Actividades 3

Prácticas de Cálculo Numérico I (doble grado)

9 de marzo de 2021

1. Normas con `matlab/octave`

El comando básico para hallar normas en `matlab/octave` es `norm`. Aplicado a un vector (fila o columna), da la norma usual euclídea. Funciona tanto para vectores reales como complejos. Así,

```
1  v = [1, -2, 10, 4];
2  disp( norm(v) )
3  v = [1+3*i; 5+i];
4  disp( norm(v) )
```

produce 11 y 6 como salida.

La norma usual, y de hecho cualquier norma en $\mathbb{R}^n$ o $\mathbb{C}^n$, se extiende a una norma de matrices por medio de

$$\|A\| = \max_{\|\vec{x}\|=1} \|A\vec{x}\|.$$

No hace falta que la matriz A sea cuadrada, es decir, las dos normas del segundo miembro pueden ser en espacios de distintas dimensiones. El comando `norm` aplicado a matrices da este resultado. No es difícil demostrar [SB93, §4.4] que $\|A\|$ admite una "fórmula explícita", coincide con la raíz cuadrada del mayor autovalor de $A^\dagger A$ (que es real y no negativo) donde $A^\dagger$ es la traspuesta conjugada.

Actividad 3.1.1. *Sabiendo que* `eig(A)` *da la lista de autovalores de* `A`*, escribe un pequeño código que genere una matriz compleja al azar y comprueba la afirmación anterior.*

Respecto a normas no usuales, añadiendo un argumento más a `norm`, con `norm(v,p)` hallaremos la norma `p` de un vector `v`. Se permite `p=Inf` para indicar la norma infinito. Por ejemplo,

```
1  v = [1,2,-3,0];
```

```
2  disp( norm(v,1) )
3  disp( norm(v,4) )
4  disp( norm(v,Inf) )
```

ofrece como salida 6, $\sqrt[4]{98}$ (que está cerca de π) y 3. Por supuesto, `norm(v,2)` es lo mismo que `norm(v)`.

Por medio de la fórmula anterior, reemplazando $\|\cdot\|$ por $\|\cdot\|_p$, estas normas inducen unas correspondientes sobre las matrices, que se escriben en `matlab/octave` con el mismo comando, `norm(A,p)`. Es fácil ver [Atk89, §7.3] que las normas de matrices correspondientes a las normas $\|\cdot\|_1$ y $\|\cdot\|_\infty$ en $\mathbb{R}^n$ o $\mathbb{C}^n$ admiten las fórmulas

$$\|A\|_1 = \max_{1\le j\le n} \sum_{i=1}^{n} |a_{ij}| \qquad \text{y} \qquad \|A\|_\infty = \max_{1\le i\le n} \sum_{i=1}^{n} |a_{ij}|.$$

No es necesario que A sea cuadrada, es decir, la n puede ser distinta en ambas expresiones. Las siguientes líneas hacen una comprobación de estas fórmulas sobre una matriz de ejemplo:

```
1  A = [1,2;-3,-4;5,6*i];
2
3  no = norm(A,1)
4  am = max( sum(abs(A)) )
5  disp(['norm(A,1) = ' num2str(no) '. A mano = ' num2str(am)])
6
7  no = norm(A,Inf)
8  am = max( sum(abs(A')) )
9  disp(['norm(A,Inf) = ' num2str(no) '. A mano = '
          ↪ num2str(am)])
```

Hay también un comando especial `norm(A,'fro')` que da la *norma de Frobenius*. Esta es la norma usual cuando se recolocan los elementos de la matriz para formar un vector. Es decir, es la raíz cuadrada de la suma de los valores absolutos al cuadrado de los elementos.

