Formation GAUSS pour EDF

Niveau I

Thierry Roncalli

Financial Econometric Research Centre, City University Business School,
Frobisher Crescent, The Barbican, London EC2Y 8HB

e-mail : roncalli@montesquieu.u-bordeaux.fr

Table des matières

1 Introduction **1**

2 Les options de change **1**
2.1 La formule de Garman et Kohlhagen [1982] . . . 1
2.1.1 La notion de procédure . . . 3
2.1.2 La notion de bibliothèque . . . 5
2.2 La méthode binomiale d'évaluation . . . 6
2.2.1 Construction de la procédure . . . 6
2.2.2 Quelques exemples d'utilisation . . . 7
2.3 Calcul du delta d'une option . . . 8
2.3.1 Méthodes analytique et numérique . . . 8
2.3.2 Application au modèle CRR . . . 11
2.3.3 Extension aux autres coefficients caractéristiques . . . 15
2.4 Les autres types d'option . . . 15
2.4.1 Les options exotiques . . . 15
2.4.2 Les options américaines . . . 21
2.5 Le calcul de la volatilité implicite . . . 26
2.5.1 L'algorithme de la bisection . . . 26
2.5.2 L'algorithme de Newton-Raphson . . . 30

3 La méthode de Monte Carlo **31**
3.1 Illustration du théorème de limite centrale . . . 31
3.2 Application au modèle CRR . . . 33
3.3 La technique de réduction de variance . . . 36
3.4 Simulation d'équations différentielles stochastiques . . . 40
3.4.1 Algorithme d'Euler-Maruyama . . . 40
3.4.2 Application au modèle GK . . . 41
3.5 Extension au cas des EDS multidimensionnelles . . . 44
3.5.1 Le schéma de Taylor de forme forte à l'ordre 0.5 . . . 44
3.5.2 Application au modèle de Hull et White . . . 45
3.5.3 Valorisation des options dans les modèles de volatilité stochastique . . . 47

4 Optimisation **47**
4.1 La programmation quadratique . . . 47
4.2 L'algorithme de quasi-Newton . . . 48
4.3 L'optimisation non linéaire avec contraintes non linéaires d'égalité et d'inégalité . 49

5 Value at Risk **53**
5.1 Le cas linéaire gaussien . . . 53
5.1.1 Génération de nombres aléatoires gaussiens . . . 53
5.1.2 La technique de l'inversion de fonction de répartition . . . 57
5.1.3 Calcul de la *VaR* par la méthode des quantiles . . . 59
5.2 Le cas non linéaire (ou la *VaR* des options) . . . 61
5.2.1 Les sources de variation des prix des options . . . 61
5.2.2 Disparition de la propriété de normalité . . . 62
5.2.3 Calcul de la *VaR* par la méthode de Monte Carlo . . . 64
5.2.4 Calcul de la *VaR* par l'approximation de Taylor . . . 65
5.2.5 *VaR* et portefeuille d'options . . . 66
5.3 Les techniques d'estimation . . . 68

5.3.1 Estimateurs classiques de la volatilité 68
5.3.2 Volatilité stochastique 68
5.3.3 La théorie des valeurs extrêmes 68
5.4 La gestion des données 68

1 Introduction

2 Les options de change

2.1 La formule de Garman et Kohlhagen [1982]

Considérons une option de change. Nous notons S_0 la valeur du sous-jacent, K le prix d'exercice, σ la volatilité du cours de change, r le taux d'intérêt instantané national et $r^\star$ le taux d'intérêt instantané étranger. La prime de l'option d'achat de maturité τ est donné par la formule suivante :

$$C = S_0 e^{-r^\star \tau} \Phi(d_1) - K e^{-r\tau} \Phi(d_2)$$

avec

$$\begin{aligned} d_1 &= \frac{\ln \frac{S_0}{K} + (r - r^\star)\tau}{\sigma\sqrt{\tau}} + \frac{1}{2}\sigma\sqrt{\tau} \\ d_2 &= d_1 - \sigma\sqrt{\tau} \end{aligned}$$

Le code GAUSS correspond au programme suivant. Quelques remarques :

1. La fin d'une instruction correspond au point-virgule. Il est donc possible de mettre plusieurs instructions sur une même ligne. Cela implique aussi qu'une instruction peut tenir sur plusieurs lignes.
2. Nous obtenons l'affichage d'une information au moyen de la commande `print`. Il existe plusieurs commandes pour agir sur le formatage : `format`, `ftos`, `printfm`, `printfmt`, etc.
3. Nous pouvons récupérer les résultats dans un fichier texte ascii au moyen des instructions `output file` et `output off`. La commande `output file` admet deux options `reset` (<) et `on` (<<).

```
new;

S0 = 3.5;
K = 3.5;
tau = 60/365;
r = 0.08;
rstar = 0.06;
sigma = 0.09;

w = sigma*sqrt(tau);
d1 = ( ln(S0/K) + (r-rstar)*tau )/w + 0.5*w;
d2 = d1 - w;

C = exp(-rstar*tau)*S0*cdfn(d1) - k*exp(-r*tau)*cdfn(d2);

print C;

output file = gk1.out reset;

print ftos(C,"Valeur de la prime d'achat : %lf",5,3);

output off;
```

```
Valeur de la prime d'achat : 0.056
```

GAUSS est un langage matriciel vectorisé. Pour calculer plusieurs primes d'option, il n'est pas nécessaire d'utiliser une boucle. Nous pouvons employer les opérateurs $E \times E$ (élément par élément). Ces opérateurs sont les suivants :

$$+ \quad - \quad .\hat{} \quad .* \quad ./$$

Dans l'exemple suivant, nous calculons la prime d'option d'achat pour cinq prix d'exercice différents. Remarquez l'utilisation de la concaténation verticale | pour définir le vecteur.

Exercice 2.1 *Utiliser l'instruction `let` pour définir la variable K.*

```
new;

S0 = 3.5;
K = 3.48|3.49|3.5|3.51|3.52;
tau = 60/365;
r = 0.08;
rstar = 0.06;
sigma = 0.09;

w = sigma.*sqrt(tau);
d1 = ( ln(S0./K) + (r-rstar).*tau )./w + 0.5*w;
d2 = d1 - w;

C = exp(-rstar.*tau).*S0.*cdfn(d1) - k.*exp(-r.*tau).*cdfn(d2);

output file = gk2.out reset;

print "            K             C ";
print K~C;

output off;
```

```
            K             C

       3.4800000      0.067304164
       3.4900000      0.061627748
       3.5000000      0.056256878
       3.5100000      0.051193347
       3.5200000      0.046437065
```

Un autre exemple de vectorisation :

```
new;

S0 = 3.5;
K = 3.48|3.49|3.5|3.51|3.52;
tau = 60/365~30/365;
r = 0.08;
rstar = 0.06;
sigma = 0.09;
```

```
w = sigma.*sqrt(tau);
d1 = ( ln(S0./K) + (r-rstar).*tau )./w + 0.5*w;
d2 = d1 - w;

C = exp(-rstar.*tau).*S0.*cdfn(d1) - k.*exp(-r.*tau).*cdfn(d2);

output file = gk3.out reset;

print C;

output off;
```

```
     0.067304164      0.049961196
     0.061627748      0.044139997
     0.056256878      0.038752763
     0.051193347      0.033803380
     0.046437065      0.029290390
```

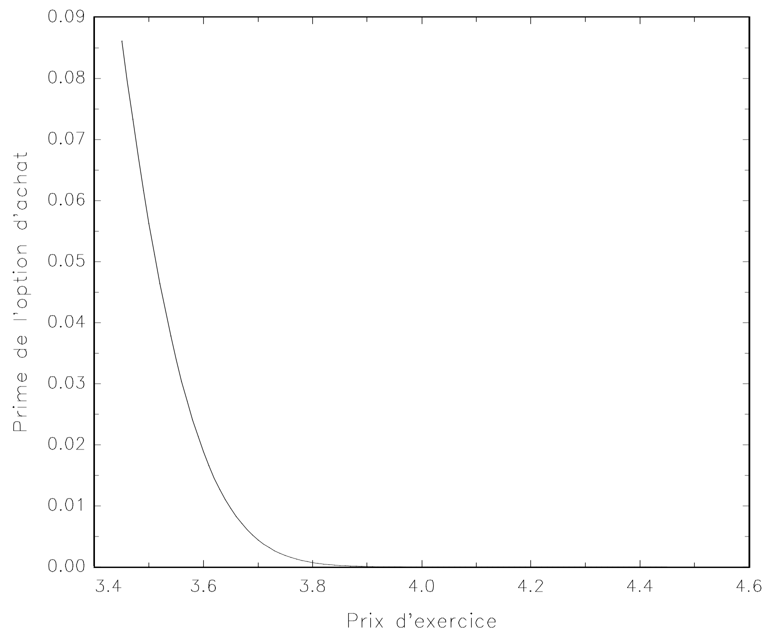
2.1.1 La notion de procédure

Une procédure est un bloc de commandes qui peut être utilisé plusieurs fois dans le programme. Cela permet d'eviter de nombreuses redondances. Nous remarquons que la formule de Garman et Kohlhagen dépend de six variables. Ces six variables seront les variables input de la procédure. La variable output sera bien sûr le prix de l'option d'achat. Quelques remarques :

1. Une procédure commence et finit toujours avec les commandes `proc` et `endp`.
2. Le nom d'un procédure ne peut dépasser 32 caractères.
3. Toutes les variables d'une procédures sont définies de façon locales.
4. GAUSS n'est pas un langage typé. Les déclarations integer, complex, real, etc n'ont pas de sens.
5. Nous pouvons placer la procédure n'importe où dans le programme.

Exercice 2.2 *Modifier le programme en utilisant la directive de compilation* `#include`*.*

```
new;
library pgraph;

proc callGK(S0,K,sigma,tau,r,rstar);
  local w,d1,d2,C;

  w = sigma.*sqrt(tau);
  d1 = ( ln(S0./K) + (r-rstar).*tau )./w + 0.5*w;
  d2 = d1 - w;

  C = exp(-rstar.*tau).*S0.*cdfn(d1) - k.*exp(-r.*tau).*cdfn(d2);
  retp(C);
endp;
```

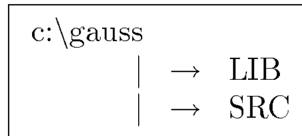


Graphique 1:

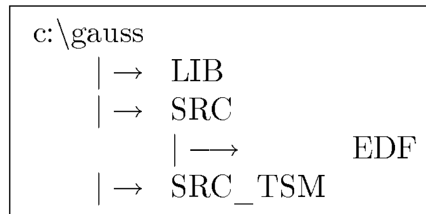
```
S0 = 3.5;
tau = 60/365;
r = 0.08;
rstar = 0.06;
sigma = 0.09;

output file = gk4.out reset;

print callGK(3.51,S0,sigma,tau,r,rstar);

output off;

PrixExercice = seqa(3.45,0.01,101);
Prime = callGK(S0,PrixExercice,sigma,tau,r,rstar);

graphset;
  _pnum = 2; _pdate = "";
  xlabel("Prix d'exercice");
  ylabel("Prime de l'option d'achat");
  graphprt("-c=1 -cf=gk4.eps");
  xy(PrixExercice,Prime);

     0.061788045
```

2.1.2 La notion de bibliothèque

Une bibliothèque est une collection de routines (les procédures) qui sont généralement utilisées de façon fréquente. Celle-ci est constituée d'un fichier **.lcg** qui se trouve dans le répertoire **LIB** et de fichiers sources avec les extensions **.src**, **.dec** et **.ext**. Par défaut, nous avons l'arborescence suivante :



Le répertoire **LIB** contient les déclarations de bibliothèques, par exemple *gauss.lcg*, *pgraph.lcg* ou *optmum.lcg*. Les fichiers sources sont dans le répertoire *src*. Il est préférable d'utiliser un autre répertoire pour sauver **ses** fichiers sources pour ne pas les mélanger avec ceux d'APTECH. Par exemple,



Ainsi, la bibliothèque **EDF** sera déclarée dans le répertoire *lib* avec le fichier *edf.lcg* et les codes sources se trouveront dans le répertoire *src/edf*. Les fichiers sources ne sont rien d'autre qu'une collection de routines. Voici comment se présente le fichier *gk.src* :

```
proc callGK(S0,K,sigma,tau,r,rstar);
  local w,d1,d2,C;

  w = sigma.*sqrt(tau);
  d1 = ( ln(S0./K) + (r-rstar).*tau )./w + 0.5*w;
  d2 = d1 - w;

  C = exp(-rstar.*tau).*S0.*cdfn(d1) - k.*exp(-r.*tau).*cdfn(d2);
  retp(C);
endp;

proc putGK(S0,K,sigma,tau,r,rstar);
  local w,d1,d2,P;

  w = sigma.*sqrt(tau);
  d1 = ( ln(S0./K) + (r-rstar).*tau )./w + 0.5*w;
  d2 = d1 - w;

  P = -exp(-rstar.*tau).*S0.*cdfn(-d1) + k.*exp(-r.*tau).*cdfn(-d2);
  retp(P);
endp;
```

Pour construire la bibliothèque, nous utilisons la commande `lib`, par exemple

```
lib edf gk.src
```

Le fichier *edf.lcg* est alors le suivant :

```
c:\gauss\src\edf\gk.src
    callgk                            : proc
    putgk                             : proc
```

Vérifions le relation de parité call/put.

```
new;
library edf;

S0 = 3.5;
K = 3.49|3.50|3.51;
r = 0.08;
rstar = 0.06;
tau = 60/365;

C = callGK(S0,K,0.09,tau,r,rstar);
P = putGK(S0,K,0.09,tau,r,rstar);

diff = (C - P) - (S0.*exp(-rstar.*tau) - K.*exp(-r.*tau));

output file = gk5.out reset;

print diff;

output off;
```

```
      0.00000000
      0.00000000
  4.4408921e-016
```

Exercice 2.3 *Transformer les procédures callGK et putGK en une seule procédure avec deux sorties C et P.*

Exercice 2.4 *Introduire une variable globale qui permet de choisir le type de l'option ("put" ou "call").*

2.2 La méthode binomiale d'évaluation

2.2.1 Construction de la procédure

Nous rappelons que la prime d'une option de change dans le modèle de Cox, Ross et Rubinstein est donnée par la formule suivante :

$$C = e^{-r\tau} \sum_{j=0}^{N} C_N^j \pi^j (1-\pi)^{N-j} \max\left(0, S_0 u^j d^{N-j} - K\right)$$

avec N le nombre d'arbitrages. Les paramètres u, d et π valent respectivement

$$\begin{aligned} u &= \exp\left(\sigma\sqrt{\frac{\tau}{N}}\right) \\ d &= \frac{1}{u} \\ \pi &= \frac{\exp\left((r-r^\star)\frac{\tau}{N}\right) - d}{u-d} \end{aligned}$$

Nous pouvons implémenter cette formule de la façon suivante :

```
proc (2) = optionCRR(S0,K,sigma,tau,r,rstar,N);
  local u,d,fa,pi_,q;
  local j,B,C,P,G;

  u = exp(sigma.*sqrt(tau./N));
  d = 1./u;
  fa = exp(-r.*tau);

  pi_ = (exp((r-rstar).*tau./N)-d)./(u-d);
  q = 1 - pi_;

  j = seqa(0,1,N+1);
  B = (N!)./((j!).*((N-j)!));

  C = 0;
  P = 0;

  j = 0;
  do until j > N;
     G = S0.*(u^j).*(d^(N-j)) - K;
     G = G .* ( G .> 0);
     C = C + B[j+1]*(pi_^j).*(q^(N-j)).*G;
     G = K - S0.*(u^j).*(d^(N-j));
     G = G .* ( G .> 0);
     P = P + B[j+1]*(pi_^j).*(q^(N-j)).*G;
     j = j + 1;
  endo;

  C = fa.*C;
  P = fa.*P;

  retp(C,P);
endp;
```

Une remarque importante. Nous avons choisi d'utiliser une boucle sur le nombre d'arbitrage pour faire porter la vectorisation sur les autres paramètres. **Comment éviter la boucle si nous ne voulons pas la vectorisation ?**

2.2.2 Quelques exemples d'utilisation

Exercice 2.5 *Ajouter la procédure* `optionCRR` *à la bibliothèque EDF.*

Dans l'exemple suivant, nous vérifions la relation de parité call/put pour le modèle CRR.

```
new;
library edf;

S0 = 3.5;
K = 3.49|3.50|3.51;
r = 0.08;
rstar = 0.06;
tau = 60/365;
```

```
N = 3;
{C,P} = optionCRR(S0,K,0.09,tau,r,rstar,N);

diff = (C - P) - (S0.*exp(-rstar.*tau) - K.*exp(-r.*tau));

output file = crr1.out reset;

print diff;

output off;


  3.6776138e-016
  1.8735014e-016
  4.7184479e-016
```

Voici un exemple de convergence du modèle de CRR vers le modèle GK.

