STochastic OPTimization library in C++

Hugo Gevret, Nicolas Langrené, Jérôme Lelong, Rafael D Lobato, Thomas Ouillon, Xavier Warin, Aditya Maheshwari

STochastic OPTimization library in C++

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

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

[5]EDF R&D, Thomas.Ouillon@edf.fr

[6]EDF R&D & FiME, Laboratoire de Finance des Marchés de l'Energie, ANR PROJECT CAESARS, Xavier.Warin@edf.fr

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

Contents

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 on 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`
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) or using a scenario tree 5.
  Regression are achieved:

  1. either by local polynomials either with basis support with the same size (subsection 4.4) or with an adapted size of the support (subsection 4.2) ,
  2. either by global polynomials (section 4.7)
  3. either by sparse grid regression (section 4.6) useful in high dimension
  4. or by kernel regression (section4.8)

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

- Gas storage problems are solved

  - either by dynamic programming (part III) in python or C++ using regression (section 4) or tree (section 5) 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 4.7)
    3. or kernel regression (section 4.8)

    As before the trinomial tree developed in tests is used in tree methods.
    Interpolation between stock points is either linear or quadratic.
  - either by the SDDP method (chapter 14) in C++ using both regressions and tree methods.

- 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 criterion of the hedged portfolio is solved in C++ by dynamic programming (part III) using the methodology in chapter 11.
- Some reservoir management is solved by the SDDP method (chapter 14) 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 4.7)
    3. or kernel regression (section 4.8)

Chapter 2

General mathematical setting

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

$$\left\{ \begin{array}{rcl} dX_s^{x,t} & = & b_a(t, X_s^{x,t})ds + \sigma_a(s, X_s^{x,t})dW_s \\ X_t^{x,t} & = & x \end{array} \right.$$

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 $T > 0$,
- σ_a is a Lipschitz continuous function of (t, x, a) defined on $[0, T] \times \mathbb{R}^d \times \mathbb{R}^n$ and taking values in the set of d-dimensional square matrices,
- b_a is a Lipschitz continuous function of (t, x, a) defined on $[0, T] \times \mathbb{R}^d \times \mathbb{R}^n$ and taking values in $\mathbb{R}^d$,
- a a control adapted to the filtration taking values in $\mathbb{R}^n$.

Suppose we want to minimize a cost function $J(t, x, a) = \mathbb{E}[\int_t^T f_a(s, X_s^{x,t}) e^{\int_t^s c_a(u, X_u^{x,t})du} ds + e^{\int_t^T c_a(u, X_u^{x,t})} g(X_T^{x,t})]$ 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

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

Under some classical assumptions on the coefficients [15], the previous equation known as the Hamilton Jacobi Bellman equation admits an unique viscosity solution ([24]).
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 $X_s^{x,t} = (X_{1,s}^{x,t}, X_{2,s}^{x,t})$ where $X_{1,s}^{x,t}$ is not controlled

$$\left\{ \begin{array}{lcl} dX_{1,s}^{x,t} & = & b(t, X_{1,s}^{x,t})ds + \sigma_(s, X_{1,s}^{x,t})dW_s \\ X_{1,t}^{x,t} & = & x \end{array} \right. \tag{2.2}$$

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

$$\left\{ \begin{array}{lcl} dX_{2,s}^{x,t} & = & b_a(t, X_{2,s}^{x,t})ds \\ X_{2,t}^{x,t} & = & x \end{array} \right.$$

  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.
  A second approach based on Dynamic Programming uses scenario trees: in this case, uncertainties evolve on a tree only taking 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. As before, uncertainties can be either described by scenarios or by a scenario tree.

  Remark 1 *The use of this method requires other assumptions that will be described 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 relies on the Eigen library:* `ArrayXd` *stands for a vector of double,* `ArrayXXd` *for a matrix of double and* `ArrayXi` *a vector of integer.*

Part II

Useful tools for stochastic control

Chapter 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 point 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. These 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 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`

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

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

All the iterators share some common features:

- the `getCount` method permits to get the number associated to the current grid point,
- the `next` method permits to go to the next point, while the `nextInc` method permits to jump forward to the `p_incr` point,
- the `isValid` method permits to check that we are still on a grid point,
- the `getNbPointRelative` method permits to get the number of points that a given iterator can iterate on,
- the `getRelativePosition` get the number of points already iterated by the iterator.

Besides, we can directly jump to a given point: this feature is useful for "mpi" when a treatment on the grid is split between some processor and threads. This possibility is given by the `jumpToAndInc` method.
Using a grid `regGrid` the following source code permits to iterate on the points of the grids and get coordinates. For each coordinate, a function f is used to fill in an array of values. As pointed out before, each type of grid has its own grid iterator that can be obtained by the `getGridIterator` method.

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

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

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

```
// Copyright (C) 2016 EDF
// All Rights Reserved
// This code is published under the GNU Lesser General Public License (GNU LGPL)
#ifndef INTERPOLATOR_H
#define INTERPOLATOR_H
#include <vector>
#include <Eigen/Dense>
/** \file Interpolator.h
 *  \brief Defines a  interpolator on a full grid
 * \author Xavier Warin
 */
namespace StOpt
{

/// \class Interpolator Interpolator.h
///  Interpolation  base class
class Interpolator
{
public :

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

    /// \brief Default Destructor
    virtual ~Interpolator() {}

    /**  \brief  interpolate
     *  \param  p_dataValues   Values of the data on the grid
     *  \return interpolated value
     */
    virtual  double apply(const Eigen::Ref< const Eigen::ArrayXd > &p_dataValues) const = 0;

    /**  \brief  interpolate and use vectorization
    *  \param  p_dataValues   Values of the data on the grid. Interpolation is achieved for all values in the first dimension
    *  \return interpolated value
    */
    virtual  Eigen::ArrayXd applyVec(const Eigen::ArrayXXd &p_dataValues) const = 0;

    /** \brief  Same as above but avoids copy for Numpy eigen mapping due to storage conventions
     *  \param  p_dataValues   Values of the data on the grid. Interpolation is achieved for all values in the first dimension
     *  \return interpolated value
     */
```

```
43      virtual  Eigen::ArrayXd applyVecPy(Eigen::Ref< Eigen::ArrayXXd, 0, Eigen::Stride<Eigen
            ::Dynamic, Eigen::Dynamic> >  p_dataValues) const = 0;
44
45  };
46  }
47  #endif
```

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 `SpaceGrid` 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.

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

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

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` with constructor

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

  where `std::vector<shared_ptr<Eigen::ArrayXd>>` is a vector of (pointer 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).

- the second one is the `RegularSpaceGrid` with constructor

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

  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)

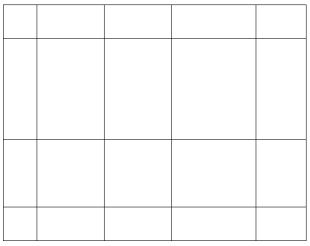


Figure 3.1: 2D general grid

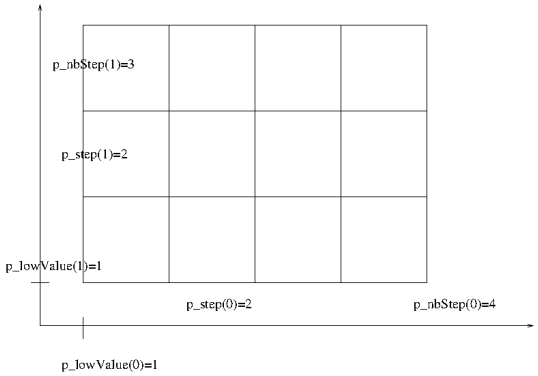


Figure 3.2: 2D regular grid

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

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

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

```
    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 $k \leq 1$

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

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

```
# 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

```
# Copyright (C) 2017 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 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)
        interpol = grid.createInterpolator(ptInterp)
        # calculate interpolated value
        interpValue = interpol.apply(data)
        # test grids function
        iDim = grid.getDimension()
        pt = grid.getExtremeValues()


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

3.2 Legendre grids

With linear interpolation, in order to get an accurate solution, it is needed to refine the mesh so that Δx goes 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_N^X interpolator from f on a grid of $N+1$ points of $[-1,1]$ $X = (x_0, \ldots, x_N)$, the unique polynomial of degree N such that

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

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

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

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

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

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

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

We have the following bound

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

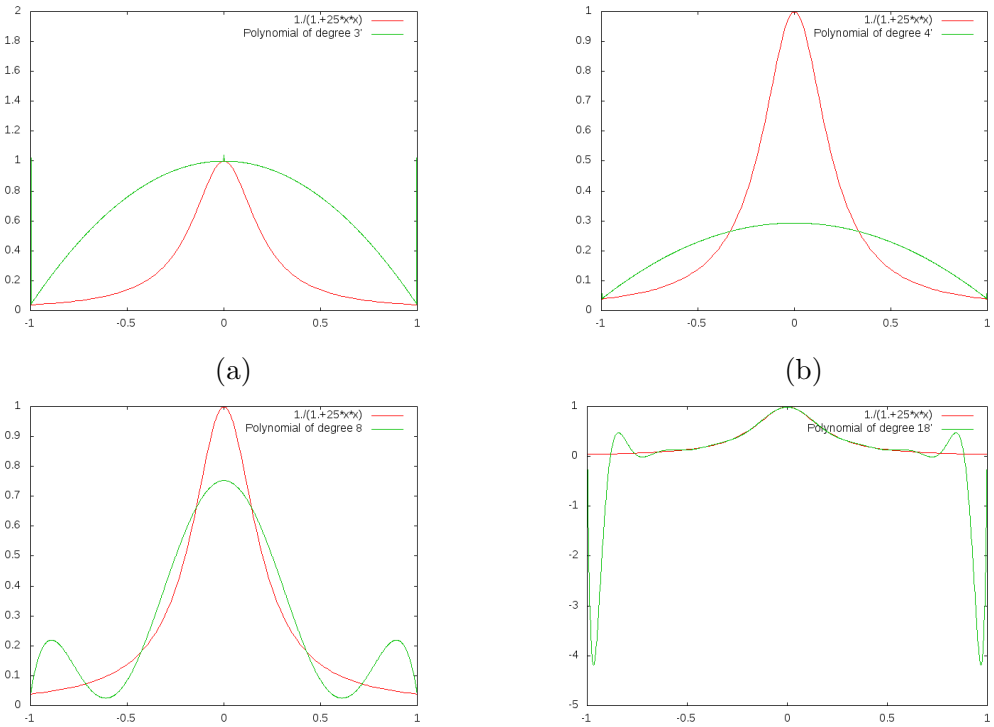
and the Erdös theorem states that

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

It is well–known that the use of a uniform grid X_u is not optimal, because as $N \longrightarrow \infty$, the Lebesgue constant satisfies

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

and the quadrature error in L_∞ increases a lot with N. Its use brings some oscillations giving the Runge effect. On Figures 3.3a, 3.3b, 3.3c, 3.3d, we plot the Runge function $\frac{1}{1+25x^2}$ against its interpolation with polynomial with equidistant interpolation. So we are interested in hav-



(a) (b) (c) (d)

Figure 3.3: Runge function $\frac{1}{1+25x^2}$ and its polynomial interpolations with degrees 3, 4, 8, and 18.

ing 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 η_i $(i = 2, \ldots, N)$ zeros of $L_N^{'}$. The η_i $(i = 2, \ldots, N)$ are eigenvalues of the matrix P

$$
\begin{aligned}
P &= \begin{pmatrix} 0 & \gamma_1 & \ldots & 0 & 0 \\ \gamma_1 & 0 & \ldots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \ldots & 0 & \gamma_{N-2} \\ 0 & 0 & \ldots & \gamma_{N-2} & 0 \end{pmatrix}, \\
\gamma_n &= \frac{1}{2}\sqrt{\frac{n(n+2)}{(n+\frac{1}{2})(n+\frac{3}{2})}}, 1 \leq n \leq N-2,
\end{aligned}
$$

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

$$
\begin{aligned}
I_N(f) &= \sum_{k=0}^{N} \tilde{f}_k L_k(x), \\
\tilde{f}_k &= \frac{1}{\gamma_k} \sum_{i=0}^{N} \rho_i f(\eta_i) L_k(\eta_i), \\
\gamma_k &= \sum_{i=0}^{N} L_k(\eta_i)^2 \rho_i,
\end{aligned}
$$

and the weights satisfies

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

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

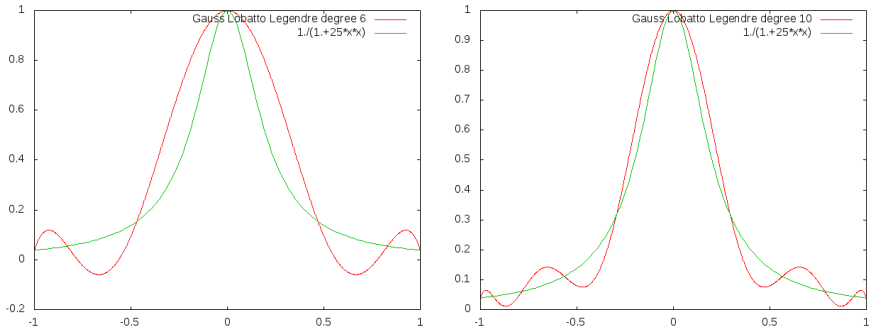


Figure 3.4: Interpolation with Gauss–Legendre–Lobatto grids

- 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

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

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

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

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

- 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 [39])

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

  $$E(x) = f(x) - I_N^X(f)(x) = \frac{f^{N+1}(\eta)}{(N+1)!} W_{N+1}^X(x) \tag{3.2}$$

  where $\eta \in [-1,1]$ and $W_{N+1}^X(x)$ is the nodal polynomial of degree $N+1$ (the polynomial with the monomial of the highest degree with coefficient 1 being 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 $I_N^X(f)(x) = I_N^{X_N,1} \times I_N^{X_N,2} \cdots \times I_N^{X_N,d}(f)(x)$ where $I_N^{X_N,i}$ stands for the interpolator in dimension i. We get the following interpolation error

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

The error associated to the min max approximation is given by Feinerman and Newman [14], Soardi [42]

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

We deduce that if f is only Lipschitz

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

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

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

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

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

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

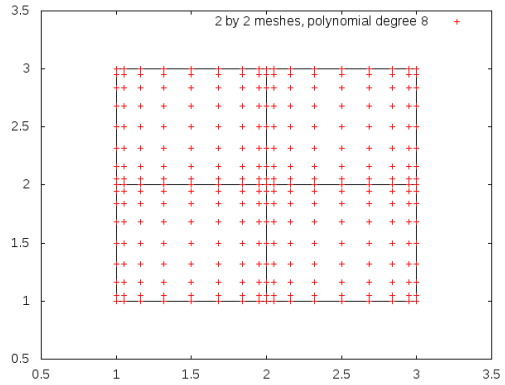


Figure 3.5: Gauss–Legendre–Lobatto points on 2×2 meshes.

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:

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

where the x_i are the interpolation points on the mesh M containing the point x. For all characteristics of this modified operator, one can see [47].

3.2.4 The C++ API

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

```
1     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.

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

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:

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

```cpp
#define INTERPOLATORSPECTRAL_H
#include <Eigen/Dense>
//#include "StOpt/core/grids/SpaceGrid.h"

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

/// forward declaration
class SpaceGrid ;

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

public :
    virtual ~InterpolatorSpectral() {}

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


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

Its constructor is given by:

```cpp
    /** \brief Constructor taking in values on the grid
     *  \param p_grid   is the grid used to interpolate
     *  \param p_values   Function value at the grids points
     */
    LegendreInterpolatorSpectral(const shared_ptr< RegularLegendreGrid>  &p_grid , const Eigen::ArrayXd &p_values) ;
```

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

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

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

```cpp
    ArrayXd lowValues = ArrayXd::Constant(2,1.); // bottom of the domain
    ArrayXd step  = ArrayXd::Constant(2,1.); // size of the mesh
    ArrayXi  nbStep = ArrayXi::Constant(2,5); // number of meshes in each direction
    ArrayXi  nPol = ArrayXi::Constant(2,2) ; // polynomial of degree 2 in each direction
    // regular
    shared_ptr<RegularLegendreGrid> regGrid(new RegularLegendreGrid(lowValues, step, nbStep, nPol));
    ArrayXd data(regGrid->getNbPoints()); // Data array
```

```
    shared_ptr<GridIterator> iterRegGrid  =  regGrid->getGridIterator(); // iterator on the
         grid points
    while (iterRegGrid->isValid())
    {
        ArrayXd pointCoord = iterRegGrid->getCoordinate();
       data(iterRegGrid->getCount()) =  f(pointCoord);
         iterRegGrid->next();
    }

    // spectral interpolator
    LegendreInterpolatorSpectral interpolator(regGrid,data);
    // interpolation point
    ArrayXd pointCoord(2, 5.2);
    // interpolated value
    double vInterp = interpolator.apply(pointCoord);
```

3.2.5 The python API

Here is an example using Legendre grids:

```
# 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.,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 )
        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()

```

```
47
48 if __name__ == '__main__':
49     unittest.main()
```

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 Γ 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 [38]. We first assume that the function we interpolate is null at the boundary. By a change of coordinate an hyper-cube domain can be changed to a domain $\omega = [0,1]^d$. Introducing the hat function $\phi^{(L)}(x) = \max(1 - |x|, 0)$ (where (L) stands for linear), we obtain the following local one dimensional hat function by translation and dilatation

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

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

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

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

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

and the space of hierarchical basis

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

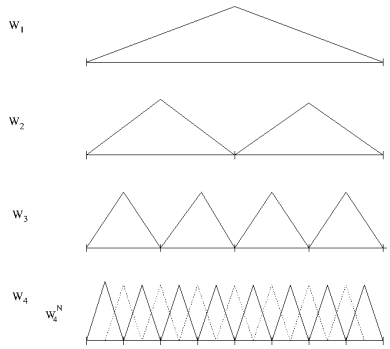


Figure 3.6: One dimensional $W^{(L)}$ spaces: $W_1^{(L)}$, $W_2^{(L)}$, $W_3^{(L)}$, $W_4^{(L)}$ and the nodal representation $W_4^{(L,N)}$

A representation of the space $W_{\underline{l}}^{(L)}$ is given in dimension 1 on figure 3.6. The sparse grid space is defined as:

$$V_n \quad = \quad \bigoplus_{|\underline{l}|_1 \leq n+d-1} W_{\underline{l}}^{(L)} \tag{3.4}$$

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

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

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

with

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

On figure 3.6 the one dimensional nodal base $W_4^{(L,N)}$ is spawned by $W_4^{(L)}$ and the dotted basis function. The space V_n can be represented as the space spawn by the $W_{\underline{l}}^{(L,N)}$ such that $|\underline{l}|_1 = n + d - 1$:

$$V_n \quad = \quad span\left\{\phi_{\underline{l},\underline{i}}^{(L)}(\underline{x}) : \underline{i} \in B_{\underline{l}}^N, |\underline{l}|_1 = n + d - 1\right\} \tag{3.5}$$

A function f is interpolated on the hierarchical basis as

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

where $\alpha_{\underline{l},\underline{i}}^{(L)}$ are called the surplus (we give on 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:

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

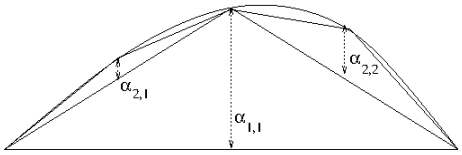


Figure 3.7: Example of hierarchical coefficients

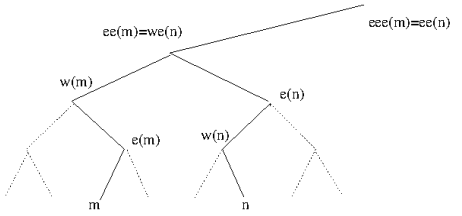


Figure 3.8: Node involved in linear, quadratic and cubic representation of a function at node m and n

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 $\underline{l} \leq 3$ in dimension 2.

In order to deal with functions not null at the boundary, two more basis are added to the

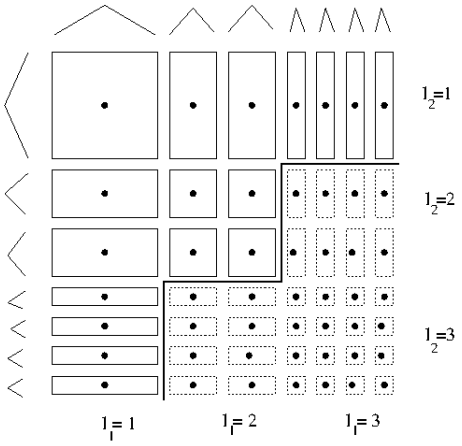


Figure 3.9: The two dimensional subspace $W_{\underline{l}}^{(L)}$ up to $l = 3$ in each dimension. The additional hierarchical functions corresponding to an approximation on the full grid are given in dashed lines.

first level as shown on figure 3.10. This approach results in many more points than the one without the boundary. As noted in [38] 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 [38]. 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

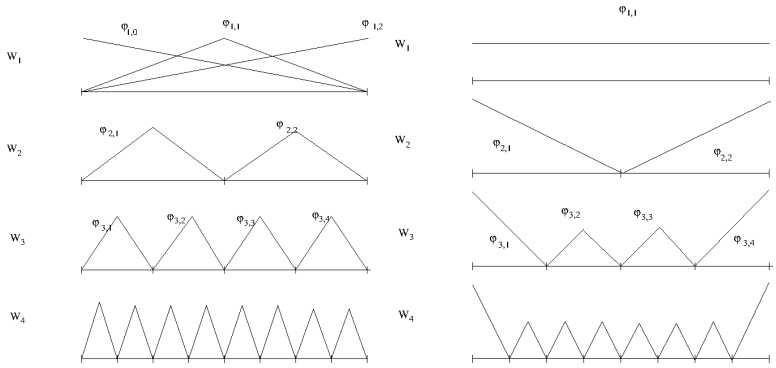


Figure 3.10: One dimensional $W^{(L)}$ spaces with linear functions with "exact" boundary (left) and "modified" boundary (right): $W_1^{(L)}$, $W_2^{(L)}$, $W_3^{(L)}$, $W_4^{(L)}$

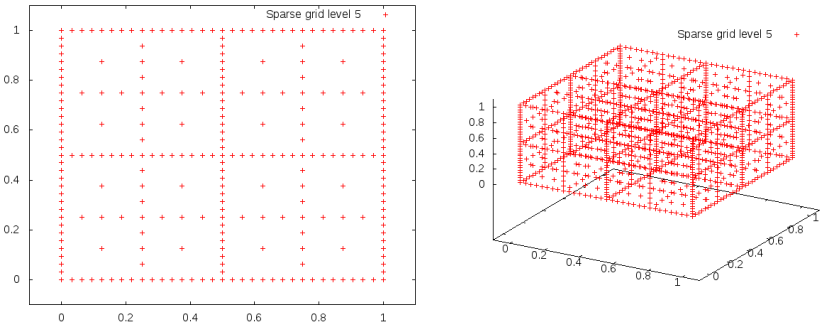


Figure 3.11: Sparse grid in dimension 2 and 3 with boundary points

boundary the left and right basis functions, other functions remaining unchanged. So the new functions basis in 1D $\tilde{\phi}$ becomes

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

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 $||\frac{\partial^{2d} u}{\partial x_1^2 ... \partial x_d^2}||_\infty < \infty$ then

$$||f - I^1(f)||_\infty = O(N^{-2} log(N)^{d-1}), \tag{3.6}$$

with N the number of points per dimension.

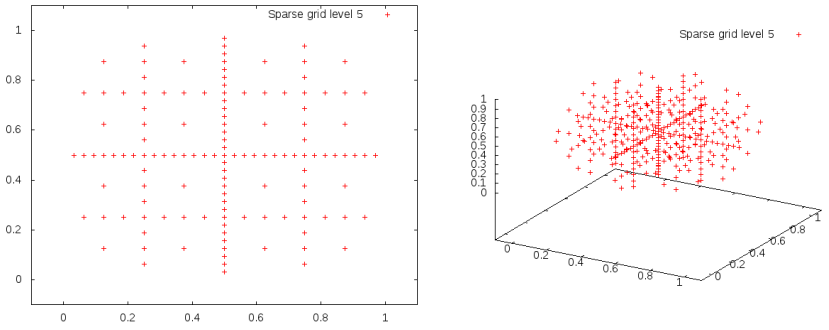


Figure 3.12: Sparse grid in dimension 2 and 3 without boundary points

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 [10] and [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

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

with $\phi^{(Q)}(x) = 1 - x^2$.
The hierarchical surplus (coefficient on the basis) in 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

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

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

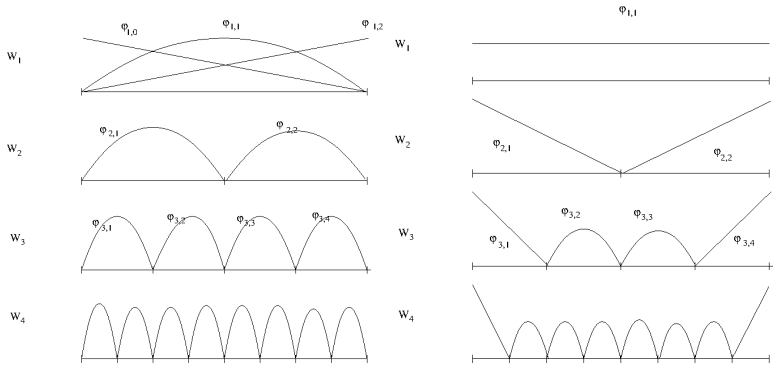


Figure 3.13: One dimensional $W^{(Q)}$ spaces with quadratic with "exact" boundary (left) and "modified" boundary (right): $W_1^{(Q)}$, $W_2^{(Q)}$, $W_3^{(Q)}$, $W_4^{(Q)}$

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

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

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

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

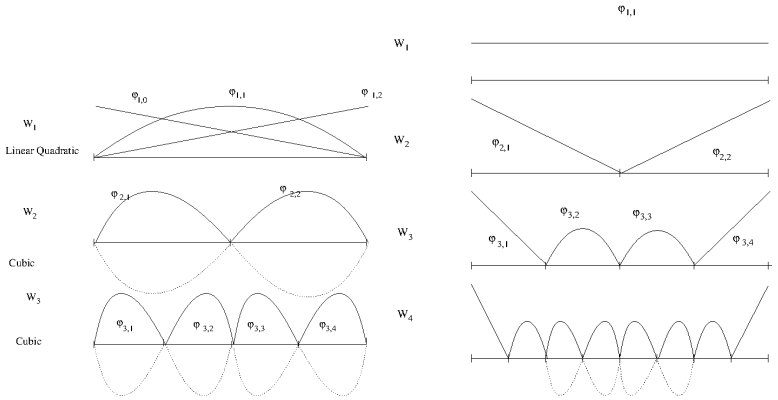


Figure 3.14: One dimensional $W^{(C)}$ spaces with cubic and "exact" boundary (left) and "modified" boundary (right): $W_1^{(C)}$, $W_2^{(C)}$, $W_3^{(C)}$, $W_4^{(C)}$

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

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

with N the number of points per dimension.

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:

$$V_n = \bigoplus_{\sum_{i=1}^d l_i w(i) \leq n+d-1} W_{\underline{l}}^{(L)} \tag{3.7}$$

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 $\alpha_{\underline{l},\underline{i}}$ are used to get an estimation of the local error. These coefficients give an estimation of the smoothness of the function value at the discrete points by representing the discrete mix second derivative of the function. There is mainly two kinds of adaptation used:

- the first one is performing local adaptation and only adds points locally [12, 19, 20, 30],

- the second one is performing adaptation at the level of the hierarchical space $W_{\underline{l}}$ (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 [26].

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

$$V_n = \bigoplus_{\sum_{i=1}^{d} l_i \leq n+d-1} W_{\underline{l}}^{(L)} \tag{3.8}$$

A set of active level $\mathcal{A}$ is created gathering all levels $\underline{l}$ such that $\sum_{i=1}^{d} l_i = n + d - 1$. All other levels are gathered in a set $\mathcal{O}$. At each level $\underline{l}$ in $\mathcal{A}$ an error is estimated $e_{\underline{l}}$ and with all local error $e_{\underline{l}}$ a global error E is calculated. Then the refinement Algorithm 1 is used noting $\mathbf{e}_k$ the canonical basis in dimension k. Sometimes, using sparse grids during time iterations,

Algorithm 1 Dimension refinement for a given tolerance η

while $E > \eta$ **do**
  select $\underline{l}$ with the highest local error $e_{\underline{l}}$
  $\mathcal{A} = \mathcal{A} \backslash \{\underline{l}\}$
  $\mathcal{O} = \mathcal{O} \cup \{\underline{l}\}$
  for $k = 1$ to d **do**
    $\underline{m} = \underline{l} + \mathbf{e}_k$
    if $\underline{m} - \mathbf{e}_q \in \mathcal{O}$ for $q \in [1, d]$ **then**
      $\mathcal{A} = \mathcal{A} \cup \{\underline{m}\}$
      Hierarchize all points belonging to $\underline{m}$
      calculate $e_{\underline{m}}$
      update E
    end if
  end for
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 η

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

3.7 C++ API

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

```
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_toHierachize   function to hierarchize
    void toHierarchize( Eigen::ArrayXd & p_toHierachize );
```

- 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_toHierachize   function to hierarchize
    void toHierarchizeVec( Eigen::ArrayXXd & p_toHierachize )
```

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
    ///        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)

```
    /// \brief 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)
    virtual void toHierarchizePByPVec(const Eigen::ArrayXXd &p_nodalValues, const  std
        ::vector<SparsePoint>  &p_sparsePoints, Eigen::ArrayXXd &p_hierarchized) const
```

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

```
#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.

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

- the second permits to hierarchize different functions together

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

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

```
    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
```

```
    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*

```
    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);
        }
        ArrayXXd hieraValues = sparseGrid.toHierarchizePByPVec(valuesFunction, sparsePoints)
            ;
```

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 `Interpolator Spectral` (similarly to Legendre grid interpolators) can be constructed:

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

This class has a member 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
```

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.

```
  // sparse grid generation
    SparseSpaceGridNoBound sparseGrid(lowValues, sizeDomain, p_level, p_weight, p_degree,
        bPrepInterp);

    //  test iterator on each level
    ArrayXd valuesFunctionTest(sparseGrid.getNbPoints());
    std::shared_ptr<SparseSet> dataSet =  sparseGrid.getDataSet();
     for (SparseSet::const_iterator iterLevel = dataSet->begin(); iterLevel != dataSet->end
         (); ++iterLevel)
      {
      // get back iterator on this level
```

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

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.

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

with

  - `p_precision` the η tolerance in the algorithm,
  - `p_fInterpol` the function permitting to calculate the nodal values,
  - `p_phi` function permitting to calculate $e_{\underline{l}}$ the local error for a given $\underline{l}$,
  - `p_phiMult` a function taking as argument all the $e_{\underline{l}}$ (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

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

```
    /// \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,
                 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.

```
# 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
        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
```

```
48          sparseGridQuad = StOptGrids.SparseSpaceGridBound(lowValues,sizeDomValues, level,
                weights,2)
49          # Hierarchize the data
50          hierarData = sparseGridQuad.toHierarchize(data)
51          # get back an interpolator
52          ptInterp = np.array([2.3,3.2,5.9],dtype=np.float)
53          interpol = sparseGridQuad.createInterpolator(ptInterp)
54          # calculate interpolated value
55          interpValue = interpol.apply(hierarData)
56          print(("Interpolated value sparse quadratic " , interpValue))
57          # now refine
58          precision = 1e-6
59          print(("Size of hierarchical array " , len(hierarData)))
60          valueAndHierar = sparseGridQuad.refine(precision,funcToInterpolate,data,hierarData)
61          print(("Size of hierarchical array after refinement " , len(valueAndHierar[0])))
62          # calculate interpolated value
63          interpol1 = sparseGridQuad.createInterpolator(ptInterp)
64          interpValue = interpol1.apply(valueAndHierar[1])
65          print(("Interpolated value sparse quadratic after refinement " , interpValue))
66          # coarsen the grid
67          precision = 1e-4
68          valueAndHierarCoarsen = sparseGridQuad.coarsen(precision,valueAndHierar[0],
                valueAndHierar[1])
69          print(("Size of hierarchical array after coarsening " , len(valueAndHierarCoarsen
                [0])))
70          # calculate interpolated value
71          interpol2 = sparseGridQuad.createInterpolator(ptInterp)
72          interpValue = interpol2.apply(valueAndHierarCoarsen[1])
73          print(("Interpolated value sparse quadratic after refinement " , interpValue))
74
75
76      # test sparse grids eliminating boundaries
77      def testSparseGridsNoBounds(self):
78          # low values
79          lowValues =np.array([1.,2.,3.],dtype=np.float)
80          # size of the domain
81          sizeDomValues = np.array([3.,4.,3.],dtype=np.float)
82          # anisotropic weights
83          weights =  np.array([1.,1.,1.])
84          # level of the sparse grid
85          level =3
86          # create the sparse grid with linear interpolator
87          sparseGridLin = StOptGrids.SparseSpaceGridNoBound(lowValues,sizeDomValues, level,
                weights,1)
88          iterGrid = sparseGridLin.getGridIterator()
89           # array to store
90          data = np.empty(sparseGridLin.getNbPoints())
91          # iterates on point
92          while( iterGrid.isValid()):
93            data[iterGrid.getCount()] = funcToInterpolate(iterGrid.getCoordinate())
94            iterGrid.next()
95          # Hierarchize the data
96          hierarData = sparseGridLin.toHierarchize(data)
97          # get back an interpolator
98          ptInterp = np.array([2.3,3.2,5.9],dtype=np.float)
99          interpol = sparseGridLin.createInterpolator(ptInterp)
100         # calculate interpolated value
101         interpValue = interpol.apply(hierarData)
102         print(("Interpolated value sparse linear" , interpValue))
103          # create the sparse grid with quadratic interpolator
104         sparseGridQuad = StOptGrids.SparseSpaceGridNoBound(lowValues,sizeDomValues, level,
                weights,2)
105         # Hierarchize the data
106         hierarData = sparseGridQuad.toHierarchize(data)
107         # get back an interpolator
108         ptInterp = np.array([2.3,3.2,5.9],dtype=np.float)
109         interpol = sparseGridQuad.createInterpolator(ptInterp)
110         # calculate interpolated value
111         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 the stochastic differential equation in the optimization problem is not controlled:

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

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 ψ_k for $k = 1$ to M, the conditional expectation can be approximated by

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

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

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

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 i can be convenient to rotate

the data set onto its principal components using the PCA method. Rotating the dataset before doing regression has been advocated in [44] and [41] 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.

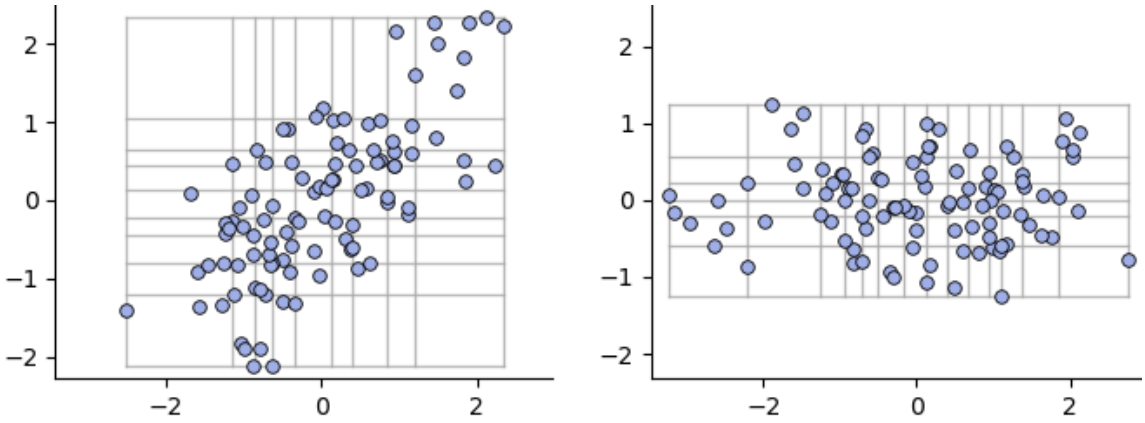


Figure 4.1: Evaluation grid: rotation

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).

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

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

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

```
         the dimension plus one)
150     /// @{
151     virtual Eigen::ArrayXd getCoordBasisFunction(const Eigen::ArrayXd &p_fToRegress) const
            = 0;
152     ///@}
153     /// \brief conditional expectation basis function coefficient calculation for multiple
            functions to regress
154     /// \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)
155     /// \return regression coordinates on the basis  (size :  number of function to regress
              \times number of meshes multiplied by the dimension plus one)
156     /// @{
157     virtual Eigen::ArrayXXd getCoordBasisFunctionMultiple(const Eigen::ArrayXXd &
            p_fToRegress) const = 0 ;
158     ///@}
159
160     /// \brief conditional expectation calculation
161     /// \param  p_fToRegress  simulations  to regress used in optimization
162     /// \return regressed value function
163     /// @{
164     virtual Eigen::ArrayXd getAllSimulations(const Eigen::ArrayXd &p_fToRegress) const = 0;
165     virtual Eigen::ArrayXXd getAllSimulationsMultiple(const Eigen::ArrayXXd &p_fToRegress)
            const = 0;
166     ///@}
167
168     /// \brief Use basis functions to reconstruct the solution
169     /// \param p_basisCoefficients basis coefficients
170     ///@{
171     virtual Eigen::ArrayXd reconstruction(const Eigen::ArrayXd   &p_basisCoefficients)
            const = 0 ;
172     virtual Eigen::ArrayXXd reconstructionMultiple(const Eigen::ArrayXXd   &
            p_basisCoefficients) const = 0;
173     /// @}
174
175     /// \brief use basis function to reconstruct a given simulation
176     /// \param p_isim               simulation number
177     /// \param p_basisCoefficients  basis coefficients to reconstruct a given conditional
            expectation
178     virtual double reconstructionASim(const int &p_isim, const Eigen::ArrayXd   &
            p_basisCoefficients) const = 0 ;
179
180     /// \brief conditional expectation reconstruction
181     /// \param  p_coordinates        coordinates to interpolate (uncertainty sample)
182     /// \param  p_coordBasisFunction regression coordinates on the basis  (size: number of
            meshes multiplied by the dimension plus one)
183     /// \return regressed value function reconstructed for each simulation
184     virtual double getValue(const Eigen::ArrayXd   &p_coordinates,
185                             const Eigen::ArrayXd   &p_coordBasisFunction) const = 0;
186
187     /// \brief conditional expectation reconstruction for a lot of simulations
188     /// \param  p_coordinates        coordinates to interpolate (uncertainty sample) size
            uncertainty dimension by number of samples
189     /// \param  p_coordBasisFunction regression coordinates on the basis  (size: number of
            meshes multiplied by the dimension plus one)
190     /// \return regressed value function reconstructed for each simulation
191     Eigen::ArrayXd  getValues(const Eigen::ArrayXXd   &p_coordinates,
192                               const Eigen::ArrayXd   &p_coordBasisFunction) const
193     {
194         Eigen::ArrayXd valRet(p_coordinates.cols());
195         for (int is  = 0; is < p_coordinates.cols(); ++is)
196             valRet(is) = getValue(p_coordinates.col(is), p_coordBasisFunction);
197         return valRet;
198     }
199
200     /// \brief permits to reconstruct a function with basis functions coefficients values
            given on a grid
201     /// \param  p_coordinates          coordinates  (uncertainty sample)
202     /// \param  p_ptOfStock            grid point
```

```
203     /// \param  p_interpFuncBasis      spectral interpolator to interpolate the basis
            functions  coefficients used in regression on the grid (given for each basis
            function)
204     virtual double getAValue(const Eigen::ArrayXd &p_coordinates,  const Eigen::ArrayXd &
            p_ptOfStock,
205                              const std::vector< std::shared_ptr<InterpolatorSpectral> > &
                                  p_interpFuncBasis) const = 0;
206
207     /// \brief is the regression date zero
208     inline bool getBZeroDate() const
209     {
210         return m_bZeroDate;
211     }
212
213     /// \brief Clone the regressor
214     virtual std::shared_ptr<BaseRegression> clone() const = 0 ;
215
216
217 };
218
219 }
220
221 #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 α_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 α_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 α_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 α_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 α_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 α_k and reconstruct the regressed solution of equation (4.1) (an array).

