Informe - Pràctica descomposició LU

Louis Clergue

26/04/15

1 Objetiu i orientació

Amb aquesta pràctica es vol aconseguir una primera experiència en la implementació d'un mètode numèric, concretament la descomposició LU d'una matriu A; amb la intenció de computar-ne el determinant, la inversa, resoldre un sistema compatible determinat i calcular mesures de l'error en fer aquestes operacions.
Un cop el problema plantejat hem de decidir com s'implementarà i quin ús posterior volem fer del programa. En el meu cas he escollit el C++ (versió C++11) com a llenguatge de programació degut a què és el que millor domino i també el en què més vull aprofundir. L'objetiu principal és que el programa sigui eficient però també, en la mesura del possible, reutilizable (per a altres mètodes p.ex).

2 Estructura de dades

En mètodes numèric té especial importància el cost operacional d'un mètode, que influeix directament en el temps d'execució; però, cal tenir present que un dels majors costos temporals és la gestió de la memòria (assignació, inicialització, accés, alliberament). Personalment considero que és tant important escollir l'estructura de dades com l'algorisme, per això he dedicat bona part de la pràctica en escollir una estructura i en crear operacions per poder implementar els algorismes amb el mínim cost adicional. En el cas de la descomposició LU manegem exclusivament matrius i vectors; l'accés sempre serà seqüèncial per files/columnes/diagonal, les mides seran fixes un cop declarades i els elements són tots reals. El contenidor més natural és un array de 1 i 2 dimensions per vectors i matrius respectivament.

2.1 Array vs Vector

Ara s'ens presenten dues opcions utilitzar arrays o vectors, el primer un contenidor original del C i el segon una classe de la STL del C++. Personalment no estava segur de quin escollir: els arrays guarden només en memòria els elements que contenen però s'ha de fer la gestió de la memòria, els vectors en canvi gestionen la memòria i tenen atributs de mida, capacitat, etc i mètodes per accedir i modificar el contingut. Semblaria que l'array, al tenir menys *features* és més eficient però el vector és molt més segur i l'ús n'és més "explícit". Vaig preguntar al meu professor de programació de la FIB i, una breu conclusió és que un vector és molt semblant a un array: els rendiments són semblants sempre que és tingui en compte el cost de les operacions que s'utilitzen. Conclusió, escullo la classe vector fent un estudi previ de les operacions que proporciona i els seus costos.

2.2 Classe Matriu

Vista la necessitat d'una estructura eficient per guardar Matrius, creem la classe Matriu un *wrapper* sobre un vector de vectors de doubles. Implementa un operador (i,j) per accedir als elements i un (i) per obtenir els vectors fila; uns quants constructors; la permutació de files (cost constant permutant referències); i les normes de matrius:

$$\|A\|_1 = max\left\{\sum_{i=1}^{n}|a_{ij}| \quad \forall j = 1:n\right\} \quad \|A\|_\infty = max\left\{\sum_{j=1}^{n}|a_{ij}| \quad \forall i = 1:n\right\}$$

3 Algorismes implementats

3.1 Descomposició LU

Volem descompondre una matriu A quadrada no singular en la següent forma PA = LU on P una matriu de permutació (invertible i ortonormal), L triangular inferior amb 1 a la diagonal i U una matriu triangular superior:

$$P\begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ a_{31} & a_{32} & \dots & a_{3n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{pmatrix} = \begin{pmatrix} 1 & & & & \\ m_{21} & 1 & & & \\ m_{31} & m_{32} & 1 & & \\ \vdots & \vdots & \ddots & \ddots & \\ m_{n1} & m_{n2} & \dots & m_{n,n-1} & 1 \end{pmatrix} \begin{pmatrix} u_{11} & u_{12} & u_{13} & \dots & u_{1n} \\ & u_{22} & u_{23} & \dots & u_{2n} \\ & & u_{33} & \dots & u_{3n} \\ & & & \ddots & \vdots \\ & & & & u_{nn} \end{pmatrix}$$

Calculats de la manera següent: els elements $a_{ij}^{(0)}$ són de la matriu A original; en el pas i-èssim es fa pivotatge si cal , es calculen els elements de L i U seguint l'esquema i es generen els $a_{ij}^{(i+1)}$

