt$_{LGR_4}$, and PROPT$_{C_4}$ solvers they are not as reliable. However, the NLopt$_T$ and NLopt$_E$ solvers are both faster and more reliable than the NLopt$_{LGR_2}$, PROPT$_{C_1}$, and PROPT$_{C_2}$ solvers.

### VII. DISCUSSION

The approach detailed in Section II yields a direct-collocation-based optimal control modeling language that is both faster and easier to use than PROPT. The results and the following discussion support this claim.

NLoptControl is easier to use than PROPT, because its syntax is more concise, and focused on building a model of the OCP in Bolza form. Differences between Listing. 2 and Listing. 3, in terms of number of lines of code and the number of characters per line of code, indicate that NLoptControl models OCPs more succinctly than PROPT. This work speculates that PROPT requires more lines of code to formulate other more practical problems as well.

In addition to PROPT’s verbosity, its syntax is flexible to the extent that modeling errors are easier to make. This claim is made because its users can more easily formulate problems that do not fit into the Bolza OCP form. As an example, consider using PROPT to model the dynamic constraints in Eqn. 5 for the moon lander problem — Line 7 in Listing. 3. When using PROPT, if the user were to forget to include the second differential equation as

\[
\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x}, t, \mathbf{u})
\]

an error would not be displayed; such overly flexible syntax can lead to modeling errors. If that same mistake were attempted in NLoptControl, the user would be alerted as

```julia
julia> dynamics!(n, [:x2[j]]);
ERROR: The number of differential equations must equal ocp.state.num.
```

Thus, NLoptControl helps avoid modeling errors better than PROPT, because NLoptControl’s syntax does not allow users to formulate problems that are not in the Bolza form, while PROPT’s syntax does.

Both NLoptControl and PROPT can be used formulate OCPs, but PROPT takes a functional approach to this task rather than a modeling approach, as with NLoptControl. Listing. 2 is compared to Listing. 3 to support this claim. Listing. 3 shows that, with PROPT, the user creates all of the components of the OCP and finally assembles them on Line 12. With NLoptControl, in Listing. 2, it is clear from the first line of code that a model named $n$ is being built. Using this approach, NLoptControl can clearly model and solve multiple OCPs at once. Such an object-oriented approach can further reduce potential modeling errors.

The benchmark results in Fig. 5a and Fig. 5b show that NLoptControl is faster than PROPT. Differences between these packages that affect speed include: differentiation methods, underlying computational language, and available direct-collocation methods.

PROPT uses symbolic automatic differentiation to calculate the derivatives. However, the structure of the Hessian matrices born from approximating an OCP using direct-collocation methods is sparse and symbolic automatic differentiation does not exploit this structure for speed. In contrast, NLoptControl uses the acyclic-coloring method to exploit the sparse structure of the Hessian matrix in conjunction with reverse automatic differentiation. Based on this difference, NLoptControl is expected to be faster than PROPT, especially when solving large problems that have a very sparse structure.

PROPT’s differentiation methods are implemented in MATLAB and NLoptControl’s are implemented in Julia. Unfortunately, the literature does not contain benchmarks of the form.