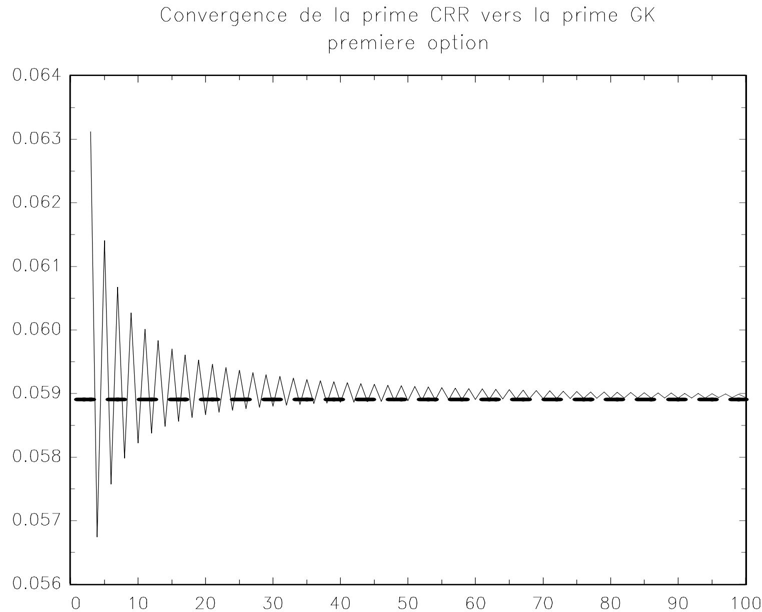
```
new;
library edf,pgraph;

S0 = 3.5;
K = 3.495|3.5;
sigma = 0.09|0.16;
r = 0.08;
rstar = 0.06;
tau = 60/365;

PrimeGK = callGK(S0,K,sigma,tau,r,rstar);

N = 100;
PrimeCRR = miss(zeros(2,N),0);

i = 3;
do until i > N;
  {PrimeCRR[.,i],P} = optionCRR(S0,K,sigma,tau,r,rstar,i);
  i = i + 1;
endo;

titre = "Convergence de la prime CRR vers la prime GK";

graphset;
  _pdate = ""; _pnum = 2; _plwidth = 0|10;
  title(titre$+"\Lpremiere option");
  graphprt("-c=1 -cf=crr1.eps");
  xy(seqa(1,1,N),PrimeCRR[1,.]'~PrimeGK[1]*ones(N,1));
```

2.3 Calcul du delta d'une option

2.3.1 Méthodes analytique et numérique

Pour le modèle de Garman et Kohlhagen, nous avons

$$\Delta_{\mathrm{C}} = e^{-r^{\star}\tau}\Phi\left(d_1\right)$$



Graphique 2:

Dans le programme suivant, nous calculons le delta d'une option d'achat de façon analytique et numérique. Pour cela, nous utilisons la procédure `gradp`. Quelques remarques :

1. `gradp` est une procédure et non une commande de GAUSS. Elle est définie dans la bibliothèque GAUSS.

2. Cette commande permet d'obtenir le vecteur gradient ou la matrice jacobienne d'une fonction $f(x)$ pour x égal à x_0. Il est donc nécessaire de transformer la procédure `callGK` en une fonction de la forme $C(S)$.

3. Pour passer une fonction comme argument d'une procédure, nous utilisons un pointeur &.

4. Le calcul analytique s'avère particulièrement intéressant lorsque nous ne disposons pas de formules analytiques (option de type CRR, options américaines et exotiques).

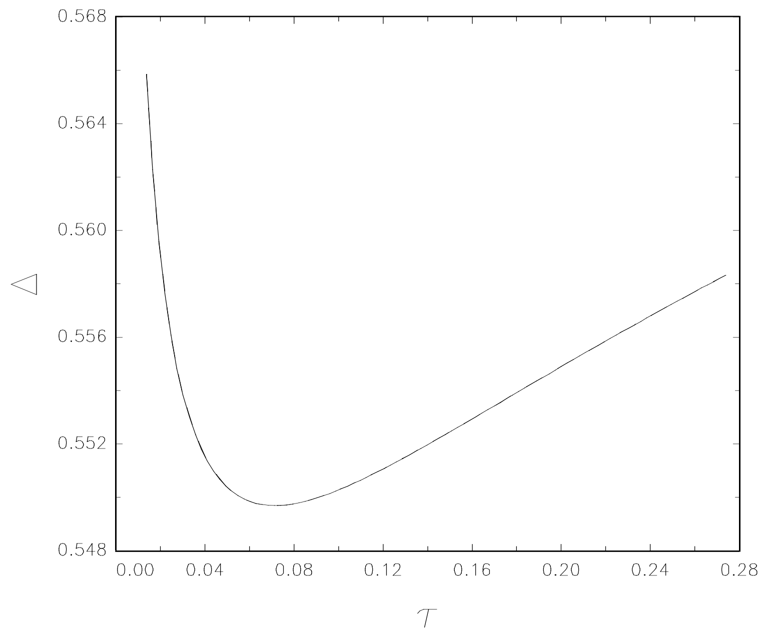
```
new;
library edf,pgraph;

S0 = 3.5;
K = 3.495;
sigma = 0.09;
r = 0.08;
rstar = 0.06;
tau = 60/365;

proc deltaGK(S0,K,sigma,tau,r,rstar);
  local w,d1,delta;
```

```
  w = sigma.*sqrt(tau);
  d1 = ( ln(S0./K) + (r-rstar).*tau )./w + 0.5*w;

  delta = exp(-rstar.*tau).*cdfn(d1);

  retp(delta);
endp;


proc FonctionAuxiliaire(S0);
  retp(callGK(S0,K,sigma,tau,r,rstar));
endp;

delta1 = deltaGK(S0,K,sigma,tau,r,rstar);
delta2 = gradp(&FonctionAuxiliaire,3.5);

output file = delta1.out reset;

print delta1;
print delta2;


output off;
```

```
      0.55315684
      0.55315690
```

Voici un nouvel exemple de vectorisation :

```
new;
library edf,pgraph;

S0 = 3.5;
K = 3.495;
sigma = 0.09;
r = 0.08;
rstar = 0.06;

tau = seqa(5/365,1/365,96);

proc deltaGK(S0,K,sigma,tau,r,rstar);
  local w,d1,delta;

  w = sigma.*sqrt(tau);
  d1 = ( ln(S0./K) + (r-rstar).*tau )./w + 0.5*w;

  delta = exp(-rstar.*tau).*cdfn(d1);

  retp(delta);
endp;


proc FonctionAuxiliaire(S0);
```



Graphique 3:

```
  retp(callGK(S0,K,sigma,tau,r,rstar));
endp;

delta1 = deltaGK(S0,K,sigma,tau,r,rstar);
delta2 = gradp(&FonctionAuxiliaire,3.5);


graphset;
  _pnum = 2; _pdate = ""; _paxht = 0.30;
  fonts("simplex simgrma");
  ylabel("\202D\201");
  xlabel("\202t\201");
  graphprt("-c=1 -cf=delta2.eps");
  xy(tau,delta1~delta2);
```

2.3.2 Application au modèle CRR

Voici un petit exemple qui montre la différence fondamentale entre les boucles `do while` et `do until` et la boucle `for`. Pour cette dernière instruction, l'indice est une variable localisée.

```
new;
library edf,pgraph;

declare matrix N;

S0 = 3.5;
```

```
K = 3.495;
sigma = 0.09;
tau = 90/365;
r = 0.08;
rstar = 0.06;


proc deltaGK(S0,K,sigma,tau,r,rstar);
  local w,d1,delta;

  w = sigma.*sqrt(tau);
  d1 = ( ln(S0./K) + (r-rstar).*tau )./w + 0.5*w;

  delta = exp(-rstar.*tau).*cdfn(d1);

  retp(delta);
endp;


proc FonctionAuxiliaire(S0);
  local C,P;

  {C,P} = optionCRR(S0,K,sigma,tau,r,rstar,N);
  retp(C);
endp;

output file = delta3.out reset;

delta1 = deltaGK(S0,K,sigma,tau,r,rstar);
print "delta1 = " delta1;

for N (1,5,1);
  delta2 = gradp(&FonctionAuxiliaire,S0);
  print delta2;
endfor;

print "N = " N;

for i (1,10,1);
  N = i;
  delta2 = gradp(&FonctionAuxiliaire,S0);
  print delta2;
endfor;

output off;
```

```
delta1 =       0.55709933
               .
               .
               .
               .
               .
N =       0.00000000
      0.55787555
```

```
    0.78299293
    0.54911614
    0.72471104
    0.54731915
    0.69518495
    0.54655285
    0.67661950
    0.54612916
    0.66357644
```

Ajoutons une procédure de calcul du DELTA pour le modèle CRR à la bibliothèque **EDF**. Pour cela, nous devons utiliser des variables **externes**. Nous construisons deux fichiers. Le fichier *.dec* permet de déclarer les variables et le fichier *.ext* permet de les rendre externes.

delta.dec

```
declare matrix _deltaCRR_K;
declare matrix _deltaCRR_sigma;
declare matrix _deltaCRR_tau;
declare matrix _deltaCRR_r;
declare matrix _deltaCRR_rstar;
declare matrix _deltaCRR_N;
```

delta.ext

```
external matrix _deltaCRR_K;
external matrix _deltaCRR_sigma;
external matrix _deltaCRR_tau;
external matrix _deltaCRR_r;
external matrix _deltaCRR_rstar;
external matrix _deltaCRR_N;
```

delta.src

```
proc deltaCRR(S0,K,sigma,tau,r,rstar,N);
  local delta;

   _deltaCRR_K = K;
   _deltaCRR_sigma = sigma;
   _deltaCRR_tau = tau;
   _deltaCRR_r = r;
   _deltaCRR_rstar = rstar;
   _deltaCRR_N = N;

  delta = gradp(&FonctionAuxiliaire,S0);

  retp(delta);
endp;


proc FonctionAuxiliaire(S0);
  local K,sigma,tau,r,rstar,N;
  local C,P;
```

```
  K = _deltaCRR_K;
  sigma = _deltaCRR_sigma;
  tau = _deltaCRR_tau;
  r = _deltaCRR_r;
  rstar = _deltaCRR_rstar;
  N = _deltaCRR_N;



  {C,P} = optionCRR(S0,_deltaCRR_K,_deltaCRR_sigma,_deltaCRR_tau,r,rstar,N);
  retp(C);
endp;
```

Pour rajouter la procédure deltaCRR à la bibliothèque EDF, nous exécutons les lignes de commandes suivantes :

```
lib edf delta.dec
lib edf delta.ext
lib edf delta.src
```

Le fichier *edf.lcg* devient :

```
c:\gauss\src\edf\gk.src
    callgk                          : proc
    putgk                           : proc

c:\gauss\src\edf\crr.src
    optioncrr                       : proc

c:\gauss\src\edf\delta.dec
    _deltacrr_k                     : matrix
    _deltacrr_sigma                 : matrix
    _deltacrr_tau                   : matrix
    _deltacrr_r                     : matrix
    _deltacrr_rstar                 : matrix
    _deltacrr_n                     : matrix

c:\gauss\src\edf\delta.ext

c:\gauss\src\edf\delta.src
    deltacrr                        : proc
    fonctionauxiliaire              : proc
```

Voici un exemple d'utilisation de la procédure deltaCRR de la bibliothèque EDF :

```
new;
library edf;

S0 = 3.5;
K = 3.495;
sigma = 0.09;
tau = 90/365;
r = 0.08;
```

```
rstar = 0.06;

output file = delta4.out reset;

i = 1;
do until i > 4;
  delta = deltaCRR(S0,K,sigma,tau,r,rstar,i);
  print delta;
  i = i + 1;
endo;

output off;
```

```
      0.55787555
      0.78299293
      0.54911614
      0.72471104
```

2.3.3 Extension aux autres coefficients caractéristiques

Nous pouvons calculer les autres coefficients caractéristiques de la même façon que précédemment.

Exercice 2.6 *Adapter les programmes précédents pour le GAMMA. Nous rappelons que*

$$\Gamma_C = \frac{e^{-r^\star \tau}\phi\left(d_1\right)}{S_0 \sigma \sqrt{\tau}}$$

Pour obtenir la matrice hessienne, utiliser la commande `hessp`.

2.4 Les autres types d'option

2.4.1 Les options exotiques

Pour valoriser les options exotiques de change dans le cas du modèle de CRR, nous devons construire deux procédures auxiliaires. La première correspond à une procédure de dénombrement de chemins binomiaux d'ordre N. Par exemple, pour N égal à 2, nous avons 4 chemins possibles

$$\begin{array}{c}(1,1)\\(1,0)\\(0,1)\\(0,0)\end{array}$$

Pour N égal à 3, nous avons 8 chemins possibles

$$\begin{array}{c}(1,1,1)\\(1,1,0)\\(1,0,1)\\(1,0,0)\\(0,1,1)\\(0,1,0)\\(0,0,1)\\(0,0,0)\end{array}$$

Le cas général n'est pas très facile à programmer. Néanmoins, l'utilisation du produit de Kronecker `.*.` et d'une procédure de réduction de matrice `trimr` permet d'obtenir un code assez court. Il suffit de remarquer que nous pouvons employer la séquence suivante

$$\begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix} \rightarrow \begin{matrix} \begin{bmatrix} 1 \\ 1 \end{bmatrix} \rightarrow \begin{matrix} [1] \\ [0] \end{matrix} \\ \begin{bmatrix} 0 \\ 0 \end{bmatrix} \rightarrow \begin{matrix} [1] \\ [0] \end{matrix} \end{matrix}$$

$$\begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \rightarrow \begin{matrix} \begin{bmatrix} 1 \\ 1 \end{bmatrix} \rightarrow \begin{matrix} [1] \\ [0] \end{matrix} \\ \begin{bmatrix} 0 \\ 0 \end{bmatrix} \rightarrow \begin{matrix} [1] \\ [0] \end{matrix} \end{matrix}$$

A partir d'un vecteur composé de 1 et 0, nous gardons la partie médiane (avec la commande `trimr`) que nous reproduisons (avec le produit de Kronecker `.*.`). La deuxième procédure qui permet de construire les chemins binomiaux du modèle de valorisation des actifs contingents de type Cox, Ross et Rubinstein est basée sur la procédure précédente. L'idée est de remplacer la valeur 1 par le coefficient u et la valeur 0 par le coefficient d. Nous pouvons utiliser alors cette procédure pour valoriser les options exotiques, par exemple les options asiatiques ou lookback. Pour obtenir la valeur de l'option exotique, nous utilisons le même algorithme :

1. Construction de l'ensemble des chemins binomiaux ;
2. Détermination de la fonction de payoff pour tous les états de la nature ;
3. Calcul du prix de l'option en utilisant la mesure de probabilité risque neutre.

