Décomposition d'algorithme en FORTRAN 95

Nicolas Depauw

4 juillet 2011

Dans ce deuxième exemple, nous implantons la décomposition LU pour la résolution de systèmes linéaires $Ax = b$. Nous manipulons donc des tableaux. Nous montrons aussi comment décomposer un grand programmme en plusieurs sous-programmes. Certains d'entre eux pouvant servir ailleurs, nous les regroupons dans un *module*. En plaçant le module et le programme qui l'utilise dans deux fichiers distincts, nous montrons comment compiler des applications composées de plusieurs fichiers sources, de façon éfficace grâce à la commande `make`.

1 La décomposition LU

Soit A une matrice dans $\mathbb{R}^{n\times n}$. Nous cherchons une matrice de permutation P (associée à la permutation π), une matrice triangulaire supérieure U et une matrice triangulaire inférieure avec des 1 sur la diagonale, toutes de même taille que A, telles que $LUP = A$.

Nous choisissons de multiplier par P à droite, ce qui correspond à une permutation des colonnes de A ainsi que des inconnues x, plutôt qu'à gauche, ce qui correspondrait à permuter les lignes de A, donc les équations (et le second membre b). En effet, en FORTRAN 95, les matrices sont stoquées colonnes par colonnes. Ainsi une permutation effective des colonnes consisterait à un échange de blocs de mémoires (contigues).

1.1 Entrées-sorties

Nous allons écrire un sous-programme. Comme un programme, il contient des déclarations et des instructions. Mais d'une part on peut l'appeler depuis un autre programme ou sous-programme (qu'on qualifiera de sur-programme pour le distinguer du sous-programme) ; et d'autre part, on peut lui fournir des arguments. En FORTRAN 95, c'est une `subroutine`. Les arguments sont passés par adresse. Pendant l'exécution du sous-progamme, les valeurs des variables arguments peuvent être lues (in) et modifiées. De sorte que le programme appelant peut ensuite lire les nouvelles valeurs des arguments (out).

Pour notre problème, les arguments du sous-programme sont le tableau `m` à deux indices, qui en entrée contient A et en sortie contient les coefficients utiles de L et U, et le tableau `trans` à un indice, qui en sortie contient de quoi nous permettre d'appliquer facilement la permutation π ainsi que son inverse. En plus du type, nous indiquons lors des déclarations, à l'aide de l'attribut `dimension`, que nous avons à faire à une matrice et un vecteur. Et pour les arguments, à l'aide de l'attribut `intent`, s'il s'agit d'entrée (`in`, lue seulement), de sortie (`out` pas lue avant d'être écrite) ou bien des deux simultanément.

La fonction `size(m,2)` renvoie le nombre de colonnes de `m` (alors que `size(m,1)` renverrai le nombre de lignes, et que `size(m)` renverrai le vecteur (`size(m,1)`, `size(m,2)`). Noter le caractère `!`

pour *commenter* le reste de la ligne : ce qui se trouve entre le `!` et la fin de la ligne n'est simplement pas lu par le compilateur.

1 ⟨*Décompositon* $A = LUP$ 1⟩≡ (9) 2▷

```
subroutine aelup(m,trans)
  !   entree : m de taille nxn contenant A
  !   sortie : m contenant (les coefs utiles de) L et U
  !            trans contenant la suite des (n-1) transpositions de colonnes
  !                  correspondant a la permutation de matrice P
  !   aelup calcul la decomposition A=LUP avec L triangulaire inferieure
  !   avec des 1 sur la diagonale, U triangulaire superieure, P de permutation
 real, dimension(:,:), intent(inout) :: m
 integer, dimension(size(m,2)-1), intent(out) :: trans
```

1.2 Initialisation

Nous aurons besoin de `n` pour la taille de la matrice, et de `perm` pour la permutation π des indices des colonnes. Nous aurons aussi besoins de quelques autres variables de type `integer` et `real` que nous déclarons maintenant. La construction `(/ ... /)` permet d'écrire un vecteur explicite, avec des coordonnées séparées par des virgules. La constrution `(...,i=i_0,i_n)` crée une suite de nombres séparés par une virgule qui se calculent avec ce qui remplacent les trois petits points à l'aide d'un indice comme dans une boucle `do...end do` : ici l'indice est `i` et il varie de `i_0` à `i_n`. Le reste de l'algorithme est une boucle répétée $n-1$ fois (formellement, on verra qu'il y a n étapes, mais la dernière ne contient aucune instruction).

