MPOPT Documentation

Release 1.0.2

https://mpopt.readthedocs.io/

Jul 04, 2024

CONTENTS

Simple to use, optimal control problem solver library in Python: GitHub.

MPOPT is an open-source, extensible, customizable and easy to use Python package that includes a collection of modules to solve multi-stage non-linear optimal control problems(OCP) in the standard Bolza form using pseudo-spectral collocation methods.

CHAPTER

ONE

INTRODUCTION

MPOPT package implements libraries to solve multi-stage or single-stage optimal control problems of the Bolza form given below,

$$\begin{aligned} \min_{x,u,t_0,t_f,s} \quad & J = M(x_0, t_0, x_f, t_f, s) + \int_0^{t_f} L(x, u, t, s)dt \\ \text{subject to} \quad & \dot{x} = f(x, u, t, s) \\ & g(x, u, t, s) \leq 0 \\ & h(x_0, t_0, x_f, t_f, s) = 0 \end{aligned}$$

The package uses collocation methods to construct a Nonlinear programming problem (NLP) representation of OCP. The resulting NLP is then solved by algorithmic differentiation based CasADi nlpsolver (NLP solver supports multiple solver plugins including IPOPT, SNOPT, sqpmethod, scpgen).

Main features of the package are :

- Customizable collocation approximation, compatible with Legendre-Gauss-Radau (LGR), Legendre-Gauss-Lobatto (LGL), Chebyshev-Gauss-Lobatto (CGL) roots.
- Intuitive definition of single/multi-phase OCP.
- Supports Differential-Algebraic Equations (DAEs).
- Customized adaptive grid refinement schemes (Extendable)
- Gaussian quadrature and differentiation matrices are evaluated using algorithmic differentiation, thus, supporting arbitrarily high number of collocation points limited only by the computational resources.
- Intuitive post-processing module to retrieve and visualize the solution
- Good test coverage of the overall package
- Active development

Access quick introduction presentation PDF.

Quick start demo *Jupyter-notebook*.

Similar solver packages: GPOPS II, ICLOCS2, PSOPT, DIRCOL, DIDO, SOS, ACADO, OPTY

Next steps: **:doc:`./installation`_**, **:doc:`./getting_started>`_**

CHAPTER

TWO

INSTALLATION

- Install from PyPI using the following terminal command. Refer getting started to solve OCPs.

```
pip install mpopt
```

- (OR) Build directly from source (Terminal). Finally, `make run` to solve the moon-lander example described below.

```
git clone https://github.com/mpopt/mpopt.git --branch master
cd mpopt
make build
make run
source env/bin/activate
```

For discussions related to installation, refer issues

Next steps: **:doc:`./getting_started>`_**, **:doc:`./examples`_**

CHAPTER

THREE

GETTING STARTED

3.1 Solve moon-lander OCP in under 10 lines

OCP :

Find optimal path to reach the moon surface with minimum fuel (u). From :Height(0m), Velocity (0m/s) from Height (10m) and velocity(-2m/s) to Height (x_0), Velocity (x_1) and Throttle (u).

$$
\begin{aligned}
\min_{x,u} \quad & J = 0 + \int_{t_0}^{t_f} u \, dt \\
\text{subject to} \quad & \dot{x_0} = x_1; \dot{x_1} = u - 1.5 \\
& x_0(t_f) = 0; \; x_1(t_f) = 0 \\
& x_0(t_0) = 10; \; x_1(t_0) = -2 \\
& x_0 \geq 0; 0 \leq u \leq 3 \\
& t_0 = 0.0; t_f = \text{free variable}
\end{aligned}
$$

```
# Moon lander OCP direct collocation/multi-segment collocation

# from context import mpopt # (Uncomment if running from source)
from mpopt import mp

# Define OCP
ocp = mp.OCP(n_states=2, n_controls=1)
ocp.dynamics[0] = lambda x, u, t: [x[1], u[0] - 1.5]
ocp.running_costs[0] = lambda x, u, t: u[0]
ocp.terminal_constraints[0] = lambda xf, tf, x0, t0: [xf[0], xf[1]]
ocp.x00[0] = [10.0, -2.0]
ocp.lbx[0][0] = 0.0
ocp.lbu[0], ocp.ubu[0] = 0, 3

# Create optimizer(mpo), solve and post process(post) the solution
mpo, post = mp.solve(ocp, n_segments=20, poly_orders=3, scheme="LGR", plot=True)
x, u, t, _ = post.get_data()
mp.plt.show()
```

- Experiment with different collocation schemes by changing “LGR” to “CGL” or “LGL” in the above script.
- Update the grid to recompute solution (Ex. n_segments=3, poly_orders=[3, 30, 3]).
- For a detailed demo of the mpopt features, refer the notebook getting_started.ipynb

For issues related to getting started examples, refer issues

Next steps: **:doc:`./examples`_**, **:doc:`./notebooks`_**

3.1.1 Overview of MPOPT

(c) 2023 Devakumar Thammisetty

MPOPT is an open-source Multi-phase Optimal Control Problem (OCP) solver based on pseudo-spectral collocation with customized adaptive grid refinement techniques.

https://mpopt.readthedocs.io/

Download this notebook: getting_started.ipynb

Tip: Ctrl + Tab (For the documentation of commands/syntax)

Install mpopt from pypi using the following. Disable after first usage

[2]:
```
#pip install mpopt
```

Import mpopt (Contains main solver modules)

[3]:
```
# from context import mpopt
from mpopt import mp
```

Defining OCP

We first create an OCP object and then polulate the object with dynamics, path_constraints, terminal_constraints and objective (running_costs, terminal_costs) * create an OCP object (Lets define moon-lander OCP)

[4]:
```
ocp = mp.OCP(n_states=2, n_controls=1, n_phases=1)
```

- Add dynamics to the OCP. The solver being multi-phase, we add phase index to dynamics definition.

Moon lander (2D)

$ = v; ‘:nbsphinx-math:dot{v}` = u - g$

- Using the standard Bolza form of OCP, $\dot{x} = f(x, u, t)$ $\implies$

[5]:
```
ocp.dynamics[0] = lambda x, u, t: [x[1], u[0] - 1.5]
```

- Add objective function, Lets get fuel optimal solution (u : Thrust level), Using standard Bolza form

$ J = 0 + ∫ _{t_0}ˆ{t_f}udt

[6]:
```
ocp.running_costs[0] = lambda x, u, t: u[0]
```

- Add terminal constraints. The target is to reach (postion : 0, velocity : 0), Using standard terminal constraints function

$h(x_f, t_f, x_0, t_0) = 0 $

[7]:
```
ocp.terminal_constraints[0] = lambda xf, tf, x0, t0: [xf[0], xf[1]]
```

- Define starting position and initial velocity

$x[phase]$

[8]:
```
ocp.x00[0] = [10.0, -2.0]
```

- Thrust is constrained between 0 and 3. $ $\implies$ $0 \leq$ ‘$u : nbsphinx - math : leq3$‘$

[9]:
```
ocp.lbu[0], ocp.ubu[0] = 0, 3
```

- Validate if the ocp is well defined

[10]:
```
ocp.validate()
```

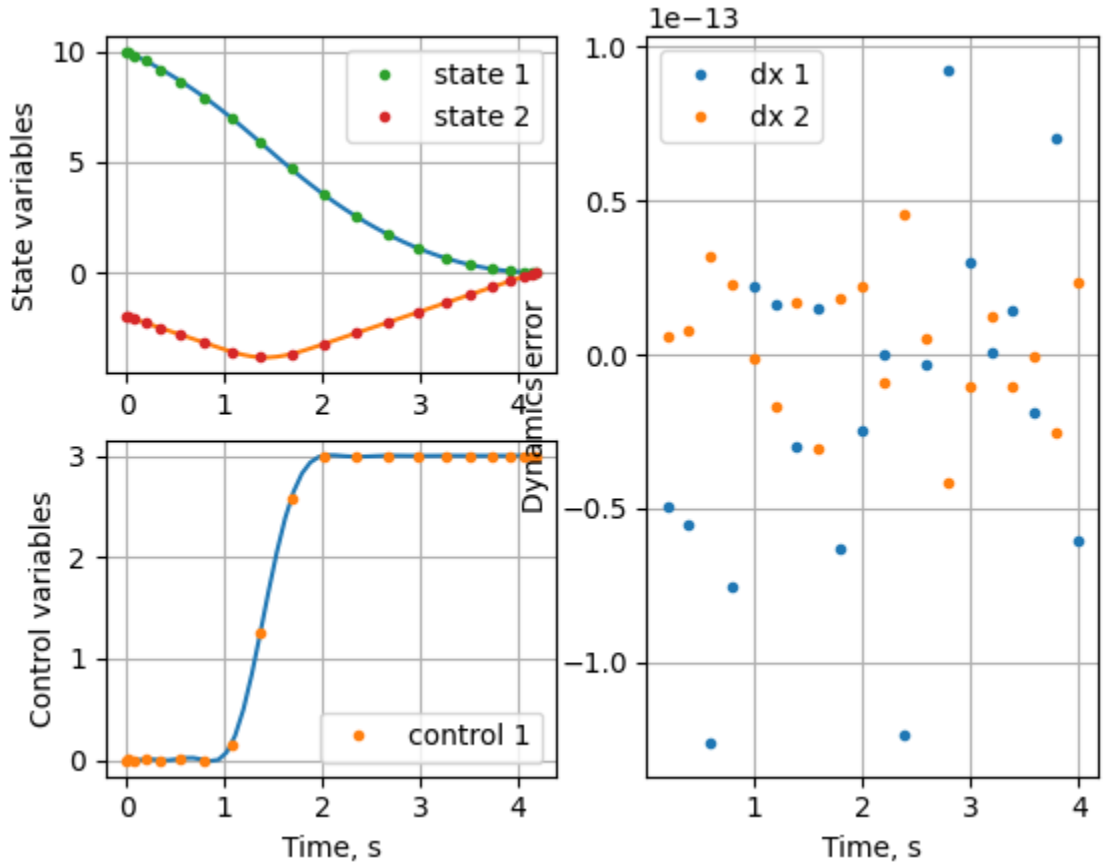
Solve and plot the results in one line

Lets solve the OCP using following pseudo-spectral approximation * Collocation using Legendre-Gauss-Radau roots * Global collocation (single segment) * Let’s plot the position and velocity evolution with time starting from 0.

The OCP is a free final time formulation,

[11]:
```
mpo, post = mp.solve(ocp, n_segments=1, poly_orders=20, scheme="LGR", plot=True)
```

```
 *********** MPOPT Summary **********
CasADi - 2024-03-10 12:06:39 WARNING("The options 't0', 'tf', 'grid' and 'output_t0'␣
↪have been deprecated.
The same functionality is provided by providing additional input arguments to the
↪'integrator' function, in particular:
 * Call integrator(..., t0, tf, options) for a single output time, or
 * Call integrator(..., t0, grid, options) for multiple grid points.
The legacy 'output_t0' option can be emulated by including or excluding 't0' in 'grid'.
Backwards compatibility is provided in this release only.") [.../casadi/core/integrator.
↪cpp:521]
 Optimal cost (J):  8.27612

 Solved in 343.132 ms
         OCP transcription time  : 322.33 ms
         NLP solution time       : 20.802 ms

 Post processed in 45.493 ms
         Solution retrieval            : 0.126 ms
         Residual in dynamics           : 6.594 ms
         Process solution and plot      : 38.773 ms
```

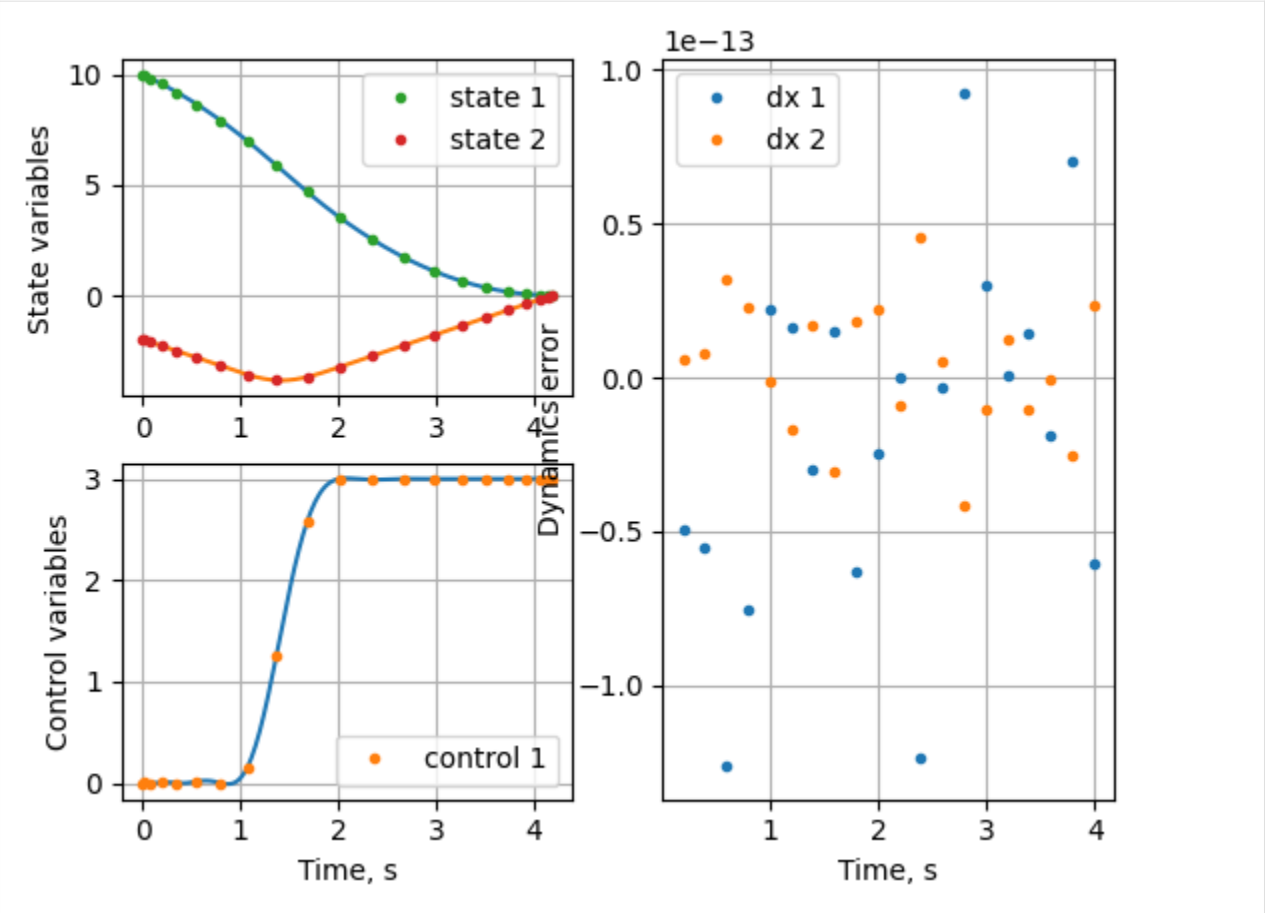


The tics in the solution represent the values at collocation nodes. The states and controls are plotted by interpolating the lagrange polynomials at only few points in a single segment (by default). Let's interpolate with better resolution for plotting.

[12]:
```
post._INTERPOLATION_NODES_PER_SEG = 200
```

Lets plot the same result again, this time the control profile is more refined.

[13]:
```
fig, axs = post.plot_phases()
```



Lets retrive the solution to see the terminal time

[14]:
```
x, u, t, _ = post.get_data()
```

Last element of t and x gives the terminal values. Exact terminal time from the analytical solution is 4.1641s.

[15]:
```
print(f"Terminal time, state : {t[-1]}, {x[-1]}")
```

```
Terminal time, state : [4.18408515], [-6.22886168e-26  2.27628981e-25]
```

The Fuel optimal solution to the moon-lander OCP is known to have bang-bang thrust profile. The selected OCP has one discontinuity. To better appoximate the solution, we can divide the domain into multiple segments.

Lets solve the problem now using better collocation approximation, based on the bang-nag nature of the thrust profile. * Number of segments : 20 * Polynomials of degree : 3

[16]:
```
mpo, post = mp.solve(ocp, n_segments=20, poly_orders=3, scheme="LGR", plot=True)
```

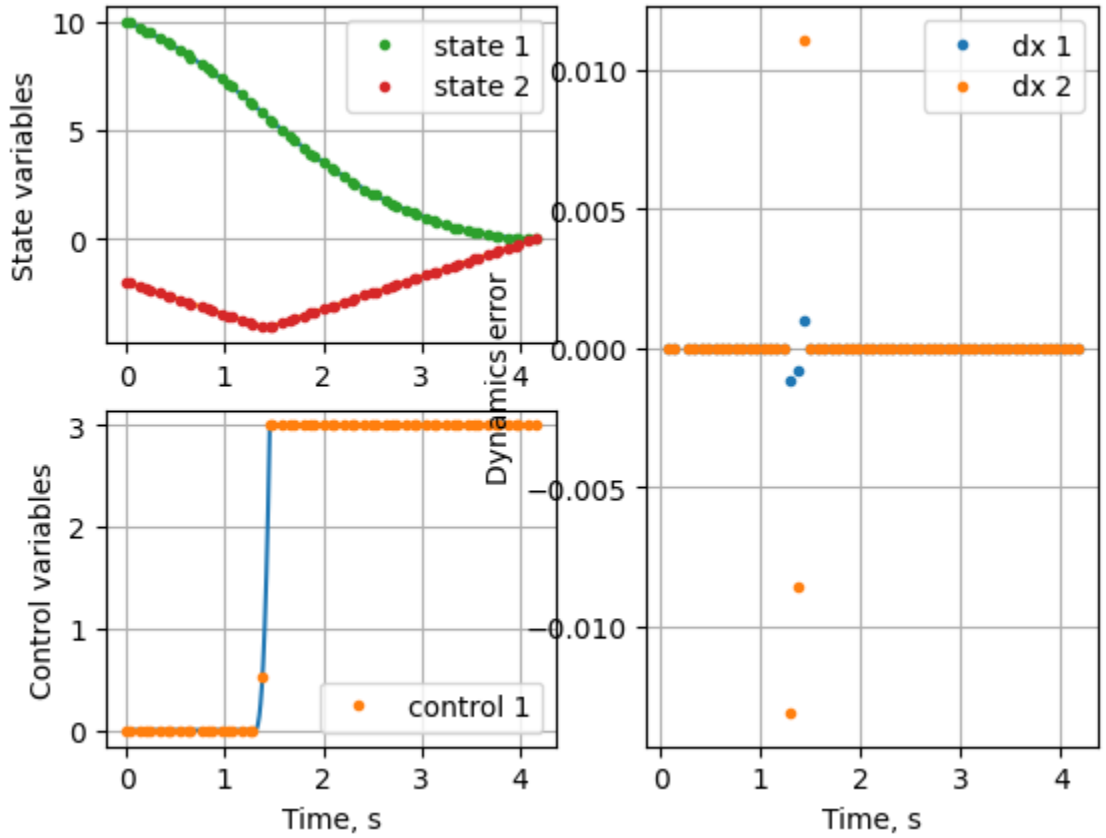
```
********** MPOPT Summary **********

Optimal cost (J):  8.24677

Solved in 45.279 ms
        OCP transcription time  : 21.198 ms
        NLP solution time       : 24.081 ms
```

(continues on next page)

(continued from previous page)

```
Post processed in 60.651 ms
        Solution retrieval             : 0.312 ms
        Residual in dynamics           : 10.062 ms
        Process solution and plot      : 50.278 ms
```

```
<__array_function__ internals>:180: VisibleDeprecationWarning: Creating an ndarray from␣
↪ragged nested sequences (which is a list-or-tuple of lists-or-tuples-or ndarrays with␣
↪different lengths or shapes) is deprecated. If you meant to do this, you must specify
↪'dtype=object' when creating the ndarray.
```



The solution is now more refined. Lets look at the terminal time.

[17]:
```
print(f"Terminal time, s : {post.get_data()[-2][-1][0]} vs 4.1641")
```

```
Terminal time, s : 4.164513728300145 vs 4.1641
```

Let's find the best solution by increasing the collocation nodes

[18]:
```
mpo, post = mp.solve(ocp, n_segments=200, poly_orders=3, scheme="LGR", plot=False)
print(f"\nTerminal time, s : {post.get_data()[-2][-1][0]} vs 4.1641")
```

```
********** MPOPT Summary **********
Optimal cost (J):  8.24623
```

(continues on next page)

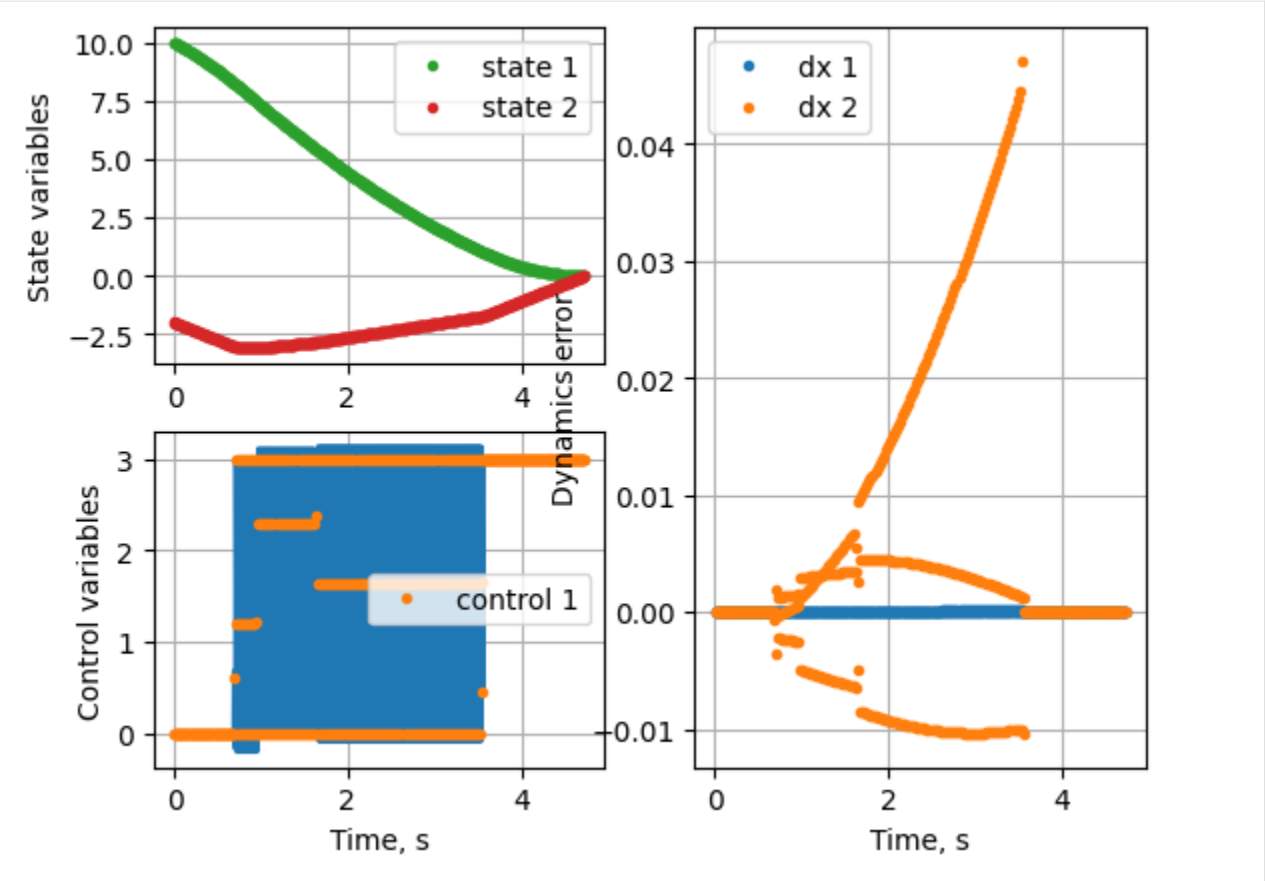
(continued from previous page)

```
 Solved in 431.338 ms
         OCP transcription time  : 242.666 ms
         NLP solution time       : 188.672 ms

 Post processed in 181.282 ms
         Solution retrieval            : 0.702 ms
         Residual in dynamics           : 180.57 ms
         Process solution and plot      : 0.009 ms

Terminal time, s : 4.164150730471649 vs 4.1641
```

Let's check how the Legendre-Gauss-Lobatto roots solve the OCP

[19]:
```
mpo, post = mp.solve(ocp, n_segments=200, poly_orders=3, scheme="LGL", plot=True)
print(f"\nTerminal time using Lobatto roots , s : {post.get_data()[-2][-1][0]} vs 4.1641
↪")
```

```
 *********** MPOPT Summary **********
 Optimal cost (J):  6.87196

 Solved in 496.378 ms
         OCP transcription time  : 230.422 ms
         NLP solution time       : 265.956 ms

 Post processed in 362.558 ms
         Solution retrieval            : 0.871 ms
         Residual in dynamics           : 154.644 ms
         Process solution and plot      : 207.043 ms

Terminal time using Lobatto roots , s : 4.723467249086362 vs 4.1641
```



Pretty bad result, huh!. The thrust profile is not smooth. Besides, so many number of segments. Lobatto roots lead to better convergence when the degree of the polynomial is higher.

Adding additional constraints to the OCP

Let's constrain the derivative of control between ± 1.

$-1 \leq \dot{u} \leq 1$

- Enable the differential constraint
- Choose bounds of the control slope

[20]:
```
ocp.diff_u[0] = 1
ocp.lbdu[0], ocp.ubdu[0] = -1, 1
```

Let's solve again by adding the above constraint.

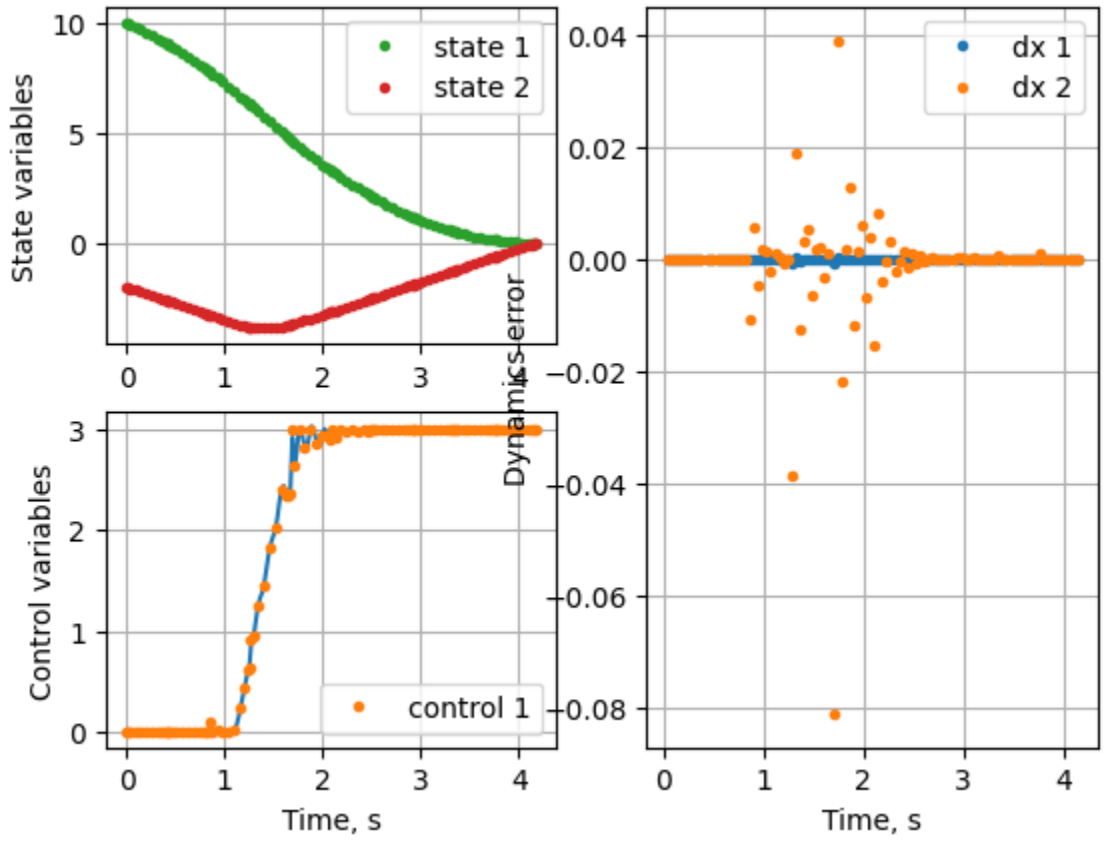
[21]:
```
mpo, post = mp.solve(ocp, n_segments=10, poly_orders=10, scheme="LGL", plot=False)
post._INTERPOLATION_NODES_PER_SEG = 20
post.plot_phases()
print(f"\nTerminal time using Lobatto roots , s : {post.get_data()[-2][-1][0]} vs 4.1641
↪")
```

```
********** MPOPT Summary **********
```

(continues on next page)

(continued from previous page)

```
Optimal cost (J):  8.1831

Solved in 107.721 ms
        OCP transcription time  : 66.337 ms
        NLP solution time       : 41.384 ms

Post processed in 20.4 ms
        Solution retrieval            : 0.17 ms
        Residual in dynamics           : 20.223 ms
        Process solution and plot      : 0.007 ms

Terminal time using Lobatto roots , s : 4.181973552017326 vs 4.1641
```



Change the number of semgents and approximating to see how the solution behaves.

Disable the slope constraint for further calculations. Also, Radau roots are found to be approximating the given solution best!

[22]:
```
ocp.diff_u[0] = 0
```

Let's see what happens if we increase the polynomial degree to 25.

