User's guide for the ROC-HJ[*] solver: Finite Differences and Semi-Lagrangian methods

March 31, 2017

Version 2.2[†]

Current developers: O. Bokanowski, A. Désilles, H. Zidani, J. Zhao.

Contents

1 Problems handled **2**

1.1 Time-dependant equations 2

1.2 Steady equations 4

1.3 Optimal trajectory reconstruction 6

2 Compilation and execution (under GCC and CMake) **6**

2.1 System requirements 6

2.2 ROC-HJ-Editor 7

2.3 Compilation and execution 7

3 Model definition header files **9**

3.1 Data describing the problem to be solved 9

3.2 Numerical schemes and associated parameters 15

3.2.1 Finite Difference Method 16

3.2.2 Semi-Lagrangian Method 16

3.3 Value problems defined implicitly 17

3.4 Execution and output parameters 18

3.5 Parameters for trajectory reconstruction 19

3.6 Output parameters and files 21

3.6.1 Savings options 21

[*]Reachability, Optimal Control, and Hamilton-Jacobi equations

[†]Including a SL solver for second order HJ equations, see Section 6.

3.6.2 Data formatting options 22
3.6.3 General data files 23
3.6.4 Files related to trajectory reconstruction 23
3.7 Advanced parameters 24

4 Graphic outputs **24**

5 Examples **25**
5.1 Rotation example 25
5.2 Eikonal equation 25

6 Second order HJB equations and SL scheme **26**

7 Source architecture **28**

References **28**

Index **30**

1 Problems handled

This is a c++ MPI/OpenMP library for solving d-dimensional Hamilton-Jacobi-Bellman equations by finite difference methods, or semi-lagrangian methods. First order and second order HJ equations time-dependent or steady equations can be solved. For optimal control applications, some algorithms of optimal trajectory reconstruction are also implemented in this library.

1.1 Time-dependant equations

The problem is to find $u = u(t,x)$ solution of

$$\frac{\partial u}{\partial t}(t,x) + \lambda(x)u(t,x) + H(t,x,u(t,x),\nabla u(t,x)) = 0, \quad (t,x) \in [0,T] \times \Omega \tag{1a}$$

$$u(t,x) = g_b(t,x), \quad t \in [0,T],\ x \in \partial\Omega \tag{1b}$$

$$u(0,x) = u_0(x), \quad x \in \Omega \tag{1c}$$

where Ω is a domain of $\mathbb{R}^d$ of the form $\prod_{i=1}^d [a_i,b_i]$, and g_b, $\lambda(x)$ and u_0 are given functions. Instead of (1b), it is possible to consider other type of boundary conditions:

$$u(t,x+k\pi) = u(t,x), \quad x \in \partial\Omega,\ k \in \mathbb{Z}^d, \quad \text{(periodic boundary condition)} \tag{2a}$$

for a given vector $\pi \in \mathbb{R}^d$, or

$$\frac{\partial u}{\partial n}(t,x) = g_{mix}(t,x,u(x)), \quad x \in \partial\Omega, \quad \text{(Neumann type boundary condition)} \tag{2b}$$

or

$$\partial_{xx} u(t,x) = 0, \quad x \in \partial\Omega \quad \text{(linear boundary extrapolation).} \tag{2c}$$

It is also possible to consider a combination of different types of boundary conditions: one can choose one of the above type of boundary condition, and force periodic boundary conditions on some given directions.

The function $H : [0,T] \times \Omega \times \mathbb{R}^d \to \mathbb{R}$ can be defined either by:

- an analytic expression, if such an expression is known (for example $H(t,x,p) := c(t,x)\|p\| + f(t,x) \cdot p$);
- or a Hamiltonian corresponding to a one-player control problem:

$$H(t,x,p) := \max_{a \in \mathcal{A}} \Big(- f(t,x,a) \cdot \nabla u - \ell(t,x,a) \Big), \tag{3a}$$

  or

$$H(t,x,p) := \min_{a \in \mathcal{A}} \Big(- f(t,x,a) \cdot \nabla u - \ell(t,x,a) \Big), \tag{3b}$$

  where A is a set of control values, of the form $\prod_{i=1}^{m_A} [\alpha_i, \beta_i]$ (with $m_A \geq 1$), and where the dynamics $f : [0,T] \times \mathbb{R}^d \times \mathbb{R}^{m_A} \times \mathbb{R}^d$ and the distributed cost $\ell : [0,T] \times \mathbb{R}^d \times \mathbb{R}^{m_A} \to \mathbb{R}$ are given functions (to be defined by the user);
- or a Hamiltonian function corresponding to a two-player game:

$$H(t,x,\nabla u) := \max_{a \in \mathcal{A}} \min_{b \in \mathcal{B}} \Big(- f(t,x,a,b) \cdot \nabla u - \ell(t,x,a,b) \Big), \tag{3c}$$

  or

$$H(t,x,\nabla u) := \min_{a \in \mathcal{A}} \max_{b \in \mathcal{B}} \Big(- f(t,x,a,b) \cdot \nabla u - \ell(t,x,a,b) \Big), \tag{3d}$$

  where $\mathcal{A}$ and $\mathcal{B}$ are control sets of the form $\prod_{i=1}^{n_A} [\alpha_i^A, \beta_i^A]$ and $\prod_{j=1}^{n_B} [\alpha_j^B, \beta_j^B]$, and f, ℓ are given functions.

Time-dependent obstacle problem

$$\begin{cases} \min\left(\frac{\partial u}{\partial t} + H(t,x,\nabla u), u(t,x) - g_{obs}(t,x)\right) = 0, \quad x \in \Omega, \quad t \in [0,T], \\ u(0,x) = u_0(x), \quad x \in \Omega \\ \text{boundary conditions on } \partial\Omega \times (0,T). \end{cases} \tag{4}$$

Here H, Ω and u_0 are as above and the function $g_{obs} : (0,T) \times \mathbb{R}^d \to \mathbb{R}$ is a given function. For some optimal control problems such as target problems, the set $\{(t,x), g_{obs}(t,x) \leq 0\}$ may represent the set of state constraints, see [2, 1].

Second order time-dependent equations Some second order equations can also be treated, of the following type

$$\frac{\partial u}{\partial t} + \lambda(x)u \tag{5a}$$

$$+\max_{a\in\mathcal{A}}\left\{-\ell(t,x,a) + r(t,x,a)u - b(t,x,a)\cdot\nabla u - \frac{1}{2}Tr(\sigma(t,x,a)\sigma(t,x,a)^T D^2 u)\right\} = 0$$

$$t \in [0,T],\ x \in \mathbb{R}^d,$$

$$u(0,x) = u_0(x), \qquad x \in \mathbb{R}^d$$

boundary conditions on $\partial\Omega \times (0,T)$.

where $\mathcal{A}$ is some non empty compact subset of $\mathbb{R}^m$ ($m \geq 1$), (of the form $\prod_{i=1}^{n_A}[\alpha_i, \beta_i]$), $b(t,x,a)$ is a vector of $\mathbb{R}^d$, $r(t,x,a)$ and $\ell(t,x,a)$ are real-valued, and $\sigma(t,x,a)$ is a $d \times p$ real matrix (for some $p \geq 1$). This problem is linked to the computation of the value function of stochastic optimal control problems. See section 6 for more details (a general semi-Lagrangian scheme is proposed in the software).

