Actividades 4

Prácticas de Cálculo Numérico I (doble grado)

23 de marzo de 2021

1. El método de Gauss-Seidel

En este caso, la representación matricial es un poco liosa y de interés principalmente teórico [SB93, §8.2]. En términos de coordenadas, el método de Gauss-Seidel es:

$$x_i^{(k+1)} = a_{ii}^{-1}\left(b_i - \sum_{j=1}^{i-1} a_{ij}x_j^{(k+1)} - \sum_{j=i+1}^{n} a_{ij}x_j^{(k)}\right)$$

donde i varía de 1 a n.

Traduciendo la fórmula símbolo a símbolo a `matlab/octave` tenemos para el ejemplo de prueba anterior con 5 iteraciones el siguiente código. Se usa de nuevo `ii` en lugar de `i` por mi manía de no modificar la definición de la unidad imaginaria.

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

n = size(A,1);
x = zeros(n,1);

for k = 1:N
        for ii = 1:n
                x(ii) = (b(ii) - A(ii,1:ii-1)*x(1:ii-1) -
                    ↪ A(ii,ii+1:n)*x(ii+1:n)
                    ↪ )/A(ii,ii);
        end
end
disp(x)
```

Nótese que, en la fórmula que define el algoritmo, para $i = 1$ el primer sumatorio es vacío y para $i = n$, lo es el segundo. Deberíamos asegurarnos de que `matlab/octave` trata correctamente las sumas vacías como cero sin dar errores. El siguiente código lo ilustra:

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

```
x = [1;1;3];

rango = 1:3;
disp( A(1,rango)*x(rango) )

rango = 2:1;
disp( A(1,rango)*x(rango) )
```

Los resultados son 10 y 0. El segundo rango es vacío, si pedimos que nos los muestre, resultará `[](1x0)`.

Otra observación, es que se ha aprovechado la precedencia de las operaciones para escribir cosas como `x(1:ii-1)` en vez de la expresión más abigarrada equivalente `x(1:(ii-1))`. Los paréntesis son innecesarios porque primero se hace la resta `ii-1` y después se construye el vector.

ACTIVIDAD 4.1.1. *Escribe el código de una función* `mgase` *con una estructura similar a* `mjac` *pero con el algoritmo de Gauss-Seidel.*

2. Comparación numérica

Intuitivamente, el algoritmo de Gauss-Seidel debe ser mejor que Jacobi porque estamos actualizando parte del vector con coordenadas de la siguiente iteración que, en principio, son más precisas. Bajo ciertas condiciones sobre la matriz (que en particular aseguran la convergencia) se puede demostrar que es así [SB93, §8.2], aunque no hay un teorema que asegure esto en general sin condiciones distintas de la convergencia de los métodos. De hecho, como se ha visto en la teoría y repasaremos más adelante, es posible crear ejemplos especiales en que el método de Gauss-Seidel no converja para todas las condiciones iniciales mientras que Jacobi sí lo hace.

Vamos a comprobar sobre nuestro ejemplo la precisión de ambos métodos. Para aprovechar las funciones que `mjac` y `mgase` que hemos construido, utilizaremos el siguiente código. Es importante notar que esto sería absurdo (muy ineficiente) para matrices grandes. Si no ves por qué, se explica a continuación. Todavía tienes una líneas de margen para pensarlo por ti mismo.

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

for k = 1:10
        [x,err] = mjac(A, b, k, 0);
        e1 = norm(x-x_exact);
        [x,err] = mgase(A, b, k, 0);
        e2 = norm(x-x_exact);
        disp( ['k = ' num2str(k) ' -> J: '
            ↪ num2str(e1,'%.5e') ' -> G-S: '
            ↪ num2str(e2,'%.5e')]  )
```

```
11 end
```

La ineficiencia antes mencionada radica en que al llamar a las funciones con los argumentos `(A, b, k, 0)` estamos forzando a que haga `k` operaciones (porque la tolerancia se ha puesto a cero y nunca se alcanza) pero no estamos aprovechando el resultado para obtener la función evaluada en `k+1`, que solo requeriría un paso más.

Por si no lo recuerdas o no lo sabes, el argumento `'%.5e'` de `num2str` indica que se muestren 5 cifras decimales en formato científico exponencial.