2 ⟨*Décompositon* $A = LUP$ 1⟩+≡ (9) ◁1

```
integer, dimension(size(m,2)) :: perm
integer :: n,j,k,ind_piv
real :: r,piv
n=size(m,1)      ! m doit etre carree
perm=(/(k,k=1,n)/) ! initialise la permutation a l'identite : perm(k)=k
do k=1,n-1
   ⟨pivotage dans la ligne 3⟩
   ⟨calculs 4⟩
end do
end subroutine aelup
```

1.3 Pivotage partiel

En vue minimiser les erreurs de calculs liées à la représentation en machine des nombres réels, on choisit à l'étape k de chercher dans la ligne k quelle colonne porte le plus grand élément (en valeur absolue). Ensuite, plutôt que d'échanger les contenus des blocs mémoire de la matrice, on échange les éléments de `perm`. Ainsi `perm(k)` est l'indice de la colonne qui porterait le numéro k si l'on avait effectué les permutations de colonnes. Le coefficient $M(i,j)$ d'indices (i,j) de la matrice M permutée est donc `m(i,perm(j))`.

3 ⟨*pivotage dans la ligne 3*⟩≡ (2)

```
ind_piv=k
piv=abs(m(k,perm(ind_piv)))
do j=k+1,n
   r=abs(m(k,perm(j)))
   if (r>piv) then
      ind_piv=j
      piv=r
   end if
end do
trans(k)=ind_piv        ! enregistre l'indice du j-eme pivot
if (ind_piv/=k) then    ! echange perm(k) et perm(ind_piv)
   j=perm(k);perm(k)=perm(ind_piv);perm(ind_piv)=j
end if
```

1.4 Élimination de Gauss

Pour justifier l'algorithme (sans pivotage pour simplifier), donnons un invariant. On va fabriquer une suite de trois matrices carrées $(A_k, L_k, U_k)_{k=1}^{n+1}$ avec pour tout k l'égalité $A_k + L_k U_k = A$; au début $A_1 = A$, $L_1 = 0$ et $U_1 = 0$, et à la fin $A_{n+1} = 0$, $U_{n+1} = U$ et $L_{n+1} = L$. À chaque étape, les coefficients non triviaux des trois matrices sont stockés dans `m` : juste avant l'étape k (la k-ème boucle), `m(i,j)` est égal à

- $A_k(i,j)$ si $\min(i,j,k) = k$,
- $U_k(i,j)$ si $\min(i,j,k) = i < k$,
- $L_k(i,j)$ si $\min(i,j,k) = j < \min(i,k)$.

Les autres coefficients des trois matrices sont nuls, sauf $L_k(i,j) = 1$ si $i = j < k$. Donc $(A_k + L_k U_k)(i,j) = A_k(i,j) + \sum_{l=1}^{\min(i,j,k-1)} L_k(i,l) U_k(l,j)$.

Admettons qu'en passant de k à $k+1$, on obtient successivement

1. $U_{k+1}(i,j) = U_k(i,j)$ pour tout les (i,j) sauf $U_{k+1}(i,j) = A_k(i,j)$ pour $k = i \leq j$,
2. $L_{k+1}(i,j) = L_k(i,j)$ pour tous les (i,j) sauf $L_{k+1}(i,j) = \frac{A_k(i,j)}{A_k(k,k)}$ pour $k = j \leq i$,
3. $A_{k+1}(i,j) = A_k(i,j) - [\min(i,j,k) = k] \frac{A_k(i,k) A_k(k,j)}{A_k(k,k)}$.

On en déduit

$$(A_{k+1}+L_{k+1}U_{k+1})(i,j)=A_{k+1}(i,j)+\sum_{l=1}^{\min(i,j,k)}L_{k+1}(i,l)U_{k+1}(l,j)$$

$$=A_k(i,j)-[\min(i,j,k)=k]\frac{A_k(i,k)A_k(k,j)}{A_k(k,k)}$$

$$+\sum_{l=1}^{\min(i,j,k-1)}L_k(i,l)U_k(l,j)+[\min(i,j,k)=k]\frac{A_k(i,k)}{A_k(k,k)}A_k(k,j)$$

$$=(A_k+L_kU_k)(i,j).$$

C'est la preuve que l'algorithme est correct.

