HAL
archives-ouvertes.fr

SHOOT2.0: An indirect grid shooting package for optimal control problems, with switching handling and embedded continuation

Pierre Martinon, Joseph Gergaud

▶ To cite this version:

Pierre Martinon, Joseph Gergaud. SHOOT2.0: An indirect grid shooting package for optimal control problems, with switching handling and embedded continuation. [Research Report] RR-7380, INRIA. 2010, pp.30. <inria-00517515v2>

HAL Id: inria-00517515
https://hal.inria.fr/inria-00517515v2

Submitted on 15 Sep 2010

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

SHOOT2.0: An indirect grid shooting package for optimal control problems, with switching handling and embedded continuation

Pierre Martinon — Joseph Gergaud

N° 7380

09/2010

Modeling, Optimization, and Control of Dynamic Systems

ISSN 0249-6399 ISRN INRIA/RR--7380--FR+ENG

INSTITUT NATIONAL
DE RECHERCHE
EN INFORMATIQUE
ET EN AUTOMATIQUE

INRIA

centre de recherche
SACLAY - ÎLE-DE-FRANCE

SHOOT2.0: An indirect grid shooting package for optimal control problems, with switching handling and embedded continuation

Pierre Martinon* , Joseph Gergaud†

Theme : Modeling, Optimization, and Control of Dynamic Systems
Applied Mathematics, Computation and Simulation
Équipe-Projet Commands

Rapport de recherche n° 7380 — 09/2010 — 27 pages

Abstract: The SHOOT2.0 package implements an indirect shooting method for optimal control problems. It is specifically designed to handle control discontinuities, with an automatic switching detection that requires no assumptions concerning the number of switchings. Special care is also devoted to the computation of the Jacobian matrix of the shooting function, using the variational system instead of classical finite differences. The package also features an embedded continuation method and an automatic (parallel) grid shooting in order to reduce the dependency to the initialization.

Key-words: optimal control, shooting method, Pontryagin's Principle, control switchings

* INRIA Saclay and CMAP, Ecole Polytechnique, **pierre.martinon@inria.fr**
† Université de Toulouse, **joseph.gergaud@enseeiht.fr**

Centre de recherche INRIA Saclay – Île-de-France
Parc Orsay Université
4, rue Jacques Monod, 91893 ORSAY Cedex
Téléphone : +33 1 72 92 59 00

SHOOT2.0: An indirect grid shooting package for optimal control problems, with switching handling and embedded continuation

Résumé : The SHOOT2.0 package implements an indirect shooting method for optimal control problems. It is specifically designed to handle control discontinuities, with an automatic switching detection that requires no assumptions concerning the number of switchings. Special care is also devoted to the computation of the Jacobian matrix of the shooting function, using the variational system instead of classical finite differences. The package also features an embedded continuation method and an automatic (parallel) grid shooting in order to reduce the dependency to the initialization.

Mots-clés : optimal control, shooting method, Pontryagin's Principle, control switchings

Contents

Introduction

The SHOOT2.0 package implements an indirect shooting method for optimal control problems. It is specifically designed to handle control discontinuities, with an automatic switching detection that requires no assumptions concerning the number of switchings. Special care is also devoted to the computation of the Jacobian matrix of the shooting function, using the variational system instead of classical finite differences. The package also features an embedded continuation method and an automatic (parallel) grid shooting in order to reduce the dependency to the initialization.

1 Algorithmic aspects

1.1 Indirect shooting

Direct methods in optimal control first discretize the problem then solve the resulting nonlinear problem. Indirect methods, on the other hand, rely on the necessary conditions given by Pontryagin's Minimum Principle. These conditions give rise to a boundary value problem that can be solved for instance by shooting methods.

Optimal control problem

As a general framework, we consider an optimal control problem in the Mayer form, in the autonomous case (meaning the dynamics f does not depend explicitly on the time t):

$$(P)\begin{cases} Min\ g(t_0, x(t_0), t_f, x(t_f)) & Objective \\ \dot{x} = f(x,u) & Dynamics \\ u \in U & Admissible\ Controls \\ x(t_0) = x_0 & Initial\ Conditions \\ c_1(x(t_f)) = 0 & Terminal\ Conditions \end{cases}$$

with $x(t) \in \mathbf{R}^n$ and $u(t) \in \mathbf{R}^m$.

Remark: a problem with an integral cost $\int_{t_0}^{t_f} l(x,u)\ dt$ can be expressed in the Mayer form by adding a state variable following the dynamics l.

Optimality necessary conditions: Pontryagin's Principle

We assume in the following that the optimal control is piecewise C^1, and that we are in the so-called *normal* case, meaning that the multiplier p_0 associated to the objective is nonzero, and can be therefore set to 1. Let us define the costate p in the same space as x, and the Hamiltonian

$$H(x,p,u) = (p|f(x,u)).$$

Pontryagin's Minimum Principle ([12]): under the assumptions
(i) $\exists$ (x,u) feasible for (P), with x absolutely continuous and u measurable.
(ii) f is continuous with respect to u and C^1 with respect to x.

(iii) g, c_1 *are* C^1 *with respect to* x.

Let (x^*, u^*) *be an optimal pair for* (P)*, then*
(i) $\exists\ p^*$ *absolutely continuous such that* x^*, p^* *satisfy*

$$\dot{x}^* = \frac{\partial H}{\partial p}(x, p, u) \quad , \quad \dot{p}^* = -\frac{\partial H}{\partial x}(x, p, u)$$

(ii) u^* *minimizes the Hamiltonian ae in* $[t_0, t_f]$.
(iii) $\exists\ \mu_1$ *such that the transversality condition hold:*

$$p^*(t_f) = \frac{\partial g}{\partial x_f}(t_0, x(t_0), t_f, x(t_f)) + (\mu_1 | \frac{\partial c_1}{\partial x_f}(x(t_f))) \quad (TC).$$

Free final time. If the final time t_f is free, the additional condition

$$H(x^*(t_f), p^*(t_f), u^*(t_f)) + \frac{\partial g}{\partial t_f}(t_0, x(t_0), t_f, x(t_f) = 0$$

must hold. For minimum time problems, this gives the condition $H(t_f) = -1$.

Boundary Value problem and shooting method

In the following we note $y = (x, p)$ and assume that the Hamiltonian minimization gives the optimal control as a function of y

$$u^*(t) = ArgMin_{w \in U}\ H(x(t), p(t), w) = \gamma(y(t)).$$

Then the state-costate dynamics derived from the Hamiltonian system can also be written as a function of y

$$\varphi(y) = \left(\frac{\partial H}{\partial p}(x, p, \gamma(y)), -\frac{\partial H}{\partial x}(x, p, \gamma(y)) \right).$$

We define the **shooting unknown** $z = p(t_0)$, such that we have $y(t_0) = (x_0, z)$. If the final time is free, the shooting unknown is defined as $z = (p(t_0), t_f)$. More generally, the software can solve problems for which the boundary conditions at t_0 for (BVP) can be written under the form $y(t_0) = y_0(z) \in \mathbf{R}^n$.
At the final time t_f, in most cases the multiplier μ_1 can be eliminated in (TC), leading to a set of equations on $y(t_f) = (x(t_f), p(t_f))$. These equations, in addition to the final conditions $c_1(x(t_f)) = 0$, constitute the boundary conditions at t_f, that we note $B_1(y(t_f))$. If the final time is free, t_f is an additional shooting unknown and is part of z. The corresponding equation $H(x^*(t_f), p^*(t_f), u^*(t_f)) + \frac{\partial g}{\partial t f}(t_0, x(t_0), t_f, x(t_f)) = 0$ is then added to the boundary conditions B_1.