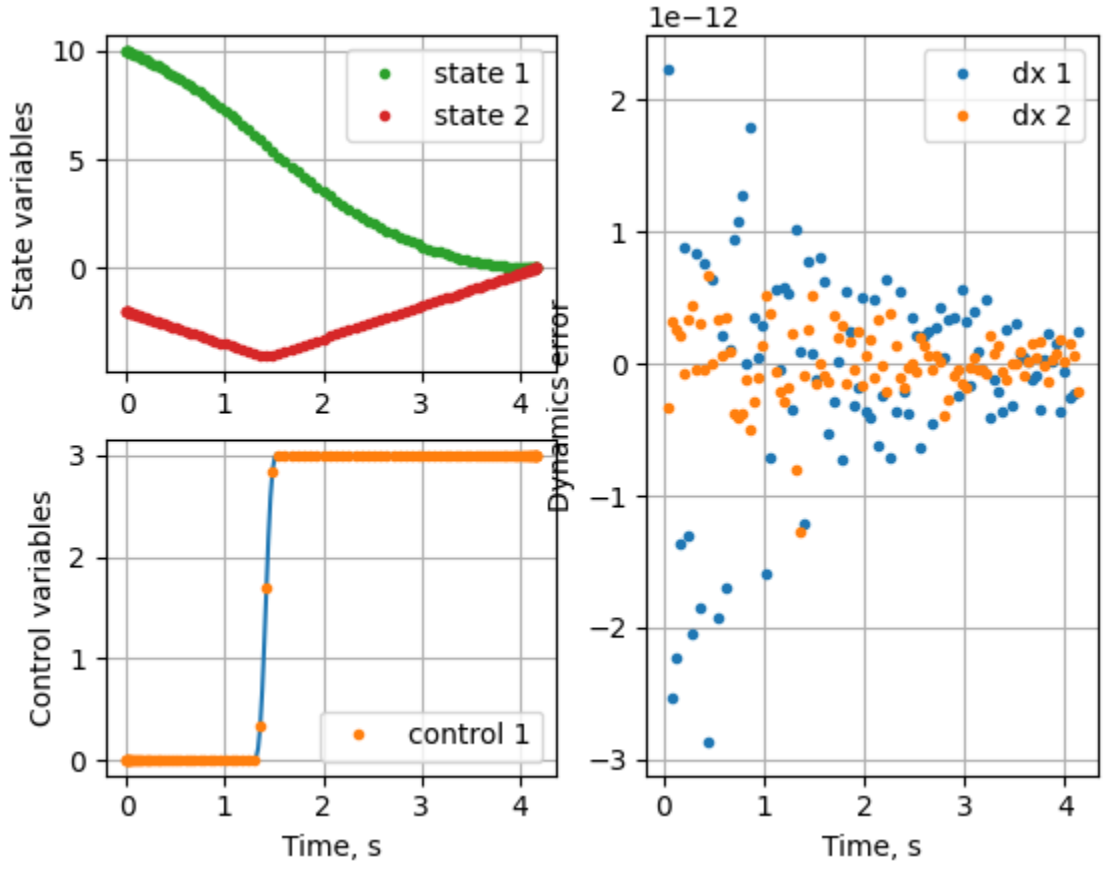
[23]:
```
mpo, post = mp.solve(ocp, n_segments=1, poly_orders=100, scheme="LGR", plot=False)
post._INTERPOLATION_NODES_PER_SEG = 200
```

(continues on next page)

(continued from previous page)

```
post.plot_phases()
print(f"\nTerminal time using Radau roots , s : {post.get_data()[-2][-1][0]} vs 4.1641")
```

```
********** MPOPT Summary **********
Optimal cost (J):  8.24747

Solved in 1750.149 ms
        OCP transcription time  : 1589.343 ms
        NLP solution time       : 160.807 ms

Post processed in 123.729 ms
        Solution retrieval             : 0.197 ms
        Residual in dynamics           : 123.524 ms
        Process solution and plot      : 0.008 ms

Terminal time using Radau roots , s : 4.164977321911994 vs 4.1641
```



The derivative and interpolation matrices are evaluated using CasADi algorithmic differentiation. Hence, high values of polynomial order such as 100 are feasible.

On the contrary, if the derivative operators are contructed using numpy's polynomial module, as the number of nodes increases, numerical noise builds up in the D matrix approximation leading to unstable solution.

Adaptive grid refinement schemes

A good guess of the number of segments and polynomial order are necessary to get exact solution. However, adaptive grid refinement techniques solve this problem by relocating the collocation points.

There are three inbuit refinement schemes in the package. First two are iterative. Refer theis for the exact details of implementaion.

The schemes are named as follows * Adaptive_h_refinement (Iterative) * Scheme-1: Based on control slope * Scheme-2: Based on residual in dynamics * Scheme-3: Direct optimization of segment widths (h)

Let's see how grid refinement changes the solution to moon-lander OCP

Let's create the optimizer object

[24]:
```
mpo = mp.mpopt_h_adaptive(ocp, 10, 4) # Use ctrl + tab for help, inside ()
```

Adaptive scheme-I

Let's solve the OCP using scheme 1. Let's limit the maximum iterations to 1. (Same as non-adaptive solution)

- control_slope

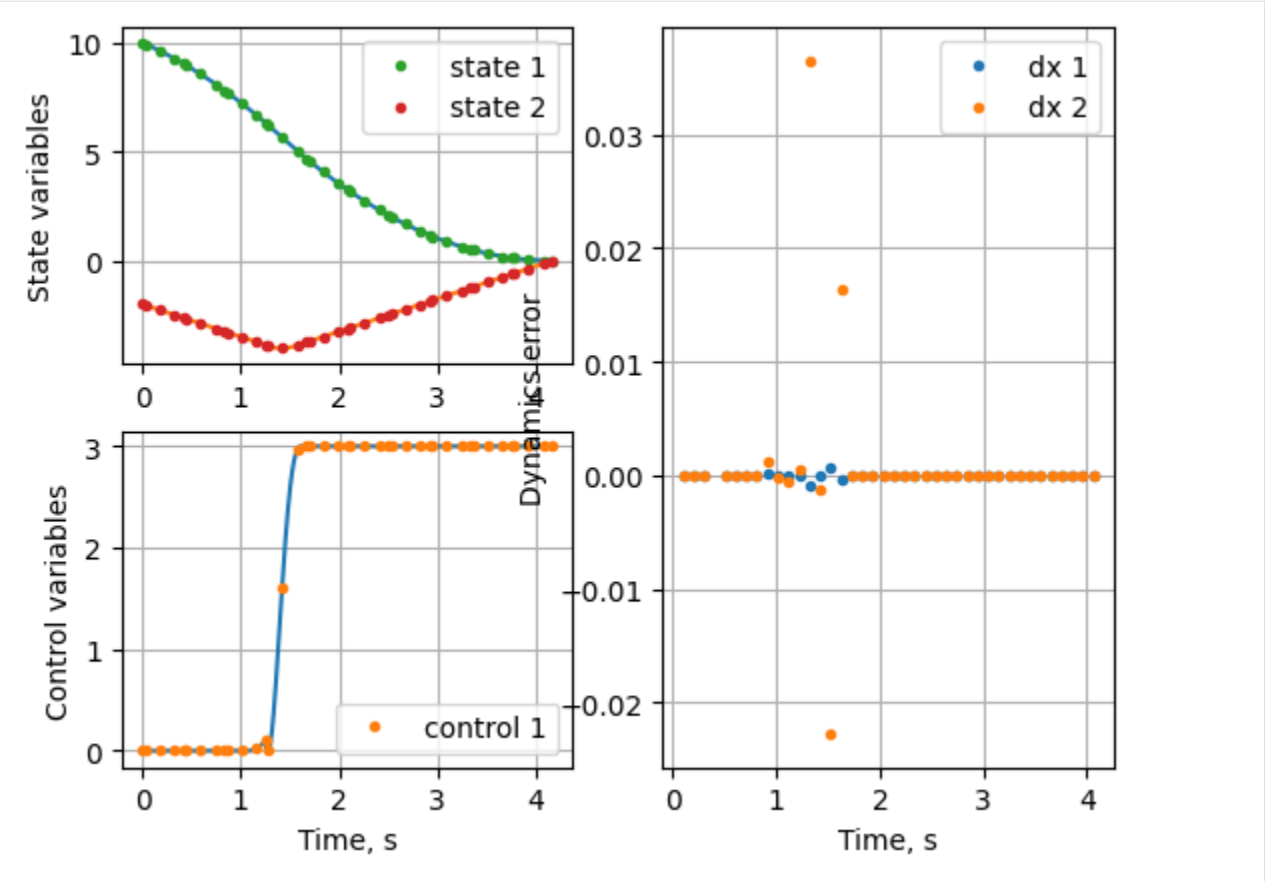
[25]:
```
solution = mpo.solve(max_iter=1, mpopt_options={'method':'control_slope'})
```

```
 *********** MPOPT H-Adaptive Summary **********

Stopping the iterations: Iteration limit exceeded
H-Adaptive Iter., max_residual : 1, 0.03650101311435586
 Optimal cost (J):  8.24933

 Solved in 48.449 ms
```

Let's process the solution and plot states and controls. This solution is same as the non-adaptive solution, since the max iterations are set to 1.

[26]:
```
post = mpo.process_results(solution, plot=True)
```

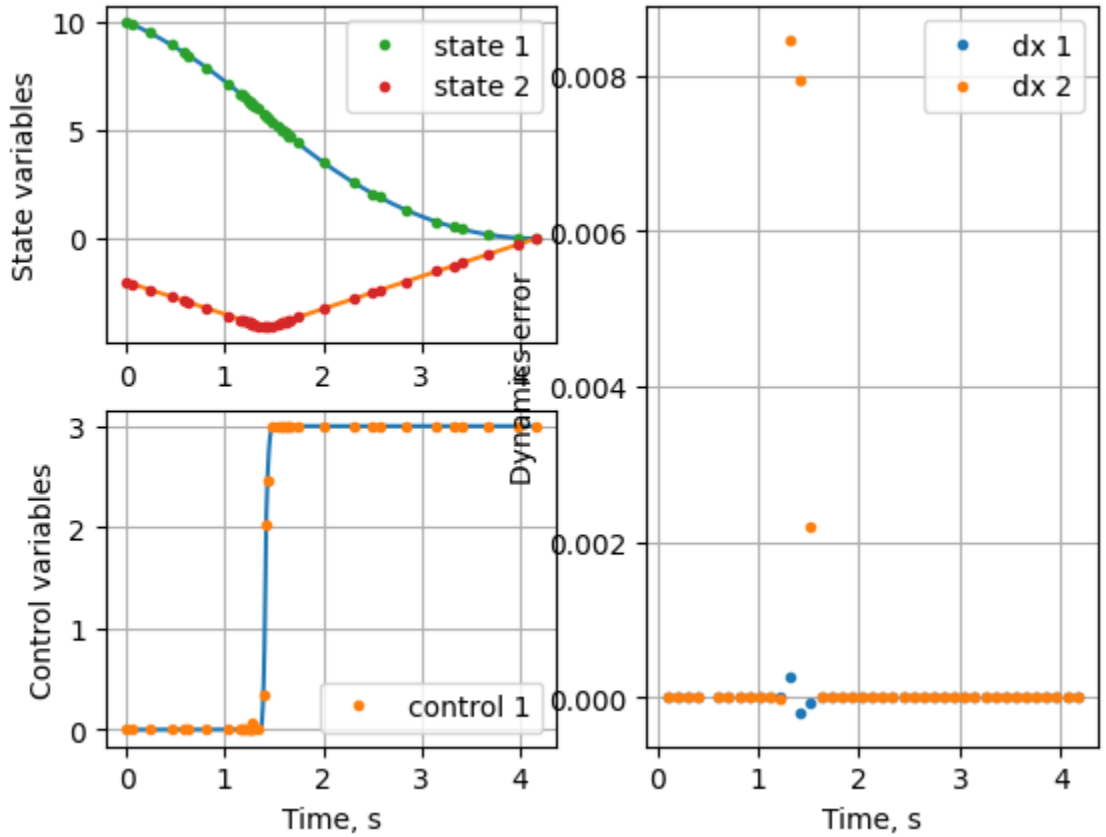
```
 Post processed in 39.596 ms
         Solution retrieval            : 0.124 ms
         Residual in dynamics           : 7.045 ms
         Process solution and plot      : 32.427 ms
```



Let's increase the number of iterations and see how the grid refinement works. Let's enable the residual plot as well.

[27]:
```
max_iter = 2
solution = mpo.solve(max_iter=max_iter, mpopt_options={'method':'control_slope'})
post = mpo.process_results(solution, plot=True)
```

```
 *********** MPOPT H-Adaptive Summary **********

Iteration : 1, 0.03650101311435586
Solved phase 0 to acceptable level 0.01, residual: 0.008459852092415718
Solved to acceptable tolerance 0.01 0.008459852092415718
H-Adaptive Iter., max_residual : 2, 0.008459852092415718
 Optimal cost (J):  8.24648

 Solved in 54.261 ms


 Post processed in 52.279 ms
         Solution retrieval            : 0.094 ms
         Residual in dynamics           : 6.343 ms
         Process solution and plot      : 45.842 ms
```



Note that the grid is refine with more points towards the discontinuity in one iteration. Let's check the terminal time.

[28]:
```
print(f"\nTerminal time using Adaptive +{max_iter-1} iteration , s : {round(post.get_
↪data()[-2][-1][0], 4)} vs 4.1641")
```

```
Terminal time using Adaptive +1 iteration , s : 4.1643 vs 4.1641
```

Let's further increase the iterations and check the final result on convergence.

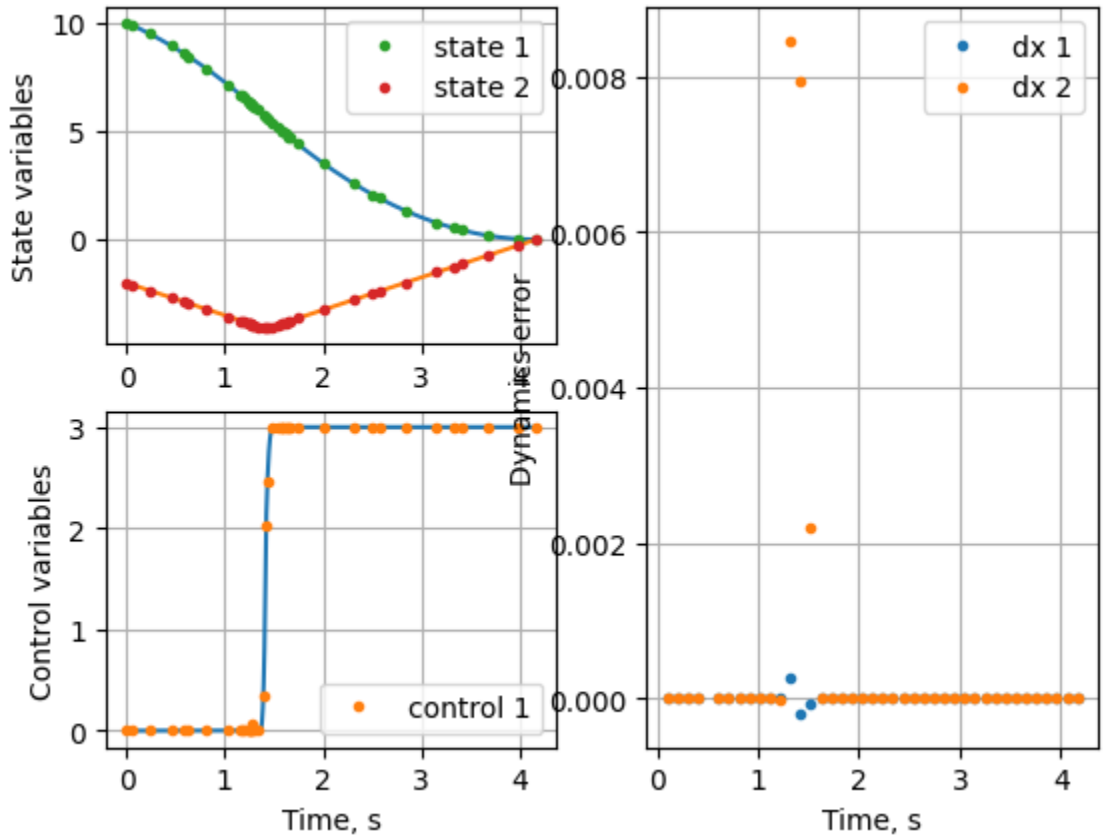
[29]:
```
max_iter = 10
solution = mpo.solve(max_iter=max_iter, mpopt_options={'method':'control_slope'})
post = mpo.process_results(solution, plot=True)
```

```
 *********** MPOPT H-Adaptive Summary **********

Iteration : 1, 0.03650101311435586
Solved phase 0 to acceptable level 0.01, residual: 0.008459852092415718
Solved to acceptable tolerance 0.01 0.008459852092415718
H-Adaptive Iter., max_residual : 2, 0.008459852092415718
 Optimal cost (J):  8.24648

 Solved in 49.589 ms
```

(continues on next page)

(continued from previous page)

```
Post processed in 45.452 ms
        Solution retrieval              : 0.101 ms
        Residual in dynamics            : 6.546 ms
        Process solution and plot       : 38.805 ms
```



[30]:
```
print(f"\nTerminal time using Adaptive +{max_iter-1} iteration , s : {round(post.get_
↪data()[-2][-1][0], 4)} vs 4.1641")
```

```
Terminal time using Adaptive +9 iteration , s : 4.1643 vs 4.1641
```

The solution converged in 3 iterations and terminal time matches to 4 th digit with 40 nodes.

Adaptive scheme-II

The method is based on the estimates of residual at the mid points of the collocation nodes. Further, two sub-methods are implemented in the software

- residual based
  - merge/split
  - equal residual segments

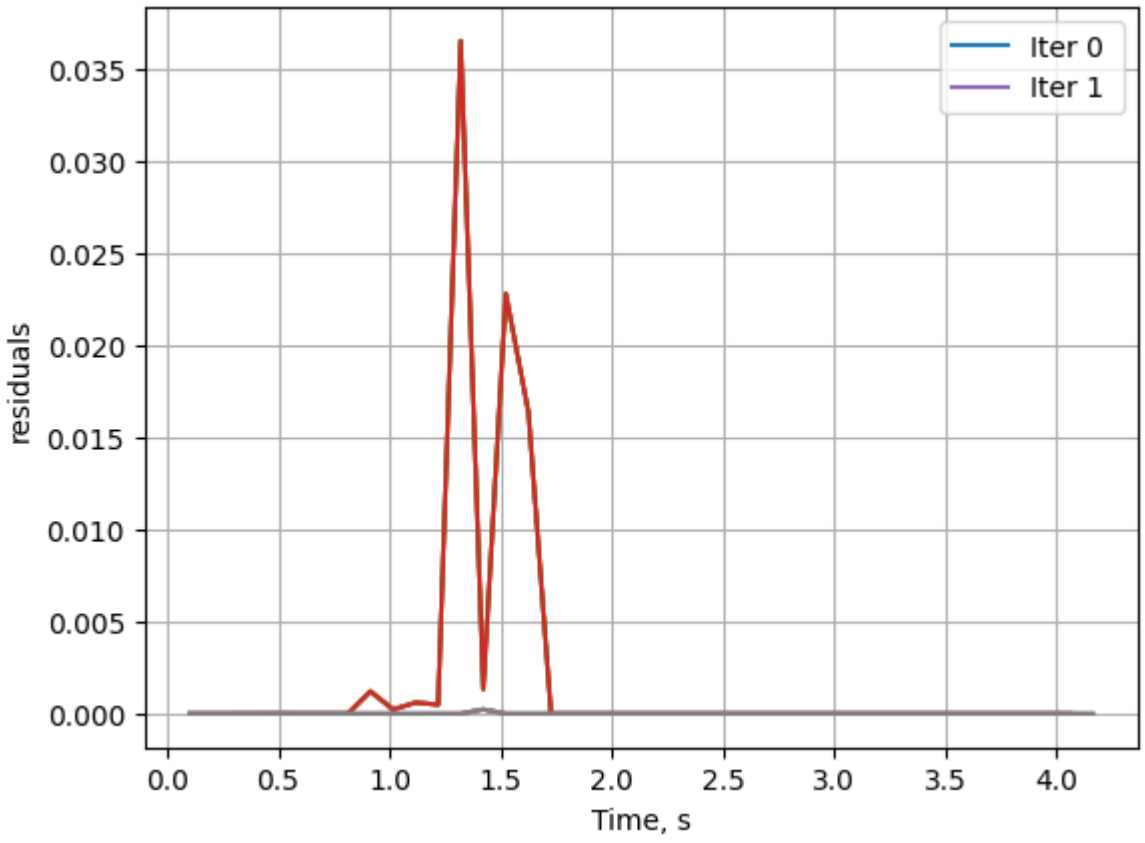
Let's solve the same OCP using Adaptive scheme-II. This time let's enable the residual evolution plots.

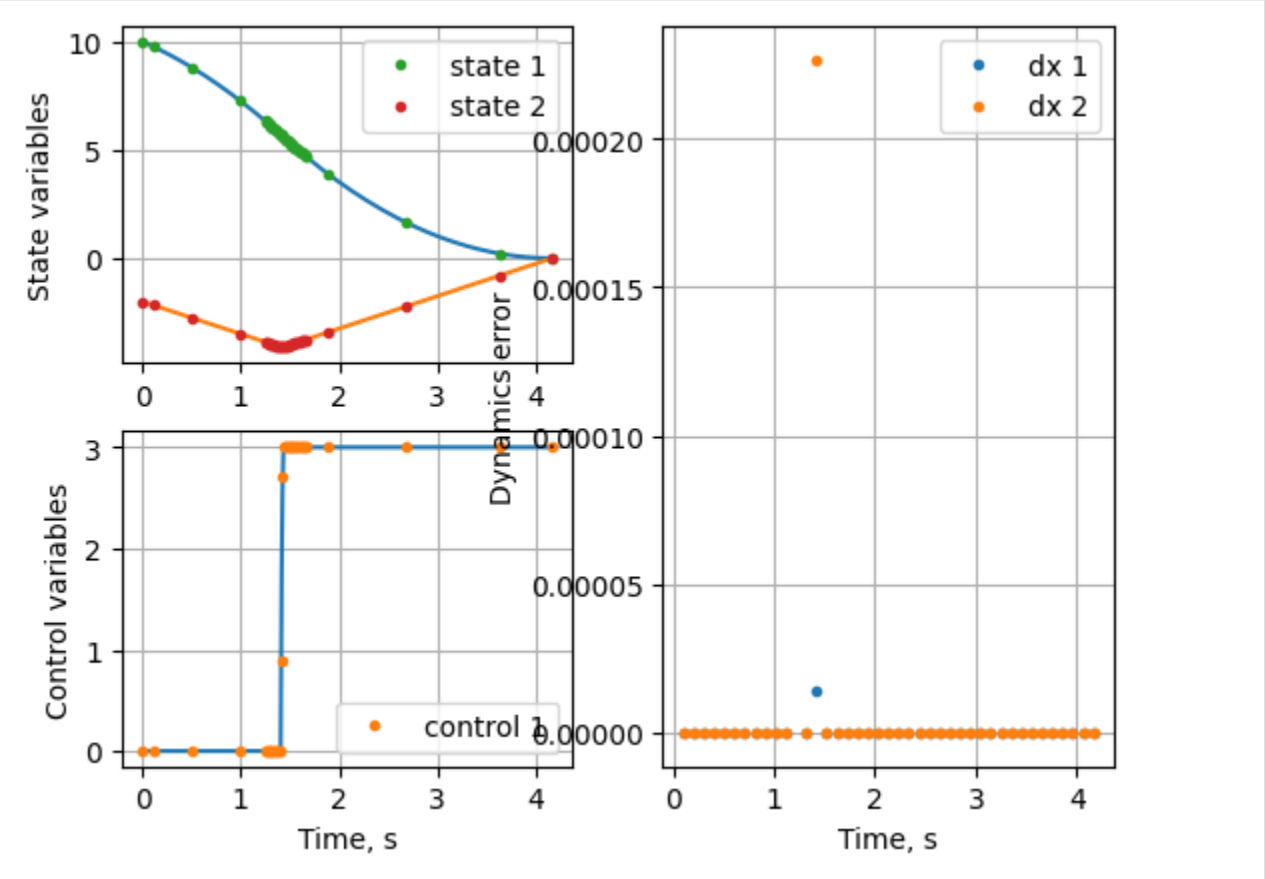
[31]:
```
mpo = mp.mpopt_h_adaptive(ocp, 10, 4) # Use ctrl + tab for help, inside ()
mpo.plot_residual_evolution = True
```

Submethod-1: merge/split

[32]:
```
max_iter = 2
solution = mpo.solve(max_iter=max_iter, mpopt_options={'method':'residual', 'sub_method':
↪'merge_split'})
post = mpo.process_results(solution, plot=True)
```

```
 *********** MPOPT H-Adaptive Summary **********

Iteration : 1, 0.03650101311435586
Solved phase 0 to acceptable tolerance 0.01
Solved to acceptable tolerance 0.01 0.0002265327670160563 5
H-Adaptive Iter., max_residual : 2, 0.0002265327670160563 5
 Optimal cost (J):  8.24624

 Solved in 83.772 ms


 Post processed in 42.848 ms
         Solution retrieval             : 0.129 ms
         Residual in dynamics           : 5.925 ms
         Process solution and plot      : 36.793 ms
```





Notice that the residual in the first iteration is higher towards the discontinuity. In the second iteration, the segments near the start and final times merged so that segments close to the discontinuity are split and the solution converged with in +1 iteration.

The solution is close to the exact solution, lets look at the terminal time.

[33]:
```
print(f"\nTerminal time using Adaptive-II scheme with +{max_iter-1} iterations , s :
↪{post.get_data()[-2][-1][0]} vs 4.1641")
```

```
Terminal time using Adaptive-II scheme with +1 iterations , s : 4.16416187794445 vs 4.
↪1641
```

Lets reduce the tolerance on the residual to see if the solution improves

[34]:
```
mpo = mp.mpopt_h_adaptive(ocp, 10, 4) # Use ctrl + tab for help, inside ()
mpo.plot_residual_evolution = True
mpo.tol_residual[0] = 1e-4
```

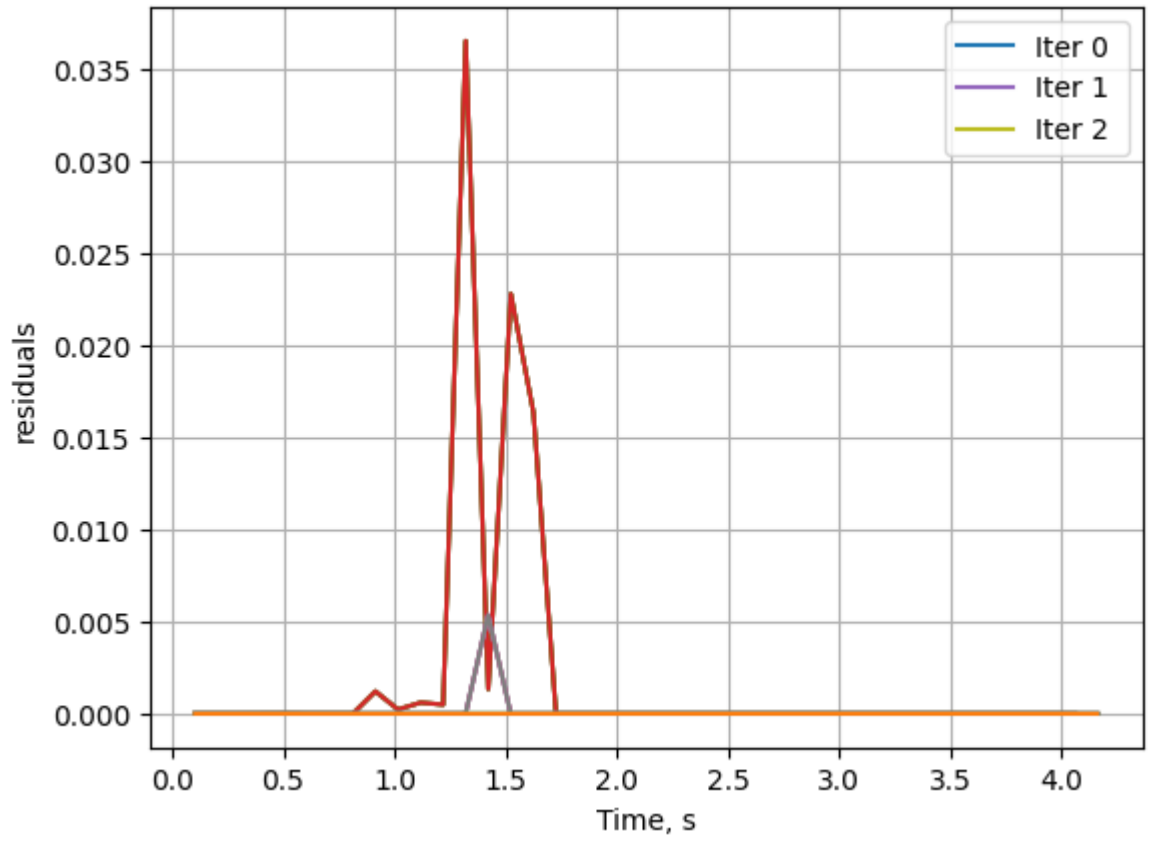
[35]:
```
max_iter = 10
solution = mpo.solve(max_iter=max_iter, mpopt_options={'method':'residual', 'sub_method':
↪'merge_split'})
post = mpo.process_results(solution, plot=True)
```

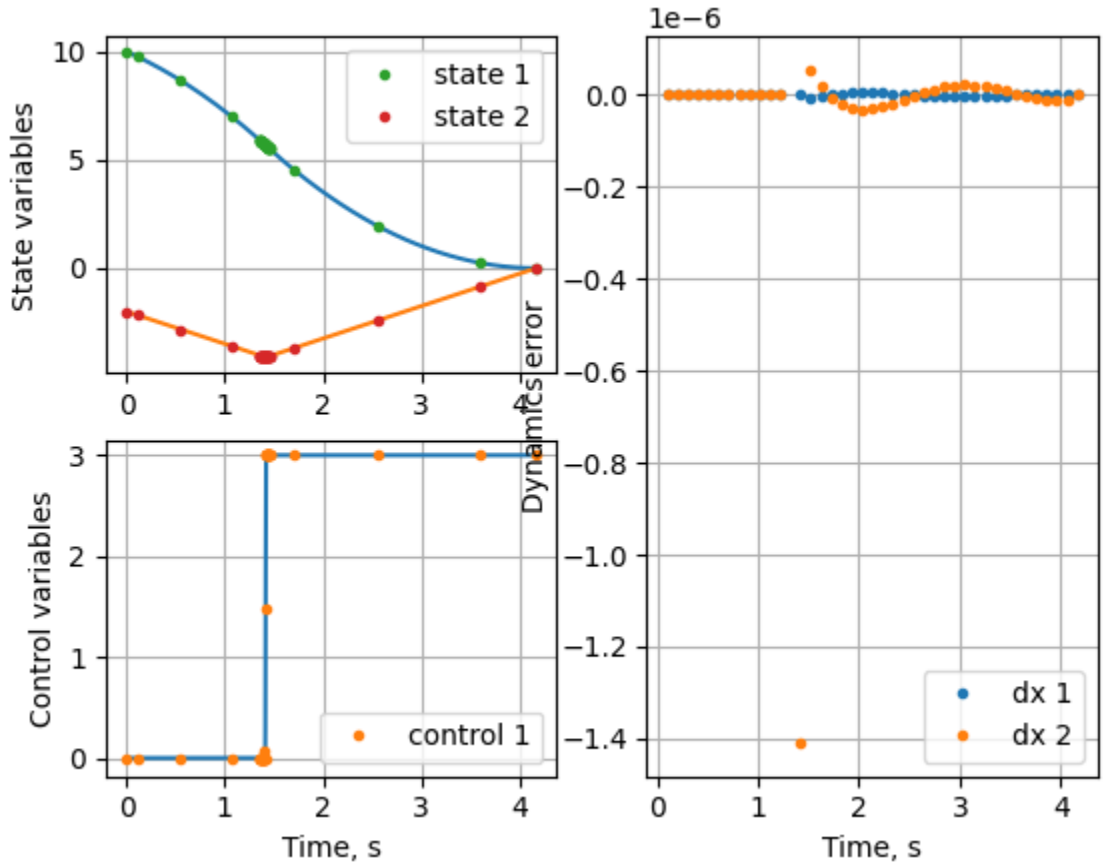
```
********** MPOPT H-Adaptive Summary **********
```

(continues on next page)

(continued from previous page)

```
Iteration : 1, 0.03650101311435586
Iteration : 2, 0.005345204401015982
Solved phase 0 to acceptable tolerance 0.0001
Solved to acceptable tolerance 0.0001 1.4089315576344685e-06
H-Adaptive Iter., max_residual : 3, 1.4089315576344685e-06
 Optimal cost (J):  8.24621

 Solved in 92.315 ms


 Post processed in 49.622 ms
         Solution retrieval             : 0.104 ms
         Residual in dynamics            : 6.128 ms
         Process solution and plot      : 43.39 ms
```





The solution doesn't change after 1st iteration because there is no segment that has residual lower than 1e-4. Residual threshold is a hyper parameters in this method. We can increase the number of segments for the method to work with strict tolerance.

