Introduction to Dido:

Solving Optimal Control Problems in MatLab

Harleigh Marsh

Applied Mathematics & Statistics Department
University of California Santa Cruz

users.soe.ucsc.edu/~hcmarsh/
github.com/harleigh
hcmarsh@soe.ucsc.edu

April 29, 2015

Given state $\mathbf{x} \in \mathbb{R}^{N_x}$ and control $\mathbf{u} \in \mathbb{R}^{N_u}$, where $N_x, N_u \in \mathbb{N}$, determine the state-control function pair $\{\mathbf{x}(\cdot), \mathbf{u}(\cdot)\}$ and (possibly) the clock time events $\{t_0, t_f\}$ to minimize the Bolza **cost functional**

$$J\left[\mathbf{x}(\cdot), \mathbf{u}(\cdot), t_0, t_f\right] = E(\mathbf{x}(t_0), \mathbf{x}(t_f), t_0, t_f) + \int_{t_0}^{t_f} F(\mathbf{x}(t), \mathbf{u}(t), t)\, dt$$

Subject to:

Dynamic Constraints: for all time $t \in [t_0, t_f]$

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

Endpoint Constraints:

$$\mathbf{e}^L \leq \mathbf{e}(\mathbf{x}(t_0), \mathbf{x}(t_f), t_0, t_f) \leq \mathbf{e}^U$$

Path Constraints (state and control):

$$\mathbf{h}^L \leq \mathbf{h}(\mathbf{x}(t), \mathbf{u}(t), t) \leq \mathbf{h}^U$$

Box Constraints on State Control and Clock Times:

$$\mathbf{x}^L \leq \mathbf{x}(t) \leq \mathbf{x}^U$$
$$\mathbf{u}^L \leq \mathbf{u}(t) \leq \mathbf{u}^U$$
$$t_0^L \leq t_0 \leq t_0^U$$
$$t_f^L \leq t_f \leq t_f^U$$

The functions $E, F, \mathbf{f}, \mathbf{e}, \mathbf{h}$ are the **problem data**:

- Pontryagin's Principal only requires differentiability with respect to the first argument.
- Dido would like twice continuously differentiable with respect to all of their arguments.

Calling Dido: `[cost, primal, dual] = dido(problem, algorithm)`

Inputs:

- `problem`: Data structure that contains the problem formulation.
- `algorithm`: Defines the number of discretization points, a guess to the optimal solution, and whether to run Dido in Accurate Mode.

Outputs:

- `cost`: Minimized cost J produced from the candidate optimal solution found by Dido.
- `primal`: Data structure containing candidate optimal solution, and the discretization nodes.

$$\mathbf{x}(\cdot) = \texttt{primal.states}$$
$$\mathbf{u}(\cdot) = \texttt{primal.controls}$$

- `dual`: Data Structure containing Costates and Control Hamiltonian (along the candidate optimal solution).

The primal data structure is the main interface for entering the problem data, which is stored in the data structure problem. The fields of primal data structure must exactly match those given in this tutorial.

- Discretization nodes (points): primal.nodes $= [t_0, \ldots, t_N]$
- Telling Dido how many nodes you want to use: algorithm.nodes $= N_n$, where $N_n = N + 1$

$$\texttt{primal.states} = \begin{bmatrix} x_1(t_0) & \cdots & x_1(t_N) \\ \vdots & & \vdots \\ x_{N_x}(t_0) & \cdots & x_{N_x}(t_N) \end{bmatrix}$$

$$\texttt{primal.controls} = \begin{bmatrix} u_1(t_0) & \cdots & u_1(t_N) \\ \vdots & & \vdots \\ u_{N_u}(t_0) & \cdots & u_{N_u}(t_N) \end{bmatrix}$$

Note that time runs horizontally, so the first state component, x_1 is accessed by primal.states(1,:)

The problem data: $E, F, \mathbf{f}, \mathbf{e}, \mathbf{h}$, as Endpoint Cost, Running Cost, Dynamics, Endpoint Constraint, Path Constraint. Each are fed into Dido through function calls.

- Function `[E, F] = costFun(primal)`
- Function `[xDot] = dynamicsFun(primal)`
- Function `[e] = eventFun(primal)`
- Function `[h] = pathFun(primal)`

And are stored in fields of the primal data structure as:

- `primal.cost = 'costFun'`
- `primal.dynamics = 'dynamicsFun'`
- `primal.events = 'eventFun'`
- `primal.path = 'pathFun'`

The function names don't matter, only their inputs and outputs: Each function must only take in in the `primal` data structure and output only the following specified data fields.

The general directory structure to a Dido project, are separate files for each function, and one main file that sets the problem and algorithm data structures.

Calculates the cost functional J.

