OptimTraj Users Guide
Version 1.5

Matthew P. Kelly

May 3, 2018

1 Introduction

OptimTraj is a matlab library designed for solving continuous-time single-phase trajectory optimization problems. I developed it while working on my PhD at Cornell, studying non-linear controller design for walking robots.

1.1 What sort of problems does OptimTraj solve?

Examples:

- Cart-Pole Swing-Up: Find the force profile to apply to the cart to swing-up the pendulum that freely hangs from it.
- Compute the gait (joint angles, rates, and torques) for a walking robot that minimizes the energy used while walking.
- Find a minimum-thrust orbit transfer trajectory for a satellite.

Details:

OptimTraj finds the optimal trajectory for a dynamical system. This trajectory is a sequence of controls (expressed as a function) that moves the dynamical system between two points in state space. The trajectory will minimize some cost function, which is typically an integral along the trajectory. The trajectory will also satisfy a set user-defined constraints. All functions in the problem description can be non-linear, but they must be smooth (C^2 continuous). OptimTraj solves problems with

- continuous dynamics
- boundary constraints
- path constraints
- integral cost function
- boundary cost function

1.2 Features:

- **Easy to Install:** no dependencies outside of Matlab [1]
- **Lots of example:** look at the `deom/` directory to see for yourself!
- **Readable source code:** easy to debug your code and figure out how the software works
- **Analytic gradients:** most methods support analytic gradients
- **Rapidly switch between methods:** helpful for debugging and experimenting
  - Trapezoidal Direct Collocation

- Hermite-Simpson Direct Collocation
- Runge–Kutta 4^{th}-order Multiple Shooting
- Chebyshev–Lobatto Orthogonal Collocation

1.3 Installation:

1. Clone or download the repository (`https://github.com/MatthewPeterKelly/OptimTraj`)
2. Add the top level folder to your Matlab path
3. (Optional) Clone or download ChebFun (`http://www.chebfun.org`) to use the Chebyshev–Lobatto method

1.4 Usage:

- Call the function `optimTraj` from inside matlab.
- `optimTraj` takes a single argument: a struct that describes your trajectory optimization problem.
- `optimTraj` returns a struct that describes the solution. It contains a full description of the problem, the transcription method that was used, and the solution (both as a vector of points and a function handle for interpolation).
- For more details, type `>> help optimTraj` at the command line, or check out some of the examples in the `demo/` directory.

1.5 License:

2 Using OptimTraj

There is a single calling sequence for using OptimTraj: **`soln = optimTraj(problem)`**. The input **`problem`** is a struct that describes a trajectory optimization problem, and the output **`soln`** is a struct that gives details regarding the solution to the trajectory optimization problem. This section is similar to the help file for **`optimTraj()`**.

[1] Chebyshev–Lobatto method requires ChebFun (`http://www.chebfun.org`)

2.1 Notation

Throughout this section we will use t for time, $\boldsymbol{x}$ for state, and $\boldsymbol{u}$ for control. We will use N for an integer, where N_t is the number of grid-points along the trajectory and N_g is the number of grid-points in the initial guess. The dimension of the state is given by N_x and the dimension of the control is given by N_u. We will use this notation: **problem.func** to indicate that the field **func** must be typed exactly as shown, while the struct **problem** can be named anything.

We will use t_0 and t_F to indicate the initial and final times. Similarly, $\boldsymbol{x}_0 = \boldsymbol{x}(t_0)$ and $\boldsymbol{x}_F = \boldsymbol{x}(t_F)$ are the initial and final state. Throughout the software, we use the convention that time is a row vector, while the dimensions of state and control are a column vector. Thus,
size(t) = [1, N_t],
size($\boldsymbol{x}$) = [N_x, N_t],
size($\boldsymbol{u}$) = [N_u, N_t],
size($\boldsymbol{x}_0$) = size($\boldsymbol{x}_F$) = [N_x, 1].

