# Métodos numéricos para sistemas de ecuaciones

# (Prácticas)

**Damián Ginestar Peiró**


UNIVERSITAT
POLITÈCNICA
DE VALÈNCIA

**UNIVERSIDAD POLITÉCNICA DE VALENCIA**

# Índice general

# Práctica 4

# Métodos iterativos

## 4.1. Métodos iterativos básicos

Los métodos iterativos para la resolución de sistemas de ecuaciones lineales suelen utilizarse para problemas de gran dimensión, ya que usan menos memoria y suelen ser más rápidos que los métodos directos. A continuación, recordaremos alguno de los métodos ietrativos más sencillos.

Se parte de un sistema de ecuaciones de la forma

$$Ax = b \ ,$$

y realizamos la descomposición de la matriz de coeficientes

$$A = D - E - F \ ,$$

donde $D$ es la diagonal de $A$, $-E$ es la parte estrictamente triangular inferior de $A$ y $-F$ es la parte estrictamente triangular superior. Se supone que los elementos de $D$ son todos no nulos.

El método de Jacobi se basa en iteraciones de la forma

$$Dx^{k+1} = (E + F)x^k + b \ ,$$

o sea,

$$x^{k+1} = D^{-1}(E + F)x^k + D^{-1}b \ .$$

Otro método similar al método de Jacobi es el método de Gauss-Seidel. Este método en componentes se escribe de la siguiente forma

$$b_i - \sum_{j=1}^{i-1} a_{ij}x_j^{k+1} - a_{ii}x_i^{k+1} - \sum_{j=i+1}^{n} a_{ij}x_j^k = 0 \ ,$$

para $i = 1, 2, \ldots, n$.

En forma matricial el método de Gauss-Seidel se escribirá como

$$(D - E)x^{k+1} = Fx^k + b \ .$$

Para implementar este método hay que resolver un sistema triangular inferior.

Análogamente, se puede definir otro método de Gauss-Seidel de la forma,

$$(D - F)x^{k+1} = Ex^k + b \ ,$$

donde se tendría que resolver un sistema triangular superior.

Por otra parte, podemos definir otra descomposición de la matriz $A$ de la forma

$$\omega A = (D - \omega E) - (\omega F + (1 - \omega)D) \ ,$$

que da lugar al método iterativo conocido como el método SOR (successive over relaxation)

$$(D - \omega E)x^{k+1} = (\omega F + (1 - \omega)D)x^k + \omega b \ ,$$

donde $\omega$ es un parámetro que puede tomar distintos valores y que sirve para mejorar la convergencia del método.

Análogamente, se puede definir otro método SOR de la forma

$$(D - \omega F)x^{k+1} = (\omega E + (1 - \omega)D)x^k + \omega b \ .$$

Un método SOR simétrico, SSOR, viene definido por las ecuaciones

$$\begin{aligned}(D - \omega E)x^{k+1/2} &= (\omega F + (1 - \omega)D)x^k + \omega b \ , \\ (D - \omega F)x^{k+1} &= (\omega E + (1 - \omega)D)x^{k+1/2} + \omega b \ .\end{aligned}$$

Por ejemplo, la siguiente función presenta una implementación sencilla del método de Gauss-Seidel.

```
function [x]=gaussseidel(matriz,vector)
% esta rutina implementa el metodo de Gauss-Seidel
% basico.
%
tol=1.e-4;
itmax=1000;
[n,n]=size(matriz);
x0=zeros(n,1);
```

```
it=0;
error=1000.0;

% calculo de las matrices
D=diag(diag(matriz));
E=-(tril(matriz)-D);
F=-(triu(matriz)-D);

while it<=itmax & error >tol
it=it+1
x=(D-E)\(F*x0+vector);
error=norm(x-x0)/ norm(x)
x0=x;
end
disp('el numero de iteraciones es')
disp(it)
disp('el error es')
disp(error)
```

## 4.2. Precondicionadores y métodos iterativos

La velocidad de convergencia de los métodos iterativos depende de las propiedades espectrales de la matriz del sistema. Así si $M$ es una matriz invertible que se aproxima de cierta manera a la matriz del sistema, $A$, los sistemas

$$\begin{aligned} Ax &= B \ , \\ M^{-1}Ax &= M^{-1}b \ , \end{aligned}$$

tienen las mismas soluciones. No obstante, es posible que las propiedades espectrales de $M^{-1}A$ sean más favorables que las de $A$. Al sistema

$$M^{-1}Ax = M^{-1}b \ ,$$

se le llama sistema precondicionado por la izquierda. Hay otras posibles estrategias para el precondicionado de un sistema como usar un precondicionador por la derecha o combinar iteraciones de distintos métodos iterativos.

