Master Ingénierie Mathématique (M2) — D5MA2J1 Informatique
Université Paris–Sud — Centre d'Orsay
AN-EDP — Année 2012 − 2013

© *JBAK <jean-baptiste.apoung@math.u-psud.fr>*

Projet informatique

Généralités

L'objectif de ce projet est le développement d'un outil de simulation des prix d'options du marché basés sur le modèle de Black&Scholes. Nous renvoyons à la nombreuse littérature pour les détails de cette modélisation, et nous focalisons sur l'implémentation dans le langage **C++** des équations différentielles stochastiques rattachées via la méthode de Monte-Carlo.

Le but visé par ce projet est la mise en oeuvre les acquis en programmation orientée objets. En particulier de bien cerner ce caractère adaptation du langage de programmation au domaine du problème à résoudre, qui se manifeste par la génération de nouveaux types et l'enrichissement de la syntaxe, ainsi qu'une utilisation efficace de la partie de la bibliothèque standard se rapportant au domaine du problème à résoudre. Tout ceci se fait bien souvent au détriment de l'efficacité d'exécution du code produit, mais laisse entrouvertes des portes pour une optimisation future.

- Nous allons dans un premier temps nous intéresser à la résolution approchée du modèle de Black&Scholes. Nous optons (notre filière l'oblige, ce cours est cette année destinée uniquement à la filière statistique du parcours) pour une approche stochastique.
- Nous nous intéresserons ensuite à un problème courant en finance, celui de la volatilité implicite. *Consistant, sous l'hypothèse de volatilité constante, d'estimer la volatilité pour un prix du sous-jacent et une maturité donnés.* Nous envisageons ici la construction d'un outil de résolution approchée d'une équation non-linéaire, ainsi que la transformation du simulateur de prix précédent pour satisfaire aux contraintes imposées par le solveur non-linéaire et permettre ainsi son emploie dans la recherchée d'une valeur approchée de la volatilité.
- Enfin, dans la limite du possible, nous bâtirons une interface graphique pour rendre le logiciel développé convivial d'utilisation par nos clients.

Chacune des étapes précédentes présente un certain nombre de difficultés à surmonter dans la conception :

- Dans la première, il sera question de fournir une conception qui permette l'ajout de nouveaux types (*Payoff*, *volatilité* etc.) sans modification de codes existants. Il sera aussi question de fournir un outil d'analyse permettant de réaliser des expériences statistiques sur la série des résultats obtenus.
- Dans la deuxième, il faudra fournir au moins deux algorithmes de résolution du problème non-linéaire (suivant que l'on dispose ou pas de la jacobienne) et un moyen de leur sélection dynamique sans modification de codes existants. L' outil de résolution du système non-linéaire devra permettre l'inversion de l'opérateur Black&Scholes, vu comme fonction de la seule volatilité supposée constante et la jacobienne dans cas est connue sous le nom de **VEGA**.
- Le dernier point concernant l'interface graphique a pour but de se familiariser avec une bibliothèque d'objets, ici d'objets graphiques et d'en faire un usage efficace et simpliste. On pourra par exemple solliciter la bibliothèque QT (voir `http://qt.nokia.com/products/`).

Rappelons cependant que les équations qui nous intéressent (modèle de Black&Scholes) admettent sous certaines hypothèses des solutions analytiques. Mais nous en faisons abstraction dans ce projet, en visant le développement des outils réutilisables dans des projets sans doute beaucoup plus complexes.

Dans la suite de ce document, la méthode de résolution sera présentée, suivie des besoins en classes **C++** pour la mise en oeuvre, accompagnés à chaque fois des moyens de validations des implémentations. Des bouts de codes utilitaires seront fournis en annexe.