1.2 Steady equations

The problem is to find $u = u(x)$ solution of an time-dependant Hamilton-Jacobi equation:

$$\lambda(x)u(x) + H(x, u(x), \nabla u(x)) = 0, \quad x \in \Omega, \tag{6a}$$

$$u(x) = g_b(x), \quad x \in \partial\Omega. \tag{6b}$$

with H given directly or in the form of (3). Obstacle problem can be also solved, of the form:

$$\min\left(\lambda u(x) + H(x, \nabla u), u(x) - g_{obs}(x)\right) = 0, \quad x \in \Omega \tag{7a}$$

$$u(x) = g_b(x), \quad x \in \partial\Omega \tag{7b}$$

(in that case, assuming that $g_b(x) \geq g_{obs}(x)$).

Some second order stationary equations of the form

$$\lambda(x)u(x) + H(x, u(x), \nabla u(x), D^2u(x)) = 0, \quad x \in \Omega, \tag{8a}$$

$$u(x) = g_b(x), \quad x \in \partial\Omega. \tag{8b}$$

can also be solved. It is advised that the function $\lambda(x)$ be strictly positive (for convergence of the solver). The domain Ω an hyperrectangle of $\mathbb{R}^d$ of the form $\prod_{i=1}^d [a_i, b_i]$.

It is possible also to solve (6a) in a subdomain $\mathcal{C}$. In that case the set $\mathcal{C}$ should be defined such that

$$\mathcal{C} := \{x,\ g_{domain}(x) < 0\} \tag{9}$$

(equivalently, $\Omega \backslash \mathcal{C} = \{x \in \Omega,\ g_{domain}(x) \geq 0\}$). The boundary conditions at the border of $\partial\mathcal{C}$ should be of Dirichlet type, fixed by the "initial" condition $u_0(x)$:

$$u(x) = u_0(x), \quad x \in \partial\mathcal{C}, \tag{10}$$

This equation can be solved by using an iterative procedure - or "value iteration algorithm", where the part $\lambda(x)u(x)$ is treated implicitly. Note that for steady equations (as well as time dependent equations), the basic finite difference (FD) scheme is based on the following iteration: for $n \geq 0$, for a fixed $\Delta t > 0$,

$$\frac{u^{n+1}(x) - u^n(x)}{\Delta t} + \lambda(x)u^{n+1}(x) + h(x, u^n(x), Du^n(x)) = 0, x \in \Omega, \tag{11}$$

where $h(x, u^n(x), Du^n(x))$ corresponds to a numerical approximation of $H(x, u^n(x), \nabla u^n(x))$ and the term $\lambda(x)u(x)$ is treated in an implicit way.

In the case the problem is defined only in a subdomain $\mathcal{C}$ as in (9), the iterations (11) are performed only at grid points $x \in \mathcal{C}$. (The parameter `COMPUTE_IN_SUBDOMAIN` should be set to `1` in that case, see section 3.1.)

For such steady equations, once u^{n+1} has been computed, iterations are then stopped as soon as $\|u^{n+1} - u^n\|_{L^\infty}$ is smaller than a given threshold. Therefore we obtain the following recursion:

$$u^{n+1}(x) = \frac{1}{1 + \Delta t\, \lambda(x)} \bigg(u^n(x) - \Delta t\, h(x, u^n(x), Du^n(x)) \bigg) = 0, \quad x \in \mathcal{C},$$

then if $\|u^{n+1} - u^n\|_{L^\infty} \geq \epsilon$ go to the next step, otherwise stop.

For the SL method, the iterative scheme is based on an implicit treatment of the $\lambda(x)u(x)$ term, and becomes, in the case of (6a):

$$u^{n+1}(x) = \min_a \frac{1}{1 + \Delta t \lambda(x)} \bigg([u^n](x + \Delta t f(t, x, a)) + \ell(t, x, a) \bigg) = 0, \quad x \in \mathcal{C}.$$

Here $[u^n]$ denotes the Q_1 interpolation of u^n on the spatial grid mesh.

1.3 Optimal trajectory reconstruction

In the case when the value function associated to an optimal control problem (governed by an ODE) has been solved by using for the corresponding HJB equation and by using the software, it is possible to construct also the optimal trajectory corresponding to the optimal control problem. In order to compute the optimal trajectory of the concerned problem, some reconstruction algorithms are included in the library (either based on the dynamic programming principle, or by using a minimal function assciated to the problem). In addition, the user has the possibility to implement other reconstruction methods and to test them with the software. See section 3.5 for more details.

2 Compilation and execution (under GCC and CMake)

2.1 System requirements

The library is available for Windows, Linux and Mac operating systems. In all cases, in order to use the library, one needs some building tools for `c/c++` to be installed:

- the CMake Build chain (`cmake`) available for all systems at http://www.cmake.org/.
- a GCC compiler[1] (`c++`, ...)

The folders are organized as follows.

Folders `src`, `include`, `lib`:

- Folder `lib` contains the pre-compiled library `libhj.a`. This library includes the implemented numerical methods (classes `HJB, HJB_SL` and `HJB_FD`), all data management methods (grids, options, ...).
- The header files containing the class declarations are in the folder `include`.
- The master `c++` file is `main.cpp` and is in the folder `src`. The folder `src` contains also the file `HJB_user.cpp` to allow the user to add some personal implementation of trajectory reconstruction methods.

These folders should not be renamed.

[1]native for Linux and Mac OS, `minGW` distribution for Windows : http://www.mingw.org/

Folders `data`, `OUTPUT`, `VALUE`: The user has to create a new model file as a C-header file, named like `data_myModel.h`, that contains the definition of the drift, distributed-cost, the hamiltonian function (if known analytically), the initial conditions, the boundary conditions, the obstacle function (if any) and all the parameters that describe the model to be solved.

Some basic knowledge of C programming syntax is necessary to describe a new problem to be solved by the HJB-Solver.

The structure of a data file is predefined and for the user's convenience several models of such a file are already available in the folder `data`. In particular:

- `data_basicmodel.h`: shows the main functions that have to be defined. It gives some basic options for the computation and for plotting results;
- `data_advancedmodel.h`: similar model implemented by an other approximation method for the Hamiltonian.

When the data file "`data_myModel.h`" is created, the user should include its name in the beginning of the file `data/data_simulation.h`: `\#include data_myModel.h`.

The file `data_simulation.h` should not be removed or renamed.

The solver's executable needs two folders named `OUTPUT` and `VALUE` where the outputs of the simulation will be stored. (If these folders are removed or renamed, the program will not be able to save outputs propertly and will signal an error.)

2.2 ROC-HJ-Editor

The ROC-HJ-Editor is available in the folder `ROC-HJ-Editor`. This application is designed to assist with the process of editing the template for a new data file. The application shows main possible options and helps the user to define a new data file adapted to the problem to be solved. When creating a new template model using the ROC-HJ-Editor, two files are generated: `data_user_myModel.h` which contains the parameters and functions that the user has to define; `data_default_myModel.h` which contains predefined parameters that the user does not need to handle (but that are mandatory for running the program). Further detailed instructions on how to construct a header file and how to compile the program are presented in the User's Manual for ROC-HJ solver.

2.3 Compilation and execution

Once the data file is defined and declared in the file `data_simulation.h` (see above), the project can be compiled and executed. The compilation/execution process is composed of several steps:

- Building the Makefile: `cmake .` (do not forget the "dot")
- Compilation: `make` (source `.cpp` files are generally in the directory `src/`)
- Options: `make clean` (cleaning some `.o` files), `make cleanall`.
- Eventually, before compilation, use "`./clean`" to first erase all unecessary files (can be done before the "`cmake .`" command.)

Sequential execution mode Execution (basic):

```
./exe
```

Execution (with options)

```
./exe -nn NN -nc NC
```

Options :

- option `-nn NN`: number of mesh points per dimension
- option `-nc NC`: number of controlsuitable for the equation to be solved.s per command's dimension.