ACTIVIDAD 3.1.2. *Para $\vec{x} \in \mathbb{R}^n - \{\vec{0}\}$, una desigualdad debida a F. Carlson afirma $\|\vec{x}\|^{-1}\|A\vec{x}\|^{-1}\big(\sum_{j=1}^{n} x_j\big)^2 \le \pi$ donde A es la matriz diagonal con $a_{ii} = i$. Toma $n = 10$ y escribe un código que genere 100000 vectores aleatorios con* `rand` *y busque el máximo valor del primer miembro para ellos. Intenta que vaya tan rápido como puedas.*

2. Ejemplos de aproximaciones sucesivas

La aproximación numérica de las soluciones de una ecuación se lleva a cabo habitualmente con métodos iterativos. Más adelante en el curso, trataremos estos métodos con más detalle. Aquí solo veremos un par de

ejemplos en una dimensión que nos den una idea como preparación al caso de sistemas de ecuaciones lineales.

Uno de los algoritmos más antiguos, quizá el que más, es un método iterativo para aproximar raíces cuadradas que responde a la fórmula:

$$x_{n+1} = \frac{1}{2}\Big(x_n + \frac{N}{x_n}\Big).$$

Aquí $N > 0$ es el número del que queremos calcular la raíz cuadrada. A pesar de la notación, no es necesario que sea un entero, aunque en este caso las aproximaciones son números racionales, lo que tiene algún interés. Partiendo de cualquier $x_0 > 0$ se tiene que $x_n \to \sqrt{N}$. Además la convergencia es bastante rápida. Por ejemplo, para $N = 2$, partiendo de $x_0 = 1$ que es la aproximación más obvia de $\sqrt{2}$, el siguiente código

```
format long
x = 1;
for k = 1:6
        x = (x+2/x)/2
end
```

produce

```
x =   1.50000000000000
x =   1.41666666666667
x =   1.41421568627451
x =   1.41421356237469
x =   1.41421356237309
x =   1.41421356237309
```

Lo cual es bastante espectacular, en seis iteraciones ya se ha estabilizado a un valor indistinguible de $\sqrt{2}$ habida cuenta el épsilon máquina.

ACTIVIDAD 3.2.1. *Busca una fórmula de recurrencia para el numerador y el denominador de las aproximaciones anteriores de $\sqrt{2}$ y escribe código que los calcule. Teniendo en cuenta que los enteros se almacenan en 64 bits, ¿hasta que iteración te deberías fiar de los resultados?*

Si damos por hecho que x_n, con $x_0 > 0$ converge (como es cierto), entonces no es difícil probar que $x_n \to \sqrt{N}$ porque tomando límites en la fórmula de recurrencia $x_n \to \ell$ implica $2\ell = \ell + N/\ell$ y la única solución positiva de esta ecuación es $\ell = \sqrt{N}$.

La idea es entonces que siempre que la convergencia no nos fastidie, con $x_{n+1} = f(x_n)$ tendremos una sucesión que tiende a una solución de $x = f(x)$. Sin embargo, la convergencia sí se estropea a menudo aunque a veces reescribir la ecuación en una forma equivalente, soluciona el problema. Veámoslo en un ejemplo muy tonto. Supongamos que queremos resolver iterando la ecuación lineal $x = 3x - 8$, que tiene obviamente como solución

$x = 4$. Despejando e intercambiando ambos miembros, podríamos reescribir la ecuación como $x = (x + 8)/3$. Resulta que en la primera forma, no da lugar a un método convergente para todo valor de partida pero en la segunda sí (esto está relacionado con que el coeficiente de x en el segundo miembro sea mayor o menor que 1). El siguiente código ilustra este punto calculando unas pocas iteraciones de $x_{n+1} = 3x_n - 8$ y de $x_{n+1} = (x_n + 8)/3$ con $x_0 = 1$.

```
x = 1;
y = 1;
for k = 1:5
        x = 3*x-8;
        y = (y+8)/3;
        disp(['primero -> ' num2str(x) ', segundo ->'
            ↪ num2str(y)])
end
```

La salida es:

```
primero -> -5, segundo ->3
primero -> -23, segundo ->3.6667
primero -> -77, segundo ->3.8889
primero -> -239, segundo ->3.963
primero -> -725, segundo ->3.9877
```

La situación para sistemas de ecuaciones lineales es similar y hay algunos métodos que reescriben el sistema de forma que sea más fácil que satisfaga la condición de convergencia. Los dos más famosos son el de Jacobi y el de Gauss-Seidel, que veremos a continuación. No son de uso general, hay infinidad de sistemas para los que no convergen y su interés se debe a que tienen éxito en otra infinidad con relevancia práctica [QSS07, §4.2].