In the general case, we obtain a Boundary Value Problem on $y = (x, p)$

$$(BVP) \begin{cases} \dot{y} = \varphi(y) & ae\ in\ [t_0, t_f] \\ y(t_0) = y_0(z) & \text{Boundary Conditions at } t_0 \\ B_1(y(t_f)) = 0 & \text{Boundary Conditions at } t_f \end{cases}$$

For a given value of the shooting unknown z, we note $y(\cdot, z)$ the solution of the Initial Value Problem

$$(IVP) \begin{cases} \dot{y} = \varphi(y) \\ y(t_0) = y_0(z) \end{cases}$$

and then define the **shooting function** S that maps z to the value of the boundary condition at the final time

$$\begin{aligned} S: \quad & \mathbf{R}^p \to \mathbf{R}^p \\ & z \mapsto B_1(y(t_f, z)) \end{aligned}$$

with $p = n$ if the final time is fixed and $p = n + 1$ for a free final time. Finding a zero of S gives a trajectory that satisfies the optimality conditions from Pontryagin's Principle. The indirect shooting method thus consists in solving the equation $S(z) = 0$, as summarized below:

$$\boxed{(P)} \xrightarrow{\textbf{Pontryagin's Principle}} \boxed{(BVP)} \xrightarrow{\textbf{Shooting method}} \boxed{S(z) = 0}.$$

This package uses the `HYBRD/HYBRJ` routines form Garbow, Hillstrom and More ([4]) to solve the nonlinear equation $S(z) = 0$. Two ODE solvers are available for the evaluation of the shooting function, the classical fixed step 4th order Runge Kutta method and the variable step `DOPRI5` from Hairer and Wanner ([9]). This package is designed to handle two specific difficulties: control discontinuities (see 1.2 Switching detection) and initialization (see 1.4 Discrete continuation and 1.5 Grid shooting).

First, the evaluation of S and its Jacobian matrix can be tricky when control discontinuities (or "switchings") are present. The key is here to detect the switchings during the solving of (IVP), and use the so-called variational equation instead of finite differences to compute the Jacobian.

Then, the matter of finding a suitable initial point is addressed by the means of continuation techniques, coupled to an exploration of a grid of initial points. The latter part is obviously better suited to lower dimension problems, which are the usual situation when using indirect methods as opposed to discretization approaches. This is an alternative to multiple shooting, the latter aiming to increase the convergence radius of the shooting method at the expense of an increased problem dimension.

1.2 Switching detection

We consider now the case of a discontinuous optimal control with switchings. A common situation is when the Hamiltonian is linear in the control and the control is bounded. We assume in the following that the optimal control can take two distinct expressions, depending on the sign of a certain **switching function** ψ. Namely, the Hamiltonian minimization gives

$$u = \gamma_1(y) \; if \; \psi(y) < 0, \quad u = \gamma_2(y) \; if \; \psi(y) > 0$$

For instance, for a scalar bounded control, u^* is either at the lower or upper bound depending on the sign of $\psi = \frac{\partial H}{\partial u}$. This can be generalized for a component of a control $u \in \mathbf{R}^m$, or for a number $k > 2$ of expressions γ_i. The zeros of ψ correspond to the switchings of the optimal control, and we assume a finite number of such switchings. We then obtain $y(\cdot, z)$ as solution of the initial value

problem with a discontinuous right hand side

$$(IVP)_{disc} \begin{cases} \dot{y} = \varphi_1(y) & if\ \psi(y) < 0 \\ \dot{y} = \varphi_2(y) & if\ \psi(y) > 0 \\ y(t_0) = y_0(z) \end{cases}$$

Trying to solve this kind of problem with a fixed step integrator (e.g. Runge Kutta 4th order) is in practice nearly impossible except for very simple cases, due to the errors at each switching. On the other hand, a variable step integrator (such as `RKF45` or `DOPRI5`) can usually handle the switchings by itself. However, this costs many very small steps at the switchings, leading to an increased cpu time, and the remaining small errors at each switching can pile up. As a result, the shooting method is likely to be slower, and also suffers from a loss of precision.

An effective way to solve this problem is to detect the switchings during the integration of $(IVP)_{disc}$, as described for instance in [9, 10]. We use here the detection method based on the dense output of the ODE solver, ie a cheap (polynomial) approximation of y on each time interval, noted $y^{dense\ output}$. This algorithm can be applied to any fixed or variable step integration method with a dense output, and is recalled below.

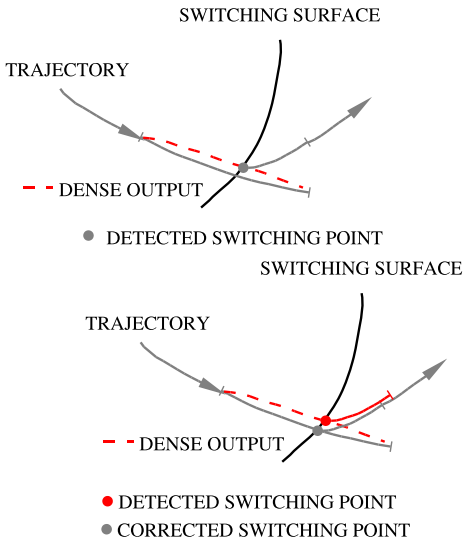
Main integration loop
compute $y(t+h)$
compute switching function $\psi(y(t+h))$
compare the signs of $\psi(y(t+h))$ and $\psi(y(t))$
If sign change **Then**
locate switching $\tau \in [t, t+h]$ by solving $\psi(y^{dense\ ouptut}(\tau)) = 0$
switch control
pursue integration on $[\tau, t+h]$ from $y^{dense\ ouptut}(\tau)$
End

A simple way to solve $\psi(y^{dense\ ouptut}(\tau)) = 0$ in order to locate the switching time τ is to use a bisection. This bisection stops either at a fixed number of iterations or as soon as the interval length is below a fixed tolerance.

Switching correction. A small flaw of the above algorithm is that the switching point $y(\tau)$ from which we resume the IVP integration is inexact, as given by the dense output. The dense output is typically one order less accurate than the actual ODE method, and these small errors can add up in case of a large number of switchings. We can correct the switching point by solving the equation $\psi(y^{ODE}(\bar{\tau})) = 0$, where $y^{ODE}(\bar{\tau})$ is given by an actual step of integration from t to $\bar{\tau}$, instead of the dense output. This equation can be solved by a newton method, using the previously located τ as initial point. In this case, the stopping criterion of the bisection for the detection can be less strict, as this point serves only as an initialization.

Remark: It is also possible to perform the whole switching detection while using actual integration instead of the dense output. However, it is faster to use the dense output for an approximate detection and then making the correction.