4.2 Adapted local polynomial basis with same probability

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 $D_{i_1,i_2,\ldots,i_d}$ where $i_j = 1$ to I_j, $M_M = \prod_{k=1,d} I_k$, and $\{D_{i_1,\ldots,i_d}\}_{(i_1,\ldots,i_d)\in[1,I_1]\times\cdots\times[1,I_d]}$ is a partition of $[\min_{k=1,N} X_{t_i}^{1,(k)}, \max_{k=1,N} X_{t_i}^{1,(k)}]\times \cdots \times[\min_{k=1,N} X_{t_i}^{d,(k)}, \max_{k=1,N} X_{t_i}^{d,(k)}]$. On each D_l, $l = (i_1, \ldots, i_d)$, depending on the selected method, ψ_l is

- either a constant function, so the global number of degrees of freedom is equal to M_M,

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

This approximation is "non-conform" in the sense that we do not 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 Cholesky 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

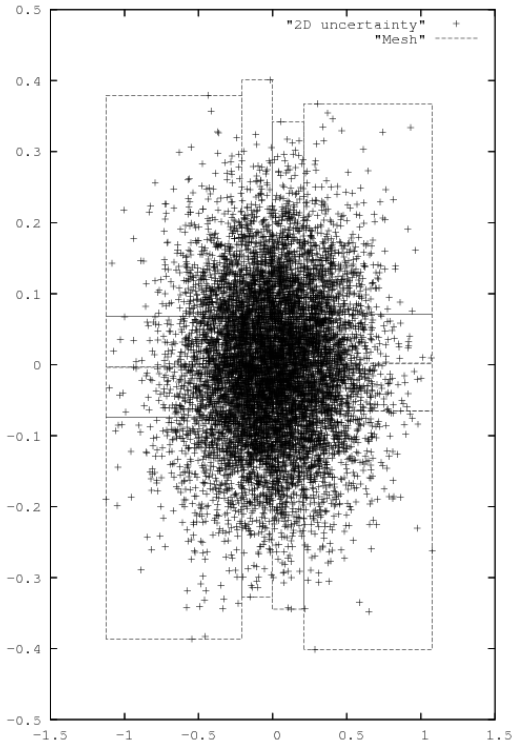


Figure 4.2: Support of 2D function basis

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 repeats this convexity. The algorithm in [31] has been implemented as an option. This algorithm may not converge when used in multi dimension but it permits to improve the convexity of the solution while iterating a few times.

4.2.2 C++ API

The constant per cell approximation

The constructor of the local constant regression object is achieved by

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

where:

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

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

where

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

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

- `p_nbMesh` is an array giving the number of meshes used in each 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.

The linear per cell approximation

The constructor of the local linear regression object is achieved by

```
1    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,

```
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 [31] implementing a generalization of the member function `getAllSimulations`:

```
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.

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.

```
// 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);
```

4.2.3 Python API

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

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

4.3 Adapted local basis by K-Means clustering methods

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

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

where μ_k is the barycenter of all points $X^j \in S_k$.
The classical Lloyd algorithm is used to calculated to calculated the cluster:

Algorithm 3 Lloyd algorithm

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

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

4: Update

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

5: **end while**

This algorithm is effective in 1D by sorting the particles coordinates. Its extension in the general case is expensive due the calculation of the Voronoi cells and the distance between all points.
We suppose that, as in the adaptive case with same probability, we want to have a partition such that the number of meshes in each direction is I_k for $k = 1, d$.
We propose a recursive algorithm to calculate the meshes. This algorithm is given by 4 and 5,

Algorithm 4 Recursive 1D Lloyd algorithm

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

Algorithm 5 Modified Lloyd algorithm

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

and an example of a resulting partition is given on figure 4.3 in 2D for $I_1 = 3, I_2 = 4$.

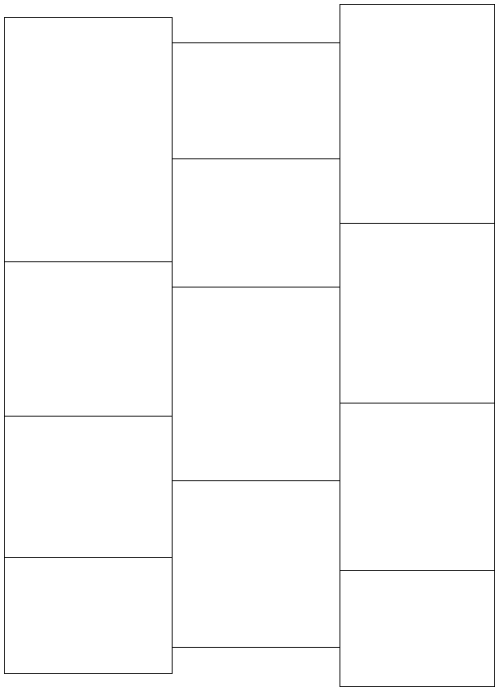


Figure 4.3: Possible 2D partition due to modified K-Means algorithm

Once the partition is achieved, regression are achieved with a constant per mesh representation:
We note $(S_k^i)_{k=1,p}$, a partition with the above method at date t_i using the Monte Carlo particles $X_{t_i}^{(k)}$, $k = 1, N$. To calculate the conditional expectation of a function g of $X_{t_{i+1}}$, we use the constant per mesh representation and we then have:

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

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

4.3.1 C++ API

The constructor of the local K-Means regression object is similar to the one obtained for the LocalConstRegression object in section 4.2.2. We then don't recall the signification of all the arguments.
A first constructor is:

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

and the second constructor permits the construct the regression matrix,

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

4.3.2 Python api

The C++ API being the same as the one for `LocalConstRegression` object, we have the same python binding as in section 4.2.3.

4.4 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.

4.5 C++ API

4.5.1 The constant per cell approximation

The constructor of the local constant regression object is achieved by

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

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

The second constructor permits the construct the regression matrix,

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

where

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

4.5.2 The linear per cell approximation

The constructor of the local linear regression object is achieved by

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

where

- `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::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.

4.5.3 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`:

```
// 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);
```

4.5.4 Python API

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

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

4.6 Sparse grid regressor

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

4.6.1 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 rescaling 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) ;
```

4.6.2 Python API

Here is a simple example of the python API:

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

4.7 Global polynomial basis

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

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

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

$$a_{n+1,k} = a_{n,k-1} - na_{n-1,k}, k > 0 \tag{4.5}$$
$$a_{n+1,0} = -na_{n-1,0} \tag{4.6}$$

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

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

  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.

4.7.2 C++ API

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

```
1    GlobalRegression(const int & p_degree, const int & p_dim, bool  p_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:

```
1   GlobalRegression(const bool &p_bZeroDate,
2           const std::shared_ptr< Eigen::ArrayXXd > &p_particles,
3           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.

```
  // total degree equal to 2
   int degree=2;
   // t is not zero
   bool bZeroDate = 0;
   // constructor with Hermite polynomials, no rotation
   GlobalRegression<Hermite> 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:

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

or by a Chebyshev one:

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

4.7.3 Python API

Here is a similar example using the second constructor

```
        import StOptReg
        nbSimul = 5000000;
        np.random.seed(1000)
        x = np.random.uniform(-.,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)
        # degree
        degree =2
        # Regressor, no rotation
        regressor = StOptReg.GlobalHermiteRegression(False,x,degree)
        y = regressor.getAllSimulations(toRegress).transpose()[0]
```

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

4.8 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:

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

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 $\mathbb{E}[Y \,|X = z]$ is given by:

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

The estimator $\hat{f}_{\mathrm{NW}}(z)$ 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:

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

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 $\mathcal{O}(M \times N)$ 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 $\mathcal{O}(N^2)$ number of operations. In StOpt we develop the methodology described in [28] permitting to get a $N \log N$ cost function.

4.8.1 The univariate case

In one dimension, StOpt uses the one dimensional Epanechnikov kernel

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

and the fast 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

$$\mathbf{S}_j = \mathbf{S}_j^{p,q} := \frac{1}{N}\sum_{i=1}^{N} K_h(x_i - z_j) x_i^p y_i^q = \frac{1}{Nh}\sum_{i=1}^{N} K\left(\frac{x_i - z_j}{h}\right) x_i^p y_i^q \,, p = 0, 1,\, q = 0, 1 \tag{4.10}$$

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 $\mathbf{S}_j$ to compute the next sum $\mathbf{S}_{j+1}$ without going through all the N input points again. Using the Epanechnikov (parabolic) kernel $K(u) = \frac{3}{4}(1 - u^2)\mathbb{1}\{|u| \leq 1\}$ we get:

$$\begin{aligned}
\mathbf{S}_j^{p,q} &= \frac{1}{Nh}\sum_{i=1}^{N} \frac{3}{4}\left(1 - \left(\frac{x_i - z_j}{h}\right)^2\right) x_i^p y_i^q \mathbb{1}\{z_j - h \leq x_i \leq z_j + h\} \\
&= \frac{1}{Nh}\frac{3}{4}\sum_{i=1}^{N}\left(1 - \frac{z_j^2}{h^2} + 2\frac{z_j}{h^2}x_i - \frac{1}{h^2}x_i^2\right) x_i^p y_i^q \mathbb{1}\{z_j - h \leq x_i \leq z_j + h\} \\
&= \frac{3}{4Nh}\left\{\left(1 - \frac{z_j^2}{h^2}\right)\mathcal{S}^{p,q}([z_j - h, z_j + h]) + 2\frac{z_j}{h^2}\mathcal{S}^{p+1,q}([z_j - h, z_j + h]) - \frac{1}{h^2}\mathcal{S}^{p+2,q}([z_j - h, z_j + h])\right\}
\end{aligned} \tag{4.11}$$

where

$$\mathcal{S}^{p,q}([L,R]) := \sum_{i=1}^{N} x_i^p y_i^q \mathbb{1}\{L \leq x_i \leq R\} \tag{4.12}$$

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

$$\begin{aligned}
\mathcal{S}^{p,q}([z_{j+1}-h, z_{j+1}+h]) &= \sum_{i=1}^{N} x_i^p y_i^q \mathbb{1}\{z_{j+1}-h \leq x_i \leq z_{j+1}+h\} \\
&= \sum_{i=1}^{N} x_i^p y_i^q \mathbb{1}\{z_j-h \leq x_i \leq z_j+h\} \\
&- \sum_{i=1}^{N} x_i^p y_i^q \mathbb{1}\{z_j-h \leq x_i < z_{j+1}-h\} + \sum_{i=1}^{N} x_i^p y_i^q \mathbb{1}\{z_j+h < x_i \leq z_{j+1}+h\} \\
&= \mathcal{S}^{p,q}([z_j-h, z_j+h]) - \mathcal{S}^{p,q}([z_j-h, z_{j+1}-h[) + \mathcal{S}^{p,q}(]z_j+h, z_{j+1}+h])
\end{aligned} \tag{4.13}$$

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

4.8.2 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 muti-dimensional case.

$$K_d(u_1, \ldots, u_d) = \frac{1}{d2^{d-1}} \sum_{k=1}^{d} K(u_k) \prod_{k_0=1}^{d} \mathbb{1}\{|u_{k_0}| < 1\} = \frac{3}{d2^{d+1}} \sum_{k=1}^{d} \left(1 - u_k^2\right) \prod_{k_0=1}^{d} \mathbb{1}\{|u_{k_0}| < 1\} \tag{4.14}$$

One can show ([28]) 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:

$$\begin{aligned}
\mathbf{S}_j &= \mathbf{S}^{p_1,p_2,q}_{k_1,k_2,j} := \frac{1}{N} \sum_{i=1}^{N} K_{d,h}(x_i - z_j) x_{k_1,i}^{p_1} x_{k_2,i}^{p_2} y_i^q \\
&= \frac{1}{N \Pi_{k=1}^{d} h_k} \sum_{i=1}^{N} K_d\left(\frac{x_{1,i} - z_{1,j}}{h_1}, \frac{x_{2,i} - z_{2,j}}{h_2}, \ldots, \frac{x_{d,i} - z_{d,j}}{h_d}\right) x_{k_1,i}^{p_1} x_{k_2,i}^{p_2} y_i^q
\end{aligned} \tag{4.15}$$

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:

$$\begin{aligned}
\mathbf{S}^{p_1,p_2,q}_{k_1,k_2,j} &= \frac{1}{N\prod_{k=1}^d h_k}\sum_{i=1}^N K_d\left(\frac{x_{1,i}-z_{1,j}}{h_1}, \frac{x_{2,i}-z_{2,j}}{h_2}, \ldots, \frac{x_{d,i}-z_{d,j}}{h_d}\right) x_{k_1,i}^{p_1} x_{k_2,i}^{p_2} y_i^q \\
&= \frac{3}{d2^{d+1}N\prod_{k=1}^d h_k}\sum_{i=1}^N\sum_{k=1}^d\left(1-\frac{(x_{k,i}-z_{k,j})^2}{h_k^2}\right) x_{k_1,i}^{p_1} x_{k_2,i}^{p_2} y_i^q \prod_{k_0=1}^d \mathbb{1}\{|x_{k_0,i}-z_{k_0,j}|\le 1\} \\
&= \frac{3}{d2^{d+1}N\prod_{k=1}^d h_k}\sum_{k=1}^d\sum_{i=1}^N\left(1-\frac{z_{k,j}^2}{h_k^2}+2\frac{z_{k,j}}{h_k^2}x_{k,i}-\frac{1}{h_k^2}x_{k,i}^2\right) x_{k_1,i}^{p_1} x_{k_2,i}^{p_2} y_i^q \prod_{k_0=1}^d \mathbb{1}\{|x_{k_0,i}-z_{k_0,j}|\le 1\} \\
&= \frac{3}{d2^{d+1}N\prod_{k=1}^d h_k}\sum_{k=1}^d\left\{\left(1-\frac{z_{k,j}^2}{h_k^2}\right)\mathcal{S}^{[0,p_1,p_2],q}_{[k,k_1,k_2]}([z_j-h_j, z_j+h_j]) + \right. \\
&= \left. 2\frac{z_{k,j}}{h_k^2}\mathcal{S}^{[1,p_1,p_2],q}_{[k,k_1,k_2]}([z_j-h_j, z_j+h_j]) - \frac{1}{h_k^2}\mathcal{S}^{[2,p_1,p_2],q}_{[k,k_1,k_2]}([z_j-h_j, z_j+h_j])\right\} \qquad (4.16)
\end{aligned}$$

where

$$\mathcal{S}^{\mathbf{p},q}_{\mathbf{k}}([\mathbf{L},\mathbf{R}]) := \sum_{i=1}^N\left(\prod_{l=1}^3 (x_{k_l,i})^{p_l}\right) y_i^q \prod_{k_0=1}^d \mathbb{1}\{L_{k_0}\le x_{k_0,i}\le R_{k_0}\} \qquad (4.17)$$

for any hypercube $[\mathbf{L},\mathbf{R}] := [L_1,R_1]\times[L_2,R_2]\times\ldots\times[L_d,R_d] \subseteq \mathbb{R}^d$, powers $\mathbf{p} := (p_1,p_2,p_3) \in \mathbb{N}^3$, $q \in \mathbb{N}$ and indices $\mathbf{k} := (k_1,k_2,k_3) \in \{1,2,\ldots,d\}^3$, 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 4.8.2), by taking advantage of the fact that the evaluation grid is rectilinear and that the supports of the kernels have a hypercube shape. Then, we are going to set up a fast sweeping algorithm using the sums on each hypercube of the partition as the unit blocks to be added and removed (subsection 4.8.2), 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 inde-

pendently as follows: for each dimension $k \in \{1, 2, \ldots, d\}$, the set of threshold points $\tilde{\mathcal{G}}_k := \{z_{k,j_k} - h_{k,j_k}\}_{j_k \in \{1,2,\ldots,M_k\}} \cup \{z_{k,j_k} + h_{k,j_k}\}_{j_k \in \{1,2,\ldots,M_k\}}$ is used to partition the k-th axis. The second row of Figure 4.4 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 $\tilde{\mathcal{G}}_k$ as $\tilde{g}_{k,1} \leq \tilde{g}_{k,2} \leq \ldots \leq \tilde{g}_{k,2M_k}$

$$\tilde{\mathcal{G}}_k = \{\tilde{g}_{k,1}, \tilde{g}_{k,2}, \ldots, \tilde{g}_{k,2M_k}\}$$

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 $z_{k,j_k} - h_{k,j_k}$ and $z_{k,j_k} + h_{k,j_k}$, $j_k \in \{1, 2, \ldots, M_k\}$, belong to $\tilde{\mathcal{G}}_k$, 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 $\left(\tilde{L}_{1,j_1}, \tilde{L}_{2,j_2}, \ldots, \tilde{L}_{d,j_d}\right)$ and $\left(\tilde{R}_{1,j_1}, \tilde{R}_{2,j_2}, \ldots, \tilde{R}_{d,j_d}\right)$ such that

$$\begin{aligned}
[z_j - h_j, z_j + h_j] &= [z_{1,j_1} - h_{1,j_1}, z_{1,j_1} + h_{1,j_1}] \times \ldots \times [z_{d,j_d} - h_{d,j_d}, z_{d,j_d} + h_{d,j_d}] \\
&= \left[\tilde{g}_{1,\tilde{L}_{1,j_1}}, \tilde{g}_{1,\tilde{R}_{1,j_1}+1}\right] \times \ldots \times \left[\tilde{g}_{d,\tilde{L}_{d,j_d}}, \tilde{g}_{d,\tilde{R}_{d,j_d}+1}\right] \\
&= \bigcup_{(l_1,\ldots,l_d) \in \{\tilde{L}_{1,j_1},\ldots,\tilde{R}_{1,j_1}\} \times \ldots \times \{\tilde{L}_{d,j_d},\ldots,\tilde{R}_{d,j_d}\}} \tilde{I}_{1,l_1} \times \ldots \times \tilde{I}_{d,l_d} \qquad (4.18)
\end{aligned}$$

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 $\tilde{I}_{1,l_1} \times \tilde{I}_{2,l_2} \times \ldots \times \tilde{I}_{d,l_d}$ 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 $k \in \{1, 2, \ldots, d\}$):

$$\mathcal{S}_{\mathbf{k}}^{\mathbf{p},q}([z_j - h_j, z_j + h_j]) = \bigcup_{(l_1,\ldots,l_d) \in \{\tilde{L}_{1,j_1},\ldots,\tilde{R}_{1,j_1}\} \times \ldots \times \{\tilde{L}_{d,j_d},\ldots,\tilde{R}_{d,j_d}\}} \mathcal{S}_{\mathbf{k}}^{\mathbf{p},q}\left(\tilde{I}_{1,l_1} \times \ldots \times \tilde{I}_{d,l_d}\right) \qquad (4.19)$$

where we assume without loss of generality that the bandwidth grid $h_j = (h_{1,j_1}, h_{2,j_2}, \ldots, h_{d,j_d})$, $j_k \in \{1, 2, \ldots, M_k\}$, $k \in \{1, 2, \ldots d\}$ is such that $\tilde{\mathcal{G}}_k$ 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.

The sum decomposition (4.19) is the cornerstone of the fast multivariate sum updating algorithm, but before going further, one can simplify the partitions $\tilde{\mathcal{G}}_k$, $k \in \{1, 2, \ldots, d\}$ while maintaining a sum decomposition of the type (4.19). Indeed, the partitions $\tilde{\mathcal{G}}_k = \{z_{k,j_k} - h_{k,j_k}, z_{k,j_k} + h_{k,j_k};\ j_k = 1, \ldots, M_k\}$ can in general produce empty intervals (intervals which do not contain any input points, cf. the grey intervals on the second row of Figure 4.4). To avoid keeping track of sums $\mathcal{S}_{\mathbf{k}}^{\mathbf{p},q}$ on the corresponding hypercubes known to be empty, one can trim the partitions $\tilde{\mathcal{G}}_k$ by shrinking each succession of empty intervals into one new partition threshold (cf. the final partition on the third row of Figure 4.4). Denote as $\mathcal{G}_k$ the resulting simplified partitions, containing the points $g_{k,1} < g_{k,2} < \ldots < g_{k,m_k}$:

$$\mathcal{G}_k = \{g_{k,1}, g_{k,2}, \ldots, g_{k,m_k}\}$$

where $2 \leq m_k \leq 2M_k$, $k \in \{1, 2, \ldots, d\}$, 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

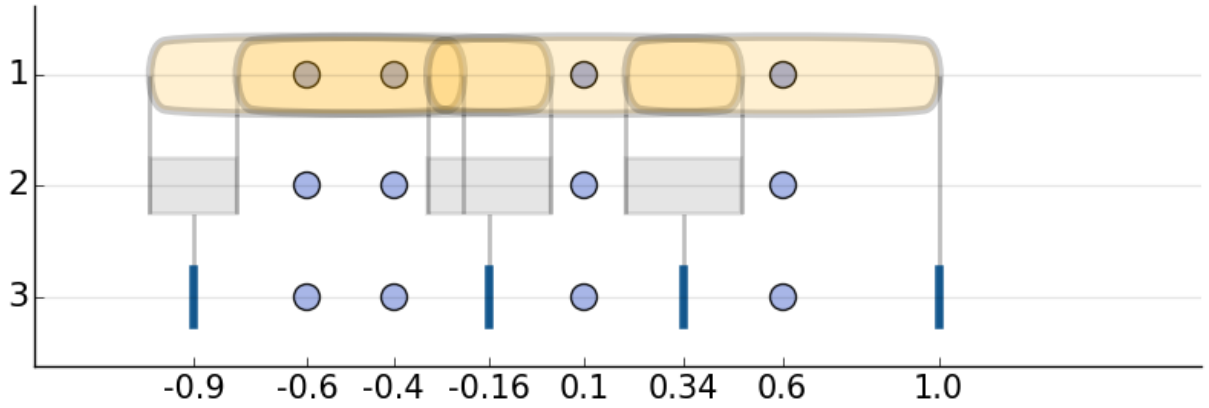


Figure 4.4: From bandwidths to partition (1D)

modified from $\tilde{\mathcal{G}}_k$ to $\mathcal{G}_k$ were empty, the following still holds:
For any evaluation point $z_j = (z_{1,j_1}, z_{2,j_2}, \ldots, z_{d,j_d}) \in \mathbb{R}^d$, $j_k \in \{1, 2, \ldots, M_k\}$, $k \in \{1, 2, \ldots d\}$, 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 $L_{k,j_k} \leq R_{k,j_k}$, $k \in \{1, 2, \ldots d\}$, such that

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

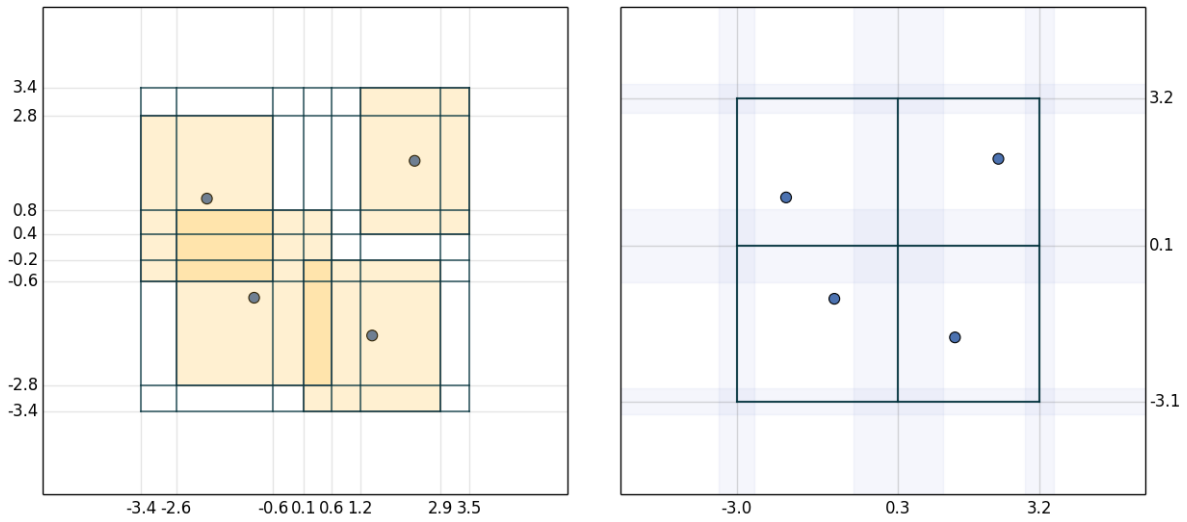


Figure 4.5: From bandwidths to partition (2D)

To complement the illustration of univariate partition given by Figure 4.4, Figure 4.5 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 $\tilde{\mathcal{G}}_k$ 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 $\mathcal{G}_k$ 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 4.8.2. The final task is to define an efficient algorithm to traverse all the hypercube unions $\bigcup_{(l_1,\ldots,l_d)\in\{L_{1,j_1},\ldots,R_{1,j_1}\}\times\ldots\times\{L_{d,j_d},\ldots,R_{d,j_d}\}} I_{1,l_1}\times\ldots\times I_{d,l_d}$, 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 $\text{idx} := (\mathbf{p}, q, \mathbf{k}) \in \{0,1,2\}\times\{0,1\}^3\times\{1,2,\ldots,d\}^3$ to summarize the polynomial $\left(\prod_{l=1}^3 (x_{k_l,i})^{p_l}\right) y_i^q$ in the sum $\mathcal{S}_{\mathbf{k}}^{\mathbf{p},q}([\mathbf{L},\mathbf{R}])$ (equation (4.17)), and introduce the compact notation

$$\mathcal{S}^{\text{idx}}_{l_1,l_2,\ldots,l_d} := \mathcal{S}_{\mathbf{k}}^{\mathbf{p},q}\left(I_{1,l_1}\times\ldots\times I_{d,l_d}\right) \tag{4.21}$$

to simplify the notation on the right-hand side of equation (4.20). In summary, $\mathcal{S}^{\text{idx}}_{l_1,l_2,\ldots,l_d}$ corresponds to the sum of the polynomials $\left(\prod_{l=1}^3 (x_{k_l,i})^{p_l}\right) y_i^q$ over all the data points within the hypercube $I_{1,l_1}\times\ldots\times I_{d,l_d}$. We precompute all the sums $\mathcal{S}^{\text{idx}}_{l_1,l_2,\ldots,l_d}$, 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 $\mathcal{T}^{\text{idx}}_{1,l_2} := \sum_{l_1=L_{1,j_1}}^{R_{1,j_1}} \mathcal{S}^{\text{idx}}_{l_1,l_2}$, 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 $\mathcal{T}^{\text{idx}}_{1,l_2} = \sum_{l_1=L_{1,1}}^{R_{1,1}} \mathcal{S}^{\text{idx}}_{l_1,l_2}$ 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 increment of j_1, we update $\mathcal{T}^{\text{idx}}_{1,l_2}$ by fast sum updating

$$\sum_{l_1=L_{1,j_1}}^{R_{1,j_1}} \mathcal{S}^{\text{idx}}_{l_1,l_2} = \sum_{l_1=L_{1,j_1-1}}^{R_{1,j_1-1}} \mathcal{S}^{\text{idx}}_{l_1,l_2} + \sum_{l_1=R_{1,j_1-1}+1}^{R_{1,j_1}} \mathcal{S}^{\text{idx}}_{l_1,l_2} - \sum_{l_1=L_{1,j_1-1}}^{L_{1,j_1-1}} \mathcal{S}^{\text{idx}}_{l_1,l_2} \tag{4.22}$$

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

$$\sum_{l_2=L_{2,j_2}}^{R_{2,j_2}} \mathcal{T}^{\text{idx}}_{1,l_2} = \sum_{l_2=L_{2,j_2-1}}^{R_{2,j_2-1}} \mathcal{T}^{\text{idx}}_{1,l_2} + \sum_{l_2=R_{2,j_2-1}+1}^{R_{2,j_2}} \mathcal{T}^{\text{idx}}_{1,l_2} - \sum_{l_2=L_{2,j_2-1}}^{L_{2,j_2-1}} \mathcal{T}^{\text{idx}}_{1,l_2} \tag{4.23}$$

Using the notation change (4.21) and equation (4.20), the resulting sum $\sum_{l_2=L_{2,j_2}}^{R_{2,j_2}} \mathcal{T}^{\text{idx}}_{1,l_2} = \sum_{l_1=L_{1,j_1}}^{R_{1,j_1}} \sum_{l_2=L_{2,j_2}}^{R_{2,j_2}} \mathcal{S}^{\text{idx}}_{l_1,l_2}$ is equal to $\mathcal{S}_{\mathbf{k}}^{\mathbf{p},q}([z_j - h_j, z_j + h_j])$, which can be used to compute the

kernel sums $\mathbf{S}_j = \mathbf{S}_{k_1,k_2,j}^{p_1,p_2,q}$ 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.

4.8.3 C++ API

The constructor permits to defines the kernel regressor:

```
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 multi-grid 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.

```

    // 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);
```

4.8.4 Python API

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

```
        import StOptReg
        nbSimul = 5000000;
        np.random.seed(1000)
        x = np.random.uniform(-.,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

Calculating conditional expectation by trees

A popular method to calculate conditional expectation consists in using scenario trees.
In the finance community, binary and trinomial trees are generally used to valuate options. When the asset is modelized by a Black Scholes model, a binary model is used, while a trinomial model is used to model mean reversion using a Vaciseck model for interest rate for example [23]. An example a trinomial tree is given in figure 5.1 for an Ornstein–Uhlenbeck model (so in dimension 1). This tree modelizes the possible evolution of a state X_t in

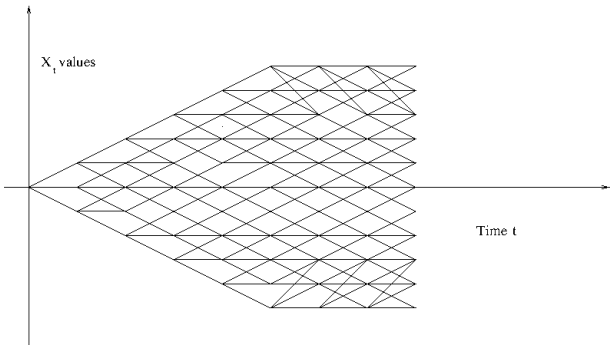


Figure 5.1: Trinomial tree

dimension 1 and each node correspond to a possible value of X_t. These trees are recombining. Nodes at each dates i are numbered from 0 to $N_i - 1$ with increasing values X_t^i of the state. From a node i at a date t, 3 nodes can be reached at date $t+1$. The probability transition to go to a node down $f(i,t)-1$ is $p_d^{t,i}$ while the probability to go to a node middle $f(i,t)$ is $p_m^{t,i}$ and the probability to go to a node up $f(i,t)+1$ is $p_u^{t,i}$.
Then conditional expectation of a function with values $g_j^{t+1} = g(X_{t+1}^j)$ at node j at date $i+1$ is simply given by:

$$\mathbb{E}[g(X_{t+1})/X_t = X_t^i] \simeq p_d^{t,i} g_{f(t,i)-1}^{t+1} + p_m^{t,i} g_{f(t,i)}^{t+1} + p_u^{t,i} g_{f(t,i)+1}^{t+1} \tag{5.1}$$

In the literature, non recombining scenario trees are used by the discrete stochastic optimization community. These non recombining trees may be obtained by reduction of some recombining trees (see [21] for example, or [27] for a more recent survey developing algorithm

minimizing the Kantorovich or Wasserstein metric between the initial tree and a subtree of the initial tree). An example of non recombining tree is given in figure 5.2.

On figure 5.2, supposing that X_2 has the possible values Y_2, Y_3 at node 2 and 3, supposing

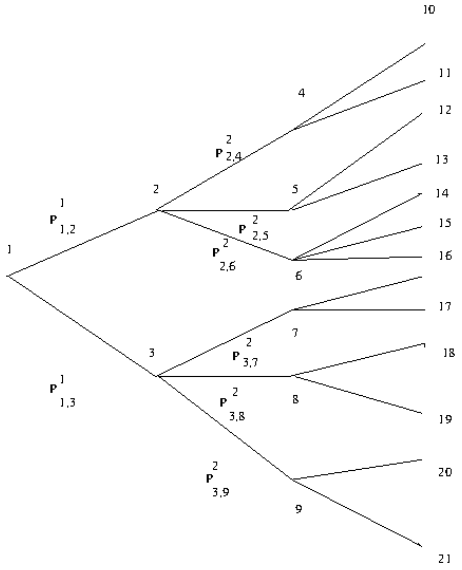


Figure 5.2: Non recombining tree

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

$$\mathbb{E}[g(X_3/X_2 = Y_3] = P_{3,7}g_7 + P_{3,8}g_8 + P_{3,9}g_9 \tag{5.2}$$

5.0.1 C++ API

Calculating conditional expectation

As explained, conditional expectation are easy to calculate with trees. The library provides a `Tree` object permitting to do such calculations.

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

with

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

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

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

As for regression objects some methods are provided to calculate conditional expectations:

- `expCond` takes an Eigen array with size the number of nodes at following date and calculate conditional expectation of values at nodes of current date,
- `expCondMultiple` does the same for multiple functions to regress (size number of function by number of nodes at following date) and return a 2 dimensional Eigen array (size number of function by number of nodes at current date).

Python API

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

```
        # Mean Reverting model
        mr = 0.3;
        sig = 0.6;

        # nb grid points
        nbStock=4

        # step
        dt = 1. / 100.

        # simulation dates
        dates =  dt * np.arange(0,16)

        # simulaton dates
        tree = Simulators.TrinomialTreeOUSimulator(mr, sig, dates)

        iFirst = 10
        iLast = 14

        # nodes at dates 5
        points = tree.getPoints(iFirst)

        # nodes at last date
        pointsNext = tree.getPoints(iLast)

        # probabilities
        proba = tree.getProbability(iFirst, iLast)

        # connection matrix
        connectAndProba =  tree.calConnected(proba);

        # to regress
        toTreeress= np.zeros( (nbStock, np.shape(pointsNext)[1]))
        for i in range(nbStock):
            toTreeress[i,:] = i + pointsNext[0,:]

        # grid for storage
        grid = StOptGrids.RegularSpaceGrid(np.array([0.]), np.array([1.]), np.array([
            nbStock - 1]))

        # conditional expectation object by trees
        tree= StOptTree.Tree(connectAndProba[1],connectAndProba[0])

        # conditional expectation taken
        valTree = tree.expCondMultiple(toTreeress)
```

Chapter 6

Continuation values objects and similar ones

In a first part we develop the different continuation objects using regression to calculate conditional expectations. Then we explain the structure of continuation object with tree to calculate these conditional expectations.

6.1 Continuation values objects with regression methods

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.

6.1.1 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 $X^{x,t} = (X_1^{x,t}, X_2^{x,t})$ where

1. the first part is given by the following equation

$$dX_{s,1}^{x,t} = b(t, X_{s,1}^{x,t})ds + \sigma_(s, X_{s,1}^{x,t})dW_s \tag{6.1}$$

   and is not controlled: the stochastic process is exogenous,

2. the second part is given by the following equation

$$dX_{s,2}^{x,t} = b_a(t)ds \tag{6.2}$$

such that the $X_2^{x,t}$ 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, $X_1^{x,t}$ holds the values of the stocks involved in the option and $X_2^{x,t}$ 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 $(X_1^{x,t}, X_2^{x,t})$.

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 α_k^i for the grid point i (see equation (4.1)),
- The second one

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

  with

  - `p_grid` the grids associated 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 α_k^i (see equation (4.1)).

The main methods provided are:

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

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

  with

- `p_grid` the grid associated to the controlled deterministic space,
- `p_condExp` the conditional expectation operator,
- `p_values` the α_k^i for all grid points i (size the number of function basis, the number of grid points)

- a second method taking as input a point to be interpolated in the grid and returning the conditional expectation at the interpolated point for all simulations:

```
Eigen::ArrayXd getAllSimulations(const Eigen::ArrayXd &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:

```
Eigen::ArrayXd getAllSimulations(const Interpolator   &p_interpol)
```

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

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

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

- a method that permits to calculate the conditional expectation for a sample of $X_1^{x,t}$:

```
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):

```
  int sizeForStock  = 4;
 // second member to regress with one stock
ArrayXXd toRegress = ArrayXXd::Constant(p_nbSimul,sizeForStock, 1.);
// grid for stock
Eigen::ArrayXd lowValues(1), step(1);
lowValues(0) = 0. ;
step(0) = 1;
Eigen::ArrayXi  nbStep(1);
nbStep(0) = sizeForStock - 1;
```

```
        // grid
        shared_ptr< RegularSpaceGrid > regular = MyMakeShared<RegularSpaceGrid>(lowValues,
            step, nbStep);
        // conditional espectation (local basis functions)
        ArrayXi  nbMesh = ArrayXi::Constant(p_nDim, p_nbMesh);
         shared_ptr<LocalLinearRegression> localRegressor = MyMakeShared<
             LocalLinearRegression>(false, x, nbMesh);

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

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

Python API

Here is an example of the use of the mapping

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


# 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):
        import StOptGrids
        import StOptReg
        # 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,:])*(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
        # Now create the continuation object
```

```
        ###################################
        contOb = StOptReg.ContinuationValue(grid,regressor,toRegressMult)
        # get back the regressed values at the point stock
        ptStock=  np.array([1.2,3.1,5.9],dtype=np.float)
        regressValues = contOb.getAllSimulations(ptStock)
        # do the same with an interpolator
        interp = grid.createInterpolator(ptStock)
        regressValuesInterp = contOb.getAllSimulations(interp)
        # test create of an interpoaltion object mixing grids for stocks and regression for uncertainties
        #
            ##########################################################################################

        gridAndRegressed = StOptReg.GridAndRegressedValue(grid,regressor,toRegressMult)
        # get back the regressed value for a point stock and an uncertainty
        valRegressed = gridAndRegressed.getValue(ptStock, x[:,0])

    # test some mapping of GneralSpaceGrid
    def testGeneralGridInheritance(self):
        from StOptGrids import GeneralSpaceGrid, RegularSpaceGrid
        from StOptReg import LocalLinearRegression, ContinuationValue

        x = np.random.randn(5)
        regressor = LocalLinearRegression([1])

        regular = RegularSpaceGrid(np.array([0.]), np.array([0.5]), np.array([3]))
        ContinuationValue(regular, regressor, x)

        general = GeneralSpaceGrid([[0., 1., 1.2, 1.5]])
        ContinuationValue(general, regressor, x)


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

6.1.2 The GridAndRegressedValue object

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

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:

```
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.

