UNIVERSITÉ PARIS-XII — FACULTÉ DE SCIENCES ÉCONOMIQUES ET DE GESTION
Master Professionnel *Méthodes Appliquées de la Statistique et de l'Économétrie pour la Recherche, l'Analyse et le Traitement de l'Information*
Année universitaire 2006-2007 — *François LEGENDRE*
F.Legendre@univ-paris12.fr — http://fj.legendre.free.fr

Projet « Mon GAUSS »

1 Exposé du projet

La programmation « orienté objet » permet au programmeur de se définir exactement les types dont il a besoin. Dans ce projet, je vous propose de vous fabriquer un « GAUSS à vous », c'est-à-dire un langage permettant de manipuler les matrices et d'exprimer simplement des calculs matriciel. On voudrait pouvoir écrire par exemple la ligne suivante.

a_chapeau = inv(!X*X) * (!X*y) ;

Pour la transposition des matrices, on le peut pas utiliser la quote (le caractère « **'** »). Ce dernier est réservé pour délimiter, littéralement, les caractères. Les opérateurs unitaires post-fixés disponibles en C++ sont la post-incrémentation (comme dans l'expression « ***i ++*** ») et la post-décrémentation (comme dans l'expression « ***i --*** »). Ces deux opérateurs représenteraient mal, à mon avis, la transposition. Je vous propose d'utiliser l'opérateur unitaire pré-fixé de négation logique pour la transposition ; cet opérateur est représenté par le point d'exclamation. La notation mathématique X′X s'exprime alors par la syntaxe « ***!X*X*** ».

Je vous propose aussi d'être bien moins permissif que GAUSS. Les matrices ne devraient pas pouvoir changer de taille au cours de leur vie, les dimensions des matrices devraient exactement être les bonnes dans les calculs, les indices devraient appartenir à leur domaine de définition, etc.

Une matrice serait déclarée de deux manières, soit avec son constructeur normal en donnant le nombre de lignes et de colonnes, soit avec son constructeur de copie en donnant l'expression qui initialise la matrice. On pourrait coder par exemple les deux lignes suivantes.

```
// Déclaration d'une matrice carrée de taille 10x10 (constructeur normal).
Matrice A(10, 10) ;
// Déclaration d'une matrice à partir d'une expression (constructeur de copie).
Matrice B( !X*X ) ;
```

Quand une matrice est déclarée en donnant ses dimensions, les éléments sont initialisés à zéro.

Les constructeurs sont en charge d'allouer, sur le tas, la mémoire nécessaire pour contenir les éléments de la matrice. Cette mémoire est obtenue par l'opérateur ***new***. Trois attributs définissent ainsi notre objet. Les deux premiers sont les dimensions de la matrice ***lignes_*** et ***colonnes_***. Le troisième attribut, ***tableau***, désigne l'adresse du début du tableau des éléments ; cet attribut est un « pointeur » c'est-à-dire une variable qui contient l'adresse d'une autre variable. Cette variable va être renseignée avec la valeur retournée par l'opérateur ***new***. Cet attribut sera ensuite utilisé pour désigner un élément de la matrice par la syntaxe « ***tableau[i]*** » où la variable ***i*** désigne un déplacement qui commence à ***0***.

Les éléments d'une matrice sont rangés en colonne. Les six termes de la matrice

$$\begin{pmatrix} a & b \\ c & d \\ e & f \end{pmatrix}$$

sont rangés en séquence sous la forme a, c, e, b, d et f. Le déplacement pour atteindre le terme b, dont l'indice de ligne est 0 et l'indice de colonne est 1, est 3. Ce déplacement résulte donc du calcul $1\times3+0$. Il faut ainsi coder « ***tableau[j*lignes_+i]*** » pour désigner le terme ***A(i, j)***.

Il est sans doute un peu dommage de stocker les éléments en colonne. L'ordre « naturel » serait plutôt en ligne puisqu'une matrice est en général lue ligne par ligne. Il est cependant préférable de ranger les éléments en colonne si l'on souhaite utiliser des bibliothèques écrites en langage FORTRAN. Ce langage, en effet, range les éléments par colonne sans doute parce qu'il est ainsi plus facile d'extraire un vecteur colonne d'une matrice rangée de la sorte.

