BocopHJB 1.0.1 – User Guide

Frédéric Bonnans* Daphné Giorgi† Benjamin Heymann*
Pierre Martinon* Olivier Tissot*

March 11, 2016

*Inria Saclay and CMAP Ecole Polytechnique
†LPMA, Sorbonne University, Paris

Contents

1 BocopHJB overview **3**
- 1.1 Key features . . . 3
- 1.2 Algorithm . . . 3
- 1.3 Workflow . . . 3

2 Example: the mouse & maze problem **5**
- 2.1 Problem description . . . 5
- 2.2 Files for the mouse & maze problem . . . 6
  - 2.2.1 Definition files . . . 6
  - 2.2.2 Source files . . . 6

3 Algorithm description **9**
- 3.1 Stochastic optimal control problem . . . 9
- 3.2 Dynamic Programming Principle . . . 9
- 3.3 Semi Lagrangian scheme . . . 10
  - 3.3.1 Time discretization . . . 10
  - 3.3.2 Space discretization . . . 11
  - 3.3.3 Control discretization . . . 11
  - 3.3.4 Simulation . . . 12

4 Description of problem files **13**
- 4.1 Definition file: problemHJB.def . . . 13
- 4.2 State discretization file: folder stateDisc/ . . . 15
- 4.3 Control discretization file: folder controlDisc/ . . . 15
- 4.4 Basic Functions for the optimal control problem . . . 15
- 4.5 More advanced features . . . 17
  - 4.5.1 State and/or control constraints . . . 17
  - 4.5.2 Non uniform control discretization . . . 17
  - 4.5.3 Out of grid evaluation . . . 18
  - 4.5.4 Switching modes . . . 18
  - 4.5.5 Brownian realization for the simulation . . . 18

A Install notes (INSTALL file) **20**

B Code structure **25**

1 BocopHJB overview

1.1 Key features

- Global optimization for both deterministic and stochastic optimal control problems.
- Handles switching between discrete modes of the system.
- Stopping time problems can be solved using switchings.
- Built-in simulation module to recompute optimal strategies.
- Supports advanced rules to define the discrete control set.
- Parallel execution with OpenMP.
- Matlab / Python scripts to read value function and simulated trajectories.

1.2 Algorithm

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.

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.

1.3 Workflow

BocopHJB package contains core files and problem files. Core files implements the HJB solver and are problem independent. Each problem is defined by a set of c/c++ files and text files located in the problem folder. Solving an optimal control problem with BocopHJB involves the following steps:

1. Problem Definition
   Define the optimal control problem by completing the problem files. This files typically define the dimension, functions, and discretization (time, state and control) of the problem.

2. Build and Run
   The build step will create the `bocophjp` executable. Running the executable will, depending on the options set in `problemHJB.def`, compute the value function and/or simulate an optimal trajectory.

3. Visualization
   You can use provided python scripts in order to load and visualize the results of the solution and simulation files. Note that plotting the value function is not always available since it is a function of n variables, where n is the state dimension.

BocopHJB package includes a folder `examples/` with several sample problems to illustrate the features of the toolbox. These examples are described in more details in the document 'A collection of examples'

2 Example: the mouse & maze problem

2.1 Problem description

To test the use of both several switching possibilities and controls, we designed the following maze problem. A mouse trapped in a maze tries to get out. This mouse has a "bomberman typed" control space. The state can be described by the variable $(x, y) \in \mathbb{R}^2$ describing the position of the unlucky punctual mouse. The mouse has 4 modes modeling its direction: north, east, west, south. In addition to the direction modes, the mouse has a control variable for its velocity, which is positive and upper-bounded. We consider a running cost of 10 per unit of time in the maze, and each change of direction costs 1 as a switching cost. The mouse starts at the red square while the exit of the maze is at the green square. The optimal trajectory is shown on Fig. 1.

