Formation GAUSS pour le CCF

Niveau I

Thierry Roncalli

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

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

Table des matières

1 Introduction **1**
- 1.1 Configuration de GAUSS . . . 1
- 1.2 Les modes commande et éditeur . . . 1
- 1.3 La gestion des erreurs de programmation . . . 1

2 Les matrices **1**
- 2.1 Déclaration des matrices . . . 1
- 2.2 Les opérateurs de concaténation . . . 2
- 2.3 Les commandes `ones`, `zeros` et `eye` . . . 4
- 2.4 Les sous-matrices . . . 5
- 2.5 Chargement de données ascii . . . 6

3 Les principales commandes de GAUSS **8**
- 3.1 Les opérateurs matriciels . . . 9
- 3.2 Les fonctions . . . 9
- 3.3 Les commandes mathématiques . . . 10
- 3.4 Les commandes statistiques . . . 11
- 3.5 Les commandes de séquence . . . 11
- 3.6 Les commandes de manipulation de matrices . . . 12

4 Exemples d'utilisation des commandes de GAUSS **12**
- 4.1 Les opérateurs matriciels . . . 12
- 4.2 Les opérateurs matriciels $E \times E$ . . . 13
- 4.3 Les fonctions . . . 17
- 4.4 Les commandes mathématiques . . . 18
- 4.5 Les commandes statistiques . . . 21
- 4.6 Les commandes de séquence . . . 21
- 4.7 Les commandes de manipulation de matrices . . . 21
- 4.8 Les nombres particuliers de GAUSS . . . 24
- 4.9 Un exemple complet : Les Moindres Carrés Ordinaires . . . 26

5 Les opérateurs conditionnels et relationnels **28**
- 5.1 Les opérateurs conditionnels . . . 28
- 5.2 Les opérateurs relationnels . . . 30

6 Les structures de contrôle **31**
- 6.1 Les boucles . . . 31
- 6.2 Les instructions `if` . . . 33
- 6.3 Les instructions `break` et `continue` . . . 34

7 Quelques exemples **35**
- 7.1 Moyenne mobile . . . 35
- 7.2 Simulation d'un processus ARCH . . . 36
- 7.3 Vectorisation . . . 37

8 La bibliothèque graphique **38**
- 8.1 La commande `xy` . . . 38
- 8.2 Les fenêtres . . . 41
  - 8.2.1 La commande `window` . . . 41

8.2.2 La commande `makewind` 42

9 Les bases de données **44**
9.1 Gestion des bases 45
9.2 Les formats v89 dos et v96 universal 47

10 Les entrées et sorties **48**
10.1 La construction d'une base de données GAUSS à partir d'un fichier ASCII : l'exemple de la base HFDF93 d'Olsen & Associates 48
10.2 Quelques exemples 51

11 Introduction au niveau II **55**
11.1 Qu'est-ce qu'une procédure ? 55
11.2 Qu'est-ce qu'un pointeur ? 56
11.3 Qu'est-ce qu'une variable externe ? 57
11.4 Qu'est-ce qu'une bibliothèque ? 57

1 Introduction

1.1 Configuration de GAUSS

Deux fichiers contrôlent la configuration de GAUSS et de la librairie graphique PGRAPH : *gauss.cfg* et *pqgrun.cfg*. Le premier fichier permet de définir les chemins d'accès de GAUSS, la mémoire disponible ou encore la valeur par défaut de certaines variables. Le type d'imprimante, les modes de conversion (par exemple, mode portrait ou paysage) et d'impression sont définis dans le fichier *pqgrun.cfg*.

1.2 Les modes commande et éditeur

Le mode éditeur est réservé à la programmation. Nous utilisons le mode commande pour modifier l'environnement (changement de répertoire par exemple avec la commande `chdir`), exécuter les programmes et sortir des résultats.

1.3 La gestion des erreurs de programmation

Lors de la compilation, GAUSS affiche dans le mode commande les erreurs de programmation. Celles-ci sont aussi enregistrées dans le fichier *gauss.err*. GAUSS accompagne ces erreurs d'un commentaire indiquant le numéro de ligne et le type d'erreur.

2 Les matrices

GAUSS manipule un type **unique** de données. Il n'y a pas de types `int`, `float`, `double`, `short`, etc comme en C++. L'objet que manipule GAUSS est une matrice complexe double précision. Un scalaire correspond donc à une matrice de dimension (1×1). Un vecteur est une matrice avec une seule colonne. Il n'y a pas de différence entre un entier et un réel. Cela implique que la syntaxe `x = 1` est équivalente à `x = 1.0`. Une matrice réelle est une matrice complexe avec une partie imaginaire nulle.

Remarque : Il existe un second type de données correspondant à une chaîne de caractère.

2.1 Déclaration des matrices

Pour déclarer et initialiser une matrice, nous employons la commande `let` en spécifiant les nombres de lignes et de colonnes. `let x[r,c]` permet de définir une matrice X de dimension $(r \times c)$. Considérons la matrice suivante

$$x = \begin{pmatrix} 1 & 2.3 & 100 \\ 2 & 3 & 5 \\ 2 & -3 & -10 \\ 6 & 25 & 12 \end{pmatrix} \tag{1}$$

Nous utilisons la commande `print` pour afficher cette matrice.

```
output file = ccf1.out reset;

let x[4,3] =  1   2.3  100.
              2   3    5
              2  -3   -10
              6   25   12 ;
```

```
print x;

output off;
```

Nous remarquons que GAUSS n'a pas fait de différence pour coder l'entier 1 et le réel 100..

```
       1.0000000        2.3000000        100.00000
       2.0000000        3.0000000        5.0000000
       2.0000000       -3.0000000       -10.000000
       6.0000000        25.000000        12.000000
```

Nous pouvons aussi déclarer une matrice en utilisant les accolades `{}` et les virgules (qui permettent de spécifier les différentes lignes de la matrice). Il existe différentes formes de commentaires, la plus utilisée est la syntaxe C ou C++ `/* ... */`.

```
/* Declaration de la matrice x[4,3] */

let x  =  {1 2.3 100. , 2 3 5 , 2 -3 -10 , 6 25 12};

print x;  /* Affichage de la matrice x */
```

Ne jamais oublier que l'objet manipulé par GAUSS est une **matrice complexe** ! Ne vous étonnez donc pas d'obtenir des nombres complexes par exemple pour l'inversion de matrices réelles quasi-singulières.

```
output file = ccf3.out reset;

let x  =  {1+1i, 2.3, 100. };
print x;

output off;
```

```
       1.0000000 +        1.0000000i
       2.3000000
       100.00000
```

2.2 Les opérateurs de concaténation

GAUSS possède deux opérateurs de concaténation : `~` pour la concaténation verticale et | pour la concaténation verticale

$$Z = \begin{bmatrix} X & \vdots & Y \end{bmatrix} \quad \texttt{Z=X~Y;}$$
$$Z = \begin{bmatrix} X \\ \dots \\ Y \end{bmatrix} \quad \texttt{Z=X|Y;} \tag{2}$$

```
new;

/* Declaration de la matrice x[4,3] */

let x  =  {1 2.3 100. , 2 3 5 , 2 -3 -10 , 6 25 12};

outwidth 256;
```

```
output file = ccf4.out reset;

y = x~x;
print "x~x = " y;

y = x|x;
print "x|x = " y;

output off;
```

Pour sauvegarder les résultats dans un fichier, nous employons les commandes `output file = ... reset;` et `output off;`. Le fichier *ccf4.out* se présente de la façon suivante :

```
x~x =
      1.0000000       2.3000000       100.00000       1.0000000       2.3000000       100.00000
      2.0000000       3.0000000       5.0000000       2.0000000       3.0000000       5.0000000
      2.0000000      -3.0000000      -10.000000       2.0000000      -3.0000000      -10.000000
      6.0000000       25.000000       12.000000       6.0000000       25.000000       12.000000
x|x =
      1.0000000       2.3000000       100.00000
      2.0000000       3.0000000       5.0000000
      2.0000000      -3.0000000      -10.000000
      6.0000000       25.000000       12.000000
      1.0000000       2.3000000       100.00000
      2.0000000       3.0000000       5.0000000
      2.0000000      -3.0000000      -10.000000
      6.0000000       25.000000       12.000000
```

Pour déclarer la matrice (1), nous pouvons donc employer ces deux opérateurs. Nous avons (~ est prioritaire sur |) :

```
x  =  1~2.3~100.|2~3~5|2~-3~-10|6~25~12;

x  =  1 ~  2.3 ~  100. |
      2 ~  3   ~  5    |
      2 ~ -3   ~ -10   |
      6 ~  25  ~  12   ;
```

GAUSS possède une matrice spéciale, la matrice nulle (ou la matrice vide). Celle-ci est initialisée par la syntaxe `{}`.

```
new;

output file = ccf6.out reset;

x = {};

print "x = ";
print x;

x = x~x;

print "x~x = ";
print x;

x = x|2.5|3.0|x;

print "x|2.5|3.0|x = ";
```

```
print x;

output off;
```

Le fichier *ccf6.out* se présente de la façon suivante :

```
x =
              {}
x~x =
              {}
x|2.5|3.0|x =

       2.5000000
       3.0000000
```

2.3 Les commandes `ones`, `zeros` et `eye`

Pour créer une matrice $(r \times c)$ de 1 (respectivement de 0), nous employons la syntaxe `x = ones(r,c);` (respectivement `x = zeros(r,c);`). La commande `eye` permet d'obtenir la matrice identité.

```
new;

output file = ccf7.out reset;

x = ones(4,2);

print "ones(4,2) = ";
print x;

y = zeros(3,2);

print "zeros(3,2) = ";
print y;

z = eye(3);

print "eye(3) = ";
print z;

w = rndu(5,2);

print "rndu(5,2) = ";
print w;

output off;
```

`rndu` permet d'obtenir des nombres au hasard de la loi uniforme $\mathcal{U}_{[0,1]}$. L'objet manipulé par GAUSS étant une matrice, nous obtenons une matrice de nombres au hasard.

```
ones(4,2) =

       1.0000000        1.0000000
```

```
       1.0000000        1.0000000
       1.0000000        1.0000000
       1.0000000        1.0000000
zeros(3,2) =

       0.0000000        0.0000000
       0.0000000        0.0000000
       0.0000000        0.0000000
eye(3) =

       1.0000000        0.0000000        0.0000000
       0.0000000        1.0000000        0.0000000
       0.0000000        0.0000000        1.0000000
rndu(5,2) =

      0.89861232       0.57215262
      0.60464747       0.95124463
      0.13038415      0.031029955
      0.99068608      0.084516632
      0.40434782       0.14087541
```

2.4 Les sous-matrices

Pour accéder à un élément d'une matrice, nous utilisons les crochets `[]`. `x[r,c]` est l'élément $X_{r,c}$ (c-à-d. l'élément de la ligne r et de la colonne c). En fait, les crochets permettent de définir une sous-matrice. Pour GAUSS, un élément est une sous-matrice de dimension (1×1). `r` et `c` sont donc des vecteurs. La notation `.` permet de sélectionner l'ensemble des lignes `x[.,c]` ou des colonnes `x[r,.]`. Pour sélectionner une suite continue de lignes ou de colonnes, nous utilisons la notation `:`.

```
new;

output file = ccf8.out reset;

x = rndu(9,4);

x12 = x[1,2];     /* element (1,2) de la matrice x */
x22 = x[2,2];     /* element (2,2) de la matrice x */

print "x(1,2) = " x12;
print "x(2,2) = " x22;

x2 = x[1 2,2];
print "x[1 2,2] = " x2;

y = x[1 2 5,2 4];
print "x[1 2 5,2 4] = " y;

y = x[1 2,.];
print "x[1 2,.] = " y;

output off;
```

```
x(1,2) =     0.0023641288
x(2,2) =       0.24587406
x[1 2,2] =
    0.0023641288
      0.24587406
x[1 2 5,2 4] =
    0.0023641288       0.56297318
      0.24587406       0.31672191
      0.35802595      0.061291279
x[1 2,.] =
      0.59192991     0.0023641288       0.64968164       0.56297318
      0.21466559       0.24587406       0.93243942       0.31672191
```

Nous pouvons employer `x[r,c]` comme une *lvalue* ou une *rvalue*. Cela permet de modifier les éléments d'une matrice.

```
new;

output file = ccf9.out reset;

@ Une nouvelle forme de commentaire @

x  =  {1 2.3 100. , 2 3 5 , 2 -3 -10 , 6 25 12};

x[2,2] = 0;
x[.,3] = ones(4,1);
x[1,1:2] = 10^5;

print x;

output off;
```

```
       10.000000        5.0000000        1.0000000
       2.0000000        0.0000000        1.0000000
       2.0000000       -3.0000000        1.0000000
       6.0000000        25.000000        1.0000000
```

Remarque : La commande `submat` permet aussi d'extraire une sous-matrice, mais elle est peu employée.

