Actividades 5

Prácticas de Cálculo Numérico I (doble grado)

22 de marzo de 2022

1. Ortogonalización

Algunos problemas de álgebra lineal se vuelven más sencillos si disponemos de bases ortogonales u ortonormales de un subespacio vectorial. Por ello, dado un conjunto de vectores linealmente independientes $\{\vec{v}_1, \ldots, \vec{v}_n\}$ tiene interés obtener $\{\vec{u}_1, \ldots, \vec{u}_n\}$ ortogonales o $\{\vec{q}_1, \ldots, \vec{q}_n\}$ ortonormales que generen el mismo subespacio. A esto es a lo que se refieren los palabros *ortogonalizar* y *ortonormalizar*. Evidentemente, el paso de los $\vec{u}_j$ a los $\vec{q}_j$ es tan fácil como *normalizar* mediante $\vec{q}_j = \vec{u}_j / \|\vec{u}_j\|$.

Casi todas las veces la simplificación proviene de que hallar coordenadas en una base ortonormal $\{\vec{q}_1, \ldots, \vec{q}_n\}$ no requiere resolver un sistema lineal, sino que basta con utilizar productos escalares. Concretamente, la solución de $x_1\vec{q}_1 + \cdots + x_n\vec{q}_n = \vec{v}$ es $x_i = \vec{v} \cdot \vec{q}_i$. El siguiente programa comprueba sobre un ejemplo, en el que la ortonormalización se reduce a normalización, que el resultado de proceder con productos escalares y resolviendo el sistema es el mismo.

```
v = [1,2,2,5]';

% vectores ortogonales (de hecho con la misma norma)
v1 = [4,5,6,7]';
v2 = [-7,-2,-3,8]';
v3 = [-5,-4,9,-2]';
v4 = [-6,9,0,-3]';

% (orto)Normalización
q1 = v1/norm(v1); q2 = v2/norm(v2); q3 = v3/norm(v3); q4 =
    ↪ v4/norm(v4);

% Con productos escalares
disp([v'*q1, v'*q2, v'*q3, v'*q4 ]')

% Resolviendo el sistema
A = [q1,q2,q3,q4];
disp(' ')
disp([ A\v ])
```

A todos nos han enseñado a ortogonalizar con un algoritmo llamado *proceso de Gram-Schmidt*, que recordaremos más adelante, pero no está implementado en `matlab/octave`, seguramente porque no es lo más eficiente desde el punto de vista numérico (en [PTVF92, §2.6] se llega a decir que es "terrible", aunque esto suena un poco exagerado). El comando directo de `matlab/octave` que ortonormaliza, con cierto algoritmo, las columnas de una matriz `A` es `orth(A)` y funciona incluso si tales columnas no son linealmente independientes, es decir, elimina de algún modo las innecesarias. Por ejemplo:

```
% Vectores de partida
v1 = [-2;1;0];
v2 = [5;0;1];
v3 = v1 + v2;
v4 = 2*v1 - 7*v2;

% Las columnas de A generan un subespacio W
A = [v1,v2,v3,v4];

% Las columnas del resultado son
% base ortonormal de W
orth(A)
```

da lugar a:

```
 0.982980   -0.020408
-0.048520    0.929684
 0.177188    0.367792
```

La tercera y la cuarta columnas de `A` son superfluas en cuanto al subespacio generado porque `v1` y `v2` ya lo generan. Sin embargo, `orth(A(:,1:2))` no da la misma base ortonormal.

El proceso de Gram-Schmidt antes mencionado, responde a las fórmulas:

$$\vec{u}_k = \vec{v}_k - \sum_{j=1}^{k-1} \langle \vec{v}_k, \vec{q}_j \rangle \vec{q}_j, \qquad \vec{q}_k = \frac{\vec{u}_k}{\|\vec{u}_k\|}$$