Problème modèle et méthode de résolution

Le problème que l'on considère est celui de la prédiction du prix d'une option à l'achat (*call*) ou à la vente (*put*) sur une action ou un portefeuille d'actions. Pour une meilleure illustration, considérons le cas d'une option d'achat (*call*).
En effet, un individu A s'assure à l'instant $t = 0$ auprès d'un individu B, en payant une somme C_0 (prime d'option), le droit d'achat à la date future $t = T$ (date d'échéance ou *maturité*) et au prix K (prix d'exercice ou *strike*), une action donc le prix actuel est S_0 et ceci quelque soit le prix de cette action à la date T. Biensûre, si l'on note S_t le prix de l'action à toute date, le contrat autorise l'individu A à ne pas exercer son option si $S_T \leq K$.
Le problème posé pour l'individu A, est alors de déterminer la valeur de C_0 pour que l'on ait $C_T = \max(S_T - K, 0)$ (appelé *Payoff*), avec quelque fois le soucis de réaliser une meilleure affaire que celle (appelée 0-risque) consistant à déposer la même somme sur un compte rémunéré à un taux de r (taux d'intérêt). L'approche standard est alors d'estimer le profit à maturité ($\overline{(\max(S_T - K, 0))}^{\sharp}$) et de tenir compte de l'hypothèse du (0-risque) pour déduire le prix actuel ($C_0 = e^{-rT}\overline{(\max(S_T - K, 0))}^{\sharp}$).

Dans tous les cas, déterminer S_T est nécessaire et c'est là qu'intervient le modèle de Black-Scholes. Selon ce modèle le prix du sous-jacent (action) est solution de l'équation différentielle stochastique

$$dS_t = (r(t) - d(t))S_t dt + \sigma(t)S_t dW_t \tag{1}$$

où $r(t)$ est le taux d'intérêt, $d(t)$ la dividende, $\sigma(t)$ la volatilité (incertitude sur le comportement), et W_t est un mouvement brownien. Dans la plupart des applications, ces paramètres sont supposés constants : $r(t) \equiv r, d(t) \equiv d, \sigma(t) \equiv \sigma$ avec $d = 0$, (ce qui permet d'en déduire une solution analytique) et seule la solution à l'instant final S_T est recherchée. Le prix de l'option se déduit alors selon la formule

$$C_0 = e^{-rT}\mathbb{E}(f(S_T)), \tag{2}$$

où $\mathbb{E}$ désigne l'espérance mathématique (moyenne), et f est le Payoff défini suivant les cas comme suit

- $f(s) = \max(s - K, 0)$ pour le **Call**.
- $f(s) = \max(K - s, 0)$ pour le **Put**.
- $f(s) \equiv f = \max\left((\frac{1}{M+1}\sum_{i=0}^{M} S_{ti}) - K, 0\right)$ pour le **Call asiatique arithmétique** où les t_i sont les instants discrets de discrétisation de l'équation (1), avec $t_M = T$
- $f(s) \equiv f = \max\left((\prod_{i=0}^{M} S_{t_i})^{\frac{1}{M+1}} - K, 0\right)$ pour le **Call asiatique géométrique**.

On constate au regard des options asiatiques la nécessité de fournir les solutions de (1) à des instants donnés autres que l'instant final, faisant ainsi des valeurs de S aux instants $t_i, i = 0, \ldots, M$, un *processus stochastique* d'horizon M de valeur initiale S_0.

Pour discrétiser (1) et déterminer ses solutions aux instants $t_i, i = 0, \ldots, M$, on passe au $\log$ et on utilise la formule d'Îto. On obtient alors après intégration entre les instants t_i et t_{i+1} la formule

$$\log S_{t_{j+1}} = \log S_{t_j} + \int_{t_j}^{t_{j+1}} \left(r(s) - d(s) - \frac{1}{2}\sigma^2(s)\right) ds + \sqrt{\left(\int_{t_j}^{t_{j+1}} \sigma(s)^2 ds\right)} N_i, \tag{3}$$

où N_i est une réalisation de la loi normale centrée $\mathcal{N}(0, 1)$. Si l'on fait l'hypothèse des coefficients constants, (3) devient

$$\log S_{t_{j+1}} = \log S_{t_j} + \left(r - d - \frac{1}{2}\sigma^2\right)(t_{j+1} - t_j) + \sigma(W_{t_{j+1}} - W_{t_j}). \tag{4}$$

On en déduit la démarche de résolution suivante :

- On résout (4) ou (3) un certain nombre de fois, en générant pour chaque résolution une réalisation de la variable aléatoire $f(S_T)$ voir formule (2).
- On calcule alors la moyenne empirique de cette variable aléatoire.
- On multiplie le résultat par e^{-rT} pour obtenir le prix de l'option selon la formule (2).

Pour la mise en oeuvre informatique, nous assimilons S, C, W à des processus stochastiques, qui pour une donnée initiale et une suite d'instant sont capables de se dérouler tout seul jusqu'à l'horizon.