### Listing 3: PROPT code needed to formulate and solve the moon lander problem

```python
import tomograph
import tomodesign
import tomolab
import tomocollocate
import tomopartial
import tomocollate
import tomocore
import tomomodel
import tomomodels
import tomoode
import tomoopt
import tomovars

# Problem definition

toms t t.f
p = tomophase('p', t, 0, t_f, 30);
setPhase(p);
tomostates x v
tomocontrols a
cbox = {{0.001<=t_f<=400, 0<=icollocate(x)<=20, -20<=icollocate(v)<=20, 0<=collocate(a)<=3}};
do = icollocate\(\{\dot{x}==v, \ddot{x}==-1.5+a\}\); cbox = icollocate\(\{a==0\}\);

tomodesign \(\{\text{objective = \integrate(setPhase(p))}\};
result = collocate(ode);
tomolab x0 = \{t_f == 1.5, \}\;final(x == 0; final(v == 0));
objective = \text{integrate}(a);
options = struct;
prob = sym2prob\(\{\text{objective, \{cbox, cbox, ode, cbox\}, x0, options}\};
result = tomoRun\('knitro', prob, 1);```

13
each of these differentiation methods in both MATLAB and Julia. However, research has shown that Julia is much faster than MATLAB for a wide range of problem types [33]. Thus, Julia may be able to run the reverse automatic differentiation method combined with the acyclic-coloring method to identify sparsity in the Hessian matrix faster than MATLAB—if it were implemented in MATLAB.

Overall, this paper speculates that NLOptControl’s unique combination of differentiation methods and computational language makes it faster than PROPT. This only a speculation since the direct-collocation methods are different between NLOptControl and PROPT. However, the following pairs of solvers can be considered roughly equivalent in terms of their direct-collocation methods: NLOptLGR, and PROPTC1, NLOptLGR, and PROPTC2, and NLOptLGR, and PROPTC4. Between these pairs, NLOptControl solves the problem roughly 14, 26, and 36 times faster than PROPT, respectively. It is unlikely that these large differences are due to either differences between collocating at Chebyshev nodes vs. LGR nodes, multiple interval vs. multiple phase methods, or some combination of the two. Thus NLOptControl is fast, which is especially important for MPC applications. A brief discussion of NLOptControl’s salient MPC functionality follows.

NLOptControl has optional functionality that helps account for the non-negligible solve-times in MPC applications. In MPC, often the control delay (i.e., solve-time) is neglected [19]. When the control delay is neglected, the current state of the plant is used to initialize the problem as opposed to initializing the problem with a prediction of what the plant’s state will be after the solve-time has elapsed. Typically neglecting the solve-time in linear model predictive control is not an issue; because in linear MPC, the quadratic program is solved so quickly that initial state of the trajectory can be set to the current state of the plant without compromising robustness [87]. However, neglecting the solve-time in NMPC is likely to deteriorate robustness [88], because the NLPs often take a non-negligible amount of time to solve, after which the state of the plant will have evolved significantly. To make matters even more challenging, ensuring that the NLP solve-times are smaller than a particular execution horizon remains an unsolved problem [89], [90], [91], [87], [92], [93]. Fortunately, for many problems, these NLP solve-times are similar and an upper limit determined based on experience. This upper limit can be used to determine a fixed execution horizon. In NLOptControl, after the user selects an execution horizon, as described in Section. III, the framework in Fig. 3b can be used to account for non-negligible solve times. Frameworks such as this, can help establish conceptual schemes to improve safety and performance in NMPC applications.

VIII. Conclusions

This paper introduces an open-source, direct-collocation method based OCP modeling language called NLOptControl. NLOptControl extends the JuMP optimization modeling language to include a natural algebraic syntax for modeling OCPs. NLOptControl is compared
against PROPT in terms of ease of use and speed. PROPT’s syntax is shown to be more verbose and error-prone than NLOptControl’s; thus NLOptControl is easier to use than PROPT. This ease of use is largely attributed to NLOptControl’s use of the JuMP optimization modeling language. In addition to being easier to use, results from the benchmark tests show that NLOptControl is much faster than PROPT. NLOptControl’s superior performance is likely due to the unique utility of the Julia programming language and the reverse automatic differentiation method in conjunction with the acyclic-coloring method to exploit the sparsity of the Hessian matrices. NLOptControl emerges as an easy to use, fast, and open-source [9] optimal control modeling language that holds great potential for not only improving existing off-line and on-line control systems but also engendering a wide variety of new ones.

ACKNOWLEDGEMENTS
