Proyecto #2: Programación avanzada en Matlab. Métodos iterativos para sistemas de ecuaciones lineales

Víctor Domínguez María Luisa Rapún
http://www.unavarra.es/personal/victor_dominguez/

19 de octubre de 2005

Capítulo 1

Introducción

> *"Well, in our country" said Alice, still panting a little, "You'd generally get to somewhere else–if you ran very fast for a long time as we've been doing."*
> *"A slow sort of country!" said the Queen. "Now, here, you see, it takes all the running you can do, to keep in the same place. If you want to get somewhere else, you must run at least twice as fast as that."*
>
> Lewis Carroll, Through the Looking Glass

En el primer apartado entraremos en aspectos más avanzados sobre vectores y matrices en Matlab haciendo un hincapié especial en la manipulación de matrices sparse. Veremos también cómo se pueden implementar funciones donde el número de argumentos de entrada y salida son variables.

En la parte matemática, tocaremos la teoría básica de métodos iterativos para sistemas de ecuaciones lineales. Empezaremos con los métodos clásicos: Jacobi, Gauss-Seidel y Relajación, para pasar luego a métodos más modernos y elaborados: el método del Gradiente y especialmente, el método del **Gradiente Conjugado**.

En este proyecto nos permitiremos hacer *algo* de Matemáticas. Animamos al lector no se *asuste* por ello y que trate de entender los enunciados y las demostraciones que se ofrecen. Para ello se asume unos conocimientos mínimos en Álgebra Lineal. En cualquier caso, e independiente de estos resultados teóricos, los algoritmos y ejercicios presentados en la segunda parte proporcionan una exposición básica sobre los métodos iterativos para sistemas lineales, tanto clásicos, como los más modernos de los que el Gradiente Conjugado es seguramente su representante más brillante.

Parte I

Matlab: programación avanzada

Capítulo 2

Las matrices revisitadas

A estas alturas el lector ya debería estar convencido sobre las capacidades de Matlab en lo que se refiere al manejo de enormes cantidades de memoria. En esta sección expondremos algunos comandos adicionales y entraremos con cierto detalle en la manipulación de matrices *sparse*[1].

2.1. Acceso a partes estructuras de una matriz

En ocasiones es importante tomar *partes* precisas de una matriz que no son necesariamente partes de filas, columnas o simplemente submatrices (estas acciones ya se trataron en el Proyecto #1). Para este fin nos pueden servir las siguientes instrucciones

```
diag     triu     tril
```

La primera toma la diagonal de una matriz, la segunda y tercera la parte triangular superior (*upper*) e inferior (*lower*) respectivamente. Sin embargo los comandos son algo más flexibles de lo que pueda parecer a simple vista.

Empecemos introduciendo una matriz

```
>> a=[11 12 13; 21 22 23; 31 32 33];
```

El resultado de los siguientes comandos es, a la luz de lo anterior, esperable

```
>> diag(a)

ans =

 11
 22
 33

>> triu(a)
```

[1] Aceptaremos este anglicismo en lo que sigue. En ocasiones este vocablo se traduce por *matrices huecas* o *matrices dispersas*.

```
ans =

 11    12    13
  0    22    23
  0     0    33

>> tril(a)

ans =

 11     0     0
 21    22     0
 31    32    33
```

El usuario puede especificar qué diagonal toma, o a partir de qué diagonal se toma la matriz triangular. La diagonal principal es la cero, subdiagonales inferiores (respectivamente superiores) se numeran consecutivamente con números enteros negativos (respectivamente positivos). El siguiente ejemplo ilustra estas características.

```
>> diag(a,-1)

ans =

    21
    32

>> tril(a,-1)

ans =

     0     0     0
    21     0     0
    31    32     0

>> triu(a,0)

ans =

     11    12    13
     0     22    23
     0     0     33
```

Ejercicio 1 *Programar una función que dada una matriz* `a` *devuelva un vector* `d` *con los elementos de la diagonal, una matriz triangular superior* `u` *con todos los elementos de* `a` *situados encima de la diagonal superior y* `l` *una matriz triangular inferior con todas las entradas de debajo de la diagonal principal*

Ejercicio 2 *¿Qué sucede si aplicamos los comandos anteriores a matrices rectangulares?*

Con `diag` podemos también construir a partir de un vector una matriz diagonal

```
>> diag([1 2])  % es lo mismo que diag([1 2],0)

ans =

     1     0
     0     2

>> diag([1 2],1)

ans =

     0     1     0
     0     0     2
     0     0     0

>> diag([1 2],-1)

ans =

     0     0     0
     1     0     0
     0     2     0
```

Obsérvese lo que sucede cuando se aplica dos veces el comando `diag`.

```
>> a=[1 2 3; 4 5 6; 7 8 9];
>> diag(diag(a))

ans =

     1     0     0
     0     5     0
     0     0     9
```

Trasposición de matrices

Dada una matriz A, la matriz transpuesta $A^\top$ es la matriz resultante de intercambiar filas y columnas de A. Esto es, las filas de A pasan a ser las columnas de $A^\top$. Esta operación se denota en Matlab con "'":

```
>> a=[1 2 3; 0 2 4];
>> a'

ans =
```

```
     1     0
     2     2
     3     4
```

Obviamente, también se aplica sobre vectores:

```
>> b=[1;2;3]; % vector COLUMNA con tres elementos
>> b'  % vemos ahora un vector FILA

ans =

    1   2   3
```

De nuevo nos encontramos con esta propiedad sobre la que ya hemos incidido: Matlab distingue entre vectores filas y columnas, y esta diferencia se hace especialmente palpable (y en ocasiones molesta) a la hora de realizar operaciones como productos matriz-vector.

Nota. En Matlab existe también el operador "`.'`". Sobre matrices reales funciona exactamente igual que el comando anterior, pero no así sobre números complejos. Mientras además de trasponer "`'`" conjuga todas las entradas[2], "`.'`" se limita a cambiar la forma de la matriz:

```
>> clear i % i es ahora la unidad imaginaria
>> a=[i 1-i; 1+i i];
>> a.'

ans =

    0 + 1.0000i        1.0000 + 1.0000i
    1.0000 - 1.0000i        0 + 1.0000i

>> a'

ans =

    0 - 1.0000i        1.0000 - 1.0000i
    1.0000 + 1.0000i        0 - 1.0000i
```

□

Cuando sobre una matriz `a` utilizamos `a(:)` obtenemos un **vector columna** construido concatenando las columnas de `a`. Técnicamente hablando, nos está mostrando la matriz tal como se guarda en memoria. Por ejemplo,

```
>> a=[1 2 3; 0 2 4];
>> a(:)
```

[2] es decir, cambia el signo a la parte imaginaria de cada elemento. Matemáticamente hablando, éste es el operador de transposición o conjugación, denotado habitualmente en matemáticas con "A^*".

```
ans =

     1
     0
     2
     2
     3
     4

>> a=a'; % (trasponemos a)
>> a(:)

ans =

    1
    2
    3
    0
    2
    4
```

Esto puede utilizarse para hacer un par de "trucos":

- En ocasiones la entrada de una función es un vector, sin importar si éste es fila o columna[3]

  Si deseamos pasarlo a columna para así manipularlo a nuestro antojo podemos utilizar

  ```
  >> b=b(:);
  ```

  que hará de `b` un vector columna, sea cual sea su formato inicial. Si lo que se desea es un vector fila, basta con transponer

  ```
  >> b=b(:)'; % b=b(:).' mejor por si es complejo
  ```

- Se puede utilizar para introducir las entradas de una matriz por columna. A modo de ejemplo, obsérvese

  ```
  >> a=zeros(4,3);
  >> a(:)=[1 2 3 4 5 6 7 8 9 10 11 12]

  a =
  ```

[3]Por ejemplo, si implementamos una función con el método de Gauss, deberíamos exigir que el término independiente fuera un vector columna.

```
    1     5     9
    2     6    10
    3     7    11
    4     8    12

>> a2=zeros(2,6);

>> a2(:)=a(:)

a2=

    1     3     5     7     9    11
    2     4     6     8    10    12
```

Nota. El comando `reshape` permite modificar las dimensiones de una matriz (o array en general). Es más flexible que el comando ":".

2.2. Más operaciones sobre matrices

Hasta ahora las operaciones matriciales que han centrado nuestra atención son las fundamentales: suma y producto. En Matlab están también implementadas otras operaciones comunes en el Álgebra Lineal. Destacamos:

`dot`: Calcula el producto escalar de dos vectores:

```
>> dot([1 2 3],[4 5 6])

ans =

    32
```

Devuelve el produto $1\cdot 4+2\cdot 5+3\cdot 6=32$. Este comando no distingue entre vectores filas y columnas, y es aplicable siempre que tengan la misma longitud.

La función se puede aplicar a matrices bien por columnas, ésta es la forma estándar[4] o por filas.

```
>> a=[1 2 3; 4 5 6]; a2=[1 1 1; 1 1 1];
>> dot(a,a2) % producto por columnas

ans =

    5     7     9
```

[4]Recuérdese la *predilección* de Matlab por las columnas.

```
>> dot(a,a2,2) % producto por filas

ans =

    6
    15
```

`sum`: calcula la suma de un vector. Es aplicable también a matrices, en el sentido del comando anterior

```
>> v=[1 2 3]; a=[1 2 3; 4 5 6];
>> sum(v)

ans =

     6

>> sum(a)

ans =

     5     7     9

>> sum(a,2)

ans =

     6
    15
```

`kron`: producto tensorial de Kronecker.

`norm`: norma de una matriz o vector. Se puede escoger entre varias normas.

```
>> v=[1 2 3];a=[1 2; 3 4];
>> [norm(v) norm(v,1) norm(v,inf)] % norma 2, 1 e infinito de v

ans =

     3.7417    6.0000    3.0000

>> [norm(a) norm(a,1) norm(a,inf)]

ans =

       5.4650    6.0000    7.0000
```

En la sección 5.3.1 comentaremos brevemente la definición de estas normas.

`rank`: rango *numérico* de una matriz[5].

`cond`: condicionamiento de una matriz. Es equivalente a calcular `norm(a)*norm(inv(a))` y da una medida de la sensibilidad del sistema a perturbaciones en el término independiente. También es importante en la convergencia de multitud de métodos iterativos.

`rcond`: estimador del inverso del condicionamiento de una matriz. Es sensiblemente más económico de calcular que `cond`

Nota sobre `dot`. Cuando el comando se aplica a dos vectores complejos, procede siempre a **conjugar el primer vector**. Es decir, matemáticamente

$$\texttt{dot(u,v)} \quad \rightsquigarrow \quad \sum_{i=1}^{n} \overline{u}_i v_i.$$

Así,

```
>> u=[1+i 2+2i]; v=[2 1];
>> dot(u,v)

ans =

   4.0000 - 4.0000i

>> dot(v,u)

ans =

   4.0000 + 4.0000i
```

2.3. Matrices sparse

Las matrices sparse son un importante subtipo de matrices que surgen en diferentes ámbitos del Análisis Numérico y de las Matemáticas en general (elementos finitos, teoría de grafos,...).

En la figura 2.1 se puede ver un ejemplo de una matriz sparse simétrica donde los puntos indican las entradas diferentes de cero. Desde una óptica puramente computacional hace falta desarrollar sistemas de almacenamiento especiales dado que la inmensa mayoría de las entradas no deben ser almacenados porque son nulas.

Matlab provee de forma muy sencilla de ese almacenamiento. Con

[5]Esto es, el número máximo de filas o columnas linealmente independientes. Una matriz tiene, en general, rango máximo por los errores de precisión de la máquina. Este comando hace una estimación del rango, eliminando este factor.

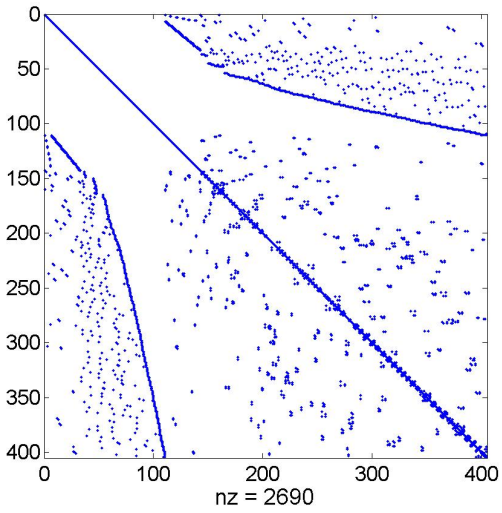


Figura 2.1: Matriz sparse 400×400 con 2690 entradas no nulas

```
>>a=sparse(100,100); b=sparse(100,1);
```

declaramos `a` como una matriz sparse 100×100 y un vector columna de 100 términos. Todas las entradas son inicialmente ceros, pero en `a` se guarda la estructura básica para introducir los elementos no nulos

```
>> a=sparse(100,100)

a =

   All zero sparse: 100-by-100

>> a(4,4)=1; a(8,9)=-4; a(80,45)=-1; a(99,100)=4;
>> a

a =

   (4,4)        1
   (8,9)       -4
  (80,45)      -1
  (99,100)      4
```

Cuando se aplica a una matriz llena la transforma en una matriz sparse

```
>> a=diag([1 2 3 4 5])
a =

     1     0     0     0     0
```

```
     0     2     0     0     0
     0     0     3     0     0
     0     0     0     4     0
     0     0     0     0     5

>> a=sparse(a)

a =

  (1,1)        1
  (2,2)        2
  (3,3)        3
  (4,4)        4
  (5,5)        5
```

Con `full` realizamos la operación inversa: transforma una matriz sparse en una matriz llena, convencional

```
>> a=full(a)

a =

     1     0     0     0     0
     0     2     0     0     0
     0     0     3     0     0
     0     0     0     4     0
     0     0     0     0     5
```

Se puede obtener el esquema de una matriz sparse (*pattern*), como el que aparece en la Figura 2.1, con el comando `spy`:

```
>> a=sparse(10,10);
>> a(1,4)=1; a(2,8)=-4; a(3,9)=1; a(7,8)=5;
>> spy(a)
```

La gráfica se muestra en una ventana separada[6].