OpenMP execution mode Some options run with OpenMP. Execution :

```
./exe -nt nbth
```

where `nbth` is the number of threads (typically try `nbth=2` or `nbth=4`).

Parallel code (Parallel MPI version only)

Execution:

```
mpirun -n MPI_PROCS ./exe -nn NN -nc NC -nt OMP_THREADS -nd MPI_GRID_DIM
```

where

- `MPI_PROCS`: number of MPI processors (default = 1)
- `NN`: number of mesh points per dimension
- `NC`: number of controls per command's dimension
- `OMP_THREADS`: number of OpenMP threads (default = 1)
- `MPI_GRID_DIM`: dimension of the MPI mesh grid decomposition (2 or 3) (default = 2).

Example 1 : `mpirun -n 2 ./exe -nn 200` will execute the program with 2 MPI processors and 200 mesh points per dimension.

Example 2 : `mpirun -n 64 ./exe -nn 500 -nc 10 -nt 4 -nd 2` will execute the program with `MPI_PROCS=64` MPI processors, `NN=500` points per dimension, `NC=10` controls per command's dimension, `OMP_THREADS=4` OpenMP threads, and `MPI_GRID_DIM=2`, the dimension of the MPI mesh grid decomposition.

Example 3 : `mpirun -n 1 ./exe -nn 500 -nc 10 -nt 4`.

The OpenMP parallelization is working for Semi Lagrangian and Finite Difference methods. When using only this method, one can indicate only one process for the `mpirun` command (for instance `mpirun -n 1 ./exe -nt 4` for using 4 threads) or, equivalently, use `./exe -nt 4`.

3 Model definition header files

The file `data/data_simulation.h` is the main user's input file and contains the parameters that define the equation to be solved. It can include an other `data_xxx.h` file (see basic examples `data_basicmodel.h`, `data_FD_2d_ex1_basic.h`).

3.1 Data describing the problem to be solved

The parameter `NAME` defines the name of the problem that will by used only for shell output during the execution.

State variables and controls .

- `DIM`: dimension d of the problem
- `XMIN[DIM] , XMAX[DIM]`: the boundary of the computation domain in each direction
- `PERIODIC[DIM]`: this parameter sets the periodicity for each component of the state variable x:

  `1` : the variable x_i is periodic

  `0` : the variable x_i periodic

For a one-player control problem :

- `cDIM` dimension n_c of the control variable a

- `UMIN[cDIM] , UMAX[cDIM]` min and max values of the controls in each direction; this defines the set of controls as a cube $\prod_{i=1,n_c}[\alpha_i, \beta_i]$
- `NCD[cDIM]` number of commands per direction; this defines a grid on the set of controls;

For a two-player control problem :

- `cDIM` and `cDIM2` dimensions n_A and n_B, of the control variables a and b respectively.
- `UMIN[cDIM] , UMAX[cDIM]` min and max values of the controls a in each direction; this defines the set of controls as a cube $a \in \prod_{i=1,n_c}[\alpha_i, \beta_i]$;
- `UMIN2[cDIM2] , UMAX2[cDIM2]` min and max values of the controls b in each direction; this defines the set of controls as a cube $b \in \prod_{i=1,n_c}[\alpha_i^2, \beta_i^2]$;
- `NCD[cDIM]` and `NCD2[cDIM2]` number of controls per direction; this defines a grid on each of two sets t of controls;

Dynamics and cost functions For a one-player control problem :

- function `dynamics`: $f(t, x, a)$: the dynamics of the controlled system
- function `distributed_cost`: $\ell(x, a)$: the running cost of the optimal control problem
- function `discount_factor` : $\lambda(x)$. For steady equations, this function should be strictly positive.

For a two-player control problem :

- function `dynamics2`: $f(t, x, a, b)$: the dynamics of the controlled system
- function `distributed_cost2`: $\ell(x, a, b)$: the running cost of the optimal control problem
- function `discount_factor` : $\lambda(x)$. For steady equations, this function should be strictly positive.

Special parameters for second order equations

- function `funcR`: implements the term $r(t, x, a)$ in (5);
- function `funcY`: implements the term $\sigma(t, x, a)$ in (5);

Additional computations

- TOPT_TYPE: when an optimal time function associated with the computed solution u is defined this parameter defines what kind of time optimal problem must be solved :
  - TOPT_TYPE=0 the minimum time function will be associated with the computed solution u as:

    $$\mathcal{T}_{min}(x) = \min\{t \in [0,T],\ u(t,x) \leq 0\} \tag{12}$$

    Note that if there is no time t such that $u(t,x) \leq 0$, $\mathcal{T}_{min}(x)$ will be attributed the value INF.
  - TOPT_TYPE=1 the exit time function (hereafter also refered as maximum time function) will be associated with the computed solution u as:

    $$\mathcal{T}_{max}(x) = \max\{t \in [0,T],\ u(t,x) \leq 0\} \tag{13}$$

    Note that if t exists in $[0,T]$ such that $u(t,x) \leq 0$, then $\mathcal{T}_{max}(x)$ will be attributed the value -INF (or some stricly negative small number).
- COMPUTE_TOPT: should be set to 1 in order that the minimal (resp. maximal) time function be saved in topt.dat, and can be used for optimal trajectory reconstruction when TRAJ_METHOD=0.
- function Vex: if known, the exact solution may be implemented (in that case, set also COMPUTE_VEX=1).

Hamiltonian function and associated parameters

- COMMANDS $\in$ {0,1,2}. This parameter indicates how the Hamiltonian function $H(x,p)$ should be defined, using an explicit definition or a definition using a minimization (or a maximization) over a discrete set of controls.

  Case "COMMANDS=0": The hamiltonian function $H(t,x,p)$ should be known explicitly. In that case, the user has to define the function Hnum with the expression of a numerical hamiltonian that is consistent with $H(x,p)$. Some examples are given in data_advancedmodel.h, data_FD_2d_ex1_advanced.h. In addition, the user must complete the function compute_Hconst to define d constants which are bounds for $\left|\frac{\partial H}{\partial p_i}(x,p,t)\right|$ $(i = 1,\dots,d)$, for $x \in \Omega$ and for possible gradient values p and time t. This is then used to set the time step Δt in order to satisfy a CFL condition. These constants may also be used in the function Hnum when the Lax-Friedrich numerical hamiltonian is used.

(The user may also define directly the constants c_i =`Hconst[i]` and initialize the previous function accordingly.)

Case "`COMMANDS=1`": This corresponds to the case when the HJB equation describes the value of an optimal control problem with **one player** When this value is chosen, the Hamitonian is calculated by optimization over the grid of controls α, as in (3). (Hence the Hamiltonian of the problem may not be explicitly known.) The function `Hnum` may remain in the header file but will not be used in the computations. Here, the program will use a numerical hamiltonian function corresponding to a finite difference approximation of $H(t,x,\nabla u) = \max_\alpha(-f(t,x,\alpha).\nabla u - \ell(t,x,\alpha))$, as follows:

$$H(t,x,p^-,p^+) = \max_\alpha \left(\sum_{i=1}^{d} \max(-f_i(t,x,\alpha),0)p_i^- + \min(-f_i(t,x,\alpha),0)p_i^+ - \ell(t,x,\alpha) \right),$$

(here in the case `OPTIM=MAXIMUM`). Examples are given in `data_basicmodel.h` (rotation), or `data_FD_2d_ex1_basic.h` (eikonal equation).

