

STochastic OPTimization library in C++ : V7 documentation

Hugo Gevret ¹ Nicolas Langrené ² Jerome Lelong ³
Rafael D. Lobato ⁴ Thomas Ouillon ⁵ Xavier Warin ⁶
Aditya Maheshwari ⁷ Pierre Gruet ⁸

¹EDF R&D, Hugo.Gevret@edf.fr

²CSIRO Data61, Australia, Nicolas.Langrene@csiro.au

³Ensimag, Laboratoire Jean Kuntzmann, 700 avenue Centrale Domaine Universitaire - 38401
St Martin d'Hères

⁴Department of Computer Science, University of Pisa, Italy, Rafael.Lobato@di.unipi.it

⁵EDF R&D, Thomas.Ouillon@edf.fr

⁶EDF R&D & FiME, Laboratoire de Finance des Marchés de l'Energie, ANR PROJECT CAE-
SARS, Xavier.Warin@edf.fr

⁷University of California, Santa Barbara, USA, aditya_maheshwari@umail.ucsb.edu

⁸EDF R&D, Pierre.Gruet@edf.fr

Contents

I	Introduction	7
1	General context	8
2	General mathematical setting	11
II	Useful tools for stochastic control	14
1	The grids and their interpolators	15
1.1	Linear grids	19
1.1.1	Definition and C++ API	19
1.1.2	The Python API	21
1.2	Legendre grids	22
1.2.1	Approximation of a function in 1 dimension	23
1.2.2	Extension in dimension d	26
1.2.3	Truncature	27
1.2.4	The C++ API	28
1.2.5	The Python API	30
1.3	Sparse grids	31
1.3.1	The sparse linear grid method	31
1.3.2	High order sparse grid methods	35
1.3.3	Anisotropy	37
1.3.4	Adaptation	37
1.3.5	C++ API	39
1.3.6	Python API	44
1.4	Full grids with refinement	46
1.4.1	C++ API	47
2	Introducing regression resolution	51
2.1	C++ global API	52
2.2	Adapted local polynomial basis with same probability	58
2.2.1	Description of the method	58
2.2.2	C++ API	59
2.2.3	Python API	61
2.3	Adapted local basis by K-Means clustering methods	62
2.3.1	C++ API	64

2.3.2	Python api	64
2.4	Local polynomial basis with meshes of same size	64
2.4.1	C++ API	64
2.4.2	An example in the linear case	66
2.4.3	Python API	66
2.5	Sparse grid regressor	66
2.5.1	C++ API	67
2.5.2	Python API	67
2.6	Global polynomial basis	68
2.6.1	Description of the method	68
2.6.2	C++ API	69
2.6.3	Python API	69
2.7	Kernel regression with Epanechnikov kernel	70
2.7.1	The univariate case	70
2.7.2	The multivariate case	72
2.7.3	C++ API	75
2.7.4	Python API	76
2.8	Kernel regression with Laplacian kernels	77
2.8.1	Reducing the problem to empirical CDF calculations	77
2.8.2	Fast computation of ECDFs	78
2.8.3	Divide-and-conquer approach	80
2.8.4	C++ API for fast summation method	80
2.8.5	Python API for fast summation method	81
2.8.6	C++ API for exact divid and conquer method	82
2.8.7	Python API for exact divide-and-conquer method	82
3	Calculating conditional expectation by trees	83
3.0.1	C++ API	84
4	Continuation values objects and similar ones	86
4.1	Continuation values objects with regression methods	86
4.1.1	Continuation values object	86
4.1.2	The GridAndRegressedValue object	90
4.1.3	The continuation cut object for concave problems	92
4.1.4	The continuaton cut objet for non concave problems	95
4.2	Continuation objects and associated with trees	99
4.2.1	Continuation object	99
4.2.2	GridTreeValues	100
4.2.3	Continuation Cut with trees	101
III	Solving optimization problems with dynamic programming methods	104
1	Creating simulators	105
1.1	Regression methods simulators	105

1.2	Simulators for trees	107
2	Using conditional expectation to solve simple problems	111
2.1	American option by regression	111
2.1.1	The American option valuing by Longstaff–Schwartz	111
2.2	American options by tree	113
2.2.1	The American option by tree	113
2.2.2	Python API	113
3	Using the general framework to manage stock problems	115
3.1	General requirement for the business object	116
3.2	Solving the problem using conditional expectation calculated by regressions .	118
3.2.1	Requirement to use the framework	118
3.2.2	Classical regression	119
3.2.3	Regressions and cuts for continuous linear transition problems with certain characteristics of concavity and convexity	131
3.2.4	Regressions and cuts for local concave/convex problems	135
3.3	Solve the problem for $X_2^{x,t}$ stochastic	142
3.3.1	Requirement to use the framework	142
3.3.2	The framework in optimization	144
3.3.3	The simulation framework	147
3.4	Solving stock problems with trees	147
3.4.1	Resolution of dynamic programming problems with the discretization of commands	147
3.4.2	Solving Dynamic Programming by solving LP problems	153
3.5	The Python API to manage stocks	160
3.5.1	Mapping to the framework	160
3.6	Special Python binding	164
3.6.1	A first binding to use the framework	164
3.6.2	Binding to store/read a regressor and a two-dimensional array	167
4	Framework to solve some switching problems with integer state constraints by regressions	169
4.1	Business object for switching problems	170
4.2	Using the framework to optimize or simulate on a time step	172
4.2.1	Optimize on one time step	172
4.2.2	Regressions and cuts for continuous linear transition problems with certain characteristics of concavity and convexity	182
4.3	Solve the problem for $X_2^{x,t}$ stochastic	189
4.3.1	Requirement to use the framework	189
4.3.2	The framework in optimization	191
4.3.3	The simulation framework	194
4.4	Solving stock problems with trees	194
4.4.1	Resolution of dynamic programming problems with the discretization of commands	195
4.4.2	Solving Dynamic Programming by solving LP problems	200

5	Using the C++ framework to solve some hedging problem	208
5.1	The problem	208
5.2	Theoretical algorithm	209
5.3	Practical algorithm based on Algorithm 10	212
6	Managing stock with dynamic programming where the transition problem is itself the result of a deterministic optimization using dynamic programming	215
6.1	General requirement for the simulator	216
6.2	General requirement for the business object	217
6.3	Use the framework in optimization	219
6.4	Use the framework in simulation	222
7	Calculating Sensitivities for Stochastic Problems	226
IV	Semi-Lagrangian methods	230
1	Theoretical background	232
1.1	Notation and regularity results	232
1.2	Temporal discretization for the HJB equation	233
1.3	Space interpolation	233
2	C++ API	235
2.1	PDE resolution	241
2.2	Simulation framework	243
V	An example with both dynamic programming with regression and PDE	250
0.3	Dynamic programming with the regression approach	252
0.4	The PDE approach	255
VI	Stochastic Dual Dynamic Programming	258
1	SDDP algorithm	259
1.1	Some general points about SDDP	259
1.2	A method, different algorithms	261
1.2.1	The basic case	261
1.2.2	Dependence of random quantities	263
1.2.3	Non-convexity and conditional cuts	265
1.3	C++ API	271
1.3.1	Inputs	271
1.3.2	Architecture	276
1.3.3	Implement your problem	276

1.3.4	Set of parameters	281
1.3.5	The black box	283
1.3.6	Outputs	283
1.4	Python API (only for regression-based methods)	284

VII Nesting Monte Carlo for general nonlinear PDEs 289

VIII Some test cases description 295

0.5	American option	296
0.5.1	testAmerican	296
0.5.2	testAmericanConvex	298
0.5.3	testAmericanForSparse	298
0.5.4	testAmericanOptionCorrel	299
0.5.5	testAmericanOptionTree	299
0.6	testSwingOption	299
0.6.1	testSwingOption2D	300
0.6.2	testSwingOption3	300
0.6.3	testSwingOptimSimu / testSwingOptimSimuMpi	300
0.6.4	testSwingOptimSimuWithHedge	301
0.6.5	testSwingOptimSimuND / testSwingOptimSimuNDMpi	301
0.7	Gas Storage	302
0.7.1	testGasStorage / testGasStorageMpi	302
0.7.2	testGasStorageMultiStage / testGasStorageMultiStageMpi	303
0.7.3	testGasStorageCut / testGasStorageCutMpi	303
0.7.4	testGasStorageCutGridAdapt1D / testGasStorageCutGridAdapt2D	304
0.7.5	testGasStorageTree/testGasStorageTreeMpi	304
0.7.6	testGasStorageTreeCut/testGasStorageTreeCutMpi	304
0.7.7	testGasStorageKernel	304
0.7.8	testGasStorageLaplacianLinearKernel	304
0.7.9	testGasStorageLaplacianConstKernel	305
0.7.10	testGasStorageLaplacianGridKernel	305
0.7.11	testGasStorageVaryingCavity	305
0.7.12	testGasStorageSwitchingCostMpi	306
0.7.13	testGasStorageSDDP	306
0.7.14	testGasStorageSDDPTree	307
0.8	testLake / testLakeMpi	307
0.9	testOptionNIGL2	307
0.10	testDemandSDDP	308
0.11	testThermalAsset	308
0.12	Reservoir variations with SDDP	308

0.12.1	testReservoirWithInflowsSDDP	308
0.12.2	testStorageWithInflowsSDDP	309
0.12.3	testStorageWithInflowsAndMarketSDDP	310
0.13	Semi-Lagrangian	311
0.13.1	testSemiLagrangCase1/testSemiLagrangCase1	311
0.13.2	testSemiLagrangCase2/testSemiLagrangCase2	311
0.13.3	testSemiLagrangCase2/testSemiLagrangCase2	312
0.14	Non emissive test case	312
0.14.1	testDPNonEmissive	312
0.14.2	testSLNonEmissive	312
0.15	Nesting for Non Linear PDE's	313
0.15.1	Some HJB test	313
0.15.2	Some Toy example: testUD2UTou	313
0.15.3	Some Portfolio optimization	314
0.16	Automatic Differentiation	315
0.16.1	testStorageAutoDiff	315
1	Some Python test cases description	330
1.1	Microgrid Management	330
1.1.1	testMicrogridBangBang	330
1.1.2	testMicrogrid	330
1.2	Dynamic Emulation Algorithm (DEA)	331
1.2.1	testMicrogridDEA	331

Part I

Introduction

Chapter 1

General context

Optimizing while dealing with uncertainties is a shared goal by many sectors in the industry. For example in the banking system:

- Some options such as American options need, in order to be valuated, to find an optimal exercise strategy to maximize the gain on average.
- When dealing with assets management, a fund manager may want to find a strategy to optimize his gains by investing in different assets while trying to satisfy some risk constraints.
- When dealing with credit risk in the case of option selling, some CVA modelization necessitates to solve some high dimensional problem in order to evaluate the option value.

In the energy financial sector, many problems involve stochastic optimization:

- some options, known as swing options, permit the owner to get some energy at some chosen dates with constraints on volumes. The price paid is either deterministic such as in the electricity market or can be an index which is an average of some commodity prices such as in the gas market.
- When some batteries are installed on a network, the battery has to be filled in or discharged optimally in order to avoid the use of some expensive thermal units.
- The optimal management of certain gas storage or certain thermal assets taking into account the prices of raw materials is a goal shared by all asset owners in the sector.
- Even in regulated energy market, when water is used to generate electricity, a common target consists in finding an optimal water management in order to maximize the profit on average.

A goal shared by many industries is the problem of risk management: which financial assets to buy to guarantee a given gain by immunizing a financial portfolio against certain uncertainties.

All these problems and many more require:

- either to resolve certain PDEs when the control must be evaluated continuously,
- or to calculate a certain conditional expectation in the event that control must be taken on certain discrete dates. The problem is then solved by a dynamic programming method.

The STochastic OPTimization library (StOpt)

<https://gitlab.com/stochastic-control/StOpt>

aims to provide tools to solve certain stochastic optimization problems encountered in finance or in industry. This library is a toolbox used to facilitate the work of developers wishing to solve certain stochastic optimization problems by providing a general framework and some objects commonly used in stochastic programming. Many effective methods are implemented and the toolkit must be flexible enough to use the library at different levels being either an expert or only wishing to use the general framework.

The Python interface allows you to use the library at a low level. The test cases are either in C++ , or in Python or in the both language.

The user is invited to consult the different test cases proposed in order to have global view of the resolution methods. All the test cases are described in the last section of the documentation and deal with problems encountered in the banking system or the energy sector.

- The American options are solved by a part of dynamic programming III in Python or C++ using regression (section 2) or using a scenario tree 3.

Regression are achieved:

1. either by local polynomials or with base support of the same size (subsection 2.4) or with an adapted size of the support (subsection 2.2) ,
2. either by global polynomials (section 2.6)
3. either by sparse grid regression (section 2.5) useful in high dimension
4. or by kernel regression (section 2.7)

In the test, a trinomial tree is developed as an example and the valorisation of an American option for the Black-Scholes model is given using this tree.

- Gas storage problems are solved
 - either by dynamic programming (part III) in Python or C++ using regression (section 2) or tree (section 3) and stock interpolation (chapter 1).

Regression are achieved:

1. either by Local polynomials with an adapted size of the support (subsection 2.2) ,
2. either by global polynomials (section 2.6)
3. or kernel regression (section 2.7)

As before the trinomial tree developed in tests is used in the tree methods.

The interpolation between stock points is either linear or quadratic.

- either by the SDDP method (chapter 1) in C++ using both regression and tree methods.
- The swing options are solved by dynamic programming (part III) in Python or C++ using regression with local polynomials with a suitable size of the support (subsection 2.2)
- The optimal management of a lake with stochastic inflows is solved by dynamic programming (part III) in Python or C++ using local polynomials with an adapted support size (subsection 2.2)
- The optimal hedging of an option using a average variance criterion of the hedged portfolio is resolved in C++ by dynamic programming (part III) using the methodology in chapter 5.
- A certain management of the reservoirs is solved by the SDDP method (chapter 1) in C++ trying to minimize the cost of the energy supply to satisfy a given demand with the possibility of buying energy at a price that can be stochastic.
- The continuous optimization of a portfolio made up of a few assets following a Heston model is achieved by solving C++ the corresponding PDE with the Monte Carlo nesting method (part VII).
- Some microgrid problems in the energy sector are solved using the Python interface by dynamic programming methods (part III) using grids with linear interpolation (subsection 1.1.1) to discretize the energy level in the battery and the different regressors using:
 1. either local polynomials with an suitable support size (subsection 2.2) ,
 2. either global polynomials (section 2.6)
 3. or kernel regression (section 2.7)

Chapter 2

General mathematical setting

In a continuous frame, the controlled state is given by a stochastic differential equation

$$\begin{cases} dX_s^{x,t} &= b_a(t, X_s^{x,t})ds + \sigma_a(s, X_s^{x,t})dW_s \\ X_t^{x,t} &= x \end{cases}$$

where

- W_t is a d -dimensional Brownian motion on a probability space $(\Omega, \mathcal{F}, \mathbb{P})$ endowed with the natural (complete and continuous line) filtration $\mathbb{F} = (\mathcal{F}_t)_{t \leq T}$ generated by W up to a fixed time horizon $T > 0$,
- σ_a is a Lipschitz continuous function of (t, x, a) defined on $[0, T] \times \mathbb{R}^d \times \mathbb{R}^n$ and taking values in the set of d -dimensional square matrices,
- b_a is a Lipschitz continuous function of (t, x, a) defined on $[0, T] \times \mathbb{R}^d \times \mathbb{R}^n$ and taking values in $\mathbb{R}^d$,
- a a control adapted to the filtration taking values in $\mathbb{R}^n$.

Suppose we want to minimize a cost function $J(t, x, a) = \mathbb{E}[\int_t^T f_a(s, X_s^{x,t})e^{\int_t^s c_a(u, X_u^{x,t})du}ds + e^{\int_t^T c_a(u, X_u^{x,t})}g(X_T^{x,t})]$ compared to the a control. It is well known [16] that the optimal value $\hat{J}(t, x) = \inf_a J(T - t, x, a)$ is a viscosity solution of the equation

$$\begin{aligned} \frac{\partial v}{\partial t}(t, x) - \inf_{a \in A} \left(\frac{1}{2} \text{tr}(\sigma_a(t, x)\sigma_a(t, x)^T D^2 v(t, x)) + b_a(t, x) Dv(t, x) \right. \\ \left. + c_a(t, x)v(t, x) + f_a(t, x) \right) &= 0 \text{ in } \mathbb{R}^d \\ v(0, x) &= g(x) \text{ in } \mathbb{R}^d \end{aligned} \tag{2.1}$$

According to certain classical hypotheses on the coefficients [16], the previous equation known as the Hamilton Jacobi Bellman equation admits a solution of unique viscosity ([25]). Solving the previous equation is quite difficult, especially in dimensions larger than 3 or 4. The library provides tools to solve this equation and simplified versions of it.

- a first method supposes that $X_s^{x,t} = (X_{1,s}^{x,t}, X_{2,s}^{x,t})$ where $X_{1,s}^{x,t}$ is not controlled

$$\begin{cases} dX_{1,s}^{x,t} &= b(t, X_{1,s}^{x,t})ds + \sigma(s, X_{1,s}^{x,t})dW_s \\ X_{1,t}^{x,t} &= x \end{cases} \quad (2.2)$$

and $X_{2,s}^{x,t}$ has no diffusion term

$$\begin{cases} dX_{2,s}^{x,t} &= b_a(t, X_{2,s}^{x,t})ds \\ X_{2,t}^{x,t} &= x \end{cases}$$

In this case, we can use Monte Carlo methods based on regression to solve the problem. The method is based on the principle of dynamic programming and can be used even if the uncontrolled SDE is controlled by a general Levy process. This method can be used even if the controlled state takes only a few discrete values.

A second approach based on dynamic programming uses scenario trees: in this case, the uncertainties evolve on a tree taking only discrete values.

- The second case is a special case of the previous one when the problem to be solved is linear and the controlled state takes values at continuous intervals. The value function must be convex or concave with respect to the controlled variables. This method, the SDDP method, is used when the dimension of the controlled state is large, which prevents the use of the dynamic programming method. As before, the uncertainties can be described either by scenarios or by a scenario tree.

Remark 1 *The use of this method requires other assumptions which will be described in the dedicated chapter.*

- A third method solves the Monte Carlo problem when a process is controlled but by an uncontrolled process. this is generally the optimization of a portfolio:
 - The value of the portfolio is deterministically controlled and discretized on a network,
 - The evolution of the portfolio is driven by an uncontrolled exogenous process: market prices.
- In the fourth method, we will assume that the state takes continuous values, we will solve equation (2.1) using semi-Lagrangian methods discretizing the Brownian motion with two values and using some interpolations on grids.
- Finally, we present a general pure Monte Carlo method based on automatic differentiation and randomization of the time step to solve general non-linear equations and which can be used to solve certain control problems.

In what follows, we assume that a temporal discretization is given for the resolution of the optimization problem. We assume that the step discretization is constant and equal to h such that $t_i = ih$. First, we describe some useful tools developed in the library for stochastic control. Then, we explain how to solve certain optimization problems using these developed tools.

Remark 2 *In the library, we rely a lot on the Eigen library: **ArrayXd** which represents a double vector, **ArrayXXd** for a double matrix and **ArrayXi** for a vector of integer.*

Part II

Useful tools for stochastic control

Chapter 1

The grids and their interpolators

In this chapter we develop the tools used to interpolate a discretized function on a given grid. A grid is a set of points in $\mathbb{R}^d$ defining meshes which can be used to interpolate a function on an open set in $\mathbb{R}^d$. These tools are used to interpolate a given function, for example at certain stock points, when it comes to storage. They are also useful for semi-Lagrangian methods, which require efficient interpolation methods. In StOpt four types of grids are currently available:

- the first and second are grids used to interpolate a function linearly on a grid;
- the third type of grid, starting from a regular grid, makes it possible to interpolate on a grid at Gauss - Lobatto points on each mesh;
- the last grid allows to interpolate a function in high dimension using the sparse grid-method. The approximation is linear, quadratic, or cubic in each direction.

Each type of grid is associated with iterators. An iterator on a grid makes it possible to iterate on all the points of the grids. All iterators derive from abstract class `GridIterator`

```
1 // Copyright (C) 2016 EDF
2 // All Rights Reserved
3 // This code is published under the GNU Lesser General Public License (GNU LGPL)
4 #ifndef GRIDITERATOR_H
5 #define GRIDITERATOR_H
6 #include <Eigen/Dense>
7
8 /** \file GridIterator.h
9  * \brief Defines an iterator on the points of a grid
10  * \author Xavier Warin
11  */
12 namespace StOpt
13 {
14
15 /// \class GridIterator GridIterator.h
16 /// Iterator on a given grid
17 class GridIterator
18 {
19
20 public :
21
22     /// \brief Constructor
23     GridIterator() {}
24
25     /// \brief Destructor
```

```

26     virtual ~GridIterator() {}
27
28     /// \brief get current coordinates
29     virtual Eigen::ArrayXd getCoordinate() const = 0 ;
30
31     /// \brief Check if the iterator is valid
32     virtual bool isValid(void) const = 0;
33
34     /// \brief iterate on point
35     virtual void next() = 0;
36
37     /// \brief iterate jumping some point
38     /// \param p_incr increment in the jump
39     virtual void nextInc(const int &p_incr) = 0;
40
41     /// \brief get counter : the integer associated the current point
42     virtual int getCount() const = 0;
43
44     /// \brief Permits to jump to a given place given the number of processors (permits to
45     use MPI and openmp)
46     /// \param p_rank processor rank
47     /// \param p_nbProc number of processor
48     /// \param p_jump increment jump for iterator
49     virtual void jumpToAndInc(const int &p_rank, const int &p_nbProc, const int &p_jump) =
50     0;
51
52     /// \brief return relative position
53     virtual int getRelativePosition() const = 0 ;
54
55     /// \brief return number of points treated
56     virtual int getNbPointRelative() const = 0 ;
57
58     /// \brief Reset the interpolator
59     virtual void reset() = 0 ;
60 };
61 #endif /* GRIDITERATOR_H */

```

All iterators share some common characteristics:

- the `getCount` method provides the number associated with the current grid point,
- the `next` method allows you to go to the next point, while the `nextInc` method allows you to go to the point `p_incr`,
- the `isValid` method checks that we are still on a grid point,
- the `getNbPointRelative` allows to get the number of points on which a given iterator can iterate,
- the `getRelativePosition` retrieves the number of points already iterated by the iterator.

In addition, we can go directly to a given point: this functionality is useful for "mpi" when some calculations on the grid are distributed on some processors and threads. This possibility is given by the method `jumpToAndInc`.

Using a grid `regGrid` the following source code makes it possible to iterate over the points of the grids and obtain coordinates. For each coordinate, a function f is used to fill an array of values. As mentioned earlier, each type of grid has its own grid iterator which can be obtained by the `getGridIterator` method.

```

1   ArrayXd data(regGrid.getNbPoints()); // create an array to store the values of the
    function f
2   shared_ptr<GridIterator> iterRegGrid = regGrid.getGridIterator();
3   while (iterRegGrid->isValid())
4   {
5       ArrayXd pointCoord = iterRegGrid->getCoordinate(); // store the coordinates of the
        point
6       data(iterRegGrid->getCount()) = f(pointCoord); // the value is stored in data at
        place iterRegGrid->getCount()
7       iterRegGrid->next(); // go to next point
8   }

```

It is also possible to “skip” certain points and repeat for the “p” points afterwards. This possibility is useful for multithreaded tasks on points.

For each type of grid, an interpolator is provided to interpolate a given function on a grid. Note that the interpolator is created **for a given point** where we want to interpolate. All interpolators (which are not spectral interpolators) derive from `Interpolator` whose source code is given below. Notice that the interpolators are templated such that interpolation can be achieved for point of different types.

```

1 // Copyright (C) 2016 EDF
2 // All Rights Reserved
3 // This code is published under the GNU Lesser General Public License (GNU LGPL)
4 #ifndef INTERPOLATOR_H
5 #define INTERPOLATOR_H
6 #include <vector>
7 #include <Eigen/Dense>
8 /** \file Interpolator.h
9  * \brief Defines a interpolator on a full grid
10  * \author Xavier Warin
11  */
12 namespace StOpt
13 {
14
15 /// \class Interpolator Interpolator.h
16 /// Interpolation base class
17 template<typename T>
18 class Interpolator
19 {
20 public :
21
22     /// \brief Default constructor
23     Interpolator() {}
24
25     /// \brief Default Destructor
26     virtual ~Interpolator() {}
27
28     /** \brief interpolate
29      * \param p_dataValues Values of the data on the grid
30      * \return interpolated value
31      */
32     virtual T apply(const Eigen::Ref< const Eigen::Array<T, Eigen::Dynamic, 1> > &
        p_dataValues) const = 0;
33
34     /** \brief interpolate and use vectorization
35      * \param p_dataValues Values of the data on the grid. Interpolation is achieved for
        all values in the first dimension
36      * \return interpolated value
37      */
38     virtual Eigen::Array<T, Eigen::Dynamic, 1> applyVec(const Eigen::Array<T, Eigen::
        Dynamic, Eigen::Dynamic> &p_dataValues) const = 0;
39
40     /** \brief Same as above but avoids copy for Numpy eigen mapping due to storage
        conventions
41      * \param p_dataValues Values of the data on the grid. Interpolation is achieved

```

```

    for all values in the first dimension
42 * \return interpolated value
43 */
44 virtual Eigen::ArrayXd applyVecPy(Eigen::Ref< Eigen::ArrayXXd, 0, Eigen::Stride<Eigen
    ::Dynamic, Eigen::Dynamic> > p_dataValues) const = 0;
45
46 };
47 }
48 #endif

```

All interpolators provide a constructor specifying the point where the interpolation is carried out and the two functions `apply` and `applyVec` interpolating either a function (and returning a value) or an array of functions returning an array of interpolated values.

All grid classes derive from an abstract class `SpaceGrid` below allowing to retrieve an iterator associated with the points of the grid (with possible jumps) and to create an interpolator associated with the grid.

```

1 // Copyright (C) 2016 EDF
2 // All Rights Reserved
3 // This code is published under the GNU Lesser General Public License (GNU LGPL)
4 #ifndef SPACEGRID_H
5 #define SPACEGRID_H
6 #include <array>
7 #include <memory>
8 #include <Eigen/Dense>
9 #include "StOpt/core/grids/GridIterator.h"
10 #include "StOpt/core/grids/Interpolator.h"
11 #include "StOpt/core/grids/InterpolatorSpectral.h"
12
13 /** \file SpaceGrid.h
14 * \brief Defines a base class for all the grids
15 * \author Xavier Warin
16 */
17 namespace StOpt
18 {
19
20 /// \class SpaceGrid SpaceGrid.h
21 /// Defines a base class for grids
22 class SpaceGrid
23 {
24 public :
25     /// \brief Default constructor
26     SpaceGrid() {}
27
28     /// \brief Default destructor
29     virtual ~SpaceGrid() {}
30
31     /// \brief Number of points of the grid
32     virtual size_t getNbPoints() const = 0;
33
34     /// \brief get back iterator associated to the grid
35     virtual std::shared_ptr< GridIterator> getGridIterator() const = 0;
36
37     /// \brief get back iterator associated to the grid (multi thread)
38     virtual std::shared_ptr< GridIterator> getGridIteratorInc(const int &p_iThread) const =
        0;
39
40     /// \brief Get back interpolator at a point Interpolate on the grid
41     /// \param p_coord coordinate of the point for interpolation
42     /// \return interpolator at the point coordinates on the grid
43     virtual std::shared_ptr<Interpolator<double>> createInterpolator(const Eigen::ArrayXd &
        p_coord) const = 0;
44
45
46

```

```

47  /// \brief Get back a spectral operator associated to a whole function
48  /// \param p_values   Function value at the grids points
49  /// \return  the whole interpolated value function
50  virtual std::shared_ptr<InterpolatorSpectral<double>> createInterpolatorSpectral(const
      Eigen::ArrayXd &p_values) const = 0;
51
52
53  /// \brief Dimension of the grid
54  virtual int getDimension() const = 0 ;
55
56  /// \brief get back bounds associated to the grid
57  /// \return in each dimension give the extreme values (min, max) of the domain
58  virtual std::vector<std::array< double, 2> > getExtremeValues() const = 0;
59
60  /// \brief test if the point is strictly inside the domain
61  /// \param  p_point point to test
62  /// \return true if the point is strictly inside the open domain
63  virtual bool isStrictlyInside(const Eigen::ArrayXd &p_point) const = 0 ;
64
65  /// \brief test if a point is inside the grid (boundary include)
66  /// \param  p_point point to test
67  /// \return true if the point is inside the open domain
68  virtual bool isInside(const Eigen::ArrayXd &p_point) const = 0 ;
69
70  /// \brief truncate a point so that it stays inside the domain
71  /// \param p_point point to truncate
72  virtual void truncatePoint(Eigen::ArrayXd &p_point) const = 0 ;
73
74 };
75 }
76 #endif /* SPACEGRID.H */

```

All the grids objects, interpolators and iterators on grids point are in

StOpt/core/grids

Grid objects are mapped to Python, giving the possibility of retrieving the iterators and interpolators associated with a grid. Examples of Python can be found in

test/Python/unit/grids

1.1 Linear grids

1.1.1 Definition and C++ API

Two kinds of grids are developed:

- the first one is the `GeneralSpaceGrid` with Constructor

```

1 GeneralSpaceGrid(const std::vector<shared_ptr<Eigen::ArrayXd> > &p_meshPerDimension
    )

```

where `std::vector<shared_ptr<Eigen::ArrayXd>>` is a vector of (pointer to) arrays defining the grid points in each dimension. In this case the grid is not regular and the mesh varies in space (see figure 1.1).

- the second one is the `RegularSpaceGrid` with Constructor

```

1 RegularSpaceGrid(const Eigen::ArrayXd &p_lowValues, const Eigen::ArrayXd &p_step,
    const Eigen::ArrayXi &p_nbStep)

```

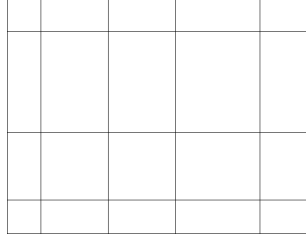


Figure 1.1: 2D general grid

The `p_lowValues` corresponds to the bottom of the grid, `p_step` the size of each mesh, `p_nbStep` the number of steps in each direction (see figure 1.2)

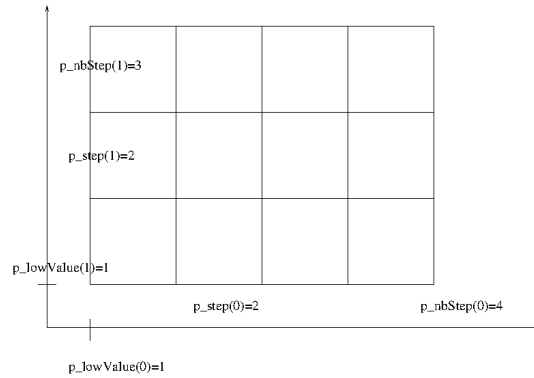


Figure 1.2: 2D regular grid

For each grid, a linear interpolator can be generated by calling the `createInterpolator` method or by directly creating the interpolator:

```
1  /** \brief Constructor
2   * \param p_grid is the grid used to interpolate
3   * \param p_point is the coordinates of the points used for interpolation
4   */
5  LinearInterpolator( const FullGrid * p_grid , const Eigen::Array<T, Eigen::Dynamic, 1> &
   p_point):
```

Its construction from a grid (`regLin`) and from an array `data` containing the values of the function at the points of the grid is given below (taking an example above for fill in the table `data`)

```
1  ArrayXd data(regGrid.getNbPoints()); // create an array to store the values of the
   function f
2  shared_ptr<GridIterator> iterRegGrid = regGrid.getGridIterator();
3  while (iterRegGrid->isValid())
4  {
5      ArrayXd pointCoord = iterRegGrid->getCoordinate(); // store the coordinate of the
   point
6      data(iterRegGrid->getCount()) = f(pointCoord); // the value is stored in the data
   at location iterRegGrid->getCount()
7      iterRegGrid->next(); // go to next point
8  }
9  // point where to interpolate
10 ArrayXd point = ArrayXd::Constant(nDim, 1. / 3.);
11 // create the interpolator
12 LinearInterpolator<double> regLin(&regGrid, point);
```

```

13 // get back the interpolated value
14 double interpReg = regLin.apply(data);

```

Let $I_{1,\Delta x}$ be the linear interpolator where the mesh is $\Delta x = (\Delta x^1, \dots, \Delta x^d)$. We obtain for a function f in $C^{k+1}(\mathbb{R}^d)$ with $k \leq 1$

$$\|f - I_{1,\Delta x}f\|_\infty \leq c \sum_{i=1}^d \Delta x_i^{k+1} \sup_{x \in [-1,1]^d} \left| \frac{\partial^{k+1} f}{\partial x_i^{k+1}} \right| \quad (1.1)$$

In particular if f is only Lipschitz

$$\|f - I_{1,\Delta x}f\|_\infty \leq K \sup_i \Delta x_i.$$

1.1.2 The Python API

The Python API allows you to use grids with a syntax similar to the C++ API. Here, we give an example with a regular grid

```

1 # Copyright (C) 2016 EDF
2 # All Rights Reserved
3 # This code is published under the GNU Lesser General Public License (GNU LGPL)
4 import numpy as np
5 import unittest
6 import random
7 import math
8 import pystopt.grids as StOptGrids
9
10 # unit test for regular grids
11 #####
12
13 class testGrids(unittest.TestCase):
14
15     # 3 dimensional test for linear interpolation on regular grids
16     def testRegularGrids(self):
17         # low value for the meshes
18         lowValues = np.array([1.,2.,3.], dtype=float)
19         # size of the meshes
20         step = np.array([0.7,2.3,1.9], dtype=float)
21         # number of steps
22         nbStep = np.array([4,5,6], dtype=np.int32)
23         # create the regular grid
24         grid = StOptGrids.RegularSpaceGrid(lowValues, step, nbStep)
25         iterGrid = grid.getGridIterator()
26         # array to store
27         data = np.empty(grid.getNbPoints())
28         # iterates on points and store values
29         while( iterGrid.isValid()):
30             #get coordinates of the point
31             pointCoord = iterGrid.getCoordinate()
32             data[iterGrid.getCount()] = math.log(1. + pointCoord.sum())
33             iterGrid.next()
34         # get back an interpolator
35         ptInterp = np.array([2.3,3.2,5.9], dtype=float)
36         interpol = grid.createInterpolator(ptInterp)
37         # calculate interpolated value
38         interpValue = interpol.apply(data)
39         print(("Interpolated value" , interpValue))
40         # test grids function
41         iDim = grid.getDimension()
42         pt = grid.getExtremeValues()
43
44 if __name__ == '__main__':
45     unittest.main()

```

A similar example can be given for a general grid with linear interpolation

```

1 # Copyright (C) 2017 EDF
2 # All Rights Reserved
3 # This code is published under the GNU Lesser General Public License (GNU LGPL)
4 import numpy as np
5 import unittest
6 import random
7 import math
8 import pystopt.grids as StOptGrids
9
10 # unit test for general grids
11 #####
12
13 class testGrids(unittest.TestCase):
14
15     # test general grids
16     def testGeneralGrids(self):
17         # low value for the mesh
18         lowValues = np.array([1.,2.,3.], dtype=float)
19         # size of the mesh
20         step = np.array([0.7,2.3,1.9], dtype=float)
21         # number of step
22         nbStep = np.array([4,5,6], dtype=np.int32)
23         # degree of the polynomial in each direction
24         degree = np.array([2,1,3], dtype=np.int32)
25
26         # list of mesh
27         mesh1= np.array([1. + 0.7*i for i in np.arange(5)] ,dtype=float)
28         mesh2= np.array([2.+2.3*i for i in np.arange(6)], dtype=float)
29         mesh3= np.array([3.+1.9*i for i in np.arange(7)], dtype=float)
30
31         # create the general grid
32         grid = StOptGrids.GeneralSpaceGrid([mesh1,mesh2,mesh3] )
33
34         iterGrid = grid.getGridIterator()
35         # array to store
36         data = np.empty(grid.getNbPoints())
37         # iterates on point
38         while( iterGrid.isValid()):
39             #get coordinates of the point
40             pointCoord = iterGrid.getCoordinate()
41             data[iterGrid.getCount()] = math.log(1. + pointCoord.sum())
42             iterGrid.next()
43         # get back an interpolator
44         ptInterp = np.array([2.3,3.2,5.9], dtype=float)
45         interpol = grid.createInterpolator(ptInterp)
46         # calculate interpolated value
47         interpValue = interpol.apply(data)
48         # test grids function
49         iDim = grid.getDimension()
50         pt = grid.getExtremeValues()
51
52
53
54 if __name__ == '__main__':
55     unittest.main()

```

1.2 Legendre grids

With linear interpolation, to obtain a precise solution, it is necessary to refine the mesh so that Δx goes to zero. Another approach consists in trying to fit a polynomial on each mesh using a high degree interpolator.

1.2.1 Approximation of a function in 1 dimension

From now on, by resizing we assume that we want to interpolate a function f on $[-1, 1]$. All the following results can be extended by tensorization in dimension greater than 1. P_N is the set of polynomials of total degree less than or equal to N . The minmax approximation of f of degree N is the polynomial $P_N^*(f)$ such that:

$$\|f - P_N^*(f)\|_\infty = \min_{p \in P_N} \|f - p\|_\infty$$

We call I_N^X interpolator from f on a grid of $N + 1$ points of $[-1, 1]$ $X = (x_0, \dots, x_N)$, the unique polynomial of degree N such that

$$I_N^X(f)(x_i) = f(x_i), 0 \leq i \leq N$$

This polynomial can be expressed in terms of the Lagrange polynomial $l_i^X, 0 \leq i \leq N$ associated with the grid (l_i^X is the unique polynomial of degree N taking a value equal to 1 at the point i and 0 at the other interpolation points).

$$I_N^X(f)(x) = \sum_{i=0}^N f(x_i) l_i^X(x)$$

The interpolation error can be expressed in terms of interpolation points:

$$\|I_N^X(f)(x) - f\|_\infty \leq (1 + \lambda_N(X)) \|f - P_N^*(f)\|_\infty$$

where $\lambda_N(X)$ is the Lebesgue constant associated with the Lagrange quadrature on the grid:

$$\lambda_N(X) = \max_{x \in [-1, 1]} \sum_{i=0}^N |l_i^X(x)|.$$

We have the following bound

$$\|I_N^X(f)(x)\|_\infty \leq \lambda_N(X) \sup_{x_i \in X} |f(x_i)| \leq \lambda_N(X) \|f\|_\infty$$

and the Erdős theorem states that

$$\lambda_N(X) > \frac{2}{\pi} \log(N + 1) - C$$

It is well-known that the use of a uniform grid X_u is not optimal, because like $N \rightarrow \infty$, the Lebesgue constant satisfies

$$\lambda_N(X_u) \simeq \frac{2^{N+1}}{eN \ln N}$$

and the quadrature error in L_∞ increases a lot with N . Its use brings some oscillations giving the Runge effect. On Figures 1.3a, 1.3b, 1.3c, 1.3d, we trace the function Runge $\frac{1}{1+25x^2}$ against its interpolation with polynomials with equidistant interpolation. We therefore wish to have a quadrature with a "optimal" Lebesgue constant. For ex-

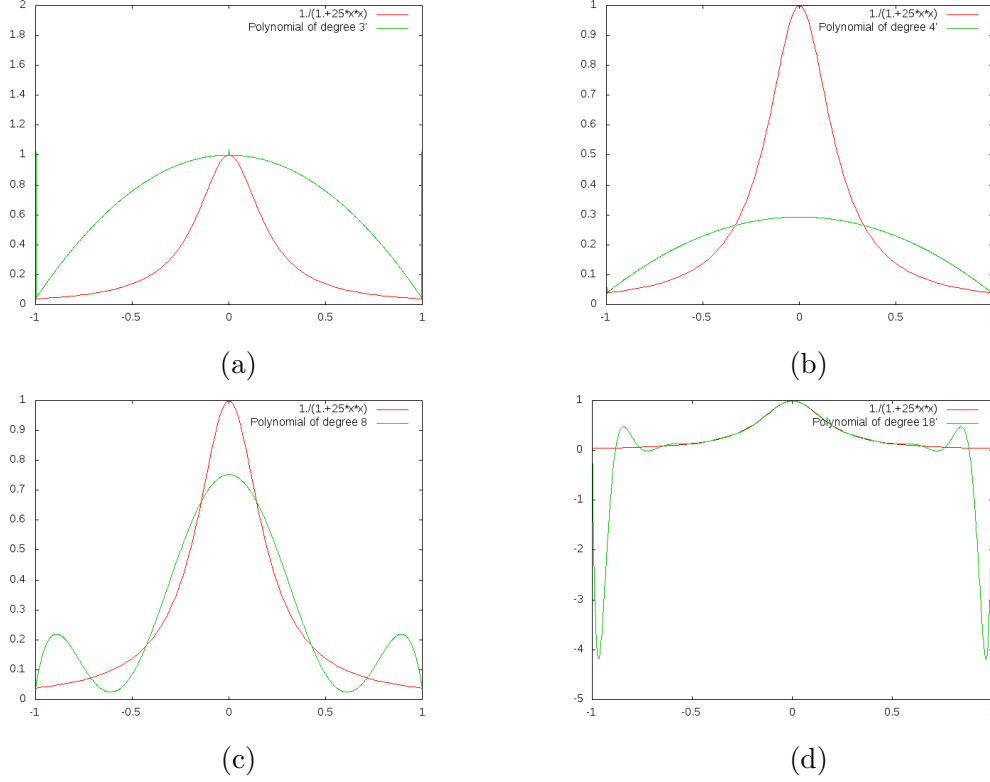


Figure 1.3: Runge function $\frac{1}{1+25x^2}$ and its polynomial interpolations at degrees 3, 4, 8, and 18.

ample Gauss–Chebyshev interpolation points (corresponding to the 0 of the polynomial $T_{N+1}(x) = \cos((N+1) \arccos(x))$) give a Lebesgue constant $\lambda_N(X_{GC})$ equal to

$$\lambda_N(X_{GC}) \simeq \frac{2}{\pi} \ln(N+1)$$

For our problem, we want to interpolate a function on meshes with great precision on the mesh while respecting the continuity of the function between the meshes. In order to ensure this continuity we want the extreme points of the resized mesh $[-1, 1]$ (therefore $-1, 1$) to be on the interpolation grid. This leads to the Gauss–Lobatto–Chebyshev interpolation grid.

In the library we choose to use the Gauss–Lobatto–Legendre interpolation grids which are as efficient as the Gauss–Lobatto–Chebyshev grids (in term of Lebesgue constant) but less costly in calculation due to the absence of trigonometric functions. We recall that the Legendre polynomial satisfies the recurrence

$$(N+1)L_{N+1}(x) = (2N+1)xL_N(x) - NL_{N-1}(x)$$

with $L_0 = 1$, $L_1(x) = x$.

These polynomials are orthogonal with the scalar product $(f, g) = \int_{-1}^1 f(x)g(x)dx$. We are interested in the derivatives of these polynomials L'_N which satisfy the recurrence

$$NL'_{N+1}(x) = (2N+1)xL'_N(x) - (N+1)L'_{N-1}(x)$$

These polynomials are orthogonal with the scalar product $(f, g) = \int_{-1}^1 f(x)g(x)(1-x^2)dx$. The Gauss–Lobatto–Legendre grids points for a grid with $N+1$ points are $\eta_1 = -1, \eta_{N+1} = 1$ and the η_i ($i = 2, \dots, N$) zeros of L'_N . The η_i ($i = 2, \dots, N$) are eigenvalues of the matrix P

$$P = \begin{pmatrix} 0 & \gamma_1 & \dots & 0 & 0 \\ \gamma_1 & 0 & \dots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \dots & 0 & \gamma_{N-2} \\ 0 & 0 & \dots & \gamma_{N-2} & 0 \end{pmatrix},$$

$$\gamma_n = \frac{1}{2} \sqrt{\frac{n(n+2)}{(n+\frac{1}{2})(n+\frac{3}{2})}}, 1 \leq n \leq N-2,$$

The interpolation $I_N(f)$ is expressed in terms of Legendre polynomials by

$$I_N(f) = \sum_{k=0}^N \tilde{f}_k L_k(x),$$

$$\tilde{f}_k = \frac{1}{\gamma_k} \sum_{i=0}^N \rho_i f(\eta_i) L_k(\eta_i),$$

$$\gamma_k = \sum_{i=0}^N L_k(\eta_i)^2 \rho_i,$$

and the weights satisfy

$$\rho_i = \frac{2}{(M+1)ML_M^2(\eta_i)}, 1 \leq i \leq N+1.$$

More details can be found in [2]. In the figure 1.4, we give the interpolation obtained with the Gauss–Lobatto–Legendre quadrature with two degrees of approximation.

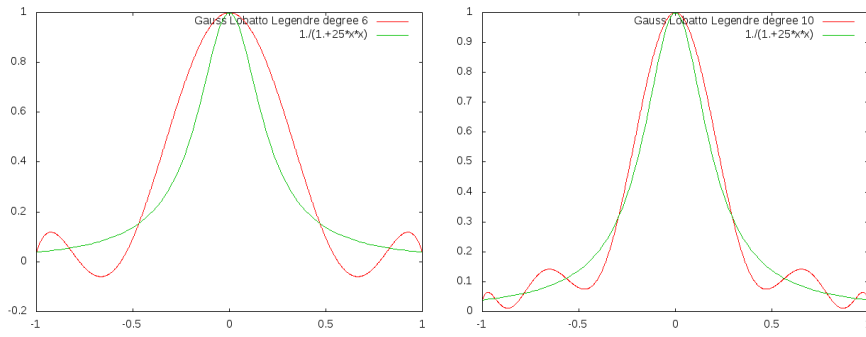


Figure 1.4: Interpolation with Gauss–Legendre–Lobatto grids

- When the function is not regular we introduce a weaker notion than the notion of derivative. We denote $w(f, \delta)$ The modulus of continuity on $[-1, 1]$ of a function f as

$$w(f, \delta) = \sup_{\substack{x_1, x_2 \in [-1, 1] \\ |x_1 - x_2| < \delta}} |f(x_1) - f(x_2)|$$

The modulus of continuity makes it possible to express the best approximation of a function by a polynomial with the Jackson theorem:

Theorem 1 *For a continuous function f on $[-1, 1]$*

$$\|f - P_N^*(f)\|_\infty \leq Kw(f, \frac{1}{N})$$

and we deduce that for an interpolation grid X

$$\begin{aligned} \|I_N^X(f)(x) - f\|_\infty &\leq M(N) \\ M(N) &\simeq Kw(f, \frac{1}{N})\lambda_N(X) \end{aligned}$$

a function is Dini–Lipschitz continuous if $w(f, \delta)\log(\delta) \rightarrow 0$ as $\delta \rightarrow 0$. It is clear that the functions of Lipschitz are Dini–Lipschitz continuously because $w(f, \delta)\log(\delta) \leq K\log(\delta)\delta$.

- When the solution is more regular, we can express the interpolation error as a function of its derivatives and we obtain the following Cauchy theorem for an interpolation grid X (see [41])

Theorem 2 *If f is C^{N+1} , and X an interpolation grid with $N + 1$ points, the interpolation error checks*

$$E(x) = f(x) - I_N^X(f)(x) = \frac{f^{N+1}(\eta)}{(N+1)!} W_{N+1}^X(x) \quad (1.2)$$

where $\eta \in [-1, 1]$ and $W_{N+1}^X(x)$ is the nodal polynomial of degree $N + 1$ (the polynomial with the monomial of the highest degree with coefficient 1 being zero at all points $N + 1$ of X)

If we partition a domain $I = [a, b]$ into a few meshes of size h and we use a Lagrange interpolator for the function $f \in C^{k+1}$, $k \leq N$ we obtain

$$\|f - I_{N,\Delta x}^X f\|_\infty \leq ch^{k+1} \|f^{(k+1)}\|_\infty$$

1.2.2 Extension in dimension d

In the dimension d , we denote P_N^* the best multivariate polynomial approximation of f of total degree less than N on $[-1, 1]^d$. On a d multidimensional grid $X = X_N^d$, we define the multivariate interpolator as the composition of one-dimensional interpolator $I_N^X(f)(x) = I_N^{X_N,1} \times I_N^{X_N,2} \dots \times I_N^{X_N,d}(f)(x)$ where $I_N^{X_N,i}$ represents for the interpolator in dimension i . We get the following interpolation error

$$\|I_N^X(f) - f\|_\infty \leq (1 + \lambda_N(X_N))^d \|f - P_N^*(f)\|_\infty,$$

The error associated with the min max approximation is given by Feinerman and Newman [15], Soardi [44]

$$\|f - P_N^*(f)\|_\infty \leq (1 + \frac{\pi^2}{4}\sqrt{d})w(f, \frac{1}{N+2})$$

We deduce that if f is only Lipschitz

$$\|I_N^X(f)(x) - f\|_\infty \leq C\sqrt{d}\frac{(1 + \lambda_N(X))^d}{N+2}$$

If the function is regular (in $C^{k+1}([-1, 1]^d)$, $k < N$) we get

$$\|f - P_N^*(f)\|_\infty \leq \frac{C_k}{N^k} \sum_{i=1}^d \sup_{x \in [-1, 1]^d} |\frac{\partial^{k+1} f}{\partial x_i^{k+1}}|$$

If we partition the domain $I = [a_1, b_1] \times \dots \times [a_d, b_d]$ in meshes of size $\Delta x = (\Delta x_1, \Delta x_2, \dots, \Delta x_d)$ and use a Lagrange interpolation on each mesh we obtain

$$\|f - I_{N, \Delta x}^X f\|_\infty \leq c \frac{(1 + \lambda_N(X))^d}{N^k} \sum_{i=1}^d \Delta x_i^{k+1} \sup_{x \in [-1, 1]^d} |\frac{\partial^{k+1} f}{\partial x_i^{k+1}}|$$

On figure 1.5 we give the Gauss–Legendre–Lobatto points in 2D for 2×2 meshes and a polynomial of degree 8 in each direction

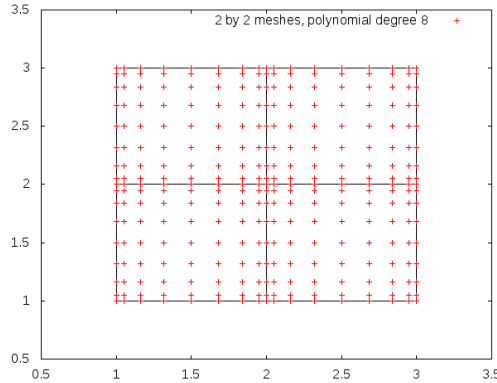


Figure 1.5: Gauss–Legendre–Lobatto points on 2×2 meshes.

1.2.3 Truncature

In order to avoid oscillations during the interpolation, a truncation is used on each mesh so that the modified interpolator $\hat{I}_{N, \Delta x}^X$ checks:

$$\hat{I}_{N, \Delta x}^X f(x) = \min_{x_i \in M} f(x_i) \wedge I_{N, \Delta x}^X f(x) \vee \max_{x_i \in M} f(x_i) \quad (1.3)$$

where the x_i are the interpolation points on the mesh M containing the point x . For all the characteristics of this modified operator, we can see [49].

1.2.4 The C++ API

The grid using Gauss–Legendre–Lobatto points can be created using this constructor:

```
1 RegularLegendreGrid(const Eigen::ArrayXd &p_lowValues, const Eigen::ArrayXd &p_step,
2                     const Eigen::ArrayXi &p_nbStep, const Eigen::ArrayXi &p_poly);
```

The `p_lowValues` corresponds to the bottom of the grid, `p_step` the size of each mesh, `p_nbStep` the number of steps in each direction (see figure 1.2). On each mesh the polynomial approximation in each dimension is specified by the table `p_poly`.

Remark 3 *If we take a polynomial of degree 1 in each direction this interpolator is equivalent to the linear interpolator. It's sort of less efficient than the linear interpolator on a Regular grid described in the section above.*

We illustrate the use of the grid, its iterator and its interpolator used to draw the figures 1.4.

```
1
2 ArrayXd lowValues = ArrayXd::Constant(1,-1.); // corner point
3 ArrayXd step= ArrayXd::Constant(1,2.); // mesh size
4 ArrayXi nbStep = ArrayXi::Constant(1,1); // number of meshes in each direction
5 ArrayXi nPol = ArrayXi::Constant(1,p_nPol); // polynomial approximation
6 // regular Legendre
7 RegularLegendreGrid regGrid(lowValues, step, nbStep, nPol);
8
9 // Data table to store values on the grid points
10 ArrayXd data(regGrid.getNbPoints());
11 shared_ptr<GridIterator> iterRegGrid = regGrid.getGridIterator();
12 while (iterRegGrid->isValid())
13 {
14     ArrayXd pointCoord = iterRegGrid->getCoordinate();
15     data(iterRegGrid->getCount()) = 1./(1.+25*pointCoord(0)*pointCoord(0)); // runge
16     // storage function
17     iterRegGrid->next();
18 }
19 // point
20 ArrayXd point(1);
21 int nbp = 1000;
22 double dx = 2./nbp;
23 for (int ip =0; ip<= nbp; ++ip)
24 {
25     point(0)= -1+ ip* dx;
26 // create interpolator
27     shared_ptr<Interpolator<double>> interp = regGrid.createInterpolator( point);
28     double interpReg = interp->apply(data); // interpolated value
29 }
```

The operator defined above is more efficient when we interpolate several functions at the same point. Its is the case for example for the valuation of a storage with regression where one wishes to interpolate all the simulations at the same level of stock.

In some cases it is more practical to build an interpolator acting on a global function. This is the case when you have only one function and you want to interpolate at several points for this function. In this specific case an interpolator deriving from the class `InterpolatorSpectral` can be constructed:

```
1 // Copyright (C) 2016 EDF
2 // All Rights Reserved
3 // This code is published under the GNU Lesser General Public License (GNU LGPL)
4 #ifndef INTERPOLATORSPECTRAL_H
```

```

5 #define INTERPOLATORSPECTRAL_H
6 #include <memory>
7 #include <Eigen/Dense>
8
9 /** \file InterpolatorSpectral.h
10 * \brief Defines an interpolator for a grid : here is a global interpolator, storing the
    representation of the function
11 * to interpolate : this interpolation is effective when interpolating the same
    function many times at different points
12 * Here it is an abstract class
13 * \author Xavier Warin
14 */
15 namespace StOpt
16 {
17
18 /// forward declaration
19 class SpaceGrid ;
20
21 /// \class InterpolatorSpectral InterpolatorSpectral.h
22 /// Abstract class for spectral operator
23 template<typename T>
24 class InterpolatorSpectral
25 {
26
27 public :
28     virtual ~InterpolatorSpectral() {}
29
30     /** \brief interpolate
31     * \param p_point coordinates of the point for interpolation
32     * \return interpolated value
33     */
34     virtual T apply(const Eigen::Array<T, Eigen::Dynamic, 1> &p_point) const = 0;
35
36
37     /** \brief Affect the grid
38     * \param p_grid the grid to affect
39     */
40     virtual void setGrid(const StOpt::SpaceGrid *p_grid) = 0 ;
41
42     /** \brief Get back grid associated to operator
43     */
44     virtual const StOpt::SpaceGrid *getGrid() const = 0;
45
46 };
47 }
48 #endif

```

Its constructor is given by:

```

1 /** \brief Constructor taking values on the grid
2 * \param p_grid is the grid used to interpolate
3 * \param p_values Value of the function at the points of the grid.
4 */
5 LegendreInterpolatorSpectral(const shared_ptr< RegularLegendreGrid> &p_grid , const
    Eigen::Array<T, Eigen::Dynamic, 1> &p_values) ;

```

This class has a member allowing to interpolate at a given point:

```

1 /** \brief interpolate
2 * \param p_point coordinates of the point for interpolation
3 * \return interpolated value
4 */
5 inline double apply(const Eigen::ArrayXd &p_point) const

```

We give an example of using this class, by interpolating a function f function in dimension 2.

```

1 ArrayXd lowValues = ArrayXd::Constant(2,1.); // bottom of the domain

```

```

2  ArrayXd step = ArrayXd::Constant(2,1.); // mesh size
3  ArrayXi nbStep = ArrayXi::Constant(2,5); // number of meshes in each direction
4  ArrayXi nPol = ArrayXi::Constant(2,2); // polynomial of degree 2 in each direction
5  // regular
6  shared_ptr<RegularLegendreGrid> regGrid(new RegularLegendreGrid(lowValues, step, nbStep
7  , nPol));
8  ArrayXd data(regGrid->getNbPoints()); // Data table
9  shared_ptr<GridIterator> iterRegGrid = regGrid->getGridIterator(); // iterator on the
10 grid points
11 while (iterRegGrid->isValid())
12 {
13     ArrayXd pointCoord = iterRegGrid->getCoordinate();
14     data(iterRegGrid->getCount()) = f(pointCoord);
15     iterRegGrid->next();
16 }
17 // spectral interpolator
18 LegendreInterpolatorSpectral<double> interpolator(regGrid,data);
19 // interpolation point
20 ArrayXd pointCoord(2, 5.2);
21 // interpolated value
22 double vInterp = interpolator.apply(pointCoord);

```

1.2.5 The Python API

Here is an example using Legendre grids:

```

1  # Copyright (C) 2016 EDF
2  # All Rights Reserved
3  # This code is published under the GNU Lesser General Public License (GNU LGPL)
4  import numpy as np
5  import unittest
6  import random
7  import math
8  import pystopt.grids as StOptGrids
9
10 # unit test for Legendre grids
11 #####
12
13 class testGrids(unittest.TestCase):
14
15     # test Legendre grids
16     def testLegendreGrids(self):
17         # low value for the mesh
18         lowValues = np.array([1.,2.,3.], dtype=float)
19         # size of the mesh
20         step = np.array([0.7,2.3,1.9], dtype=float)
21         # number of step
22         nbStep = np.array([4,5,6], dtype=np.int32)
23         # degree of the polynomial in each direction
24         degree = np.array([2,1,3], dtype=np.int32)
25         # create the Legendre grid
26         grid = StOptGrids.RegularLegendreGrid(lowValues,step,nbStep,degree )
27         iterGrid = grid.getGridIterator()
28         # array to store
29         data = np.empty(grid.getNbPoints())
30         # iterates on point
31         while( iterGrid.isValid()):
32             #get coordinates of the point
33             pointCoord = iterGrid.getCoordinate()
34             data[iterGrid.getCount()] = math.log(1. + pointCoord.sum())
35             iterGrid.next()
36         # get back an interpolator
37         ptInterp = np.array([2.3,3.2,5.9], dtype=float)
38         interpol = grid.createInterpolator(ptInterp)
39

```

```

40         # calculate interpolated value
41         interpValue = interpol.apply(data)
42         print(("Interpolated value Legendre" , interpValue))
43         # test grids function
44         iDim = grid.getDimension()
45         pt = grid.getExtremeValues()
46
47
48 if __name__ == '__main__':
49     unittest.main()

```

1.3 Sparse grids

A representation of a function of dimension d for d small (less than 4) is obtained by tensorization in the preceding methods of interpolation. When the function is smooth and its cross derivatives are bounded, the function can be represented using the sparse grid methods. This method makes it possible to represent the function with much less points than the traditional one without losing too much by interpolating. The sparse grid method was used for the first time by assuming that the function f to be represented is zero at the boundary Γ of the domain. This assumption is important because it makes it possible to limit the explosion of the number of points to the dimension of the problem. In many application this assumption is not realistic or it is impossible to work on $f - f|_{\Gamma}$. In this library we will assume that the function is not zero at the border and provide grid object, iterators and interpolators to interpolate certain functions depicted on the sparse grid. However, for the sake of clarity of presentation, we will start with the case of a function disappearing on the border.

1.3.1 The sparse linear grid method

We recall some classic results on sparse grids which can be found in [40]. We first assume that the function we are interpolating is zero at the border. By a change of coordinates a hyper-cube domain can be changed to a domain $\omega = [0, 1]^d$. By introducing the hat function $\phi^{(L)}(x) = \max(1 - |x|, 0)$ (where (L) means linear), we obtain the following one-dimensional local hat function by translation and dilation

$$\phi_{l,i}^{(L)}(x) = \phi^{(L)}(2^l x - i)$$

according to the level l and the index i , $0 < i < 2^l$. The grid points used for the interpolation are noted $x_{l,i} = 2^{-l}i$. In the dimension d , we introduce the basis functions

$$\phi_{\underline{l},\underline{i}}^{(L)}(x) = \prod_{j=1}^d \phi_{l_j,i_j}^{(L)}(x_j)$$

via a tensorial approach for a point $\underline{x} = (x_1, \dots, x_d)$, a multi-level $\underline{l} := (l_1, \dots, l_d)$ and a multi-index $\underline{i} := (i_1, \dots, i_d)$. The grid points used for interpolation are noted $x_{\underline{l},\underline{i}} := (x_{l_1,i_1}, \dots, x_{l_d,i_d})$.

We then present the index set

$$B_{\underline{l}} := \{\underline{i} : 1 \leq i_j \leq 2^{l_j} - 1, i_j \text{ odd}, 1 \leq j \leq d\}$$

and the hierarchical base space.

$$W_{\underline{l}}^{(L)} := \text{span} \left\{ \phi_{\underline{l}, \underline{i}}^{(L)}(\underline{x}) : \underline{i} \in B_{\underline{l}} \right\}$$

A representation of the space $W_{\underline{l}}^{(L)}$ is given in dimension 1 on the figure 1.6. The sparse

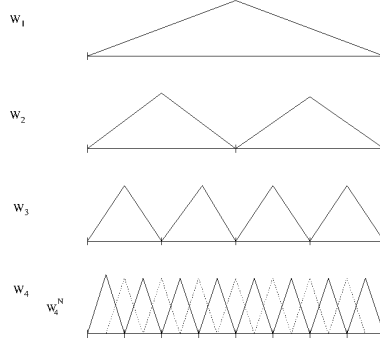


Figure 1.6: One dimensional $W_{\underline{l}}^{(L)}$ spaces: $W_1^{(L)}$, $W_2^{(L)}$, $W_3^{(L)}$, $W_4^{(L)}$ and the nodal representation $W_4^{(L,N)}$

mesh space is defined as follows:

$$V_n = \bigoplus_{|\underline{l}|_1 \leq n+d-1} W_{\underline{l}}^{(L)} \quad (1.4)$$

Remark 4 The conventional full grid space is defined as $V_n^F = \bigoplus_{|\underline{l}|_\infty \leq n} W_{\underline{l}}^{(L)}$.

At hierarchical increment space $W_{\underline{l}}^{(L)}$ corresponds to a nodal functions space $W_{\underline{l}}^{(L,N)}$ such that

$$W_{\underline{l}}^{(L,N)} := \text{span} \left\{ \phi_{\underline{l}, \underline{i}}^{(L)}(\underline{x}) : \underline{i} \in B_{\underline{l}}^N \right\}$$

with

$$B_{\underline{l}}^N := \left\{ \underline{i} : 1 \leq i_j \leq 2^{l_j} - 1, 1 \leq j \leq d \right\}.$$

In the figure 1.6 the one-dimensional nodal base $W_4^{(L,N)}$ is generated by $W_4^{(L)}$ and the dotted base function. The space V_n can be represented as the space spawn by the $W_{\underline{l}}^{(L,N)}$ such that $|\underline{l}|_1 = n + d - 1$:

$$V_n = \text{span} \left\{ \phi_{\underline{l}, \underline{i}}^{(L)}(\underline{x}) : \underline{i} \in B_{\underline{l}}^N, |\underline{l}|_1 = n + d - 1 \right\} \quad (1.5)$$

A function f is interpolated on the hierarchical basis like

$$I^{(L)}(f) = \sum_{|\underline{l}|_1 \leq n+d-1, \underline{i} \in B_{\underline{l}}} \alpha_{\underline{l}, \underline{i}}^{(L)} \phi_{\underline{l}, \underline{i}}^{(L)}$$

where $\alpha_{\underline{l}, \underline{i}}^{(L)}$ are called the surplus (we give on the figure 1.7 a representation of these coefficients). These surpluses associated with a function f are calculated in the one-dimensional

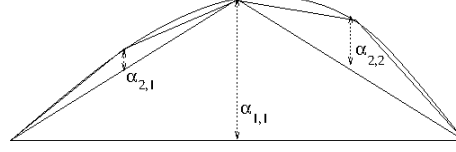


Figure 1.7: Example of hierarchical coefficients

case for a node $m = x_{l,i}$ as the difference of the value of the function at the node and the linear representation of the calculated function with neighboring nodes. For example in figure 1.8, the hierarchical value is given by the relation:

$$\alpha^{(L)}(m) := \alpha_{l,i}^{(L)} = f(m) - 0.5(f(e(m)) + f(w(m)))$$

where $e(m)$ is the east neighbor of m and $w(m)$ the west one. The procedure is generalized in dimension d by successive hierarchy in all directions. On figure 1.9, we give a representation

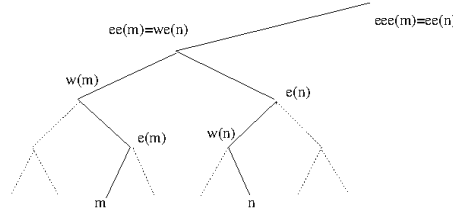


Figure 1.8: Node involved in the linear, quadratic and cubic representation of a function at the node m and n

of the subspace W for $l \leq 3$ in dimension 2.

In order to deal with functions non-zero at the boundary, two more basis are added to the

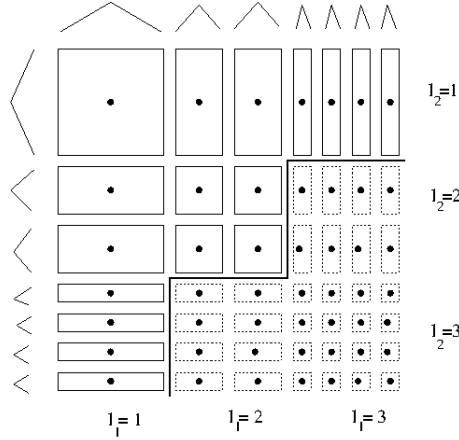


Figure 1.9: The two-dimensional subspace $W_l^{(L)}$ up to $l = 3$ in each dimension. Additional hierarchical functions corresponding to an approximation on the complete grid are indicated by dotted lines.

first level as shown on figure 1.10. This approach leads to many more points than the one

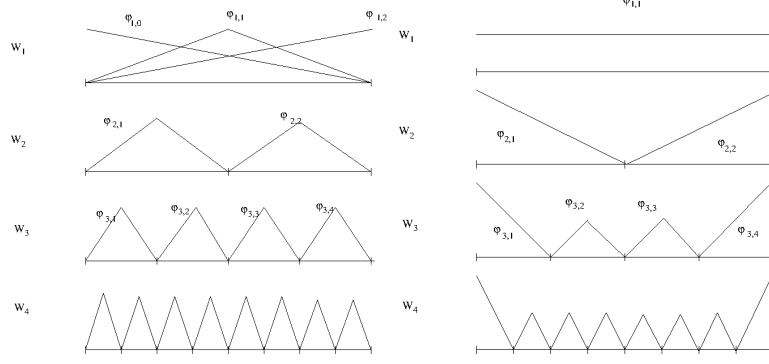


Figure 1.10: One dimensional $W^{(L)}$ spaces with linear functions with “exact” boundary (left) and “modified” boundary (right): $W_1^{(L)}$, $W_2^{(L)}$, $W_3^{(L)}$, $W_4^{(L)}$

without the border. As indicated in [40] for $n = 5$, in dimension 8 have almost 2.8 million points in this approximation but only 6401 inside the domain. On the figure 1.11 we give the points of the grids including boundary points in dimension 2 and 3 for a level 5 of the sparse grid.

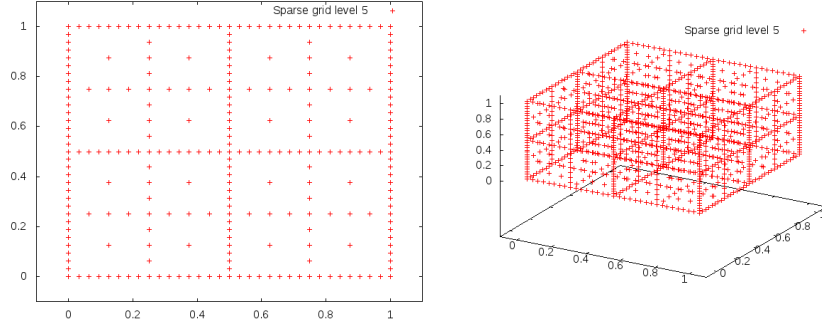


Figure 1.11: Sparse grid in dimension 2 and 3 with boundary points

If the boundary conditions are not important (infinite truncated domain in finance for example) the hat functions near the borders are modified by extrapolation (see figure 1.10) as explained in [40]. At level 1, we have only one degree of freedom assuming that the function is constant on the domain. At all the other levels, we extrapolate linearly towards the boundary the basis functions left and right, the other functions remaining unchanged. So the new function base in 1D $\tilde{\phi}$ becomes

$$\tilde{\phi}_{l,i}^{(L)}(x) = \begin{cases} 1 & \text{if } l = 1 \text{ and } i = 1 \\ \begin{cases} 2 - 2^l x & \text{if } x \in [0, 2^{-l+1}] \\ 0 & \text{else} \end{cases} & \text{if } l > 1 \text{ and } i = 1 \\ \begin{cases} 2^l(x - 1) + 2 & \text{if } x \in [1 - 2^{-l+1}, 1] \\ 0 & \text{else} \end{cases} & \text{if } l > 1 \text{ and } i = 2^l - 1 \\ \phi_{l,i}^{(L)}(x) & \text{otherwise} \end{cases}$$

On the figure 1.12 we give the grid points by eliminating the boundary points in dimension 2 and 3 for a level 5 of the sparse grid.

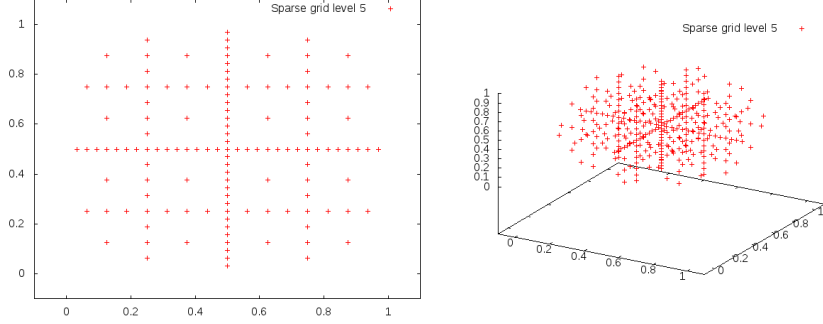


Figure 1.12: Sparse grid in dimensions 2 and 3 without boundary points

The interpolation error associated with the linear operator $I^1 := I^{(L)}$ is related to the regularity of the crossed derivatives of the function [10, 11, 12]. If f is zero at the boundary and admits derivatives such as $\|\frac{\partial^{2d}u}{\partial x_1^2 \dots \partial x_d^2}\|_\infty < \infty$ then

$$\|f - I^1(f)\|_\infty = O(N^{-2} \log(N)^{d-1}), \quad (1.6)$$

with N the number of points per dimension.

1.3.2 High order sparse grid methods

Changing the interpolator allows us to obtain a higher convergence rate mainly in the region where the solution is smooth. By following [11] and [12], it is possible to obtain higher order interpolators. Using a quadratic interpolator, the reconstruction on the nodal base gives a quadratic function on the support of the hat function previously defined and a continuous function of the whole domain. The polynomial quadratic base is defined on $[2^{-l}(i-1), 2^{-l}(i+1)]$ by

$$\phi_{l,i}^{(Q)}(x) = \phi^{(Q)}(2^l x - i)$$

with $\phi^{(Q)}(x) = 1 - x^2$.

The hierarchical surplus (coefficient on the basis) in a dimension is the difference between the value function at the node and the quadratic representation of the function using nodes available at the previous level. With the notation of figure 1.8

$$\begin{aligned} \alpha(m)^{(Q)} &= f(m) - \left(\frac{3}{8}f(w(m)) + \frac{3}{4}f(e(m)) - \frac{1}{8}f(ee(m))\right) \\ &= \alpha(m)^{(L)}(m) - \frac{1}{4}\alpha(m)^{(L)}(e(m)) \\ &= \alpha(m)^{(L)}(m) - \frac{1}{4}\alpha(m)^{(L)}(df(m)) \end{aligned}$$

where $df(m)$ is the direct father of the node m in the tree structure.

Once again, the quadratic surplus of dimension d is obtained by successive hierarchization in the different dimensions.

In order to take into account the boundary conditions, two linear functions $1 - x$ and x are added at the first level (see figure 1.13).

A version with modified boundary conditions can be derived for example by using linear interpolation at the boundary so that

$$\tilde{\phi}_{l,i}^{(Q)}(x) = \begin{cases} \tilde{\phi}_{l,i}^{(L)} & \text{if } i = 1 \text{ or } i = 2^l - 1, \\ \phi_{l,i}^{(Q)}(x) & \text{otherwise} \end{cases}$$

In the case of the cubic representation, on figure 1.8 we need 4 points to define a function

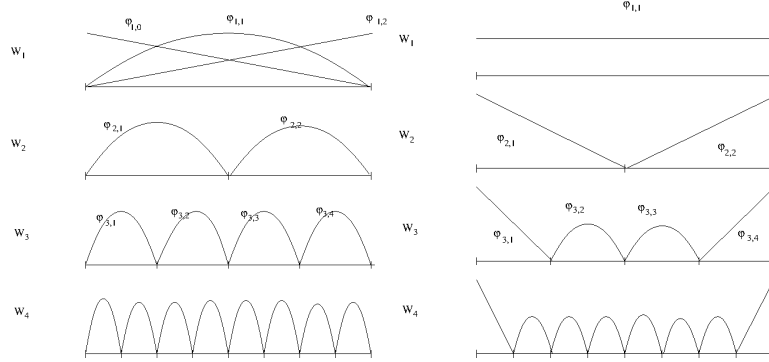


Figure 1.13: One dimensional $W^{(Q)}$ spaces with quadratic with “exact” boundary (left) and “modified” boundary (right): $W_1^{(Q)}$, $W_2^{(Q)}$, $W_3^{(Q)}$, $W_4^{(Q)}$

basis. In order to keep the same data structure, we use a cubic function base at node m with value 1 at this node and 0 at the node $e(m)$, $w(m)$ and $ee(m)$ and we keep only the basic function between $w(m)$ and $e(m)$ [11].

Note that there are two basic types of function depending on the position in the tree. The basic functions are given on $[2^{-l+1}i, 2^{-l+1}(i+1)]$ by

$$\begin{aligned} \phi_{l,2i+1}^{(C)}(x) &= \phi^{(C),1}(2^l x - (2i+1)), \text{ if } i \text{ even} \\ &= \phi^{(C),2}(2^l x - (2i+1)), \text{ if } i \text{ odd} \end{aligned}$$

with $\phi^{(C),1}(x) = \frac{(x^2-1)(x-3)}{3}$, $\phi^{(C),2}(x) = \frac{(1-x^2)(x+3)}{3}$.

The surplus coefficient can be defined as above as the difference between the value function at the node and the cubic representation of the function at the parent node. Due to the two basis functions involved there are two types of cubic coefficient.

- For a node $m = x_{l,8i+1}$ or $m = x_{l,8i+7}$, $\alpha^{(C)}(m) = \alpha^{(C,1)}(m)$, with

$$\alpha^{(C,1)}(m) = \alpha^{(Q)}(m) - \frac{1}{8}\alpha^{(Q)}(df(m))$$

- For a node $m = x_{l,8i+3}$ or $m = x_{l,8i+5}$, $\alpha^{(C)}(m) = \alpha^{(C,2)}(m)$, with

$$\alpha^{(C,2)}(m) = \alpha^{(Q)}(m) + \frac{1}{8}\alpha^{(Q)}(df(m))$$

Note that a cubic representation is not available for $l = 1$ so a quadratic approximation is used. As before, the boundary conditions are treated by adding two linear basis functions

at the first level and a modified version is available. We choose the following basis functions as defined in figure 1.14:

$$\tilde{\phi}_{l,i}^{(C)}(x) = \begin{cases} \tilde{\phi}_{l,i}^{(Q)} & \text{if } i \in \{1, 3, 2^l - 3, 2^l - 1\}, \\ \phi_{l,i}^{(C)}(x) & \text{otherwise} \end{cases}$$

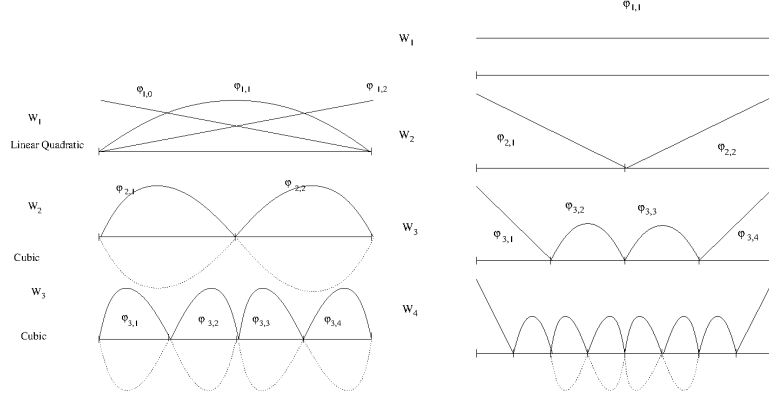


Figure 1.14: One dimensional $W^{(C)}$ spaces with cubic and “exact” boundary (left) and “modified” boundary (right): $W_1^{(C)}$, $W_2^{(C)}$, $W_3^{(C)}$, $W_4^{(C)}$

According to [10, 11, 12], if the function f is zero at the boundary and admits derivatives such as $\sup_{\alpha_i \in \{2, \dots, p+1\}} \left\{ \left\| \frac{\partial^{\alpha_1 + \dots + \alpha_d} u}{\partial x_1^{\alpha_1} \dots \partial x_d^{\alpha_d}} \right\|_{\infty} \right\} < \infty$ then the interpolation error can be generalized for $I^2 := I^{(Q)}$, $I^3 := I^{(C)}$ by:

$$\|f - I^p(f)\|_{\infty} = O(N^{-(p+1)} \log(N)^{d-1}), \quad p = 2, 3$$

with N the number of points per dimension.

1.3.3 Anisotropy

In many situations, there is no point in refining as much in each direction. For example, when we process multidimensional storage, we expect the mesh to be the same order in each direction. When the different storages have very different sizes, we want to further refine the storage with the largest capacity. In order to treat this anisotropy, an extension of the sparse grids can be obtained by defining the weight w in each direction. The definition 1.4 is replaced by:

$$V_n = \bigoplus_{\sum_{i=1}^d l_i w(i) \leq n+d-1} W_l^{(L)} \quad (1.7)$$

1.3.4 Adaptation

When the solution is not smooth, typically Lipschitz, there is no hope of obtaining convergence results for the classical rare grids (see above the interpolation error linked to crossed derivatives of the function). The classic sparse grids must therefore be adapted so that the

solution is refined near the singularities. In all adaptation methods, the hierarchical surplus $\alpha_{l,i}$ is used to obtain an estimate of the local error. These coefficients give an estimate of the regularity of the value of the function at the discrete points by representing the second derivative of the discrete mixture of the function. There are mainly two types of adaptation used:

- the first performs a local adaptation and only adds points locally [13, 20, 21, 32],
- the second performs the adaptation at the level of the hierarchical space W_l (anisotropic sparse grid). This approach detects the important dimensions that need to be refined and refines all the points of that dimension [17]. This refinement is also achieved in areas where the solution can be smooth. A more local version has been developed in [27].

In the current version of the library, only the dimension adaptation is available. Details of the algorithm can be found in [17]. After a first initialization with a space

$$V_n = \bigoplus_{\sum_{i=1}^d l_i \leq n+d-1} W_l^{(L)} \quad (1.8)$$

An active level set $\mathcal{A}$ is created grouping all the levels l so that $\sum_{i=1}^d l_i = n + d - 1$. All the other levels are grouped into a set $\mathcal{O}$. At each level l in $\mathcal{A}$ an error is estimated e_l and with any local error e_l a global error E is calculated. Then the refinement algorithm 1 is used by noting $\mathbf{e}_k$ the canonical basis in the dimension k . Sometimes, using sparse grids during time

Algorithm 1 Dimension refinement for a given tolerance η

```

1: while  $E > \eta$  do
2:   select  $l$  with the highest local error  $e_l$ 
3:    $\mathcal{A} = \mathcal{A} \setminus \{l\}$ 
4:    $\mathcal{O} = \mathcal{O} \cup \{l\}$ 
5:   for  $k = 1$  to  $d$  do
6:      $\underline{m} = l + \mathbf{e}_k$ 
7:     if  $\underline{m} - \mathbf{e}_q \in \mathcal{O}$  for  $q \in [1, d]$  then
8:        $\mathcal{A} = \mathcal{A} \cup \{\underline{m}\}$ 
9:       Hierarchize all points belonging to  $\underline{m}$ 
10:      calculate  $e_{\underline{m}}$ 
11:      update  $E$ 
12:     end if
13:   end for
14: end while

```

iterations, it can be interesting to enlarge the meshes. A similar algorithm 2 can be used to eliminate the levels with a very small local error.

Algorithm 2 Dimension coarsening for a given tolerance η

$\mathcal{B}$ all elements of $\mathcal{A}$ with a local error below η
while $\mathcal{B}$ non nonempty **do**
 select $\underline{l} \in \mathcal{B}$ with the lowest local error $e_{\underline{l}}$
 for $k = 1$ to d **do**
 $\underline{m} = \underline{l} - \mathbf{e}_k$
 if $m_k > 0$ **then**
 if $\underline{m} + \mathbf{e}_q \in \mathcal{B}$ for $q \in [1, d]$ **then**
 $\mathcal{A} = \mathcal{A} \setminus \{\underline{m} + \mathbf{e}_q, q \in [1, d]\}$
 $\mathcal{B} = \mathcal{B} \setminus \{\underline{m} + \mathbf{e}_q, q \in [1, d]\}$
 $\mathcal{A} = \mathcal{A} \cup \{\underline{m}\}$
 Add $\underline{m}$ to $\mathcal{B}$ if local error below η
 $\mathcal{O} = \mathcal{O} \setminus \{\underline{m}\}$
 Break
 end if
 end if
 end for
 if $\underline{l} \in \mathcal{B}$ **then**
 $\mathcal{B} = \mathcal{B} \setminus \{\underline{l}\}$
 end if
end while

1.3.5 C++ API

The construction of the sparse grid, including the boundary points is carried out by the following constructor

```
1 SparseSpaceGridBound(const Eigen::ArrayXd &p_lowValues, const Eigen::ArrayXd &  
  p_sizeDomain, const int &p_levelMax, const Eigen::ArrayXd &p_weight,  
2                      const size_t &p_degree)
```

with

- `p_lowValues` corresponds to the bottom of the grid,
- `p_sizeDomain` corresponds to the size of the resolution domain in each dimension,
- `p_levelMax` is the level of the sparse grids, the n in the equation 1.7,
- `p_weight` the weight of anisotropic sparse grids, the equation w 1.7,
- `p_degree` is equal to 1 (linear interpolator), or 2 (quadratic interpolator) or 3 (for cubic interpolator),

With the same notations the construction eliminating boundary points is carried out by the following constructor

```
1 SparseSpaceGridNoBound(const Eigen::ArrayXd &p_lowValues, const Eigen::ArrayXd &  
  p_sizeDomain, const int &p_levelMax, const Eigen::ArrayXd &p_weight,  
2                      const size_t &p_degree)
```

The data structure of type `SparseSet` to store the sparse grid is defined by a map with keys an array A storing a multiple level and values a map with keys an array B storing the multi index associated with a point (A,B) and values the number of points (A,B):

```
1  #define SparseSet std::map< Eigen::Array<char,Eigen::Dynamic,1 > , std::map< Eigen
    ::Array<unsigned int,Eigen::Dynamic,1> , size_t, OrderTinyVector< unsigned int > > ,
    OrderTinyVector< char> >
```

It is sometimes convenient to retrieve this data structure from the `SparseGrid` object: this is done by the following method:

```
1  std::shared_ptr<SparseSet> getDataSet() const ;
```

The two preceding classes have two specific member functions for hierarchizing (see the section above) the known value function at the points of the grid for the entire grid.

- the first work on a single function:

```
1  /// \brief Hierarchize a function defined on the grid
2  /// \param p_toHierarchize function to hierarchize
3  void toHierarchize( Eigen::ArrayXd & p_toHierarchize );
```

- the second work on a matrix, allowing to hierarchize several functions in a single call (each row corresponds to a function representation)

```
1
2  /// \brief Hierarchize a set of functions defined on the grid
3  /// \param p_toHierarchize function to hierarchize
4  void toHierarchizeVec( Eigen::ArrayXXd & p_toHierarchize )
```

The two classes have two specific member functions to hierarchize point by point a value function at given points in the sparse grid:

- the first work on a single function:

```
1  /// \brief Hierarchize certain points defined on sparse grids
2  /// Hierarchization is carried out point by point
3  /// \param p_nodalValues function to hierarchize
4  /// \param p_sparsePoints vector of sparse points to hierarchize (all
5  /// points must belong to the structure of the dataset)
6  /// \param p_hierarchized array of all hierarchical values (it is updated)
7  virtual void toHierarchizePByP(const Eigen::ArrayXd &p_nodalValues, const std::
    vector<SparsePoint> &p_sparsePoints, Eigen::ArrayXd &p_hierarchized) const
```

- the second work on a matrix, allowing to hierarchize several functions in a single call (each row corresponds to a function representation)

```
1  /// \brief Hierarchize certain points defined on the sparse grids for a set of
2  /// functions
3  /// The hierarchy is carried out point by point
4  /// \param p_nodalValues functions to hierarchize (the row corresponds to
5  /// the function number)
6  /// \param p_sparsePoints vector of sparse points to hierarchize (all
7  /// points must belong to the structure of the dataset)
8  /// \param p_hierarchized array of all hierarchical values (it is updated)
9  virtual void toHierarchizePByPVec(const Eigen::ArrayXXd &p_nodalValues, const std
    ::vector<SparsePoint> &p_sparsePoints, Eigen::ArrayXXd &p_hierarchized) const
```

The `SparsePoint` object is only a "typedef":

```

1 #define SparsePoint std::pair< Eigen::Array<char, Eigen::Dynamic, 1> , Eigen::Array<
   unsigned int, Eigen::Dynamic, 1> >

```

where the first array allows to store the multi-level associated with the point and the second the associated multi-index .

It is finally possible to hierarchize all the points associated with a multi level. As before, two methods are available:

- a first makes it possible to hierarchize all the points associated with a given level. The hierarchical values are updated with these new values.

```

1  /// \brief Hierarchize all points defined at a given level of the sparse grids
2  ///      Hierarchization is carried out point by point
3  /// \param p_nodalValues      function to hierarchize
4  /// \param p_iterLevel        iterator on the level of the point to
   hierarchize
5  /// \param p_hierarchized      array of all hierarchized values (it is updated)
6  virtual void toHierarchizePByPLevel(const Eigen::ArrayXd &p_nodalValues, const
   SparseSet::const_iterator &p_iterLevel, Eigen::ArrayXd &p_hierarchized) const

```

- the second permits to hierarchize different functions together

```

1  /// \brief Hierarchize all points defined on a given level of the sparse grids for
   a set of functions
2  ///      Hierarchization is performed point by point
3  /// \param p_nodalValues      function to hierarchize (the row corresponds to
   the function number)
4  /// \param p_iterLevel        iterator on the level of the point to hierarchize
5  /// \param p_hierarchized      array of all hierarchized values (it is updated)
6  virtual void toHierarchizePByPLevelVec(const Eigen::ArrayXXd &p_nodalValues, const
   SparseSet::const_iterator &p_iterLevel, Eigen::ArrayXXd &p_hierarchized)
   const

```

In the following example, sparse grids with boundary points is constructed. The values of a function f at each coordinates are stored in an array `valuesFunction`, storing the functions 2 to be interpolated. The 2 global functions are hierarchized (see the section above) in the array `hierarValues`, and then the interpolation can be performed using these hierarchized values.

```

1  ArrayXd lowValues = ArrayXd::Zero(5); // bottom of the grid
2  ArrayXd sizeDomain = ArrayXd::Constant(5,1.); // size of the grid
3  ArrayXd weight = ArrayXd::Constant(5,1.); // weights
4  int degree =1 ; // linear interpolator
5  level = 4 ; // level of the sparse grid
6
7  // sparse grid generation
8  SparseSpaceGridBound sparseGrid(lowValues, sizeDomain, level, weight, degree);
9
10 // grid iterators
11 shared_ptr<GridIterator > iterGrid = sparseGrid.getGridIterator();
12 ArrayXXd valuesFunction(2,sparseGrid.getNbPoints());
13 while (iterGrid->isValid())
14 {
15     ArrayXd pointCoord = iterGrid->getCoordinate();
16     valuesFunction(0,iterGrid->getCount()) = f(pointCoord) ;
17     valuesFunction(1,iterGrid->getCount()) = f(pointCoord)+1 ;
18     iterGrid->next();
19 }
20
21 // Hierarchize
22 ArrayXXd hieraValues =valuesFunction;

```

```

23     sparseGrid.toHierarchizeVec(hieraValues);
24
25     // interpolate
26     ArrayXd pointCoord = ArrayXd::Constant(5,0.66);
27     shared_ptr<Interpolator<double> > interpolator = sparseGrid.createInterpolator(
28         pointCoord);
29     ArrayXd interVal = interpolator->applyVec(hieraValues);

```

Remark 5 *The Point-by-point hierarchization on the global grid could have been calculated as below:*

```

1     std::vector<SparsePoint> sparsePoints(sparseGrid.getNbPoints());
2     std::shared_ptr<SparseSet> dataSet = sparseGrid.getDataSet();
3     // iterate on points
4     for (typename SparseSet::const_iterator iterLevel = dataSet->begin(); iterLevel !=
5         dataSet->end(); ++iterLevel)
6         for (typename SparseLevel::const_iterator iterPosition = iterLevel->second.begin();
7             iterPosition != iterLevel->second.end(); ++iterPosition)
8         {
9             sparsePoints[iterPosition->second] = make_pair(iterLevel->first, iterPosition->
10                 first);
11         }
12     ArrayXXd hieraValues = sparseGrid.toHierarchizePByPVec(valuesFunction, sparsePoints
13         );

```

In some cases, it is more convenient to build an interpolator acting on a global function. This is the case when you have only one function and you want to interpolate at several points for this function. In this specific case an interpolator deriving from the class `InterpolatorSpectral` (similarly to Legendre grid interpolators) can be constructed:

```

1     /** \brief Constructor taking in values on the grid
2     * \param p_grid is the sparse grid used to interpolate
3     * \param p_values Function values on the sparse grid
4     */
5     SparseInterpolatorSpectral(const shared_ptr< SparseSpaceGrid> &p_grid , const Eigen::
6         ArrayXd &p_values)

```

This class has a member to interpolate at a given point:

```

1     /** \brief interpolate
2     * \param p_point coordinates of the point for interpolation
3     * \return interpolated value
4     */
5     inline double apply(const Eigen::ArrayXd &p_point) const

```

See the 1.2 section for an example (similar but with Legendre grids) to use this object. Sometimes, we want to iterate on points on a given level. In the example below, for each level an iterator over all points belonging to a given level is retrieved and the values of a function f at each point are calculated and stored.

```

1     // sparse grid generation
2     SparseSpaceGridNoBound sparseGrid(lowValues, sizeDomain, p_level, p_weight, p_degree,
3         bPrepInterp);
4
5     // test iterator on each level
6     ArrayXd valuesFunctionTest(sparseGrid.getNbPoints());
7     std::shared_ptr<SparseSet> dataSet = sparseGrid.getDataSet();
8     for (SparseSet::const_iterator iterLevel = dataSet->begin(); iterLevel != dataSet->end
9         (); ++iterLevel)
10     {
11         // get back iterator on this level

```

```

10  shared_ptr<SparseGridIterator> iterGridLevel = sparseGrid.getLevelGridIterator(iterLevel
    );
11  while(iterGridLevel->isValid())
12  {
13      Eigen::ArrayXd pointCoord = iterGridLevel->getCoordinate();
14      valuesFunctionTest(iterGridLevel->getCount()) = f(pointCoord);
15      iterGridLevel->next();
16  }
17  }

```

Finally, adaptation can be realized with two member functions:

- A first one permits to refine adding points where the error is important. Note that a function is provided to calculate from hierarchical values the error at each level of the sparse grid and that a second one is provided to get a global error from the error calculated at each level. This makes it possible to specialize the refining depending, for example, whether the calculation is achieved for integration or interpolation purpose.

```

1  /// \brief Dimension adaptation nest
2  /// \param p_precision      precision required for adaptation
3  /// \param p_fInterpol      function to interpolate
4  /// \param p_phi            function for the error on a given level in the
    m_dataSet structure
5  /// \param p_phiMult        from an error defined at different levels, return a
    global error at the different levels
6  /// \param p_valuesFunction  an array storing the nodal values
7  /// \param p_hierarValues    an array storing hierarchized values (updated)
8  void refine(const double &p_precision, const std::function<double(const Eigen::
    ArrayXd &p_x)> &p_fInterpol,
9              const std::function< double(const SparseSet::const_iterator &, const
    Eigen::ArrayXd &)> &p_phi,
10             const std::function< double(const std::vector< double> &) > &p_phiMult
    ,
11             Eigen::ArrayXd &p_valuesFunction,
12             Eigen::ArrayXd &p_hierarValues);

```

with

- `p_precision` the η tolerance in the algorithm,
 - `p_fInterpol` the function permitting to calculate the nodal values,
 - `p_phi` function permitting to calculate e_l the local error for a given l ,
 - `p_phiMult` a function taking as argument all the e_l (local errors) and giving back the global error E ,
 - `p_valuesFunction` an array storing the nodal values (updated during refinement)
 - `p_hierarValues` an array storing the hierarchized values (updated during refinement)
- A second one enlarges the mesh, eliminating the point where the error is too small

```

1  /// \brief Dimension adaptation coarsening: modify the structure of the data by
    trying to delete al thel levels with local error
2  ///      below a local precision
3  /// \param p_precision      Precision under which coarsening will be realized
4  /// \param p_phi            function for the error on a given level in the
    m_dataSet structure
5  /// \param p_valuesFunction  an array storing the nodal values (modified on the
    new structure)

```

```

6      /// \param p_hierarValues   Hierarchical values on a data structure (modified on
      the new structure)
7      void coarsen(const double &p_precision, const std::function< double(const
      SparseSet::const_iterator &, const Eigen::ArrayXd &)> &p_phi,
8                  Eigen::ArrayXd &p_valuesFunction,
9                  Eigen::ArrayXd &p_hierarValues);

```

with arguments similar to the previous function.