This minor modification can improve the convergence of the shooting method, as well as the conservation of numerical invariants of the ODE system (such as the Hamiltonian). Graphs below illustrate the difference between basic switching detection and switching correction.



1.3 Jacobian evaluation and Variational system

The most straightforward method to compute the Jacobian of the shooting function is to use finite differences. This is done in the `HYBRD` solver with a step of $h_j = \sqrt{\epsilon}|x_j|$, where ϵ is the error on the shooting function evaluation. However, when using a variable step ODE solver, the time steps can differ at the points for the finite differences. This can impair the approximation of the Jacobian by finite differences (see [2, 9]).

Another option to compute the Jacobian is to use the method described in [9], which is equivalent to the "internal differentiation"[1] from [2].

1.3.1 Smooth (C^1) case

In the smooth case, we consider the solution $y(\cdot, z)$ of the ODE system

$$(IVP) \begin{cases} \dot{y}(t) = \varphi(y(t)) \\ y(t_0) = y_0 \end{cases}$$

Then the derivatives $Y = \frac{\partial y}{\partial y_0}(y_0)$ are solution of the variational system

$$(VAR) \begin{cases} \dot{Y}(t) = \frac{\partial \varphi}{\partial y}(y(t))\ Y(t) \\ Y(t_0) = I \end{cases}$$

Here the initial condition is actually of the form $y(t_0) = y_0(z)$, and we are interested in the derivatives of $y(\cdot, z)$ with respect to z. Assuming that we have

[1] while "external differentiation" refers to the usual finite differences

$y(t_0) = [x_0; z]$ for the sake of simplicity[2], the variational equation for $\frac{\partial y}{\partial z_j}(t, z)$ is

$$(VAR)_j \begin{cases} \dot{Y}_j(t) = \frac{\partial \varphi}{\partial y}(y(t))Y_j(t) \\ Y_j(t_0) = \begin{pmatrix} 0 & \cdots & 1 & \cdots & 0 \end{pmatrix}^T \end{cases}$$

where the right hand side can be approximated by finite differences if not available analytically

$$\dot{Y}(:, j) \approx \frac{1}{h}\ (\varphi(y + hY(:, j)) - \varphi(y)).$$

Solving the variational system (VAR) along with (IVP) provides both $y(t_f, z)$ and $\frac{\partial y}{\partial z}(t_f, z)$. The Jacobian of the shooting function can be computed from the latter, as $S(z) = B_1(y(t_f, z))$:

$$Jac_S(z) = \frac{\partial B_1}{\partial y}(y(t_f))\ \frac{\partial y}{\partial z}(t_f, z).$$

With a fixed step integration, using finite differences for the right hand side of (VAR) has the same overall complexity as using finite differences to compute the Jacobian of the shooting function, namely $(n + 1) \times N_{steps}$ evaluations of φ. This still holds with a variable step method if we assume that solving both (IVP) and (VAR) for the Jacobian takes roughly the same number of steps as solving only (IVP) for the shooting function. This can be enforced in the ODE solver, and is recommended in the discontinuous case.

1.3.2 Discontinuous case

In presence of discontinuities, jumps occur in the variational equation, that must be computed at each control switching. Note that this implies the detection of said control switchings, for instance by the method described in 1.2. Assuming that a switching from φ_1 to φ_2 occurs at $\tau \in]t_0, t_f[$, we have to perform the update (see [7])

$$Y \leftarrow Y + (\varphi_1(y_\tau) - \varphi_2(y_{\tau^+}))\tau'(y_0).$$

The switching time is determined by the equation $\psi(y(\tau, y_0)) = 0$, thus by the implicit function theorem, assuming that $\frac{\partial \psi}{\partial y}(y_\tau) \neq 0$:

$$\tau'(y_0) = -\frac{\frac{\partial \psi}{\partial y}(y_\tau)Y(\tau, y_0)}{\frac{\partial \psi}{\partial y}(y_\tau)\varphi_1(y_\tau)}.$$

These updates are done automatically with the switching detection routine.

NB. It should be noted that using the variational system without the switching detection leads to a wrong Jacobian matrix, as the jumps described above are not computed. Therefore, it is not recommended to use the variational system when the switching detection is not available.

Time steps. Computing the shooting function and its Jacobian matrix should use the same sequence of time steps, at least to ensure that the same switchings are detected. The shooting function only requires the solving of

[2] ie z is actually the initial costate $p(0)$

(IVP), while its Jacobian matrix needs to solve both (IVP) and (VAR). This usually leads to different time steps when using a variabel step ODE solver. To prevent this, we ignore the variables of (VAR) in the formulas for the initial stepsize and the stepsize control. This has been added in the `DOPRI5` code as option `ITOL=2`.

1.4 Discrete continuation method

Continuation techniques are commonly used to find a suitable initialization for the shooting method, starting with an easier problem and progressively going back to the original problem. Let us define the continuation parameter $\lambda \in [\lambda_0, \lambda_f]$ and a family of problems (P_λ) such that $(P_{\lambda_f}) = (P)$ and we know a solution of (P_{λ_0}). We define the homotopy

$$\begin{array}{rl} \mathcal{H}: & \mathbf{R}^n \times [\lambda_0, \lambda_f] \to \mathbf{R}^n \\ & (\lambda, z) \mapsto S_\lambda(z) \end{array}$$

where S_λ is the shooting function associated to the problem (P_λ). The aim of the continuation is to follow the zero path of $\mathcal{H}$ from $\lambda = \lambda_0$ to $\lambda = \lambda_f$. Note that the existence of such a path is not guaranteed in general, and of course depends on the family (P_λ). For practical purposes, suitable continuations often involve some physical parameters and / or regularization of the original problem. There are several classes of methods to perform this path following, for instance differential continuation or simplicial homotopy (see [1, 5, 6, 8, 11], and the HAMPATH[3] package). We use here a discrete continuation with a linear prediction, which does not require smoothness of the zero path and can be seen a very simple predictor-corrector method.

$\lambda = \lambda_0$; step = maxstep
While ($\lambda < \lambda_f - \epsilon$) and (step > minstep) and (iter < maxiter)
    iter = iter + 1; $\lambda = \lambda$ + step
    compute initialization for shooting at level λ (prediction)
    perform shooting attempt at level λ (correction)
    If shooting successful
        update path with new solution
    Else
        back to previous solution; reduce stepsize (step = step / 2)
    End if
End while

We note (z_n, λ_n) the sequence of zeros of the homotopy computed by the algorithm. The simplest way to compute the initialization $z^{init}_{\lambda_{n+1}}$ for the next shooting attempt is simply to take the solution of the latest successful shooting, ie

$$z^{init}_{\lambda_{n+1}} = z_n.$$

A little better is a linear prediction based on the two previous solutions

$$z^{init}_{\lambda_{n+1}} = z_n + \frac{\lambda - \lambda_n}{\lambda_n - \lambda_{n-1}}(z_n - z_{n-1}).$$

[3] http://apo.enseeiht.fr/hampath/

Numerical experiments indicate that this approach is generally superior to the basic initialization by the previous solution. Higher order (quadratic or cubic) predictions do not seem to improve the path following.

