A rendre pour le lundi 1^{er} mars à mon bureau (5D1), en binôme

Consignes

Sont à rendre: un petit rapport contenant les quelques démonstrations mathématiques demandées, les résultats numériques et les codes sources COMMENTÉS.
Une archive des sources est à envoyer par mail à `temam@math.jussieu.fr` avec comme sujet `TP2 C++`. Une classe implémentant un générateur congruenciel sur $[0,1]$ vous est fournie à l'adresse:
(pour récupérer ce prototype, vous devez copier le répertoire: `cp -r RepertoireSource Destination /share/thot/users/enseignement/etemam/TP2/myRand`)

Partie A: Préliminaires et loi normale

Pour chacune des classes on procèdera à la validation du code écrit en vérifiant certaines propriétés statistiques associées aux variables aléatoires simulées (moyenne/variance empirique, théorème de limite centrale, Berry-Esseen...)

1. A partir de la classe `myRand` vérifier le théorème central limite et estimer une vitesse de convergence pour ce dernier.
2. On veut maintenant simuler des lois normales: on pourra à cet effet utiliser l'algorithme dit de Box et Muller qui utilise l'identité en loi suivante que l'on montrera: si $U_1, U_2 \sim \mathcal{U}([0,1])$ alors $(\cos(2\pi U_1)\sqrt{-2\ln(U_2)}, \sin(2\pi U_1)\sqrt{-2\ln(U_2)}) \sim \mathcal{N}(0, I_2)$ vecteur Gaussien standard de $\mathbb{R}^2$. Pour ne pas perdre une simulation sur deux, on considère la classe:

```
class Normale{
        myRand U;
        double mean, variance;
        bool *needtogenerate;
        double *Ncurr;
public:
        Normale(double mean_i=0., double variance_i=1., bool needtogeneratei=false);
        double simule() const;
        ~Normale();
};
```

   Ecrire le(s) constructeur(s) ainsi que la méthode `simule()` de la classe `Normale`. La classe, telle qu'elle est écrite est-elle correcte?
3. Testez à l'aide de fonctions annexes, le "bon comportement" de la classe écrite.
   a) Ecrire une fonction `double getEmpiricalMean(const Normale & N,long NMC)`, qui va renvoyer la moyenne empirique associée à `NMC` réalisations indépendantes de l'instance `N` de `Normale` considérée.
   b) Procéder de même pour la variance empirique, fonction `getEmpiricalVariance` de même prototype que la précédente.
   c) Ecrire une fonction de prototype `double * MonteCarlo(const Normale & N,long NMC, double alpha)` qui renverra la moyenne empirique associée à `NMC` réalisations de lois normales et la taille de l'intervalle de confiance associé au seuil `alpha`.
   d) Tracer en fonction du nombre de réalisations `NMC` que l'on fera varier entre 10^4 et 10^6 pour un pas judicieusement choisi la moyenne empirique et l'intervalle de confiance à 95% associé.
   e) Modifier la classe Normale de sorte à utiliser de façon générique (quelque soit la loi) l'approximation de Monte Carlo. On suggère de les faire hériter d'une même classe de base, `Loi_Proba` :

```
class Loi_Proba{
public:
        virtual double Simule() const=0;
};
```

   C'est cette classe qui sera désormais en argument des fonctions :
   - `double getEmpiricalMean(const Loi_Proba &,long)`
   - `double getEmpirical_Variance(const Loi_Proba &,long)`
   - `double * MonteCarlo (const Loi_Proba &,long, double )`

Partie B: PayOffs

Contrairement au premier TP, nous allons utiliser l'héritage et la généricité pour modéliser les payoffs et les options. Voici le prototype imposé:

```
1  class PayOff
2  {
3  public:
4          PayOff(){};
5          virtual double operator()(double Spot) const=0;
6          virtual ~PayOff(){};
7          virtual PayOff* clone() const=0;
8          virtual double * getParam(int nbParam) const=0;
9  };
10
11 class VanillaOption
12 {
13         double Expiry;
14         PayOff*  ThePayOffPtr;
15 public:
16         VanillaOption(const PayOff & ThePayOffi, double Expiryi);
17         VanillaOption(const VanillaOption& original);
18         VanillaOption & operator=(const VanillaOption & original);
19         ~VanillaOption();
20         double getExpiry() const;
21         double OptionPayOff(double Spot) const;
22 };
```

La procédure `getParam()` permet d'accéder aux paramètres de l'option si nécessaire.

1. Ecrire les classes dérivées de `PayOff` pour le call -`PayOffCall`- et le put - `PayOffPut`- (on écrira toutes les procédures sauf `virtual PayOff* clone() const;`).
2. A la ligne 15, pourquoi n'a t-on pas utiliser `PayOff & ThePayOff` ou plus simplement `PayOff ThePayOff`?
3. Pourquoi a t-on besoin des lignes 17,18 et 19? Détailler votre réponse.
4. Expliquer le code suivant:

```
PayOff* PayOffCall::clone() const
{
        return new PayOffCall(*this);
}
```