```
    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)

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

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

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

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

Python API

The python API is similar to the one of the `ContinuationValue` object (see Section 6.1.1).

6.1.3 The continuation cut object

Suppose the control problem is continuous and that the state of the system has the dynamic given by (6.1) et (6.2). This is the case of some storages modelized 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 margin process $X_{s,1}^{x,t}$, the Bellman curve can be approximated by cuts (see 6.1) 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

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

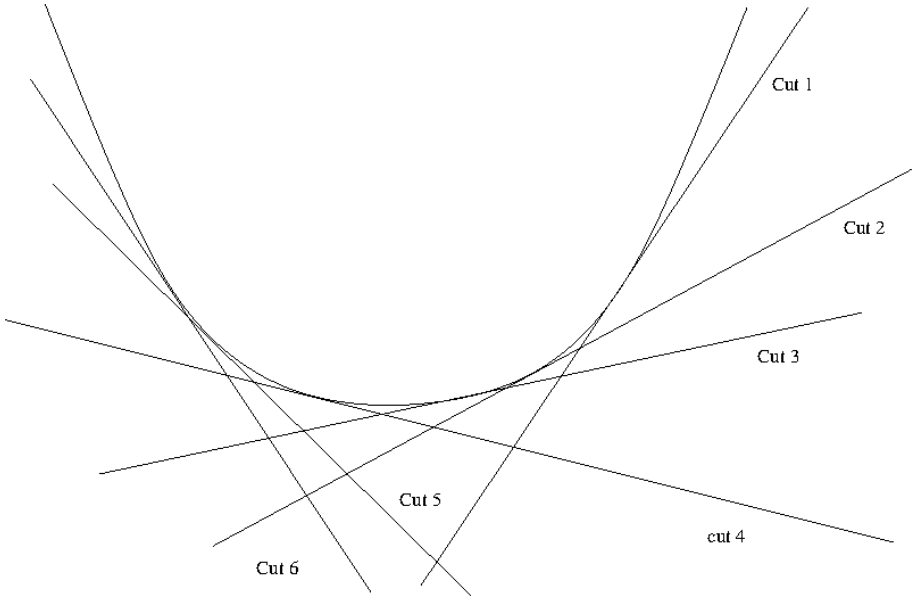


Figure 6.1: Bellman cuts

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

$$\mathbb{E}\left[a_i(X_{s,1}^{x,t})|X_{s',1}^{x,t}\right] = \sum_{j=1}^{N} a_{i,j}\psi_j(X_{s',1}^{x,t}) \tag{6.3}$$

where the ψ_j correspond to some basis function.

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 $a_{i,j}^k$ 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 (6.3). 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 $\hat{a}^k_{i,j} = a^k_{i,j}$, $i = 1, \dots d$. Then the conditional cut can be written as:

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

The main methods provided are:

- a first method used in simulation permitting to load for grid point i the coefficient α^i_k associated to the function g_i,

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

  with

  - `p_grid` the grid associated 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 $j = 1, \dots, N$ and all stock points.

- a second method taking as input a description of an hypercube (nb storages,2) describes by its extreme coordinates:

  - (i,0) coordinate corresponds to minimal coordinate value in dimension i
  - (i,1) coordinate corresponds to maximal coordinate value in dimension i

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

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

  - First dimension correspond 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.

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

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

Python API

Here is an example of the use of the mapping

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


# unit test for continuation values
####################################

class testContValues(unittest.TestCase):

    # unit test for continuation cuts
    def testSimpleGridsAndRegressor(self):
        # low value for the meshes
        lowValues =np.array([1.,2.,3.],dtype=np.float)
        # size of the meshes
        step = np.array([0.7,2.3,1.9],dtype=np.float)
        # number of steps
        nbStep = np.array([3,2,4], dtype=np.int32)
        # create the regular grid
        ##########################
        grid = StOptGrids.RegularSpaceGrid(lowValues,step,nbStep)
        # simulation
        nbSimul =10000
        np.random.seed(1000)
        x = np.random.uniform(-1.,1.,size=(1,nbSimul));
        # mesh
        nbMesh = np.array([16],dtype=np.int32)
        # Create the regressor
        ######################
        regressor = StOptReg.LocalLinearRegression(False,x,nbMesh)
        # regressed values
        toReal = (2+x[0,:]+(1+x[0,:])*(1+x[0,:]))
        # function to regress
        toRegress = toReal + 4*np.random.normal(0.,1,nbSimul)
        # fictitous cuts with 0 sensibility (1 dimension)
        toRegressCuts = np.zeros(shape=(4*len(toRegress),grid.getNbPoints()))
        for i in range(toRegressCuts.shape[1]):
           toRegressCuts[:len(toRegress),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]],
```

```
                                [lowValues[2],lowValues[2]+step[2]*nbStep[2]]])
        regressCuts = contOb.getCutsAllSimulations(hyperCube)


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

6.2 Continuation objects and associated with trees

6.2.1 Continuation object

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

C++ API

As for `ContinuationValue` object two constructors are provided:

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

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

  with

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

The main methods provided are:

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

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

  with

  - `p_grid` the grid associated to the controlled deterministic space,
  - `p_values` the continuation values for all nodes and stock points (size number of nodes by number of grid points at current date)

- a second method taking as input a point to be interpolated in the grid and returning the conditional expectation at the interpolated point for all nodes:

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

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

```
    Eigen::ArrayXd getValueAtNodes(const Interpolator   &p_interpol)
```

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

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

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

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

- a method that permits to get back all conditional expectations for all nodes:

```
   double getValues() const
```

Python API

Used importing the StOptTree module, the syntax is similar to the c++ one. Continuing example in section 5.0.1;

```
        # continuation object
        continuation = StOptTree.ContinuationValueTree(grid, tree,  toTreeress.transpose())

        # interpolation point
        ptStock = np.array([0.5*nbStock])

        #  conditional expectation using continuation object
        treeByContinuation = continuation.getValueAtNodes(ptStock);
```

6.2.2 GridTreeValues

This object permits to interpolate in some grid values for some function defined on nodes values and grid values.
The constructor

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

where:

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

The methods:

- The following one permits to interpolate at a given stock point for a given node

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

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

- The second one gives the interpolated values at all nodes

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

Python API

Importing the StOptTree, previous constructor and methods are available.

6.2.3 Continuation Cut with trees

As with regressor (section 6.1.3), we can provide cuts when approximating some concave or convex function at each node of the tree.

C++ API

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

- The second one

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

  with

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

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

The main methods provided are:

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

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

  with

  - `p_grid` the grid associated to the controlled deterministic space,
  - `p_condExp` the conditional expectation operator by tree,
  - `p_values` the a_i coefficients to reconstruct the cuts: its size corresponds to the number of cut coefficients. Then the element i of `p_values` permits to store the a_i coefficients for all nodes and all stock points.

- a second method taking as input a description of an hypercube (nb storages,2) describes by its extreme coordinates:

  - (i,0) coordinate corresponds to minimal coordinate value in dimension i
  - (i,1) coordinate corresponds to maximal coordinate value in dimension i

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

  It return an array of cuts coefficients for all nodes in the tree at grid points inside the hypercube.

  - First dimension corresponds to the number of cuts coefficients by the number of nodes.
  - The second dimension corresponds to the number of points in the hypercube `p_hypStock`.

- a method to get an array of cuts for a give node.

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

  where:

  - `p_hypStock` corresponds to an hypercube used to select some stock points as previously,
  - `p_node` corresponds to the node number in the tree.

  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.

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

Python API

Importing the `StOptTree` module, the `ContinuationCutsTree` object is available in python.

Part III

Solving optimization problems with dynamic programming methods

Chapter 7

Creating simulators

In order to optimize the control problem, the user has to develop some simulators permitting to draw some trajectories of the uncertainties. This trajectories are used while optimizing or in a simulation part testing the optimal control.

7.1 Simulators for regression 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`.

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

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

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 $X^k_{p,i}$ and we give below the meaning of the different methods of `SimulatorDPBase`:

- the `getParticle` method gives at the current optimization/simulation date t_i the Markov states $X^k_{p,i}$ in a matrix A such that $A(p,k) = X^k_{p,i}$,
- 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.

7.2 Simulators for trees

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

```
// Copyright (C) 2019 EDF
// All Rights Reserved
// This code is published under the GNU Lesser General Public License (GNU LGPL)
#ifndef SIMULATORDPBASETREE_H
#define SIMULATORDPBASETREE_H
#include <Eigen/Dense>
#include "geners/BinaryFileArchive.hh"

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

namespace StOpt
{
/// \class SimulatorDPBaseTree SimulatorDPBaseTree.h
/// Abstract class for simulator used in dynamic programming with trees
class SimulatorDPBaseTree
{
protected :

    std::shared_ptr<gs::BinaryFileArchive> m_binForTree ; ///< archive for tree
    Eigen::ArrayXd m_dates ; ///< list of dates in the archive
    int m_idateCur ; ///< current date index
    Eigen::ArrayXXd  m_nodesCurr ; ///< storing coordinates of the nodes at current date  (dim, nbnodes)
    Eigen::ArrayXXd  m_nodesNext; ///< storing coordinates of the nodes at next date (dim, nbnodes)
    std::vector<double>  m_proba ; ///<  value stores probability to go from on node   at index m_dateCurc to node  at next date m_dateNext.
    std::vector< std::vector< std::array<int, 2> > >  m_connected ; ///<for each node at current  date, give a list of connected nodes at next date and index in probability vector
    /// \brief  load a date
    void load(const int &p_idateCur);

public :

    /// \brief Constructor
    SimulatorDPBaseTree() {}

    /// \brief Constructor : use in backward
    /// \param  p_binforTree  binary geners archive  with structure
    ///         - dates      ->  eigen array of dates, size ndate
    ///         - nodes     ->  nDate array , each array containing nodes coordinates  with size  (ndim, nbNodes)
    ///         - proba      ->  probabilities to go from node to another from a date to the next date
    ///         - connected  -> connecton matrix for a node at current date to go to a node at next date
    ///
    SimulatorDPBaseTree(const std::shared_ptr<gs::BinaryFileArchive>   &p_binForTree);

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

    /// \brief a step forward for simulations
    virtual void  stepForward() = 0;
```

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

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

While designing its tree the user has to call the based simulator constructor by providing a geners archive giving

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

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

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

The different methods the use has to provide are

- 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 samples are generated to give discrete uncertainties in the tree.

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

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

Chapter 8

Using conditional expectation 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.

8.1 American option by regression

8.1.1 The American option valuing by Longstaff–Schwartz

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

$$P_t = \text{esssup}_{\tau \in \mathcal{T}_{[t,T]}} \mathbb{E}(g(\tau, X_\tau) \mid \mathcal{F}_t) \quad \text{for} \;\; t \leq T \;\; \mathbb{P} - \text{a.s.} \,, \tag{8.1}$$

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

We recall the classical Longstaff–Schwartz Algorithm 6 estimating the empirical conditional expectation using the regression estimation previously seen.

Algorithm 6 Algorithm with regression [optimal exercise time estimation]

Initialization:
Set $\hat{\tau}_\kappa^{1,\pi,(j)} := T$, $j \leq N$
Backward induction:
for $i = \kappa - 1$ to 0 **do**
set $\hat{\tau}_i^{1,\pi} := t_i \mathbf{1}_{A_i^1} + \hat{\tau}_{i+1}^{1,\pi} \mathbf{1}_{(A_i^1)^c}$ where $A_i^1 := \{g(t_i, X_{t_i}) \geq \hat{\mathbb{E}}[g(\hat{\tau}_{i+1}^{1,\pi}, X_{\hat{\tau}_{i+1}^{1,\pi}}) \mid \mathcal{F}_{t_i}]\}$.
end for
Price estimator at 0: $\hat{P}_0^{1,\pi} := \hat{\mathbb{E}}[g(\hat{\tau}_0^{1,\pi}, X_{\hat{\tau}_0^{1,\pi}})]$.

American option by regression with the C++ API

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

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

American option with the Python API

Using the python API the American resolution is given below:

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

# american option by Longstaff-Schwartz
# p_sim         Monte Carlo simulator
# p_payOff      Option pay off
# p_regressor   regressor object
def resolution(p_simulator, p_payOff, p_regressor) :

    step = p_simulator.getStep()
    # asset simulated under the neutral risk probability : get the trend of first asset to
        get interest rate
    expRate = np.exp(-step * p_simulator.getMu()[0])
    # Terminal
    particle = p_simulator.getParticles()
    Cash = p_payOff.operator(particle)

    for iStep in range(0, p_simulator.getNbStep()):
        asset = p_simulator.stepBackwardAndGetParticles()
        payOffLoc = p_payOff.operator(asset)
        isLastStep = False
        if iStep == p_simulator.getNbStep() - 1 :
            isLastStep = True

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

    return maths.fsum(Cash) / len(Cash)
```

8.2 American options by tree

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

Algorithm 7 Algorithm with tree: Bellman value (0 interest rate)

Initialization:
Set $P^j := g(X_T^j)$, $j \leq N(\kappa)$ ▷ Number of node at last date
Backward induction:
for $i = \kappa - 1$ to 0 **do**
  set $P^j = \max[\mathbb{E}[P \mid X_{t_i}^j], g(X_{t_i}^j]$, $j \leq N(i)$
end for
Price estimator at 0: P^0.

8.2.1 The American option by tree

```
    // a backward simulator
    MeanRevertingSimulatorTree< OneDimData<OneDimRegularSpaceGrid, double> > backSimulator1
        (binArxiv, futureGrid, sigma, mr);

    // strike of put
    double strike = 50.;

    // actualization
    double actu = exp(r * dates(dates.size() - 1));
    // spot provided by simulator
    ArrayXd spot = backSimulator1.getSpotValues() * actu;
    // actualized value for payoff
    ArrayXd val1 = (strike - spot).cwiseMax(0.) / actu;
    for (int istep = 0; istep < nbDtStep; ++istep)
    {
        // one step backward to update probabilities and connectons between nodes
        backSimulator1.stepBackward();
        // probabilities
        std::vector<double>  proba = backSimulator1.getProba();
        // get connection between nodes
        std::vector< std::vector<std::array<int, 2>  > >  connected = backSimulator1.
            getConnected();
        // conditional expectation operator
        StOpt::Tree tree(proba, connected);
        //interest rates
        actu = exp(r * dates(dates.size() - 1 - (istep + 1) * nInc));
        // spot : add interest rate
        spot = backSimulator1.getSpotValues() * actu;
        // pay off
        ArrayXd payOff = (strike - spot).cwiseMax(0.) / actu;
        //actualize value
        val1 = tree.expCond(val1);
        // arbitrage
        val1 = (val1 > payOff).select(val1, payOff);
    }

    double finalValue = val1(0);
```

8.2.2 Python API

```
        # backward simulator
        backSimulator = Simulators.MeanRevertingSimulatorTree(archiveToRead, futureGrid,
            sigma, mr)

        # strike
        strike  = 50.

        # actu
        actu = np.exp(r*dates[indexT[-1]])
        # spot : add interest rate
        spot = backSimulator.getSpotValues()*actu
        # actualized payoff
        val1= np.where( strike-spot>0,strike-spot,0)/actu
        for istep in np.arange(np.shape(indexT)[0]-1):
            # backward in simulator
            backSimulator.stepBackward()
            # get back probability matrix
            proba = backSimulator.getProba()
            # and connection matrix
            connected = backSimulator.getConnected()
            # creta tree for conditional expectation
            tree=StOptTree.Tree(proba,connected)
            # interest rates
            actu = np.exp(r*dates[indexT[-2-istep]])
            # spot : add interest rate
            spot = backSimulator.getSpotValues()*actu
            # pay off
            payOff = np.where( strike-spot>0,strike-spot,0)/actu
            # actualize value
            val1 = tree.expCond(val1)
            # arbitrage
            val1 = np.where( val1> payOff, val1, payOff)

        final = val1[0]
```

Chapter 9

Using the general framework to manage stock problems

In this chapter the state is separated into three parts $X^{x,t} = (X_1^{x,t}, X_2^{x,t}, I_t)$. $(X_1^{x,t}, X_2^{x,t})$, which corresponds to the special case of chapter 6 where $X_1^{x,t}$ is not controlled and $X_2^{x,t}$ is controlled. Two cases can be tackled:

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

I_t takes some integers values and is here to describe some finite discrete regimes (to treat some switching problems). A general framework is available to solve this kind of problem. First, the second part $X_2^{x,t}$ is discretized on a grid as explained in chapter 6.

- Either a full grid is used for $X_2^{x,t}$ and two types of resolutions either sequential or parallel be can considered:
  - a resolution can be achieved sequentially or a parallelization with MPI on the calculations can be achieved (speed up but no size up). This approach can be used for problems in small dimension.
  - a resolution can be achieved with a parallelization by the MPI framework by spreading the work to be achieved on the grid points, and spread the data between processors (speed up and size up). We will denote this parallelization technique a "distribution" technique. This approach is necessary to tackle very big optimization problems where the global solution cannot be stored in the memory of a single processor.
- or the grid for $X_2^{x,t}$ is not full (so sparse) and only a parallelization by thread and MPI can be achieved on the calculations (speed up and no size up). With sparse grids, only the case $X_2^{x,t}$ deterministic is treated.

In the case of the MPI parallelization technique distributing task and data (full grids only), [32] and [43] 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 9.1) 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 9.1, 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.

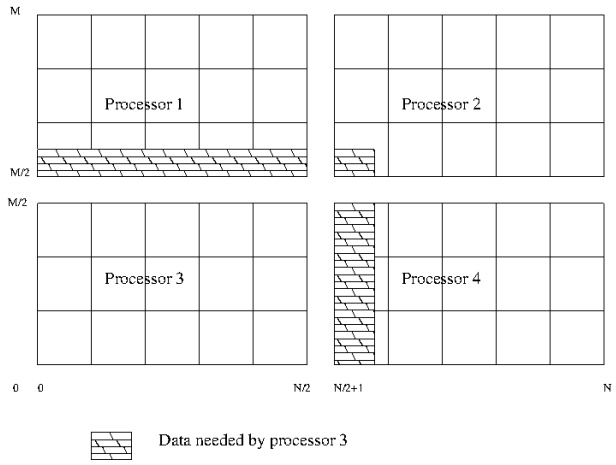


Figure 9.1: Data to send to processor 3

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

9.1 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`.

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

```
 *      \author Xavier Warin
 */

namespace StOpt
{

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


public :

    OptimizerBase() {}

    virtual ~OptimizerBase() {}

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

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


    /// \brief Get the number of regimes allowed for the asset to be reached  at the current time step
    virtual   int getNbRegime() 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 Δ. 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 $X_2^{t,x}$ 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.

9.2 Solving the problem using conditional expectation calculated by regressions

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

9.2.1 Requirement to use the framework

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

- the `stepSimulateControl` method is used after optimization using the optimal controls calculated in the optimization part. From a state `p_state` (storing the $X^{x,t}$), the 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.

9.2.2 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` itself derived from `OptimizerBaseInterp`.

```
// Copyright (C) 2016 EDF
// All Rights Reserved
// This code is published under the GNU Lesser General Public License (GNU LGPL)
#ifndef OPTIMIZERDPBASE_H
#define OPTIMIZERDPBASE_H
#include <Eigen/Dense>
#include "StOpt/core/utils/StateWithStocks.h"
#include "StOpt/core/grids/SpaceGrid.h"
#include "StOpt/regression/BaseRegression.h"
#include "StOpt/regression/ContinuationValue.h"
#include "StOpt/regression/GridAndRegressedValue.h"
#include "StOpt/dp/SimulatorDPBase.h"
#include "StOpt/dp/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
///  Base class for optimizer for Dynamic Programming  with regression methods
class OptimizerDPBase : public OptimizerBaseInterp
{


public :

    OptimizerDPBase() {}

    virtual ~OptimizerDPBase() {}

    /// \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 ;

    /// \brief defines a step in optimization
    /// \param p_grid      grid at arrival step after command
    /// \param p_stock     coordinates of the stock point to treat
    /// \param p_condEsp   continuation values for each regime
    /// \param p_phiIn     for each regime  gives the solution calculated at the previous step ( next time step by Dynamic Programming resolution) : structure of the 2D array ( nb simulation ,nb stocks )
    /// \return   a pair  :
    ///             - for each regimes (column) gives the solution for each particle (row)
    ///             - for each control (column) gives the optimal control for each
```

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

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

- the `stepOptimize` method is used in optimization. We want to calculate the optimal value at current t_i at a grid point `p_stock` using a grid `p_grid` at the next date t_{i+1}, the continuation values for all regimes `p_condEsp` permitting to calculate conditional expectation of the optimal value function calculated at the previously treated time step t_{i+1}. From a grid point `p_stock` it calculates the function values and the optimal controls. It returns a pair where the

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

- the `stepSimulate` method is used after optimization using the continuation values calculated in the optimization part. From a state `p_state` (storing the $X^{x,t}$), the continuation values calculated in optimization `p_continuation`, the optimal functions values along the current trajectory are stored in `p_phiInOut`.

In the case of a gas storage [45], 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 the `Optimize GasStorage.h` 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:

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

with

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

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

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

The main method is

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

with

- `p_phiIn` the vector (its size corresponds to the number of regimes) of 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_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:

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

with:

- `p_ar` is the archive where controls and solutions are dumped,
- `p_name` is a base name used in the archive to store the solution and the control,
- `p_phiInPrev` is the solution at the previous time step used to calculate the continuation values that are stored,
- `p_control` stores the optimal controls calculated at the current time step,
- `p_condExp` is the conditional expectation object permitting to calculate conditional expectation of functions defined at the previous time step treated `p_phiInPrev` and permitting to store a representation of the optimal control.
- `p_bOneFile` is set to one if the continuation and optimal controls calculated by each processor are dumped on a single file. Otherwise the continuation and optimal controls calculated by each processor are dumped on different files (one by processor). If the problem gives continuation and optimal control values on the global grid that can be stored in the memory of the computation node, it can be more interesting to dump the continuation/control values in one file for the simulation of the optimal policy.

Remark 9 *As for the* ***`TransitionStepRegressionDP`*** *and the* ***`TransitionStepRegressionDPSparse`*** *object, their* ***`dumpContinuationValues`*** *does not need a* ***`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)
#ifdef 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/FinalStepDPDist.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::FinalStepDPDist(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)
    {
        Eigen::ArrayXXd asset = 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();
60
61 }
62 #endif
```

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

Table 9.1: Which `TransitionStepRegression` object to use depending on the grid used and the type of parallelization used.

	Full grid	Sparse grid
Sequential	`TransitionStepRegressionDP`	`TransitionStepRegressionDPSparse`
Parallelization on calculations threads and MPI	`TransitionStepRegressionDP`	`TransitionStepRegressionDPSparse`
Distribution of calculations and data	`TransitionStepRegressionDPDist`	Not available

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 6) 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

```
SimulateStepRegressionDist(gs::BinaryFileArchive &p_ar, const int &p_iStep, const std::string &p_nameCont,
                           const shared_ptr<FullGrid> &p_pGridFollowing, 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`),
  - `p_Optimize` the Optimizer describing the transition from one time step to the following one,
  - `p_OneFile` equal to `true` if a single archive is used to store continuation values.

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

  This object implements the method `oneStep`

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

  where:

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

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

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

```cpp
#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"


/** \file 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
/// \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
/// \param p_bOneFile                do we store continuation values  in only one file
double SimulateRegressionDist(const std::shared_ptr<StOpt::FullGrid> &p_grid,
                              const std::shared_ptr<StOpt::OptimizerDPBase > &p_optimize,
                              const std::function<double(const int &, const Eigen::ArrayXd &, const Eigen::ArrayXd &)>  &p_funcFinalValue,
                              const Eigen::ArrayXd &p_pointStock,
                              const int &p_initialRegime,
                              const std::string   &p_fileToDump,
                              const bool &p_bOneFile)
{
    boost::mpi::communicator world;
    // from the optimizer get back the simulator
    std::shared_ptr< StOpt::SimulatorDPBase> simulator = p_optimize->getSimulator();
    int nbStep = simulator->getNbStep();
    std::vector< StOpt::StateWithStocks> states;
    states.reserve(simulator->getNbSimul());
    for (int is = 0; is < simulator->getNbSimul(); ++is)
        states.push_back(StOpt::StateWithStocks(p_initialRegime, p_pointStock, Eigen::ArrayXd::Zero(simulator->getDimension())));
    std::string toDump = p_fileToDump ;
    // test if one file generated
    if (!p_bOneFile)
        toDump +=  "_" + boost::lexical_cast<std::string>(world.rank());
    gs::BinaryFileArchive ar(toDump.c_str(), "r");
    // name for continuation object in archive
    std::string nameAr = "Continuation";
    // cost function
    Eigen::ArrayXXd costFunction = Eigen::ArrayXXd::Zero(p_optimize->getSimuFuncSize(), simulator->getNbSimul());
    for (int istep = 0; istep < nbStep; ++istep)
    {
        StOpt::SimulateStepRegressionDist(ar, nbStep - 1 - istep, nameAr, p_grid, p_optimize, p_bOneFile).oneStep(states, costFunction);

        // new stochastic state
        Eigen::ArrayXXd particles =  simulator->stepForwardAndGetParticles();
        for (int is = 0; is < simulator->getNbSimul(); ++is)
            states[is].setStochasticRealization(particles.col(is));

    }
    // final : accept to exercise if not already done entirely (here suppose one
```

```
        function to follow)
69      for (int is = 0; is < simulator->getNbSimul(); ++is)
70          costFunction(0, is) += p_funcFinalValue(states[is].getRegime(), states[is].
                getPtStock(), states[is].getStochasticRealization()) * simulator->getActu
                ();
71
72      return costFunction.mean();
73  }
74
75  #endif /* SIMULATEREGRESSIONDIST_H */
```

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

- Simulation step using the optimal controls calculated in optimization:

```
1   SimulateStepRegressionControlDist(gs::BinaryFileArchive &p_ar,  const int &p_iStep
        ,  const std::string &p_nameCont,
2                                     const   std::shared_ptr<FullGrid> &p_pGridCurrent
                                          ,
3                                     const   std::shared_ptr<FullGrid> &
                                          p_pGridFollowing,
4                                     const  std::shared_ptr<OptimizerDPBase > &
                                          p_pOptimize,
5                                     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 `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`

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

where:

- `p_statevector` stores for all the simulations the state: this state is updated by application of the optimal commands,

– `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:

```
// Copyright (C) 2016 EDF
// All Rights Reserved
// This code is published under the GNU Lesser General Public License (GNU LGPL)
#ifdef USE_MPI
#ifndef SIMULATEREGREGRESSIONCONTROLDIST_H
#define SIMULATEREGREGRESSIONCONTROLDIST_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/SimulateStepRegressionControlDist.h"
#include "StOpt/dp/OptimizerDPBase.h"
#include "StOpt/dp/SimulatorDPBase.h"


/** \file SimulateRegressionControlDist.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
/// \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
```

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

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

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

Table 9.2: Which simulation object to use depending on the TransitionStepRegression object used.

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

9.2.3 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 14. In order to use this cuts a business object using some LP solver has to be created. This business object is derived from `OptimizerDPCutBase` itself derived from `OptimizerBaseInterp`.

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

```
48      /// \return    For each regimes (column) gives the solution for each particle , and cut
            (row)
49      ///            For a given simulation , cuts components (C) at a point stock \$ \bar S
            \f$  are given such that the cut is given by
50      ///            \f$  C[0] + \sum_{i=1}^d C[i] (S_i - \bat S_i)   \f$
51      virtual Eigen::ArrayXXd   stepOptimize(const   std::shared_ptr< StOpt::SpaceGrid> &
            p_grid, const Eigen::ArrayXd   &p_stock,
52                                             const std::vector< StOpt::ContinuationCuts  > &
                                                   p_condEsp) const = 0;
53
54
55      /// \brief defines a step in simulation
56      /// Control are recalculated during simulation using a local optimzation using the LP
57      /// \param p_grid          grid at arrival step after command
58      /// \param p_continuation  defines the continuation operator for each regime
59      /// \param p_state         defines the state value (modified)
60      /// \param p_phiInOut      defines the value functions (modified) : size number of
            functions  to follow
61      virtual void stepSimulate(const std::shared_ptr< StOpt::SpaceGrid>   &p_grid, const std
            ::vector< StOpt::ContinuationCuts  > &p_continuation,
62                                StOpt::StateWithStocks &p_state,
63                                Eigen::Ref<Eigen::ArrayXd> p_phiInOut) const = 0 ;
64
65
66      /// \brief Get the number of regimes allowed for the asset to be reached  at the
            current time step
67      ///    If \f$ t \f$ is the current time, and $\f$ dt \f$  the resolution step,  this is
             the number of regime allowed on \f$[ t- dt, t[\f$
68      virtual   int getNbRegime() const = 0 ;
69
70      /// \brief get the simulator back
71      virtual std::shared_ptr< StOpt::SimulatorDPBase > getSimulator() const = 0;
72
73      /// \brief get back the dimension of the control
74      virtual int getNbControl() const = 0 ;
75
76      /// \brief get size of the  function to follow in simulation
77      virtual int getSimuFuncSize() const = 0;
78
79  };
80  }
81  #endif /* OPTIMIZERDPCUTBASE_H */
```

We detail the different methods to implement in addition to the methods of `OptimizerBaseInterp`:

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

- the `stepSimulate` method is used after optimization using the continuation cuts values calculated in the optimization part. From a state `p_state` (storing the $X^{x,t}$), the continuation cuts values calculated in optimization `p_continuation`, the optimal cash flows along the current trajectory are stored in `p_phiInOut`.

In the case of a gas storage the Optimize object is given in the `OptimizeGasStorageCut.h` file.

The framework in optimization using some cuts methods

Once an Optimizer object describing to the Business problem to solve with cuts 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:

```
1    TransitionStepRegressionDPCutDist(const  shared_ptr<FullGrid> &p_pGridCurrent,
2                                      const  shared_ptr<FullGrid> &p_pGridPrevious,
3                                      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

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

with

- `p_phiIn` the vector (its size corresponds to the number of regimes) of 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:

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

with:

- `p_ar` is the archive where controls and solutions are dumped,
- `p_name` is a base name used in the archive to store the solution and the control,
- `p_phiInPrev` is the solution at the previous time step used to calculate the continuation cuts values that are stored,
- `p_condExp` is the conditional expectation object permitting to calculate conditional expectation of functions defined at the previous time step treated `p_phiInPrev`.
- `p_bOneFile` is set to one if the continuation cuts values calculated by each processor are dumped on a single file. Otherwise the continuation cuts values calculated by each processor are dumped on different files (one 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

```
1  // Copyright (C) 2019 EDF
2  // All Rights Reserved
3  // This code is published under the GNU Lesser General Public License (GNU LGPL)
4  #ifdef USE_MPI
5  #include <fstream>
6  #include <memory>
7  #include <functional>
8  #include <boost/lexical_cast.hpp>
9  #include <boost/mpi.hpp>
10 #include <Eigen/Dense>
11 #include "geners/BinaryFileArchive.hh"
12 #include "StOpt/core/grids/FullGrid.h"
13 #include "StOpt/regression/BaseRegression.h"
14 #include "StOpt/dp/FinalStepDPCutDist.h"
15 #include "StOpt/dp/TransitionStepRegressionDPCutDist.h"
16 #include "StOpt/core/parallelism/reconstructProc0Mpi.h"
17 #include "StOpt/dp/OptimizerDPCutBase.h"
18 #include "StOpt/dp/SimulatorDPBase.h"
19
20
21 using namespace std;
22 using namespace Eigen;
23
```

```
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();
    // final values
    vector< shared_ptr< ArrayXXd > >  valueCutsNext = StOpt::FinalStepDPCutDist(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)
    {
        ArrayXXd asset = simulator->stepBackwardAndGetParticles();
        // conditional expectation operator
        p_regressor->updateSimulations(((iStep == (simulator->getNbStep() - 1)) ? true :
            false), asset);
        // transition object
        StOpt::TransitionStepRegressionDPCutDist transStep(p_grid, p_grid, p_optimize);
        vector< shared_ptr< ArrayXXd > > valueCuts  = transStep.oneStep(valueCutsNext,
            p_regressor);
        transStep.dumpContinuationCutsValues(ar, nameAr, iStep, valueCutsNext, p_regressor,
             p_bOneFile);
        valueCutsNext = valueCuts;
    }
    // reconstruct a small grid for interpolation
    ArrayXd  valSim = StOpt::reconstructProc0Mpi(p_pointStock, p_grid, valueCutsNext[
        p_initialRegime], p_optimize->getDimensionToSplit());
    return ((world.rank() == 0) ? valSim.head(simulator->getNbSimul()).mean() : 0.);

}
#endif
```

An example without distribution of the data can be found in the `DynamicProgrammingByRegressionCut.cpp` 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

```
1  SimulateStepRegressionCutDist(gs::BinaryFileArchive &p_ar, const int &p_iStep, const
       std::string &p_nameCont,
2                                const shared_ptr<FullGrid> &p_pGridFollowing, const
                                     shared_ptr<OptimizerDPCutBase > &p_pOptimize,
3                                const bool &p_bOneFile)
```

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`

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

where:

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

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

```
// Copyright (C) 2019 EDF
// All Rights Reserved
// This code is published under the GNU Lesser General Public License (GNU LGPL)
#ifndef SIMULATEREGREGRESSIONCUTDIST_H
#define SIMULATEREGREGRESSIONCUTDIST_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/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 optimizaton achived
      with transition problems solved with cuts.
 *        A simple grid  is used
 *  \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 &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());
```

```
58      for (int istep = 0; istep < nbStep; ++istep)
59      {
60          StOpt::SimulateStepRegressionCutDist(ar, nbStep - 1 - istep, nameAr, p_grid,
                p_optimize, p_bOneFile).oneStep(states, costFunction);
61
62          // new stochastic state
63          Eigen::ArrayXXd particles =  simulator->stepForwardAndGetParticles();
64          for (int is = 0; is < simulator->getNbSimul(); ++is)
65              states[is].setStochasticRealization(particles.col(is));
66
67      }
68      // final : accept to exercise if not already done entirely (here suppose one function
            to follow)
69      for (int is = 0; is < simulator->getNbSimul(); ++is)
70          costFunction(0, is) += p_funcFinalValue(states[is].getRegime(), states[is].
                getPtStock(), states[is].getStochasticRealization())(0);
71
72      return costFunction.mean();
73  }
74
75  #endif /* SIMULATEREGRESSIONCUTDIST_H */
```

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

9.3 Solving the problem for $X_2^{x,t}$ stochastic

In this part we suppose that $X_2^{x,t}$ is controlled but is stochastic.