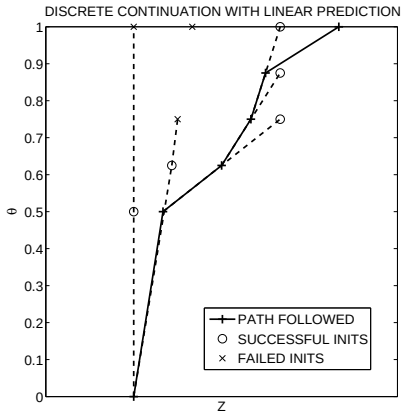
The maximal step for the continuation parameter λ is typically equal to $\lambda_f - \lambda_0$, so that the path following may be completed in only 1 iteration if the first shooting attempt succeeds. The default value for the normalized maximal step (set in the .init file see 2.3) is therefore 1. One may want to decrease this value if the path following seems to become unstable, for instance with points branching out from the initial path. This can also be used to force the computation of solutions at certain prescribed values of the continuation parameter.

The minimal step is used to stop the continuation if the stepsize becomes too small, without having to wait until the maximal number of iterations is reached. As the step for λ can only decrease, it is indeed unlikely that the continuation can succeed when this step becomes very small. Attempts to increase again the step after a certain number of successful shootings seem to make the path following more unstable, and yield no overall benefit.

The path following may fail to reach the prescribed value for the continuation parameter in two distinct cases. First, the path stops at a certain limit value for the continuation parameter, and further progress is impossible. This case is usually detected by reaching the minimal stepsize for λ, and the path following should stop at the limit value. Second, the path may go away "to infinity", meaning that one or more components of the solution tend to infinity. This case is usually detected by reaching the maximum number of iterations, and can be checked by plotting the continuation path saved in the *.contpath* file.

Here is a schematic example of a path following for $\lambda \in [0, 1]$.

Iter	Step	Theta	Shooting
1	1	1	Fail
2	0.5	0.5	Success
3	0.5	1	Fail
4	0.25	0.75	Fail
5	0.125	0.625	Success
6	0.125	0.75	Success
7	0.125	0.875	Success
8	0.125	1	Success



Alternately, the continuation can be used to explore a range of values for a certain parameters, by setting the lower, upper bounds and maximal stepsize for the continuation parameter accordingly.

1.5 Grid shooting

Another way to find a correct initialization is basically to try several ones, which can be automated in order to explore a grid of initial points. This procedure can be a simple way to obtain convergence, and may also detect local solutions, provided the computational cost remains reasonable.

Two aspects make this idea interesting in our context of solving optimal control problems with an indirect shooting method. First, the dimension of the problem is usually small, typically around 10, as it is more or less the dimension of the state variables.

Then, the shooting method tends to be fast when it converges, but also when it diverges. For instance, the newton method usually fails after only one iteration for a starting point outside the domain of definition of the shooting function, which can be quite small in practice (although not precisely known). Therefore, the overall cost of making many failed attempts may be quite acceptable, allowing for several thousands of shooting attempts in less than one hour on a standard computer.

This procedure can be combined with the continuation approach described earlier. In this case, for each point of the grid we perform a full path following instead of a simple shooting attempt. At the end of the grid all the obtained solutions are sorted by value of the objective, with a count of the successful shootings for each solution. Besides increasing the chance of finding a suitable initial point for the shooting method, grid shooting also allows for exploring the solution space, and can detect multiples or local solutions. This can be useful for real-life applications, for which a local solution may happen to be more suitable for practical use. Grid shooting is also interesting for benchmarking purposes: due to the high sensitiveness of the shooting method to the initial point, collecting convergence results over a large number of initializations is more reliable than using a single test. For instance, the simulations in [3] study the relevance of using the Hamilton-Jacobi-Bellman approach to initialize the shooting method.

We consider the initialization grid defined by the lower bounds $z^L \in \mathbf{R}^n$, upper bounds $z^U \in \mathbf{R}^n$ and range vector $r \in \mathbf{N}^n$. The grid shooting algorithm will perform an automated sequence of shootings for all initial points

$$z_0 = z^L + \Sigma_{i=1}^n k_i \frac{z_i^U - z_i^L}{r_i}, \quad \forall k_i \in [0, r_i].$$

Each initial point is defined by the vector of indices $k = (k_i)_{i=1,n}$, and the total number of grid points is $N = \Pi_{i=1}^n (r_i + 1)$. In practice, we use a single loop instead of embedded `DO` loops for each dimension. This loop increments the vector of indexes k from $k = (0, \ldots, 0)$ to $k = r$.

k=(0,...,0)
While k $\neq$ r **Do**
    Compute grid point $z_0 = z^L + \Sigma_{i=1}^{n} k_i \frac{z_i^U - z_i^L}{r_i}$
    Perform shooting (or full path following for continuation) with z_0
    Increment indexes vector k for next grid point
        carry = true; i = 1
        While (carry & $i \leq n$) **Do**
            If $k_i < r_i$ **Then**
                $k_i = k_i + 1$; carry = false
            Else
                $k_i = 0$; carry = true; i=i+1
            End if
        End do
End do

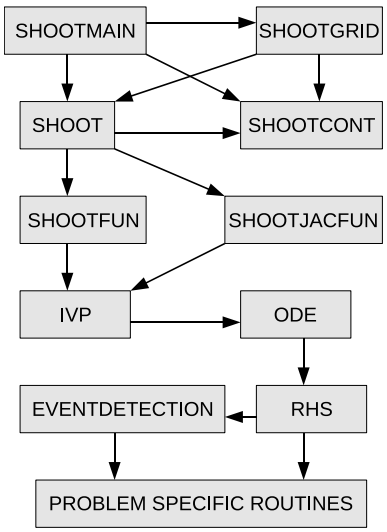
It is worth noting that there is no need to generate and store the full grid, as each point can be computed directly from the vector of indexes $k = (k_i)_{i=1,n}$. Moreover, this grid shooting can be completely parallelized, as each attempt is independent from the others and the order does not matter. A parallel version using OPENMP is available and can benefit from multi-core CPUs. Numerical experiments indicate that the total computation time is divided by the number of cores, as expected.

2 Software overview

This section presents an overview of the main subroutines of the SHOOT2.0 package, as well as a description of the user-supplied subroutines required to solve a problem, and the list of the input and output files. The code is written in Fortran90/95, and has been tested with the compilers ifort[4], gfortran[5] and g95[6].

[4] http://software.intel.com/en-us/intel-compilers/
[5] http://gcc.gnu.org/fortran/
[6] http://www.g95.org/

2.1 Main subroutines



- ShootMain: main program.
- ShootGrid: performs shooting over an initialization grid.
- ShootCont: shooting method with discrete continuation.
- Shoot: indirect shooting method.
- ShootFun: computes the shooting function S.
- ShootJacFun: computes the Jacobian matrix of S.
- IVP: solves the Initial Value Problem required for S or Jac_S.
- ODE: solves the ODE system for the Initial Value Problem.
- RHS: computes right hand side for the ODE system.
- EventDetection: detects switchings during ODE integration.
- Problem Specific Routines: user supplied subroutines, see 2.4.

2.2 Times structure

The shoot2.0 package can handle problems with elaborate times structures, comprising fixed or free final time as well as multi-phase dynamics. Control switchings are handled specifically by the switching detection method, and therefore do not appear explicitly in the times structure.

