THE OPTIMAL CONTROL SOLVER

BOCOP 2.1.0 - User Guide

Frédéric Bonnans, Pierre Martinon
Daphné Giorgi, Vincent Grélard, Stéphan Maindrault, Olivier Tissot, Jinyan Liu

October 30, 2017

Contents

1 Bocop overview

The Bocop project aims to develop an open-source toolbox for solving optimal control problems, with collaborations involving industrial and academic partners. Optimal control (optimization of dynamical systems governed by differential equations) has numerous applications in the fields of transportation, energy, process optimization, and biology. It began in 2010 in the framework of the Inria-Saclay initiative for an open source optimal control toolbox, and is supported by the team Commands.

The original Bocop package implements a local optimization method. The optimal control problem is approximated by a finite dimensional optimization problem (NLP) using a time discretization (the direct transcription approach). The NLP problem is solved by the well known software Ipopt, using sparse exact derivatives computed by Adol-C/Colpack or CppAD.

The second package BocopHJB implements a global optimization method. Similarly to the Dynamic Programming approach, the optimal control problem is solved in two steps. First we solve the Hamilton-Jacobi-Bellman equation satisfied by the value function of the problem. Then we simulate the optimal trajectory from any chosen initial condition. The computational effort is essentially taken by the first step, whose result, the value function, can be stored for subsequent trajectory simulations.

Please visit the website for the latest news and updates.

Website: http://bocop.org
Contact: Pierre Martinon (`pierre.martinon@inria.fr`).

1.1 Features

Platforms

The native platform for Bocop is Linux, but Bocop can also be run under MacOS X and Windows. See the specified instructions for each platform.

Interface

The current GUI for Bocop is written in Qt, and we provide visualization scripts for Matlab and Python. Bocop can also be called in command line mode, especially for experienced users. However, we recommend using the GUI, at least for the first steps. Some graphical features such as interpolation for the initial point are much easier to use with the GUI, and it features a buit-in visualization of the solution.

Core

The core files for Bocop are written in C++ and released under the Eclipse Public License. User supplied functions can be written in plain C/C++, and do not require advanced programming skills since they mostly translate the mathematical equations of the problem.

Thirdparty

Bocop currently uses the nonlinear solver Ipopt (with MUMPS as linear solver) for solving the discretized problem resulting from the direct transcription of the optimal control problem. Derivatives of the objective and constraints by automatic differentation are computed by automatic differentiation using ADOL-C (with ColPack for the sparsity), or CppAD.

1.2 Algorithm

The so-called direct approach transforms the infinite dimensional optimal control problem (OCP) into a finite dimensional optimization problem (NLP). This is done by a discretization in time applied to the state and control variables, as well as the dynamics equation. These methods are usually less precise than indirect methods based on Pontryagin's Maximum Principle, but more robust with respect to the initialization. Also, they are more straightforward to apply, hence their wide use in industrial applications. We refer the reader to for instance [4] and [9] for more details on direct transcription methods and NLP algorithms.

Optimal control problem (ODE): noting x the state and u the control

$$(OCP)\begin{cases} \min\ J(x(\cdot),u(\cdot)) = g_0(t_f,x(t_f)) & \text{Objective (Mayer form)} \\ \dot{x}(t) = f(t,x(t),u(t)) \quad \forall t \in [0,t_f] & \text{Dynamics} \\ u(t) \in U \quad \text{for a.e. } t \in [0,t_f] & \text{Admissible Controls} \\ g(x(t),u(t)) \leq 0 & \text{Path Constraints} \\ \Phi(x(0),x(t_f)) = 0 & \text{Boundary Conditions} \end{cases}$$

Summary of the time discretization:

$t \in [0,t_f]$	$\rightarrow$	$\{t_0 = 0, \ldots, t_N = t_f\}$
$x(\cdot), u(\cdot)$	$\rightarrow$	$X = \{x_0, \ldots, x_N, u_0, \ldots, u_{N-1}, t_f\}$
Criterion	$\rightarrow$	$\min\ g_0(t_f, x_N)$
Dynamics $(ex: Euler)$	$\rightarrow$	$x_{i+1} = x_i + hf(x_i,u_i) \quad i = 0,\ldots,N-1$
Admissible Controls	$\rightarrow$	$u_i \in \mathbf{U} \quad i = 0,\ldots,N$
Path Constraints	$\rightarrow$	$g(x_i,u_i) \leq 0 \quad i = 0,\ldots,N$
Boundary Conditions	$\rightarrow$	$\Phi(x_0,x_N) = 0$

We therefore obtain a nonlinear programming problem on the discretized state and control variables

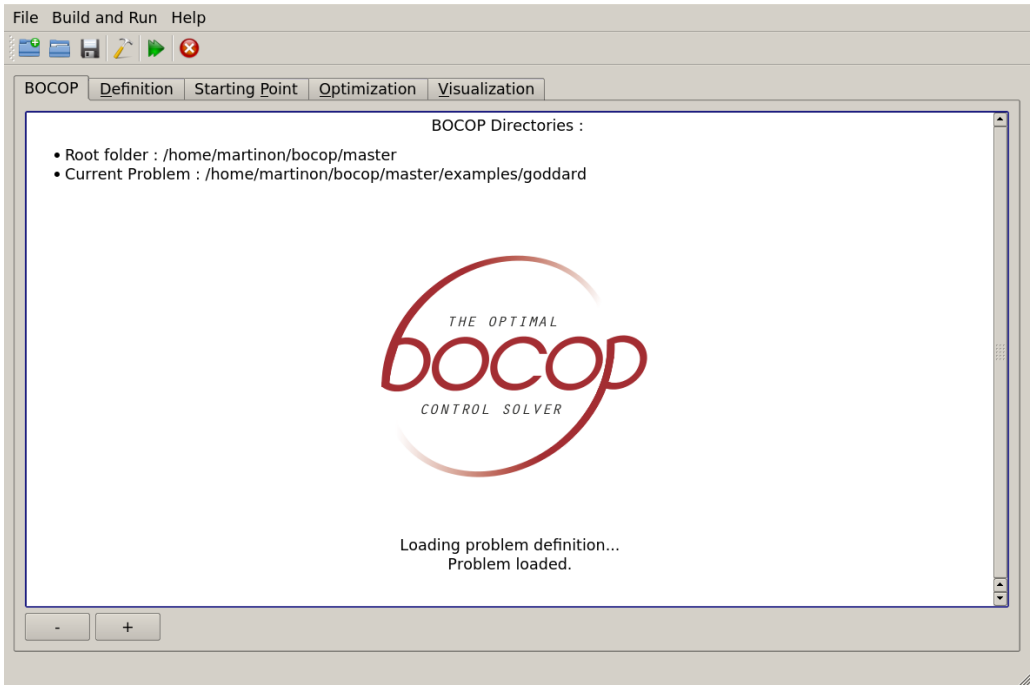
$$(NLP)\begin{cases} \min\ F(X) \\ LB \leq C(X) \leq UB \end{cases}$$

In Bocop the discretized nonlinear optimization problem is solved by the Ipopt solver [11] that implements a primal-dual interior point algorithm. The derivatives required for the optimization are computed by the automatic differentiation tool Adol-C, see [12].

2 Getting started

Bocop files include
- core files: sources files for Bocop
- thirdparty files: thirdparty software used by Bocop
- problem files: each problem has its own directory in which definition, input and output files are stored. You can choose any location for your problem files.

Bocop features are organized in 4 main modules



Qt GUI: Main window

1. Problem Definition
   Define the optimal control problem, by providing dimensions and names for the variables, as well as the functions for the objective, dynamics and constraints. Editing one of the functions is the only case where (re)compilation is needed. Other parameters such as bounds, constants, or even dimensions, can be modified without recompilation. Of course, the functions should still be consistent with the dimensions provided. Definition includes the choice of the discretization method and number of steps, in order to transform the optimal control problem into a nonlinear programming problem. Changing these settings does not require recompilation, and can be used to refine a solution from a coarse discretization to a more accurate one. Any implicit Runge Kutta method can be used, predefined methods include Euler, Midpoint, RK4, Gauss, and Lobatto formulas.

2. Starting Point

Set the starting point for the discretized NLP problem at iteration 0. As the nonlinear programming solver uses a local method, convergence may depend strongly on having a good starting point. Bocop features a graphical interface to generate initial trajectories, using constant, piecewise linear, splines functions or external file. It is also possible to use an existing solution file as initialization, even with a different discretization (see optimization tab). Note: this *starting point* for the NLP iterations should not be confused with the *initial point* of the optimal control problem, ie the values of the state variables at initial time t_0.

3. Optimization
   The "Optimize" button launches Ipopt to solve the discretized optimization problem. This actually runs the **bocop** executable in the current problem directory. The options for the NLP solver IPOPT can be set here, and are saved in a standard configuration file *ipopt.opt*. This tab also includes some optional features such as batch optimization, parameter identification, and the choice of an existing solution as starting point.

4. Visualization
   Read and display the contents of the solution file generated after optimization. Visualization includes state and control variables, as well as scalar optimization parameters (for instance free final time). Constraints, including dynamics constraints, are also shown, as well as their multipliers. Groups of variables can be shown together by double-clicking on the group lablel after the + symbol (single click will expand the group). Variables are drawn as functions of the independent variable (ie, time); two variables within a group can be drawn as a $y(x)$ graph by selecting the first variable x, holding **Ctrl** then selecting the second variable y.

2.1 Test problem: Goddard

Note: The files corresponding to the Goddard problem are given in appendix D p.37.

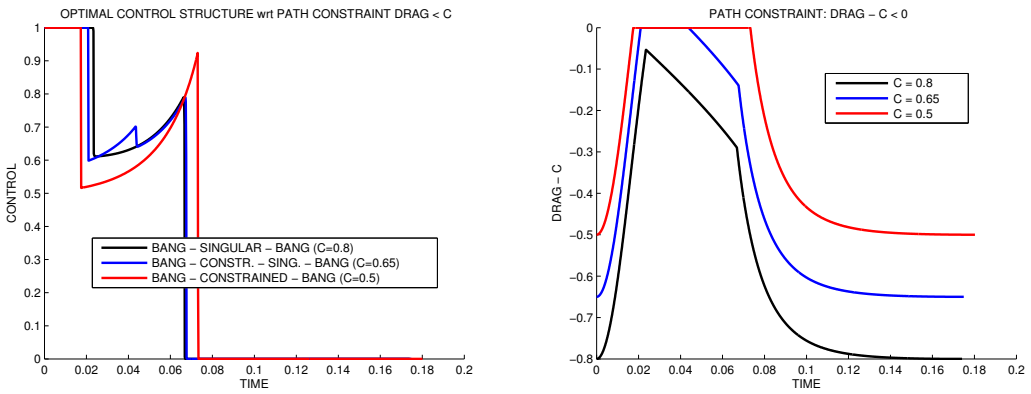
This well-known problem (see for instance [6, 10]) models the ascent of a rocket through the atmosphere, and we restrict here ourselves to vertical (monodimensional) trajectories. The state variables are the altitude, speed and mass of the rocket during the flight, for a total dimension of 3. The rocket is subject to gravity, thrust and drag forces. The final time is free, and the objective is to reach a certain altitude with a minimal fuel consumption, ie a maximal final mass. All units are renormalized.

$$(P_3)\begin{cases} \max\ m(T) \\ \dot{r} = v \\ \dot{v} = -\frac{1}{r^2} + \frac{1}{m}(T_{max}u - D(r,v)) \\ \dot{m} = -bu \\ u(\cdot) \in [0,1] \\ r(0) = 1,\ v(0) = 0, m(0) = 1, \\ r(T) = 1.01 \\ D(r(\cdot), v(\cdot)) \leq C \\ T\ free \end{cases}$$

The drag is $D(r,v) := Av^2\rho(r)$, with the volumic mass is $\rho(r) := exp(-k*(r-r0))$. We use the parameters $b = 7$, $T_{max} = 3.5$, $A = 310$, $k = 500$ and $r0 = 1$.

The Hamiltonian is an affine function of the control, so singular arcs may occur. We consider here a path constraint limiting the value of the drag effect: $D(r,v) \leq C$. We will see that depending on the value of C, the control structure changes. In the unconstrained case, the optimal trajectory presents a singular arc with a non-maximal thrust. When C is set under the maximal value attained by the drag in the unconstrained case, a constrained arc appears. If C is small enough, the singular arc is completely replaced by the constrained arc.

• *Numerical simulations: problem* ***goddard***
Discretization: Gauss (4th order) 1000 steps.