Case `COMMANDS=2`: This corresponds to the case when the HJB equation is related to a **two player** game, and the Hamiltonian is defined as a *min/max* or a *max/min* over as set of controls, as in (3c) or (3d). The function `Hnum` may remain in the header file, but will not be used in the computations. Here, in the case of a *max/min* Hamiltonian function (i.e., `OPTIM=MAXMIN`):

$$H(t,x,\nabla u) := \max_\alpha \min_\beta(-f(t,x,\alpha,\beta).\nabla u - \ell(t,x,\alpha,\beta)),$$

the program will use a numerical hamiltonian function corresponding to a finite difference approximation similar to the one-player case.

- `OPTIM` $\in$ `{MINIMUM, MAXIMUM, MINMAX , MAXMIN}`. when `COMMANDS` is `1` or 2, this parameter defines the choice of the optimization type to use in the definition of the Hamiltonian function.

Boundary conditions The parameters are:

- function `v0`: corresponds to the initial data u_0. This function is also used to initialize the iterations in the case of steady equations.

- `BOUNDARY` $\in$`{0,1,2,3}`:

  `BOUNDARY=0`: there is no boundary treatment. The value of `v0` will be used in the initialisation to set the boundary (ghost cell) values, and these values will not change afterwards.

BOUNDARY=1: utilizes Dirichlet boundary conditions for each direction d which is not periodic (i.e., such that PERIODIC[d]=0). The boundary condition should be defined in the function g_border.

BOUNDARY=2: utilizes a mixed boundary condition of the form $\frac{\partial v}{\partial n} = g_{mix}(t, x, v(x))$ (because of the box domain, this corresponds to $\pm\frac{\partial v}{\partial x_i} = g_{mix}(t, x, v(x))$). The function g_bordermix is then used (for each direction d which is not periodic, i.e., such that PERIODIC[d]=0).

BOUNDARY=3: utilizes a special $v_{x_i x_i} = 0$ boundary condition for each direction d which is not periodic (note that this kind of approximation is in general unstable).

Furthermore, if PERIODIC[d]=1, then periodic boundary conditions will be used in the variable x_d.

- g_border, g_bordermix : used to set different types of boundary conditions.
- BORDERSIZE[dim]: list of integers to define the number of ghost cells used in each direction (at left or right boundary of the domain). For instance, for dim=2, BORDERSIZE[dim]={2,2} defines a boundary with two ghost cells in each direction. In this case, if the mesh size is Nx*Ny, then the grid including ghosts cells is of size (Nx+4)*(Ny+4).
- EXTERNAL_v0 $\in$ {0,1} : If set to 0 (default value), it initializes the first iterate u^0 for $n = 0$ with the function v0. Otherwise, if set to 1, then the initialization of u^0 is done by using the values stored in the file VF.dat, located in OUTPUT/VF.dat (or OUTPUT/VF_PROCxx.dat if parallel MPI is used).

Initial conditions

- EXTERNAL_v0 $\in$ {0,1} : If set to 0 (default value), it initializes the first iterate u^0 for $n = 0$ with the function v0. Otherwise, if set to 1, then the initialization of u^0 is done by using the values stored in the file VF.dat, located in OUTPUT/VF.dat (or OUTPUT/VF_PROCxx.dat if parallel MPI is used).

Computation in a subdomain

- COMPUTE_IN_SUBDOMAIN $\in$ {0,1}: determines if subdomain computations should be done, so that the evaluation of $u^{n+1}(x)$ is done only at grid mesh points x such that $g_{domain}(x) < 0$. (For front propagation problems and in the presence of obstacle terms, this can be used in order to reduce significantly the CPU time.)
- function g_domain(x): function used to define the subdomain.

Obstacle terms Instead of solving $u_t + H(t, x, u, \nabla u) = 0$, the solver can also treat HJ obstacle equations such as

$$\min\left(\frac{\partial u}{\partial t} + H(t, x, u, \nabla u),\ u - g(t, x)\right) = 0. \tag{14}$$

where $g(t, x)$ is defined by the user. In particular it forces to have $u(t, x) \geq g(t, x)$ in the case of (14).

Also, the following obstacle equations can be considered:

$$\max\left(\frac{\partial u}{\partial t} + H(t, x, u, \nabla u),\ u - \tilde{g}(t, x)\right) = 0 \tag{15}$$

or

$$\max\left(\min\left(\frac{\partial u}{\partial t} + H(t, x, u, \nabla u),\ u - g(t, x)\right), u - \tilde{g}(t, x)\right) = 0, \tag{16}$$

where $\tilde{g}(t, x)$ is user-defined. We will have $u(t, x) \leq \tilde{g}(t, x)$, in the case of (15), or $u(t, x) \in [g(t, x), \tilde{g}(t, x)]$ in the case of (16).

For this, the following parameters are used:

- `OBSTACLE` $\in$ `{0,1}`

  `0` : no obstacle g taken into account.

  `1` : Equation (14) is treated, with the obstacle g .

- function `g_obstacle`

- `OBSTACLE_TILDE` $\in$ `{0,1}`

  `0` : no obstacle $\tilde{g}$ taken into account.

  `1` : Equation (15) is treated, with the obstacle $\tilde{g}$.

- function `g_obstacle_tilde`

- In the case we set `OBSTACLE=1` and `OBSTACLE_TILDE=1` then (16) is treated, and both functions g and $\tilde{g}$ must be defined. (The obstacle functions should satisfy $g(x) \leq \tilde{g}(x)$ in order to avoid undesired results).

- `PRECOMPUTE_OBSTACLE` $\in$`{0,1}`: gives the possibility to precompute all obstacle terms, before doing the main iterations. This can reduce CPU time when the obstacle functions are not time-dependant.

3.2 Numerical schemes and associated parameters

Discretization parameters

- `ND[DIM]`: tab containing the size mesh in each direction (cartesian mesh)
- `MESH` $\in$ `{0,1}`: default is `1`. It utilizes `ND[i]+1` points in direction `i`. (If `MESH=0` the mesh points are at the center of the mesh cells, and there are `ND[i]` points in direction `i`)
- `DT`: the time step used for the solver, for the evolutive equation.
  - For the finite difference approach, if `DT=0`, then time step `DT` is computed so that the CFL condition be satisfied, that is, such that

  ```
   DT * (Hconst[0]/dx[0] + Hconst[1]/dx[1] + ... ) <= CFL
  ```

  where the CFL number belongs to $[0, 1]$. Otherwise, if `DT>0`, then the value `DT` is used for the time step.
  - For the semi-lagrangian approach and for the stationnary equation, the parameter h in the iterative procedure is also set to `DT`

Stopping criteria

- `T`: terminal time
- `MAX_ITERATION` : to stop the program when this maximum number of iterations is reached.
- `EPSILON` : for the stopping criteria. If set to `0`, do nothing. If set to some positive value ϵ, then the program will stop at iteration $n+1$ as soon as

  $$\|u^{n+1} - u^n\|_{L^\infty} := \max_i |u_i^{n+1} - u_i^n| \ \le \ \epsilon.$$

  (the L^∞ error between two successive steps is smaller than ϵ).

Therefore, in the case of a time dependant problem, iterations are performed until $t_n = T$, where the program stops. The parameter `EPSILON` should be set to `0`, and `T` is used to fix the terminal time. (`MAX_ITERATION` should be set to a sufficiently large value).

In the case of a steady problem, `EPSILON` should be set to a strictly positive (small) value, and `MAX_ITERATION` should be also used to limit the number of maximum iterations in the case of convergence problems.

The parameter `MAX_ITERATION` can also be set to `1` or a small integer value for debugging purposes.

The choice of the solver

- `METHOD` $\in$ `{MDF,MSL}`:

  <u>`MDF`</u>: Finite Difference Method. The finite difference method can be used with all possible values of the parameter `COMMANDS` described before.