Nous aurons alors à définir les nouveaux types suivants :

1. **`VariableAleatoireNormale`** nécessaire pour générer le mouvement Brownien (ou processus de Wiener) W.
2. **`ProcessusStochastique`** qui sera un type de base pour les types **`MouvementBrownien`** (W), **`Action`** (S) et **`Option`** (P ou C).

Les détails sont fournis dans la suite. En guise d'illustration les courbes ci-dessous donnent l'évolution des prix d'options d'achat et de vente de certaines options en fonction du prix actuel de l'action.

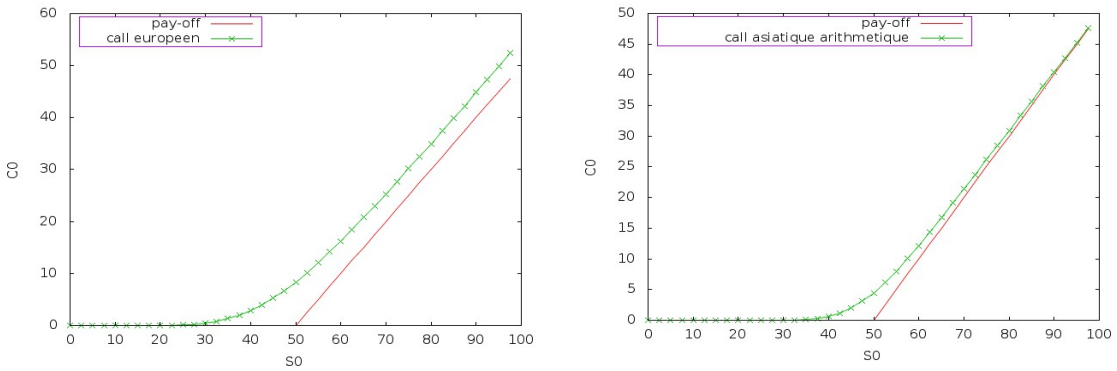

FIGURE 1 – Evolution en fonction du prix initial de l'action, d'une option européenne (à gauche) et une option asiatique arithmétique (à droite) pour des paramètres fournis dans le script Listing-1 ci-dessous.

Listing 1 – fichier :testCallAsiat.cpp

```
void test_call_asiatique(){
typedef unsigned long Int;
Timer time;
double K = 50., T=1.;
double r = 0.1, d = 0., sigma = 0.3;
Int nb_tirages_MonteCarlo = 50000;
time.tic();
Action action(S, T/12., 13);
action.setParams(r,d,sigma);
PayOffCall pK(K);
OptionAsiatique call(pK, action,nb_tirages_MonteCarlo);
Int horizon = call.horizon();
double dx = 2*K/ 40.;
std::ofstream os("eds_template_asiatique.dat");
for(double S0 = 0.; S0 < 2. * K; S0 += dx){
 call.nouveauChemin(S0);
 os<< S0 <<"  " << call.payOff(S0) <<"    "<< call[0] <<"\n";}
time.toc ("eds_template_asiatique");
std::cout<< time <<"\n";
}
```

Besoins pour la première partie

Besoin - 1 *Classes utilitaires à écrire*

Afin de mener à bien ce projet, un certain nombre de classes sont à écrire. **On prendra bien soin d'utiliser la compilation séparée lorsque c'est possible et de ne déclarer qu'une classe par fichier.** Un **`Makefile`** aussi complet que possible devra être utilisé.

Besoin-1-1 : **`class Tableau`**

On aura besoin d'une classe modélisant un tableau dynamique de réels. Les objets de ce type devront disposer des opérations arithmétiques usuelles sur les vecteurs. On devra en fournir une ou faire un usage du conteneur spécial **`std::valarray<double>`**, de la bibliothèque **`C++`**. On fournira un fichier **`Tableau.hpp`** et un fichier **`Tableau.cpp`**.

Besoin-1-2 : **`class VariableAleatoire`**

On définira une classe pour gérer les variables aléatoires. Les données membres seront telles qu'à chaque tirage une instance retourne directement un tableau de réalisations. La taille de ce tableau sera une donnée membre de la classe. La fonction membre correspondante sera **`void realisations(Tableau& v)const`**. Une copie de ce tirage sera stockée dans la classe afin de permettre des calculs statistiques telles que moyenne empirique et autres, ou pour une optimisation (antithétique).
Cette classe disposera entre autre des fonctions membres :

- **`VariableAleatoire(Int dim)`**, constructeurs (on respectera la règle des 5).
- **`void ajusterDimension(Int dim)`**, permettant d'ajustement ne nombre de réalisation à faire.
- **`virtual void seed(Int aseed)`**, permettant d'initialiser le générateur de nombre aléatoire.