1.3.6 Python API

Here is an example of the Python API used for interpolation with Sparse grids with boundary points and without boundary points. Adaptation and coarsening is available with an error calculated for interpolation only.

```

1  # Copyright (C) 2016 EDF
2  # All Rights Reserved
3  # This code is published under the GNU Lesser General Public License (GNU LGPL)
4  import numpy as np
5  import unittest
6  import random
7  import math
8  import pystopt.grids as StOptGrids
9
10 # function used
11 def funcToInterpolate( x):
12     return math.log(1. + x.sum())
13
14 # unit test for sparse grids
15 #####
16
17 class testGrids(unittest.TestCase):
18
19
20     # test sparse grids with boundaries
21     def testSparseGridsBounds(self):
22         # low values
23         lowValues = np.array([1.,2.,3.])
24         # size of the domain
25         sizeDomValues = np.array([3.,4.,3.])
26         # anisotropic weights
27         weights = np.array([1.,1.,1.])
28         # level of the sparse grid
29         level =3
30         # create the sparse grid with linear interpolator
31         sparseGridLin = StOptGrids.SparseSpaceGridBound(lowValues,sizeDomValues, level,
32                                                         weights,1)
33         iterGrid = sparseGridLin.getGridIterator()
34         # array to store
35         data = np.empty(sparseGridLin.getNbPoints())
36         # iterates on point
37         while( iterGrid.isValid()):
38             data[iterGrid.getCount()] = funcToInterpolate(iterGrid.getCoordinate())
39             iterGrid.next()
40         # Hierarchize the data
41         hierarData = sparseGridLin.toHierarchize(data)
42         # get back an interpolator
43         ptInterp = np.array([2.3,3.2,5.9],dtype=float)
44         interpol = sparseGridLin.createInterpolator(ptInterp)
45         # calculate interpolated value
46         interpValue = interpol.apply(hierarData)
47         print(("Interpolated value sparse linear" , interpValue))
48         # create the sparse grid with quadratic interpolator
49         sparseGridQuad = StOptGrids.SparseSpaceGridBound(lowValues,sizeDomValues, level,
50                                                         weights,2)

```

```

49     # Hierarchize the data
50     hierarData = sparseGridQuad.toHierarchize(data)
51     # get back an interpolator
52     ptInterp = np.array([2.3,3.2,5.9],dtype=float)
53     interpol = sparseGridQuad.createInterpolator(ptInterp)
54     # calculate interpolated value
55     interpValue = interpol.apply(hierarData)
56     print(("Interpolated value sparse quadratic " , interpValue))
57     # now refine
58     precision = 1e-6
59     print(("Size of hierarchical array " , len(hierarData)))
60     valueAndHierar = sparseGridQuad.refine(precision,funcToInterpolate,data,hierarData)
61     print(("Size of hierarchical array after refinement " , len(valueAndHierar[0])))
62     # calculate interpolated value
63     interpol1 = sparseGridQuad.createInterpolator(ptInterp)
64     interpValue = interpol1.apply(valueAndHierar[1])
65     print(("Interpolated value sparse quadratic after refinement " , interpValue))
66     # coarsen the grid
67     precision = 1e-4
68     valueAndHierarCoarsen = sparseGridQuad.coarsen(precision,valueAndHierar[0],
69                                                    valueAndHierar[1])
70     print(("Size of hierarchical array after coarsening " , len(valueAndHierarCoarsen
71                                                                    [0])))
72     # calculate interpolated value
73     interpol2 = sparseGridQuad.createInterpolator(ptInterp)
74     interpValue = interpol2.apply(valueAndHierarCoarsen[1])
75     print(("Interpolated value sparse quadratic after refinement " , interpValue))
76
77     # test sparse grids eliminating boundaries
78     def testSparseGridsNoBounds(self):
79         # low values
80         lowValues = np.array([1.,2.,3.],dtype=float)
81         # size of the domain
82         sizeDomValues = np.array([3.,4.,3.],dtype=float)
83         # anisotropic weights
84         weights = np.array([1.,1.,1.])
85         # level of the sparse grid
86         level =3
87         # create the sparse grid with linear interpolator
88         sparseGridLin = StOptGrids.SparseSpaceGridNoBound(lowValues,sizeDomValues, level,
89                                                            weights,1)
90         iterGrid = sparseGridLin.getGridIterator()
91         # array to store
92         data = np.empty(sparseGridLin.getNbPoints())
93         # iterates on point
94         while( iterGrid.isValid()):
95             data[iterGrid.getCount()] = funcToInterpolate(iterGrid.getCoordinate())
96             iterGrid.next()
97         # Hierarchize the data
98         hierarData = sparseGridLin.toHierarchize(data)
99         # get back an interpolator
100        ptInterp = np.array([2.3,3.2,5.9],dtype=float)
101        interpol = sparseGridLin.createInterpolator(ptInterp)
102        # calculate interpolated value
103        interpValue = interpol.apply(hierarData)
104        print(("Interpolated value sparse linear" , interpValue))
105        # create the sparse grid with quadratic interpolator
106        sparseGridQuad = StOptGrids.SparseSpaceGridNoBound(lowValues,sizeDomValues, level,
107                                                            weights,2)
108        # Hierarchize the data
109        hierarData = sparseGridQuad.toHierarchize(data)
110        # get back an interpolator
111        ptInterp = np.array([2.3,3.2,5.9],dtype=float)
112        interpol = sparseGridQuad.createInterpolator(ptInterp)
113        # calculate interpolated value
114        interpValue = interpol.apply(hierarData)
115        print(("Interpolated value sparse quadratic " , interpValue))
116        # test grids function

```

```

114     iDim = sparseGridQuad.getDimension()
115     pt = sparseGridQuad.getExtremeValues()
116     # now refine
117     precision = 1e-6
118     print(("Size of hierarchical array " , len(hierarData)))
119     valueAndHierar = sparseGridQuad.refine(precision,funcToInterpolate,data,hierarData)
120     print(("Size of hierarchical array after refinement " , len(valueAndHierar[0])))
121     # calculate interpolated value
122     interpol1 = sparseGridQuad.createInterpolator(ptInterp)
123     interpValue = interpol1.apply(valueAndHierar[1])
124     print(("Interpolated value sparse quadratic after coarsening " , interpValue))
125     # coarsen the grid
126     precision = 1e-4
127     valueAndHierarCoarsen = sparseGridQuad.coarsen(precision,valueAndHierar[0],
128                                                    valueAndHierar[1])
129     print(("Size of hierarchical array after coarsening " , len(valueAndHierarCoarsen
130                                                                [0])))
131     # calculate interpolated value
132     interpol2 = sparseGridQuad.createInterpolator(ptInterp)
133     interpValue = interpol2.apply(valueAndHierarCoarsen[1])
134     print(("Interpolated value sparse quadratic after coarsening " , interpValue))
135
136 if __name__ == '__main__':
137     unittest.main()

```

1.4 Full grids with refinement

This section provides a detailed overview of the full grid case in dimensions one and two, where grid refinement is permitted. These grids can be used to solve stock problems in dimensions one and two where grid refinement is possible. These grids have the particular property of being able to refine according to a criterion without conforming: a mesh can be refined locally, independently of the refinement of the mesh around it. On figure 1.15, a grid is given with (non conformed) local refinement.

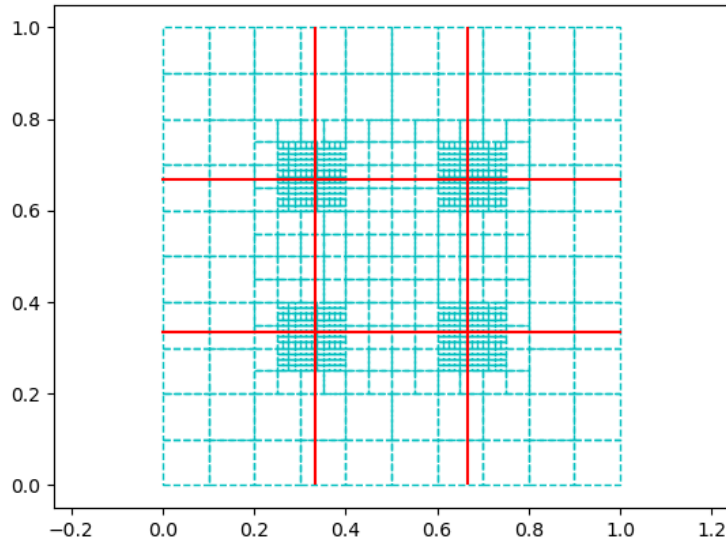


Figure 1.15: 2D grid refined

The grids begin with a regular grid (same space in each direction for the meshes) and, deciding a refinement, a mesh can be split. In 1D, the mesh is split in 2 meshes with same size. In dimension 2, the mesh is split in both direction giving 4 new meshes of same size.

1.4.1 C++ API

We only give the construction of a 2 dimensionnal grid. The 1D case is similar. A grid is composed of meshes.

Meshes

In 2D, a mesh is created associated a to grid:

```
1 Mesh2D( const double & p_xL,      const double & p_xR , const double & p_yB, const
        double & p_yT,
2      const int & p_verticeLT, const int & p_verticeLB, const int & p_verticeRB, const
        int & p_verticeRT)
```

where

- `p_xL` corresponds to the minimal x coordinate a point inside the mesh,
- `p_xR` corresponds to the maximal x coordinate a point inside the mesh,
- `p_yB` corresponds to the minimal y coordinate a point inside the mesh,
- `p_yT` corresponds to the maximal y coordinate a point inside the mesh,
- `p_verticeLT` correspond the point number in the grid it belongs to of the left top edge of the mesh.

- `p_verticeLB` correspond the point number in the grid it belongs to of the left bottom edge of the mesh.
- `p_verticeRB` correspond the point number in the grid it belongs to of the right bottom edge of the mesh.
- `p_verticeRT` correspond the point number in the grid it belongs to of the right top edge of the mesh .

A mesh can be split:

```
1  std::pair< std::array< std::shared_ptr< Mesh2D > ,4 > , std::vector<Eigen::Array<double,
    Eigen::Dynamic,1> > > > split(const std::vector< Eigen::ArrayXd > & p_vertices)
```

where

- `p_vertices` is the list of points coordinates of the grid it belong to
- The split method return 4 meshes resulting from split with coordinates of new points generated by the split.

Grid with adaptation

Three constructors of the `GridAdapt2D` are provided:

The first constructor

```
1  GridAdapt2D( const double & p_xMin , const double & p_xMax, const int & p_nbMeshX,
    const double & p_yMin, const double & p_yMax, const int & p_nbMeshY)
```

where

- `p_xMin` minimum x coordinate of the grid
- `p_xMax` maximum x coordinate of the grid
- `p_nbMeshX` number of meshes in x direction for the initial grid generation. Each mesh have the same width
- `p_yMin` minimum y coordinate of the grid
- `p_yMax` maximum y coordinate of the grid
- `p_nbMeshY` number of meshes in y direction for the initial grid generation. Each mesh have the same height.

A second constructor permits to create the initial grid as before and gives a set of points : the points correspond to coordinates (integer) of the edges of the meshes inside the initial grid.

```
1  GridAdapt2D( const double & p_xMin , const double & p_xMax, const int & p_nbMeshX,
    const double & p_yMin, const double & p_yMax, const int & p_nbMeshY, \
2  const std::vector< Eigen::ArrayXi > & p_points )
```

A last constructor adds a list of pair of mesh and position in the grid and the list of edges coordinates.

```

1  GridAdapt2D(const double & p_xMin , const double & p_xMax, const double & p_yMin, const
    double & p_yMax, \
2  const std::list< std::pair<std::shared_ptr< Mesh2D> , std::shared_ptr< std::vector<
    Eigen::Array<int, Eigen::Dynamic,1> > > > & p_meshes, const std::vector<
    Eigen::Array<double, Eigen::Dynamic,1> > > & p_points );

```

In the figure 1.16, mesh A is constructed with

```

1  Mesh2D(0.,2. , 0., 2., 3, 0, 1, 4)

```

and B with

```

1  Mesh2D(2. , 3., 2. , 3. , 10,4,9, 11)

```

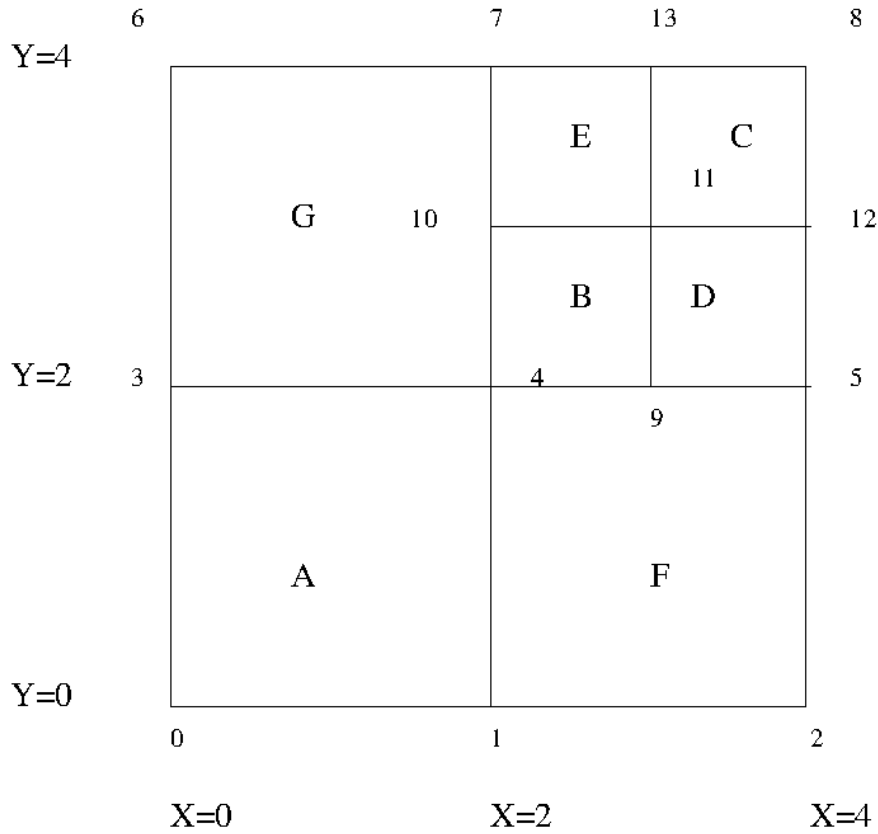


Figure 1.16: Exemple of grid refined

Constructing the grid with the last constructor, `p_meshes` is composed of

- a pair A and coordinates $([0, 0])$ as the mesh has the coordinate $[0, 0]$ in the initial grid of size 2×2 ,
- a pair B a mesh resulting from a refinement , and its coordinates $([1, 1], [0, 0])$ as it belong to an initial mesh M (resulting from the merge of meshes B, C, D, E) with coordinates $[1, 1]$ and its coordinates in M are $[0, 0]$.
- a pair C a mesh resulting from a refinement , and its coordinates $([1, 1], [1, 1])$,
- a pair D a mesh resulting from a refinement , and its coordinates $([1, 1], [1, 0])$,

- a pair E a mesh resulting from a refinement , and its coordinates $([1, 1], [0, 1])$,
- a pair F and coordinates $([1, 0])$
- a pair G and coordinates $([0, 1])$.

The list of points `p_points` is given by

$((0, 0), (2, 0), (4, 0), (0, 2), (2, 2), (4, 2), (0, 4), (2, 4), (4, 4), (3, 2), (2, 3), (3, 3), (4, 3), (3, 4))$

The `GridAdapt2D` has the following methods

```
1 void splitMesh( std::pair<std::shared_ptr< Mesh2D>, std::shared_ptr< std::vector< Eigen::ArrayXi > > > & p_meshToSplit);
```

where `p_meshToSplit` is a pair composed of the mesh to split and its coordinates in the grid (for example $([1, 1], [0, 0])$ are the coordinates of mesh B in the grid in figure 1.16). The method split the mesh in 4 meshes added in the grid, removing the old mesh which has been split.

```
1 std::list< std::pair<std::shared_ptr< Mesh2D>, std::shared_ptr< std::vector< Eigen::ArrayXi > > > > intersect(const double & p_xMin, const double & p_xMax, const double & p_yMin, const double & p_yMax) const
```

from an hypercube defined by its coordinates `p_xMin`, `p_xMax`, `p_yMin`, `p_yMax`, gives back the list of all the meshes in the grid (with coordinates of the meshes in the grid) with a non empty intersection with the hypercube.

```
1 shared_ptr< Mesh2D> getMeshWithPoint(const double &p_x, const double &p_y) const
```

from a points with coordinates (p_x, p_y) , the method gives back the mesh it belongs to.

Chapter 2

Introducing regression resolution

Suppose that the stochastic differential equation in the optimization problem is not controlled:

$$dX_s^{x,t} = b(s, X_s^{x,t})ds + \sigma(s, X_s^{x,t})dW_s$$

This case is for example encountered during the valuation of American options in finance, when an arbitration is carried out between the pay-off and the expected future gain if it is not exercised right now. In order to estimate this conditional expectation (according of Markov state), first suppose that a set of N Monte Carlo Simulation is available at dates t_i for a process $X_t := X_t^{0,x}$ where x is the initial state at date $t = 0$ and that we want to estimate $f(x) := \mathbb{E}[g(t+h, X_{t+h}) \mid X_t = x]$ for a given x and a given function g . This function f is located in the infinite dimensional space of the functions L_2 . In order to approximate it, we try to find it in a finite dimensional space. By choosing a set of basis functions ψ_k for $k = 1$ to M , the conditional expectation can be approximated by

$$f(x) \simeq \sum_{k=1}^M \alpha_k \psi_k(X_t) \quad (2.1)$$

where $(\hat{\alpha}_k^{t_i, N})_{k \leq M}$ minimizes

$$\sum_{\ell=1}^N \left| g(X_{t+h}^\ell) - \sum_{k=1}^M \alpha_k \psi_k(X_t^\ell) \right|^2 \quad (2.2)$$

over $(\alpha_k)_{k \leq M} \in \mathbb{R}^M$. We have to solve a quadratic optimization problem of the form

$$\min_{\alpha \in \mathbb{R}^M} \|A\alpha - B\|^2 \quad (2.3)$$

Classically the previous equation is reduced to the normal equation

$$A' A \alpha = A' B, \quad (2.4)$$

which is solved by a Cholesky-type approach when the matrix $A' A$ is defined otherwise the solution with the standard L_2 minimum can be calculated using the pseudo-inverse of $A' A$. When the different components of $X^{x,t}$ are strongly correlated it can be convenient to rotate

the dataset on its main components using the PCA method. Rotating the dataset before performing a regression has been recommended in [46] and [43] for example. The right-hand side of Figure 2.1 illustrates the new evaluation grid obtained on the same dataset. We can observe better coverage and fewer empty areas when using local regression that we will detail in this section.

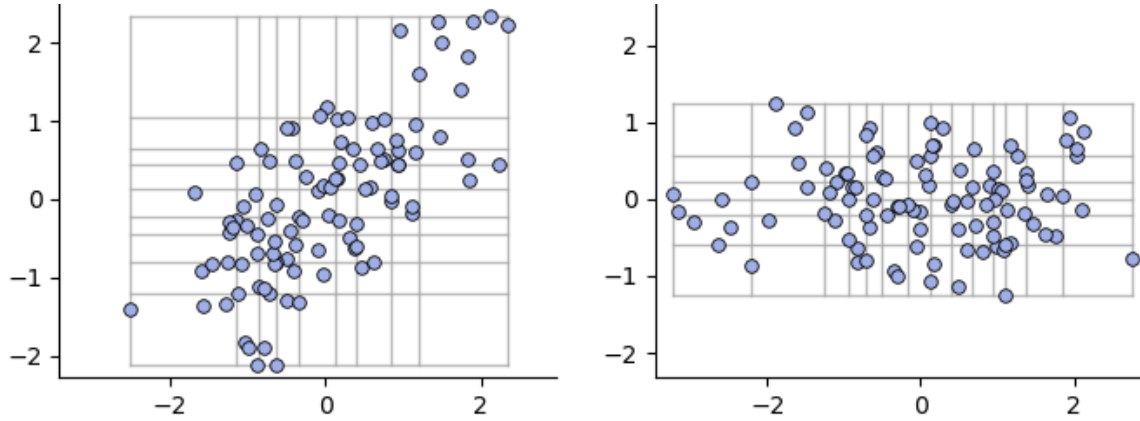


Figure 2.1: Evaluation grid: rotation

2.1 C++ global API

All regression classes are derived from the `BaseRegression` abstract class, which stores a pointer to the “particles” (a matrix storing the simulations of $X^{x,t}$: the first dimension of the matrix corresponds to the dimension of $X^{x,t}$, and the second dimension corresponds to the number of particle), and stores if the current date t is 0 (then the conditional expectation is only an expectation).

```

1 // Copyright (C) 2016, 2017 EDF
2 // All Rights Reserved
3 // This code is published under the GNU Lesser General Public License (GNU LGPL)
4 #ifndef BASEREGRESSION_H
5 #define BASEREGRESSION_H
6 #include <memory>
7 #include <vector>
8 #include <iostream>
9 #include <utility>
10 #include <type_traits>
11 #include <Eigen/Dense>
12 #include <Eigen/SVD>
13 #include "StOpt/core/grids/InterpolatorSpectral.h"
14 #include "StOpt/core/grids/LinearInterpolatorSpectral.h"
15
16 /** \file BaseRegression.h
17  * \brief Base class to define regressor for stochastic optimization by Monte Carlo
18  * \author Xavier Warin
19  */
20 namespace StOpt
21 {
22
23 /// \class BaseRegression BaseRegression.h
24 /// Base class for regression
25 class BaseRegression
26 {

```

```

27 protected :
28
29     bool m_bZeroDate ; //< Is the regression date zero ?
30     bool m_bRotationAndRescale ; //< do we rescale particles and do a rotation with SVD on
    data
31     Eigen::ArrayXd m_meanX ; //< store scaled factor in each direction (average of
    particles values in each direction)
32     Eigen::ArrayXd m_etypX ; //< store scaled factor in each direction (standard
    deviation of particles in each direction)
33     Eigen::MatrixXd m_svdMatrix ; //< svd matrix transposed used to transform particles
34     Eigen::ArrayXd m_sing ; //< singular values associated to SVD
35     Eigen::ArrayXXd m_particles ; //< Particles used to regress: first dimension :
    dimension of the problem , second dimension : the number of particles. These
    particles are rescaled and a rotation with SVD is achieved to avoid degeneracy in
    case of high correlations
36
37     // rotation for data and rescaling
38     void preProcessData();
39
40 public :
41
42     /// \brief Default constructor
43     BaseRegression();
44
45     /// \brief Default destructor
46     virtual ~BaseRegression() {}
47
48     /// \brief Default constructor
49     BaseRegression(const bool &p_bRotationAndRescale);
50
51     /// \brief Constructor storing the particles
52     /// \param p_bZeroDate first date is 0?
53     /// \param p_particles particles used for the meshes.
54     /// First dimension : dimension of the problem,
55     /// second dimension : the number of particles
56     /// \param p_bRotationAndRescale do we rescale particle
57     BaseRegression(const bool &p_bZeroDate, const Eigen::ArrayXXd &p_particles, const bool
    &p_bRotationAndRescale);
58
59     /// \brief Constructor used in simulation, no rotation
60     /// \param p_bZeroDate first date is 0?
61     /// \param p_bRotationAndRescale do we rescale particle
62     BaseRegression(const bool &p_bZeroDate, const bool &p_bRotationAndRescale);
63
64
65     /// \brief Last constructor used in simulation
66     /// \param p_bZeroDate first date is 0?
67     /// \param p_meanX scaled factor in each direction (average of particles
    values in each direction)
68     /// \param p_etypX scaled factor in each direction (standard deviation of
    particles in each direction)
69     /// \param p_svdMatrix svd matrix transposed used to transform particles
70     /// \param p_bRotationAndRescale do we rescale particle
71
72     BaseRegression(const bool &p_bZeroDate, const Eigen::ArrayXd &p_meanX, const Eigen
    ::ArrayXd &p_etypX, const Eigen::MatrixXd &p_svdMatrix, const bool &
    p_bRotationAndRescale);
73
74     /// \brief Copy constructor
75     /// \param p_object object to copy
76     BaseRegression(const BaseRegression &p_object);
77
78     /// \brief update the particles used in regression and construct the matrices
79     /// \param p_bZeroDate first date is 0?
80     /// \param p_particles particles used for the meshes.
81     /// First dimension : dimension of the problem,
82     /// second dimension : the number of particles
83     void updateSimulationsBase(const bool &p_bZeroDate, const Eigen::ArrayXXd &p_particles)
    ;

```

```

84
85     /// \brief Get some local accessors
86     ///@{
87     virtual inline Eigen::ArrayXXd getParticles() const
88     {
89         return m_particles ;
90     }
91
92     /// \brief Get bRotationAndRescale
93     virtual inline bool getBRotationAndRescale() const
94     {
95         return m_bRotationAndRescale ;
96     }
97
98     /// \brief Get average of simulation per dimension
99     virtual inline Eigen::ArrayXd getMeanX() const
100    {
101        return m_meanX;
102    }
103
104    /// \brief get standard deviation per dimension
105    virtual inline Eigen::ArrayXd getEtypX() const
106    {
107        return m_etyX;
108    }
109
110    /// \brief get back the SVD matrix used for rescaling particles
111    virtual inline Eigen::MatrixXd getSvdMatrix() const
112    {
113        return m_svdMatrix;
114    }
115
116    /// \brief get back singular values
117    virtual inline Eigen::ArrayXd getSing() const
118    {
119        return m_sing;
120    }
121
122    /// \brief Get dimension of the problem
123    virtual inline int getDimension() const
124    {
125        return m_particles.rows();
126    }
127
128    /// \brief Get the number of simulations
129    virtual inline int getNbSimul() const
130    {
131        return m_particles.cols() ;
132    }
133
134    /// \brief get back particle by its number
135    /// \param p_iPart particle number
136    /// \return the particle (if no particle, send back an empty array)
137    virtual Eigen::ArrayXd getParticle(const int &p_iPart) const;
138
139    /// \brief get the number of basis functions
140    virtual int getNumberOfFunction() const = 0 ;
141
142    ///@}
143    /// \brief Constructor storing the particles
144    /// \brief update the particles used in regression and construct the matrices
145    /// \param p_bZeroDate first date is 0?
146    /// \param p_particles particles used for the meshes.
147    ///
148    /// First dimension : dimension of the problem,
149    /// second dimension : the number of particles
150    virtual void updateSimulations(const bool &p_bZeroDate, const Eigen::ArrayXXd &
151        p_particles) = 0 ;
152
153    /// \brief conditional expectation basis function coefficient calculation

```

```

152     /// \param p_fToRegress function to regress associated to each simulation used in
153     /// \return regression coordinates on the basis (size : number of meshes multiplied by
154     /// @{
155     virtual Eigen::ArrayXd getCoordBasisFunction(const Eigen::ArrayXd &p_fToRegress) const
156     = 0;
157     ///@}
158     /// \brief conditional expectation basis function coefficient calculation for multiple
159     /// \param p_fToRegress function to regress associated to each simulation used in
160     /// \return regression coordinates on the basis (size : number of function to regress
161     /// \times number of meshes multiplied by the dimension plus one)
162     /// @{
163     virtual Eigen::ArrayXXd getCoordBasisFunctionMultiple(const Eigen::ArrayXXd &
164     p_fToRegress) const = 0 ;
165     ///@}
166     /// \brief conditional expectation calculation
167     /// \param p_fToRegress simulations to regress used in optimization
168     /// \return regressed value function
169     /// @{
170     virtual Eigen::ArrayXd getAllSimulations(const Eigen::ArrayXd &p_fToRegress) const = 0;
171     virtual Eigen::ArrayXXd getAllSimulationsMultiple(const Eigen::ArrayXXd &p_fToRegress)
172     const = 0;
173     ///@}
174     /// \brief Use basis functions to reconstruct the solution
175     /// \param p_basisCoefficients basis coefficients
176     /// @{
177     virtual Eigen::ArrayXd reconstruction(const Eigen::ArrayXd &p_basisCoefficients)
178     const = 0 ;
179     virtual Eigen::ArrayXXd reconstructionMultiple(const Eigen::ArrayXXd &
180     p_basisCoefficients) const = 0;
181     /// @}
182     /// \brief use basis function to reconstruct a given simulation
183     /// \param p_isim simulation number
184     /// \param p_basisCoefficients basis coefficients to reconstruct a given conditional
185     /// expectation
186     virtual double reconstructionASim(const int &p_isim, const Eigen::ArrayXd &
187     p_basisCoefficients) const = 0 ;
188     /// \brief conditional expectation reconstruction
189     /// \param p_coordinates coordinates to interpolate (uncertainty sample)
190     /// \param p_coordBasisFunction regression coordinates on the basis (size: number of
191     /// meshes multiplied by the dimension plus one)
192     /// \return regressed value function reconstructed for each simulation
193     virtual double getValue(const Eigen::ArrayXd &p_coordinates,
194     const Eigen::ArrayXd &p_coordBasisFunction) const = 0;
195     /// \brief For autodiff
196     template<typename T>
197     T getValueAutoDiff(const Eigen::ArrayXd &p_coordinates,
198     const Eigen::Array<T, Eigen::Dynamic, 1> &p_coordBasisFunction)
199     const
200     {
201     if constexpr (std::is_same_v<T, double>)
202     return static_cast<T>(getValue(p_coordinates, p_coordBasisFunction.template cast<double>
203     >()));
204     }
205     // template<typename T>
206     // T getValueAutoDiff(const Eigen::ArrayXd &p_coordinates,
207     // const Eigen::Array<T, Eigen::Dynamic, 1> &p_coordBasisFunction)
208     // const
209     // {

```

```

204 // if constexpr (std::is_same_v<T,double>)
205 // {
206 //     return getValue(p_coordinates, p_coordBasisFunction);
207 // }
208 // else
209 // {
210 //     if (auto ptr = dynamic_cast<const LocalLinearRegression*>(this))
211 //     {
212 //         return ptr->template getValueAutoDiff<T>(p_coordinates, p_coordBasisFunction);
213 //     }
214 //
215 //     //
216 //     std::cout<< "Auto differentiaon not availabel for this type" << std::endl ;
217 //     abort();
218 // }
219 // }
220
221 /// \brief conditional expectation reconstruction for a lot of simulations
222 /// \param p_coordinates coordinates to interpolate (uncertainty sample) size
223 /// \param p_coordBasisFunction regression coordinates on the basis (size: number of
224 /// meshes multiplied by the dimension plus one)
225 /// \return regressed value function reconstructed for each simulation
226 Eigen::ArrayXd getValues(const Eigen::ArrayXd &p_coordinates,
227                          const Eigen::ArrayXd &p_coordBasisFunction) const
228 {
229     Eigen::ArrayXd valRet(p_coordinates.cols());
230     for (int is = 0; is < p_coordinates.cols(); ++is)
231     {
232         const Eigen::ArrayXd coordinates = p_coordinates.col(is);
233         valRet(is) = getValue(coordinates, p_coordBasisFunction);
234     }
235     return valRet;
236 }
237
238 /// \brief permits to reconstruct a function with basis functions coefficients values
239 /// given on a grid
240 /// \param p_coordinates coordinates (uncertainty sample)
241 /// \param p_ptOfStock grid point
242 /// \param p_interpFuncBasis spectral interpolator to interpolate the basis
243 /// functions coefficients used in regression on the grid (given for each basis
244 /// function)
245 virtual double getAValue(const Eigen::ArrayXd &p_coordinates, const Eigen::ArrayXd &
246                         p_ptOfStock,
247                         const std::vector< std::shared_ptr<InterpolatorSpectral<double>
248                         >> &p_interpFuncBasis) const = 0;
249
250 // template<typename T>
251 // T getAValueAutoDiff(const Eigen::ArrayXd &p_coordinates,
252 //                     const Eigen::Array<T, Eigen::Dynamic, 1> &p_ptOfStock,
253 //                     const std::vector< std::shared_ptr<InterpolatorSpectral<T>> > &
254 //                     p_interpFuncBasis) const
255 // {
256 //     std::cout << "Methode not implemented for this regressor" << std::endl ;
257 //     abort();
258 //     return T(0);
259 // }
260
261 /// \brief is the regression date zero
262 inline bool getBZeroDate() const
263 {
264     return m_bZeroDate;
265 }
266
267 /// \brief Clone the regressor
268 virtual std::shared_ptr<BaseRegression> clone() const = 0 ;
269
270 };

```

```

265
266 }
267
268 #endif

```

All regression classes share the same constructors:

- a first constructor stores the members of the class and calculates the matrices for the regression: it is used for example to build a regression object at each time step of a resolution method,
- the second constructor is used to prepare data which will be shared by all future regressions. It must be used with the `updateSimulation` method to update the actual matrix construction. In a resolution method with many time steps, the object will be constructed only once and at each time step, the Markov state will be updated by the method `updateSimulation`.

All regression classes share the common methods:

- `updateSimulationBase` (see above),
- `getCoordBasisFunction` takes the values $g(t+h, X_{t+h})$ for all simulations and returns the coefficients α_k of the basis functions,
- `getCoordBasisFunctionMultiple` is used if we want to do the previous calculation on several g functions in a single call. In the matrix given as an argument, the first dimension has a size equal to the number of Monte Carlo simulations, while the second dimension has a size equal to the number of functions to regress. At output, the first dimension has a size equal to the number of functions to regress and the second equal to the number of basis functions.
- `getAllSimulations` takes the values $g(t+h, X_{t+h})$ for all simulations and returns the regressed values for all simulations $f(X_t)$
- `getAllSimulationsMultiple` is used if we want to do the previous calculation on several g functions in a single call. In the matrix given as an argument, the first dimension has a size equal to the number of Monte Carlo simulations, while the second dimension has a size equal to the number of functions to regress. The regressed values are rendered in the same format.
- `reconstruction` takes as input the coefficient α_k of the basis functions and returns all $f(X_t)$ for the stored simulations by applying the equation (2.1).
- `reconstructionMultiple` is used if we want to do the previous calculation on several g functions in a single call. As input the coefficients α_k of the basis functions are given (number of function to regress for the first dimension, number of basis functions for second dimension). Consequently, the $f(X_t)$ for all the simulations and all the functions f are returned (number of Monte Carlo simulations in the first dimension, number of functions to regress in the second dimension).

- **reconstructionASim** takes a number of simulations $isim$ (optimization part) and α_k coefficient of the basis functions as input and returns $f(X_t^{isim})$ by applying the equation (2.1),
- **getValue** takes as its first argument samples of X_t , the basis function α_k and reconstructs the regressed solution of the equation (2.1).
- **getValues** takes as its first argument samples of X_t (dimension of uncertainty by number of samples), the basis function α_k and reconstructs the regressed solution of the equation (2.1) (an array).

2.2 Adapted local polynomial basis with same probability

The description of the method and its properties can be found in [9]. We simply recall the methodology. These adapted local methods can benefit from a rotation in its main axis through the PCA method. Rotation is activated by a flag in the objects builder.

2.2.1 Description of the method

The method essentially consists in applying a non-conforming finite element approach rather than a spectral type method as presented above.

The idea is to use, at each time step t_i , a set of functions $\psi_q, q \in [0, M_M]$ with local hyper cube support $D_{i_1, i_2, \dots, i_d}$ where $i_j = 1$ to I_j , $M_M = \prod_{k=1, d} I_k$, and $\{D_{i_1, \dots, i_d}\}_{(i_1, \dots, i_d) \in [1, I_1] \times \dots \times [1, I_d]}$ is a partition of $[\min_{k=1, N} X_{t_i}^{1, (k)}, \max_{k=1, N} X_{t_i}^{1, (k)}] \times \dots \times [\min_{k=1, N} X_{t_i}^{d, (k)}, \max_{k=1, N} X_{t_i}^{d, (k)}]$. On each $D_l, l = (i_1, \dots, i_d)$, depending on the selected method, ψ_l is

- either a constant function, so the global number of degrees of freedom is equal to M_M ,
- or a linear function with $1 + d$ degrees of freedom, so the global number of degrees of freedom is equal to $M_M * (1 + d)$.

This approximation is "non conform" in the sense that we do not ensure the continuity of the approximation. However, it has the advantage of being able to adapt to any function, even discontinuous. In order to avoid oscillations and to allow classical regression by the Cholesky method, the supports are chosen so as to contain roughly the same number of particles.

On the figure 2.2, we have drawn an example of supports in the case of $16 = 4 \times 4$ local base cells, in the dimension 2.

Sometimes we can further leverage knowledge about the value of continuation. In the case of an American basket option for example, we have a convexity of this continuation value as a function of the underlying price. It is possible to modify the previous algorithm to try to impose that the numerical method repeats this convexity. The algorithm in [33] has been optionally implemented. This algorithm may not converge when used in multi-dimensional but it allows to improve the convexity of the solution by iterating several times.

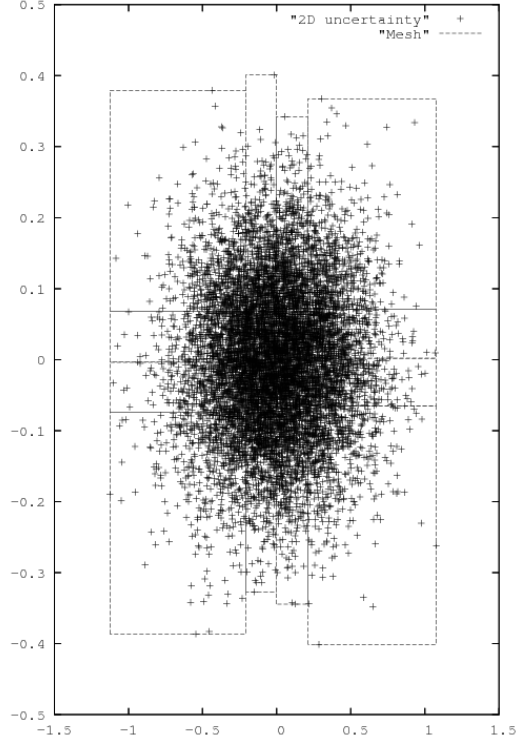


Figure 2.2: Support of 2D function basis

2.2.2 C++ API

The constant per cell approximation

The constructor of the local constant regression object is obtained by

```
1 LocalConstRegression(const Eigen::ArrayXi &p_nbMesh, bool p_bRotationAndRecale =
   false);
```

where:

- **p_nbMesh** is an array giving the number of meshes used in each direction ((4,4) for the figure 2.2 for example).
- **p_bRotationAndRecale** is an optional argument defined as **false** by default which means that no data rotation in its axis of main components is performed. In the case of rotation, the directions are sorted with their decreasing singular values and the number of meshes in **p_nbMesh** is defined for these sorted directions: **p_nbMesh(0)** is associated with first direction with the highest singular value, **p_nbMesh(1)** with the direction associated with the second highest singular value etc.

The second constructor allows to build the regression matrix,

```
1 LocalConstRegression(const bool &p_bZeroDate,
2   const shared_ptr< ArrayXXd> &p_particles,
3   const Eigen::ArrayXi &p_nbMesh,
4   bool p_bRotationAndRecale = false)
```

where

- `p_bZeroDate` is `true` if the regression date is 0,
- `p_particles` the particles X_t for all simulations (dimension of X_t for first dimension, number of Monte Carlo simulations in second dimension),
- `p_nbMesh` is an array giving the number of meshes used in each direction (4,4) for the figure 2.2,
- `p_bRotationAndRecalc` is an optional argument defined as `false` by default which means that no data rotation in its axis of main components is performed. In the case of rotation, the directions are sorted with their decreasing singular values and the number of meshes in `p_nbMesh` is defined for these sorted directions: `p_nbMesh(0)` is associated with first direction with the highest singular value, `p_nbMesh(1)` with the direction associated with the second highest singular value etc.

Linear approximation per cell

The constructor of the local linear regression object is obtained by

```
1 LocalLinearRegression(const Eigen::ArrayXi &p_nbMesh, bool p_bRotationAndRecalc =
   false);
```

where

- `p_nbMesh` is an array giving the number of meshes used in each direction ((4,4) for the figure 2.2 for example),
- `p_bRotationAndRecalc` is an optional argument defined as `false` by default which means that no data is rotated in its main components axis. In the case of rotation, the directions are sorted with their decreasing singular values and the number of meshes in `p_nbMesh` is defined for these sorted directions: `p_nbMesh(0)` is associated with the first direction with the highest singular value, `p_nbMesh(1)` with the direction associated with the second highest singular value, etc.

The second constructor allows to build the regression matrix,

```
1 LocalLinearRegression(const bool &p_bZeroDate,
2     const shared_ptr< ArrayXXd> &p_particles,
3     const Eigen::ArrayXi &p_nbMesh,
4     bool p_bRotationAndRecalc = false)
```

where

- `p_bZeroDate` is `true` if the regression date is 0,
- `p_particles` the particles X_t for all simulations (dimension of X_t for the first dimension, number of Monte Carlo simulations in the second dimension),
- `p_nbMesh` is an array giving the number of meshes used in each direction (4,4) for the figure 2.2

- `p_bRotationAndRecalc` is an optional argument defined as `false` by default which means that no data is rotated in its main components axis. In the case of rotation, the directions are sorted with their decreasing singular values and the number of meshes in `p_nbMesh` are defined for these sorted directions: `p_nbMesh(0)` is associated with first direction with the highest singular value, `p_nbMesh(1)` with the direction associated with the second highest singular value etc.

This class can benefit from the methodology in [33] implementing a generalization of the member function `getAllSimulations`:

```
1 Eigen::ArrayXd getAllSimulationsConvex(const Eigen::ArrayXd &p_fToRegress, const int &
  p_nbIterMax)
```

where

- `p_fToRegress` is the set of points that we want to regress while preserving the convexity of the regressed function value
- `p_nbIterMax` is the maximum number of iterations of the method.

It returns the regressed values for all the simulations of the uncertainties.

An example in the linear case

Below, we give a small example where `toRegress` corresponds to $g(t + h, X_{t+h})$ for all simulations and `x` store X_t for all simulations.

```
1 // create the mesh for a 2 dim problem, 4 meshes per direction
2 ArrayXi nbMesh = ArrayXi::Constant(2, 4);
3 // t is not zero
4 bool bZeroDate = 0;
5 // constructor, no rotation of the data
6 LocalLinearRegression localRegressor(nbMesh);
7 // update particles values
8 localRegressor.updateSimulations(bZeroDate, x);
9 // regressed values
10 ArrayXd regressedValues = localRegressor.getAllSimulations(toRegress);
```

2.2.3 Python API

Here is a similar example using the second constructor of the linear case

```
1 import pystopt.regression as StOptReg
2 nbSimul = 5000000;
3 np.random.seed(000)
4 x = np.random.uniform(-.,1.,size=(1,nbSimul));
5 # real function
6 toReal = (2+x[0,:]+(+x[0,:])*(1+x[0,:]))
7 # function to regress
8 toRegress = toReal + 4*np.random.normal(0.,nbSimul)
9 # mesh
10 nbMesh = np.array([6],dtype=np.int32)
11 # Regressor without rotation of data
12 regressor = StOptReg.LocalLinearRegression(False,x,nbMesh)
13 y = regressor.getAllSimulations(toRegress).transpose()[0]
```

Of course the constant per cell case in Python is similar. As in C++, the linear case allows us to try to regress while preserving the convexity using the `getAllSimulationsConvex` method.

2.3 Adapted local basis by K-Means clustering methods

This method can be interesting when a small number of particles is available to calculate the regressions and we propose a K-Means clustering method to cluster simulations together. The classical K-Means clustering method is as follows: N points X^k with $k = 1$ to N are given. A partition of the domain $S = (S_m)_{m \leq p}$ with $p \leq N$ domains is obtained by minimizing

$$\arg \min_S \sum_{k=1}^p \sum_{X^j \in S_k} \|X^j - \mu_k\|^2,$$

where μ_k is the barycenter of all points $X^j \in S_k$.

The classic Lloyd algorithm is used to calculate the cluster:

Algorithm 3 Lloyd algorithm

- 1: Choose p points as initialization for μ_k^1 , $k = 1, p$
- 2: **while** Not converged **do**
- 3: affect each particle to its Voronoi cell:

$$S_k^l = \{X^i : \|X^i - \mu_k^l\| \leq \|X^i - \mu_m^l\|, m = 1, p\}$$

- 4: Update

$$\mu_k^{l+1} = \frac{1}{|S_k^l|} \sum_{X^i \in S_k^l} X^i$$

- 5: **end while**
-

This algorithm is effective in 1D by sorting the coordinates of the particles. Its extension in the general case is costly due to the calculation of the Voronoi cells and the distance between all points.

We suppose that, as in the adaptive case with same probability, we want to have a partition such that the number of meshes in each direction is I_k for $k = 1, d$.

We propose a recursive algorithm to calculate the meshes. This algorithm is given by 4 and 5,

Algorithm 4 Recursive 1D Lloyd algorithm

```
1: procedure RECURKMEANS( $id, X_{i_1, \dots, i_{id-1}}, S_{i_1, \dots, i_{id-1}}$ )
2:   Sort the particle in dimension  $id$  and use Lloyd algorithm to partition  $S_{i_1, \dots, i_{id-1}}$  in
   the dimension  $id$  and get  $S^{i_1, \dots, i_{id}}, i_{id} = 1, I_{id}$ .
3:   for  $i_{id} = 1, I_{id}$  do
4:      $X_{i_1, \dots, i_{id}} = \{X \in X_{i_1, \dots, i_{id-1}} / X \in S_{i_1, \dots, i_{id}}\}$ 
5:     if  $id < d$  then
6:       RecurKMeans( $id + 1, X_{i_1, \dots, i_{id}}, S_{i_1, \dots, i_{id}}$ )
7:     end if
8:   end for
9: end procedure
```

Algorithm 5 Modified Lloyd algorithm

```
1:  $S = \mathbb{R}^d, X = \{X^i / i = 1, N\}$ 
2: RecurKMeans(1,  $X, S$ )
```

and an example of a resulting partition is given on figure 2.3 in 2D for $I_1 = 3, I_2 = 4$.

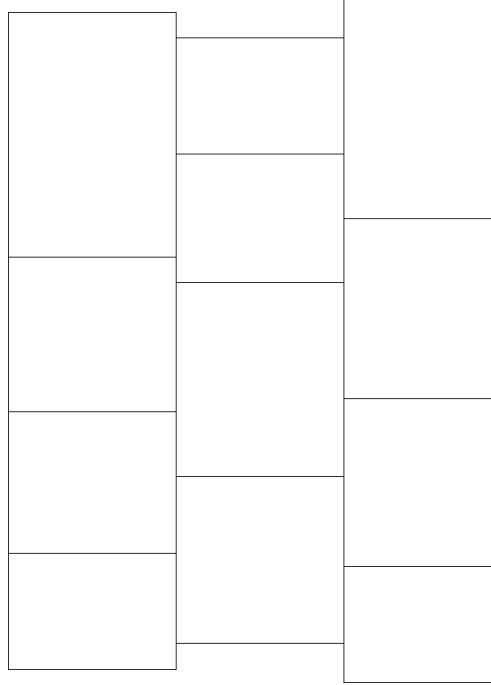


Figure 2.3: Possible 2D partition due to modified K-Means algorithm

Once the partition has been performed, a regression is performed with a constant representation per mesh:

We note $(S_k^i)_{k=1,p}$, a partition with the above method at the date t_i using Monte Carlo particles $X_{t_i}^{(k)}$, $k = 1, N$. To calculate the conditional expectation of a function g of $X_{t_{i+1}}$,

we use the constant representation by mesh and we then have:

$$\mathbb{E}[g(X_{t_{i+1}})/X_{t_i} = X_{t_i}^k] \simeq \frac{1}{|S_p^i|} \sum_{j/X_{t_i}^j \in S_p^i} g(X_{t_{i+1}}^j)$$

where $X_{t_{i+1}}^k \in |S_p^i|$.

2.3.1 C++ API

The constructor of the local K-Means regression object is similar to that of the LocalConstRegression object in the 2.2.2 section. We do not then recall the signification of all the arguments.

A first constructor is:

```
1 LocalKMeansRegression(const Eigen::ArrayXi &p_nbMesh, bool p_bRotationAndRecalc =
   false);
```

and the second constructor permits to build the regression matrix,

```
1 LocalKMeansRegression(const bool &p_bZeroDate,
2   const shared_ptr< ArrayXXd> &p_particles,
3   const Eigen::ArrayXi &p_nbMesh,
4   bool p_bRotationAndRecalc = false)
```

2.3.2 Python api

Since the C++ API is the same as that of the LocalConstRegression object, we have the same Python binding as in the 2.2.3 section.

2.4 Local polynomial basis with meshes of same size

In certain cases, instead of using adapted meshes, one can prefer to fix the mesh with a constant step in each direction with I_k meshes in each direction so that the total number of cells is $M_M = \prod_{k=1,d} I_k$. On each cell as in section 2.2, one can have two approximations:

- either a linear per mesh representation, so the global number of degrees of freedom is equal to M_M ,
- or a linear function with $1 + d$ degrees of freedom, so the global number of degrees of freedom is equal to $M_M * (1 + d)$.

Because we define in each direction, the domain for the local basis, we do not use any data rotation.

2.4.1 C++ API

The constant per cell approximation

The constructor of the local constant regression object is achieved by

```

1 LocalSameSizeConstRegression(const Eigen::ArrayXd &p_lowValues, const Eigen::ArrayXd &
  p_step, const Eigen::ArrayXi &p_nbStep);

```

- `p_lowValues` is an array giving the first point of the grid in each direction,
- `p_step` is an array giving the size of the meshes in each direction,
- `p_nbStep` is an array giving the number of meshes used in each direction.

The second constructor allows to build the regression matrix,

```

1 LocalSameSizeConstRegression(const bool &p_bZeroDate,
2                               const std::shared_ptr< Eigen::ArrayXXd > &p_particles,
3                               const Eigen::ArrayXd &p_lowValues,
4                               const Eigen::ArrayXd &p_step,
5                               const Eigen::ArrayXi &p_nbStep);

```

where

- `p_bZeroDate` is `true` if the regression date is 0,
- `p_particles` the particles X_t for all simulations (dimension of X_t for first dimension, number of Monte Carlo simulations in second dimension),
- `p_lowValues` is an array giving the first point of the grid in each direction,
- `p_step` is an array giving the size of the meshes in each direction,
- `p_nbStep` is an array giving the number of meshes used in each direction.

The linear per cell approximation

The constructor of the local linear regression object is achieved by

```

1 LocalSameSizeLinearRegression(const Eigen::ArrayXd &p_lowValues, const Eigen::ArrayXd
  &p_step, const Eigen::ArrayXi &p_nbStep);

```

where

- `p_lowValues` is an array giving the first point of the grid in each direction,
- `p_step` is an array giving the size of the meshes in each direction,
- `p_nbStep` is an array giving the number of meshes used in each direction.

The second constructor allows to build the regression matrix,

```

1 LocalSameSizeLinearRegression(const bool &p_bZeroDate,
2                               const std::shared_ptr< Eigen::ArrayXXd > &p_particles,
3                               const Eigen::ArrayXd &p_lowValues,
4                               const Eigen::ArrayXd &p_step,
5                               const Eigen::ArrayXi &p_nbStep);

```

where

- `p_bZeroDate` is `true` if the regression date is 0,

- `p_particles` the particles X_t for all simulations (dimension of X_t for first dimension, number of Monte Carlo simulations in second dimension),
- `p_lowValues` is an array giving the first point of the grid in each direction,
- `p_step` is an array giving the size of the meshes in each direction,
- `p_nbStep` is an array giving the number of meshes used in each direction.

2.4.2 An example in the linear case

Below we give a small example where `toRegress` is the array to regress as a function of `x` an array “`x`” in dimension `p_nDim`:

```

1  // create a random ‘x’ array
2  shared_ptr<ArrayXXd> x(new ArrayXXd(ArrayXXd::Random(p_nDim, p_nbSimul)));
3  // create the mesh by getting min and max value on the samples
4  double xMin = x->minCoeff() - tiny;
5  double xMax = x->maxCoeff() + tiny;
6  ArrayXd lowValues = ArrayXd::Constant(p_nDim, xMin);
7  ArrayXd step = ArrayXd::Constant(p_nDim, (xMax - xMin) / p_nMesh);
8  ArrayXi nbStep = ArrayXi::Constant(p_nDim, p_nMesh);
9  // constructor
10 LocalLinearRegression localRegressor(lowValues, step, nbStep);
11 // update particles values
12 localRegressor.updateSimulations(bZeroDate, x);
13 // regressed values
14 ArrayXd regressedValues = localRegressor.getAllSimulations(toRegress);

```

2.4.3 Python API

Here is a similar example using the second constructor of the linear case

```

1  import pystopt.regression as StOptReg
2  nbSimul = 5000000;
3  np.random.seed(000)
4  x = np.random.uniform(-.,1.,size=(1,nbSimul));
5  # real function
6  toReal = (2+x[0,:]+(+x[0,:])*(1+x[0,:]))
7  # function to regress
8  toRegress = toReal + 4*np.random.normal(0.,nbSimul)
9  # mesh
10 nStep = 20
11 lowValue = np.array([-1.0001],dtype=np.float)
12 step = np.array([2.0002/nStep],dtype=np.float)
13 nbMesh = np.array([nStep],dtype=np.int32)
14 # Regressor
15 regressor = StOptReg.LocalSameSizeLinearRegression(False,x,lowValue,step,nbMesh)
16 y = regressor.getAllSimulations(toRegress).transpose()[0]

```

Of course, the constant per cell in Python is similar.

2.5 Sparse grid regressor

In the case of a sparse regressor, the grid is an object `SparseSpaceGridNoBound` (extrapolation for the boundary conditions). The basis functions are given by the section 1.3 for a linear, quadratic or cubic function base. No data rotation is available.

2.5.1 C++ API

Two specific constructor are available:

- The first one to be used with the `updateSimulations` methods

```
1 SparseRegression(const int &p_levelMax, const Eigen::ArrayXd &p_weight, const int
   &p_degree, bool p_bNoRescale = false);
```

where

- `p_levelMax` corresponds to n in the equation (1.4),
 - `p_weight` the weight of the sparse anisotropic grids (see equation (1.7),
 - `p_degree` is equal to 1 (linear basis function), or 2 (quadratic basis) or 3 (for cubic basis functions),
 - `p_bNoRescale` if `true` no rescaling of the particles is used. Otherwise a new scaling of the mesh size is obtained (as for local basis functions, see section 2.2)
- The second take the same arguments as the first constructor but adds a Boolean to check if the regression date is 0 and the particles X_t (here the scaling is always performed):

```
1 SparseRegression(const bool &p_bZeroDate,
2                  const shared_ptr< Eigen::ArrayXXd > &p_particles,
3                  const int &p_levelMax, const Eigen::ArrayXd &p_weight,
4                  const int &p_degree);
```

A simple example to express the regression of `toRegress`

```
1 // second member to regress
2 ArrayXd toRegress(p_nbSimul);
3 // for testing
4 toRegress.setConstant(.);
5 shared_ptr<ArrayXXd> x(new ArrayXXd(ArrayXXd::Random(p_nDim, p_nbSimul)));
6 // constructor : the current date is not zero
7 bool bZeroDate = 0;
8 // constructor
9 SparseRegression sparseRegressor(p_level, weight, p_degree);
10 sparseRegressor.updateSimulations(bZeroDate, x); // update the state
11 // then just calculate function basis coefficient
12 ArrayXd regressedFuntionCoeff = sparseRegressor.getCoordBasisFunction(toRegress);
13 // use the getValue method to get back the regressed values
14 for (int is = 0; is < p_nbSimul; ++is)
15 {
16     Map<ArrayXd> xloc(x->col(is).data(), p_nDim);
17     double reg = sparseRegressor.getValue(xloc, regressedFuntionCoeff);
18 }
19 // get back all values once for all
20 ArrayXd regressedAllValues = localRegressor.getValues(*x, regressedFuntionCoeff);
```

2.5.2 Python API

Here is a simple example of the Python API:

```

1 import pystopt.regression as StOptReg
2 nbSimul = 2000000;
3 np.random.seed(000)
4 x = np.random.uniform(-.,1.,size=(1,nbSimul));
5 # real function
6 toReal = (2+x[0,:]+(x[0,:])*(1+x[0,:]))
7 # function to regress
8 toRegress = toReal + 4*np.random.normal(0.,,nbSimul)
9 # level for sparse grid
10 iLevel = 5;
11 # weight for anisotropic sparse grids
12 weight= np.array([],dtype=np.int32)
13 # Regressor degree
14 regressor = StOptReg.SparseRegression(False,x,iLevel, weight, )
15 y = regressor.getAllSimulations(toRegress)
16 # get back basis function
17 regressedFuntionCoeff= regressor.getCoordBasisFunction(toRegress)
18 # get back all values
19 ySecond= regressor.getValues(x,regressedFuntionCoeff)

```

2.6 Global polynomial basis

2.6.1 Description of the method

In this section, the $\psi_k(X_t)$ involved in equation 2.1 are given polynomials. The available polynomials are the canonical polynomials, those of Hermite and Chebyshev.

- Hermite polynomials $H_m(x) = (-1)^n e^{\frac{x^2}{2}} \frac{d^n}{dx^n} e^{-\frac{x^2}{2}}$ are orthogonal with respect to the weight $w(x) = e^{-\frac{x^2}{2}}$ and we get

$$\int_{-\infty}^{+\infty} w(x) H_m(x) H_n(x) dx = \delta_{mn} \sqrt{2\pi n!}$$

they satisfy the recurrence:

$$H_{n+1}(x) = xH_n(x) - H'_n(x)$$

assuming $H_n(x) = \sum_{k=0}^n a_{n,k} x^k$, we get the recurrence

$$a_{n+1,k} = a_{n,k-1} - na_{n-1,k}, k > 0 \quad (2.5)$$

$$a_{n+1,0} = -na_{n-1,0} \quad (2.6)$$

- Chebyshev polynomials are $T_{N+1}(x) = \cos((N+1) \arccos(x))$. They are orthogonal to the weight $w(x) = \frac{1}{\sqrt{1-x^2}}$ and

$$\int_{-1}^1 T_N(x) T_M(x) w(x) dx = \begin{cases} 0, & \text{if } M \neq N \\ \pi, & \text{if } M = N = 0 \\ \frac{\pi}{2}, & \text{if } M = N \neq 0 \end{cases}$$

They satisfy the following recurrence:

$$T_{N+2}(x) = 2xT_{N+1}(x) - T_N(x)$$

Optionally, data rotation is possible even if the advantage of rotation seems limited for global polynomials.

2.6.2 C++ API

The class `GlobalRegression` is templated by the type of the polynomial (`Canonical`, `Tchebychev` or `Hermite`) The first constructor:

```
1 GlobalRegression(const int & p_degree, const int & p_dim, bool p_bRotationAndRecalc =  
    false);
```

where `p_degree` is the total degree of the polynomial approximation, `p_dim` is the dimension of the problem, `p_bRotationAndRecalc` is an optional flag set to `true` if data rotation must be performed (by default, no rotation). A second constructor is provided:

```
1 GlobalRegression(const bool & p_bZeroDate,  
2     const std::shared_ptr< Eigen::ArrayXXd > & p_particles,  
3     const int & p_degree, bool p_bRotationAndRecalc = false)
```

where

- `p_bZeroDate` is `true` if the regression date is 0,
- `p_particles` the particles X_t for all simulations (dimension of X_t for first dimension, number of Monte Carlo simulations in second dimension),
- `p_degree` is the total degree of the polynomial approximation,
- `p_bRotationAndRecalc` is an optional flag set to `true` if data rotation is to be performed (default, no rotation)

Below, we give a small example where `toRegress` corresponds to $g(t + h, X_{t+h})$ for all simulations and `x` store X_t for all simulations.

```
1 // total degree equal to 2  
2 int degree=2;  
3 // t is not zero  
4 bool bZeroDate = 0;  
5 // constructor with Hermite polynomials, no rotation  
6 GlobalRegression<Hermite> localRegressor(degree,x.rows());  
7 // update particles values  
8 localRegressor.updateSimulations(bZeroDate, x);  
9 // regressed values  
10 ArrayXd regressedValues = localRegressor.getAllSimulations(toRegress);
```

In the example above, the Hermite regression can be replaced by the canonical regression:

```
1 GlobalRegression<Canonical> localRegressor(degree,x.rows());
```

or by a Chebyshev:

```
1 GlobalRegression<Tchebychev> localRegressor(degree,x.rows());
```

2.6.3 Python API

Here is a similar example using the second constructor

```
1 import pystopt.regression as StOptReg  
2 nbSimul = 5000000;  
3 np.random.seed(1000)  
4 x = np.random.uniform(-.,1.,size=(1,nbSimul));  
5 # real function
```

```

6      toReal = (2+x[0,:]+(1+x[0,:])*(1+x[0,:]))
7      # function to regress
8      toRegress = toReal + 4*np.random.normal(0.,,nbSimul)
9      # degree
10     degree =2
11     # Regressor, no rotation
12     regressor = StOptReg.GlobalHermiteRegression(False,x,degree)
13     y = regressor.getAllSimulations(toRegress).transpose()[0]

```

The available regressors are `GlobalHermiteRegression` as in the example above , `GlobalCanonicalRegression` and `GlobalTchebychevRegression` with obvious correspondence.

2.7 Kernel regression with Epanechnikov kernel

Let $(x_1, y_1), (x_2, y_2), \dots, (x_N, y_N)$ be a sample of N input points x_i and output points y_i drawn from a joint distribution (X, Y) . The kernel density estimator (aka Parzen–Rosenblatt estimator) of the density of X at the evaluation point z is given by:

$$\hat{f}_{\text{KDE}}(z) := \frac{1}{N} \sum_{i=1}^N K_h(x_i - z) \quad (2.7)$$

where $K_h(u) := \frac{1}{h} K\left(\frac{u}{h}\right)$ with kernel K and bandwidth h . The Nadaraya–Watson kernel regression estimator of $\mathbb{E}[Y | X = z]$ is given by:

$$\hat{f}_{\text{NW}}(z) := \frac{\sum_{i=1}^N K_h(x_i - z) y_i}{\sum_{i=1}^N K_h(x_i - z)} \quad (2.8)$$

The estimator $\hat{f}_{\text{NW}}(z)$ performs a kernel-weighted local average of the response points y_i that are such that their corresponding inputs x_i are close to the evaluation point z . It can be described as a locally constant regression. More generally, locally linear regressions can be performed:

$$\hat{f}_{\text{L}}(z) := \min_{\alpha(z), \beta(z)} \sum_{i=1}^N K_h(x_i - z) [y_i - \alpha(z) - \beta(z)x_i]^2 \quad (2.9)$$

The well known computational problem with the implementation of the kernel smoothers (2.7)-(2.8)-(2.9) is that their direct evaluation on a set of M evaluation points would require $\mathcal{O}(M \times N)$ operations. In particular, when the evaluation points coincide with the input points $x_1, x_2, \dots, x_N$, a direct evaluation requires a quadratic $\mathcal{O}(N^2)$ number of operations. StOpt implements the methodology described in [29] to bring the computational cost down to $\mathcal{O}(N \log N)$.

2.7.1 The univariate case

In one dimension, StOpt uses the one-dimensional Epanechnikov kernel

$$K(u) = \frac{3}{4}(1 - u^2)\mathbb{1}\{|u| \leq 1\}$$

and the fast summation algorithm is used: Let $(x_1, y_1), (x_2, y_2), \dots, (x_N, y_N)$ be a sample of N input points (source) x_i and output points y_i , and let $z_1, z_2, \dots, z_M$ be a set of M evaluation (target) points. Without loss of generality, we assume that the input points and evaluation points are sorted: $x_1 \leq x_2 \leq \dots \leq x_N$ and $z_1 \leq z_2 \leq \dots \leq z_M$. In order to compute the kernel density estimator (2.7), kernel regression (2.8) and locally linear regression (2.9) for every evaluation point z_j , we need to compute sums of the type

$$\mathbf{S}_j = \mathbf{S}_j^{p,q} := \frac{1}{N} \sum_{i=1}^N K_h(x_i - z_j) x_i^p y_i^q = \frac{1}{Nh} \sum_{i=1}^N K\left(\frac{x_i - z_j}{h}\right) x_i^p y_i^q, p = 0, 1, q = 0, 1 \quad (2.10)$$

for each $j \in \{1, 2, \dots, M\}$. Direct and independent evaluation of these sums would require $\mathcal{O}(N \times M)$ operations (a sum of N terms for each $j \in \{1, 2, \dots, M\}$). The idea of fast sum updating consists in using the information in the sum $\mathbf{S}_j$ to calculate the next sum $\mathbf{S}_{j+1}$ without going through all the N input points again. Using the Epanechnikov kernel (parabolic) $K(u) = \frac{3}{4}(1 - u^2)\mathbb{1}\{|u| \leq 1\}$ we get:

$$\begin{aligned} \mathbf{S}_j^{p,q} &= \frac{1}{Nh} \sum_{i=1}^N \frac{3}{4} \left(1 - \left(\frac{x_i - z_j}{h}\right)^2\right) x_i^p y_i^q \mathbb{1}\{z_j - h \leq x_i \leq z_j + h\} \\ &= \frac{1}{Nh} \frac{3}{4} \sum_{i=1}^N \left(1 - \frac{z_j^2}{h^2} + 2\frac{z_j}{h^2}x_i - \frac{1}{h^2}x_i^2\right) x_i^p y_i^q \mathbb{1}\{z_j - h \leq x_i \leq z_j + h\} \\ &= \frac{3}{4Nh} \left\{ \left(1 - \frac{z_j^2}{h^2}\right) \mathcal{S}^{p,q}([z_j - h, z_j + h]) + 2\frac{z_j}{h^2} \mathcal{S}^{p+1,q}([z_j - h, z_j + h]) - \frac{1}{h^2} \mathcal{S}^{p+2,q}([z_j - h, z_j + h]) \right\} \end{aligned} \quad (2.11)$$

where

$$\mathcal{S}^{p,q}([L, R]) := \sum_{i=1}^N x_i^p y_i^q \mathbb{1}\{L \leq x_i \leq R\} \quad (2.12)$$

These sums $\mathcal{S}^{p,q}([z_j - h, z_j + h])$ can be evaluated quickly from $j = 1$ to $j = M$ as long as the input points x_i and the evaluation points z_j are sorted in ascending order. Indeed,

$$\begin{aligned} \mathcal{S}^{p,q}([z_{j+1} - h, z_{j+1} + h]) &= \sum_{i=1}^N x_i^p y_i^q \mathbb{1}\{z_{j+1} - h \leq x_i \leq z_{j+1} + h\} \\ &= \sum_{i=1}^N x_i^p y_i^q \mathbb{1}\{z_j - h \leq x_i \leq z_j + h\} \\ &\quad - \sum_{i=1}^N x_i^p y_i^q \mathbb{1}\{z_j - h \leq x_i < z_{j+1} - h\} + \sum_{i=1}^N x_i^p y_i^q \mathbb{1}\{z_j + h < x_i \leq z_{j+1} + h\} \\ &= \mathcal{S}^{p,q}([z_j - h, z_j + h]) - \mathcal{S}^{p,q}([z_j - h, z_{j+1} - h]) + \mathcal{S}^{p,q}([z_j + h, z_{j+1} + h]) \end{aligned} \quad (2.13)$$

Therefore, we can simply update the sum $\mathcal{S}^{p,q}([z_j - h, z_{j+1} + h])$ for the evaluation point z_j to obtain the next sum $\mathcal{S}^{p,q}([z_{j+1} - h, z_{j+1} + h])$ for the next evaluation point z_{j+1} by subtracting the terms $x_i^p y_i^q$ for which x_i is between $z_j - h$ and $z_{j+1} - h$, and adding the

terms $x_i^p y_i^q$ for which x_i is between $z_j + h$ and $z_{j+1} + h$. This can be achieved in a fast $\mathcal{O}(M + N)$ operations by going through the input points x_i , stored in increasing order at a cost of $\mathcal{O}(N \log N)$ operations, and through the evaluation points z_j , stored in increasing order at a cost of $\mathcal{O}(M \log M)$ operations.

2.7.2 The multivariate case

We now turn to the multivariate case. Let d be the dimension of the inputs. We consider again a sample $(x_1, y_1), (x_2, y_2), \dots, (x_N, y_N)$ of N input points x_i and output points y_i , where the input points are now multivariate:

$$x_i = (x_{1,i}, x_{2,i}, \dots, x_{d,i}), \quad i \in \{1, 2, \dots, N\}$$

The StOpt library implements the additive Epanechnikov kernel in the multi-dimensional case ([29]).

$$K_d(u_1, \dots, u_d) = \frac{1}{d2^{d-1}} \sum_{k=1}^d K(u_k) \prod_{k_0=1}^d \mathbb{1}\{|u_{k_0}| < 1\} = \frac{3}{d2^{d+1}} \sum_{k=1}^d (1 - u_k^2) \prod_{k_0=1}^d \mathbb{1}\{|u_{k_0}| < 1\} \quad (2.14)$$

We can show ([29]) that calculating the multivariate version of the smoothing kernels (2.7), (2.8) and (2.9) boils down to the calculation of the following sums:

$$\begin{aligned} \mathbf{s}_j &= \mathbf{s}_{k_1, k_2, j}^{p_1, p_2, q} := \frac{1}{N} \sum_{i=1}^N K_{d,h}(x_i - z_j) x_{k_1, i}^{p_1} x_{k_2, i}^{p_2} y_i^q \\ &= \frac{1}{N \prod_{k=1}^d h_k} \sum_{i=1}^N K_d \left(\frac{x_{1,i} - z_{1,j}}{h_1}, \frac{x_{2,i} - z_{2,j}}{h_2}, \dots, \frac{x_{d,i} - z_{d,j}}{h_d} \right) x_{k_1, i}^{p_1} x_{k_2, i}^{p_2} y_i^q \end{aligned} \quad (2.15)$$

for each evaluation point $z_j = (z_{1,j}, z_{2,j}, \dots, z_{d,j}) \in \mathbb{R}^d$, $j \in \{1, 2, \dots, M\}$, for powers $p_1, p_2, q = 0, 1$ and for dimension indices $k_1, k_2 = 1, 2, \dots, d$.

Kernel expansion

Using the multivariate kernel (2.14), we can expand the sum (2.15) as follows:

$$\begin{aligned}
\mathbf{S}_j &= \frac{1}{N \prod_{k=1}^d h_k} \sum_{i=1}^N K_d \left(\frac{x_{1,i} - z_{1,j}}{h_1}, \frac{x_{2,i} - z_{2,j}}{h_2}, \dots, \frac{x_{d,i} - z_{d,j}}{h_d} \right) x_{k_1,i}^{p_1} x_{k_2,i}^{p_2} y_i^q \\
&= \frac{3}{d 2^{d+1} N \prod_{k=1}^d h_k} \sum_{i=1}^N \sum_{k=1}^d \left(1 - \frac{(x_{k,i} - z_{k,j})^2}{h_k^2} \right) x_{k_1,i}^{p_1} x_{k_2,i}^{p_2} y_i^q \prod_{k_0=1}^d \mathbb{1}\{|x_{k_0,i} - z_{k_0,j}|\leq 1\} \\
&= \frac{3}{d 2^{d+1} N \prod_{k=1}^d h_k} \sum_{k=1}^d \sum_{i=1}^N \left(1 - \frac{z_{k,j}^2}{h_k^2} + 2 \frac{z_{k,j}}{h_k^2} x_{k,i} - \frac{1}{h_k^2} x_{k,i}^2 \right) x_{k_1,i}^{p_1} x_{k_2,i}^{p_2} y_i^q \prod_{k_0=1}^d \mathbb{1}\{|x_{k_0,i} - z_{k_0,j}|\leq 1\} \\
&= \frac{3}{d 2^{d+1} N \prod_{k=1}^d h_k} \sum_{k=1}^d \left\{ \left(1 - \frac{z_{k,j}^2}{h_k^2} \right) \mathcal{S}_{[k,k_1,k_2]}^{[0,p_1,p_2],q}([z_j - h_j, z_j + h_j]) + \right. \\
&\quad \left. + 2 \frac{z_{k,j}}{h_k^2} \mathcal{S}_{[k,k_1,k_2]}^{[1,p_1,p_2],q}([z_j - h_j, z_j + h_j]) - \frac{1}{h_k^2} \mathcal{S}_{[k,k_1,k_2]}^{[2,p_1,p_2],q}([z_j - h_j, z_j + h_j]) \right\} \tag{2.16}
\end{aligned}$$

where

$$\mathcal{S}^{\text{idx}}([\mathbf{L}, \mathbf{R}]) := \mathcal{S}_{\mathbf{k}}^{\mathbf{p},q}([\mathbf{L}, \mathbf{R}]) := \sum_{i=1}^N \left(\prod_{l=1}^3 (x_{k_l,i})^{p_l} \right) y_i^q \prod_{k_0=1}^d \mathbb{1}\{L_{k_0} \leq x_{k_0,i} \leq R_{k_0}\} \tag{2.17}$$

for any hyperrectangle $[\mathbf{L}, \mathbf{R}] := [L_1, R_1] \times [L_2, R_2] \times \dots \times [L_d, R_d] \subseteq \mathbb{R}^d$, powers $\mathbf{p} := (p_1, p_2, p_3) \in \mathbb{N}^3$, $q \in \mathbb{N}$, indices $\mathbf{k} := (k_1, k_2, k_3) \in \{1, 2, \dots, d\}^3$, and where $[z_j - h_j, z_j + h_j] := [z_{1,j} - h_{1,j}, z_{1,j} + h_{1,j}] \times \dots \times [z_{d,j} - h_{d,j}, z_{d,j} + h_{d,j}]$. To simplify notations, we make use of the multi-index $\text{idx} := (\mathbf{p}, q, \mathbf{k})$.

To sum up what has been established so far, computing multivariate kernel smoothers (kernel density estimation, kernel regression, locally linear regression) boils down to computing sums of type (2.17) on hyperrectangles of the type $[z_j - h_j, z_j + h_j]$ for every evaluation point $j \in \{1, 2, \dots, M\}$. In the univariate case, these sums could be computed efficiently by sorting the input points x_i , $i \in \{1, 2, \dots, N\}$ and updating the sums from one evaluation point to the next (equation (2.13)). We now describe a similar efficient fast sum updating algorithm for multivariate sums (2.17), first established in [29]. To do so, we first partition the input data into a multivariate rectilinear grid (subsection 2.7.2), taking advantage of the fact that the evaluation grid is rectilinear and that the supports of the kernels have a hyperrectangle shape. Then, we set up a fast sweeping algorithm using the sums on each hyperrectangle of the partition as the unit blocks to be added and removed (subsection 2.7.2), unlike the univariate case where the input points themselves were being added and removed iteratively.

Data partition

The first stage of the multivariate fast sum updating algorithm is to partition the sample of input points into boxes. To do so, define the sorted lists $\tilde{\mathcal{G}}_k = \{\tilde{g}_{k,1}, \tilde{g}_{k,2}, \dots, \tilde{g}_{k,2M_k}\} := \text{sort}\left(\{z_{k,j_k} - h_{k,j_k}\}_{j_k \in \{1,2,\dots,M_k\}} \cup \{z_{k,j_k} + h_{k,j_k}\}_{j_k \in \{1,2,\dots,M_k\}}\right)$ in each dimension $k \in \{1, 2, \dots, d\}$, and define the partition intervals $\tilde{I}_{k,l} := [\tilde{g}_{k,l}, \tilde{g}_{k,l+1}]$ for $l \in \{1, 2, \dots, 2M_k - 1\}$. By definition of $\tilde{\mathcal{G}}_k$, all the bandwidths edges $z_{k,j_k} - h_{k,j_k}$ and $z_{k,j_k} + h_{k,j_k}$, $j_k \in \{1, 2, \dots, M_k\}$, belong

to $\tilde{\mathcal{G}}_k$. Therefore, there exists some indices $\tilde{L}_{k,j_k}$ and $\tilde{R}_{k,j_k}$ such that $[z_{k,j_k} - h_{k,j_k}, z_{k,j_k} + h_{k,j_k}] = [\tilde{g}_{k,\tilde{L}_{k,j_k}}, \tilde{g}_{k,\tilde{R}_{k,j_k}+1}] = \bigcup_{l_k \in \{\tilde{L}_{k,j_k}, \dots, \tilde{R}_{k,j_k}\}} \tilde{I}_{k,l_k}$. From there, for any evaluation point $z_j = (z_{1,j_1}, z_{2,j_2}, \dots, z_{d,j_d}) \in \mathbb{R}^d$, the box $[z_j - h_j, z_j + h_j] \subset \mathbb{R}^d$ can be decomposed into a union of smaller boxes:

$$\begin{aligned} [z_j - h_j, z_j + h_j] &= [z_{1,j_1} - h_{1,j_1}, z_{1,j_1} + h_{1,j_1}] \times \dots \times [z_{d,j_d} - h_{d,j_d}, z_{d,j_d} + h_{d,j_d}] \\ &= [\tilde{g}_{1,\tilde{L}_{1,j_1}}, \tilde{g}_{1,\tilde{R}_{1,j_1}+1}] \times \dots \times [\tilde{g}_{d,\tilde{L}_{d,j_d}}, \tilde{g}_{d,\tilde{R}_{d,j_d}+1}] \\ &= \bigcup_{(l_1, \dots, l_d) \in \{\tilde{L}_{1,j_1}, \dots, \tilde{R}_{1,j_1}\} \times \dots \times \{\tilde{L}_{d,j_d}, \dots, \tilde{R}_{d,j_d}\}} \tilde{I}_{1,l_1} \times \dots \times \tilde{I}_{d,l_d} \end{aligned} \quad (2.18)$$

In other words, the set of boxes $\tilde{I}_{1,l_1} \times \tilde{I}_{2,l_2} \times \dots \times \tilde{I}_{d,l_d}$ s.t. $l_k \in \{\tilde{L}_{k,j_k}, \tilde{L}_{k,j_k} + 1, \dots, \tilde{R}_{k,j_k}\}$ in each dimension $k \in \{1, 2, \dots, d\}$ forms a partition of the box $[z_j - h_j, z_j + h_j]$. Consequently, the sum (2.17) evaluated on the box $[z_j - h_j, z_j + h_j]$ can be decomposed as follows:

$$\mathcal{S}^{\text{idx}}([z_j - h_j, z_j + h_j]) = \sum_{(l_1, \dots, l_d) \in \{\tilde{L}_{1,j_1}, \dots, \tilde{R}_{1,j_1}\} \times \dots \times \{\tilde{L}_{d,j_d}, \dots, \tilde{R}_{d,j_d}\}} \mathcal{S}^{\text{idx}}(\tilde{I}_{1,l_1} \times \dots \times \tilde{I}_{d,l_d}) \quad (2.19)$$

where we assume without loss of generality that the bandwidth grid $h_j = (h_{1,j_1}, h_{2,j_2}, \dots, h_{d,j_d})$, $j_k \in \{1, 2, \dots, M_k\}$, $k \in \{1, 2, \dots, d\}$ is such that the list $\tilde{\mathcal{G}}_k$ does not contain any input $x_{k,i}$, $i \in \{1, 2, \dots, N\}$ (as such boundary points would be counted twice in the right-hand side of (2.19)). The sum decomposition (2.19) is the cornerstone of the fast multivariate sum updating algorithm, but before going further, one can simplify the partitions $\tilde{\mathcal{G}}_k$ while maintaining a sum decomposition of the type (2.19). Indeed, in general some intervals $\tilde{I}_{k,l}$ might be empty (i.e. they might not contain any input point $x_{k,i}$). To avoid keeping track of sums $\mathcal{S}^{\text{idx}}$ on boxes known to be empty, one can trim the partitions $\tilde{\mathcal{G}}_k$ by replacing each succession of empty intervals by one new partition threshold. For example, if $\tilde{I}_{k,l} = [\tilde{g}_{k,l}, \tilde{g}_{k,l+1}]$ is empty, one can remove the two points $\tilde{g}_{k,l}$ and $\tilde{g}_{k,l+1}$ and replace them by, for example, $(\tilde{g}_{k,l} + \tilde{g}_{k,l+1})/2$. Denote by $\mathcal{G}_k = \{g_{k,1}, g_{k,2}, \dots, g_{k,m_k}\}$ the sorted simplified list, where $2 \leq m_k \leq 2M_k$, $k \in \{1, 2, \dots, d\}$, and $m := \prod_{k=1}^d m_k \leq 2^d M$. Define the new partition intervals $I_{k,l} := [g_{k,l}, g_{k,l+1}]$, $l \in \{1, 2, \dots, m_k - 1\}$. Because the trimming from $\tilde{\mathcal{G}}_k$ to $\mathcal{G}_k$ only affects the empty intervals, the following still holds:

Lemma 2.7.1 *For any evaluation point $z_j = (z_{1,j_1}, z_{2,j_2}, \dots, z_{d,j_d}) \in \mathbb{R}^d$, $j_k \in \{1, 2, \dots, M_k\}$, $k \in \{1, 2, \dots, d\}$, there exists indices $(L_{1,j_1}, L_{2,j_2}, \dots, L_{d,j_d})$ and $(R_{1,j_1}, R_{2,j_2}, \dots, R_{d,j_d})$ where L_{k,j_k} and $R_{k,j_k} \in \{1, 2, \dots, m_k - 1\}$ with $L_{k,j_k} \leq R_{k,j_k}$ such that*

$$\mathcal{S}^{\text{idx}}([z_j - h_j, z_j + h_j]) = \sum_{(l_1, \dots, l_d) \in \{L_{1,j_1}, \dots, R_{1,j_1}\} \times \dots \times \{L_{d,j_d}, \dots, R_{d,j_d}\}} \mathcal{S}^{\text{idx}}(I_{1,l_1} \times \dots \times I_{d,l_d}) \quad (2.20)$$

For later use, we introduce the compact notation $\mathcal{S}_{l_1, l_2, \dots, l_d}^{\text{idx}} := \mathcal{S}^{\text{idx}}(I_{1,l_1} \times \dots \times I_{d,l_d})$. Recalling equation (2.17), the sum $\mathcal{S}_{l_1, l_2, \dots, l_d}^{\text{idx}}$ corresponds to the sum of the polynomials $(\prod_{i=1}^3 (x_{k_i, i})^{p_i}) y_i^q$ over all the data points within the box $I_{1,l_1} \times \dots \times I_{d,l_d}$.

Fast multivariate sweeping algorithm

So far, we have shown that computing multivariate kernel smoothers is based on the computation of the kernel sums (2.15), which can be decomposed into sums of the type (2.17),

which can themselves be decomposed into the smaller sums (2.20) by decomposing every kernel support of every evaluation point onto the box partition described in the previous subsection 2.7.2. The final task is to define an efficient algorithm to traverse all the hyperrectangle unions $\bigcup_{(l_1, \dots, l_d) \in \{L_{1,j_1}, \dots, R_{1,j_1}\} \times \dots \times \{L_{d,j_d}, \dots, R_{d,j_d}\}} I_{1,l_1} \times \dots \times I_{d,l_d}$, so as to compute the right-hand side sums in equations (2.20) (Lemma 2.7.1) in an efficient fast sum updating way similar to the univariate (2.13). We precompute all the sums $\mathcal{S}_{l_1, l_2, \dots, l_d}^{\text{idx}}$ with $\text{idx} = (\mathbf{p}, q, \mathbf{k}) \in \{0, 1, 2\} \times \{0, 1\}^3 \times \{1, 2, \dots, d\}^3$, and use them as input material for fast multivariate sum updating.

We start with the bivariate case. We first provide an algorithm to compute the sums $\mathcal{T}_{1, l_2}^{\text{idx}} := \sum_{l_1=L_{1,j_1}}^{R_{1,j_1}} \mathcal{S}_{l_1, l_2}^{\text{idx}}$, for every $l_2 \in \{1, 2, \dots, m_2 - 1\}$ and every index interval $[L_{1,j_1}, R_{1,j_1}]$, $j_1 \in \{1, 2, \dots, M_1\}$. Starting with $j_1 = 1$, we first compute $\mathcal{T}_{1, l_2}^{\text{idx}} = \sum_{l_1=L_{1,1}}^{R_{1,1}} \mathcal{S}_{l_1, l_2}^{\text{idx}}$ for every $l_2 \in \{1, 2, \dots, m_2 - 1\}$. Then we iteratively increment j_1 from $j_1 = 1$ to $j_1 = M_1$. After each incrementation of j_1 , we update $\mathcal{T}_{1, l_2}^{\text{idx}}$ by fast sum updating

$$\sum_{l_1=L_{1,j_1}}^{R_{1,j_1}} \mathcal{S}_{l_1, l_2}^{\text{idx}} = \sum_{l_1=L_{1,j_1-1}}^{R_{1,j_1-1}} \mathcal{S}_{l_1, l_2}^{\text{idx}} + \sum_{l_1=R_{1,j_1-1}+1}^{R_{1,j_1}} \mathcal{S}_{l_1, l_2}^{\text{idx}} - \sum_{l_1=L_{1,j_1-1}}^{L_{1,j_1}-1} \mathcal{S}_{l_1, l_2}^{\text{idx}} \quad (2.21)$$

The second stage is to perform a fast sum updating in the second dimension, with the sums $\mathcal{T}_{1, l_2}^{\text{idx}} = \sum_{l_1=L_{1,j_1}}^{R_{1,j_1}} \mathcal{S}_{l_1, l_2}^{\text{idx}}$ as input material. Our goal is to compute the sums $\mathcal{T}_2^{\text{idx}} := \sum_{l_2=L_{2,j_2}}^{R_{2,j_2}} \mathcal{T}_{1, l_2}^{\text{idx}}$ for every index interval $[L_{2,j_2}, R_{2,j_2}]$, $j_2 \in \{1, 2, \dots, M_2\}$. In a similar manner, we start from $j_2 = 1$ with the initial sum $\mathcal{T}_2^{\text{idx}} = \sum_{l_2=L_{2,1}}^{R_{2,1}} \mathcal{T}_{1, l_2}^{\text{idx}}$. We then increment j_2 from $j_2 = 1$ to $j_2 = M_2$ iteratively. After each incrementation of j_2 , we update $\mathcal{T}_2^{\text{idx}}$ by fast sum updating:

$$\sum_{l_2=L_{2,j_2}}^{R_{2,j_2}} \mathcal{T}_{1, l_2}^{\text{idx}} = \sum_{l_2=L_{2,j_2-1}}^{R_{2,j_2-1}} \mathcal{T}_{1, l_2}^{\text{idx}} + \sum_{l_2=R_{2,j_2-1}+1}^{R_{2,j_2}} \mathcal{T}_{1, l_2}^{\text{idx}} - \sum_{l_2=L_{2,j_2-1}}^{L_{2,j_2}-1} \mathcal{T}_{1, l_2}^{\text{idx}} \quad (2.22)$$

Using Lemma 2.7.1, the resulting sum $\sum_{l_2=L_{2,j_2}}^{R_{2,j_2}} \mathcal{T}_{1, l_2}^{\text{idx}} = \sum_{l_1=L_{1,j_1}}^{R_{1,j_1}} \sum_{l_2=L_{2,j_2}}^{R_{2,j_2}} \mathcal{S}_{l_1, l_2}^{\text{idx}}$ is equal to $\mathcal{S}^{\text{idx}}([z_j - h_j, z_j + h_j])$, which can be used to compute the kernel sums $\mathbf{S}_j$ using equation (2.16), from which the bivariate kernel smoothers can be computed.

This ends the description of the fast sum updating algorithm in the bivariate case. Finally, the general multivariate case is a straightforward extension of the bivariate case (see [29]).

2.7.3 C++ API

The constructor allows to define the kernel regressor:

```
1 LocalGridKernelRegression(const bool &p_bZeroDate,
2                             const std::shared_ptr< Eigen::ArrayXXd > &p_particles,
3                             const double &p_coefBandWidth,
4                             const double &p_coefNbGridPoint,
5                             const bool &p_bLinear);
```

where

- `p_bZeroDate` is `true` if the regression date is 0,

- `p_particles` the particles X_t for all simulations (dimension of X_t for first dimension, number of Monte Carlo simulations in second dimension),
- `p_coeffBandWidth` between 0 and 1 defines the percentage of points to use to define the bandwidth for each point.
- `p_coefNbGridPoint` is a multiplying factor defining the number of points z used for the multi-grid approximation: a PCA is used to define a data rotation. The kernel regression is performed according the basis defined by the eigenvectors associated with the PCA. The number of points along the axes defined by the eigenvectors is given as a function of the singular value associated with the eigenvector. The total number of evaluation points along the axes of the new base is approximately the number of simulations (`p_particles.cols()`) by `p_coefNbGridPoint`.
- `p_bLinear` when set to `false` indicates that the simple estimate of the density of the kernel (2.8) is used. When `p_bLinear` is `true`, the linear regression of the kernel (2.9) is used.

Below we give a small example where `toRegress` corresponds to $g(t + h, X_{t+h})$ for all simulations and x stores X_t for all simulations.

```

1
2 // t is not zero
3 bool bZeroDate = 0;
4 // proportion of points used to define the bandwidth
5 double prop = 0.1;
6 // multiplicative factor equal to one: number of evaluation points equal to the
  number of particles
7 double q = 1.
8 // choose a linear regression
9 bool bLin= true;
10 // constructor
11 LocalGridKernelRegression kernelReg(bZeroDate, x, prop, q, bLin);
12 // update particles values
13 localRegressor.updateSimulations(bZeroDate, x);
14 // regressed values
15 ArrayXd regressedValues = localRegressor.getAllSimulations(toRegress);

```

2.7.4 Python API

As usual the Python constructors are similar to the C++ constructors. Here is a small example of the use of the kernel regression method.

```

1 import pystopt.regression as StOptReg
2 nbSimul = 5000000;
3 np.random.seed(1000)
4 x = np.random.uniform(-.,1.,size=(1,nbSimul));
5 # real function
6 toReal = (2+x[0,:]+(x[0,:])*(1+x[0,:]))
7 # function to regress
8 toRegress = toReal + 4*np.random.normal(0.,nbSimul)
9 # bandwidth
10 bandwidth = 0.1
11 # factor for the number of points
12 factPoint=1
13 # Regressor
14 regressor = StOptReg.LocalGridKernelRegression(False,x,bandwidth,factPoint, True)
15 # get regressed
16 y = regressor.getAllSimulations(toRegress).transpose()[0]

```

2.8 Kernel regression with Laplacian kernels

This subsection is based on the work [30]. Keeping in mind the notations of section 2.7, we focus on the Laplacian kernel, defined by

$$K(u) = \frac{1}{2}e^{-|u|} \quad (2.23)$$

The Laplacian kernel density estimator is defined by:

$$\hat{f}_{\text{KDE}}(z) = \frac{1}{N} \sum_{i=1}^N \frac{1}{2h} e^{-\frac{|x_i - z|}{h}} \quad (2.24)$$

2.8.1 Reducing the problem to empirical CDF calculations

This kernel density estimator can be decomposed as follows in one dimension:

$$\begin{aligned} \hat{f}_{\text{KDE}}(z) &= \frac{1}{N} \sum_{i=1}^N \frac{1}{2h} e^{-\frac{|x_i - z|}{h}} \\ &= \frac{1}{2hN} \left(\sum_{i=1}^N e^{\frac{x_i - z}{h}} \mathbb{1}\{x_i \leq z\} + \sum_{i=1}^N e^{\frac{z - x_i}{h}} \mathbb{1}\{x_i > z\} \right) \\ &= \frac{1}{2hN} \left(e^{-\frac{z}{h}} \sum_{i=1}^N e^{\frac{x_i}{h}} \mathbb{1}\{x_i \leq z\} + e^{\frac{z}{h}} \sum_{i=1}^N e^{-\frac{x_i}{h}} \mathbb{1}\{x_i > z\} \right) \\ &= \frac{1}{2h} \left(e^{-\frac{z}{h}} F_N(z; x, we^{\frac{x}{h}}) + e^{\frac{z}{h}} \bar{F}_N(z; x, we^{-\frac{x}{h}}) \right) \end{aligned} \quad (2.25)$$

where the empirical CDF F_N and the empirical complementary CDF $\bar{F}_N$ are defined by equations

$$F_N(z) = F_N(z; x, y) \triangleq \frac{1}{N} \sum_{i=1}^N y_i \mathbb{1}\{x_{1,i} \leq z_1, \dots, x_{d,i} \leq z_d\}. \quad (2.26)$$

$$\bar{F}_N(z) = \bar{F}_N(z; x, y) \triangleq \frac{1}{N} \sum_{i=1}^N y_i \mathbb{1}\{x_{1,i} > z_1, \dots, x_{d,i} > z_d\}. \quad (2.27)$$

Crucially, such a CDF decomposition of KDE also holds in the multivariate setting. The multivariate Laplacian kernel is defined by

$$K_d(u) = \frac{1}{2^d} e^{-|u|} = \frac{1}{2^d} e^{-\sum_{k=1}^d |u_k|} \quad (2.28)$$

and the multivariate Laplacian kernel density estimator is given by

$$\hat{f}_{\text{KDE}}(z) = \frac{1}{2^d N \prod_{k=1}^d h_k} \sum_{i=1}^N e^{-\left| \frac{x_i - z}{h} \right|} = \frac{1}{2^d N \prod_{k=1}^d h_k} \sum_{i=1}^N e^{-\sum_{k=1}^d \frac{|x_{k,i} - z_k|}{h_k}} \quad (2.29)$$

where $h = (h_1, h_2, \dots, h_d) \in \mathbb{R}^d$ is a multivariate bandwidth.

We introduce :

$$F_N(z, \delta) = F_N(z, \delta; x, y) \triangleq \frac{1}{N} \sum_{i=1}^N y_i \mathbb{1}\{x_{1,i} \leq_{\delta_1} z_1, \dots, x_{d,i} \leq_{\delta_d} z_d\} \quad (2.30)$$

where $\delta = \{\delta_1, \delta_2, \dots, \delta_d\} \in \{-1, 1\}^d$, and where the generalized inequality operator $\leq_c$ corresponds to $\leq$ (lower or equal) if $c \geq 0$, and to $<$ (strictly lower) if $c < 0$. In particular $F_N(z) = F_N(z, 1; x, y)$ and $\bar{F}_N(z) = F_N(-z, -1; -x, y)$ respectively.

Using the same approach as equation (2.25), the sum (2.29) can be decomposed as follows:

$$\begin{aligned} \hat{f}_{\text{KDE}}(z) &= \frac{1}{2^d N \prod_{k=1}^d h_k} \sum_{i=1}^N \prod_{k=1}^d \left(e^{-\frac{z_k}{h_k}} e^{\frac{x_{k,i}}{h_k}} \mathbb{1}\{x_{k,i} \leq z_k\} + e^{\frac{z_k}{h_k}} e^{-\frac{x_{k,i}}{h_k}} \mathbb{1}\{-x_{k,i} < -z_k\} \right) \\ &= \frac{1}{2^d N \prod_{k=1}^d h_k} \sum_{i=1}^N \sum_{\delta \in \{-1, 1\}^d} \prod_{k=1}^d e^{-\frac{\delta_k z_k}{h_k}} e^{\frac{\delta_k x_{k,i}}{h_k}} \mathbb{1}\{\delta_k x_{k,i} \leq_{\delta_k} \delta_k z_k\} \\ &= \frac{1}{2^d \prod_{k=1}^d h_k} \sum_{\delta \in \{-1, 1\}^d} e^{-\sum_{k=1}^d \frac{\delta_k z_k}{h_k}} \frac{1}{N} \sum_{i=1}^N e^{\sum_{k=1}^d \frac{\delta_k x_{k,i}}{h_k}} \mathbb{1}\{\delta_1 x_{1,i} \leq_{\delta_1} \delta_1 z_1, \dots, \delta_d x_{d,i} \leq_{\delta_d} \delta_d z_d\} \\ &= \frac{1}{2^d \prod_{k=1}^d h_k} \sum_{\delta \in \{-1, 1\}^d} e^{-\sum_{k=1}^d \frac{\delta_k z_k}{h_k}} F_N(\delta z, \delta; \delta x, y) \end{aligned} \quad (2.31)$$

with $y_i = y_i(\delta) := e^{\sum_{k=1}^d \frac{\delta_k x_{k,i}}{h_k}}$. Equation (2.31) shows that the computation of the multivariate Laplacian kernel density estimator can be decomposed into the computation of 2^d generalized empirical CDF (2.30).

2.8.2 Fast computation of ECDFs

We present two ways to compute efficiently an expression such as 2.31. The first is based on the use of a rectilinear grid as evaluation points and an interpolation is needed to get back the conditional expectation at the sample points. The second one, based on the divide and conquer approach, is exact but much more costly.

Fast sum updating in lexicographical order

Let $z_j = (z_{1,j}, z_{2,j}, \dots, z_{d,j}) \in \mathbb{R}^d$, $j \in \{1, 2, \dots, M\}$, be a set of M evaluation (target) points.

We require this evaluation grid to be rectilinear, i.e., the M evaluation points $z_1, z_2, \dots, z_M$ lie on a regular grid with possibly non-uniform mesh, of dimension $M_1 \times M_2 \times \dots \times M_d = M$:

$$\mathbf{z} = \{(z_{1,j_1}, z_{2,j_2}, \dots, z_{d,j_d}) \in \mathbb{R}^d, j_k \in \{1, 2, \dots, M_k\}, k \in \{1, 2, \dots, d\}\}$$

For convenience, we extend the definition of the grid with the notational conventions $z_{k,0} \triangleq -\infty$ and $z_{k,M_k+1} \triangleq \infty$.

In each dimension $k \in \{1, 2, \dots, d\}$, the vector $(z_{k,1}, z_{k,2}, \dots, z_{k,M_k}) \in \mathbb{R}^{M_k}$ is assumed to be sorted in increasing order:

$$z_{k,1} < z_{k,2} < \dots < z_{k,M_k}, \quad k \in \{1, 2, \dots, d\}$$

We partition the input data $\mathbf{x}$ along this evaluation grid $\mathbf{z}$. For each evaluation grid index $(j_1, j_2, \dots, j_d) \in \{1, 2, \dots, M_1 + 1\} \times \dots \times \{1, 2, \dots, M_d + 1\}$ we define the following local sum

$$s_{j_1, j_2, \dots, j_d} := \frac{1}{N} \sum_{i=1}^N y_i \mathbb{1}\{z_{1, j_1-1} < x_{1,i} \leq z_{1, j_1}, \dots, z_{d, j_d-1} < x_{d,i} \leq z_{d, j_d}\} \quad (2.32)$$

Together, the sums (2.32) form a generalized multivariate histogram (classical histogram in the case $y \equiv 1$). For completeness, the computation of the local sums (2.32) is detailed in algorithm 6.

Algorithm 6 Fast computation of local sums by independent sorting in each dimension

```

1: Input: sample  $x_i = (x_{1,i}, \dots, x_{d,i})$ ,  $i = 1, 2, \dots, N$ 
2: Input : evaluation grid  $(z_{1,j_1}, z_{2,j_2}, \dots, z_{d,j_d})$ ,  $j_k \in \{1, 2, \dots, M_k\}$ ,  $k \in \{1, 2, \dots, d\}$ 
3: Define index matrix INDEX[ $k, i$ ]  $\triangleright$  local sum index  $\in \{1, 2, M_k + 1\}$  where
    $k = 1, 2, \dots, d$  and  $i = 1, 2, \dots, N$ 
4: for do ( $k = 1, 2, \dots, d$ )
5:   Sort the set  $\{x_{k,1}, \dots, x_{k,N}\}$  in increasing order, using for example quicksort or merge-
   sort ( $\mathcal{O}(N \log N)$ ): define the permutation  $\phi_k : \{1, 2, \dots, N\} \mapsto \{1, 2, \dots, N\}$ 
   such that

$$x_{k, \phi_k(1)} < x_{k, \phi_k(2)} < \dots < x_{k, \phi_k(N)} \quad (2.33)$$

6:    $x_{\text{idx}} = 1$   $\triangleright$  input index  $\in \{1, 2, \dots, N\}$ 
7:    $z_{\text{idx}} = 1$   $\triangleright$  evaluation grid index  $\in \{1, 2, \dots, M_k\}$ 
8:   while do ( $x_{\text{idx}} \leq N$ )
9:     if then ( $x_{k, \phi_k(x_{\text{idx}})} \leq z_{k, z_{\text{idx}}}$ )
10:      INDEX[ $k, \phi_k(x_{\text{idx}})$ ] =  $z_{\text{idx}}$ 
11:       $x_{\text{idx}} += 1$ 
12:     else
13:        $z_{\text{idx}} += 1$ 
14:     end if
15:   end while
16: end for
17:  $s_{j_1, j_2, \dots, j_d} = 0$ ,  $\forall (j_1, j_2, \dots, j_d) \in \{1, 2, \dots, M_1 + 1\} \times \dots \times \{1, 2, \dots, M_d + 1\}$ 
18: for do ( $i = 1, 2, \dots, N$ )
19:    $s_{\text{INDEX}[1,i], \text{INDEX}[2,i], \dots, \text{INDEX}[d,i]} += y_i / N$ 
20: end for
21: return  $s_{j_1, j_2, \dots, j_d} = \frac{1}{N} \sum_{i=1}^N y_i \mathbb{1}\{z_{1, j_1-1} < x_{1,i} \leq z_{1, j_1}, \dots, z_{d, j_d-1} < x_{d,i} \leq z_{d, j_d}\}$ 
   for every local sum index  $(j_1, j_2, \dots, j_d) \in \{1, 2, \dots, M_1 + 1\} \times \dots \times \{1, 2, \dots, M_d + 1\}$ 

```

In particular, using equation (2.26), the following key equality holds:

$$F_N(z) = \sum_{l_1=1}^{j_1} \sum_{l_2=1}^{j_2} \dots \sum_{l_d=1}^{j_d} s_{l_1, l_2, \dots, l_d} \quad (2.34)$$

for any evaluation point $z = (z_{1, j_1}, z_{2, j_2}, \dots, z_{d, j_d}) \in \mathbf{z}$.