Goddard problem: from ***Bang-Singular-Bang*** *to* ***Bang-Constrained-Bang*** *structure due to drag path constraint*

3 Solving your problem

3.1 Optimal control problem

We consider an optimal control problem of the form:

$$(P)\begin{cases} Min\ J(t_0, y(t_0), t_f, y(t_f), \pi) & \text{Objective} \\ \dot{y}(t) = f(t, u(t), y(t), z(t), \pi) & \text{Dynamics} \\ \Phi_l \leq \Phi(t_0, y(t_0), t_f, y(t_f), \pi) \leq \Phi_u & \text{Boundary conditions} \\ y_l \leq y \leq y_u,\ u_l \leq u \leq u_u,\ z_l \leq z \leq z_u,\ \pi_l \leq \pi \leq \pi_u & \text{Bounds} \\ g_l \leq g(t, u(t), y(t), z(t), \pi) \leq g_u & \text{Path constraints} \end{cases}$$

with $y(\cdot)$ the state variables, $u(\cdot)$ the control, $z(\cdot)$ the optional algebraic variables and π the optional variables to be optimized (unlike numerical constants of the problem).

Notes:
- We assume there is no integral criterion (Mayer formulation). If needed, define an additional state variable containing the objective value.
- If the final time is free, time will be internally normalized to $[0, 1]$ *and the dynamics multiplied by* $t_f - t_0$*. The value of* t_f *is automatically added at the end of* π *(therefore, do not count* t_f *when setting the dimension of* π*).*

In Bocop, the problem (P) is defined by

- **4 functions (C/C++)** corresponding to J, f, Φ, g:
  `criterion`, `dynamics`, `boundarycond`, `pathcond`. Each one of these functions is written in a *.tpp* file, which can be edited with any text editor.

- **3 definition files (text)**:
  - *problem.def* for general definition and settings (see B.1 p.30)
  - *problem.bounds* for the bounds (see B.2 p.31)
  - *problem.constants* contains the optional constant values for the problem. These values remain unchanged and are not part of the optimization process, unlike the optimization variables π.

Each problem has its own directory with all its files. These files are typically generated through the GUI, which allows to create, load and save problems. Experienced users may prefer to edit the .def, .bounds and .constants files directly. You can also copy the files from another problem and then modify them either through the GUI or directly.

3.2 Problem definition

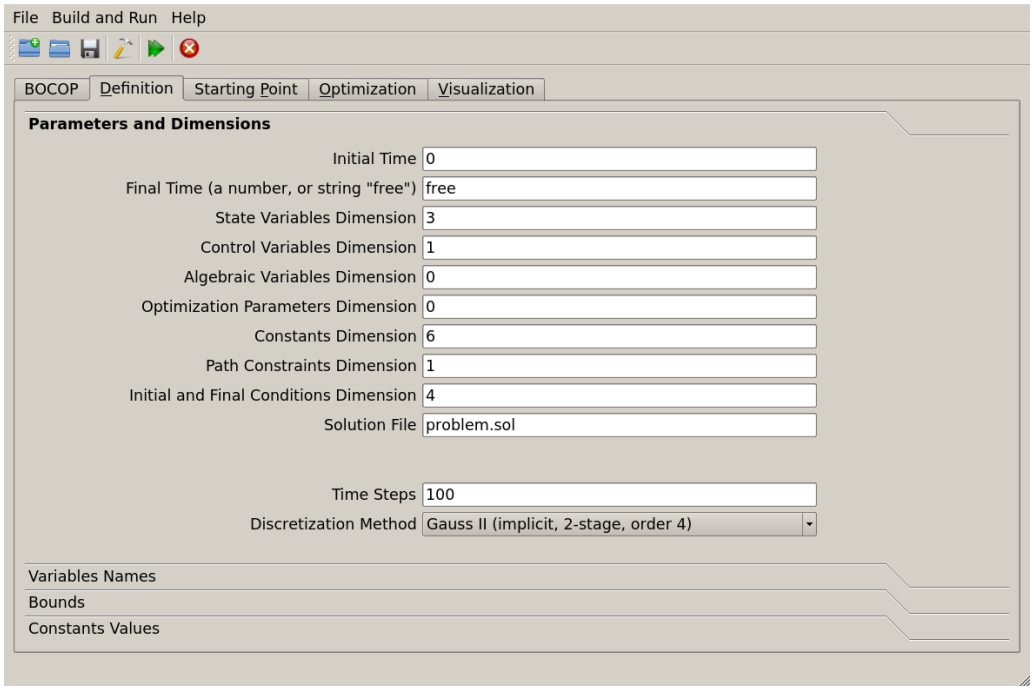
Select "New problem" and follow the steps to define your problem, discretization, initialization and NLP options. Bocop will ask for a directory in which all files related to the problem will be written.

3.2.1 Problem general definition

Set here the dimensions for the variables and constraints of your problem, as well as the initial and final times (set "`free`" if t_f is free). We recommend renaming each variable

and condition (default names are `state.0`, `state.1`, `control.0`, etc).

Warning: be careful not to use spaces in variable or constraints names !



Qt GUI: definition

3.2.2 Time discretization

The default list of discretization formulas includes: Euler (explicit and implicit), Midpoint (implicit), Gauss II and Lobatto III. We refer interested readers to for instance [7, 8] for more details on ODE integration schemes.

Method	stages	order
Euler (explicit)	1	1
Euler (implicit)	1	1
Midpoint (implicit)	1	2
Gauss II (implicit)	2	4
Lobatto III C (implicit)	3	6

3.2.3 Problem functions

- criterion: $J(t_0, y(t_0), t_f, y(t_f), \pi)$
- dynamics: $\dot{y}(t) = f(t, u(t), y(t), z(t), \pi)$
- boundary conditions: $\Phi(t_0, y(t_0), t_f, y(t_f), \pi)$
- path conditions: $g(t, u(t), y(t), z(t), \pi)$

Bocop provides templates files for these 4 functions, so you just have to fill in the equations. The syntax is standard C/C++, therefore **arrays begin at index 0**. See

D.2, p.37 for an example. Note also that Bocop provides pre-defined methods which can be used in these 4 functions definition. See B.4, p.32 for the complete list.

Important note on variables type: for real variables (as opposed to integer), the type to use is either the classic C "double", or the template "Tdouble". Any value that can change during the optimization, such as state or control variables and optimization parameters, must be declared as a "Tdouble". Values that remain constant during optimization can use the normal "double" type. Note that any expression containing a Tdouble value is also of type Tdouble. The Tdouble types are used for the computation of derivatives by automatic differentiation (Adol-C).

Once done, you can build the executable by clicking the "Build problem" button. This will launch cmake and make processes and generate an executable file "bocop" (or bocop.exe for Windows) in your problem folder. Standard build is in release mode, you can also choose a debug build in the build menu. Debug builds include additional compiler messages and internal checks to help spot errors, but are typically slower. Note that you should clean the problem files if you change the type of build (option "Clean problem" in build menu). You must rebuild the problem each time the C++ files are modified.

Command line build and run

You can also build and run bocop directly in a terminal, without using the GUI. This may be useful e.g. for scripting. To build bocop, go into your problem folder and create a build/ folder. In the build/ folder, execute the commands

```
cmake -DPROBLEM_DIR:PATH=<pathToProblem> <pathToBocopPackage>
make
```

- add the flag `-DCMAKE_BUILD_TYPE=Release` for a non-debug build
- on windows: `cmake -G ''MSYS Makefiles'' -DPROBLEM_DIR:PATH=<pathToProblem> <pathtoBocopPackage>`

You should then have a bocop executable in your problem folder, run it with `./bocop` (on windows: bocop)

External functions

You can use external functions when defining the four functions for Bocop. In this case, complete the two files *dependencies.hpp* and either *dependencies.tpp* or *dependencies.cpp*, which contain the declaration and location of the external functions

- All functions must be declared in *dependencies.hpp*.
- A function using Tdoubles must begin with **template < class Tdouble >**, is written in a .tpp file, and included in *dependencies.tpp*.
- A function having no Tdouble arguments must be written in a regular .cpp file, and included in *dependencies.cpp*.

See appendix D p.37 for an example.

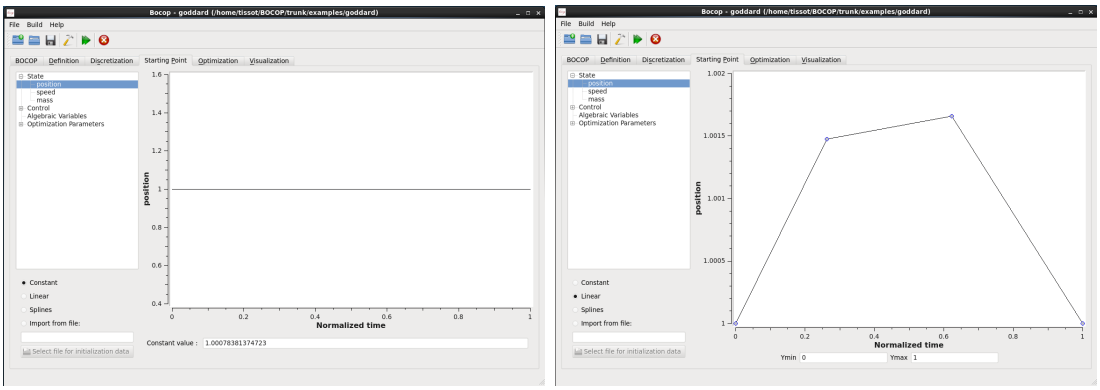
3.2.4 Bounds

Set here the bounds for y, u, z, π as well as Φ, g. For each component, you can specify a lower bound only, an upper bound only, both, an equality condition, or no bounds.

3.3 Starting Point

Bocop offers four kinds of initialization for the discretized values of y, u, z.
- **Constant**: constant value over the time interval $[0, T]$.
- **Linear**: piecewise linear interpolation over the gridpoints given by the user.
- **Splines**: same as above with a spline interpolation on the given points.
- **Import from file**: use external file to initialize a variable. This is for advanced users because the file has to be formatted as a Bocop .init file.

You can choose a different initialization type for each component of the state and control variables. Initial guesses are also required for the scalar optimization variables π (e.g. free final time). Both the GUI and bocop executable will use a default value of 0.1 for the initialization, unless specified otherwise.



Qt GUI: constant and linear initialization

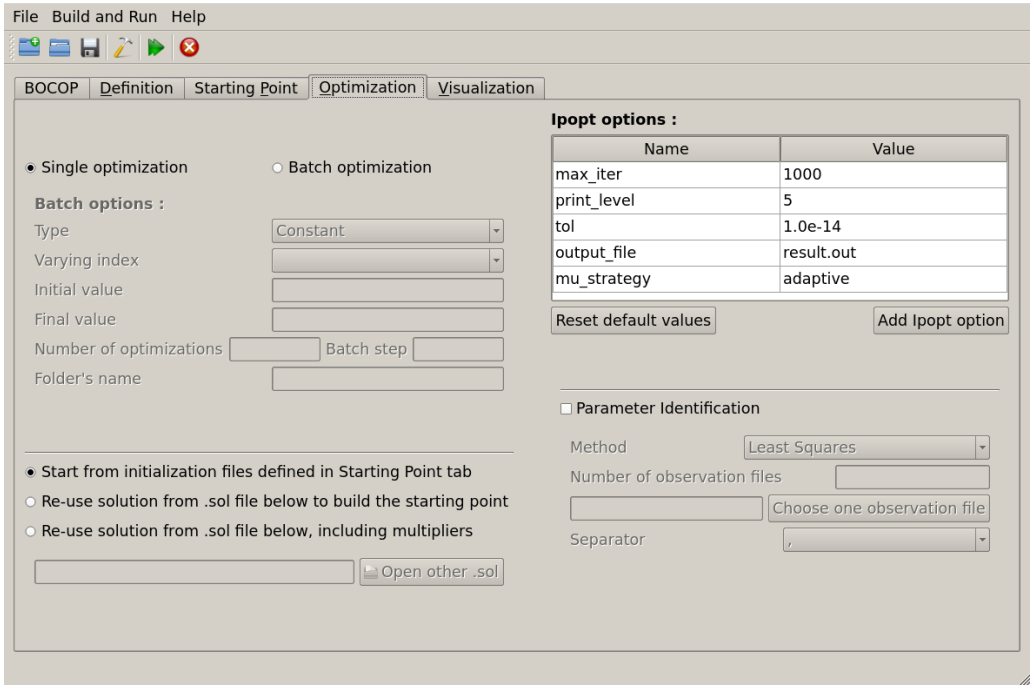
If using the interpolation, first set the number and range of gridpoints. Then edit the gridpoints, and set the values in the window by dragging the mouse. New gridpoints can be added directly by clicking on the line (not too close from existing points). Moving a point to an existing one will merge the two. Note that you do not have to put points exactly at 0 and T: the function will be interpolated over $[0, T]$ regardless. Select "Ok" in the menu when finished. You can then visualize the interpolated function (linear or splines). Note that a spline initialization is not necessarily "better" than a piecewise linear one. The best choice depends mostly on the shape of the function you want to provide. In the example below, you can see how to approximate a discontinuity using a piecewise linear function. As you can see, the spline interpolation would lead to a completely different initialization (be careful of bounds when using spline interpolation).