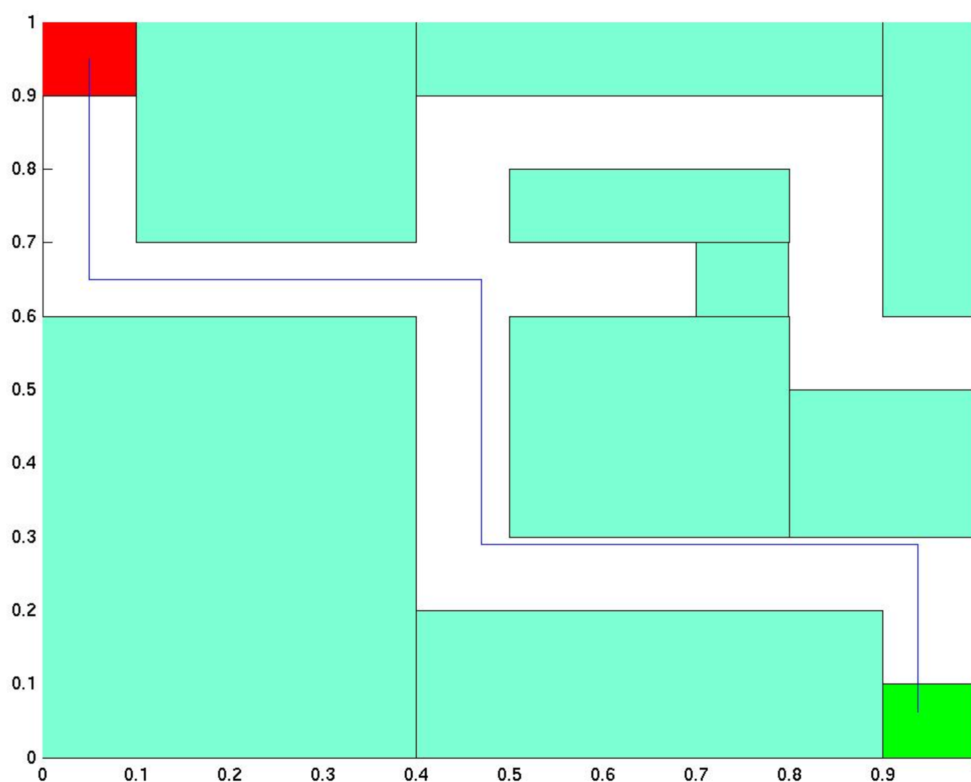


Figure 1: The Maze and the mouse trajectory according to BocopHJB

You can run this test and display the results with the following commands in terminal. Locally from the problem folder (`examples/maze/`):

```
> ./build
> ./bocophjb
```

Or from the root of the package:

```
> sh bocop build examples/maze
> sh bocop run examples/maze
```

2.2 Files for the mouse & maze problem

2.2.1 Definition files

`problemHJB.def`, `stateDisc/state.grid`, `controlDisc/control.grid`, `controlDisc/control.combination`.

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

# Initial and final time :
time.initial double 0
time.final double 3

# Dimensions :
state.dimension integer 2
control.dimension integer 1
constant.dimension integer 0
brownian.dimension integer 0

# Control :
discretization.control.type string uniform
combination.control.type string uniform

# Time discretization :
discretization.time integer 50

# Grid type :
grid.type string uniform

# Interpolation :
# Inner : linear ; other
# Outer : final value ; projection ; user function
interpolation.inner string linear
interpolation.outer string user_function

# Switching mode :
switching.mode integer 4

# Names :
state.0 string x1
state.1 string x2
control.0 string u

# Simulation :
simulation.type string from_computed_sol
simulation.noise string none

solution.file string valueFunction.sol
```

```
# Discretization of the state :
discretization.state.0 integer 30
discretization.state.1 integer 30

# Minimum of the state grid :
minimum.state.0 double 0
minimum.state.1 double 0

# Maximum of the state grid :
maximum.state.0 double 1
maximum.state.1 double 1
```

```
# Discretization :
discretization.control.0 integer 11

# Minimum of the control grid :
minimum.control.0 double 0

# Maximum of the control grid :
maximum.control.0 double 1
```

2.2.2 Source files

`dynamicsHJB.cpp`

```
/**
  * Drift function which describes the deterministic part of the dynamics.
  */
#include "header_drift"
{
    double u1 = control[0];

    switch(mode)
    {
    case 0 : // UP
        state_dynamics[0] = 0.0;
        state_dynamics[1] = u1;
        break;
    case 1 : // DOWN
        state_dynamics[0] =  0.0;
        state_dynamics[1] = -u1;
        break;
    case 2 : // LEFT
        state_dynamics[0] = -u1;
        state_dynamics[1] =  0.0;
        break;
    case 3 : // RIGHT
        state_dynamics[0] = u1;
        state_dynamics[1] = 0.0;
        break;
    }

}

/**
  * Volatility function which describes the stochastic part of the dynamics.
  */
#include "header_volatility"
```

```
{
// This function is unused since the problem is deterministic.
```

costFunctions.cpp

```
/**
  * Running cost for the computation of the criterion.
  */
#include "header_runningCost"
{
    double x1 = state[0];
    double x2 = state[1];

    if ( (x1 > 0.9) && (x2 < 0.1) )
        running_cost = 0;
    else
        running_cost = 10;
}

/**
  * Final cost for the computation of the criterion.
  */
#include "header_finalCost"
{
    final_cost = 0;
}

/**
  * Switching cost for the computation of the criterion.
  */
#include "header_switchingCost"
{

    if (current_mode == next_mode)
        switching_cost = 0;
    else
        switching_cost = 1;
}
```

constraints.cpp

```
/**
  *  User function used to check if a state is admissible or not.
  */
#include "header_checkAdmissibleState"
{

// We use state contraints to describe  the maze (position of walls).
    double x = state[0];
    double y = state[1];

    if( (x>1) ||  (x<0) || (y<0) || (y>1) ){return false;}

    if( (x<0.4) && (y<0.6) ){return false;}
    if( (x>=0.1) && (x<0.4) && (y>=0.7) ){return false;}
    if( (x>=0.4) && (x<0.9) && (y<0.2) ){return false;}

    if( (x>=0.4) && (x<0.9) && (y>=0.9) ){return false;}
    if( (x>=0.9) && (y>=0.6) ){return false;}


    if( (x>=0.5) && (x<0.8) && (y>=0.7) && (y<0.8) ){return false;}
    if( (x>=0.5) && (x<0.8) && (y>=0.3) && (y<0.6) ){return false;}
    if( (x>=0.7) && (x<0.8) && (y>=0.6) && (y<0.7) ){return false;}

    if( (x>=0.8) && (y>=0.3) && (y<0.5) ){return false;}

    return true;
}

/**
  *  User function used to check if a combination of controls and a state is admissible or not.
  */
#include "header_checkAdmissibleControlState"
{
    return true;
}
```

simulation.cpp

```
/**
  *\fn void simulationStartingPoint(std::vector<double>& starting_point)
  * User function to define the starting point of the simulation.
  */
#include "header_simulationStartingPoint"
{
  starting_point[0] = 0.05;
  starting_point[1] = 0.95;
```

```
}

/**
  *\fn void simulationStartingMode(int& starting_mode)
  * User function to define the starting mode of the simulation.
  */
#include "header_simulationStartingMode"
{
  starting_mode = 0;
}
```

optionalFunctions.cpp