2.5 Chargement de données ascii

Il existe des commandes spéciales pour manipuler une base de données. Ces commandes sont construites, comme dans tous les langages statistiques, à partir des tableaux multidimensionnels. **Dans GAUSS, nous pouvons considérer une base de données comme une matrice ; les lignes correspondent aux observations et les colonnes représentent les variables.** Les modules GaussX et Markov adoptent une syntaxe de type SAS. Par exemple, pour obtenir la régression de la variable y sur les variables x1 et x2, nous avons

```
OLS y c x1 x2;
```

Lorsque GaussX interprète cette ligne de commande, il forme le vecteur des endogènes à partir de y et la matrice des régresseurs à partir de c, x1 et x2. La programmation en langage GAUSS

oblige l'utilisateur à raisonner avec des matrices et non avec d'autres objets (variable, observation, coupe transversale, etc). Cela implique que l'utilisateur doit avoir de bonnes connaissances en calcul matriciel et en analyse numérique.

Voyons un exemple. Considérons la base de données suivante

x_1	x_2	x_3	x_4	x_5	y_1	y_2	y_3
1	3.06	1.34	8.48	28	359.27	102.96	578.49
1	3.19	1.44	9.16	35	415.76	114.38	650.86
1	3.3	1.54	9.9	37	435.11	118.23	684.87
1	3.4	1.71	11.02	36	440.17	120.45	680.47
1	3.48	1.89	11.64	29	410.66	116.25	642.19
1	3.6	1.99	12.73	47	530.33	140.27	787.41
1	3.68	2.22	13.88	50	557.15	143.84	818.06
1	3.72	2.43	14.5	35	472.8	128.2	712.16
1	3.92	2.43	15.47	33	471.76	126.65	722.23
1	4.15	2.31	16.61	40	538.3	141.05	811.44
1	4.35	2.39	17.4	38	547.76	143.71	816.36
1	4.37	2.63	18.83	37	539	142.37	807.78
1	4.59	2.69	20.62	56	677.6	173.13	983.53
1	5.23	3.35	23.76	88	943.85	223.21	1292.99
1	6.04	5.81	26.52	62	893.42	198.64	1179.64
1	6.36	6.38	27.45	51	871	191.89	1134.78
1	7.04	6.14	30.28	29	793.93	181.27	1053.16
1	7.81	6.14	25.4	22	850.36	180.56	1085.91
1	8.09	6.19	28.84	38	967.42	208.24	1246.99
1	9.24	6.69	34.36	41	1102.61	235.43	1401.94

Pour charger cette base de données, construisons d'abord un fichier ascii *base1.asc* contenant les données

```
1  3.06  1.34  8.48   28  359.27   102.96  578.49
1  3.19  1.44  9.16   35  415.76   114.38  650.86
1  3.3   1.54  9.9    37  435.11   118.23  684.87
1  3.4   1.71  11.02  36  440.17   120.45  680.47
1  3.48  1.89  11.64  29  410.66   116.25  642.19
1  3.6   1.99  12.73  47  530.33   140.27  787.41
1  3.68  2.22  13.88  50  557.15   143.84  818.06
1  3.72  2.43  14.5   35  472.8    128.2   712.16
1  3.92  2.43  15.47  33  471.76   126.65  722.23
1  4.15  2.31  16.61  40  538.3    141.05  811.44
1  4.35  2.39  17.4   38  547.76   143.71  816.36
1  4.37  2.63  18.83  37  539      142.37  807.78
1  4.59  2.69  20.62  56  677.6    173.13  983.53
1  5.23  3.35  23.76  88  943.85   223.21  1292.99
1  6.04  5.81  26.52  62  893.42   198.64  1179.64
1  6.36  6.38  27.45  51  871      191.89  1134.78
1  7.04  6.14  30.28  29  793.93   181.27  1053.16
1  7.81  6.14  25.4   22  850.36   180.56  1085.91
1  8.09  6.19  28.84  38  967.42   208.24  1246.99
1  9.24  6.69  34.36  41  1102.61  235.43  1401.94
```

Nous pouvons alors charger cette base de données dans une matrice avec l'instruction `load` en spécifiant les nombres de lignes et de colonnes. `data` est donc une matrice de dimension (20×8). La variable `x1235` correspond aux variables x_1, x_2, x_3 et x_5. C'est une sous-matrice de `data`. Les opérateurs `meanc` et `stdc` permettent de calculer la moyenne et l'écart-type. **Ce sont des opérateurs matriciels**. Cela veut dire que `meanc` calcule la moyenne de chaque colonne de la matrice et renvoie un vecteur contenant ces moyennes. La moyenne et l'écart-type de la variable x_3 sont 3.385 et 1.9676.

```
new;

output file = ccf10.out reset;

load data[20,8] = base1.asc;

x = data[.,1 2 3 4 5];  /* x = data[.,1:5]   */
y = data[.,6:8];        /* y = data[.,6 7 8] */
y1 = y[.,1];            /* y = data[.,6]     */

x1235 = x[.,1:3 5];

m = meanc(x1235);
s = stdc(x1235);

print "       Moyenne          Ecart-type";
print m~s;

output off;
```

```
       Moyenne          Ecart-type

       1.0000000        0.0000000
       4.9310000        1.8565613
       3.3855000        1.9676154
       41.600000        14.716264
```

3 Les principales commandes de GAUSS

La version 3.2.26 de GAUSS contient 556 commandes repertoriées dans l'aide (sans compter les opérateurs) ! La pratique montre qu'il suffit d'en connaître moins de 50 pour programmer, car beaucoup de ces commandes sont très spécialisées et sont donc peu employées. Voici une liste des commandes indispensables à connaitre : `abs`, `break`, `call`, `cdfn`, `ceil`, `chol`, `cols`, `conj`, `corrx`, `counts`, `countwts`, `cumsumc`, `delif`, `det`, `do until`, `do while`, `eigh`, `else`, `elseif`, `endp`, `exp`, `eye`, `fft`, `ffti`, `floor`, `ftos`, `gamma`, `gradp`, `hessp`, `hsec`, `if`, `indexcat`, `inv`, `invpd`, `lagn`, `ln`, `load`, `maxc`, `maxindc`, `minc`, `minindc`, `miss`, `output`, `packr`, `pinv`, `proc`, `qqr`, `real`, `recserar`, `reshape`, `retp`, `return`, `rev`, `rotater`, `rows`, `saved`, `selif`, `seqa`, `shiftr`, `sortc`, `sqrt`, `stdc`, `svd`, `trimr`, `vec`, `vech` et `xpnd`.

3.1 Les opérateurs matriciels

`.'`	`B = A.';`	opérateur de transposition des éléments d'une matrice
`'`	`B = A';`	opérateur de transposition matricielle ($A^\top$ pour une matrice réelle et A^* pour une matrice complexe)
`!`	`B = A!;`	opérateur factoriel
`.^`	`C = B.^A;`	opérateur exposant
`*`	`C = A*B;`	produit matriciel
`*~`	`C = A*~B;`	produit direct horizontal
`.*`	`C = A.*B;`	produit d'Hadamard $C = A \odot B$
`.*.`	`C = A.*.B;`	produit de Kronecker $C = A \otimes B$
`./`	`C = A./B;`	division élément par élément
`/`	`x = b/A;`	solution d'un système équations linéaires $Ax = b$
`+`	`C = A+B;`	addition matricielle
`-`	`C = A-B;`	soustraction matricielle
`|`	`C = A|B;`	concaténation verticale
`~`	`C = A~B;`	concaténation horizontale

3.2 Les fonctions

ABS	`y = abs(x);`	retourne la valeur absolue ou le module complexe de x
COS	`y = cos(x);`	retourne le cosinus de x
EXP	`y = exp(x);`	retourne l'exponentielle de x
GAMMA	`y = gamma(x);`	retourne la valeur de $\Gamma(x) = \int_0^\infty t^{x-1}e^{-t}dt$
LN	`y = ln(x);`	retourne le logarithme de x
SIN	`y = sin(x);`	retourne le sinus de x
SQRT	`y = sqrt(x);`	retourne la racine carrée de x
TAN	`y = tan(x);`	retourne la tangente de x

3.3 Les commandes mathématiques

CHOL	`P = chol(A);`	Calcule la décomposition de Cholesky $A = P^\top P$ avec P une matrice triangulaire supérieure
DET	`d = det(A);`	Calcule le déterminant d'une matrice carrée
DFFT	`y = dfft(x);`	Calcule la transformée discrète de Fourier
DFFTI	`x = dffti(x);`	Calcule la transformée discrète inverse de Fourier
EIGHV	`{va,ve} = eighv(M);`	Calcule les valeurs propres et les vecteurs propres d'une matrice hermitienne
EIGV	`{va,ve} = eigv(M);`	Calcule les valeurs propres et les vecteurs propres d'une matrice carrée
FFT	`y = fft(x);`	Calcule la transformée rapide de Fourier
FFTI	`x = ffti(x);`	Calcule la transformée rapide inverse de Fourier
GRADP	`y = gradp(&f,x);`	Calcule le Jacobien de la fonction f en x
HESSP	`y = hessp(&f,x);`	Calcule la matrice Hessienne de la fonction f en x
INV	`B = inv(A);`	Calcule l'inverse d'une matrice carrée $B = A^{-1}$
INVPD	`B = invpd(A);`	Calcule l'inverse d'une matrice p.d.s. $B = A^{-1}$
PINV	`B = pinv(A);`	Calcule l'inverse généralisée de Moore-Penrose $B = A^+$
POLYCHAR	`c = polychar(A);`	Calcule le polynôme caractéristique de la matrice carrée A
POLYMULT	`c = polymult(c1,c2);`	multiplication polynômiale
POLYROOT	`r = polyroot(c);`	racines d'un polynôme

3.4 Les commandes statistiques

CDFCHIC	`y = cdfchic(x,n);`	fonction de répartition complémentaire de la loi chi-deux à n degrés de liberté
CDFN	`y = cdfn(x);`	fonction de répartition de la loi normale centrée et réduite
CDFNC	`y = cdfnc(x);`	fonction de répartition complémentaire de la loi normale centrée et réduite
CDFTC	`y = cdftc(x,n);`	fonction de répartition complémentaire de la loi de Student à n degrés de liberté
CONV	`c = conv(a,b,0,0);`	convolution des vecteurs a et b
CORRX	`C = corrx(x);`	matrice de corrélation des données x
MEANC	`M = meanc(x);`	moyennes des données x
MEDIAN	`M = median(x);`	médianes des données x
RNDN	`u = rndn(k,l);`	matrice $(k \times l)$ de nombres aléatoires gaussiens
RNDSEED	`rndseed s;`	initialise le générateur des nombres aléatoires
RNDU	`u = rndu(k,l);`	matrice $(k \times l)$ de nombres aléatoires uniformes
STDC	`sigma = stdc(x);`	écarts-types des données s
VCX	`V = vcx(x);`	matrice de variance-covariance des données x

3.5 Les commandes de séquence

RECSERAR	`y = recserar(x,y0,a);`	séquence autorégressive ARX
SEQA	`y = seqa(y0,h,n);`	séquence additive $y_n = y_0 + nh$

3.6 Les commandes de manipulation de matrices

COLS	`n = cols(A);`	nombre de colonnes de la matrice A
CUMSUMC	`y = cumsumc(x);`	sommes cumulées des colonnes d'une matrice
COMPLEX	`z = complex(x,y);`	création d'une matrice complexe $z = x + iy$
DIAG	`d = diag(A);`	diagonale de la matrice A
EYE	`I = eye(n);`	matrice identité
IMAG	`y = imag(z);`	partie imaginaire d'une matrice complexe
MAXC	`m = maxc(x);`	vecteur des valeurs maximales de chaque colonne de la matrice x
MAXINDC	`pos = maxindc(x);`	vecteur des positions des valeurs maximales
MINC	`m = minc(x);`	vecteur des valeurs minimales de chaque colonne de la matrice x
MININDC	`pos = minindc(x);`	vecteur des positions des valeurs minimales
MISS	`y = miss(x,v);`	remplace dans la matrice x la valeur v par une donnée manquante
MISSRV	`y = missrv(x,v);`	remplace dans la matrice x les données manquantes par la valeur v
ONES	`A = ones(m,n);`	création de la matrice $\mathbf{1}_{m\times n}$
PACKR	`y = packr(x);`	élimination des lignes de la matrice x contenant des données manquantes
REAL	`x= real(z);`	partie réelle d'une matrice complexe
REV	`y = rev(x);`	renversement des lignes de la matrice
ROWS	`m = rows(A);`	nombre de lignes de la matrice A
SUMC	`s = sumc(x);`	sommes cumulées des colonnes d'une matrice
VEC	`v = vec(x);`	opérateur *vec*
VECH	`v = vech(x);`	opérateur *vech*
TRIMR	`y = trimr(x,a,b);`	élimination des a premières et b dernières lignes
ZEROS	`A = zeros(m,n);`	création de la matrice $\mathbf{0}_{m\times n}$