El resto de operaciones que hemos visto están adaptados al nuevo entorno. Así, los operadores

```
:      triu      tril      diag
```

devuelven matrices sparse. Las operaciones

```
*      +      .*      dot      '
```

[6]Ésta es, sorprendentemente, nuestra primera salida gráfica en Matlab.

están asimismo optimizadas[7].

Por otro lado, aparecen una nueva serie de comandos, todos ellos devuelven matrices sparse, entre los que merece la pena destacar

`spdiags` análoga a `diag`;

`speye` devuelve una matriz diagonal, o similar (análoga a `eye`;

`spones` análoga a `ones`;

`sprand, sprandn` construye una matriz sparse con entradas aleatorias (similar a `rand`);

Ejercicio 3 *Con la ayuda de* Matlab*, deduce la sintaxis concreta de los comandos anteriores y comprueba con algún ejemplo cómo funcionan.*

[7]La forma de generar código eficiente no es tan simple como pudiera parecer. Por ejemplo, si se aplica la función `dot` a dos vectores sparse es preciso saber antes qué entradas hay que multiplicar. De otra forma podríamos estar dedicando un alto porcentaje de nuestro esfuerzo en simplemente calcular productos por cero. Afortunadamente, Matlab hace ese trabajo por nosotros.

Capítulo 3

Programación de funciones

Veremos a continuación cómo se pueden programar funciones en el que tanto el número de argumentos de entrada como de salida sean variables. Esta característica dota de una mayor flexibilidad a la programación en Matlab.

Los comandos esenciales que precisamos son

```
varargin      nargin      varargout      nargout
```

La instrucciones `nargin` y `nargout` informan respectivamente sobre el número de variables de entrada y el número de variables de salida (*var*iable *in*put *arg*ument) y *var*iable *out*put *arg*ument).

La función puede aceptar una serie de argumentos fijos, al estilo de las funciones que programamos en el proyecto anterior, y un conjunto de argumentos opcionales, que pueden ser o no especificados por el usuario. Para acceder a se utilizan respectivamente `varargin` (*var*iable *in*put *arg*ument) y `varargout` *var*iable *out*put *arg*ument) mediante llaves `{ }`.

Mostramos a continuación un ejemplo de utilización conjunta de estas nuevas instrucciones. En la cabecera se informa de qué hace la función, dependiendo de los argumentos de entrada y salida.

```
01    % MILU
02    %
03    % [L,U]   = MILU(A)   Devuelve U triang superior, L permutacion de
04    %                      una triang inferior con 1s en la diagonal tal
05    %                      que A=LU
06    % [L,U,X] = MILU(A,B) Devuelve U triang superior, L permutacion de
07    %                      una triang.inferior con 1s en la diagonal tal
08    %                      que A=LU y la solucion  del sistema AX=B
09
10    function varargout=milu(a,varargin)
11
12    [l,u]=lu(a);        % descomposicion LU
13
14    if nargin==1 & nargout==2
15        varargout{1}=l;
16        varargout{2}=u;
```

```
17    elseif nargin==2 & nargout==3
18        b=varargin{1};  % leemos el primer argumento opcional...
19        b=b(:);         % y lo transformamos en un vector columna
20        varargout{1}=l; varargout{2}=u;
21        varargout{3}=u\(l\b);  % solucion del sistema
22        end
23    end
```

Como puede comprobarse, la función precisa de un argumento obligatorio, la matriz `a` y devuelve dos o tres argumentos según se requiera. Obsérvese los resultados que se han obtenido[1].

```
>> a=[1 3; 2 4];
>> [l,u]=milu(a)

l =

    0.5000    1.0000
    1.0000         0

u =

     2     4
     0     1

>> [l,u,x]=milu(a,[1 2])

l =

    0.5000    1.0000
    1.0000         0

u =

     2     4
     0     1

x =

     1
     0

>> [l,u]=milu(a,[1 2]) %falta un arg. de salida
```

[1] Observa como la línea `19` permite introducir un vector fila como término independiente. El programa lo pasa a columna automáticamente.

```
??? Error using ==> milu
Too many output arguments.

>> [l,u,x]=milu(a) %faltan arg. de entrada
??? Error using ==> milu
Too many output arguments.
```

Ejercicio 4 *A partir de la función anterior implementar una función que opera según la "siguiente cabecera"*

```
% MILU2
%
% R      = MILU2(A)    Si A es simetrica definida positiva devuelve
%                      R triang superior tal que A=R'R
% [R,X]  = MILU2(A,B)  Si A es simetrica definida positiva devuelve
%                      R triang superior tal que A=R'R y la solucion
%                      del sistema AX=B
% [L,U]  = MILU2(A)    Devuelve U triang superior, L permutacion de
%                      una triang. inferior con 1s en la diagonal
%                      tal que A=LU
% [L,U,X]= MILU2(A,B)  Devuelve U triang superior, L permutacion de
%                      una triang. inferior con 1s en la diagonal
%                      tal que A=LU y la solucion del sistema AX=B
```

Nota: Realizar la comparación `A'==A` para testar si la matriz es simétrica. ¿Qué devuelve esta comparación? ¿Cómo se puede utilizar para ver si efectivamente la matriz es simétrica?. ¿Qué otras posibilidades podrías utilizar?..

Parte II

Matrices sparse en Matemáticas. Métodos iterativos para sistemas de ecuaciones lineales

Capítulo 4

Matrices sparse en Matemáticas

Este capítulo sirve para ilustrar algunos usos del almacenamiento sparse. En la primera sección tratamos el método de Gauss para este tipo de matrices y revisamos de nuevo el problema del *relleno* y cómo se puede minimizar con reordenamientos adecuados. Sirve asimismo para ahondar en la problemática de los métodos directos y allanar y fundamentar los métodos iterativos que se tratan en los siguientes capítulos.

4.1. Método de Gauss con matrices sparse

Recordemos que en Matlab la resolución de sistemas de ecuaciones utilizando

```
>> x=a\b;
```

Puede plantearse la cuestión de si es adecuado utilizar "\" para matrices con formas especiales, como por ejemplo, matrices triangulares o permutaciones de éstas, para las que la eliminación gaussiana no es necesaria puesto que las variables se pueden despejar de forma progresiva. Afortunadamente Matlab reconoce alguna de las estructura básicas y resuelve de una manera optima el sistema[1].

Ya hemos visto que el método de Gauss es matemáticamente equivalente a calcular dos matrices L y U triangular inferior y superior respectivamente con $1s$ en la diagonal de L de forma que

$$A = LU.$$

En este caso, si se dispone de esta descomposición, el sistema se reduce a la resolución de dos sistemas triangulares,

$$A\mathbf{x} = \mathbf{b}, \qquad \rightsquigarrow \qquad L\mathbf{y} = \mathbf{b}, \quad U\mathbf{x} = \mathbf{y}.$$

o en Matlab[2]

[1] Si se consulta la ayuda para este comando se puede leer que se reconocen tipos diferentes de matriz. Por ejemplo, sparse o llena, simétrica o no simétrica, triangular o permutación de ésta, bandeada (los elementos concentrados en torno a la diagonal), de Hessemberg (con todos los elementos por debajo de la subdiagonal principal nulos),... y aplica el método directo más conveniente en cada caso. Testar si una matriz pertenece a uno de estos tipos se puede realizar en un número de operaciones despreciable respecto a las del método de Gauss.

[2] La colocación de los paréntesis es esencial. Con `u l b` se **calcula primero** el producto $\mathtt{u}^{-1}\mathtt{l}$, que es más costoso.

```
>> [l,u]=lu(a);
>> x=u\(l\b);
```

Los dos sistemas triangulares se resuelve en $\mathcal{O}(2n^2)$ operaciones mientras que la descomposición, precisa de $\mathcal{O}(4n^3/3)$. Ahora bien, esta descomposición se puede utilizar reiteradamente con diferentes términos independientes.

Una característica muy habitual del método de Gauss para matrices sparse es el efecto *relleno* (en inglés *fill in*), esto es, la inserción de nuevos elementos en el proceso. Por ejemplo, tras un único paso del método de Gauss

$$\begin{bmatrix} x & x & x & x & x & x \\ x & x & & & & \\ x & & x & & & \\ x & & & x & & \\ x & & & & x & \\ x & & & & & x \end{bmatrix} \sim \begin{bmatrix} x & x & x & x & x & x \\ & x & x & x & x & x \\ & x & x & x & x & x \\ & x & x & x & x & x \\ & x & x & x & x & x \\ & x & x & x & x & x \end{bmatrix}$$

y la matriz es ahora llena. Sin llegar a esos extremos, es habitual que las necesidades de memoria se multipliquen al aplicar la eliminación gaussiana. En la descomposición LU esto se refleja en un incremento en el número de entradas no nulas de la matriz con los problemas que ello lleva consigo (mayores requerimientos de memoria, mayor número de operaciones,...).

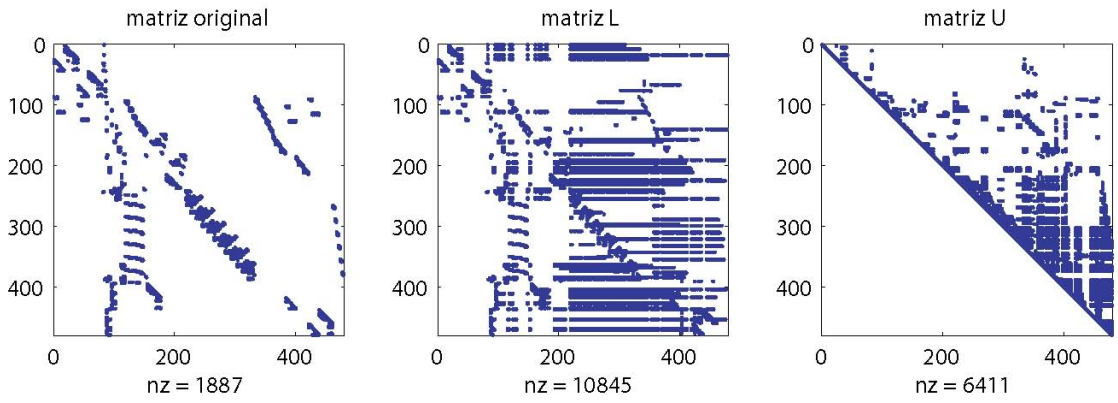


Figura 4.1: Efecto relleno.

La figura 4.1 es un buen ejemplo de este comportamiento. Se puede obtener mediante las siguientes instrucciones en Matlab (que se pueden recoger para mayor comodidad en un fichero script)

```
load 'west0479' % matriz sparse de ejemplo en Matlab
a=west0479;
[l,u]=lu(a);
close all
subplot(131)
```

```
spy(a)
title('matriz original')
subplot(132)
spy(l)
title('matriz L')
subplot(133)
spy(u)
title('matriz U')
```

Puede observarse que el número de entradas necesarias para guardar la descomposición es sensiblemente superior a las entradas de la matriz original. Una forma de atenuar este efecto es reordenar adecuadamente las ecuaciones e incógnitas de la matriz de forma apropiada.

Dado que en aplicaciones prácticas las matrices simétricas son muy comunes[3] nos restringiremos en lo que sigue a esta familia de matrices. Por tanto, se plantea la necesidad de reordenar la matriz para minimizar el efecto relleno de una forma tal que la matriz continúe siendo simétrica. Es por ello necesario permutar, tanto las filas (ecuaciones) como las columnas incógnitas.

En los comandos

`symrcm` `symmmd`

están implementados dos algoritmos de reordenación muy populares: el algoritmo **inverso de Cuthill-McKee** y el algoritmo de **mínimo grado**.

Este tipo de técnicas tuvieron su origen en la resolución de ecuaciones en derivadas parciales mediante métodos numéricos. Un buen ejemplo lo encontramos en la figura 4.2 obtenido al aplicar estas dos reordenaciones a una matriz proveniente de la resolución de la ecuación de Laplace por elementos finitos. Observa como se reduce el número de entradas en las matrices L y U cuando se aplica alguno de los algoritmos anteriores.

En cualquier caso, debe recordarse que **la descomposición LU debe calcularse sin pivotaje**, lo que se consigue con el comando[4]

```
>> [l,u]=lu(a,0);
```

En cualquier caso, la solución del (o de los) sistemas se lleva a cabo las siguientes órdenes

Solución con reordenamiento...

[3] O al menos matrices con estructura simétrica. Esto es, $a(i,j) \neq 0 \Leftrightarrow a(j,i) \neq 0$. Todo lo que sigue es igualmente válido para este tipo de matrices.

[4] El segundo argumento, que sólo está disponible si la matriz `a` es sparse, fija un umbral para la realización del pivotaje parcial. El valor 1 es el de defecto. De esta forma, valores próximos a cero obligan a que el pivotaje se produzca sólo cuando el desequilibrio entre el pivote y el resto de elementos de la columna sea muy acusado. Surge ahora la cuestión de estabilidad del método sin pivotaje. Recuérdese no obstante que para matrices definidas positivas o diagonal dominantes, el método de Gauss sin pivotaje es estable numéricamente.