2.2.1 Final time

The final time can be either fixed of free, in which case it is part of the shooting unknown, as the last component of z. A free t_f implies the additional optimality condition $H(x(t_f), p(t_f), u(t_f)) + \frac{\partial g}{\partial tf}(t_0, x(t_0), t_f, x(t_f)) = 0$, which is added as the last component of the shooting function $S(z)$. The type of final time is set in the input file (see 2.3): 1 stands for a fixed final time, and -1 for a free final time.

2.2.2 Multi-phase problems

We refer here as a multi-phase problem a system whose dynamics is governed by several sets of equations, depending on the time. For instance, in the case

of trajectory optimization for a multi-stage space launcher, the flight is divided into several phases corresponding to the each propulsion system. Each time interval with a certain set of equation is called a phase, and the times delimiting the phases can be either fixed or free. In the latter case, the optimal value for this phase time is part of the shooting unknowns, and the corresponding equation must be defined in the subroutine PHASECONDITION.

The current phase is stored in the global variable IPAR(19), and can be used in the user supplied subroutines, see 2.4. The phase is typically required to compute the control and/or dynamics, as well as the phase conditions in the free time case. Phase times are specified in the time structure in the input file (see 2.3): a 2 indicates a fixed phase time, whose value must be specified; -2 indicates a free phase time, whose value is part of the shooting unknown z.

For instance, a problem with free initial costate, two free phase times t_1, t_2 and free final time would give $z = [p(0); t_1; t_2; t_f]$.

2.3 Input and output files

All input and output files for a given problem will use the same prefix, with different extensions. The algorithm requires a single initialization file, say *Problem.init*, and typically generates a solution file *Problem.sol*. Discrete continuation can produce an additional path file *Problem.contpath* storing the path following. Grid shooting also outputs a *Problem.gridcv* file which summarizes the convergence results over the grid, as well as *Problem-cvxx.sol* files for each solution found. The Matlab scripts *sol.m* and *contpath.m* can visualize the .sol and .contpath files. The .init file contains the following entries (see appendix A for an example).

Shooting method

- Shooting, continuation and verbose mode
  Shooting: -1: only compute $S(z)$, 0: basic shooting, 1: grid shooting.
  Continuation: 0: no continuation, 1: use continuation.
  Verbose: from -1 (no output) to 2 (most info).

- State, Control, Switch dimensions
  dimension n, m of state and control variables, and size of the vector returned by the subroutine SWITCH (should be set to 1 if unused).

- Number of arcs and times structure
  number of arcs (at least 1, see below for arc definition).
  time structure (at least initial and final times):
  0: (fixed) initial time
  1: fixed final time, -1: free final time
  2: fixed phase time, -2: free phase time.
  values for fixed and free times[7].

[7] put any value for free times that are part of z

- Shooting unknown dimension and initial guess
  size of the shooting unknown[8], size of initial unknown values.
  starting point for shooting unknown.
- Integrator choice, fixed steps, relative and absolute tolerances
  4: Runge Kutta 4th order, 5: DOPRI5.
  number of steps for `RK4`, tolerances for `DOPRI5`.
- Switching detection mode
  -1: disabled, 0: use switching detection, 1: use switching correction
- Jacobian mode
  0: usual finite differences, 1: use variational system.
- Target convergence
  tolerance (on shooting function norm) for a successful shooting.

Continuation

- Initial, final value and max normalized step for continuation parameter
  Initial and final values λ_0, λ_1 for the continuation parameter λ, and maximal normalized stepsize for λ during the path following (see 1.4).
- Max iterations and iterations output frequency
  maximal number of allowed iterations for the continuation.
  output frequency (< 1: no output, $n > 0$: display every n iteration(s)).
- Prediction type
  prediction mode for the continuation (0: constant, 1: linear).
- Output sol and path
  generate solution file *.sol* (1: enabled, other: disabled).
  generate path following file *.contpath* (1: enabled, other: disabled).

Grid shooting

- Lower bounds for grid shooting (see 1.5).
- Upper bounds for grid shooting (see 1.5).
- Range for grid shooting (see 1.5).

Problem specific parameters

- Parameters
  size and value of problem specific parameters.

2.4 User supplied subroutines

The following subroutines are specific to each problem and must be provided:
- InitialConditions: computes the initialization $y(t_0) = y_0(z)$.
- FinalConditions: computes the boundary conditions $B_1(y(t_f))$.
- Control: computes the optimal control minimizing the Hamiltonian.
- Dynamics: computes the dynamics for the state and costate variables.

[8] should be equal to size of initial unknown values + number of free times

These optional subroutines can be left empty[9] if unused:
- InitPar: specific initializations, such as table interpolation.
- Switch: computes the switching function used for switching detection.
- BangBangControl: control according to the switching function.
- PhaseCondition: equations to be satisfied by the free phase times.

General notations:
- `y(nxp)` denotes the state-costate pair (x, p).
- `u(m)` denotes the control.
- `nfree0` is the dimension of the shooting unknown z.
- `ipar(lipar),rpar(lrpar)` are global variables arrays described in 2.5.

2.4.1 InitialConditions

```
Subroutine InitialConditions(z,dim,y,lipar,ipar,lrpar,rpar)
  implicit none
  integer, intent(in) :: dim, lipar, lrpar
  integer, intent(inout) :: ipar(lipar)
  real(kind=8), intent(in) :: z(nz)
  real(kind=8), intent(inout) :: rpar(lrpar)
  real(kind=8), intent(out) :: y(dim)
...
```

This subroutine defines the initial conditions for the state and costate at initial time, namely $y(t_0) = (x(t_0), p(t_0))$, in the first `nxp` components of `y`. Typically, the state x is set according to $x(t_0) = x_0$, while the free initial costate is given by the shooting unknown as $p(t_0) = z$. More generally, $y(t_0)$ is defined by the initial and transversality conditions at t_0, with the "missing parts" coming from the shooting unknown z.

If the variational system is used for the Jacobian, then the derivatives of $y(t_0)$ with respect to z must also be filled (the $\frac{\partial y(t_0)}{\partial z}$ are stored by lines after the first `nxp` components of `y`).

2.4.2 FinalConditions

```
Subroutine FinalConditions(y,s,dsdy,lipar,ipar,lrpar,rpar)
  implicit none
  real(kind=8), intent(in) :: y(nxp)
  integer, intent(in) :: lipar, lrpar
  integer, intent(inout) :: ipar(lipar)
  real(kind=8), intent(inout) :: rpar(lrpar)
  real(kind=8), intent(inout) :: s(nfree0), dsdy(nfree0,nxp)
...
```

This subroutine computes the value of the boundary conditions at t_f for the shooting function. On entry the state-costate pair $y = (x, p)$ at the final time is provided in `y`. In the free final time case, the last component of `s` contains $H(t_f)$ on entry. On exit the shooting function value `s` must be filled with the value of $B_1(y(t_f))$. If the variational system is used to compute the Jacobian

[9] but must still be present for the compilation

matrix of the shooting function, then the derivatives of theses conditions must also be filled in the matrix `dsdy`.