Una posible elección de $M$, que se denomina precondicionador del sistema, es el precondiconador de Jacobi

$$m_{ij} = \begin{cases} a_{ij} & \text{si } i = j \\ 0 & \text{si } i \neq j \end{cases} \ .$$

Otro tipo de precondicionadores, en general, más eficientes son los basados en descomposiciones $LU$ incompletas de la matriz $A$.

Si se tiene una matriz simétrica y definida positiva, $A$, como matriz de coeficientes de un sistema, un método iterativo bastante eficiente en general para resolver el sistema es el método del *Gradiente Conjugado*. Este método sigue una iteración de la forma:

$$x^i = x^{i-1} + \alpha_i p^i \ ,$$

o, equivalentemente, para el residuo,

$$r^i = b - Ax^i \ ,$$

se tiene la iteración

$$r^i = r^{i-1} - \alpha_i q^i \ , \quad q^i = Ap^i \ .$$

El coeficiente $\alpha_i$ se elige como

$$\alpha_i = \frac{\left(r^{i-1}\right)^T r^i}{p^{iT} A p^i} \ ,$$

y la dirección de búsqueda se actualiza de la forma

$$p^i = r^i + \beta_{i-1} p^{i-1} \ ,$$

donde

$$\beta_i = \frac{\left(r^i\right)^T r^i}{\left(r^{i-1}\right)^T r^{i-1}} \ .$$

Una implementación del método del Gradiente Conjugado precondicionado mediante un precondicionador $M$ viene dada por el Algoritmo 1.

## 4.3. Métodos iterativos en Matlab

Matlab tiene implementadas distintas funciones para la reolución de sistemas de ecuaciones mediante métodos iterativos como, por ejemplo, la función `pcg( )` para matrices simétricas y definidas positivas, utilizando el método del gradiente precondicionado.

Las distintas formas que admite la llamada de esta función las podéis consultar en la ayuda de Matlab. Una llamada bastante general de esta función es de la forma

**Algoritmo 1** Método del gradiente conjugado precondicionado

1: Calcular $r^0 = b - Ax^0$ a partir de un vector inicial $x^0$.
2: **for** $i = 1, 2, \ldots$ **do**
3: Resolver $Mz^{i-1} = r^{i-1}$
4: $\rho_{i-1} = (r^{i-1})^T z^{i-1}$
5: **if** $i = 1$ **then**
6: $p^1 = z^0$
7: **end if**
8: **if** $i \neq 1$ **then**
9: $\beta_{i-1} = \rho_{i-1}/\rho_{i-2}$
10: $p^i = z^{i-1} + \beta_{i-1} p^{i-1}$
11: **end if**
12: $q^i = Ap^i$
13: $\alpha_i = \rho_{i-1}/p^{iT} q^i$
14: $x^i = x^{i-1} + \alpha_i p^i$
15: $r^i = r^{i-1} - \alpha_i q^i$
16: Comprobar el criterio de parada. Si se satisface salir.
17: **end for**

```
[x,flag,relres,iter,resvec]=pcg(A,b,tol,maxit,M1,M2,x0)
```

Esta función trata de resolver un sistema de ecuaciones lineales de la forma $Ax = b$, donde $A$ es una matriz dispersa simétrica y definida positiva. Como salidas esta función tiene `x` que es el vector solución. `flag` es un entero que nos indica distintas condiciones de terminación de la función. Así, si se cunple

`flag`=0, entonces la función ha convergido con la tolerancia `tol` deseada con un número menor de iteraciones que las iteraciones máximas `maxit`.

`flag`=1, entonces se han hecho el número máximo de iteraciones sin conseguirse la convergencia.

`flag`=2, el precondicionador `M`=`M1*M2` está mal condicionado.

`flag`=3, la función `pcg( )` se ha 'estancado', o sea, dos iteraciones consecutivas han producido el mismo resultado.

`flag`=4, alguna de las cantidades calculadas en el proceso se ha hecho demasido grande o demasiado pequeña para seguir el cálculo.

La salida `relres` corresponde a un error relativo asociado al residuo

$$\texttt{relres} = \left( \frac{\|b - Ax\|}{\|b\|} \right) .$$

`iter` devuelve el número de iteraciones necesarias para alcanzar la convergencia pedida. Y `resvec` nos devuelve un vector con las normas del residuo en cada iteración incluyendo $\|b - Ax_0\|$.