9.3.1 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` itself derived from `OptimizerBase`.

```
1   // Copyright (C) 2016 EDF
2   // All Rights Reserved
3   // This code is published under the GNU Lesser General Public License (GNU LGPL)
4   #ifndef OPTIMIZERNOREGRESSIONDPBASE_H
5   #define OPTIMIZERNOREGRESSIONDPBASE_H
6   #include <Eigen/Dense>
7   #include "StOpt/core/utils/StateWithStocks.h"
8   #include "StOpt/core/grids/SpaceGrid.h"
9   #include "StOpt/regression/BaseRegression.h"
10  #include "StOpt/regression/GridAndRegressedValue.h"
11  #include "StOpt/dp/SimulatorDPBase.h"
12  #include "StOpt/dp/OptimizerBaseInterp.h"
13
14  /** \file OptimizerNoRegressionDPBase.h
15   *  \brief Define an abstract class for Dynamic Programming problems solve by Monte Carlo
         but without regression method
16   *         to compute conditional expectation.
17   *     \author Xavier Warin
18   */
19
20  namespace StOpt
21  {
22
23  /// \class OptimizerNoRegressionDPBase OptimizerNoRegressionDPBase.h
24  ///  Base class for optimizer for Dynamic Programming solved without regression method  to
        compute conditional expectation.
25  class OptimizerNoRegressionDPBase : public OptimizerBaseInterp
```

```
{


public :

    OptimizerNoRegressionDPBase() {}

    virtual ~OptimizerNoRegressionDPBase() {}

    /// \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 ;

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



    /// \brief Defines a step in simulation using interpolation in controls
    /// \param p_grid          grid at arrival step after command
    /// \param p_control       defines the controls
    /// \param p_state         defines the state value (modified)
    /// \param p_phiInOut      defines the value function (modified): size number of functions to follow
    virtual void stepSimulateControl(const std::shared_ptr< StOpt::SpaceGrid>   &p_grid,
        const std::vector< StOpt::GridAndRegressedValue  > &p_control,
                                     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 \f$ t \f$ is the current time, and $\f$ dt \f$  the resolution step,  this is the number of regime allowed on \f$[ t- dt, t[\f$
    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 /* OPTIMIZERDPBASE_H */
```

In addition to the methods of `OptimizerBase` the following method is needed:

- the `stepOptimize` method is used in optimization. We want to calculate the optimal value regressed at current t_i at a grid point `p_stock` using a grid `p_grid` at the next date t_{i+1},

  From a grid point `p_stock` it calculates the function values regressed and the optimal controls regressed. It returns a pair where the

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

In this case of the optimization of an actualized portfolio with dynamic:

$$dX_2^{x,t} = X_2^{x,t} \frac{dX_1^{x,t}}{X_1^{x,t}}$$

where $X_1^{x,t}$ is the risky asset value, the Optimize object is given in the `OptimizePortfolio.h` file.

9.3.2 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:

```
1    TransitionStepDPDist(const  shared_ptr<FullGrid> &p_pGridCurrent,
2    const  shared_ptr<FullGrid> &p_pGridPrevious,
3    const  std::shared_ptr<BaseRegression> &p_regressorCurrent,
4    const  std::shared_ptr<BaseRegression> &p_regressorPrevious,
5    const  shared_ptr<OptimizerNoRegressionDPBase > &p_pOptimize):
```

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

```
1    std::pair< std::shared_ptr< std::vector< Eigen::ArrayXXd > > , std::shared_ptr< std::
         vector< Eigen::ArrayXXd  > > >   oneStep(const std::vector< Eigen::ArrayXXd  > &
         p_phiIn)
```

with

- `p_phiIn` the vector (its size corresponds to the number of regimes) of 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:

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

with:

- `p_ar` is the archive where controls and solutions are dumped,
- `p_name` is a base name used in the archive to store the solution and the control,
- `p_control` stores the optimal controls calculated at the current time step,
- `p_bOneFile` is set to one if the optimal controls calculated by each processor are dumped on a single file. Otherwise the optimal controls calculated by each processor are dumped on different files (one by processor). If the problem gives optimal control values on the global grid that can be stored in the memory of the computation node, it can be more interesting to dump the control values in one file for the simulation of the optimal policy.

Remark 17 *As for the* ***`TransitionStepDP`****, its* ***`dumpValues`*** *does not need a* ***`p_bOneFile`*** *argument: obviously optimal controls are stored in a single file.*

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

```cpp
// Copyright (C) 2016 EDF
// All Rights Reserved
// This code is published under the GNU Lesser General Public License (GNU LGPL)
#ifdef USE_MPI
#include <fstream>
#include <boost/mpi.hpp>
#include <memory>
#include <functional>
#include <boost/lexical_cast.hpp>
#include <Eigen/Dense>
#include "geners/BinaryFileArchive.hh"
#include "StOpt/core/grids/FullGrid.h"
#include "StOpt/regression/LocalConstRegression.h"
#include "StOpt/regression/GridAndRegressedValue.h"
#include "StOpt/dp/FinalStepDPDist.h"
#include "StOpt/dp/TransitionStepDPDist.h"
#include "StOpt/core/parallelism/reconstructProc0Mpi.h"
#include "test/c++/tools/dp/OptimizePortfolioDP.h"

using namespace std;
using namespace Eigen;

double  DynamicProgrammingPortfolioDist(const shared_ptr<StOpt::FullGrid> &p_grid,
                                        const shared_ptr<OptimizePortfolioDP> &p_optimize,
                                        const ArrayXi &p_nbMesh,
                                        const function<double(const int &, const ArrayXd &,
                                             const ArrayXd &)>  &p_funcFinalValue,
                                        const ArrayXd &p_initialPortfolio,
                                        const string   &p_fileToDump,
                                        const bool &p_bOneFile
                                       )
{
    // initialize simulation
    p_optimize->initializeSimulation();
    // store regressor
    shared_ptr<StOpt::LocalConstRegression> regressorPrevious;

    // store final regressed values in  object valuesStored
    shared_ptr< vector< ArrayXXd > > valuesStored = make_shared< vector<ArrayXXd> >(
        p_optimize->getNbRegime());
    {
        vector< shared_ptr< ArrayXXd > >  valuesPrevious = StOpt::FinalStepDPDist(p_grid,
            p_optimize->getNbRegime(), p_optimize->getDimensionToSplit())(p_funcFinalValue,
             *p_optimize->getCurrentSim());
        // regressor operator
        regressorPrevious = make_shared<StOpt::LocalConstRegression>(false, *p_optimize->
            getCurrentSim(), p_nbMesh);
        for (int iReg = 0; iReg < p_optimize->getNbRegime(); ++iReg)
            (*valuesStored)[iReg] = regressorPrevious->getCoordBasisFunctionMultiple(
                valuesPrevious[iReg]->transpose()).transpose();
    }
    boost::mpi::communicator world;
    string toDump = p_fileToDump ;
    // test if one file generated
    if (!p_bOneFile)
        toDump +=  "_" + boost::lexical_cast<string>(world.rank());
    shared_ptr<gs::BinaryFileArchive> ar;
    if ((!p_bOneFile) || (world.rank() == 0))
        ar = make_shared<gs::BinaryFileArchive>(toDump.c_str(), "w");
    // name for object in archive
    string nameAr = "OptimizePort";
    // iterate on time steps
    for (int iStep = 0; iStep < p_optimize->getNbStep(); ++iStep)
    {
        // step backward for simulations
        p_optimize->oneStepBackward();
        // create regressor at the given date
        bool bZeroDate = (iStep == p_optimize->getNbStep() - 1);
```

```
63          shared_ptr<StOpt::LocalConstRegression> regressorCur = make_shared<StOpt::
                LocalConstRegression>(bZeroDate, *p_optimize->getCurrentSim(), p_nbMesh);
64          // transition object
65          StOpt::TransitionStepDPDist transStep(p_grid, p_grid, regressorCur,
                regressorPrevious, p_optimize);
66          pair< shared_ptr< vector< ArrayXXd> >, shared_ptr< vector< ArrayXXd > > >
                valuesAndControl = transStep.oneStep(*valuesStored);
67          // dump control values
68          transStep.dumpValues(ar, nameAr, iStep,  *valuesAndControl.second, p_bOneFile);
69          valuesStored = valuesAndControl.first;
70          // shift regressor
71          regressorPrevious = regressorCur;
72      }
73      // interpolate at the initial stock point and initial regime( 0 here)  (take first
            particle)
74      shared_ptr<ArrayXXd> topRows = make_shared<ArrayXXd>((*valuesStored)[0].topRows(1));
75      return StOpt::reconstructProc0Mpi(p_initialPortfolio, p_grid, topRows, p_optimize->
            getDimensionToSplit()).mean();
76  }
77  #endif
```

An example without distribution of the data can be found in the `DynamicProgrammingPortfolio.cpp` file.

9.3.3 The framework in simulation

Not special framework is available in simulation. Use the function `SimulateStepRegressionControl` or `SimulateStepRegressionControlDist` described in the section 9.2.2.

9.4 Solving stock problems with trees

In this section we detail how to solve problems

- Either by discretizing the control,
- Either by solving a Linear Program problem approximating the Bellman values by cuts.

9.4.1 Solving dynamic programming problems with control discretization

Requirement to use the framework

The `OptimizerDPTreeBase` is the based object from which each business object should be derived.

```
1   // Copyright (C) 2019 EDF
2   // All Rights Reserved
3   // This code is published under the GNU Lesser General Public License (GNU LGPL)
4   #ifndef OPTIMIZERDPTREEBASE_H
5   #define OPTIMIZERDPTREEBASE_H
6   #include <Eigen/Dense>
7   #include "StOpt/core/grids/SpaceGrid.h"
8   #include "StOpt/tree/Tree.h"
9   #include "StOpt/tree/StateTreeStocks.h"
10  #include "StOpt/tree/ContinuationValueTree.h"
11  #include "StOpt/tree/GridTreeValue.h"
```

```cpp
#include "StOpt/dp/SimulatorDPBaseTree.h"
#include "StOpt/dp/OptimizerBase.h"

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

namespace StOpt
{

/// \class OptimizerDPTreeBase OptimizerDPTreeBase.h
///  Base class for optimizer for Dynamic Programming  with tree methods
class OptimizerDPTreeBase : public OptimizerBase
{


public :

    OptimizerDPTreeBase() {}

    virtual ~OptimizerDPTreeBase() {}

    /// \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 ;

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


    /// \brief defines a step in simulation
    /// Notice that this implementation is not optimal but is convenient if the control is discrete.
    /// By avoiding interpolation in control we avoid non admissible control
    /// Control are recalculated during simulation.
    /// \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::GridTreeValue  > &p_continuation,
                              StOpt::StateTreeStocks &p_state,
                              Eigen::Ref<Eigen::ArrayXd> p_phiInOut) const = 0 ;


    /// \brief Defines a step in simulation using interpolation in controls
    /// \param p_grid          grid at arrival step after command
```

```
71      /// \param p_control       defines the controls
72      /// \param p_state         defines the state value (modified)
73      /// \param p_phiInOut      defines the value function (modified): size number of
            functions to follow
74      virtual void stepSimulateControl(const std::shared_ptr< StOpt::SpaceGrid>   &p_grid,
            const std::vector< StOpt::GridTreeValue  > &p_control,
75                                       StOpt::StateTreeStocks &p_state,
76                                       Eigen::Ref<Eigen::ArrayXd> p_phiInOut) const = 0 ;
77
78
79
80      /// \brief Get the number of regimes allowed for the asset to be reached  at the
            current time step
81      ///    If \f$ t \f$ is the current time, and $\f$ dt \f$  the resolution step,  this is
             the number of regime allowed on \f$[ t- dt, t[\f$
82      virtual   int getNbRegime() const = 0 ;
83
84      /// \brief get the simulator back
85      virtual std::shared_ptr< StOpt::SimulatorDPBaseTree > getSimulator() const = 0;
86
87      /// \brief get back the dimension of the control
88      virtual int getNbControl() const = 0 ;
89
90      /// \brief get size of the  function to follow in simulation
91      virtual int getSimuFuncSize() const = 0;
92
93  };
94  }
95  #endif /* OPTIMIZERDPTREEBASE_H */
```

As the `OptimizerDPTreeBase` class is derived from `OptimizerBase`, we detail the additional methods required:

- the `stepOptimize` method is used in optimization. We want to calculate the optimal value at current t_i at a grid point `p_stock` using a grid `p_grid` at the next date t_{i+1}, the continuation values for all regimes `p_condEsp` permitting to calculate conditional expectation of the optimal value function calculated at the previously treated time step t_{i+1}. From a grid point `p_stock` it calculates the function values and the optimal controls. It returns a pair where the
  - first element is a matrix (first dimension is the number of nodes, second dimension the number of regimes) giving the function value,
  - second element is a matrix (first dimension is the number of nodes, 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: this continuation values are stored in a GridTreeValue object for interpolation. From a state `p_state` (storing the $X^{x,t}$), the continuation values calculated in optimization `p_continuation`, the optimal functions values stored in `p_phiInOut`.
- the `stepSimulateControl` simulate the strategy by direct interpolation of the control for a given node in the tree (sampled) and a position in the stock.

The framework

Most object use with regression have their equivalent with tree methods.
In optimization:

`TransitionStepTreeDPDist` permits to solve the transition at one date for all the grid point and the nodes of the tree using distribution (MPI).

```
1    TransitionStepTreeDPDist(const  std::shared_ptr<FullGrid> &p_pGridCurrent,
2                             const  std::shared_ptr<FullGrid> &p_pridPrevious,
3                             const  std::shared_ptr<OptimizerDPTreeBase > &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

A similar object is available without the MPI distribution framework `TransitionStep TreeDP` with still enabling parallelization with threads and MPI on the calculations on the full grid points.
The main method is

```
1    std::pair< std::vector< std::shared_ptr< Eigen::ArrayXXd > >, std::vector<  std::
         shared_ptr<  Eigen::ArrayXXd > > >  oneStep(const std::vector< std::shared_ptr<
         Eigen::ArrayXXd > > &p_phiIn,
2            const std::shared_ptr< Tree>     &p_condExp) const
```

with

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

returning a pair:

- first element is a vector of matrix with new optimal values at the current time step (each element of the vector corresponds to a regime and each matrix is a number of nodes at current date 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 nodes at the current date by number of stock points matrix).

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

```
1     void dumpContinuationValues(std::shared_ptr<gs::BinaryFileArchive> p_ar,
2                                 const std::string &p_name, const int &p_iStep,
3                                 const std::vector< std::shared_ptr< Eigen::ArrayXXd > > &
                                      p_phiIn,
4                                 const std::vector< std::shared_ptr< Eigen::ArrayXXd > > &
                                      p_control,
5                                 const std::shared_ptr< Tree>      &p_tree,
6                                 const bool &p_bOneFile) const;
```

with:

- `p_ar` is the archive where controls and solutions are dumped,
- `p_name` is a base name used in the archive to store the solution and the control,
- `p_phiInPrev` is the solution at the previous time step used to calculate the continuation values that are stored,
- `p_control` stores the optimal controls calculated at the current time step,
- `p_tree` is the conditional expectation object permitting to calculate conditional expectation of functions defined at the previous time step treated `p_phiInPrev` and permitting to store a representation of the optimal control.
- `p_bOneFile` is set to one if the continuation and optimal controls calculated by each processor are dumped on a single file. Otherwise the continuation and optimal controls calculated by each processor are dumped on different files (one by processor).

Remark 18 *The `p_bOneFile` is not present for `TransitionStepTreeDP` objects.*

In simulation (see detail in section for regressions)

- A first object permitting the recalculation of the optimal control in simulation.

```
SimulateStepTreeDist(gs::BinaryFileArchive &p_ar, const int &p_iStep, const std::
    string &p_nameCont,
                        const  std::shared_ptr<FullGrid> &p_pGridFollowing, const
                            std::shared_ptr<OptimizerDPTreeBase > &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 from one time step to the following one,
  - `p_OneFile` equal to `true` if a single archive is used to store continuation values.

  This object implements the method `oneStep`

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

  where:

  - `p_statevector` store the states for the all the simulations: this state is updated by application of the optimal command,

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

Remark 19 *The version without distribution of the Bellman values is available in* ***`SimulateStepTree`*** *object.*

- A second object directly interpolating the control

```
SimulateStepTreeControlDist(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<OptimizerDPTreeBase > &
                                    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 `true` if a single archive is used to store continuation values.

This object implements the method `oneStep`

```
  void oneStep(std::vector<StateTreeStocks > &p_statevector, Eigen::ArrayXXd  &
      p_phiInOut) const;
```

where:

– `p_statevector` stores for all the simulations the state: this state is updated by application of the optimal commands,

– `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.

Remark 20 *The non distributed version is given by the* ***`SimulateStepTreeControl`*** *object.*

9.4.2 Solving Dynamic Programming by solving LP problems

This approach can only be used for continuous problems with convex or concave Bellman values. See section 9.2.3 for some explanation of the cut approximation.

Requirement to use the framework

The business object developed should derive from the `OptimizerDPCutBase` object derived from the `OptimizerBase` object.

```
// Copyright (C) 2019 EDF
// All Rights Reserved
// This code is published under the GNU Lesser General Public License (GNU LGPL)
#ifndef OPTIMIZERDPCUTTREEBASE_H
#define OPTIMIZERDPCUTTREEBASE_H
#include <Eigen/Dense>
#include "StOpt/core/utils/StateWithStocks.h"
#include "StOpt/core/grids/SpaceGrid.h"
#include "StOpt/tree/Tree.h"
#include "StOpt/tree/StateTreeStocks.h"
#include "StOpt/tree/ContinuationCutsTree.h"
#include "StOpt/dp/SimulatorDPBaseTree.h"
#include "StOpt/dp/OptimizerBase.h"

/** \file OptimizerDPCutBase.h
 *  \brief Define an abstract class for Dynamic Programming problems solved by tree methods using cust to approximate
 *         Bellman values
 *     \author Xavier Warin
 */

namespace StOpt
{

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


public :

    OptimizerDPCutTreeBase() {}

    virtual ~OptimizerDPCutTreeBase() {}

    /// \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 ;

    /// \brief defines a step in optimization
    /// \param p_grid      grid at arrival step after command
    /// \param p_stock     coordinates of the stock point to treat
    /// \param p_condEsp   continuation values for each regime
    /// \return    For each regimes (column) gives the solution for each particle , and cut (row)
```

```
50      ///            For a given simulation , cuts components (C) at a point stock \$ \bar S
            \f$  are given such that the cut is given by
51      ///            \f$  C[0] + \sum_{i=1}^d C[i] (S_i - \bat S_i)   \f$
52      virtual Eigen::ArrayXXd   stepOptimize(const   std::shared_ptr< StOpt::SpaceGrid> &
            p_grid, const Eigen::ArrayXd   &p_stock,
53                                             const std::vector< StOpt::ContinuationCutsTree
                                                   > &p_condEsp) const = 0;
54
55
56      /// \brief defines a step in simulation
57      /// Control are recalculated during simulation using a local optimzation using the LP
58      /// \param p_grid          grid at arrival step after command
59      /// \param p_continuation  defines the continuation operator for each regime
60      /// \param p_state         defines the state value (modified)
61      /// \param p_phiInOut      defines the value functions (modified) : size number of
            functions  to follow
62      virtual void stepSimulate(const std::shared_ptr< StOpt::SpaceGrid>   &p_grid, const std
            ::vector< StOpt::ContinuationCutsTree  > &p_continuation,
63                                StOpt::StateTreeStocks &p_state,
64                                Eigen::Ref<Eigen::ArrayXd> p_phiInOut) const = 0 ;
65
66
67      /// \brief Get the number of regimes allowed for the asset to be reached  at the
            current time step
68      ///    If \f$ t \f$ is the current time, and $\f$ dt \f$  the resolution step,  this is
             the number of regime allowed on \f$[ t- dt, t[\f$
69      virtual   int getNbRegime() const = 0 ;
70
71      /// \brief get the simulator back
72      virtual std::shared_ptr< StOpt::SimulatorDPBaseTree > getSimulator() const = 0;
73
74      /// \brief get back the dimension of the control
75      virtual int getNbControl() const = 0 ;
76
77      /// \brief get size of the  function to follow in simulation
78      virtual int getSimuFuncSize() const = 0;
79
80  };
81  }
82  #endif /* OPTIMIZERDPCUTTREEBASE_H */
```

We detail the different methods to implement in addition to the methods of `OptimizerBase`:

- the `stepOptimize` method is used in optimization. We want to calculate the optimal value and the corresponding sensibilities with respect to the stocks at current t_i at a grid point `p_stock` using a grid `p_grid` at the next date t_{i+1}, the continuation cuts values for all regimes `p_condEsp` permitting to calculate an upper estimation (when maximizing) of conditional expectation of the optimal values using some optimization calculated at the previously treated time step t_{i+1}. From a grid point `p_stock` it calculates the function values and the corresponding sensibilities. It returns a matrix (first dimension is the number of nodes at the current cate 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 are stored in `p_phiInOut`.

In the case of a gas storage the Optimize object is given in the `OptimizeGasStorageTreeCut.h` file.

The framework in optimization using some cuts methods

We process exactly as in section 9.2.3. The framework provides a `TransitionStepTreeDPCutDist` object in MPI that permits to solve the optimization problem with distribution of the data on one time step with the following constructor:

```
1    TransitionStepTreeDPCutDist(const  std::shared_ptr<FullGrid> &p_pGridCurrent,
2                                const  std::shared_ptr<FullGrid> &p_pGridPrevious,
3                                const  std::shared_ptr<OptimizerDPCutTreeBase > &
                                    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 21 *A similar object is available without the MPI distribution framework* ***`TransitionStepTreeDPCut`*** *with still enabling parallelization with threads and MPI on the calculations on the full grid points.*

The main method is

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

with

- `p_phiIn` the vector (its size corresponds to the number of regimes) of 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 nodes at next date by the number of stock plus one. The number of columns is equal to the number of stock points on the grid. In the row, the number of simulations by the number of stock plus one value are stored as follows:
  - The first values (number of nodes at next date : 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 nodes at the current date by (the number of storage plus one)) by the number of stock points). The structure of the output is then similar to the 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:

```
    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< Tree>     &
                                        p_condExp,
                                    const bool &p_bOneFile) const
```

with:

- `p_ar` is the archive where controls and solutions are dumped,
- `p_name` is a base name used in the archive to store the solution and the control,
- `p_phiInPrev` is the solution at the previous time step used to calculate the continuation cuts values that are stored,
- `p_condExp` is the conditional expectation object permitting to calculate conditional expectation of functions defined at the previous time step treated `p_phiInPrev`.
- `p_bOneFile` is set to one if the continuation cuts values calculated by each processor are dumped on a single file. Otherwise the continuation cuts values calculated by each processor are dumped on different files (one by processor).

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

```
// Copyright (C) 2019 EDF
// All Rights Reserved
// This code is published under the GNU Lesser General Public License (GNU LGPL)
#ifdef 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/tree/Tree.h"
#include "StOpt/dp/FinalStepDPCutDist.h"
#include "StOpt/dp/TransitionStepTreeDPCutDist.h"
#include "StOpt/core/parallelism/reconstructProc0Mpi.h"
#include "StOpt/dp/OptimizerDPCutTreeBase.h"
#include "StOpt/dp/SimulatorDPBaseTree.h"


using namespace std;
using namespace Eigen;

double  DynamicProgrammingByTreeCutDist(const shared_ptr<StOpt::FullGrid> &p_grid,
                                        const shared_ptr<StOpt::OptimizerDPCutTreeBase > &
                                            p_optimize,
                                        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::SimulatorDPBaseTree> simulator = p_optimize->getSimulator();
    // final values
```

```
35      vector< shared_ptr< ArrayXXd > >  valueCutsNext = StOpt::FinalStepDPCutDist(p_grid,
            p_optimize->getNbRegime(), p_optimize->getDimensionToSplit())(p_funcFinalValue,
            simulator->getNodes());
36      // dump
37      boost::mpi::communicator world;
38      string toDump = p_fileToDump ;
39      // test if one file generated
40      if (!p_bOneFile)
41          toDump +=  "_" + boost::lexical_cast<string>(world.rank());
42      shared_ptr<gs::BinaryFileArchive> ar;
43      if ((!p_bOneFile) || (world.rank() == 0))
44          ar = make_shared<gs::BinaryFileArchive>(toDump.c_str(), "w");
45      // name for object in archive
46      string nameAr = "ContinuationTree";
47      for (int iStep = 0; iStep < simulator->getNbStep(); ++iStep)
48      {
49          simulator->stepBackward();
50          // probabilities
51          std::vector<double>  proba = simulator->getProba();
52          // get connection between nodes
53          std::vector< std::vector<std::array<int, 2>  > >  connected = simulator->
                getConnected();
54          // conditional expectation operator
55          shared_ptr<StOpt::Tree> tree = std::make_shared<StOpt::Tree>(proba, connected);
56          // transition object
57          StOpt::TransitionStepTreeDPCutDist transStep(p_grid, p_grid, p_optimize);
58          vector< shared_ptr< ArrayXXd > > valueCuts  = transStep.oneStep(valueCutsNext, tree
                );
59          transStep.dumpContinuationCutsValues(ar, nameAr, iStep, valueCutsNext, tree,
                p_bOneFile);
60          valueCutsNext = valueCuts;
61      }
62      // reconstruct a small grid for interpolation
63      ArrayXd  valSim = StOpt::reconstructProc0Mpi(p_pointStock, p_grid, valueCutsNext[
            p_initialRegime], p_optimize->getDimensionToSplit());
64      return ((world.rank() == 0) ? valSim(0) : 0.);
65
66  }
67  #endif
```

An example without distribution of the data can be found in the `DynamicProgramming ByTreeCut.cpp` file.

Using the framework in simulation

Similarly to section 9.2.3, we can use the cuts calculated in optimization to test the optimal strategy found. In order to simulate one step of the optimal policy, an object `SimulateStep TreeCutDist` is provided with constructor

```
1   SimulateStepTreeCutDist(gs::BinaryFileArchive &p_ar,  const int &p_iStep,  const std::
        string &p_nameCont,
2                           const  std::shared_ptr<FullGrid> &p_pGridFollowing,
3                           const  std::shared_ptr<OptimizerDPCutTreeBase > &p_pOptimize,
4                           const bool &p_bOneFile);
```

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

This object implements the method `oneStep`

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

where:

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

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

```
// Copyright (C) 2019 EDF
// All Rights Reserved
// This code is published under the GNU Lesser General Public License (GNU LGPL)
#ifndef SIMULATEREGTREECUTDIST_H
#define SIMULATEREGTREECUTDIST_H
#include <functional>
#include <memory>
#include <Eigen/Dense>
#include <boost/mpi.hpp>
#include "geners/BinaryFileArchive.hh"
#include "StOpt/core/grids/FullGrid.h"
#include "StOpt/tree/StateTreeStocks.h"
#include "StOpt/dp/SimulateStepTreeCutDist.h"
#include "StOpt/dp/OptimizerDPCutBase.h"
#include "StOpt/dp/SimulatorDPBase.h"


/** \file SimulateTreeCutDist.h
 *  \brief Defines a simple program showing how to use simulations when optimizaton achived
      with transition problems solved with cuts and uncertainties on a tree
 *        A simple grid  is used
 *  \author Xavier Warin
 */


/// \brief Simulate the optimal strategy , mpi version, Bellman cuts used to allow LP
    resolution of transition problems when uncertainties are defined on a tree
/// \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
```

```
29 /// \param p_pointStock             initial point stock
30 /// \param p_initialRegime          regime at initial date
31 /// \param p_fileToDump             name associated to dumped bellman values
32 /// \param p_bOneFile               do we store continuation values  in only one file
33 double SimulateTreeCutDist(const std::shared_ptr<StOpt::FullGrid> &p_grid,
34                            const std::shared_ptr<StOpt::OptimizerDPCutTreeBase > &
                                   p_optimize,
35                            const std::function< Eigen::ArrayXd(const int &, const Eigen::
                                   ArrayXd &, const Eigen::ArrayXd &)>  &p_funcFinalValue,
36                            const Eigen::ArrayXd &p_pointStock,
37                            const int &p_initialRegime,
38                            const std::string   &p_fileToDump,
39                            const bool &p_bOneFile)
40 {
41     boost::mpi::communicator world;
42     // from the optimizer get back the simulator
43     std::shared_ptr< StOpt::SimulatorDPBaseTree> simulator = p_optimize->getSimulator();
44     int nbStep = simulator->getNbStep();
45     std::vector< StOpt::StateTreeStocks> states;
46     states.reserve(simulator->getNbSimul());
47     for (int is = 0; is < simulator->getNbSimul(); ++is)
48         states.push_back(StOpt::StateTreeStocks(p_initialRegime, p_pointStock, 0));
49     std::string toDump = p_fileToDump ;
50     // test if one file generated
51     if (!p_bOneFile)
52         toDump +=  "_" + boost::lexical_cast<std::string>(world.rank());
53     gs::BinaryFileArchive ar(toDump.c_str(), "r");
54     // name for continuation object in archive
55     std::string nameAr = "ContinuationTree";
56     // cost function
57     Eigen::ArrayXXd costFunction = Eigen::ArrayXXd::Zero(p_optimize->getSimuFuncSize(),
           simulator->getNbSimul());
58     for (int istep = 0; istep < nbStep; ++istep)
59     {
60         StOpt::SimulateStepTreeCutDist(ar, nbStep - 1 - istep, nameAr, p_grid, p_optimize,
               p_bOneFile).oneStep(states, costFunction);
61
62         // new date
63         simulator->stepForward();
64         for (int is = 0; is < simulator->getNbSimul(); ++is)
65             states[is].setStochasticRealization(simulator->getNodeAssociatedToSim(is));
66     }
67     // final : accept to exercise if not already done entirely (here suppose one function
           to follow)
68     for (int is = 0; is < simulator->getNbSimul(); ++is)
69         costFunction(0, is) += p_funcFinalValue(states[is].getRegime(), states[is].
               getPtStock(), simulator->getValueAssociatedToNode(states[is].
               getStochasticRealization()))(0);
70
71     return costFunction.mean();
72 }
73
74 #endif /* SIMULATETREECUTDIST_H */
```

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

Chapter 10

The Python API

10.1 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

```
# 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 dp.TransitionStepRegressionDP as trans
import dp.FinalStepDP as final


def DynamicProgrammingByRegression(p_grid, p_optimize, p_regressor, p_funcFinalValue,
    p_pointStock, p_initialRegime, p_fileToDump, key1="Continuation" , key2 = "Control"):

    # from the optimizer get back the simulation
    simulator = p_optimize.getSimulator()
    # final values
    valuesNext = final.FinalStepDP(p_grid, p_optimize.getNbRegime()).operator(
        p_funcFinalValue, simulator.getParticles())

    archiveToWrite = StOptGeners.BinaryFileArchive(p_fileToDump, "w")
    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 = trans.TransitionStepRegressionDP(p_grid, p_grid, p_optimize)
        valuesAndControl = transStep.oneStep(valuesNext, p_regressor)
        valuesNext = valuesAndControl[0]
        control =  valuesAndControl[1]
```

```
        # Dump the continuation values in the archive:
        archiveToWrite.dumpGridAndRegressedValue(key1, nsteps - 1 - iStep, valuesNext,
            p_regressor, p_grid)
        archiveToWrite.dumpGridAndRegressedValue(key2, nsteps - 1 - iStep, control,
            p_regressor, p_grid)

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

Some examples are available in the test directory (for example for swing options).
Another approach more effective in term of computational cost consists in mapping the simulator object derived from the `SimulatorDPBase` object and optimizer object derived from the `OptimizerDPBase` object and to use the high level python mapping of `TransitionStepRegressionDP` and `SimulateStepRegression`. In the test part of the library some Black-Scholes simulator and some Mean reverting simulator for a future curve deformation are developed and some examples of the mapping are achieved in the `Pybind11Simulators.cpp` file. Similarly the optimizer class for swings options, optimizer for a fictitious swing in dimension 2, optimizer for a gas storage, optimizer for a gas storage with switching cost are mapped to python in the `Pybind11Optimizers.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:

```
# Copyright (C) 2016 EDF
# All Rights Reserved
# This code is published under the GNU Lesser General Public License (GNU LGPL)
import math
import imp
import numpy as np
import unittest
import StOptGrids
import StOptReg
import StOptGlobal
import StOptGeners
import Utils
import Simulators as sim
import Optimizers as opt

# unit test for global shape
############################

class OptimizerConstruction(unittest.TestCase):

    def test(self):
        try:
            imp.find_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
            ###########
            N = 3   # number of exercise dates
            swiOpt = opt.OptimizerSwingBlackScholes(payOff,N)
            # link simulator to optimizer
            swiOpt.setSimulator(bsSim)
            # archive
            ########
            ar = StOptGeners.BinaryFileArchive("Archive","w")
            # regressor
            ###########
            nMesh = np.array([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], dtype=np.int32)
            gridArrival =  StOptGrids.RegularSpaceGrid(lowValues,step,nbStep)
            gridStart   =  StOptGrids.RegularSpaceGrid(lowValues,step,nbStep-1)
            # pay off function for swing
            ############################
            payOffBasket = Utils.BasketCall(strike);
            payoff = Utils.PayOffSwing(payOffBasket,N)
            dir(payoff)
            # final step
            #############
            asset =bsSim.getParticles()
            fin = StOptGlobal.FinalStepDP(gridArrival,1)
            values = fin.set( payoff,asset)
            # transition time step
            ######################
            # on step backward and get  asset
            asset = bsSim.stepBackwardAndGetParticles()
            # update regressor
            regressor.updateSimulations(0,asset)
            transStep = StOptGlobal.TransitionStepRegressionDP(gridStart,gridArrival,swiOpt
                )
            valuesNextAndControl=transStep.oneStep(values,regressor)
            transStep.dumpContinuationValues(ar,"Continuation",1,valuesNextAndControl[0],
                valuesNextAndControl[1],regressor)
            # simulate time step
            ####################
            nbSimul= 10
            vecOfStates =[] # state of each simulation
            for i in np.arange(nbSimul):
                # one regime, all with same stock level (dimension 2), same realization of
                    simulation (dimension 3)
                vecOfStates.append(StOptGlobal.StateWithStocks(1, np.array([0.]) , np.zeros
                    (1)))
            arRead = StOptGeners.BinaryFileArchive("Archive","r")
            simStep = StOptGlobal.SimulateStepRegression(arRead,1,"Continuation",
                gridArrival,swiOpt)
            phi = np.zeros((1,nbSimul))
            VecOfStateNext = simStep.oneStep(vecOfStates, phi)

```

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

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)

```
# 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.FinalStepDP(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:

```
# 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])

    # final : accept to exercise if not already done entirely
    for i in range(simulator.getNbSimul()) :
        costFunction[0,i] += p_funcFinalValue.set(states[i].getRegime(), states[i].
            getPtStock(), states[i].getStochasticRealization()) * simulator.getActu()

    # average gain/cost
    return costFunction.mean()
```

Equivalent using MPI and the distribution of calculations and data can be used using the `mpi4py` package. An example of its use can be found in the MPI version of a swing optimization and valorization.

10.2 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.

10.2.1 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.

```
# 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 StOptGrids
import StOptReg
import StOptGeners


# unit test for dumping binary archive of regressed value and Read then
#########################################################################

class testBinaryArchiveStOpt(unittest.TestCase):


      def testSimpleStorageAndLectureRecGrid(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 perburbed 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
        archiveToWrite.dumpGridAndRegressedValue("toStore", 3,listToReg, regressor,grid)


        # Read the regressed values
        ###########################
        archiveToRead =  StOptGeners.BinaryFileArchive("MyArchive","r")
        contValues = archiveToRead.readGridAndRegressedValue(3,"toStore")


        # list of 2 continuation values
        ###############################
        stockPoint = np.array([1.5,3.,5.])
        uncertainty = np.array([0.])
        value =contValues[0].getValue(stockPoint,uncertainty)


    # non regular grid
    def testSimpleStorageAndLectureNonRegular(self):

        # create the Legendre grid
        grid = StOptGrids.GeneralSpaceGrid([[ 1., 1.7, 2.4, 3.1, 3.8 ],
                                            [2., 4.3, 6.6, 8.9, 11.2, 15.],
                                            [3., 4.9, 5.8, 7.7, 10.,20.]])

        # simulate the perburbed 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
        archiveToWrite.dumpGridAndRegressedValue("toStore", 3,listToReg, regressor,grid)


```

```
        # Read the regressed values
        ###########################
        archiveToRead =  StOptGeners.BinaryFileArchive("MyArchive","r")
        contValues = archiveToRead.readGridAndRegressedValue(3,"toStore")


        # list of 2 continuation values
        ###############################
        stockPoint = np.array([1.5,3.,5.])
        uncertainty = np.array([0.])
        value =contValues[0].getValue(stockPoint,uncertainty)

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

10.2.2 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 particles (meaning that one or more functions are regressed), it it 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.

