STochastic OPTimization library in C++
Hugo Gevret, Jerome Lelong, Xavier Warin

To cite this version:
Hugo Gevret, Jerome Lelong, Xavier Warin. STochastic OPTimization library in C++. [Research Report] EDF Lab. 2016. hal-01361291v1

HAL Id: hal-01361291
https://hal.archives-ouvertes.fr/hal-01361291v1
Submitted on 7 Sep 2016 (v1), last revised 20 Feb 2019 (v7)

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
STochastic OPTimization library in C++

Hugo Gevret $^1$    Jerome Lelong $^2$    Xavier Warin $^3$

$^1$EDF R&D
$^2$Ensimag
$^3$EDF R&D & FiME, Laboratoire de Finance des Marchés de l’Energie, xavier.warin@edf.fr
Contents

I Introduction

II Useful tools for stochastic control

1 The grids and their interpolators

1.1 Linear grids

1.1.1 Definition and C++ API

1.1.2 The python API

1.2 Legendre grids

1.2.1 Approximation of a function in 1 dimension

1.2.2 Extension in dimension $d$

1.2.3 Truncation

1.2.4 The C++ API

1.2.5 The python API

1.3 Sparse grids

1.3.1 The linear sparse grid method

1.4 High order sparse grid methods

1.5 Anisotropy

1.6 Adaptation

1.7 C++ API

1.8 Python API

2 Introducing the regression resolution

2.1 C++ global API

2.2 Adapted local polynomial basis

2.2.1 Description of the method

2.3 C++ API

2.3.1 The constant per cell approximation

2.3.2 The linear per cell approximation

2.3.3 An example in the linear case

2.4 Python API

2.5 Local polynomial basis with meshes of same size

2.6 C++ API

2.6.1 The constant per cell approximation

2.6.2 The linear per cell approximation
7.2 Time discretization for HJB equation ................................................. 106
7.3 Space interpolation ........................................................................... 106

8 C++ API ................................................................................................. 108
8.1 PDE resolution .................................................................................. 116
8.2 Simulation framework ....................................................................... 118

V An example with both dynamic programming with regression and PDE .... 125
8.3 The dynamic programming with regression approach ....................... 127
8.4 The PDE approach ............................................................................ 131

VI Stochastic Dual Dynamic Programming ............................................. 135

9 SDDP algorithm .................................................................................... 136
9.1 Some general points about SDDP ...................................................... 136
9.2 A method, different algorithms ......................................................... 138
9.2.1 The basic case ............................................................................. 138
9.2.2 Dependence of the random quantities ......................................... 142
9.2.3 Non-convexity and conditionnal cuts .......................................... 145
9.3 C++ API ............................................................................................ 148
9.3.1 Inputs ............................................................................................ 148
9.3.2 Set of parameters ......................................................................... 151
9.3.3 The black box ............................................................................... 151
9.3.4 Outputs ......................................................................................... 152
Part I

Introduction
The STtochastic OPTimization library (StOpt)

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

aims at providing tools for solving some stochastic optimization problems encountered in finance or in the industry. In a continuous setting, the controlled state is given by a stochastic differential equation

\[
\begin{align*}
\text{d}X^{x,t}_s &= b_a(t, X^{x,t}_s)ds + \sigma_a(s, X^{x,t}_s)dW_s \\
X^{x,t}_t &= x
\end{align*}
\]

where

- \(W_t\) is a \(d\)-dimensional Brownian motion on a probability space \((\Omega, \mathcal{F}, \mathbb{P})\) endowed with the natural (completed and right-continuous) filtration \(\mathbb{F} = (\mathcal{F}_t)_{t \leq T}\) generated by \(W\) up to some fixed time horizon \(T > 0\),
- \(\sigma_a\) 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

\[
J(t, x, a) = \mathbb{E}\left[\int_T^T f_a(s, X^{x,t}_s)e^{\int_t^s c_a(u, X^{x,t}_u)du}ds + e^{\int_T^T c_a(u, X^{x,t}_u)}g(X^{x,t}_T)\right]
\]

with respect to the control \(a\). It is well known \[1\] that the optimal value \(\hat{J}(t, x) = \inf_a J(T - t, x, a)\) is a viscosity solution of the equation

\[
\frac{\partial v}{\partial t}(t, x) - \inf_{a \in A} \left( \frac{1}{2} tr(\sigma_a(t, x)\sigma_a(t, x)^T D^2 v(t, x)) + b_a(t, x)Dv(t, x) \\
+ c_a(t, x)v(t, x) + f_a(t, x) \right) = 0 \text{ in } \mathbb{R}^d
\]

\(v(0, x) = g(x) \text{ in } \mathbb{R}^d\) \hspace{1cm} (1)

Under some classical assumptions on the coefficients [], the previous equation known as the Hamilton Jacobi Bellman equation admits an unique viscosity solution (2). The resolution of the previous equation is quite hard especially in dimension greater than 3 or 4.

The library provides tools to solve this equation and simplified versions of it.

- a first method supposes that \(X^{x,t}_s = (X^{x,t}_{1,s}, X^{x,t}_{2,s})\) where \(X^{x,t}_{1,s}\) is not controlled

\[
\begin{align*}
\text{d}X^{x,t}_{1,s} &= b(t, X^{x,t}_{1,s})ds + \sigma_t(s, X^{x,t}_{1,s})dW_s \\
X^{x,t}_{1,t} &= x
\end{align*}
\]

and \(X^{x,t}_{2,s}\) has no diffusion term

\[
\begin{align*}
\text{d}X^{x,t}_{2,s} &= b_a(t, X^{x,t}_{2,s})ds \\
X^{x,t}_{2,t} &= x
\end{align*}
\]
In this case we can use Monte Carlo methods based on regression to solve the problem. The method is based on the Dynamic Programming principle and can be used even if the non controlled SDE is driven by a general Levy process. This method can be used even if the controlled state takes only some discrete values.

- The second case is a special case of the previous one when the problem to solve is linear and when the controlled state takes some values in some continuous intervals. The value function has to 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 high, preventing the use of the Dynamic Programming method.

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

- A third method permits to solve the problem with Monte Carlo when a process is controlled but by the mean of an uncontrolled process. This typically the case of the optimization of a portfolio:
  - The portfolio value is controlled and deterministically discretized on a grid,
  - The portfolio evolution is driven by an exogenous process not controlled: the market prices.

- In the last method, we will suppose that the state takes continuous values, we will solve equation (1) using Semi Lagrangian methods discretizing the Brownian motion with two values and using some interpolations on grids.

In the sequel, we suppose that a time discretization is given for the resolution of the optimization problem. We suppose 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 some optimization problems using these developed tools.

**Remark 2** In the library, we heavily relies on the Eigen library. “ArrayXd” stands for a vector of double, “ArrayXXd” for a matrix of double and “ArrayXi” 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 function discretized on a given grid. A grid is a set of points in \( \mathbb{R}^d \) defining some meshes that can be used to interpolate a function on an open set in \( \mathbb{R}^d \). These tools are used to interpolate a function given for example at some stock points, when dealing with storages. There are also useful for Semi Lagrangian methods, which need effective interpolation methods. In StOpt currently four kinds of grids are available:

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

To each kind of grids are associated some iterators. An iterator on a grid permits to iterate on all points of the grids. All iterators derive from the abstract class `GridIterator`.

```cpp
// Copyright (C) 2016 EDF
// All Rights Reserved
// This code is published under the GNU Lesser General Public License (GNU LGPL)

#ifdef GRIDITERATOR_H
#define GRIDITERATOR_H
#include <Eigen/Dense>

/** \file GridIterator.h
 * \brief Defines an iterator on the points of a grid
 * \author Xavier Warin
 */

namespace StOpt {

/// \class GridIterator GridIterator.h
/// Iterator on a given grid

class GridIterator {

```
public :

  /// \brief Constructor
  GridIterator() {}

  /// \brief Destructor
  virtual ~GridIterator() {}

  /// \brief get current coordinates
  virtual Eigen::ArrayXd getCoordinate() const = 0;

  /// \brief Check if the iterator is valid
  virtual bool isValid(void) const = 0;

  /// \brief iterate on point
  virtual void next() = 0;

  /// \brief iterate jumping some point
  /// param p_incr increment in the jump
  virtual void nextInc(const int &p_incr) = 0;

  /// \brief get counter : the integer associated the current point
  virtual int getCount() const = 0;

  /// \brief Permits to jump to a given place given the number of processors (permits to use MPI and openmp)
  /// param p_rank processor rank
  /// param p_nbProc number of processor
  /// param p_jump increment jump for iterator
  virtual void jumpToAndInc(const int &p_rank, const int &p_nbProc, const int &p_jump) = 0;

  /// \brief return relative position
  virtual int getRelativePosition() const = 0;

  /// \brief return number of points treated
  virtual int getNbPointRelative() const = 0;

  /// \brief Reset the interpolator
  virtual void reset() = 0;
};
#endif /* GRIDITERATOR_H */

All the iterators share some common features:

- the “getCount” method permits to get the number associated to the current grid point,
- the “next” method permits to go to the next point, while the “nextInc” method permits to jump forward to the “p_incr” point,
- the “isValid” method permits to check that we are still on a grid point,
• the “getNbPointRelative” method permits to get the number of points that a given iterator can iterate on,

• the “getRelativePosition” get the number of points already iterated by the iterator.

Besides, we can directly jump to a given point : this feature is useful for “mpi” when a treatment on the grid is split between some processor and threads. This possibility is given by the “jumpToAndInc” method.

Using a grid “regGrid” the following source code permits to iterate on the points of the grids and get coordinates. For each coordinate, a function \( f \) is used to fill in an array of values. As pointed out before, each type of grid has its own grid iterator that can be obtained by the “getGridIterator” method.

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

It is also possible to “jump” some points and iterate to “p” points after. This possibility is useful for multithreaded tasks on points.

To each kind of grids, an interpolator is provided to interpolate a function given on a grid. Notice that the interpolator is created for a given point where we want to interpolate. All interpolators (not being spectral interpolators) derive from “Interpolator.h” whose source code is given below

```cpp
// Copyright (C) 2016 EDF
// All Rights Reserved
// This code is published under the GNU Lesser General Public License (GNU LGPL)

#ifndef INTERPOLATOR_H
#define INTERPOLATOR_H

#include <vector>
#include <Eigen/Dense>

/** \file Interpolator.h
 * \brief Defines a interpolator on a full grid
 * \author Xavier Warin
 */

namespace StOpt
{

// \class Interpolator Interpolator.h
// \Interpolation base class

public :
```
All interpolators provide a constructor specifying the point where the interpolation is achieved and the two functions “apply” and “applyVec” interpolating either a function (and sending back a value) or an array of functions sending back an array of interpolated values.

All the grid classes derive from an abstract class “SpaceGrids.h” below permitting to get back an iterator associated to the points of the grid (with possible jumps) and to create an interpolator associated to the grid.
public:

/// \brief Default constructor
SpaceGrid() {}

/// \brief Default destructor
virtual ~SpaceGrid() {};

/// \brief Number of points of the grid
virtual size_t getNbPoints() const = 0;

/// \brief get back iterator associated to the grid
virtual std::shared_ptr<GridIterator> getGridIterator() const = 0;

/// \brief get back iterator associated to the grid (multi thread)
virtual std::shared_ptr<GridIterator> getGridIteratorInc(const int &p_iThread) const = 0;

/// \brief Get back interpolator at a point Interpolate on the grid
/// \param p_coord coordinate of the point for interpolation
/// \return interpolator at the point coordinates on the grid
virtual std::shared_ptr<Interpolator> createInterpolator(const Eigen::ArrayXd &p_coord) const = 0;

/// \brief Get back a spectral operator associated to a whole function
/// \param p_values Function value at the grids points
/// \return the whole interpolated value function
virtual std::shared_ptr<InterpolatorSpectral> createInterpolatorSpectral(const Eigen::ArrayXd &p_values) const = 0;

/// \brief Dimension of the grid
virtual int getDimension() const = 0;

/// \brief get back bounds associated to the grid
/// \return in each dimension give the extreme values (min, max) of the domain
virtual std::vector<std::array<double, 2>> getExtremeValues() const = 0;

/// \brief test if the point is strictly inside the domain
/// \param p_point point to test
/// \return true if the point is strictly inside the open domain
virtual bool isStrictlyInside(const Eigen::ArrayXd &p_point) const = 0;

/// \brief test if a point is inside the grid (boundary include)
/// \param p_point point to test
/// \return true if the point is inside the open domain
virtual bool isInside(const Eigen::ArrayXd &p_point) const = 0;

/// \brief truncate a point that it stays inside the domain
/// \param p_point point to truncate
virtual void truncatePoint(Eigen::ArrayXd &p_point) const = 0;
All the grids objects, interpolators and iterators on grids point are in

\[ \textit{StOpt/core/grids} \]

The grids objects are mapped with python, giving the possibility to get back the iterators and the interpolators associated to a grid. Python examples can be found in

\[ \textit{test/python/unit/grids} \]

## 1.1 Linear grids

### 1.1.1 Definition and C++ API

Two kinds of grids are developed:

- the first one is the "GeneralSpaceGrid.h" with constructor

\[
\text{GeneralSpaceGrid( const std::vector<shared_ptr<Eigen::ArrayXd> > & p_meshPerDimension )}
\]

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

![Figure 1.1: 2D general grid](image)

- the second one is the "RegularSpaceGrid" with constructor

\[
\text{RegularSpaceGrid( const Eigen::ArrayXd &p_lowValues , const Eigen::ArrayXd &p_step , const Eigen::ArrayXi &p_nbStep )}
\]

The \( p\_lowValues \) correspond 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).
For each grid, a linear interpolator can be generated by call to the `createInterpolator`
method or by creating directly the interpolator:

```cpp
/** \brief Constructor
 * \param p_grid is the grid used to interpolate
 * \param p_point is the coordinates of the points used for interpolation
 */
LinearInterpolator( const FullGrid * p_grid , const Eigen::ArrayXd & p_point );
```

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

```cpp
ArrayXd data( regGrid.getNbPoints() ); // create an array to store the values of the function f
shared_ptr<GridIterator> iterRegGrid = regGrid.getGridIterator();
while ( iterRegGrid->isValid() )
{
    ArrayXd pointCoord = iterRegGrid->getCoordinate(); // store the coordinate of the point
    data(iterRegGrid->getCount()) = f(pointCoord); // the value is stored in data at place iterRegGrid->getCount()
    iterRegGrid->next(); // go to next point
}
// point where to interpolate
ArrayXd point = ArrayXd::Constant(nDim, 1. / 3.);
// create the interpolator
LinearInterpolator regLin(&regGrid, point);
// get back the interpolated value
double interpReg = regLin.apply(data);
```

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

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

(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 makes it possible to use the grids with a similar syntax to the C++ API. We give here an example with a regular grid.

```python
import numpy as np
import unittest
import random
import math
import StOptGrids

# unit test for regular grids
class testGrids(unittest.TestCase):
    # 3 dimensional test for linear interpolation on regular grids
    def testRegularGrids(self):
        # low value for the meshes
        lowValues = np.array([1., 2., 3.], dtype=np.float)
        # size of the meshes
        step = np.array([0.7, 2.3, 1.9], dtype=np.float)
        # number of steps
        nbStep = np.array([4, 5, 6], dtype=np.int32)
        # create the regular grid
        grid = StOptGrids.RegularSpaceGrid(lowValues, step, nbStep)
        iterGrid = grid.getGridIterator()
        # array to store
        data = np.empty(grid.getNbPoints())
        # iterates on points and store values
        while(iterGrid.isValid()):
            # get coordinates of the point
            pointCoord = iterGrid.getCoordinate()
            data[iterGrid.getCount()] = math.log(1. + pointCoord.sum())
            iterGrid.next()
        # get back an interpolator
        ptInterp = np.array([2.3, 3.2, 5.9], dtype=np.float)
        interpol = grid.createInterpolator(ptInterp)
        # calculate interpolated value
        interpValue = interpol.apply(data)
        print(f"Interpolated value {interpValue}")
        # test grids function
        iDim = grid.getDimension()
        pt = grid.getExtremeValues()

if __name__ == '__main__':
    unittest.main()
```
1.2 Legendre grids

With linear interpolation, in order to get an accurate solution, it is needed to refine the mesh so that $\Delta x$ go to zero. Another approach consists in trying to fit on each mesh a polynomial by using a high degree interpolator.

1.2.1 Approximation of a function in 1 dimension.

From now, by re-scaling we suppose 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 the polynomials of total degree below 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 $I^X_N$ interpolator from $f$ on a grid of $N + 1$ points of $[-1, 1]$ $X = (x_0, .., x_N)$, the unique polynomial of degree $N$ such that

$$I^X_N(f)(x_i) = f(x_i), 0 \leq i \leq N$$

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

$$I^X_N(f)(x) = \sum_{i=0}^{N} f(x_i)l^X_i(x)$$

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

$$||I^X_N(f)(x) - f||_{\infty} \leq (1 + \lambda_N(X)) ||f - P^*_N(f)||_{\infty}$$

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

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

We have the following bound

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

and the Erdös 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 as $N \rightarrow \infty$, the Lebesgue constant satisfies

$$\lambda_N(X_u) \simeq \frac{2^{N+1}}{cNlnN}$$
and the quadrature error in $L_\infty$ increases a lot with $N$. Its use brings some oscillations giving the Runge effect. On figure we plot the Runge function $\frac{1}{1+25x^2}$ against its interpolation with polynomial with equidistant interpolation.

So we are interested in having quadrature with an “optimal” Lebesgue constant. For example 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 high accuracy on the mesh while respecting the continuity of the function between the meshes. In order to ensure this continuity we want the extreme points on the re-scaled mesh $[-1, -1]$ (so $-1, 1$) 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 is as efficient as the Gauss Lobatto Chebyshev grids (in term of the Lebesgue constant) but computationally less costly due to absence of trigonometric function. 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 $L_N'$ that 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 grids with $N+1$ points are $\eta_1 = -1, \eta_{N+1} = 1$ and the $\eta_i (i = 2, ..., N)$ zeros of $L_N'$. The $\eta_i (i = 2, ..., N)$ are eigenvalues of the matrix $P$

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

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

The interpolation $I_N(f)$ is expressed in terms of the 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 [4]. In 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](image)

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

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

18
The modulus of continuity permits 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 a grid of interpolation \( X \)

\[
\|I_N^X(f)(x) - f\|_\infty \leq M(N)
\]

\[
M(N) \approx Kw(f, \frac{1}{N})\lambda_N(X)
\]

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

- When the solution is more regular we can express the interpolation error as a function of its derivatives and we get the following Cauchy theorem for an interpolation grid \( X \) (see [3])

**Theorem 2** If \( f \) is \( C^{N+1} \), and \( X \) an interpolation grid with \( N + 1 \) points, then the interpolation error verifies

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

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 null at all the \( N + 1 \) points of \( X \))

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

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

### 1.2.2 Extension in dimension \( d \)

In dimension \( d \), we note \( P^*_N \) the best multivariate polynomial approximation of \( f \) of total degree lesser 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 \( I_N^X(f)(x) = I_N^{X_N^1} \times I_N^{X_N^2} \times \cdots \times I_N^{X_N^d}(f)(x) \) where \( I_N^{X_N^i} \) stands 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 to the min max approximation is given by Feinerman and Newman [5], Soardi [6]

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

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

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

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

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

![Figure 1.5: Gauss Legendre Lobatto points on 2 x 2 meshes.](image)

**1.2.3 Troncature**

In order to avoid oscillations while interpolating, a troncature is used on each mesh such that the modified interpolator $I_{N, \Delta x}^X$ satisfies:

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

where the $x_i$ are the interpolation points on the mesh $M$ containing the point $x$. For all characteristics of this modified operator, one can see [24].
1.2.4 The C++ API

The grid using Gauss Legendre Lobatto points can be created by the use of this constructor:

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

The `p_lowValues` correspond 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 `p_poly` array.

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

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

```cpp
ArrayXd lowValues = ArrayXd::Constant(1,−1.); // corner point
ArrayXd step = ArrayXd::Constant(1,2.); // size of the meshes
ArrayXi nbStep = ArrayXi::Constant(1,1); // number of mesh in each direction
ArrayXi nPol = ArrayXi::Constant(1,p_nPol); // polynomial approximation
// regular Legendre
RegularLegendreGrid regGrid(lowValues, step, nbStep, nPol);

// Data array to store values on the grid points
ArrayXd data(regGrid.getNbPoints());
while (iterRegGrid.isValid())
{
    ArrayXd pointCoord = iterRegGrid->getCoordinate();
    data(iterRegGrid->getCount()) = 1./(1.+25*pointCoord(0)*pointCoord(0));
    // store runge function
    iterRegGrid->next();
}
// point
ArrayXd point(1);
int npb = 1000;
double dx = 2./nbp;
for (int ip = 0; ip <= npb; ++ip)
{
    point(0) = −1+ ip* dx;
    // create interpolator
    shared_ptr<Interpolator> interp = regGrid.createInterpolator( point);
    double interpReg = interp->apply(data); // interpolated value
}
```

The previously defined operator is more effective when we interpolate many function at the same point. Its is the case for example for the valorization of a storage with regression where you want to interpolate all the simulations at the same stock level.
In some cases, it is more convenient to construct an interpolator acting on a global function. It is the case when you have a single function and you want to interpolate at many points for this function. In this specific case, an interpolator deriving from the class InterpolatorSpectral can be constructed:

```cpp
// Copyright (C) 2016 EDF
// All Rights Reserved
// This code is published under the GNU Lesser General Public License (GNU LGPL)
#ifndef INTERPOLATORSPECTRAL_H
#define INTERPOLATORSPECTRAL_H
#include <Eigen/Dense>
#include "StOpt/core/grid/SpaceGrid.h"

/** \file InterpolatorSpectral.h
 * \brief Defines an interpolator for a grid: here is a global interpolator, storing the representation of the function
 * to interpolate: this interpolation is effective when interpolating the same function many times at different points
 * Here it is an abstract class
 */
namespace StOpt {

/// forward declaration
class SpaceGrid ;

/// \class InterpolatorSpectral InterpolatorSpectral.h
/// Abstract class for spectral operator
class InterpolatorSpectral {

public :

virtual ~InterpolatorSpectral() {}  

/** \brief interpolate
 * \param p_point coordinates of the point for interpolation
 * \return interpolated value
 */
virtual double apply(const Eigen::ArrayXd &p_point) const = 0;

/** \brief Affect the grid
 * \param p_grid the grid to affect
 */
virtual void setGrid(const StOpt::SpaceGrid *p_grid) = 0 ;
};
#endif
```

Its constructor is given by:

```cpp
/** \brief Constructor taking in values on the grid
 * \param p_grid is the grid used to interpolate
 */
```
This class has a member permitting to interpolate at a given point:

```cpp
/** \brief interpolate
 * \param p_point coordinates of the point for interpolation
 * \return interpolated value
 */
inline double apply(const Eigen::ArrayXd &p_point) const
```

We give an example of the use of this class, interpolating a function \( f \) in dimension 2.

```cpp
ArrayXd lowValues = ArrayXd::Constant(2, 1.); // bottom of the domain
ArrayXd step = ArrayXd::Constant(2, 1.); // size of the mesh
ArrayXi nbStep = ArrayXi::Constant(2, 5); // number of meshes in each direction
ArrayXi nPol = ArrayXi::Constant(2, 2); // polynomial of degree 2 in each direction
// regular
shared_ptr<RegularLegendreGrid> regGrid(new RegularLegendreGrid(lowValues,
    step, nbStep, nPol));
ArrayXd data(regGrid->getNbPoints()); // Data array
shared_ptr<GridIterator> iterRegGrid = regGrid->getGridIterator(); // iterator on the grid points
while (iterRegGrid->isValid())
{
    ArrayXd pointCoord = iterRegGrid->getCoordinate();
    data(iterRegGrid->getCount()) = f(pointCoord);
    iterRegGrid->next();
}
// spectral interpolator
LegendreInterpolatorSpectral interpolator(regGrid, data);
// interpolation point
ArrayXd pointCoord(2, 5.2);
// interpolated value
double vInterp = interpolator.apply(pointCoord);
```

### 1.2.5 The python API

Here is an example using Legendre grids:

```python
# Copyright (C) 2016 EDF
# All Rights Reserved
# This code is published under the GNU Lesser General Public License (GNU LGPL)
import numpy as np
import unittest
import random
import math
import StOptGrids
```
1.3 Sparse grids

A representation of a function in dimension $d$ for $d$ small (less than 4) is achieved by
tensorization in the previous interpolation methods. When the function is smooth and when its
cross derivatives are bounded, one can represent the function using the sparse grid methods.
This methods permits to represent the function with far less points than classical without
losing too much while interpolating. The sparse grid method was first used supposing that
the function $f$ to represent is null at the boundary $\Gamma$ of the domain. This assumption is
important because it permits to limit the explosion of the number of points with the dimen-
sion 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 suppose that the function is not null at the boundary and provide grid object, iterators and interpolators to interpolate some functions represented on the sparse grid. Nevertheless, for the sake of clarity of the presentation, we will begin with the case of a function vanishing on the boundary.