We propose a simple fast summation algorithm, Algorithm 7, to compute the ECDFs $F_N(z)$ for every $z \in \mathbf{z}$ in lexicographical order based on the local sum decomposition (2.34). One can easily verify that the number of operations is proportional to $M_1 \times M_2 \times \dots \times M_d = M$. As the computation of the local sums (2.32) costs $\mathcal{O}(N \log N)$ operations (or only $\mathcal{O}(N)$ if the grid is uniform or the data already sorted), the overall computational complexity of Algorithm 7 is $\mathcal{O}(M + N \log N)$, or $\mathcal{O}(N \log N)$ when $M \approx N$ (respectively $\mathcal{O}(M + N)$ and $\mathcal{O}(N)$ when the grid is uniform or the data already sorted).

Algorithm 7 Fast joint empirical cumulative distribution function

```

1: Input :precomputed sums  $s_{l_1, l_2, \dots, l_d}$ 
2:  $\mathcal{S}_{1, l_2, l_3, \dots, l_d} = 0$ 
3: for do ( $j_1 = 1, \dots, M_1 + 1$ )
4:    $\mathcal{S}_{1, l_2, l_3, \dots, l_d} += s_{j_1, l_2, l_3, \dots, l_d}, \forall l_k \in \{1, 2, \dots, M_k + 1\}, k \in \{2, 3, \dots, d\}$  ▷ Here
    $\mathcal{S}_{1, l_2, l_3, \dots, l_d} = \sum_{l_1=1}^{j_1} s_{l_1, l_2, \dots, l_d}, \forall l_k \in \{1, 2, \dots, M_k + 1\}, k \in \{2, 3, \dots, d\}$ 
5:    $\mathcal{S}_{2, l_3, \dots, l_d} = 0$ 
6:   for do ( $j_2 = 1, \dots, M_2 + 1$ ) ▷ Here
7:      $\mathcal{S}_{2, l_3, \dots, l_d} += \mathcal{S}_{1, j_2, l_3, \dots, l_d}, \forall l_k \in \{1, 2, \dots, M_k + 1\}, k \in \{3, \dots, d\}$ 
      $\mathcal{S}_{2, l_3, \dots, l_d} = \sum_{l_1=1}^{j_1} \sum_{l_2=1}^{j_2} s_{l_1, l_2, \dots, l_d}, \forall l_k \in \{1, \dots, M_k + 1\}, k \in \{3, \dots, d\}$ 
8:      $\mathcal{S}_d = 0$ 
9:     for do ( $j_d = 1, \dots, M_d + 1$ )
10:       $\mathcal{S}_d += \mathcal{S}_{d-1, j_d}$ 
11:    ▷ Here  $\mathcal{S}_d = \sum_{l_1=1}^{j_1} \sum_{l_2=1}^{j_2} \dots \sum_{l_d=1}^{j_d} s_{l_1, l_2, \dots, l_d} \triangleright = F_N(z_{1, j_1}, z_{2, j_2}, \dots, z_{d, j_d}) = F_N(z)$  from
      equation (2.34)
12:    end for
13:  end for
14: end for
15: return  $F_N(z)$  for all  $z \in \mathbf{z}$ 

```

2.8.3 Divide-and-conquer approach

Consider the case when the evaluation points z_j are equal to the input points x_i . The calculation of the ECDFs $\{F_N(x_i)\}_{i=1, N}$ (equation (2.26)) corresponds to a domination problem in dimension d . An algorithm based on a recursive divide-and-conquer sequence has first been proposed in [6] for this problem. An adaptation was proposed in [9] to solve this problem for the case of the calculation of conditional expectation using Malliavin weights. The computational complexity was shown to be $\mathcal{O}(c(d)N \log(N)^{(d-1) \vee 1})$. We do not repeat the construction of this algorithm here, and the interested reader can refer to [30].

2.8.4 C++ API for fast summation method

The constructor allows to define the Laplacian kernel regressor with the fast summation method:

```

1 LaplacianGridKernelRegression(const bool &p_bZeroDate,
2                               const Eigen::ArrayXXd &p_particles,

```

```

3      const Eigen::ArrayXd    &p_h,
4      const double          &p_coefNbGridPoint,
5      const bool             &p_bLinear);

```

where

- `p_bZeroDate` is `true` if the regression date is 0,
- `p_particles` the particles X_t for all simulations (dimension of X_t for first dimension, number of Monte Carlo simulations in second dimension),
- `p_h` bandwidth array per dimension.
- `p_coefNbGridPoint` is a multiplying factor defining the number of points z used for the multi-grid approximation. The total number of points on the rectilinear grid is the number of samples by `p_coefNbGridPoint`.
- `p_bLinear` when set to `false` indicates that the simple estimate of the density of the kernel (2.8) is used. When `p_bLinear` is `true`, the linear regression of the kernel (2.9) is used.

Below we give a small example where `toRegress` corresponds to $g(t + h, X_{t+h})$ for all simulations and `x` store X_t for all simulations.

```

1      // t is not zero
2      bool bZeroDate = false;
3
4      // use linear regression
5      bool bLinear = true;
6
7      // bandwidth
8      ArrayXd hB = ArrayXd::Constant(1, p_h);
9
10     // test regression object
11     double q = 50; // coeff for the number of grid points used
12     LaplacianGridKernelRegression kernelReg(bZeroDate, x, hB, q, bLinear);
13
14     // regress on grid
15     ArrayXd regressed = kernelReg.getAllSimulations(y);

```

2.8.5 Python API for fast summation method

For example, using the linear regressor we get the following Python code:

```

1      import pystopt.regression as StOptReg
2      nbSimul = 100000;
3      np.random.seed(1000)
4      x = np.random.uniform(-1.,1.,size=(1,nbSimul));
5      # real function
6      toReal = (4+x[0,:]+(3+x[0,:])*(2+x[0,:]))
7      # function to regress
8      toRegress = toReal + 4*np.random.normal(0.,1,nbSimul)
9      # bandwidth
10     bandwidth = 0.05*np.ones(1)
11     # multiplicative for rectilinear grid
12     coeffMul = 2
13     # Linear or constant regressions
14     bLinear= False

```

```

15     # Regressor
16     regressor = StOptReg.LaplacianGridKernelRegression(False,x,bandwidth, coeffMul,
17                                                         bLinear)
18     # test particles
19     y = regressor.getAllSimulations(toRegress).transpose()

```

2.8.6 C++ API for exact divid and conquer method

The constructor allows to defines the Laplacian kernel regressor with the divide and conquer method and a constant regression

```

1 LaplacianConstKernelRegression(const bool &p_bZeroDate,
2                                const std::shared_ptr< Eigen::ArrayXXd > &p_particles,
3                                const double &p_h);

```

where

- `p_bZeroDate` is true if the regression date is 0,
- `p_particles` the particles X_t for all simulations (dimension of X_t for first dimension, number of Monte Carlo simulations in second dimension),
- `p_h` bandwidth array per dimension.

And the following one deals with the linear regression case.

```

1 LaplacianLinearKernelRegression(const bool &p_bZeroDate,
2                                 const std::shared_ptr< Eigen::ArrayXXd > &p_particles,
3                                 const double &p_h);

```

where

- `p_bZeroDate` is true if the regression date is 0,
- `p_particles` the particles X_t for all simulations (dimension of X_t for first dimension, number of Monte Carlo simulations in second dimension),
- `p_h` bandwidth array per dimension.

2.8.7 Python API for exact divide-and-conquer method

As usual, the Python constructor is similar to the C++ constructor. For example, using the linear regressor we get the following Python code:

```

1 import pystopt.regression as StOptReg
2 nbSimul = 100000;
3 np.random.seed(1000)
4 x = np.random.uniform(-1.,1.,size=(1,nbSimul));
5 # real function
6 toReal = (4+x[0,:]+(3+x[0,:])*(2+x[0,:]))
7 # function to regress
8 toRegress = toReal + 4*np.random.normal(0.,1,nbSimul)
9 # bandwidth
10 bandwidth = 0.05*np.ones(1)
11 # Regressor
12 regressor = StOptReg.LaplacianConstKernelRegression(False,x,bandwidth)
13 # test particles
14 y = regressor.getAllSimulations(toRegress).transpose()

```

Chapter 3

Calculating conditional expectation by trees

A popular method for calculating conditional expectation is to use scenario trees. In the financial community, binary and trinomial trees are generally used to evaluate options. When the asset is modeled by a Black-Scholes model, a binary model is used, while a trinomial model is used to model the average reversion using a Vasicek model for interest rate, for example [24]. An example of a trinomial tree is given in the figure 3.1 for an Ornstein–Uhlenbeck model (so in dimension 1). This tree models the possible evolution of

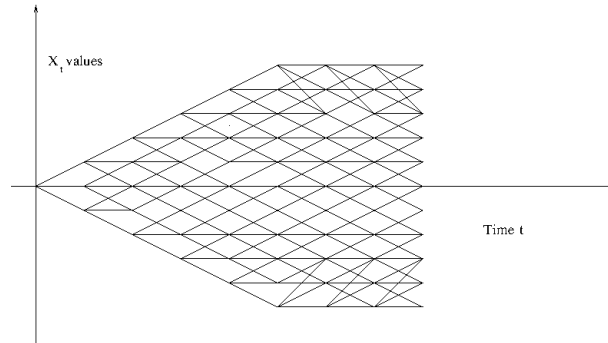


Figure 3.1: Trinomial tree

a state X_t in dimension 1 and each node corresponds to a possible value of X_t . These trees recombine. The nodes at each dates i are numbered from 0 to $N_i - 1$ with increasing values X_t^i of the state.

From a node i on a date t , 3 nodes can be reached on the date $t + 1$. The probability transition from going to a node down $f(i, t) - 1$ is $p_d^{t,i}$ while the probability of going to a middle node $f(i, t)$ is $p_m^{t,i}$ and the probability of going to a node up to $f(i, t) + 1$ is $p_u^{t,i}$.

The conditional expectation of a function with the values $g_j^{t+1} = g(X_{t+1}^j)$ at the node j on the date $i + 1$ is simply given by:

$$\mathbb{E}[g(X_{t+1})/X_t = X_t^i] \simeq p_d^{t,i} g_{f(t,i)-1}^{t+1} + p_m^{t,i} g_{f(t,i)}^{t+1} + p_u^{t,i} g_{f(t,i)+1}^{t+1} \quad (3.1)$$

In the literature, non-recombining scenario trees are used by the discrete stochastic optimization community. These non-recombining trees can be obtained by reducing certain

recombining trees (see [22] for example, or [28] for a more recent survey developing algorithm minimizing the Kantorovich or Wasserstein metric between the initial tree and a subtree of the initial tree). An example of non-recombining tree is given in the figure 3.2.

On figure 3.2, assuming that X_2 has the possible values Y_2, Y_3 at node 2 and 3, assuming

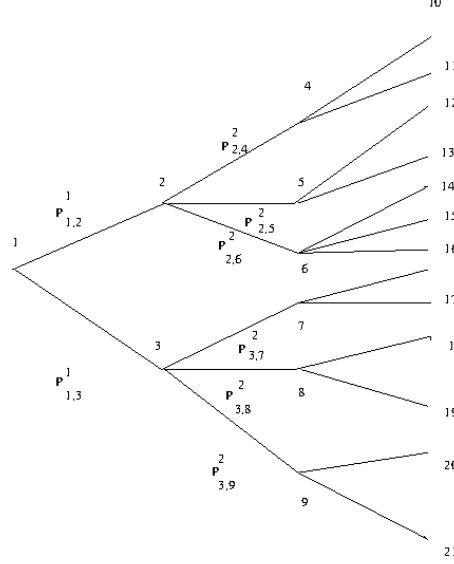


Figure 3.2: Non-recombining tree

that X_3 have discrete values at nodes $4, \dots, 9$ on the date $t = 3$ and that values of $g(X_3)$ have the value g_i at the node i on the date 3, then

$$\mathbb{E}[g(X_3)/X_2 = Y_3] = P_{3,7}g_7 + P_{3,8}g_8 + P_{3,9}g_9 \quad (3.2)$$

3.0.1 C++ API

Calculation of conditional expectation

As explained, conditional expectations are easy to calculate with trees. The library provides an object `Tree` allowing to make such calculations.

```
1 Tree(const std::vector<double> &p_proba, const std::vector< std::vector< std::array<int, 2> > > &p_connected)
```

with

- `p_proba` a probability vector at a given date defining the probability transition between the nodes at the current date and the nodes at the next date.
- `p_connected` the connection between the nodes and the index in the probability vector.

`p_proba[p_connected[i][j].second]`

is the probability of going from the node i at the current date to the node `p_connected[i][j].first` at the next date. So `p_connected[i].size()` gives the number of nodes connected to the node i .

For regression objects, some methods are provided to calculate conditional expectations:

- `expCond` takes an Eigen array with the size equal to the number of nodes at the next date and calculates the conditional expectation of the node values of current date,
- `expCondMultiple` does the same so that several functions regress (size of the number of functions by number of nodes at the following date) and return a two-dimensional Eigen array (size of functions by number of nodes at the current date).

Python API

The Python interface for tree is obtained importing the `StOptTree` module. An example taking a trinomial simulator is given below

```
1      # Mean Reverting model
2      mr = 0.3;
3      sig = 0.6;
4
5      # nb grid points
6      nbStock=4
7
8      # step
9      dt = 1. / 100.
10
11     # simulation dates
12     dates = dt * np.arange(0,16)
13
14     # simulaton dates
15     tree = Simulators.TrinomialTreeOUSimulator(mr, sig, dates)
16
17     iFirst = 10
18     iLast = 14
19
20     # nodes at dates 5
21     points = tree.getPoints(iFirst)
22
23     # nodes at last date
24     pointsNext = tree.getPoints(iLast)
25
26     # probabilities
27     proba = tree.getProbability(iFirst, iLast)
28
29     # connection matrix
30     connectAndProba = tree.calConnected(proba);
31
32     # to regress
33     toTreeress= np.zeros( (nbStock, np.shape(pointsNext)[1]))
34     for i in range(nbStock):
35         toTreeress[i,:] = i + pointsNext[0,:]
36
37     # grid for storage
38     grid = StOptGrids.RegularSpaceGrid(np.array([0.]), np.array([1.]), np.array([
39         nbStock - 1]))
40
41     # conditional expectation object by trees
42     tree= StOptTree.Tree(connectAndProba[1],connectAndProba[0])
43
44     # conditional expectation taken
45     valTree = tree.expCondMultiple(toTreeress)
```

Chapter 4

Continuation values objects and similar ones

In the first part, we develop the different continuation objects by using regression to calculate conditional expectations. Next, we explain the structure of the continuation object with tree to calculate the conditional expectations.

4.1 Continuation values objects with regression methods

In the first part we describe a way to store and use continuation values calculated when using regression methods to estimate conditional expectations. In a second part, we introduce an object used to interpolate a function both discretized on grids for its deterministic part and estimated by regressor for its stochastic part. The second object is similar to the first in spirit, but being dedicated to interpolation is more effective to use in simulations performed after the optimization part of a problem.

A third object is the continuation cut object used to approximate the concave or convex Bellman values by cuts.

It is used when the transition problem is solved using a LP.

At last, another continuation cut object is added to solve problems where the function to approximate by cuts is only locally concave. There is a need in this case to be able to localize where the function is concave and reconstruct domains where concavity is respected.

4.1.1 Continuation values object

A special case is the case where the state $X^{x,t}$ in the equation (2.1) can be separated into two parts $X^{x,t} = (X_1^{x,t}, X_2^{x,t})$ where

1. the first part is given by the following equation

$$dX_{s,1}^{x,t} = b(s, X_{s,1}^{x,t})ds + \sigma(s, X_{s,1}^{x,t})dW_s \quad (4.1)$$

and is not controlled: the stochastic process is exogenous,

2. the second part is given by the following equation

$$dX_{s,2}^{x,t} = b_a(t)ds \quad (4.2)$$

so that $X_2^{x,t}$ is a degenerate version of 2.1 without distribution, a representing the control.

This first case is encountered, for example, in the valuation of American options in finance. In this case, $X_1^{x,t}$ contains the values of the assets involved in the option and $X_2^{x,t}$ is for example an integer value process equal to one if the option is not exercised and to 0 if it has already been exercised.

Another classical case that occurs when dealing with stocks for example is a gas storage valuation. In this simple case, the process $X_1^{x,t}$ is the market value of the gas and $X_2^{x,t}$ is the position (in volume) in the gas storage. The library proposes to store the conditional expectation for all states $X_2^{x,t}$.

- $X_2^{x,t}$ will be stored at the points of the grid (see section 1)
- for each point i of the grid the conditional expectation of a function $g_i(X_2^{x,t})$ associated with the point i using a regressor (see chapter 2) can be calculated and stored so that the continuation value C is a function of $(X_1^{x,t}, X_2^{x,t})$.

C++ API

Regarding regressions, two constructors are provided

- The first is the default construction: it is used in the simulation algorithm with the `loadForSimulation` method to store the basis coefficients α_k^i for the grid point i (see the equation (2.1)),
- The second

```
1 ContinuationValue(const shared_ptr< SpaceGrid > & p_grid ,
2                  const shared_ptr< BaseRegression > & p_condExp ,
3                  const Eigen::ArrayXXd & p_cash)
```

with

- `p_grid` the grids associated with the deterministic control space,
- `p_condExp` the conditional expectation operator
- `p_cash` the function to regress as a function of the grid position (first dimension the number of simulations, second dimension the size of the grid)

This constructor builds for all points i all the α_k^i (see equation (2.1)).

The main methods provided are:

- a first method used in simulation allowing to load for the grid point i the coefficient α_k^i associated with the function g_i ,

```

1 void loadForSimulation(const shared_ptr< SpaceGrid > & p_grid ,
2                       const shared_ptr< BaseRegression > & p_condExp,
3                       const Eigen::ArrayXXd &p_values)

```

with

- `p_grid` the grid associated with the controlled deterministic space,
 - `p_condExp` the conditional expectation operator,
 - `p_values` the α_k^i for all grid points i (size the number of basis function, the number of grid points)
- a second method taking as input a point to be interpolated in the grid and returning the conditional expectation at the interpolated point for all simulations:

```

1 Eigen::ArrayXd getAllSimulations(const Eigen::ArrayXd &p_ptOfStock)

```

- a method taking as input an interpolator in the grid and returning the conditional expectation for all simulations at the interpolated point used to build the interpolator:

```

1 Eigen::ArrayXd getAllSimulations(const Interpolator<double> &p_interpol)

```

- a method taking as input a simulation number used in optimization and a point used to interpolate in the grid and returning the conditional expectation at the interpolated point for the given simulation used in optimization.

```

1 double getASimulation(const int &p_isim, const Eigen::ArrayXd &p_ptOfStock)

```

- a method taking as input a simulation number used in optimization and an interpolator in the grid and returning the conditional expectation at the interpolated point used to construct the interpolator for the given simulation used in optimization:

```

1 double getASimulation(const int &p_isim, const Interpolator<double> &
   p_interpol)

```

- a method which calculates the conditional expectation for a sample of $X_1^{x,t}$:

```

1 double getValue(const Eigen::ArrayXd &p_ptOfStock, const Eigen::ArrayXd &
   p_coordinates) const

```

where:

- `p_ptOfStock` the point where we interpolate the conditional expectation (a realization of $X_2^{x,t}$)
- `p_coordinates` the sample of $X_1^{x,t}$ used to estimate the conditional expectation
- and the function returns $C(X_1^{x,t}, X_2^{x,t})$.

Below we regress an identical function for all the points of the grid (here a grid of 4 points in dimension 1):

```

1      int sizeForStock = 4;
2      // second member to regress with a stock
3      ArrayXXd toRegress = ArrayXXd::Constant(p_nbSimul, sizeForStock, 1.);
4      // grid for stock
5      Eigen::ArrayXd lowValues(1), step(1);
6      lowValues(0) = 0. ;
7      step(0) = 1;
8      Eigen::ArrayXi nbStep(1);
9      nbStep(0) = sizeForStock - 1;
10     // grid
11     shared_ptr< RegularSpaceGrid > regular = MyMakeShared<RegularSpaceGrid>(lowValues,
12         step, nbStep);
13     // conditional expectation (local basis functions)
14     ArrayXi nbMesh = ArrayXi::Constant(p_nDim, p_nbMesh);
15     shared_ptr<LocalLinearRegression> localRegressor = MyMakeShared<
16         LocalLinearRegression>(false, x, nbMesh);
17
18     // create continuation value object
19     ContinuationValue continuation(regular, localRegressor, toRegress);
20
21     // regress with continuation value object
22     ArrayXd ptStock(1) ;
23     ptStock(0) = sizeForStock / 2; // point where we regress
24     // compute regression values for the current point for all the simulations
25     ArrayXd regressedByContinuation = continuation.getAllSimulations(ptStock);

```

Python API

Here is an example of the use of the mapping

```

1  # Copyright (C) 2016 EDF
2  # All Rights Reserved
3  # This code is published under the GNU Lesser General Public License (GNU LGPL)
4  import numpy as np
5  import unittest
6  import random
7  import math
8
9
10 # unit test for continuation values
11 #####
12
13 class testContValues(unittest.TestCase):
14
15     # test a regular grid for stocks and a local function basis for regression
16     def testSimpleGridsAndRegressor(self):
17         import pystopt.grids as StOptGrids
18         import pystopt.regression as StOptReg
19         # low value for the meshes
20         lowValues = np.array([1., 2., 3.], dtype=float)
21         # size of the meshes
22         step = np.array([0.7, 2.3, 1.9], dtype=float)
23         # number of steps
24         nbStep = np.array([3, 2, 4], dtype=np.int32)
25         # create the regular grid
26         #####
27         grid = StOptGrids.RegularSpaceGrid(lowValues, step, nbStep)
28         # simulation
29         nbSimul = 10000
30         np.random.seed(1000)
31         x = np.random.uniform(-1., 1., size=(1, nbSimul));
32         # mesh
33         nbMesh = np.array([16], dtype=np.int32)
34         # Create the regressor
35         #####
36         regressor = StOptReg.LocalLinearRegression(False, x, nbMesh)

```

```

37     # regressed values
38     toReal = (2+x[0,:]+(1+x[0,:])*(1+x[0,:]))
39     # function to regress
40     toRegress = toReal + 4*np.random.normal(0.,1,nbSimul)
41     # create a matrix (number of stock points by number of simulations)
42     toRegressMult = np.zeros(shape=(len(toRegress),grid.getNbPoints()))
43     for i in range(toRegressMult.shape[1]):
44         toRegressMult[:,i] = toRegress
45     # Now create the continuation object
46     #####
47     contOb = StOptReg.ContinuationValue(grid,regressor,toRegressMult)
48     # get back the regressed values at the point stock
49     ptStock= np.array([1.2,3.1,5.9],dtype=float)
50     regressValues = contOb.getAllSimulations(ptStock)
51     # do the same with an interpolator
52     interp = grid.createInterpolator(ptStock)
53     regressValuesInterp = contOb.getAllSimulations(interp)
54     # test create of an interpoaltion object mixing grids for stocks and regression for
55     #      uncertainties
56     #####
57     gridAndRegressed = StOptReg.GridAndRegressedValue(grid,regressor,toRegressMult)
58     # get back the regressed value for a point stock and an uncertainty
59     valRegressed = gridAndRegressed.getValue(ptStock, x[:,0])
60     # Now test simulations one by one
61     #####
62     for i in range(nbSimul):
63         regressed = gridAndRegressed.getValue(ptStock,x[:,i] )
64         diff = regressed -regressValues[i]
65         self.assertAlmostEqual(diff,0., 7, "test regression simulation by simulation")
66
67     # test some mapping of GneralSpaceGrid
68     def testGeneralGridInheritance(self):
69         from pystopt.grids import GeneralSpaceGrid, RegularSpaceGrid
70         from pystopt.regression import LocalLinearRegression, ContinuationValue
71
72         x = np.random.randn(5)
73         regressor = LocalLinearRegression([1])
74
75         regular = RegularSpaceGrid(np.array([0.]), np.array([0.5]), np.array([3]))
76         ContinuationValue(regular, regressor, x)
77
78         general = GeneralSpaceGrid([[0., 1., 1.2, 1.5]])
79         ContinuationValue(general, regressor, x)
80
81 if __name__ == '__main__':
82     unittest.main()

```

4.1.2 The GridAndRegressedValue object

As explained above, when we want to interpolate a partially discretized function on a grid and by regression, a specific object can be used. As for the continuation object, it has a `getValue` to estimate the function at a state with both a deterministic and a stochastic part.

C++ API

The object has five constructors and we have only described the two most commonly used:

- The first one

```

1  GridAndRegressedValue(const std::shared_ptr< SpaceGrid > &p_grid ,
2                        const std::shared_ptr< BaseRegression > &p_reg,
3                        const Eigen::ArrayXXd &p_values)

```

with

- `p_grid` the grid associated with the deterministic control space,
- `p_reg` the regressor object
- `p_values` the functions at certain points of the deterministic and stochastic grid.

- A second constructor stores only the grid and the regressor:

```

1  GridAndRegressedValue(const std::shared_ptr< SpaceGrid > &p_grid ,
2                        const std::shared_ptr< BaseRegression > &p_reg)

```

The main methods are the following ones:

- the main method for calculating the function $C(X_{1,s}^{x,t}, X_{2,s}^{x,t})$ value for a point $X_s^{x,t} = (X_{1,s}^{x,t}, X_{2,s}^{x,t})$ where $X_{2,s}^{x,t}$ is on the grid and $X_{1,s}^{x,t}$ is the part treated by regression.

```

1  double getValue(const Eigen::ArrayXd &p_ptOfStock, const Eigen::ArrayXd &
2                 p_coordinates) const

```

where:

- `p_ptOfStock` $X_{2,s}^{x,t}$ part of $X_s^{x,t}$
- `p_coordinates` $X_{1,s}^{x,t}$ part of $X_s^{x,t}$.

- the method `getRegressedValues` which allows to obtain all the regression coefficients for all points of the grid. The returned array has a size (number of basis functions, number of points on the grid)

```

1  Eigen::ArrayXXd getRegressedValues() const

```

- the method `setRegressedValues` allows to store all the values of the regressed coefficients on a grid of a function of $X_s^{x,t} = (X_{1,s}^{x,t}, X_{2,s}^{x,t})$.

```

1  void setRegressedValues(const Eigen::ArrayXXd &p_regValues)

```

where `p_regValues` has a size (number of function basis, number of points on the grid).

Python API

The Python API is similar to that of the `ContinuationValue` object (see Section 4.1.1).

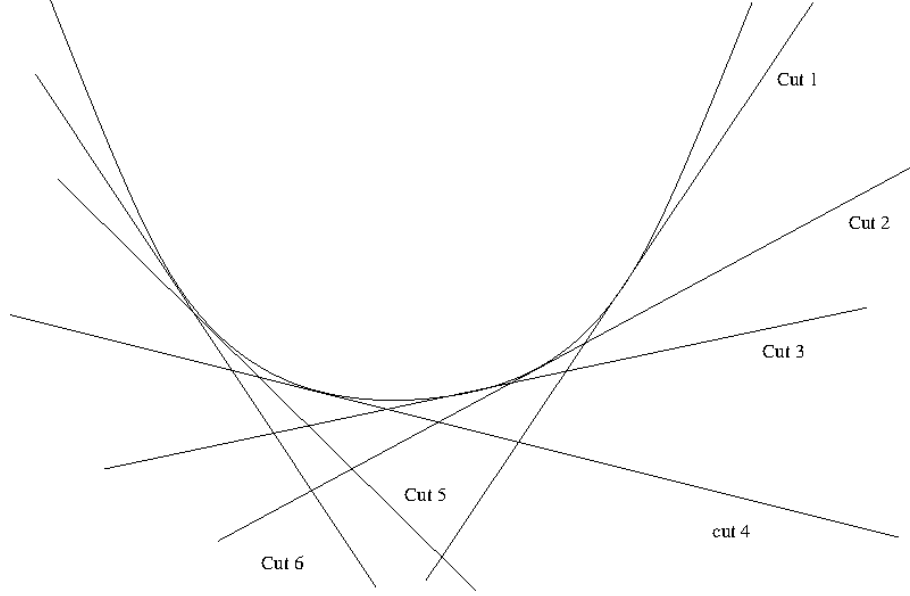


Figure 4.1: Bellman cuts

4.1.3 The continuation cut object for concave problems

Suppose that the control problem is continuous and that the state of the system has the dynamics given by (4.1) et (4.2). This is the case for certain modeled storage associated with the maximization of a certain objective function. The Bellman value associated with this problem is then concave. For a given value of a margin process $X_{s,1}^{x,t}$, the Bellman curve can be approximated by cuts (see 4.1) By solving a PL for a given uncertainty and state in the storage levels $\hat{y}_i$ in dimension d , we obtain a cut

$$\kappa(X_{s,1}^{x,t}, y) = a_0(X_{s,1}^{x,t}) + \sum_{i=1}^g a_i(X_{s,1}^{x,t})(y_i - \hat{y}_i)$$

For $s' \leq s$ a conditional cut can be obtained by calculating

$$\theta(X_{s',1}^{x,t}, y) = \mathbb{E} \left[a_0(X_{s,1}^{x,t}) | X_{s',1}^{x,t} \right] + \sum_{i=1}^d \mathbb{E} \left[a_i(X_{s,1}^{x,t}) | X_{s',1}^{x,t} \right] (y_i - \hat{y}_i)$$

Using a regressor (see chapter 2) it is possible to represent each conditional cut on a basis for $j = 0, \dots, d$.

$$\mathbb{E} \left[a_i(X_{s,1}^{x,t}) | X_{s',1}^{x,t} \right] = \sum_{j=1}^N a_{i,j} \psi_j(X_{s',1}^{x,t}) \quad (4.3)$$

where ψ_j corresponds to a basis function.

C++ API

The first is the default construction: it is used in the simulation algorithm with the `loadForSimulation` method to store the database. Regarding regressions, two constructors are provided

- The first is the default construction: it is used in simulation algorithm with the `loadForSimulation` method to store the basis coefficients $a_{i,j}^k$ for the grid point k ,
- The second is

```
1 ContinuationCuts(const shared_ptr< SpaceGrid > & p_grid ,
2                 const shared_ptr< BaseRegression > & p_condExp ,
3                 const Eigen::ArrayXXd &p_values)
```

with

- `p_grid` the grids associated with the deterministic control space,
- `p_condExp` the conditional expectation operator
- `p_values` the coefficients of the cut to regress according to the position of the grid (first dimension the number of simulations by the number of components of the cut (nb storage+1), second dimension the size of the grid)

This constructor builds for all stock points the coefficients $a_{i,j}$ of the cuts (4.3). Note that for a stock point k with the coordinates y^k , the stored coefficients are $\hat{a}_{0,j}^k = a_{0,j}^k - \sum_{i=1}^d a_{i,j}^k y_i^k$ and the $\hat{a}_{i,j}^k = a_{i,j}^k$, $i = 1, \dots, d$. The conditional cut can then be written:

$$\theta(X_{s',1}^{x,t}, y) = \sum_{j=1}^N \hat{a}_{0,j} \psi_j(X_{s',1}^{x,t}) + \sum_{i=1}^d \sum_{j=1}^N \hat{a}_{i,j} \psi_j(X_{s',1}^{x,t}) y_i$$

The main methods provided are:

- a first method used in simulation allowing to load for the grid point i the coefficient α_k^i associated with the function g_i ,

```
1 void loadForSimulation(const shared_ptr< SpaceGrid > & p_grid ,
2                       const shared_ptr< BaseRegression > & p_condExp ,
3                       const Eigen::Array<Eigen::ArrayXXd> &p_values)
```

with

- `p_grid` the grid associated with the controlled deterministic space,
- `p_condExp` the conditional expectation operator,
- `p_values` the $a_{i,j}$ coefficients to reconstruct the cuts: its size corresponds to the number of cutting coefficients. The i element of `p_values` then allows you to store the coefficients $a_{i,j}$ for $j = 1, \dots, N$ and all stock points.
- a second method taking as input the description of a hypercube (nb storages,2) described by its extreme coordinates:

- The coordinate (i,0) corresponds to the minimum coordinate value in dimension i
- The coordinate (i,1) corresponds to the maximum coordinate value in dimension i

```
1 Eigen::ArrayXXd getCutsAllSimulations(const Eigen::ArrayXXd &p_hypStock) const
```

It return an array of cuts coefficients for all particles state stored in its BaseRegression member.

- The first dimension corresponds to the number of cut coefficients by the number of simulations.
- The second dimension corresponds to the number of points in the hypercube p_hypStock.

- a method to obtain an array of cuts for a given uncertainty.

```
1 Eigen::ArrayXXd getCutsASim(const Eigen::ArrayXXd &p_hypStock, const Eigen::ArrayXd &p_coordinates) const
```

where:

- p_hypStock corresponds to a hypercube used to select certain stock points as before,
- p_coordinates corresponds to the coordinates of the uncertainty to be considered.

It returns an array with in first dimension the cut coefficient number, the second dimension corresponds to the number of the cut (corresponding to a stock point in the hypercube).

- a method that permits to get the coefficients calculated.

```
1 const Eigen::Array< Eigen::ArrayXXd, Eigen::Dynamic, 1 > &getValues() const
```

Python API

Here is an example of the use of the mapping

```
1 # Copyright (C) 2016 EDF
2 # All Rights Reserved
3 # This code is published under the GNU Lesser General Public License (GNU LGPL)
4 import numpy as np
5 import unittest
6 import random
7 import math
8 import pystopt.grids as StOptGrids
9 import pystopt.regression as StOptReg
10
11
12 # unit test for continuation values
13 #####
14
15 class testContValues(unittest.TestCase):
```

```

16
17 # unit test for continuation cuts
18 def testSimpleGridsAndRegressor(self):
19     # low value for the meshes
20     lowValues = np.array([1.,2.,3.], dtype=float)
21     # size of the meshes
22     step = np.array([0.7,2.3,1.9], dtype=float)
23     # number of steps
24     nbStep = np.array([3,2,4], dtype=np.int32)
25     # create the regular grid
26     #####
27     grid = StOptGrids.RegularSpaceGrid(lowValues, step, nbStep)
28     # simulation
29     nbSimul = 10000
30     np.random.seed(1000)
31     x = np.random.uniform(-1.,1., size=(1,nbSimul));
32     # mesh
33     nbMesh = np.array([16], dtype=np.int32)
34     # Create the regressor
35     #####
36     regressor = StOptReg.LocalLinearRegression(False, x, nbMesh)
37     # regressed values
38     toReal = (2+x[0,:]+(1+x[0,:])*(1+x[0,:]))
39     # function to regress
40     toRegress = toReal + 4*np.random.normal(0.,1,nbSimul)
41     # fictitious cuts with 0 sensibility (1 dimension)
42     toRegressCuts = np.zeros(shape=(4*len(toRegress), grid.getNbPoints()))
43     for i in range(toRegressCuts.shape[1]):
44         toRegressCuts[:,len(toRegress),i] = toRegress
45
46     # Now create the continuation cut object
47     #####
48     contOb = StOptReg.ContinuationCut(grid, regressor, toRegressCuts)
49     hyperCube = np.array([[lowValues[0], lowValues[0]+step[0]*nbStep[0],
50                             [lowValues[1], lowValues[1]+step[1]*nbStep[1]],
51                             [lowValues[2], lowValues[2]+step[2]*nbStep[2]]])
52     regressCuts = contOb.getCutsAllSimulations(hyperCube)
53
54
55 if __name__ == '__main__':
56     unittest.main()

```

4.1.4 The continuation cut objet for non concave problems

This continuation cut object is associated with 1D and 2D grids, the refinement of which is described in Section 1.4. As in the previous section, this continuation object is used to optimize a functional via dynamic programming, solving transition problems with LP and cuts. Since the evaluated function is not concave, the cuts must be generated in a domain where the solution is concave. For example, in Figure 1.15, the grid is refined, and the function approximated by cuts is assumed to be locally concave in domains separated by a blue line. Therefore, concavity is broken for all meshes crossed by the blue lines. Figure 4.2 shows all of these meshes in blue.

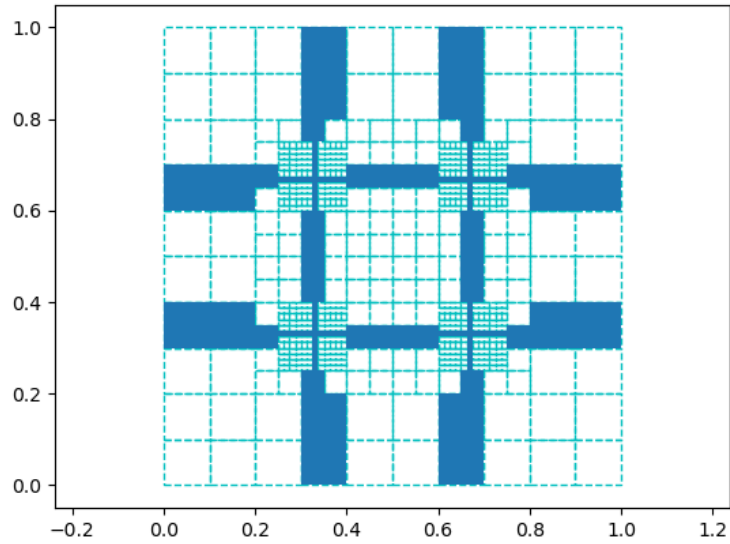


Figure 4.2: Local meshes with broken concavity

Figure 4.3 shows the meshes gathered into grids (rectangles) where the function is concave. The number of grids generated should be as low as possible to reduce the computing time for the stock problem in dynamic programming. In this case, the number of linear programming (LP) resolutions is proportional to the number of grids generated.

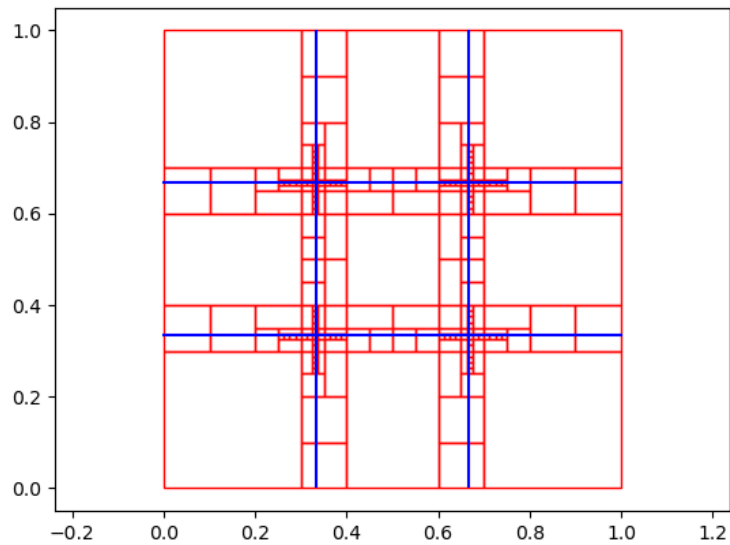


Figure 4.3: Meshes gathered in domain where function is concave

C++ API

The continuation object has the following main constructor

```
1 ContinuationCutsGridAdaptNonConcave(const std::shared_ptr< GridAdaptBase > &p_grid,
2                                     const std::shared_ptr< BaseRegression > &p_condExp,
3                                     const Eigen::ArrayXXd &p_values)
```

with

- **p_grid** an adapting grid derived from **GridAdaptBase**, either 1D or 2D,
- **p_condExp** the conditional expectation operator similar to all regression objects
- **p_values** the coefficients of the cut to regress according to the position of the grid (first dimension the number of simulations by the number of components of the cut (nb storage+1), second dimension the size of the grid).

This constructor builds for all stock points the coefficients $a_{i,j}$ of the cuts (4.3). Note that for a stock point k with the coordinates y^k , the stored coefficients are $\hat{a}_{0,j}^k = a_{0,j}^k - \sum_{i=1}^d a_{i,j}^k y_i^k$ and the $\hat{a}_{i,j}^k = a_{i,j}^k$, $i = 1, \dots, d$. This is similar to the description in 4.1.3.

A first method permits to load the continuation values components in an existing continuation cut object:

```
1 void loadForSimulation(const shared_ptr< GridAdaptBase > & p_grid ,
2                       const shared_ptr< BaseRegression > & p_condExp,
3                       const Eigen::Array<Eigen::ArrayXXd> &p_values)
```

with

- **p_grid** the grid associated with the controlled deterministic space,
- **p_condExp** the conditional expectation operator,
- **p_values** the $a_{i,j}$ coefficients to reconstruct the cuts: its size corresponds to the number of cutting coefficients. The i element of **p_values** then allows you to store the coefficients $a_{i,j}$ for $j = 1, \dots, N$ and all stock points.

A second method permits, an hypercube being given, to get back all meshed (stored as grids) intersecting the hypercube and the associated cuts for all the simulation used in optimization

```
1 std::vector< std::pair <std::shared_ptr< GridAdaptBase>, std::shared_ptr<Eigen::ArrayXXd>
  > > getCutsAllSimulations(const Eigen::ArrayXXd &p_hypStock) const;
```

with

- **p_hypStock** the hypercube of size (1,2) in dimension 1, (2,2) in dimension 2. The $(i, 0)$ element corresponds to the minimal coordinate in dimension i , while the $(i, 1)$ element corresponds to the maximal coordinate in dimension i .
- The return is the list of grids intersecting the hypercube and an array giving the cuts for the given grids:

- the first dimension is the number simulations by the numbers of elements describing the cuts,
- the second dimension is the number of points composing the grid.

A third method does the same but for a simulation described by the realization of the uncertainties.

```
1  std::vector< std::pair <std::shared_ptr< GridAdaptBase>, std::shared_ptr<Eigen::
    ArrayXXd> > > getCutsASim(const Eigen::ArrayXXd & p_hypStock, const Eigen::
    ArrayXd &p_coordinates) const
```

with

- `p_hypStock` the hypercube as above,
- `p_coordinates` the coordinates of uncertainties where the cuts are calculated

It returns the list of meshes (stored as grids) intersecting the hypercube and an array giving the cuts for the given grid:

- the first dimension is the number of elements describing the cuts,
- the second dimension is the number of points composing the grid.

A fourth method does the same but for a simulation in optimization identified by it integer `p_isim`.

```
1  std::vector< std::pair <std::shared_ptr< GridAdaptBase>, std::shared_ptr<Eigen::
    ArrayXXd> > > getCutsASim(const Eigen::ArrayXXd & p_hypStock, const int &
    p_isim) const
```

A fourth and fifth constructors do the same as `getCutsASim` but try to gather grids/meshes together to get the smallest number of possible grids. Each grid in the vector of grids resulting from this call is either a grid where the solution is concave, either it corresponds to a single mesh where concavity is broken. These grids result from an algorithm given figures similar to figure 4.3. The return is the same as `getCutsASim`.

```
1  std::vector< std::pair <std::shared_ptr< GridAdaptBase>, std::shared_ptr<Eigen::ArrayXXd
    > > > getCutsConcGatherASim(const Eigen::ArrayXXd & p_hypStock, const Eigen::
    ArrayXd &p_coordinates, const bool & p_bStrict) const
```

- `p_hypStock` the hypercube as above,
- `p_coordinates` the coordinates of uncertainties where the cuts are calculated
- `p_bStrict` is a bool. When true, 2 grids are gathered only if a strict concavity is verified. The strict concavity is checked using derivatives of the function and the values of the function at each node. When false, only a pseudo concavity is checked : the 2 grids are gathered if the derivatives are decreasing in all directions.

```
1  std::vector< std::pair <std::shared_ptr< GridAdaptBase>, std::shared_ptr<Eigen::ArrayXXd
    > > > getCutsConcGatherASim(const Eigen::ArrayXXd & p_hypStock, const int &
    p_isim, const bool & p_bStrict)
```

- `p_hypStock` the hypercube as above,
- `p_isim` the number of the simulation used
- `p_bStrict` is a bool. When true, 2 grids are gathered only if a strict concavity is verified. The strict concavity is checked using derivatives of the function and the values of the function at each node. When false, only a pseudo concavity is checked : the 2 grids are gathered if the derivatives are decreasing in all directions.

4.2 Continuation objects and associated with trees

4.2.1 Continuation object

Likewise, instead of using a regressor, a tree can be used to create a continuation object.

C++ API

As for the object `ContinuationValue`, two constructors are provided:

- A default constructor, allowing to load the grid coefficients at each node of the tree with the `loadForSimulation` method,
- And the second:

```

1 ContinuationValueTree(const std::shared_ptr< SpaceGrid > &p_grid,
2                       const std::shared_ptr< Tree > &p_condExp,
3                       const Eigen::ArrayXXd &p_valuesNextDate)

```

with

- `p_grid` the grids associated with the deterministic control space,
- `p_condExp` the tree object used to calculate the conditional expectation: by taking a few values defined at the nodes of the following date, it calculates the expected values conditional on each node at the current date.
- `p_valuesNext` the value of the function at the following date (first dimension the number of nodes on the following date, second dimension the size of the grid)

The main methods provided are:

- a first method used in simulation allowing to load for the grid point i the expected value of the function g_i (valuesNextDate) for all the nodes in the tree at the current date,

```

1 void loadForSimulation(const shared_ptr< SpaceGrid > & p_grid ,const Eigen::
   ArrayXXd &p_values)

```

with

- `p_grid` the grid associated with the controlled deterministic space,

- `p_values` the continuation values for all the nodes and stock points (size: the number of nodes by number of grid points at the current date)
- a second method taking as input a point to be interpolated in the grid and returning the conditional expectation at the interpolated point for all nodes:

```
1 Eigen::ArrayXd getValueAtNodes(const Eigen::ArrayXd &p_ptOfStock)
```

- a method taking as input an interpolator in the grid and returning the conditional expectation for all the nodes at the interpolated point used to build the interpolator:

```
1 Eigen::ArrayXd getValueAtNodes(const Interpolator<double> &p_interpol)
```

- a method taking as input a node number used in optimization and a point used to interpolate in the grid and returning the conditional expectation at the interpolated point for the given node used in optimization.

```
1 double getValueAtANode(const int &p_node, const Eigen::ArrayXd &p_ptOfStock)
```

- a method taking as input a simulation number used in optimization and an interpolator in the grid and returning the conditional expectation at the interpolated point used to build the interpolator for the given node used in optimization:

```
1 double getValueAtANode(const int &p_node, const Interpolator<double> &
    p_interpol)
```

- a method which makes it possible to recover all the conditional expectations for all the nodes:

```
1 double getValues() const
```

Python API

Used when importing the `StOptTree` module, the syntax is similar to that of `c++`. Following examples in the section 3.0.1;

```
1 # continuation object
2 continuation = StOptTree.ContinuationValueTree(grid, tree, toTreeress.transpose())
3
4 # interpolation point
5 ptStock = np.array([0.5*nbStock])
6
7 # conditional expectation using continuation object
8 treeByContinuation = continuation.getValueAtNodes(ptStock);
```

4.2.2 GridTreeValues

This object allows you to interpolate in certain grid values for a function defined at the node values and the grid values.

The constructor

```
1 GridTreeValue(const std::shared_ptr< SpaceGrid > &p_grid,
2 const Eigen::ArrayXXd &p_values)
```

where:

- `p_grid` the grids associated with the deterministic control space,
- `p_values` value to store at nodes and on grid (size : number of nodes at the current date by number of points in the grid)

The methods:

- The following is used to interpolate at a given stock point for a given node

```
1 double getValue(const Eigen::ArrayXd &p_ptOfStock, const int &p_node) const
```

- `p_ptOfStock` corresponds to a value of $X_{2,s}^{x,t}$ part of $X_s^{x,t}$
- `p_node` node number in the tree describing $X_{1,s}^{x,t}$

- The second gives the interpolated values at all the nodes

```
1 Eigen::ArrayXd getValues(const Eigen::ArrayXd &p_ptOfStock) const
```

Python API

Importing the `StOptTree`, previous constructor and methods are available.

4.2.3 Continuation Cut with trees

As for the regressor (section 4.1.3), we can provide cuts during the approximation of a concave or convex function at each node of the tree.

C++ API

- The first is the default construction: it is used in the simulation algorithm with the `loadForSimulation` method to load the values at the grid points nodes,
- The second

```
1 ContinuationCutsTree(const std::shared_ptr< SpaceGrid > &p_grid,
2                       const std::shared_ptr< Tree > &p_condExp,
3                       const Eigen::ArrayXXd &p_values)
```

with

- `p_grid` the grids associated with the controlled deterministic space,
- `p_condExp` the conditional expectation operator for tree
- `p_values` the coefficients of the cut from which we take the conditional expectation depending on the grid position (first dimension the number of nodes by the number of components of the cut (nb storage+1), second dimension the grid size)

This constructor constructs for all the stock points the coefficients of the cuts a_i for $i = 0, d$. Note that for a stock point k with the coordinates y^k , the stored coefficients are $\hat{a}_0^k = a_0^k - \sum_{i=1}^d a_i^k y_i^k$ and the $\hat{a}_i^k = a_i^k$, $i = 1, \dots, d$ so that a cut has an affine representation at a point y : $\hat{a}_0^k + \sum_{i=1}^d \hat{a}_i^k y_i$.

The main methods provided are:

- a first method used in simulation allowing to load for the grid point i the values of cuts at the nodes.

```
1 void loadForSimulation(const shared_ptr< SpaceGrid > & p_grid ,
2                       const shared_ptr< Tree > & p_condExp ,
3                       const const std::vector< Eigen::ArrayXXd > & p_values)
```

with

- **p_grid** the grid associated with the deterministic controlled space,
 - **p_condExp** the conditional expectation operator by tree,
 - **p_values** the a_i coefficients to reconstruct the cuts: its size corresponds to the number of cutting coefficients. The element i of **p_values** then allows you to store the coefficients a_i for all the nodes and all the stock points.
- a second method taking as input the description of a hypercube (nb storages,2) described by its extreme coordinates:
 - (i,0) coordinate corresponds to minimum coordinate value in dimension i
 - (i,1) coordinate corresponds to maximum coordinate value in dimension i

```
1 Eigen::ArrayXXd getCutsAllNodes(const Eigen::ArrayXXd &p_hypStock) const
```

It returns an array of cut coefficients for all nodes in the tree structure at the grid points inside the hypercube.

- The first dimension corresponds to the number of cuts coefficients by the number of nodes.
 - The second dimension corresponds to the number of points in the hypercube **p_hypStock**.
- a method to obtain an array of cuts for a given node.

```
1 Eigen::ArrayXXd getCutsANode(const Eigen::ArrayXXd &p_hypStock, const int &p_node
    ) const
```

where:

- **p_hypStock** corresponds to a hypercube used to select certain stock points as before,
- **p_node** corresponds to the node number in the tree structure.

It returns an array with in the first dimension the cut coefficient number, the second dimension corresponds to the number of the cutting (corresponding to a stock point in the hypercube).

- a method for calculating the coefficients.

```
1  const std::vector< Eigen::ArrayXXd>      getValues() const
```

Python API

By importing the `StOptTree` module, the `ContinuationCutsTree` object is available in Python.

Part III

Solving optimization problems with dynamic programming methods

Chapter 1

Creating simulators

In order to optimize the control problem, the user must develop simulators allowing to trace some trajectories of uncertainties. These trajectories are used during optimization or in a simulation part testing the optimal control.

1.1 Regression methods simulators

In the following, we assume that we have developed a Simulator generating Monte Carlo simulations at different optimizations dates. In order to use the different frameworks developed in the following, we assume that the simulator is derived from the abstract class `SimulatorDPBase`.

```
1 // Copyright (C) 2016 EDF
2 // All Rights Reserved
3 // This code is published under the GNU Lesser General Public License (GNU LGPL)
4 #ifndef SIMULATORDPBASE_H
5 #define SIMULATORDPBASE_H
6 #include <Eigen/Dense>
7
8 /* \file SimulatorDPBase.h
9  * \brief Abstract class for simulators for Dynamic Programming Programms
10  * \author Xavier Warin
11  */
12
13 namespace StOpt
14 {
15     /// \class SimulatorDPBase SimulatorDPBase.h
16     /// Abstract class for simulator used in dynamic programming
17     class SimulatorDPBase
18     {
19
20
21     public :
22
23         /// \brief Constructor
24         SimulatorDPBase() {}
25         /// \brief Destructor
26         virtual ~SimulatorDPBase() {}
27         /// \brief get current markovian state : dimension of the problem for the first
28             dimension , second dimension the number of Monte Carlo simulations
29         virtual Eigen::MatrixX<double> getParticles() const = 0;
30         /// \brief a step forward for simulations
31         virtual void stepForward() = 0;
32         /// \brief a step backward for simulations
33         virtual void stepBackward() = 0;
```

```

33  /// \brief a step forward for simulations
34  /// \return current particles (markovian state as assets for example) (dimension of
    the problem times simulation number)
35  virtual Eigen::MatrixXd stepForwardAndGetParticles() = 0;
36  /// \brief a step backward for simulations
37  /// \return current particles (markovian state as assets for example) (dimension of
    the problem times simulation number)
38  virtual Eigen::MatrixXd stepBackwardAndGetParticles() = 0;
39  /// \brief get back dimension of the regression
40  virtual int getDimension() const = 0;
41  /// \brief get the number of steps
42  virtual int getNbStep() const = 0;
43  /// \brief Get the current step size
44  virtual double getStep() const = 0;
45  /// \brief Get current time
46  virtual double getCurrentStep() const = 0 ;
47  /// \brief Number of Monte Carlo simulations
48  virtual int getNbSimul() const = 0;
49  /// \brief Permit to actualize for one time step (interest rate)
50  virtual double getActuStep() const = 0;
51  /// \brief Permits to actualize at the initial date (interest rate)
52  virtual double getActu() const = 0 ;
53
54 };
55 }
56 #endif /* SIMULATORDPBASE_H */

```

Suppose that the Simulator is a Black-Scholes simulator for the assets P , simulating M the Monte Carlo simulations, on the dates $N+1$ $t_0, \dots, t_N$, the Markov state for the particle j , the date t_i , the Monte Carlo simulation k and the asset p is $X_{p,i}^k$ and we give below the meaning of the different methods of `SimulatorDPBase`:

- the `getParticle` method gives at the current optimization/simulation date t_i the Markov states $X_{p,i}^k$ in a matrix A such that $A(p, k) = X_{p,i}^k$,
- the `stepForward` method is used while simulating the assets evolution in forward: a step forward is realized from t_i to t_{i+1} and Brownian motions used for the assets are updated at the new time step,
- the `stepBackward` method is used for the simulation of the asset from the last date to time 0. This method is used during an asset optimization by Dynamic Programming,
- the method `stepForwardAndGetParticles`: second and first method in one call,
- the method `stepBackwardAndGetParticles`: third and first method in one call,
- the method `getDimension` returns the number of assets,
- the method `getNbStep` returns the number of steps (N),
- the method `getStep` returns the number of steps $t_{i+1} - t_i$ at the current time t_i ,
- the method `getNbSimul` returns M .
- the method `getActuStep` returns the actualization factor over a time step
- the method `getActu` returns a discount factor at date “0”.

1.2 Simulators for trees

In order to develop solvers using tree-based methods, the user has to create a simulator derived from the `SimulatorDPBaseTree` class. This simulator reads at each date in a `geners` archive, the values of the uncertainties at the nodes and the probability transition. It is used deterministically in backward mode: the values of the nodes are all explored sequentially. In forward mode, it allows to sample discrete values of the state through the tree.

```

1 // Copyright (C) 2019 EDF
2 // All Rights Reserved
3 // This code is published under the GNU Lesser General Public License (GNU LGPL)
4 #ifndef SIMULATORDPBASETREE_H
5 #define SIMULATORDPBASETREE_H
6 #include <Eigen/Dense>
7 #include "geners/BinaryFileArchive.hh"
8
9 /* \file SimulatorDPBaseTree.h
10 * \brief Abstract class for simulators for Dynamic Programming Programms with tree
11 * \author Xavier Warin
12 */
13
14 namespace StOpt
15 {
16     /// \class SimulatorDPBaseTree SimulatorDPBaseTree.h
17     /// Abstract class for simulator used in dynamic programming with trees
18     class SimulatorDPBaseTree
19     {
20     protected :
21
22         std::shared_ptr<gs::BinaryFileArchive> m_binForTree ; ///< archive for tree
23         Eigen::ArrayXd m_dates ; ///< list of dates in the archive
24         int m_idateCur ; ///< current date index
25         Eigen::ArrayXXd m_nodesCurr ; ///< storing coordinates of the nodes at current date (
26             dim, nbnodes)
27         Eigen::ArrayXXd m_nodesNext; ///< storing coordinates of the nodes at next date (dim,
28             nbnodes)
29         std::vector<double> m_proba ; ///< value stores probability to go from on node at
30             index m_dateCur to node at next date m_dateNext.
31         std::vector< std::vector< std::array<int, 2> > > m_connected ; ///

```

```

51
52 /// \brief sample one simulation in forward mode
53 /// \param p_nodeStart starting node
54 /// \param p_randUni uniform random in [0,1]
55 /// \return node reached
56 int getNodeReachedInForward(const int &p_nodeStart, const double &p_randUni) const ;
57
58
59 /// \brief a step backward for simulations
60 virtual void stepBackward() = 0;
61 /// \brief get back dimension of the problem
62 virtual int getDimension() const
63 {
64     return m_nodesCurr.rows();
65 }
66 /// \brief get the number of steps
67 virtual int getNbStep() const
68 {
69     return m_dates.size() - 1;
70 }
71 /// \brief Number of nodes at current date
72 virtual int getNbNodes() const
73 {
74     return m_nodesCurr.cols();
75 }
76 /// \brief Number of nodes at next date
77 virtual int getNbNodesNext() const
78 {
79     return m_nodesNext.cols();
80 }
81
82 /// \brief get back dates
83 inline Eigen::ArrayXd getDates() const
84 {
85     return m_dates;
86 }
87
88 /// \brief get back the last date index
89 inline int getBackLastDateIndex() const
90 {
91     return m_dates.size() - 1;
92 }
93
94 /// \brief get back connection matrix :for each node at current date, give the node
    connected
95 std::vector< std::vector< std::array<int, 2 > > > getConnected() const
96 {
97     return m_connected ;
98 }
99
100 /// \brief get back probabilities
101 inline std::vector< double > getProba() const
102 {
103     return m_proba;
104 }
105
106 /// \brief get current nodes
107 inline Eigen::ArrayXXd getNodes() const
108 {
109     return m_nodesCurr ;
110 }
111
112 /// \brief get nodes at next date
113 inline Eigen::ArrayXXd getNodesNext() const
114 {
115     return m_nodesNext ;
116 }
117
118 /// \brief Get number of simulations used in forward

```

```

119     virtual inline int getNbSimul() const = 0;
120
121
122     /// \brief Get node number associated to a node
123     /// \param p_nodeIndex index of the node
124     virtual Eigen::ArrayXd getValueAssociatedToNode(const int &p_nodeIndex) const = 0;
125
126     /// \brief get node associated to a simulation
127     /// \param p_isim simulation number
128     /// \return number of the node associated to a simulation
129     virtual int getNodeAssociatedToSim(const int &p_isim) const = 0;
130 };
131 }
132 #endif /* SIMULATORDPBASETREE_H */

```

When designing their tree, the user must call the based simulator constructor by providing a `geners` archive giving

- An Eigen `ArrayXd` of the set of dates (size N) associated with the tree.
- A vector of probabilities P at each of the first $N - 1$ dates.
- A vector of vector of pair of int at each of the first $N - 1$ dates. Such a vector v , at a given date, has the size equal to the number of nodes in the tree at this date. For a node i , $v[i]$ is the vector of arrival nodes number and probability index in P . Then $v[i][j].first$ is the number of a node at next date connected to node i at current date. The transition probability is given by $P[v[i][j].second]$.

In the `geners` archive, storage is carried out as in the `dump` function in the file `TrinomialTreeOUSimulator.cpp` storing the connection matrix probabilities of a trinomial tree.

```

1 void TrinomialTreeOUSimulator::dump(const std::string &p_name, const Eigen::ArrayXi &
   p_index)
2 {
3     gs::BinaryFileArchive binArxiv(p_name.c_str(), "w");
4     ArrayXd ddates(p_index.size());
5     for (int i = 0; i < p_index.size(); ++i)
6         ddates(i) = m_dates(p_index(i));
7     binArxiv << gs::Record(ddates, "dates", "");
8     for (int i = 0 ; i < p_index.size(); ++i)
9     {
10         ArrayXXd points = getPoints(p_index(i));
11         binArxiv << gs::Record(points, "points", "");
12     }
13     for (int i = 0 ; i < p_index.size() - 1; ++i)
14     {
15         ArrayXXd proba = getProbability(p_index(i), p_index(i + 1));
16         pair< vector< vector< std::array<int, 2> > >, vector< double > > conAndProb =
            calConnected(proba);
17         binArxiv << gs::Record(conAndProb.second, "proba", "");
18         binArxiv << gs::Record(conAndProb.first, "connection", "");
19     }
20     binArxiv.flush();

```

The different methods that the user has to provide are

- the `stepForward` method is used when simulating the evolution of forward assets: a step forward is made from t_i to t_{i+1} and samples are generated to give discrete uncertainties in the tree.

- the `stepBackward` method is used when optimizing an asset from the last date to time 0 by Dynamic Programming. The structure of the tree should be updated (probabilities, connection between nodes)
- the `getNbSimul` giving the number of samples used in forward mode,
- the `getValueAssociatedToNode` method taking the number of a node and giving back the state associated with this node,
- the `getNodeAssociatedToSim` method giving for a trajectory number in forward mode, the number of the node visited at current date.

An example of simulator for HJM model with trinomial tree for the OU process is `MeanRevertingSimulatorTree`.

Chapter 2

Using conditional expectation to solve simple problems

In this chapter, we give some examples for valuing an American option. This use of conditional expectation operators can be extended to many stochastic problems using these previously developed objects.

2.1 American option by regression

2.1.1 The American option valuing by Longstaff–Schwartz

Suppose in this example that the payoff of the American option is given by g and that the interest rate is 0. The value of the option is given by

$$P_t = \text{esssup}_{\tau \in \mathcal{T}_{[t,T]}} \mathbb{E}(g(\tau, X_\tau) \mid \mathcal{F}_t) \quad \text{for } t \leq T \quad \mathbb{P} - \text{a.s.}, \quad (2.1)$$

where $\mathcal{T}_{[t,T]}$ denotes the set of stopping times with values in $[t, T]$.

We recall the classic Longstaff–Schwartz Algorithm 8 estimating the empirical conditional expectation using the regression estimation seen previously.

Algorithm 8 Algorithm with regression [optimal exercise time estimation]

Initialization:

Set $\hat{\tau}_\kappa^{1,\pi,(j)} := T, j \leq N$

Backward induction:

for $i = \kappa - 1$ to 0 **do**

set $\hat{\tau}_i^{1,\pi} := t_i \mathbf{1}_{A_i^1} + \hat{\tau}_{i+1}^{1,\pi} \mathbf{1}_{(A_i^1)^c}$ where $A_i^1 := \{g(t_i, X_{t_i}) \geq \hat{\mathbb{E}}[g(\hat{\tau}_{i+1}^{1,\pi}, X_{\hat{\tau}_{i+1}^{1,\pi}}) \mid \mathcal{F}_{t_i}]\}$.

end for

Price estimator at 0: $\hat{P}_0^{1,\pi} := \hat{\mathbb{E}}[g(\hat{\tau}_0^{1,\pi}, X_{\hat{\tau}_0^{1,\pi}})]$.

American option by regression with the C++ API

We value in the algorithm below an American option using a simulator `p_sim`, a regressor `p_regressor`, a payoff function `p_payoff`:

```
1  double step = p_sim.getStep(); // time step increment
2  // asset simulated under the neutral risk probability: get the trend of the first
   asset to get the interest rate
3  double expRate = exp(-step * p_sim.getMu()(0));
4  // Terminal pay off
5  VectorXd Cash(p_payOff(p_sim.getParticles()));
6  for (int iStep = 0; iStep < p_sim.getNbStep(); ++iStep)
7  {
8      shared_ptr<ArrayXXd> asset(new ArrayXXd(p_sim.stepBackwardAndGetParticles())); //
        asset = Markov state
9      VectorXd payOffLoc = p_payOff(*asset); // pay off
10     // update conditional expectation operator for current Markov state
11     p_regressor.updateSimulations(((iStep == (p_sim.getNbStep() - 1)) ? true : false),
        asset);
12     // conditional expectation
13     VectorXd condEspec = p_regressor.getAllSimulations(Cash) * expRate;
14     // arbitrage between pay off and cash delivered after
15     Cash = (condEspec.array() < payOffLoc.array()).select(payOffLoc, Cash * expRate);
16 }
17 return Cash.mean();
```

American option with the Python API

Using the Python API the American resolution is given below:

```
1  # Copyright (C) 2016 EDF
2  # All Rights Reserved
3  # This code is published under the GNU Lesser General Public License (GNU LGPL)
4  import numpy as np
5  import math as maths
6
7  # american option by Longstaff-Schwartz
8  # p_sim          Monte Carlo simulator
9  # p_payOff       Option pay off
10 # p_regressor     regressor object
11 def resolution(p_simulator, p_payOff, p_regressor) :
12
13     step = p_simulator.getStep()
14     # asset simulated under the neutral risk probability : get the trend of first asset to
        get interest rate
15     expRate = np.exp(-step * p_simulator.getMu()[0])
16     # Terminal
17     particle = p_simulator.getParticles()
18     Cash = p_payOff.operator(particle)
19
20     for iStep in range(0, p_simulator.getNbStep()):
21         asset = p_simulator.stepBackwardAndGetParticles()
22         payOffLoc = p_payOff.operator(asset)
23         isLastStep = False
24         if iStep == p_simulator.getNbStep() - 1 :
25             isLastStep = True
26
27         p_regressor.updateSimulations(isLastStep, asset)
28         # conditional expectation
29         condEspec = p_regressor.getAllSimulations(Cash).squeeze() * expRate
30         # arbitrage
31         Cash = np.where(condEspec < payOffLoc, payOffLoc, Cash * expRate)
32
33     return maths.fsum(Cash) / len(Cash)
```

2.2 American options by tree

Using trees, American options are solved calculating the Bellman values at each date instead of valuing them as expectation of payoff at optimal stopping time.

Algorithm 9 Algorithm with tree: Bellman value (0 interest rate)

Initialization:

Set $P^j := g(X_T^j)$, $j \leq N(\kappa)$

▷ Number of node at last date

Backward induction:

for $i = \kappa - 1$ to 0 **do**

 set $P^j = \max[\mathbb{E}[P \mid X_{t_i}^j], g(X_{t_i}^j)$, $j \leq N(i)$

end for

Price estimator at 0: P^0 .

2.2.1 The American option by tree

```

1  // a backward simulator
2  MeanRevertingSimulatorTree< OneDimData<OneDimRegularSpaceGrid, double> > backSimulator1
   (binArxiv, futureGrid, sigma, mr);
3
4  // strike of put
5  double strike = 50.;
6
7  // actualization
8  double actu = exp(r * dates(dates.size() - 1));
9  // spot provided by simulator
10 ArrayXd spot = backSimulator1.getSpotValues() * actu;
11 // actualized value for payoff
12 ArrayXd val1 = (strike - spot).cwiseMax(0.) / actu;
13 for (int istep = 0; istep < nbDtStep; ++istep)
14 {
15     // one step backward to update probabilities and connectons between nodes
16     backSimulator1.stepBackward();
17     // probabilities
18     std::vector<double> proba = backSimulator1.getProba();
19     // get connection between nodes
20     std::vector< std::vector<std::array<int, 2> > > connected = backSimulator1.
        getConnected();
21     // conditional expectation operator
22     StOpt::Tree tree(proba, connected);
23     //interest rates
24     actu = exp(r * dates(dates.size() - 1 - (istep + 1) * nInc));
25     // spot : add interest rate
26     spot = backSimulator1.getSpotValues() * actu;
27     // pay off
28     ArrayXd payOff = (strike - spot).cwiseMax(0.) / actu;
29     //actualize value
30     val1 = tree.expCond(val1);
31     // arbitrage
32     val1 = (val1 > payOff).select(val1, payOff);
33 }
34
35 double finalValue = val1(0);

```

2.2.2 Python API

```

1      # backward simulator
2      backSimulator = Simulators.MeanRevertingSimulatorTree(archiveToRead, futureGrid,
3                                                             sigma, mr)
4
5      # strike
6      strike = 50.
7
8      # actu
9      actu = np.exp(r*dates[indexT[-1]])
10     # spot : add interest rate
11     spot = backSimulator.getSpotValues()*actu
12     # actualized payoff
13     val1= np.where( strike-spot>0,strike-spot,0)/actu
14     for istep in np.arange(np.shape(indexT)[0]-1):
15         # backward in simulator
16         backSimulator.stepBackward()
17         # get back probability matrix
18         proba = backSimulator.getProba()
19         # and connection matrix
20         connected = backSimulator.getConnected()
21         # creta tree for conditional expectation
22         tree=StOptTree.Tree(proba,connected)
23         # interest rates
24         actu = np.exp(r*dates[indexT[-2-istep]])
25         # spot : add interest rate
26         spot = backSimulator.getSpotValues()*actu
27         # pay off
28         payOff = np.where( strike-spot>0,strike-spot,0)/actu
29         # actualize value
30         val1 = tree.expCond(val1)
31         # arbitrage
32         val1 = np.where( val1> payOff, val1, payOff)
33
34     final = val1[0]

```

Chapter 3

Using the general framework to manage stock problems

In this chapter the state is separated into three parts $X^{x,t} = (X_1^{x,t}, X_2^{x,t}, I)$. $(X_1^{x,t}, X_2^{x,t})$, which corresponds to the special case of chapter 4 where $X_1^{x,t}$ is not controlled and $X_2^{x,t}$ is controlled. Two cases can be tackled:

- the first case corresponds to the case where $X_2^{x,t}$ is deterministic (think of storage management),
- the second case corresponds to the case where $X_2^{x,t}$ is stochastic (think of portfolio optimization).

I_t takes integers values and is here to describe some finite discrete regimes (to deal with some switching problems). A general framework is available to solve this type of problem. First of all, the second part $X_2^{x,t}$ is discretized on a grid as explained in chapter 4.

- Either a full grid is used for $X_2^{x,t}$ and two types of sequential or parallel resolutions be can considered:
 - a resolution can be obtained sequentially or a parallelization with MPI of the computations can be carried out (speed up but no size up). This approach can be used for small dimension problems.
 - a resolution can be obtained with a parallelization by the MPI framework by spreading the work to be done on the points of the grid, and by distributing the data between processors (speed up and size up). We will call this parallelization technique a “distribution” technique. This approach is necessary to tackle very large optimization problems where the overall solution cannot be stored in the memory of a single processor.
- or the grid for $X_2^{x,t}$ is not full (therefore sparse) and only a parallelization by thread and MPI can be carried out on the calculations (acceleration and no increase in size). With sparse grids, only the deterministic $X_2^{x,t}$ case is treated.

In the case of the MPI parallelization technique distributing task and the data (full grids only), [34] and [45] are used. Suppose that the grid is the same at each time step (only here to facilitate the case), and that we have 4 processors (figure 3.1) then:

- at the last time step, the final values at each point of each simulation are calculated (each processor calculates the values for its own grid points),
- in the previous time step, from a grid point, specific to a processor, we are able to locate the grid points reached in the next time step by all the commands,
- on figure 3.1, we give the points belonging to other processors which can be reached from points belonging to processor 3,
- certain MPI communications are carried out by bringing back the data (values calculated at the previously processed time step) necessary for processor 3 to be able to update the calculated value by dynamic programming at the moment for all the points belonging to processor 3,
- all communications between all processors are done together.

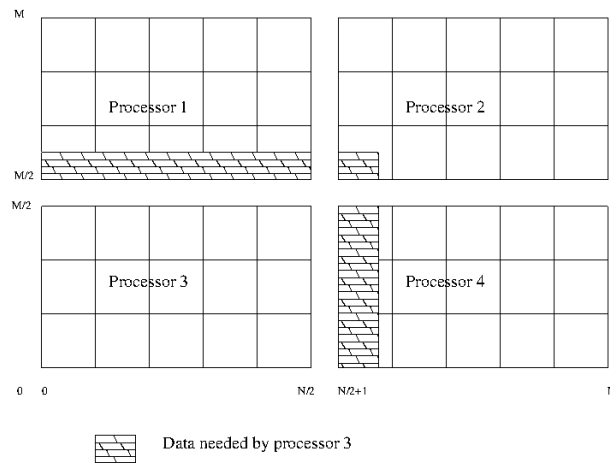


Figure 3.1: Data to send to processor 3

The global state of the problem is stored in the `StateWithStocks` object.

3.1 General requirement for the business object

In order to use the framework, the developer must describe the problem he wants to solve in one step from a state $X^{x,t}$. This business object must offer common methods and it is derived from `OptimizerBase`.

```

1 // Copyright (C) 2016 EDF
2 // All Rights Reserved
3 // This code is published under the GNU Lesser General Public License (GNU LGPL)
4 #ifndef OPTIMIZERBASE_H
5 #define OPTIMIZERBASE_H
6 #include <Eigen/Dense>
7
8 /** \file OptimizerBase.h
9  * \brief Define an abstract class for Dynamic Programming problems solved by Monte Carlo
10  * methods

```

```

10  *      \author Xavier Warin
11  */
12
13 namespace StOpt
14 {
15
16 /// \class OptimizerBase OptimizerBase.h
17 /// Base class for optimizer for Dynamic Programming with and without regression methods
18 class OptimizerBase
19 {
20
21
22 public :
23
24     OptimizerBase() {}
25
26     virtual ~OptimizerBase() {}
27
28     /// \brief defines the dimension to split for MPI parallelism
29     /// For each dimension return true is the direction can be split
30     virtual Eigen::Array< bool, Eigen::Dynamic, 1> getDimensionToSplit() const = 0 ;
31
32     /// \brief defines the diffusion cone for parallelism
33     /// \param p_regionByProcessor region (min max) treated by the processor for
34     /// the different regimes treated
35     /// \return returns in each dimension the min max values in the stock that can be
36     /// reached from the grid p_gridByProcessor for each regime
37     virtual std::vector< std::array< double, 2> > getCone(const std::vector< std::array<
38         double, 2> > &p_regionByProcessor) const = 0;
39
40
41     /// \brief Get the number of regimes allowed for the asset to be reached at the
42     /// current time step
43     virtual int getNbRegime() const = 0 ;
44
45     /// \brief get back the dimension of the control
46     virtual int getNbControl() const = 0 ;
47
48     /// \brief get size of the function to follow in simulation
49     virtual int getSimuFuncSize() const = 0;
50 };
51 #endif /* OPTIMIZERBASE_H */

```

We detail all the methods to be implemented for all the resolution methods (with or without regressions).

- the `getNbRegime` makes it possible to obtain the number of regimes of the problem: for example, in switching problems, when there is a switching cost, the working regime has to be incorporated in the state. Another example is the case of the conditional delta to calculate for an asset: two regimes can be used: one to calculate the value of the asset and the second to calculate the Δ . This number of regimes can depend on time: in this case for a current resolution date t the `getNbRegime` method sends the number of regimes at the very beginning of the time step (in t^-) such such that a switch to a new regime can take place in t^+ .
- the `getSimulator` method is used to retrieve the simulator giving the Monte Carlo simulations,
- the `getSimuFuncSize` method is used in simulation to define the number of functions

to follow in the simulation part. For example in a stochastic target problem where the target is a given wealth with a given probability, one may want to follow the evolution of the probability at each time step and of the wealth obtained in trading. In this case the `getSimuFuncSize` returns 2.

- the `getCone` method is only relevant if the MPI framework with distribution is used. As argument it takes a vector of size the dimension of the grid. Each component of the vector is an array containing the minimum and maximum coordinate values of the points of the current grid defining an hyper cube $H1$. It returns for each dimension, the min and max coordinates of the hyper cube $H2$ containing the points which can be reached by applying a command from a point of the grid in $H1$.
- the `getDimensionToSplit` method is used to define in the MPI framework with distribution the directions to be divided for the solution on the processors. For each dimension, it returns a Boolean where `true` means that the direction is a candidate for splitting.
- the `stepSimulateControl` method is used after the optimization using the optimal controls calculated in the optimization part. From a state `p_state` (storing the $X^{x,t}$), the calculated optimal control in optimization `p_control`, the values of optimal functions along the current path are stored in `p_phiInOut`. The state `p_state` is updated at the end of the function call.

In the first part, we present the framework of the problems where the conditional expectation is calculated by regression (case where $X_2^{t,x}$ is not controlled). Next, we develop the framework that does not use regression for the conditional expectation calculations. All conditional expectations are calculated using exogenous particles and interpolation. This will generally be the case for portfolio optimization.

3.2 Solving the problem using conditional expectation calculated by regressions

In this part, we assume that $X_2^{x,t}$ is controlled and deterministic so that regression methods can be used.

3.2.1 Requirement to use the framework

The `OptimizerBaseInterp` is an optimizer class common to all regression methods used by dynamic programming for stocks problems. We detail the methods of `OptimizerBaseInterp` which is a derived from `OptimizerBase`. Only one method is added:

- the `stepSimulateControl` method is used after optimization using the optimal controls calculated in the optimization part. From a state `p_state` (storing the $X^{x,t}$), the calculated optimal control in the `p_control` optimization, the values of the optimal functions along the current path are stored in `p_phiInOut`. The state `p_state` is updated at the end of the call function.

3.2.2 Classical regression

By classical regression, we mean regression problems with storages where the optimal command is calculated on one time step and estimated by testing all possible discretized commands.

In order to use the framework with regression for conditional expectation, a business object describing the business at a time step from a state is derived from `OptimizerDPBase` itself derived from `OptimizerBaseInterp`.

```
1 // Copyright (C) 2016 EDF
2 // All Rights Reserved
3 // This code is published under the GNU Lesser General Public License (GNU LGPL)
4 #ifndef OPTIMIZERDPBASE_H
5 #define OPTIMIZERDPBASE_H
6 #include <Eigen/Dense>
7 #include "StOpt/core/utils/StateWithStocks.h"
8 #include "StOpt/core/grids/SpaceGrid.h"
9 #include "StOpt/core/grids/FullGrid.h"
10 #include "StOpt/regression/BaseRegression.h"
11 #include "StOpt/regression/ContinuationValue.h"
12 #include "StOpt/regression/GridAndRegressedValue.h"
13 #include "StOpt/regression/GridAndRegressedValueAutoDiff.h"
14 #include "StOpt/dp/SimulatorDPBase.h"
15 #include "StOpt/dp/OptimizerBaseInterp.h"
16
17 /** \file OptimizerDPBase.h
18  * \brief Define an abstract class for Dynamic Programming problems solved by regression
19  * \author Xavier Warin
20  */
21
22 namespace StOpt
23 {
24
25     /// \class OptimizerDPBase OptimizerDPBase.h
26     /// Base class for optimizer for Dynamic Programming with regression methods
27     class OptimizerDPBase : public OptimizerBaseInterp
28     {
29
30     public :
31
32         OptimizerDPBase() {}
33
34         virtual ~OptimizerDPBase() {}
35
36         /// \brief defines the diffusion cone for parallelism
37         /// \param p_regionByProcessor region (min max) treated by the processor for
38         /// the different regimes treated
39         /// \return returns in each dimension the min max values in the stock that can be
40         /// reached from the grid p_gridByProcessor for each regime
41         virtual std::vector< std::array< double, 2> > getCone(const std::vector< std::array<
42             double, 2> > &p_regionByProcessor) const = 0;
43
44         /// \brief defines the dimension to split for MPI parallelism
45         /// For each dimension return true is the direction can be split
46         virtual Eigen::Array< bool, Eigen::Dynamic, 1> getDimensionToSplit() const = 0 ;
47
48         /// \brief defines a step in optimization
49         /// \param p_grid grid at arrival step after command
50         /// \param p_stock coordinates of the stock point to treat
51         /// \param p_condEsp continuation values for each regime
52         /// \param p_phiIn for each regime gives the solution calculated at the previous
53         /// step ( next time step by Dynamic Programming resolution) : structure of the 2D
54         /// array ( nb simulation ,nb stocks )
55         /// \return a pair :
```

```

52     /// - for each regimes (column) gives the solution for each particle (row)
53     /// - for each control (column) gives the optimal control for each
        particle (rows)
54     /// .
55     virtual std::pair< Eigen::ArrayXXd, Eigen::ArrayXXd> stepOptimize(const std::
        shared_ptr< StOpt::SpaceGrid> &p_grid, const Eigen::ArrayXd &p_stock,
56         const std::vector< StOpt::ContinuationValue > &p_condEsp,
57         const std::vector< std::shared_ptr< Eigen::ArrayXXd > > &p_phiIn) const = 0;
58
59
60     /// \brief defines a step in simulation
61     /// Notice that this implementation is not optimal but is convenient if the control is
        discrete.
62     /// By avoiding interpolation in control we avoid non admissible control
63     /// Control are recalculated during simulation.
64     /// \param p_grid grid at arrival step after command
65     /// \param p_continuation defines the continuation operator for each regime
66     /// \param p_state defines the state value (modified)
67     /// \param p_phiInOut defines the value functions (modified) : size number of
        functions to follow
68     virtual void stepSimulate(const std::shared_ptr< StOpt::SpaceGrid> &p_grid, const std
        ::vector< StOpt::GridAndRegressedValue > &p_continuation,
69         StOpt::StateWithStocks<double> &p_state,
70         Eigen::Ref<Eigen::ArrayXd> p_phiInOut) const = 0 ;
71
72
73     /// \brief Defines a step in simulation using interpolation in controls
74     /// \param p_grid grid at arrival step after command
75     /// \param p_control defines the controls
76     /// \param p_state defines the state value (modified)
77     /// \param p_phiInOut defines the value function (modified): size number of
        functions to follow
78     virtual void stepSimulateControl(const std::shared_ptr< StOpt::SpaceGrid> &p_grid,
        const std::vector< StOpt::GridAndRegressedValue > &p_control,
79         StOpt::StateWithStocks<double> &p_state,
80         Eigen::Ref<Eigen::ArrayXd> p_phiInOut) const = 0 ;
81
82
83     /// \brief Defines a step in simulation with auto differentiation
84     /// this method should be implemented in clinets class discribing the problem to
        solve
85     /// template< typename T>
86     /// stepSimulateControlAutoDiff(const std::shared_ptr< StOpt::FullGrid> &p_grid,
        const std::vector< StOpt::GridAndRegressedValueAutoDiff<T> > &p_control,
87         StOpt::StateWithStocks<T> &p_state,
88         Eigen::Ref<Eigen::Array<T,Eigen::Dynamic,1>
        p_phiInOut)
89
90
91
92
93     /// \brief Get the number of regimes allowed for the asset to be reached at the
        current time step
94     /// If \f$ t \f$ is the current time, and \f$ dt \f$ the resolution step, this is
        the number of regime allowed on \f$[ t- dt, t[\f$
95     virtual int getNbRegime() const = 0 ;
96
97     /// \brief get the simulator back
98     virtual std::shared_ptr< StOpt::SimulatorDPBase > getSimulator() const = 0;
99
100    /// \brief get back the dimension of the control
101    virtual int getNbControl() const = 0 ;
102
103    /// \brief get size of the function to follow in simulation
104    virtual int getSimuFuncSize() const = 0;
105
106 };
107 }
108 #endif /* OPTIMIZERDPBASE_H */

```

We detail the different methods derived from `OptimizerDPBase` to implement in addition to the `OptimizerBaseInterp` methods:

- the `stepOptimize` method is used in optimization. We want to calculate the optimal value at the current t_i at a grid point `p_stock` using a `p_grid` grid at the following date t_{i+1} , the continuation values for all the `p_condEsp` regimes allowing to calculate the conditional expectation of the optimal value function calculated in the previous one treated time step t_{i+1} . From a grid point `p_stock` it calculates the function values and the optimal controls. It returns a pair where the
 - the first element is a matrix (first dimension is the number of simulations, the second dimension the number of regimes) giving the value of the function,
 - the second element is a matrix (the first dimension is the number of simulations, the second dimension the number of controls) giving the optimal control.
- the `stepSimulate` method is used after the optimization using the continuation values calculated in the optimization part. From a `p_state` state (storing the $X^{x,t}$), the calculated continuation values in optimization `p_continuation`, the optimal functions values along the current path are stored in `p_phiInOut`.

In the case of [47] gas storage, the storage holder can inject gas from the grid into the storage (by paying the market price) or withdraw gas from the storage on the grid (by receiving the market price). In this case the `Optimize` object is given in the file `OptimizeGasStorage.h`. You can take a look at the implementation of the `getCone` method.

The framework in optimization with classical regressions

Once an `Optimizer` is derived for the project, and assuming a full grid is used for inventory discretization, the framework provides a `TransitionStepRegressionDPDist` object in MPI which allows to solve the optimization problem with data distribution over a time step with the following constructor:

```

1  TransitionStepRegressionDPDist(const shared_ptr<FullGrid> &p_pGridCurrent,
2                                const shared_ptr<FullGrid> &p_pGridPrevious,
3                                const shared_ptr<OptimizerDPBase> &p_pOptimize,
4                                const boost::mpi::communicator &p_world)

```

with

- `p_pGridCurrent` is the grid at the current time step (t_i),
- `p_pGridPrevious` is the grid at the previously processed time step (t_{i+1}),
- `p_pOptimize` the optimizer object
- `p_world` The MPI communicator

Remark 6 A similar object is available without the MPI distribution framework *TransitionStepRegressionDP* with always the activation of parallelization with threads and MPI on calculations on the full points grid.

Remark 7 *In the case of sparse grids with only parallelization on the calculations (threads and MPI) `TransitionStepRegressionDPSparse` object can be used.*

The main method is

```
1  std::vector< shared_ptr< Eigen::ArrayXXd > > OneStep(const std::vector< shared_ptr<
    Eigen::ArrayXXd > > &p_phiIn,
2      const shared_ptr< BaseRegression> &p_condExp)
```

with

- `p_phiIn` the vector (its size corresponds to the number of regimes) of the matrix of optimal values calculated at the previous time iteration for each regime. Each matrix is a number of simulations per matrix of number of stock points matrix.
- `p_condExp` the conditional expectation operator,

returning a pair:

- The first element is a vector of matrix with new optimal values at the current time step (each element of the vector corresponds to a regime and each matrix is a number of simulations per matrix of number of stock points).
- The second element is vector of matrix with new optimal controls at the current time step (each element of the vector corresponds to a control and each matrix is a number of simulations per number of stock points matrix).

Remark 8 *All `TransitionStepRegressionDP` derive from a `TransitionStepRegressionBase` object having a pure virtual `OneStep` method.*

A second method is provided permitting to dump the continuation values of the problem and the optimal control at each time step:

```
1  void dumpContinuationValues(std::shared_ptr<gs::BinaryFileArchive> p_ar , const std:::
    string &p_name, const int &p_iStep,
2      const std::vector< std::shared_ptr< Eigen::ArrayXXd > > &
    p_phiInPrev,
3      const std::vector< std::shared_ptr< Eigen::ArrayXXd > > &
    p_control,
4      const std::shared_ptr<BaseRegression> &p_condExp,
5      const bool &p_bOneFile) const
```

with:

- `p_ar` is the archive where the controls and solutions are dumped,
- `p_name` is a base name used in the archive to store the solution and the control,
- `p_phiInPrev` is the previous time step solution used to calculate the continuation values that are stored,
- `p_control` stores the optimal controls calculated at the current time step,
- `p_condExp` is the conditional expectation object allowing to calculate the conditional expectation of the functions defined at the previous time step processed `p_phiInPrev` and allowing to store a representation of the optimal control.

- `p_bOneFile` is set if the continuation and optimal controls calculated by each processor are flushed to a single file. Otherwise, the continuation and optimal controls calculated by each processor are flushed to different files (one by processor). If the problem results in optimal continuation and control values on the global grid that can be stored in the memory of the compute node, it may be more interesting to dump the continuation/control values to a file for policy simulation optimal.

Remark 9 *As for the `TransitionStepRegressionDP` object and the `TransitionStepRegressionDPSparse` object, their `dumpContinuationValues` does not need an argument `p_bOneFile`: the optimal controls are obviously stored in a single file.*

Here we give a simple example of temporal resolution using this method when MPI data distribution is used

```

1 // Copyright (C) 2016 EDF
2 // All Rights Reserved
3 // This code is published under the GNU Lesser General Public License (GNU LGPL)
4 #ifdef USE_MPI
5 #include <fstream>
6 #include <memory>
7 #include <functional>
8 #include <boost/lexical_cast.hpp>
9 #include <boost/mpi.hpp>
10 #include <Eigen/Dense>
11 #include "generators/BinaryFileArchive.hh"
12 #include "StOpt/core/grids/FullGrid.h"
13 #include "StOpt/regression/BaseRegression.h"
14 #include "StOpt/dp/FinalStepDPDist.h"
15 #include "StOpt/dp/TransitionStepRegressionDPDist.h"
16 #include "StOpt/core/parallelism/reconstructProcOMpi.h"
17 #include "StOpt/dp/OptimizerDPBase.h"
18 #include "StOpt/dp/SimulatorDPBase.h"
19
20
21 using namespace std;
22
23 double DynamicProgrammingByRegressionDist(const shared_ptr<StOpt::FullGrid> &p_grid,
24     const shared_ptr<StOpt::OptimizerDPBase> &p_optimize,
25     shared_ptr<StOpt::BaseRegression> &p_regressor,
26     const function<double(const int &, const Eigen::ArrayXd &, const Eigen::ArrayXd &)>
27         &p_funcFinalValue,
28     const Eigen::ArrayXd &p_pointStock,
29     const int &p_initialRegime,
30     const string &p_fileToDump,
31     const bool &p_bOneFile,
32     const boost::mpi::communicator &p_world)
33 {
34     // from the optimizer get back the simulator
35     shared_ptr<StOpt::SimulatorDPBase> simulator = p_optimize->getSimulator();
36     // final values
37     vector< shared_ptr< Eigen::ArrayXXd > > valuesNext = StOpt::FinalStepDPDist(p_grid,
38         p_optimize->getNbRegime(), p_optimize->getDimensionToSplit(), p_world)(
39         p_funcFinalValue, simulator->getParticles().array());
40     // dump
41     string toDump = p_fileToDump ;
42     // test if one file generated
43     if (!p_bOneFile)
44         toDump += "_" + boost::lexical_cast<string>(p_world.rank());
45     shared_ptr<gs::BinaryFileArchive> ar;
46     if ((!p_bOneFile) || (p_world.rank() == 0))
47         ar = make_shared<gs::BinaryFileArchive>(toDump.c_str(), "w");
48     // name for object in archive
49     string nameAr = "Continuation";

```

```

47   for (int iStep = 0; iStep < simulator->getNbStep(); ++iStep)
48   {
49       Eigen::ArrayXXd asset = simulator->stepBackwardAndGetParticles();
50       // conditional expectation operator
51       p_regressor->updateSimulations(((iStep == (simulator->getNbStep() - 1)) ? true :
52                                     false), asset);
53       // transition object
54       StOpt::TransitionStepRegressionDPDist transStep(p_grid, p_grid, p_optimize, p_world
55       );
56       pair< vector< shared_ptr< Eigen::ArrayXXd > >, vector< shared_ptr< Eigen::ArrayXXd
57               > > > valuesAndControl = transStep.oneStep(valuesNext, p_regressor);
58       transStep.dumpContinuationValues(ar, nameAr, iStep, valuesNext, valuesAndControl.
59               second, p_regressor, p_bOneFile);
60       valuesNext = valuesAndControl.first;
61   }
62   // reconstruct a small grid for interpolation
63   return StOpt::reconstructProcOMpi(p_pointStock, p_grid, valuesNext[p_initialRegime],
64       p_optimize->getDimensionToSplit(), p_world).mean();
65 }
66 #endif

```

An example without data distribution can be found in the file `DynamicProgrammingByRegression.cpp`. We finally give an array with the different `TransitionStepRegression` objects to use depending on the type of parallelization used.

Table 3.1: Which object `TransitionStepRegression` to use depending on the grid used and the type of parallelization used.

	Full grid	Sparse grid
Sequential	<code>TransitionStepRegressionDP</code>	<code>TransitionStepRegressionDPSparse</code>
Parallelization on calculations threads and MPI	<code>TransitionStepRegressionDP</code>	<code>TransitionStepRegressionDPSparse</code>
Distribution of calculations and data	<code>TransitionStepRegressionDPDist</code>	Not available

The framework in simulation with classical regressions

Once the optimization is done, the continuation values are saved in a file (or in some files) at each time step. In order to simulate the optimal policy, we can use the previously calculated continuation values (see chapter 4) or we can use the optimal controls calculated in optimization. In continuous optimization, the use of control is more efficient in terms of computational cost. When the control is discrete, the interpolation of the controls can lead to inadmissible controls and the simulation with the value function is more precise.

When simulating optimal control, two cases can occur:

- In most cases (small dimension case), the optimal control or optimal function value can be stored in the computer's node memory and the function values and controls are stored in a single file. In this case, an optimum control simulation can be easily performed by distributing the Monte Carlo simulations on the available compute nodes: this can be done using the `SimulateStepRegression` or `SimulateStepRegressionControl` objects at each time step of the simulation.
- When it comes to very large problems, optimization is achieved by distributing the data across the processors and it is impossible to store the optimal command on

a node. In this case, the optimal controls and the optimal solutions are stored in the memory of the node which was used to calculate them in optimization. The simulations are reorganized at each time step and gathered so as to occupy the same part of the global grid. Each processor will then get from the other processors a version of the optimal control or solution it needs. This methodology is used in the `SimulateStepRegressionDist` and `SimulateStepRegressionControlDist` objects.

We detail the simulations objects using the optimal function value calculated in optimization and the optimal control for the case of very large problems.

- Simulation step using the value function calculated in optimization:

In order to simulate one step of the optimal policy, an object `SimulateStepRegressionDist` is provided with constructor

```
1 SimulateStepRegressionDist(gs::BinaryFileArchive &p_ar,  const int &p_iStep,  const
   std::string &p_nameCont,
2                               const shared_ptr<FullGrid> &p_pGridFollowing,  const
   shared_ptr<OptimizerDPBase > &p_pOptimize,
3                               const bool &p_bOneFile,  const boost::mpi::communicator &
   p_world )
```

where

- `p_ar` is the binary archive where the continuation values are stored,
- `p_iStep` is the number associated with the current time step (0 at the start date of the simulation, the number is increased by one at each simulated time step),
- `p_nameCont` is the base name for continuation values,
- `p_pGridFollowing` is the grid at the next time step (`p_iStep+1`),
- `p_pOptimize` the Optimizer describing the passage from one time step to the next,
- `p_bOneFile` equal to `true` if only one archive is used to store continuation values.
- `p_world` The MPI communicator

Remark 10 *A version without data distribution but with multithreaded and with possible MPI on calculations is available with the object `SimulateStepRegression`. The `p_OneFile` argument is omitted during construction.*

This object implements the method `oneStep`

```
1 void oneStep(std::vector<StateWithStocks<double> > &p_statevector , Eigen::ArrayXXd &
   p_phiInOut)
```

where:

- `p_statevector` stores the states of all simulations: this state is updated by applying the optimal command,

- `p_phiInOut` stores the gain/cost functions for all the simulations: it is updated by the function call. The size of the array is $(nb, nbSimul)$ where nb is given by the `getSimuFuncSize` method of the optimizer and $nbSimul$ the number of Monte Carlo simulations.