```
# Copyright (C) 2017 EDF
# All Rights Reserved
# This code is published under the GNU Lesser General Public License (GNU LGPL)
import numpy as np
import StOptReg
import StOptGeners

# unit test to show how to store some regression object and basis function coefficients associated
#
    ##########################################################################################################


def  createData():

   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
```

```
31
32      r,c = XX1.shape
33
34      X =  np.reshape(XX1,(r*c,1))[:,0]
35      I =  np.reshape(XX2,(r*c,1))[:,0]
36      Y =  np.reshape(Y,(r*c,1))[:,0]
37
38      xMatrix = np.zeros((2,len(X)))
39      xMatrix[0,:] = X
40      xMatrix[1,:] = I
41
42      return xMatrix, Y
43
44
45
46
47  # main
48
49  xMatrix, y = createData()
50
51  # 2 dimensional regression 2 by 2 meshes
52  nbMesh = np.array([2,2],dtype=np.int32)
53  regressor = StOptReg.LocalLinearRegression(False,xMatrix,nbMesh)
54
55  # coefficients
56  coeff = regressor.getCoordBasisFunction(y)
57
58  print("Regressed coeff", coeff)
59
60
61  # store them in a matrix
62  coeffList = np.zeros(shape=(1,3*2*2))
63  coeffList[0,:]=coeff.transpose()
64
65
66  # archive write for  regressors
67  archiveWriteForRegressor =StOptGeners.BinaryFileArchive("archive","w")
68
69  # store
70  step =1
71  archiveWriteForRegressor.dumpSome2DArray("RegCoeff",step,coeff)
72  archiveWriteForRegressor.dumpSomeRegressor("Regressor",step,regressor)
73
74  # archive Read for  regressors
75  archiveReadForRegressor =StOptGeners.BinaryFileArchive("archive","r")
76
77  # get back
78  values = archiveReadForRegressor.readSome2DArray("RegCoeff",step)
79  reg = archiveReadForRegressor.readSomeRegressor("Regressor",step)
80  print("Regressed coeff ", values)
81  print("Reg",reg)
```

Chapter 11

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 [48]
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 $\mathcal{T} = \{t_0 = 0, t_1, \ldots, t_{N-1}, t_N = T\}$ 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 $t = t_0, \ldots, t_N$.
At last we suppose that the risk free rate is zero so that a bond has always a value of 1.

11.1 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 [33], [6] definition, we define:

Definition 1 *A $(\bar{m}, \bar{l})$ self-financing strategy $\mathcal{V} = (\mathcal{V}_{t_i})_{i=0,\ldots,N-1}$ is a pair of adapted process $(m_{t_i}, l_{t_i})_{i=0,\ldots,N-1}$ defined for $(\bar{m}, \bar{l}) \in (0,\infty) \times (0,\infty)$ such that:*

- $0 \leq m_{t_i} \leq \bar{m}, \quad 0 \leq l_{t_i} \leq \bar{l} \qquad P.a.s. \quad \forall i = 0, \ldots, N-1,$
- $m_{t_i} l_{t_i} = 0 \quad P.a.s. \quad \forall i = 0, \ldots, N-1.$

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 23 *The strategies defined in [33] 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 $\nu \in \Theta^{(\bar{m},\bar{l})}$.

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:

$$x - \sum_{i=0}^{N-1}(1+\lambda)l_{t_i}S_{t_i} + \sum_{i=0}^{N-1}(1-\lambda)m_{t_i}S_{t_i} + \sum_{i=0}^{N-1} l_{t_i}S_{t_N} - \sum_{i=0}^{N-1} m_{t_i}S_{t_N}. \tag{11.1}$$

Remark 24 *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 [33] [6], we define the risk minimal strategy minimizing the $\mathcal{L}^2$ risk of the hedge portfolio:

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

$$\begin{aligned}\hat{\mathcal{V}} =& \arg \min_{\mathcal{V}=(m,l)\in\Theta^{(\bar{m},\bar{l})}} \mathbb{E}[(H - x + \sum_{i=0}^{N-1}(1+\lambda)l_{t_i}S_{t_i} - \\ & \sum_{i=0}^{N-1}(1-\lambda)m_{t_i}S_{t_i} - \sum_{i=0}^{N-1} l_{t_i}S_{t_N} + \sum_{i=0}^{N-1} m_{t_i}S_{t_N})^2].\end{aligned} \tag{11.2}$$

11.2 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 $\nu = (\nu_i)_{i=0,\dots,N-1}$ with:

$$\nu_i = \sum_{j=0}^{i}(m_{t_j} - l_{t_j}), \forall i = 0,\dots,N-1.$$

Using the property that $m_{t_i}l_{t_i} = 0, \quad \forall i = 0,\dots,N-1$, we get $|\nu_i - \nu_{i-1}| = l_{t_i} + m_{t_i}$ with the convention that $\nu_{-1} = 0$ and

$$G_T(\mathcal{V}) = \hat{G}_T(\nu) = x - \sum_{i=0}^{N-1}\lambda|\Delta\nu_{i-1}|S_{t_i} + \sum_{i=0}^{N-1}\nu_i\Delta S_i,$$

where $\Delta S_i = S_{t_{i+1}} - S_{t_i}$, $\Delta\nu_i = \nu_{i+1} - \nu_i$.
We then introduce $\hat{\Theta}^{(\bar{m},\bar{l})}$ the set of adapted random variable $(\nu_i)_{i=0,\dots,N-1}$ such that

$$-\bar{m} \le \nu_i - \nu_{i-1} \le \bar{l}, \forall i = 1,\dots,N-1.$$

The problem (11.2) can be rewritten as done in [40] finding $\hat{\nu} = (\hat{\nu}_i)_{i=0,\dots,N-1}$ satisfying:

$$\hat{\nu} = \arg \min_{\nu \in \hat{\Theta}^{(\bar{m},\bar{l})}} \mathbb{E}[\big(H - x - \hat{G}_T(\nu)\big)^2]. \tag{11.3}$$

We introduce the spaces κ_i, $i = 0, \dots, N$ of the $\mathcal{F}_{t_i}$-measurable and square integrable random variables. We define for $i \in 0, \dots, N$, $V_i \in \kappa_i$ as:

$$\begin{aligned} V_N =& H, \\ V_i =& \mathbb{E}[H - \sum_{j=i}^{N-1} \nu_j \Delta S_j + \lambda \sum_{j=i}^{N-1} |\Delta \nu_{j-1}| S_{t_j} \quad |\mathcal{F}_{t_i}], \forall i = 0, \dots, N-1. \end{aligned} \tag{11.4}$$

then

$$\begin{aligned} \mathbb{E}[(H - x - \hat{G}_T(\nu))^2] =& E[\big(\big(V_N - \nu_{N-1}\Delta S_{N-1} + \lambda |\Delta \nu_{N-2}| S_{t_{N-1}} - V_{N-1}\big) + & (11.5) \\ & \sum_{i=2}^{N-1} \big(V_i + \lambda |\Delta \nu_{i-2}| S_{t_{i-1}} - \nu_{i-1}\Delta S_{i-1} - V_{i-1}\big) + & (11.6) \\ & (V_1 + \lambda |\nu_0| S_{t_0} - \nu_0 \Delta S_0 - x) \big)^2] & (11.7) \end{aligned}$$

Due to the definition (11.4), we have that

$$E[V_i + \lambda |\Delta \nu_{i-2}| S_{t_{i-1}} - \nu_{i-1}\Delta S_{i-1} - V_{i-1} | \mathcal{F}_{t_{i-1}}] = 0, \forall i = 1, \dots, N, \tag{11.8}$$

so that

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

and iterating the process gives

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

Then we can write the problem (11.3) as:

$$\begin{aligned} \hat{\nu} =& \arg \min_{\nu \in \hat{\Theta}^{(\bar{m},\bar{l})}} \mathbb{E}[\big(V_N - \nu_{N-1}\Delta S_{N-1} + \lambda |\Delta \nu_{N-2}| S_{t_{N-1}} - V_{N-1}\big)^2] + \\ & \sum_{i=2}^{N-1} \mathbb{E}[\big(V_i + \lambda |\Delta \nu_{i-2}| S_{t_{i-1}} - \nu_{i-1}\Delta S_{i-1} - V_{i-1}\big)^2] + \\ & \mathbb{E}[(V_1 + \lambda |\nu_0| S_{t_0} - \nu_0 \Delta S_0 - x)^2] \end{aligned} \tag{11.9}$$

We introduce the space

$$\rho_i^{\bar{m},\bar{l}}(\eta) = \{(V,\nu)/V,\nu \text{ are } \mathbb{R} \text{ valued } \mathcal{F}_{t_i}\text{-adapted with } -\bar{m} \leq \nu - \eta \leq \bar{l}\},$$

and the space

$$\begin{aligned}\hat{\rho}_i^{\bar{m},\bar{l}}(\eta) =& \{(V,\nu_i,\ldots,\nu_{N-1})/V \text{ is } \mathbb{R} \text{ valued, } \mathcal{F}_{t_i}\text{-adapted , the } \nu_j, j \geq i \text{ are } \mathbb{R} \text{ valued} \\ & \mathcal{F}_{t_j}\text{-adapted with } \bar{m} \leq \nu_i - \eta \leq \bar{l}, \bar{m} \leq \nu_{j+1} - \nu_j \leq \bar{l} \text{ for } i \leq j < N-1\},\end{aligned}$$

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 $\bar{R}$ is taken ω by ω and stores the optimal trading gain function on each trajectory. Then $\bar{R}$ satisfies at date t_i with an asset value S_{t_i} for an investment ν_{i-1} chosen at date t_{i-1}:

$$\begin{aligned}\bar{R}(t_i, S_{t_i}, \nu_{i-1}) =& H - \sum_{j=i}^{N-1} \nu_j \Delta S_j + \lambda \sum_{j=i}^{N-1} |\Delta\nu_{j-1}| S_{t_j}, \\ &= R(t_{i+1}, S_{t_{i+1}}, \nu_i) - \nu_i \Delta S_i + \lambda |\Delta\nu_{i-1}| S_{t_i},\end{aligned}$$

and at the date t_i according to equation (11.5) the optimal control is the control ν associated to the minimization problem:

$$\min_{(V,\nu)\in\rho_i^{\bar{m},\bar{l}}(\nu_{i-1})} \mathbb{E}[(\bar{R}(t_{i+1}, S_{t_{i+1}}, \nu) - \nu\Delta S_i + \lambda|\nu - \nu_{i-1}|S_{t_i} - V)^2 | \mathcal{F}_{t_i}]$$

This leads to the Algorithm 8.

Algorithm 8 Backward resolution for $\mathcal{L}^2$ minimization problem avoiding conditional expectation iteration.

$\bar{R}(t_N, S_{t_{N-1}}(\omega), \nu_{N-1}) = H(\omega), \quad \forall \nu_{N-1}$

for $i = N, 2$ **do**

$$\begin{aligned}(\tilde{V}(t_{i-1}, S_{t_{i-1}}, \nu_{i-2}), \nu_{i-1}) = \quad & \arg\min_{(V,\nu)\in\rho_{i-1}^{\bar{m},\bar{l}}(\nu_{i-2})} \mathbb{E}[(\bar{R}(t_i, S_{t_i}, \nu) - \\ & \nu\Delta S_{i-1} + \lambda|\nu - \nu_{i-2}|S_{t_{i-1}} - V)^2 | \mathcal{F}_{t_{i-1}}] \qquad (11.10)\end{aligned}$$

$$\bar{R}(t_{i-1}, S_{t_{i-1}}, \nu_{i-2}) = \bar{R}(t_i, S_{t_i}, \nu_{i-1}) - \nu_{i-1}\Delta S_{i-1} + \lambda|\Delta\nu_{i-2}|S_{t_{i-1}}$$

end for

$\nu_0 = \arg\min_{\nu\in[-\bar{m},\bar{l}]} \mathbb{E}[(\bar{R}(t_1, S_{t_1}, \nu) + \lambda|\nu|S_{t_0} - \nu\Delta S_0 - x)^2]$

Remark 25 *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 8 is replaced by*

$$(\tilde{V}, \nu_0) = \arg\min_{(V,\nu)} \mathbb{E}[(V(t_1, S_{t_1}, \nu) + \lambda|\nu|S_{t_0} - \nu\Delta S_0 - V)^2 + R(t_1, S_{t_1}, \nu)],$$

and last line of Algorithm 8 by

$$(\tilde{V}, \nu_0) = \arg\min_{(V,\nu)\in\mathbb{R}\times[-\bar{m},\bar{l}]} \mathbb{E}[(\bar{R}(t_1, S_{t_1}, \nu) + \lambda|\nu|S_{t_0} - \nu\Delta S_0 - V)^2].$$

Remark 26 *In the two algorithm presented an argmin has to be achieved: a discretization in ν_{i-2} has to be achieved on a grid $[\nu_{i-1} - m, \nu_{i-1} + l]$.*

11.3 Practical algorithm based on Algorithm 8

Starting from the theoretical Algorithm 8, we aim at getting an effective implementation based on a representation of the function $\tilde{V}$ depending on time, S_t and the position ν_t in the hedging assets.

- In order to represent the dependency in the hedging position we introduce a time dependent grid

  $$\mathcal{Q}_i := (\xi k)_{k=-(i+1)\lfloor \frac{\bar{m}}{\xi} \rfloor, \ldots, (i+1)\lfloor \frac{\bar{l}}{\xi} \rfloor}$$

  where ξ is the mesh size associated to the set of grids $(\mathcal{Q}_i)_{i=0,N}$ and, if possible, chosen such that $\frac{\bar{l}}{\xi} = \lfloor \frac{\bar{l}}{\xi} \rfloor$ and $\frac{\bar{m}}{\xi} = \lfloor \frac{\bar{m}}{\xi} \rfloor$.

- To represent the dependency in S_t we will use a Monte Carlo method using simulated path $\left((S_{t_i}^{(j)})_{i=0,\ldots,N} \right)_{j=1,\ldots,M}$ and calculate the arg min in equation (11.10) using a methodology close to the one described in [8]: suppose that we are given at each date t_i $(D_q^i)_{q=1,\ldots,Q}$ a partition of $[\min_{j=1,M} S_{t_i}^{(j)}, \max_{j=1,M} S_{t_i}^{(j)}]$ such that each cell contains the same number of samples. We use the Q cells $(D_q^i)_{q=1,\ldots,Q}$ to represent the dependency of $\tilde{V}$ and ν in the S_{t_i} 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 $\hat{V}^q(t_i, S, k) = a_i^q + b_i^q S$ 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 $(l_i^q(j))_{j=1,\frac{M}{Q}}$ 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 ν on a grid $\eta = ((k+r)\xi)_{r=-\lfloor \frac{\bar{m}}{\xi} \rfloor, \ldots, \lfloor \frac{\bar{l}}{\xi} \rfloor}$, and by testing the one giving a $\hat{V}^q$ value minimizing the $\mathcal{L}^2$ risk so solving equation (11.10).
The Algorithm 9 permits to find the optimal $\nu_i^{(j)}(k)$ command using Algorithm 8 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 27 *It is possible to use different discretization ξ to define the set η and the set $\mathcal{Q}_i$. Then an interpolation is needed to get the $\bar{R}$ 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 [45].*

Remark 28 *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 $\mathcal{Q}_i$ at each date t_i.*

Then the global discretized version of Algorithm 8 is given on Algorithm 10 where $H^{(j)}$ correspond to the j the Monte Carlo realization of the payoff.

Algorithm 9 Optimize minimal hedging position $(\hat{\nu}_{t_i}^{(l)}(k))_{l=1,\dots,M}$ at date t_{i-1}

procedure OPTIMALCONTROL($\bar{R}(t_{i+1},.,.), k, S_{t_i}, S_{t_{i+1}}$)
  for $q = 1, Q$ **do**
    $P = \infty$,
    for $k = -\lfloor \frac{\bar{m}}{\xi} \rfloor, \dots, \lfloor \frac{\bar{l}}{\xi} \rfloor$ **do**

$$(a_i^q, b_i^q) = \arg\min_{(a,b)\in\mathbb{R}^2} \sum_{j=1}^{\frac{M}{Q}} \big(\bar{R}(t_{i+1}, S_{t_{i+1}}^{l_i^q(j)}, (k+l)\xi) - (k+l)\xi \Delta S_i^{l_i^q(j)} + \lambda |l\xi| S_{t_i}^{l_i^q(j)} - (a + b S_{t_i}^{l_i^q(j)}) \big)^2$$

$$\tilde{P} = \sum_{j=1}^{\frac{M}{Q}} \big(\bar{R}(t_{i+1}, S_{t_{i+1}}^{l_i^q(j)}, (k+l)\xi) - (k+l)\xi \Delta S_i^{l_i^q(j)} + \lambda |l\xi| S_{t_i}^{l_i^q(j)} - (a_i^q + b_i^q S_{t_i}^{l_i^q(j)}) \big)^2$$

      if $\tilde{P} < P$ **then**
        $\nu^q = k\xi$, $P = \tilde{P}$
      end if
    end for
    for $j = 1, \frac{M}{Q}$ **do**
      $\hat{\nu}_i^{(l_i^q(j))}(k) = \nu^q$
    end for
  end forreturn $(\hat{\nu}_{t_i}^{(j)}(k))_{j=1,\dots,M}$
end procedure

Algorithm 10 Global backward resolution algorithm, optimal control and optimal variance calculation

for $\nu \in \mathcal{Q}_{N-1}$ **do**
  for $j \in [1, M]$ **do**
    $\bar{R}(t_N, S^{(j)}_{t_N}, \nu) = H^{(j)}$
  end for
end for
for $i = N, 2$ **do**
  for $k\xi \in \mathcal{Q}_{i-2}$ **do**
    $(\nu^{(j)}_{i-1}(k))_{j=1,M} = \text{OptimalControl}(\bar{R}(t_i, ., .), k, S_{t_{i-1}}, S_{t_i})$,
    for $j \in [1, M]$ **do**

$$\bar{R}(t_{i-1}, S^{(j)}_{t_{i-1}}, k\xi) = \bar{R}(t_i, S^{(j)}_{t_i}, \nu^{(j)}_{i-1}(k)) - \nu^{(j)}_{i-1}(k)\Delta S^{(j)}_{i-1} + \lambda|\nu^{(j)}_{i-1}(k) - k\xi|S^{(j)}_{t_{i-1}}$$

    end for
  end for
end for
$P = \infty$,
for $k = -\lfloor \frac{\bar{m}}{\xi} \rfloor, \ldots, \lfloor \frac{\bar{l}}{\xi} \rfloor$ **do**

$$\tilde{P} = \sum_{j=1}^{M} \big(\bar{R}(t_1, S^{(j)}_{t_1}, k\xi) - k\xi\Delta S^{(j)}_0 + \lambda|k|\xi S_0 - x)^2$$

  if $\tilde{P} < P$ **then**
    $\nu_0 = k\xi$, $P = \tilde{P}$
  end if
end for

$$Var = \frac{1}{M}\sum_{j=1}^{M} \big(\bar{R}(t_1, S^{(j)}_{t_1}, \nu_0) - \nu_0\Delta S^{(j)}_0 + \lambda|\nu_0|S_0 - x\big)^2$$

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 12

Theoretical background

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

12.1 Notation and regularity results

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

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

and $|w|_1 = |w|_0 + [w]_1$. $C_1(Q)$ will stand for the space of functions with a finite $|\quad|_1$ 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 $\hat{K}$:

$$\sup_a |g|_1 + |\sigma_a|_1 + |b_a|_1 + |f_a|_1 + |c_a|_1 \leq \hat{K} \tag{12.1}$$

A classical result [24] 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* (12.1)*, 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 Δx being given, thereafter $(i_1\Delta x, \ldots, i_d\Delta x)$ with $\bar{i} = (i_1, \ldots, i_d) \in \mathbf{Z}^d$ will correspond to the coordinates of a mesh $M_{\bar{i}}$ defining a hypercube in dimension d. For an interpolation grid $(\xi_i)_{i=0,\ldots,N} \in [-1,1]^N$, and for a mesh $\bar{i}$, the point $y_{\bar{i},\tilde{j}}$ 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 $(y_{\bar{i},\tilde{j}})_{\bar{i},\tilde{j}}$ the set of all the grids points on the whole domain.
We notice that for regular mesh with constant volume Δx^d, we have the following relation for all $x \in \mathbb{R}^d$:

$$\min_{\bar{i},\tilde{j}} |x - y_{\bar{i},\tilde{j}}| \leq \Delta x. \tag{12.2}$$

12.2 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.

$$\begin{aligned} v_h(t+h,x) &= \inf_{a\in A}\left[\sum_{i=1}^{q}\frac{1}{2q}(v_h(t,\phi^+_{a,h,i}(t,x))+v_h(t,\phi^-_{a,h,i}(t,x))) \right. \\ &\quad \left. +f_a(t,x)h+c_a(t,x)hv_h(t,x)\right] \\ &:= v_h(t,x)+\inf_{a\in A}L_{a,h}(v_h)(t,x) \end{aligned} \tag{12.3}$$

with

$$\begin{aligned} L_{a,h}(v_h)(t,x) &= \sum_{i=1}^{q}\frac{1}{2q}(v_h(t,\phi^+_{a,h,i}(t,x))+v_h(t,\phi^-_{a,h,i}(t,x))-2v_h(t,x)) \\ &\quad +hc_a(t,x)v_h(t,x)+hf_a(t,x) \\ \phi^+_{a,h,i}(t,x) &= x+b_a(t,x)h+(\sigma_a)_i(t,x)\sqrt{hq} \\ \phi^-_{a,h,i}(t,x) &= x+b_a(t,x)h-(\sigma_a)_i(t,x)\sqrt{hq} \end{aligned}$$

where $(\sigma_a)_i$ is the i-th column of σ_a. We note that it is also possible to choose other types of discretization in the same style as those defined in [34].
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:

$$v_h(t,x)=(1-\frac{t}{h})g(x)+\frac{t}{h}v_h(h,x),\forall t\in[0,h]. \tag{12.4}$$

We first recall the following result:

Proposition 2 *Under the condition on the coefficients given by equation* (12.1)*, the solution* v_h *of equations* (12.3) *and* (12.4) *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*

$$|v-v_h|_0\leq Ch^{\frac{1}{4}}. \tag{12.5}$$

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

$$|v_h|_0 \leq C, \tag{12.6}$$

$$|v_h(t,x)-v_h(t,y)| \leq C|x-y|,\forall(x,y)\in Q^2. \tag{12.7}$$

12.3 Space interpolation

The space resolution of equation (12.3) is a achieved on a grid. The ϕ^+ and ϕ^- have to be computed by the use of an interpolator I such that:

$$\begin{aligned} v_h(t,\phi^+_{a,h,i}(t,x)) &\simeq I(v_h(t,.))(\phi^+_{a,h,i}(t,x)), \\ v_h(t,\phi^-_{a,h,i}(t,x)) &\simeq I(v_h(t,.))(\phi^-_{a,h,i}(t,x)). \end{aligned}$$

In order to easily prove the convergence of the scheme to the viscosity solution of the problem, the monotony of the scheme is generally required leading to some linear interpolator slowly converging. An adaptation to high order interpolator where the function is smooth can be achieved using Legendre grids and Sparse grids with some truncation (see [47], [46]).

Chapter 13

C++ API

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

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

a first object `SemiLagrangEspCond` is available:

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

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

namespace StOpt
{

/// \class  SemiLagrangEspCond SemiLagrangEspCond.h
/// calculate semi Lagrangian operator for previously defined process.
class SemiLagrangEspCond
{
    ///\brief interpolator
    std::shared_ptr<InterpolatorSpectral> m_interpolator;

    /// \brief store extremal values for the grid (min, max coordinates in each dimension)
    std::vector <std::array< double, 2>  >  m_extremalValues;

    /// \brief Do we use modification of volatility  to stay in the domain
    bool m_bModifVol ;

public :

    /// \brief Constructor
    /// \param  p_interpolator    Interpolator storing the grid
    /// \param  p_extremalValues  Extremal values of the grid
```

```
40      /// \param  p_bModifVol        do we modify volatility to stay in the domain
41      SemiLagrangEspCond(const std::shared_ptr<InterpolatorSpectral> &p_interpolator, const
            std::vector <std::array< double, 2>  > &p_extremalValues, const bool &p_bModifVol);
42
43      /// \brief Calculate \f$ \frac{1}{2d} \sum_{i=1}^d \phi(x+ b dt + \sigma_i \sqrt{dt})+
             \phi(x+ b dt - \sigma_i \sqrt{dt} \f$
44      ///        where \f$ \sigma_i \f$ is column \f$ i\f$ of \f$ \sigma \f$
45      /// \param p_x                 beginning point
46      /// \param p_b                 trend
47      /// \param p_sig               volatility matrix
48      /// \param  p_dt               Time step size
49      /// \return  (the value calculated,true) if point inside the domain, otherwise (0.,
            false)
50      std::pair<double, bool>  oneStep(const Eigen::ArrayXd   &p_x, const Eigen::ArrayXd &p_b
            , const Eigen::ArrayXXd &p_sig, const double &p_dt) const;
51
52
53  };
54  }
55  #endif
```

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 [47]).

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_sig` the matrix volatility of the process,

such that the interpolation is achieved for a time step h at points $p\_x + p\_bh \pm p\_sig\sqrt{h}$. 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` object has to be constructed. This object permits to define the PDE to solve (with it optimization problem if any).

```
1  // Copyright (C) 2016 EDF
2  // All Rights Reserved
3  // This code is published under the GNU Lesser General Public License (GNU LGPL)
4  #ifndef OPTIMIZERSLBASE_H
5  #define OPTIMIZERSLBASE_H
6  #include <vector>
7  #include <Eigen/Dense>
```

```cpp
#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,
```

```
69                                const Eigen::ArrayXd &p_phiInPt,
70                                Eigen::Ref<Eigen::ArrayXd> p_phiInOut) const  = 0 ;
71
72
73      /// \brief defines a step in simulation using the control calculated in optimization
74      /// \param p_gridNext      grid at the next step
75      /// \param p_controlInterp the optimal controls interpolator
76      /// \param p_state         state array (can be modified)
77      /// \param p_iReg          regime number
78      /// \param p_gaussian      unitary Gaussian realization
79      /// \param p_phiInOut      defines the value functions (modified) to follow
80      virtual void stepSimulateControl(const SpaceGrid    &p_gridNext,
81                                       const  std::vector< std::shared_ptr<
                                             InterpolatorSpectral> >   &p_controlInterp,
82                                       Eigen::Ref<Eigen::ArrayXd>  p_state,   int &p_iReg,
83                                       const Eigen::ArrayXd &p_gaussian,
84                                       Eigen::Ref<Eigen::ArrayXd> p_phiInOut) const  = 0 ;
85
86      /// \brief get number of regimes
87      virtual   int getNbRegime() const = 0 ;
88
89      /// \brief get back the dimension of the control
90      virtual int getNbControl() const = 0 ;
91
92      /// \brief do we modify the volatility to stay in the domain
93      virtual  bool getBModifVol() const = 0 ;
94
95      /// \brief get the number of Brownians involved in semi Lagrangian for simulation
96      virtual int getBrownianNumber() const = 0 ;
97
98      /// \brief get size of the  function to follow in simulation
99      virtual int getSimuFuncSize() const = 0;
100
101     /// \brief Permit to deal with some boundary points that do not need boundary
            conditions
102     ///        Return false if all points on the boundary need some boundary conditions
103     /// \param  p_point  potentially on the  boundary
104     virtual bool isNotNeedingBC(const Eigen::ArrayXd &p_point)  const = 0;
105 };
106 }
107 #endif /* OPTIMIZERSLBASE_H */
```

The main methods associated to this object are:

- `stepOptimize` is use to calculate the solution of the PDE at one point.
  - It takes a point of the grid used `p_point`,
  - and apply the semi-Lagrangian scheme `p_semiLag` at this point,
  - at a date given by `p_time`.

  It returns a pair containing:
  - the function value calculated at `p_point` for each regime,
  - the optimal control calculated at `p_point` for each control.
- `stepSimulate` is used when the PDE is associated to an optimization problem and we want to simulate an optimal policy using the function values calculated in the optimization part. The arguments are:
  - `p_gridNext` defining the grid used at the following time step,

  - `p_semiLag` the semi-Lagrangian operator constructed with an interpolator using the following time solution,
  - `p_state` the vector defining the current state for the current regime,
  - `p_iReg` the current regime number,
  - `p_gaussian` is the vector of gaussian random variables used to calculate the Brownian involved in the underlying process for the current simulation,
  - `p_phiInP` at the value of the function calculated in optimization at next time step for the given point,
  - `p_phiInOut` storing the cost functions: the size of the array is the number of functions to follow in simulation.

- `stepSimulateControl` is used when the PDE is associated to an optimization problem and we want to simulate an optimal policy using the optimal controls calculated in the optimization part. The arguments are:
  - `p_gridNext` defining the grid used at the following time step,
  - `p_controlInterp` a vector (for each control) of interpolators in controls
  - `p_state` the vector defining the current state for the current regime,
  - `p_iReg` the current regime number,
  - `p_gaussian` is the vector of gaussian random variables used to calculate the Brownian involved in the underlying process for the current simulation.
  - `p_phiInOut` storing the cost functions: the size of the array is the number of functions to follow in simulation.

  On return the `p_state` vector is modified, the `p_iReg` is modified and the cost function `p_phiInOut` is modified for the current trajectory.

- the `getCone` method is only relevant if the 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 $H1$. 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 $H1$. If no optimization is achieved, it returns the hyper cube $H2$ containing the points reached by the semi-Lagrangian scheme. For explanation of the parallel formalism see chapter 9.

- 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 [47]) is given below:

```
#include <iostream>
#include "StOpt/core/utils/constant.h"
#include "test/c++/tools/semilagrangien/OptimizeSLCase3.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::getCone(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);
    xReached[0][1] = p_xInit[0][1]  + m_alphaMax * sqrt(m_dt) ;
    return xReached;
}

pair< ArrayXd, ArrayXd> OptimizerSLCase3::stepOptimize(const ArrayXd   &p_point,
        const vector< shared_ptr<SemiLagrangEspCond> > &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::infty;
    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 std::vector< shared_ptr< StOpt::SemiLagrangEspCond > > &p_semiLag,
                                    Eigen::Ref<Eigen::ArrayXd>  p_state,  int &,
                                    const Eigen::ArrayXd &p_gaussian, const Eigen::ArrayXd &,
                                    Eigen::Ref<Eigen::ArrayXd>) const
{
    double vMin = StOpt::infty;
    double alphaOpt = -1;
    ArrayXd b(1);
```

```
58      ArrayXXd sig(1, 1) ;
59      ArrayXd proba = p_state ;
60      // recalculate the optimal alpha
61      for (int iAl = 0; iAl < m_alphaMax / m_stepAlpha; ++iAl)
62      {
63          double alpha = iAl * m_stepAlpha;
64          b(0) = -alpha * m_mu / m_sig;// trend
65          sig(0) = alpha;// volatility with one Brownian
66          pair<double, bool> lagrang = p_semiLag[0]->oneStep(proba, b, sig, m_dt);// test the
                control
67          if (lagrang.second)
68          {
69              if (lagrang.first < vMin)
70              {
71                  vMin = lagrang.first;
72                  alphaOpt = alpha;
73              }
74          }
75      }
76      proba(0) +=  alphaOpt * p_gaussian(0) * sqrt(m_dt);
77      // truncate if necessary
78      p_gridNext.truncatePoint(proba);
79      p_state = proba ;
80
81  }
82
83
84  void OptimizerSLCase3:: stepSimulateControl(const SpaceGrid    &p_gridNext,
85          const  vector< shared_ptr< InterpolatorSpectral> >   &p_controlInterp,
86          Eigen::Ref<Eigen::ArrayXd>  p_state,   int &,
87          const ArrayXd &p_gaussian,
88          Eigen::Ref<Eigen::ArrayXd>) const
89  {
90      ArrayXd proba = p_state ;
91      double alphaOpt = p_controlInterp[0]->apply(p_state);
92      proba(0) +=  alphaOpt * p_gaussian(0) * sqrt(m_dt);
93      // truncate if necessary
94      p_gridNext.truncatePoint(proba);
95      p_state = proba ;
96  }
```

13.1 PDE resolution

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

```
1  // Copyright (C) 2016 EDF
2  // All Rights Reserved
3  // This code is published under the GNU Lesser General Public License (GNU LGPL)
4  #ifndef TRANSITIONSTEPSEMILAGRANG_H
5  #define TRANSITIONSTEPSEMILAGRANG_H
6  #ifdef OMP
7  #include <omp.h>
8  #endif
9  #include <functional>
10 #include <memory>
11 #include <Eigen/Dense>
12 #include "geners/BinaryFileArchive.hh"
13 #include "StOpt/semilagrangien/TransitionStepSemilagrangBase.h"
14 #include "StOpt/core/grids/SpaceGrid.h"
15 #include "StOpt/core/grids/InterpolatorSpectral.h"
16 #include "StOpt/semilagrangien/OptimizerSLBase.h"
17
```

```cpp
/** \file TransitionStepSemilagrang.h
 * \brief Solve one step  of explicit semi Lagrangian scheme
 * \author Xavier Warin
 */


namespace StOpt
{

/// \class TransitionStepSemilagrang TransitionStepSemilagrang.h
///        One step  of semi Lagrangian scheme
class TransitionStepSemilagrang : public TransitionStepSemilagrangBase
{
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) const;

    /// \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) const;
};
}
#endif /* TRANSITIONSTEPSEMILAGRANG_H */
```

It constructor takes the following arguments:

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

A first method `oneStep` take the following arguments:

Table 13.1: Which `TransitionStepSemilagrang` object to use depending on the grid used and the type of parallelization used.

	Full grid	Sparse grid
Sequential	`TransitionStepSemilagrang`	`TransitionStepSemilagrang`
Parallelization on calculations threads and MPI	`TransitionStepSemilagrang`	`TransitionStepSemilagrang`
Distribution of calculations and data (MPI)	`TransitionStepSemilagrangDist`	Not available

- `p_phiIn` describes for each regime the solution previously calculated on the grid at the previous time,
- `p_time` is the current time step,
- `p_boundaryFunc` is a function giving the Dirichlet solution of the problem depending on the number of regimes and the position on the boundary.

It gives back an estimation of the solution at the current date on the current grid for all the regimes and an estimation of the optimal control calculated for all the controls.
A last method `dumpValues` method permits to dump the solution calculated `p_phiIn` at the step `p_istep+1` and the optimal control at step `p_istep` in an archive `p_ar`.
A version using the distribution of the data and calculations can be found in the `TransitionStepSemilagrangDist` object. An example of a time recursion in sequential can be found in the `semiLagrangianTime` function and an example with distribution can be found in the `semiLagrangianTimeDist` function. In both functions developed in the test chapter the analytic solution of the problem is known and compared to the numerical estimation obtained with the semi-Lagrangian method.

13.2 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

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

  where

  - `p_ar` is the binary archive where the continuation values are stored,

- `p_iStep` is the number associated to the current time step (0 at the beginning date of simulation, the number is increased by one at each time step simulated),
- `p_name` is the base name to search in the archive,
- `p_GridNext` is the grid at the next time step (`p_iStep+1`),
- `p_Optimize` is the Optimizer describing the transition from one time step to the following one,
- `p_OneFile` equals `true` if a single archive is used to store continuation values.

Remark 29 *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`

```
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_statevector` store the continuous state (continuous state size by number of simulations)
- `p_iReg` for each simulation give the current regime number,
- `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.

Remark 30 *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:

```
// Copyright (C) 2016 EDF
// All Rights Reserved
// This code is published under the GNU Lesser General Public License (GNU LGPL)
#ifdef USE_MPI
#include <boost/random.hpp>
#include <memory>
#include <Eigen/Dense>
#include "geners/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;
    // 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::SimulateStepSemilagrangDist(ar, p_nbStep - 1 - istep, nameAr, p_grid, p_optimize, p_bOneFile).oneStep(gaussian, states, regime, costFunction);
    }
    // final cost to add
    for (int is = 0; is < p_nbSimul; ++is)
        costFunction(0, is) += p_funcFinalValue(regime(is), states.col(is));
    // average gain/cost
    return costFunction.mean();
}
#endif
```

A sequential or parallelized on calculations version of the previous example is given in the `semiLagrangianSimuDist.cpp` 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

```
    SimulateStepSemilagrangControlDist(gs::BinaryFileArchive &p_ar,  const int &p_iStep,  const std::string &p_name ,
                                       const std::shared_ptr<FullGrid>   &p_gridCur,
                                       const std::shared_ptr<FullGrid> &p_gridNext,
                                       const std::shared_ptr<OptimizerSLBase > &p_pOptimize ,
```

```
5                                          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 `true` if a single archive is used to store continuation values.

Remark 31 *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`

```
1 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_statevector` stores the continuous state (continuous state size by number of simulations)
- `p_iReg` for each simulation gives the current regime number,
- `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:

```
1 // Copyright (C) 2016 EDF
2 // All Rights Reserved
3 // This code is published under the GNU Lesser General Public License (GNU LGPL)
4 #ifdef USE_MPI
5 #include <memory>
6 #include <boost/random.hpp>
7 #include <Eigen/Dense>
8 #include "geners/BinaryFileArchive.hh"
9 #include "StOpt/semilagrangien/OptimizerSLBase.h"
```

```cpp
#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 the `semiLagrangianSimuControl.cpp` file.

Remark 32 *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 13.2: Which simulation object to use depending on the TransitionStepSemilagrang object used.