1.3.1 The linear sparse grid method

We recall some classical results on sparse grids that can be found in [9]. We first assume that the function we interpolate is null at the boundary. By a change of coordinate an hyper-cube domain can be changed to a domain $\omega = [0,1]^d$. Introducing the hat function $\phi_{(L)}(x) = \max(1 - |x|, 0)$ (where $(L)$ stands for linear), we obtain the following local one dimensional hat function by translation and dilatation

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

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

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

via a tensor approach for a point $x = (x_1, ..., x_d)$, a multi-level $\underline{l} := (l_1, ..., l_d)$ and a multi-index $\underline{i} := (i_1, ..., i_d)$. The grid points used for interpolation are noted $x_{\underline{l},\underline{i}} := (x_{l_1,i_1}, ..., x_{l_d,i_d})$. We next introduce 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 space of hierarchical basis

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

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

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

(1.4)

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

At a space of hierarchical increments $W_{\underline{l}}^{(L)}$ corresponds a space of nodal function $W_{\underline{l}}^{(L,N)}$ such that

$$W_{\underline{l}}^{(L,N)} := \text{span} \left\{ \phi_{\underline{l},\underline{i}}^{(L)}(x) : \underline{i} \in B_{\underline{l}}^N \right\}$$
Figure 1.6: One dimensional $W^{(L)}$ spaces: $W_1^{(L)}, W_2^{(L)}, W_3^{(L)}, W_4^{(L)}$ and the nodal representation $W_4^{(L,N)}$

with

$$B^N_l := \{ i : 1 \leq i_j \leq 2^j - 1, 1 \leq j \leq d \}.$$ 

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

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

A function $f$ is interpolated on the hierarchical basis as

$$I^{(L)}(f) = \sum_{|l|_1 \leq n+d-1, i \in B_l} \alpha^{(L)}(L)_{l,i} \phi^{(L)}_{l,i},$$

where $\alpha^{(L)}_{l,i}$ are called the surplus (we give on figure 1.7 a representation of these coefficients). These surplus associated to a function $f$ are calculated in the one dimension 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 function calculated with neighboring nodes. For example on figure 1.8 the hierarchical value is given by the relation:

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

where $e(m)$ is the east neighbor of $m$ and $w(m)$ the west one. The procedure is generalized in $d$ dimension by successive hierarchization in all the directions. On figure 1.9 we give a representation of the $W$ subspace for $l \leq 3$ in dimension 2.

In order to deal with functions not null at the boundary, two more basis are added to the
Figure 1.8: Node involved in linear, quadratic and cubic representation of a function at node $m$ and $n$

Figure 1.9: The two dimensional subspace $W(L)$ up to $l = 3$ in each dimension. The additional hierarchical functions corresponding to an approximation on the full grid are given in dashed lines.

first level as shown on figure [1.10]. This approach results in many more points than the one without the boundary. As noted in [9] for $n = 5$, in dimension 8 you have nearly 2.8 millions points in this approximation but only 6401 inside the domain. On figure [1.11] we give the grids points with 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 domain truncated in finance for example) the hat functions near the boundaries are modified by extrapolation (see figure
Figure 1.10: One dimensional $W^{(L)}$ spaces with linear functions with “exact” boundary (left) and “modified” boundary (right): $W_1^{(L)}$, $W_2^{(L)}$, $W_3^{(L)}$, $W_4^{(L)}$

as explained in [9]. On level 1, we only have one degree of freedom assuming the function is constant on the domain. On all other levels, we extrapolate linearly towards the boundary the left and right basis functions, other functions remaining unchanged. So the new functions basis in 1D $\tilde{\phi}$ becomes

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

if $l > 1$ and $i = 1$

if $l > 1$ and $i = 2^l - 1$

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

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

The interpolation error associated to the linear operator $I^1 := I^{(L)}$ is linked to the regularity of the cross derivatives of the function [10, 17, 18]. If $f$ is null at the boundary and admits derivatives such that $\|\partial_{x_1}^{d_y}...\partial_{x_d}^{d_y}\|_\infty < \infty$ then

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

with $N$ the number of points per dimension.
1.4 High order sparse grid methods

Changing the interpolator enables us to get a higher rate of convergence mainly in region
where the solution is smooth. Following [17] and [18], it is possible to get higher order
interpolators. Using a quadratic interpolator, the reconstruction on the nodal basis gives a
quadratic function on the support of the previously defined hat function and a continuous
function of the whole domain. The polynomial quadratic basis is defined on $[2^{-l}(i-1), 2^{-l}(i+1)]$ by

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

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

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

$$\alpha(m)^{(Q)} = f(m) - \frac{3}{8} f(w(m)) + \frac{3}{4} f(e(m)) - \frac{1}{8} f(ee(m)))$$

$$= \alpha(m)^{(L)}(m) - \frac{1}{4} \alpha(m)^{(L)}(e(m))$$

$$= \alpha(m)^{(L)}(m) - \frac{1}{4} \alpha(m)^{(L)}(df(m))$$

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

Once again the quadratic surplus in 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 such that

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

In the case of the cubic representation, on figure 1.8 we need 4 points to define a function
basis. In order to keep the same data structure, we use a cubic function basis at node $m$
with value 1 at this node and 0 at the node $e(m)$, $w(m)$ and $ee(m)$ and we only keep the
basis function between $w(m)$ and $e(m)$ [17].

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

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

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

The coefficient surplus can be defined as before as the difference between the value function
at the node and the cubic representation of the function at the father node. Because of the
two basis functions involved there are two kind 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
  
  $$\alpha^{(C,2)}(m) = \alpha^{(Q)}(m) + \frac{1}{8}\alpha^{(Q)}(df(m))$$

Notice that a cubic representation is not available for $l = 1$ so a quadratic approximation is used. As before boundary conditions are treated by adding two linear functions basis at the first level and a modified version is available. We choose the following basis functions as defined on figure 1.14:

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

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, 17, 18], if the function $f$ is null at the boundary and admits derivatives such that $\sup_{\alpha_i \in \{2, \ldots, p+1\}} \left\{ \left\| \frac{\partial^{\alpha_1 + \ldots + \alpha_d} f}{\partial x_1 \ldots \partial x_d} \right\|_{\infty} \right\} < \infty$ 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.

### 1.5 Anisotropy

In many situations, it is useless to refine as much in each direction. For example, when dealing with multidimensional storages we expect the mesh size to be of the same order in each direction. When the different storages have very different sizes, we want to refine more the storage with the highest capacity. In order to treat this anisotropy an extension of Sparse grids can be achieved by defining weight $w$ in each direction. The definition 1.4 is replaced by:

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

### 1.6 Adaptation

When the solution is not smooth, typically Lipschitz, there is no hope to get convergence results for classical Sparse Grids (see above the interpolation error linked to the cross derivatives of the function). So classical sparse grids have to be adapted such that the solution is refined near singularities. In all adaptations methods hierarchical surplus $\alpha_{l,i}$ are used to get an estimation of the local error. These coefficients give an estimation of the smoothness of the function value at the discrete points by representing the discrete mix second derivative of the function. There is mainly two kinds of adaptation used:

- the first one is performing local adaptation and only adds points locally [32, 8, 33, 12],
- the second one is performing adaptation at the level of the hierarchical space $W_l$ (anisotropic sparse grid). This approach detects important dimensions that needs refinement and refines all the points in this dimension [11]. This refinement is also achieved in areas where the solution can be smooth. A more local version has been developed in [34].

In the current version of the library only dimension adaptation is available. Details on the algorithm can be found in [11]. After a first initialization with a first initialization with a space

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

A set of active level $A$ is created gathering all levels $l$ such that $\sum_{i=1}^d l_i = n + d - 1$. All other levels are gathered in a set $O$. At each level $l$ in $A$ an error is estimated $e_l$ and with all
while $E > \eta$ do
    select $l$ with the highest local error $e_l$
    $A = A \setminus \{l\}$
    $O = O \cup \{l\}$
    for $k = 1$ to $d$ do
        $m = l + e_k$
        if $m - e_q \in O$ for $q \in [1, d]$ then
            $A = A \cup \{m\}$
            Hierarchize all points belonging to $m$
            calculate $e_m$
            update $E$
        end if
    end for
end while

Algorithm 1: Dimension refinement for a given tolerance $\eta$

local error $e_l$ a global error $E$ is calculated. Then the refinement algorithm 1 is used noting $e_k$ the canonical basis in dimension $k$. Sometimes, using sparse grids during time iterations, it can be interesting to coarsen the meshes. A similar algorithm 2 can be used to eliminate levels with a very small local error.
\( \mathcal{B} \) all elements of \( \mathcal{A} \) with a local error below \( \eta \)

**while** \( \mathcal{B} \) non nonempty do

select \( l \in \mathcal{B} \) with the lowest local error \( e_l \)

for \( k = 1 \) to \( d \) do

\( m = l - e_k \)

if \( m_k > 0 \) then

if \( m + e_q \in \mathcal{B} \) for \( q \in [1, d] \) then

\( \mathcal{A} = \mathcal{A} \setminus \{ m + e_q, q \in [1, d] \} \)

\( \mathcal{B} = \mathcal{B} \setminus \{ m + e_q, q \in [1, d] \} \)

\( \mathcal{A} = \mathcal{A} \cup \{ m \} \)

Add \( m \) to \( \mathcal{B} \) if local error below \( \eta \)

\( \mathcal{O} = \mathcal{O} \setminus \{ m \} \)

Break

end if

end if

end for

if \( l \in \mathcal{B} \) then

\( \mathcal{B} = \mathcal{B} \setminus \{ l \} \)

end if

end while

**Algorithm 2**: Dimension coarsening for a given tolerance \( \eta \)

### 1.7 C++ API

The construction of the Sparse Grid including boundary point is done 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, const size_t &p_degree)
```

with

- \( p\text{lowValues} \) corresponds to the bottom of the grid,
- \( p\text{sizeDomain} \) corresponds to the size of the resolution domain in each dimension,
- \( p\text{levelMax} \) is the level of the sparse grids, the \( n \) in equation 1.7
- \( p\text{weight} \) the weight for anisotropic sparse grids, the \( w \) in equation 1.7
- \( p\text{degree} \) 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 done by the following constructor
The data structure of type `SparseSet` to store the sparse grid is defined by a map with keys an array \( A \) storing a multi level and values a map with keys an array \( B \) storing the multi index associated to a point \((A,B)\) and values the number of point \((A,B)\):

\[
\text{#define SparseSet std::map< Eigen::Array<char,Eigen::Dynamic,1>, std::map< Eigen::Array<unsigned int,Eigen::Dynamic,1>, size_t, OrderTinyVector< unsigned int >, OrderTinyVector< char > >, size_t , OrderTinyVector< unsigned int > >, OrderTinyVector< char > >}
\]

It is sometimes convenient to get back this data structure from the SparseGrid object: this is achieved by the following method:

\[
\text{std::shared_ptr<SparseSet> getDataSet() const ;}
\]

The previous two classes own two specific member functions to hierarchize (see section above) the value function known at the grids points for the whole grid.

- the first work on a single function:

\[
\begin{align*}
\text{/// \&\& Hierarchize a function defined on the grid} \\
\text{/// \&\& param p_toHierachize function to hierachize} \\
\text{void toHierarchize(Eigen::ArrayXd & p_toHierachize);} \\
\end{align*}
\]

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

\[
\begin{align*}
\text{/// \&\& Hierarchize a set of functions defined on the grid} \\
\text{/// \&\& param p_toHierachize set of functions to hierachize} \\
\text{void toHierarchizeVec(Eigen::ArrayXXd & p_toHierachize);}
\end{align*}
\]

The two classes own 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:

\[
\begin{align*}
\text{/// \&\& Hierarchize some points defined on the sparse grid} \\
\text{/// \&\& Hierarchization is performed point by point} \\
\text{/// \&\& param p_nodalValues function to hierarchize} \\
\text{/// \&\& param p_sparsePoints vector of sparse points to hierarchize (all points should belong to the dataset structure)} \\
\text{/// \&\& param p_hierarchized array of all hierarchized values (it is updated)} \\
\text{virtual void toHierarchizePByP(const Eigen::ArrayXd & p_nodalValues, const std::vector<SparsePoint> & p_sparsePoints, Eigen::ArrayXd & p_hierarchized) const}
\end{align*}
\]

- the second work on a matrix, permitting to hierarchize many functions in a single call (each row corresponds to a function representation)
Hierarchize some points defined on the sparse grids for a set of functions.

Hierarchization is performed point by point.

- **\param p\_nodalValues** functions to hierarchize (the row corresponds to the function number)
- **\param p\_sparsePoints** vector of sparse points to hierarchize (all points should belong to the dataset structure)
- **\param p\_hierarchized** array of all hierarchized values (it is updated)

```cpp
virtual void toHierarchizePByPVec(const Eigen::ArrayXXd & p\_nodalValues, const std::vector<SparsePoint> &p\_sparsePoints, Eigen::ArrayXXd &p\_hierarchized) const
```

The \textit{SparsePoint} object is only a type def:

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

where the first array permits to store the multi level associated to the point and the second the multi index associated.

At last it is possible to hierarchize all points associated to a multi level. As before two methods are available:

- a first permits to hierarchize all the points associated to a given level. Hierarchized values are updated with these new values.

```cpp
virtual void toHierarchizePByPLevel(const Eigen::ArrayXd & p\_nodalValues, const SparseSet::const_iterator &p\_iterLevel, Eigen::ArrayXd &p\_hierarchized) const
```

- the second permits to hierarchize differents functions together

```cpp
virtual void toHierarchizePByPLevelVec(const Eigen::ArrayXXd & p\_nodalValues, const SparseSet::const_iterator &p\_iterLevel, Eigen::ArrayXXd &p\_hierarchized) const
```
In the following example, the sparse grids with boundary points is constructed. The values of a function $f$ at each coordinates are stored in an array $valuesFunction$, storing 2 functions to interpolate. The 2 global functions are hierarchized (see section above) in the array $hierarValues$, and then the interpolation can be achieved using these hierarchized values.

```cpp
ArrayXd lowValues = ArrayXd::Zero(5); // bottom of the grid
ArrayXd sizeDomain = ArrayXd::Constant(5,1.); // size of the grid
ArrayXd weight = ArrayXd::Constant(5,1.); // weights
int degree = 1; // linear interpolator
bool bPrepInterp = true; // precalculate neighbors of nodes
level = 4; // level of the sparse grid

// sparse grid generation
SparseSpaceGridBound sparseGrid(lowValues, sizeDomain, level, weight, degree, bPrepInterp);

// grid iterators
shared_ptr/GridIterator> iterGrid = sparseGrid/GridIterator();
ArrayXXd valuesFunction(1, sparseGrid/NbPoints());
while (iterGrid->isValid())
{
    ArrayXd pointCoord = iterGrid/getCoordinate();
    valuesFunction(0, iterGrid/getCount()) = f(pointCoord);
    valuesFunction(1, iterGrid/getCount()) = f(pointCoord)+1;
    iterGrid/next();
}

// Hierarchize
SparseSpaceGridBound sparseGrid NghGrid(5, 0.66);
ArrayXXd hierarValues = valuesFunction;
sparseGrid.toHierarchizeVec(hierarValues);

// interpolate
ArrayXd pointCoord = ArrayXd::Constant(5, 0.66);
shared_ptr/Interpolatorder> interpolator = sparseGrid.createInterpolatorder(pointCoord);
ArrayXd interVal = interpolator/applyVec(hierarValues);
```

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

```cpp
std::vector/SparsePoint> sparsePoints(sparseGrid/NbPoints());
std::shared_ptr/SparseSet> dataSet = sparseGrid/DataSet();

// iterate on points
for (typename SparseSet::const_iterator iterLevel = dataSet/begin();
    iterLevel != dataSet/end(); ++iterLevel)
    for (typename SparseLevel::const_iterator iterPosition = iterLevel/second.begin();
        iterPosition != iterLevel/second.end(); ++iterPosition)
    {
        sparsePoints/iterPosition/second] = make_pair(iterLevel/first, iterPosition/first);
    }
```
In some cases, it is more convenient to construct an interpolator acting on a global function. It is the case when you have a single function and you want to interpolate at many points for this function. In this specific case an interpolator deriving from the class InterpolatorSpectral (similarly to Legendre grid interpolators) can be constructed:

```cpp
SparseInterpolatorSpectral(const shared_ptr<SparseSpaceGrid>& p_grid, const Eigen::ArrayXd &p_values)
```

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

```cpp
inline double apply(const Eigen::ArrayXd &p_point) const
```

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

```cpp
// sparse grid generation
SparseSpaceGridNoBound sparseGrid(lowValues, sizeDomain, p_level, p_weight, p_degree, bPrepInterp);
// test iterator on each level
ArrayXd valuesFunctionTest(sparseGrid.getNbPoints());
std::shared_ptr<SparseSet> dataSet = sparseGrid.getDataSet();
for (SparseSet::const_iterator iterLevel = dataSet->begin(); iterLevel != dataSet->end(); ++iterLevel)
{
    // get back iterator on this level
    shared_ptr<SparseGridIterator> iterGridLevel = sparseGrid.getLevelGridIterator(iterLevel);
    while(iterGridLevel->isValid())
    {
        Eigen::ArrayXd pointCoord = iterGridLevel->getCoordinate();
        valuesFunctionTest(iterGridLevel->getCount()) = f(pointCoord);
        iterGridLevel->next();
    }
}
```

At last adaptation can be realized with two member functions:

- A first one permits to refine adding points where the error is important. Notice that a function is provided to calculate from the 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 permits to specialize the refining depending for example if the calculation is achieved for integration or interpolation purpose.

```cpp
/// \brief Dimension adaptation nest
/// \param p_precision precision required for adaptation
/// \param p_fInterpol function to interpolate
/// \param p_phi function for the error on a given level in the m_dataSet structure
/// \param p_phiMult from an error defined on different levels, send back a global error on the different levels
/// \param p_valuesFunction an array storing the nodal values
/// \param p_hierarValues an array storing hierarchized values (updated)

void refine(const double &p_precision, const std::function<double(const Eigen::ArrayXd &p_x)>& p_fInterpol,
            const std::function<double(const SparseSet::const_iterator &, const Eigen::ArrayXd &)> &p_phi,
            const std::function<double(const std::vector<double> &)> &p_phiMult,
            Eigen::ArrayXd &p_valuesFunction,
            Eigen::ArrayXd &p_hierarValues);
```

with

- `p_precision` the $\eta$ 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 permits to coarsen the mesh, eliminating point where the error is too small

```cpp
/// \brief Dimension adaptation coarsening : modify data structure by trying to remove all levels with local error below a local precision
/// \param p_precision Precision under which coarsening will be realized
/// \param p_phi function for the error on a given level in the m_dataSet structure
/// \param p_valuesFunction an array storing the nodal values (modified on the new structure)
/// \param p_hierarValues Hierarchical values on a data structure (modified on the new structure)

void coarsen(const double &p_precision, const std::function<double(const SparseSet::const_iterator &, const Eigen::ArrayXd &)> &p_phi,
             const std::function<double(const std::vector<double> &)> &p_phiMult,
             Eigen::ArrayXd &p_valuesFunction,
             Eigen::ArrayXd &p_hierarValues);
```
with arguments similar to the previous function.

### 1.8 Python API

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

```python
# Copyright (C) 2016 EDF
# All Rights Reserved
# This code is published under the GNU Lesser General Public License (GNU LGPL)
import numpy as np
import unittest
import random
import math
import StOptGrids

# function used
def funcToInterpolate(x):
    return math.log(1. + x.sum())

# unit test for sparse grids
class testGrids(unittest.TestCase):

    # test sparse grids with boundaries
    def testSparseGridsBounds(self):
        # low values
        lowValues = np.array([1., 2., 3.])
        # size of the domain
        sizeDomValues = np.array([3., 4., 3.])
        # anisotropic weights
        weights = np.array([1., 1., 1.])
        # level of the sparse grid
        level = 3
        # create the sparse grid with linear interpolator
        sparseGridLin = StOptGrids.SparseSpaceGridBound(lowValues,
                                                        sizeDomValues,
                                                        level,
                                                        weights, 1)
        iterGrid = sparseGridLin.getGridIterator()
        # array to store
        data = np.empty(sparseGridLin.getNbPoints())
        # iterates on point
        while( iterGrid.isValid() ):
            data[iterGrid.getCount()] = funcToInterpolate(iterGrid.
                                                      getCoordinate())
            iterGrid.next()
        # Hierarchize the data
```

39
hierarData = sparseGridLin.toHierarchize(data)
# get back an interpolator
ptInterp = np.array([2.3, 3.2, 5.9], dtype=np.float)
interp = sparseGridLin.createInterpolator(ptInterp)
# calculate interpolated value
interpValue = interp.apply(hierarData)
print("Interpolated value sparse linear", interpValue)
# create the sparse grid with quadratic interpolator
sparseGridQuad = StOptGrids.SparseSpaceGridBound(lowValues, sizeDomValues, level, weights, 2)
# Hierarchize the data
hierarData = sparseGridQuad.toHierarchize(data)
# get back an interpolator
ptInterp = np.array([2.3, 3.2, 5.9], dtype=np.float)
interp = sparseGridQuad.createInterpolator(ptInterp)
# calculate interpolated value
interpValue = interp.apply(hierarData)
print("Interpolated value sparse quadratic", interpValue)
# now refine
precision = 1e-6
print("Size of hierarchical array", len(hierarData))
valueAndHierar = sparseGridQuad.refine(precision, funcToInterpolate, data, hierarData)
print("Size of hierarchical array after refinement", len(valueAndHierar[0]))
# calculate interpolated value
interp1 = sparseGridQuad.createInterpolator(ptInterp)
interpValue = interp1.apply(valueAndHierar[1])
print("Interpolated value sparse quadratic after refinement", interpValue)
# coarsen the grid
precision = 1e-4
valueAndHierarCoarsen = sparseGridQuad.coarsen(precision, valueAndHierar[0], valueAndHierar[1])
print("Size of hierarchical array after coarsening", len(valueAndHierarCoarsen[0]))
# calculate interpolated value
interp2 = sparseGridQuad.createInterpolator(ptInterp)
interpValue = interp2.apply(valueAndHierarCoarsen[1])
print("Interpolated value sparse quadratic after refinement", interpValue)

# test sparse grids eliminating boundaries
def testSparseGridsNoBounds(self):
    # low values
    lowValues = np.array([1., 2., 3.], dtype=np.float)
    # size of the domain
    sizeDomValues = np.array([3., 4., 3.], dtype=np.float)
    # anisotropic weights
    weights = np.array([1., 1., 1.])
    # level of the sparse grid
    level = 3
    # create the sparse grid with linear interpolator
sparseGridLin = StOptGrids.SparseSpaceGridNoBound(lowValues, sizeDomValues, level, weights, 1)
iterGrid = sparseGridLin.getGridIterator()
# array to store
data = np.empty(sparseGridLin.getNbPoints())
# iterates on point
while (iterGrid.isValid()):
    data[iterGrid.getCount()] = funcToInterpolate(iterGrid.getCoordinate())
    iterGrid.next()
# Hiearchize the data
hierarData = sparseGridLin.toHiearchize(data)
# get back an interpolator
ptInterp = np.array([2.3, 3.2, 5.9], dtype=np.float)
interpol = sparseGridLin.createInterpolator(ptInterp)
# calculate interpolated value
interpValue = interpol.apply(hierarData)
print("Interpolaed value sparse linear", interpValue)
# create the sparse grid with quadratic interpolator
sparseGridQuad = StOptGrids.SparseSpaceGridNoBound(lowValues, sizeDomValues, level, weights, 2)
# Hiearchize the data
hierarData = sparseGridQuad.toHiearchize(data)
# get back an interpolator
ptInterp = np.array([2.3, 3.2, 5.9], dtype=np.float)
interpol = sparseGridQuad.createInterpolator(ptInterp)
# calculate interpolated value
interpValue = interpol.apply(hierarData)
print("Interpolaed value sparse quadratic", interpValue)
# test grids function
iDim = sparseGridQuad.getDimension()
pt = sparseGridQuad.getExtremeValues()
# now refine
precision = 1e-6
print("Size of hierarchical array", len(hierarData))
valueAndHierar = sparseGridQuad.refine(precision, funcToInterpolate, data, hierarData)
print("Size of hierarchical array after refinement", len(valueAndHierar[0]))
# calculate interpolated value
interpol1 = sparseGridQuad.createInterpolator(ptInterp)
interpValue = interpol1.apply(valueAndHierar[1])
print("Interpolaed value sparse quadratic after coarsening", interpValue)
# coarsen the grid
precision = 1e-4
valueAndHierarCoarsen = sparseGridQuad.coarsen(precision, valueAndHierar[0], valueAndHierar[1])
print("Size of hierarchical array after coarsening", len(valueAndHierarCoarsen[0]))
# calculate interpolated value
interpol2 = sparseGridQuad.createInterpolator(ptInterp)
interpValue = interpol2.apply(valueAndHierarCoarsen[1])
print("Interpolaed value sparse quadratic after coarsening")
interpValue))

if __name__ == '__main__':
    unittest.main()
Chapter 2

Introducing the regression resolution

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

\[ dX^{x,t} = b(t, X^{x,t}_s)ds + \sigma(s, X^{x,t}_s)dW_s \]

This case is for example encountered while valuing American options in finance, when an arbitrage is realized between the pay off and the expected future gain if not exercising at the current time. In order to estimate this conditional expectation (depending on the Markov state), first suppose that a set of \( N \) Monte Carlo Simulation are 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 \( f \) lies in the infinite dimensional space of the \( L_2 \) functions. In order to approximate it, we try to find it in a finite dimensional space. Choosing a set of basis functions \( \psi_k \) for \( k = 1 \) to \( M \), the conditional expectation can be approximated by

\[ f(x) \simeq \sum_{k=1}^{M} \alpha_k \psi_k(X_t) \]  

(2.1)

where \( (\alpha^*_{t, N})_{k \leq M} \) minimizes

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

(2.2)

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

\[ \min_{\alpha \in \mathbb{R}^M} \| A\alpha - B \|^2 \]  

(2.3)

Classically the previous equation is reduced to the normal equation

\[ A'\alpha = A'B \]  

(2.4)

which is solved by a Cholesky like approach when the matrix \( A'A \) is definite otherwise the solution with the minimum \( L_2 \) norm can be computed.
2.1 C++ global API

All the regression classes derive 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 particle number), and stores if the current date $t$ is 0 (then the conditional expectation is only an expectation).

