Formation GAUSS pour le CCF

Niveau II

Thierry Roncalli

Université Montesquieu-Bordeaux IV, Avenue Léon Duguit, 33608 Pessac Cedex

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

Table des matières

1 Introduction

- Un langage de programmation n'est pas seulement un catalogue de commandes.
- Un programme doit pouvoir être utilisé par des personnes qui n'ont pas participé à son élaboration.
- La meilleure façon d'apprendre à programmer est de copier les autres.

2 La notion de procédure

2.1 Les fonctions en ligne

Considérons la fonction

$$f(x) = \sin(x)\cos(x)e^x \tag{1}$$

Dans l'exemple suivant, nous utilisons la fonction plusieurs fois.

```
new;

y2 = sin(2)*cos(2)*exp(2);

/*
...
*/

y3 = sin(3)*cos(3)*exp(3);

/*
...
*/

y4 = sin(4)*cos(4)*exp(4);
```

Cette fonction n'est pas difficile à écrire. Si nous voulons éviter les redondances d'écriture, nous pouvons utiliser les fonctions en ligne (*inline function*) avec la commande `fn`. Nous pouvons placer cette fonction n'importe où dans le programme, par exemple au début ou à la fin. Cela n'a pas d'importance, car le lien est effectué lors de la compilation.

```
new;


y2 = f(2);

/*
...
*/

y3 = f(3);

/*
...
*/
```

```
y4 = f(4);


/*
** Fonction f(x) = sin(x)*cos(x)*exp(x)
*/

fn f(x) = sin(x)*cos(x)*exp(x);
```

La syntaxe de la commande `fn` est

```
fn f(x,y,z,...)  = ...
```

Elle accepte donc plusieurs arguments (ceux-ci peuvent être bien sûr des matrices complexes). Considérons la fonction d'actualisation $\frac{1}{(1+r)^\tau}$. Dans l'exemple suivant, nous utilisons une fonction en ligne avec deux arguments r et τ pour définir le facteur d'actualisation.

```
new;


fa = P(0.05,1);

/*
...
*/

fa = P(0.05,2);

/*
...
*/

fa = P(0.05,3);

output file = ccf3.out reset;

taux_interet = 0.05;
maturite = 3;

print "P(taux_interet,maturite) = " P(taux_interet,maturite);

tau = 0.05;
r = 3;

print "P(tau,r) = " P(tau,r);

output off;

/*
** Facteur d'actualisation
*/

fn P(r,tau) = 1/((1+r)^tau);
```

Cet exemple montre clairement que **les paramètres de la fonction sont des variables locales**. Leurs noms n'ont donc pas d'importance, c'est l'ordre des variables qui compte. Le fichier *ccf3.out* se présente ainsi :

```
P(taux_interet,maturite) =       0.86383760
P(tau,r) =       0.86383760
```

Ces fonctions en ligne sont cependant difficiles à utiliser pour des expressions relativement complexes. Considérons par exemple la formule de Black et Scholes [1973]. Soient S_0 le prix du sous-jacent, K le prix d'exercice, σ la volatilité de l'actif, τ la maturité de l'option et r le taux d'intérêt. Nous avons

$$\left\{ \begin{array}{rcl} C & = & S_0\Phi\left(d_1\right) - Ke^{-r\tau}\Phi\left(d_2\right) \\ d_1 & = & \frac{\ln\left(\frac{S_0}{K}\right)+r\tau}{\sigma\sqrt{\tau}} + \frac{1}{2}\sigma\sqrt{\tau} \\ d_2 & = & d_1 - \sigma\sqrt{\tau} \end{array} \right. \tag{2}$$

```
/*
** La formule de Black et Scholes [1973]
*/

new;

S0 = 100;
K = 98;
sigma = 0.15;
tau = 95/365;
r = 0.08;

w = sigma*sqrt(tau);

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

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

output file = ccf4.out reset;

print ftos(C,"Prime de l'option d'achat : %lf Francs",6,3);

output off;

Prime de l'option d'achat :  5.420 Francs
```

Exercice : Ecrire la fonction en ligne correspondante `fn BlackScholes(S0,K,sigma,tau,r) = ...;`.

2.2 Les procédures

Il est plus facile d'utiliser une procédure pour définir la fonction de Black et Scholes. La syntaxe basique d'une procédure est

```
proc procedure(x,y,z,...);
  local retour,...;
```

```
  ...

  retour = ...;

  retp(retour);
endp;
```

Nous appelons cette procédure de la façon suivante :

```
retour = procedure(x,y,z,...);
```

Les variable `x`, `y` et `z` sont des variables **locales**. Nous pouvons définir d'autres variables locales avec l'instruction `local`. `retour` étant une variable locale, la syntaxe suivante est donc correcte :

```
z = procedure(retour,x,y,...);
```

Les noms des variables ne sont pas importants, car ces variables sont utilisées localement dans la procédure et n'existent pas à l'extérieur de celle-ci.

```
new;

S0 = 100;
K = 98;
sigma = 0.15;
tau = 95/365;
r = 0.08;

output file = ccf5.out reset;

C = EuropeanBScall(S0,K,sigma,tau,r);
print ftos(C,"Prime de l'option d'achat : %lf Francs",6,3);

C = EuropeanBScall(S0,100,sigma,tau,r);
print ftos(C,"Prime de l'option d'achat : %lf Francs",6,3);

C = EuropeanBScall(K,S0,sigma,tau,r);
print ftos(C,"Prime de l'option d'achat : %lf Francs",6,3);

output off;


/*
**  Calcul de la prime d'une option d'achat
**  par la formule de Black et Scholes [1973]
*/

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

  w = sigma.*sqrt(tau);

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

```
  C = S0.*cdfn(d1)-K.*exp(-r.*tau).*cdfn(d2);

  retp(C);
endp;
```

```
Prime de l'option d'achat :  5.420 Francs
Prime de l'option d'achat :  4.162 Francs
Prime de l'option d'achat :  3.021 Francs
```

Comment utiliser cette procédure chaque fois que nous en avons besoin ? La méthode copier/coller n'est pas la plus efficace. Une solution est d'utiliser les directives de compilation. `#include` permet d'insérer un fichier. Ainsi, nous pouvons donc créer un fichier *bs.inc* ne contenant que la procédure de Black et Scholes et l'appeler chaque fois que nous en avons besoin.

- fichier *bs.inc* :

```
/*
**  Calcul de la prime d'une option d'achat
**  par la formule de Black et Scholes [1973]
*/

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

  w = sigma.*sqrt(tau);

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

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

  retp(C);
endp;
```

- fichier *ccf6.prg* :

```
new;

#include bs.inc;

S0 = 100;
K = 98;
sigma = 0.15;
tau = 95/365;
r = 0.08;

C = EuropeanBScall(S0,K,sigma,tau,r);
print ftos(C,"Prime de l'option d'achat : %lf Francs",6,3);

C = EuropeanBScall(S0,100,sigma,tau,r);
print ftos(C,"Prime de l'option d'achat : %lf Francs",6,3);

C = EuropeanBScall(K,S0,sigma,tau,r);
print ftos(C,"Prime de l'option d'achat : %lf Francs",6,3);
```

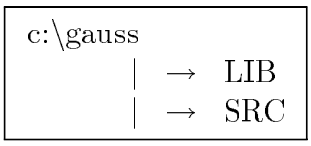
Cette méthode n'est pas efficace car elle suppose que nous sachions localiser l'ensemble des procédures. Une procédure peut faire appel à une autre procédure. Il faut donc insérer aussi le code de l'autre procédure. Pour utiliser par exemple **TSM**, nous devons spécifier :

```
new;

#include ARMA1.SRC
#include ARMA2.SRC
#include DENOISE.SRC
#include FDML.SRC
#include FILTER.SRC
#include GMM.SRC
#include KALMAN$.SRC
#include KALMAN2.SRC
#include MATRIX.SRC
#include MODEL.SRC
#include OPTMUM2.SRC
#include QMF.SRC
#include RANDOM.SRC
#include SPECTRAL.SRC
#include SSM.SRC
#include TDML.SRC
#include TOOLS.SRC
#include TSM.DEC
#include TSM.EXT
#include TSM.SRC
#include TSMPLOT.SRC
#include TSREG.SRC
#include VARX.SRC
#include WAVELET.SRC
#include WPACKET.SRC
```

3 La notion de bibliothèque

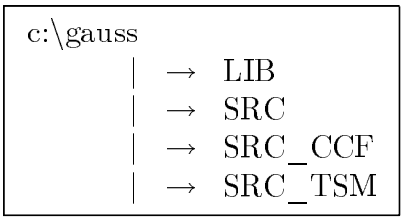
3.1 Création de la bibliothèque CCF

Une solution efficace est d'utiliser une bibliothèque. Celle-ci est constitué 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 **CCF** sera déclarée dans le répertoire *lib* avec le fichier *ccf.lcg* et les codes sources se trouveront dans le répertoire *src_ ccf*. Pour construire une bibliothèque, nous utilisons la commande `lib`, par exemple

```
lib ccf
```

Il y a alors création d'un fichier vide *ccf.lcg* dans le répertoire *lib*.

3.2 Modification de la bibliothèque CCF

Ajoutons à cette bibliothèque la procédure de Black et Scholes. Nous créons dans un premier temps le fichier *bs.src* sous le répertoire *src_ ccf* :

```
/*
**  Calcul de la prime d'une option d'achat
**  par la formule de Black et Scholes [1973]
*/

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

  w = sigma.*sqrt(tau);

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

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

  retp(C);
endp;
```

Nous ajoutons ensuite la procédure `EuropeanBScall` à la librairie **ccf** avec l'instruction

```
lib ccf bs.src
```

Si le répertoire courant n'est pas *src_ ccf*, nous spécifions le chemin :

```
lib ccf c:\gauss\src_ccf\bs.src
```

Le fichier *ccf.lcg* devient :

```
c:\gauss\src_ccf\bs.src
    europeanbscall                  : proc
```

Nous pouvons maintenant utiliser la bibliothèque **ccf**. Pour cela, nous utilisons l'instruction `library`.

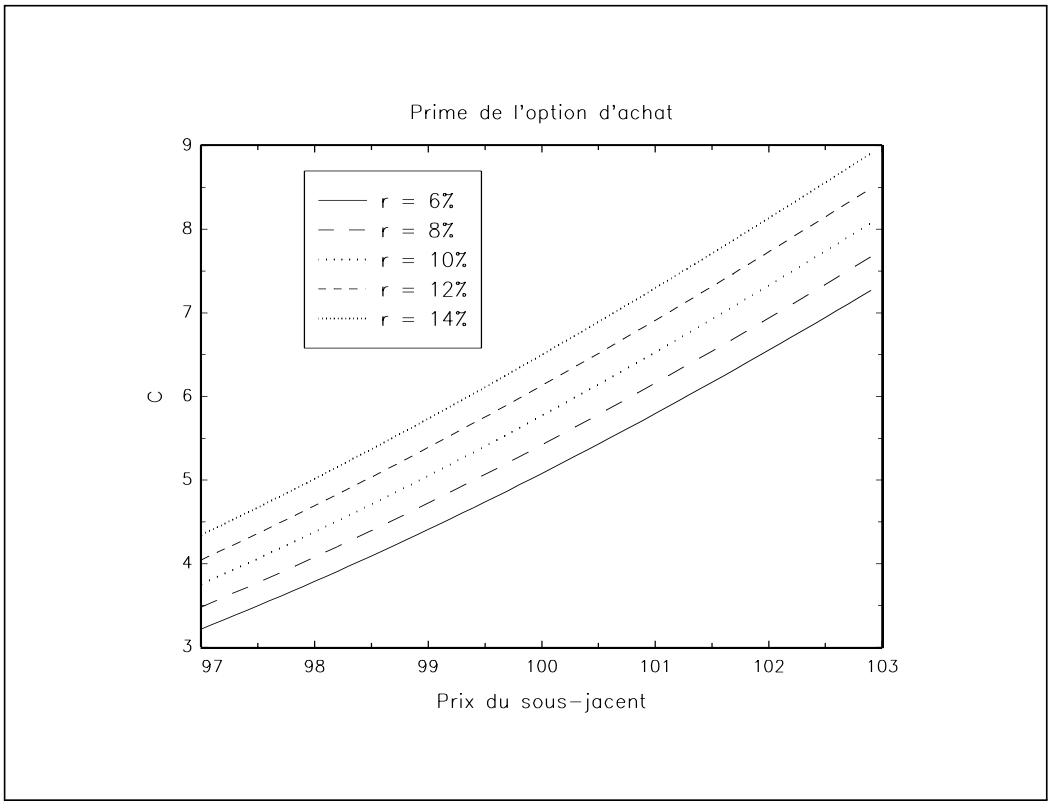
```
new;
library ccf;

S0 = 100;
K = 98;
sigma = 0.15;
tau = 95/365;
r = 0.08;

output file = ccf8.out reset;

C = EuropeanBScall(S0,K,sigma,tau,r);
print ftos(C,"Prime de l'option d'achat : %lf Francs",6,3);

C = EuropeanBScall(S0,100,sigma,tau,r);
print ftos(C,"Prime de l'option d'achat : %lf Francs",6,3);

C = EuropeanBScall(K,S0,sigma,tau,r);
print ftos(C,"Prime de l'option d'achat : %lf Francs",6,3);

output off;
```

```
Prime de l'option d'achat :  5.420 Francs
Prime de l'option d'achat :  4.162 Francs
Prime de l'option d'achat :  3.021 Francs
```

Nous pouvons utiliser plusieurs bibliothèques, par exemple **ccf** et **pgraph**.

```
new;
library ccf,pgraph;

SousJacent = seqa(97,0.1,60);
PrixExercice = 98;
Volatilite = 0.15;
Maturite = 95/365;
TauxInteret = seqa(0.06,0.02,5)';

C = EuropeanBScall(SousJacent,PrixExercice,Volatilite,Maturite,TauxInteret);


graphset;
  xy(SousJacent,C)

graphset;
  _pdate = ""; _pnum = 2; _plwidth = 0|0|5|0|5;
  title("Prime de l'option d'achat");
  xlabel("Prix du sous-jacent");
  ylabel("C");
  _plegstr = "r = 6%\000r = 8%\000r = 10%\000r = 12%\000r = 14%";
  _plegctl = {2 6 2 4};
  graphprt("-c=1 -cf=ccf9.eps -w=5");
  xy(SousJacent,C)
```

Le fichier suivant *random.src* contient plusieurs procédures pour simuler des variables aléatoires. Il existe d'autres fichiers du même nom, par exemple celui de **TSM** ou celui de **GAUSS**. Si nous



Graphique 1:

exécutons la commande `lib ccf random.src`, nous devons nous assurer que le répertoire courant est bien *src_ccf*.

```
/*
**  Simulation d'une v.a. de Bernoulli
*/

proc RNDbernoulli(p,r,c);
  local u;

  u = rndu(r,c);
  u = u .<= p;

  retp(u);
endp;


/*
**  Simulation d'une v.a. Binomiale
*/

proc rndb(N,p,r,c);
  local bern,u;

  bern = RNDbernoulli(p,N,r*c);

  u = sumc(bern);
```

```
  u = reshape(u,r,c);

  retp(u);
endp;


/*
**  Simulation d'une v.a. normale multidimensionnelle
*/

proc rndmn(mu,SIGMA,Ns);
  local u,P,dim;

  dim = rows(SIGMA);
  P = chol(SIGMA)';
  u = mu + P*rndn(dim,Ns);

  retp(u');
endp;
```

Le fichier *ccf.lcg* devient :

```
c:\gauss\src_ccf\bs.src
    europeanbscall                  : proc

c:\gauss\src_ccf\random.src
    rndbernoulli                    : proc
    rndb                            : proc
    rndmn                           : proc
```

Voyons un nouvel exemple d'utilisation de la bibliothèque **CCF**.