An example of using this method to simulate an optimal policy with distribution is given below:

```

1 // Copyright (C) 2016 EDF
2 // All Rights Reserved
3 // This code is published under the GNU Lesser General Public License (GNU LGPL)
4 #ifndef SIMULATEREGRESSIONDIST_H
5 #define SIMULATEREGRESSIONDIST_H
6 #include <functional>
7 #include <memory>
8 #include <Eigen/Dense>
9 #include <boost/mpi.hpp>
10 #include "generators/BinaryFileArchive.hh"
11 #include "StOpt/core/grids/FullGrid.h"
12 #include "StOpt/core/utils/StateWithStocks.h"
13 #include "StOpt/dp/SimulateStepRegressionDist.h"
14 #include "StOpt/dp/OptimizerDPBase.h"
15 #include "StOpt/dp/SimulatorDPBase.h"
16
17
18 /** \file SimulateRegressionDist.h
19  * \brief Defines a simple program showing how to use simulation
20  *      A simple grid is used
21  * \author Xavier Warin
22  */
23
24
25 /// \brief Simulate the optimal strategy , mpi version
26 /// \param p_grid          grid used for deterministic state (stocks for
27   example)
28 /// \param p_optimize      optimizer defining the optimization between two
29   time steps
30 /// \param p_funcFinalValue function defining the final value
31 /// \param p_pointStock    initial point stock
32 /// \param p_initialRegime regime at initial date
33 /// \param p_fileToDump    name associated to dumped bellman values
34 /// \param p_bOneFile      do we store continuation values in only one file
35 /// \param p_world         MPI communicator
36 double SimulateRegressionDist(const std::shared_ptr<StOpt::FullGrid> &p_grid,
37                               const std::shared_ptr<StOpt::OptimizerDPBase > &
38                               p_optimize,
39                               const std::function<double(const int &, const Eigen::
40                               ArrayXd &, const Eigen::ArrayXd &)> &
41                               p_funcFinalValue,
42                               const Eigen::ArrayXd &p_pointStock,
43                               const int &p_initialRegime,
44                               const std::string &p_fileToDump,
45                               const bool &p_bOneFile,
46                               const boost::mpi::communicator &p_world)
47 {
48     // from the optimizer get back the simulator
49     std::shared_ptr< StOpt::SimulatorDPBase> simulator = p_optimize->getSimulator();
50     int nbStep = simulator->getNbStep();
51     std::vector< StOpt::StateWithStocks<double>> states;
52     states.reserve(simulator->getNbSimul());
53     for (int is = 0; is < simulator->getNbSimul(); ++is)
54         states.push_back(StOpt::StateWithStocks<double>(p_initialRegime, p_pointStock,
55                                                         Eigen::ArrayXd::Zero(simulator->getDimension())));
56     std::string toDump = p_fileToDump ;
57     // test if one file generated
58     if (!p_bOneFile)

```

```

53     toDump += " " + boost::lexical_cast<std::string>(p_world.rank());
54     gs::BinaryFileArchive ar(toDump.c_str(), "r");
55     // name for continuation object in archive
56     std::string nameAr = "Continuation";
57     // cost function
58     Eigen::ArrayXXd costFunction = Eigen::ArrayXXd::Zero(p_optimize->getSimuFuncSize()
59     , simulator->getNbSimul());
60     for (int istep = 0; istep < nbStep; ++istep)
61     {
62         StOpt::SimulateStepRegressionDist(ar, nbStep - 1 - istep, nameAr, p_grid,
63         p_optimize, p_bOneFile, p_world).oneStep(states, costFunction);
64
65         // new stochastic state
66         Eigen::ArrayXXd particles = simulator->stepForwardAndGetParticles();
67         for (int is = 0; is < simulator->getNbSimul(); ++is)
68             states[is].setStochasticRealization(particles.col(is));
69     }
70     // final : accept to exercise if not already done entirely (here suppose one
71     // function to follow)
72     for (int is = 0; is < simulator->getNbSimul(); ++is)
73         costFunction(0, is) += p_funcFinalValue(states[is].getRegime(), states[is].
74         getPtStock(), states[is].getStochasticRealization()) * simulator->getActu
75         ();
76     return costFunction.mean();
77 }
78 #endif /* SIMULATEREGRESSIONDIST_H */

```

The version of the previous example using a single archive storing the control/solution is given in the `SimulateRegression.h` file.

- Simulation step using the optimal controls calculated in optimization:

```

1     SimulateStepRegressionControlDist(gs::BinaryFileArchive &p_ar, const int &p_iStep
2     , const std::string &p_nameCont,
3     const std::shared_ptr<FullGrid> &p_pGridCurrent
4     , const std::shared_ptr<FullGrid> &
5     p_pGridFollowing,
6     const std::shared_ptr<OptimizerDPBase> &
7     p_pOptimize,
8     const bool &p_bOneFile,
9     const boost::mpi::communicator &p_world);

```

where

- `p_ar` is the binary archive where the continuation values are stored,
- `p_iStep` is the number associated with the current time step (0 at the start date of simulation, the number is increased by one at each simulated time step),
- `p_nameCont` is the base name of the control values,
- `p_pGridCurrent` is the grid at the current time step (`p_iStep`),
- `p_pGridFollowing` is the grid at the next time step (`p_iStep+1`),
- `p_pOptimize` is the Optimizer describing the passage from one time step to the next,
- `p_bOneFile` is equal to `true` if only one archive is used to store continuation values.

– p_world The MPI communicator

Remark 11 *A version in which a single archive storing the control/solution is used is available with the object `SimulateStepRegressionControl`*

This object implements the method `oneStep`

```
1 void oneStep(std::vector<StateWithStocks<double> > &p_statevector , Eigen::ArrayXd &
    p_phiInOut)
```

where:

- `p_statevector` stores the state for all simulations: this state is updated by applying optimal commands,
- `p_phiInOut` stores the gain/cost functions for all simulations: it is updated by the function call. The size of the array is $(nb, nbSimul)$ where nb is given by the `getSimuFuncSize` method of the optimizer and $nbSimul$ the number of Monte Carlo simulations.

An example of using this method to simulate an optimal policy with distribution is given below:

```
1 // Copyright (C) 2016 EDF
2 // All Rights Reserved
3 // This code is published under the GNU Lesser General Public License (GNU LGPL)
4 #ifndef USE_MPI
5 #ifndef SIMULATEREGRESSIONCONTROLDIST_H
6 #define SIMULATEREGRESSIONCONTROLDIST_H
7 #include <functional>
8 #include <memory>
9 #include <Eigen/Dense>
10 #include <boost/mpi.hpp>
11 #include "generators/BinaryFileArchive.hh"
12 #include "StOpt/core/grids/FullGrid.h"
13 #include "StOpt/core/utils/StateWithStocks.h"
14 #include "StOpt/dp/SimulateStepRegressionControlDist.h"
15 #include "StOpt/dp/OptimizerDPBase.h"
16 #include "StOpt/dp/SimulatorDPBase.h"
17
18
19 /** \file SimulateRegressionControlDist.h
20 * \brief Defines a simple program showing how to use simulation
21 * A simple grid is used
22 * \author Xavier Warin
23 */
24
25
26 /// \brief Simulate the optimal strategy using optimal controls calculated in
27 /// optimization , mpi version
28 /// \param p_grid grid used for deterministic state (stocks for
29 /// example)
30 /// \param p_optimize optimizer defining the optimization between two
31 /// time steps
32 /// \param p_funcFinalValue function defining the final value
33 /// \param p_pointStock initial point stock
34 /// \param p_initialRegime regime at initial date
35 /// \param p_fileToDump name associated to dumped bellman values
36 /// \param p_bOneFile do we store continuation values in only one file
37 /// \param p_world MPI communicator
38 double SimulateRegressionControlDist(const std::shared_ptr<StOpt::FullGrid> &p_grid,
```

```

36         const std::shared_ptr<StOpt::OptimizerDPBase > &
37             p_optimize,
38         const std::function<double(const int &, const
39             Eigen::ArrayXd &, const Eigen::ArrayXd &)> &
40             p_funcFinalValue,
41         const Eigen::ArrayXd &p_pointStock,
42         const int &p_initialRegime,
43         const std::string &p_fileToDump,
44         const bool &p_bOneFile,
45         const boost::mpi::communicator &p_world)
46 {
47     // from the optimizer get back the simulator
48     std::shared_ptr< StOpt::SimulatorDPBase> simulator = p_optimize->getSimulator();
49     int nbStep = simulator->getNbStep();
50     std::vector< StOpt::StateWithStocks<double>> states;
51     states.reserve(simulator->getNbSimul());
52     for (int is = 0; is < simulator->getNbSimul(); ++is)
53         states.push_back(StOpt::StateWithStocks<double>(p_initialRegime, p_pointStock,
54             Eigen::ArrayXd::Zero(simulator->getDimension())));
55     std::string toDump = p_fileToDump ;
56     // test if one file generated
57     if (!p_bOneFile)
58         toDump += "_" + boost::lexical_cast<std::string>(p_world.rank());
59     gs::BinaryFileArchive ar(toDump.c_str(), "r");
60     // name for continuation object in archive
61     std::string nameAr = "Continuation";
62     // cost function
63     Eigen::ArrayXXd costFunction = Eigen::ArrayXXd::Zero(p_optimize->getSimuFuncSize()
64         , simulator->getNbSimul());
65     for (int istep = 0; istep < nbStep; ++istep)
66     {
67         StOpt::SimulateStepRegressionControlDist(ar, nbStep - 1 - istep, nameAr,
68             p_grid, p_grid, p_optimize, p_bOneFile, p_world).oneStep(states,
69             costFunction);
70
71         // new stochastic state
72         Eigen::ArrayXXd particules = simulator->stepForwardAndGetParticles();
73         for (int is = 0; is < simulator->getNbSimul(); ++is)
74             states[is].setStochasticRealization(particules.col(is));
75     }
76     // final : accept to exercise if not already done entirely (here suppose one
77     // function to follow)
78     for (int is = 0; is < simulator->getNbSimul(); ++is)
79         costFunction(0, is) += p_funcFinalValue(states[is].getRegime(), states[is].
80             getPtStock(), states[is].getStochasticRealization()) * simulator->getActu
81             ();
82
83     return costFunction.mean();
84 }
85 #endif /* SIMULATEREGRESSIONCONTROLDIST_H */
86 #endif

```

The version of the previous example using a single archive storing the control/solution is given in the `SimulateRegressionControl.h` file.

In the table below, we indicate which simulation object should be used at each time step depending on the `TransitionStepRegressionDP` object used in optimization.

Table 3.2: Which simulation object to use depending on the TransitionStepRegression object used.

	TransitionStepRegressionDP	TransitionStepRegressionDPDist bOneFile = true	TransitionStepRegressionDPDist bOneFile = false	TransitionStepRegressionDPSparse
SimulateStepRegression	Yes	Yes	No	Yes
SimulateStepRegressionControl	Yes	Yes	No	Yes
SimulateStepRegressionDist	No	Yes	Yes	No
SimulateStepRegressionControlDist	No	Yes	Yes	No

3.2.3 Regressions and cuts for continuous linear transition problems with certain characteristics of concavity and convexity

During the optimization for example a storage, one can want to solve the transition problem on certain time steps by supposing that the uncertainties are known. The Bellman values for a given uncertainty are then concave compared to the storage when one tries to maximize certain gains for example. When the problem is continuous linear, we can use bender cuts to approximate the Bellman value with respect to the storage level as in the SDDP method 1. To use this cuts a business object using an LP solver must be created. This business object is derived from `OptimizerDPCutBase` itself derived from `OptimizerBaseInterp`.

```
1 // Copyright (C) 2019 EDF
2 // All Rights Reserved
3 // This code is published under the GNU Lesser General Public License (GNU LGPL)
4 #ifndef OPTIMIZERDPCUTBASE_H
5 #define OPTIMIZERDPCUTBASE_H
6 #include <Eigen/Dense>
7 #include "StOpt/core/utils/StateWithStocks.h"
8 #include "StOpt/core/grids/SpaceGrid.h"
9 #include "StOpt/regression/BaseRegression.h"
10 #include "StOpt/regression/ContinuationCuts.h"
11 #include "StOpt/dp/SimulatorDPBase.h"
12 #include "StOpt/dp/OptimizerBase.h"
13
14 /** \file OptimizerDPCutBase.h
15  * \brief Define an abstract class for Dynamic Programming problems solved by regression
16  *        methods using cust to approximate
17  *        Bellman values
18  *        \author Xavier Warin
19  */
20 namespace StOpt
21 {
22
23     /// \class OptimizerDPCutBase OptimizerDPCutBase.h
24     /// Base class for optimizer for Dynamic Programming with regression methods and cuts, so
25     /// using LP to solve transitional problems
26     class OptimizerDPCutBase : public OptimizerBase
27     {
28
29     public :
30
31         OptimizerDPCutBase() {}
32
33         virtual ~OptimizerDPCutBase() {}
34
35         /// \brief defines the diffusion cone for parallelism
36         /// \param p_regionByProcessor region (min max) treated by the processor for
37         /// the different regimes treated
38         /// \return returns in each dimension the min max values in the stock that can be
39         /// reached from the grid p_gridByProcessor for each regime
40         virtual std::vector< std::array< double, 2> > getCone(const std::vector< std::array<
41             double, 2> > &p_regionByProcessor) const = 0;
42
43         /// \brief defines the dimension to split for MPI parallelism
44         /// For each dimension return true is the direction can be split
45         virtual Eigen::Array< bool, Eigen::Dynamic, 1> getDimensionToSplit() const = 0 ;
46
47         /// \brief defines a step in optimization
48         /// \param p_grid grid at arrival step after command
49         /// \param p_stock coordinates of the stock point to treat
50         /// \param p_condEsp continuation values for each regime
51         /// \return For each regimes (column) gives the solution for each particle , and cut
52         /// (row)
```

```

49  ///      For a given simulation , cuts components (C) at a point stock \f$ \bar S
50  \f$ are given such that the cut is given by
51  ///      \f$ C[0] + \sum_{i=1}^d C[i] (S_i - \bar S_i) \f$
52  virtual Eigen::ArrayXXd stepOptimize(const std::shared_ptr< StOpt::SpaceGrid> &
53  p_grid, const Eigen::ArrayXd &p_stock,
54  const std::vector< StOpt::ContinuationCuts > &
55  p_condEsp) const = 0;
56
57  /// \brief defines a step in simulation
58  /// Control are recalculated during simulation using a local optimization using the LP
59  /// \param p_grid grid at arrival step after command
60  /// \param p_continuation defines the continuation operator for each regime
61  /// \param p_state defines the state value (modified)
62  /// \param p_phiInOut defines the value functions (modified) : size number of
63  functions to follow
64  virtual void stepSimulate(const std::shared_ptr< StOpt::SpaceGrid> &p_grid, const std
65  ::vector< StOpt::ContinuationCuts > &p_continuation,
66  StOpt::StateWithStocks<double> &p_state,
67  Eigen::Ref<Eigen::ArrayXd> p_phiInOut) const = 0 ;
68
69  /// \brief Get the number of regimes allowed for the asset to be reached at the
70  current time step
71  /// If \f$ t \f$ is the current time, and \f$ dt \f$ the resolution step, this is
72  the number of regime allowed on \f$[ t- dt, t[\f$
73  virtual int getNbRegime() const = 0 ;
74
75  /// \brief get the simulator back
76  virtual std::shared_ptr< StOpt::SimulatorDPBase > getSimulator() const = 0;
77
78  /// \brief get back the dimension of the control
79  virtual int getNbControl() const = 0 ;
80
81  /// \brief get size of the function to follow in simulation
82  virtual int getSimuFuncSize() const = 0;
83
84 };
85 }
86 #endif /* OPTIMIZERDPCUTBASE_H */

```

We detail the different methods to implement in addition to the `OptimizerBaseInterp` methods:

- the `stepOptimize` method is used in optimization. We want to calculate the optimal value and the corresponding sensibilities with respect to the stocks at current t_i at a grid point `p_stock` using a grid `p_grid` at the next date t_{i+1} , the continuation cuts values for all regimes `p_condEsp` permitting to calculate an upper estimation (when maximizing) of conditional expectation of the optimal values using some optimization calculated at the previously treated time step t_{i+1} . From a grid point `p_stock` it calculates the function values and the corresponding sensibilities. It returns a matrix (first dimension is the number of simulations by the number of cuts components (number of storage +1), second dimension the number of regimes) giving the function value and sensibilities.
- the `stepSimulate` method is used after optimization using the continuation cuts values calculated in the optimization part. From a state `p_state` (storing the $X^{x,t}$), the continuation cuts values calculated in optimization `p_continuation`, the optimal cash flows along the current trajectory are stored in `p_phiInOut`.

In the case of gas storage, the Optimize object is given in the `OptimizeGasStorageCut.h` file.

The framework in optimization using certain cutting methods

Once an Optimizer object describing the business problem to be solved with cuts is created for the project, and assuming that a full grid is used for the discretization of the stock, the framework provides a `TransitionStepRegressionDPCutDist` object in MPI which solves the optimization problem with data adistribution over a time step with the following constructor:

```
1 TransitionStepRegressionDPCutDist(const shared_ptr<FullGrid> &p_pGridCurrent,
2                                 const shared_ptr<FullGrid> &p_pGridPrevious,
3                                 const shared_ptr<OptimizerDPCutBase> &p_pOptimize,
4                                 const boost::mpi::communicator &p_world):
```

with

- `p_pGridCurrent` is the grid at the current time step (t_i),
- `p_pGridPrevious` is the grid at the previously processed time step (t_{i+1}),
- `p_pOptimize` the optimizer object
- `p_world` The MPI communicator

The construction is very similar to the classical regression methods using only the discretization by command.

Remark 12 *A similar object is available without the MPI distribution framework `TransitionStepRegressionDPCut` with always the activation of parallelization with threads and MPI on calculations on the full grid points.*

The main method is

```
1 std::vector< shared_ptr< Eigen::ArrayXXd > > OneStep(const std::vector< shared_ptr<
2 Eigen::ArrayXXd > > &p_phiIn,
3             const shared_ptr< BaseRegression> &p_condExp)
```

with

- `p_phiIn` the vector (its size corresponds to the number of regimes) of the matrix of optimal values and sensitivities calculated at the previous time iteration for each regime. Each matrix has a number of rows equal to the number of simulations by the number of stock plus one. The number of columns is equal to the number of stock points on the grid. In the row, the number of simulations by the number of stock plus one value is stored as follows:
 - The first values (number of simulations : NS) correspond to the optimal Bellman values at a given stock point,
 - The NS values following corresponds to sensitivities $\frac{\partial V}{\partial S_1}$ at first storage
 - The NS values following corresponds to sensitivities at the second storage...

— ...

- `p_condExp` the conditional expectation operator,

returning a vector of matrix with new optimal values and sensitivities to the current time step (each element of the vector corresponds to a regime and each matrix has a size equal to (number of simulations by (the number of storage plus one)) by the number of stock points). The structure of the output is then similar to the `p_phiIn` input.

A second method is provided allowing to dump the continuation values, cuts of the problem and the optimal control at each time step:

```
1 void dumpContinuationCutsValues(std::shared_ptr<gs::BinaryFileArchive> p_ar , const std
  ::string &p_name, const int &p_iStep,
2                               const std::vector< std::shared_ptr< Eigen::ArrayXXd > > &
  p_phiInPrev,
3                               const std::shared_ptr<BaseRegression> &p_condExp,
4                               const bool &p_bOneFile) const
```

with:

- `p_ar` is the archive where controls and solutions are dumped,
- `p_name` is a base name used in the archive to store the solution and the control,
- `p_phiInPrev` is the solution to the previous time step used to calculate the values of continuation cuts which are stored,
- `p_condExp` is the conditional expectation object allowing to calculate the conditional expectation of the functions defined at the preceding time step processed `p_phiInPrev`.
- `p_bOneFile` is set if the continuation cut values calculated by each processor are dumped on a single file. Otherwise the continuation cut values calculated by each processor are dumped on different files (one per processor). If the problem results from the cuts of the values on the global grid that can be stored in the memory of the compute node, it may be more interesting to dump them in a single file for the simulation of the optimal policy.

Here we give a simple example of temporal resolution using this method when MPI data distribution is used

```
1 // Copyright (C) 2019 EDF
2 // All Rights Reserved
3 // This code is published under the GNU Lesser General Public License (GNU LGPL)
4 #ifdef USE_MPI
5 #include <fstream>
6 #include <memory>
7 #include <functional>
8 #include <boost/lexical_cast.hpp>
9 #include <boost/mpi.hpp>
10 #include <Eigen/Dense>
11 #include "generators/BinaryFileArchive.hh"
12 #include "StOpt/core/grids/FullGrid.h"
13 #include "StOpt/regression/BaseRegression.h"
14 #include "StOpt/dp/FinalStepDPCutDist.h"
15 #include "StOpt/dp/TransitionStepRegressionDPCutDist.h"
16 #include "StOpt/core/parallelism/reconstructProcOMpi.h"
17 #include "StOpt/dp/OptimizerDPCutBase.h"
18 #include "StOpt/dp/SimulatorDPBase.h"
```

```

19
20
21 using namespace std;
22 using namespace Eigen;
23
24 double DynamicProgrammingByRegressionCutDist(const shared_ptr<StOpt::FullGrid> &p_grid,
25 const shared_ptr<StOpt::OptimizerDPCutBase > &p_optimize,
26 shared_ptr<StOpt::BaseRegression> &p_regressor,
27 const function< ArrayXd(const int &, const ArrayXd &, const ArrayXd &)> &
28     p_funcFinalValue,
29 const ArrayXd &p_pointStock,
30 const int &p_initialRegime,
31 const string &p_fileToDump,
32 const bool &p_bOneFile,
33 const boost::mpi::communicator &p_world)
34 {
35     // from the optimizer get back the simulator
36     shared_ptr< StOpt::SimulatorDPBase> simulator = p_optimize->getSimulator();
37     // final values
38     vector< shared_ptr< ArrayXXd > > valueCutsNext = StOpt::FinalStepDPCutDist(p_grid,
39 p_optimize->getNbRegime(), p_optimize->getDimensionToSplit(), p_world)(
40     p_funcFinalValue, simulator->getParticles().array());
41     // dump
42     string toDump = p_fileToDump ;
43     // test if one file generated
44     if (!p_bOneFile)
45         toDump += "_" + boost::lexical_cast<string>(p_world.rank());
46     shared_ptr<gs::BinaryFileArchive> ar;
47     if ((!p_bOneFile) || (p_world.rank() == 0))
48         ar = make_shared<gs::BinaryFileArchive>(toDump.c_str(), "w");
49     // name for object in archive
50     string nameAr = "Continuation";
51     for (int iStep = 0; iStep < simulator->getNbStep(); ++iStep)
52     {
53         ArrayXXd asset = simulator->stepBackwardAndGetParticles();
54         // conditional expectation operator
55         p_regressor->updateSimulations(((iStep == (simulator->getNbStep() - 1)) ? true :
56         false), asset);
57         // transition object
58         StOpt::TransitionStepRegressionDPCutDist transStep(p_grid, p_grid, p_optimize,
59 p_world);
60         vector< shared_ptr< ArrayXXd > > valueCuts = transStep.oneStep(valueCutsNext,
61 p_regressor);
62         transStep.dumpContinuationCutsValues(ar, nameAr, iStep, valueCutsNext, p_regressor,
63 p_bOneFile);
64         valueCutsNext = valueCuts;
65     }
66     // reconstruct a small grid for interpolation
67     ArrayXd valSim = StOpt::reconstructProc0Mpi(p_pointStock, p_grid, valueCutsNext[
68 p_initialRegime], p_optimize->getDimensionToSplit(), p_world);
69     return ((p_world.rank() == 0) ? valSim.head(simulator->getNbSimul()).mean() : 0.);
70 }
71 #endif

```

An example without data distribution can be found in the file `DynamicProgrammingByRegressionCut.cpp`.

3.2.4 Regressions and cuts for local concave/convex problems

Managing stocks and taking into account real behavior of assets, global concavity can be broken. The value function to estimate is only locally concave and previous LP resolution has to be adapted. StOpt provides a framework to deal with such cases for grids in 1D or 2D (seen section 1.4), and continuation objects to deal with cuts (section 4.1.4). To use this

cuts a business object using an LP solver must be created.

```

1 // Copyright (C) 2025 EDF
2 // All Rights Reserved
3 // This code is published under the GNU Lesser General Public License (GNU LGPL)
4 #ifndef OPTIMIZERDPCUTGRIDADAPTBASE_H
5 #define OPTIMIZERDPCUTGRIDADAPTBASE_H
6 #include <memory>
7 #include <Eigen/Dense>
8 #include "StOpt/core/utils/StateWithStocks.h"
9 #include "StOpt/core/grids/GridAdaptBase.h"
10 #include "StOpt/regression/BaseRegression.h"
11 #include "StOpt/regression/ContinuationCutsGridAdaptNonConcave.h"
12 #include "StOpt/dp/SimulatorDPBase.h"
13 #include "StOpt/dp/OptimizerBase.h"
14
15 /** \file OptimizerDPCutGridAdaptBase.h
16  * \brief Define an abstract class for Dynamic Programming problems solved by regression
17  *        methods using cust to approximate
18  *        Bellman values
19  *        Special version for stocks, concavity non supposed (it is imposed by mesh).
20  *        Adaptation of the grid provided.
21  *        \author Xavier Warin
22  */
23 namespace StOpt
24 {
25     /// \class OptimizerDPCutGridAdaptBase OptimizerDPCutGridAdaptBase.h
26     /// Base class for optimizer for Dynamic Programming with regression methods and cuts, so
27     /// using LP to solve transitional problems
28     class OptimizerDPCutGridAdaptBase : public OptimizerBase
29     {
30     public :
31
32         OptimizerDPCutGridAdaptBase() {}
33
34         virtual ~OptimizerDPCutGridAdaptBase() {}
35
36         /// \brief defines the dimension to split for MPI parallelism
37         /// For each dimension return true is the direction can be split
38         virtual Eigen::Array< bool, Eigen::Dynamic, 1> getDimensionToSplit() const = 0 ;
39
40         /// \brief defines a step in optimization
41         /// \param p_stock coordinates of the stock point to treat
42         /// \param p_condEsp continuation values
43         /// \return For a given simulation , cuts components (C) at a point stock \ $ \bar S
44         /// \f$ are given such that the cut is given by
45         /// \f$ C[0] + \sum_{i=1}^d C[i] (S_i - \bar S_i) \quad \f$
46         virtual Eigen::ArrayXd stepOptimize(const Eigen::ArrayXd &p_stock, const StOpt::
47             ContinuationCutsGridAdaptNonConcave &p_condEsp) const = 0;
48
49         /// \brief defines a step in simulation
50         /// Control are recalculated during simulation using a local optimization using the LP
51         /// \param p_continuation defines the continuation operator
52         /// \param p_state defines the state value (modified)
53         /// \param p_phiInOut defines the value functions (modified) : size number of
54         /// functions to follow
55         virtual void stepSimulate(const StOpt::ContinuationCutsGridAdaptNonConcave &
56             p_continuation,
57             StOpt::StateWithStocks<double> &p_state,
58             Eigen::Ref<Eigen::ArrayXd> p_phiInOut) const = 0 ;
59
60         /// \brief define a refinement of the grid to get a given precision
61         /// \param p_gridToAdapt grid to be adapted

```

```

61  /// \return True if a refinement has been achieved in the grid
62  virtual bool refineGrid( std::shared_ptr<GridAdaptBase> & p_gridToAdapt, const Eigen
    ::ArrayXXd & pCuts ) =0;
63
64  // virtual void setRefinementToFalse()=0;
65
66  /// \brief get the simulator back
67  virtual std::shared_ptr< StOpt::SimulatorDPBase > getSimulator() const = 0;
68
69  /// \brief get back the dimension of the control
70  virtual int getNbControl() const { return 0;}
71
72  /// \brief get size of the function to follow in simulation
73  virtual int getSimuFuncSize() const = 0;
74
75 };
76 }
77 #endif /* OPTIMIZERDPCUTBASE_H */

```

We detail the main methods associated to the object:

- the `stepOptimize` method is used in optimization. We want to calculate the optimal value and the corresponding sensibilities with respect to the stocks at current t_i at a grid point `p_stock` using a grid `p_grid` at the next date t_{i+1} , the continuation cuts values `p_condEsp` permitting to calculate an upper estimation (when maximizing) of conditional expectation of the optimal values using some optimization calculated at the previously treated time step t_{i+1} . From a grid point `p_stock` it calculates the function values and the corresponding sensibilities. It returns a vector (the number of simulations by the number of cuts components (number of storage +1)) giving the function value and sensibilities.
- the `stepSimulate` method is used after optimization using the continuation cuts values calculated in the optimization part. From a state `p_state` (storing the $X^{x,t}$), the continuation cuts values calculated in optimization `p_continuation`, the optimal cash flows along the current trajectory are stored in `p_phiInOut`.

In the case of fictitious gas storage in 2D , the Optimize object is given in the `Optimize GasStorageCutGridAdapt2D.h` file.

The framework in optimization using certain cutting methods local non concavity

Once an Optimizer object describing the business problem to be solved with cuts is created for the project, and assuming that an adapted grid is used for the discretization of the stock, the framework provides a `TransitionStepRegressionDPCutGridAdapt` object in MPI which solves the optimization problem over a time step with the following constructor:

```

1 TransitionStepRegressionDPCutGridAdapt(std::shared_ptr<GridAdaptBase> &p_pGridCurrent,
2   const std::shared_ptr<GridAdaptBase> &p_pGridPrevious,
3   const std::shared_ptr<OptimizerDPCutGridAdaptBase > &p_pOptimize,
4   const boost::mpi::communicator &p_world
5   );

```

with

- `p_pGridCurrent` is the grid at the current time step (t_i),

- `p_pGridPrevious` is the grid at the previously processed time step (t_{i+1}),
- `p_pOptimize` the optimizer object
- `p_world` The MPI communicator

The construction is very similar to the classical regression methods using only the discretization by command. The main method is

```
1 Eigen::ArrayXXd oneStep(const Eigen::ArrayXXd &p_phiIn, const std::shared_ptr<
  BaseRegression> &p_condExp)
```

with

- `p_phiIn` a matrix of optimal values and sensitivities calculated at the previous time iteration for each regime. Each matrix has a number of rows equal to the number of simulations by the number of stock plus one. The number of columns is equal to the number of stock points on the grid. In the row, the number of simulations by the number of stock plus one value is stored as follows:
 - The first values (number of simulations : NS) correspond to the optimal Bellman values at a given stock point,
 - The NS values following corresponds to sensitivities $\frac{\partial V}{\partial S_1}$ at first storage
 - The NS values following corresponds to sensitivities at the second storage...
 - ...
- `p_condExp` the conditional expectation operator,

returning a matrix with new optimal values and sensitivities to the current time step (size equal to (number of simulations by (the number of storage plus one)) by the number of stock points). The structure of the output is then similar to the `p_phiIn` input.

A second method is provided allowing to dump the continuation values, cuts of the problem and the optimal control at each time step:

```
1 void dumpContinuationCutsValues(std::shared_ptr<gs::BinaryFileArchive> p_ar, const std
  ::string &p_name, const int &p_iStep, const Eigen::ArrayXXd &p_phiIn, const std
  ::shared_ptr<BaseRegression> &p_condExp) const
```

with:

- `p_ar` is the archive where controls and solutions are dumped,
- `p_name` is a base name used in the archive to store the solution and the control,
- `p_phiIn` is the solution to the previous time step used to calculate the values of continuation cuts which are stored,
- `p_condExp` is the conditional expectation object allowing to calculate the conditional expectation of the functions defined at the preceding time step processed `p_phiInPrev`.

Here we give a simple example of temporal resolution using this method

```

1 // Copyright (C) 2025 EDF
2 // All Rights Reserved
3 // This code is published under the GNU Lesser General Public License (GNU LGPL)
4 #include <fstream>
5 #include <memory>
6 #include <functional>
7 #ifdef USE_MPI
8 #include <boost/mpi.hpp>
9 #endif
10 #include <boost/lexical_cast.hpp>
11 #include <Eigen/Dense>
12 #include "geners/BinaryFileArchive.hh"
13 #include "StOpt/regression/BaseRegression.h"
14 #include "StOpt/core/grids/GridAdapt2D.h"
15 #include "StOpt/dp/FinalStepDPCut.h"
16 #include "StOpt/dp/TransitionStepRegressionDPCutGridAdapt.h"
17 #include "StOpt/dp/OptimizerDPCutGridAdaptBase.h"
18
19 using namespace std;
20 using namespace Eigen;
21
22
23 double DynamicProgrammingByRegressionCutGridAdapt2D(vector<shared_ptr<StOpt::GridAdaptBase
24 > > &p_grids,
25 const shared_ptr<StOpt::OptimizerDPCutGridAdaptBase > &p_optimize,
26 const shared_ptr<StOpt::BaseRegression> &p_regressor,
27 const string &p_fileToDump
28 #ifdef USE_MPI
29 , const boost::mpi::communicator &p_world
30 #endif
31 )
32 {
33     // from the optimizer get back the simulator
34     shared_ptr< StOpt::SimulatorDPBase> simulator = p_optimize->getSimulator();
35     // final cut values
36     shared_ptr<ArrayXXd> valueCutsNext = make_shared<ArrayXXd>(ArrayXXd::Zero(3 *
37 simulator->getNbSimul(), p_grids[simulator->getNbStep()->getNbPoints()]));
38     shared_ptr<gs::BinaryFileArchive> ar = make_shared<gs::BinaryFileArchive>(p_fileToDump.
39 c_str(), "w");
40     // name for object in archive
41     string nameAr = "Continuation";
42
43     // iterate on time steps
44     for (int iStep = 0; iStep < simulator->getNbStep(); ++iStep)
45     {
46 #ifdef USE_MPI
47         if (p_world.rank() == 0)
48         {
49             cout << " iStep " << iStep << endl ;
50         }
51 #else
52         cout << " iStep " << iStep << endl ;
53 #endif
54         ArrayXXd asset = simulator->stepBackwardAndGetParticles();
55         // conditional expectation operator
56         p_regressor->updateSimulations(((iStep == (simulator->getNbStep() - 1)) ? true :
57 false), asset);
58         // transition object
59         StOpt::TransitionStepRegressionDPCutGridAdapt transStep(p_grids[ simulator->
60 getNbStep() - iStep - 1], p_grids[ simulator->getNbStep() - iStep], p_optimize
61 #ifdef USE_MPI
62 , p_world
63 #endif
64 );
65         shared_ptr<ArrayXXd> valueCuts = make_shared<ArrayXXd>(transStep.oneStep(*
66 valueCutsNext, p_regressor));
67         // dump continuation values
68         transStep.dumpContinuationCutsValues(ar, nameAr, iStep, *valueCutsNext,

```

```

        p_regressor);
63
64        valueCutsNext = valueCuts;
65    }
66    // Only keep the first nbSimul values of the cuts (other components are for
        derivatives with respect to stock value)
67    return valueCutsNext->col(0).head(simulator->getNbSimul()).mean();
68 }

```

The simulation framework using cuts to approximate the Bellman's values

Once the optimization carried out, the values of the continuation cuts are saved in a file at each time step. In order to simulate an optimal policy time step, a `SimulateStepRegressionCutGridAdapt` object is provided with constructor

```

1  SimulateStepRegressionCutGridAdapt(gs::BinaryFileArchive &p_ar,
2  const int &p_iStep,
3  const std::string &p_nameCont,
4  const std::shared_ptr<StOpt::OptimizerDPCutGridAdaptBase > &p_pOptimize,
5  const boost::mpi::communicator &p_world)

```

where

- `p_ar` is the binary archive where the continuation values are stored,
- `p_iStep` is the number associated with the current time step (0 at the start date of the simulation, the number is increased by one at each simulated time step),
- `p_nameCont` is the base name for continuation values,
- `p_Optimize` the Optimizer describing the transition problem solved using a LP program.
- `p_world` The MPI communicator

This object implements the `oneStep` method

```

1 void oneStep(std::vector<StateWithStocks<double> > &p_statevector , Eigen::ArrayXXd &
    p_phiInOut)

```

where:

- `p_statevector` stores the states of all the simulations: this state is updated by applying the optimal command,
- `p_phiInOut` stores the gain/cost functions for all simulations: it is updated by the function call. The size of the array is $(nb, nbSimul)$ where nb is given by the `getSimuFuncSize` method of the optimizer and $nbSimul$ the number of Monte Carlo simulations.

An example of using this method to simulate an optimal policy with distribution is given below:

```

1 // Copyright (C) 2025 EDF
2 // All Rights Reserved
3 // This code is published under the GNU Lesser General Public License (GNU LGPL)
4 #ifndef SIMULATEREGRESSIONCUTGRIDADAPT2D_H
5 #define SIMULATEREGRESSIONCUTGRIDADAPT2D_H
6 #include <Eigen/Dense>
7 #include <functional>
8 #include <memory>
9 #ifdef USE_MPI
10 #include <boost/mpi.hpp>
11 #endif
12 #include "geners/BinaryFileArchive.hh"
13 #include "StOpt/core/utils/StateWithStocks.h"
14 #include "StOpt/core/grids/GridAdapt2D.h"
15 #include "StOpt/regression/BaseRegression.h"
16 #include "StOpt/dp/SimulateStepRegressionCutGridAdapt.h"
17 #include "StOpt/dp/OptimizerDPCutGridAdaptBase.h"
18 #include "StOpt/dp/SimulatorDPBase.h"
19
20
21 /** \file SimulateRegressionCutGridAdapt2D.h
22  * \brief Defines a simple program showing how to use simulations when optimizat on achieved
23  *         with transition problems solved with cuts.
24  *         A 2D grid with adaptation is used. Cuts ae done on each mesh so no concavty is
25  *         required
26  * \author Xavier Warin
27 */
28
29 /// \brief Simulate the optimal strategy , Bellman cuts used to allow LP resolution of
30 /// transition problems
31 /// \param p_optimize          optimizer defining the optimization between two time
32 ///                             steps
33 /// \param p_pointStock        initial point stock
34 /// \param p_fileToDump        name of the file used to dump continuation values in
35 ///                             optimization
36 /// \param p_world             MPI communicator
37 double SimulateRegressionCutGridAdapt2D(const std::shared_ptr<StOpt::
38     OptimizerDPCutGridAdaptBase > &p_optimize,
39     const Eigen::ArrayXd &p_pointStock,
40     const std::string &p_fileToDump
41 #ifdef USE_MPI
42     , const boost::mpi::communicator &p_world
43 #endif
44 )
45 {
46     // from the optimizer get back the simulator
47     std::shared_ptr< StOpt::SimulatorDPBase> simulator = p_optimize->getSimulator();
48     int nbStep = simulator->getNbStep();
49     std::vector< StOpt::StateWithStocks<double>> states;
50     int initRegime = 0; // no regime
51     states.reserve(simulator->getNbSimul());
52     for (int is = 0; is < simulator->getNbSimul(); ++is)
53         states.push_back(StOpt::StateWithStocks<double>(initRegime, p_pointStock, Eigen::
54             ArrayXd::Zero(simulator->getDimension())));
55     gs::BinaryFileArchive ar(p_fileToDump.c_str(), "r");
56     // name for continuation object in archive
57     std::string nameAr = "Continuation";
58     // cost function
59     Eigen::ArrayXXd costFunction = Eigen::ArrayXXd::Zero(p_optimize->getSimuFuncSize(),
60         simulator->getNbSimul());
61     // iterate on time steps
62     for (int istep = 0; istep < nbStep; ++istep)
63     {
64         StOpt::SimulateStepRegressionCutGridAdapt(ar, nbStep - 1 - istep, nameAr,
65             p_optimize
66 #ifdef USE_MPI
67             , p_world
68 #endif
69     }
70 }

```

```

60                                     ).oneStep(states, costFunction);
61         // new stochastic state
62         Eigen::ArrayXXd particles = simulator->stepForwardAndGetParticles();
63         for (int is = 0; is < simulator->getNbSimul(); ++is)
64             states[is].setStochasticRealization(particles.col(is));
65     }
66     // average gain/cost
67     return costFunction.mean();
68 }
69 #endif /* SIMULATEREGRESSIONCUTGRIDCUTADPAT2D_H */

```

3.3 Solve the problem for $X_2^{x,t}$ stochastic

In this part, we assume that $X_2^{x,t}$ is controlled but is stochastic.

3.3.1 Requirement to use the framework

To use the framework, a business object describing the business at a time step from a state is derived from `OptimizerNoRegressionDPBase` itself derived from `OptimizerBase`.

```

1 // Copyright (C) 2016 EDF
2 // All Rights Reserved
3 // This code is published under the GNU Lesser General Public License (GNU LGPL)
4 #ifndef OPTIMIZERNOREGRESSIONDPBASE_H
5 #define OPTIMIZERNOREGRESSIONDPBASE_H
6 #include <Eigen/Dense>
7 #include "StOpt/core/utils/StateWithStocks.h"
8 #include "StOpt/core/grids/SpaceGrid.h"
9 #include "StOpt/regression/BaseRegression.h"
10 #include "StOpt/regression/GridAndRegressedValue.h"
11 #include "StOpt/dp/SimulatorDPBase.h"
12 #include "StOpt/dp/OptimizerBaseInterp.h"
13
14 /** \file OptimizerNoRegressionDPBase.h
15  * \brief Define an abstract class for Dynamic Programming problems solve by Monte Carlo
16  *         but without regression method
17  *         to compute conditional expectation.
18  *         \author Xavier Warin
19  */
20 namespace StOpt
21 {
22
23     /// \class OptimizerNoRegressionDPBase OptimizerNoRegressionDPBase.h
24     /// Base class for optimizer for Dynamic Programming solved without regression method to
25     /// compute conditional expectation.
26     class OptimizerNoRegressionDPBase : public OptimizerBaseInterp
27     {
28     public :
29
30         OptimizerNoRegressionDPBase() {}
31
32         virtual ~OptimizerNoRegressionDPBase() {}
33
34         /// \brief defines the diffusion cone for parallelism
35         /// \param p_regionByProcessor region (min max) treated by the processor for
36         ///         the different regimes treated
37         /// \return returns in each dimension the min max values in the stock that can be
38         ///         reached from the grid p_gridByProcessor for each regime
39         virtual std::vector< std::array< double, 2> > getCone(const std::vector< std::array<
40             double, 2> > &p_regionByProcessor) const = 0;

```

```

39
40 /// \brief defines the dimension to split for MPI parallelism
41 /// For each dimension return true is the direction can be split
42 virtual Eigen::Array< bool, Eigen::Dynamic, 1> getDimensionToSplit() const = 0 ;
43
44 /// \brief defines a step in optimization
45 /// \param p_stock coordinates of the stock point to treat
46 /// \param p_valNext Optimized values at next time step for each regime
47 /// \param p_regressorCur Regressor at the current date
48 /// \return a pair :
49 /// - for each regimes (column) gives the solution for each particle (row)
50 /// - for each control (column) gives the optimal control for each
    particle (rows)
51 ///
52 virtual std::pair< Eigen::ArrayXXd, Eigen::ArrayXXd> stepOptimize(const Eigen::
    ArrayXd &p_stock,
53     const std::vector< GridAndRegressedValue > &p_valNext,
54     std::shared_ptr< BaseRegression > p_regressorCur) const = 0;
55
56
57
58 /// \brief Defines a step in simulation using interpolation in controls
59 /// \param p_grid grid at arrival step after command
60 /// \param p_control defines the controls
61 /// \param p_state defines the state value (modified)
62 /// \param p_phiInOut defines the value function (modified): size number of
    functions to follow
63 virtual void stepSimulateControl(const std::shared_ptr< StOpt::SpaceGrid> &p_grid,
    const std::vector< StOpt::GridAndRegressedValue > &p_control,
64     StOpt::StateWithStocks<double> &p_state,
65     Eigen::Ref<Eigen::ArrayXd> p_phiInOut) const = 0 ;
66
67
68
69 /// \brief Get the number of regimes allowed for the asset to be reached at the
    current time step
70 /// If \f$ t \f$ is the current time, and \f$ dt \f$ the resolution step, this is
    the number of regime allowed on \f$[ t- dt, t[\f$
71 virtual int getNbRegime() const = 0 ;
72
73 /// \brief get the simulator back
74 virtual std::shared_ptr< StOpt::SimulatorDPBase > getSimulator() const = 0;
75
76 /// \brief get back the dimension of the control
77 virtual int getNbControl() const = 0 ;
78
79 /// \brief get size of the function to follow in simulation
80 virtual int getSimuFuncSize() const = 0;
81
82 };
83 }
84 #endif /* OPTIMIZERDPBASE_H */

```

In addition to the OptimizerBase methods, the following method is required:

- the `stepOptimize` method is used in optimization. We want to calculate the optimal value regressed to t_i current at a grid point `p_stock` using a `p_grid` grid at the following date t_{i+1} ,

From a `p_stock` grid point, it calculates the regressed function values and the regressed optimal controls. It returns a pair where the

- the first element is a matrix (the first dimension is the number of functions in the regression, the second dimension the number of regimes) giving the value of the regressed function,

- the second element is a matrix (the first dimension is the number of functions in the regression, the second dimension the number of controls) giving the optimal regressed control.

In this case of the optimization of an updated portfolio with dynamic:

$$dX_2^{x,t} = X_2^{x,t} \frac{dX_1^{x,t}}{X_1^{x,t}}$$

where $X_1^{x,t}$ is the value of the risky asset, the Optimize object is given in the file `OptimizePortfolio.h`.

3.3.2 The framework in optimization

Once an Optimizer is derived for the project, and assuming a full grid is used for stock discretization, the framework provides a `TransitionStepDPDist` object in MPI which solves the optimization problem with the distribution of data over a time step with the following constructor:

```
1  TransitionStepDPDist(const shared_ptr<FullGrid> &p_pGridCurrent,
2  const shared_ptr<FullGrid> &p_pGridPrevious,
3  const std::shared_ptr<BaseRegression> &p_regressorCurrent,
4  const std::shared_ptr<BaseRegression> &p_regressorPrevious,
5  const shared_ptr<OptimizerNoRegressionDPBase> &p_pOptimize,
6  const boost::mpi::communicator &p_world):
```

with

- `p_pGridCurrent` is the grid at the current time step (t_i),
- `p_pGridPrevious` is the grid at the previously processed time step (t_{i+1}),
- `p_regressorCurrent` is a regressor at the current date (to evaluate the function at the current date)
- `p_regressorPrevious` is a previously processed time step regressor (t_{i+1}) allowing to evaluate a function at date t_{i+1} ,
- `p_pOptimize` the optimizer object
- `p_world` The MPI communicator

Remark 13 *A similar object is available without the MPI distribution framework `TransitionStepDP` with always the activation of parallelization with threads and MPI on the calculations on the full grid points.*

Remark 14 *The case of sparse grids is not currently dealt with in the framework.*

The main method is

```
1  std::pair< std::shared_ptr< std::vector< Eigen::ArrayXXd > >, std::shared_ptr< std::vector< Eigen::ArrayXXd > > > oneStep(const std::vector< Eigen::ArrayXXd > &p_phiIn)
```

with

- **p_phi** In the vector (its size corresponds to the number of regimes) of the matrix of calculated optimal values regressed at the previous time iteration for each regime. Each matrix is a number of regressor function on the previous date by number of matrix stock points.

return a pair:

- the first element is a vector of matrix with new optimal values regressed at the current time step (each element of the vector corresponds to a regime and each matrix is a number of functions regressed at the current date by the number of stock points of the matrix).
- the second element is a vector of matrix with new optimal controls regressed at the current time step (each element of the vector corresponds to a control and each matrix is a number of controls regressed by the number of matrix stock points).

Remark 15 *All `TransitionStepDP` derive from a `TransitionStepBase` object with a pure virtual `OneStep` method.*

A second method is provided allowing to dump the optimal control at each time step:

```
1 void dumpValues(std::shared_ptr<gs::BinaryFileArchive> p_ar ,  
2               const std::string &p_name, const int &p_iStep,  
3               const std::vector< Eigen::ArrayXXd > &p_control, const bool &  
                  p_bOneFile) const
```

with:

- **p_ar** is the archive where controls and solutions are dumped,
- **p_name** is a base name used in the archive to store the solution and the control,
- **p_control** stores the optimal controls calculated at the current time step,
- **p_bOneFile** is set to one if the optimal controls calculated by each processor are dumped on a single file. Otherwise the optimal controls calculated by each processor are dumped on different files (one by processor). If the problem gives optimal control values on the global grid that can be stored in the memory of the computation node, it can be more interesting to dump the control values in one file for the simulation of the optimal policy.

Remark 16 *As for the `TransitionStepDP`, its `dumpValues` does not need a `p_bOneFile` argument: obviously optimal controls are stored in a single file.*

Here we give a simple example of temporal resolution using this method when MPI data distribution is used

```

1 // Copyright (C) 2016 EDF
2 // All Rights Reserved
3 // This code is published under the GNU Lesser General Public License (GNU LGPL)
4 #ifndef USE_MPI
5 #include <fstream>
6 #include <boost/mpi.hpp>
7 #include <memory>
8 #include <functional>
9 #include <boost/lexical_cast.hpp>
10 #include <Eigen/Dense>
11 #include "geners/BinaryFileArchive.hh"
12 #include "StOpt/core/grids/FullGrid.h"
13 #include "StOpt/regression/LocalConstRegression.h"
14 #include "StOpt/regression/GridAndRegressedValue.h"
15 #include "StOpt/dp/FinalStepDPDist.h"
16 #include "StOpt/dp/TransitionStepDPDist.h"
17 #include "StOpt/core/parallelism/reconstructProcOMpi.h"
18 #include "test/c++/tools/dp/OptimizePortfolioDP.h"
19
20 using namespace std;
21 using namespace Eigen;
22
23 double DynamicProgrammingPortfolioDist(const shared_ptr<StOpt::FullGrid> &p_grid,
24                                       const shared_ptr<OptimizePortfolioDP> &p_optimize,
25                                       const ArrayXi &p_nbMesh,
26                                       const function<double(const int &, const ArrayXd &,
27                                                           const ArrayXd &)> &p_funcFinalValue,
28                                       const ArrayXd &p_initialPortfolio,
29                                       const string &p_fileToDump,
30                                       const bool &p_bOneFile,
31                                       const boost::mpi::communicator &p_world
32 {
33     // initialize simulation
34     p_optimize->initializeSimulation();
35     // store regressor
36     shared_ptr<StOpt::LocalConstRegression> regressorPrevious;
37
38     // store final regressed values in object valuesStored
39     shared_ptr< vector< ArrayXXd > > valuesStored = make_shared< vector<ArrayXXd> >(
40         p_optimize->getNbRegime());
41     {
42         vector< shared_ptr< ArrayXXd > > valuesPrevious = StOpt::FinalStepDPDist(p_grid,
43             p_optimize->getNbRegime(), p_optimize->getDimensionToSplit(), p_world)(
44             p_funcFinalValue, *p_optimize->getCurrentSim());
45         // regressor operator
46         regressorPrevious = make_shared<StOpt::LocalConstRegression>(false, *p_optimize->
47             getCurrentSim(), p_nbMesh);
48         for (int iReg = 0; iReg < p_optimize->getNbRegime(); ++iReg)
49             (*valuesStored)[iReg] = regressorPrevious->getCoordBasisFunctionMultiple(
50                 valuesPrevious[iReg]->transpose()).transpose();
51     }
52     string toDump = p_fileToDump ;
53     // test if one file generated
54     if (!p_bOneFile)
55         toDump += "_" + boost::lexical_cast<string>(p_world.rank());
56     shared_ptr<gs::BinaryFileArchive> ar;
57     if ((!p_bOneFile) || (p_world.rank() == 0))
58         ar = make_shared<gs::BinaryFileArchive>(toDump.c_str(), "w");
59     // name for object in archive
60     string nameAr = "OptimizePort";
61     // iterate on time steps
62     for (int iStep = 0; iStep < p_optimize->getNbStep(); ++iStep)
63     {
64         // step backward for simulations
65         p_optimize->oneStepBackward();
66         // create regressor at the given date
67         bool bZeroDate = (iStep == p_optimize->getNbStep() - 1);

```

```

63     shared_ptr<StOpt::LocalConstRegression> regressorCur = make_shared<StOpt::
        LocalConstRegression>(bZeroDate, *p_optimize->getCurrentSim(), p_nbMesh);
64     // transition object
65     StOpt::TransitionStepDPDist transStep(p_grid, p_grid, regressorCur,
        regressorPrevious, p_optimize, p_world);
66     pair< shared_ptr< vector< ArrayXXd> >, shared_ptr< vector< ArrayXXd > > >
        valuesAndControl = transStep.oneStep(*valuesStored);
67     // dump control values
68     transStep.dumpValues(ar, nameAr, iStep, *valuesAndControl.second, p_bOneFile);
69     valuesStored = valuesAndControl.first;
70     // shift regressor
71     regressorPrevious = regressorCur;
72 }
73 // interpolate at the initial stock point and initial regime( 0 here) (take first
    particle)
74 shared_ptr<ArrayXXd> topRows = make_shared<ArrayXXd>((*valuesStored)[0].topRows(1));
75 return StOpt::reconstructProcOMpi(p_initialPortfolio, p_grid, topRows, p_optimize->
    getDimensionToSplit(), p_world).mean();
76 }
77 #endif

```

An example without data distribution can be found in the file `DynamicProgrammingPortfolio.cpp`.

3.3.3 The simulation framework

No special framework is available in simulation. Use the `SimulateStepRegressionControl` or `SimulateStepRegressionControlDist` function described in the 4.2.1 section.

3.4 Solving stock problems with trees

In this section we detail how to solve problems

- Either by discretizing the control,
- Or by solving a Linear Program problem approaching Bellman values by cuts.

3.4.1 Resolution of dynamic programming problems with the discretization of commands

Framework usage condition

The `OptimizerDPTreeBase` is the base object from which each business object is to be derived.

```

1 // Copyright (C) 2019 EDF
2 // All Rights Reserved
3 // This code is published under the GNU Lesser General Public License (GNU LGPL)
4 #ifndef OPTIMIZERDPTREEBASE_H
5 #define OPTIMIZERDPTREEBASE_H
6 #include <Eigen/Dense>
7 #include "StOpt/core/grids/SpaceGrid.h"
8 #include "StOpt/tree/Tree.h"
9 #include "StOpt/tree/StateTreeStocks.h"
10 #include "StOpt/tree/ContinuationValueTree.h"
11 #include "StOpt/tree/GridTreeValue.h"
12 #include "StOpt/dp/SimulatorDPBaseTree.h"
13 #include "StOpt/dp/OptimizerBase.h"

```

```

14
15 /** \file OptimizerDPTreeBase.h
16 * \brief Define an abstract class for Dynamic Programming problems solved by tree
    methods
17 * \author Xavier Warin
18 */
19
20 namespace StOpt
21 {
22
23 /// \class OptimizerDPTreeBase OptimizerDPTreeBase.h
24 /// Base class for optimizer for Dynamic Programming with tree methods
25 class OptimizerDPTreeBase : public OptimizerBase
26 {
27
28
29 public :
30
31     OptimizerDPTreeBase() {}
32
33     virtual ~OptimizerDPTreeBase() {}
34
35     /// \brief defines the diffusion cone for parallelism
36     /// \param p_regionByProcessor region (min max) treated by the processor for
    the different regimes treated
37     /// \return returns in each dimension the min max values in the stock that can be
    reached from the grid p_gridByProcessor for each regime
38     virtual std::vector< std::array< double, 2> > getCone(const std::vector< std::array<
    double, 2> > &p_regionByProcessor) const = 0;
39
40     /// \brief defines the dimension to split for MPI parallelism
41     /// For each dimension return true is the direction can be split
42     virtual Eigen::Array< bool, Eigen::Dynamic, 1> getDimensionToSplit() const = 0 ;
43
44     /// \brief defines a step in optimization
45     /// \param p_grid grid at arrival step after command
46     /// \param p_stock coordinates of the stock point to treat
47     /// \param p_condEsp continuation values for each regime
48     /// \return a pair :
49     /// - for each regimes (column) gives the solution for each node in the
    tree (row)
50     /// - for each control (column) gives the optimal control for each node in
    the tree (rows)
51     /// .
52     virtual std::pair< Eigen::ArrayXXd, Eigen::ArrayXXd> stepOptimize(const std::
    shared_ptr< StOpt::SpaceGrid> &p_grid, const Eigen::ArrayXd &p_stock,
    const std::vector< StOpt::ContinuationValueTree > &p_condEsp) const = 0;
53
54
55
56     /// \brief defines a step in simulation
57     /// Notice that this implementation is not optimal but is convenient if the control is
    discrete.
58     /// By avoiding interpolation in control we avoid non admissible control
59     /// Control are recalculated during simulation.
60     /// \param p_grid grid at arrival step after command
61     /// \param p_continuation defines the continuation operator for each regime
62     /// \param p_state defines the state value (modified)
63     /// \param p_phiInOut defines the value functions (modified) : size number of
    functions to follow
64     virtual void stepSimulate(const std::shared_ptr< StOpt::SpaceGrid> &p_grid, const std
    ::vector< StOpt::GridTreeValue > &p_continuation,
    StOpt::StateTreeStocks &p_state,
    Eigen::Ref<Eigen::ArrayXd> p_phiInOut) const = 0 ;
65
66
67
68
69     /// \brief Defines a step in simulation using interpolation in controls
70     /// \param p_grid grid at arrival step after command
71     /// \param p_control defines the controls
72     /// \param p_state defines the state value (modified)

```

```

73  /// \param p_phiInOut    defines the value function (modified): size number of
                          functions to follow
74  virtual void stepSimulateControl(const std::shared_ptr< StOpt::SpaceGrid>    &p_grid,
                          const std::vector< StOpt::GridTreeValue > &p_control,
75                                  StOpt::StateTreeStocks &p_state,
76                                  Eigen::Ref<Eigen::ArrayXd> p_phiInOut) const = 0 ;
77
78
79
80  /// \brief Get the number of regimes allowed for the asset to be reached at the
                          current time step
81  /// If \f$ t \f$ is the current time, and \f$ dt \f$ the resolution step, this is
                          the number of regime allowed on \f$[ t- dt, t[\f$
82  virtual int getNbRegime() const = 0 ;
83
84  /// \brief get the simulator back
85  virtual std::shared_ptr< StOpt::SimulatorDPBaseTree > getSimulator() const = 0;
86
87  /// \brief get back the dimension of the control
88  virtual int getNbControl() const = 0 ;
89
90  /// \brief get size of the function to follow in simulation
91  virtual int getSimuFuncSize() const = 0;
92
93 };
94 }
95 #endif /* OPTIMIZERDPTREEBASE_H */

```

Since the `OptimizerDPTreeBase` class is derived from `OptimizerBase`, we detail the additional methods required:

- the `stepOptimize` method is used in optimization. We want to calculate the optimal value at the current t_i at a grid point `p_stock` using a `p_grid` grid at the following date t_{i+1} , the continuation values for all the `p_condEsp` regimes making it possible to calculate the conditional expectation of the optimal value function calculated at the previously processed time step t_{i+1} . From a grid point `p_stock` it calculates the function values and the optimal controls. It returns a pair where the
 - the first element is a matrix (the first dimension is the number of nodes, the second dimension the number of regimes) giving the value of the function,
 - the second element is a matrix (the first dimension is the number of nodes, the second dimension the number of controls) giving the optimal control.
- the `stepSimulate` method is used after the optimization using the continuation values calculated in the optimization part: these continuation values are stored in a `GridTreeValue` object for interpolation. From a `p_state` state (storing the $X^{x,t}$), the continuation values calculated in the optimization `p_continuation`, the optimal values of the stored functions in `p_phiInOut`.
- the `stepSimulateControl` simulates the strategy by direct interpolation of the control for a given node in the tree (sampled) and a position in the stock.

The framework

Most of the objects used with regression have their counterparts with tree methods. In optimization:

TransitionStepTreeDPDist resolves the transition to the date for all grid point and tree nodes using the distribution (MPI).

```

1 TransitionStepTreeDPDist(const std::shared_ptr<FullGrid> &p_pGridCurrent,
2                          const std::shared_ptr<FullGrid> &p_pGridPrevious,
3                          const std::shared_ptr<OptimizerDPTreeBase> &p_pOptimize,
4                          const boost::mpi::communicator &p_world)

```

with

- `p_pGridCurrent` is the grid at the current time step (t_i),
- `p_pGridPrevious` is the grid at the previously processed time step (t_{i+1}),
- `p_pOptimize` the optimizer object
- `p_world` The MPI communicator

A similar object is available without the MPI distribution framework `TransitionStepTreeDP` with always the activation of parallelization with threads and MPI on calculations on the full grid points.

The main method is

```

1 std::pair< std::vector< std::shared_ptr< Eigen::ArrayXXd > >, std::vector< std:::
  shared_ptr< Eigen::ArrayXXd > > > oneStep(const std::vector< std::shared_ptr<
  Eigen::ArrayXXd > > &p_phiIn,
2      const std::shared_ptr< Tree> &p_condExp) const

```

with

- `p_phiIn` the vector (its size corresponds to the number of regimes) of the matrix of optimal values calculated at the previous time iteration for each regime. Each matrix is a number of nodes the next date per number of stock points matrix.
- `p_condExp` the conditional expectation operator,

return a pair:

- first element is a vector of matrix with new optimal values at the current time step (each element of the vector corresponds to a regime and each matrix is a number of nodes at current date by number of points of stock matrix).
- The second element is a vector of matrix with new optimal controls at the current time step (each element of the vector corresponds to a control and each matrix is a number of nodes at the current date by number of points of stock matrix).

A second method is provided allowing to dump the continuation values of the problem and the optimal control at each time step:

```

1 void dumpContinuationValues(std::shared_ptr<gs::BinaryFileArchive> p_ar,
2                          const std::string &p_name, const int &p_iStep,
3                          const std::vector< std::shared_ptr< Eigen::ArrayXXd > > &
4                          p_phiIn,
5                          const std::vector< std::shared_ptr< Eigen::ArrayXXd > > &
6                          p_control,
7                          const std::shared_ptr< Tree> &p_tree,
8                          const bool &p_bOneFile) const;

```

with:

- `p_ar` is the archive where the controls and solutions are dumped,
- `p_name` is a base name used in the archive to store the solution and the control,
- `p_phiInPrev` is the previous time step solution used to calculate the continuation values that are stored,
- `p_control` stores the optimal controls calculated at the current time step,
- `p_tree` is the conditional expectation object allowing to calculate the conditional expectation of the functions defined at the previous time step processed `p_phiInPrev` and allowing to store a representation of the optimal control.
- `p_bOneFile` is set to one if the continuation and optimal controls calculated by each processor are dumped on a single file. Otherwise the continuation and optimal controls calculated by each processor are dumped on different files (one per processor).

Remark 17 *The `p_bOneFile` is not present for `TransitionStepTreeDP` objects.*

In simulation (see detail in section for regressions)

- A first object allowing the recalculation of the optimal control in simulation.

```

1 SimulateStepTreeDist(gs::BinaryFileArchive &p_ar,  const int &p_iStep,  const std::
   string &p_nameCont,
2                               const std::shared_ptr<FullGrid> &p_pGridFollowing, const
   std::shared_ptr<OptimizerDPTreeBase > &p_pOptimize,
3                               const bool &p_bOneFile,
4                               const boost::mpi::communicator &p_world)

```

where

- `p_ar` is the binary archive where the continuation values are stored,
- `p_iStep` is the number associated with the current time step (0 at the start date of the simulation, the number is increased by one at each simulated time step),
- `p_nameCont` is the base name for continuation values,
- `p_GridFollowing` is the grid at the next time step (`p_iStep+1`),
- `p_Optimize` the Optimizer describing the transition from one time step to the next,
- `p_OneFile` equal to `true` if only one archive is used to store continuation values.
- `p_world` The MPI communicator

This object implements the method `oneStep`

```

1 void oneStep(std::vector<StateTreeStocks > &p_statevector, Eigen::ArrayXXd &
   p_phiInOut) const

```

where:

- `p_statevector` stores the states of all the simulations: this state is updated by applying the optimal command,
- `p_phiInOut` stores the gain/cost functions for all simulations: it is updated by the function call. The size of the array is $(nb, nbSimul)$ where nb is given by the `getSimuFuncSize` method of the optimizer and $nbSimul$ the number of Monte Carlo simulations.

Remark 18 *The version without distribution of Bellman values is available in the object `SimulateStepTree`.*

- A second object directly interpolating the control

```

1 SimulateStepTreeControlDist(gs::BinaryFileArchive &p_ar,  const int &p_iStep,  const
   std::string &p_nameCont,
2                               const std::shared_ptr<FullGrid> &p_pGridCurrent,
3                               const std::shared_ptr<FullGrid> &p_pGridFollowing,
4                               const std::shared_ptr<OptimizerDPTreeBase > &
   p_pOptimize,
5                               const bool &p_bOneFile,
6                               const boost::mpi::communicator &p_world);

```

where

- `p_ar` is the binary archive where the continuation values are stored,
- `p_iStep` is the number associated with the current time step (0 at the start date of the simulation, the number is increased by one at each simulated time step),
- `p_nameCont` is the base name of the control values,
- `p_GridCurrent` is the grid at the current time step (`p_iStep`),
- `p_GridFollowing` is the grid at the next time step (`p_iStep+1`),
- `p_Optimize` is the Optimizer describing the transition from one time step to the next,
- `p_OneFile` is equal to `true` if only one archive is used to store continuation values.
- `p_world` The MPI communicator

This object implements the method `oneStep`

```

1 void oneStep(std::vector<StateTreeStocks > &p_statevector, Eigen::ArrayXXd &
   p_phiInOut) const;

```

where:

- `p_statevector` stores the state for all the simulations : this state is updated by applying optimal commands,
- `p_phiInOut` stores the gain/cost functions for all simulations: it is updated by the function call. The size of the array is $(nb, nbSimul)$ where nb is given by the `getSimuFuncSize` method of the optimizer and $nbSimul$ the number of Monte Carlo simulations.

Remark 19 *The undistributed version is given by the object `SimulateStepTreeControl`.*

3.4.2 Solving Dynamic Programming by solving LP problems

This approach can only be used for continuous problems with convex or concave Bellman values. See section 4.2.2 for an explanation of the cut approximation.

Framework usage requirement

The expanded business object must derive from the `OptimizerDPCutBase` object derived from the `OptimizerBase` object.

```
1 // Copyright (C) 2019 EDF
2 // All Rights Reserved
3 // This code is published under the GNU Lesser General Public License (GNU LGPL)
4 #ifndef OPTIMIZERDPCUTTREEBASE_H
5 #define OPTIMIZERDPCUTTREEBASE_H
6 #include <Eigen/Dense>
7 #include "StOpt/core/utils/StateWithStocks.h"
8 #include "StOpt/core/grids/SpaceGrid.h"
9 #include "StOpt/tree/Tree.h"
10 #include "StOpt/tree/StateTreeStocks.h"
11 #include "StOpt/tree/ContinuationCutsTree.h"
12 #include "StOpt/dp/SimulatorDPBaseTree.h"
13 #include "StOpt/dp/OptimizerBase.h"
14
15 /** \file OptimizerDPCutBase.h
16  * \brief Define an abstract class for Dynamic Programming problems solved by tree
17  *        methods using cust to approximate
18  *        Bellman values
19  *        \author Xavier Warin
20  */
21 namespace StOpt
22 {
23
24     /// \class OptimizerDPCutTreeBase OptimizerDPCutTreeBase.h
25     /// Base class for optimizer for Dynamic Programming with tree methods and cuts, so using
26     /// LP to solve transitional problems
27     class OptimizerDPCutTreeBase : public OptimizerBase
28     {
29     public :
30
31         OptimizerDPCutTreeBase() {}
32
33         virtual ~OptimizerDPCutTreeBase() {}
34
35         /// \brief defines the diffusion cone for parallelism
36         /// \param p_regionByProcessor region (min max) treated by the processor for
37         ///        the different regimes treated
38         /// \return returns in each dimension the min max values in the stock that can be
39         ///        reached from the grid p_gridByProcessor for each regime
40         virtual std::vector< std::array< double, 2> > getCone(const std::vector< std::array<
41         double, 2> > &p_regionByProcessor) const = 0;
42
43         /// \brief defines the dimension to split for MPI parallelism
44         /// For each dimension return true is the direction can be split
45         virtual Eigen::Array< bool, Eigen::Dynamic, 1> getDimensionToSplit() const = 0 ;
46
47         /// \brief defines a step in optimization
48         /// \param p_grid grid at arrival step after command
49         /// \param p_stock coordinates of the stock point to treat
50         /// \param p_condEsp continuation values for each regime
51         /// \return For each regimes (column) gives the solution for each particle , and cut
52         ///        (row)
```

```

50  ///      For a given simulation , cuts components (C) at a point stock \S \bar S
51  ///      \f$ are given such that the cut is given by
52  ///      \f$ C[0] + \sum_{i=1}^d C[i] (S_i - \bar S_i) \f$
53  virtual Eigen::ArrayXXd stepOptimize(const std::shared_ptr< StOpt::SpaceGrid> &
54  p_grid, const Eigen::ArrayXd &p_stock,
55  const std::vector< StOpt::ContinuationCutsTree
56  > &p_condEsp) const = 0;
57
58  /// \brief defines a step in simulation
59  /// Control are recalculated during simulation using a local optimization using the LP
60  /// \param p_grid grid at arrival step after command
61  /// \param p_continuation defines the continuation operator for each regime
62  /// \param p_state defines the state value (modified)
63  /// \param p_phiInOut defines the value functions (modified) : size number of
64  /// functions to follow
65  virtual void stepSimulate(const std::shared_ptr< StOpt::SpaceGrid> &p_grid, const std
66  ::vector< StOpt::ContinuationCutsTree > &p_continuation,
67  StOpt::StateTreeStocks &p_state,
68  Eigen::Ref<Eigen::ArrayXd> p_phiInOut) const = 0 ;
69
70  /// \brief Get the number of regimes allowed for the asset to be reached at the
71  /// current time step
72  /// If \f$ t \f$ is the current time, and \f$ dt \f$ the resolution step, this is
73  /// the number of regime allowed on \f$[ t- dt, t[\f$
74  virtual int getNbRegime() const = 0 ;
75
76  /// \brief get the simulator back
77  virtual std::shared_ptr< StOpt::SimulatorDPBaseTree > getSimulator() const = 0;
78
79  /// \brief get back the dimension of the control
80  virtual int getNbControl() const = 0 ;
81
82  /// \brief get size of the function to follow in simulation
83  virtual int getSimuFuncSize() const = 0;
84
85 };
86 }
87 #endif /* OPTIMIZERDPCUTTREEBASE_H */

```

We detail the different methods to implement in addition to the `OptimizerBase` methods:

- the `stepOptimize` method is used in optimization. We want to calculate the optimal value and the corresponding sensitivities with respect to stocks at t_i current at a grid point `p_stock` using a grid `p_grid` at the next date t_{i+1} , the continuation cuts the values for all the regimes `p_condEsp` allowing to calculate a upper estimate (during the maximization) of the conditional expectation of the optimal values by using an optimization calculated at the previously treated time step t_{i+1} . From a `p_stock` grid point, it calculates the function values and the corresponding sensitivities. It returns a matrix (the first dimension is the number of nodes at the current cate by the number of components cuts (number of storage +1), the second dimension the number of regimes) giving the value of the function and sensitivities.
- the `stepSimulate` method is used after the optimization using the continuation cuts the values calculated in the optimization part. From a `p_state` state (storing the $X^{x,t}$), the sequence cuts the calculated values in optimization `p_continuation`, the optimal cash flows are stored in `p_phiInOut`.

In the case of a gas storage the `Optimize` object is given in the file `OptimizeGasStorageTreeCut.h`.

The framework in optimization using certain cutting methods

We treat it exactly as in 4.2.2 section. The framework provides an `TransitionStepTreeDPCutDist` object in MPI which solves the optimization problem with data distribution over a time step with the following constructor:

```
1  TransitionStepTreeDPCutDist(const std::shared_ptr<FullGrid> &p_pGridCurrent,
2                               const std::shared_ptr<FullGrid> &p_pGridPrevious,
3                               const std::shared_ptr<OptimizerDPCutTreeBase> &
4                               p_pOptimize,
                               const boost::mpi::communicator &p_world);
```

with

- `p_pGridCurrent` is the grid at the current time step (t_i),
- `p_pGridPrevious` is the grid at the previously processed time step (t_{i+1}),
- `p_pOptimize` the optimizer object
- `p_world` The MPI communicator

The construction is very similar to the classical regression methods using only the discretization by command.

Remark 20 *A similar object is available without the MPI distribution framework `TransitionStepTreeDPCut` with always the activation of parallelization with threads and MPI on calculations on the full points grid .*

The main method is

```
1  std::vector< std::shared_ptr< Eigen::ArrayXXd > > oneStep(const std::vector< std:::
2  shared_ptr< Eigen::ArrayXXd > > &p_phiIn,
   const std::shared_ptr< Tree> &p_condExp) const
```

with

- `p_phiIn` the vector (its size corresponds to the number of regimes) of the matrix of optimal values and sensitivities calculated at the previous time iteration for each regime. Each matrix has a number of rows equal to the number of nodes on the next date by the number of stock plus one. The number of columns is equal to the number of stock points on the grid. In the row, the number of simulations by the number of stock plus one value is stored as follows:
 - The first values (number of nodes on the following date: NS) correspond to the optimal Bellman values at a given stock point,
 - The NS values following corresponds to sensitivities $\frac{\partial V}{\partial S_1}$ to first storage
 - The NS values following corresponds to sensitivities to the second storage...
 - ...
- `p_condExp` the conditional expectation operator,

returning a vector of matrix with new optimal values and sensitivities at the current time step (each element of the vector corresponds to a regime and each matrix has a size equal to (number of nodes at the current date by (the number of storage plus one)) by the number of stock points). The structure of the output is then similar to the `p_phiIn` input.

A second method is provided allowing to dump the continuation values and the cuts of the problem and the optimal control at each time step:

```

1 void dumpContinuationCutsValues(std::shared_ptr<gs::BinaryFileArchive> p_ar, const std
  ::string &p_name, const int &p_iStep,
2                               const std::vector< std::shared_ptr< Eigen::ArrayXXd > >
  &p_phiInPrev, const std::shared_ptr< Tree> &
3                               p_condExp,
  const bool &p_bOneFile) const

```

with:

- `p_ar` is the archive where controls and solutions are dumped,
- `p_name` is a base name used in the archive to store the solution and the control,
- `p_phiInPrev` is the solution to the previous time step used to calculate the values of continuation cuts which are stored,
- `p_condExp` is the conditional expectation object allowing to calculate the conditional expectation of the functions defined at the preceding time step processed `p_phiInPrev`.
- `p_bOneFile` is set to one if the continuation cuts values calculated by each processor are dumped on a single file. Otherwise the continuation cuts values calculated by each processor are dumped on different files (one per processor).

Here we give a simple example of a temporal resolution using this method when MPI data distribution is used

```

1 // Copyright (C) 2019 EDF
2 // All Rights Reserved
3 // This code is published under the GNU Lesser General Public License (GNU LGPL)
4 #ifdef USE_MPI
5 #include <fstream>
6 #include <memory>
7 #include <functional>
8 #include <boost/lexical_cast.hpp>
9 #include <boost/mpi.hpp>
10 #include <Eigen/Dense>
11 #include "generators/BinaryFileArchive.hh"
12 #include "StOpt/core/grids/FullGrid.h"
13 #include "StOpt/tree/Tree.h"
14 #include "StOpt/dp/FinalStepDPCutDist.h"
15 #include "StOpt/dp/TransitionStepTreeDPCutDist.h"
16 #include "StOpt/core/parallelism/reconstructProcOMpi.h"
17 #include "StOpt/dp/OptimizerDPCutTreeBase.h"
18 #include "StOpt/dp/SimulatorDPBaseTree.h"
19
20
21 using namespace std;
22 using namespace Eigen;
23
24 double DynamicProgrammingByTreeCutDist(const shared_ptr<StOpt::FullGrid> &p_grid,
25                                       const shared_ptr<StOpt::OptimizerDPCutTreeBase> &
26                                       p_optimize,
27                                       const function< ArrayXd(const int &, const ArrayXd
28                                       &, const ArrayXd &>> &p_funcFinalValue,

```

```

27         const ArrayXd &p_pointStock,
28         const int &p_initialRegime,
29         const string &p_fileToDump,
30         const bool &p_bOneFile,
31         const boost::mpi::communicator &p_world)
32 {
33     // from the optimizer get back the simulator
34     shared_ptr< StOpt::SimulatorDPBaseTree> simulator = p_optimize->getSimulator();
35     // final values
36     vector< shared_ptr< ArrayXXd > > valueCutsNext = StOpt::FinalStepDPCutDist(p_grid,
37         p_optimize->getNbRegime(), p_optimize->getDimensionToSplit(), p_world)(
38         p_funcFinalValue, simulator->getNodes());
39     // dump
40     string toDump = p_fileToDump ;
41     // test if one file generated
42     if (!p_bOneFile)
43         toDump += "_" + boost::lexical_cast<string>(p_world.rank());
44     shared_ptr<gs::BinaryFileArchive> ar;
45     if ((!p_bOneFile) || (p_world.rank() == 0))
46         ar = make_shared<gs::BinaryFileArchive>(toDump.c_str(), "w");
47     // name for object in archive
48     string nameAr = "ContinuationTree";
49     for (int iStep = 0; iStep < simulator->getNbStep(); ++iStep)
50     {
51         simulator->stepBackward();
52         // probabilities
53         std::vector<double> proba = simulator->getProba();
54         // get connection between nodes
55         std::vector< std::vector<int, 2> > > connected = simulator->
56             getConnected();
57         // conditional expectation operator
58         shared_ptr<StOpt::Tree> tree = std::make_shared<StOpt::Tree>(proba, connected);
59         // transition object
60         StOpt::TransitionStepTreeDPCutDist transStep(p_grid, p_grid, p_optimize, p_world);
61         vector< shared_ptr< ArrayXXd > > valueCuts = transStep.oneStep(valueCutsNext, tree
62             );
63         transStep.dumpContinuationCutsValues(ar, nameAr, iStep, valueCutsNext, tree,
64             p_bOneFile);
65         valueCutsNext = valueCuts;
66     }
67     // reconstruct a small grid for interpolation
68     ArrayXd valSim = StOpt::reconstructProcOMpi(p_pointStock, p_grid, valueCutsNext[
69         p_initialRegime], p_optimize->getDimensionToSplit(), p_world);
70     return ((p_world.rank() == 0) ? valSim(0) : 0.);
71 }
72 #endif

```

An example without data distribution can be found in the file `DynamicProgrammingByTreeCut.cpp`.

Using the framework in simulation

Similar to the 4.2.2 section, we can use the cuts calculated in optimization to test the optimal strategy found. In order to simulate an optimal policy step, a `SimulateStepTreeCutDist` object is provided with the constructor

```

1     SimulateStepTreeCutDist(gs::BinaryFileArchive &p_ar, const int &p_iStep, const std::
2         string &p_nameCont,
3         const std::shared_ptr<FullGrid> &p_pGridFollowing,
4         const std::shared_ptr<OptimizerDPCutTreeBase > &p_pOptimize,
5         const bool &p_bOneFile,
6         const boost::mpi::communicator &p_world);

```

where

- `p_ar` is the binary archive where the continuation values are stored,
- `p_iStep` is the number associated with the current time step (0 at the start date of the simulation, the number is increased by one at each simulated time step),
- `p_nameCont` is the base name for continuation values,
- `p_GridFollowing` is the grid at the next time step (`p_iStep+1`),
- `p_Optimize` the Optimizer describing the transition problem solved using an LP program.
- `p_OneFile` equal to `true` if only one archive is used to store continuation values.
- `p_world` The MPI communicator

Remark 21 *A version without data distribution but with multithreaded and with possible MPI on the calculations is available with the `SimulateStepTreeCut` object. The `p_OneFile` argument is omitted during construction.*

This object implements the method `oneStep`

```
1 void oneStep(std::vector<StateTreeStocks > &p_statevector, Eigen::ArrayXXd &p_phiInOut)
   const
```

where:

- `p_statevector` stores the states of all the simulations: this state is updated by applying the optimal command,
- `p_phiInOut` stores the gain/cost functions for all simulations: it is updated by the function call. The size of the array is $(nb, nbSimul)$ where nb is given by the `get SimuFuncSize` method of the optimizer and $nbSimul$ the number of Monte Carlo simulations.

An example of using this method to simulate an optimal policy with distribution is given below:

```
1 // Copyright (C) 2019 EDF
2 // All Rights Reserved
3 // This code is published under the GNU Lesser General Public License (GNU LGPL)
4 #ifndef SIMULATEREGTRECUTDIST_H
5 #define SIMULATEREGTRECUTDIST_H
6 #include <functional>
7 #include <memory>
8 #include <Eigen/Dense>
9 #include <boost/mpi.hpp>
10 #include "geners/BinaryFileArchive.hh"
11 #include "StOpt/core/grids/FullGrid.h"
12 #include "StOpt/tree/StateTreeStocks.h"
13 #include "StOpt/dp/SimulateStepTreeCutDist.h"
14 #include "StOpt/dp/OptimizerDPCutBase.h"
15 #include "StOpt/dp/SimulatorDPBase.h"
16
17
18 /** \file SimulateTreeCutDist.h
19  * \brief Defines a simple program showing how to use simulations when optimization achieved
20  * with transition problems solved with cuts and uncertainties on a tree
```

```

20 *          A simple grid is used
21 * \author Xavier Warin
22 */
23
24
25 /// \brief Simulate the optimal strategy , mpi version, Bellman cuts used to allow LP
26 /// resolution of transition problems when uncertainties are defined on a tree
27 /// \param p_grid          grid used for deterministic state (stocks for example)
28 /// \param p_optimize      optimizer defining the optimization between two time
29 /// steps
30 /// \param p_funcFinalValue function defining the final value cuts
31 /// \param p_pointStock    initial point stock
32 /// \param p_initialRegime regime at initial date
33 /// \param p_fileToDump    name associated to dumped bellman values
34 /// \param p_bOneFile      do we store continuation values in only one file
35 /// \param p_world         MPI communicator
36 double SimulateTreeCutDist(const std::shared_ptr<StOpt::FullGrid> &p_grid,
37                           const std::shared_ptr<StOpt::OptimizerDPCutTreeBase > &
38                             p_optimize,
39                           const std::function< Eigen::ArrayXd(const int &, const Eigen::
40                             ArrayXd &, const Eigen::ArrayXd &)> &p_funcFinalValue,
41                           const Eigen::ArrayXd &p_pointStock,
42                           const int &p_initialRegime,
43                           const std::string &p_fileToDump,
44                           const bool &p_bOneFile,
45                           const boost::mpi::communicator &p_world)
46 {
47     // from the optimizer get back the simulator
48     std::shared_ptr< StOpt::SimulatorDPBaseTree> simulator = p_optimize->getSimulator();
49     int nbStep = simulator->getNbStep();
50     std::vector< StOpt::StateTreeStocks> states;
51     states.reserve(simulator->getNbSimul());
52     for (int is = 0; is < simulator->getNbSimul(); ++is)
53         states.push_back(StOpt::StateTreeStocks(p_initialRegime, p_pointStock, 0));
54     std::string toDump = p_fileToDump ;
55     // test if one file generated
56     if (!p_bOneFile)
57         toDump += "_" + boost::lexical_cast<std::string>(p_world.rank());
58     gs::BinaryFileArchive ar(toDump.c_str(), "r");
59     // name for continuation object in archive
60     std::string nameAr = "ContinuationTree";
61     // cost function
62     Eigen::ArrayXXd costFunction = Eigen::ArrayXXd::Zero(p_optimize->getSimuFuncSize(),
63         simulator->getNbSimul());
64     for (int istep = 0; istep < nbStep; ++istep)
65     {
66         StOpt::SimulateStepTreeCutDist(ar, nbStep - 1 - istep, nameAr, p_grid, p_optimize,
67             p_bOneFile, p_world).oneStep(states, costFunction);
68
69         // new date
70         simulator->stepForward();
71         for (int is = 0; is < simulator->getNbSimul(); ++is)
72             states[is].setStochasticRealization(simulator->getNodeAssociatedToSim(is));
73     }
74     // final : accept to exercise if not already done entirely (here suppose one function
75     // to follow)
76     for (int is = 0; is < simulator->getNbSimul(); ++is)
77         costFunction(0, is) += p_funcFinalValue(states[is].getRegime(), states[is].
78             getPtStock(), simulator->getValueAssociatedToNode(states[is].
79                 getStochasticRealization()))(0);
80
81     return costFunction.mean();
82 }
83
84 #endif /* SIMULATETREECUTDIST_H */

```

The version of the previous example using a single archive storing the control/solution is given in `SimulateTreeCut.h` file.

3.5 The Python API to manage stocks

3.5.1 Mapping to the framework

To use the Python API, it is possible to use only the mapping of grids, continuation values, and regression object and program an equivalent of `TransitionStepRegressionDP` and of `SimulateStepRegression`, `SimulateStepRegressionControl` in Python. No mapping is currently available for `TransitionStepDP`. An example using Python is given by

```
1 # Copyright (C) 2016 EDF
2 # All Rights Reserved
3 # This code is published under the GNU Lesser General Public License (GNU LGPL)
4 import numpy as np
5 import pystopt.regression as reg
6
7 class TransitionStepRegressionDP:
8
9     def __init__(self, p_pGridCurrent, p_pGridPrevious, p_pOptimize):
10
11         self.m_pGridCurrent = p_pGridCurrent
12         self.m_pGridPrevious = p_pGridPrevious
13         self.m_pOptimize = p_pOptimize
14
15     def oneStep(self, p_phiIn, p_condExp):
16
17         nbRegimes = self.m_pOptimize.getNbRegime()
18         phiOut = list(range(nbRegimes))
19         nbControl = self.m_pOptimize.getNbControl()
20         controlOut = list(range(nbControl))
21
22         # only if the processor is working
23         if self.m_pGridCurrent.getNbPoints() > 0:
24
25             # allocate for solution
26             for iReg in range(nbRegimes):
27                 phiOut[iReg] = np.zeros((p_condExp.getNbSimul(), self.m_pGridCurrent.
28                                         getNbPoints()))
29
30             for iCont in range(nbControl):
31                 controlOut[iCont] = np.zeros((p_condExp.getNbSimul(), self.m_pGridCurrent.
32                                         getNbPoints()))
33
34             # number of threads
35             nbThreads = 1
36
37             contVal = []
38
39             for iReg in range(len(p_phiIn)):
40                 contVal.append(reg.ContinuationValue(self.m_pGridPrevious, p_condExp,
41                                                     p_phiIn[iReg]))
42
43             # create iterator on current grid treated for processor
44             iterGridPoint = self.m_pGridCurrent.getGridIteratorInc(0)
45
46             # iterates on points of the grid
47             for iIter in range(self.m_pGridCurrent.getNbPoints()):
48
49                 if iterGridPoint.isValid():
50                     pointCoord = iterGridPoint.getCoordinate()
51                     # optimize the current point and the set of regimes
52                     solutionAndControl = self.m_pOptimize.stepOptimize(self.m_pGridPrevious
53                                     , pointCoord, contVal, p_phiIn)
54
55             # copy solution
56             for iReg in range(self.m_pOptimize.getNbRegime()):
57                 phiOut[iReg][:, iterGridPoint.getCount()] = solutionAndControl[0][:,
```

```

54         iReg]
55         for iCont in range(nbControl):
56             controlOut[iCont][:,iterGridPoint.getCount()] = solutionAndControl
57                 [1][:,iCont]
58             iterGridPoint.nextInc(nbThreads)
59
60     res = []
61     res.append(phiOut)
62     res.append(controlOut)
63     return res

```

Some examples are available in the test directory (for example for swing options).

Another approach more effective in term of computational cost consists in mapping the simulator object derived from the `SimulatorDPBase` object and optimizer object derived from the `OptimizerDPBase` object and to use the high level Python mapping of `TransitionStepRegressionDP` and `SimulateStepRegression`. In the test part of the library some Black-Scholes simulator and some Mean reverting simulator for a future curve deformation are developed and some examples of the mapping are achieved in the `Pybind11Simulators.cpp` file. Likewise, optimizer class for swings options, optimizer for a fictitious swing in dimension 2, optimizer for a gas storage, optimizer for a gas storage with switching cost are mapped to Python in the `Pybind11Optimizers.cpp` file.

In the example below, we describe the use of this high level interface for swing options with a Black-Scholes simulator: in this example we give the mapping of the most used objects:

```

1  # Copyright (C) 2016 EDF
2  # All Rights Reserved
3  # This code is published under the GNU Lesser General Public License (GNU LGPL)
4  import math
5  import importlib.util
6  import numpy as np
7  import unittest
8  import pystopt.grids as StOptGrids
9  import pystopt.regression as StOptReg
10 import pystopt.generators as StOptGenerators
11 import pystopt.glob as StOptGlobal
12 import pystopttest.utils as Utils
13 import pystopttest.simulators as sim
14 import pystopttest.optimizers as opt
15
16 # unit test for global shape
17 #####
18
19 class OptimizerConstruction(unittest.TestCase):
20
21     def test(self):
22         # test MPI
23         moduleMpi4Py=importlib.util.find_spec('mpi4py')
24         if (moduleMpi4Py is not None):
25             from mpi4py import MPI
26             initialValues = np.zeros(1,dtype=float) + 1.
27             sigma = np.zeros(1) + 0.2
28             mu = np.zeros(1) + 0.05
29             corr = np.ones((1,1),dtype=float)
30             # number of step
31             nStep = 30
32             # exercise dates
33             dates = np.linspace(0., 1., nStep + 1)
34             T= dates[len(dates) - 1]
35             nbSimul = 10 # simulation number (optimization and simulation)
36             # simulator
37             #####

```

```

38     bsSim = sim.BlackScholesSimulator(initialValues, sigma, mu, corr, T, len(dates) -
39         1, nbSimul, False)
40     strike = 1.
41     # Pay off
42     payOff= Utils.BasketCall(strike)
43     # optimizer
44     #####
45     N = 3 # number of exercise dates
46     swiOpt = opt.OptimizerSwingBlackScholes(payOff,N)
47     # link simulator to optimizer
48     swiOpt.setSimulator(bsSim)
49     # archive
50     #####
51     ar = StOptGeners.BinaryFileArchive("Archive","w")
52     # regressor
53     #####
54     nMesh = np.array([1])
55     regressor = StOptReg.LocalLinearRegression(nMesh)
56     # Grid
57     #####
58     # low value for the meshes
59     lowValues =np.array([0.],dtype=float)
60     # size of the meshes
61     step = np.array([1.],dtype=float)
62     # number of steps
63     nbStep = np.array([N], dtype=np.int32)
64     gridArrival = StOptGrids.RegularSpaceGrid(lowValues,step,nbStep)
65     gridStart = StOptGrids.RegularSpaceGrid(lowValues,step,nbStep-1)
66     # pay off function for swing
67     #####
68     payOffBasket = Utils.BasketCall(strike);
69     payoff = Utils.PayOffSwing(payOffBasket,N)
70     dir(payoff)
71     # final step
72     #####
73     asset =bsSim.getParticles()
74     fin = StOptGlobal.FinalStepDP(gridArrival,1)
75     values = fin.set( payoff,asset)
76     # transition time step
77     #####
78     # on step backward and get asset
79     asset = bsSim.stepBackwardAndGetParticles()
80     # update regressor
81     regressor.updateSimulations(0,asset)
82     transStep = StOptGlobal.TransitionStepRegressionDP(gridStart,gridArrival,swiOpt)
83     valuesNextAndControl=transStep.oneStep(values,regressor)
84     transStep.dumpContinuationValues(ar,"Continuation",1,valuesNextAndControl[0],
85         valuesNextAndControl[1],regressor)
86     # simulate time step
87     #####
88     nbSimul= 10
89     vecOfStates =[] # state of each simulation
90     for i in np.arange(nbSimul):
91         # one regime, all with same stock level (dimension 2), same realization of
92         # simulation (dimension 3)
93         vecOfStates.append(StOptGlobal.StateWithStocks(1, np.array([0.]) , np.zeros(1))
94             )
95         arRead = StOptGeners.BinaryFileArchive("Archive","r")
96         simStep = StOptGlobal.SimulateStepRegression(arRead,1,"Continuation",gridArrival,
97             swiOpt)
98         phi = np.zeros((1,nbSimul))
99         VecOfStateNext = simStep.oneStep(vecOfStates, phi)
100
101 if __name__ == '__main__':
102     unittest.main()

```

Its variation in terms of timing loop for optimization is given below (note that the object TransitionStepRegressionDP is the result of the mapping between Python and c++ and

given in the module StOptGlobal)

```
1 # Copyright (C) 2016 EDF
2 # All Rights Reserved
3 # This code is published under the GNU Lesser General Public License (GNU LGPL)
4 import pystopt.grids as StOptGrids
5 import pystopt.regression as StOptReg
6 import pystopt.generators as StOptGenerators
7 import pystopt.glob as StOptGlobal
8
9
10 def DynamicProgrammingByRegressionHighLevel(p_grid, p_optimize, p_regressor,
11      p_funcFinalValue, p_pointStock, p_initialRegime, p_fileToDump) :
12
13     # from the optimizer get back the simulation
14     simulator = p_optimize.getSimulator()
15     # final values
16     fin = StOptGlobal.FinalStepDP(p_grid, p_optimize.getNbRegime())
17     valuesNext = fin.set(p_funcFinalValue, simulator.getParticles())
18     ar = StOptGenerators.BinaryFileArchive(p_fileToDump, "w")
19     nameAr = "Continuation"
20     nsteps = simulator.getNbStep()
21     # iterate on time steps
22     for iStep in range(nsteps) :
23         asset = simulator.stepBackwardAndGetParticles()
24         # conditional expectation operator
25         if iStep == (simulator.getNbStep() - 1):
26             p_regressor.updateSimulations(True, asset)
27         else:
28             p_regressor.updateSimulations(False, asset)
29
30     # transition object
31     transStep = StOptGlobal.TransitionStepRegressionDP(p_grid, p_grid, p_optimize)
32     valuesAndControl = transStep.oneStep(valuesNext, p_regressor)
33     transStep.dumpContinuationValues(ar, nameAr, nsteps - 1 - iStep, valuesNext,
34         valuesAndControl[1], p_regressor)
35     valuesNext = valuesAndControl[0]
36
37     # interpolate at the initial stock point and initial regime
38     return (p_grid.createInterpolator(p_pointStock).applyVec(valuesNext[p_initialRegime])).
39         mean()
```

Likewise, a Python temporal loop in simulation using the control previously calculated in optimization can be given as an example by:

```
1 # Copyright (C) 2016 EDF
2 # All Rights Reserved
3 # This code is published under the GNU Lesser General Public License (GNU LGPL)
4 import numpy as np
5 import pystopt.grids as StOptGrids
6 import pystopt.regression as StOptReg
7 import pystopt.generators as StOptGenerators
8 import pystopt.glob as StOptGlobal
9
10
11 # Simulate the optimal strategy , threaded version
12 # p_grid                grid used for deterministic state (stocks for example)
13 # p_optimize            optimizer defining the optimization between two time steps
14 # p_funcFinalValue      function defining the final value
15 # p_pointStock          initial point stock
16 # p_initialRegime       regime at initial date
17 # p_fileToDump          name of the file used to dump continuation values in
18 #                       optimization
19 def SimulateRegressionControl(p_grid, p_optimize, p_funcFinalValue, p_pointStock,
20     p_initialRegime, p_fileToDump) :
21
22     simulator = p_optimize.getSimulator()
23     nbStep = simulator.getNbStep()
```

```

22     states = []
23     particle0 = simulator.getParticles()[0,0]
24
25     for i in range(simulator.getNbSimul()) :
26         states.append(StOptGlobal.StateWithStocks(p_initialRegime, p_pointStock, particle0)
27                     )
28
29     ar = StOptGeners.BinaryFileArchive(p_fileToDump, "r")
30     # name for continuation object in archive
31     nameAr = "Continuation"
32     # cost function
33     costFunction = np.zeros((p_optimize.getSimuFuncSize(), simulator.getNbSimul()))
34
35     # iterate on time steps
36     for istep in range(nbStep) :
37         NewState = StOptGlobal.SimulateStepRegressionControl(ar, istep, nameAr, p_grid,
38                     p_optimize).oneStep(states, costFunction)
39         # different from C++
40         states = NewState[0]
41         costFunction = NewState[1]
42         # new stochastic state
43         particles = simulator.stepForwardAndGetParticles()
44
45         for i in range(simulator.getNbSimul()) :
46             states[i].setStochasticRealization(particles[:,i])
47
48     # final : accept to exercise if not already done entirely
49     for i in range(simulator.getNbSimul()) :
50         costFunction[0,i] += p_funcFinalValue.set(states[i].getRegime(), states[i].
51             getPtStock(), states[i].getStochasticRealization()) * simulator.getActu()
52
53     # average gain/cost
54     return costFunction.mean()

```

The equivalent of using MPI and the distributing calculations and data can be used using the `mpi4py` package. An example of its use can be found in the MPI version of a swing optimization and valuation.

3.6 Special Python binding

Some specific features have been added to the Python interface to increase the flexibility of the library. A special mapping of the `gens` library was carried out for certain specific needs.

3.6.1 A first binding to use the framework

The `BinaryFileArchive` in the Python module `StOptGeners` permits for:

- a grid on point,
- a list of numpy array (dimension 2) of size the number of simulations used by the number of points on the grid (the size of the list corresponds to the number of regimes used in the event of a regime switching problem: if a regime, this list contains only one item which is a two-dimensional array)
- a regressor

to create a regressed set of values from numpy arrays and store them in the archive. This feature allows you to store the continuation values associated with a problem.

The `dumpGridAndRegressedValue` dump method in the `BinaryFileArchive` class permits this dump.