En cuanto a las entradas, `A` es la matriz del sistema. `b` es el vector independiente. `tol` es la tolerancia requerida. Si no se especifica, se usa el valor de defecto `tol`= $10^{-6}$.

`maxit` es el número de iteraciones máximas permitidas. El valor por defecto es el mínimo de la dimensión del sistema y 20.

`M1` y `M2` son dos matrices tales que el precondicionador del sistema se calcula como `M=M1*M2` y, de este modo, la función `pcg( )` resuelve el sistema precondiconado por la izquierda, o sea, el sistema

$$M^{-1}Ax = M^{-1}b \ .$$

Si se usa `M1=M2=[]` la función `pcg( )` no usa precondicionador.

Por último `x0` es el valor inicial de la solución. Si no se indica, Matlab usa el vector de ceros como valor por defecto.

En Matlab hay implementados diversos métodos de Krylov. Por ejemplo, se tienen las funciones `gmres()`, `bicg()`, `bicgstab()` para matrices no simétricas.

Así, si introduciomos el código:

```
matriz=gallery('poisson',50);
spy(matriz)
title('poisson␣50')
```

obtenemos la matriz asociada a un problema de Poisson, que se muestra en la Figura 4.1.

Podemos comparar la velocidad de convergencia de distintos métodos iterativos, para ello, escribimos:

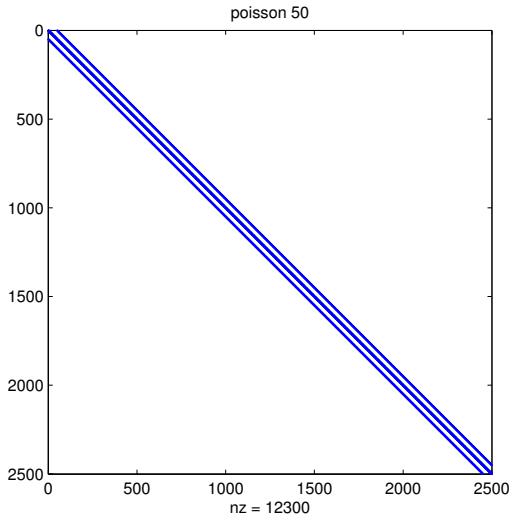



Figura 4.1: Matriz de Poisson.

```
 b=ones(length(matriz),1);
 max_it = 1000;
 tol = 1.e-5;
 restrt =50;

[x,flag,relres,iter,resvec1]=gmres(matriz,b,restrt,tol,max_it);
[x,flag,relres,iter,resvec2]=bicg(matriz,b,tol,max_it);
[x,flag,relres,iter,resvec3]=bicgstab(matriz,b,tol,max_it);
rescec3=resvec3(1:2:end);

 figure
 semilogy(1:length(resvec1),resvec1,'r',1:length(resvec2),...
                resvec2,'b',1:length(resvec3),resvec3,'k')
 legend('gmres', 'bicg', 'bicgstab')
 grid
 title('convergencia␣poisson␣50')
```

obteniendo la gráfica mostrada en la Figura 4.2. Hay que tener en cuenta que en la función `bicgstab()` el vector `resvec` nos muestra las normas de los residuos cada media iteración, mientras que en los otros métodos `resvec` nos da las normas de los residuos por iteración, con lo cual, para medir el coste de cada método, habría que tener en cuenta el coste por iteración.

Podemos repetir el proceso para la matriz `weston0479` del Matrix Market. El patrón de esta matriz se muestra en la Figura 4.3.

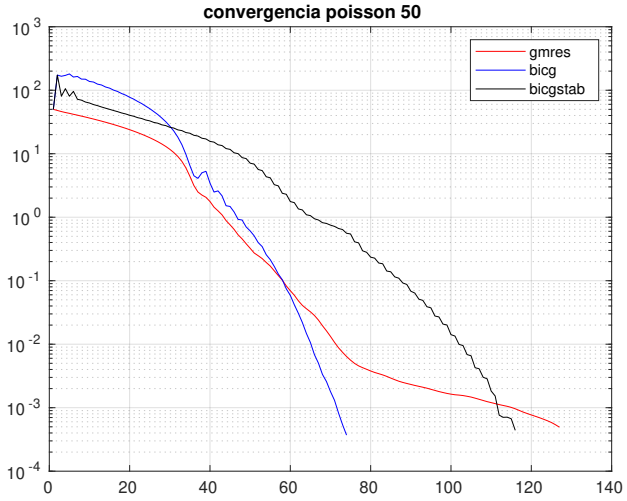



Figura 4.2: Velocidad de convergencia de distintos métodos para la matriz de Poisson.