Besoin-1-3 : **`class VariableAleatoireNormale`**

Cette classe définit un sous-type de VariableAleatoire et modélise la variable aléatoire normale centrée $\mathcal{N}(0,1)$. On pourra s'inspirer de la classe **`VariableAleatoireUniforme`** fournie en annexe, voir Listing-6.

Besoin-1-4 : **`class ProcessusStochastique`**

Cette classe permet de décrire tout processus stochastique, comme le prix des actions, le prix des options et le mouvement brownien unidimensionnel. Elle devra contenir :

- Deux tableaux : un pour les instants, et l'autre pour les valeurs du processus à ces instants. Ainsi que des fonctions d'accès associées.
- Un **`unsigned long`** définissant l'*horizon*.
- **`virtual void derouler()=0`** qui lance la simulation du processus. C'est-à-dire la génération à partir des données initiales, de toutes les valeurs du processus jusqu'à l'horizon.
- **`double pas(unsigned long i)const`** retournant la longueur du pas entre les instants i et $i+1$.
- L'opérateur surchargé **`<<`** pour afficher sur deux colonnes les instants et les valeurs associées.
- Deux constructeurs : l'un prenant la valeur initiale, le pas et le nombre d'instants et l'autre la valeur initiale et le tableau des instants. Le second contrairement au premier permettra de gérer des instants non nécessairement équi-distribués.

Besoin-1-5 : **`class MouvementBrownien`**

Cette classe dérive de la classe **`ProcessusStochastique`**, et implémente le mouvement brownien (processus de Wiener). Elle devra contenir en autres :

- Une donnée membre de type **`VariableAleatoireNormale`**.
- Et pour des besoins d'optimisation d'espace mémoire, un tableau de bonne taille pour l'appel de la fonction membre **`realisations`** de la classe **`VariableAleatoireNormale`**.

Besoin-1-6 : `class Action`

Cette classe modélise le prix d'une action (ou d'un produit sous-jacent). Elle dérive de la classe **`ProcessusStochastique`**, et contient (c'est un choix de modélisation lié au fait que nous assimilons l'action ou l'actif à son prix) :

- Un mouvement brownien
- Et les paramètres suivants :
  - Un réel identifiant le taux d'intérêt, **`double m_r`**
  - Un réel identifiant la dividende , **`double m_d`**
  - Un réel identifiant la volatilité , **`double m_sigma`**
- Une fonction membre **`void setInitial(double x)`** qui charge la valeur initiale de l'action.
- Une fonction membre **`void setParams(double r, double d, double sigma)`**.

Besoin-1-7 : `class PayOff`

Cette classe modélise le pay-Off. Ce sera une classe de base de type foncteur, reléguant à ses classes dérivées, qui seront **`PayOffCall, PayOffPut`**, le soin d'implémenter la fonction : **`virtual double eval()const =0;`**. Ainsi, l'implémentation de l'opérateur faisant du **`PayOff`** un foncteur aura la forme suivante :

Listing 2 – fichier :PayOff.cpp

```
double PayOff::operator()(double s) const{ return this->eval(s);}
```

Note : *il serait judicieux d'implémenter la classe* ***`PayOff`*** *via le design pattern* **Pont**. *Dans la limite du possible, justifier ce bien fondée et le mettre en application. Un bonus sera octroyé pour la réponse à cette question. On justifiera la nécessité du constructeur virtuel via le clônage.*

Besoin-1-8 : `class Option`

Cette classe modélisera le prix du produit dérivé. Elle dérivera de la classe **`ProcessusStochastique`**, et contiendra en données membres :

- Un pointeur ou référence sur le **`PayOff`**. *On critiquera un stockage par valeur.*
- Une référence non constante (**`Action& m_action`**) sur l'action dont elle est rattachée.
- Une fonction membre **`void nouveauChemin(double s)`** qui déroulera le processus à partir d'une nouvelle valeur initiale s. (ici valeur initiale du prix de l'action). Il est clair qu'une nouvelle simulation de l'action sera nécessaire.
- Un constructeur prenant en entrée, une référence constante au PayOff (*on justifiera la nécessité de cette référence constante*) une référence non constante à un objet de type **`Action`** et le nombre de tirage qu'il faut effectuer pour approcher la moyenne.