4 Exemples d'utilisation des commandes de GAUSS

4.1 Les opérateurs matriciels

GAUSS possède de nombreux opérateurs matriciels. Les plus importants sont l'addition `+`, la soustraction `-`, le produit matriciel `*`, l'opérateur transposée `'` et le produit de kronecker `.*.`.

```
new;

output file = ccf11.out reset;

rndseed 123;        @ initialisation du compteur de nombres aleatoires @

x = rndn(3,3);      /* Nombres aleatoires N(0,1) */

y = x';
xx = x*x;
xtx = x'x;          /* xtx = x'*x; */
z = x + x;
w = x - (x'x);
```

```
print "x = " x;
print "x' = " y;
print "x*x = " xx;
print "x'x = " xtx;
print "x + x = " z;
print "x - x'x = " w;

output off;

x =
      0.37608074       0.23892314      -0.83087951
      0.63743639       -1.4876279       0.69592528
      -2.0496829       0.35205503       0.25788499
x' =
      0.37608074       0.63743639       -2.0496829
      0.23892314       -1.4876279       0.35205503
     -0.83087951       0.69592528       0.25788499
x*x =
       1.9967745      -0.55808966      -0.36047648
      -2.1349668        2.6103391       -1.3854420
      -1.0750160      -0.92265386        2.0145482
x'x =
       4.7489617       -1.5800150      -0.39745212
      -1.5800150        2.3940639       -1.1430045
     -0.39745212       -1.1430045        1.2411774
x + x =
      0.75216148       0.47784628       -1.6617590
       1.2748728       -2.9752558        1.3918506
      -4.0993657       0.70411007       0.51576997
x - x'x =
      -4.3728810        1.8189381      -0.43342738
       2.2174513       -3.8816918        1.8389298
      -1.6522308        1.4950596      -0.98329243
```

4.2 Les opérateurs matriciels $E \times E$

GAUSS possède des opérateurs matriciels particuliers, les opérateurs $E \times E$. Nous les appelons aussi les opérateurs élément par élement. Ces opérateurs sont les suivants : `+`, `-`, `.^`, `.*` et `./`. Considérons l'opérations $z = x$ **op** y avec **op** un opérateur $E \times E$. Dans l'analyse classique, il n'est pas possible d'additionner deux matrices de dimensions différentes. L'opération est dite non conforme. `+` étant un opérateur $E \times E$ dans GAUSS, nous pouvons dans certains cas additionner deux matrices n'ayant pas les mêmes dimensions. Dans ce cas, l'opération est faite élément par élément. Nous avons

$$\begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix} + \begin{pmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{pmatrix} = \begin{pmatrix} 2 & 5 \\ 4 & 7 \\ 6 & 9 \end{pmatrix} \tag{3}$$

et

$$\begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix} + \begin{pmatrix} 1 & 2 & 3 & 4 \end{pmatrix} = \begin{pmatrix} 2 & 3 & 4 & 5 \\ 3 & 4 & 5 & 6 \\ 4 & 5 & 6 & 7 \end{pmatrix} \tag{4}$$

Mais cette opération n'est pas autorisée

$$\begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix} + \begin{pmatrix} 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \end{pmatrix} \tag{5}$$

Le tableau suivant indique les cas où l'opérations $z = x$ **op** y est autorisée.

x	y	z
Matrice $(r \times c)$	Matrice $(r \times c)$	Matrice $(r \times c)$
Matrice $(r \times c)$	Vecteur $(r \times 1)$	Matrice $(r \times c)$
Matrice $(r \times c)$	Vecteur $(1 \times c)$	Matrice $(r \times c)$
Vecteur $(r \times 1)$	Matrice $(r \times c)$	Matrice $(r \times c)$
Vecteur $(1 \times c)$	Matrice $(r \times c)$	Matrice $(r \times c)$
Vecteur $(r \times 1)$	Vecteur $(r \times 1)$	Vecteur $(r \times 1)$
Vecteur $(1 \times c)$	Vecteur $(1 \times c)$	Vecteur $(1 \times c)$
Vecteur $(1 \times c)$	Vecteur $(r \times 1)$	Matrice $(r \times c)$
Vecteur $(r \times 1)$	Vecteur $(1 \times c)$	Matrice $(r \times c)$
scalaire	Matrice $(r \times c)$	Matrice $(r \times c)$
Matrice $(r \times c)$	scalaire	Matrice $(r \times c)$

Dans ce premier exemple, nous considérons un vecteur ligne et un vecteur colonne.

```
new;

output file = ccf12.out reset;

u = seqa(1,5,3);
v = 2*seqa(1,1,4)';

x = u + v;
y = u .* v;
z = u ./ v;

print "u = " u;
print "v = "; print v;
print "u + v = " x;
print "u .* v = " y;
print "u ./ v = " z;

output off;

u =
       1.0000000
       6.0000000
       11.000000
v =
       2.0000000       4.0000000       6.0000000       8.0000000
u + v =
       3.0000000       5.0000000       7.0000000       9.0000000
       8.0000000       10.000000       12.000000       14.000000
```

```
       13.000000        15.000000        17.000000        19.000000
u .* v =
       2.0000000        4.0000000        6.0000000        8.0000000
       12.000000        24.000000        36.000000        48.000000
       22.000000        44.000000        66.000000        88.000000
u ./ v =
      0.50000000       0.25000000       0.16666667       0.12500000
       3.0000000        1.5000000        1.0000000       0.75000000
       5.5000000        2.7500000        1.8333333        1.3750000
```

Nous considérons un vecteur colonne et une matrice dans l'exemple suivant :

```
new;

output file = ccf13.out reset;

u = seqa(1,5,3);
v = rndu(3,4);

x = u + v;
y = u .* v;
z = u ./ v;

print "u = " u;
print "v = "; print v;
print "u + v = " x;
print "u .* v = " y;
print "u ./ v = " z;

output off;

u =
       1.0000000
       6.0000000
       11.000000
v =

      0.23393318       0.79361543       0.68520876       0.25894610
      0.85714781       0.47327842       0.68653820       0.70254734
      0.54616210      0.075421128       0.62735179       0.74703324
u + v =
       1.2339332        1.7936154        1.6852088        1.2589461
       6.8571478        6.4732784        6.6865382        6.7025473
       11.546162        11.075421        11.627352        11.747033
u .* v =
      0.23393318       0.79361543       0.68520876       0.25894610
       5.1428869        2.8396705        4.1192292        4.2152841
       6.0077832       0.82963240        6.9008697        8.2173657
u ./ v =
       4.2747250        1.2600561        1.4594093        3.8618075
       6.9999595        12.677527        8.7394992        8.5403497
       20.140541        145.84773        17.534022        14.724914
```

Ces opérateurs $E \times E$ sont très rapides. Ils permettent dans de nombreux cas d'éviter les boucles (supposées être *time consuming*). Considérons par exemple une série temporelle x_t. Soit la série temporelle y_t définie de la façon suivante

$$y_t = x_t x_{t-1} x_{t-2} x_{t-3} \tag{6}$$

Pour construire cette série, nous utilisons l'opérateur `.*` (la commande `lagn(x,n)` permet de définir x_{t-n}).

```
new;

u = rndn(1000,1);       /* Simulation de 1000 nombres aleatoires N(0,1)  */
x = 10 + 2*u;           /* Simulation de 1000 nombres aleatoires N(10,2) */

y = x.*lag1(x).*lagn(x,2).*lagn(x,3);
```

Nous cherchons à calculer les coefficients de corrélation entre la série x_t et les séries $y_t^{(i)}$ en utilisant les opérateurs matriciels de GAUSS sans employer la commande `corrx`. La formule du coefficient de corrélation est

$$r = \frac{\frac{1}{N}\sum_{t=1}^{N}(x_t - \bar{x})(y_t - \bar{y})}{\sqrt{\frac{1}{N}\sum_{t=1}^{N}(x_t - \bar{x})^2}\sqrt{\frac{1}{N}\sum_{t=1}^{N}(y_t - \bar{y})^2}} \tag{7}$$

Il n'est pas nécessaire d'utiliser une boucle pour calculer les différents coefficients. Les variables `yc` et `xc` sont les variables centrées. L'opérateur transposée dans la ligne de commande `yc = y - meanc(y)'` est nécessaire pour soustraire la moyenne de chaque variable à la colonne correspondante. `xc .* xc` correspond à un vecteur dont les composantes sont $(x_t - \bar{x})^2$. Les éléments de la matrice `xy` de dimension (1000×10) sont $(x_t - \bar{x})\left(y_t^{(i)} - \bar{y}^{(i)}\right)$.

```
new;

output file = ccf15.out reset;

rndseed 123456;

u = rndn(1000,10);

/*
**  Simulation d'une marche aleatoire
**
**  y(t) = y(t-1) + u(t)
**  y(t0) = 10;
**  u(t) = N(0,1)
*/

y = recserar(u,10*ones(1,10),ones(1,10));  /* 10 series de 1000 observations */

x = recserar(rndn(1000,1),10,1);           /* 1 serie de 1000 observations   */

yc = y - meanc(y)';
xc = x - meanc(x);
```

```
xy = xc .* yc;

covXY = meanc(xy);

sigmaX = sqrt( meanc( xc .* xc ) );
sigmaY = sqrt(
               meanc(yc.*yc)
             );

corrXY = covXY./(sigmaX.*sigmaY);

print corrXY;

output off;
```

```
      0.29278330
      0.57043840
      0.13468502
     -0.12381810
      0.56863515
      0.12194396
    -0.041144659
      0.13892588
      0.32042815
      0.40755735
```

4.3 Les fonctions

Dans le programme suivant, nous utilisons quelques fonctions de GAUSS. Nous remarquons en particulier qu'elles sont définies sur le domaine des complexes $\mathbb{C}$. A titre d'exemple, essayez les commandes suivantes

```
i = sqrt(-1);
j = sqrt(i);
k = ln(j);
l = cos(k);
```

```
new;

output file = ccf16.out reset;

rndseed 123456;

u = rndn(3,2);

x = abs(u);
y = sqrt(u);
z = cos(u);
w = ln(u);

print "u = " u;
print "abs(u) = " x;
```

```
print "sqrt(u) = " y;
print "cos(u) = " z;
print "ln(u) = " w;

y2 = y^2;

print "sqrt(u)^2 = " y2;
print real(y2);

output off;

u =
      0.24023662       0.26977652
      -1.0624093      -0.94190738
      -1.1962536        1.8982432
abs(u) =
      0.24023662       0.26977652
       1.0624093       0.94190738
       1.1962536        1.8982432
sqrt(u) =
      0.49013939                          0.51940015
       0.0000000 +        1.0307324i        0.0000000 +       0.97051913i
       0.0000000 +        1.0937338i        1.3777675
cos(u) =
      0.97128170       0.96383048
      0.48676887       0.58824663
      0.36584695      -0.32162660
ln(u) =
      -1.4261309                          -1.3101614
     0.060539289 +        3.1415927i     -0.059848330 +        3.1415927i
      0.17919471 +        3.1415927i       0.64092882
sqrt(u)^2 =
      0.24023662                          0.26977652
      -1.0624093 -    5.7593326e-20i      -0.94190738 -    5.1060901e-20i
      -1.1962536 -    6.4849040e-20i        1.8982432

      0.24023662       0.26977652
      -1.0624093      -0.94190738
      -1.1962536        1.8982432
```

4.4 Les commandes mathématiques

Dans l'exemple suivant, nous utilisons une matrice X réelle. Mais les commandes que nous employons sont aussi valides pour une matrice X complexe. Il existe plusieurs procédures d'inversion de matrice. En particulier, la procédure `invpd` doit être privilégiée pour les matrices symétriques définies positives. Nous pouvons aussi utiliser la procédure générale `inv`, mais les temps de calcul sont plus longs.

```
new;

output file = ccf17.out reset;

rndseed 123;
```

```
X = rndu(3,3);

invX = inv(x);    /* Gauss est case-insensitive     */

xx = x'x;

invXX = invpd(XX);

y = invXX*XX;     /* Gauss est un langage numerique */

det1 = det(XX);
det2 = det(invXX);

print "X = " X;
print "X^(-1) = " INVx;
print "(X'X)^(-1) = " INVxx;
print "y = " y;
print;
print "det(X'X) = " det1;
print "det((X'X)^(-1)) = " det2;
print "det1*det2 = " (det1*det2);

output off;
```

```
X =
      0.75039609       0.32091203       0.17838965
      0.90603338       0.35711708       0.22111402
      0.78643830       0.39808191       0.12466522
X^(-1) =
      -37.710106        26.879172        6.2866844
       52.828567       -40.521148       -3.7241800
       69.197925       -40.172183       -19.745368
(X'X)^(-1) =
       2184.0644       -3104.7585       -3813.3890
      -3104.7585        4446.6905        5356.9855
      -3813.3890        5356.9855        6792.0367
y =
       1.0000000    1.1296519e-13    5.5372373e-14
   1.6786572e-13       1.0000000    3.6137759e-14
  -9.1926466e-14   -3.9190873e-14       1.0000000

det(X'X) =    1.3307337e-06
det((X'X)^(-1)) =        751465.14
det1*det2 =       1.0000000
```

