Actividades 2

Prácticas de Cálculo Numérico I (doble grado)

22 de febrero de 2022

1. Vectorización de la descomposición LU

Aprovechándonos de que las matrices y vectores son estructuras de datos básicas en `matlab/octave`, en vez de tres bucles podemos utilizar solo dos en el código antes indicado para la descomposición LU. Cambiamos el más interno, el que involucra `j` por una sola línea que considera una porción de la `ii`-ésima fila de `U` como un todo en vez de recorrerla elemento a elemento con el bucle en `j`:

```
U(ii,k:n) = U(ii,k:n) - L(ii,k)*U(k,k:n);
```

Es decir, esto reemplaza a las líneas 9, 10 y 11.

La filosofía es que en `matlab/octave` las matrices o vectores sirven a veces como sustituto para bucles. ¿Hay alguna ventaja en operar un trozo de fila de `U` como un vector en vez de operar sus elementos? A fin de cuentas, sumar dos vectores de m coordenadas o hacer un bucle que recorra m sumas es lo mismo. La ventaja es que el trabajo se deja a la parte que no vemos, la parte no interpretada, y `matlab/octave` es muy bueno operando internamente con matrices y vectores. A diferencia de lo que nos ocurriría a nosotros trabajando a mano, le va a costar más acceder por separado a cada coordenada que tratar el vector como un todo, o al menos es lo que dice la documentación (véanse los apuntes sobre el rendimiento más adelante).

Con mentalidad informática, uno puede pensar que el `k:n` que aparece en la nueva línea, exigirá reservar memoria tres veces para vectores que son iguales, entonces es de sospechar que reemplazando el interior del bucle `for` en `k` por

```
v = k:n;
for ii = k+1:n
    L(ii,k) = U(ii,k)/U(k,k);
    U(ii,v) = U(ii,v) - L(ii,k)*U(k,v);
end
```

obtendremos alguna ventaja en cuanto al rendimiento. Es decir, el código completo sería

```
n = size(A,1);
U = A;
L = eye(n);

for k = 1:n-1
    v = k:n;
    for ii = k+1:n
        L(ii,k) = U(ii,k)/U(k,k);
        U(ii,v) = U(ii,v) - L(ii,k)*U(k,v);
    end
end
```

ACTIVIDAD 2.1.1. *Hay un teorema [QS07, §5.1] que dice que una matriz A no tiene descomposición LU "pura" (sin pivotes) si y solo si los menores angulares, exceptuando A, son no nulos. Busca una matriz entera 3×3 con menores angulares $\Delta_1, \Delta_3 \neq 0$ y $\Delta_2 = 0$ y comprueba qué tipo de error o resultado produce el código anterior.*

Para entender más acerca de cómo funciona el código anterior, apliquémoslo a la matriz

$$A = \begin{pmatrix} 2 & 1 & -1 & 1 \\ 4 & 3 & 0 & 3 \\ -2 & 1 & 4 & -2 \\ 2 & 3 & 0 & -4 \end{pmatrix}$$

e introduzcamos en el código unas línea que muestren las matrices `L` y `U` después del bucle interior. Lo que obtenemos para `L` en cada uno de los tres pasos de `k` es:

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 2 & 1 & 0 & 0 \\ -1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 \end{pmatrix}, \quad \begin{pmatrix} 1 & 0 & 0 & 0 \\ 2 & 1 & 0 & 0 \\ -1 & 2 & 1 & 0 \\ 1 & 2 & 0 & 1 \end{pmatrix}, \quad \begin{pmatrix} 1 & 0 & 0 & 0 \\ 2 & 1 & 0 & 0 \\ -1 & 2 & 1 & 0 \\ 1 & 2 & 3 & 1 \end{pmatrix}.$$

y para `U`:

$$\begin{pmatrix} 2 & 1 & -1 & 1 \\ 0 & 1 & 2 & 1 \\ 0 & 2 & 3 & -1 \\ 0 & 2 & 1 & -5 \end{pmatrix}, \quad \begin{pmatrix} 2 & 1 & -1 & 1 \\ 0 & 1 & 2 & 1 \\ 0 & 0 & -1 & -3 \\ 0 & 0 & -3 & -7 \end{pmatrix}, \quad \begin{pmatrix} 2 & 1 & -1 & 1 \\ 0 & 1 & 2 & 1 \\ 0 & 0 & -1 & -3 \\ 0 & 0 & 0 & 2 \end{pmatrix}.$$