2.2 Constructing the Input to OptimTraj

The input to **optimTraj()** is a single struct, which we will call the **problem** struct. The **problem** struct has four fields. The **problem.func** struct contains a set of user-defined function handles to the dynamics, objective, and constraint functions. The **problem.bounds** struct contains all constant bounds on trajectory (time, state, and control). The **problem.guess** struct contains an initial guess for the trajectory. Finally, the **problem.options** struct contains options for both **optimTraj()** and **fmincon()**, which it calls to solve the underlying non-linear program.

problem.func

There are five fields in the **problem.func** struct, each of which is a user-defined function handle. The only mandatory field is `func.dynamics`, and at least one of either `func.pathObj` or `func.bndObj`. All other fields can simply be omitted or left empty `[]`. Here we will list the prototype for each function handle: `outArgs = funName(inArgs)`. The user can pass additional parameters using the Matlab anonymous function syntax: `funHandle = @( funName(inArgs, extraParams) );`.

Both of the direct collocation methods (trapezoidal and Hermite-Simpson) support analytic gradients. The details regarding the construction of these function handles is provided in the help files for each of these methods: `>> help trapezoid` and `>> help hermiteSimpson`. If you're using analytic gradients, then you should look at the `demo/gradientsTutorial` for an simple tutorial example and `demo/fiveLinkBiped` for a complicated example.

- **problem.func.dynamics** : $\rightarrow$ $\dot{\boldsymbol{x}}$ = **dynamics**($t, \boldsymbol{x}, \boldsymbol{u}$)
  where $\dot{\boldsymbol{x}}$ is the time derivative of $\boldsymbol{x}$ and size($\dot{\boldsymbol{x}}$)==size($\boldsymbol{x}$)
- **problem.func.pathObj** : $\rightarrow$ $\boldsymbol{J}_P$ = **pathObj**($t, \boldsymbol{x}, \boldsymbol{u}$)
  where $\boldsymbol{J}_P$ is the integrand of the objective function and size($\boldsymbol{J_P}$)==size(t)
- **problem.func.pathCst** : $\rightarrow$ $[\boldsymbol{C}_P^a, \boldsymbol{C}_P^b]$ = **pathCst**($t, \boldsymbol{x}, \boldsymbol{u}$)
  where $\boldsymbol{C}_P^a \leq \boldsymbol{0}$ is path inequality constraint, and $\boldsymbol{C}_P^b = \boldsymbol{0}$ is the path equality constraint. Either may be left empty `[ ]`. The number of columns in each must equal that of time, but the number of rows in each is arbitrary (it just must be consistent between function calls).
- **problem.func.bndObj** : $\rightarrow$ J_B = **bndObj**($t_0, \boldsymbol{x}_0, t_F, \boldsymbol{x}_F$)
  where J_B is a scalar cost associated with the boundary points of the trajectory.
- **problem.func.bndCst** : $\rightarrow$ $[\boldsymbol{C}_B^a, \boldsymbol{C}_B^b]$ = **bndCst**($t_0, \boldsymbol{x}_0, t_F, \boldsymbol{x}_F$)
  where $\boldsymbol{C}_B^a \leq \boldsymbol{0}$ is boundary inequality constraint, and $\boldsymbol{C}_B^b = \boldsymbol{0}$ is the boundary equality constraint. Either may be left empty `[]`. Each is a column vector of arbitrary length, provided that it is consistent between function calls.

problem.bounds

The bounds struct provides constant bounds on the state and control along the trajectory, as well as the time and state on the boundaries. All fields are either scalar or a column vector, and can be omitted (or left empty) if not needed. If you need to include a bound on only part of the state or control, then set the remaining entries to $\pm\infty$. For example: **bounds.state.low = [0;-inf;0;-inf];** sets a bound only for the first and third element of the state vector. All entries relating to time are scalar, entries relating to state $\boldsymbol{x}$ are column vectors of length N_x, and entries relating to control $\boldsymbol{u}$ are column vectors of length N_u.

