STochastic OPTimization library in C++
Hugo Gevret, Nicolas Langrené, Jerome Lelong, Xavier Warin, Aditya Maheshwari

To cite this version:
Hugo Gevret, Nicolas Langrené, Jerome Lelong, Xavier Warin, Aditya Maheshwari. STochastic OPTimization library in C++. [Research Report] EDF Lab. 2018. hal-01361291v7

HAL Id: hal-01361291
https://hal.archives-ouvertes.fr/hal-01361291v7
Submitted on 20 Feb 2019

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 \(^1\) Nicolas Langrené \(^2\) Jerome Lelong \(^3\)
Xavier Warin \(^4\) Aditya Maheshwari \(^5\)

\(^1\)EDF R&D, Hugo.Gevret@edf.fr
\(^2\)data61 CSIRO, locked bag 38004 docklands vic 8012 Australia, Nicolas.Langrene@data61.csiro.au
\(^3\)Ensimag, Laboratoire Jean Kuntzmann, 700 avenue Centrale Domaine Universitaire - 38401 St Martin d’Hres
\(^4\)EDF R&D & FiME, Laboratoire de Finance des Marchés de l’Energie, ANR PROJECT CAESARS, Xavier.Warin@edf.fr
\(^5\)University of California, Santa Barbara, USA, aditya_maheshwari@umail.ucsb.edu
## Contents

1 Introduction \hspace{1cm} 6
   1 General context \hspace{1cm} 7
   2 General mathematical setting \hspace{1cm} 10

II Useful tools for stochastic control \hspace{1cm} 12
   3 The grids and their interpolators \hspace{1cm} 13
      3.1 Linear grids \hspace{1cm} 18
         3.1.1 Definition and C++ API \hspace{1cm} 18
         3.1.2 The python API \hspace{1cm} 20
      3.2 Legendre grids \hspace{1cm} 22
         3.2.1 Approximation of a function in 1 dimension \hspace{1cm} 22
         3.2.2 Extension in dimension \(d\) \hspace{1cm} 26
         3.2.3 Truncature \hspace{1cm} 26
         3.2.4 The C++ API \hspace{1cm} 27
         3.2.5 The python API \hspace{1cm} 30
      3.3 Sparse grids \hspace{1cm} 31
      3.4 High order sparse grid methods \hspace{1cm} 34
      3.5 Anisotropy \hspace{1cm} 37
      3.6 Adaptation \hspace{1cm} 37
      3.7 C++ API \hspace{1cm} 39
      3.8 Python API \hspace{1cm} 45

4 Introducing the regression resolution \hspace{1cm} 49
   4.1 C++ global API \hspace{1cm} 50
   4.2 Adapted local polynomial basis \hspace{1cm} 56
      4.2.1 Description of the method \hspace{1cm} 57
   4.3 C++ api \hspace{1cm} 58
      4.3.1 The constant per cell approximation \hspace{1cm} 58
      4.3.2 The linear per cell approximation \hspace{1cm} 59
      4.3.3 An example in the linear case \hspace{1cm} 60
   4.4 Python API \hspace{1cm} 60
   4.5 Local polynomial basis with meshes of same size \hspace{1cm} 60
12.3.3 Implement your problem ........................................ 189
12.3.4 Set of parameters ............................................. 191
12.3.5 The black box .................................................. 192
12.3.6 Outputs .......................................................... 193
12.4 Python API .......................................................... 194

VII Nesting Monte carlo for general non linear PDEs 207

VIII Some test cases description 213

13 Some test cases description in C++ 214
13.1 American option .................................................. 214
  13.1.1 testAmerican ................................................ 214
  13.1.2 testAmericanConvex ...................................... 216
  13.1.3 testAmericanForSparse ................................... 217
  13.1.4 testAmericanOptionCorrel ................................ 217
  13.2 testSwingOption ............................................... 217
    13.2.1 testSwingOption2D ...................................... 218
    13.2.2 testSwingOption3 ....................................... 218
    13.2.3 testSwingOptimSimu / testSwingOptimSimuMpi ............ 219
    13.2.4 testSwingOptimSimuWithHedge .......................... 219
    13.2.5 testSwingOptimSimuND / testSwingOptimSimuNDMpi ...... 219
  13.3 Gas Storage ................................................... 220
    13.3.1 testGasStorage / testGasStorageMpi .................... 220
    13.3.2 testGasStorageCut / testGasStorageCutMpi .............. 221
    13.3.3 testGasStorageKernel ................................... 221
    13.3.4 testGasStorageVaryingCavity ............................ 221
    13.3.5 testGasStorageSwitchingCostMpi ......................... 222
    13.3.6 testGasStorageSDDP ...................................... 222
  13.4 testLake / testLakeMpi ..................................... 223
  13.5 testOptionNIGL2 ............................................. 223
  13.6 testDemandSDDP .............................................. 223
  13.7 Reservoir variations with SDDP ............................. 224
    13.7.1 testReservoirWithInflowsSDDP ......................... 224
    13.7.2 testStorageWithInflowsSDDP ........................... 225
    13.7.3 testStorageWithInflowsAndMarketSDDP .................. 226
  13.8 Semi-Lagrangian .............................................. 226
    13.8.1 testSemiLagrangCase1/testSemiLagrangCase1 .......... 226
    13.8.2 testSemiLagrangCase2/testSemiLagrangCase2 .......... 227
    13.8.3 testSemiLagrangCase2/testSemiLagrangCase2 .......... 227
  13.9 Non emimissive test case ................................... 228
Part I

Introduction
Chapter 1

General context

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

- Some options such as American options necessitate, in order to be valuated, to find an optimal exercise strategy to maximize the gain on average.

- When dealing with assets management, a fund manager may want to find a strategy to optimize his gains by investing in different assets while trying to satisfy some risk constraints.

- When dealing with credit risk in the case of option selling, some CVA modelization necessitates to solve some high dimensional problem in order to evaluate the option value.

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

- Some options, known as swing options, permit the owner to get some energy at some chosen dates with constraints on volumes. The price paid is either deterministic such as in the electricity market or can be an index which is an average of some commodity prices such as in the gas market.

- When some batteries are installed on a network, the battery has to be filled in or discharged optimally in order to avoid the use of some costly thermal units.

- The optimal management of some gas storages or some thermal assets taking into account commodities prices is a target shared by all asset owners in the sector.

- Even in regulated energy market, when some water is used to produce electricity, a common target consists in finding an optimal management of the water in order to maximize the profit on average.

A target shared by many industries is the risk management problem: which financial assets to buy to secure a given earning by immunizing a financial portfolio to some uncertainties. All this problems and many others necessitates:

- either to solve some PDEs when the control has to be evaluated continuously,
• or to calculate some conditional expectation in the case where the control has to be taken at some discrete dates. The problem is then solved by some dynamic programming method.

The STochastic OPTimization library (StOpt) [https://gitlab.com/stochastic-control/StOpt](https://gitlab.com/stochastic-control/StOpt) aims at providing tools for solving some stochastic optimization problems encountered in finance or in the industry. This library is a toolbox used to ease the work of developers wanting to solve some stochastic optimization problems by providing a general framework and some commonly used objects in stochastic programming. Many effective methods are implemented and the toolbox should be flexible enough to use the library at different levels either being an expert or only wanting to use the general framework.

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

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

• American options are solved by dynamic programming part III in python or C++ using regression (section 4).
  Regression are achieved:
  
  1. either by local polynomials either with basis support with the same size (subsection 4.3) or with an adapted size of the support (subsection 4.2) ,
  2. either by global polynomials (section 4.9)
  3. either by sparse grid regression (section 4.8) useful in high dimension
  4. or by kernel regression (section 4.10)

• Gas storage problems are solved
  – either by dynamic programming (part III) in python or C++ using regression (section 4) and stock interpolation (chapter 3). Regression are achieved :
    1. either by Local polynomials with an adapted size of the support (subsection 4.2) ,
    2. either by global polynomials (section 4.9)
    3. or kernel regression (section 4.10)
  Interpolation between stock points is either linear or quadratic.
  – either by the SDDP method (chapter 12) in C++.

• Swing options are solved by dynamic programming (part III) in python or C++ using regression with local polynomials with an adapted size of the support (subsection 4.2)

• The optimal management of a lake with stochastic inflows is solved by dynamic programming (part III) in python or C++ using local polynomials with an adapted size of the support (subsection 4.2)
- The optimal hedge of an option using a mean variance criterium of the hedged portfolio is solved in C++ by dynamic programming (part III) using the methodology in chapter 9.

- Some reservoir management is solved by the SDDP method (chapter 12) in C++ trying to minimize the cost of providing some energy to satisfy a given demand with the possibility to buy some energy at price that can be stochastic.

- The continuous optimization of a portfolio composed of some assets following an Heston model is achieved in C++ solving the corresponding PDE with the Monte Carlo nesting method (part VII).

- Some microgrid problems in the energy sector is solved using the python interface by dynamic programming methods (part III) using the grids with linear interpolation (subsection 3.1.1) to discretize the energy level in the battery and different regressors using:
  1. either local polynomials with an adapted size of the support (subsection 4.2),
  2. either global polynomials (section 4.9)
  3. or kernel regression (section 4.10)
Chapter 2

General mathematical setting

In a continuous setting, the controlled state is given by a stochastic differential equation
\[
\begin{align*}
    dX^{x,t}_s &= b_a(t, X^{x,t}_s)ds + \sigma_a(s, X^{x,t}_s)dW_s \\
    X^{x,t}_t &= x
\end{align*}
\]
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
\[
J(t, x, a) = \mathbb{E}\left[\int_t^T f_a(s, X^{x,t}_s)e^{\int_s^t c_a(u, X^{x,t}_u)du}ds + e^{\int_t^T c_a(u, X^{x,t}_u)g(X^{x,t}_T)}\right]
\]
with respect to the control \( a \). It is well known \([15]\) that the optimal value \( \hat{J}(t, x) = \inf_a J(T - t, x, a) \) is a viscosity solution of the equation
\[
\frac{\partial v}{\partial t}(t, x) - \inf_{a \in A} \left( \frac{1}{2} \text{tr}\left( \sigma_a(t, x)\sigma_a(t, x)^T D^2v(t, x) \right) + b_a(t, x)Dv(t, x) \\
+ c_a(t, x)v(t, x) + f_a(t, x) \right) = 0 \text{ in } \mathbb{R}^d
\]
\[v(0, x) = g(x) \text{ in } \mathbb{R}^d\] (2.1)
Under some classical assumptions on the coefficients \([15]\), the previous equation known as the Hamilton Jacobi Bellman equation admits an unique viscosity solution \([22]\).

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 \( X_{x,t} = (X_{1,s}^{x,t}, X_{2,s}^{x,t}) \) where \( X_{1,s}^{x,t} \) is not controlled

\[
\begin{align*}
\left\{ \begin{array}{l}
\frac{dX_{1,s}^{x,t}}{ds} = b(t, X_{1,s}^{x,t}) ds + \sigma(s, X_{1,s}^{x,t}) dW_s \\
X_{1,t}^{x,t} = x
\end{array} \right.
\]

\tag{2.2}
\]

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

\[
\begin{align*}
\left\{ \begin{array}{l}
\frac{dX_{2,s}^{x,t}}{ds} = b_a(t, X_{2,s}^{x,t}) ds \\
X_{2,t}^{x,t} = x
\end{array} \right.
\]

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 fourth method, we will suppose that the state takes continuous values, we will solve equation (2.1) using Semi Lagrangian methods discretizing the Brownian motion with two values and using some interpolations on grids.

• At last we present a pure Monte Carlo general method based on automatic differentiation and randomization of the time step to solve general non linear equations and that can be used to solve some control problems.

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 rely on the [Eigen library](http://eigen.tuxfamily.org/). “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 3

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)
#ifndef GRIDITERATOR_H
#define GRIDITERATOR_H
#include <Eigen/Dense>

/**
 \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
{

```

```
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” the 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 ( ) ) ; // c r e a t e an array to s t o r e the
values of the function $f$
shared_ptr < GridIterator > iterRegGrid = regGrid . getGridIterator ( ) ;
while ( iterRegGrid −> isValid ( ) )
{
    ArrayXd pointCoord = iterRegGrid −> getCoordinate ( ) ; // s t o r e 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>
/** \ile Interpolator.h
 * \brief Defines a interpolator on a full grid
 * \author Xavier Warin
 */
namespace StOpt
{
/// \class Interpolator Interpolator.h
/// Interpolation base class
class Interpolator
{
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

`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

`test/python/unit/grids`

### 3.1 Linear grids

#### 3.1.1 Definition and C++ API

Two kinds of grids are developed:

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

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

where `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 3.1).

![Figure 3.1: 2D general grid](image)

- the second one is the "RegularSpaceGrid" with constructor

```cpp
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 3.2).

For each grid, a linear interpolator can be generated by call to the `createInterpolator` method or by creating directly the interpolator:
Its construction from a grid \((\text{regLin})\) and an array \(\text{data}\) containing the values of the function at the grids points is given below (taking an example above to fill in the array \(\text{data}\)):

```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 \(\Delta x = (\Delta x^1, \ldots, \Delta x^d)\). We get for a function \(f\) in \(C^{k+1}(\mathbb{R}^d)\) with \(k \leq 1\)

\[
\|f - I_{1,\Delta x}f\|_\infty \leq c \sum_{i=1}^{d} \Delta x_i^{k+1} \sup_{x \in [-1,1]^d} |\partial^{k+1} f / \partial x_i^{k+1}| \tag{3.1}
\]

In particular if \(f\) is only Lipschitz

\[
\|f - I_{1,\Delta x}f\|_\infty \leq K \sup_i \Delta x_i.
\]
3.1.2 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
# 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

# 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(("Interpolated value", interpValue))
        # test grids function
        iDim = grid.getDimension()
        pt = grid.getExtremeValues()

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

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

```python
# Copyright (C) 2017 EDF
# All Rights Reserved
```
# This code is published under the GNU Lesser General Public License (GNU LGPL)

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

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

    # test general grids
def testGeneralGrids(self):
        # low value for the mesh
        lowValues = np.array([1., 2., 3.], dtype=np.float)
        # size of the mesh
        step = np.array([0.7, 2.3, 1.9], dtype=np.float)
        # number of step
        nbStep = np.array([4, 5, 6], dtype=np.int32)
        # degree of the polynomial in each direction
        degree = np.array([2, 1, 3], dtype=np.int32)

        # list of mesh
        mesh1 = np.array([1. + 0.7*i for i in np.arange(5)], dtype=np.float)
        mesh2 = np.array([2. + 2.3*i for i in np.arange(6)], dtype=np.float)
        mesh3 = np.array([3. + 1.9*i for i in np.arange(7)], dtype=np.float)

        # create the general grid
        grid = StOptGrids.GeneralSpaceGrid([mesh1, mesh2, mesh3]

        iterGrid = grid.getGridIterator()
        # array to store
        data = np.empty(grid.getNbPoints())
        # iterates on point
        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)
        interp = grid.createInterpolator(ptInterp)
        # calculate interpolated value
        interpValue = interp.apply(data)
        print("Interpolated value Legendre", interpValue)

    # test grids function
    iDim = grid.getDimension()
    pt = grid.getExtremeValues()

if __name__ == '__main__':
```

21
3.2 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.

3.2.1 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 $N \to \infty$, the Lebesgue constant satisfies
\[
\lambda_N(X_u) \simeq \frac{2^{N+1}}{e N \ln N}
\]
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 term 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 satisfies
\[
\rho_i = \frac{2}{(M+1)ML^2_M(\eta_i)}, \ 1 \leq i \leq N+1.
\]

More details can be found in [2]. In figure 3.4 we give the interpolation obtained with the Gauss Lobatto Legendre quadrature with two degrees of approximation.

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

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 \( w(f, \delta)\log(\delta) \leq K\log(\delta)\delta \).

- 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 [36])

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

\[
||f - I_{N,\Delta x}^X f||_\infty \leq ch^{k+1}||f^{(k+1)}||_\infty
\]
3.2.2 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 $I_N^X(f)(x) = I_{X_{N,1}}^X \times I_{X_{N,2}}^X \cdots \times I_{X_{N,d}}^X(f)(x)$ where $I_{X_{N,i}}^X$ 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 \[14\], Soardi \[39\]

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

We deduce that if $f$ is only Lipschitz

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

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

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

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

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

On figure 3.5 we give the Gauss Legendre Lobatto points in 2D for $2 \times 2$ meshes and a polynomial of degree 8 in each direction

3.2.3 Troncature

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

$$\hat{I}_{N,\Delta x}^X f(x) = \min_{x_i \in M} f(x_i) \land I_{N,\Delta x}^X f(x) \lor \max_{x_i \in M} f(x_i)$$ (3.3)

where the $x_i$ are the interpolation points on the mesh $M$ containing the point $x$. For all caracteristics of theis modified operator, one can see [44].
3.2.4 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 3.2). 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 3.4.

```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());
shared_ptr<GridIterator> iterRegGrid = regGrid.getGridIterator();
while (iterRegGrid->isValid()) {
    ArrayXd pointCoord = iterRegGrid->getCoordinate();
    data(iterRegGrid->getCount()) = 1. / (1. + 25*pointCoord(0)*pointCoord(0));  
    // store runge function
    iterRegGrid->next();
```
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 case 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:
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:

LegendreInterpolatorSpectral(const shared_ptr< RegularLegendreGrid> & p_grid , const Eigen::ArrayXd &p_values) ;

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

/** \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.

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
3.2.5 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

# unit test for Legendre grids
# # # # # # # # # # # # # # # # # # # # # # # # # # # # #

class testGrids(unittest.TestCase):

    # test Legendre grids
def testLegendreGrids(self):
        # low value for the mesh
        lowValues = np.array([1.0, 2.0, 3.0], dtype=np.float)
        # size of the mesh
        step = np.array([0.7, 2.3, 1.9], dtype=np.float)
        # number of step
        nbStep = np.array([4, 5, 6], dtype=np.int32)
        # degree of the polynomial in each direction
        degree = np.array([2, 1, 3], dtype=np.int32)

        # create the Legendre grid
        grid = StOptGrids.RegularLegendreGrid(lowValues, step, nbStep, degree)
        iterGrid = grid.getGridIterator()

        # array to store
        data = np.empty(grid.getNbPoints())

        # iterates on point
        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(("Interpolated value Legendre", interpValue))

        # test grids function
        iDim = grid.getDimension()
        pt = grid.getExtremeValues()
```
3.3 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 dimension of the problem. In many application this assumption is not realistic or it is impossible to work on $f - f|_{\Gamma}$. In this library we will 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.

3.3.1 The linear sparse grid method

We recall some classical results on sparse grids that can be found in [35]. 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

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

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

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

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

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

and the space of hierarchical basis

$$W^{(L)}_{\bar{i}} := \text{span} \left\{ \phi^{(L)}_{\bar{i}}(x) : \bar{i} \in B_{\bar{i}} \right\}$$

A representation of the space $W^{(L)}_{\bar{i}}$ is given in dimension 1 on figure 3.6. The sparse grid
Figure 3.6: One dimensional $W^{(L)}$ spaces: $W^{(L)}_1$, $W^{(L)}_2$, $W^{(L)}_3$, $W^{(L)}_4$ and the nodal representation $W^{(L,N)}_4$

space is defined as:

$$V_n = \bigoplus_{|l|_1 \leq n + d - 1} W^{(L)}_l$$

(3.4)

**Remark 4** The conventional full grid space is defined as $V^F_n = \bigoplus_{|l|_\infty \leq n} W^{(L)}_l$

At a space of hierarchical increments $W^{(L)}_l$ corresponds a space of nodal function $W^{(L,N)}_l$ such that

$$W^{(L,N)}_l := \text{span} \left\{ \phi^{(L)}_l(\xi) : \mathbf{i} \in B^N_l \right\}$$

with

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

On figure 3.6 the one dimensional nodal base $W^{(L,N)}_4$ is spawned by $W^{(L)}_4$ and the dotted basis function. The space $V_n$ can be represented as the space spawn by the $W^{(L,N)}_l$ such that $|l|_1 = n + d - 1$:

$$V_n = \text{span} \left\{ \phi^{(L)}_l(\xi) : \mathbf{i} \in B^N_l, |l|_1 = n + d - 1 \right\}$$

(3.5)

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

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

where $\alpha^{(L)}_l$ are called the surplus (we give on figure 3.7 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 3.8 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 3.9, 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 first level as shown on figure 3.10. This approach results in many more points than the one without the boundary. As noted in [35] for \( n = 5 \), in dimension 8 you have nearly 2.8 millions points in this approximation but only 6401 inside the domain. On figure 3.11 we give the grids points with boundary points in dimension 2 and 3 for a level 5 of the sparse grid.

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 3.10) as explained in [35]. 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

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

On figure 3.12 we give the grids points eliminating boundary points in dimension 2 and 3 for a level 5 of the sparse grid.

The interpolation error associated to the linear operator \( I_1 := I^{(L)} \) is linked to the regularity of the cross derivatives of the function [9 10 11]. If \( f \) is null at the boundary and admits derivatives such that \( \|\frac{\partial^{2d} f}{\partial x_1 \ldots \partial x_2}\|_{\infty} < \infty \) then

\[
\|f - I_1(f)\|_{\infty} = O(N^{-2 \log(N)^{d-1}}),
\] (3.6)

with \( N \) the number of points per dimension.
3.4 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 \cite{10} and \cite{11}, 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 3.8

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

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

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

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 3.13).

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

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

In the case of the cubic representation, on figure 3.8 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 $e(m)$ [10].

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

$$\phi^{(C)}_{l,2i+1}(x) = \phi^{(C),1}(2^l x - (2i + 1)), \text{if } i \text{ even}$$

$$\phi^{(C)}_{l,2i+2}(x) = \phi^{(C),2}(2^l x - (2i + 1)), \text{if } i \text{ odd}$$

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 3.14:

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

According to [9] [10] [11], if the function $f$ is null at the boundary and admits derivatives such that $\sup_{\alpha \in \{2, \ldots, p+1\}} \left\{ \left\| \frac{\partial^{\alpha_1+\ldots+\alpha_l+m}}{\partial x_1^{\alpha_1}\ldots\partial x_l^{\alpha_l}} f \right\|_\infty \right\} < \infty$ then the interpolation error can be generalized for $I^2 := I^{(Q)}, I^3 := I^{(C)}$ by:

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

with $N$ the number of points per dimension.
3.5 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 3.4 is replaced by:

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

(3.7)

3.6 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 \( \alpha_{L} \) 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 \([12, 19, 20, 27]\),
- 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 \([16]\). This refinement is also achieved in areas where the solution can be smooth. A more local version has been developed in \([24]\).

In the current version of the library only dimension adaptation is available. Details on the algorithm can be found in \([16]\). After a first initialization with a first initialization with a
space

\[ V_n = \bigoplus_{\sum_{i=1}^{d} l_i \leq n+d-1} W^{(L)}_{l} \]  

(3.8)

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 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,

**Algorithm 1** Dimension refinement for a given tolerance \( \eta \)

1: \textbf{while} \( E > \eta \) \textbf{do}
2: \quad select \( l \) with the highest local error \( e_l \)
3: \quad \( A = A \setminus \{l\} \)
4: \quad \( O = O \cup \{l\} \)
5: \quad \textbf{for} \( k = 1 \) to \( d \) \textbf{do}
6: \quad \quad \( m = l + e_k \)
7: \quad \quad \textbf{if} \( m - e_q \in O \) for \( q \in [1,d] \) \textbf{then}
8: \quad \quad \quad \( A = A \cup \{m\} \)
9: \quad \quad \quad \text{Hierarchize all points belonging to} \: m
10: \quad \quad \text{calculate} \: e_m
11: \quad \quad \text{update} \: E
12: \quad \textbf{end if}
13: \textbf{end for}
14: \textbf{end while}

it can be interesting to coarsen the meshes. A similar algorithm 2 can be used to eliminate levels with a very small local error.
Algorithm 2 Dimension coarsening for a given tolerance $\eta$

$\mathcal{B}$ 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

3.7 C++ API

The construction of the Sparse Grid including boundary point is done by the following constructor

```cpp
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

- $p_{lowValues}$ corresponds to the bottom of the grid,
- $p_{sizeDomain}$ corresponds to the size of the resolution domain in each dimension,
- $p_{levelMax}$ is the level of the sparse grids, the $n$ in equation 3.7,
- $p_{weight}$ the weight for anisotropic sparse grids, the $w$ in equation 3.7,
- $p_{degree}$ is equal to 1 (linear interpolator), or 2 (quadratic interpolator) or 3 (for cubic interpolator),

With the same notations the construction eliminating boundary points is done by the following constructor
SparseSpaceGridNoBound(const Eigen::ArrayXd &p_lowValues, const Eigen::ArrayXd &p_sizeDomain, const int &p_levelMax, const Eigen::ArrayXd &p_weight, const size_t &p_degree)

The data structure of type SparseSet to store the sparse grid is defined by a map with keys an array A storing a 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):

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

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

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:

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

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

/// \brief Hierarchize a set of functions defined on the grid
/// \param p_toHierarchize set of functions to hierarchize
void toHierarchizeVec( Eigen::ArrayXXd & p_toHierarchize );

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:

/// \brief Hierarchize some points defined on the sparse grids
/// \brief Hierarchization is performed point by point
/// \param p_nodalValues function to hierarchize
/// \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)
virtual void toHierarchizePByP(const Eigen::ArrayXd &p_nodalValues, const std::vector<SparsePoint> &p_sparsePoints, Eigen::ArrayXd &p_hierarchized) const

- the second work on a matrix, 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 toHierarchizePByPLLevel(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 toHierarchizePByPLLevelVec(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 \( \text{valuesFunction} \), storing 2 functions to interpolate. The 2 global functions are hierarchized (see section above) in the array \( \text{hierarValues} \), 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.getGridIterator();
ArrayXXd valuesFunction(1, sparseGrid.getNbPoints());
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;
ArrayXXd hieraValues = valuesFunction;
sparseGrid.toHierarchizeVec(hieraValues);

// interpolate
ArrayXd pointCoord = ArrayXd::Constant(5,0.66);
shared_ptr<Interpolator> interpolator = sparseGrid.createInterpolator(pointCoord);
ArrayXd interVal = interpolator->applyVec(hieraValues);
```

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

```cpp
std::vector<SparsePoint> sparsePoints(sparseGrid.getNbPoints());
std::shared_ptr<SparseSet> dataSet = sparseGrid.getDataSet();

// 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
apply(const Eigen::ArrayXd &point) const
```

See section 3.2 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
SparseSpaceGridNoBound sparseGrid(lowValues, sizeDomain, p_level, p_weight, p_degree, bPrepInterp);

std::shared_ptr<SparseSet> dataSet = sparseGrid.getDataSet();
for (SparseSet::const_iterator iterLevel = dataSet->begin(); iterLevel != dataSet->end(); ++iterLevel)
{
    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.

### 3.8 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
```

45
hierarData = sparseGridLin.toHierarchize(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("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)
interpol = sparseGridQuad.createInterpolator(ptInterp)
# calculate interpolated value
interpValue = interpol.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
interpol1 = sparseGridQuad.createInterpolator(ptInterp)
interpValue = interpol1.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
interpol2 = sparseGridQuad.createInterpolator(ptInterp)
interpValue = interpol2.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.SpaseSpaceGridNoBound(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
hierarData = sparseGridLin.toHierarchize(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(("Interpolated value sparse linear", interpValue))

# create the sparse grid with quadratic interpolator
sparseGridQuad = StOptGrids.SpaseSpaceGridNoBound(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)
interpol = sparseGridQuad.createInterpolator(ptInterp)

# calculate interpolated value
interpValue = interpol.apply(hierarData)

print(("Interpolated 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(("Interpolated 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(("Interpolated value sparse quadratic after coarsening"))
interpValue))

if __name__ == '__main__':
    unittest.main()
Chapter 4

Introducing the regression resolution

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

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

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 of 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 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

\[ f(x) \simeq \sum_{k=1}^{M} \alpha_k \psi_k(X_t) \tag{4.1} \]

where \( (\alpha_k^t_{i,N})_{k \leq M} \) minimizes

\[ \sum_{l=1}^{N} \left| g(X_{t+h}^{l}) - \sum_{k=1}^{M} \alpha_k \psi_k(X_t^{l}) \right|^2 \tag{4.2} \]

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

\[ \min_{\alpha \in \mathbb{R}^M} \|A\alpha - B\|^2 \tag{4.3} \]

Classically the previous equation is reduced to the normal equation

\[ A' A \alpha = A' B \tag{4.4} \]

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 using the pseudo inverse of \( A' A \). When the different component of \( X^{x,t} \) are highly correlated it can be convinient to rotate
the data set onto its principal components using the PCA method. Rotating the dataset before doing regression has been advocated in [41] and [38] for example. The right-hand side of Figure 4.1 illustrates the new evaluation grid obtained on the same dataset. One can observe the better coverage and the fewer empty areas when using local regression that we will detail in this section.

![Figure 4.1: Evaluation grid: rotation](image)

4.1 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
#include <memory>
#include <vector>
#include <iostream>
#include <Eigen/Dense>
#include <Eigen/SVD>
#include "StOpt/core/grids/InterpolatorSpectral.h"

namespace StOpt {

```
/// \class BaseRegression BaseRegression.h
/// Base class for regression

class BaseRegression
{
    protected :
    
    bool m_bZeroDate ; ///< Is the regression date zero ?
    bool m_bRotationAndRescale ; ///< do we rescale particles and do a
                                    rotation with SVD on data
    Eigen::ArrayXd m_meanX ; ///< store scaled factor in each direction ( 
                              average of particles values in each direction)
    Eigen::ArrayXd m_etypX ; ///< store scaled factor in each direction ( 
                              standard deviation of particles in each direction)
    Eigen::MatrixXd m_svdMatrix ; ///< svd matrix transposed used to 
                                   transform particles
    Eigen::ArrayXd m_sing  ; ///< singular values associated to SVD
    Eigen::MatrixXd m_particles ; /////< Particles used to regress: first 
                                   dimension : dimension of the problem , second dimension : the number 
                                   of particles. These particles are rescaled and a rotation with SVD is 
                                   achieved to avoid degeneracy in case of high correlations

    // rotation for data and rescaling
    void preProcessData();

    public :

    /// \brief Default constructor
    BaseRegression();

    /// \brief Default destructor
    virtual ~BaseRegression() {} 

    /// \brief Default constructor
    BaseRegression(const bool &p_bRotationAndRescale);

    /// \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 
    /// second dimension : the number of 
    /// particles
    /// \param p_bRotationAndRescale do we rescale particle 
    /// Data are rescaled and a
    BaseRegression(const bool &p_bZeroDate, const std::shared_ptr<Eigen::ArrayXXd> &p_particles, const bool &p_bRotationAndRescale);

    /// \brief Constructor used in simulation, no rotation
    /// \param p_bZeroDate first date is 0?
    /// \param p_bRotationAndRescale do we rescale particle 
    BaseRegression(const bool &p_bZeroDate, const bool &p_bRotationAndRescale );
BaseRegression(const bool &p_bZeroDate, const Eigen::ArrayXd &p_meanX,
               const Eigen::ArrayXd &p_etypX, const Eigen::MatrixXd &p_svdMatrix,
               const bool &p_bRotationAndRescale);

/// \brief Copy constructor
/// \param p_object object to copy
BaseRegression(const BaseRegression &p_object);

/// \brief update the particles used in regression and construct the matrices
/// \param p_bZeroDate first date is 0?
/// \param p_particles particles used for the meshes.
/// Firs dimension : dimension of the problem,
/// second dimension : the number of particles
void updateSimulationsBase(const bool &p_bZeroDate, const std::shared_ptr<Eigen::ArrayXXd> &p_particles);