2.4.3 Control

```
Subroutine Control(y,u,lipar,ipar,lrpar,rpar)
  implicit none
  integer, intent(in) :: lipar, lrpar
  integer, intent(inout) :: ipar(lipar)
  real(kind=8), intent(inout) :: rpar(lrpar)
  real(kind=8), intent(in) :: y(nxp)
  real(kind=8), intent(out) :: u(m)
...
```

This subroutine provides the optimal control that minimizes the Hamiltonian, according to Pontryagin's Minimum Principle. On entry the state-costate pair $y = (x, p)$ at current time is provided in `y`. On exit the control value must be provided in `u`.

2.4.4 Dynamics

```
Subroutine Dynamics(y,u,f,mode,lipar,ipar,lrpar,rpar)
  implicit none
  integer, intent(in) :: lipar, lrpar
  integer, intent(inout) :: ipar(lipar)
  real(kind=8), intent(inout) :: rpar(lrpar)
  integer, intent(in) :: mode
  real(kind=8), intent(in) :: y(nxp), u(m)
  real(kind=8), intent(out) :: f(nxp)
...
```

This subroutine provides the dynamics for the state and costate variables. On entry the value of the state, costate and control at the current time are given in `y,u`. If `mode`=0, the full dynamics $\dot{y} = (\dot{x}, \dot{p})$ is required in `f`. If `mode`=1, only the state dynamics $\dot{x}$ is required in the first half of `f`.

2.4.5 InitPar

```
Subroutine InitPar(lipar,ipar,lrpar,rpar)
  implicit none
  integer, intent(in) :: lipar, lrpar
  integer, intent(inout) :: ipar(lipar)
  real(kind=8), intent(inout) :: rpar(lrpar)
...
```

This optional subroutine is called at the beginning of each shooting attempt (and not for every shooting function call, unlike InitialConditions). It may be used for one-time initializations, for instance related to discrete continuation or table interpolations (splines computations).

2.4.6 Switch

```
Subroutine Switch(y,psi,lipar,ipar,lrpar,rpar)
  implicit none
```

```
  real(kind=8), intent(in) :: y(nxp)
  integer, intent(in) :: lipar, lrpar
  integer, intent(inout) :: ipar(lipar)
  real(kind=8), intent(inout) :: rpar(lrpar)
  real(kind=8), intent(out) :: psi(npsi)
...
```

This optional subroutine provides the switching function ψ used for the switching detection. On entry the value of the state and costate at the current time are given in `y`. On exit the switching function must be provided in the first component of `psi`. Additional components may be used to store other values that the user would like to compute at each time step for visualization purposes.

2.4.7 BangBangControl

```
Subroutine BangBangControl(y,u,lipar,ipar,lrpar,rpar)
  implicit none
  integer, intent(in) :: lipar, lrpar
  integer, intent(inout) :: ipar(lipar)
  real(kind=8), intent(inout) :: rpar(lrpar)
  real(kind=8), intent(in) :: y(nxp)
  real(kind=8), intent(out) :: u(m)

  select case (ipar(17))
  case (-1)
     !switching function is negative
     u(1) = ...
  case (1)
     !switching function is positive
     u(1) = ...
  end select

end Subroutine BangBangControl
```

This subroutine is used to compute the optimal control when using the automatic detection of switchings. Switching detection is enabled by setting the corresponding flag to 0 or 1 in the input file *.init* (see 2.3), and requires the subroutine SWITCH above that defines the switching function. The user has to provide the two expressions of the optimal control depending on the sign of the switching function. This sign at the current time step is automatically stored in the global variable `ipar(17)`.

2.5 Global variables

The two arrays `ipar` and `rpar`, of size `lipar` and `lrpar`, contain the integer and real global variables for the SHOOT2.0 package. Most settings and parameters for the algorithms are stored in these arrays, while other parts are for internal use. Below is a short description of the values that can be of use in the user-supplied subroutines.

2.5.1 Integer global variables: IPAR

- `ipar(2)`: mode for embedded continuation (0: disabled, 1: enabled)
- `ipar(3)`: Jacobian mode (0: Finite difference, 1: Variational System)

- `ipar(19)`: current phase number (starts at phase 1)
- `ipar(30)`: total number of phases, computed from time structure

2.5.2 Real global variables: RPAR

- `rpar(1)`: objective value (from last call to IVP)
- `rpar(2)`: norm of the shooting function (idem)
- `rpar(3)`: λ (continuation parameter)
- `rpar(4)`: Hamiltonian value (from last call to RHS)
- `rpar(12)`: initial value λ_0 for continuation parameter
- `rpar(13)`: final value λ_1 for continuation parameter
- `rpar(roffset1+1:roffset1+npar)`: optional parameters defined at the end of the *.init* file, see 2.3

3 Illustration problems

3.1 Step by step example

We now illustrate the method on a very simple optimal control problem:

$$(P)\begin{cases} Min\ \int_0^2 |u(t)|\ dt \\ \dot{x}_1 = x_2 \\ \dot{x}_2 = u \\ |u| \leq 1 \\ x(0) = (0,0) \\ x(2) = (0.5,0) \end{cases}$$

The Hamiltonian is defined by

$$H : (t,x,p,u) \mapsto |u| + p_1\ x_2 + p_2\ u$$

and the costate p is solution of the adjoint equation

$$\dot{p}_1 = 0, \quad \dot{p}_2 = -p_1.$$

The optimal control is discontinuous

$$\begin{cases} u = -sgn(p_2) & if\ \psi(t,x,p) < 0 \\ u = 0 & if\ \psi(t,x,p) > 0 \end{cases}$$

with the switching function

$$\psi : (t,x,p) \mapsto 1 - |p_2|.$$

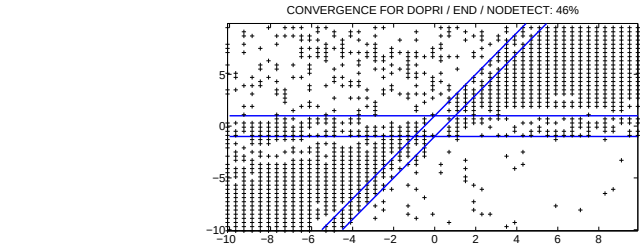
The unknown for the shooting method is the initial costate and the shooting function is defined by

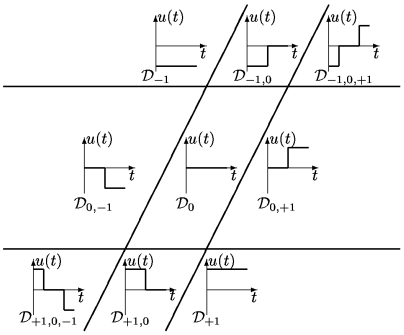
$$\begin{array}{lll} S: & \mathbf{R}^2 & \to \mathbf{R}^2 \\ & z = p(0) & \mapsto x(2) - (0.5,0) \end{array}$$

The user-supplied subroutines and the input file corresponding to this simple example are given in appendix A. The objective $Min\ \int_0^2 |u(t)|\ dt$ is defined as

a third component for the state, whose corresponding costate is equal to 1.

We test the shooting method on a 50×50 grid over $[-10, 10]^2$ for the initial costate. We compare the two approaches for the Jacobian, finite differences and variational system, with or without switching detection. We note that depending on the initial costate $p(0)$, we have 9 possible control structures with 0, 1 or 2 switchings, and that convergence is related to these regions.