```
/**
  * User function to compute the value of the value function for the points outside the grid.
  */
#include "header_userOutOfGridValueFunction"
{
// we return a huge value to prevent exit from the grid
    result = 10000;
}

/**
  *  User function used to define the discretized controls.
  */
#include "header_userControlDiscretization"
{
//unused function for this example (see control.discretization in problemHJB.def)
    return 0;
}

/**
  *  User function used to compute the combinations of controls.
  *  Each line of the resulting matrix is a combination of controls (u_0,..., u_p).
  */
#include "header_userControlCombination"
{
//unused function for this example (see control.discretization in problemHJB.def)

    return vector< vector<double> >();
}

/**
  *  User function used to compute the combinations of controls when it depends of state.
  *  Each line of the resulting matrix is a combination of controls (u_0,..., u_p).
  */
#include "header_userControlCombinationStateDependent"
{
//unused function for this example (see control.discretization in problemHJB.def)

    return vector< vector<double> >();
}
```

3 Algorithm description

3.1 Stochastic optimal control problem

Let y_t be a stochastic process described by

$$\begin{cases} dy_t = f(t, u_t, y_t)dt + \sigma(t, u_t, y_t)dW_t \\ y_0 = x \end{cases} \tag{1}$$

where the *control* $u_t \in U$ and $t \in [0, \infty[$, W_t is a standard Brownian motion and the *drift* f and the *volatility* σ are Lipschitz and bounded.

We define $\mathcal{U}$ the set of mappings with value in U adapted to the filtration generated by the Brownian motion (which means that we can take $u(t)$ as a function of the past history of the Brownian). We want to solve the stochastic optimal control problem

$$\min_{u \in \mathcal{U}} \mathbb{E} \left(\int_{t_0}^{T} \ell(t, u_s, y_s)ds + \phi(y_T) \right) \tag{2}$$

where ℓ is the *running cost* and ϕ the *final cost.*

Remark: Our framework includes additional state and control constraints of the form $g(t, u(t), y(t)) \leq 0$. *It also handles switchings between several modes, which allows in particular to solve stopping time problems, on/off state of plants, etc.*

3.2 Dynamic Programming Principle

We define the value function $V(x, t)$ such that

$$V(x, t) := \min_{u \in \mathcal{U}} \mathbb{E} \left(\int_{t}^{T} \ell(t, u_s, y_s)ds + \phi(y_T) \,\middle|\, y_t = x \right)$$

and

$$V(x, T) = \phi(x)$$

Let us take $\tau \in (t_0, T)$. We can write

$$V(y_0, t_0) = \min_{u \in \mathcal{U}} \mathbb{E}_{t_0} \left(\int_{t_0}^{\tau} \ell(t, u_s, y_s)ds + \int_{\tau}^{T} \ell(t, u_s, y_s)ds + \phi(y_T) \right)$$

which leads to the dynamic programming equation

$$V(y_0, t_0) = \min_{u \in \mathcal{U}} \mathbb{E}_{t_0} \left(\int_{t_0}^{\tau} \ell(t, u_s, y_s)ds + V(y_\tau, \tau) \right) \tag{3}$$

We can discretize on time the stochastic process (using for instance an Euler scheme), so that we have y^{k+1} as a function of y^k, σ^k, u^k. Let $t_k = h_0 k$ with $t_N = T$. The discretized problem is

$$\min_{u_k \in \mathcal{U}} \mathbb{E} \left(h_0 \sum_{k=0}^{N-1} \ell(t_k, u^k, y^k) + \phi(y^N) \right)$$

where we set $y^k = y(t_k)$ and $u^k = u(t_k)$. The value function is defined as

$$V^k(x) := \min_{u \in \mathcal{U}} \mathbb{E} \left(h_0 \sum_{j=0}^{N-1} \ell(t_i, u^j, y^j) + \phi(y^N) \,\middle|\, y^k = x \right)$$

which leads to

$$V^k(x) := \min_{u \in U} \mathbb{E}_x \left(h_0 l(t_k, u, x) + V^{k+1}(y^{k+1}) \right) \tag{4}$$

with final condition

$$V^N(x) = \phi(x) \tag{5}$$

We can extend this reasoning to cases where the dynamics and the cost functions depend of a mode : a diesel engine for example which can be turned off or on. If we denote M the number of modes, with a subscript i (or j) the functions corresponding to the mode i and c_{ij} the switching cost from mode i to mode j (assuming that $c_{ii} = 0$), this leads to

$$V_i^k(x) = \min_{j \in \{0,\dots,M\}} \left(c_{ij} + \min_{u \in U} \left\{ h_0 \ell_j(t_k, u, x) + \mathbb{E}_x \left[V_j^{k+1}\left(y^{k+1}\right) \right] \right\} \right) \tag{6}$$

The algorithm used to compute the Value function at t_k is the following

---

Algorithm 1 Compute V^k

Require: $0 \leq k \leq N$
  for $x \in$ Grid **do**
    if $k = N$ **then**
      $V^N(x) = \phi(x)$
    else
      for $i \in \{0, \dots, M\}$ **do**
        $\widetilde{V}_i^k(x) = \min_{u \in U} \left(h_0 \ell_j(t_k, u, x) + \mathbb{E}_x \left[V_j^{k+1}\left(y^{k+1}\right) \right] \right)$
      end for
      for $i \in \{0, \dots, M\}$ **do**
        $V_i^k(x) = \min_{j \in \{0,\dots,M\}} \left(c_{ij} + \widetilde{V}_j^k(x) \right)$
      end for
    end if
  end for

This algorithm is independent of the way of calculating $\mathbb{E}_x \left[V_j^{k+1}\left(y^{k+1}\right) \right]$. A classical method is to use an interpolation on the grid of V^{k+1} and an Euler scheme for the dynamics: this is the semi-Lagrangian method, as used in BocopHJB.

3.3 Semi Lagrangian scheme

3.3.1 Time discretization

Remark: in the following we drop the argument t^k in functions f, l for clarity.

In the deterministic case, we naturally discretize the dynamics:

$$y^{k+1} = y^k + h_0 f(u^k, y^k) \tag{7}$$

In the stochastic case, remembering that a Brownian motion has independent increments following a Normal law, $W(t_{k+1}) - W(t_k) \sim \sqrt{h_0}\mathcal{N}(0,1)$, we obtain

$$y^{k+1} = y^k + h_0 f(u^k, y^k) + \sqrt{h_0}\sigma(u^k, y^k)\mathcal{N}(0,1) \tag{8}$$

According to [3], $\mathcal{N}(0,1)$ can be replaced by any law with the same first two moments. We use a binary choice and obtain

$$y^{k+1} \simeq y^k + h_0 f(u^k, y^k) + \alpha\sqrt{h_0} e \sigma_{\mathcal{X}}(u^k, y^k) \tag{9}$$
$$\mathbb{P}(e = 1) = \mathbb{P}(e = -1) = \frac{1}{2}$$

where $\mathcal{X}$ follows an uniform distribution on $\{1, \ldots, q\}$ and we have to choose α such that the expected value and the variance of this approximated process correspond to the ones of the original process in (8). Since the normal distribution and the random variable e are centered, and e and $\mathcal{X}$ are independent, the expected value is the same for any α. The variance in (8) is $h_0\sigma\sigma^T$. The variance in (9) writes

$$\mathbb{E}\left(\alpha\sqrt{h_0} e\sigma_{\mathcal{X}}(\alpha\sqrt{h_0} e\sigma_{\mathcal{X}})^T\right) = \alpha^2 h_0 \mathbb{E}\left(e^2\sigma_{\mathcal{X}}\sigma_{\mathcal{X}}^T\right) = \alpha^2 h_0 \frac{1}{q}\sum_{s=1}^{q}\sigma_s\sigma_s^T = \frac{\alpha^2}{q} h_0\sigma\sigma^T$$

therefore we have $\alpha = \sqrt{q}$. Plugging (9) in (3) we obtain

$$V^k(x) = \min_{u\in U}\left(h_0\ell(u,x) + \frac{1}{2q}\sum_{s=1}^{q} V^{k+1}\left(x + h_0 f(u,x) \pm \sqrt{q h_0}\sigma_s(u,x)\right)\right). \tag{10}$$

3.3.2 Space discretization

We know the value of V at the points of the grid, and we want to interpolate at the point y. We choose the coefficients $\alpha_i \in [0,1]$ such that $y_j = (1-\alpha_j)x_{i_j} + \alpha_j x_{i_j+1}$. We interpolate the value function at the point y as follows (see [2]):

$$V^{k+1}(y) = \sum_{(k_1,\ldots,k_n)\in\{0,1\}^n}\left[\prod_{j=1}^{n}(1-\alpha_j)^{1-k_j}\alpha_j^{k_j}\right] V^{k+1}(x_{i_1+k_1},\ldots,x_{i_n+k_n})$$