Quelques commentaires sur le code :

- `S` est une matrice de dimension $2^N \times (N+1)$ avec N le nombre d'arbitrages du modèle CRR. Chaque ligne de la matrice correspond à une trajectoire du sous-jacent sous la mesure de probabilité neutre au risque.
- `ST = S[.,Narbitrage+1];` permet de sélectionner la $N+1$-ième colonne de la matrice `S`. `ST` est donc un vecteur de dimension 2^N comprenant les valeurs du sous-jacent à l'échéance de l'option pour l'ensemble des états de la nature.
- `Sm = minc(S');` correspond aux valeurs prises par la fonction $\min_{t_0 \leq t \leq T} S(t)$.
- Nous obtenons la fonction de payoff $G(T) = \max\left(0, S(T) - \min_{t_0 \leq t \leq T} S(t)\right)$ en employant les opérateurs élément par élément `.*` et `.>`.

```
/*
** Procedure de denombrement des chemins binomiaux
*/


proc (1) = _denombrementCheminsBinomiaux(Narbitrage);
  local x,cn,xi,i;
```

```
  x = zeros(2^Narbitrage,Narbitrage);
  cn = 2^(Narbitrage-1);
  xi = ones(cn,1)|zeros(cn,1);
  x[.,1] = xi;
  i = 2;
  do until i > Narbitrage;
    cn = cn/2;
    xi = trimr(xi,cn,cn);
    x[.,i] = ones(2^(i-1),1).*.xi;
    i = i + 1;
  endo;

  retp(x);
endp;


/*
** Procedure de construction des chemins binomiaux
*/


proc (3) = _constructionCheminsBinomiaux(S0,sigma,tau,r,delta,Narbitrage);
  local u,d,pi_,fa;
  local x,S,Si,i,xi,cn,j,prob;

  u = exp(sigma*sqrt(tau/Narbitrage));    /* coefficient de hausse   */
  d = 1/u;                                /* coefficient de baisse   */
  pi_ = (exp((r-delta)*tau/Narbitrage)-d)
        /(u-d);                           /* probabilite binomiale   */
  fa = exp(-r*tau);                       /* facteur d'actualisation */

  x = _denombrementCheminsBinomiaux(Narbitrage);
  S = zeros(2^Narbitrage,Narbitrage+1);
  S[.,1] = S0*ones(2^Narbitrage,1);
  Si = S0;
  i = 1;
  do until i > Narbitrage;
    xi = x[.,i];
    cn = xi .== 1;
    Si = Si .* (u.*cn + d.*(1-cn));
    S[.,i+1] = Si;
    i = i + 1;
  endo;

  j = sumc(x');
  prob = (pi_^j).*((1-pi_)^(Narbitrage-j));

  retp(S,prob,fa);
endp;


/*
**
```

```
*/


proc (2) = FloatingStrikeOption(S0,sigma,tau,r,rstar,Narbitrage);
  local S,prob,fa;
  local ST,Sm,Gcall,Gput;
  local floatCall,floatPut;

  {S,prob,fa} = _constructionCheminsBinomiaux(S0,sigma,tau,r,rstar,Narbitrage);

  ST = S[.,Narbitrage+1];
  Sm = meanc(S');

  Gcall = ST - Sm;
  Gcall = Gcall .* (Gcall .> 0);

  Gput = Sm - ST;
  Gput = Gput .* (Gput .> 0);

  floatCall = fa*(prob'Gcall);
  floatPut = fa*(prob'Gput);

  retp(floatCall,floatPut);
endp;


proc (2) = FixedStrikeOption(S0,K,sigma,tau,r,rstar,Narbitrage);
  local S,prob,fa;
  local Sm,Gcall,Gput;
  local fixedCall,fixedPut;

  {S,prob,fa} = _constructionCheminsBinomiaux(S0,sigma,tau,r,rstar,Narbitrage);

  Sm = meanc(S');

  Gcall = Sm - K;
  Gcall = Gcall .* (Gcall .> 0);

  Gput = K - Sm;
  Gput = Gput .* (Gput .> 0);

  fixedCall = fa*(prob'Gcall);
  fixedput = fa*(prob'Gput);

  retp(fixedCall,fixedPut);
endp;


proc (2) = LookBackOption(S0,sigma,tau,r,rstar,Narbitrage);
  local S,prob,fa;
  local ST,Smin,Smax,Gcall,Gput;
  local lbCall,lbPut;
```

```
  {S,prob,fa} = _constructionCheminsBinomiaux(S0,sigma,tau,r,rstar,Narbitrage);

  ST = S[.,Narbitrage+1];
  Smin = minc(S');
  Smax = maxc(S');

  Gcall = ST - Smin;
  Gcall = Gcall .* (Gcall .> 0);

  Gput = Smax - ST;
  Gput = Gput .* (Gput .> 0);

  lbCall = fa*(prob'Gcall);
  lbPut = fa*(prob'Gput);

  retp(lbCall,lbPut);
endp;
```

Voyons un exemple d'utilisation de la première procédure :

```
new;
library edf;

x = _denombrementCheminsBinomiaux(4);

output file = exotic1.out reset;

call printfmt(x,1);

output off;
```

```
              1                1                1                1
              1                1                1                0
              1                1                0                1
              1                1                0                0
              1                0                1                1
              1                0                1                0
              1                0                0                1
              1                0                0                0
              0                1                1                1
              0                1                1                0
              0                1                0                1
              0                1                0                0
              0                0                1                1
              0                0                1                0
              0                0                0                1
              0                0                0                0
```

Nous pouvons retrouver le prix de l'option d'achat européenne à partir des trajectoires binomiales, puisque nous avons sous la mesure de probabilité neutre au risque

$$C = e^{-r\tau} E' \left[G(T) \right]$$

avec

$$G(T) = (S(T) - K)_+$$

```
new;
library edf;

S0 = 3.5;
K = 3.49;
sigma = 0.09;
r = 0.08;
rstar = 0.06;
tau = 60/365;
N = 3;

{C,P} = optionCRR(S0,K,0.09,tau,r,rstar,N);

{S,prob,fa} = _constructionCheminsBinomiaux(S0,sigma,tau,r,rstar,N);

output file = exotic2.out reset;

print "Chemins binomiaux :";
call printfmt(S,1);

print "Fonction de probabilite risque-neutre :";
call printfmt(prob,1);

ST = S[.,N+1];
G1 = ST - K; G2 = K - ST;
G1 = G1 .* (G1 .> 0); G2 = G2 .* (G2 .> 0);

print "C = " C;
print "P = " P;
print "fa*(prob'G1) = " fa*(prob'G1) ;
print "fa*(prob'G2) = " fa*(prob'G2) ;
output off;
```

```
Chemins binomiaux :
            3.5       3.5745181       3.6506228       3.7283478
            3.5       3.5745181       3.6506228       3.5745181
            3.5       3.5745181             3.5       3.5745181
            3.5       3.5745181             3.5       3.4270354
            3.5       3.4270354             3.5       3.5745181
            3.5       3.4270354             3.5       3.4270354
            3.5       3.4270354       3.3555918       3.4270354
            3.5       3.4270354       3.3555918       3.2856377
Fonction de probabilite risque-neutre :
       0.1412212
      0.12996438
      0.12996438
      0.11960485
      0.12996438
      0.11960485
      0.11960485
      0.11007109
C =      0.065742529
P =      0.044497969
fa*(prob'G1) =      0.065742529
```

```
fa*(prob'G2) =      0.044497969
```

Voici un exemple de valorisation d'une option look-back :

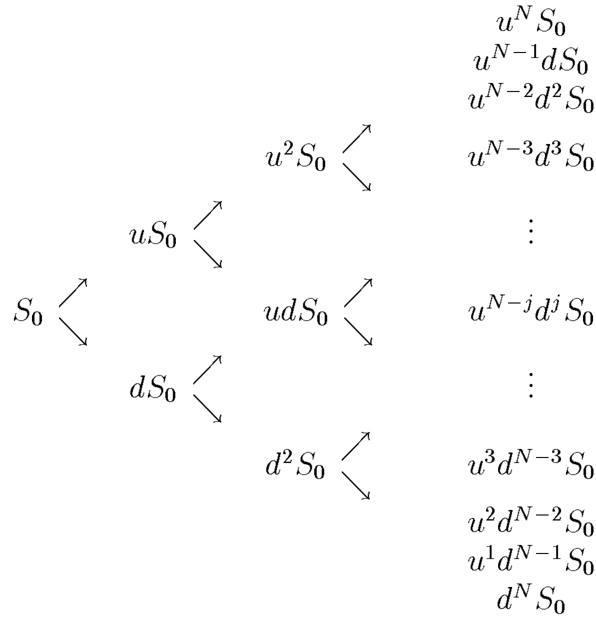
```
new;
library edf;

S0 = 3.5;
K = 3.49;
sigma = 0.09;
r = 0.08;
rstar = 0.06;
tau = 60/365;
N = 3;

{C,P} = optionCRR(S0,K,0.09,tau,r,rstar,N);

{Clb,Plb} = LookBackOption(S0,sigma,tau,r,rstar,N);

output file = exotic3.out reset;

print "option europeenne";
print C~P;
print "option look-back";
print Clb~Plb;

output off;
```

```
option europeenne
     0.065742529      0.044497969
option look-back
     0.078293046      0.067683532
```

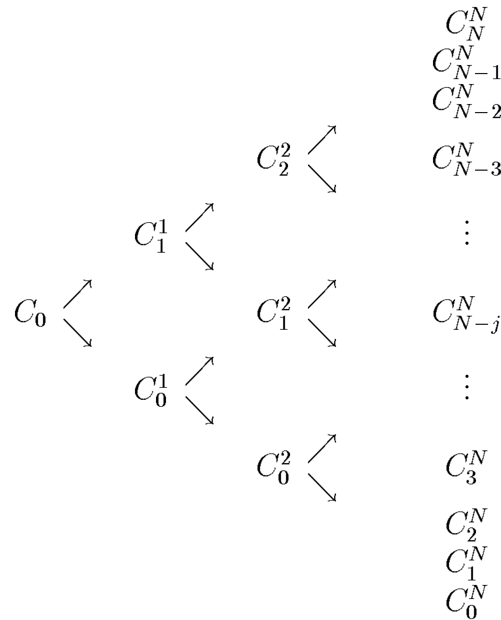
2.4.2 Les options américaines

Nous nous intéressons uniquement à la valorisation des options américaines dans le modèle de CRR. Pour les modèles en temps continu, nous pouvons utiliser les algorithmes d'approximation quadratique et de différences finies (voir GAUSS et la finance).

La technique de remontée de l'arbre. Nous rappelons que l'arbre binomial du sous-jacent se présente de la façon suivante :



L'arbre des valeurs intrinsèques de l'option est alors de la forme suivante :



Dans le cas des options américaines, la valeur intrinsèque de l'option est

$$C_j^m = \max\left(\left(u^j d^{m-j} S_0 - K\right)_+, e^{-r\frac{\tau}{N}}\left[\pi C_{j+1}^{m+1} + (1-\pi) C_j^{m+1}\right]\right)$$

Quelques remarques :

1. Nous utilisons un vecteur pour coder l'arbre binomial. Nous avons ainsi

$$\text{arbre}(S) = \begin{bmatrix} S_0 \\ uS_0 \\ dS_0 \\ u^2 S_0 \\ udS_0 \\ d^2 S_0 \\ \vdots \\ d^N S_0 \end{bmatrix}$$

2. Le calcul de $\pi C_{j+1}^{m+1} + (1-\pi)\, C_j^{m+1}$ est vectorisé en utilisant le principe suivant :

$$C^2 = \begin{bmatrix} C_2^2 \\ C_1^2 \\ C_0^2 \end{bmatrix} \longrightarrow \pi \begin{bmatrix} C_2^2 \\ C_1^2 \end{bmatrix} + (1-\pi) \begin{bmatrix} C_1^2 \\ C_0^2 \end{bmatrix}$$

   Cela revient à construire deux vecteurs. Pour le premier vecteur, nous éliminons la dernière composante et pour le second vecteur la première composante.

3. La valeur de l'option américaine correspond à la valeur intrinsèque C_0.

```
proc (2) = OptionAmericaine(S0,K,sigma,tau,r,rstar,N);
  local u,d,pi_,fa,Tree_S,Tree_O;
  local arbreS,arbreC,k1,k2,i,Si,j;
  local PayOff,Vi,Vi_,_Vi,C;

  arbreS = zeros((n+1)*(n+2)/2,1);
  arbreC = zeros((n+1)*(n+2)/2,1);

  /* Parametres du modele */

  u = exp(sigma*sqrt(tau/n));
  d = 1/u;
  pi_ = (exp((r-rstar)*tau/n)-d)/(u-d);
  fa = exp(-r*tau/n);

  /* Arbre du sous-jacent */

  k1 = 0;
  i = 0;
  do until i > n;
    k2 = k1+1+i;
    j = seqa(0,1,i+1);
    Si = S0*(d^j).*(u^rev(j));
    arbreS[1+k1:k2] = Si;
    k1 = k2;
    i = i + 1;
  endo;

  /* Algorithme de remontee de l'arbre */

  PayOff = Si - K;
  PayOff = PayOff.*(PayOff.>0);

  Vi = PayOff;
  k2 = (n+1)*(n+2)/2;
  k1 = k2 - n;
  arbreC[k1:k2] = Vi;
  i = n - 1;
  do while i >= 0;
    k2 = k1 - 1;
    k1 = k2 - i;
    Vi_ = trimr(Vi,0,1);
    _Vi = trimr(Vi,1,0);
```

```
    Vi = fa*(pi_*Vi_ + (1-pi_)*_Vi);

    Si = arbreS[k1:k2];
    PayOff = Si - K;
    PayOff = PayOff.*(PayOff.>0);
    Vi = maxc((PayOff~Vi)');

    arbreC[k1:k2] = Vi;
    i = i - 1;
  endo;

  C = arbreC[1];

  retp(C,arbreC);
endp;
```

Voici un exemple d'utilisation de la procédure `optionAmericaine`.

```
new;
library edf;

S0 = 3.5;
K = 3.51;
sigma = 0.19;
tau = 90/365;
r = 0.08;
rstar = 0.075;
N = 4;

{Cam,arbreC} = OptionAmericaine(S0,K,sigma,tau,r,rstar,N);
{Ceu,Peu} = OptionCRR(S0,K,sigma,tau,r,rstar,N);

output file = am1.out reset;

print "option americaine =" Cam;
print "option europeenne =" Ceu;

output off;

option americaine =       0.12078477
option europeenne =       0.12062903
```

Exercice 2.7 *Elaborer une procédure de valorisation de l'option européenne basée sur la technique de remontée de l'arbre. Nous rappelons que nous avons*

$$C_j^m = e^{-r\frac{\tau}{N}}\left[\pi C_{j+1}^{m+1} + (1-\pi)\, C_j^{m+1}\right]$$

Représentation graphique de l'arbre. Nous pouvons très facilement visualiser un arbre binomial en considérant un arrangement des valeurs de l'arbre dans une matrice carrée. Nous avons

$$\begin{array}{cccc}
\cdot & \cdot & \cdot & C_3^3 \\
\cdot & \cdot & C_2^2 & \cdot \\
\cdot & C_1^1 & \cdot & C_2^3 \\
C_0 & \cdot & C_1^2 & \cdot \\
\cdot & C_0^1 & \cdot & C_1^3 \\
\cdot & \cdot & C_0^2 & \cdot \\
\cdot & \cdot & \cdot & C_0^3
\end{array}$$

L'idée est ensuite de remplacer l'affichage de $\cdot$ par un caractère blanc. Dans la procédure, $\cdot$ correspond à une valeur manquante et nous modifions l'affichage de la valeur manquante à l'aide de l'instruction `Msym`.

```
proc (0) = AffichageArbre(x);
  local L,n,Arbre,j,xj,indx;

  L = rows(x);
  n = (-1+sqrt(1+8*L))/2;

  Arbre = miss(zeros(2*n-1,n),0);

  j = 1;
  do until j > n;
    xj = SelectionArbitrage(x,j);
    indx = seqa(n+1-j,2,j);
    Arbre[indx,j] = xj;
    j = j + 1;
  endo;

  Msym " ";

  print Arbre;

  Msym ".";

  retp;
endp;


proc (1) = SelectionArbitrage(x,j);
  local L,n,indx,xj;

  L = rows(x);
  n = (-1+sqrt(1+8*L))/2;

  indx = j*(j-1)/2;
  xj = x[1+indx:indx+j];

  retp(xj);
endp;
```

Voici la représentation de l'arbre des valeurs intrinsèques de l'option américaine précédente :

```
new;
library edf;

S0 = 3.5;
K = 3.51;
sigma = 0.19;
tau = 90/365;
r = 0.08;
rstar = 0.075;
N = 4;

{Cam,arbreC} = OptionAmericaine(S0,K,sigma,tau,r,rstar,N);

output file = am2.out reset;

print "Arbre des valeurs intrinseques de l'option americaine d'achat";
call affichageArbre(arbreC);

output off;
```

```
Arbre des valeurs intrinseques de l'option americaine d'achat

                                                                                 0.71685032
                                                               0.52208485
                                             0.33855348                          0.33629382
                        0.20627337                             0.16446692
     0.12078477                             0.080433732                          0.00000000
                       0.039336697                             0.00000000
                                             0.00000000                          0.00000000
                                                               0.00000000
                                                                                 0.00000000
```

Remarque. Nous utilisons la commande `outwidth` pour "élargir" l'espace d'affichage.