Submethod -2: Equal area segments

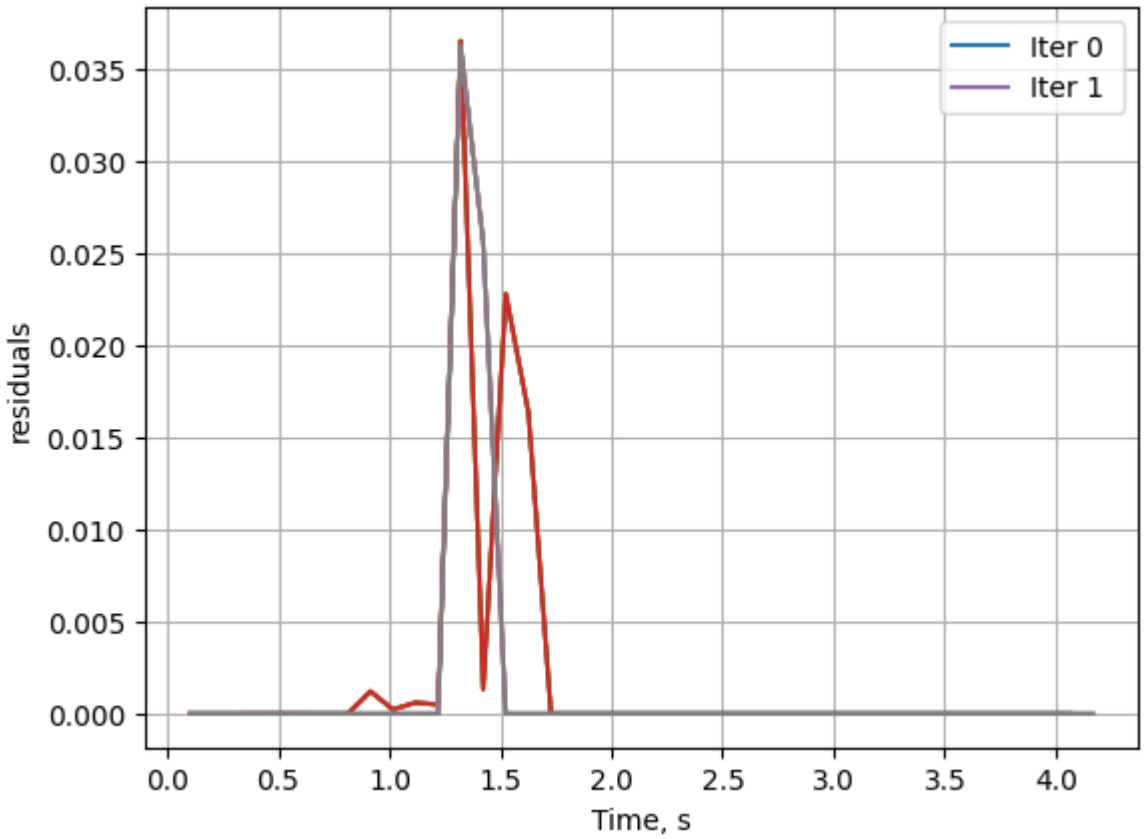
[36]:
```
mpo = mp.mpopt_h_adaptive(ocp, 10, 4) # Use ctrl + tab for help, inside ()
mpo.plot_residual_evolution = True
max_iter = 2
solution = mpo.solve(max_iter=max_iter, mpopt_options={'method':'residual', 'sub_method':
↪'equal_area'})
post = mpo.process_results(solution, plot=True)
```

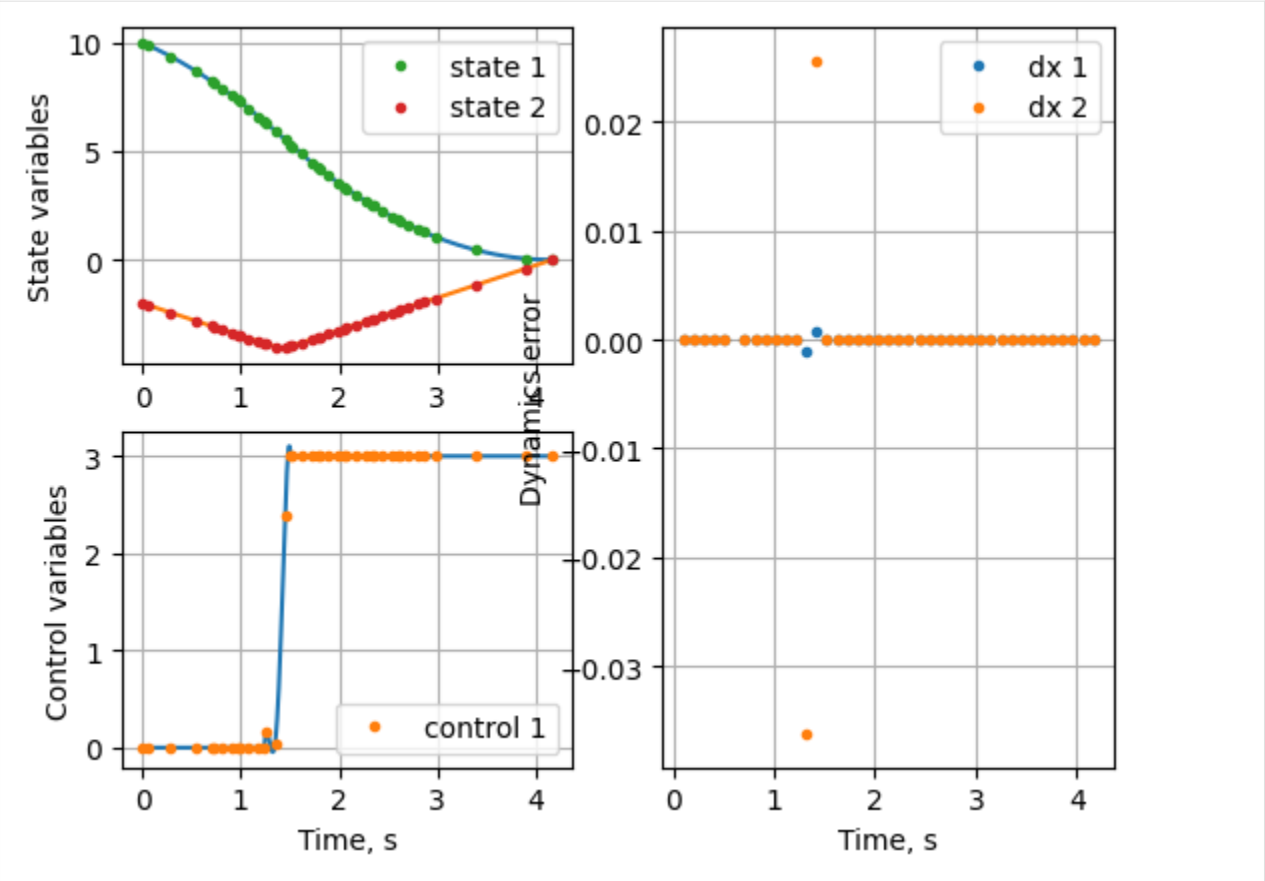
```
 *********** MPOPT H-Adaptive Summary **********

Iteration : 1, 0.03650101311435586
Stopping the iterations: Iteration limit exceeded
H-Adaptive Iter., max_residual : 2, 0.03620404668261404
 Optimal cost (J):  8.24711

 Solved in 66.926 ms


 Post processed in 89.304 ms
```

(continues on next page)

(continued from previous page)

```
Solution retrieval              : 0.115 ms
Residual in dynamics             : 6.486 ms
Process solution and plot       : 82.702 ms
```





Notice that the residual is reduced by concentrating the nodes close to the discontinuity.

The terminal free time in this case is given below

[37]:
```
print(f"\nTerminal time using Adaptive-II scheme-2 with +{max_iter-1} iterations , s :
↪{post.get_data()[-2][-1][0]} vs 4.1641")
```

```
Terminal time using Adaptive-II scheme-2 with +1 iterations , s : 4.164738266741216 vs 4.
↪1641
```

Adaptive schem-III : Direct optimization

The third scheme solves for the optimal segment widths along with the original OCP itself. Hence, no iterative procedure is involved.

For the moon lander problem with one disconitnuiy,one needs only 3 segments to get the exact thrust profile. Further, being bang bang profile, the thrust is constant at both corners.

Let's use only 3 segments and minimum order polynomials to capture the solution to check the effectiveness of the method

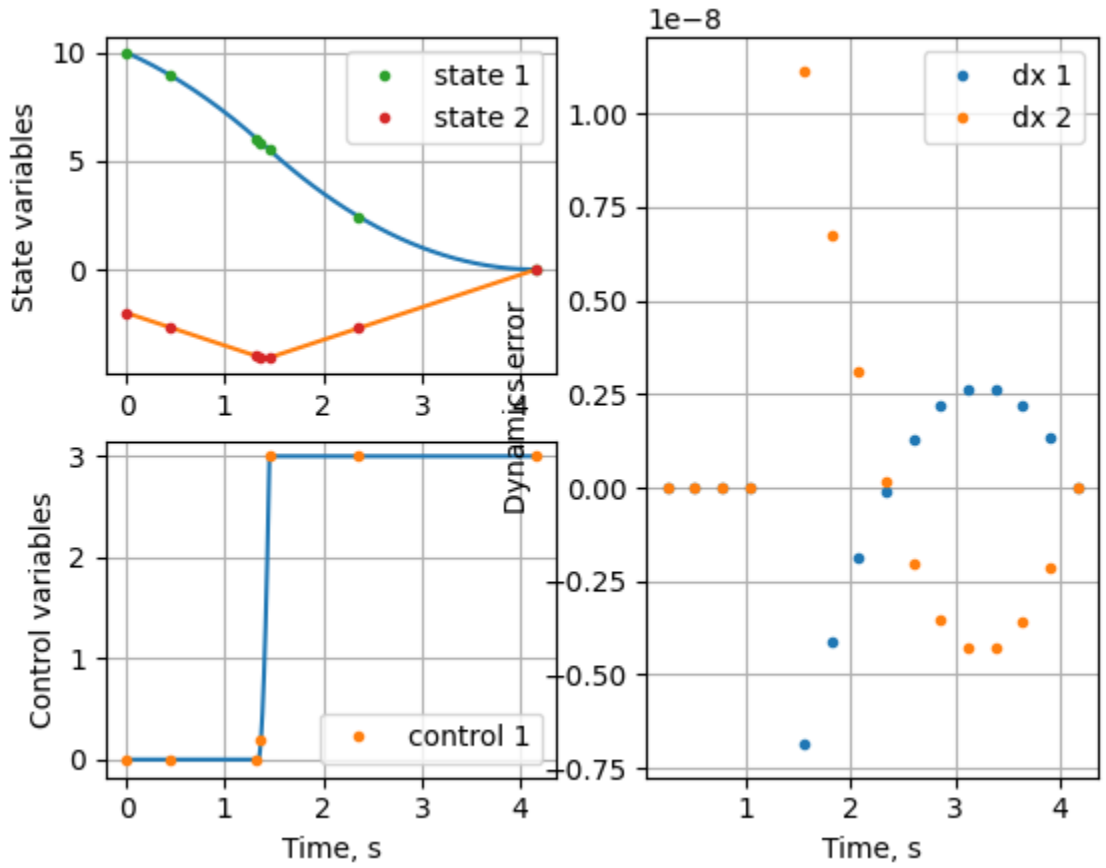
Initialize the optimizer

[38]:
```
mpo = mp.mpopt_adaptive(ocp, 3, 2) # Use ctrl + tab for help, inside ()
mpo.mid_residuals=True
```

Solve the NLP and process the results

[39]:
```
solution = mpo.solve()
post = mpo.process_results(solution, plot=True)
```

```
Optimal segment width fractions: [0.31820188 0.03099379 0.65080433]

 Post processed in 21.747 ms
        Solution retrieval             : 0.146 ms
        Residual in dynamics           : 2.102 ms
        Process solution and plot      : 19.499 ms
```



Notice that the middle segment width is 0.03. That is the default minimu width of the segment implemented in the method. Let's reduce the minimum allowed segment width to 1e-6 and resolve the problem.

[40]:
```
mpo = mp.mpopt_adaptive(ocp, 3, 2) # Use ctrl + tab for help, inside ()
mpo.lbh[0] = 1e-6
```

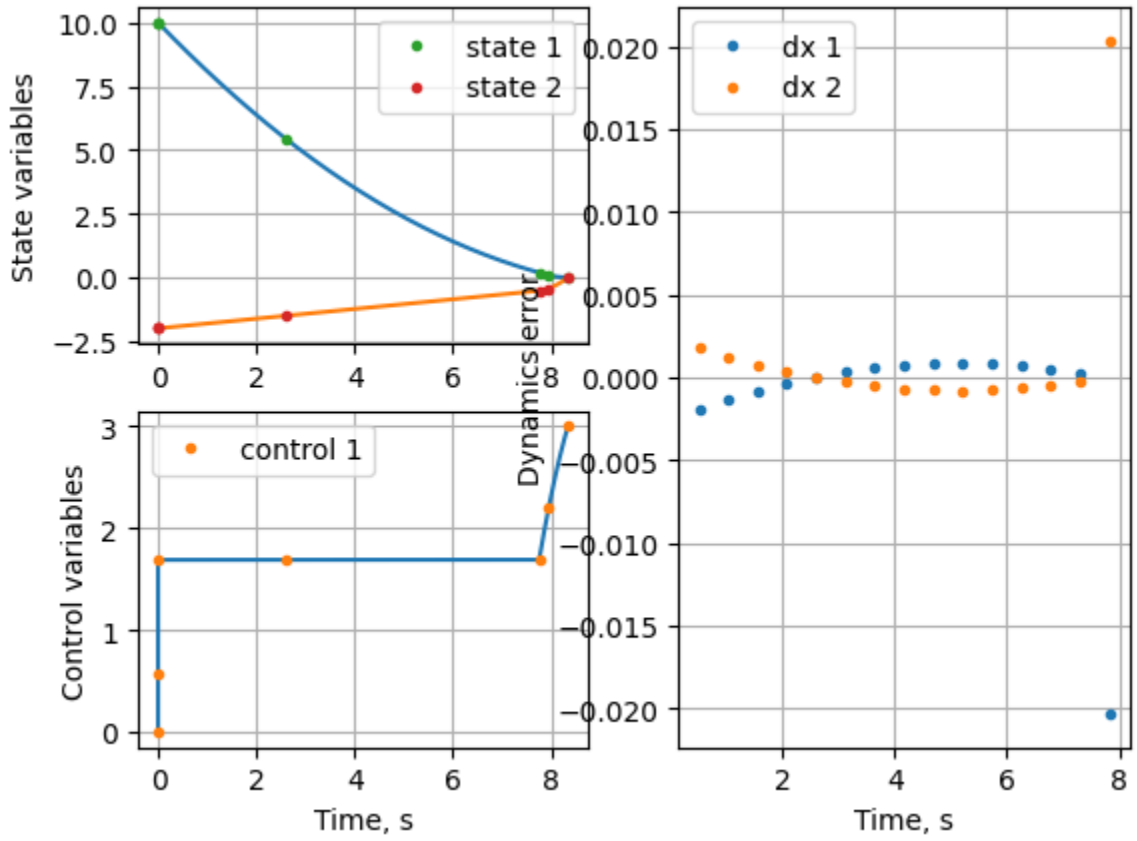
[41]:
```
solution = mpo.solve()
post = mpo.process_results(solution, plot=True)
```

```
Optimal segment width fractions: [2.62292233e-06 9.29740025e-01 7.02573521e-02]

 Post processed in 23.777 ms
        Solution retrieval             : 0.109 ms
        Residual in dynamics           : 2.372 ms
```

(continues on next page)

(continued from previous page)

```
        Process solution and plot      : 21.295 ms
```



Notice that the discontinuity is exactly captured using the method with minimum possible collocation nodes.

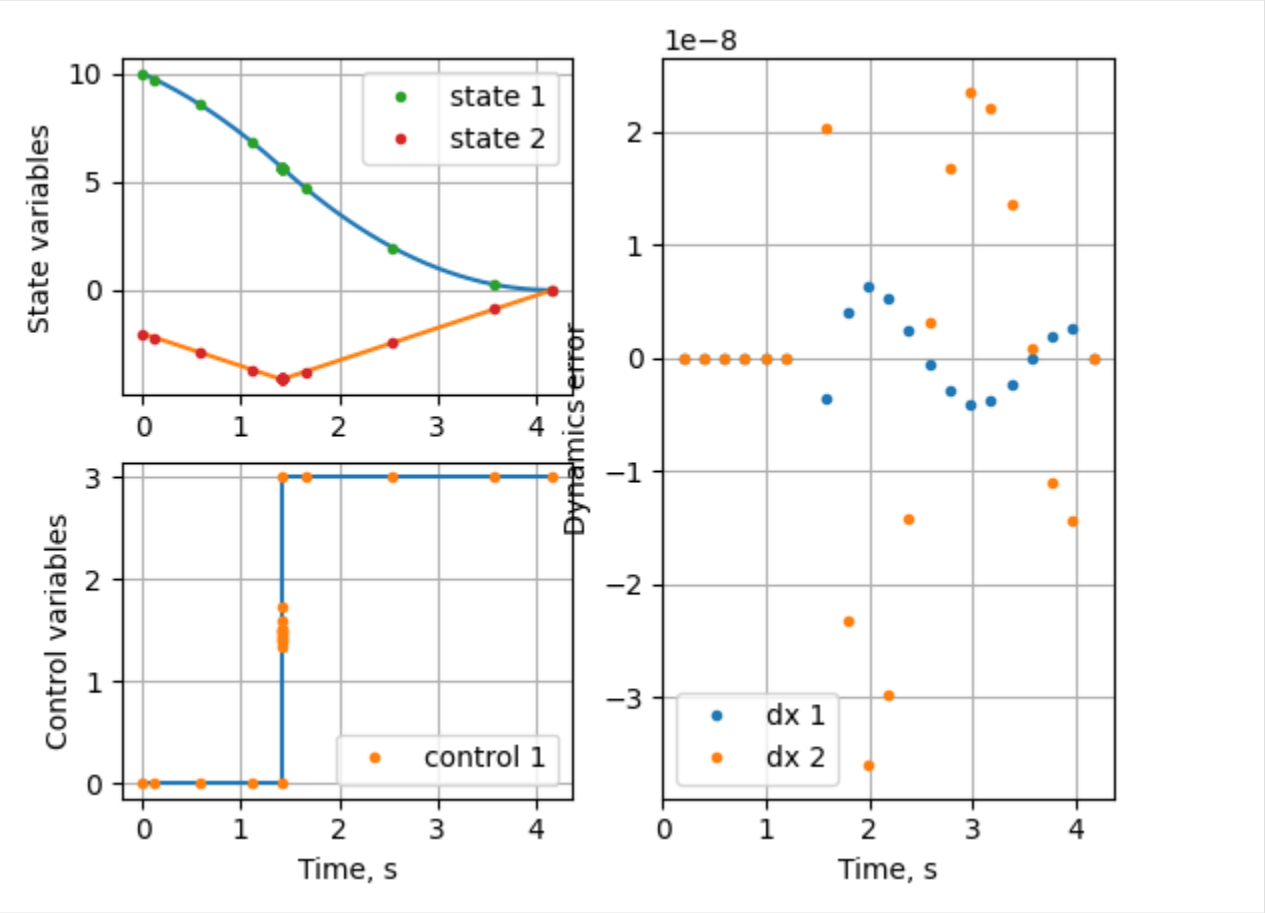
[42]:
```
print(f"\nTerminal time using Adaptive scheme-III , s : {post.get_data()[-2][-1][0]} vs␣
↪4.1641")
```

```
Terminal time using Adaptive scheme-III , s : 8.3414904388322 vs 4.1641
```

Let's check the robustness of the method by increasing the number of segments

[43]:
```
mpo = mp.mpopt_adaptive(ocp, 5, 4) # Use ctrl + tab for help, inside ()
mpo.lbh[0] = 1e-6
solution = mpo.solve()
post = mpo.process_results(solution, plot=True)
```

```
Optimal segment width fractions: [3.39881653e-01 1.43712770e-05 1.40014871e-05 1.
↪48705948e-05
 6.60075103e-01]

 Post processed in 26.976 ms
         Solution retrieval             : 0.119 ms
         Residual in dynamics           : 3.358 ms
         Process solution and plot      : 23.498 ms
```



Notice that the time taken by the algorithm is significantly higher compared to iterative schemes. However, the solution is accurate using the Adaptive scheme-III.

Conclusion

Using simple moon-lander test case OCP various features of the solver package in its current form are demonstrated. Notice that the OCP is defined only once in the beginning. This sheet can be used for all other OCP's merely by changing the OCP definition in the beginning.

CHAPTER

FOUR

EXAMPLES

4.1 Single phase

Refer examples folder in GitHub

4.1.1 Van der pol Oscillator

(c) 2023 Devakumar Thammisetty

MPOPT is an open-source Multi-phase Optimal Control Problem (OCP) solver based on pseudo-spectral collocation with customized adaptive grid refinement techniques.

https://mpopt.readthedocs.io/

Download this notebook: vanderpol.ipynb

Install mpopt from pypi using the following. Disable after first usage

Import mpopt (Contains main solver modules)

[1]:
```
#!pip install mpopt
from mpopt import mp
```

OCP definition

Van der Pol OCP: https://web.casadi.org/docs/#a-simple-test-problem

$$
\begin{aligned}
\min_{x,u} \quad & J = 0 + \int_{t_0}^{t_f} \left(x_0^2 + x_1^2 + u^2\right) dt \\
\text{subject to} \quad & \dot{x_0} = (1 - x_1^2) \times x_0 - x_1 + u \\
& \dot{x_1} = x_0 \\
& x_1 \geq -0.25 \\
& -1 \leq u \leq 1 \\
& x_0(t_0) = 0;\ x_1(t_0) = 1; \\
& t_0 = 0.0;\ t_f = 10
\end{aligned}
$$

We first create an OCP object and then polulate the object with dynamics, path_constraints, terminal_constraints and objective (running_costs, terminal_costs)

[2]:
```
ocp = mp.OCP(n_states=2, n_controls=1)
```

[3]:
```
def dynamics(x, u, t):
    return [(1 - x[1] * x[1]) * x[0] - x[1] + u[0], x[0]]
```

[4]:
```
def running_cost(x, u, t):
    return x[0] * x[0] + x[1] * x[1] + u[0] * u[0]
```

[5]:
```
ocp.dynamics[0] = dynamics
ocp.running_costs[0] = running_cost
```

Initial state

[6]:
```
ocp.x00[0] = [0, 1]
```

Box constraints

[7]:
```
ocp.lbu[0] = -1.0
ocp.ubu[0] = 1.0
ocp.lbx[0][1] = -0.25
ocp.lbtf[0] = 10.0
ocp.ubtf[0] = 10.0
```

[8]:
```
ocp.validate()
```

Solve and plot the results in one line

Lets solve the OCP using following pseudo-spectral approximation * Collocation using Legendre-Gauss-Radau roots * Let's plot the position and velocity evolution with time starting from 0.

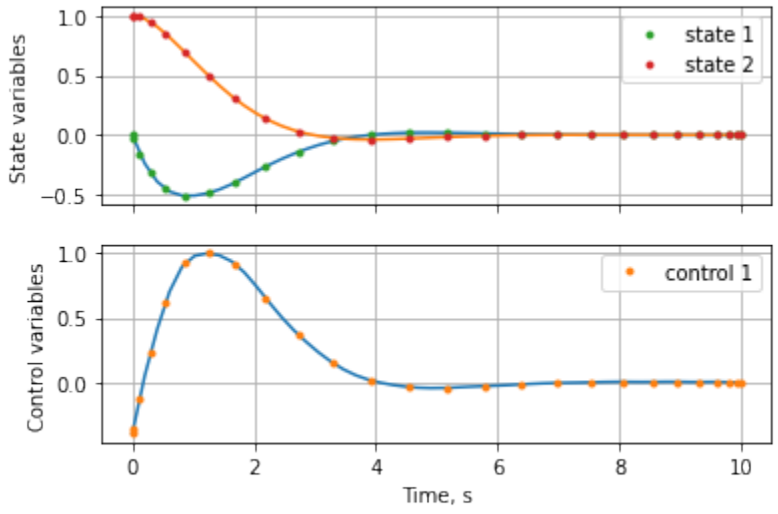
[9]:
```
mpo, post = mp.solve(ocp, n_segments=1, poly_orders=25, scheme="LGR", plot=True)
```

```
******************************************************************************
This program contains Ipopt, a library for large-scale nonlinear optimization.
 Ipopt is released as open source code under the Eclipse Public License (EPL).
         For more information visit http://projects.coin-or.org/Ipopt
******************************************************************************

Total number of variables...:       76
                     variables with only lower bounds:       25
                variables with lower and upper bounds:       26
                     variables with only upper bounds:        0
Total number of equality constraints...:       52
Total number of inequality constraints...:       25
        inequality constraints with only lower bounds:        0
   inequality constraints with lower and upper bounds:       25
        inequality constraints with only upper bounds:        0


Number of Iterations...: 11
```

(continues on next page)

(continued from previous page)

```
                                   (scaled)                 (unscaled)
Objective...:   2.8734932991287625e+00    2.8734932991287625e+00
Dual infeasibility...:   3.4605651677566129e-13    3.4605651677566129e-13
Constraint violation...:   4.6562753652779065e-13    4.6562753652779065e-13
Complementarity...:   2.5153452656320664e-09    2.5153452656320664e-09
Overall NLP error...:   2.5153452656320664e-09    2.5153452656320664e-09


Number of objective function evaluations             = 12
Number of objective gradient evaluations             = 12
Number of equality constraint evaluations            = 12
Number of inequality constraint evaluations          = 12
Number of equality constraint Jacobian evaluations   = 12
Number of inequality constraint Jacobian evaluations = 12
Number of Lagrangian Hessian evaluations             = 11
Total CPU secs in IPOPT (w/o function evaluations)   =      0.016
Total CPU secs in NLP function evaluations           =      0.002

EXIT: Optimal Solution Found.
      solver  :   t_proc      (avg)   t_wall      (avg)    n_eval
       nlp_f  |  53.00us (  4.42us)  52.65us (  4.39us)        12
       nlp_g  | 327.00us ( 27.25us) 323.85us ( 26.99us)        12
    nlp_grad  |  58.00us ( 58.00us)  55.45us ( 55.45us)         1
  nlp_grad_f  |  77.00us (  5.92us)  77.00us (  5.92us)        13
  nlp_hess_l  | 112.00us ( 10.18us) 115.53us ( 10.50us)        11
   nlp_jac_g  | 515.00us ( 39.62us) 515.59us ( 39.66us)        13
       total  |  19.01ms ( 19.01ms)  19.51ms ( 19.51ms)         1
```



Retrive the solution

x: states, u: Controls, t:time, a:Algebraic variables

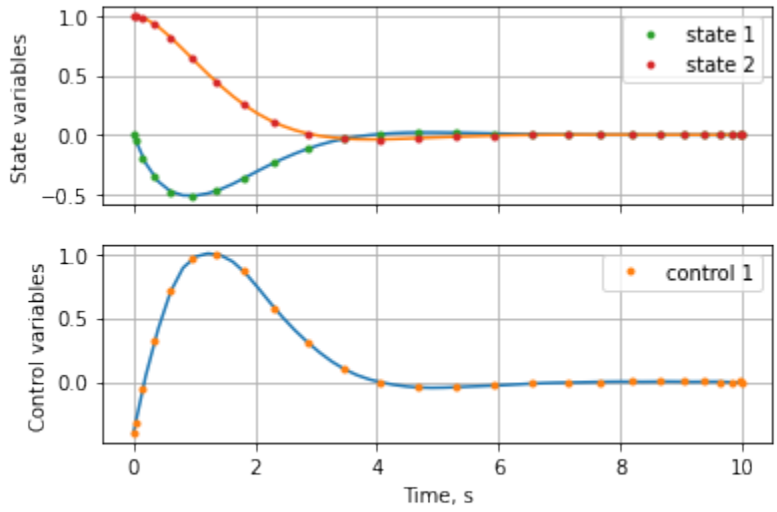
[10]:
```
x, u, t, a = post.get_data()
print(f"Terminal time, state : {t[-1][0]:.4f} vs 10 (Exact), {x[-1]}")
```

```
Terminal time, state : 10.0000 vs 10 (Exact), [-0.00178384 -0.00014044]
```

Solve again with Chebyshev-Gauss-Lobatto (CGL) roots

[11]:
```
mpo, post = mp.solve(ocp, n_segments=1, poly_orders=25, scheme="CGL", plot=True)
```

```
Total number of variables...:       76
                     variables with only lower bounds:       25
                variables with lower and upper bounds:       26
                     variables with only upper bounds:        0
Total number of equality constraints...:       52
Total number of inequality constraints...:       25
        inequality constraints with only lower bounds:        0
   inequality constraints with lower and upper bounds:       25
        inequality constraints with only upper bounds:        0


Number of Iterations...: 12

                                   (scaled)                 (unscaled)
Objective...:   2.8734060736207896e+00    2.8734060736207896e+00
Dual infeasibility...:   2.6383995374325467e-12    2.6383995374325467e-12
Constraint violation...:   2.1178614417749486e-12    2.1178614417749486e-12
Complementarity...:   2.5713635670826573e-09    2.5713635670826573e-09
Overall NLP error...:   2.5713635670826573e-09    2.5713635670826573e-09


Number of objective function evaluations             = 13
Number of objective gradient evaluations             = 13
Number of equality constraint evaluations            = 13
Number of inequality constraint evaluations          = 13
Number of equality constraint Jacobian evaluations   = 13
Number of inequality constraint Jacobian evaluations = 13
Number of Lagrangian Hessian evaluations             = 12
Total CPU secs in IPOPT (w/o function evaluations)   =      0.013
Total CPU secs in NLP function evaluations           =      0.001

EXIT: Optimal Solution Found.
      solver  :   t_proc      (avg)   t_wall      (avg)    n_eval
       nlp_f  |  58.00us (  4.46us)  57.81us (  4.45us)        13
       nlp_g  | 358.00us ( 27.54us) 355.20us ( 27.32us)        13
    nlp_grad  |  68.00us ( 68.00us)  67.78us ( 67.78us)         1
  nlp_grad_f  |  86.00us (  6.14us)  83.08us (  5.93us)        14
  nlp_hess_l  | 127.00us ( 10.58us) 126.75us ( 10.56us)        12
   nlp_jac_g  | 536.00us ( 38.29us) 610.09us ( 43.58us)        14
       total  |  20.51ms ( 20.51ms)  20.98ms ( 20.98ms)         1
```



[12]:
```
x, u, t, a = post.get_data()
print(f"Terminal time, state : {t[-1][0]:.4f} vs 10s (Exact), {x[-1]}")
```

```
Terminal time, state : 10.0000 vs 10s (Exact), [-0.00178269 -0.00013973]
```

Solve again with Legendre-Gauss-Lobatto (LGL) roots

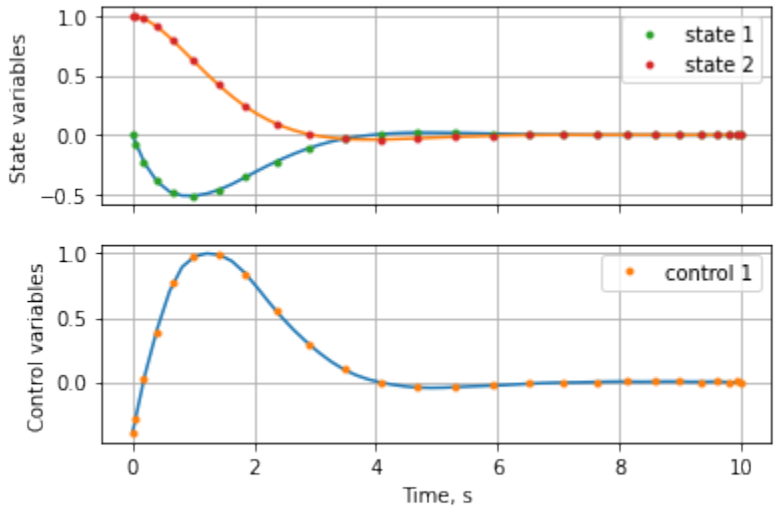
[13]:
```
mpo, post = mp.solve(ocp, n_segments=1, poly_orders=25, scheme="LGL", plot=True)
```

```
Total number of variables...:       76
                     variables with only lower bounds:       25
                variables with lower and upper bounds:       26
                     variables with only upper bounds:        0
Total number of equality constraints...:       52
Total number of inequality constraints...:       25
        inequality constraints with only lower bounds:        0
   inequality constraints with lower and upper bounds:       25
        inequality constraints with only upper bounds:        0


Number of Iterations...: 11

                                   (scaled)                 (unscaled)
Objective...:   2.8734849959084205e+00    2.8734849959084205e+00
Dual infeasibility...:   8.2869030765986375e-12    8.2869030765986375e-12
Constraint violation...:   6.0937921375625592e-12    6.0937921375625592e-12
Complementarity...:   2.6832881949699023e-09    2.6832881949699023e-09
Overall NLP error...:   2.6832881949699023e-09    2.6832881949699023e-09


Number of objective function evaluations             = 12
Number of objective gradient evaluations             = 12
Number of equality constraint evaluations            = 12
Number of inequality constraint evaluations          = 12
```

(continues on next page)

(continued from previous page)

```
Number of equality constraint Jacobian evaluations   = 12
Number of inequality constraint Jacobian evaluations = 12
Number of Lagrangian Hessian evaluations             = 11
Total CPU secs in IPOPT (w/o function evaluations)   =      0.015
Total CPU secs in NLP function evaluations           =      0.002

EXIT: Optimal Solution Found.
      solver  :   t_proc      (avg)   t_wall      (avg)    n_eval
       nlp_f  |  57.00us (  4.75us)  54.18us (  4.51us)        12
       nlp_g  | 330.00us ( 27.50us) 326.96us ( 27.25us)        12
    nlp_grad  |  53.00us ( 53.00us)  52.09us ( 52.09us)         1
  nlp_grad_f  |  77.00us (  5.92us)  76.10us (  5.85us)        13
  nlp_hess_l  | 119.00us ( 10.82us) 116.95us ( 10.63us)        11
   nlp_jac_g  | 510.00us ( 39.23us) 516.94us ( 39.76us)        13
       total  |  18.17ms ( 18.17ms)  17.80ms ( 17.80ms)         1
```



[14]:
```
x, u, t, a = post.get_data()
print(f"Terminal time, state : {t[-1][0]:.4f} vs 10s (Exact), {x[-1]}")
```

```
Terminal time, state : 10.0000 vs 10s (Exact), [-0.00178275 -0.00014031]
```

[ ]:

4.1.2 Moon lander (2D) Example

(c) 2023 Devakumar Thammisetty

MPOPT is an open-source Multi-phase Optimal Control Problem (OCP) solver based on pseudo-spectral collocation with customized adaptive grid refinement techniques.