/// \brief Get some local accessors
///@{
virtual inline Eigen::ArrayXXd getParticles() const
{
    return m_particles ;
}

/// \brief Get bRotationAndRescale
virtual inline bool getBRotationAndRescale() const
{
    return m_bRotationAndRescale ;
}

/// \brief Get average of simulation per dimension
virtual inline Eigen::ArrayXd getMeanX() const
{
    return m_meanX ;
}

/// \brief get standard deviation per dimension
virtual inline Eigen::ArrayXd getEtypX() const
{
    return m_etypX ;
}

/// \brief get back the SVD matrix used for rescaling particles
virtual inline Eigen::MatrixXd getSvdMatrix() const
{   return m_svdMatrix;
}

/// \brief get back singular values
defined inline Eigen::ArrayXd getSing() const
{
   return m_sing;
}

/// \brief Get dimension of the problem
defined inline int getDimension() const
{
   return m_particles.rows();
}

/// \brief Get the number of simulations
defined inline int getNbSimul() const
{
   return m_particles.cols();
}

/// \brief get back particle by its number
/// \param p_iPart particle number
/// \return the particle (if no particle, send back an empty array)
defined Eigen::ArrayXd getParticle(const int &p_iPart) const;

/// \brief get the number of basis functions
defined int getNumberOfFunction() const = 0;

/// @}
/// \brief Constructor storing the particles
/// \brief update the particles used in regression and construct the matrices
/// \param p_bZeroDate first date is 0?
/// \param p_particles particles used for the meshes.
/// \param p_particles First dimension: dimension of the problem,
/// second dimension: the number of particles
defined void updateSimulations(const bool &p_bZeroDate, const std::shared_ptr<Eigen::ArrayXXd> &p_particles) = 0;

/// \brief conditional expectation basis function coefficient calculation
/// \param p_fToRegress function to regress associated to each simulation used in optimization
/// \return regression coordinates on the basis (size: number of meshes multiplied by the dimension plus one)
/// @
defined Eigen::ArrayXd getCoordBasisFunction(const Eigen::ArrayXd &p_fToRegress) const = 0;

/// \brief conditional expectation basis function coefficient calculation
/// for multiple functions to regress
/// \param p_fToRegress function to regress associated to each simulation used in optimization (size: number of functions to regress
(times the number of Monte Carlo simulations)

/// return regression coordinates on the basis (size: number of function to regress \times number of meshes multiplied by the dimension plus one)

/// @} virtual Eigen::ArrayXXd getCoordBasisFunctionMultiple(const Eigen::ArrayXXd &p_fToRegress) const = 0;
/// @} returning regression coordinates on the basis (size: number of meshes multiplied by the dimension plus one)

virtual Eigen::ArrayXd getAllSimulations(const Eigen::ArrayXd &p_fToRegress) const = 0;
virtual Eigen::ArrayXXd getAllSimulationsMultiple(const Eigen::ArrayXXd &p_fToRegress) const = 0;
/// @} brief conditional expectation calculation
/// \param p_fToRegress simulations to regress used in optimization
/// \return regressed value function

virtual Eigen::ArrayXd reconstruction(const Eigen::ArrayXd & p_basisCoefficients) const = 0;
virtual Eigen::ArrayXXd reconstructionMultiple(const Eigen::ArrayXXd & p_basisCoefficients) const = 0;
/// @} brief Use basis functions to reconstruct the solution
/// \param p_basisCoefficients basis coefficients

virtual double reconstructionASim(const int &p_isim, const Eigen::ArrayXd &p_basisCoefficients) const = 0;
/// @} brief use basis function to reconstruct a given simulation
/// \param p_isim simulation number
/// \param p_basisCoefficients basis coefficients to reconstruct a given conditional expectation

double getValue(const Eigen::ArrayXd & p_coordinates,
                const Eigen::ArrayXd &p_coordBasisFunction) const = 0;
/// \brief conditional expectation reconstruction
/// \param p_coordinates coordinates to interpolate (uncertainty sample)

/// \param p_coordBasisFunction regression coordinates on the basis (size: number of meshes multiplied by the dimension plus one)
/// \return regressed value function reconstructed for each simulation

double getValues(const Eigen::ArrayXd & p_coordinates,
                 const Eigen::ArrayXd &p_coordBasisFunction) const = 0;
/// \brief conditional expectation reconstruction for a lot of simulations
/// \param p_coordinates coordinates to interpolate (uncertainty sample) size uncertainty dimension by number of samples
/// \param p_coordBasisFunction regression coordinates on the basis (size: number of meshes multiplied by the dimension plus one)
/// \return regressed value function reconstructed for each simulation
const
{
    Eigen::ArrayXd valRet(p_coordinates.cols());
    for (int is = 0; is < p_coordinates.cols(); ++is)
        valRet(is) = getValue(p_coordinates.col(is), p_coordBasisFunction);
    return valRet;
}

/// \rief permits to reconstruct a function with basis functions coefficients values given on a grid
/// \param p_coordinates coordinates (uncertainty sample)
/// \param p_ptOfStock grid point
/// \param p_interpFuncBasis spectral interpolator to interpolate the basis functions coefficients used in regression on the grid (given for each basis function)
virtual double getAValue(const Eigen::ArrayXd &p_coordinates, const Eigen::ArrayXd &p_ptOfStock, const std::vector<std::shared_ptr<InterpolatorSpectral>> &p_interpFuncBasis) const = 0;

/// \brief is the regression date zero
inline bool getBZeroDate() const
{
    return m_bZeroDate;
}

/// \brief Clone the regressor
virtual std::shared_ptr<BaseRegression> clone() const = 0;

}

#endif

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 (4.1).

• “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 \( isim \) (optimization part) and \( \alpha_k \) coefficient of the basis functions as input and returns \( f(X_t^{isim}) \) by applying equation (4.1),

• “getValue” takes as first argument a sample of \( X_t \), the basis function \( \alpha_k \) and reconstruct the regressed solution of equation (4.1).

• “getValues” takes as first argument some samples of \( X_t \) (array size dimension of uncertainty by number of samples), the basis function \( \alpha_k \) and reconstruct the regressed solution of equation (4.1) (an array).

### 4.2 Adapted local polynomial basis

The description of the method and its properties can be found in [8]. We just recall the methodology. These local adapted methods can benefit from a rotation in the its principal axis using the PCA method. The rotation is activated by a flag in the constructor of the objects?
4.2.1 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 \( \big[\min_{k=1,N} X_{t_i}^{1,(k)}, \max_{k=1,N} X_{t_i}^{1,(k)}\big] \times \cdots \times \big[\min_{k=1,N} X_{t_i}^{d,(k)}, \max_{k=1,N} X_{t_i}^{d,(k)}\big] \).

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 \( M_M \times (1 + d) \).

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 4.2, we have plotted an example of supports in the case of \( 6 = 4 \times 4 \) local basis cells, in dimension 2.

Sometimes we can do further exploiting knowledge on the continuation value. In the case of an american basket option for example we have convexity of this continuation value with respect to the underlying prices. It is possible to modify the previous algorithm to try to...
impose that the numerical method respects this convexity. The algorithm in [28] has been implemented as an option. This algorithm may not converge when used in multidimension but it permits to improve the convexity of the solution while iterating a few times.

4.3 C++ api

4.3.1 The constant per cell approximation

The constructor of the local constant regression object is achieved by

```cpp
LocalConstRegression(const Eigen::ArrayXi &p_nbMesh, bool p_bRotationAndReccale = false);
```

where:

- `p_nbMesh` is an array giving the number of meshes used in each direction (4, 4) for the figure 4.2 for example).

- `p_bRotationAndReccale` is an optimal argument by default set to False meaning that no rotation of the data in its principal components axis is achieved. In the case of rotation, the direction are sorted with their singular values decreasing and the number of meshes in `p_nbMesh` are defined for these sorted directions: `p_nbMesh(0)` is associated with first direction with the highest singular value, `p_nbMesh(1)` with the direction associated to the second highest singular value etc..

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,
                     bool p_bRotationAndReccale = false)
```

where

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

- `p_particles` the particles `X_t` for all simulations (dimension of `X_t` for first dimension, number of Monte Carlo simulations in second dimension),

- `p_nbMesh` is an array giving the number of meshes used in each directions (4, 4) for the figure 4.2,

- `p_bRotationAndReccale` is an optimal argument by default set to False meaning that no rotation of the data in its principal components axis is achieved. In the case of rotation, the direction are sorted with their singular values decreasing and the number of meshes in `p_nbMesh` are defined for these sorted directions: `p_nbMesh(0)` is associated with first direction with the highest singular value, `p_nbMesh(1)` with the direction associated to the second highest singular value etc...
4.3.2 The linear per cell approximation

The constructor of the local linear regression object is achieved by

\[
\text{LocalLinearRegression( const Eigen::ArrayXi &p_nbMesh, bool p_bRotationAndRecale = false );}
\]

where

- \( p_{nbMesh} \) is an array giving the number of meshes used in each direction (\( (4,4) \) for the figure 4.2 for example),

- \( p_{bRotationAndRecale} \) is an optimal argument by default set to False meaning that no rotation of the data in its principal components axis is achieved. In the case of rotation, the direction are sorted with their singular values decreasing and the number of meshes in \( p_{nbMesh} \) are defined for these sorted directions : \( p_{nbMesh}(0) \) is associated with first direction with the highest singular value, \( p_{nbMesh}(1) \) with the direction associated to the second highest singular value etc...

The second constructor permits the construct the regression matrix,

\[
\text{LocalLinearRegression( const bool &p_bZeroDate,  
const shared_ptr<ArrayXXd> &p_particles,  
const Eigen::ArrayXi &p_nbMesh,  
bool p_bRotationAndRecale = false )}
\]

where

- \( p_{bZeroDate} \) is true if the regression date is 0,

- \( p_{particles} \) the particles \( X_t \) for all simulations (dimension of \( X_t \) for first dimension, number of Monte Carlo simulations in second dimension),

- \( p_{nbMesh} \) is an array giving the number of meshes used in each directions (\( 4,4 \)) for the figure 4.2

- \( p_{bRotationAndRecale} \) is an optimal argument by default set to False meaning that no rotation of the data in its principal components axis is achieved. In the case of rotation, the direction are sorted with their singular values decreasing and the number of meshes in \( p_{nbMesh} \) are defined for these sorted directions : \( p_{nbMesh}(0) \) is associated with first direction with the highest singular value, \( p_{nbMesh}(1) \) with the direction associated to the second highest singular value etc...

This class can benefit of the methodology in [28] implementing a generalization of the member function ‘getAllSimulations’ :

\[
\text{Eigen::ArrayXd getAllSimulationsConvex( const Eigen::ArrayXd &p_fToRegress,  
const int &p_nbIterMax )}
\]

where

- \( p_{fToRegress} \) is the set of points we want to regress preserving convexity of the regressed function value

- \( p_{nbIterMax} \) is the maximal number of iteration of the method.

It returns the regressed values for all simulations of the uncertainties.
4.3.3 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, no rotation of the data
LocalLinearRegression localRegressor(nbMesh);

// update particles values
localRegressor.updateSimulations(bZeroDate, x);

// regressed values
ArrayXd regressedValues = localRegressor.getAllSimulations(toRegress);
```

4.4 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, :])*(1 + x[0, :])

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

# mesh
nbMesh = np.array([6], dtype=np.int32)

# Regressor without rotation of data
regressor = StOptReg.LocalLinearRegression(False, x, nbMesh)
y = regressor.getAllSimulations(toRegress).transpose()[0]
```

Of course the constant per cell case in python is similar. As in C++ the linear case permits to try to regress preserving convexity by using the `getAllSimulationsConvex` method.

4.5 Local polynomial basis with meshes of same size

In some cases, instead of using adapted meshes, on can prefer to fix the mesh with a constant step in each direction with \( I_k \) meshes in each direction so that the total number of cells is \( M_M = \prod_{k=1}^{d} I_k \). On each cell as in section 4.2, one can have two approximations:

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

Because we define in each direction, the domain for the local basis, we don’t use any rotation of the data.
4.6 C++ api

4.6.1 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,

```cpp
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

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

4.6.2 The linear per cell approximation

The constructor of the local linear regression object is achieved by

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

where

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

The second constructor permits the construct the regression matrix,
LocalSameSizeLinearRegression(const bool &p_bZeroDate,
const std::shared_ptr<Eigen::ArrayXd> &p_particles,
const Eigen::ArrayXd &p_lowValues,
const Eigen::ArrayXd &p_step,
const Eigen::ArrayXi &p_nbStep)

where

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

### 4.6.3 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 \( p\_nDim \):

```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);
```

### 4.7 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.,1.,size=(1,nbSimul));
# real function
toReal = (2+x[0,:]+(x[0,:])*(1+x[0,:]))
```
# function to regress
toRegress = toReal + 4*np.random.normal(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.

## 4.8 Sparse grid regressor

In the case of a sparse regressor, the grid is an object “SparseSpaceGridNoBound” (extrap-
olation for the boundary conditions). The basis functions are given by the section 3.3 for
linear, quadratic or cubic function basis. No rotation of the data is available.

### 4.8.1 C++ API

Two specific constructor are available:

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

  ```
  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 (3.4),
  - `p_weight` the weight for anisotropic sparse grids (see equation (3.7),
  - `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 4.2)

- 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):

  ```
  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”
// 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
toRegress);
// 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);
}
// get back all values once for all
ArrayXd regressedAllValues = localRegressor.getValues(*x,
        regressedFuntionCoeff);

4.8.2 Python API

Here is a simple example of the python API:

import StOptReg
nbSimul = 2000000;
np.random.seed(000)
x = np.random.uniform(-1.,1.,size=(1,nbSimul));
# real function
toReal = (2+x[0,:] + (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)
# get back basis function
regressedFuntionCoeff= regressor.getCoordBasisFunction(toRegress)
# get back all values
ySecond= regressor.getValues(x,regressedFuntionCoeff)
4.9 Global polynomial basis

4.9.1 Description of the method

In this section, the $\psi_k(X_t)$ involved in equation 4.1 are some given polynomials. Available polynomials are the canonical one, the Hermite and the Chebyshev ones.

- Hermite polynomials $H_n(x) = (-1)^n e^{\frac{x^2}{2}} \frac{d^n}{dx^n} e^{-\frac{x^2}{2}}$ are orthogonal with respect to the weight $w(x) = e^{-\frac{x^2}{2}}$ and we get
  \[
  \int_{-\infty}^{+\infty} H_m(x)H_n(x)dx = \delta_{mn}\sqrt{2\pi n!}
  \]
  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
  \[
  a_{n+1,k} = a_{n,k-1} - na_{n-1,k}, k > 0
  \]
  \[
  a_{n+1,0} = -na_{n-1,0}
  \]

- 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
  \[
  \int_{-1}^{1} T_N(x)T_M(x)w(x)dx = \begin{cases} 
  0, & \text{if } M \neq N \\
  \pi, & \text{if } M = N = 0 \\
  \frac{\pi}{2}, & \text{if } M = N \neq 0
  \end{cases}
  \]
  They satisfy the following recurrence:
  \[
  T_{N+2}(x) = 2xT_{N+1}(x) - T_N(x)
  \]

As an option rotation of the data is possible even if the advantage of the rotation seem to be limited for global polynomials.

4.9.2 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, bool p_bRotationAndRecale = false);
```

where $p\_degree$ is the total degree of the polynomial approximation, $p\_dim$ is the dimension of the problem, $p\_bRotationAndRecale$ is an optional flag set to true if rotation of the data should be achieved (default is no rotation). A second constructor is provided:
GlobalRegression(const bool &p_bZeroDate,
    const std::shared_ptr<Eigen::ArrayXXd> &p_particles,
    const int &p_degree, bool p_bRotationAndRecale = false)

where

- \( p_{bZeroDate} \) is true if the regression date is 0,
- \( p_{particles} \) the particles \( X_t \) for all simulations (dimension of \( X_t \) for first dimension, number of Monte Carlo simulations in second dimension),
- \( p_{degree} \) is the total degree of the polynomial approximation,
- \( p_{bRotationAndRecale} \) is an optional flag set to true if rotation of the data should be achieved (default is no rotation)

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

```cpp
  // total degree equal to 2
  int degree = 2;
  // t is not zero
  bool bZeroDate = 0;
  // constructor with Hermite polynomials, no rotation
  GlobalRegression<Hermit> localRegressor(degree, x.rows());
  // update particles values
  localRegressor.updateSimulations(bZeroDate, x);
  // regressed values
  ArrayXd regressedValues = localRegressor.getAllSimulations(toRegress);
```

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

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

or by a Chebyshev one :

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

### 4.9.3 Python API

Here is a similar example using the second constructor

```python
import StOptReg
nbSimul = 5000000;
np.random.seed(1000)
x = np.random.uniform(-1, 1, size=(1, nbSimul));
# real function
toReal = (2+x[0,:])+(x[0,:]+(1+x[0,:])*np.random.normal(0, nbSimul))
# function to regress
toRegressor = toReal + 4*np.random.normal(0, nbSimul)
# degree
degree  = 2
# Regressor, no rotation
regressor = StOptReg.GlobalHermiteRegression(False, x, degree)
y = regressor.getAllSimulations(toRegressor).transpose()[0]
```
Available regressors are “GlobalHermiteRegression” as in the example above, “GlobalCanonicalRegression” and “GlobalTchebychevRegression” with an obvious correspondence.

4.10 Kernel regression

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

\[
\hat{f}_{\text{KDE}}(z) := \frac{1}{N} \sum_{i=1}^{N} K_h(x_i - z) \tag{4.7}
\]

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

\[
\hat{f}_{\text{NW}}(z) := \frac{\sum_{i=1}^{N} K_h(x_i - z)y_i}{\sum_{i=1}^{N} K_h(x_i - z)} \tag{4.8}
\]

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

\[
\hat{f}_{\text{L}}(z) := \min_{\alpha(z), \beta(z)} \sum_{i=1}^{N} K_h(x_i - z) [y_i - \alpha(z) - \beta(z)x_i]^2 \tag{4.9}
\]

The well known computational problem with the implementation of the kernel smoothers (4.7)-(4.8)-(4.9) is that their direct evaluation on a set of \(M\) evaluation points would require \(O(M \times N)\) operations. In particular, when the evaluation points coincide with the input points \(x_1, x_2, \ldots, x_N\), a direct evaluation requires a quadratic \(O(N^2)\) number of operations. In StOpt we develop the methodology described in [25] permitting to get a \(N \log N\) cost function.

4.10.1 The univariate case

In one dimension, StOpt uses the one dimensional Epanechnikov kernel

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

and the fast summing algorithm is used: Let \((x_1, y_1), (x_2, y_2), \ldots, (x_N, y_N)\) be a sample of \(N\) input (source) points \(x_i\) and output points \(y_i\), and let \(z_1, z_2, \ldots, z_M\) be a set of \(M\) evaluation (target) points. Without loss of generality, we assume that the input points and evaluation points are sorted: \(x_1 \leq x_2 \leq \ldots \leq x_N\) and \(z_1 \leq z_2 \leq \ldots \leq z_M\). In order to compute the kernel density estimator (4.7), the kernel regression (4.8) and the locally linear regression (4.9) for every evaluation point \(z_j\), one needs to compute sums of the type

\[
S_j = S_j^{p,q} := \frac{1}{N} \sum_{i=1}^{N} K_h(x_i - z_j)x_i^py_i^q = \frac{1}{Nh} \sum_{i=1}^{N} K\left( \frac{x_i - z_j}{h} \right)x_i^py_i^q, p = 0, 1, q = 0, 1 \tag{4.10}
\]

67
for every \( j \in \{1, 2, \ldots, M\} \). The direct, independent evaluation of these sums would require \( \mathcal{O}(N \times M) \) operations (a sum of \( N \) terms for each \( j \in \{1, 2, \ldots, M\} \)). The idea of fast sum updating is to use the information from the sum \( S_j \) to compute the next sum \( S_{j+1} \) without going through all the \( N \) input points again. Using the Epanechnikov (parabolic) kernel \( K(u) = \frac{3}{4}(1 - u^2)1_\{|u| \leq 1\} \) we get:

\[
S_{j+1}^{p,q} = \sum_{i=1}^{N} \left(1 - \frac{(x_i - z_j)^2}{h^2}\right) x_i^p y_i^q 1\{z_j - h \leq x_i \leq z_j + h\} \\
= \sum_{i=1}^{N} \left(1 - \frac{z_j^2}{h^2} + \frac{2z_j}{h^2} x_i - \frac{1}{h^2} x_i^2\right) x_i^p y_i^q 1\{z_j - h \leq x_i \leq z_j + h\} \\
= \frac{3}{4Nh} \left\{1 - \frac{z_j^2}{h^2}\right\} S^{p,q}(\{z_j - h, \ z_j + h\}) + \frac{2z_j}{h^2} S^{p+1,q}(\{z_j - h, \ z_j + h\}) - \frac{1}{h^2} S^{p+2,q}(\{z_j - h, \ z_j + h\})
\]

(4.11)

where

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

(4.12)

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

\[
S^{p,q}(\{z_{j+1} - h, \ z_{j+1} + h\}) = \sum_{i=1}^{N} x_i^p y_i^q 1\{z_{j+1} - h \leq x_i \leq z_{j+1} + h\} \\
= \sum_{i=1}^{N} x_i^p y_i^q 1\{z_j - h \leq x_i \leq z_j + h\} \\
- \sum_{i=1}^{N} x_i^p y_i^q 1\{z_j - h < x_i < z_{j+1} - h\} + \sum_{i=1}^{N} x_i^p y_i^q 1\{z_j + h < x_i \leq z_{j+1} + h\} \\
= S^{p,q}(\{z_j - h, \ z_j + h\}) - S^{p,q}(\{z_j - h, \ z_{j+1} - h\}) + S^{p,q}(\{z_{j+1} - h, \ z_{j+1} + h\}) \\
\]

(4.13)

Therefore one can simply update the sum \( S^{p,q}(\{z_j - h, \ z_{j+1} + h\}) \) for the evaluation point \( z_j \) to obtain the next sum \( S^{p,q}(\{z_{j+1} - h, \ z_{j+1} + h\}) \) for the next evaluation point \( z_{j+1} \) by subtracting the terms \( x_i^p y_i^q \) for which \( x_i \) lie between \( z_j - h \) and \( z_{j+1} - h \), and adding the terms \( x_i^p y_i^q \) for which \( x_i \) lie between \( z_j + h \) and \( z_{j+1} + h \). This can be achieved in a fast \( \mathcal{O}(M + N) \) operations by going through the input points \( x_i \), stored in increasing order at a cost of \( \mathcal{O}(N \log N) \) operations, and through the evaluation points \( z_j \), stored in increasing order at a cost of \( \mathcal{O}(M \log M) \) operations.

### 4.10.2 The multivariate case

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

\[ x_i = (x_{1,i}, x_{2,i}, \ldots, x_{d,i}) \ , \ i \in \{1, 2, \ldots, N\} \]

StOpt library uses the additive Epanechnikov kernel in the mutli-dimensional case.

\[
K_d(u_1, \ldots, u_d) = \frac{1}{d^{2d-1}} \sum_{k=1}^{d} K(u_k) \prod_{k_0=1}^{d} \mathbb{1}\{|u_{k_0}| < 1\} = \frac{3}{d^{2d+1}} \sum_{k=1}^{d} (1 - u_{k}^2) \prod_{k_0=1}^{d} \mathbb{1}\{|u_{k_0}| < 1\}
\]

(4.14)

One can show (25) that the computation of the multivariate version of the kernels smoothers \((4.7), (4.8)\) and \((4.9)\) boils down to the computation of the following sums:

\[
S_j = S_{k_1, k_2, j}^{p_1, p_2, q} := \frac{1}{N} \sum_{i=1}^{N} K_d \left( \frac{x_{1,i} - z_{1,j}}{h_1}, \frac{x_{2,i} - z_{2,j}}{h_2}, \ldots, \frac{x_{d,i} - z_{d,j}}{h_d} \right) x_{k_1,i}^{p_1} x_{k_2,i}^{p_2} y_i^q
\]

\[
= \frac{3}{d^{2d+1} N} \sum_{k_1, k_2}^{d} \left( 1 - \frac{z_{k,i}^2}{h_k^2} \right) x_{k_1,i}^{p_1} x_{k_2,i}^{p_2} y_i^q \prod_{k_0=1}^{d} \mathbb{1}\{|x_{k_0,i} - z_{k_0,j}| \leq 1\}
\]

(4.15)

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

**Kernel development**

Using the multivariate kernel (4.14), one can develop the sum (4.15) as follows:

\[
S_{k_1, k_2, j}^{p_1, p_2, q} = \frac{1}{N} \sum_{i=1}^{N} K_d \left( \frac{x_{1,i} - z_{1,j}}{h_1}, \frac{x_{2,i} - z_{2,j}}{h_2}, \ldots, \frac{x_{d,i} - z_{d,j}}{h_d} \right) x_{k_1,i}^{p_1} x_{k_2,i}^{p_2} y_i^q
\]

\[
= \frac{3}{d^{2d+1} N} \sum_{k_1, k_2}^{d} \left( 1 - \frac{z_{k,i}^2}{h_k^2} \right) x_{k_1,i}^{p_1} x_{k_2,i}^{p_2} y_i^q \prod_{k_0=1}^{d} \mathbb{1}\{|x_{k_0,i} - z_{k_0,j}| \leq 1\}
\]

\[
= \frac{3}{d^{2d+1} N} \sum_{k_1, k_2}^{d} \left\{ \left( 1 - \frac{z_{k,i}^2}{h_k^2} \right) S_{[k_1, k_2]}^{p_1, p_2, q}([z_j - h_j, z_j + h_j]) + \frac{2z_{k,i}}{h_k} S_{[k_1, k_2]}^{p_1, p_2, q}([z_j - h_j, z_j + h_j]) \right\}
\]

(4.16)

where

\[
S_{[k_1, k_2]}^{p_1, p_2, q}([z_j - h_j, z_j + h_j]) := \sum_{i=1}^{N} \left( \prod_{l=1}^{d} (x_{k_1,i})^{p_{1,l}} \right) y_i^q \prod_{k_0=1}^{d} \mathbb{1}\{L_{k_0} \leq x_{k_0,i} \leq R_{k_0}\}
\]

(4.17)

for any hypercube \([L, R] := [L_1, R_1] \times [L_2, R_2] \times \ldots \times [L_d, R_d] \subseteq \mathbb{R}^d\), powers \(p := (p_1, p_2, p_3) \in \mathbb{N}^3\), \(q \in \mathbb{N}\) and indices \(k := (k_1, k_2, k_3) \in \{1, 2, \ldots, d\}^3\), and where \([z_j - h_j, z_j + h_j] := [z_{1,j} - h_{1,j}, z_{1,j} + h_{1,j}] \times [z_{2,j} - h_{2,j}, z_{2,j} + h_{2,j}] \times \ldots \times [z_{d,j} - h_{d,j}, z_{d,j} + h_{d,j}]\)
To sum up what has been obtained so far, computing multivariate kernel smoothers (kernel density estimation, kernel regression, locally linear regression) boils down to computing sums of the type \((4.17)\) on hypercubes of the type \([z_j - h_j, z_j + h_j]\) for every evaluation point \(j \in \{1, 2, \ldots, M\}\). In the univariate case, these sums could be computed efficiently by sorting the input points \(x_i, i \in \{1, 2, \ldots, N\}\) and updating the sums from one evaluation point to the next (equation \((4.13)\)). Our goal is now to set up a similar efficient fast sum updating algorithm for the multivariate sums \((4.17)\). To do so, we are first going to partition the input data into a multivariate rectilinear grid (subsection \(4.10.2)\), unlike the univariate case where the input points themselves were being added and removed iteratively.

**Data partition**

The first stage of the multivariate fast sum updating algorithm is to partition the sample of input points into a rectilinear grid. To do so, we partition each dimension independently as follows: for each dimension \(k \in \{1, 2, \ldots, d\}\), the set of threshold points \(\tilde{G}_k := \{z_{k,jk} - h_{k,jk}\}_{jk \in \{1,2,\ldots,M_k\}} \cup \{z_{k,jk} + h_{k,jk}\}_{jk \in \{1,2,\ldots,M_k\}}\) is used to partition the \(k\)-th axis. The second row of Figure 4.3 illustrates this partition on a set of 4 points, where for simplicity the evaluation points are the same as the input points. Denote the sorted points of the partition \(\tilde{G}_k\) as \(\tilde{g}_{k,1} \leq \tilde{g}_{k,2} \leq \ldots \leq \tilde{g}_{k,2M_k}\)

\[
\tilde{G}_k = \{\tilde{g}_{k,1}, \tilde{g}_{k,2}, \ldots, \tilde{g}_{k,2M_k}\}
\]

and define the partition intervals \(\tilde{I}_{k,l} := [\tilde{g}_{k,l}, \tilde{g}_{k,l+1}]\) for \(l \in \{1, 2, \ldots, 2M_k - 1\}\).

Because for each dimension \(k \in \{1, 2, \ldots, d\}\), all the bandwidths edges \(z_{k,jk} - h_{k,jk}\) and \(z_{k,jk} + h_{k,jk}\), \(jk \in \{1,2,\ldots,M_k\}\), belong to \(\tilde{G}_k\), there exists, for any evaluation point \(z_j = (z_{1,j_1}, z_{2,j_2}, \ldots, z_{d,j_d}) \in \mathbb{R}^d\), some indices \((\tilde{L}_{1,j_1}, \tilde{L}_{2,j_2}, \ldots, \tilde{L}_{d,j_d})\) and \((\tilde{R}_{1,j_1}, \tilde{R}_{2,j_2}, \ldots, \tilde{R}_{d,j_d})\) such that

\[
[z_j - h_j, z_j + h_j] = [z_{1,j_1} - h_{1,j_1}, z_{1,j_1} + h_{1,j_1}] \times \ldots \times [z_{d,j_d} - h_{d,j_d}, z_{d,j_d} + h_{d,j_d}]
\]

\[
= [\tilde{g}_{1,\tilde{L}_{1,j_1}}, \tilde{g}_{1,\tilde{R}_{1,j_1}+1}] \times \ldots \times [\tilde{g}_{d,\tilde{L}_{d,j_d}}, \tilde{g}_{d,\tilde{R}_{d,j_d}+1}]
\]

\[
\tilde{I}_{1,j_1} \times \ldots \times \tilde{I}_{d,j_d} \quad (4.18)
\]

and, consequently, such that the sum \((4.17)\) on the hypercube \([z_j - h_j, z_j + h_j]\) is equal to the sum of sums \((4.17)\) on all the hypercubes of the partition included in \([z_j - h_j, z_j + h_j]\) (namely all the hypercubes \(\tilde{I}_{1,j_1} \times \tilde{I}_{2,j_2} \times \ldots \times \tilde{I}_{d,j_d}\) such that \(l_k \in \{\tilde{L}_{k,j_k}, \tilde{L}_{k,j_k} + 1, \ldots, \tilde{R}_{k,j_k}\}\) in each dimension \(k \in \{1, 2, \ldots, d\}\):

\[
S_k^{pq}(z_j - h_j, z_j + h_j) = \bigcup_{(l_1, \ldots, l_d) \in \{\tilde{L}_{1,j_1}, \ldots, \tilde{R}_{1,j_1}\} \times \ldots \times \{\tilde{L}_{d,j_d}, \ldots, \tilde{R}_{d,j_d}\}} S_k^{pq}(\tilde{I}_{1,l_1} \times \ldots \times \tilde{I}_{d,l_d}) \quad (4.19)
\]
where we assume without loss of generality that the bandwidth grid $h_j = (h_{1,j}, h_{2,j}, \ldots, h_{d,j})$, $j_k \in \{1, 2, \ldots, M_k\}$, $k \in \{1, 2, \ldots\}$ is such that $\tilde{G}_k$ does not contain any input $x_{k,i}$, $i \in \{1, 2, \ldots, N\}$, to ensure there is no input point on the boundaries of the inner hypercubes.