4 ⟨*calculs* 4⟩≡ (2)

```
m(k+1:n,perm(k))=m(k+1:n,perm(k))/m(k,perm(k))
do j=k+1,n
   m(k+1:n,perm(j))=m(k+1:n,perm(j))-m(k+1:n,perm(k))*m(k,perm(j))
end do
```

2 La résolution $LUPx = b$

On va résoudre successivement les triangulaires $Lz = b$ et $Uy = z$, puis $Px = y$.

2.1 Entrées-sorties

On procède à une résolution en place, c'est à dire que le même tableau `b` contient en entrée le vecteur b, et en sortie le vecteur x. Les coefficients utiles de L et U sont dans le tableau `m`, tandis que le tableau `trans` contient la suite des transpositions de colonnes opérées lors de la décomposition $A = LUP$: à l'étape `k`, les colonnes d'indice `k` et `trans(k)` ont été (virtuellement) échangées.

5 ⟨*Résolution* $LUPx = b$ 5⟩≡ (9) 6▷

```
subroutine lupxeb(m,trans,b)
  real, dimension(:,:), intent(in) :: m
  integer, dimension(size(m,2)-1), intent(out) :: trans
  real, dimension(size(m,1)), intent(inout) :: b
  integer, dimension(size(m,2)) :: perm
  integer :: n,k,j,ind_piv
  real :: r
  n=size(m,1)      ! m doit etre carree
```

2.2 Initialisation

Il s'agit de restaurer la permutation.

6 ⟨*Résolution* $LUPx = b$ 5⟩+≡ (9) ◁5 7▷

```
perm=(/(k,k=1,n)/)
do k=1,n-1
   ind_piv=trans(k)       ! enregistre l'indice du k-eme pivot
   if (ind_piv/=k) then   ! echange perm(k) et perm(ind_piv)
      j=perm(k);perm(k)=perm(ind_piv);perm(ind_piv)=j
   end if
end do
```

2.3 Une descente, une remontée, une permutation

Après la première boucle, b contient le vecteur z. Après la seconde, il contient le vecteur y. Enfin on permute ses éléments pour restaurer x.

7 ⟨*Résolution* $LUPx = b$ 5⟩+≡ (9) ◁6

```
! resolution Lz=b
do k=2,n
   b(k:n)=b(k:n)-m(k:n,perm(k-1))*b(k-1)
end do
! resolution Uy=z
b(n)=b(n)/m(n,perm(n))
do k=n-1,1,-1
   b(1:k)=b(1:k)-m(1:k,perm(k+1))*b(k+1)
   b(k)=b(k)/m(k,perm(k))
end do
! resolution de Px=y
do k=n-1,1,-1
   ind_piv=trans(k)          ! enregistre l'indice du k-eme pivot
   if (ind_piv/=k) then      ! echange b(k) et b(ind_piv)
      r=b(k);b(k)=b(ind_piv);b(ind_piv)=r
   end if
end do
end subroutine lupxeb
```

3 Produit matrice vecteur

Nous donnons ici un exemple de fonction, qui prend en entrée une matrice et un vecteur, et renvoie en sortie le vecteur résultant de leur produit. Dans une fonction, le résultat est une variable qu'il faut déclarer et qui porte le même nom que la fonction.

Notre fonction `pmv` est un cas particulier de la fonction prédéfinie `matmul`. `matmul` multiplie deux matrices, ou une matrice et un vecteur (ou un vecteur et une matrice). Il est souvent plus efficace de reprogrammer à la main `matmul` pour le cas particulier que l'on considère.

8 ⟨*Produit matrice vecteur* 8⟩≡ (9)

```
function pmv(a,x)
  real, dimension(:,:), intent(in) :: a
  real, dimension(size(a,2)), intent(in) :: x
  real, dimension(size(a,1)) :: pmv
  integer :: n,j
  pmv=0.0
  n=size(a,2)
  do j=1,n
     pmv=pmv+a(:,j)*x(j)
  end do
end function pmv
```

4 Le module dans un fichier séparé

Un `module` en FORTRAN 95 est une unité de compilation qui, à la différence d'un programme, n'engendre pas directement un exécutable, mais plutôt une librairie destinée à être insérée dans d'autres programmes. Un module contient une section de déclarations où l'on pourrait déclarer des variables, et surtout une section, introduite par `contains` qui contient les sous-programmes (`subroutine` et `function`).