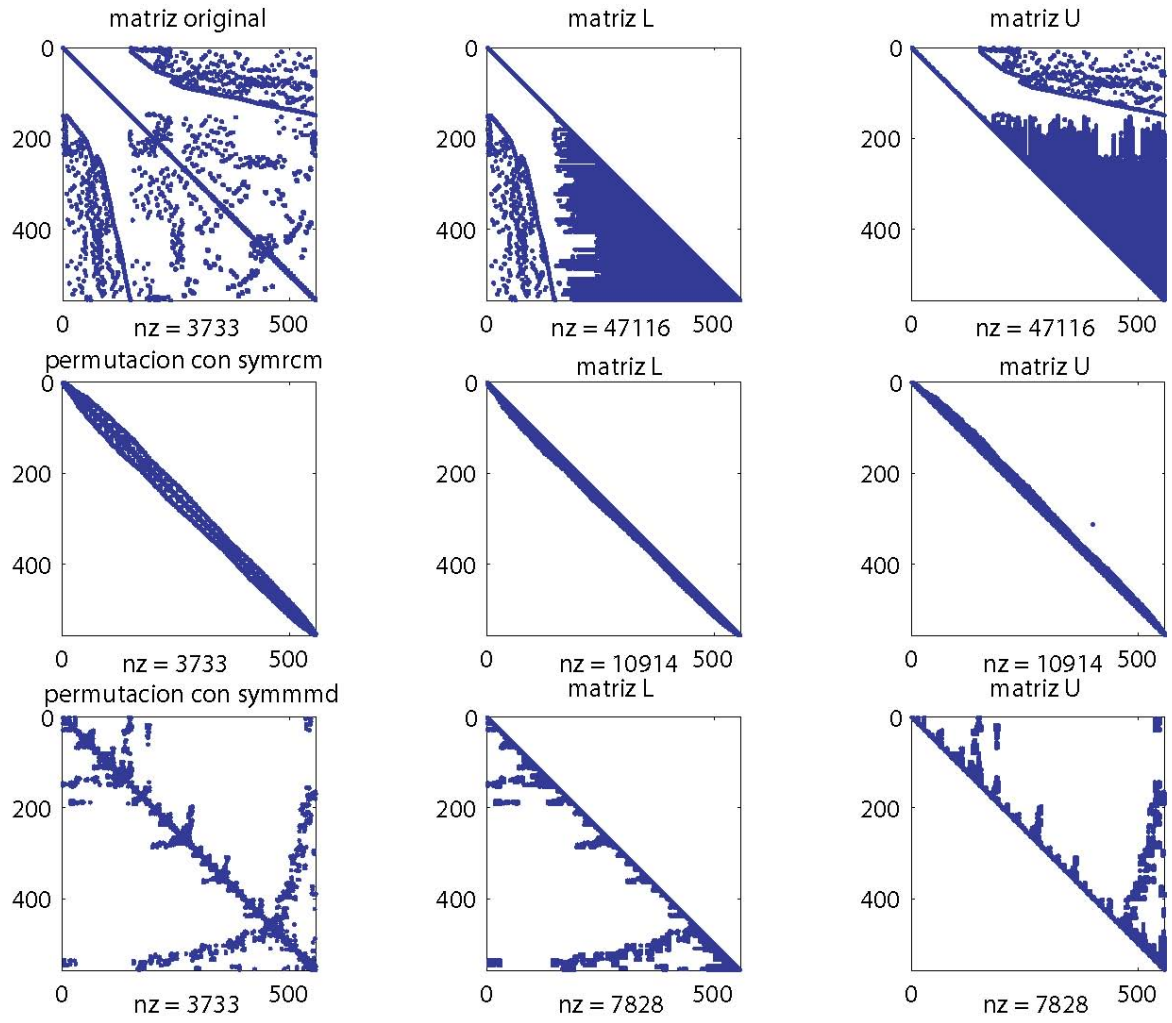


Figura 4.2: Resultado de reordenar las filas y columnas con `symrcm` y `symmmd`.

```
01  p=symmmd(a);          % permutacion. vale tb p=symrcm(a);
02  [l,u]=lu(a(p,p));     % l y u son ahora "mas sparse"
03  x=l\b(p); x=u\x;      % resolvemos los dos sistemas triangulares
04  x(p)=x;               % reordenamos las incognitas
```

En la línea `01`, `p` recoge una permutación de `1:n` (`n` es el número de filas y columnas de `a`) que reduce el efecto relleno de L y U.

Notas finales

- Si la matriz es además simétrica definida positiva podemos utilizar la descomposición de Cholesky,

$$A = LL^\top$$

donde L es triangular inferior con elementos sobre la diagonal estrictamente positivos. El sistema se reduce ahora a dos sistemas triangulares

$$L\mathbf{y} = \mathbf{b}, \qquad L^\top \mathbf{x} = \mathbf{y}.$$

La instrucción correspondiente en Matlab es `chol`, que devuelve R triangular superior de forma que

$$A = R^\top R$$

(es decir, en la notación de estos apuntes[5], $L = R^\top$). Así, el algoritmo anterior, con reordenación adecuada, queda ahora

```
01    p=symmmd(a);          % permutacion.... vale tb p=symrcm(a);
02    r=chol(a(p,p));       % l es ahora "mas sparse"
03    x=r'\b(p); x=r\x;     % resolvemos los dos sistemas triangulares
04    x(p)=x;               % recuperamos la solucion original
```

- La orden `x=a\b` realiza todo el proceso anterior de forma automática si `a` es sparse. Sin embargo en la forma que lo hemos expuesto podemos resolver reiteradamente sistemas de ecuaciones con la misma matriz de coeficientes al disponer de la descomposición LU ó $LL^\top$.

- Queda más allá de los contenidos de este curso explicar cómo funcionan los algoritmos implementados en `symrcm` y `symmmd`. Se puede señalar no obstante que ambos se basan en identificar la estructura de una matriz con la de un grafo no dirigido, donde dos nodos `i` y `j` están conectados si y sólo si `a(i,j)`$\neq$`0`. El proceso de eliminación gaussiana se ve como la *eliminación* sistemática de nodos y la creación de nuevos enlaces entre los nodos restantes. Concretamente, si hacemos ceros con la fila i en la fila j, aparecen nuevos enlaces entre los nodos que estaban unidos con el nodo i y los conectados con j. El problema se reescribe ahora en término de *retirar* los nodos en el orden adecuado de forma que se minimice el número de nuevos ejes creados (figura 4.3).

- Los gráficos de la figura 4.2 han sido creados a través del siguiente fichero script.

```
% Calculamos la matriz de elementos finitos para
% un problema sobre un dominio en forma de L

[p,e,t]=initmesh('lshapeg','Hmax',0.2);% malla inicial
[p,e,t]=refinemesh('lshapeg',p,e,t);% refinamiento
[a,b]=assempde('lshapeb',p,e,t,1,0,1);
```

[5] Recuerda que el operador trasposición $^\top$ en Matlab es '

```
% a es la matriz (sparse) y b el termino independiente

% Primera fila de dibujos
[l,u]=lu(a,0);% no pivotaje
figure(1)
subplot(331)
spy(a); title('matriz original')
subplot(332)
spy(l); title('matriz L')
subplot(333)
spy(u); title('matriz U')

% Segunda fila de dibujos
p=symrcm(a);% reordenamiento filas y columnas
[l,u]=lu(a(p,p),0);
figure(1)
subplot(334)
spy(a(p,p)); title('matriz permutada con symrcm')
subplot(335)
spy(l); title('matriz L')
subplot(336)
spy(u); title('matriz U')

% Tercera fila de dibujos
p=symmmd(a);% reordenamiento filas y columnas
[l,u]=lu(a(p,p),0);
figure(1)
subplot(337)
spy(a(p,p)); title('matriz permutada con symmmd')
subplot(338)
spy(l); title('matriz L')
subplot(339)
spy(u); title('matriz U')
```

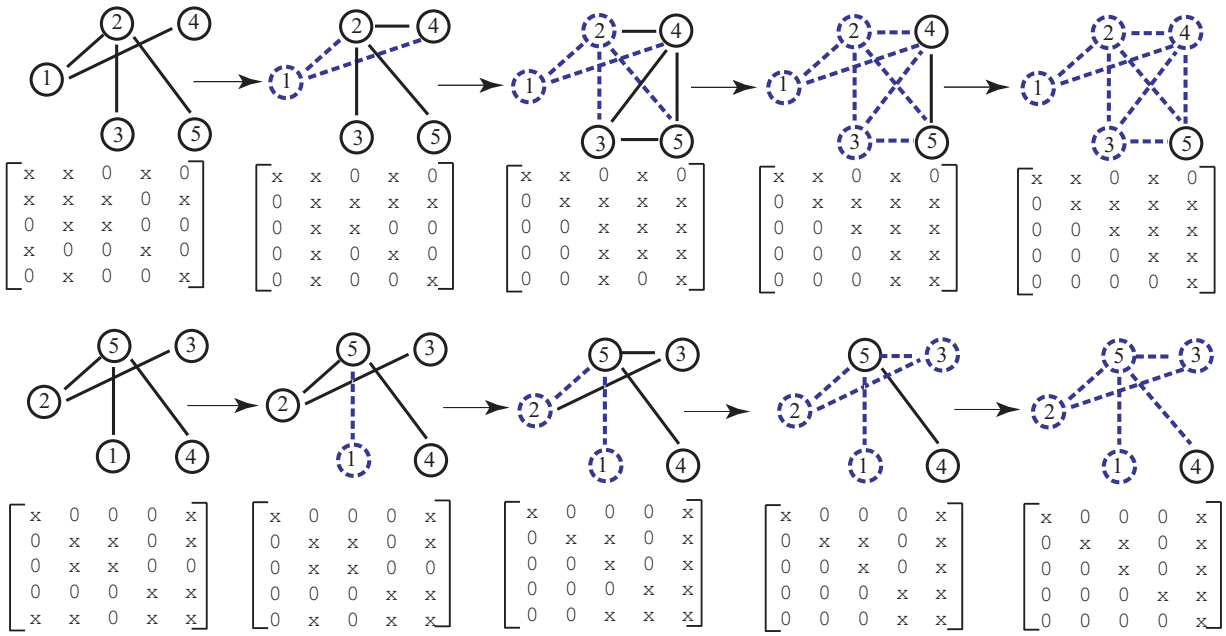


Figura 4.3: Eliminación gaussiana y su representación como un grafo. Efecto del reordenamiento.

Capítulo 5

Métodos iterativos

5.1. Introducción

Cuando un sistema de ecuaciones es de tamaño moderado[1], casi nadie duda en utilizar el método de Gauss en alguna de sus múltiples variantes (incluidas las descomposiciones matriciales).

Los métodos iterativos sin embargo se vuelven imprescindibles en problemas con matrices grandes y sparse donde el método de Gauss presenta las dificultades que ya hemos señalado.

Esencialmente, un método iterativo toma un sistema lineal

$$A\mathbf{x} = \mathbf{b}$$

y construye una sucesión de vectores de forma que

$$\mathbf{x}_m \to \mathbf{x}, \quad \text{cuando} \quad m \to \infty.$$

La forma de construir esta sucesión depende, obviamente, de la matriz A y del término independiente.

Existen multitud de esquemas diferentes pero todos comparten las siguientes desventajas:

- No siempre convergen, dependen de propiedades de la matriz (¡y no del término independiente!).
- Los resultados teóricos sobre convergencia son a menudo pobres. Esto es, en general existe convergencia en situaciones mucho más generales de lo que la teoría predice.
- La velocidad de convergencia puede ser extremadamente lenta.

Sin embargo, como aspectos muy positivos conviene citar

- Son métodos no destructivos. No modifican la matriz del sistema y, en general, precisan sólo multiplicar por la matriz del sistema, o por partes de ella[2].

[1] es difícil definir qué se entiende por tamaño moderado. Ciertamente, tamaños enormes en la decada de los años 70 son ahora perfectamente manejables en un pequeño PC.

[2] Visto los problemas del método de Gauss para matrices sparse, éste es sin lugar a dudas uno de los puntos fuertes de los métodos iterativos.

- Son más estables frente a los errores de redondeo[3].
- Se dispone en cada paso de una aproximación de la solución.

5.2. Detalles sobre su implementación

Como método iterativo que es, no podemos esperar calcular exactamente la solución, sino hallar una aproximación con una tolerancia prefijada. Por tanto debemos fijar un **criterio de parada** que termine el método cuando la solución se considere suficientemente buena[4].

Un posible criterio es medir la diferencia entre dos iteraciones consecutivas

$$\|\mathbf{x}_{m+1} - \mathbf{x}_m\|$$

en alguna norma que queda a elección del programador o del usuario. Si la diferencia es pequeña, se considera que estamos cerca de la solución y se finaliza el método.

De nuevo nos encontramos con el problema de definir "pequeño". Por ejemplo, una diferencia de 1.4 entre dos iteraciones puede ser grande si la solución tiene un tamaño $\|\mathbf{x}\| \approx 10^{-1}$ o pequeña si $\|\mathbf{x}\| \approx 10^{10}$.

Por ello se puede hacer un promedio entre el error absoluto y el relativo. Concretamente, si `x` y `y` son dos iteraciones consecutivas, se puede fijar el criterio

```
01      aux=norm(x-y);  % norma 2 entre x e y
02      if (aux<eps1+eps2*aux)
03          disp('Convergencia alcanzada...'
04          ..........
05      end
```

En la línea `02` tenemos en cuenta tanto el error absoluto (`eps1`) como el error relativo (`eps2`). Estos valores son parámetros que fijan el criterio de parada y pueden ser dados por el usuario.