Le constructeur normal est défini par la ligne suivante.

Matrice(const size_t & arg1, const size_t & arg2 = 1) :

Le second argument comporte une valeur par défaut. Aussi la syntaxe « ***Matrice y(10)*** » (qui permet de déclarer un vecteur) est-elle équivalente à « ***Matrice y(10, 1)*** ». La deuxième ligne

lignes_(arg1), colonnes_(arg2) {

permet d'initialiser les attributs ***lignes_*** et ***colonnes_*** à partir des deux arguments. Ensuite, la troisième ligne

Matrice & A = *this ;

déclare une référence pour désigner plus facilement l'instance courante. Cette déclaration ne crée pas nécessairement une nouvelle variable ; le compilateur « comprend » que cette référence est simplement une manière de désigner l'instance courante. L'identificateur ***this*** est réservé, par le C++, pour désigner l'adresse de l'instance courante. Il me semble plus expressif de déclarer une référence sur cette instance. Soit la méthode ***lignes()*** qui retourne le nombre de lignes d'une matrice. Les syntaxes « ***lignes()*** », « ***this->lignes()*** » et « ***A.lignes()*** » désignent toutes la même chose. La dernière syntaxe me semble la plus expressive.

Les lignes

```
if ( (A.lignes() == 0) || (A.colonnes() == 0) ) {
  fatal("Les arguments du constructeur normal sont invalides") ; }
```

vérifient la validité des arguments. Je me contente de vérifier que les arguments ne sont pas nuls. Il faudrait vérifier que le calcul « ***A.lignes()*A.colonnes()*** » ne sature pas la capacité de représentation d'une variable de type ***size_t***.

La ligne

```
if ( (A.tableau = new double[A.lignes()*A.colonnes()]) == 0 ) {
```

fait deux choses à la fois. En premier lieu, elle appelle l'opérateur ***new*** pour allouer de la mémoire sur le tas. Un tableau de ***A.lignes()*A.colonnes()*** éléments de type ***double*** doit pouvoir tenir dans cet emplacement de mémoire. L'opérateur ***new*** retourne un pointeur qui désigne cet emplacement. Ce pointeur est affecté à l'attribut ***tableau*** de l'instance courante. En second lieu, le test d'égalité à zéro de cet attribut est effectué pour voir si l'opérateur ***new*** a bien été capable d'allouer la mémoire. Cette ligne unique pourrait être décomposée en les deux lignes suivantes.

```
A.tableau = new double[A.lignes()*A.colonnes()] ;
if ( A.tableau == 0 ) {
```

Il est assez fréquent, en C ou en C++, de composer de la sorte deux instructions. C'est possible parce que l'opérateur d'affectation (représenté par le signe « ***=*** ») est un opérateur. Aussi l'affectation est-elle une expression et non une instruction particulière. On peut par exemple écrire « ***a = b = c = 0 ;*** ». L'opérateur « ***=*** » est associatif à droite. Cette dernière expression s'interprète comme « ***a = (b = (c = 0)) ;*** ». La plupart des opérateurs sont associatifs à gauche : l'expression « ***a + b + c*** » s'interprète comme « ***(a + b) + c*** ».

Les lignes

```
for ( size_t i = 0 ; i < A.lignes() ; ++ i ) {
  for ( size_t j = 0 ; j < A.colonnes() ; ++ j ) {
    A.imprudent(i, j) = 0. ; } }
```

initialisent les éléments de la matrice à zéro. La méthode ***imprudent(i, j)*** est définie dans la section ***private*** de la déclaration de l'objet. Elle ne peut être appelée que depuis les méthodes définies dans la déclaration de l'objet (et non depuis l'extérieur). La méthode ***imprudent(i, j)*** permet d'accéder aux éléments de la matrice sans que la validité des indices ***i*** et ***j*** ne soit contrôlée. C'est pour cette raison que la méthode est ***private***.