9 ⟨*aln.f95* 9⟩≡

```
module aln
  implicit none
contains
  ⟨Décompositon A = LUP 1⟩
  ⟨Résolution LUPx = b 5⟩
  ⟨Produit matrice vecteur 8⟩
end module aln
```

5 Le programme test

Pour tester nos sous-programmes de résolution, nous allons résoudre un système linéaire $Ax = b$, pour un $b = Ax_0$ fabriqué à partir de x_0. Un indicateur de qualité de notre algorithme sera la norme de l'erreur $e = \|x - x_0\|_\infty$, que l'on espère très petit.

5.1 Vue d'ensemble

Nous effectuons en réalité deux tests, pour deux matrices différentes. Pour éviter de taper deux fois les mêmes instructions, nous utilisons un sous-programme interne. Le programme présente donc une section `contains`.

10 ⟨*test.f95* 10⟩≡

```
program test
  ⟨déclarations 11⟩
  ⟨initialise x0 12⟩
  ⟨les deux tests 13⟩
contains
  ⟨sous-programme interne 14⟩
end program test
```

5.2 Déclarations

La déclaration `use aln` prévient le compilateur qu'on va appeler des sous-programmes ou fonctions du module `aln`. L'attribut `parameter` signale que `dim` est une constante, qui ne sera jamais modifiée par le programme.

11 ⟨*déclarations* 11⟩≡ (10)

```
use aln
implicit none
integer, parameter :: dim=5
real, dimension(dim,dim) :: a,lup
integer, dimension(dim-1) :: trans
real, dimension(dim) :: x0,x,b
integer :: i,j,k
```

5.3 Initialisation aléatoire de x_0

Le vecteur x_0 est engendré par le générateur pseudo-aléatoire, à l'aide des sous-programmes prédéfinies `random_seed` et `random_number`.

Pour executer un sous-programme, on l'apelle avec `call`, sans oublier de lui passer les bon arguments.

12 ⟨*initialise x0* 12⟩≡ (10)

```
call random_seed      ! initialise le generateur pseudo aleatoire
call random_number(x0) ! tire au hasard dans [0,1] les coeff de x
print*,"vecteur x0 aleatoire:"
print*,x0
```

5.4 L'initialisation des deux tests

Pour initialiser la matrice a, nous engendrons la suite de ses coefficients à l'aide de deux boucles imbriquées, puis nous lui donnons sa forme carrée avec reshape.

13 ⟨*les deux tests* 13⟩≡ (10)

```
print*,"premier essai : *****************"
! symetrique definie positive bien conditionnee (diag. dom.),
a = reshape((/((1/(1.+abs(i-j)),i=1,dim),j=1,dim)/),(/dim,dim/))
do i=1,dim
   a(i,i)=-1.1*(sum(a(:,i))-a(i,i))
end do
call calcul_et_affiche
print*,"deuxieme essai ******************"
! symetrique definie positive mal conditionnee (mat. de Hilbert),
a = reshape((/((1.0/(1+i+j),i=1,dim),j=1,dim)/),(/dim,dim/))
call calcul_et_affiche
```

5.5 Le déroulement d'un test

C'est là que l'on utilise les sous-programmes du module. La fonction prédéfinie `maxval` donne la valeur maximale d'un vecteur. Noter que `maxloc` donne l'indice de l'(du premier) élément de valeur maximale, et qu'on aurait donc pu l'utiliser dans la sous-routine `aelup`.