Otros criterios se basan en calcular $\mathbf{r} = \mathbf{b} - A\mathbf{x}$, el residuo de $\mathbf{x}$, y calcular el tamaño. Es decir, medimos cuanto le falta a $\mathbf{x}$ para ser solución del sistema. En este caso tenemos de nuevo dos componentes, relativa, relacionada con el tamaño del término independiente, y otra absoluta:

```
01      aux=norm(b-a*x); % residuo
02      if (aux<eps1+eps2*norm(b))
03          disp('Convergencia alcanzada...'
04          ..........
05      end
```

[3]Gauss comentaba en una carta que los cálculos se podían realizar aún cuando "se estuviese medio dormido" o "pensando en cosas más importantes". No hay que olvidar que en su época todos los cálculos se hacían a mano.

[4]Esto no es grave. Los sistemas de ecuaciones lineales suelen venir de métodos que calculan *soluciones* aproximadas de problemas físicos e ingenieriles. No tiene pues sentido obcecarse en calcular la solución *exacta* de un problema *aproximado*

Este método puede ser preferible si se dispone ya del residuo y no hay que calcularlo *ex professo*.

Por otro lado, y desde un punto de vista computacional, debemos evitar que el método entre en un bucle infinito. Es decir, fijar un número máximo de iteraciones de forma que si se supera, se termine el método con un mensaje de error[5].

5.3. Métodos iterativos clásicos

5.3.1. Conocimientos previos

Al hablar de convergencia, debemos ocuparnos en formas el tamaño de un vector, es decir, lo que se conoce como norma de un vector. Así, dado un vector

$$\mathbf{x} := (x_1, x_2, \ldots, x_n)^\top \in \mathbb{R}^n$$

tenemos entre sus normas habituales,

$$\begin{aligned} \|\mathbf{x}\|_1 &:= |x_1| + |x_2| + \ldots + |x_n| \qquad \|\mathbf{x}\|_2 := \sqrt{x_1^2 + x_2^2 + \ldots + x_n^2} \\ \|\mathbf{x}\|_\infty &:= \max_{i=1,\ldots,n} |x_i|. \end{aligned}$$

Todas ellas son equivalentes, dado que

$$\|\mathbf{x}\|_2 \leq \|\mathbf{x}\|_1 \leq \sqrt{n}\|\mathbf{x}\|_2, \quad \|\mathbf{x}\|_\infty \leq \|\mathbf{x}\|_1 \leq n\|\mathbf{x}\|_\infty, \quad \|\mathbf{x}\|_\infty \leq \|\mathbf{x}\|_2 \leq \sqrt{n}\|\mathbf{x}\|_\infty$$

aunque esas constantes de equivalencia dependen de n. Todas ellas están implementadas en Matlab, como ya se vio en la sección 2.2. Existe un número infinito de normas, algunas de ellas están asociadas a matrices simétricas definidas positivas y tienen una gran importancia, como por ejemplo en los métodos de tipo Galerkin que trataremos más adelante.

Toda norma vectoriales definen a su vez una norma sobre las matrices

$$\|A\| = \sup_{\mathbf{x}\in\mathbb{R}^n} \frac{\|A\mathbf{x}\|}{\|\mathbf{x}\|}$$

denominada **norma inducida**[6]. Es fácil ver que

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

La norma $\|\cdot\|_2$ tiene una expresión más complicada que hace difícil su cálculo real. En concreto

$$\|A\|_2 = \sqrt{\rho(A^\top A)}$$

[5] Existe otro problema: que la solución crezca de forma que supere la cantidad máxima representable en coma flotante *overflow*. Matlab devolvería `inf` o `-inf`.

[6] La norma matricial mide cuánto cambia el tamaño de un vector al multiplicarlo por la matriz

donde $\rho(B)$, denominado **radio espectral**, denota el mayor de los valores absolutos de los valores propios de B[7]. En cualquier caso, la orden `norm` aplicada a matrices devuelve las correspondientes normas matriciales.

Una norma vectorial y su norma matricial inducida se relacionan con

$$\underbrace{\|A\mathbf{x}\|}_{\text{norma vectorial}} \leq \underbrace{\|A\|}_{\text{norma matricial}} \underbrace{\|\mathbf{x}\|}_{\text{norma vectorial}}$$

5.3.2. Definición y condiciones de convergencia

Los métodos iterativos lineales (o basados en iteraciones afines) comienzan considerando una partición de A t

$$A = M - N$$

con M invertible. Así, si $\mathbf{x}$ es solución de $A\mathbf{x} = \mathbf{b}$,

$$M\mathbf{x} = N\mathbf{x} + \mathbf{b}.$$

El método iterativo consiste en

- Tomar $\mathbf{x}_0$ una aproximación de la solución. Si no se conoce se puede tomar por ejemplo el vector nulo.
- Resolver en cada paso m

$$M\mathbf{x}_{m+1} = N\mathbf{x}_m + \mathbf{b} \tag{5.1}$$

Es fácil ver que si (5.1) converge a algún $\mathbf{x}$, entonces el vector es la solución del sistema. Aún es más, restando dos iteraciones sucesivas es inmediato comprobar que

$$\mathbf{x}_{m+k+1} - \mathbf{x} = (M^{-1}N)(\mathbf{x}_{m+k} - \mathbf{x}) = \ldots = (M^{-1}N)^k(\mathbf{x}_{m+1} - \mathbf{x}).$$

Denotando por $B = M^{-1}N$ se tiene el siguiente resultado

Teorema 1 *El método iterativo converge si y sólo si existe k tal que $\|B^k\| < 1$.*

Un detalle al que a veces se presta poca atención es que la convergencia del método ocurre sea cuál sea $\mathbf{x}_0$, es decir, independientemente de cómo se arranque el método y de cualquier término independiente. Obviamente, si se tiene una estimación buena de la solución el método convergerá en menos iteraciones, pero no es una condición para asegurar la convergencia[8].

No es difícil ver que

$$\rho(B) \leq \|B\|.$$

para cualquier norma inducida, y por tanto

$$(\rho(B^k))^{1/k} \leq (\|B^k\|)^{1/k} \qquad \forall k \in \mathbb{N}.$$

[7] Si A es simétrica se verifica fácilmente que $\|A\|_2 = \rho(A)$

[8] Esta propiedad se pierde en general cuando se resuelve ecuaciones y sistemas no lineales. En este caso, los métodos numéricos habitualmente convergen sólo si se arranca suficientemente cerca de la solución.

Es más, se puede probar que, de nuevo para toda norma inducida,

$$\lim_{k\to\infty} \|B^k\|^{1/k} = \rho(B)$$

Por tanto se concluye que una **condición equivalente de convergencia** es que todos los valores propios de $B = M^{-1}N$ tengan valor absoluto menor estrictamente que 1.

Teorema 2 *El método iterativo converge si y sólo si* $\|\rho(B)\| < 1$.

El resultado anterior puede utilizarse para probar que

$$\|\mathbf{x}_{m+1} - \mathbf{x}\| \leq C\rho(B)^m \|\mathbf{x}_1 - \mathbf{x}_0\|$$

para todo $m \geq m_0$ y C una constante independiente de m. A la cantidad $\rho(B)$ recibe en ocasiones el nombre de **velocidad asintótica de convergencia**. La estimación anterior justifica esta denominación dado que que mide la reducción media del error esperada en cada iteración.

Nota final

Nótese que si $M = A$ entonces $N = 0$, por tanto $B = 0$ y trivialmente

$$\rho(B) = 0.$$

Es decir, hay convergencia en una única iteración[9].

En general, uno asegura la convergencia cuando M *recoge* la información más importante de A, de forma que $N = M - A$ tenga un *tamaño* pequeño. Por otro lado, se debe tener en cuenta la definición del método (5.1) y que por tanto en cada iteración hay que resolver un sistema de ecuaciones. Así, interesa que M sea sencilla desde el punto de vista de la resolución del sistema lineal (por ejemplo, diagonal, triangular,...) y que la vez recoja la mayor información posible de A.

5.3.3. Métodos de Jacobi, Gauss-Seidel y Relajación

Sea el sistema $A\mathbf{x} = \mathbf{b}$ donde

$$A = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{bmatrix}$$

La familia de métodos que vamos a exponer comienzan trabajando sobre la siguiente partición de A

$$D = \begin{bmatrix} a_{11} & & & \\ & a_{22} & & \\ & & \ddots & \\ & & & a_{nn} \end{bmatrix}, \; L = \begin{bmatrix} 0 & 0 & \cdots & 0 \\ -a_{21} & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ -a_{n1} & -a_{n2} & \cdots & 0 \end{bmatrix}, \; U = \begin{bmatrix} 0 & -a_{12} & \cdots & -a_{1n} \\ 0 & 0 & \cdots & -a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 0 \end{bmatrix},$$

de forma que

$$A = D - L - U.$$

[9] La iteración es ahora resolver directamente el sistema de ecuaciones.

Método de Jacobi

Consiste en tomar en la definición del método iterativo $M = D$ y $N = L + U$, es decir

$$D\mathbf{x}_{m+1} = (L + U)\mathbf{x}_m + \mathbf{b}.$$

Para calcular $\mathbf{x}_{m+1}$ hay que resolver por tanto un sistema diagonal cuyo coste es despreciable. Visto componente a componente, tenemos que si

$$\mathbf{x}_{m+1} = (x_1^{(m+1)}, x_2^{(m+1)}, \ldots, x_n^{(m+1)})^\top \in \mathbb{R}^n$$

entonces

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

Una matriz es diagonal dominante por filas si

$$|a_{ii}| > \sum_{j\neq i} |a_{ij}|, \qquad \forall i$$

y por columnas si

$$|a_{jj}| > \sum_{i\neq j} |a_{ij}|, \qquad \forall j.$$

Teorema 3 *Si la matriz es diagonal dominante por filas o columnas el método de Jacobi converge.*

Nota La situación anterior es una constante en los métodos iterativos. En general se enuncian condiciones suficientes para asegurar la convergencia, pero pocos resultados dan condiciones necesarias, y aún éstos suelen dar hipótesis complicadas de comprobar.

Ejercicio 5 (matemático) *Probar el teorema anterior.*
(**Ayuda.** Construir la matriz $B = D^{-1}(L + U)$. Probar que $\|B\|_\infty < 1$ si es dominante por filas ó $\|B\|_1 < 1$ si es dominante por columnas. Aplicando el teorema de la sección 5.3.2, se tiene el resultado.)

Una descripción del método de Jacobi en pseudocódigo es la siguiente: fijamos

`x`= aproximación inicial, `mmax`= número máximo de iteraciones
`eps1`=tolerancia absoluta, `eps2` tolerancia relativa.

y tomamos como criterio de parada la norma 1 de la diferencia entre dos iteraciones sucesivas del método. La elección de dicha norma es libre, podemos tomar por ejemplo la norma 2 o la norma infinito.

Una primera versión de nuestro algoritmo es la siguiente

Jacobi

```
01  for m=1:mmax
02      error=0; y=x
03      for i=1:n
04          x_i = b_i
05          for j = 1 : i − 1
06              x_i = x_i − a_ij y_j
07          end
08          for j = i + 1 : n
09              x_i = x_i − a_ij y_j
10          end
11          x_i = x_i/a_ii .
12          error=error+|y_i − x_i|
13      end
14      if error<eps1+eps2*norm(x)
15          return
16      end
17  end
18  disp('numero máximo de iteraciones alcanzado')
19  return
```

Ejercicio 6 *Implementar el método de Jacobi en* Matlab *con la siguiente cabecera*

```
% JACOBI
%
% X    = JACOBI(A,B)          aplica el metodo de Jacobi para la
%                             resolucion del sistema AX=B
%
%[X,IT]= JACOBI(A,B)          devuelve en IT el numero de
%                             iteraciones calculadas
%
%[X,IT]= JACOBI(A,B,ITMAX)    ITMAX es el numero max. de iteraciones
%
%[X,IT]= JACOBI(A,B,ITMAX,... TOLABS,TOLREL son la tolerancia
%           EPS1,EPS2)    relativa y absoluta
%
%[X,IT]= JACOBI(A,B,ITMAX,...
%         EPS1,EPS2,X0)   Arranca el metodo con X0
%
```

Solución. He aquí una implementación de este método

```
01    % JACOBI
02    %
```

```
03    % X    = JACOBI(A,B)          aplica el metodo de Jacobi para la
04    %                             resolucion del sistema AX=B
05    %
06    %[X,IT]= JACOBI(A,B)          devuelve en IT el numero de
07    %                             iteraciones calculadas
08    %
09    %[X,IT]= JACOBI(A,B,ITMAX)    ITMAX es el numero max. de iteraciones
10    %
11    %[X,IT]= JACOBI(A,B,ITMAX,... TOLABS,TOLREL son la tolerancia
12    %           EPS1,EPS2)    relativa y absoluta
13    %
14    %[X,IT]= JACOBI(A,B,ITMAX,...
15    %         EPS1,EPS2,X0)   Arranca el metodo con X0
16    %
17    function [x, varargout]=jacobi(a,b,varargin)
18
19    % valores por defecto
20    n=length(a); mmax=100;
21    eps1=1e-4;  % tolerancia absoluta
22    eps2=1e-4;  % tolerancia relativa
23    x=zeros(n,1);
24    if nargin>2
25        mmax=varargin{1};
26     end
27    if nargin>3
28        eps1=varargin{2};
29    end
30    if nargin>4
31        eps2=varargin{3};
32    end
33    if nargin>5
34        x(:)=varargin{3}; %x es un vector columna
35    end
36
37    % Metodo de Jacobi
38
39    for m=1:mmax
40        error=0;
41        y=x;
42        for i=1:n
43            v=[1:i-1 i+1:n];
44            x(i)=(b(i)-a(i,v)*y(v))/a(i,i);
45        end
46        error=norm(x-y,1); % otras normas con norm(x-y,2),norm(x-y,inf)
47        if (error<eps1+eps2*norm(x))
48            break
```

```
49          end
50      end
51      if (m==mmax)
52          disp('numero maximo de iteraciones sobrepasado')
53      end
54
55      %salida
56
57      if (nargout>1)
58          varargout{1}=m;
59      end
60      return
```

Obsérvese que el código realmente dedicado al método de Jacobi se reduce a apenas diez líneas (`39-50`) con el resto de la subrutina dedicada al control del algoritmo y a la salida y entrada de datos.