<table>
<tr><td colspan="3">u_1</td></tr>
<tr><td rowspan="2">m_1</td><td colspan="2">u_2</td></tr>
<tr><td>m_2</td><td></td></tr>
</table>

$$u_{ij} = a_{ij}^{(i)} \quad \forall j = i \div n$$
$$m_{ij} = \frac{a_{ji}^{(i)}}{a_{ii}^{(i)}} \quad \forall j = i+1 \div n$$
$$a_{jk}^{(i+1)} = a_{jk}^{(i)} - m_{ij} \cdot u_{ik} \quad \forall j = i+1 \div n,\ k = i+1 \div n$$

En el pas 1 es generen els u_1 i els m_1, en el pas 2: u_2 i m_2, etc. Algorisme implementat:

```
for i = 1 ÷ n
    pivot(i)
    for j = i + 1 ÷ n
        m_ij = a_ji \ a_i,i
        a_ji = m_ij
        for k = i + 1 ÷ n
            a_jk = a_jk − m_ij ∗ a_i,k
```

On pivot(i) permuta la fila i per la que li toca segons un pivotatge parcial esglaonat, és a dir: permutem la fila i per la fila

$$\underset{j\in\{i\dots n\}}{\operatorname{argmax}} \frac{|a_{ji}|}{\max\limits_{k=i\div n} |a_{jk}|}$$

Implementat amb l'algorisme:

```
    pivot_row = i
    pivot_el = 0
    for j = i ÷ n : //per a cada fila j
        max_j = |a_ij|
        for k = j + 1 ÷ n : //per a cada columna k de la fila j
            max_j = max{|a_jk|, max_j}
        max_j = |a_ji|/max_j // pivot/max
        if max_j > pivot_el :
            pivot_el = max_j
            pivot_row = j
    if pivot_row ≠ i :
        A_i ↔ A_pivot_row //permuta files
        P_i ↔ P_pivot_row //permuta vector p
        signP = ¬signP //canvia paritat
```

⋆ En primera instància no vaig guardar la paritat de la permutació i calculant el determinant aquest podia tenir signe erroni.

Un cop feta la substitució la matriu A ha quedat modificada i ara conté les matrius L i U alhora:

$$A = \begin{pmatrix} u_{11} & u_{12} & u_{13} & \dots & u_{1n} \\ m_{21} & u_{22} & u_{23} & \dots & u_{2n} \\ m_{31} & m_{32} & u_{33} & \dots & u_{3n} \\ \vdots & \vdots & \ddots & \ddots & \vdots \\ m_{n1} & m_{n2} & \dots & m_{n,n-1} & u_{nn} \end{pmatrix}$$

3.2 Resolució Ax=b

Ara que tenim la descomposició $PA = LU$ podem aprofitar-la per resoldre un sistema compatible determinat $Ax = b$

$$Ax = b \iff PAx = Pb \iff LUx = Pb \iff \begin{matrix} Ly = Pb \\ Ux = y \end{matrix} \Big\}$$

Que consisteix en resoldre 2 sistemes triangulars.

3.2.1 Substitució endavant

Volem trobar la solució y de :

$$Ly = Pb \iff \begin{pmatrix} 1 & & & & \\ m_{21} & 1 & & & \\ m_{31} & m_{32} & 1 & & \\ \vdots & \vdots & \ddots & \ddots & \\ m_{n1} & m_{n2} & \dots & m_{n,n-1} & 1 \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_n \end{pmatrix} = P \begin{pmatrix} b_1 \\ b_2 \\ b_3 \\ \vdots \\ b_n \end{pmatrix}$$

Per a tractar Pb aprofitem el vector de permutació P generat al llarg dels pivotatges. Sabem que $P_i = j$ implica que a la fila i de la matriu LU hi ha la fila j de la matriu A. Aprofitant això, el nostre sistema esdevé:

$$\begin{pmatrix} 1 & & & & \\ m_{21} & 1 & & & \\ m_{31} & m_{32} & 1 & & \\ \vdots & \vdots & \ddots & \ddots & \\ m_{n1} & m_{n2} & \dots & m_{n,n-1} & 1 \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_n \end{pmatrix} = \begin{pmatrix} b_{P_1} \\ b_{P_2} \\ b_{P_3} \\ \vdots \\ b_{P_n} \end{pmatrix}$$

I ara podem aplicar una substitució endavant, per trobar y.

