

DM2 MP2I : Pivot de Gauss

Consignes

Il y a 5 fichiers sources `affiche.c`, `solveT.c`, `transvection.c`, `triangulariser.c` et `synthese.c` à rendre pour ce devoir. Chacun contient un certain nombre de fonctions à écrire mais aucun ne possède de main. L'archive du devoir contient les squelettes de ces fichiers ainsi que le fichier d'en-tête `bibli.h`. Une fois complétés, ces fichiers `.c` sont placés dans une archive `dm2.zip` à déposer sur [cahier de prépa](#).

En CPGE, il est d'usage que les variables soient écrites dans des polices de caractères différentes. Par exemple une matrice A est notée A dans un contexte de code et A dans une preuve mathématique.

- Les fichiers sources `affiche.c`, `solveT.c`, `triangulariser.c` et `synthese.c` sont pré-remplis. Par défaut, une valeur de retour aberrante est donnée dans le corps de chaque fonction. Si vous n'arrivez pas à écrire le code d'une fonction, **laissez le code par défaut**. De la sorte, votre fichier compilera et le script de correction pourra évaluer les autres fonctions.

Si un de vos fichiers source ne compile pas, aucune des fonctions qu'il contient n'est évaluée et **la note pour ce fichier est nulle**.

- Les fichiers sont conçus pour être compilés séparément. Dans un prochain cours, nous parlerons de la compilation séparée plus en détail. Voici l'essentiel à savoir pour ce devoir : tous vos fichiers importent le fichier d'en-tête `bibli.h` et rien d'autre. La directive de préprocessing

`# include "bibli.h"` est donc la seule inclusion autorisée dans vos fichiers.

- La taille des matrices est indiquée dans le fichier `bibli.h` par une directive `# define N 4`. Il s'agit de la seule directive de définition autorisée (mais il est toujours possible de changer N en 4 en 3 ou 5 pour les tests). En particulier, aucun autre fichier de votre projet ne doit contenir une définition de N .

- Le plus souvent, une fonction f demandée est suivie d'une fonction

`int cpx_f(void)` qui indique l'ordre de grandeur de sa complexité. On veut majorer ici, en fonction de N , le nombre d'opérations arithmétiques et de comparaisons que réalise la fonction lorsqu'on l'appelle. En cas de doute, on considère toujours le pire cas pour la complexité.

Les complexités des algorithmes de ce devoir étant toutes polynômiales en fonction du nombre de lignes N des matrices et vecteurs, il s'agit ici de donner le degré du polynôme qui exprime la complexité de f . Si on pense que la fonction a une complexité quadratique, on écrit

`int cpx_f(void){return 2}`. Si, au contraire, on pense que le nombre d'opérations effectuées lors de l'appel à f est constant, alors on écrit `int cpx_f(void){return 0}`.

- Vos fichiers ne doivent comporter aucun `main`. Pour tester les fonctions contenues dans `solveT`, créez un fichier `solveT_bench.c` dont voici l'allure générale :

```

1 #include bibli.h"
2
3 void bench_trig_inversible(void){
4     matrix mat;
5     for (int i=0; i<N; i++){
6         for (int j=0; j<N; j++){
7             if (j<i) mat[i][j]=0.;
8             else mat[i][j]=i*N+j+1;
9         }
10    affiche_mat(mat);
11    printf(" trig_inversible (mat)=%d\n",trig_inversible(mat));
12 }
13
14 int main(void){
15     bench_trig_inversible();
16     retrn 0;
17 }
18

```

Pour compiler ce fichier, procéder en trois temps :

- Construire, si ce n'est pas déjà fait, le fichier objet `affiche.o` qui contient les commandes d'affichage.
- Construire les 2 fichiers objets `solveT.o` et `solveT_bench.o` :

```
$ gcc -c solveT.c
$ gcc -c solveT_bench.c
```

- Pratiquer l'édition de liens :

```
$ gcc affiche.o solveT.o solveT_bench.o -o solveT_bench.out
```

Présentation

On considère un système de n équations à n inconnues de la forme suivante :

$$\begin{cases} a_{1,1}x_1 + a_{1,2}x_2 + \dots + a_{1,n}x_n = b_1 \\ a_{2,1}x_1 + a_{2,2}x_2 + \dots + a_{2,n}x_n = b_2 \\ \vdots + \vdots + \vdots + \vdots = \vdots \\ a_{n,1}x_1 + a_{n,2}x_2 + \dots + a_{n,n}x_n = b_n \end{cases}$$

On lui associe sa représentation matricielle : $AX = B$ avec

$$A = \begin{pmatrix} a_{1,1} & a_{1,2} & \dots & a_{1,n} \\ a_{2,1} & a_{2,2} & \dots & a_{2,n} \\ \vdots & & \ddots & \vdots \\ a_{n,1} & a_{n,2} & \dots & a_{n,n} \end{pmatrix}, X = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} \text{ et } B = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix}$$

On dit qu'une matrice est *triangulaire supérieure* si les coefficients en-dessous de la diagonale sont tous nuls. Par exemple, $\begin{pmatrix} 3 & 4 & 5 & 1 \\ 0 & -1 & 0 & 2 \\ 0 & 0 & -4 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}$ est triangulaire supérieure.