El resultado es:

```
k = 1 -> J: 1.41421e+00 -> G-S: 1.73205e+00
k = 2 -> J: 5.57773e-01 -> G-S: 2.14580e-01
k = 3 -> J: 1.35483e-01 -> G-S: 5.07620e-02
k = 4 -> J: 5.60952e-02 -> G-S: 4.80372e-03
k = 5 -> J: 1.70180e-02 -> G-S: 1.42026e-03
k = 6 -> J: 5.53404e-03 -> G-S: 1.17154e-04
k = 7 -> J: 2.14433e-03 -> G-S: 3.79722e-05
k = 8 -> J: 5.10085e-04 -> G-S: 3.35890e-06
k = 9 -> J: 2.23854e-04 -> G-S: 9.70198e-07
k = 10 -> J: 7.92543e-05 -> G-S: 1.06250e-07
```

Una condición típica que asegura la convergencia de los métodos de Jacobi y Gauss-Seidel es la de ser *diagonalmente dominante por filas*, lo que significa que cada $|a_{ii}|$ sea mayor o igual que la suma de los valores absolutos del resto de los elementos de la fila i-ésima. Sabiendo que `rand` genera matrices aleatorias con elementos entre 0 y 1, la siguiente línea genera matrices aleatorias diagonalmente dominantes por filas de tamaño `N`

```
1 A = rand(n) + (n-1)*eye(n)
```

ACTIVIDAD 4.2.1. *Subsana la ineficiencia del programa anterior para comparar los métodos en el ejemplo de prueba y estudia su efecto sobre* $A\vec{x} = \vec{b}$ *con* A *una matriz aleatorias diagonalmente dominantes por filas grande generada con las líneas anteriores y* $\vec{b}$ *generado con* `rand(n,1)`.

Por ejemplo, al resolver la actividad anterior, en una prueba para `n = 100` se obtuvo

```
k = 1 -> J: 1.69636e-02 -> G-S: 9.10007e-03
k = 2 -> J: 8.42881e-03 -> G-S: 1.11833e-03
k = 3 -> J: 4.19373e-03 -> G-S: 6.91222e-05
k = 4 -> J: 2.08657e-03 -> G-S: 1.09428e-05
k = 5 -> J: 1.03817e-03 -> G-S: 1.14720e-06
k = 6 -> J: 5.16536e-04 -> G-S: 8.70231e-08
k = 7 -> J: 2.57000e-04 -> G-S: 1.47590e-08
k = 8 -> J: 1.27870e-04 -> G-S: 1.10525e-09
k = 9 -> J: 6.36210e-05 -> G-S: 1.35638e-10
k = 10 -> J: 3.16544e-05 -> G-S: 1.72836e-11
```

Por supuesto, como los números son aleatorios, el resultado no se puede replicar pero da una idea de la situación típica.

Ahora vamos con una situación atípica. Un estudio del radio espectral (hecho en la teoría) lleva a que la matriz

$$A = \begin{pmatrix} 1 & 1 & 2 \\ \beta & 1 & 1 \\ -5/2 & \beta^{-1} & 1 \end{pmatrix} \qquad \text{con} \quad \beta = \frac{5+\sqrt{21}}{2},$$

genéricamente da lugar a sistemas de ecuaciones lineales $A\vec{x} = \vec{b}$ para los que el método de Gauss-Seidel no converge mientras que el método de Jacobi lo hace para cualesquiera condiciones iniciales. Tomemos $\vec{b} = A(1,1,1)^t$ para que la solución sea $x = y = z = 1$ y veamos lo que ocurre con ambos métodos partiendo de $\vec{x} = \vec{0}$. El siguiente código, salvo cambios triviales, viene de concatenar códigos anteriores.

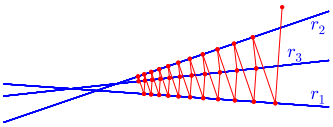
```
N = 40;

bet = (5+sqrt(21))/2;
A = [1,1,2; bet,1,1; -5/2,1/bet,1];
b = A*ones(3,1);

n = size(A,1);

% Jacobi
Ap = A - diag(diag(A));
Dinv = diag(A).^-1;
x = zeros(n,1);
for k = 1:N
        x = Dinv.*(b-Ap*x);
end
disp([num2str(N) ' iteraciones de Jacobi:'])
disp(x)

% Gauss -Seidel
n = size(A,1);
x = zeros(n,1);
for k = 1:N
        for ii = 1:n
                x(ii) = (b(ii) - A(ii,1:ii-1)*x(1:ii-1) -
                ↪ A(ii,ii+1:n)*x(ii+1:n)
                ↪ )/A(ii,ii);
        end
end
disp([num2str(N) ' iteraciones de Gauss-Seidel:'])
disp(x)
```

Cuando lo ejecutamos tal como está, con `N = 40`, se obtiene con el método de Jacobi `1.00037`, `1.00071`, `0.99972`, que está muy cerca de la solución. Sin embargo, el método de Jacobi produce las coordenadas astronómicas `-1.9115e+18`, `6.9862e+18`, `-6.2370e+18`.