Vemos que estas tres matrices corresponden a los diferentes pasos de la eliminación de Gauss tal como la llevaríamos a cabo nosotros "a mano". Por otra parte, las columnas de L se van rellenando con los coeficientes que usamos para crear ceros bajo los pivotes.

2. Ahorrando memoria

Para almacenar los elementos de una matriz $n \times n$ necesitamos n^2 posiciones de memoria (o algo proporcional a ello). Al efectuar la descomposición LU, necesitamos también guardar los elementos de L y de U. Ya que L y U están medio vacías, economizando tendríamos ocupadas del orden de $n^2 + n^2/2 + n^2/2 = 2n^2$ posiciones de memoria. Procediendo con los programas anteriores no estamos aplicando dicha economía porque guardamos todos los elementos de L y U, aunque sean nulos.

En realidad, en el caso de la descomposición LU sin pivotes hay una manera sencilla de proceder que permite reducir las posiciones de memoria necesarias a la mitad. Es el *algoritmo de Crout* [PTVF92, §2.3]. La idea es sencilla: sobreescribir la matriz A con los elementos de las matrices L y U. Aunque una matriz triangular tiene genéricamente $n(n+1)/2$ elementos no nulos, sabemos que la diagonal de L está formada por n unos y no hace falta almacenarlos, así que la aritmética cuadra perfectamente:

$$\frac{n(n+1)}{2} + \left(\frac{n(n+1)}{2} - n\right) = n^2.$$

Con una pequeña salvedad (véase la siguiente actividad), el código es como antes, identificando con `A` las matrices `L` y `U`, que ahora desaparecen:

```
A = [1,2,1;3,4,5;5,8,1];
n = size(A,1);
for k = 1:n-1
    v = k+1:n;
    for ii = k+1:n
        A(ii,k) = A(ii,k)/A(k,k);
        A(ii,v) = A(ii,v) - A(ii,k)*A(k,v);
    end
end
disp('A')
disp(A)
```

La matriz `A` contiene toda la información: la mitad sobre la diagonal, incluida, produce U y el resto completando la diagonal con unos, da L. Para aprender nuevos comandos, completemos el código anterior para que muestre L y U. En `matlab/octave` existen `tril` y `triu` que dan la parte triangular inferior y superior de una matriz, por tanto U es simplemente `triu(A)`. Estos dos comandos admiten un segundo argumento, un entero positivo o negativo, que indica cuántas posiciones queremos separarnos de la diagonal. Con ello, `tril(A,-1)` da L salvo la diagonal. En definitiva, L y U aparecerán en el formato habitual añadiendo las líneas:

```
disp('L =')
disp(tril(A,-1) + eye(n))
```

```
disp('U =')
disp(triu(A))
```

ACTIVIDAD 2.2.1. *Comprueba si el programa funciona con el* `v = k:n` *que habíamos usado antes. ¿Por qué quitamos la primera coordenada a* `v`*?*

3. Sistemas lineales con LU

Recordemos que la descomposición LU no era un capricho teórico sino que tenía como propósito resolver rápidamente sistemas de ecuaciones lineales. Es una realización de la reducción de Gauss. El hecho clave es que si $A = LU$, resolver $A\vec{x} = \vec{b}$ se reduce a resolver sucesivamente los sistemas $L\vec{y} = \vec{b}$ y $U\vec{x} = \vec{y}$. Ahora bien, un sistema lineal con una matriz triangular es muy fácil de resolver. Simplemente basta despejar y sustituir sucesivamente de abajo a arriba, si la matriz es triangular superior, o de arriba a abajo, si la matriz es triangular inferior. Concretamente:

El sistema $L\vec{x} = \vec{b}$ con L triangular inferior, tiene como solución

$$x_i = \frac{1}{l_{ii}}\Big(b_i - \sum_{j=1}^{i-1} l_{ij}x_j\Big), \qquad \text{con} \quad x_1 = \frac{b_1}{l_{11}}.$$

Y el sistema $U\vec{x} = \vec{b}$ con U triangular superior, tiene como solución

$$x_i = \frac{1}{u_{ii}}\Big(b_i - \sum_{j=i+1}^{n} u_{ij}x_j\Big), \qquad \text{con} \quad x_n = \frac{b_n}{u_{nn}}.$$

ACTIVIDAD 2.3.1. *Crea funciones* `ltrs` *y* `utrs` *(de lower y upper triangular system) que resuelvan sistemas triangulares usando estas fórmulas.*

Una vez definidas las funciones de la actividad anterior, es inmediato construir nuestro algoritmo para resolver sistemas lineales. Con `mlu` se indica en el siguiente código la función correspondiente a nuestra implementación de la descomposición LU.

```
1 [L,U] = mlu(A);
2 y = ltrs(L,b);
3 x = utrs(U,y);
4 disp(x)
```

Seguro que con esto no ganaremos en velocidad al `A\b` de `matlab/octave`, pero sí será más eficiente y rápido para sistemas grandes que calcular la

inversa `inv(A)` y multiplicarla por `b`. Recuerda, por otra parte, que nuestro algoritmo no es infalible porque no todas las matrices tienen una descomposición LU, aunque, en cierto sentido, hay probabilidad nula de que nos topemos con una excepción eligiendo matrices al azar.

ACTIVIDAD 2.3.2. *Define la función* `mlu` *para que el código anterior sea operativo.*

4. La descomposición LU con pivotes

La técnica de los pivotes requiere calcular el elemento máximo de una columna, el pivote, y pasarlo a la diagonal intercambiando filas. El algoritmo será igual que antes salvo intercalar este procedimiento. Para apreciar la ventaja de esta complicación sobre un ejemplo, véase [Atk89, §8.2].

El máximo de un vector `v` se calcula en `matlab/octave` con `max(v)`. Si sustituimos `v` por una matriz lo que obtendremos es el vector fila dado por los máximos de las columnas. Nosotros más que el máximo, lo que queremos saber es el índice del máximo, la posición en al que se alcanza, para así trasladarlo a su nueva posición. Es lo que en el contexto del análisis a veces se escribe arg máx. Por ejemplo, el máximo de $f(x) = 1-x^2$ es 1 y $\arg\max f = 0$ porque el máximo se alcanza en $x = 0$. Para que `matlab/octave` nos dé la posición del máximo basta recuperar su salida con dos variables, la primera será el máximo y la segunda su posición. El siguiente código es un ejemplo de su funcionamiento.

```
1 v = [3,5,-7,1];
2 m = max(v);
3 disp(['Valor máximo de las coordenadas ' num2str(m)])
4 [m, j] = max(v);
5 disp(['Posición ' num2str(j)])
```

En rigor el pivote no se calcula con el máximo de los elementos de una columna sino con el máximo de sus valores absolutos. Eso tiene tan fácil solución como introducir la función `abs` que toma valores absolutos (o normas en el caso de números complejos) de los elementos de una matriz. De esta forma, cambiando `max(v)` por `max(abs(v))` la salida pasa a ser 7 alcanzado en la posición 3.