La décomposition de Cholesky d'une matrice p.d.s X correspond à la commande `chol`. Cette procédure renvoie la matrice triangulaire supérieure Q telle que $X = Q^\top Q$. La matrice triangulaire inférieure $\mathbb{P}$ correspond à $Q^\top$. Considérons la loi gaussienne multidimensionnelle $\mathcal{N}(\mu, \Sigma)$. Nous avons $\mathcal{N}(\mu, \Sigma) = \mu + \mathbb{P}\mathcal{N}(\mathbf{0}, \mathbf{I})$. Pour simuler des vecteurs aléatoires issus de la loi $\mathcal{N}(\mu, \Sigma)$, nous utilisons donc les commandes `rndn` (pour simuler des vecteurs issus de la loi $\mathcal{N}(\mathbf{0}, \mathbf{I})$) et `chol` (pour obtenir la décomposition de cholesky de la matrice de covariance Σ).

```
new;
```

```
output file = ccf18.out reset;

rndseed 123;

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

P = chol(SIGMA)';

print "P = " P;
print "P*P' = " P*P';

u = mu + P*rndn(3,1);     /* Simulation d'un vecteur aleatoire N(mu,SIGMA) */

print "u = " u;

u = mu + P*rndn(3,1000);  /* Simulation de 1000 vecteurs aleatoires */

@
  Attention, u est de dimension 3*1000. Il faut transposer
  cette matrice pour que les colonnes correspondent aux series
@

u = u';

moy = meanc(u);
COV = vcx(u);

print "Moyenne empirique = " moy;
print "Matrice de covariance empirique = " COV;

output off;
```

```
P =
       1.4832397        0.0000000        0.0000000
      0.33709993        1.0422877        0.0000000
       0.0000000      -0.57565680        1.2118660
P*P' =
       2.2000000       0.50000000        0.0000000
      0.50000000        1.2000000      -0.60000000
       0.0000000      -0.60000000        1.8000000
u =
       10.557818
       12.375803
      0.85554763
Moyenne empirique =
       10.071559
       12.011857
       1.9695983
Matrice de covariance empirique =
       2.3349252       0.57306541     -0.082588386
```

```
     0.57306541        1.1510044      -0.58354702
    -0.082588386      -0.58354702       1.6826909
```

Nous remarquons que la moyenne et la covariance empiriques des 1000 vecteurs aléatoires sont proches des valeurs théoriques. Pour mettre en évidence la convergence asymptotique vers la moyenne théorique, nous pouvons remplacer la ligne de commande `moy = meanc(u);` par `moy = cumsumc(u)./seqa(1,1,1000);`.

4.5 Les commandes statistiques

4.6 Les commandes de séquence

4.7 Les commandes de manipulation de matrices

Pour connaître le nombre de colonnes et de lignes d'une matrice, nous utilisons les commandes `cols` et `rows`. Nous introduisons dans l'exemple suivant la commande `ftos` qui permet de convertir un nombre (*float*) en chaîne de caractères (*string*).

```
new;

output file = ccf19.out reset;

let A[4,3] = 1  2  3
             4  5  6
             7  8  9
             10 11 12;

n = cols(A);        /* Nombre de colonnes de la matrice A */
m = rows(A);        /* Nombre de lignes de la matrice A   */

print ftos(m,"m = %lf",2,0);
print ftos(n,"n = %lf",2,0); print;  /* Plusieurs instructions par lignes */


d = diag(A);        /* Diagonale de A */

print "diag(A) = ";
call printfmt(d,1);

s = sumc(A);

print "sumc(A) = ";
call printfmt(s,1);

cs = cumsumc(A);

print "cumsumc(A) = ";
call printfmt(cs,1);

max = maxc(A);

print "maxc(A) = ";
call printfmt(max,1);
```

```
indx = maxindc(A);

print "maxindc(A) = ";
call printfmt(indx,1);

output off;

m =  4
n =  3

diag(A) =
               1
               5
               9
sumc(A) =
              22
              26
              30
cumsumc(A) =
               1               2               3
               5               7               9
              12              15              18
              22              26              30
maxc(A) =
              10
              11
              12
maxindc(A) =
               4
               4
               4
```

Les trois opérateurs `vec`, `vecr` et `trimr` sont peu employés par les utilisateurs de GAUSS, mais ils sont indispensables pour une programmation efficace. `trimr` permet d'éliminer les premières et dernières lignes. Ainsi `trimr(x,2,3)` définit une nouvelle matrice sans les deux premières et les trois dernières lignes. Il est préférable d'utiliser cette commande plutôt que l'opérateur de sélection : (par exemple, si le nombres de lignes de `x` est 1000, `trimr(x,2,3)` est préférable à `x[3:997,.]`). La commande `vec` correspond à l'opérateur matriciel vec (A) (`vecr` correspond à vec $\left(A^{\top}\right)$)[1].

```
new;

output file = ccf20.out reset;

let A[4,3] = 1  2  3
             4  5  6
             7  8  9
             10 11 12;
```

[1] `vecr` est plus rapide que `vec`.

```
v = vec(A);
vr = vecr(A);

print "            vec(A)           vecr(A)";
call printfmt(v~vr,1);

B = trimr(A,0,2);

print "trimr(A,0,2) = ";
call printfmt(B,1);

output off;
```

```
            vec(A)           vecr(A)
               1                1
               4                2
               7                3
              10                4
               2                5
               5                6
               8                7
              11                8
               3                9
               6               10
               9               11
              12               12
trimr(A,0,2) =
               1                2                3
               4                5                6
```

La commande `reshape` permet de définir une matrice à partir des éléments d'une autre matrice en modifiant les nombres de lignes et de colonnes.

```
new;

output file = ccf21.out reset;

let A[3,3] = 1  2  3
             4  5  6
             7  8  9;


B = reshape(A,2,4);

print "reshape(A,2,4) = ";
call printfmt(B,1);

C = reshape(A,6,4);

print "reshape(A,6,4) = ";
call printfmt(C,1);

output off;
```

```
reshape(A,2,4) =
               1               2               3               4
               5               6               7               8
reshape(A,6,4) =
               1               2               3               4
               5               6               7               8
               9               1               2               3
               4               5               6               7
               8               9               1               2
               3               4               5               6
```

Considérons une série temporelle journalière. Nous désirons construire les séries hebdomadaires pour chaque jour de la semaine.

```
new;

output file = ccf22.out reset;

x = rndn(7*100,1);

y = reshape(x,100,7);

call printfmt(x[8:14],1);
print;
call printfmt(y[2,.],1);

output off;

     -0.42214705
      -1.4373913
      0.69198098
     -0.83539713
     -0.71647858
     -0.60252403
     -0.71657524

     -0.42214705       -1.4373913       0.69198098      -0.83539713
     -0.71647858      -0.60252403      -0.71657524
```

4.8 Les nombres particuliers de GAUSS

Certains nombres particuliers sont définis dans GAUSS, par exemple $+\infty$ et $-\infty$. Certaines opérations sont autorisées dans certains cas. Par exemple, l'addition de $+\infty$ et $+\infty$ donne $+\infty$. Nous avons aussi

$$\begin{aligned} (+\infty) * (+\infty) &= +\infty \\ (-\infty) + (-\infty) &= -\infty \\ (+\infty) * (-\infty) &= -\infty \\ 0/(+\infty) &= 0 \\ 1/0 &= +\infty \\ -1/0 &= -\infty \end{aligned} \tag{8}$$

Dans d'autres cas, les opérations sont interdites, comme par exemple $0 * (+\infty)$. La valeur manquante (*missing value*) est un codage très particulier. Elle n'est pas un nombre et ressemble

plus à un caractère. Par défaut, elle correspond au symbole point (.). Nous pouvons modifier sa représentation avec la commande msym.

```
new;

Msym .;

output file = ccf23.out reset;

plusINFINI = __INFp;
moinsINFINI = __INFn;
plusINDEFINI = __INDEFp;
moinsINDEFINI = __INDEFn;
machineEPSILON = __macheps;

print plusINFINI;
print plusINDEFINI;
print moinsINFINI;
print moinsINDEFINI;
print machineEPSILON;

ValeurManquante = miss(0,0);

print ftos(ValeurManquante,"valeur manquante : %-*.*s",5,1);
print ftos(ValeurManquante,"valeur manquante : %lf",5,1);

Msym "NA";

print ftos(ValeurManquante,"valeur manquante : %lf",5,1);

Msym -9999.99;

print ftos(ValeurManquante,"valeur manquante : %lf",5,1);

output off;
```

```
            +INF
               .
            -INF
               .
   2.2300000e-16
valeur manquante : .
valeur manquante :     .
valeur manquante :    NA
valeur manquante : -9999.99
```

Plusieurs commandes concernent les valeurs manquantes : ismiss, miss, missex, missrv, packr et scalmiss.

```
new;

output file = ccf24.out reset;

let A[3,3] = 1  2  .
```

```
            4  5  6
            7  .  9;

print "A = " A;

B = missrv(A,0);
print "B = " B;

C = miss(A,0);
print "C = " C;

D = miss(C,1);
print "D = " D;

output off;

A =
       1.0000000        2.0000000                .
       4.0000000        5.0000000        6.0000000
       7.0000000                .        9.0000000
B =
       1.0000000        2.0000000        0.0000000
       4.0000000        5.0000000        6.0000000
       7.0000000        0.0000000        9.0000000
C =
       1.0000000        2.0000000                .
       4.0000000        5.0000000        6.0000000
       7.0000000                .        9.0000000
D =
               .        2.0000000                .
       4.0000000        5.0000000        6.0000000
       7.0000000                .        9.0000000
```

4.9 Un exemple complet : Les Moindres Carrés Ordinaires

```
new;

output file = ccf25.out reset;

load data[20,8] = base1.asc;

x = data[.,3 4];
y = data[.,6];

Nobs = rows(y);             /* Nombre d'observations */
constante = ones(Nobs,1);   /* La constante          */
x = constante~x;
k = cols(x);                /* Nombre d'exogenes     */

beta = invpd(x'x)*x'y;      /* coefficients estimes  */

u = y - x*beta;             /* residus                       */
SCR = u'u;                  /* Somme des carres des residus */
```

```
SCT = y'y;                  /* Somme des carres totaux      */
SCE = SCT - SCR;            /* Somme des carres expliques   */
R2 = SCE/SCT;               /* R2 de la regression          */

ddl = Nobs - k;             /* Degre de liberte   */
sigma = sqrt(SCR/ddl);      /* Ecart-type estime  */

Mcov = (sigma^2)*invpd(x'x);  /* Matrice de covariance des coefficients */
stderr = sqrt(diag(Mcov));    /* Ecart-type des coefficients            */

tstudent = beta ./ stderr;            /* Test T de student : beta[i] = 0 */
pvalue = 2*cdftc(abs(tstudent),ddl);  /* Risque de premiere espece       */

du = u - lag1(u);           /* u(t) - u(t-1)                               */
du = trimr(du,1,0);         /* La premiere donnee est une valeur manquante */

DW = sumc(du^2) / sumc(u^2);  /* statistique Durbin-Watson  */

Mcorr = corrvc(Mcov);
Mcorr2 = 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;
print Mcorr2;
```

```
Nombre d'observations :    20
Nombre de regresseurs :     3

Degres de liberte :        17

Ecart-type estime :     74.50
R2 :                    0.990
Durbin-Watson :         1.550

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

```
       130.99676        53.392137        2.4534842      0.025233564
       4.8879925        27.197684       0.17972091       0.85949647
       26.183783        6.8242584        3.8368687     0.0013207920

Matrice de covariance des estimateurs :

       2850.7203        809.76072       -282.05513
       809.76072        739.71401       -175.88231
      -282.05513       -175.88231        46.570503

Matrice de correlation des estimateurs :

       1.0000000       0.55763174      -0.77410744
      0.55763174        1.0000000      -0.94762123
     -0.77410744      -0.94762123        1.0000000

       1.0000000       0.55763174      -0.77410744
      0.55763174        1.0000000      -0.94762123
     -0.77410744      -0.94762123        1.0000000
```

5 Les opérateurs conditionnels et relationnels

5.1 Les opérateurs conditionnels

GAUSS possède cinq opérateurs logiques : `not`, `and`, `or`, `xor` and `eqv`. Il n'existe pas de type booléen TRUE et FALSE. FALSE correspond à l'**entier** nul 0. La représentation 4 bits de 0 doit être exacte. Cela veut dire que l'expression `ln(exp(sqrt(x)^2)) - x` ne correspond pas forcément à la variable FALSE. TRUE est représentée par n'importe quelle valeur différente de 0. Par exemple, `(not 1)`, `(not 2)` et `(not -3.56)` donne FALSE, c'est-à-dire 0. Mais, par convention, la variable TRUE générée par les commandes GAUSS est égale à 1. Par exemple, nous pouvons donner les valeurs 1 ou -10 à TRUE, mais `(not 0)` donne 1, c'est-à-dire TRUE.