Download this notebook: moon_lander.ipynb

Install mpopt from pypi using the following. Disable after first usage

Import mpopt (Contains main solver modules)

[1]:
```
#!pip install mpopt
from mpopt import mp
```

Defining OCP

The Fuel optimal solution to the moon-lander OCP is known to have bang-bang thrust profile. The selected OCP has one discontinuity.

$$
\begin{aligned}
\min_{x,u} \quad & J = 0 + \int_{t_0}^{t_f} u \, dt \\
\text{subject to} \quad & \dot{x_0} = x_1; \dot{x_1} = u - 1.5 \\
& x_0(t_f) = 0; \; x_1(t_f) = 0 \\
& x_0(t_0) = 10; \; x_1(t_0) = -2 \\
& x_0 \geq 0; 0 \leq u \leq 3 \\
& t_0 = 0.0; t_f = \text{free variable}
\end{aligned}
$$

We first create an OCP object and then polulate the object with dynamics, path_constraints, terminal_constraints and objective (running_costs, terminal_costs)

[2]:
```
ocp = mp.OCP(n_states=2, n_controls=1, n_phases=1)
```

[3]:
```
ocp.dynamics[0] = lambda x, u, t: [x[1], u[0] - 1.5]
```

[4]:
```
ocp.running_costs[0] = lambda x, u, t: u[0]
```

[5]:
```
ocp.terminal_constraints[0] = lambda xf, tf, x0, t0: [xf[0], xf[1]]
```

Initial state

[6]:
```
ocp.x00[0] = [10.0, -2.0]
```

Box constraints

[7]:
```
ocp.lbx[0][0] = 0.0
ocp.lbu[0], ocp.ubu[0] = 0, 3
```

[8]:
```
ocp.validate()
```

Solve and plot the results in one line

Lets solve the OCP using following pseudo-spectral approximation * Collocation using Legendre-Gauss-Radau roots * Let's plot the position and velocity evolution with time starting from 0.

The OCP is a free final time formulation,

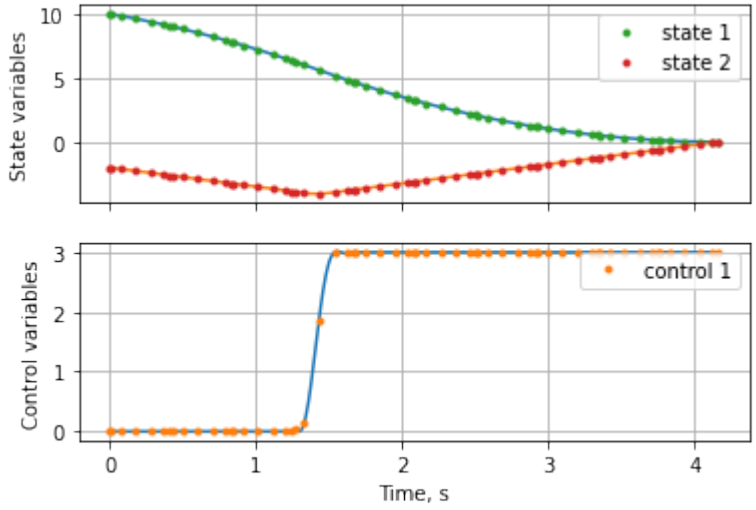
[9]:
```
mpo, post = mp.solve(ocp, n_segments=10, poly_orders=6, scheme="LGR", plot=True)
```

```
******************************************************************************
This program contains Ipopt, a library for large-scale nonlinear optimization.
```

(continues on next page)

(continued from previous page)

```
 Ipopt is released as open source code under the Eclipse Public License (EPL).
         For more information visit http://projects.coin-or.org/Ipopt
******************************************************************************

Total number of variables...:      182
                     variables with only lower bounds:       61
                variables with lower and upper bounds:       61
                     variables with only upper bounds:        0
Total number of equality constraints...:      124
Total number of inequality constraints...:       60
        inequality constraints with only lower bounds:        0
   inequality constraints with lower and upper bounds:       60
        inequality constraints with only upper bounds:        0


Number of Iterations...: 32

                                   (scaled)                 (unscaled)
Objective...:   8.247725507578303e+00    8.247725507578303e+00
Dual infeasibility...:   1.9684090468707893e-10    1.9684090468707893e-10
Constraint violation...:   4.0178316229599886e-10    4.0178316229599886e-10
Complementarity...:   4.4108642187588750e-09    4.4108642187588750e-09
Overall NLP error...:   4.4108642187588750e-09    4.4108642187588750e-09


Number of objective function evaluations             = 33
Number of objective gradient evaluations             = 33
Number of equality constraint evaluations            = 33
Number of inequality constraint evaluations          = 33
Number of equality constraint Jacobian evaluations   = 33
Number of inequality constraint Jacobian evaluations = 33
Number of Lagrangian Hessian evaluations             = 32
Total CPU secs in IPOPT (w/o function evaluations)   =      0.046
Total CPU secs in NLP function evaluations           =      0.004

EXIT: Optimal Solution Found.
      solver  :   t_proc      (avg)   t_wall      (avg)    n_eval
       nlp_f  | 145.00us (  4.39us) 144.99us (  4.39us)        33
       nlp_g  | 781.00us ( 23.67us) 774.17us ( 23.46us)        33
    nlp_grad  |  47.00us ( 47.00us)  44.91us ( 44.91us)         1
  nlp_grad_f  | 213.00us (  6.26us) 212.28us (  6.24us)        34
  nlp_hess_l  | 274.00us (  8.56us) 269.97us (  8.44us)        32
   nlp_jac_g  |   1.03ms ( 30.24us)   1.04ms ( 30.44us)        34
       total  |  52.43ms ( 52.43ms)  51.88ms ( 51.88ms)         1
```



Lets retrive the solution to see the terminal time.

x: states, u: Controls, t:time, a:Algebraic variables in case OCP has differential algebraic equations (DAEs)

Last element of t and x gives the terminal values. Exact terminal time from the analytical solution is 4.1641s.

[10]:
```
x, u, t, a = post.get_data()
print(f"Terminal time, state : {t[-1][0]:.4f} vs 4.1641s (Exact), {x[-1]}")
```

```
Terminal time, state : 4.1652 vs 4.1641s (Exact), [9.85970078e-37 0.00000000e+00]
```

Solve again with Chebyshev-Gauss-Lobatto (CGL) roots

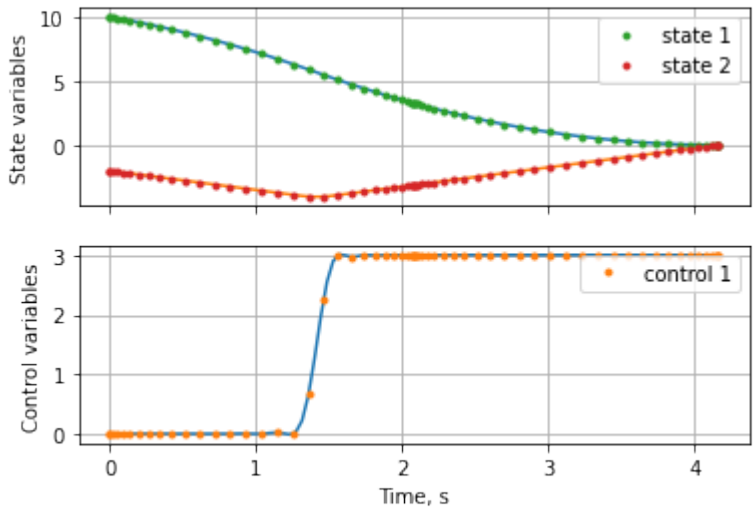
[11]:
```
mpo, post = mp.solve(ocp, n_segments=2, poly_orders=30, scheme="CGL", plot=True)
```

```
Total number of variables...:      182
                     variables with only lower bounds:       61
                variables with lower and upper bounds:       61
                     variables with only upper bounds:        0
Total number of equality constraints...:      124
Total number of inequality constraints...:       60
        inequality constraints with only lower bounds:        0
   inequality constraints with lower and upper bounds:       60
        inequality constraints with only upper bounds:        0


Number of Iterations...: 41

                                   (scaled)                 (unscaled)
Objective...:   8.2457172048588543e+00    8.2457172048588543e+00
Dual infeasibility...:   1.3972013274602247e-12    1.3972013274602247e-12
Constraint violation...:   3.4862335240859466e-11    3.4862335240859466e-11
Complementarity...:   3.9978925923296842e-09    3.9978925923296842e-09
Overall NLP error...:   3.9978925923296842e-09    3.9978925923296842e-09
```

(continues on next page)

(continued from previous page)

```
Number of objective function evaluations             = 42
Number of objective gradient evaluations             = 42
Number of equality constraint evaluations            = 42
Number of inequality constraint evaluations          = 42
Number of equality constraint Jacobian evaluations   = 42
Number of inequality constraint Jacobian evaluations = 42
Number of Lagrangian Hessian evaluations             = 41
Total CPU secs in IPOPT (w/o function evaluations)   =      0.150
Total CPU secs in NLP function evaluations           =      0.009

EXIT: Optimal Solution Found.
      solver  :   t_proc      (avg)   t_wall      (avg)    n_eval
       nlp_f  | 188.00us (  4.48us) 183.99us (  4.38us)        42
       nlp_g  |   2.48ms ( 58.98us)   2.49ms ( 59.19us)        42
    nlp_grad  | 114.00us (114.00us) 113.29us (113.29us)         1
  nlp_grad_f  | 273.00us (  6.35us) 266.81us (  6.20us)        43
  nlp_hess_l  | 336.00us (  8.20us) 332.45us (  8.11us)        41
   nlp_jac_g  |   3.81ms ( 88.63us)   3.80ms ( 88.47us)        43
       total  | 167.97ms (167.97ms) 167.12ms (167.12ms)         1
```



[12]:
```
x, u, t, a = post.get_data()
print(f"Terminal time, state : {t[-1][0]:.4f} vs 4.1641s (Exact), {x[-1]}")
```

```
Terminal time, state : 4.1661 vs 4.1641s (Exact), [0. 0.]
```

Solve again with Legendre-Gauss-Lobatto (LGL) roots

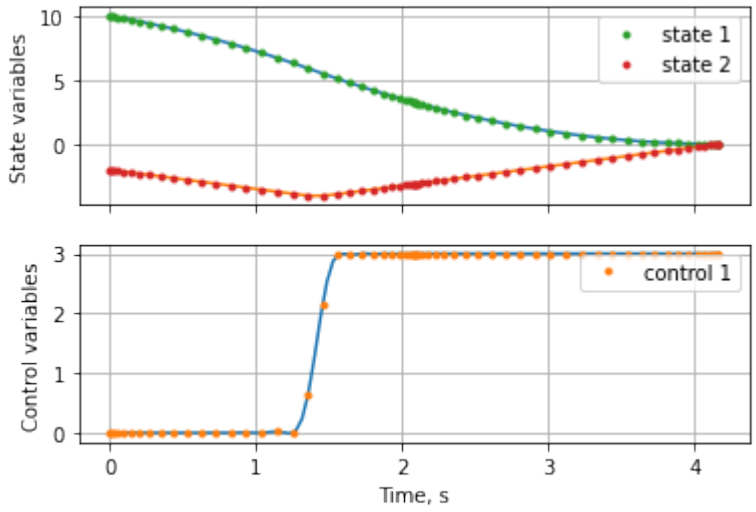
[13]:
```
mpo, post = mp.solve(ocp, n_segments=2, poly_orders=30, scheme="LGL", plot=True)
```

```
Total number of variables...:      182
                     variables with only lower bounds:       61
                variables with lower and upper bounds:       61
                     variables with only upper bounds:        0
Total number of equality constraints...:      124
```

(continues on next page)

(continued from previous page)

```
Total number of inequality constraints...:       60
        inequality constraints with only lower bounds:        0
   inequality constraints with lower and upper bounds:       60
        inequality constraints with only upper bounds:        0


Number of Iterations...: 42

                                   (scaled)                 (unscaled)
Objective...:   8.2425586640613506e+00    8.2425586640613506e+00
Dual infeasibility...:   6.8055248936271795e-11    6.8055248936271795e-11
Constraint violation...:   6.9194650009762881e-12    8.8133944586843427e-12
Complementarity...:   4.7886457777884481e-09    4.7886457777884481e-09
Overall NLP error...:   4.7886457777884481e-09    4.7886457777884481e-09


Number of objective function evaluations             = 43
Number of objective gradient evaluations             = 43
Number of equality constraint evaluations            = 43
Number of inequality constraint evaluations          = 43
Number of equality constraint Jacobian evaluations   = 43
Number of inequality constraint Jacobian evaluations = 43
Number of Lagrangian Hessian evaluations             = 42
Total CPU secs in IPOPT (w/o function evaluations)   =      0.139
Total CPU secs in NLP function evaluations           =      0.010

EXIT: Optimal Solution Found.
      solver  :   t_proc      (avg)   t_wall      (avg)    n_eval
       nlp_f  | 184.00us (  4.28us) 181.26us (  4.22us)        43
       nlp_g  |   2.61ms ( 60.79us)   2.62ms ( 60.86us)        43
    nlp_grad  | 113.00us (113.00us) 112.84us (112.84us)         1
  nlp_grad_f  | 283.00us (  6.43us) 276.32us (  6.28us)        44
  nlp_hess_l  | 350.00us (  8.33us) 342.76us (  8.16us)        42
   nlp_jac_g  |   3.91ms ( 88.80us)   3.93ms ( 89.37us)        44
       total  | 149.56ms (149.56ms) 149.21ms (149.21ms)         1
```



[14]:
```
x, u, t, a = post.get_data()
print(f"Terminal time, state : {t[-1][0]:.4f} vs 4.1641s (Exact), {x[-1]}")
```

```
Terminal time, state : 4.1662 vs 4.1641s (Exact), [3.68621737e-35 0.00000000e+00]
```

4.1.3 Hypersensitive OCP

(c) 2023 Devakumar Thammisetty

MPOPT is an open-source Multi-phase Optimal Control Problem (OCP) solver based on pseudo-spectral collocation with customized adaptive grid refinement techniques.

https://mpopt.readthedocs.io/

Download this notebook: hypersensitive.ipynb

Install mpopt from pypi using the following. Disable after first usage

Import mpopt (Contains main solver modules)

[1]:
```
#!pip install mpopt
from mpopt import mp
```

Defining OCP

Hypersensitive OCP: https://www.gpops2.com/Examples/HyperSensitive.html

$$
\begin{aligned}
\min_{x,u} \quad & J = 0 + \frac{1}{2}\int_{t_0}^{t_f} (x^2 + u^2)\, dt \\
\text{subject to} \quad & \dot{x} = -x^3 + u \\
& x_0(t_0) = 1; t_0 = 0; \\
& x_1(t_f) = 0; t_f = 1000
\end{aligned}
$$

We first create an OCP object and then polulate the object with dynamics, path_constraints, terminal_constraints and objective (running_costs, terminal_costs)

[2]:
```
ocp = mp.OCP(n_states=1, n_controls=1, n_phases=1)
```

[3]:
```
ocp.dynamics[0] = lambda x, u, t: [-x[0] * x[0] * x[0] + u[0]]
ocp.running_costs[0] = lambda x, u, t: 0.5 * (x[0] * x[0] + u[0] * u[0])
ocp.terminal_constraints[0] = lambda xf, tf, x0, t0: [xf[0] - 1.0]
```

Initial state

[4]:
```
ocp.x00[0] = 1
```

Box constraints

[5]:
```
ocp.lbtf[0] = ocp.ubtf[0] = 1000.0
```

Scale the time variable

[6]:
```
ocp.scale_t = 1 / 1000.0
```

[7]:
```
ocp.validate()
```

Solve and plot the results in one line

Lets solve the OCP using following pseudo-spectral approximation * Collocation using Legendre-Gauss-Radau roots * Let's plot the position and velocity evolution with time starting from 0.

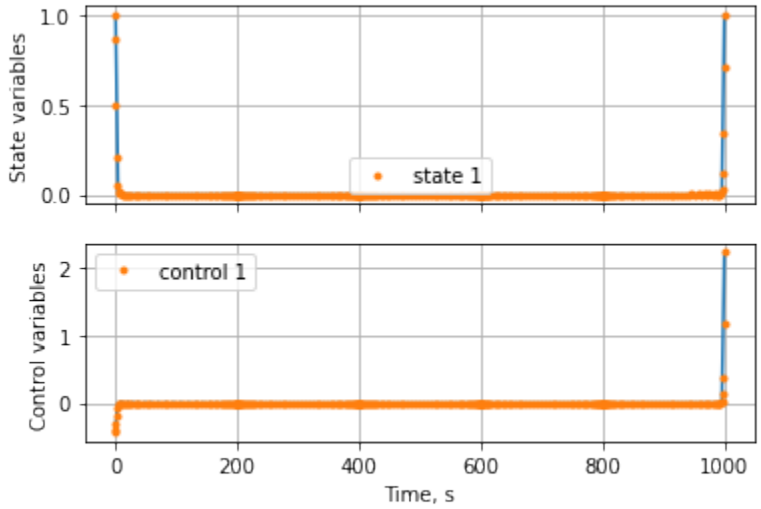
[8]:
```
mpo, post = mp.solve(ocp, n_segments=5, poly_orders=50, scheme="LGR", plot=True)
```

```
******************************************************************************
This program contains Ipopt, a library for large-scale nonlinear optimization.
 Ipopt is released as open source code under the Eclipse Public License (EPL).
         For more information visit http://projects.coin-or.org/Ipopt
******************************************************************************

Total number of variables...:      501
                     variables with only lower bounds:        0
                variables with lower and upper bounds:        0
                     variables with only upper bounds:        0
Total number of equality constraints...:      252
Total number of inequality constraints...:        0
        inequality constraints with only lower bounds:        0
   inequality constraints with lower and upper bounds:        0
        inequality constraints with only upper bounds:        0


Number of Iterations...: 12

                                   (scaled)                 (unscaled)
Objective...:   1.1498050755273090e+00    1.1498050755273090e+00
Dual infeasibility...:   4.7184478546569153e-16    4.7184478546569153e-16
Constraint violation...:   1.4589005542615039e-14    1.1368683772161603e-13
Complementarity...:   0.0000000000000000e+00    0.0000000000000000e+00
Overall NLP error...:   1.4589005542615039e-14    1.1368683772161603e-13


Number of objective function evaluations             = 30
Number of objective gradient evaluations             = 13
Number of equality constraint evaluations            = 42
Number of inequality constraint evaluations          = 0
Number of equality constraint Jacobian evaluations   = 13
Number of inequality constraint Jacobian evaluations = 0
Number of Lagrangian Hessian evaluations             = 12
Total CPU secs in IPOPT (w/o function evaluations)   =      0.066
Total CPU secs in NLP function evaluations           =      0.010

EXIT: Optimal Solution Found.
      solver  :   t_proc      (avg)   t_wall      (avg)    n_eval
       nlp_f  | 340.00us ( 11.33us) 340.41us ( 11.35us)        30
       nlp_g  |   4.85ms (115.38us)   4.78ms (113.76us)        42
    nlp_grad  | 266.00us (266.00us) 265.36us (265.36us)         1
```

(continues on next page)

(continued from previous page)

```
 nlp_grad_f  | 322.00us ( 23.00us) 322.67us ( 23.05us)        14
 nlp_hess_l  | 592.00us ( 49.33us) 595.87us ( 49.66us)        12
  nlp_jac_g  |   2.76ms (197.43us)   2.75ms (196.68us)        14
      total  |  76.82ms ( 76.82ms)  76.14ms ( 76.14ms)         1
```



Retrive the solution

x: states, u: Controls, t:time, a:Algebraic variables

[9]:
```
x, u, t, a = post.get_data()
print(f"Terminal time, state : {t[-1][0]:.4f} vs 1000 (Exact), {x[-1]}")
```

```
Terminal time, state : 1000.0000 vs 1000 (Exact), [1.]
```

Solve again with Chebyshev-Gauss-Lobatto (CGL) roots

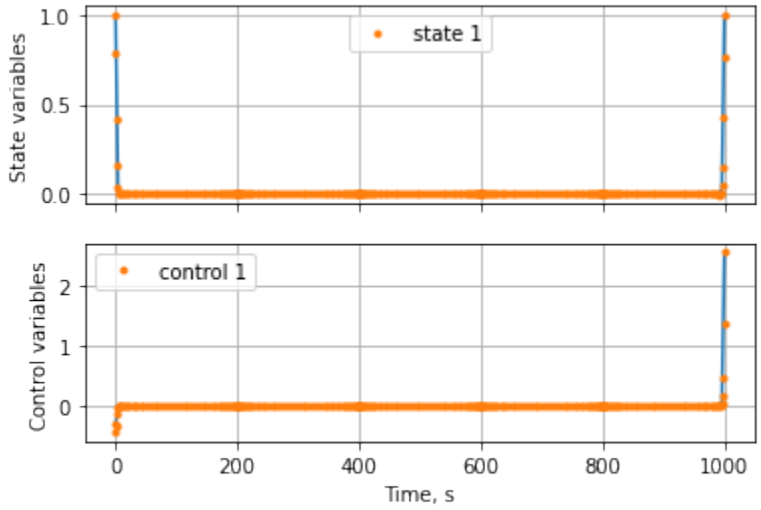
[10]:
```
mpo, post = mp.solve(ocp, n_segments=5, poly_orders=50, scheme="CGL", plot=True)
```

```
Total number of variables...:      501
                     variables with only lower bounds:        0
                variables with lower and upper bounds:        0
                     variables with only upper bounds:        0
Total number of equality constraints...:      252
Total number of inequality constraints...:        0
        inequality constraints with only lower bounds:        0
   inequality constraints with lower and upper bounds:        0
        inequality constraints with only upper bounds:        0


Number of Iterations...: 12

                                   (scaled)                 (unscaled)
Objective...:   1.1406025536022588e+00    1.1406025536022588e+00
Dual infeasibility...:   5.5128124287762148e-13    5.5128124287762148e-13
Constraint violation...:   5.6878507039202523e-12    1.8587797967484221e-11
```

(continues on next page)

(continued from previous page)

```
Complementarity...:   0.0000000000000000e+00    0.0000000000000000e+00
Overall NLP error...:   5.6878507039202523e-12    1.8587797967484221e-11


Number of objective function evaluations             = 36
Number of objective gradient evaluations             = 13
Number of equality constraint evaluations            = 49
Number of inequality constraint evaluations          = 0
Number of equality constraint Jacobian evaluations   = 13
Number of inequality constraint Jacobian evaluations = 0
Number of Lagrangian Hessian evaluations             = 12
Total CPU secs in IPOPT (w/o function evaluations)   =      0.061
Total CPU secs in NLP function evaluations           =      0.010

EXIT: Optimal Solution Found.
      solver  :   t_proc      (avg)   t_wall      (avg)    n_eval
       nlp_f  | 425.00us ( 11.81us) 407.26us ( 11.31us)        36
       nlp_g  |   5.59ms (114.14us)   5.58ms (113.88us)        49
    nlp_grad  | 264.00us (264.00us) 263.05us (263.05us)         1
  nlp_grad_f  | 325.00us ( 23.21us) 325.10us ( 23.22us)        14
  nlp_hess_l  | 614.00us ( 51.17us) 617.11us ( 51.43us)        12
   nlp_jac_g  |   2.82ms (201.79us)   2.81ms (201.06us)        14
       total  |  78.41ms ( 78.41ms)  77.65ms ( 77.65ms)         1
```



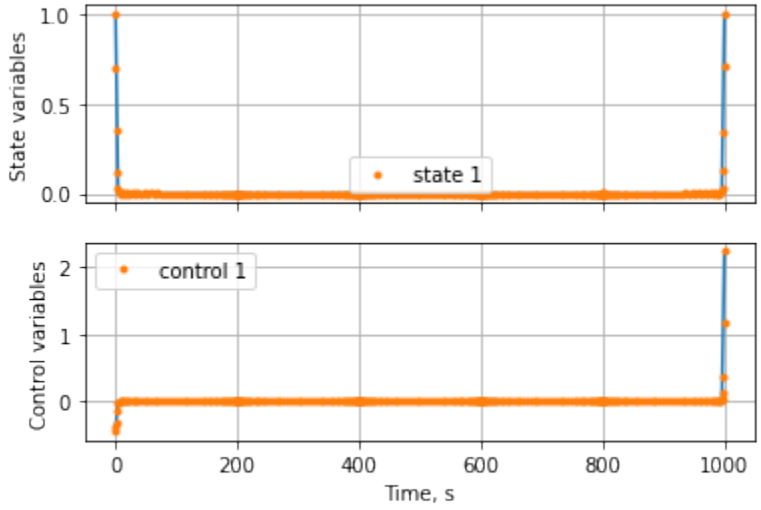
[11]:
```
x, u, t, a = post.get_data()
print(f"Terminal time, state : {t[-1][0]:.4f} vs 1000s (Exact), {x[-1]}")
```

```
Terminal time, state : 1000.0000 vs 1000s (Exact), [1.]
```

Solve again with Legendre-Gauss-Lobatto (LGL) roots

[12]:
```
mpo, post = mp.solve(ocp, n_segments=5, poly_orders=50, scheme="LGL", plot=True)
```

```
Total number of variables...:      501
                     variables with only lower bounds:        0
                variables with lower and upper bounds:        0
                     variables with only upper bounds:        0
Total number of equality constraints...:      252
Total number of inequality constraints...:        0
        inequality constraints with only lower bounds:        0
   inequality constraints with lower and upper bounds:        0
        inequality constraints with only upper bounds:        0


Number of Iterations...: 12

                                   (scaled)                 (unscaled)
Objective...:   1.1502075893651909e+00    1.1502075893651909e+00
Dual infeasibility...:   1.0234087882698972e-13    1.0234087882698972e-13
Constraint violation...:   4.7414571594509346e-13    1.3997691894473974e-12
Complementarity...:   0.0000000000000000e+00    0.0000000000000000e+00
Overall NLP error...:   4.7414571594509346e-13    1.3997691894473974e-12


Number of objective function evaluations             = 36
Number of objective gradient evaluations             = 13
Number of equality constraint evaluations            = 49
Number of inequality constraint evaluations          = 0
Number of equality constraint Jacobian evaluations   = 13
Number of inequality constraint Jacobian evaluations = 0
Number of Lagrangian Hessian evaluations             = 12
Total CPU secs in IPOPT (w/o function evaluations)   =      0.064
Total CPU secs in NLP function evaluations           =      0.011

EXIT: Optimal Solution Found.
      solver  :   t_proc      (avg)   t_wall      (avg)    n_eval
       nlp_f  | 423.00us ( 11.75us) 431.21us ( 11.98us)        36
       nlp_g  |   6.12ms (124.82us)   6.12ms (124.86us)        49
    nlp_grad  | 264.00us (264.00us) 263.12us (263.12us)         1
  nlp_grad_f  | 326.00us ( 23.29us) 327.97us ( 23.43us)        14
  nlp_hess_l  | 611.00us ( 50.92us) 610.94us ( 50.91us)        12
   nlp_jac_g  |   2.82ms (201.07us)   2.82ms (201.54us)        14
       total  |  77.49ms ( 77.49ms)  76.93ms ( 76.93ms)         1
```



[13]:
```
x, u, t, a = post.get_data()
print(f"Terminal time, state : {t[-1][0]:.4f} vs 1000s (Exact), {x[-1]}")
```

```
Terminal time, state : 1000.0000 vs 1000s (Exact), [1.]
```

[ ]:

4.2 Multi-phase

Refer examples folder in GitHub

4.2.1 Two-phase Schwartz OCP

(c) 2023 Devakumar Thammisetty

MPOPT is an open-source Multi-phase Optimal Control Problem (OCP) solver based on pseudo-spectral collocation with customized adaptive grid refinement techniques.

https://mpopt.readthedocs.io/

Download this notebook: twophaseschwartz.ipynb

Install mpopt from pypi using the following. Disable after first usage

Import mpopt (Contains main solver modules)

[1]:
```
#!pip install mpopt
from mpopt import mp
```

Defining OCP

OCP: https://tomopt.com/docs/propt/tomlab_propt123.php

$$
\begin{aligned}
\min_{x,u} \quad & J = 5(x_0(t_f)^2 + x_1(t_f)^2) + \int_{t_0}^{t_f} 0dt \\
\text{subject to} \quad & \dot{x_0} = x_1 \\
& \dot{x_1} = u - 0.1(1 + 2x_0^2)x_1 \\
\textit{Phase 1:} \quad & 1 - 9(x_0 - 1)^2 - \left(\frac{x_1 - 0.4}{0.3}\right)^2 \leq 0 \\
& x_1 \geq -0.8 \\
& -1 \leq u \leq 1 \\
& x_0(t_0) = 1;\ x_1(t_0) = 1; \\
& t_0 = 0;\ t_f = 1 \\
\textit{Phase 2:} \quad & t_0 = 1;\ t_f = 2.9 \\
& x \in \mathbb{R}^2;\ u \in \mathbb{R}
\end{aligned}
$$

We first create an OCP object and then polulate the object with dynamics, path_constraints, terminal_constraints and objective (running_costs, terminal_costs)

[2]:
```
ocp = mp.OCP(n_states=2, n_controls=1, n_phases=2)
```

[3]:
```
# Step-1 : Define dynamics
def dynamics0(x, u, t):
    return [x[1], u[0] - 0.1 * (1.0 + 2.0 * x[0] * x[0]) * x[1]]


ocp.dynamics = [dynamics0, dynamics0]
```

[4]:
```
# Step-2: Add path constraints
def path_constraints0(x, u, t):
    return [
        1.0 - 9.0 * (x[0] - 1) * (x[0] - 1) - (x[1] - 0.4) * (x[1] - 0.4) / (0.3 * 0.3)
    ]


ocp.path_constraints[0] = path_constraints0
```

[5]:
```
# Step-3: Add terminal cost
def terminal_cost1(xf, tf, x0, t0):
    return 5 * (xf[0] * xf[0] + xf[1] * xf[1])


ocp.terminal_costs[1] = terminal_cost1
```

Initial state and Final guess

[6]:
```
ocp.x00[0] = [1, 1]
ocp.x00[1] = [1, 1]
```

(continues on next page)

(continued from previous page)

```
ocp.xf0[0] = [1, 1]
ocp.xf0[1] = [0, 0]
```

Box constraints

[7]:
```
ocp.lbx[0][1] = -0.8
ocp.lbu[0], ocp.ubu[0] = -1, 1
ocp.lbt0[0], ocp.ubt0[0] = 0, 0
ocp.lbtf[0], ocp.ubtf[0] = 1, 1
ocp.lbtf[1], ocp.ubtf[1] = 2.9, 2.9
```

[8]:
```
ocp.validate()
```

Solve and plot the results in one line

Lets solve the OCP using following pseudo-spectral approximation * Collocation using Legendre-Gauss-Radau roots * Let's plot the position and velocity evolution with time starting from 0.