Besoin-1-9 : `class OptionAsiatique`

On fournira cette classe pour simuler les options asiatiques. On proposera une implémentation pour gérer les options asiatiques de type **arithmétique** et de type **géométrique**. **Note :** *On peut utiliser le design pattern* ***`Etat`*** *pour gérer cette variation.*

Validations pour la première partie

Pour valider les développements ci-dessus, il faudra fournir des fichiers tests. Un exemple de fichier test est fourni en annexe (voir Listing- 8).

Besoin-1-10 : `Un fichier test pour évaluer les` *Put* `et` *Call* `européens`

Besoin-1-11 : `Un fichier test pour évaluer les` *Call* `asiatiques avec moyennes arithmétique et géométriques`

.

Besoins pour la deuxième partie

Besoin - 2 *Classes utilitaires pour la Volatilité Implicite*

La simulation de la volatilité implicite nécessite la résolution d'une équation non-linéaire. On peut alors faire usage de la méthode de **bissection** ou de **Quasi-Newton** avec évaluation exacte ou approchée de la dérivée. Par soucis de *réutilisabilité*, on fournira un *solveur* générique d'équation non-linéaires. On devra alors modifier le simulateur Black-Scholes précédemment construit pour respecter les concepts exigés par le solveur non-linéaire, afin d'utiliser ce dernier dans des simulations de la volatilité implicite. Il est alors nécessaire de fournir les classes suivantes :

Besoin-2-1 : **`class ParametresSolveurNonLineaire`**
Cette classe permettra de fournir des informations au solveur, comme le nombre maximum d'itérations, la tolérance. Elle devra stocker le résidu (sa norme) et la valeur de l'itération courante. Puisqu'elle stocke ces informations, il lui est possible d'informer le solveur de la nécessité ou pas de poursuivre la résolution, par le biais des fonctions membres

- **`bool estTermine()const;`** signalant que la résolution doit s'interrompre.
- **`bool aConverge()const;`** signalant que l'interruption est due à la convergence de l'algorithme.

Besoin-2-2 : **`template<typename Equation, typename ParamSolveur>`**
`class SolveurNonLineaire`
Cette classe est la classe du solveur à proprement dit. Elle est paramétrée par le type **`Equation`**, qui désigne la classe encapsulant l'équation à résoudre et le type **`ParamSolveur`** qui décrit le type des paramètres du solveur tels que définis précédemment. **Note :** *Le type* ***`Equation`*** *doit satisfaire les contraintes suivantes :*

- *disposer d'une fonction membre constante définissant l'équation à résoudre sous la forme* $f(x) = 0$ *(la fonction ne s'appellera pas forcément f).*
- *disposer éventuellement d'une fonction membre constante définissant la dérivée (jacobienne) de l'équation, ou son approximation par différences finies.*

La classe **`SolveurNonLineaire`** devra disposer des fonctions suivantes :

- un constructeur qui prend en argument une instance de la classe de l'équation et des pointeurs vers les fonctions membre définissant l'équation à résoudre et éventuellement sa dérivée ou une approximation par différence finie.
- une fonction **`void newton(double& x, double cible, ParamSolveur& param, const std::function<void (const ParamSolveur&, double)>& prox)const;`** qui résout l'équation par un algorithme de Quasi-Newton.
- une fonction **`void bissection(double& x, double xhigh, double cible,ParamSolveur& param, const std::function<void (const ParamSolveur&, double)> & prox)const;`** qui résout l'équation par un algorithme de bissection.

Listing 3 – deux fonctions membres de la classe SolveurNonLineaire

```
typedef std::function<void (const ParamSolveur&, double)> CallBackType;
//Resolution par un algorithme de newton
void newton(double& x, double cible,ParamSolveur& param,
            const CallBackType& prox)const;
//Resolution par un algorithme de bissection
void bissection(double& x, double xsup, double cible,
                ParamSolveur& param,const CallBackType& prox)const;
```

Validations pour la deuxième partie

Besoin-2-3 : `Validation simple du solveur`
Fabriquer un problème modèle et l'utiliser pour tester l'implémentation du solveur. Voir par exemple Listing-9.