- **problem.bounds.initialTime.low** = t_0^-
- **problem.bounds.initialTime.upp** = t_0^+
- **problem.bounds.finalTime.low** = t_F^-
- **problem.bounds.finalTime.upp** = t_F^+
- **problem.bounds.initialState.low** = $\boldsymbol{x}_0^-$
- **problem.bounds.initialState.upp** = $\boldsymbol{x}_0^+$
- **problem.bounds.finalState.low** = $\boldsymbol{x}_F^-$
- **problem.bounds.finalState.upp** = $\boldsymbol{x}_F^+$
- **problem.bounds.state.low** = $\boldsymbol{x}^-$
- **problem.bounds.state.upp** = $\boldsymbol{x}^+$
- **problem.bounds.control.low** = $\boldsymbol{u}^-$
- **problem.bounds.control.upp** = $\boldsymbol{u}^+$

problem.guess

The guess struct provides the optimization with an initialization. All fields are mandatory. Internally, `optimTraj` uses the guess struct to determine the dimension of the state and control. The number of grid points in the guess struct does not correspond to the number of grid points in the solution. Instead, `optimTraj` constructs the solution grid using information from the options struct, and then uses interpolation of the data in guess to evaluate the initial value of the solution grid.

- **problem.guess.time** = t_g `size(`t_g`) = [1,` N_g`]`
- **problem.guess.state** = $\boldsymbol{x}_g$ `size(`$\boldsymbol{x}_g$`) = [`N_x`,` N_g`]`
- **problem.guess.control** = $\boldsymbol{u}_g$ `size(`$\boldsymbol{u}_g$`) = [`N_u`,` N_g`]`

problem.options

The options struct provides options for both `optimTraj` and `fmincon`, which is called by `optimTraj` to solve the underlying nonlinear program. All fields, including **options** itself may be omitted. If **problem.options** is a struct array, then `optimTraj` will run a sequence of trajectory optimizations, one for each element of the options struct array. The solution to each sub-problem is used to initialize the following. This is used for manual mesh refinement, and is illustrated in several of the example problems in the `demo/` directory.

The `trapezoid`, `hermiteSimpson`, and `rungeKutta` methods work without any external dependencies. The `chebyshev` method is written entirely in `optimTraj`, but relies on ChebFun[6] for computing some of the low-level implementation details for the orthogonal polynomials. ChebFun is easy to download and install from `https://github.com/chebfun/chebfun`. The final method, `gpops` is actually just a wrapper for the professional software GPOPS-II[8], for which a license can be obtained from `http://www.gpops2.com/`.

Each method has a single field devoted to it in the options struct. The options in each of these fields are used exclusively by the method for which they are named. This allows for method-specific options, like the number of sub-steps in the multiple shooting method. The field **problem.options.gpops** is passed to GPOPS-II as the **setup** struct, allowing the user to specify low-level options in GPOPS-II.

- **problem.options.nlpOpt** = a struct of options that are passed directly to fmincon
- **problem.options.verbose** = how much detail to provide?
  $0 \rightarrow$ no printing, $1 \rightarrow$ default, $2 \rightarrow$ extra warnings, $3 \rightarrow$ debug.
- **problem.options.defaultAccuracy** = used to set the default settings for all methods
  possible values: `'low'` `'medium'` `'high'`
- **problem.options.method** = a string, specifying which method to use.
  - **'trapezoid'** $\rightarrow$ trapezoidal direct collocation
  - **'hermiteSimpson'** $\rightarrow$ Hermite–Simpson direct collocation
  - **'chebyshev'** $\rightarrow$ Chebyshev–Lobatto orthogonal collocation
  - **'rungeKutta'** $\rightarrow$ 4^{th}-order Runge–Kutta Multiple Shooting
  - **'gpops'** $\rightarrow$ wrapper for calling GPOPS-II
- **problem.options.trapezoid.nGrid** = number of grid-points in `trapezoid`
- **problem.options.hermiteSimpson.nSegment** = number of segments in `hermiteSimpson`
- **problem.options.chebyshev.nColPts** = number of collocation points in `chebyshev`
- **problem.options.rungeKutta.nSegment** = number of segments in `rungeKutta`
- **problem.options.rungeKutta.nSubStep** = number of sub-steps per segment in `rungeKutta`
- **problem.options.gpops** = low-level options for `gpops` (passed like `setup`)

3 Details for the Output of OptimTraj

The output of **optimTraj()** is a single struct, which we will call the **soln** struct. The **soln** struct has four fields. The **soln.grid** struct contains value of the trajectory along the exact grid-points that were used by the non-linear program. The **soln.interp** struct contains a set of function handles that are used for method-consistent interpolation the solution trajectory. The **soln.info** struct contains a variety of details about the solution to the trajectory optimizaiton problem. Finally, the **soln.problem** struct is a copy of the input to `optimTraj`, including all default options that were used.

soln.grid

The grid struct contains trajectory evaluated at the grid-points used by the transcriptions algorithm. The spacing and number of the grid-points is dependent on the transcription method and options that were passed to it.