Con las siguientes instrucciones:

```
matriz=mmread('west0479.mtx');
b=ones(length(matriz),1);

max_it = 500;
tol = 1.e-6;
restrt =50;
[x,flag,relres,iter,resvec2] = bicg(matriz,b,tol,max_it);
[x,flag,relres,iter,resvec3] = bicgstab(matriz,b,tol,max_it);
 resvec3=resvec3(1:2:end);

figure
semilogy(1:length(resvec2),resvec2,'b',...
1:length(resvec3),resvec3,'k')
xlabel('iteraciones')
ylabel('error')
legend('bicg', 'bicgstab')
grid
title('convergencia␣weston')
```

obtenemos la velocidad de convergencia de los métodos `bicg()` y `bicgstab()` para esta matriz que se muestra en la Figura 4.4. Se observa pues que estos métodos no consiguen encontrar la solución del sistema.

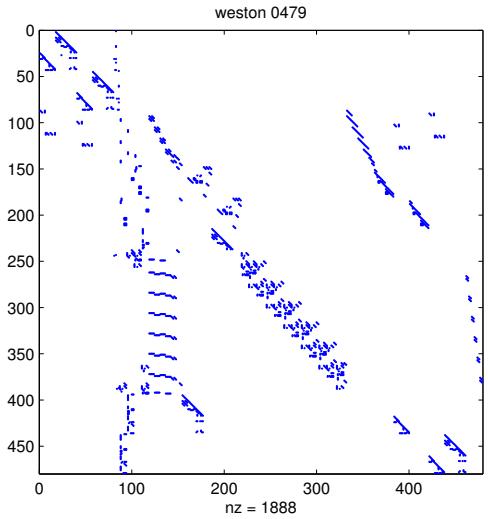



Figura 4.3: Matriz Weston0479 del Matrix Market

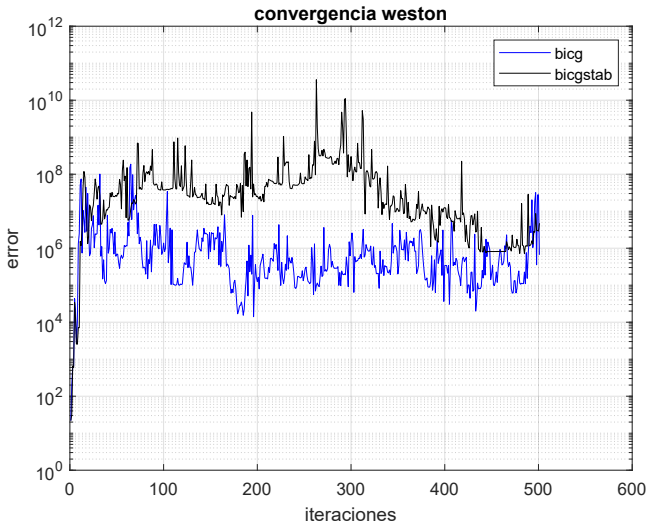



Figura 4.4: Velocidad de convergencia para la matriz Weston 0479.

### 4.3.1. Método BiCG

El método del bigradiente conjugado (BiCG) busca aproximaciones de la solución en el espacio de Krylov $K_n(A, r_0)$ con residuos ortogonales a $K_n\left(A^T, r_0\right)$. Para poder describir el algoritmo del método debemos estudiar cómo generar bases de estos subespacios que formen un sistema biortogonal. El algoritmo de biortogonalizacion de Lanczos construye estas bases recursivamente y el método BiGC y sus propiedades se deducen de este proceso.

Si estamos interesados en resolver un sistema dual de la forma $A^T x^* = b$, el método BiCG también nos da la solución. Los métodos que biortogonalizan bases tienen la ventaja de que usan fórmulas de recurrencia reducidas y, por tanto, el almacenamiento no aumenta con el número de iteraciones.