donde se sobreentiende que para $k = 1$ estamos tomando $\vec{u}_1 = \vec{v}_1$. Aquí hay una aclaración que hacer cuando se trabaja con números complejos y es que en álgebra lineal hay una tradición de definir el producto escalar usual entre vectores columna de $\mathbb{C}^n$ como $\langle \vec{a}, \vec{c} \rangle = \vec{a}^t \overline{\vec{c}}$ (lineal en el primer argumento) mientras que en casi el resto de las áreas se toma $\langle \vec{a}, \vec{c} \rangle = \overline{\vec{a}^t} \vec{c}$ (lineal en el segundo argumento). Esta segunda definición es más simple de implementar en `matlab/octave`, pues se escribiría `a'*c`, y, si la usamos, debemos cambiar $\langle \vec{v}_k, \vec{q}_j \rangle$ por $\langle \vec{q}_j, \vec{v}_k \rangle$ en la fórmula del proceso de Gram-Schmidt. Con estas precauciones, una posible implementación sobre un ejemplo es la siguiente, donde las columnas de `Q` son los vectores ortonormalizados, los $\vec{q}_j$.

```
A = [3,9; 6,4; 2,-1];

m = size(A,2);
Q = A;
Q(:,1) = A(:,1)/norm(A(:,1));

for k = 2:m
        u_k = A(:,k);
        for j = 1:k-1
                u_k = u_k - Q(:,j)' * A(:,k) * Q(:,j);
        end
        Q(:,k) = u_k/norm( u_k );
end
```

El resultado exacto para este ejemplo es $\vec{q}_1 = \frac{1}{7}(3,6,2)^t$, $\vec{q}_2 = \frac{1}{7}(6,-2,-3)^t$.

Para cualquier entrada, si escribimos `norm(Q'*Q-eye(m))` debe obtenerse un valor próximo a cero porque el elemento i,j de `Q'*Q` es el producto escalar (con la segunda definición) de la columna i por la columna j de `Q` y entonces simbólicamente resulta la matriz identidad. Por supuesto, el épsilon máquina evitará, en casos genéricos, que obtengamos un cero exacto.

ACTIVIDAD 5.1.1. *Compara con ejemplos aleatorios de cientos de coordenadas la rapidez de* `orth` *con respecto a la implementación anterior del proceso de Gram-Schmidt.*

ACTIVIDAD 5.1.2. *Escribe un programa que, dada una matriz $m \times n$, indique qué pares de sus columnas son ortogonales. Para evitar problemas con el épsilon máquina, introduce un parámetro de tolerancia para identificar productos escalares pequeños con cero.*

2. Las matrices de Householder

Aunque es muy posible que te lo ocultaran en la asignatura correspondiente de álgebra lineal, hay una fórmula sencilla para hallar la matriz de la simetría por un plano que contiene al origen con vector normal unitario $\vec{v}$. Esta fórmula es la *matriz de Householder* $I - 2\vec{v}\vec{v}^t$ donde, como es habitual, el vector $\vec{v}$ se considera en columna. Esto se extiende de $\mathbb{R}^3$ a $\mathbb{R}^n$ sustituyendo planos por hiperplanos. En particular, tenemos una forma sencilla de construir *matrices ortogonales*. Recuerda, que reciben este nombre las matrices reales no singulares que cumplen $A^{-1} = A^t$ o, equivalentemente, cuyas columnas forman una base ortonormal de $\mathbb{R}^n$. Además las matrices de Householder son simétricas. El siguiente código comprueba que la matriz es ortogonal con un ejemplo aleatorio en dimensión `N`.

```
N = 10
```

```
2
3  v = rand(N,1);
4  v = v/norm(v);
5  Q = eye(N)-2*v*v';
6  norm(Q'*Q-eye(N))
```

El resultado no es cero, por culpa del épsilon máquina. El truco también funciona con números complejos, siendo la matriz de Householder $I - 2\vec{v}\vec{v}^\dagger$ donde † indica la traspuesta conjugada, aunque tiene menos relevancia en cálculo numérico. La terminología al uso es llamar *matrices unitarias* a las matrices no singulares complejas que cumplen $A^{-1} = A^\dagger$. De nuevo, esto equivale a que las columnas de A formen una base ortonormal de $\mathbb{C}^n$. Las matrices de Householder complejas son unitarias y además *hermíticas* (iguales a su traspuesta conjugada). El análogo complejo del código anterior solo requiere modificar la tercera línea:

```
1  N = 10
2
3  v = rand(N,1)+i*rand(N,1);
4  v = v/norm(v);
5  Q = eye(N)-2*v*v';
6  norm(Q'*Q-eye(N))
```

Si alguna vez impartes clases de álgebra lineal, la siguiente actividad te puede resultar útil. Si quieres romper la simetría de las matrices ortogonales obtenidas, puedes multiplicar dos resultados porque el producto de matrices simétricas no es, en general, una matriz simétrica.