```cpp
// Copyright (C) 2016 EDF
// All Rights Reserved
// This code is published under the GNU Lesser General Public License (GNU LGPL)
#ifndef BASEREGRESSION_H
#define BASEREGRESSION_H
#include <memory>
#include <vector>
#include <Eigen/Dense>
#include "StOpt/core/grids/InterpolatorSpectral.h"

/** \file BaseRegression.h
 * \brief Base class to define regressor for stochastic optimization by Monte Carlo
 * \author Xavier Warin
 */
namespace StOpt
{
    /// \class BaseRegression BaseRegression.h
    /// Base class for regression
    class BaseRegression
    {
    protected :
        bool m_bZeroDate ; ///< Is the regression date zero ?
        std::shared_ptr<Eigen::ArrayXXd> m_particles ; ///< Particles used to regress: first dimension : dimension of the problem , second dimension : the number of particles
    public :
        /// \brief Default constructor
        BaseRegression() {}
        /// \brief Default destructor
        virtual ~BaseRegression() {}
        /// \brief Constructor storing the particles
        /// \param p_bZeroDate first date is 0?
        /// \param p_particles particles used for the meshes.
        /// First dimension : dimension of the problem,
        /// second dimension : the number of particles
        BaseRegression(const bool &p_bZeroDate , const std::shared_ptr<Eigen::ArrayXXd> &p_particles) : m_bZeroDate(p_bZeroDate) , m_particles(p_particles) {}
    }
}
```
BaseRegression(const bool &p_bZeroDate) : m_bZeroDate(p_bZeroDate) {}

BaseRegression(const BaseRegression &p_object) : m_bZeroDate(p_object.getBZeroDate()), m_particles(p_object.getParticles()) {}

void updateSimulationsBase(const bool &p_bZeroDate, const std::shared_ptr<Eigen::ArrayXXd> &p_particles) {
    m_bZeroDate = p_bZeroDate;
    m_particles = p_particles;
}

inline std::shared_ptr<Eigen::ArrayXXd> getParticles() const {
    return m_particles;
}

inline const Eigen::ArrayXXd &getParticlesRef() const {
    return *m_particles;
}

inline int getDimension() const {
    return m_particles->rows();
}

inline int getNbSimul() const {
    return m_particles->cols();
}

virtual int getNumberOfFunction() const = 0;

/// \brief Constructor storing the particles
virtual void updateSimulations(const bool &p_bZeroDate, const std::shared_ptr<Eigen::ArrayXXd> &p_particles) = 0;

virtual Eigen::ArrayXd getCoordBasisFunction(const Eigen::ArrayXd &p_fToRegress) const = 0;

virtual Eigen::ArrayXXd getCoordBasisFunctionMultiple(const Eigen::ArrayXXd &p_fToRegress) const = 0;

virtual Eigen::ArrayXd getAllSimulations(const Eigen::ArrayXd &p_fToRegress) const = 0;

virtual Eigen::ArrayXXd getAllSimulationsMultiple(const Eigen::ArrayXXd &p_fToRegress) const = 0;

virtual Eigen::ArrayXd reconstruction(const Eigen::ArrayXd &p_basisCoefficients) const = 0;

virtual Eigen::ArrayXXd reconstructionMultiple(const Eigen::ArrayXXd &p_basisCoefficients) const = 0;

virtual double reconstructionASim(const int &p_isim, const Eigen::ArrayXd &p_basisCoefficients) const = 0;

virtual double getValue(const Eigen::ArrayXd &p_coordinates,
All regression classes share the same constructors:

- a first constructor stores the members of the class and computes 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 some data which will be shared by all future regressions. It has to be used with the ‘updateSimulation’ method to update the effective 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 ‘updateSimulation’ method.

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 $\alpha_k$ of the basis functions,

- “getCoordBasisFunctionMultiple” is used if we want to do the previous calculation on multiple $g$ functions in one call. In the matrix given as 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. As output, the first dimension has a size equal to the number of function 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 $f(X_t)$

- “getAllSimulationMultiple” is used if we want to do the previous calculation on multiple $g$ functions in one call. In the matrix given as 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 given back in the same format.

- “reconstruction” takes the $\alpha_k$ coefficient of the basis functions as input and returns all the $f(X_t)$ for the simulations stored by applying equation (2.1).

- “reconstructionMultiple” is used if we want to do the previous calculation on multiple $g$ functions in one call. As input the $\alpha_k$ coefficients of the basis functions are given (number of function to regress for first dimension, number of basis functions for second dimension). As a result the $f(X_t)$ for all simulations and all $f$ functions are sent back (number of Monte Carlo simulations in first dimension, number of function to regress en second dimension).

- “reconstructionASim” takes a simulation number $i_{sim}$ (optimization part) and $\alpha_k$ coefficient of the basis functions as input and returns $f(X_{t,i_{sim}})$ by applying equation (2.1),

- “getValue” takes as first argument a sample of $X_t$, the basis function $\alpha_k$ and reconstruct the regressed solution of equation (2.1).

### 2.2 Adapted local polynomial basis

The description of the method and its properties can be found in [16]. We just recall the methodology

#### 2.2.1 Description of the method

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

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

48
This approximation is “non-conform” in the sense that we do not assure the continuity of the approximation. However, it has the advantage to be able to fit any, even discontinuous, function. In order to avoid oscillations and to allow classical regression by the Choleski method, the supports are chosen so that they contain roughly the same number of particles.

On Figure 2.1 we have plotted an example of supports in the case of $6 = 4 \times 4$ local basis cells, in dimension 2.

![Figure 2.1: Support of 2D function basis](image)

2.3 C++ api

2.3.1 The constant per cell approximation

The constructor of the local constant regression object is achieved by

```cpp
LocalConstRegression(const Eigen::ArrayXi &p_nbMesh);
```

where $p_{\text{nbMesh}}$ is an array giving the number of meshes used in each direction ( $(4, 4)$ for the figure 2.1, for example).

The second constructor permits the construct the regression matrix,

```cpp
LocalConstRegression(const bool &p_bZeroDate,
                     const shared_ptr<ArrayXXd> &p_particles,
                     const Eigen::ArrayXi &p_nbMesh);
```

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 directions (4, 4) for the figure 2.1.

### 2.3.2 The linear per cell approximation

The constructor of the local linear regression object is achieved by

```cpp
LocalLinearRegression (const Eigen::ArrayXi &p_nbMesh);
```

where $p_{nbMesh}$ is an array giving the number of meshes used in each direction (4, 4) for the figure 2.1 for example).

The second constructor permits the construct the regression matrix,

```cpp
LocalLinearRegression (const bool &p_bZeroDate ,
const shared_ptr<ArrayXXd> &p_particles, 
const Eigen::ArrayXi &p_nbMesh);
```

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 directions (4, 4) for the figure 2.1.

### 2.3.3 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.

```cpp
// create the mesh for a 2 dim problem, 4 meshes per direction
ArrayXi nbMesh = ArrayXi::Constant(2, 4);
// t is not zero
bool bZeroDate = 0;
// constructor
LocalLinearRegression localRegressor(nbMesh);
// update particles values
localRegressor.updateSimulations(bZeroDate, x);
// regressed values
ArrayXd regressedValues = localRegressor.getAllSimulations(toRegress);
```
### 2.4 Python API

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

```python
import StOptReg
nbSimul = 5000000;
np.random.seed(000)
x = np.random.uniform(-1.,1.,size=(1,nbSimul));
# real function
toReal = (2+x[0,:]+(x[0,:])*x[0,:])
# function to regress
toRegress = toReal + 4*np.random.normal(0.,nbSimul)
# mesh
nbMesh = np.array([6], dtype=np.int32)
regressor = StOptReg.LocalLinearRegression(False,x,nbMesh)
y = regressor.getAllSimulations(toRegress).transpose()[0]
```

Of course the constant per cell case in python is similar.

### 2.5 Local polynomial basis with meshes of same size

In some 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 $M = \prod_{k=1,d} I_k$. On each cell as in section 2.2, one can have two approximations:

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

### 2.6 C++ api

#### 2.6.1 The constant per cell approximation

The constructor of the local constant regression object is achieved by

```cpp
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 permits the construct the regression matrix,
LocalSameSizeConstRegression( const bool &p_bZeroDate ,
const std::shared_ptr< Eigen::ArrayXXd >& p_particles ,
const Eigen::ArrayXd &p_lowValues ,
const Eigen::ArrayXd &p_step ,
const Eigen::ArrayXi &p_nbStep );

where

- \textit{p\_bZeroDate} is true if the regression date is 0,
- \textit{p\_particles} the particles \( X_t \) for all simulations (dimension of \( X_t \) for first dimension, number of Monte Carlo simulations in second dimension),
- \textit{p\_lowValues} is an array giving the first point of the grid in each direction,
- \textit{p\_step} is an array giving the size of the meshes in each direction,
- \textit{p\_nbStep} is an array giving the number of meshes used in each direction.

### 2.6.2 The linear per cell approximation

The constructor of the local linear regression object is achieved by

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

where

- \textit{p\_lowValues} is an array giving the first point of the grid in each direction,
- \textit{p\_step} is an array giving the size of the meshes in each direction,
- \textit{p\_nbStep} is an array giving the number of meshes used in each direction.

The second constructor permits the construct the regression matrix,

LocalSameSizeLinearRegression( const bool &p_bZeroDate ,
const std::shared_ptr< Eigen::ArrayXXd >& p_particles ,
const Eigen::ArrayXd &p_lowValues ,
const Eigen::ArrayXd &p_step ,
const Eigen::ArrayXi &p_nbStep )

where

- \textit{p\_bZeroDate} is true if the regression date is 0,
- \textit{p\_particles} the particles \( X_t \) for all simulations (dimension of \( X_t \) for first dimension, number of Monte Carlo simulations in second dimension),
- \textit{p\_lowValues} is an array giving the first point of the grid in each direction,
- \textit{p\_step} is an array giving the size of the meshes in each direction,
- \textit{p\_nbStep} is an array giving the number of meshes used in each direction.
2.6.3 An example in the linear case

Below we give a small example where “toRegress” is the array to regress with respect to an array “x” in dimension \( p_n \text{Dim} \):

```cpp
// create a random 'x' array
shared_ptr<ArrayXXd> x(new ArrayXXd(ArrayXXd::Random(p_nDim, p_nbSimul)));

// create the mesh by getting min and max value on the samples
double xMin = x->minCoeff() - tiny;
double xMax = x->maxCoeff() + tiny;
ArrayXd lowValues = ArrayXd::Constant(p_nDim, xMin);
ArrayXd step = ArrayXd::Constant(p_nDim, (xMax - xMin) / p_nMesh);
ArrayXi nbStep = ArrayXi::Constant(p_nDim, p_nMesh);

// constructor
LocalLinearRegression localRegressor(lowValues, step, nbStep);
// update particles values
localRegressor.updateSimulations(bZeroDate, x);
// regressed values
ArrayXd regressedValues = localRegressor.getAllSimulations(toRegress);
```

2.7 Python API

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

```python
import StOptReg
nbSimul = 5000000;
np.random.seed(000)
x = np.random.uniform(-1.0, 1.0, size=(1, nbSimul));

# real function
toReal = (2+x[0,:]+(+x[0,:])*(1+x[0,:]))

# function to regress
toRegress = toReal + 4*np.random.normal(0.0, nbSimul)

# mesh
nStep = 20
lowValue = np.array([-1.0001], dtype=np.float)
step = np.array([2.0002/nStep], dtype=np.float)
nbMesh = np.array([nStep], dtype=np.int32)

# Regressor
regressor = StOptReg.LocalSameSizeLinearRegression(False, x, lowValue, step, nbMesh)
y = regressor.getAllSimulations(toRegress).transpose()[0]
```

Of course the constant per cell case in python is similar.

2.8 Sparse grid regressor

In the case of a sparse regressor, the grid is an object “SparseSpaceGridNoBound” (extrap-
oration for the boundary conditions). The basis functions are given by the section 1.3 for linear, quadratic or cubic function basis.
### 2.8.1 C++ API

Two specific constructors are available:

- The first one to be used with the “updateSimulations” methods:

```cpp
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 for anisotropic sparse grids (see equation (1.7)),
- `p_degree` is equal to (linear basis function), or 2 (quadratic basis) or 3 (for cubic basis functions),
- `p_bNoRescale` if true no re scaling of the particles is used. Otherwise a re scaling of the mesh size is achieved (as for local basis functions, see section 2.2)

- The second one 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 re scaling is always achieved):

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

A simple example to express the regression of “toRegress”

```cpp
// second member to regress
ArrayXd toRegress(p_nbSimul);
// for testing
toRegress.setConstant(.);
shared_ptr<ArrayXXd> x(new ArrayXXd(ArrayXXd::Random(p_nDim, p_nbSimul)));
// constructor: the current date is not zero
bool bZeroDate = 0;
// constructor
SparseRegression sparseRegressor(p_level, weight, p_degree);
sparseRegressor.updateSimulations(bZeroDate, x); // update the state
// then just calculate function basis coefficient
ArrayXd regressedFuntionCoeff = sparseRegressor.getCoordBasisFunction(toRegressor);
// use the getValue method to get back the regressed values
for (int is = 0; is < p_nbSimul; ++is)
{
    Map<ArrayXd> xloc(x->col(is).data(), p_nDim);
    double reg = sparseRegressor.getValue(xloc, regressedFuntionCoeff);
}
```
2.8.2 Python API

Here is a simple example of the python API:

```python
import StOptReg

nbSimul = 2000000;
np.random.seed(000)
x = np.random.uniform(-1., 1., size=(1, nbSimul));

# real function
toReal = (2+x[0,:]+(1+x[0,:])*(1+x[0,:]))

# function to regress
toRegress = toReal + 4*np.random.normal(0., nbSimul)

# level for sparse grid
iLevel = 5;

# weight for anisotropic sparse grids
weight= np.array([], dtype=np.int32)

# Regressor degree
regressor = StOptReg.SparseRegression(False, x, iLevel, weight, )
y = regressor.getAllSimulations(toRegress)
```

2.9 Global polynomial basis

2.9.1 Description of the method

In this section, the $\psi_k(X_t)$ involved in equation 2.1 are some given polynomials. Available polynomials are the canonical one, the Hermite and the Chebyshev ones.

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

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

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

$$a_{n+1,k} = a_{n,k-1} - na_{n-1,k}, k > 0$$
$$a_{n+1,0} = -na_{n-1,0}$$

(2.5) (2.6)

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

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

They satisfy the following recurrence :

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

The “GlobalRegression” class is template by the type of the polynomial (“Canonical”, “Tchebychev” or “Hermite”) The first constructor:

```cpp
GlobalRegression(const int & p_degree, const int & p_dim);
```

where `p_degree` is the total degree of the polynomial approximation, `p_dim` is the dimension of the problem.

A second constructor is provided:

```cpp
GlobalRegression(const bool & p_bZeroDate,
                 const std::shared_ptr<Eigen::ArrayXXd> & p_particles,
                 const int & p_degree);
```

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.

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.

```cpp
// total degree equal to 2
int degree = 2;
// t is not zero
bool bZeroDate = 0;
// constructor with Hermite polynomials
GlobalRegression<Hermite> localRegresser(degree, x.rows());
// update particles values
localRegresser.updateSimulations(bZeroDate, x);
// regressed values
ArrayXd regressedValues = localRegresser.getAllSimulations(toRegress);
```

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

```cpp
GlobalRegression<Canonical> localRegresser(degree, x.rows());
```

or by a Chebyshev one:

```cpp
GlobalRegression<Tchebychev> localRegresser(degree, x.rows());
```

2.9.3 Python API

Here is a similar example using the second constructor

```python
import StOptReg
nbSimul = 5000000;
np.random.seed(000)
x = np.random.uniform(-1, 1, size=(1, nbSimul));
# real function
```
toReal = \( (2 + x[0,:]) + (x[0,:]) \times (1 + x[0,:]) \)

# function to regress
toRegress = toReal + 4*\text{np.random.normal}(0.,\text{nbSimul})

# degree
degree = 2

# Regressor
regressor = \text{StOptReg.GlobalHermiteRegression}(\text{False}, x, degree)
y = regressor.get\text{AllSimulations}(toRegress).\text{transpose()}[0]

Available regressors are “GlobalHermiteRegression” as in the example above, “GlobalCanonicalRegression” and “GlobalTchebychevRegression” with an obvious correspondence.
Chapter 3

Continuation values objects and similar ones

In a first part we describe a way to store and use continuation values calculated during the use of 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 realized after the optimization part of a problem.

3.1 Continuation values object

A special case is the case where the state \( X^{x,t} \) in equation (1) can be separated into two parts \( X^{x,t} = (X^{x,t}_1, X^{x,t}_2) \) where

1. the first part is given by the following equation

\[
dX^{x,t}_1 = b(t, X^{x,t}_s)ds + \sigma(t, X^{x,t}_s)dW_s
\]

and is not controlled: the stochastic process is exogenous,

2. the second part is given by the following equation

\[
dX^{x,t}_2 = a(t)ds
\]

such that the \( X^{x,t}_2 \) is a degenerated version of \( X^{x,t}_1 \) without diffusion, \( a \) representing the control.

This first case is for example encountered while valuing American options in finance. In this case, \( X^{x,t}_1 \) holds the values of the stocks involved in the option and \( X^{x,t}_2 \) is for example an integer valued process equal to one if the option is not exercised and zero if it has already been exercised.

Another classical case happening while dealing with stocks for example is a Gas Storage valuation. In this simple case, the process \( X^{x,t}_1 \) is the value of the gas on the market and \( X^{x,t}_2 \) is the position (in volume) in the gas storage. The library offers to store the conditional expectation for all the states \( X^{x,t}_2 \).
• $X^{x,t}_2$ will be stored on a grid of points (see section [1]).

• For each point $i$ of the grid the conditional expectation of a function $g_i(X^{x,t}_2)$ associated to the point $i$ using a regressor (see section [1]) can be calculated and stored such that the continuation value $C$ is a function of $(X^{x,t}_1, X^{x,t}_2)$.

### 3.1.1 C++ API

As for regressions two constructors are provided:

• The first one is the default construction: it is used in simulation algorithm with the “loadForSimulation” method to store the basis coefficients $\alpha^i_k$ for the grid point $i$ (see equation (2.1)),

• The second one

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

with

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

This constructor constructs for all point $i$ all the $\alpha^i_k$ (see equation (2.1)).

The main methods provided are:

• A first method used in simulation permitting to load for grid point $i$ the coefficient $\alpha^i_k$ associated to the function $g_i$,

```cpp
void loadForSimulation (const shared_ptr< SpaceGrid > & p_grid, 
                        const shared_ptr< BaseRegression > & p_condExp, 
                        const Eigen::ArrayXXd & p_values)
```

with

- `p_grid` the grid associated to the controlled deterministic space,
- `p_condExp` the conditional expectation operator,
- `p_values` the $\alpha^i_k$ for all grid points $i$ (size the number of function basis, 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:
Eigen::ArrayXd getAllSimulations(const Eigen::ArrayXd &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 construct the interpolator:

Eigen::ArrayXd getAllSimulations(const Interpolator &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:

double getASimulation(const int &p_isim, const Eigen::ArrayXd &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:

double getASimulation(const int &p_isim, const Interpolator & p_interpol)

- a method that permits to calculate the conditional expectation for a sample of $X_{x,t}^1$:

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 $X_{x,t}^2$)
- $p_{.coordinates}$ the sample of $X_{x,t}^1$ used to estimate the conditional expectation
- and the function returns $C(X_{1,x,t}^1, X_{2,x,t}^2)$.

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

```cpp
int sizeForStock = 4;
// second member to regress with one stock
ArrayXXd toRegress = ArrayXXd::Constant(p_nbSimul, sizeForStock, 1.);
// grid for stock
Eigen::ArrayXd lowValues(1), step(1);
lowValues(0) = 0.;
step(0) = 1;
Eigen::ArrayXi nbStep(1);
bStep(0) = sizeForStock - 1;
// grid
shared_ptr<RegularSpaceGrid> regular = MyMakeShared<
    RegularSpaceGrid>(lowValues, step, nbStep);
// conditional expectation (local basis functions)
```
ArrayXi nbMesh = ArrayXi::Constant(p_nDim, p_nbMesh);
shared_ptr<LocalLinearRegression> localRegressor = MyMakeShared<
    LocalLinearRegression>(false, x, nbMesh);

    // creation continuation value object
ContinuationValue continuation(regular, localRegressor, toRegress);

    // regress with continuation value object
ArrayXd ptStock(1);
ptStock(0) = sizeForStock / 2; // point where we regress
    // calculation the regression values for the current point for all
    // the simulations
ArrayXd regressedByContinuation = continuation.getAllSimulations(
    ptStock);

3.1.2 Python API

Here is an example of the use of the mapping

# Copyright (C) 2016 EDF
# All Rights Reserved
# This code is published under the GNU Lesser General Public License (GNU
# LGPL)
import numpy as np
import unittest
import random
import math
import StOptGrids
import StOptReg

# unit test for continuation values
class testContValues(unittest.TestCase):

    # test a regular grid for stocks and a local function basis for
    # regression
    def testSimpleGridsAndRegressor(self):
        # low value for the meshes
        lowValues = np.array([1., 2., 3.], dtype=np.float)
        # size of the meshes
        step = np.array([0.7, 2.3, 1.9], dtype=np.float)
        # number of steps
        nbStep = np.array([3, 2, 4], dtype=np.int32)
        # create the regular grid
        grid = StOptGrids.RegularSpaceGrid(lowValues, step, nbStep)
        # simulation
        nbSimul = 10000
        np.random.seed(1000)
        x = np.random.uniform(-1., 1., size=(1, nbSimul));
        # mesh
        nbMesh = np.array([16], dtype=np.int32)
```python
# Create the regressor
# LocalLinearRegression(False,x,nbMesh)
regressor = StOptReg.LocalLinearRegression(False,x,nbMesh)

# regressed values
toReal = (2+x[0,:]+(1+x[0,:])*x[0,:])

# function to regress
toRegressor = toReal + 4*np.random.normal(0.,1.,nbSimul)
# create a matrix (number of stock points by number of simulations)
toRegressorMult = np.zeros(shape=(len(toRegressor),grid.getNbPoints()))
for i in range(toRegressorMult.shape[1]):
    toRegressorMult[:,i] = toRegressor

# Now create the continuation object
contOb = StOptReg.ContinuationValue(grid,regressor,toRegressorMult)

# get back the regressed values at the point stock
ptStock = np.array([1.2,3.1,5.9],dtype=np.float)
regressValues = contOb.getAllSimulations(ptStock)
```

### 3.2 The GridAndRegressedValue object

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

#### 3.2.1 C++ API

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

- The first one

  ```cpp
  GridAndRegressedValue(const std::shared_ptr<SpaceGrid> &p_grid,
            const std::shared_ptr<BaseRegression> &
            p_reg,
            const Eigen::ArrayXXd &p_values)
  ```

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

- A second constructor only stores the grid and regressor:

  ```cpp
  GridAndRegressedValue(const std::shared_ptr<SpaceGrid> &p_grid,
            const std::shared_ptr<BaseRegression> &p_reg)
  ```
The main methods are the following ones:

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

```cpp
double getValue(const Eigen::ArrayXd &ptOfStock, const Eigen::ArrayXd &coordinates) const
```

where:

- `ptOfStock` $X_{2,s}^{x,t}$ part of $X_{s}^{x,t}$
- `coordinates` $X_{1,s}^{x,t}$ part of $X_{s}^{x,t}$.

- the method ‘getRegressedValues’ that permits to get all regression coefficients for all points of the grid. The array returned has a size (number of function basis, number of points on the grid)

```cpp
Eigen::ArrayXXd getRegressedValues() const
```

- the method ‘setRegressedValues’ permits to store all the values regressed coefficients on a grid of a function of $X_{s}^{x,t} = (X_{1,s}^{x,t}, X_{2,s}^{x,t})$.