	`TransitionStepSemilagrang`	`TransitionStepSemilagrangDist` `bOneFile = true`	`TransitionStepSemilagrangDist` `bOneFile = false`
`SimulateStepSemilagrang`	Yes	Yes	No
`SimulateStepSemilagrangControl`	Yes	Yes	No
`SimulateStepSemilagrangDist`	No	Yes	Yes
`SimulateStepSemilagrangControlDist`	No	Yes	Yes

Part V

An example with both dynamic programming with regression and PDE

In this chapter we give an example where 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 λ 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.

$$\tilde{c}(l, L) = \bar{\beta}(c_\infty + (c_0 - c_\infty)e^{\beta L})(1 + l)l$$

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

$$\begin{cases} \partial_t v + \alpha(m - D)\partial_D v + \frac{1}{2}\sigma^2 \partial^2_{DD} v + (D - L)^+ \partial_Q v + \Pi(D, L) \\ +sL^{1-\alpha} - y(D - L)^+ + \sup_l \{l\partial_L v - \tilde{c}(l, L)\} = 0 \\ v_T = 0 \end{cases} \tag{13.1}$$

and the carbon price $y(t, D_t, Q_t, L_t)$ is given by:

$$\begin{cases} \partial_t y + \alpha(m - D)\partial_D y + \frac{1}{2}\sigma^2 \partial^2_{DD} y + (D - L)^+ \partial_Q y + l^* \partial_L y = 0 \\ y_T = \lambda 1_{Q_T \geq K} \end{cases} \tag{13.2}$$

and l^* is the optimal control in equation (13.1). The previous equation can be solved with the semi-Lagrangian method.
After a time discretization with a step δt a dynamic programming equation can be given by

$$\begin{aligned} v(T - \delta t, D, Q, L) &= \sup_l (\Pi(D, L) + sL^{1-\alpha} - y_{T-\delta t}(D - L)^+ - \tilde{c}(l, L))\delta t + \\ &\quad \mathbb{E}_{T-\delta t}(V(T, D_T^{T-\delta t, D}, Q + (D - L)^+ \delta t, L + l\delta t)) \qquad (13.3) \\ Y(T - \delta t, D, Q, L) &= \mathbb{E}_{T-\delta t}(Y(T, D_T^{T-\delta t, D}, Q + (D - L)^+ \delta t, L + l^* \delta t)) \qquad (13.4) \end{aligned}$$

The previous equations (13.3) and (13.4) 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 `lMax` and a discretization step to test the controls `lStep`.

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

13.3 The dynamic programming with regression approach

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

```
1  // Copyright (C) 2016 EDF
2  // All Rights Reserved
3  // This code is published under the GNU Lesser General Public License (GNU LGPL)
4  #include "StOpt/core/utils/constant.h"
5  #include "OptimizeDPEmissive.h"
6
7  using namespace std ;
8  using namespace StOpt;
9  using namespace Eigen;
10
11
12 // constructor
13 OptimizeDPEmissive::OptimizeDPEmissive(const double &p_alpha,
14                                        const std::function<double(double, double)> &p_PI,
15                                        const std::function< double(double, double) >     &
                                              p_cBar,  const double   &p_s, const double &
                                              p_lambda,
```

```
16                                          const double &p_dt,
17                                          const double   &p_maturity,
18                                          const  double &p_lMax, const double &p_lStep, const
                                                 std::vector <std::array< double, 2>  >   &
                                                p_extrem):
19      m_alpha(p_alpha), m_PI(p_PI),
20      m_cBar(p_cBar), m_s(p_s), m_lambda(p_lambda), m_dt(p_dt), m_maturity(p_maturity),
            m_lMax(p_lMax), m_lStep(p_lStep),
21      m_extrem(p_extrem)
22  {}
23
24  Array< bool, Dynamic, 1> OptimizeDPEmissive::getDimensionToSplit() const
25  {
26      Array< bool, Dynamic, 1> bDim = Array< bool, Dynamic, 1>::Constant(2, true);
27      return  bDim ;
28  }
29
30  // for parallelism
31  std::vector< std::array< double, 2> > OptimizeDPEmissive::getCone(const  vector<  std::
        array< double, 2>  > &p_xInit) const
32  {
33      vector< array< double, 2> > xReached(2);
34      xReached[0][0] = p_xInit[0][0]  ; // Q only increases
35      xReached[0][1] = m_extrem[0][1]  ; // whole domain due to demand which is unbounded
36      xReached[1][0] = p_xInit[1][0]  ;  // L only increases
37      xReached[1][1] = p_xInit[1][1] + m_lMax * m_dt  ; // maximal increase given by the
            control
38      return xReached;
39  }
40
41  // one step in optimization from stock point for all simulations
42  std::pair< ArrayXXd, ArrayXXd> OptimizeDPEmissive::stepOptimize(const   std::shared_ptr<
        StOpt::SpaceGrid> &p_grid, const ArrayXd   &p_stock,
43          const  std::vector< ContinuationValue> &p_condEsp,
44          const std::vector < std::shared_ptr< ArrayXXd > > &) const
45  {
46      std::pair< ArrayXXd, ArrayXXd> solutionAndControl;
47      // to store final solution (here two regimes)
48      solutionAndControl.first = ArrayXXd::Constant(m_simulator->getNbSimul(), 2, -StOpt::
            infty);
49      solutionAndControl.second =  ArrayXXd::Constant(m_simulator->getNbSimul(), 1, -StOpt::
            infty);
50      // demand
51      ArrayXd demand = m_simulator->getParticles().array().row(0).transpose();
52      // Gain (size number of simulations)
53      ArrayXd gain(m_simulator->getNbSimul());
54      double gainSubvention =  m_s * pow(p_stock(1), 1. - m_alpha); // subvention for non
            emissive energy
55      for (int is = 0 ; is < m_simulator->getNbSimul(); ++is)
56          gain(is) = m_PI(demand(is), p_stock(1)) + gainSubvention ; // gain by production
                and subvention
57      ArrayXd ptStockNext(2);
58      // time to maturity
59      double timeToMat = m_maturity - m_simulator->getCurrentStep();
60      // interpolator at the new step
61      for (int is = 0 ; is < m_simulator->getNbSimul(); ++is)
62      {
63          for (int  iAl = 0; iAl < m_lMax / m_lStep ; ++iAl) // test all command for
                investment between 0 and lMax
64          {
65              double  l = iAl * m_lStep;
66              // interpolator at the new step
67              ptStockNext(0) =  p_stock(0) + std::max(demand(is) - p_stock(1), 0.) * m_dt;
68              ptStockNext(1) =  p_stock(1) +  l * m_dt ;
69              // first test we are inside the domain
70              if (p_grid->isInside(ptStockNext))
71              {
72                  // create an interpolator at the arrival point
73                  std::shared_ptr<StOpt::Interpolator>  interpolator = p_grid->
```

```
                    createInterpolator(ptStockNext);
74                  // calculate Y for this simulation with the optimal control
75                  double yLoc = p_condEsp[1].getASimulation(is, *interpolator);
76                  // local gain
77                  double gainLoc = (gain(is) - yLoc * std::max(demand(is) - p_stock(1), 0.) -
                         m_cBar(l, p_stock(1))) * m_dt;
78                  //  gain + conditional expectation of future gains
79                  double condExp = gainLoc + p_condEsp[0].getASimulation(is, *interpolator);
80                  if (condExp >  solutionAndControl.first(is, 0)) // test optimality of the
                        control
81                  {
82                      solutionAndControl.first(is, 0) = condExp;
83                      solutionAndControl.first(is, 1) = yLoc;
84                      solutionAndControl.second(is, 0) = l;
85                  }
86              }
87          }
88          // test if solution acceptable
89          if (StOpt::almostEqual(solutionAndControl.first(is, 0), - StOpt::infty, 10))
90          {
91              // fix boundary condition
92              solutionAndControl.first(is, 0) =  timeToMat * (m_PI(demand(is), p_stock(1)) +
                    m_s * pow(p_stock(1), 1. - m_alpha) - m_lambda * std::max(demand(is) -
                    p_stock(1), 0.));
93              solutionAndControl.first(is, 1) = m_lambda ; // Q est maximal !!
94              solutionAndControl.second(is, 0) = 0. ; // fix control to zero
95          }
96      }
97      return solutionAndControl;
98  }
99
100 // one step in simulation for current simulation
101 void OptimizeDPEmissive::stepSimulate(const std::shared_ptr< StOpt::SpaceGrid>   &p_grid,
        const std::vector< StOpt::GridAndRegressedValue > &p_continuation,
102                                       StOpt::StateWithStocks &p_state,
103                                       Ref<ArrayXd> p_phiInOut) const
104 {
105     ArrayXd ptStock = p_state.getPtStock();
106     ArrayXd ptStockNext(ptStock.size());
107     double vOpt = - StOpt::infty;
108     double gainOpt = 0.;
109     double lOpt = 0. ;
110     double demand = p_state.getStochasticRealization()(0); // demand for this simulation
111     ptStockNext(0) =  ptStock(0) + std::max(demand - ptStock(1), 0.) * m_dt;
112     double   gain  = m_PI(demand, ptStock(1)) +  m_s * pow(ptStock(1), 1. - m_alpha) ; //
            gain from production and subvention
113     double yOpt = 0. ;
114     for (int  iAl = 0; iAl < m_lMax / m_lStep ; ++iAl) // test all command for investment
            between 0 and lMax
115     {
116         double  l = iAl * m_lStep;
117         // interpolator at the new step
118         ptStockNext(1) =  ptStock(1) +  l * m_dt ;
119         // first test we are inside the domain
120         if (p_grid->isInside(ptStockNext))
121         {
122             // calculate Y for this simulation with the control
123             double yLoc = p_continuation[1].getValue(ptStockNext, p_state.
                    getStochasticRealization());
124             // local gain
125             double gainLoc = (gain - yLoc * std::max(demand - ptStock(1), 0.) - m_cBar(l,
                    ptStock(1))) * m_dt;
126             //  gain + conditional expectation of future gains
127             double condExp = gainLoc +  p_continuation[0].getValue(ptStockNext, p_state.
                    getStochasticRealization());
128
129             if (condExp >  vOpt) // test optimality of the control
130             {
131                 vOpt = condExp;
```

```
                gainOpt = gainLoc;
                lOpt = l;
                yOpt = yLoc;
            }
        }
    }
    p_phiInOut(0) += gainOpt; // follow v value
    p_phiInOut(1) = yOpt ; // follow y value
    ptStockNext(1) =  ptStock(1) +  lOpt * m_dt ; // update state due to control
    p_state.setPtStock(ptStockNext);
}
```

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

13.4 The PDE approach

We can do the same with the PDE approach using a simulator for the OU demand (`AR1Simulator`). We then define an optimizer (`OptimizeSLEmissive`) and the methods used to optimize and simulate, and the `getCone` method for parallelization:

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

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

// constructor
OptimizeSLEmissive::OptimizeSLEmissive(const double &p_alpha,  const double &p_m, const
    double &p_sig, const std::function<double(double, double)> &p_PI,
                                       const std::function< double(double, double) >     &
                                           p_cBar,  const double   &p_s, const double &p_dt
                                           ,
                                       const  double &p_lMax, const double &p_lStep, const
                                            std::vector <std::array< double, 2>  >   &
                                           p_extrem):
    m_alpha(p_alpha), m_m(p_m), m_sig(p_sig), m_PI(p_PI), m_cBar(p_cBar), m_s(p_s), m_dt(
        p_dt),
    m_lMax(p_lMax), m_lStep(p_lStep), m_extrem(p_extrem) {}

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


// for parallelism
vector< array< double, 2> >  OptimizeSLEmissive::getCone(const  vector<  array< double, 2>
     > &p_xInit) const
{
    vector< array< double, 2> > xReached(3);
    xReached[0][0] = p_xInit[0][0]  +   m_alpha * (m_m - m_extrem[0][1]) * m_dt -  m_sig *
        sqrt(m_dt); // demand "cone" driven by maximal value allowed for demand
    xReached[0][1] = p_xInit[0][1]  +  m_alpha * m_m * m_dt +   m_sig * sqrt(m_dt) ; // low
         value for demand is taken equal to 0
    xReached[1][0] = p_xInit[1][0]  ;// Q only increases
    xReached[1][1] = p_xInit[1][1]  + m_extrem[0][1] * m_dt  ; // Q increase bounded by
        maximal demand
    xReached[2][0] = p_xInit[2][0]  ; // L only increases
```

```
32      xReached[2][1] = p_xInit[2][1] + m_lMax * m_dt  ;// maximal increase given by the
            control
33      return xReached;
34  }
35
36
37  // one step in optimization from current point
38  std::pair< ArrayXd, ArrayXd> OptimizeSLEmissive::stepOptimize(const ArrayXd   &p_point,
39          const vector< shared_ptr<SemiLagrangEspCond> > &p_semiLag,
40          const double &, const ArrayXd &) const
41  {
42      pair< ArrayXd, ArrayXd> solutionAndControl;
43      solutionAndControl.first.resize(2);
44      solutionAndControl.second.resize(1);
45      ArrayXXd sig = ArrayXXd::Zero(3, 1) ;
46      sig(0, 0) = m_sig;
47      double vOpt = - StOpt::infty;
48      double yOpt = 0. ;
49      double lOpt = 0 ;
50      ArrayXd b(3);
51      b(0) =  m_alpha * (m_m - p_point(0)) ; // trend
52      b(1) =  max(p_point(0) - p_point(2), 0.);
53      // gain already possible to calculate (production and subvention)
54      double gainFirst = m_PI(p_point(0), p_point(2)) + m_s * pow(p_point(2), 1. - m_alpha)
            ;
55      for (int iAl = 0; iAl < m_lMax / m_lStep ; ++iAl) // test all command for investment
            between 0 and lMax
56      {
57          double  l = iAl * m_lStep;
58          b(2) =  l ;
59          pair<double, bool> lagrangY = p_semiLag[1]->oneStep(p_point, b, sig, m_dt); // for
                the control calculate y
60          if (lagrangY.second) // is the control admissible
61          {
62              pair<double, bool> lagrang = p_semiLag[0]->oneStep(p_point, b, sig, m_dt);  //
                    one step for v
63              // gain function
64              double gain = m_dt * (gainFirst - lagrangY.first * b(1)  - m_cBar(l, p_point(2)
                    ));
65              double arbitrage = gain + lagrang.first;
66              if (arbitrage > vOpt) // optimality  of the control
67              {
68                  vOpt = arbitrage; // upgrade solution v
69                  yOpt =  lagrangY.first; // store y
70                  lOpt = l; // upgrade optimal control
71              }
72          }
73      }
74
75      if (StOpt::almostEqual(vOpt, - StOpt::infty, 10))
76      {
77          std::cout << " Reduce time step " << std::endl ;
78          abort();
79      }
80      solutionAndControl.first(0) =  vOpt; // send back v function
81      solutionAndControl.first(1) =  yOpt; // send back y function
82      solutionAndControl.second(0) =  lOpt; // send back optimal control
83      return solutionAndControl;
84  }
85
86  // one step in simulation for current simulation
87  void OptimizeSLEmissive::stepSimulate(const SpaceGrid   &p_gridNext,
88                                        const  std::vector< std::shared_ptr< StOpt::
                                              SemiLagrangEspCond> > &p_semiLag,
89                                        Ref<ArrayXd>  p_state,   int &,
90                                        const ArrayXd &p_gaussian,
91                                        const ArrayXd &,
92                                        Ref<ArrayXd> p_phiInOut) const
93  {
```

```
94      ArrayXd state = p_state;
95      ArrayXXd sig = ArrayXXd::Zero(3, 1) ; // diffusion matrix for semi Lagrangian
96      sig(0, 0) = m_sig;
97      double vOpt = - StOpt::infty;
98      double lOpt = 0 ;
99      double yOpt = 0;
100     ArrayXd b(3);
101     b(0) =  m_alpha * (m_m - p_state(0)) ; // trend for D (independent of control)
102     b(1) =  max(p_state(0) - p_state(2), 0.); // trend for Q (independent of control)
103     double gainFirst = m_PI(p_state(0), p_state(2)) + m_s * pow(p_state(2), 1. - m_alpha)
            ; // gain for production and subvention
104     for (int iAl = 0; iAl < m_lMax / m_lStep ; ++iAl) // recalculate the optimal control
105     {
106         double  l = iAl * m_lStep;
107         b(2) =  l ;
108         pair<double, bool> lagrangY = p_semiLag[1]->oneStep(p_state, b, sig, m_dt); //
                calculate y for this control
109         if (lagrangY.second)
110         {
111             pair<double, bool> lagrang = p_semiLag[0]->oneStep(p_state, b, sig, m_dt); //
                    calculate the function value v
112             // gain function
113             double gain = m_dt * (gainFirst - lagrangY.first * b(1)  - m_cBar(l, p_state(2)
                    ));
114             double arbitrage = gain + lagrang.first;
115             if (arbitrage > vOpt) // arbitrage
116             {
117                 vOpt = arbitrage; // upgrade solution
118                 yOpt =  lagrangY.first; // upgrade y value
119                 lOpt = l; // upgrade optimal control
120             }
121         }
122     }
123     // gain function
124     p_phiInOut(0) += m_dt * (gainFirst - yOpt * b(1) - m_cBar(lOpt, state(2))); // store v
            value
125     p_phiInOut(1) = yOpt; // store y value
126     // update state
127     state(0) += m_alpha * (m_m - p_state(0)) * m_dt + m_sig * p_gaussian(0) * sqrt(m_dt);
            // demand (no control)
128     state(1) += b(1) * m_dt; //Q
129     state(2) += lOpt * m_dt; //L
130     // truncate if necessary to stay inside domain.
131     p_gridNext.truncatePoint(state);
132     p_state = state ;
133 }
```

The three dimensional grids used can be some full grids or some sparse grids. Both versions of the resolution can be found in a test case (`testSLNonEmissive.cpp`).

Part VI

Stochastic Dual Dynamic Programming

Chapter 14

SDDP algorithm

14.1 Some general points about SDDP

Stochastic Dual Dynamic Programming (SDDP) is an approximate dynamic programming algorithm developed by Pereira and Pinto in 1991 [36].

To describe how SDDP works, we will consider a class of linear programs that have $T+1$ stages denoted $\{0, 1, \ldots, t, \ldots, T\}$. 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 is the decision variable at time t. If the data process is stagewise independent, x_t also denotes the state at time $t+1$.
- $\omega_t \in \Omega_t$ is the random data process at time t, where Ω_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 ω_t.
- $\mathscr{Q}_t(x_{t-1}) = \mathbb{E}[Q_t(x_{t-1}, \omega_t)]$.

Decision process

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

$x_0 \rightarrow$ observation of $\omega_1 \rightarrow$ decision of $x_1 \ldots$
$\rightarrow$ decision of $x_{T-1} \rightarrow$ observation of $\omega_T \rightarrow$ decision of x_T

A rigorous formulation of the multistage stochastic linear program to solve is the following:

$$V^* = \min_{\substack{A_0 x_0 = \omega_0 \\ x_0 \geq 0}} c_0^\top x_0 + \mathbb{E}\left[\min_{\substack{E_1 x_0 + A_1 x_1 = \omega_1 \\ x_1 \geq 0}} c_1^\top x_1 + \mathbb{E}\left[\cdots + \mathbb{E}\left[\min_{\substack{E_T x_{T-1} + A_T x_T = \omega_T \\ x_T \geq 0}} c_T^\top x_T\right]\right]\right] \quad (14.1)$$

The deterministic equivalent of this problem (14.1) is achieved by discretizing ω_t (or by using directly ω_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 $(\Omega_t)_{t\in\{0,1,\ldots,T\}}$ are of reasonable size.

Dynamic programming principle

Dynamic programming involves splitting up the problem (14.1) in a series of sub-problem bounded together by a state variable. The aim is to compute backwards the functions Q_t and $\mathscr{Q}_t$. They fulfill the following equations:

$$[LP_t] \begin{cases} & Q_t(x_{t-1}, \omega_t) = \min c_t^\top x_t + \mathscr{Q}_{t+1}(x_t) \\ s.c. & \quad A_t x_t = \omega_t - E_t x_{t-1}, \quad [\pi_t(\omega_t)] \\ & \quad x_t \geqslant 0 \end{cases} \quad (14.2)$$

$$\mathscr{Q}_t(x_{t-1}) = \mathbb{E}[Q_t(x_{t-1}, \omega_t)] \quad (14.3)$$

The function $Q(x_{t-1}, \omega_t)$ stands for the expected value of a future cost knowing the state x_{t-1} and the random data ω_t. $\mathscr{Q}_t(x_{t-1})$ is the expected value of the future cost knowing the state x_{t-1}. The dynamic programming principle ensures that $V^* = \mathscr{Q}_1(x_0)$.

Given $\mathscr{Q}_T(\cdot)$, the successive computations are achieved backwards switching between the resolution of the linear sub-problem (14.2) and the computation of (14.3).

The implementation of dynamic programming involves approximating successively the two value functions with equations (14.2 - 14.3) 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 $\mathscr{Q}_{t+1}(x_t)$ 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 [36]. 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 $\mathscr{Q}_t(x_{t-1})$. Indeed in the frame of a linear problem with complete recourse 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*.

14.2 A method, different algorithms

The method implemented in this library is based on the different situations shown in a technical report of PSR program [35] 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 algorithms of SDDP.

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

14.2.1 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 ω_t.
- At each stage, the linear sub-problem solution space is non-empty and bounded.

In this case the functions $\mathscr{Q}_t(\cdot)$ 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:

- $\{0, 1, \ldots, T\}$, the set of stages, where T is the time horizon.
- $n = 0$, is the counter of the number of iterations (*backward-forward*). n is incremented at the end of each iteration.
- $p \in \mathbb{R}$, the precision to reach for the convergence test.
- $n_{step} \in \mathbb{N}$, the number of iterations achieved between 2 convergence tests.
- $n_{iterMax} \in \mathbb{N}$, the maximal number of iterations.
- $x_0 \in \mathbb{R}^n_+$, the initial vector state.
- $L \in \mathbb{N}$, the number of scenarios used in the backward pass.
- $G \in \mathbb{N}$, the number of scenarios used in the forward pass. It gives also the number of new cuts computed at every iteration (*backward-forward*) and the number of states near which the Benders cuts are computed.

Forward pass

The aim of this pass is to explore new feasible vector state and to get an estimation of the upper bound of the optimal cost. To this end the current strategy is simulated for a set of G scenarios. The set of scenarios could be historical chronicles or random draws. Algorithm 11 presents the forward pass at the n-th iteration of the SDDP method.

Backward pass

The aim of the backward pass is to add, at each stage, a set of new Benders cuts 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 $(x_t^g)_{g=1,\ldots,G}$ obtained during the forwards pass. Algorithm 12 presents the backward 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 $n_{iterMax}$ (an iteration is made of the succession of *backward-forward* passes) which shall not be exceeded. The second one is a test of convergence towards each other between the forward and the backward cost. The convergence test uses the following indicator:

$$\psi_{n_{step}i} = \left| \frac{\bar{z}_{n_{step}i} - \underline{z}_{n_{step}i}}{\bar{z}_{n_{step}i}} \right|, \quad \text{with } i \in \mathbb{N} \tag{14.10}$$

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

Algorithm 11 Run of forward pass (n^{th} iteration)

1: Simulate sets $\{(\omega_t^g), t \in \{1, \ldots, T\}\}$ of equally distributed scenarios: for $g \in \Omega^{\mathscr{G}} = \{1, \ldots, G\}$.
2: **for** $g \in \Omega_{\mathscr{G}}$ **do**
3: Solve the following linear sub-problem.

$$[AP_0^n] \begin{cases} & Q_0 = \min\limits_{x_0, \theta_1} c_0^\top x_0 + \theta_1 \\ u.c. & A_0 x_0 = \omega_0, \quad [\pi_0(\omega_0)] \\ & x_0 \geqslant 0 \\ & \theta_1 + (\beta_1^j)^\top x_0 \geqslant \alpha_1^j, \quad j \in \{1, \ldots, G, \ldots, nG\} \end{cases} \tag{14.4}$$

4: Store the primal solution (x_0^g) of the problem $[AP_{0,g}^n]$.
5: **for** $t \in \{1, \ldots, T\}$ **do**
6: Solve the following linear sub-problem.

$$[AP_{t,g}^n] \begin{cases} & Q_t^g(x_{t-1}^g, \omega_t^g) = \min\limits_{x_t, \theta_{t+1}} c_t^\top x_t + \theta_{t+1} \\ s.c. & A_t x_t = \omega_t^g - E_t x_{t-1}^g, \quad [\pi_t(\omega_t^g)] \\ & x_t \geqslant 0 \\ & \theta_{t+1} + (\beta_{t+1}^j)^\top x_t \geqslant \alpha_{t+1}^j, \quad j \in \{1, \ldots, G, \ldots, nG\} \end{cases} \tag{14.5}$$

7: Store the primal solution (x_t^g) of the problem $[AP_{t,g}^n]$.
8: **end for**
9: 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: Compute the total cost in forward pass at iteration n: $\bar{z}_n = \frac{1}{G} \sum_{g=1}^G \bar{z}_n^g$.

14.2.2 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 (14.2 - 14.3) 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, an ARMA process of order 1 is often chosen. In the random data vector ω_t two different parts has to

Algorithm 12 Run of backward pass

for $t = T, T-1, \ldots, 1$ **do**
  for x_{t-1}^g, $g \in \{1, \ldots, G\}$ **do**
    for ω_t^l, $l \in \{1, \ldots, L\}$ **do**
      Solve the following linear sub-problem.

$$[AP_{t,l}^{n,g}] \begin{cases} & Q_t^l(x_{t-1}^g, \omega_t^l) = \min\limits_{x_t, \theta_{t+1}} c_t^\top x_t + \theta_{t+1} \\ s.c. & A_t x_t = \omega_t^l - E_t x_{t-1}^g, \quad [\pi_t(\omega_t^l)] \\ & x_t \geqslant 0 \\ & \theta_{t+1} + (\beta_{t+1}^j)^\top x_t \geqslant \alpha_{t+1}^j, \quad j \in \{1, \ldots, G, \ldots, (n+1)G\} \end{cases} \tag{14.6}$$

      Store the dual solution $\pi_t(\omega_t^l)$ and the primal one $Q_t^l(x_{t-1}^g, \omega_t^l)$ of the linear sub-problem $[AP_{t,l}^{n,g}]$.
      Compute the cut that goes with the l^{th} hazard draw:

$$\begin{cases} \alpha_{t,l}^g = Q_t^l(x_{t-1}^g, \omega_t^l) + \pi_t(\omega_t^l)^\top E_t x_{t-1}^g \\ \beta_{t,l}^g = E_t^\top \pi_t(\omega_t^l) \end{cases} \tag{14.7}$$

    end for
    Compute the g^{th} new Benders cut at time t at iteration n. It is defined as the mean value of the cuts obtained before:

$$\begin{cases} \alpha_t^k = \frac{1}{L} \sum\limits_{l=1}^{L} \alpha_{t,l}^g \\ \beta_t^k = \frac{1}{L} \sum\limits_{l=1}^{L} \beta_{t,l}^g \end{cases} \quad \text{where } k = nG + g \tag{14.8}$$

  end for
end for
Solve the following linear sub-problem:

$$[AP_0^n] \begin{cases} & Q_0 = \min\limits_{x_0, \theta_1} c_0^\top x_0 + \theta_1 \\ s.c. & A_0 x_0 = \omega_0, \quad [\pi_0(\omega_0)] \\ & x_0 \geqslant 0 \\ & \theta_1 + (\beta_1^j)^\top x_0 \geqslant \alpha_1^j, \quad j \in \{1, \ldots, G, \ldots, (n+1)G\} \end{cases} \tag{14.9}$$

Save the cost *backward* $\underline{z}_n = Q_0$.

be distinguished from now on:

- ω_t^{ind} is the random data vector corresponding to the independent random quantities.

- ω_t^{dep} is the random data vector corresponding to the time-related random quantities.

And ω_t^{dep} fulfills the following recurrence equation:

$$\frac{\omega_t^{\text{dep}} - \mu_{\omega,t}}{\sigma_{\omega,t}} = \psi_1 \frac{\omega_{t-1}^{\text{dep}} - \mu_{\omega,t-1}}{\sigma_{\omega,t-1}} + \psi_2 \epsilon_t \tag{14.11}$$

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

$$[LP'_t] \begin{cases} & Q_t(x_{t-1}, \omega_{t-1}^{\text{dep}}, \omega_t) = \min c_t^\top x_t + \mathscr{Q}_{t+1}(x_t, \omega_t^{\text{dep}}) \\ u.c. & \quad A_t x_t = P\omega_t^{\text{dep}} - E_t x_{t-1}, \qquad [\pi_t(\omega_t)] \\ & \quad x_t \geqslant 0 \end{cases} \tag{14.12}$$

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

The variable ω_t^{dep} is a random process. Thus the above problem is solved using specific values ω_t^l of this variable. To get them we apply a Markov process that is we simulate different values of the white noise ϵ_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 ω_t^{dep}. The computation of Benders cuts is then a bit different:

$$\begin{aligned} \frac{\partial Q_t(x_{t-1}, \omega_{t-1}^{\text{dep}}, \omega_t)}{\partial \omega_{t-1}^{\text{dep}}} &= \frac{\partial Q_t(x_{t-1}, \omega_{t-1}^{\text{dep}}, \omega_t)}{\partial \omega_t^{\text{dep}}} \frac{\partial \omega_t^{\text{dep}}}{\partial \omega_{t-1}^{\text{dep}}} \\ &= \pi_t(\omega_t)^\top P \psi_1 \frac{\sigma_{\omega,t}}{\sigma_{\omega,t-1}}, \end{aligned} \tag{14.13}$$

Backward pass has to be modified as presented in Algorithm 13. Some new computation steps have to be taken into account.

14.2.3 Non-convexity and conditional 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 convexity property 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, Pereira, Campodonico, and Kelman [35] suggested a new algorithm in order to efficiently approximate the Bellman functions using explicitly the dependence

Algorithm 13 Run of backward pass with time-related random quantities (AR1 process)

Pick up the set of the following pairs: $\{x_t^g, \omega_t^{\text{dep},g}\}$ for $g \in \{1, \ldots, G\}$, $t \in \{1, \ldots, T\}$.
for $t = T, T-1, \ldots, 1$ **do**
  for $(x_{t-1}^g, \omega_{t-1}^{\text{dep},g})$, $g \in \{1, \ldots, G\}$ **do**
    for $l \in \{1, \ldots, L\}$ **do**
      Produce a value for the white noise ϵ_t^l.
      Compute the element $\hat{\omega}_t^l$ knowing the previous random quantity $\omega_{t-1}^{\text{dep},g}$:

$$\hat{\omega}_t^l = \sigma_{\omega,t} \left(\psi_1 \frac{\omega_{t-1}^{\text{dep},g} - \mu_{\omega,t-1}}{\sigma_{\omega,t-1}} + \psi_2 \epsilon_t^l \right) + \mu_{\omega,t} \tag{14.14}$$

      Solve the following linear sub-problem.

$$[AP_{t,l}^{'n,g}] \begin{cases} & Q_t^l(x_{t-1}^g, \omega_{t-1}^{\text{dep},g}, \hat{\omega}_t^l) = \min\limits_{x_t, \theta_{t+1}} c_t^\top x_t + \theta_{t+1} \\ u.c. & A_t x_t = P\hat{\omega}_t^l - E_t x_{t-1}^g, \quad [\pi_t(\hat{\omega}_t^l)] \\ & x_t \geqslant 0 \\ & \theta_{t+1} + (\beta_{t+1}^j)^\top x_t + (\gamma_{t+1}^j)^\top \hat{\omega}_t^l \geqslant \alpha_{t+1}^j, \quad j \in \{1, \ldots, G, \ldots, (n+1)G\} \end{cases} \tag{14.15}$$

      Store the dual solution $\pi_t(\omega_t^l)$ and the primal one $Q_t^l(x_{t-1}^g, \omega_{t-1}^{\text{dep},g}, \hat{\omega}_t^l)$ of the primal problem $[AP_{t,l}^{'n,g}]$.
      Compute the cut that goes with the l^{th} hazard draw:

$$\begin{cases} \alpha_{t,l}^g = Q_t^l(x_{t-1}^g, \omega_{t-1}^{\text{dep},g}, \hat{\omega}_t^l) + \pi_t(\hat{\omega}_t^l)^\top \left(E_t x_{t-1}^g - \psi_1 \frac{\sigma_{\omega,t}}{\sigma_{\omega,t-1}} P \omega_{t-1}^{\text{dep},g} \right) \\ \beta_{t,l}^g = E_t^\top \pi_t(\hat{\omega}_t^l) \\ \gamma_{t,l}^g = \psi_1 \frac{\sigma_{\omega,t}}{\sigma_{\omega,t-1}} P^\top \pi_t(\hat{\omega}_t^l) \end{cases} \tag{14.16}$$

    end for
    Compute the g^{th} new Benders cut at time t at iteration n defined as the mean value of the cuts obtained before for $k = nG + g$:

$$\alpha_t^k = \frac{1}{L} \sum_{l=1}^{L} \alpha_{t,l}^g \ , \quad \beta_t^k = \frac{1}{L} \sum_{l=1}^{L} \beta_{t,l}^g \ , \quad \gamma_t^k = \frac{1}{L} \sum_{l=1}^{L} \gamma_{t,l}^g$$

  end for
end for
Solve the following linear sub-problem.

$$[AP_0^{'n}] \begin{cases} & Q_0 = \min\limits_{x_0, \theta_1} c_0^\top x_0 + \theta_1 \\ u.c. & A_0 x_0 = \omega_0, \quad [\pi_0(\omega_0)] \\ & x_{\geqslant} 0 \\ & \theta_1 + (\beta_1^j)^\top x_0 + (\gamma_1^j)^\top \omega_0^{\text{dep}} \geqslant \alpha_1^j, \quad j \in \{1, \ldots, G, \ldots, (n+1)G\} \end{cases} \tag{14.17}$$

Save the backward cost $\underline{z}_n = Q_0$.

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 [35] and in [17].

In the library, we propose two approaches to deal with this non convexity:

A tree approach

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 $\zeta_1, \ldots, \zeta_M$. The following Markov model is then used:

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

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

However when the dimension of the non convex uncertainties is low, it is an approach to consider. The finite number of states , and the probabilities linking states between two successive time step leads to a tree representation of the uncertainties. As the approach is classical, we don't detail this version of the algorithm. We only detail the second approach which is in fact in spirit very similar.

A regression based approach

In that case the modelization used in the library is somewhat different from the one described in both articles. In order to avoid tree discretization in this second approach, 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_t^u at each stage t
- $\tilde{I}$ is a partition of the space of the non-convex random quantities S.

$$\tilde{I} = \{\underline{I} = (i_1, \ldots, i_d), i_1 \in \{1, \ldots, I_1\}, \ldots, i_d \in \{1, \ldots, I_d\}\}$$

- $\{D_{\underline{I}}\}_{\underline{I} \in \tilde{I}}$ is the set of meshes of the set of scenarios.
- $M_M = \prod_{k=1}^{d} I_k$ denotes the number of meshes at each stage.

The backward step with both time-related and non-convex random quantities is presented in Algorithm 14.

Algorithm 14 Run of the backward pass with time-related (AR1) and non-convex random quantities

Pick up the set of the following pairs: $\{x_t^g, \omega_t^{\text{dep},g}\}$ for $g \in \{1, \dots, G\}$, $t \in \{1, \dots, T\}$.
for $t = T, T-1, \dots, 1$ **do**
  Generate values for the non-convex random quantities at time t knowing the scenarios at time $t-1$: s_t^u, $u \in \{1, \dots, U\}$.
  for $(x_{t-1}^g, \omega_{t-1}^{\text{dep},g})$, $g \in \{1, \dots, G\}$ **do**
    for $u \in \{1, \dots, U\}$ **do**
      Consider a scenario s_t^u in the mesh $D_{\underline{I}}$.
      for $l \in \{1, \dots, L\}$ **do**
        Produce a value for the white noise ϵ_t^l.
        Compute the element $\hat{\omega}_t^l$ knowing the previous random quantity $\omega_{t-1}^{\text{dep},g}$:

$$\hat{\omega}_t^l = \sigma_{\omega,t} \left(\psi_1 \frac{\omega_{t-1}^{\text{dep},g} - \mu_{\omega,t-1}}{\sigma_{\omega,t-1}} + \psi_2 \epsilon_t^l \right) + \mu_{\omega,t} \tag{14.18}$$

        Pick up the cuts corresponding the mesh $D_{\underline{I}}$: $\left\{ \alpha_{t+1}^{\underline{I},j}(s), \beta_{t+1}^{\underline{I},j}(s), \gamma_{t+1}^{\underline{I},j}(s) \right\}$, $j \in \{1, \dots, (n+1)G\}$.
        Solve the following linear sub-problem:

$$[AP_{t,l}^{'n,g}] \begin{cases} Q_t^l(x_{t-1}^g, \omega_{t-1}^{\text{dep},g}, \hat{\omega}_t^l, s_t^u) = \min\limits_{x_t, \theta_{t+1}} c_t(s_t^u)^\top x_t + \theta_{t+1}(s_t^u) \\ s.c. \quad A_t(s_t^u) x_t = P\hat{\omega}_t^l - E_t x_{t-1}^g, \quad [\pi_t(\hat{\omega}_t^l, s_t^u)] \\ x_t \geqslant 0 \\ \theta_{t+1}(s_t^u) + (\beta_{t+1}^{\underline{I},j}(s_t^u))^\top x_t + (\gamma_{t+1}^{\underline{I},j}(s_t^u))^\top \hat{\omega}_t^l \geqslant \alpha_{t+1}^{\underline{I},j}(s_t^u), \\ \quad j \in \{1, \dots, G, \dots, nG\} \end{cases} \tag{14.19}$$

        Store the dual solution $\pi_t(\hat{\omega}_t^l)$ and the primal solution $Q_t^l(x_{t-1}^g, \omega_{t-1}^{\text{dep},g}, \hat{\omega}_t^l, s_t^u)$ of the problem $[AP_{t,l}^{'n,g}]$.
        Calculate the corresponding cut at the l^{th} draw of uncertainties:

$$\begin{aligned} \hat{\alpha}_{t,l}^{g,\underline{I}}(s_t^u) &= Q_t^l(x_{t-1}^g, \omega_{t-1}^{\text{dep},g}, \hat{\omega}_t^l) + \pi_t(\hat{\omega}_t^l)^\top \left(E_t x_{t-1}^g - \psi_1 \frac{\sigma_{\omega,t}}{\sigma_{\omega,t-1}} P \omega_{t-1}^{\text{dep},g} \right) \\ \hat{\beta}_{t,l}^{g,\underline{I}}(s_t^u) &= E_t^\top \pi_t(\hat{\omega}_t^l) \\ \hat{\gamma}_{t,l}^{g,\underline{I}}(s_t^u) &= \psi_1 \frac{\sigma_{\omega,t}}{\sigma_{\omega,t-1}} P^\top \pi_t(\hat{\omega}_t^l) \end{aligned}$$

      end for

Compute the cut for a non-convex random quantity s_t^u at time t at iteration n: it is defined as the weighted average on the L Benders cut obtained before:

$$\hat{\alpha}_t^{g,\underline{I}}(s_t^u) = \frac{1}{L}\sum_{l=1}^{L} \hat{\alpha}_{t,l}^{g,\underline{I}}(s_t^u)$$
$$\hat{\beta}_t^{g,\underline{I}}(s_t^u) = \frac{1}{L}\sum_{l=1}^{L} \hat{\beta}_{t,l}^{g,\underline{I}}(s_t^u), \qquad j = nG + g$$
$$\hat{\gamma}_t^{g,\underline{I}}(s_t^u) = \frac{1}{L}\sum_{l=1}^{L} \hat{\gamma}_{t,l}^{g,\underline{I}}(s_t^u)$$

end for

for $\underline{I}_i, i \in \{1, \ldots, M_M\}$ **do**

Compute the g^{th} new cut of the mesh $D_{\underline{I}_i}$ at time t at iteration n defined as the conditional expectation with respect to the scenario u at time t:

$$\begin{cases} \alpha_t^{j,I}(s_{t-1}^u) = \mathbb{E}\left[\hat{\alpha}_t^{g,\underline{I}}(s_t^u)|s_{t-1}^u\right], \\ \beta_t^{j,I}(s_{t-1}^u) = \mathbb{E}\left[\hat{\beta}_t^{g,\underline{I}}(s_t^u)|s_{t-1}^u\right], \qquad j = nG + g \\ \gamma_t^{j,I}(s_{t-1}^u) = \mathbb{E}\left[\hat{\gamma}_t^{g,\underline{I}}(s_t^u)|s_{t-1}^u\right] \end{cases} \tag{14.20}$$

end for

end for

end for

Solve the initial linear sub problem $[AP_0'^n]$.

Save the backward cost $\underline{z}_n = Q_0$.

14.3 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

- the `backwardForwardSDDPTree` class for the tree method,
- the `backwardForwardSDDP` class for the regression based approach.

These classes takes as input three non-defined classes.

14.3.1 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. The class is used both with the tree approach and the regression based approach.
  In some ways this class is the place where the technical aspects of the problem are adjusted. This class describes backward and forward passes. Four methods should be implemented:
  - `updateDates`: establishes the new set of dates: $(t, t + dt)$.
  - `oneStepForward`: solves the different transition linear problems during the forward pass for a particle, a random vector and an initial state:
    * the state $(x_{t-dt}, w_t^{\text{dep}})$ is given as input 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^{dep} (demand, flow constraints) and permits to get the optimal x_t.
    * Using iid sampling, w_{t+dt}^{dep} is estimated.
    * return $(x_t, w_{t+dt}^{\text{dep}})$ as the following state and (x_t, w_t^{dep}) that will be used as the state to visit during next backward resolution.
  - `oneStepBackward`: solves the different transition linear problems during the backward pass for a particle, a random vector and an initial state.
    * The vector (x_t, w_t^{dep}) is given as input if $t \geq 0$; otherwise, the input is $(x_{-dt}, w_0^{\text{dep}})$.
    * If $t \geq 0$, sample to calculate w_{t+dt}^{dep} in order to get the state $(x_t, w_{t+dt}^{\text{dep}})$ at the beginning of the period of resolution of the LP. If $t < 0$, the state is $(x_{-dt}, w_0^{\text{dep}})$.
    * Solve the LP from date t to next date $t + dt$ (if equally spaced periods) for the variable x_{t+dt}.

    - Return the 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.