  <u>`MSL`</u>: Semi-Lagrangian Method. This method does not use the numerical Hamiltonian function `Hnum`, rather mainly the dynamics and the distributed costs functions. **Important:** It is assumed that the Hamiltonian is of the form (3a), (3b). The parameter `COMMANDS` has no effect on the method. (The case of $min - max$ or $max - min$, i.e. (3c) or (3d) is not yet supported by the software.)

3.2.1 Finite Difference Method

This method is used when `METHOD=MDF`. The user has to set the following scheme discretization parameters:

- `CFL` $\in]0, 1[$: the constant of the CFL condition that is used in finite difference methods to adjust the time step `DT`.

- `TYPE_SCHEME` $\in$ `{LF,ENO2}`: type of spacial discretization

  `LF` : Lax-Friedrich scheme (first order scheme)

  `ENO2` : ENO scheme of second order to approximate the derivatives ∇u

- `TYPE_RK` $\in$ `{RK1,RK2,RK3}`: Time discretization by a Runge-Kutta method of order 1, 2 or 3.

  `RK1` : RK method of order 1

  `RK2` : RK method of order 2

  `RK3` : RK method of order 3

3.2.2 Semi-Lagrangian Method

This method is used when `METHOD=MSL`. One needs to define :

- `TYPE_SCHEME` $\in$ `{STA,EVO}`:

  `STA` : stationnary case, for solving (6a). In this case, the scheme is based on an iterative procedure.

`EVO` : dynamic case, for solving (1). It is also advised to use this mode to solve steady equations, and a stopping criteria based on the parameter `EPSILON`.

- `ORDER` $\in$`{1,2}`: this value is set to `1` for solving first order equations and to `2` to solve second order equations (see other section below for second order HJB equations).

- `TYPE_STA_LOOP` $\in$ `{NORMAL, SPECIAL}`: Mesh loop order during mainloop for the stationary case

  `NORMAL` : normal ordering loop

  `SPECIAL` : special ordering loop (makes 2^d loops at each iteation, modifiying the current values of the data `v` during each loop)

- `TYPE_RK` $\in$ `{RK1_EULER,RK2_HEUN,RK2_PM}`:

  `RK1_EULER` : RK1 Euler scheme

  `RK2_HEUN` : RK2 Heun scheme

  `RK2_PM` : RK2 Mid-point scheme

- `INTERPOLATION` $\in$ `{BILINEAR,PRECOMPBL,DIRPERDIR}`:

  `BILINEAR` : (default value) bilinear interpolation ($Q1$ interpolation)

  `PRECOMPBL` : precompute the interpolation coefficients (to use with tiny mesh sizes)

  `DIRPERDIR` : another interpolation method (obsolete)

- `P_INTERMEDIATE` : number of intermediate time steps to go from t_n to $t_n + \Delta t$ for computing a characteristic for a given RK method.

3.3 Value problems defined implicitly

Some special computations can be made in addition to solving the HJB equation in the case when the computed d-dimensional solution function u is used to characterize the epigraph of an other value function v of a $d-1$ dimensional problem. It is assumed that both value functions are related to each other as follows:

$$v(t, x_1, \ldots, x_{d-1}) = \inf\{z,\ u(t, x_1, \ldots, x_{d-1}, z) \leq 0\}. \tag{17}$$

So the value function v can be considered as being implicitly defined by the computed function u.

- `VALUE_PB`$\in$`{0,1}` to allow these additional computations, set `VALUE_PB=1`. Otherwise, set it to `0`.

Also when `VALUE_PB=1` it is assumed that only the last state variable, x_d, can be used to define an implicit value function. This convention must be taken into account when defining the dynamical system and its parameters in the header file.

3.4 Execution and output parameters

The main purpose of the library is to solve the HJB - type equations. In addition, it is possible to compute and save some associated functions during the main computation. It is also possible to work with optimal trajectory algorithms, after the computation of the solution. The main computation to solve the PDE is most time and memory consuming part of the work. Once the solution computed and saved one can run the program many times to compute different trajectories from this data without need to make the main loop computation at each time. To choose the execution modes the user can use the following parameters (see below).

All the output files generated during an execution are generaly saved in the folder `OUTPUT/` (the user may use other names for the output files and output directory, see file `include/stdafx.h`, although it is recommanded not to modify the filenames).

- `COMPUTE_MAIN_LOOP`:

  `1` : the main computational loop (iterative scheme) is called

  `0` : the main loop is not called, only data initializations, trajectory computations and savings are performed. This is useful if a previous HJB computation has been made and that only new trajectories have to be computed. There is no need to recompute the value function or the minimal time functions. The program will then load the previous value of minimal time functions in order to perform trajectory reconstructions, or particular savings.

- `COMPUTE_VEX`$\in$`{0,1}`: will save the exact solution on the grid in a file.

  `1` : the solution programmed in the user function `Vex` will be saved.

  `0` : no saving.

  In particular it allows to ocompute errors (see below) relatively to the value of `Vex`.

- `COMPUTE_TOPT`$\in$`{0,1}`: will compute an optimal time function associated with the solution u. Recall that the type of optimal time function is defined by the parameter TOPT_TYPE (see (12) and (13)). The computed function will be saved at the end of the computation.

- PRECOMPUTE_COORD:

  1 : precomputes the coordinates: faster computations but more memory demanding.

  0 : no precomputations.

- CHECK_ERROR $\in$ {0,1}: is set to 1 then computes the error every CHECK_ERROR_STEP steps (L^∞ and L^1 error computations) Errors are relative to the Vex function. Furthermore, the parameter C_THRESHOLD defines the region where the errors will be computed : the region of points x such that |Vex(t,x)|<C_THRESHOLD.

3.5 Parameters for trajectory reconstruction

The dynamics used for the trajectory reconstruction algorithms is the one defined in the function dynamics (in the case COMMANDS=0 or 1) or dynamics2 (in the case COMMANDS=2).

Related reconstruction procedures are in the files src/compute_ot.h, src/compute_otval.h, src/find_oc.h. For two-player games, the reconstruction procedure is programmed in src/compute_ot2.h and find_oc2.h.

The following parameters are used:

- TRAJPT: $\in$ {0,1,...} : set to 1 or greater for trajectory reconstruction, 0 otherwise. When TRAJPT=N, with $N \geq 1$ this defines the number of trajectories to compute for N initial conditions.

- initialpoint[TRAJPT*DIM]: the set of coordinates of all initial points (put the list of all the TRAJPT*DIM coordinates, one point after another, if there is more than one initial point).

- TRAJ_METHOD: $\in$ {0,1}. There are two methods of trajectory reconstruction :

  0: based on the minimal time (if TOPT_TYPE=0), or exit time (if TOPT_TYPE=1)

  1: based on the value function (utilizes VFxx.dat files).

  Furthermore, if COMMANDS=1, then the function dynamics is used for the dynamics (one control). If COMMANDS=2, the function dynamics2 is used (two controls).

- time_TRAJ_START: starting time t_0 used in the case TRAJ_METHOD=1. Can be any value in $[0, T[$. The program will construct an approximated trajectory such that $\dot{y}(t) = f(t, y(t), \alpha(t))$, $t \in [t_0, T]$, and starting with $y(t_0) = x_0$ as initial point. More precisely, if $v(t_n, x)$ is the value function at time t_n, the procedure aims at constructing a control $a = a^n$ (and associated trajectory $y^a_{t_n,x}(t_{n+1})$) that realizes a minimum in

$$\inf_a v(t_{n+1}, y^a_{t_n,x}(t_{n+1})) + h\ell(t_n, y^a_{t_n,x}(t_{n+1}), a) \tag{18}$$

is minimal (resp. maximal) in the case `OPTIM=MAXIMUM` (resp. the case `OPTIM=MINIMUM`).