Figure 4.3: From bandwidths to partition (1D)

The sum decomposition (4.19) is the cornerstone of the fast multivariate sum updating algorithm, but before going further, one can simplify the partitions $\tilde{G}_k$, $k \in \{1, 2, \ldots\}$ while maintaining a sum decomposition of the type (4.19). Indeed, the partitions $\tilde{G}_k = \{z_{k,j_k} - h_{k,j_k}, z_{k,j_k} + h_{k,j_k}; j_k = 1, \ldots, M_k\}$ can in general produce empty intervals (intervals which do not contain any input points, cf. the grey intervals on the second row of Figure 4.3). To avoid keeping track of sums $S_{p,q}^k$ on the corresponding hypercubes known to be empty, one can trim the partitions $\tilde{G}_k$ by shrinking each succession of empty intervals into one new partition threshold (cf. the final partition on the third row of Figure 4.3). Denote as $G_k$ the resulting simplified partitions, containing the points $g_{k,1} < g_{k,2} < \ldots < g_{k,m_k}$:

$$G_k = \{g_{k,1}, g_{k,2}, \ldots, g_{k,m_k}\}$$

where $2 \leq m_k \leq 2M_k$, $k \in \{1, 2, \ldots\}$, and $m := \prod_{k=1}^d m_k \leq 2^d M$. Define the partition intervals $I_{k,l} := [g_{k,l}, g_{k,l+1}]$, $l \in \{1, 2, \ldots, m_k - 1\}$. Because the only intervals to have been modified from $\tilde{G}_k$ to $G_k$ were empty, the following still holds:

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

$$S_{p,q}^k([z_j - h_j, z_j + h_j]) = \bigcup_{(l_1, \ldots, l_d) \in \{L_{1,j_1}, \ldots, R_{1,j_1}\} \times \ldots \times \{L_{d,j_d}, \ldots, R_{d,j_d}\}} S_{p,q}^k(I_{l_1,1} \times \ldots \times I_{l_d,d}) \quad (4.20)$$

To complement the illustration of univariate partition given by Figure 4.3, Figure 4.4 provides a bivariate partition example. There are four points, each at the center of their respective rectangular kernel (in orange). On the left-hand side, the bandwidths boundaries are used to produce the partitions $\tilde{G}_k$ in each dimension. One can see that most of the resulting hypercubes (rectangles) are empty. On the right-hand side, the empty hypercubes are removed/merged, resulting in the trimmed partitions $G_k$ in each dimension. Remark that this is a simple example for which every final hypercube only contains one point.
Fast multivariate sweeping algorithm

So far, we have shown that computing multivariate kernel smoothers is based on the computation of the kernel sums (4.15), which can be decomposed into sums of the type (4.17), which themselves can be decomposed into the smaller sums (4.20) by decomposing every kernel support of every evaluation point onto the rectilinear partition described in the previous subsection 4.10.2. The final task is to define an efficient algorithm to traverse all the hypercube unions

$$\bigcup_{(l_1, \ldots, l_d) \in \{L_1, j_1, \ldots, R_1, j_1\} \times \ldots \times \{L_d, j_d, \ldots, R_d, j_d\}^d} I_{1,j_1} \times \ldots \times I_{d,j_d}$$

so as to compute the right-hand side sums (4.20) in an efficient fast sum updating fashion that extends the univariate updating (4.13).

First, to simplify notations, we introduce the multi-index $$\text{idx} := (p, q, k) \in \{0, 1, 2\} \times \{0, 1\} \times \{1, 2, \ldots, d\}^3$$ to summarize the polynomial

$$\left(\prod_{l=1}^3 (x_{k_l,i})^{p_l}\right) y^q_i$$

in the sum $$S^{p,q}_k ([L, R])$$ (equation (4.17)), and introduce the compact notation

$$S^{\text{idx}}_{l_1,l_2,\ldots,l_d} := S^{p,q}_k (I_{1,j_1} \times \ldots \times I_{d,j_d})$$

(4.21) to simplify the notation on the right-hand side of equation (4.20). In summary, $$S^{\text{idx}}_{l_1,l_2,\ldots,l_d}$$ corresponds to the sum of the polynomials $$\left(\prod_{l=1}^3 (x_{k_l,i})^{p_l}\right) y^q_i$$ over all the data points within the hypercube $$I_{1,j_1} \times \ldots \times I_{d,j_d}$$. We precompute all the sums $$S^{\text{idx}}_{l_1,l_2,\ldots,l_d}$$, and use them as the input material for the fast multivariate sum updating.

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

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

(4.22)
The second stage is to perform a fast sum updating in the second dimension, with the sums $T_{1,2}^{idx} = \sum_{l_2=L_{1,2}}^{R_{1,2}} S_{l_1,l_2}^{idx}$ as input material. Our goal is to compute the sums $T_{2}^{idx} := \sum_{l_2=L_{2,2}}^{R_{2,2}} T_{1,2}^{idx}$ for every indices interval $[L_{2,2}, R_{2,2}]$, $j_2 \in \{1, 2, \ldots, M_2\}$. In a similar manner, we start with $j_2 = 1$ and the initial sum $T_{2}^{idx} = \sum_{l_2=L_{2,1}}^{R_{2,1}} T_{1,2}^{idx}$. We then increment $j_2$ from $j_2 = 1$ to $j_2 = M_2$ iteratively. After each incrementation of $j_2$, we update $T_{2}^{idx}$ by fast sum updating:

$$
\sum_{l_2=L_{2,j_2}}^{R_{2,j_2}} T_{1,2}^{idx} = \sum_{l_2=L_{2,j_2-1}}^{R_{2,j_2-1}} T_{1,2}^{idx} + \sum_{l_2=R_{2,j_2}-1+1}^{R_{2,j_2}} T_{1,2}^{idx} - \sum_{l_2=L_{2,j_2-1}}^{L_{2,j_2}-1} T_{1,2}^{idx} \quad (4.23)
$$

Using the notation change $(4.21)$ and equation $(4.20)$, the resulting sum $\sum_{l_2=L_{2,j_2}}^{R_{2,j_2}} T_{1,2}^{idx} = \sum_{l_2=L_{1,j_2}}^{R_{1,j_2}} S_{l_1,l_2}^{idx}$ is equal to $S_{k,q}^{p,q}([z_j - h_j, z_j + h_j])$, which can be used to compute the kernel sums $S_j = S_{k_1,k_2}^{p_1,p_2,q}$ using equation $(4.16)$, from which the bivariate kernel smoothers (kernel density estimator, kernel regression, locally linear regression) can be computed.

This ends the description of the fast sum updating algorithm in the bivariate case. Finally, the general multivariate case is a straightforward extension of the bivariate case.

### 4.10.3 C++ API

The constructor permits to defines the kernel regressor:

```cpp
LocalGridKernelRegression(const bool & p_bZeroDate,
                          const std::shared_ptr<Eigen::ArrayXXd> & p_particles,
                          const double & p_coeffBandWidth,
                          const double & p_coefNbGridPoint,
                          const bool & p_bLinear);
```

where

- $p_{bZeroDate}$ is true if the regression date is 0,
- $p_{particles}$ the particles $X_t$ for all simulations (dimension of $X_t$ for first dimension, number of Monte Carlo simulations in second dimension),
- $p_{coeffBandWidth}$ between 0 and 1 defines the percentage of points to be used to define the bandwidth for each point.
- $p_{coefNbGridPoint}$ is a multiplicative factor defining the number of points $z$ used for the multigrid approximation : a PCA is used to define a rotation of the data. The kernel regression is achieved according the base defined by the eigenvectors associated to the PCA. The number of points along the axes defined by the eigenvectors is given according the singular value associated to the eigenvector. The total number of evaluation points along the axes of the new base is roughly the number of simulations ($p_{particles}.cols()$) by $p_{coefNbGridPoint}$.
- $p_{bLinear}$ when set to false states that the simple kernel density estimation $(4.8)$ is used. When $p_{bLinear}$ is true, the linear kernel regression $(4.9)$ is used.
Below we give a small example where “toRegress” corresponds to $g(t + h, X_{t+h})$ for all simulations and $x$ store $X_t$ for all simulations.

```cpp
// t is not zero
bool bZeroDate = 0;
// proportion of points used to define bandwidth
double prop = 0.1;
// multiplicative factor equal to one: number of evaluation points equals to the number of particles
double q = 1.
// choose a linear regression
bool bLin = true;
// constructor
LocalGridKernelRegression kernelReg(bZeroDate, x, prop, q, bLin);
// update particles values
localRegressor.updateSimulations(bZeroDate, x);
// regressed values
ArrayXd regressedValues = localRegressor.getAllSimulations(toRegress);
```

### 4.10.4 Python API

As usual, the python constructors are similar to the c++ constructors. Here is a small example of the use of the kernel regression method.

```python
import StOptReg

nbSimul = 5000000;
np.random.seed(1000)
x = np.random.uniform(-1., 1., size=(1, nbSimul));
# real function
toReal = (2 + x[0, :]) + (+x[0, :]) * (1 + x[0, :])
# function to regress
toRegress = toReal + 4 * np.random.normal(0., nbSimul)
# bandwidth
bandwidth = 0.1
# factor for the number of points
factPoint = 1
# Regressor
regressor = StOptReg.LocalGridKernelRegression(False, x, bandwidth, factPoint, True)
# nb simul
nbSimul = regressor.getNbSimul()
# particles
part = regressor.getParticles()
# get regressed
y = regressor.getAllSimulations(toRegress).transpose()[0]
```
Chapter 5

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.

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

It is use when the transition problem is solve using a LP.

5.1 Continuation values object

A special case is the case where the state $X^{x,t}$ in equation (2.1) 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

   $$dX^{x,t}_{s,1} = b(t, X^{x,t}_{s,1})ds + \sigma(s, X^{x,t}_{s,1})dW_s$$

   and is not controlled: the stochastic process is exogenous,

2. the second part is given by the following equation

   $$dX^{x,t}_{s,2} = b_a(t)ds$$

   such that the $X^{x,t}_2$ is a degenerated version of [2.1] 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 0 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 $X_{1}^{x,t}$ is the value of the gas on the market and $X_{2}^{x,t}$ is the position (in volume) in the gas storage. The library offers to store the conditional expectation for all the states $X_{2}^{x,t}$.

- $X_{2}^{x,t}$ will be stored on a grid of points (see section 3).
- for each point $i$ of the grid the conditional expectation of a function $g_{i}(X_{2}^{x,t})$ associated to the point $i$ using a regressor (see section 3) can be calculated and stored such that the continuation value $C$ is a function of $(X_{1}^{x,t},X_{2}^{x,t})$.

### 5.1.1 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 (4.1)),

- 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 (4.1)).

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 \( \alpha_i \) 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:

```cpp
Eigen::ArrayXd getAllSimulations(const Eigen::ArrayXd &p_ptOfStock)
```

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

```cpp
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.

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

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

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

• a method that permits to calculate the conditional expectation for a sample of \( X_{x,t} \):

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

where:

- \( p.ptOfStock \) the point where we interpolate the conditional expectation (a realization of \( X_{2,x,t} \))
- \( p.coordinates \) the sample of \( X_{1,x,t} \) used to estimate the conditional expectation
- and the function returns \( C(X_{1,x,t},X_{2,x,t}) \).

Below we regress an identical function for all 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;
```
5.1.2 Python API

Here is an example of the use of the mapping

```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
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)
```
5.2 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 be used. As for the continuation it has a “getValue” to estimate the function at a state with both a deterministic and a stochastic part.

5.2.1 C++ API

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

- The first one

```
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 \( C(X_{1,s}^{x,t}, X_{2,s}^{x,t}) \) value for a point \( X_{s}^{x,t} = (X_{1,s}^{x,t}, X_{2,s}^{x,t}) \) where \( X_{2,s}^{x,t} \) is on the grid and \( X_{1,s}^{x,t} \) is the part treated by regression.

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

where:

- `p_ptOfStock` \( X_{2,s}^{x,t} \) part of \( X_{s}^{x,t} \)
- `p_coordinates` \( X_{1,s}^{x,t} \) part of \( X_{s}^{x,t} \).

- the method ‘getRegressedValues’ 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 &p_regValues)
```

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

### 5.2.2 Python API

The python API is similar to the one of the ContinuationValue object (voir section 5.3.2).

### 5.3 The continuation cut object

Suppose the control problem is continuous and that the state of the system has the dynamic given by [5.1] and [5.2]. This is the case of some storages modeled associated to the maximization of a certain objective function. Then the Bellman value associated to this problem is concave. For a given value of some markovian process \( X_{s,1}^{x,t} \), the Bellman curve
can be approximated by cuts (see 5.1). Solving a PL for a given uncertainty and a given state in the storage levels \( \hat{y}_i \) in dimension \( d \), we get a cut

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

For \( s' \leq s \) a conditional cut can be obtained calculating

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

Than using a regressor (see chapter 4) it is possible to represent each conditional cut on a basis for \( j = 0, \ldots, d \).

\[
\mathbb{E} \left[ a_i(X_{s,1}^{x,t})|X_{s',1}^{x,t} \right] = \sum_{j=1}^{N} a_{i,j} \psi_j(X_{s',1}^{x,t}) \quad (5.3)
\]

where the \( \psi_j \) correspond to some basis function.

### 5.3.1 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 \( a_{k,j} \) for the grid point \( k \);
- The second one
ContinuationCuts(const shared_ptr< SpaceGrid > & p_grid,
const shared_ptr< BaseRegression > & p_condExp,
const Eigen::ArrayXXd & p_values)

with

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

This constructor constructs for all stock points the \( a_{i,j} \) coefficients of the cuts (5.3).

Notice that for a stock point \( k \) with coordinates \( y^k \), the coefficients stored are \( \hat{a}^k_{0,j} = a^k_{0,j} - \sum_{i=1}^d a^k_{i,j} y^k_i \) and the \( \hat{a}^k_{i,j} = a^k_{i,j}, i = 1, \ldots d \). Then the conditional cut can be written as:

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

The main methods provided are:

- a first method used in simulation 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 \( a_{i,j} \) coefficients to reconstruct the cuts: its size corresponds to the number of cut coefficients. Then the element \( i \) of `p_values` permits to store the \( a_{i,j} \) coefficients for \( j = 1, \ldots, N \) and all stock points.

- a second method taking as input a description of an hypercube (nb storages,2) describes by its extremal coordinates :
  - \((i,0)\) coordinate corresponds to minimal coordinate value in dimension \( i \)
  - \((i,1)\) coordinate corresponds to maximal coordinate value in dimension \( i \)

```cpp
Eigen::ArrayXXd getCutsAllSimulations(const Eigen::ArrayXXd & p_hypStock) const
```
It returns an array of cuts coefficients for all particles state stored in its BaseRegression member.

- First dimension corresponds to the number of cuts coefficients by the number of simulations.
- The second dimension corresponds to the number of points in the hypercube \( p_{hypStock} \).

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

\[
\text{Eigen} \colon \text{ArrayXXd} \ \text{getCutsASim} (\ \text{const} \ \text{Eigen} \colon \text{ArrayXXd} \ \& p_{hypStock}, \ \text{const} \ \text{Eigen} \colon \text{ArrayXd} \ \& p_{coordinates}) \ \text{const}
\]

where:

- \( p_{hypStock} \) corresponds to an hypercube used to select some stock points as previously,
- \( p_{coordinates} \) corresponds to the coordinates of the uncertainty to consider.

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

- a method that permits to get the coefficients calculated.

\[
\text{const} \ \text{Eigen} \colon \text{Array<} \ \text{Eigen} \colon \text{ArrayXXd, Eigen} \colon \text{Dynamic, 1} > \ \& \ \text{getValues} (\ \text{const})
\]

### 5.3.2 Python API

Here is an example of the use of the mapping

```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
import StOptReg

# unit test for continuation values
# # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #
class testContValues(unittest.TestCase):
    # unit test for continuation cuts
    def testSimpleGridsAndRegressor(self):
```

83
# 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)

# Create the regressor
regressor = StOptReg.LocalLinearRegression(False, x, nbMesh)

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

# function to regress
toRegress = toReal + 4*np.random.normal(0., 1, nbSimul)

# fictitious cuts with 0 sensibility (1 dimension)
toRegressCuts = np.zeros(shape=(4*len(toRegress), grid.getNbPoints(1))):
    for i in range(toRegressCuts.shape[1], range(len(toRegress))):
        toRegressCuts[:, i] = toRegress

# Now create the continuation cut object
contOb = StOptReg.ContinuationCut(grid, regressor, toRegressCuts)

hyperCube = np.array([[lowValues[0], lowValues[0]+step[0]*nbStep[0]],
                       [lowValues[1], lowValues[1]+step[1]*nbStep[1]],

regressCuts = contOb.getCutsAllSimulations(hyperCube)

if __name__ == '__main__':
    unittest.main()
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>

/* \file SimulatorDPBase.h
   * \brief Abstract class for simulators for Dynamic Programming Programms
   * \author Xavier Warin
   */

namespace StOpt
{
    //\class SimulatorDPBase SimulatorDPBase.h
    //\abstract class for simulator used in dynamic programming
    class SimulatorDPBase
    {
    public :

        //\brief Constructor
        SimulatorDPBase() {} 
        //\brief Destructor
        virtual ~SimulatorDPBase() {} 
        //\brief get current markovian state : dimension of the problem for the first dimension , second dimension the number of Monte Carlo simulations
        virtual Eigen::MatrixXd getParticles() const = 0;
        //\brief a step forward for simulations
        virtual void stepForward() = 0;
        //\brief a step backward for simulations
        virtual void stepBackward() = 0;
        //\brief a step forward for simulations
        //\return current particles (markovian state as assets for example) (dimension of the problem times simulation number)
        virtual Eigen::MatrixXd stepForwardAndGetParticles() = 0;
        //\brief a step backward for simulations
        //\return current particles (markovian state as assets for example) (dimension of the problem times simulation number)
        virtual Eigen::MatrixXd stepBackwardAndGetParticles() = 0;
        //\brief get back dimension of the regression
        virtual int getDimension() const = 0;
        //\brief get the number of steps
        virtual int getNbStep() const = 0;
        //\brief Get the current step size
        virtual double getStep() const = 0;
    }
}
```
Supposing that the Simulator is a Black Scholes simulator for \( P \) assets, simulating \( M \) Monte Carlo simulations, at \( N + 1 \) dates \( t_0, \ldots, t_N \), 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 6

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.

6.1 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

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

(6.1)

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.

Algorithm 3 Algorithm with regression [optimal exercise time estimation]

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

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

Price estimator at 0: \( \hat{P}_0^{1,\pi} := \mathbb{E}[g(\hat{\tau}_0^{1,\pi}, X_{\hat{\tau}_0^{1,\pi}})] \).
6.1.1 American option with the C++ API

We value in the algorithm below an American option using a simulator \textit{p\_sim}, a regressor \textit{p\_regressor}, a payoff function \textit{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 payoff
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); // payoff
    // 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 payoff and cash delivered after
    Cash = (condEspec.array() < payoffLoc.array()).select(payoffLoc, Cash * expRate);
}
return Cash.mean();
```

6.2 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 payoff
# 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 7

Using the general framework to manage stock problems

In this chapter the state is separated into three parts $X^{x,t} = (X^{x,t}_1, X^{x,t}_2, I_t)$, which corresponds to the special case of chapter 5 where $X^{x,t}_1$ is not controlled and $X^{x,t}_2$ is controlled. Two cases can be tackled:

- the first case corresponds to the case where $X^{x,t}_2$ is deterministic (think of storage management),

- the second case corresponds to the case where $X^{x,t}_2$ 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 $X^{x,t}_2$ is discretized on a grid as explained in chapter 5.

- Either a full grid is used for $X^{x,t}_2$ 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 $X^{x,t}_2$ 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 $X^{x,t}_2$ deterministic is treated.

In the case of the MPI parallelization technique distributing task and data (full grids only), 29 and 40 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 7.1) 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 [7.1] 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 7.1: Data to send to processor 3](image)

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

### 7.1 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”

```cpp
// 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
#define OPTIMIZERBASE_H
#include <Eigen/Dense>
#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 if 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 Get the number of regimes allowed for the asset to be reached at the current time step
virtual int getNbRegime() const = 0;  

/// \brief get the simulator back
virtual std::shared_ptr< StOpt::SimulatorDPBase > getSimulator() const = 0;  

/// \brief get back the dimension of the control
virtual int getNbControl() const = 0;  

/// \brief get size of the function to follow in simulation
virtual int getSimuFuncSize() const = 0;  

};

#endif /* OPTIMIZERBASE_H */
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 $H1$. It returns for each dimension, the coordinates min and max of the hyper cube $H2$ containing the points that can be reached by applying a command from a grid point in $H1$.

- 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 $p\_state$ (storing the $X^{x,t}$), the optimal control calculated in optimization $p\_control$, the optimal functions values along the current trajectory are stored in $p\_phiInOut$. The state $p\_state$ 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 $X^{x,t}_2$ 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.
7.2 Solving the problem using conditional expectation calculated by regressions

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

7.2.1 Requirement to use the framework

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

- the “stepSimulateControl” method is used after optimization using the optimal controls calculated in the optimization part. From a state $p\_state$ (storing the $X_{x,t}$), the optimal control calculated in optimization $p\_control$, the optimal functions values along the current trajectory are stored in $p\_phiInOut$. The state $p\_state$ is updated during at the end of the call function.

7.2.2 Classical regression

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

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

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

#ifdef 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/OptimizerBaseInterp.h"

/** \file OptimizerDPBase.h
 * \brief Define an abstract class for Dynamic Programming problems solved by regression methods
 * \author Xavier Warin
 */
namespace StOpt {

```
/// \class OptimizerDPBase OptimizerDPBase.h
/// \brief Base class for optimizer for Dynamic Programming with regression methods

class OptimizerDPBase : public OptimizerBaseInterp
{

public :

  OptimizerDPBase() {}  
  virtual ~OptimizerDPBase() {}  

  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::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;

};
We detail the different methods derived from “OptimizerDPBase.h” to implement in addition to the methods of “OptimizerBaseInterp.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 $p_{\text{stock}}$ 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 $p_{\text{state}}$ (storing the $X^{x,t}$), the continuation values calculated in optimization $p_{\text{continuation}}$, the optimal function values along the current trajectory are stored in $p_{\text{phiInOut}}$.

In the case of a gas storage [42], 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.

The framework in optimization with classical regressions

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

- $p_{\text{pGridCurrent}}$ is the grid at the current time step ($t_i$),
- $p_{\text{pGridPrevious}}$ is the grid at the previously treated time step ($t_{i+1}$),
- $p_{\text{pOptimize}}$ 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
std::vector< std::shared_ptr< Eigen::ArrayXXd > > OneStep(const std::vector< std::shared_ptr< Eigen::ArrayXXd > >& p_phiIn ,
const std::shared_ptr< BaseRegression>& p_condExp)

with

- \( p_{\text{phiIn}} \) 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.

- \( p_{\text{condExp}} \) 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).

**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:

```cpp
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
```

with:

- \( p_{\text{ar}} \) is the archive where controls and solutions are dumped,

- \( p_{\text{name}} \) is a base name used in the archive to store the solution and the control,

- \( p_{\text{phiInPrev}} \) is the solution at the previous time step used to calculate the continuation values that are stored,

- \( p_{\text{control}} \) stores the optimal controls calculated at the current time step,

- \( p_{\text{condExp}} \) is the conditional expectation object permitting to calculate conditional expectation of functions defined at the previous time step treated \( p_{\text{phiInPrev}} \) and permitting to store a representation of the optimal control.
- \texttt{p\_bOneFile} is set to one if the continuation and optimal controls calculated by each processor are dumped on a single file. Otherwise the continuation and optimal controls calculated by each processor are dumped on different files (one 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 “\texttt{TransitionStepRegressionDP}” and the “\texttt{TransitionStepRegressionDPSparse}” object, their “\texttt{dumpContinuationValues}” doesn’t need a \texttt{p\_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
// Copyright (C) 2016 EDF
// All Rights Reserved
// This code is published under the GNU Lesser General Public License (GNU LGPL)
#elseif defined USE_MPI
#include <fstream>
#include <memory>
#include <functional>
#include <boost/lexical_cast.hpp>
#include <boost/mpi.hpp>
#include <Eigen/Dense>
#include "geners/BinaryFileArchive.hh"
#include "StOpt/core/grids/FullGrid.h"
#include "StOpt/regression/BaseRegression.h"
#include "StOpt/dp/FinalStepRegressionDPDist.h"
#include "StOpt/dp/TransitionStepRegressionDPDist.h"
#include "StOpt/core/parallelism/reconstructProc0Mpi.h"
#include "StOpt/dp/OptimizerDPBase.h"
#include "StOpt/dp/SimulatorDPBase.h"

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
```
vector< shared_ptr< Eigen::ArrayXXd > > valuesNext = StOpt::FinalStepRegressionDPDist(p_grid, p_optimize->getNbRegime(), p_optimize->getDimensionToSplit())(p_funcFinalValue, simulator->getParticles().array());

// dump
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 = "Continuation";
for (int iStep = 0; iStep < simulator->getNbStep(); ++iStep)
{
    shared_ptr<Eigen::ArrayXXd> asset = make_shared<Eigen::ArrayXXd>(simulator->stepBackwardAndGetParticles());
    // conditional expectation operator
    p_regressor->updateSimulations(((iStep == (simulator->getNbStep() - 1)) ? true : false), asset);
    // transition object
    StOpt::TransitionStepRegressionDPDist transStep(p_grid, p_grid, p_optimize);
    pair< vector< shared_ptr< Eigen::ArrayXXd > >, vector< shared_ptr< Eigen::ArrayXXd > > > valuesAndControl = transStep.oneStep(valuesNext, p_regressor);
    transStep.dumpContinuationValues(ar, nameAr, iStep, valuesNext, valuesAndControl.second, p_regressor, p_bOneFile);
    valuesNext = valuesAndControl.first;
}
// reconstruct a small grid for interpolation
return StOpt::reconstructProc0Mpi(p_pointStock, p_grid, valuesNext[p_initialRegime], p_optimize->getDimensionToSplit()).mean();

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.

Table 7.1: Which “TransitionStepRegression” object to use depending on the grid used and the type of parallelization used.

<table>
<thead>
<tr>
<th>Parallelization on calculations threads and MPI</th>
<th>Full grid</th>
<th>Sparse grid</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequential</td>
<td>“TransitionStepRegressionDP”</td>
<td>“TransitionStepRegressionDPSparse”</td>
</tr>
<tr>
<td>Parallelization on calculations and data</td>
<td>“TransitionStepRegressionDP”</td>
<td>“TransitionStepRegressionDPSparse”</td>
</tr>
<tr>
<td>Distribution of calculations and data</td>
<td>“TransitionStepRegressionDPDist”</td>
<td>Not available</td>
</tr>
</tbody>
</table>
The framework in simulation with classical regressions

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 5) 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`),
- \textit{p.Optimize} the Optimizer describing the transition from one time step to the following one,
- \textit{p.OneFile} equal to true if a single archive is used to store continuation values.

\textbf{Remark 10} A version without distribution of data but with multithreaded and with MPI possible on calculations is available with the object \textit{SimulateStepRegression}. The \textit{p.OneFile} argument is omitted during construction.

This object implements the method “oneStep”

\begin{verbatim}
void oneStep(std::vector<StateWithStocks> &p_statevector, Eigen::
            ArrayXXd &p_phiInOut)
\end{verbatim}

where:

- \textit{p.statevector} store the states for the all the simulations: this state is updated by application of the optimal command,
- \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 “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:

\begin{verbatim}
// Copyright (C) 2016 EDF
// All Rights Reserved
// This code is published under the GNU Lesser General Public License (GNU LGPL)
#define SIMULATEREGEREGRESSIONDIST_H
#include <functional>
#include <memory>
#include <Eigen/Dense>
#include <boost/mpi.hpp>
#include "geners/BinaryFileArchive.hh"
#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 SimulateRegressionDist.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
\end{verbatim}
### \param \ p_{\text{grid}}
grid used for deterministic state (stocks for example)

### \param \ p_{\text{optimize}}
optimizer defining the optimization between two time steps

### \param \ p_{\text{funcFinalValue}}
function defining the final value

### \param \ p_{\text{pointStock}}
initial point stock

### \param \ p_{\text{initialRegime}}
regime at initial date

### \param \ p_{\text{fileToDump}}
name associated to dumped bellman values

### \param \ p_{\text{bOneFile}}
do we store continuation values in only one file

```cpp
double SimulateRegressionDist ( const std :: shared_ptr<StOpt :: FullGrid>& p_grid ,
   const std :: shared_ptr<StOpt :: FullGrid>& 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
    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
    gs :: BinaryFileArchive ar ( toDump . c_str () , "r" ) ;
    // 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 )
      {.....
```

104
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”


where:

- \textit{p.statevector} stores for all the simulations the state: this state is updated by application of the optimal commands,

- \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 “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
#include <functional>
#include <memory>
#include <Eigen/Dense>
#include <boost/mpi.hpp>
#include "geners/BinaryFileArchive.hh"
#include "StOpt/core/grids/FullGrid.h"
#include "StOpt/core/utils/StateWithStocks.h"
#include "StOpt/dp/SimulateStepRegressionControlDist.h"
#include "StOpt/dp/OptimizerDPBase.h"
#include "StOpt/dp/SimulatorDPBase.h"

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

/// \brief Simulate the optimal strategy using optimal controls calculated in optimization, mpi version
/// \param p.grid grid used for deterministic state (stocks for example)
/// \param p_optimize optimizer defining the optimization between two time steps
/// \param p_funcFinalValue function defining the final value
/// \param p_pointStock initial point stock
/// \param p_initialRegime regime at initial date
/// \param p_fileToDump name associated to dumped bellman values
```
```cpp
/// \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 particles = simulator->stepForwardAndGetParticles();
    for (int is = 0; is < simulator->getNbSimul(); ++is)
    {
      states[is].setStochasticRealization(particles.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());
  }

  return 0;
}
```

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.
Table 7.2: 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;&lt;br&gt;&quot;bOneFile&quot; = True</th>
<th>&quot;TransitionStepRegressionDPDist&quot;&lt;br&gt;&quot;bOneFile&quot; = False</th>
<th>&quot;TransitionStepRegression.DPSparse&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>No</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>
7.2.3 Regressions and cuts for linear continuous transition problems with some concavity, convexity features.