$$\begin{aligned} y_1 &= b_{P_1} \\ y_i &= b_{P_i} - \sum_{j=1}^{i-1} m_{ij} \cdot y_j \quad \forall i = 2 \div n \end{aligned}$$

Implementat literalment

```
1  y_1 = b_{P_1}
2  for i = 2 ÷ n :
3      sum = 0
4      for j = 1 ÷ i − 1 : //multiplicació fila i
5          sum = sum + a_ij * y_j
6      y_i = b_{P_i} − sum
```

⋆ En primera instància un podría decidir llegir el vector i permutar-lo; més eficient: guardar-lo al lloc que toca (permutat); o, encara millor guardar-lo no permutat (guanyem generalització i es pot reutilizar el vector b ja que no es modifica) que és el que faig.

3.2.2 Substitució enrere

Ara que tenim y de $Ly = Pb$ polem resoldre:

$$Ux = y \iff \begin{pmatrix} u_{11} & u_{12} & u_{13} & \dots & u_{1n} \\ & u_{22} & u_{23} & \dots & u_{2n} \\ & & u_{33} & \dots & u_{3n} \\ & & & \ddots & \vdots \\ & & & & u_{nn} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_n \end{pmatrix}$$

amb una substitució enrere.

$$x_n = \frac{y_n}{u_{ii}}$$
$$x_i = \frac{y_i - \sum_{j=i+1}^{n} u_{ij} \cdot x_j}{u_{ii}} \quad \forall i = n-1 \div 1$$

Molt similar a l'anterior:

```
1  x_n = y_n
2  for i = n − 1 ÷ 1 :
3      sum = 0
4      for j = i + 1 ÷ n : //multiplicació fila i
5          sum = sum + a_ij ∗ x_j
6      x_i = (y_i − sum)/a_ii
```

3.3 Inversa

Ara que tenim implementat els mètodes de substitució endavant i enrere podem aprofitar-los per calcular la inversa de la matriu A. Busquem $X \in \mathcal{M}_n$ tal que $AX = XA = I$, sabem d'algebra lineal que $AX = I \implies XA = I$ així que volem resoldre:

$$AX = I \iff PAX = P \iff LUX = P \iff \left.\begin{matrix} LY = P \\ UX = Y \end{matrix}\right\}$$

Tots els càlculs es fan i s'escriuen sobre la mateixa matriu $A^{-1} = b_{ij}$ que són vectors guardats en columnes (per poder usar els mètodes ja implementats). Tot i que siguin vectors columnes, mantindré la notació de b_{ij} element de la fila i i columna j.

3.3.1 LY=P

Com que L és invertible $Y = L^{-1}P$ amb L triangular inferior amb 1s a la diagonal i P matriu de permutació: Y serà una permutació d'una matriu triangular inferior amb 1s a la diagonal. Podem computar de les següents maneres

1. Podem resoldre $2n$ sistemes triangulars amb els vectors columna de la identitat aprofitant les substitucions implementades. És la solució que primer s'ens acudeix però no la més eficient: estaríem generant n vectors en memòria (ja sabem que això és molt lent) i tampoc usem que Y és triangular inferior amb 1 a la diagonal.

2. Per aprofitar-ho podem resoldre $AX = I$ com:

$$PAX = P \iff PAXP^t = I \iff LUXP^t = I \iff \begin{cases} LY = I \\ UZ = Y \\ X = ZP \end{cases}$$

   I llavors en el primer pas $LY = I$ ja sabem que Y és triangular inferior amb 1s a la diagonal; però hem afegit un 3r pas $X = ZP$ on hem de permutar la solució trobada.

3. La solució final és resoldre el sistema tal com s'ha plantejat $\begin{cases} LY = P \\ UX = Y \end{cases}$ sabent que Y és triangular inferior amb 1s a la diagonal però permutada!