ACTIVIDAD 2.4.1. *Escribe un programa que utilice una discretización de la función* $f(x) = x^2e^{-x}$ *y el comando* `max` *para hallar una aproximación del máximo de* f *en* $[0, 4]$ *y el valor en el que se alcanza. Haz también los cálculos analíticamente y compara los resultados.*

Lo segundo que tenemos que pensar es cómo intercambiar dos filas en `matlab/octave`. Una solución fea y poco eficiente, que no escribo para que no dé malas ideas a nadie, es intercambiar uno a uno los elementos de las filas. Más elegante es tratar las filas como vectores e intercambiarlas como se haría en C/C++, u otros lenguajes, por medio de una variable temporal intermedia. Por ejemplo, intercambiaríamos las filas 2 y 3 de `A` mediante

```
1 A = [1,2,1;3,4,5;5,8,1];
2 temp = A(2,:);
3 A(2,:) = A(3,:);
4 A(3,:) = temp;
```

Sin embargo hay algo todavía más breve que recuerda mucho a la solución para el intercambio de variables en python. Podemos "vectorizar" también los índices de las filas a intercambiar y conseguir nuestro propósito cambiando su orden con una sola instrucción:

```
1 A = [1,2,1;3,4,5;5,8,1];
2 A([2 3],:) = A([3 2],:);
```

Tras estas consideraciones, el pseucódigo visto en la clase de teoría, con la matriz ejemplo `A` considerada allí, se traduce en:

```
1  A = [3,2,-1;6,6,2;-1,1,3];
2
3  n = size(A,1);
4  U = A;
5  L = eye(n);
6  P = eye(n);
7
8  for k = 1:n-1
9          [m,j] = max( abs(U(k:n,k)) );
10         if j~=1
11                 r = j+k-1;
12                 U([k,r], k:n) = U([r,k], k:n);
13                 L([k,r], 1:k-1) = L([r,k], 1:k-1);
14                 P([k,r], :) = P([r,k], :);
15         end
16     v = k:n;
17     for ii = k+1:n
18         L(ii,k) = U(ii,k)/U(k,k);
19         U(ii,v) = U(ii,v) - L(ii,k)*U(k,v);
20     end
21 end
```

La gestión de los pivotes se consigue con las líneas 9–15, el resto es el mismo código empleado para la descomposición LU "pura".

Veamos su efecto paso a paso sobre

```
A = [1,2,1,-1; 2,2,-4,0; 1,4,1,3; 1,2,0,5]
```

Para ellos, modificamos el código anterior para que nos muestre `U` después de la línea 15 y después de la línea 20. Es decir, cuando ha permutado las filas

y cuando aplica reducción de Gauss. En la primera pasada los resultados son:

$$U_1 = \begin{pmatrix} 2 & 2 & -4 & 0 \\ 1 & 2 & 1 & -1 \\ 1 & 4 & 1 & 3 \\ 1 & 2 & 0 & 5 \end{pmatrix} \quad \rightarrow \quad U_2 = \begin{pmatrix} 2 & 2 & -4 & 0 \\ 0 & 1 & 3 & -1 \\ 0 & 3 & 3 & 3 \\ 0 & 1 & 2 & 5 \end{pmatrix}$$

Comprobamos que U_1 coincide con la matriz original salvo permutar las dos primeras filas, de este modo el 2, que es el mayor elemento de la primera columna, está en el lugar del primer pivote. La matriz U_2 muestra el resultado de la eliminación de Gauss con este pivote. Observamos que el elemento u_{32} es mayor que el u_{22}, que naturalmente actuaría como pivote en la descomposición LU "pura" , por tanto debemos intercambiar las filas segunda y tercera. Lo que resulta es:

$$U_1 = \begin{pmatrix} 2 & 2 & -4 & 0 \\ 0 & 3 & 3 & 3 \\ 0 & 1 & 3 & -1 \\ 0 & 1 & 2 & 5 \end{pmatrix} \quad \rightarrow \quad U_2 = \begin{pmatrix} 2 & 2 & -4 & 0 \\ 0 & 3 & 3 & 3 \\ 0 & 0 & 2 & -2 \\ 0 & 0 & 1 & 4 \end{pmatrix}.$$

En la tercera y última pasada no es necesaria ninguna permutación de filas porque $u_{33} > u_{43}$. En este caso el condicional `if` evita que se haga un intercambio trivial.

ACTIVIDAD 2.4.2. *Define una función* `plu` *tal que* `[L,U,P]=plu(A)` *funcione como el comando correspondiente* `lu` *en* `matlab/octave`.