where the sum is made on the 2^n elements of $\{0,1\}^n$.

When a point doesn't belong to the grid we cannot interpolate the value function at this point. A typical choice is to take the value of the nearest point of the grid. Depending on the problem, another sensible choice can be to take the final cost.

3.3.3 Control discretization

The minimizer of (10) is approximated by discretizing the control set U.

3.3.4 Simulation

BocopHJB includes a built-in module to simulate the optimal strategies provided by the dynamic programming algorithm. At each time step, the optimal control is taken as the minimizer of (4) over the discrete control set.

4 Description of problem files

In BocopHJB a problem is defined by the following files:

- a set of **(C/C++)** files:
  - *constraints.cpp* for the constraints of the problem (state and/or control-state)
  - *costFunctions.cpp* for the running, final and switching cost functions
  - *dynamicsHJB.cpp* for the drift and volatility
  - *simulation.cpp* for the initial conditions of the simulated trajectory
  - *optionalFunctions.cpp* for several optional functions see 4.5

- a set of text files:
  - *problemHJB.def* for general definition and settings
  - *stateDisc/* folder for state discretization
  - *controlDisc/* folder for control discretization

4.1 Definition file: problemHJB.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 control, however this is not mandatory. The line format is the following: `keyword type value`, where the keywords are listed below and the type can be integer, double or string.

- Initial and final time
  - `time.initial`: initial time t_0
  - `time.final`: final time t_f

- Dimensions
  - `state.dimension`: dimension of state variables y
  - `control.dimension`: dimension of control variables u
  - `constants.dimension`: number of numerical constants
  - `brownian.dimension`: dimension of brownian motion W

- Control discretization
  - `discretization.control.type`: discretization for each component of the control, can be "uniform" (automatic), "user_function" (see 4.5.2), or "user_file". The values for the i-th control component are in the files `controlDisc/control.i.disc` and must be filled manually if option is set to "user_file".
  - `combination.control.type`: how to build the discretized control set. It can be "uniform" (automatic), "user_function"(see 4.5.2), or "user_file". The control set is written in the file `controlDisc/control.combination`, one element per row. As above, the file must be filled manually if option is set to "user_file", in which case "discretization.control.type" is ignored.

- Time discretization
  - `discretization.time`: number of time steps

- Grid type
  - `grid.type`: type of state grid, for now the only available option is "uniform".

- Interpolation
  - `interpolation.inner`: type of interpolation for the points inside the grid, for now the only available option is "linear_interpolation".
  - `interpolation.outer`: type of interpolation for the points outside the grid, can be "final_value" for the final value, "projection" for the projection on the nearest point of the grid, or "user_function" (see 4.5.3) for a specific function coded by the user.

- Switching mode
  - `switching.modes`: number of modes among which the system can switch. Set to 1 if there are no switchings.

- Simulation
  - `simulation.directory`: the name of an existing directory inside the problem directory where the simulation results will be saved. The simulated trajectory consist in the files *simulatedTrajectory.[times,states,controls, modes]* that contain the values for $(t, x(t), u(t))$ and the mode.

    - `simulation.type`: can be "none" (only compute the value function, no trajectory simulation), "from_computed_sol" (first compute the value function, then simulate the optimal trajectory from the given initial conditions), or "from_sol_file" (read a previously computed value function file then simulate the optimal trajectory).

    - `simulation.noise`: type of noise (i.e. realization of the Brownian for the simulation), can be "none", "gaussian", or "user_function" (see 4.5.5). This parameter has no effect for deterministic problem with `brownian.dimension` set to 0.

    - `simulation.starting.mode`: set the initial mode for the simulation; "auto" picks the initial mode i_0 giving the lowest value of $V(t_0, x_0, i_0)$, "user_function" lets the user set explicitly the initial mode i_0 in simulation.cpp (see 4.4).

- Names
  - `state.i`: name of component i of y
  - `control.i`: name of component i of u

- Constants
  - `constant.i`: name and value of ith constant, the name replaces the type for constants (ex: `constant.0 c0 1.0`)

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

- Output frequency
  - `timestep.output.frequency`: frequency of the displayed output (in the terminal), can be 0 for no output at all, 1 to output every time step, or n (with n an integer less than the number of time step) to output only the time steps which are multiple of n.

4.2 State discretization file: folder stateDisc/

This file *state.grid* gives, for each component of the state, the lower and upper bounds and the number of discretization steps (uniformly spread). For instance, 10 steps in $[0, 1]$ give the discretized set $\{0, 0.1, \dots, 1\}$.
- `discretization.state.i`: number of discretization steps for component i
- `minimum.state.i`: lower bound for component i
- `maximum.state.i`: upper bound for component i

4.3 Control discretization file: folder controlDisc/

The files to be completed depend on the options "discretization.control.type" and "control.combination.type".

- **Discretized control set:** if "control.combination.type" is set to
  - "uniform", the control set will be built automatically by taking the values from each control component (see below).
  - "user_function" or "user_function_statedependent": the control set will be built by the corresponding user function (see 4.5.2).
  - "user_file": complete the file *control.combination*, each row containing a m-tuple where m is the dimension of the control space.

- **IF CONTROL.COMBINATION.TYPE=UNIFORM.**
  Individual control components: if "discretization.control.type" is set to
  - "uniform": complete the file *control.grid* with a syntax similar to *state.grid*. Individual files *control.i.disc* will be written automatically.
  - "user_function": the control component will be discretized by the corresponding user function (see 4.5.2).
  - "user_file": complete the individual files *control.i.disc* for each component of the control. Each file contains the set of discretized values for the corresponding component.

Example: Assume we have a problem with a two-dimensional control u with $u_0 \in \{0, 1\}$ and $u_1 \in \{0, 1\}$. Setting control.combination.type to "uniform" gives the discretized control set $\{(0, 0), (0, 1), (1, 0), (1, 1)\}$. If we want to impose the constraint $u_0 \geq u_1$, we can define directly the control set with *control.combination.type* set to *user_file*, and write the file `controlDisc/control.combination` as follows

