SOCP (Shooting for Optimal Control Problems)

User Manual

Bruno Hérissé

Last updated: 2019, August

Abstract

This document is a manual for using SOCP in your own applications. This document presents basic functionality to start working with the library. For detailed documentation, use the files generated by doxygen (provided that this software is setup on your computer). It is recalled that the library is released under the 3-clause BSD license. If you use SOCP to produce results in a scientific work, cite it as:

B. Hérissé, *Shooting for Optimal Control Problems (SOCP).*
`https://github.com/bherisse/socp`, 2019.

---

1 Software architecture

This C++ library implements an indirect shooting algorithm for optimal control problems based on the Pontryagin Maximum Principle (see [3] for a review on optimal control methods). It includes additional features such as multiple shooting for switching systems, hybrid systems and also for numerical robustness; a discrete homotopy algorithm for initialization problems; parallel computing to fasten numerical integration. The library has been tested and validated on Windows and Linux platforms.

Figure 1 presents the global architecture of the library as a class diagram. The library consists of two main classes:

- the `model` class is an abstract class that defines the set of functions that must be implemented by the user application. Among these functions, one finds
  - the dynamic model of the system that consists of the state $\dot{x}(t) = f(t, x(t), u(t))$ of the vehicle and the costate $\dot{p} = -\frac{\partial H}{\partial x}(t, x(t), p(t), u(t)))$ (where H is the Hamiltonian of the optimal control problem),
  - a dynamic model integration function that uses the `odeTools` toolbox (Runge-Kutta method of order 4 with fixed steps for example),
  - the function that must be solved by the solver (basically, the two point boundary value problem).

This class being abstract, it can not be instantiated, only the application class (`userModel` in Figure 1) can be instantiated. Notice that some virtual functions of the `model` class have default implementation, but need to be redefined according to the target application.

- the `shooting` class contains the necessary functions to solve the two point boundary value problem implemented in the `model` class. For this, a modified Newton method from the CMinpack library [2] is used. The `mThread` class provides tools for parallel computing if both the computer has multiple CPUs and a multiple shooting method is used.

The `map` class is an optional abstract class, it can be used mainly for modeling state constraints (obstacles, dynamic pressure constraint, etc.) as a penalization function in the cost function of the optimal control problem. However, it could also be used for modeling other state dependent functions. For a given state of the system, it can return the value of the aforementioned function, as well as its gradient. A `model` class can instantiate as many `map` as needed.