Shooting method convergence - control structures

The table summarizes the convergence results for the grid shootings, with tolerances of 10^{-8} for the ODE solver. For each run we indicate the percentage of successful shootings over the grid, and the best convergence (norm of the shooting function) obtained. We observe that the switching detection and correction improve the precision of the shooting method, with a better norm. Using the variational system gives better chances of success than the basic finite differences.

Jacobian	No detection		Detection		Correction	
FD	**53%**	$5.75\ 10^{-7}$	**67%**	$2.72\ 10^{-15}$	**67%**	$1.24\ 10^{-16}$
VAR	**68%**	$5.75\ 10^{-7}$	**79%**	$2.72\ 10^{-15}$	**80%**	$1.24\ 10^{-16}$

Simple bang-bang problem - grid shooting

3.2 Orbital transfer

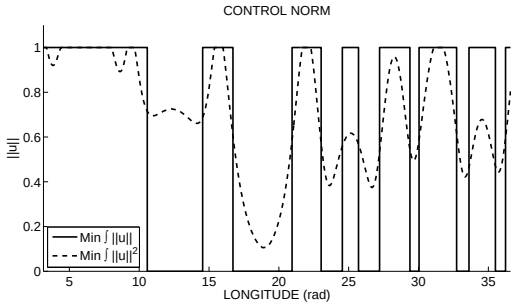
We show here an orbital transfer problem studied in [7], from an elliptic transfer orbit to the geostationary orbit. We consider a satellite with a low thrust electro-ionic propulsion, with thrusts ranging from 10 Newtons to 0.1 Newton. The forces applied to the satellite are the Earth attraction and the engine thrust, giving the dynamics

$$\ddot{r} = -\mu \frac{r}{|r|^3} + \frac{T}{m}.$$

The objective is to maximize the payload, i.e., to minimize the fuel consumption during the transfer. Unlike the minimum-time transfer, the optimal trajectories present a bang-bang control, with either full thrust (at apogees and perigees) or no thrust (see figure). An interesting continuation approach is to go from an energy type criterion to the mass criterion as

$$Min \int_{t_0}^{t_f} \lambda |u(t)| + (1-\lambda)|u(t)|^2 \, dt.$$

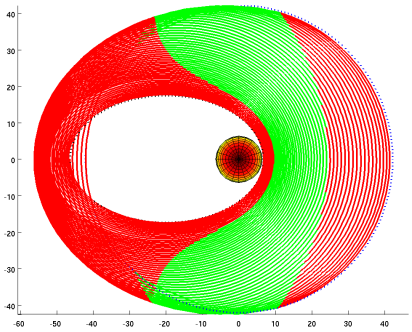
The graph below shows the smooth control for $\lambda = 0$ and the discontinuous control for $\lambda = 1$ (for a 10N thrust).



Orbital transfer - smooth and discontinuous control depending on objective

The solution from the energy criterion is usually sufficient to initialize the shooting method and solve the problem for the mass criterion. This continuation is quite effective, but solving the problem for the energy criterion becomes difficult for low thrusts. We test the grid shooting with embedded continuation on the $1N$ transfer, with `dopri5` (tolerances 10^{-8}) as ODE solver. With the simple grid $\{-0.1, 0.1\}^n$ for the initial costate $p(0)$ (ie for each component of $p(0)$ we only try the two values ± 0.1, for a total of 128 shooting attempts), we are able to solve the problem for a maximal thrust as low as $0.1N$.

For thrusts of $10N$, $1N$ and $0.1N$, the optimal trajectories present 14, 119 and 1195 control switchings respectively. The graph below shows the optimal trajectory for a 1N thrust, with the thrust arcs (in red) located at apogees and perigees of the orbits, and arcs with no thrust (in green).



Orbital transfer - Trajectory for a 1N thrust (119 switchings)

We detail now the tests on the $1N$ transfer to compare several options. For the Jacobian, we use either finite differences (FD) or variational system (VAR). For the switchings, we use either no detection, basic detection, or detection with correction. The table below sums up the convergence results for these six configurations. For each grid shooting we indicate the success ratio (percentage of successful shootings over the grid) and the best convergence obtained (ie lowest norm of the shooting function).

Jacobian	No detection		Detection		Correction	
FD	**25%**	$1.25\ 10^{-5}$	**25%**	$4.83\ 10^{-8}$	**25%**	$3.06\ 10^{-9}$
VAR	**87%**	$6.51\ 10^{-5}$	**89%**	$2.80\ 10^{-8}$	**89%**	$2.19\ 10^{-10}$

Orbital transfer (1N) - grid shooting CV results

Here the interest of switching detection is clear, as well as the effectiveness of the grid shooting with embedded continuation. Overall, we once again observe that the switching detection and correction improve the precision of the shooting method. The variational system, on the other hand, gives better chances of success than the basic finite differences. We also notice that using the variational system without the switching detection is much slower, which is probably due to the wrong Jacobian (see 1.3).

Jacobian	No detection		Detection		Correction	
FD	1326 / 337	**3.93**	1106 / 281	**3.94**	1152 / 293	**3.93**
VAR	5479 / 1384	**3.96**	1025 / 261	**3.93**	1017 / 259	**3.93**

Orbital transfer (1N) - Total CPU and clock times (s), cpu / clock time ratio

All tests were run on a quad-core 3GHz Xeon processor, using the parallel (OPENMP) version of the grid shooting. The times indicate that the actual computation time (clock time) is roughly 4 times smaller than the total CPU time (summed on all cores), as expected on a quad-core CPU.

References

[1] E. Allgower and K. Georg. *Numerical Continuation Methods.* Springer-Verlag, Berlin, 1990.

[2] H.G. Bock. Numerical treatment of inverse problems in chemical reaction kinetics. In K.H. Ebert, P. Deuflhard, and W. Jäger, editors, *Modelling of Chemical Reaction Systems*, volume 18 of *Springer Series in Chemical Physics*, pages 102–125. Springer, Heidelberg, 1981.

[3] E. Cristiani and P. Martinon. Initialization of the shooting method via the hamilton-jacobi-bellman approach. *Journal of Optimization Theory and Applications*, 146(2):321–346, 2010.

[4] B.S. Garbow, K.E. Hillstrom, and J.J. More. *User Guide for Minpack-1.* National Argonne Laboratory, Illinois, 1980.

[5] J. Gergaud and T. Haberkorn. Homotopy method for minimum consumption orbit transfer problem. *Control, Optimization and Calculus of Variations*, 12(2):294–310, 2006.

[6] J. Gergaud, T. Haberkorn, and P. Martinon. Low thrust minimum-fuel orbital transfer: an homotopic approach. *Journal of Guidance, Control and Dynamics*, 27(6):1046–1060, 2004.

[7] J. Gergaud and P. Martinon. Using switching detection and variational equations for the shooting method. *Optimal Control Applications and Methods*, 28(2):95–116, 2007.

[8] T. Haberkorn. *Transfert orbital à poussée faible avec minimisation de la consommation : résolution par homotopie différentielle.* PhD thesis, INP Toulouse, 2004.