```
0 0
1 0
1 1
```

4.4 Basic Functions for the optimal control problem

The user has to write the functions which define the problem: the drift and the (optional) volatility to describe the dynamics, the running cost, the final cost and the (optional)

switching cost to describe the criterion to optimize; if there are constraints, the functions to check the admissibility of the states and the controls; and some other optional functions, if the user wants to give its own functions to discretize the single controls, to make the control combinations, or to interpolate inside and/or outside the grid.

The dynamics functions f and σ are in dynamicsHJB.cpp:

```
// Drift function which describes the deterministic part of the dynamics.
void drift(const double& initial_time,
           const double& final_time,
           const double& time,
           const vector<double>& control,
           const vector<double>& state,
           const int mode,
           const int dim_constant,
           const double* constants,
           vector<double>&  state_dynamics)
```

```
// Volatility function which describes the stochastic part of the dynamics.
void volatility(const double& initial_time,
                const double& final_time,
                const double& time,
                const vector<double>& control,
                const vector<double>& state,
                const int mode,
                const int dim_constant,
                const double* constants,
                vector<double>&  volatility_dynamics)
```

Cost functions are in costFunctions.cpp:

```
// Running cost for the computation of the criterion.
void runningCost(const double& initial_time,
                 const double& final_time,
                 const double& time,
                 const vector<double>& control,
                 const vector<double>& state,
                 const int mode,
                 const int dim_constant,
                 const double* constants,
                 double& running_cost)
```

```
// Final cost for the computation of the criterion.
void finalCost(const double& initial_time,
               const double& final_time,
               const vector<double>& state,
               const int mode,
               const int dim_constant,
               const double* constants,
               double& final_cost)
```

For the simulation step, one has to set the initial state and mode in file *simulation.cpp*. Keep in mind that
- the initial point must be strictly in the interior of the state grid (ie not a boundary point).
- the initial mode must be between 0 and NbModes-1.