```
// Copyright (C) 2016 EDF
// All Rights Reserved
// This code is published under the GNU Lesser General Public License (GNU LGPL)
#ifndef OPTIMIZERSDDPBASE_H
#define OPTIMIZERSDDPBASE_H
#include <Eigen/Dense>
#include "StOpt/sddp/SDDPCutOptBase.h"
#include "StOpt/core/grids/OneDimRegularSpaceGrid.h"
#include "StOpt/core/grids/OneDimData.h"
#include "StOpt/sddp/SimulatorSDDPBase.h"
#include "StOpt/sddp/SimulatorSDDPBaseTree.h"


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

namespace StOpt
{

/// \class OptimizerSDDPBase OptimizerSDDPBase.h
///  Base class for optimizer for Dynamic Programming
class OptimizerSDDPBase
{


public :

    OptimizerSDDPBase() {}

    virtual ~OptimizerSDDPBase() {}


    /// \brief Optimize the LP during backward resolution
    /// \param p_linCut cuts used for the PL (Benders for the Bellman value at the end of
        the time step)
    /// \param p_aState store the state, and 0.0 values
    /// \param p_particle  the particle n dimensional value associated to the regression
    /// \param p_isample   sample number for independant uncertainties
    /// \return  a vector with the optimal value and the derivatives if the function value
        with respect to each state
    virtual Eigen::ArrayXd oneStepBackward(const StOpt::SDDPCutOptBase &p_linCut, const std
        ::tuple< std::shared_ptr<Eigen::ArrayXd>, int, int > &p_aState, const Eigen::
        ArrayXd &p_particle, const int &p_isample) const = 0;

    /// \brief Optimize the LP during forward resolution
    /// \param p_aParticle a particule in simulation part to get back cuts
    /// \param p_linCut cuts used for the PL  (Benders for the Bellman value at the end of
        the time step)
    /// \param p_state  store the state, the particle number used in optimization and mesh
        number associated to the particle. As an input it constains the current state
    /// \param p_stateToStore for backward resolution we need to store \f$ (S_t,A_{t-1},D_{
        t-1}) \f$  where p_state in output is \f$ (S_t,A_{t},D_{t}) \f$
    /// \param p_isimu           number of teh simulation used
    virtual double oneStepForward(const Eigen::ArrayXd &p_aParticle, Eigen::ArrayXd &
        p_state,  Eigen::ArrayXd &p_stateToStore, const StOpt::SDDPCutOptBase &p_linCut,
                                  const int &p_isimu) const = 0 ;
```

```
51
52
53      /// \brief update the optimizer for new date
54      ///         - In Backward mode, LP resolution achieved at date p_dateNext,
55      ///            starting with uncertainties given at date p_date and  evolving to give
            uncertainty at date p_dateNext,
56      ///         - In Forward mode,  LP resolution achieved  at date p_date,
57      ///            and uncertainties evolve till date p_dateNext
58      ///         .
59      virtual void updateDates(const double &p_date, const double &p_dateNext) = 0 ;
60
61      /// \brief Get an admissible state for a given date
62      /// \param  p_date   current date
63      /// \return an admissible state
64      virtual Eigen::ArrayXd oneAdmissibleState(const double &p_date) = 0 ;
65
66      /// \brief get back state size
67      virtual int getStateSize() const = 0;
68
69      /// \brief get the backward simulator back
70      virtual std::shared_ptr< StOpt::SimulatorSDDPBase > getSimulatorBackward() const = 0;
71
72      /// \brief get the forward simulator back
73      virtual std::shared_ptr< StOpt::SimulatorSDDPBase > getSimulatorForward() const = 0;
74
75  };
76  }
77  #endif /* OPTIMIZERSDDPBASE_H */
```

- A 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.

At the opposite of the class where the transition is described, the simulator are of course different for the tree approach and the regression based approach as the first gives only a finite number of states.

An abstract class for simulators using the regression based methods is defined below:

```
1  // Copyright (C) 2016 EDF
2  // All Rights Reserved
3  // This code is published under the GNU Lesser General Public License (GNU LGPL)
4  #ifndef SIMULATORSDDPBASE_H
5  #define SIMULATORSDDPBASE_H
6  #include <Eigen/Dense>
7
8  /* \file SimulatorBase.h
9   * \brief Abstract class for simulators for SDDP method
10 * \author Xavier Warin
11 */
12 namespace StOpt
13 {
14 /// \class SimulatorSDDPBase SimulatorSDDPBase.h
15 /// Abstract class for simulators used for SDDP
16 class SimulatorSDDPBase
17 {
18 public :
19
20     /// \brief Constructor
21     SimulatorSDDPBase() {}
22
```

```
23      /// \brief Destructor
24      virtual ~SimulatorSDDPBase() {}
25
26      /// \brief Get back the number of particles (used in regression part)
27      virtual  int getNbSimul() const = 0;
28      /// \brief Get back the number of sample used (simulation at each time step , these
            simulations are independent of the state)
29      virtual  int getNbSample() const = 0;
30      /// \brief Update the simulator for the date :
31      /// \param p_idateCurr   index in date array
32      virtual void updateDateIndex(const int &p_idateCur)  = 0;
33      /// \brief get one simulation
34      /// \param p_isim  simulation number
35      /// \return the particle associated to p_isim
36      /// \brief get  current Markov state
37      virtual  Eigen::VectorXd getOneParticle(const int &p_isim) const = 0;
38      /// \brief get  current Markov state
39      virtual  Eigen::MatrixXd getParticles() const = 0;
40      /// \brief Reset the simulator (to use it again for another SDDP sweep)
41      virtual  void resetTime() = 0;
42      /// \brief in simulation  part of SDDP reset  time  and reinitialize uncertainties
43      /// \param p_nbSimul  Number of simulations to update
44      virtual  void updateSimulationNumberAndResetTime(const int &p_nbSimul) = 0;
45  };
46  }
47  #endif /* SIMULATORSDDPBASE_H */
```

An abstract class derived from the previous class for simulators using tree methods is defined below:

```
1   // Copyright (C) 2019 EDF
2   // All Rights Reserved
3   // This code is published under the GNU Lesser General Public License (GNU LGPL)
4   #ifndef SIMULATORSDDPBASETREE_H
5   #define SIMULATORSDDPBASETREE_H
6   #include <memory>
7   #include <Eigen/Dense>
8   #include "geners/BinaryFileArchive.hh"
9   #include "StOpt/core/utils/comparisonUtils.h"
10  #include "StOpt/dp/SimulatorDPBaseTree.h"
11  #include "StOpt/sddp/SimulatorSDDPBase.h"
12
13  /* \file SimulatorBaseTree.h
14   * \brief Base class for simulators for SDDP method with uncertainties breaking concavity/
         convexity in a tree
15  * \author Xavier Warin
16  */
17  namespace StOpt
18  {
19  /// \class SimulatorSDDPBaseTree SimulatorSDDPBaseTree.h
20  /// Base class for simulators used for SDDP  with uncertainties breaking concavity/
        convexity in a Tree
21  class SimulatorSDDPBaseTree : public SimulatorSDDPBase, public SimulatorDPBaseTree
22  {
23
24  public :
25
26      /// \brief Constructor
27      /// \param  p_binforTree  binary geners archive  with structure
28      ///         - dates      ->  eigen array of dates, size ndate
29      ///         - nodes     ->  nDate array , each array containing nodes coordinates  with
             size  (ndim, nbNodes)
30      ///         - proba      -> for a point i at a given date and a point j at next date ,
            prob(i,j) gives the probability to go from node i to node j.
31      ///
32      SimulatorSDDPBaseTree(const std::shared_ptr<gs::BinaryFileArchive>   &p_binForTree):
            SimulatorDPBaseTree(p_binForTree) {}
33
```

```
34
35      /// \brief Destructor
36      virtual ~SimulatorSDDPBaseTree() {}
37
38      /// \brief
39      /// \brief Get back the number of particles
40      virtual int getNbSimul() const
41      {
42          return 0;
43      }
44
45      /// \brief Get back the number of sample used (simulation at each time step , these
            simulations are independent of the state)
46      virtual int getNbSample() const
47      {
48          return 0 ;
49      }
50
51
52      /// \brief get one simulation
53      /// \param p_isim  simulation number
54      /// \return the particle associated to p_isim
55      virtual Eigen::VectorXd getOneParticle(const int &p_isim) const
56      {
57          return  m_nodesCurr.col(getNodeAssociatedToSim(p_isim));
58      }
59
60      /// \brief get  current Markov state
61      virtual  Eigen::MatrixXd getParticles() const
62      {
63          return Eigen::MatrixXd();
64      }
65
66      /// \brief Reset the simulator (to use it again for another SDDP sweep)
67      virtual  void resetTime() {}
68
69      /// \brief in simulation  part of SDDP reset  time  and reinitialize uncertainties
70      /// \param p_nbSimul  Number of simulations to update
71      virtual  void updateSimulationNumberAndResetTime(const int &p_nbSimul) {}
72
73      /// \brief Update the simulator for the date :
74      /// \param p_idateCurr   index in date array
75      virtual void updateDateIndex(const int &p_idateCur)
76      {
77          load(p_idateCur);
78      }
79
80
81  };
82  }
83
84  #endif /* SIMULATORSDDPBASETREE_H */
```

14.3.2 Architecture

We only detail the SDDP architecture for the regression based approach as the tree approach uses the same algorithm.
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 the one in other time-steps, `forwardSDDP`, `backwardSDDP`, `backwardForwardSDDP` inputs have been omitted for simplification). A more rigorous theoretical explanation is available in the previous part.

Remark 33 *In order to use the tree method,*
`forwardSDDP`, `backwardSDDP`, `backwardforwardSDDP` can be replaced by some specialized version for trees called
`forwardSDDPTree`, `backwardSDDPTree`, `backwardforwardSDDPTree` in the library.

Three colors have been used: blue parts correspond to the use of functions implemented in the **TransitionOptimizer** class, red parts correspond to the use of **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 an StOpt user is only the **TransitionOptimizer** and the **Simulator** (blue and red parts), other functions and described loops are already implemented and managed by the library.

Algorithm 15 Run of backwardforwardSDDP(),the main function)

1: Init: $x_t^g =$ **TransitionOptimizer.oneAdmissibleState(t)**, for $g \in \{1, \dots, G\}$ and $t \in \{1, \dots, T-1\}$, $n = 0$, $\psi = \infty$.
2: **while** $\psi > \epsilon$ and $n < n_{max}$ **do**
3: **StOpt**
4: $V_b =$ backwardSDDP() *Using the previously computed set $(x_t^g)_{t,g}$ and creating a set of cuts.*
5: $V_f =$ 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**

14.3.3 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

Algorithm 16 Run of `forwardSDDP` (n^{th} iteration)

for $g \in \Omega_{\mathscr{G}}$ **do**

  $iStep = 0$: index of the date stored in the simulator

  for $t \in \{0, \dots, T\}$ **do**

    TransitionOptimizer.updateDates$(t, t+1)$: update the required data following the current time step (iterator over current time step, average demand,...)

    SimulatorSim.updateDateIndex$(iStep)$: give the random quantities (ω_t^g) for the scenario g at time t

    StOpt Read the previously computed files to gather $\alpha_{t+1}^j, \beta_{t+1}^j$, for $j \in \{1, \dots, G, \dots, nG\}$

    TransitionOptimizer.oneStepForward():
    Solve the following linear sub-problem.

$$[AP_{t,g}^n] \begin{cases} & Q_t^g(x_{t-1}^g, \omega_t^g) = \min\limits_{x_t, \theta_{t+1}} c_t^\top x_t + \theta_{t+1} \\ s.c. \quad & A_t x_t = \omega_t^g - E_t x_{t-1}^g, \quad [\pi_t(\omega_t^g)] \\ & x_t \geqslant 0 \\ & \theta_{t+1} + (\beta_{t+1}^j)^\top x_t \geqslant \alpha_{t+1}^j, \quad j \in \{1, \dots, G, \dots, nG\} \end{cases} \tag{14.21}$$

    Compute the cost for current time step $c_t^\top x_t^g$
    Return: the primal solution (x_t^g) of the problem

    StOpt Store the primal solution (x_t^g) of the problem $[AP_{t,g}^n]$

    $iStep = iStep + 1$

  end for

  StOpt Compute the cost for scenario g, at iteration n: $\bar{z}_n^g = \sum_{t=0}^T c_t x_t^g$

end for

StOpt Compute the total cost in forward pass at iteration n: $\bar{z}_n = \frac{1}{G} \sum_{g=1}^G \bar{z}_n^g$

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 **TransitionOptimizer** class should be specific to your problem (it is given as an argument of the `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 **TransitionOptimizer** class inherits from the class `OptimizerSDDPBase`. You will then have to implement the following functions.

Algorithm 17 Run of `backwardSDDP`

$iStep = NbStep$: update the simulator time index to give the uncertainty at T
for $t = T, T-1, \dots, 0$ **do**
  StOpt Read the previously computed files to gather x_{t-1}^g, for $g \in \{1, \dots, G, \}$
  TransitionOptimizer.updateDates$(t-1, t)$: update the required data following the current time step (iterator over current time step, average demand,...)
  SimulatorOpt.updateDateIndex$(iStep)$: give the random quantities for the L scenarios at time t
  StOpt Read the previously computed files to gather $\alpha_{t+1}^j, \beta_{t+1}^j$, for $j \in \{1, \dots, G, \dots, nG\}$
  for x_{t-1}^g, $g \in \{1, \dots, G\}$ **do**
    for ω_t^l, $l \in \{1, \dots, L\}$ **do**
      TransitionOptimizer.oneStepBackward()
      Solve the following linear sub-problem.

$$[AP_{t,l}^{n,g}] \begin{cases} & Q_t^l(x_{t-1}^g, \omega_t^l) = \min_{x_t, \theta_{t+1}} c_t^\top x_t + \theta_{t+1} \\ s.c. & A_t x_t = \omega_t^l - E_t x_{t-1}^g, \quad [\pi_t(\omega_t^l)] \\ & x_t \geqslant 0 \\ & \theta_{t+1} + (\beta_{t+1}^j)^\top x_t \geqslant \alpha_{t+1}^j, \ j \in \{1, \dots, G, \dots, (n+1)G\} \end{cases} \tag{14.22}$$

      Return: the dual solution $\pi_t(\omega_t^l)$ and the primal one $Q_t^l(x_{t-1}^g, \omega_t^l)$ of the linear sub-problem $[AP_{t,l}^{n,g}]$
      $iStep = iStep + 1$,
    end for
    StOpt Compute the g^{th} new Benders cut at time t at iteration n: α_t^j, β_t^j, for $j \in \{(n-1)G, (n-1)G+1, \dots, nG\}$
  end for
end for
StOpt Save the cost *backward* $\underline{z}_n = Q_0$

- The `updateDates` function allows to update the data stored by the optimizer, fitting the times indicated as argument.

```
1
2     /// \brief update the optimizer for new date
3     ///         - In Backward mode, LP resolution achieved at date p_dateNext,
4     ///            starting with uncertainties given at date p_date and  evolving to
          give uncertainty at date p_dateNext,
5     ///         - In Forward mode,  LP resolution achieved  at date p_date,
6     ///            and uncertainties evolve till date p_dateNext
7     ///         .
```

If your transition problem 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 `p_dateNext` 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 give an admissible state (that means a state respecting all the constraints) for the time step given as an argument.

```
1    /// \param  p_date   current date
```

- The `oneStepBackward` function allows to compute one step of the backward pass.

```
1    /// \return  a vector with the optimal value and the derivatives if the function
         value with respect to each state
```

The first argument is the cuts already selected for the current time step. 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 $cuts(0,j) = \alpha_{t+1}^j$, $cuts(i,j) = \beta_{i-1,t+1}^j$ $j \in \{1,\dots,G,\dots,(n+1)G\}$,$i \in \{1,\dots,nb_{state}\}$.
You will 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 `p_astate` (**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 $nb_{state}+1$ containing as a first argument the value of the objective function at the solution, and then for $i \in \{1,\dots,nb_{state}\}$ 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.

```
1    /// \param p_isimu           number of teh simulation used
```

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 $[AP_{t,g}^n]$ (for a given scenario g and a given time step t) which appears for both the forward and the backward pass. This function creating and generating the linear problem will be called in 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.

A simulator for the regression based method

An implemented **Simulator** deriving 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.

```
1    virtual  int getNbSimul() const = 0;
```

- The `getNbSample` function returns the number of simulations of random quantities that are not used in the regression part. 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.

```
1    virtual  int getNbSample() const = 0;
```

- The `updateDateIndex` function is really similar to the optimizer one. However you just have one argument (the time step index) here. It is also here that you have to generate new random quantities for the resolution.

```
1    /// \param p_idateCurr   index in date array
```

- The `getOneParticle` and the `getParticles` functions should return the quantities used in regression part.

```
1    /// \brief get  current Markov state
```

```
1    /// \brief get  current Markov state
```

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

A simulator for the tree approach

A simulator using tree should be derived from the class `SimulatorSDDPBaseTree`. As the class `SimulatorSDDPBaseTree` is derived from the `SimulatorSDDPBase` class all previously described methods have to be given.
Besides a geners archive is used to load:

- The dates used for the simulator to estimate the set of possible states,
- At each date, a set of d dimensional points defining the set of discrete values of the state in the tree,
- At each date a two dimensional array giving the probability transition in the tree to go from a node i at the current date to a node j at the following date.

Then the simulator implemented should call the based constructor loading the archive:

```
1     /// \brief Get back the number of particles
2     virtual int getNbSimul() const
3     {
4         return 0;
5     }
6
7     /// \brief Get back the number of sample used (simulation at each time step , these
          simulations are independent of the state)
```

Then the user should have generated such an archive. An example using a trinomial tree method for an AR1 class is given in the c++ test cases in the simulator directory by the class `MeanRevertingSimulatorTree`.

The methods to implement necessary are the following one:

- The `getNodeAssociatedToSim` method give for a simulation identified by the particle number the node in the tree visited

- The `stepForward` method updates the simulation date index by one, and samples the nodes visited in forward resolution.

14.3.4 Set of parameters

Implementing some regression based method

The basic function `backwardForwardSDDP` should be called to use the SDDP part of the library with conditional cuts calculated by regressions. 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.

```
/// \brief Achieve forward and backward sweep by SDDP
/// \param  p_optimizer           defines the optimiser necessary to optimize a step for one simulation solving a LP
/// \param  p_nbSimulCheckForSimu defines the number of simulations to check convergence
/// \param  p_initialState        initial state at the beginning of simulation
/// \param  p_finalCut            object of final cuts
/// \param  p_dates               vector of exercised dates, last date corresponds to the final cut object
/// \param  p_meshForReg          number of mesh for regression in each direction
/// \param  p_nameRegressor       name of the archive to store regressors
/// \param  p_nameCut             name of the archive to store cuts
/// \param  p_nameVisitedStates   name of the archive to store visited states
/// \param  p_iter                maximum iteration of SDDP, on return the number of iterations achieved
/// \param  p_accuracy            accuracy asked , on return estimation of accuracy achieved (expressed in %)
/// \param  p_nStepConv           every p_nStepConv convergence is checked
/// \param  p_stringStream        dump all print messages
/// \param  p_bPrintTime          if true print time at each backward and forward step
/// \return backward and forward valorization
template<  class LocalRegressionForSDDP>
std::pair<double, double> backwardForwardSDDP(std::shared_ptr<OptimizerSDDPBase>    &p_optimizer,
        const int   &p_nbSimulCheckForSimu,
        const Eigen::ArrayXd &p_initialState,
        const SDDPFinalCut &p_finalCut,
        const Eigen::ArrayXd &p_dates,
        const Eigen::ArrayXi &p_meshForReg,
        const std::string &p_nameRegressor,
        const std::string &p_nameCut,
        const std::string &p_nameVisitedStates,
```

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 `ostringstream` in which the result of the optimization will be stored.

Implementing a tree based method

The basic function `backwardForwardSDDPTree` should be called to use the SDDP part of the library with conditional cuts calculated with trees.

```
/// \brief Achieve forward and backward sweep by SDDP with tree
/// \param  p_optimizer           defines the optimiser necessary to optimize a step for one simulation solving a LP
/// \param  p_nbSimulCheckForSimu defines the number of simulations to check convergence
/// \param  p_initialState        initial state at the beginning of simulation
/// \param  p_finalCut            object of final cuts
/// \param  p_dates               vector of exercised dates, last date corresponds to the final cut object
```

```
/// \param  p_nameCut              name of the archive to store cuts
/// \param  p_nameVisitedStates    name of the archive to store visited states
/// \param  p_iter                 maximum iteration of SDDP, on return the number of
    iterations achieved
/// \param  p_accuracy             accuracy asked , on return estimation of accuracy
    achieved (expressed in %)
/// \param  p_nStepConv            every p_nStepConv convergence is checked
/// \param  p_stringStream         dump all print messages
/// \param  p_bPrintTime           if true print time at each backward and forward step
/// \return backward and forward valorization
std::pair<double, double> backwardForwardSDDPTree(std::shared_ptr<OptimizerSDDPBase>    &
    p_optimizer,
        const int   &p_nbSimulCheckForSimu,
        const Eigen::ArrayXd &p_initialState,
        const SDDPFinalCutTree &p_finalCut,
        const Eigen::ArrayXd &p_dates,
        const std::string &p_nameCut,
        const std::string &p_nameVisitedStates,
        int &p_iter,
        double &p_accuracy,
        const int &p_nStepConv,
        std::ostringstream &p_stringStream,
        bool  p_bPrintTime = false)
```

14.3.5 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 **Parallelization** during the problem resolution is managed automatically. During compilation, if the compiler detects an MPI (Message Passing Interface)library problem resolution will be achieved in a parallelized manner.
- The **cut management**. All the cuts added at each iteration are currently serialized and stored in an archive initialized by the user. No cuts are pruned. In the future one can consider to work on cuts management [37].
- 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.

14.3.6 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

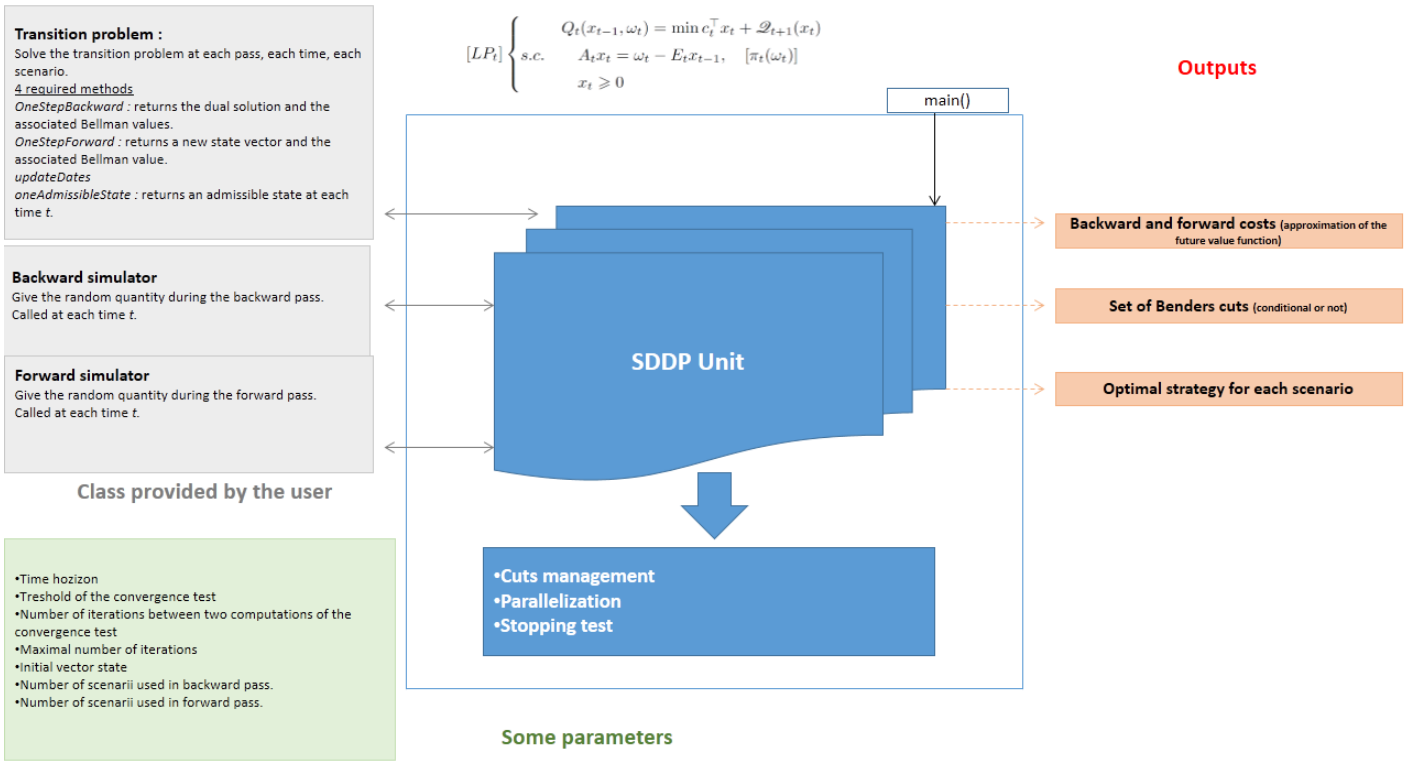


Figure 14.1: Current architecture of the generic SDDP unit

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.

14.4 Python API (only for regression based methods)

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

```
1 import StOptSDDP
2 dir(StOptSDDP)
```

that should give

$$['OptimizerSDDPBase','SDDPFinalCut','SimulatorSDDPBase','\_\_doc\_\_','\_\_file\_\_','\_\_name\_\_','\_\_package\_\_','backwardForwardSDDP']$$

The `backwardForwardSDDP` realizes 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/Pybind11SDDPOptimizers.cpp

```cpp
// Copyright (C) 2019 EDF
// All Rights Reserved
// This code is published under the GNU Lesser General Public License (GNU LGPL)
#include <pybind11/pybind11.h>
#include <pybind11/eigen.h>
#include <pybind11/stl_bind.h>
#include <pybind11/stl.h>
#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"
#include "test/c++/python/FutureCurveWrap.h"

/** \file Pybind11SDDPOptimizers.cpp
*  \brief permits to map Optimizers for SDDP
* \author Xavier Warin
*/


/// \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
    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) { }

};

namespace py = pybind11;

PYBIND11_MODULE(SDDPOptimizers, m)
{


    py::class_<OptimizeDemandSDDPWrap, std::shared_ptr<OptimizeDemandSDDPWrap>, StOpt::
        OptimizerSDDPBase >(m, "OptimizeDemandSDDP")
    .def(py::init< const  double &, const double &,  const  FutureCurve &,
         const double &,
         const std::shared_ptr<SimulatorGaussianSDDP> &,
         const std::shared_ptr<SimulatorGaussianSDDP> &>())
    .def("getSimulatorBackward", &OptimizeDemandSDDP<SimulatorGaussianSDDP>::
```

```
        getSimulatorBackward)
    .def("getSimulatorForward", &OptimizeDemandSDDP<SimulatorGaussianSDDP>::
        getSimulatorForward)
    .def("oneAdmissibleState", &OptimizeDemandSDDP<SimulatorGaussianSDDP>::
        oneAdmissibleState)
    ;
}
```

The wrapper used to expose the SDDP simulator is given in `StOpt/test/c++/python/Pybind11Simulators.cpp`
Then it is possible to use the mapping to write a Python version of `testDemandSDDP.cpp`

```
# 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

# unitest 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) / 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";
        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