La méthode ***imprudent(i, j)*** est définie de deux façons différentes. On trouve ainsi les lignes suivantes.

```
const double & imprudent(const size_t & i, const size_t & j) const {
  return tableau[j*lignes_ + i] ; }
double & imprudent(const size_t & i, const size_t & j) {
  return tableau[j*lignes_ + i] ; }
```

La première méthode s'applique à une instance de ***Matrice*** qui est ***const***. Elle retourne une référence de type ***const double***. La seconde méthode s'applique à une instance de ***Matrice*** qui est non ***const***. Elle retourne une référence non ***const***.

Dans la définition de la méthode ***bidule*** suivante

```
void bidule(const Matrice & B) {
  Matrice & A = *this ;
  A.imprudent(0, 0) = 10*B.imprudent(0, 0) ; }
```

le compilateur, pour ***B***, appelle la version ***const*** de ***imprudent*** parce que la matrice ***B*** est déclarée ***const***. En revanche, pour la matrice ***A***, la version non ***const*** de ***imprudent*** est utilisée. À gauche du signe « ***=*** », on ne peut faire figurer qu'une référence non ***const***. Ainsi on parvient bien à exprimer ce que l'on veut. La variable qui est déclarée ***const*** ne peut pas être modifiée alors que la variable déclarée non ***const*** peut être librement affectée dans le corps de la méthode.

Le C++ permet de définir, pour un objet, une méthode pour chacun des opérateurs habituel du langage. Habituellement, une méthode reçoit un nom. Par exemple, soit l'objet ***Matrice*** et soient ***A***, ***B*** et ***C*** trois instances de ce type. L'addition des matrices pourrait être définie en utilisant une méthode habituelle qui recevrait le nom, par exemple, ***additionner***. On utiliserait alors la syntaxe « ***C = A.additionner(B) ;*** ». La déclaration suivante, dans le corps de la déclaration de l'objet ***Matrice***, serait utilisée pour la méthode ***additionner***.

```
class Matrice {
  // Attributs privés de l'objet.
  public :
  Matrice additionner(const Matrice & B) const {
    // Corps de la méthode ; il faut là définir ce que fait 'additionner'.
  }
} ;
```

La méthode retourne une instance de l'objet ***Matrice***; l'argument de la méthode est la matrice ***B*** et l'instance courante est la matrice ***A***.

Pour pouvoir utiliser la syntaxe « ***C = A + B ;*** », il faut définir l'opérateur « ***+*** » comme suit.

```
class Matrice {
  // Attributs privés de l'objet.
  public :
  Matrice operator+(const Matrice & B) const {
    // Corps de la méthode ; il faut là définir ce que fait l'opérateur '+'.
  }
} ;
```

Le compilateur transforme alors l'expression « ***C = A + B ;*** » en l'expression « ***C = A.operator+(B) ;*** ».

Dans le projet, je me contente de définir les cinq opérateurs suivants. Tout d'abord, deux opérateurs qui sont essentiels : ***operator()*** et ***operator=***. Le premier va permettre l'accès aux éléments d'une matrice sous la syntaxe « ***A(0,0)*** » ou « ***y(0)*** ». Le second va donner un sens à la syntaxe « ***A = B*** » ; cet opérateur va correspondre à l'affectation – la recopie des éléments de la matrice ***B***. Ensuite, les opérateurs ***operator**** et ***operator!*** sont définis pour, respectivement, la multiplication des matrices et la transposition des matrices. Enfin, je définis aussi l'opérateur ***operator%*** pour la multiplication terme à terme des matrices.