```
new;
library ccf;

output file = ccf10.out reset;

bern = rndbernoulli(0.25,20000,1);

print "===================================";
print meanc(bern)~stdc(bern);
print "-----------------------------------";
print 0.25~sqrt(0.25*0.75);
print;

b = rndb(50,0.25,20000,1);

print "===================================";
print meanc(b)~stdc(b);
print "-----------------------------------";
print 50*0.25~sqrt(50* 0.25*0.75);
```

```
print;

let sigma = 10 12 2;
let mu[3,3] =     2.2   0.5   0.0
                  0.5   1.2  -0.6
                  0.0  -0.6   1.8;

n = rndmn(sigma,MU,20000);

print "====================================";
print meanc(n);
print sigma;
print "------------------------------------";
print vcx(n);
print mu;
print "------------------------------------";

output off;

====================================
      0.24985000       0.43293689
------------------------------------
      0.25000000       0.43301270

====================================
       12.506250        3.0639240
------------------------------------
       12.500000        3.0618622

====================================

       9.9991579
       11.994682
       2.0007549

       10.000000
       12.000000
       2.0000000
------------------------------------

       2.1968676       0.50114657     0.0039588933
      0.50114657        1.2040375      -0.59589715
    0.0039588933      -0.59589715        1.7912970

       2.2000000       0.50000000        0.0000000
      0.50000000        1.2000000      -0.60000000
       0.0000000      -0.60000000        1.8000000
------------------------------------
```

Exercice : Ajoutez à la librairie CCF des procédures de simulation de v.a. chi-deux, student et Fisher.

3.3 Lisibilité d'une bibliothèque

Une bonne bibliothèque doit comporter des procédures dont les utilisateurs ont souvent besoin. Il faut penser que cette bibliothèque sera partagée par plusieurs utilisateurs et qu'elle sera construite par différentes personnes. Un utilisateur doit pouvoir rapidement utiliser une procédure qu'il n'a pas écrite. Il est donc impératif de créer un manuel d'utilisation des procédures. Ce manuel doit comporter trois parties : une vue d'ensemble avec des détails sur les algorithmes employés, l'ensemble des syntaxes des procédures et des exemples d'utilisation. De plus, les fichiers sources doivent supporter l'aide en ligne (commande help on:**). Voici un exemple de fichier qui permet l'aide en ligne.**

```
/*
** random2.src - CCF.
** (C) Copyright 1997 by Credit Commercial de France
** All Rights Reserved.
**
**  Format                          Purpose                                 Line
** ============================================================================
**  u = RNDbernoulli(p,r,c);        Simulation de nombres aleatoires issus
**                                  d'une variable aleatoire de bernoulli    21
**
**  u = RNDb(N,p,r,c);              Simulation de nombres aleatoires issus
**                                  d'une variable aleatoire binomiale       53
**
**  u = RNDmn(mu,SIGMA,Ns);         Simulation de vecteurs aleatoires issus
**                                  d'une loi normale multi-dimensionnelle   87
**
*/


/*
** rndBernoulli
**
** Purpose: Simulation de nombres aleatoires issus d'une
**          variable aleatoire de Bernoulli
**
** Format:  u = RNDbernoulli(p,r,c);
**
** Input:   p - scalaire/vecteur r*1/vecteur 1*c/matrice r*c, parametre
**              de la loi de Bernoulli
**          r - scalaire, nombre de lignes
**          c - scalaire, nombre de colonnes
**
** Output:  u - matrice r*c, matrice des nombres pseudo-aleatoires
**
** Remarks: Les proprietes des nombres pseudo-aleatoires de u sont -
**
**            E(u) = p, Var(u) = p*(1-p)
**            0 < p < 1,  u = 0,1
**
*/
```

```
proc RNDbernoulli(p,r,c);
  local u;

  u = rndu(r,c);
  u = u .<= p;

  retp(u);
endp;


/*
** rndb
**
** Purpose: Simulation de nombres aleatoires issus d'une
**          variable aleatoire binomiale
**
** Format:  u = RNDb(N,p,r,c);
**
** Input:   N - scalaire, parametre N de la loi binomiale
**          p - scalaire, parametre p de la loi binomiale
**          r - scalaire, nombre de lignes
**          c - scalaire, nombre de colonnes
**
** Output:  u - matrice r*c, matrice des nombres pseudo-aleatoires
**
** Remarks: Les proprietes des nombres pseudo-aleatoires de u sont -
**
**            E(u) = N*p, Var(u) = N*p*(1-p)
**            0 < p < 1,  N entier, u = 0,1,..,N
**
*/

proc rndb(N,p,r,c);
  local bern,u;

  bern = RNDbernoulli(p,N,r*c);

  u = sumc(bern);
  u = reshape(u,r,c);

  retp(u);
endp;


/*
** rndmn
**
** Purpose: Simulation de vecteurs aleatoires issus d'une
**          loi normale multi-dimensionnelle
**
** Format:  u = rndmn(mu,SIGMA,Ns);
**
** Input:   mu -
```

```
**       SIGMA -
**          Ns -
**
** Output:  u -
**
** Remarks:
**
*/

proc rndmn(mu,SIGMA,Ns);
  local u,P,dim;

  dim = rows(SIGMA);
  P = chol(SIGMA)';
  u = mu + P*rndn(dim,Ns);

  retp(u');
endp;
```

Nous pouvons utiliser plusieurs formats de matrice pour le paramètre p. Mais que se passe-t-il si l'utilisateur emploie un mauvais format ?

```
new;
library ccf;

output file = ccf11.out reset;

u = rndbernoulli(0.25,100,3);
u = rndbernoulli(0.25~0.5~0.6,100,3);
p = rndu(100,1);
u = rndbernoulli(p,100,3);
p = rndu(100,3);
u = rndbernoulli(p,100,3);

p = rndu(100,4);
u = rndbernoulli(p,100,3);

output off;
```

```
C:\GAUSS\SRC_CCF\RANDOM.SRC(9) : error G0036 : Matrices are not conformable
Currently active call: RNDBERNOULLI [9]
```

Une bonne bibliothèque doit aussi comporter une gestion des erreurs.

4 La notion de variable locale

L'existence d'une variable locale est temporaire. Elle n'est définie qu'à l'intérieur de la procédure. Une fois la procédure exécutée, elle disparait. Dans l'exemple suivant, lorsque la ligne de commande `u = f(x)` est exécutée, la procédure utilise une copie de la variable `x`. C'est pourquoi `x` vaut 4 et non 16. GAUSS fonctionne donc comme le langage C ou C++ et non comme Fortran.

```
new;
```

```
output file = ccf12.out reset;

x = f(2);

print;

u = f(x);

print "4. x = "  x;

/*
print "5. y = "  y;
*/

output off;


proc f(x);
  local y;

  print "1. x = " x;

  x = x^2;

  print "2. x = " x;

  y = x;

  print "3. y = " x;

  retp(y);
endp;

1. x =        2.0000000
2. x =        4.0000000
3. y =        4.0000000

1. x =        4.0000000
2. x =        16.000000
3. y =        16.000000
4. x =        4.0000000
```

5 La notion de variable globale ou externe

5.1 Une première version des moindres carrés ordinaires

Considérons une première version des moindres carrés ordinaires. Utilisons pour cela une procédure avec plusieurs arguments de retour. La syntaxe est :

```
proc (n) = procedure(x,y,z,...);
  local ret1,...;
```

```
  ...

  ret1 = ...;

  retp(ret1,ret2,...,retn);
endp;
```

N'oubliez pas de déclarer le nombre de retour avec `proc (n) =` . Pour utiliser cette procédure, nous utilisons la syntaxe suivante (avec des accolades) :

```
{ret1,ret2,...,retn} = procedure(x,y,z,...);
```

```
/*
** Une mauvaise version des MCO
*/

proc (9) = mco_(y,x);
  local Nobs,k,xxinv,beta,u;
  local SCR,SCT,SCE,R2,ddl,sigma,Mcov;
  local stderr,tstudent,pvalue,du,DW,Mcorr;

  Nobs = rows(y);
  k = cols(x);

  xxinv = invpd(x'x);
  beta = xxinv*x'y;

  u = y - x*beta;
  SCR = u'u;
  SCT = y'y;
  SCE = SCT - SCR;
  R2 = SCE/SCT;

  ddl = Nobs - k;
  sigma = sqrt(SCR/ddl);

  Mcov = (sigma^2)*xxinv;
  stderr = sqrt(diag(Mcov));

  tstudent = beta ./ stderr;
  pvalue = 2*cdftc(abs(tstudent),ddl);

  du = u - lag1(u);
  du = trimr(du,1,0);

  DW = sumc(du^2) / sumc(u^2);

  Mcorr = Mcov ./ stderr ./ stderr';

  print ftos(Nobs, "Nombre d'observations : %lf",5,0);
  print ftos(k,    "Nombre de regresseurs : %lf",5,0);
  print;
  print ftos(ddl,  "Degres de liberte :     %lf",5,0);
```

```
  print;
  print ftos(sigma,"Ecart-type estime :     %lf",5,2);
  print ftos(R2,   "R2 :                    %lf",5,3);
  print ftos(DW,   "Durbin-Watson :         %lf",5,3);
  print;
  print "      coefficient      ecart-type       T-student"\
        "      Prob. marginale";
  print "-----------------------------------------------"\
        "---------------------";
  print beta~stderr~tstudent~pvalue;
  print;
  print "Matrice de covariance des estimateurs :";
  print Mcov;
  print;
  print "Matrice de correlation des estimateurs :";
  print Mcorr;

  retp(beta,stderr,tstudent,pvalue,Mcov,Mcorr,u,sigma,DW);
endp;
```

Après avoir ajouté cette procédure à la librairie **CCF** (avec `lib ccf mco_.src`), le fichier *ccf.lcg* devient :

```
c:\gauss\src_ccf\bs.src
    europeanbscall                  : proc

c:\gauss\src_ccf\random.src
    rndbernoulli                    : proc
    rndb                            : proc
    rndmn                           : proc

c:\gauss\src_ccf\mco_.src
    mco_                            : proc
```

Voyons un exemple d'utilisation :

```
new;
library ccf;

x = rndu(100,4)*100;
coeff = 1|2|3|4;
y = x*coeff + rndn(100,1)*2;

output file = ccf13.out reset;

{b,std,t,p,Mv,Mr,u,sig,durbin} = mco_(y,x);

Mcov = 4*invpd(x'x);
stderr = sqrt(diag(Mcov));
Mcorr = Mcov ./ stderr ./ stderr';
print;
print "Matrice de covariance theorique : ";
```

```
print Mcorr;

output off;

Nombre d'observations :   100
Nombre de regresseurs :     4

Degres de liberte :        96

Ecart-type estime :      2.18
R2 :                    1.000
Durbin-Watson :         2.192

      coefficient      ecart-type       T-student      Prob. marginale
-----------------------------------------------------------------------

       1.0142793     0.0063769085        159.05502   4.2854292e-118
       2.0040525     0.0068463305        292.71922   1.8041351e-143
       2.9880017     0.0074332208        401.97941   1.1037546e-156
       3.9872034     0.0077542224        514.19771   6.0632967e-167

Matrice de covariance des estimateurs :

   4.0664962e-05   -6.4429853e-06   -2.0899541e-05   -1.0986244e-05
  -6.4429853e-06    4.6872242e-05   -1.7432752e-05   -2.3785541e-05
  -2.0899541e-05   -1.7432752e-05    5.5252771e-05   -1.2716882e-05
  -1.0986244e-05   -2.3785541e-05   -1.2716882e-05    6.0127965e-05

Matrice de correlation des estimateurs :

       1.0000000      -0.14757714      -0.44090958      -0.22217787
     -0.14757714        1.0000000      -0.34255559      -0.44804013
     -0.44090958      -0.34255559        1.0000000      -0.22063040
     -0.22217787      -0.44804013      -0.22063040        1.0000000

Matrice de covariance theorique :

       1.0000000      -0.14757714      -0.44090958      -0.22217787
     -0.14757714        1.0000000      -0.34255559      -0.44804013
     -0.44090958      -0.34255559        1.0000000      -0.22063040
     -0.22217787      -0.44804013      -0.22063040        1.0000000
```

Si vous n'avez pas besoin des arguments de retour, vous pouvez employez la commande `call`.

```
new;
library ccf;

x = rndu(100,4)*100;
coeff = 1|2|3|4;
y = x*coeff + rndn(100,1)*2;

call mco_(y,x);
```

La programmation de la procédure `mco_` pose plusieurs problèmes. Comment éviter l'affichage des résultats ? Une solution non efficace est de définir un troisième argument d'entrée. Comme il y a beaucoup d'arguments de sortie, la syntaxe de la procédure est difficile à retenir. De plus, certaines informations sont parfois inutiles.

5.2 Une seconde version

Une solution est d'utiliser des variables globales externes. La valeur de ces variables n'est pas définie au moment de la compilation, mais au moment de l'exécution. Considérons par exemple une variable booléenne `_print` qui permet d'afficher ou de ne pas afficher les résultats. Cette variable permet alors d'influencer le programme. Considérons aussi une variable `_mco_DW` qui définit la statistique de Durbin et Watson. Cette variable permet de récupérer des résultats. Pour définir des variables, nous utilisons deux fichiers, *ccf.dec* et *ccf.ext*.

```
/*
** ccf.dec - CCF.
** (C) Copyright 1997 by Credit Commercial de France
** All Rights Reserved.
**
** Global variable definitions for the CCF library
*/

declare matrix _print = 1;
declare matrix _mco_DW;
```

```
/*
** ccf.ext - CCF.
** (C) Copyright 1997 by Credit Commercial de France
** All Rights Reserved.
**
** External variable declarations for the CCF library
*/

external matrix _print;
external matrix _mco_DW;
```

Ces variables font partie de la bibliothèque CCF. Nous les ajoutons avec la commande `lib ccf ccf.dec`.

```
c:\gauss\src_ccf\bs.src
    europeanbscall                  : proc

c:\gauss\src_ccf\random.src
    rndbernoulli                    : proc
    rndb                            : proc
    rndmn                           : proc

c:\gauss\src_ccf\mco_.src
    mco_                            : proc
```

```
c:\gauss\src_ccf\ccf.dec
    _print                          : matrix
    _mco_dw                         : matrix
```

La procédure devient :

```
/*
** mco.src - CCF.
** (C) Copyright 1997 by Credit Commercial de France
** All Rights Reserved.
**
**  Format                          Purpose                                 Line
** ============================================================================
**  {beta,stderr,Mcov,u} = mco(y,x);
**                                  Moindres carres ordinaires               15
**
*/


/*
** mco
**
** Purpose: Moindres carres ordinaires
**
** Format:  {beta,stderr,Mcov,u} = mco(y,x);
**
** Input:   y - vecteur Nobs*1, variable endogene
**          x - matrice Nobs*K, variables exogenes
**
** Output:  beta - vecteur K*1, ceofficients estimes
**        stderr - vecteur K*1, ecart-types estimes
**          Mcov - matrice K*K, matrice de covariance
**             u - vecteur Nobs*1, residus
**
** Globals:  _print - scalaire 1 (defaut), affiche les resultats
**                    scalaire 0, n'affiche pas les resultats
**
**          _mco_DW - scalaire, statistique de Durbin-Watson
**
*/


proc (4) = mco(y,x);
  local Nobs,k,xxinv,beta,u;
  local SCR,SCT,SCE,R2,ddl,sigma,Mcov;
  local stderr,tstudent,pvalue,du,DW,Mcorr;

  Nobs = rows(y);
  k = cols(x);

  xxinv = invpd(x'x);
  beta = xxinv*x'y;

  u = y - x*beta;
```

```
  SCR = u'u;
  SCT = y'y;
  SCE = SCT - SCR;
  R2 = SCE/SCT;

  ddl = Nobs - k;
  sigma = sqrt(SCR/ddl);

  Mcov = (sigma^2)*xxinv;
  stderr = sqrt(diag(Mcov));

  tstudent = beta ./ stderr;
  pvalue = 2*cdftc(abs(tstudent),ddl);

  du = u - lag1(u);
  du = trimr(du,1,0);

  DW = sumc(du^2) / sumc(u^2);