When optimizing for example a storage, we may want to solve the transition problem on some time steps supposing that uncertainties are known. The Bellman values for a given uncertainty are then concave with respect to the storage when trying to maximize some earnings for example. When the problem is linear continuous, we can use some bender cuts to approximate the Bellman value with respect to the storage level as in the SDDP method. In order to use this cuts a business object using some LP solver has to be created. This business object is derived is derived from “OptimizerDPCutBase.h” itself derived from “OptimizerBaseInterp.h”.

```cpp
// Copyright (C) 2019 EDF
// All Rights Reserved
// This code is published under the GNU Lesser General Public License (GNU LGPL)
#if !defined OPTIMIZERDPCUTBASE_H
#define OPTIMIZERDPCUTBASE_H
#include <Eigen/Dense>
#include "StOpt/core/utils/StateWithStocks.h"
#include "StOpt/core/grids/SpaceGrid.h"
#include "StOpt/regression/BaseRegression.h"
#include "StOpt/regression/ContinuationCuts.h"
#include "StOpt/dp/SimulatorDPBase.h"
#include "StOpt/dp/OptimizerBase.h"

/** \ile OptimizerDPCutBase.h
* \brief Define an abstract class for Dynamic Programming problems solved by regression methods using cuts to approximate Bellman values
* \author Xavier Warin
*/
namespace StOpt {

/// \class OptimizerDPCutBase OptimizerDPCutBase.h
/// Base class for optimizer for Dynamic Programming with regression methods and cuts, so using LP to solve transitional problems
class OptimizerDPCutBase : public OptimizerBase {

public :

    OptimizerDPCutBase() {}

    virtual ~OptimizerDPCutBase() {}

    /// \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

```cpp
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.grid grid at arrival step after command
///param p.stock coordinates of the stock point to treat
///return For each regimes (column) gives the solution for each
particle, and cut (row)
///For a given simulation, cuts components (C) at a point
stock \( \sum_{i=1}^d C[i] (S_i - \text{bat } S_i) \) are given such that the cut is given by
virtual Eigen::ArrayXXd stepOptimize(const std::shared_ptr<StOpt::SpaceGrid> &p_grid, const Eigen::ArrayXd &p_stock,

```cpp
virtual void stepSimulate(const std::shared_ptr<StOpt::SpaceGrid> &p_grid, const std::vector<StOpt::ContinuationCuts> &p_continuation,
StOpt::StateWithStocks &p_state,
Eigen::Ref<Eigen::ArrayXd> p_phiInOut) const = 0;
```

///brief defines a step in simulation
///Control are recalculated during simulation using a local optimization
using the LP
///param p.grid grid at arrival step after command
///param p.continuation defines the continuation operator for each
regime
///param p.state defines the state value (modified)
///param p.phiInOut defines the value functions (modified) : size
number of functions to follow
virtual void stepSimulate(const std::shared_ptr<StOpt::SpaceGrid> &p_grid, const std::vector<StOpt::ContinuationCuts> &p_continuation,
StOpt::StateWithStocks &p_state,
Eigen::Ref<Eigen::ArrayXd> p_phiInOut) const = 0;

///brief Get the number of regimes allowed for the asset to be reached
at the current time step
///If \( t \) is the current time, and \( \Delta t \) the resolution
step, this is the number of regime allowed on \( t \) \( t \Delta t \) \( t \)
virtual int getNbRegime() const = 0;

///brief get the simulator back
virtual std::shared_ptr<StOpt::SimulatorDPBase> getSimulator() const = 0;

///brief get back the dimension of the control
virtual int getNbControl() const = 0;
We detail the different methods to implement in addition to the methods of “OptimizerBaseInterp.h”:

- the “stepOptimize” methods is used in optimization. We want to calculate the optimal value and the corresponding sensibilities with respect to the stocks at current $t_i$ at a grid point $p_{stock}$ using a grid $p_{grid}$ at the next date $t_{i+1}$, the continuation cuts values for all regimes $p_{condEsp}$ permitting to calculate an upper estimation (when maximizing) of conditional expectation of the optimal values using some optimization calculated at the previously treated time step $t_{i+1}$. From a grid point $p_{stock}$ it calculates the function values and the corresponding sensibilities. It returns a matrix (first dimension is the number of simulations by the number of cuts components (number of storage +1), second dimension the number of regimes) giving the function value and sensibilities.

- the “stepSimulate” method is used after optimization using the continuation cuts values calculated in the optimization part. From a state $p_{state}$ (storing the $X_{x,t}$), the continuation cuts values calculated in optimization $p_{continuation}$, the optimal cash flows along the current trajectory are stored in $p_{phiInOut}$.

In the case of a gas storage the Optimize object is given in [this file](#).

### The framework in optimization using some cuts methods

Once an Optimizer object describing to the Business problem to solve with cust is created for the project, and supposing that a full grid is used for the stock discretization, the framework provides a [“TransitionStepRegressionDPCutDist”](#) object in MPI that permits to solve the optimization problem with distribution of the data on one time step with the following constructor:

```
TransitionStepRegressionDPCutDist (const shared_ptr<FullGrid> & p_pGridCurrent,
   const shared_ptr<FullGrid> & p_pGridPrevious,
   const shared_ptr<OptimizerDPCutBase> & 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
The construction is very similar to classical regression methods only using command discretization.

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

The main method is

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

with

- `p_phiIn` the vector (its size corresponds to the number of regimes) of matrix of optimal values and sensibilities calculated at the previous time iteration for each regime. Each matrix has a number of rows equal to the number of simulations by the number of stock plus one. The number of columns is equal to the number of stock points on the grid. In the row, the number of simulations by the number of stock plus one value are stored as follows:
  - The first values (number of simulations: NS) corresponds to the optimal Bellman values at a given stock point,
  - The NS values following corresponds to sensibilities $\frac{\partial V}{\partial S_1}$ to first storage
  - The NS values following corresponds to sensibilities to the second storage...
  - ...
- `p_condExp` the conditional expectation operator,

returning a vector of matrix with new optimal values and sensibilities at the current time step (each element of the vector corresponds to a regime and each matrix has a size equal to the (number of simulations by (the number of storage plus one)) by the number of stock points). The structure of the output is then similar to the input `p_phiIn`.

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

```cpp
void dumpContinuationCutsValues(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::shared_ptr< BaseRegression > &p_condExp ,
const bool &p_bOneFile ) const
```

with:

- `p_ar` is the archive where controls and solutions are dumped,
- `p_name` is a base name used in the archive to store the solution and the control,
• \textit{p.phiInPrev} is the solution at the previous time step used to calculate the continuation cuts values that are stored,

• \textit{p.condExp} is the conditional expectation object permitting to calculate conditional expectation of functions defined at the previous time step treated \(p.phiInPrev\).

• \textit{p.bOneFile} is set to one if the continuation cuts values calculated by each processor are dumped on a single file. Otherwise the continuation cuts values calculated by each processor are dumped on different files (one by processor). If the problem gives continuation cuts values on the global grid that can be stored in the memory of the computation node, it can be more interesting to dump them in one file for the simulation of the optimal policy.

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

```cpp
// Copyright (C) 2019 EDF
// All Rights Reserved
// This code is published under the GNU Lesser General Public License (GNU LGPL)
#ifndef USE_MPI
#include <fstream>
#include <memory>
#include <functional>
#include <boost/lexical_cast.hpp>
#include <boost/mpi.hpp>
#include <Eigen/Dense>
#include "generators/BinaryFileArchive.hh"
#include "StOpt/core/grids/FullGrid.h"
#include "StOpt/regression/BaseRegression.h"
#include "StOpt/dp/FinalStepRegressionDPCutDist.h"
#include "StOpt/dp/TransitionStepRegressionDPCutDist.h"
#include "StOpt/core/parallelism/reconstructProc0Mpi.h"
#include "StOpt/dp/OptimizerDPCutBase.h"
#include "StOpt/dp/SimulatorDPBase.h"
using namespace std;
using namespace Eigen;

double DynamicProgrammingByRegressionCutDist(const shared_ptr<StOpt::FullGrid> &p_grid,
const shared_ptr<StOpt::OptimizerDPCutBase> &p_optimize,
shared_ptr<StOpt::BaseRegression> &p_regressor,
const function<ArrayXd(const int &, const ArrayXd &, const ArrayXd &)> &p_funcFinalValue,
const 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();
```
An example without distribution of the data can be found in [this file](#).

The framework in simulation using cuts to approximate the Bellman values

Once the optimization has been achieved, continuation cuts values are dumped in one file (or some files) at each time step. In order to simulate the optimal policy, we use the continuation cuts values previously calculated.

- For most of the case (small dimensional case), the optimal cut function values can be stored in the memory of the computing node and cuts values 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 “SimulateStepRegressionCut” object 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 cuts values on one node. In this case, optimal cuts values 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 “SimulateStepRegressionCutDist” objects.

We detail the simulations objects using the optimal cuts values calculated in optimization for the case of very big problems.

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

```c++
SimulateStepRegressionCutDist(gs::BinaryFileArchive &p_ar, const int &p_iStep, const std::string &p_nameCont,
                               const shared_ptr<FullGrid> &p_pGridFollowing,
                               const shared_ptr<OptimizerDPCutBase> &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`),

• `p_Optimize` the Optimizer describing the transition problem solved using a LP program.

• `p_OneFile` equal to true if a single archive is used to store continuation values.

**Remark 13** A version without distribution of data but with multithreaded and with MPI possible on calculations is available with the object “SimulateStepRegressionCut”. The `p_OneFile` argument is omitted during construction.

This object implements the method “oneStep”

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

where:

• `p_statevector` store the states for all the simulations: this state is updated by application of the optimal command,
• \texttt{p\_phiInOut} stores the gain/cost functions for all the simulations: it is updated by the function call. The size of the array is \((nb, nbSimul)\) where \(nb\) is given by the “getSimuFuncSize” method of the optimizer and \(nbSimul\) the number of Monte Carlo simulations.

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

```c++
// Copyright (C) 2019 EDF
// All Rights Reserved
// This code is published under the GNU Lesser General Public License (GNU LGPL)

#define SIMULATEREGRESSIONCUTDIST_H
#define SIMULATEREGRESSIONCUTDIST_H
#include <functional>
#include <memory>
#include <Eigen/Dense>
#include <boost/mpi.hpp>
#include "gener/BinaryFileArchive.hh"
#include "StOpt/core/grids/FullGrid.h"
#include "StOpt/core/utils/StateWithStocks.h"
#include "StOpt/dp/SimulateStepRegressionCutDist.h"
#include "StOpt/dp/OptimizerDPCutBase.h"
#include "StOpt/dp/SimulatorDPBase.h"

/** \file SimulateRegressionCutDist.h
 * \brief Defines a simple program showing how to use simulations when optimization achieved with transition problems solved with cuts.
 * \par \author Xavier Warin
 */

/** \brief Simulate the optimal strategy, mpi version, Bellman cuts used to allow LP resolution of transition problems
 * \param p\_grid grid used for deterministic state (stocks for example)
 * \param p\_optimize optimizer defining the optimization between two time steps
 * \param p\_funcFinalValue function defining the final value cuts
 * \param p\_pointStock initial point stock
 * \param p\_initialRegime regime at initial date
 * \param p\_fileToDump name associated to dumped bellman values
 * \param p\_bOneFile do we store continuation values in only one file
 */
double SimulateRegressionCutDist(const std::shared_ptr<StOpt::FullGrid> &p\_grid,
    const std::shared_ptr<StOpt::OptimizerDPCutBase> &p\_optimize,
    const std::function<
        Eigen::ArrayXd(const int &, const Eigen::ArrayXd &, const Eigen::ArrayXd &),
        &p\_funcFinalValue,
        const Eigen::ArrayXd &, const Eigen::ArrayXd &)> &p\_pointStock,
    const Eigen::ArrayXd &p\_pointStock);
The version of the previous example using a single archive storing the control/solution is given in [this file](#).

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

In this part we suppose that $X_{x,t}^2$ is controlled but is stochastic.
7.3.1 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)

#ifndef OPTIMIZERNOREGRESSIONDPBASE_H
#define OPTIMIZERNOREGRESSIONDPBASE_H
#include <Eigen/Dense>
#include "StOpt/core/args/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/OptimizerBaseInterp.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.
   // \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
   // \for each dimension return true is the direction can be split
   virtual Eigen::Array<bool, Eigen::Dynamic, 1> getDimensionToSplit() const = 0;

   OptimizerNoRegressionDPBase() {}
   virtual ~OptimizerNoRegressionDPBase() {}
In addition to the methods of “OptimizerBase.h” the following method is needed:

```cpp
virtual std::pair<Eigen::ArrayXXd, Eigen::ArrayXXd> stepOptimize(const Eigen::ArrayXd &p_stock, const std::vector<GridAndRegressedValue> &p_valNext, std::shared_ptr<BaseRegression> p_regressorCur) const = 0;
```

```cpp
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;
```

```cpp
virtual int getNbRegime() const = 0;
```

```cpp
virtual std::shared_ptr<StOpt::SimulatorDPBase> getSimulator() const = 0;
```

```cpp
virtual int getNbControl() const = 0;
```

```cpp
virtual int getSimuFuncSize() const = 0;
```
• 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:

$$dX^{x,t}_2 = X^{x,t}_2 \frac{dX^{x,t}_1}{X^{x,t}_1}$$

where $X^{x,t}_1$ is the risky asset value, the Optimize object is given in this file.

### 7.3.2 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:

```cpp
TransitionStepDPDist (const shared_ptr<FullGrid> &p_pGridCurrent,
const shared_ptr<FullGrid> &p_pGridPrevious,
const std::shared_ptr<BaseRegression> &p_regressorCurrent,
const std::shared_ptr<BaseRegression> &p_regressorPrevious,
const shared_ptr<OptimizerNoRegressionDPBase> &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 ($t_{i+1}$),
- $p_{regressorCurrent}$ 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 14** 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 15  *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

- $p_{\text{phiIn}}$ 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 16 *All “TransitionStepDP” derive from a “TransitionStepBase” object having a pure virtual “OneStep” method.*

A second method is provided permitting to dump the 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 :

- $p_{\text{ar}}$ is the archive where controls and solutions are dumped,
- $p_{\text{name}}$ is a base name used in the archive to store the solution and the control,
- $p_{\text{control}}$ stores the optimal controls calculated at the current time step,
- $p_{\text{bOneFile}}$ 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 17 *As for the “TransitionStepDP”, its “dumpValues” doesn’t need a $p_{\text{bOneFile}}$ 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.
for (int iReg = 0; iReg < p_optimize->getNbRegime(); ++iReg)
  (*valuesStored)[iReg] = regressorPrevious->
geCoordBasisFunctionMultiple(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,
    *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()).mean();
} #endif

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

7.3.3 The framework in simulation

Not special framework is available in simulation. Use the function “SimulateStepRegressionControl” or “SimulateStepRegressionControlDist” described in the section 7.2.3.
Chapter 8

The Python API

8.1 Mapping to the framework

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

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, 2018 EDF
# All Rights Reserved
# This code is published under the GNU Lesser General Public License (GNU LGPL)
import StOptReg
import StOptGeners
import TransitionStepRegressionDP as trans
import FinalStepRegressionDP as final
def DynamicProgrammingByRegression(p_grid, p_optimize, p_regressor,
    p_funcFinalValue, p_pointStock, p_initialRegime, p_fileToDump, key1="

126
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 "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 swing 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:
import math
import numpy as np
import unittest
import StOptGrids
import StOptReg
import StOptGlobal
import Utils
import Simulators as sim
import Optimizers as opt

# unit test for global shape


class OptimizerConstruction(unittest.TestCase):
    def test(self):
        try:
            import_module('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.), dtype=np.float)
            # 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, len(dates) - 1, nbSimul, False)
            strike = 1.
            # Pay off
            payoff = Utils.BasketCall(strike)
            # optimizer
            # simObj = opt.OptimizerSwingBlackScholes(payoff, N)
            # link simulator to optimizer
            simObj.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)
gridArrival = StOptGrids.RegularSpaceGrid(lowValues, step, nbStep)
gridStart = StOptGrids.RegularSpaceGrid(lowValues, step, nbStep -1)

# payoff function for swing

# final step

# on step backward and get asset
asset = bsSim.getParticles()
fin = StOptGlobal.FinalStepRegressionDP(gridArrival,1)
values = fin.set(payoff,asset)

# transition time step

# simulate time step

nbSimul = 10
vecOfStates = []
for i in np.arange(nbSimul):
    vecOfStates.append(StOptGlobal.StateWithStocks(1, np.array([0.], dtype=0), np.zeros(1)))
arRead = StOptGlobal.BinaryFileArchive("Archive","r")
phi = np.zeros((1,nbSimul))
NewState = VecOfStateNext = simStep.oneStep(vecOfStates,phi)
print("New vector of state (different of C++)")
print("New vector of state", NewState[0])
for i in np.arange(len(NewState[0])):
Its declination in term of a time nest for optimization is given below (please notice that the “TransitionStepRegressionDP” object is the result of the mapping between python and c++ and given in the “StOptGlobal” module)

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

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 = StOptGeners.BinaryFileArchive(p_fileToDump, "w")
    nameAr = "Continuation"
    nsteps = simulator.getNbStep()
    # iterate on time steps
    for iStep in range(nsteps):
        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, nsteps - 1 -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 using the control previously calculated in optimization can be given as an example 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 StOptGeners
import StOptGlobal

# 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_funcFinalValue function defining the final value
# 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 SimulateRegressionControl(p_grid, p_optimize, p_funcFinalValue, p_pointStock, p_initialRegime, p_fileToDump):
    simulator = p_optimize.getSimulator()
    nbStep = simulator.getNbStep()
    states = []
    particle0 = simulator.getParticles()[:, 0]

    for i in range(simulator.getNbSimul()) :
        states.append(StOptGlobal.StateWithStocks(p_initialRegime, p_pointStock, particle0))

    ar = StOptGeners.BinaryFileArchive(p_fileToDump, "r")
    # name for continuation object in archive
    nameAr = "Continuation"
    # cost function
    costFunction = np.zeros((p_optimize.getSimuFuncSize(), simulator.getNbSimul()))

    # iterate on time steps
    for istep in range(nbStep) :
        NewState = StOptGlobal.SimulateStepRegressionControl(ar, istep, nameAr, p_grid, p_optimize).oneStep(states, costFunction)
        # different from C++
        states = NewState[0]
        costFunction = NewState[1]
        # new stochastic state
        particles = simulator.stepForwardAndGetParticles()

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

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.

### 8.2 Special python binding

Some specific features have been added to the python interface to increase the flexibility of the library. A special mapping of the geners library has been achieved for some specific needs.

#### 8.2.1 A first binding to use the framework

The “BinaryFileArchive” in the python module “StOptGeners” permits for:

- a grid on point,

- a list of numpy array (dimension 2) of size the number of simulations used by the number of points on the grid (the size of the list corresponds to the number of regimes used in case of a regime switching problem: if one regime, this list contains only one item which is a two dimensional array)

- a regressor

To create a set of regressed values of the numpy arrays values and store them in the archive. This functionality permits to store the continuation values associated to a problem.

The dump method “dumpGridAndRegressedValue” in the “BinaryFileArchive” class permits this dump.

It is also possible to get back the continuation values obtained using the “readGridAndRegressedValue” method.

```python
import numpy as np
import unittest
import random
import StOptGrids
import StOptReg
import StOptGeners
```
# unit test for dumping binary archive of regressed value and Read then

```python
class testBinaryArchiveStOpt(unittest.TestCase):
    def testSimpleStorageAndLecture(self):
        # low value for the mesh
        lowValues = np.array([1., 2., 3.], dtype=np.float)
        # size of the mesh
        step = np.array([0.7, 2.3, 1.9], dtype=np.float)
        # number of step
        nbStep = np.array([4, 5, 6], dtype=np.int32)
        # degree of the polynomial in each direction
        degree = np.array([2, 1, 3], dtype=np.int32)
        # create the Legendre grid
        grid = StOptGrids.RegularLegendreGrid(lowValues, step, nbStep, degree)

        # simulate the perturbed values
        nbSimul = 40000
        np.random.seed(1000)
        x = np.random.uniform(-1., 1., size=(1, nbSimul));
        # mesh
        nbMesh = np.array([16], dtype=np.int32)
        # Create the regressor
        regressor = StOptReg.LocalLinearRegression(False, x, nbMesh)

        # regressed values same values for each point of the grid
        toReal = (2 + x[0, :] + (1 + x[0, :]) * (1 + x[0, :]))
        # function to regress
        toRegress = toReal + 4 * np.random.normal(0., 1, nbSimul)
        # create a matrix (number of stock points by number of simulations)
        toRegressMult = np.zeros(shape=(len(toRegress), grid.getNbPoints()))
        for i in range(toRegressMult.shape[1]):
            toRegressMult[:, i] = toRegress
        # into a list : two times to test 2 regimes
        listToReg = []
        listToReg.append(toRegressMult)
        listToReg.append(toRegressMult)

        # Create the binary archive to dump
        archiveToWrite = StOptGeners.BinaryFileArchive("MyArchive", "w")
        # step 1
        archiveToWrite.dumpGridAndRegressedValue("toStore", 1, listToReg, regressor, grid)
        # step 3
```

133
8.2.2 Binding to store/read a regressor and some two dimensional array

Sometimes, users prefer to avoid the use of the framework provided and prefer to only use the python binding associated to the regression methods. When some regressions are achieved for different set of particules (meaning that one or more functions are regressed), it is possible to dump the regressor used and some values associated to these regressions:

- the “dumpSome2DArray”, “readSome2DArray” permits to dump and read 2 dimensional numpy arrays,
- the “dumpSomeRegressor”, “readSomeRegressor” permits to dump and read a regressor.
X1=np.arange(0.0, 2.2, 0.01)
X2=np.arange(0.0, 1.1, 0.005)
Y=np.zeros((len(X1), len(X2)))
for i in range(len(X1)):
    for j in range(len(X2)):
        if i < len(X1)//2:
            if j < len(X2)//2:
                Y[i, j]=X1[i]+X2[j]
            else:
                Y[i, j]=4*X1[i]+4*X2[j]
        else:
            if j < len(X2)//2:
                Y[i, j]=2*X1[i]+X2[j]
            else:
                Y[i, j]=2*X1[i]+3*X2[j]
XX1, XX2 = np.meshgrid(X1, X2)
Y=Y.T
r, c = XX1.shape
X = np.reshape(XX1, (r*c, 1))[:, 0]
I = np.reshape(XX2, (r*c, 1))[:, 0]
Y = np.reshape(Y, (r*c, 1))[:, 0]
xMatrix = np.zeros((2, len(X)))
xMatrix[0, :] = X
xMatrix[1, :] = I
return xMatrix, Y

# main
xMatrix, y = createData()

# 2 dimensional regression 2 by 2 meshes
nbMesh = np.array([2, 2], dtype=np.int32)
regressor = StOptReg.LocalLinearRegression(False, xMatrix, nbMesh)

# coefficients
coeff = regressor.getCoordBasisFunction(y)
print("Regressed coeff", coeff)

# store them in a matrix
coeffList = np.zeros(shape=(1, 3*2*2))
coeffList[0, :] = coeff.transpose()
# archive write for regressors
archiveWriteForRegressor = StOptGeners.BinaryFileArchive("archive","w")

# store
step = 1
archiveWriteForRegressor.dumpSome2DArray("RegCoeff",step,coeff)
archiveWriteForRegressor.dumpSomeRegressor("Regressor",step,regressor)

# archive Read for regressors
archiveReadForRegressor = StOptGeners.BinaryFileArchive("archive","r")

# get back
values = archiveReadForRegressor.readSome2DArray("RegCoeff",step)
reg = archiveReadForRegressor.readSomeRegressor("Regressor",step)
print("Regressed coeff",values)
print("Reg",reg)
print("newValue",reg.getValue([0.4,0.6],values))
Chapter 9

Using the C++ framework to solve some hedging problem

In this chapter we present an algorithm developed in StOpt to solve some hedging problem supposing that a mean variance criterion is chosen. The methodology follows the article [45].

In this section we suppose that $(\Omega, \mathcal{F}, (\mathcal{F}_t)_{t \in [0,T]})$ is a filtered probability space. We define a set of trading dates $\mathcal{T} = \{t_0 = 0, t_1, ..., t_{N-1}, t_N = T\}$ and we suppose that we are given an asset used as an hedging product $(S_t)_{t_0,t_N}$ which is almost surely positive, square integrable so that $\mathbb{E}[S_T^2] < \infty$ and adapted so that $S_t$ is $\mathcal{F}_t$-measurable for $t = t_0, .., t_N$.

At last we suppose that the risk-free rate is zero so that a bond has always a value of 1.

9.1 The problem

We suppose that we are given a contingent claim $H \in \mathcal{L}^2(P)$ which is supposed to be a $\mathcal{F}_T$-measurable random variable. In the case of a European call option on an asset $S_t$ with strike $K$ and maturity $T$, $H(\omega) = (S_T(\omega) - K)^+$. We are only interested in self financing strategies with limited orders, so with bounded controls. Extending [30], [6] definition, we define:

**Definition 1** A $(\bar{m}, \bar{l})$ self-financing strategy $\mathcal{V} = (\nu_t)_{t_0, ..., N-1}$ is a pair of adapted process $(m_t, l_t)_{t_0, ..., N-1}$ defined for $(\bar{m}, \bar{l}) \in (0, \infty) \times (0, \infty)$ such that:

- $0 \leq m_t \leq \bar{m}, \quad 0 \leq l_t \leq \bar{l} \; \text{P.a.s.} \quad \forall i = 0, ..., N - 1$,
- $m_t l_t = 0 \; \text{P.a.s.} \quad \forall i = 0, ..., N - 1$.

In this definition $m_t$ corresponds to the number of shares sold at date $t$, and $l_t$ the number of share bought at this date.

**Remark 18** The strategies defined in [30] and [6] do not impose that $m_t l_t = 0$ so a buy and sell control could happen at the same given date.

We note $\Theta^{(\bar{m}, \bar{l})}$ the set of $(\bar{m}, \bar{l})$ self-financing strategy and with obvious notations $\nu = (m, l)$ for $\nu \in \Theta^{(\bar{m}, \bar{l})}$. 

137
We consider a model of proportional cost, so that an investor buying a share at date \( t \) will pay \((1 + \lambda)S_t\) and an investor selling this share will only receive \((1 - \lambda)S_t\). Assuming no transaction cost on the last date \( T \), the terminal wealth of an investor with initial wealth \( x \) is given by:

\[
x - \sum_{i=0}^{N-1} (1 + \lambda)l_{ti}S_{ti} + \sum_{i=0}^{N-1} (1 - \lambda)m_{ti}S_{ti} + \sum_{i=0}^{N-1} l_{ti}S_{tN} - \sum_{i=0}^{N-1} m_{ti}S_{tN}.
\]

(9.1)

**Remark 19** The transaction costs on the last date \( T \) are related to the nature of the contract. In the case of a pure financial contract, the investor will sell the asset and then some transaction costs have to be paid to clear the final position. On energy market for example, the contract is often associated to physical delivery and no special fees are to be paid. Besides on these markets, even if the contract is purely financial, futures markets are rather illiquid meaning large transaction costs whereas spot markets are much more liquid so that neglecting final transaction costs is justified.

As in [30] [6], we define the risk minimal strategy minimizing the \( L^2 \) risk of the hedge portfolio:

**Definition 2** A \((\bar{m}, \bar{l})\) self-financing strategy \( \hat{V} = (\bar{m}, \bar{l}) \) is global risk minimizing for the contingent claim \( H \) and the initial capital \( x \) if:

\[
\hat{V} = \arg \min_{\nu = (m, l) \in \Theta (\bar{m}, \bar{l})} \mathbb{E}[(H - x + \sum_{i=0}^{N-1} (1 + \lambda)l_{ti}S_{ti} - \sum_{i=0}^{N-1} (1 - \lambda)m_{ti}S_{ti} - \sum_{i=0}^{N-1} l_{ti}S_{tN} + \sum_{i=0}^{N-1} m_{ti}S_{tN})^2].
\]

(9.2)

### 9.2 Theoretical algorithm

We suppose that the process is Markov and that the payoff \( H \) is a function of the asset value at maturity only to simplify the presentation for the Monte Carlo method proposed. We introduce the global position \( \nu = (\nu_i)_{i=0,...,N-1} \) with:

\[
\nu_i = \sum_{j=0}^{i} (m_{tj} - l_{tj}), \forall i = 0, ..., N - 1.
\]

Using the property that \( m_{ti}l_{ti} = 0, \forall i = 0, ..., N - 1 \), we get \( |\nu_i - \nu_{i-1}| = l_{ti} + m_{ti} \) with the convention that \( \nu_{-1} = 0 \) and

\[
G_T(\nu) = \hat{G}_T(\nu) = x - \sum_{i=0}^{N-1} \lambda |\Delta \nu_{i-1}| S_{ti} + \sum_{i=0}^{N-1} \nu_i \Delta S_t,
\]

where \( \Delta S_t = S_{t_i + 1} - S_{t_i}, \Delta \nu_i = \nu_{i+1} - \nu_i \).