```
new;

output file = ccf26.out reset;

TRUE = 1;
FALSE = 0;

print "not";

print (not TRUE);
print (not FALSE);

print "and";

print (TRUE and TRUE);
print (TRUE and FALSE);
print (FALSE and FALSE);
print (FALSE and TRUE);

print "or";
```

```
print (TRUE or TRUE);
print (TRUE or FALSE);
print (FALSE or FALSE);
print (FALSE or TRUE);

print "xor";

print (TRUE xor TRUE);
print (TRUE xor FALSE);
print (FALSE xor FALSE);
print (FALSE xor TRUE);

print "eqv";

print (TRUE eqv TRUE);
print (TRUE eqv FALSE);
print (FALSE eqv FALSE);
print (FALSE eqv TRUE);

output off;
```

```
not
       0.0000000
       1.0000000
and
       1.0000000
       0.0000000
       0.0000000
       0.0000000
or
       1.0000000
       1.0000000
       0.0000000
       1.0000000
xor
       0.0000000
       1.0000000
       0.0000000
       1.0000000
eqv
       1.0000000
       0.0000000
       1.0000000
       0.0000000
```

Il existe des opérateurs similaires $E \times E$. Dans ce cas, nous précédons l'opérateur d'un point (.).

```
new;

output file = ccf27.out reset;

let x[2,2] = 1 0
             2 0;
```

```
let y[2,2] = 1 5
             0 0;
z = .not x;
print z;

z = x .and y;
print z;

z = ( (.not x) .xor y ) .or y;
print z;

output off;

       0.0000000        1.0000000
       0.0000000        1.0000000

       1.0000000        0.0000000
       0.0000000        0.0000000

       1.0000000        1.0000000
       0.0000000        1.0000000
```

5.2 Les opérateurs relationnels

Ces opérateurs se présentent sous deux formes, symbolique ou caractère :

==	/=	>	<	>=	<=
EQ	NE	GT	LT	GE	LE

Les opérateurs similaires $E \times E$ sont :

.==	./=	.>	.<	.>=	.<=
.EQ	.NE	.GT	.LT	.GE	.LE

```
new;

output file = ccf28.out reset;

let x[2,2] = 1 0
             2 0;

let y[2,2] = 1 5
             4 2;


z = x > y;
print "x > y : " z;

z = x <= y;
print "x <= y : " z;

z = x == y;
```

```
print "x == y : " z;
print;

z = x .== y;
print "x .== y" z;

z = x ./= y;
print "x ./= y" z;

z = (x ./= y) .and (x .== y);
print "(x ./= y) .and (x .== y)" z;

output off;

x > y :        0.0000000
x <= y :        1.0000000
x == y :        0.0000000

x .== y
       1.0000000        0.0000000
       0.0000000        0.0000000
x ./= y
       0.0000000        1.0000000
       1.0000000        1.0000000
(x ./= y) .and (x .== y)
       0.0000000        0.0000000
       0.0000000        0.0000000
```

GAUSS possède aussi des opérateurs relationnels flous (*fuzzy*) sous forme de commandes :

FEQ	FNE	FGT	FLT	FGE	FLE
DOTFEQ	DOTFNE	DOTFGT	DOTFLT	DOTFGE	DOTFLE

6 Les structures de contrôle

6.1 Les boucles

Trois formes de boucles existent

```
do while expr1; expr2 ; endo;
do until expr1; expr2 ; endo;
for i(start,end,incr); expr ; endfor;
```

Pour `do while` (`do until`), l'expression `expr2` est exécutée tant que l'expression `expr1` est vraie (fausse).

```
new;

output file = ccf29.out reset;

som1 = 0;

i = 1;
```

```
do while i <= 100;
  som1 = som1 + i^2;
  i = i + 1;
endo;

som2 = 0;

i = 1;
do until i > 100;
  som2 = som2 + i^2;
  i = i + 1;
endo;

som3 = 0;

i = 1;
do until i > 100;
  som3 = som3 + i^2;
  i = i + 2;
endo;

print ftos(som1,"somme des carres des 100 premiers entiers : %lf",10,0);
print ftos(som2,"somme des carres des 100 premiers entiers : %lf",10,0);
print ftos(som3,"somme des carres des 50 premiers entiers pairs : %lf",10,0);

output off;
```

```
somme des carres des 100 premiers entiers :     338350
somme des carres des 100 premiers entiers :     338350
somme des carres des 50 premiers entiers pairs :     166650
```

L'expression conditionnelle de la boucle peut être plus complexe comme dans l'exemple suivant.

```
new;

output file = ccf30.out reset;

rndseed 123456789;

i = 1;
do while (rndu(1,1) >= 0.5);
  print "pile";
  i = i + 1;
endo;

output off;
```

```
pile
pile
pile
pile
pile
```

Remarque : nous pouvons aussi utiliser une boucle infinie, par exemple `do while 1`.

6.2 Les instructions `if`

Plusieurs formes existent :

1.
```
if expr;
  ...
endif;
\item  if expr;
  ...
else;
  ...
endif;
\item  if expr1;
  ...
elseif expr2;
  ...
else;
  ...
endif;
```

L'expression est évaluée si la condition est vraie (`if 1` est donc équivalent à `if 2`). La troisième forme est identique à celle du langage Ada 95 (`elsif`).

```
new;

output file = ccf31.out reset;

rndseed 123;

print; print "if 1"; print;

if rndu(1,1) >= 0.5;
  print "pile";
endif;

print; print "if 2"; print;

if rndu(1,1) >= 0.5;
  print "pile";
else;
  print "face";
endif;

print; print "if 3"; print;

i = 1;
do until i > 100;

  if rndu(1,1) >= 0.5;
    print "pile ";;
  else;
    print "face ";;
  endif;
```

```
  i = i + 1;
endo;


output off;

if 1

pile

if 2

face

if 3

face pile face face pile face face face pile face pile face pile pile pile pile
face pile pile pile pile face pile pile face pile pile face pile face pile face
pile pile face face pile pile face face pile pile pile pile face pile pile face
pile pile pile face pile pile pile pile face face face face pile pile face pile
face pile pile pile face pile pile pile face pile face face face face pile face
face face pile pile face face pile face pile pile pile face face face pile pile
face pile pile pile
```

6.3 Les instructions `break` et `continue`

Ces commandes permettent d'agir sur les boucles. `break` permet de sortir de la boucle (*bottom*) alors que `continue` permet de sauter au début de la boucle (`top`). Ces commandes sont les mêmes que celles du C.

```
new;

output file = ccf32.out reset;

rndseed 123;

s1 = 0;
s2 = 0;

i = 1;
do until i > 100;

  e = rndu(1,1);

  if e <= 0.48;
    print "gagne ";;
    s1 = s1 + 1;
  elseif e <= 0.96;
    print "perdu ";;
    s2 = s2 + 1;
  else;
    print; print; print ftos(i,"pour i = %lf, on recommence",1,0); print;
```

```
    continue;
  endif;

  if abs(s1-s2) > 10;
    print; print;
    print ftos(i,"le jeu s'arrete pour i = %lf",1,0);
    break;
  endif;

  i = i + 1;
endo;

output off;
```

```
perdu gagne gagne perdu gagne gagne perdu gagne gagne gagne perdu gagne

pour i = 13, on recommence

gagne perdu perdu perdu perdu gagne perdu perdu perdu perdu gagne perdu perdu ga
gne perdu perdu gagne perdu gagne perdu gagne perdu perdu gagne gagne perdu perd
u perdu gagne perdu

pour i = 43, on recommence

perdu

pour i = 44, on recommence

gagne perdu perdu gagne perdu perdu perdu gagne perdu perdu

le jeu s'arrete pour i = 53
```

7 Quelques exemples

7.1 Moyenne mobile

Dans l'exemple suivant, nous simulons un processus ARMA(2,1), puis nous calculons une moyenne mobile à l'aide de la boucle `do until`. Nous ignorons les extrémités en utilisant la commande `continue`.

```
new;
library pgraph;

u = rndn(100,1)*0.5;
e = missrv(u - 0.3*lag1(u),0);

/*
**  Simulation d'un processus ARMA(2,1)
**
**  y(t) = 0.5*y(t-1) + 0.3*y(t-2) + u(t) - 0.3*u(t-1)
**
**  u(t) = N(0,0.5)
**
```

```
**  y(0) = 0
**  y(1) = 0
*/

y = recserar(e,0|0,0.5|0.3);

m = zeros(100,1);
NA = miss(0,0);

i = 1;
do until i > 100;

  if ( i < 10 ) or ( i > 90 );
    m[i] = NA;
    i = i + 1;
    continue;
  endif;

  indx = seqa(i-9,1,19);
  m[i] = meanc(y[indx]);

  i = i + 1;
endo;

graphset;
   _pdate = ""; _pnum = 2;
   t = seqa(1,1,100);
   xy(t,y~m);
```

Remarque : Ce programme comporte de nombreuses inefficiences. Par exemple, nous pouvons éviter la boucle en employant l'opérateur de convolution (`conv`). Si nous désirons garder la boucle, nous pouvons éviter l'appel systématique de la commande seqa. **Comment modifier ce code pour calculer la moyenne mobile de plusieurs processus ARMA ?**

7.2 Simulation d'un processus ARCH

Pour simuler un processus ARCH, nous utilisons le générateur de nombres aléatoires gaussiens. Les vecteurs `u` et `h2` représentent les éléments u_t et h_t^2. L'affectation des éléments est différente selon le vecteur. `u` est parfaitement déclaré avant la boucle. Il est initialisé au vecteur nul. Ensuite, nous affectons chaque élément par la commande `u[t] = ut;`. Pour le vecteur `h`, nous choisissons une affectation indirecte, la concaténation des éléments. Cela implique qu'à chaque itération, il y a redimension du vecteur et donc modification de l'espace mémoire alloué. Cette deuxième solution est moins rapide que la première. Mais elle peut être très utile dans le cas où les dimensions du vecteur sont inconnues.

```
new;
library pgraph;

/*
**  Simulation d'une composante ARCH(1,1)
**
**  e(t) = N(0,h(t))
**
```

```
**  h(t)^2 = alpha0 + alpha1*u(t-1)^2 + beta1*h(t-1)^2
**
*/

alpha0 = 1.5;
alpha1 = 0.20;
beta1 = 0.50;

h2 = {};              /* Matrice nulle */
u = zeros(100,1);

h2t = alpha0;

t = 1;
do until t > rows(u);

  h2 = h2|h2t;
  ut = rndn(1,1)*sqrt(h2t);
  u[t] = ut;
  h2t = alpha0 + alpha1*(ut^2) + beta1*h2t;

  t = t + 1;
endo;

graphset;
  _pdate = ""; _pnum = 2;
  title("composante ARCH");
  xy(seqa(1,1,rows(u)),u);

  title("variance conditionnelle");
  xy(seqa(1,1,rows(u)),h2);
```

7.3 Vectorisation

Comment modifier le code précédent pour simuler plusieurs composantes ARCH. Une solution évidente est d'ajouter une boucle. Cette méthode n'est pas efficiente en langage GAUSS. Une meilleure solution est de vectoriser le code, c'est-à-dire d'utiliser au maximum les opérateurs et les commandes vectorielles pour privilégier le calcul matriciel au calcul scalaire. `u` et `h2` deviennent des matrices de dimension $Nobs \times K$. Chaque colonne représente une série. Remarquez que nous simulons les données avec des vecteurs lignes.