  Mcorr = Mcov ./ stderr ./ stderr';

  if _print == 1;

    print ftos(Nobs, "Nombre d'observations : %lf",5,0);
    print ftos(k,    "Nombre de regresseurs : %lf",5,0);
    print;
    print ftos(ddl,  "Degres de liberte :     %lf",5,0);
    print;
    print ftos(sigma,"Ecart-type estime :     %lf",5,2);
    print ftos(R2,   "R2 :                    %lf",5,3);
    print ftos(DW,   "Durbin-Watson :         %lf",5,3);
    print;
    print "      coefficient      ecart-type       T-student"\
          "      Prob. marginale";
    print "-----------------------------------------------"\
          "----------------------";
    print beta~stderr~tstudent~pvalue;
    print;
    print "Matrice de covariance des estimateurs :";
    print Mcov;
    print;
    print "Matrice de correlation des estimateurs :";
    print Mcorr;
  endif;

  _mco_DW = DW;

  retp(beta,stderr,Mcov,u);
endp;
```

Le fichier *ccf.lcg* se présente alors de la façon suivante :

```
c:\gauss\src_ccf\bs.src
    europeanbscall                  : proc
```

```
c:\gauss\src_ccf\random.src
    rndbernoulli                    : proc
    rndb                            : proc
    rndmn                           : proc

c:\gauss\src_ccf\mco_.src
    mco_                            : proc

c:\gauss\src_ccf\ccf.dec
    _print                          : matrix
    _mco_dw                         : matrix

c:\gauss\src_ccf\mco.src
    mco                             : proc
```

Voyons un exemple d'utilisation :

```
new;
library ccf;

x = rndu(100,4)*100;
coeff = 1|2|3|4;
y = x*coeff + rndn(100,1)*2;

call mco(y,x);

output file = ccf15.out reset;

_print = 0; /* N'affiche pas les statistiques */

call mco(y,x);

print "Durbin-Watson" _mco_DW;

{theta,stderr,Mcov,residu} = mco(y,x);

DW = _mco_DW;
print "theta" theta;
print "DW" DW;

output off;
```

```
Durbin-Watson       1.8985782
theta
       1.0017263
       2.0018081
       3.0096714
       3.9884942
DW       1.8985782
```

6 La notion de pointeur

6.1 Un exemple

Nous définissons un pointeur d'une procédure `f` en employant l'opérande `&`. `&f` est un pointeur de la procédure `f`. Nous disons aussi que `&f` pointe sur la procédure `f`. **`&f` n'est pas une procédure, mais un scalaire entier.** La notion de pointeur en GAUSS est proche de celle en C ou C++. `&f` est un scalaire, qui permet de repérer l'emplacement de la procédure `f`. `&f` est donc lié à l'emplacement mémoire de la procédure.

Voyons un exemple. Le pointeur de la procédure `h` vaut donc 132.

```
new;

output file = ccf16.out reset;

pointeur1 = &f;
pointeur2 = &g;
pointeur3 = &h;

print pointeur1;  /* Un pointeur est un scalaire */
print pointeur2;
print pointeur3;
print pointeur1+pointeur2;

output off;

proc g(x);
  local y;
  y = x^2 .* sin(x);
  retp(y);
endp;

proc h(x);
  local y;
  y = x^2;
  retp(y);
endp;


proc f(x);
  local y;
  y = x^2 .* cos(x);
  retp(y);
endp;
```

```
       44.000000
       88.000000
       132.00000
       132.00000
```

Les pointeurs étant des scalaires, nous pouvons donc définir des vecteurs de pointeurs.

```
new;

output file = ccf17.out reset;

pointeur1 = &f;
pointeur2 = &g;
pointeur3 = &h;

pointeurs = &f|&g|&h;

pointeur123 = pointeur1|pointeur2|pointeur3;

print pointeur1;
print pointeur2;
print pointeur3;

print pointeurs;

print pointeur123;

output off;

proc g(x);
  local y;
  y = x^2 .* sin(x);
  retp(y);
endp;


proc h(x);
  local y;
  y = x^2;
  retp(y);
endp;


proc f(x);
  local y;
  y = x^2 .* cos(x);
  retp(y);
endp;
```

```
       44.000000
       88.000000
       132.00000

       44.000000
       88.000000
       132.00000

       44.000000
       88.000000
       132.00000
```

6.2 Résolution numérique des EDO

6.2.1 L'algorithme de Runge Kutta à l'ordre 4

Considérons un système de K équations différentielles ordinaires d'ordre 1 :

$$\left\{\begin{array}{rcl} dx_1(t)/dt & = & f_1(t, x_1(t), \ldots, x_K(t)) \\ & \vdots & \\ dx_k(t)/dt & = & f_k(t, x_1(t), \ldots, x_K(t)) \\ & \vdots & \\ dx_K(t)/dt & = & f_K(t, x_1(t), \ldots, x_K(t)) \end{array}\right. \tag{3}$$

L'écriture vectorielle de ce système est

$$\frac{dX(t)}{dt} = f(t, X(t)) \tag{4}$$

avec

$$X(t) = \begin{pmatrix} x_1(t) \\ \vdots \\ x_k(t) \\ \vdots \\ x_K(t) \end{pmatrix} \quad \text{et} \quad f(t,x) = \begin{pmatrix} f_1(t, x_1, \ldots, x_K) \\ \vdots \\ f_k(t, x_1, \ldots, x_K) \\ \vdots \\ f_K(t, x_1, \ldots, x_K) \end{pmatrix} \tag{5}$$

Nous pouvons résoudre ce système par l'algorithme numérique de Runge Kutta à l'ordre 4. Soient h un pas de discrétisation et X_i la solution numérique de $X(t_i)$ avec $t_i = t_0 + ih$. L'algorithme de Runge Kutta à l'ordre 4 consiste à résoudre les relations suivantes

$$X_{i+1} = X_i + \frac{h}{6}[k_1 + 2k_2 + 2k_3 + k_4] \tag{6}$$

avec

$$\left\{\begin{array}{rcl} k_1 & = & f(t_i, X_i) \\ k_2 & = & f\left(t_i + \frac{h}{2}, X_i + \frac{h}{2}k_1\right) \\ k_3 & = & f\left(t_i + \frac{h}{2}, X_i + \frac{h}{2}k_2\right) \\ k_4 & = & f(t_i + h, X_i + hk_3) \end{array}\right. \tag{7}$$

La procédure `RungeKutta4` comporte 5 arguments. `Xstart` est le vecteur des valeurs initiales X_0 (car nous avons un problème de Cauchy). `N` est le nombre de points de discrétisation du segment $[t_{\text{start}}, t_{\text{end}}]$ et permet de calculer le pas de discrétisation h. L'argument `f` est un pointeur. Nous aurions pu écrire `proc (3) = RungeKutta4(&f,Xstart,tstart,tend,N);`. **Cela veut dire que nous passons un scalaire comme argument et non une procédure.** Nous définissons ensuite `f` localement comme une procédure. `RungeKutta4` comprend alors qu'il doit utiliser ce scalaire pour repérer l'emplacement de la procédure `f`.

```
/*
** rk4.src -  CCF library.
** (C) Copyright 1996 by Thierry Roncalli (TOOLS library).
** (C) Copyright 1997 by Credit Commercial de France (CCF library).
** All Rights Reserved.
**
**  Format                          Purpose                                 Line
** ============================================================================
**  {t,x,dx} = RungeKutta4(&f,Xstart,tstart,tend,N);
**                                  Integrate a system of ODEs using
```

```
**                                the Runge-kutta #4 algorithm             17
**
*/


/*
** RungeKutta4
**
** Purpose: Integrate a system of ODEs using the Runge-Kutta #4 algorithm
**
** Format:  {t,x,dx} = RungeKutta4(&f,Xstart,tstart,tend,N);
**
** Input:   &f - pointeur to a procedure that computes f(t,X);
**      Xstart - vector K*1, initial value X(tstart)
**      tstart - scalar, initial value of the time
**        tend - scalar, final value of the time
**           N - scalar, number of points
**
** Output:   t - vector N*1, time
**           x - matrix N*K, numerical solution of x(t)
**          dx - matrix N*K, numerical solution of dx(t)/dt
**
*/

proc (3) = RungeKutta4(f,Xstart,tstart,tend,N);
  local f:proc;
  local h,t,h2,K,x,dx;
  local i,ti,xi,dxi;
  local k1,k2,k3,k4,k5;

  h = (tend-tstart)/(N-1);
  t = seqa(tstart,h,N);
  h2 = h/2;
  K = rows(Xstart);

  x = zeros(K,N);
  xi = Xstart;
  x[.,1] = xi;

  dx = zeros(K,N);
  dx[.,1] = miss(zeros(K,1),0);

  i = 2;
  do until i > N;

    ti = t[i-1];

    k1 = f(ti,xi);
    k2 = f(ti+h2,xi+h2*k1);
    k3 = f(ti+h2,xi+h2*k2);
    k4 = f(ti+h,xi+h*k3);

    dxi = (h/6)*(k1+2*k2+2*k3+k4);
```

```
    xi = xi + dxi;

    x[.,i] = xi;
    dx[.,i] = dxi;

    i = i + 1;

  endo;

  x = x';
  dx = dx';

  dx = dx / h;

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

Le fichier *ccf.lcg* devient :

```
c:\gauss\src_ccf\bs.src
    europeanbscall                  : proc

c:\gauss\src_ccf\random.src
    rndbernoulli                    : proc
    rndb                            : proc
    rndmn                           : proc

c:\gauss\src_ccf\mco_.src
    mco_                            : proc

c:\gauss\src_ccf\ccf.dec
    _print                          : matrix
    _mco_dw                         : matrix

c:\gauss\src_ccf\mco.src
    mco                             : proc

c:\gauss\src_ccf\rk4.src
    rungekutta4                     : proc
```

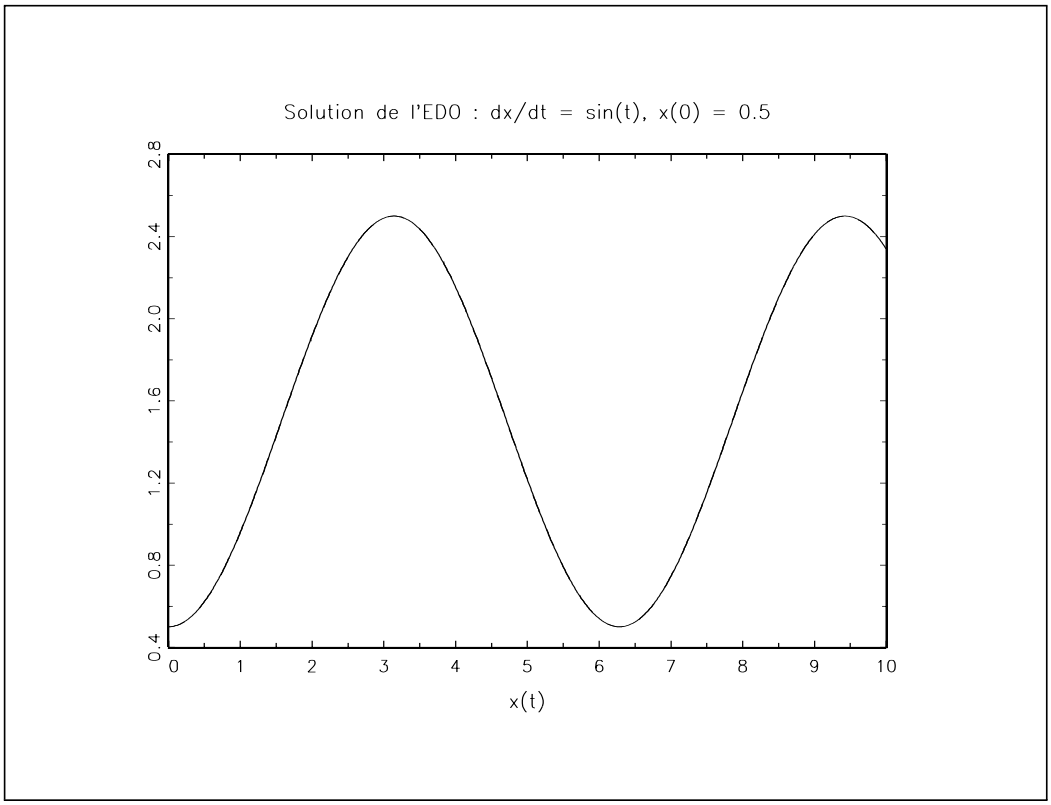
6.2.2 Un exemple d'application

Voyons un exemple simple d'utilisation de `RungeKutta4`. Considérons l'EDO suivante

$$\begin{cases} dx(t)/dt &= \sin(t) \\ x(0) &= 0.5 \end{cases} \tag{8}$$

Nous cherchons à résoudre (8) pour $t \in [0, 10]$. Nous rappelons que la solution est

$$x(t) = 1.5 - \cos(t)$$



Graphique 2:

```
/*
** Resolution d'une EDO
*/

new;
library ccf,pgraph;

proc f(t,x);
  retp(sin(t));
endp;

{t,x,dx} = RungeKutta4(&f,0.5,0,10,1000);

solution = 1.5 - cos(t);

graphset;
  _pdate = "";
  title("Solution de l'EDO : dx/dt = sin(t), x(0) = 0.5");
  xlabel("t");
  xlabel("x(t)");
  graphprt("-c=1 -cf=ccf18.eps -w=5");
  xy(t,x~solution);
```

6.2.3 Le papillon de Lorenz

Le système de Lorenz est un système de trois EDO :

$$\left\{\begin{array}{lcl} \frac{dx(t)}{dt} & = & \sigma\left[y\left(t\right)-x\left(t\right)\right] \\ \frac{dy(t)}{dt} & = & -x\left(t\right)z\left(t\right)+Rx\left(t\right)-y\left(t\right) \\ \frac{dz(t)}{dt} & = & x\left(t\right)y\left(t\right)-\beta z\left(t\right) \end{array}\right. \tag{9}$$

Les paramètres du modèle sont σ, β et R. Pour des valeurs particulières des paramètres, le système converge vers des attracteurs étranges.

```
/*
** Le papillon de Lorenz
*/

new;
library ccf,pgraph;

sigma = 10; beta = 8/3; R = 28;

proc lorenz(t,u);
  local x,y,z,dx,dy,dz,du;

  x = u[1]; y = u[2]; z = u[3];

  dx = sigma*(y-x);
  dy = -x*z + R*x - y;
  dz = x*y - beta*z;

  du = dx|dy|dz;

  retp(du);
endp;


u0 = -8|-9|25;
{t,u,du} = RungeKutta4(&lorenz,u0,0,50,2500);

graphset;
  _pdate = ""; _pnum = 2; _pnumht = 0.18;  _paxht = 0.18;
  title("Le papillon de Lorenz");
  xlabel("x");
  ylabel("y");
  zlabel("z");
  graphprt("-c=1 -cf=ccf19a.eps -w=5");
  xyz(u[1:1000,1],u[1:1000,2],u[1:1000,3]);

graphset;
  begwind;
  window(2,2,0);

    _pdate = ""; _pnum = 2; _pnumht = 0.25;  _paxht = 0.25;
```

```
setwind(1);

  xlabel("x");
  ylabel("y");
  xy(u[.,1],u[.,2]);

setwind(2);

  xlabel("x");
  ylabel("z");
  xy(u[.,1],u[.,3]);

setwind(3);

  xlabel("y");
  ylabel("z");
  xy(u[.,2],u[.,3]);

setwind(4);

  xlabel("t");
  ylabel("x");
  xy(t,u[.,1]);

  graphprt("-c=1 -cf=ccf19b.eps -w=5");

endwind;
```

Pour représenter les trajectoires 3D, nous utilisons la procédure `xyz` de la librairie **PGRAPH**. La commande `begwind` permet d'initialiser le mode fenêtre. Pour le terminer, nous utilisons `endwind;`. La sélection des fenêtres se fait avec `setwind`.