2.5 Le calcul de la volatilité implicite

Il existe plusieurs méthodes pour calculer numériquement la volatilité implicite d'une option. Les deux plus célèbres sont celles de la bisection et de Newton-Raphson.

2.5.1 L'algorithme de la bisection

Considérons l'équation unidimensionnelle

$$g(x) = y_0$$

Cela revient à chercher x_0 tel que

$$f(x_0) = 0$$

avec

$$f(x) = g(x) - y_0$$

Supposons que la fonction g est strictement croissante. Soient deux scalaires a et b tels que

$$f(a) < 0 < f(b)$$

La solution de l'équation vérifie alors la condition $a < x_0 < b$. Considérons le point c tel que $c = \frac{a+b}{2}$. Alors si $f(c) < 0$, la solution appartient à l'intervalle $[c, b]$. Dans le cas contraire, la solution appartient à l'intervalle $[a, c]$. Nous avons réduit de moitié l'intervalle qui contient la solution. Nous pouvons recommencer l'opération. Dans ce cas, la longueur du nouvel intervalle sera quatre fois plus petite que l'intervalle originel. En réitérant l'opération un certain nombre de fois, nous pouvons trouver un intervalle relativement petit qui encadre la solution. Quelques remarques :

1. La procédure suivante est vectorisée. Ceci explique pourquoi nous utilisons les opérateurs $E \times E$ plutôt que la structure de contrôle `if ... else ... endif`.

2. Vous pouvez employer une variable externe qui permet à l'utilisateur de fixer lui-même le critère de convergence.

3. Vous pouvez aussi calculer le nombre optimal d'itérations et vérifier que

$$N_{\text{iter}} = \left[\frac{\ln(b-a) - \ln \varepsilon}{\ln 2} \right]$$

4. La commande `ERRORLOG` permet d'afficher des messages d'erreur, qui sont stockés dans le fichier *gauss.err*.

```
proc bisection(f,a,b);
  local f:proc;
  local ya,yb,Nobs,c,yc,indx1,indx2;

  ya = f(a); yb = f(b);
  Nobs = rows(ya);

  if ( maxc(ya) > 0 ) or ( minc(yb) < 0 );
    ERRORLOG "erreur : mauvaise initialisation de a et b.";
    end;
  endif;

  do while maxc(abs(a-b)) > 0.0000001;  /* critere de convergence */
    c = (a+b)/2;
    yc = f(c);
    indx1 = yc.<0;
    indx2 = 1 - indx1;
    a = indx1.*c + indx2.*a;
    b = indx1.*b + indx2.*c;
  endo;
  c = (a+b)/2;
  retp(c);
endp;
```

Dans l'exemple suivant, nous cherchons, dans un premier temps, à résoudre l'équation

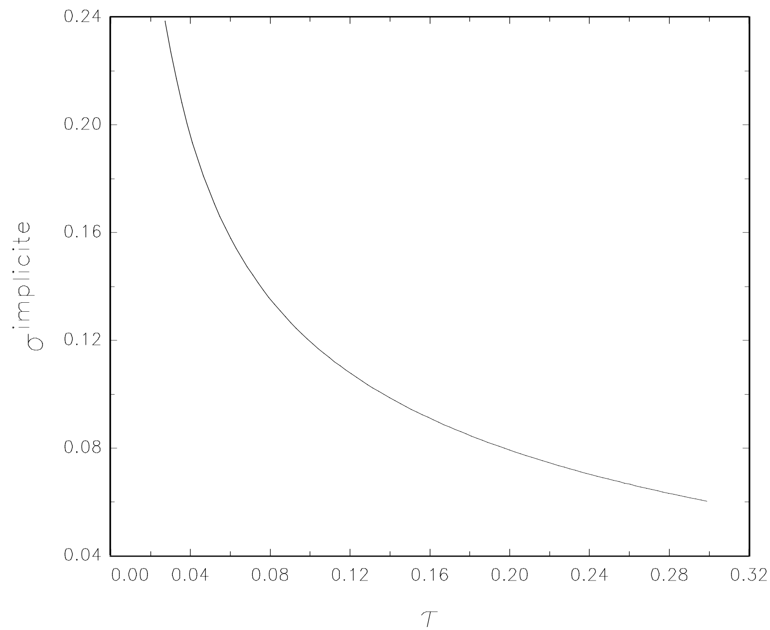
$$x^2 + \exp x = 3$$

Dans un second temps, nous illustrons le principe de vectorisation.

```
new;
library edf;

output file = bisect1.out reset;

proc fonction(x);
  retp(x^2 + exp(x) - 3);
endp;

solution = bisection(&fonction,0,4);

print "La solution est " solution;

proc fonction2(x);
  local y;
  y = x[1]^2 + exp(x[1]) - 3 |
      x[2]^3 + exp(x[2]) - 3  ;
  retp(y);
endp;

solution = bisection(&fonction2,0|0,4|4);

print "Les solutions sont " solution;

output off;

La solution est       0.83448410
Les solutions sont
      0.83448410
      0.86017227
```

Calculer la volatilité implicite d'une option est alors très simple. La seule difficulté est de choisir a et b correctement.

```
new;
library edf,pgraph;

S0 = 3.5;
K = 3.5;
tau = 60/365;
r = 0.08;
rstar = 0.06;

C = 0.056;

proc FonctionObjectif(sigma);
   retp(callGK(S0,K,sigma,tau,r,rstar) - C );
endp;

output file = volimp1.out reset;

print callGK(S0,K,0.01,tau,r,rstar);
print callGK(S0,K,0.50,tau,r,rstar);
```



Graphique 4:

```
volimp = bisection(&FonctionObjectif,0.01,0.50);

print "Volatilite implicite = " volimp;

output off;


tau = seqa(10/365,1/365,100);
e = ones(rows(tau),1);
volimp = bisection(&FonctionObjectif,0.01*e,0.50*e);

graphset;
  _pnum = 2; _pdate = ""; _paxht = 0.25;
  fonts("simplex simgrma");
  xlabel("\202t\201");
  ylabel("\202s\201[implicite]");
  graphprt("-c=1 -cf=volimp1.eps");
  xy(tau,volimp);

     0.013029378
      0.28506695
Volatilite implicite =      0.089539058
```

2.5.2 L'algorithme de Newton-Raphson

Basé sur une expansion de Taylor à l'ordre 1, celui-ci prend la forme suivante :

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}$$

Dans notre cas, f correspond à la différence entre le prix théorique de l'option (donné par la formule de Garman et Kohlhagen) et la valeur observée sur le marché et f' n'est rien d'autre que le Vega de l'option. Nous rappelons qu'il est égal à

$$s_0 e^{-r^\star \tau} \sqrt{\tau} \phi(d_1)$$

L'algorithme de Newton-Raphson nécessite une valeur de départ. Manaster et Koehler préconisent de prendre

$$\sigma_{\mathrm{sv}} = \frac{2}{\tau} \left| \ln\left(\frac{S_0}{K}\right) + (r - r^\star)\tau \right|$$

```
proc VolatiliteImplicite(S0,K,tau,r,rstar,C);
  local sv,sigma,w,d1,Vega;

  sigma = 2*abs(ln(S0./K) + (r-rstar).*tau)./tau;
  sv = 1 + sigma;

  do while maxc(abs(sigma-sv)) > 0.0000001;  /* critere de convergence */
    sv = sigma;
    w = sv.*sqrt(tau);
    d1 = ( ln(S0./K) + (r-rstar).*tau )./w + 0.5*w;
    Vega = S0.*exp(-rstar.*tau).*sqrt(tau).*pdfn(d1);
    sigma = sv - (callGK(S0,K,sv,tau,r,rstar) - C)./Vega;
  endo;

  retp(sigma);
endp;
```

Nous retrouvons bien les résultats précédents.

```
new;
library edf,pgraph;

S0 = 3.5;
K = 3.5;
tau = 60/365;
r = 0.08;
rstar = 0.06;

C = 0.056;

volimp = VolatiliteImplicite(S0,K,tau,r,rstar,C);

output file = volimp2.out reset;

print "Volatilite implicite = " volimp;
```

```
output off;

tau = seqa(10/365,1/365,100);
e = ones(rows(tau),1);
volimp = VolatiliteImplicite(S0,K,tau,r,rstar,C);

graphset;
  _pnum = 2; _pdate = ""; _paxht = 0.25;
  fonts("simplex simgrma");
  xlabel("\202t\201");
  ylabel("\202s\201[implicite]");
  xy(tau,volimp);

Volatilite implicite =      0.089539042
```

3 La méthode de Monte Carlo

3.1 Illustration du théorème de limite centrale

Soit X une variable aléatoire. Nous avons

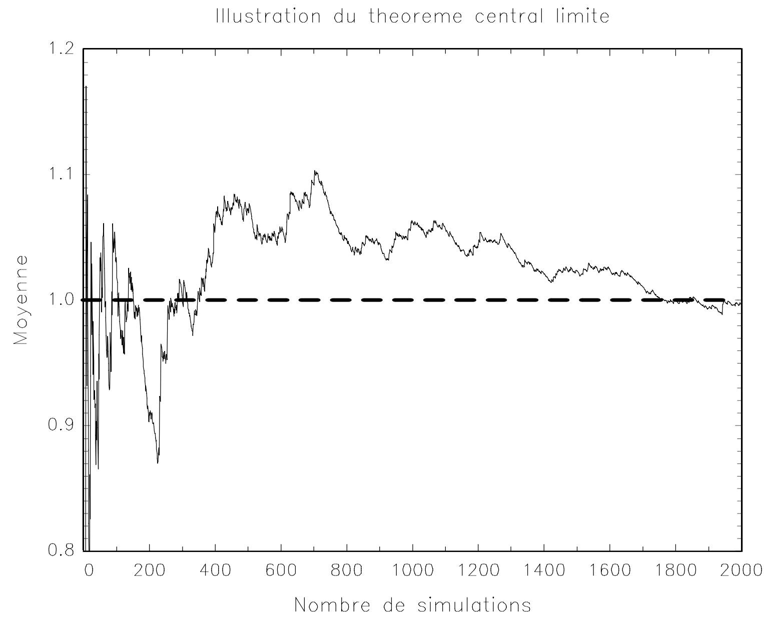
$$\bar{X}_N \xrightarrow[N\to\infty]{\text{pr}} E[X]$$

De nombreuses méthodes de Monte Carlo sont basées sur ce résultat. Nous pouvons illustrer ce théorème avec la loi de probabilité χ_1^2. Pour générerer des nombres aléatoires, nous utilisons la relation

$$\chi_1^2 = [\mathcal{N}(0,1)]^2$$

Nous avons l'impression que $\bar{X}_N$ a "tendance" à se rapprocher de la valeur unitaire lorsque le nombre de simulations augmente.

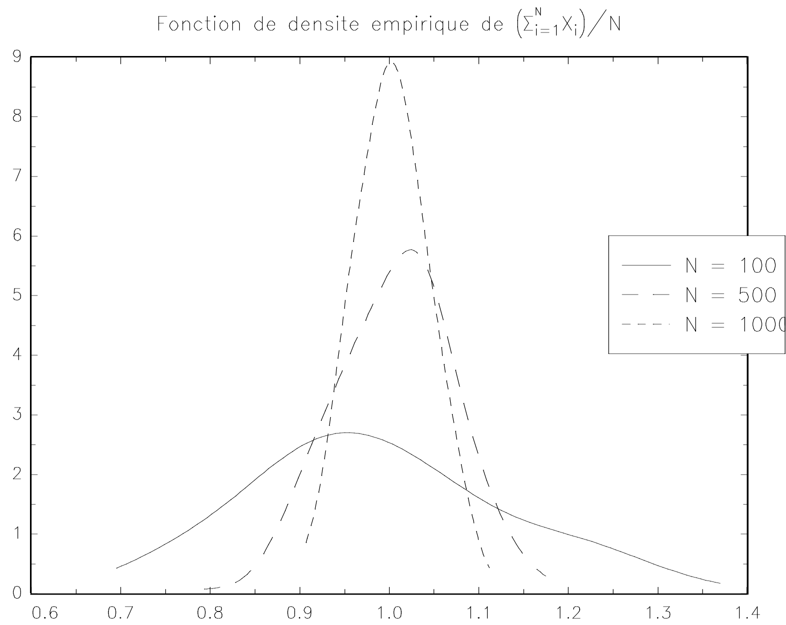
```
new;
library pgraph;

rndseed 123;

Ns = 2000;
n = seqa(1,1,Ns);

u = rndn(Ns,1);
chi2 = u^2;

xbar = cumsumc(chi2)./n;

graphset;
  _pnum = 2; _pdate = "";
  title("Illustration du theoreme central limite");
  xlabel("Nombre de simulations");
  ylabel("Moyenne");
  _pline = 1~1~0~1~Ns~1~1~5~10;
  ytics(0.8,1.2,0.1,10);
  graphprt("-c=1 -cf=tcl1.eps");
  xy(n,xbar);
```



Graphique 5:

Pour montrer la convergence, nous pouvons calculer les fonctions de densité empiriques (à l'aide de la méthode des noyaux) pour différentes valeurs de N. Nous vérifions que celles-ci sont plus concentrées autour de la moyenne lorsque N augmente.

```
new;
library pgraph,tsm,optmum;

rndseed 123;

Nr = 200;

Ns = 100;
u = rndn(Ns,Nr);
chi2 = u^2;
xbar1 = meanc(chi2);

Ns = 500;
u = rndn(Ns,Nr);
chi2 = u^2;
xbar2 = meanc(chi2);

Ns = 1000;
u = rndn(Ns,Nr);
chi2 = u^2;
xbar3 = meanc(chi2);

{x1,d1,f1,retcode} = Kernel(xbar1);
```



Graphique 6:

```
{x2,d2,f2,retcode} = Kernel(xbar2);
{x3,d3,f3,retcode} = Kernel(xbar3);

graphset;
  _pnum = 2; _pdate = ""; _pltype = 6|1|3;
  fonts("simplex simgrma");
  title("Fonction de densite empirique de "\
        "\202\034S\201]i=1[[[N]]X]i[\202v5\201N");
  _plegstr = "N = 100\000N = 500\000N = 1000";
  _plegctl = {2 6 7 3};
  graphprt("-c=1 -cf=tcl2.eps");
  xy(x1~x2~x3,d1~d2~d3);
```

3.2 Application au modèle CRR

Pour calculer les primes d'options dans le modèle CRR, nous avons uniquement besoin d'un générateur de nombres aléatoires de Bernoulli. Ceux-ci peuvent être facilement obtenus à partir de nombres aléatoires uniformes. C'est le principe de génération par inversion de la fonction de répartition.

```
new;

M = 3;

N = 10;
p = 0.25;
```

```
u = rndu(N,M);
bernoulli = u .<= p;
binomiale = sumc(bernoulli);

output file = mc_crr1.out reset;

print "simulation de nombres aleatoires de Bernoulli";
print bernoulli;
print "simulation de nombres aleatoires de la loi Binomiale";
print binomiale;

M = 250;

u = rndu(N,M);
bernoulli = u .<= p;
binomiale = sumc(bernoulli);
print "Moyenne theorique = " N*p;
print "Moyenne empirique = " meanc(binomiale);
print "Variance theorique = " N*p*(1-p);
print "Variance empirique = " stdc(binomiale)^2;

output off;
```

```
simulation de nombres aleatoires de Bernoulli

      0.00000000       0.00000000        1.0000000
      0.00000000       0.00000000       0.00000000
      0.00000000       0.00000000        1.0000000
      0.00000000       0.00000000       0.00000000
      0.00000000       0.00000000       0.00000000
      0.00000000       0.00000000       0.00000000
      0.00000000        1.0000000        1.0000000
      0.00000000       0.00000000       0.00000000
      0.00000000       0.00000000        1.0000000
      0.00000000       0.00000000        1.0000000
simulation de nombres aleatoires de la loi Binomiale

      0.00000000
       1.0000000
       5.0000000
Moyenne theorique =        2.5000000
Moyenne empirique =        2.5680000
Variance theorique =        1.8750000
Variance empirique =        1.7644337
```