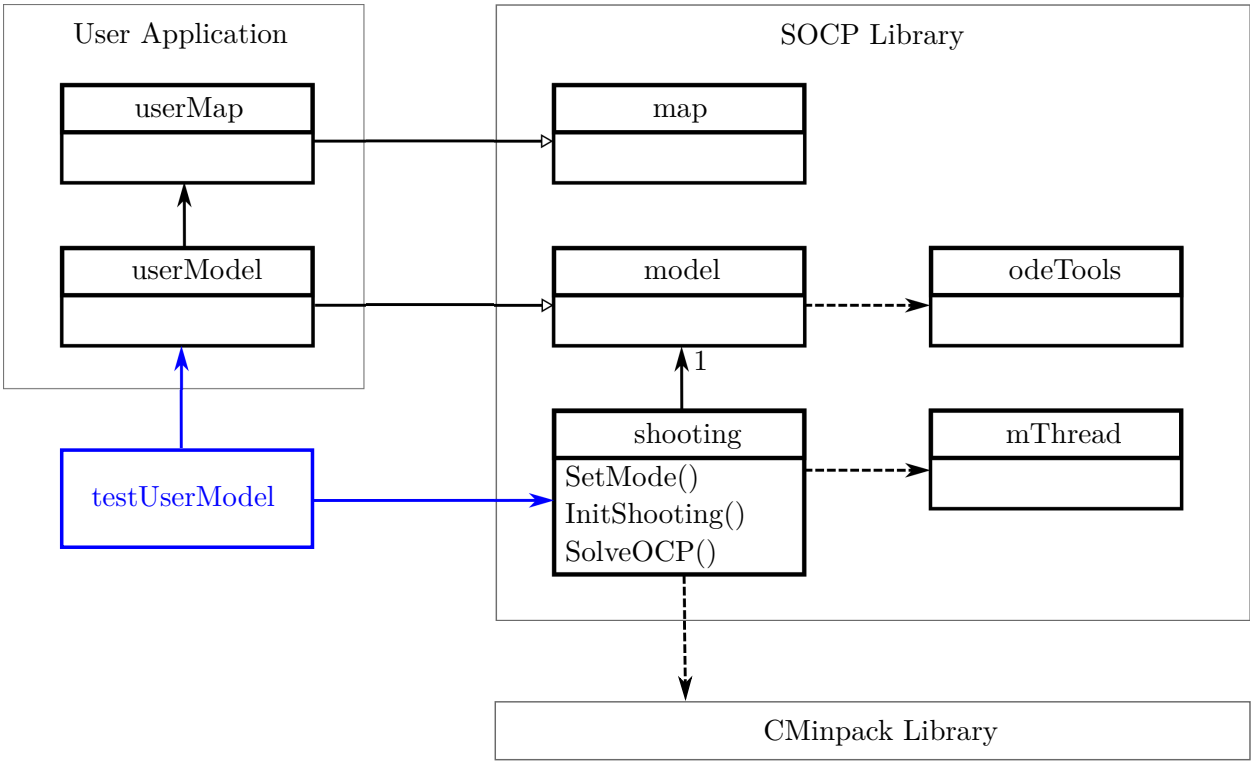

Figure 1: Class Diagram of the SOCP Library

2 Basic principles

To use the SOCP library, it suffices to implement the application class `userModel` and to implement a user test `testUserModel` (see Figure 1). Basically, the user code `testUserModel` should include the following tasks:

1. Create a `my_UserModel` object that instantiates the `userModel` class.

2. Create a `my_Shooting` object that instantiates the `shooting` class by passing `my_UserModel` as an argument when creating the object.

3. Define the boundary value problem using the `SetMode()` function. It consists of defining both free or fixed states (initial, intermediate and final states) and free or fixed times (initial, intermediate and final times).

4. Initialize the problem with the `InitShooting()` function. By calling this function, the fixed states and fixed times are defined. In addition, an estimate of the costates and of the free states/times is provided.

5. Solve the problem by calling the `SolveOCP()` function.

3 Example: Goddard's problem

The Goddard's problem presented in this section is described in details in [1].

3.1 Modeling

In this section, we describe the modeling of the optimal control problem that is implemented in the `goddard` class (see file "src/models/goddard/goddard.cpp").

The vehicle dynamics is:

$$\left\{ \begin{array}{rcl} \dot{r} & = & v \\ \dot{v} & = & -\frac{D(r,v)}{m}\frac{v}{\|v\|} - g(r) + C\frac{u}{m} \\ \dot{m} & = & -b\|u\| \end{array} \right. \tag{1}$$

where:

- r is the position vector, v is the velocity and m is the mass of the vehicle;
- $D(r,v) \geq 0$ is the norm of the drag force, $g(r)$ is the gravity force and $C > 0$ is the maximum thrust force;
- b is a positive parameter that corresponds to the maximum mass flow rate;
- u is the normalized control such that $\|u\| \leq 1$, it corresponds to the main axis of the body of the launcher.

Subject to this dynamics, we search for minimizing the total propellant consumption, that is the following cost:

$$C(u) = \int_0^{t_f} \|u(t)\|\, dt, \tag{2}$$

such that $(r, v, m)(0) = (r_0, v_0, m_0)$ and $r(t_f) = r_f$, $v(t_f)$ and $m(t_f)$ remaining free.