ACTIVIDAD 3.2.2. *Para un ejemplo convergente $x_{n+1} = ax_n + b$, escribe un programa que dibuje las rectas $y = x$, $y = ax + b$ y la sucesión de puntos (x_0, x_1), (x_1, x_1), (x_1, x_2), (x_2, x_2), etc. unida con línea discontinua.*

3. El método de Jacobi

Ahora queremos resolver un sistema de ecuaciones lineales compatible determinado $A\vec{x} = \vec{b}$, con matriz cuadrada, y deseamos definir un esquema iterativo $\vec{x}^{(k+1)} = A'\vec{x}^{(k)} + \vec{b}'$ que converja a la solución. Como hablaremos a veces de coordenadas y utilizaremos n para el tamaño de la matriz, escribimos $\vec{x}^{(k)}$ en vez de $\vec{x}_n$, que sería más coherente con lo anterior.

En el método de Jacobi las iteraciones vienen dadas por:

$$\vec{x}^{(k+1)} = D^{-1}\big(\vec{b} - (A - D)\vec{x}^{(k)}\big)$$

donde $D = \mathrm{diag}(a_{11}, \ldots, a_{nn})$, es decir, es la matriz resultante al dejar solo la diagonal de A. Es un simple ejercicio comprobar que el método es consistente, es decir que si sustituimos $\vec{x}^{(k+1)}$ y $\vec{x}^{(k)}$ por la solución de $A\vec{x} = \vec{b}$, la ecuación es una identidad.

El método en coordenadas es:

$$x_i^{(k+1)} = a_{ii}^{-1}\Big(b_i - \sum_{j=1,\, j\neq i}^{n} a_{ij} x_j^{(k)}\Big), \qquad i = 1, 2, \ldots, n.$$

Tomemos como ejemplo el sistema $A\vec{x} = \vec{b}$ con

$$A = \begin{pmatrix} 5 & 2 & 1 \\ -3 & 6 & 1 \\ 2 & 3 & 5 \end{pmatrix}, \quad \vec{b} = \begin{pmatrix} 10 \\ 6 \\ 20 \end{pmatrix}, \quad \text{que tiene solución} \quad \vec{b} = \begin{pmatrix} 1 \\ 1 \\ 3 \end{pmatrix}.$$

En `matlab/octave` el comando `diag` aplicado a un vector, genera la matriz que tiene en la diagonal las coordenadas del vector y en el resto ceros. Cuando se aplica a una matriz, hace prácticamente lo contrario, devuelve el vector formado por la diagonal de la matriz. Por ejemplo

```
A = [5,2,1; -3,6,1; 2,3,5];
diag(A)
diag([1;1])
```

da lugar al vector columna $(5, 6, 5)$ y a la matriz identidad 2×2.

De esta forma, el $A - D$ anterior será `A - diag(diag(A))`. Valiéndonos de esto, en vez de programar el método con coordenadas, que sería lo más habitual, vamos a proceder con matrices. No le dejaremos calcular a `matlab/octave` la inversa D^{-1} para que no aplique algoritmos generales de la inversa que son lentos (en realidad "sabe" que es diagonal) y esa es la razón de definir `Dinv` en el siguiente código que aplica 5 iteraciones del método de Jacobi partiendo de $\vec{x} = \vec{0}$, en nuestro ejemplo:

```
N = 5;
A = [5,2,1; -3,6,1; 2,3,5];
b = [10;6;20];

n = size(A,1);
Ap = A - diag(diag(A));
Dinv = diag(A).^-1;

x = zeros(n,1);
for k = 1:N
        x = Dinv.*(b-Ap*x);
end
disp(x)
```

El resultado es (`0.99440, 1.01178, 2.98907`), lo cual no es una mala aproximación de la solución.