Partie C: Discrétisation d'un portefeuille autofinançant

Soit un espace de probabilité filtré $(\Omega, \mathbb{F}, (\mathbb{F}_t)_{t\geq 0}, \mathbb{P})$ muni d'un mouvement Brownien unidimensionnel standard W. On se place dans le cadre du modèle de Black et Scholes, i.e. on suppose qu'un actif financier $(S_t)_{t\geq 0}$ suit une dynamique de type Brownien géométrique, $S_t = S_0 \exp((\mu - \sigma^2/2)t + \sigma W_t)$, où S_0 est le cours initial de l'actif, μ est le rendement de l'actif sous la probabilité historique $\mathbb{P}$, σ le paramètre de volatilité.
Le propos de cet exercice est d'implémenter une stratégie de gestion dynamique permettant de répliquer à échéance $T > 0$ fixée le pay-off d'une option. On utilisera à cet effet un portefeuille autofinançant $(V_t)_{t\in[0,T]}$ composé d'actif risqué $(S_t)_{t\in[0,T]}$ et d'actif sans risque $(S^0_t)_{t\in[0,T]}$ que l'on supposera suivre la dynamique $dS^0_t = rS^0_t dt$. Ainsi, si $(\delta_t)_{t\in[0,T]}$ désigne la quantité d'actif risqué détenue dans le portefeuille à l'instant t, l'hypothèse d'autofinancement amène à la dynamique suivante pour le portefeuille

$$dV_t = \delta_t dS_t + (V_t - \delta_t S_t) r dt. \tag{1}$$

Par réplication on entend que $V_T = H(S_T),\ p.s.$ où H est le pay-off d'une option.

1. Résoudre le problème de réplication en cherchant (V_t, δ_t) sous la forme $v(t, S_t), \delta(t, S_t)$. On précisera les hypothèses nécessaires et/ou suffisantes sur le pay-off H ainsi que l'expression de δ en indiquant à quelle EDP elle est reliée et le processus dont le générateur infinitésimal intervient dans ladite EDP.
2. On veut ici discrétiser l'équation d'autofinancement (1) pour un pas $h = T/N,\ N \in \mathbb{N}^*$ fixé. On va tout d'abord écrire pour cela une classe `BrownienGeo` de prototype

```
class BrownienGeo{
        double mu, sigma, S0;
        double * SCurr;
```

```
        Normale N;
public:
        BrownienGeo(double mui=.05,double sigmai=.2,double S0i=100.);
        double getS0() const;
        double getVol() const;
        double Update(double Delta);
        ~BrownienGeo();
};
```

Par convention, à chaque appel de `getS0` on remettra le contenu de `SCurr` à la valeur `S0`. La méthode `Update` mettra à jour le contenu de `SCurr` en renvoyant $S_{t+\Delta}|S_t =$(`*SCurr`). Ecrire les méthodes de `BrownienGeo`.

3. On introduit une classe `Portfolio`

```
class Portfolio{
        double V,S,delta,r;
public:
        Portfolio(double ri=0.1, double Vi=10., double Si=100., double deltai=0.);
        void Update(double h, double Stip1, double ti, double newdelta);
        double getV() const;
};
```

où `V,S,delta` vont respectivement stocker les valeurs successives aux différents instants de discrétisation des valeurs $(V_{t_i}, S_{t_i}, \delta_{t_i})_{i\in[0,N]}$. La méthode `Update` met en oeuvre la discrétisation de (1) entre les instants t_i et t_{i+1}. Les variables `Stip1` et `newdelta` contiennent respectivement les valeurs de $S_{t_{i+1}}$ et $\delta(t_{i+1}, S_{t_{i+1}})$ qui serviront à mettre à jour les champs `S,delta` en fin de méthode. La méthode `getV` renvoie la valeur courante du portefeuille. Ecrire la classe `Portfolio`.

4. Enfin, on considère la classe

```
class DynamicHedging{
        BrownienGeo BG;
        double r;
        double Expiry;
        PayOff & PO;
public:
        DynamicHedging (PayOff & POi,double mui=.05, double sigmai=.2,double S0i=100,
                double ri=.02,  double Expiryi=1);
        DynamicHedging (PayOff & POi,const BrownienGeo & BGi,
                double ri=.02, double Expiryi=1);
        double GetHedge(int NS);
        double getValueANA(double t, double St) const;
        double getDeltaANA(double t,double St) const;
};
```

où `getValueANA, getDeltaANA` renvoient respectivement les valeurs analytiques du prix de l'option et de la couverture associée à l'instant `t` en argument lorsque $S_{\texttt{t}} =$`St` pour le pay-off `PO` en donnée membre. La fonction `GetHedge` renverra elle l'écart $e_T^h = H(S_T) - V_T^h$, où V_T^h désigne la valeur finale du portefeuille autofinançant discrétisé avec un pas de h.

a) En faisant tendre h vers 0, vérifier que $e_T^h \to 0$ en un sens que l'on précisera. On pourra faire varier h de 10^{-1} à 10^{-3}.

b) Ecrire un algorithme de Monte-Carlo associé à des réalisations $(e_T^{h,m})_{m\in\mathbb{N}^*}$ indépendantes de e_T^h. Représenter la variance empirique associée en fonction de h évoluant comme à la question précédente. On prendra $m \in \left[10^4, 10^6\right]$. On pourra à ce propos utiliser la classe `Loi_Proba`...

c) Estimer la vitesse de convergence de e_T^h en fonction de h. Si vous êtes amenés à utiliser une méthode de Monte Carlo, pensez à choisir m de sorte que l'erreur statistique soit négligeable devant l'erreur de discrétisation. On caractérisera à ce propos précisément la vitesse de convergence en norme $L^p, p \geq 1$ donné de e_T^h.