On dit qu'un système est sous forme *triangulaire supérieure* si sa matrice associée est triangulaire supérieure. L'allure d'un tel système est donc la suivante :

$$\begin{cases} a_{1,1}x_1 + a_{1,2}x_2 + \dots + a_{1,n}x_n = b_1 \\ a_{2,2}x_2 + \dots + a_{2,n}x_n = b_2 \\ \vdots \\ (0) \quad \ddots \quad \vdots = \vdots \\ a_{n,n}x_n = b_n \end{cases}$$

Par théorème, un système triangulaire supérieur a une solution unique si et seulement si tous les coefficients diagonaux de la matrice sont non nuls.

L'objectif de ce problème est de mettre le système sous forme *triangulaire supérieure* avant de le résoudre.

Pour ce devoir, on se donne deux types `matrix` et `vector` :

```
1 # define N 3
2
3 typedef double matrix [N][N];
4 typedef double vector [N];
```

Avec ce réglage, on travaille par défaut en dimension 3 mais il suffit de modifier `N` pour changer cela. Par exemple `# define N 5` fait travailler avec des matrices 5×5 .

Le mot clé `typedef` est utilisé pour définir un alias de type : une fonction comme `bool triangulariser(matrix a, vector b);` prend en paramètres une matrice et un vecteur sans qu'il soit besoin de préciser à `gcc` qu'il s'agit d'une matrice $N \times N$ et d'un vecteur à N coefficients.

1 Affichage

Dans le fichier `affiche.c`, toutes les fonctions d'affichage effectuent un dernier changement de ligne avant de terminer. Le spécifieur de format imposé pour les flottants est `%5.2f`

Q.1 Écrire la fonction `void affiche_mat(matrix mat);` qui affiche le contenu d'une matrice.

```
1 assert (N==3); // réglage : # define N 4
2 matrix a = {{1.,2.,0.},{1.,0.,-1.},{2.,1.,5.}} ;
3 affiche_mat(a);
```

Listing 1 – code C

Listing 2 – rendu sur console

```
| 1.00  2.00  0.00 |
| 1.00  0.00 -1.00 |
| 2.00  1.00  5.00 |
```

Q.2 Écrire la fonction `void affiche_syst(matrix mat,vector v);` qui affiche un système sous la forme suivante :

```
7 assert (N==4); // réglage : # define N 4
8 matrix a = {{1.,2.,0., 8.},
9             {2.,1.,-1.,10.},
10            {-1.,5.,3.,2.},
11            {1.,4.,2.,1.}} ;
12 vector b = {1.,0.,-1,2.};
13 affiche_syst(a,b);
```

Listing 3 – code C

Listing 4 – rendu sur console

```
| 1.00  2.00  0.00  8.00 | , | 1.00 |
| 2.00  1.00 -1.00 10.00 | , | 0.00 |
| -1.00  5.00  3.00  2.00 | , | -1.00 |
| 1.00  4.00  2.00  1.00 | , | 2.00 |
```

2 Système triangulaire

Q.3 Écrire la fonction

`bool trig_inversible(matrix M)` qui renvoie le booléen `true` si la matrice M est triangulaire supérieure et inversible.

Q.4 Compléter avec

`int cpx_trig_inversible(void)`; qui donne la complexité de votre fonction en fonction de N .

Q.5 Résolution de système triangulaire :

(a) Écrire la fonction

`void solveT(matrix A, vector B, vector R)` qui prend en paramètres une matrice A et deux vecteurs B et R . La fonction remplit le vecteur R qui est la solution au système (supposé triangulaire supérieur à solution unique) $AR = B$.

La matrice A est supposée triangulaire inversible, on ne demande pas de vérifier ce fait.

```
18  assert (N==3);
19  matrix a =
    { {2.,3.,2.},{0.,-1.,5.},{0.,0.,2.} };
20  vector b = {4.,6.,2.};
21  vector r;
22  solveT(a,b,r);
23  affiche_vect(r);
```

Listing 5 – code C

Listing 6 – rendu sur console

```
| 2.50 |
| -1.00 |
| 1.00 |
```

(b) Écrire le fonction

`int cpx_solveT(void)`; qui donne la complexité de votre fonction en fonction de N .