Warm start. *Additionaly, you can use an existing solution file to build the starting point. See 3.4.2.*

3.4 Optimization

Set here the options for the NLP solver Ipopt. This will generate a standard *ipopt.opt* configuration file in the current problem directory. You can manually edit this file as well before launching the optimization.

The "Run" button will launch the NLP solver, and is the same as running the executable "bocop" in the current problem directory. In the GUI you can follow the optimization in the first tab.



Qt GUI: optimization

3.4.1 Return codes

The bocop executable will return the error code from the solver Ipopt. Here is the complete list of return codes in the file *IpReturnCodes_inc.h* from the Ipopt package. In short, this means that besides the classical 0, a return code of 1 usually indicates a successful convergence. Users may also want to check solutions with a return code of 3, which may be acceptable as well. We refer the users to the ipopt manual for a thorough description of these return codes.

```
/** Return codes for the Optimize call for an application */
enum ApplicationReturnStatus
  {
    Solve_Succeeded=0,
    Solved_To_Acceptable_Level=1,
    Infeasible_Problem_Detected=2,
    Search_Direction_Becomes_Too_Small=3,
```

```
    Diverging_Iterates=4,
    User_Requested_Stop=5,
    Feasible_Point_Found=6,

    Maximum_Iterations_Exceeded=-1,
    Restoration_Failed=-2,
    Error_In_Step_Computation=-3,
    Maximum_CpuTime_Exceeded=-4,
    Not_Enough_Degrees_Of_Freedom=-10,
    Invalid_Problem_Definition=-11,
    Invalid_Option=-12,
    Invalid_Number_Detected=-13,

    Unrecoverable_Exception=-100,
    NonIpopt_Exception_Thrown=-101,
    Insufficient_Memory=-102,
    Internal_Error=-199
  };
```

Please note that in the terminal an exit value outside $[0, 255]$ returns an exit code modulo 256. For example, exit -2 gives an exit code of 254 (-2 % 256 = 254).

3.4.2 Use an existing solution as starting point (warm start)

The option **"Start from initialization files defined in Starting Point tab"** will launch the optimization using the initialization values defined in Starting point tab.

The option **"Re-use solution form .sol file below to build the starting point"** will launch the optimization using a previous solution as an initial guess, instead of the one defined in the Starting Point tab. This can be useful to refine an existing solution while modifying some parameters (for instance constants, but also the discretization formula or number of steps), or to improve convergence by using a solution from a simpler version of the problem. The only requirement is that the dimensions of the variables (state, control, parameters) must match.

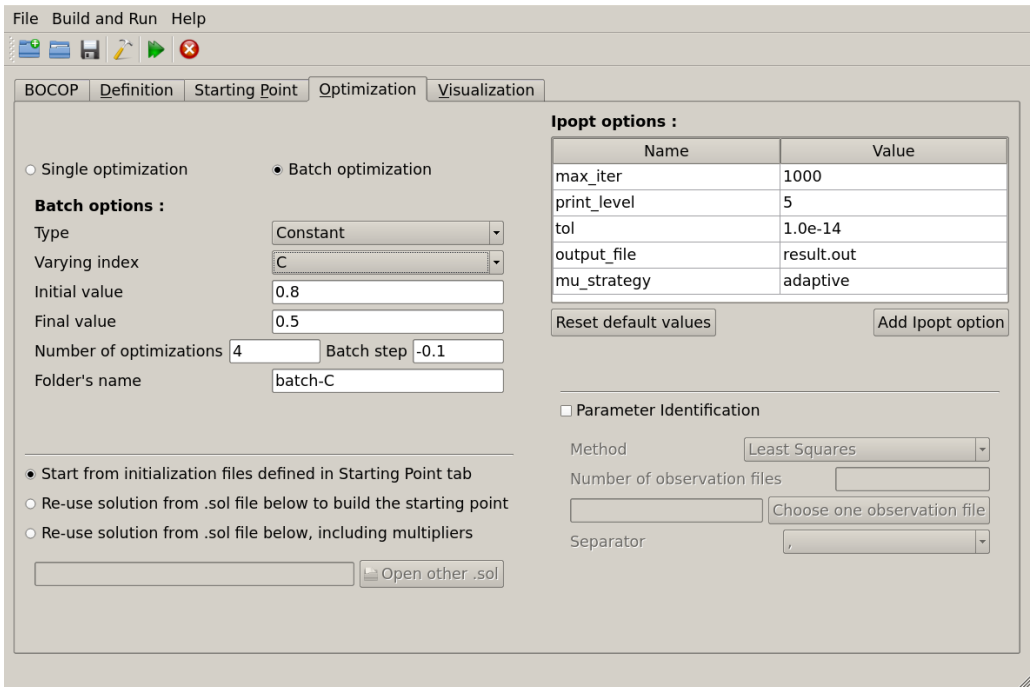
The solution file used for the initialization is indicated in the textfield in the visualization module. By default, Bocop will use the file "problem.sol" in the current problem directory. Only the primal variables are initialized, the multipliers for the constraints are reset to their default starting values. This corresponds to the so-called 'cold start' option in Ipopt.

The option **"Re-use solution form .sol file below, including multipliers"** is similar to the previous one, but will also initialize the multipliers using the data from the provided solution file. This corresponds to the so-called 'warm start' option in Ipopt. Generally speaking, this second option will give a faster convergence if the structure of the two solutions is close. If the solutions differ too much, actually give better results.

Batch optimization. *The two options above can be used in batch optimization (see 3.4.3), leading to a discrete continuation algorithm in which each problem is solved in sequence using the solution from the previous one as starting point. The first problem will still use the starting point defined in the Starting Point tab.*

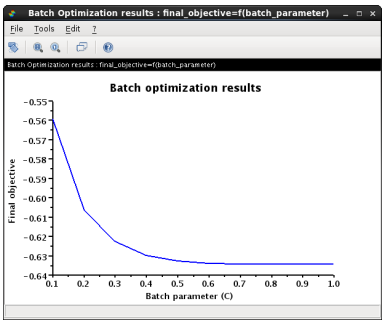
3.4.3 Batch optimization

Batch optimization allows you to launch an automatic sequence of optimization with a varying value for the time steps number or for one of the constants of the problem. Bocop will ask for the range and number of values to consider for the varying value. Solution files are stored in a separate folder, and a log file is created for the batch. This is useful for instance to explore the impact of a constant on the solution of the problem (objective value, convergence) or to observe the evolution of the convergence speed and of the precision of the solution depending on the time discretization.



You can find below an illustration of the batch optimization for the Goddard problem. Here we test different values for the upper bound enforced on the dynamic pressure. The graph plots the objective (minus final mass, as we maximize $mass(tf)$) with respect to the limit. We observe that the objective gets better when the constraint is less strict (ie a greater upper bound), and remains constant above a certain threshold, as the constraint becomes inactive.

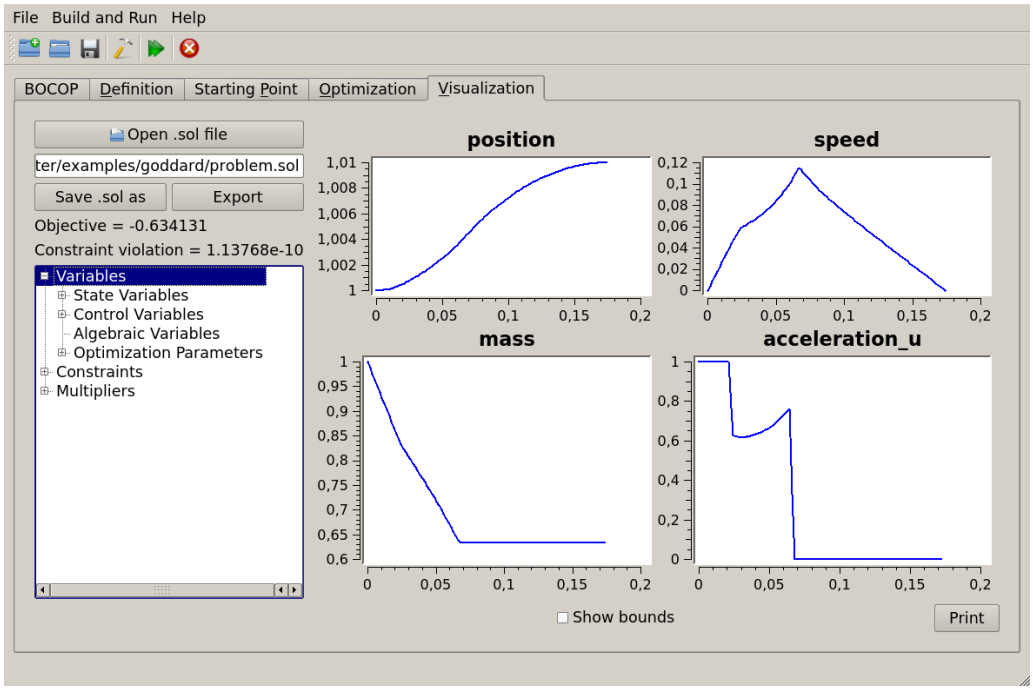
```
# ** Batch optimization parameters **
# Index of the varying constant : 5
# Lower value of this constant : 0.1
# Upper value of this constant : 1
# Number of steps in this interval (number of optimizations) : 10

# ** Optimization results **
#  n   | constant value  | final objective  |  constraint viol. | nb iter. | status |
1        0.1               -0.559391          1.05226e-07         115        0
2        0.2               -0.606197          3.03664e-08         45         0
3        0.3               -0.6226            4.64634e-07         36         0
4        0.4               -0.62968           7.12004e-07         17         0
5        0.5               -0.632737          6.45525e-07         35         0
6        0.6               -0.633834          8.12976e-07         34         0
7        0.7               -0.634064          5.24183e-08         55         0
8        0.8               -0.634072          1.67586e-07         33         0
9        0.9               -0.634072          1.25914e-08         27         0
10       1                 -0.634074          6.43808e-10         26         0

# Successful optimizations : 10/10
```



Discrete continuation. *On the other hand, batch optimization can also be combined with re-optimization: by choosing one of the "Re-use solution" buttons, each problem in the batch will use the solution of the previous problem as initial point. This is a way to perform a simple continuation method, if the original problem is too difficult to solve. This case is illustrated on the goddard problem with a progressive introduction of a dynamic pressure constraint.*

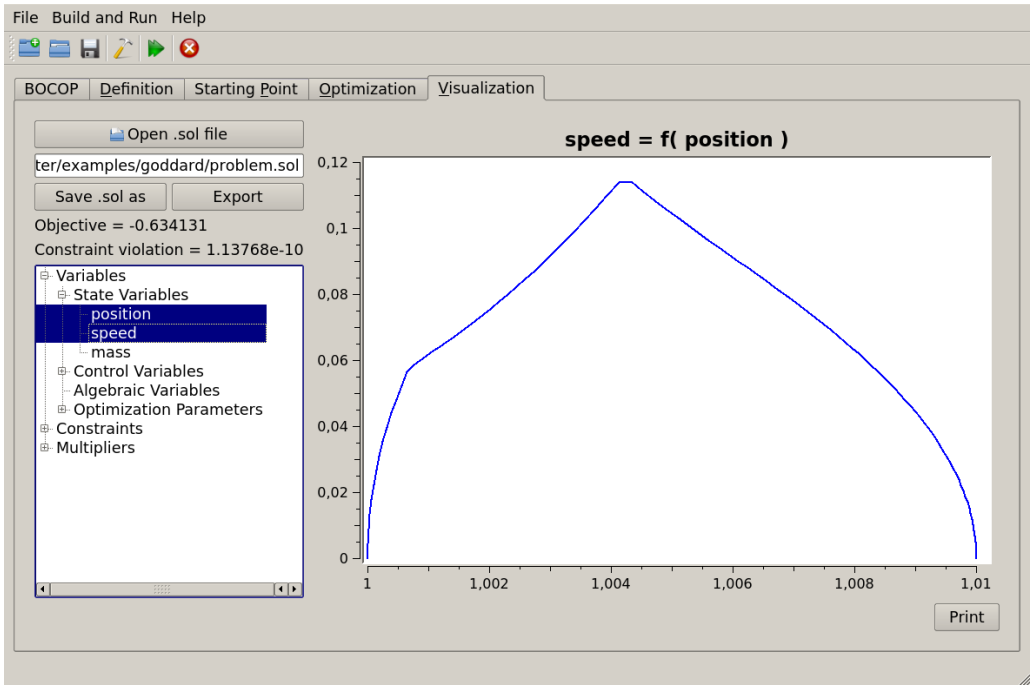
3.5 Visualization and Export

The display menu allows you to visualize the discretized variables (state, control) and the scalar optimization parameters. The value of boundary conditions and path constraints can also be checked (as well as the discretized dynamics constraints). Finally, the multipliers are divided in three groups: multipliers the boundary conditions, multipliers for the path constraints, and the adjoint states (cf Pontryagin's Maximum Principle).