El método de biortogonalización de Lanczos nos da como resultado dos matrices $V_n = [v_1|\cdots|v_n]$ y $W_n = [w_1|\cdots|w_n]$ tales que los vectores $v_i$ son base de $K_n(A, r_0)$ y los vectores $w_i$ son base de $K_n\left(A^T, r_0\right)$ tales que

$$W_n^T V_n = I_n, \tag{4.1}$$

y

$$W_n^T A V_n = T_n, \tag{4.2}$$

donde $I_n$ es la matriz identidad de tamaño $n$ y

$$T_n = \begin{pmatrix} \alpha_1 & \beta_2 & 0 & \cdots & 0 \\ \delta_2 & \alpha_2 & \beta_3 & \cdots & 0 \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ 0 & \cdots & \delta_{n-1} & \alpha_{n-1} & \beta_n \\ 0 & \cdots & 0 & \delta_n & \alpha_n \end{pmatrix}.$$

de forma que se satisface que

$$\begin{aligned} AV_n &= V_n T_n + \delta_{n+1} v_{n+1} e_n^T, \\ AW_n^T &= W_n T_n^T + \beta_{n+1} w_{n+1} e_n^T. \end{aligned}$$

El proceso de biortogonalización de Lanczos se resume en el Algoritmo 2.

El método BiCG se basa en el proceso de biortogonalización de Lanczos partiendo de los vectores $v_1 = \frac{r_0}{\|r_0\|_2}$ y $w_1$ arbitrario tal que $(v_1, w_1) \neq 0$. Se suele utilizar $w_1 = v_1$. Este método se puede estructurar en el Algoritmo 3.

Una implementación de este método viene dada en la función `mybicg`.

**Algoritmo 2** Método de biortogonalización de Lanczos

Consideramos $v_1$ y $w_1$ tales que $(v_1, w_1) = 1$.
Escogemos $\beta_1 = \delta_1 = 0$.
**for** k=1,...,n **do**
    $\alpha_k = (Av_k, w_k)$
    $\tilde{v}_{k+1} = Av_k - \alpha_k v_k - \beta_k v_{k-1}$
    $\tilde{w}_{k+1} = A^T w_k - \alpha_k w_k - \delta_k w_{k-1}$
    $\delta_{k+1} = |(\tilde{v}_{k+1}, \tilde{w}_{k+1})|^2$
    **if** $\delta_{k+1} = 0$ **then**
        **Stop**
    **end if**
    $\beta_{k+1} = \frac{(\tilde{v}_{k+1}, \tilde{w}_{k+1})}{\delta_{k+1}}$
    $w_{k+1} = \frac{\tilde{w}_{k+1}}{\beta_{k+1}}$
    $v_{k+1} = \frac{\tilde{v}_{k+1}}{\delta_{k+1}}$
**end for**

```
function [x,error] = mybicg(A,b,x0,tol,maxit)
%
%   funcion mybicg
x=x0;
r = b - A*x;
normr = norm(r);
errres = normr/norm(b);
rt = r;
rho = 1;
rho1=1;

p = r;
pt = rt;
q = A*p;
qt = A'*pt;

W = [pt];
AV = [q];

% bucle
error=[errres];
it=0;
```

**Algoritmo 3** Método BiCG.

1: Aproximación inicial $x_0$
2: $r_0 = b - Ax_0$
3: Elegir $r_0^*$ adecuadamente
4: $p_0 = r_0$, $p_0^* = r_0^*$
5: $n = 1$
6: Se fija la precisión y $maxit$
7: **while** error> precisión y $n < maxit$ **do**
8: $r_{n-1}^T r_{n-1}^*$
9: $w = p_{n-1}^T A \left(p_{n-1}^*\right)^T$
10: $\alpha_n \frac{z}{w}$
11: $x_n = x_{n-1} + \alpha_n p_{n-1}$
12: $x_n^* = x_{n-1}^* + \alpha_n p_{n-1}^*$
13: $r_n = r_{n-1} - \alpha_n A p_{n-1}$
14: $r_n^* = r_{n-1}^* - \alpha_n A^T p_{n-1}^*$
15: $w = r_n^T r_n^*$
16: $\beta_n = \frac{w}{z}$
17: $p_n = r_n + \beta_n p_{n-1}$
18: $p_n^* = r_n^* + \beta_n p_{n-1}^*$
19: cálculo del error
20: n=n+1
21: **end while**

```
while ((errres>=tol)&(it<=maxit))
     it=it+1;
     rho = r' * rt;
     alpha = rho / (q'*pt);
     x = x + alpha * p;
     errres=norm(b-A*x)/norm(b);
     error=[error,errres];
     r = r - alpha*q;
     rt = rt - alpha*qt;
     rho1 = r' * rt;
     beta = rho1/rho;
     p = r + beta * p;
```

```
      q = A * p;
      pt = rt + beta * pt;
      qt = A' * pt;
end
```

Matlab tiene implementadas un gran número de matrices test. Se pueden ver con la instrucción help gallery en la línea de comandos. La matriz *toeppen* es una matriz pentadiagonal Toeplitz dispersa. (Una matriz Toeplitz es una matriz cuadrada en la que los elementos de sus diagonales (de izquierda a derecha) son constantes).