ACTIVIDAD 5.2.1. *Escribe un programa que genere un vector aleatorio no nulo en dimensión N con coordenadas aleatorias en $[-k, k] \cap \mathbb{Z}$ y utiliza las matrices de Householder para obtener una matriz ortogonal de la forma $d^{-1}A$ con $d \in \mathbb{Z}^+$ y $A \in \mathcal{M}_{N\times N}(\mathbb{Z})$.*

Una observación muy útil para lo que viene después, es que dado un vector $\vec{x}$ es fácil construir una matriz de Householder que al ser aplicada a este vector anula todas sus coordenadas excepto la primera. Basta tomar

$$\vec{v} = \frac{\vec{w}}{\|\vec{w}\|} \qquad \text{con} \quad \vec{w} = \vec{x} \pm \|\vec{x}\|\vec{e}_1$$

donde $\vec{e}_1$ es el vector $(1, 0, 0, \ldots, 0)^t$ y el signo $\pm$ lo podemos elegir a nuestra voluntad. El siguiente código muestra que esto funciona con el signo positivo:

```
1  N = 4;
2  x = 2*rand(N,1)-1;
3  v = x;
4  v(1) = v(1)+norm(x);
5  v = v/norm(v);
6  Q = eye(N)-2*v*v';
7  Q*x
```

A estas alturas, seguro que sabes que hay cierto peligro en cálculo numérico al restar dos números próximos, por tanto es buena política elegir el signo positivo si $x_1 \geq 0$ y el signo negativo si $x_1 < 0$.

ACTIVIDAD 5.2.2. *Escribe una función llamada* `vh` *tal que* `vh(x)` *dé el vector unitario* `v` *correspondiente a* `x` *siguiendo la política de signos anterior.*

3. La descomposición QR

A pesar de que la terminología no sea muy esclarecedora, la descomposición QR de una matriz es casi equivalente a la ortonormalización de sus columnas. La Q se refiere a una matriz cuyas columnas son ortonormales (y, por tanto, si es cuadrada es ortogonal o unitaria) y R es una matriz triangular superior, quizá completada con ceros.

Si recordamos las fórmulas del proceso de Gram-Schmidt, obtendremos la Q y la R con unos ligeros añadidos en el código que almacenen los coeficientes en la R. En una función:

```
1  % Descomposición QR (reducida) con Gram-Schmidt
2  function [Q,R] = mQR(A)
3          m = size(A,2);
4          Q = A;
5          R = zeros(m);
6          R(1,1) = norm(A(:,1));  % nueva
7          Q(:,1) = A(:,1)/R(1,1);
8
9          for k = 2:m
10                 u_k = A(:,k);
11                 for j = 1:k-1
12                         R(j,k) = Q(:,j)' * A(:,k);  % nueva
13                         u_k = u_k - R(j,k) * Q(:,j);
14                 end
15                 R(k,k) = norm( u_k );  % nueva
16                 Q(:,k) = u_k/R(k,k);
17         end
18 end
```

Las líneas marcadas como nuevas son solo definiciones para que se almacene el valor de los elementos de R.

El comando nativo de `matlab/octave` para la descomposición QR es `qr`. Si ejecutamos

```
1 A = [-1,-5,-10; -2,2,-8; -2,-4,-5];
2
3 [Q,R] = qr(A)
4 [Q,R] = mQR(A)
```

veremos que, sobre este ejemplo, nuestra función y la de `matlab/octave` dan lo mismo. La unicidad para matrices cuadradas no singulares está asegurada

imponiendo que los elementos de la diagonal de R sean positivos.