Qt GUI: visualization

You can either display grouped graphs by double-clicking a category name, or single graphs by expanding the list and double-clicking an item. Moreover, you can display a variable depending on another by pressing CTRL[1] and selecting the two variables to display.



Qt GUI: visualization of one state depending on another

You can also use matlab or python commands to plot custom graphs, using the variables read from the .sol file. See the functions provided in the "scripts" folder.

From the visualization window you can export the solution in raw text files: 1 file per component, 1 value per line. This files may be easier to read from different environments. By default, all components are exported. You can choose the target directory for the export files. All files have the suffix *.export*. The file *state_times.export* contains the grid points of the time discretization, that match the state variables values. The file *control_times.export* contains the interior points for each stage of the time discretization, that match the control variables values.

3.6 Parameters identification

If there are unknown constants in the model, a parameters identification is necessary before launching the optimization. In this case we suppose to have some observation data, that come from an experiment at fixed control.
Here below we show how to proceed to make a parameters identification, referring to the

[1] on Linux and Windows; or the Command key on Mac

Jackson problem for the illustrations.

The observation data have to be stored in a $*.csv$ file, respecting the following order:

- Time : one column of observation times.
- Observations: one matrix of observation values.
- Weights: one matrix of weights corresponding to the observations. If an observation is missing at time t_i, the corresponding weight is set to 0.

The columns separators can be choosen between ',', ';', ':', ⟨Tab⟩ and ⟨Space⟩.

```
1 # Data file for parameter identification
2 # taken from jackson solution with u= 0.5
3
4 # Time          Observations            Weights
5 0.4,    0.9144, 0.0753, 0,      1,      1,      0
6 0.8,    0.8928, 0.0810, 0,      1,      1,      0
7 1.2,    0,      0,      0.0424, 0,      0,      1
8 1.6,    0.8627, 0.0791, 0,      1,      1,      0
9 2,      0,      0,      0.0739, 0,      0,      1
10 2.4,   0.8342, 0.0764, 0,      1,      1,      0
11 3.2,   0.8067, 0.0739, 0,      1,      1,      0
12 4,     0.7799, 0.0715, 0.1488, 1,      1,      1
```

Figure 1: Comma-separated observation data for Jackson problem

We must declare a theoretical measure function, which is coded in C++, and saved in the file *measure.tpp*.

```
Functions
Measures
// Tdouble variables correspond to values that can change during optimization:
// states and optimization parameters.
// Values that remain constant during optimization use standard types (double, int, ...).

#include "header_measure"
{
   // MEASURES FOR JACKSON PROBLEM

   switch (k) {
   case 0 :
            measures[0] = state[0];
            measures[1] = state[1];
            break;
   case 1 :
            measures[0] = state[2];
            break;
   }
}
```

Figure 2: *measure.tpp* file

We define the dynamic of the problem as usual, but we declare the parameters as optimization parameters, set to $Tdouble$ type, instead of constants.

In the BOCOP Optimization tab, we enable the *Parameter identification* feature and select :

- Observation file: name of the observation file, which has to be located in the current problem folder.
- Separator: select the character that is used in the observation file to separate the columns.

Functions

Dynamics

```
// Output :
// state_dynamics : vector giving the expression of the dynamic of each state variable.

#include "header_dynamics"
{
        // DYNAMICS FOR JACKSON PROBLEM

        Tdouble a = state[0];
        Tdouble b = state[1];
        Tdouble u = control[0];
        //double k1 = constants[0];
        //double k2 = constants[1];
        //double k3 = constants[2];

        Tdouble k1 = optimvars[0];
        Tdouble k2 = optimvars[1];
        Tdouble k3 = optimvars[2];

        state_dynamics[0] = -u*(k1*a-k2*b);
        state_dynamics[1] = u*(k1*a-k2*b) - (1-u)*k3*b;
        state_dynamics[2] = (1-u)*k3*b;
}
```

Figure 3: *dynamics.tpp* file

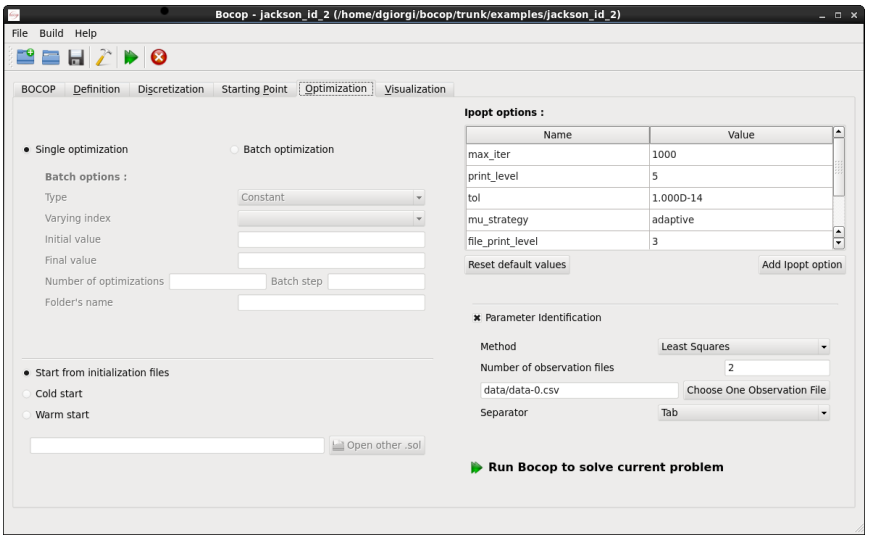


Figure 4: BOCOP Parameter identification feature

- Method: choose between
  - Least squares: the criterion is

$$\sum_k \sum_i w_{ki}(\varphi_i(t_k, y_{t_k}, \vartheta) - m_{ki})^2$$

  - Least squares with criterion: the criterion is

$$J(t_0, y(t_0), t_f, y(t_f), \pi) + \sum_k \sum_i w_{ki}(\varphi_i(t_k, y_{t_k}, \vartheta) - m_{ki})^2$$

– Manual: the criterion is

$$J(t_0, y(t_0), t_f, y(t_f), \pi) + \sum_k \varphi_0(t_k, y_{t_k}, \vartheta)$$

$\varphi_0(t_k, y_{t_k}, \vartheta)$ will be defined by users in the file *measure.tpp*.

where we note

$\varphi_i(t_k, y_{tk}, \vartheta) \in \mathbb{R}^p$	:	theoretical measure function
m_{ki}	:	observation values
w_{ki}	:	weights of the observations
ϑ	:	parameters vector

We can now launch an optimization, making sure that we fixed the same control that was used for the experiment. The optimization parameters will give us the values of the identified parameters. We can now replace these values in the dynamics, declaring them as constants, and launch the optimization without parameter identification.

3.7 Delay problems

Since version 2.1.0 Bocop can handle some kinds of delay problems. Generally speaking, we refer to 'delay problems' as optimal control problems for which some of the OCP functions depend on past state and control variables, namely the dynamics

$$\dot{y}(t) = f(t, u(t), y(t), u(t-\tau_1), \cdots, u(t-\tau_M), y(t-\tau_{M+1}), \cdots, u(t-\tau_{M+N}))$$

and path constraints

$$g_l \leq g(t, u(t), y(t), u(t-\tau_1), \cdots, u(t-\tau_M), y(t-\tau_{M+1}), \cdots, u(t-\tau_{M+N})) \leq g_u.$$

In both functions, defined in *dynamics.tpp* and *pathcond.tpp*, the values of $y_i(t-\tau)$ or $u_i(t-\tau)$ can be computed by interpolation on the time steps or stages respectively, using the function **interpolation** defined in `core/headers/publicTools.hpp`. This function takes two vectors X, Y of same size N and a value x, and returns the interpolated value y at x (using linear interpolation). The optional parameter *verbose* has a default value of 0; setting it to 1 will issue warnings if x is outside the range of X. Note that the interpolation will return Y_1 if $x \leq X_1$ and Y_N if $x \geq X_N$.

```
Tdouble interpolation(const double x_value, const vector<double> &x_data,
const vector<Tdouble> &y_data, const int set_data_size, const int verbose)
```

Here are two sample code snippets computing the value of past states and controls. Note that the expression $t-\tau$ is computed using the **normalized times** $s \in [0, 1]$ instead of the absolute times $t \in [t_0, t_f]$ for reasons related to the computation of derivatives by automatic differentiation. Therefore we interpolate at time $s - \tau/(t_f - t_0)$. The values for times before t_0 must be set explicitely for $t < t_0 + \tau$, unless they are supposed to be equal to the values at t_0 (since the interpolation would return these initial values).

```
...
double tau = constants[0];
Tdouble yi_past;

if (normalized_time < tau/fixed_final_time)
  yi_past = ...
else
  yi_past = interpolation(normalized_time-tau/fixed_final_time,
  past_steps,past_states[0],current_step);
```

or

```
...
double tau = constants[0];
Tdouble ui_past;

if (normalized_time < tau/fixed_final_time)
  ui_past = ...
else
  ui_past = interpolation(normalized_time-tau/fixed_final_time,
  past_stages,past_controls[i],current_stage);
```

Remarks
- Bocop can only handle delay problems with fixed final time. This is due to the interpolation requiring the past time $t - \tau$ to be always in the same time sub-interval $[t_i, t_{i+1}]$. With a free final time, the time steps will change according to the value of t_f, therefore the index i corresponding to $t-\tau$ could change. An exception would be the case of a delay proportional to the final time, so that $t-\tau$ would stay in the same sub-interval regardless of the value of t_f. Technically, the past time $t-\tau$ must be given by a non-varying expression of type **double**, and not a varying expression of type **Tdouble**. Therefore one must use the **normalized time** values. For free final time problems, one can reformulate the problem with fixed final time and try a batch optimization while varying the constant t_f.
- since the functions can access the whole sequence of past state and control variables, several different delays can appear in a single function.
- note that while only the past values of the control and state variables are currently passed to the functions, technically the *future* values could be accessed the same way by interpolating in the correct sub-interval. This may be useful for some problems where the independent variable t is not a physical time and functions may depend on the so-called 'future' variables. We may extend this feature in future versions.

Examples in `examples/harvest` and `examples/leukemia` provide an illustration of delay problems.

3.8 Experimental features

3.8.1 Parametrized control

Bocop provides built-in functions to implement a parametrized control. This can be useful, for instance if convergence is difficult due to oscillations in the control. In practice,

this is formulated as a path constraint on the control

$$u(t) - \hat{u}(t, \alpha)$$

where $\hat{u}$ defines the parametrized control.

i) Piecewise polynomial case: we define $\hat{u}$ as a piecewise polynomial function or order p over a uniform time subdivision $[\tau_0 = 0, \ldots, \tau_{N_u} = T]$.

ii) Trigonometric case: here we consider the whole time interval $[0, T]$ and define

$$\hat{u}(t) = \alpha_0 + \Sigma_{k=1}^{p}(\alpha_{2k-1}\ cos(kt) + \alpha_{2k}\ sin(kt))$$

The function *parametrizedcontrol* returns the value of $\hat{u}$.
It takes the following arguments:
- control_type: 1 for piecewise linear, 2 for trigonometric
- control_intervals: N_u for the polynomial case
- control_degree: p
- coefficients: the address of the first coefficient. The coefficients are part of the optimization variables π, so this is of the form `&optimvars[i]`. There are $(p+1)(N_u)$ coefficients in the polynomial case, and $(2p+1)$ in the trigonometric case. Do not forget to set the dimension of π accordingly.
- time, initial_time, final_time: t, t_0, t_f.