We then introduce \( \hat{\Theta}(\bar{m}, \bar{l}) \) the set of adapted random variable \((\nu_i)_{i=0,...,N-1}\) such that

\[-\bar{m} \leq \nu_i - \nu_{i-1} \leq \bar{l}, \forall i = 1, ..., N - 1.
\]

138
The problem \([9.2]\) can be rewritten as done in \([37]\) finding \(\hat{\nu} = (\hat{\nu}_i)_{i=0,..,N-1}\) satisfying:

\[
\hat{\nu} = \arg\min_{\nu \in \Theta^{(n,l)}} \mathbb{E}[(H - x - \hat{G}_T(\nu))^2].
\] (9.3)

We introduce the spaces \(\kappa_i, i = 0,..,N\) of the \(\mathcal{F}_t\)-measurable and square integrable random variables. We define for \(i \in 0,..,N\) , \(V_i \in \kappa_i\) as:

\[
V_N = H,
\]

\[
V_i = \mathbb{E}[H - \sum_{j=i}^{N-1} \nu_j \Delta S_j + \lambda \sum_{j=i}^{N-1} |\Delta \nu_{j-1}| S_{t_j} |_{\mathcal{F}_t}], \forall i = 0,..,N-1.
\] (9.4)

then

\[
\mathbb{E}[(H - x - \hat{G}_T(\nu))^2] = \mathbb{E}[(V_N - \nu_{N-1} \Delta S_{N-1} + \lambda |\Delta \nu_{N-2}| S_{t_{N-1}} - V_{N-1}) + \sum_{i=2}^{N-1} (V_i + \lambda |\Delta \nu_{i-2}| S_{t_{i-1}} - \nu_{i-1} \Delta S_{i-1} - V_{i-1}) + (V_1 + \lambda |\nu_0| S_{t_0} - \nu_0 \Delta S_0 - x)^2]
\] (9.5)

Due to the definition \([9.4]\), we have that

\[
\mathbb{E}[V_i + \lambda |\Delta \nu_{i-2}| S_{t_{i-1}} - \nu_{i-1} \Delta S_{i-1} - V_{i-1}|_{\mathcal{F}_{t_{i-1}}}] = 0, \forall i = 1,..,N,\]

so that

\[
\mathbb{E}[(H - x - \hat{G}_T(\nu))^2] = \mathbb{E}[\mathbb{E}[(V_N - \nu_{N-1} \Delta S_{N-1} + \lambda |\Delta \nu_{N-2}| S_{t_{N-1}} - V_{N-1})^2 |_{\mathcal{F}_{t_{N-1}}}] + \sum_{i=2}^{N-1} \mathbb{E}[(V_i + \lambda |\Delta \nu_{i-2}| S_{t_{i-1}} - \nu_{i-1} \Delta S_{i-1} - V_{i-1})^2] + (V_1 + \lambda |\nu_0| S_{t_0} - \nu_0 \Delta S_0 - x)^2]
\]

and iterating the process gives

\[
\mathbb{E}[(H - x - \hat{G}_T(\nu))^2] = \mathbb{E}[\mathbb{E}[(V_N - \nu_{N-1} \Delta S_{N-1} + \lambda |\Delta \nu_{N-2}| S_{t_{N-1}} - V_{N-1})^2] + \sum_{i=2}^{N-1} \mathbb{E}[(V_i + \lambda |\Delta \nu_{i-2}| S_{t_{i-1}} - \nu_{i-1} \Delta S_{i-1} - V_{i-1})^2] + \mathbb{E}[(V_1 + \lambda |\nu_0| S_{t_0} - \nu_0 \Delta S_0 - x)^2]
\]

Then we can write the problem \([9.3]\) as:

\[
\hat{\nu} = \arg\min_{\nu \in \Theta^{(n,l)}} \mathbb{E}[(V_N - \nu_{N-1} \Delta S_{N-1} + \lambda |\Delta \nu_{N-2}| S_{t_{N-1}} - V_{N-1})^2] + \sum_{i=2}^{N-1} \mathbb{E}[(V_i + \lambda |\Delta \nu_{i-2}| S_{t_{i-1}} - \nu_{i-1} \Delta S_{i-1} - V_{i-1})^2] + \mathbb{E}[(V_1 + \lambda |\nu_0| S_{t_0} - \nu_0 \Delta S_0 - x)^2]
\] (9.9)
We introduce the space
\[ \rho_i^{m,i} (\eta) = \{(V, \nu) / V, \nu \} \text{ valued } \mathcal{F}_t, -m \leq \nu \leq \eta \leq l, \]
and the space
\[ \rho_i^{m,i} (\eta) = \{(V, \nu_i, \ldots, \nu_N - 1) / V \} \text{ valued, } \mathcal{F}_t, -m \leq \nu_j - \eta \leq l, \]
\[ \text{adapted with } \nu_j, j \geq i \text{ are } \mathbb{R} \text{ valued } \mathcal{F}_t, \]

Similarly to the scheme introduced in [4] to improve the methodology proposed in [18] to solve Backward Stochastic Differential Equations, we can propose an algorithm where the update for \( \tilde{R} \) is taken \( \omega \) by \( \omega \) and stores the optimal trading gain function on each trajectory. Then \( \tilde{R} \) satisfies at date \( t_i \) with an asset value \( S_{t_i} \) for an investment \( \nu_{i-1} \) chosen at date \( t_{i-1} \):
\[
\tilde{R}(t_i, S_{t_i}, \nu_{i-1}) = H - \sum_{j=i}^{N-1} \nu_j \Delta S_j + \lambda \sum_{j=i}^{N-1} |\Delta \nu| S_j,
\]
and at the date \( t_i \) according to equation (9.5) the optimal control is the control \( \nu \) associated to the minimization problem:
\[
\min_{(V, \nu) \in \rho_i^{m,i}(\nu_{i-1})} \mathbb{E}[(\tilde{R}(t_{i+1}, S_{t_{i+1}}, \nu) - \nu \Delta S_i + \lambda |\nu - \nu_{i-1}| S_{t_i} - V)^2 | \mathcal{F}_{t_i}]
\]
This leads to the algorithm [4]

\begin{algorithm}
\begin{enumerate}
\item \( \tilde{R}(t_N, S_{t_{N-1}}(\omega), \nu_{N-1}) = H(\omega), \quad \forall \nu_{N-1} \)
\item \textbf{for} \( i = N, 2 \) \textbf{do}
\item \( (\tilde{V}(t_{i-1}, S_{t_{i-1}}, \nu_{i-2}), \nu_{i-1}) = \arg \min_{(V, \nu) \in \rho_i^{m,i}(\nu_{i-2})} \mathbb{E}[(\tilde{R}(t_i, S_{t_i}, \nu) - \nu \Delta S_i + \lambda |\nu - \nu_{i-1}| S_{t_i} - V)^2 | \mathcal{F}_{t_{i-1}}] \) \quad (9.10)
\item \( \tilde{R}(t_{i-1}, S_{t_{i-1}}, \nu_{i-2}) = \tilde{R}(t_i, S_{t_i}, \nu_{i-1}) - \nu_{i-1} \Delta S_{i-1} + \lambda |\Delta \nu_{i-2}| S_{t_{i-1}} \)
\item \textbf{end for}
\item \( \nu_0 = \arg \min_{\nu \in [-m, l]} \mathbb{E}[(\tilde{R}(t_1, S_{t_1}, \nu) + \lambda |\nu| S_{t_0} - \nu \Delta S_0 - x)^2] \)
\end{enumerate}
\end{algorithm}

\textbf{Remark 20} In order to treat the case of mean variance hedging that consists in finding the optimal strategy and the initial wealth to hedge the contingent claim the last line of algorithm ?? is replaced by
\[
(\tilde{V}, \nu_0) = \arg \min_{(V, \nu)} \mathbb{E}[(V(t_1, S_{t_1}, \nu) + \lambda |\nu| S_{t_0} - \nu \Delta S_0 - V)^2 + \tilde{R}(t_1, S_{t_1}, \nu)],
\]
and last line of algorithm [4] by
\[
(\tilde{V}, \nu_0) = \arg \min_{(V, \nu) \in \mathbb{R} \times [-m, l]} \mathbb{E}[(\tilde{R}(t_1, S_{t_1}, \nu) + \lambda |\nu| S_{t_0} - \nu \Delta S_0 - V)^2].
\]
Remark 21 In the two algorithm presented an argmin has to be achieved: a discretization in \( \nu_{i-2} \) has to be achieved on a grid \([\nu_{i-1} - m, \nu_{i-1} + l]\)

### 9.3 Practical algorithm based on algorithm 4

Starting from the theoretical algorithm 4, we aim at getting an effective implementation based on a representation of the function \( \tilde{V} \) depending on time, \( S_t \) and the position \( \nu_t \) in the hedging assets.

- In order to represent the dependency in the hedging position we introduce a time dependent grid
  \[
  Q_i := (\xi k)_{k=-(i+1),\lfloor \frac{m}{\xi} \rfloor}^{(i+1),\lfloor \frac{l}{\xi} \rfloor}
  \]
  where \( \xi \) is the mesh size associated to the set of grids \( (Q_i)_{i=0,N} \) and, if possible, chosen such that \( \frac{l}{\xi} = \lfloor \frac{l}{\xi} \rfloor \) and \( \frac{m}{\xi} = \lfloor \frac{m}{\xi} \rfloor \).

- To represent the dependency in \( S_t \) we will use a Monte Carlo method using simulated path \((S_t^{(j)})_{i=0,M}\) and calculate the arg min in equation (9.10) using a methodology close to the one described in [8]: suppose that we are given at each date \( t_i \) a partition of \([\min_{j=1,M} S_t^{(j)}, \max_{j=1,M} S_t^{(j)}]\) such that each cell contains the same number of samples. We use the \( Q \) cells \((D_i^q)_{q=1,Q}\) to represent the dependency of \( \tilde{V} \) and \( \nu \) in the \( S_t \) variable.

  On each cell \( q \) we search for \( \hat{V}^q \) a linear approximation of the function \( \tilde{V} \) at a given date \( t_i \) and for a position \( k \xi \) so that \( \hat{V}^q(t_i, S, k) = a^q + b^q S \) is an approximation of \( \tilde{V}(t_i, S, k \xi) \). On the cell \( q \) the optimal numerical hedging command \( \hat{\nu}^q(k) \) for a position \( k \xi \) can be seen as a sensibility so it is natural to search for a constant control per cell \( q \) when the value function is represented as a linear function.

Let us note \((l_i^q(j))_{j=1,M}^q\) the set of all samples belonging to the cell \( q \) at date \( t_i \). On each mesh the optimal control \( \hat{\nu}^q \) is obtained by discretizing the command \( \nu \) on a grid \( \eta = ((k + r) \xi)_{r=-(\lfloor \frac{m}{\xi} \rfloor),\lfloor \frac{\ell}{\xi} \rfloor} \) and by testing the one giving a \( \hat{V}^q \) value minimizing the \( L^2 \) risk so solving equation (9.10).

The algorithm 5 permits to find the optimal \( \nu^{(j)}(k) \) command using algorithm 4 at date \( t_i \), for a hedging position \( k \xi \) and for all the Monte Carlo simulations \( j \). For each command tested on the cell \( q \) the corresponding \( \hat{V}^q \) function is calculated by regression.

**Remark 22** It is possible to use different discretization \( \xi \) to define the set \( \eta \) and the set \( Q_i \). Then an interpolation is needed to get the \( \tilde{R} \) values at a position not belonging to the grid. An example of the use of such an interpolation for gas storage problem tracking the optimal cash flow generated along the Monte Carlo strategies can be found in [42].

**Remark 23** This algorithm permits to add some global constraint on the global liquidity of the hedging asset. This is achieved by restricting the possible hedging positions to a subset of \( Q_i \) at each date \( t_i \).

Then the global discretized version of algorithm 4 is given on algorithm 6 where \( H^{(j)} \) correspond to the \( j \) th Monte Carlo realization of the payoff.
Algorithm 5 Optimize minimal hedging position \((\hat{\nu}_{t_i}^{(l)}(k))_{l=1,\ldots,M}\) at date \(t_{i-1}\)

1: \textbf{procedure} \textsc{OptimalControl}( \(\bar{R}(t_{i+1}, \ldots), k, S_{t_i}, S_{t_{i+1}}\) )

2: \hspace{1em} for \(q = 1, Q\) do

3: \hspace{2em} \(P = \infty\),

4: \hspace{2em} for \(k = -\lfloor \frac{\bar{m}}{\xi} \rfloor, \ldots, \lfloor \frac{\bar{l}}{\xi} \rfloor\) do

5: \hspace{3em} \((a^q, b^q) = \arg \min_{(a,b) \in \mathbb{R}^2} \sum_{j=1}^{M} (\bar{R}(t_{i+1}, S_{t_{i+1}}^{(j)}, (k + l)\xi) -\)

6: \hspace{5em} (k + l)\xi \Delta S_{t_i}^{(j)} + \lambda |l\xi| S_{t_i}^{(j)} - (a + bS_{t_i}^{(j)})^2 \]

7: \hspace{3em} \(\bar{P} = \sum_{j=1}^{M} (\bar{R}(t_{i+1}, S_{t_{i+1}}^{(j)}, (k + l)\xi) - (k + l)\xi \Delta S_{t_i}^{(j)} + \)

8: \hspace{5em} \lambda |l\xi| S_{t_i}^{(j)} - (a^q + b^qS_{t_i}^{(j)})^2 \]

9: \hspace{3em} \textbf{if} \(\bar{P} < P\) \textbf{then}

10: \hspace{4em} \(\nu^q = k\xi, P = \bar{P}\)

11: \hspace{4em} \textbf{end if}

12: \hspace{3em} \textbf{end for}

13: \hspace{1em} \textbf{for} \(j = 1, \frac{M}{Q}\) do

14: \hspace{2em} \(\hat{\nu}_{t_i}^{(l)(j)}(k) = \nu^q\)

15: \hspace{2em} \textbf{end for}

16: \hspace{1em} \textbf{return} \((\hat{\nu}_{t_i}^{(l)}(k))_{l=1,\ldots,M}\)

17: \textbf{end procedure}
Algorithm 6 Global backward resolution algorithm, optimal control and optimal variance calculation

1: for $\nu \in Q_{N-1}$ do
2:     for $j \in [1, M]$ do
3:         $\bar{R}(t_N, S^{(j)}_{t_N}, \nu) = H^{(j)}$
4:     end for
5: end for
6: for $i = N, 2$ do
7:     for $k \xi \in Q_{i-2}$ do
8:         $(\nu_{i-1,1}^{(j)}(k))_{j=1,M} = \text{OptimalControl} (\bar{R}(t_i, ..), k, S_{t_{i-1}}, S_{t_i})$
9:     end for
10:     for $j \in [1, M]$ do
11:         $\bar{R}(t_{i-1}, S^{(j)}_{t_{i-1}}, k \xi) = \bar{R}(t_i, S^{(j)}_{t_i}, \nu_{i-1}^{(j)}(k)) - \nu_{i-1}^{(j)}(k) \Delta S^{(j)}_{i-1} + \lambda |\nu_{i-1}^{(j)}(k) - k \xi| S^{(j)}_{t_{i-1}}$
12:     end for
13: end for
14: $P = \infty,$
15: for $k = -\lfloor \frac{\bar{P}}{T} \rfloor, \ldots, \lfloor \frac{\bar{P}}{T} \rfloor$ do
16:     $\tilde{P} = \sum_{j=1}^{M} (\bar{R}(t_1, S^{(j)}_{t_1}, k \xi) - k \xi \Delta S^{(j)}_0 + \lambda |k \xi S_0 - x|^2$
17:     if $\tilde{P} < P$ then
18:         $\nu_0 = k \xi, P = \tilde{P}$
19:     end if
20: end for
21: $Var = \frac{1}{M} \sum_{j=1}^{M} (\bar{R}(t_1, S^{(j)}_{t_1}, \nu_0) - \nu_0 \Delta S^{(j)}_0 + \lambda |\nu_0| S_0 - x)^2$
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 10

Theoretical background

In this part, we are back to the resolution of equation (2.1).

10.1 Notation and regularity results

We denote by $\wedge$ the minimum and $\vee$ 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 (2.1) for a given $\hat{K}$:

$$\sup_a |g|_1 + |\sigma_a|_1 + |b_a|_1 + |f_a|_1 + |c_a|_1 \leq \hat{K} \quad (10.1)$$

A classical result [22] gives us the existence and uniqueness of the solution in the space of bounded Lipschitz functions:

**Proposition 1** If the coefficients of the equation (2.1) satisfy (10.1), there exists a unique viscosity solution of the equation (2.1) belonging to $C_1(Q)$. If $u_1$ and $u_2$ are respectively sub and super solution of equation (2.1) 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 $(i_1, \ldots, i_d) \in \mathbb{Z}^d$ will correspond to the coordinates of a mesh $M_i$ defining a hyper-cube in dimension $d$. For an interpolation grid $(\xi_i)_{i=0\ldots N} \in [-1, 1]^N$, and for a mesh $\tilde{i}$, the point $y_{\tilde{i}, \tilde{j}}$ with $\tilde{j} = (j_1, \ldots, j_d) \in [0, N]^d$ will have the coordinate $(\Delta x(i_1 + 0.5(1 + \xi_{j_1})), \ldots, \Delta x(i_d + 0.5(1 + \xi_{j_d})))$. We denote $(y_{\tilde{i}, \tilde{j}})$ 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 $x \in \mathbb{R}^d$:

$$\min_{\tilde{i}, \tilde{j}} |x - y_{\tilde{i}, \tilde{j}}| \leq \Delta x. \quad (10.2)$$
10.2 Time discretization for HJB equation

The equation (2.1) is discretized in time by the scheme proposed by Camilli Falcone [13] for a time discretization $h$.

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

with

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

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

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

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 [31].

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 $h$:

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

We first recall the following result:

**Proposition 2** Under the condition on the coefficients given by equation (10.1), the solution $v_h$ of equations (10.3) and (10.4) is uniquely defined and belongs to $C^1(Q)$. We check that if $h \leq (16 \sup_a \{ |\sigma_a|_1^2 + |b_a|_1^2 + 1 \} \wedge 2 \sup_a |c_a|_0)^{-1}$, there exists $C$ such that

$$|v - v_h|_0 \leq C h^{\frac{1}{4}}.$$  (10.5)

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

$$|v_h|_0 \leq C,$$  (10.6)

$$|v_h(t, x) - v_h(t, y)| \leq C |x - y|, \forall (x, y) \in Q^2.$$  (10.7)

10.3 Space interpolation

The space resolution of equation (10.3) is achieved on a grid. The $\phi^+$ and $\phi^-$ have to be computed by the use of an interpolator $I$ such that:

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

$$v_h(t, \phi_{a,h,i}^-(t, x)) \approx I(v_h(t, .)(\phi_{a,h,i}^-(t, x))).$$
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 [44], [43]).
Chapter 11

C++ API

In order to achieve the interpolation and calculate the semi Lagrangian value

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

a first object “SemiLagrangEspCond” is available:

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

#ifndef SEMILAGRANGESPCOND_H
#define SEMILAGRENCESPCOND_H

#include <Eigen/Dense>
#include <map>
#include <array>
#include <vector>
#include "StOpt/core/util/constant.h"
#include "StOpt/core/grids/InterpolatorSpectral.h"

namespace StOpt
{

/// \file SemiLagrangEspCond.h
/// \brief Semi Lagrangian method for process \f$ d x_t = b dt + \sigma dW_t \f$
/// \f$ \times \mathbf{R}^m \f$ matrix and \f$ W_t \f$ with values in \f$ \mathbf{R}^m \f$

class SemiLagrangEspCond
{
    //\brief interpolator
    std::shared_ptr<InterpolatorSpectral> m_interpolator;
}

```

149
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 3). 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 [44]).

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 $p_x + p_{bh} \pm p_{\text{sig}}\sqrt{h}$. 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
#include <vector>
#include <Eigen/Dense>
#include "StOpt/core/grids/SpaceGrid.h"
#include "StOpt/core/grids/FullGrid.h"
#include "StOpt/core/grids/InterpolatorSpectral.h"
#include "StOpt/semilagrangien/SemiLagrangEspCond.h"

/** \file OptimizerSLBase.h
 * \brief Define an abstract class for Dynamic Programming problems
 * \author Xavier Warin
 */

namespace StOpt
{

/// \class OptimizerSLBase OptimizerSLBase.h
/// Base class for optimizer for resolution by semi Lagrangian methods of HJB equations
class OptimizerSLBase
{

public :

OptimizerSLBase() {}

virtual ~OptimizerSLBase() {}

/// \brief define 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
/// 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_point coordinates of the point to treat
/// \param p_semiLag semi Lagrangian operator for each regime for
/// solution at the previous step
/// \param p_time current date
/// \param p_phiInPt value of the function at the previous time step at
/// p_point for each regime
/// \return a pair:
/// - first an array of the solution (for each regime)
/// - second an array of the optimal controls (for each control)
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;

/// \brief defines a step in simulation
/// \param p_gridNext grid at the next step
/// \param p_semiLag semi Lagrangian operator at the current step
/// in each regime
/// \param p_state state array (can be modified)
/// \param p_iReg regime number
/// \param p_gaussian unitary Gaussian realization
/// \param p_phiInPt value of the function at the next time step at
/// p_point for each regime
/// \param p_phiInOut defines the value functions (modified) to
/// follow
virtual void stepSimulate(const SpaceGrid &p_gridNext,
    const std::vector< std::shared_ptr<StOpt::
    SemiLagrangEspCond> > &p_semiLag,
    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;

/// \brief defines a step in simulation using the control calculated in
/// optimization
/// \param p_gridNext grid at the next step
/// \param p_controlInterp the optimal controls interpolator
/// \param p_state state array (can be modified)
/// \param p_iReg regime number
/// \param p_gaussian unitary Gaussian realization
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 7.

- 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”).
And example of the derivation of such an optimizer for a simple stochastic target problem (described in paragraph 5.3.4 in [44]) is given below:

```cpp
#include <iostream>
#include "StOpt/core/utils/constant.h"
#include "test/c++/tools/semilagrange/optimizers/SLCase3.h"

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

OptimizerSLCase3::OptimizerSLCase3(const double &p_mu, const double &p_sig,  
    const double &p_dt, const double &p_alphaMax, const double &p_stepAlpha):
    m_dt(p_dt), m_mu(p_mu), m_sig(p_sig), m_alphaMax(p_alphaMax), m_stepAlpha(p_stepAlpha) {}

vector< array<double, 2> > OptimizerSLCase3::getCones(const vector< array<double, 2> > &p_xInit) const
{
    vector< array<double, 2> > xReached(1);
    xReached[0][0] = p_xInit[0][0] - m_alphaMax * m_mu / m_sig * m_dt -  
    m_alphaMax * sqrt(m_dt); // trend
    xReached[0][1] = p_xInit[0][1] + m_alphaMax * sqrt(m_dt); // volatility with one Brownian
    return xReached;
}

pair<ArrayXd, ArrayXd> OptimizerSLCase3::stepOptimize(const ArrayXd &p_point,  
    const vector< shared_ptr<SemiLagrangianCond> > &p_semiLag, const  
    double &, const Eigen::ArrayXd &) 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 < 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(p_point, b, sig, m_dt); // test the control
        if (lagrang.second) 
        {  
            if (lagrang.first < vMin)  
            {  
                vMin = lagrang.first;  
                solutionAndControl.second(0) = alpha;
            }
        }
    }
    solutionAndControl.first(0) = vMin;
}"
return solutionAndControl;

void OptimizerSLCase3::stepSimulate(const StOpt::SpaceGrid &p_gridNext,  
    const 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 vector<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);
11.1 PDE resolution

Once the problem is described, a time recursion can be achieved using the “Transition-StepSemilagrangian” object in a sequential resolution of the problem. This object permits to solve the problem on one time step.

```cpp
namespace StOpt
{
    // \class TransitionStepSemilagrangian TransitionStepSemilagrangian.h
    // \brief One step of semi Lagrangian scheme
    class TransitionStepSemilagrangian : public TransitionStepSemilagrangianBase
    {
        private :
            std::shared_ptr<SpaceGrid> m_gridCurrent ; ///< global grid at current time step
            std::shared_ptr<SpaceGrid> m_gridPrevious ; ///< global grid at previous time step
            std::shared_ptr<OptimizerSLBase> m_optimize ; ///< optimizer solving the problem for one point and one step
        public :
            /// \brief Constructor
```
TransitionStepSemilagrang(const std::shared_ptr<SpaceGrid>& p_gridCurrent,
   const std::shared_ptr<SpaceGrid>& p_gridPrevious,
   const std::shared_ptr<OptimizerSLBase>& p_optimize);

/// \brief One time step for resolution
/// \param p_phiIn for each regime the function value (on the grid)
/// \param p_time current date
/// \param p_boundaryFunc Function at the boundary to impose Dirichlet conditions (depending on regime and position)
/// \return solution obtained after one step of dynamic programming and the optimal control
std::pair<std::vector<std::shared_ptr<Eigen::ArrayXd>>, std::vector<std::shared_ptr<Eigen::ArrayXd>>> oneStep(const std::vector<std::shared_ptr<Eigen::ArrayXd>>& p_phiIn, const double &p_time, const std::function<double(const int&, const Eigen::ArrayXd&)> &p_boundaryFunc);

/// \brief Permits to dump continuation values on archive
/// \param p_ar archive to dump in
/// \param p_name name used for object
/// \param p_iStep Step number or identifier for time step
/// \param p_phiIn for each regime the function value
/// \param p_control for each control, the optimal value
void dumpValues(std::shared_ptr<gs::BinaryFileArchive> p_ar, const std::string &p_name, const int &p_iStep, const std::vector<std::shared_ptr<Eigen::ArrayXd>>& p_phiIn, const std::vector<std::shared_ptr<Eigen::ArrayXd>>& p_control);
Table 11.1: 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>&quot;TransitionStepSemilagrang&quot;</td>
<td>&quot;TransitionStepSemilagrang&quot;</td>
</tr>
<tr>
<td>Parallelization on calculations threads and MPI</td>
<td>&quot;TransitionStepSemilagrang&quot;</td>
<td>&quot;TransitionStepSemilagrang&quot;</td>
</tr>
<tr>
<td>Distribution of calculations and data (MPI)</td>
<td>&quot;TransitionStepSemilagrangDist&quot;</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 \( p_{\text{phiIn}} \) at the step \( p_{\text{istep}} + 1 \) and the optimal control at step \( p_{\text{istep}} \) in an archive \( p_{\text{ar}} \).

A version using the distribution of the data and calculations can be found in the \( \text{TransitionStepSemilagrangDist} \) object. An example of a time recursion in sequential can be found in the \( \text{semiLagrangianTime} \) function and an example with distribution can be found in the \( \text{semiLagrangianTimeDist} \) 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.

### 11.2 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

  ```c++
  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
  - \( p_{\text{ar}} \) is the binary archive where the continuation values are stored,
  - \( p_{\text{iStep}} \) 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),
  - \( p_{\text{name}} \) is the base name to search in the archive,
  - \( p_{\text{GridNext}} \) is the grid at the next time step (\( p_{\text{iStep}} + 1 \)),

  159
- \( p\text{\_Optimize} \) 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 24** 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 25** 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)

#include <boost/random.hpp>
#include <memory>
#include <Eigen/Dense>
#include "generes/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::ArrayXXd 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)
      for (int is = 0; is < gaussian.rows(); ++is)
        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();
}
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

| 1 | SimulateStepSemilagrangControlDist(gs::BinaryFileArchive &p_ar, |
| 2 | const int &p_iStep, const std::string &p_name, |
| 3 | const std::shared_ptr<FullGrid> &p_gridCur, |
| 4 | const std::shared_ptr<FullGrid> &p_gridNext, |
| 5 | const std::shared_ptr<OptimizerSLBase> &p_pOptimize, |
| 6 | 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_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 26** 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”

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

where:

- *p_gaussian* is a two dimensional array (number of Brownian in the modelization by the number of Monte Carlo simulations).
- \( p_{\text{statevector}} \) stores the continuous state (continuous state size by number of simulations).
- \( p_{i\text{Reg}} \) for each simulation gives 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.

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)
#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 ;
  // sore 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::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 27 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 11.2: Which simulation object to use depending on the TransitionStepSemilagrang object used.

<table>
<thead>
<tr>
<th></th>
<th>“TransitionStepSemilagrang”</th>
<th>“TransitionStepSemilagrangDist” “bOneFile”=True</th>
<th>“TransitionStepSemilagrangDist” “bOneFile”= False</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.

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
  \[
  \Pi(D_t, L_t) = p_t D_t - (D_t - L_t)^+,
  \]

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

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

\[
\begin{align*}
\partial_t v + \alpha(m - D)\partial_D v + \frac{1}{2} \sigma^2 \partial^2_D v + (D - L)^+ \partial_Q v + \Pi(D, L) \\
+ sL^{1-a} - y(D - L)^+ + \sup_{l \in L} \{l \partial_L v - \tilde{c}(l, L)\} = 0 \\
v_T = 0
\end{align*}
\] (11.1)
and the carbon price \( y(t, D_t, Q_t, L_t) \) is given by:

\[
\begin{aligned}
\partial_t y + \alpha(m - D)\partial_D y + \frac{1}{2}\sigma^2\partial^2_{DD} y + (D - L)^+\partial_Q y + l^*\partial_L y &= 0 \\
y_T &= \lambda(Q_t \geq K)
\end{aligned}
\] (11.2)

and \( l^* \) is the optimal control in equation (11.1). 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

\[
\begin{aligned}
v(T - \delta t, D, Q, L) &= \sup_l (\Pi(D, L) + sL^{1-\alpha} - y_{T-\delta t}(D - L)^+ - \tilde{c}(l, L))\delta t + \\
&\quad \mathbb{E}_{T-\delta t}(V(T, D^{T-\delta t,D}, Q + (D - L)^+\delta t, L + l\delta t)) \\
Y(T - \delta t, D, Q, L) &= \mathbb{E}_{T-\delta t}(Y(T, D^{T-\delta t,D}, Q + (D - L)^+\delta t, L + l^*\delta t))
\end{aligned}
\] (11.3) (11.4)

The previous equations (11.3) and (11.4) 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.

### 11.3 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:

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

// for parallelism
std::vector<std::array<double, 2>> OptimizeDPmissive::getCone(const std::vector<
    std::array<double, 2>> &p_xInit) const
{
    vector< array<double, 2> > xReached(2);
    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< ArrayXXd, ArrayXXd> OptimizeDPmissive::stepOptimize(const std::shared_ptr<
    StOpt::SpaceGrid> &p_grid, const ArrayXd &p_stock, 
    const std::vector< ContinuationValue> &p_condEsp, 
    const std::vector< std::shared_ptr< ArrayXXd >>> &p)
{
    std::pair< ArrayXXd, ArrayXXd> solutionAndControl;
    // to store final solution (here two regimes)
    solutionAndControl.first = ArrayXXd::Constant(m_simulator->getNbSimul(), 
        2, StOpt::infty);
    solutionAndControl.second = ArrayXXd::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.) * 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].getAsSimulation(is, *interpolator);
            // local gain
            double gainLoc = (gain(is) - yLoc * std::max(demand(is) -
                p_stock(1), 0.) - m_cBar(1, p_stock(1))) * m_dt;
            // gain + conditional expectation of future gains
            double condExp = gainLoc + p_condEsp[0].getAsSimulation(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::
    infnty, 10))
{
    // fix boundary condition
    solutionAndControl.first(is, 0) = timeToMat * (m_PI(demand(is),
        p_stock(1)) + m_s * pow(p_stock(1), 1. - m_alpha) - m_lambda *
            std::max(demand(is) - p_stock(1), 0.));
    solutionAndControl.first(is, 1) = m_lambda; // Q est maximal !
    solutionAndControl.second(is, 0) = 0. ; // fix control to zero
}
return solutionAndControl;

// one step in simulation for current simulation
void OptimizeDPEmissive::stepSimulate(const std::shared_ptr<StOpt::SpaceGrid>& p_grid, const std::vector<StOpt::GridAndRegressedValue>& p_continuation, 
                                      StOpt::StateWithStocks &p_state, 
                                      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(l, 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
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.

11.4 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_pi, const double &p_cBar, const std::function<double(double, double)>& p_PI, const std::function<double(double, double)>& p_sBar, const double &p_s, const double &p_dt, const double &p_lMax, const double &p_lStep, const std::vector<std::array<double, 2>>& p_extrem) :
    m_alpha(p_alpha), m_m(p_m), 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(3);
    xReached[0][0] = p_xInit[0][0] + m_alpha * (m_m - m_extrem[0][1]) * m_dt - m_s * 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_s * 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 increase 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
            }
        }
    }
}
```cpp
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<
                                           StOpt::SemiLagrangEspCond>
                                           & 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) = mSig;
    double vOpt = -StOpt::infty;
    double lOpt = 0;
    double yOpt = 0;
    ArrayXd b(3);
    b(0) = mAlpha * (mm - 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 = mPI(p_state(0), p_state(2)) + ms * pow(p_state(2),
                                         1. - mAlpha); // gain for production and subvention
    for (int iAl = 0; iAl < mLMax / mLStep; ++iAl) // recalculate the optimal control
    {
        double l = iAl * mLStep;
        b(2) = l;
        pair<double, bool> lagrangY = p_semiLag[1]->oneStep(p_state, b, sig,
                                           mLdt); // calculate y for this control
        if (lagrangY.second)
        {
            pair<double, bool> lagrang = p_semiLag[0]->oneStep(p_state, b,
                                           sig, mLdt); // calculate the function value v
            // gain function
            double gain = mLdt * (gainFirst - lagrangY.first * b(1) - mcBar
                                         (1, p_state(2)));
            double arbitrage = gain + lagrang.first;
            if (arbitrage > vOpt) // arbitrage
            {
                vOpt = arbitrage; // upgrade solution
                yOpt = lagrangY.first; // upgrade y value
            }
        }
    }
}
```
lOpt = 1; // upgrade optimal control
}
}

// gain function
p_phiInOut(0) += m_dt * (gainFirst - yOpt * b(1) - m_cBar(lOpt, state(2)));
 p_phiInOut(1) = yOpt; // store y value
// update state
state(0) += m_alpha * (m_m - p_state(0)) * m_dt + m_sig * p_gaussian(0) *
sqrt(m_dt); // demand (no control)
state(1) += b(1) * m_dt; //Q
state(2) += lOpt * m_dt; //L
// truncate if necessary to stay inside domain.
p_gridNext.truncatePoint(state);
p_state = state;

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 12

SDDP algorithm

12.1 Some general points about SDDP

SDDP is an approximate dynamic programming algorithm developed by Pereira and Pinto in 1991 [33].

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:

\[
\begin{align*}
    x_0 &\rightarrow \text{observation of } \omega_1 \rightarrow \text{decision of } x_1 \\
    &\rightarrow \text{decision of } x_{T-1} \rightarrow \text{observation of } \omega_T \rightarrow \text{decision of } x_T
\end{align*}
\]
A rigorous formulation of the multistage stochastic linear program to solve is the following:

\[
V^* = \min_{A_0 x_0 = \omega_0, x_1 \geq 0} c_0^T x_0 + \mathbb{E} \left[ \min_{x_1 \geq 0} c_1^T x_1 + \mathbb{E} \left[ \min_{x_T \geq 0} c_T^T x_T \right] \right] \tag{12.1}
\]

The deterministic equivalent of this problem (12.1) 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 \( (\Omega_t)_{t \in \{0, \ldots, T\}} \) are of reasonable size.

**Dynamic programming principle**

Dynamic programming involves splitting up the problem (12.1) in a series of sub-problem bounded together by a state variable. The aim is to compute backwards the functions \( Q_t \) and \( Q_{t+1} \). They fulfill the following equations:

\[
\begin{align*}
[LP_t] & \quad \begin{cases} 
Q_t(x_{t-1}, \omega_t) = \min c_t^T x_t + Q_{t+1}(x_t) \\
\text{s.c.} \quad A_t x_t = \omega_t - E_t x_{t-1}, \quad [\pi_t(\omega_t)] \\
x_t \geq 0 
\end{cases} \\
\mathcal{D}_t(x_{t-1}) & = \mathbb{E}[Q_t(x_{t-1}, \omega_t)] 
\end{align*} \tag{12.2}
\]

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 \( \omega_t \). \( \mathcal{D}_t(x_{t-1}) \) is the expected value of the future cost knowing the state. The dynamic programming principle insures that \( V^* = \mathcal{D}_1(x_0) \).

Given \( \mathcal{D}_T(\cdot) \), the successive computations are achieved backwards switching between the resolution of the linear sub-problem (12.2) and the computation of (12.3).

The implementation of dynamic programming involves approximating successively the two value functions with equations (12.2) - (12.3) 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"). Besides a linear approximation of \( \mathcal{D}_{t+1}(x_t) \) must be available in order to cast the transition problem into a LP.

**SDDP algorithm**

SDDP is a method used to solve stochastic multi-stage problem described in [33]. SDDP is based on Benders decomposition described in [5]. Please note that SDDP was developed in order to solve hydro thermal scheduling problem.

SDDP limits the curse of dimensionality by avoiding \( a \ priori \) 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*.

### 12.2 A method, different algorithms

The method implemented in this library is based on the different situations shown in a technical report of PSR program [32] 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.
- \( z \) denotes the optimal cost obtained in backward pass.
- \( \beta^j \) denotes the slope of the \( j^{th} \) Benders cut.
- \( \alpha^j \) denotes the intercept of the \( j^{th} \) Benders cut.

**12.2.1 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.
- \( n_{\text{iter Max}} \in \mathbb{N} \), 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.

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. At each time step \( G \) cuts are added using the \( G \) visited states \((x^g_t)_{g=1,...,G}\) obtained during the forwards pass.

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 \( n_{\text{iter Max}} \) (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:

\[
\psi_{n_{\text{step}}} = \left| \frac{\bar{z}_{n_{\text{step}i}} - \bar{z}_{n_{\text{step} i}}}{\bar{z}_{n_{\text{step} i}}} \right|, \quad \text{with } i \in \mathbb{N}
\]  
(12.10)
Algorithm 7 Run of forward pass (\(n^{th}\) iteration)

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

2: for \(g \in \Omega^g\) do

3: Solve the following linear sub-problem.

\[
\begin{align*}
\text{\([AP^0_{\theta}]\)} & \\
Q_0 \cdot \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^j_1)^T x_0 \geq \alpha^j_1, \quad j \in \{1, \ldots, G, \ldots, nG\}
\end{align*}
\]  

(12.4)

4: Store the primal solution \((x^g_0)\) of the problem \([AP^0_{\theta, g}]\).

5: for \(t \in \{1, \ldots, T\}\) do

6: Solve the following linear sub-problem.

\[
\begin{align*}
\text{\([AP^n_{t, g}]\)} & \\
Q^g_t(x^g_{t-1}, \omega^g_t) = & \min_{x_t, \theta_{t+1}} & c_t^T x_t + \theta_{t+1} \\
\text{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^j_{t+1})^T x_t \geq \alpha^j_{t+1}, \quad j \in \{1, \ldots, G, \ldots, nG\}
\end{align*}
\]  

(12.5)

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

8: end for

9: Compute the cost for scenario \(g\), at iteration \(n\): \(z^g_n = \sum_{t=0}^T c_t x^g_t\)

10: end for

11: Compute the total cost in forward pass at iteration \(n\): \(z_n = \frac{1}{G} \sum_{g=1}^G z^g_n\)

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.

12.2.2 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 (12.2 - 12.3) 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:

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

(12.11)

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.

$$x_t^{\text{dep}} = (x_t, \omega_t^{\text{dep}})$$

The Bellman function satisfies from now on the following two-stages linear problem at time $t$:

$$[LP_t'] \begin{cases} 
Q_t(x_{t-1}, \omega_{t-1}^{\text{dep}}, \omega_t) = \min c_t^T x_t + \mathcal{Q}_{t+1}(x_t, \omega_t^{\text{dep}}) \\
u.c. A_t x_t = P_t^{\omega_t^{\text{dep}}} - E_t x_{t-1}, [\pi_t(\omega_t)] \\
x_t \geq 0
\end{cases}$$

(12.12)

with $P$ the matrix such that $\omega_t = P \omega_t^{\text{dep}}$.

The variable $\omega_t^{\text{dep}}$ is a random process. Thus the above problem is solved using specific values $\omega_t^l$ of this variable. To get them we apply a Markov process that is we simulate different values of the white noise $\epsilon_t^l$.

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 $\omega_t^{\text{dep}}$. The computation of Benders cuts is then a bit different:

$$\frac{\partial Q_t(x_{t-1}, \omega_{t-1}^{\text{dep}}, \omega_t)}{\partial \omega_{t-1}^{\text{dep}}} = \frac{\partial Q_t(x_{t-1}, \omega_{t-1}^{\text{dep}}, \omega_t)}{\partial \omega_t^{\text{dep}}} \frac{\partial \omega_t^{\text{dep}}}{\partial \omega_{t-1}^{\text{dep}}} = \pi_t(\omega_t)^T P \psi_1 \frac{\sigma_{\omega,t}}{\sigma_{\omega,t-1}}$$

(12.13)

Backward pass has to be modified in the following manner. Some new computation steps have to be taken into account.

### 12.2.3 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 left-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 [32] 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 [32] and in [17].

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

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

This model makes easier the implementation of the SDP. But it implies discretization mistakes that are hard to quantify. It is also tough to discretize with efficiency a random process.

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. These conditional cuts are computed using the methods in section 4: two methods are available in the library. Both use adaptive support. The first uses a constant per cell approximation while the second uses a linear per cell approximation.

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 4: 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 $s^u_t$ at each stage $t$.
- $\tilde{I}$ is a partition of the space of the non-convex random quantities $S$.
  \[ \tilde{I} = \{ I = (i_1, ..., i_d) : i_1 \in \{1, ..., I_1\}, ..., i_d \in \{1, ..., I_d\} \} \]
- $\{D_I\}_{I \in \tilde{I}}$ 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.

12.3 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.

12.3.1 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:
  - `updateDates` : set new set of dates (“$t$”, “$t+dt$”)
  - `oneStepForward` : solves the different transition linear problem during the forward pass for a particle, a random vector and an initial state. :
    * the state $(x_t, w^{dep}_t)$ is given as input of the function,
    * the $s_t$ values are restored by the simulator,
the LP is solved between dates \( t \) and \( t + dt \) for the given \( s_t \) and the constraints due to \( w_{t}^{\text{dep}} \) (demand, flow constraints) and permits to get the optimal \( x_{t+dt} \).

Using iid sampling, \( w_{t+dt}^{\text{dep}} \) is estimated

return \((x_{t+dt},w_{t+dt}^{\text{dep}})\) as the following state and \((x_{t+dt},w_{t+dt}^{\text{dep}})\) that will be used as the state to visit during next backward resolution.

- oneStepBackward : solves the different transition linear problem during the backward pass for a particle, a random vector and an initial state.

The state \((x_{t+dt},w_{t}^{\text{dep}})\) is given as input if \( t \geq 0 \) otherwise input is \((x_{0},w_{0}^{\text{dep}})\)

If \( t \geq 0 \), sample to calculate \( w_{t+dt}^{\text{dep}} \) in order to get the state \((x_{t+dt},w_{t+dt}^{\text{dep}})\) at the beginning of the period of resolution of the LP.

Solve the LP from date \( t + dt \) to next date \( t + 2dt \) (if equally spaced periods).

Return the function value and the dual that will be used for cuts estimations.

- oneAdmissibleState : returns an admissible state at time \( t \) (respect only the constraints)

**TransitionOptimizer** should derive from the **OptimizerSDDPBase** class defined below.