14 ⟨*sous-programme interne* 14⟩≡ (10)

```
subroutine calcul_et_affiche
  print*, "matrice a:"
  do k=1,dim
     print*,a(k,1:dim)
  end do
  lup=a; call aelup(lup,trans)
  print*, "matrice lup:"
  do k=1,dim
     print*,lup(k,1:dim)
  end do
  b=pmv(a,x0)
  x=b; call lupxeb(lup,trans,x)
  print*," pivots :",trans
  print*,"erreur :",maxval(abs(x-x0))
end subroutine calcul_et_affiche
```

6 Compilation et assemblage

Comme nous avons maintenant deux fichiers fortran `aln.f95` et `test.f95`, le premier étant un module et le second un programme utilisant le premier, la compilation est plus compliquée. Il faut d'abord compiler `aln.f95` avec l'option `-c` qui produit un fichier binaire objet `.o` et un fichier d'interface `.mod`. On peut alors seulement compiler `test.f95`, avec l'option `-c` pour produire son fichier objet `.o`. Enfin on peut assembler les fichiers objets du programme et du module pour produire l'executable.

Afin de ne pas recompiler `aln` si seulement `test` est changé, et pour simplifier la commande, on utilise l'utilitaire `make`. Pour cela on écrit dans un fichier nommé `makefile` les dépendances et les règles énoncées si dessus.

La forme de base est une ligne

```
cible : dependances
```

suivie de une ou plusieurs lignes commençant par `<tab>` et donnant les commandes à exécuter pour produire la `cible`.

Le langage de `make` permet de définir et d'utiliser des variables. Certaines sont prédéfinies : `$@` représente la cible, c'est-à-dire le nom du fichier à produire. `$+` représente les dépendances, c'est-à-dire la liste des fichiers utilisés dans la production de la cible. `$<` représente la première dépendance.

Lorsque la commande `make` est lancée, elle vise la première cible, ici l'exécutable `test`.

Détail obscur réservé aux plus curieux, la ligne `# -*- makefile-mode -*-` est un commentaire pour `make`, mais une indication pour le choix du mode par `emacs` lors de l'édition du fichier source de ce document (avec le mode mineur MMM).

15 ⟨*makefile* 15⟩≡ 16 ▷

```
# -*- makefile-mode -*-
WARN=-Wall -Wextra
COMP=gfortran $(WARN)
test : test.o aln.o
        $(COMP) -o $@ $+
test.o : test.f95 aln.mod
        $(COMP) -c $<
aln.o aln.mod : aln.f95
        $(COMP) -c $<
```

7 Et encore de la programmation documentée

Nous rajoutons dans le `makefile` les commandes pour produire d'une part les sources en FORTRAN 95 (`*.f95`) et d'autre part cette documentation pdf (`rsl.pdf`) à partir d'un unique fichier `rsl.nw`, texte au format `noweb`. Pour obtenir le fichier `pdf` il faudra lancer la commande `make rsl.pdf`

Noter que ce fichier `makefile` devra lui-même être produit à partir du document source littérale `rsl.nw` par la commande `notangle -Rmakefile -t2 rsl.nw >makefile` (l'option `-t2` préservant les `tab`, essentiels pour `make`).

16 ⟨*makefile* 15⟩+≡ ◁15

```
# -*- makefile-mode -*-
# pour produire ces documents
aln.f95 test.f95 : rsl.nw
	notangle -R$@ $< > $@
rsl.pdf : rsl.tex
	pdflatex rsl
	pdflatex rsl
rsl.tex : rsl.nw
	noweave -delay -index $<  | sed -e s/-/-/g >$@
clean :
	rm -f rsl.log rsl.aux aln.mod aln.o test.o
purge :
	rm -f test aln.f95 test.f95 rsl.pdf rsl.tex *~
```

A Résumé des éléments de FORTRAN 95 rencontrés

Rappelons que le numéro souligné correspond à la *définition*, c'est-à-dire en fait à la première occurrence, et donc peut-être à une explication.

B Bouts de code

C Exercice

1. Pour `x` et `y` deux vecteurs réels de même taille, que calcule l'expression `sum(x*y)` ?
2. Enrichir la librairie de fonctions pour calculer le produit d'un vecteur (ligne) et d'une matrice, puis de deux matrices. Essayer et comparer différents ordres des boucles sur des matrices et vecteurs de grandes tailles.
3. Enrichir la librairie d'une décomposition LU sans pivotage. Proposer des tests mettant en valeur l'intérêt et les limites du pivotage.
4. Enrichir la librairie d'une décomposition LU avec pivotage des lignes ($A = PLU$), et d'une autre pivotant à la fois lignes et colonnes ($A = PLU\tilde{P}$). Proposer des tests comparant l'efficacité des trois sortes de pivotage sur des matrices de grandes tailles.