Pour simuler la prime CRR, nous procédons de la façon suivante :

1. Nous générons des nombres aléatoires b_i de Bernoulli de paramètre π.
2. Nous simulons des trajectoires du sous-jacent sous la mesure de probabilité risque-neutre en utilisant la formule

$$S_{i+1} = [u \times b_i + d \times (1 - b_i)] \, S_i$$

3. Pour chaque trajectoire simulée s, nous calculons la valeur du pay-off g_s.

4. La prime simulée correspond alors à $e^{-r\tau}\sum_{s=1}^{S} g_s$.

Remarque. Pour simuler les trajectoires du sous-jacent, il n'est pas nécessaire d'utiliser une boucle. Dans la procédure, nous utilisons la commande vectorisée `cumprodc`, mais nous aurions aussi pu employer `recsercp`.

```
proc mcCRR(S0,K,sigma,tau,Sigma,r,rstar,N,Ns);
  local u,d,pi_,w,b,simulationS,ST,Sm,Prime,G;

  u = exp(sigma*sqrt(tau/n));
  d = 1/u;
  pi_ = (exp((r-rstar)*tau/n)-d)/(u-d);

  w = rndu(n,Ns);
  b = w .<= pi_;
  simulationS = S0*cumprodc(ones(1,Ns) | b*u + (1-b)*d);

  ST = simulationS[N+1,.]';
  Sm = meanc(simulationS);

  Prime = zeros(6,1);

  /* call europeen */
  G = (ST-K);
  G = G.*(G.>0);
  Prime[1] = meanc(G);

  /* put europeen */
  G = (K-ST);
  G = G.*(G.>0);
  Prime[2] = meanc(G);

  /* call asiatique Fixed Strike */
  G = (Sm-K);
  G = G.*(G.>0);
  Prime[3] = meanc(G);

  /* put asiatique Fixed Strike */
  G = (K-Sm);
  G = G.*(G.>0);
  Prime[4] = meanc(G);

  Prime = exp(-r*tau)*Prime;

  retp(Prime);
endp;
```

Voici une illustration de la procédure `mcCRR`.

```
new;
library edf;

S0 = 3.5;
K = 3.49;
sigma = 0.09;
r = 0.08;
rstar = 0.06;
tau = 60/365;
N = 3;

rndseed 123;
Ns = 10000;

{C,P} = optionCRR(S0,K,sigma,tau,r,rstar,N);
{Cas,Pas} = FixedStrikeOption(S0,K,sigma,tau,r,rstar,N);
Prime = mcCRR(S0,K,sigma,tau,Sigma,r,rstar,N,Ns);

output file = mc_crr2.out reset;

print ftos(C,"Valeur de l'option d'achat europeenne :  %lf",6,5);
print ftos(Prime[1],"Valeur simulee :                       %lf",6,5);
print ftos(P,"Valeur de l'option de vente europeenne : %lf",6,5);
print ftos(Prime[2],"Valeur simulee :                       %lf",6,5);
print ftos(Cas,"Valeur de l'option d'achat asiatique :   %lf",6,5);
print ftos(Prime[3],"Valeur simulee :                       %lf",6,5);
print ftos(Pas,"Valeur de l'option de vente asiatique :  %lf",6,5);
print ftos(Prime[4],"Valeur simulee :                       %lf",6,5);

output off;
```

```
Valeur de l'option d'achat europeenne :  0.06574
Valeur simulee :                         0.06664
Valeur de l'option de vente europeenne : 0.04450
Valeur simulee :                         0.04443
Valeur de l'option d'achat asiatique :   0.03671
Valeur simulee :                         0.03692
Valeur de l'option de vente asiatique :  0.02115
Valeur simulee :                         0.02126
```

3.3 La technique de réduction de variance

Il existe différentes méthodes pour accélérer la convergence de la méthode de Monte Carlo. La plus célèbre est sûrement la technique de réduction de variance basée sur les variables antithétiques. L'idée est d'obtenir à partir des même nombres aléatoires une seconde trajectoire du processus. Nous pouvons ainsi associer à g_s une valeur simulée antithétique $g_s^\star$. Nous définissons alors l'accélérateur de Monte Carlo par l'opération suivante

$$g_s \leftarrow \frac{g_s + g_s^\star}{2}$$

Cela revient à considérer la moyenne de la valeur simulée et de celle antithétique. Pour le modèle de CRR, nous pouvons utiliser le fait que si w est un nombre aléatoire uniforme $\mathcal{U}_{[0,1]}$, alors $1 - w$ l'est aussi.

```
proc (3) = mcCRR2(S0,K,sigma,tau,Sigma,r,rstar,N,Ns);
  local u,d,pi_,w,b,simulationS1,simulationS2,ST,Sm,Prime1,Prime2,G,Prime;

  u = exp(sigma*sqrt(tau/n));
  d = 1/u;
  pi_ = (exp((r-rstar)*tau/n)-d)/(u-d);

  w = rndu(n,Ns);
  b = w .<= pi_;
  simulationS1 = S0*cumprodc(ones(1,Ns) | b*u + (1-b)*d);

  w = 1 - w;       /* variable antithetique */
  b = w .<= pi_;
  simulationS2 = S0*cumprodc(ones(1,Ns) | b*u + (1-b)*d);

  ST = simulationS1[N+1,.]';
  Sm = meanc(simulationS1);

  Prime1 = zeros(6,1);

  /* call europeen */
  G = (ST-K);
  G = G.*(G.>0);
  Prime1[1] = meanc(G);

  /* put europeen */
  G = (K-ST);
  G = G.*(G.>0);
  Prime1[2] = meanc(G);

  /* call asiatique Fixed Strike */
  G = (Sm-K);
  G = G.*(G.>0);
  Prime1[3] = meanc(G);

  /* put asiatique Fixed Strike */
  G = (K-Sm);
  G = G.*(G.>0);
  Prime1[4] = meanc(G);

  Prime1 = exp(-r*tau)*Prime1;

  /* Reduction de la variance */

  ST = simulationS2[N+1,.]';
  Sm = meanc(simulationS2);

  Prime2 = zeros(6,1);

  /* call europeen */
  G = (ST-K);
  G = G.*(G.>0);
  Prime2[1] = meanc(G);
```

```
  /* put europeen */
  G = (K-ST);
  G = G.*(G.>0);
  Prime2[2] = meanc(G);

  /* call asiatique Fixed Strike */
  G = (Sm-K);
  G = G.*(G.>0);
  Prime2[3] = meanc(G);

  /* put asiatique Fixed Strike */
  G = (K-Sm);
  G = G.*(G.>0);
  Prime2[4] = meanc(G);

  Prime2 = exp(-r*tau)*Prime2;

  Prime = 0.5*(Prime1 + Prime2);

  retp(Prime,Prime1,Prime2);
endp;
```

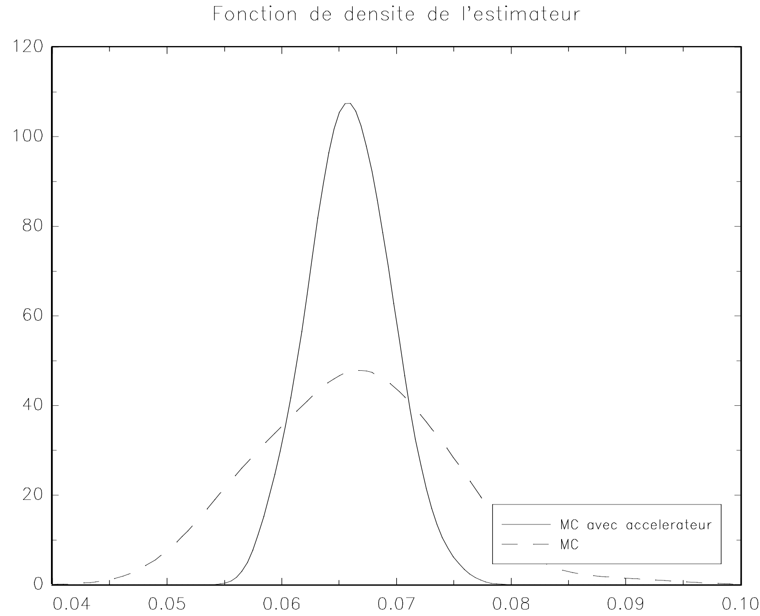
Nous reprenons l'exemple précédent.

```
new;
library edf;

S0 = 3.5;
K = 3.49;
sigma = 0.09;
r = 0.08;
rstar = 0.06;
tau = 60/365;
N = 3;

rndseed 123;
Ns = 10000;

{C,P} = optionCRR(S0,K,sigma,tau,r,rstar,N);
{Cas,Pas} = FixedStrikeOption(S0,K,sigma,tau,r,rstar,N);
{Prime,Prime1,Prime2} = mcCRR2(S0,K,sigma,tau,Sigma,r,rstar,N,Ns);

output file = mc_crr3.out reset;

print ftos(C,"Valeur de l'option d'achat europeenne :  %lf",6,5);
print ftos(Prime[1],"Valeur simulee :                           %lf",6,5);
print ftos(P,"Valeur de l'option de vente europeenne : %lf",6,5);
print ftos(Prime[2],"Valeur simulee :                           %lf",6,5);
print ftos(Cas,"Valeur de l'option d'achat asiatique :   %lf",6,5);
print ftos(Prime[3],"Valeur simulee :                           %lf",6,5);
print ftos(Pas,"Valeur de l'option de vente asiatique :  %lf",6,5);
print ftos(Prime[4],"Valeur simulee :                           %lf",6,5);
```

```
output off;

Valeur de l'option d'achat europeenne :  0.06574
Valeur simulee :                         0.06618
Valeur de l'option de vente europeenne : 0.04450
Valeur simulee :                         0.04493
Valeur de l'option d'achat asiatique :   0.03671
Valeur simulee :                         0.03691
Valeur de l'option de vente asiatique :  0.02115
Valeur simulee :                         0.02138
```

Le programme suivant illustre la différence de vitesse de convergence de la méthode de Monte Carlo avec et sans accélérateur.

```
new;
library edf,tsm,optmum,pgraph;

S0 = 3.5;
K = 3.49;
sigma = 0.09;
r = 0.08;
rstar = 0.06;
tau = 60/365;
N = 3;

rndseed 123;
Ns = 100;

Nr = 250;

mc = zeros(Nr,2);

i = 1;
do until i > Nr;
  {Prime,Prime1,Prime2} = mcCRR2(S0,K,sigma,tau,Sigma,r,rstar,N,Ns);
   mc[i,1] = Prime[1];
   mc[i,2] = Prime1[1];
   i = i + 1;
endo;

_Kernel[1] = 0.04; _Kernel[2] = 0.10;
{x1,d1,F1,retcode} = Kernel(mc[.,1]);
{x2,d2,F2,retcode} = Kernel(mc[.,2]);

graphset;
  _pnum = 2; _pdate = "";
  title("Fonction de densite de l'estimateur");
  _plegstr = "MC avec accelerateur\000MC";
  _plegctl = 1;
  graphprt("-c=1 -cf=mc_crr4.eps");
  xy(x1~x2,d1~d2);
```



Graphique 7:

3.4 Simulation d'équations différentielles stochastiques

3.4.1 Algorithme d'Euler-Maruyama

Considérons l'EDS :

$$\begin{cases} dX(t) & = & \mu(t, X(t))\, dt + \sigma(t, X(t))\, dW(t) \\ X(t_0) & = & x_0 \end{cases}$$

Le schéma d'Euler-Maruyama correspond à la discrétisation suivante du processus :

$$X(t_{i+1}) = X(t_i) + \mu(t_i, X(t_i))(t_{i+1} - t_i) + \sigma(t_i, X(t_i))\sqrt{(t_{i+1} - t_i)}\varepsilon_i$$

avec $\varepsilon_i \sim \mathcal{N}(0, 1)$. Dans le cas d'un pas de discrétisation constant $h = t_{i+1} - t_i$, nous avons

$$X(t_{i+1}) = X(t_i) + \mu(t_i, X(t_i))\, h + \sigma(t_i, X(t_i))\sqrt{h}\varepsilon_i$$

```
proc (2) = EulerMaruyama(x0,mu,sigma,t0,TT,N,Ns);
  local mu:proc,sigma:proc;
  local k,t,x,sqrt_k,u,i,ti,xi;

  k = (TT-t0)/(N-1);
  t = seqa(t0,k,N);
  x = zeros(N,Ns);

  x[1,.] = x0.*ones(1,Ns);
  sqrt_k = sqrt(k);
```

```
  u = rndn(N,Ns);

  i = 1;
  do until i > N-1;
    ti = t[i];
    xi = x[i,.];
    x[i+1,.] = xi + mu(ti,xi)*k + sqrt_k*sigma(ti,xi).*u[i+1,.];
    i = i + 1;
  endo;

  retp(t,x);
endp;
```

Nous rappelons que sous la mesure de probabilité risque-neutre $\mathbb{P}'$, nous avons

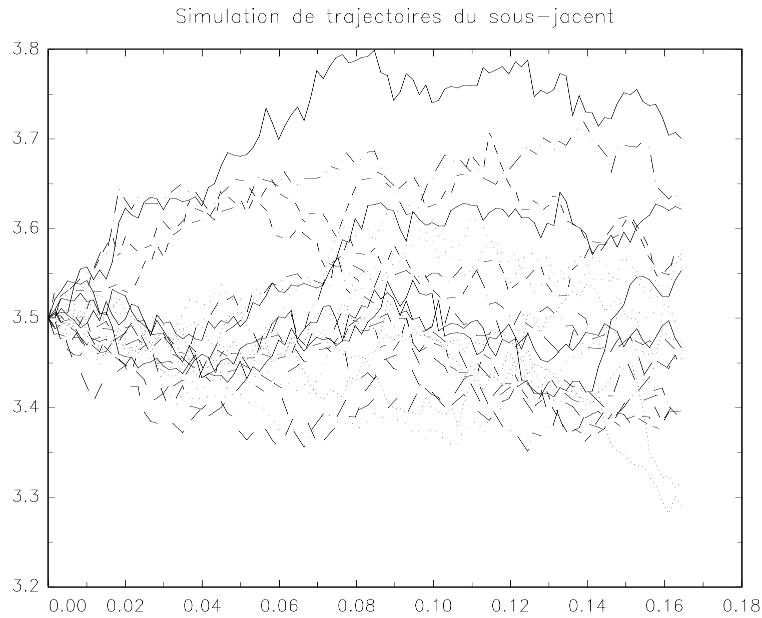
$$\begin{cases} dS(t) &= (r - r^{\star}) S(t)\, dt + \sigma S(t)\, dW'(t) \\ S(t_0) &= S_0 \end{cases}$$

Dans le programme qui suit, nous simulons vingt trajectoires du processus sous-jacent de l'option de change.

```
new;
library edf,pgraph;

S0 = 3.5;
K = 3.49;
sigma = 0.09;
r = 0.08;
rstar = 0.06;
tau = 60/365;

proc muProc(t,S);
  retp( (r-rstar)*S );
endp;

proc sigmaProc(t,S);
  retp( sigma*S );
endp;

{t,S} = EulerMaruyama(S0,&muProc,&sigmaProc,0,tau,100,20);

graphset;
  _pnum = 2; _pdate = "";
  title("Simulation de trajectoires du sous-jacent");
  graphprt("-c=1 -cf=eds1.eps");
  xy(t,S);
```

3.4.2 Application au modèle GK

Voici un exemple très simple de valorisation de l'option européenne par la méthode de Monte Carlo.