```cpp
void setRegressedValues(const Eigen::ArrayXXd &regValues)
```

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

### 3.2.2 Python API

The python API is similar to the one of the ContinuationValue object (voir section [3.1.2]).
Part III

Solving optimization problems with dynamic programming methods
In the sequel, we suppose that we have developed a Simulator generating some Monte Carlo simulations at the different optimization dates. In order to use the different frameworks developed in the sequel we suppose that the Simulator is derived from the abstract class “SimulatorDPBase.h”

```cpp
// Copyright (C) 2016 EDF
// All Rights Reserved
// This code is published under the GNU Lesser General Public License (GNU LGPL)
#ifndef SIMULATORDPBASE_H
#define SIMULATORDPBASE_H
#include <Eigen/Dense>

namespace StOpt
{
    // Abstract class for simulator used in dynamic programming
    class SimulatorDPBase
    {
    public:
        virtual ~SimulatorDPBase() {}
        virtual Eigen::MatrixXd getParticles() const = 0;
        virtual void stepForward() = 0;
        virtual void stepBackward() = 0;
        virtual Eigen::MatrixXd stepForwardAndGetParticles() = 0;
        virtual Eigen::MatrixXd stepBackwardAndGetParticles() = 0;
        virtual int getDimension() const = 0;
        virtual int getNbStep() const = 0;
        virtual double getStep() const = 0;
        virtual double getCurrentStep() const = 0;
        virtual int getNbSimul() const = 0;
    }
};
```

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

- 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 simulation of the asset from the last date to time 0. This method is used during an asset optimization by Dynamic Programming,

- the “stepForwardAndGetParticles” method: second and first method in one call,

- the “stepBackwardAndGetParticles” method: third and first method in one call,

- the “getDimension” method returns the number of assets,

- the “getNbStep” method returns the number of step ($N$),

- the “getStep” method returns the time step $t_{i+1} - t_i$ at the current time $t_i$,

- the “getNbSimul” method returns $M$.

- the “getActuStep” method return the actualization factor on one time step

- the “getActu” method returns an actualization factor at the “0” date.
Chapter 4

Using conditional expectation estimated by regressions to solve simple problems

In this chapter we give some examples to value an American option. This use of the conditional expectation operators can be extended to many stochastic problem using this previously developed objects.

4.1 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

$$P_t = \operatorname{esssup}_{\tau \in \mathcal{T}_{[t,T]}} \mathbb{E}(g(\tau, X_\tau) \mid \mathcal{F}_t) \quad \text{for } t \leq T \quad \mathbb{P} - \text{a.s. }, \quad (4.1)$$

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

We recall the classical Longstaff Schwartz algorithm estimating the empirical conditional expectation using the regression estimation previously seen.

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

Backward induction:
for $i = \kappa - 1$ to 0 do
set $\hat{\tau}_{i+1, \pi} := t_i 1_{A_i} + \hat{\tau}_{i+1, \pi} 1_{(A_i)^c}$ where $A_i := \{g(t_i, X_{t_i}) \geq \hat{\mathbb{E}}[g(\hat{\tau}_{i+1, \pi}, X_{\hat{\tau}_{i+1, \pi}}) \mid \mathcal{F}_{t_i}]\}$.
end for

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

Algorithm 3: Algorithm with regression [optimal exercise time estimation]
4.1.1 American option 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 \( p\_payoff \):

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

4.2 American option with the Python API

Using the python API the American resolution is given below:

```python
# Copyright (C) 2016 EDF
# All Rights Reserved
# This code is published under the GNU Lesser General Public License (GNU LGPL)
import numpy as np
import math as maths

# american option by Longstaff–Schwartz
# p_sim Monte Carlo simulator
# p_payOff Option pay off
# p_regressor regressor object
def resolution(p_simulator, p_payOff, p_regressor):
    step = p_simulator.getStep()
    # asset simulated under the neutral risk probability: get the trend of
    # first asset to get interest rate
    expRate = np.exp(-step * p_simulator.getMu()[0])
    # Terminal
    particle = p_simulator.getParticles()
    Cash = p_payOff.operator(particle)
    for iStep in range(0, p_simulator.getNbStep()):
```
asset = p_simulator.stepBackwardAndGetParticles()
payOffLoc = p_payOff.operator(asset)
isLastStep = False
if iStep == p_simulator.getNbStep() - 1:
    isLastStep = True

p_regressor.updateSimulations(isLastStep, asset)

# conditional expectation
condEspec = p_regressor.getAllSimulations(Cash).squeeze() * expRate
# arbitrage
Cash = np.where(condEspec < payOffLoc, payOffLoc, Cash * expRate)

return maths.fsum(Cash) / len(Cash)
Chapter 5

Using the general framework to manage stock problems

In this chapter the state is separated into three parts $X^{x,t} = (X_{1}^{x,t}, X_{2}^{x,t}, I_{t})$, which corresponds to the special case of chapter 3 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 some integers values and is here to describe some finite discrete regimes (to treat some switching problems). A general framework is available to solve this kind of problem. First, the second part $X_{2}^{x,t}$ is discretized on a grid as explained in chapter 3.

- Either a full grid is used for $X_{2}^{x,t}$ and two types of resolutions either sequential or parallel be can considered:
  - a resolution can be achieved sequentially or a parallelization with MPI on the calculations can be achieved (speed up but no size up). This approach can be used for problems in small dimension.
  - a resolution can be achieved with a parallelization by the MPI framework by spreading the work to be achieved on the grid points, and spread the data between processors (speed up and size up). We will denote this parallelization technique a “distribution” technique. This approach is necessary to tackle very big optimization problems where the global solution cannot be stored in the memory of a single processor.

- or the grid for $X_{2}^{x,t}$ is not full (so sparse) and only a parallelization by thread and MPI can be achieved on the calculations (speed up and no size up). With sparse grids, only the case $X_{2}^{x,t}$ deterministic is treated.

In the case of the MPI parallelization technique distributing task and data (full grids only), [19] and [20] are used. Suppose that the grid is the same at each time step (only here to ease the case), and that we have 4 processors (figure 5.1) then:
• at the last time step, the final values at each point for each simulation are computed
  (each processor computes the values for its own grid points),

• at the previous time step, from a grid point own by a processor, we are able to localize
  the grids points attained at the next time step by all the commands,

• on figure 5.1 we give the points owned by other processors that can be reached from
  points owned by processor 3,

• some MPI communications are achieved bringing back the data (values calculated at
  the previous treated time step) needed by processor 3 to be able to update the value
  calculated by dynamic programming at the current time for all the points owned by
  processor 3,

• all the communications between all processors are achieved together.

![Figure 5.1: Data to send to processor 3](image)

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

### 5.1 General requirement about business object

In order to use the framework, the developer has to describe the problem he wants to solve
on one time step staring from a state $X^{x,t}$. This business object has to offer some common
methods and it is derived from “OptimizerBase.h”

```c
// Copyright (C) 2016 EDF
// All Rights Reserved
// This code is published under the GNU Lesser General Public License (GNU LGPL)
#ifndef OPTIMIZERBASE_H
#define OPTIMIZERBASE_H
#include <Eigen/Dense>
#include "StOpt/core/utils/StateWithStocks.h"
```
```cpp
#include "StOpt/core/grids/SpaceGrid.h"
#include "StOpt/regression/BaseRegression.h"
#include "StOpt/regression/ContinuationValue.h"
#include "StOpt/regression/GridAndRegressedValue.h"
#include "StOpt/dp/SimulatorDPBase.h"

/** \file OptimizerBase.h
 * \brief Define an abstract class for Dynamic Programming problems solved by Monte Carlo methods
 * \author Xavier Warin */

namespace StOpt {

/// \class OptimizerBase OptimizerBase.h
/// Base class for optimizer for Dynamic Programming with and without regression methods

class OptimizerBase { 

public:

    OptimizerBase () {}

    virtual ~OptimizerBase () {}

    /// \brief defines the dimension to split for MPI parallelism
    /// For each dimension return true is the direction can be split
    virtual Eigen::Array< bool , Eigen::Dynamic, 1> getDimensionToSplit () const = 0 ;

    /// \brief defines the diffusion cone for parallelism
    /// \param p_regionByProcessor region (min max) treated by the processor for the different regimes treated
    /// \return returns in each dimension the min max values in the stock that can be reached from the grid p_gridByProcessor for each regime
    virtual std::vector< std::array< double , 2> > getCone ( const std::vector< std::array< double , 2> > &p_regionByProcessor ) const = 0 ;

    /// \brief Defines a step in simulation using interpolation in controls
    /// \param p_grid grid at arrival step after command
    /// \param p_control defines the controls
    /// \param p_state defines the state value (modified)
    /// \param p_phiInOut defines the value function (modified): size number of functions to follow
    virtual void stepSimulateControl ( const std::shared_ptr< StOpt::SpaceGrid> &p_grid , const std::vector< StOpt::GridAndRegressedValue > & p_control ,
                                      StOpt::StateWithStocks &p_state ,
                                      Eigen::Ref< Eigen::ArrayXd > p_phiInOut ) const = 0 ;

};
```
We detail all the methods that have to be implemented for all resolution methods (with or without regressions).

- the “getNbRegime” permits to get the number of regimes of the problem: for example, in switching problems, when there is a cost of switching, the working regime has to be incorporated in the state. Another example is the case of conditional delta to calculate for an asset: two regimes can be used: one to calculate the asset value and the second one to calculate the $\Delta$. This number of regimes can be time dependent: in this case for a current resolution date $t$ the “getNbRegime” method send the number of regimes at the very beginning of the time step (in $t^-$) such that a switch to a new regime can occurred in $t^+$.

- the “getSimulator” method is used to get back 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 the wealth obtained while 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 take a vector of size the dimension of the grid. Each component of the vector is an array containing the minimal and maximal coordinates values of points in the current grid defining an hyper cube $H_1$. It returns for each dimension, the coordinates min and max of the hyper cube $H_2$ containing the points that can be reached by applying a command from a grid point in $H_1$.

- the “getDimensionToSplit” method permits to define in the MPI framework with distribution which directions to split for solution distribution on 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 optimization using the optimal controls calculated in the optimization part. From a state \( p_{\text{state}} \) (storing the \( X^{x,t} \)), the optimal control calculated in optimization \( p_{\text{control}} \), the optimal functions values along the current trajectory are stored in \( p_{\phi \text{InOut}} \). The state \( p_{\text{state}} \) is updated during at the end of the call function.

In a first part we present the framework for problems where conditional expectation is calculated by regression (case where \( X_2^{x,t} \) is not controlled). Then we develop the framework not using regression for conditional expectation calculations. All conditional expectation are calculated using exogenous particles and interpolation. This will be typically the case for portfolio optimization.

5.2 Solving the problem using conditional expectation calculated by regressions

In this part we suppose that \( X_2^{x,t} \) is controlled and deterministic so regression methods can be used.

5.2.1 Requirement to use the framework

In order to use the framework with regression for conditional expectation, a business object describing the business on one time step from one state is derived from “OptimizerDPBase.h” itself derived from “OptimizerBase.h”.

```cpp
// Copyright (C) 2016 EDF
// All Rights Reserved
// This code is published under the GNU Lesser General Public License (GNU LGPL)

#ifndef OPTIMIZERDPBASE_H
#define OPTIMIZERDPBASE_H

#include <Eigen/Dense>
#include "StOpt/core/utils/StateWithStocks.h"
#include "StOpt/core/grids/SpaceGrid.h"
#include "StOpt/regression/BaseRegression.h"
#include "StOpt/regression/ContinuationValue.h"
#include "StOpt/regression/GridAndRegressedValue.h"
#include "StOpt/dp/SimulatorDPBase.h"
#include "StOpt/dp/OptimizerBase.h"

namespace StOpt

/** \file OptimizerDPBase.h
 * \brief Define an abstract class for Dynamic Programming problems solved by regression methods
 * \author Xavier Warin
 */
```
/** \class OptimizerDPBase OptimizerDPBase.h
 * \brief Base class for optimizer for Dynamic Programming with regression methods
 * \class OptimizerDPBase : \public OptimizerBase
 */

public:

OptimizerDPBase() {}

virtual ~OptimizerDPBase() {}\n
/** \brief defines the diffusion cone for parallelism
 * \param p_regionByProcessor region (min max) treated by the processor for the different regimes treated
 * \return returns in each dimension the min max values in the stock that can be reached from the grid p_gridByProcessor for each regime
 */
virtual std::vector<std::array<double, 2>> getCone(const std::vector<std::array<double, 2>>& p_regionByProcessor) const = 0;

/** \brief defines the dimension to split for MPI parallelism
 * \brief For each dimension return true if the direction can be split
 */
virtual Eigen::Array<bool, Eigen::Dynamic, 1> getDimensionToSplit() const = 0;

/** \brief defines a step in optimization
 * \param p_grid grid at arrival step after command
 * \param p_stock coordinates of the stock point to treat
 * \param p_condEsp continuation values for each regime
 * \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)
 * \return a pair:
 * - for each regimes (column) gives the solution for each particle (row)
 * - for each control (column) gives the optimal control for each particle (rows)
 */
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::ContinuationValue>& p_condEsp,
        const std::vector<std::shared_ptr<Eigen::ArrayXXd>>& p_phiIn)
        const = 0;

/** \brief defines a step in simulation
 * \brief Notice that this implementation is not optimal but is convenient if the control is discrete.
 * \brief By avoiding interpolation in control we avoid non admissible control
 * \brief Control are recalculated during simulation.
 */
We detail the different methods to implement in addition to the methods of “OptimizerBase.h”:

- the “stepOptimize” methods is used in optimization. We want to calculate the optimal value at current $t_i$ at a grid point $p_{stock}$ using a grid $p_{grid}$ at the next date $t_{i+1}$,
the continuation values for all regimes $p\_condEsp$ permitting to calculate conditional expectation of the optimal value function calculated at the previously 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

- first element is a matrix (first dimension is the number of simulations, second dimension the number of regimes) giving the function value,
- second element is a matrix (first dimension is the number of simulations, second dimension the number of controls) giving the optimal control.

- the “stepSimulate” method is used after optimization using the continuation values calculated in the optimization part. From a state $p\_state$ (storing the $X^{x,t}$), the continuation values calculated in optimization $p\_continuation$, the optimal functions values along the current trajectory are stored in $p\_phiInOut$.

In the case of a gas storage [21], the holder of the storage can inject gas from the network in the storage (paying the market price) or withdraw gas from the storage on the network (receiving the market price). In this case the Optimize object is given in [this file](#). You can have a look at the implementation of the “getCone” method.

### 5.2.2 The framework in optimization

Once an Optimizer is derived for the project, and supposing that a full grid is used for the stock discretization, the framework provides a “TransitionStepRegressionDPDist” object in MPI that permits to solve the optimization problem with distribution of the data on one time step with the following constructor:

```cpp
TransitionStepRegressionDPDist ( const shared_ptr<FullGrid> & p_pGridCurrent ,
                                 const shared_ptr<FullGrid> & p_pGridPrevious ,
                                 const shared_ptr<OptimizerDPBase > & p_pOptimize ) :
```

with

- $p\_pGridCurrent$ is the grid at the current time step ($t_i$),
- $p\_pGridPrevious$ is the grid at the previously treated time step ($t_{i+1}$),
- $p\_pOptimize$ the optimizer object

**Remark 6** A similar object is available without the MPI distribution framework “TransitionStepRegressionDP” with still enabling parallelization with threads and MPI on the calculations on the full grid points.

**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
\begin{verbatim}
std::vector< shared_ptr< Eigen::ArrayXXd > > OneStep(const std::vector<
    shared_ptr< Eigen::ArrayXXd > > &p.phiIn,
    const shared_ptr< BaseRegression> &p_condExp)
\end{verbatim}

with

- \textit{p\_phiIn} the vector (its size corresponds to the number of regimes) of matrix of optimal values calculated at the previous time iteration for each regime. Each matrix is a number of simulations by number of stock points matrix.

- \textit{p\_condExp} the conditional expectation operator,

returning 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 simulations by number of stock points matrix).

- 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 simulations by number of stock points matrix).

\textbf{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:

\begin{verbatim}
void dumpContinuationValues(std::shared_ptr<gs::BinaryFileArchive> p_ar,
    const std::string &p_name, const int &p_iStep,
    const std::vector< std::shared_ptr< Eigen::ArrayXXd > > &p_phiInPrev,
    const std::vector< std::shared_ptr< Eigen::ArrayXXd > > &p_control,
    const std::shared_ptr< BaseRegression> &p_condExp,
    const bool &p_bOneFile) const
\end{verbatim}

with:

- \textit{p\_ar} is the archive where controls and solutions are dumped,

- \textit{p\_name} is a base name used in the archive to store the solution and the control,

- \textit{p\_phiInPrev} is the solution at the previous time step used to calculate the continuation values that are stored,

- \textit{p\_control} stores the optimal controls calculated at the current time step,

- \textit{p\_condExp} is the conditional expectation object permitting to calculate conditional expectation of functions defined at the previous time step treated \textit{p\_phiInPrev} and permitting to store a representation of the optimal control.
• $p_{\text{bOneFile}}$ 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 by processor). If the problem gives continuation and 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 continuation/control values in one file for the simulation of the optimal policy.

Remark 9 As for the “TransitionStepRegressionDP” and the “TransitionStepRegressionDPDist” object, their “dumpContinuationValues” doesn’t need a $p_{\text{bOneFile}}$ argument: obviously optimal controls and solutions are stored in a single file.

We give here a simple example of a time resolution using this method when the MPI distribution of data is used

```cpp
using namespace std;

double DynamicProgrammingByRegressionDist(const shared_ptr<StOpt::FullGrid>& p_grid,
                                          const shared_ptr<StOpt::OptimizerDPBase>& p_optimize,
                                          shared_ptr<StOpt::BaseRegression>& p_regressor,
                                          const function<
double(const int &, const Eigen::ArrayXd &, const Eigen::ArrayXd &)
                                          &p_funcFinalValue,
                                          const Eigen::ArrayXd &p_pointStock,
                                          const int &p_initialRegime,
                                          const string &p_fileToDump,
                                          const bool &p_bOneFile)
{
    // from the optimizer get back the simulator
    shared_ptr<StOpt::SimulatorDPBase> simulator = p_optimize->getSimulator();
    // final values
```
Table 5.1: Which “TransitionStepRegression” object to use depending on the grid used and the type of parallelization used.

<table>
<thead>
<tr>
<th>Parallelization</th>
<th>Full grid</th>
<th>Sparse grid</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequential</td>
<td>“TransitionStepRegressionDP”</td>
<td>“TransitionStepRegressionDP_Sparse”</td>
</tr>
<tr>
<td>Parallelization on calculations threads and MPI</td>
<td>“TransitionStepRegressionDP”</td>
<td>“TransitionStepRegressionDP_Sparse”</td>
</tr>
<tr>
<td>Distribution of calculations and data</td>
<td>“TransitionStepRegressionDP_Dist”</td>
<td>Not available</td>
</tr>
</tbody>
</table>

An example without distribution of the data can be found in this file. We give at last a table with the different “TransitionStepRegression” objects to use depending on the type of parallelization used.
5.2.3 The framework in simulation

Once the optimization has been achieved, continuation values are dumped in one file (or some files) at each time step. In order to simulate the optimal policy, we can use the continuation values previously calculated (see chapter 3) or we can use the optimal controls calculated in optimization. In continuous optimization, using the control is more effective in term of computational cost. When the control is discrete, interpolation of the controls can lead to non admissible controls and simulation with the value function is more accurate.

While simulating the optimal control, two cases can occur:

- For most of the case (small dimensional case), the optimal control or the optimal function value can be stored in the memory of the computing node and function values and controls are stored in a single file. In this case a simulation of the optimal control can easily be achieved by distributing the Monte Carlo simulations on the available calculations nodes: this can be achieved by using the “SimulateStepRegression” or “SimulateStepRegressionControl” objects at each time step of the simulation.

- When dealing with very large problems, optimization is achieved by distributing the data on the processors and it is impossible to store the optimal command on one node. In this case, optimal controls and optimal solutions are stored in the memory of the node that has been used to calculate them in optimization. Simulations are reorganized at each time step and gathered so that they occupy the same part of the global grid. Each processor will then get from other processors a localized version of the optimal control or solution that 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 big 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

```cpp
SimulateStepRegressionDist(gs::BinaryFileArchive &p_ar, const int &p_iStep, const std::string &p_nameCont,
                          const shared_ptr<FullGrid> &p_GridFollowing, const shared_ptr<
                          OptimizerDPBase> &p_pOptimize,
                          const bool &p_bOneFile)
```

where

- `p_ar` is the binary archive where the continuation values are stored,
- `p_iStep` is the number associated to the current time step (0 at the beginning date of simulation, the number is increased by one at each time step simulated),
- `p_nameCont` is the base name for continuation values,
- `p_GridFollowing` is the grid at the next time step (`p_iStep + 1`),
Optimize the Optimizer describing the transition from one time step to the following one,

- \( p_{\text{OneFile}} \) equal to true if a single archive is used to store continuation values.

Remark 10 A version without distribution of data but with multithreaded and with MPI possible on calculations is available with the object “SimulateStepRegression”. The \( p_{\text{OneFile}} \) argument is omitted during construction.

This object implements the method “oneStep”

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

where:

- \( p_{\text{statevector}} \) store the states for the all the simulations: this state is updated by application of the optimal command,

- \( p_{\text{phiInOut}} \) 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 the use of this method to simulate an optimal policy with distribution is given below:

```cpp
// Copyright (C) 2016 EDF
// All Rights Reserved
// This code is published under the GNU Lesser General Public License (GNU LGPL)
#include <functional>
#include <memory>
#include <Eigen/Dense>
#include <boost/mpi.hpp>
#include ”generals/VersionBinaryFileArchive.h”
#include ”StOpt/core/grids/FullGrid.h”
#include ”StOpt/core/utils/StateWithStocks.h”
#include ”StOpt/dp/SimulateStepRegressionDist.h”
#include ”StOpt/dp/OptimizerDPBase.h”
#include ”StOpt/dp/SimulatorDPBase.h”

/** ile SimulateStepRegression.h 
 * \brief Defines a simple program showing how to use simulation 
 * A simple grid is used 
 * \author Xavier Warin 
 */

/// \brief Simulate the optimal strategy, mpi version
```

82
double SimulateRegressionDist ( const std::shared_ptr<StOpt::FullGrid> & p_grid ,
    const std::shared_ptr<StOpt::OptimizerDPBase> & p_optimize ,
    const std::function<double(const int &,
                    const Eigen::ArrayXd &, const Eigen::ArrayXd &)> &p_funcFinalValue ,
    const Eigen::ArrayXd & p_pointStock ,
    const int & p_initialRegime ,
    const std::string & p_fileToDump ,
    const bool & p_bOneFile )
{
    boost::mpi::communicator world ;
    // from the optimizer get back the simulator
    std::shared_ptr< StOpt::SimulatorDPBase > simulator = p_optimize ->
        getSimulator () ;
    int nbStep = simulator -> getNbStep () ;
    std::vector< StOpt::StateWithStocks > states ;
    states . reserve ( simulator -> getNbSimul () ) ;
    for ( int is = 0 ; is < simulator -> getNbSimul () ; ++is )
        states . push_back ( StOpt::StateWithStocks ( p_initialRegime ,
                        p_pointStock ,
                        Eigen::ArrayXd::Zero ( simulator -> getDimension () ) ) ) ;
    std::string toDump = p_fileToDump ;
    // test if one file generated
    if ( ! p_bOneFile )
        toDump += "_" + boost::lexical_cast<std::string>(world . rank () ) ;
    gs::BinaryFileArchive ar ( toDump . c_str ( ) , " r " ) ;
    // name for continuation object in archive
    std::string nameAr = " Continuation " ;
    // cost function
    Eigen::ArrayXXd costFunction = Eigen::ArrayXXd::Zero ( p_optimize ->
        getSimuFuncSize () , simulator -> getNbSimul () ) ;
    for ( int istep = 0 ; istep < nbStep ; ++istep )
    {
        StOpt::SimulateStepRegressionDist ( ar , nbStep - 1 - istep , nameAr ,
                        p_grid ,
                        p_optimize , p_bOneFile ) . oneStep ( states , costFunction ) ;
    }
    // new stochastic state
    Eigen::ArrayXXd particles = simulator ->
        stepForwardAndGetParticles () ;
    for ( int is = 0 ; is < simulator -> getNbSimul () ; ++is )
The version of the previous example using a single archive storing the control/solution is given in this file.

- **Simulation step using the optimal controls calculated in optimization**:

```cpp
SimulateStepRegressionControlDist(gs::BinaryFileArchive &p_ar, const int &p_iStep, const std::string &p_nameCont,
const std::shared_ptr<FullGrid> &p_pGridCurrent,
const std::shared_ptr<FullGrid> &p_pGridFollowing,
const std::shared_ptr<OptimizerDPBase> &p_pOptimize,
const bool &p_bOneFile);
```

where

- `p_ar` is the binary archive where the continuation values are stored,
- `p_iStep` is the number associated to the current time step (0 at the beginning date of simulation, the number is increased by one at each time step simulated),
- `p_nameCont` is the base name for 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 following one,
- `p_OneFile` equals to true if a single archive is used to store continuation values.

**Remark 11** A version where a single archive storing the control/solution is used is available with the object "SimulateStepRegressionControl"

This object implements the method "oneStep"
void oneStep(std::vector<StateWithStocks>& p_statevector, Eigen::ArrayXd &p_phiInOut)

where:

- \( p\text{\_statevector} \) stores for all the simulations the state: this state is updated by application of the optimal commands,

- \( p\text{\_phiInOut} \) 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 the use of this method to simulate an optimal policy with distribution is given below:
/// \param p_bOneFile do we store continuation values in only one file
double SimulateRegressionControlDist (const std::shared_ptr<StOpt::FullGrid> &p_grid,
    const std::shared_ptr<StOpt::OptimizerDPBase> &p_optimize,
    const std::function<double(const int, const Eigen::ArrayXd &,
        const Eigen::ArrayXd &)> &p_funcFinalValue,
    const Eigen::ArrayXd &p_pointStock,
    const int &p_initialRegime,
    const std::string &p_fileToDump,
    const bool &p_bOneFile)
{
    boost::mpi::communicator world;
    // from the optimizer get back the simulator
    std::shared_ptr<StOpt::SimulatorDPBase> simulator = p_optimize->
        getSimulator();
    int nbStep = simulator->getNbStep();
    std::vector<StOpt::StateWithStocks> states;
    states.reserve(simulator->getNbSimul());
    for (int is = 0; is < simulator->getNbSimul(); ++is)
        states.push_back(StOpt::StateWithStocks(p_initialRegime,
            p_pointStock, Eigen::ArrayXd::Zero(simulator->getDimension()));
    std::string toDump = p_fileToDump;
    // test if one file generated
    if (!p_bOneFile)
        toDump += "_" + boost::lexical_cast<std::string>(world.rank());
    gs::BinaryFileArchive ar(toDump.c_str(), "r");
    // name for continuation object in archive
    std::string nameAr = "Continuation";
    // cost function
    Eigen::ArrayXXd costFunction = Eigen::ArrayXXd::Zero(p_optimize->
        getSimuFuncSize(), simulator->getNbSimul());
    for (int istep = 0; istep < nbStep; ++istep)
    {
        StOpt::SimulateStepRegressionControlDist(ar, nbStep - 1 - istep,
            nameAr, p_grid, p_grid, p_optimize, p_bOneFile).oneStep(
                states, costFunction);
        // new stochastic state
        Eigen::ArrayXXd particules = simulator->
            stepForwardAndGetParticles();
        for (int is = 0; is < simulator->getNbSimul(); ++is)
            states[is].setStochasticRealization(particules.col(is));
    }
    // final : accept to exercise if not already done entirely (here suppose one function to follow)
    for (int is = 0; is < simulator->getNbSimul(); ++is)
        costFunction(0, is) += p_funcFinalValue(states[is].getRegime(),
            states[is].getPtStock(), states[is].getStochasticRealization());
The version of the previous example using a single archive storing the control/solution is given in [this file].

In the table below we indicate which simulation object should be used at each time step depending on the “TransitionStepRegressionD” object used in optimization.

### 5.3 Solving the problem for $X_{2}^{x,t}$ stochastic

In this part we suppose that $X_{2}^{x,t}$ is controlled but is stochastic.

#### 5.3.1 Requirement to use the framework

In order to use the framework, a business object describing the business on one time step from one state is derived from “OptimizerNoRegressionDPBase.h” itself derived from “OptimizerBase.h”.