```cpp
/** \file OptimizerSDDPBase.h
 * \brief Define an abstract class for Stochastic Dual Dynamic Programming problems
 * \author Xavier Warin */

class OptimizerSDDPBase
{
    public:
        OptimizerSDDPBase() {}
}
```

185
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 independent 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 particle 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 constains the current state
/// \param p_stateToStore for backward resolution we need to store $f(S_{t},A_{\{t-1\}};D_{\{t-1\}})$ where p_state in output is $f(S_{t},A_{\{t\}};D_{\{t\}})$
/// \param p_isimu number of teh simulation used
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_isimu) const = 0 ;

/// \brief update the optimizer for new date
/// − In Backward mode, LP resolution achieved at date p_dateNext
/// − starting with uncertainties given at date p_date and evolving to give uncertainty at date p_dateNext,
/// − In Forward mode, LP resolution achieved at date p_date,
/// and uncertainties evolve till date p_dateNext
///
/// 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(const double & p_date) = 0 ;

/// \brief get back state size
virtual int getStateSize() const = 0;
- 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.

```cpp
// 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 Constructor
    SimulatorSDDPBase() {}

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

    /// \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;

};
#endif /* OPTIMIZERSDDPBASE_H */
```
virtual void updateDates(const double &p_date) = 0;

virtual Eigen::VectorXd getOneParticle(const int &p_isim) const = 0;

definition of the class

virtual Eigen::MatrixXd getParticles() const = 0;

virtual void resetTime() = 0;

virtual void updateSimulationNumberAndResetTime(const int &p_nbSimul) = 0;

#endif /* SIMULATORSDDPBASE_H */
12.3.2 Architecture

The SDDP handling part of the library is built following the scheme described below.

In the following pseudo-code you have to keep in mind that some small shortcuts have been used in view of making the reading reader-friendly (for example linear sub-problem in the initial case \((t = 0)\) should be a bit different than the one in other time-steps, \texttt{forwardSDDP()}, \texttt{backwardSDDP()}, \texttt{backwardforwardSDDP()} inputs have been omitted for simplification). A more rigorous theoretical explanation is available in the previous part.

Three colors have been used: blue parts correspond to the use of functions implemented in the \texttt{TransitionOptimizer} class, red parts correspond to the use of \texttt{Simulator(Sim or Opt)} functions while grey parts correspond to generic functions totally handled by the library. To be more accurate, what you have to implement as a StOpt user is only the \texttt{TransitionOptimizer} and the \texttt{Simulator} (blue and red part), other functions and described loops are already implemented and managed by the library.

12.3.3 Implement your problem

In the following section, some tips and explanations will be given in view of helping you implementing your problem in the library. It is advised to have a look at the examples provided by the library. It will give you a better understanding of what is needed to compute the SDDP method through StOpt (folder test/c++/tools/sddp for the optimizer examples, test/c++/tools/simulators for the simulators one, and test/c++/functional for the main instances).

Implement your own TransitionOptimizer class

As described above, your \texttt{TransitionOptimizer} class should be specific to your problem (it’s given as an argument of the \texttt{backwardforwardSDDP} function). Hence, you have to implement it by yourself following certain constraints in view of making it fitting the library requirements.

First, make it sure that your \texttt{TransitionOptimizer} class heritates from the class \texttt{OptimizerSDDPBase}. You will then have to implement the following functions.

- The \texttt{updateDates} function allows to update the data stored by the optimizer, fitting the times indicated as argument.

  

  

  

  

  

  

  

  

  

  

  

  

  

  

  

  

  

  

  

  

  

  

  

  

  

  

  

  

  

  

  

  

  

  

  

  

  

  

  

  

If your transition problem depends on the time, you should for instance store those arguments value. Following your needs you could also update data such as the average demand at current and at next time step in a gas storage problem. The \texttt{pDateNext} argument is used as the current time step in the backward pass. Hence, you should store the values for both the arguments current and next time step.
- The **oneAdmissibleState** function gives an admissible state (that means a state respecting all the constraints) for the time step given as an argument:

  ```cpp
  /// \return an admissible state
  ```

- The **oneStepBackward** function allows to compute one step of the backward pass:

  ```cpp
  virtual Eigen::ArrayXd oneStepBackward(const std::unique_ptr<StOpt::SDDPCutBase> &p_linCut,
                                          const std::tuple<Eigen::ArrayXd, int, int> &p_aState,
                                          const Eigen::ArrayXd &p_particle,
                                          const int &p_isample) const = 0;
  ```

  The first argument is the cuts already selected for the current time step. It is easy to handle them, just use the `getcutsAssociatedToAParticle` function as described in the examples that you can find in the test folder (*OptimizeReservoirWithInflowsSDDP.h* without regression or *OptimizeGasStorageSDDP.h* with regression). You will then have the needed cuts as an array `cuts` that you can link to the values described in the theoretical part at the time step `t` by:

  \[
  cuts(0,j) = \alpha_{t+1}^j, \quad cuts(i,j) = \beta_{t-1,t+1}^j, \quad j \in \{1, ..., G, ..., (n+1)G\}, \quad i \in \{1, ..., nbState\}.
  \]

  You will have to add the cuts to your constraints by yourself, using this array and your solver functionalities.

  Moreover, as an argument you have the object containing the state at the beginning of the time step `pastate` (**have in mind that this argument is given as an Eigen array**), `p_particle` contains the random quantities in which the regression over the expectation of the value function will be based (the computational cost is high so have a look at the theoretical part to know when you really need to use this), finally the last argument is an integer giving in which scenario index the resolution will be done.

  The function returns a 1-dimensional array of size `nbState + 1` containing as a first argument the objective function, and then for `i \in \{1, ..., nbState\}` it contains the derivatives of the objective function compared to each of the `i` dimensions of the state (you have to find a way to have it by using the dual solution for instance).

- The **oneStepForward** function allows to compute one step of the forward pass:

  ```cpp
  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 Eigen::ArrayXd &p_particle, const int &p_isample) const = 0;
  ```

  As you can see, the **oneStepForward** is quite similar to the **oneStepBackward**. A tip, used in the examples and that you should use, is to build a function generating and solving the linear problem \[ AP_{n,t,g} \] (for a given scenario `g` and a given time step `t`) which appears for both the forward and the backward pass. This function creating and generating the linear problem will be called in both our functions **oneStepForward** and **oneStepBackward**. Take care that in the forward pass the current time step is given through the function `updateDates(current date,next date)` by the argument `current date` while in the backward pass the current time is given through the argument `next date` (this is a requirement needed to compute the regressions as exposed in the theoretical part). Finally note that the two previously described functions are **const** functions and you have to consider that during your implementation.
The other functions that you have to implement are simple functions (accessors) easy to understand.

**Implement your own Simulator class**

This simulator should be the object that will allow you to build some random quantities following a desired law. It should be given as an argument of your optimizer. You can implement it by yourself, however a set of simulators (gaussian, AR1, MeanReverting,...) are given in the test folder you could directly use it if it fits your problem requirements.

An implemented Simulator derivating from the SimulatorSDDPBase class needs to implement those functions:

- The `getNbSimul` function returns the number of simulations of random quantities used in regression part. It is the $U$ hinted in the theoretical part.
  
  ```cpp
  virtual int getNbSimul() const = 0;
  ```

- The `getNbSample` function returns the number of simulations of random quantities that are not used in the regression part. It is the $G$ hinted in the theoretical part. For instance, in some instances we need a gaussian random quantity in view of computing the noise when we are in the ”dependence of the random quantities” part.
  
  ```cpp
  virtual int getNbSample() const = 0;
  ```

- The `updateDates` function is really similar to the optimizer one. However you just have one argument (the current time step) here. It is also here that you have to generate new random quantities for the resolution.
  
  ```cpp
  virtual void updateDates(const double &p_date) = 0;
  ```

- The `getOneParticle` and the `getParticles` functions should return the quantities used in regression part.
  
  ```cpp
  virtual Eigen::VectorXd getOneParticle(const int &p_isim) const = 0;
  ```

  ```cpp
  virtual Eigen::MatrixXd getParticles() const = 0;
  ```

- The two last functions `resetTime` and `updateSimulationNumberAndResetTime` are quite explicit.

**12.3.4 Set of parameters**

The basic function `backwardForwardSDDP` should be called to use the SDDP part of the library. This function is templated by the regressor used :

- “LocalConstRegressionForSDDP” regressor permits to use a constant per mesh approximation of the SDDP cuts,
• “LocalLinearRegressionForSDDP” regressor permits to use a linear approximation per mesh of the SDDP cuts.

Most of the arguments are pretty clear (You can see examples in test/c++/functional). The strings correspond to names that will be given by the files which will store cuts, visited states or regressor data. \( p_{nbSimulCheckForSimu} \) corresponds to the number of simulations (number of forward pass called) when we have to check the convergence by comparing the outcome given by the forward pass and the one given by the backward pass. \( p_{nStepConv} \) indicates when the convergence is checked (each \( p_{nStepConv} \) iteration). \( p_{finalCut} \) corresponds to the cut used at the last time step: when the final value function is zero, the last cut is given by an all zero array of size \( nb_{state} + 1 \). \( p_{dates} \) is an array made up with all the time steps of the study period given as doubles, \( p_{iter} \) correspond to the maximum number of iterations. Finally, \( p_{stringStream} \) is an \texttt{ostringstream} in which the result of the optimization will be stored.

12.3.5 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 \textbf{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 \textbf{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 \cite{34}. 

192
• **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.

### 12.3.6 Outputs

The outputs of the SDDP library are not currently 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.

Once the convergence is achieved, the user should rerun some simulations adding some flag to store the results needed by the application (distribution cost etc...) : these results will be post-processed by the user.
12.4 Python API

A high level Python mapping is also available in the SDDP part. The backward-forward C++ function is exposed in Python by the SDDP module “StOptSDDP”. In this mapping only the linear per mesh regressor is used.

```python
import StOptSDDP
dir(StOptSDDP)
```

that should give

`['OptimizerSDDPBase', 'SDDPFinalCut', 'SimulatorSDDPBase', '__doc__', '__file__', '__name__', '__package__', 'backwardForwardSDDP']`

The “backwardForwardSDDP” realize the forward backward SDDP sweep giving a SDDP optimizer and a SDDP uncertainty simulator. The initial final cuts for the last time steps are provided by the “SDDPFinalCut” object.

To realize the mapping of SDDP optimizers and simulators written in C++ it is necessary to create a Boost Python wrapper. In order to expose the C++ optimizer class “OptimizeDemandSDDP” used in the test case “testDemandSDDP.cpp”, the following wrapper can be found in “StOpt/test/c++/python/BoostPythonSDDPOptimizers.cpp”

```c++
// Copyright (C) 2016 EDF
// All Rights Reserved
// This code is published under the GNU Lesser General Public License (GNU LGPL)
#include <boost/version.hpp>
#include "StOpt/core/grids/OneDimRegularSpaceGrid.h"
#include "StOpt/core/grids/OneDimData.h"
#include "StOpt/sddp/OptimizerSDDPBase.h"
#include "test/c++/tools/sddp/OptimizeDemandSDDP.h"
#include "test/c++/tools/simulators/SimulatorGaussianSDDP.h"
#if defined __WIN32
#include <boost/shared_ptr.hpp>
#else
#include "StOpt/python/BoostToStdSharedPtr.h"
#endif

#define BOOST 编译器版本 
if BOOST_VERSION < 105600
// map std::shared_ptr to boost python
namespace boost
{
    template<class T> T *get_pointer(std::shared_ptr<T> p)
    {
        return p.get();
    }
}
#endif
#endif
#endif
```

194
#include <boost/python.hpp>
#include "test/c++/python/FutureCurveWrap.h"

/** \file BoostPythonSDDPOptimizers.cpp
* \brief permits to map Optimizers for SDDP
* \author Xavier Warin */

#ifndef DEBUG
# undef DEBUG
#endif
#include <Python.h>
#define DEBUG
#else
#include <Python.h>
#endif
#include <numpy/arrayobject.h>
#include "StOpt/python/NumpyConverter.hpp"

using namespace boost::python;

/// wrapper for Optimizer for demand test case in SDDP
class OptimizeDemandSDDPWrap : public OptimizeDemandSDDP<
    SimulatorGaussianSDDP>
{
public:
    
    /// \brief Constructor
    /// \param p_sigD volatility for demand
    /// \param p_kappaD AR coefficient for demand
    /// \param p_timeDAverage average demand
    /// \param p_spot Spot price
    /// \param p_simulatorBackward backward simulator
    /// \param p_simulatorForward Forward simulator
    #ifdef WIN32
        OptimizeDemandSDDPWrap(const double &p_sigD, const double &p_kappaD,
            const FutureCurve &p_timeDAverage,
            const double &p_spot,
            const boost::shared_ptr<SimulatorGaussianSDDP> &
            p_simulatorBackward,
            const boost::shared_ptr<SimulatorGaussianSDDP> &
            p_simulatorForward):
            OptimizeDemandSDDP(p_sigD, p_kappaD,
                std::make_shared< StOpt::OneDimData< StOpt::
                OneDimRegularSpaceGrid, double > >(static_cast<
                StOpt::OneDimData< StOpt::
                OneDimRegularSpaceGrid, double > >(p_timeDAverage)),
                p_spot, make_shared_ptr<SimulatorGaussianSDDP>(
                p_simulatorBackward), make_shared_ptr<
                SimulatorGaussianSDDP>(p_simulatorForward)) { }
    #else

        

    #endif
}
OptimizeDemandSDDPWrap(const double &p_sigD, const double &p_kappaD, 
const FutureCurve &p_timeDAverage, 
const double &p_spot, 
const std::shared_ptr<SimulatorGaussianSDDP> & p_simulatorBackward, 
const std::shared_ptr<SimulatorGaussianSDDP> & p_simulatorForward):

OptimizeDemandSDDP(p_sigD, p_kappaD, 
std::make_shared< StOpt::OneDimData< StOpt::OneDimRegularSpaceGrid, double > >(static_cast<
StOpt::OneDimData< StOpt::OneDimRegularSpaceGrid, double > >(p_timeDAverage)),
p_spot, p_simulatorBackward, p_simulatorForward) {
}
#endif

#ifdef _WIN32
namespace boost
{

template <>
OptimizeDemandSDDPWrap const volatile *get_pointer< class 
OptimizeDemandSDDPWrap const volatile >(class OptimizeDemandSDDPWrap const volatile *c)
{
    return c;
}
}
#endif

BOOST_PYTHON_MODULE(SDDPOptimizers)
{
    Register<Eigen::ArrayXd>();
    Register<Eigen::ArrayXXd>();

    // map optimizer for demand test case
    #ifdef _WIN32
    class_<OptimizeDemandSDDPWrap, std::shared_ptr<OptimizeDemandSDDPWrap>, 
bases<StOpt::OptimizerSDDPBase> >("OptimizeDemandSDDP",
        init< const double &, const double &, const FutureCurve &,
        const double &,
        const boost::shared_ptr<SimulatorGaussianSDDP> &,
        const boost::shared_ptr<SimulatorGaussianSDDP> &>()
    )
    #else
    class_<OptimizeDemandSDDPWrap, std::shared_ptr<OptimizeDemandSDDPWrap>, 
bases<StOpt::OptimizerSDDPBase> >("OptimizeDemandSDDP",
        init< const double &, const double &, const FutureCurve &,
        const double &,
        const std::shared_ptr<SimulatorGaussianSDDP> &,
        const std::shared_ptr<SimulatorGaussianSDDP> &)
    #endif
}
The wrapper used to expose the SDDP simulator is given in “StOpt/test/c++/python/BoostPythonSimulators.cpp” Then it is possible to use the mapping to write a Python version of “testDemandSDDP.cpp”

```python
# Copyright (C) 2016 EDF
# All Rights Reserved
# This code is published under the GNU Lesser General Public License (GNU LGPL)
import StOptGrids
import StOptSDDP
import StOptGlobal
import Utils
import SDDPSimulators as sim
import SDDPOptimizers as opt
import numpy as NP
import unittest
import math
import imp
import sddp.backwardForwardSDDP as bfSDDP # import of the function written in python

# unittest equivalent of testDemandSDDP : here MPI version
# High level python interface : at level of the backwardForwardSDDP c++ file

def demandSDDPFunc(p_sigD, p_sampleOptim, p_sampleCheckSimul):

    maturity = 40
    nstep = 40;