```
new;
library pgraph;

/*
**  Simulation de plusieurs composantes ARCH
**
*/

alpha0 = 1.5;
alpha1 = 0.20;
beta1 = 0.50;
```

```
@ Vous pouvez essayez :

alpha0 = 1.5~3;
alpha1 = 0.20~0.50;
beta1 = 0.50~0.20;

K = 2;

@

Nobs = 100;     /* Nombre d'observations */
K = 3;          /* Nombre de series      */

u = zeros(Nobs,K);
h2 = u;

h2t = alpha0.*ones(1,K);

t = 1;
do until t > Nobs;

  h2[t,.] = h2t;
  ut = rndn(1,K).*sqrt(h2t);
  u[t,.] = ut;
  h2t = alpha0 + alpha1.*(ut^2) + beta1.*h2t;

  t = t + 1;
endo;

graphset;
  _pdate = ""; _pnum = 2;
  title("composante ARCH");
  xy(seqa(1,1,Nobs),u);

  title("variance conditionnelle");
  xy(seqa(1,1,Nobs),h2);
```

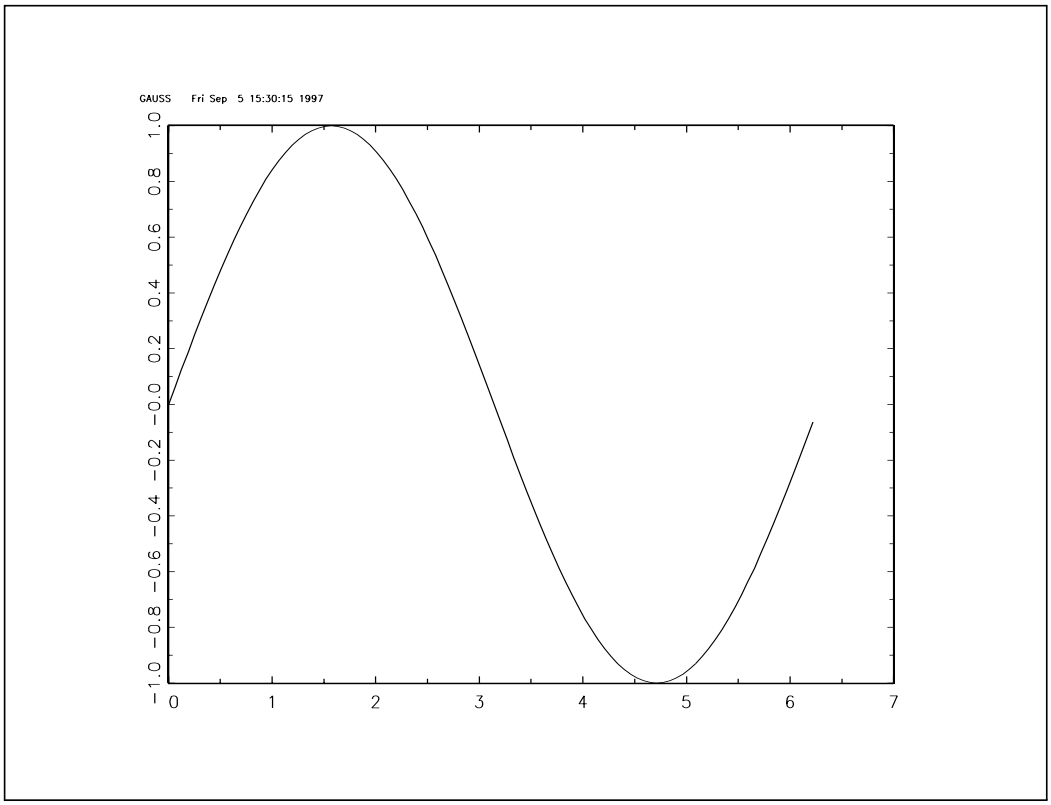
8 La bibliothèque graphique

La bibliothèque graphique de GAUSS est très riche. Elle est construite à partir de GraphiC et intègre les principales routines de cette librairie: graphes 2D (`xy`, `logx`, `logy`, `loglog`, `polar`), histogrammes (`hist`, `histf`, `histp`), graphes 3D (`contour`, `surface`, `xyz`), diagrammes (`bar`, `box`). Elle supporte les fenêtres multiples (`begwind`, `makewind`, `nextwind`, `setwind`, `window`), la transparence, les légendes, etc. Maîtriser parfaitement cette bibliothèque demande cependant quelques heures de manipulation.

8.1 La commande xy

Pour utiliser la bibliothèque graphique de GAUSS, nous devons l'activer avec la commande `library pgraph;`. La commande `graphset` permet de réinitialiser les variables globales.

```
new;
```

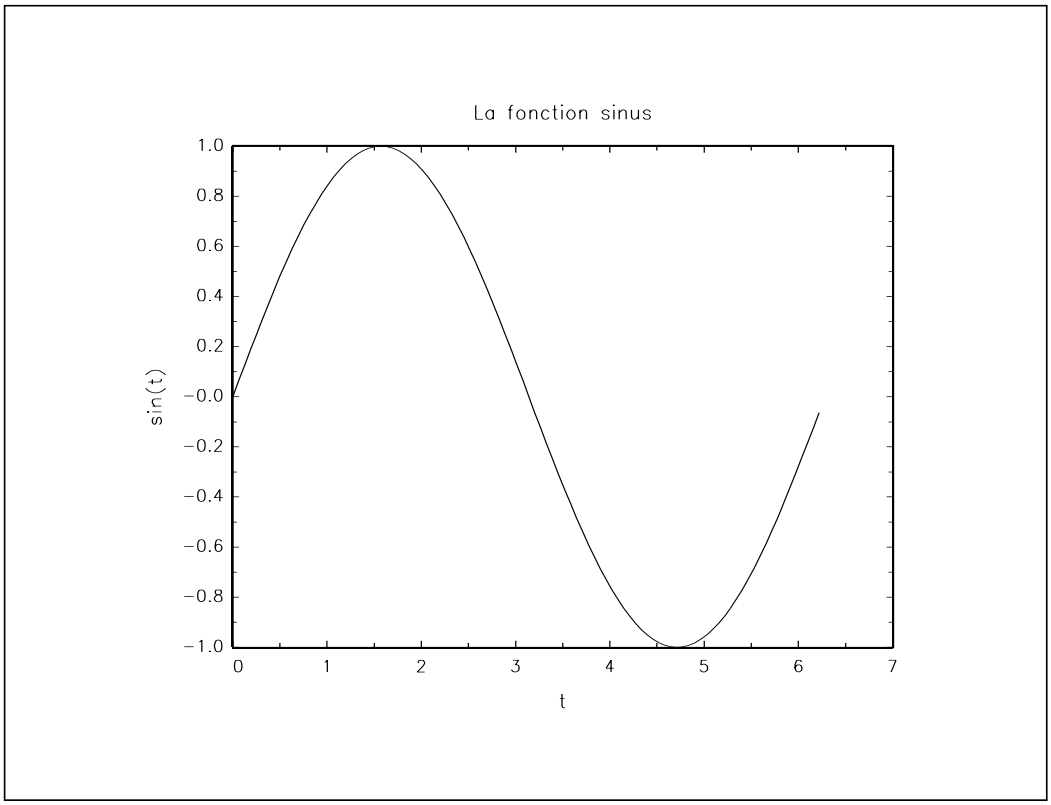


Graphique 1:

```
library pgraph;

t = seqa(0,2*pi/100,100);
x = sin(t);

graphset;
  graphprt("-c=1 -cf=ccf36.eps -w=5");
  xy(t,x);
```

Nous pouvons très simplement modifier le graphique (1) pour ajouter un titres (`title`), des labels (`xlabel` et `ylabel`) ou changer l'orientation des axes (`_pnum`).

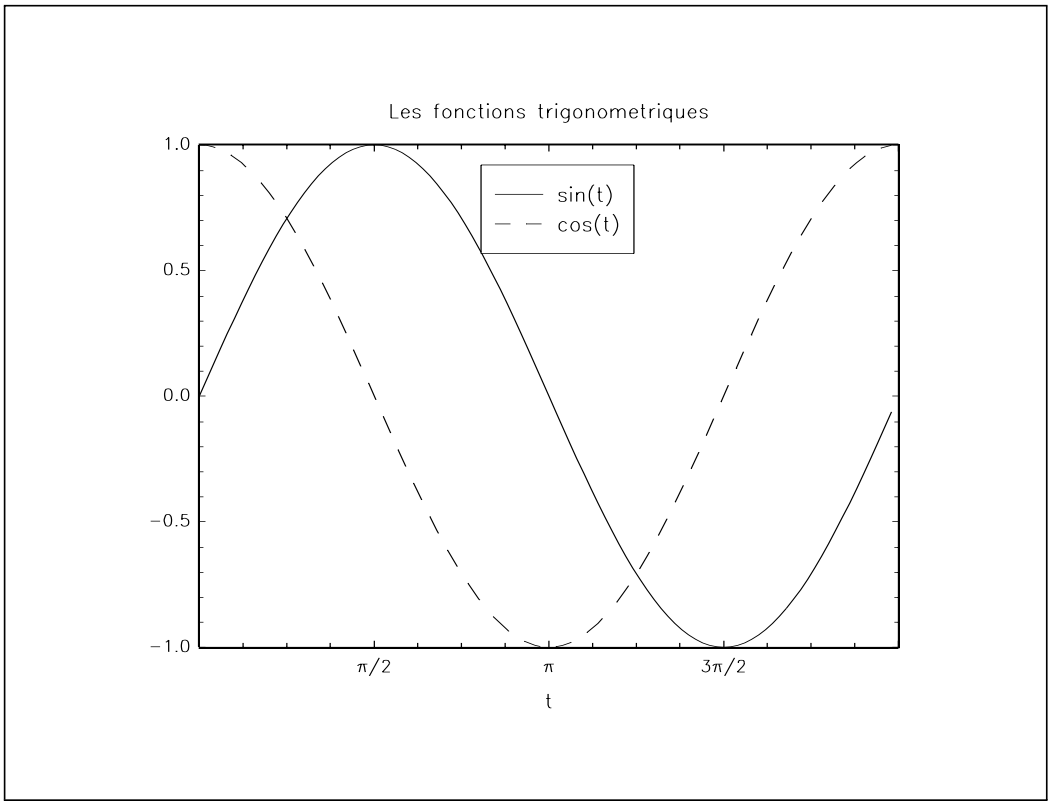
```
new;
library pgraph;

t = seqa(0,2*pi/100,100);
x = sin(t);

graphset;
    _pdate = "";
    _pnum = 2;
    title("La fonction sinus");
    xlabel("t");
    ylabel("sin(t)");
    graphprt("-c=1 -cf=ccf37.eps -w=5");
    xy(t,x);
```



Graphique 2:

La commande `xy` accepte des matrices pour les entrées, ce qui permet d'afficher plusieurs courbes. Nous plaçons le texte de la légende (qui correspond à la variable `_plegstr`) en employant la variable globale `_plegctl`. Pour modifier l'échelle des axes, nous pouvons utiliser les commandes `scale` ou `xtics` et `ytics`. Pour accéder aux lettres grecques et aux symboles mathématiques, nous activons la police de caractères `simgrma`.

```
new;
library pgraph;

t = seqa(0,2*pi/100,100);
x1 = sin(t);
x2 = cos(t);