Sin embargo, cuando usamos el ejemplo:

```
A = [1,1,2; 1,5,1; 1,1,1; 1,5,0];

[Q,R] = mQR(A)
[Q,R] = qr(A)
```

vemos una gran diferencia. Nuestra función da

$$Q = \frac{1}{2}\begin{pmatrix} 1 & -1 & 1 \\ 1 & 1 & 1 \\ 1 & -1 & -1 \\ 1 & 1 & 1 \end{pmatrix} \in \mathcal{M}_{4\times 3} \qquad \text{y} \qquad R = \begin{pmatrix} 2 & 6 & 2 \\ 0 & 4 & -1 \\ 0 & 0 & 1 \end{pmatrix} \in \mathcal{M}_{3\times 3}.$$

Es lo que en la teoría se ha llamado la descomposición QR *reducida* en la que R es cuadrada, pero Q no necesariamente.

La de `matlab/octave` produce siempre una Q cuadrada (ortogonal o unitaria), lo cual es más útil en las aplicaciones más importantes. Además, no impone $r_{ii} \geq 0$, lo cual afecta a la unicidad. Su resultado en el ejemplo es:

$$Q = \frac{1}{2}\begin{pmatrix} -1 & 1 & 1 & 1 \\ -1 & -1 & 1 & -1 \\ -1 & 1 & -1 & -1 \\ -1 & -1 & -1 & 1 \end{pmatrix} \in \mathcal{M}_{4\times 4} \quad \text{y} \quad R = \begin{pmatrix} -2 & -6 & -2 \\ 0 & -4 & -1 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{pmatrix} \in \mathcal{M}_{4\times 3}.$$

La eficiencia de nuestra función para matrices cuadradas no es muy buena en comparación con `qr` de `matlab/octave`. Consideremos

```
N = 100;
A = rand(N);

disp('Nuestra función')
tic
[Q,R] = mQR(A);
norm(Q*R-A)
toc

disp('')
disp('La de matlab/octave')
tic
[Q,R] = qr(A);
norm(Q*R-A)
toc
```

Tal como está, una ejecución en mi ordenador con `octave` dio `0.0798371` segundos con nuestra función basada en Gram-Schmidt y `0.00107002` con `qr`. Es decir, `qr` fue 70 veces más rápido. Al cambiar la primera línea a `N = 500` los tiempos pasaron a ser `1.98888` y `0.0308831` que todavía es 60

veces más rápido. Los errores cometidos al aproximar `Q*R` por `A`, que debieran ser iguales, son comparables, aunque resultaron menores con Gram-Schmidt.

ACTIVIDAD 5.3.1. *Haz un programa que muestre la gráfica del tiempo requerido por nuestra función dividido por el requerido por* `qr` *para una matriz aleatoria* $N \times N$ *cuando* N *varía entre* 100 *y* 500 *de* 10 *en* 10. *Por alguna razón los resultados son más exagerados en* `octave` *que en* `matlab`.

ACTIVIDAD 5.3.2. *Estudia qué ocurre con los autovalores de* `Q'*Q` *y de* `Q*Q'` *en nuestra descomposición* QR *cuando la matriz de partida es* $n \times m$ *con* $m < n$. *Como se ha mencionado en la teoría, el comando* `eig(A)` *devuelve los autovalores de la matriz* `A`.

Ahora vamos a implementar la descomposición QR con matrices de Householder. Para ello utilizamos la función `vh(x)` que daba un vector tal que la correspondiente matriz de Householder aplica `x` en un múltiplo de $\vec{e}_1$. El algoritmo consiste en ir considerando sucesivamente como vector `x` los elementos de la matriz por debajo o en la diagonal para cada una de las columnas, con ello la matriz de partida, supuesta cuadrada, se irá transformando en una triangular superior. En una función, esto es:

```
1  function [Q,R] = mQR2(A)
2          n = size(A,1);
3          Q = eye(n);
4          R = A;
5          for k = 1:n-1
6                  x = R(k:n,k);
7                  v = zeros(n,1);
8                  v(k:n) = vh(x);
9                  % dim Qt = n. Es la matriz de Householder
                       ↪ completada
10                 % con k-1 unos porque v(1:k-1) es nulo
11                 Qt = eye(n)-2*v*v';
12                 Q = Qt*Q;
13                 R = Qt*R;
14                 % Para que sea exactamente triangular
15                 R(k+1:n,k) =zeros(n-k,1);
16         end
17         % Esto es lo mismo que la inversa
18         % porque Q es ortogonal o unitaria
19         Q = Q';
20 end
```