    # optimizer parameters
    kappaD = 0.2; # mean reverting coef of demand
    spot = 3; # spot price

    # define a a time grid
timeGrid = StOptGrids.OneDimRegularSpaceGrid(0., maturity / nstep, nstep)

    # periodicity factor
    iPeriod = 52;
    # average demande values
demValues = []

    for i in list(range(nstep + 1)) :
        demValues.append(2. + 0.4 * math.cos((math.pi * i * iPeriod) / 197)
```
# define average demand
demGrid = Util.FutureCurve(timeGrid, demValues)

initialState = demGrid.get(0.)*np.ones(1)

finCut = StOptSDDP.SDDPFinalCut(np.zeros((2,1)))

# here cuts are not conditional to an uncertainty
nbMesh = np.array([],np.int32)
nbUncertainties = 1;

# backward simulator
backwardSimulator = sim.SimulatorGaussianSDDP(nbUncertainties, p_sampleOptim)
# forward simulator
forwardSimulator = sim.SimulatorGaussianSDDP(nbUncertainties)

# Create the optimizer
optimizer = opt.OptimizeDemandSDDP(p_sigD, kappaD, demGrid, spot, backwardSimulator, forwardSimulator)

# optimisation dates
dates = np.linspace(0., maturity,nstep + 1);

# names for archive
nameRegressor = "RegressorDemand";
nameCut = "CutDemand";
nameVisitedStates = "VisitedStateDemand";

# precision parameter
nIterMax = 40
accuracyClose = 1.
accuracy = accuracyClose / 100.
nstepIterations = 4; # check for convergence between nstepIterations step

values = StOptSDDP.backwardForwardSDDP(optimizer, p_sampleCheckSimul, initialState, finCut, dates, nbMesh, nameRegressor, nameCut, nameVisitedStates, nIterMax, accuracy, nstepIterations);

print("Values ", values)
return values

# unittest equivalent of testDemandSDDP : here low interface python version
# Low level python interface : use backwardForwardSDDP.py

def demandSDDPFuncLowLevel(p_sigD, p_sampleOptim, p_sampleCheckSimul):
    maturity = 40
nstep = 40;

# optimizer parameters
kappaD = 0.2; # mean reverting coef of demand
spot = 3; # spot price

# define a a time grid
timeGrid = StOptGrids.OneDimRegularSpaceGrid(0., maturity / nstep, nstep)

# periodicity factor
iPeriod = 52;
# average demande values
demValues = []

for i in list(range(nstep + 1)):
    demValues.append(2. + 0.4 * math.cos((math.pi * i * iPeriod) / nstep))

# define average demand
demGrid = Utils.FutureCurve(timeGrid, demValues)

initialState = demGrid.get(0.)*NP.ones(1)

finCut = StOptSDDP.SDDPFinalCut(NP.zeros((2,1)))

# here cuts are not conditional to an uncertainty
nbMesh = NP.array([],NP.int32)
nbUncertainties = 1;

# backward simulator
backwardSimulator = sim.SimulatorGaussianSDDP(nbUncertainties, p_sampleOptim)

# forward simulator
forwardSimulator = sim.SimulatorGaussianSDDP(nbUncertainties)

# Create the optimizer
optimizer = opt.OptimizeDemandSDDP(p_sigD, kappaD, demGrid, spot, backwardSimulator, forwardSimulator)

# optimisation dates
dates = NP.linspace(0., maturity, nstep + 1);

# names for archive
nameRegressor = "RegressorDemand";
nameCut = "CutDemand";
nnameVisitedStates = "VisitedStateDemand";

# precision parameter
nIterMax = 40
accuracyClose = 1.
accuracy = accuracyClose / 100.
nstepIterations = 4; # check for convergence between nstepIterations
values = bSDDP.backwardForwardSDDP(optimizer, p_sampleCheckSimul,
    initialState, finCut, dates, nbMesh, nameRegressor,
    nameCut, nameVisitedStates, nIterMax,
    accuracy, nstepIterations);

return values

class testDemandSDDP(unittest.TestCase):
    def testDemandSDDP1D(self):
        try:
            imp.find_module('mpi4py')
            found = True
        except:
            print("Not parallel module found ")
            found = False

        if found:
            from mpi4py import MPI
            world = MPI.COMM_WORLD
            sigD = 0.6;
            sampleOptim = 500;
            sampleCheckSimul = 500;
            values = demandSDDPFun(sigD, sampleOptim, sampleCheckSimul)

            if (world.rank==0):
                print("Values is ",values)

    def testDemandSDDP1DLowLevel(self):
        sigD = 0.6;
        sampleOptim = 500;
        sampleCheckSimul = 500;
        demandSDDPFunLowLevel(sigD, sampleOptim, sampleCheckSimul)

if __name__ == '__main__':
    unittest.main()
Algorithm 8 Run of backward pass

1: for $t = T, T - 1, ..., 1$ do
2:     for $x^{g}_{t-1}, g \in \{1, ..., G\}$ do
3:         for $\omega^{l}_{t}, l \in \{1, ..., L\}$ do
4:             Solve the following linear sub-problem.
\[
\begin{align*}
AP_{t,l}^{n,g} & \quad \begin{cases} 
Q^{l}_{t}(x^{g}_{t-1}, \omega^{l}_{t}) = \min_{x_{t},\theta_{t+1}} c_{t}^{\top} x_{t} + \theta_{t+1} \\
\text{s.c.} & A_{t} x_{t} = \omega^{l}_{t} - E_{t} x^{g}_{t-1}, \quad [\pi_{t}(\omega^{l}_{t})] \\
x_{t} \geq 0 \\
\theta_{t+1} + (\beta_{t+1}^{j})^{\top} x_{t} \geq \alpha_{t+1}^{j}, \quad j \in \{1, ..., G, ..., (n+1)G\}
\end{cases} \\
\end{align*}
\tag{12.6}
\]
5:             Store the dual solution $\pi_{t}(\omega^{l}_{t})$ and the primal one $Q^{l}(x^{g}_{t-1}, \omega^{l}_{t})$ of the linear sub-problem $[AP_{t,l}^{n,g}]$
6:             Compute the cut that goes with the $l^{th}$ hazard draw:
\[
\begin{align*}
\alpha^{g}_{t,l} &= Q^{l}_{t}(x^{g}_{t-1}, \omega^{l}_{t}) + \pi_{t}(\omega^{l}_{t})^{\top} E_{t} x^{g}_{t-1} \\
\beta^{g}_{t,l} &= E_{t}^{\top} \pi_{t}(\omega^{l}_{t})
\end{align*}
\tag{12.7}
\]
7:         end for
8:     end for
9: end for
10: Compute the $g^{th}$ new Benders cut at time $t$ at iteration $n$: is defined as the mean value of the cuts obtained before:
\[
\begin{align*}
\alpha^{k}_{t} &= \frac{1}{T} \sum_{l=1}^{L} \alpha^{g}_{t,l} \\
\beta^{k}_{t} &= \frac{1}{L} \sum_{l=1}^{L} \beta^{g}_{t,l}
\end{align*}
\tag{12.8}
\]
11: Solve the following linear sub-problem. ;
\[
\begin{align*}
AP_{0}^{n} & \quad \begin{cases} 
Q_{0} = \min_{x_{0},\theta_{1}} c_{0}^{\top} x_{0} + \theta_{1} \\
\text{s.c.} & A_{0} x_{0} = \omega_{0}, \quad [\pi_{0}(\omega_{0})] \\
x_{0} \geq 0 \\
\theta_{1} + (\beta_{1}^{j})^{\top} x_{0} \geq \alpha_{1}^{j}, \quad j \in \{1, ..., G, ..., (n+1)G\}
\end{cases} \\
\end{align*}
\tag{12.9}
\]
12: Save the cost backward $z_{n} = Q_{0}$
Algorithm 9 Run of backward pass with time-related random quantities (AR1 process)

1: Pick up the set of the following pairs: \( \{x^g_t, \omega^g_{t, \text{dep}}\} \) for \( g \in \{1, ..., G\}; \quad t \in \{1, ..., T\} \)

2: for \( t = T, T - 1, ..., 1 \) do

3: \hspace{1em} for \( (x^g_{t-1}, \omega^g_{t-1, \text{dep}}), \quad g \in \{1, ..., G\} \) do

4: \hspace{2em} for \( l \in \{1, ..., L\} \) do

5: \hspace{3em} Produce a value for the white noise \( \epsilon^l_t \);

6: \hspace{3em} Compute the element \( \hat{\omega}^l_t \) knowing the previous random quantity \( \omega^g_{t-1} \):

\[
\hat{\omega}^l_t = \sigma_{w,t} \left( \psi_1 \frac{\omega^g_{t-1} - \mu_{w,t-1}}{\sigma_{w,t-1}} + \psi_2 \epsilon^l_t \right) + \mu_{w,t} \tag{12.14}
\]

7: \hspace{3em} Solve the following linear sub-problem:

\[
\begin{bmatrix}
\mathcal{AP}^n_{t,l}^g
\end{bmatrix} \begin{aligned}
&Q_t^g(x^g_{t-1, \text{dep}}, \hat{\omega}^l_t) = \min_{x_t, \theta_t} c_t^\top x_t + \theta_t + 1 \\
&\text{subject to:} \\
&A_t x_t = P_t \hat{\omega}^l_t - E_t x^g_{t-1}, \\
&x_t \geq 0 \\
&\theta_t + (\beta^g_{t,l})^\top x_t + (\gamma^g_{t,l})^\top \hat{\omega}^l_t \geq \alpha^g_{t,l}, \quad j \in \{1, ..., G, ..., (n+1)G\}
\end{aligned} \tag{12.15}
\]

8: \hspace{3em} Store the dual solution \( \pi_t(\omega^l_t) \) and the primal one \( Q^g_t(x^g_{t-1, \text{dep}}, \hat{\omega}^l_t) \) of the primal problem \( \mathcal{AP}^n_{t,l}^g \)

9: \hspace{3em} Compute the cut that goes with the \( l^{th} \) hazard draw:

\[
\begin{aligned}
&\alpha^g_{t,l} = Q_t^g(x^g_{t-1, \text{dep}}, \hat{\omega}^l_t) + \pi_t(\omega^l_t) - \left( E_t x^g_{t-1} - \psi_1 \frac{\sigma_{w,t-1}}{\sigma_{w,t-1}} P \omega^g_{t-1} \right) \\
&\beta^g_{t,l} = E_t^\top \pi_t(\omega^l_t) \\
&\gamma^g_{t,l} = \psi_1 \frac{\sigma_{w,t-1}}{\sigma_{w,t-1}} P^\top \pi_t(\hat{\omega}^l_t) \tag{12.16}
\end{aligned}
\]

10: \hspace{1em} end for

11: Compute the \( g^{th} \) new Benders cut at time \( t \) at iteration \( n \) defined as the mean value of the cuts obtained before for \( k = nG + g \)

\[
\begin{aligned}
\alpha^k_t &= \frac{1}{L} \sum_{l=1}^{L} \alpha^g_{t,l}, \quad \beta^k_t = \frac{1}{L} \sum_{l=1}^{L} \beta^g_{t,l}, \quad \gamma^k_t = \frac{1}{L} \sum_{l=1}^{L} \gamma^g_{t,l}
\end{aligned}
\]

12: end for

13: end for

14: Solve the following linear sub-problem.

\[
\begin{aligned}
&Q_0 = \min_{x_0, \theta_1} c_0^\top x_0 + \theta_1 \\
&\text{subject to:} \\
&A_0 x_0 = \omega_0, \quad [\pi_0(\omega_0)] \\
&x_0 \geq 0 \\
&\theta_1 + (\beta^g_{l})^\top x_0 + (\gamma^g_{l})^\top \omega^\text{dep}_0 \geq \alpha^g_{l}, \quad j \in \{1, ..., G, ..., (n+1)G\}
\end{aligned} \tag{12.17}
\]

15: Save the backward cost \( z_n = Q_0 \)
Algorithm 10 Run of the backward pass with time-related (AR1) and non-convex random quantities

1: Pick up the set of the following pairs : \( \{x_t^g, \omega_{t,dep}^g\} \) pour \( g \in \{1, ..., G\} \), \( t \in \{1, ..., T\} \)

2: for \( t = T, T - 1, ..., 1 \) do

3: Generate values for the non-convex random quantities at time \( t \) knowing the scenarios at time \( t - 1 \) : \( s_t^u, u \in \{1, ..., U\} \)

4: for \( (x_{t-1}^g, \omega_{t-1,dep}^g) \), \( g \in \{1, ..., G\} \) do

5: for \( u \in \{1, ..., U\} \) do

6: Consider a scenario \( s_t^u \) in the mesh \( D_L \);

7: for \( l \in \{1, ..., L\} \) do

8: Produce a value for the white noise \( \epsilon_t^l \) ;

9: Compute the element \( \omega_t^l \) knowing the previous random quantity \( \omega_{t-1,dep}^g \):

\[
\hat{\omega}_t^l = \sigma_{\omega,t} \left( \psi_1 \frac{\omega_{t-1,dep}^g - \mu_{\omega,t-1}}{\sigma_{\omega,t-1}} + \psi_2 \epsilon_t^l \right) + \mu_{\omega,t} \quad (12.18)
\]

10: Pick up the cuts corresponding the mesh \( D_L : \{\alpha_t^{Lj}(s), \beta_t^{Lj}(s), \gamma_t^{Lj}(s)\} \), \( j \in \{1; ..., (n + 1)G\} \)

11: Solve the following linear sub-problem :

\[
[AP_{t,l}^{n,g}] = \begin{cases} 
Q_t^l(x_{t-1}^g, \omega_{t-1,dep}^g, \hat{\omega}_t^l, s_t^u) = \min_{x_t, \theta_{t+1}} c_t(s_t^u)^\top x_t + \theta_{t+1}(s_t^u) \\
\text{s.c. } A_t(s_t^u)x_t = P^\top \omega_t^l - E_t x_{t-1}^g, \quad [\pi_t(\hat{\omega}_t^l, s_t^u)] \\
x_t \geq 0 \\
\theta_{t+1}(s_t^u) + (\beta_t^{Lj}(s_t^u))^{\top} x_t + (\gamma_t^{Lj}(s_t^u))^{\top} \omega_t^l \geq \alpha_t^{Lj}(s_t^u), \\
\quad j \in \{1, ..., G, ..., nG\} \quad (12.19)
\end{cases}
\]

12: Store the dual solution \( \pi_t(\hat{\omega}_t^l) \) and the primal solution \( Q_t^l(x_{t-1}^g, \omega_{t-1,dep}^g, \hat{\omega}_t^l, s_t^u) \) of the problem \( [AP_{t,l}^{n,g}] \)

13: Calculate the corresponding cut at the \( l^{th} \) draw of uncertainties :

\[
\hat{\alpha}_{t,l}^{g,L}(s_t^u) = Q_t^l(x_{t-1}^g, \omega_{t-1,dep}^g, \hat{\omega}_t^l) + \pi_t(\hat{\omega}_t^l)^\top \left( E_t x_{t-1}^g - \psi_1 \sigma_{\omega,t} P^\top \omega_{t-1}^g \right) \\
\hat{\beta}_{t,l}^{g,L}(s_t^u) = E_t^\top \pi_t(\hat{\omega}_t^l) \\
\hat{\gamma}_{t,l}^{g,L}(s_t^u) = \psi_1 \sigma_{\omega,t} P^\top \pi_t(\hat{\omega}_t^l) 
\]

14: end for
Algorithm 11 Run of the backward pass with time-related (AR1) and non-convex random quantities

15: Compute the cut for a non-convex random quantity $s_u$ at time $t$ at iteration $n$: it is defined as the weighted average on the $L$ Benders cut obtained before:

$$
\hat{\alpha}_t^g(s_u^t) = \frac{1}{L} \sum_{l=1}^L \hat{\alpha}_{t,l}^g(s_u^t)
$$

$$
\hat{\beta}_t^g(s_u^t) = \frac{1}{L} \sum_{l=1}^L \hat{\beta}_{t,l}^g(s_u^t), \quad j = nG + g
$$

$$
\hat{\gamma}_t^g(s_u^t) = \frac{1}{L} \sum_{l=1}^L \hat{\gamma}_{t,l}^g(s_u^t)
$$

16: end for

17: for $I_i, i \in \{1, ..., M\}$ do

18: Compute the $g^{th}$ new cut of the mesh $D_{L_i}$ at time $t$ at iteration $n$ defined as the conditional expectation with respect to the scenario $u$ at time $t$

$$
\left\{
\begin{align*}
\alpha_t^i(s_{I_i-1}^u) &= \mathbb{E} \left[ \hat{\alpha}_t^g(s_u^t) | s_{I_i-1}^u \right], \\
\beta_t^i(s_{I_i-1}^u) &= \mathbb{E} \left[ \hat{\beta}_t^g(s_u^t) | s_{I_i-1}^u \right], \quad j = nG + g \\
\gamma_t^i(s_{I_i-1}^u) &= \mathbb{E} \left[ \hat{\gamma}_t^g(s_u^t) | s_{I_i-1}^u \right]
\end{align*}
\right.
$$

19: end for

20: end for

21: end for

22: Solve the initial linear sub problem $[AP_0^n]$

23: Save the backward cost $z_n = Q_0$
Algorithm 12 Run of backwardforwardSDDP(), the main function

1: Init: \( x_t^g = \text{TransitionOptimizer.onesadmissiblestate}(t) \), for \( g \in \{1,...,G\} \) and \( t \in \{1,...,T-1\} \), \( n = 0 \)

2: while \( \psi > \epsilon \) and \( n < n_{\text{max}} \) do
3: \( \text{StOpt} \)
4: \( V_b = \text{backwardSDDP()} \) Using the previously computed set \( (x_t^g)_{t,g} \) and creating a set of cuts
5: \( V_f = \text{forwardSDDP()} \) Simulation using the cuts created in all the backward passes and update the set \( (x_t^g)_{t,g} \)
6: \( \psi = \frac{V_f - V_b}{V_f} \)
7: \( n = n + 1 \)
8: end while

Algorithm 13 Run of forwardSDDP() (n\textsuperscript{th} iteration)

1: for \( g \in \Omega_g \) do
2: \( \text{for } t \in \{0,...,T\} \) do
3: \( \text{TransitionOptimizer.updatedates}(t,t+1): \) update the required data following the current time step (iterator over current time step, average demand,...)
4: \( \text{SimulatorSim.updatedates}(t): \) give the random quantities \( (\omega_t^g) \) for the scenario \( g \) at time \( t \)
5: \( \text{StOpt} \) Read the previously computed files to gather \( \alpha_{t+1}^j, \beta_{t+1}^j \), for \( j \in \{1,...,G,...,nG\} \)
6: \( \text{TransitionOptimizer.onestepforward():} \) Solve the following linear sub-problem.
\[
\begin{aligned}
[AP_{t,g}^n] & \quad \begin{cases} 
Q_t^g(x_{t-1}^g, \omega_t^g) = \min_{x_t, \theta_{t+1}} c_t^\top x_t + \theta_{t+1} \\
\text{s.c.} \\
A_t x_t = \omega_t^g - E_t x_{t-1}^g, \\
x_t \geq 0 \\
\theta_{t+1} + (\beta_{t+1}^j)^\top x_t \geq \alpha_{t+1}^j, \quad j \in \{1,...,G,...,nG\}
\end{cases} \\
\end{aligned}
\tag{12.21}
\]
7: \( \text{StOpt} \) Store the primal solution \( (x_t^g) \) of the problem \( [AP_{t,g}^n] \)
8: \( \text{end for} \)
9: \( \text{StOpt} \) Compute the cost for scenario \( g \), at iteration \( n \) : \( \bar{z}_n^g = \sum_{t=0}^{T} c_t x_t^g \)
10: end for
11: \( \text{StOpt} \) Compute the total cost in forward pass at iteration \( n \) : \( \bar{z}_n = \frac{1}{G} \sum_{g=1}^{G} \bar{z}_n^g \)
Algorithm 14 Run of `backwardSDDP()`

1: for $t = T, T - 1, \ldots, 0$ do

2: StOpt Read the previously computed files to gather $x_{t-1}^g$, for $g \in \{1, \ldots, G, \}$

3: TransitionOptimizer.updatedates(t-1,t): update the required data following the current time step (iterator over current time step, average demand,...)

4: SimulatorOpt.updatedates(t): give the random quantities for the $L$ scenarios at time $t$

5: StOpt Read the previously computed files to gather $\alpha_{t+1}^j, \beta_{t+1}^j$, for $j \in \{1, \ldots, G, \ldots, nG\}$

6: for $x_{t-1}^g, g \in \{1, \ldots, G\}$ do

7: for $\omega_l^t, l \in \{1, \ldots, L\}$ do

8: \textbf{TransitionOptimizer.onestepbackward()}

9: Solve the following linear sub-problem.

\[ \begin{cases} Q_t^l(x_{t-1}^g, \omega_l^t) = \min_{x_t, \theta_{t+1}} c_t^\top x_t + \theta_{t+1} \\ \text{s.c.} & A_t x_t = \omega_l^t - E_t x_{t-1}^g, \quad [\pi_t(\omega_l^t)] \\ & x_t \geq 0 \\ & \theta_{t+1} + (\beta^j_{t+1})^\top x_t \geq \alpha^j_{t+1}, \quad j \in \{1, \ldots, G, \ldots, (n+1)G\} \end{cases} \] \hfill (12.22)

10: \textbf{Return:} the dual solution $\pi_t(\omega_l^t)$ and the primal one $Q_t^l(x_{t-1}^g, \omega_l^t)$ of the linear sub-problem $[AP_{n,g}^{n,g}]$\end{Verbatim}

11: StOpt Compute the $g^{th}$ new Benders cut at time $t$ at iteration $n : \alpha_t^j, \beta_t^j$, for $j \in \{(n-1)G, (n-1)G + 1, \ldots, nG\}$

12: end for

13: end for

14: StOpt Save the cost backward $z_n = Q_0$
Part VII

Nesting Monte carlo for general non linear PDEs
The method described is has been studied in [48], [47] and is using some ideas in [21], [46].

Our goal is to solve the general full non linear equation

\[-\partial_t u - Lu(t, x) = f(t, x, u(t, x), Du(t, x), D^2 u(t, x)),\]

\[u_T = g, \quad t < T, \quad x \in \mathbb{R}^d,\]

with

\[Lu(t, x) := \mu Du(t, x) + \frac{1}{2} \sigma \sigma^\top : D^2 u(t, x)\]

so that \(L\) is the generator associated to

\[X_t = x + \mu t + \sigma dW_t,\]

with \(\mu \in \mathbb{R}^d\), and \(\sigma \in \mathbb{M}^d\) is some constant matrix.

In the whole article, \(\rho\) is the density of a general random variable following a gamma law so that

\[\rho(x) = \lambda^\alpha x^{\alpha - 1} e^{-\lambda x} \Gamma(\alpha), \quad 1 \geq \alpha > 0.\]

The associated cumulated distribution function is

\[F(x) = \frac{\gamma(\alpha, \lambda x)}{\Gamma(\alpha)}\]

where \(\gamma(s, x) = \int_0^x t^{s-1} e^{-t} dt\) is the incomplete gamma function and \(\Gamma(s) = \int_0^\infty t^{s-1} e^{-t} dt\) is the gamma function.

The methodology follows the ideas of [48] and [46].

We suppose here that \(\sigma\) is non degenerated so that \(\sigma^{-1}\) exists.

Let set \(p \in \mathbb{N}^+\). For \((N_0, ..., N_{p-1}) \in \mathbb{N}^p\), we introduce the sets of i-tuple, \(Q_i = \{k = (k_1, ..., k_i)\}\) for \(i \in \{1, ..., p\}\) where all components \(k_j \in [1, N_{j-1}]\). Besides we define \(Q^p = \bigcup_{i=1}^p Q_i\).

We construct the sets \(Q_i^o\) for \(i = 1, ..., p\), such that

\[Q_1^o = Q_1\]

and the set \(Q_i^o\) for \(i > 1\) are defined by recurrence :

\[Q_{i+1}^o = \{(k_1, ..., k_i, k_{i+1})/(k_1, ..., k_i) \in Q_i, k_{i+1} \in \{1, ..., N_{i+1}, 1_1, ..., (N_{i+1})_1, l_2, ..., (N_{i+1})_2\}\}

so that to a particle noted \((k_1, ..., k_i) \in Q_i^o\) such that \(k_i \in \mathbb{N}\), we associate two fictitious particles noted \(k^1 = (k_1, ..., k_{i-1}, (k_i)_1)\) and \(k^2 = (k_1, ..., k_{i-1}, (k_i)_2)\).

To a particle \(k = (k_1, ..., k_i) \in Q_i^o\) we associate its original particle \(o(k) \in Q_i\) such that

\[o(k) = (\hat{k}_1, ..., \hat{k}_i)\] where \(k_j = l\) if \(k_j = l_1\) or \(l_2\).

For \(k = (k_1, ..., k_i) \in Q_i^o\) we introduce the set of its non fictitious sons

\[\hat{Q}(k) = \{l = (k_1, ..., k_i, m)/m \in \{1, ..., N_i\}\} \subset Q_{i+1}^o,\]
and the set of all sons
\[ \tilde{Q}(k) = \{ l = (k_1, ..., k_i, m) / m \in \{1, ..., N_i, 1, ..., (N_i)_{1}, 1_{2}, ..., (N_i)_{2}\} \} \subset Q_{i+1}^o. \]

By convention \( \check{Q}(\emptyset) = \{ l = (m) / m \in \{1, ..., N_0\} \} = Q_1 \). Reciprocally the ancestor \( k \) of a particle \( \check{k} \) in \( \tilde{Q}(k) \) is noted \( \check{k}^- \).

We define the order of a particle \( k \in Q_i^p, i \geq 0 \), by the function \( \kappa \):
\[
\kappa(k) = 0 \text{ for } k_i \in \mathbb{N}, \\
\kappa(k) = 1 \text{ for } k_i = l_1, l \in \mathbb{N} \\
\kappa(k) = 2 \text{ for } k_i = l_2, l \in \mathbb{N}
\]

We define the sequence \( \tau_k \) of switching increments i.i.d. random variables with density \( \rho \) for \( k \in Q^p \). The switching dates are defined as :
\[
\begin{align*}
\tilde{T}_j &= \tau(j) \land T, j \in \{1, .., N_0\} \\
\tilde{T}_k &= (T \land \tau_k) \land T, k = (k_1, .., k_i) \in Q_i, \check{k} \in \check{Q}(k)
\end{align*}
\tag{12.25}
\]

By convention \( T_k = T_{o(k)} \) and \( \tau_k = \tau_{o(k)} \). For \( k = (k_1, .., k_i) \in Q_i^p \) and \( \check{k} = (k_1, .., k_i, k_{i+1}) \in \check{Q}(k) \) we define the following trajectories :
\[
\begin{align*}
W_s^\check{k} &= W_{s-T_k}^k + 1_{\kappa(\check{k})=0} W_{s-T_k}^{\rho(\check{k})} - 1_{\kappa(\check{k})=1} W_{s-T_k}^{\rho(\check{k})}, \text{ and} \\
X_s^\check{k} &= x + \mu s + \sigma W_s^\check{k}, \quad \forall s \in [T_k, \tilde{T}_k],
\end{align*}
\tag{12.26, 12.27}
\]

where the \( W^k \) for \( k \) in \( Q^p \) are independent \( d \)-dimensional Brownian motions, independent of the \( (\tau_k)_{k \in Q^p} \).

In order to understand what these different trajectories represent, suppose that \( d = 1, \mu = 0, \sigma = 1 \) and let us consider the original particle \( k = (1, 1, 1) \) such that \( T_{(1,1,1)} = T \).

Following equation (12.26),
\[
\begin{align*}
X_T^{(1,1,1)} &= W_{T(1)} + W_{T(1)-T(1)}^{(1,1)} + W_{T-T(1)}^{(1,1)} \\
X_T^{(1,1,1)} &= -W_{T(1)} + W_{T(1)-T(1)}^{(1,1)} + W_{T-T(1)}^{(1,1)} \\
X_T^{(1,1,1)} &= W_{T(1)} + W_{T(1)-T(1)}^{(1,1)} + W_{T-T(1)}^{(1,1)} \\
X_T^{(2,1,1)} &= -W_{T(1)} + W_{T(1)-T(1)}^{(1,1)} + W_{T-T(1)}^{(1,1)} \\
X_T^{(1,1,1)} &= -W_{T(1)} + W_{T(1)-T(1)}^{(1,1)} + W_{T-T(1)}^{(1,1)} \\
\end{align*}
\]

such that all particles are generated from the \( \tilde{W}^k \) used to define \( X_T^{(1,1,1)} \).
Using the previous definitions, we consider the estimator defined by:

\[
\begin{align*}
\bar{u}_n &= \frac{1}{N_0} \sum_{j=1}^{N_0} \phi(0, T_{(j)}, X_{(j)}^{(j)}, \bar{u}_{(j)}, D\bar{u}_{(j)}, D^2\bar{u}_{(j)}), \\
\bar{u}_k &= \frac{1}{N_i} \sum_{\bar{k} \in Q(k)} \frac{1}{2} \left( \phi(T_k, T_{\bar{k}}, X_{T_k}^{\bar{k}}, \bar{u}_{\bar{k}}, D\bar{u}_{\bar{k}}, D^2\bar{u}_{\bar{k}}) \right) \\
\bar{D}\bar{u}_k &= \frac{1}{N_i} \sum_{\bar{k} \in Q(k)} \left( \frac{1}{2} \left( \phi(T_k, T_{\bar{k}}, X_{T_k}^{\bar{k}}, \bar{u}_{\bar{k}}, D\bar{u}_{\bar{k}}, D^2\bar{u}_{\bar{k}}) \right) - \phi(T_k, T_{\bar{k}}, X_{T_k}^{\bar{k}}, \bar{u}_{\bar{k}}, D\bar{u}_{\bar{k}}, D^2\bar{u}_{\bar{k}}) \right) \\
\bar{D}^2\bar{u}_k &= \frac{1}{N_i} \sum_{\bar{k} \in Q(k)} \left( \frac{1}{2} \left( \phi(T_k, T_{\bar{k}}, X_{T_k}^{\bar{k}}, \bar{u}_{\bar{k}}, D\bar{u}_{\bar{k}}, D^2\bar{u}_{\bar{k}}) \right) + \phi(T_k, T_{\bar{k}}, X_{T_k}^{\bar{k}}, \bar{u}_{\bar{k}}, D\bar{u}_{\bar{k}}, D^2\bar{u}_{\bar{k}}) \right) - 2\phi(T_k, T_{\bar{k}}, X_{T_k}^{\bar{k}}, \bar{u}_{\bar{k}}, D\bar{u}_{\bar{k}}, D^2\bar{u}_{\bar{k}}), \\
\bar{u}_k &= g(X_k), \quad \text{for } k \in Q^p, \\
\bar{D}\bar{u}_k &= Dg(X_k), \quad \text{for } k \in Q^p, \\
\bar{D}^2\bar{u}_k &= D^2g(X_k), \quad \text{for } k \in Q^p
\end{align*}
\]

(12.28)
where $\phi$ is defined by:
\[
\phi(s,t,x,y,z,\theta) := \frac{1_{\{t \geq T\}}}{F(T-s)}g(x) + \frac{1_{\{t\leq T\}}}{\rho(t-s)}f(t,x,y,z,\theta).
\] (12.29)

and
\[
\forall^k = \sigma^{-\top} \frac{\tilde{W}_{T^k_{T_k^{-}}}^k}{T_k - T_k^-}
\]
\[
\forall^k = (\sigma^\top)^{-1} \frac{\tilde{W}_{T^k_{T_k^{-}}}^k (\tilde{W}_{T^k_{T_k^{-}}}^k)^\top - (T_k - T_k^-)I_d}{(T_k - T_k^-)^2} \sigma^{-1}
\] (12.30)

As explained before, the $u$ and $Du$ term in $f$ are treated as explained in [48] and only the $D^2u$ treatment is the novelty of this scheme.

**Remark 28** In practice, we just have the $g$ value at the terminal date $T$ and we want to apply the scheme even if the derivatives of the final solution is not defined. We can close the system for $k$ in $Q^p$, replacing $\phi$ by $g$ and taking some value for $N_{p+1}$:
\[
\tilde{u}_k^p = \frac{1}{N_{p+1}} \sum_{k \in \hat{Q}(k)} \frac{1}{2} (g(X_{\tilde{k}^k}) + g(X_{\tilde{k}^1})),
\]
\[
D\tilde{u}_k^p = \frac{1}{N_{p+1}} \sum_{k \in \hat{Q}(k)} \frac{1}{2} (g(X_{\tilde{k}^k}) - g(X_{\tilde{k}^1})),
\]
\[
D^2\tilde{u}_k^p = \frac{1}{N_{p+1}} \sum_{k \in \hat{Q}(k)} \frac{1}{2} (g(X_{\tilde{k}^k}) + g(X_{\tilde{k}^1}) - 2g(X_{\tilde{k}^2}))
\]

**Remark 29** In the case where the coefficient are not constant, some Euler scheme can be added as explained in [48].

An effective algorithm for this scheme is given these two functions:

**Algorithm 15** Outer Monte Carlo algorithm ($V$ generates unit Gaussian RV, $\tilde{V}$ generates RV with gamma law density)

1: **procedure** PDEEval($\mu$, $\sigma$, $g$, $f$, $T$, $p$, $x_0$, $\{N_0,..,N_{p+1}\}$, $V$, $\tilde{V}$)
2: \[ u_M = 0 \]
3: $x(0,:) = x_0(:)$ \quad \triangleright x$ is a matrix of size $1 \times n$
4: **for** $i = 1, N_0$ **do**
5: \[ (u, Du, D^2u) = \text{EvalUDUD2U}(x_0, \mu, \sigma, g, T, V, \tilde{V}, p, 1, 0, 0) \]
6: \[ u_M = u_M + u(0) \]
7: **end for**
8: **return** $\frac{U_M}{N_0}$
9: **end procedure**
Algorithm 16 Inner Monte Carlo algorithm where \( t \) is the current time, \( x \) the array of particles positions of size \( m \times d \), and \( l \) the nesting level.

1: \textbf{procedure} EvalUDUD2U\((x, \mu, \sigma, g, T, \bar{V}, p, m, t, l)\)
2: \>
3: \>
4: \>
5: \>
6: \>
7: \>
8: \>
9: \>
10: \>
11: \>
12: \>
13: \>
14: \>
15: \>
16: \>
17: \>
18: \>
19: \>
20: \>
21: \>
22: \>
23: \>
24: \>
25: \>
26: \>
27: \>
28: \>
29: \>
30: \>
31: \>
32: \textbf{end procedure}

\[\begin{align*}
\tau &= \min(\bar{V}(), T - t), \quad \triangleright \text{Sample the time step} \\
G &= \bar{V}() \quad \triangleright \text{Sample the } n \text{ dimensional Gaussian vector} \\
xS(1 : m, :) &= x(:, \mu \tau + \sigma G \sqrt{\tau}) \\
xS(m + 1 : 2m, :) &= x(:, \mu \tau) \\
xS(2m + 1 : 3m, :) &= x(:, \mu \tau - \sigma G \sqrt{\tau}) \\
tS &= t + \tau \quad \triangleright \text{New date} \\
\text{if } ts \geq T \text{ or } l = p \text{ then} \\
g_1 &= g(xS(1 : m, :)); g_2 = (xS(m + 1 : 2m, :)); g_3 = g(xS(2m + 1 : 3m, :)) \\
u(:, :) &= \frac{1}{2}(g_1 + g_3) \\
Du(:, :) &= \frac{1}{2}(g_1 - g_3) \sigma^{-\top} G \\
D^2u(:, :, :) &= \frac{1}{2}(g_1 + g_3 - 2g_2)\sigma^{-\top} GG^{-\top} - \mathbf{I}_d \sigma^{-1} \\
\text{if } l \neq p \text{ then} \\
(u(:, :) + Du(:, :, :) + D^2u(:, :, :) \right) &= \frac{1}{f(\tau)} \\
\text{end if} \\
\text{else} \\
y(:, :) &= 0; z(:, :) = 0; \theta(:, :, :) = 0 \\
\text{for } j = 1, N_{t+1} \text{ do} \\
(y, z, \theta) &= \text{EvalUDUD2U}(xS, \mu, \sigma, g, T, \bar{V}, p, 3m, tS, l + 1) \\
\text{end for} \\
(y, z, \theta) &= N_{t+1} \\
\text{for } q = 1, m \text{ do} \\
f_1 &= f(ts, xS(q), y(q), z(q, :], \theta(q, :) \right) \\
f_2 &= f(ts, xS(m + q), y(m + q), z(m + q, :], \theta(m + q, :) \right) \\
f_3 &= f(ts, xS(2m + q), y(2m + q), z(2m + q, :], \theta(2m + q, :) \right) \\
u(i) &= \frac{1}{2}(f_1 + f_3) \\
Du(i, :) &= \frac{1}{2}(f_1 - f_3) \sigma^{-\top} G \\
D^2u(i, :, :) &= \frac{1}{2}(f_1 + f_3 - 2f_2)\sigma^{-\top} GG^{-\top} - \mathbf{I}_d \sigma^{-1} \\
\text{end for} \\
\text{end if}
\end{align*}\]
Part VIII

Some test cases description
Chapter 13

Some test cases description in C++

In this part, we describe the functional test cases of the library. The c++ version of these test cases can be found in “test/c++/functional” while their python equivalent (when existing) can be found in “test/python/functional”. We describe here in details the c++ test cases.

13.1 American option

The library gives some test cases for the Bermudean option problem ([7] for details on the bermudean option problem). All Bermudean test cases use a basket option payoff. The reference for the converged methods can be found in [7].

13.1.1 testAmerican

The test case in this file permits to test during the Dynamic Programming resolution different regressors:

• either using some local functions basis with support of same size:
  – Either using a constant per mesh representation of the function (“LocalSameSizeConstRegression” regressor)
  – Either using a linear per mesh representation of the function (“LocalSameSizeLinearRegression” regressor)

• either using some function basis with adaptive support ([7])
  – Either using a constant per mesh representation of the function (“LocalConstRegression” regressor)
  – Either using a linear per mesh representation of the function (“LocalLinearRegression” regressor)

• Either using global polynomial regressor:
  – Either using Hermite polynomials,
  – Either using Canonical polynomials (monomes),
- Either using Tchebychev polynomials.

- Either using sparse regressor,
- Either using kernel regressors:
  - either using constant kernel regressor,
  - either using linear kernel regressor.

**testAmericanLinearBasket1D**
Test 1D problem with “LocalLinearRegression” regressor.

**testAmericanConstBasket1D**
Test 1D problem with “LocalConstRegression” regressor.

**testAmericanSameSizeLinearBasket1D**
Test 1D problem with “LocalSameSizeLinearRegression” regressor.

**testAmericanSameSizeConstBasket1D**
Test 1D problem with LocalSameSizeConstRegression regressor.

**testAmericanGlobalBasket1D**
Test 1D problem with global Hermite, Canonical and Tchebychev regressor.

**testAmericanGridKernelConstBasket1D**
Test 1D problem with classical kernel regression

**testAmericanGridKernelLinearBasket1D**
Test 1D problem with linear kernel regression

**testAmericanLinearBasket2D**
Test 2D problem with “LocalLinearRegression” regressor.

**testAmericanConstBasket2D**
Test 2D problem with “LocalConstRegression” regressor.

**testAmericanSameSizeLinearBasket2D**
Test 2D problem with “LocalSameSizeLinearRegression” regressor.
testAmericanSameSizeConstBasket2D
Test 2D problem with LocalSameSizeConstRegression regressor.

testAmericanGlobalBasket2D
Test 2D problem with global Hermite, Canonical and Tchebychev regressor.

testAmericanGridKernelConstBasket2D
Test 2D problem with classical kernel regression.

testAmericanGridKernelLinearBasket1D
Test 2D problem with linear kernel regression.

testAmericanBasket3D
Test 3D problem with “LocalLinearRegression” regressor.

testAmericanGlobalBasket3D
Test 3D problem with global Hermite, Canonical and Tchebychev regressor.

testAmericanGridKernelLinearBasket3D
Test 3D problem with linear kernel regression.

testAmericanBasket4D
Test 4D problem with “LocalLinearRegression” regressor.

13.1.2 testAmericanConvex
Three test cases with basket american options are implemented trying to keep convexity of the solution.

testAmericanLinearConvexBasket1D
Linear adapted regression in 1D preserving the convexity at each time step.

testAmericanLinearConvexBasket2D
Linear adapted regression in 2D trying to preserve the convexity at each time step.

testAmericanLinearConvexBasket3D
Linear adapted regression in 3D trying to preserve the convexity at each time step.
13.1.3  testAmericanForSparse

This test case is here to test sparse grid regressors (see section 3.3). As described before we can use a linear, quadratic or cubic representation on each cell. The reference is the same as in the testAmerican subsection so linked to a Bermudean basket option.

testAmericanSparseBasket1D

Use sparse grids in 1D (so equivalent to full grid) for linear, quadratic or cubic representation.

testAmericanSparseBasket2D

Use sparse grids in 2D for linear, quadratic or cubic representation.

testAmericanSparseBasket3D

Use sparse grids in 3D for linear, quadratic or cubic representation.

testAmericanSparseBasket4D

Use sparse grids in 4D for linear, quadratic or cubic representation.

13.1.4  testAmericanOptionCorrel