- **soln.grid.time =** t `size(`t`) = [1, `` `$N_t$`]`
- **soln.grid.state =** $\boldsymbol{x}$ `size(`$\boldsymbol{x}$`) = [`N_x`, `N_t`]`
- **soln.grid.control =** $\boldsymbol{u}$ `size(`$\boldsymbol{u}$`) = [`N_u`, `N_t`]`

soln.interp

The interp struct contains two function handles, one for the state and another for control. Each function handle takes a single argument (time), and returns either the state or control vector at each point in the time vector. The interpolation method is consistent with whatever transcription method was used. For example, if you used the `trapezoid` method, then **soln.interp.state** will use linear interpolation between the control grid points. This method-consistent interpolation is particularly important for high-order methods, such as `chebyshev`. Note: in both cases below, the function name `'specialInterpolation'` is just a placeholder. It will be different for each method.

- **soln.interp.state :** $\rightarrow$ $\boldsymbol{x}$ **= specialInterpolation(**t**)**
  where $\boldsymbol{x} = \boldsymbol{x}(t)$ is the value of the state at each point in t and `size(`$\boldsymbol{x}$`)==[`N_x`, length(`t`)]`
- **soln.interp.control :** $\rightarrow$ $\boldsymbol{u}$ **= specialInterpolation(**t**)**
  where $\boldsymbol{u} = \boldsymbol{u}(t)$ is the value of the state at each point in t and `size(`$\boldsymbol{u}$`)==[`N_u`, length(`t`)]`

- **soln.interp.collCst** : $\rightarrow$ ε = **specialInterpolation**(t)
  where $\boldsymbol{\varepsilon} = \boldsymbol{\varepsilon}(t)$ is the value of the collocation constraint. This will be zero at each collocation point and non-zero elsewhere. It is used as an error metric. Currently only available for the `trapezoid` and `hermiteSimpson` methods. `size(`$\boldsymbol{\varepsilon}$`)==size(`$\boldsymbol{x}$`)`

soln.info

The info struct provides details regarding the performance of both `optimTraj` and `fmincon`, as well as general information about the solution. All solvers return the fields described here, with two exceptions. First, many of the methods output additional information, such as error metrics for the direct collocation methods. Type `>>help [method name]` for details. Second, the `gpops` method does not use `fmincon`. Instead of returning the information from `fmincon`, the entire output of GPOPS-II is placed in the field **soln.info.gpops**.

If the **problem.options** is a struct array, then **soln** will also be a struct array, with one solution for each of the sub-problems specified by the options struct.

- **soln.info.nlpTime** = total time spent in the nonlinear programming solver.
- **soln.info.exitFlag** = exit flag returned by the nonlinear programmign solver.
- **soln.info.objVal** = value of the objective function
- **soln.info.*** = all fields from the `fmincon problem struct`
- **soln.info.error** = integral of absolute error in collocation constraint along each segment of the trajectory. Currently only available for the `trapezoid` and `hermiteSimpson` methods.
- **soln.info.maxError** = `max(max(`**soln.info.error**`))`

soln.problem

The problem struct is similar to the problem struct that was passed in, but `optimTraj` fills in all omitted fields with the defaults that were actually used by the program.

4 Technical Details

4.1 Problem Statement

Minimize the objective function:

$$\min_{t_0, t_F, \boldsymbol{x}(t), \boldsymbol{u}(t)} J_B\big(t_0, t_F, \boldsymbol{x}(t_0), \boldsymbol{x}(t_F)\big) + \int_{t_0}^{t_F} J_P\big(\tau, \boldsymbol{x}(\tau), \boldsymbol{u}(\tau)\big)\, d\tau$$

Subject to the constraints:

$\dot{\boldsymbol{x}}(t) = \boldsymbol{f}\big(t, \boldsymbol{x}(t), \boldsymbol{u}(t)\big)$	**system dynamics**
$\boldsymbol{C}_P\big(t, \boldsymbol{x}(t), \boldsymbol{u}(t)\big) \le \boldsymbol{0}$	**path constraints**
$\boldsymbol{C}_B\big(t_0, t_F, \boldsymbol{x}(t_0), \boldsymbol{x}(t_F)\big) \le \boldsymbol{0}$	**boundary constraints**
$\boldsymbol{x}^- \le \boldsymbol{x}(t) \le \boldsymbol{x}^+$	**constant bounds on state**
$\boldsymbol{u}^- \le \boldsymbol{u}(t) \le \boldsymbol{u}^+$	**constant bounds on control**
$t^- \le t_0 < t_F \le t^+$	**bounds on initial and final time**
$\boldsymbol{x}_0^- \le \boldsymbol{x}(t_0) \le \boldsymbol{x}_0^+$	**bound on initial state**
$\boldsymbol{x}_F^- \le \boldsymbol{x}(t_F) \le \boldsymbol{x}_F^+$	**bound on final state**

Assuming that the user-defined objective, dynamics, and constraint functions (J_P, J_B, $\boldsymbol{f}$, $\boldsymbol{C}_P$, $\boldsymbol{C}_B$) are smooth. The user must provide an initial guess for the decision variables t_0, t_F, $\boldsymbol{x}(t)$, and $\boldsymbol{u}(t)$, and ensure that a feasible solution exists.

4.2 Trapezoidal Direct Collocation: `trapezoid`

Trapezoidal direct collocation works by making the assumption that the optimal trajectory can be approximated using a low-order spline. In this case, the dynamics, objective function, and control trajectories are approximated using a linear spline, and the state trajectory is the quadratic spline, obtained by integration of the (linear) dynamics spline. Integration of a linear spline is computed using the trapezoid rule, hence the name. The implementation of trapezoidal direct collocation in OptimTraj is almost entirely based on the method as it is described in [3].

4.3 Hermite-Simpson Direct Collocation: `hermiteSimpson`

Hermite-Simpson direct collocation works by making the assumption that the optimal trajectory can be approximated using a medium-order spline. In this case, the dynamics, objective function, and control trajectories are approximated using a quadratic spline, and the state trajectory is a cubic hermite spline, obtained by integration of the (quadratic) dynamics spline. Integration of a quadratic spline is computed using Simpson's rule. There are two versions of this method: separated and compressed. In the separated form, the state at the mid-point of each trajectory segment is included as a decision variable, and the Hermite-interpolation is enforced with a constraint. In the compressed form, the state at the mid-point is computed from the definition of the Hermite-interpolant during optimization. OptimTraj implements the separated Hermite-Simpson method, almost entirely based on the method as it is described in [3].

4.4 Chebyshev–Lobatto Orthogonal Collocation: `chebyshev`

Chebyshev–Lobatto orthogonal collocation works by representing the entire trajectory using a high-order Chebyshev orthogonal polynomial. The implementation here might also be called pseudospectral or global collocation, since the entire trajectory is represented using a single segment, rather than several segments. Orthogonal collocation methods can be divided into three categories: Gauss, Radau, and Lobatto. In a Gauss method, neither end-point of a segment is a collocation point, in a Radau method, a single end-point of the segment is a collocation point, and in a Lobatto method, both end-points are collocation points [7].

The implementation here uses the ChebFun toolbox [6] for computing the Chebyshev–Lobatto collocation points, and also for interpolation of the solution. More details regarding orthogonal polynomials and calculations with them can be found in [1] and [11]. The details of the collocation method itself are largely drawn from [12].

4.5 Runge–Kutta 4^{th}-order Multiple Shooting: `rungeKutta`

A multiple shooting method works by breaking the trajectory into segments, and approximating each segment using an explicit simulation. In this case, we use a 4^{th}-order Runge–Kutta method for the simulation. A defect constraint is used to ensure that the end of each trajectory segment correctly lines up with the next. The interpolation of the solution trajectory, for both control and state, is approximated using a cubic Hermite spline. The implementation here, except for interpolation, is as described in [3].

