OC Solver Suite Manual

Dan Ryan, `ryan@nimbios.org`

July 23, 2012

Overview

This package is designed so that an optimal control problem can be defined in a generic way so that various solvers can be invoked on the same problem. To solve an OC problem using this package the user should supply one Matlab function (the "make problem function") that defines the problem and one Matlab script (the "solve problem script") that declares parameter values (or a range of parameter values), calls the "make problem function" invokes one (or more) of the OC solver functions provided in this package and processes/graphs the results.

Currently, there are three solvers available:

- **`fb_sweep`**: This solver utilizes a forward-backward sweep method to compute the optimal control. It is the fastest solver when it works, but it will often fail to converge.
- **`single_shooting`**: This solver discretizes the control function using a piecewise linear basis the resulting nonlinear programming problem using a single shooting discretization. It is much more robust, but considerably slower than `fb_sweep`.
- **`bvp_solver`**: This solver utilizes the native Matlab boundary value problem solvers, `bvp5c` and `bvp4c`, to directly solve the optimality system. It performs exceptionally well when a good guess for the optimal states and adjoints OR a good guess for the optimal control is provided.

These solvers are designed to solve the problem:

$$\min_{\vec{u}(t)\in\mathcal{U}} J(\vec{u}) := \int_{T_0}^{T_F} f(t,\vec{x}(t),\vec{u}(t))\,dt \tag{1}$$

subject to

$$\frac{d\vec{x}}{dt} = \vec{g}(t,\vec{x},\vec{u}), \quad \vec{x}(0) = \vec{x}_0\,. \tag{2}$$

We use t for time, $\vec{x}$ for the state variable, $\vec{u}$ for the control function (vector valued if more than one control), and $\vec{\lambda}$ for the adjoint variable. In (1), $\mathcal{U}$ is such that each component of $\vec{u}$ is subject to a constant lower and upper bound (possibly $\pm\infty$). We define the Hamiltonian function, $\mathcal{H}$ by

$$\mathcal{H}(t,\vec{x},\vec{\lambda},\vec{u}) = f(t,\vec{x},\vec{u}) + \vec{\lambda}\cdot\vec{g}(t,\vec{x},\vec{u})\,. \tag{3}$$

Then, the adjoint variable $\vec{\lambda}$ is defined the by equation

$$\frac{d\vec{\lambda}}{dt} = -\nabla_{\vec{x}}\mathcal{H}(t,\vec{x},\vec{\lambda},\vec{u}), \quad \vec{\lambda}(0) = \vec{0}\,. \tag{4}$$

The "Make Problem Function"

The user should write a Matlab function file that returns a Matlab structure describing the optimal control problem. The problem structure, `prob`, must have fields that describe all the relevant data of the problem. Any problem parameters should be declared as global variables in this function, but no actual values need to be specified. This is done so that parameter values can be manipulated at run-time.

The following table shows which fields `prob` is required to have for the various solvers:

Field	**fb_sweep**	**single_shooting**	**bvp_solver**
objective	X	X	X
stateRHS	X	X	X
adjointRHS	X	X	X
ControlBounds	X	X	X
dHdu		X	
ControlChar	X		X
optJac			Optional

Table 1: Required fields for problem structure.

The table below describes each of these fields:

Field	**Type**	**Description**
`objective`	function handle that accepts $(t, \vec{x}, \vec{u})$	the objective integrand value f from (1)
`stateRHS`	function handle that accepts $(t, \vec{x}, \vec{u})$	the RHS of the state equation $\vec{g}$ from (2)
`adjointRHS`	function handle that accepts $(t, \vec{x}, \vec{\lambda}, \vec{u})$	the RHS of the adjoint equation $-\nabla_{\vec{x}}\mathcal{H}$ from (4)
`ControlBounds`	(#Controls) $\times$ 2 matrix	lower (first column) and upper (second column) bounds on each component of the control function
`dHdu`	function handle that accepts $(t, \vec{x}, \vec{\lambda}, \vec{u})$	the partial derivative (or gradient) of the Hamiltonian $\mathcal{H}$ with respect to $\vec{u}$
`ControlChar`	function handle that accepts $(t, \vec{x}, \vec{\lambda})$	a function that computes the optimal control in terms of the state and adjoint; found by applying Pontryagin's Min/Max Principle
`optJac`	function handle that accepts $(t, \vec{y})$	the Jacobian of the optimality system where $\vec{y} = \begin{pmatrix} \vec{x} \\ \vec{\lambda} \end{pmatrix}$

All of these function handles (except for `optJac`) need to be vectorized in their second component (the time dimension) so that if a row vector of times, `t`, is provided the function will return an array with number of columns equal to `length(t)`.