Nótese además que la línea `43` permite implementar en una única línea (`44`) el producto por la parte no diagonal de a, que en el pseudocódigo ocupa 6 líneas (`05-10`)

El código anterior, sin embargo, es claramente optimizable. Los ejercicios siguientes ahondan en los aspectos mejorables.

Ejercicio 7 *Programar una nueva opción de salida que devuelva un vector* `error` *de longitud* $m-1$ *de forma que* `error(m)` *sea la diferencia entre* $\mathbf{x}_m$ *y* $\mathbf{x}_{m+1}$.

Ejercicio 8 *Otra forma alternativa de implementar el método es reemplazar las líneas* `42-45` *por el producto*

```
x=y+(b-a*y+d.*y)./d
```

o equivalentemente

```
x=x+(b-a*y)./d
```

donde `d` *es un vector columna que contiene la diagonal de la matriz. Observa que realizamos operaciones de más (multiplicamos por la diagonal para luego restar su contribución), pero el costo es despreciable y la operación es ahora puramente matricial. Implementa esta nueva forma y comprueba el resultado. ¿Obtienes mejoras en el redimiento del método*[10]*?*

Método de Gauss-Seidel

El método de Gauss-Seidel[11] consiste en tomar $M = D - L$ y $N = U$, es decir

$$(D - L)\mathbf{x}_{m+1} = U\mathbf{x}_m + \mathbf{b}. \tag{5.2}$$

[10]Deberás probar con matrices grandes. La orden `rand` te puede valer para ese fin. Una forma de asegurarte la convergencia es generar matrices diagonal dominantes. ¿Cómo se puede hacer esto?

[11]El nombre de este algorimo es muy curioso. Gauss no diseñó exactamente este método y Seidel, que lo estudió a finales del siglo XIX, desaconsejaba su uso. Gauss desarrolló un método muy similar cuando trataba de resolver un problema de geodesia que llevaba a un sistema lineal que no era compatible determinado.

Para calcular $\mathbf{x}_{m+1}$ hay que resolver un sistema, en este caso triangular. Repasando con cuidado las operaciones, observamos que

$$x_i^{(m+1)} = \frac{1}{a_{ii}}\Big[b_i - \sum_{j=1}^{i-1} a_{ij}x_j^{(m+1)} - \sum_{j=i+1}^{n} a_{ij}x_j^{(m)}\Big], \qquad i = 1, \ldots, n.$$

Escrito en forma algorítmica,

Gauss-Seidel

```
01  for m=1:mmax
02      error=0; y=x
03      for i=1:n
04          x_i = b_i
05          for j = 1 : i − 1
06              x_i = x_i − a_ij x_j
07          end
08          for j = i + 1 : n
09              x_i = x_i − a_ij y_j
10          end
11          x_i = x_i/a_ii .
12          error=error+|y_i − x_i|
13      end
14      if error<eps1+eps2*norm(x)
15          return
16      end
17  end
18  disp('numero máximo de iteraciones alcanzado')
19  return
```

Así pues, la única diferencia con el método de Jacobi es que Gauss-Seidel procede a utilizar la nueva componente calculada x_i tan pronto es posible (líneas `05-07`) mientras que Jacobi sólo las utiliza en la nueva iteración. Esta propiedad crea la sensación de que Gauss-Seidel es superior a Jacobi. No hay razones matemáticas que permitan apoyar esta impresión. De hecho existen matrices para las que Jacobi converge y Gauss-Seidel diverge, aunque hace falta construir un ejemplo *ex professo* (no es fácil encontrar uno) para comprobar esta afirmación. Desde un punto de vista práctico, si Jacobi converge es altamente probable que lo haga también Gauss-Seidel y generalmente, éste lo hará en menos iteraciones.

Teorema 4 *El método de Gauss-Seidel converge si*

- *la matriz es diagonal dominante por filas o columnas;*
- *la matriz es simétrica definida positiva.*

Hemos señalado que habitualmente los resultados de convergencia son muy incompletos. Sin embargo, existe una teoría completa para una familia particular de matrices: matrices triadiagonales y tridiagonales por bloques. Es decir matrices de la forma

$$\begin{bmatrix} A_1 & B_1 & & \\ C_2 & A_2 & B_2 & \\ & \ddots & \ddots & \ddots \\ & & C_n & A_n \end{bmatrix}$$

donde A_i, B_i y C_i son submatrices. En este caso, el método de Jacobi converge si y sólo si Gauss-Seidel converge y este caso el método de Gauss-Seidel converge a doble velocidad que el primero.

Ejercicio 9 *Programa el método de Gauss-Seidel modificando de forma apropiada la función del Ejercicio 6.*

Ejercicio 10 *De manera similar a lo que se propuso en el ejercicio 8, podemos implementar la parte central del método mediante*

```
x=l\(b-u*y)
```

o equivalentemente

```
x=x+l\(b-a*y)
```

donde

```
l=tril(a,0);      u=triu(a,1);
```

Implementa el método resultante
(**Ayuda:** recuerda los comandos `diag`, `triu` y `tril` de la sección 2.3)

Ejercicio 11 *El método de Gauss-Seidel tiene una curiosa asimetría. Si observamos, la primera componente de cada $\mathbf{x}_m$, comprobamos que ésta se calcula utilizando los valores de la anterior iteración $\mathbf{x}_{m-1}$, mientras que la última utiliza todas las componentes de la nueva iteración $\mathbf{x}_m$. Se puede plantear el método de Gauss-Seidel inverso que es el que resulta de intercambiar los papeles de L y U en (5.2). Ahora la situación es justamente la inversa. ¿Mejora la velocidad de convergencia del método?*

Ejercicio 12 *El método de Gauss-Seidel simetrizado trata de solventar la asimetría señalada en el ejercicio anterior. Consiste en* encajar *dos iteraciones, una con Gauss-Seidel y otra con el método de Gauss-Seidel inverso. Es decir, dado $\mathbf{x}_m$, se aplica el método de Gauss-Seidel para obtener $\mathbf{x}_{m+1/2}$ para luego obtener con el método de Gauss-Seidel inverso $\mathbf{x}_{m+1}$.*

Implementa este método. ¿Se reduce el número de iteraciones necesarias para alcanzar la convergencia? ¿Y el costo por iteración? ¿Te parece rentable esta aproximación?.

Método de Relajación

El método de relajación es una simple modificación sobre el método de Gauss-Seidel consistente en realizar un *promedio* sobre la aproximación que proporcionaría Gauss-Seidel y la iteración anterior.

Concretamente, la expresión es

$$x_i^{(m+1)} = (1-\omega)x_i^{(m)} + \frac{\omega}{a_{ii}}\Big[b_i - \sum_{j=1}^{i-1} a_{ij}x_j^{(m+1)} - \sum_{j=i+1}^{n} a_{ij}x_j^{(m)}\Big], \qquad i = 1, \ldots, n.$$

Si $\omega = 1$ recuperamos el método de Gauss-Seidel. El objetivo es escoger ω adecuado para acelerar la convergencia del método.

No es fácil encontrar el valor óptimo de ω, aunque necesariamente $\omega \in (0, 2)$ pues en caso contrario el método diverge.

Si la matriz es simétrica definida positiva o tridiagonal por bloques el método de relajación converge. También converge si es diagonal dominante (por filas o columnas), pero, salvo para matrices tridiagonales por bloques, poco se puede decir acerca de la elección óptima del parámetro ω[12].

Desde un punto de vista práctico, se suele determinar este parámetro mediante "ensayo–error". De forma algo sorprendente, es muy habitual que $\omega > 1$ en problemas prácticos de ahí el nombre que se le da a veces de sobrerelajación (*overrelaxed*).

Por último, aunque no sea inmediato, se puede comprobar que este método encaja en el marco anterior, sin más que tomar

$$M = \frac{1}{\omega}D - L, \qquad N = \frac{1-\omega}{\omega}D + U$$

Ejercicio 13 *Implementar el método de Relajación a partir del método de Gauss-Seidel. Incluir como nuevo argumento de entrada el parámetro ω. Un posible valor por defecto podría ser $\omega = 1$ con lo que tendríamos el método de Gauss-Seidel.*

Ejercicio 14 *De nuevo, el método de relajación se puede escribir en la forma*

```
x=y+{\tt m}(b-a*x)
```

donde m es una matriz adecuada. ¿Cuál es esa matriz?

Nota final

Los métodos clásicos son en la actualidad poco utilizados en la práctica y han sido reemplazados por métodos más potentes como los métodos de tipo Krylov de los que probablemente el método del Gradiente Conjugado es el representante más famoso. Sin embargo, son utilizados como *precondicionadores* de estos métodos, es decir, como un *preproceso* que acelera la convergencia de estos métodos más potentes.

[12] Y aún en este caso, la determinación exacta del parámetro óptimo exige resolver un problema bastante complicado.

5.4. Métodos de tipo gradiente

5.4.1. Introducción

Notación Recordemos que todos los vectores se consideran como vectores columna. Dada una matriz simétrica definida positiva, la cantidad que a cada par de vectores $\mathbf{x}$, $\mathbf{y}$ le asigna el número real

$$\mathbf{x}^\top A\mathbf{y}$$

es un producto escalar[13]. En particular, si A es la identidad,

$$\mathbf{x}^\top I_n\mathbf{y} = \mathbf{x}^\top\mathbf{y}$$

es simplemente el producto escalar euclídeo de $\mathbf{x}$ por $\mathbf{y}$. Denotaremos

$$\mathbf{x} \perp \mathbf{y} \quad \Longleftrightarrow \quad \mathbf{x}^\top\mathbf{y} = 0$$

y diremos en este caso que $\mathbf{x}$ e $\mathbf{y}$ son ortogonales. La misma notación se puede extender al producto definido por A, de forma que

$$\mathbf{x} \perp_A \mathbf{y} \quad \Longleftrightarrow \quad \mathbf{x}^\top A\mathbf{y} = 0$$

y en este caso $\mathbf{x}$ e $\mathbf{y}$ se dicen $A-$ortogonales.

La norma asociada al producto escalar anterior se conoce como **norma de energía** y su expresión viene dada, obviamente, por

$$\|\mathbf{x}\|_A := \sqrt{\mathbf{x}^\top A\mathbf{x}}.$$

Todas normas son equivalentes en $\mathbb{R}^n$, pero para esta norma se tiene además

$$\lambda_n\|\mathbf{x}\|_2 \leq \|\mathbf{x}\|_A \leq \lambda_1\|\mathbf{x}\|_2$$

donde $\lambda_1 \geq \lambda_2 \geq .... \geq \lambda_n > 0$ son los valores propios[14] de A. La cantidad

$$\kappa(A) = \frac{\lambda_1}{\lambda_n},$$

que será relevante en lo que sigue, es conocida como **condicionamiento** de la matriz[15].

Construimos la función

$$F(\mathbf{x}) = \frac{1}{2}\mathbf{x}^\top A\mathbf{x} - \mathbf{x}^\top\mathbf{b}$$

conocido como funcional de energía. Como A es definida positiva

$$F(\mathbf{x}) \to \infty, \quad \text{cuando} \quad \|\mathbf{x}\|_2 \to \infty,$$

[13] Esto es, cumple las propiedades que definen un producto escalar: 1) $(\alpha\mathbf{x}+\beta\mathbf{y})^\top A\mathbf{z} = \alpha\mathbf{x}^\top A\mathbf{y}+\beta\mathbf{z}^\top A\mathbf{z}$; 2) $\mathbf{x}^\top A\mathbf{y} = \mathbf{y}^\top A\mathbf{x}$; 3) $\mathbf{x}^\top A\mathbf{x} > 0$ si $\mathbf{x} \neq 0$.

[14] Todos reales por ser A simétrica y positivos por ser definida positiva

[15] La noción de condicionamiento se puede extender para matrices arbitrarias reemplazando los valores propios por los denominados valores singulares, o más en general, definiéndolo como el producto de la norma de A por la norma de su inversa. En general el condicionamiento de la matriz mide la sensibilidad del sistema lineal asociado a variaciones del término independiente.

y por tanto F tiene al menos un mínimo. Tras unos simples cálculos, se comprueba que

$$\nabla F = A\mathbf{x} - \mathbf{b}.$$

Por tanto existe un único extremo que necesariamente es mínimo. Hemos llegado por tanto a la siguiente conclusión:

> **Los problemas resolver $A\mathbf{x} = \mathbf{b}$ y encontrar el mínimo de F son equivalentes**

En esta observación se basan los métodos de descenso: en lugar de resolver el sistema de ecuaciones $A\mathbf{x} = \mathbf{b}$, nos preocupamos en buscar el mínimo de F. Obsérvese que es esencial que A sea definida positiva para que estos argumentos sean válidos.

5.4.2. Métodos de descenso. Aspectos generales

La idea de estos métodos es sencilla. Consiste en dada una aproximación $\mathbf{x}_m$ de $\mathbf{x}$, seguir los siguientes pasos

- Calcular una dirección de *descenso* $\mathbf{d}_m$.
- *Descender* una cantidad ξ_m, tomando como nueva aproximación

$$\mathbf{x}_{m+1} = \mathbf{x}_m + \xi_m \mathbf{d}_m.$$

Se procede así hasta que hay convergencia. Dos aspectos determinan el método: qué dirección se toma y cuánto se desciende.