Las líneas 12 y 13 son en realidad derrochadoras, porque solo necesitamos hacer esta operación para un bloque de la matriz y no necesitamos construir toda la `Qt`. Se deja como ejercicio, perfeccionar el código.

Si lo aplicamos a una matriz no cuadrada dará la descomposición QR *completa*, es decir, con Q cuadrada (ortogonal o unitaria), como el comando nativo `qr` de `matlab/octave`. De hecho, el siguiente código muestra que sobre la matriz no cuadrada considerada antes, se obtiene lo mismo:

```
1  A = [1,1,2; 1,5,1; 1,1,1; 1,5,0];
```

```
2  [Q,R] = qr(A)
3  [Q,R] = mQR2(A)
```

ACTIVIDAD 5.3.3. *Repite la comparativa anterior para matrices aleatorias $N \times N$ cuando N varía entre* 100 *y* 500 *de* 10 *en* 10*, pero ahora con la nueva función* `mQR2`.

4. Aplicación a la resolución de sistemas

Si $A = QR$ es la descomposición QR (completa) de una matriz no singular A, como Q es ortogonal o unitaria, se tiene $Q^{-1} = Q^\dagger$ y por tanto el sistema $A\vec{x} = \vec{b}$ equivale a $R\vec{x} = Q^\dagger\vec{b}$, el cual es muy sencillo porque R es triangular superior al que se puede aplicar sustitución regresiva, que es muy rápida. Por otro lado, calcular la descomposición QR no es gratis y la teoría dice que LU conllevará asintóticamente $2n^3/3$ para resolver un sistema genérico $n \times n$ y QR requerirá el doble. En realidad, tanto con LU como con QR la parte de sustitución regresiva o progresiva es menor en comparación con lo que tardan las descomposiciones. Por tanto, está más cerca de una confirmación experimental, un código del tipo:

```
1  N = 3000
2  A = rand(N);
3  b = rand(N,1);
4
5  tic
6  [L,U,P] = lu(A);
7  toc
8
9  tic
10 [Q,R] = qr(A);
11 toc
```

Realmente se obtienen de esta forma factores más grandes que dos, en `octave` mucho mayores que en `matlab`. No tengo explicación para ello.

ACTIVIDAD 5.4.1. *Modifica ligeramente el código de la actividad anterior para haga la comparativa de tiempos entre* `lu` *y* `qr` *cuando* `N` *está en el rango* `1000:100:3000`.

Recuerda que, si has sido obediente u ordenado, tienes implementadas unas funciones de días anteriores `ltrs` y `utrs` para resolver, con sustitución progresiva y regresiva, un sistema lineal con matriz triangular inferior y superior, respectivamente. Según lo dicho, la manera de resolver un sistema con QR sería del tipo:

```
1  N = 3
2  A = rand(N);
```

```
3  b = A*ones(N,1);
4
5  [Q,R] = qr(A);
6  x = utrs(R,Q'*b);
7
8  norm(x-ones(N,1))
```

El algoritmo en sí son las líneas 5 y 6. Las primeras fuerzan a que la solución tenga todas sus coordenadas uno y la última calcula el error, que para `N` pequeño debe ser comparable al épsilon máquina.

ACTIVIDAD 5.4.2. *Aunque sea una tarea retrospectiva, escribe un código que resuelva un sistema con la descomposición LU usando* `lu`. *Solo lo habíamos hecho para la función que habíamos construido nosotros. La única dificultad está en el uso de la matriz de permutación con los pivotes. Comprueba que funciona viendo que* `norm(x-A\b)` *es pequeño.*

Referencias

[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.