We introduce the Hamiltonian function H as follows:

$$H = \|u\| + \langle p_r, v\rangle - \frac{D(r,v)}{m}\frac{\langle p_v, v\rangle}{\|v\|} - \langle p_v, g(r)\rangle + C\frac{\langle p_v, u\rangle}{m} - b\|u\|p_m \tag{3}$$

Then, the dynamics is extended introducing the costate functions $p_r(t)$, $p_v(t)$ and $p_m(t)$:

$$\left\{\begin{array}{rcl} \dot{r} & = & v \\ \dot{v} & = & -\frac{D(r,v)}{m}\frac{v}{\|v\|} - g(r) + C\frac{u}{m} \\ \dot{m} & = & -b\|u\| \\ \dot{p_r} & = & -\frac{\partial H}{\partial r} \\ \dot{p_v} & = & -\frac{\partial H}{\partial v} \\ \dot{p_m} & = & -\frac{\partial H}{\partial m} \end{array}\right. \tag{4}$$

Applying the PMP, one obtains:

$$u(t) = \left\{\begin{array}{rl} -\frac{p_v(t)}{\|p_v(t)\|} & \text{if} \quad \Psi(t) < 0 \\ 0 & \text{if} \quad \Psi(t) > 0 \\ u_s & \text{if} \quad \Psi(t) \equiv 0 \end{array}\right. \tag{5}$$

where Ψ is the switching function

$$\Psi(t) = 1 - \frac{C}{m(t)}\,\|p_v(t)\| - b\,p_m(t). \tag{6}$$

The control u_s is the singular control that appears when $\Psi(t) \equiv 0$ in a non empty interval. It can be obtained from $\ddot{\Psi}(t) = 0$ (see its expression directly in the code).

From [1], one deduces that the structure of the control is of type Bang-Singular-Off, that is their exist (t_1, t_2) such that:

$$u(t) = \left\{\begin{array}{rl} -\frac{p_v(t)}{\|p_v(t)\|}, & t \in (0, t_1) \\ u_s, & t \in (t_1, t_2) \\ 0, & t \in (t_2, t_f) \end{array}\right. \tag{7}$$

This control sequence being defined, the boundary value problem consists of finding $(p_r(0), p_v(0), p_m(0))$ and (t_1, t_2, t_f) such that

$$\left\{\begin{array}{rcl} \Psi(t_1) & = & 0 \\ \Psi(t_2) & = & 0 \\ r(t_f) & = & r_f \\ p_v(t_f) & = & 0 \\ p_m(t_f) & = & 0 \\ H(t_f) & = & 0 \end{array}\right. \tag{8}$$

3.2 Numerical test

In this section, we present basic usage of the SOCP library to compute optimal trajectory for the Goddard's problem. We assume that the working folder is "tests" (see for example the file "tests/testGoddard.cpp").

To use the Goddard model for shooting, we first need to include the headers:

```
#include "../src/socp/shooting.hpp"
#include "../src/models/goddard/goddard.hpp"
```

Then, the Goddard object is created in the main function (the path to the log file "trace/-goddard/trace.dat" if given as an argument), as well as the shooting object. The Goddard object is passed to the shooting object as an argument. The second argument means that the multiple shooting parameter is nMultiShooting=3: the time interval $(0, t_f)$ is subdivided in 3 intervals $(0, t_1)$, (t_1, t_2) and (t_2, t_f).

```
// new Goddard object
goddard my_goddard(std::string("../../trace/goddard/trace.dat"));
int goddardDim = my_goddard.GetDim(); // dimension of the system
// new shooting object
int nMultiShooting = 3; // multiple shooting parameter
shooting my_shooting(my_goddard, nMultiShooting);
```