En el método del Gradiente, como en el método de Gradiente Conjugado, se toma ξ_m de forma que

$$F(\mathbf{x}_{m+1}) = \min_{t\in\mathbb{R}} F(\mathbf{x}_m + t\mathbf{d}_m)$$

es decir, se trata de minimizar, una vez escogida la dirección de descenso $\mathbf{d}_m$, el funcional de energía. Definiendo

$$g(t) = F(\mathbf{x}_m + t\mathbf{d}_m)$$

podemos comprobar que

$$0 = g'(t) = t\mathbf{d}_m^\top A\mathbf{d}_m - \mathbf{d}_m^\top \underbrace{(\mathbf{b} - A\mathbf{x}_m)}_{\mathbf{r}_m}$$

y por tanto la longitud de descenso viene dada por

$$\xi_m := \frac{\mathbf{r}_m^\top \mathbf{d}_m}{\mathbf{d}_m^\top A\mathbf{d}_m} \tag{5.3}$$

Nótese que $\mathbf{r}_m = \mathbf{b} - A\mathbf{x}_m$ es el **residuo** de $\mathbf{x}_m$, que ya ha surgido en estas notas. Se puede probar fácilmente que

$$\mathbf{r}_{m+1} = \mathbf{b} - A\mathbf{x}_{m+1} = \mathbf{b} - A\mathbf{x}_m - \xi_m A\mathbf{d}_m = \mathbf{r}_m - \xi_m A\mathbf{d}_m,$$

y por tanto el residuo en pasos sucesivos satisface una relación similar a la que cumple $\mathbf{x}_m$. Sin embargo, que este tipo de recurrencia puede verse afectada por errores de redondeo, por lo que en ocasiones se calcula cada cierto número de iteraciones el residuo de acuerdo a su definición para evitar este problema

El algoritmo resultante es el siguiente:

Método de descenso

```
01  x_0,inicial, r_0 = b_0 − Ax_0
02  for m=0:mmax
03      Escoger d_m
04      ξ_m := (r_m^⊤ d_m) / (d_m^⊤ A d_m)
05      x_{m+1} = x_m + ξ_m d_m
06      r_{m+1} = r_m − ξ_m A d_m
07  end
```

5.4.3. Método del Gradiente

Dado que $-\nabla F(\mathbf{x}_m) = \mathbf{b} - A\mathbf{x}_m = \mathbf{r}_m$, se sigue la dirección de máximo descenso es la del residuo. Por tanto, ésta parece una buena elección para $\mathbf{d}_m$. El método así definido es el método del gradiente

Método del Gradiente

```
01  x_0  inicial; r_0 = b − Ax_0; r_0 = r^⊤ r
02  for m=0:mmax
03      p_m = Ar_m
04      ξ_m = r_m / (r_m^⊤ p_m)
05      x_{m+1} = x_m + ξ_m r_m
06      r_{m+1} = r_m − ξ_m p_m
07      r_{m+1} = r_{m+1}^⊤ r_{m+1}
07      if √r_{m+1} ≤ eps‖b‖
08          break
09      end
10  end
```

En el paso 06 estamos calculando el residuo de la solución, y en r_{m+1} guardamos su norma al cuadrado.

El criterio de parada se toma ahora a partir del residuo. Hemos escogido el basado en el tamaño relativo del residuo (eps$\|\mathbf{b}\|$). La norma que utilizamos es la euclídea, que es connatural al método.

Ejercicio 15 *Programa el método del Gradiente*

Solución. Una posible implementación del método es la que sigue

```
01    % GRADIENTE
02    %
03    % X      = GRADIENTE(A,B)         Aplica el met. del gradiente para
04    %                                 la resolucion del sistema AX=B
05    %
06    % X      = GRADIENTE(A,B,ITMAX)   ITMAX: numero max. de iteraciones
07    %
08    % X      = GRADIENTE(A,B,ITMAX... TOLREL tolerancia
09    %           TOL)                  relativa
10    %
11    % X      = GRADIENTE(A,B,ITMAX... X0 es el valor inicial
12    %           TOL, X0)
13    %
14    %[X,IT]  = GRADIENTE(A,B,ITMAX... Devuelve en IT el numero de
15    %           TOL,X0)                iteraciones calculadas
16
17    %[X,IT,R]= GRADIENTE(A,B,ITMAX)   R es un historial del metodo:
18    %           TOL,X0)                R(i) es el residuo en el paso i
19
20    function [x,varargout]= gradiente(a,b,varargin);
21
22    n=length(a); x=zeros(n,1); mmax=40;
23    tol=1e-6;
24
25    if nargin>2
26        mmax=varargin{1};
27    end
28    if nargin>3
29        tol=varargin{2};
30    end
31    if (nargin>4)
32        x=varargin{4};
33    end
34
35    r=b-a*x; res(1)=dot(r,r); aux=norm(b);
36    for m=1:mmax
37        p=a*r;
38        xi=res(m)/dot(r,p);
29        x=x+xi*r;
40        r=r-xi*p;
41        res(m+1)=dot(r,r);
42        if (sqrt(res(m))<tol*aux);
43           break
```

```
44          end
45      end
46
47      if (m==mmax)
48          disp('numero maximo de iteraciones sobrepasadas')
49      end
50      if nargout>1
51          varargout{1}=m;
52      end
53      if nargout>2
54          varargout{2}=sqrt(res(:));
55      end
56      return
```

Prueba el método con un sistema con matriz simétrica definida positiva (nota: para toda matriz A, $A^\top A$ es simétrica definida positiva.)

□

La gráfica (5.1) se ha construido utilizando las instrucciones

```
>> n=40;a=rand(n,n); a=a*a'; x=ones(n,1); b=a*x;
>> [x,it,r]=gradiente(a,b,100,1e-5);
>> semilogy(r)
```

La escala utilizada para medir la norma del residuo es logarítmica. Nótese su fuerte comportamiento oscilatorio.

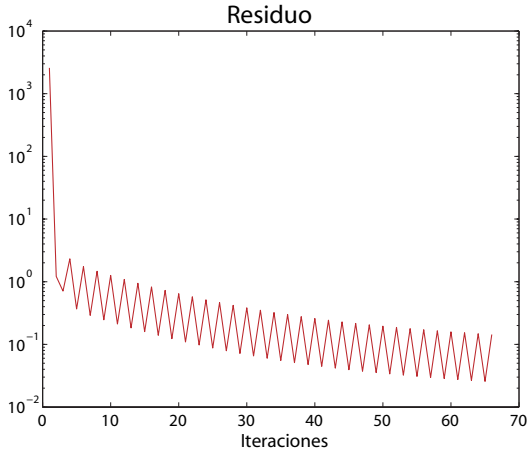


Figura 5.1: Residuo en las sucesivas iteraciones del método del Gradiente

Ejercicio 16 *En este ejercicio tratamos de nuevo aspectos de la implementación en* Matlab*. Concretamente, ¿qué pasa si el usuario desea especificar el vector de arranque (x_0 en la notación del método) pero desea dejar el número máximo de iteraciones y la tolerancia por defecto?. Como estándar en* Matlab*, se envía vacío (*`[ ]`*), de forma que los argumentos intermedios se ignoran. Por ejemplo,*

```
>> x=gradiente(A,b,[],1e-5)
```

especificaría la tolerancia pero **no** *el número máximo de iteraciones. Implementa las modificaciones necesarias en el programa anterior para que la función soporte este estándar.*
(**Ayuda:** La función `isempty` puede resultarte útil.)

Breves notas sobre el estudio del método del Gradiente

La clave del análisis es la relación

$$F(\mathbf{x}_{m+1}) - F(\mathbf{x}) = \frac{1}{2}(\mathbf{x}_{m+1} - \mathbf{x})^\top A(\mathbf{x}_{m+1} - \mathbf{x}) = \frac{1}{2}\| \underbrace{\mathbf{x}_{m+1} - \mathbf{x}}_{\mathbf{e}_{m+1}} \|_A^2$$

donde claramente $\mathbf{e}_{m+1}$ es el error entre la solución exacta y la numérica. Por tanto, como $F(\mathbf{x}_{m+1}) \leq F(\mathbf{x}_m)$,

$$\|\mathbf{e}_{m+1}\|_A \leq \|\mathbf{e}_m\|_A$$

luego en cada iteración hay una reducción del error en la norma de energía. Sin embargo, en ningún caso implica que el residuo se reduzca en cada iteración, como bien podemos comprobar en la figura 5.1. Aún es más, de la elección hecha de ξ_m se sigue que

$$\|\mathbf{e}_{m+1}\|_A \leq \min_{\alpha\in\mathbb{R}} \|\mathbf{x} - \mathbf{x}_m - \alpha\mathbf{r}_m\|_A = \min_{\alpha\in\mathbb{R}} \|\mathbf{e}_m - \alpha A\mathbf{e}_m\|_A \leq \left[\min_{\alpha\in\mathbb{R}} \|I - \alpha A\|_A\right] \|\mathbf{e}_m\|_A.$$

Proposición 5 *Sean λ_1 y λ_n el mayor y menor valor propio de A. Entonces*

$$\min_{\alpha\in\mathbb{R}} \|I - \alpha A\|_A = \min_{\alpha\in\mathbb{R}} \|I - \alpha A\|_2 = \frac{\lambda_1 - \lambda_n}{\lambda_1 + \lambda_n} < 1.$$

La convergencia se escribe de forma muy cómoda en términos del condicionamiento de la matriz

$$\kappa(A) = \frac{\lambda_1}{\lambda_n}$$

que permite dar la estimación

$$\|\mathbf{e}_{m+1}\|_A \leq \frac{\kappa(A) - 1}{\kappa(A) + 1} \|\mathbf{e}_m\|_A.$$

Es inmediato observar que

- Hay convergencia para toda A simétrica definida positiva.
- Si $\kappa(A) >> 1$, la convergencia puede ser muy lenta.

Ejercicio 17 (puramente matemático) *Sabiendo que dada A simétrica definida positiva existe B simétrica definida positiva tal que $BB^\top = B^2 = A$ probar la identidad*

$$\|I - \alpha A\|_A = \|I - \alpha A\|_2$$

utilizada en la proposición 5.
(**Ayuda:** Obsérvese que $\|\mathbf{x}\|_A = \|B\mathbf{x}\|_2$. Utilizar ahora que

$$\|I - \alpha A\|_A = \sup_{\mathbf{x}\in\mathbb{R}^n} \frac{\|(I - \alpha A)\mathbf{x}\|_A}{\|\mathbf{x}\|_A} = \sup_{\mathbf{x}\in\mathbb{R}^n} \frac{\|B(I - \alpha A)B^{-1}(B\mathbf{x})\|_2}{\|B\mathbf{x}\|_2}$$

y completar la demostración)

Ejercicio 18 (demostración de la Proposición 5) *Para toda matriz simétrica C se cumple que*

$$\|C\| = \max_j |\lambda_j|$$

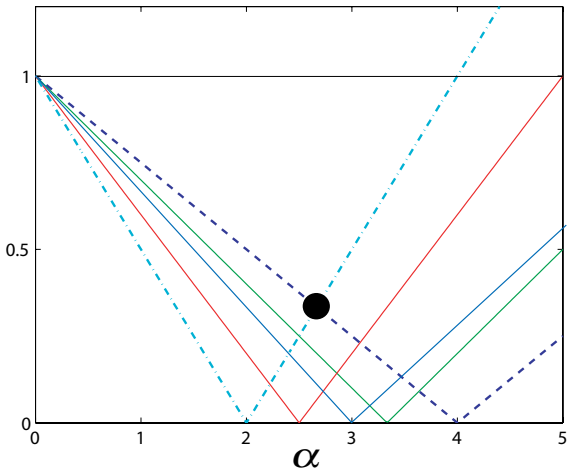
con λ_j el valor propio $j-$ésimo. Por tanto

$$\|I - \alpha A\|_A = \|I - \alpha A\|_2 = \max_j |1 - \alpha\lambda_j|.$$

Define $g_c(\alpha) = |1 - c\alpha|$ y traza la gráfica de estas funciones para varias valores de c. Utiliza esta propiedad para probar que $\alpha = 2/(\lambda_1 + \lambda_n)$ es el valor que hace mínimo $\|I - \alpha A\|_2$ y que para este valor,

$$\|I - \alpha A\|_2 = \frac{\lambda_1 - \lambda_n}{\lambda_1 + \lambda_n}.$$

(**Ayuda:** Toma varios valores de c en la definición de la función $g_c(\alpha)$ y dibuja la gráficas resultantes. Obtendrás algo similar a esto



¿Cuáles son las gráficas de los valores extremos? ¿Cuánto vale el corte?)

5.4.4. El Gradiente Conjugado

El método del Gradiente Conjugado trata de resolver alguna de las dificultades observadas con el método del Gradiente, como por ejemplo el comportamiento oscilatorio del residuo.

Observemos que en un método de descenso, y por la elección de ξ_m hecha en (5.4.2),

$$\mathbf{r}_{m+1}^\top \mathbf{d}_m = \mathbf{r}_m^\top \mathbf{d}_m - \xi_m \mathbf{d}_m^\top A \mathbf{d}_m = 0,$$

por lo que $\mathbf{r}_{m+1} \perp \mathbf{d}_m$. Esto es el, el residuo del paso $m+1$ es ortogonal **a la dirección de descenso** anterior.

Sin embargo, en general ya **no es cierto** que

$$\mathbf{r}_{m+1} \perp \mathbf{d}_{m-1} \tag{5.4}$$

y por tanto se pierde esta propiedad de ortogonalidad.

