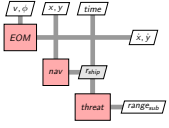
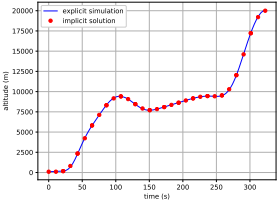
Optimizing Trajectories with Dymos

Rob Falck

October 29, 2019





Dymos is an open-source library for modeling dynamic systems within OpenMDAO

- Support for multiple optimal control techniques
- Doesn't impose the trajectory optimization as the "top-level" problem
  - Treats the trajectory as on component of a larger system
  - Design a system to satisfy multiple design reference trajectories
- Find the best system to fly a trajectory, and the best trajectory that can be flown by the system

Dymos leverages the strengths of OpenMDAO

- Analytic derivatives and automatic detection of sparsity patterns
  - Adjoint differentiation tends to be faster for shooting methods
  - Pseudospectral methods tend to be faster in forward mode
  - Bidirectional sparsity for problems which would "break" pure forward or reverse sparsity
- Parallelization of expensive models via MPI
- Access to OpenMDAO solvers within the ODE

Dymos Numerical Methods

- Pseudospectral methods
  - High-Order Gauss-Lobatto (OTIS)
  - Radau Pseudospectral Method (GPOPS, DIDO)
- Shooting methods
  - RK4
  - Solver-based pseudospectral methods

Dymos Numerical Methods

- Pseudospectral methods
  - High-Order Gauss-Lobatto (OTIS)
  - Radau Pseudospectral Method (GPOPS, DIDO)
- Shooting methods
  - RK4
  - Solver-based pseudospectral methods

The problem solved by Dymos

- Minimize/maximize some objective quantity subject to:
  - Dynamics (provided by an OpenMDAO system)
  - Dynamic controls
  - Static Controls (design parameters)
  - Boundary Constraints
  - Path Constraints

The Dymos UI is heavily influenced by OTIS

- A *trajectory* is broken into one or more *phases*
- In each *phase* states and controls are represented on one or more polynomial *segments* in time
- Phases can be linked via constraint (supports parallel execution) or connected in series
- Phase linkages are very general, supporting branched trajectories

Notable features of Dymos

- Vectorized states, controls, and parameters
- Access to the first and second time derivatives of controls
- Output available on arbitrary grids

Goals for today

- Solve an optimal control problem with Dymos
- Learn what to do when things don't go well

Dymos by Example

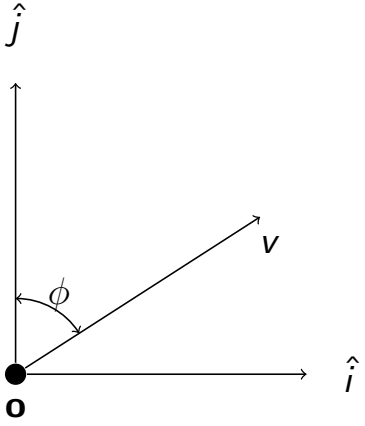
The Enemy Submarine Problem

An enemy submarine determined to sink your ship is sitting, torpedoes armed, exactly halfway between you and your home port. The submarine submerges to some fixed depth, and your ship now has no further information about its position. The enemy sub has to be directly underneath your ship to sink it, but the sub can track your moves with precision and respond efficiently. If your ship is fast enough, though, you will be able to set a wide course around the sub and reach port safely.

How much faster than the sub does your ship have to be to guarantee you can avoid the sub and get home?

Source: `https://fivethirtyeight.com/features/damn-the-torpedoes-two-puzzles-ahead/`

The Enemy Submarine Problem: ODE



$$\dot{x} = v \sin \phi$$
$$\dot{y} = v \cos \phi$$

The Enemy Submarine Problem: ODE

- Give the sub a speed of 1 (unitless).
- The sub can, at best, be *time* units in distance away from its starting point

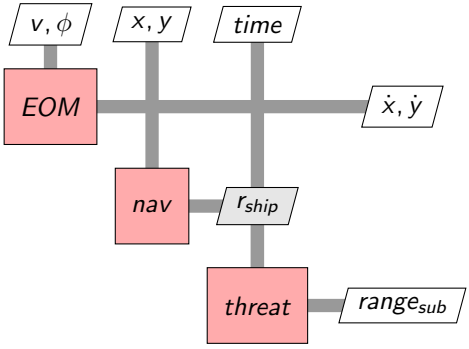
$$r_{sub} = time$$

The Enemy Submarine Problem: ODE

- Path constraint: To avoid the sub, our ship must be greater than r_{sub} units away from its origin at any given time.

$$r_{ship} = \sqrt{x^2 + y^2}$$
$$r_{ship} - r_{sub} \geq 0$$

The Donner Sub Problem: Notional ODE XDSM



The Donner Sub Problem: ODE

```
import openmdao.api as om

from nav_comp import NavComp
from ship_eom_comp import ShipEOMComp

class DonnerSubODE(om.Group):

    def initialize(self):
        self.options.declare('num_nodes', types=int)

    def setup(self):
        nn = self.options['num_nodes']

        self.add_subsystem('eom_comp', ShipEOMComp(num_nodes=nn))
        self.add_subsystem('nav_comp', NavComp(num_nodes=nn))
        self.add_subsystem('threat_comp', om.ExecComp('sub_range = r_ship - time',
                                                      sub_range={'shape': (nn,)},
                                                      r_ship={'shape': (nn,)},
                                                      time={'shape': (nn,)}))

        self.connect('nav_comp.r_ship', 'threat_comp.r_ship')
```