6.2.4 Les EDO d'ordre supérieur à deux

Pour résoudre des systèmes d'EDO d'ordre supérieur à deux, nous les reformulons en système d'ordre un. Considérons par exemple le problème suivant

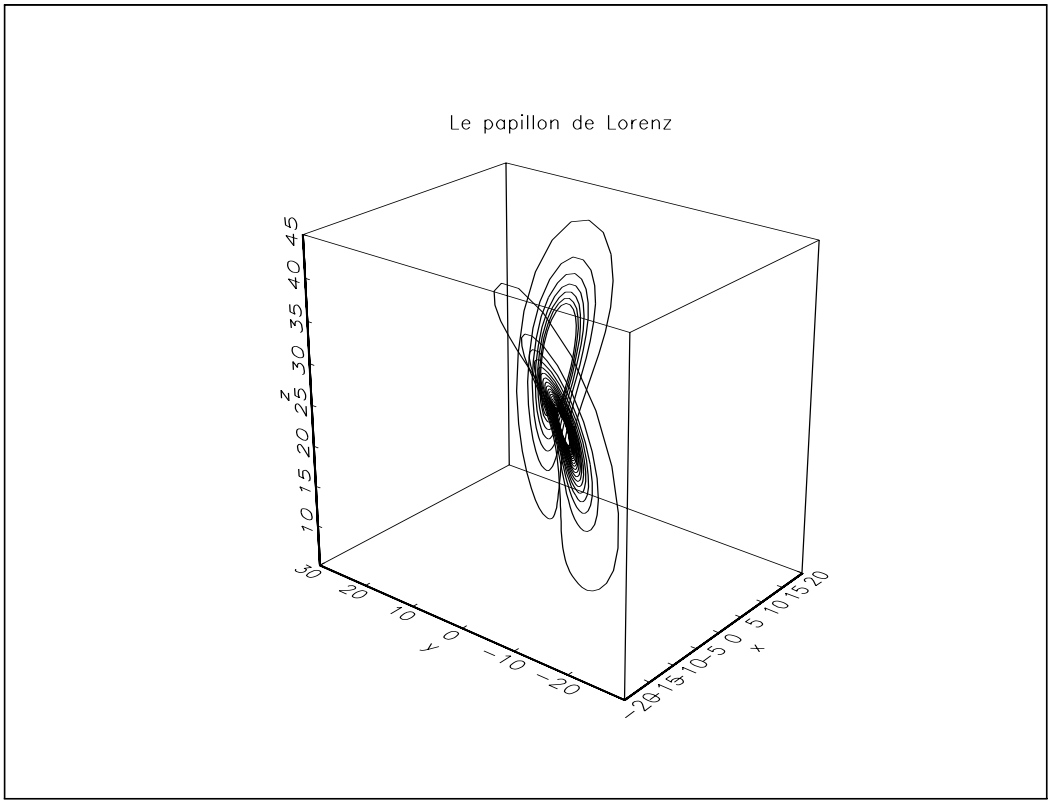
$$\left\{\begin{array}{rcl} x'' & = & -\sin(t) \\ x(0) & = & 0 \\ x'(0) & = & 1 \end{array}\right. \tag{10}$$

Résoudre cette EDO d'ordre 2 revient à résoudre le système

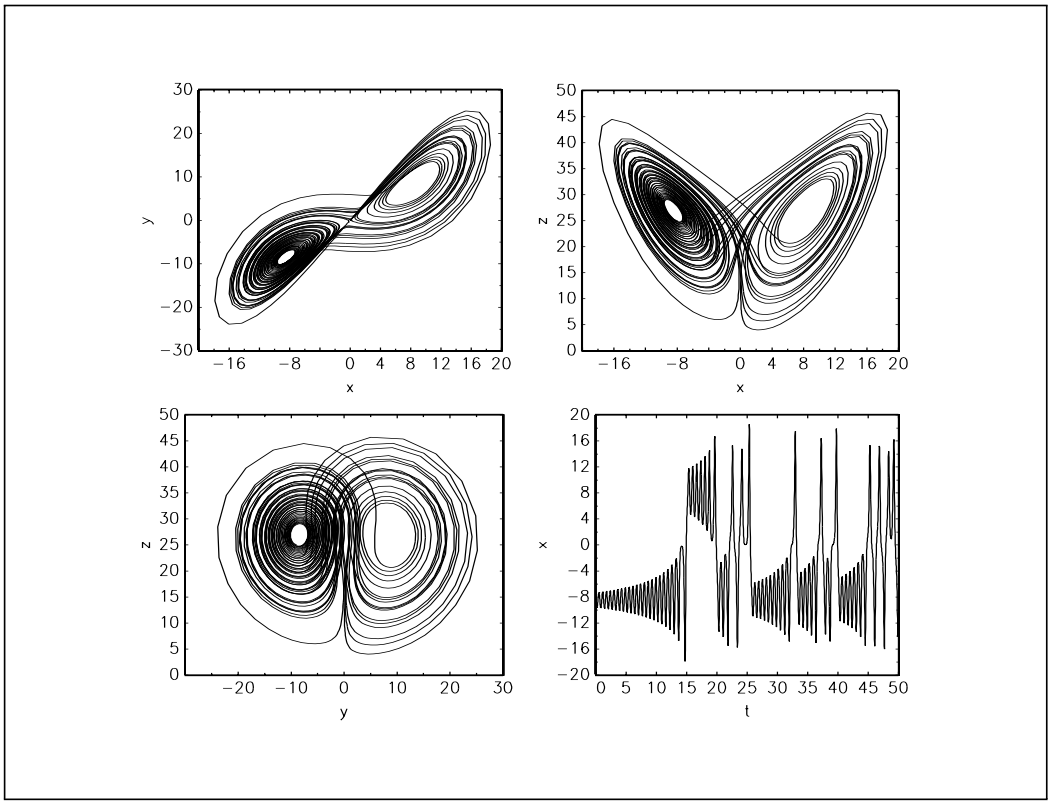
$$\left\{\begin{array}{rcl} x' & = & v(t) \\ v' & = & -\sin(t) \\ x(0) & = & 0 \\ v(0) & = & 1 \end{array}\right. \tag{11}$$

Nous rappelons que la solution est

$$\left\{\begin{array}{rcl} x(t) & = & \sin(t) \\ v(t) & = & \cos(t) \end{array}\right.$$



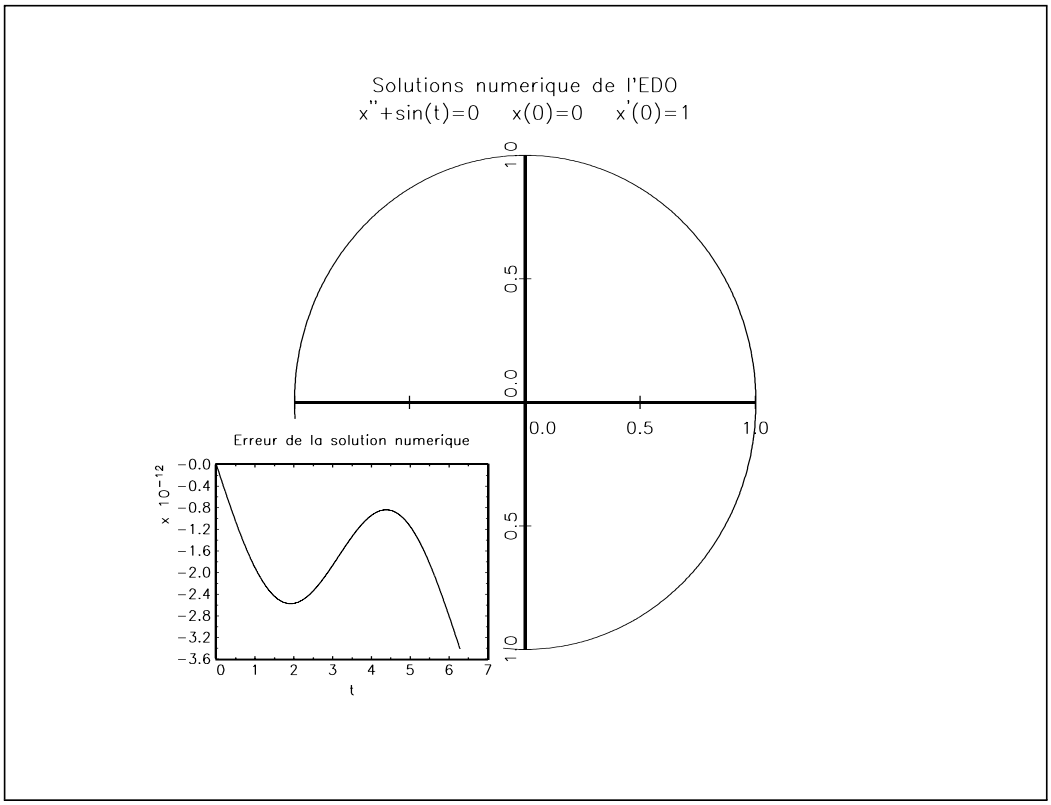
Graphique 3:



Graphique 4:

```
new;
library ccf,pgraph;

proc f(t,x);
  local dx;

  dx = x[2]       |
      -sin(t)     ;

  retp(dx);
endp;

{t,x,dx} = RungeKutta4(&f,0|1,0,2*pi,1000);

xc = complex(x[.,1],x[.,2]);     /* coordonnees complexes de x */

{r,theta} = topolar(xc);         /* coordonnees polaires de x  */


graphset;
  begwind;
  makewind(9,6.855,0,0,0);
  makewind(4,3,0.25,0.25,0);

  _pdate="";

  setwind(1);

    title("Solutions numerique de l'EDO"\
          "\Lx['']+sin(t)=0   x(0)=0   x['](0)=1");
    _pticout = 1;
    polar(r,theta);

  setwind(2);

    graphset;
    _pnum = 2; _paxht = 0.25; _pnumht = 0.25; _ptitlht = 0.25;
    title("Erreur de la solution numerique");
    xlabel("t");
    xy(t,x[.,1]-sin(t));

    graphprt("-c=1 -cf=ccf20.eps -w=5");

endwind;
```

Les procédures `topolar` et `tocart` permettent de passer des coordonnées cartésiennes aux coordonnées polaires et inversement.



Graphique 5:

7 Utilisation des bibliothèques

7.1 La bibliothèque GAUSS

7.1.1 Distinction entre les commandes et les procédures

Nous n'avons pas distingué les commandes des procédures de GAUSS. En fait, le nombre de commandes de GAUSS est assez restreint. La bibliothèque **GAUSS** étant toujours en mémoire, ces procédures peuvent être considérées comme faisant partie du langage GAUSS. Prenons l'exemple de la **procédure** `cumsumc`. Elle est construite avec la **commande** `recserar`.

```
/*
** cumsumc.src
** (C) Copyright 1988-1996 by Aptech Systems, Inc.
** All Rights Reserved.
**
**> cumsumc
**
**  Purpose:    Computes the cumulative sums of the columns of a matrix.
**
**  Format:     y = CUMSUMC( x );
**
**  Input:      x    NxK matrix.
**
**  Output:     y    NxK matrix containing the cumulative sums of the columns
**                   of x.
**
```

```
**  Remarks:   This is based on the recursive series function recserar. That
**             function could be called directly as:
**                 recserar( X, X[1,.], ones(1,cols(X)) );
**             to accomplish the same thing.
**
**  Example:   let x[3,2] = 1  -3
**                          2   2
**                          3  -1;
**
**             y = cumsumc( x );
**
**             The result is:
**
** See Also:  cumprodc, recsercp, recserar
*/

proc cumsumc(x);
    if rows(x) == 1;
        retp(x);
    else;
        retp( recserar(x,x[1,.],ones(1,cols(x)) ) );
    endif;
endp;
```

Nous pouvons initialiser la bibliothèque **GAUSS** avec la commande `library`. Mais cela n'est pas nécessaire. Par défaut, celle-ci est toujours chargée en mémoire.

```
new;
library gauss;

x = rndu(3,4);
y = cumsumc(x);
```

7.1.2 Les procédures de dérivation

La procédure `gradp` permet de calculer la matrice jacobienne d'un système de fonctions ou le vecteur gradient d'une fonction. Nous obtenons la matrice hessienne avec la procédure `hessp`. Les syntaxes sont

```
g = gradp(&f,x);
```

et

```
h = hessp(&f,x);
```

```
new;

proc f1(x);
  local y;
  y = x.*cos(x);
  retp(y);
endp;

proc f2(x);
```

```
  local y;
  y = x[1]*cos(x[2])*x[3];
  retp(y);
endp;


x0 = 1|2|3;
g1 = gradp(&f1,x0);
g2 = gradp(&f2,x0);
h = hessp(&f2,x0);

output file = ccf22.out reset;

print "Matrice jacobienne :" g1;
print;
print "Vecteur gradient :"; print g2;
print "Matrice hessienne :" h;

output off;

Matrice jacobienne :
     -0.30116869        0.0000000        0.0000000
       0.0000000       -2.2347417        0.0000000
       0.0000000        0.0000000       -1.4133525

Vecteur gradient :
      -1.2484405       -2.7278923      -0.41614684
Matrice hessienne :
       0.0000000       -2.7278853      -0.41614900
      -2.7278853        1.2484712      -0.90929612
     -0.41614900      -0.90929612   -6.7282827e-07
```

Remarque : La procédure `gradp` utilise l'algorithme *forward difference*. D'autres procédures de dérivation existent dans la bibliothèque **OPTMUM**, par exemple `gradre` (basée sur la méthode de l'extrapolation de Richardson) ou `gradcd` (basée sur la méthode des différences centrales).

7.1.3 Les procédures d'intégration

Il existe deux catégories de procédures d'intégration. La première concerne les intégrations avec des bornes constantes, alors que pour la seconde catégorie, les bornes d'intégrations sont des fonctions.

`y = intquad1(&f,xl);`	$\int_{\underline{xl}}^{\overline{xl}} f\,(x)\,dx$	Quadrature de Gauss-Legendre
`y = intsimp(&f,xl,tol);`	$\int_{\underline{xl}}^{\overline{xl}} f\,(x)\,dx$	Algorithme de Simpson
`y = intquad2(&f,xl,yl);`	$\int_{\underline{xl}}^{\overline{xl}} \int_{\underline{yl}}^{\overline{yl}} f\,(x,y)\,dy\,dx$	Quadrature de Gauss-Legendre
`y = intquad3(&f,xl,yl,zl);`	$\int_{\underline{xl}}^{\overline{xl}} \int_{\underline{yl}}^{\overline{yl}} \int_{\underline{zl}}^{\overline{zl}} f\,(x,y,z)\,dz\,dy\,dx$	Quadrature de Gauss-Legendre
`y = intgrat2(&f,xl,gl);`	$\int_{\underline{xl}}^{\overline{xl}} \int_{\underline{gl(x)}}^{\overline{gl(x)}} f\,(x,y)\,dy\,dx$	Quadrature de Gauss-Legendre
`y = intgrat3(&f,xl,gl,hl);`	$\int_{\underline{xl}}^{\overline{xl}} \int_{\underline{gl(x)}}^{\overline{gl(x)}} \int_{\underline{hl(x,y)}}^{\overline{hl(x,y)}} f\,(x,y,z)\,dz\,dy\,dx$	Quadrature de Gauss-Legendre

```
new;
```

```
proc f(x);
  local y;
  y = cos(x).*sin(x);
  retp(y);
endp;

output file = ccf23.out reset;

print "_intord = " _intord;
intquad1(&f,1|0);

_intord = 2;
intquad1(&f,1|0);

intsimp(&f,1|0,1e-5);

output off;

_intord =        12.000000
      0.35403671
      0.35253925
      0.35403671
```

7.2 La bibliothèque PGRAPH

Les variables externes ont deux fonctions : une fonction de contrôle et une fonction de sauvegarde des résultats. L'utilisateur n'a pas toujours accès à toutes les variables externes. Dans ce cas, celles-ci permettent d'échanger des informations entre les procédures. Par exemple, la procédure `title` définit une variable externe `_ptitle` qui sera ensuite utilisée par la procédure `xy`. La procédure `graphset` permet d'initialiser l'ensemble des variables globales à leur valeur par défaut.

```
new;
library ccf,pgraph;

rndseed 123456789;

Ns = 500;
X = rndb(50,0.5,Ns,1);

Xbar = cumsumc(X)./seqa(1,1,Ns);
m = 50*0.5;

output file = ccf24.out reset;

print "Avant";
print "=====";

print "_ptitle : " _ptitle;
print "_pdate : " _pdate;
print "_pxlabel : " _pxlabel;
print "_pline : " _pline;

graphset;
```

```
  _pdate = "";
  _pnum = 2;
  title("Illustration de la loi des grands nombres");
  _pline = 1~1~0~m~Ns~m~1~5~0;
  xlabel("Nombre de simulations");
  graphprt("-c=1 -cf=ccf24.eps -w=5");
  xy(seqa(1,1,Ns),Xbar);


print;
print "Apres";
print "=====";

print "_ptitle : " _ptitle;
print "_pdate : " _pdate;
print "_pxlabel : " _pxlabel;
print "_pline : " _pline;


output off;
```

```
Avant
=====
_ptitle :
_pdate : GAUSS
_pxlabel :
_pline :        0.0000000

Apres
=====
_ptitle : Illustration de la loi des grands nombres
_pdate :
_pxlabel : Nombre de simulations
_pline :        1.0000000        1.0000000        0.0000000        25.000000
       500.00000        25.000000        1.0000000        5.0000000
       0.0000000
```