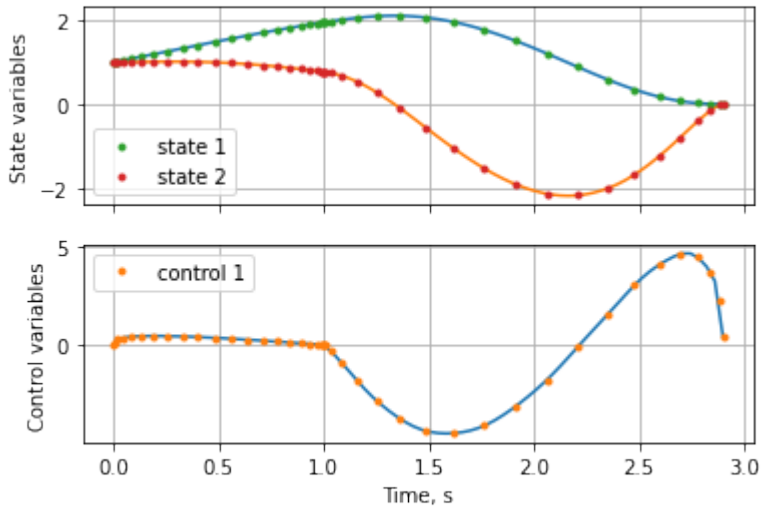
[9]:
```
# ocp.du_continuity[0] = 1
mpo, post = mp.solve(ocp, n_segments=1, poly_orders=20, scheme="LGR", plot=True)
```

```
******************************************************************************
This program contains Ipopt, a library for large-scale nonlinear optimization.
 Ipopt is released as open source code under the Eclipse Public License (EPL).
         For more information visit http://projects.coin-or.org/Ipopt
******************************************************************************

Total number of variables...:      125
                     variables with only lower bounds:       21
                variables with lower and upper bounds:       21
                     variables with only upper bounds:        0
Total number of equality constraints...:       88
Total number of inequality constraints...:       41
        inequality constraints with only lower bounds:        0
   inequality constraints with lower and upper bounds:       20
        inequality constraints with only upper bounds:       21


Number of Iterations...: 9

                                   (scaled)                 (unscaled)
Objective...:   6.4094433643382967e-22    6.4094433643382967e-22
Dual infeasibility...:   4.9116413862415506e-11    4.9116413862415506e-11
Constraint violation...:   1.5082379789532752e-12    1.5181189638724391e-12
Complementarity...:   2.5210268504311982e-09    2.5210268504311982e-09
Overall NLP error...:   2.5210268504311982e-09    2.5210268504311982e-09


Number of objective function evaluations             = 11
Number of objective gradient evaluations             = 10
```

(continues on next page)

(continued from previous page)

```
Number of equality constraint evaluations            = 11
Number of inequality constraint evaluations          = 11
Number of equality constraint Jacobian evaluations   = 10
Number of inequality constraint Jacobian evaluations = 10
Number of Lagrangian Hessian evaluations             = 9
Total CPU secs in IPOPT (w/o function evaluations)   =      0.022
Total CPU secs in NLP function evaluations           =      0.001

EXIT: Optimal Solution Found.
      solver  :   t_proc      (avg)   t_wall      (avg)    n_eval
       nlp_f  |  35.00us (  3.18us)  35.40us (  3.22us)        11
       nlp_g  | 335.00us ( 30.45us) 333.19us ( 30.29us)        11
    nlp_grad  |  57.00us ( 57.00us)  56.50us ( 56.50us)         1
  nlp_grad_f  |  47.00us (  4.27us)  44.33us (  4.03us)        11
  nlp_hess_l  | 122.00us ( 13.56us) 123.24us ( 13.69us)         9
   nlp_jac_g  | 523.00us ( 47.55us) 511.58us ( 46.51us)        11
       total  |  29.37ms ( 29.37ms)  28.73ms ( 28.73ms)         1
```



Retrive the solution

x: states, u: Controls, t:time, a:Algebraic variables

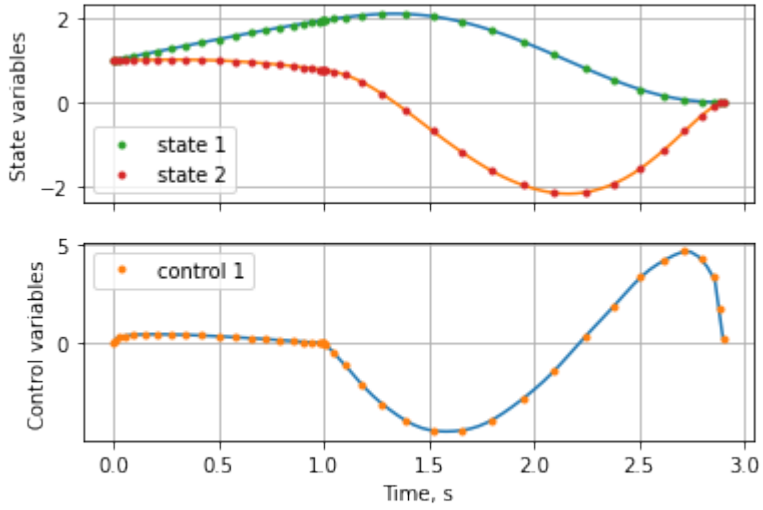
[10]:
```
x, u, t, a = post.get_data()
print(f"Terminal time, state : {t[-1][0]:.4f} vs 2.9 (Exact), {x[-1]}")
```

```
Terminal time, state : 2.9000 vs 2.9 (Exact), [ 9.53605506e-12 -6.10348435e-12]
```

Solve again with Chebyshev-Gauss-Lobatto (CGL) roots

[11]:
```
mpo, post = mp.solve(ocp, n_segments=1, poly_orders=20, scheme="CGL", plot=True)
```

```
Total number of variables...:      125
                     variables with only lower bounds:       21
                variables with lower and upper bounds:       21
                     variables with only upper bounds:        0
Total number of equality constraints...:       88
Total number of inequality constraints...:       41
        inequality constraints with only lower bounds:        0
   inequality constraints with lower and upper bounds:       20
        inequality constraints with only upper bounds:       21


Number of Iterations...: 9

                                   (scaled)                 (unscaled)
Objective...:   6.7496749138057129e-22    6.7496749138057129e-22
Dual infeasibility...:   5.0501516418909649e-11    5.0501516418909649e-11
Constraint violation...:   1.5804024755539103e-12    1.5804024755539103e-12
Complementarity...:   2.5216728438899515e-09    2.5216728438899515e-09
Overall NLP error...:   2.5216728438899515e-09    2.5216728438899515e-09


Number of objective function evaluations             = 11
Number of objective gradient evaluations             = 10
Number of equality constraint evaluations            = 11
Number of inequality constraint evaluations          = 11
Number of equality constraint Jacobian evaluations   = 10
Number of inequality constraint Jacobian evaluations = 10
Number of Lagrangian Hessian evaluations             = 9
Total CPU secs in IPOPT (w/o function evaluations)   =      0.024
Total CPU secs in NLP function evaluations           =      0.002

EXIT: Optimal Solution Found.
      solver  :   t_proc      (avg)   t_wall      (avg)    n_eval
       nlp_f  |  37.00us (  3.36us)  35.08us (  3.19us)        11
       nlp_g  | 338.00us ( 30.73us) 335.83us ( 30.53us)        11
    nlp_grad  |  58.00us ( 58.00us)  56.75us ( 56.75us)         1
  nlp_grad_f  |  50.00us (  4.55us)  47.37us (  4.31us)        11
  nlp_hess_l  | 127.00us ( 14.11us) 125.98us ( 14.00us)         9
   nlp_jac_g  | 515.00us ( 46.82us) 519.92us ( 47.27us)        11
       total  |  27.65ms ( 27.65ms)  27.21ms ( 27.21ms)         1
```



[12]:
```
x, u, t, a = post.get_data()
print(f"Terminal time, state : {t[-1][0]:.4f} vs 2.9s (Exact), {x[-1]}")
```

```
Terminal time, state : 2.9000 vs 2.9s (Exact), [ 9.78414149e-12 -6.26610513e-12]
```

Solve again with Legendre-Gauss-Lobatto (LGL) roots

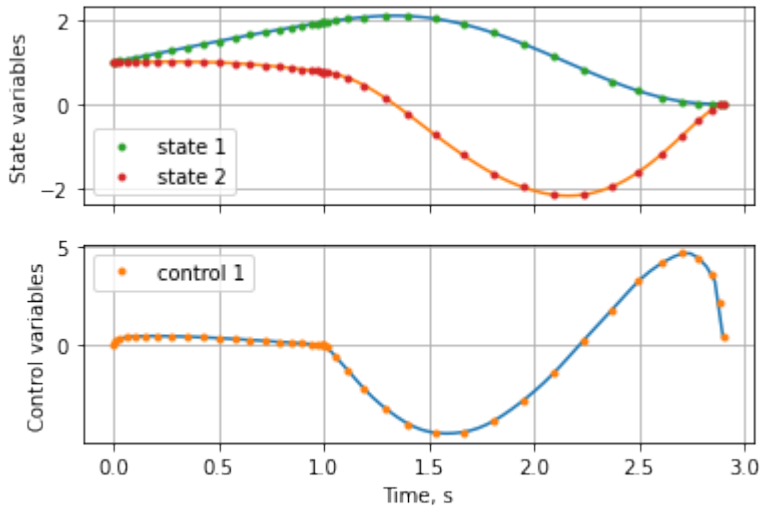
[13]:
```
mpo, post = mp.solve(ocp, n_segments=1, poly_orders=20, scheme="LGL", plot=True)
```

```
Total number of variables...:      125
                     variables with only lower bounds:       21
                variables with lower and upper bounds:       21
                     variables with only upper bounds:        0
Total number of equality constraints...:       88
Total number of inequality constraints...:       41
        inequality constraints with only lower bounds:        0
   inequality constraints with lower and upper bounds:       20
        inequality constraints with only upper bounds:       21


Number of Iterations...: 9

                                   (scaled)                 (unscaled)
Objective...:   6.7267963401089627e-22    6.7267963401089627e-22
Dual infeasibility...:   4.9171301510338480e-11    4.9171301510338480e-11
Constraint violation...:   1.5127898933542383e-12    1.5127898933542383e-12
Complementarity...:   2.5210877471234159e-09    2.5210877471234159e-09
Overall NLP error...:   2.5210877471234159e-09    2.5210877471234159e-09


Number of objective function evaluations             = 11
Number of objective gradient evaluations             = 10
Number of equality constraint evaluations            = 11
Number of inequality constraint evaluations          = 11
```

(continues on next page)

(continued from previous page)

```
Number of equality constraint Jacobian evaluations   = 10
Number of inequality constraint Jacobian evaluations = 10
Number of Lagrangian Hessian evaluations             = 9
Total CPU secs in IPOPT (w/o function evaluations)   =      0.021
Total CPU secs in NLP function evaluations           =      0.001

EXIT: Optimal Solution Found.
      solver  :   t_proc      (avg)   t_wall      (avg)    n_eval
       nlp_f  |  35.00us (  3.18us)  33.39us (  3.04us)        11
       nlp_g  | 334.00us ( 30.36us) 332.27us ( 30.21us)        11
    nlp_grad  |  57.00us ( 57.00us)  56.27us ( 56.27us)         1
  nlp_grad_f  |  50.00us (  4.55us)  46.60us (  4.24us)        11
  nlp_hess_l  | 123.00us ( 13.67us) 123.49us ( 13.72us)         9
   nlp_jac_g  | 526.00us ( 47.82us) 532.02us ( 48.37us)        11
       total  |  27.35ms ( 27.35ms)  26.99ms ( 26.99ms)         1
```



[14]:
```
x, u, t, a = post.get_data()
print(f"Terminal time, state : {t[-1][0]:.4f} vs 2.9s (Exact), {x[-1]}")
```

```
Terminal time, state : 2.9000 vs 2.9s (Exact), [ 9.76831449e-12 -6.25427524e-12]
```

4.2.2 Multi-stage Launch Vehicle (Delta III) Ascent to Orbit

(c) 2023 Devakumar Thammisetty

MPOPT is an open-source Multi-phase Optimal Control Problem (OCP) solver based on pseudo-spectral collocation with customized adaptive grid refinement techniques.

https://mpopt.readthedocs.io/

Download this notebook: multi_stage_launch_vehicle_ascent.ipynb

Install mpopt from pypi using the following. Disable after first usage

Import mpopt (Contains main solver modules)

[1]:
```
#!pip install mpopt
from mpopt import mp
import numpy as np
import casadi as ca
```

Defining OCP

OCP: http://dx.doi.org/10.13140/RG.2.2.19519.79528

$$\min_{x,u} \quad J = -x_6(t_f) + \int_{t_0}^{t_f} 0dt$$

subject to

Define
$\mathbf{r} = [x_0, x_1, x_2];\ \mathbf{v} = [x_3, x_4, x_5];\ m = x_6;\ \mathbf{u} = [u_0, u_1, u_2];$
$\mu = 3.986012e14;\ \omega_e = [0, 0, 7.29211585e-5]^T;\ R_e = 6378145;$
$\rho_0 = 1.225;\ H_0 = 7200;\ C_d = 0.5;\ A^{\text{ref}} = 4\pi;$
$T^0 = 4854100;\ T^1 = 2968600;\ T^2 = 1083100; T^3 = 110094;$
$\dot{m}^0 = 1723.273;\ \dot{m}^1 = 1044.682;\ \dot{m}^2 = 366.092;\ \dot{m}^3 = 24.029;$

Dynamics:
$\dot{\mathbf{r}} = \mathbf{v}$
$$\dot{\mathbf{v}} = -\frac{\mu}{|\mathbf{r}|^3}\mathbf{r} + \frac{T^p}{m}\mathbf{u} - \frac{C_d A^{\text{ref}} \rho_0 |\mathbf{v} - (\omega_e \times \mathbf{r})|(\mathbf{v} - (\omega_e \times \mathbf{r}))e^{\frac{|\mathbf{r}|-R_e}{H_0}}}{2m}$$
$\dot{m} = -\dot{m}^p \qquad p = 0, 1, 2, 3$

Path constraints:
$|\mathbf{u}| = 1$
$|\mathbf{r}| \geq R_e$

Terminal constraints:
$t_0^0 = 0;\ t_0^1 = 75.2;\ t_0^2 = 150.4;\ t_0^3 = 261$
$t_f^0 = 75.2;\ t_f^1 = 150.4;\ t_f^2 = 261$
Let $\mathbf{r_f} = [x_0(t_f^3), x_1(t_f^3), x_2(t_f^3)];\ \mathbf{v_f} = [x_3(t_f^3), x_4(t_f^3), x_5(t_f^3)];$
$a_f(\mathbf{r_f}, \mathbf{v_f}) = 24361140; e_f(\mathbf{r_f}, \mathbf{v_f}) = 0.7308;$
$i_f(\mathbf{r_f}, \mathbf{v_f}) = \frac{28.5\pi}{180}; \Omega_f(\mathbf{r_f}, \mathbf{v_f}) = \frac{269.8\pi}{180};$
$\omega_f(\mathbf{r_f}, \mathbf{v_f}) = \frac{130.5\pi}{180};$
Where $a_f, e_f, i_f, \Omega_f, \omega_f$ are computed as follows
$\mathbf{h} = \mathbf{r_f} \times \mathbf{v_f};\ \mathbf{n} = [0, 0, 1]^T \times \mathbf{v_f};$
$\mathbf{e} = \frac{1}{\mu}\mathbf{v_f} \times \mathbf{h} - \frac{\mathbf{r_f}}{|\mathbf{r_f}|};$
$e_f = |\mathbf{e}|;\ a_f = -\frac{\mu}{|\mathbf{v_f}|^2 - \frac{2\mu}{|\mathbf{r_f}|}};\ i_f = \cos^{-1}\left(\frac{\mathbf{h}.[0,0,1]^T}{|\mathbf{h}|}\right);$
if $\mathbf{n}[1]$>0: $\Omega_f = \cos^{-1}\left(\frac{\mathbf{n}.[0,0,1]^T}{|\mathbf{n}|}\right)$ else: $\Omega_f = (2\pi - \Omega_f);$
if e[2]>0: $\omega_f = \cos^{-1}\left(\frac{\mathbf{n}.\mathbf{e}}{|\mathbf{n}|e_f}\right)$ else: $\omega_f = (2\pi - \omega_f);$

Event constraints:
$\mathbf{r}(t_f^p) - \mathbf{r}(t_0^{p+1}) = 0;\ \mathbf{v}(t_f^p) - \mathbf{v}(t_0^{p+1}) = 0; \quad p = 0, 1, 2$
$m(t_f^p) - m(t_0^{p+1}) = [13680, 6840, 8830] \quad p = 0, 1, 2$

Initial conditions:
$\mathbf{r}(t_0^0) = [5605222.973, 0, 3043387.761]; \mathbf{v}(t_0^0) = [0, 408.74, 0];$
$m(t_0^0) = 301454$

We first create an OCP object and then polulate the object with dynamics, path_constraints, terminal_constraints and objective (running_costs, terminal_costs)

[2]:
```
ocp = mp.OCP(n_states=7, n_controls=3, n_phases=4)
```

[3]:
```
# Initialize parameters
Re = 6378145.0  # m
```

(continues on next page)

(continued from previous page)

```
omegaE = 7.29211585e-5
rho0 = 1.225
rhoH = 7200.0
Sa = 4 * np.pi
Cd = 0.5
muE = 3.986012e14
g0 = 9.80665

# Variable initialization
lat0 = 28.5 * np.pi / 180.0
r0 = np.array([Re * np.cos(lat0), 0.0, Re * np.sin(lat0)])
v0 = omegaE * np.array([-r0[1], r0[0], 0.0])
m0 = 301454.0
mf = 4164.0
mdrySrb = 19290.0 - 17010.0
mdryFirst = 104380.0 - 95550.0
mdrySecond = 19300.0 - 16820.0
x0 = np.array([r0[0], r0[1], r0[2], v0[0], v0[1], v0[2], m0])
```

[4]:
```
# Step-1 : Define dynamics
# Thrust(N) and mass flow rate(kg/s) in each stage
Thrust = [6 * 628500.0 + 1083100.0, 3 * 628500.0 + 1083100.0, 1083100.0, 110094.0]
mdot = [
    (6 * 17010.0) / (75.2) + (95550.0) / (261.0),
    (3 * 17010.0) / (75.2) + (95550.0) / (261.0),
    (95550.0) / (261.0),
    16820.0 / 700.0,
]


def dynamics(x, u, t, param=0, T=0.0, mdot=0.0):
    r = x[:3]
    v = x[3:6]
    m = x[6]
    r_mag = ca.sqrt(r[0] * r[0] + r[1] * r[1] + r[2] * r[2])
    v_rel = ca.vertcat(v[0] + r[1] * omegaE, v[1] - r[0] * omegaE, v[2])
    v_rel_mag = ca.sqrt(v_rel[0] * v_rel[0] + v_rel[1] * v_rel[1] + v_rel[2] * v_rel[2])
    h = r_mag - Re
    rho = rho0 * ca.exp(-h / rhoH)
    D = -rho / (2 * m) * Sa * Cd * v_rel_mag * v_rel
    g = -muE / (r_mag * r_mag * r_mag) * r

    xdot = [
        x[3],
        x[4],
        x[5],
        T / m * u[0] + param * D[0] + g[0],
        T / m * u[1] + param * D[1] + g[1],
        T / m * u[2] + param * D[2] + g[2],
        -mdot,
    ]
    return xdot
```

(continues on next page)

(continued from previous page)

```
def get_dynamics(param):
    dynamics0 = lambda x, u, t: dynamics(
        x, u, t, param=param, T=Thrust[0], mdot=mdot[0]
    )
    dynamics1 = lambda x, u, t: dynamics(
        x, u, t, param=param, T=Thrust[1], mdot=mdot[1]
    )
    dynamics2 = lambda x, u, t: dynamics(
        x, u, t, param=param, T=Thrust[2], mdot=mdot[2]
    )
    dynamics3 = lambda x, u, t: dynamics(
        x, u, t, param=param, T=Thrust[3], mdot=mdot[3]
    )

    return [dynamics0, dynamics1, dynamics2, dynamics3]


ocp.dynamics = get_dynamics(0)
```

[5]:
```
# Step-2: Add path constraints
def path_constraints0(x, u, t):
    return [
        u[0] * u[0] + u[1] * u[1] + u[2] * u[2] - 1,
        -u[0] * u[0] - u[1] * u[1] - u[2] * u[2] + 1,
        -ca.sqrt(x[0] * x[0] + x[1] * x[1] + x[2] * x[2]) / Re + 1,
    ]


ocp.path_constraints = [path_constraints0] * ocp.n_phases
```

[6]:
```
# Step-3: Add terminal cost and constraints
def terminal_cost3(xf, tf, x0, t0):
    return -xf[-1] / m0


ocp.terminal_costs[3] = terminal_cost3


def terminal_constraints3(x, t, x0, t0):
    # https://space.stackexchange.com/questions/1904/how-to-programmatically-calculate-
↪orbital-elements-using-position-velocity-vecto
    # http://control.asu.edu/Classes/MAE462/462Lecture06.pdf
    h = ca.vertcat(
        x[1] * x[5] - x[4] * x[2], x[3] * x[2] - x[0] * x[5], x[0] * x[4] - x[1] * x[3]
    )

    n = ca.vertcat(-h[1], h[0], 0)
    r = ca.sqrt(x[0] * x[0] + x[1] * x[1] + x[2] * x[2])
```

(continues on next page)

(continued from previous page)

```
    e = ca.vertcat(
        1 / muE * (x[4] * h[2] - x[5] * h[1]) - x[0] / r,
        1 / muE * (x[5] * h[0] - x[3] * h[2]) - x[1] / r,
        1 / muE * (x[3] * h[1] - x[4] * h[0]) - x[2] / r,
    )

    e_mag = ca.sqrt(e[0] * e[0] + e[1] * e[1] + e[2] * e[2])
    h_sq = h[0] * h[0] + h[1] * h[1] + h[2] * h[2]
    v_mag = ca.sqrt(x[3] * x[3] + x[4] * x[4] + x[5] * x[5])

    a = -muE / (v_mag * v_mag - 2.0 * muE / r)
    i = ca.acos(h[2] / ca.sqrt(h_sq))
    n_mag = ca.sqrt(n[0] * n[0] + n[1] * n[1])

    node_asc = ca.acos(n[0] / n_mag)
    # if n[1] < -1e-12:
    node_asc = 2 * np.pi - node_asc

    argP = ca.acos((n[0] * e[0] + n[1] * e[1]) / (n_mag * e_mag))
    # if e[2] < 0:
    #    argP = 2*np.pi - argP

    a_req = 24361140.0
    e_req = 0.7308
    i_req = 28.5 * np.pi / 180.0
    node_asc_req = 269.8 * np.pi / 180.0
    argP_req = 130.5 * np.pi / 180.0

    return [
        (a - a_req) / (Re),
        e_mag - e_req,
        i - i_req,
        node_asc - node_asc_req,
        argP - argP_req,
    ]


ocp.terminal_constraints[3] = terminal_constraints3
```

Scale the variables

[7]:
```
ocp.scale_x = [
    1 / Re,
    1 / Re,
    1 / Re,
    1 / np.sqrt(muE / Re),
    1 / np.sqrt(muE / Re),
    1 / np.sqrt(muE / Re),
    1 / m0,
]
ocp.scale_t = np.sqrt(muE / Re) / Re
```

Initial state and Final guess

[8]:
```
# Intial guess
# Initial guess estimation
def ae_to_rv(a, e, i, node, argP, th):
    p = a * (1.0 - e * e)
    r = p / (1.0 + e * np.cos(th))

    r_vec = np.array([r * np.cos(th), r * np.sin(th), 0.0])
    v_vec = np.sqrt(muE / p) * np.array([-np.sin(th), e + np.cos(th), 0.0])

    cn, sn = np.cos(node), np.sin(node)
    cp, sp = np.cos(argP), np.sin(argP)
    ci, si = np.cos(i), np.sin(i)

    R = np.array(
        [
            [cn * cp - sn * sp * ci, -cn * sp - sn * cp * ci, sn * si],
            [sn * cp + cn * sp * ci, -sn * sp + cn * cp * ci, -cn * si],
            [sp * si, cp * si, ci],
        ]
    )

    r_i = np.dot(R, r_vec)
    v_i = np.dot(R, v_vec)

    return r_i, v_i


# Target conditions
a_req = 24361140.0
e_req = 0.7308
i_req = 28.5 * np.pi / 180.0
node_asc_req = 269.8 * np.pi / 180.0
argP_req = 130.5 * np.pi / 180.0
th = 0.0
rf, vf = ae_to_rv(a_req, e_req, i_req, node_asc_req, argP_req, th)
```

Intial guess

[9]:
```
# Timings
t0, t1, t2, t3, t4 = 0.0, 75.2, 150.4, 261.0, 924.0
# Interpolate to get starting values for intermediate phases
xf = np.array([rf[0], rf[1], rf[2], vf[0], vf[1], vf[2], mf + mdrySecond])
x1 = x0 + (xf - x0) / (t4 - t0) * (t1 - t0)
x2 = x0 + (xf - x0) / (t4 - t0) * (t2 - t0)
x3 = x0 + (xf - x0) / (t4 - t0) * (t3 - t0)

# Update the state discontinuity values across phases
x0f = np.copy(x1)
x0f[-1] = x0[-1] - (6 * 17010.0 + 95550.0 / t3 * t1)
x1[-1] = x0f[-1] - 6 * mdrySrb

x1f = np.copy(x2)
x1f[-1] = x1[-1] - (3 * 17010.0 + 95550.0 / t3 * (t2 - t1))
```

(continues on next page)

(continued from previous page)

```
x2[-1] = x1f[-1] - 3 * mdrySrb

x2f = np.copy(x3)
x2f[-1] = x2[-1] - (95550.0 / t3 * (t3 - t2))
x3[-1] = x2f[-1] - mdryFirst

# Step-8b: Initial guess for the states, controls and phase start and final
#             times
ocp.x00 = np.array([x0, x1, x2, x3])
ocp.xf0 = np.array([x0f, x1f, x2f, xf])
ocp.u00 = np.array([[1, 0, 0], [1, 0, 0], [0, 1, 0], [0, 1, 0]])
ocp.uf0 = np.array([[0, 1, 0], [0, 1, 0], [0, 1, 0], [0, 1, 0]])
ocp.t00 = np.array([[t0], [t1], [t2], [t3]])
ocp.tf0 = np.array([[t1], [t2], [t3], [t4]])
```

Box constraints

[10]:
```
rmin, rmax = -2 * Re, 2 * Re
vmin, vmax = -10000.0, 10000.0
rvmin = [rmin, rmin, rmin, vmin, vmin, vmin]
rvmax = [rmax, rmax, rmax, vmax, vmax, vmax]
lbx0 = rvmin + [x0f[-1]]
lbx1 = rvmin + [x1f[-1]]
lbx2 = rvmin + [x2f[-1]]
lbx3 = rvmin + [xf[-1]]
ubx0 = rvmax + [x0[-1]]
ubx1 = rvmax + [x1[-1]]
ubx2 = rvmax + [x2[-1]]
ubx3 = rvmax + [x3[-1]]
ocp.lbx = np.array([lbx0, lbx1, lbx2, lbx3])
ocp.ubx = np.array([ubx0, ubx1, ubx2, ubx3])

# Bounds for control inputs
umin = [-1, -1, -1]
umax = [1, 1, 1]
ocp.lbu = np.array([umin] * ocp.n_phases)
ocp.ubu = np.array([umax] * ocp.n_phases)

# Bounds for phase start and final times
ocp.lbt0 = np.array([[t0], [t1], [t2], [t3]])
ocp.ubt0 = np.array([[t0], [t1], [t2], [t3]])
ocp.lbtf = np.array([[t1], [t2], [t3], [t4 - 100]])
ocp.ubtf = np.array([[t1], [t2], [t3], [t4 + 100]])

# Event constraint bounds on states : State continuity/disc.
lbe0 = [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, -6 * mdrySrb]
lbe1 = [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, -3 * mdrySrb]
lbe2 = [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, -mdryFirst]
ocp.lbe = np.array([lbe0, lbe1, lbe2])
ocp.ube = np.array([lbe0, lbe1, lbe2])
```