[9] E. Hairer, S. P. Nørsett, and G. Wanner. *Solving ordinary differential equations. I*, volume 8 of *Springer Series in Computational Mathematics.* Springer-Verlag, Berlin, 1993.

[10] R. Mannshardt. One-step methods of any order for odes with discontinuous rhs. *Numer. Math.*, 31:131–152, 1978.

[11] P. Martinon. *Numerical resolution of optimal control problems by a Piecewise Linear continuation method.* PhD thesis, INP Toulouse, 2005.

[12] L. Pontryagin, V. Boltyanski, R. Gamkrelidze, and E. Michtchenko. *The Mathematical Theory of Optimal Processes.* Wiley Interscience, New York, 1962.

A Input file and user supplied subroutines

A.1 Input file sample.init

```
*** Shooting method ***
Shooting, continuation and verbose mode
1 0 1
State, Control, Switch dimensions
3 1 1
Number of arcs and times structure
1
0 1
0d0 2d0
Shooting unknown dimension and initial guess
2 2
-1.4 -1.4
Integrator choice, fixed steps, relative and absolute tolerances
5 100 1d-8 1d-8
Switching detection mode (-1: disabled  0: dense output)
-1
Jacobian mode (0:FD  1:VAR)
0
Target convergence
1d-4

*** Embedded continuation ***
Initial, final value and max normalized step for homotopic parameter
0d0 1d0 1d0
Max iterations and iterations output frequency
100 -1
Prediction type
1
Output sol and path
1 1

*** Grid shooting ***
Lower bounds
-10.1 -10.1
Upper bounds
9.9 9.9
Range
50 50

*** Problem specific parameters ***
1
1
```

A.2 Subroutines in file sample.f90

```
  Subroutine InitialConditions(y,lipar,ipar,lrpar,rpar)
    implicit none
    integer, intent(in) :: lipar, lrpar
    integer, intent(inout) :: ipar(lipar)
    real(kind=8), intent(inout) :: rpar(lrpar)
    real(kind=8), intent(inout) :: y(nxp)

    !local
    integer :: contmode, bangbangmode, bangbangmode0
    real(kind=8) :: lambda

    !global vars
    contmode = ipar(2)
    lambda  = rpar(3)

    !CI
    y(1:3) = 0d0
    !CT (obj)
    y(6) = 1d0

    !criterion
    if (contmode == 0) then
       lambda = rpar(roffset1+1)
```

```
      rpar(3) = lambda
   end if

   !disable switchings detection for smooth control
   bangbangmode0 = ipar(9)
   if (lambda < 1d0) then
      bangbangmode = -1
   else
      bangbangmode = bangbangmode0
   end if
   ipar(10) = bangbangmode

 end Subroutine InitialConditions

 Subroutine FinalConditions(y,s,dsdy,lipar,ipar,lrpar,rpar)
   implicit none
   real(kind=8), intent(in) :: y(nxp)
   integer, intent(in) :: lipar, lrpar
   integer, intent(inout) :: ipar(lipar)
   real(kind=8), intent(inout) :: rpar(lrpar)
   real(kind=8), intent(inout) :: s(nfree0), dsdy(nfree0,nxp)

   !CF
   s(1) = y(1) - 0.5d0
   s(2) = y(2)

   !derivatives for shooting function jacobian
   dsdy = 0d0
   dsdy(1,1) = 1d0
   dsdy(2,2) = 1d0

 end Subroutine FinalConditions

 Subroutine Control(y,u,lipar,ipar,lrpar,rpar)
   implicit none
   integer, intent(in) :: lipar, lrpar
   integer, intent(inout) :: ipar(lipar)
   real(kind=8), intent(inout) :: rpar(lrpar)
   real(kind=8), intent(in) :: y(nxp)
   real(kind=8), intent(out) :: u(m)

   !Local
   real(kind=8) :: lambda, p2, signp2

   lambda  = rpar(3)

   p2 = y(ns+2)
   signp2 = sign(1d0,p2)

   if (lambda < 1d0) then
      if (abs(p2) <= lambda) then
         u(1) = 0d0
      elseif (abs(p2) > 2d0 - lambda) then
         u(1) = - signp2
      else
         u(1) = - signp2 * (abs(p2)-lambda) / 2d0 / (1d0-lambda)
      end if
   else
      if (abs(p2) <= 1d0) then
         u(1) = 0d0
      else
         u(1) = - signp2
      end if
   end if

 end Subroutine Control

 Subroutine Dynamics(y,u,f,mode,lipar,ipar,lrpar,rpar)
   implicit none
   integer, intent(in) :: lipar, lrpar
   integer, intent(inout) :: ipar(lipar)
   real(kind=8), intent(inout) :: rpar(lrpar)
   integer, intent(in) :: mode
   real(kind=8), intent(in) :: y(nxp), u(m)
```

```
  real(kind=8), intent(out) :: f(nxp)

  !local
  real(kind=8) :: lambda

  lambda = rpar(3)

  !mode: 0 for state/costate dynamics, 1 for state dynamics only
  f(1) = y(2)
  f(2) = u(1)
  f(3) = lambda * abs(u(1)) + (1d0-lambda)*u(1)**2

  f(ns+1) = 0d0
  f(ns+2) = - y(ns+1)
  f(ns+3) = 0d0

end Subroutine Dynamics

Subroutine Switch(y,psi,lipar,ipar,lrpar,rpar)
  implicit none
  real(kind=8), intent(in) :: y(nxp)
  integer, intent(in) :: lipar, lrpar
  integer, intent(inout) :: ipar(lipar)
  real(kind=8), intent(inout) :: rpar(lrpar)
  real(kind=8), intent(out) :: psi(npsi)

  psi(1) = 1d0 - abs(y(ns+2))

end Subroutine Switch

Subroutine BangBangControl(y,u,lipar,ipar,lrpar,rpar)
  implicit none
  integer, intent(in) :: lipar, lrpar
  integer, intent(inout) :: ipar(lipar)
  real(kind=8), intent(inout) :: rpar(lrpar)
  real(kind=8), intent(in) :: y(nxp)
  real(kind=8), intent(out) :: u(m)

  !local
  real(kind=8) :: p2, signp2

  u = 0d0
  p2 = y(ns+2)
  signp2 = sign(1d0,p2)

  select case (ipar(17))
  case (-1)
     u(1) = -signp2
  case (1)
     u(1) = 0d0
  case default
     write(0,*) 'ERROR: Control >>> Unknown switchflag...',ipar(17)
     stop
  end select

end Subroutine BangBangControl
```

Centre de recherche INRIA Saclay – Île-de-France
Parc Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 Orsay Cedex (France)

Centre de recherche INRIA Bordeaux – Sud Ouest : Domaine Universitaire - 351, cours de la Libération - 33405 Talence Cedex
Centre de recherche INRIA Grenoble – Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier
Centre de recherche INRIA Lille – Nord Europe : Parc Scientifique de la Haute Borne - 40, avenue Halley - 59650 Villeneuve d'Ascq
Centre de recherche INRIA Nancy – Grand Est : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex
Centre de recherche INRIA Paris – Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex
Centre de recherche INRIA Rennes – Bretagne Atlantique : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex
Centre de recherche INRIA Sophia Antipolis – Méditerranée : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399