7.3 La bibliothèque OPTMUM

Cette bibliothèque considère le problème suivant

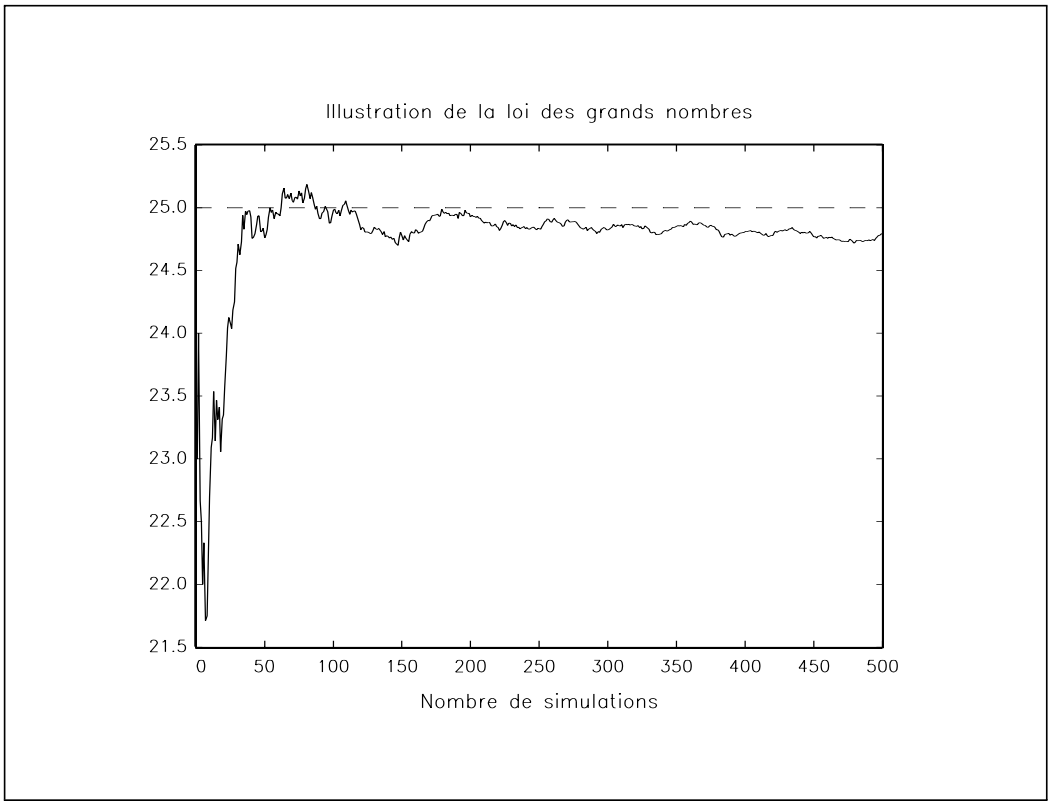
$$X_0 = \underset{X \in \mathbb{R}^N}{\arg\min}\, f(X) \tag{12}$$

La syntaxe de la procédure `optmum` est

```
{x0,f0,g0,retcode} = optmum(&f,sv);
```

`&f` est le pointeur de la procédure qui calcule $f(X)$ et `sv` est le vecteur $\mathbb{R}^N$ des valeurs de départ pour l'algorithme d'optimisation. `optmum` définit quatre arguments de sortie : `x0` la solution du problème (12), `f0` la valeur prise par la fonction à l'optimum, `g0` le vecteur gradient correspondant et `retcode` un scalaire indiquant le code de retour de l'algorithme. Pour les problèmes de maximisation, nous utilisons la correspondance suivante

$$\arg\max f(X) = \arg\min -f(X) \tag{13}$$



Graphique 6:

7.3.1 Exemple n°1

Considérons la fonction

$$f(x,y) = (x-3)^2 + (y-x-3)^2 + xy \tag{14}$$

L'écriture vectorielle est

$$f(X) = (X_1-3)^2 + (X_2-X_1-3)^2 + X_1X_2 \tag{15}$$

avec

$$X = \begin{bmatrix} x \\ y \end{bmatrix} \tag{16}$$

```
new;
library optmum;

proc fonction(x);
  local x1,x2,y;

  x1 = x[1];
  x2 = x[2];

  y = (x1-3)^2 + (x2-x1-3)^2 + x1*x2;

  retp(y);
endp;

sv = 1|1;
```

```
output file = ccf25.out reset;

{xmin,fmin,gmin,retcode} = optmum(&fonction,sv);

call optprt(xmin,fmin,gmin,retcode);

output off;

================================================================================
 OPTMUM Version 3.1.4                                          9/19/97   2:04 pm
================================================================================

return code =    0
normal convergence

Value of objective function        7.714286

Parameters    Estimates    Gradient
----------------------------------------
P01              0.8572        0.0000
P02              3.4286        0.0000

Number of iterations        8
Minutes to convergence     0.00267
```

La procédure `optprt` permet l'affichage formaté des résultats. Attention, cette procédure utilise aussi des informations définies par les variables externes. Quel est le problème avec le code suivant ?

```
{x1,f1,g1,retcode1} = optmum(&f,sv1);

{x2,f2,g2,retcode2} = optmum(&g,sv2);

call optprt(x1,f1,g1,retcode1);
```

7.3.2 Exemple n°2

La bibliothèque **OPTMUM** utilise de nombreuses variables globales. Certaines permettent de contrôler l'optimisation, par exemple la tolérance sur le gradient (`_opgtol`) ou le choix de l'algorithme (`_opalgr`), alors que d'autres permettent de récupérer des résultats, par exemple la valeur finale de la matrice hessienne (`_opfhess`).

```
new;
library optmum;

proc fonction(x);
  local x1,x2,y;

  x1 = x[1];
  x2 = x[2];

  y = (x1-3)^2 + (x2-x1-3)^2 + x1*x2;
```

```
  retp(y);
endp;

sv = 1|1;

output file = ccf26.out reset;

_opgtol = 0.000000001;
_opalgr = 5;
{xmin,fmin,gmin,retcode} = optprt(optmum(&fonction,sv));

print;
print "Matrice Hessienne : ";
print _opfhess;

output off;

================================================================================
 OPTMUM Version 3.1.4                                          9/19/97   2:08 pm
================================================================================

return code =    0
normal convergence

Value of objective function        7.714286

Parameters    Estimates    Gradient
----------------------------------------
P01              0.8571        0.0000
P02              3.4286       -0.0000

Number of iterations        3
Minutes to convergence     0.00100

Matrice Hessienne :

       4.0000125      -0.99999489
     -0.99999489        2.0000001
```

7.3.3 Exemple n°3

Considérons l'estimation d'un modèle AR(1)-ARCH(1)

$$\left\{ \begin{array}{rcl} y_t & = & \phi y_{t-1} + u_t \\ u_t | \, \mathcal{F}_{t-1} & \sim & \mathcal{N}\left(0, h_t^2\right) \\ h_t^2 & = & \alpha_0 + \alpha_1 h_{t-1}^2 \end{array} \right. \tag{17}$$

```
new;
library optmum;

rndseed 123;

y = recserar(rndn(1000,1),0,0.3);
```

```
proc ml(theta);
  local phi,alpha0,alpha1,u,h2,logL;

  phi = theta[1];
  alpha0 = theta[2]^2;
  alpha1 = theta[3]^2;

  u = y-phi*lag1(y);
  h2 = alpha0 + alpha1*u.*u;

  u = trimr(u,1,0);
  h2 = trimr(h2,1,0);

  logL = -0.5*ln(2*pi)-0.5*ln(h2)-0.5*(u^2)./h2;

  retp(-sumc(logL));
endp;

sv = 0.5|0.5|0.0;

output file = ccf27.out reset;

_opparnm = "phi"|"alpha0"|"alpha1";

{xmin,fmin,gmin,retcode} = optprt(optmum(&ml,sv));

output off;
```

```
================================================================================
 OPTMUM Version 3.1.4                                          9/19/97   2:31 pm
================================================================================

return code =    0
normal convergence

Value of objective function     1447.844864

Parameters    Estimates    Gradient
-----------------------------------------
phi              0.3133        0.0025
alpha0          -1.0308        0.0001
alpha1           0.0000        0.0000

Number of iterations       21
Minutes to convergence     0.03850
```

La valeur prise par α_0 est négative. Ne jamais oublier que GAUSS est un langage numérique et que nous pouvons obtenir des résultats parfois étranges.

7.4 La bibliothèque ARIMA

Cette bibliothèque contient des procédures pour l'identification, l'estimation, la prévision et la simulation des processus univariés ARIMA(p,d,q).

```
new;
library arima;

u = rndn(1000,1)*0.5;

y = recserar(u-0.25*lag1(u),0.25|0.75,0.25|-0.35);

output file = ccf28.out reset;

{b,ll,e,covb,aic,sbc} = arima(0,y,2,0,1,0);

output off;

Model:  ARIMA(2,0,1)


Final Results:

Iterations Until Convergence:   4

Log Likelihood:   -727.699601         Number of Residuals: 1000
AIC           :   1461.399202         Error Variance     : 0.251638914
SBC           :   1476.122468         Standard Error     : 0.501636237

DF: 997     Adj. SSE: 250.955947159        SSE: 250.883997616

            Coefficients     Std. Errors      T-Ratio    Approx. Prob.
AR1           0.17815960      0.07887107      2.25887          0.02411
AR2          -0.35543338      0.03011740    -11.80160          0.00000
MA1           0.20476184      0.08415907      2.43303          0.01515


Total Computation Time: 0.82 (seconds)

AR Roots and Moduli:

Real :    0.25062   0.25062
Imag.:    1.65851  -1.65851
Mod. :    1.67734   1.67734

MA Root:   4.88372
```

7.5 La bibliothèque TSM

TSM contient les procédures suivantes :

1. ARMA processes.
   (a) **arma_ML**: Conditional maximum likelihood for Vector ARMA models
   (b) **arma_CML**: Conditional maximum likelihood for Vector ARMA models under linear restrictions
   (c) **arma_to_VAR1**: VAR(1) representation of a Vector ARMA process

(d) **arma_roots**: roots of the VAR(1) representation of a Vector ARMA process

(e) **canonical_arma**: Canonical representation of a Vector ARMA process (infinite AR and MA orders)

(f) **arma_autocov**: Autocovariances and autocorrelations of a Vector ARMA process

(g) **arma_impulse**: Responses to Forecast Errors of a Vector ARMA process

(h) **arma_orthogonal**: Responses to Orthogonal Impulses of a Vector ARMA process

(i) **arma_fevd**: Forecast Error Variance Decomposition of a Vector ARMA process

(j) **arma_to_SSM**: State space form of a Vector ARMA model

(k) **Hankel**: Hankel matrix for multivariate time series

2. VARX processes.

(a) **varx_LS**: Multivariate Least Squares Estimation of VARX processes

(b) **varx_CLS**: Multivariate Least Squares Estimation of VARX processes under linear restrictions

(c) **varx_ML**: Maximum Likelihood of VARX processes

(d) **varx_CML**: Maximum Likelihood of VARX processes under linear restrictions

3. Spectral analysis.

(a) **fourier**: Fourier transform

(b) **inverse_fourier**: Inverse Fourier transform

(c) **fourier2**: Fourier transform of two real time series

(d) **PDGM**: Periodogram of a univariate time series

(e) **PDGM2**: Periodogram of a multivariate time series

(f) **CPDGM**: Cross-periodogram

(g) **CSpectrum**: Coherency, cross-amplitude spectra and phase spectra

(h) **Smoothing**: Data windowing in the frequency domain

4. Maximum Likelihood Estimation.

(a) Time domain estimation.

i. **TD_ml**: Estimation in the time domain

ii. **TD_cml**: Estimation in the time domain under linear restrictions

iii. **TDml_derivatives**: Computes the Jacobian, the gradient, the Hessian and the Information matrices in the time domain

(b) Frequency domain estimation for univariate processes.

i. **FD_ml**: Estimation in the frequency domain

ii. **FD_cml**: Estimation in the frequency domain under linear restrictions

iii. **FDml_derivatives**: Computes the Jacobian, the gradient, the Hessian and the Information matrices in the frequency domain

5. Univariate Models.

(a) **sm_LL**: Local level/random walk plus noise model

(b) **sm_LLT**: Local linear trend model

(c) **BSM**: Basic structural model

(d) **sm_cycle**: Cycle model

(e) **arfima**: Fractional ARMA model with constraints

(f) **canonical_arfima**: Canonical representation of a fractional ARMA process

(g) **sgf_arfima**: Spectral generating function of a fractional ARMA process

6. State space models and the Kalman filter.

(a) **SSM**: Print the state space model

(b) **SSM_build**: Build the state space model

(c) **SSM_ic**: Initial conditions for the state space model

(d) **KFiltering**: Kalman filtering

(e) **KF_matrix**: Matrices defined by the Kalman Filter

(f) **KF_gain**: Compute the gain matrices K_t

(g) **KF_ml**: Maximum likelihood of the innovations process

(h) **KSmoothing**: Smoothing

(i) **KForecasting**: Forecasting

(j) **ARE**: Algebraic Riccati equation

(k) **sgf_SSM**: Spectral generating function of a time-invariant state space model

(l) **SSM_autocov**: Autocovariances and autocorrelations of a time-invariant state space model

(m) **SSM_impulse**: Responses to Forecast Errors of a time-invariant state space model

(n) **SSM_orthogonal**: Responses to Orthogonal Impulses of a time-invariant state space model

(o) **SSM_fevd**: Forecast Error Variance Decomposition of a time-invariant state space model

(p) **SSM_Hankel**: Hankel matrix of a time-invariant state space model

7. Resampling and simulation.

(a) **Bootstrap**: Boot-strapping a matrix

(b) **bootstrap_SSM**: Bootstrapping state space models

(c) **surrogate**: FT Surrogate data technique

(d) **Kernel**: Density estimation with the Kernel method

(e) **RND_arma**: Simulation of Vector ARMA processes

(f) **RND_arfima**: Simulation of fractional ARMA processes

(g) **RND_SSM**: Simulation of state space models

8. Estimation tools for time series analysis.

(a) **FLS**: Flexible least squares

(b) **GFLS**: Generalized flexible least squares of KALABA and TESFATSION [1990]

(c) **GFLS2**: Generalized flexible least squares of LÜTKEPOHL and HERWARTZ [1996]

(d) **GMM**: Generalized method of moments

(e) **RLS**: Recursive least squares

9. Time-Frequency analysis.

   (a) Quadrature mirror filters.

      i. **Coiflet**: Coiflet filters

      ii. **Daubechies**: Daubechies filters

      iii. **Haar**: Haar filters

      iv. **Pollen**: Pollen filters

   (b) Wavelet analysis.

      i. Periodic discrete wavelet transform.

         A. **iwt**: Inverse wavelet transform of a vector

         B. **iwt_matrix**: matrix associated with the inverse wavelet transform

         C. **wt**: Wavelet transform of a vector

         D. **wt_matrix**: matrix associated with the wavelet transform

      ii. Wavelet Tools.

         A. **extract** : Wavelet decomposition coefficients subband extraction

         B. **insert**: Wavelet decomposition coefficients subband insertion

         C. **Scalogram**: Scalogram of the wavelet decomposition coefficients

         D. **select**: Wavelet decomposition coefficients subband selection

         E. **split**: Wavelet decomposition coefficients subband split

         F. **wPlot**: Wavelet decomposition coefficients plot

   (c) Wavelet packet analysis.

      i. Wavelet packet transform.