[11]:
```
ocp.validate()
```

Solve with atmospheric drag disabled

[12]:
```
ocp.dynamics = get_dynamics(0)
mpo = mp.mpopt(ocp, 1, 11)
sol = mpo.solve()
```

```
******************************************************************************
This program contains Ipopt, a library for large-scale nonlinear optimization.
 Ipopt is released as open source code under the Eclipse Public License (EPL).
         For more information visit http://projects.coin-or.org/Ipopt
******************************************************************************

Total number of variables...:      474
                     variables with only lower bounds:        0
                variables with lower and upper bounds:      474
                     variables with only upper bounds:        0
Total number of equality constraints...:      374
Total number of inequality constraints...:      276
        inequality constraints with only lower bounds:        0
   inequality constraints with lower and upper bounds:      132
        inequality constraints with only upper bounds:      144


Number of Iterations...: 320

                                   (scaled)                 (unscaled)
Objective...:  -2.722244439176720e-02   -2.722244439176720e-02
Dual infeasibility...:   1.5901114178071890e-09    1.5901114178071890e-09
Constraint violation...:   1.9999240413043351e-08    1.9999240413043351e-08
Complementarity...:   2.5890735722630404e-09    2.5890735722630404e-09
Overall NLP error...:   1.9999240413043351e-08    1.9999240413043351e-08


Number of objective function evaluations             = 806
Number of objective gradient evaluations             = 312
Number of equality constraint evaluations            = 807
Number of inequality constraint evaluations          = 807
Number of equality constraint Jacobian evaluations   = 323
Number of inequality constraint Jacobian evaluations = 323
Number of Lagrangian Hessian evaluations             = 321
Total CPU secs in IPOPT (w/o function evaluations)   =      4.019
Total CPU secs in NLP function evaluations           =      0.182

EXIT: Solved To Acceptable Level.
      solver  :   t_proc      (avg)   t_wall      (avg)    n_eval
       nlp_f  |   2.18ms (  2.71us)   2.15ms (  2.66us)       806
       nlp_g  |  62.39ms ( 77.31us)  61.96ms ( 76.78us)       807
    nlp_grad  | 161.00us (161.00us) 159.72us (159.72us)         1
  nlp_grad_f  |   1.81ms (  5.78us)   1.76ms (  5.63us)       313
  nlp_hess_l  |  37.45ms (117.03us)  37.34ms (116.69us)       320
   nlp_jac_g  |  51.28ms (158.28us)  51.17ms (157.94us)       324
       total  |   4.24 s (  4.24 s)   4.22 s (  4.22 s)         1
```

Solve the problem with drag enabled now with revised initial guess

[13]:
```
ocp.dynamics = get_dynamics(1)
ocp.validate()

mpo._ocp = ocp
sol = mpo.solve(
    sol, reinitialize_nlp=True, nlp_solver_options={"ipopt.acceptable_tol": 1e-6}
)
```

```
Total number of variables...:      474
                     variables with only lower bounds:        0
                variables with lower and upper bounds:      474
                     variables with only upper bounds:        0
Total number of equality constraints...:      374
Total number of inequality constraints...:      276
        inequality constraints with only lower bounds:        0
   inequality constraints with lower and upper bounds:      132
        inequality constraints with only upper bounds:      144


Number of Iterations...: 119

                                   (scaled)                 (unscaled)
Objective...:  -2.497798107538465e-02   -2.497798107538465e-02
Dual infeasibility...:   1.2976399535433416e-08    1.2976399535433416e-08
Constraint violation...:   6.8977004524795049e-09    6.8977004524795049e-09
Complementarity...:   2.5143327708305730e-09    2.5143327708305730e-09
Overall NLP error...:   6.8977004524795049e-09    1.2976399535433416e-08


Number of objective function evaluations             = 262
Number of objective gradient evaluations             = 120
Number of equality constraint evaluations            = 262
Number of inequality constraint evaluations          = 262
Number of equality constraint Jacobian evaluations   = 120
Number of inequality constraint Jacobian evaluations = 120
Number of Lagrangian Hessian evaluations             = 119
Total CPU secs in IPOPT (w/o function evaluations)   =      1.941
Total CPU secs in NLP function evaluations           =      0.106

EXIT: Optimal Solution Found.
      solver  :   t_proc      (avg)   t_wall      (avg)    n_eval
       nlp_f  | 801.00us (  3.06us) 779.38us (  2.97us)       262
       nlp_g  |  26.59ms (101.48us)  26.88ms (102.59us)       262
    nlp_grad  | 183.00us (183.00us) 248.72us (248.72us)         1
  nlp_grad_f  | 786.00us (  6.50us) 762.01us (  6.30us)       121
  nlp_hess_l  |  36.99ms (310.85us)  37.56ms (315.60us)       119
   nlp_jac_g  |  31.55ms (260.72us)  32.24ms (266.46us)       121
       total  |   2.07 s (  2.07 s)   2.10 s (  2.10 s)         1
```

Retrive data from the solution

[14]:
```
post = mpo.process_results(sol, plot=False, scaling=False)
```

(continues on next page)

(continued from previous page)

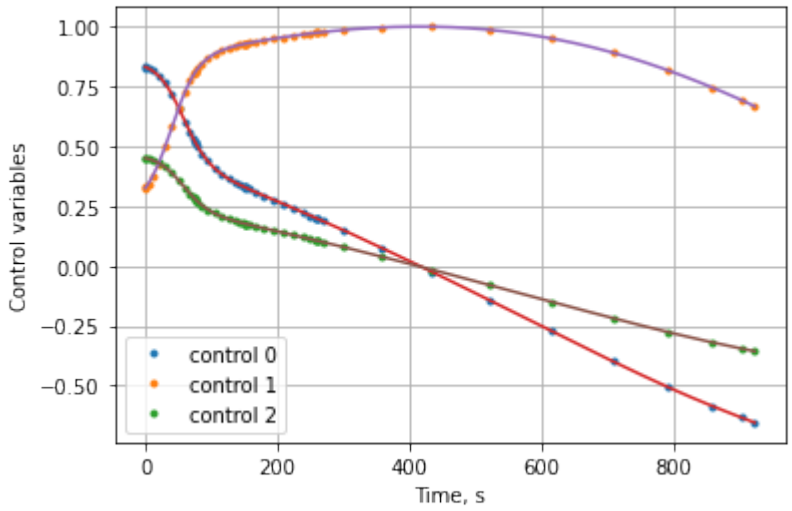
```
mp.post_process._INTERPOLATION_NODES_PER_SEG = 200
x, u, t, _ = post.get_data(interpolate=True)
print("Final time : ", round(t[-1], 4), "(MPOPT) vs 924.1413s(PSOPT)")
print("Final mass : ", round(-sol["f"].full()[0, 0] * m0, 4), "(MPOPT) vs 7529.7123kg␣
↪(GPOPS-II)")
```

```
Final time :  924.1393 (MPOPT) vs 924.1413s(PSOPT)
Final mass :  7529.7123 (MPOPT) vs 7529.7123kg (GPOPS-II)
```

Plot results

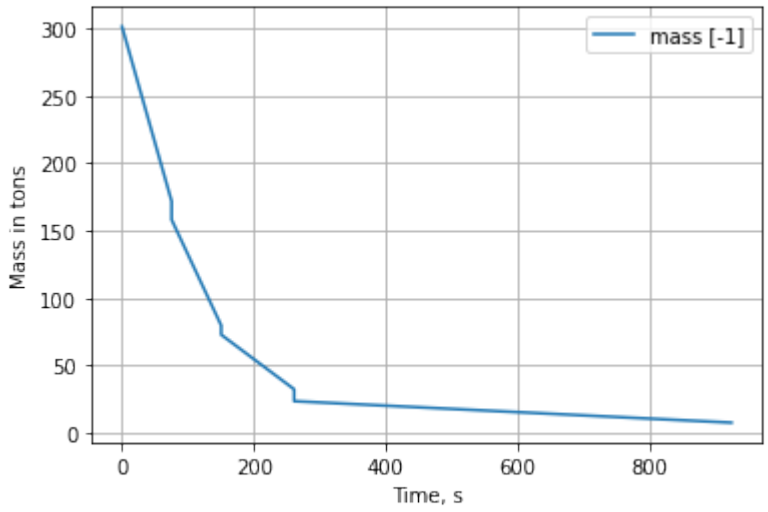
plot steering data (Thrust vector)

[15]:
```
figu, axsu = post.plot_u()
```



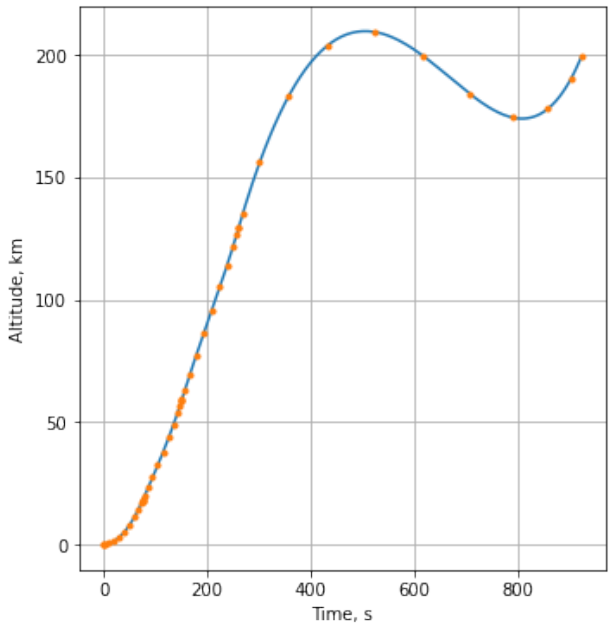
plot mass of the vehicle

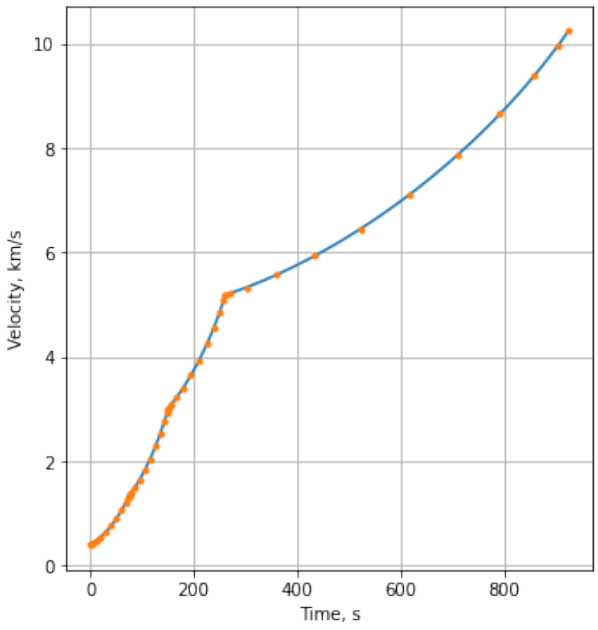
[16]:
```
# Plot mass
figm, axsm = post.plot_single_variable(
    x * 1e-3,
    t,
    [[-1]],
    axis=0,
    fig=None,
    axs=None,
    tics=["-"] * 15,
    name="mass",
    ylabel="Mass in tons",
)
```



Plot height and velocity profile

[17]:
```
mp.plt.rcParams["figure.figsize"] = (12,6)
# Compute and plot altitude, velocity
r = 1e-3 * (np.sqrt(x[:, 0] ** 2 + x[:, 1] ** 2 + x[:, 2] ** 2) - Re)
v = 1e-3 * np.sqrt(x[:, 3] ** 2 + x[:, 4] ** 2 + x[:, 5] ** 2)
y = np.column_stack((r, v))
fig, axs = post.plot_single_variable(y, t, [[0], [1]], axis=0)

x, u, t, _ = post.get_data(interpolate=False)
r = 1e-3 * (np.sqrt(x[:, 0] ** 2 + x[:, 1] ** 2 + x[:, 2] ** 2) - Re)
v = 1e-3 * np.sqrt(x[:, 3] ** 2 + x[:, 4] ** 2 + x[:, 5] ** 2)
y = np.column_stack((r, v))
fig, axs = post.plot_single_variable(
    y, t, [[0], [1]], axis=0, fig=fig, axs=axs, tics=["."] * 15
)
axs[0].set(ylabel="Altitude, km", xlabel="Time, s")
axs[1].set(ylabel="Velocity, km/s", xlabel="Time, s")

# Plot mass at the collocation nodes
figm, axsm = post.plot_single_variable(
    x * 1e-3, t, [[-1]], axis=0, fig=figm, axs=axsm, tics=["."] * 15
)
```





[ ]:

4.2.3 SpaceX Falcon9 Launch Trajectory (Booster Recovery)

(c) 2023 Devakumar Thammisetty

MPOPT is an open-source Multi-phase Optimal Control Problem (OCP) solver based on pseudo-spectral collocation with customized adaptive grid refinement techniques.

https://mpopt.readthedocs.io/

Download this notebook: falcon9_to_orbit.ipynb

Install mpopt from pypi using the following. Disable after first usage

Import mpopt (Contains main solver modules)

[1]:
```
#!pip install mpopt
from mpopt import mp
import numpy as np
import casadi as ca
```

Defining OCP

OCP: (Multi stage launch vehicle OCP : Delta-III rocket) http://dx.doi.org/10.13140/RG.2.2.19519.79528

$$\min_{x,u} \quad J = -x_6(t_f) + \int_{t_0}^{t_f} 0dt$$

subject to

Define

$\mathbf{r} = [x_0, x_1, x_2];\ \mathbf{v} = [x_3, x_4, x_5];\ m = x_6;\ \mathbf{u} = [u_0, u_1, u_2];$

$\mu = 3.986012e14;\ \omega_e = [0, 0, 7.29211585e-5]^T;\ R_e = 6378145;$

$\rho_0 = 1.225;\ H_0 = 7200;\ C_d = 0.5;\ A^{\text{ref}} = 4\pi;$

$T^0 = 4854100;\ T^1 = 2968600;\ T^2 = 1083100; T^3 = 110094;$

$\dot{m}^0 = 1723.273;\ \dot{m}^1 = 1044.682;\ \dot{m}^2 = 366.092;\ \dot{m}^3 = 24.029;$

Dynamics:

$\dot{\mathbf{r}} = \mathbf{v}$

$$\dot{\mathbf{v}} = -\frac{\mu}{|\mathbf{r}|^3}\mathbf{r} + \frac{T^p}{m}\mathbf{u} - \frac{C_d A^{\text{ref}} \rho_0 |\mathbf{v} - (\omega_e \times \mathbf{r})|(\mathbf{v} - (\omega_e \times \mathbf{r}))e^{\frac{|\mathbf{r}|-R_e}{H_0}}}{2m}$$

$\dot{m} = -\dot{m}^p \qquad p = 0, 1, 2, 3$

Path constraints:

$|\mathbf{u}| = 1$

$|\mathbf{r}| \geq R_e$

Terminal constraints:

$t_0^0 = 0;\ t_0^1 = 75.2;\ t_0^2 = 150.4;\ t_0^3 = 261$

$t_f^0 = 75.2;\ t_f^1 = 150.4;\ t_f^2 = 261$

Let $\mathbf{r_f} = [x_0(t_f^3), x_1(t_f^3), x_2(t_f^3)];\ \mathbf{v_f} = [x_3(t_f^3), x_4(t_f^3), x_5(t_f^3)];$

$a_f(\mathbf{r_f}, \mathbf{v_f}) = 24361140; e_f(\mathbf{r_f}, \mathbf{v_f}) = 0.7308;$

$i_f(\mathbf{r_f}, \mathbf{v_f}) = \frac{28.5\pi}{180}; \Omega_f(\mathbf{r_f}, \mathbf{v_f}) = \frac{269.8\pi}{180};$

$\omega_f(\mathbf{r_f}, \mathbf{v_f}) = \frac{130.5\pi}{180};$

Where $a_f, e_f, i_f, \Omega_f, \omega_f$ are computed as follows

$\mathbf{h} = \mathbf{r_f} \times \mathbf{v_f};\ \mathbf{n} = [0, 0, 1]^T \times \mathbf{v_f};$

$$\mathbf{e} = \frac{1}{\mu}\mathbf{v_f} \times \mathbf{h} - \frac{\mathbf{r_f}}{|\mathbf{r_f}|};$$

$$e_f = |\mathbf{e}|;\ a_f = -\frac{\mu}{|\mathbf{v_f}|^2 - \frac{2\mu}{|\mathbf{r_f}|}};\ i_f = \cos^{-1}\left(\frac{\mathbf{h}.[0,0,1]^T}{|\mathbf{h}|}\right);$$

if $\mathbf{n}[1]>0$: $\Omega_f = \cos^{-1}\left(\frac{\mathbf{n}.[0,0,1]^T}{|\mathbf{n}|}\right)$ else: $\Omega_f = (2\pi - \Omega_f)$;

if e[2]>0: $\omega_f = \cos^{-1}\left(\frac{\mathbf{n}.\mathbf{e}}{|\mathbf{n}|e_f}\right)$ else: $\omega_f = (2\pi - \omega_f)$;

Event constraints:

$\mathbf{r}(t_f^p) - \mathbf{r}(t_0^{p+1}) = 0;\ \mathbf{v}(t_f^p) - \mathbf{v}(t_0^{p+1}) = 0; \quad p = 0, 1, 2$

$m(t_f^p) - m(t_0^{p+1}) = [13680, 6840, 8830] \quad p = 0, 1, 2$

Initial conditions:

$\mathbf{r}(t_0^0) = [5605222.973, 0, 3043387.761]; \mathbf{v}(t_0^0) = [0, 408.74, 0];$

$m(t_0^0) = 301454$

We first create an OCP object and then polulate the object with dynamics, path_constraints, terminal_constraints and objective (running_costs, terminal_costs)

[2]:
```
ocp = mp.OCP(n_states=7, n_controls=4, n_phases=3)
```

[3]:
```
# Initialize parameters
Re = 6378145.0  # m
```

(continues on next page)

(continued from previous page)

```
omegaE = 7.29211585e-5
rho0 = 1.225
rhoH = 7200.0
Sa = 4 * np.pi
Cd = 0.5
muE = 3.986012e14
g0 = 9.80665

# Variable initialization
lat0 = 28.5 * np.pi / 180.0
r0 = np.array([Re * np.cos(lat0), 0.0, Re * np.sin(lat0)])
# v0     = omegaE*np.array([-r0[1], r0[0], 0.0])
v0 = omegaE * np.array([0.1, 0.1, 0.1])
m0 = 431.6e3 + 107.5e3
mf = 107.5e3 - 103.5e3
mdryBooster = 431.6e3 - 409.5e3
mdrySecond = mf
x0 = np.array([r0[0], r0[1], r0[2], v0[0], v0[1], v0[2], m0])
q_max = 80 * 1e3
```

[4]:
```
# Step-1 : Define dynamics
# Thrust(N) and mass flow rate(kg/s) in each stage
Thrust = [9 * 934.0e3, 934.0e3, 934.0e3]


def stage_dynamics(x, u, t, param=0, T=0.0):
    r = x[:3]
    v = x[3:6]
    m = x[6]
    r_mag = ca.sqrt(r[0] * r[0] + r[1] * r[1] + r[2] * r[2])
    v_rel = v  # ca.vertcat(v[0] + r[1] * omegaE, v[1] - r[0] * omegaE, v[2])
    v_rel_mag = ca.sqrt(v_rel[0] * v_rel[0] + v_rel[1] * v_rel[1] + v_rel[2] * v_rel[2])
    h = r_mag - Re
    rho = rho0 * ca.exp(-h / rhoH)
    D = -rho / (2 * m) * Sa * Cd * v_rel_mag * v_rel
    g = -muE / (r_mag * r_mag * r_mag) * r

    xdot = [
        x[3],
        x[4],
        x[5],
        T * u[3] / m * u[0] + param * D[0] + g[0],
        T * u[3] / m * u[1] + param * D[1] + g[1],
        T * u[3] / m * u[2] + param * D[2] + g[2],
        -T * u[3] / (340.0 * g0),
    ]
    return xdot


def get_dynamics(param):
    dynamics0 = lambda x, u, t: stage_dynamics(x, u, t, param=param, T=Thrust[0])
    dynamics1 = lambda x, u, t: stage_dynamics(x, u, t, param=param, T=Thrust[1])
```

(continues on next page)

(continued from previous page)

```
    dynamics2 = lambda x, u, t: stage_dynamics(x, u, t, param=param, T=Thrust[2])

    return [dynamics0, dynamics1, dynamics2]


ocp.dynamics = get_dynamics(0)
```

[5]:
```
# Step-2: Add path constraints
def path_constraints0(x, u, t):
    return [
        u[0] * u[0] + u[1] * u[1] + u[2] * u[2] - 1,
        -u[0] * u[0] - u[1] * u[1] - u[2] * u[2] + 1,
        -ca.sqrt(x[0] * x[0] + x[1] * x[1] + x[2] * x[2]) / Re + 1,
    ]


def path_constraints2(x, u, t, dynP=0, gs=0):
    r_mag = ca.sqrt(x[0] * x[0] + x[1] * x[1] + x[2] * x[2])
    h = r_mag - Re
    rho = rho0 * ca.exp(-h / rhoH)
    v_sq = x[3] * x[3] + x[4] * x[4] + x[5] * x[5]
    r_rf = ca.vertcat(x[0] - x0[0], x[1] - x0[1], x[2] - x0[2])
    r_rf_mag = ca.sqrt(r_rf[0] * r_rf[0] + r_rf[1] * r_rf[1] + r_rf[2] * r_rf[2])
    rf_mag = np.sqrt(x0[0] * x0[0] + x0[1] * x0[1] + x0[2] * x0[2])
    glide_slope_factor = np.cos(80.0 * np.pi / 180.0)

    return [
        dynP * 0.5 * rho * v_sq / q_max - 1.0,
        u[0] * u[0] + u[1] * u[1] + u[2] * u[2] - 1,
        -u[0] * u[0] - u[1] * u[1] - u[2] * u[2] + 1,
        -ca.sqrt(x[0] * x[0] + x[1] * x[1] + x[2] * x[2]) / Re + 1,
        gs
        * (
            r_rf_mag * glide_slope_factor
            - (r_rf[0] * x0[0] + r_rf[1] * x0[1] + r_rf[2] * x0[2]) / rf_mag
        ),
    ]


ocp.path_constraints = [path_constraints0, path_constraints0, path_constraints2]
```

[6]:
```
# Step-3: Add terminal cost and constraints
def terminal_cost1(xf, tf, x0, t0):
    return -xf[6] / m0


ocp.terminal_costs[1] = terminal_cost1


def terminal_constraints1(x, t, x0, t0):
    h = ca.vertcat(
```

(continues on next page)

(continued from previous page)

```
        x[1] * x[5] - x[4] * x[2], x[3] * x[2] - x[0] * x[5], x[0] * x[4] - x[1] * x[3]
    )

    n = ca.vertcat(-h[1], h[0], 0)
    r = ca.sqrt(x[0] * x[0] + x[1] * x[1] + x[2] * x[2])

    e = ca.vertcat(
        1 / muE * (x[4] * h[2] - x[5] * h[1]) - x[0] / r,
        1 / muE * (x[5] * h[0] - x[3] * h[2]) - x[1] / r,
        1 / muE * (x[3] * h[1] - x[4] * h[0]) - x[2] / r,
    )

    e_mag = ca.sqrt(e[0] * e[0] + e[1] * e[1] + e[2] * e[2])
    h_sq = h[0] * h[0] + h[1] * h[1] + h[2] * h[2]
    v_mag = ca.sqrt(x[3] * x[3] + x[4] * x[4] + x[5] * x[5])

    a = -muE / (v_mag * v_mag - 2.0 * muE / r)
    i = ca.acos(h[2] / ca.sqrt(h_sq))
    n_mag = ca.sqrt(n[0] * n[0] + n[1] * n[1])

    node_asc = ca.acos(n[0] / n_mag)
    # if n[1] < -1e-12:
    node_asc = 2 * np.pi - node_asc

    argP = ca.acos((n[0] * e[0] + n[1] * e[1]) / (n_mag * e_mag))
    # if e[2] < 0:
    #    argP = 2*np.pi - argP

    a_req = 6593145.0  # 24361140.0
    e_req = 0.0076
    i_req = 28.5 * np.pi / 180.0
    node_asc_req = 269.8 * np.pi / 180.0
    argP_req = 130.5 * np.pi / 180.0

    return [
        (a - a_req) / (Re),
        e_mag - e_req,
        i - i_req,
        node_asc - node_asc_req,
        argP - argP_req,
    ]


def terminal_constraints2(x, t, x_0, t_0):
    return [
        (x[0] - x0[0]) / Re,
        (x[1] - x0[1]) / Re,
        (x[2] - x0[2]) / Re,
        (x[3] - x0[3]) / np.sqrt(muE / Re),
        (x[4] - x0[4]) / np.sqrt(muE / Re),
        (x[5] - x0[5]) / np.sqrt(muE / Re),
    ]
```

(continues on next page)

(continued from previous page)

```
ocp.terminal_constraints[1] = terminal_constraints1
ocp.terminal_constraints[2] = terminal_constraints2
```

Scale the variables

[7]:
```
ocp.scale_x = np.array(
    [
        1 / Re,
        1 / Re,
        1 / Re,
        1 / np.sqrt(muE / Re),
        1 / np.sqrt(muE / Re),
        1 / np.sqrt(muE / Re),
        1 / m0,
    ]
)
ocp.scale_t = np.sqrt(muE / Re) / Re
```

Initial state and Final guess

[8]:
```
# Initial guess estimation
# User defined functions
def ae_to_rv(a, e, i, node, argP, th):
    p = a * (1.0 - e * e)
    r = p / (1.0 + e * np.cos(th))

    r_vec = np.array([r * np.cos(th), r * np.sin(th), 0.0])
    v_vec = np.sqrt(muE / p) * np.array([-np.sin(th), e + np.cos(th), 0.0])

    cn, sn = np.cos(node), np.sin(node)
    cp, sp = np.cos(argP), np.sin(argP)
    ci, si = np.cos(i), np.sin(i)

    R = np.array(
        [
            [cn * cp - sn * sp * ci, -cn * sp - sn * cp * ci, sn * si],
            [sn * cp + cn * sp * ci, -sn * sp + cn * cp * ci, -cn * si],
            [sp * si, cp * si, ci],
        ]
    )

    r_i = np.dot(R, r_vec)
    v_i = np.dot(R, v_vec)

    return r_i, v_i


# Target conditions
a_req = 6593145.0  # 24361140.0
e_req = 0.0076
```

(continues on next page)

(continued from previous page)

```
i_req = 28.5 * np.pi / 180.0
node_asc_req = 269.8 * np.pi / 180.0
argP_req = 130.5 * np.pi / 180.0
th = 0.0
rf, vf = ae_to_rv(a_req, e_req, i_req, node_asc_req, argP_req, th)
xf = np.array([rf[0], rf[1], rf[2], vf[0], vf[1], vf[2], mf])
```

Intial guess

[9]:
```
# Timings
# Timings
t0, t1, t2, t3 = 0.0, 131.4, 453.4, 569.7

# Interpolate to get starting values for intermediate phases
x1 = x0 + (xf - x0) / (t2 - t0) * (t1 - t0)

# Update the state discontinuity values across phases
x0f = np.copy(x1)
x0f[-1] = x0[-1] - (9 * 934e3 / (340.0 * g0) * t1)
mFirst_leftout = 409.5e3 - (9 * 934e3 / (340.0 * g0) * t1)
x1[-1] = x0f[-1] - (mdryBooster + mFirst_leftout)

# Step-8b: Initial guess for the states, controls and phase start and final
#             times
ocp.x00 = np.array([x0, x1, x0f])
ocp.xf0 = np.array([x0f, xf, x0])
ocp.u00 = np.array([[1, 0, 0, 1.0], [1, 0, 0, 1], [0, 1, 0, 1]])
ocp.uf0 = np.array([[0, 1, 0, 1.0], [0, 1, 0, 1], [1, 0, 0, 0.5]])
ocp.t00 = np.array([[t0], [t1], [t1]])
ocp.tf0 = np.array([[t1], [t2], [t3]])
```

Box constraints

[10]:
```
# Step-8c: Bounds for states
rmin, rmax = -2 * Re, 2 * Re
vmin, vmax = -10000.0, 10000.0

lbx0 = [rmin, rmin, rmin, vmin, vmin, vmin, x0f[-1]]
lbx1 = [rmin, rmin, rmin, vmin, vmin, vmin, xf[-1]]
lbx2 = [rmin, rmin, rmin, vmin, vmin, vmin, mdryBooster]
ubx0 = [rmax, rmax, rmax, vmax, vmax, vmax, x0[-1]]
ubx1 = [rmax, rmax, rmax, vmax, vmax, vmax, 107.5e3]
ubx2 = [rmax, rmax, rmax, vmax, vmax, vmax, x0f[-1] - 107.5e3]

ocp.lbx = np.array([lbx0, lbx1, lbx2])
ocp.ubx = np.array([ubx0, ubx1, ubx2])

# Bounds for control inputs
ocp.lbu = np.array(
    [[-1.0, -1.0, -1.0, 1.0], [-1.0, -1.0, -1.0, 1.0], [-1.0, -1.0, -1.0, 0.38]]
)
ocp.ubu = np.array([[1.0, 1.0, 1.0, 1.0], [1.0, 1.0, 1.0, 1.0], [1.0, 1.0, 1.0, 1.0]])
```

(continues on next page)

(continued from previous page)

```
# Bounds for phase start and final times
ocp.lbt0 = np.array([[t0], [t1], [t1]])
ocp.ubt0 = np.array([[t0], [t1], [t1]])
ocp.lbtf = np.array([[t1], [t2 - 50], [t3 - 100]])
ocp.ubtf = np.array([[t1], [t2 + 50], [t3 + 100]])

# Event constraint bounds on states : State continuity/disc.
lbe0 = [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, -(mdryBooster + mFirst_leftout)]
lbe1 = [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, -107.5e3]
ocp.lbe = np.array([lbe0, lbe1])
ocp.ube = np.array([lbe0, lbe1])

ocp.phase_links = [(0, 1), (0, 2)]
```