Inputs: The `primal` data structure.

Outputs: `[E, F]`

- `E` a 1 by 1 scalar representing the End Point cost calculation.
- `F` a 1 by N_n row vector representing the Running cost

Useful code-bits based on the structure of primal:

- $\mathbf{x}(t_0)$ = `primal.states(:,1)` a N_x by 1 column vector.
- $\mathbf{x}(t_f)$ = `primal.states(:,end)` a N_x by 1 column vector.
- t_0 = `primal.nodes(1)` a 1 by 1 scalar.
- t_f = `primal.nodes(end)` a 1 by 1 scalar.

No running cost? Set `F=0` where 0 is the 1 by 1 scalar zero. No Endpoint cost? Set `E=0` where 0 is the 1 by 1 scalar zero.

Calculates the dynamics, $\dot{\mathbf{x}}(t) = \mathbf{f}(\mathbf{x}(t), \mathbf{u}(t), t)$, at the time nodes chosen by Dido.

Inputs: The `primal` data structure.

Outputs: `xDot` a N_x by N_n matrix representing $\dot{\mathbf{x}}$ at the nodes chosen by Dido.

Useful code-bits based on the structure of primal:

- Accessing state element: $\mathbf{x_i}$ = `primal.states(i, :)`
- Accessing control element: $\mathbf{u_i}$ = `primal.controls(i, :)`
- Accessing the time nodes: `primal.nodes`

Forces Dido to satisfy the endpoint conditions while searching for a candidate optimal solution.

Inputs: The `primal` data structure.

Outputs: `e` a N_e by 1 column vector describing the boundary conditions to your problem.

Useful code-bits based on the structure of primal:

- $\mathbf{x}(t_0)$ = `primal.states(:,1)` a N_x by 1 column vector.
- $\mathbf{x}(t_f)$ = `primal.states(:,end)` a N_x by 1 column vector.
- t_0 = `primal.nodes(1)` a 1 by 1 scalar.
- t_f = `primal.nodes(end)` a 1 by 1 scalar.

Forces Dido to satisfy the path constraints at the chosen time nodes while searching for a candidate optimal solution.

Inputs: The `primal` data structure.

Outputs: `h` a N_h by N_n column vector that contains the calculated path constraints.

Useful code-bits based on the structure of primal:

- Accessing state element: $\mathbf{x_i}$ = `primal.states(i, :)`
- Accessing control element: $\mathbf{u_i}$ = `primal.controls(i, :)`
- Accessing the time nodes: `primal.nodes`

The **problem bounds**, separated into upper and lower, are the vector values: $\mathbf{x}^L, \mathbf{x}^U, \mathbf{u}^L, \mathbf{u}^U$, etc. These are specified in the main problem file to your project. Dido requires that all of the problem bounds are stored into one data structure with the following format (bolded part): I use the name 'bounds'.

- $\mathbf{x}^L$ is accessed by bounds.**lower.states**
- $\mathbf{x}^U$ is accessed by bounds.**upper.states**
- $\mathbf{u}^L$ is accessed by bounds.**lower.controls**
- $\mathbf{u}^U$ is accessed by bounds.**upper.controls**
- $\mathbf{e}^L$ is accessed by bounds.**lower.events**
- $\mathbf{e}^U$ is accessed by bounds.**upper.events**
- $\mathbf{h}^L$ is accessed by bounds.**lower.path**
- $\mathbf{h}^U$ is accessed by bounds.**upper.path**
- t_0^L, t_f^L are given by bounds.**lower.time**
- t_0^U, t_f^U are given by bounds.**upper.time**

Storing the bounds into the `problem` data structure, we must access the field 'bounds': `problem.bounds = bounds`

- Telling Dido the number of discretization samples to make: `algorithim.nodes` $= N_n$ where N_n should start around 16 for a first run.
- Dido can run in an accurate mode, though this is only advisable if you have success in non-accurate mode (which actually is quite accurate and is the default mode in which Dido is ran): `algorithim.mode = 'accurate'`
- Making a guess. All algorithms need a guess to get them started. Dido can make its own guess, but often we should supply our own guess. The guess has three fields: a guess each for states control and time. The number of guesses, N_g must be greater than or equal to 2.
  - `guess.states` must be a N_x by N_g matrix
  - `guess.controls` must be a N_u by N_g matrix
  - `guess.time` must be a 1 by N_g matrix

And, storing the guess into the `algorithm` data structure is given as `algorithim.guess = guess`

See my GitHub repository at `github.com/harleigh`

Goto the DidoProjects repo, and there will be two examples about the Brachistochrone problem, implemented in Dido. The code is full of documentation. Both solve the terminal end-point Brachistochrone problem but with two different formulations: Brach1 and Brach2 from Michael Ross' text.