         A. **iwpkt**: Inverse wavelet packet transform

         B. **wpkPlot**: Wavelet packet table plot

         C. **wpkt**: Wavelet packet transform

      ii. Wavelet packet basis.

         A. **Basis**: Wavelet packet basis selection

         B. **BasisPlot**: *Time-frequency* plane tilings plot

         C. **BestBasis**: Best basis selection (pruning algorithm)

         D. **BestLevel**: Best level selection

         E. **Entropy**: Shannon entropy cost function

         F. **isBasis**: check whether $\mathcal{B}$ is a basis

         G. **LogEnergy**: Log-energy cost function

         H. **LpNorm**: ℓ^p norm cost function

   (d) Thresholding methods.

      i. **SemiSoft**: Semi-soft shrinkage

ii. **Thresholding**: Quantile thresholding
iii. **VisuShrink**: Visu shrinkage (or universal thresholding)
iv. **WaveShrink**: Wavelet shrinkage (hard and soft shrinkages)

10. Filters.

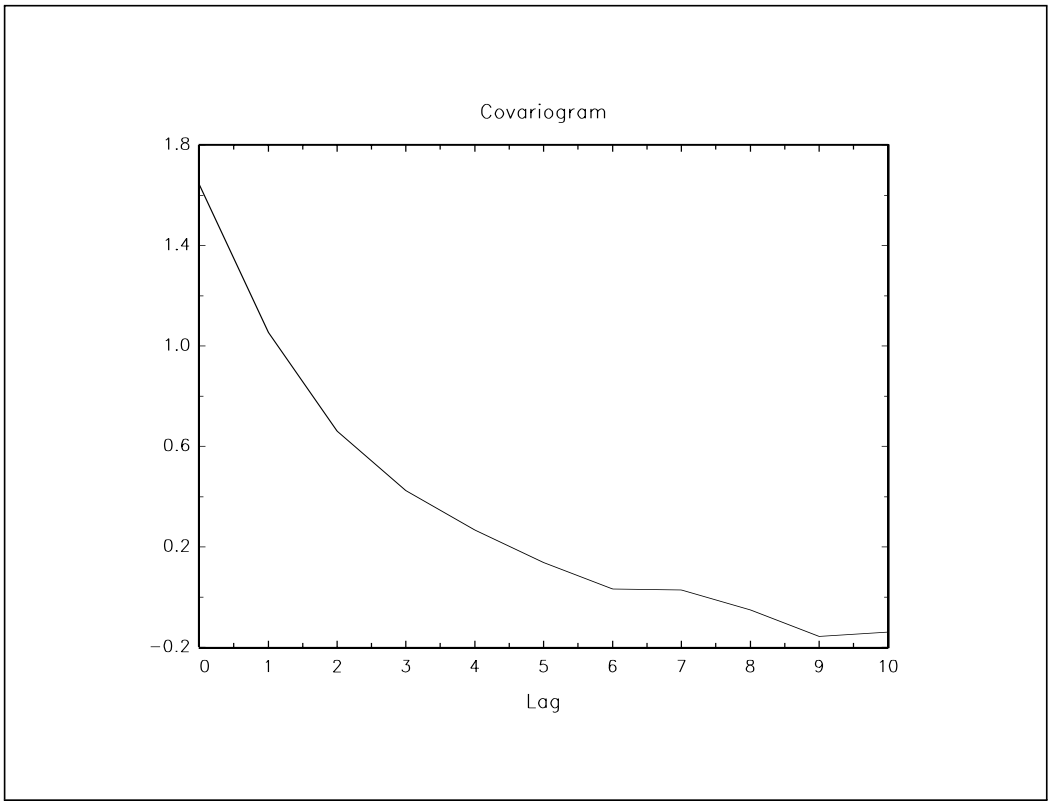
    (a) **arma_Filter**: ARMA filtering
    (b) **fractional_Filter**: Fractional filtering
    (c) **garch_Filter**: GARCH filtering
    (d) **Linear_Filter**: Linear filtering
    (e) **Savistky_Golay**: Savitsky-Golay smoothing filter

11. TSM tools.

    (a) Matrix operators.
        i. **Commutation_**: Commutation matrix
        ii. **Duplication_**: Duplication matrix
        iii. **Elimination_**: Elimination matrix
        iv. **vech_**: vech operator
        v. **xpnd_**: xpnd operator
        vi. **xpnd2**: Procedure for coding square matrices
    (b) Optimization under linear restrictions.
        i. **Explicit_to_Implicit**: Convert explicit linear restrictions $C\theta = c$ to implicit linear restrictions $\theta = R\gamma + r$
        ii. **Implicit_to_Explicit**: Convert implicit linear restrictions $\theta = R\gamma + r$ to explicit linear restrictions $C\theta = c$
        iii. **optmum2**: General nonlinear optimization under linear restrictions

De nombreuses autres procédures sont disponibles, mais elles ne font pas partie de la bibliothèque **TSM**. Elles sont considérées comme des exemples d'application des procédures de base. Cela explique le nombre important d'exemples (la dernière version en compte 270).

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

Nobs = 500;
s = seqa(1,1,Nobs);

let SIGMA = {1 0.5,0.5 1};
Y0 = miss(0,0);
x = RND_arma(0.6|0.3|0|0.5,1,0,SIGMA,Y0,Nobs);
x = x - meanc(x)';

x1 = x[.,1]; x2 = x[.,2];

_fourier = 0;

{lambda,I} = PDGM(x1);
```



Graphique 7:

```
cov = real(inverse_fourier(I));

graphset;
    _pdate = ""; _pnum = 2;
    title("Covariogram");
    _pltype = 6|1;
    xlabel("Lag");
    graphprt("-c=1 -cf=ccf29.eps -w=5");
    xy(seqa(0,1,11),cov[1:11]);
```

8 De nouvelles procédures

8.1 Quelques procédures très simples

L'utilisateur peut construire de nombreuses procédures de base très utiles qui complètent assez bien la bibliothèque GAUSS.

```
/*
**  Calcule les rendements
*/

proc (1) = Rendements(x);
  local xlag,r;

  xlag = lag1(x);
```

```
  r = (x - xlag) ./ xlag;

  retp(r);
endp;


/*
**  Calcule la trace d'une matrice
*/

proc (1) = Mtrace(A);
  local tr;

  tr = sumc(diag(A));

  retp(tr);
endp;


/*
**  Calcule la fonction de probabilite de la loi binomiale
*/

proc (2) = pdfb(N,p);
  local x,q,C,prob;

  x = seqa(0,1,N+1);
  q = 1-p;
  C = N! ./ ( x! .* (N-x)! );
  prob = C .* (p.^x) .* (q.^(N-x));

  retp(x,prob);
endp;


/*
**  Calcule la fonction de densite de la loi exponentielle
*/

proc pdfe(x,lambda);
  retp(lambda.*exp(-lambda.*x));
endp;
```

Voici un exemple d'utilisation de ces procédures :

```
new;
library ccf;

output file = ccf30.out reset;

{x,prob} = pdfb(10,0.5);

call printfmt(x~prob,1);
```

```
x = rndu(10,1);
xx = x .* x';

print;
print "Trace = " Mtrace(xx) sumc(x^2);

output off;
```

```
               0     0.0009765625
               1      0.009765625
               2      0.043945313
               3        0.1171875
               4       0.20507813
               5       0.24609375
               6       0.20507813
               7        0.1171875
               8      0.043945313
               9      0.009765625
              10     0.0009765625

Trace =        3.9650942        3.9650942
```

8.2 L'analyse en composantes principales

```
/*
**  Analyse en Composantes Principales
**
*/

proc (4) = ACP(x);
  local k,N;
  local sigma,M,va,ve,w,QLT,QLTc;

  k = cols(x);
  N = rows(x);

  sigma = sqrt((N-1)/N)*stdc(x);
  x = (x - meanc(x)') ./ sigma';

  M = (x'x)/N;
  {va,ve} = eighv(M);
  w = rev(sortmc(va~ve',1));
  va = w[.,1];
  ve = w[.,2:(K+1)]';

  QLT = va/k;
  QLTc = cumsumc(QLT);

  _acp_sat = ve.*sqrt(va');
  _acp_repres_var = _acp_sat^2;
  _acp_contr_var = _acp_repres_var^2 ./ va';

  retp(va,ve,QLT,QLTc);
```

```
endp;

new;
library ccf;

data = { 10 10 10,
         18  8 12,
         4 17 12,
         6 10 17,
         10 12 14,
         10 12  8};

output file = ccf31.out reset;

{va,ve,QLT,QLTc} = acp(data);

print "Valeurs propres : ";
print va;
print "Vecteurs propres : ";
print ve;
print "Qualite de representation des variables";
print _acp_repres_var;

output off;

Valeurs propres :

       1.7584525
       1.0707209
      0.17082664
Vecteurs propres :

      0.71757176      -0.11397629       0.68709547
     -0.68114401      -0.32075233       0.65814951
     -0.14537403       0.94028046       0.30779708
Qualite de representation des variables

      0.90544341      0.013909302      0.080647286
      0.81584664       0.11015796      0.073995399
     0.037162446       0.94665360      0.016183952
```

8.3 Résolution numérique des EDS

Considérons l'équation différentielle stochastique

$$\left\{ \begin{array}{rcl} dX(t) & = & \mu(t, X(t))\, dt + \sigma(t, X(t))\, dW(t) \\ X(t_0) & = & X_0 \end{array} \right. \tag{18}$$

Sous certaines conditions, la solution de cette équation existe et est unique. C'est un processus de diffusion. L'algorithme d'Euler-Maruyama permet d'obtenir une simulation de ce processsus de diffusion. Il correspond à la récursion suivante

$$\left\{ \begin{array}{rcl} x_{i+1} & = & x_i + \mu(t_i, x_i)\, h + \sigma(t_i, x_i)\, \sqrt{h} u_i \\ x_0 & = & X_0 \end{array} \right. \tag{19}$$

avec h le pas de mesure ($t_i = t_0 + ih$), u_i un nombre aléatoire gaussien $\mathcal{N}(0,1)$ et x_i la simulation de la variable aléatoire $X(t_i)$. Dans la procédure suivante, nous avons vectorisé l'algorithme, puisque nous obtenons M simulations du processus. `mu` et `sigma` sont des pointeurs de procédure. Il convient donc de les déclarer localement comme des procédures. Remarquez la ligne de commande `x[1,.]  = x0.*ones(1,M);` Quelles sont les dimensions de `x0` ? Pouvons-nous simuler des processus de diffusion différents ?

```
proc (2) = Euler_Maruyama(x0,mu,sigma,t0,T,N,M);
  local mu:proc,sigma:proc;
  local h,t_,x,k1,u,i,ti,xi;

  h = (T-t0)/N;
  t_ = seqa(t0,h,N+1);
  x = zeros(N+1,M);

  x[1,.] = x0.*ones(1,M);
  k1 = sqrt(h);

  u = rndn(N+1,M);

  i = 1;
  do until i > N;
    ti = t_[i];
    xi = x[i,.];
    x[i+1,.] = xi + mu(ti,xi)*h + k1.*sigma(ti,xi).*u[i+1,.];
    i = i + 1;
  endo;

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

Nous pouvons utiliser la procédure `Euler_Maruyama` pour illustrer la méthode de Monte Carlo en finance. Considérons le problème de valorisation d'une option de type Black et Scholes. Le prix du sous-jacent $S(t)$ est un mouvement brownien géométrique

$$\left\{ \begin{array}{rcl} dS(t) & = & \mu S(t)\, dt + \sigma S(t)\, dW(t) \\ S(t_0) & = & S_0 \end{array} \right. \tag{20}$$

La solution de la prime de l'option d'achat est donnée par le théorème de Feynman-Kac :

$$C(t_0) = e^{-r\tau} E'\left[(S(T) - K)_+ \middle| \mathcal{F}_t \right] \tag{21}$$

Le changement de probabilité est donné par le théorème de Girsanov. Sous la mesure de probabilité neutre au risque, nous avons

$$\left\{ \begin{array}{rcl} dS(t) & = & rS(t)\, dt + \sigma S(t)\, dW'(t) \\ S(t_0) & = & S_0 \end{array} \right. \tag{22}$$

Pour calculer la solution de la prime de l'option d'achat, nous pouvons employer la méthode de Monte Carlo. Soit S_n une simulation de $S(T)$, nous avons

$$C(t_0) \simeq e^{-r\tau} \left(\frac{1}{N} \sum_{n=1}^{N} (S_n - K)_+ \right) \tag{23}$$

```
new;
library ccf,pgraph;

S0 = 100;
K = 98;
sigma = 0.15;
tau = 95/365;
r = 0.08;

C = EuropeanBScall(S0,K,sigma,tau,r);

proc A(t,x);
  retp(r*x);
endp;

proc B(t,x);
  retp(sigma*x);
endp;

rndseed 123;
Ns = 300;

{t,S} = Euler_Maruyama(S0,&A,&B,0,tau,100,Ns);

ST = S[101,.]';
PayOff = (ST - K) .* ( (ST-K) .> 0 );
Csimul = cumsumc(PayOff) ./ seqa(1,1,Ns);

graphset;
  _pdate = ""; _pnum = 2;
  title("Simulation de Monte-Carlo de la prime d'une option d'achat"\
        "\LEtude de la convergence");
  _pline = 1~1~0~C~Ns~C~1~2~10;
  xlabel("Nombre de Simulations");
  ylabel("Prime");
  ytics(3,7,1,10);
  graphprt("-c=1 -cf=ccf32.eps -w=5");
  xy(seqa(1,1,Ns),Csimul);
```

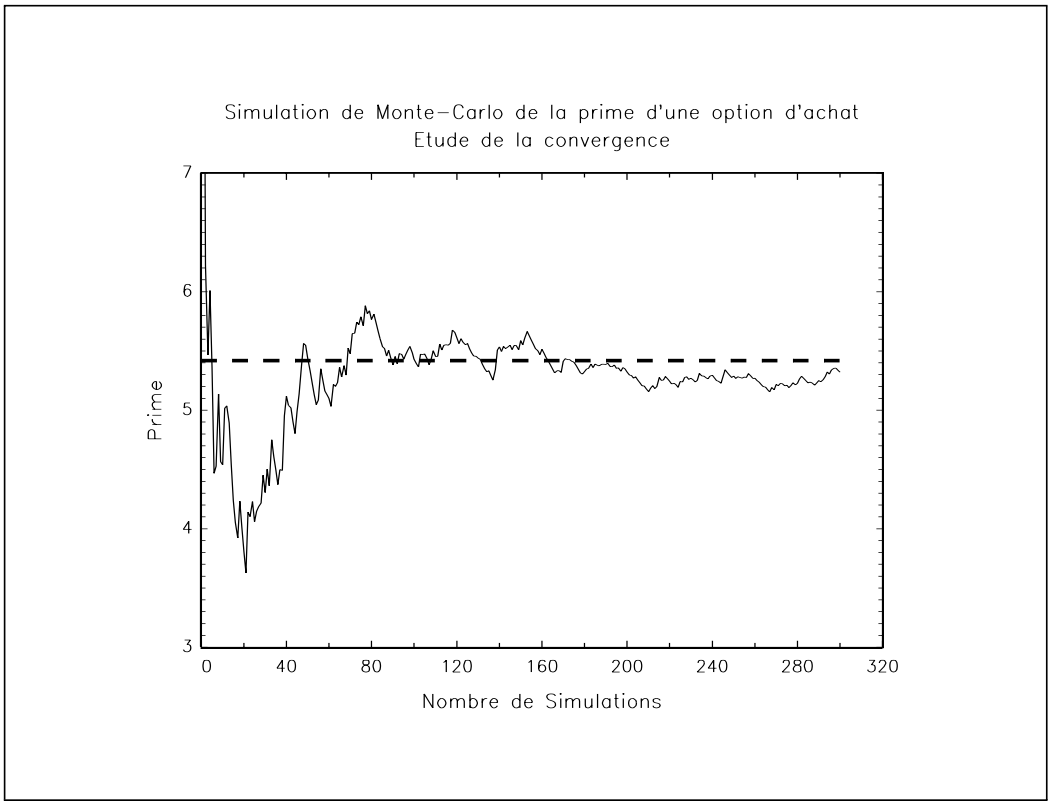
9 Éléments de programmation avancée

9.1 Gestion des erreurs

9.1.1 La commande ERRORLOG

La commande `errorlog` permet d'afficher un message d'erreur. Les commandes `screen on` et `screen off` n'ont pas d'effet sur l'affichage des erreurs. Nous employons la commande `error` pour définir un code d'erreur, qui peut être testé par `scalerr`.

```
new;

output file = ccf33.out reset;
```



Graphique 8:

```
beta = mco(rndn(100,1),rndn(200,4));
scalerr(beta);

beta = mco(rndn(10,1),rndn(10,20));
scalerr(beta);

beta = mco(rndn(10,1),rndn(10,2));
scalerr(beta);

y = rndn(10,1);
x = rndn(10,2);
beta = mco(y,x~x);
scalerr(beta);


output off;


proc mco(y,x);
  local Nobs,K,beta;

  Nobs = rows(y);
  K = cols(x);

  if rows(x) /= Nobs;
    ERRORLOG "erreur : y et x n'ont pas le meme nombre d'observations.";
```

```
    retp(error(1));
  endif;

  if K > Nobs;
    ERRORLOG "erreur : Le nombre d'observations est moins grand que le "\
             "nombre de variables.";
    retp(error(2));
  endif;

  beta = invpd(x'x)*x'y;

  retp(beta);
endp;

erreur : y et x n'ont pas le meme nombre d'observations.
       1.0000000
erreur : Le nombre d'observations est moins grand que le nombre de variables.
       2.0000000
       0.0000000

C:\GAUSS\CCF2\CCF33.PRG(40) : error G0121 : Matrix not positive definite
Currently active call: MCO [40]
```

9.1.2 L'autorisation d'un calcul impossible

Dans certains cas, il est intéressant d'autoriser un calcul impossible (l'inversion d'une matrice singulière, la décomposition de Cholesky d'une matrice non p.d.s., etc), par exemple pour ne pas interrompre le programme ou pour indiquer l'erreur correspondante ou encore pour changer d'algorithme. Ceci est permis par la commande `trap`.