This work was supported by the Automotive Research Center (ARC) in accordance with Cooperative Agreement W56HZV-14-2-0001 U.S. Army Tank Automotive Research, Development and Engineering Center (TARDEC) Warren, MI. Benoit Legat provided a thoughtful review of this paper and discussions with the several Julia developers including Miles Lubin, Tony Kelman, and Chris Rackauckas were helpful.

REFERENCES
[1] M. A. Patterson, A. V. Rao, Gpops-ii: A matlab software for solving multiple-phase optimal control problems using hp-adaptive gaussian quadrature collocation methods and sparse nonlinear programming, ACM Transactions on Mathematical Software (TOMS) 41 (1) (2014) 1.
[2] J. P. Sanchez, D. Garcia Yarnoz, Asteroid retrieval missions enabled by invariant manifold dynamics, Acta Astronautica 127 (2016) 667 – 677, asteroid mission; Easily retrievable objects; Libration point orbits; Low thrust; Trajectory designs; URL http://dx.doi.org/10.1016/j.actaastro.2016.05.034
[3] M. M. E. Per E. Rutquist, PROPT - Matlab Optimal Control Software, Tomlab Optimization Inc., 1260 SE Bishop Blvd Ste E, Pullman, WA 99163, USA, 1st Edition (June 2016). URL https://tomopt.com/docs/TOMLAB_PROPT.pdf
[4] T. Torsis, C. Schulz, F. Friedl, A. Rao, Constrained trajectory optimization using pseudospectral methods, in: AIAA Atmospheric Flight Mechanics Conference and Exhibit, 2008, p. 6218.
[5] W. Kang, N. Bedrossian, Pseudospectral optimal control theory makes debut flight, saves nasa 1 m in under three hours (2007).
[6] A. Holmqvist, F. Magnusson, Open-loop optimal control of batch chromatographic separation processes using direct collocation, Journal of Process Control 46 (2016) 55–74.
[7] T. Utstomo, T. W. Berge, J. T. Gravdahl, Non-linear model predictive control for constrained robot navigation in row crops, in: Industrial Technology (ICIT), 2015 IEEE International Conference on, IEEE, 2015, pp. 357–362.
[8] J. V. Frasch, A. Gray, M. Zanon, H. J. Ferreau, S. Sager, F. Borrelli, M. Diehl, An auto-generated nonlinear mpc algorithm for real-time obstacle avoidance of ground vehicles, in: European Control Conference, IEEE, 2013, pp. 4136–4141.
[9] H. Fedbo, NLOptControl, https://github.com/JuliaMPC/NLOptControl.jl (2017).
[10] A. V. Rao, D. A. Benson, C. Darby, M. A. Patterson, C. Francolin, I. Sanders, G. T. Huntington, Algorithm 902: Gpops, a matlab software for solving multiple-phase optimal control problems using the gauss pseudospectral method, ACM Transactions on Mathematical Software (TOMS) 37 (2) (2010) 22.
[11] V. M. Becerra, Solving complex optimal control problems at no cost with psopt, in: Computer-Aided Control System Design (CACSD), 2010 IEEE International Symposium on, IEEE, 2010, pp. 1391–1396.
with \texttt{NLOptControl} using a single interval with 30 LGR nodes. \texttt{NLOptControl} calculates these trajectories reasonably well.

\textbf{APPENDIX B}

\textbf{MOON LANDER PROBLEM}

\textbf{A. Closed-loop}

Fig. 7 shows the closed-loop solution to the moon lander problem using \texttt{NLOptControl}. The closed-loop trajectory of the plant is very close to the analytic solution. Additionally, all of the solve-times are well below the chosen execution horizon $t_{ex}$ of 0.2 s; thus \texttt{NLOptControl} solves this NMPC problem in real-time.

The NLP solver for this example is \texttt{IPOPT} and an hp-method in \texttt{NLOptControl} is used with 4 intervals and 10 LGR nodes.

\textbf{B. Open-loop}

In Fig. 8, it can be seen that both \texttt{NLOptControl} and \texttt{PROPT} determine the analytic solution accurately, with 30 LGR and Chebyshev nodes, respectively. However, there is an overshoot in the solution of the control with both \texttt{NLOptControl} and \texttt{PROPT}. This is due to the bang-bang nature of the analytic solution. It is noted that this overshoot may be mitigated using either mesh refinement \cite{72,75,74} or radial basis functions \cite{94}.

\textbf{APPENDIX C}

\textbf{BENCHMARK PROBLEM}

This section provides an example of the type of solutions that are obtained from the benchmark between \texttt{NLOptControl} and \texttt{PROPT}. For \texttt{NLOptControl}, the hp-method with LGR nodes and four intervals and 10 collocation points per interval is used. \texttt{PROPT} is set to use four phases and 10 collocation points per phase and Chebyshev nodes. Fig. 9 compares the results of these solvers, where it can be seen that position trajectories are close. Starting at a speed of $15 \frac{m}{s}$, the solutions obtained from both \texttt{PROPT} and \texttt{NLOptControl} apply maximum acceleration from $t_0 = 0$ s to $t_f = 5.1$ s while avoiding collision with the obstacle and reaching the desired goal position. The trajectories for \texttt{NLOptControl} exhibit large oscillations in the $\alpha$ trajectory. These oscillations may be an artifact of the Runge phenomenon and seem to be reduced with \texttt{PROPT} as it uses Chebyshev nodes.

There is no analytic solution to this problem.
Fig. 8: State and control trajectories using NLOptControl (with 30 LGR nodes) and PROPT (with 30 Chebyshev nodes) compared to the analytic solution for moon lander problem.

Fig. 9: State and control trajectories using NLOptControl (with 4 intervals and 10 LGR nodes) and PROPT (with 4 intervals and 10 Chebyshev nodes) for the kinematic ground vehicle problem.