Una forma de evitar, o de mitigar, el aspecto oscilante del método de Gradiente, es exigir que la dirección de descenso $\mathbf{d}_m$ satisfaga (5.4). Dado que

$$\mathbf{r}_{m+1} = \mathbf{r}_m - \xi_m A\mathbf{d}_m,$$

concluimos que

$$\mathbf{d}_{m-1}^\top \mathbf{r}_{m+1} = 0 \quad \Longleftrightarrow \quad \mathbf{d}_{m-1} \perp A\mathbf{d}_m \quad \Longleftrightarrow \quad \mathbf{d}_{m-1} \perp_A \mathbf{d}_m.$$

Optamos por tomar, por tanto, como dirección de descenso una perturbación de la *dirección natural* (el residuo) en la dirección del descenso anterior para que se satisfaga la propiedad de ortogonalidad anterior. Así

$$\mathbf{d}_m = \mathbf{r}_m + \tau_m \mathbf{d}_{m-1}.$$

De esta forma

$$\mathbf{d}_m \perp_A \mathbf{d}_{m-1} \quad \Longleftrightarrow \quad \mathbf{r}_m^\top A\mathbf{d}_{m-1} + \tau_m \mathbf{d}_{m-1}^\top A\mathbf{d}_{m-1} = 0 \quad \Longleftrightarrow \quad \tau_m = -\frac{\mathbf{r}_m^\top A\mathbf{d}_{m-1}}{\mathbf{d}_{m-1}^\top A\mathbf{d}_{m-1}}$$

Con todo esto, la primera versión del método del Gradiente Conjugado es la que sigue

Gradiente Conjugado - Primera versión

```
01  x_0  Inicial;  d_0 = b - A x_0;  r_0 = r^T r
02  for m=0:mmax
03      p_m = A d_m
04      xi_m = (r_m^T d_m) / (d_m^T p_m)
05      x_{m+1} = x_m + xi_m d_m
06      r_{m+1} = r_m - xi_m p_m
07      r_{m+1} = r_{m+1}^T r_{m+1}
08      if sqrt(r_{m+1}) <= eps||b||
09          break
10      end
11      tau_{m+1} = -(r_{m+1}^T p_m) / (d_m^T p_m)
12      d_{m+1} = r_{m+1} + tau_{m+1} d_m
13  end
14  disp('Numero maximo de iteraciones alcanzadas')
```

Como puede verse, las modificaciones sobre el método del Gradiente son mínimas. En cada paso, es necesario calcular un producto matriz–vector (línea 03) y dos productos escalares (04 y 07) además de guardar la norma del residuo de la iteración anterior.

Podemos guardar los residuos de todas las iteraciones para disponer de información del historial de la convergencia del método.

Listaremos a continuación algunas propiedades entre los residuos y las direcciones de descenso que permite dar una expresión distinta del método, y deducir alguna de las propiedades de convergencia.

Lema 6 *Para todo* $m \geq 0$

$$\mathbf{r}_m^\top \mathbf{d}_m = \mathbf{r}_m^\top \mathbf{r}_m.$$

Por tanto

$$\xi_m = \frac{\mathbf{r}_m^\top \mathbf{r}_m}{\mathbf{d}_m A \mathbf{d}_m}.$$

Demostración. Como $\mathbf{r}_m \perp \mathbf{d}_{m-1}$,

$$\mathbf{r}_m^\top \mathbf{d}_m = \mathbf{r}_m^\top \mathbf{r}_m + \tau_m \underbrace{\mathbf{r_m}^\top \mathbf{d}_{m-1}}_{} = 0. \tag{5.5}$$

El segundo resultado es inmediato de la definición de ξ_m. □

Lema 7 *Para todo* $m \geq 0$ *se satisfacen las siguientes relaciones*

i) $\mathbf{d}_{m+1} \perp_A \mathbf{d}_m$.

ii) $\mathbf{r}_{m+1} \perp \mathbf{d}_m, \qquad \mathbf{r}_{m+1} \perp \mathbf{d}_{m-1}$.

iii) $\mathbf{r}_{m+1} \perp \mathbf{r}_m$.

Demostración Los puntos i) y ii) ya se han probado. Para probar iii) basta comprobar que

$$\begin{aligned} \mathbf{r}_{m+1}^\top \mathbf{r}_m &= \mathbf{r}_m^\top \mathbf{r}_m - \xi_m \mathbf{d}_m^\top A \mathbf{r}_m = \mathbf{r}_m^\top \mathbf{r}_m - \xi_m \mathbf{d}_m^\top A \mathbf{d}_m + \underbrace{\tau_m \mathbf{d}_m^\top A \mathbf{d}_{m-1}}_{=0} \\ &= \mathbf{r}_m^\top \mathbf{r}_m - \xi_m^\top \mathbf{d}_m^\top A \mathbf{d}_m. \end{aligned}$$

El resultado es consecuencia del Lema 6 □

Lema 8 *Para todo* $m \geq 0$

$$\tau_{m+1} := \frac{\mathbf{r}_{m+1}^\top \mathbf{r}_{m+1}}{\mathbf{r}_m^\top \mathbf{r}_m}.$$

Demostración. Del Lema anterior sabemos que $\mathbf{r}_{m+1} \perp \mathbf{r}_m$. Así,

$$\tau_{m+1} = -\frac{\mathbf{r}_{m+1}^\top A \mathbf{d}_m}{\mathbf{d}_m A \mathbf{d}_m} = \frac{1}{\xi_m} \frac{\mathbf{r}_{m+1}^\top (\mathbf{r}_{m+1} - \mathbf{r}_m)}{\mathbf{d}_m A \mathbf{d}_m} = \frac{1}{\xi_m} \frac{\mathbf{r}_{m+1}^\top \mathbf{r}_{m+1}}{\mathbf{d}_m^\top A \mathbf{d}_m} = \frac{\mathbf{r}_{m+1}^\top \mathbf{r}_{m+1}}{\mathbf{r}_m^\top \mathbf{d}_m} = \frac{\mathbf{r}_{m+1}^\top \mathbf{r}_{m+1}}{\mathbf{r}_m^\top \mathbf{r}_m}$$

donde en el último paso hemos utilizado el Lema 6. □

Los lemas 6 y 9 dan expresiones más simples de los parámetros ξ_m y τ_m. De hecho más económicas, dado que $\mathbf{r}_m^\top \mathbf{r}_m$ es conocido del paso anterior y por tanto no es preciso calcularlo de nuevo.

Ejercicio 19 *Modificar el algoritmo del Gradiente Conjugado con las nuevas expresiones de* ξ_m *y* τ_m. *Observa que ahora sólo es necesario un producto matriz-vector y un producto escalar.*

El resultado más sorprendente y que descubre parte de las buenas propiedades del Gradiente Conjugado es que estas propiedades de ortogonalidad se extienden a todo el conjunto de residuos y direcciones generados en los pasos anteriores.

Lema 9 *Para todo* $m \leq n$

i) $\mathbf{d}_m \perp_A \mathbf{d}_\ell, \quad \forall \ell \leq m-1.$

ii) $\mathbf{r}_{m+1} \perp \mathbf{d}_\ell, \quad \forall \ell \leq m.$

iii) $\mathbf{r}_{m+1} \perp \mathbf{r}_\ell, \quad \ell \leq m$

La tercera propiedad implica en particular la convergencia del Gradiente Conjugado en aritmética exacta en a lo sumo n pasos dado que no puede haber $n+1$ vectores ortogonales en $\mathbb{R}^n$. Sin embargo, en aplicaciones prácticas puede requerir de más de n iteraciones por los errores introducidos por la precisión de la máquina.

Ejercicio 20 *Modificar el programa* `gradiente` *para implementar en una función de nombre* `gradconjugado` *el método del Gradiente Conjugado.*

Hemos ejecutado como antes

```
>> n=40;a=rand(n,n); a=a*a'; x=ones(n,1); b=a*x;
>> [x,it,r]=gradconjugado(a,b,100,1e-5);
>> semilogy(r)
```

y hemos desplegado en la figura 5.2 las normas de los residuos en cada paso. Como se

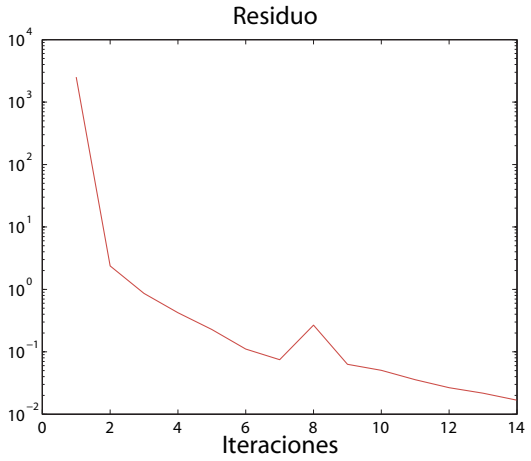


Figura 5.2: Residuo para el método del Gradiente Conjugado

puede comprobar la convergencia es más rápida y hemos reducido notablemente el molesto comportamiento oscilatorio del residuo.

Análisis del método

En esta sección probamos el Lema 9. En la demostración juega un papel muy importante el subespacio

$$\mathcal{K}_m(A, \mathbf{r}_0) := \mathbb{R}\langle \mathbf{r}_0, A\mathbf{r}_0, \ldots, A^m\mathbf{r}_0\rangle$$

esto es, el subespacio formado por todas las combinaciones lineales

$$\lambda_0\mathbf{r}_0 + \lambda_1 A\mathbf{r}_1 + \ldots + \lambda_m A^m\mathbf{r}_0, \qquad \lambda_i \in \mathbb{R}.$$

Antes de proseguir unos simples comentarios:

1. El subespacio tiene a lo sumo dimensión $m+1$.
2. Si $\mathbf{q} \in \mathcal{K}_m(A, \mathbf{r}_0)$, entonces $A\mathbf{q} \in \mathcal{K}_{m+1}(A, \mathbf{r}_0)$. Es decir, $A\mathcal{K}_m(A, \mathbf{r}_0) \subset \mathcal{K}_{m+1}(A, \mathbf{r}_0)$.
3. Es fácil comprobar que $\mathbf{r}_m, \mathbf{d}_m \in \mathcal{K}_m(A, \mathbf{r}_0)$

Estos subespacios reciben el nombre de **subespacios de Krylov** y son clave en multitud de métodos numéricos para la resolución de sistemas lineales.

La demostración se llevará a cabo por inducción sobre m. Para $m = 0$ es un simple ejercicio de comprobación (nótese que en este caso $\mathbf{r}_0 = \mathbf{d}_0$ y por tanto ii) y iii) coinciden). Supongamos pues que el resultado está probado para m. Concretamente, supongamos que

i) $\mathbf{d}_m \perp_A \mathbf{d}_\ell$, $\forall \ell \leq m-1$ y $\{\mathbf{d}_0, \mathbf{d}_1, \ldots, \mathbf{d}_m\}$ es una base de $\mathcal{K}_m(A, \mathbf{r}_0)$.

ii) $\mathbf{r}_{m+1} \perp \mathbf{d}_\ell$, $\forall \ell \leq m$.

iii) $\mathbf{r}_{m+1} \perp \mathbf{r}_\ell$, $\ell \leq m$ y $\{\mathbf{r}_0, \mathbf{r}_1, \ldots, \mathbf{r}_{m+1}\}$ es una base ortogonal de $\mathcal{K}_{m+1}(A, \mathbf{r}_0)$.

Veamos como se sigue cumpliendo los puntos i)–iii para $m+1$.

i) $\mathbf{d}_{m+1} \perp_A \mathbf{d}_\ell$, $\forall \ell \leq m$:

El resultado está probado ya para $\ell = m$. Para $\ell \leq m-1$, utilizamos que

$$\mathbf{d}_{m+1} = \mathbf{r}_{m+1} + \tau_{m+1}\mathbf{d}_m.$$

Por tanto el resultado se tiene una vez que hayamos probado que

$$\mathbf{r}_{m+1} \perp_A \mathbf{r}_\ell, \qquad \mathbf{d}_m \perp_A \mathbf{d}_\ell.$$

El segundo es ya conocido (hipótesis de inducción). Por otro lado,

$$\mathbf{r}_{m+1} \perp_A \mathbf{r}_\ell \quad \Longleftrightarrow \quad \mathbf{r}_{m+1} \perp A\mathbf{r}_\ell$$

Como $A\mathbf{r}_\ell \in \mathcal{K}_{\ell+1}(A, \mathbf{r}_0)$ y una base de este subespacio es $\{\mathbf{r}_0, \mathbf{r}_1, \ldots, \mathbf{r}_{\ell+1}\} \subset$, concluimos que

$$A\mathbf{r}_\ell = \alpha_0\mathbf{r}_0 + \alpha_1\mathbf{r}_1 + \ldots + \alpha_{\ell+1}\mathbf{r}_{\ell+1}$$

con $\alpha_j \in \mathbb{R}$ adecuados. El resultado se sigue ahora de iii) puesto que $\mathbf{r}_{m+1} \perp \mathbf{r}_\ell$, para todo $\ell \leq m$.
Por último si $\mathbf{d}_{m+1} \neq \mathbf{0}$ entonces $\{\mathbf{d}_0, \ldots, \mathbf{d}_{m+1}\}$ es una base de $\mathcal{K}_{m+1}(A, \mathbf{r}_0)$ por ser $A-$ortogonales y por tanto linealmente independientes. □

ii) $\mathbf{r}_{m+2} \perp \mathbf{d}_\ell, \quad \forall \ell \leq m+1$

Para $\ell = m, m+1$ es cierto por construcción (véase Lema 7). Para el resto de valores $\ell \leq m$, utilizamos

$$\mathbf{r}_{m+2} = \mathbf{r}_{m+1} - \xi_{m+1} A\mathbf{d}_{m+1}.$$

Por inducción, $\mathbf{r}_{m+1} \perp \mathbf{d}_\ell$, mientras que para el otro término, observamos

$$(A\mathbf{d}_{m+1}) \perp \mathbf{r}_\ell \quad \Longleftrightarrow \quad \mathbf{d}_{m+1} \perp_A \mathbf{r}_\ell.$$