The "Solve Problem Script"

In addition to defining the problem with the "make problem function", the user should write a script that makes the problem and then invokes one or more of the solvers (as well as any desired post-processing of results or report generation). In this script, all parameters that were defined (via a global declaration) in the "make problem function" should be again declared global and values assigned. This can also be done interactively at the Matlab command prompt.

Description of Solvers

The solvers have all been standardized to return the same type of output, namely a Matlab structure (called `soln` from here on) with the following fields:

- `J`: the optimized objective value (scalar),
- `x`: the optimal state vector trajectory (function handle),
- `lam`: the optimal adjoint vector trajectory (function handle),
- `u`: the optimal control trajectory (function handle).

The fields `x`, `lam` and `u` have been vectorized to accept row vectors of time points at which you wish the function to be evaluated. If you only wish to have specified components returned (as opposed to the entire vector) use the auxiliary function `heval` (similar to `deval` but operates on any function handle, hence the "h"). For example if your state vector `soln.x` has 3 components but you only want the second and third values evaluated at the row vector `t`, use the command: `heval(soln.x, t, 2:3)`.

For all solvers, the optimal state and adjoint trajectories were computed post-optimization with a final sweep through the state and adjoint ODEs using the computed optimal control and the ODE solver `odevr7` (which outputs vector values). The vector values were then interpolated using piecewise-cubic Hermite interpolation (`'pchip'` mode) and the Matlab griddedInterpolant class. This class is new as of Matlab 2011b, so if you are getting errors regarding this unknown class, you need an updated version of Matlab. For indirect methods (`fb_sweep` and `bvp_solver`) the control is also interpolated via `'pchip'` at points evenly spaced across the time domain (and number of points specified by the option `nINTERP_PTS`).

`fb_sweep`

The syntax for calling `fb_sweep` is

```
soln = fb_sweep(prob, x0, tspan)
soln = fb_sweep(prob, x0, tspan, options)
```

where `prob` is a structure with the required fields from Table 1, `x0` is a vector of initial conditions for the states, `tspan` is a vector with `tspan(1)` $= T_0$ and `tspan(end)` $= T_F$. The optional argument `options` is a Matlab structure that is checked for the fields listed below and any values provided will be used instead of the default value:

Field	**Default**	**Description**
`uRelTol`	10^{-7}	relative tolerance for the change in u between sweeps
`uAbsTol`	10^{-7}	absolute tolerance for the change in u between sweeps
`RelTol`	5×10^{-14}	relative tolerance for `odevr7` ODE solver
`AbsTol`	5×10^{-14}	absolute tolerance for `odevr7` ODE solver
`nSWEEPS`	50	maximum number of sweeps before terminating
`nERROR_PTS`	1001	number of points where the change in u is calculated
`nINTERP_PTS`	1001	number of points used to interpolate the optimal control vector into a function
`u0`	$\vec{u}(t) \equiv \vec{u}_{\min}$	initial guess for control; can be a vector of equally spaced control values or a function handle

Note that if the sweep fails to converge before `nSWEEPS` is reached, it will return an empty structure for `soln`. If need be, you can test if the `soln` returned by `fb_sweep` is not empty using `isfield(soln, 'u')`.

`single_shooting`

The syntax to call `single_shooting` is

```
soln = single_shooting(prob, x0, controlPtArray)
soln = single_shooting(prob, x0, controlPtArray, options)
```

where `prob` is a Matlab structure with the required fields listed in Table 1, `x0` is a vector of initial conditions for the state and `controlPtArray` is a row vector of points used to form the piecewise linear basis functions for the control.