Graphique 8:

```
new;
library edf;

S0 = 3.5;
K = 3.49;
sigma = 0.09;
r = 0.08;
rstar = 0.06;
tau = 60/365;

proc muProc(t,S);
  retp( (r-rstar)*S );
endp;

proc sigmaProc(t,S);
  retp( sigma*S );
endp;

N = 60;    /* Nombre de points de discretisation, 1 simulation par jour */
Ns = 1000; /* Nombre de simulations */

{t,S} = EulerMaruyama(S0,&muProc,&sigmaProc,0,tau,N,Ns);

ST = S[N,.]';
G = ST - K;
G = G .* (G .> 0);
Cs = exp(-r*tau)*meanc(G);
```

```
C = callGK(S0,K,sigma,tau,r,rstar);

output file = eds2.out reset;

print ftos(C,"Valeur theorique : %lf",6,5);
print ftos(Cs,"Valeur simulee : %lf",6,5);

output off;

Valeur theorique : 0.06163
Valeur simulee : 0.06445
```

Pour les options exotiques, il convient d'être prudent avec la méthode de Monte Carlo. Nous rappelons que pour une option look-back, nous avons

$$G(T) = \max\left(0, S(T) - \min_{t_0 \leq t \leq T} S(t)\right)$$

Avec ce type d'option, il y a un risque de biais négatif de la valeur de l'option, car si nous prenons un pas de discrétisation trop grand, la valeur $\min_{t_0 \leq t \leq T} S(t)$ est sur-estimée.

```
new;
library edf,pgraph;

S0 = 3.5;
sigma = 0.09;
r = 0.08;
rstar = 0.06;
tau = 60/365;

proc muProc(t,S);
  retp( (r-rstar)*S );
endp;

proc sigmaProc(t,S);
  retp( sigma*S );
endp;

Ns = 1000; /* Nombre de simulations */


N = 20;    /* 1 simulation tous les 3 jours */

{t,S} = EulerMaruyama(S0,&muProc,&sigmaProc,0,tau,N,Ns);

ST = S[N,.]';
Smin = minc(S);
G = ST - Smin;
G = G .* (G .> 0);
Cs1 = exp(-r*tau)*meanc(G);


N = 120;    /* 2 simulations par jour */
```

```
{t,S} = EulerMaruyama(S0,&muProc,&sigmaProc,0,tau,N,Ns);

ST = S[N,.]';
Smin = minc(S);
G = ST - Smin;
G = G .* (G .> 0);
Cs2 = exp(-r*tau)*meanc(G);

output file = eds3.out reset;

print ftos(Cs1,"Valeur simulee (N = 20) : %lf",6,5);
print ftos(Cs2,"Valeur simulee (N = 120) : %lf",6,5);

output off;

Valeur simulee (N = 20) : 0.08813
Valeur simulee (N = 120) : 0.09791
```

3.5 Extension au cas des EDS multidimensionnelles

3.5.1 Le schéma de Taylor de forme forte à l'ordre 0.5

Considérons l'EDS multidimentionnelle :

$$\begin{cases} dX(t) &= \mu(t, X(t))\, dt + \Sigma(t, X(t))\, dW(t) \\ X(t_0) &= X_0 \end{cases}$$

avec $W(t)$ un processus de Wiener de covariance ρt. Le schéma de Taylor de forme forte à l'ordre 0.5 correspond à la discrétisation suivante du processus :

$$X(t_{i+1}) = X(t_i) + \mu(t_i, X(t_i))(t_{i+1} - t_i) + \Sigma(t_i, X(t_i))\sqrt{(t_{i+1} - t_i)}\varepsilon_i$$

avec $\varepsilon_i \sim \mathcal{N}(0, \rho)$. Dans le cas d'un pas de discrétisation constant $h = t_{i+1} - t_i$, nous avons

$$X(t_{i+1}) = X(t_i) + \mu(t_i, X(t_i))\, h + \Sigma(t_i, X(t_i))\sqrt{h}\varepsilon_i$$

Pour simuler ε_i, nous utilisons la propriété suivante :

$$\begin{aligned} \mathcal{N}(0, \rho) &= \mathbf{P}\mathcal{N}(0, I) \\ \mathbf{P} &= \operatorname{chol}(\rho) \end{aligned}$$

```
proc (2) = EDSm(X0,MU,SIGMA,RHO,t0,TT,N);
  local MU:proc,SIGMA:proc;
  local m,k,t,X,sqrt_k,P,u,i,ti,Xi;

  m = rows(X0);

  k = (TT-t0)/(N-1);
  t = seqa(t0,k,N);
  X = zeros(m,N);

  x[.,1] = X0;
  sqrt_k = sqrt(k);
```

```
  P = chol(RHO)';
  u = rndn(rows(RHO),N);
  u = P*u;

  i = 1;
  do until i > N-1;
    ti = t[i];
    Xi = X[.,i];
    X[.,i+1] = xi + MU(ti,Xi)*k + sqrt_k*SIGMA(ti,Xi)*u[.,i+1];
    i = i + 1;
  endo;

  retp(t,X');
endp;
```

3.5.2 Application au modèle de Hull et White

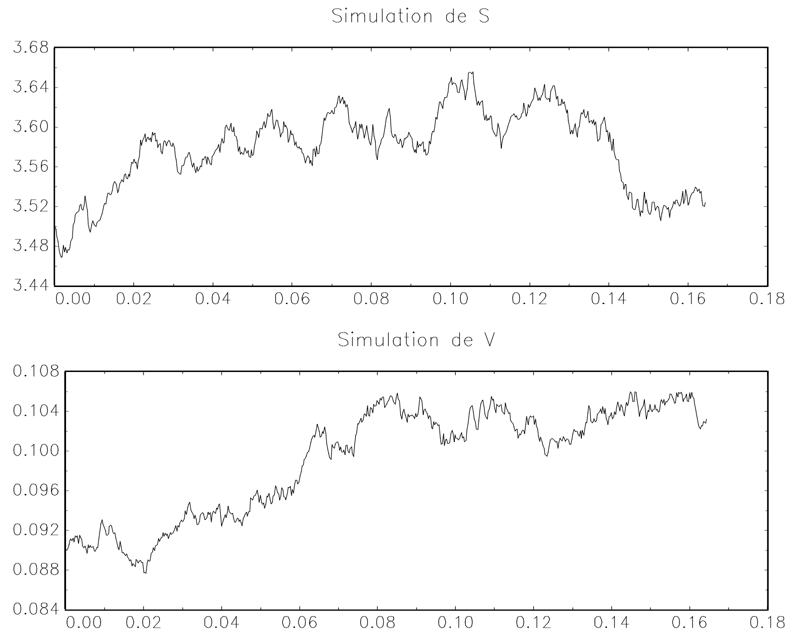
Considérons le processus le plus simple de Hull et White. Nous avons sous la mesure de probabilité risque-neutre

$$\left[\begin{array}{c} dS(t) \\ dV(t) \end{array} \right] = \left[\begin{array}{c} (r - r^{\star}) S(t) \\ \mu V(t) \end{array} \right] dt + \left[\begin{array}{cc} V(t) S(t) & 0 \\ 0 & \xi V(t) \end{array} \right] dW'(t)$$

avec

$$E\left[W'(t) W'(t)^{\top} \right] = \left[\begin{array}{cc} 1 & \rho \\ \rho & 1 \end{array} \right]$$

```
new;
library edf,pgraph;

S0 = 3.5;
V0 = 0.09;

K = 3.49;
r = 0.08;
rstar = 0.06;
tau = 60/365;

rho_ = -0.5;
nu = 0.9;
eta = 0.3;


RHO = ( 1 ~ rho_ ) | (rho_ ~ 1 );

proc muProc(t,X);
  local S,V,mu;
  S = X[1];
  V = X[2];
  mu = (r-rstar)*S |
        nu*V       ;
  retp( mu );
endp;
```



Graphique 9:

```
proc sigmaProc(t,X);
  local S,V,sig;
  S = X[1];
  V = X[2];
  sig = ( V*S ~ 0     ) |
        ( 0   ~ eta*V) ;
  retp( sig );
endp;

N = 600;
{t,X} = EDSm(S0|V0,&muProc,&sigmaProc,RHO,0,tau,N);

graphset;
  _pnum = 2; _pnumht = 0.25; _ptitlht = 0.30; _pdate = "";
  begwind;
  window(2,1,0);
  setwind(1);
    title("Simulation de S");
    xy(t,X[.,1]);
  setwind(2);
    title("Simulation de V");
    xy(t,X[.,2]);
    graphprt("-c=1 -cf=hw1.eps");
  endwind;
```

3.5.3 Valorisation des options dans les modèles de volatilité stochastique

Reprener le programme précédent et valoriser l'option de change européenne. Vous pouvez vectoriser le code en remarquant que la décomposition de Cholesky est

$$\mathbf{P} = \begin{bmatrix} 1 & 0 \\ \rho & \sqrt{1-\rho^2} \end{bmatrix}$$

4 Optimisation

4.1 La programmation quadratique

```
/*
**> QProg
**
**  Purpose:  solves the quadratic programming problem
**
**  Format:
**            { x,u1,u2,u3,u4,ret } = QProg( start,q,r,a,b,c,d,bnds );
**
**
**  Input: start   Kx1 vector,  starting values
**
**             q   KxK matrix,  model coefficient matrix
**
**             r   Kx1 vector,  model constant vector
**
**             a   MxK matrix,  equality constraint coefficient matrix
**                 if no equality constraints in model, set to zero
**
**             b   Mx1 vector,  equality constraint constant vector
**                 if set to zero and M > 1, b is set to Mx1 vector
**                 of zeros
**
**             c   NxK matrix,  inequality constraint coefficient matrix
**                  if no inequality constraints in model, set to zero
**
**             d   Nx1 vector,  inequality constraint constant vector
**                 if set to zero and N > 1, d is set to Mx1 vector
**                 of zeros
**
**          bnds   Kx2 vector, bounds on x, the first column contains
**                 the lower bounds on x, and the second column the
**                 upper bounds, if zero bounds for all elements of x
**                 are set to the plus and minus _qpbgnum
**
**  Output:  x   Kx1 vector,  coefficients at the minimum of the function
**
**          u1   Mx1 vector,  Lagrangian coefficients of equality constraints
**
**          u2   Nx1 vector,  Lagrangian coefficients of inequality constraints
**
**          u3   Kx1 vector,  Lagrangian coefficients of lower bounds
```

```
**
**          u4   Kx1 vector,  Lagrangian coefficients of upper bounds
**
**          ret  return code:  0, successful termination
**                             1, max iterations exceeded
**                             2, machine accuracy is insufficient to
**                                maintain decreasing function values
**                             3, model matrices not conformable
**                            <0, active constraints inconsistent
**
**  Globals:  _qprog_maxit - scalar, maximum number of iterations,
**            default = 1000
**
**  Remarks:  QProg solves the standard quadratic programming problem:
**
**         minimize  0.5 * x'Qx - x'R
**
**         subject to constraints,
**
**             Ax  = B
**             Cx >= D
**
**         and bounds,
**
**             bnds[.,1] <= x <= bnds[.,2]
**
*/
```

4.2 L'algorithme de quasi-Newton

```
/* >proc QNewton
**
**  Format:  { x,f,g,retcode } = QNewton(&fct,x0)
**
**  Input:   &fct     pointer to a procedure that computes the function to
**                    be minimized.  This procedure must have one input
**                    argument, a vector of parameter values, and one
**                    output argument, the value of the function evaluated
**                    at the input vector of parameter values.
**
**             x0     Kx1 vector of start values
**
**  Output:     x     Kx1 vector of parameters at minimum
**
**              f     scalar function evaluated at x
**
**              g     Kx1 gradient evaluated at x
**
**        retcode     return code:
**
**                       0   normal convergence
**                       1   forced exit
**                       2   maximum number of iterations exceeded
```

```
**                        3   function calculation failed
**                        4   gradient calculation failed
**                        5   step length calculation failed
**                        6   function cannot be evaluated at initial
**                            parameter values
**
**  Globals:  _qn_RelGradTol    scalar, convergence tolerance for relative
**                              gradient of estimated coefficients.
**                              Default = 1e-5.
**
**            _qn_GradProc      scalar, pointer to a procedure that computes
**                              the gradient of the function with respect to
**                              the parameters. This procedure must have a
**                              single input argument, a Kx1 vector of parameter
**                              values, and a single output argument, a Kx1
**                              vector of gradients of the function with respect
**                              to the parameters evaluated at the vector of
**                              parameter values.
**
**            _qn_MaxIters      scalar, maximum number of iterations.
**                              Default = 1e+5. Termination can be forced
**                              by pressing C on the keyboard.
**
**            _qn_PrintIters    scalar, if 1, print iteration information.
**                              Default = 0.  Can be toggled during iterations
**                              by pressing P on the keyboard.
**
**            _qn_ParNames      Kx1 vector, labels for parameters
**
**          _qn_RandRadius      scalar, If zero, no random search is attempted.  If nonzero
**                              it is the radius of random search which is invoked whenever
**                              the usual line search fails.  Default = .01.
**
**            __output          scalar, if 1, prints results.
**
**
**  Remarks:  Pressing C on the keyboard will terminate iterations, and
**            pressing P will toggle iteration output.
**
**            QNewton is recursive, that is, it can call a version of itself
**            with another function and set of global variables,
*/
```