Exemple: methane bioreactor, the path constraint is in the *pathcond.tpp* file:

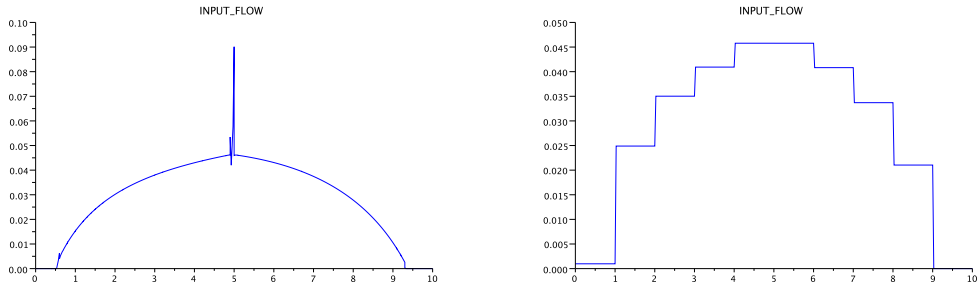
```
int control_type = constants[7];
int control_degree = constants[8];
int control_intervals = constants[9];

// Parametrized control (optimvars must contain the correct number of coefficients)
// piecewise polynomial case: (degree+1)*(intervals)
// trigonometric case: (1+2*degree)
path_constraints[0] = control[0] - parametrizedcontrol(control_type,
control_intervals, control_degree, &optimvars[0], time, initial_time, final_time);
```

If you want to impose continuity in the piecewise polynomial case, you can use the function *junctionconditions*, that will compute the $N_u - 1$ conditions as part of the boundary conditions. It takes the same arguments as *parametrizedcontrol*, minus the time t, and plus the address of the first junction condition in $\Phi(\ldots)$.

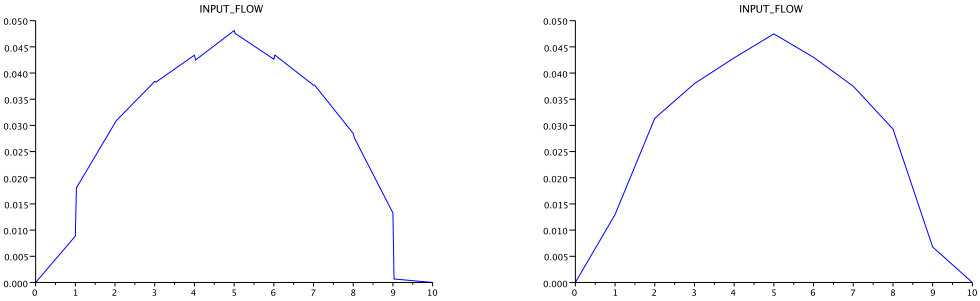
Exemple: methane bioreactor, the junction conditions are in *boundarycond.tpp*:

```
// INITIAL CONDITIONS FOR METHANE PROBLEM
// y(4)=0
boundary_conditions[0] = initial_state[3];

// PERIODICITY CONDITION FOR y(1:3)
boundary_conditions[1] = final_state[0] - initial_state[0];
boundary_conditions[2] = final_state[1] - initial_state[1];
boundary_conditions[3] = final_state[2] - initial_state[2];

// CONTINUITY FOR PIECEWISE LINEAR CONTROL (set dim_boundary_conditions accordingly)
// this will add (intervals-1) conditions
int control_type = constants[7];
int control_degree = constants[8];
int control_intervals = constants[9];
junctionconditions(control_type, control_intervals, control_degree, &optimvars[0],
initial_time, final_time, &boundary_conditions[4]);
```

Here we show the unconstrained, piecewise constant, piecewise linear, and continuous piecewise linear controls.



Methane bioreactor: unconstrained and piecewise constant controls



Methane bioreactor: piecewise linear and continuous piecewise linear controls

A Readme and Install files

A.1 Readme

```
BOCOP - TOOLBOX FOR OPTIMAL CONTROL
-----------------------------------

Bocop is an open-source toolbox for solving optimal control problems.
Bocop implements direct transcription method, BocopHJB a dynamic programming approach.
The project is supported by Inria-Saclay and is developed by the team Commands.
Authors: Frederic Bonnans, Jinyan Liu, and Pierre Martinon;
Daphne Giorgi, Vincent Grelard, Benjamin Heymann and Olivier Tissot.
Bocop is free for academic and industrial use, under the Eclipse Public License (EPL).
Website: http://bocop.org
```

A.2 Install (Linux)

```
**************************
BOCOP INSTALL NOTES - LINUX
**************************

In the following, <BOCOP> is the folder of the extracted package.
The full path to this folder must NOT contain blanks/spaces.

***************
A. PREREQUISITES

Bocop requires g++, gfortran and CMake.
Please install them if necessary (using yum, apt-get or the system tools)

****************************
B. HOW TO LAUNCH BOCOP (GUI)

The Bocop user interface exists in both 32 and 64 bit.
If unsure of your OS version, just try both, one at least should work ;-)
To launch the Bocop interface (Qt) on 64 bit Linux:
- double click on BocopGui64 in <BOCOP>/qtgui or
- execute the command "./BocopGui64" in a terminal at <BOCOP>/qtgui
For the 32 bit version just replace 64 by 32.

On first launch the GUI will open the default problem "goddard".
You can check your installation by running this problem (green triangle icon).
After some compilation steps, you should see the optimization begin.
Some dependencies have to be compiled the first time you build a problem.
This can take several minutes.

After optimization, the solution file can be visualized directly from
the Qt interface, or using the Matlab script (bocop.m), located in the
folder scripts/.

For more information on using bocop and on the examples of optimal control
problems solved by bocop, please refer to the UserGuide and the Examples
notes available on the Download page (bocop.org).

Note: the settings for the GUI should be saved in
```

```
$home/.config/Inria/Bocop-x.x.x.conf

***********************************
C. HOW TO LAUNCH BOCOP (COMMAND LINE)

To build the bocop executable without using the GUI, you can copy the script
build.sh in <BOCOP>/examples/default into your problem folder, and run it by:
  sh build.sh

The script will invoke cmake/make to build a binary 'bocop'. Run it with
  ./bocop

You can edit the script to change the build type from Debug (default) to Release.
```

A.3 Install (Windows)

```
*****************************
BOCOP INSTALL NOTES - WINDOWS
*****************************

In the following, <BOCOP> is the folder of the extracted package.
The full path to this folder must NOT contain blanks/spaces.
For instance C:\Bocop-x.x.x is fine, but NOT C:\Program Files\...

For win8 or win8.1 users: if Bocop is installed on disk C:, you may need
to run "BocopGui" as administrator to prevent access rights errors.
Alternately, install Bocop on another disk (not C:).


****************
A. PREREQUISITES

Bocop for windows requires MinGW and CMake, both included in the package.
For Cmake, during the install choose the option to add the CMake path
in the user PATH environment variable.

A.1 MINGW MANUAL INSTALL
A complete MinGW archive is included in the Bocop package for convenience.
This archive corresponds to a MinGW install with the following components
- msys developer toolkit (include msys base)
- g++
- gfortran
- in MinGW Base System, libpthreadgc-dev
Bocop install also adds <mingw>/msys/1.0/bin;<mingw>/bin to the PATH variable.


**********************
B. HOW TO LAUNCH BOCOP

To launch the Bocop user interface (Qt), double click the BocopGui link on Desktop.
The executable is located in folder <BOCOP>/qtgui.

On first launch the GUI will open the default problem "goddard".
You can check your installation by running this problem (green triangle icon).
After some compilation steps, you should see the optimization begin.
```

```
Please note that some antivirus software can block the compilation process.
In this case you should disable the antivirus, since Bocop requires this step.

After optimization, the solution file can be visualized directly from
the Qt interface, or using the Matlab script (bocop.m), located in the
folder scripts/.

For more information on using bocop and on the examples of optimal control
problems solved by bocop, please refer to the UserGuide and the Examples
notes available on the Download page (bocop.org).


***********************************
C. HOW TO LAUCH BOCOP (COMMAND LINE)

To build the bocop executable without using the GUI, you can copy the script build.sh
given in <BOCOP>/examples/default into your problem folder, and run it with
  sh build.sh

The script will invoke cmake and make to build a binary 'bocop.exe', run it with
  ./bocop

Alternately, you can do the steps manually in your problem folder
  mkdir build
  cd build
  cmake -G "MSYS Makefiles" -DPROBLEM_DIR=.. <BOCOP>
  make -j

Then go back to the problem folder and run
  cd ..
  ./bocop
```

A.4 Install (MacOS)

```
****************************
BOCOP INSTALL NOTES - MAC OS
****************************

In the following, <BOCOPHJB> is the folder of the extracted package.
The full path to this folder must NOT contain blanks/spaces.

****************
A. PREREQUISITES

Bocop requires C/C++ and Fortran compilers, as well as CMake.
You can install them with HomeBrew (see A.1) or manually (see A.2).

A.1 HOMEBREW (https://brew.sh)

First install homebrew with the following command in a terminal window:

   sudo /usr/bin/ruby -e "$(curl -fsSL https://raw.githubusercontent.com/
   Homebrew/install/master/install)"
```

Then install cmake and the compilers:

```
   brew install cmake
   brew reinstall gcc --without-multilib
```

Installation can take some time (about an hour on a modern mac).

A.2.1 XCODE

Download and install Xcode from the appstore. Accept the Xcode license.
NOTE: some security updates can reset this license agreement.
If you get errors such as 'cannot execute C programs' when trying to
build your problem, run the following command to re-accept the license:

```
  xcode-select --install
```

A.2.2 CMAKE

1) Get cmake from internet, put it in /Applications

2) Check that the file 'cmake', 'ccmake' are in the directory
/Applications/CMake.app/Contents/bin/

3) Open a terminal and create symbolic links to /usr/bin as follows:

```
  sudo ln -s /Applications/CMake.app/Contents/bin/ccmake /usr/bin/ccmake
  sudo ln -s /Applications/CMake.app/Contents/bin/cmake /usr/bin/cmake
```

4) Check the result by typing in terminal

```
  which cmake
  which ccmake
```

The answers should be

```
  /usr/bin/cmake
  /usr/bin/ccmake
```

A.2.3 GFORTRAN

1) uninstall the older gfortran if you have one (if not skip to step 2)
open a terminal and type:

```
  sudo rm -r /usr/local/gfortran /usr/local/bin/gfortran
```

enter your password
check the uninstallation by typing gfortran in the terminal:
it shouldn't be recognized (command not found)

2) download the compatible gfortran (gfortran-6.3-bin.tar.gz as of 2017/01)
https://sourceforge.net/projects/hpc/files/hpc/g77/

3) Install
open a terminal and type:

```
  cd ~/Downloads (or the folder you specified)
  gunzip gfortran-5.0-bin.tar.gz
  sudo tar -xvf gfortran-5.0-bin.tar -C/.
```

check the installation by typing gfortran in the terminal:
you shouldn't have 'command not found' but a message like 'no input file'

```
**********************
B. HOW TO LAUNCH BOCOP

To launch the Bocop interface (Qt) on 32 bits Linux:
- double click on BocopGui32 in <BOCOP>/qtgui or
- run "./BocopGui32" in a terminal at <BOCOP>/qtgui

To launch the Bocop interface (Qt) on 64 bits Linux:
- double click on BocopGui64 in <BOCOP>/qtgui or
- run "./BocopGui64" in a terminal at <BOCOP>/qtgui

On first launch the GUI will open the default problem "goddard".
You can check your installation by running this problem (green triangle icon).
After some compilation steps, you should see the optimization begin.
Some dependencies have to be compiled the first time you build a problem.
This can take several minutes.

After optimization, the solution file can be visualized directly from
the Qt interface, or using the Matlab script (bocop.m), located in the
folder scripts/.

For more information on using bocop and on the examples of optimal control
problems solved by bocop, please refer to the UserGuide and the Examples
notes available on the Download page (bocop.org).

Note: the settings for the GUI should be saved in
$home/.config/Inria/Bocop-x.x.x.conf

***********************************
C. HOW TO LAUCH BOCOP (COMMAND LINE)

To build the bocop executable without using the GUI, you can copy the script
build.sh from <BOCOP>/examples/default into your problem folder, and run it
  sh build.sh

The script will invoke cmake and make to build a binary 'bocop', run it with
  ./bocop

Alternately, you can do the steps manually in your problem folder
  mkdir build
  cd build
  cmake -DPROBLEM_DIR=.. <BOCOP>
  make -j

Then go back to the problem folder and run
  cd ..
  ./bocop
```

B Description of problem files

B.1 Definition file: Problem.def

This file defines the dimensions and names for the variables, as well as several general parameters. Note that the ordering of the lines in this file does not matter. Blank lines can be used for more clarity, as well as comments beginning by #. We recommend renaming every variable and constraint, however this is not mandatory.