```cpp
// Copyright (C) 2016 EDF
// All Rights Reserved
// This code is published under the GNU Lesser General Public License (GNU LGPL)
#endif OPTIMIZERNOREGRESSIONDPBASE_H
#define OPTIMIZERNOREGRESSIONDPBASE_H
#include <Eigen/Dense>
#include "StOpt/core/utils/StateWithStocks.h"
#include "StOpt/core/grids/SpaceGrid.h"
#include "StOpt/regression/BaseRegression.h"
#include "StOpt/regression/GridAndRegressedValue.h"
#include "StOpt/dp/SimulatorDPBase.h"
#include "StOpt/dp/OptimizerBase.h"

/** \file OptimizerNoRegressionDPBase.h
 * \brief Define an abstract class for Dynamic Programming problems solve by Monte Carlo but without regression method to compute conditional expectation.
 * \author Xavier Warin
 */
namespace StOpt {

/// \class OptimizerNoRegressionDPBase OptimizerNoRegressionDPBase.h
/// Base class for optimizer for Dynamic Programming solved without regression method to compute conditional expectation.
class OptimizerNoRegressionDPBase : public OptimizerBase
```
Table 5.2: Which simulation object to use depending on the TransitionStepRegression object used.

<table>
<thead>
<tr>
<th></th>
<th>&quot;TransitionStepRegressionDP&quot;</th>
<th>&quot;TransitionStepRegressionDPDist&quot; &quot;bOneFile&quot; = True</th>
<th>&quot;TransitionStepRegressionDPDist&quot; &quot;bOneFile&quot; = False</th>
<th>&quot;TransitionStepRegressionDPSparse&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;SimulateStepRegression&quot;</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>&quot;SimulateStepRegressionControl&quot;</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>&quot;SimulateStepRegressionDist&quot;</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>&quot;SimulateStepRegressionControlDist&quot;</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>
public:

    OptimizerNoRegressionDPBase() {}

virtual ~OptimizerNoRegressionDPBase() {};

    /// \brief defines the diffusion cone for parallelism
    /// \param p_regionByProcessor region \( \text{min max} \) treated by the
    /// processor for the different regimes treated
    /// \return returns in each dimension the \( \text{min max} \) values in the stock
    /// that can be reached from the grid \( p_{\text{gridByProcessor}} \) for each regime
    virtual std::vector<std::array<double, 2>> getCone(const std::vector<std::array<double, 2>>& p_regionByProcessor) const = 0;

    /// \brief defines the dimension to split for MPI parallelism
    /// For each dimension return true is the direction can be split
    virtual Eigen::Array<bool, Eigen::Dynamic, 1> getDimensionToSplit() const = 0;

    /// \brief defines a step in optimization
    /// \param p_stock coordinates of the stock point to treat
    /// \param p_valNext Optimized values at next time step for each regime
    /// \param p_regressorCur Regressor at the current date
    /// \return a pair:
    ///     - for each regimes (column) gives the solution for each
    ///         particle (row)
    ///     - for each control (column) gives the optimal control
    ///         for each particle (rows)
    virtual std::pair<Eigen::ArrayXd, Eigen::ArrayXXd> stepOptimize(const Eigen::ArrayXd& p_stock, const std::vector<StOpt::GridAndRegressedValue>& p_valNext, std::shared_ptr<BaseRegression> p_regressorCur) const = 0;

    /// \brief Defines a step in simulation using interpolation in controls
    /// \param p_grid grid at arrival step after command
    /// \param p_control defines the controls
    /// \param p_state defines the state value (modified)
    /// \param p_phiInOut defines the value function (modified): size
    ///     number of functions to follow
    virtual void stepSimulateControl(const std::shared_ptr<StOpt::SpaceGrid>& p_grid, const std::vector<StOpt::GridAndRegressedValue>& p_control, const StOpt::StateWithStocks& p_state, Eigen::Ref<Eigen::ArrayXd> p_phiInOut) const = 0;
In addition to the methods of “OptimizerBase.h” the following method is needed:

- the “stepOptimize” methods is used in optimization. We want to calculate the optimal value regressed at current \( t_i \) at a grid point \( p\_{stock} \) using a grid \( p\_{grid} \) at the next date \( t_{i+1} \).

From a grid point \( p\_{stock} \) it calculates the function values regressed and the optimal controls regressed. It returns a pair where the

- first element is a matrix (first dimension is the number of functions in the regression, second dimension the number of regimes) giving the function value regressed,
- second element is a matrix (first dimension is the number of functions in the regression, second dimension the number of controls) giving the optimal control regressed.

In this case of the optimization of an actualized 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 risky asset value, the Optimize object is given in this file

### 5.3.2 The framework in optimization

Once an Optimizer is derived for the project, and supposing that a full grid is used for the stock discretization, the framework provides a “TransitionStepDPDist” object in MPI that permits to solve the optimization problem with distribution of the data on one time step with the following constructor:
with

- \( p\_pGridCurrent \) is the grid at the current time step \( (t_i) \),
- \( p\_pGridPrevious \) is the grid at the previously treated 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 regressor at the previously treated time step \( (t_{i+1}) \) permitting to evaluate a function at date \( t_{i+1} \),
- \( p\_pOptimize \) the optimizer object

**Remark 12** A similar object is available without the MPI distribution framework “TransitionStepDP” with still enabling parallelization with threads and MPI on the calculations on the full grid points.

**Remark 13** The case of sparse grids in currently not treated in the framework.

The main method is

```cpp
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 matrix of optimal values calculated regressed at the previous time iteration for each regime. Each matrix is a number of function regressor at the previous date by number of stock points matrix.

returning a pair:

- 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 regressed functions at the current date by the number of stock points matrix).
- second element is a vector of matrix with new optimal regressed controls at the current time step (each element of the vector corresponds to a control and each matrix is a number of regressed controls by the number of stock points matrix).

**Remark 14** All “TransitionStepDP” derive from a “TransitionStepBase” object having a pure virtual “OneStep” method.
A second method is provided permitting to dump the optimal control at each time step:

```cpp
void dumpValues(std::shared_ptr<gs::BinaryFileArchive> p_ar,
                const std::string &p_name, const int &p_iStep,
                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 15** As for the “TransitionStepDP”, its “dumpValues” doesn’t need a `p_bOneFile` argument: obviously optimal controls are stored in a single file.

We give here a simple example of a time resolution using this method when the MPI distribution of data is used.
```cpp
const shared_ptr<OptimizePortfolioDP>& p_optimize,
const ArrayXi &p_nbMesh,
const function<double(const int &,
    const ArrayXd &, const ArrayXd &)> &p_funcFinalValue,
const ArrayXd &p_initialPortfolio,
const string &p_fileToDump,
const bool &p_bOneFile
)
{
    // initialize simulation
    p_optimize->initializeSimulation();
    // store regressor
    shared_ptr<StOpt::LocalConstRegression> regressorPrevious;

    // store final regressed values in object valuesStored
    shared_ptr<vector<ArrayXXd>> valuesStored = make_shared<vector<ArrayXXd>>(p_optimize->getNbRegime());
    {
        vector<shared_ptr<ArrayXXd>> valuesPrevious = StOpt::FinalStepRegressionDPDist(p_grid, p_optimize->getNbRegime(), p_optimize->getDimensionToSplit())(p_funcFinalValue, *p_optimize->getCurrentSim());
        // regressor operator
        regressorPrevious = make_shared<StOpt::LocalConstRegression>(false, p_optimize->getCurrentSim(), p_nbMesh);
        for (int iReg = 0; iReg < p_optimize->getNbRegime(); ++iReg)
            (*valuesStored)[iReg] = regressorPrevious->getCoordBasisFunctionMultiple(valuesPrevious[iReg]->transpose()).transpose();
    }

    boost::mpi::communicator world;
    string toDump = p_fileToDump;
    // test if one file generated
    if (!p_bOneFile)
        toDump += "," + boost::lexical_cast<string>(world.rank());
    shared_ptr<gs::BinaryFileArchive> ar;
    if (!p_bOneFile || (world.rank() == 0))
        ar = make_shared<gs::BinaryFileArchive>(toDump.c_str(), "w");
    // name for object in archive
    string nameAr = "OptimizePort";
    // iterate on time steps
    for (int iStep = 0; iStep < p_optimize->getNbStep(); ++iStep)
    {
        // step backward for simulations
        p_optimize->oneStepBackward();
        // create regressor at the given date
        bool bZeroDate = (iStep == p_optimize->getNbStep() - 1);
        shared_ptr<StOpt::LocalConstRegression> regressorCur = make_shared<StOpt::LocalConstRegression>(bZeroDate, p_optimize->getCurrentSim(), p_nbMesh);
        // transition object
        StOpt::TransitionStepDPDist transStep(p_grid, p_grid, regressorCur,
```
regressorPrevious, p_optimize);
    pair< shared_ptr< vector< ArrayXXd > >, shared_ptr< vector< ArrayXXd > > > valuesAndControl = transStep.oneStep(*valuesStored);
    // dump control values
    transStep.dumpValues(ar, nameAr, iStep, *valuesAndControl.second, p_bOneFile);
    valuesStored = valuesAndControl.first;
    // shift regressor
    regressorPrevious = regressorCur;
}
// interpolate at the initial stock point and initial regime (0 here) (take first particle)
shared_ptr< ArrayXXd > topRows = make_shared< ArrayXXd >((*valuesStored)[0].topRows(1));
    return StOpt::reconstructProc0Mpi( p_initialPortfolio, p_grid, topRows, p_optimize->getDimensionToSplit() );
} #endif

An example without distribution of the data can be found in this file.

5.3.3 The framework in simulation

Not special framework is available in simulation. Use the function “SimulateStepRegressionControl” or “SimulateStepRegressionControlDist” described in the section 5.2.3.
Chapter 6
The Python API

In order to use the Python API, it is possible to use only the mapping of the grids, continuation values, and regression object and to 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

```python
# Copyright (C) 2016 EDF
# All Rights Reserved
# This code is published under the GNU Lesser General Public License (GNU LGPL)
import numpy as np
import StOptReg as reg
import StOptGrids
import StOptGlobal

class TransitionStepRegressionDP:
    def __init__(self, p_pGridCurrent, p_pGridPrevious, p_pOptimize):
        self.m_pGridCurrent = p_pGridCurrent
        self.m_pGridPrevious = p_pGridPrevious
        self.m_pOptimize = p_pOptimize

    def oneStep(self, p_phiIn, p_condExp):
        nbRegimes = self.m_pOptimize.getNbRegime()
        phiOut = list(range(nbRegimes))
        nbControl = self.m_pOptimize.getNbControl()
        controlOut = list(range(nbControl))

        # only if the processor is working
        if self.m_pGridCurrent.getNbPoints() > 0:
            # allocate for solution
            for iReg in range(nbRegimes):
                phiOut[iReg] = np.zeros((p_condExp.getNbSimul(), self.m_pGridCurrent.getNbPoints()))

            for iCont in range(nbControl):
```
controlOut[iCont] = np.zeros((p_condExp.getNbSimul(), self.m_pGridCurrent.getNbPoints()))

# number of threads
nbThreads = 1

contVal = []

for iReg in range(len(p_phiIn)):
    contVal.append(reg.ContinuationValue(self.m_pGridPrevious, p_condExp, p_phiIn[iReg]))

# create iterator on current grid treated for processor
iterGridPoint = self.m_pGridCurrent.getGridIteratorInc(0)

# iterates on points of the grid
for iIter in range(self.m_pGridCurrent.getNbPoints()):
    if iterGridPoint.isValid():
        pointCoord = iterGridPoint.getCoordinate()
        # optimize the current point and the set of regimes
        solutionAndControl = self.m_pOptimize.stepOptimize(self.m_pGridPrevious, pointCoord, contVal, p_phiIn)

        # copy solution
        for iReg in range(self.m_pOptimize.getNbRegime()):
            phiOut[iReg][:, iterGridPoint.getCount()] = solutionAndControl[0][:, iReg]

        for iCont in range(nbControl):
            controlOut[iCont][:, iterGridPoint.getCount()] = solutionAndControl[1][:, iCont]

        iterGridPoint.nextInc(nbThreads)

res = []
res.append(phiOut)
res.append(controlOut)
return res

This object can be used as in a time step optimization as follows

# Copyright (C) 2016 EDF
# All Rights Reserved
# This code is published under the GNU Lesser General Public License (GNU LGPL)
import TransitionStepRegressionDP as trans
import FinalStepRegressionDP as final

def DynamicProgrammingByRegression(p_grid, p_optimize, p_regressor,
    p_funcFinalValue, p_pointStock, p_initialRegime, p_fileToDump):
    # from the optimizer get back the simulation
    simulator = p_optimize.getSimulator()
# final values
valuesNext = final.FinalStepRegressionDP(p_grid, p_optimize.getNbRegime()).operator(p_funcFinalValue, simulator.getParticules())

# iterate on time steps
for iStep in range(simulator.getNbStep()) :
    asset = simulator.stepBackwardAndGetParticules()
    if iStep == (simulator.getNbStep() - 1):
        p_regressor.updateSimulations(True, asset)
    else:
        p_regressor.updateSimulations(False, asset)

# transition object
transStep = trans.TransitionStepRegressionDP(p_grid, p_grid, p_optimize)
valuesAndControl = transStep.oneStep(valuesNext, p_regressor)
valuesNext = valuesAndControl[0]

# interpolate at the initial stock point and initial regime
return (p_grid.createInterpolator(p_pointStock).applyVec(valuesNext[p_initialRegime]).mean())

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 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 BoostPythonSimulators.cpp file. Similarly the 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 BoostPythonOptimizers.cpp file.

In the example below we describe the use of this high level interface for the swing options with a Black Scholes simulator : we give in this example the mapping of the mostly used objects:
class OptimizerConstruction(unittest.TestCase):

def test(self):
    try:
        import mpi4py
        found = True
    except:
        print("Not parallel module found ")
        found = False

    if found:
        from mpi4py import MPI
        comm = MPI.COMM_WORLD
        initialValues = np.zeros(1,dtype=np.float) + 1.
        sigma = np.zeros(1.) + 0.2
        mu = np.zeros(1.) + 0.05
        corr = np.ones((1.,1.))
        # number of step
        nStep = 30
        # exercise dates
        dates = np.linspace(0.,1.,nStep + 1)
        T= dates[len(dates) - 1]
        nbSimul = 10 # simulation number (optimization and simulation)
        # simulator
        bsSim = sim.BlackScholesSimulator(initialValues,sigma,mu,corr,
        T, nStep - 1, nbSimul, False)
        strike = 1.
        # Pay off
        payOff= Utils.BasketCall(strike)
        # optimizer
        swiOpt = opt.OptimizerSwingBlackScholes(payOff,N)
        # link simulator to optimizer
        swiOpt.setSimulator(bsSim)
        # archive
        ar = StOptGlobal.BinaryFileArchive("Archive","w")
        # regressor
        nMesh = 1
        regressor = StOptReg.LocalLinearRegression(nMesh)
        # Grid
        # low value for the meshes
        lowValues =np.array([0.], dtype=np.float)
        # size of the meshes
        step = np.array([1.], dtype=np.float)
        # number of steps
        nbStep = np.array([N-1], dtype=np.int32)
        grid = StOptGrids.RegularSpaceGrid(lowValues, step, nbStep)
# pay off function for swing

```python
payOffBasket = Utils.BasketCall(strike);
payoff = Utils.PayOffSwing(payOffBasket,N)
dir (payoff)

print("payoff", payoff.set(0.,np.array([0.5],dtype=np.float),
        np.array([1.],dtype=np.float)))
```

# final step

```python
asset = bsSim.getParticles()
fin = StOptGlobal.FinalStepRegressionDP(grid,1)
values = fin.set ( payoff, asset )
```

# transition time step

```python
# on step backward and get asset
asset = bsSim.stepBackwardAndGetParticles()
# update regressor
regressor.updateSimulations(0, asset)
transStep = StOptGlobal.TransitionStepRegressionDP(grid,grid,
          swiOpt)
valuesNextAndControl = transStep.oneStep(values, regressor)
transStep.dumpContinuationValues (ar,"Continuation",1,
         valuesNextAndControl[0],valuesNextAndControl[1], regressor)
```

# simulate time step

```python
nbSimul=10
vecOfStates=[] # state of each simulation
for i in np.arange(nbSimul):
    # one regime, all with same stock level (dimension 2), same
    # realization of simulation (dimension 3)
vecOfStates.append(StOptGlobal.StateWithStocks(1,
                                              np.array([1.])))

arRead = StOptGlobal.BinaryFileArchive("Archive","r")
phi = np.zeros((1,nbSimul))
NewState = VecOfStateNext = simStep.oneStep(vecOfStates, phi)
```

# print new state (different of C++)

```python
print("New vector of state", NewState[0])
for i in np.arange(len(NewState[0])):
    print(" i",i,"Stock",NewState[0][i].getPtStock(),"Regime",
          NewState[0][i].getRegime(),"Stochastic reali",
          NewState[0][i].getStochasticRealization())
print("New cost function", NewState[1])
```

if __name__ == '__main__':
    unittest.main()
import StOptGrids
import StOptReg
import StOptGlobal

def DynamicProgrammingByRegressionHighLevel(p_grid, p_optimize, p_regressor, p_funcFinalValue, p_pointStock, p_initialRegime, p_fileToDump):
    # from the optimizer get back the simulation
    simulator = p_optimize.getSimulator()
    # final values
    fin = StOptGlobal.FinalStepRegressionDP(p_grid, p_optimize.getNbRegime())
    valuesNext = fin.set(p_funcFinalValue, simulator.getParticles())
    ar = StOptGlobal.BinaryFileArchive(p_fileToDump, "w")
    nameAr = "Continuation"
    # iterate on time steps
    for iStep in range(simulator.getNbStep()):
        asset = simulator.stepBackwardAndGetParticles()
        # conditional expectation operator
        if iStep == (simulator.getNbStep() - 1):
            p_regressor.updateSimulations(True, asset)
        else:
            p_regressor.updateSimulations(False, asset)

        # transition object
        transStep = StOptGlobal.TransitionStepRegressionDP(p_grid, p_grid, p_optimize)
        valuesAndControl = transStep.oneStep(valuesNext, p_regressor)
        transStep.dumpContinuationValues(ar, nameAr, iStep, valuesNext, valuesAndControl[1], p_regressor)
        valuesNext = valuesAndControl[0]
    
    # interpolate at the initial stock point and initial regime
    return (p_grid.createInterpolator(p_pointStock).applyVec(valuesNext[p_initialRegime]).mean())

Similarly a python time nest in simulation can be given as an example by:

# Copyright (C) 2016 EDF
# All Rights Reserved
# This code is published under the GNU Lesser General Public License (GNU LGPL)
import numpy as np
import StOptGlobal
import math

# Basket payoff (until get full mapping)
def almostEqual(x, y, ulp):
    return abs(x - y) <= 2.220446049250313e-16 * abs(x + y) * ulp #
    2.220446049250313e-16 = std::numeric_limits::epsilon() in C++
# Simulate the optimal strategy, threaded version

# p_grid
grid used for deterministic state (stocks for example)

# p_optimize
optimizer defining the optimization between two time steps

# p_payOff
function defining the final value (pay off)

# p_pointStock
initial point stock

# p_initialRegime
regime at initial date

# p_fileToDump
name of the file used to dump continuation values in optimization

def SimulateRegressionHighLevel(p_grid, p_optimize, p_payOff, p_pointStock, p_initialRegime, p_fileToDump):
    # from the optimizer get back the simulation
    simulator = p_optimize.getSimulator()
    nbStep = simulator.getNbStep()
    states = []

    for i in range(simulator.getNbSimul()):
        states.append(StOptGlobal.StateWithStocks(p_initialRegime, p_pointStock, np.zeros(simulator.getDimension())))

    # name for continuation object in archive
    nameAr = "Continuation"

    # cost function
    costFunction = np.zeros(simulator.getNbSimul())

    # iterate on time steps
    arRead = StOptGlobal.BinaryFileArchive(p_fileToDump, "r")

    for istep in range(nbStep):

        # different from C++
        states = NewState[0]
        costFunction = NewState[1]

        # new stochastic state
        particules = simulator.stepForwardAndGetParticles()

        for i in range(simulator.getNbSimul()):
            states[i].setStochasticRealization(particules[:, i])

    # update actualization
    p_optimize.incrementActuTimeStep()

    # final : accept to exercise if not already done entirely
    for i in range(simulator.getNbSimul()):
        costFunction[i] += p_payOff.set(states[i].getRegime(), states[i].getPtStock(), states[i].getStochasticRealization()) * p_optimize.getActuStep()

    # average gain/cost
    return costFunction.mean()
Equivalent using MPI and the distribution of 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 valorization.
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 7

Theoretical background

In this part, we are back to the resolution of equation (1).

7.1 Notation and regularity results

We denote by ∧ the minimum and ∨ the maximum. We denote by | | the Euclidean norm of a vector, \( Q := (0, T] \times \mathbb{R}^d \). For a bounded function \( w \), we set

\[
|w|_0 = \sup_{(t,x) \in Q} |w(t, x)|, \quad [w]_1 = \sup_{(s,x) \neq (t,y)} \frac{|w(s, x) - w(t, y)|}{|x - y| + |t - s|^{\frac{1}{2}}}
\]

and \( |w|_1 = |w|_0 + [w]_1 \). \( C_1(Q) \) will stand for the space of functions with a finite \( | |_1 \) norm.

For \( t \) given, we denote

\[
||w(t, .)||_{\infty} = \sup_{x \in \mathbb{R}^d} |w(t, x)|
\]

We use the classical assumption on the data of (1) for a given \( \hat{K} \):

\[
\sup_a |g|_1 + |\sigma_a|_1 + |b_a|_1 + |f_a|_1 + |c_a|_1 \leq \hat{K} \tag{7.1}
\]

A classical result \([2]\) gives us the existence and uniqueness of the solution in the space of bounded Lipschitz functions:

**Proposition 1** If the coefficients of the equation (1) satisfy (7.1), there exists a unique viscosity solution of the equation (1) belonging to \( C_1(Q) \). If \( u_1 \) and \( u_2 \) are respectively sub and super solution of equation (1) satisfying \( u_1(0, .) \leq u_2(0, .) \) then \( u_1 \leq u_2 \).

A spatial discretization length of the problem \( \Delta x \) being given, thereafter \( (i_1 \Delta x, ..., i_d \Delta x) \) with \( \vec{i} = (i_1, ..., i_d) \in \mathbb{Z}^d \) will correspond to the coordinates of a mesh \( M_{\vec{i}} \) defining a hyper-cube in dimension \( d \). For an interpolation grid \( (\xi_i)_{i=0...N} \in [-1, 1]^N \), and for a mesh \( \vec{i} \), the point \( y_{\vec{i}, \vec{j}} \) with \( \vec{j} = (j_1, ..., j_d) \in [0, N]^d \) will have the coordinate \( \Delta x(i_1 + 0.5(1 + \xi_{j_1})), ..., \Delta x(i_d + 0.5(1 + \xi_{j_d})) \). We denote \( (y_{\vec{i}, \vec{j}})_{\vec{i}, \vec{j}} \) the set of all the grids points on the whole domain.

We notice that for regular mesh with constant volume \( \Delta x^d \), we have the following relation for all \( x \in \mathbb{R}^d \):

\[
\min_{\vec{i}, \vec{j}} |x - y_{\vec{i}, \vec{j}}| \leq \Delta x. \tag{7.2}
\]
7.2 Time discretization for HJB equation

The equation (1) is discretized in time by the scheme proposed by Camilli Falcone [22] for a time discretization $h$.

$$v_h(t + h, x) = \inf_{a \in A} \left[ \sum_{i=1}^{q} \frac{1}{2q} (v_h(t, \phi_{a,h,i}^+(t, x)) + v_h(t, \phi_{a,h,i}^-(t, x))) + f_a(t, x)h + c_a(t, x)hv_h(t, x) \right]$$

$$:= v_h(t, x) + \inf_{a \in A} L_{a,h}(v_h)(t, x)$$ (7.3)

with

$$L_{a,h}(v_h)(t, x) = \sum_{i=1}^{q} \frac{1}{2q} (v_h(t, \phi_{a,h,i}^+(t, x)) + v_h(t, \phi_{a,h,i}^-(t, x)) - 2v_h(t, x))$$

$$+hc_a(t, x)v_h(t, x) + hf_a(t, x)$$

$$\phi_{a,h,i}^+(t, x) = x + b_a(t, x)h + (\sigma_a)i(t, x)\sqrt{hq}$$

$$\phi_{a,h,i}^-(t, x) = x + b_a(t, x)h - (\sigma_a)i(t, x)\sqrt{hq}$$

where $(\sigma_a)i$ is the $i$-th column of $\sigma_a$. We note that it is also possible to choose other types of discretization in the same style as those defined in [23].

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 time linear interpolation once the solution has been calculated at date $h$:

$$v_h(t, x) = (1 - \frac{t}{h})g(x) + \frac{t}{h}v_h(h, x), \forall t \in [0, h].$$ (7.4)

We first recall the following result:

**Proposition 2** Under the condition on the coefficients given by equation (7.1), the solution $v_h$ of equations (7.3) and (7.4) is uniquely defined and belongs to $C_1(Q)$. We check that if $h \leq (16 \sup_a \{|\sigma_a|^2 + |b_a|^2 + 1\} \wedge 2 \sup_a |c_a|^0)^{-1}$, there exists $C$ such that

$$|v - v_h|_0 \leq Ch^{\frac{1}{4}}.$$ (7.5)

Moreover, there exists $C$ independent of $h$ such that

$$|v_h|_0 \leq C,$$ (7.6)

$$|v_h(t, x) - v_h(t, y)| \leq C|x - y|, \forall (x, y) \in Q^2.$$ (7.7)

7.3 Space interpolation

The space resolution of equation (7.3) is achieved on a grid. The $\phi^+$ and $\phi^-$ have to be computed by the use of an interpolator $I$ such that:

$$v_h(t, \phi_{a,h,i}^+(t, x)) \simeq I(v_h(t, .))(\phi_{a,h,i}^+(t, x)),$$

$$v_h(t, \phi_{a,h,i}^-(t, x)) \simeq I(v_h(t, .))(\phi_{a,h,i}^-(t, x)).$$
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 [24], [25]).
Chapter 8

C++ API

In order to achieve 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:

```cpp
namespace StOpt
{

/// brief Semi Lagrangian method for process \(dx_t = b \, dt + \sigma \, dW_t\) 
/// where \(X_t, h \) \(\sigma\) with values in \(\mathbb{R}, n \, \mathbb{R}\) , \(\mathbb{R}\) \(\mathbb{R}\) 
/// \(W_t \) \(W_t\) matrix and \(W_t\) \(W_t\) with values in \(\mathbb{R} \mathbb{R}\) 

class SemiLagrangEspCond
{
    //\brief interpolator
    std::shared_ptr<InterpolatorSpectral> m_interpolator;
}
```
Its constructor uses the following arguments:

- a first one “p_interpolator” defines a “spectral” interpolator on a grid : this “spectral” interpolator is constructed from a grid and a function to interpolate (see section 1). In our case, it will be used to interpolate the solution from the previous time step,

- a second one “p_extremalValues” defines for each dimension the minimal and maximal coordinates of points belonging to the grid,

- a third one “p_bModifVol” if set to “true” permits to achieve a special treatment when points to interpolate are outside the grid : the volatility of the underlying process is modified (keeping the same mean and variance) trying to keep points inside the domain (see [24]).

This object has the method “oneStep” taking

- $p_x$ the foot of the characterize (for each dimension),
• $p_b$ the trend of the process (for each dimension),

• $p_{\text{sig}}$ the matrix volatility of the process,

such that the interpolation is achieved for a time step $h$ at points $p_x + p_b h \pm p_{\text{sig}} \sqrt{h}$. It returns a pair $(a, b)$ where $a$ contains the calculated value if the $b$ value is true. When the interpolation is impossible to achieve, the $b$ value is set to false.

In order to use the API, an object deriving from the “OptimizerSLBase.h” object has to be constructed. This object permits to define the PDE to solve (with it optimization problem if any).

```cpp
namespace StOpt
{
    class OptimizerSLBase
    {
        public:
            OptimizerSLBase() {}
            virtual ~OptimizerSLBase() {};

            virtual std::vector<std::array<double, 2>> getCone(const std::vector<std::array<double, 2>>& p_regionByProcessor) const = 0;
    }
}
```
virtual Eigen::Array< bool, Eigen::Dynamic, 1> getDimensionToSplit() const = 0 ;

virtual std::pair< Eigen::ArrayXd, Eigen::ArrayXd> stepOptimize(const Eigen::ArrayXd &p_point, const std::vector< std::shared_ptr<SemiLagrangEspCond> > & p_semiLag, const double &p_time, const Eigen::ArrayXd &p_phiInPt) const = 0 ;

virtual void stepSimulate(const SpaceGrid &p_gridNext, const std::vector< std::shared_ptr<StOpt::SemiLagrangEspCond> > & p_semiLag, const Eigen::Ref< Eigen::ArrayXd > p_state, int & p_iReg, const Eigen::ArrayXd &p_gaussian, const Eigen::ArrayXd &p_phiInPt, Eigen::Ref< Eigen::ArrayXd > p_phiInOut) const = 0 ;

virtual void stepSimulateUsingControlCalculatedInOptimization(const SpaceGrid &p_gridNext, const std::vector< std::shared_ptr<StOpt::SemiLagrangEspCond> > & p_semiLag, const Eigen::Ref< Eigen::ArrayXd > p_state, int & p_iReg, const Eigen::ArrayXd &p_gaussian) const = 0 ;
The main methods associated to this object are:

- “stepOptimize” is used to calculate the solution of the PDE at one point.
  - It takes a point of the grid used \( p \_\text{point} \),
  - and apply the semi Lagrangian scheme \( p \_\text{semiLag} \) at this point,
  - at a date given by \( p \_\text{time} \).

  It returns a pair containing:
  - the function value calculated at \( p \_\text{point} \) for each regime,
  - the optimal control calculated at \( p \_\text{point} \) for each control.

- “stepSimulate” is used when the PDE is associated to an optimization problem and we want to simulate an optimal policy using the function values calculated in the optimization part. The arguments are:
- \textit{p.gridNext} defining the grid used at the following time step,
- \textit{p.semiLag} the semi Lagrangian operator constructed with an interpolator using the following time solution,
- \textit{p.state} the vector defining the current state for the current regime,
- \textit{p.iReg} the current regime number,
- \textit{p.gaussian} is the vector of gaussian random variables used to calculate the Brownian involved in the underlying process for the current simulation,
- \textit{p.phiInP} at the value of the function calculated in optimization at next time step for the given point,
- \textit{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 to an optimization problem and we want to simulate an optimal policy using the optimal controls calculated in the optimization part. The arguments are:
  - \textit{p.gridNext} defining the grid used at the following time step,
  - \textit{p.controlInterp} a vector (for each control) of interpolators in controls
  - \textit{p.state} the vector defining the current state for the current regime,
  - \textit{p.iReg} the current regime number,
  - \textit{p.gaussian} is the vector of gaussian random variables used to calculate the Brownian involved in the underlying process for the current simulation.
  - \textit{p.phiInOut} storing the cost functions: the size of the array is the number of functions to follow in simulation.

On return the \textit{p.state} vector is modified, the \textit{p.iReg} is modified and the cost function \textit{p.phiInOut} is modified for the current trajectory.

- the “getCone” method is only relevant if the distribution for data (so MPI) is used. As argument it take a vector of size the dimension of the grid. Each component of the vector is an array containing the minimal and maximal coordinates values of points of the current grid defining an hyper cube \( H_1 \). It returns for each dimension, the coordinates min and max of the hyper cube \( H_2 \) containing the points that can be reached by applying a command from a grid point in \( H_1 \). If no optimization is achieved, it returns the hyper cube \( H_2 \) containing the points reached by the semi Lagrangian scheme. For explanation of the parallel formalism see chapter 5

- the “getDimensionToSplit” method is only relevant if the distribution for data (so MPI) is used. The method permits to define which directions to split for solution distribution on processors. For each dimension it returns a Boolean where “true” means that the direction is a candidate for splitting,

- the “isNotNeedingBC” permits to define for a point on the boundary of the grid if a boundary condition is needed (“True” is returned) or if no boundary is needed (return “false”).

113
And example of the derivation of such an optimizer for a simple stochastic target problem (described in paragraph 5.3.4 in [24]) is given below:

```cpp
#include <iostream>
#include "StOpt/core/utils/constant.h"
#include "test/c++/tools/semilagrangeOptimizeSLCase3.h"

using namespace StOpt;
using namespace Eigen;
using namespace std;

OptimizerSLCase3::OptimizerSLCase3(const double &pMu, const double &pSig, const double &pdt, const double &pAlphaMax, const double &pStepAlpha):
    mDtl(pDtl), mMu(pMu), mSig(pSig), mAlphaMax(pAlphaMax), mStepAlpha(pStepAlpha) {}

vector< array< double, 2 > > OptimizerSLCase3::get Cone(const vector< array< double, 2 > > &pInit) const
{
    vector< array< double, 2 > > xReached(1);
    xReached[0][0] = pInit[0][0] - mAlphaMax * mMu / mSig * mDtl  
                    - mAlphaMax * sqrt(mDtl);
    xReached[0][1] = pInit[0][1] + mAlphaMax * sqrt(mDtl);
    return xReached;
}

pair< ArrayXd, ArrayXd > OptimizerSLCase3::stepOptimize(const ArrayXd & pPoint,  
const vector< shared_ptr<SemiLagrangianOptimizeSLCase3> > & pSemiLag, const double & mDtl, const Eigen::VectorXd &) const
{
    pair< ArrayXd, ArrayXd > solutionAndControl;
    solutionAndControl.first.resize(1);
    solutionAndControl.second.resize(1);
    ArrayXd b(1);
    ArrayXXd sig(1, 1);
    double vMin = StOpt::inf;
    for ( int iAl = 0; iAl < mAlphaMax / mStepAlpha; ++iAl)
    {
        double alpha = iAl * mStepAlpha;
        b(0) = -alpha * mMu / mSig; // trend
        sig(0) = alpha; // volatility with one Brownian
        pair< double, bool > lagrang = pSemiLag[0]->oneStep(pPoint, b, sig, mDtl); // test the control
        if (lagrang.second)
            if (lagrang.first < vMin)
                vMin = lagrang.first;
        solutionAndControl.second(0) = alpha;
    }
    solutionAndControl.first(0) = vMin;
```

114
return solutionAndControl;
}

void OptimizerSLCase3::stepSimulate(const StOpt::SpaceGrid &p_gridNext,  
const std::vector< std::shared_ptr< StOpt::
    SemiLagrangEspCond > > &p_semiLag, 
Eigen::Ref<Eigen::ArrayXd> &p_state, int &,
    const Eigen::ArrayXd &p_gaussian, const 
Eigen::ArrayXd &, 
Eigen::Ref<Eigen::ArrayXd>) const
{
    double vMin = StOpt::infty;
    double alphaOpt = -1;
    ArrayXd b(1);
    ArrayXXd sig(1, 1);
    ArrayXd proba = p_state;
    // recalculate the optimal alpha
    for (int iAl = 0; iAl < m_alphaMax / m_stepAlpha; ++iAl)
    {
        double alpha = iAl * m_stepAlpha;
        b(0) = -alpha * m_mu / m_sig; // trend
        sig(0) = alpha; // volatility with one Brownian
        pair<double, bool> lagrang = p_semiLag[0]->oneStep(proba, b, sig, 
            m_dt); // test the control
        if (lagrang.second)
        {
            if (lagrang.first < vMin)
            {
                vMin = lagrang.first;
                alphaOpt = alpha;
            }
        }
        proba(0) += alphaOpt * p_gaussian(0) * sqrt(m_dt);
        // truncate if necessary
        p_gridNext.truncatePoint(proba);
        p_state = proba;
    }
}

void OptimizerSLCase3::stepSimulateControl(const SpaceGrid &p_gridNext,  
const std::vector< std::shared_ptr< InterpolatorSpectral > > &p_controlInterp,  
Eigen::Ref<Eigen::ArrayXd> &p_state, int &, 
const ArrayXd &p_gaussian, 
Eigen::Ref<Eigen::ArrayXd>) const
{
    ArrayXd proba = p_state;
    double alphaOpt = p_controlInterp[0]->apply(p_state);
    proba(0) += alphaOpt * p_gaussian(0) * sqrt(m_dt);
    // truncate if necessary
    p_gridNext.truncatePoint(proba);
8.1 PDE resolution

Once the problem is described, a time recursion can be achieved using the “Transition-StepSemilagrang” object in a sequential resolution of the problem. This object permits to solve the problem on one time step.
It constructor takes the following arguments:

- **p_gridCurrent** a grid describing the meshes at the current date,
- **p_gridPrevious** a grid describing the meshes at the previously treated date,
- **p_optimize** an object derived from the “OptimizerSLBase” and describing the problem to solve at a given date and a given point of the current grid.

A first method “oneStep” take 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 depending on the number of regimes and the position on the boundary.
Table 8.1: Which “TransitionStepSemilagrang” object to use depending on the grid used and the type of parallelization used.

<table>
<thead>
<tr>
<th></th>
<th>Full grid</th>
<th>Sparse grid</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequential</td>
<td>“TransitionStepSemilagrang”</td>
<td>“TransitionStepSemilagrang”</td>
</tr>
<tr>
<td>Parallelization on calculations threads and MPI</td>
<td>“TransitionStepSemilagrang”</td>
<td>“TransitionStepSemilagrang”</td>
</tr>
<tr>
<td>Distribution of calculations and data (MPI)</td>
<td>“TransitionStepSemilagrangDist”</td>
<td>Not available</td>
</tr>
</tbody>
</table>

It gives back an estimation of the solution at the current date on the current grid for all the regimes and an estimation of the optimal control calculated for all the controls.

A last method “dumpValues” method permits to dump the solution calculated $p_{phiIn}$ at the step $p_{istep} + 1$ and the optimal control at step $p_{istep}$ in an archive $p_{ar}$.

A version using the distribution of the data and calculations can be found in the TransitionStepSemilagrangDist object. An example of a time recursion in sequential can be found in the semiLagrangianTime function and an example with distribution can be found in the semiLagrangianTimeDist function. In both functions developed in the test chapter the analytic solution of the problem is known and compared to the numerical estimation obtained with the semi Lagrangian method.

8.2 Simulation framework

Once the optimal controls and the value functions are calculated, one can simulate the optimal policy by using the function values (recalculating the optimal control for each simulation) or using directly the optimal controls calculated in optimization:

- Calculate the optimal strategy in simulation
  by using the function values calculated in optimization:

In order to simulate one step of the optimal policy, an object “SimulateStepSemilagrangDist” is provided with constructor

```cpp
SimulateStepSemilagrangDist(gs::BinaryFileArchive &p_ar, const int &p_iStep, const std::string &p_name,
const std::shared_ptr<FullGrid> &p_gridNext, const std::shared_ptr<
OptimizerSLBase> &p_pOptimize,
const bool &p_bOneFile);
```

where

- $p_{ar}$ is the binary archive where the continuation values are stored,
- $p_{iStep}$ is the number associated to the current time step (0 at the beginning date of simulation, the number is increased by one at each time step simulated),
- $p_{name}$ is the base name to search in the archive,
- $p_{GridNext}$ is the grid at the next time step ($p_{iStep} + 1$),
- $p_{\text{Optimize}}$ is the Optimizer describing the transition from one time step to the following one,
- $p_{\text{OneFile}}$ equals to true if a single archive is used to store continuation values.

**Remark 16** A version without distribution of data but only multithreaded and parallelized with MPI on data is available with the object “SimulateStepSemilagrang”

This object implements the method “oneStep”

```cpp
void oneStep(const Eigen::ArrayXXd & p_gaussian , Eigen::ArrayXXd & p_statevector , Eigen::ArrayXi & p_iReg , Eigen::ArrayXd & p_phiInOuts )
```

where:

- $p_{\text{gaussian}}$ is a two dimensional array (number of Brownian in the modelization by the number of Monte Carlo simulations).
- $p_{\text{statevector}}$ store the continuous state (continuous state size by number of simulations)
- $p_{\text{iReg}}$ for each simulation give the current regime number,
- $p_{\text{phiInOut}}$ 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.

**Remark 17** The previous object “SimulateStepSemilagrangDist” is used with MPI for problems of quite high dimension. In the case of small dimension (below or equal to three), the parallelization with MPI or the sequential calculations can be achieved by the “SimulateStepSemilagrang” object.

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

```cpp
// Copyright (C) 2016 EDF
// All Rights Reserved
// This code is published under the GNU Lesser General Public License (GNU LGPL)
#ifdef USE_MPI
#include <boost/random.hpp>
#include <memory>
#include <Eigen/Dense>
#include "generators/BinaryFileArchive.hh"
#include "StOpt/semilagrangien/OptimizerSLBase.h"
#include "StOpt/semilagrangien/SimulateStepSemilagrangDist.h"

using namespace std;

double semiLagrangianSimuDist(const shared_ptr<StOpt::FullGrid> & p_grid,
```
const shared_ptr<StOpt::OptimizerSLBase> &p_optimize,
const function<double(const int &, const Eigen::ArrayXd &)> &p_funcFinalValue,
const int &p_nbStep,
const Eigen::ArrayXd &p_stateInit,
const int &p_initialRegime,
const int &p_nbSimul,
const string &p_fileToDump,
const bool &p_bOneFile)
{
  boost::mpi::communicator world;
  // store states in a regime
  Eigen::ArrayXXd states(p_stateInit.size(), p_nbSimul);
  for (int is = 0; is < p_nbSimul; ++is)
    states.col(is) = p_stateInit;
  // store the regime number
  Eigen::ArrayXi regime = Eigen::ArrayXi::Constant(p_nbSimul, p_initialRegime);
  // test if one file generated
  string toDump = p_fileToDump;
  if (!p_bOneFile)
    toDump += "_" + boost::lexical_cast<string>(world.rank());
  gs::BinaryFileArchive ar(toDump.c_str(), "r");
  // name for continuation object in archive
  string nameAr = "Continuation";
  // cost function
  Eigen::ArrayXXd costFunction = Eigen::ArrayXXd::Zero(p_optimize->
    getSimuFuncSize(), p_nbSimul);
  // random generator and Gaussian variables
  boost::mt19937 generator;
  boost::normal_distribution<double> normalDistrib;
  boost::variate_generator<boost::mt19937 &, boost::normal_distribution
  <double>> normalRand(generator, normalDistrib);
  Eigen::ArrayXd gaussian(p_optimize->getBrownianNumber(), p_nbSimul);
  // iterate on time steps
  for (int istep = 0; istep < p_nbStep; ++istep)
  {
    for (int id = 0; id < gaussian.cols(); ++id)
      gaussian(id, is) = normalRand();
    StOpt::SimulateStepSemilagrangDist(ar, p_nbStep - 1 - istep,
      nameAr, p_grid, p_optimize, p_bOneFile).oneStep(gaussian,
      states, regime, costFunction);
  }
  // final cost to add
  for (int is = 0; is < p_nbSimul; ++is)
    costFunction(0, is) += p_funcFinalValue(regime(is), states.col(is));
  // average gain/cost
  return costFunction.mean();
} #endif
A sequential or parallelized on calculations version of the previous example is given in this file.

- Calculate the optimal strategy in simulation by interpolation of the optimal control calculated in optimization:

In order to simulate one step of the optimal policy, an object “SimulateStepSemilagrangControlDist” is provided with constructor

where

- \( p_ar \) is the binary archive where the continuation values are stored,
- \( p_iStep \) is the number associated to the current time step (0 at the beginning date of simulation, the number is increased by one at each time step simulated),
- \( p_name \) is the base name to search 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 transition from one time step to the following one,
- \( p_OneFile \) equals to true if a single archive is used to store continuation values.

**Remark 18** The previous object “SimulateStepSemilagrangControlDist” is used with MPI distribution of data for problems of quite high dimension. In the case of small dimension (below or equal to three), the parallelization with MPI or the sequential calculations can be achieved by the “SimulateStepSemilagrangControl” object.

This object implements the method “oneStep”

where:

- \( p_gaussian \) is a two dimensional array (number of Brownian in the modelization by the number of Monte Carlo simulations).
- \textit{p.statevector} stores the continuous state (continuous state size by number of simulations)
- \textit{p.iReg} for each simulation gives the current regime number,
- \textit{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 “\text{getSimuFuncSize}” method of the optimizer and \(nbSimul\) the number of Monte Carlo simulations.

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