- If `TRAPJT=0` then one should define accordingly `initialpoint[0]={}`.

Now we describe some stopping criteria which are used in the trajectory reconstruction routines.

- `g_target`: a user-defined function, such that `g_target(x)<=0` if `x` belongs to a "target" set.
- `TARGET_STOP`: $\in$ `{0,1}`: default is `0`. If we set `TARGET_STOP=1`, then the program will furthermore stop the trajectory reconstruction when

  `g_target(x)<=0`

  This option can be useful when the trajectory reconstruction is based on the value function (`TRAJ_METHOD=1`) and that we want to furthermore stop the trajectory reconstruction when a given region is reached.
- `min_TRAJ_STOP` (double): it will stop the trajectory reconstruction when

  `val(x)<=min_TRAJ_STOP`

  and declare this as a successfull trajectory reconstruction (`success=1` in the output file `successTrajectory.dat`). Notice that in the case when `TRAJ_METHOD` is `0`, linked to optimal time reconstruction procedures, the value `val(x)` corresponds to the value of `topt(x)` at point `x`; in the case when `TRAJ_METHOD=1`, `val(x)` corresponds to the value function at the given time of reconstruction and at point `x`.

  In general the minimal time is zero on a target set, and it can be numerically difficult to reach exactly `topt(x)=0`. So this parameter can be set to a small non zero value to allow some margin error to stop the trajectory reconstruction.
- `max_TRAJ_STOP` (double): The trajectory reconstruction is also stopped when

  `val(x)>=max_TRAJ_STOP`

and will declare this as an unsuccesfull trajectory reconstruction (success=0 in the output file successTrajectory.dat). This can be used to detect when val(x)=INF or some large value, showing that we enter a forbidden region and that the trajectory reconstruction should be stopped.

In the case COMMANDS=2 (in the presence of an adverse control u2), it is possible to reconstruct trajectories by setting TRAJ_METHOD=0 and using the following; the function dynamics2 is used to describe the dynamics for the two-control case.

- ADVERSE_CONTROL ∈ {0,1}: default value is 0. It will then find the best strategy of controls u for the worst case situation on the adverse control u2. If ADVERSE_CONTROL=1, then it will use the adverse control u2 as defined by the function u2_adverse.
- u2_adverse(t,x,u2): user-defined function, to define in u2 an adverse control depending on time t and position x.

The following can be used for more outputs concerning trajectory reconstructions:

- PRINT_TRAJ: ∈ {0,1}: default is 0. If set to 1, more outputs on trajectory reconstruction are given in the current window.

3.6 Output parameters and files

Output files are generally contained in the directory OUTPUT/. (It is possible to use other names for the output files and output directory, see file include/stdafx.h, although it is recommanded not to do so.)

3.6.1 Savings options

- FILE_PREFIX[] this character string will be prefixed to all files that will be saved during the execution. By default, this parameter is equal to the empty string. In this case the default file names will be used (see later about all generated files). When using the default file names, each new execution of the program will erase all previously generated data. Hence one way to keep precomputed datas is to use the prefix FILE_PREFIX parameter in order to add it to the output filenames.
- SAVE_VF_ALL ∈ {0,1}: to save the value of u each SAVE_VF_ALL_STEP iterations. The default names of generated files are OUTPUT/VFn.dat where n is the number of file.

- `SAVE_VF_FINAL` $\in$ `{0,1}`: to save the value of u at the last iteration. The default name of the generated file is `OUTPUT/VF.dat`. (Set to `1` in particular if recomputations with different `COUPE` parameters - see below - is used, while the value is not recomputed - `COMPUTE_MAIN_LOOP=0`)
- `SAVE_VF_FINAL_ONSET` $\in$ `{0,1}`. Set to 1 to save the interpolated values of the final value function u on a given, user-defined set of points. The data must be defined in `OUTPUT/X_user.dat` (default name): each line of this file must contain the d coordinates of a point for which one want to compute and save the value function. The generated file is `OUTPUT/XV_user.dat` (default name).
- `SAVE_VALUE_ALL` $\in$ `{0,1}`. Assumes `VALUE_PB=1`. Set to 1 to save the implicit value v at each `SAVE_VALUE_ALL_STEP` iteration. The default names of generated files are `OUTPUT/VALUE_n.dat` where n is the number of file.
- `SAVE_VALUE_FINAL` $\in$ `{0,1}`. Assumes `VALUE_PB=1`. Set to 1 to save the value of the implicit value function v at the last iteration. The default name of the generated file is `OUTPUT/VALUE.dat`.
- The file `OUTPUT/VEX.dat` is saved if `COMPUTE_VEX=1`. It contains the final exact solution value as programmed in `Vex`.
- `COUPE_DIMS[DIM]`, `COUPE_VALS[DIM]`:
  - list of integers n_i (0 or 1) to define the two variables used for the cut.
  - list of values (c_i or 0.) to define the position of the cut. The values c_i are used only when $n_i = 0$.
  - The output is in the file `output.dat`
  - Example : `COUPE_DIMS[DIM]={1,0,1}` and `COUPE_VALS[DIM]={0.,0.5,0.}` defines a cut in the plane $x_2 = 0.5$. The data will be saved as files with default names `OUTPUT/coupe.dat` (and `coupeex.dat` for exact solution) .

3.6.2 Data formatting options

- `FORMAT_FULLDATA` $\in$ `{0,1}`: defines the format of the data files for value functions savings.
  - `FORMAT_FULLDATA=1` The file is structured as follows, on each line (case :

    ```
    i1  i2  ....  id  val
    ```

    where `val` corresponds to the value $u(T, x)$ at mesh point $x = (x_{i_1}, \dots, x_{i_d})$.
  - `FORMAT_FULLDATA=0`, on each line, in the same order, only the value

    ```
    val
    ```

    will be present in the file.

3.6.3 General data files

- `OUTPUT/filePrefix.dat`. This file contains the value of `FILE_PREFIX[]` parameter. It is used also by the visualization routines to retrieve all the other files generated for a given model. **Important remark.** The name of this file does not change if a non empty `FILE_PREFIX[]` is defined. This is the unique file that has always the same name. All the other files are given here with their default names, but are named with the user's prefix as follows:

  ```
  OUTPUT/FILE_PREFIX DefaultName.dat
  ```

- `OUTPUT/data.dat` (default name). This file is saved when the main loop computation is finished. It contains the most important parameters values for the solved problem : the dimension of the state variable and controls, the computational domain, the number of grid points in each direction. Theses data are essential to retrieve the real values of space variables from their integer indexes saved in the formatted value files. This file is used in particular by the plotting functions for matlab/octave given with the library (see the section below).

- `OUTPUT/Dt.dat` (default name). This file contains the value of the time step used for the solution iterations. This value is used by the trajectory reconstruction algorithms.

3.6.4 Files related to trajectory reconstruction

- `OUTPUT/traj-n.dat`: in case `TRAJPT`$\geq$ 1, corresponds to the trajectory number `n`. Each lines has the form

  ```
  x1 x2 ...  xd a1 ...  ap t
  ```

  where `xi` are the coordinates of the point at time `t`, and `aj` the corresponding control parameters.

- `OUTPUT/successTrajectories.dat` Each line `n` of this file has the form

  ```
  x1 x2 ...  xd t success
  ```

  and indicates whether the trajectory number `n` was successful or not (`success=1` or `0`) and shows the corresponding terminal coordinates (`xi`) and time `t` of the last constructed point.