It is also possible to retrieve the continuation values obtained using the `readGridAndRegressedValue` method.

```

1  # Copyright (C) 2016 EDF
2  # All Rights Reserved
3  # This code is published under the GNU Lesser General Public License (GNU LGPL)
4  import numpy as np
5  import unittest
6  import random
7  import pystopt.grids as StOptGrids
8  import pystopt.regression as StOptReg
9  import pystopt.generators as StOptGenerators
10
11
12  # unit test for dumping binary archive of regressed value and Read then
13  #####
14
15  class testBinaryArchiveStOpt(unittest.TestCase):
16
17
18      def testSimpleStorageAndLectureRecGrid(self):
19
20          # low value for the mesh
21          lowValues = np.array([1.,2.,3.],dtype=np.float64)
22          # size of the mesh
23          step = np.array([0.7,2.3,1.9],dtype=np.float64)
24          # number of step
25          nbStep = np.array([4,5,6], dtype=np.int32)
26          # degree of the polynomial in each direction
27          degree = np.array([2,1,3], dtype=np.int32)
28          # create the Legendre grid
29          grid = StOptGrids.RegularLegendreGrid(lowValues,step,nbStep,degree )
30
31
32          # simulate the perburbed values
33          #####
34          nbSimul =40000
35          np.random.seed(1000)
36          x = np.random.uniform(-1.,1.,size=(1,nbSimul));
37          # mesh
38          nbMesh = np.array([16],dtype=np.int32)
39          # Create the regressor
40          #####
41          regressor = StOptReg.LocalLinearRegression(False,x,nbMesh)
42
43          # regressed values same values for each point of the grid
44          #####
45          toReal = (2+x[0,:]+(1+x[0,:])*(1+x[0,:]))
46          # function to regress
47          toRegress = toReal + 4*np.random.normal(0.,1,nbSimul)
48          # create a matrix (number of stock points by number of simulations)
49          toRegressMult = np.zeros(shape=(len(toRegress),grid.getNbPoints()))
50          for i in range(toRegressMult.shape[1]):
51              toRegressMult[:,i] = toRegress
52          # into a list : two times to test 2 regimes
53          listToReg = []
54          listToReg.append(toRegressMult)
55          listToReg.append(toRegressMult)
56
57
58          # Create the binary archive to dump
59          #####

```

```

60     archiveToWrite = StOptGens.BinaryFileArchive("MyArchive","w")
61     # step 1
62     archiveToWrite.dumpGridAndRegressedValue("toStore", 1,listToReg, regressor,grid)
63     # step 3
64     archiveToWrite.dumpGridAndRegressedValue("toStore", 3,listToReg, regressor,grid)
65
66
67     # Read the regressed values
68     #####
69     archiveToRead = StOptGens.BinaryFileArchive("MyArchive","r")
70     contValues = archiveToRead.readGridAndRegressedValue(3,"toStore")
71
72
73     # list of 2 continuation values
74     #####
75     stockPoint = np.array([1.5,3.,5.])
76     uncertainty = np.array([0.])
77     value =contValues[0].getValue(stockPoint,uncertainty)
78
79
80     # non regular grid
81     def testSimpleStorageAndLectureNonRegular(self):
82
83         # create the Legendre grid
84         grid = StOptGrids.GeneralSpaceGrid([[ 1., 1.7, 2.4, 3.1, 3.8 ],
85                                             [2., 4.3, 6.6, 8.9, 11.2, 15.],
86                                             [3., 4.9, 5.8, 7.7, 10.,20.]])
87
88         # simulate the perburbed values
89         #####
90         nbSimul =40000
91         np.random.seed(1000)
92         x = np.random.uniform(-1.,1.,size=(1,nbSimul));
93         # mesh
94         nbMesh = np.array([16],dtype=np.int32)
95         # Create the regressor
96         #####
97         regressor = StOptReg.LocalLinearRegression(False,x,nbMesh)
98
99         # regressed values same values for each point of the grid
100        #####
101        toReal = (2+x[0,:]+(1+x[0,:])*(1+x[0,:]))
102        # function to regress
103        toRegress = toReal + 4*np.random.normal(0.,1,nbSimul)
104        # create a matrix (number of stock points by number of simulations)
105        toRegressMult = np.zeros(shape=(len(toRegress),grid.getNbPoints()))
106        for i in range(toRegressMult.shape[1]):
107            toRegressMult[:,i] = toRegress
108        # into a list : two times to test 2 regimes
109        listToReg = []
110        listToReg.append(toRegressMult)
111        listToReg.append(toRegressMult)
112
113
114        # Create the binary archive to dump
115        #####
116        archiveToWrite = StOptGens.BinaryFileArchive("MyArchive","w")
117        # step 1
118        archiveToWrite.dumpGridAndRegressedValue("toStore", 1,listToReg, regressor,grid)
119        # step 3
120        archiveToWrite.dumpGridAndRegressedValue("toStore", 3,listToReg, regressor,grid)
121
122
123        # Read the regressed values
124        #####
125        archiveToRead = StOptGens.BinaryFileArchive("MyArchive","r")
126
127        # print content
128        archiveToRead.print(0)

```

```

129     # detailed
130     archiveToRead.print(1)
131
132     contValues = archiveToRead.readGridAndRegressedValue(3,"toStore")
133
134
135     # list of 2 continuation values
136     #####
137     stockPoint = np.array([1.5,3.,5.])
138     uncertainty = np.array([0.])
139     value =contValues[0].getValue(stockPoint,uncertainty)
140
141 if __name__ == '__main__':
142     unittest.main()

```

3.6.2 Binding to store/read a regressor and a two-dimensional array

Sometimes users prefer to avoid using the provided framework and prefer to only use the Python binding associated with regression methods. When some regressions are performed for different sets of particles (meaning that one or more functions are regressed), it is possible to dump the regressor used and some values associated with these regressions:

- the `dumpSome2DArray`, `readSome2DArray` allows you to dump and read 2-dimensional numpy arrays,
- the `dumpSomeRegressor`, `readSomeRegressor` allows to dump and read a regressor.

```

1  # Copyright (C) 2017 EDF
2  # All Rights Reserved
3  # This code is published under the GNU Lesser General Public License (GNU LGPL)
4  import numpy as np
5  import pystopt.regression as StOptReg
6  import pystopt.generators as StOptGenerators
7
8  # unit test to show how to store some regression object and basis function coefficients
   associated
9  #
   #####
10
11 def createData():
12
13     X1=np.arange(0.0 , 2.2 , 0.01 )
14     X2=np.arange(0.0 , 1.1 , 0.005 )
15     Y=np.zeros((len(X1),len(X2)))
16     for i in range(len(X1)):
17         for j in range(len(X2)):
18             if i < len(X1)//2:
19                 if j < len(X2)//2:
20                     Y[i,j]=X1[i]+X2[j]
21                 else:
22                     Y[i,j]=4*X1[i]+4*X2[j]
23             else:
24                 if j < len(X2)//2:
25                     Y[i,j]=2*X1[i]+X2[j]
26                 else:
27                     Y[i,j]=2*X1[i]+3*X2[j]
28
29     XX1, XX2 = np.meshgrid(X1,X2)
30     Y=Y.T

```

```

31
32     r,c = XX1.shape
33
34     X = np.reshape(XX1,(r*c,1))[:,0]
35     I = np.reshape(XX2,(r*c,1))[:,0]
36     Y = np.reshape(Y,(r*c,1))[:,0]
37
38     xMatrix = np.zeros((2,len(X)))
39     xMatrix[0,:] = X
40     xMatrix[1,:] = I
41
42     return xMatrix, Y
43
44
45
46
47 # main
48
49 xMatrix, y = createData()
50
51 # 2 dimensional regression 2 by 2 meshes
52 nbMesh = np.array([2,2],dtype=np.int32)
53 regressor = StOptReg.LocalLinearRegression(False,xMatrix,nbMesh)
54
55 # coefficients
56 coeff = regressor.getCoordBasisFunction(y)
57
58 print("Regressed coeff", coeff)
59
60
61 # store them in a matrix
62 coeffList = np.zeros(shape=(1,3*2*2))
63 coeffList[0,:]=coeff.transpose()
64
65
66 # archive write for regressors
67 archiveWriteForRegressor =StOptGeners.BinaryFileArchive("archive","w")
68
69 # store
70 step =1
71 archiveWriteForRegressor.dumpSome2DArray("RegCoeff",step,coeff)
72 archiveWriteForRegressor.dumpSomeRegressor("Regressor",step,regressor)
73 # store a one D array
74 name1 = "AllSteps"
75 name2 = "Product"
76 for i in range(5):
77     x = i*np.ones(10)
78     archiveWriteForRegressor.dumpSome1DArray(name1, name2, x)
79
80 # archive Read for regressors
81 archiveReadForRegressor =StOptGeners.BinaryFileArchive("archive","r")
82
83 # the content
84 print("deb of content")
85 archiveReadForRegressor.print(0)
86
87 # get back
88 values = archiveReadForRegressor.readSome2DArray("RegCoeff",step)
89 reg = archiveReadForRegressor.readSomeRegressor("Regressor",step)
90 print("Regressed coeff ", values)
91 print("Reg",reg)
92
93 for i in range(5):
94     x =archiveReadForRegressor.readSome1DArray(name1, name2, i)
95     print("X ", x)

```

Chapter 4

Framework to solve some switching problems with integer state constraints by regressions

Switching problem with integer state are special case of the previous chapter where the state is separated in 3 parts as previously $X^{x,t} = (X_1^{x,t}, X_2^{x,t}, I_t)$ where $X_1^{x,t}$ is not controlled, I_t is an integer giving the regime mode and $X_2^{x,t}$ is a purely deterministic tensor taking **some integer values**. As the values taken by $X_2^{x,t}$ are purely integer no interpolation is needed. Therefore, in each regime, $X_2^{x,t}$ is discretized on a special grid taking some discrete integer values. The grid is a full grid, and as previously for stocks problems, some MPI parallelization can be carried out to calculate the asset values at the different deterministic cases.

Exemple of some switching problems We want to manage a thermal asset where prices of electricity and gas are given by a one factor model. Then $X_1^{x,t}$ is two dimensional:

- $X_{1,1}^{x,t}$ is the price for electricity
- $X_{1,2}^{x,t}$ is the price of gas multiplied by some heat rate.

When the asset is on, the gain is $X_{1,1}^{x,t} - X_{1,2}^{x,t}$ and the asset is off, the gain is null. The asset has therefore two regime (I_t takes 2 values for exemple 0 (off) and 1 (on)).

Some constraints are added :

- When the asset is “on”, it has to be “on” during at least $nMinOn$ time steps and the state $X_2^{x,t}$ is one dimensional and can take some values from 0 to $nMinoOn - 1$ storing the number of time step since it is on. The state value $nMinoOn - 1$ corresponds to the case where the number of time steps it is on at least $nMinoOn$ and the asset is allowed to switch off.
- When the asset is “off”, it has to be *off* during at least $nMinOff$ time steps and the state $X_2^{x,t}$ is one dimensional and can take some values from 0 to $nMinoOff - 1$ storing the number of time step since it is off.

Some cost can be added, for exemple when the asset switch from *off* to *on*.

4.1 Business object for switching problems

As previously, in order to use the framework, the developer describes the problem he wants to solve in one step from a state $X^{x,t}$. This business object must offer common methods and it is derived from `OptimizerSwitchBase`.

```
1 // Copyright (C) 2021 EDF
2 // All Rights Reserved
3 // This code is published under the GNU Lesser General Public License (GNU LGPL)
4 #ifndef OPTIMIZERSWITCHBASE_H
5 #define OPTIMIZERSWITCHBASE_H
6 #include <Eigen/Dense>
7 #include "StOpt/core/utils/types.h"
8 #include "StOpt/core/utils/StateWithIntState.h"
9 #include "StOpt/regression/BaseRegression.h"
10 #include "StOpt/dp/SimulatorDPBase.h"
11 #include "StOpt/core/grids/RegularSpaceIntGrid.h"
12
13 /** \file OptimizerSwitchingBase.h
14  * \brief Define an abstract class for Switching problems solved by regression methods
15  * and with an integer state
16  * \author Xavier Warin
17  */
18 namespace StOpt
19 {
20
21 /// \class OptimizerSwitchBase OptimizerSwitchBase.h
22 /// Base class for optimizer for Dynamic Programming with regression methods
23 class OptimizerSwitchBase
24 {
25
26
27 public :
28
29     OptimizerSwitchBase() {}
30
31     virtual ~OptimizerSwitchBase() {}
32
33     /// \brief defines the diffusion cone for parallelism
34     /// \param p_iRegime regime used
35     /// \param p_regionByProcessor region (min max) treated by the processor for
36     /// the different regimes treated
37     /// \return returns in each dimension the min max values in the stock that can be
38     /// reached from the grid p_gridByProcessor for each regime
39     virtual std::vector< std::array<int, 2 > > getCone(const int &p_iReg, const std::
40     vector< std::array<int, 2 > > &p_regionByProcessor) const = 0;
41
42     /// \brief defines the dimension to split for MPI parallelism
43     /// For each regime : for each dimension return true is the direction can be
44     split
45     virtual std::vector< Eigen::Array< bool, Eigen::Dynamic, 1> > getDimensionToSplit()
46     const = 0 ;
47
48     /// \brief defines a step in optimization
49     /// \param p_grid grid at arrival step after command (integer states) for each
50     regime
51     /// \param p_iReg regime treated
52     /// \param p_state coordinates of the deterministic integer state
53     /// \param p_condExp Conditional expectation operator
54     /// \param p_phiIn for each regime gives the solution calculated at the previous
55     step ( next time step by Dynamic Programming resolution) : structure of the 2D
56     array ( nb simulation ,nb stocks )
57     /// \return a pair :
58     /// - for each regimes (column) gives the solution for each particle (row)
59     /// - for each control (column) gives the optimal control for each
60     particle (rows)
```

```

52  ///
53  virtual Eigen::ArrayXd      stepOptimize(const      std::vector<std::shared_ptr<
      RegularSpaceIntGrid> >    &p_grid,
54                                     const      int &p_iReg,
55                                     const      Eigen::ArrayXi    &p_state,
56                                     const      std::shared_ptr< BaseRegression>    &
      p_condExp,
57                                     const      std::vector < std::shared_ptr< Eigen::
      ArrayXXd > > &p_phiIn) const = 0;
58
59
60  /// \brief defines a step in simulation
61  /// Notice that this implementation is not optimal but is convenient if the control is
      discrete.
62  /// By avoiding interpolation in control we avoid non admissible control
63  /// Control are recalculated during simulation.
64  /// \param p_grid      grid at arrival step after command for each regime
65  /// \param p_condExp    Conditional expectation operator reconstructing conditionnal
      expectation from basis functions for each state
66  /// \param p_basisFunc  Basis functions par each point of the grid state for each
      regime
67  /// \param p_state      defines the state value (modified)
68  /// \param p_phiInOut   defines the value functions (modified) : size number of
      functions to follow
69  virtual void stepSimulate(const std::vector< std::shared_ptr< RegularSpaceIntGrid> >
      &p_grid,
70                                     const std::shared_ptr< BaseRegression>    &p_condExp,
71                                     const std::vector< Eigen::ArrayXXd >    &p_basisFunc,
72                                     StOpt::StateWithIntState &p_state,
73                                     Eigen::Ref<Eigen::ArrayXd> p_phiInOut) const = 0 ;
74
75
76  /// \brief Get the number of regimes allowed for the asset to be reached at the
      current time step
77  /// If \f$ t \f$ is the current time, and \f$ dt \f$ the resolution step, this is
      the number of regime allowed on \f$[ t- dt, t[\f$
78  virtual      int getNbRegime() const = 0 ;
79
80  /// \brief get the simulator back
81  virtual std::shared_ptr< StOpt::SimulatorDPBase > getSimulator() const = 0;
82
83  /// \brief get size of the function to follow in simulation
84  virtual int getSimuFuncSize() const = 0;
85
86 };
87 }
88 #endif /* OPTIMIZERSWITCHBASE_H */

```

- the `getNbRegime` makes it possible to obtain the number of regimes of the problem. The number of regimes is fixed.
- the `getSimulator` method is used to retrieve the simulator giving the Monte Carlo simulations,
- the `getSimuFuncSize` method is used in simulation to define the number of functions to follow in the simulation part (see chapter 3).
- the `getCone` method is only relevant if the MPI framework with distribution is used. As argument it takes the regime and a vector of size the dimension of the grid for this regime. Each component of the vector is an array containing for the given regime, the minimum and maximum integer coordinate values of the deterministic state of the current grid defining an hyper cube H_1 . It returns for each dimension, the min

and max integer coordinates of the hyper cube $H2$ containing the state which can be reached from a state of the grid in $H1$.

- the `getDimensionToSplit` method is used to define in the MPI framework with distribution the directions to be divided for the solution on the processors. For each regime, for each dimension, it returns a Boolean where `true` means that the direction is a candidate for splitting.
- the `stepOptimize` method is used in optimization. We want to calculate the optimal value at the current t_i at a grid state `p_state`, and for the regime `p_iereg` using a vector `p_grid` of grids of state (one for each regime) at the following date t_{i+1} , `p_condEsp` is the object permitting to calculate conditional expectations of some functions calculated in the previous one treated time step t_{i+1} . At a grid point `p_state` the function calculates the function values. It returns an array (its dimension is the number of simulations) giving the value of the function for the regime treated.
- the `stepSimulate` method is used after the optimization using the continuation values calculated at each (integer) point of the grid in the optimization part. From a `p_state` state (storing the $X^{x,t}$), the conditional expectation operator `p_condExp`, and the basis functions `p_basisFunc` for each point of the grid state for each regime, the values followed during simulation along the current path are stored in `p_phiInOut`.

4.2 Using the framework to optimize or simulate on a time step

4.2.1 Optimize on one time step

Once an Optimizer is derived for the project, and assuming a `RegularSpaceIntGrid` grid is used for inventory discretization, the framework provides a `TransitionStepRegressionSwitchDist` object in MPI which allows to solve the optimization problem with data distribution over a time step with the following constructor:

```

1  TransitionStepRegressionSwitchDist(const vector< shared_ptr<RegularSpaceIntGrid> > &
2      p_pGridCurrent,
3      const vector< shared_ptr<RegularSpaceIntGrid> > &
4      p_pGridPrevious,
5      const sshared_ptr<OptimizerSwitchBase> &p_pOptimize,
6      const boost::mpi::communicator &p_world)
```

with

- `p_pGridCurrent` is the vector of grids at the current date: one by regime (so it is a vector) containing the grids describing the discrete integer values taken by the deterministic state.
- `p_pGridPrevious` is the vector of grids at the previous time step treated (so the next time step as the resolution is backward)
- `p_pOptimize` is the business model describing the physical problem to optimize;

- `p_world` The MPI communicator

Remark 22 A similar object is available without the MPI distribution framework *TransitionStepRegressionDP* with always the activation of parallelization with threads and MPI on calculations on the full points grid.

Remark 23 In the case of sparse grids with only parallelization on the calculations (threads and MPI) *TransitionStepRegressionDPSparse* object can be used.

The main method is

```
1  std::vector< shared_ptr< Eigen::ArrayXXd > > OneStep(const std::vector< shared_ptr<
    Eigen::ArrayXXd > > &p_phiIn,
2      const shared_ptr< BaseRegression> &p_condExp)
```

with

- `p_phiIn` the vector (its size corresponds to the number of regimes) of the matrix of optimal values calculated at the previous time iteration for each regime. Each matrix is a number of simulations per matrix of number of stock points matrix.
- `p_condExp` the conditional expectation operator,

returning a pair:

- The first element is a vector of matrix with new optimal values at the current time step (each element of the vector corresponds to a regime and each matrix is a number of simulations per matrix of number of stock points).
- The second element is vector of matrix with new optimal controls at the current time step (each element of the vector corresponds to a control and each matrix is a number of simulations per number of stock points matrix).

Remark 24 All *TransitionStepRegressionDP* derive from a *TransitionStepRegressionBase* object having a pure virtual *OneStep* method.

A second method is provided permitting to dump the continuation values of the problem and the optimal control at each time step:

```
1  void dumpContinuationValues(std::shared_ptr<gs::BinaryFileArchive> p_ar , const std::
    string &p_name, const int &p_iStep,
2      const std::vector< std::shared_ptr< Eigen::ArrayXXd > > &
    p_phiInPrev,
3      const std::vector< std::shared_ptr< Eigen::ArrayXXd > > &
    p_control,
4      const std::shared_ptr<BaseRegression> &p_condExp,
5      const bool &p_bOneFile) const
```

with:

- `p_ar` is the archive where the controls and solutions are dumped,
- `p_name` is a base name used in the archive to store the solution and the control,

- `p_phiInPrev` is the previous time step solution used to calculate the continuation values that are stored,
- `p_control` stores the optimal controls calculated at the current time step,
- `p_condExp` is the conditional expectation object allowing to calculate the conditional expectation of the functions defined at the previous time step processed `p_phiInPrev` and allowing to store a representation of the optimal control.
- `p_bOneFile` is set if the continuation and optimal controls calculated by each processor are flushed to a single file. Otherwise, the continuation and optimal controls calculated by each processor are flushed to different files (one by processor). If the problem results in optimal continuation and control values on the global grid that can be stored in the memory of the compute node, it may be more interesting to dump the continuation/control values to a file for policy simulation optimal.

Remark 25 *As for the `TransitionStepRegressionDP` object and the `TransitionStepRegressionDPSparse` object, their `dumpContinuationValues` does not need an argument `p_bOneFile`: the optimal controls are obviously stored in a single file.*

Here we give a simple example of temporal resolution using this method when MPI data distribution is used

```

1 // Copyright (C) 2016 EDF
2 // All Rights Reserved
3 // This code is published under the GNU Lesser General Public License (GNU LGPL)
4 #ifdef USE_MPI
5 #include <fstream>
6 #include <memory>
7 #include <functional>
8 #include <boost/lexical_cast.hpp>
9 #include <boost/mpi.hpp>
10 #include <Eigen/Dense>
11 #include "geners/BinaryFileArchive.hh"
12 #include "StOpt/core/grids/FullGrid.h"
13 #include "StOpt/regression/BaseRegression.h"
14 #include "StOpt/dp/FinalStepDPDist.h"
15 #include "StOpt/dp/TransitionStepRegressionDPDist.h"
16 #include "StOpt/core/parallelism/reconstructProcOMP.h"
17 #include "StOpt/dp/OptimizerDPBase.h"
18 #include "StOpt/dp/SimulatorDPBase.h"
19
20
21 using namespace std;
22
23 double DynamicProgrammingByRegressionDist(const shared_ptr<StOpt::FullGrid> &p_grid,
24     const shared_ptr<StOpt::OptimizerDPBase> &p_optimize,
25     shared_ptr<StOpt::BaseRegression> &p_regressor,
26     const function<double(const int &, const Eigen::ArrayXd &, const Eigen::ArrayXd &)>
27         &p_funcFinalValue,
28     const Eigen::ArrayXd &p_pointStock,
29     const int &p_initialRegime,
30     const string &p_fileToDump,
31     const bool &p_bOneFile,
32     const boost::mpi::communicator &p_world)
33 {
34     // from the optimizer get back the simulator
35     shared_ptr<StOpt::SimulatorDPBase> simulator = p_optimize->getSimulator();
36     // final values

```

```

36     vector< shared_ptr< Eigen::ArrayXXd > > valuesNext = StOpt::FinalStepDPDist(p_grid,
        p_optimize->getNbRegime(), p_optimize->getDimensionToSplit(), p_world)(
        p_funcFinalValue, simulator->getParticles().array());
37     // dump
38     string toDump = p_fileToDump ;
39     // test if one file generated
40     if (!p_bOneFile)
41         toDump += "_" + boost::lexical_cast<string>(p_world.rank());
42     shared_ptr<gs::BinaryFileArchive> ar;
43     if ((!p_bOneFile) || (p_world.rank() == 0))
44         ar = make_shared<gs::BinaryFileArchive>(toDump.c_str(), "w");
45     // name for object in archive
46     string nameAr = "Continuation";
47     for (int iStep = 0; iStep < simulator->getNbStep(); ++iStep)
48     {
49         Eigen::ArrayXXd asset = simulator->stepBackwardAndGetParticles();
50         // conditional expectation operator
51         p_regressor->updateSimulations(((iStep == (simulator->getNbStep() - 1)) ? true :
            false), asset);
52         // transition object
53         StOpt::TransitionStepRegressionDPDist transStep(p_grid, p_grid, p_optimize, p_world
            );
54         pair< vector< shared_ptr< Eigen::ArrayXXd > >, vector< shared_ptr< Eigen::ArrayXXd
            > > > valuesAndControl = transStep.oneStep(valuesNext, p_regressor);
55         transStep.dumpContinuationValues(ar, nameAr, iStep, valuesNext, valuesAndControl.
            second, p_regressor, p_bOneFile);
56         valuesNext = valuesAndControl.first;
57     }
58     // reconstruct a small grid for interpolation
59     return StOpt::reconstructProcOMpi(p_pointStock, p_grid, valuesNext[p_initialRegime],
        p_optimize->getDimensionToSplit(), p_world).mean();
60 }
61 }
62 #endif

```

An example without data distribution can be found in the file `DynamicProgrammingByRegression.cpp`. We finally give an array with the different `TransitionStepRegression` objects to use depending on the type of parallelization used.

Table 4.1: Which object `TransitionStepRegression` to use depending on the grid used and the type of parallelization used.

	Full grid	Sparse grid
Sequential	<code>TransitionStepRegressionDP</code>	<code>TransitionStepRegressionDPSparse</code>
Parallelization on calculations threads and MPI	<code>TransitionStepRegressionDP</code>	<code>TransitionStepRegressionDPSparse</code>
Distribution of calculations and data	<code>TransitionStepRegressionDPDist</code>	Not available

The framework in simulation with classical regressions

Once the optimization is done, the continuation values are saved in a file (or in some files) at each time step. In order to simulate the optimal policy, we can use the previously calculated continuation values (see chapter 4) or we can use the optimal controls calculated in optimization. In continuous optimization, the use of control is more efficient in terms of computational cost. When the control is discrete, the interpolation of the controls can lead to inadmissible controls and the simulation with the value function is more precise.

When simulating optimal control, two cases can occur:

- In most cases (small dimension case), the optimal control or optimal function value can be stored in the compute node memory and the function values and controls are stored in a single file. In this case, an optimum control simulation can be easily performed by distributing the Monte Carlo simulations on the available compute nodes: this can be done using the `SimulateStepRegression` or `SimulateStepRegressionControl` objects at each time step of the simulation.
- When it comes to very large problems, optimization is achieved by distributing the data across the processors and it is impossible to store the optimal command on a node. In this case, the optimal controls and the optimal solutions are stored in the memory of the node which was used to calculate them in optimization. The simulations are reorganized at each time step and gathered so as to occupy the same part of the global grid. Each processor will then get from the other processors a version of the optimal control or solution it needs. This methodology is used in the `SimulateStepRegressionDist` and `SimulateStepRegressionControlDist` objects.

We detail the simulations objects using the optimal function value calculated in optimization and the optimal control for the case of very large problems.

- Simulation step using the value function calculated in optimization:

In order to simulate one step of the optimal policy, an object `SimulateStepRegressionDist` is provided with constructor

```

1  SimulateStepRegressionDist(gs::BinaryFileArchive &p_ar,  const int &p_iStep,  const
    std::string &p_nameCont,
2                                const  shared_ptr<FullGrid> &p_pGridFollowing,  const
    shared_ptr<OptimizerDPBase > &p_pOptimize,
3                                const bool &p_bOneFile,  const boost::mpi::communicator &
    p_world )

```

where

- `p_ar` is the binary archive where the continuation values are stored,
- `p_iStep` is the number associated with the current time step (0 at the start date of the simulation, the number is increased by one at each simulated time step),
- `p_nameCont` is the base name for continuation values,
- `p_GridFollowing` is the grid at the next time step (`p_iStep+1`),
- `p_Optimize` the Optimizer describing the passage from one time step to the next,
- `p_OneFile` equal to `true` if only one archive is used to store continuation values.
- `p_world` The MPI communicator

Remark 26 *A version without data distribution but with multithreaded and with possible MPI on calculations is available with the object `SimulateStepRegression`. The `p_OneFile` argument is omitted during construction.*

This object implements the method `oneStep`

```

1 void oneStep(std::vector<StateWithStocks<double> > &p_statevector , Eigen::ArrayXXd &
  p_phiInOut)

```

where:

- `p_statevector` stores the states of all simulations: this state is updated by applying the optimal command,
- `p_phiInOut` stores the gain/cost functions for all the simulations: it is updated by the function call. The size of the array is $(nb, nbSimul)$ where nb is given by the `getSimuFuncSize` method of the optimizer and $nbSimul$ the number of Monte Carlo simulations.

An example of using this method to simulate an optimal policy with distribution is given below:

```

1 // Copyright (C) 2016 EDF
2 // All Rights Reserved
3 // This code is published under the GNU Lesser General Public License (GNU LGPL)
4 #ifndef SIMULATEREGRESSIONDIST_H
5 #define SIMULATEREGRESSIONDIST_H
6 #include <functional>
7 #include <memory>
8 #include <Eigen/Dense>
9 #include <boost/mpi.hpp>
10 #include "generators/BinaryFileArchive.hh"
11 #include "StOpt/core/grids/FullGrid.h"
12 #include "StOpt/core/utils/StateWithStocks.h"
13 #include "StOpt/dp/SimulateStepRegressionDist.h"
14 #include "StOpt/dp/OptimizerDPBase.h"
15 #include "StOpt/dp/SimulatorDPBase.h"
16
17
18 /** \file SimulateRegressionDist.h
19  * \brief Defines a simple program showing how to use simulation
20  * A simple grid is used
21  * \author Xavier Warin
22  */
23
24
25 /// \brief Simulate the optimal strategy , mpi version
26 /// \param p_grid grid used for deterministic state (stocks for
  example)
27 /// \param p_optimize optimizer defining the optimization between two
  time steps
28 /// \param p_funcFinalValue function defining the final value
29 /// \param p_pointStock initial point stock
30 /// \param p_initialRegime regime at initial date
31 /// \param p_fileToDump name associated to dumped bellman values
32 /// \param p_bOneFile do we store continuation values in only one file
33 /// \param p_world MPI communicator
34 double SimulateRegressionDist(const std::shared_ptr<StOpt::FullGrid> &p_grid,
35                               const std::shared_ptr<StOpt::OptimizerDPBase> &
  p_optimize,
36                               const std::function<double(const int &, const Eigen::
  ArrayXd &, const Eigen::ArrayXd &)> &
  p_funcFinalValue,
37                               const Eigen::ArrayXd &p_pointStock,
38                               const int &p_initialRegime,
39                               const std::string &p_fileToDump,
40                               const bool &p_bOneFile,
41                               const boost::mpi::communicator &p_world)
42 {
43     // from the optimizer get back the simulator

```

```

44     std::shared_ptr< StOpt::SimulatorDPBase> simulator = p_optimize->getSimulator();
45     int nbStep = simulator->getNbStep();
46     std::vector< StOpt::StateWithStocks<double>> states;
47     states.reserve(simulator->getNbSimul());
48     for (int is = 0; is < simulator->getNbSimul(); ++is)
49         states.push_back(StOpt::StateWithStocks<double>(p_initialRegime, p_pointStock,
50             Eigen::ArrayXd::Zero(simulator->getDimension())));
51     std::string toDump = p_fileToDump ;
52     // test if one file generated
53     if (!p_bOneFile)
54         toDump += "_" + boost::lexical_cast<std::string>(p_world.rank());
55     gs::BinaryFileArchive ar(toDump.c_str(), "r");
56     // name for continuation object in archive
57     std::string nameAr = "Continuation";
58     // cost function
59     Eigen::ArrayXXd costFunction = Eigen::ArrayXXd::Zero(p_optimize->getSimuFuncSize()
60         , simulator->getNbSimul());
61     for (int istep = 0; istep < nbStep; ++istep)
62     {
63         StOpt::SimulateStepRegressionDist(ar, nbStep - 1 - istep, nameAr, p_grid,
64             p_optimize, p_bOneFile, p_world).oneStep(states, costFunction);
65
66         // new stochastic state
67         Eigen::ArrayXXd particles = simulator->stepForwardAndGetParticles();
68         for (int is = 0; is < simulator->getNbSimul(); ++is)
69             states[is].setStochasticRealization(particles.col(is));
70     }
71     // final : accept to exercise if not already done entirely (here suppose one
72     // function to follow)
73     for (int is = 0; is < simulator->getNbSimul(); ++is)
74         costFunction(0, is) += p_funcFinalValue(states[is].getRegime(), states[is].
75             getPtStock(), states[is].getStochasticRealization()) * simulator->getActu
76             ();
77
78     return costFunction.mean();
79 }
80 #endif /* SIMULATEREGRESSIONDIST_H */

```

The version of the previous example using a single archive storing the control/solution is given in the `SimulateRegression.h` file.

- Simulation step using the optimal controls calculated in optimization:

```

1     SimulateStepRegressionControlDist(gs::BinaryFileArchive &p_ar,  const int &p_iStep
2         ,  const std::string &p_nameCont,
3             const std::shared_ptr<FullGrid> &p_pGridCurrent
4             ,  const std::shared_ptr<FullGrid> &
5                 p_pGridFollowing,
6                 const std::shared_ptr<OptimizerDPBase > &
7                     p_pOptimize,
8                     const bool &p_bOneFile,
9                     const boost::mpi::communicator &p_world);

```

where

- `p_ar` is the binary archive where the continuation values are stored,
- `p_iStep` is the number associated with the current time step (0 at the start date of simulation, the number is increased by one at each simulated time step),
- `p_nameCont` is the base name of the control values,

- `p_GridCurrent` is the grid at the current time step (`p_iStep`),
- `p_GridFollowing` is the grid at the next time step (`p_iStep+1`),
- `p_Optimize` is the Optimizer describing the passage from one time step to the next,
- `p_OneFile` is equal to `true` if only one archive is used to store continuation values.
- `p_world` The MPI communicator

Remark 27 *A version in which a single archive storing the control/solution is used is available with the object `SimulateStepRegressionControl`*

This object implements the method `oneStep`

```
1 void oneStep(std::vector<StateWithStocks<double> > &p_statevector , Eigen::ArrayXd &
   p_phiInOut)
```

where:

- `p_statevector` stores the state for all simulations: this state is updated by applying optimal commands,
- `p_phiInOut` stores the gain/cost functions for all simulations: it is updated by the function call. The size of the array is $(nb, nbSimul)$ where nb is given by the `getSimuFuncSize` method of the optimizer and $nbSimul$ the number of Monte Carlo simulations.

An example of using this method to simulate an optimal policy with distribution is given below:

```
1 // Copyright (C) 2016 EDF
2 // All Rights Reserved
3 // This code is published under the GNU Lesser General Public License (GNU LGPL)
4 #ifdef USE_MPI
5 #ifndef SIMULATEREGRESSIONCONTROLDIST_H
6 #define SIMULATEREGRESSIONCONTROLDIST_H
7 #include <functional>
8 #include <memory>
9 #include <Eigen/Dense>
10 #include <boost/mpi.hpp>
11 #include "geners/BinaryFileArchive.hh"
12 #include "StOpt/core/grids/FullGrid.h"
13 #include "StOpt/core/utils/StateWithStocks.h"
14 #include "StOpt/dp/SimulateStepRegressionControlDist.h"
15 #include "StOpt/dp/OptimizerDPBase.h"
16 #include "StOpt/dp/SimulatorDPBase.h"
17
18
19 /** \file SimulateRegressionControlDist.h
20  * \brief Defines a simple program showing how to use simulation
21  * A simple grid is used
22  * \author Xavier Warin
23  */
24
25
26 /// \brief Simulate the optimal strategy using optimal controls calculated in
27 /// optimization , mpi version
28 /// \param p_grid grid used for deterministic state (stocks for
29 /// example)
```

```

28 /// \param p_optimize          optimizer defining the optimization between two
    time steps
29 /// \param p_funcFinalValue    function defining the final value
30 /// \param p_pointStock        initial point stock
31 /// \param p_initialRegime     regime at initial date
32 /// \param p_fileToDump        name associated to dumped bellman values
33 /// \param p_bOneFile          do we store continuation values in only one file
34 /// \param p_world              MPI communicator
35 double SimulateRegressionControlDist(const std::shared_ptr<StOpt::FullGrid> &p_grid,
36                                     const std::shared_ptr<StOpt::OptimizerDPBase > &
37                                     p_optimize,
38                                     const std::function<double(const int &, const
39                                     Eigen::ArrayXd &, const Eigen::ArrayXd &)> &
40                                     p_funcFinalValue,
41                                     const Eigen::ArrayXd &p_pointStock,
42                                     const int &p_initialRegime,
43                                     const std::string &p_fileToDump,
44                                     const bool &p_bOneFile,
45                                     const boost::mpi::communicator &p_world)
46 {
47     // from the optimizer get back the simulator
48     std::shared_ptr< StOpt::SimulatorDPBase> simulator = p_optimize->getSimulator();
49     int nbStep = simulator->getNbStep();
50     std::vector< StOpt::StateWithStocks<double>> states;
51     states.reserve(simulator->getNbSimul());
52     for (int is = 0; is < simulator->getNbSimul(); ++is)
53         states.push_back(StOpt::StateWithStocks<double>(p_initialRegime, p_pointStock,
54                                                         Eigen::ArrayXd::Zero(simulator->getDimension())));
55     std::string toDump = p_fileToDump ;
56     // test if one file generated
57     if (!p_bOneFile)
58         toDump += "_" + boost::lexical_cast<std::string>(p_world.rank());
59     gs::BinaryFileArchive ar(toDump.c_str(), "r");
60     // name for continuation object in archive
61     std::string nameAr = "Continuation";
62     // cost function
63     Eigen::ArrayXXd costFunction = Eigen::ArrayXXd::Zero(p_optimize->getSimuFuncSize()
64                                                         , simulator->getNbSimul());
65     for (int istep = 0; istep < nbStep; ++istep)
66     {
67         StOpt::SimulateStepRegressionControlDist(ar, nbStep - 1 - istep, nameAr,
68                                                   p_grid, p_grid, p_optimize, p_bOneFile, p_world).oneStep(states,
69                                                         costFunction);
70
71         // new stochastic state
72         Eigen::ArrayXXd particules = simulator->stepForwardAndGetParticles();
73         for (int is = 0; is < simulator->getNbSimul(); ++is)
74             states[is].setStochasticRealization(particules.col(is));
75     }
76     // final : accept to exercise if not already done entirely (here suppose one
77     // function to follow)
78     for (int is = 0; is < simulator->getNbSimul(); ++is)
79         costFunction(0, is) += p_funcFinalValue(states[is].getRegime(), states[is].
80                                                 getPtStock(), states[is].getStochasticRealization()) * simulator->getActu
81                                                 ();
82
83     return costFunction.mean();
84 }
85 #endif /* SIMULATEREGRESSIONCONTROLDIST_H */
86 #endif

```

The version of the previous example using a single archive storing the control/solution is given in the `SimulateRegressionControl.h` file.

In the table below, we indicate which simulation object should be used at each time step depending on the `TransitionStepRegressionDP` object used in optimization.

Table 4.2: Which simulation object to use depending on the TransitionStepRegression object used.

	TransitionStepRegressionDP	TransitionStepRegressionDPDist bOneFile = true	TransitionStepRegressionDPDist bOneFile = false	TransitionStepRegressionDPSparse
SimulateStepRegression	Yes	Yes	No	Yes
SimulateStepRegressionControl	Yes	Yes	No	Yes
SimulateStepRegressionDist	No	Yes	Yes	No
SimulateStepRegressionControlDist	No	Yes	Yes	No

4.2.2 Regressions and cuts for continuous linear transition problems with certain characteristics of concavity and convexity

During the optimization for example a storage, one can want to solve the transition problem on certain time steps by supposing that the uncertainties are known. The Bellman values for a given uncertainty are then concave compared to the storage when one tries to maximize certain gains for example. When the problem is continuous linear, we can use bender cuts to approximate the Bellman value with respect to the storage level as in the SDDP method 1. To use this cuts a business object using an LP solver must be created. This business object is derived from `OptimizerDPCutBase` itself derived from `OptimizerBaseInterp`.

```
1 // Copyright (C) 2019 EDF
2 // All Rights Reserved
3 // This code is published under the GNU Lesser General Public License (GNU LGPL)
4 #ifndef OPTIMIZERDPCUTBASE_H
5 #define OPTIMIZERDPCUTBASE_H
6 #include <Eigen/Dense>
7 #include "StOpt/core/utils/StateWithStocks.h"
8 #include "StOpt/core/grids/SpaceGrid.h"
9 #include "StOpt/regression/BaseRegression.h"
10 #include "StOpt/regression/ContinuationCuts.h"
11 #include "StOpt/dp/SimulatorDPBase.h"
12 #include "StOpt/dp/OptimizerBase.h"
13
14 /** \file OptimizerDPCutBase.h
15  * \brief Define an abstract class for Dynamic Programming problems solved by regression
16  *        methods using cust to approximate
17  *        Bellman values
18  *        \author Xavier Warin
19  */
20 namespace StOpt
21 {
22
23     /// \class OptimizerDPCutBase OptimizerDPCutBase.h
24     /// Base class for optimizer for Dynamic Programming with regression methods and cuts, so
25     /// using LP to solve transitional problems
26     class OptimizerDPCutBase : public OptimizerBase
27     {
28     public :
29
30         OptimizerDPCutBase() {}
31
32         virtual ~OptimizerDPCutBase() {}
33
34         /// \brief defines the diffusion cone for parallelism
35         /// \param p_regionByProcessor region (min max) treated by the processor for
36         /// the different regimes treated
37         /// \return returns in each dimension the min max values in the stock that can be
38         /// reached from the grid p_gridByProcessor for each regime
39         virtual std::vector< std::array< double, 2> > getCone(const std::vector< std::array<
40             double, 2> > &p_regionByProcessor) const = 0;
41
42         /// \brief defines the dimension to split for MPI parallelism
43         /// For each dimension return true is the direction can be split
44         virtual Eigen::Array< bool, Eigen::Dynamic, 1> getDimensionToSplit() const = 0 ;
45
46         /// \brief defines a step in optimization
47         /// \param p_grid grid at arrival step after command
48         /// \param p_stock coordinates of the stock point to treat
49         /// \param p_condEsp continuation values for each regime
50         /// \return For each regimes (column) gives the solution for each particle , and cut
51         /// (row)
```

```

49  ///      For a given simulation , cuts components (C) at a point stock \f$ \bar S
50  \f$ are given such that the cut is given by
51  ///      \f$ C[0] + \sum_{i=1}^d C[i] (S_i - \bar S_i) \f$
52  virtual Eigen::ArrayXXd stepOptimize(const std::shared_ptr< StOpt::SpaceGrid> &
53  p_grid, const Eigen::ArrayXd &p_stock,
54  const std::vector< StOpt::ContinuationCuts > &
55  p_condEsp) const = 0;
56
57  /// \brief defines a step in simulation
58  /// Control are recalculated during simulation using a local optimization using the LP
59  /// \param p_grid grid at arrival step after command
60  /// \param p_continuation defines the continuation operator for each regime
61  /// \param p_state defines the state value (modified)
62  /// \param p_phiInOut defines the value functions (modified) : size number of
63  functions to follow
64  virtual void stepSimulate(const std::shared_ptr< StOpt::SpaceGrid> &p_grid, const std
65  ::vector< StOpt::ContinuationCuts > &p_continuation,
66  StOpt::StateWithStocks<double> &p_state,
67  Eigen::Ref<Eigen::ArrayXd> p_phiInOut) const = 0 ;
68
69  /// \brief Get the number of regimes allowed for the asset to be reached at the
70  current time step
71  /// If \f$ t \f$ is the current time, and \f$ dt \f$ the resolution step, this is
72  the number of regime allowed on \f$[ t- dt, t[\f$
73  virtual int getNbRegime() const = 0 ;
74
75  /// \brief get the simulator back
76  virtual std::shared_ptr< StOpt::SimulatorDPBase > getSimulator() const = 0;
77
78  /// \brief get back the dimension of the control
79  virtual int getNbControl() const = 0 ;
80
81  /// \brief get size of the function to follow in simulation
82  virtual int getSimuFuncSize() const = 0;
83
84 };
85 }
86 #endif /* OPTIMIZERDPCUTBASE_H */

```

We detail the different methods to implement in addition to the `OptimizerBaseInterp` methods:

- the `stepOptimize` method is used in optimization. We want to calculate the optimal value and the corresponding sensibilities with respect to the stocks at current t_i at a grid point `p_stock` using a grid `p_grid` at the next date t_{i+1} , the continuation cuts values for all regimes `p_condEsp` permitting to calculate an upper estimation (when maximizing) of conditional expectation of the optimal values using some optimization calculated at the previously treated time step t_{i+1} . From a grid point `p_stock` it calculates the function values and the corresponding sensibilities. It returns a matrix (first dimension is the number of simulations by the number of cuts components (number of storage +1), second dimension the number of regimes) giving the function value and sensibilities.
- the `stepSimulate` method is used after optimization using the continuation cuts values calculated in the optimization part. From a state `p_state` (storing the $X^{x,t}$), the continuation cuts values calculated in optimization `p_continuation`, the optimal cash flows along the current trajectory are stored in `p_phiInOut`.

In the case of gas storage, the Optimize object is given in the `OptimizeGasStorageCut.h` file.

The framework in optimization using certain cutting methods

Once an Optimizer object describing the business problem to be solved with cuts is created for the project, and assuming that a full grid is used for the discretization of the stock, the framework provides a `TransitionStepRegressionDPCutDist` object in MPI which solves the optimization problem with data adistribution over a time step with the following constructor:

```
1 TransitionStepRegressionDPCutDist(const shared_ptr<FullGrid> &p_pGridCurrent,
2                                 const shared_ptr<FullGrid> &p_pGridPrevious,
3                                 const shared_ptr<OptimizerDPCutBase> &p_pOptimize,
4                                 const boost::mpi::communicator &p_world):
```

with

- `p_pGridCurrent` is the grid at the current time step (t_i),
- `p_pGridPrevious` is the grid at the previously processed time step (t_{i+1}),
- `p_pOptimize` the optimizer object
- `p_world` The MPI communicator

The construction is very similar to the classical regression methods using only the discretization by command.

Remark 28 *A similar object is available without the MPI distribution framework `TransitionStepRegressionDPCut` with always the activation of parallelization with threads and MPI on calculations on the full grid points.*

The main method is

```
1 std::vector< shared_ptr< Eigen::ArrayXXd > > OneStep(const std::vector< shared_ptr<
2 Eigen::ArrayXXd > > &p_phiIn,
3             const shared_ptr< BaseRegression> &p_condExp)
```

with

- `p_phiIn` the vector (its size corresponds to the number of regimes) of the matrix of optimal values and sensitivities calculated at the previous time iteration for each regime. Each matrix has a number of rows equal to the number of simulations by the number of stock plus one. The number of columns is equal to the number of stock points on the grid. In the row, the number of simulations by the number of stock plus one value is stored as follows:
 - The first values (number of simulations : NS) correspond to the optimal Bellman values at a given stock point,
 - The NS values following corresponds to sensitivities $\frac{\partial V}{\partial S_1}$ at first storage
 - The NS values following corresponds to sensitivities at the second storage...

— ...

- `p_condExp` the conditional expectation operator,

returning a vector of matrix with new optimal values and sensitivities to the current time step (each element of the vector corresponds to a regime and each matrix has a size equal to (number of simulations by (the number of storage plus one)) by the number of stock points). The structure of the output is then similar to the `p_phiIn` input.

A second method is provided allowing to dump the continuation values and cuts of the problem and the optimal control at each time step:

```
1 void dumpContinuationCutsValues(std::shared_ptr<gs::BinaryFileArchive> p_ar , const std
  ::string &p_name, const int &p_iStep,
2                               const std::vector< std::shared_ptr< Eigen::ArrayXXd > > &
  p_phiInPrev,
3                               const std::shared_ptr<BaseRegression> &p_condExp,
4                               const bool &p_bOneFile) const
```

with:

- `p_ar` is the archive where controls and solutions are dumped,
- `p_name` is a base name used in the archive to store the solution and the control,
- `p_phiInPrev` is the solution to the previous time step used to calculate the values of continuation cuts which are stored,
- `p_condExp` is the conditional expectation object allowing to calculate the conditional expectation of the functions defined at the preceding time step processed `p_phiInPrev`.
- `p_bOneFile` is set if the continuation cut values calculated by each processor are dumped on a single file. Otherwise the continuation cut values calculated by each processor are dumped on different files (one per processor). If the problem results from the cuts of the values on the global grid that can be stored in the memory of the compute node, it may be more interesting to dump them in a single file for the simulation of the optimal policy.

Here we give a simple example of temporal resolution using this method when MPI data distribution is used

```
1 // Copyright (C) 2019 EDF
2 // All Rights Reserved
3 // This code is published under the GNU Lesser General Public License (GNU LGPL)
4 #ifdef USE_MPI
5 #include <fstream>
6 #include <memory>
7 #include <functional>
8 #include <boost/lexical_cast.hpp>
9 #include <boost/mpi.hpp>
10 #include <Eigen/Dense>
11 #include "generators/BinaryFileArchive.hh"
12 #include "StOpt/core/grids/FullGrid.h"
13 #include "StOpt/regression/BaseRegression.h"
14 #include "StOpt/dp/FinalStepDPCutDist.h"
15 #include "StOpt/dp/TransitionStepRegressionDPCutDist.h"
16 #include "StOpt/core/parallelism/reconstructProcOMpi.h"
17 #include "StOpt/dp/OptimizerDPCutBase.h"
18 #include "StOpt/dp/SimulatorDPBase.h"
```

```

19
20
21 using namespace std;
22 using namespace Eigen;
23
24 double DynamicProgrammingByRegressionCutDist(const shared_ptr<StOpt::FullGrid> &p_grid,
25 const shared_ptr<StOpt::OptimizerDPCutBase > &p_optimize,
26 shared_ptr<StOpt::BaseRegression> &p_regressor,
27 const function< ArrayXd(const int &, const ArrayXd &, const ArrayXd &)> &
28     p_funcFinalValue,
29 const ArrayXd &p_pointStock,
30 const int &p_initialRegime,
31 const string &p_fileToDump,
32 const bool &p_bOneFile,
33 const boost::mpi::communicator &p_world)
34 {
35     // from the optimizer get back the simulator
36     shared_ptr< StOpt::SimulatorDPBase> simulator = p_optimize->getSimulator();
37     // final values
38     vector< shared_ptr< ArrayXXd > > valueCutsNext = StOpt::FinalStepDPCutDist(p_grid,
39 p_optimize->getNbRegime(), p_optimize->getDimensionToSplit(), p_world)(
40     p_funcFinalValue, simulator->getParticles().array());
41     // dump
42     string toDump = p_fileToDump ;
43     // test if one file generated
44     if (!p_bOneFile)
45         toDump += "_" + boost::lexical_cast<string>(p_world.rank());
46     shared_ptr<gs::BinaryFileArchive> ar;
47     if ((!p_bOneFile) || (p_world.rank() == 0))
48         ar = make_shared<gs::BinaryFileArchive>(toDump.c_str(), "w");
49     // name for object in archive
50     string nameAr = "Continuation";
51     for (int iStep = 0; iStep < simulator->getNbStep(); ++iStep)
52     {
53         ArrayXXd asset = simulator->stepBackwardAndGetParticles();
54         // conditional expectation operator
55         p_regressor->updateSimulations(((iStep == (simulator->getNbStep() - 1)) ? true :
56         false), asset);
57         // transition object
58         StOpt::TransitionStepRegressionDPCutDist transStep(p_grid, p_grid, p_optimize,
59 p_world);
60         vector< shared_ptr< ArrayXXd > > valueCuts = transStep.oneStep(valueCutsNext,
61 p_regressor);
62         transStep.dumpContinuationCutsValues(ar, nameAr, iStep, valueCutsNext, p_regressor,
63 p_bOneFile);
64         valueCutsNext = valueCuts;
65     }
66     // reconstruct a small grid for interpolation
67     ArrayXd valSim = StOpt::reconstructProc0Mpi(p_pointStock, p_grid, valueCutsNext[
68 p_initialRegime], p_optimize->getDimensionToSplit(), p_world);
69     return ((p_world.rank() == 0) ? valSim.head(simulator->getNbSimul()).mean() : 0.);
70 }
71 #endif

```

An example without data distribution can be found in the file `DynamicProgrammingByRegressionCut.cpp`.

The simulation framework using cuts to approximate the Bellman's values

Once the optimization carried out, the values of the continuation cuts are saved in a file (or in certain files) at each time step. In order to simulate the optimal policy, we use previously calculated continuation cut values.

- In most cases (small dimension case), the optimal values of the cut function can be

stored in the compute node memory and the cut values are stored in a single file. In this case, a simulation of the optimal control can be easily performed by distributing the Monte Carlo simulations on the available compute nodes: this can be done by using the object `SimulateStepRegressionCut` at each time step of the simulation.

- When it comes with very large problems, optimization is achieved by distributing the data across the processors and it is impossible to store the optimal cuts values on a node. In this case, the optimal cuts values are stored in the memory of the node which was used to calculate them in optimization. The simulations are reorganized at each time step and gathered so as to occupy the same part of the global grid. Each processor will then get from the other processors a localized version of the optimal control or solution that it needs. This methodology is used in the `SimulateStepRegressionCutDist` objects.

We detail the simulations objects using the optimal cut values calculated in optimization for the case of very large problems.

In order to simulate an optimal policy time step, a `SimulateStepRegressionCutDist` object is provided with constructor

```
1 SimulateStepRegressionCutDist(gs::BinaryFileArchive &p_ar,  const int &p_iStep,  const
   std::string &p_nameCont,
2                               const shared_ptr<FullGrid> &p_pGridFollowing, const
   shared_ptr<OptimizerDPCutBase > &p_pOptimize,
3                               const bool &p_bOneFile, const boost::mpi::communicator &
   p_world )
```

where

- `p_ar` is the binary archive where the continuation values are stored,
- `p_iStep` is the number associated with the current time step (0 at the start date of the simulation, the number is increased by one at each simulated time step),
- `p_nameCont` is the base name for continuation values,
- `p_GridFollowing` is the grid at the next time step (`p_iStep+1`),
- `p_Optimize` the Optimizer describing the transition problem solved using a LP program.
- `p_OneFile` equal to `true` if only one archive is used to store continuation values.
- `p_world` The MPI communicator

Remark 29 *A version without data distribution but with multithreaded and with MPI possible on the calculations is available with the object `SimulateStepRegressionCut`. The `p_OneFile` argument is omitted during construction.*

This object implements the `oneStep` method

```
1 void oneStep(std::vector<StateWithStocks<double> > &p_statevector , Eigen::ArrayXXd &
   p_phiInOut)
```

where:

- `p_statevector` stores the states of all the simulations: this state is updated by applying the optimal command,
- `p_phiInOut` stores the gain/cost functions for all simulations: it is updated by the function call. The size of the array is $(nb, nbSimul)$ where nb is given by the `getSimuFuncSize` method of the optimizer and $nbSimul$ the number of Monte Carlo simulations.

An example of using this method to simulate an optimal policy with distribution is given below:

```
1 // Copyright (C) 2019 EDF
2 // All Rights Reserved
3 // This code is published under the GNU Lesser General Public License (GNU LGPL)
4 #ifndef SIMULATEREGRESSIONCUTDIST_H
5 #define SIMULATEREGRESSIONCUTDIST_H
6 #include <functional>
7 #include <memory>
8 #include <Eigen/Dense>
9 #include <boost/mpi.hpp>
10 #include "generators/BinaryFileArchive.hh"
11 #include "StOpt/core/grids/FullGrid.h"
12 #include "StOpt/core/utils/StateWithStocks.h"
13 #include "StOpt/dp/SimulateStepRegressionCutDist.h"
14 #include "StOpt/dp/OptimizerDPCutBase.h"
15 #include "StOpt/dp/SimulatorDPBase.h"
16
17
18 /** \file SimulateRegressionCutDist.h
19  * \brief Defines a simple program showing how to use simulations when optimizatoin achived
20  *         with transition problems solved with cuts.
21  *         A simple grid is used
22  * \author Xavier Warin
23  */
24
25 /// \brief Simulate the optimal strategy , mpi version, Bellman cuts used to allow LP
26         resolution of transition problems
27 /// \param p_grid          grid used for deterministic state (stocks for example)
28 /// \param p_optimize      optimizer defining the optimization between two time
29         steps
30 /// \param p_funcFinalValue function defining the final value cuts
31 /// \param p_pointStock    initial point stock
32 /// \param p_initialRegime regime at initial date
33 /// \param p_fileToDump    name associated to dumped bellman values
34 /// \param p_bOneFile      do we store continuation values in only one file
35 /// \param p_world         MPI communicator
36 double SimulateRegressionCutDist(const std::shared_ptr<StOpt::FullGrid> &p_grid,
37                                 const std::shared_ptr<StOpt::OptimizerDPCutBase > &
38                                 p_optimize,
39                                 const std::function< Eigen::ArrayXd(const int &, const
40                                 Eigen::ArrayXd &, const Eigen::ArrayXd &)> &
41                                 p_funcFinalValue,
42                                 const Eigen::ArrayXd &p_pointStock,
43                                 const int &p_initialRegime,
44                                 const std::string &p_fileToDump,
45                                 const bool &p_bOneFile,
46                                 const boost::mpi::communicator &p_world)
47 {
48     // from the optimizer get back the simulator
49     std::shared_ptr< StOpt::SimulatorDPBase> simulator = p_optimize->getSimulator();
50     int nbStep = simulator->getNbStep();
51     std::vector< StOpt::StateWithStocks<double>> states;
```

```

47     states.reserve(simulator->getNbSimul());
48     for (int is = 0; is < simulator->getNbSimul(); ++is)
49         states.push_back(StOpt::StateWithStocks<double>(p_initialRegime, p_pointStock,
50             Eigen::ArrayXd::Zero(simulator->getDimension())));
51     std::string toDump = p_fileToDump ;
52     // test if one file generated
53     if (!p_bOneFile)
54         toDump += "_" + boost::lexical_cast<std::string>(p_world.rank());
55     gs::BinaryFileArchive ar(toDump.c_str(), "r");
56     // name for continuation object in archive
57     std::string nameAr = "Continuation";
58     // cost function
59     Eigen::ArrayXXd costFunction = Eigen::ArrayXXd::Zero(p_optimize->getSimuFuncSize(),
60         simulator->getNbSimul());
61     for (int istep = 0; istep < nbStep; ++istep)
62     {
63         StOpt::SimulateStepRegressionCutDist(ar, nbStep - 1 - istep, nameAr, p_grid,
64             p_optimize, p_bOneFile, p_world).oneStep(states, costFunction);
65
66         // new stochastic state
67         Eigen::ArrayXXd particles = simulator->stepForwardAndGetParticles();
68         for (int is = 0; is < simulator->getNbSimul(); ++is)
69             states[is].setStochasticRealization(particles.col(is));
70     }
71     // final : accept to exercise if not already done entirely (here suppose one function
72     // to follow)
73     for (int is = 0; is < simulator->getNbSimul(); ++is)
74         costFunction(0, is) += p_funcFinalValue(states[is].getRegime(), states[is].
75             getPtStock(), states[is].getStochasticRealization())(0);
76
77     return costFunction.mean();
78 }
79
80 #endif /* SIMULATEREGRESSIONCUTDIST_H */

```

The version of the previous example using a single archive storing the control/solution is given in `SimulateRegressionCut.h` file.

4.3 Solve the problem for $X_2^{x,t}$ stochastic

In this part, we assume that $X_2^{x,t}$ is controlled but is stochastic.

4.3.1 Requirement to use the framework

To use the framework, a business object describing the business at a time step from a state is derived from `OptimizerNoRegressionDPBase` itself derived from `OptimizerBase`.

```

1 // Copyright (C) 2016 EDF
2 // All Rights Reserved
3 // This code is published under the GNU Lesser General Public License (GNU LGPL)
4 #ifndef OPTIMIZERNOREGRESSIONDPBASE_H
5 #define OPTIMIZERNOREGRESSIONDPBASE_H
6 #include <Eigen/Dense>
7 #include "StOpt/core/utils/StateWithStocks.h"
8 #include "StOpt/core/grids/SpaceGrid.h"
9 #include "StOpt/regression/BaseRegression.h"
10 #include "StOpt/regression/GridAndRegressedValue.h"
11 #include "StOpt/dp/SimulatorDPBase.h"
12 #include "StOpt/dp/OptimizerBaseInterp.h"
13
14 /** \file OptimizerNoRegressionDPBase.h

```

```

15  * \brief Define an abstract class for Dynamic Programming problems solve by Monte Carlo
    * but without regression method
16  *         to compute conditional expectation.
17  *         \author Xavier Warin
18  */
19
20 namespace StOpt
21 {
22
23     /// \class OptimizerNoRegressionDPBase OptimizerNoRegressionDPBase.h
24     /// Base class for optimizer for Dynamic Programming solved without regression method to
    compute conditional expectation.
25     class OptimizerNoRegressionDPBase : public OptimizerBaseInterp
26     {
27
28     public :
29
30         OptimizerNoRegressionDPBase() {}
31
32         virtual ~OptimizerNoRegressionDPBase() {}
33
34         /// \brief defines the diffusion cone for parallelism
35         /// \param p_regionByProcessor region (min max) treated by the processor for
    the different regimes treated
36         /// \return returns in each dimension the min max values in the stock that can be
    reached from the grid p_gridByProcessor for each regime
37         virtual std::vector< std::array< double, 2> > getCone(const std::vector< std::array<
    double, 2> > &p_regionByProcessor) const = 0;
38
39         /// \brief defines the dimension to split for MPI parallelism
40         /// For each dimension return true is the direction can be split
41         virtual Eigen::Array< bool, Eigen::Dynamic, 1> getDimensionToSplit() const = 0 ;
42
43         /// \brief defines a step in optimization
44         /// \param p_stock coordinates of the stock point to treat
45         /// \param p_valNext Optimized values at next time step for each regime
46         /// \param p_regressorCur Regressor at the current date
47         /// \return a pair :
48         /// - for each regimes (column) gives the solution for each particle (row)
49         /// - for each control (column) gives the optimal control for each
    particle (rows)
50         ///
51         virtual std::pair< Eigen::ArrayXXd, Eigen::ArrayXXd> stepOptimize(const Eigen::
    ArrayXd &p_stock,
52         const std::vector< GridAndRegressedValue > &p_valNext,
53         std::shared_ptr< BaseRegression > p_regressorCur) const = 0;
54
55
56
57
58         /// \brief Defines a step in simulation using interpolation in controls
59         /// \param p_grid grid at arrival step after command
60         /// \param p_control defines the controls
61         /// \param p_state defines the state value (modified)
62         /// \param p_phiInOut defines the value function (modified): size number of
    functions to follow
63         virtual void stepSimulateControl(const std::shared_ptr< StOpt::SpaceGrid> &p_grid,
    const std::vector< StOpt::GridAndRegressedValue > &p_control,
64         StOpt::StateWithStocks<double> &p_state,
65         Eigen::Ref<Eigen::ArrayXd> p_phiInOut) const = 0 ;
66
67
68
69         /// \brief Get the number of regimes allowed for the asset to be reached at the
    current time step
70         /// If \f$ t \f$ is the current time, and \f$ dt \f$ the resolution step, this is
    the number of regime allowed on \f$[ t- dt, t[\f$
71         virtual int getNbRegime() const = 0 ;
72

```

```

73  /// \brief get the simulator back
74  virtual std::shared_ptr< StOpt::SimulatorDPBase > getSimulator() const = 0;
75
76  /// \brief get back the dimension of the control
77  virtual int getNbControl() const = 0 ;
78
79  /// \brief get size of the function to follow in simulation
80  virtual int getSimuFuncSize() const = 0;
81
82 };
83 }
84 #endif /* OPTIMIZERDPBASE_H */

```

In addition to the `OptimizerBase` methods, the following method is required:

- the `stepOptimize` method is used in optimization. We want to calculate the optimal value regressed to t_i current at a grid point `p_stock` using a `p_grid` grid at the following date t_{i+1} ,

From a `p_stock` grid point, it calculates the regressed function values and the regressed optimal controls. It returns a pair where the

- the first element is a matrix (the first dimension is the number of functions in the regression, the second dimension the number of regimes) giving the value of the regressed function,
- the second element is a matrix (the first dimension is the number of functions in the regression, the second dimension the number of controls) giving the optimal regressed control.

In this case of the optimization of an updated portfolio with dynamic:

$$dX_2^{x,t} = X_2^{x,t} \frac{dX_1^{x,t}}{X_1^{x,t}}$$

where $X_1^{x,t}$ is the value of the risky asset, the `Optimize` object is given in the file `OptimizePortfolio.h`.

4.3.2 The framework in optimization

Once an `Optimizer` is derived for the project, and assuming a full grid is used for stock discretization, the framework provides a `TransitionStepDPDist` object in MPI which solves the optimization problem with the distribution of data over a time step with the following constructor:

```

1  TransitionStepDPDist(const shared_ptr<FullGrid> &p_pGridCurrent ,
2  const shared_ptr<FullGrid> &p_pGridPrevious ,
3  const std::shared_ptr<BaseRegression> &p_regressorCurrent ,
4  const std::shared_ptr<BaseRegression> &p_regressorPrevious ,
5  const shared_ptr<OptimizerNoRegressionDPBase > &p_pOptimize ,
6  const boost::mpi::communicator &p_world):

```

with

- `p_pGridCurrent` is the grid at the current time step (t_i),

- `p_pGridPrevious` is the grid at the previously processed time step (t_{i+1}),
- `p_regressorCurrent` is a regressor at the current date (to evaluate the function at the current date)
- `p_regressorPrevious` is a previously processed time step regressor (t_{i+1}) allowing to evaluate a function at date t_{i+1} ,
- `p_pOptimize` the optimizer object
- `p_world` The MPI communicator

Remark 30 *A similar object is available without the MPI distribution framework `TransitionStepDP` with always the activation of parallelization with threads and MPI on the calculations on the full grid points.*

Remark 31 *The case of sparse grids is not currently dealt with in the framework.*

The main method is

```
1  std::pair< std::shared_ptr< std::vector< Eigen::ArrayXXd > > , std::shared_ptr< std::vector< Eigen::ArrayXXd > > > oneStep(const std::vector< Eigen::ArrayXXd > & p_phiIn)
```

with

- `p_phiIn` the vector (its size corresponds to the number of regimes) of the matrix of calculated optimal values regressed at the previous time iteration for each regime. Each matrix is a number of regressor function on the previous date by number of matrix stock points.

return a pair:

- the first element is a vector of matrix with new optimal values regressed at the current time step (each element of the vector corresponds to a regime and each matrix is a number of functions regressed at the current date by the number of stock points of the matrix).
- the second element is a vector of matrix with new optimal controls regressed at the current time step (each element of the vector corresponds to a control and each matrix is a number of controls regressed by the number of matrix stock points).

Remark 32 *All `TransitionStepDP` derive from a `TransitionStepBase` object with a pure virtual `OneStep` method.*

A second method is provided allowing to dump the optimal control at each time step:

```
1  void dumpValues(std::shared_ptr<gs::BinaryFileArchive> p_ar ,
2                const std::string &p_name, const int &p_iStep,
3                const std::vector< Eigen::ArrayXXd > &p_control, const bool &p_bOneFile) const
```

with:

- `p_ar` is the archive where controls and solutions are dumped,
- `p_name` is a base name used in the archive to store the solution and the control,
- `p_control` stores the optimal controls calculated at the current time step,
- `p_bOneFile` is set to one if the optimal controls calculated by each processor are dumped on a single file. Otherwise the optimal controls calculated by each processor are dumped on different files (one by processor). If the problem gives optimal control values on the global grid that can be stored in the memory of the computation node, it can be more interesting to dump the control values in one file for the simulation of the optimal policy.

Remark 33 *As for the `TransitionStepDP`, its `dumpValues` does not need a `p_bOneFile` argument: obviously optimal controls are stored in a single file.*

Here we give a simple example of temporal resolution using this method when MPI data distribution is used

```

1 // Copyright (C) 2016 EDF
2 // All Rights Reserved
3 // This code is published under the GNU Lesser General Public License (GNU LGPL)
4 #ifdef USE_MPI
5 #include <fstream>
6 #include <boost/mpi.hpp>
7 #include <memory>
8 #include <functional>
9 #include <boost/lexical_cast.hpp>
10 #include <Eigen/Dense>
11 #include "geners/BinaryFileArchive.hh"
12 #include "StOpt/core/grids/FullGrid.h"
13 #include "StOpt/regression/LocalConstRegression.h"
14 #include "StOpt/regression/GridAndRegressedValue.h"
15 #include "StOpt/dp/FinalStepDPDist.h"
16 #include "StOpt/dp/TransitionStepDPDist.h"
17 #include "StOpt/core/parallelism/reconstructProcOMpi.h"
18 #include "test/c++/tools/dp/OptimizePortfolioDP.h"
19
20 using namespace std;
21 using namespace Eigen;
22
23 double DynamicProgrammingPortfolioDist(const shared_ptr<StOpt::FullGrid> &p_grid,
24                                       const shared_ptr<OptimizePortfolioDP> &p_optimize,
25                                       const ArrayXi &p_nbMesh,
26                                       const function<double(const int &, const ArrayXd &,
27                                                             const ArrayXd &)> &p_funcFinalValue,
28                                       const ArrayXd &p_initialPortfolio,
29                                       const string &p_fileToDump,
30                                       const bool &p_bOneFile,
31                                       const boost::mpi::communicator &p_world
32 ) {
33     // initialize simulation
34     p_optimize->initializeSimulation();
35     // store regressor
36     shared_ptr<StOpt::LocalConstRegression> regressorPrevious;
37
38     // store final regressed values in object valuesStored
39     shared_ptr<vector<ArrayXXd>> valuesStored = make_shared<vector<ArrayXXd>>(<
40         p_optimize->getNbRegime());

```

```

41     vector< shared_ptr< ArrayXXd > > valuesPrevious = StOpt::FinalStepDPDist(p_grid,
        p_optimize->getNbRegime(), p_optimize->getDimensionToSplit(), p_world)(
        p_funcFinalValue, *p_optimize->getCurrentSim());
42     // regressor operator
43     regressorPrevious = make_shared<StOpt::LocalConstRegression>(false, *p_optimize->
        getCurrentSim(), p_nbMesh);
44     for (int iReg = 0; iReg < p_optimize->getNbRegime(); ++iReg)
45         (*valuesStored)[iReg] = regressorPrevious->getCoordBasisFunctionMultiple(
            valuesPrevious[iReg]->transpose()).transpose();
46 }
47 string toDump = p_fileToDump ;
48 // test if one file generated
49 if (!p_bOneFile)
50     toDump += "_" + boost::lexical_cast<string>(p_world.rank());
51 shared_ptr<gs::BinaryFileArchive> ar;
52 if ((!p_bOneFile) || (p_world.rank() == 0))
53     ar = make_shared<gs::BinaryFileArchive>(toDump.c_str(), "w");
54 // name for object in archive
55 string nameAr = "OptimizePort";
56 // iterate on time steps
57 for (int iStep = 0; iStep < p_optimize->getNbStep(); ++iStep)
58 {
59     // step backward for simulations
60     p_optimize->oneStepBackward();
61     // create regressor at the given date
62     bool bZeroDate = (iStep == p_optimize->getNbStep() - 1);
63     shared_ptr<StOpt::LocalConstRegression> regressorCur = make_shared<StOpt::
        LocalConstRegression>(bZeroDate, *p_optimize->getCurrentSim(), p_nbMesh);
64     // transition object
65     StOpt::TransitionStepDPDist transStep(p_grid, p_grid, regressorCur,
        regressorPrevious, p_optimize, p_world);
66     pair< shared_ptr< vector< ArrayXXd > >, shared_ptr< vector< ArrayXXd > > >
        valuesAndControl = transStep.oneStep(*valuesStored);
67     // dump control values
68     transStep.dumpValues(ar, nameAr, iStep, *valuesAndControl.second, p_bOneFile);
69     valuesStored = valuesAndControl.first;
70     // shift regressor
71     regressorPrevious = regressorCur;
72 }
73 // interpolate at the initial stock point and initial regime( 0 here) (take first
    particle)
74 shared_ptr<ArrayXXd> topRows = make_shared<ArrayXXd>((*valuesStored)[0].topRows(1));
75 return StOpt::reconstructProcOMpi(p_initialPortfolio, p_grid, topRows, p_optimize->
    getDimensionToSplit(), p_world).mean();
76 }
77 #endif

```

An example without data distribution can be found in the file `DynamicProgrammingPortfolio.lio.cpp`.

4.3.3 The simulation framework

No special framework is available in simulation. Use the `SimulateStepRegressionControl` or `SimulateStepRegressionControlDist` function described in the 4.2.1 section.

4.4 Solving stock problems with trees

In this section we detail how to solve problems

- Either by discretizing the control,
- Or by solving a Linear Program problem approaching Bellman values by cuts.

4.4.1 Resolution of dynamic programming problems with the discretization of commands

Framework usage condition

The OptimizerDPTreeBase is the base object from which each business object is to be derived.

```
1 // Copyright (C) 2019 EDF
2 // All Rights Reserved
3 // This code is published under the GNU Lesser General Public License (GNU LGPL)
4 #ifndef OPTIMIZERDPTREEBASE_H
5 #define OPTIMIZERDPTREEBASE_H
6 #include <Eigen/Dense>
7 #include "StOpt/core/grids/SpaceGrid.h"
8 #include "StOpt/tree/Tree.h"
9 #include "StOpt/tree/StateTreeStocks.h"
10 #include "StOpt/tree/ContinuationValueTree.h"
11 #include "StOpt/tree/GridTreeValue.h"
12 #include "StOpt/dp/SimulatorDPBaseTree.h"
13 #include "StOpt/dp/OptimizerBase.h"
14
15 /** \file OptimizerDPTreeBase.h
16  * \brief Define an abstract class for Dynamic Programming problems solved by tree
17  * \author Xavier Warin
18  */
19
20 namespace StOpt
21 {
22
23     /// \class OptimizerDPTreeBase OptimizerDPTreeBase.h
24     /// Base class for optimizer for Dynamic Programming with tree methods
25     class OptimizerDPTreeBase : public OptimizerBase
26     {
27
28     public :
29
30         OptimizerDPTreeBase() {}
31
32         virtual ~OptimizerDPTreeBase() {}
33
34         /// \brief defines the diffusion cone for parallelism
35         /// \param p_regionByProcessor region (min max) treated by the processor for
36         /// the different regimes treated
37         /// \return returns in each dimension the min max values in the stock that can be
38         /// reached from the grid p_gridByProcessor for each regime
39         virtual std::vector< std::array< double, 2> > getCone(const std::vector< std::array<
40         double, 2> > &p_regionByProcessor) const = 0;
41
42         /// \brief defines the dimension to split for MPI parallelism
43         /// For each dimension return true is the direction can be split
44         virtual Eigen::Array< bool, Eigen::Dynamic, 1> getDimensionToSplit() const = 0 ;
45
46         /// \brief defines a step in optimization
47         /// \param p_grid grid at arrival step after command
48         /// \param p_stock coordinates of the stock point to treat
49         /// \param p_condEsp continuation values for each regime
50         /// \return a pair :
51         /// - for each regimes (column) gives the solution for each node in the
52         /// tree (row)
53         /// - for each control (column) gives the optimal control for each node in
54         /// the tree (rows)
55         /// .
56         virtual std::pair< Eigen::ArrayXXd, Eigen::ArrayXXd> stepOptimize(const std::
57         shared_ptr< StOpt::SpaceGrid> &p_grid, const Eigen::ArrayXd &p_stock,
```

```

53         const std::vector< StOpt::ContinuationValueTree > &p_condEsp) const = 0;
54
55
56     /// \brief defines a step in simulation
57     /// Notice that this implementation is not optimal but is convenient if the control is
58     /// discrete.
59     /// By avoiding interpolation in control we avoid non admissible control
60     /// Control are recalculated during simulation.
61     /// \param p_grid          grid at arrival step after command
62     /// \param p_continuation  defines the continuation operator for each regime
63     /// \param p_state         defines the state value (modified)
64     /// \param p_phiInOut      defines the value functions (modified) : size number of
65     ///                        functions to follow
66     virtual void stepSimulate(const std::shared_ptr< StOpt::SpaceGrid> &p_grid, const std
67     ::vector< StOpt::GridTreeValue > &p_continuation,
68     StOpt::StateTreeStocks &p_state,
69     Eigen::Ref<Eigen::ArrayXd> p_phiInOut) const = 0 ;
70
71     /// \brief Defines a step in simulation using interpolation in controls
72     /// \param p_grid          grid at arrival step after command
73     /// \param p_control       defines the controls
74     /// \param p_state         defines the state value (modified)
75     /// \param p_phiInOut      defines the value function (modified): size number of
76     ///                        functions to follow
77     virtual void stepSimulateControl(const std::shared_ptr< StOpt::SpaceGrid> &p_grid,
78     const std::vector< StOpt::GridTreeValue > &p_control,
79     StOpt::StateTreeStocks &p_state,
80     Eigen::Ref<Eigen::ArrayXd> p_phiInOut) const = 0 ;
81
82     /// \brief Get the number of regimes allowed for the asset to be reached at the
83     /// current time step
84     /// If \f$ t \f$ is the current time, and \f$ dt \f$ the resolution step, this is
85     /// the number of regime allowed on \f$[ t- dt, t[\f$
86     virtual int getNbRegime() const = 0 ;
87
88     /// \brief get the simulator back
89     virtual std::shared_ptr< StOpt::SimulatorDPBaseTree > getSimulator() const = 0;
90
91     /// \brief get back the dimension of the control
92     virtual int getNbControl() const = 0 ;
93
94     /// \brief get size of the function to follow in simulation
95     virtual int getSimuFuncSize() const = 0;
96 };
97
98 #endif /* OPTIMIZERDPTREEBASE_H */

```

Since the `OptimizerDPTreeBase` class is derived from `OptimizerBase`, we detail the additional methods required:

- the `stepOptimize` method is used in optimization. We want to calculate the optimal value at the current t_i at a grid point `p_stock` using a `p_grid` grid at the following date t_{i+1} , the continuation values for all the `p_condEsp` regimes making it possible to calculate the conditional expectation of the optimal value function calculated at the previously processed time step t_{i+1} . From a grid point `p_stock` it calculates the function values and the optimal controls. It returns a pair where the
 - the first element is a matrix (the first dimension is the number of nodes, the second dimension the number of regimes) giving the value of the function,

- the second element is a matrix (the first dimension is the number of nodes, the second dimension the number of controls) giving the optimal control.
- the `stepSimulate` method is used after the optimization using the continuation values calculated in the optimization part: these continuation values are stored in a `GridTreeValue` object for interpolation. From a `p_state` state (storing the $X^{x,t}$), the continuation values calculated in the optimization `p_continuation`, the optimal values of the stored functions in `p_phiInOut`.
- the `stepSimulateControl` simulates the strategy by direct interpolation of the control for a given node in the tree (sampled) and a position in the stock.

The framework

Most of the objects used with regression have their counterparts with tree methods.

In optimization:

`TransitionStepTreeDPDist` resolves the transition to the date for all grid point and tree nodes using the distribution (MPI).

```
1 TransitionStepTreeDPDist(const std::shared_ptr<FullGrid> &p_pGridCurrent,
2                          const std::shared_ptr<FullGrid> &p_pGridPrevious,
3                          const std::shared_ptr<OptimizerDPTreeBase> &p_pOptimize,
4                          const boost::mpi::communicator &p_world)
```

with

- `p_pGridCurrent` is the grid at the current time step (t_i),
- `p_pGridPrevious` is the grid at the previously processed time step (t_{i+1}),
- `p_pOptimize` the optimizer object
- `p_world` The MPI communicator

A similar object is available without the MPI distribution framework `TransitionStepTreeDP` with always the activation of parallelization with threads and MPI on calculations on the full grid points.

The main method is

```
1 std::pair< std::vector< std::shared_ptr< Eigen::ArrayXXd > >, std::vector< std::shared_ptr< Eigen::ArrayXXd > > >
2   oneStep(const std::vector< std::shared_ptr< Eigen::ArrayXXd > > &p_phiIn,
3          const std::shared_ptr< Tree> &p_condExp) const
```

with

- `p_phiIn` the vector (its size corresponds to the number of regimes) of the matrix of optimal values calculated at the previous time iteration for each regime. Each matrix is a number of nodes the next date per number of stock points matrix.
- `p_condExp` the conditional expectation operator,

return a pair:

- first element is a vector of matrix with new optimal values at the current time step (each element of the vector corresponds to a regime and each matrix is a number of nodes at current date by number of points of stock matrix).
- The second element is a vector of matrix with new optimal controls at the current time step (each element of the vector corresponds to a control and each matrix is a number of nodes at the current date by number of points of stock matrix).

A second method is provided allowing to dump the continuation values of the problem and the optimal control at each time step:

```

1 void dumpContinuationValues(std::shared_ptr<gs::BinaryFileArchive> p_ar,
2                             const std::string &p_name, const int &p_iStep,
3                             const std::vector< std::shared_ptr< Eigen::ArrayXXd > > &
4                             p_phiIn,
5                             const std::vector< std::shared_ptr< Eigen::ArrayXXd > > &
6                             p_control,
7                             const std::shared_ptr< Tree> &p_tree,
8                             const bool &p_bOneFile) const;

```

with:

- `p_ar` is the archive where the controls and solutions are dumped,
- `p_name` is a base name used in the archive to store the solution and the control,
- `p_phiInPrev` is the previous time step solution used to calculate the continuation values that are stored,
- `p_control` stores the optimal controls calculated at the current time step,
- `p_tree` is the conditional expectation object allowing to calculate the conditional expectation of the functions defined at the previous time step processed `p_phiInPrev` and allowing to store a representation of the optimal control.
- `p_bOneFile` is set to one if the continuation and optimal controls calculated by each processor are dumped on a single file. Otherwise the continuation and optimal controls calculated by each processor are dumped on different files (one per processor).

Remark 34 *The `p_bOneFile` is not present for `TransitionStepTreeDP` objects.*

In simulation (see detail in section for regressions)

- A first object allowing the recalculation of the optimal control in simulation.

```

1 SimulateStepTreeDist(gs::BinaryFileArchive &p_ar, const int &p_iStep, const std::
2 string &p_nameCont,
3 const std::shared_ptr<FullGrid> &p_pGridFollowing, const
4 std::shared_ptr<OptimizerDPTreeBase > &p_pOptimize,
5 const bool &p_bOneFile,
6 const boost::mpi::communicator &p_world)

```

where

- `p_ar` is the binary archive where the continuation values are stored,

- `p_iStep` is the number associated with the current time step (0 at the start date of the simulation, the number is increased by one at each simulated time step),
- `p_nameCont` is the base name for continuation values,
- `p_GridFollowing` is the grid at the next time step (`p_iStep+1`),
- `p_Optimize` the Optimizer describing the transition from one time step to the next,
- `p_OneFile` equal to `true` if only one archive is used to store continuation values.
- `p_world` The MPI communicator

This object implements the method `oneStep`

```
1 void oneStep(std::vector<StateTreeStocks > &p_statevector, Eigen::ArrayXXd &
   p_phiInOut) const
```

where:

- `p_statevector` stores the states of all the simulations: this state is updated by applying the optimal command,
- `p_phiInOut` stores the gain/cost functions for all simulations: it is updated by the function call. The size of the array is $(nb, nbSimul)$ where nb is given by the `getSimuFuncSize` method of the optimizer and $nbSimul$ the number of Monte Carlo simulations.

Remark 35 *The version without distribution of Bellman values is available in the object `SimulateStepTree`.*

- A second object directly interpolating the control

```
1 SimulateStepTreeControlDist(gs::BinaryFileArchive &p_ar, const int &p_iStep, const
   std::string &p_nameCont,
2                                     const std::shared_ptr<FullGrid> &p_pGridCurrent,
3                                     const std::shared_ptr<FullGrid> &p_pGridFollowing,
4                                     const std::shared_ptr<OptimizerDPTreeBase > &
   p_pOptimize,
5                                     const bool &p_bOneFile,
6                                     const boost::mpi::communicator &p_world);
```

where

- `p_ar` is the binary archive where the continuation values are stored,
- `p_iStep` is the number associated with the current time step (0 at the start date of the simulation, the number is increased by one at each simulated time step),
- `p_nameCont` is the base name of the control values,
- `p_GridCurrent` is the grid at the current time step (`p_iStep`),
- `p_GridFollowing` is the grid at the next time step (`p_iStep+1`),
- `p_Optimize` is the Optimizer describing the transition from one time step to the next,

- `p_OneFile` is equal to `true` if only one archive is used to store continuation values.
- `p_world` The MPI communicator

This object implements the method `oneStep`

```
1 void oneStep(std::vector<StateTreeStocks > &p_statevector, Eigen::ArrayXXd &
    p_phiInOut) const;
```

where:

- `p_statevector` stores the state for all the simulations : this state is updated by applying optimal commands,
- `p_phiInOut` stores the gain/cost functions for all simulations: it is updated by the function call. The size of the array is $(nb, nbSimul)$ where nb is given by the `getSimuFuncSize` method of the optimizer and $nbSimul$ the number of Monte Carlo simulations.

Remark 36 *The undistributed version is given by the object `SimulateStepTreeControl`.*

4.4.2 Solving Dynamic Programming by solving LP problems

This approach can only be used for continuous problems with convex or concave Bellman values. See section 4.2.2 for an explanation of the cut approximation.

Framework usage requirement

The expanded business object must derive from the `OptimizerDPCutBase` object derived from the `OptimizerBase` object.

```
1 // Copyright (C) 2019 EDF
2 // All Rights Reserved
3 // This code is published under the GNU Lesser General Public License (GNU LGPL)
4 #ifndef OPTIMIZERDPCUTTREEBASE_H
5 #define OPTIMIZERDPCUTTREEBASE_H
6 #include <Eigen/Dense>
7 #include "StOpt/core/utils/StateWithStocks.h"
8 #include "StOpt/core/grids/SpaceGrid.h"
9 #include "StOpt/tree/Tree.h"
10 #include "StOpt/tree/StateTreeStocks.h"
11 #include "StOpt/tree/ContinuationCutsTree.h"
12 #include "StOpt/dp/SimulatorDPBaseTree.h"
13 #include "StOpt/dp/OptimizerBase.h"
14
15 /** \file OptimizerDPCutBase.h
16  * \brief Define an abstract class for Dynamic Programming problems solved by tree
17  *       methods using cust to approximate
18  *       Bellman values
19  *       \author Xavier Warin
20  */
21 namespace StOpt
22 {
23
24 /// \class OptimizerDPCutTreeBase OptimizerDPCutTreeBase.h
25 /// Base class for optimizer for Dynamic Programming with tree methods and cuts, so using
    LP to solve transitional problems
```

```

26 class OptimizerDPCutTreeBase : public OptimizerBase
27 {
28
29
30 public :
31
32     OptimizerDPCutTreeBase() {}
33
34     virtual ~OptimizerDPCutTreeBase() {}
35
36     /// \brief defines the diffusion cone for parallelism
37     /// \param p_regionByProcessor      region (min max) treated by the processor for
38     /// the different regimes treated
39     /// \return returns in each dimension the min max values in the stock that can be
40     /// reached from the grid p_gridByProcessor for each regime
41     virtual std::vector< std::array< double, 2> > getCone(const std::vector< std::array<
42     double, 2> > &p_regionByProcessor) const = 0;
43
44     /// \brief defines the dimension to split for MPI parallelism
45     /// For each dimension return true is the direction can be split
46     virtual Eigen::Array< bool, Eigen::Dynamic, 1> getDimensionToSplit() const = 0 ;
47
48     /// \brief defines a step in optimization
49     /// \param p_grid      grid at arrival step after command
50     /// \param p_stock      coordinates of the stock point to treat
51     /// \param p_condEsp    continuation values for each regime
52     /// \return      For each regimes (column) gives the solution for each particle , and cut
53     /// (row)
54     /// For a given simulation , cuts components (C) at a point stock  $\bar{S}$ 
55     ///  $\{f\}$  are given such that the cut is given by
56     ///  $\{f\} C[0] + \sum_{i=1}^d C[i] (S_i - \bar{S}_i) \{f\}$ 
57     virtual Eigen::ArrayXXd stepOptimize(const std::shared_ptr< StOpt::SpaceGrid> &
58     p_grid, const Eigen::ArrayXd &p_stock,
59     const std::vector< StOpt::ContinuationCutsTree
60     > &p_condEsp) const = 0;
61
62     /// \brief defines a step in simulation
63     /// Control are recalculated during simulation using a local optimization using the LP
64     /// \param p_grid      grid at arrival step after command
65     /// \param p_continuation defines the continuation operator for each regime
66     /// \param p_state      defines the state value (modified)
67     /// \param p_phiInOut    defines the value functions (modified) : size number of
68     /// functions to follow
69     virtual void stepSimulate(const std::shared_ptr< StOpt::SpaceGrid> &p_grid, const std
70     ::vector< StOpt::ContinuationCutsTree > &p_continuation,
71     StOpt::StateTreeStocks &p_state,
72     Eigen::Ref<Eigen::ArrayXd> p_phiInOut) const = 0 ;
73
74     /// \brief Get the number of regimes allowed for the asset to be reached at the
75     /// current time step
76     /// If  $t$  is the current time, and  $dt$  the resolution step, this is
77     /// the number of regime allowed on  $[t-dt, t]$ 
78     virtual int getNbRegime() const = 0 ;
79
80     /// \brief get the simulator back
81     virtual std::shared_ptr< StOpt::SimulatorDPBaseTree > getSimulator() const = 0;
82
83     /// \brief get back the dimension of the control
84     virtual int getNbControl() const = 0 ;
85
86     /// \brief get size of the function to follow in simulation
87     virtual int getSimuFuncSize() const = 0;
88
89 };
90
91 #endif /* OPTIMIZERDPCUTTREEBASE_H */

```

We detail the different methods to implement in addition to the `OptimizerBase` methods:

- the `stepOptimize` method is used in optimization. We want to calculate the optimal value and the corresponding sensitivities with respect to stocks at t_i current at a grid point `p_stock` using a grid `p_grid` at the next date t_{i+1} , the continuation cuts the values for all the regimes `p_condEsp` allowing to calculate a upper estimate (during the maximization) of the conditional expectation of the optimal values by using an optimization calculated at the previously treated time step t_{i+1} . From a `p_stock` grid point, it calculates the function values and the corresponding sensitivities. It returns a matrix (the first dimension is the number of nodes at the current cate by the number of components cuts (number of storage + 1), the second dimension the number of regimes) giving the value of the function and sensitivities.
- the `stepSimulate` method is used after the optimization using the continuation cuts the values calculated in the optimization part. From a `p_state` state (storing the $X^{x,t}$), the sequence cuts the calculated values in optimization `p_continuation`, the optimal cash flows are stored in `p_phiInOut`.

In the case of a gas storage the `Optimize` object is given in the file `OptimizeGasStorageTreeCut.h`.

The framework in optimization using certain cutting methods

We treat it exactly as in 4.2.2 section. The framework provides an `TransitionStepTreeDPCutDist` object in MPI which solves the optimization problem with data distribution over a time step with the following constructor:

```

1  TransitionStepTreeDPCutDist(const  std::shared_ptr<FullGrid> &p_pGridCurrent,
2                                const  std::shared_ptr<FullGrid> &p_pGridPrevious,
3                                const  std::shared_ptr<OptimizerDPCutTreeBase > &
4                                p_pOptimize,
                                const  boost::mpi::communicator &p_world);

```

with

- `p_pGridCurrent` is the grid at the current time step (t_i),
- `p_pGridPrevious` is the grid at the previously processed time step (t_{i+1}),
- `p_pOptimize` the optimizer object
- `p_world` The MPI communicator

The construction is very similar to the classical regression methods using only the discretization by command.

Remark 37 *A similar object is available without the MPI distribution framework `TransitionStepTreeDPCut` with always the activation of parallelization with threads and MPI on calculations on the full points grid .*

The main method is

```

1  std::vector< std::shared_ptr< Eigen::ArrayXXd > > oneStep(const std::vector< std::
   shared_ptr< Eigen::ArrayXXd > > &p_phiIn,
2  const std::shared_ptr< Tree> &p_condExp) const

```

with

- **p_phiIn** the vector (its size corresponds to the number of regimes) of the matrix of optimal values and sensitivities calculated at the previous time iteration for each regime. Each matrix has a number of rows equal to the number of nodes on the next date by the number of stock plus one. The number of columns is equal to the number of stock points on the grid. In the row, the number of simulations by the number of stock plus one value is stored as follows:
 - The first values (number of nodes on the following date: NS) correspond to the optimal Bellman values at a given stock point,
 - The NS values following corresponds to sensitivities $\frac{\partial V}{\partial S_1}$ to first storage
 - The NS values following corresponds to sensitivities to the second storage...
 - ...
- **p_condExp** the conditional expectation operator,

returning a vector of matrix with new optimal values and sensitivities at the current time step (each element of the vector corresponds to a regime and each matrix has a size equal to (number of nodes at the current date by (the number of storage plus one)) by the number of stock points). The structure of the output is then similar to the **p_phiIn** input.

A second method is provided allowing to dump the continuation values and the cuts of the problem and the optimal control at each time step:

```

1  void dumpContinuationCutsValues(std::shared_ptr<gs::BinaryFileArchive> p_ar, const std
   ::string &p_name, const int &p_iStep,
2  const std::vector< std::shared_ptr< Eigen::ArrayXXd > >
   &p_phiInPrev, const std::shared_ptr< Tree> &
   p_condExp,
3  const bool &p_bOneFile) const

```

with:

- **p_ar** is the archive where controls and solutions are dumped,
- **p_name** is a base name used in the archive to store the solution and the control,
- **p_phiInPrev** is the solution to the previous time step used to calculate the values of continuation cuts which are stored,
- **p_condExp** is the conditional expectation object allowing to calculate the conditional expectation of the functions defined at the preceding time step processed **p_phiInPrev**.
- **p_bOneFile** is set to one if the continuation cuts values calculated by each processor are dumped on a single file. Otherwise the continuation cuts values calculated by each processor are dumped on different files (one per processor).

Here we give a simple example of a temporal resolution using this method when MPI data distribution is used

```

1 // Copyright (C) 2019 EDF
2 // All Rights Reserved
3 // This code is published under the GNU Lesser General Public License (GNU LGPL)
4 #ifdef USE_MPI
5 #include <fstream>
6 #include <memory>
7 #include <functional>
8 #include <boost/lexical_cast.hpp>
9 #include <boost/mpi.hpp>
10 #include <Eigen/Dense>
11 #include "geners/BinaryFileArchive.hh"
12 #include "StOpt/core/grids/FullGrid.h"
13 #include "StOpt/tree/Tree.h"
14 #include "StOpt/dp/FinalStepDPCutDist.h"
15 #include "StOpt/dp/TransitionStepTreeDPCutDist.h"
16 #include "StOpt/core/parallelism/reconstructProcOMpi.h"
17 #include "StOpt/dp/OptimizerDPCutTreeBase.h"
18 #include "StOpt/dp/SimulatorDPBaseTree.h"
19
20
21 using namespace std;
22 using namespace Eigen;
23
24 double DynamicProgrammingByTreeCutDist(const shared_ptr<StOpt::FullGrid> &p_grid,
25                                         const shared_ptr<StOpt::OptimizerDPCutTreeBase> &
26                                         p_optimize,
27                                         const function<ArrayXd(const int &, const ArrayXd
28                                         &, const ArrayXd &>> &p_funcFinalValue,
29                                         const ArrayXd &p_pointStock,
30                                         const int &p_initialRegime,
31                                         const string &p_fileToDump,
32                                         const bool &p_bOneFile,
33                                         const boost::mpi::communicator &p_world)
34 {
35     // from the optimizer get back the simulator
36     shared_ptr<StOpt::SimulatorDPBaseTree> simulator = p_optimize->getSimulator();
37     // final values
38     vector< shared_ptr< ArrayXXd > > valueCutsNext = StOpt::FinalStepDPCutDist(p_grid,
39                                         p_optimize->getNbRegime(), p_optimize->getDimensionToSplit(), p_world)(
40                                         p_funcFinalValue, simulator->getNodes());
41
42     // dump
43     string toDump = p_fileToDump ;
44     // test if one file generated
45     if (!p_bOneFile)
46         toDump += "_" + boost::lexical_cast<string>(p_world.rank());
47     shared_ptr<gs::BinaryFileArchive> ar;
48     if ((!p_bOneFile) || (p_world.rank() == 0))
49         ar = make_shared<gs::BinaryFileArchive>(toDump.c_str(), "w");
50     // name for object in archive
51     string nameAr = "ContinuationTree";
52     for (int iStep = 0; iStep < simulator->getNbStep(); ++iStep)
53     {
54         simulator->stepBackward();
55         // probabilities
56         std::vector<double> proba = simulator->getProba();
57         // get connection between nodes
58         std::vector< std::vector<std::array<int, 2> > > > connected = simulator->
59             getConnected();
60         // conditional expectation operator
61         shared_ptr<StOpt::Tree> tree = std::make_shared<StOpt::Tree>(proba, connected);
62         // transition object
63         StOpt::TransitionStepTreeDPCutDist transStep(p_grid, p_grid, p_optimize, p_world);
64         vector< shared_ptr< ArrayXXd > > valueCuts = transStep.oneStep(valueCutsNext, tree
65         );
66         transStep.dumpContinuationCutsValues(ar, nameAr, iStep, valueCutsNext, tree,

```

```

        p_bOneFile);
        valueCutsNext = valueCuts;
60     }
61     // reconstruct a small grid for interpolation
62     ArrayXd valSim = StOpt::reconstructProc0Mpi(p_pointStock, p_grid, valueCutsNext[
63         p_initialRegime], p_optimize->getDimensionToSplit(), p_world);
64     return ((p_world.rank() == 0) ? valSim(0) : 0.);
65 }
66 }
67 #endif

```

An example without data distribution can be found in the file `DynamicProgrammingByTreeCut.cpp`.

Using the framework in simulation

Similar to the 4.2.2 section, we can use the cuts calculated in optimization to test the optimal strategy found. In order to simulate an optimal policy step, a `SimulateStepTreeCutDist` object is provided with the constructor

```

1  SimulateStepTreeCutDist(gs::BinaryFileArchive &p_ar,  const int &p_iStep,  const std::
      string &p_nameCont,
2      const std::shared_ptr<FullGrid> &p_pGridFollowing,
3      const std::shared_ptr<OptimizerDPCutTreeBase > &p_pOptimize,
4      const bool &p_bOneFile,
5      const boost::mpi::communicator &p_world);

```

where

- `p_ar` is the binary archive where the continuation values are stored,
- `p_iStep` is the number associated with the current time step (0 at the start date of the simulation, the number is increased by one at each simulated time step),
- `p_nameCont` is the base name for continuation values,
- `p_GridFollowing` is the grid at the next time step (`p_iStep+1`),
- `p_Optimize` the Optimizer describing the transition problem solved using an LP program.
- `p_OneFile` equal to `true` if only one archive is used to store continuation values.
- `p_world` The MPI communicator

Remark 38 *A version without data distribution but with multithreaded and with possible MPI on the calculations is available with the `SimulateStepTreeCut` object. The `p_OneFile` argument is omitted during construction.*

This object implements the method `oneStep`

```

1  void oneStep(std::vector<StateTreeStocks > &p_statevector, Eigen::ArrayXXd &p_phiInOut)
      const

```

where:

- `p_statevector` stores the states of all the simulations: this state is updated by applying the optimal command,
- `p_phiInOut` stores the gain/cost functions for all simulations: it is updated by the function call. The size of the array is $(nb, nbSimul)$ where nb is given by the `getSimuFuncSize` method of the optimizer and $nbSimul$ the number of Monte Carlo simulations.