The next step consists of defining a time sequence $v_t = \{t_0, t_1, t_2, t_f\}$ and a state sequence $v_X = \{X_0, X_1, X_2, X_f\}$, where $X = (r, v, m, p_r, p_v, p_m)$. The fixed variables $t_0 = 0$, (r_0, v_0, m_0) and r_f can be easily set from the specifications, the other components can only be guessed. Here, the guesses were obtained using a quadratic term in the cost as explained in [1].

```
std::vector<real> v_t(nMultiShooting+1); // time sequence
v_t[0] = 0;      // t_0
v_t[1] = 0.02;   // t_1
v_t[2] = 0.08;   // t_2
v_t[3] = 0.23;   // t_f

std::vector<model::mstate> v_X(nMultiShooting+1); // state sequence
// X_0
v_X[0] = std::vector<real>(2*goddardDim);
v_X[0][0] =      0.999949994;     // x
v_X[0][1] =      0.0001;          // y
v_X[0][2] =      0.01;            // z
v_X[0][3] =      1e-10;           // vx (!=0 to avoid dividing by 0)
v_X[0][4] =      1e-10;           // vy (!=0 to avoid dividing by 0)
v_X[0][5] =      1e-10;           // vz (!=0 to avoid dividing by 0)
v_X[0][6] =      1.0;             // mass
v_X[0][7] =      -7.10102;        // p_x
v_X[0][8] =      0.00679084;      // p_y
v_X[0][9] =      0.679075;        // p_z
v_X[0][10] =     -0.306356;       // p_vx
v_X[0][11] =     0.000464046;     // p_vy
v_X[0][12] =     0.0464035;       // p_vz
v_X[0][13] =     0.060187;        // p_mass
// X_1
v_X[1] = std::vector<real>(2*goddardDim);
v_X[1][0] =      1.00036;         // x
v_X[1][1] =      9.90333e-05;     // y
v_X[1][2] =      0.00990333;      // z
v_X[1][3] =      0.036732;        // vx
v_X[1][4] =      -9.43224e-05;    // vy
v_X[1][5] =      -0.00943201;     // vz
v_X[1][6] =      0.886583;        // mass
v_X[1][7] =      -6.65616;        // p_x
v_X[1][8] =      0.00684417;      // p_y
v_X[1][9] =      0.684408;        // p_z
v_X[1][10] =     -0.235946;       // p_vx
v_X[1][11] =     0.000464632;     // p_vy
v_X[1][12] =     0.0464619;       // p_vz
v_X[1][13] =     0.0440185;       // p_mass
// X_2
v_X[2] = std::vector<real>(2*goddardDim);
v_X[2][0] =      1.00398;         // x
v_X[2][1] =      8.46801e-05;     // y
v_X[2][2] =      0.00846804;      // z
v_X[2][3] =      0.0790628;       // vx
v_X[2][4] =      -0.000410491;    // vy
v_X[2][5] =      -0.0410482;      // vz
```

```
v_X[2][6] =     0.665539;       // mass
v_X[2][7] =     -2.7927;        // p_x
v_X[2][8] =     0.0072468;      // p_y
v_X[2][9] =     0.724672;       // p_z
v_X[2][10] =    -0.188382;      // p_vx
v_X[2][11] =    0.000832271;    // p_vy
v_X[2][12] =    0.0832255;      // p_vz
v_X[2][13] =    0.0129186;      // p_mass
// X_3
v_X[3] = std::vector<real>(2*goddardDim);
v_X[3][0] =     1.01;           // x
v_X[3][1] =     0.0;            // y
v_X[3][2] =     0.0;            // z
v_X[3][3] =     -0.0305203;     // vx
v_X[3][4] =     -0.000571002;   // vy
v_X[3][5] =     -0.0571006;     // vz
v_X[3][6] =     0.595219;       // mass
v_X[3][7] =     -1.38952;       // p_x
v_X[3][8] =     0.00742634;     // p_y
v_X[3][9] =     0.742625;       // p_z
v_X[3][10] =    -1.52212e-09;   // p_vx
v_X[3][11] =    1.24119e-09;    // p_vy
v_X[3][12] =    -1.22946e-09;   // p_vz
v_X[3][13] =    -1.98758e-12;   // p_mass
```