Dans les expressions, un grand nombre de variables sont créées temporairement. Par exemple, l'expression

inv(!X*X) * (!X*y)

engendre six temporaires. En effet, cette expression se décompose en

```
Matrice TMP1 ( !X ) ;
Matrice TMP2 ( TMP1*X ) ;
Matrice TMP3 ( inv(TMP2) ) ;
Matrice TMP4 ( !X ) ;
Matrice TMP5 ( TMP4*y ) ;
Matrice TMP6 ( TMP3*TMP5 ) ;
```

Les méthodes qui définissent les opérateurs retournent donc, en général, une ***Matrice*** qui correspond justement à cette variable temporaire. L'opérateur ***operator=*** fait exception à cette règle. Il est défini comme retournant une ***const*** référence sur une ***Matrice***. En fait, cet opérateur se contente de retourner une ***const*** référence sur l'instance à partir de laquelle il est appelé. L'expression « ***A = B*** » est, en effet, transformée par le compilateur en l'expression « ***A.operator=(B)*** ». Cette expression pourrait être placée à droite du signe « = », par exemple, sous la forme « ***C = (A = B)*** ». En retournant une ***const*** référence sur la matrice ***A***, on permet justement à l'affectation de constituer une expression.

Il est *a priori* coûteux de retourner un objet temporaire. Soit l'objet ***Gigantesque*** et la méthode ***bidule*** qui retourne un ***Gigantesque*** temporaire. Ceci serait codé comme suit.

```
class Gigantesque {
  // Attributs privés de l'objet.
  public :
  Gigantesque bidule() {
    Gigantesque temporaire ; // Le constructeur par défaut est utilisé.
    // Ici l'instance 'temporaire' de Gigantesque est convenablement définie.
    return temporaire ; }
} ;
```

Ceci ne manque pas d'inquiéter le programmeur soucieux d'efficacité. Dans le corps de la méthode ***bidule***, une première instance de l'objet ***Gigantesque*** est construite ; ceci est *a priori* coûteux. On a ensuite l'impression que l'exécution de l'instruction ***return*** provoque la création d'une seconde instance de l'objet ***Gigantesque*** à l'aide du constructeur de copie ; cette seconde instance étant la variable retournée par la méthode. En fait, le programmeur soucieux d'efficacité doit là être rassuré. Le compilateur se rend compte que l'objet temporaire qui est créé dans le corps de la méthode est destiné à être retourné par cette méthode. C'est donc cet objet qui est directement retourné par la méthode et non une copie superfétatoire de ce temporaire.

Pour autant, le calcul matriciel est un domaine compliqué. Les possibilités d'optimisation sont nombreuses et notre projet reste très inefficace. Je vous invite à utiliser une bibliothèque toute prête de calculs matriciels que l'on trouve librement sur Internet.

Il est simple de définir l'opérateur « **<<** » sur un flux pour une ***Matrice***. J'ai choisi de faire précéder le premier élément d'une accolade ouvrante, de séparer chaque ligne par un point-virgule et de faire suivre le dernier élément d'une accolade fermante. Dans cette fonction, pour désigner un élément de la matrice ***A***, on ne peut pas utiliser la méthode ***imprudent***. On est obligé d'utiliser l'opérateur ***operator()*** puisque l'on n'est pas dans la définition de l'objet ***Matrice***