An example of using this method to simulate an optimal policy with distribution is given below:

```

1 // Copyright (C) 2019 EDF
2 // All Rights Reserved
3 // This code is published under the GNU Lesser General Public License (GNU LGPL)
4 #ifndef SIMULATEREGTRECUTDIST_H
5 #define SIMULATEREGTRECUTDIST_H
6 #include <functional>
7 #include <memory>
8 #include <Eigen/Dense>
9 #include <boost/mpi.hpp>
10 #include "geners/BinaryFileArchive.hh"
11 #include "StOpt/core/grids/FullGrid.h"
12 #include "StOpt/tree/StateTreeStocks.h"
13 #include "StOpt/dp/SimulateStepTreeCutDist.h"
14 #include "StOpt/dp/OptimizerDPCutBase.h"
15 #include "StOpt/dp/SimulatorDPBase.h"
16
17
18 /** \file SimulateTreeCutDist.h
19  * \brief Defines a simple program showing how to use simulations when optimization achieved
20  *         with transition problems solved with cuts and uncertainties on a tree
21  *         A simple grid is used
22  * \author Xavier Warin
23  */
24
25 /// \brief Simulate the optimal strategy , mpi version, Bellman cuts used to allow LP
26 /// resolution of transition problems when uncertainties are defined on a tree
27 /// \param p_grid grid used for deterministic state (stocks for example)
28 /// \param p_optimize optimizer defining the optimization between two time
29 /// steps
30 /// \param p_funcFinalValue function defining the final value cuts
31 /// \param p_pointStock initial point stock
32 /// \param p_initialRegime regime at initial date
33 /// \param p_fileToDump name associated to dumped bellman values
34 /// \param p_bOneFile do we store continuation values in only one file
35 /// \param p_world MPI communicator
36 double SimulateTreeCutDist(const std::shared_ptr<StOpt::FullGrid> &p_grid,
37                             const std::shared_ptr<StOpt::OptimizerDPCutTreeBase > &
38                             p_optimize,
39                             const std::function< Eigen::ArrayXd(const int &, const Eigen::
40                             ArrayXd &, const Eigen::ArrayXd &)> &p_funcFinalValue,
41                             const Eigen::ArrayXd &p_pointStock,
42                             const int &p_initialRegime,
43                             const std::string &p_fileToDump,
44                             const bool &p_bOneFile,
45                             const boost::mpi::communicator &p_world)
46 {
47     // from the optimizer get back the simulator
48     std::shared_ptr< StOpt::SimulatorDPBaseTree> simulator = p_optimize->getSimulator();
49     int nbStep = simulator->getNbStep();
50     std::vector< StOpt::StateTreeStocks> states;
51     states.reserve(simulator->getNbSimul());
52     for (int is = 0; is < simulator->getNbSimul(); ++is)
53         states.push_back(StOpt::StateTreeStocks(p_initialRegime, p_pointStock, 0));
54 }

```

```

50     std::string toDump = p_fileToDump ;
51     // test if one file generated
52     if (!p_bOneFile)
53         toDump += "_" + boost::lexical_cast<std::string>(p_world.rank());
54     gs::BinaryFileArchive ar(toDump.c_str(), "r");
55     // name for continuation object in archive
56     std::string nameAr = "ContinuationTree";
57     // cost function
58     Eigen::ArrayXXd costFunction = Eigen::ArrayXXd::Zero(p_optimize->getSimuFuncSize(),
59         simulator->getNbSimul());
60     for (int istep = 0; istep < nbStep; ++istep)
61     {
62         StOpt::SimulateStepTreeCutDist(ar, nbStep - 1 - istep, nameAr, p_grid, p_optimize,
63             p_bOneFile, p_world).oneStep(states, costFunction);
64
65         // new date
66         simulator->stepForward();
67         for (int is = 0; is < simulator->getNbSimul(); ++is)
68             states[is].setStochasticRealization(simulator->getNodeAssociatedToSim(is));
69     }
70     // final : accept to exercise if not already done entirely (here suppose one function
71     // to follow)
72     for (int is = 0; is < simulator->getNbSimul(); ++is)
73         costFunction(0, is) += p_funcFinalValue(states[is].getRegime(), states[is].
74             getPtStock(), simulator->getValueAssociatedToNode(states[is].
75                 getStochasticRealization()))(0);
76
77     return costFunction.mean();
78 }
79
80 #endif /* SIMULATETREECUTDIST_H */

```

The version of the previous example using a single archive storing the control/solution is given in `SimulateTreeCut.h` file.

Chapter 5

Using the C++ framework to solve some hedging problem

In this chapter we present an algorithm developed in StOpt to solve some hedging problem supposing that a mean variance criterion is chosen. The methodology follows the article [50] In this section we suppose that $(\Omega, \mathcal{F}, (\mathcal{F}_t)_{t \in [0, T]})$ is a filtered probability space. We define a set of trading dates $\mathcal{T} = \{t_0 = 0, t_1, \dots, t_{N-1}, t_N = T\}$ and we suppose that we are given an asset used as an hedging product $(S_t)_{t_0, t_N}$ which is almost surely positive, square integrable so that $\mathbb{E}[S_t^2] < \infty$ and adapted so that S_t is $\mathcal{F}_t$ -measurable for $t = t_0, \dots, t_N$. At last we suppose that the risk free rate is zero so that a bond has always a value of 1.

5.1 The problem

Suppose we are given a contingent claim $H \in \mathcal{L}^2(P)$ which is assumed to be a random variable $\mathcal{F}_T$ -measurable. In the case of a European call option on an asset S_t with strike K and maturity T , $H(\omega) = (S_T(\omega) - K)^+$.

We are only interested in self-financing strategies with limited orders, therefore with limited controls. By extending the definition [35], [7], we define:

Definition 1 A $(\bar{m}, \bar{l})$ self-financing strategy $\mathcal{V} = (\mathcal{V}_{t_i})_{i=0, \dots, N-1}$ is a pair of adapted process $(m_{t_i}, l_{t_i})_{i=0, \dots, N-1}$ defined for $(\bar{m}, \bar{l}) \in (0, \infty) \times (0, \infty)$ such that:

- $0 \leq m_{t_i} \leq \bar{m}, \quad 0 \leq l_{t_i} \leq \bar{l} \quad P.a.s. \quad \forall i = 0, \dots, N-1,$
- $m_{t_i} l_{t_i} = 0 \quad P.a.s. \quad \forall i = 0, \dots, N-1.$

In this definition m_t corresponds to the number of shares sold on the date t , and l_t the number of shares purchased on this date.

Remark 39 The strategies defined in [35] and [7] do not require that $m_t l_t = 0$ therefore a buy and sell check could take place on the same given date.

We denote $\Theta^{(\bar{m}, \bar{l})}$ the set of $(\bar{m}, \bar{l})$ self-financing strategy and with obvious notations $\nu = (m, l)$ for $\nu \in \Theta^{(\bar{m}, \bar{l})}$.

We consider a proportional cost model, so that an investor buying a stock on date t will pay $(1 + \lambda)S_t$ and an investor selling that stock will only receive $(1 - \lambda)S_t$. Assuming no transaction cost at the last date T , the final wealth of an investor with initial wealth x is given by:

$$x - \sum_{i=0}^{N-1} (1 + \lambda)l_{t_i}S_{t_i} + \sum_{i=0}^{N-1} (1 - \lambda)m_{t_i}S_{t_i} + \sum_{i=0}^{N-1} l_{t_i}S_{t_N} - \sum_{i=0}^{N-1} m_{t_i}S_{t_N}. \quad (5.1)$$

Remark 40 *The transaction costs at the last date T are linked to the nature of the contract. In the case of a pure financial contract, the investor will sell the asset and then some transaction fees will need to be paid to offset the final position. In the energy market, for example, the contract is often associated with a physical delivery and no specific costs are payable. Furthermore in these markets, even if the contract is purely financial, the futures markets are rather illiquid, which means significant transaction costs while the spot markets are much more liquid, which justifies neglecting final transaction costs.*

As in [35] [7], we define the minimal risk strategy minimizing the $\mathcal{L}^2$ risk of the hedging portfolio:

Definition 2 *A self-financing strategy $(\bar{m}, \bar{l})$ $\hat{\mathcal{V}} = (\hat{m}, \hat{l})$ is the minimization of the overall risk for the contingent claim H and the initial capital x if:*

$$\begin{aligned} \hat{\mathcal{V}} = \arg \min_{\nu=(m,l) \in \Theta(\bar{m}, \bar{l})} \mathbb{E}[(H - x + \sum_{i=0}^{N-1} (1 + \lambda)l_{t_i}S_{t_i} - \\ \sum_{i=0}^{N-1} (1 - \lambda)m_{t_i}S_{t_i} - \sum_{i=0}^{N-1} l_{t_i}S_{t_N} + \sum_{i=0}^{N-1} m_{t_i}S_{t_N})^2]. \end{aligned} \quad (5.2)$$

5.2 Theoretical algorithm

it is assumed that the process is Markov and that the gain H is a function of the asset value at maturity only to simplify the presentation of the proposed the Monte Carlo method.

We introduce the global position $\nu = (\nu_i)_{i=0, \dots, N-1}$ with:

$$\nu_i = \sum_{j=0}^i (m_{t_j} - l_{t_j}), \forall i = 0, \dots, N-1.$$

Using the property $m_{t_i}l_{t_i} = 0$, $\forall i = 0, \dots, N-1$, we get $|\nu_i - \nu_{i-1}| = l_{t_i} + m_{t_i}$ with the convention that $\nu_{-1} = 0$ and

$$G_T(\mathcal{V}) = \hat{G}_T(\nu) = x - \sum_{i=0}^{N-1} \lambda |\Delta \nu_{i-1}| S_{t_i} + \sum_{i=0}^{N-1} \nu_i \Delta S_i,$$

where $\Delta S_i = S_{t_{i+1}} - S_{t_i}$, $\Delta \nu_i = \nu_{i+1} - \nu_i$.

We then introduce $\hat{\Theta}(\bar{m}, \bar{l})$ the set of adapted random variables $(\nu_i)_{i=0, \dots, N-1}$ such that

$$-\bar{m} \leq \nu_i - \nu_{i-1} \leq \bar{l}, \forall i = 1, \dots, N-1.$$

The problem (5.2) can be rewritten as it was done in [42] by finding $\hat{\nu} = (\hat{\nu}_i)_{i=0,\dots,N-1}$ satisfactory:

$$\hat{\nu} = \arg \min_{\nu \in \hat{\Theta}(\bar{m}, \bar{l})} \mathbb{E}[(H - x - \hat{G}_T(\nu))^2]. \quad (5.3)$$

We introduce the spaces κ_i , $i = 0, \dots, N$ of the $\mathcal{F}_{t_i}$ -measurable and square integrable random variables. We define for $i \in 0, \dots, N$, $V_i \in \kappa_i$ as:

$$\begin{aligned} V_N &= H, \\ V_i &= \mathbb{E}[H - \sum_{j=i}^{N-1} \nu_j \Delta S_j + \lambda \sum_{j=i}^{N-1} |\Delta \nu_{j-1}| S_{t_j} | \mathcal{F}_{t_i}], \forall i = 0, \dots, N-1. \end{aligned} \quad (5.4)$$

then

$$\mathbb{E}[(H - x - \hat{G}_T(\nu))^2] = \mathbb{E}[(V_N - \nu_{N-1} \Delta S_{N-1} + \lambda |\Delta \nu_{N-2}| S_{t_{N-1}} - V_{N-1}) + \quad (5.5)$$

$$\sum_{i=2}^{N-1} (V_i + \lambda |\Delta \nu_{i-2}| S_{t_{i-1}} - \nu_{i-1} \Delta S_{i-1} - V_{i-1}) + \quad (5.6)$$

$$(V_1 + \lambda |\nu_0| S_{t_0} - \nu_0 \Delta S_0 - x)^2] \quad (5.7)$$

Due to the definition (5.4), we have that

$$\mathbb{E}[V_i + \lambda |\Delta \nu_{i-2}| S_{t_{i-1}} - \nu_{i-1} \Delta S_{i-1} - V_{i-1} | \mathcal{F}_{t_{i-1}}] = 0, \forall i = 1, \dots, N, \quad (5.8)$$

so that

$$\begin{aligned} \mathbb{E}[(H - x - \hat{G}_T(\nu))^2] &= \mathbb{E}[\mathbb{E}[(V_N - \nu_{N-1} \Delta S_{N-1} + \lambda |\Delta \nu_{N-2}| S_{t_{N-1}} - V_{N-1})^2 | \mathcal{F}_{t_{N-1}}] + \\ &\quad \mathbb{E}[\sum_{i=2}^{N-1} (V_i + \lambda |\Delta \nu_{i-2}| S_{t_{i-1}} - \nu_{i-1} \Delta S_{i-1} - V_{i-1})^2 + \\ &\quad (V_1 + \lambda |\nu_0| S_{t_0} - \nu_0 \Delta S_0 - x)^2] \end{aligned}$$

and iterating the process gives

$$\begin{aligned} \mathbb{E}[(H - x - \hat{G}_T(\nu))^2] &= \mathbb{E}[(V_N - \nu_{N-1} \Delta S_{N-1} + \lambda |\Delta \nu_{N-2}| S_{t_{N-1}} - V_{N-1})^2] + \\ &\quad \sum_{i=2}^{N-1} \mathbb{E}[(V_i + \lambda |\Delta \nu_{i-2}| S_{t_{i-1}} - \nu_{i-1} \Delta S_{i-1} - V_{i-1})^2] + \\ &\quad \mathbb{E}[(V_1 + \lambda |\nu_0| S_{t_0} - \nu_0 \Delta S_0 - x)^2] \end{aligned}$$

Then we can write the problem (5.3) as:

$$\begin{aligned} \hat{\nu} &= \arg \min_{\nu \in \hat{\Theta}(\bar{m}, \bar{l})} \mathbb{E}[(V_N - \nu_{N-1} \Delta S_{N-1} + \lambda |\Delta \nu_{N-2}| S_{t_{N-1}} - V_{N-1})^2] + \\ &\quad \sum_{i=2}^{N-1} \mathbb{E}[(V_i + \lambda |\Delta \nu_{i-2}| S_{t_{i-1}} - \nu_{i-1} \Delta S_{i-1} - V_{i-1})^2] + \\ &\quad \mathbb{E}[(V_1 + \lambda |\nu_0| S_{t_0} - \nu_0 \Delta S_0 - x)^2] \end{aligned} \quad (5.9)$$

We introduce the space

$$\rho_i^{\bar{m}, \bar{l}}(\eta) = \{(V, \nu)/V, \nu \text{ are } \mathbb{R} \text{ valued } \mathcal{F}_{t_i}\text{-adapted with } -\bar{m} \leq \nu - \eta \leq \bar{l}\},$$

and the space

$$\hat{\rho}_i^{\bar{m}, \bar{l}}(\eta) = \{(V, \nu_i, \dots, \nu_{N-1})/V \text{ is } \mathbb{R} \text{ valued, } \mathcal{F}_{t_i}\text{-adapted, the } \nu_j, j \geq i \text{ are } \mathbb{R} \text{ valued } \mathcal{F}_{t_j}\text{-adapted with } \bar{m} \leq \nu_i - \eta \leq \bar{l}, \bar{m} \leq \nu_{j+1} - \nu_j \leq \bar{l} \text{ for } i \leq j < N-1\},$$

As in the scheme introduced in [4] to improve the methodology proposed in [19] to solve Backward Stochastic Differential Equations, we can propose an algorithm where the update for $\bar{R}$ is taken ω by ω and stores the optimal trading payoff function on each trajectory. Then $\bar{R}$ satisfies the date t_i with an asset value S_{t_i} for an investment ν_{i-1} chosen at the date t_{i-1} :

$$\begin{aligned} \bar{R}(t_i, S_{t_i}, \nu_{i-1}) &= H - \sum_{j=i}^{N-1} \nu_j \Delta S_j + \lambda \sum_{j=i}^{N-1} |\Delta \nu_{j-1}| S_{t_j}, \\ &= R(t_{i+1}, S_{t_{i+1}}, \nu_i) - \nu_i \Delta S_i + \lambda |\Delta \nu_{i-1}| S_{t_i}, \end{aligned}$$

and at the date t_i according to the equation (5.5) the optimal control is the control ν associated with the minimization problem:

$$\min_{(V, \nu) \in \rho_i^{\bar{m}, \bar{l}}(\nu_{i-1})} \mathbb{E}[(\bar{R}(t_{i+1}, S_{t_{i+1}}, \nu) - \nu \Delta S_i + \lambda |\nu - \nu_{i-1}| S_{t_i} - V)^2 | \mathcal{F}_{t_i}]$$

This leads to the 10 algorithm.

Algorithm 10 Backward resolution for $\mathcal{L}^2$ minimization problem avoiding conditional expectation iteration.

- 1: $\bar{R}(t_N, S_{t_{N-1}}(\omega), \nu_{N-1}) = H(\omega), \quad \forall \nu_{N-1}$
- 2: **for** $i = N, 2$ **do**
- 3:

$$\begin{aligned} (\tilde{V}(t_{i-1}, S_{t_{i-1}}, \nu_{i-2}), \nu_{i-1}) &= \arg \min_{(V, \nu) \in \rho_{i-1}^{\bar{m}, \bar{l}}(\nu_{i-2})} \mathbb{E}[(\bar{R}(t_i, S_{t_i}, \nu) - \\ &\quad \nu \Delta S_{i-1} + \lambda |\nu - \nu_{i-2}| S_{t_{i-1}} - V)^2 | \mathcal{F}_{t_{i-1}}] \end{aligned} \quad (5.10)$$

- 4: $\bar{R}(t_{i-1}, S_{t_{i-1}}, \nu_{i-2}) = \bar{R}(t_i, S_{t_i}, \nu_{i-1}) - \nu_{i-1} \Delta S_{i-1} + \lambda |\Delta \nu_{i-2}| S_{t_{i-1}}$
 - 5: **end for**
 - 6: $\nu_0 = \arg \min_{\nu \in [-\bar{m}, \bar{l}]} \mathbb{E}[(\bar{R}(t_1, S_{t_1}, \nu) + \lambda |\nu| S_{t_0} - \nu \Delta S_0 - V)^2]$
-

Remark 41 In order to deal with the case of the average variance coverage which consists in finding the optimal strategy and the initial wealth to hedge the contingent claim, the last line of the 10 algorithm is replaced by

$$(\tilde{V}, \nu_0) = \arg \min_{(V, \nu)} \mathbb{E}[(V(t_1, S_{t_1}, \nu) + \lambda |\nu| S_{t_0} - \nu \Delta S_0 - V)^2 + R(t_1, S_{t_1}, \nu)],$$

and last line of Algorithm 10 by

$$(\tilde{V}, \nu_0) = \arg \min_{(V, \nu) \in \mathbb{R} \times [-\bar{m}, \bar{l}]} \mathbb{E}[(\bar{R}(t_1, S_{t_1}, \nu) + \lambda |\nu| S_{t_0} - \nu \Delta S_0 - V)^2].$$

Remark 42 *In the two algorithms presented an argmin must be carried out: a discretization in ν_{i-2} must be carried out on a grid $[\nu_{i-1} - m, \nu_{i-1} + l]$.*

5.3 Practical algorithm based on Algorithm 10

From the theoretical Algorithm 10, we aim to obtain an efficient implementation based on a representation of the function $\tilde{V}$ as a function of time, S_t and the position ν_t in plan assets.

- In order to represent the dependency in the hedging position, we introduce a time dependent grid

$$\mathcal{Q}_i := (\xi k)_{k=-(i+1)\lfloor \frac{\bar{m}}{\xi} \rfloor, \dots, (i+1)\lfloor \frac{\bar{l}}{\xi} \rfloor}$$

where ξ is the mesh size associated with the set of grids $(\mathcal{Q}_i)_{i=0,N}$ and, if possible, chosen such that $\frac{\bar{l}}{\xi} = \lfloor \frac{\bar{l}}{\xi} \rfloor$ and $\frac{\bar{m}}{\xi} = \lfloor \frac{\bar{m}}{\xi} \rfloor$.

- To represent the dependency in S_t we will use a Monte Carlo method using the simulated path $\left((S_{t_i}^{(j)})_{i=0, \dots, N} \right)_{j=1, \dots, M}$ and calculate the arg min in the equation (5.10) using a methodology similar so that described in [9]: suppose we are given at each date t_i $(D_q^i)_{q=1, \dots, Q}$ a partition of $[\min_{j=1, \dots, M} S_{t_i}^{(j)}, \max_{j=1, \dots, M} S_{t_i}^{(j)}]$ so that each cell contains the same number of samples. We use the cells Q $(D_q^i)_{q=1, \dots, Q}$ to represent the dependency of $\tilde{V}$ and ν in the S_{t_i} variable.

On each cell q we search $\hat{V}^q$ a linear approximation of the function $\tilde{V}$ at a given date t_i and for a position $k\xi$ so that $\hat{V}^q(t_i, S, k) = a_i^q + b_i^q S$ is an approximation of $\tilde{V}(t_i, S, k\xi)$. On cell q the optimal numeric hedging command $\hat{\nu}^q(k)$ for a position $k\xi$ can be seen as a sensitivity, so it is natural to look for a constant control per cell q when the value function is represented as a linear function.

Let us denote $(l_i^q(j))_{j=1, \frac{M}{Q}}$ the set of all the samples belonging to the cell q at date t_i . On each mesh the optimal control $\hat{\nu}^q$ is obtained by discretizing the command ν on a grid $\eta = ((k+r)\xi)_{r=-\lfloor \frac{\bar{m}}{\xi} \rfloor, \dots, \lfloor \frac{\bar{l}}{\xi} \rfloor}$, and testing the one giving a value $\hat{V}^q$ minimizing the risk $\mathcal{L}^2$ to solve the equation (5.10).

The 11 algorithm allows to find the optimal $\nu_i^{(j)}(k)$ command using the 10 algorithm at the date t_i , for a position of hedging $k\xi$ and for all Monte Carlo simulations j . For each command tested on cell q the corresponding function $\hat{V}^q$ is calculated by regression.

Remark 43 *It is possible to use a different discretization ξ to define the set η and the set $\mathcal{Q}_i$. Then an interpolation is necessary to obtain the values $\bar{R}$ at a position not belonging to the grid. An example of using such an interpolation for a gas storage problem following the optimal cash flow generated along Monte Carlo strategies can be found in [47].*

Remark 44 *This algorithm makes it possible to add a certain global constraint on the overall liquidity of the hedging asset. This is achieved by limiting the possible hedge positions to a subset of $\mathcal{Q}_i$ at each date t_i .*

Then, the global discretized version of the 10 algorithm is given on the 12 algorithm where $H^{(j)}$ correspond to j the Monte Carlo realization of the gain.

Algorithm 11 Optimize minimal hedging position $(\hat{\nu}_{t_i}^{(l)}(k))_{l=1,\dots,M}$ at date t_{i-1}

```

1: procedure OPTIMALCONTROL(  $\bar{R}(t_{i+1}, \cdot, \cdot), k, S_{t_i}, S_{t_{i+1}}$  )
2:   for  $q = 1, Q$  do
3:      $P = \infty$ ,
4:     for  $k = -\lfloor \frac{\bar{m}}{\xi} \rfloor, \dots, \lfloor \frac{\bar{l}}{\xi} \rfloor$  do
5:        $(a_i^q, b_i^q) = \arg \min_{(a,b) \in \mathbb{R}^2} \sum_{j=1}^{\frac{M}{Q}} (\bar{R}(t_{i+1}, S_{t_{i+1}}^{l_i^q(j)}, (k+l)\xi) -$ 
         $(k+l)\xi \Delta S_i^{l_i^q(j)} +$ 
         $\lambda |l\xi| S_{t_i}^{l_i^q(j)} - (a + b S_{t_i}^{l_i^q(j)})^2$ 
6:        $\tilde{P} = \sum_{j=1}^{\frac{M}{Q}} (\bar{R}(t_{i+1}, S_{t_{i+1}}^{l_i^q(j)}, (k+l)\xi) - (k+l)\xi \Delta S_i^{l_i^q(j)} +$ 
         $\lambda |l\xi| S_{t_i}^{l_i^q(j)} - (a_i^q + b_i^q S_{t_i}^{l_i^q(j)})^2$ 
7:       if  $\tilde{P} < P$  then
8:          $\nu^q = k\xi, P = \tilde{P}$ 
9:       end if
10:    end for
11:    for  $j = 1, \frac{M}{Q}$  do
12:       $\hat{\nu}_i^{(l_i^q(j))}(k) = \nu^q$ 
13:    end for
14:  end for return  $(\hat{\nu}_{t_i}^{(j)}(k))_{j=1,\dots,M}$ 
15: end procedure

```

Algorithm 12 Global backward resolution algorithm, optimal control and optimal variance calculation

```

1: for  $\nu \in \mathcal{Q}_{N-1}$  do
2:   for  $j \in [1, M]$  do
3:      $\bar{R}(t_N, S_{t_N}^{(j)}, \nu) = H^{(j)}$ 
4:   end for
5: end for
6: for  $i = N, 2$  do
7:   for  $k\xi \in \mathcal{Q}_{i-2}$  do
8:      $(\nu_{i-1}^{(j)}(k))_{j=1,M} = \text{OptimalControl}(\bar{R}(t_i, \cdot, \cdot), k, S_{t_{i-1}}, S_{t_i}),$ 
9:     for  $j \in [1, M]$  do
10:       $\bar{R}(t_{i-1}, S_{t_{i-1}}^{(j)}, k\xi) = \bar{R}(t_i, S_{t_i}^{(j)}, \nu_{i-1}^{(j)}(k)) -$ 
11:         $\nu_{i-1}^{(j)}(k)\Delta S_{i-1}^{(j)} + \lambda|\nu_{i-1}^{(j)}(k) - k\xi|S_{t_{i-1}}^{(j)}$ 
12:    end for
13:  end for
14:   $P = \infty,$ 
15:  for  $k = -\lfloor \frac{\bar{m}}{\xi} \rfloor, \dots, \lfloor \frac{\bar{l}}{\xi} \rfloor$  do
16:     $\tilde{P} = \sum_{j=1}^M (\bar{R}(t_1, S_{t_1}^{(j)}, k\xi) - k\xi\Delta S_0^{(j)} + \lambda|k|\xi S_0 - x)^2$ 
17:    if  $\tilde{P} < P$  then
18:       $\nu_0 = k\xi, P = \tilde{P}$ 
19:    end if
20:  end for
21:   $Var = \frac{1}{M} \sum_{j=1}^M (\bar{R}(t_1, S_{t_1}^{(j)}, \nu_0) - \nu_0\Delta S_0^{(j)} + \lambda|\nu_0|S_0 - x)^2$ 

```

Chapter 6

Managing stock with dynamic programming where the transition problem is itself the result of a deterministic optimization using dynamic programming

In this section the state is the same as in the section 3: it is separated into three parts $X^{x,t} = (X_1^{x,t}, X_2^{x,t}, I)$. $(X_1^{x,t}, X_2^{x,t})$ corresponds to the special case of chapter 4 where $X_1^{x,t}$ is not controlled and $X_2^{x,t}$ is controlled and deterministic as in a storage problem. I_t takes integers values and is here to describe some finite discrete states :

- for example some regimes (to deal with some switching problems).
- some time constraints increasing the state such as ramp constraints in thermal assets.

We suppose that on the period $[t, t + \Delta t]$, the process $X_1^{x,t}$ is kept constant while on this period $(X_2^{x,s}, I_s)$ for $s \in [t, t + \Delta t]$ is the solution of a deterministic optimization problem where the terminal condition at $t + \Delta t$ is given by some Bellman values previously calculated. A classical example is the management of an asset with hourly commands taken but optimizing the asset every week taking into account a stochastic state kept constant during the week. Then manager only observes the uncertainties every week and a deterministic optimization is achieved during the corresponding week.

This optimization can be carried out using the framework developed in a previous section but it is more effective to use a more specific framework where the deterministic problems can be solved by Dynamic Programming for each simulation.

In this framework a transition step will result in a deterministic optimization on all trajectories. The characteristic of modelization are :

- The stock grids used can be time dependant but they are full grids,
- During the deterministic optimization the number of regimes I can be increased to take in account some additionnal discrete constraints that are only relevant inside the transition period.

- The global problem is currently solve using regression to calculate conditional expectations,
- The direct storage of the control calculated in optimization and its used in simulation is not currently developped. Then continuation values calculated in optimization in the stochastic part and the deterministic part are stored and reused in simulation to recalculate locally the optimal control.

6.1 General requirement for the simulator

In order to optimize the control problem, the user must develop simulators allowing to trace some trajectories of uncertainties. This trajectories are used during optimization or in a simulation part using the continuation values calculated in optimization. In the following, we assume that we have developed a Simulator generating Monte Carlo simulations at different stochastic optimizations dates (as the uncertainties are kept constant on each transition step needing the deterministic optimization). In order to use the different frameworks developed in the following, we assume that the simulator is derived from the abstract class `SimulatorMultiStageDPBase` itself derived for `SimulatorDPBase` (see at the beginning of part III).

```

1 // Copyright (C) 2023 EDF
2 // All Rights Reserved
3 // This code is published under the GNU Lesser General Public License (GNU LGPL)
4 ///
5 #ifndef SIMULATORMULTISTAGEDPBASE_H
6 #define SIMULATORMULTISTAGEDPBASE_H
7 #include <Eigen/Dense>
8 #include "StOpt/dp/SimulatorDPBase.h"
9
10 /* \file SimulatorMultiStageDPBase.h
11 * \brief Abstract class for simulators for Dynamic Programming Programms when a
12 *        deterministic optimization
13 *        by DP is achieved on a transition step in time
14 * \author Benoit Clair, Xavier Warin
15 */
16 namespace StOpt
17 {
18     /// \class SimulatorMultiStageDPBase SimulatorMultiStageDPBase.h
19     /// Abstract class for simulator used in dynamic programming when a deterministic
20     /// optimization by DP is achieved on a transition step in time
21     class SimulatorMultiStageDPBase : public SimulatorDPBase
22     {
23     private :
24
25         int m_iperiodCur ; ///< Number of current period during optimization or simulation in
26                             ///< deterministic for the current time step
27     public :
28
29         /// \brief Constructor
30         SimulatorMultiStageDPBase() : SimulatorDPBase() {}
31         /// \brief Destructor
32         virtual ~SimulatorMultiStageDPBase() {}
33         /// \brief Returns number of periods of current timestep
34         virtual int getNbPeriodsInTransition() const = 0;
35         ///< Set period treated
36         void setPeriodInTransition(const int &p_iperiodCur)

```

```

37 {
38     m_iperiodCur = p_iperiodCur;
39 }
40 ///< Get period treated
41 int getPeriodInTransition() const
42 {
43     return m_iperiodCur ;
44 }
45 };
46 }
47 #endif /* SIMULATORMULTISTAGEDPBASE_H */

```

We only give here the additional methods specific to this derived class:

- the `getNbPeriodsInTransition` gives back at the current time step the number of periods used for the deterministic optimization on the current transition problem,
- the `setPeriodInTransition` permits to update the index of the transition period,
- the `getPeriodInTransition` permits to get back the index of the transition period.

6.2 General requirement for the business object

In order to use the framework, the developer must describe the problem he wants to solve at a period number of a given transition step from a state $X^{x,t}$. This business object must offer common methods and it is derived from `OptimizerMultiStageDPBase`.

```

1 // Copyright (C) 2023 EDF
2 // All Rights Reserved
3 // This code is published under the GNU Lesser General Public License (GNU LGPL)
4 #ifndef OPTIMIZERMULTISTAGEDPBASE_H
5 #define OPTIMIZERMULTISTAGEDPBASE_H
6 #include <Eigen/Dense>
7 #include "StOpt/core/utils/StateWithStocks.h"
8 #include "StOpt/core/grids/SpaceGrid.h"
9 #include "StOpt/dp/OptimizerBase.h"
10 #include "StOpt/dp/SimulatorMultiStageDPBase.h"
11 #include "StOpt/regression/GridAndRegressedValue.h"
12 #include "StOpt/regression/ContinuationValue.h"
13
14 /** \file OptimizerMultiStageDPBase.h
15  *  \brief Define an abstract class for Dynamic Programming problems where each transition
16  *         problem
17  *         is itself solved using a dynamic programming approach
18  *         \author Benoit Clair, Xavier Warin
19  */
20 namespace StOpt
21 {
22
23     ///< \class OptimizerMultiStageDPBase OptimizerMultiStageDPBase.h
24     ///< Base class for optimizer for Dynamic Programming with regressions where each each
25     ///< transition problem
26     ///< is itself solved using a deterministic DP
27     class OptimizerMultiStageDPBase : public OptimizerBase
28     {
29     public :
30
31         OptimizerMultiStageDPBase() {}
32
33         virtual ~OptimizerMultiStageDPBase() {}

```

```

34
35
36
37 /// \brief defines a step in optimization
38 /// \param p_grid      grid at arrival step after command
39 /// \param p_stock     coordinates of the stock point to treat
40 /// \param p_condEsp   continuation values for each regime
41 /// \param p_phiIn     for each regime gives the solution calculated at the
    previous step ( next time step by Dynamic Programming resolution) : structure of
    the 2D array ( nb simulation ,nb stocks )
42 /// \return for each regimes (column) gives the solution for each particle (row)
43 [[nodiscard]] virtual Eigen::ArrayXXd stepOptimize(const std::shared_ptr<StOpt::
    SpaceGrid> &p_grid,
44             const Eigen::ArrayXd &p_stock,
45             const std::vector< std::shared_ptr<ContinuationValue> > &p_condEsp,
46             const std::vector<std::shared_ptr<Eigen::ArrayXXd>> &p_phiIn) const = 0;
47
48 /// \brief defines a step in simulation
49 /// Notice that this implementation is not optimal but is convenient if the control is
    discrete.
50 /// By avoiding interpolation in control we avoid non admissible control
51 /// Control are recalculated during simulation.
52 /// \param p_grid      grid at arrival step after command
53 /// \param p_continuation defines the continuation operator for each regime
54 /// \param p_state      defines the state value (modified)
55 /// \param p_phiInOut   defines the value functions (modified) : size number of
    functions to follow
56 virtual void stepSimulate(const std::shared_ptr< StOpt::SpaceGrid> &p_grid, const std
    ::vector< StOpt::GridAndRegressedValue > &p_continuation,
57                        StOpt::StateWithStocks<double> &p_state,
58                        Eigen::Ref<Eigen::ArrayXd> p_phiInOut) const = 0 ;
59
60
61 // number of regime used in deterministic part
62 virtual int getNbDetRegime() const = 0 ;
63
64
65 /// \brief get the simulator back
66 virtual std::shared_ptr< SimulatorMultiStageDPBase > getSimulator() const = 0;
67
68 };
69 }
70 #endif /* OPTIMIZERMULTISTAGEDPBASE_H */

```

This class is derived from the class `OptimizerBase` (see section 3.1. The different methods added in the derived class to define in the business class are :

- the `getNbDetRegime` makes it possible to obtain the number of regimes of the deterministic problem for the current transition problem
- the `getSimulator` method is used to retrieve the simulator giving the Monte Carlo simulations,
- the `stepOptimize` method is used in optimization during the deterministic optimization on a period of a given stochastic transition problem. From a grid point `p_stock` it calculates the function values on each trajectory. It return a matrix (first dimension is the number of simulations, the second dimension the number of regimes) giving the value of the function at the current point in the stock..

6.3 Use the framework in optimization

Once an Optimizer is derived for the project, and assuming a full grid is used for inventory discretization, the framework provides a `TransitionStepMultiStageRegressionDPDist` object in MPI which allows to solve the optimization problem with data distribution over a time step with the 2 following constructors:

- Not storing the deterministic continuation values interpolated on trajectories.

```
1 TransitionStepMultiStageRegressionDPDist(const shared_ptr<FullGrid> &
2     p_pGridCurrent,
3     const shared_ptr<FullGrid> &p_pGridPrevious,
4     const shared_ptr<OptimizerDPBase> &p_pOptimize,
5     const boost::mpi::communicator &p_world)
```

- Storing the deterministic continuation values interpolated on trajectories.

```
1 TransitionStepMultiStageRegressionDPDist(const shared_ptr<FullGrid> &
2     p_pGridCurrent,
3     const shared_ptr<FullGrid> &p_pGridPrevious,
4     const shared_ptr<OptimizerDPBase> &p_pOptimize,
5     const std::shared_ptr<gs::BinaryFileArchive> &
6     p_arGen,
7     const std::string &p_nameDump,
8     const boost::mpi::communicator &p_world)
```

with

- `p_pGridCurrent` is the grid at the current time step (t_i),
- `p_pGridPrevious` is the grid at the previously processed time step (t_{i+1}),
- `p_pOptimize` the optimizer object,
- `p_arGen` is the name of the geners archive used to store deterministic continuation values (interpolated on simulation),
- `p_nameDump` is the name used to store the deterministic continuation values in the archive,
- `p_world` The MPI communicator.

Remark 45 *A similar object is available without the MPI distribution framework `TransitionStepMultiStageRegressionDP` with always the activation of parallelization with threads and MPI on calculations on the full points grid.*

The main method is

```
1 std::vector< shared_ptr< Eigen::ArrayXXd> > OneStep(const std::vector< shared_ptr<
2     Eigen::ArrayXXd> > &p_phiIn,
3     const shared_ptr< BaseRegression> &p_condExp)
```

with

- `p_phiIn` the vector (its size corresponds to the number of regimes) of the matrix of optimal values calculated at the previous time iteration for each regime. Each matrix is of size : a number of simulations per the number of stock points.
- `p_condExp` the conditional expectation operator used only on the last period of the transition step,

returning a vector of matrix with new optimal values at the current period of the transition time step (each element of the vector corresponds to a regime and each matrix is of size the number of simulations per the number of stock points).

A second method is provided permitting to dump the stochastic continuation values of the problem at each time step:

```

1 void dumpContinuationValues( std::shared_ptr<gs::BinaryFileArchive> p_ar ,
2                             const std::string &p_name, const int &p_iStep,
3                             const std::vector< std::shared_ptr< Eigen::ArrayXXd > > &
4                             p_phiInPrev,
5                             const std::shared_ptr<BaseRegression> &p_condExp,
6                             const bool &p_bOneFile) const

```

with:

- `p_ar` is the archive where the continuation values are dumped,
- `p_name` is a base name used in the archive to store the continuation values,
- `p_phiInPrev` is the previous time step solution used to calculate the continuation values that are stored,
- `p_condExp` is the conditional expectation object allowing to calculate the conditional expectation of the functions defined at the previous time step processed `p_phiInPrev`.
- `p_bOneFile` is set to `true` if the continuation values calculated by each processor are flushed to a single file. Otherwise, the continuation values calculated by each processor are flushed to different files (one by processor). If the problem results in optimal continuation values on the global grid that can be stored in the memory of the compute node, it may be more interesting to dump the continuation values in a single file.

Here we give a simple example of temporal resolution using this method when MPI data distribution is used

```

1 // Copyright (C) 2023 EDF
2 // All Rights Reserved
3 // This code is published under the GNU Lesser General Public License (GNU LGPL)
4 #ifdef USE_MPI
5 #include <fstream>
6 #include <memory>
7 #include <functional>
8 #include <boost/lexical_cast.hpp>
9 #include <boost/mpi.hpp>
10 #include <Eigen/Dense>
11 #include "geners/BinaryFileArchive.hh"
12 #include "StOpt/core/grids/FullGrid.h"
13 #include "StOpt/regression/BaseRegression.h"
14 #include "StOpt/dp/FinalStepDPDist.h"
15 #include "StOpt/dp/TransitionStepMultiStageRegressionDPDist.h"

```

```

16 #include "StOpt/core/parallelism/reconstructProcOMpi.h"
17 #include "StOpt/dp/OptimizerMultiStageDPBase.h"
18 #include "StOpt/dp/SimulatorMultiStageDPBase.h"
19
20
21 using namespace std;
22
23 double DynamicProgrammingByRegressionMultiStageDist(const shared_ptr<StOpt::FullGrid> &
    p_grid,
24     const shared_ptr<StOpt::OptimizerMultiStageDPBase> &p_optimize,
25     shared_ptr<StOpt::BaseRegression> &p_regressor,
26     const function<double(const int &, const Eigen::ArrayXd &, const Eigen::ArrayXd &)>
        &p_funcFinalValue,
27     const Eigen::ArrayXd &p_pointStock,
28     const int &p_initialRegime,
29     const string &p_fileToDump,
30     const bool &p_bOneFile,
31     const boost::mpi::communicator &p_world)
32 {
33     // from the optimizer get back the simulator
34     shared_ptr< StOpt::SimulatorMultiStageDPBase> simulator = p_optimize->getSimulator();
35     // final values
36     vector< shared_ptr< Eigen::ArrayXXd > > valuesNext = StOpt::FinalStepDPDist(p_grid,
        p_optimize->getNbRegime(), p_optimize->getDimensionToSplit(), p_world)(
        p_funcFinalValue, simulator->getParticles().array());
37     // dump
38     string toDump = p_fileToDump ;
39     // test if one file generated
40     if (!p_bOneFile)
41         toDump += "_" + boost::lexical_cast<string>(p_world.rank());
42     shared_ptr<gs::BinaryFileArchive> ar;
43     if ((!p_bOneFile) || (p_world.rank() == 0))
44         ar = make_shared<gs::BinaryFileArchive>(toDump.c_str(), "w");
45     // name for object in archive
46     string nameAr = "Continuation";
47     // Name for deterministic continuation in archive
48     string nameArContValDet = "ContinuationDet";
49     for (int iStep = 0; iStep < simulator->getNbStep(); ++iStep)
50     {
51         Eigen::ArrayXXd asset = simulator->stepBackwardAndGetParticles();
52         // conditional expectation operator
53         p_regressor->updateSimulations(((iStep == (simulator->getNbStep() - 1)) ? true :
            false), asset);
54         // transition object : constructor with dump on archive of deterministic
            continuation/Bellmna values
55         string toStorBellDet = nameArContValDet + boost::lexical_cast<string>(iStep);
56         // transition object
57         StOpt::TransitionStepMultiStageRegressionDPDist transStep(p_grid, p_grid,
            p_optimize, ar, toStorBellDet, p_bOneFile, p_world);
58         // dump stochastic continuation value
59         vector< shared_ptr< Eigen::ArrayXXd > > values = transStep.oneStep(valuesNext,
            p_regressor);
60         transStep.dumpContinuationValues(ar, nameAr, iStep, valuesNext, p_regressor,
            p_bOneFile);
61         valuesNext = values;
62     }
63     // reconstruct a small grid for interpolation
64     return StOpt::reconstructProcOMpi(p_pointStock, p_grid, valuesNext[p_initialRegime],
        p_optimize->getDimensionToSplit(), p_world).mean();
65 }
66 }
67 #endif

```

An example without data distribution can be found in the file `DynamicProgrammingByRegressionMultiStage.cpp`.

6.4 Use the framework in simulation

Once the optimization is done, the continuation values are saved in a file (or in some files) at each time step. In order to simulate the optimal policy, we use the previously calculated continuation values (see chapter 4) calculated in optimization.

While simulating strategies, two cases can occur:

- In most cases (small dimension case), the optimal function value can be stored in the computer's node memory and the function values are stored in a single file. In this case, an optimal simulation can be easily performed by distributing the Monte Carlo simulations on the available compute nodes: this can be done using the `SimulateStepMultiStageRegression` object at each time step of the simulation.
- When it comes to very large problems, optimization is achieved by distributing the data across the processors and it is impossible to store the continuation values on a node. In this case, the continuation values are stored in the memory of the node which was used to calculate them in optimization. The simulations are reorganized at each time step and gathered so as to occupy the same part of the global grid. Each processor will then get from the other processors a version of the continuation values it needs. This methodology is used in the `SimulateStepMultiStageRegressionDist` object.

In order to simulate one step of the optimal policy, an object `SimulateStepMultiStageRegressionDist` is provided with constructor

```
1 SimulateStepMultiStageRegressionDist(std::shared_ptr<gs::BinaryFileArchive> &p_ar,
2   const int &p_iStep,
3   const std::string &p_nameCont, const std::string &
4   p_nameDetCont,
5   const std::shared_ptr<FullGrid> &p_pGridCurrent,
6   const std::shared_ptr<SpaceGrid> &p_pGridFollowing,
7   const std::shared_ptr<StOpt::
8   OptimizerMultiStageDPBase> &p_pOptimize,
9   const boost::mpi::communicator &p_world)
```

where

- `p_ar` is the binary archive where the continuation values are stored,
- `p_iStep` is the number associated with the current time step (0 at the start date of the simulation, the number is increased by one at each simulated time step),
- `p_nameCont` is the base name for stochastic continuation values,
- `p_nameDetCont` is the base name for deterministic continuation values for all the periods of the transition,
- `p_pGridCurrent` is the grid at current step `p_iStep`,
- `p_pGridFollowing` is the grid at the next time step (`p_iStep+1`),
- `p_pOptimize` the Optimizer describing the passage from one time step to the next,

- `p_OneFile` equal to `true` if only one archive is used to store continuation values.
- `p_world` The MPI communicator

Remark 46 *A version without data distribution but with multithreaded and with possible MPI on calculations is available with the object `SimulateStepMultiStageRegression`. The `p_pGridCurrent` and `p_OneFile` argument are omitted during construction*

This object implements the method `oneStep`

```
1 void oneStep(std::vector<StateWithStocks<double> > &p_statevector , Eigen::ArrayXXd &
    p_phiInOut)
```

where:

- `p_statevector` stores the states of all simulations: this state is updated by applying the optimal command,
- `p_phiInOut` stores the gain/cost functions for all the simulations: it is updated by the function call. The size of the array is $(nb, nbSimul)$ where nb is given by the `getSimuFuncSize` method of the optimizer and $nbSimul$ the number of Monte Carlo simulations.

An example of using this method to simulate an optimal policy with distribution is given below:

```
1 // Copyright (C) 2023 EDF
2 // All Rights Reserved
3 // This code is published under the GNU Lesser General Public License (GNU LGPL)
4 #ifndef SIMULATEMULTISTAGEREGRESSIONDIST_H
5 #define SIMULATEMULTISTAGEREGRESSIONDIST_H
6 #include <functional>
7 #include <memory>
8 #include <Eigen/Dense>
9 #include <boost/mpi.hpp>
10 #include "geners/BinaryFileArchive.hh"
11 #include "StOpt/core/grids/FullGrid.h"
12 #include "StOpt/core/utils/StateWithStocks.h"
13 #include "StOpt/dp/SimulateStepMultiStageRegressionDist.h"
14 #include "StOpt/dp/OptimizerMultiStageDPBase.h"
15 #include "StOpt/dp/SimulatorMultiStageDPBase.h"
16
17
18 /** \file SimulateMultiStageRegressionDist.h
19  * \brief Defines a simple program showing how to use simulation when at each transition
20  *         step in time
21  *         a deterministic optimization of nPeriod is achieved
22  *         A simple grid is used
23  * \author Xavier Warin
24  */
25
26 /// \brief Simulate the optimal strategy , mpi version, using continuation values
27 ///         when a deterministic optimization of nPeriod is achieved.
28 /// \param p_grid          grid used for deterministic state (stocks for example)
29 /// \param p_optimize      optimizer defining the optimization between two time
30 ///                        steps
31 /// \param p_funcFinalValue function defining the final value
32 /// \param p_pointStock    initial point stock
33 /// \param p_initialRegime regime at initial date
34 /// \param p_fileToDump    name associated to dumped bellman values
```

```

34 /// \param p_bOneFile          do we store continuation values in only one file
35 /// \param p_world             MPI communicator
36 double SimulateMultiStageRegressionDist(const std::shared_ptr<StOpt::FullGrid> &p_grid,
37                                         const std::shared_ptr<StOpt::
38                                         OptimizerMultiStageDPBase > &p_optimize,
39                                         const std::function<double(const int &, const Eigen
40                                         ::ArrayXd &, const Eigen::ArrayXd &)> &
41                                         p_funcFinalValue,
42                                         const Eigen::ArrayXd &p_pointStock,
43                                         const int &p_initialRegime,
44                                         const std::string &p_fileToDump,
45                                         const bool &p_bOneFile,
46                                         const boost::mpi::communicator &p_world)
47 {
48     // from the optimizer get back the simulator
49     std::shared_ptr< StOpt::SimulatorMultiStageDPBase> simulator = p_optimize->getSimulator
50     ();
51     int nbStep = simulator->getNbStep();
52     std::vector< StOpt::StateWithStocks<double>> states;
53     states.reserve(simulator->getNbSimul());
54     for (int is = 0; is < simulator->getNbSimul(); ++is)
55         states.push_back(StOpt::StateWithStocks<double>(p_initialRegime, p_pointStock,
56         Eigen::ArrayXd::Zero(simulator->getDimension())));
57     std::string toDump = p_fileToDump ;
58     // test if one file generated
59     if (!p_bOneFile)
60         toDump += "_" + boost::lexical_cast<std::string>(p_world.rank());
61     std::shared_ptr<gs::BinaryFileArchive> ar = std::make_shared<gs::BinaryFileArchive>(
62     toDump.c_str(), "r");
63     std::cout << " FILEDUMPSIM " << toDump << std::endl ;
64     // name for continuation object in archive
65     std::string nameAr = "Continuation";
66     // Name for deterministic continuation in archive
67     std::string nameArContValDet = "ContinuationDet";
68     // cost function
69     Eigen::ArrayXXd costFunction = Eigen::ArrayXXd::Zero(p_optimize->getSimuFuncSize(),
70     simulator->getNbSimul());
71     for (int istep = 0; istep < nbStep; ++istep)
72     {
73         // Number of transition achieved for the current time step
74         int nbPeriodsOfCurrentStep = simulator->getNbPeriodsInTransition();
75         std::vector<Eigen::ArrayXXd> costFunctionPeriod(nbPeriodsOfCurrentStep);
76         for (int iPeriod = 0; iPeriod < nbPeriodsOfCurrentStep; ++iPeriod)
77             costFunctionPeriod[iPeriod] = Eigen::ArrayXXd::Zero(p_optimize->getSimuFuncSize
78             (), simulator->getNbSimul());
79         costFunctionPeriod[0] = costFunction;
80         std::string toStorBellDet = nameArContValDet + boost::lexical_cast<std::string>(
81         nbStep - 1 - istep);
82
83         StOpt::SimulateStepMultiStageRegressionDist(ar, nbStep - 1 - istep, nameAr,
84         toStorBellDet, p_grid, p_grid, p_optimize, p_bOneFile, p_world).oneStep(states,
85         costFunctionPeriod);
86
87         // new stochastic state
88         Eigen::ArrayXXd particles = simulator->stepForwardAndGetParticles();
89         for (int is = 0; is < simulator->getNbSimul(); ++is)
90             states[is].setStochasticRealization(particles.col(is));
91         // store current cost function
92         costFunction = costFunctionPeriod[nbPeriodsOfCurrentStep - 1];
93     }
94     // final : accept to exercise if not already done entirely (here suppose one function
95     to follow)
96     for (int is = 0; is < simulator->getNbSimul(); ++is)
97         costFunction(0, is) += p_funcFinalValue(states[is].getRegime(), states[is].
98         getPtStock(), states[is].getStochasticRealization()) * simulator->getActu();
99     return costFunction.mean();
100 }

```

```
90  
91 #endif /* SIMULATEMULTISTAGEREGRESSIONDIST_H */
```

The version of the previous example using a single archive storing the control/solution is given in the `SimulateMultiStageRegression.h` file.

Chapter 7

Calculating Sensitivities for Stochastic Problems

Calculating delta with log-normal models can be achieved using the tangent process [9], [47]. Other Greeks can be evaluated similarly. However, when using different price models, it can be useful to evaluate sensitivities with respect to the parameters of the price model using automatic differentiation.

All sensitivities can be computed in a single valuation, independently of the price model. Note that Greeks are generally calculated by freezing the optimal strategy.

Moreover, optimizing a stochastic problem is complex and involves many tasks that cannot be differentiated. When the problem is not continuous, as is the case for American and swing options, it is impossible to differentiate it directly. Each integer decision has to be relaxed (smoothed), and even for continuous problems, dynamic programming methods involve max functions that must be regularized.

For all these reasons, sensitivities are generally calculated after the optimal strategy has been determined, using the optimal control computed and stored during the optimization phase. They are evaluated using Monte Carlo simulations, following the state process associated with the optimal control.

Once all Monte Carlo simulations have been performed, the value is estimated as the average payoff over all trajectories, and sensitivities are obtained via reverse differentiation.

In StOpt, two libraries are tested (although other libraries should also work):

- The Adept library <https://www.met.reading.ac.uk/clouds/adept/>,
- The Stan Math library <https://mc-stan.org/math/>.

The StOpt library in C++ provides limited support for automatic differentiation in classical dynamic programming problems (see Chapter 3), potentially involving integer state constraints such as switching problems (see Chapter 4), which are smoothed during the simulation of the optimal control.

- The regressions used include the following regressors:

- Global regressors (see Section 2.6),
- LocalConstRegressor (see Section 2.2),
- LocalLinearRegressor (see Section 2.2)
- The grids used to manage the stochastic variables include:
 - Linear grids: GeneralSpaceGrid, RegularSpaceGrid (see Section 1.1.1),
 - Legendre grids: RegularLegendreGrid (see Section 1.2)

Since regressions are used, a business object deriving from the `OptimizerDPBase` class (see Section 3.2.2) must be employed.

Note that for stochastic problems, `StOpt` stores the continuation values and the interpolated optimal control. To estimate sensitivities, the simulation phase must directly use the stored optimal control rather than the continuation values.

Therefore, a method similar to `stepSimulateControl`, used in `OptimizerDPBase`, should be implemented. This method, defined in the derived business object, has the following interface:

```

1 template<typename T>
2 void stepSimulateControlAutoDiff(
3     const std::shared_ptr<StOpt::FullGrid> &p_grid,
4     const std::vector<StOpt::GridAndRegressedValueAutoDiff<T>> &p_control,
5     StOpt::StateWithStocks<T> &p_state,
6     Eigen::Ref<Eigen::Array<T, Eigen::Dynamic, 1>> p_phiInOut)

```

This interface is similar to `stepSimulateControl`, but uses a templated type (e.g., `adouble` from Adept or `var` from Stan Math). Additionally, the `GridAndRegressedValue` object storing the control is replaced by `GridAndRegressedValueAutoDiff`, which supports automatic differentiation types.

```

1 /** \file GridAndRegressedValueAutoDiff.h
2  * \brief Permits to store a function value defined in dimension \f$n = p+q \f$
3  * defined partly on a grid (dimension \f$p \f$) and partly by regression (dimension \f$q \f$)
4  * This is a version used for autodiff.
5  * \author Xavier Warin
6  */
7
8
9 namespace StOpt
10 {
11     /// \class GridAndRegressedValueAutoDiff GridAndRegressedValueAutoDiff.h
12     /// Permits to store a function value partly defined on a grid and by regression
13     template<typename T>
14     class GridAndRegressedValueAutoDiff
15     {
16     private :
17         std::shared_ptr< FullGrid > m_grid ; ///< grid used to define stock points
18         std::shared_ptr< BaseRegression > m_reg ; ///< regressor
19         std::vector< std::shared_ptr<InterpolatorSpectral<T>> > m_interpFuncBasis; ///< vector
20             of spectral operator associated to the grid used for each regressed function basis
21     public :
22         /// \brief Default constructor
23         GridAndRegressedValueAutoDiff() {}
24
25     }
26

```

```

27  /// \brief Constructor used for deserialization
28  /// \param p_grid          grid for stocks
29  /// \param p_reg           regressor
30  /// \param p_interpFuncBasis spectral interpolator associated to each regression
    function basis
31  GridAndRegressedValueAutoDiff(const std::shared_ptr< FullGrid > &p_grid,
32                                const std::shared_ptr< BaseRegression > &p_reg,
33                                const std::vector< std::shared_ptr<InterpolatorSpectral<T>> > &
                                    p_interpFuncBasis): m_grid(p_grid), m_reg(p_reg),
                                    m_interpFuncBasis(p_interpFuncBasis) {}
34
35
36
37  T getValueAutoDiff(const Eigen::Array<T, Eigen::Dynamic, 1> & p_ptOfStock,
38                    const Eigen::Array<T, Eigen::Dynamic, 1> & p_coordinates) const
39  {
40      return dispatchRegressionGetAValue(*m_reg, p_coordinates, p_ptOfStock,
41                                         m_interpFuncBasis);
42  }
43
44  void cloneFromDouble(const std::shared_ptr< FullGrid > &p_grid,
                        const std::shared_ptr< BaseRegression > &p_reg,

```

The equivalent of the `SimulateStepRegressionControl` object is the `SimulateStepRegressionControlAutoDiff` object, which retrieves the optimal control computed during the optimization phase as a function, and applies the control using the type defined by the autodiff library in use for all simulations.

```

1  /// \class SimulateStepRegressionControlAutoDiff SimulateStepRegressionControlAutoDiff.h
2  /// One step in forward simulation using an auto diff library
3  /// Templated object is the asset
4  template< class Optimizer, typename T>
5  class SimulateStepRegressionControlAutoDiff
6  {
7  private :
8
9      std::shared_ptr<FullGrid> m_pGridFollowing ; ///< global grid at following time step
10     std::shared_ptr<Optimizer > m_pOptimize ; ///< optimizer solving the problem
        for one point and one step
11     std::vector< GridAndRegressedValueAutoDiff<T> > m_control ; ///< to store the optimal
        control calculated in optimization
12
13 public :
14
15     /// \brief default
16     SimulateStepRegressionControlAutoDiff() {}
17     virtual ~SimulateStepRegressionControlAutoDiff() {}
18
19     /// \brief Constructor
20     /// \param p_ar          Archive where continuation values are stored
21     /// \param p_iStep       Step number identifier
22     /// \param p_nameCont     Name use to store continuation valuation
23     /// \param p_pGridFollowing grid at following time step
24     /// \param p_pOptimize    Optimize object defining the transition step
25     /// \param p_world        MPI communicator
26     SimulateStepRegressionControlAutoDiff(gs::BinaryFileArchive &p_ar, const int &p_iStep,
        const std::string &p_nameCont,
27
28                                     const std::shared_ptr<FullGrid> &
                                        p_pGridFollowing,
29                                     const std::shared_ptr<Optimizer > &p_pOptimize):
        m_pGridFollowing(p_pGridFollowing),
        m_pOptimize(p_pOptimize)
30  {
31     std::vector<GridAndRegressedValue> control;
32     gs::Reference< std::vector<GridAndRegressedValue> > (p_ar, (p_nameCont + "Control").
        c_str(), boost::lexical_cast<std::string>(p_iStep).c_str()).restore(0, &control);
33     m_control.resize(control.size());

```

```

33     for (int i=0; i < control.size(); ++i)
34     {
35         m_control[i].cloneFromDouble( std::static_pointer_cast<FullGrid>(control[i].getGrid
36             ()), control[i].getRegressor(), control[i]);
37     }
38 }
39
40 /// \brief Define one step arbitraging between possible commands
41 /// \param p_statevector    Vector of states (regime, stock descriptor, uncertainty)
42 /// \param p_phiInOut       actual contract value modified at current time step by
43                             applying an optimal command (size : number of function to follow by number of
44                             simulations)
45 void oneStep(std::vector<StateWithStocks<T> > &p_statevector, Eigen::Array<T, Eigen::
46     Dynamic, Eigen::Dynamic> &p_phiInOut) const
47 {
48     int is = 0 ;
49     // no OPEN MP as not safe with some auto diff libraries
50     for (is = 0; is < static_cast<int>(p_statevector.size()); ++is)
51     {
52         m_pOptimize->stepSimulateControlAutoDiff(m_pGridFollowing, m_control, p_statevector[
53             is], p_phiInOut.col(is));
54     }
55 }
56 };

```

Note that MPI is not available when computing sensitivities, contrary to the `SimulateStepRegressionControl` version.

At the end of this document, an example using Stan Math for gas storage is provided.

Part IV

Semi-Lagrangian methods

For the semi-Lagrangian methods, the C++ API is the only one available (no Python API is currently developed).

Chapter 1

Theoretical background

In this part, we return to solving the equation (2.1).

1.1 Notation and regularity results

We denote by $\wedge$ the minimum and $\vee$ the maximum. We denote by $|\cdot|$ the Euclidean norm of a vector, $Q := (0, T] \times \mathbb{R}^d$. For a bounded function w , we define

$$|w|_0 = \sup_{(t,x) \in Q} |w(t, x)|, \quad [w]_1 = \sup_{(s,x) \neq (t,y)} \frac{|w(s, x) - w(t, y)|}{|x - y| + |t - s|^{\frac{1}{2}}}$$

and $|w|_1 = |w|_0 + [w]_1$. $C_1(Q)$ will stand for the space of functions with a finite $|\cdot|_1$ norm. For t given, we denote

$$\|w(t, \cdot)\|_\infty = \sup_{x \in \mathbb{R}^d} |w(t, x)|$$

We use the classical assumption on the data of (2.1) for a given $\hat{K}$:

$$\sup_a |g|_1 + |\sigma_a|_1 + |b_a|_1 + |f_a|_1 + |c_a|_1 \leq \hat{K} \quad (1.1)$$

A classic result [25] gives us the existence and uniqueness of the solution in space of bounded Lipschitz functions:

Proposition 1 *If the coefficients of the equation (2.1) satisfy (1.1), there is a unique viscosity solution of the equation (2.1) belonging to $C_1(Q)$. If u_1 and u_2 are respectively sub and super solution of the equation (2.1) satisfying $u_1(0, \cdot) \leq u_2(0, \cdot)$ then $u_1 \leq u_2$.*

A spatial discretization length of the problem Δx being given, thereafter $(i_1 \Delta x, \dots, i_d \Delta x)$ with $\bar{i} = (i_1, \dots, i_d) \in \mathbf{Z}^d$ will correspond to the coordinates of a mesh $M_{\bar{i}}$ defining a hypercube in dimension d . For an interpolation grid $(\xi_i)_{i=0, \dots, N} \in [-1, 1]^N$, and for a mesh $\bar{i}$, the point $y_{\bar{i}, \tilde{j}}$ with $\tilde{j} = (j_1, \dots, j_d) \in [0, N]^d$ will have the coordinate $(\Delta x(i_1 + 0.5(1 + \xi_{j_1})), \dots, \Delta x(i_d + 0.5(1 + \xi_{j_d})))$. We denote $(y_{\bar{i}, \tilde{j}})_{\bar{i}, \tilde{j}}$ the set of all the points of the grid over the whole domain.

We notice that for a regular mesh with constant volume Δx^d , we have the following relation for all $x \in \mathbb{R}^d$:

$$\min_{\bar{i}, \tilde{j}} |x - y_{\bar{i}, \tilde{j}}| \leq \Delta x. \quad (1.2)$$

1.2 Temporal discretization for the HJB equation

The equation (2.1) is discretized in time by the scheme proposed by Camilli Falcone [14] for a discretization in time h .

$$\begin{aligned} v_h(t+h, x) &= \inf_{a \in A} \left[\sum_{i=1}^q \frac{1}{2q} (v_h(t, \phi_{a,h,i}^+(t, x)) + v_h(t, \phi_{a,h,i}^-(t, x))) \right. \\ &\quad \left. + f_a(t, x)h + c_a(t, x)hv_h(t, x) \right] \\ &:= v_h(t, x) + \inf_{a \in A} L_{a,h}(v_h)(t, x) \end{aligned} \quad (1.3)$$

with

$$\begin{aligned} L_{a,h}(v_h)(t, x) &= \sum_{i=1}^q \frac{1}{2q} (v_h(t, \phi_{a,h,i}^+(t, x)) + v_h(t, \phi_{a,h,i}^-(t, x)) - 2v_h(t, x)) \\ &\quad + hc_a(t, x)v_h(t, x) + hf_a(t, x) \\ \phi_{a,h,i}^+(t, x) &= x + b_a(t, x)h + (\sigma_a)_i(t, x)\sqrt{hq} \\ \phi_{a,h,i}^-(t, x) &= x + b_a(t, x)h - (\sigma_a)_i(t, x)\sqrt{hq} \end{aligned}$$

where $(\sigma_a)_i$ is the i -th column of σ_a . It is noted that it is also possible to choose other types of discretization in the same style as those defined in [36].

In order to define the solution at each date, a condition on the value chosen for v_h between 0 and h is required. We choose a temporal linear interpolation once the solution has been calculated at the date h :

$$v_h(t, x) = (1 - \frac{t}{h})g(x) + \frac{t}{h}v_h(h, x), \forall t \in [0, h]. \quad (1.4)$$

We first recall the following result:

Proposition 2 *Under the condition on the coefficients given by the equation (1.1), the solution v_h of the equations (1.3) and (1.4) is uniquely defined and belongs to $C_1(Q)$. We check that if $h \leq (16 \sup_a \{|\sigma_a|_1^2 + |b_a|_1^2 + 1\} \wedge 2 \sup_a |c_a|_0)^{-1}$, there is C such that*

$$|v - v_h|_0 \leq Ch^{\frac{1}{4}}. \quad (1.5)$$

Moreover, there exists C independent of h such that

$$|v_h|_0 \leq C, \quad (1.6)$$

$$|v_h(t, x) - v_h(t, y)| \leq C|x - y|, \forall (x, y) \in Q^2. \quad (1.7)$$

1.3 Space interpolation

The spacial resolution of the equation (1.3) is obtained on a grid. The ϕ^+ and ϕ^- must be calculated using an interpolator I such that:

$$\begin{aligned} v_h(t, \phi_{a,h,i}^+(t, x)) &\simeq I(v_h(t, \cdot))(\phi_{a,h,i}^+(t, x)), \\ v_h(t, \phi_{a,h,i}^-(t, x)) &\simeq I(v_h(t, \cdot))(\phi_{a,h,i}^-(t, x)). \end{aligned}$$

In order to easily prove the convergence of the scheme to the viscosity solution of the problem, the monotony of the scheme is generally required leading to some linear interpolator slowly converging. An adaptation to high order interpolator where the function is smooth can be achieved using Legendre grids and Sparse grids with some truncation (see [49], [48]).

Chapter 2

C++ API

To perform the interpolation and calculate the semi-Lagrangian value

$$\sum_{i=1}^q \frac{1}{2q} (v_h(t, \phi_{a,h,i}^+(t, x)) + v_h(t, \phi_{a,h,i}^-(t, x)))$$

a first object `SemiLagrangEspCond` is available:

```
1 // Copyright (C) 2016 EDF
2 // All Rights Reserved
3 // This code is published under the GNU Lesser General Public License (GNU LGPL)
4 #ifndef SEMILAGRANGESPCOND_H
5 #define SEMILAGRANGESPCOND_H
6 #include <Eigen/Dense>
7 #include <map>
8 #include <array>
9 #include <vector>
10 #include "StOpt/core/utils/constant.h"
11 #include "StOpt/core/grids/InterpolatorSpectral.h"
12
13 /** \file SemiLagrangEspCond.h
14 * \brief Semi Lagrangian method for process \f$ d x_t = b dt + \sigma dW_t \f$
15 * where \f$ X_t, b \f$ with values in \f$ \{\mathbb{R}\}^n \f$, \f$ \sigma \f$ a \f$ \mathbb{R}^n \f$
16 * \times \mathbb{R}^m \f$ matrix and \f$ W_t \f$ with values in \f$ \mathbb{R}^m \f$
17 */
18
19 namespace StOpt
20 {
21
22 /// \class SemiLagrangEspCond SemiLagrangEspCond.h
23 /// calculate semi Lagrangian operator for previously defined process.
24 class SemiLagrangEspCond
25 {
26     ///\brief interpolator
27     std::shared_ptr<InterpolatorSpectral<double>> m_interpolator;
28
29     /// \brief store extremal values for the grid (min, max coordinates in each dimension)
30     std::vector<std::array<double, 2>> m_extremalValues;
31
32     /// \brief Do we use modification of volatility to stay in the domain
33     bool m_bModifVol ;
34
35 public :
36
37     /// \brief Constructor
38     /// \param p_interpolator Interpolator storing the grid
39     /// \param p_extremalValues Extremal values of the grid
```

```

40  /// \param p_bModifVol      do we modify volatility to stay in the domain.
41  ///                        If activated, when not modification of volatility give a
42  point inside the domain, truncation is achieved
43  SemiLagrangEspCond(const std::shared_ptr<InterpolatorSpectral<double>> &p_interpolator,
44  const std::vector<std::array< double, 2> > &p_extremalValues, const bool &
45  p_bModifVol);
46
47  /// \brief Calculate \f$ \frac{1}{2d} \sum_{i=1}^d \phi(x+ b dt + \sigma_i \sqrt{dt}) +
48  \phi(x+ b dt - \sigma_i \sqrt{dt}) \f$
49  /// where \f$ \sigma_i \f$ is column \f$ i \f$ of \f$ \sigma \f$
50  /// \param p_x              beginning point
51  /// \param p_b              trend
52  /// \param p_sig            volatility matrix
53  /// \param p_dt            Time step size
54  /// \return (the value calculated, true) if point inside the domain, otherwise (0.,
55  false)
56  std::pair<double, bool> oneStep(const Eigen::ArrayXd &p_x, const Eigen::ArrayXd &p_b
57  , const Eigen::ArrayXXd &p_sig, const double &p_dt) const;
58
59 };
60 }
61 #endif

```

Its constructor uses the following arguments:

- a first `p_interpolator` defines a “spectral” interpolator on a grid: this “spectral” interpolator is built from a grid and a function to be interpolated (see section 1). In our case, it will be used to interpolate the solution from the previous time step,
- a second `p_extremalValues` defines for each dimension the minimum and maximum coordinates of the points belonging to the grid,
- a third one `p_bModifVol` if set to `true` allows special processing when the points to be interpolated are outside the grid: the volatility of the underlying process is modified (keeping the same mean and the same variance) trying to keep points inside the domain (see [49]).

This object has the `oneStep` method taking

- `p_x` the foot of the characterize (for each dimension),
- `p_b` the process trend (for each dimension),
- `p_sig` the volatility of the process matrix,

such that the interpolation is carried out for a time step h at the points $p_x + p_bh \pm p_sig\sqrt{h}$. It returns a pair (a, b) where a contains the calculated value if the value b is `true`. When interpolation is not possible, the value b is set to `false`.

In order to use the API, an object deriving from the `OptimizerSLBase` object must be constructed. This object is used to define the PDE to be solved (with its optimization problem if applicable).

```

1  // Copyright (C) 2016 EDF
2  // All Rights Reserved
3  // This code is published under the GNU Lesser General Public License (GNU LGPL)
4  #ifndef OPTIMIZERSLBASE_H

```

```

5 #define OPTIMIZERSLBASE_H
6 #include <vector>
7 #include <Eigen/Dense>
8 #include "StOpt/core/grids/SpaceGrid.h"
9 #include "StOpt/core/grids/FullGrid.h"
10 #include "StOpt/core/grids/InterpolatorSpectral.h"
11 #include "StOpt/semilagrangien/SemiLagrangEspCond.h"
12
13 /** \file OptimizerSLBase.h
14  * \brief Define an abstract class for Dynamic Programming problems
15  * \author Xavier Warin
16  */
17
18 namespace StOpt
19 {
20
21 /// \class OptimizerSLBase OptimizerSLBase.h
22 /// Base class for optimizer for resolution by semi Lagrangian methods of HJB equations
23 class OptimizerSLBase
24 {
25
26
27 public :
28
29     OptimizerSLBase() {}
30
31     virtual ~OptimizerSLBase() {}
32
33
34     /// \brief define the diffusion cone for parallelism
35     /// \param p_regionByProcessor region (min max) treated by the processor for
36     /// the different regimes treated
37     /// \return returns in each dimension the min max values in the stock that can be
38     /// reached from the grid p_gridByProcessor for each regime
39     virtual std::vector< std::array< double, 2> > getCone(const std::vector< std::array<
40     double, 2> > &p_regionByProcessor) const = 0;
41
42
43     /// \brief defines the dimension to split for MPI parallelism
44     /// For each dimension return true is the direction can be split
45     virtual Eigen::Array< bool, Eigen::Dynamic, 1> getDimensionToSplit() const = 0 ;
46
47
48     /// \brief defines a step in optimization
49     /// \param p_point coordinates of the point to treat
50     /// \param p_semiLag semi Lagrangian operator for each regime for solution at the
51     /// previous step
52     /// \param p_time current date
53     /// \param p_phiInPt value of the function at the previous time step at p_point for
54     /// each regime
55     /// \return a pair :
56     /// - first an array of the solution (for each regime)
57     /// - second an array of the optimal controls ( for each control)
58     virtual std::pair< Eigen::ArrayXd, Eigen::ArrayXd> stepOptimize(const Eigen::ArrayXd
59     &p_point,
60     const std::vector< std::shared_ptr<SemiLagrangEspCond> > &p_semiLag,
61     const double &p_time,
62     const Eigen::ArrayXd &p_phiInPt) const = 0;
63
64
65     /// \brief defines a step in simulation
66     /// \param p_gridNext grid at the next step
67     /// \param p_semiLag semi Lagrangian operator at the current step in each regime
68     /// \param p_state state array (can be modified)
69     /// \param p_iReg regime number
70     /// \param p_gaussian unitary Gaussian realization
71     /// \param p_phiInPt value of the function at the next time step at p_point for
72     /// each regime
73     /// \param p_phiInOut defines the value functions (modified) to follow
74     virtual void stepSimulate(const SpaceGrid &p_gridNext,
75     const std::vector< std::shared_ptr< StOpt::

```

```

67         SemiLagrangEspCond> > &p_semiLag,
68         Eigen::Ref<Eigen::ArrayXd> p_state,    int &p_iReg,
69         const Eigen::ArrayXd &p_gaussian,
70         const Eigen::ArrayXd &p_phiInPt,
71         Eigen::Ref<Eigen::ArrayXd> p_phiInOut) const = 0 ;
72
73     /// \brief defines a step in simulation using the control calculated in optimization
74     /// \param p_gridNext      grid at the next step
75     /// \param p_controlInterp the optimal controls interpolator
76     /// \param p_state         state array (can be modified)
77     /// \param p_iReg          regime number
78     /// \param p_gaussian      unitary Gaussian realization
79     /// \param p_phiInOut      defines the value functions (modified) to follow
80     virtual void stepSimulateControl(const SpaceGrid &p_gridNext,
81                                     const std::vector< std::shared_ptr<
82                                         InterpolatorSpectral<double>>> &p_controlInterp
83                                     ,
84                                     Eigen::Ref<Eigen::ArrayXd> p_state,    int &p_iReg,
85                                     const Eigen::ArrayXd &p_gaussian,
86                                     Eigen::Ref<Eigen::ArrayXd> p_phiInOut) const = 0 ;
87
88     /// \brief get number of regimes
89     virtual int getNbRegime() const = 0 ;
90
91     /// \brief get back the dimension of the control
92     virtual int getNbControl() const = 0 ;
93
94     /// \brief do we modify the volatility to stay in the domain
95     virtual bool getBModifVol() const = 0 ;
96
97     /// \brief get the number of Brownians involved in semi Lagrangian for simulation
98     virtual int getBrownianNumber() const = 0 ;
99
100    /// \brief get size of the function to follow in simulation
101    virtual int getSimuFuncSize() const = 0;
102
103    /// \brief Permit to deal with some boundary points that do not need boundary
104    ///        conditions
105    ///        Return false if all points on the boundary need some boundary conditions
106    /// \param p_point potentially on the boundary
107    virtual bool isNotNeedingBC(const Eigen::ArrayXd &p_point) const = 0;
108 };
109 }
110 #endif /* OPTIMIZERSLBASE_H */

```

The main methods associated with this object are:

- **stepOptimize** is used to calculate the solution of the PDE at a point.
 - It takes a point from the used grid **p_point**,
 - and applies the semi-Lagrangian scheme **p_semiLag** at this point,
 - on a date given by **p_time**.

It returns a pair containing:

- the value of the function calculated at **p_point** for each regime,
 - the optimal control calculated at **p_point** for each control.
- **stepSimulate** is used when the PDE is associated with an optimization problem and we want to simulate an optimal policy using the function values calculated in the optimization part. The arguments are:

- `p_gridNext` defining the grid used at the next time step,
 - `p_semiLag` the semi-Lagrangian operator built with an interpolator using the following temporal solution,
 - `p_state` the vector defining the current state for the current regime,
 - `p_iReg` the current regime number,
 - `p_gaussian` is the vector of Gaussian random variables used to calculate the Brownian involved in the underlying process of the current simulation,
 - `p_phiInP` to the value of the function calculated in optimization at the next time step for the given point,
 - `p_phiInOut` storing the cost functions: the size of the array is the number of functions to follow in simulation.
- `stepSimulateControl` is used when the PDE is associated with an optimization problem and we want to simulate an optimal policy using the optimal controls calculated in the optimization part. The arguments are:
 - `p_gridNext` defining the grid used at the next time step,
 - `p_controlInterp` a vector (for each control) of interpolators in controls
 - `p_state` the vector defining the current state for the current regime,
 - `p_iReg` the current regime number,
 - `p_gaussian` is the vector of Gaussian random variables used to calculate the Brownian involved in the underlying process of the current simulation.
 - `p_phiInOut` storing the cost functions: the size of the array is the number of functions to follow in simulation.

On return, the vector `p_state` is modified, the `p_iReg` is modified and the cost function `p_phiInOut` is modified for the current trajectory.

- the `getCone` method is only relevant if the data distribution (therefore MPI) is used. As argument it takes a vector of size the dimension of the grid. Each component of the vector is an array containing the minimum and maximum coordinate values of the points of the current grid defining a hyper cube $H1$. It returns for each dimension, the min and max coordinates of the hyper cube $H2$ containing the points that can be reached by applying a command from a point of the grid in $H1$. If no optimization is carried out, it returns the hyper cube $H2$ containing the points that can be reached by the semi-Lagrangian diagram. For an explanation of the parallel formalism, see the 3 chapter.
- the `getDimensionToSplit` method is only relevant if the data distribution (therefore MPI) is used. The method makes it possible to define the directions to be divided for the distribution of solution on the processors. For each dimension, it returns a Boolean where `true` means that the direction is a candidate for splitting,

- the `isNotNeedingBC` allows to define for a point on the limit of the grid if a boundary condition is necessary (`true` is returned) or if no limit is necessary (return `false`).

An example of a derived object of such an optimizer for a simple stochastic target problem (described in paragraph 5.3.4 in [49]) is given below:

```

1 #include <iostream>
2 #include "StOpt/core/utils/constant.h"
3 #include "test/c++/tools/semilagrangien/OptimizeSLCase3.h"
4
5 using namespace StOpt;
6 using namespace Eigen ;
7 using namespace std ;
8
9 OptimizerSLCase3::OptimizerSLCase3(const double &p_mu, const double &p_sig, const double &
10     p_dt, const double &p_alphaMax, const double &p_stepAlpha):
11     m_dt(p_dt), m_mu(p_mu), m_sig(p_sig), m_alphaMax(p_alphaMax), m_stepAlpha(p_stepAlpha)
12 {}
13
14 vector< std::array< double, 2> > OptimizerSLCase3::getCone(const vector< std::array<
15     double, 2> > &p_xInit) const
16 {
17     vector< std::array< double, 2> > xReached(1);
18     xReached[0][0] = p_xInit[0][0] - m_alphaMax * m_mu / m_sig * m_dt - m_alphaMax * sqrt
19         (m_dt);
20     xReached[0][1] = p_xInit[0][1] + m_alphaMax * sqrt(m_dt) ;
21     return xReached;
22 }
23
24 pair< ArrayXd, ArrayXd> OptimizerSLCase3::stepOptimize(const ArrayXd &p_point,
25     const vector< shared_ptr<SemiLagrangEspCond> > &p_semiLag, const double &, const
26     Eigen::ArrayXd &) const
27 {
28     pair< ArrayXd, ArrayXd> solutionAndControl;
29     solutionAndControl.first.resize(1);
30     solutionAndControl.second.resize(1);
31     ArrayXd b(1);
32     ArrayXXd sig(1, 1) ;
33     double vMin = StOpt::infy;
34     for (int iA1 = 0; iA1 < m_alphaMax / m_stepAlpha; ++iA1)
35     {
36         double alpha = iA1 * m_stepAlpha;
37         b(0) = -alpha * m_mu / m_sig; // trend
38         sig(0) = alpha; // volatility with one Brownian
39         pair<double, bool> lagrang = p_semiLag[0]->oneStep(p_point, b, sig, m_dt); // test
40             the control
41         if (lagrang.second)
42         {
43             if (lagrang.first < vMin)
44             {
45                 vMin = lagrang.first;
46                 solutionAndControl.second(0) = alpha;
47             }
48         }
49     }
50     solutionAndControl.first(0) = vMin;
51     return solutionAndControl;
52 }
53
54 void OptimizerSLCase3::stepSimulate(const StOpt::SpaceGrid &p_gridNext,
55     const std::vector< shared_ptr< StOpt::
56         SemiLagrangEspCond > > &p_semiLag,
57     Eigen::Ref<Eigen::ArrayXd> p_state, int &,
58     const Eigen::ArrayXd &p_gaussian, const Eigen::ArrayXd
59         &,
60     Eigen::Ref<Eigen::ArrayXd>) const

```

```

55     double vMin = StOpt::infy;
56     double alphaOpt = -1;
57     ArrayXd b(1);
58     ArrayXXd sig(1, 1) ;
59     ArrayXd proba = p_state ;
60     // recalculate the optimal alpha
61     for (int iA1 = 0; iA1 < m_alphaMax / m_stepAlpha; ++iA1)
62     {
63         double alpha = iA1 * m_stepAlpha;
64         b(0) = -alpha * m_mu / m_sig; // trend
65         sig(0) = alpha; // volatility with one Brownian
66         pair<double, bool> lagrang = p_semiLag[0]->oneStep(proba, b, sig, m_dt); // test the
        control
67         if (lagrang.second)
68         {
69             if (lagrang.first < vMin)
70             {
71                 vMin = lagrang.first;
72                 alphaOpt = alpha;
73             }
74         }
75     }
76     proba(0) += alphaOpt * p_gaussian(0) * sqrt(m_dt);
77     // truncate if necessary
78     p_gridNext.truncatePoint(proba);
79     p_state = proba ;
80 }
81 }
82
83
84 void OptimizerSLCase3:: stepSimulateControl(const SpaceGrid &p_gridNext,
85     const vector< shared_ptr< InterpolatorSpectral<double>> > &p_controlInterp,
86     Eigen::Ref<Eigen::ArrayXd> p_state, int &,
87     const ArrayXd &p_gaussian,
88     Eigen::Ref<Eigen::ArrayXd>) const
89 {
90     ArrayXd proba = p_state ;
91     double alphaOpt = p_controlInterp[0]->apply(p_state);
92     proba(0) += alphaOpt * p_gaussian(0) * sqrt(m_dt);
93     // truncate if necessary
94     p_gridNext.truncatePoint(proba);
95     p_state = proba ;
96 }

```

2.1 PDE resolution

Once the problem is described, temporal recursion can be achieved by using the **Transition StepSemilagrang** object in a sequential resolution of the problem. This object allows to solve the problem in one time step.

```

1 // Copyright (C) 2016 EDF
2 // All Rights Reserved
3 // This code is published under the GNU Lesser General Public License (GNU LGPL)
4 #ifndef TRANSITIONSTEPSEMILAGRANG_H
5 #define TRANSITIONSTEPSEMILAGRANG_H
6 #ifdef OMP
7 #include <omp.h>
8 #endif
9 #ifdef USE_MPI
10 #include <boost/mpi.hpp>
11 #endif
12 #include <functional>
13 #include <memory>
14 #include <Eigen/Dense>

```

```

15 #include "geners/BinaryFileArchive.hh"
16 #include "StOpt/semilagrangien/TransitionStepSemilagrangBase.h"
17 #include "StOpt/core/grids/SpaceGrid.h"
18 #include "StOpt/core/grids/InterpolatorSpectral.h"
19 #include "StOpt/semilagrangien/OptimizerSLBase.h"
20
21 /** \file TransitionStepSemilagrang.h
22  * \brief Solve one step of explicit semi Lagrangian scheme
23  * \author Xavier Warin
24  */
25
26
27 namespace StOpt
28 {
29
30 /// \class TransitionStepSemilagrang TransitionStepSemilagrang.h
31 /// One step of semi Lagrangian scheme
32 class TransitionStepSemilagrang : public TransitionStepSemilagrangBase
33 {
34 private :
35
36     std::shared_ptr<SpaceGrid> m_gridCurrent ; ///< global grid at current time step
37     std::shared_ptr<SpaceGrid> m_gridPrevious ; ///< global grid at previous time step
38     std::shared_ptr<OptimizerSLBase > m_optimize ; ///< optimizer solving the problem for
39         one point and one step
40
41 #ifdef USE_MPI
42     boost::mpi::communicator m_world; ///< Mpi communicator
43 #endif
44
45 public :
46
47     /// \brief Constructor
48     TransitionStepSemilagrang(const std::shared_ptr<SpaceGrid> &p_gridCurrent,
49                             const std::shared_ptr<SpaceGrid> &p_gridPrevious,
50                             const std::shared_ptr<OptimizerSLBase > &p_optimize
51                             , const boost::mpi::communicator &p_world
52                             );
53
54     /// \brief One time step for resolution
55     /// \param p_phiIn for each regime the function value ( on the grid)
56     /// \param p_time current date
57     /// \param p_boundaryFunc Function at the boundary to impose Dirichlet conditions (
58     /// depending on regime and position)
59     /// \return solution obtained after one step of dynamic programming and the optimal
60     /// control
61     std::pair< std::vector< std::shared_ptr< Eigen::ArrayXd > >, std::vector< std::
62         shared_ptr< Eigen::ArrayXd > > > oneStep(const std::vector< std::shared_ptr<
63         Eigen::ArrayXd > > &p_phiIn, const double &p_time, const std::function<double(
64         const int &, const Eigen::ArrayXd &)> &p_boundaryFunc) const;
65
66     /// \brief Permits to dump continuation values on archive
67     /// \param p_ar archive to dump in
68     /// \param p_name name used for object
69     /// \param p_iStep Step number or identifier for time step
70     /// \param p_phiIn for each regime the function value
71     /// \param p_control for each control, the optimal value
72     void dumpValues(std::shared_ptr<gs::BinaryFileArchive> p_ar, const std::string &p_name,
73         const int &p_iStep, const std::vector< std::shared_ptr< Eigen::ArrayXd > > &
74         p_phiIn,
75         const std::vector< std::shared_ptr< Eigen::ArrayXd > > &p_control)
76         const;
77 };
78
79 #endif /* TRANSITIONSTEPSEMILAGRANG_H */

```

Its constructor takes the following arguments:

Table 2.1: Which object `TransitionStepSemilagrang` to use depending on the grid used and the type of parallelization used.

	Full grid	Sparse grid
Sequential	<code>TransitionStepSemilagrang</code>	<code>TransitionStepSemilagrang</code>
Parallelization on calculations threads and MPI	<code>TransitionStepSemilagrang</code>	<code>TransitionStepSemilagrang</code>
Distribution of calculations and data (MPI)	<code>TransitionStepSemilagrangDist</code>	Not available

- `p_gridCurrent` a grid describing the meshes at the current date,
- `p_gridPrevious` a grid describing the meshes at the previously processed date,
- `p_optimize` an object derived from `OptimizerSLBase` and describing the problem to be solved at a given date and point in the current grid.

A first method `oneStep` takes the following arguments:

- `p_phiIn` describes for each regime the solution previously calculated on the grid at the previous time,
- `p_time` is the current time step,
- `p_boundaryFunc` is a function giving the Dirichlet solution of the problem according to the number of regimes and the position on the boundary.

It returns an estimate of the solution at the current date on the current grid for all the regimes and an estimate of the optimal control calculated for all the controls.

A last method `dumpValues` allows to dump calculated the solution `p_phiIn` at step `p_istep+1` and optimal control at step `p_istep` in a `p_ar` archive.

A version using the distribution of data and calculations can be found in the `TransitionStepSemilagrangDist` object. An example of sequential temporal recursion can be found in the function `semiLagrangianTime` and an example with distribution can be found in the `semiLagrangianTimeDist` function. In the two functions developed in the test chapter, the analytical solution of the problem is known and compared to the numerical estimate obtained with the semi-Lagrangian method.

2.2 Simulation framework

Once the optimal controls and value functions have been calculated, the optimal policy can be simulated using the function values (recalculate the optimal control for each simulation) or by directly using the optimal controls calculated in optimization

- Calculate the optimal strategy in simulation using the function values calculated in optimization:
In order to simulate an optimal policy step, a `SimulateStepSemilagrangDist` object is provided with constructor

```

1 SimulateStepSemilagrangDist(gs::BinaryFileArchive &p_ar,  const int &p_iStep,
2     const std::string &p_name ,
3         const std::shared_ptr<FullGrid> &p_gridNext, const std
4             ::shared_ptr<OptimizerSLBase > &p_pOptimize ,
3         const bool &p_bOneFile,
4         const boost::mpi::communicator &p_world);

```

where

- `p_ar` is the binary archive where the continuation values are stored,
- `p_iStep` is the number associated with the current time step (0 at the start date of the simulation, the number is increased by one at each simulated time step),
- `p_name` is the base name to search for in the archive,
- `p_GridNext` is the grid at the next time step (`p_iStep+1`),
- `p_Optimize` is the Optimizer describing the passage from one time step to the next,
- `p_OneFile` is equal to `true` if only one archive is used to store continuation values.
- `p_world` The MPI communicator

Remark 47 *A version without data distribution but only multithreaded and parallel with MPI on data is available with the object `SimulateStepSemilagrang`.*

This object implements the `oneStep` method

```

1 void oneStep(const Eigen::ArrayXXd & p_gaussian, Eigen::ArrayXXd &p_statevector , Eigen
2     ::ArrayXi &p_iReg, Eigen::ArrayXd &p_phiInOuts)

```

where:

- `p_gaussian` is a two-dimensional array (number of Brownian in the modeling by the number of Monte Carlo simulations).
- `p_statevector` stores the continuous state (size of continuous state per number of simulations)
- `p_iReg` for each simulation, gives the current regime number,
- `p_phiInOut` stores the gain/cost functions for all simulations: it is updated by the function call. The size of the array is $(nb, nbSimul)$ where nb is given by the `getSimuFuncSize` method of the optimizer and $nbSimul$ the number of Monte Carlo simulations.

Remark 48 *The previous object `SimulateStepSemilagrangDist` is used with MPI for high dimensional problems. In the case of small dimension (less than or equal to three), parallelization with MPI or the sequential calculations can be carried out by the object `SimulateStepSemilagrang`.*

An example of using this method to simulate an optimal policy with distribution is given below:

```

1 // Copyright (C) 2016 EDF
2 // All Rights Reserved
3 // This code is published under the GNU Lesser General Public License (GNU LGPL)
4 #ifdef USE_MPI
5 #include <boost/random.hpp>
6 #include <boost/mpi.hpp>
7 #include <memory>
8 #include <Eigen/Dense>
9 #include "geners/BinaryFileArchive.hh"
10 #include "StOpt/semilagrangien/OptimizerSLBase.h"
11 #include "StOpt/semilagrangien/SimulateStepSemilagrangDist.h"
12
13 using namespace std;
14
15 double semilagrangianSimuDist(const shared_ptr<StOpt::FullGrid> &p_grid,
16                               const shared_ptr<StOpt::OptimizerSLBase> &p_optimize,
17                               const function<double(const int &, const Eigen::ArrayXd
18                                     &> &p_funcFinalValue,
19                               const int &p_nbStep,
20                               const Eigen::ArrayXd &p_stateInit,
21                               const int &p_initialRegime,
22                               const int &p_nbSimul,
23                               const string &p_fileToDump,
24                               const bool &p_bOneFile,
25                               const boost::mpi::communicator &p_world)
26 {
27     // store states in a regime
28     Eigen::ArrayXXd states(p_stateInit.size(), p_nbSimul);
29     for (int is = 0; is < p_nbSimul; ++is)
30         states.col(is) = p_stateInit;
31     // store the regime number
32     Eigen::ArrayXi regime = Eigen::ArrayXi::Constant(p_nbSimul, p_initialRegime);
33     // test if one file generated
34     string toDump = p_fileToDump;
35     if (!p_bOneFile)
36         toDump += "_" + boost::lexical_cast<string>(p_world.rank());
37     gs::BinaryFileArchive ar(toDump.c_str(), "r");
38     // name for continuation object in archive
39     string nameAr = "Continuation";
40     // cost function
41     Eigen::ArrayXXd costFunction = Eigen::ArrayXXd::Zero(p_optimize->getSimuFuncSize(), p_nbSimul);
42     // random generator and Gaussian variables
43     boost::mt19937 generator;
44     boost::normal_distribution<double> normalDistrib;
45     boost::variate_generator<boost::mt19937 &, boost::normal_distribution<double> >
46         normalRand(generator, normalDistrib);
47     Eigen::ArrayXXd gaussian(p_optimize->getBrownianNumber(), p_nbSimul);
48     // iterate on time steps
49     for (int istep = 0; istep < p_nbStep; ++istep)
50     {
51         for (int is = 0; is < gaussian.cols(); ++is)
52             for (int id = 0; id < gaussian.rows(); ++id)
53                 gaussian(id, is) = normalRand();
54
55         StOpt::SimulateStepSemilagrangDist(ar, p_nbStep - 1 - istep, nameAr, p_grid,
56             p_optimize, p_bOneFile, p_world).oneStep(gaussian, states, regime,
57             costFunction);
58     }
59     // final cost to add
60     for (int is = 0; is < p_nbSimul; ++is)
61         costFunction(0, is) += p_funcFinalValue(regime(is), states.col(is));
62     // average gain/cost
63     return costFunction.mean();
64 }
65 #endif

```

A sequential or parallel version on the calculations of the previous example is given in the file `semiLagrangianSimuDist.cpp`.

- Calculate the optimal strategy in simulation
by interpolation of the optimal control calculated in optimization:
In order to simulate an optimal policy step, a `SimulateStepSemilagrangControlDist` object is provided with constructor

```

1 SimulateStepSemilagrangControlDist(gs::BinaryFileArchive &p_ar,  const int &
2   p_iStep,  const std::string &p_name ,
3           const std::shared_ptr<FullGrid> &p_gridCur,
4           const std::shared_ptr<FullGrid> &p_gridNext,
5           const std::shared_ptr<OptimizerSLBase > &
6           p_pOptimize ,
7           const bool &p_bOneFile,
8           const boost::mpi::communicator &p_world )

```

where

- `p_ar` is the binary archive where the continuation values are stored,
- `p_iStep` is the number associated with the current time step (0 at the start date of simulation, the number is increased by one at each simulated time step),
- `p_name` is the base name to search for in the archive,
- `p_GridCur` is the grid at the current time step (`p_iStep`),
- `p_GridNext` is the grid at the next time step (`p_iStep+1`),
- `p_Optimize` is the Optimizer describing the passage from one time step to the next,
- `p_OneFile` is equal to `true` if only one archive is used to store continuation values.
- `p_world` The MPI communicator

Remark 49 *The previous object `SimulateStepSemilagrangControlDist` is used with the MPI distribution of data for high dimensional problems. In the case of small dimension (less than or equal to three), the parallelization with MPI or the sequential calculations can be carried out by the object `SimulateStepSemilagrangControl`.*

This object implements the method `oneStep`

```

1 void oneStep((const Eigen::ArrayXXd & p_gaussian, Eigen::ArrayXXd &p_statevector ,
2             Eigen::ArrayXi &p_iReg, Eigen::ArrayXd &p_phiInOuts)

```

where:

- `p_gaussian` is a two dimensional array (number of Brownian in the modeling by the number of Monte Carlo simulations).
- `p_statevector` store continuous state (size of continuous state per number of simulations)
- `p_iReg` for each simulation gives the current regime number,

- `p_phiInOut` stores the gain/cost functions for all simulations: it is updated by the function call. The size of the array is $(nb, nbSimul)$ where nb is given by the `getSimuFuncSize` method of the optimizer and $nbSimul$ the number of Monte Carlo simulations.

An example of using this method to simulate an optimal policy with distribution is given below:

```

1 // Copyright (C) 2016 EDF
2 // All Rights Reserved
3 // This code is published under the GNU Lesser General Public License (GNU LGPL)
4 #ifdef USE_MPI
5 #include <memory>
6 #include <boost/random.hpp>
7 #include <boost/mpi.hpp>
8 #include <Eigen/Dense>
9 #include "gers/BinaryFileArchive.hh"
10 #include "StOpt/semilagrangien/OptimizerSLBase.h"
11 #include "StOpt/semilagrangien/SimulateStepSemilagrangControlDist.h"
12
13 using namespace std;
14
15 double semiLagrangianSimuControlDist(const shared_ptr<StOpt::FullGrid> &p_grid,
16                                     const shared_ptr<StOpt::OptimizerSLBase> &
17                                     p_optimize,
18                                     const function<double(const int &, const Eigen::
19                                     ArrayXd &)> &p_funcFinalValue,
20                                     const int &p_nbStep,
21                                     const Eigen::ArrayXd &p_stateInit,
22                                     const int &p_initialRegime,
23                                     const int &p_nbSimul,
24                                     const string &p_fileToDump,
25                                     const bool &p_bOneFile,
26                                     const boost::mpi::communicator &p_world)
27 {
28     // store states in a regime
29     Eigen::ArrayXXd states(p_stateInit.size(), p_nbSimul);
30     for (int is = 0; is < p_nbSimul; ++is)
31         states.col(is) = p_stateInit;
32     // store the regime number
33     Eigen::ArrayXi regime = Eigen::ArrayXi::Constant(p_nbSimul, p_initialRegime);
34     // test if one file generated
35     string toDump = p_fileToDump;
36     if (!p_bOneFile)
37         toDump += "_" + boost::lexical_cast<string>(p_world.rank());
38     gs::BinaryFileArchive ar(toDump.c_str(), "r");
39     // name for continuation object in archive
40     string nameAr = "Continuation";
41     // cost function
42     Eigen::ArrayXXd costFunction = Eigen::ArrayXXd::Zero(p_optimize->getSimuFuncSize
43     (), p_nbSimul);
44     // random generator and Gaussian variables
45     boost::mt19937 generator;
46     boost::normal_distribution<double> normalDistrib;
47     boost::variate_generator<boost::mt19937 &, boost::normal_distribution<double> >
48     normalRand(generator, normalDistrib);
49     Eigen::ArrayXXd gaussian(p_optimize->getBrownianNumber(), p_nbSimul);
50     // iterate on time steps
51     for (int istep = 0; istep < p_nbStep; ++istep)
52     {
53         for (int is = 0; is < gaussian.cols(); ++is)
54             for (int id = 0; id < gaussian.rows(); ++id)
55                 gaussian(id, is) = normalRand();
56
57         StOpt::SimulateStepSemilagrangControlDist(ar, p_nbStep - 1 - istep, nameAr,
58         p_grid, p_grid, p_optimize, p_bOneFile, p_world).oneStep(gaussian, states,

```

```

        regime, costFunction);
54     }
55     // final cost to add
56     for (int is = 0; is < p_nbSimul; ++is)
57         costFunction(0, is) += p_funcFinalValue(regime(is), states.col(is));
58     // average gain/cost
59     return costFunction.mean();
60 }
61 #endif

```

The sequential version (or parallelized on calculations) of the preceding example is given in the file `semiLagrangianSimuControl.cpp`.