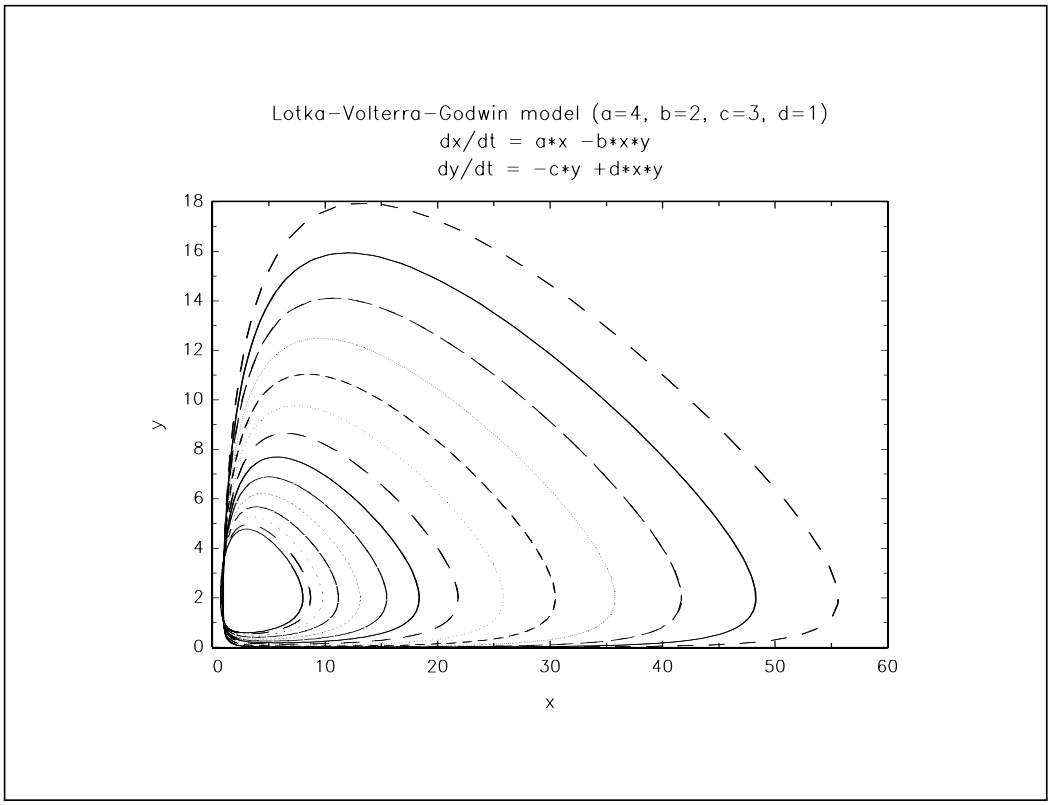
```
new;

output file = ccf34.out reset;

y = rndn(10,1);
x = rndn(10,2);
beta = mco(y,x~x);
scalerr(beta);
ismiss(beta);

output off;


proc mco(y,x);
  local old,invxx,beta;

  old = trapchk(1);
  trap 1,1;
    invxx = invpd(x'x);
  trap old,1;

  if scalerr(invxx);
    ERRORLOG "erreur : La matrice X'X est singuliere";
```

```
    retp(error(0));
    /*
      invxx = pinv(x'x);
    */
  endif;

  beta = invxx*x'y;

  retp(beta);
endp;

erreur : La matrice X'X est singuliere
       0.0000000
       1.0000000
```

9.2 Compilation et externalité des variables

Lors de la compilation, le statut des variables externes n'est résolu que lors de l'exécution. Voyons un exemple. Dans la procédure `lvg`, les variables `a`, `b`, `c` et `d` ne sont pas initialisées.

```
new;
library ccf,pgraph;

declare external a,b,c,d;

a = 4; b = 2; c = 3; d = 1;

proc lvg(t,z);
  local x,y,dz;

  x = z[1];
  y = z[2];

  dz = a*x - b*x*y           |
       -c*y + d*x*y          ;

  retp(dz);
endp;

x = {};
y = {};
z0 = 1|1;

i = 0.1;
do until i > 1.5;
  {t,z,dz} = RungeKutta4(&lvg,z0,0,3,500);
  x = x~z[.,1];
  y = y~z[.,2];
  i = i + 0.1;
  c =  c + i;
endo;

graphset;
```



Graphique 9:

```
_pnum = 2; _pdate = "";
fonts("simplex simgrma");
title("Lotka-Volterra-Godwin model (a=4, b=2, c=3, d=1)"\
      "\Ldx/dt = a*x -b*x*y "\
      "\Ldy/dt = -c*y +d*x*y");
xlabel("x");
ylabel("y");
graphprt("-c=1 -cf=ccf35.eps -w=5");
xy(x,y);
```

Remarque : Il n'était pas nécessaire de déclarer comme externes les paramètres, car les variables d'une procédure non locales sont considérées automatiquement comme des variables externes.

9.3 L'échange d'informations entre procédures

9.3.1 L'échange par les procédures auxiliaires

Nous pouvons utiliser des procédures auxiliaires pour échanger des informations. C'est le cas de la procédure `optmum`.

```
/* optmum.src - General Nonlinear Optimization
** (C) Copyright 1988-1994 by Aptech Systems, Inc.
** All Rights Reserved.
**
**   Written by Ronald Schoenberg
*/
```

```
proc (4) = optmum(fnct,x0);
    local x,f0,g,retcode;
    local Lopfhess,Lopitdta,LLoutput;

    LLoutput = __output;

    { x,f0,g,retcode,Lopfhess,Lopitdta } = _optmum(fnct,x0,
      _opalgr,_opdelta,_opdfct,_opditer,_opgdmd,_opgdprc,
      _opgrdh,_opgtol,_ophsprc,_opkey,_opmbkst,_opmdmth,
      _opmiter,_opmtime,_opmxtry,_opparnm,_oprteps,_opshess,
      _opstep,_opstmth,_opusrch,_opusrgd,_opusrhs,LLoutput,
      __title);

    _opfhess = Lopfhess;
    _opitdta = Lopitdta;

    retp(x,f0,g,retcode);
endp;

/*
** optutil.src
** (C) Copyright 1988-1994 by Aptech Systems, Inc.
** All Rights Reserved.
*/

proc (6) = _optmum(fnct,x0,
    Lopalgr,Lopdelta,Lopdfct,Lopditer,Lopgdmd,Lopgdprc,
    Lopgrdh,Lopgtol,Lophsprc,Lopkey,Lopmbkst,Lopmdmth,
    Lopmiter,Lopmtime,Lopmxtry,Lopparnm,Loprteps,
    Lopshess,Lopstep,Lopstmth,Lopusrch,Lopusrgd,
    Lopusrhs,LLoutput,LLtitle);

    local x,g,s,h,iter,ky,old,vof,d,dfct,f0,parnms,bksteps,dx,
        smallval,relgrad,algrm,stepm,pg,k0,k1,k2,lr,lf,ll,np,rteps,
        tstart,oldt,w0,w1,w2,w3,w4,w5,w6,w7,w8,w9,w10,chiter,h0,
        ttime,isctu,mask,fmt,omat,ret,hsmeth,beta,vof1,vofs,ds,it,oldfmt,
        tmpop,e1,e2;
    local midalg,midstp,midhs,hessm,c0,c1,skip;
    local Lopfhess,Lopitdta;
    local fnct:proc;

/*
  ...
*/

  retp(x,vof,g,scalerr(ret),Lopfhess,Lopitdta);
endp;
```

9.3.2 L'échange par les variables globales

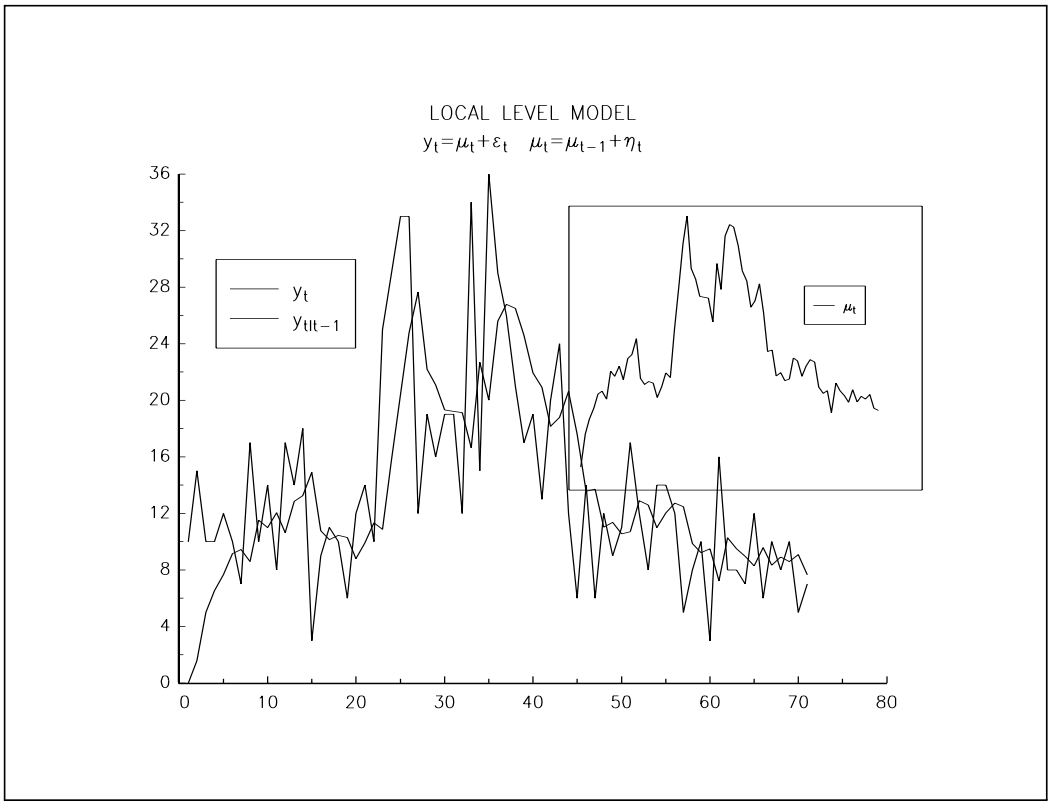
L'échange d'information entre procédures se fait généralement par des variables externes. C'est le cas par exemple de la bibliothèque **TSM**. La procédure `SSM_build` définit le modèle espace-état. Ce modèle est en fait stocké dans des variables externes que peut utiliser la procédure `KFiltering`.

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

load purse[] = purse.asc;
Nobs = rows(purse);

{beta,stderr,Mcov,Logl} =  sm_LL(purse,1|1);

Z = 1; d = 0; H = beta[2]^2;
T = 1; c = 0; R = 1; Q = beta[1]^2;

call SSM_build(Z,d,H,T,c,R,Q,0);
call KFiltering(purse,0,0);

yc = KF_matrix(1);     /* y[t|t-1]      */
a = KF_matrix(3);      /* a[t]          */

indx = seqa(1,1,Nobs);

graphset;
  fonts("simplex simgrma");
  _pdate = ""; _pnum = 2;
  begwind;
  makewind(9,6.855,0,0,0);
  makewind(4,3,5,2.5,1);
  setwind(1);
    _pframe = 0;
    title("LOCAL LEVEL MODEL"\
          "\Ly]t[=\202m\201]t[+\202e\201]t[  "\
          "\202m\201]t[=\202m\201]t-1[+\202c\201]t[");
    _plegctl = {2 6 1 4}; _pltype = 6;
    _plegstr = "y]t[\0y]t|t-1[";
    xy(indx,purse~yc);
  nextwind;
    title(""); _pnum = 0; _pframe = 0; _paxes = 0; _pbox = 15;
    _plegctl = {2 9 6 4};
    _plegstr = "\202m\201]t[";
    _pltype = 6; _pcolor = 14;
    xy(indx,a);
    graphprt("-c=1 -cf=ccf37.eps -w=5");
  endwind;
```

9.4 Exporter une procédure dans une autre procédure : application à la méthode du maximum de vraisemblance

Pour exporter une procédure `proc1` dans une autre procédure `proc2`, nous utilisons une variable externe dont la valeur est le pointeur de `proc1`. **Comment modifier le code si nous spécifions `local ml:proc;` dans la procédure `MaximumVraisemblance` ?**

```
/*
** ccf.dec - CCF.
** (C) Copyright 1997 by Credit Commercial de France
```



Graphique 10:

```
** All Rights Reserved.
**
** Global variable definitions for the CCF library
*/

declare matrix _print = 1;
declare matrix _mco_DW;
declare matrix _acp_sat;
declare matrix _acp_repres_var;
declare matrix _acp_contr_var;
declare matrix _mv_fonc;
declare matrix _mv_param;
declare matrix _mv_Mcov = 1;   /* estimateur de la matrice de covariance */
/*
** ccf.ext - CCF.
** (C) Copyright 1997 by Credit Commercial de France
** All Rights Reserved.
**
** External variable declarations for the CCF library
*/

external matrix _print;
external matrix _mco_DW;
external matrix _acp_sat;
external matrix _acp_repres_var;
```

```
external matrix _acp_contr_var;
external matrix _mv_fonc;
external matrix _mv_param;
external matrix _mv_Mcov;

proc _mv_critere(beta);
  local f,logL;

  f = _mv_fonc;
  local f:proc;

  logL = f(beta);

  retp( -sumc(logL) );
endp;

proc (4)  = MaximumVraisemblance(ml,sv);
  local ml_;
  local Nobs,k,ddl,beta,LogL,grd,retcode,h,Jac,Mcov;
  local stderr,tstudent,pvalue,nom,mask,omat,fmt;

  ml_ = ml;
  local ml_:proc;

  Nobs = rows(ml_(sv));
  k = rows(sv);
  ddl = Nobs-k;