We also define a state mode sequence $m_X = \{m_{X_0}, m_{X_1}, m_{X_2}, m_{X_f}\}$ and a time mode sequence m_t. Since $t_0 = 0$ is fixed and (t_1, t_2, t_f) are free, we set $m_t = \{0, 1, 1, 1\}$ (the mode is 0 for fixed time and 1 for free time). Since (r_0, v_0, m_0) is fixed, we set $m_{X_0} = \{0, 0, 0\}$. The states (r_1, v_1, m_1) and (r_2, v_2, m_2) can be considered as free, but they are not subject to constraints, then we set $m_{X_1} = \{2, 2, 2\}$ and $m_{X_2} = \{2, 2, 2\}$ (practically, this means that the costate variables are continuous at t_1 and t_2). Finally, only the position component of the state (r_f, v_f, m_f) is fixed, the other components are free but constrained by the transversality conditions, then $m_{X_f} = \{0, 1, 1\}$. The corresponding C++ code is

```
std::vector<int> m_t(nMultiShooting+1,0); // mode=0 by default (fixed)
m_t[1] = 1;     // t_1 (free)
m_t[2] = 1;     // t_2 (free)
m_t[3] = 1;     // t_f (free)

std::vector< std::vector<int> > m_X(nMultiShooting+1);
m_X[0] = std::vector<int>(goddardDim,0); // (fixed)
m_X[1] = std::vector<int>(goddardDim,2); // (free unconstrained)
m_X[2] = std::vector<int>(goddardDim,2); // (free unconstrained)
m_X[3] = std::vector<int>(goddardDim,0); // (fixed)
m_X[3][3] = 1;  // vxf (free)
m_X[3][4] = 1;  // vyf (free)
m_X[3][5] = 1;  // vzf (free)
m_X[3][6] = 1;  // mf  (free)
```

It remains to initialize the shooting object with these sequences

```
my_shooting.SetMode(m_t, m_X);  // call it before InitShooting()
my_shooting.InitShooting(v_t, v_X);
```

Now, solve the problem

```
my_shooting.SolveOCP();
```

From this solution, it is possible to solve problems with other parameters using a discrete continuation technique that is implemented in the `shooting` class. For example, it is possible to

change the final desired position from $r_f = (1.01, 0, 0)$ to $r_f = (1.01, 0, 0.005)$. To solve this new problem, we use a continuation step of 0.1 (10%), this means that the algorithm will solve the problem step by step from the initial solution: first it will be solved for $r_f = (1.01, 0, 0.0095)$, then for $r_f = (1.01, 0, 0.0090)$, ..., and finally for $r_f = (1.01, 0, 0.005)$.

```
my_shooting.GetSolution(v_t, v_X);       // get previously computed solution
v_X[3][2] = 0.005;                       // change the position in the z-axis
my_shooting.SetDesiredState(v_t, v_X);   // set the new problem as desired
my_shooting.SolveOCP(0.1);               // continuation parameter =0.1
```

It is also possible to modify a model parameter. For example, to change the parameter b from 7 to 8 from the previously computed solution with continuation step of 0.1, do

```
my_shooting.SolveOCP(0.1,my_goddard.GetParameterData().b,8);
```

References

[1] F. Bonnans, P. Martinon, and E. Trélat. Singular arcs in the generalized goddard's problem. *Journal of optimization theory and applications*, 139(2):439–461, 2008.

[2] F. Devernay. C/C++ minpack. `http://devernay.free.fr/hacks/cminpack/`, 2007.

[3] E. Trélat. Optimal control and applications to aerospace: some results and challenges. *Journal of Optimization Theory and Applications*, 154(3):713–758, 2012.