Se puede crear una descomposición con *pivotes totales*, lo que significa que se permite intercambiar filas y también intercambiar columnas con el fin de elegir como pivote un elemento aún mayor en valor absoluto aunque no esté en la columna seleccionada [SB93, §4.1]. Si nos atenemos a [PTVF92, p.37], es casi tan bueno como lo que hemos hecho y por tanto no merecen la pena las complicaciones que entraña. Apenas se usa en la práctica.

5. Apuntes sobre el rendimiento

Siguiendo este enlace de la ayuda de `matlab`:

`https://www.mathworks.com/help/matlab/matlab_prog/techniques-for-improving-performance.html`

verás algunos consejos para mejorar el rendimiento. Dentro de "*Programming Practices for Performance*" lo que hemos hecho para perfeccionar el

código inicial de la factorización LU está dentro del segundo consejo: "*Vectorize –Instead of writing loop-based code, consider using MATLAB matrix and vector operations*" y en parte del tercero: "*Place independent operations outside loops*". Ahora vamos a comprobar experimentalmente si nuestro código ha mejorado.

Para estudiar el rendimiento de un bloque de código, se suele encerrar entre los comandos `tic` y `toc` lo que muestra en la ventana de comandos un mensaje con el tiempo transcurrido entre ellos. Para comparar los algoritmos queremos una matriz grande de prueba. Con `rand(`n`)` se obtiene una aleatoria, en cierto sentido, de tamaño $n \times n$. Por ejemplo, si quisiéramos medir cuánta tarda nuestro ordenador en crear 100 matrices cuadradas de dimensión 2022, escribiríamos:

```
1  tic
2  for k = 1:100
3      A = rand(2022);
4  end
5  toc
```

Con ello obtengo como salida en mi ordenador:

```
Elapsed time is 3.29011 seconds.
```

Por supuesto este tiempo presenta variaciones de acuerdo con la velocidad del ordenador e incluso con el momento en que se ejecute, por la carga no uniforme de la CPU.

Si en vez de ver el resultado en la ventana de comandos, queremos guardar el tiempo transcurrido en una variable, es conveniente usar `t0 = tic` que almacena en `t0` el tiempo en el que se llega a `tic` y `toc(t0)` que da el tiempo en el que se llega a `toc` menos `t0`, es decir, el tiempo transcurrido.

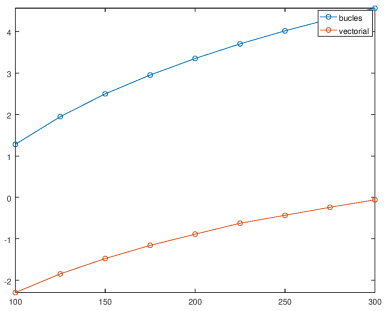
El siguiente código, guarda en los vectores `times1` y `times2` los valores del tiempo usando nuestro algoritmo inicial para la descomposición LU con bucles y el algoritmo optimizado quitando un bucle y definiendo `v`. Esos vectores se dibujan en escala semilogarítmica en una misma gráfica para compararlos. La gráfica se almacena en el fichero `figura.png`.

```
1  % Rango de dimensiones
2  D = 100:25:300;
3
4  times1 = [];
5  times2 = [];
6
7  for n = D
8          A = rand(n);
9
10         % Triple bucle
11         t0 = tic;
12         U = A;
13         L = eye(n);
```