Remark 50 *In the previous example, we assume that only one function is tracked in simulation, and that we return an average for that function value accordingly.*

Table 2.2: Which simulation object to use depending on the TransitionStepSemilagrang object used.

	TransitionStepSemilagrang	TransitionStepSemilagrangDist bOneFile = true	TransitionStepSemilagrangDist bOneFile = false
SimulateStepSemilagrang	Yes	Yes	No
SimulateStepSemilagrangControl	Yes	Yes	No
SimulateStepSemilagrangDist	No	Yes	Yes
SimulateStepSemilagrangControlDist	No	Yes	Yes

Part V

An example with both dynamic
programming with regression and
PDE

In this chapter we give an example where dynamic programming with regressions and PDE can be used. It allows to compare the resolution and the solution obtained by both methods.

In this example we take the following notations:

- D_t is a demand process (in electricity) with an Ornstein–Uhlenbeck dynamic:

$$dD_t = \alpha(m - D_t)dt + \sigma dW_t,$$

- Q_t is the cumulative carbon emission due to the production of electricity to meet demand,

$$dQ_t = (D_t - L_t)^+ dt,$$

- L_t the total investment capacity in non-emissive technology to produce electricity

$$L_t = \int_0^t l_s ds$$

where l_s is an investment intensity in non-emissive technology at the date s ,

- Y_t is the carbon price where

$$Y_t = \mathbb{E}_t(\lambda 1_{Q_T \geq H}),$$

with λ and H given.

We introduce the following functions:

- the function of the price of electricity which is a function of the demand and the global investment of non-emitting technology.

$$p_t = (1 + D_t)^2 - L_t,$$

- the profit function by selling electricity is given by

$$\Pi(D_t, L_t) = p_t D_t - (D_t - L_t)^+,$$

- $\tilde{c}(l_t, L_t)$ is the investment cost for new non-emissive technology capabilities.

$$\tilde{c}(l, L) = \bar{\beta}(c_\infty + (c_0 - c_\infty)e^{\beta L})(1 + l)l$$

The value of the company that sells electricity is given by $V(t, D_t, Q_t, L_t)$. It satisfies the coupling equations:

$$\begin{cases} \partial_t v + \alpha(m - D)\partial_D v + \frac{1}{2}\sigma^2\partial_{DD}^2 v + (D - L)^+\partial_Q v + \Pi(D, L) \\ + sL^{1-\alpha} - y(D - L)^+ + \sup_l \{l\partial_L v - \tilde{c}(l, L)\} = 0 \\ v_T = 0 \end{cases} \quad (1)$$

and the carbon price $y(t, D_t, Q_t, L_t)$ is given by:

$$\begin{cases} \partial_t y + \alpha(m - D)\partial_D y + \frac{1}{2}\sigma^2\partial_{DD}^2 y + (D - L)^+\partial_Q y + l^*\partial_L y = 0 \\ y_T = \lambda 1_{Q_T \geq K} \end{cases} \quad (2)$$

and l^* is the optimal control in equation (1). The preceding equation can be solved with the semi-Lagrangian method.

After a temporal discretization with a step δt a dynamic programming equation can be given by

$$v(T - \delta t, D, Q, L) = \sup_l (\Pi(D, L) + sL^{1-\alpha} - y_{T-\delta t}(D - L)^+ - \tilde{c}(l, L))\delta t + \mathbb{E}_{T-\delta t}(V(T, D_T^{T-\delta t, D}, Q + (D - L)^+\delta t, L + l\delta t)) \quad (3)$$

$$Y(T - \delta t, D, Q, L) = \mathbb{E}_{T-\delta t}(Y(T, D_T^{T-\delta t, D}, Q + (D - L)^+\delta t, L + l^*\delta t)) \quad (4)$$

The previous equations (3) and (4) can be solved with regression methods.

In order to use the frameworks previously developed in parallel, we must define some common variables for both methods.

- The number of regimes to use (obtained by the `getNbRegime` method) is 2: one to store the value v , one for the value y ,
- In the example we want to follow during the simulations the values of the functions v and y so we fix the number of functions obtained by the `getSimuFuncSize` method to 2.
- In order to test the controls in optimization and simulation we define a maximum investment intensity `lMax` and a discretization step to test the controls `lStep`.

In the following, we store the optimal functions in optimization and recalculate the optimal control in simulation.

0.3 Dynamic programming with the regression approach

All we have to do is specify an optimizer (`OptimizedDPEmissive`) defining the methods used to optimize and simulate, and the `getCone` method for parallelization:

```

1 // Copyright (C) 2016 EDF
2 // All Rights Reserved
3 // This code is published under the GNU Lesser General Public License (GNU LGPL)
4 #include "StOpt/core/utils/constant.h"
5 #include "OptimizedDPEmissive.h"
6
7 using namespace std ;
8 using namespace StOpt;
9 using namespace Eigen;
10
11
12 // constructor
13 OptimizedDPEmissive::OptimizedDPEmissive(const double &p_alpha,
14                                           const std::function<double(double, double)> &p_PI,
15                                           const std::function<double(double, double)> &
16                                           p_cBar, const double &p_s, const double &
17                                           p_lambda,
```

```

16         const double &p_dt,
17         const double &p_maturity,
18         const double &p_lMax, const double &p_lStep, const
            std::vector< std::array< double, 2> > &
            p_extrem):
19     m_alpha(p_alpha), m_PI(p_PI),
20     m_cBar(p_cBar), m_s(p_s), m_lambda(p_lambda), m_dt(p_dt), m_maturity(p_maturity),
        m_lMax(p_lMax), m_lStep(p_lStep),
21     m_extrem(p_extrem)
22 {}
23
24 Array< bool, Dynamic, 1> OptimizedDPEmissive::getDimensionToSplit() const
25 {
26     Array< bool, Dynamic, 1> bDim = Array< bool, Dynamic, 1>::Constant(2, true);
27     return bDim ;
28 }
29
30 // for parallelism
31 std::vector< std::array< double, 2> > OptimizedDPEmissive::getCone(const vector< std::
    array< double, 2> > &p_xInit) const
32 {
33     vector< std::array< double, 2> > xReached(2);
34     xReached[0][0] = p_xInit[0][0] ; // Q only increases
35     xReached[0][1] = m_extrem[0][1] ; // whole domain due to demand which is unbounded
36     xReached[1][0] = p_xInit[1][0] ; // L only increases
37     xReached[1][1] = p_xInit[1][1] + m_lMax * m_dt ; // maximal increase given by the
        control
38     return xReached;
39 }
40
41 // one step in optimization from stock point for all simulations
42 std::pair< ArrayXXd, ArrayXXd> OptimizedDPEmissive::stepOptimize(const std::shared_ptr<
    StOpt::SpaceGrid> &p_grid, const ArrayXd &p_stock,
43     const std::vector< ContinuationValue> &p_condEsp,
44     const std::vector< std::shared_ptr< ArrayXXd > > &) const
45 {
46     std::pair< ArrayXXd, ArrayXXd> solutionAndControl;
47     // to store final solution (here two regimes)
48     solutionAndControl.first = ArrayXXd::Constant(m_simulator->getNbSimul(), 2, -StOpt::
        infy);
49     solutionAndControl.second = ArrayXXd::Constant(m_simulator->getNbSimul(), 1, -StOpt::
        infy);
50     // demand
51     ArrayXd demand = m_simulator->getParticles().array().row(0).transpose();
52     // Gain (size number of simulations)
53     ArrayXd gain(m_simulator->getNbSimul());
54     double gainSubvention = m_s * pow(p_stock(1), 1. - m_alpha); // subvention for non
        emissive energy
55     for (int is = 0 ; is < m_simulator->getNbSimul(); ++is)
56         gain(is) = m_PI(demand(is), p_stock(1)) + gainSubvention ; // gain by production
        and subvention
57     ArrayXd ptStockNext(2);
58     // time to maturity
59     double timeToMat = m_maturity - m_simulator->getCurrentStep();
60     // interpolator at the new step
61     for (int is = 0 ; is < m_simulator->getNbSimul(); ++is)
62     {
63         for (int iA1 = 0; iA1 < m_lMax / m_lStep ; ++iA1) // test all command for
            investment between 0 and lMax
64         {
65             double l = iA1 * m_lStep;
66             // interpolator at the new step
67             ptStockNext(0) = p_stock(0) + std::max(demand(is) - p_stock(1), 0.) * m_dt;
68             ptStockNext(1) = p_stock(1) + l * m_dt ;
69             // first test we are inside the domain
70             if (p_grid->isInside(ptStockNext))
71             {
72                 // create an interpolator at the arrival point
73                 std::shared_ptr<StOpt::Interpolator<double>> interpolator = p_grid->

```

```

        createInterpolator(ptStockNext);
74         // calculate Y for this simulation with the optimal control
75         double yLoc = p_condEsp[1].getASimulation(is, *interpolator);
76         // local gain
77         double gainLoc = (gain(is) - yLoc * std::max(demand(is) - p_stock(1), 0.) -
            m_cBar(1, p_stock(1))) * m_dt;
78         // gain + conditional expectation of future gains
79         double condExp = gainLoc + p_condEsp[0].getASimulation(is, *interpolator);
80         if (condExp > solutionAndControl.first(is, 0)) // test optimality of the
            control
81         {
82             solutionAndControl.first(is, 0) = condExp;
83             solutionAndControl.first(is, 1) = yLoc;
84             solutionAndControl.second(is, 0) = 1;
85         }
86     }
87 }
88 // test if solution acceptable
89 if (StOpt::almostEqual(solutionAndControl.first(is, 0), - StOpt::infty, 10))
90 {
91     // fix boundary condition
92     solutionAndControl.first(is, 0) = timeToMat * (m_PI(demand(is), p_stock(1)) +
        m_s * pow(p_stock(1), 1. - m_alpha) - m_lambda * std::max(demand(is) -
        p_stock(1), 0.));
93     solutionAndControl.first(is, 1) = m_lambda ; // Q est maximal !!
94     solutionAndControl.second(is, 0) = 0. ; // fix control to zero
95 }
96 }
97 return solutionAndControl;
98 }
99
100 // one step in simulation for current simulation
101 void OptimizedDPEmissive::stepSimulate(const std::shared_ptr< StOpt::SpaceGrid> &p_grid,
    const std::vector< StOpt::GridAndRegressedValue > &p_continuation,
102                                     StOpt::StateWithStocks<double> &p_state,
103                                     Ref<ArrayXd> p_phiInOut) const
104 {
105     ArrayXd ptStock = p_state.getPtStock();
106     ArrayXd ptStockNext(ptStock.size());
107     double vOpt = - StOpt::infty;
108     double gainOpt = 0.;
109     double lOpt = 0. ;
110     double demand = p_state.getStochasticRealization()(0); // demand for this simulation
111     ptStockNext(0) = ptStock(0) + std::max(demand - ptStock(1), 0.) * m_dt;
112     double gain = m_PI(demand, ptStock(1)) + m_s * pow(ptStock(1), 1. - m_alpha) ; //
        gain from production and subvention
113     double yOpt = 0. ;
114     for (int iA1 = 0; iA1 < m_lMax / m_lStep ; ++iA1) // test all command for investment
        between 0 and lMax
115     {
116         double l = iA1 * m_lStep;
117         // interpolator at the new step
118         ptStockNext(1) = ptStock(1) + l * m_dt ;
119         // first test we are inside the domain
120         if (p_grid->isInside(ptStockNext))
121         {
122             // calculate Y for this simulation with the control
123             double yLoc = p_continuation[1].getValue(ptStockNext, p_state.
                getStochasticRealization());
124             // local gain
125             double gainLoc = (gain - yLoc * std::max(demand - ptStock(1), 0.) - m_cBar(1,
                ptStock(1))) * m_dt;
126             // gain + conditional expectation of future gains
127             double condExp = gainLoc + p_continuation[0].getValue(ptStockNext, p_state.
                getStochasticRealization());
128
129             if (condExp > vOpt) // test optimality of the control
130             {
131                 vOpt = condExp;

```

```

132         gainOpt = gainLoc;
133         l0pt = 1;
134         y0pt = yLoc;
135     }
136 }
137 }
138 p_phiInOut(0) += gainOpt; // follow v value
139 p_phiInOut(1) = y0pt ; // follow y value
140 ptStockNext(1) = ptStock(1) + l0pt * m_dt ; // update state due to control
141 p_state.setPtStock(ptStockNext);
142 }

```

This 2-dimensional case for inventories can be handled by interpolation on the full two-dimensional grid and on a two-dimensional sparse grid. The two versions of the resolution are given in a test case (`testDPNonEmissive.cpp`).

0.4 The PDE approach

The same can be done with the PDE approach using a simulator for the OU demand (`AR1Simulator`). We then define an optimizer (`OptimizeSLEmissive`) and the methods used to optimize and simulate, and the method `getCone` for parallelization:

```

1 #include <iostream>
2 #include "StOpt/core/utils/constant.h"
3 #include "OptimizeSLEmissive.h"
4
5 using namespace StOpt;
6 using namespace Eigen ;
7 using namespace std ;
8
9 // constructor
10 OptimizeSLEmissive::OptimizeSLEmissive(const double &p_alpha, const double &p_m, const
    double &p_sig, const std::function<double(double, double)> &p_PI,
11     const std::function< double(double, double) > &
        p_cBar, const double &p_s, const double &p_dt
12     ,
        const double &p_lMax, const double &p_lStep, const
        std::vector< std::array< double, 2> > &
        p_extrem):
13     m_alpha(p_alpha), m_m(p_m), m_sig(p_sig), m_PI(p_PI), m_cBar(p_cBar), m_s(p_s), m_dt(
        p_dt),
14     m_lMax(p_lMax), m_lStep(p_lStep), m_extrem(p_extrem) {}
15
16 Array< bool, Dynamic, 1> OptimizeSLEmissive::getDimensionToSplit() const
17 {
18     Array< bool, Dynamic, 1> bDim = Array< bool, Dynamic, 1>::Constant(3, true);
19     return bDim ;
20 }
21
22
23 // for parallelism
24 vector< std::array< double, 2> > OptimizeSLEmissive::getCone(const vector< std::array<
    double, 2> > &p_xInit) const
25 {
26     vector< std::array< double, 2> > xReached(3);
27     xReached[0][0] = p_xInit[0][0] + m_alpha * (m_m - m_extrem[0][1]) * m_dt - m_sig *
        sqrt(m_dt); // demand "cone" driven by maximal value allowed for demand
28     xReached[0][1] = p_xInit[0][1] + m_alpha * m_m * m_dt + m_sig * sqrt(m_dt) ; // low
        value for demand is taken equal to 0
29     xReached[1][0] = p_xInit[1][0] ; // Q only increases
30     xReached[1][1] = p_xInit[1][1] + m_extrem[0][1] * m_dt ; // Q increase bounded by
        maximal demand
31     xReached[2][0] = p_xInit[2][0] ; // L only increases

```

```

32     xReached[2][1] = p_xInit[2][1] + m_lMax * m_dt ;// maximal increase given by the
        control
33     return xReached;
34 }
35
36
37 // one step in optimization from current point
38 std::pair< ArrayXd, ArrayXd> OptimizeSLEmissive::stepOptimize(const ArrayXd &p_point,
39     const vector< shared_ptr<SemiLagrangEspCond> > &p_semiLag,
40     const double &, const ArrayXd &) const
41 {
42     pair< ArrayXd, ArrayXd> solutionAndControl;
43     solutionAndControl.first.resize(2);
44     solutionAndControl.second.resize(1);
45     ArrayXXd sig = ArrayXXd::Zero(3, 1) ;
46     sig(0, 0) = m_sig;
47     double vOpt = - StOpt::infy;
48     double yOpt = 0. ;
49     double lOpt = 0 ;
50     ArrayXd b(3);
51     b(0) = m_alpha * (m_m - p_point(0)) ; // trend
52     b(1) = max(p_point(0) - p_point(2), 0.);
53     // gain already possible to calculate (production and subvention)
54     double gainFirst = m_PI(p_point(0), p_point(2)) + m_s * pow(p_point(2), 1. - m_alpha)
55     ;
56     for (int iAl = 0; iAl < m_lMax / m_lStep ; ++iAl) // test all command for investment
        between 0 and lMax
57     {
58         double l = iAl * m_lStep;
59         b(2) = l ;
60         pair<double, bool> lagrangY = p_semiLag[1]->oneStep(p_point, b, sig, m_dt); // for
            the control calculate y
61         if (lagrangY.second) // is the control admissible
62         {
63             pair<double, bool> lagrang = p_semiLag[0]->oneStep(p_point, b, sig, m_dt); //
                one step for v
64             // gain function
65             double gain = m_dt * (gainFirst - lagrangY.first * b(1) - m_cBar(l, p_point(2)
66             ));
67             double arbitrage = gain + lagrang.first;
68             if (arbitrage > vOpt) // optimality of the control
69             {
70                 vOpt = arbitrage; // upgrade solution v
71                 yOpt = lagrangY.first; // store y
72                 lOpt = l; // upgrade optimal control
73             }
74         }
75     }
76     if (StOpt::almostEqual(vOpt, - StOpt::infy, 10))
77     {
78         std::cout << " Reduce time step " << std::endl ;
79         abort();
80     }
81     solutionAndControl.first(0) = vOpt; // send back v function
82     solutionAndControl.first(1) = yOpt; // send back y function
83     solutionAndControl.second(0) = lOpt; // send back optimal control
84     return solutionAndControl;
85 }
86 // one step in simulation for current simulation
87 void OptimizeSLEmissive::stepSimulate(const SpaceGrid &p_gridNext,
88     const std::vector< std::shared_ptr< StOpt::
        SemiLagrangEspCond> > &p_semiLag,
89     Ref<ArrayXd> p_state, int &,
90     const ArrayXd &p_gaussian,
91     const ArrayXd &,
92     Ref<ArrayXd> p_phiInOut) const
93 {

```

```

94   ArrayXd state = p_state;
95   ArrayXXd sig = ArrayXXd::Zero(3, 1) ; // diffusion matrix for semi Lagrangian
96   sig(0, 0) = m_sig;
97   double vOpt = - StOpt::infty;
98   double lOpt = 0 ;
99   double yOpt = 0;
100  ArrayXd b(3);
101  b(0) = m_alpha * (m_m - p_state(0)) ; // trend for D (independent of control)
102  b(1) = max(p_state(0) - p_state(2), 0.); // trend for Q (independent of control)
103  double gainFirst = m_PI(p_state(0), p_state(2)) + m_s * pow(p_state(2), 1. - m_alpha)
104  ; // gain for production and subvention
105  for (int iA1 = 0; iA1 < m_lMax / m_lStep ; ++iA1) // recalculate the optimal control
106  {
107      double l = iA1 * m_lStep;
108      b(2) = l ;
109      pair<double, bool> lagrangY = p_semiLag[1]->oneStep(p_state, b, sig, m_dt); //
110      // calculate y for this control
111      if (lagrangY.second)
112      {
113          pair<double, bool> lagrang = p_semiLag[0]->oneStep(p_state, b, sig, m_dt); //
114          // calculate the function value v
115          // gain function
116          double gain = m_dt * (gainFirst - lagrangY.first * b(1) - m_cBar(l, p_state(2)
117          ));
118          double arbitrage = gain + lagrang.first;
119          if (arbitrage > vOpt) // arbitrage
120          {
121              vOpt = arbitrage; // upgrade solution
122              yOpt = lagrangY.first; // upgrade y value
123              lOpt = l; // upgrade optimal control
124          }
125      }
126  }
127  // gain function
128  p_phiInOut(0) += m_dt * (gainFirst - yOpt * b(1) - m_cBar(lOpt, state(2))); // store v
129  // value
130  p_phiInOut(1) = yOpt; // store y value
131  // update state
132  state(0) += m_alpha * (m_m - p_state(0)) * m_dt + m_sig * p_gaussian(0) * sqrt(m_dt);
133  // demand (no control)
134  state(1) += b(1) * m_dt; //Q
135  state(2) += lOpt * m_dt; //L
136  // truncate if necessary to stay inside domain.
137  p_gridNext.truncatePoint(state);
138  p_state = state ;
139 }

```

The three dimensional grids used can be some full grids or some sparse grids. Both versions of the resolution can be found in a test case (testSLNonEmissive.cpp).

Part VI

Stochastic Dual Dynamic
Programming

Chapter 1

SDDP algorithm

1.1 Some general points about SDDP

Stochastic Dual Dynamic Programming (SDDP) is an approximate dynamic programming algorithm developed by Pereira and Pinto in 1991 [38].

To describe the operation of SDDP, we will consider a class of linear programs which have steps $T + 1$ stages denoted $\{0, 1, \dots, t, \dots, T\}$. We limit our class of problems to linear programs with a relatively complete recourse: the feasible region of the linear program at each step is nonempty and bounded.

Let us formalize the variables and constraints used in the SDDP problem.

Notations used

The notations described here are used in the general case.

- x_t is the decision variable at time t . If the data process is step-independent, x_t also designates the state at time $t + 1$.
- $\omega_t \in \Omega_t$ is the random data process at time t , where Ω_t is the random data set.
- c_t is the vector of cost at time t .
- A_t and E_t denote matrices of constraints.
- $Q_t(x_{t-1}, \omega_t)$ is the expected value of the problem at time t , knowing the state x_{t-1} and the random data ω_t .
- $\mathcal{Q}_t(x_{t-1}) = \mathbb{E}[Q_t(x_{t-1}, \omega_t)]$.

Decision process

The random data process ω_t is discovered gradually. Thus from an initial state x_0 , the state variables $(x_t)_{t \in \{0, 1, \dots, T\}}$ are determined in a non-anticipatory manner. The scheme is as follows:

$x_0 \rightarrow$ observation of $\omega_1 \rightarrow$ decision of $x_1 \dots$
 $\rightarrow$ decision of $x_{T-1} \rightarrow$ observation of $\omega_T \rightarrow$ decision of x_T

A rigorous formulation of the multi-step stochastic linear program to solve is as follows:

$$V^* = \min_{\substack{A_0 x_0 = \omega_0 \\ x_0 \geq 0}} c_0^\top x_0 + \mathbb{E} \left[\min_{\substack{E_1 x_0 + A_1 x_1 = \omega_1 \\ x_1 \geq 0}} c_1^\top x_1 + \mathbb{E} \left[\dots + \mathbb{E} \left[\min_{\substack{E_T x_{T-1} + A_T x_T = \omega_T \\ x_T \geq 0}} c_T^\top x_T \right] \right] \right] \quad (1.1)$$

The deterministic equivalent of this problem (1.1) is obtained by discretizing ω_t (or by directly using ω_t if discrete). The number of variables in this problem increases exponentially with the number of steps. It cannot be solved directly even if T or $(\Omega_t)_{t \in \{0,1,\dots,T\}}$ are of reasonable size.

Dynamic programming principle

Dynamic programming consists of dividing the problem (1.1) into a series of subproblems delimited by a state variable. The goal is to calculate backwards the functions Q_t and $\mathcal{Q}_t$. They fulfill the following equations:

$$[LP_t] \begin{cases} Q_t(x_{t-1}, \omega_t) = \min c_t^\top x_t + \mathcal{Q}_{t+1}(x_t) \\ A_t x_t = \omega_t - E_t x_{t-1}, \quad [\pi_t(\omega_t)] \\ x_t \geq 0 \end{cases} \quad (1.2)$$

$$\mathcal{Q}_t(x_{t-1}) = \mathbb{E}[Q_t(x_{t-1}, \omega_t)] \quad (1.3)$$

The function $Q(x_{t-1}, \omega_t)$ represents the expected value of a future cost knowing the state x_{t-1} and the random data ω_t . $\mathcal{Q}_t(x_{t-1})$ is the expected value of the future cost knowing the state x_{t-1} . The dynamic programming principle guarantees that $V^* = \mathcal{Q}_1(x_0)$.

Given $\mathcal{Q}_T(\cdot)$, the successive computations are achieved backwards switching between the resolution of the linear sub-problem (1.2) and the computation of (1.3).

The implementation of dynamic programming consists in successively approximating the two value functions by equations (1.2 - 1.3) by discretizing the state space and by solving the linear subproblems. The number of discretization points increases exponentially with the dimension of the state vector and becomes enormous for our applications (“curse of dimensionality”). Additionally, a linear approximation of $\mathcal{Q}_{t+1}(x_t)$ must be available to convert the transition problem to LP.

SDDP algorithm

SDDP is a method used to solve a multistep stochastic problems described in [38]. SDDP is based on Benders decomposition described in [5]. Please note that SDDP was developed in order to solve hydro-thermal scheduling problem.

SDDP limits the curse of dimensionality by avoiding *a priori* a complete discretization of the state space. Each SDDP iteration is a two-step process. The first step consists in generating a sequence of realistic states x_t^* from which in the second step the value

functions are estimated in their neighborhood. By repeating these two steps successively, the approximation of the value function becomes more and more precise. SDDP is also composed of two passes calculated alternately:

- one pass back: the objective is to improve the number of cut Benders in the vicinity of well-chosen candidate states. It also provides a lower limit of the optimal cost.
- a forward pass: the goal is to provide a set of new candidate states. An estimate of the upper bound of the optimal cost is also calculated.

On the other hand, the SDDP method is based on the form of the future value function $\mathcal{Q}_t(x_{t-1})$. Indeed within the framework of a linear problem with complete recourse, the function value is convex and linear by pieces. It can therefore be approached by taking the supremum of a family of minor affine functions. These affine functions are called *optimality cuts* or *Benders cuts*.

1.2 A method, different algorithms

The method implemented in this library is based on the different situations presented in a technical report of PSR program [37] where three different cases of the basic problem are solved by SDDP. All three cases are implemented in the library. Other cases could be added to those that exist in the future.

Notations

These notations will be used to present the different algorithms of SDDP.

- $\bar{z}$ designates the optimal cost obtained in forward pass.
- $\underline{z}$ denotes the optimal cost obtained in return.
- β_t^j denotes the slope of the j^{th} Benders cut.
- α_t^j denotes the intercept of the j^{th} Benders cut.

1.2.1 The basic case

To describe this case, the notations above are used. We focus on multistep stochastic problems with the following properties.

- Random quantities at different stages are independent.
- Random quantities at time t are summarized in ω_t .
- At each step, the linear subproblem solution space is nonempty and bounded.

In this case, the functions $\mathcal{Q}_t(\cdot)$ are convex. The primal and dual solutions of the linear problem exist and define optimal cuts. We can now describe precisely how the implemented algorithm works.

Initialization

The following values are fixed:

- $\{0, 1, \dots, T\}$, the set of steps, where T is the time horizon.
- $n = 0$, is the counter of the number of iterations (*backward-forward*). n is incremented at the end of each iteration.
- $p \in \mathbb{R}$, the precision to be reached for the convergence test.
- $n_{step} \in \mathbb{N}$, the number of iterations performed between 2 convergence tests.
- $n_{iterMax} \in \mathbb{N}$, the maximum number of iterations.
- $x_0 \in \mathbb{R}_+^n$, the initial state of the vector.
- $L \in \mathbb{N}$, the number of scenarios used in the backward pass.
- $G \in \mathbb{N}$, the number of scenarios used in the forward pass. It also gives the number of new cuts calculated at each iteration (*backward-forward*) and the number of states near which the Benders cuts are calculated.

Forward pass

The purpose of this pass is to explore new feasible vector state and to obtain an estimate of the upper limit of the optimal cost. To this end, the current strategy is simulated for a set of G scenarios. The set of scenarios can be historical chronicles or raffles. The 13 algorithm presents the forward pass at the n -th iteration of the SDDP method.

Backward pass

The purpose of the backward pass is to add, at each stage, a set of new Benders cuts and to provide a new lower bound estimate of the optimal operational cost. For that, we have a set of scenarios of the random quantities (the dimension of the set is L) recorded during the initialization. At each time step cuts G are added using the states G visited $(x_t^g)_{g=1, \dots, G}$ obtained during the passage forward. The 14 algorithm presents the backward pass.

Stopping test

In the literature about SDDP lots of stopping criterions were used and their efficiency has been proved. However a criterion is suitable for each particular problem. Thus it is tough to bring out one which is generic. Due to genericity requirements, two classical criterions are implemented in the library. These can be customized by the user. The first one defines a maximal number of iterations $n_{iterMax}$ (an iteration is made of the succession of *backward-forward* passes) which shall not be exceeded. The second one is a test of convergence towards each other between the forward and the backward cost. The convergence test uses the following indicator:

$$\psi_{n_{step}i} = \left| \frac{\bar{z}_{n_{step}i} - \underline{z}_{n_{step}i}}{\bar{z}_{n_{step}i}} \right|, \quad \text{with } i \in \mathbb{N} \quad (1.10)$$

This is calculated every n_{step} iterations. If it is less than a p threshold, the process stops, otherwise it continues. The threshold is set by the user.

Algorithm 13 Run of forward pass (n^{th} iteration)

- 1: Simulate defines $\{(\omega_t^g), t \in \{1, \dots, T\}\}$ of scenarios also distributed: for $g \in \Omega^g = \{1, \dots, G\}$.
- 2: **for** $g \in \Omega^g$ **do**
- 3: Solve the following linear subproblem.

$$[AP_0^n] \begin{cases} Q_0 = \min_{x_0, \theta_1} c_0^\top x_0 + \theta_1 \\ u.c. \quad A_0 x_0 = \omega_0, \quad [\pi_0(\omega_0)] \\ x_0 \geq 0 \\ \theta_1 + (\beta_1^j)^\top x_0 \geq \alpha_1^j, \quad j \in \{1, \dots, G, \dots, nG\} \end{cases} \quad (1.4)$$

- 4: Store the primal solution (x_0^g) of the problem $[AP_{0,g}^n]$.
- 5: **for** $t \in \{1, \dots, T\}$ **do**
- 6: Solve the following linear subproblem.

$$[AP_{t,g}^n] \begin{cases} Q_t^g(x_{t-1}^g, \omega_t^g) = \min_{x_t, \theta_{t+1}} c_t^\top x_t + \theta_{t+1} \\ s.c. \quad A_t x_t = \omega_t^g - E_t x_{t-1}^g, \quad [\pi_t(\omega_t^g)] \\ x_t \geq 0 \\ \theta_{t+1} + (\beta_{t+1}^j)^\top x_t \geq \alpha_{t+1}^j, \quad j \in \{1, \dots, G, \dots, nG\} \end{cases} \quad (1.5)$$

- 7: Store the primal solution (x_t^g) of the problem $[AP_{t,g}^n]$.
 - 8: **end for**
 - 9: Calculate the cost of the scenario g , at iteration n : $\bar{z}_n^g = \sum_{t=0}^T c_t x_t^g$.
 - 10: **end for**
 - 11: Calculate the total cost of the forward pass at iteration n : $\bar{z}_n = \frac{1}{G} \sum_{g=1}^G \bar{z}_n^g$.
-

1.2.2 Dependence of random quantities

In the previous case, we limit our problem to independent random quantities in the different stages. The resolution of the SDDP was obtained on the state vector x_t in the basic case.

But sometimes, in real life, the random quantities can be temporarily correlated. In a hydraulic problem, for example, there is a time dependence of the results. Time dependencies may also exist in the request. However, with time-bound random quantities, Bellman's recurrence formula (1.2 - 1.3) does not hold and classical SDDP can not be applied.

However if the Bellman functions are convex with respect to the temporal random quantities, it suffices to increase the dimension of the state vector by the dimension of the temporal random quantities to find ourselves in the configuration of the base case. In this case, the resolution of a linear program of reasonable size for each random draw is sufficient to calculate new Benders cuts in the neighborhood of a candidate state.

There are a few options for representing the time dependence of random quantities. However, in order to not increase the size of the problem too much, an ARMA process of order 1 is often chosen. In the random data vector ω_t two different parts must be distinguished from now on:

Algorithm 14 Run of backward pass

- 1: **for** $t = T, T - 1, \dots, 1$ **do**
- 2: **for** $x_{t-1}^g, g \in \{1, \dots, G\}$ **do**
- 3: **for** $\omega_t^l, l \in \{1, \dots, L\}$ **do**
- 4: Solve the following linear subproblem.

$$[AP_{t,l}^{n,g}] \left\{ \begin{array}{ll} Q_t^l(x_{t-1}^g, \omega_t^l) = \min_{x_t, \theta_{t+1}} c_t^\top x_t + \theta_{t+1} \\ s.c. \quad A_t x_t = \omega_t^l - E_t x_{t-1}^g, \quad [\pi_t(\omega_t^l)] \\ x_t \geq 0 \\ \theta_{t+1} + (\beta_{t+1}^j)^\top x_t \geq \alpha_{t+1}^j, \quad j \in \{1, \dots, G, \dots, (n+1)G\} \end{array} \right. \quad (1.6)$$

- 5: Store the dual solution $\pi_t(\omega_t^l)$ and the primal solution $Q_t^l(x_{t-1}^g, \omega_t^l)$ of the linear subproblem $[AP_{t,l}^{n,g}]$.
- 6: Calculate the cutoff that goes with the l^{th} hazard draw:

$$\left\{ \begin{array}{l} \alpha_{t,l}^g = Q_t^l(x_{t-1}^g, \omega_t^l) + \pi_t(\omega_t^l)^\top E_t x_{t-1}^g \\ \beta_{t,l}^g = E_t^\top \pi_t(\omega_t^l) \end{array} \right. \quad (1.7)$$

- 7: **end for**
- 8: Calculate the g^{th} new Benders cut at time t at iteration n . It is defined as the average value of the cuts obtained before:

$$\left\{ \begin{array}{l} \alpha_t^k = \frac{1}{L} \sum_{l=1}^L \alpha_{t,l}^g \\ \beta_t^k = \frac{1}{L} \sum_{l=1}^L \beta_{t,l}^g \end{array} \right. \quad \text{where } k = nG + g \quad (1.8)$$

- 9: **end for**
- 10: **end for**
- 11: Solve the following linear subproblem:

$$[AP_0^n] \left\{ \begin{array}{ll} Q_0 = \min_{x_0, \theta_1} c_0^\top x_0 + \theta_1 \\ s.c. \quad A_0 x_0 = \omega_0, \quad [\pi_0(\omega_0)] \\ x_0 \geq 0 \\ \theta_1 + (\beta_1^j)^\top x_0 \geq \alpha_1^j, \quad j \in \{1, \dots, G, \dots, (n+1)G\} \end{array} \right. \quad (1.9)$$

- 12: Save the cost *backward* $z_n = Q_0$.
-

- ω_t^{ind} is the vector of random data corresponding to independent random quantities.
- ω_t^{dep} is the vector of random data corresponding to the time-related random quantities.

And ω_t^{dep} fulfills the following recurrence equation:

$$\frac{\omega_t^{\text{dep}} - \mu_{\omega,t}}{\sigma_{\omega,t}} = \psi_1 \frac{\omega_{t-1}^{\text{dep}} - \mu_{\omega,t-1}}{\sigma_{\omega,t-1}} + \psi_2 \epsilon_t \quad (1.11)$$

To apply Bellman's recurrence formula, the state of the vector must consist of the decision variable x_t and the time-bound random quantities ω_t^{dep} . The dimension of the vector state is then increased. $x_t^{\text{dep}} = (x_t, \omega_t^{\text{dep}})^\top$ designates the new state vector. The Bellman function now satisfies the following two-step linear problem at time t :

$$[LP'_t] \begin{cases} Q_t(x_{t-1}, \omega_{t-1}^{\text{dep}}, \omega_t) = \min c_t^\top x_t + \mathcal{Q}_{t+1}(x_t, \omega_t^{\text{dep}}) \\ u.c. \quad A_t x_t = P \omega_t^{\text{dep}} - E_t x_{t-1}, \quad [\pi_t(\omega_t)] \\ x_t \geq 0 \end{cases} \quad (1.12)$$

with P the matrix such that $\omega_t = P \omega_t^{\text{dep}}$.

The variable ω_t^{dep} is a random process. Thus, the above problem is solved by using specific ω_t^l values of this variable. To get them, we apply a Markov process, i.e. we simulate different values of white noise ϵ_t^l .

The new shape of the state vector involves changes in the sensitivity of the Bellman function. It is therefore a function depending on the decision variable x_t but also on the vector of random quantity linked to time ω_t^{dep} . The calculation of the Benders cuts is then a little different:

$$\begin{aligned} \frac{\partial Q_t(x_{t-1}, \omega_{t-1}^{\text{dep}}, \omega_t)}{\partial \omega_{t-1}^{\text{dep}}} &= \frac{\partial Q_t(x_{t-1}, \omega_{t-1}^{\text{dep}}, \omega_t)}{\partial \omega_t^{\text{dep}}} \frac{\partial \omega_t^{\text{dep}}}{\partial \omega_{t-1}^{\text{dep}}} \\ &= \pi_t(\omega_t)^\top P \psi_1 \frac{\sigma_{\omega,t}}{\sigma_{\omega,t-1}}, \end{aligned} \quad (1.13)$$

The backward passage should be changed as shown in the 15 algorithm. Some new calculation steps must be taken into account.

1.2.3 Non-convexity and conditional cuts

Some random quantities can introduce non-convexity preventing us from applying the classical SDDP algorithm. Indeed when the random quantities appear on the left side of the linear constraints or in the cost function (typically A_t and/or c_t become random) the property of convexity of the Bellman functions with respect to the random quantities is not no longer observed.

In the context of a production management problems, the situation often arise. For example, sometimes the unit running cost of plants is random. It is also observed when we deal with the uncertainty of spot prices for use in stochastic mid-term scheduling.

In a technical report, Pereira, Campodonico, and Kelman [37] suggested a new algorithm to efficiently approximate Bellman functions by explicitly using the dependence of Bellman functions on these random quantities. This new algorithm is based on a combination of

Algorithm 15 Execution of the backward pass with time-bound random quantities (AR1 process)

- 1: Take the following set of pairs: $\{x_t^g, \omega_t^{\text{dep},g}\}$ for $g \in \{1, \dots, G\}$, $t \in \{1, \dots, T\}$.
- 2: **for** $t = T, T-1, \dots, 1$ **do**
- 3: **for** $(x_{t-1}^g, \omega_{t-1}^{\text{dep},g})$, $g \in \{1, \dots, G\}$ **do**
- 4: **for** $l \in \{1, \dots, L\}$ **do**
- 5: Produce a value for white noise ϵ_t^l .
- 6: calculate the element $\hat{\omega}_t^l$ knowing the previous random quantity $\omega_{t-1}^{\text{dep},g}$:

$$\hat{\omega}_t^l = \sigma_{\omega,t} \left(\psi_1 \frac{\omega_{t-1}^{\text{dep},g} - \mu_{\omega,t-1}}{\sigma_{\omega,t-1}} + \psi_2 \epsilon_t^l \right) + \mu_{\omega,t} \quad (1.14)$$

- 7: Solve the following linear subproblem.

$$[AP'_{t,l}] \begin{cases} Q_t^l(x_{t-1}^g, \omega_{t-1}^{\text{dep},g}, \hat{\omega}_t^l) = \min_{x_t, \theta_{t+1}} c_t^\top x_t + \theta_{t+1} \\ u.c. \quad A_t x_t = P \hat{\omega}_t^l - E_t x_{t-1}^g, \quad [\pi_t(\hat{\omega}_t^l)] \\ x_t \geq 0 \\ \theta_{t+1} + (\beta_{t+1}^j)^\top x_t + (\gamma_{t+1}^j)^\top \hat{\omega}_t^l \geq \alpha_{t+1}^j, \quad j \in \{1, \dots, G, \dots, (n+1)G\} \end{cases} \quad (1.15)$$

- 8: Store the dual solution $\pi_t(\hat{\omega}_t^l)$ and the primal solution $Q_t^l(x_{t-1}^g, \omega_{t-1}^{\text{dep},g}, \hat{\omega}_t^l)$ of the primal problem $[AP'_{t,l}]$.
- 9: Calculate the cutoff that goes with the draw l^{th} :

$$\begin{cases} \alpha_{t,l}^g = Q_t^l(x_{t-1}^g, \omega_{t-1}^{\text{dep},g}, \hat{\omega}_t^l) + \pi_t(\hat{\omega}_t^l)^\top \left(E_t x_{t-1}^g - \psi_1 \frac{\sigma_{\omega,t}}{\sigma_{\omega,t-1}} P \omega_{t-1}^{\text{dep},g} \right) \\ \beta_{t,l}^g = E_t^\top \pi_t(\hat{\omega}_t^l) \\ \gamma_{t,l}^g = \psi_1 \frac{\sigma_{\omega,t}}{\sigma_{\omega,t-1}} P^\top \pi_t(\hat{\omega}_t^l) \end{cases} \quad (1.16)$$

- 10: **end for**
- 11: Calculate the g^{th} new Benders cut at time t at iteration n defined as the average value of the cuts obtained before for $k = nG + g$:

$$\alpha_t^k = \frac{1}{L} \sum_{l=1}^L \alpha_{t,l}^g, \quad \beta_t^k = \frac{1}{L} \sum_{l=1}^L \beta_{t,l}^g, \quad \gamma_t^k = \frac{1}{L} \sum_{l=1}^L \gamma_{t,l}^g$$

- 12: **end for**
- 13: **end for**
- 14: Solve the following linear subproblem.

$$[AP'_0] \begin{cases} Q_0 = \min_{x_0, \theta_1} c_0^\top x_0 + \theta_1 \\ u.c. \quad A_0 x_0 = \omega_0, \quad [\pi_0(\omega_0)] \\ x_0 \geq 0 \\ \theta_1 + (\beta_1^j)^\top x_0 + (\gamma_1^j)^\top \omega_0^{\text{dep}} \geq \alpha_1^j, \quad j \in \{1, \dots, G, \dots, (n+1)G\} \end{cases} \quad (1.17)$$

- 15: Save the back cost $\underline{z}_n = Q_0$.
-

SDDP and ordinary stochastic dynamic programming. The SDP part deals with the non-convex random quantities, while the other random quantities are dealt with in the SDDP part. It is an extension of the classic SDDP algorithm. It is described in detail in [37] and in [18].

In the library, we propose two approaches to deal with this non convexity:

A tree approach

In [18], the spot price p_t is considered a state. The set of achievable spot prices is discretized into a set of M points $\zeta_1, \dots, \zeta_M$. The following Markov model is then used:

$$\mathbb{P}(p_t = \zeta_j | p_{t-1} = \zeta_i) = \rho_{ij}(t),$$

This model makes easier the implementation of the SDP. But it implies discretization mistakes that are hard to quantify. It is also tough to discretize with efficiency a random process when only a small number of scenarios is available.

However when the dimension of the non convex uncertainties is low, it is an approach to consider. The finite number of states, and the probabilities linking states between two successive time step leads to a tree representation of the uncertainties. As the approach is classical, we don't detail this version of the algorithm. We only detail the second approach which is in fact in spirit very similar.

A regression based approach

In this case the modeling used in the library is somewhat different from that described in the two articles. In order to avoid the discretization with trees in this second approach, the evolution of non-convex random quantities is decided by Monte Carlo simulations. At each stage, a fixed number of Monte-Carlo simulations is provided. Anyway, despite this difference, the overall view of this brand new algorithm is similar to that described in two articles:

- The non-convex random quantities depend on the realization of the previous one according to a mathematical model (Markov chain).
- At each step, the Bellman functions are approximated by the conditional realization of these random quantities.
- We have used conditional cuts to give an estimate of the Bellman functions. These conditional cuts are calculated using the methods of the 2 section: two methods are available in the library. Both use adaptive support. The first uses a constant approximation per cell while the second uses a linear approximation per cell.

In our algorithm the characteristics of the conditional cuts are revealed thanks to a conditional expectation calculation.

However, conditional expectation calculations are not easy when the exact distribution of the random variable is not known. Some techniques exist but in the library a specific one is used and described above in chapter 2: it is based on local linear regression.

Regression, dynamic stochastic programming and SDDP

The execution of the backward pass in the new algorithm combining SDDP and SDP using local linear regression is described below.

Before describing this algorithm in detail, let's introduce some notations:

- S is the space of the non-convex random quantities.
- d is the dimension of the space of non-convex random quantities S
- At each step, U Monte Carlo simulations in S are provided. We thus obtain scenarios U denoted by s_t^u at each step t
- $\tilde{I}$ is a partition of the space of non-convex random quantities S .

$$\tilde{I} = \{\underline{I} = (i_1, \dots, i_d), i_1 \in \{1, \dots, I_1\}, \dots, i_d \in \{1, \dots, I_d\}\}$$

- $\{D_{\underline{I}}\}_{\underline{I} \in \tilde{I}}$ is the set of meshes of the set of scenarios.
- $M_M = \prod_{k=1}^d I_k$ designate the number of meshes at each step.

The backward step with both random time-related and non-convex quantities is presented in the algorithm 16.

Algorithm 16 Run of the backward pass with time-related (AR1) and non-convex random quantities

- 1: Pick up the set of the following pairs: $\{x_t^g, \omega_t^{\text{dep},g}\}$ for $g \in \{1, \dots, G\}$, $t \in \{1, \dots, T\}$.
- 2: **for** $t = T, T-1, \dots, 1$ **do**
- 3: Generate values for the non-convex random quantities at time t knowing the scenarios at time $t-1$: s_t^u , $u \in \{1, \dots, U\}$.
- 4: **for** $(x_{t-1}^g, \omega_{t-1}^{\text{dep},g})$, $g \in \{1, \dots, G\}$ **do**
- 5: **for** $u \in \{1, \dots, U\}$ **do**
- 6: Consider a scenario s_t^u in the mesh D_I .
- 7: **for** $l \in \{1, \dots, L\}$ **do**
- 8: Produce a value for the white noise ϵ_t^l .
- 9: Compute the element $\hat{\omega}_t^l$ knowing the previous random quantity $\omega_{t-1}^{\text{dep},g}$:

$$\hat{\omega}_t^l = \sigma_{\omega,t} \left(\psi_1 \frac{\omega_{t-1}^{\text{dep},g} - \mu_{\omega,t-1}}{\sigma_{\omega,t-1}} + \psi_2 \epsilon_t^l \right) + \mu_{\omega,t} \quad (1.18)$$

- 10: Pick up the cuts corresponding the mesh D_I : $\{\alpha_{t+1}^{I,j}(s), \beta_{t+1}^{I,j}(s), \gamma_{t+1}^{I,j}(s)\}$, $j \in \{1, \dots, (n+1)G\}$.
- 11: Solve the following linear sub-problem:

$$[AP'_{t,l}{}^{n,g}] \begin{cases} Q_t^l(x_{t-1}^g, \omega_{t-1}^{\text{dep},g}, \hat{\omega}_t^l, s_t^u) = \min_{x_t, \theta_{t+1}} c_t(s_t^u)^\top x_t + \theta_{t+1}(s_t^u) \\ s.c. \quad A_t(s_t^u)x_t = P\hat{\omega}_t^l - E_t x_{t-1}^g, \quad [\pi_t(\hat{\omega}_t^l, s_t^u)] \\ x_t \geq 0 \\ \theta_{t+1}(s_t^u) + (\beta_{t+1}^{I,j}(s_t^u))^\top x_t + (\gamma_{t+1}^{I,j}(s_t^u))^\top \hat{\omega}_t^l \geq \alpha_{t+1}^{I,j}(s_t^u), \\ j \in \{1, \dots, G, \dots, nG\} \end{cases} \quad (1.19)$$

- 12: Store the dual solution $\pi_t(\hat{\omega}_t^l)$ and the primal solution $Q_t^l(x_{t-1}^g, \omega_{t-1}^{\text{dep},g}, \hat{\omega}_t^l, s_t^u)$ of the problem $[AP'_{t,l}{}^{n,g}]$.
- 13: Calculate the corresponding cut at the l^{th} draw of uncertainties:

$$\begin{aligned} \hat{\alpha}_{t,l}^{g,I}(s_t^u) &= Q_t^l(x_{t-1}^g, \omega_{t-1}^{\text{dep},g}, \hat{\omega}_t^l) + \pi_t(\hat{\omega}_t^l)^\top \left(E_t x_{t-1}^g - \psi_1 \frac{\sigma_{\omega,t}}{\sigma_{\omega,t-1}} P \omega_{t-1}^{\text{dep},g} \right) \\ \hat{\beta}_{t,l}^{g,I}(s_t^u) &= E_t^\top \pi_t(\hat{\omega}_t^l) \\ \hat{\gamma}_{t,l}^{g,I}(s_t^u) &= \psi_1 \frac{\sigma_{\omega,t}}{\sigma_{\omega,t-1}} P^\top \pi_t(\hat{\omega}_t^l) \end{aligned}$$

- 14: **end for**
-

15: Compute the cut for a non-convex random quantity s_t^u at time t at iteration n : it is defined as the weighted average on the L Benders cut obtained before:

$$\begin{aligned}\hat{\alpha}_t^{g,I}(s_t^u) &= \frac{1}{L} \sum_{l=1}^L \hat{\alpha}_{t,l}^{g,I}(s_t^u) \\ \hat{\beta}_t^{g,I}(s_t^u) &= \frac{1}{L} \sum_{l=1}^L \hat{\beta}_{t,l}^{g,I}(s_t^u), \quad j = nG + g \\ \hat{\gamma}_t^{g,I}(s_t^u) &= \frac{1}{L} \sum_{l=1}^L \hat{\gamma}_{t,l}^{g,I}(s_t^u)\end{aligned}$$

16: **end for**

17: **for** $\underline{I}_i, i \in \{1, \dots, M_M\}$ **do**

18: Compute the g^{th} new cut of the mesh $D_{\underline{I}_i}$ at time t at iteration n defined as the conditional expectation with respect to the scenario u at time t :

$$\left\{ \begin{array}{l} \alpha_t^{j,I}(s_{t-1}^u) = \mathbb{E} \left[\hat{\alpha}_t^{g,I}(s_t^u) | s_{t-1}^u \right], \\ \beta_t^{j,I}(s_{t-1}^u) = \mathbb{E} \left[\hat{\beta}_t^{g,I}(s_t^u) | s_{t-1}^u \right], \\ \gamma_t^{j,I}(s_{t-1}^u) = \mathbb{E} \left[\hat{\gamma}_t^{g,I}(s_t^u) | s_{t-1}^u \right] \end{array} \right., \quad j = nG + g \quad (1.20)$$

19: **end for**

20: **end for**

21: **end for**

22: Solve the initial linear sub problem $[AP_0'^n]$.

23: Save the backward cost $\underline{z}_n = Q_0$.

1.3 C++ API

The SDDP part of the stochastic library is in C++ code. This unit is a classic black box: specific inputs must be provided to obtain the expected results. In the SDDP unit, the backward and forward passes are performed successively until the stop criterion is reached. In this unit, the succession of passes is carried out by

- the `backwardForwardSDDPTree` class for the tree method,
- the `backwardForwardSDDP` class for the regression-based approach.

These classes takes three non-defined classes as input.

1.3.1 Inputs

The user must implement three classes.

- A class where the transition problem is described which is shown in the example `TransitionOptimizer`. This class is at the core of the problem resolution. Therefore, great flexibility is left to the user to implement this class. The class is used with both the tree approach and the regression-based approach.

In a way, this class is where the technical aspects of the problem are adjusted. This class describes backward and forward passes. Four methods must be implemented:

- `updateDates`: establish the new set of dates: $(t, t + dt)$.
- `oneStepForward`: solves the various linear transition problems during the forward pass for a particle, a random vector and an initial state:
 - * the state $(x_{t-dt}, w_t^{\text{dep}})$ is given as input to the function.
 - * the s_t values are restored by the simulator.
 - * the LP is solved between the dates t and $t + dt$ for the given s_t and the constraints due to w_t^{dep} (demand, flow constraints) and allows to get the optimum x_t .
 - * Using iid sampling, w_{t+dt}^{dep} is estimated.
 - * return $(x_t, w_{t+dt}^{\text{dep}})$ as next state and (x_t, w_t^{dep}) which to be used as a state to visit in the next backward resolution.
- `oneStepBackward`: solves different linear transition problems during the backward pass for a particle, a random vector and an initial state.
 - * The vector (x_t, w_t^{dep}) is given as input if $t \geq 0$; otherwise, the input is $(x_{-dt}, w_0^{\text{dep}})$.
 - * If $t \geq 0$, sample to calculate w_{t+dt}^{dep} to obtain the state $(x_t, w_{t+dt}^{\text{dep}})$ at the start of the LP resolution period. If $t < 0$, the state is $(x_{-dt}, w_0^{\text{dep}})$.
 - * Solve the LP from date t to next date $t + dt$ (if equally spaced periods) for the variable x_{t+dt} .

- * Return the value of the function and the dual that will be used for the cut estimates.
- oneAdmissibleState: returns an admissible state at time t (respect only the constraints).

TransitionOptimizer must derive from the OptimizerSDDPBase class defined below.

```

1 // Copyright (C) 2016 EDF
2 // All Rights Reserved
3 // This code is published under the GNU Lesser General Public License (GNU LGPL)
4 #ifndef OPTIMIZERSDDPBASE_H
5 #define OPTIMIZERSDDPBASE_H
6 #include <Eigen/Dense>
7 #include "StOpt/sddp/SDDPCutOptBase.h"
8 #include "StOpt/core/grids/OneDimRegularSpaceGrid.h"
9 #include "StOpt/core/grids/OneDimData.h"
10 #include "StOpt/sddp/SimulatorSDDPBase.h"
11 #include "StOpt/sddp/SimulatorSDDPBaseTree.h"
12
13
14 /** \file OptimizerSDDPBase.h
15  * \brief Define an abstract class for Stochastic Dual Dynamic Programming problems
16  * \author Xavier Warin
17  */
18
19 namespace StOpt
20 {
21
22 /// \class OptimizerSDDPBase OptimizerSDDPBase.h
23 /// Base class for optimizer for Dynamic Programming
24 class OptimizerSDDPBase
25 {
26
27
28 public :
29
30     OptimizerSDDPBase() {}
31
32     virtual ~OptimizerSDDPBase() {}
33
34
35     /// \brief Optimize the LP during backward resolution
36     /// \param p_linCut cuts used for the PL (Benders for the Bellman value at the end of
37     /// the time step)
38     /// \param p_aState store the state, and 0.0 values
39     /// \param p_particle the particle n dimensional value associated to the regression
40     /// \param p_isample sample number for independant uncertainties
41     /// \return a vector with the optimal value and the derivatives if the function value
42     /// with respect to each state
43     virtual Eigen::ArrayXd oneStepBackward(const StOpt::SDDPCutOptBase &p_linCut, const std
44     ::tuple< std::shared_ptr<Eigen::ArrayXd>, int, int > &p_aState, const Eigen::
45     ArrayXd &p_particle, const int &p_isample) const = 0;
46
47
48     /// \brief Optimize the LP during forward resolution
49     /// \param p_aParticle a particle in simulation part to get back cuts
50     /// \param p_linCut cuts used for the PL (Benders for the Bellman value at the end of
51     /// the time step)
52     /// \param p_state store the state, the particle number used in optimization and mesh
53     /// number associated to the particle. As an input it contains the current state
54     /// \param p_stateToStore for backward resolution we need to store \f$ (S_t, A_{t-1}, D_{
55     t-1}) \f$ where p_state in output is \f$ (S_t, A_t, D_t) \f$
56     /// \param p_isimu number of teh simulation used
57     virtual double oneStepForward(const Eigen::ArrayXd &p_aParticle, Eigen::ArrayXd &
58     p_state, Eigen::ArrayXd &p_stateToStore, const StOpt::SDDPCutOptBase &p_linCut,
59     const int &p_isimu) const = 0 ;
60

```

```

51
52
53     /// \brief update the optimizer for new date
54     ///     - In Backward mode, LP resolution achieved at date p_dateNext,
55     ///     starting with uncertainties given at date p_date and evolving to give
56     ///     uncertainty at date p_dateNext,
57     ///     - In Forward mode, LP resolution achieved at date p_date,
58     ///     and uncertainties evolve till date p_dateNext
59     virtual void updateDates(const double &p_date, const double &p_dateNext) = 0 ;
60
61     /// \brief Get an admissible state for a given date
62     /// \param p_date current date
63     /// \return an admissible state
64     virtual Eigen::ArrayXd oneAdmissibleState(const double &p_date) = 0 ;
65
66     /// \brief get back state size
67     virtual int getStateSize() const = 0;
68
69     /// \brief get the backward simulator back
70     virtual std::shared_ptr< StOpt::SimulatorSDDPBase > getSimulatorBackward() const = 0;
71
72     /// \brief get the forward simulator back
73     virtual std::shared_ptr< StOpt::SimulatorSDDPBase > getSimulatorForward() const = 0;
74
75 };
76 }
77 #endif /* OPTIMIZERSDDPBASE_H */

```

- A forward pass simulator: `SimulatorSim`
- A backward pass simulator: `SimulatorOpt`. This simulator can use an underlying process to generate scenarios, a set of historical chronicles, or a discrete set of scenarios. Often, in the case of a test carried out, a Boolean is enough to distinguish the forward and the backward simulator.

At the opposite of the class where the transition is described, the simulator is of course different for the tree approach and the regression approach because the first gives only a finite number of states.

An abstract class for simulators using the regression-based methods is defined below:

```

1 // Copyright (C) 2016 EDF
2 // All Rights Reserved
3 // This code is published under the GNU Lesser General Public License (GNU LGPL)
4 #ifndef SIMULATORSDDPBASE_H
5 #define SIMULATORSDDPBASE_H
6 #include <Eigen/Dense>
7
8 /* \file SimulatorBase.h
9  * \brief Abstract class for simulators for SDDP method
10  * \author Xavier Warin
11  */
12 namespace StOpt
13 {
14     /// \class SimulatorSDDPBase SimulatorSDDPBase.h
15     /// Abstract class for simulators used for SDDP
16     class SimulatorSDDPBase
17     {
18     public :
19
20         /// \brief Constructor
21         SimulatorSDDPBase() {}
22

```

```

23     /// \brief Destructor
24     virtual ~SimulatorSDDPBase() {}
25
26     /// \brief Get back the number of particles (used in regression part)
27     virtual int getNbSimul() const = 0;
28     /// \brief Get back the number of sample used (simulation at each time step , these
29     ///     simulations are independent of the state)
30     virtual int getNbSample() const = 0;
31     /// \brief Update the simulator for the date :
32     /// \param p_idateCurr index in date array
33     virtual void updateDateIndex(const int &p_idateCur) = 0;
34     /// \brief get one simulation
35     /// \param p_isim simulation number
36     /// \return the particle associated to p_isim
37     /// \brief get current Markov state
38     virtual Eigen::VectorXd getOneParticle(const int &p_isim) const = 0;
39     /// \brief get current Markov state
40     virtual Eigen::MatrixXd getParticles() const = 0;
41     /// \brief Reset the simulator (to use it again for another SDDP sweep)
42     virtual void resetTime() = 0;
43     /// \brief in simulation part of SDDP reset time and reinitialize uncertainties
44     /// \param p_nbSimul Number of simulations to update
45     virtual void updateSimulationNumberAndResetTime(const int &p_nbSimul) = 0;
46 };
47 #endif /* SIMULATORSDDPBASE_H */

```

An abstract class derived from the previous class for simulators using tree-based methods is defined below:

```

1 // Copyright (C) 2019 EDF
2 // All Rights Reserved
3 // This code is published under the GNU Lesser General Public License (GNU LGPL)
4 #ifndef SIMULATORSDDPBASETREE_H
5 #define SIMULATORSDDPBASETREE_H
6 #include <memory>
7 #include <Eigen/Dense>
8 #include "geners/BinaryFileArchive.hh"
9 #include "StOpt/core/utils/comparisonUtils.h"
10 #include "StOpt/dp/SimulatorDPBaseTree.h"
11 #include "StOpt/sddp/SimulatorSDDPBase.h"
12
13 /* \file SimulatorBaseTree.h
14 * \brief Base class for simulators for SDDP method with uncertainties breaking concavity/
15 *     convexity in a tree
16 * \author Xavier Warin
17 */
18 namespace StOpt
19 {
20     /// \class SimulatorSDDPBaseTree SimulatorSDDPBaseTree.h
21     /// Base class for simulators used for SDDP with uncertainties breaking concavity/
22     ///     convexity in a Tree
23     class SimulatorSDDPBaseTree : public SimulatorSDDPBase, public SimulatorDPBaseTree
24     {
25     public :
26
27         /// \brief Constructor
28         /// \param p_binForTree binary generators archive with structure
29         ///     - dates -> eigen array of dates, size ndate
30         ///     - nodes -> nDate array , each array containing nodes coordinates with
31         ///         size (ndim, nbNodes)
32         ///     - proba -> for a point i at a given date and a point j at next date ,
33         ///         prob(i,j) gives the probability to go from node i to node j.
34         SimulatorSDDPBaseTree(const std::shared_ptr<gs::BinaryFileArchive> &p_binForTree):
35             SimulatorDPBaseTree(p_binForTree) {}
36     };
37 }

```

```

34
35     /// \brief Destructor
36     virtual ~SimulatorSDDPBaseTree() {}
37
38     /// \brief
39     /// \brief Get back the number of particles
40     virtual int getNbSimul() const
41     {
42         return 0;
43     }
44
45     /// \brief Get back the number of sample used (simulation at each time step , these
46     /// simulations are independent of the state)
47     virtual int getNbSample() const
48     {
49         return 0 ;
50     }
51
52     /// \brief get one simulation
53     /// \param p_isim simulation number
54     /// \return the particle associated to p_isim
55     virtual Eigen::VectorXd getOneParticle(const int &p_isim) const
56     {
57         return m_nodesCurr.col(getNodeAssociatedToSim(p_isim));
58     }
59
60     /// \brief get current Markov state
61     virtual Eigen::MatrixXd getParticles() const
62     {
63         return Eigen::MatrixXd();
64     }
65
66     /// \brief Reset the simulator (to use it again for another SDDP sweep)
67     virtual void resetTime() {}
68
69     /// \brief in simulation part of SDDP reset time and reinitialize uncertainties
70     /// \param p_nbSimul Number of simulations to update
71     virtual void updateSimulationNumberAndResetTime(const int &p_nbSimul) {}
72
73     /// \brief Update the simulator for the date :
74     /// \param p_idateCurr index in date array
75     virtual void updateDateIndex(const int &p_idateCur)
76     {
77         load(p_idateCur);
78     }
79
80
81 };
82 }
83
84 #endif /* SIMULATORSDDPBASSETREE_H */

```

1.3.2 Architecture

We only detail the SDDP architecture for the regression based approach because the tree approach uses the same algorithm.

The SDDP management part of the library is built following the scheme described below.

In the following pseudo-code, you have to keep in mind that some small shortcuts were used in order to make reading friendly (e.g. linear subproblem in the initial case ($t = 0$) should be a little different from that of the other time steps, the entries **forwardSDDP**, **backwardSDDP**, **backwardForwardSDDP** have been omitted for simplicity). A more rigorous theoretical explanation is available in the previous part.

Remark 51 *To use the tree method,*

forwardSDDP, backwardSDDP, backwardforwardSDDP can be replaced with specialized version for trees called

forwardSDDPTree, backwardSDDPTree, backwardforwardSDDPTree in the library.

Three colors have been used: the blue parts correspond to the use of the functions implemented in the **TransitionOptimizer** class, the red parts correspond to the use of the **Simulator** (Sim or Opt) functions while the grey parts correspond to generic functions totally handled by the library. To be more precise, what you need to implement as a StOpt user is only the **TransitionOptimizer** and the **Simulator** (blue and red parts), other functions and loops described are already implemented and managed by the library.

Algorithm 17 Run of backwardforwardSDDP(), the main function)

- 1: Init: $x_t^g = \text{TransitionOptimizer.oneAdmissibleState}(t)$, for $g \in \{1, \dots, G\}$ and $t \in \{1, \dots, T - 1\}$, $n = 0$, $\psi = \infty$.
 - 2: **while** $\psi > \epsilon$ and $n < n_{max}$ **do**
 - 3: **StOpt**
 - 4: $V_b = \text{backwardSDDP}()$ Using the previously computed set $(x_t^g)_{t,g}$ and create a set of cuts.
 - 5: $V_f = \text{forwardSDDP}()$ Simulation using the cuts created in all the backward passes and update the set $(x_t^g)_{t,g}$.
 - 6:
$$\psi = \frac{V_f - V_b}{V_f}$$
 - 7:
$$n = n + 1$$
 - 8: **end while**
-

1.3.3 Implement your problem

In the next section, some tips and explanations will be given to help you implementing your problem in the library. It is advisable to consult the examples provided by the

Algorithm 18 Run of forwardSDDP (n^{th} iteration)

```
1: for  $g \in \Omega_g$  do
2:    $iStep = 0$ : index of the date stored in the simulator
3:   for  $t \in \{0, \dots, T\}$  do
4:     TransitionOptimizer.updateDates( $t, t + 1$ ): update the required data fol-
       lowing the current time step (iterator over current time step, average de-
       mand,...)
5:     SimulatorSim.updateDateIndex( $iStep$ ): give the random quantities ( $\omega_t^g$ )
       for the scenario  $g$  at time  $t$ 
6:     StOpt Read the previously computed files to gather  $\alpha_{t+1}^j, \beta_{t+1}^j$ , for  $j \in$ 
        $\{1, \dots, G, \dots, nG\}$ 
7:     TransitionOptimizer.oneStepForward():
       Solve the following linear sub-problem.
       
$$[AP_{t,g}^n] \begin{cases} Q_t^g(x_{t-1}^g, \omega_t^g) = \min_{x_t, \theta_{t+1}} c_t^\top x_t + \theta_{t+1} \\ s.c. \quad A_t x_t = \omega_t^g - E_t x_{t-1}^g, \quad [\pi_t(\omega_t^g)] \\ x_t \geq 0 \\ \theta_{t+1} + (\beta_{t+1}^j)^\top x_t \geq \alpha_{t+1}^j, \quad j \in \{1, \dots, G, \dots, nG\} \end{cases} \quad (1.21)$$

       Compute the cost for current time step  $c_t^\top x_t^g$ 
       Return: the primal solution ( $x_t^g$ ) of the problem
8:     StOpt Store the primal solution ( $x_t^g$ ) of the problem  $[AP_{t,g}^n]$ 
9:      $iStep = iStep + 1$ 
10:   end for
11:   StOpt Compute the cost for scenario  $g$ , at iteration  $n$ :  $\bar{z}_n^g = \sum_{t=0}^T c_t x_t^g$ 
12: end for
13: StOpt Compute the total cost in forward pass at iteration  $n$ :  $\bar{z}_n = \frac{1}{G} \sum_{g=1}^G \bar{z}_n^g$ 
```

library. This will give you a better understanding of what is needed to calculate the SDDP method via StOpt (`test/c++/tools/sddp` folder for the optimization examples, `test/c++/tools/simulators` for the one simulators, and `test/c++/functional` for the main instances).

Implement your own TransitionOptimizer class

As described above, your **TransitionOptimizer** class must be specific to your problem (it is given as an argument to the `backwardForwardSDDP` function). Therefore, you must implement it yourself under certain constraints in order to adapt it to the requirements of the library.

First, make sure your **TransitionOptimizer** class inherits from the `OptimizerSDDPBase` class. You will then need to implement the following functions.

Algorithm 19 Run of backwardSDDP

```
1:  $iStep = NbStep$ : update the simulator time index to give the uncertainty at  $T$ 
2: for  $t = T, T - 1, \dots, 0$  do
3:   StOpt Read the previously computed files to gather  $x_{t-1}^g$ , for  $g \in \{1, \dots, G\}$ 
4:   TransitionOptimizer.updateDates( $t-1, t$ ): update the required data following
   the current time step (iterator over current time step, average demand,...)
5:   SimulatorOpt.updateDateIndex( $iStep$ ): give the random quantities for the  $L$ 
   scenarios at time  $t$ 
6:   StOpt Read the previously computed files to gather  $\alpha_{t+1}^j, \beta_{t+1}^j$ , for  $j \in$ 
    $\{1, \dots, G, \dots, nG\}$ 
7:   for  $x_{t-1}^g, g \in \{1, \dots, G\}$  do
8:     for  $\omega_t^l, l \in \{1, \dots, L\}$  do
9:       TransitionOptimizer.oneStepBackward()
       Solve the following linear sub-problem.
       
$$[AP_{t,l}^{n,g}] \begin{cases} Q_t^l(x_{t-1}^g, \omega_t^l) = \min_{x_t, \theta_{t+1}} c_t^\top x_t + \theta_{t+1} \\ \text{s.t.} \quad A_t x_t = \omega_t^l - E_t x_{t-1}^g, \quad [\pi_t(\omega_t^l)] \\ x_t \geq 0 \\ \theta_{t+1} + (\beta_{t+1}^j)^\top x_t \geq \alpha_{t+1}^j, \quad j \in \{1, \dots, G, \dots, (n+1)G\} \end{cases} \quad (1.22)$$

       Return: the dual solution  $\pi_t(\omega_t^l)$  and the primal one  $Q_t^l(x_{t-1}^g, \omega_t^l)$  of the
       linear sub-problem  $[AP_{t,l}^{n,g}]$ 
10:       $iStep = iStep + 1$ ,
11:    end for
12:    StOpt Compute the  $g^{th}$  new Benders cut at time  $t$  at iteration  $n$ :  $\alpha_t^j, \beta_t^j$ , for
     $j \in \{(n-1)G, (n-1)G + 1, \dots, nG\}$ 
13:  end for
14: end for
15: StOpt Save the cost backward  $z_n = Q_0$ 
```

- The `updateDates` function is used to update the data stored by the optimizer, by adjusting the times indicated in argument.

```
1
2  /// \brief update the optimizer for new date
3  ///      - In Backward mode, LP resolution achieved at date p_dateNext,
4  ///      starting with uncertainties given at date p_date and evolving to
   give uncertainty at date p_dateNext,
5  ///      - In Forward mode, LP resolution achieved at date p_date,
6  ///      and uncertainties evolve till date p_dateNext
7  ///      .
```

If your transition problem is time dependant, for example, you should store the value of these arguments. Depending on your needs, you can also update data such as the

average demand at the current stage and at next time step of a gas storage problem. The `p_dateNext` argument is used as the current time step in the backward pass. Therefore, you should store the values of the current and next time step arguments.

- The `oneAdmissibleState` function gives an admissible state (i.e. a state respecting all the constraints) for the time step given in argument.

```
1  /// \param p_date    current date
```

- The `oneStepBackward` function is used to compute a step of the backward pass.

```
1  /// \return a vector with the optimal value and the derivatives if the function
    value with respect to each state
```

The first argument is the cuts already selected for the current time step. They are easy to manage, just use the `getCutsAssociatedToAParticle` function as described in the examples you can find in the test folder (`OptimizeReservoirWithInflowsSDDP.h` without regression or `OptimizeGasStorageSDDP.h` with regression). You will then have the necessary cuts in the form of an array *cuts* that you can link to the values described in the theoretical part at the time step t by $cuts(0, j) = \alpha_{t+1}^j$, $cuts(i, j) = \beta_{i-1, t+1}^j$ $j \in \{1, \dots, G, \dots, (n+1)G\}$, $i \in \{1, \dots, nb_{state}\}$.

You will need to add the cuts to your constraints yourself, using this table and your solver features.

In addition, as an argument, you have the object containing the state at the start of the time step `p_astate` (**keep in mind that this argument is given as an Eigen array**), `p_particle` contains the random quantities on which the regression on the expectation of the value function will be based (the computational cost is high so take a look at the theoretical part to know when you really need to use), finally the last argument is an integer indicating in which scenario index the resolution will be performed.

The function returns a one-dimensional array of size $nb_{state} + 1$ containing as first argument the value of the objective function to the solution, then for $i \in \{1, \dots, nb_{state}\}$ it contains the derivatives of the objective function with respect to each of the dimensions i of the state (we must find a way to have it using the dual solution for example).

- The `oneStepForward` function allows you to compute a step of the forward pass.

```
1  /// \param p_isimu    number of teh simulation used
```

As you can see, the `oneStepForward` is quite similar to the `oneStepBackward`. A trick, used in the examples and which you should use, is to build a function generating and solving the linear problem $[AP_{t,g}^n]$ (for a given scenario g and a given time step t) which appears for both the forward and the backward pass. This function creating and generating the linear problem will be called in our two functions `oneStepForward` and `oneStepBackward`. Make sure that in the forward pass, the current time step is given via the function `updateDates(current date, next date)` by the current date argument while in the backward pass the current time is given via the next date argument (this is a necessary requirement to compute the regressions exposed in the theoretical part).

Finally note that the two functions previously described are `const` functions and that you must take them into account during your implementation.

- The other functions that you must implement are simple functions (accessors) that are easy to understand.

Implement your own Simulator class

This simulator must be the object that will allow you to build random quantities according to a desired law. It must be given as an argument of your optimizer. You can implement it yourself, but a set of simulators (gaussian, AR1, MeanReverting,...) is given in the test folder, you can use it directly if it meets your problem requirements.

A simulator for the regression based method

An implemented **Simulator** deriving from the `SimulatorSDDPBase` class must implement these functions:

- The `getNbSimul` function returns the number of simulations of random quantities used in the regression part. This is the U mentioned in the theoretical part.

```
1 virtual int getNbSimul() const = 0;
```

- The `getNbSample` function returns the number of simulations of random quantities that are not used in the regression part. This is the G mentioned in the theoretical part. For example, in some cases we need a Gaussian random quantity to calculate the noise when we are in the “dependence of random quantities” part.

```
1 virtual int getNbSample() const = 0;
```

- The `updateDateIndex` function is very similar to that of the optimizer. However you have one argument (the index of the time step) here. It is also where you need to generate new random amounts for solving.

```
1 /// \param p_idateCurr index in date array
```

- The `getOneParticle` and the `getParticles` functions should return the quantities used in regression part.

```
1 /// \brief get current Markov state
```

```
1 /// \brief get current Markov state
```

- The two last functions `resetTime` and `updateSimulationNumberAndResetTime` are quite explicit.

A simulator for the tree approach

A simulator using tree must be derived from the `SimulatorSDDPBaseTree` class. Since the `SimulatorSDDPBaseTree` class is derived from the `SimulatorSDDPBase` class, all of the previously described methods must be given.

Besides a `geners` archive is used to load:

- The dates used by the simulator to estimate the set of possible states,
- At each date, a set of d dimensional points defining the set of discrete values of the state in the tree,
- At each date a two-dimensional array giving the probability of transition in the tree to go from a node i on the current date to a node j on the following date.

Then the implemented simulator must call the based constructor loading the archive:

```
1  /// \brief Get back the number of particles
2  virtual int getNbSimul() const
3  {
4      return 0;
5  }
6
7  /// \brief Get back the number of sample used (simulation at each time step , these
    simulations are independent of the state)
```

Then the user must have generated such an archive. An example of using a trinomial tree method for an AR1 class is given in the c++ test cases in the simulator directory by the `MeanRevertingSimulatorTree` class.

The necessary methods to be implemented are as follows:

- The `getNodeAssociatedToSim` method gives for a simulation identified by the particle number the node in the visited tree.
- The `stepForward` method updates the simulation date index by one, and samples visited nodes in forward resolution.

1.3.4 Set of parameters

Implementing some regression-based method

The base function `backwardForwardSDDP` must be called to use the SDDP part of the library with conditional cuts calculated by regressions. This function is modeled by the regressor used:

- `LocalConstRegressionForSDDP` The regressor allows to use a constant approximation by mesh of SDDP cuts,

- **LocalLinearRegressionForSDDP** The regressor allows to use a linear approximation by mesh of SDDP cuts.

```

1
2 /// \brief Achieve forward and backward sweep by SDDP
3 /// \param p_optimizer defines the optimiser necessary to optimize a step for
   one simulation solving a LP
4 /// \param p_nbSimulCheckForSimu defines the number of simulations to check convergence
5 /// \param p_initialState initial state at the beginning of simulation
6 /// \param p_finalCut object of final cuts
7 /// \param p_dates vector of exercised dates, last date corresponds to the
   final cut object
8 /// \param p_meshForReg number of mesh for regression in each direction
9 /// \param p_nameRegressor name of the archive to store regressors
10 /// \param p_nameCut name of the archive to store cuts
11 /// \param p_nameVisitedStates name of the archive to store visited states
12 /// \param p_iter maximum iteration of SDDP, on return the number of
   iterations achieved
13 /// \param p_accuracy accuracy asked , on return estimation of accuracy
   achieved (expressed in %)
14 /// \param p_nStepConv every p_nStepConv convergence is checked
15 /// \param p_outputStream dump all print messages
16 /// \param p_world MPI communicator
17 /// \param p_bPrintTime if true print time at each backward and forward step
18 /// \return backward and forward valorization
19 template< class LocalRegressionForSDDP>
20 std::pair<double, double> backwardForwardSDDP(const std::shared_ptr<OptimizerSDDPBase> &
   p_optimizer,
21         const int &p_nbSimulCheckForSimu,
22         const Eigen::ArrayXd &p_initialState,
23         const SDDPFinalCut &p_finalCut,
24         const Eigen::ArrayXd &p_dates,
25         const Eigen::ArrayXi &p_meshForReg,
26         const std::string &p_nameRegressor,

```

Most of the arguments are pretty straightforward (you can see examples in `test/c++/functional`). The strings correspond to the names which will be given by the files which will store cuts, visited states or regressor data. `p_nbSimulCheckForSimu` corresponds to the number of simulations (number of forward passes called) when it is necessary to check the convergence by comparing the result given by the forward pass and that given by the backward pass. `p_nStepConv` indicates when the convergence is verified (each `p_nStepConv` iteration). `p_finalCut` corresponds to the cut used at the last time step: when the function of final value is zero, the last cut is given by an all zero array of size $nb_{state} + 1$. `p_dates` is an array made up with all the time steps of the study period given as doubles, `p_iter` corresponds to the maximum number of iterations. Finally, `p_stringStream` is a `ostream` in which the result of the optimization will be stored.

Implementing a tree based method

The base function `backwardForwardSDDPTree` must be called to use the SDDP part of the library with conditional cuts calculated with trees.

```

1
2 /// \brief Achieve forward and backward sweep by SDDP with tree
3 /// \param p_optimizer defines the optimiser necessary to optimize a step for
   one simulation solving a LP
4 /// \param p_nbSimulCheckForSimu defines the number of simulations to check convergence
5 /// \param p_initialState initial state at the beginning of simulation
6 /// \param p_finalCut object of final cuts

```

```

7 /// \param p_dates          vector of exercised dates, last date corresponds to the
   final cut object
8 /// \param p_nameCut        name of the archive to store cuts
9 /// \param p_nameVisitedStates name of the archive to store visited states
10 /// \param p_iter           maximum iteration of SDDP, on return the number of
   iterations achieved
11 /// \param p_accuracy       accuracy asked , on return estimation of accuracy
   achieved (expressed in %)
12 /// \param p_nStepConv      every p_nStepConv convergence is checked
13 /// \param p_stringStream    dump all print messages
14 /// \param p_world           MPI communicator
15 /// \param p_bPrintTime      if true print time at each backward and forward step
16 /// \return backward and forward valorization
17 std::pair<double, double> backwardForwardSDDPTree(std::shared_ptr<OptimizerSDDPBase> &
   p_optimizer,
18     const int &p_nbSimulCheckForSimu,
19     const Eigen::ArrayXd &p_initialState,
20     const SDDPFinalCutTree &p_finalCut,
21     const Eigen::ArrayXd &p_dates,
22     const std::string &p_nameCut,
23     const std::string &p_nameVisitedStates,
24     int &p_iter,
25     double &p_accuracy,
26     const int &p_nStepConv,

```

1.3.5 The black box

The algorithms described above are applied. As stated earlier, the user controls the implementation of the business side of the problem (transition problem). But in the library some things are handled automatically and the user should be aware of:

- The **Parallelization** When solving the problem is handled automatically. During compilation, if the compiler detects a problem with the MPI (Message Passing Interface) library, the resolution will be carried out in a parallelized manner.
- The **cut management**. All the cuts added with each iteration are currently serialized and stored in an archive initialized by the user. No cut is pruned. In the future, we can consider working on [39] cut management.
- A **double stop criterion** is hardly used by the library: a convergence test and a maximum number of iterations. If either of the two criteria exceeds the user-defined thresholds, the resolution stops automatically. Once again further work could be considered on this subject.

1.3.6 Outputs

SDDP library outputs are not currently defined. Thus during the resolution of an SDDP problem, only the number of iterations, the evolution of the backward and forward costs and of the convergence criterion are recorded.

However, while iterating backwards and forwards the value of the Bellman functions and the associated Benders cuts, the different states visited during the forward pass and the

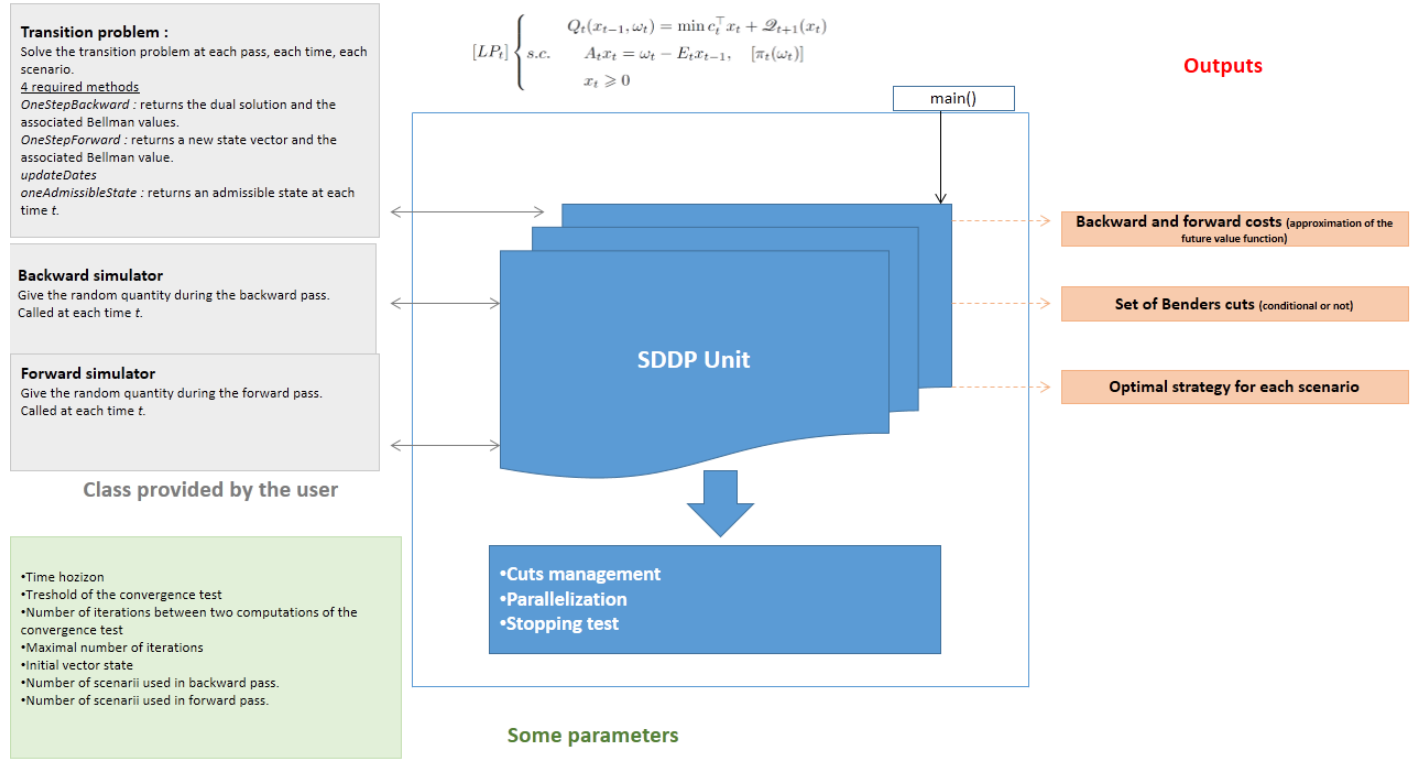


Figure 1.1: Current architecture of the generic SDDP unit

evolution of costs are stored at each moment of the time horizon. This information is useful for users and easy to enter.

Once the convergence has been achieved, the user must restart certain simulations by adding a flag to store the results necessary for the application (distribution cost etc.): these results will be post-processed by the user.

1.4 Python API (only for regression-based methods)

A high level Python mapping is also available in the SDDP part. The backward-forward C++ function is exposed in Python by the SDDP `StOptSDDP` module. In this mapping, only the linear per mesh regressor is used.

```
1 import pystopt.sddp.StOptSDDP
2 dir(StOptSDDP)
```

that should give

```
['OptimizerSDDPBase', 'SDDPFinalCut', 'SimulatorSDDPBase', '__doc__', '__file__', '__name__', '__package__', 'backwardForwardSDDP']
```

The `backwardForwardSDDP` performs the forward and backward SDDP sweep giving an SDDP optimizer and an SDDP uncertainty simulator. The initial final cuts for the last time steps are provided by the `SDDPFinalCut` object.

Perform the mapping of SDDP optimizers and simulators written in C++ it is necessary to

create a Boost Python wrapper. In order to expose the C++ optimization class `OptimizeDemandSDDP` used in the test case `testDemandSDDP.cpp`, the following wrapper can be found in

`StOpt/test/c++/python/Pybind11SDDPOptimizers.cpp`

```

1 // Copyright (C) 2019 EDF
2 // All Rights Reserved
3 // This code is published under the GNU Lesser General Public License (GNU LGPL)
4 #ifndef SDDPPYTHON
5 #include <pybind11/pybind11.h>
6 #include <pybind11/eigen.h>
7 #include <pybind11/stl_bind.h>
8 #include <pybind11/stl.h>
9 #include "StOpt/core/grids/OneDimRegularSpaceGrid.h"
10 #include "StOpt/core/grids/OneDimData.h"
11 #include "StOpt/sddp/OptimizerSDDPBase.h"
12 #include "test/c++/tools/sddp/OptimizeDemandSDDP.h"
13 #include "test/c++/tools/simulators/SimulatorGaussianSDDP.h"
14 #include "test/c++/python/FutureCurveWrap.h"
15
16 /** \file Pybind11SDDPOptimizers.cpp
17 * \brief permits to map Optimizers for SDDP
18 * \author Xavier Warin
19 */
20
21
22 /// \wrapper for Optimizer for demand test case in SDDP
23 class OptimizeDemandSDDPWrap : public OptimizeDemandSDDP<SimulatorGaussianSDDP>
24 {
25 public :
26
27     /// \brief Constructor
28     /// \param p_sigD volatility for demand
29     /// \param p_kappaD AR coefficient for demand
30     /// \param p_timeDAverage average demand
31     /// \param p_spot Spot price
32     /// \param p_simulatorBackward backward simulator
33     /// \param p_simulatorForward Forward simulator
34     OptimizeDemandSDDPWrap(const double &p_sigD, const double &p_kappaD,
35                             const FutureCurve &p_timeDAverage,
36                             const double &p_spot,
37                             const std::shared_ptr<SimulatorGaussianSDDP> &
38                                 p_simulatorBackward,
39                             const std::shared_ptr<SimulatorGaussianSDDP> &p_simulatorForward
40                             ):
41         OptimizeDemandSDDP(p_sigD, p_kappaD,
42                             std::make_shared< StOpt::OneDimData< StOpt::
43                                 OneDimRegularSpaceGrid, double> >(static_cast<StOpt::
44                                 OneDimData< StOpt::OneDimRegularSpaceGrid, double> >(
45                                     p_timeDAverage)),
46                             p_spot, p_simulatorBackward, p_simulatorForward) { }
47 };
48
49 namespace py = pybind11;
50
51 void initSDDPOptimizers(py::module_ &m)
52 {
53     py::class_<OptimizeDemandSDDPWrap, std::shared_ptr<OptimizeDemandSDDPWrap>, StOpt::
54         OptimizerSDDPBase >(m, "OptimizeDemandSDDP")
55         .def(py::init< const double &, const double &, const FutureCurve &,
56             const double &,
57             const std::shared_ptr<SimulatorGaussianSDDP> &,
58             const std::shared_ptr<SimulatorGaussianSDDP> &>())

```

```

56 .def("getSimulatorBackward", &OptimizeDemandSDDP<SimulatorGaussianSDDP>::
    getSimulatorBackward)
57 .def("getSimulatorForward", &OptimizeDemandSDDP<SimulatorGaussianSDDP>::
    getSimulatorForward)
58 .def("oneAdmissibleState", &OptimizeDemandSDDP<SimulatorGaussianSDDP>::
    oneAdmissibleState)
59 ;
60 }
61 #endif

```

The wrapper used to expose the SDDP simulator is given in

StOpt/test/c++/python/Pybind11Simulators.cpp

Then it is possible to use the mapping to write a Python version of testDemandSDDP.cpp

```

1 # Copyright (C) 2016 EDF
2 # All Rights Reserved
3 # This code is published under the GNU Lesser General Public License (GNU LGPL)
4 import pystopt.grids as StOptGrids
5 import pystopt.sddp as StOptSDDP
6 import pystopt.glob as StOptGlobal
7 import pystopttest.utils
8 import pystopttest.SDDPSimulators as sim
9 import pystopttest.SDDPOptimizers as opt
10 import numpy as NP
11 import unittest
12 import math
13 import importlib.util
14 import sddp.backwardForwardSDDP as bfSDDP # import of the function written in python
15
16 # unittest equivalent of testDemandSDDP : here MPI version
17 # High level python interface : at level of the backwardForwardSDDP c++ file
18 #####
19 def demandSDDPFunc(p_sigD, p_sampleOptim, p_sampleCheckSimul):
20
21     maturity = 40
22     nstep = 40;
23
24     # optimizer parameters
25     kappaD = 0.2; # mean reverting coef of demand
26     spot = 3 ; # spot price
27
28     # define a a time grid
29     timeGrid = StOptGrids.OneDimRegularSpaceGrid(0., maturity / nstep, nstep)
30
31     # periodicity factor
32     iPeriod = 52;
33     # average demande values
34     demValues = []
35
36     for i in list(range(nstep + 1)) :
37         demValues.append(2. + 0.4 * math.cos((math.pi * i * iPeriod) / nstep))
38
39     # define average demand
40     demGrid = pystopttest.utils.FutureCurve(timeGrid, demValues)
41
42     initialState = demGrid.get(0.)*NP.ones(1)
43
44     finCut = StOptSDDP.SDDPFinalCut(NP.zeros((2,1)))
45
46     # here cuts are not conditional to an uncertainty
47     nbMesh = NP.array([],NP.int32)
48     nbUncertainties = 1;
49
50     # backward simulator
51     backwardSimulator = sim.SimulatorGaussianSDDP(nbUncertainties,p_sampleOptim)
52     # forward simulator
53     forwardSimulator = sim.SimulatorGaussianSDDP(nbUncertainties)
54

```

```

55     # Create the optimizer
56     optimizer = opt.OptimizeDemandSDDP(p_sigD, kappaD, demGrid, spot,
    backwardSimulator, forwardSimulator)
57
58     # optimisation dates
59     dates = NP.linspace( 0., maturity,nstep + 1);
60
61     # names for archive
62     nameRegressor = "RegressorDemand";
63     nameCut = "CutDemand";
64     nameVisitedStates = "VisitedStateDemand";
65
66     # precision parameter
67     nIterMax = 40
68     accuracyClose = 1.
69     accuracy = accuracyClose / 100.
70     nstepIterations = 4; # check for convergence between nstepIterations step
71
72     values = StOptSDDP.backwardForwardSDDP(optimizer, p_sampleCheckSimul, initialState
    , finCut, dates, nbMesh, nameRegressor, nameCut, nameVisitedStates, nIterMax,
    accuracy, nstepIterations);
73
74
75     print("Values " , values)
76     return values
77
78
79 # unittest equivalent of testDemandSDDP : here low interface python version
80 # Low level python interface : use backwardForwardSDDP.py
81 #####
82 def demandSDDPFuncLowLevel(p_sigD, p_sampleOptim ,p_sampleCheckSimul):
83
84     maturity = 40
85     nstep = 40;
86
87     # optimizer parameters
88     kappaD = 0.2; # mean reverting coef of demand
89     spot = 3 ; # spot price
90
91     # define a a time grid
92     timeGrid = StOptGrids.OneDimRegularSpaceGrid(0., maturity / nstep, nstep)
93
94
95     # periodicity factor
96     iPeriod = 52;
97     # average demande values
98     demValues = []
99
100     for i in list(range(nstep + 1)) :
101         demValues.append(2. + 0.4 * math.cos((math.pi * i * iPeriod) / nstep))
102
103     # define average demand
104     demGrid =pystopttest.utils.FutureCurve(timeGrid, demValues)
105
106     initialState = demGrid.get(0.)*NP.ones(1)
107
108     finCut = StOptSDDP.SDDPFinalCut(NP.zeros((2,1)))
109
110     # here cuts are not conditional to an uncertainty
111     nbMesh = NP.array([],NP.int32)
112     nbUncertainties = 1;
113
114     # backward simulator
115     backwardSimulator = sim.SimulatorGaussianSDDP(nbUncertainties,p_sampleOptim)
116     # forward simulator
117     forwardSimulator = sim.SimulatorGaussianSDDP(nbUncertainties)
118
119     # Create the optimizer
120     optimizer = opt.OptimizeDemandSDDP(p_sigD, kappaD, demGrid, spot,
    backwardSimulator, forwardSimulator)

```

```

121
122     # optimisation dates
123     dates = NP.linspace( 0., maturity, nstep + 1);
124
125     # names for archive
126     nameRegressor = "RegressorDemand";
127     nameCut = "CutDemand";
128     nameVisitedStates = "VisitedStateDemand";
129
130     # precision parameter
131     nIterMax = 40
132     accuracyClose = 1.
133     accuracy = accuracyClose / 100.
134     nstepIterations = 4; # check for convergence between nstepIterations step
135
136     values = bfSDDP.backwardForwardSDDP(optimizer, p_sampleCheckSimul, initialState,
137                                         finCut, dates, nbMesh, nameRegressor,
138                                         nameCut, nameVisitedStates, nIterMax,
139                                         accuracy, nstepIterations);
140
141     return values
142
143 class testDemandSDDP(unittest.TestCase):
144     def testDemandSDDP1D(self):
145         moduleMpi4Py=importlib.util.find_spec('mpi4py')
146         if (moduleMpi4Py is not None):
147             from mpi4py import MPI
148             world = MPI.COMM_WORLD
149             found = True
150             sigD = 0.6 ;
151             sampleOptim = 500;
152             sampleCheckSimul = 500;
153
154             values = demandSDDPFunc(sigD, sampleOptim ,sampleCheckSimul)
155
156             if (world.rank==0):
157                 print("Values is ",values)
158
159     def testDemandSDDP1DLowLevel(self):
160         sigD = 0.6 ;
161         sampleOptim = 500;
162         sampleCheckSimul = 500;
163         demandSDDPFuncLowLevel(sigD, sampleOptim ,sampleCheckSimul)
164
165
166 if __name__ == '__main__':
167     unittest.main()

```

Part VII

Nesting Monte Carlo for general nonlinear PDEs

The described method has been studied in [53], [52] and uses some ideas in [23], [51]. Our goal is to solve the complete nonlinear equation

$$\begin{aligned} (-\partial_t u - \mathcal{L}u)(t, x) &= f(t, x, u(t, x), Du(t, x), D^2u(t, x)), \\ u_T &= g, \quad t < T, \quad x \in \mathbb{R}^d, \end{aligned} \quad (23)$$

with

$$\mathcal{L}u(t, x) := \mu Du(t, x) + \frac{1}{2} \sigma \sigma^\top : D^2u(t, x)$$

so that $\mathcal{L}$ is the generator associated with

$$X_t = x + \mu t + \sigma dW_t,$$

with $\mu \in \mathbb{R}^d$, and $\sigma \in \mathbb{M}^d$ is a constant matrix.

Throughout the article, ρ is the density of a general random variable following a gamma law so that

$$\rho(x) = \lambda^\alpha x^{\alpha-1} \frac{e^{-\lambda x}}{\Gamma(\alpha)}, \quad 1 \geq \alpha > 0. \quad (24)$$

The associated cumulative distribution function is

$$F(x) = \frac{\gamma(\alpha, \lambda x)}{\Gamma(\alpha)}$$

where $\gamma(s, x) = \int_0^x t^{s-1} e^{-t} dt$ is the incomplete gamma function and $\Gamma(s) = \int_0^\infty t^{s-1} e^{-t} dt$ is the gamma function.

The methodology follows the ideas of [53] and [51].

It is assumed here that σ is not degenerate so that σ^{-1} exists.

Let set $p \in \mathbb{N}^+$. For $(N_0, \dots, N_{p-1}) \in \mathbb{N}^p$, we introduce the sets of i-tuple, $Q_i = \{k = (k_1, \dots, k_i)\}$ for $i \in \{1, \dots, p\}$ where all components $k_j \in [1, N_{j-1}]$. Besides we define $Q^p = \cup_{i=1}^p Q_i$.