  _mv_fonc = ml;    /* C'est un pointeur */

  {beta,LogL,grd,retcode} = optmum(&_mv_critere,sv);

  LogL = -LogL;

  if _mv_Mcov == 3;
    /*
       Estimateur de White
    */
  elseif _mv_Mcov == 2;
    /*
       Estimateur OPG
    */
  else;
    h = hessp(&_mv_critere,beta);  /* _mv_critere est une procedure */
    jac = gradp(ml,beta);            /* ml est un pointeur */
    Mcov = inv(h);
  endif;

  stderr = sqrt(diag(Mcov));
  tstudent = beta./stderr;
  pvalue = 2*cdftc(abs(tstudent),ddl);

  if _print == 1;
```

```
    print ftos(Nobs,"Nombre d'observations : %*.*lf",23,0);
    print ftos(k,"Nombre de parametres : %*.*lf",24,0);
    print ftos(ddl,"Nombre de degres de liberte : %*.*lf",17,0);
    print ftos(Logl,"Valeur de la fonction de vraisemblance : %*.*lf",14,5);
    print;

    if ( _mv_param /= 0 ) and ( rows(_mv_param) == K );
      nom = _mv_param;
    else;
      nom = 0$+"P"$+ftocv(seqa(1,1,K),2,0);
    endif;

    mask = 0~1~1~1~1;
    let fmt[5,3] =  "-*.*s"  8 8  "*.*lf" 16 6 "*.*lf" 16 6
                    "*.*lf" 17 6  "*.*lf" 16 6;
    omat = nom~beta~stderr~tstudent~pvalue;
    print "Parametres      estimation       std.err.    "\
        "  t-statistic         p-value   ";
    print "----------------------------------------"\
          "--------------------------------";
    call printfm(omat,mask,fmt);

  endif;

  retp(beta,stderr,Mcov,Logl);
endp;
```

Nous pouvons utiliser la méthode du maximum de vraisemblance pour estimer les modèles de réponses qualitatives. Considérons les modèles de choix binomiaux de la forme

$$y_i = \begin{cases} 1 & \text{avec la probabilité } p_i \\ 0 & \text{avec la probabilité } q_i = 1 - p_i \end{cases} \tag{24}$$

Nous spécifions p_i de la façon suivante

$$p_i = \Pr\{y_i = 1\} = F\left(x_i^\top \beta\right)$$

où F est la fonction de répartition d'une variable aléatoire. La vraisemblance de l'observation i est alors

$$L_i(y_i, \beta) = p_i^{y_i} (1 - p_i)^{1 - y_i} \tag{25}$$

d'où

$$\ell_i(y_i, \beta) = \ln L_i(y_i, \beta) = y_i \ln p_i + (1 - y_i) \ln(1 - p_i)$$

Les deux modèles les plus couramment utilisés sont les modèles logit et probit. Pour le modèle logit, F est la fonction logistique et pour le modèle probit, F est la fonction de répartition de la loi $\mathcal{N}(0, 1)$.

Considérons 10 individus. Les facteurs A et B peuvent-ils expliquer le fait que certains de ces individus possèdent le caractère C ?

Possède la caractère C	Facteur A	Facteur B
oui	14	15
non	12	18
non	2	25
oui	16	12
non	6	55
non	4	12
oui	12	12
non	3	23
oui	7	7
oui	10	55

Nous définissons la variable y_i avec $y_i = 1$ si l'individu i possède le caractère C et $y_i = 0$ dans le cas contraire. Les variables explicatives `x` regrouppent une constante et les facteurs A et B. Pour trouver $\hat{\beta}_{\mathrm{MV}}$, nous avons choisi comme point initial $\{0.1, 0.1, 0.1\}$. Généralement, l'estimateur des moindres carrés ordinaires sert à initialiser l'algorithme.

```
new;
library ccf,optmum;

let x[10,3] =  1  14  15
               1  12  18
               1   2  25
               1  16  12
               1   6  55
               1   4  12
               1  12  12
               1   3  23
               1   7   7
               1  10  55;

let y = 1 0 0 1 0 0 1 0 1 1;

fn logistique(x) = exp(x)./(1+exp(x));

proc logit(beta);
  local p,q;
  p = logistique(x*beta);
  q = 1 - p;
  retp( y.*ln(p) + (1-y).*ln(q) );
endp;

output file = ccf38.out reset;

_mv_param = "Constante"|"Var1"|"Autre";

{beta,stderr,Mcov,Logl} = MaximumVraisemblance(&logit,zeros(3,1)+0.1);

print;
print "           Individu            O/N        prob";
print "-------------------------------------------------------";
prob = logistique(x*beta);
call printfmt(seqa(1,1,10)~y~prob,1);
```

```
output off;

Nombre d'observations :                      10
Nombre de parametres :                        3
Nombre de degres de liberte :                 7
Valeur de la fonction de vraisemblance :       -4.07340

Parametres      estimation       std.err.      t-statistic         p-value
-----------------------------------------------------------------------
Constant       -3.430582        2.592568        -1.323237        0.227339
Var1            0.432967        0.246640         1.755463        0.122612
Autre          -0.010347        0.044839        -0.230754        0.824105

           Individu            O/N        prob
-------------------------------------------------
               1                1       0.92242418
               2                0       0.82903113
               3                0       0.05607714
               4                1       0.96684108
               5                0       0.19752265
               6                0       0.13908911
               7                1       0.83765142
               8                0      0.085516016
               9                1       0.38408546
              10                1       0.58176323
```

9.5 La programmation modulaire

9.5.1 Décomposition d'un algorithme complexe en algorithmes simples

Pour des raisons faciles à comprendre, il est souvent préférable de décomposer un algorithme complexe en algorithmes simples et de faire correspondre à chacun d'eux une procédure. C'est ainsi que fonctionne la procédure `optmum`. Elle appelle de nombreuses procédures correspondant à des algorithmes élémentaires (calcul de la matrice hessienne, du vecteur gradient, implémentation d'une line search cubique, de brent, etc).

```
/*
** optmum.lcg - Optimization Library
** (C) Copyright 1988-1996 by Aptech Systems, Inc.
** All Rights Reserved.
*/

optmum.dec

gradient.dec

optutil.src
    _optmum                          : proc
    _strsch                          : proc
    _sctu2                           : proc
    _stepl2                          : proc
    _deriv2                          : proc
```

```
    _gdfd                           : proc
    _gdcd                           : proc
    _hssp                           : proc
    _stepb2                         : proc
    _usrsch2                        : proc
    _brackt2                        : proc
    _brent2                         : proc
    _half2                          : proc

optmum.src

gradient.src
```

9.5.2 Les procédures génériques

9.5.3 Le coût de la modularité

Il y a deux sources de coût de la modularité : l'appel de la procédure (c'est-à-dire la recherche de l'emplacment mémoire) et la copie des arguments d'entrée. Ce coût peut être très élevé pour les procédures très simples, comme dans l'exemple suivant :

```
new;

output file = ccf39.out reset;

@<---------------------- 1 ---------------------------->@

t0 = hsec;

i = 1;
do until i > 50000;
  x = sqrt(i)*ln(i)^2;
  i = i + 1;
endo;

print ftos((hsec-t0)/100,"Temps de calcul : %lf",5,2);

@<---------------------- 2 ---------------------------->@

t0 = hsec;

i = 1;
do until i > 50000;
  x = fonction1(i);
  i = i + 1;
endo;

print ftos((hsec-t0)/100,"Temps de calcul : %lf",5,2);

@<---------------------- 3 ---------------------------->@

t0 = hsec;

i = 1;
```

```
do until i > 50000;
  x = fonction2(i);
  i = i + 1;
endo;

print ftos((hsec-t0)/100,"Temps de calcul : %lf",5,2);

output off;

proc fonction1(i);
  retp( sqrt(i)*ln(i)^2 );
endp;


proc fonction2(i);
  local x1,x2;
  x1 = sqrt(i);
  x2 = ln(i)^2;
  x = x1*x2;
  retp( x );
endp;
```

```
Temps de calcul :  2.64
Temps de calcul :  3.57
Temps de calcul :  4.56
```

En général, ce coût n'est pas élevé pour les calculs numériques *lourds*.

```
new;

output file = ccf40.out reset;

@<---------------------- 1 ---------------------------->@

t0 = hsec;

i = 1;
do until i > 10;
  u = rndn(10*i,10*i);
  u = u'u;
  x = orth(invpd(u)*chol(u));
  i = i + 1;
endo;

print ftos((hsec-t0)/100,"Temps de calcul : %lf",5,2);

@<---------------------- 2 ---------------------------->@

t0 = hsec;

i = 1;
do until i > 10;
  x = fonction(i);
  i = i + 1;
```

```
endo;

print ftos((hsec-t0)/100,"Temps de calcul : %lf",5,2);

output off;

proc fonction(i);
  local u,x;
  u = rndn(10*i,10*i);
  u = u'u;
  x = orth(invpd(u)*chol(u));
  retp( x );
endp;

Temps de calcul :  2.63
Temps de calcul :  2.64
```

9.6 Les procédures sans argument de retour

9.6.1 Un exemple

9.6.2 Les procédures de type reset

Les procédures sans arguments peuvent servir à initialiser les variables globales à leurs valeurs par défaut.

```
proc (0) = CCFset;
  _print = 1;
  _mv_Mcov = 1;
  retp;
endp;

new;
library ccf;
CCFset;

/*
 ...
*/

CCFset;
```

9.6.3 Les mots clés

```
/*
**> chdir
**
**  Purpose:   Change directory.
**
**  Format:    chdir dirstr;
**
**  Input:     dirstr    literal, directory to change to.
**
**  Remarks:   This is for interactive use. Use ChangeDir() in
```

```
**              a program.
**
**              The working directory is listed in the status
**              report on the UNIX version.
*/

#ifDOS

keyword chdir(dir);
        dir = "cd " $+ dir;
        dos ^dir;
endp;

#else

keyword chdir(dir);
    if (ChangeDir(dir) $== "");
        print "Cannot change directory to " dir;
    endif;
endp;

#endif
```

9.7 Programmation événementielle

Il existe quelques programmes GAUSS (par exemple **ComPar** ou **BackPropagation**) qui ne nécessitent aucune connaissance sur le langage et qui utilisent un environnement Input/Output. GAUSS possède quelques primitives pour la programmation événementielle. La version finale pour Windows 95/NT comprendra sûrement un ensemble complet de commandes GUI (façon o-matrix). C'est donc l'occasion d'apprendre à créer des interfaces pour des utilisateurs qui ne connaissent pas GAUSS.

```
new;

call DataMenu;

proc (0) = DataMenu;
  local st;
  local titre,choix;

  let string st[4,1] = "Conversion Donnees ---> GAUSS"
                       "Conversion GAUSS ---> Donnees"
                       "Modification Base de donnees GAUSS"
                       "Quitter";

   titre = "Gestion des bases de donnees GAUSS";

   choix = 0;
   do until choix == 4;

     choix = _menu_general(titre,st);

     if choix == 1;
```

```
      call _DataMenu_1;
    elseif choix == 2;
      call _DataMenu_2;
    elseif choix == 3;
      call _DataMenu_3;
    endif;

  endo;

  cls;
  end;

  retp;
endp;

proc (0) = _DataMenu_1;
  local st;
  local titre,choix;

  let string st[4,1] = "CVS ---> GAUSS"
                       "SAS ---> GAUSS"
                       "ASCII ---> GAUSS"
                       "Retour au menu princpal";

  titre = "Conversion d'une base de donnees au format GAUSS";

  choix = 0;
  do until choix == 4;

    choix = _menu_general(titre,st);

    if choix == 1;
      call cvs2gauss;
    endif;

  endo;

  call DataMenu;
  retp;
endp;

proc (0) = _DataMenu_2;
  local st;
  local titre,choix;

  let string st[4,1] = "GAUSS ---> CVS"
                       "GAUSS ---> SAS"
                       "GAUSS ---> ASCII"
                       "Retour au menu princpal";

  titre = "Conversion d'une base de donnees GAUSS";

  choix = 0;
```

```
   do until choix == 4;

     choix = _menu_general(titre,st);

   endo;

   call DataMenu;

   retp;
endp;


proc (0) = _DataMenu_3;
  local st;
  local titre,choix;

  let string st[5,1] = "Chargement d'une base GAUSS"
                       "Renommer une variable"
                       "Ajouter une variable"
                       "Ajouter une observation"
                       "Retour au menu princpal";

   titre = "Modification d'une base de donnees GAUSS";

   choix = 0;
   do until choix == 5;

     choix = _menu_general(titre,st);

   endo;

   call DataMenu;
   retp;
endp;


proc _menu_general(titre,st);
  local N,longueur,i,choix;
  local str0,str1,indx,l;

  N = rows(st);
  longueur = zeros(N,1);
  i = 1;
  do until i > N;
    longueur[i] = strlen(st[i]);
    i = i + 1;
  endo;

  longueur = maxc(longueur);

  if longueur <= 20;
    longueur = 20;
  endif;
```

```
  longueur = longueur + 10;

  cls;

  str0 = chrs(45*ones(1,longueur+2));
  str1 = chrs(124~zeros(1,longueur)~124);

  /* Cadre */

  l = maxc(ceil((longueur/2) - strlen(titre)/2) | 0);
  locate 2,10+l; print upper(titre);

  locate 5,10; print str0;
  i = 6;
  do until i > (2*N+7);
    locate i,10; print str1;
    i = i + 1;
  endo;
  locate (2*N+8),10; print str0;

  /* Choix */

  i = 1;
  do until i > N;
    locate (4+2*i),12; print ftos(i,"%lf.",1,0)$+st[i];
    i = i + 1;
  endo;
  locate (2*N+6),15; print "Votre choix : ";

  indx = seqa(1,1,N);
  choix = 0;

  do until sumc(choix .== indx);
    locate (2*N+6),30; print "           ";
    locate (2*N+6),30; choix = con(1,1);
  endo;

  retp(choix);
endp;

proc (0) =  cvs2gauss;

  retp;
endp;
```

Ecran n°1

```
     GESTION DES BASES DE DONNEES GAUSS


-----------------------------------------------
| 1.Conversion Donnees ---> GAUSS             |
|                                             |
| 2.Conversion GAUSS ---> Donnees             |
|                                             |
| 3.Modification Base de donnees GAUSS        |
|                                             |
| 4.Quitter                                   |
|                                             |
|    Votre choix :  ?                         |
|                                             |
-----------------------------------------------
```

Ecran n°2

CONVERSION D'UNE BASE DE DONNEES AU FORMAT GAUSS

```
-----------------------------------
| 1.CVS ---> GAUSS                |
|                                 |
| 2.SAS ---> GAUSS                |
|                                 |
| 3.ASCII ---> GAUSS              |
|                                 |
| 4.Retour au menu princpal       |
|                                 |
|    Votre choix :  ?             |
|                                 |
-----------------------------------
```

Ecran n°3

CONVERSION D'UNE BASE DE DONNEES GAUSS

```
-----------------------------------
| 1.GAUSS ---> CVS                |
|                                 |
| 2.GAUSS ---> SAS                |
|                                 |
| 3.GAUSS ---> ASCII              |
|                                 |
| 4.Retour au menu princpal       |
|                                 |
|    Votre choix :  ?             |
|                                 |
-----------------------------------
```

Ecran n°4

MODIFICATION D'UNE BASE DE DONNEES GAUSS

```
---------------------------------------
| 1.Chargement d'une base GAUSS       |
|                                     |
| 2.Renommer une variable             |
|                                     |
| 3.Ajouter une variable              |
|                                     |
| 4.Ajouter une observation           |
|                                     |
| 5.Retour au menu princpal           |
|                                     |
|    Votre choix :  ?                 |
|                                     |
---------------------------------------
```

9.8 Le style de programmation

Il est conseillé d'adopter un style de programmation. Cela rend les programmes plus clairs et plus faciles à déchiffrer. Il est donc préférable d'utiliser des majuscules, des minuscules, des tabulations, des espacements, des sauts de lignes et adopter des conventions pour les variables externes. Choisissez avec précaution le nom des variables ou des procédures. Pensez surtout qu'un utilisateur externe peut un jour ou l'autre décider de modifier votre programme. Facilitez lui donc la tâche.