Doncs l'algorisme queda, per a cada columna j (Pt el vector de permutació invers, el de P^t):

$$\begin{aligned} k &= Pt_j \quad \text{“la columna j té un 1 a la posició k”} \\ y_{ij} &= 0 \quad i = 1 \div k-1 \\ y_{kj} &= 1 \\ y_{ij} &= -\sum_{s=k}^{i-1} m_{is} \cdot y_{sj} \quad i = k+1 \div n \end{aligned}$$

L'implementació en pseudo-codi:

```
1  for j = 1 ÷ n :
2      k = Pt_j  // tenim k-1 zeros...
3      for i = 1 ÷ k :
4          b_ij = 0
5      b_ik = 1  // ...i un 1
6      for i = k+1 ÷ n :  //substitució normal
7          sum = 0
8          for s = k+1 ÷ i-1 :
9              sum = sum + a_is * b_sj
10         b_ij = -sum
```

3.3.2 UX=Y

Aquesta substitució és plena (genera A^{-1}) i l'única millora seria no accedir als elements de Y que són nuls i directament guardar la suma negativa, però això complica el codi (s'hauria d'especialitzar la substitució enrere accedint al vector i comparant) i no val la pena. Per això utilitzo el mètode de substitució enrere descrit prèviament per a cada columna.

3.4 Determinant

Amb la descomposició LU és molt ràpid càlcular el determinant de A:

$$PA = LU \iff A = P^{-1}LU \implies det(A) = det(P^{-1}) \cdot det(L) \cdot det(U) = \pm 1 \cdot 1 \cdot det(U) = \pm \prod_{i=1}^{n} u_{ii}$$

On el signe és el signe de la permutació de la matriu P que s'ha anat guardant en un booleà mentres es feien els pivotatges.

```
1  if signP : det = -1
2  else     : det = 1
3  for i = 1 ÷ n : det = det * a_ii
```

4 Comentaris

Al llarg de la pràctica han sortits detalls tècnics que han sigut reflexionats per tal de ser el més òptim possible, dels quals els més importants estan aquí

4.1 $\|PA - LU\|_\infty$

Aquesta operació és bastant costosa pel càlcul de $L \cdot U$ però no hem d'incrementar el cost amb processos “inútils” com :

1. Permutar A o, pitjor, generar la matriu PA
2. Generar i guardar la matriu $L \cdot U$
3. Guardar la resta d'aquestes matrius

La millor opció (que he trobat) és calcular per a cada a_{ij} de A l'element $(L \cdot U)_{ij}$ corresponent tenint en compta la permutació ($\|PA - LU\|_\infty = \|A - P^{-1}LU\|_\infty$, P permutació de files) computar la substracció i usar aquesta en el comput de la norma. Així assignem a memòria que només necessitem un cop.

4.2 Escriptura aleatoria i push_back

Per inicialitzar un vector utilitzem dues tècniques:

Accés aleatori
Ens generem els n elements del vector, és a dir, es creen n instàncies per defecte; així podem accedir als elements amb $v[i]$ i guardar els elements on volem.

push_back
Si els elements es guarden de forma seqüèncial del primer a l'últim, podem evitar inicialitzar n element i només canviar la capacitat del vector (els elements que pot contenir) i utilitzar $v.push\_back(q)$ per afegir q al final del vector.

4.3 Comprovació

Està molt bé implementar el mètode però, funciona? Podem provar el funcionament general amb alguns casos senzills de dimensions petites però no és suficient. Per això he creat un programa que em genera matrius i vectors aleatoris amb les dimensions que desitji i els elements en el rang que li indiqui. D'aquesta manera he pogut provar matrius de dimensions 10, 50, 100, 200, 500 i 1000 principalment.
$\star$ Nota: les matrius aleatories tenen rang màxim amb probabilitat 1 així que cap problema.

4.4 Cost temporal

També volia saber quin temps tardava el meu programa i en quina mesura augmentava en funció de la dimensió, així que vaig aprofitar el generador: Aquest són temps compilant amb g++ -std=c+11 -O2 sobre un Intel Core i3 M380:

Dimensió	10	20	50	100	300	500	1000
Temps	1.543 ms	2.83 ms	8.49 ms	31.678 ms	0.272 s	0.924 s	7.768 s

4.5 Proporció de costos

Utilitzant l'eina gprof de GNU puc saber quines són les operacions que més temps empren. Com que per matrius petites el procés és quasi instantani i no s'obté suficient informació, els resultats són per matrius 500x500; sense optimitzacions del compilador. Els costos de les functions en proporció temporal són:

- 37.5% càlcul de la inversa
- 31.4% descomposició LU
- 24.3% $\|PA - LU\|_\infty$
- 0.1% resolució Ax=b
- 0.0% det(A)

Cal remarcar que 36.6% del temps es fan accesos a elements de les matrius; i que 24% del temps de la descomposició és pel pivotatge.