[11]:
```
ocp.validate()
```

Solve with atmospheric drag disabled

[12]:
```
ocp.dynamics = get_dynamics(0)
mpo = mp.mpopt(ocp, 5, 6)
sol = mpo.solve()
```

```
******************************************************************************
This program contains Ipopt, a library for large-scale nonlinear optimization.
 Ipopt is released as open source code under the Eclipse Public License (EPL).
         For more information visit http://projects.coin-or.org/Ipopt
******************************************************************************

Total number of variables...:      956
                     variables with only lower bounds:        0
                variables with lower and upper bounds:      956
                     variables with only upper bounds:        0
Total number of equality constraints...:      746
Total number of inequality constraints...:      641
        inequality constraints with only lower bounds:        0
   inequality constraints with lower and upper bounds:      300
        inequality constraints with only upper bounds:      341


Number of Iterations...: 153

                                   (scaled)                 (unscaled)
Objective...:  -2.9204480860707361e-02   -2.9204480860707361e-02
Dual infeasibility...:   1.9040474700928261e-07    1.9040474700928261e-07
Constraint violation...:   6.7175836255209499e-06    6.7175836255209499e-06
Complementarity...:   2.5059035596800655e-09    2.5059035596800655e-09
Overall NLP error...:   6.7175836255209499e-06    6.7175836255209499e-06
```

(continues on next page)

(continued from previous page)

```
Number of objective function evaluations             = 203
Number of objective gradient evaluations             = 154
Number of equality constraint evaluations            = 203
Number of inequality constraint evaluations          = 203
Number of equality constraint Jacobian evaluations   = 154
Number of inequality constraint Jacobian evaluations = 154
Number of Lagrangian Hessian evaluations             = 153
Total CPU secs in IPOPT (w/o function evaluations)   =      2.930
Total CPU secs in NLP function evaluations           =      0.117

EXIT: Solved To Acceptable Level.
      solver  :   t_proc      (avg)   t_wall      (avg)    n_eval
       nlp_f  | 744.00us (  3.67us) 720.91us (  3.55us)       203
       nlp_g  |  28.51ms (140.46us)  28.24ms (139.09us)       203
    nlp_grad  | 384.00us (384.00us) 383.90us (383.90us)         1
  nlp_grad_f  |   1.51ms (  9.72us)   1.41ms (  9.13us)       155
  nlp_hess_l  |  34.36ms (224.61us)  34.18ms (223.38us)       153
   nlp_jac_g  |  41.84ms (269.92us)  41.74ms (269.30us)       155
       total  |   3.07 s (  3.07 s)   3.06 s (  3.06 s)         1
```

Solve the problem with drag enabled now with revised initial guess

[13]:
```
ocp.dynamics = get_dynamics(1)
ocp.path_constraints[2] = lambda x, u, t: path_constraints2(x, u, t, dynP=1, gs=0)
ocp.validate()

mpo._ocp = ocp
sol = mpo.solve(
    sol, reinitialize_nlp=True, nlp_solver_options={"ipopt.acceptable_tol": 1e-6}
)
```

```
Total number of variables...:      956
                     variables with only lower bounds:        0
                variables with lower and upper bounds:      956
                     variables with only upper bounds:        0
Total number of equality constraints...:      746
Total number of inequality constraints...:      641
        inequality constraints with only lower bounds:        0
   inequality constraints with lower and upper bounds:      300
        inequality constraints with only upper bounds:      341


Number of Iterations...: 61

                                   (scaled)                 (unscaled)
Objective...:  -3.1423421485119993e-02   -3.1423421485119993e-02
Dual infeasibility...:   6.7634340985309009e-12    6.7634340985309009e-12
Constraint violation...:   1.9999893324795792e-08    1.9999893324795792e-08
Complementarity...:   9.1006695117999755e-10    9.1006695117999755e-10
Overall NLP error...:   1.9999893324795792e-08    1.9999893324795792e-08


Number of objective function evaluations             = 172
```

(continues on next page)

(continued from previous page)

```
Number of objective gradient evaluations             = 63
Number of equality constraint evaluations            = 172
Number of inequality constraint evaluations          = 172
Number of equality constraint Jacobian evaluations   = 63
Number of inequality constraint Jacobian evaluations = 63
Number of Lagrangian Hessian evaluations             = 62
Total CPU secs in IPOPT (w/o function evaluations)   =      1.441
Total CPU secs in NLP function evaluations           =      0.079

EXIT: Solved To Acceptable Level.
      solver  :   t_proc      (avg)   t_wall      (avg)    n_eval
       nlp_f  | 539.00us (  3.13us) 523.99us (  3.05us)       172
       nlp_g  |  25.06ms (145.73us)  24.79ms (144.13us)       172
    nlp_grad  | 305.00us (305.00us) 303.70us (303.70us)         1
  nlp_grad_f  | 579.00us (  9.05us) 660.80us ( 10.33us)        64
  nlp_hess_l  |  25.98ms (419.10us)  25.94ms (418.32us)        62
   nlp_jac_g  |  23.84ms (372.56us)  23.79ms (371.73us)        64
       total  |   1.55 s (  1.55 s)   1.54 s (  1.54 s)         1
```

Retrive data from the solution

[14]:
```
post = mpo.process_results(sol, plot=False, scaling=False)
mp.post_process._INTERPOLATION_NODES_PER_SEG = 200
x, u, t, _ = post.get_data(phases=ocp.phase_links[0], interpolate=False)
print("Payload:\nMass to orbit : ", round(x[-1][-1], 4), "kg(MPOPT) vs 17310kg(Beigler␣
↪paper)")
print("Time to orbit (SECO) : ", round(t[-1][0], 4), "s(MPOPT) vs 453.4s(Beigler paper)")
x, u, t, _ = post.get_data(phases=ocp.phase_links[1], interpolate=False)
print("\nBooster:\n Mass at landing : ", round(x[-1][-1], 4), "kg(MPOPT) vs␣
↪22100kg(Beigler paper)")
print("Time of landing : ", round(t[-1][0], 4), "s(MPOPT) vs 569.7s(Beigler paper)")
```

```
Payload:
Mass to orbit :  16940.3665 kg(MPOPT) vs 17310kg(Beigler paper)
Time to orbit (SECO) :  454.6864 s(MPOPT) vs 453.4s(Beigler paper)

Booster:
 Mass at landing :  22100.0 kg(MPOPT) vs 22100kg(Beigler paper)
Time of landing :  579.5091 s(MPOPT) vs 569.7s(Beigler paper)
```

Plot results

Plot trajectory of launch to orbit (Second stage) and Booster return to Ship.

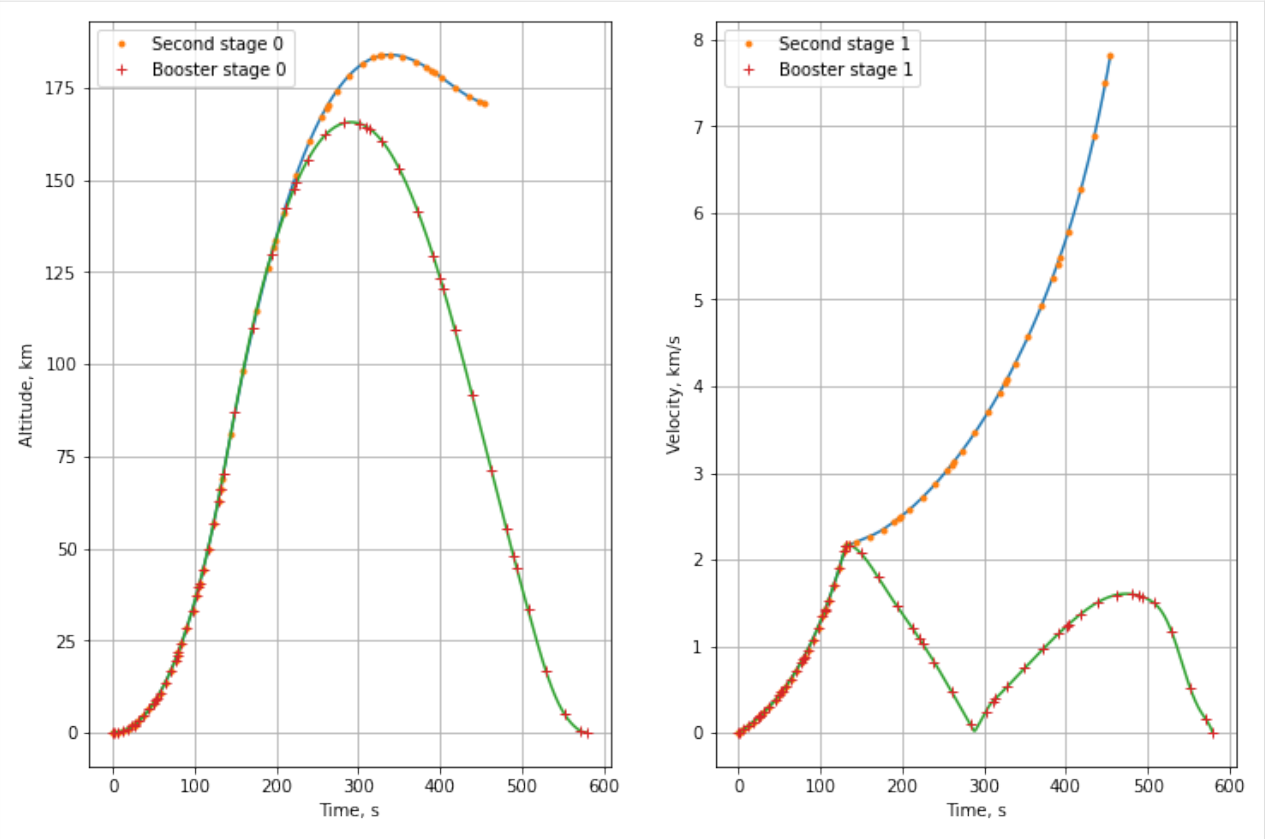
[15]:
```
mp.plt.rcParams["figure.figsize"] = (12,8)
x0, u0, t0, _ = post.get_data(phases=ocp.phase_links[0], interpolate=True)
x1, u1, t1, _ = post.get_data(phases=ocp.phase_links[1], interpolate=True)
x0o, u0o, t0o, _ = post.get_data(phases=ocp.phase_links[0], interpolate=False)
x1o, u1o, t1o, _ = post.get_data(phases=ocp.phase_links[1], interpolate=False)

r0 = 1e-3 * (np.sqrt(x0[:, 0] ** 2 + x0[:, 1] ** 2 + x0[:, 2] ** 2) - Re)
v0 = 1e-3 * np.sqrt(x0[:, 3] ** 2 + x0[:, 4] ** 2 + x0[:, 5] ** 2)
```

(continues on next page)

(continued from previous page)

```
y0 = np.column_stack((r0, v0))
fig0, axs0 = post.plot_single_variable(y0, t0, [[0], [1]], axis=0)

r0o = 1e-3 * (np.sqrt(x0o[:, 0] ** 2 + x0o[:, 1] ** 2 + x0o[:, 2] ** 2) - Re)
v0o = 1e-3 * np.sqrt(x0o[:, 3] ** 2 + x0o[:, 4] ** 2 + x0o[:, 5] ** 2)
y0o = np.column_stack((r0o, v0o))
fig0o, axs0o = post.plot_single_variable(
    y0o,
    t0o,
    [[0], [1]],
    axis=0,
    fig=fig0,
    axs=axs0,
    tics=["."] * 15,
    name="Second stage",
)

r1 = 1e-3 * (np.sqrt(x1[:, 0] ** 2 + x1[:, 1] ** 2 + x1[:, 2] ** 2) - Re)
v1 = 1e-3 * np.sqrt(x1[:, 3] ** 2 + x1[:, 4] ** 2 + x1[:, 5] ** 2)
y1 = np.column_stack((r1, v1))
fig1, axs1 = post.plot_single_variable(y1, t1, [[0], [1]], axis=0, fig=fig0o, axs=axs0o)

r1o = 1e-3 * (np.sqrt(x1o[:, 0] ** 2 + x1o[:, 1] ** 2 + x1o[:, 2] ** 2) - Re)
v1o = 1e-3 * np.sqrt(x1o[:, 3] ** 2 + x1o[:, 4] ** 2 + x1o[:, 5] ** 2)
y1o = np.column_stack((r1o, v1o))
fig1o, axs1o = post.plot_single_variable(
    y1o,
    t1o,
    [[0], [1]],
    axis=0,
    fig=fig1,
    axs=axs1,
    tics=["+"] * 15,
    name="Booster stage",
)

axs1o[0].set(ylabel="Altitude, km")
axs1o[1].set(ylabel="Velocity, km/s")
```

[15]:
```
[Text(0, 0.5, 'Velocity, km/s')]
```



Next steps: **:doc:`notebooks`_**, **:doc:`documentation`_**

CHAPTER

FIVE

CODE DOCUMENTATION

5.1 MPOPT package implementation

5.1.1 Collocation

Collocation class

class mpopt.mpopt.**Collocation**(*poly_orders: List = []*, *scheme: str = 'LGR'*, *polynomial_type: str = 'lagrange'*)

Bases: object

Collocation functionality for optimizer

Functionality related to polynomial basis, respective differential and integral matrices calculation is implemented here.

Numpy polynomial modules is used for calculating differentiation and quadrature weights. Hence, computer precision can affect these derivative calculations

D_MATRIX_METHOD = 'symbolic'

TVAR = SX(t)

__init__(*poly_orders: List = []*, *scheme: str = 'LGR'*, *polynomial_type: str = 'lagrange'*)

Initialize the collocation object

:param : poly_orders: List of polynomial degrees used in collocation :param : scheme: Scheme used to define collocation nodes (Possible options - “LG”, “LGR”, “LGL”, “CGL”)

LG - Legendre-Gauss (LG) LGR - Legendre-Gauss-Radau (LGR) LGL - Legendre-Gauss-Lobatto (LGL) CGL - Chebyshev-Gauss-Lobatto (CGL)

:param
[polynomial_type: polynomials used in state and control approximation]

- lagrange

get_composite_differentiation_matrix(*poly_orders: List = None*, *order: int = 1*)

Get composite differentiation matrix for given collocation approximation

:param
[poly_orders: order of the polynomials used in collocation with each] element representing one segment

get_composite_interpolation_Dmatrix_at(*taus*, *poly_orders: List = None*, *order: int = 1*)

Get differentiation matrix corresponding to given basis polynomial degree at nodes different from collocation nodes

:param
[taus: List of scaled taus (between 0 and 1) with length of list equal] to length of poly_orders (= number of segments)

:param
[poly_orders: order of the polynomials used in collocation with each] element representing one segment

Returns
D: Composite differentiation matrix

get_composite_interpolation_matrix(*taus*, *poly_orders: List = None*)

Get interpolation matrix corresponding to given basis polynomial degree for colloation.

:param
[taus: List of scaled taus (between 0 and 1) with length of list equal] to length of poly_orders (= number of segments). Note- taus are not assumed to have overlap between segments(end element != start of next phase)

:param
[poly_orders: order of the polynomials used in collocation with each] element representing one segment

Returns
I: composite interpolation matrix

get_composite_quadrature_weights(*poly_orders: List = None*, *tau0=None*, *tau1=None*)

Get composite quadrature weights for given collocation approximation

:param
[poly_orders: order of the polynomials used in collocation with each] element representing one segment

get_diff_matrices(*poly_orders: List = None*, *order: int = 1*)

Get differentiation matrices for given collocation approximation

:param : poly_orders: order of the polynomials used in collocation with each element representing one segment

get_diff_matrix(*key*, *taus: ndarray = None*, *order: int = 1*)

Get differentiation matrix corresponding to given basis polynomial degree

:param : degree: order of the polynomial used in collocation :param : taus: Diff matrix computed at these nodes if not None.

Returns
D: Differentiation matrix

classmethod get_diff_matrix_fn(*polynomial_type: str = 'lagrange'*)

Return a function that returns differentiation matrix

:param : polynomial_type: (lagrange)

Returns
Diff matrix function with arguments (degree, taus_at=None)

`get_interpolation_Dmatrices_at`(*taus*, *keys: List = None*, *order: int = 1*)

Get differentiation matrices at the interpolated nodes (taus), different from the collocation nodes.

:param
[taus: List of scaled taus (between 0 and 1) with length of list equal] to length of poly_orders (= number of segments)

:param : keys: keys of the roots and polys Dict element containing roots of the legendre polynomials and polynomials themselves

Returns

Dict
[(key, value)] key - segment number (starting from 0) value - Differentiation matrix(C) such that DX_tau = D*X_colloc where

X_colloc is the values of states at the collocation nodes

`get_interpolation_matrices`(*taus*, *poly_orders: List = None*)

Get interpolation matrices corresponding to each poly_order at respective element in list of taus

:param
[taus: List of scaled taus (between 0 and 1) with length of list equal] to length of poly_orders (= number of segments).

:param
[poly_orders: order of the polynomials used in collocation with each] element representing one segment

Returns

(key, value)
key - segment number (starting from 0) value - interpolation matrix(C) such that X_new = C*X

Return type
Dict

`get_interpolation_matrix`(*taus*, *degree*)

Get interpolation matrix corresponsing nodes (taus) where the segment is approximated with polynomials of degree (degree)

:param : taus: Points where interpolation is performed :param : degree: Order of the collocation polynomial

`classmethod get_lagrange_polynomials`(*roots*)

Get basis polynomials given the collocation nodes

:param : roots: Collocation points

`classmethod get_polynomial_function`(*polynomial_type: str = 'lagrange'*)

Get function which returns basis polynomials for the collocation given polynomial degree

:param : polynomial_type: str, 'lagrange'

Returns
poly_basis_fn: Function which returns basis polynomials

`get_quad_weight_matrices`(*keys: List = None*, *tau0=None*, *tau1=None*)

Get quadrature weights for given collocation approximation

:param : keys: keys of the Dict element (roots and polys), Normally these keys are equal to the order of the polynomial

get_quadrature_weights(*key*, *tau0=None*, *tau1=None*)

Get quadrature weights corresponding to given basis polynomial degree

:param : degree: order of the polynomial used in collocation

classmethod get_quadrature_weights_fn(*polynomial_type: str = 'lagrange'*)

Return a function that returns quadrature weights for the cost function approx.

:param : polynomial_type: (lagrange)

Returns
quadrature weights function with arguments (degree)

init_polynomials(*poly_orders*) → None

Initialize roots of the polynomial and basis polynomials

:param : poly_orders: List of polynomial degrees used in collocation

init_polynomials_with_customized_roots(*roots_dict: Dict = None*) → None

Initialize polynomials with predefined roots

:param : roots_dict: Dictionary with a key for the roots and polys (Ideally not numbers as they are already taken by the regular polynominals)

Collocation Roots class

class mpopt.mpopt.**CollocationRoots**(*scheme: str = 'LGR'*)

Bases: object

Functionality related to commonly used gauss quadrature schemes such as

Legendre-Gauss (LG) Legendre-Gauss-Radau (LGR) Legendre-Gauss-Lobatto (LGL) Chebyshev-Gauss-Lobatto (CGL)

__init__(*scheme: str = 'LGR'*)

Get differentiation matrix corresponding to given basis polynomial degree

:param : degree: order of the polynomial used in collocation

classmethod get_collocation_points(*scheme: str*)

Get function that returns collocation points for the given scheme

:param : scheme: quadrature scheme to find the collocation points

returns: Function, that retuns collocation points when called with polynomial degree

static roots_chebyshev_gauss_lobatto(*tau_min=-1*, *tau_max=1*)

Get Chebyshev-gauss-lobatto collocation points in the interval [_TAU_MIN, _TAU_MAX]

args: None

returns: a function that returns collocation points given polynomial degree

static roots_legendre_gauss(*tau_min=-1*, *tau_max=1*)

Get legendre-gauss-radau collocation points in the interval [_TAU_MIN, _TAU_MAX)

args: None

returns: a function that returns collocation points given polynomial degree

static roots_legendre_gauss_lobatto(*tau_min=-1*, *tau_max=1*)

Get legendre-gauss-lobatto collocation points in the interval [_TAU_MIN, _TAU_MAX]

args: None

returns: a function that returns collocation points given polynomial degree

static roots_legendre_gauss_radau(*tau_min=-1*, *tau_max=1*)

Get legendre-gauss-radau (Left aligned) collocation points in the interval [_TAU_MIN, _TAU_MAX]

args: None

returns: a function that returns collocation points, given polynomial degree

5.1.2 OCP definition

OCP definition class

class mpopt.mpopt.**OCP**(*n_states: int = 1*, *n_controls: int = 1*, *n_phases: int = 1*, *n_params=0*, ***kwargs*)

Bases: `object`

Define Optimal Control Problem

Optimal control problem definition in standard Bolza form.

Examples of usage:

```
>>> ocp = OCP(n_states=1, n_controls=1, n_phases=1)
>>> ocp.dynamics[0] = lambda x, u, t, a: [u[0]]
>>> ocp.path_constraints[0] = lambda x, u, t, a: [x[0] + u[0]]
>>> ocp.running_costs[0] = lambda x, u, t, a: x[0]
>>> ocp.terminal_costs[0] = lambda xf, tf, x0, t0, a: xf[0]
>>> ocp.terminal_constraints[0] = lambda xf, tf, x0, t0, a: [xf[0] + 2]
```

LB_DYNAMICS = 0

LB_PATH_CONSTRAINTS = -inf

LB_TERMINAL_CONSTRAINTS = 0

UB_DYNAMICS = 0

UB_PATH_CONSTRAINTS = 0

UB_TERMINAL_CONSTRAINTS = 0

__init__(*n_states: int = 1*, *n_controls: int = 1*, *n_phases: int = 1*, *n_params=0*, ***kwargs*)

Initialize OCP object For all phases, number of states and controls are assumed to be same. It's easy to connect when the states and controls are same in multi-phase OCP.

:param : n_states: number of state variables in the OCP :param : n_controls: number of control variables in the OCP :param : n_params: number of algebraic parameters in each phase :param : n_phases: number of phases in the OCP

Returns

OCP class object with default initialization of methods and parameters

get_dynamics(*phase: int = 0*)

Get dynamics function for the given phase

:param : phase: index of the phase (starting from 0)

Returns
dynamics: system dynamics function with arguments x, u, t, a

get_path_constraints(*phase: int = 0*)

Get path constraints function for the given phase

:param : phase: index of the phase (starting from 0)

Returns
path_constraints: path constraints function with arguments x, u, t, a

get_running_costs(*phase: int = 0*)

Get running_costs function for the given phase

:param : phase: index of the phase (starting from 0)

Returns
running_costs: system running_costs function with arguments x, u, t, a

get_terminal_constraints(*phase: int = 0*)

Get terminal_constraints function for the given phase

:param : phase: index of the phase (starting from 0)

Returns
terminal_constraints: system terminal_constraints function with arguments x, u, t, a

get_terminal_costs(*phase: int = 0*)

Get terminal_costs function for the given phase

:param : phase: index of the phase (starting from 0)

Returns
terminal_costs: system terminal_costs function with arguments x, u, t, a

has_path_constraints(*phase: int = 0*) → bool

Check if given phase has path constraints in given OCP

:param : phase: index of phase

Returns
status: bool (True/False)

has_terminal_constraints(*phase: int = 0*) → bool

Check if given phase has terminal equality constraints in given OCP

:param : phase: index of phase

Returns
status: bool (True/False)

validate() → None

Validate dimensions and initialization of attributes

5.1.3 mpopt base class

class mpopt.mpopt.**mpopt**(*problem: OCP*, *n_segments: int = 1*, *poly_orders: List[int] = [9]*, *scheme: str = 'LGR'*, ***kwargs*)

Bases: `object`

Multiphase Optimal Control Problem Solver

This is the base class, implementing the OCP discretization, transcription and calls to NLP solver

Examples :

```
>>> # Moon lander problem
>>> from mpopt import mp
>>> ocp = mp.OCP(n_states=2, n_controls=1, n_phases=1)
>>> ocp.dynamics[0] = lambda x, u, t: [x[1], u[0] - 1.5]
>>> ocp.running_costs[0] = lambda x, u, t: u[0]
>>> ocp.terminal_constraints[0] = lambda xf, tf, x0, t0: [xf[0], xf[1]]
>>> ocp.x00[0] = [10, -2]
>>> ocp.lbu[0] = 0; ocp.ubu[0] = 3
>>> ocp.lbtf[0] = 3; ocp.ubtf[0] = 5
>>> opt = mp.mpopt(ocp, n_segments=20, poly_orders=[3]*20)
>>> solution = opt.solve()
>>> post = opt.process_results(solution, plot=True)
```

__init__(*problem: OCP*, *n_segments: int = 1*, *poly_orders: List[int] = [9]*, *scheme: str = 'LGR'*, ***kwargs*)

Initialize the optimizer :param n_segments: number of segments in each phase :param poly_orders: degree of the polynomial in each segment :param problem: instance of the OCP class

static compute_interpolation_taus_corresponding_to_original_grid(*nodes_req*, *seg_widths*, *tau0=0*, *tau1=1*)

Compute the taus on original solution grid corresponding to the required interpolation nodes

:param : nodes_req: target_nodes :param : seg_widths: width of the segments whose sum equal 1

Returns

taus: List of taus in each segment corresponding to nodes_req on target_grid

compute_states_from_solution_dynamics(*solution*, *phase: int = 0*, *nodes=None*)

solution : NLP solution

create_nlp() → Tuple

Create Nonlinear Programming problem for the given OCP

Parameters

None

Returns

(nlp_problem, nlp_bounds)

nlp_problem

Dictionary (f, x, g, p) f - Objective function x - Optimization variables vector g - constraint vector p - parameter vector

nlp_bounds

Dictionary (lbx, ubx, lbg, ubg) lbx - Lower bound for the optimization variables (x) ubx - Upper bound for the optimization variables (x) lbu - Lower bound for the constraints vector (g) ubu - Upper bound for the constraints vector (g)

Return type
Tuple

create_solver(*solver: str = 'ipopt'*, *options: Dict = {}*) → None

Create NLP solver

:param
[solver: Optimization method to be used in nlp_solver (List of plugins] avaiable at http://casadi.sourceforge.net/v2.0.0/api/html/d6/d07/classcasadi_1_1NlpSolver.html)

:param
[options: Dictionary] List of options for the optimizer (Based on CasADi documentation)

Returns
None

Updates the nlpsolver object in the present optimizer class

create_variables() → None

Create casadi variables for states, controls, time and segment widths which are used in NLP transcription

Initialized casadi varaibles for optimization.

args: None returns : None

discretize_phase(*phase: int*) → Tuple

Discretize single phase of the Optimal Control Problem

Parameters
phase – index of the phase (starting from 0)

returns :
Tuple : Constraint vector (G, Gmin, Gmax) and objective function (J)

get_discretized_dynamics_constraints_and_cost_matrices(*phase: int = 0*) → Tuple

Get discretized dynamics, path constraints and running cost at each collocation node in a list

:param : phase: index of phase

Returns

(f, c, q)
f - List of constraints for discretized dynamics c - List of constraints for discretized path constraints q - List of constraints for discretized running costs

Return type
Tuple

get_dynamics_residuals(*solution*, *nodes=None*, *grid_type=None*, *residual_type=None*, *plot=False*, *fig=None*, *axs=None*)

Compute residual of the system dynamics at given taus (Normalized [0, 1]) by interpolating the given solution onto a fixed grid consisting of single segment per phase with

roots at given target_nodes.

:param : grid_type: target grid type (normalized between 0 and 1) :param : solution: solution of the NLP as reported by the solver :param : nodes: grid where the residual is computed (between tau0, tau1) in a list (nodes[0] -> Nodes for phase0) :param : options: Options for the target grid :param : residual_type:

None - Actual residual values “relative” - Scaled residual between -1 and 1

Returns
residuals: residual vector for the dynamics at the given taus

`get_dynamics_residuals_single_phase`(*solution*, *phase: int = 0*, *target_nodes: List = None*)

Compute residual of the system dynamics at given taus (Normalized [0, 1]) by interpolating the given solution onto a fixed grid consisting of single segment per phase with

roots at given target_nodes.

:param : target_nodes: target grid nodes (normalized between 0 and 1) :param : solution: solution of the NLP as reported by the solver

Returns
residuals: residual vector for the dynamics at the given taus

`get_event_constraints`() → Tuple

Estimate the constraint vectors for linking the phases

Parameters
`None`

Returns

Constraint vectors (E, Emin, Emax) containing
phase linking constraints, discontinuities across states, controls and time variables.

Return type
Tuple

`static get_interpolated_time_grid`(*t_orig*, *taus: ndarray*, *poly_orders: ndarray*, *tau0: float*, *tau1: float*)

Update the time vector with the interpolated grid across each segment of the original optimization problem

:param : t_orig: Time grid of the original optimization problem (unscaled/scaled) :param : taus: grid of the interpolation taus across each segment of the original OCP :param : poly_orders: Order of the polynomials across each segment used in solving OCP

Returns
time: Interpolated time grid

`get_nlp_constrains_for_control_input_at_mid_colloc_points`(*phase: int = 0*) → Tuple

Get NLP constrains on control input at mid points of the collocation nodes

Box constraints on control input

:param : phase: index of the corresponding phase

returns:

Tuple
[(DU, DUmin, DUmax)] mU - CasADi vector of constraints for the control input at mid colloc points mUmin - Respective lowever bound vector mUmax - Respective upper bound vector

`get_nlp_constrains_for_control_slope_continuity_across_segments`(*phase: int = 0*) → Tuple

Get NLP constrains to maintain control input slope continuity across segments

:param : phase: index of the corresponding phase

Returns

(DU, DUmin, DUmax)

DU - CasADi vector of constraints for the slope of control input
across segments

DUmin - Respective lowever bound vector DUmax - Respective upper bound vector

Return type
Tuple

`get_nlp_constraints_for_control_input_slope`(*phase: int = 0*) → Tuple

Get NLP constraints slope on control input (U)

:param : phase: index of the corresponding phase

returns:

Tuple
[(DU, DUmin, DUmax)] DU - CasADi vector of constraints for the slope of control input DUmin - Respective lowever bound vector DUmax - Respective upper bound vector

`get_nlp_constraints_for_dynamics`(*f: List = []*, *phase: int = 0*) → Tuple

Get NLP constraints for discretized dynamics

:param : f: Discretized vector of dynamics function evaluated at collocation nodes :param : phase: index of the phase corresponding to the given dynamics

returns:

Tuple
[(F, Fmin, Fmax)] F - CasADi vector of constraints for the dynamics Fmin - Respective lowever bound vector Fmax - Respective upper bound vector

`get_nlp_constraints_for_path_contraints`(*c: List = []*, *phase: int = 0*) → Tuple

Get NLP constraints for discretized path constraints

:param : c: Discretized vector of path constraints evaluated at collocation nodes :param : phase: index of the corresponding phase

returns:

Tuple
[(C, Cmin, Cmax)] C - CasADi vector of constraints for the path constraints Cmin - Respective lowever bound vector Cmax - Respective upper bound vector

`get_nlp_constraints_for_terminal_contraints`(*phase: int = 0*) → Tuple

Get NLP constraints for discretized terminal constraints

:param : phase: index of the corresponding phase

returns:

Tuple
[(TC, TCmin, TCmax, J)] TC - CasADi vector of constraints for the terminal constraints TCmin - Respective lowever bound vector TCmax - Respective upper bound vector J - Terminal cost

`get_nlp_variables`(*phase: int*) → Tuple

Retrieve optimization variables and their bounds for a given phase

:param : phase: index of the phase (starting from 0)

Returns

(Z, Zmin, Zmax)
Z - Casadi SX vector containing optimization variables for the given phase (X, U, t0, tf) Zmin - Lower bound for the variables in 'Z' Zmax - Upper bound for the variables in 'Z'

Return type
Tuple

`get_residual_grid_taus`(*phase: int = 0*, *grid_type: str = None*)

Select the non-collocation nodes in a given phase

This is often useful in estimation of residual once the OCP is solved. Starting and end nodes are not included.