```c++
// Copyright (C) 2016 EDF
// All Rights Reserved
// This code is published under the GNU Lesser General Public License (GNU LGPL)
#ifndef USE_MPI
#include <memory>
#include <boost/random.hpp>
#include <Eigen/Dense>
#include "gener/BinaryFileArchive.hh"
#include "StOpt/semilagrangien/OptimizerSLBase.h"
#include "StOpt/semilagrangien/SimulateStepSemilagrangControlDist.h"

using namespace std;

double semiLagrangianSimuControlDist(const shared_ptr<StOpt::FullGrid> & p_grid,
    const shared_ptr<StOpt::OptimizerSLBase> & p_optimize,
    const function<double(const int &, const Eigen::ArrayXd &)> & p_funcFinalValue,
    const int & p_nbStep,
    const Eigen::ArrayXd & p_stateInit,
    const int & p_initialRegime,
    const int & p_nbSimul,
    const string & p_fileToDump,
    const bool & p_bOneFile)
{
    boost::mpi::communicator world;
    // store states in a regime
    Eigen::ArrayXXd states(p_stateInit.size(), p_nbSimul);
    for (int is = 0; is < p_nbSimul; ++is)
        states.col(is) = p_stateInit;
    // sence the regime number
    Eigen::ArrayXi regime = Eigen::ArrayXi::Constant(p_nbSimul, p_initialRegime);
    // test if one file generated
    string toDump = p_fileToDump;
    if (!p_bOneFile)
        toDump += "_" + boost::lexical_cast<string>(world.rank());
    gs::BinaryFileArchive ar(toDump.c_str(), "r");
    // name for continuation object in archive
    return 0;
}
```

122
string nameAr = "Continuation";

// cost function
Eigen::ArrayXXd costFunction = Eigen::ArrayXXd::Zero(p_optimize->getSimuFuncSize(), p_nbSimul);

// random generator and Gaussian variables
boost::mt19937 generator;
boost::normal_distribution<double> normalDistrib;
boost::variate_generator<boost::mt19937 &, boost::normal_distribution<double>> normalRand(generator, normalDistrib);

Eigen::ArrayXXd gaussian(p_optimize->getBrownianNumber(), p_nbSimul);

// iterate on time steps
for (int istep = 0; istep < p_nbStep; ++istep)
{
    for (int is = 0; is < gaussian.cols(); ++is)
        for (int id = 0; id < gaussian.rows(); ++id)
            gaussian(id, is) = normalRand();

    StOpt::SimulateStepSemilagrangControlDist(ar, p_nbStep - 1 - istep, nameAr, p_grid, p_grid, p_optimize, p_bOneFile).
    oneStep(gaussian, states, regime, costFunction);
}

// final cost to add
for (int is = 0; is < p_nbSimul; ++is)
    costFunction(0, is) += p_funcFinalValue(regime(is), states.col(is));

// average gain/cost
return costFunction.mean();

#endif

The sequential (or parallelized on calculations) version of the previous example is given in this file.

Remark 19 In the previous example, we suppose that only one function is followed in simulation, and that we send back an average for this value function as a result.
Table 8.2: Which simulation object to use depending on the TransitionStepSemilagrang object used.

<table>
<thead>
<tr>
<th></th>
<th>“TransitionStepSemilagrang”</th>
<th>“TransitionStepSemilagrangDist”</th>
<th>“TransitionStepSemilagrangDist”</th>
</tr>
</thead>
<tbody>
<tr>
<td>“SimulateStepSemilagrang”</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>“SimulateStepSemilagrangControl”</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>“SimulateStepSemilagrangDist”</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>“SimulateStepSemilagrangControlDist”</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Part V

An example with both dynamic programming with regression and PDE
In this chapter we give an example where both dynamic programming with regressions and PDE can be used. It permits to compare the resolution and the solution obtained by both methods. All information about the modelization can be obtained by [26].

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 electricity production to satisfy the 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 intensity of investment in non emissive technology at date $s$,

- $Y_t$ is the carbon price where
  \[ Y_t = \mathbb{E}_t(\lambda 1_{Q_T \geq H}), \]
  with $\lambda$ and $H$ given.

We introduce the following functions:

- the electricity price function which is a function of demand and the global investment of non emissive technology.
  \[ p_t = (1 + D_t)^2 - L_t, \]

- the profit function by selling electricity is given by
  \[ \Pi(D_t, L_t) = p_t D_t - (D_t - L_t)^+, \]

- $\bar{c}(l_t, L_t)$ is the investment cost for new capacities of non emissive technology.
  \[ \bar{c}(l, L) = \bar{\beta}(c_\infty + (c_0 - c_\infty)e^{\beta L})(1 + l)l \]

The value of the firm selling electricity is given by $V(t, D_t, Q_t, L_t)$. It satisfies the coupling equations:

\[
\begin{cases}
\partial_t v + \alpha(m - D)\partial_D v + \frac{1}{2}\sigma^2 \partial^2_{DD} v + (D - L)^+ \partial_Q v + \Pi(D, L) \\
+ sL^{1-a} - y(D - L)^+ + \sup_{l} \{l \partial_L v - \bar{c}(l, L)\} = 0 \\
\end{cases}
\]

\[ v_T = 0 \] (8.1)
and the carbon price \( y(t, D_t, Q_t, L_t) \) is given by:

\[
\begin{aligned}
\partial_t y + \alpha (m - D) \partial_D y + \frac{1}{2} \sigma^2 \partial^2_{DD} y + (D - L)^+ \partial_Q y + l^* \partial_L y &= 0 \\
y_T &= \lambda 1_{Q_T \geq K}
\end{aligned}
\]  

(8.2)

and \( l^* \) is the optimal control in equation (8.1). The previous equation can be solved with the Semi Lagrangian method.

After a time discretization with a step \( \delta t \) a dynamic programming equation can be given by

\[
v(T - \delta t, D, Q, L) = \sup_l \left( \Pi(D, L) + sL^{1-\alpha} - y_{T-\delta t}(D - L)^+ - \tilde{c}(l, L) \right) \delta t + \\
\mathbb{E}_{T-\delta t}(V(T, D_{T-\delta t}, Q + (D - L)^+ \delta t, L + l\delta t))
\]  

(8.3)

\[
Y(T - \delta t, D, Q, L) = \mathbb{E}_{T-\delta t}(Y(T, D_{T-\delta t}, Q + (D - L)^+ \delta t, L + l^* \delta t))
\]  

(8.4)

The previous equations (8.3) and (8.4) can be solved with the regression methods.

In order to use the previously developed frameworks in parallel, we have to define for both method some common variables.

- The number of regimes to use (obtained by the "getNbRegime' method) is 2: one to store the \( v \) value, one for the \( y \) value,

- In the example we want to follow during simulations the functions values \( v \) and \( y \) so we set the number of function obtained by the "getSimuFuncSize" method to 2.

- In order to test the controls in optimization and simulation we define a maximal intensity of investment \( l^\text{Max} \) and a discretization step to test the controls \( l^\text{Step} \).

In the sequel we store the optimal functions in optimization and recalculate the optimal control in simulation.

### 8.3 The dynamic programming with regression approach

All we have to do is to specify an optimizer defining the methods used to optimize and simulate, and the "getCone" method for parallelization:

```cpp
// Copyright (C) 2016 EDF
// All Rights Reserved
// This code is published under the GNU Lesser General Public License (GNU LGPL)
#include "StOpt/core/utils/constant.h"
#include "OptimizeDPEmissive.h"

using namespace std ;
using namespace StOpt ;
using namespace Eigen ;

// constructor
OptimizeDPEmissive::OptimizeDPEmissive(const double &p_alpha ,
const std::function<double(double,double)> &p_PI ,
```
const std::function<double(double, double)> &p_cBar, const double &p_s, const double &p_lambda,
const double &p_dt,
const double &p_maturity,
const double &p_lMax, const double &p_lStep,
const std::vector<std::array<double, 2>> &p_extrem):
    m_alpha(p_alpha), m_PI(p_PI),
    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),
    m_extrem(p_extrem)
{};
}

Array<bool, Dynamic, 1> OptimizeDPEmissive::getDimensionToSplit() const
{
    Array<bool, Dynamic, 1> bDim = Array<bool, Dynamic, 1>::Constant(2, true);
    return bDim;
}

// for parallelism
std::vector<std::array<double, 2>> OptimizeDPEmissive::getCone(const std::shared_ptr<StOpt::SpaceGrid> &p_grid,
    const std::array<double, 2> &p_xInit) const
{
    std::vector<std::array<double, 2>> xReached
    xReached[0][0] = p_xInit[0][0]; // Q only increases
    xReached[0][1] = m_extrem[0][1]; // whole domain due to demand which is unbounded
    xReached[1][0] = p_xInit[1][0]; // L only increases
    xReached[1][1] = p_xInit[1][1] + m_lMax * m_dt; // maximal increase given by the control
    return xReached;
}

// one step in optimization from stock point for all simulations
std::pair<ArrayXd, ArrayXd> OptimizeDPEmissive::stepOptimize(const std::shared_ptr<StOpt::SpaceGrid> &p_grid,
    const ArrayXd &p_stock,
    const std::vector<ContinuationValue> &p_condEsp,
    const std::vector<ContinuationValue> &p_condStrat) const
{
    std::pair<ArrayXd, ArrayXd> solutionAndControl;
    // to store final solution (here two regimes)
    solutionAndControl.first = ArrayXd::Constant(m_simulator->getNbSimul(),
        2, -StOpt::infty);
    solutionAndControl.second = ArrayXd::Constant(m_simulator->getNbSimul(),
        1, -StOpt::infty);
    // demand
    ArrayXd demand = m_simulator->getParticles().array().row(0).transpose();
    // Gain (size number of simulations)
    ArrayXd gain = m_simulator->getNbSimul());
    double gainSubvention = m_s * pow(p_stock(1), 1 - m_alpha); // subvention for non emissive energy
for (int is = 0; is < m_simulator->getNbSimul(); ++is)
    gain(is) = m_PI(demand(is), p_stock(1)) + gainSubvention; // gain by
          production and subvention
ArrayXd ptStockNext(2);
    // time to maturity
double timeToMat = m_maturity - m_simulator->getCurrentStep();
    // interpolator at the new step
for (int is = 0; is < m_simulator->getNbSimul(); ++is)
{
    for (int iAl = 0; iAl < m_lMax / m_lStep; ++iAl) // test all
        command for investment between 0 and lMax
    {
        double l = iAl * m_lStep;
        // interpolator at the new step
        ptStockNext(0) = p_stock(0) + std::max(demand(is) - p_stock(1),
            0.0) * m_dt;
        ptStockNext(1) = p_stock(1) + l * m_dt;
        // first test we are inside the domain
        if (p_grid->isInside(ptStockNext))
            {
                // create an interpolator at the arrival point
                std::shared_ptr<StOpt::Interpolator> interpolator = p_grid->
                    createInterpolator(ptStockNext);
                // calculate Y for this simulation with the optimal control
                double yLoc = p_condEsp[1].getASimulation(is, *interpolator);
                // local gain
                double gainLoc = (gain(is) - yLoc * std::max(demand(is) -
                    p_stock(1), 0.0) - m_cBar(l, p_stock(1))) * m_dt;
                // gain plus conditional expectation of future gains
                double condExp = gainLoc + p_condEsp[0].getASimulation(is, *
                    interpolator);
                if (condExp > solutionAndControl.first(is, 0)) // test
                    optimality of the control
                    {
                        solutionAndControl.first(is, 0) = condExp;
                        solutionAndControl.first(is, 1) = yLoc;
                        solutionAndControl.second(is, 0) = 1;
                    }
            }
    }
    // test if solution acceptable
    if (StOpt::almostEqual(solutionAndControl.first(is, 0), -StOpt::
        infty, 10))
    {
        // fix boundary condition
        solutionAndControl.first(is, 0) = timeToMat * (m_PI(demand(is),
            p_stock(1)) + m_s * pow(p_stock(1), 1.0 - m_alpha) - m_lambda *
            std::max(demand(is) - p_stock(1), 0.0));
        solutionAndControl.first(is, 1) = m_lambda; // Q est maximal !!
        solutionAndControl.second(is, 0) = 0.0; // fix control to zero
    }
}
        return solutionAndControl;
}
void OptimizeDPEmissive::stepSimulate(const std::shared_ptr<StOpt::SpaceGrid> &p_grid, const std::vector<StOpt::GridAndRegressedValue> &p_continuation, 
    StOpt::StateWithStocks &p_state, 
    const Ref<ArrayXd> &p_phiInOut) const 
{
    ArrayXd ptStock = p_state.getPtStock();
    ArrayXd ptStockNext(ptStock.size());
    double vOpt = -StOpt::infty;
    double gainOpt = 0.;
    double lOpt = 0.;
    double demand = p_state.getStochasticRealization()(0); // demand for this simulation

    ptStockNext(0) = ptStock(0) + std::max(demand - ptStock(1), 0.) * m_dt;
    double gain = m_PI(demand, ptStock(1)) + m_s * pow(ptStock(1), 1. -
    m_alpha); // gain from production and subvention
    double yOpt = 0.;
    for (int iAl = 0; iAl < m_lMax / m_lStep; ++iAl) // test all command for investment between 0 and lMax
    {
        double l = iAl * m_lStep;
        // interpolator at the new step
        ptStockNext(1) = ptStock(1) + l * m_dt;
        // first test we are inside the domain
        if (p_grid->isInside(ptStockNext))
        {
            // calculate Y for this simulation with the control
            double yLoc = p_continuation[1].getValue(ptStockNext, p_state, 
            getStochasticRealization());
            // local gain
            double gainLoc = (gain - yLoc * std::max(demand - ptStock(1), 0.)
            - m_cBar(1, ptStock(1))) * m_dt;
            // gain + conditional expectation of future gains
            double condExp = gainLoc + p_continuation[0].getValue(
            ptStockNext, p_state.getStochasticRealization());

            if (condExp > vOpt) // test optimality of the control
            {
                vOpt = condExp;
                gainOpt = gainLoc;
                lOpt = l;
                yOpt = yLoc;
            }
        }
    }
    p_phiInOut(0) += gainOpt; // follow v value
    p_phiInOut(1) = yOpt; // follow y value
    ptStockNext(1) = ptStock(1) + lOpt * m_dt; // update state due to control
    p_state.setPtStock(ptStockNext);
}

This case in dimension 2 for the stocks can be treated with interpolation on the full 2
dimensional grid and on a 2 dimensional sparse grid. Both versions of the resolution are given in a test case.

### 8.4 The PDE approach

We can do the same with the PDE approach using a simulator for the OU demand. We then define an optimizer and the methods used to optimize and simulate, and the “getCone” method for parallelization:

```cpp
#include <iostream>
#include "StOpt/core/utils/constant.h"
#include "OptimizeSLEmissive.h"

using namespace StOpt;
using namespace Eigen;
using namespace std;

// constructor
OptimizeSLEmissive::OptimizeSLEmissive(const double &p_alpha, const double &p_m, const double &p_sig, const std::function<double(double, double)> &p_PI,
        const std::function<double(double, double)> &p_cBar, const double &p_s, const double &p_dt,
        const double &p_lMax, const double &p_lStep, const double &p_extrem)
    : 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),
      m_lMax(p_lMax), m_lStep(p_lStep), m_extrem(p_extrem) {}

Array< bool, Dynamic, 1> OptimizeSLEmissive::getDimensionToSplit() const
{
    Array< bool, Dynamic, 1> bDim = Array< bool, Dynamic, 1>::Constant(3, true);
    return bDim ;
}

// for parallelism
vector< array< double, 2> > OptimizeSLEmissive::getCone(const vector< array< double, 2> > &p_xInit) const
{
    vector< array< double, 2> > xReached;
    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
    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
    xReached[1][0] = p_xInit[1][0]; // Q only increases
    xReached[1][1] = p_xInit[1][1] + m_extrem[0][1] * m_dt ; // Q increases bounded by maximal demand
    xReached[2][0] = p_xInit[2][0] ; // L only increases
}```
xReached[2][1] = p_xInit[2][1] + m_lMax * m_dt; // maximal increase given by the control
return xReached;

// one step in optimization from current point
std::pair<ArrayXd, ArrayXd> OptimizeSLEmissive::stepOptimize(const ArrayXd &p_point,
    const vector<shared_ptr<SemiLagrangEspCond>> &p_semiLag,
    const double &, const ArrayXd &) const
{
    pair<ArrayXd, ArrayXd> solutionAndControl;
solutionAndControl.first.resize(2);
solutionAndControl.second.resize(1);
ArrayXXd sig = ArrayXXd::Zero(3, 1);
sig(0, 0) = m_sig;
double vOpt = -StOpt::infty;
double yOpt = 0.;
double lOpt = 0;
ArrayXd b(3);
b(0) = m_alpha * (m_m - p_point(0)); // trend
b(1) = max(p_point(0) - p_point(2), 0.);
// gain already possible to calculate (production and subvention)
double gainFirst = m_PI(p_point(0), p_point(2)) + m_s * pow(p_point(2), 1. - m_alpha);
for (int iAl = 0; iAl < m_lMax / m_lStep; ++iAl) // test all command for investment between 0 and lMax
{
    double l = iAl * m_lStep;
b(2) = l;
    pair<double, bool> lagrangY = p_semiLag[1]->oneStep(p_point, b, sig, m_dt); // for the control calculate y
    if (lagrangY.second) // is the control admissible
    {
        pair<double, bool> lagrang = p_semiLag[0]->oneStep(p_point, b, sig, m_dt); // one step for v
        // gain function
        double gain = m_dt * (gainFirst - lagrangY.first * b(1) - m_cBar
            (1, p_point(2)));
        double arbitrage = gain + lagrang.first;
        if (arbitrage > vOpt) // optimality of the control
        {
            vOpt = arbitrage; // upgrade solution v
            yOpt = lagrangY.first; // store y
            lOpt = l; // upgrade optimal control
        }
    }
}
if (StOpt::almostEqual(vOpt, -StOpt::infty, 10))
{
    std::cout << " Reduce time step " << std::endl;
    abort();
}
solutionAndControl.first(0) = vOpt; // send back v function
solutionAndControl.first(1) = yOpt; // send back y function
solutionAndControl.second(0) = lOpt; // send back optimal control
return solutionAndControl;

// one step in simulation for current simulation
void OptimizeSLEmissive::stepSimulate(const SpaceGrid &p_gridNext,
        const std::vector<std::shared_ptr<
            StOpt::SemiLagrangEsCond>> & p_semiLag,
        Ref<ArrayXd> p_state, int &,
        const ArrayXd &p_gaussian,
        const ArrayXd &,
        Ref<ArrayXd> p_phiInOut) const
{  
    ArrayXd state = p_state;
    ArrayXXd sig = ArrayXXd::Zero(3, 1); // diffusion matrix for semi Lagrangian
    sig(0, 0) = m_sig;
    double vOpt = -StOpt::infty;
    double lOpt = 0;
    double yOpt = 0;
    ArrayXd b(3);
    b(0) = m_alpha * (m_m - p_state(0)); // trend for D (independent of control)
    b(1) = max(p_state(0) - p_state(2), 0.); // trend for Q (independent of control)
    double gainFirst = m_PI(p_state(0), p_state(2)) + m_s * pow(p_state(2),
        1. - m_alpha); // gain for production and subvention
    for (int iAl = 0; iAl < m_lMax / m_lStep; ++iAl) // recalculate the optimal control
    {
        double l = iAl * m_lStep;
        b(2) = l;
        pair<double, bool> lagrangY = p_semiLag[1]->oneStep(p_state, b, sig,
            m_dt); // calculate y for this control
        if (lagrangY.second)
        {
            // calculate the function value v
            pair<double, bool> lagrang = p_semiLag[0]->oneStep(p_state, b,
                sig, m_dt); // calculate the function value v
            double gain = m_dt * (gainFirst - lagrangY.first * b(1) - m_eBar
                (1, p_state(2)));
            double arbitrage = gain + lagrang.first;
            if (arbitrage > vOpt) // arbitrage
            {
                vOpt = arbitrage; // upgrade solution
                yOpt = lagrangY.first; // upgrade y value
                lOpt = l; // upgrade optimal control
            }
        }
    }
}
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.
Part VI

Stochastic Dual Dynamic Programming
Chapter 9

SDDP algorithm

9.1 Some general points about SDDP

SDDP is an approximate dynamic programming algorithm developed by Pereira and Pinto in 1991 [27].

To describe how SDDP works, we will consider a class of linear programs that have $T$ stages denoted $\{0,1,...,t,...,T\}$. We restrict our class of problems to linear programs with relatively complete recourse: the feasible region of the linear program in each stage is nonempty and bounded.

Let us formalize now the variables and constraints used in the SDDP problem.

Notations used

The notations described here are used in the general case.

- $x_t$ the state variable at time $t$,
- $\omega_t \in \Omega_t$ the random data process at time $t$, where $\Omega_t$ is the set of random data.
- $c_t$ is the cost vector at time $t$,
- $A_t$ and $E_t$ denote constraints matrices.
- $Q_t(x_{t-1}, \omega_t)$ is the expected value of the problem at time $t$, knowing the state $x_{t-1}$ and the random data $\omega_t$.
- $Q_t(x_{t-1}) = E[Q_t(x_{t-1}, \omega_t)]$

Decision process

The random data process $\omega_t$ is discovered gradually. Thus from an initial state $x_0$, the state variables $(x_t)_{t \in \{0,1,...,T\}}$ are determined in a non-anticipative way. The scheme is the following:

$x_0 \rightarrow$ observation of $\omega_1 \rightarrow$ decision of $x_1$ ....
$\rightarrow$ decision of $x_{T-1} \rightarrow$ observation of $\omega_T \rightarrow$ decision of $x_T$
A rigorous formulation of the multistage stochastic linear program to solve is the following:

\[
V^* = \min_{A_0 x_0 = \omega_0 \atop x_1 \geq 0} c_0^\top x_0 + \mathbb{E} \left[ \min_{E_1 x_0 + A_1 x_1 = \omega_1} c_1^\top x_1 + \mathbb{E} \left[ \ldots + \mathbb{E} \left[ \min_{E_T x_T = \omega_T} c_T^\top x_T \right] \right] \right] \tag{9.1}
\]

The deterministic equivalent of this problem (9.1) is achieved by discretizing \( \omega_t \) (or by using directly \( \omega_t \) if discrete). The number of variables of this problem increases exponentially with the number of stages. It cannot be solved directly even if \( T \) or \( \Omega_t \in \{0, 1, \ldots, T\} \) are of reasonable size.

**Dynamic programming principle**

Dynamic programming involves splitting up the problem (9.1) in a series of sub-problem bounded together by a state variable. The aim is to compute backwards the functions \( Q_t \) and \( \mathcal{Q}_t \). They fulfill the following equations:

\[
\begin{align*}
Q_t(x_{t-1}, \omega_t) &= \min c_t^\top x_t + \mathcal{Q}_{t+1}(x_t) \\
\text{s.c.} \quad A_t x_t &= \omega_t - E_t x_{t-1}, \quad [\pi_t(\omega_t)] \\
x_t &\geq 0
\end{align*}
\tag{9.2}
\]

\[
\mathcal{Q}_t(x_{t-1}) = \mathbb{E}[Q_t(x_{t-1}, \omega_t)] \tag{9.3}
\]

The function \( Q(x_{t-1}, \omega_t) \) stands for the expected value of a future cost knowing the state \( x_{t-1} \) the random data \( \omega_t \). \( \mathcal{Q}_t(x_{t-1}) \) is the expected value of the future cost knowing the state. The dynamic programming principle insures 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 (9.2) and the computation of (9.3).

The implementation of dynamic programming involves approximating successively the two value functions with equations (9.2 - 9.3) by discretizing the state space and solving the linear sub-problems. The number of discretization points increases exponentially with the dimension of the state vector and becomes huge for our applications ("curse of dimensionality").

**SDDP algorithm**

SDDP is a method used to solve stochastic multi-stage problem described in [27]. SDDP is based on Benders decomposition described in [28]. Please note that SDDP was developed in order to solve hydro thermal scheduling problem.

SDDP gets round the curse of dimensionality by avoiding a priori complete discretization of the state space. Each SDDP iteration is a two-stage process. The first step involves generating a sequence of realistic states \( x_t^* \) from which in the second step the value functions are estimated in their neighborhood. By repeating successively these two steps the approximation of the value function becomes more and more accurate. SDDP is also made of two passes computed alternatively:
• a backward pass: the aim is to improve the number of Benders cut in the neighborhood of well-chosen candidate states. It provides also a lower bound of the optimal cost.

• a forward pass: the aim is to provide a set of new candidate states. An estimation of the upper bound of the optimal cost is also computed.

On the other hand SDDP method stands on the shape of the future value function $\mathcal{Q}_t(x_{t-1})$. Indeed in the frame of a linear problem with complete recurse the value function is convex and piecewise linear. It can therefore be approximated by taking the supremum of a family of minoring affine functions. These affine functions are called *optimality cuts* or *Benders cuts*.

### 9.2 A method, different algorithms

The method implemented in this library is based on the different situations shown in a technical report of PSR program [29] where three different cases of the basic problem are solved by SDDP. The three cases are implemented in the library. Other cases could be added to those existing in the future.

**Notations**

These notations will be used to present the different algorithm of SDDP.

- $\bar{z}$ denotes the optimal cost obtained in forward pass.
- $\underline{z}$ denotes the optimal cost obtained in backward pass.
- $\beta_j^t$ denotes the slope of the $j^{th}$ Benders cut.
- $\alpha_j^t$ denotes the intercept of the $j^{th}$ Benders cut.

### 9.2.1 The basic case

To describe this case the notations shown above are used. We focus on stochastic multi-stage problems with the following properties.

- Random quantities in different stages are independent.
- The random quantities at time $t$ is summarized in $\omega_t$.
- At each stage, the linear sub-problem solution space is non-empty 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 is working.
Initialization
The following values are fixed:

- \{0, 1, ..., T\}, 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 reach for the convergence test.
- \(n_{\text{step}} \in \mathbb{N}\), the number of iterations achieved between 2 convergence tests.
- \(n_{\text{iterMax}} \in \mathbb{N}\), the maximal number of iterations.
- \(x_0 \in \mathbb{R}^n_+\), the initial vector state.
- \(L \in \mathbb{N}\), the number of scenarios used in the backward pass.
- \(G \in \mathbb{N}\), the number of scenarios used in the forward pass. It gives also the number of new cuts computed at every iteration (backward-forward) and the number of states near which the Benders cuts are computed.