- Initial and final time
  - `time.free`: flag for free times ("none" or "final")
  - `time.initial`: initial time t_0
  - `time.final`: final time t_f (1 for free t_f)

- Dimensions
  - `state.dimension`: dimension of state variables y
  - `control.dimension`: dimension of control variables u
  - `algebraic.dimension`: dimension of algebraic variables z
  - `parameter.dimension`: dimension of parameters π
  - `boundarycond.dimension`: dimension of initial/final conditions Φ
  - `constraint.dimension`: dimension of path constraints g
  - `constants.dimension`: number of numerical constants for (P)

- Discretization
  - `discretization.steps`: number of time steps N
  - `discretization.method`: method for discretization (cf 3.2.2 p. 11)

- Optimization
  - `optimization.type`: "single" or "batch" optimization
  - `batch.type`: type of the variable explored in batch optimization (can be a constant or the number of time steps)
  - `batch.index`: index of the constant explored in batch optimization (in $[0..dim-1]$)
  - `batch.nrange`: number of optimizations in the batch
  - `batch.lowerbound`: lower bound for the constant
  - `batch.upperbound`: upper bound for the constant
  - `batch.directory`: path for the results of the batch optimization

- Initialization
  - `initialization.type`: "from_init_file" as default, "from_sol_file_cold" for re-optimization (cold start), "from_sol_file_warm" for warm start
  - `initialization.file`: "no_file" as default, name of a solution file for re-optimization

- Parameter identification
  - `paramid.type`: flag for the method ("false", "LeastSquare", "LeastSquareWithCriterion", or "Manual")
  - `paramid.separator`: character used to separate the columns in the observation file
  - `paramid.file`: "no_file" as default, name of one file for parameter identification
  - `paramid.dimension`: data set size, number of observation files

- Names
  - `state.i`: name of component i of y, ($i \in [0..dim-1]$)
  - `control.i`: name of component i of u
  - `parameter.i`: name of component i of π
  - `boundarycond.i`: name of component i of Φ
  - `constraint.i`: name of component i of g
  - `algebraic.i`: name of component i of z

- Solution file
  - `solution.file`: name of the solution file (default "problem.sol")

- Iteration output frequency
  - `iteration.output.frequency`: frequency of the Ipopt iterations' output (0 means no output, k means output every k iterations)

B.2 Bounds file: Problem.bounds

The first line recalls the dimensions for Φ, y, u, z, π, g. The lower and upper bounds are in tri-column format. The first and second columns are for the lower and upper bounds. The third column indicates the constraint type. In Bocop 1.0.x this type was represented by an integer between 0 and 4, and since version 1.1.0 by a string for better readability. Bocop is able to read both of these representations, so there is no need to modify older files.

The bounds follow in blocks, for Φ, y, u, z, π, g respectively. Inactive bounds will be set as $\pm$ 2e20, and Bocop will check the consistency of the given bounds values and types.

Type of bound
- **free (or 0)**: no bounds (values are ignored)
- **lower (or 1)**: lower bound only (upper bound ignored)
- **upper (or 2)**: upper bound only (lower bound ignored)
- **both (or 3)**: two bounds (implies lower bound lower than upper bound)
- **equal (or 4)**: equality (implies lower bound equal to upper bound)

B.3 Functions for the optimal control problem

The user has to write four functions to define the problem: one for the criterion to optimize, one for the dynamics of the problem, one for the path conditions and a last one for the boundary conditions. Keep in mind than the bounds for the variables x, u, z, π, and constraints Φ and g are defined in the *problem.bounds* file.

Important note on variables type: for real variables (as opposed to integer), the type to use is either the classic C "double", or the template "Tdouble". Any value that can change during the optimization, such as state or control variables and optimization parameters, must be declared as a "Tdouble". Values that remain constant during optimization can use the normal "double" type. Note that any expression containing a Tdouble value is also of type Tdouble. The Tdouble types are used for the computation of derivatives by automatic differentiation (Adol-C).

Note: In Bocop 1.0.x the headers of these functions were written explicitely in the functions files. Since version 1.1.0 they are saved in separated files (header_boundarycond, header_criterion, header_dynamics, header_pathcond), and are included at the beginning of the functions files. If migrating from 1.0.x versions, please use the provided update script.

```
// function for the criterion to optimize: J
template<class Tdouble> void crit(const int dim_state, const Tdouble initial_time,
const Tdouble* initial_state, const Tdouble final_time, const Tdouble* final_state,
const int dim_optimvars, const Tdouble* optimvars, const int dim_constants,
const double* constants, Tdouble &criterion)
```

```
// function for the dynamics of the problem: f
template<class Tdouble> void dynamics(const Tdouble time, const double normalized_time,
const Tdouble initial_time, const Tdouble final_time, const int dim_state, const Tdouble* state,
const int dim_control, const Tdouble* control, const int dim_algebraicvars, const Tdouble* algebraicvars,
const int dim_optimvars, const Tdouble* optimvars, const int dim_constants, const double* constants,
Tdouble* state_dynamics)
```

```
// function for the equality and inequality constraints on the path: g
template<class Tdouble> void pathcond(const int dim_path_constraints, const Tdouble time,
const double normalized_time, const Tdouble initial_time, const Tdouble final_time,
const int dim_state, const Tdouble* state, const int dim_control, const Tdouble* control,
const int dim_algebraicvars, const Tdouble* algebraicvars, const int dim_optimvars,
const Tdouble* optimvars, const int dim_constants, const double* constants, Tdouble* path_constraints)
```

```
// function for the initial and final conditions: Phi
template<class Tdouble> void boundarycond(const int dim_boundary_conditions, const int dim_state,
const Tdouble initial_time, const Tdouble* initial_state, const Tdouble final_time,
const Tdouble* final_state, const int dim_optimvars, const Tdouble* optimvars, const int dim_constants,
const double* constants, Tdouble* boundary_conditions)
```

B.4 Public Tools

In these functions definition, the user can call pre-defined methods. Here is the complete list of these methods :

```
/**
 * This function writes the content of the file named filename in the vector v
 * assuming that filename contains one column of double values.
 * return : size of v.
 */
int readFileToVector(const string filename, vector<double> &v)
```

```
/**
 * This function writes the content of the nbCol column of the csv file named filename in the vector v
 * assuming that file is well formated.
 * The default separator is set to ';', if you use another separator you have to specify it.
 * return : size of v.
 */
int readCSVToVector(const string filename, vector<double> &v, const int nbCol, const char separator=';')
```

```
/**
 * This function writes the content of the colName column of the csv file named filename in the vector v
 * assuming that file is well formated.
 * The default separator is set to ';', if you use another separator you have to specify it.
 * return : size of v.
 */
int readCSVToVector(const string filename, vector<double> &v, const string colName, const char separator=';')
```

```
/**
 * This function calculates the linear interpolation of the vector data at time normalized_time
 * the result is stocked in value. It returns a flag that indicates if the time was out of bound.
 * It is assumed that the time discretization is uniform.
 * return : interpolated value.
 */
double normalizedTimeInterpolation(const double normalized_time, const vector<double>& data)
```

```
/**
 * This function returns the linear interpolation of the vector y_data at x_value renormalized by x_data.
 * This is a generalization of normalizedTimeInterpolation().
 * return : interpolated value.
 */
double interpolation(const double x_value, const vector<double>& x_data, const vector<double>& y_data)
```

B.5 Solution file: Problem.sol

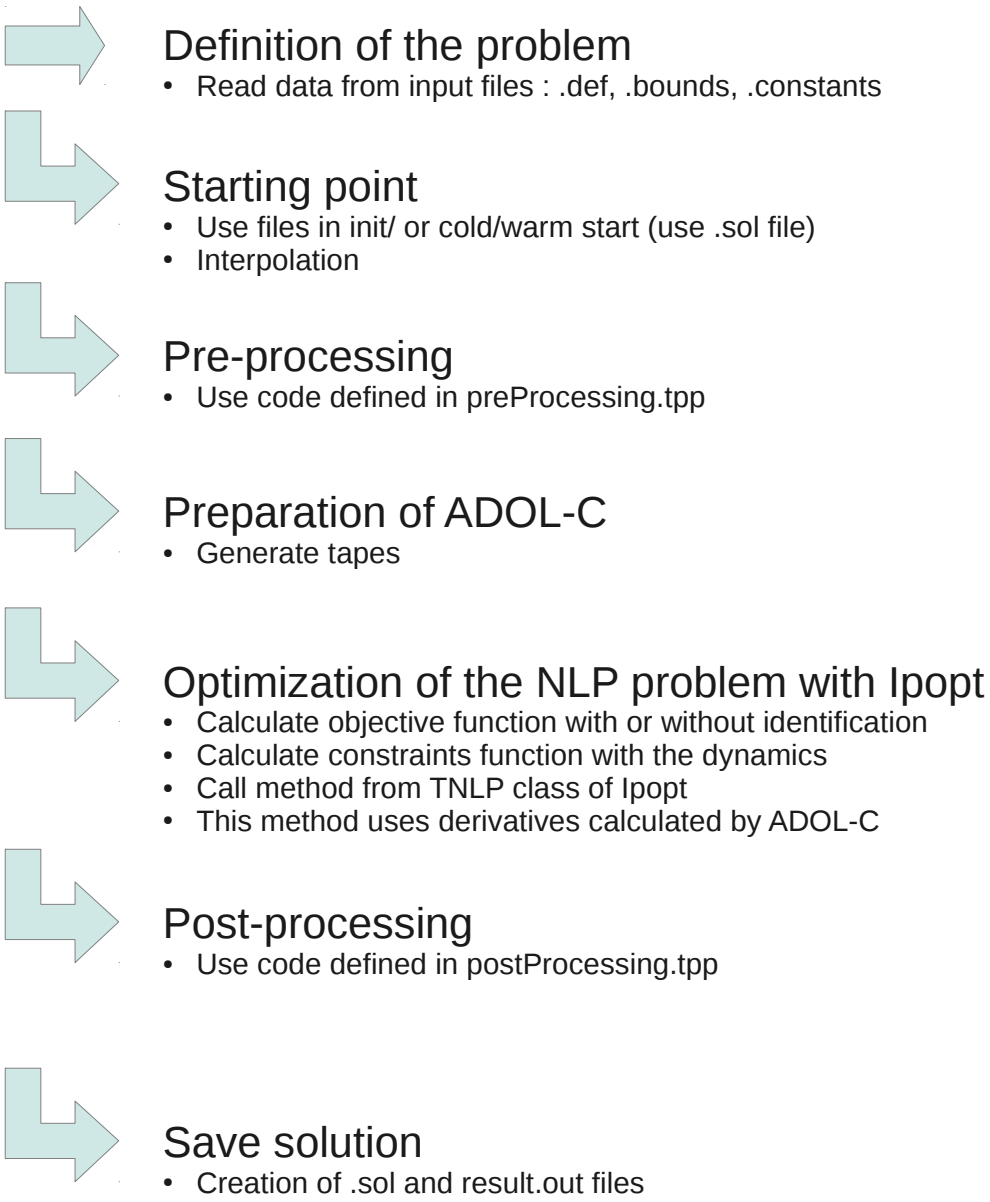
The solution file is generated at the end of the optimization. It is designed to be as self-sufficient as possible, so that you only have to backup the .sol files in your experiments. For instance, a standalone .sol file can be visualized in the GUI, even if you do not have the other files for the problem. If you want to read the solution file without using the GUI, you can use the script *read_solution_file.sce*. Also, a .sol file contains a copy of all the input files used to obtain this particular solution.