```
// Starting point definition.
void simulationStartingPoint(vector<double>& starting_point)
```

```
// Starting mode definition.
void simulationStartingMode(int& starting_mode)
```

4.5 More advanced features

In this part we describe some optional more advanced functions.

4.5.1 State and/or control constraints

State and control admissibility functions are in constraints.cpp:

```
// User function used to check if a combination of controls is admissible or not.
bool checkAdmissibleControl(const vector<double> control,
                            const int dim_constant,
                            const double* constants)
```

```
// User function used to check if a state is admissible or not.
bool checkAdmissibleState(const double initial_time,
                          const double final_time,
                          const double time,
                          const vector<double> state,
                          const int mode,
                          const int dim_constant,
                          const double* constants)
```

```
// User function used to check if a combination of controls and a state is admissible or not.
bool checkAdmissibleControlState(const double initial_time,
                                 const double final_time,
                                 const double time,
                                 const vector<double> control,
                                 const vector<double> state,
                                 const int mode,
                                 const int dim_constant,
                                 const double* constants)
```

4.5.2 Non uniform control discretization

Control discretization functions are in optionalFunctions.cpp. :

This function allows to define explicitly the discretized values taken by each component of the control.

```
// User function used to define the discretized controls.
// The user has to fill the values of m_discretizedControl[i][j] with i=0,...,m_dimControl
// and j=0,...,m_discretizedControl[i].size()
int userControlDiscretization()
```

This function allows to define explicitly the elements of the discret control set. Each element is an m-tuple, where m is the dimension of the control space. It can be used in particular to enforce some constraints on the control.

```
// User function to compute the combinations of controls.
// Each line of the resulting matrix is a combination of controls (u_0,..., u_p)
vector< vector<double> > userControlCombination(const int dim_constant,
                                                const double* constants)
```

The next function is similar but also take into account the state variables.

```
// User function to compute the combinations of controls when it depends on state.
// Each line of the resulting matrix is a combination of controls (u_0,..., u_p)
vector< vector<double> > userControlCombinationStateDependent(const double initial_time,
                                                              const double final_time,
                                                              const double time,
                                                              const vector<double> state,
                                                              const int mode,
                                                              const int dim_constant,
                                                              const double* constants)
```

4.5.3 Out of grid evaluation

Interpolation of the value function when it is out of the grid is in optionalFunctions.cpp:

```
// User function to compute the value of the value function for the points outside the grid.
void userOutOfGridValueFunction(const double initial_time,
                        const double final_time,
                        const double time,
                        const vector<double>& state,
                        const int dim_constant,
                        const double* constants,
                        double& result)
```

4.5.4 Switching modes

If the system has several modes (set in problemHJB.def) we must define the cost of switching from one mode to another. Modes are numbered from 0 to NbModes-1.

```
// Switching cost for the computation of the criterion.
void switchingCost(const int initial_mode,
                   const int final_mode,
                   const int dim_constant,
                   const double* constants,
                   double& switching_cost)
```

4.5.5 Brownian realization for the simulation

If simulation.noise is set to user_function, user_noise() in optionalFunctions.cpp defines the Brownian realization used in the simulation.

```
// User function to compute the noise for the simulation.
std::vector<double> user_noise()
```

References

[1] Kristian Debrabant and Espen Jakobsen. Semi-lagrangian schemes for linear and fully non-linear diffusion equations. *Mathematics of Computation*, 82(283):1433–1462, 2013.

[2] Maurizio Falcone and Roberto Ferretti. *Semi-Lagrangian approximation schemes for linear and Hamilton-Jacobi equations*. SIAM, 2013.

[3] Harold Kushner and Paul G Dupuis. *Numerical methods for stochastic control problems in continuous time*, volume 24. Springer Science & Business Media, 2013.

[4] Huyên Pham. *Continuous-time stochastic control and optimization with financial applications*, volume 61. Springer Science & Business Media, 2009.

A Install notes (INSTALL file)

```
************************
BOCOP HJB INSTALL NOTES
************************

----------------------------------------
LINUX

In the following, <BOCOPHJB> is the directory in which you have extracted
the package. Please make sure that there are no blanks or spaces in the
path name to this folder.


A. PREREQUISITES

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


B. HOW TO LAUNCH BOCOPHJB

First we recommend that you compile and run a test case. To do so you can call
the following commands from <BOCOPHJB>:

> ./bocop build examples/maze
> ./bocop run examples/maze

To define a new problem you can call the following command:

> ./bocop create_problem PROBLEM_NAME

Once you have completed the input files located in <BOCOPHJB>/problems/PROBLEM_NAME
as described in the documentation. You have to compile (build) and run BocopHJB:

> ./bocop build problems/PROBLEM_NAME
> ./bocop run problems/PROBLEM_NAME

If you want to visualize the simulation results you can call the following command:

> ./bocop visualize -s -d problems/PROBLEM_NAME

To visualize the value function you can call the following commands:

> ./bocop visualize -v -d problems/PROBLEM_NAME -m MODE_VALUE -t TIME_VALUE
```

```
Example:
> ./bocop visualize -v -d <BOCOPHJB>/examples/maze -m 0 -t 0

NB: you can use the -h option to print an help message.


----------------------------------------
MAC OS

In the following, <BOCOPHJB> is the directory in which you have extracted
the package. Please make sure that there are no blanks or spaces in the
path name to this folder.


A. PREREQUISITES

BocopHJB requires Xcode and CMake.
Please install them if necessary according to the following guideline.