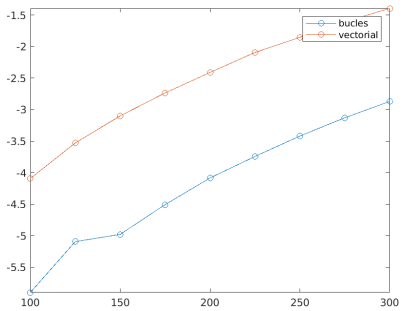
```
14          for k = 1:n-1
15                  for ii = k+1:n
16                          L(ii,k) = U(ii,k)/U(k,k);
17                          for j = k:n
18                                  U(ii,j) = U(ii,j)
                                            ↪ -L(ii,k)*U(k,j);
19                          end
20                  end
21          end
22          times1 = [times1, toc(t0)];
23
24          % Vectorial
25          t0 = tic;
26          U = A;
27          L = eye(n);
28          for k = 1:n-1
29                  v = k:n;
30                  for ii = k+1:n
31                          L(ii,k) = U(ii,k)/U(k,k);
32                          U(ii,v) = U(ii,v) -L(ii,k)*U(k,v);
33                  end
34          end
35          times2 = [times2, toc(t0)];
36
37  end
38
39  plot(D, log(times1),'-o', D, log(times2),'-o')
40  legend('bucles', 'vectorial')
41  axis tight
42
43  saveas(gcf,'figura.png')
```

La única relativa novedad del código anterior es que los vectores `times1` y `times2` no se crean desde el principio con su tamaño final sino que al principio están vacíos y se van añadiendo coordenadas dinámicamente.

Las cosas funcionan como un esperaría con `octave`, pero, sorprendentemente, con `matlab` *online* o `matlab` instalado, el algoritmo vectorial es más lento para dimensiones medianamente grandes (al menos con las últimas versiones). Aquí están dos de las figuras obtenidas.



`Octave`



`Matlab` *online*

No tengo una explicación clara para ello. Si se me permite un brindis al sol en la línea R. Stallman, un problema general con el *software* que no es

libre, es que todavía es más difícil, si no imposible, saber qué está ocurriendo por debajo, limitando las posibilidades para mejorar nuestros programas. Sospecho que el intérprete de `matlab` tiene unos trucos para cambiar internamente de alguna forma los bucles (¿es posible que sea "inteligente" hasta el punto de que paralelice?). Aquí hay algunas explicaciones sobre algo relacionado:

www.mathworks.com/matlabcentral/answers/54522-why-is-indexing-vectors-matrices-in-matlab-very-inefficient

Hay una asimetría en `matlab/octave` entre filas y columnas respecto al rendimiento, debida a que la memoria es algo secuencial, las matrices bidimensionales a la postre se guardarán como tiras de números, y hay una diferencia apreciable entre modificar un montón de posiciones de memoria consecutivas y modificar otras que no lo son. El siguiente código, ilustra este punto:

```
N = 10000;
A = rand(N);
tic
for k = 1:N
    A(:,k) = 1;
end
toc

A = rand(N);
tic
for k = 1:N
    A(k,:) = 1;
end
toc
```

ACTIVIDAD 2.5.1. *Ejecuta el código anterior quizá cambiando el valor de* `N` *para que a tu ordenador le cueste al menos unas décimas de segundo. ¿Qué tiempo es mayor, el primero o el segundo? Intenta conjeturar a partir de ello si* `matlab/octave` *guarda internamente las matrices por filas o por columnas.*

Por si te interesa la historia de la informática, los dos grandes lenguajes que monopolizaron por mucho tiempo la computación científica, FORTRAN y C, seguían políticas diferentes en el orden de almacenamiento de los elementos de matrices.

ACTIVIDAD 2.5.2. *Modifica el algoritmo para factorización LU sin pivotes intercambiando los dos índices en L y en U, de forma que ofrezca las traspuestas de L y U. ¿Es el algoritmo en esta forma más rápido o más lento que el original para matrices grandes?*

Referencias

[Atk89] K. E. Atkinson. *An introduction to numerical analysis*. John Wiley & Sons, Inc., New York, second edition, 1989.

[PTVF92] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical recipes in C*. Cambridge University Press, Cambridge, second edition, 1992. The art of scientific computing.

[QS07] A. Quarteroni and F. Saleri. *Cálculo Científico con MATLAB y Octave*. Springer, Milan, 2007.

[SB93] J. Stoer and R. Bulirsch. *Introduction to numerical analysis*, volume 12 of *Texts in Applied Mathematics*. Springer-Verlag, New York, second edition, 1993. Translated from the German by R. Bartels, W. Gautschi and C. Witzgall.