The Donner Sub Problem: ODE

```
1  import openmdao.api as om
2
3  from nav_comp import NavComp
4  from ship_eom_comp import ShipEOMComp
5
6  class DonnerSubODE(om.Group):
7
8      def initialize(self):
9          self.options.declare('num_nodes', types=int)
10
11     def setup(self):
12         nn = self.options['num_nodes']
13
14         self.add_subsystem('eom_comp', ShipEOMComp(num_nodes=nn))
15         self.add_subsystem('nav_comp', NavComp(num_nodes=nn))
16         self.add_subsystem('threat_comp', om.ExecComp('sub_range = r_ship - time',
17                                                       sub_range={'shape': (nn,)},
18                                                       r_ship={'shape': (nn,)},
19                                                       time={'shape': (nn,)}))
20
21         self.connect('nav_comp.r_ship', 'threat_comp.r_ship')
```

All ODE models accept num_nodes

The Donner Sub Problem: ODE

```
1  import openmdao.api as om
2
3  from nav_comp import NavComp
4  from ship_eom_comp import ShipEOMComp
5
6  class DonnerSubODE(om.Group):
7
8      def initialize(self):
9          self.options.declare('num_nodes', types=int)
10
11     def setup(self):
12         nn = self.options['num_nodes']
13
14         self.add_subsystem('eom_comp', ShipEOMComp(num_nodes=nn))
15         self.add_subsystem('nav_comp', NavComp(num_nodes=nn))
16         self.add_subsystem('threat_comp', om.ExecComp('sub_range = r_ship - time',
17                                                       sub_range={'shape': (nn,)},
18                                                       r_ship={'shape': (nn,)},
19                                                       time={'shape': (nn,)}))
20
21         self.connect('nav_comp.r_ship', 'threat_comp.r_ship')
```

Size ExecComp variables appropriately ⟶

Setting up the problem

- One phase (single ODE, no intermediate boundary constraints)
- Two state variables (x, y)
- One design parameter (v)
- One dynamic control (ϕ)
- Objective: minimize v

Setting up the problem: Create the problem, trajectory, and phase

```
import openmdao.api as om
import dymos as dm

from donner_sub_ode import DonnerSubODE

# Create the problem
p = om.Problem(model=om.Group())

# Add the trajectory (optional for single phase problems)
traj = dm.Trajectory()

# Create the phase
phase = dm.Phase(ode_class=DonnerSubODE,
                 transcription=dm.GaussLobatto(num_segments=20, order=3, compressed=False))

# Add the phase to the trajectory, and the trajectory to the model
traj.add_phase('phase0', phase=phase)
p.model.add_subsystem('traj', traj)
```

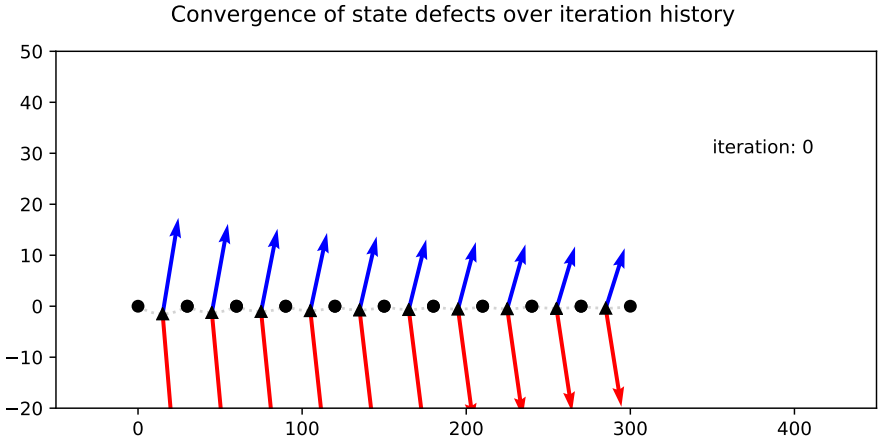
What is a transcription, anyway?

- Converts a continuous optimal control problem into a discrete NLP problem
- Key feature: The grid (or mesh), which controls the accuracy of the integration.
  - More (or higher order) segments gives more accuracy at greater expense

High-Order Legendre Gauss Lobatto Transcription

- Developed by Herman and Conway
- Generalization of the older Hermite-Simpson method pioneered by Dickmanns and Wells and later Paris and Hargraves.

High-Order Legendre Gauss Lobatto Transcription



Radau Pseudospectral Method

- Rao, Patterson, and Garg, et.al. at Univeristy of Florida.
- More variables than Gauss-Lobatto but only a single evaluation of the ODE per iteration.

Radau Pseudospectral Transcription