Forward pass
The aim of this pass is to explore new feasible vector state and to get an estimation of the upper bound of the optimal cost. To this end the current strategy is simulated for a set of \(G\) scenarios. The set of scenarios could be historical chronicles or random draws.
Algorithm 4: Run of forward pass ($n^{th}$ iteration)

Simulate sets $\{(\omega^g_t), t \in \{1, \ldots, T\}\}$ of equally distributed scenarios: for $g \in \Omega^g = \{1, \ldots, G\}$:

for $g \in \Omega^g$ do

Solve the following linear sub-problem:

\[ [AP^n_{0,g}] \] \[
\begin{cases}
Q_0 = \min_{x_0, \theta_1} c_0^\top x_0 + \theta_1 \\
u.c.
A_0 x_0 = \omega_0, \quad [\pi_0(\omega_0)] \\
x_0 \geq 0 \\
\theta_1 + (\beta_1^g)^\top x_0 \geq \alpha_1^g \quad j \in \{1, \ldots, G, \ldots, nG\}
\end{cases}
\] \quad (9.4)

Store the primal solution $(x^g_0)$ of the problem $[AP^n_{0,g}]$.

end

for $t \in \{1, \ldots, T\}$ do

Solve the following linear sub-problem:

\[ [AP^n_{t,g}] \] \[
\begin{cases}
Q^g_t(x^g_{t-1}, \omega^g_t) = \min_{x_t, \theta_{t+1}} c_t^\top x_t + \theta_{t+1} \\
s.c.
A_t x_t = \omega^g_t - E_t x^g_{t-1}, \quad [\pi_t(\omega^g_t)] \\
x_t \geq 0 \\
\theta_{t+1} + (\beta^g_j)^\top x_t \geq \alpha^g_{j, t+1}, \quad j \in \{1, \ldots, G, \ldots, nG\}
\end{cases}
\] \quad (9.5)

Store the primal solution $(x^g_t)$ of the problem $[AP^n_{t,g}]$.

end

Compute the cost for scenario $g$, at iteration $n$:

\[ \bar{z}^g_n = \sum_{t=0}^{T} c_t x^g_t; \]

end

Compute the total cost in forward pass at iteration $n$:

\[ \bar{z}_n = \frac{1}{G} \sum_{g=1}^{G} \bar{z}^g_n \]

---

**Backward pass**

The aim of the backward pass is to add at each stage a set of new Benders cut and to provide a new estimation of the lower bound of the optimal operational cost. To this end we have scenarios set of the random quantities (dimension of the set is $L$) recorded during the initialization.
Algorithm 5: Run of backward pass

for \( t = T, T-1, \ldots, 1 \) do
for \( \omega_{t-1}^g, g \in \{1, \ldots, G\} \) do
for \( \omega_t^l, l \in \{1, \ldots, L\} \) do

Solve the following linear sub-problem:

\[
\begin{aligned}
& \min_{x_t, \theta_{t+1}} \ c_t^T x_t + \theta_{t+1} \\
\text{s.c.} & \ A_t x_t = \omega_t^l - E_t x_{t-1}^g, \quad [\pi_t(\omega_t^l)] \\
& x_t \geq 0 \\
& \theta_{t+1} + (\beta_{t+1}^j)^T x_t \geq \alpha_{t+1}^j, \quad j \in \{1, \ldots, G, \ldots, (n+1)G\}
\end{aligned}
\]  

(9.6)

Store 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 \( [AP_{t,l}^{n,g}] \)

Compute the cut that goes with the \( l^{th} \) hazard draw:

\[
\begin{aligned}
\alpha_{t,l}^g &= Q_{t}^l(x_{t-1}^g, \omega_t^l) + \pi_t(\omega_t^l)^T E_t x_{t-1}^g \\
\beta_{t,l}^g &= E_t^T \pi_t(\omega_t^l)
\end{aligned}
\]  

(9.7)

end

Compute the \( g^{th} \) new Benders cut at time \( t \) at iteration \( n \): is defined as the mean value of the cuts obtained before:

\[
\begin{aligned}
\alpha_t^k &= \frac{1}{L} \sum_{l=1}^L \alpha_{t,l}^g \\
\beta_t^k &= \frac{1}{L} \sum_{l=1}^L \beta_{t,l}^g
\end{aligned}
\]  

(9.8)

end

Solve the following linear sub-problem:

\[
\begin{aligned}
& \min_{x_0, \theta_1} \ c_0^T x_0 + \theta_1 \\
\text{s.c.} & \ A_0 x_0 = \omega_0, \quad [\pi_0(\omega_0)] \\
& x_0 \geq 0 \\
& \theta_1 + (\beta_1^j)^T x_0 \geq \alpha_1^j, \quad j \in \{1, \ldots, G, \ldots, (n+1)G\}
\end{aligned}
\]  

(9.9)

Save the cost backward \( z_n = Q_0 \)

---

**Stopping test**

In the literature about SDDP lots of stopping criterion 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 criterion are implemented in the library. These can be customized by the user. The first one defines a maximal number of iterations \( n_{\text{iterMax}} \) (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:

\[ \psi_{n_{\text{step}}^i} = \left| \bar{z}_{n_{\text{step}}^i} - \bar{z}_{n_{\text{step}}^i} \right|, \quad \text{with } i \in \mathbb{N} \]  

(9.10)

This one is computed every \( n_{\text{step}} \) iterations. If it is lesser than a threshold \( p \) the process stops, otherwise it goes on. The threshold is fixed by the user.

### 9.2.2 Dependence of the random quantities

In the previous case we restrict our problem to independent random quantities in the different stages. The resolution of the SDDP was achieved 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 exists time-related dependency of the outcomes. Time-related dependencies can also exist in the demand. Yet with time-related random quantities the Bellman recurrence formula (9.2-9.3) does not hold and the classical SDDP can not be applied.

However if the Bellman functions are convex with respect to the time-related random quantities one has only to increase the dimension of the state vector by the dimension of the time-related random quantities to be back in the configuration of the basic case. In this case solving a linear program of reasonable size for each hazard draw is enough to compute new Benders cuts computation in the neighborhood of a candidate state.

There exists a few options to represent the time-related dependency of the random quantities. However in order to not increase too much the dimension of the problem, a ARMA process of order 1 is often chosen. In the random data vector \( \omega_t \) two different parts has to be distinguished from now on:

- \( \omega_t^{\text{ind}} \) is the random data vector corresponding to the independent random quantities.
- \( \omega_t^{\text{dep}} \) is the random data vector corresponding to the time-related random quantities.

And \( \omega_t^{\text{dep}} \) fulfills the following recurrence equation:

\[ \frac{\omega_t^{\text{dep}} - \mu_{\omega,t}}{\sigma_{\omega,t}} = \psi_1 \frac{\omega_{t-1}^{\text{dep}} - \mu_{\omega,t-1}}{\sigma_{\omega,t-1}} + \psi_2 \epsilon_t \]  

(9.11)

To apply the Bellman recurrence formula the vector state should be made of the decision variable \( x_t \) and the time-related random quantities \( \omega_t^{\text{dep}} \). Dimension of the vector state is then increased. \( x_t^{\text{dep}} = (x_t, \omega_t^{\text{dep}})^\top \) denotes the new state vector. The Bellman function satisfies from now on the following two-stages linear problem at time \( t \):

\[
\begin{align*}
[LP_t^f] \quad \min_{x_t} \quad & Q_t(x_{t-1}, \omega_{t-1}^{\text{dep}}, \omega_t) = x_t^\top \pi_t(x_t, \omega_t^{\text{dep}}) \\
\text{u.c.} \quad & A_{t-1} x_t = P \omega_t^{\text{dep}} - E_t x_{t-1}, \quad [\pi_t(\omega_t)] \\
& x_t \geq 0
\end{align*}
\]

(9.12)

with \( P \) the matrix such that \( \omega_t = P \omega_t^{\text{dep}} \).
The variable $\omega_{\text{dep}}^t$ is a random process. Thus the above problem is solved using specific values $\omega_{\text{dep}}^t$ of this variable. To get them we apply a Markov process that is we simulate different values of the white noise $\epsilon_{\text{dep}}^t$.

The new form of the state vector implies changes in the sensitivity of the Bellman function. Thus it is a function depending on the decision variable $x_t$ but also on the the time-related random quantity vector $\omega_{\text{dep}}^t$. The computation of Benders cuts is then a bit different:

$$\frac{\partial Q_t(x_{t-1}, \omega_{\text{dep}}^{t-1}, \omega_t)}{\partial \omega_{\text{dep}}^{t-1}} = \frac{\partial Q_t(x_{t-1}, \omega_{\text{dep}}^{t-1}, \omega_t)}{\partial \omega_t^{\text{dep}}} \cdot \frac{\partial \omega_t^{\text{dep}}}{\partial \omega_{\text{dep}}^{t-1}}$$

$$= \pi_t(\omega_t)^\top \psi_1 \frac{\sigma_{\omega,t}}{\sigma_{\omega,t-1}}.$$

(9.13)

Backward pass has to be modified in the following manner. Some new computation steps have to be taken into account.
Algorithm 6: Run of backward pass with time-related random quantities (AR1 process)

Pick up the set of the following pairs: \( \{ x^g_t, \omega^d_{t, g} \} \) for \( g \in \{ 1, \ldots, G \} \), \( t \in \{ 1, \ldots, T \} \)

for \( t = T, T - 1, \ldots, 1 \) do

for \( \{ x^g_t, \omega^d_{t, g} \}, g \in \{ 1, \ldots, G \} \) do

for \( l \in \{ 1, \ldots, L \} \) do

Produce a value for the white noise \( c^l_t \);

Compute the element \( \omega^l_t \) knowing the previous random quantity \( \omega^d_{t-1} \):

\[
\hat{\omega}^l_t = \sigma_{\omega, t} \left( \psi_1 \frac{\omega^d_{t-1} - \mu_{\omega, t-1}}{\sigma_{\omega, t-1}} + \psi_2 s^l_t \right) + \mu_{\omega, t} \tag{9.14}
\]

Solve the following linear sub-problem:

\[
\begin{align*}
[AP^{'n,g}] & \quad \begin{cases} 
Q_t^l(x^g_{t-1}, \omega^d_{t-1}, \omega^l_t) = \min_{x_t, \theta_{t+1}} c^T_t x_t + \theta_{t+1} \\
\text{u.c.} \quad A_t x_t = P_t^2 - E_t x^g_{t-1}, \quad [\pi_t(\omega^l_t)] \\
\quad x_t \geq 0 \\
\quad \theta_{t+1} + (\beta^l_{t+1})^T x_t + (\gamma^l_{t+1})^T \omega^l_t \geq \alpha^l_{t+1}, \\
\quad j \in \{ 1, \ldots, G \}
\end{cases}
\end{align*} \tag{9.15}
\]

Store the dual solution \( \pi_t(\omega^l_t) \) and the primal one \( Q_t^l(x^g_{t-1}, \omega^d_{t-1}, \omega^l_t) \) of the primal problem \([AP^{'n,g}]\)

Compute the cut that goes with the \( k \)th hazard draw:

\[
\begin{align*}
\alpha^g_{t,l} &= Q_t^l(x^g_{t-1}, \omega^d_{t-1}, \omega^l_t) + \pi_t(\omega^l_t)^T (E_t x^g_{t-1} - \psi_1 \sigma_{\omega, t-1} P_t^1 \omega^d_{t-1}) \\
\beta^g_{t,l} &= E_t^T \pi_t(\omega^l_t) \\
\gamma^g_{t,l} &= \psi_1 \sigma_{\omega, t-1} P^T \pi_t(\omega^l_t) \\
\end{align*} \tag{9.16}
\]

Compute the \( g \)th new Benders cut at time \( t \) at iteration \( n \) defined as the mean value of the cuts obtained before:

\[
\begin{align*}
\alpha_t &= \frac{1}{L} \sum_{l=1}^L \alpha^g_{t,l} \\
\beta_t &= \frac{1}{L} \sum_{l=1}^L \beta^g_{t,l} \\
\gamma_t &= \frac{1}{L} \sum_{l=1}^L \gamma^g_{t,l} \\
& \quad k = nG + g
\end{align*} \tag{9.17}
\]

end

Solve the following linear sub-problem:

\[
[AP_{n}^{'n}] \quad \begin{cases} 
Q_0 = \min_{x_0, \theta_1} c_0^T x_0 + \theta_1 \\
\text{u.c.} \quad A_0 x_0 = \omega_0, \quad [\pi_0(\omega_0)] \\
\quad x_0 \geq 0 \\
\quad \theta_1 + (\beta^g_1)^T x_0 + (\gamma^g_1)^T \omega_0^{d_{\omega, t}} \geq \alpha^g_1, \\
\quad j \in \{ 1, \ldots, G \}
\end{cases}
\] \tag{9.18}

Save the backward cost \( z_n = Q_0 \)

144
9.2.3 Non-convexity and conditionnal cuts

Some random quantities may introduce non-convexity preventing us to apply the classical algorithm of SDDP. Indeed when the random quantities appear on the right-hand side of the linear constraints or in the cost function (typically $A_t$ and/or $c_t$ become random) the property of non-convexity of the Bellman functions with respect to the random quantities is not anymore observed.

In the frame of a management production problem the situation happened often. For example sometimes the unit operation cost of plants are random. It is also observed when we deal with spot price uncertainty for use in stochastic mid-term scheduling.

In a technical report [29] Pereira and Pinto suggested a new algorithm in order to efficiently approximate the Bellman functions using explicitly the dependence of the Bellman functions with respect to these random quantities. This new algorithm is based on a combination of SDDP and ordinary stochastic dynamic programming. The SDP part deals with the non-convex random quantities, whereas the other random quantities are treated in the SDDP part. It is an extension of the classical SDDP algorithm. It is described in detail in [29] and in [30].

In that case the modelization used in the library is somewhat different from the one described in both articles. Indeed in the articles it is based on a finite number of non-convex random quantities. A discretization of the non-convex random quantity space is the necessary.

In [30] spot price $p_t$ is regarded as a state. The set of feasible spot price is discretized into in a set of $M$ points $\zeta_1, \ldots, \zeta_M$. The following Markov model is then used:

$$ P(p_t = \zeta_j | p_{t-1} = \zeta_i) = \rho_{ij}(t) $$

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.

In order to overcome these points in the library the evolution of the non-convex random quantities is decided by Monte Carlo simulations. At each stage a fixed number of Monte-Carlo simulations is provided. Anyway in spite of this difference the global view of this brand new algorithm is similar to that one described in both articles:

- The non-convex random quantities depend on the realization of the previous one according to a mathematical model (Markov chain).
- At each stage Bellman functions are approximated through the conditional realization of these random quantities.
- We used conditional cuts to give an estimation of the Bellman functions.

In our algorithm the features of the conditional cuts are revealed thanks to a conditional expectation computation.

Yet conditional expectation computations are not easy when the exact distribution of the random variable is not known. A few 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, stochastic dynamic programming and SDDP

The run of the backward pass in the new algorithm combining SDDP and SDP using local linear regression is described below.

Before describing in detail this algorithm, let us introduce a few notations:

- $S$ is the space of the non-convex random quantities.
- $d$ is the dimension of the space of the non-convex random quantities $S$.
- At each stage $U$ Monte Carlo simulations in $S$ are provided. Thus we get $U$ scenarios denoted $s_t^u$ at each stage $t$.
- $\tilde{I}$ is a partition of the space of the non-convex random quantities $S$.

\[ \tilde{I} = \{ I = (i_1, ..., i_d) : i_1 \in \{1, ..., I_1\}, ..., i_d \in \{1, ..., I_d\} \} \]

- $\{D_L\}_{L \in \tilde{I}}$ is the set of meshes of the set of scenarios.
- $M_M = \prod_{k=1}^{d} I_k$ denotes the number of meshes at each stage.
Algorithm 7: Run of the backward pass with time-related (AR1) and non-convex random quantities

Pick up the set of the following pairs: \( \{x^q_t, \omega^q_{t,dep} \} \) for \( g \in \{1, \ldots, G \} \), \( t \in \{1, \ldots, T \} \)

\begin{algorithm}
\begin{algorithmic}
\For{t = T, T - 1, \ldots, 1}
\State Generate values for the non-convex random quantities at time \( t \) knowing the scenarios at time \( t - 1 \): \( s^u_t, u \in \{1, \ldots, U\} \)
\For{(\( \omega_{t-1}^q, \omega_{t-1,dep}^q \)), \( g \in \{1, \ldots, G\} \)}
\For{u \in \{1, \ldots, U\}}
\State Produce a value for the white noise \( e^t_u \);
\State Compute the element \( \omega^t_u \) knowing the previous random quantity \( \omega_{t-1}^q \):
\[ \hat{\omega}_t^u = \sigma_{\omega,t} \left( \psi_1 \frac{\omega_{t-1}^{dep,q}}{\sigma_{\omega,t-1}} + \psi_2 e^t_u \right) + \mu_{\omega,t} \] (9.19)
\EndFor
\State Pick up the cuts corresponding the mesh \( D_L : \{ \alpha^L_{t,l} (s), \beta^L_{t,l} (s), \gamma^L_{t,l} (s) \} \). \( j \in \{1; \ldots; (n + 1)G\} \)
\State Solve the following linear sub-problem:
\[ \begin{align*}
&[\text{AP}^n_{t,l}] \\
\text{s.c.} & \quad A_I (s^u_t) x_t = P \omega_{t-1}^q - E_t x^u_{t-1}, \quad [\pi_t (\omega^t_u, s^u_t)] \\
& \quad x_t \geq 0 \\
& \quad \theta_{t+1} (s^u_t) + (\dot{\beta}^L_{t,l} (s^u_t))' x_t + (\gamma^L_{t,l} (s^u_t))' x_t \geq \alpha^L_{t,l} (s^u_t), \quad j \in \{1, \ldots, G, \ldots, (n + 1)G\}
\end{align*} \] (9.20)
\State Store the dual solution \( \pi_t (\omega^t_u) \) and the primal solution \( Q^l_t (s^u_{t-1}, \omega^q_{t-1,dep}, \omega^t_u, s^u_t) \) of the problem \([\text{AP}^n_{t,l}]\)
\State Calculate the corresponding cut at the \( l^{th} \) draw of uncertainties:
\[ \begin{align*}
& \dot{\alpha}^L_{t,l} (s^u_t) = Q^l_t (s^u_{t-1}, \omega^q_{t-1,dep}, \omega^t_u) + \pi_t (\omega^t_u)'^{T} \left( E_t x^u_{t-1} - \psi_1 \frac{\omega^q_{t-1,dep}}{\pi_{t-1} \omega_{t-1}} \right) \omega^q_{t-1,dep} \\
& \dot{\beta}^L_{t,l} (s^u_t) = E_t \pi_t (\omega^t_u) \\
& \dot{\gamma}^L_{t,l} (s^u_t) = \psi_t \frac{\omega^q_{t-1,dep}}{\pi_{t-1} \omega_{t-1}}^{T} \pi_t (\omega^t_u)
\end{align*} \] (9.21)
\EndFor
\EndFor
\State Compute the cut for a non-convex random quantity \( s^u_t \) at time \( t \) at iteration \( n \); it is defined as the weighted average on the \( L \) Benders cut obtained before:
\[ \begin{align*}
& \dot{\alpha}^L_{t,l} (s^u_t) = \frac{1}{L} \sum_{l=1}^{L} \dot{\alpha}^L_{t,l} (s^u_t) \\
& \dot{\beta}^L_{t,l} (s^u_t) = \frac{1}{L} \sum_{l=1}^{L} \dot{\beta}^L_{t,l} (s^u_t), \quad j = nG + g \\
& \dot{\gamma}^L_{t,l} (s^u_t) = \frac{1}{L} \sum_{l=1}^{L} \dot{\gamma}^L_{t,l} (s^u_t)
\end{align*} \] (9.22)
\EndFor
\EndFor
\State Compute the \( g^{th} \) new cut of the mesh \( D_L \) at time \( t \) at iteration \( n \); it is defined as the conditional expectation with respect to the scenario \( u \) at time \( t \)
\[ \begin{align*}
& \dot{\alpha}^L_{t,l} (s^u_{t-1}) = \mathbb{E} \left[ \dot{\alpha}^L_{t,l} (s^u_t) | s^u_{t-1} \right] \\
& \dot{\beta}^L_{t,l} (s^u_{t-1}) = \mathbb{E} \left[ \dot{\beta}^L_{t,l} (s^u_t) | s^u_{t-1} \right], \quad j = nG + g \\
& \dot{\gamma}^L_{t,l} (s^u_{t-1}) = \mathbb{E} \left[ \dot{\gamma}^L_{t,l} (s^u_t) | s^u_{t-1} \right]
\end{align*} \] (9.23)
\EndFor
\State Solve the initial linear sub problem \([\text{AP}^{0,n}]\)
\State Save the backward cost \( z_n = Q_0 \)
\end{algorithm}
\end{algorithm}
9.3 C++ API

The SDDP part of the stochastic library is in C++ code. This unit is a classical black box: specific inputs have to be provided in order to get the expected results. In the SDDP unit backward and forward pass are achieved successively until the stopping criterion is reached. In this unit the succession of passes is realized by backwardForwardSDDP class. This class takes as input three non-defined classes.

9.3.1 Inputs

The user has to implement three classes.

- One class where the transition problem is described which is denoted in the example TransitionOptimizer. This class is at the core of the problem resolution. Therefore much flexibility is let to the user to implement this class. In some ways this class is the place where the technical aspects of the problem are adjusted. This class describes backward and forward pass. Four methods should be implemented:
  - oneStepForward: solves the different transition linear problem during the forward pass for a particle, a random vector and an initial state.
  - oneStepBackward: solves the different transition linear problem during the backward pass for a particle, a random vector and an initial state.
  - updateDates: update the dates at time $t$.
  - oneAdmissibleState: returns an admissible state at time $t$ (respect only the constraints)

TransitionOptimizer should derive from the OptimizerSDDPBase class defined below.

```
// Copyright (C) 2016 EDF
// All Rights Reserved
// This code is published under the GNU Lesser General Public License (GNU LGPL)

#define OPTIMIZERSDDPBASE_H
#define OPTIMIZERSDDPBASE_H
#include <Eigen/Dense>
#include "StOpt/sddp/SDDPCutBase.h"
#include "StOpt/core/grids/OneDimRegularSpaceGrid.h"
#include "StOpt/core/grids/OneDimData.h"
#include "StOpt/sddp/SimulatorSDDPBase.h"

/** 
 * \brief Define an abstract class for Stochastic Dual Dynamic Programming problems 
 * \author Xavier Warin 
 */
```
namespace StOpt
{
    
    //\class OptimizerSDDPBase OptimizerSDDPBase.h
    // Base class for optimizer for Dynamic Programming
    class OptimizerSDDPBase
    {

    public :

        OptimizerSDDPBase() {}

        virtual ~OptimizerSDDPBase() {} 

        
        //\brief Optimize the LP during backward resolution
        //\param p_linCut cuts used for the PL (Benders for the Bellman value
        //at the end of the time step)
        //\param p_aState store the state, and 0.0 values
        //\param p_particle the particle n dimensional value associated to the
        //regression
        //\param p_isample sample number for independant uncertainties
        //\return a vector with the optimal value and the derivatives if the
        //function value with respect to each state
        virtual Eigen::ArrayXd oneStepBackward(const std::unique_ptr<StOpt::SDDPCutBase>& p_linCut, const std::tuple<std::shared_ptr<Eigen::ArrayXd>, int, int>& p_aState, const Eigen::ArrayXd &p_particle, const int &p_isample) const = 0;

        
        //\brief Optimize the LP during forward resolution
        //\param p_aParticle a particule in simulation part to get back cuts
        //\param p_linCut cuts used for the PL (Benders for the Bellman value
        //at the end of the time step)
        //\param p_state store the state, the particle number used in
        //optimization and mesh number associated to the particle. As an input
        //it constrains the current state
        //\param p_stateToStore for backward resolution we need to store  \f$ ( S_{t},A_{\{t-1\}},D_{\{t-1\}} ) \f$ where p_state in output is  \f$ ( S_{t},A_{\{t\}},D_{\{t\}} ) \f$
        //\param p_isample sample number for independant uncertainties
        virtual double oneStepForward(const Eigen::ArrayXd &p_aParticle, Eigen::ArrayXd &p_state, Eigen::ArrayXd &p_stateToStore, const std::unique_ptr<StOpt::SDDPCutBase>& p_linCut, const int &p_isample) const = 0;

        
        // brief update the optimizer for new date
        virtual void updateDates(const double &p_date, const double &p_dateNext) = 0;

        
        // brief Get an admissible state for a given date
        //\param p_date current date
// \\return an admissible state
virtual std::shared_ptr<Eigen::ArrayXd> oneAdmissibleState(double date) = 0;

/// \brief get back state size
virtual int getStateSize() const = 0;

/// \brief get the backward simulator back
virtual std::shared_ptr<StOpt::SimulatorSDDPBase> getSimulatorBackward() const = 0;

/// \brief get the forward simulator back
virtual std::shared_ptr<StOpt::SimulatorSDDPBase> getSimulatorForward() const = 0;
}
#endif /* OPTIMIZERSDDPBASE.H */

- A simulator for forward pass: SimulatorSim
- A simulator for backward pass: 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 realized test case a Boolean is enough to distinguish the forward and the backward simulator.

An abstract class for simulators is defined below.

// Copyright (C) 2016 EDF
// All Rights Reserved
// This code is published under the GNU Lesser General Public License (GNU LGPL)
#ifndef SIMULATORSDDPBASE_H
#define SIMULATORSDDPBASE_H
#include <Eigen/Dense>

/* \file SimulatorBase.h
* \brief Abstract class for simulators for SDDP method
* \author Xavier Warin
*/
namespace StOpt
{
  //\class SimulatorSDDPBase SimulatorSDDPBase.h
  //\abstract class for simulators used for SDDP
  class SimulatorSDDPBase
  {
  public:
    //\brief Get back the number of particles (used in regression part)
    virtual int getNbSimul() const = 0;
    //\brief Get back the number of sample used (simulation at each time step, these simulations are independent of the state)
    virtual int getNbSample() const = 0;
}
9.3.2 Set of parameters

We have seen above that providing a set of parameters in the initialization phase was necessary (convergence criterion, initial state vector, …). They depend on each problem and are based on technical skills. Suggestion to the intention of the user: provide a main where all these parameters are defined and where the sequence of computations to achieve is described. Basic class backwardForwardSDDP should be called.

9.3.3 The black box

The algorithms described above are applied. As said before the user controls the implementation of the business side of the problem (transition problem). But in the library a few things are managed automatically and the user has to be aware of:

- The Parallelization during the problem resolution is managed automatically. During compilation, if the compiler detects an MPI (Message Passing Interface) library problem resolution will be achieved in a parallelized manner.

- The cut management. All the cuts added at each iteration are currently serialized and stored in an archive initialized by the user. No cuts are pruned. In the future one can consider to work on cuts management [31].

- A double stopping criterion is barely used by the library: a convergence test and a maximal number of iterations. If one of the two criteria goes over the thresholds defined by the user resolution stops automatically. Once again further work could be considered on that topic.
9.3.4 Outputs

The outputs of the SDDP library are not currently clearly defined. Thus during the resolution of a SDDP problem only the number of iterations, the evolution of the backward and forward costs and of the convergence criterion are logged.

Yet while iterating backward and forward pass the value of the Bellman functions and the related Benders cuts, the different states visited during the forward pass and the costs evolution are stored at each time of the time horizon. These information are helpful for the users and easy to catch.

But the outputs should be suitable for the operational requirements. That is a point that should be adjusted after a first testing phase.
Bibliography


[26] R. Aid, Z.J. Ren, N. Touzi, Transition to non-emissive electricity production under optimal subsidy and endogeneous carbon price