ACTIVIDAD 3.3.1. *En el código anterior pon una matriz* `A` *para la que el método no converja y haz que muestre las normas de las sucesivas iteraciones.*

En la práctica sería muy incómodo ir probando con diferentes números de iteraciones hasta conseguir la precisión requerida. Es más interesante establecer un número máximo de iteraciones y un criterio de parada del algoritmo. Hay varias políticas posibles. Aquí utilizaremos (siguiendo [MF99]) que el algoritmo pare cuando $\|\vec{x}^{(k+1)} - \vec{x}^{(k)}\|/\|\vec{x}^{(k)}\|$sea menor que cierta tolerancia establecida. Esto quiere decir que el error relativo al aproximar $\vec{x}^{(k)}$ por $\vec{x}^{(k+1)}$ es menor que dicha tolerancia, lo que sugiere que se están acercando a un límite.

```
Nmax = 100;
tol = 0.1;
A = [5,2,1; -3,6,1; 2,3,5];
b = [10;6;20];

n = size(A,1);
Ap = A - diag(diag(A));
Dinv = diag(A).^-1;

x = zeros(n,1);
for k = 1:Nmax
        x_old = x;
        x = Dinv.*(b-Ap*x);
        err = norm(x-x_old)/norm(x_old+eps);
        if err < tol
                disp(['Se han usado ',num2str(k),'
                        ↪ iteraciones'])
                disp('La aproximación a la solución es:')
                disp(x)
                break
        end
end
```

Tal como está, con `tol = 0.1`, este código nos informa que solo se han usado 4 iteraciones (de las `Nmax` de que disponemos) y la aproximación será (`1.03733, 0.99333, 3.04133`). Reduciendo la tolerancia a `tol = 1e-4`, se necesitan 10 iteraciones y la aproximación es (`1.00005, 0.99998, 3.00006`).

Si creamos una función `mjac` respondiendo al prototipo:

```
function [x,err] = mjac (A, b, Nmax, tol)
        ...
end
```

que devuelva la aproximación de la solución con el criterio de parada anterior, así como el último error relativo estimado, lo podemos combinar con el siguiente código para decidir automáticamente si estamos satisfechos con el resultado o no.

```
1  Nmax = 20;
2  tol = 1e-4;
3  A = [5,2,1; -3,6,1; 2,3,5];
4  %A = [5,2,1; -3,1,1; 2,3,1];
5  b = [10;6;20];
6
7  [x,err] = mjac(A, b, Nmax, tol);
8
9  if err<tol
10         disp('La aproximación a la solución es:')
11         disp(x)
12 else
13         disp('No se consigue una aproximación')
14         disp('con los parámetros de partida')
15         disp(err)
16         if err> 0.1
17                 disp('El error es muy grande,')
18                 disp('es posible que el método no converja')
19         end
20 end
```

Ejecutándolo tal como está, nos mostrará (`1.00005, 0.99998, 3.00006`), la aproximación antes mencionada. Si bajamos el número máximo de iteraciones con `Nmax = 5` nos avisará con el mensaje de las líneas 13–14. Si quitamos el `%` que comenta la línea 4 para activar la matriz con la que el método no converge, saltará el mensaje de las líneas 17–18.

ACTIVIDAD 3.3.2. *Escribe el código de la función* `mjac`.

Referencias

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

[MF99] J.H. Mathews and K.D. Fink. *Métodos numéricos con MATLAB*. Prentice Hall, 1999.

[QSS07] A. Quarteroni, R. Sacco, and F. Saleri. *Numerical mathematics*, volume 37 of *Texts in Applied Mathematics*. Springer-Verlag, Berlin, second edition, 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.