3 Triangularisation

Il faut maintenant mettre un système quelconque sous forme triangulaire supérieure. Cette opération est réalisée par des *transvections* successives notées $L_i \leftarrow L_i + \lambda L_j$. Il s'agit d'ajouter à la ligne courante i un certain nombre de fois la ligne j de façon à annuler le coefficient $a_{i,j}$.

3.1 Transvections

Les fonctions de cette section sont à écrire dans `transvections.c`.

Q.6 Transvections :

(a) Écrire la fonction

`void transvection_v(vector V, int i, int j, double lm)` qui réalise la transvection $L_i \leftarrow L_i + \lambda L_j$ sur le vecteur V .

```

27     vector v = {1.,2.,3.,5.};
28     printf("le vecteur v avant transvection\n");
29     affiche_vect(v);
30     printf(" L_2 <- L_2 -2L_0\n");
31     transvection_v(v,2,0,-2); // L_2 <- L_2
                                -2L_0
32     printf("le vecteur v après transvection\n");
33     affiche_vect(v);

```

Listing 7 – code C

Listing 8 – rendu sur console

```

le vecteur v avant transvection :
| 1.00|
| 2.00|
| 3.00|
| 5.00|

L_2 <- L_2 -2L_0
le vecteur v après transvection :
| 1.00|
| 2.00|
| 1.00|
| 5.00|

```

(b) Donner la complexité de votre fonction en écrivant `int cpx_transvection_m(void)`;

(c) Écrire la fonction

`void transvection_m(matrix A, int i, int j, double lm)` qui réalise la transvection $L_i \leftarrow L_i + \lambda L_j$ sur la matrice A .

```

46     assert (N==3);
47     matrix a =
        {{1.,2.,0.},{1.,0.,-1.},{2.,1.,5.}};
48     printf("Avant transvection :\n");
49     affiche_mat(a);
50     // L_2 <- L_2 -2L_0
51     transvection_m(a,2,0,-2);
52     printf("Après transvection :\n");
53     affiche_mat(a);

```

Listing 9 – code C

Listing 10 – rendu sur console

```

Avant transvection :
| 1.00 2.00 0.00|
| 1.00 0.00 -1.00|
| 2.00 1.00 5.00|

Après transvection :
| 1.00 2.00 0.00|
| 1.00 0.00 -1.00|
| 0.00 -3.00 5.00|

```

(d) Donner la complexité de votre fonction en écrivant `int cpx_transvection_m(void)`;

On se donne deux fonctions outils pour réaliser des échanges de lignes :

Q.7 Écrire la fonction

`void swap_v(vector v, int i, int j)` qui échange les lignes i et j du vecteur v .

Donner la complexité de votre fonction

Q.8 Écrire la fonction

`void swap_m(matrix A, int i, int j)` qui échange les lignes i et j de la matrice A .

Donner la complexité de votre fonction.

Recherche du pivot

On écrit les fonctions suivantes dans `traingulariser.c`.

Lorsqu'on veut éliminer la i -ème variable, il faut chercher un *pivot*, c'est à dire un coefficient de cette variable dans une des équations entre la ligne i et la dernière.

Ainsi, pour éliminer la variable numéro 6, on cherche un pivot dans les lignes 6,7,8,9...

On pourrait bien sûr se contenter de prendre comme pivot le coefficient de la variable x_i ligne i , mais on préfère chercher le plus grand coefficient (en valeur absolue) de x_i dans toutes les lignes après i . On fait cela pour éviter les divisions par des tous petits nombres en valeur absolue.

En effet, diviser par un petit nombre positif, c'est obtenir un résultat avec une grande valeur absolue. Or, plus la valeur absolue du résultat à virgule flottante est grande moins la précision est bonne.

Par exemple pour éliminer la variable y (qui porte le numéro 2),

$$\begin{pmatrix} 1 & 10 & 6 & 7 \\ 0 & 2 & 5 & 1 \\ 0 & 1 & 1 & 1 \\ 0 & -3 & -5 & -8 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ t \end{pmatrix} = \begin{pmatrix} 5 \\ 0 \\ 2 \\ -1 \end{pmatrix},$$

On choisit -3 comme pivot puisque c'est le coefficient de y qui a la plus grande valeur absolue après la ligne 2. Il faudra donc échanger L_4 et L_2 .

Q.9 Écrire la fonction

int pivot(matrix A, **int** i) qui renvoie le numéro de ligne (entre i et $N - 1$) où le coefficient de la variable x_i a la plus grande valeur absolue.