Each .sol file is organized as follows:

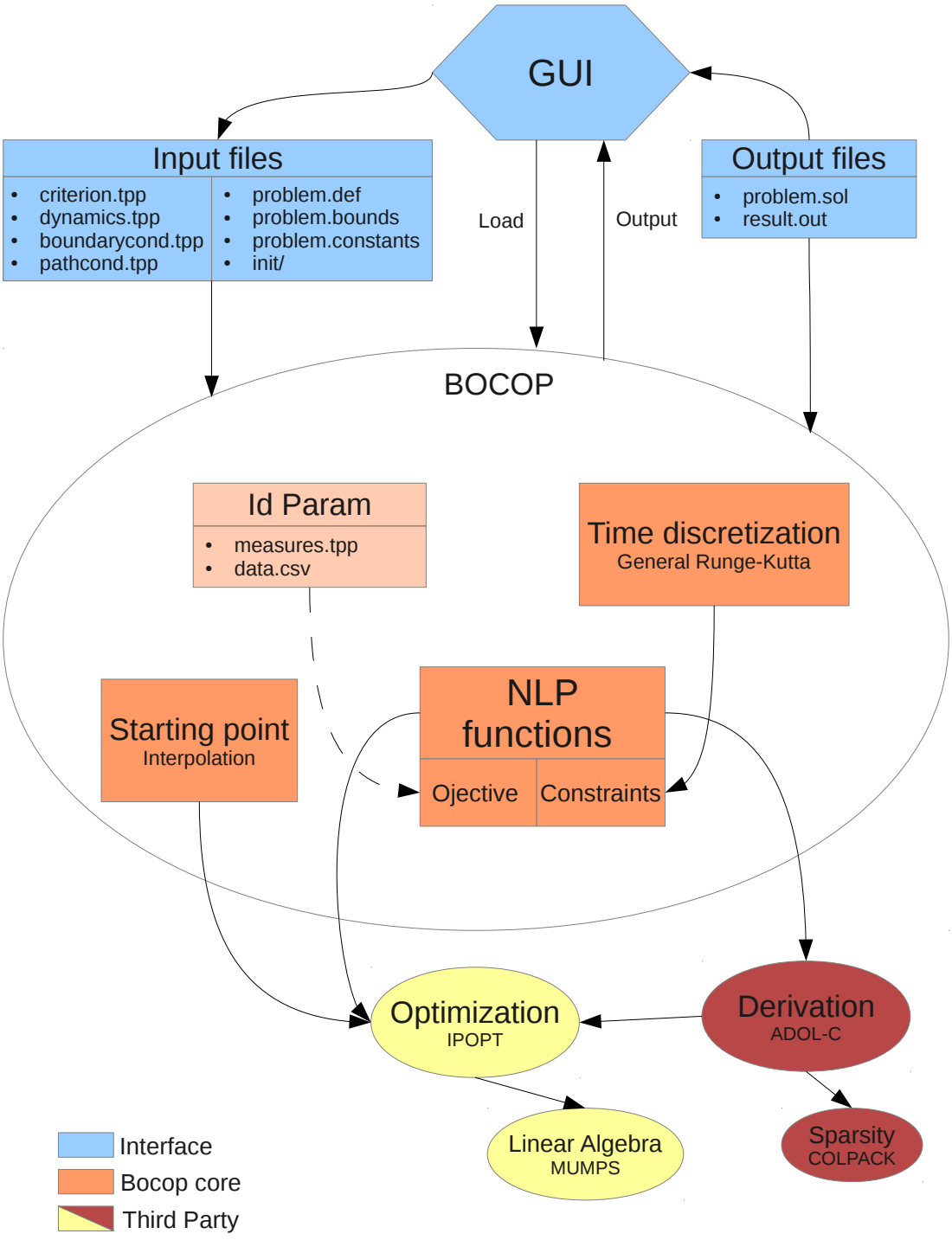
- Copy of the *.def* file
- Copy of the *.bounds* file
- Copy of the *.constants* file
- Copy of all the *.init* files

- Solution
  - objective value, L^2 and inf norm of the constraints
  - number of stages of the discretization method
  - time steps (including interior stages)
  - state variable 0, then 1, etc
  - control variable 0, then 1, etc
  - algebraic variable 0, then 1, etc
  - scalar optimization parameters (e.g. free final time)
  - boundary conditions (lower bound - value - upper bound - type)
  - path constraint 0, then 1, etc (1st line: lower bound - upper bound - type)
  - dynamic constraint 0, then 1, etc (1st line: lower bound - upper bound - type)
  - multipliers, sorted by the corresponding constraints
  - coefficients of the discretization method

C Bocop code structure



Bocop execution workflow



Bocop code organization

D Files for the Goddard problem

D.1 Definition files

These are the files *problem.def*, *problem.bounds* and *problem.constants*.

```
# This file defines all dimensions and parameters
# values for your problem :

# Initial and final time :
time.free string final
time.initial double 0
time.final double 1

# Dimensions :
state.dimension integer 3
control.dimension integer 1
algebraic.dimension integer 0
parameter.dimension integer 1
constant.dimension integer 6
boundarycond.dimension integer 4
constraint.dimension integer 1

# Discretization :
discretization.steps integer 100
discretization.method string gauss

# Optimization :
optimization.type string batch
batch.type integer 0
batch.index integer 5
batch.nrange integer 3
batch.lowerbound double 0.3
batch.upperbound double 0.9
batch.directory string batch-C

# Initialization :
initialization.type string from_init_file
initialization.file string none

# Parameter identification :
paramid.type string false
paramid.separator string ,
paramid.file string no_directory
paramid.dimension integer 0

# Names :
state.0 string position
state.1 string speed
state.2 string mass
control.0 string acceleration_u
parameter.0 string finaltime
boundarycond.0 string r(0)
boundarycond.1 string v(0)
boundarycond.2 string m(0)
boundarycond.3 string r(f)
constraint.0 string drag_minus_C
constant.0 string Tmax
constant.1 string A
constant.2 string k
constant.3 string r0
constant.4 string b
constant.5 string C

# Solution file :
solution.file string problem.sol

# Iteration output frequency :
iteration.output.frequency integer 0
```

```
# This file contains all the bounds of your problem.
# Bounds are stored in standard format :
# [lower bound]  [upper bound] [type of bound]

# Dimensions (i&f conditions, y, u, z, p, path constraints) :
4 3 1 0 1 1

# Bounds for the initial and final conditions :
1 1 equal
0 0 equal
1 1 equal
1.01 2e+20 lower

# Bounds for the state variables :
>0:1:0 0 2e+20 lower
>1:1:1 0 0 free
>2:1:2 0.1 2e+20 lower

# Bounds for the control variables :
0 1 both

# Bounds for the algebraic variables :

# Bounds for the optimization parameters :
0 2e+20 lower

# Bounds for the path constraints :
-2e+20 0 upper
```

```
# This file contains the values of the constants of your problem.
# Number of constants used in your problem :
6

# Values of the constants :
3.500000000000000e+00
3.100000000000000e+02
5.000000000000000e+02
1.000000000000000e+00
7.000000000000000e+00
5.000000000000000e-01
```

D.2 Source files

Here are the actual C files that compute the functions J, f, Φ, g for the goddard problem. Note that we use external functions named `drag,grav,thrust`.

```
{
// DYNAMICS FOR GODDARD PROBLEM
// dr/dt = v
// dv/dt = (Thrust(u) - Drag(r,v)) / m - grav(r)
// dm/dt = -b*|u|

  //constants
double Tmax = constants[0];
double A = constants[1];
double k = constants[2];
double r0 = constants[3];
double b = constants[4];

  //variables
Tdouble r = state[0];
Tdouble v = state[1];
Tdouble m = state[2];

  //dynamics
state_dynamics[0] = v;
state_dynamics[1] = (thrust(control[0],Tmax) - drag(r,v,A,k,r0)) / m - grav(r);
state_dynamics[2] = - b * control[0];
}
```

```
{
// CRITERION FOR GODDARD PROBLEM
// MAXIMIZE FINAL MASS
criterion = -final_state[2];
}
```

```
{
// INITIAL CONDITIONS FOR GODDARD PROBLEM
// r0 = 1    v0 = 0   m0 = 1
// MODELED AS 1 <= r0 <= 1, etc

boundary_conditions[0] = initial_state[0];
boundary_conditions[1] = initial_state[1];
boundary_conditions[2] = initial_state[2];

// FINAL CONDITIONS FOR GODDARD PROBLEM
// rf >= 1.01 MODELED AS   1.01 <= rf

boundary_conditions[3] = final_state[0];
}
```

```
{
// CONSTRAINT ON MAX DRAG FOR GODDARD PROBLEM
// Drag <= C ie Drag - C <= 0

double A = constants[1];
double k = constants[2];
double r0 = constants[3];
double C = constants[5];

Tdouble r = state[0];
Tdouble v = state[1];

path_constraints[0] = drag(r,v,A,k,r0) - C;
}
```

The external functions are defined in the files *drag.tpp*, *grav.tpp* and *thrust.tpp*, in our case located in the "model" subdirectory.

```
// FUNCTION FOR GODDARD DRAG
// drag = A v^2 exp (-k(r-r0))

template<class Tdouble> Tdouble drag(const Tdouble r, const Tdouble v,
const double A, const double k, const double r0)
{
Tdouble drag;
drag = A * v * v * exp(-k*(fabs(r)-r0));
return drag;
}
```

```
// FUNCTION FOR GRAVITY
// g = 1 / r^2

template<class Tdouble> Tdouble grav(const Tdouble r)
{
Tdouble grav;
grav = 1e0 / r / r;
return grav;
}
```

```
// FUNCTION FOR THRUST (GODDARD)
// T = u * Tmax

template<class Tdouble> Tdouble thrust(const Tdouble u, const double Tmax)
{
Tdouble thrust;
thrust = u * Tmax;
return thrust;
}
```

Finally, here are two files *dependencies.hpp* and *dependencies.tpp*, which contain the declarations and location of the external functions.

```
// function for the goddard drag
template<class Tdouble> Tdouble drag(const Tdouble r,const Tdouble v,
const double A, const double k, const double r0);

//function for gravity
template<class Tdouble> Tdouble grav(const Tdouble r);

//function for goddard thrust
template<class Tdouble> Tdouble thrust(const Tdouble u, const double Tmax);
```

dependencies.tpp

```
#include "./model/grav.tpp"
#include "./model/drag.tpp"
#include "./model/thrust.tpp"
```

E Third party software information

Current dependencies for Bocop are:
- GUI: Qt, visualization Matlab and Python
- NLP solver: Ipopt (EPL), uses MUMPS (public domain)
- Derivatives: Adol-C (GPL / CPL), uses ColPack (LGPL). Alternative: CppAD (EPL)

• Qt
The interface for Bocop uses the Qt framework. See qt-project.org.

• Ipopt
Ipopt[11] is used to solve the NLP (Non Linear Programming) problem resulting from the discretization of the optimal control problem, see https://projects.coin-or.org/Ipopt. Ipopt uses the standard BLAS and LAPACK librairies and also requires an external linear solver. We currently use MUMPS[1, 2], see mumps.enseeiht.fr/.

• Adol-C
Adol-C[12] is used to compute the derivatives of the objective and constraints by automatic differentiation, see https://projects.coin-or.org/ADOL-C. Sparsity patterns are computed using the Colpack[5] package, see www.cscapes.org/download/ColPack/.

• CppAD
Since 2.1.0 Bocop can also use CppAD[3] instead of Adol-C/Colpack to compute the derivatives. See https://www.coin-or.org/CppAD/.