# unitest 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)

```

```
122         # optimisation dates
123         dates = NP.linspace( 0., maturity,nstep + 1);
124 
125         # names for archive
126         nameRegressor = "RegressorDemand";
127         nameCut = "CutDemand";
128         nameVisitedStates = "VisitedStateDemand";
129 
130         #  precision parameter
131         nIterMax = 40
132         accuracyClose =  1.
133         accuracy = accuracyClose / 100.
134         nstepIterations = 4; # check for convergence between nstepIterations step
135 
136         values = bfSDDP.backwardForwardSDDP(optimizer,  p_sampleCheckSimul, initialState,
                finCut, dates,  nbMesh, nameRegressor,
137                                     nameCut, nameVisitedStates, nIterMax,
138                                     accuracy, nstepIterations);
139 
140         return values
141 
142 
143 class testDemandSDDP(unittest.TestCase):
144     def testDemandSDDP1D(self):
145         try:
146             imp.find_module('mpi4py')
147             found = True
148         except:
149             print("Not parallel module found ")
150             found = False
151 
152         if found :
153             from mpi4py import MPI
154             world = MPI.COMM_WORLD
155 
156             sigD = 0.6 ;
157             sampleOptim = 500;
158             sampleCheckSimul = 500;
159 
160             values = demandSDDPFunc(sigD, sampleOptim ,sampleCheckSimul)
161 
162             if (world.rank==0):
163                     print("Values is ",values)
164 
165     def testDemandSDDP1DLowLevel(self):
166         sigD = 0.6 ;
167         sampleOptim = 500;
168         sampleCheckSimul = 500;
169         demandSDDPFuncLowLevel(sigD, sampleOptim ,sampleCheckSimul)
170 
171 
172 if __name__ == '__main__':
173     unittest.main()
```

Part VII

Nesting Monte Carlo for general non linear PDEs

The method described is has been studied in [51], [50] and is using some ideas in [22], [49].
Our goal is to solve the general full non linear equation

$$\begin{aligned}(-\partial_t u - \mathcal{L}u)(t,x) &= f(t,x,u(t,x),Du(t,x),D^2u(t,x)),\\ u_T &= g, \quad t<T,\ x\in\mathbb{R}^d,\end{aligned} \tag{14.23}$$

with

$$\mathcal{L}u(t,x) := \mu Du(t,x) + \frac{1}{2}\sigma\sigma^\top : D^2u(t,x)$$

so that $\mathcal{L}$ is the generator associated 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, ρ is the density of a general random variable following a gamma law so that

$$\rho(x) = \lambda^\alpha x^{\alpha-1}\frac{e^{-\lambda x}}{\Gamma(\alpha)}, 1\geq\alpha>0. \tag{14.24}$$

The associated cumulative distribution function is

$$F(x) = \frac{\gamma(\alpha,\lambda x)}{\Gamma(\alpha)}$$

where $\gamma(s,x)=\int_0^x t^{s-1}e^{-t}dt$ is the incomplete gamma function and $\Gamma(s)=\int_0^\infty t^{s-1}e^{-t}dt$ is the gamma function.
The methodology follows the ideas of [51] and [49].
We suppose here that σ is non degenerated so that σ^{-1} exists.
Let set $p\in\mathbb{N}^+$. For $(N_0,\dots,N_{p-1})\in\mathbb{N}^p$, we introduce the sets of i-tuple, $Q_i=\{k=(k_1,\dots,k_i)\}$ for $i\in\{1,\dots,p\}$ where all components $k_j\in[1,N_{j-1}]$. Besides we define $Q^p=\cup_{i=1}^p Q_i$.
We construct the sets Q_i^o for $i=1,\dots,p$, such that

$$Q_1^o = Q_1$$

and the set Q_i^o for $i>1$ are defined by recurrence:

$$Q_{i+1}^o = \{(k_1,\dots,k_i,k_{i+1})/(k_1,\dots,k_i)\in Q_i^o, k_{i+1}\in\{1,\dots,N_{i+1},1_1,\dots,(N_{i+1})_1,1_2,\dots,(N_{i+1})_2\}\}$$

so that to a particle noted $(k_1,\dots,k_i)\in Q_i^o$ such that $k_i\in\mathbb{N}$, we associate two fictitious particles noted $k^1=(k_1,\dots,k_{i-1},(k_i)_1)$ and $k^2=(k_1,\dots,k_{i-1},(k_i)_2)$.
To a particle $k=(k_1,\dots,k_i)\in Q_i^o$ we associate its original particle $o(k)\in Q_i$ such that $o(k)=(\hat{k}_1,\dots,\hat{k}_i)$ where $\hat{k}_j=l$ if $k_j=l$, l_1 or l_2.
For $k=(k_1,\dots,k_i)\in Q_i^o$ we introduce the set of its non fictitious sons

$$\tilde{Q}(k) = \{l=(k_1,\dots,k_i,m)/m\in\{1,\dots,N_i\}\}\subset Q_{i+1}^o,$$

and the set of all sons

$$\hat{Q}(k) = \{l = (k_1, \ldots, k_i, m)/m \in \{1, \ldots, N_i, 1_1, \ldots, (N_i)_1, 1_2, \ldots, (N_i)_2\}\} \subset Q^o_{i+1}.$$

By convention $\tilde{Q}(\emptyset) = \{l = (m)/m \in \{1, \ldots, N_0\}\} = Q_1$. Reciprocally the ancestor k of a particle $\tilde{k}$ in $\tilde{Q}(k)$ is noted $\tilde{k}^-$.
We define the order of a particle $k \in Q^o_i$, $i \geq 0$, by the function κ:

$$\begin{aligned}\kappa(k) =& 0 \text{ for } k_i \in \mathbb{N},\\ \kappa(k) =& 1 \text{ for } k_i = l_1, l \in \mathbb{N}\\ \kappa(k) =& 2 \text{ for } k_i = l_2, l \in \mathbb{N}\end{aligned}$$

We define the sequence τ_k of switching increments i.i.d. random variables with density ρ for $k \in Q^p$. The switching dates are defined as:

$$\left\{ \begin{array}{lcl} T_{(j)} & = & \tau_{(j)} \wedge T, j \in \{1, ., N_0\} \\ T_{\tilde{k}} & = & (T_k + \tau_{\tilde{k}}) \wedge T, k = (k_1, \ldots, k_i) \in Q_i, \tilde{k} \in \tilde{Q}(k) \end{array} \right. \tag{14.25}$$

By convention $T_k = T_{o(k)}$ and $\tau_k = \tau_{o(k)}$. For $k = (k_1, \ldots, k_i) \in Q^o_i$ and $\tilde{k} = (k_1, \ldots, k_i, k_{i+1}) \in \hat{Q}(k)$ we define the following trajectories:

$$W^{\tilde{k}}_s := W^k_{T_k} + \mathbf{1}_{\kappa(\tilde{k})=0} \bar{W}^{o(\tilde{k})}_{s-T_k} - \mathbf{1}_{\kappa(\tilde{k})=1} \bar{W}^{o(\tilde{k})}_{s-T_k}, \quad \text{and} \tag{14.26}$$

$$X^{\tilde{k}}_s := x + \mu s + \sigma W^{\tilde{k}}_s, \quad \forall s \in [T_k, T_{\tilde{k}}], \tag{14.27}$$

where the $\bar{W}^k$ for k in Q^p are independent d-dimensional Brownian motions, independent of the $(\tau_k)_{k \in Q^p}$.
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 (14.26),

$$\begin{aligned} X^{(1,1,1)}_T =& \bar{W}^{(1)}_{T_{(1)}} + \bar{W}^{(1,1)}_{T_{(1,1)} - T_{(1)}} + \bar{W}^{(1,1,1)}_{T - T_{(1,1)}} \\ X^{(1_1,1,1)}_T =& - \bar{W}^{(1)}_{T_{(1)}} + \bar{W}^{(1,1)}_{T_{(1,1)} - T_{(1)}} + \bar{W}^{(1,1,1)}_{T - T_{(1,1)}} \\ X^{(1,1_1,1)} =& \bar{W}^{(1)}_{T_{(1)}} - \bar{W}^{(1,1)}_{T_{(1,1)} - T_{(1)}} + \bar{W}^{(1,1,1)}_{T - T_{(1,1)}} \\ X^{(1_2,1_1,1)}_T =& - \bar{W}^{(1,1)}_{T_{(1,1)} - T_{(1)}} + \bar{W}^{(1,1,1)}_{T - T_{(1,1)}} \\ & \ldots \end{aligned}$$

such that all particles are generated from the $\bar{W}^k$ used to define $X^{(1,1,1)}_T$.

Using the previous definitions, we consider the estimator defined by:

$$
\begin{cases}
\bar{u}^p_{\emptyset} = \dfrac{1}{N_0}\displaystyle\sum_{j=1}^{N_0} \phi\big(0, T_{(j)}, X^{(j)}_{T_{(j)}}, \bar{u}^p_{(j)}, D\bar{u}^p_{(j)}, D^2\bar{u}^p_{(j)}\big), \\
\bar{u}^p_k = \dfrac{1}{N_i}\displaystyle\sum_{\tilde{k}\in\tilde{Q}(k)} \dfrac{1}{2}\big(\phi\big(T_k, T_{\tilde{k}}, X^{\tilde{k}}_{T_{\tilde{k}}}, \bar{u}^p_{\tilde{k}}, D\bar{u}^p_{\tilde{k}}, D^2\bar{u}^p_{\tilde{k}}\big) + \\
\qquad \phi\big(T_k, T_{\tilde{k}}, X^{\tilde{k}^1}_{T_{\tilde{k}}}, \bar{u}^p_{\tilde{k}^1} D\bar{u}^p_{\tilde{k}^1}, D^2\bar{u}^p_{\tilde{k}^1}\big)\big), \quad \text{for } k=(k_1,\dots,k_i)\in Q^o_i, 0<i<p, \\
D\bar{u}^p_k = \dfrac{1}{N_i}\displaystyle\sum_{\tilde{k}\in\tilde{Q}(k)} \mathbb{V}^{\tilde{k}}\dfrac{1}{2}\big(\phi\big(T_k, T_{\tilde{k}}, X^{\tilde{k}}_{T_{\tilde{k}}}, \bar{u}^p_{\tilde{k}}, D\bar{u}^p_{\tilde{k}}, D^2\bar{u}^p_{\tilde{k}}\big) - \\
\qquad \phi\big(T_k, T_{\tilde{k}}, X^{\tilde{k}^1}_{T_{\tilde{k}}}, \bar{u}^p_{\tilde{k}^1} D\bar{u}^p_{\tilde{k}^1}, D^2\bar{u}^p_{\tilde{k}^1}\big)\big), \quad \text{for } k=(k_1,\dots,k_i)\in Q^o_i, 0<i<p, \\
D^2\bar{u}^p_k = \dfrac{1}{N_i}\displaystyle\sum_{\tilde{k}\in\tilde{Q}(k)} \mathbb{W}^{\tilde{k}}\dfrac{1}{2}\big(\phi\big(T_k, T_{\tilde{k}}, X^{\tilde{k}}_{T_{\tilde{k}}}, \bar{u}^p_{\tilde{k}}, D\bar{u}^p_{\tilde{k}}, D^2\bar{u}^p_{\tilde{k}}\big) + \\
\qquad \phi\big(T_k, T_{\tilde{k}}, X^{\tilde{k}^1}_{T_{\tilde{k}}}, \bar{u}^p_{\tilde{k}^1}, D\bar{u}^p_{\tilde{k}^1}, D^2\bar{u}^p_{\tilde{k}^1}\big) - \\
\qquad 2\phi\big(T_k, T_{\tilde{k}}, X^{\tilde{k}^2}_{T_{\tilde{k}}}, \bar{u}^p_{\tilde{k}^2}, D\bar{u}^p_{\tilde{k}^2}, D^2\bar{u}^p_{\tilde{k}^2}\big)\big), \quad \text{for } k=(k_1,\dots,k_i)\in Q^o_i, 0<i<p, \\
\bar{u}^p_k = g(X^k_{T_k}), \quad \text{for } k\in Q^o_p, \\
D\bar{u}^p_k = Dg(X^k_{T_k}), \quad \text{for } \tilde{k}\in Q^o_p, \\
D^2\bar{u}^p_k = D^2 g(X^k_{T_k}), \quad \text{for } \tilde{k}\in Q^o_p
\end{cases}
\tag{14.28}
$$

where ϕ is defined by:

$$\phi(s,t,x,y,z,\theta) := \frac{\mathbf{1}_{\{t\geq T\}}}{\overline{F}(T-s)}g(x)+\frac{\mathbf{1}_{\{t<T\}}}{\rho(t-s)}f(t,x,y,z,\theta). \tag{14.29}$$

and

$$\mathbb{V}^k = \sigma^{-\top}\frac{\bar{W}^k_{T_k-T_{k^-}}}{T_k-T_{k^-}}$$

,

$$\mathbb{W}^k = (\sigma^\top)^{-1}\frac{\bar{W}^k_{T_k-T_{k^-}}(\bar{W}^k_{T_k-T_{k^-}})^\top-(T_k-T_{k^-})I_d}{(T_k-T_{k^-})^2}\sigma^{-1} \tag{14.30}$$

As explained before, the u and Du term in f are treated as explained in [51] and only the D^2u treatment is the novelty of this scheme.

Remark 34 *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^o replacing ϕ by g and taking some value for N_{p+1}:*

$$\begin{aligned}
\bar{u}_k^p =& \frac{1}{N_{p+1}}\sum_{\tilde{k}\in\tilde{Q}(k)}\frac{1}{2}\big(g\big(X^{\tilde{k}}_{T_{\tilde{k}}}\big)+g\big(X^{\tilde{k}^1}_{T_{\tilde{k}}}\big)\big),\\
D\bar{u}_k^p =& \frac{1}{N_{p+1}}\sum_{\tilde{k}\in\tilde{Q}(k)}\mathbb{V}^{\tilde{k}}\frac{1}{2}\big(g\big(X^{\tilde{k}}_{T_{\tilde{k}}}\big)-g\big(X^{\tilde{k}^1}_{T_{\tilde{k}}},\big)\big),\\
D^2\bar{u}_k^p =& \frac{1}{N_{p+1}}\sum_{\tilde{k}\in\tilde{Q}(k)}\mathbb{W}^{\tilde{k}}\frac{1}{2}\big(g\big(X^{\tilde{k}}_{T_{\tilde{k}}},\big)+g\big(X^{\tilde{k}^1}_{T_{\tilde{k}}}\big)-2g\big(X^{\tilde{k}^2}_{T_{\tilde{k}}}\big)\big)
\end{aligned}$$

Remark 35 *In the case where the coefficient are not constant, some Euler scheme can be added as explained in [51].*

An effective algorithm for this scheme is given these two functions:

Algorithm 18 Outer Monte Carlo algorithm (V generates unit Gaussian RV, $\tilde{V}$ generates RV with gamma law density)

procedure PDEEVAL(μ, σ, g, f, T, p, x_0, $\{N_0,\dots,N_{p+1}\}$, V, $\tilde{V}$)
  $u_M = 0$
  $x(0,:) = x_0(:)$ ▷ x is a matrix of size $1\times n$
  for $i = 1, N_0$ **do**
    $(u, Du, D^2u) = \text{EvalUDUD2U}(x_0,\mu,\sigma,g,T,V,\tilde{V},p,1,0,0)$
    $u_M = u_M + u(0)$
  end for
return $\frac{U_M}{N_0}$
end procedure

Algorithm 19 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.

procedure EvalUDUD2U($x, \mu, \sigma, g, T, V, \tilde{V}, p, m, t, l$)
  $\tau = \min(\tilde{V}(), T - t)$, ▷ Sample the time step
  $G = V()$ ▷ Sample the n dimensional Gaussian vector
  $xS(1 : m, :) = x(:) + \mu\tau + \sigma G\sqrt{\tau}$
  $xS(m + 1 : 2m, :) = x(:) + \mu\tau$
  $xS(2m + 1 : 3m, :) = x(:) + \mu\tau - \sigma G\sqrt{\tau}$
  $tS = t + \tau$ ▷ New date
  if $ts \geq T$ or $l = p$ **then**
    $g_1 = g(xS(1 : m, :)); g_2 = (xS(m + 1 : 2m, :)); g_3 = g(xS(2m + 1 : 3m, :))$
    $u(:) = \frac{1}{2}(g_1 + g_3)$
    $Du(:, :) = \frac{1}{2}(g_1 - g_3) \quad \sigma^{-\top} G$
    $D^2u(:, :, :) = \frac{1}{2}(g_1 + g_3 - 2g_2)\sigma^{-\top}\frac{GG^\top - \mathbf{I}_d}{\tau}\sigma^{-1}$
    if $l \neq p$ **then**
      $(u(:), Du(:, :), D^2u(:, :, :))/ = \frac{1}{F(\tau)}$
    end if
  else
    $y(:) = 0; z(:, :) = 0; \theta(:, :, :) = 0$
    for $j = 1, N_{l+1}$ **do**
      $(y, z, \theta)+ =$EvalUDUD2U$(xS, \mu, \sigma, g, T, V, \tilde{V}, p, 3m, tS, l + 1)$
    end for
    $(y, z, \theta)/ = N_{l+1}$
    for $q = 1, m$ **do**
      $f_1 = f(ts, xS(q), y(q), z(q, :), \theta(q, :, :))$
      $f_2 = f(ts, xS(m + q), y(m + q), z(m + q, :), \theta(m + q, :, :))$
      $f_3 = f(ts, xS(2m + q), y(2m + q), z(2m + q, :), \theta(2m + q, :, :))$
      $u(i) = \frac{1}{2}(f_1 + f_3)$
      $Du(i, :) = \frac{1}{2}(f_1 - f_3)\sigma^{-\top} G$
      $D^2u(i, :, :) = \frac{1}{2}(f_1 + f_3 - 2f_2)\sigma^{-\top}\frac{GG^\top - \mathbf{I}_d}{\tau}\sigma^{-1}$
    end for
  end if
return (u, Du, D^2u)
end procedure

Part VIII

Some test cases description

Chapter 15

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.

15.1 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].

15.1.1 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 (`LocalSameSizeConstRegressio` 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.

15.1.2 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.

15.1.3 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.

15.1.4 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.

15.1.5 testAmericanOptionTree

Simple test case in 1D, to test the tree method on American options. An Ornstein–Uhlenbeck process using a near to zero mean reverting parameter is used to be near the BS model. The OU model is approximated by a trinomial tree.

15.2 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 [25] for description of the swing problem and the backward resolution techniques. Due to classical results on the Snell envelop 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 6) 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 [25] 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 9.2.2. 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.

15.2.1 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 $B = 2A$.

15.2.2 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.

15.2.3 testSwingOptimSimu / testSwingOptimSimuMpi

This test case takes the problem described in section 15.2, solves it using the framework 9.2.2. 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.

15.2.4 testSwingOptimSimuWithHedge

The test case takes the problem described in section 15.2, solves it using regression (`LocalLinearRegressio` regressor) while calculating the optimal hedge by the conditional tangent method as explained in [45]. 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 9.2.2 can be used to calculate and store the optimal hedge. This is achieved by the creation of a dedicated optimizer `OptimizeSwingWithHedge`.

15.2.5 testSwingOptimSimuND / testSwingOptimSimuNDMpi

The test case takes the problem described in section 15.2, 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 15.2.1. 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.

15.3 Gas Storage

15.3.1 testGasStorage / testGasStorageMpi

The model used is a mean reverting model similar to the one described in [45]. We keep only one factor in equation (8) in [45]. The problem consists in maximizing the gain from a gas storage by the methodology described in [45]. 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 [47], 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.

15.3.2 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 9.2.3) 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 conditional cuts 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.

15.3.3 testGasStorageTree/testGasStorageTreeMpi

Optimize a storage for gas price modeled by an HJM model approximated by a tree. The grids are simple regular grids. In MPI, Bellman values are either stored in one file or multiple files.

15.3.4 testGasStorageTreeCut/testGasStorageTreeCutMpi

Gas storage is optimized and simulated using cuts and tree for uncertainties. Gas price modeled by an HJM model approximated by a trinomial tree The grids are simple regular grids. In MPI, Bellman values are either stored in one file or multiple files.

15.3.5 testGasStorageKernel

The model used is a mean reverting model similar to the one described in [45]. We keep only one factor in equation (8) in [45].The problem consists in maximizing the gain from a gas storage by the methodology described in [45]. The specificity here is that a kernel regression method is used.

testSimpleStorageKernel

Use the linear kernel regression method to solve the Gas Storage problem.

15.3.6 testGasStorageVaryingCavity

The stochastic model is the same as in section 15.3.1. 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.

15.3.7 testGasStorageSwitchingCostMpi

The test case is similar to the one in section 15.3.1 (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.

15.3.8 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 14) 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 $N = 10$.

15.3.9 testGasStorageSDDPTree

testSimpleStorageDeterministicCutTree

The volatility is set to zero to get a deterministic problem.

- First by Dynamic Programming, the optimal control is calculated and tested in simulation.
- Then backward part of SDDP and forward part are tested using a grid of point for the storage

testSimpleStorageCutTree

In stochastic, the backward and forward resolution of the SDDP solver with tree are tested using points defined on a grid. Convergence is checked by comparing results coming from a DP solver with regressions.

testSimpleStorageSDDPTree1D1Step

In stochastic, the global SDDP solver iterating forward and backward is used to valuate the gas storage. Comparison with dynamic programming methods with regressions is achieved.

15.4 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.

15.5 testOptionNIGL2

In this test case we suppose that the log of an asset value follows an NIG process [3]. We want to price a call option supposing that we use the mean variance criterion using the algorithm developed in chapter 11.
First an optimization is achieved then in a simulation part the optimal hedging strategy is tested.

15.6 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, σ_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

$$\begin{aligned} V &= P\mathbb{E}\left[\sum_{i=0}^{N} D_i\right] \\ &= (N+1)D_0 P \end{aligned}$$

This can be done (artificially) using SDDP.

testDemandSDDP1DDeterministic

It takes $\sigma_d = 0$.

testDemandSDDP1D

It solves the stochastic problem.

15.7 Reservoir variations with SDDP

15.7.1 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 σ_i. We suppose that we have to satisfy at M dates a demand given by independent centered Gaussian variables with standard deviation σ_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:

$$\mathbb{E}\left[\sum_{i=0}^{M} q_t S_t\right]$$

Each time we check that forward and backward methods converge to the same value. Because of the independence of uncertainties the dimension of the Markov state is equal to N.

testSimpleStorageWithInflowsSDDP1DDeterminist

$\sigma_i = 0$ for inflows and $\sigma_d = 0.$ for demand. N taken equal to 1.

testSimpleStorageWithInflowsSDDP2DDeterminist

$\sigma_i = 0$ for inflows and $\sigma_d = 0.$ for demand. N taken equal to 2.

testSimpleStorageWithInflowsSDDP5DDeterminist

$\sigma_i = 0$ for inflows and $\sigma_d = 0.$ for demand. N taken equal to 5.

testSimpleStorageWithInflowsSDDP1D

$\sigma_i = 0.6$, $\sigma_d = 0.8$ for demand. $N = 1$

testSimpleStorageWithInflowsSDDP2D

$\sigma_i = 0.6$ for inflows, $\sigma_d = 0.8$ for demand. $N = 2$

testSimpleStorageWithInflowsSDDPD

$\sigma_i = 0.6$ for inflows, $\sigma_d = 0.8$ for demand. $N = 5$.

15.7.2 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 $X^0 = X$, σ 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).

testSimpleStorageWithInflowsSDDP1DDeterministic

All parameters σ are set to 0. $N = 1$.

testSimpleStorageWithInflowsSDDP2DDeterministic

All parameters σ are set to 0. $N = 2$.

testSimpleStorageWithInflowsSDDP5DDeterministic

All parameters σ are set to 0. $N = 5$.

testSimpleStorageWithInflowsSDDP10DDeterministic

All parameters σ are set to 0. $N = 10$.

testSimpleStorageWithInflowsSDDP1D

$\sigma = 0.3$ for inflows, $\sigma = 0.4$ for demand. $N = 1$.

testSimpleStorageWithInflowsSDDP5D

$\sigma = 0.3$ for inflows, $\sigma = 0.4$ for demand. $N = 5$.

15.7.3 testStorageWithInflowsAndMarketSDDP

This is the same problem as 15.7.2, 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. $N = 5$.

15.8 Semi-Lagrangian

15.8.1 testSemiLagrangCase1/testSemiLagrangCase1

Test Semi-Lagrangian deterministic methods for HJB equation. This corresponds to the second test case without control in [47] (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.

15.8.2 testSemiLagrangCase2/testSemiLagrangCase2

Test Semi-Lagrangian deterministic methods for HJB equation. This corresponds to the first case without control in [47] (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.

15.8.3 testSemiLagrangCase2/testSemiLagrangCase2

Test Semi-Lagrangian deterministic methods for HJB equation. This corresponds to the stochastic target test case 5.3.4 in [47].

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.

15.9 Non emimissive test case

15.9.1 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.

15.9.2 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.

15.10 Nesting for Non Linear PDE's

15.10.1 Some HJB test

The control problem where $\mathcal{A}$ is the set of adapted integrable processes.

$$dX = 2\sqrt{\theta}\alpha dt + \sqrt{2}dW_t,$$
$$V = \inf_{\alpha\in\mathcal{A}} E[\int_0^T |\alpha_s|^2 dt + g(X_T)]$$

The HJB equation corresponding

$$(-\partial_t u - \mathcal{L}u)(t,x) = f(Du(t,x))$$

$$\mathcal{L}u(t,x) := \mu Du(t,x) + \frac{1}{2}\sigma\sigma^\top : D^2u(t,x), \tag{15.1}$$
$$f(z) = -\theta||z||_2^2 \tag{15.2}$$

such that a solution is

$$u(t,x) = -\frac{1}{\theta}\log\left(\mathbb{E}[e^{-\theta g(x+\sqrt{2}W_{T-t})}]\right). \tag{15.3}$$

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.

testHJCExact

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.

testHJBEuler

We use a resolution function supposing that the SDE is discretized by an Euler scheme.

15.10.2 Some Toy example: testUD2UTou

We want to solve:

$$(-\partial_t u - \mathcal{L}u)(t, x) = f(u, Du(t, x), D^2u(t, x))$$

with

$$\begin{array}{ll}
\mu = & \frac{\mu_0}{d}\mathbb{I}_d, \\
\sigma = & \frac{\sigma_0}{\sqrt{d}}\mathbf{I}_d, \\
f(t,x,y,z,\theta) = & \cos(\sum_{i=1}^d x_i)(\alpha + \frac{1}{2}\sigma_0^2)e^{\alpha(T-t)} + \sin(\sum_{i=1}^d x_i)\mu_0 e^{\alpha(T-t)} + a\sqrt{d}\cos(\sum_{i=1}^d x_i)^2 e^{2\alpha(T-t)} \\
& + \frac{a}{\sqrt{d}}(-e^{2\alpha(T-t)}) \vee (e^{2\alpha(T-t)} \wedge (y\sum_{i=1}^d \theta_{i,i})),
\end{array}$$

with a solution

$$u(t, x) = e^{\alpha(T-t)}\cos(\sum_{i=1}^d x_i)$$

15.10.3 Some Portfolio optimization

We assume that we dispose of $d = 4$ securities all of them being defined by a Heston model:

$$\begin{aligned}
dS_t^i &= \mu^i S_t^i dt + \sqrt{Y_t^i} S_t^i dW_t^{(2i-1)} \\
dY_t^i &= k^i(m^i - Y_t^i)dt + c^i\sqrt{Y_t^i}dW_t^{(2i)},
\end{aligned}$$

where $W = (W^{(1)}, \ldots, W^{(2d)})$ is a Brownian motion in $\mathbb{R}^{2d}$.
The non-risky asset S^0 has a 0 return so $dS_t^0 = 0$, $t \in [0, 1]$.
The investor chooses an adapted process $\{\kappa_t, t \in [0, T]\}$ with values in $\mathbb{R}^n$, where κ_t^i is the amount he decides to invest into asset i.
The portfolio dynamic is given by:

$$dX_t^\kappa = \kappa_t \cdot \frac{dS_t}{S_t} + (X_t^\kappa - \kappa_t \cdot \mathbf{1})\frac{dS_t^0}{S_t^0} = \kappa_t \cdot \frac{dS_t}{S_t}.$$

Let $\mathcal{A}$ be the collection of all adapted processes κ with values in $\mathbb{R}^d$ and which are integrable with respect to S. Given an absolute risk aversion coefficient $\eta > 0$, the portfolio optimization problem is defined by:

$$v_0 := \sup_{\kappa \in \mathcal{A}} \mathbb{E}\left[-\exp\left(-\eta X_T^\kappa\right)\right]. \tag{15.4}$$

The problem doesn't depend on the s^i. As in [52], we can guess that the solution can be expressed as

$$v(t,x,y^1,\dots,y^d) = e^{-\eta x}u(y^1,\dots,y^d)$$

and using Feyman Kac it is easy to see that then a general solution can be written

$$v(t,x,y) = -e^{-\eta x}\mathbb{E}[\prod_{i=1}^{d}\exp\left(-\frac{1}{2}\int_t^T \frac{(\mu^i)^2}{\tilde{Y}_s^i}ds\right)] \tag{15.5}$$

with

$$\tilde{Y}_t^i = y^i \quad \text{and} \qquad\qquad d\tilde{Y}_t^i = k^i(m^i - \tilde{Y}_t^i)dt + c^i\sqrt{\tilde{Y}_t^i}dW_t^i,$$

where y^i 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 (14.23) in dimension $d+1$ where

$$\mu = (0, k^1(m^1 - y^1), \dots, k^d(m^d - y^d))^\top, \qquad \sigma = \begin{pmatrix} \bar{\sigma} & 0 & \cdots & \cdots & 0 \\ 0 & c\sqrt{m^1} & 0 & \cdots & 0 \\ 0 & \cdots & \ddots & \cdots & 0 \\ 0 & \cdots & \cdots & \ddots & 0 \\ 0 & \cdots & \cdots & 0 & c\sqrt{m^d} \end{pmatrix}$$

always with the same terminal condition

$$g(x) = -e^{-\eta x}$$

and

$$f(x,y,z,\theta) = -\frac{1}{2}\bar{\sigma}^2\theta_{11} + \frac{1}{2}\sum_{i=1}^{d}(c^i)^2((y^i)^2 - m^i)\theta_{i+1,i+1} - \sum_{i=1}^{d}\frac{\mu^i z_1}{2y^i\theta_{11}}. \tag{15.6}$$

In order to have f Lipschitz, we truncate the control limiting the amount invested by taking

$$f_M(y,z,\theta) = -\frac{1}{2}\bar{\sigma}^2\theta_{11} + \frac{1}{2}\sum_{i=1}^{d}(c^i)^2((y^i)^2 - m^i)\theta_{2,2} + \sup_{\substack{\eta = (\eta^1,\dots,\eta^d) \\ 0 \le \eta^i \le M, i = 1,d}} \sum_{i=1}^{d}\left(\frac{1}{2}(\eta^i)^2 y^i\theta_{11} + (\eta^i)\mu^i z_1\right).$$

testPortfolioExact

The Ornstein–Uhlenbeck process used as a driving process is simulated exactly.

testPortfolioEuler

The Ornstein–Uhlenbeck process used as a driving process is simulated using an Euler scheme.

Chapter 16

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.

16.1 Microgrid Management

16.1.1 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 [[29], 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).

16.1.2 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 16.1.1. The user can switch between this and the previous setting by changing the parameter `controlType` in the `parameters.py` file.

16.2 Dynamic Emulation Algorithm (DEA)

16.2.1 testMicrogridDEA

In this section we discuss the implementation of the Dynamic Emulation Algorithm developed in [29]. In that paper the authors reformulate the stochastic control problem as an "iterative sequence of machine learning tasks". The philosophy of DEA is to combine together Regression Monte Carlo (RMC) with Design of Experiments. The algorithm has the following properties:

- The learning for the continuation value function at each step in the backward-iteration of the Dynamic Programming Equation is completely modularized. As a result, the user can seamlessly switch between different regression schemes (for example: adaptive local polynomial basis in -2D or -1D, multivariate kernel regression, etc.) across different time-steps;

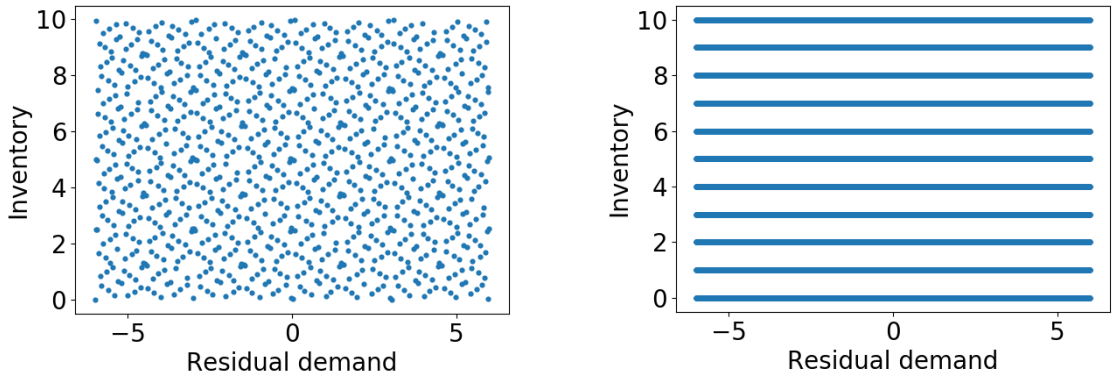
- The empirical approximation uses distinct designs $\mathcal{D}_t$ at each t; thus the user can have design sites independently chosen for different t's, which also eliminates the requirement to store the full history of the simulated paths of (X_t). One-step paths can now replace the full history. In Figure 16.1 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 [29] the authors discuss several further designs which can be easily implemented.

- Batched designs, i.e. a partially nested scheme that generates multiple X_t-paths from the same unique design site, can be accommodated.

- Simulation budget (i.e. the size of $\mathcal{D}_t$) can vary through the time-steps and need not be fixed as in standard RMC.

Several different experiments have confirmed the significant effect of the design $\mathcal{D}_t$ on the performance of the RMC algorithms. DEA allows us to test for this effect by allowing the user to easily specify $\mathcal{D}_t$. The structure of this library allows for easy implementation of such modular algorithms. As a proof of concept, we re-implement the microgrid example of Section 16.1.1 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 16.1b. Moreover, for these t's we used $|\mathcal{D}_t| = 22,000 = N_t$ unique design sites;



(a) Sobol-2D QMC sequence

(b) Gridded design

Figure 16.1: Illustration of two simulation designs. In both panels the X_t-coordinate is on the x-axis and I_t on the y-axis.

- The other 30 steps (first 75%) use design sites allocated according to Sobol-2D as in figure 16.1a with a global polynomial basis regression scheme. For these, we build a batched design of 1000 unique sites, each replicated 10 times for a total simulation budget of $N_t = 1000 \times 10 = 10^4$.

Bibliography

[1] Clemence Alasseur, Alessandro Balata, Sahar Ben Aziza, Aditya Maheshwari, Peter Tankov, and Xavier Warin. Regression monte carlo for microgrid management. arXiv preprint arXiv:1802.10352, 2018.

[2] Mejdi Azaïez, Monique Dauge, and Yvon Maday. Méthodes spectrales et des éléments spectraux. 1993.

[3] Ole E Barndorff-Nielsen. Processes of normal inverse gaussian type. Finance and stochastics, 2(1):41–68, 1997.

[4] Christian Bender and Robert Denk. A forward scheme for backward sdes. Stochastic processes and their applications, 117(12):1793–1812, 2007.

[5] JF Benders. Partitioning procedures for solving mixed-variables programming problems. Computational Management Science, 2(1):3–19, 2005.

[6] Eric Beutner. Mean–variance hedging under transaction costs. Mathematical Methods of Operations Research, 65(3):539–557, 2007.

[7] Bruno Bouchard, Xiaolu Tan, Xavier Warin, and Yiyi Zou. Numerical approximation of bsdes using local polynomial drivers and branching processes. Monte Carlo Methods and Applications, 23(4):241–263, 2017.

[8] Bruno Bouchard and Xavier Warin. Monte-carlo valuation of american options: facts and new algorithms to improve existing methods. In Numerical methods in finance, pages 215–255. Springer, 2012.

[9] Hans-Joachim Bungartz. Dünne Gitter und deren Anwendung bei der adaptiven Lösung der dreidimensionalen Poisson-Gleichung. Technische Universität München, 1992.

[10] Hans-Joachim Bungartz. Concepts for higher order finite elements on sparse grids. In Houston Journal of Mathematics: Proceedings of the 3rd Int. Conf. on Spectral and High Order Methods, Houston, pages 159–170, 1996.

[11] Hans-Joachim Bungartz. A multigrid algorithm for higher order finite elements on sparse grids. Electronic Transactions on Numerical Analysis, 6:63–77, 1997.

[12] Hans-Joachim Bungartz and Michael Griebel. Sparse grids. Acta numerica, 13:147–269, 2004.

[13] Fabio Camilli and Maurizio Falcone. An approximation scheme for the optimal control of diffusion processes. ESAIM: Mathematical Modelling and Numerical Analysis, 29(1):97–122, 1995.

[14] Robert P Feinerman and Donald J Newman. Polynomial approximation. 1974.

[15] Wendell H Fleming and Halil Mete Soner. Controlled Markov processes and viscosity solutions, volume 25. Springer Science & Business Media, 2006.

[16] Thomas Gerstner and Michael Griebel. Dimension–adaptive tensor–product quadrature. Computing, 71(1):65–87, 2003.

[17] Anders Gjelsvik, Michael M Belsnes, and Arne Haugstad. An algorithm for stochastic medium-term hydrothermal scheduling under spot price uncertainty. In Proceedings of 13th Power Systems Computation Conference, 1999.

[18] Emmanuel Gobet, Jean-Philippe Lemor, Xavier Warin, et al. A regression-based monte carlo method to solve backward stochastic differential equations. The Annals of Applied Probability, 15(3):2172–2202, 2005.

[19] Michael Griebel. Adaptive sparse grid multilevel methods for elliptic pdes based on finite differences. Computing, 61(2):151–179, 1998.

[20] Michael Griebel. Sparse grids and related approximation schemes for higher dimensional problems. Citeseer, 2005.

[21] Holger Heitsch and Werner Römisch. Scenario reduction algorithms in stochastic programming. Computational optimization and applications, 24(2-3):187–206, 2003.

[22] Pierre Henry-Labordere, Nadia Oudjane, Xiaolu Tan, Nizar Touzi, and Xavier Warin. Branching diffusion representation of semilinear pdes and monte carlo approximation. arXiv preprint arXiv:1603.01727, 2016.

[23] John C Hull. Options futures and other derivatives. Pearson Education India, 2003.

[24] Hitoshi Ishii and Pierre-Luis Lions. Viscosity solutions of fully nonlinear second-order elliptic partial differential equations. Journal of Differential equations, 83(1):26–78, 1990.

[25] Patrick Jaillet, Ehud I Ronn, and Stathis Tompaidis. Valuation of commodity-based swing options. Management science, 50(7):909–921, 2004.

[26] John D Jakeman and Stephen G Roberts. Local and dimension adaptive sparse grid interpolation and quadrature. arXiv preprint arXiv:1110.0010, 2011.

[27] Ali Koc and Soumyadip Ghosh. Optimal scenario tree reductions for the stochastic unit commitment problem. In Proceedings of the Winter Simulation Conference, page 10. Winter Simulation Conference, 2012.

[28] Nicolas Langrené and Xavier Warin. Fast and stable multivariate kernel density estimation by fast sum updating. arXiv preprint arXiv:1712.00993, 2017.

[29] Michael Ludkovski and Aditya Maheshwari. Simulation methods for stochastic storage problems: A statistical learning perspective. arXiv preprint arXiv:1803.11309, 2018.

[30] Xiang Ma and Nicholas Zabaras. An adaptive hierarchical sparse grid collocation algorithm for the solution of stochastic differential equations. Journal of Computational Physics, 228(8):3084–3113, 2009.

[31] Alessandro Magnani and Stephen P Boyd. Convex piecewise-linear fitting. Optimization and Engineering, 10(1):1–17, 2009.

[32] Constantinos Makassikis, Stéphane Vialle, and Xavier Warin. Large scale distribution of stochastic control algorithms for gas storage valuation. In Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on, pages 1–8. IEEE, 2008.

[33] M Motoczyński. Multidimensional variance-optimal hedging in discrete-time model - a general approach. Mathematical Finance, 10(2):243–257, 2000.

[34] Rémi Munos and Hasnaa Zidani. Consistency of a simple multidimensional scheme for hamilton–jacobi–bellman equations. Comptes Rendus Mathematique, 340(7):499–502, 2005.

[35] Mario Pereira, Nora Campodonico, and Rafael Kelman. Application of stochastic dual dp and extensions to hydrothermal scheduling. Online Rep., http://www. psr-inc. com. br/reports. asp, PSRI Technical Rep, 12:99, 1999.

[36] Mario VF Pereira and Leontina MVG Pinto. Multi-stage stochastic optimization applied to energy planning. Mathematical programming, 52(1-3):359–375, 1991.

[37] Laurent Pfeiffer, Romain Apparigliato, and Sophie Auchapt. Two methods of pruning Benders' cuts and their application to the management of a gas portfolio. PhD thesis, INRIA, 2012.

[38] Dirk Michael Pflüger. Spatially adaptive sparse grids for high-dimensional problems. PhD thesis, Technische Universität München, 2010.

[39] Alfio Maria Quarteroni, Riccardo Sacco, and Fausto Saleri. Méthodes numériques pour le calcul scientifique: programmes en MATLAB. Springer Science & Business Media, 2000.

[40] Martin Schweizer. Variance-optimal hedging in discrete time. Mathematics of Operations Research, 20(1):1–32, 1995.

[41] David W Scott. Multivariate density estimation: theory, practice, and visualization. John Wiley & Sons, 2015.

[42] Paolo M Soardi. Serie di Fourier in piu variabili, volume 26. Pitagora, 1984.

[43] 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.

[44] MP Wand. Fast computation of multivariate kernel estimators. Journal of Computational and Graphical Statistics, 3(4):433–445, 1994.

[45] Xavier Warin. Gas storage hedging. In Numerical Methods in Finance, pages 421–445. Springer, 2012.

[46] Xavier Warin. Adaptive sparse grids for time dependent hamilton-jacobi-bellman equations in stochastic control. arXiv preprint arXiv:1408.4267, 2014.

[47] Xavier Warin. Some non-monotone schemes for time dependent hamilton–jacobi–bellman equations in stochastic control. Journal of Scientific Computing, 66(3):1122–1147, 2016.

[48] Xavier Warin. Variance optimal hedging with application to electricity markets. arXiv preprint arXiv:1711.03733, 2017.

[49] Xavier Warin. Variations on branching methods for non linear pdes. arXiv preprint arXiv:1701.07660, 2017.

[50] Xavier Warin. Monte carlo for high-dimensional degenerated semi linear and full non linear pdes. arXiv preprint arXiv:1805.05078, 2018.

[51] Xavier Warin. Nesting monte carlo for high-dimensional non linear pdes. arXiv preprint arXiv:1804.08432, 2018.

[52] Thaleia Zariphopoulou. A solution approach to valuation with unhedgeable risks. Finance and stochastics, 5(1):61–82, 2001.