En commentaire, juste en dessous du code de la fonction, donner la complexité de votre fonction (en nombre d'opérations arithmétiques et de comparaisons) en fonction de N et i . Il s'agit de donner un polynôme à deux variables N et i , qui, multiplié par une constante positive, majore le nombre d'opérations.

```
63  assert (N==4); // penser à mettre #
    define N 4
64  matrix a =
        {{1.,20.,0.,8.},{0.,1.,-1.,10.},
        {0.,-5.,3.,2.},{0.,4.,2.,1.}};
65  affiche_mat(a);
66  printf("pivot(a,1)=%d\n",pivot(a,1));
67
```

Listing 11 – code C

Listing 12 – rendu sur console

```
| 1.00 20.00 0.00 8.00 |
| 0.00 1.00 -1.00 10.00 |
| 0.00 -5.00 3.00 2.00 |
| 0.00 4.00 2.00 1.00 |
```

```
pivot(a,1)=2
```

L'algorithme de mise sous forme triangulaire consiste à parcourir chaque colonne i en cherchant à chaque fois le meilleur pivot, à faire les échanges correspondants et à appliquer toutes les transvections pour que la colonne i soit constituée de 0 à partir de la ligne $i + 1$. **Bien sûr, toute action sur la matrice du système doit être répercutée sur le vecteur des seconds membres.**

Q.10 Écrire la fonction

bool triangulariser(matrix a, vector b) qui modifie la matrice A et le vecteur B de sorte que le système $AX = B$ devienne, si c'est possible, triangulaire supérieur.

La fonction renvoie un booléen qui vaut vrai si cette tâche a pu être menée à bien et faux sinon (donc si A n'est pas inversible). On peut montrer que la non inversibilité de A est équivalente à la découverte d'un pivot nul lors d'une des étapes de l'algorithme.

```

74  assert (N==4); // # define N 4
75  matrix a = {{1.,2.,0., 8.},
76             {2.,1.,-1.,10.},
77             {-1.,5.,3.,2.},
78             {1.,4.,2.,1.}};
79  vector b = {1.,0.,-1.,2.};
80
81  printf("Avant triangul. :\n");
82  affiche_syst(a,b);
83  printf("triangul.(a,b) :%d\n",
84         triangulariser(a,b));
85  printf("Après triangul. :\n");
86  affiche_syst(a,b);
87
88  //pivot nul colonne 0 :
89  a[0][0]=0.; a[1][0]=0.;
90  a[2][1]=0.; a[3][0]=0.;
91  printf("Avant triangul. :\n");
92  affiche_syst(a,b);
93  printf("triangul.(a,b) :%d\n",
94         triangulariser(a,b));

```

Listing 13 – code C

Listing 14 – rendu sur console

```

Avant triangul. :
| 1.00  2.00  0.00  8.00|, | 1.00|
| 2.00  1.00 -1.00 10.00|, | 0.00|
|-1.00  5.00  3.00  2.00|, |-1.00|
| 1.00  4.00  2.00  1.00|, | 2.00|

triangul.(a,b) :1
Après triangul. :
| 2.00  1.00 -1.00 10.00|, | 0.00|
| 0.00  5.50  2.50  7.00|, |-1.00|
| 0.00  0.00  0.91 -8.45|, | 2.64|
| 0.00  0.00  0.00 -0.60|, | 1.80|

Avant triangul. :
| 0.00  1.00 -1.00 10.00|, | 0.00|
| 0.00  5.50  2.50  7.00|, |-1.00|
| 0.00  0.00  0.91 -8.45|, | 2.64|
| 0.00  0.00  0.00 -0.60|, | 1.80|

triangul.(a,b) :0

```

Q.11 Écrire la fonction

`int` cpx_triangulariser(**void**); pour exprimer la complexité de la fonction précédente.

3.2 Synthèse

On écrit les fonctions suivantes dans `synthese.c`.

Q.12 Écrire la fonction

`bool` resoudre(matrix a, vector b, vector r) qui prend deux vecteurs et une matrice en paramètres. La fonction retourne vrai si le système $AX = B$ possède une solution unique et, dans ce cas, met la solution dans `r`.

```

115  // penser à écrire #define N 4
116  assert (N==4);
117  matrix a = {{1.,2.,0.,8.},
118             {2.,1.,-1.,10.},
119             {-1.,5.,3.,2.},
120             {1.,4.,2.,1.}};
121  vector b = {1.,0.,-1.,2.};
122  vector r;
123  affiche_syst(a,b);
124  printf("resoudre(a,b,r)=%d\n",
125         resoudre(a,b,r));
125  affiche_vect(r);

```

Listing 15 – code C

Listing 16 – rendu sur console

```

| 1.00  2.00  0.00  8.00|, | 1.00|
| 2.00  1.00 -1.00 10.00|, | 0.00|
|-1.00  5.00  3.00  2.00|, |-1.00|
| 1.00  4.00  2.00  1.00|, | 2.00|

resoudre(a,b,r)=1
|-5.00|
|15.00|
|-25.00|
|-3.00|

```