:param : phase: Index of the phase :param : grid_type: Type of non-collocation nodes (fixed, mid-points, spectral)

Returns
points: List of normalized collocation points in each segment of the phase

`get_segment_width_parameters`(*solution: Dict*) → List

Get segment widths in all phases

All segment widths are considered equal

:param
[solution: Solution to the nlp from wich the seg_width parameters are] computed (if Adaptive)

Returns

seg_widths: numerical values for the fractions of the segment widths
that equal 1 in each phase

`get_solver_warm_start_input_parameters`(*solution: Dict = None*)

Create dictionary of objects for warm starting the solver using results in 'solution'

:param : solution: Solution of nlp_solver

Returns
dict: (x0, lam_x0, lam_g0)

`get_state_second_derivative`(*solution*, *grid_type='spectral'*, *nodes=None*, *plot=False*, *fig=None*, *axs=None*)

Compute residual of the system states at given taus (Normalized [0, 1]) by interpolating the given solution onto a fixed grid consisting of single segment per phase with

roots at given target_nodes.

:param : grid_type: target grid type (normalized between 0 and 1) :param : solution: solution of the NLP as reported by the solver :param : options: Options for the target grid

Returns
residuals: residual vector for the states at the given taus

`get_state_second_derivative_single_phase`(*solution*, *phase: int = 0*, *nodes: List = None*, *grid_type: str = None*, *residual_type: str = None*)

Compute residual of the system dynamics at given taus (Normalized [0, 1]) by interpolating the given solution onto a fixed grid consisting of single segment per phase with

roots at given target_nodes.

:param : target_nodes: target grid nodes (normalized between 0 and 1) :param : solution: solution of the NLP as reported by the solver :param : residual_type: 'relative' if relative is req.

Returns
residuals: residual vector for the dynamics at the given taus

get_states_residuals(*solution*, *phases=None*, *nodes=None*, *residual_type=None*, *plot=False*, *fig=None*, *axs=None*)

Compute residual of the system dynamics at given taus (Normalized [0, 1]) by interpolating the given solution onto a fixed grid consisting of single segment per phase with

roots at given target_nodes.

:param : grid_type: target grid type (normalized between 0 and 1) :param : solution: solution of the NLP as reported by the solver :param : phases: As a list with indices ex. [0,1] :param : nodes: grid where the residual is computed (between tau0, tau1) in a list (nodes[0] -> Nodes for phase0) :param : options: Options for the target grid

Returns
residuals: residual vector for the dynamics at the given taus

init_segment_width() → None

Initialize segment width in each phase

Segment width is normalized so that sum of all the segment widths equal 1

args: None returns: None

init_solution_per_phase(*phase: int*) → ndarray

Initialize solution vector at all collocation nodes of a given phase.

The initial solution for a given phase is estimated from the initial and terminal conditions defined in the OCP. Simple linear interpolation between initial and terminal conditions is used to estimate solution at interior collocation nodes.

:param : phase: index of phase

Returns
initialized solution for given phase

Return type
solution

init_trajectories(*phase: int = 0*) → Function

Initialize trajectories of states, constrols and time variables

:param : phase: index of the phase

Returns

trajectories: CasADi function which returns states, controls and time variable for the given phase when called with NLP solution vector of all phases
t0, tf - unscaled AND x, u, t - scaled trajectories

initialize_solution() → ndarray

Initialize solution for the NLP from given OCP description

Parameters
None

Returns
Initialized solution for the NLP

Return type
solution

interpolate_single_phase(*solution*, *phase: int = 0*, *target_nodes: ndarray = None*, *grid_type=None*, *options: Set = {}*)

Interpolate the solution at given taus

:param : solution: solution as reported by nlp solver :param : phase: index of the phase :param : target_nodes: List of nodes at which interpolation is performed

Returns

Tuple - (X, DX, DU)
X - Interpolated states DX - Derivative of the interpolated states based on PS polynomials DU - Derivative of the interpolated controls based on PS polynomials

process_results(*solution*, *plot: bool = True*, *scaling: bool = False*, *residual_x: bool = False*, *residual_dx: bool = True*)

Post process the solution of the NLP

:param : solution: NLP solution as reported by the solver :param : plot: bool

True - Plot states and variables in a single plot with states in a subplot and controls in another.
False - No plot

:param
[scaling: bool] True - Plot the scaled variables False - Plot unscaled variables meaning, original solution to the problem

:param
[residuals: bool] To plot norma of the residual in the dynamics evaluated at points different from collocation nodes

Returns
post: Object of post_process class (Initialized)

solve(*initial_solution: Dict = None*, *reinitialize_nlp: bool = False*, *solver: str = 'ipopt'*, *nlp_solver_options: Dict = {}*, *mpopt_options: Dict = {}*, ***kwargs*) → Dict

Solve the Nonlinear Programming problem

:param
[init_solution: Dictionary containing initial solution with keys] x or x0 - Initional solution for the nlp variables

:param
[reinitialize_nlp: (True, False)] True - Reinitialize NLP solver object False - Use already created object if available else create new one

:param
[nlp_solver_options: Options to be passed to the nlp_solver while creating] the solver object, not while solving (like initial conditions)

:param : mpopt_options: Options dict for the optimizer

Returns
solution: Solution as reported by the given nlp_solver object

validate()

Validate initialization of the optimizer object

5.1.4 Adaptive grid refinement scheme-I & II

mpopt h-adaptive class

class mpopt.mpopt.**mpopt_h_adaptive**(*problem: OCP, n_segments: int = 1, poly_orders: List[int] = [9], scheme: str = 'LGR', **kwargs*)

Bases: *mpopt*

Multi-stage Optimal control problem (OCP) solver which implements iterative procedure to refine the segment width in each phase adaptively while keeping the same number of segments

Examples :

```
>>> # Moon lander problem
>>> from mpopt import mp
>>> ocp = mp.OCP(n_states=2, n_controls=1, n_params=0, n_phases=1)
>>> ocp.dynamics[0] = lambda x, u, t, a: [x[1], u[0] - 1.5]
>>> ocp.running_costs[0] = lambda x, u, t, a: u[0]
>>> ocp.terminal_constraints[0] = lambda xf, tf, x0, t0, a: [xf[0], xf[1]]
>>> ocp.x00[0] = [10, -2]
>>> ocp.lbu[0] = 0; ocp.ubu[0] = 3
>>> ocp.lbtf[0] = 3; ocp.ubtf[0] = 5
>>> opt = mp.mpopt_h_adaptive(ocp, n_segments=3, poly_orders=[2]*3)
>>> solution = opt.solve()
>>> post = opt.process_results(solution, plot=True)
```

__init__(*problem: OCP, n_segments: int = 1, poly_orders: List[int] = [9], scheme: str = 'LGR', **kwargs*)

Initialize the optimizer :param n_segments: number of segments in each phase :param poly_orders: degree of the polynomial in each segment :param problem: instance of the OCP class

compute_seg_width_based_on_input_slope(*solution*)

Compute the optimum segment widths based on slope of the control signal.

:param : solution: nlp solution as reported by the solver

Returns

segment_widths: optimized segment widths based on present solution

compute_seg_width_based_on_residuals(*solution, method: str = 'merge_split'*)

Compute the optimum segment widths based on residual of the dynamics in each segment.

:param : solution: nlp solution as reported by the solver

Returns

segment_widths: optimized segment widths based on present solution

static compute_segment_widths_at_times(*times, n_segments, t0, tf*)

Compute seg_width fractions corresponding to given times and number of segments

static compute_time_at_max_values(*t_grid, t_orig, du_orig, threshold: float = 0*)

Compute the times corresponding to max value of the given variable (du_orig)

:param : t_grid: Fixed grid :param : t_orig: time corresponding to collocation nodes and variable (du_orig) :param : du_orig: Variable to decide the output times

Returns

time: Time corresponding to max, slope of given variable

static get_roots_wrt_equal_area(*residuals*, *n_segments*)

get_segment_width_parameters(*solution*, *options: Dict = {'method': 'residual', 'sub_method': 'merge_split'}*)

Compute optimum segment widths in every phase based on the given solution to the NLP

:param : solution: solution to the NLP :param : options: Dictionary of options if required (Computation method etc.)

method
Method used to refine the grid 'residual'

'merge_split' 'equal_area'

'control_slope'

Returns
seg_widths: Computed segment widths in a 1-D list (each phase followed by previous)

static merge_split_segments_based_on_residuals(*max_residuals*, *segment_widths*, *ERR_TOL: float = 0.001*)

Merge/Split existing segments based on residual errors

Merge consecutive segments with residual below tolerance

:param : max_residuals: max residual in dynamics of each segment :param : segment_widths: Segment width corresponding to the residual

Returns
segment_widths: Updated segment widths after merging/splitting

refine_segment_widths_based_on_residuals(*residuals*, *segment_widths*, *ERR_TOL: float = 0.001*, *method: str = 'merge_split'*)

Refine segment widths based on residuals of dynamics

:param : residuals: residual matrix of dynamics of each segment :param : segment_widths: Segment width corresponding to the residual

Returns
segment_widths: Updated segment widths after refining

solve(*initial_solution: Dict = None*, *reinitialize_nlp: bool = False*, *solver: str = 'ipopt'*, *nlp_solver_options: Dict = {}*, *mpopt_options: Dict = {}*, *max_iter: int = 10*, ***kwargs*) → Dict

Solve the Nonlinear Programming problem

:param
[init_solution: Dictionary containing initial solution with keys] x or x0 - Initional solution for the nlp variables

:param
[reinitialize_nlp: (True, False)] True - Reinitialize NLP solver object False - Use already created object if available else create new one

:param
[nlp_solver_options: Options to be passed to the nlp_solver while creating] the solver object, not while solving (like initial conditions)

:param
[mpopt_options: Options dict for the optimizer] 'method': 'residual' or 'control_slope' 'sub_method': (if method is residual)

'merge_split' 'equal_area'

Returns
solution: Solution as reported by the given nlp_solver object

5.1.5 Adaptive grid refinement scheme-III

mpopt-adaptive class

class mpopt.mpopt.**mpopt_adaptive**(*problem: OCP, n_segments: int = 1, poly_orders: List[int] = [9], scheme: str = 'LGR', **kwargs*)

Bases: *mpopt*

Multi-stage Optimal control problem (OCP) solver which implements seg-widths as optimization variables and solves for them along with the optimization problem.

Examples :

```
>>> # Moon lander problem
>>> from mpopt import mp
>>> ocp = mp.OCP(n_states=2, n_controls=1, n_phases=1)
>>> ocp.dynamics[0] = lambda x, u, t: [x[1], u[0] - 1.5]
>>> ocp.running_costs[0] = lambda x, u, t: u[0]
>>> ocp.terminal_constraints[0] = lambda xf, tf, x0, t0: [xf[0], xf[1]]
>>> ocp.x00[0] = [10, -2]
>>> ocp.lbu[0] = 0; ocp.ubu[0] = 3
>>> ocp.lbtf[0] = 3; ocp.ubtf[0] = 5
>>> opt = mp.mpopt_adaptive(ocp, n_segments=3, poly_orders=[2]*3)
>>> solution = opt.solve()
>>> post = opt.process_results(solution, plot=True)
```

__init__(*problem: OCP, n_segments: int = 1, poly_orders: List[int] = [9], scheme: str = 'LGR', **kwargs*)

Initialize the optimizer :param n_segments: number of segments in each phase :param poly_orders: degree of the polynomial in each segment :param problem: instance of the OCP class

create_solver(*solver: str = 'ipopt', options: Dict = {}*) → None

Create NLP solver

:param
[solver: Optimization method to be used in nlp_solver (List of plugins] avaiable at http://casadi.sourceforge.net/v2.0.0/api/html/d6/d07/classcasadi_1_1NlpSolver.html)

:param
[options: Dictionary] List of options for the optimizer (Based on CasADi documentation)

Returns
None

Updates the nlpsolver object in the present optimizer class

discretize_phase(*phase: int*) → Tuple

Discretize single phase of the Optimal Control Problem

Parameters
phase – index of the phase (starting from 0)

returns :
Tuple : Constraint vector (G, Gmin, Gmax) and objective function (J)

`get_nlp_constrains_for_segment_widths`(*phase: int = 0*) → Tuple

Add additional constraints on segment widths to the original NLP

`get_nlp_variables`(*phase: int = 0*)

Retrieve optimization variables and their bounds for a given phase

:param : phase: index of the phase (starting from 0)

Returns

(Z, Zmin, Zmax)
Z - Casadi SX vector containing optimization variables for the given phase (X, U, t0, tf) Zmin - Lower bound for the variables in 'Z' Zmax - Upper bound for the variables in 'Z'

Return type
Tuple

`init_solution_per_phase`(*phase: int*) → ndarray

Initialize solution vector at all collocation nodes of a given phase.

The initial solution for a given phase is estimated from the initial and terminal conditions defined in the OCP. Simple linear interpolation between initial and terminal conditions is used to estimate solution at interior collocation nodes.

:param : phase: index of phase

Returns
initialized solution for given phase

Return type
solution

`init_trajectories`(*phase: int = 0*) → Function

Initialize trajectories of states, constrols and time variables

:param : phase: index of the phase

Returns

trajectories: CasADi function which returns states, controls and time

variable for the given phase when called with NLP solution vector of all phases

t0, tf - unscaled AND x, u, t - scaled trajectories

`process_results`(*solution*, *plot: bool = True*, *scaling: bool = False*, *residual_x: bool = False*, *residual_dx: bool = True*)

Post process the solution of the NLP

:param : solution: NLP solution as reported by the solver :param : plot: bool

True - Plot states and variables in a single plot with states in a subplot and controls in another. False - No plot

:param
[scaling: bool] True - Plot the scaled variables False - Plot unscaled variables meaning, original solution to the problem

:param
[residuals: bool] To plot norma of the residual in the dynamics evaluated at points different from collocation nodes

Returns
post: Object of post_process class (Initialized)

solve(*initial_solution: Dict = None*, *reinitialize_nlp: bool = False*, *solver: str = 'ipopt'*, *nlp_solver_options: Dict = {}*, *mpopt_options: Dict = {}*, ***kwargs*) → Dict

Solve the Nonlinear Programming problem

:param
[init_solution: Dictionary containing initial solution with keys] x or x0 - Initional solution for the nlp variables

:param
[reinitialize_nlp: (True, False)] True - Reinitialize NLP solver object False - Use already created object if available else create new one

:param
[nlp_solver_options: Options to be passed to the nlp_solver while creating] the solver object, not while solving (like initial conditions)

:param : mpopt_options: Options dict for the optimizer

Returns
solution: Solution as reported by the given nlp_solver object

5.1.6 Processing results

Post-processing class

class mpopt.mpopt.**post_process**(*solution: Dict = {}*, *trajectories: List = None*, *options: Dict = {}*)

Bases: object

Process the results of mpopt optimizer for further processing and interactive
visualization

This class contains various methods for processing the solution of OCP

Examples

```
>>> post = post_process(solution, trajectories, options)
```

__init__(*solution: Dict = {}*, *trajectories: List = None*, *options: Dict = {}*)

Initialize the post processor object

:param
[solution: Dictionary (x0, ...)] x0 - Solution to the NLP variables (all phases)

:param
[trajectories: casadi function which returns (x, u, t, t0, tf) given solution] x - scaled states u - scaled controls t - unscaled time corresponding to x, u (unscaled for simplicity in NLP transcription) t0 - scaled initial times tf - scaled final times

:param
[options: Dictionary]

Essential information related to OCP, post processing and nlp optimizer
are stored in this dictionary

`get_data`(*phases: List = []*, *interpolate: bool = False*)

Get solution corresponding to given phases (original/interpolated)

:param : phases: List of phase indices :param : interpolate: bool

True - Interpolate the original grid (Refine the grid for smooth plot) False - Return original data

Returns

(x, u, t, a)
x - interpolated states u - interpolated controls t - interpolated time grid

Return type
Tuple

`get_interpolated_data`(*phases*, *taus: List = []*)

Interpolate the original solution across given phases

:param : phases: List of phase indices :param : taus: collocation grid points across which interpolation is performed

Returns

(x, u, t, a)
x - interpolated states u - interpolated controls t - interpolated time grid

Return type
Tuple

`static get_interpolated_time_grid`(*t_orig*, *taus: ndarray*, *poly_orders: ndarray*, *tau0: float*, *tau1: float*)

Update the time vector with the interpolated grid across each segment of the original optimization problem

:param : t_orig: Time grid of the original optimization problem (unscaled/scaled) :param : taus: grid of the interpolation taus across each segment of the original OCP :param : poly_orders: Order of the polynomials across each segment used in solving OCP

Returns
time: Interpolated time grid

`get_interpolation_taus`(*n: int = 75*, *taus_orig: ndarray = None*, *method: str = 'uniform'*)

Nodes across the normalized range [0, 1] or [-1, 1], to interpolate the data for post processing such as plotting

:param : n: number of interpolation nodes :param : taus_orig: original grid across which interpolation is to be performed :param : method: ("uniform", "other")

"uniform" : returns equally spaced grid points "other": returns mid points of the original grid recursively until 'n' elements

Returns
taus: interpolation grid

static get_non_uniform_interpolation_grid(*taus_orig*, *n: int = 75*)

Increase the resolution of the given taus preserving the sparsity of the given grid

:param : taus_orig: original grid to be refined :param : n: max number of points in the refined grid.

Returns
taus: refined grid

get_original_data(*phases: List = []*)

Get optimized result for multiple phases

:param : phases: Optional, List of phases to retrieve the data from.

Returns

(x, u, t, a)
x - states u - controls t - corresponding time vector

Return type
Tuple

get_trajectories(*phase: int = 0*)

Get trajectories of states, controls and time vector for single phase

:param : phase: index of the phase

Returns

(x, u, t, a)
x - states u - controls t - corresponding time vector

Return type
Tuple

classmethod plot_all(*x*, *u*, *t*, *tics: str = None*, *fig=None*, *axs=None*, *legend: bool = True*, *name: str = ''*)

Plot states and controls

:param : x: states data :param : u: controls data :param : t: time grid

Returns

(fig, axs)
fig - handle to the figure obj. of plot (matplotlib figure object) axs - handle to the axis of plot (matplotlib figure object)

Return type
Tuple

static plot_curve(*ax*, *x*, *t*, *name=None*, *ylabel=''*, *tics=['-', '-', '-', '-', '-', '-', '-', '-', '-', '-', '-', '-', '-', '-', '-', '-', '-', '-', '-', '-']*, *legend_index=None*)

2D Plot given data (x, y)

:param : ax: Axis handle of matplotlib plot :param : x: y-axis data - numpy ndarray with first dimention having data :param : t: x-axis data

plot_phase(*phase: int = 0*, *interpolate: bool = True*, *fig=None*, *axs=None*)

Plot states and controls across given phase

:param : phase: index of phase :param : interpolate: bool

True - Plot refined data False - Plot original data

Returns

(fig, axs)
fig - handle to the figure obj. of plot (matplotlib figure object) axs - handle to the axis of plot (matplotlib figure object)

Return type
Tuple

plot_phases(*phases: List = None*, *interpolate: bool = True*, *residuals: bool = True*, *fig=None*, *axs=None*, *tics: List = ['-', '-', '-', '-', '-', '-', '-', '-', '-', '-', '-', '-', '-', '-', '-', '-']*, *name: str = ''*)

Plot states and controls across given phases

:param : phases: List of indices corresponding to phases :param : interpolate: bool

True - Plot refined data False - Plot original data

Returns

(fig, axs)
fig - handle to the figure obj. of plot (matplotlib figure object) axs - handle to the axis of plot (matplotlib figure object)

Return type
Tuple

classmethod plot_residuals(*time*, *residuals*, *phases: List = [0]*, *name=None*, *fig=None*, *axs=None*, *tics=['.', '.', '.', '.', '.', '.', '.', '.', '.', '.', '.', '.', '.', '.', '.', '.']*)

Plot residual in dynamics

classmethod plot_single_variable(*var_data*, *t*, *dims*, *name: str = None*, *ylabel: str = None*, *axis=1*, *fig=None*, *axs=None*, *tics=['-', '-', '-', '-', '-', '-', '-', '-', '-', '-', '-', '-', '-', '-', '-', '-']*)

Plot given numpy array in various subplots

:param : var_data: input data to be plotted :param : t: xdata of the plot :param : dims: List of dimentions of the data to be plotted (List of list indicates

each internal list plotted in respective subplot)

:param : name: Variable name :param : ylabel: ylabel for the plot :param : axis: int, 0 or 1

0 - subplots are stacked horizintally 1 - subplots are stacked vertically

Returns

(fig, axs)
fig - handle to the figure obj. of plot (matplotlib figure object) axs - handle to the axis of plot (matplotlib figure object)

Return type
Tuple

plot_u(*dims: List = None*, *phases: List = None*, *axis: int = 1*, *interpolate: bool = True*, *fig=None*, *axs=None*, *tics=None*, *name='control'*, *ylabel='Control variables'*)

Plot given dimenstions of controls across given phases stacked vertically or horizontally

:param
[dims: List of dimentions of the control to be plotted (List of list indicates] each internal list plotted in respective subplot)

:param : phases: List of phases to plot :param : interpolate: bool

True - Plot refined data False - Plot original data

Returns

(fig, axs)
fig - handle to the figure obj. of plot (matplotlib figure object) axs - handle to the axis of plot (matplotlib figure object)

Return type
Tuple

`plot_x`(*dims: List = None*, *phases: List = None*, *axis: int = 1*, *interpolate: bool = True*, *fig=None*, *axs=None*, *tics=['-', '-', '-', '-', '-', '-', '-', '-', '-', '-', '-', '-', '-', '-', '-', '-']*)

Plot given dimenstions of states across given phases stacked vertically or horizontally

:param
[dims: List of dimentions of the state to be plotted (List of list indicates] each internal list plotted in respective subplot)

:param : phases: List of phases to plot :param : interpolate: bool

True - Plot refined data False - Plot original data

Returns

(fig, axs)
fig - handle to the figure obj. of plot (matplotlib figure object) axs - handle to the axis of plot (matplotlib figure object)

Return type
Tuple

`static sort_residual_data`(*time*, *residuals*, *phases: List = [0]*)

Sort the given data corresponding to plases

Next steps: Developer notes

CHAPTER

SIX

DEVELOPER NOTES

TODO

CHAPTER

SEVEN

RESOURCES

- Detailed implementation aspects of MPOPT are part of the master thesis.
- Quick introduction presentation.
- List of solved examples
- Features of MPOPT in *Jupyter Notebooks*

A pdf version of this documentation can be downloaded from PDF document

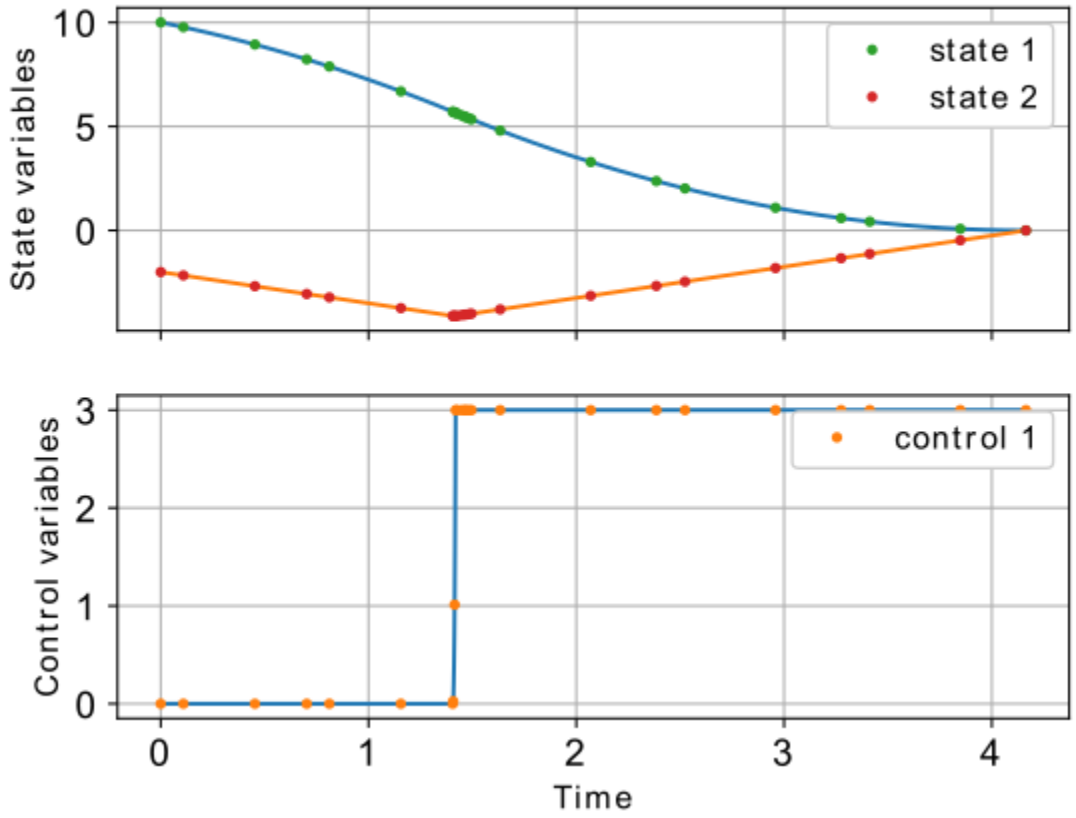
A must read Jupyter notebook on MPOPT features *Getting Started*

CHAPTER

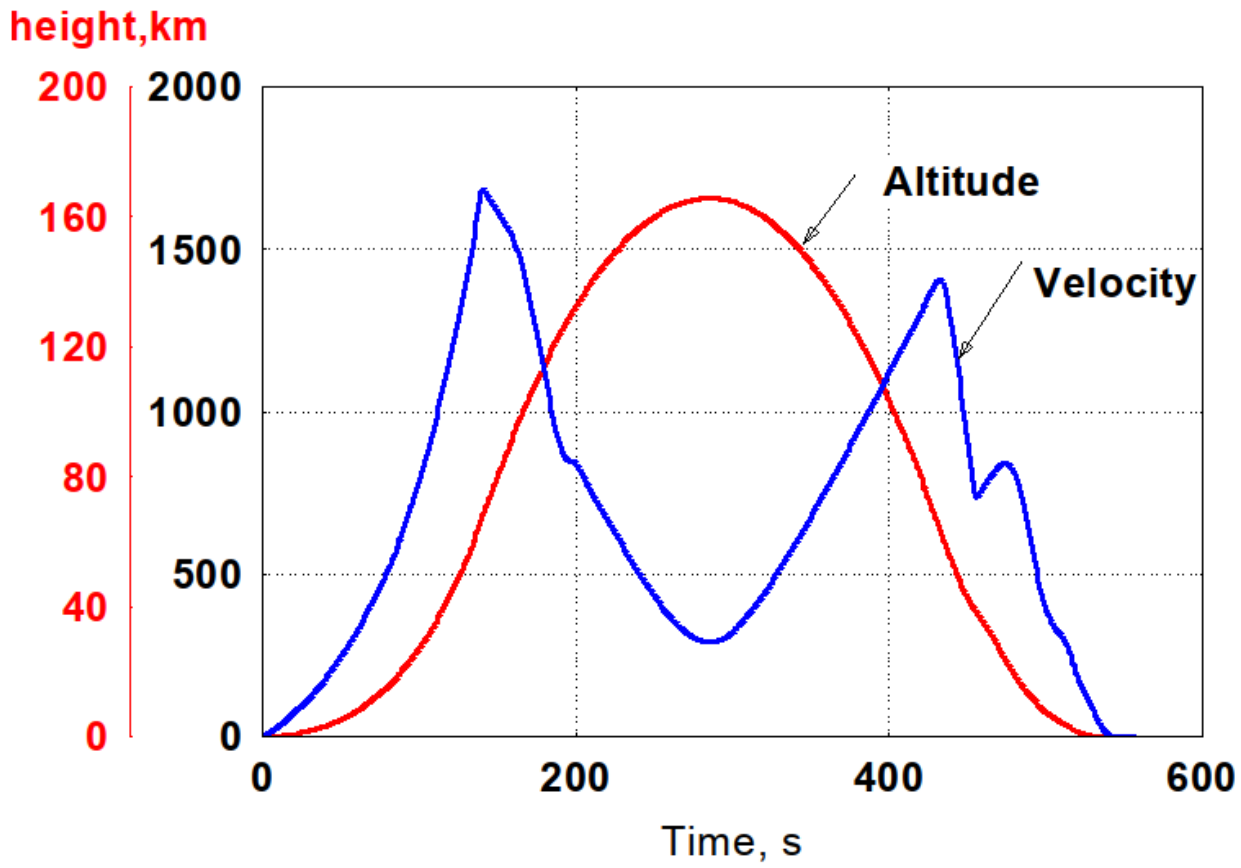
EIGHT

CASE STUDIES

- Quick demo of the solver using simple moon-lander fuel minimization OCP (bang-bang type control), refer *Quick features demo notebook* for more details. The image below shows the optimal altitude and the velocity profile (states) along with the optimal throttle (controls) to get minimum fuel trajectory to land on the Moon.



- A complex real-world example of The SpaceX falcon9 rocket orbital launch with the booster recovery results are shown below. OCP is defined to find the optimal trajectory and the thrust profile for booster return, refer *SpaceX Falcon9 booster recovery notebook* for more details. The first image below is the MPOPT solution using adaptive mesh and the second one is the real-time data of the SpaceX Falcon9 launch of NROL76 mission. The ballistic altitude profile of the booster is evident in both MPOPT solution and the real-time telemetry. Further, the MPOPT velocity solution compares well with the real-time data even though the formulation is only a first order representation of the actual booster recovery problem.


height,km
200
160
120
80
40
0
2000
1500
1000
500
0
Altitude
Velocity
0
200
400
600
Time, s

CHAPTER

NINE

FEATURES AND LIMITATIONS

While MPOPT is able to solve any Optimal control formulation in the Bolza form, the present limitations of MPOPT are,

- Only continuous functions and derivatives are supported
- Dynamics and constraints are to be written in CasADi variables (Familiarity with casadi variables and expressions is expected)
- The adaptive grid though successful in generating robust solutions for simple problems, doesnt have a concrete proof on convergence.

CHAPTER

TEN

AUTHORS

- **Devakumar THAMMISETTY**
- **Prof. Colin Jones** (Co-author)

CHAPTER

ELEVEN

LICENSE

This project is licensed under the GNU LGPL v3 - see the LICENSE file for details

CHAPTER

TWELVE

ACKNOWLEDGMENTS

- **Petr Listov**

CHAPTER

THIRTEEN

CITE

- D. Thammisetty, "Development of a multi-phase optimal control software for aerospace applications (mpopt)," Master's thesis, Lausanne, EPFL, 2020.

BibTex entry:

@mastersthesis{thammisetty2020development,
title={Development of a multi-phase optimal control software for aerospace applications (mpopt)}, author={Thammisetty, Devakumar}, year={2020}, school={Master's thesis, Lausanne, EPFL}}

CHAPTER

FOURTEEN

INDICES AND TABLES

- genindex
- modindex
- search

INDEX

Symbols

C

D

G

H

I

L

M

O

P

R

S

T

U

V