F Changelog

```
BOCOP CHANGELOG

2017/10/17: bocop-2.1.0

  - dynamics function now has access to past state and control variables,
  in order to solve delay problems; pathcond has access to past states.
  Restricted to fixed final time case. New examples: harvest, leukemia.
  - problem dimensions are now passed to preProcessing function.
  - functions for Heaviside and 2Dinterpolation added to publicTools
  - [gui] visualization now show bounds for state variables (optional).
  - [gui] bugfix for GUI unable to locate obervations files in parameter
  identification problems. See jackson example.
  - [gui] bugfix for GUI sometimes resetting problem discretization to
  default (Midpoint with 100 time steps).
  - [core] ADOL-C options for sparse hessian computation have been adjusted
  and should give better performance in most cases.
  - [core] time discretizations are now hardcoded (BocopDefinition.cpp)
  instead of reading .disc files at runtime. Bocop executables therefore
  do not need access to Bocop root package anymore at runtime.
  Available methods: Euler (explicit), Euler (implicit), Midpoint (imp.),
  Gauss II and Lobatto IIIC.
  - [core] L2 norm of constraint violation is now correctly saved in .sol file.
  - [core] some .tpp files were merged with their repective .hpp files.
  - [core] a few minor memory leaks were fixed.
  - [thirdparty, linux, mac] update to latest ADOL-C 2.6.3 and COLPACK.
  Should give reduced memory usage as well as faster computation of derivatives.
  - [thirdparty] build process should be faster, quieter and more foolproof.
  - [thirdparty] Experimental: support for CppAD instead of AdolC/Colpack to
  compute derivatives. Enabled with CMake option USE_CPPAD.
  - [scripts] benchmark scripts were updated.
  - [windows] Updated Cmake and MinGW to latest versions (now c++11 compliant).

2017/02/06: bocop-2.0.5

  - package: improved windows installer will automatically offer to install Cmake and MinGW.
  It should also not require PATH edits or reboots anymore (the latter may depend on windows
  version).
  - package: link to GUI added to desktop.
  - package: root directory for Bocop package should now be properly set up after install.
  - IO: float precision for .sol file set to 15 instead of 20 to avoid spurious digits.
  - GUI: added red lines for bounds for state, control and algebraic variables in visualization tab.
  - GUI: 'Clean problem' in Build&Run menu now checks for file existence, thus preventing
  warnings for trying to remove missing files.
  - GUI: 'Save .sol as' in visualization tab should now overwrite properly if confirmed.
  - GUI: reverted bounds format (in .bounds file) for state variables from block (experimental)
  to normal.
  - GUI: fixed some mis-positionning between bounds and labels in .bounds file (this bug was
  cosmetic only and had no impact).
  - GUI: in starting point tab, Ymin and Ymax should now be updated according to the figure
  (linear or spline interpolations).
  - GUI: constants are now numbered from 0 to n-1 for consistency with c++ code.
  - GUI: tab 'Discretization' was merged into 'Definition' (method and time steps).
  - GUI: prompt for clean and rebuild if Cmake fails to build the executable.
```

```
  - GUI: prompt for build if executable not present when launching the optimization.
  - GUI: splitted some functions for better code clarity
  - GUI: fixed bug in export function for algebraic variables (time stages).
  - example/contrast added: contrast optimization in magnetic resonance imaging.
  - example/swimmer added: optimal swimming for 3-link Purcell type micro-swimmer.
  - example/jackson: reorganization of the different versions of the parameter identification.
  - example/methane: renamed to bioreactor; fixed improper use of Tdouble variables
  (initial_time, final_time) in call to function parametrizedcontrol (pathcond.tpp).
  - example/stateconstraint3: removed some bounds that could make the problem unfeasible.
  - core: option for iteration output was disabled for now for debugging.


2015/11/13: bocop-2.0.4
  - GUI: removed the ability to edit the C++ function of the problem in Definition tab.
  The .tpp files can be handled by any text editor (GUIcreator, gedit, notepad++, ...)
  This feature had caused some pretty annoying bugs (e.g. recently mixing files
  criterion.tpp and dynamics.tpp) and was getting increasingly complicated to maintain
  properly. Be careful not to edit .def / .bounds / .constants files simultaneously from
  the GUI and an external editor. This will likely mess up with your problem definition.
  - GUI: fixed the broken button to stop an ongoing optimization.
  - GUI: fixed some extra newlines when displaying output from compilation and optimization.
  - GUI: problem is no longer cleaned systematically when building, to speedup the build process.
  - GUI: added a Clean menu entry that deletes the bocop executable and build/ subfolder.
  - GUI: version number is now checked by reading the file VERSION.
  Running mismatched versions is not recommended.
  - GUI: removed 'Save Problem as' tab in File menu (use manual copy instead).
  - GUI/core: default starting point values set to 0.1 instead of 0 to reduce the risk of
  math errors (log, division by zero, etc).
  - core: CMakeLists edited so that default build now uses the same flags as Debug build
  - matlab: added a new launcher function that allows to call bocop from matlab.
  The launcher can pass several input parameters and automatically parses the solution.
  - matlab: window for visualization tree was shifted to avoid overlap with main window.


2015/09/25: bocop-2.0.3
  - GUI: add new option iteration output frequency to save the iterations from Ipopt
  - GUI: the run option rebuilds the executable if the source files have been modified
  or if there is no executable in the problem directory
  - GUI: remove jpeg format for export
  - GUI: add explicit error message when cmake is not installed
  - GUI: add new option to compile in debug mode (default compilation is release mode)
  - core: one build directory per problem instead of one global build directory
  - core: slight modification in problem.bounds syntax to prepare block definition
  - core bugfix: correct values of control initialization saved in sol file
  - bugfix: change Ipopt configure to fix an error that happened on some Linux
  (the error was "don't have function for random number generator")
  - Third parties: update to latest working versions
  - Third parties: we distribute a pre-built version of Ipopt for Windows because of
  some error that occur when a recent version of Matlab is installed

2015/03/25: bocop-2.0.2
  - GUI: initialization can import data from an external file in Starting Point tab
  (any previous .init file is saved as .init.backup)
  - automatic backup of .sol and .def files (as .backup)
  - core: add pre-defined functions in publicTools which can be called directly
```

```
  in user files (such as reading a data file or interpolations over time).
    Description of these functions added in User Guide (appendix) and doxygen.
  - core: add header_* for pre/postProcessing.cpp and measures.tpp
  - GUI: better description for cold/warm start options of optimization restart
  - GUI: more detailled messages at the end of the optimization
  - GUI: button "Save .sol as" in Visualization tab to rename .sol file directly
  - GUI: option "Save problem as" in File menu to copy current problem folder
  - add pre/post-processing files in default example
  - Doxygen: minor changes in BocopDefinition and BocopProblem

2014/09/16: bocop-2.0.1
  - core bugfix: batch folder created when calling a batch optimization
    from command line
  - core bugfix: parameter identification allows the final time to be equal
    to the final time of observation files
  - float precision set to 20 in the solution file
  - GUI bugfix: batch conversion between number of optimizations and batch step
  - GUI : keep memory of the batch informations after doing a single optimization

2014/07/20: bocop-2.0.0
  - cmake: reintroduce cmake flag for unused variables
  - GUI bugfix: choose trunk when README doesn't exist or is broken
  - package: tar.gz files for linux package instead of sh
  - Third parties: update to latest working versions and build via cmake

2014/05/05: bocop-1.1.6
  - GUI: added a clean step before each build for more robustness
  - core bugfix: bad reading of observation files on some systems
    (ex: Mac, Ubuntu 12.04) corrected
  - cmake: set the compilers to cc and c++ for Mac
  - GUI bugfix: visualization and export adapted also to the parameter
    identification case

2014/04/08: bocop-1.1.5
  - extended parameter identification: 'manual' mode, handling of several
    observation files
  - optional pre and post-processing functions specific to each problem
  - batch mode is now available also on the discretization steps
  - GUI: batch mode now allows to set number of optimizations or value step
  - GUI: gui style was changed for compatibility with ubuntu 12.04
  - GUI: build folder created if not present; build made in this folder
  - GUI: update to GUI5 for compatibility with Maverick
  - GUI: enabled keyboard navigation on the visualization tree view
  - *.tpp files added to the sources list and header_* removed; this way
    the compiler is able to detect changes in the user's functions
  - change in the templates inclusions in order to detect changes
    in the optional functions (measure.tpp, pre/postprocessing.cpp)
  - default values are now provided for ALL missing values in problem.def
  - experimental: block initialization for bounds available on command line
    (currently not compatible with GUI usage)
  - NSIS installer for Windows package
  - minor corrections in compilation flags and symbolic links
  - added cpu time and iterations in solution file
  - reintroduced an example of parametrized control in methane problem
  - Matlab: update of the visualization script
```

```
2013/11/15: bocop-1.1.4
  - building process now uses CMake for better multi-platform support
  - GUI: new help menu with bocop informations
  - GUI: added safety check to detect version number and core location
  - GUI: several fixes to allow partial definition of a problem

2013/07/24: bocop-1.1.3
  - basic functionalities for parameter identification (least square)
  - removed Scilab interface, GUI gui is now the main interface
  - visualization scripts for Scilab and Matlab (experimental)
  - several interface improvements (definition, starting point, visualization)
  - folder "problems" renamed to "examples", some reorganization for bocop
    core files as well
  - makefile is no longer modified when changing problems; this allows
    running several instances of bocop to solve different problems at the
    same time (note that solving two instances of the same problem in parallel
    is still not supported at the moment)
  - added safety check to detect uninitialized values in user functions
  - GUI: export now saves average_control in addition to stage controls
  - GUI: added a clean step before each build for more robustness

2013/03/30: bocop-1.1.2
  - precompiled package with GUI Gui for Mac
  - Scilab gui: variables export, average control displayed by default,
    changed variables names for easier use in custom scripts
  - GUI gui: global multiple graphs for variables, constraints, multipliers

2013/03/14: bocop-1.1.1
  - precompiled package with GUI Gui for windows
  - GUI gui: multiple graphs on double click, spline interpolation display,
    variables export, average control displayed by default
  - "Solved at acceptable level" is now counted as a successful optimization
  - bugfix: a few glitches in GUI gui

2013/02/15: bocop-1.1.0
  - new GUI GUI available for linux precompiled packages.
  - multipliers for the constraints, as well as the adjoint states,
    are now saved in .sol files and available for visualization.
    Reoptimization is now possible as a warm start, reusing the
    multipliers from the previous solution for initialization in
    addition to the primal variables. Reoptimization using only the
    primal variables is now labeled as cold start.
  - indexes for variables, constraints, parameters and constants
    was shifted from the range 1..dim to 0..dim-1 for consistency
    with the C code of the functions. Same for initialization files.
    A shell script is provided to update problems from 1.0.x versions.
  - decreasing bounds are now allowed for batch optimization.
  - default discretization is now Lobatto III-C.
  - default Ipopt tolerance is now 10e-10.
  - bugfix: removed erroneous blank lines in .def and .bounds files.

2012/05/20: bocop-1.0.3
  - Packages with precompiled binaries for thirdparty codes.
  - Parametrized control, trigonometric or piecewise polynomial.
```

```
  - Time discretization is no longer uniform, it can be refined
    manually (experimental feature).
  - Batch optimizations for a varying parameter.
  - Separate initialization file for each variable.
  - Inclusion of "disc_problem.hpp" in criterion.tpp is no
    longer necessary. Please edit files created with older versions.
  - Headers for the functions (boundarycond.tpp, criterion.tpp,
    dynamics.tpp,pathcond.tpp) have changed. Please edit files created
    with older versions, using /problems/default files headers as a model.
  - Output of control and algebraic variables is now done at the stages
    times of the discretization method. State variables output still uses
    the usual discretization times.

2012/02/07: bocop-1.0.2
  - Reoptimization from an existing solution
  - Installation procedure available for Linux, Mac OS, and Windows
  - Memory leak corrected in ADOL-C/sparse/sparsedrivers.cpp

2012/01/15: bocop-1.0.1
  - Visualization in GUI made easier to use (tree of variables to plot)

2011/10/11: bocop-1.0.0-beta
  - First version of BOCOP!
```

References

[1] P. R. Amestoy, I. S. Duff, J. Koster, and J.-Y. L'Excellent. A fully asynchronous multifrontal solver using distributed dynamic scheduling. *SIAM Journal of Matrix Analysis and Applications*, 23(1):15–41, 2001.

[2] P. R. Amestoy, A. Guermouche, J.-Y. L'Excellent, and S. Pralet. Hybrid scheduling for the parallel solution of linear systems. *Parallel Computing*, 32(2):136–156, 2006.

[3] B.M. Bell and J.V. Burke. Algorithmic differentiation of implicit functions and optimal values. In Christian H. Bischof, H. Martin Bücker, Paul D. Hovland, Uwe Naumann, and J. Utke, editors, *Advances in Automatic Differentiation*, pages 67–77. Springer, 2008.

[4] J.T. Betts. *Practical methods for optimal control using nonlinear programming*. Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, 2001.

[5] A. Gebremedhin, A. Pothen, and A. Walther. Exploiting sparsity in jacobian computation via coloring and automatic differentiation: a case study in a simulated moving bed process. In C. Bischof et al, editor, *Lecture Notes in Computational Science and Engineering 64*, pages 339–349. Springer, 2008. Proceedings of the Fifth International Conference on Automatic Differentiation (AD2008).

[6] R.H. Goddard. *A Method of Reaching Extreme Altitudes*, volume 71(2) of *Smithsonian Miscellaneous Collections*. Smithsonian institution, City of Washington, 1919.

[7] 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.

[8] E. Hairer and G. Wanner. *Solving ordinary differential equations. II*, volume 14 of *Springer Series in Computational Mathematics*. Springer-Verlag, Berlin, 1996.

[9] J. Nocedal and S.J. Wright. *Numerical optimization*. Springer-Verlag, New York, 1999.

[10] H. Seywald and E.M. Cliff. Goddard problem in presence of a dynamic pressure limit. *Journal of Guidance, Control, and Dynamics*, 16(4):776–781, 1993.

[11] A. Wächter and L.T. Biegler. On the implementation of a primal-dual interior point filter line search algorithm for large-scale nonlinear programming. *Mathematical Programming*, 106(1):25–57, 2006.

[12] A. Walther and A. Griewank. Getting started with adol-c. In U. Naumann and O. Schenk, editors, *Combinatorial Scientific Computing*. Chapman-Hall CRC Computational Science, 2012.