4.3 L'optimisation non linéaire avec contraintes non linéaires d'égalité et d'inégalité

```
/*> sqpSolve
**
**  Purpose:  solve the nonlinear programming problem
**
**  Format:   { x,f,lagr,retcode } = sqpSolve(&fct,start)
**
**
```

```
**  Input:    &fct      pointer to a procedure that computes the function to
**                      be minimized.  This procedure must have one input
**                      argument, a vector of parameter values, and one
**                      output argument, the value of the function evaluated
**                      at the input vector of parameter values.
**
**           start      Kx1 vector of start values
**
**
**  Output:      x      Kx1 vector of parameters at minimum
**
**               f      scalar, function evaluated at x
**
**            lagr      vector, created using VPUT.  Contains the Lagrangean
**                      for the constraints.  The may be extracted with the
**                      VREAD command using the following strings:
**
**                        "lineq"    - Lagrangeans of linear equality
                                          constraints,
**                        "nlineq"   - Lagrangeans of nonlinear equality
                                          constraints
**                        "linineq"  - Lagrangeans of linear inequality
                                          constraints
**                        "nlinineq" - Lagrangeans of nonlinear inequality
**                                       constraints
**                        "bounds"   - Lagrangeans of bounds
**
**                      Whenever a constraint is active, its associated
**                      Lagrangean will be nonzero.
**
**
**         retcode      return code:
**
**                         0   normal convergence
**                         1   forced exit
**                         2   maximum number of iterations exceeded
**                         3   function calculation failed
**                         4   gradient calculation failed
**                         5   Hessian calculation failed
**                         6   line search failed
**                         7   error with constraints
**
**  Input Globals:
**
**   _sqp_A            MxK matrix, linear equality constraint coefficients
**   _sqp_B            Mx1 vector, linear equality constraint constants
**
**                      These globals are used to specify linear equality
**                      constraints of the following type:
**
**                         _sqp_A * X = _sqp_B
**
**                      where X is the Kx1 unknown parameter vector.
```

```
**
**   _sqp_EqProc       scalar, pointer to a procedure that computes
**                      the nonlinear equality constraints.  For example,
**                      the statement:
**
**                          _sqp_EqProc = &eqproc;
**
**                      tells CO that nonlinear equality constraints
**                      are to be placed on the parameters and where the
**                      procedure computing them is to be found.
**                         The procedure must have one input argument, the
**                      Kx1 vector of parameters, and one output argument,
**                      the Rx1 vector of computed constraints that are
**                      to be equal to zero.  For example, suppose that
**                      you wish to place the following constraint:
**
**                         P[1] * P[2] = P[3]
**
**                      The proc for this is:
**
**                         proc eqproc(p);
**                            retp(p[1]*[2]-p[3]);
**                         endp;
**
**   _sqp_C             MxK matrix, linear inequality constraint coefficients
**   _sqp_D             Mx1 vector, linear inequality constraint constants
**
**                       These globals are used to specify linear inequality
**                       constraints of the following type:
**
**                         _sqp_C * X >= _sqp_D
**
**                       where X is the Kx1 unknown parameter vector.
**
**   _sqp_IneqProc     scalar, pointer to a procedure that computes
**                      the nonlinear inequality constraints.  For example
**                      the statement:
**
**                           _sqp_EqProc = &ineqproc;
**
**                      tells CO that nonlinear equality constraints
**                      are to be placed on the parameters and where the
**                      procedure computing them is to be found.
**                      The procedure must have one input argument, the
**                      Kx1 vector of parameters, and one output argument,
**                      the Rx1 vector of computed constraints that are
**                      to be equal to zero.  For example, suppose that
**                      you wish to place the following constraint:
**
**                         P[1] * P[2] >= P[3]
**
**                      The proc for this is:
**
```

```
**                          proc ineqproc(p);
**                             retp(p[1]*[2]-p[3]);
**                          endp;
**
**    _sqp_Bounds        Kx2 matrix, bounds on parameters.  The first column
**                        contains the lower bounds, and the second column the
**                        upper bounds.  If the bounds for all the coefficients
**                        are the same, a 1x2 matrix may be used.
**                        Default = { -1e256 1e256 }
**
**    _sqp_GradProc      scalar, pointer to a procedure that computes the
**                        gradient of the function with respect to the
**                        parameters.  For example, the statement:
**
**                              _sqp_GradProc=&gradproc;
**
**                        tells CO that a gradient procedure exists as well
**                        where to find it.  The user-provided procedure has
**                        two input arguments, a Kx1 vector of parameter values
**                        and an NxP matrix of data.  The procedure returns a
**                        single output argument, an NxK matrix of gradients
**                        of the log-likelihood function with respect to the
**                        parameters evaluated at the vector of parameter values.
**
**                        Default = 0, i.e., no gradient procedure has been
**                        provided.
**
**    _sqp_HessProc      scalar, pointer to a procedure that computes the
**                        hessian, i.e., the matrix of second order partial
**                        derivatives of the function with respect to the
**                        parameters.  For example, the instruction:
**
**                             _sqp_HessProc=&hessproc;
**
**                        will tell OPTMUM that a procedure has been provided
**                        for the computation of the hessian and where to find
**                        it.  The procedure that is provided by the user must
**                        have two input arguments, a Px1 vector of parameter
**                        values and an NxK data matrix.  The procedure returns
**                        a single output argument, the PxP symmetric matrix of
**                        second order derivatives of the function evaluated at
**                        the parameter values.
**
**
**
**    _sqp_MaxIters      scalar, maximum number of iterations. Default = 1e+5.
**                       Termination can be forced by pressing C on the keyboard
**
**    _sqp_DirTol        scalar, convergence tolerance for gradient of estimated
**                        coefficients.  Default = 1e-5.  When this criterion has
**                        been satisifed NLPSolve will exit the iterations.
**
**    _sqp_ParNames      Kx1 character vector,  parameter names
```

```
**
**   _sqp_PrintIters   scalar, if nonzero, prints iteration information.
**                     Default = 0.  Can be toggled during iterations by
**                     pressing P on the keyboard.
**
**   _sqp_FeasibleTest scalar, if nonzero, parameters are tested for feasibility
**                      before computing function in line search.  If function
**                      is defined outside inequality boundaries, then this test
**                      can be turned off.
**
**   _sqp_RandRadius   scalar, If zero, no random search is attempted.  If
**                     nonzero, it is the radius of random search which is
**                     invoked whenever the usual line search fails.
**                     Default = .01.
**
**    __output          scalar, if nonzero, results are printed.  Default = 0.
**
**
**  Remarks:  Pressing C on the keyboard will terminate iterations, and
**            pressing P will toggle iteration output.
**
**            NLPSolve is recursive, that is, it can call a version of itself
**            with another function and set of global variables,
*/
```

5 Value at Risk

La notion de Value at Risk (VaR) est une notion relativement simple. La mise en place des techniques de base pose peu de difficultés en terme de programmation. L'apparition de techniques beaucoup plus sophistiquées nécessite cependant l'utilisation d'un langage de programmation efficace. Une autre difficulté concerne la gestion des données.

5.1 Le cas linéaire gaussien

Le plus simple pour comprendre la VaR est de calculer celle-ci dans le cas linéaire gaussien. Considérons le cas général N actifs. Nous notons $\mathbf{p}$ le vecteur aléatoire des prix des actifs. Nous avons

$$\mathbf{p} \sim \mathcal{N}(\boldsymbol{\mu}, \Sigma)$$

5.1.1 Génération de nombres aléatoires gaussiens

Dans les exemples qui suivent, nous utilisons des données simulées. Soit une matrice Q telle que

$$\Sigma = QQ^{\top}$$

Nous avons alors la propriété suivante :

$$\mathcal{N}(\boldsymbol{\mu}, \Sigma) = \boldsymbol{\mu} + Q\mathcal{N}(\mathbf{0}, I)$$

Trois algorithmes basés sur cette propriété sont présentés dans l'annexe E du document technique de RiskMetrics™ :

1. La décomposition de Cholesky implique que

$$Q = \mathbf{P}$$

2. Avec la décomposition vecteurs propres/valeurs propres $\Sigma = V\Lambda V^\top$, nous avons

$$Q = V\Lambda^{1/2}$$

3. Enfin, si nous employons la décomposition en valeurs singulières $\Sigma = USV^\top$, nous avons

$$Q = US^{1/2}$$

```
new;

output file = rndmn.out reset;

let MU = 1 2 3;
let SIGMA[3,3] =  4  2 -1
                  2  6  2
                 -1  2  3;

Ns = 10000;
N = rows(mu);

/* simulation des nombres aleatoires N(0,I) */

epsilon = rndn(N,Ns);


/* Decomposition de Cholesky */

P = chol(SIGMA)';
Q = P;
y = mu + Q*epsilon;
print vcx(y');


/* Decomposition vecteurs propres/valeurs propres */

{lambda,V} = eighv(SIGMA);
LAMBDA = diagrv(eye(N),lambda);
Q = V*sqrt(LAMBDA);
y = mu + Q*epsilon;
print vcx(y');


/* Decomposition valeurs singulieres */

{U,S,V} = svdusv(SIGMA);
Q = U*sqrt(S);
y = mu + Q*epsilon;
print vcx(y');

output off;
```

```
       3.8940801        1.9620788      -0.98557876
       1.9620788        5.9677492        1.9733583
     -0.98557876        1.9733583        2.9698403

       4.0185606        2.0272175      -0.99393407
       2.0272175        5.9508918        1.9858337
     -0.99393407        1.9858337        2.9649171

       3.9417917        1.9320398       -1.0079588
       1.9320398        5.8329444        1.9669788
      -1.0079588        1.9669788        3.0025541
```

Dans la suite du document, nous utilisons la procédure suivante pour générer des nombres aléatoires multidimensionnels :

```
proc (1) = rndmn(mu,SIGMA,Ns);
  local dim,u,Pchol;
  local oldtrap;

  dim = rows(mu);

  oldtrap = trapchk(1);
  trap 1,1;
  Pchol = chol(SIGMA)';
  trap oldtrap,1;
  if scalerr(Pchol);
    ERRORLOG "error: SIGMA is not a positive definite matrix.";
    retp(error(0));
  endif;

  u = mu + Pchol*rndn(dim,Ns);

  retp(u');
endp;
```

Voici un premier exemple d'utilisation de la procédure avec la directive de compilation `#include`.

```
new;
#include rndmn.src;

rndseed 123;

output file = var1.out reset;   @<----->@

let MU = 1 2 3;
let SIGMA[3,3] =  4  2 -1
                  2  6  2
                 -1  2  3;
var = diag(sigma);
stderr = sqrt(var);
Mcorr = SIGMA ./ stderr ./ stderr';

print "Matrice de correlation : "; print Mcorr;
```

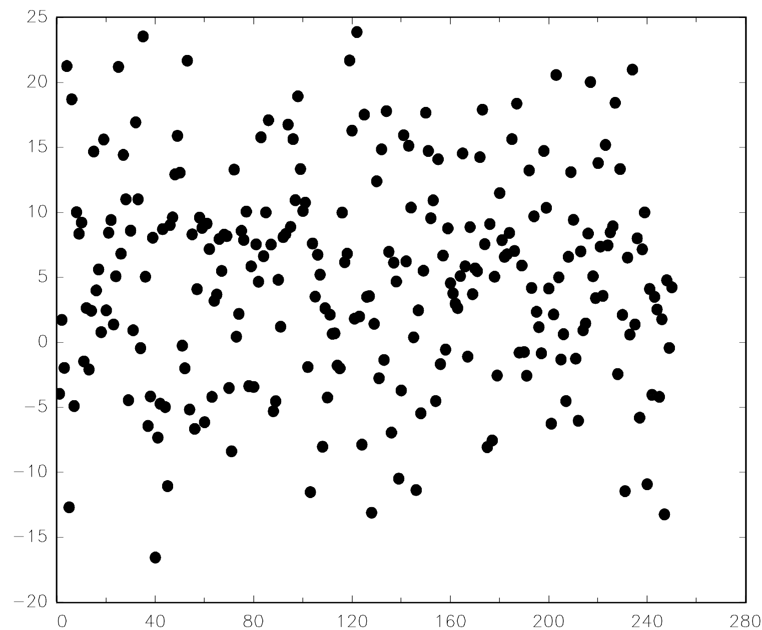
```
output off;                    @<----->@

u = rndmn(mu,sigma,100);
m = meanc(u);
V = vcx(u);
C = corrx(u);

print "Vecteur des moyennes estimees : "; print m;
print "Matrice de covariance estimee : "; print V;

output file = var1.out on;      @<----->@

print "Matrice de correlation estimee : "; print C;

output off;                    @<----->@

Matrice de correlation :

       1.0000000       0.40824829      -0.28867513
      0.40824829        1.0000000       0.47140452
     -0.28867513       0.47140452        1.0000000
Matrice de correlation estimee :

       1.0000000       0.51886647      -0.24234757
      0.51886647        1.0000000       0.38017606
     -0.24234757       0.38017606        1.0000000
```

Nous considérons un portefeuille dont le vecteur des stratégies est

$$\theta = \begin{bmatrix} 1 \\ 3 \\ -1 \end{bmatrix}$$

Nous supposons que la valeur actuelle du portefeuille est de 4 Francs. Nous simulons 250 réalisations de la valeur du portefeuille pour la période suivante.

```
new;
library pgraph;

#include rndmn.src;

rndseed 123;

let MU = 1 2 3;
let SIGMA[3,3] =  4  2 -1
                  2  6  2
                 -1  2  3;
theta = 1|3|-1;
ValeurPortefeuille_t0 = 4;

Nobs = 250;
ValeurPortefeuille_t1 = rndmn(mu,sigma,Nobs)*theta;

graphset;
```



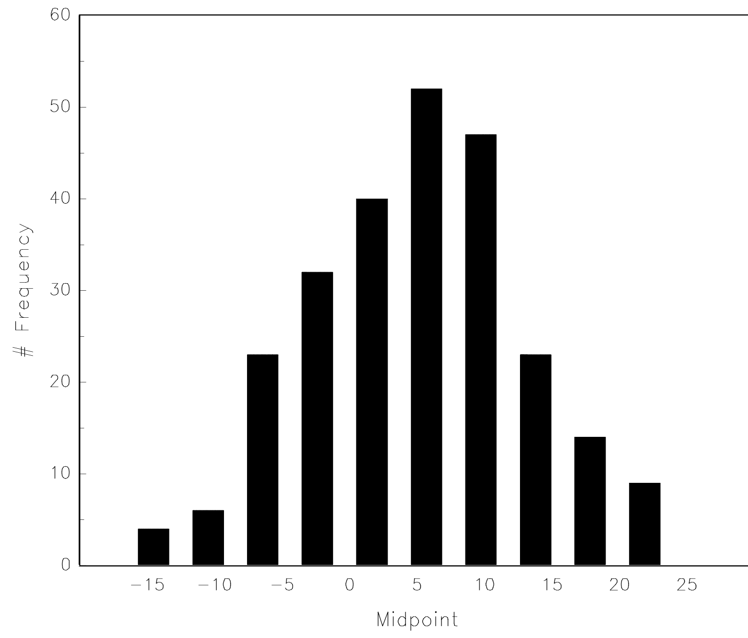
Graphique 10:

```
  _pdate = ""; _pnum = 2;
  _plctrl = -1; _pstype = 8;
  graphprt("-c=1 -cf=var2a.eps");
  xy(seqa(1,1,Nobs),ValeurPortefeuille_t1);

graphset;
  _pdate = ""; _pnum = 2;
  graphprt("-c=1 -cf=var2b.eps");
  call hist(ValeurPortefeuille_t1,10);

graphset;
  _pdate = ""; _pnum = 2;
  _pboxctl = 0.5 |
             0 |
             1 ;
  graphprt("-c=1 -cf=var2c.eps");
  box(0,ValeurPortefeuille_t1);
```

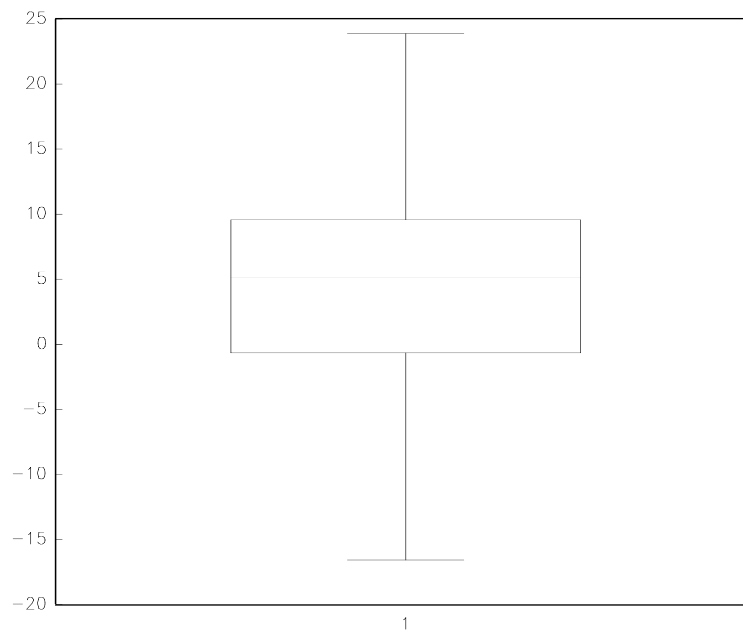
5.1.2 La technique de l'inversion de fonction de répartition

Le calcul de la *VaR* n'est rien d'autre que l'inversion de la fonction de répartition théorique de la **variation** de la valeur du portefeuille pour un seuil donné α (en général 1, 2.5 ou 5%). Il est souvent difficile de calculer la fonction de répartition théorique, mais nous pouvons l'estimer avec des méthodes relativement simples comme celle du noyau.

```
new;
```



Graphique 11:



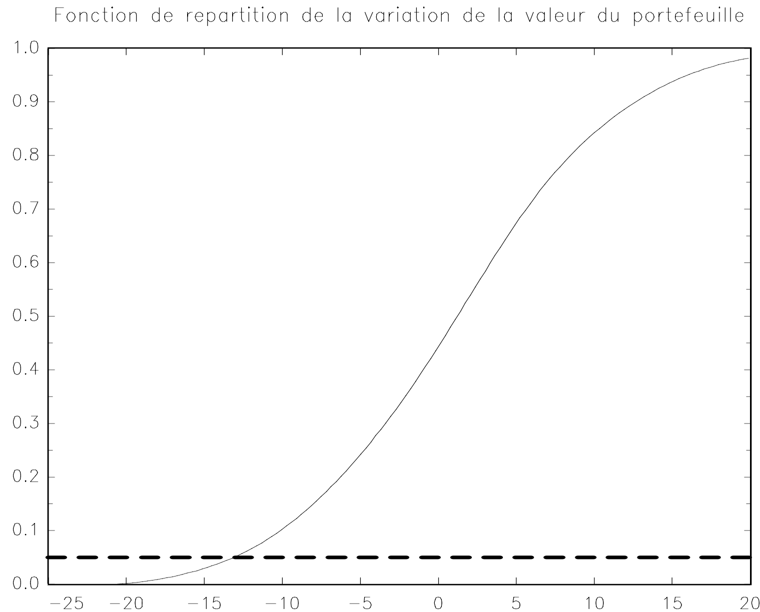
Graphique 12:

```
library tsm,optmum,pgraph;

#include rndmn.src;

rndseed 123;

let MU = 1 2 3;
let SIGMA[3,3] =  4  2 -1
                  2  6  2
                 -1  2  3;
theta = 1|3|-1;
ValeurPortefeuille_t0 = 4;

Nobs = 250;
ValeurPortefeuille_t1 = rndmn(mu,sigma,Nobs)*theta;

deltaP = ValeurPortefeuille_t1 - ValeurPortefeuille_t0;

{absc,dens,repart,retcode} = Kernel(deltaP);

alpha = 0.05;

graphset;
  _pdate = ""; _pnum = 2;
  title("Fonction de repartition de la variation de la valeur du portefeuille");
  _pline = 1~1~-30~alpha~30~alpha~1~5~10;
  graphprt("-c=1 -cf=var3.eps");
  xy(absc,repart);
```