It is required that `controlPtArray(1)` $= T_0$ and `controlPtArray(end)` $= T_F$. The basis functions used to discretize u will be piecewise linear tent functions $\{\phi_i(t)\}_{i=1}^{k}$ ($k =$ `length(controlPtArray)`) such that $\phi_i(t)$ is one at `controlPtArray(i)` and decreases linearly to zero at `controlPtArray(i-1)` and `controlPtArray(i+1)`.

The `options` structure supports the fields listed below

Field	**Default**	**Description**
`TolX`	10^{-6}	step size tolerance for `fmincon` NLP algorithm
`TolFun`	10^{-5}	tolerance for size of objective gradient in `fmincon`
`RelTol`	5×10^{-14}	relative tolerance for `odevr7` ODE solver
`AbsTol`	5×10^{-14}	absolute tolerance for `odevr7` ODE solver
`MinMax`	`'Min'`	set to `'Max'` if maximization of objective is desired
`Algorithm`	`'sqp'`	algorithm used in `fmincon`
`u0`	$\vec{u}(t) \equiv \vec{u}_{\min}$	initial guess for control; can be a vector of values of u at `controlPtArray` or a function handle

bvp_solver

The syntax to call `bvp_solver` is

```
soln = bvp_solver(prob, x0, tspan)
soln = bvp_solver(prob, x0, tspan, options)
```

where `prob` is the Matlab problem structure with fields corresponding to Table 1, `x0` is the state initial values, and `tspan` is a vector with `tspan(1)` $= T_0$ and `tspan(end)` $= T_F$.

The `options` structure supports the following fields

Field	**Default**	**Description**
`bvpRelTol`	10^{-7}	relative tolerance for the chosen Matlab bvp solver
`bvpAbsTol`	10^{-7}	absolute tolerance for the bvp solver
`RelTol`	5×10^{-14}	relative tolerance for `odevr7` ODE solver
`AbsTol`	5×10^{-14}	absolute tolerance for `odevr7` ODE solver
`MAX_MESH_SIZE`	$50,000$	maximum number of mesh points for the bvp solver
`nINTERP_PTS`	1001	number of interpolation points used to build the optimal control function
`Solver`	`'bvp5c'`	Matlab solver to use; can change to `'bvp4c'` if desired
`InitMesh`	`linspace(T0, TF, 11)`	initial mesh for the bvp solver
`y0`	$\begin{pmatrix} \vec{x}_0 \\ \vec{0} \end{pmatrix}$	initial guess for the state/adjoint vector $\vec{y}$; can be a constant vector or function handle
`u0`	$\vec{u}(t) \equiv \vec{u}_{\min}$	initial guess for control; only used if `y0` is not provided; must be a function handle

Notes On Auxiliary Functions

This suite of OC solvers makes extensive use of the ODE solver `odevr7` authored by Larry Shampine and distributed with this suite. It is a vectorized Runge-Kutta(7,8) routine that uses control of error and residuals at intermediate points between time steps. It is very efficient for solving ODEs with stringent error tolerances.

Two other helper functions are

```
[x, lam, J] = compute_x_lam_J(prob, x0, tspan, u, RelTol, AbsTol)
[x, lam] = compute_x_lam(prob, x0, tspan, u, RelTol, AbsTol)
```

which both take problem information and a control function `u` (a function handle) as input and return computed states, adjoints (and objective value `J` if specified). These may be of interest as they can easily compute values for non-optimal controls if desired.

The function `heval` is useful if you want to plot or access individual components of the state, adjoint or vector valued control. It is called as

```
values = heval(f, t, range)
```

where `f` is a vector valued function handle, `t` is a row vector of times where you wish to evaluate `f` and `range` is an array specifying which components of `f` you want returned. The output will have `size(values)= (length(range), length(t)).`

The function `vectorInterpolant` effectively extends the griddedInterpolant class to vector valued functions. Its syntax is

```
f = vectorInterpolant(t, v, mode)
```

where `t` is a row vector of times, `v` is an array with vector values for each point in `t` and `mode` is a string specifying which interpolation mode to use (eg `'nearest', ... 'linear', 'pchip', 'spline'`) and `f` is a function handle for the interpolation function.