We construct the sets Q_i^o for $i = 1, \dots, p$, such that

$$Q_1^o = Q_1$$

and the set Q_i^o for $i > 1$ are defined by recurrence:

$$Q_{i+1}^o = \{(k_1, \dots, k_i, k_{i+1}) / (k_1, \dots, k_i) \in Q_i^o, k_{i+1} \in \{1, \dots, N_{i+1}, 1_1, \dots, (N_{i+1})_1, 1_2, \dots, (N_{i+1})_2\}\}$$

so that to a particle noted $(k_1, \dots, k_i) \in Q_i^o$ such that $k_i \in \mathbb{N}$, we associate two fictitious particles noted $k^1 = (k_1, \dots, k_{i-1}, (k_i)_1)$ and $k^2 = (k_1, \dots, k_{i-1}, (k_i)_2)$.

To a particle $k = (k_1, \dots, k_i) \in Q_i^o$ we associate its original particle $o(k) \in Q_i$ such that $o(k) = (\hat{k}_1, \dots, \hat{k}_i)$ where $\hat{k}_j = l$ if $k_j = l, l_1$ or l_2 .

For $k = (k_1, \dots, k_i) \in Q_i^o$ we introduce the set of its non fictitious sons

$$\tilde{Q}(k) = \{l = (k_1, \dots, k_i, m) / m \in \{1, \dots, N_i\}\} \subset Q_{i+1}^o,$$

and the set of all sons

$$\hat{Q}(k) = \{l = (k_1, \dots, k_i, m)/m \in \{1, \dots, N_i, 1_1, \dots, (N_i)_1, 1_2, \dots, (N_i)_2\}\} \subset Q_{i+1}^o.$$

By convention $\tilde{Q}(\emptyset) = \{l = (m)/m \in \{1, \dots, N_0\}\} = Q_1$. Reciprocally the ancestor k of a particle $\tilde{k}$ in $\tilde{Q}(k)$ is noted $\tilde{k}^-$.

We define the order of a particle $k \in Q_i^o$, $i \geq 0$, by the function κ :

$$\begin{aligned}\kappa(k) &= 0 \text{ for } k_i \in \mathbb{N}, \\ \kappa(k) &= 1 \text{ for } k_i = l_1, l \in \mathbb{N} \\ \kappa(k) &= 2 \text{ for } k_i = l_2, l \in \mathbb{N}\end{aligned}$$

We define the sequence τ_k of switching increments i.i.d. random density variables ρ for $k \in Q^p$. The switching dates are defined as follows:

$$\begin{cases} T_{(j)} &= \tau_{(j)} \wedge T, j \in \{1, \dots, N_0\} \\ T_{\tilde{k}} &= (T_k + \tau_{\tilde{k}}) \wedge T, k = (k_1, \dots, k_i) \in Q_i, \tilde{k} \in \tilde{Q}(k) \end{cases} \quad (25)$$

By convention $T_k = T_{o(k)}$ and $\tau_k = \tau_{o(k)}$. For $k = (k_1, \dots, k_i) \in Q_i^o$ and $\tilde{k} = (k_1, \dots, k_i, k_{i+1}) \in \hat{Q}(k)$ we define the following trajectories:

$$W_s^{\tilde{k}} := W_{T_k}^k + \mathbf{1}_{\kappa(\tilde{k})=0} \bar{W}_{s-T_k}^{o(\tilde{k})} - \mathbf{1}_{\kappa(\tilde{k})=1} \bar{W}_{s-T_k}^{o(\tilde{k})}, \quad \text{and} \quad (26)$$

$$X_s^{\tilde{k}} := x + \mu s + \sigma W_s^{\tilde{k}}, \quad \forall s \in [T_k, T_{\tilde{k}}], \quad (27)$$

where the $\bar{W}^k$ for k in Q^p are independent d -dimensional Brownian motions, independent of the $(\tau_k)_{k \in Q^p}$.

To understand what these different trajectories represent, suppose that $d = 1$, $\mu = 0$, $\sigma = 1$ and consider the original particle $k = (1, 1, 1)$ such that $T_{(1,1,1)} = T$.

Following equation (26),

$$\begin{aligned}X_T^{(1,1,1)} &= \bar{W}_{T_{(1)}}^{(1)} + \bar{W}_{T_{(1,1)}-T_{(1)}}^{(1,1)} + \bar{W}_{T-T_{(1,1)}}^{(1,1,1)} \\ X_T^{(1_1,1,1)} &= -\bar{W}_{T_{(1)}}^{(1)} + \bar{W}_{T_{(1,1)}-T_{(1)}}^{(1,1)} + \bar{W}_{T-T_{(1,1)}}^{(1,1,1)} \\ X_T^{(1,1_1,1)} &= \bar{W}_{T_{(1)}}^{(1)} - \bar{W}_{T_{(1,1)}-T_{(1)}}^{(1,1)} + \bar{W}_{T-T_{(1,1)}}^{(1,1,1)} \\ X_T^{(1_2,1_1,1)} &= -\bar{W}_{T_{(1,1)}-T_{(1)}}^{(1,1)} + \bar{W}_{T-T_{(1,1)}}^{(1,1,1)} \\ &\dots\end{aligned}$$

such that all particles are generated from the $\bar{W}^k$ used to define $X_T^{(1,1,1)}$.

Using the previous definitions, we consider the estimator defined by:

$$\left\{ \begin{array}{l} \bar{u}_\emptyset^p = \frac{1}{N_0} \sum_{j=1}^{N_0} \phi(0, T_{(j)}, X_{T_{(j)}}^{(j)}, \bar{u}_{(j)}^p, D\bar{u}_{(j)}^p, D^2\bar{u}_{(j)}^p), \\ \bar{u}_k^p = \frac{1}{N_i} \sum_{\tilde{k} \in \tilde{Q}(k)} \frac{1}{2} (\phi(T_k, T_{\tilde{k}}, X_{T_{\tilde{k}}}^{\tilde{k}}, \bar{u}_{\tilde{k}}^p, D\bar{u}_{\tilde{k}}^p, D^2\bar{u}_{\tilde{k}}^p) + \\ \quad \phi(T_k, T_{\tilde{k}}, X_{T_{\tilde{k}}}^{\tilde{k}^1}, \bar{u}_{\tilde{k}^1}^p, D\bar{u}_{\tilde{k}^1}^p, D^2\bar{u}_{\tilde{k}^1}^p)), \quad \text{for } k = (k_1, \dots, k_i) \in Q_i^o, 0 < i < p, \\ D\bar{u}_k^p = \frac{1}{N_i} \sum_{\tilde{k} \in \tilde{Q}(k)} \mathbb{V}^{\tilde{k}} \frac{1}{2} (\phi(T_k, T_{\tilde{k}}, X_{T_{\tilde{k}}}^{\tilde{k}}, \bar{u}_{\tilde{k}}^p, D\bar{u}_{\tilde{k}}^p, D^2\bar{u}_{\tilde{k}}^p) - \\ \quad \phi(T_k, T_{\tilde{k}}, X_{T_{\tilde{k}}}^{\tilde{k}^1}, \bar{u}_{\tilde{k}^1}^p, D\bar{u}_{\tilde{k}^1}^p, D^2\bar{u}_{\tilde{k}^1}^p)), \quad \text{for } k = (k_1, \dots, k_i) \in Q_i^o, 0 < i < p, \\ D^2\bar{u}_k^p = \frac{1}{N_i} \sum_{\tilde{k} \in \tilde{Q}(k)} \mathbb{W}^{\tilde{k}} \frac{1}{2} (\phi(T_k, T_{\tilde{k}}, X_{T_{\tilde{k}}}^{\tilde{k}}, \bar{u}_{\tilde{k}}^p, D\bar{u}_{\tilde{k}}^p, D^2\bar{u}_{\tilde{k}}^p) + \\ \quad \phi(T_k, T_{\tilde{k}}, X_{T_{\tilde{k}}}^{\tilde{k}^1}, \bar{u}_{\tilde{k}^1}^p, D\bar{u}_{\tilde{k}^1}^p, D^2\bar{u}_{\tilde{k}^1}^p) - \\ \quad 2\phi(T_k, T_{\tilde{k}}, X_{T_{\tilde{k}}}^{\tilde{k}^2}, \bar{u}_{\tilde{k}^2}^p, D\bar{u}_{\tilde{k}^2}^p, D^2\bar{u}_{\tilde{k}^2}^p)), \quad \text{for } k = (k_1, \dots, k_i) \in Q_i^o, 0 < i < p, \\ \bar{u}_k^p = g(X_{T_k}^k), \quad \text{for } k \in Q_p^o, \\ D\bar{u}_k^p = Dg(X_{T_k}^k), \quad \text{for } \tilde{k} \in Q_p^o, \\ D^2\bar{u}_k^p = D^2g(X_{T_k}^k), \quad \text{for } \tilde{k} \in Q_p^o \end{array} \right.$$

(28)

where ϕ is defined by:

$$\phi(s, t, x, y, z, \theta) := \frac{\mathbf{1}_{\{t \geq T\}}}{F(T-s)} g(x) + \frac{\mathbf{1}_{\{t < T\}}}{\rho(t-s)} f(t, x, y, z, \theta). \quad (29)$$

and

$$\mathbb{V}^k = \sigma^{-\top} \frac{\bar{W}_{T_k - T_{k-}}^k}{T_k - T_{k-}}$$

,

$$\mathbb{W}^k = (\sigma^\top)^{-1} \frac{\bar{W}_{T_k - T_{k-}}^k (\bar{W}_{T_k - T_{k-}}^k)^\top - (T_k - T_{k-}) I_d}{(T_k - T_{k-})^2} \sigma^{-1} \quad (30)$$

As explained previously, the terms u and Du in f are treated as explained in [53] and only the D^2u treatment is new to this scheme.

Remark 52 *In practice, we just have the value g at the deadline T and we want to apply the scheme even if the derivatives of the final solution is not defined. We can close the system for k in Q_p° by replacing ϕ with g and taking a value for N_{p+1} :*

$$\begin{aligned} \bar{u}_k^p &= \frac{1}{N_{p+1}} \sum_{\tilde{k} \in \tilde{Q}(k)} \frac{1}{2} (g(X_{T_{\tilde{k}}}^{\tilde{k}}) + g(X_{T_{\tilde{k}}}^{\tilde{k}^1})), \\ D\bar{u}_k^p &= \frac{1}{N_{p+1}} \sum_{\tilde{k} \in \tilde{Q}(k)} \mathbb{V}^{\tilde{k}} \frac{1}{2} (g(X_{T_{\tilde{k}}}^{\tilde{k}}) - g(X_{T_{\tilde{k}}}^{\tilde{k}^1})), \\ D^2\bar{u}_k^p &= \frac{1}{N_{p+1}} \sum_{\tilde{k} \in \tilde{Q}(k)} \mathbb{W}^{\tilde{k}} \frac{1}{2} (g(X_{T_{\tilde{k}}}^{\tilde{k}}) + g(X_{T_{\tilde{k}}}^{\tilde{k}^1}) - 2g(X_{T_{\tilde{k}}}^{\tilde{k}^2})) \end{aligned}$$

Remark 53 *In the case where the coefficient are not constant, some Euler scheme can be added as explained in [53].*

An efficient algorithm for this scheme has these two functions:

Algorithm 20 External Monte Carlo algorithm (V generates unit Gaussian RV, $\tilde{V}$ generates an RV with gamma law density)

```

1: procedure PDEEVAL( $\mu, \sigma, g, f, T, p, x_0, \{N_0, \dots, N_{p+1}\}, V, \tilde{V}$ )
2:    $u_M = 0$ 
3:    $x(0, :) = x_0(:)$   $\triangleright x$  is a matrix of size  $1 \times n$ 
4:   for  $i = 1, N_0$  do
5:      $(u, Du, D^2u) = \text{EvalUDUD2U}(x_0, \mu, \sigma, g, T, V, \tilde{V}, p, 1, 0, 0)$ 
6:      $u_M = u_M + u(0)$ 
7:   end for
8: return  $\frac{u_M}{N_0}$ 
9: end procedure
```

Algorithm 21 Internal Monte Carlo algorithm where t is the current time, x the array of particles positions of size $m \times d$, and l the nesting level.

```

1: procedure EVALUDUD2U( $x, \mu, \sigma, g, T, V, \tilde{V}, p, m, t, l$ )
2:    $\tau = \min(\tilde{V}(), T - t),$  ▷ Sample the time step
3:    $G = V()$  ▷ Sample the  $n$  dimensional Gaussian vector
4:    $xS(1 : m, :) = x(:) + \mu\tau + \sigma G\sqrt{\tau}$ 
5:    $xS(m + 1 : 2m, :) = x(:) + \mu\tau$ 
6:    $xS(2m + 1 : 3m, :) = x(:) + \mu\tau - \sigma G\sqrt{\tau}$ 
7:    $tS = t + \tau$  ▷ New date
8:   if  $ts \geq T$  or  $l = p$  then
9:      $g_1 = g(xS(1 : m, :)); g_2 = (xS(m + 1 : 2m, :)); g_3 = g(xS(2m + 1 : 3m, :))$ 
10:     $u(:) = \frac{1}{2}(g_1 + g_3)$ 
11:     $Du(:, :) = \frac{1}{2}(g_1 - g_3) \sigma^{-\top} G$ 
12:     $D^2u(:, :, :) = \frac{1}{2}(g_1 + g_3 - 2g_2)\sigma^{-\top} \frac{GG^\top - \mathbf{I}_d}{\tau} \sigma^{-1}$ 
13:    if  $l \neq p$  then
14:       $(u(:), Du(:, :), D^2u(:, :, :)) / = \frac{1}{F(\tau)}$ 
15:    end if
16:  else
17:     $y(:) = 0; z(:, :) = 0; \theta(:, :, :) = 0$ 
18:    for  $j = 1, N_{l+1}$  do
19:       $(y, z, \theta) + = \text{EvalUDUD2U}(xS, \mu, \sigma, g, T, V, \tilde{V}, p, 3m, tS, l + 1)$ 
20:    end for
21:     $(y, z, \theta) / = N_{l+1}$ 
22:    for  $q = 1, m$  do
23:       $f_1 = f(ts, xS(q), y(q), z(q, :), \theta(q, :, :))$ 
24:       $f_2 = f(ts, xS(m + q), y(m + q), z(m + q, :), \theta(m + q, :, :))$ 
25:       $f_3 = f(ts, xS(2m + q), y(2m + q), z(2m + q, :), \theta(2m + q, :, :))$ 
26:       $u(i) = \frac{1}{2}(f_1 + f_3)$ 
27:       $Du(i, :) = \frac{1}{2}(f_1 - f_3)\sigma^{-\top} G$ 
28:       $D^2u(i, :, :) = \frac{1}{2}(f_1 + f_3 - 2f_2)\sigma^{-\top} \frac{GG^\top - \mathbf{I}_d}{\tau} \sigma^{-1}$ 
29:    end for
30:  end if
31: return  $(u, Du, D^2u)$ 
32: end procedure

```

Part VIII

Some test cases description

Description of some test cases in C++ In this part, we describe the functional test cases of the library. The C++ version of these test cases can be found in `test/c++/functional` while their Python equivalent (if it exists) is in `test/Python/functional`. We describe the C++ test cases in detail here.

0.5 American option

The library gives some test cases for the Bermudean options problem ([8] for more details on the Bermudean option problem). All Bermudean test cases use a basket option payoff. The reference for converged methods can be found in [8].

0.5.1 testAmerican

The test case in this file makes it possible to test during the Dynamic Programming resolution different regressors:

- either by using some local functions with the same size support:
 - Either by using a constant representation by mesh of the function (`LocalSameSizeConstRegression` regressor)
 - Either by using a linear representation by mesh of the function (`LocalSameSizeLinearRegression` regressor)
- either using some function basis with adaptive support ([8])
 - Either by using a constant representation per mesh of the function (`LocalConstRegression` regressor)
 - Either by using a linear representation by mesh of the function (`LocalLinearRegression` regressor)
- Either using the global polynomial regressor:
 - Either using Hermite polynomials,
 - Either using Canonical polynomials (monomes),
 - Either using Tchebychev polynomials.
- Either using a sparse regressor,
- Either using kernel regressors:
 - either using a constant kernel regressor,
 - either using linear kernel regressor.

testAmericanLinearBasket1D

Test 1D problem with `LocalLinearRegression` regressor.

testAmericanConstBasket1D

Test 1D problem with LocalConstRegression regressor.

testAmericanSameSizeLinearBasket1D

Test 1D problem with LocalSameSizeLinearRegression regressor.

testAmericanSameSizeConstBasket1D

Test 1D problem with LocalSameSizeConstRegression regressor.

testAmericanGlobalBasket1D

Test 1D problem with global Hermite, Canonical and Tchebychev regressor.

testAmericanGridKernelConstBasket1D

Test 1D problem with classical kernel regression

testAmericanGridKernelLinearBasket1D

Test 1D problem with linear kernel regression

testAmericanLinearBasket2D

Test 2D problem with LocalLinearRegression regressor.

testAmericanConstBasket2D

Test 2D problem with LocalConstRegression regressor.

testAmericanSameSizeLinearBasket2D

Test 2D problem with LocalSameSizeLinearRegression regressor.

testAmericanSameSizeConstBasket2D

Test 2D problem with LocalSameSizeConstRegression regressor.

testAmericanGlobalBasket2D

Test 2D problem with global Hermite, Canonical and Tchebychev regressor.

testAmericanGridKernelConstBasket2D

Test 2D problem with classical kernel regression

testAmericanGridKernelLinearBasket1D

Test 2D problem with linear kernel regression

testAmericanBasket3D

Test 3D problem with `LocalLinearRegression` regressor.

testAmericanGlobalBasket3D

Test 3D problem with global Hermite, Canonical and Tchebychev regressor.

testAmericanGridKernelLinearBasket3D

Test 3D problem with linear kernel regression.

testAmericanBasket4D

Test 4D problem with `LocalLinearRegression` regressor.

0.5.2 testAmericanConvex

Three test cases with basket American options are implemented trying to keep the convexity of the solution

testAmericanLinearConvexBasket1D

Linear regression adapted in 1D preserving the convexity at each time step.

testAmericanLinearConvexBasket2D

Linear regression adapted in 2D trying to preserve the convexity at each time step.

testAmericanLinearConvexBasket3D

Linear regression adapted in 3D trying to preserve the convexity at each time step.

0.5.3 testAmericanForSparse

This test case is there to test sparse grid regressors (see section 1.3). As described earlier, we can use linear, quadratic or cubic representation on each cell. The reference is the same as in the `testAmerican` subsection thus linked to a Bermudean basket option.

testAmericanSparseBasket1D

Use sparse 1D grids (thus equivalent to a full grid) for linear, quadratic or cubic representation.

testAmericanSparseBasket2D

Use sparse 2D grids for linear, quadratic or cubic representation.

testAmericanSparseBasket3D

Use sparse in 3D grids for linear, quadratic or cubic representation.

testAmericanSparseBasket4D

Use sparse 4D grids for linear, quadratic or cubic representation.

0.5.4 testAmericanOptionCorrel

Same case as before but with correlations between assets. Used to test that the rotation due to the PCA analysis is working correctly.

testAmericCorrel

Check in 2D that

- Local Constant by mesh regression with and without rotation gives the same result,
- Local Linear regression by mesh with and without rotation gives the same result,
- Global regression with and without rotation gives the same result.

0.5.5 testAmericanOptionTree

Simple 1D test case, to test the tree method on American options. An Ornstein–Uhlenbeck process using an average return parameter close to zero is used to be close to the BS model. The OU model is approximated by a trinomial tree.

0.6 testSwingOption

The swing option problem is the generalization of the American option using a Black-Scholes model for the underlying asset: on a set of dates **nStep** (chosen equal to 20 here), we can choose N dates (N equal to three) to exercise the option. At each exercise date t , we get the pay-off $(S_t - K)^+$ where S_t is the value of the underlying asset on the date t . See [26] for a description of the swing problem and backward solving techniques. Due to the classic results on the Snell envelop for the European payoff, the analytical value of this problem is the sum of the payoffs at the last dates N where we can exercise (remember that the value of an American call is the European value). The Markov state of the problem at a given date t is given by the value of the underlying (Markov) and the number of exercises already performed on the date t . This test case can be run in parallel with MPI. In all test cases, we use a **LocalLinearRegression** to evaluate the conditional expectations used during the Dynamic Programming approach.

testSwingOptionInOptimization

After having calculated the analytical solution of this problem,

- a first resolution is provided using the `resolutionSwing` function. For this simple problem, only a regressor is needed to decide whether to exercise on the current date or not.
- a second resolution is provided in the `resolutionSwingContinuation` function using the `Continuation` object (see chapter 4) allowing to store continuation values for a value of the underlying and for a stock level. This example is provided here to show how to use this object on a simple test case. This approach is not optimal here because obtaining the continuation value of an asset value and a stock level (only discrete here) means an unnecessary interpolation on the stock grids (here we choose a `RegularSpaceGrid` to describe the stock level and interpolate linearly between stock grids). In the case of the swing with varying amounts to exercise [26] or the gas storage problem, this object is very useful,
- A final resolution is provided using the general framework described and the `DynamicProgrammingByRegressionDist` function described in the 3.2.2 subsection. Again, the framework is needed for this simple test case, but it shows that it can be used even for some very simple cases.

0.6.1 testSwingOption2D

Here we assume that we have two swing options similar to value and we solve the problem by ignoring that the stocks are independent: this means that we are solving the problem on a two-dimensional grid (for the stocks) instead of twice the same problem on a grid with a stock.

- we start with an evaluation of the solution for a single swing with the `resolutionSwing` function giving a value A .
- then we solve the two-dimensional problem (in stock) giving a value B with our framework with the function `DynamicProgrammingByRegressionDist`.

Next, we check that $B = 2A$.

0.6.2 testSwingOption3

We do the same as before, but the management of three similar swing options is achieved by solving as a three dimensional stock problem.

0.6.3 testSwingOptimSimu / testSwingOptimSimuMpi

This test case takes the problem described in the 0.6 section, resolves it using the framework 3.2.2. Once the optimization using the regression (`LocalLinearRegression` regressor) has been performed, a simulation part is used using the previously calculated Bellman values.

We check that the values obtained in optimization and simulation are close. The two files of test cases (`testSwingOptimSimu/testSwingOptimSimuMpi`) use the two versions of MPI parallelization distributing or not the data on the processors.

0.6.4 `testSwingOptimSimuWithHedge`

The test case takes up the problem described in the section 0.6, solves it using the regression (`LocalLinearRegression` regressor) when calculating the optimal coverage by the conditional tangent method as explained in [47]. After optimization, a simulation part implements the optimal control and the associated optimal hedging policy. We verify:

- That the optimization and simulation values are close
- That the simulated hedge has an average almost equal to zero,
- That the hedged swing simulations yield a reduced standard deviation from the value of the unhedged option obtained by the unhedged simulation.

This test case shows that the multiple regimes introduced in the 3.2.2 framework can be used to calculate and store the optimal hedging policy. This is achieved by creating a dedicated `OptimizeSwingWithHedge` optimizer.

0.6.5 `testSwingOptimSimuND` / `testSwingOptimSimuNDMpi`

The test case takes the problem described in the 0.6 section, suppose we have two similar options to value and we ignore that the options are independent giving a problem to solve with two stocks managed jointly as in subsection 0.6.1. After optimizing the problem using regression (`LocalLinearRegression` regressor) We simulate the optimal control for this two-dimensional problem and check that the optimization and simulation values are close. In `testSwingOptimSimuND` MPI parallelization, if enabled, parallelize only the calculation, while in `testSwingOptimSimuNDMpi` the data is also distributed on processors. In the latter, two options are tested,

- in `testSwingOptionOptim2DSimuDistOneFile` the Bellman values are distributed on the different processors but before being dumped, they are recombined to give a single file for the simulation.
- in `testSwingOptionOptim2DSimuDistMultipleFile` Bellman values are distributed across different processors but each processor dumps its own Bellman Values. During simulation, each processor reads back its own Bellman values.

The same large-dimensional problem may only be achievable with the second approach.

0.7 Gas Storage

0.7.1 testGasStorage / testGasStorageMpi

The model used is a mean return model similar to the one described in [47]. We keep only one factor in equation (8) in [47]. The problem consists in maximizing the gain of a gas storage by the methodology described in [47]. All test cases are composed of three parts:

- an optimization is performed by regression (`LocalLinearRegression` regressor),
- a first simulation of the optimal control using the continuation values stored during the optimization part,
- a second simulation using directly the optimal controls stored during the optimization part.

We check that the three previously calculated values are close.

Using a dynamic programming method, we need to interpolate in the stock grid to get the Bellman values at a stock point. Usually, a simple linear interpolator is used (giving a monotone scheme). As explained in [49], it is possible to use still monotonous higher order schemes. We are testing different interpolators. In all test cases, we use a `LocalLinearRegression` to evaluate conditional expectations. The MPI version makes it possible to test the distribution of data when using parallelization.

testSimpleStorage

We use a classic regular grid with equally spaced points to discretize the gas stock and a linear interpolator to interpolate in the stock.

testSimpleStorageLegendreLinear

We use a Legendre grid with linear interpolation, so the result should be the same as above.

testSimpleStorageLegendreQuadratic

We use a quadratic interpolator for the stock level.

testSimpleStorageLegendreCubic

We use a cubic interpolator for the stock level.

testSimpleStorageSparse

We use a sparse grid interpolator (equivalent to a full grid interpolator because it is a one dimensional problem). We only test the sparse grid with a linear interpolator.

0.7.2 testGasStorageMultiStage / testGasStorageMultiStageMpi

The stochastic model used for uncertainties is the same as in section 0.7.1. The problem consists in maximizing the gain of a gas storage but supposing that on each transition step a deterministic optimization on a given number of periods is achieved. Fictitiously, the number of regimes is set to two during the deterministic optimization on a transition step (while the number of regimes used during the stochastic optimization is equal to one).

All test cases are composed of two parts:

- an optimization is performed by regression (`LocalLinearRegression` regressor),
- a simulation of the optimal control using the continuation values stored during the optimization part.

We check that the two calculated values are close.

In all test cases, we use a `LocalLinearRegression` to evaluate conditional expectations. We use a classic regular grid with equally spaced points to discretize the gas stock and a linear interpolator to interpolate in the stock. The MPI version makes it possible to test the distribution of data when using parallelization.

testSimpleStorageMultiStage

The deterministic problem is solved on 3 periods with a stock constraint constant in time.

testSimpleStorageVaryingCavityMultiStage

The stocks constraints are time varying and we suppose that the deterministic problem is optimized on 2 periods on each transition step.

0.7.3 testGasStorageCut / testGasStorageCutMpi

We take the previous gas storage problem and solve the transition problem without discretizing the command using an LP solver (see section 4.2.2) The test cases are composed of an optimization part followed by a simulation part comparing the results obtained.

testSimpleStorageCut

Test case without MPI distribution of stocks points. A simple Regular grid object is used and conditional cuts are calculated using Local Linear Regressions.

testSimpleStorageCutDist

Test using MPI distribution. In all cases a Local Linear Regressor is used. A file is used to store the conditional cuts.

- A first case uses a Regular grid,
- A second case uses a RegularLegendre grid.

0.7.4 testGasStorageCutGridAdapt1D / testGasStorageCutGridAdapt2D

Using an adapting grid in 1D and 2D (see section 1.4), solve a real and fictitious (2D) gas storage problem without supposing that the solution is concave with respect to the storage level. The problem is solved with cuts gathering domain where the solution is concave (see Continuation object in 4.1.4).

testSimpleStorageMultipleFileCutDist

Test using MPI distribution. A Local Linear Regressor is used for cuts and a Regular grid is used. Bender cuts are stored locally by each processor.

0.7.5 testGasStorageTree/testGasStorageTreeMpi

Optimizing storage for the price of gas modeled by a HJM model approximated by a tree. The grids are simple regular grids. In MPI, Bellman values are either stored in one file or in multiple files.

0.7.6 testGasStorageTreeCut/testGasStorageTreeCutMpi

Gas storage is optimized and simulated using cuts and trees for uncertainties. Gas price modeled by a HJM model approximated by a trinomial tree. The grids are simple regular grids. In MPI, Bellman values are either stored in one file or in multiple files.

0.7.7 testGasStorageKernel

The model used is a mean reverting model similar to the one described in [47]. We keep only one factor in equation (8) in [47]. The problem consists in maximizing the gain of a gas storage by the methodology described in [47]. The specificity here is that a kernel regression method is used.

testSimpleStorageKernel)

Use the linear kernel regression method to solve the Gas Storage problem using the Epanechnikov kernel.

0.7.8 testGasStorageLaplacianLinearKernel

The model used is a mean reverting model similar to the one described in [47]. We keep only one factor in equation (8) in [47]. The problem consists in maximizing the gain of a gas storage by the methodology described in [47]. The specificity here is that a Laplacian kernel regression method is used.

testSimpleStorageLaplacianLinearKernel

Use the linear kernel regression method to solve the Gas Storage problem using the Laplacian kernel using the divide and conquer method.

0.7.9 testGasStorageLaplacianConstKernel

The model used is a mean reverting model similar to the one described in [47]. We keep only one factor in equation (8) in [47]. The problem consists in maximizing the gain of a gas storage by the methodology described in [47]. The specificity here is that a Laplacian kernel regression method is used.

testSimpleStorageLaplacianConstKernel

Use the constant kernel regression method to solve the Gas Storage problem using the Laplacian kernel using the divide and conquer method.

0.7.10 testGasStorageLaplacianGridKernel

The model used is a mean reverting model similar to the one described in [47]. We keep only one factor in equation (8) in [47]. The problem consists in maximizing the gain of a gas storage by the methodology described in [47]. The specificity here is that a Laplacian kernel regression method is used.

testSimpleStorageLaplacianGridKernel

Use the constant kernel regression method to solve the Gas Storage problem using the Laplacian kernel using the fast summation method.

0.7.11 testGasStorageVaryingCavity

The stochastic model is the same as in the section 0.7.1. As before, all test cases are composed of three parts:

- an optimization is performed by regression (`LocalLinearRegression` regressor),
- a first simulation of the optimal control using the continuation values stored during the optimization part,
- a second simulation using directly the optimal controls stored during the optimization part.

We check that the three previously calculated values are close on this test case where the grid describing the gas storage constraint is variable in time. This makes it possible to check the splitting of the grids during the parallelization.

0.7.12 testGasStorageSwitchingCostMpi

The test case is similar to that of the 0.7.1 section (therefore using regression methods): we added an extra cost when switching from one regime to another. The additional cost results in the fact that the Markov state is composed of the asset price, the level of stock and the current regime in which we are (the latter is not present in another test case on the gas storage). This test case shows that our framework solves regime switching problems. As before, all test cases are composed of three parts:

- an optimization is performed by regression (`LocalLinearRegression` regressor),
- a first simulation of the optimal control from the sequence values stored during the optimization part,
- a second simulation using directly the optimal controls stored during the optimization part.

We check that the three previously calculated values are close.

0.7.13 testGasStorageSDDP

Modeling the asset is similar to the other test case. We assume that we have N similar independent storages. So solving the problem with N stocks should yield N times the value of a stock.

- First, the storage value is calculated by dynamic programming giving the value A ,
- Then the SDDP method (chapter 1) is used to value the problem giving the B value. Benders cuts must be made subject to price level.

We check that B is close to NA .

testSimpleStorageSDDP1D

Test the case $N = 1$.

testSimpleStorageSDDP2D

Test the case $N = 2$.

testSimpleStorageSDDP10D

Test the case $N = 10$.

0.7.14 testGasStorageSDDPTree

testSimpleStorageDeterministicCutTree

The volatility is set to zero to obtain a deterministic problem.

- First by Dynamic Programming, the optimal control is calculated and tested in simulation.
- Then backward part of SDDP and forward part are tested using a point grid for storage

testSimpleStorageCutTree

In stochastics, the backward and forward resolution of the SDDP solver with tree are tested using points defined on a grid. Convergence is checked by comparing results from a DP solver with regressions.

testSimpleStorageSDDPTree1D1Step

In stochastics, the global SDDP solver iterating forward and backward is used to value the gas storage. The comparison with dynamic programming methods with regressions is carried out.

0.8 testLake / testLakeMpi

This is the case of a reservoir with inflows following an AR1 model. We can withdraw water from the reservoir (maximum withdrawal rate given) to produce energy by selling it at a given price (taken equal to 1 per unit of volume). We wish to maximize the expected gains obtained by optimal management of the lake. The problem allows to show how some stochastic flows can be taken into account with dynamic programming with regression (`LocalLinearRegression` regressor used).

The test case is composed of three parts:

- an optimization is performed by regression (`LocalLinearRegression` regressor),
- a first simulation of the optimal control from the continuation values stored during the optimization part,
- a second simulation using directly the optimal controls stored during the optimization part.

We check that the three previously calculated values are close.

0.9 testOptionNIGL2

In this test case we assume that the log of an asset value follows a NIG process [3]. We want to price a call option assuming that we use the mean variance criterion using the algorithm developed in chapter 5.

First, an optimization is carried out then in a simulation part, the optimal hedging strategy is tested.

0.10 testDemandSDDP

This test case is the simplest using the SDDP method. We assume that we have a demand according to an AR 1 model

$$D^{n+1} = k(D^n - D) + \sigma_d g + kD,$$

where D is the average demand, σ_d the standard deviation of the demand on a time step, k the average reverting coefficient, $D^0 = D$, and g a centered Gaussian variable on the unit. We have to meet demand by buying energy at a P price. We want to calculate the next expected value

$$\begin{aligned} V &= P\mathbb{E}\left[\sum_{i=0}^N D_i\right] \\ &= (N+1)D_0P \end{aligned}$$

This can be done (artificially) using SDDP.

testDemandSDDP1DDeterministic

It takes $\sigma_d = 0$.

testDemandSDDP1D

It solves the stochastic problem.

0.11 testThermalAsset

Solve a switching problem estimating the value of a thermal asset described in paragraph 4. The two assets follow an HJM model with one factor each. The objective function consists in maximizing the gain in expectation managing the asset that can be switch on or off with some specific minimal time in each regime. A switching cost switching on the asset is also added.

0.12 Reservoir variations with SDDP

0.12.1 testReservoirWithInflowsSDDP

For this SDDP test case, we assume that we have N similar independent reservoirs with inflows given at all times by independent centered Gaussian variables with standard deviation σ_i . We suppose that we must satisfy the dates M a demand given by independent Gaussian variables centered with a standard deviation σ_d . In order to meet the demand, we can buy

water in the amount q_t at a deterministic price S_t or withdraw water from the reservoir at a pace lower than a withdrawal rate. Under the demand constraint, we want to minimize:

$$\mathbb{E} \left[\sum_{i=0}^M q_t S_t \right]$$

Each time we check that the forward and backward methods converge to the same value. Due to the independence of uncertainties, the dimension of the Markov state is equal to N .

testSimpleStorageWithInflowsSDDP1DDeterminist

$\sigma_i = 0$ for inflows and $\sigma_d = 0$. for demand. N taken equal to 1.

testSimpleStorageWithInflowsSDDP2DDeterminist

$\sigma_i = 0$ for inflows and $\sigma_d = 0$. for demand. N taken equal to 2.

testSimpleStorageWithInflowsSDDP5DDeterminist

$\sigma_i = 0$ for inflows and $\sigma_d = 0$. for demand. N taken equal to 5.

testSimpleStorageWithInflowsSDDP1D

$\sigma_i = 0.6$, $\sigma_d = 0.8$ for demand. $N = 1$

testSimpleStorageWithInflowsSDDP2D

$\sigma_i = 0.6$ for inflows, $\sigma_d = 0.8$ for demand. $N = 2$

testSimpleStorageWithInflowsSDDPD

$\sigma_i = 0.6$ for inflows, $\sigma_d = 0.8$ for demand. $N = 5$.

0.12.2 testStorageWithInflowsSDDP

For this SDDP test case, we assume that we have similar independent reservoirs N with inflows following an AR1 model:

$$X^{n+1} = k(X^n - X) + \sigma g + X,$$

with $X^0 = X$, σ the associated standard deviation, g a Gaussian variable centered on the unit.

We suppose that we have to satisfy at M dates a demand following an AR1 process too. In order to satisfy the demand, we can buy some water with quantity q_t at a deterministic price S_t or withdraw water from the reservoir at a pace lower than a withdrawal rate. Under the demand constraint, we want to minimize:

$$\mathbb{E} \left[\sum_{i=0}^M q_t S_t \right]$$

Each time we check that the forward and backward methods converge to the same value. Due to the structure of uncertainties the dimension of the Markov state is equal to $2N + 1$ (N storage, N inflows, and demand).

testSimpleStorageWithInflowsSDDP1DDeterministic

All parameters σ are set to 0. $N = 1$.

testSimpleStorageWithInflowsSDDP2DDeterministic

All parameters σ are set to 0. $N = 2$.

testSimpleStorageWithInflowsSDDP5DDeterministic

All parameters σ are set to 0. $N = 5$.

testSimpleStorageWithInflowsSDDP10DDeterministic

All parameters σ are set to 0. $N = 10$.

testSimpleStorageWithInflowsSDDP1D

$\sigma = 0.3$ for inflows, $\sigma = 0.4$ for demand. $N = 1$.

testSimpleStorageWithInflowsSDDP5D

$\sigma = 0.3$ for inflows, $\sigma = 0.4$ for demand. $N = 5$.

0.12.3 testStorageWithInflowsAndMarketSDDP

This is the same problem as 0.12.2, but the price S_t follow an AR 1 model. We use an SDDP approach to solve this problem. Due to price dependencies, the reduction of SDDP must be effected conditionally on the price level.

testSimpleStorageWithInflowsAndMarketSDDP1DDeterministic

All volatilities set to 0. $N = 1$.

testSimpleStorageWithInflowsAndMarketSDDP2DDeterministic

All volatilities set to 0. $N = 2$.

testSimpleStorageWithInflowsAndMarketSDDP5DDeterministic

All volatilities set to 0. $N = 5$.

testSimpleStorageWithInflowsAndMarketSDDP10DDeterministic

All volatilities set to 0. $N = 10$.

testSimpleStorageWithInflowsAndMarketSDDP1D

$\sigma = 0.3$ for inflows, $\sigma = 0.4$ for demand, $\sigma = 0.6$ for the spot price. $N = 1$.

testSimpleStorageWithInflowsAndMarketSDDP5D

$\sigma = 0.3$ for inflows, $\sigma = 0.4$ for demand, $\sigma = 0.6$ for the spot price. $N = 5$.

0.13 Semi-Lagrangian

0.13.1 testSemiLagrangCase1/testSemiLagrangCase1

Test Semi-Lagrangian deterministic methods for HJB equation. This corresponds to the second test case without control in [49] (2 dimensional test case).

TestSemiLagrang1Lin

Test the Semi-Lagrangian method with the linear interpolator.

TestSemiLagrang1Quad

Test the Semi-Lagrangian method with the quadratic interpolator.

TestSemiLagrang1Cubic

Test the Semi-Lagrangian method with the cubic interpolator.

TestSemiLagrang1SparseQuad

Test the sparse grid interpolator with a quadratic interpolation.

TestSemiLagrang1SparseQuadAdapt

Test the sparse grid interpolator with a quadratic interpolation and some adaptation in the meshing.

0.13.2 testSemiLagrangCase2/testSemiLagrangCase2

Test Semi-Lagrangian deterministic methods for HJB equation. This corresponds to the first case without control in [49] (2 dimensional test case).

TestSemiLagrang2Lin

Test the Semi-Lagrangian method with the linear interpolator.

TestSemiLagrang2Quad

Test the Semi-Lagrangian method with the quadratic interpolator.

TestSemiLagrang2Cubic

Test the Semi-Lagrangian method with the cubic interpolator.

TestSemiLagrang2SparseQuad

Test the sparse grid interpolator with a quadratic interpolation.

0.13.3 testSemiLagrangCase2/testSemiLagrangCase2

Test Semi-Lagrangian deterministic methods for HJB equation. This corresponds to the stochastic target test case 5.3.4 in [49].

TestSemiLagrang3Lin

Test the Semi-Lagrangian method with the linear interpolator.

TestSemiLagrang3Quad

Test the Semi-Lagrangian method with the quadratic interpolator.

TestSemiLagrang3Cubic

Test the Semi-Lagrangian method with the cubic interpolator.

0.14 Non emissive test case

0.14.1 testDPNonEmissive

Solve the problem described in part V by dynamic programming and regression.

- first an optimization is realized,
- the an simulation part permit to test the controls obtained.

0.14.2 testSLNonEmissive

Solve the problem described in part V by the Semi-Lagrangian method.

- first an optimization is realized,
- the an simulation part permit to test the controls obtained.

0.15 Nesting for Non Linear PDE's

0.15.1 Some HJB test

The control problem where $\mathcal{A}$ is the set of adapted integrable processes.

$$dX = 2\sqrt{\theta}\alpha dt + \sqrt{2}dW_t,$$

$$V = \inf_{\alpha \in \mathcal{A}} E[\int_0^T |\alpha_s|^2 dt + g(X_T)]$$

The HJB equation corresponding

$$(-\partial_t u - \mathcal{L}u)(t, x) = f(Du(t, x))$$

$$\mathcal{L}u(t, x) := \mu Du(t, x) + \frac{1}{2} \sigma \sigma^\top : D^2 u(t, x), \quad (31)$$

$$f(z) = -\theta \|z\|_2^2 \quad (32)$$

such that a solution is

$$u(t, x) = -\frac{1}{\theta} \log (\mathbb{E}[e^{-\theta g(x + \sqrt{2}W_{T-t})}]). \quad (33)$$

We use the nesting method with $\mu = 0$, $\sigma = \sqrt{2}I_d$. These test cases are in the `test/c++/unit/branching` directory.

testHJCCConst

In this test case, we use a special resolution function assuming that the parameters of the PDE are constant: this allows us to precompute the inverse of some matrices.

testHJCEXact

We test here the particular case where the SDE can be exactly simulated with a scheme

$$X_{t+dt} = A(t, dt)X_t + B(t, dt) + C(t, dt)g$$

with a g Gaussian-centered unit vector.

testHJBEuler

We use a resolution function assuming that the SDE is discretized by an Euler scheme.

0.15.2 Some Toy example: testUD2UTou

We want to solve:

$$(-\partial_t u - \mathcal{L}u)(t, x) = f(u, Du(t, x), D^2 u(t, x))$$

with

$$\begin{aligned}\mu &= \frac{\mu_0}{d} \mathbb{I}_d, \\ \sigma &= \frac{\sigma_0}{\sqrt{d}} \mathbf{I}_d, \\ f(t, x, y, z, \theta) &= \cos(\sum_{i=1}^d x_i) \left(\alpha + \frac{1}{2} \sigma_0^2 \right) e^{\alpha(T-t)} + \sin(\sum_{i=1}^d x_i) \mu_0 e^{\alpha(T-t)} + a \sqrt{d} \cos(\sum_{i=1}^d x_i)^2 e^{2\alpha(T-t)} \\ &\quad + \frac{a}{\sqrt{d}} (-e^{2\alpha(T-t)}) \vee (e^{2\alpha(T-t)} \wedge (y \sum_{i=1}^d \theta_{i,i})),\end{aligned}$$

with a solution

$$u(t, x) = e^{\alpha(T-t)} \cos\left(\sum_{i=1}^d x_i\right)$$

0.15.3 Some Portfolio optimization

We assume that we dispose of $d = 4$ securities all of them being defined by a Heston model:

$$\begin{aligned}dS_t^i &= \mu^i S_t^i dt + \sqrt{Y_t^i} S_t^i dW_t^{(2i-1)} \\ dY_t^i &= k^i (m^i - Y_t^i) dt + c^i \sqrt{Y_t^i} dW_t^{(2i)},\end{aligned}$$

where $W = (W^{(1)}, \dots, W^{(2d)})$ is a Brownian motion in $\mathbb{R}^{2d}$.

The non-risky asset S^0 has a 0 return so $dS_t^0 = 0$, $t \in [0, 1]$.

The investor chooses an adapted process $\{\kappa_t, t \in [0, T]\}$ with values in $\mathbb{R}^n$, where κ_t^i is the amount he decides to invest into asset i .

The portfolio dynamic is given by:

$$dX_t^\kappa = \kappa_t \cdot \frac{dS_t}{S_t} + (X_t^\kappa - \kappa_t \cdot \mathbf{1}) \frac{dS_t^0}{S_t^0} = \kappa_t \cdot \frac{dS_t}{S_t}.$$

Let $\mathcal{A}$ be the collection of all adapted processes κ with values in $\mathbb{R}^d$ and which are integrable with respect to S . Given an absolute risk aversion coefficient $\eta > 0$, the portfolio optimization problem is defined by:

$$v_0 := \sup_{\kappa \in \mathcal{A}} \mathbb{E} [-\exp(-\eta X_T^\kappa)]. \quad (34)$$

The problem doesn't depend on the s^i . As in [54], we can guess that the solution can be expressed as

$$v(t, x, y^1, \dots, y^d) = e^{-\eta x} u(y^1, \dots, y^d)$$

and using Feynman Kac it is easy to see that a general solution can be written

$$v(t, x, y) = -e^{-\eta x} \mathbb{E} \left[\prod_{i=1}^d \exp \left(-\frac{1}{2} \int_t^T \frac{(\mu^i)^2}{\tilde{Y}_s^i} ds \right) \right] \quad (35)$$

with

$$\tilde{Y}_t^i = y^i \quad \text{and} \quad d\tilde{Y}_t^i = k^i (m^i - \tilde{Y}_t^i) dt + c^i \sqrt{\tilde{Y}_t^i} dW_t^i,$$

where y^i is the initial volatility value on the date 0 for the asset i .

We assume, in our example, that all the assets have the same parameters which are equal to the parameters taken in the two-dimensional case. We also assume that the initial conditions are the same as before.

By choosing $\bar{\sigma} > 0$, we can write the problem as the equation (23) in dimension $d + 1$ where

$$\mu = (0, k^1(m^1 - y^1), \dots, k^d(m^d - y^d))^\top, \quad \sigma = \begin{pmatrix} \bar{\sigma} & 0 & \dots & \dots & 0 \\ 0 & c\sqrt{m^1} & 0 & \dots & 0 \\ 0 & \dots & \ddots & \dots & 0 \\ 0 & \dots & \dots & \ddots & 0 \\ 0 & \dots & \dots & 0 & c\sqrt{m^d} \end{pmatrix}$$

always with the same terminal condition

$$g(x) = -e^{-\eta x}$$

and

$$f(x, y, z, \theta) = -\frac{1}{2}\bar{\sigma}^2\theta_{11} + \frac{1}{2}\sum_{i=1}^d (c^i)^2((y^i)^2 - m^i)\theta_{i+1,i+1} - \sum_{i=1}^d \frac{\mu^i z_1}{2y^i\theta_{11}}. \quad (36)$$

In order to have f Lipschitz, we truncate the control limiting the amount invested by taking

$$f_M(y, z, \theta) = -\frac{1}{2}\bar{\sigma}^2\theta_{11} + \frac{1}{2}\sum_{i=1}^d (c^i)^2((y^i)^2 - m^i)\theta_{2,2} + \sup_{\substack{\eta = (\eta^1, \dots, \eta^d) \\ 0 \leq \eta^i \leq M, i = 1, d}} \sum_{i=1}^d \left(\frac{1}{2}(\eta^i)^2 y^i \theta_{11} + (\eta^i) \mu^i z_1 \right).$$

testPortfolioExact

The Ornstein–Uhlenbeck process used as a driving process is simulated exactly.

testPortfolioEuler

The Ornstein–Uhlenbeck process used as a driving process is simulated using an Euler scheme.

0.16 Automatic Differentiation

0.16.1 testStorageAutoDiff

This case optimizes a gas storage facility with realistic injection and withdrawal capacities, as well as a tunnel constraint that the gas stock must satisfy.

The business object is given below:

```

1 // Copyright (C) 2021 EDF
2 // All Rights Reserved
3 // This code is published under the GNU Lesser General Public License (GNU LGPL)
4 #ifndef OPTIMIZE_STORAGE_H
5 #define OPTIMIZE_STORAGE_H
6 #include <memory>
7 #include <Eigen/Dense>
8 #include <StOpt/dp/OptimizerDPBase.h>
9 #include <StOpt/core/Utils/StateWithStocks.h>
10 #include <StOpt/regression/BaseRegression.h>
11 #include <StOpt/core/grids/Interpolator.h>
12 #include <StOpt/core/Utils/constant.h>
13 #include <StOpt/regression/ContinuationValue.h>
14 #include "StOpt/regression/GridAndRegressedValue.h"
15 #include "StOpt/regression/GridAndRegressedValueAutoDiff.h"
16 #include "test/c++/tools/Utils/Capa.h"
17
18
19 /** \file OptimizeStorage.h
20  * Optimize storage commands
21  * Injection and withdrawal are given by time step
22  * \author Xavier Warin
23  */
24 template<class Simulator>
25 class OptimizeStorage : public StOpt::OptimizerDPBase
26 {
27 private:
28
29     const Capa &m_injectionVolume ; ///< Injection volume capacity
30     const Capa &m_withDrawalVolume ; ///< Withdrawal volume capacity
31     const double m_alphaADiff; // autodiff regularization
32
33     /// store the simulator
34     std::shared_ptr<Simulator> m_simulator;
35
36 public:
37
38     /// \brief Constructor
39     /// \param p_injectionVolume Injection volume capacity
40     /// \param p_withDrawalVolume Withdrawal volume capacity
41     OptimizeStorage(const Capa &p_injectionVolume, const Capa &p_withDrawalVolume, double
42         p_alphaADiff = 20.): m_injectionVolume(p_injectionVolume), m_withDrawalVolume(
43         p_withDrawalVolume), m_alphaADiff(p_alphaADiff) {}
44
45     /// \brief define the diffusion cone for parallelism (not needed if not parallelism
46     /// required )
47     /// \return returns in each dimension the min max values in the stock that can be
48     /// reached from the grid p_gridByProcessor for each regime
49     std::vector< std::array< double, 2> > getCone(const std::vector< std::array< double,
50         2> > &) const
51     {
52         std::vector< std::array< double, 2> > extrGrid(1);
53         extrGrid[0][0] = - StOpt::infy;
54         extrGrid[0][1] = StOpt::infy;
55         return extrGrid;
56     }
57
58     /// \brief defines the dimension to split for MPI parallelism (not needed if no
59     /// parallelism )
60     /// For each dimension return true is the direction can be split
61     Eigen::Array< bool, Eigen::Dynamic, 1> getDimensionToSplit() const
62     {
63         Eigen::Array< bool, Eigen::Dynamic, 1> bDim = Eigen::Array< bool, Eigen::Dynamic,
64             1>::Constant(1, true);
65         return bDim ;
66     }
67 }

```

```

62  /// \brief defines a step in optimization
63  /// \param p_grid      grid at arrival step after command
64  /// \param p_stock     coordinate of the stock point to treat at current time step
65  /// \param p_condEsp   continuation values for each regime (permitting to interpolate
66  ///                    in stocks the regresses values)
67  /// \param p_phiIn     for each regime gives the solution calculated at the previous
68  ///                    step ( next time step by Dynamic Programming resolution)
69  /// \return a pair :
70  ///                    - for each regimes (column) gives the solution for each particle (row)
71  ///                    - for each control (column) gives the optimal control for each
72  ///                    particle (rows)
73  std::pair< Eigen::ArrayXXd, Eigen::ArrayXXd> stepOptimize(const std::shared_ptr<
74  StOpt::SpaceGrid> &p_grid, const Eigen::ArrayXd &p_stock,
75  const std::vector<StOpt::ContinuationValue> &p_condEsp,
76  const std::vector< std::shared_ptr< Eigen::ArrayXXd > > &p_phiIn) const
77  {
78  // std::shared_ptr<MeanRevertingNDSimulator> ptSimul= std::static_pointer_cast<
79  // MeanRevertingNDSimulator>( m_simulator);
80  int nbSimul = m_simulator->getNbSimul();
81  std::pair< Eigen::ArrayXXd, Eigen::ArrayXXd> solutionAndControl;
82  solutionAndControl.first.resize(nbSimul, 1);
83  solutionAndControl.second.resize(nbSimul, 1);
84  // get back spot
85  Eigen::ArrayXd spot = m_simulator->getSpot();
86  // get current time Step number : here simulator
87  int iStep = static_cast<int>(m_simulator->getCurrentStep() + StOpt::tiny);
88  // injection Max
89  double maxStorage = p_grid->getExtremeValues()[0][1];
90  double injectionRate = m_injectionVolume.getValue(iStep, p_stock(0));
91  double injectionMax = std::min(maxStorage - p_stock(0) - StOpt::tiny, injectionRate
92  );
93  // withdrawl max (algebraic so negative)
94  double minStorage = p_grid->getExtremeValues()[0][0];
95  double withdrawalRate = m_withDrawalVolume.getValue(iStep, p_stock(0));
96  double withdrawalMax = std::min(p_stock(0) - minStorage - StOpt::tiny,
97  withdrawalRate);
98
99  // 2 comparisons to avoid withdrawalMax= withdrawalMax =0
100 double smallDec = StOpt::tiny * std::max(std::max(std::fabs(withdrawalMax), std::
101 fabs(injectionMax)), 2.);
102 // test if point is possible
103 if (-withdrawalMax > (injectionMax + smallDec))
104 {
105     solutionAndControl.first.col(0) = Eigen::ArrayXd::Constant(nbSimul, - StOpt::
106 infty);
107     solutionAndControl.second.col(0) = Eigen::ArrayXd::Constant(nbSimul, - StOpt::
108 infty);
109     std::cout << " IMPOS iStep " << iStep << "Pt " << p_stock(0) << " WITH " <<
110 withdrawalMax << " injectionMax " << injectionMax << " diff " <<
111 withdrawalMax + injectionMax << " minStorage Next step " << minStorage << "
112 max storage Next step " << maxStorage << " injectionRate " <<
113 injectionRate << " withdrawalRate " << withdrawalRate << std::endl ;
114     return solutionAndControl;
115 }
116
117 // cash, optimal command, condi
118 Eigen::ArrayXd cashOpt = Eigen::ArrayXd::Constant(nbSimul, - StOpt::infty);
119 Eigen::ArrayXd commandOpt = Eigen::ArrayXd::Constant(nbSimul, - StOpt::infty);
120 Eigen::ArrayXd condOpt = Eigen::ArrayXd::Constant(nbSimul, - StOpt::infty);
121 // test max withdrawal command
122 {
123     double command = -withdrawalMax;
124     Eigen::ArrayXd nextStock = p_stock + command;
125     // interpolator next pt
126     std::shared_ptr<StOpt::Interpolator<double>> interpolatorNextStock = p_grid->
127 createInterpolator(nextStock);
128     // interpolate stock
129     Eigen::ArrayXd cashNext = interpolatorNextStock->applyVec(*p_phiIn[0]);

```

```

116         // conditional expectation
117         Eigen::ArrayXd condExp = - command * spot + p_condEsp[0].getAllSimulations(*
            interpolatorNextStock);
118         for (int is = 0; is < nbSimul; ++is)
119         {
120             if (condExp(is) > condOpt(is))
121             {
122                 condOpt(is) = condExp(is);
123                 cashOpt(is) = - command * spot(is) + cashNext(is);
124                 commandOpt(is) = command;
125             }
126         }
127     }
128     // test max injection command
129     {
130         double command = injectionMax;
131         Eigen::ArrayXd nextStock = p_stock + command;
132         // interpolator next pt
133         std::shared_ptr<StOpt::Interpolator<double>> interpolatorNextStock = p_grid->
            createInterpolator(nextStock);
134         // interpolate stock
135         Eigen::ArrayXd cashNext = interpolatorNextStock->applyVec(*p_phiIn[0]);
136         // conditional expectation
137         Eigen::ArrayXd condExp = - command * spot + p_condEsp[0].getAllSimulations(*
            interpolatorNextStock);
138         for (int is = 0; is < nbSimul; ++is)
139         {
140             if (condExp(is) > condOpt(is))
141             {
142                 condOpt(is) = condExp(is);
143                 cashOpt(is) = - command * spot(is) + cashNext(is);
144                 commandOpt(is) = command;
145             }
146         }
147     }
148     double stockMaxReached = p_stock(0) + injectionMax;
149     double stockMinReached = p_stock(0) - withdrawalMax;
150     // test interior points
151     Eigen::ArrayXd grid1D = *(std::static_pointer_cast<StOpt::FullGrid>(p_grid)->
        get1DGrids()[0]);
152     int iMin = 0;
153     while (grid1D(iMin) <= stockMinReached)
154     {
155         iMin += 1;
156         if (iMin == (grid1D.size() - 1))
157             break;
158     }
159     int iMax = grid1D.size() - 1;
160     while (grid1D(iMax) >= stockMaxReached)
161     {
162         iMax -= 1;
163         if (iMax == 0)
164             break;
165     }
166     if (iMin <= iMax)
167     {
168         for (int ipt = iMin; ipt <= iMax; ++ipt)
169         {
170             if (std::fabs(grid1D(ipt)) > StOpt::tiny)
171             {
172                 // command
173                 double command = grid1D(ipt) - p_stock(0);
174                 Eigen::ArrayXd nextStock = p_stock + command;
175                 // interpolator next pt
176                 std::shared_ptr<StOpt::Interpolator<double>> interpolatorNextStock =
                    p_grid->createInterpolator(nextStock);
177                 // interpolate stock
178                 Eigen::ArrayXd cashNext = interpolatorNextStock->applyVec(*p_phiIn[0]);
179                 // conditional expectation

```

```

180         Eigen::ArrayXd condExp = - command * spot + p_condEsp[0].
            getAllSimulations(*interpolatorNextStock);
181         for (int is = 0; is < nbSimul; ++is)
182         {
183             if (condExp(is) > condOpt(is))
184             {
185                 condOpt(is) = condExp(is);
186                 cashOpt(is) = - command * spot(is) + cashNext(is);
187                 commandOpt(is) = command;
188             }
189         }
190     }
191 }
192 }
193 for (int is = 0; is < nbSimul; ++is)
194 {
195     solutionAndControl.first(is, 0) = cashOpt(is);
196     solutionAndControl.second(is, 0) = commandOpt(is);
197 }
198 return solutionAndControl;
199 }
200
201 /// \brief defines a step in simulation
202 /// This implementation is for test and example purpose
203 /// \param p_grid          grid at arrival step after command
204 /// \param p_continuation  defines the continuation operator for each regime
205 /// \param p_state         defines the state value (modified)
206 /// \param p_phiInOut      defines the value functions (modified): size number of
207                          functions to follow
208 void stepSimulate(const std::shared_ptr< StOpt::SpaceGrid> &p_grid, const std::vector
209 < StOpt::GridAndRegressedValue > &p_continuation,
210 StOpt::StateWithStocks<double> &p_state, Eigen::Ref<Eigen::ArrayXd>
211 p_phiInOut) const
212 {
213     //std::shared_ptr<MeanRevertingNDSimulator> ptSimul= std::static_pointer_cast<
214     MeanRevertingNDSimulator>( m_simulator);
215     // optimal stock attained
216     Eigen::ArrayXd ptStockCur = p_state.getPtStock();
217     Eigen::ArrayXd ptStockMax(ptStockCur);
218     // spot price
219     double spot = m_simulator->getSpotUncertainties(p_state.getStochasticRealization())
220     ;
221     // std::cout << " SimReop spot " << spot << " OU " << p_state.
222     getStochasticRealization().transpose() << std::endl ;
223     // size of the stock
224     double minStorage = p_grid->getExtremeValues()[0][0];
225     double maxStorage = p_grid->getExtremeValues()[0][1];
226     // get current time Step number : here simulator
227     int iStep = static_cast<int>(m_simulator->getCurrentStep() + StOpt::tiny);
228     // if injection
229     double injectionRate = m_injectionVolume.getValue(iStep, ptStockCur(0));
230     double injectionMax = std::min(maxStorage - p_state.getPtStock()(0) - StOpt::tiny,
231 injectionRate);
232     double withdrawalRate = m_withDrawalVolume.getValue(iStep, ptStockCur(0));
233     double withdrawalMax = std::min(p_state.getPtStock()(0) - minStorage - StOpt::tiny,
234 withdrawalRate);
235     double cashOpt = -1e30;
236     double commandOpt = -1e30;
237     double condOpt = -1e30;
238     // test max withdrawal
239     {
240         double command = -withdrawalMax;
241         Eigen::ArrayXd nextStock = ptStockCur + command;
242         double gainLoc = - command * spot;
243         double continuation = p_continuation[0].getValue(nextStock, p_state.
244 getStochasticRealization());
245         double espCond = gainLoc + continuation ;
246         if (espCond > condOpt)
247         {

```

```

239         condOpt = espCond;
240         cashOpt = gainLoc ;
241         commandOpt = command;
242     }
243 }
244 {
245     double command = injectionMax;
246     Eigen::ArrayXd nextStock = ptStockCur + command;
247     double gainLoc = - command * spot;
248     double continuation = p_continuation[0].getValue(nextStock, p_state.
        getStochasticRealization());
249     double espCond = gainLoc + continuation ;
250     if (espCond > condOpt)
251     {
252         condOpt = espCond;
253         cashOpt = gainLoc ;
254         commandOpt = command;
255     }
256 }
257 double stockMaxReached = ptStockCur(0) + injectionMax;
258 double stockMinReached = ptStockCur(0) - withdrawalMax;
259 Eigen::ArrayXd grid1D = *(std::static_pointer_cast<StOpt::FullGrid>(p_grid)->
    get1DGrids()[0]);
260
261 int iMin = 0;
262 while (grid1D(iMin) <= stockMinReached)
263 {
264     iMin += 1;
265     if (iMin == (grid1D.size() - 1))
266         break;
267 }
268 int iMax = grid1D.size() - 1;
269 while (grid1D(iMax) >= stockMaxReached)
270 {
271     iMax -= 1;
272     if (iMax == 0)
273         break;
274 }
275 if (iMin <= iMax)
276 {
277     for (int ipt = iMin; ipt <= iMax; ++ipt)
278     {
279         if (std::fabs(grid1D(ipt)) > StOpt::tiny)
280         {
281             // command
282             double command = grid1D(ipt) - ptStockCur(0);
283             Eigen::ArrayXd nextStock = ptStockCur + command;
284             double gainLoc = - command * spot;
285             double continuation = p_continuation[0].getValue(nextStock, p_state.
                getStochasticRealization());
286             double espCond = gainLoc + continuation ;
287             if (espCond > condOpt)
288             {
289                 condOpt = espCond;
290                 cashOpt = gainLoc ;
291                 commandOpt = command;
292             }
293         }
294     }
295 }
296 // for return
297 p_state.setPtStock(ptStockCur + commandOpt);
298 p_phiInOut(0) += cashOpt ;
299 p_phiInOut(1) = commandOpt ;
300 p_phiInOut(2) = -commandOpt * spot;
301 }
302
303 /// \brief Defines a step in simulation using interpolation in controls
304 /// \param p_grid      grid at arrival step after command

```

```

305 /// \param p_control      defines the controls
306 /// \param p_state        defines the state value (modified)
307 /// \param p_phiInOut     defines the value function (modified): size number of
                          functions to follow
308 virtual void stepSimulateControl(const std::shared_ptr< StOpt::SpaceGrid> &p_grid,
                          const std::vector< StOpt::GridAndRegressedValue > &p_control,
309                               StOpt::StateWithStocks<double> &p_state,
310                               Eigen::Ref<Eigen::ArrayXd> p_phiInOut) const
311 {
312     // cast
313     //std::shared_ptr<MeanRevertingNDSimulator> ptSimul= std::static_pointer_cast<
                          MeanRevertingNDSimulator>( m_simulator);
314     Eigen::ArrayXd ptStock = p_state.getPtStock();
315     // spot price
316     double spotPrice = m_simulator->getSpotUncertainties(p_state.
                          getStochasticRealization());
317     // std::cout << " SimCont spot " << spotPrice << " OU " << p_state.
                          getStochasticRealization().transpose() << std::endl ;
318     // optimal control
319     double control = p_control[0].getValue(p_state.getPtStock(), p_state.
                          getStochasticRealization());
320     double maxStorage = p_grid->getExtremeValues()[0][1];
321     double minStorage = p_grid->getExtremeValues()[0][0];
322     control = std::max(std::min(maxStorage - ptStock(0), control), minStorage - ptStock
                          (0));
323     p_phiInOut(0) -= control * spotPrice;
324     ptStock(0) += control ;
325     ptStock(0) = std::min(std::max(minStorage, ptStock(0)), maxStorage); // avoid
                          rounding error
326     p_state.setPtStock(ptStock);
327     p_phiInOut(1) = control ;
328     p_phiInOut(2) = -control * spotPrice;
329     double tangent = m_simulator->getTangent(p_state.getStochasticRealization());
330     p_phiInOut(3) = -control * tangent;
331     p_phiInOut(4) = -control * spotPrice * (m_simulator->getTrendDeriv(0) + p_state.
                          getStochasticRealization()(0));
332     p_phiInOut(5) = -control * spotPrice * (m_simulator->getTrendDeriv(1) + p_state.
                          getStochasticRealization()(1));
333 }
334
335
336
337 template<typename T, typename Expr>
338 void stepSimulateControlAutoDiff(const std::shared_ptr< StOpt::FullGrid> &p_grid,
                          const std::vector< StOpt::GridAndRegressedValueAutoDiff<T> > &p_control,
339                               StOpt::StateWithStocks<T> &p_state,
340                               Expr &&p_phiInOut)
341 {
342     // foward reference to temporary
343     auto &&phiInOut = p_phiInOut;
344     // // cast
345     // std::shared_ptr<MeanRevertingNDSimulatorT<T>> ptSimul= std::static_pointer_cast
                          <MeanRevertingNDSimulatorT<T>>( m_simulator);
346     // Templated code using the autodiff type
347     Eigen::Array<T, Eigen::Dynamic, 1> ptStock = p_state.getPtStock();
348     // spot price
349     T spotPrice = m_simulator->getSpotUncertaintiesAD(p_state.getStochasticRealization
                          ());
350     // std::cout << " SimContAD spot " << spotPrice << " OU " << p_state.
                          getStochasticRealization().transpose() << std::endl ;
351     // optimal control
352     T control = p_control[0].getValueAutoDiff(p_state.getPtStock(), p_state.
                          getStochasticRealization());
353     T maxStorage = T(p_grid->getExtremeValues()[0][1]);
354     T minStorage = T(p_grid->getExtremeValues()[0][0]);
355     T upperControl = maxStorage - ptStock(0);
356     T lowerControl = minStorage - ptStock(0);
357     using StOpt::toDouble;
358     if (toDouble(control) > toDouble(upperControl))

```

```

359         control = upperControl;
360         if (toDouble(control) < toDouble(lowerControl))
361             control = lowerControl;
362         phiInOut(0) -= control * spotPrice;
363         ptStock(0) += control ;
364         p_state.setPtStock(ptStock);
365         p_phiInOut(1) = control ;
366         p_phiInOut(2) = -control * spotPrice;
367     }
368
369
370
371     ///\brief store the simulator
372     inline void setSimulator(const std::shared_ptr<StOpt::SimulatorDPBase> &p_simulator)
373     {
374         m_simulator = std::static_pointer_cast<Simulator>(p_simulator) ;
375     }
376
377     /// \brief get the simulator back
378     inline std::shared_ptr< StOpt::SimulatorDPBase > getSimulator() const
379     {
380         return m_simulator ;
381     }
382
383
384     /// \brief get size of the function to follow in simulation
385     inline int getSimuFuncSize() const
386     {
387         return 6;
388     }
389
390     /// \brief get number of regimes
391     inline int getNbRegime() const
392     {
393         return 1;
394     }
395
396     /// \brief number of controls
397     inline int getNbControl() const
398     {
399         return 1;
400     }
401 };
402 #endif

```

Notice that the following methods are available:

- **stepOptimize** is used during the optimization phase to compute the optimal control for a given stock level at a given date. The interpolated optimal control and continuation values are calculated once all grid points have been processed for that date.
- **stepSimulate** is used during the simulation phase, relying on the continuation values stored during optimization to compute the optimal control by locally optimizing it using these continuation values.
- **stepSimulateControl** is used during simulation when one wants to directly interpolate the control without relying on continuation values.
- **stepSimulateControlAutoDiff** is used during simulation with interpolated controls to compute the optimal control using a type similar to **double**, which stores the data required for automatic differentiation.

```

1   int seedOpt = 123456;
2
3   auto simulatorOpt =
4       std::make_shared<MeanRevertingNDSimulator>(
5           fut,
6           sig,
7           mr,
8           correl,
9           nbStep,
10          nbSimOpt,
11          bForward,
12          baseNameForSim,
13  #ifdef USE_MPI
14          world,
15  #endif
16          seedOpt
17      );
18
19   auto optimizer =
20       std::make_shared<OptimizeStorage<MeanRevertingNDSimulator>>(
21           theInjCapacity,
22           theWithdrawalCapacity
23      );
24
25   optimizer->setSimulator(simulatorOpt);
26
27   DynamicProgrammingStorage(
28       tunnel,
29       optimizer,
30       regressor,
31       initStorageVolume,
32       baseBellmanAndCont
33  #ifdef USE_MPI
34      , world
35  #endif
36  );
37
38   // forward
39   bForward = true;
40   // // Simulation control + delta classique
41   int seedControl = 7891;
42
43   auto simulatorSimControl =
44       std::make_shared<MeanRevertingNDSimulator>(
45           fut,
46           sig,
47           mr,
48           correl,
49           nbStep,
50           nbSimSim,
51           bForward,
52           baseNameForSim,
53  #ifdef USE_MPI
54           world,
55  #endif
56           seedControl
57      );
58
59   optimizer->setSimulator(simulatorSimControl);
60
61   auto resControl =
62       simulatePriceControlUsingControlStorageSensi(
63           tunnel,
64           optimizer,
65           initStorageVolume,
66           baseBellmanAndCont
67  #ifdef USE_MPI
68           , world

```

```

69 #endif
70 );
71 std::cout << " Value with control " << std::get<2>(resControl) << std::endl ;
72 std::cout << " Tangent " << std::get<3>(resControl).transpose() << std::endl;
73 std::cout << " Vol deriv " << std::get<4>(resControl).transpose() << std::endl;
74
75 /// Autodiff Adept
76 #ifdef ADEPT
77
78 auto sensA =
79     calculateSensiUsingControlAdept(
80         fut,
81         sig,
82         mr,
83         correl,
84         nbStep,
85         nbSimSim,
86         batchSize,
87         baseNameForSim,
88         seedControl,
89         theInjCapacity,
90         theWithdrawalCapacity,
91         tunnel,
92         initStorageVolume,
93         baseBellmanAndCont
94     );
95 cout << " ADEPT " << endl ;
96 cout << "Grad with autodiff delta = " << std::get<0>(sensA).transpose() << endl;
97 cout << "Grad with autodiff vega = " << std::get<1>(sensA).transpose() << endl;
98 cout << "Grad with autodiff mean reverting" << std::get<2>(sensA).transpose() <<
    endl;
99 cout << "Grad with autodiff correl" << std::get<3>(sensA) << endl;
100 #endif
101 /// Autodiff stan math
102 #ifdef STANMATH
103
104 auto sensB =
105     calculateSensiUsingControlStan(
106         fut,
107         sig,
108         mr,
109         correl,
110         nbStep,
111         nbSimSim,
112         batchSize,
113         baseNameForSim,
114         seedControl,
115         theInjCapacity,
116         theWithdrawalCapacity,
117         tunnel,
118         initStorageVolume,
119         baseBellmanAndCont
120     );
121 cout << "STAN MATH" << endl ;
122 cout << "Grad with autodiff delta = " << std::get<0>(sensB).transpose() << endl;
123 cout << "Grad with autodiff vega = " << std::get<1>(sensB).transpose() << endl;
124 cout << "Grad with autodiff mean reverting" << std::get<2>(sensB).transpose() <<
    endl;
125 cout << "Grad with autodiff correl" << std::get<3>(sensB) << endl;
126 #endif

```

The case is divided into four parts:

- The `DynamicProgrammingStorage` function (line 26) optimizes the storage using a two-factor model and a business optimizer (therefore using the `stepOptimize` method).
- `simulatePriceControlUsingControlStorageSensi` simulates the storage using Monte

Carlo, returning the simulated value and sensitivities with respect to futures prices and volatilities, computed via manual differentiation.

- `calculateSensiUsingControlAdept` (line 78) uses the Adept library to perform Monte Carlo simulation of the asset, returning both the value and sensitivities computed via automatic differentiation.
- `calculateSensiUsingControlStan` (line 104) uses the Stan Math library to perform the same calculations.

```

1  using adept::adouble;
2
3  if (p_batchSize == 0)
4      throw std::runtime_error("p_batchSize must be strictly positive");
5
6  if (p_nbSimul == 0)
7      throw std::runtime_error("p_nbSimul must be strictly positive");
8
9  cpu_timer timer1;
10 timer1.start();
11
12 const size_t nbBatch =
13     (p_nbSimul + p_batchSize - 1) / p_batchSize;
14
15 std::vector<ArrayXd> deltaBatch(
16     nbBatch,
17     ArrayXd::Zero(p_future.size())
18 );
19
20 std::vector<ArrayXd> vegaBatch(
21     nbBatch,
22     ArrayXd::Zero(p_sig.size())
23 );
24 std::vector<ArrayXd> mrSenBatch(
25     nbBatch,
26     ArrayXd::Zero(p_a.size())
27 );
28 std::vector<ArrayXXd> correlSenBatch(
29     nbBatch,
30     ArrayXXd::Zero(p_correl.rows(), p_correl.cols())
31 );
32
33 std::vector<double> valueBatch(nbBatch, 0.0);
34 std::vector<size_t> nbSimulPerBatch(nbBatch, 0);
35
36 for (size_t ibatch = 0; ibatch < nbBatch; ++ibatch)
37 {
38     const size_t firstSim = ibatch * p_batchSize;
39
40     nbSimulPerBatch[ibatch] =
41         std::min(p_batchSize, p_nbSimul - firstSim);
42 }
43
44 std::exception_ptr firstException = nullptr;
45 #ifndef _OPENMP
46     #pragma omp parallel for schedule(dynamic)
47 #endif
48 for (long long ibatchLL = 0; ibatchLL < static_cast<long long>(nbBatch); ++ibatchLL)
49 {
50     const size_t ibatch = static_cast<size_t>(ibatchLL);
51
52     try
53     {
54         const size_t nbSimulBatch = nbSimulPerBatch[ibatch];
55     }

```

```

56     if (nbSimulBatch == 0)
57         continue;
58
59     // Stack Adept
60     adept::Stack stack;
61
62     // Local to the batch
63     Array<adouble, Eigen::Dynamic, 1> futAdept = p_future.cast<adouble>();
64     Array<adouble, Eigen::Dynamic, 1> sigAdept = p_sig.cast<adouble>();
65     Array<adouble, Dynamic, 1> mrAdept = p_a.cast<adouble>();
66     Matrix<adouble, Dynamic, Dynamic> correlAdept = p_correl.cast<adouble>();
67
68     stack.new_recording();
69
70     // Seed depends only on p_seed and ibatch,
71     // and independant of the thread number
72     const int seedBatch =
73         makeSeedForBatch(p_seed, ibatch);
74
75     // useless as the simulator does not dump in forward mode.
76     const string nameArchBatch =
77         p_nameArch + "_batch_" + std::to_string(ibatch);
78
79     std::shared_ptr<SimulatorDPBase> simulator =
80         std::make_shared<MeanRevertingNDSimulatorT<adouble>>>(
81             p_future,
82             p_sig,
83             p_a,
84             p_correl,
85             futAdept,
86             sigAdept,
87             mrAdept,
88             correlAdept,
89             p_nbStep,
90             nbSimulBatch,
91             true,
92             nameArchBatch,
93             seedBatch
94         );
95
96     std::shared_ptr<OptimizeStorage<MeanRevertingNDSimulatorT<adouble>>> optimizer
97         =
98         std::make_shared<OptimizeStorage<MeanRevertingNDSimulatorT<adouble>>>(
99             p_injectionVolume,
100             p_withDrawalVolume
101         );
102
103     optimizer->setSimulator(simulator);
104
105     adouble value =
106         simulatePriceControlUsingControlStorageAutoDiff<OptimizeStorage<
107             MeanRevertingNDSimulatorT<adouble>>, adouble>(
108             p_tunnel,
109             optimizer,
110             p_pointStock,
111             p_fileForBellmanAndCont
112         );
113
114     value.set_gradient(1.0);
115     stack.reverse();
116
117     for (int i = 0; i < p_future.size(); ++i)
118         deltaBatch[ibatch][i] = futAdept(i).get_gradient();
119
120     for (int i = 0; i < p_sig.size(); ++i)
121         vegaBatch[ibatch][i] = sigAdept(i).get_gradient();
122     for (int i = 0; i < p_a.size(); ++i)
123         mrSenBatch[ibatch][i] = mrAdept(i).get_gradient();
124     for (int j = 0; j < p_correl.cols(); ++j)

```

```

123         for (int i = 0; i < p_correl.rows(); ++i)
124             correlSenBatch[ibatch](i, j) = correlAdept(i, j).get_gradient();
125
126         valueBatch[ibatch] = value.value();
127
128     }
129     catch (...)
130     {
131     #ifdef _OPENMP
132         #pragma omp critical
133     #endif
134         {
135             if (!firstException)
136                 firstException = std::current_exception();
137         }
138     }
139 }
140 }
141
142 if (firstException)
143     std::rethrow_exception(firstException);
144
145 ArrayXd delta = ArrayXd::Zero(p_future.size());
146 ArrayXd vega = ArrayXd::Zero(p_sig.size());
147 ArrayXd mrSensi = ArrayXd::Zero(p_a.size());
148 ArrayXXd correlSensi = ArrayXXd::Zero(p_correl.rows(), p_correl.cols());
149
150 double valueMean = 0.0;
151
152 for (size_t ibatch = 0; ibatch < nbBatch; ++ibatch)
153 {
154     const double weight =
155         static_cast<double>(nbSimulPerBatch[ibatch]) /
156         static_cast<double>(p_nbSimul);
157
158     delta += weight * deltaBatch[ibatch];
159     vega += weight * vegaBatch[ibatch];
160     mrSensi += weight * mrSenBatch[ibatch];
161     correlSensi += weight * correlSenBatch[ibatch];
162
163     valueMean += weight * valueBatch[ibatch];
164 }
165
166 timer1.stop();
167
168 boost::chrono::duration<double> durOpt1 =
169     boost::chrono::nanoseconds(timer1.elapsed().user);
170
171 const double timeSimul = durOpt1.count();
172
173 #ifdef _OPENMP
174     const int nbThreads = omp_get_max_threads();
175 #else
176     const int nbThreads = 1;
177 #endif
178
179 cout << "AutoDiff Adept OpenMP batched value " << valueMean
180     << " time " << timeSimul
181     << " nbBatch " << nbBatch
182     << " batchSize " << p_batchSize
183     << " nbSimul " << p_nbSimul
184     << " nbOpenMPThreads " << nbThreads
185     << endl;
186
187 return make_tuple(delta, vega, mrSensi, correlSensi);
188 }

```

In the example above, we use the Adept library to define a type that is used in the

simulation in the following call:

```

1 adouble value =
2     simulatePriceControlUsingControlStorageAutoDiff<OptimizeStorage<
3         MeanRevertingNDSimulatorT<adouble>>, adouble>(
4         p_tunnel,
5         optimizer,
6         p_pointStock,
7         p_fileForBellmanAndCont
8     );

```

Note that the type used in the price model is `adouble`, which enables automatic differentiation.

The `simulatePriceControlUsingControlStorageAutoDiff` function essentially performs a standard loop over time steps, and at each step calls the templated version of the `SimulateStepRegressionControlAutoDiff` object:

```

1 StOpt::SimulateStepRegressionControlAutoDiff<Optimizer, T>(
2     ar, istep, nameAr, gridNextStep, p_optimizer
3 ).oneStep(states, gainAndContFunction);

```

as shown in:

```

1 /// \brief Simulate the storage management using the Control values calculated by Dynamic
2   Programming
3 /// \param p_tunnel the tunnel for the storage
4 /// \param p_optimize the optimizer
5 /// \param p_pointStock initial position in storage
6 template< class Optimizer, typename T>
7 T simulatePriceControlUsingControlStorageAutoDiff(const Tunnel &p_tunnel,
8     const std::shared_ptr<Optimizer> &p_optimizer,
9     const Eigen::ArrayXd &p_pointStock,
10    const std::string &p_fileForControl)
11 {
12     // from the optimizer get back the simulator
13     std::shared_ptr< MeanRevertingNDSimulatorT<T>> simulator = std::static_pointer_cast<
14         MeanRevertingNDSimulatorT<T>>(p_optimizer->getSimulator());
15     int nbStep = simulator->getNbStep();
16     std::vector< StOpt::StateWithStocks<T>> states;
17     states.reserve(simulator->getNbSimul());
18     {
19         Eigen::Array<T, Eigen::Dynamic, Eigen::Dynamic> initPart = simulator->
20             getParticlesAD().array();
21         for (int is = 0; is < simulator->getNbSimul(); ++is)
22             states.push_back(StOpt::StateWithStocks<T>(0, p_pointStock.template cast<T>(),
23                 initPart.col(is)));
24     }
25     Eigen::Array<T, Eigen::Dynamic, 1> stockNext = Eigen::Array<T, Eigen::Dynamic, 1>::
26         Constant(simulator->getNbSimul(), T(p_pointStock(0)));
27
28     // test if one file generated
29     gs::BinaryFileArchive ar(p_fileForControl.c_str(), "r");
30     // name for continuation object in archive
31     std::string nameAr = "Continuation";
32     // gain function trajectories
33     Eigen::Array<T, Eigen::Dynamic, Eigen::Dynamic> gainAndContFunction = Eigen::Array<T,
34         Eigen::Dynamic, Eigen::Dynamic>::Zero(p_optimizer->getSimuFuncSize(), simulator->
35             getNbSimul());
36     std::shared_ptr< std::fstream> fileForResultsPosition;
37     std::shared_ptr< std::fstream> fileForResultsPrice ;
38
39     // use one Control file
40     // bool bOneFile = true;
41     for (int istep = 0; istep < nbStep; ++istep)
42     {

```

```

38     // grid on next step
39     // std::shared_ptr<StOpt::FullGrid> gridNextCurr= p_tunnel.getGrid(istep);
40     std::shared_ptr<StOpt::FullGrid> gridNextStep = p_tunnel.getGrid(istep + 1);
41     // optimal control
42     StOpt::SimulateStepRegressionControlAutoDiff<Optimizer, T>(ar, istep, nameAr,
43         gridNextStep, p_optimizer).oneStep(states, gainAndContFunction);
44
45     // step forward
46     // new stochastic state for next step
47     Eigen::Array<T, Eigen::Dynamic, Eigen::Dynamic> particules = simulator->
48         stepForwardAndGetParticlesAD();
49     // for next step : new stochastic state
50     for (int is = 0; is < simulator->getNbSimul(); ++is)
51         states[is].setStochasticRealization(particules.col(is));
52 }
53 return gainAndContFunction.row(0).mean();
54 }

```

Chapter 1

Some Python test cases description

This part is devoted to some test cases only available in Python. These examples use the low level Python interface.

1.1 Microgrid Management

1.1.1 testMicrogridBangBang

A microgrid is a collection of renewable energy sources, a diesel generator, and a battery for energy storage. The objective is to match the residual demand (difference between the demand of electricity and supply from renewables) while minimizing the total expected cost of running the microgrid. In particular a penalty is assessed for insufficient supply that leads to blackouts. The setup is similar to the one described in [[31], Section 7]. We take the diesel generator as the only control d_t ; output/input from/into the battery is then a function of the residual demand X_t (exogenous stochastic process), inventory level of the battery I_t , and d_t . The diesel generator operates under two regimes: OFF and ON. When it is OFF it does not supply power $d_t = 0$, however when it is ON the power output is a deterministic function of the state $d_t = d(X_t, I_t)$. As a result, the problem is a standard stochastic control model with switching-type bang-bang control.

We parameterize the algorithm to easily switch between multiple approximation schemes for the conditional expectation at the core of the Dynamic Programming equation. Particularly the following schemes are implemented:

- Regularly spaced grid for I_t and local polynomial basis in X_t for each level of the grid.
- Adaptive piecewise-defined polynomial basis in 2D for (X_t, I_t) .
- Global 2D polynomial basis on (X_t, I_t) .
- Bivariate Kernel regression on (X_t, I_t) .

1.1.2 testMicrogrid

We extend the previous example to include the recent work [1] where the action space for the control is $d_t \in \{0\} \cup [1, 10]$ kW, rather than being bang-bang. As a result, the optimal

control is chosen in two steps: first the controller picks the regime: ON or OFF; if ON, she then decides the optimal, continuous level of the diesel output. Due to the additional flexibility available to the controller compared to the previous example, we expect to observe lower cost compared to Section 1.1.1. The user can switch between this and the previous setting by changing the parameter `controlType` in the `parameters.py` file.

1.2 Dynamic Emulation Algorithm (DEA)

1.2.1 testMicrogridDEA

In this section we discuss the implementation of the Dynamic Emulation Algorithm developed in [31]. In that paper the authors reformulate the stochastic control problem as an “iterative sequence of machine learning tasks”. The philosophy of DEA is to combine together Regression Monte Carlo (RMC) with Design of Experiments. The algorithm has the following properties:

- The learning for the continuation value function at each step in the backward-iteration of the Dynamic Programming Equation is completely modularized. As a result, the user can seamlessly switch between different regression schemes (for example: adaptive local polynomial basis in -2D or -1D, multivariate kernel regression, etc.) across different time-steps;
- The empirical approximation uses distinct designs $\mathcal{D}_t$ at each t ; thus the user can have design sites independently chosen for different t ’s, which also eliminates the requirement to store the full history of the simulated paths of (X_t) . One-step paths can now replace the full history. In Figure 1.1 we present examples of two possible designs we use in the implementation. The image in the left panel represents a space-filling design using a Sobol sequence in -2D. This design is appropriate for a bivariate regression over (X_t, I_t) . On the right, we present another space-filling design in -1D with a regularly spaced grid in I_t (y-axis) and a -1D Sobol sequence in X_t (x-axis). In [31] the authors discuss several further designs which can be easily implemented.
- Batched designs, i.e. a partially nested scheme that generates multiple X_t -paths from the same unique design site, can be accommodated.
- Simulation budget (i.e. the size of $\mathcal{D}_t$) can vary through the time-steps and need not be fixed as in standard RMC.

Several different experiments have confirmed the significant effect of the design $\mathcal{D}_t$ on the performance of the RMC algorithms. DEA allows us to test for this effect by allowing the user to easily specify $\mathcal{D}_t$. The structure of this library allows for easy implementation of such modular algorithms. As a proof of concept, we re-implement the microgrid example of Section 1.1.1 with the following specifications:

- The 10 time-steps (25% of the total of 40 time-steps) closest to maturity use adaptive local polynomial basis in -1D with gridded design similar to the Figure 1.1b. Moreover, for these t ’s we used $|\mathcal{D}_t| = 22,000 = N_t$ unique design sites;

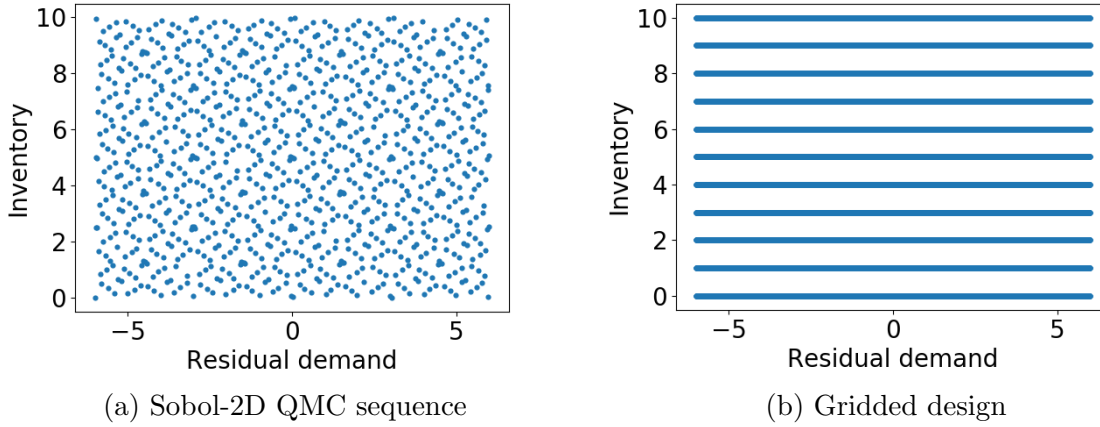


Figure 1.1: Illustration of two simulation designs. In both panels the X_t -coordinate is on the x-axis and I_t on the y-axis.

- The other 30 steps (first 75%) use design sites allocated according to Sobol-2D as in figure 1.1a with a global polynomial basis regression scheme. For these, we build a batched design of 1000 unique sites, each replicated 10 times for a total simulation budget of $N_t = 1000 \times 10 = 10^4$.

Bibliography

- [1] Clemence Alasseur, Alessandro Balata, Sahar Ben Aziza, Aditya Maheshwari, Peter Tankov, and Xavier Warin. Regression Monte Carlo for microgrid management. ESAIM: Proceedings and Surveys, 65:46–67, 2019.
- [2] Mejdî Azaïez, Monique Dauge, and Yvon Maday. Méthodes spectrales et des éléments spectraux. 1993.
- [3] Ole E Barndorff-Nielsen. Processes of normal inverse gaussian type. Finance and stochastics, 2(1):41–68, 1997.
- [4] Christian Bender and Robert Denk. A forward scheme for backward SDEs. Stochastic processes and their applications, 117(12):1793–1812, 2007.
- [5] JF Benders. Partitioning procedures for solving mixed-variables programming problems. Computational Management Science, 2(1):3–19, 2005.
- [6] J. L. Bentley. Multidimensional divide-and-conquer. Communications of the ACM, 23(4):214–229, 1980.
- [7] Eric Beutner. Mean–variance hedging under transaction costs. Mathematical Methods of Operations Research, 65(3):539–557, 2007.
- [8] Bruno Bouchard, Xiaolu Tan, Xavier Warin, and Yiyi Zou. Numerical approximation of BSDEs using local polynomial drivers and branching processes. Monte Carlo Methods and Applications, 23(4):241–263, 2017.
- [9] Bruno Bouchard and Xavier Warin. Monte-Carlo valuation of american options: facts and new algorithms to improve existing methods. In Numerical methods in finance, pages 215–255. Springer, 2012.
- [10] Hans-Joachim Bungartz. Dünne Gitter und deren Anwendung bei der adaptiven Lösung der dreidimensionalen Poisson-Gleichung. Technische Universität München, 1992.
- [11] Hans-Joachim Bungartz. Concepts for higher order finite elements on sparse grids. In Houston Journal of Mathematics: Proceedings of the 3rd Int. Conf. on Spectral and High Order Methods, Houston, pages 159–170, 1996.
- [12] Hans-Joachim Bungartz. A multigrid algorithm for higher order finite elements on sparse grids. Electronic Transactions on Numerical Analysis, 6:63–77, 1997.

- [13] Hans-Joachim Bungartz and Michael Griebel. Sparse grids. Acta numerica, 13:147–269, 2004.
- [14] Fabio Camilli and Maurizio Falcone. An approximation scheme for the optimal control of diffusion processes. ESAIM: Mathematical Modelling and Numerical Analysis, 29(1):97–122, 1995.
- [15] Robert P Feinerman and Donald J Newman. Polynomial approximation. 1974.
- [16] Wendell H Fleming and Halil Mete Soner. Controlled Markov processes and viscosity solutions, volume 25. Springer Science & Business Media, 2006.
- [17] Thomas Gerstner and Michael Griebel. Dimension–adaptive tensor–product quadrature. Computing, 71(1):65–87, 2003.
- [18] Anders Gjelsvik, Michael M Belsnes, and Arne Haugstad. An algorithm for stochastic medium-term hydrothermal scheduling under spot price uncertainty. In Proceedings of 13th Power Systems Computation Conference, 1999.
- [19] Emmanuel Gobet, Jean-Philippe Lemor, Xavier Warin, et al. A regression-based monte carlo method to solve backward stochastic differential equations. The Annals of Applied Probability, 15(3):2172–2202, 2005.
- [20] Michael Griebel. Adaptive sparse grid multilevel methods for elliptic PDEs based on finite differences. Computing, 61(2):151–179, 1998.
- [21] Michael Griebel. Sparse grids and related approximation schemes for higher dimensional problems. Citeseer, 2005.
- [22] Holger Heitsch and Werner Römisch. Scenario reduction algorithms in stochastic programming. Computational optimization and applications, 24(2-3):187–206, 2003.
- [23] Pierre Henry-Labordère, Nadia Oudjane, Xiaolu Tan, Nizar Touzi, and Xavier Warin. Branching diffusion representation of semilinear PDEs and Monte Carlo approximation. Annales de l’Institut Henri Poincaré, Probabilités et Statistiques, 55(1):184–210, 2019.
- [24] John C Hull. Options futures and other derivatives. Pearson Education India, 2003.
- [25] Hitoshi Ishii and Pierre-Luis Lions. Viscosity solutions of fully nonlinear second-order elliptic partial differential equations. Journal of Differential equations, 83(1):26–78, 1990.
- [26] Patrick Jaillet, Ehud I Ronn, and Stathis Tompaidis. Valuation of commodity-based swing options. Management science, 50(7):909–921, 2004.
- [27] John D Jakeman and Stephen G Roberts. Local and dimension adaptive sparse grid interpolation and quadrature. arXiv preprint arXiv:1110.0010, 2011.
- [28] Ali Koc and Soumyadip Ghosh. Optimal scenario tree reductions for the stochastic unit commitment problem. In Proceedings of the Winter Simulation Conference, page 10. Winter Simulation Conference, 2012.

- [29] Nicolas Langrené and Xavier Warin. Fast and stable multivariate kernel density estimation by fast sum updating. Journal of Computational and Graphical Statistics, 28(3):596–608, 2019.
- [30] Nicolas Langrené and Xavier Warin. Fast multivariate empirical cumulative distribution function with connection to kernel density estimation. arXiv preprint arXiv:2005.03246, 2020.
- [31] Michael Ludkovski and Aditya Maheshwari. Simulation methods for stochastic storage problems: A statistical learning perspective. Energy Systems, 11(2):377–415, 2020.
- [32] Xiang Ma and Nicholas Zabaras. An adaptive hierarchical sparse grid collocation algorithm for the solution of stochastic differential equations. Journal of Computational Physics, 228(8):3084–3113, 2009.
- [33] Alessandro Magnani and Stephen P Boyd. Convex piecewise-linear fitting. Optimization and Engineering, 10(1):1–17, 2009.
- [34] Constantinos Makassikis, Stéphane Vialle, and Xavier Warin. Large scale distribution of stochastic control algorithms for gas storage valuation. In Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on, pages 1–8. IEEE, 2008.
- [35] M Motoczyński. Multidimensional variance-optimal hedging in discrete-time model - a general approach. Mathematical Finance, 10(2):243–257, 2000.
- [36] Rémi Munos and Hasnaa Zidani. Consistency of a simple multidimensional scheme for hamilton–jacobi–bellman equations. Comptes Rendus Mathématique, 340(7):499–502, 2005.
- [37] Mario Pereira, Nora Campodonico, and Rafael Kelman. Application of stochastic dual dp and extensions to hydrothermal scheduling. Online Rep., <http://www.psr-inc.com.br/reports.asp>, PSRI Technical Rep, 12:99, 1999.
- [38] Mario VF Pereira and Leontina MVG Pinto. Multi-stage stochastic optimization applied to energy planning. Mathematical programming, 52(1-3):359–375, 1991.
- [39] Laurent Pfeiffer, Romain Apparigliato, and Sophie Auchapt. Two methods of pruning Benders’ cuts and their application to the management of a gas portfolio. PhD thesis, INRIA, 2012.
- [40] Dirk Michael Pflüger. Spatially adaptive sparse grids for high-dimensional problems. PhD thesis, Technische Universität München, 2010.
- [41] Alfio Maria Quarteroni, Riccardo Sacco, and Fausto Saleri. Méthodes numériques pour le calcul scientifique: programmes en MATLAB. Springer Science & Business Media, 2000.
- [42] Martin Schweizer. Variance-optimal hedging in discrete time. Mathematics of Operations Research, 20(1):1–32, 1995.

- [43] David W Scott. Multivariate density estimation: theory, practice, and visualization. John Wiley & Sons, 2015.
- [44] Paolo M Soardi. Serie di Fourier in piu variabili, volume 26. Pitagora, 1984.
- [45] Stéphane Vialle, Xavier Warin, Constantinos Makassikis, and Patrick Mercier. Stochastic control optimization & simulation applied to energy management: From 1-d to nd problem distributions, on clusters, supercomputers and grids. In Grid@ Mons conference, 2008.
- [46] MP Wand. Fast computation of multivariate kernel estimators. Journal of Computational and Graphical Statistics, 3(4):433–445, 1994.
- [47] Xavier Warin. Gas storage hedging. In Numerical Methods in Finance, pages 421–445. Springer, 2012.
- [48] Xavier Warin. Adaptive sparse grids for time dependent hamilton-jacobi-bellman equations in stochastic control. arXiv preprint arXiv:1408.4267, 2014.
- [49] Xavier Warin. Some non-monotone schemes for time dependent hamilton–jacobi–bellman equations in stochastic control. Journal of Scientific Computing, 66(3):1122–1147, 2016.
- [50] Xavier Warin. Variance optimal hedging with application to electricity markets. arXiv preprint arXiv:1711.03733, 2017.
- [51] Xavier Warin. Variations on branching methods for non linear pdes. arXiv preprint arXiv:1701.07660, 2017.
- [52] Xavier Warin. Monte Carlo for high-dimensional degenerated semi linear and full non linear PDEs. arXiv preprint arXiv:1805.05078, 2018.
- [53] Xavier Warin. Nesting Monte Carlo for high-dimensional non linear PDEs. Monte Carlo Methods and Applications, 24(4):225–247, 2018.
- [54] Thaleia Zariphopoulou. A solution approach to valuation with unhedgeable risks. Finance and Stochastics, 5(1):61–82, 2001.