2 Le programme

Ci après figure le programme complet de notre « GAUSS à nous ». Ce programme est essentiellement destiné à montrer les capacités du C++ à fournir un environnement de programmation spécialisé.

```
1  #include <iostream> // cin, cout et cerr
2
3  using namespace std ;
4
5  inline void
6  fatal(const string & message)
7  {
8    cerr << "Erreur fatale, message '" << message << "'." << endl ;
9    exit(1) ;
10 }
11
12 class Matrice {
13
14   const size_t lignes_ ;      // Nombre de lignes de la matrice.
15   const size_t colonnes_ ; // Nombre de colonnes de la matrice.
16   double * tableau ;          // Pointeur vers le début de l'emplacement alloué sur le tas.
17
18   // Les deux méthodes suivantes sont 'private', c'est-à-dire propres à la classe. On
19   // ne peut pas les appeler depuis l'extérieur de la classe. Les éléments de la matrice
```

```
// sont rangés en colonne comme en Fortran. Il serait ainsi possible d'utiliser des
// sous-programmes de bibliothèques écrites en Fortran.
// Référence (indices non contrôlés) à un élément de la matrice en lecture (référence
// const à une matrice déclarée const).
const double & imprudent(const size_t & i, const size_t & j) const {
  return tableau[j*lignes_ + i] ; }
// Référence (indices non contrôlés) à un élément de la matrice en lecture (référence
// non const à une matrice déclarée non const).
double & imprudent(const size_t & i, const size_t & j) {
  return tableau[j*lignes_ + i] ; }

public :
// Constructeur normal.
Matrice(const size_t & arg1, const size_t & arg2 = 1) :
  lignes_(arg1), colonnes_(arg2) {
  Matrice & A = *this ; // L'instance courante s'appelle 'A'.
  if ( (A.lignes() == 0) || (A.colonnes() == 0) ) {
    fatal("Les arguments du constructeur normal sont invalides") ; }
  // Allocation sur le tas, à l'aide de l'opérateur new, d'un emplacement de taille
  // suffisante. L'opérateur new retourne 0 en cas d'échec.
  if ( (A.tableau = new double[A.lignes()*A.colonnes()]) == 0 ) {
    fatal("Échec de l'allocation") ; }
  // Initialisation à zéro de la matrice.
  for ( size_t i = 0 ; i < A.lignes() ; ++ i ) {
    for ( size_t j = 0 ; j < A.colonnes() ; ++ j ) {
      A.imprudent(i, j) = 0. ; } }
}
// Constructeur de copie.
Matrice(const Matrice & B) : lignes_(B.lignes()), colonnes_(B.colonnes()) {
  Matrice & A = *this ; // L'instance courante s'appelle 'A'.
  // Allocation sur le tas.
  if ( (tableau = new double[A.lignes()*A.colonnes()]) == 0 ) {
    fatal("Échec de l'allocation") ; }
  // Recopie de la matrice B.
  for ( size_t i = 0 ; i < A.lignes() ; ++ i ) {
    for ( size_t j = 0 ; j < A.colonnes() ; ++ j ) {
      A.imprudent(i, j) = B.imprudent(i, j) ; } }
}
// Destructeur.
~Matrice() {
  if ( tableau == 0 ) { // Ceci ne devrait jamais arriver.
    fatal("Erreur interne à la bibliothèque, prévenir F. Legendre") ; }
  // La mémoire précédemment obtenue est retournée au système d'exploitation.
  delete [] tableau ;
}
// A.lignes() et A.colonnes() : nombre de lignes et de colonnes de A.
size_t lignes() const { return lignes_ ; }
size_t colonnes() const { return colonnes_ ; }

// Opérateur = ; affectation : A = B.
const Matrice & operator=(const Matrice & B) {
  Matrice & A = *this ; // L'instance courante s'appelle 'A'.
  if ( (A.lignes() != B.lignes()) || (A.colonnes() != B.colonnes()) ) {
    cerr << "A.lignes() = " << A.lignes() << endl ;
    cerr << "A.colonnes() = " << A.colonnes() << endl ;
    cerr << "B.lignes() = " << B.lignes() << endl ;
    cerr << "B.colonnes() = " << B.colonnes() << endl ;