De nuevo, como $\mathbf{r}_\ell \subset \mathcal{K}_\ell(A, \mathbf{r}_0)$ y por el punto i)

$$\mathbf{r}_\ell = \beta_0 \mathbf{d}_0 + \beta_1 \mathbf{d}_1 + \ldots + \beta_\ell \mathbf{d}_\ell \tag{5.6}$$

por lo que el resultado se sigue de la propiedad i) ya que $\mathbf{d}_{m+1} \perp_A \mathbf{d}_\ell$ para $\ell = 0, \ldots, m$. □

iii) $\mathbf{r}_{m+2} \perp \mathbf{r}_\ell, \quad \forall \ell \leq m+1.$

Tomemos $\ell \leq m$. Entonces

$$\begin{aligned} \mathbf{r}_{m+2} \perp \mathbf{r}_\ell \quad &\Longleftrightarrow \quad (\mathbf{r}_{m+1} - \xi_{m+1} A\mathbf{d}_{m+1}) \perp \mathbf{r}_\ell \iff (A\mathbf{d}_{m+1}) \perp \mathbf{r}_\ell \\ &\Longleftrightarrow \quad (\mathbf{d}_{m+1}) \perp_A \mathbf{r}_\ell. \end{aligned}$$

donde hemos utilizado la hipótesis de inducción en el segundo paso. La última propiedad se prueba recurriendo de nuevo a los espacios de Krylov, concretamente $\mathbf{r}_\ell \in \mathcal{K}_\ell(A, \mathbf{r}_0$, utilizando (5.6) y la propiedad de ortogonalidad de i). □

Un examen más cuidadoso de la demostración anterior comprueba que ésta falla en el caso de que en un paso $\mathbf{r}_{m+1}$ ó $\mathbf{d}_{m+1}$ sea nulos. En cualquiera de estos casos

$$\mathcal{K}_m(A, \mathbf{r}_0) = \mathcal{K}_{m+1}(A, \mathbf{r}_0).$$

Ahora bien, un análisis algo más detallado nos descubre que esto sucede si y sólo si ha habido convergencia en el paso m (si $\mathbf{d}_{m+1} = \mathbf{0}$) o en el paso $m+1$ (si $\mathbf{r}_{m+1} = \mathbf{0}$)[16].

Notas finales

El Gradiente Conjugado fue propuesto por Hestenes y Stiefel en 1952. Lo más curioso es que el método fue desarrollado de forma independiente por estos dos autores. Hestenes se encontró con Stiefel en una conferencia en UCLA (University of California, Los Angeles) y le comentó a grandes trazos el método en el que estaba trabajando. Stiefel estaba impresionado sobre las buenas propiedades que mostraba este método hasta que

[16] Esto es, si la dirección de descenso es la nula simplemente es porque ya hemos alcanzado la solución, y viceversa, si el residuo es nulo, la dirección de descenso para la siguiente iteración es la nula

cayó en la cuenta de que se trataba del mismo método sobre el que independiente él estaba trabajando[17].

Hemos visto que el método del Gradiente Conjugado es un método directo, en tanto en cuanto da la solución en un número finito de pasos, concretamente n, el número de filas de la matriz. Sin embargo, se programa como un método iterativo, de forma que se busca la convergencia en muchas menos iteraciones. En cada iteración se tiene la estimación

$$\|\mathbf{e}_{m+1}\|_A \leq 2\left(\frac{\sqrt{\kappa(A)}-1}{\sqrt{\kappa(A)}+1}\right)^m \|\mathbf{e}_0\|_A.$$

donde de nuevo $\mathbf{e}_m := \mathbf{x}_m - \mathbf{x}$ es el error en el paso $m-$ésimo y $\mathbf{e}_0$ el error inicial.

El resultado anterior es algo incompleto pero sirve para hacer patente la sensibilidad del método ante el condicionamiento de una matriz que, aunque menor que en el caso del método del Gradiente (por la raíz cuadrada), sigue siendo importante. Si la matriz tiene un condicionamiento moderado, el método del Gradiente Conjugado dará una solución aceptable en unas pocas iteraciones. Sin embargo, esto no suele ser un caso habitual. Las matrices que surgen en la resolución de ecuaciones en derivadas parciales, por ejemplo elementos o volúmenes finitos, suelen ser, demás de matrices sparse, mal condicionadas por lo que el método del Gradiente Conjugado requiere de bastantes iteraciones[18].

Estas razones hicieron que el Gradiente Conjugado se tomara como un método más, sin concederle especial relevancia y confundiéndose con la multitud de métodos para la resolución de sistemas de ecuaciones lineales que iban surgiendo a su alrededor. Esta situación se prolongó durante casi 20 años hasta que las ideas del **precondicionamiento** cambiaron radicalmente esta situación y encumbraron al Gradiente Conjugado a su posición actual[19].

El precondicionamiento consiste, a grandes rasgos, en cambiar el sistema por uno equivalente[20]

$$A\mathbf{x} = \mathbf{b}, \qquad \rightsquigarrow \qquad \underbrace{L^{-\top}AL^{-1}}_{B}\mathbf{y} = \underbrace{L^{-\top}\mathbf{b}}_{\mathbf{c}}, \quad \mathbf{x} = L^{-1}\mathbf{y}$$

y aplicar el método al sistema $B\mathbf{y} = \mathbf{c}$. A la matriz L se le conoce como **precondicionador** del sistema[21]

En general, la matriz B no se construye, porque el producto por el precondicionador hace que se pierdan algunas de las buenas propiedades de la matriz original[22]. Es más, uno

[17] Es posible que la posibilidad de expresar algunos de los parámetros (ej. τ) de forma distinta contribuyera a esta confusión. No es de todas formas un caso aislado en las Ciencias en general y en las Matemáticas en particular. En ocasiones parece como si las ideas estuviesen flotando en el ambiente a la espera de que alguien diera el paso final de plasmarlas.

[18] $\mathcal{O}(n^{1/3})$ en problemas en $3D$, $\mathcal{O}(n^{1/2})$ en 2D

[19] La *American Mathematic Society* lo sitúo entre los diez algoritmos más relevantes del siglo XX, junto con, por ejemplo, la Transformada Rápida de Fourier (FFT).

[20] Utilizamos la notación $L^{-\top} = (L^{-1})^\top$, es decir invertir y trasponer (o equivalentemente, trasponer e invertir)

[21] La idea del precondicionamiento se extiende a casi todos los métodos iterativos. En el caso general suele ser más sencillo. Aquí, el producto a izquierda por $L^{-\top}$ y a derecha por $L^\top$ se hace para que la matriz B sea de nuevo simétrica definida positiva.

[22] La inversa de una matriz sparse, no es sparse (en general). Se puede comprender fácilmente las

conoce L y no su inversa, y el hecho de calcular L^{-1} suele ser un problema tan complejo como resolver el sistema original.

Ahora bien, y he aquí la ventaja del método, **sólo necesitamos calcular productos por** B. Obviamente, cualquier producto por la inversa de L, $\mathbf{p} = L^{-1}\mathbf{r}$, se hace resolviendo el correspondiente sistema de ecuaciones

$$L\mathbf{p} = \mathbf{r}.$$

La buena noticia es que es posible construir matrices L para las que la resolución del sistema anterior es fácil de hacer (por ejemplo, L puede ser triangular y sparse y con un número de entradas comparable a la de A) y tales que la matriz producto $B = L^{-\top}AL^{-1}$ tenga un condicionamiento mucho menor, esto es, $\kappa(B) << \kappa(A)$.

El método del Gradiente Conjugado rara vez se programa sin precondicionar. De hecho, **el comando de** Matlab con el Gradiente Conjugado implementado es `pcg` de *preconditioned conjugate gradient*.

Un buen precondicionador es la descomposición de Cholesky incompleta[23]. En Matlab se encuentra implementada en el comando `cholinc`. Va más allá de los contenidos de este curso describir detalladamente en qué consiste y por qué funciona, pero no está de más saber de su existencia.

En última medida, el "éxito" del Gradiente Conjugado originó el nacimiento de una familia entera de métodos, los conocidos como **métodos de Krylov**, que trataban de extender las buenas propiedades del método a matrices más generales. Por ejemplo, BiCG, BiCGSTAB, CGS QMR, GMRES, MINRES, ... algunos de estos métodos requieren que la matriz sea simétrica, otros son válidos para matrices arbitrarias.

Ejercicio 21 *En este ejercicio ilustra el efecto del precondicionamiento en la convergencia del Gradiente Conjugado y con ello trata de convencer de la necesidad de utilizarlo en la resolución de (muy) grandes sistemas de ecuaciones. Las siguientes instrucciones*[24]

```
>> [p,e,t]=initmesh('lshapeg','Hmax',0.05);
>> [a,b]=assempde('lshapeb',p,e,t,1,0,1);
```

devuelven en `a` *una matriz sparse simétrica definida proveniente de resolver la ecuación de Poisson (una ecuación en derivadas parciales) por el método de elementos finitos en un dominio en forma de 'L'. La matriz tiene aproximadamente* 2100 *filas con* 14500 *elementos no nulos*[25].

Con el comando `spy` *se puede ver la forma de* `a`*, su tamaño y el número de entradas no nulas. Aplica el comando* `pcg` *que contiene implementado el Gradiente Conjugado. Por ejemplo, con*

```
>>[x,flag, relres,iter,resvec] = pcg(a,b,1e-7,100);
```

consecuencias que tiene invertir una matriz de, digamos, 20.000 filas con 150.000 elementos no nulos. Su inversa tiene ahora del orden de 4×10^7 entradas no nulas con lo que las necesidades de memoria se han multiplicado aproximadamente por 265.

[23] Se trata de, *a grosso modo*, aplicar el método de Cholesky pero restringiendo la creación de nuevas entradas. El resultado es una matriz L tal que $L^\top L \approx A$ en algún sentido, y que por tanto, $L^{-\top}AL^{-1} \approx I_n$ y su condicionamiento es muy reducido.

[24] Su ejecución puede llevar algo de tiempo...

[25] Por cierto, el logo de Matlab es la solución de un problema de este tipo

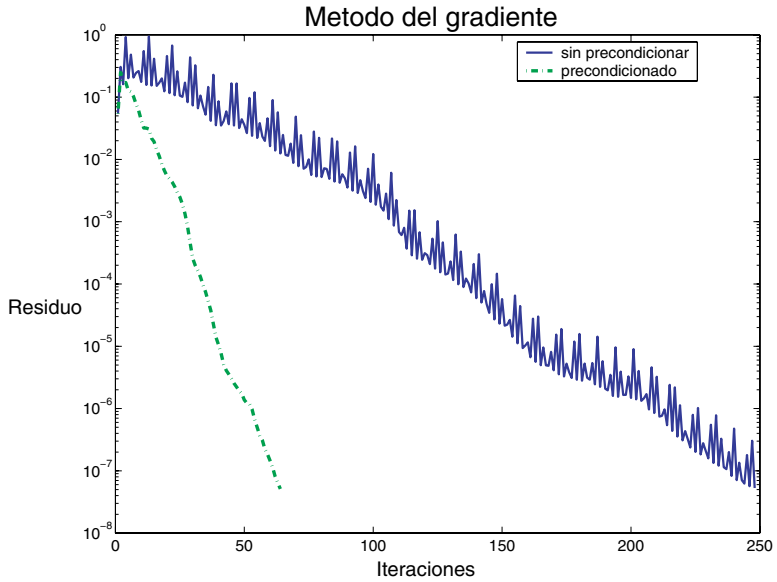


Figura 5.3: Precondicionamiento versus no precondicionamiento

aplicas el método con una tolerancia relativa de 10^{-7} *y un número máximo de 100 iteraciones. En* `relres` *se recoge el residuo relativo de la solución calculada (*$\|\mathbf{b} - A\mathbf{x}\|/\|\mathbf{b}\|$*), en* `resvec` *un historial del residuo en cada paso y en* `flag` *una variable que informa sobre qué ha sucedido en la aplicación del método. Así* `flag=1` *indica que se ha alcanzado el número máximo de iteraciones sin convergencia, mientras que si* `flag=0` *entonces ha habido convergencia. Para ver el historial de la convergencia del método se puede ejecutar*[26]

```
>> semilogy(resvec);
```

Observa cómo es necesario o aumentar el número de iteraciones máximas o disminuir la tolerancia para obtener convergencia.

A continuación vamos a utilizar un precondicionador muy sencillo basado en la descomposición de Cholesky incompleta. Teclea

```
>> R=cholinc(a,'0');
```

De nuevo, con los comandos habituales puedes ver tanto la forma como el número de entradas no nulas de R*.*

Modificando las instrucciones anteriores con

```
>>[x2,flag2, relres2,iter2,resvec2] = pcg(a,b,[],100,R',R);
```

estás aplicando el método del Gradiente Conjugado precondicionado con R*. ¿Disminuye el número de iteraciones? (ver* `iter2`*). ¿Y el tiempo de cálculo?. ¿Puedes ser ahora más exigente con la tolerancia?.*

Ejercicio 22 *Con la orden* `helpwin` *lee la ayuda que proporciona* Matlab *para* `pcg`*. ¿Qué otros métodos iterativos están implementados?. Consulta también la ayuda de* `cholinc` *y* `luinc`*.*

[26] Observa que el eje OY de la gráfica es logarítmico.

Índice de figuras

Índice general