Besoin-2-4 : `Modification attendue`
Sous les hypothèses de volatilité constante. Modifier la classe **`Option`** de sorte que l'on puisse l'utiliser pour estimer la volatilité pour une valeur donnée du sous-jacent dans un call européen. Valider l'implémentation sur des exemples.

Extensions

Plusieurs extensions sont possibles pour produire un simulateur encore plus générique. La présente partie, bien que hors projet sert de pistes d'exploration future.

1. Traitement du cas des paramètres (volatilité, taux d'intérêt et divident) non constants.
2. Définition d'une interface utilisatrice avec la bibliothèque QT `http://qt.nokia.com/products/`.
3. Implémentation du solveur d'équation non-linéaires via le design pattern **`Etat`** pour gérer les variations de stratégies. Et bien d'autres (*me solliciter le cas échéant*).

Utilitaires

Utilitaire-1 : Outil de mesure du temps d'exécution

Listing 4 – fichier :Timer.hpp

```
/*-*- Mode: C; indent-tabs-mode:t; c-basic-offset: 4;tab-width: 4 -*-*/
/* SafeMiraBlackScholes  Copyright (C) APOUNG KAMGA Jean-Baptiste  2012
 *                              <jean-baptiste.apoung@math.u-psud.fr>
 * SafeMiraBlackScholes is free software: you can redistribute it and/or ↩
    modify it
 * under the terms of the GNU General Public License as published by the
 * Free Software Foundation, either version 3 of the License, or
 * (at your option) any later version.
 *
 * SafeMiraBlackScholes is distributed in the hope that it will be useful↩
    , but
 * WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
 * See the GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License ↩
    along
 * with this program.  If not, see <http://www.gnu.org/licenses/>.
 */
#ifndef _TIMER_HPP_
#define _TIMER_HPP_
#include <iosfwd>
/**
\class Timer pour chronométrer les durées d'exécution des bouts de code
```

```
 Exemple d'utilisation
   @code
   #include "Timer.hpp"
   voif f(){ }
   void timerSample(){
   Timer timer;
    timer.tic();
    f();
    timer.toc("duree permière appel de f()");
    timer.tic();
    f();
    timer.toc("duree second appel  de f()");
    std::cout<< timer << std::endl;
    }
@endcode
*/
class Timer{
public:
//LIFE CYCLE
Timer();
Timer(const Timer&) = delete;
Timer(Timer&&) = delete;
~Timer();
//OPERATORS
Timer& operator=(Timer&&) = delete;
Timer& operator=(const Timer&p) = delete;
//MODIFIERS
void tic();
void toc(const char *);
private:
class TimerImp;
TimerImp* m_imp;
void put(std::ostream&) const;
//IO
friend std::ostream&
operator<<(std::ostream& os, const Timer& timer){
timer.put(os); return os;
}
};
#endif // _TIMER_HPP_
```

Listing 5 – fichier :Timer.cpp

```
#include "Timer.hpp"
#include <unordered_map>
#include <string>
#include <iostream>
#include <chrono>
class Timer::TimerImp{
friend class Timer;
typedef std::chrono::high_resolution_clock ClockType;
typedef std::chrono::time_point < ClockType > TimePointType;
typedef std::unordered_map < std::string, double > DurationsStoredType;
typedef std::chrono::duration < double, std::ratio< 1,1000 >>
MilliSecondsType;
typedef std::chrono::duration < double, std::ratio< 1, 1 >>
SecondsType;
```

```
 TimePointType m_start;
 TimePointType m_end;
 DurationsStoredType m_durations;

 void tic(){ m_start = ClockType::now();}
 void toc(const char *cs){
 m_end = ClockType::now();
 double elapse_time =
 std::chrono::duration_cast < SecondsType > (m_end - m_start).count();
 std::string s(cs);
 for(auto & c:s) { c = std::toupper(c);}
    m_durations.insert(std::make_pair(s, elapse_time));
  }
 void put(std::ostream & os) const{
    for(auto v:m_durations)
      os << v.first << " : " << v.second << " s\n";
  }
};
////////////////////////////////////////////////////////////////////
Timer::Timer():m_imp(new TimerImp()){
}
Timer::~Timer(){
  delete m_imp;
}
void Timer::tic(){
  m_imp->tic();
}
void Timer::toc(const char *s){
  m_imp->toc(s);
}
void Timer::put(std::ostream & os) const {
  m_imp->put(os);
}
```