graphset;
    fonts("simplex simgrma");
    _pdate = "";
    _pnum = 2;
    title("Les fonctions trigonometriques");
    xlabel("t");
    _plegstr = "sin(t)\000cos(t)";
    _plegctl = {2 6 4 5};
    xtics(0,2*pi,pi/2,4);
    ytics(-1,1,0.5,5);
    labelX = "0 \202p\201/2 \202p \2013\202p\201/2 2\202p";
    asclabel(labelX,0);
    graphprt("-c=1 -cf=ccf38.eps -w=5");
    xy(t,x1~x2);
```



Graphique 3:

8.2 Les fenêtres

Pour afficher plusieurs graphiques simultanément, nous devons découper l'écran en plusieurs fenêtres. Pour cela, nous initialisons le mode fenêtre avec la commande `begwind;`. Nous déclarons ensuite les différentes fenêtres avec les commandes `window` et `makewind`. Pour accéder à la i-ième fenêtre, nous employons la commande `setwind(i);`. `endwind` enfin permet d'obtenir l'affichage du graphe.

8.2.1 La commande `window`

Cette commande permet de générer des fenêtres de même dimension. Elle comporte trois arguments, le nombre de lignes, le nombre de colonnes et le mode d'affichage (transparent/non transparent).

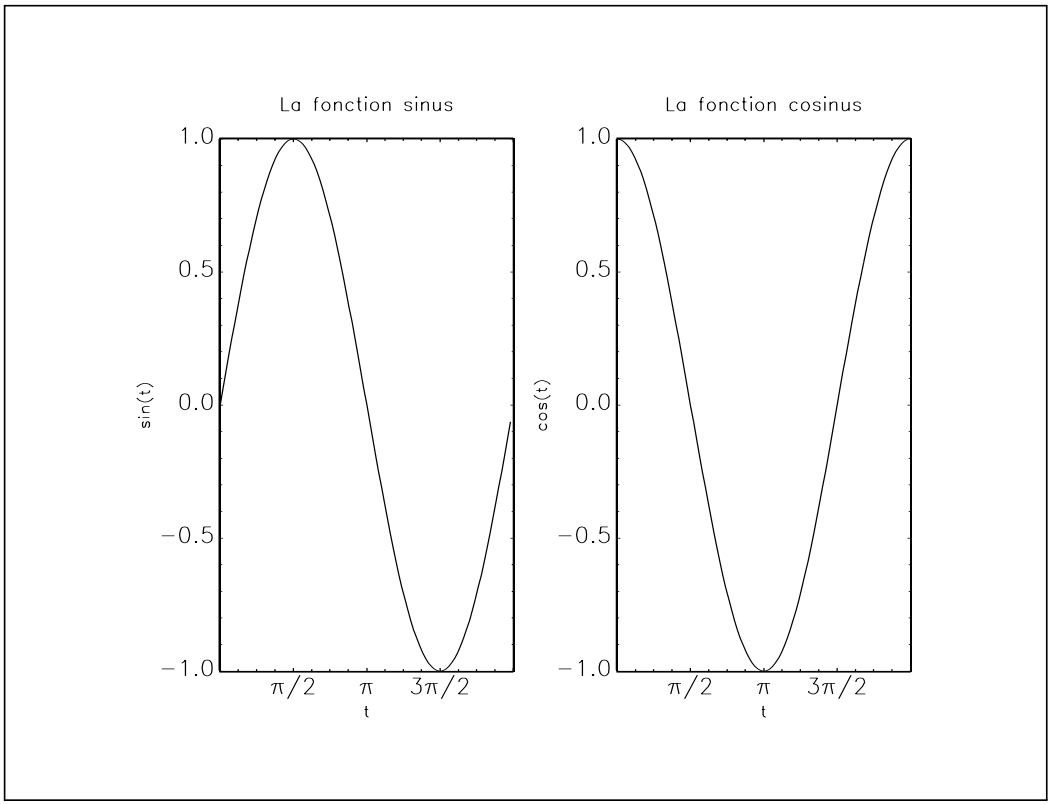
```
new;
library pgraph;

t = seqa(0,2*pi/100,100);
x1 = sin(t);
x2 = cos(t);

graphset;
    begwind;
    window(1,2,0);

      fonts("simplex simgrma");
      _pdate = "";
```

```
      _pnum = 2;
      xtics(0,2*pi,pi/2,4);
      ytics(-1,1,0.5,5);
      labelX = "0 \202p\201/2 \202p \2013\202p\201/2 2\202p";
      asclabel(labelX,0);

      _paxht = 0.25;
      _ptitlht = 0.30;
      _pnumht = 0.35;

    setwind(1);

      title("La fonction sinus");
      xlabel("t");
      ylabel("sin(t)");
      xy(t,x1);

    setwind(2);

      title("La fonction cosinus");
      xlabel("t");
      ylabel("cos(t)");
      xy(t,x2);

      graphprt("-c=1 -cf=ccf39.eps -w=5");

   endwind;
```

8.2.2 La commande `makewind`

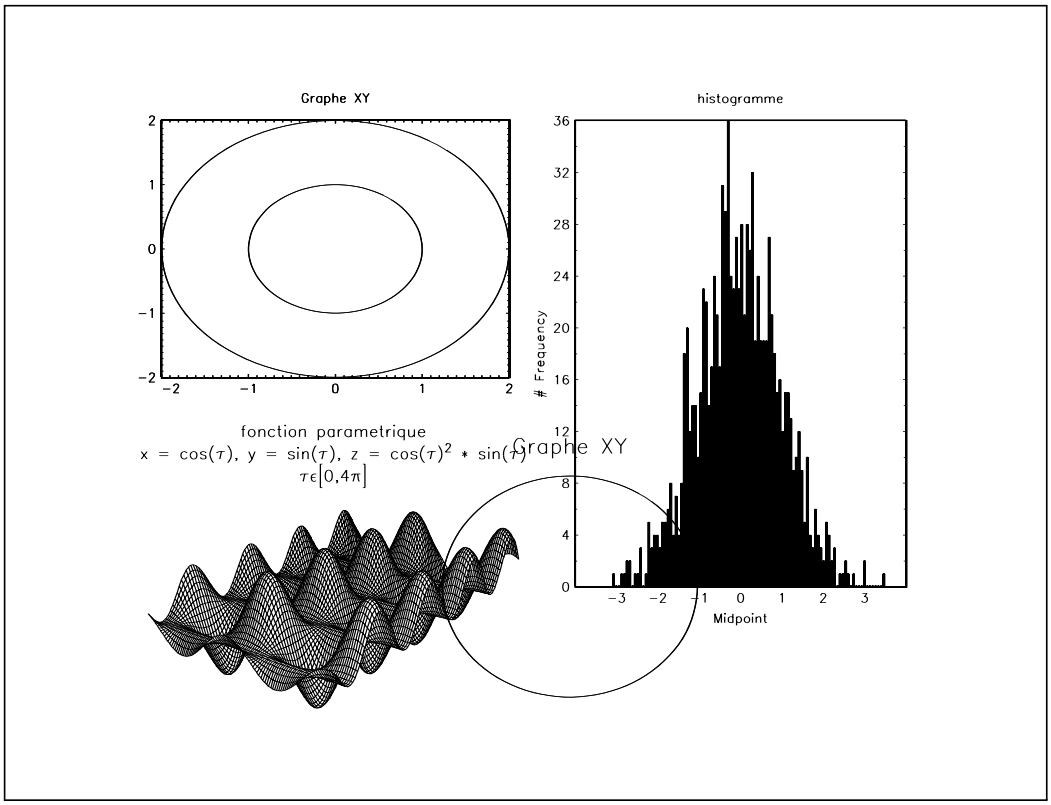
Cette commande permet de générer des fenêtres de dimensions variables. Elle comporte 5 arguments : les dimensions horizontale et verticale, les positions horizontale et verticale et le mode d'affichage. L'unité de mesure est le pouce et la dimension de l'écran est 9 pouces $\times$ 6.855 pouces.

```
new;
library pgraph;

t = seqa(1,4*pi/101,101);
x = cos(t);
y = sin(t);
z = (x'^2) .* y;

n = rndn(1000,1);

graphset;
   begwind;
   makewind(9/2,6.855/2,0,0,0);
   makewind(9/2,6.855-1,9/2,1,0);
   makewind(9/2,6.855/2,0,6.855/2,0);
   makewind(9/2,6.855/2,0,6.855/2,1);
   makewind(3,3,3.5,0.25,1);
```



Graphique 4:

```
_pdate="";

setwind(1);

  fonts("simplex simgrma");
  title("fonction parametrique"\
        "\Lx = cos(\202t\201),"\
        " y = sin(\202t\201),"\
        " z = cos(\202t\201)[2] * sin(\202t\201)"\
        "\L\202t!@[\2010,4\202p@]");
  _paxes = 0; _pframe = 0; _psurf = {0,0}; _pzclr = {1,9,4}; _ptitlht = 0.25;
  volume(3,2,1);
  surface(x',y,z);

setwind(2);
  graphset;
  title("histogramme");
  _ptitlht = 0.20; _pnumht = 0.20; _paxht = 0.20;
  _pnum = 2;
  call hist(n,100);

setwind(3);
  title("Graphe XY");
  xtics(-2,2,1,10);
  ytics(-2,2,1,10);
  xy(x,y);
```



Graphique 5:

```
setwind(4);
  xy(2*x,2*y);

setwind(5);
  _ptitlht = 0.50;
  _pnum = 0; _paxes = 0;
  xy(2*x,2*y);

graphprt("-c=1 -cf=ccf40.eps -w=5");

endwind;
```

9 Les bases de données

Nous pouvons sauvegarder les matrices dans un fichier avec l'instruction `save`. L'extension du fichier est *fmt*. Pour charger ces matrices, nous utilisons la commande `load`. Considérons l'exemple suivant. Nous générons la matrice `x = rndn(1000,2);`. `save x` crée un fichier *x.fmt*. `load x` permet de charger la matrice du fichier *x.fmt*. L'instruction `load data = x` charge les données de la matrice du fichier *x.fmt* dans la matrice `data`. Nous pouvons aussi sauvegarder la matrice `x` sous un autre nom, par exemple `save y = x`.

Les bases de données présentent de nombreuses ressemblances avec les matrices. Mais les outils de gestion de ces bases sont plus souples que ceux des matrices.

9.1 Gestion des bases

Nous pouvons employer la commande `saved` pour créer une base de données. Les arguments sont la matrice des données, le nom de la base et les noms des variables. Deux fichiers sont alors créés : *uneBase.dat* et *uneBase.dht*.

```
new;

load data[20,8] = base1.asc;

let nom_des_variables = constante x2 x3 x4 x5 y1 y2 y3;
call saved(data,"uneBase",nom_des_variables);
```

La commande `getname` permet d'obtenir les noms de variables de la base de données. GAUSS possède de nombreuses commandes dont l'argument peut être une base de données ou une matrice, par exemple `dstat`. L'utilisateur peut penser qu'il est équivalent de charger la base de données dans une matrice, puis d'exécuter la commande `dstat` avec cette matrice. Pourtant, il y a une différence importante. La matrice occupe de la mémoire vive, alors que la base de données n'en occupe pas (ou presque). Cela implique que pour des bases de données très grosses, le programme peut s'arrêter car il n'y a pas assez de mémoire vive.

```
new;

output file = ccf42.out reset;

nom = getname("uneBase");
print $nom;

call dstat("uneBase",nom[1 3 5 7 8]);

output off;
```

```
        CONSTANT
              X2
              X3
              X4
              X5
              Y1
              Y2
              Y3

-------------------------------------------------------------------------------
Variable       Mean   Std Dev    Variance   Minimum   Maximum     Valid Missing
-------------------------------------------------------------------------------
CONSTANT     1.0000    0.0000      0.0000    1.0000    1.0000        20    0
X3           3.3855    1.9676      3.8715    1.3400    6.6900        20    0
X5          41.6000   14.7163    216.5684   22.0000   88.0000        20    0
Y2         156.5365   39.4681   1557.7276  102.9600  235.4300        20    0
Y3         904.5630  247.1616  61088.8428  578.4900 1401.9400        20    0
```

La commande `loadd` charge une base de données dans une matrice. Cette instruction ne fonctionne que pour des bases de données relativement petites.

```
new;

output file = ccf43.out reset;

y = loadd("uneBase");

let indx = 2 3 6;

print corrx(y[.,indx]);

output off;

       1.0000000       0.95484050       0.92269517
      0.95484050        1.0000000       0.90451570
      0.92269517       0.90451570        1.0000000
```

GAUSS possède de nombreuses commandes pour la gestion des bases : `close`, `closeall`, `create`, `eof`, `open`, `readr`, `seekr` et `writer`. Pour obtenir les nombres d'observations et de variables, nous employons `rowsf` et `colsf`.

```
new;
cls;

output file = ccf44.out reset;

open fichier = uneBase for read;

Nobs = rowsf(fichier);
k = colsf(fichier);

print ftos(Nobs,"Nombre d'observations : %lf",2,0);
print ftos(k,   "Nombre de variables :   %lf",2,0);

closeall;

output off;

Nombre d'observations : 20
Nombre de variables :    8
```

Dans l'exemple suivant, nous utilisons la commande `readr` pour lire les données. La commande `seekr` permet de se positionner dans la base et `seekr(fichier,-1)` renvoie la position dans la base.

```
new;

data = {};

open fichier = uneBase for read;

do until eof(fichier);

  y = readr(fichier,2);
```

```
  data = data|y;

  pos = seekr(fichier,-1);
  print ftos(pos,"Position dans le fichier : %lf",2,0);

endo;

call close(fichier);
```

Lorsque nous utilisons la commande `saved`, nous avons une contrainte, puisque la matrice de données est en mémoire vive. Pour des grosses bases de données (plus de 128 Mo par exemple), cela n'est pas efficient. Une solution est de créer la base séquentiellement. Dans l'exemple suivant, la base *revBase* est une copie de *uneBase*, mais les observations sont rangés dans l'ordre contraire.