4.6 GPOPS-II: `gpops`

OptimTraj includes a wrapper to the software GPOPS-II [8], a professionally developed trajectory optimization library for Matlab. It implements a nicely optimized version of Radau orthogonal collocation with adaptive meshing [5]. There are a few choice in GPOPS-II for both collocation metho and the adaptive meshing. It also supports analytic gradients using automatic differentiation. It is included in OptimTraj for two reasons: 1) GPOPS-II can be used to benchmark and verify the methods in OptimTraj, and 2) GPOPS-II provides a collection of methods that are not otherwise available in OptimTraj.

4.7 Resources for Learning Trajectory Optimization

The single best resource for learning about trajectory optimization is the textbook by John T. Betts: "Practical Methods for Optimal Control and Estimation Using Nonlinear Programming" [3]. The textbook by

Bryson and Ho: "Applied Optimal Control" [4]. Both of these books are also excellent for learning about nonlinear programming and optimization.

There are two good review papers about trajectory optimization. The paper by Betts [2] is more focused on direct collocation and shooting methods, while the paper by Rao [9] is more focused on orthogonal collocation methods.

Understanding orthogonal collocation is rather challenging, and I found that it was first necessary to get a solid understanding of orthogonal polynomials and function approximation. I started by reading "Approximation Theory and Approximation Practice" [11] by Trefethen, and then his paper on barycentric interpolation [1]. Some intuition can also be obtained by reading the source code of ChebFun [6]:
`http://www.chebfun.org/`

Russ Tedrake at MIT has also provided some excellent resources for learning about trajectory optimization, and robotics in general. The first is his online course on underactuated robotics. You can download the pdf of the course notes [10], or access the full course and video lectures at:
http://ocw.mit.edu/courses/
Second, his group at MIT are producing a planning, control, and analysis toolbox called Drake, which includes trajectory optimization. It is open source, so it is another good place to read the source code and figure out how it works:
`https://github.com/RobotLocomotion/drake`

I wrote a tutorial paper for learning trajectory optimization. It goes into depth covering all technical content for the two direct collocation methods in this library, as well as a variety of related topics: `https://epubs.siam.org/doi/pdf/10.1137/16M1062569`

There is also a tutorial page on my website. It contains a high-level summary and links to a variety of other resources including a video, slides, documents, and code samples:
`http://www.matthewpeterkelly.com/tutorials/trajectoryOptimization/index.html`

References

[1] J.-P. Berrut and L. N. Trefethen. Barycentric Lagrange Interpolation. *SIAM Review*, 46(3):501–517, 2004.

[2] J. T. Betts. A Survey of Numerical Methods for Trajectory Optimization. *Journal of Guidance, Control, and Dynamics*, pages 1–56, 1998.

[3] J. T. Betts. *Practical Methods for Optimal Control and Estimation Using Nonlinear Programming*. Siam, Philadelphia, PA, 2010.

[4] A. E. Bryson and Y.-C. Ho. *Applied Optimal Control*. Taylor & Francis, 1975.

[5] C. L. Darby, W. W. Hagar, and A. V. Rao. An hp-adaptive pseudospectral method for solving optimal control problems. *Optimal Control Applications and Methods*, 32:476–502, 2011.

[6] T. A. Driscoll, N. Hale, and L. N. Trefethen. *Chebfun Guide*. Pafnuty Publications, Oxford, 1 edition, 2014.

[7] D. Garg, M. Patterson, W. W. Hager, A. V. Rao, D. a. Benson, and G. T. Huntington. A unified framework for the numerical solution of optimal control problems using pseudospectral methods. *Automatica*, 46(11):1843–1851, 2010.

[8] M. A. Patterson and A. V. Rao. GPOPS II : A MATLAB Software for Solving Multiple-Phase Optimal Control Problems Using hp Adaptive Gaussian Quadrature Collocation Methods and Spa and rse Nonlinear Programming. 39(3):1–41, 2013.

[9] A. Rao. A survey of numerical methods for optimal control. *Advances in the Astronautical Sciences*, 135:497–528, 2009.

[10] R. Tedrake. Underactuated Robotics: Learning, Planning, and Control for Efficient and Agile Machines. Technical report, 2009.

[11] L. N. Trefethen. *Approximation Theory and Approximation Practice*. SIAM, 2013.

[12] J. Vlassenbroeck and R. V. Dooren. A Chebyshev technique for solving nonlinear optimal control problems. *Automatic Control, IEEE ...*, 33(4), 1988.