Utilitaire-2 : Exemples d'implémentations

Listing 6 – fichier :VariableAleatoireUniforme.hpp

```
#include "VariableAleatoire.hpp"
class VariableAleatoireUniforme : public VariableAleatoire{
  std::random_device rd;
  mutable std::mt19937 m_gen;
  mutable std::uniform_real_distribution <double> m_dist;
public:
  VariableAleatoireUniforme(Unsigned long dim = 1);
  VariableAleatoireUniforme(const VariableAleatoireUniforme&)=default;
  VariableAleatoireUniforme(VariableAleatoireUniforme&&)=default;
  VariableAleatoireUniforme&
  operator=(const VariableAleatoireUniforme&)=default;
  VariableAleatoireUniforme&
  operator=(VariableAleatoireUniforme&&)=default;
  ~VariableAleatoireUniforme(){;}
  void realisations(Tableau& v)const override;
};
```

Listing 7 – fichier :VariableAleatoireUniforme.cpp

```
#include "VariableAleatoireUniforme.hpp"

VariableAleatoireUniforme::VariableAleatoireUniforme(Int dim)
:VariableAleatoire(dim),m_gen(rd()),m_dist(0, 1){
 ;
 }

void VariableAleatoireUniforme::realisations(Tableau& v)const{
  for(int i = 0; i < m_dimension; i++) {
   v[i]= m_realisations[i] = m_dist(m_gen);
 }
```

Utilitaire-3 : Exemples de scripts de test

Listing 8 – fichier :testCallEuropee.cpp

```
void test_BS(){
typedef unsigned long Int;

Timer time;
double S = 50.;
double K = 50.;
double T = 1.;
double sigma = 0.3;
double r = 0.1;
double d =0.;
Int nb_tirage_MonteCarlo = 50000;

time.tic();
Action action(K, T, 2);
action.setParams(r,d,sigma);

PayOffCall pK(K);
Option call(pK, action,nb_tirage_MonteCarlo);

//Int horizon = call.horizon();
int M = 40;
double dx = 2*K/ M;
std::ofstream os("eds_template.dat");
for(double S0 = 0.; S0 < 2. * K; S0 += dx)
{
 call.nouveauChemin(S0);
 os<< S0 <<"  " << call.payOff(S0) <<"    "<< call[0] <<"\n";
}
time.toc ("eds_template");
std::cout<< time <<"\n";
}
```

Listing 9 – Exemple d'utilisation du Solveur non-linéaire

```
#include <iostream>
#include "SolveurNonLineaire.hpp"
#include "ParametresSolveurNonLineaire.hpp"

// Définition de l'équation à résoudre
struct Equation{
 double f(double x) const{return x-2;}
 double df(double x)const{return 1.;}
 };
 // Exemple de call back
 struct Visiteur{
 void operator()(const ParametresSolveurNonLineaire& param, double x){
    std::cout<<" Iteration : "<< param.iterationCourante()
             <<" Residu : " << param.residuCourant() <<"\n";
   if(param.aConverge())
    std::cout <<" Solution :  "<< x <<"\n";
  }
 };

//Fonction principale
int main(int argc, char** argv){
//definition des paramètres
double atol = 1e-8;
unsigned long maxiter = 10000;
ParametresSolveurNonLineaire param(atol, maxiter);

//definition du callback
auto callback = [=](const ParametresSolveurNonLineaire& param, double x)
  {
    std::cout<<" Iteration : "<< param.iterationCourante()
             <<" Residu : " << param.residuCourant() <<"\n";
  };

  //instantiation de l'equation
  Equation eq;

  //instantiation du solveur
  SolveurNonLineaire<Equation, ParametresSolveurNonLineaire>
  solveur(eq, &Equation::f, &Equation::df);

  // appel du solveur
  double x    = 0.;// valeur initiale
  double xsup = 2.;//suppose que la solution est dans [x,xssup]
  double cible = 0;// on résout f(x) = cible

 // methode de bissection
  solveur.bissection(x,xsup,cible,param, callback);
 //affichage de la solution
  std::cout<< " La solution par bissection est : "<< x <<"\n";
  // méthode de quasi-Newton
  x = 0;
  param.recharge(); // reinitialisation des paramètres
  solveur.newton(x,cible,param, Visiteur()); // on a change de callback
  std::cout<< " La solution par quasi-newton est : "<< x <<"\n";
 return 0;
 }
```

Utilitaire-4 : Illustration de l'interface graphique du projet