    fatal("Formats invalides pour l'opérateur '='") ; }
  for ( size_t i = 0 ; i < A.lignes() ; ++ i ) {
    for ( size_t j = 0 ; j < A.colonnes() ; ++ j ) {
      A.imprudent(i, j) = B.imprudent(i, j) ; } }
  return A ; }

// Opérateur (..., ...) ; les indices, qui commencent en 0, sont contrôlés. C'est une
// référence sur un double qui est retournée : on peut ainsi écrire 'A(0, 0) = 12 ;'.
double & operator()(const size_t & i, const size_t & j = 0) {
  Matrice & A = *this ; // L'instance courante s'appelle 'A'.
  if ( (i >= A.lignes()) || (j >= A.colonnes()) ) {
    cerr << "i = " << i << " A.lignes() = " << A.lignes() << endl ;
    cerr << "j = " << j << " A.colonnes() = " << A.colonnes() << endl ;
    fatal("Les indices sont invalides") ; }
  return A.imprudent(i, j) ; }
// Même méthode mais qui retourne une référence const pour une const Matrice.
const double & operator()(const size_t & i, const size_t & j = 0) const {
  const Matrice & A = *this ; // L'instance courante s'appelle 'A'.
  if ( (i >= A.lignes()) || (j >= A.colonnes()) ) {
    cerr << "i = " << i << " A.lignes() = " << A.lignes() << endl ;
    cerr << "j = " << j << " A.colonnes() = " << A.colonnes() << endl ;
    fatal("Les indices sont invalides") ; }
  return A.imprudent(i, j) ; }

// Opérateur * ; multiplication matricielle : C = A*B.
Matrice operator*(const Matrice & B) const {
  const Matrice & A = *this ; // L'instance courante s'appelle 'A'.
  if (A.colonnes() != B.lignes()) {
    cerr << "A.colonnes() = " << A.colonnes() << endl ;
    cerr << "B.lignes() = " << B.lignes() << endl ;
    fatal("Formats invalides pour l'opérateur '*'") ; }
  // Création d'une matrice temporaire, C, pour enregistrer le résultat.
  Matrice C(A.lignes(), B.colonnes()) ;
```

```
110     for ( size_t i = 0 ; i < A.lignes() ; ++ i ) {
111       for ( size_t j = 0 ; j < B.colonnes() ; ++ j ) {
112         double somme = 0. ;
113         for ( size_t k = 0 ; k < A.colonnes() ; ++ k ) {
114           somme += A.imprudent(i, k) * B.imprudent(k, j) ; }
115         C.imprudent(i, j) = somme ; } }
116     return C ; }
117
118   // Opérateur % ; multiplication terme à terme : C = A%B.
119   Matrice operator%(const Matrice & B) const {
120     const Matrice & A = *this ; // L'instance courante s'appelle 'A'.
121     if ( (A.lignes() != B.lignes()) || (A.colonnes() != B.colonnes()) ) {
122       cerr << "A.lignes() = " << A.lignes() << endl ;
123       cerr << "A.colonnes() = " << A.colonnes() << endl ;
124       cerr << "B.lignes() = " << B.lignes() << endl ;
125       cerr << "B.colonnes() = " << B.colonnes() << endl ;
126       fatal("Formats invalides pour l'opérateur '%'") ; }
127     Matrice C(A.lignes(), A.colonnes()) ; // Temporaire pour enregistrer le résultat.
128     for ( size_t i = 0 ; i < A.lignes() ; ++ i ) {
129       for ( size_t j = 0 ; j < A.colonnes() ; ++ j ) {
130         C.imprudent(i, j) = A.imprudent(i, j) * B.imprudent(i, j) ; } }
131     return C ; }
132
133   // Opérateur ! ; transposition : B = !A.
134   Matrice operator!() const {
135     const Matrice & A = *this ;              // L'instance courante s'appelle 'A'.
136     Matrice B(A.colonnes(), A.lignes()) ; // Temporaire pour enregistrer le résultat.
137     for ( size_t i = 0 ; i < A.lignes() ; ++ i ) {
138       for ( size_t j = 0 ; j < A.colonnes() ; ++ j ) {
139         B.imprudent(j, i) = A.imprudent(i, j) ; } }
140     return B ; }
141
142 } ;
143
144 // Représentation externe d'une matrice sur un flux de sortie. La matrice est affichée