A.1 XCODE

Download and install Xcode from the appstore. Please note that you have to accept
Xcode license in order to use the C++ compiler.

A.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/ccmake /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


B. HOW TO LAUNCH BOCOPHJB
```

First we recommend that you compile and run a test case. To do so you can call the following commands from <BOCOPHJB>:

```
> ./bocop build examples/maze
> ./bocop run examples/maze
```

To define a new problem you can call the following command:

```
> ./bocop create_problem PROBLEM_NAME
```

Once you have completed the input files located in <BOCOPHJB>/problems/PROBLEM_NAME as described in the documentation. You have to compile (build) and run BocopHJB:

```
> ./bocop build problems/PROBLEM_NAME
> ./bocop run problems/PROBLEM_NAME
```

If you want to visualize the simulation results you can call the following command:

```
> ./bocop visualize -s -d problems/PROBLEM_NAME
```

To visualize the value function you can call the following commands:

```
> ./bocop visualize -v -d problems/PROBLEM_NAME -m MODE_VALUE -t TIME_VALUE
```

Example:

```
> ./visualize_solution -d <BOCOPHJB>/examples/maze -m 0 -t 0
```

NB: you can use the -h option to print an help message.

WINDOWS

In the following, <BOCOPHJB> is the directory in which you have extracted the package. Please make sure that there are no blanks or spaces in the path name to this folder.

WARNING : BocopHJB must be installed in a directory without any blanks or spaces, in particular not in Program Files !

A. PREREQUISITES

BocopHJB requires MinGW and CMake to run on Windows.

A.1 MINGW

Due to some incompatibilities with the latest MinGW version, we
recommend that you use the provided full MinGW archive, available on the
Download page of bocop.org.
Simply extract the archive to a location without spaces in its
name (for instance C:\, but NOT C:\Program Files\).

In the following, <MINGW> is the installation target directory
(for example C:\MinGW which is the preferred one).

* Change the Path environment variable, as explained here :

- Right-click on your "My Computer" icon and select "Properties".
- Click on the "Advanced" tab, then on the "Environment Variables" button.
- Click on the PATH entry and edit it.
- Scroll to the BEGINNING of the string and add the directories for your MinGW:

  <MinGW>\msys\1.0\bin;<MinGW>\bin;

Note: we recommend to put the two directories for MinGW at the beginning of the
PATH to avoid the confusion with other versions of files such as sed.exe or
libtools that may be present in your system folders. Such files can be installed
by other applications, and may not be compatible with the building process in
Bocop.

A.2 CMAKE

The building process requires CMake. You can download the installer here:
http://www.cmake.org/cmake/resources/software.html

During the installation process choose the option to add the CMake path
in the Path environment variable.

When this is done please reboot your computer to update the Path
environment variable.
IMPORTANT:
CMake under Windows assumes building with Visual Studio by default.
Since we currently use MinGW instead, we have to add the option
-G "MSYS Makefiles" as stated below.

A.3 PYTHON

Visualization process need Python 2.7 installed.

B. HOW TO LAUNCH BOCOPHJB

First we recommend that you compile and run a test case. To do so you can call the following commands from <BOCOPHJB>:

```
> sh bocop build examples/maze
> sh bocop run examples/maze
```

/!\ Please note that the check need python installed to be passed /!\

To define a new problem you can call the following command:

```
> sh bocop create_problem PROBLEM_NAME
```

Once you have completed the input files located in <BOCOPHJB>/problems/PROBLEM_NAME as described in the documentation. You have to compile (build) and run BocopHJB:

```
> sh bocop build problems/PROBLEM_NAME
> sh bocop run problems/PROBLEM_NAME
```

If you want to visualize the simulation results you can call the following command:

```
> sh bocop visualize -s -d problems/PROBLEM_NAME
```

To visualize the value function you can call the following commands:

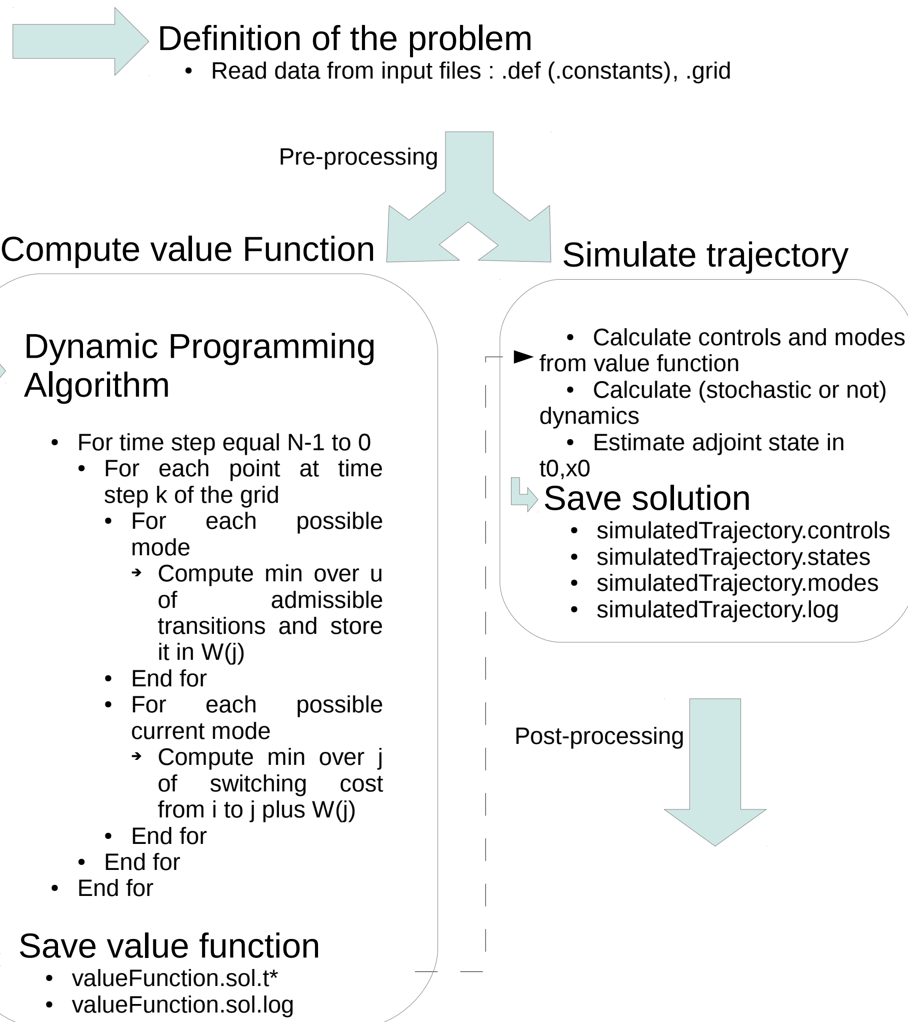
```
> sh bocop visualize -v -d problems/PROBLEM_NAME -m MODE_VALUE -t TIME_VALUE
```

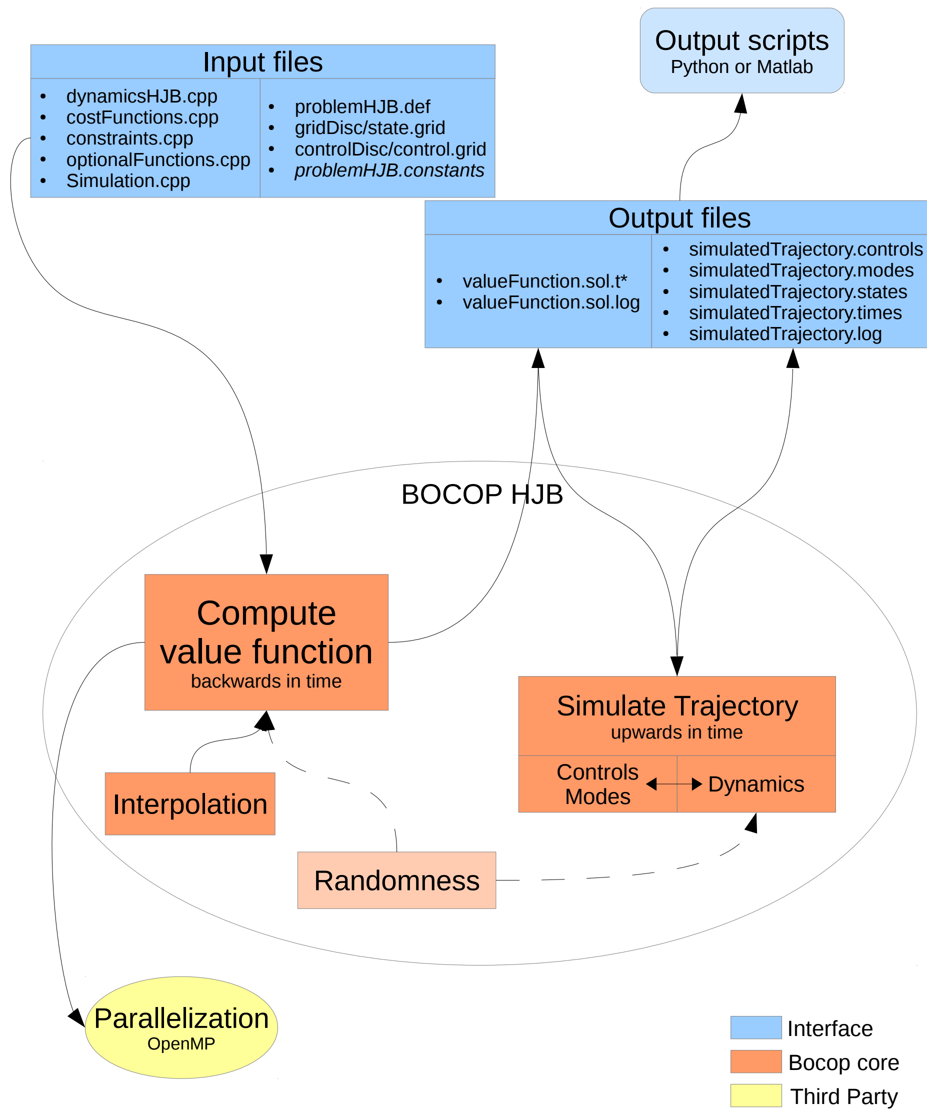
Example:

```
> sh bocop visualize -v -d <BOCOPHJB>/examples/maze -m 0 -t 0
```

NB: you can use the -h option to print an help message.

B Code structure





Input files
dynamicsHJB.cpp
costFunctions.cpp
constraints.cpp
optionalFunctions.cpp
Simulation.cpp
problemHJB.def
gridDisc/state.grid
controlDisc/control.grid
problemHJB.constants
Output scripts
Python or Matlab
Output files
valueFunction.sol.t*
valueFunction.sol.log
simulatedTrajectory.controls
simulatedTrajectory.modes
simulatedTrajectory.states
simulatedTrajectory.times
simulatedTrajectory.log
BOCOP HJB
Compute value function
backwards in time
Simulate Trajectory
upwards in time
Controls
Modes
Dynamics
Interpolation
Randomness
Parallelization
OpenMP
Interface
Bocop core
Third Party