```
new;

Base = "uneBase";
Base2 = "revBase";

nom_des_variables = getname(Base);
k = rows(nom_des_variables);

create fichier = ^Base2 with ^nom_des_variables,k,4;
call close(fichier);

print $getname(Base2);

new;

open fichier1 = uneBase for read;
open fichier2 = revBase for append;

Nobs = rowsf(fichier1);

i = 0;
do until i > Nobs -1;

  pos = seekr(fichier1,Nobs-i);
  y = readr(fichier1,1);
  call writer(fichier2,y);

  i = i + 1;
endo;

closeall;
```

9.2 Les formats v89 dos et v96 universal

Deux formats sont désormais disponibles. Avec l'ancien format v89 dos, deux fichiers sont nécessaires pour constituer une base. Le fichier *.dat* contient les données (format binaire) et le fichier *.dht* contient les informations (par exemple, les noms des variables). Avec le nouveau format, nous n'avons plus qu'un seul fichier avec une extension *dat*.

10 Les entrées et sorties

Le paragraphe suivant est extrait de "GAUSS et la Finance".

10.1 La construction d'une base de données GAUSS à partir d'un fichier ASCII : l'exemple de la base HFDF93 d'Olsen & Associates

HFDF93 de la société suisse Olsen & Associates est une base de données en temps continu sur les cours de change. Le fichier ASCII *dem&usd.dat* contient l'évolution du cours de change DEM/USD pour la période allant du 1er octobre 1992 au 30 septembre 1993. Ce fichier est cependant très difficile à exploiter, puisqu'il contient plus de 1 300 000 observations et représente 67.3 Mo d'information. Le fichier *dem&usd.dat* se présente de la façon suivante :

```
;(****************** P R O P R I E T A R Y   F I L E **********************)
;(* This file is part of the Olsen & Associates data distribution: HFDF93 *)
;(* The entire data distribution as well as this file are the property of *)
;(* Olsen & Associates and are proprietary and confidential. It is not    *)
;(* permitted to disclose, copy or distribute this file, neither wholly   *)
;(* nor in part, except by written permission of Olsen & Associates.      *)
;(* Olsen & Associates shall not be liable for any loss or damage         *)
;(* whatsoever caused arising directly or indirectly in connection with   *)
;(* the use of the data set.                                              *)
;(***************************************************************************)
;
;                          Olsen & Associates
;              Research Institute for Applied Economics.
;           Seefeldstrasse 233, CH-8008 Zurich, Switzerland.
;                          E-mail: hfdf@olsen.ch
;                        Telephone: 41-1-386-48-48
;
; Tick extraction for DEM^USD
; ---------------------------
;
; Filtering: O&A standard filter for historical tests
;
;   date     time       price    country      filter
;                                    city     Good (1) or Bad (0)
;CCYY-MM-DD  (GMT)    bid    ask          bank
1992-10-01 00:00:14 1.4116 1.4121 392 01 0058 1
1992-10-01 00:00:54 1.4108 1.4118 036 02 0130 1
...
```

GAUSS possède de nouvelles commandes pour la gestion des fichiers I/O : `close`, `closeall`, `eof`, `fcheckerr`, `fclearerr`, `fflush`, `fgets`, `fgetsa`, `fgetsat`, `fgetst`, `fopen`, `fputs`, `fputst`, `fseek`, `fstrerror`, `ftell`. Ces nouvelles commandes permettent notamment de manipuler de gros fichiers ASCII. Le programme suivant construit à partir du fichier *dem&usd.dat* un fichier ASCII *dem&usd.asc* contenant les informations suivantes

Annee	Mois	Jour	Heure (en secondes)	cours BID

Chaque ligne d'observation du fichier *dem&usd.dat* est traitée comme une chaîne de caractères. Nous manipulons ensuite cette chaîne de caractères afin d'extraire et/ou construire les informations qui nous intéressent. Nous pouvons ensuite créer la nouvelle base de données (`call fputst(fichier2,str);`).

```
new;

fichier1 = fopen("dem&usd.dat","r");
fichier2 = fopen("dem&usd.asc","w");

i = 1;
do until i > 50;
  str = fgets(fichier1,100);
  i = i + 1;
endo;

blanc = "   ";

i = 1;
do until eof(fichier1);
  str = fgets(fichier1,60);

  {date_,str} = token(str);
  {heure,str} = token(str);
  {bid,str} = token(str);
  {ask,str} = token(str);
  {pays,str} = token(str);
  {ville,str} = token(str);
  {banque,str} = token(str);
  {filtre,str} = token(str);

  annee = strsect(date_,1,4);
  mois = strsect(date_,6,2);
  jour = strsect(date_,9,2);

  heures = strsect(heure,1,2);
  minutes = strsect(heure,4,2);
  secondes = strsect(heure,7,2);

  temps = 3600*stof(heures) + 60*stof(minutes) + stof(secondes);
  temps = ftos(temps,"%*.*lf",6,0);

  str = annee $+ blanc $+ mois $+ blanc $+ jour $+ blanc $+ temps;
  str = str $+ blanc $+ bid;

  call fputst(fichier2,str);

endo;

closeall;
```

Le fichier *dem&usd.asc* se présente de la façon suivante :

```
1992   10   01      300   1.4113
1992   10   01      314   1.4103
1992   10   01      326   1.4110
1992   10   01      340   1.4110
1992   10   01      348   1.4105
...
```

Nous pouvons aussi utiliser les nouvelles commandes de gestion des fichiers I/O pour créer une base de données GAUSS. L'avantage de manipuler une base de données au format GAUSS est la possibilité de lecture selective des observations et des variables. Nous pouvons par exemple sélectionner les cours du Lundi, construire les rendements horaires, minutes, 30 secondes ou 15 secondes, ne considérer que les cours cotés entre 11 heures et 13 heures pour le mois de janvier 1993, etc.

```
new;

fichier1 = fopen("dem&usd.dat","r");

let vname = jour heure bid ask filtre;
create fichier2 = forex with ^vname,5,4;

i = 1;
do until i > 50;
  str = fgets(fichier1,100);
  i = i + 1;
endo;

date_de_reference = 1992|01|01;

i = 1;
do until eof(fichier1);
  str = fgets(fichier1,60);

  {date_,str} = token(str);
  {heure,str} = token(str);
  {bid,str} = token(str);
  {ask,str} = token(str);
  {pays,str} = token(str);
  {ville,str} = token(str);
  {banque,str} = token(str);
  {filtre,str} = token(str);

  annee = strsect(date_,1,4);
  mois = strsect(date_,6,2);
  jour = strsect(date_,9,2);

  heures = strsect(heure,1,2);
  minutes = strsect(heure,4,2);
  secondes = strsect(heure,7,2);
```

```
  jour = etdays(date_de_reference,stof(annee)|stof(mois)|stof(jour));
  heure = 3600*stof(heures) + 60*stof(minutes) + stof(secondes);
  bid = stof(bid);
  ask = stof(ask);
  filtre = stof(filtre);

  x = jour~heure~bid~ask~filtre;
  call writer(fichier2,x);

endo;

closeall;
```

Il faut 10 minutes pour construire la base de données GAUSS précédente avec un Pentium 100. La lecture de la base de données est ensuite très rapide (environ 1 minute pour la base précédente, qui contient 1 300 000 observations).

10.2 Quelques exemples

```
/*
** tools7.src - TOOLS library.
** (C) Copyright 1997 by Thierry Roncalli.
** All Rights Reserved.
**
**  Format                          Purpose                                 Line
** ============================================================================
**  newstr = string_replace(str,substr1,substr2);
**                                  replace a substring of a string by
**                                  another substring                        31
**
**  call text_replace(finput,foutput,text);
**                                  replace substrings of a ascii file by
**                                  another substrings                       82
**
**  call text_upper(finput,foutput);
**                                  convert characters of a ascii file to
**                                  uppercase                               125
**
**  call text_lower(finput,foutput);
**                                  convert characters of a ascii file to
**                                  lowercase                               157
**
**  call text_print(finput);        print ascii files to the screen and/or
**                                  auxiliary output                        189
**
*/


/*
** string_replace
**
```

```
** Purpose: replace a substring of a string by another substring
**
** Format:  newstr = string_replace(str,substr1,substr2);
**
** Input:     str - string, the string
**        substr1 - string, the substring to be replaced
**        substr2 - string, the substring which replaces substr1
**
** Output: newstr - string, the new string
**
*/

proc (1) = string_replace(str,substr1,substr2);
  local newstr,l,l1,pos;

  newstr = str;

  l =  strlen(str);
  l1 =  strlen(substr1);

  pos = 1;

  do while pos == 1;
    {newstr,pos} = _string_replace(newstr,substr1,substr2,l,l1);
  endo;

  retp(newstr);
endp;


proc (2) = _string_replace(str,substr1,substr2,l,l1);
  local pos,str1,str2,newstr;

  pos = strindx(str,substr1,1);

  if pos == 0;
    retp(str,0);
  endif;

  str1 = strsect(str,1,pos-1);
  str2 = strsect(str,pos+l1,l);

  newstr = str1 $+ substr2 $+ str2;

  retp(newstr,1);
endp;


/*
** text_replace
**
** Purpose: replace substrings of a ascii file by another substrings
**
```

```
** Format:  call text_replace(finput,foutput,text);
**
** Input:  finput - string, input file
**        foutput - string, output file
**           text - N*2 string array, the substrings
**                  text[.,1] ---> substrings to be replaced
**                  text[.,2] ---> substrings which replace text[.,1]
**
*/

proc (0) = text_replace(finput,foutput,text);
  local N,fr,fw,str,i;

  N = rows(text);

  fr = fopen(finput,"r");
  fw = fopen(foutput,"w");


  do until eof(fr);
    str = fgetsa(fr,1);

    i = 1;
    do until i > N;
      str = string_replace(str,text[i,1],text[i,2]);
      i = i + 1;
    endo;

    call fputs(fw,str);
  endo;

  fr = close(fr);
  fw = close(fw);

  retp;
endp;


/*
** text_upper
**
** Purpose: convert characters of a ascii file to uppercase
**
** Format:  call text_upper(finput,foutput);
**
** Input:  finput - string, input file
**        foutput - string, output file
**
*/

proc (0) = text_upper(finput,foutput);
  local fr,fw,str;
```

```
  fr = fopen(finput,"r");
  fw = fopen(foutput,"w");


  do until eof(fr);
    str = fgetsa(fr,1);

    call fputs(fw,upper(str));
  endo;

  fr = close(fr);
  fw = close(fw);

  retp;
endp;


/*
** text_lower
**
** Purpose: convert characters of a ascii file to lowercase
**
** Format:  call text_lower(finput,foutput);
**
** Input:  finput - string, input file
**         foutput - string, output file
**
*/

proc (0) = text_lower(finput,foutput);
  local fr,fw,str;

  fr = fopen(finput,"r");
  fw = fopen(foutput,"w");


  do until eof(fr);
    str = fgetsa(fr,1);

    call fputs(fw,lower(str));
  endo;

  fr = close(fr);
  fw = close(fw);

  retp;
endp;


/*
** text_print
**
** Purpose: print ascii files to the screen and/or auxiliary output
```

```
**
** Format:  call text_print(finput);
**
** Input:  finput - string, input file
**
*/

proc (0) = text_print(finput);
  local fr,str;

  fr = fopen(finput,"r");


  do until eof(fr);
    str = fgetsa(fr,1);
    print strsect(str,1,strlen(str)-1);
  endo;

  fr = close(fr);

  retp;
endp;
```

11 Introduction au niveau II

11.1 Qu'est-ce qu'une procédure ?

```
new;

output file = ccf48.out reset;

x = 2;
y = f(x);
print y;
y = f(y);
print y;
x = f(3);
print x;

output off;

proc f(x);
  retp( x^2);
endp;
```

```
       4.0000000
       16.000000
       9.0000000
```

```
new;

output file = ccf49.out reset;
```

```
x = 2;
y = f(x);
print y;
print "x = " x;
y = f(y);
print y;
x = f(3);
print x;

output off;

proc f(x);
  x = x^2;
  retp(x);
endp;

       4.0000000
x =        2.0000000
       16.000000
       9.0000000

new;

output file = ccf50.out reset;

y = rndn(5,1);
x = f(y);
print y~x;

output off;

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

     -0.66618707       0.34891244
       1.5889368     -0.045797170
     -0.24387024      0.057712940
      -1.3677260       0.37727301
      0.46615825       0.19411757
```

11.2 Qu'est-ce qu'un pointeur ?

```
new;

output file = ccf51.out reset;

pointeur1 = &f;
pointeur2 = &g;

print pointeur1;
```

```
print pointeur2;
print pointeur1+pointeur2;

gradp(&h,seqa(1,1,4));

h;

output off;

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


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


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

       44.000000
       88.000000
       132.00000

       2.0000000       0.00000000       0.00000000       0.00000000
      0.00000000        4.0000000       0.00000000       0.00000000
      0.00000000       0.00000000        6.0000001       0.00000000
      0.00000000       0.00000000       0.00000000        8.0000000

D:\WINGAUSS\CCF\CCF51.PRG(14) : error G0159 : Wrong number of parameters
Currently active call: H [14]
```

11.3 Qu'est-ce qu'une variable externe ?

11.4 Qu'est-ce qu'une bibliothèque ?