3.7 Advanced parameters

- function `init_data()`: it is empty by default. It is called when the execution starts, before the initialization of all HJB objects. The the user can complete this function if necessary to initialize some model specific parameters. This can be useful for some complex applications.
- The function `post_data()`: this function is empty by default. It is called at the end of the execution, after all standard HJB computations. The user can complete this function if necessary to define some model specific data transformations. This can be useful for some complex applications.

4 Graphic outputs

Matlab/Octave: the file `OUTPUT/output_view.m` can be executed by typing `'output_view'` in a Matlab or Octave user interface (we advise to first set the current Matlab's directory to `OUTPUT/`). For Octave users, it is better to set the parameter `OCTAVE=1` in the file `output_view.m`. Some parameters are given below:

<u>For the first figure:</u>

- `level_set` : a level set value.
- `PLOT_CONTOUR`$\in$`{0,1}` : if set to `1`, will plot contours of the output with `level_set` value.
- `PLOT_REACHABLE`$\in$`{0,1}` : if set to `1`, will plot the 2d set of points where output value is $\leq$ `level_set`).

This plot is based on the output value which is in the file `VF.dat` for `DIM=2`, and a priori in the file `coupe.dat` for `DIM>2`.

<u>For the second figure:</u>

- `PLOT_3d`$\in$`{0,1}`: set this parameter to `1` in order to obtain a 3d plot of the result (or of `coupe.dat` if available).
- `PLOT_TMIN`$\in$`{0,1}`: (default is `0`). Set this parameter to `1` in order to obtain a 3d graph of the minimal time (to reach a target). Will then use output `topt.dat`, and will draw several contour plots ranging from 0 to T (plotting `topt.dat` is only ok for `DIM=2`).

<u>For a third figure:</u>

- PLOT_3d_EX$\in${0,1}: set this parameter to 1 in order to obtain a 3d graph of the exact value (or of coupeex.dat if available).

Also for the figures above,

- AXIS_EQUAL$\in${0,1}, if set to 1, will make "axis equal",
- TRAJECTORY$\in${0,1}, if set to 1, will line-plot the list of first two components (x_1, x_2) of file "trajectory.dat".

Paraview: (obsolete) This option is only working for some 2d/3d cases.
Launch "paraview", then load the files in `VTK/*` (load `tab*` and so on).

5 Examples

5.1 Rotation example

We give two ways to program an advection-like example, in files `data_basicmodel.h` and `data_advancedmodel.h`. We consider the equation

$$\partial_t u + \max(0, -f(x) \cdot \nabla u) = 0.$$

with the parameters $\Omega = [-2, 2]^2$, $T = 0.5$, $f(x_1, x_2) = 2\pi(-x_2, x_1)$. (Hence the Hamiltonian $H(x, p, t) = \max(0, -f(x) \cdot p)$.)

In the first way (`data_basicmodel.h`), we define dynamics

$$f(x, a) = af(x)$$

with two control values $a \in \{0, 1\}$.

In the second way (`data_advancedmodel.h`), we define the Lax-Friedrich numerical hamiltonian H_{num} associated to H (see (21)).

5.2 Eikonal equation

see the files `data_FD_2d_ex1_basic.h` and `data_FD_2d_ex1_advanced.h` The problem solved is

$$\partial_t u + c(x, t)\|\nabla u\| = 0, \quad x \in [-2, 2]^d, \quad t \in [0, T]. \tag{19}$$

$$u(0, x) = u_0(x), \quad x \in [-2, 2]^d, \tag{20}$$

with $d = 2$, $T = 1$, here $c(x, t) \equiv 1$, and with some (radially symetric) initial data $u_0(x)$.

In the file `data_FD_2d_ex1_basic.h`, a scheme is programmed using a control-discretisation of $\|\nabla u\|$ as follows:

$$\|\nabla u\| \sim \max_{k=1,\dots,\mathtt{NCD}} \langle(\cos(\theta_k),\sin(\theta_k),\nabla u\rangle, \quad \theta_k = \frac{2k\pi}{\mathtt{NCD}}$$

where `NCD` is the number of controls.

In the file `data_FD_2d_ex1_advanced.h`, a scheme is programmed using a Lax-Friedriech numerical approximation H_{num} associated to H :

$$H_{num}(x,(p_1^-,p_1^+),\dots,(p_d^-,p_d^+),t) := H(x,\frac{p^-+p^+}{2},t) - \sum_{i=1}^{d} c_i \left(\frac{p_i^+-p_i^-}{2}\right) \tag{21}$$

The exact solution is given for comparison.

6 Second order HJB equations and SL scheme

The problem solved is the following second order Hamilton-Jacobi (HJ) equation

$$\frac{\partial u}{\partial t} + \lambda(x)u + \tag{22a}$$
$$\max_{a\in\mathcal{A}}\Big(-\frac{1}{2}Tr(\sigma(t,x,a)\sigma^T(t,x,a)D^2u) - b(t,x,a)\cdot\nabla u + r(t,x,a)u - \ell(t,x,a)\Big) = 0$$
$$t\in(0,T),\ x\in\mathbb{R}^d,$$
$$u(0,x) = u_0(x), \qquad x\in\mathbb{R}^d \tag{22b}$$

where $\mathcal{A}$ is some non empty compact subset of $\mathbb{R}^m$ ($m \geq 1$), $b(t,x,a)$ is a vector of $\mathbb{R}^d$, $r(t,x,a)$, $\ell(t,x,a)$, are real-valued, and $\sigma(t,x,a)$ is a $d\times p$ real matrix (for some $p \geq 1$). This problem is linked to the computation of the value function of stochastic optimal control problems.

It is also possible to consider a corresponding steady equation of the form (8a) (that is, equation (22a) alone with no term $\frac{\partial u}{\partial t}$).

It is also possible to consider obstacle equations as for (14), (15) or (16).

The `c++` proposed solver is based on an SL method. [2, 3]

[2]For advanced users: it is programmed in function `"secondorder_itSL_evo"` that might be available in the source file `HJB_SL.cpp`, or in `/include/secondorder_itSL_evo.h`.

[3] **Scheme definition:** we consider the following SL scheme, implemented on the points x of the grid $\mathcal{G}$. The initialization is done by

$$v^0(x) = u_0(x), \quad x\in\mathcal{G}.$$

For $n = 0,\dots N_T - 1$ (time iterations) (or untill some stopping criteria is satisfied in the case of

An example of data file is given in `data/data_SL_order2_2d_diffusion.h`.

User inputs:

- `ORDER = 2`
- `PARAMP` (denoted p below) This is in general set to $p = d$ in the scheme.
- function `Sigma`: $\sigma(x, k, a, t)$. A vector of format `double[DIM]`, also denoted σ^k below. This vector should be programmed so as to correspond to the kth column vector of the matrix $\sqrt{p}\,\sigma$, that is $(\sqrt{p}\,\sigma_{ik}(t, x, a))_{i=1,\dots,d}$. More generally, for consistency of the scheme with the PDE, the following relation should hold:
$$\frac{1}{p}\sum_{k=1,\dots,p} \sigma^k(t, x, a)\sigma^k(t, x, a)^T = \sigma(t, x, a)\sigma(t, x, a)^T.$$
- function `Drift`: $b(x, k, a, t)$, also denoted $b^k(t, x, a)$ below. This is used to define a set of p vectors $b^k(t, x, a)$ $(k = 1, \dots, p)$ of $\mathbb{R}^d$. For consistency of the scheme with the PDE, the following relation should hold:
$$\sum_{k=1,\dots,p} b^k(t, x, a) = b(t, x, a).$$
For instance the user can set $b^1(t, x, a) = B(t, x, a)$ and $b^k = 0$, $\forall k \geq 2$. One can also set, with $p = d$: $b^k =$ the k-th component of the b vector, the other components beeing set to zero.
- function `funcR`: $r(t, x, a)$
- function `distributed_cost(x,t,x)` : distributive cost $\ell(t, x, a)$.
- function `discount_factor(x)`: $\lambda(x)$.