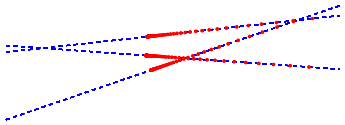
3. El algoritmo de Kaczmarz

Un algoritmo iterativo conceptualmente muy sencillo para resolver sistemas de ecuaciones lineales consiste en considerar cada una de las ecuaciones

que lo componen como un hiperplano e ir haciendo proyecciones sucesivas en cada uno de ellos. Si no has entendido esta frase, seguro que la primera de las siguientes figuras, correspondiente a dos incógnitas y por tanto hiperplanos que son rectas, te da alguna luz.



Tres ecuaciones, dos incógnitas



Iteraciones en la zona central

La segunda figura muestra que es para sistemas que sean "ligeramente incompatibles" se obtiene algo que va oscilando entre soluciones aproximadas.

En un sistema real $m \times n$ la m-ésima ecuación es el hiperplano $\vec{a}_i \cdot \vec{x} = b_i$ donde $\vec{a}_i$ es la i-ésima fila considerada como vector columna y este vector es normal al hiperplano. Proyectar sobre el hiperplano es lo mismo que quitar la componente que está sobre $\vec{a}_i$ y si recordamos que la proyección de un vector sobre otro unitario venía dada por el producto escalar, no debería extrañarnos la fórmula del *algoritmo de Kaczmarz*

Para sistemas de números reales $m \times n$, la fórmula es:

$$\vec{x}^{(k+1)} = \vec{x}^{(k)} + \left(b_i - \vec{a}_i \cdot \vec{x}^{(k)}\right) \frac{\vec{a}_i}{\|\vec{a}_i\|^2}$$

donde i es el resto al dividir k por m más uno. Esta elección de i es solo una forma de decir que se van recorriendo cíclicamente todas las filas.

Para sistemas complejos hay que hacer algún pequeño cambio introduciendo conjugados porque resulta que el vector normal al hiperplano $a_1x_1 + \ldots a_nx_n = b$ en $\mathbb{C}^n$ no es exactamente el dado por sus coeficientes sino por sus coeficientes conjugados.

Copiando el algoritmo tal como está, tenemos el siguiente código que lo aplica a un sistema de prueba 3×3 con solución $(1, 1, 2)$.

```
% Número iteraciones
N = 100
% Sistema
A = [2,2,1; 8,4,1; 4,7,-4];
b = [6; 14; 3];

% número de filas
m = size(A,1);
% número de comunas
n = size(A,2);

% Vector inicial
```

```
13  x = zeros(n,1);
14
15  for k = 1:N
16          ii = mod(k,m) +1;
17          fila = A( ii , : );
18          x = x + (b(ii) - fila*x)*fila'/norm(fila)^2;
19  end
20
21  disp(x)
```

Con las 100 iteraciones indicadas se obtiene $(\texttt{1.00222}, \texttt{0.99659}, \texttt{1.99589})$, mientras que con 25 iteraciones dan $(\texttt{1.12047}, \texttt{0.81596}, \texttt{1.77239})$, lo que difícilmente nos haría sospechar la solución real. Cabe entonces preguntarse qué interés tiene un algoritmo que ya es lento en un sistema 3×3. Su gran virtud es su sencillez y su aplicabilidad a sistemas de dimensiones increíblemente grandes, intratables por otros métodos que sean *dispersos*, esto es, con una matriz con muchos ceros, porque en ese caso $\vec{a}_i \cdot \vec{x}^{(k)}$ y $\|\vec{a}_i\|^2$ típicamente conllevarán muy poco esfuerzo computacional.

Un ejemplo práctico en el que se ha aplicado, aunque no es el método más común, es en tomografía axial computarizada (TAC). Allí cada uno de los rayos X atraviesa solo una proporción muy pequeña de los píxeles tridimensionales (*voxels*) de la muestra, lo que se relaciona con que cada ecuación contenga muy pocas incógnitas [Cha04]. Por otro lado, la información es muy redundante y por tanto nos enfrentamos a un sistema teóricamente incompatible.

Es fácil percatarse de que el código anterior no está muy optimizado: cada m veces estamos pidiendo que calcule la misma norma. Una solución para no hacerlo sería normalizar las ecuaciones de manera que cada fila de la matriz tuviera norma 1. Una primera solución es hacer un bucle previo con este fin. En las versiones modernas de `matlab/octave` hay comandos como `vecnorm` y `normr` que nos pueden ayudar a simplificar y acelerar el proceso. Por otro lado, si en `v` almacenamos los inversos de las normas, en las versiones modernas `v.*A` no da error para multiplicar cada fila por el factor normalizante. En versiones antiguas la sintaxis es más rígida y requiere cosas como `bsxfun(@times, v.', A)`.

ACTIVIDAD 4.3.1. *Aplica alguna de las soluciones sugeridas u otra que se te ocurra para que* `norm(fila)` *no aparezca en el código anterior.*

Referencias

[Cha04] F. Chamizo. Modelización II (un pase de modelos). `http://matematicas.uam.es/~fernando.chamizo/libreria/libreria.html`, 2004.

[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.