5.1.3 Calcul de la *VaR* par la méthode des quantiles

Nous pouvons utiliser la méthode de Monte Carlo et la commande `quantile` pour calculer très simplement la *VaR*.

```
new;

#include rndmn.src;

rndseed 123;

let MU = 1 2 3;
let SIGMA[3,3] =  4  2 -1
                  2  6  2
                 -1  2  3;
theta = 1|3|-1;
ValeurPortefeuille_t0 = 4;

Nobs = 2500;
ValeurPortefeuille_t1 = rndmn(mu,sigma,Nobs)*theta;

deltaP = ValeurPortefeuille_t1 - ValeurPortefeuille_t0;
```



Graphique 13:

```
output file = var4.out reset;

call dstat(0,deltaP);

e = {0.01, 0.025, 0.05, 0.5, 0.95, 0.975, 0.99};

q = quantile(deltaP,e);
nom = "1%" | "2.5%" | "5%" | "Mediane" | "95%" | "97.5%" | "99%";

print; print;
print "Quantiles estimes";
print "-----------------";
call printfmt(nom~q,0~1);

alpha = 0.05;
VaR = abs(quantile(deltaP,alpha));
print;
print ftos(VaR,"Value at Risk (0.05) = %lf Francs",5,4);

output off;
```

```
--------------------------------------------------------------------------------
Variable        Mean    Std Dev    Variance   Minimum   Maximum     Valid Missing
--------------------------------------------------------------------------------
X1            0.1353     8.0695     65.1161  -26.7665   24.9883      2500    0

Quantiles estimes
```

```
-----------------
      1%       -18.721006
    2.5%       -16.077599
      5%       -13.109183
 Mediane     -0.021859735
     95%        13.672428
   97.5%        15.712769
     99%        19.134416

Value at Risk (0.05) = 13.1092 Francs
```

Dans le cas linéaire gaussien, il n'est pas nécessaire d'utiliser la méthode de Monte carlo. Nous pouvons calculer celle-ci en employant la fonction gaussienne inverse.

```
new;

#include rndmn.src;

rndseed 123;

let MU = 1 2 3;
let SIGMA[3,3] =  4  2 -1
                  2  6  2
                 -1  2  3;
theta = 1|3|-1;
ValeurPortefeuille_t0 = 4;

Moyenne = theta'MU - ValeurPortefeuille_t0;
Variance = theta'SIGMA*theta;

alpha = 0.05;
VaR = abs(Moyenne + cdfni(alpha)*sqrt(Variance));

output file = var5.out reset;

print ftos(VaR,"Value at Risk (0.05) = %lf Francs",5,4);

output off;

Value at Risk (0.05) = 13.0556 Francs
```

5.2 Le cas non linéaire (ou la *VaR* des options)

Le calcul de la *VaR* est plus difficile dans le cas non linéaire. Il existe principalement deux méthodes pour résoudre ce type de problèmes : l'approximation de Taylor et la méthode de Monte Carlo.

5.2.1 Les sources de variation des prix des options

Celles-ci sont principalement le prix du sous-jacent et la maturité de l'option.

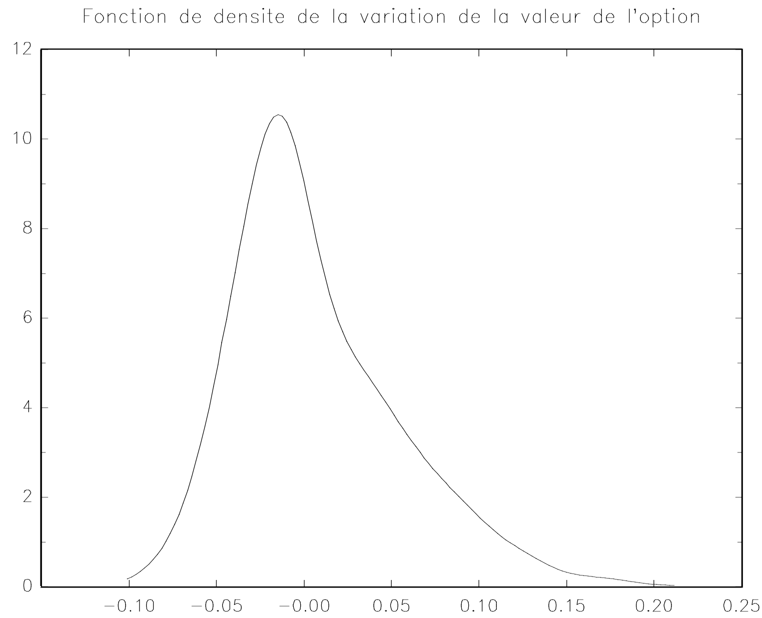
```
new;
library edf;
```

```
S0 = 3.5; K = 3.51; sigma = 0.19; tau = 90/365;
r = 0.08; rstar = 0.075;
Narbitrage = 4;

output file = var6.out reset;

{Cam,arbreC} = OptionAmericaine(S0,K,sigma,tau,r,rstar,Narbitrage);
print Cam;

deltaTau = -1/365;
{Cam,arbreC} = OptionAmericaine(S0,K,sigma,tau+deltaTau,r,rstar,Narbitrage);
print Cam;

deltaS0 = 0.01;
{Cam,arbreC} = OptionAmericaine(S0+deltaS0,K,sigma,tau,r,rstar,Narbitrage);
print Cam;

{Cam,arbreC} = OptionAmericaine(S0+deltaS0,K,sigma,tau+deltaTau,r,rstar,Narbitrage);
print Cam;

output off;

      0.12078477
      0.12010642
      0.12417366
      0.12349092
```

5.2.2 Disparition de la propriété de normalité

La propriété de normalité disparait dans le cas non linéaire. Voyons un exemple avec une option américaine.

```
new;
library edf,tsm,optmum,pgraph;

S0 = 3.5; K = 3.51; sigma = 0.19; tau = 90/365; r = 0.08; rstar = 0.075;
Narbitrage = 4;

deltaTau = -7/365;  /* 7 jours */

proc muProc(t,S);
  retp( (r-rstar)*S );
endp;

proc sigmaProc(t,S);
  retp( sigma*S );
endp;

/* calcul de la valeur de l'option en t0 */

{C0,arbreC} = OptionAmericaine(S0,K,sigma,tau,r,rstar,Narbitrage);

/* Valeurs simulees de l'option en t1 */
```



Graphique 14:

```
Ns = 2000;
{t,S} = EulerMaruyama(S0,&muProc,&sigmaProc,0,abs(deltaTau),10,Ns);

S1 = S[10,.]';
C1 = zeros(Ns,1);
i = 1;
do until i > Ns;
  {C1[i],arbreC} = OptionAmericaine(S1[i],K,sigma,tau+deltaTau,r,rstar,Narbitrage);
  i = i + 1;
endo;


deltaC = C1 - C0;

{absc,dens,repart,retcode} = Kernel(deltaC);

graphset;
  _pdate = ""; _pnum = 2;
  title("Fonction de densite de la variation de la valeur de l'option");
  graphprt("-c=1 -cf=var7.eps");
  xy(absc,dens);
```

5.2.3 Calcul de la *VaR* par la méthode de Monte Carlo

Voici une illustration simple de calcul de la *VaR*. Remarquez que nous n'avons plus la symmétrie entre position courte et position longue.

```
new;
library edf;

S0 = 3.5; K = 3.51; sigma = 0.19; tau = 90/365; r = 0.08; rstar = 0.075;
Narbitrage = 4;

deltaTau = -7/365;  /* 7 jours */

proc muProc(t,S);
  retp( (r-rstar)*S );
endp;

proc sigmaProc(t,S);
  retp( sigma*S );
endp;

/* t0 */

{C0,arbreC} = OptionAmericaine(S0,K,sigma,tau,r,rstar,Narbitrage);

/* t1 */

Ns = 1000;
{t,S} = EulerMaruyama(S0,&muProc,&sigmaProc,0,abs(deltaTau),10,Ns);

S1 = S[10,.]';
C1 = zeros(Ns,1);
i = 1;
do until i > Ns;
  {C1[i],arbreC} = OptionAmericaine(S1[i],K,sigma,tau+deltaTau,r,rstar,Narbitrage);
  i = i + 1;
endo;

output file = var8.out reset;

/* Cas d'une position longue */

deltaC = C1 - C0;
alpha = 0.05;
VaR = abs(quantile(deltaC,alpha));

print "Position longue";
print "---------------";
print ftos(VaR,"Value at Risk (0.05) = %lf Francs",5,4);
print;

/* Cas d'une position courte */

deltaC = - (C1 - C0);
```

```
alpha = 0.05;
VaR = abs(quantile(deltaC,alpha));

print "Position courte";
print "---------------";
print ftos(VaR,"Value at Risk (0.05) = %lf Francs",5,4);

output off;

Position longue
---------------
Value at Risk (0.05) = 0.0551 Francs

Position courte
---------------
Value at Risk (0.05) = 0.0989 Francs
```

5.2.4 Calcul de la *VaR* par l'approximation de Taylor

Avec une expansion de Taylor à l'ordre 1, nous avons

$$\Delta C = \Delta S \frac{\partial C}{\partial S} + \Delta \tau \frac{\partial C}{\partial \tau} + \ldots$$

En général, nous négligeons l'influence des autres paramètres sur les variations du prix de l'option. Des auteurs suggèrent cependant de prendre en compte le gamma de l'option. Nous avons alors

$$\Delta C = \Delta S \frac{\partial C}{\partial S} + \Delta \tau \frac{\partial C}{\partial \tau} + \frac{1}{2} (\Delta S)^2 \frac{\partial^2 C}{\partial S^2}$$

```
new;
library edf;

S0 = 3.5; K = 3.51; sigma = 0.19; tau = 90/365; r = 0.08; rstar = 0.075;
Narbitrage = 4;

/* Calcul des coefficients caracteristiques */

proc call_S(S);
  local C,arbreC;
  {C,arbreC} = OptionAmericaine(S,K,sigma,tau,r,rstar,Narbitrage);
  retp(C);
endp;

proc call_t(t);
  local C,arbreC;
  {C,arbreC} = OptionAmericaine(S0,K,sigma,t,r,rstar,Narbitrage);
  retp(C);
endp;

DELTA = gradp(&call_S,S0);
GAMMA_ = hessp(&call_S,S0);
THETA = gradp(&call_t,tau);
```

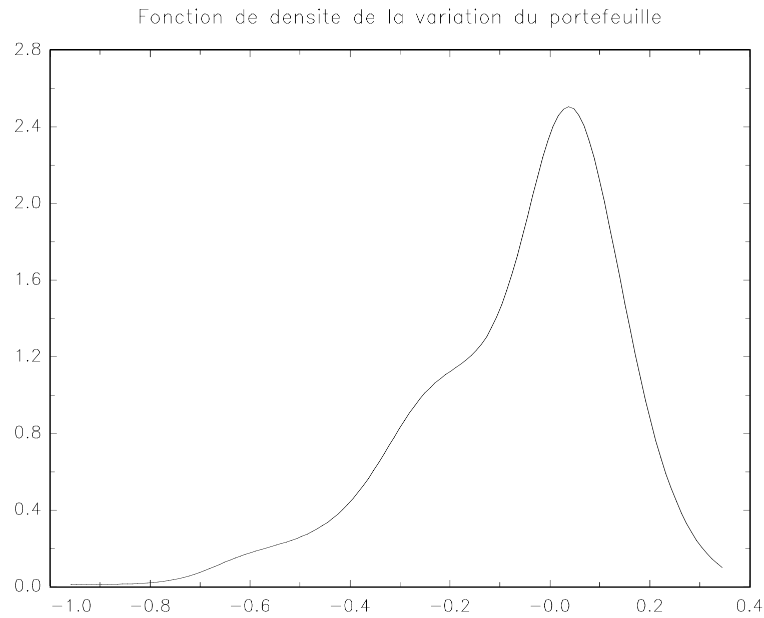
```
/* Simulation de S(t+deltaTau) */

deltaTau = -7/365;
alpha = 0.05;

proc muProc(t,S);
  retp( (r-rstar)*S );
endp;

proc sigmaProc(t,S);
  retp( sigma*S );
endp;

Ns = 1000;
{t,S} = EulerMaruyama(S0,&muProc,&sigmaProc,0,abs(deltaTau),10,Ns);

S1 = S[10,.]';

deltaS = S1 - S0;

output file = var9.out reset;

/* Approximation de Taylor a l'ordre 1 */

deltaC = deltaS * DELTA + deltaTau * THETA;
VaR = abs(quantile(deltaC,alpha));
print "Taylor a l'ordre 1";
print ftos(VaR,"Value at Risk (0.05) = %lf Francs",5,4);

/* Approximation de Taylor a l'ordre 2 */

deltaC = deltaS * DELTA + deltaTau * THETA + 0.5 * (deltaS^2) * GAMMA_;
VaR = abs(quantile(deltaC,alpha));
print "Taylor a l'ordre 2";
print ftos(VaR,"Value at Risk (0.05) = %lf Francs",5,4);

output off;

Taylor a l'ordre 1
Value at Risk (0.05) = 0.0549 Francs
Taylor a l'ordre 2
Value at Risk (0.05) = 0.0549 Francs
```

5.2.5 *VaR* et portefeuille d'options

On peut généraliser les exemples précédents lorsque plusieurs options composent le portefeuille. Dans ce cas, la fonction de densité de variation du portefeuille peut prendre des formes assez singulières.

```
new;
library edf,tsm,optmum,pgraph;

S0 = 3.5; K = 3.51; sigma = 0.19; r = 0.08; rstar = 0.075;
```

```
theta_Cam = -5;
theta_PutLookBack = 6;

/* Option americaine d'achat */
tau1 = 90/365;

/* Option Look-back de vente */
tau2 = 160/365;

Narbitrage = 4;

deltaTau = -7/365;

proc muProc(t,S);
  retp( (r-rstar)*S );
endp;

proc sigmaProc(t,S);
  retp( sigma*S );
endp;

/* t0 */

{Cam,arbreC} = OptionAmericaine(S0,K,sigma,tau1,r,rstar,Narbitrage);
{C,PutLookBack} = LookBackOption(S0,sigma,tau2,r,rstar,Narbitrage);

ValeurPortefeuille_t0 = theta_Cam*Cam + theta_PutLookBack*PutLookBack;

/* t1 */

Ns = 1000;
{t,S} = EulerMaruyama(S0,&muProc,&sigmaProc,0,abs(deltaTau),10,Ns);

ST = S[10,.]';
Cam = zeros(Ns,1);
PutLookBack = zeros(Ns,1);

i = 1;
do until i > Ns;
  {Cam[i],arbreC} = OptionAmericaine(ST[i],K,sigma,tau1+deltaTau,r,rstar,Narbitrage);
  {C,PutLookBack[i]} = LookBackOption(ST[i],sigma,tau2+deltaTau,r,rstar,Narbitrage);
  i = i + 1;
endo;

ValeurPortefeuille_t1 = theta_Cam*Cam + theta_PutLookBack*PutLookBack;

deltaP = ValeurPortefeuille_t1 - ValeurPortefeuille_t0;

{absc,dens,repart,retcode} = Kernel(deltaP);

graphset;
  _pdate = ""; _pnum = 2;
```



Graphique 15:

```
title("Fonction de densite de la variation du portefeuille");
graphprt("-c=1 -cf=var10.eps");
xy(absc,dens);
```

5.3 Les techniques d'estimation

5.3.1 Estimateurs classiques de la volatilité

5.3.2 Volatilité stochastique

5.3.3 La théorie des valeurs extrêmes

5.4 La gestion des données