Same case as before but with correlations between assets. Permits to test that rotation due to the PCA analysis works correctly.

testAmericCorrel

Check in 2D that

- Local Constant per mesh regression with and without rotation give the same result,
- Local Linear per mesh regression with and without rotation give the same result,
- Global regression with and without rotation give the same result.

13.2  testSwingOption

The swing option problem is the generalization of the American option using a Black Scholes model for the underlying asset: out of a set of “nStep” dates (chosen equal to 20 here) we can choose N dates (N equal to three) to exercise the option. At each exercise date t, we get the pay-off \((S_t - K)^+\) where \(S_t\) is the value of the underlying asset at date \(t\). See [23] for description of the swing problem and the backward resolution techniques. Due to classical results on the Snell envelope for European payoff, the analytical value of this problem is the sum of the \(N\) payoff at the \(N\) last dates where we can exercise (recall that the value of an American call is the value of the European one). The Markov state of the problem at a given date \(t\) is given by the value of the underlying (Markov) and the number of exercises
already achieved at date \( t \). This test case can be run in parallel with MPI. In all test cases, we use a “LocalLinearRegression” to evaluate the conditional expectations used during the Dynamic Programming approach.

**testSwingOptionInOptimization**

After having calculated the analytical solution for this problem,

- a first resolution is provided using the “resolutionSwing” function. For this simple problem, only a regressor is necessary to decide if we exercise at the current date of not.

- a second resolution is provided in the “resolutionSwingContinuation” function using the “Continuation” object (see chapter 5) permitting to store continuation values for a value of the underlying and for a stock level. This example is provide here to show how to use this object on a simple test case. This approach is here not optimal because getting the continuation value for an asset value and a stock level (only discrete here) means some unnecessary interpolation on the stock grids (here we choose a “RegularSpaceGrid” to describe the stock level and interpolate linearly between the stock grids). In the case of swing with varying quantities to exercise or the gas storage problem, this object is very useful,

- A last resolution is provided using the general framework described and the “DynamicProgrammingByRegressionDist” function described in subsection 7.2.3. Once again the framework is necessary for this simple test case, but it shows that it can be used even for some very simple cases.

**13.2.1 testSwingOption2D**

Here we suppose that we have two similar swing options to price and we solve the problem ignoring that the stocks are independent: this means that we solve the problem on a two dimensional grid (for the stocks) instead of two times the same problem on a grid with one stock.

- we begin by an evaluation of the solution for a single swing with the “resolutionSwing” function giving a value \( A \).

- then we solve the 2 dimensional (in stock) problem giving a value \( B \) with our framework with the “DynamicProgrammingByRegressionDist” function.

Then we check that \( B = 2A \).

**13.2.2 testSwingOption3**

We do the same as previously but the management of three similar swing options is realized by solving as a three dimensional stock problem.
13.2.3 testSwingOptimSimu / testSwingOptimSimuMpi

This test case takes the problem described in section 13.2 solves it using the framework. Once the optimization using regression ("LocalLinearRegression" regressor) is achieved, a simulation part is used using the previously calculated Bellman values. We check the the values obtained in optimization and simulation are close. The two test case files (testSwingOptimSimu/testSwingOptimSimuMpi) use the two versions of MPI parallelization distributing or not the data on the processors.

13.2.4 testSwingOptimSimuWithHedge

The test case takes the problem described in section 13.2 solves it using regression ("LocalLinearRegression" regressor) while calculating the optimal hedge by the conditional tangent method as explained in [42]. After optimization, a simulation part implement the optimal control and the optimal hedge associated. We check :

- That values in optimization and simulation are close
- That the hedge simulated has an average nearly equal to zero,
- That the hedged swing simulations give a standard deviation reduced compared to the non hedged option value obtained by simulation without hedge.

This test case shows are that the multiple regimes introduced in the framework can be used to calculate and store the optimal hedge. This is achieved by the creation of a dedicated optimizer “OptimizeSwingWithHedge”.

13.2.5 testSwingOptimSimuND / testSwingOptimSimuNDMpi

The test case takes the problem described in section 13.2 suppose that we have two similar options to valuate and that we ignore that the options are independent giving a problem to solve with two stocks managed jointly as in subsection 13.2.1. After optimizing the problem using regression ("LocalLinearRegression" regressor) we simulate the optimal control for this two dimensional problem and check that values in optimization and simulation are close. In "testSwingOptimSimuND" Mpi parallelization, if activated, only parallelize the calculation, while in “testSwingOptimSimuNDMpi” the data are also distributed on processors. In the latter, two options are tested,

- in “testSwingOptionOptim2DSimuDistOneFile” the Bellman values are distributed on the different processors but before being dumped they are recombine to give a single file for simulation.
- in “testSwingOptionOptim2DSimuDistMultipleFile” the Bellman values are distributed on the different processors but each processor dumps its own Bellman Values. During the simulation, each processor rereads its own Bellman values.

In the same problem in high dimension may be only feasible with the second approach.


13.3 Gas Storage

13.3.1 testGasStorage / testGasStorageMpi

The model used is a mean reverting model similar to the one described in [42]. We keep only one factor in equation (8) in [42]. The problem consists in maximizing the gain from a gas storage by the methodology described in [42]. All test cases are composed of three parts:

- an optimization is realized by regression ("LocalLinearRegression" regressor),
- a first simulation of the optimal control using the continuation values stored during the optimization part,
- a second simulation directly using the optimal controls stored during the optimization part.

We check that the three previously calculated values are close.

Using dynamic programming method, we need to interpolate into the stock grid to get the Bellman values at one stock point. Generally a simple linear interpolator is used (giving a monotone scheme). As explicated in [44], it is possible to use higher order schemes still being monotone. We test different interpolators. In all test case we use a “LocalLinearRegression” to evaluate the conditional expectations. The MPI version permits to test the distribution of the data when using parallelization.

testSimpleStorage

We use a classical regular grid with equally spaces points to discretize the stock of gas and a linear interpolator to interpolate in the stock.

testSimpleStorageLegendreLinear

We use a Legendre grid with linear interpolation, so the result should be the same as above.

testSimpleStorageLegendreQuadratic

We use a quadratic interpolator for the stock level.

testSimpleStorageLegendreCubic

We use a cubic interpolator for the stock level.

testSimpleStorageSparse

We use a sparse grid interpolator (equivalent to a full grid interpolator because it is a one dimensional problem). We only test the sparse grid with a linear interpolator.
13.3.2 testGasStorageCut / testGasStorageCutMpi

We take the previous gas storage problem and solve the transition problem without discretizing the command by using a LP solver (see section 7.2.3). The test cases are composed of an optimization part followed by a simulation part comparing the results obtained.

testSimpleStorageCut

Test case without mpi distribution of the stocks points. A simple Regular grid object is used and conditionalcust are calculated using Local Linear Regressions.

testSimpleStorageCutDist

Test using MPI distribution. In all cases a Local Linear Regressor is used. One file is used to store the conditional cuts.

- A first case uses a Regular grid,
- A second case used a RegularLegendre grid.

testSimpleStorageMultipleFileCutDist

Test using MPI distribution. A Local Linear Regressor is used for cuts and a Regular grid is used. Bender cuts are locally stored by each processor.

13.3.3 testGasStorageKernel

The model used is a mean reverting model similar to the one described in [42]. We keep only one factor in equation (8) in [42]. The problem consists in maximizing the gain from a gas storage by the methodology described in [42]. The specificity here is that a kernel regression method is used.

testSimpleStorageKernel

Use the linear kernel regression method to solve the Gas Storage problem.

13.3.4 testGasStorageVaryingCavity

The stochastic model is the same as in section 13.3.1. As previously, all test cases are composed of three parts:

- an optimization is realized by regression (“LocalLinearRegression” regressor),
- a first simulation of the optimal control using the continuation values stored during the optimization part,
- a second simulation directly using the optimal controls stored during the optimization part.
We check that the three previously calculated values are close on this test case where the grid describing the gas storage constraint is time varying. This permits to check the splitting of the grids during parallelization.

### 13.3.5 testGasStorageSwitchingCostMpi

The test case is similar to the one in section 13.3.1 (so using regression methods) : we added some extra cost when switching from each regime to the other. The extra cost results in the fact that the Markov state is composed of the asset price, the stock level and the current regime we are (the latter is not present in other test case on gas storage). This test case shows that our framework permits to solve regime switching problems. As previously all test cases are composed of three parts :

- an optimization is realized by regression ("LocalLinearRegression" regressor),
- a first simulation of the optimal control using the continuation values stored during the optimization part,
- a second simulation directly using the optimal controls stored during the optimization part.

We check that the three previously calculated values are close.

### 13.3.6 testGasStorageSDDP

The modelization of the asset is similar to the other test case. We suppose that we have $N$ similar independent storages. So solving the problem with $N$ stocks should give $N$ times the value of one stock.

- First the value of the storage is calculated by dynamic programming giving value $A$,
- then the SDDP method (chapter 12) is used to valuate the problem giving the $B$ value. The Benders cuts have to be done conditionally to the price level.

We check that $B$ is close to $NA$.

#### testSimpleStorageSDDP1D

Test the case $N = 1$.

#### testSimpleStorageSDDP2D

Test the case $N = 2$.

#### testSimpleStorageSDDP10D

Test the case $N = 10$. 
13.4 testLake / testLakeMpi

This is the case of a reservoir with inflows following an AR1 model. We can withdraw water from the reservoir (maximal withdrawal rate given) to produce energy by selling it at a given price (taken equal to 1 by unit volume). We want to maximize the expected earnings obtained by an optimal management of the lake. The problem permits to show how some stochastic inflows can be taken into account with dynamic programming with regression ("LocalLinearRegression" regressor used).

The test case is compose of three parts:

- an optimization is realized by regression ("LocalLinearRegression" regressor),
- a first simulation of the optimal control using the continuation values stored during the optimization part,
- a second simulation directly using the optimal controls stored during the optimization part.

We check that the three previously calculated values are close.

13.5 testOptionNIGL2

In this test case we suppose that the log of an asset value follows an NIG process. We want to price a call option supposing that we use the mean variance criterium using the algorithm developped in chapter 9. First an optimization is achieved then in a simulation part the optimal hedging strategy is tested.

13.6 testDemandSDDP

This test case is the most simple using the SDDP method. We suppose that we have a demand following an AR 1 model

\[ D^{n+1} = k(D^n - D) + \sigma_d g + kD, \]

where \( D \) is the average demand, \( \sigma_d \) the standard deviation of the demand on one time step, \( k \) the mean reverting coefficient, \( D^0 = D \), and \( g \) a unit centered Gaussian variable. We have to satisfy the demand by buying energy at a price \( P \). We want to calculate the following expected value

\[ V = P \mathbb{E}(\sum_{i=0}^{N} D_i) \]

\[ = (N + 1)D_0P \]

This can be done (artificially) using SDDP.
13.7 Reservoir variations with SDDP

13.7.1 testReservoirWithInflowsSDDP

For this SDDP test case, we suppose that we dispose of $N$ similar independent reservoirs with inflows given at each time time by independent centered Gaussian variables with standard deviation $\sigma_i$. We suppose that we have to satisfy at $M$ dates a demand given by independent centered Gaussian variables with standard deviation $\sigma_d$. In order to satisfy the demand, we can buy some water with quantity $q_t$ at a deterministic price $S_t$ or withdraw water from the reservoir at a pace lower than a withdrawal rate. Under the demand constraint, we want to minimize:

$$E\left(\sum_{t=0}^{M} q_t S_t\right)$$

Each time we check that forward and backward methods converge to the same value. Because of the independence of uncertainties the dimension of the Markov state is equal to $N$.

testSimpleStorageWithInflowsSDDP1DDeterministic

$\sigma_i = 0$ for inflows and $\sigma_d = 0$. for demand. $N$ taken equal to 1.

testSimpleStorageWithInflowsSDDP2DDeterministic

$\sigma_i = 0$ for inflows and $\sigma_d = 0$. for demand. $N$ taken equal to 2.

testSimpleStorageWithInflowsSDDP5DDeterministic

$\sigma_i = 0$ for inflows and $\sigma_d = 0$. for demand. $N$ taken equal to 5.

testSimpleStorageWithInflowsSDDP1D

$\sigma_i = 0.6$, $\sigma_d = 0.8$ for demand. $N = 1$

testSimpleStorageWithInflowsSDDP2D

$\sigma_i = 0.6$ for inflows, $\sigma_d = 0.8$ for demand. $N = 2$
testSimpleStorageWithInflowsSDDPD

$\sigma_i = 0.6$ for inflows, $\sigma_d = 0.8$ for demand. $N = 5.$

13.7.2 testStorageWithInflowsSDDP

For this SDDP test case, we suppose that we dispose of $N$ similar independent reservoirs with inflows following an AR1 model:

$$X^{n+1} = k(X^n - X) + \sigma g + X,$$

with $X^0 = X$, $\sigma$ the standard deviation associated, $g$ some unit centered Gaussian variable.

We suppose that we have to satisfy at $M$ dates a demand following an AR1 process too. In order to satisfy the demand, we can buy some water with quantity $q_t$ at a deterministic price $S_t$ or withdraw water from the reservoir at a pace lower than a withdrawal rate. Under the demand constraint, we want to minimize:

$$\mathbb{E}\left(\sum_{i=0}^{M} q_t S_t\right)$$

Each time we check that forward and backward methods converge to the same value. Because of the structure of the uncertainties the dimension of the Markov state is equal to $2N + 1$ ($N$ storage, $N$ inflows, and demand).

testSimpleStorageWithInflowsSDDPDeterministic

All parameters $\sigma$ are set to 0. $N = 1.$

testSimpleStorageWithInflowsSDDP1DDeterministic

All parameters $\sigma$ are set to 0. $N = 2.$

testSimpleStorageWithInflowsSDDP5DDeterministic

All parameters $\sigma$ are set to 0. $N = 5.$

testSimpleStorageWithInflowsSDDP10DDeterministic

All parameters $\sigma$ are set to 0. $N = 10.$

testSimpleStorageWithInflowsSDDP1DDeterministic

$\sigma = 0.3$ for inflows, $\sigma = 0.4$ for demand. $N = 1.$

testSimpleStorageWithInflowsSDDP5DDeterministic

$\sigma = 0.3$ for inflows, $\sigma = 0.4$ for demand. $N = 5.$
### 13.7.3 testStorageWithInflowsAndMarketSDDP

This is the same problem as [13.7.2](#) but the price $S_t$ follow an AR 1 model. We use a SDDP approach to solve this problem. Because of the price dependencies, the SDDP cut have to be done conditionally to the price level.

**testSimpleStorageWithInflowsAndMarketSDDP1DDeterministic**

All volatilities set to 0. $N = 1$.

**testSimpleStorageWithInflowsAndMarketSDDP2DDeterministic**

All volatilities set to 0. $N = 2$.

**testSimpleStorageWithInflowsAndMarketSDDP5DDeterministic**

All volatilities set to 0. $N = 5$.

**testSimpleStorageWithInflowsAndMarketSDDP10DDeterministic**

All volatilities set to 0. $N = 10$.

**testSimpleStorageWithInflowsAndMarketSDDP1D**

$\sigma = 0.3$ for inflows, $\sigma = 0.4$ for demand, $\sigma = 0.6$ for the spot price. $N = 1$.

**testSimpleStorageWithInflowsAndMarketSDDP5D**

$\sigma = 0.3$ for inflows, $\sigma = 0.4$ for demand, $\sigma = 0.6$ for the spot price. $N = 5$.

### 13.8 Semi-Lagrangian

#### 13.8.1 testSemiLagrangCase1/testSemiLagrangCase1

Test Semi-Lagrangian deterministic methods for HJB equation. This corresponds to the second test case without control in [44](#) (2 dimensional test case).

**TestSemiLagrang1Lin**

Test the Semi-Lagrangian method with the linear interpolator.

**TestSemiLagrang1Quad**

Test the Semi-Lagrangian method with the quadratic interpolator.

**TestSemiLagrang1Cubic**

Test the Semi-Lagrangian method with the cubic interpolator.
TestSemiLagrang1SparseQuad
Test the sparse grid interpolator with a quadratic interpolation.

TestSemiLagrang1SparseQuadAdapt
Test the sparse grid interpolator with a quadratic interpolation and some adaptation in the meshing.

13.8.2 testSemiLagrangCase2/testSemiLagrangCase2
Test Semi-Lagrangian deterministic methods for HJB equation. This corresponds to the first case without control in [44] (2 dimensional test case).

TestSemiLagrang2Lin
Test the Semi-Lagrangian method with the linear interpolator.

TestSemiLagrang2Quad
Test the Semi-Lagrangian method with the quadratic interpolator.

TestSemiLagrang2Cubic
Test the Semi-Lagrangian method with the cubic interpolator.

TestSemiLagrang2SparseQuad
Test the sparse grid interpolator with a quadratic interpolation.

13.8.3 testSemiLagrangCase2/testSemiLagrangCase2
Test Semi-Lagrangian deterministic methods for HJB equation. This corresponds to the stochastic target test case 5.3.4 in [44].

TestSemiLagrang3Lin
Test the Semi-Lagrangian method with the linear interpolator.

TestSemiLagrang3Quad
Test the Semi-Lagrangian method with the quadratic interpolator.

TestSemiLagrang3Cubic
Test the Semi-Lagrangian method with the cubic interpolator.
13.9 Non emissive test case

13.9.1 testDPNonEmissive
Solve the problem described in part [V] by dynamic programming and regression.

- first an optimization is realized,
- the an simulation part permit to test the controls obtained.

13.9.2 testSLNonEmissive
Solve the problem described in part [V] by the Semi-Lagrangian method.

- first an optimization is realized,
- the an simulation part permit to test the controls obtained.

13.10 Nesting for Non Linear PDE’s

13.10.1 Some HJB test
The control problem where $A$ is the set of adapted integrable processes.

$$dX = 2\sqrt{\theta}\alpha dt + \sqrt{2}dW_t, \quad V = \inf_{\alpha \in A} E\left[\int_0^T |\alpha_s|^2 dt + g(X_T)\right]$$

The HJB equation corresponding

$$(-\partial_t u - Lu)(t, x) = f(Du(t, x))$$

$$L u(t, x) := \mu Du(t, x) + \frac{1}{2} \sigma \sigma^T : D^2 u(t, x), \quad f(z) = -\theta ||z||_2^2 \quad (13.1)$$

such that a solution is

$$u(t, x) = -\frac{1}{\theta} \log \left( E[e^{-\theta g(x+\sqrt{2}W_{T-1})}] \right). \quad (13.3)$$

We use the nesting method with $\mu = 0$, $\sigma = \sqrt{2}I_d$. These test case are located in the “test/c++/unit/branching” directory.

testHJCConst
In this test case, we use a special resolution function supposing that the parameters of the PDE are constant : this permits us to precalculate the inverse of some matrices.
We test here the special case where the SDE can be exactly simulated with a scheme
\[ X_{t+dt} = A(t, dt)X_t + B(t, dt) + C(t, dt)g \]
with \( g \) Gaussian centered unit vector.

We use a resolution function supposing that the SDE is discretized by an Euler scheme.

### 13.10.2 Some Toy example : testUD2UTou

We want to solve:
\[ (-\partial_t u - \mathcal{L}u)(t, x) = f(u, Du(t, x), D^2 u(t, x)) \]
with
\[
\begin{align*}
\mu &= \mu_0 \frac{d}{\sigma^2} I_d, \\
\sigma &= \sigma_0 \frac{d}{\sqrt{d}} I_d, \\
f(t, x, y, z, \theta) &= \cos(\sum_{i=1}^d x_i)(\alpha + \frac{1}{2} \sigma^2)\exp(\alpha(t-t)) + \sin(\sum_{i=1}^d x_i)\mu_0 \exp(\alpha(t-t)) + a \sqrt{d} \cos(\sum_{i=1}^d x_i)^2 \exp(2\alpha(t-t)) \\
&\quad + \frac{a}{\sqrt{d}} (e^{2\alpha(t-t)} \vee e^{-2\alpha(t-t)}) \vee (e^{2\alpha(t-t)} \land (y \sum_{i=1}^d \theta_{i,i})),
\end{align*}
\]
with a solution
\[ u(t, x) = e^{\alpha(t-t)} \cos(\sum_{i=1}^d x_i) \]

### 13.10.3 Some Portfolio optimization

We assume that we dispose of \( d = 4 \) securities all of them being defined by a Heston model:
\[
\begin{align*}
\text{for } i = 1, \ldots, d:
\frac{dS^i_t}{S^i_t} &= \mu^i S^i_t dt + \sqrt{Y^i_t S^i_t} dW^{(2i-1)}_t, \\
\frac{dY^i_t}{Y^i_t} &= k^i (m^i - Y^i_t) dt + \sigma^i \sqrt{Y^i_t} dW^{(2i)}_t,
\end{align*}
\]
where \( W = (W^{(1)}, \ldots, W^{(2d)}) \) is a Brownian motion in \( \mathbb{R}^{2d} \).

The non-risky asset \( S^0 \) has a 0 return so \( dS^0_t = 0, t \in [0, 1] \).

The investor chooses an adapted process \( \{\kappa_t, t \in [0, T]\} \) with values in \( \mathbb{R}^n \), where \( \kappa^i_t \) is the amount he decides to invest into asset \( i \).

The portfolio dynamic is given by:
\[ dX^\kappa_t = \kappa_t \cdot \frac{dS^\kappa_t}{S_t} + (X^\kappa_t - \kappa_t \cdot 1) \frac{dS^0_t}{S^0_t} = \kappa_t \cdot \frac{dS_t}{S_t}. \]

Let \( \mathcal{A} \) be the collection of all adapted processes \( \kappa \) with values in \( \mathbb{R}^d \) and which are integrable with respect to \( S \). Given an absolute risk aversion coefficient \( \eta > 0 \), the portfolio optimization problem is defined by:
\[
u_0 := \sup_{\kappa \in \mathcal{A}} \mathbb{E}[-\exp(-\eta X^\kappa_T)]. \tag{13.4}
\]
The problem doesn’t depend on the $s^i$. As in [49], we can guess that the solution can be expressed as
\[ v(t, x, y^1, ..., y^d) = e^{-\eta x}u(y^1, ..., y^d) \]
and using Feynman Kac it is easy to see that then a general solution can be written
\[ v(t, x, y) = -e^{-\eta x}\mathbb{E}[\prod_{i=1}^{d} \exp \left( -\frac{1}{2} \int_t^T \frac{(\mu^i)^2}{\tilde{Y}_i^s} ds \right)] \]
with
\[ \tilde{Y}_i^s = y^i \quad \text{and} \quad d\tilde{Y}_i^s = k^i(m^i - \tilde{Y}_i^s)dt + c^i\sqrt{\tilde{Y}_i^s}dW_i^s, \]
where $y^i$ corresponds to the initial value of the volatility at date 0 for asset $i$.

We suppose in our example that all assets have the same parameters that are equal to the parameters taken in the two dimensional case. We also suppose that the initial conditions are the same as before.

Choosing $\bar{\sigma} > 0$, we can write the problem as equation (12.23) in dimension $d + 1$ where
\[ \mu = (0, k^1(m^1 - y^1), ..., k^d(m^d - y^d))^\top, \quad \sigma = \begin{pmatrix} \bar{\sigma} & 0 & ... & ... & 0 \\ 0 & c\sqrt{m^1} & 0 & ... & 0 \\ 0 & ... & \ddots & ... & 0 \\ 0 & ... & ... & 0 & 0 \\ 0 & ... & ... & 0 & c\sqrt{m^d} \end{pmatrix} \]
always with the same terminal condition
\[ g(x) = -e^{-\eta x} \]
and
\[ f(x, y, z, \theta) = -\frac{1}{2}\sigma^2\theta_{11} + \frac{1}{2} \sum_{i=1}^{d} (c^i)^2((y^i)^2 - m^i)\theta_{i+1,i+1} - \sum_{i=1}^{d} \frac{\mu^i z_1}{2y^i\theta_{11}}. \]

In order to have $f$ Lipschitz, we truncate the control limiting the amount invested by taking
\[ f_M(y, z, \theta) = -\frac{1}{2}\sigma^2\theta_{11} + \frac{1}{2} \sum_{i=1}^{d} (c^i)^2((y^i)^2 - m^i)\theta_{2,2} + \sup_{\eta = (\eta^1, ..., \eta^d)} \sum_{i=1}^{d} \left( \frac{1}{2}(\eta^i)^2y^i\theta_{11} + (\eta^i)\mu^i z_1 \right), \]
\[ 0 \leq \eta^i \leq M, \quad i = 1, d \]

**testPortfolioExact**

The Ornstein Uhlenbeck process used as a driving process is simulated exactly.

**testPortfolioEuler**

The Ornstein Uhlenbeck process used as a driving process is simulated using an Euler scheme.
Chapter 14

Some python test cases description

This part is devoted to some test cases only available in python. These examples uses the low level python interface.

14.1 Microgrid Management

14.1.1 testMicrogridBangBang

A microgrid is a collection of renewable energy sources, a diesel generator, and a battery for energy storage. The objective is to match the residual demand (difference between the demand of electricity and supply from renewables) while minimizing the total expected cost of running the microgrid. In particular a penalty is assessed for insufficient supply that leads to blackouts. The setup is similar to the one described in [26, Section 7]. We take the diesel generator as the only control \( d_t \); output/input from/into the battery is then a function of the residual demand \( X_t \) (exogenous stochastic process), inventory level of the battery \( I_t \), and \( d_t \). The diesel generator operates under two regimes: OFF and ON. When it is OFF it does not supply power \( d_t = 0 \), however when it is ON the power output is a deterministic function of the state \( d_t = d(X_t, I_t) \). As a result, the problem is a standard stochastic control model with switching-type bang-bang control.

We parameterize the algorithm to easily switch between multiple approximation schemes for the conditional expectation at the core of the Dynamic Programming equation. Particularly the following schemes are implemented:

- Regularly spaced grid for \( I_t \) and local polynomial basis in \( X_t \) for each level of the grid.
- Adaptive piecewise-defined polynomial basis in -2D for \( (X_t, I_t) \).
- Global 2D polynomial basis on \( (X_t, I_t) \).
- Bivariate Kernel regression on \( (X_t, I_t) \).

14.1.2 testMicrogrid

We extend the previous example to include the recent work [1] where the action space for the control is \( d_t \in \{0\} \cup [1,10] \) kW, rather than being bang-bang. As a result, the optimal
control is chosen in two steps: first the controller picks the regime: ON or OFF; if ON, she then decides the optimal, continuous level of the diesel output. Due to the additional flexibility available to the controller compared to the previous example, we expect to observe lower cost compared to Section 14.1.1. The user can switch between this and the previous setting by changing the parameter “controlType” in the “parameters.py” file.

14.2 Dynamic Emulation Algorithm (DEA)

14.2.1 testMicrogridDEA

In this section we discuss the implementation of the Dynamic Emulation Algorithm developed in [26]. In that paper the authors reformulate the stochastic control problem as an “iterative sequence of machine learning tasks”. The philosophy of DEA is to combine together Regression Monte Carlo (RMC) with Design of Experiments. The algorithm has the following properties:

- The learning for the continuation value function at each step in the backward-iteration of the Dynamic Programming Equation is completely modularized. As a result, the user can seamlessly switch between different regression schemes (for example: adaptive local polynomial basis in -2D or -1D, multivariate kernel regression, etc.) across different time-steps;

- The empirical approximation uses distinct designs $D_t$ at each $t$; thus the user can have design sites independently chosen for different $t$’s, which also eliminates the requirement to store the full history of the simulated paths of $(X_t)$. One-step paths can now replace the full history. In Figure 14.1 we present examples of two possible designs we use in the implementation. The image in the left panel represents a space-filling design using a Sobol sequence in -2D. This design is appropriate for a bivariate regression over $(X_t, I_t)$. On the right, we present another space-filling design in -1D with a regularly spaced grid in $I_t$ (y-axis) and a -1D Sobol sequence in $X_t$ (x-axis). In [26] the authors discuss several further designs which can be easily implemented.

- Batched designs, i.e. a partially nested scheme that generates multiple $X_t$-paths from the same unique design site, can be accommodated.

- Simulation budget (i.e. the size of $D_t$) can vary through the time-steps and need not be fixed as in standard RMC.

Several different experiments have confirmed the significant effect of the design $D_t$ on the performance of the RMC algorithms. DEA allows us to test for this effect by allowing the user to easily specify $D_t$. The structure of this library allows for easy implementation of such modular algorithms. As a proof of concept, we re-implement the microgrid example of Section 14.1.1 with the following specifications:

- The 10 time-steps (25% of the total of 40 time-steps) closest to maturity use adaptive local polynomial basis in -1D with gridded design similar to the Figure 14.1b. Moreover, for these $t$’s we used $|D_t| = 22,000 = N_t$ unique design sites;
• The other 30 steps (first 75%) use design sites allocated according to Sobol-2D as in figure 14.1a with a global polynomial basis regression scheme. For these, we build a batched design of 1000 unique sites, each replicated 10 times for a total simulation budget of $N_t = 1000 \times 10 = 10^4$. 

Figure 14.1: Illustration of two simulation designs. In both panels the $X_t$-coordinate is on the x-axis and $I_t$ on the y-axis.
Bibliography


[40] Stéphane Vialle, Xavier Warin, Constantinos Makassikis, and Patrick Mercier. Stochastic control optimization & simulation applied to energy management: From 1-d to nd problem distributions, on clusters, supercomputers and grids. In *Grid@ Mons conference*, 2008.