steady equations), for all grid points $x \in \mathcal{G}$, we consider

$$v^{n+1}(x) := \min_{a\in\mathcal{A}} \frac{e^{-r(t,x,a)\Delta t}}{1 + \lambda(x)\Delta t}\Bigg(\frac{1}{2p}\sum_{\substack{k=1,\dots,p\\ \epsilon=-1,1}} [v^n](y_x^{k,\epsilon,u}(\Delta t)) + \Delta t\,\ell(t, x, a)\Bigg). \qquad (23)$$

where the "characteristics" $y_x(h)$ can be defined, at iteration $t = t_n$, for some $h \geq 0$, for instance by:

$$y_x^{k,\epsilon,a}(h) := x + B^k(t, x, a)h + \epsilon\Sigma^k(t, x, a)\sqrt{h}, \quad \epsilon \in \{-1, 1\},\ k = 1, \dots, p. \qquad (24)$$

Also, $[v^n](y)$ denotes some interpolation of v^n at point y (typically Q_1). When using the definition (24), the scheme is of expected order $O(\Delta t) + O(\frac{\epsilon}{\Delta t})$ where ϵ is of the order of the interpolation error $\|v^n - [v^n]\|_\infty$. For smooth data, this interpolation error is or order Δx^2 for Q^1 interpolation, where Δx is the spatial mesh size.

7 Source architecture

All the computation algorithms and data management functions are implemented in the following classes :

- `Mesh`. This class defines the data structure for the grid data. It implements all needed functions to access (for reading and writing) the grid data.
- The class `Commands`: This class defines the data structure for the set of controls.
- `HJB`. This class defines the general HJB problem. The class has a first main attribute which is a pointer to a `Mesh` object that contains all the information about the computation domain and its discretization, and a second attribute which is a pointer to a `Commands` object that contains the information about the set of controls. All parameters related to the definition of the problem (dynamics, cost functions, Hamiltonian function) are attributes of this class. It contains some general methods : different interpolation functions, trajectory reconstruction methods and savings management. This class is the base class for two derived classes : `HJB_FD` and `HJB_SL` .
- `HJB_FD`. This class inherits from `HJB` and defines the main loop computations for finite difference methods.
- `HJB_SL`. This class inherits from `HJB` and defines the main loop computations for semi-Lagrangian methods.

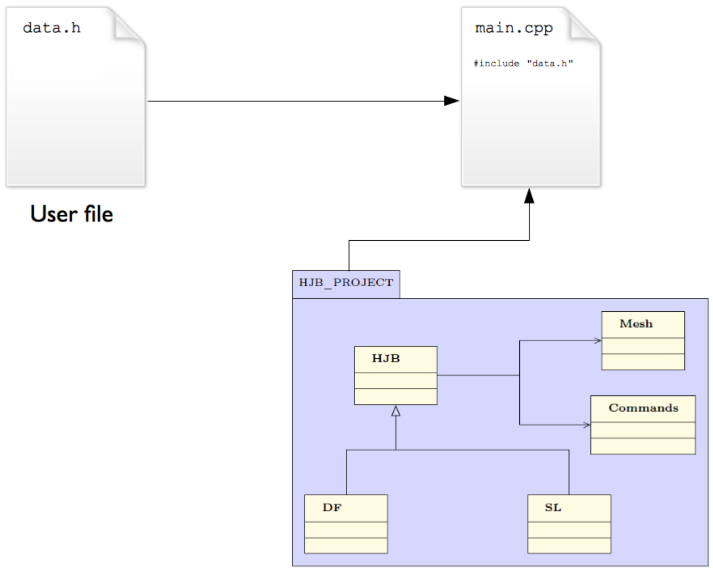
All the classes described above are already compiled in the library `libhj.a`. Their headers are in the folder `include`.

We remind that the user needs to code the problem parameters into a `c++` header file and to include its name in the file `data/data_simulation.h`. For example, if the user data file is called `data_myexample.h`, then the line `#include "data_myexample.h"` (and only this one) must be included in the file `data/data_simulation.h`.

References

[1] A. Altarovici, O. Bokanowski, and H. Zidani. A general Hamilton-Jacobi framework for non-linear state-constrained control problems. *ESAIM: Control, Optimisation and Calculus of Variations*, 19(337–357), 2013 (2012 electronic).

[2] O. Bokanowski, N. Forcadel, and H. Zidani. Reachability and minimal times for state constrained nonlinear problems without any controllability assumption. *SIAM J. Control Optim.*, 48(7):4292–4316, 2010.


data.h
User file
main.cpp
#include "data.h"
HJB_PROJECT
Mesh
HJB
Commands
DF
SL

Index

BORDERSIZE, 13
BOUNDARY, 12
CFL, 16
CHECK_ERROR_STEP, 19
CHECK_ERROR, 19
COMMANDS, 11
COMPUTE_MAIN_LOOP, 18
COMPUTE_TOPT, 18
COMPUTE_VEX, 18
COUPE_DIMS, 22
COUPE_VALS, 22
C_THRESHOLD, 19
DIM, 9
DT, 15
EPSILON, 15
EXTERNAL_v0, 13
FILE_PREFIX, 21
FORMAT_FULLDATA, 22
Hconst, 12
Hnum, 12
MAX_ITERATION, 15
MESH, 15
METHOD, 16
NAME, 9
NCD2, 10
NCD, 10
ND, 15
OPTIM, 12
PERIODIC, 9
PRECOMPUTE_COORD, 19
SAVE_VALUE_ALL_STEP, 22
SAVE_VALUE_ALL, 22
SAVE_VALUE_FINAL, 22
SAVE_VF_ALL_STEP, 21
SAVE_VF_ALL, 21
SAVE_VF_FINAL_ONSET, 22
SAVE_VF_FINAL, 22
TOPT_TYPE, 11
TYPE_RK for finite difference methods, 16
T, 15
UMAX2, 10
UMAX, 10
UMIN2, 10
UMIN, 10
Vex, 11
XMAX, 9
XMIN, 9
cDIM2, 10
cDIM, 9, 10
discount_factor2, 10
discount_factor, 10
distributed_cost2, 10
distributed_cost, 10
dynamics2, 10
dynamics, 10
funcR, 10
funcY, 10
g_bordermix, 13
g_border, 13
v0, 12

class
- Commands, 28
- HJB_FD, 28
- HJB_SL, 28
- HJB, 28
- Mesh, 28

folders
- OUTPUT, 7
- VALUE, 7
- data, 7
- include, 6
- lib, 6
- src, 6

obstacle terms:
- OBSTACLE_TILDE, 14
- OBSTACLE, 14

- `g_obstacle_tilde`, 14
- `g_obstacle`, 14

other functions
- `init_data()`, 24
- `post_data()`, 24

ROC-HJ-Editor, 7

semi-Lagrangian method, 16
- `INTERPOLATION`, 17
- `ORDER`, 17
- `P_INTERMEDIATE`, 17
- `TYPE_RK`, 17
- `TYPE_SCHEME`, 16
- `TYPE_STA_LOOP`, 17

subdomain computations, 5, 13
- `COMPUTE_IN_SUBDOMAIN`, 13
- `g_domain`, 13

trajectory reconstruction
- `TARGET_STOP`, 20
- `g_target`, 20
- `max_TARGET_STOP`, 20
- `min_TARGET_STOP`, 20

value problem, 18