145 // par ligne alors qu'elle est enregistrée par colonne. Le caractère ';' sépare les dif-
146 // férentes lignes. La suite de nombres est encadrée par les caractères '{' et '}'.
147 ostream & operator<<(ostream & arg1, const Matrice & A)
148 {
149   arg1 << "{ " ;
150   for ( size_t i = 0 ; i < A.lignes() ; ++ i ) {
151     for ( size_t j = 0 ; j < A.colonnes() ; ++ j ) {
152       arg1 << A(i, j) << ' ' ; }
153     if ( i != (A.lignes()-1) ) {
154       arg1 << "; " ; } }
155   arg1 << '}' ;
156   return arg1 ;
157 }
158
159 Matrice
160 inv(const Matrice & A)
161 {
162   if ( A.lignes() != A.colonnes() ) {
163     cerr << " A.lignes() = " << A.lignes() << endl ;
164     cerr << " A.colonnes() = " << A.colonnes() << endl ;
165     fatal("La matrice n'est pas carrée dans 'inv(...)'") ; }
166   Matrice B(A.lignes(), A.lignes()) ;
167   double det ;
168   switch ( A.lignes() ) {
169
170     case 1 : B(0, 0) = 1 / A(0, 0) ;
171     break ;
172
173     case 2 : det = A(0, 0)*A(1, 1) - A(1, 0)*A(0, 1) ;
174     B(0, 0) = A(1, 1) / det ;
175     B(1, 1) = A(0, 0) / det ;
176     B(1, 0) = -A(1, 0) / det ;
177     B(0, 1) = -A(0, 1) / det ;
178     break ;
179
180     default : cout << "L'inversion d'une matrice de taille > 2 n'est pas encore "
181       "implémentée, se plaindre auprès de F. Legendre" << endl ;
182     break ;
183   }
184   return B ;
185 }
186
187
188 int
189 main()
190 {
191   Matrice A(2, 3) ; // Constructeur normal.
192   // Pour montrer que les éléments de la matrice sont initialisés à 0.
193   cout << "A = " << A << endl ;
194   // Affectation individuelle des éléments de la matrice.
195   A(0, 0) = 1 ; A(1, 0) = 2 ; A(0, 1) = 3 ; A(1, 1) = 4 ;  A(0, 2) = 5 ; A(1, 2) = 6 ;
196   // Impression de contrôle.
197   cout << "A = " << A << endl ;
198
199   Matrice B(A) ; // Constructeur de copie.
```

```
200   cout << "B = " << B << endl ;
201   cout << "A%B = " << A%B << endl ; // Test de l'opérateur '%'.
202   cout << "!A = " << !A << endl ;    // Test de l'opérateur '!'.
203   Matrice C(!A) ;                   // Constructeur de copie à partir d'un temporaire.
204   cout << "A*C = " << A*C << endl ;  // Test de l'opérateur '*'.
205
206   // Test de l'opérateur d'affectation.
207   Matrice D(2, 3) ;
208   Matrice E(2, 3) ;
209   E = D = A%B ;
210   cout << "D = " << D << endl ;
211   cout << "E = " << E << endl ;
212
213   // Test de l'inversion des matrices.
214   Matrice F(A*!A) ;
215   cout << "F*inv(F) = " << F*inv(F) << endl ;
216   cout << "inv(F)*F = " << inv(F)*F << endl ;
217
218   // Programmation des moindres carrés ordinaires.
219   Matrice X(5, 2) ;
220   X(0, 0) = -1 ; X(0, 1) = 1 ;
221   X(1, 0) = 3 ; X(1, 1) = 1 ;
222   X(2, 0) = -4 ; X(2, 1) = 1 ;
223   X(3, 0) = 2 ; X(3, 1) = 1 ;
224   X(4, 0) = -1 ; X(4, 1) = 1 ;
225   Matrice y(5) ;
226   for ( size_t i = 0 ; i < X.lignes() ; ++ i ) {
227     y(i) = 3*X(i, 0) + 4 + i%2-.5 ; } // 'i%2-.5' est égal à -.5, .5, -.5, ...
228   Matrice a_chapeau( inv(!X*X) * (!X*y) ) ;
229   cout << "â = " << a_chapeau << endl ;
230
231   return 0 ; // Retour au système d'exploitation avec le code '0'.
232 }
```

L'exécution de ce programme conduit à la sortie suivante.

```
1  A = { 0 0 0 ; 0 0 0 }
2  A = { 1 3 5 ; 2 4 6 }
3  B = { 1 3 5 ; 2 4 6 }
4  A%B = { 1 9 25 ; 4 16 36 }
5  !A = { 1 2 ; 3 4 ; 5 6 }
6  A*C = { 35 44 ; 44 56 }
7  D = { 1 9 25 ; 4 16 36 }
8  E = { 1 9 25 ; 4 16 36 }
9  F*inv(F) = { 1 6.21725e-015 ; 8.88178e-015 1 }
10 inv(F)*F = { 1 8.88178e-015 ; 6.21725e-015 1 }
11 â = { 3.17532 ; 3.93506 }
```