La matriz toeppen se obtiene haciendo:

```
P = gallery(toeppen,N,A,B,C,D,E)
```

donde `N` es un número entero y `A,B,C,D,E` son escalares. `P` es la matriz dispersa `N` $\times$ `N` pentadiagonal Toeplitz con diagonales `P(3,1)=A`, `P(2,1)=B`, `P(1,1)=C`, `P(1,2)=D`, `P(1,3)=E`. Por defecto: `(A,B,C,D,E)=(1,-10,0,10,1)` que es una matriz de Rutishauser.

Un ejemplo de matriz Toeppens se obtiene con las instrucciones:

```
N=10000;
A=gallery('toeppen',N, 2,3, 8, 3.5, 4.5);
```

Podemos resolver uns sistema asociado a esta matriz Toeppen y comparar el funcionamiento de la funcion `mybicg` con la función `bicg` de Matlab, con el siguiente código:

```
b=zeros(N,1);
b(1)=1;
[x,flag,relres,iter,resvec] = bicg(A,b,tol, maxit);
[x,err1]=mybicg(A,b,x0,tol,maxit);
vec1=0:size(resvec,1)-1;
vec2=0:length(err1)-1;

semilogy(vec1,resvec/norm(b),'r*',vec2,err1,'k')
legend('bicgMatlab','mybicg')
xlabel('Iteraciones')
ylabel('Residuo relativo')
```

Al ejecutarlo se obtiene la gráfica mostrada en la Figura 4.5.

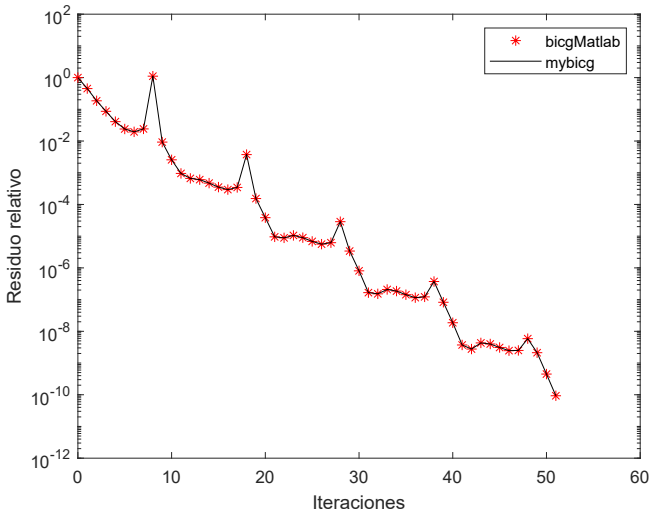



Figura 4.5: Velocidad de convergencia para la matriz Toeppens con las funciones `bicg` y `mybicg`.

### 4.3.2. Método GMRES

Uno de los métodos más populares para resolver un sistema de la forma

$$Ax = b,$$

donde la matriz $A$ es no simétrica, es el método GMRES que es un método de proyección basado en minimizar en cada iteración la norma euclídea del residuo. Se trata de encontrar

$$x_k = x_0 + Q_k y_k,$$

tal que la norma de $r_k$ sea mínima. Se tiene que,

$$r_k = b - Ax_k = r_0 - AQ_k y_k = \|r_0\| q_1 - AQ_k y_k$$

así

$$\begin{aligned} \|r_k\| &= \|\|r_0\| q_1 - AQ_k y_k\| \\ &= \|\|r_0\| Q_{k+1} e_1 - Q_{k+1} \underline{H}_k y_k\| \\ &= \|\|r_0\| e_1 - \underline{H}_k y_k\|. \end{aligned}$$

Resolviendo el problema sobredeterminado

$$\underline{H}_k y_k = \|r_0\| e_1$$

se tienen las iteraciones

$$x_k = x_0 + Q_k y_k$$

que minimizan el residuo.

El método básico GMRES se presenta en el Algoritmo 4.

**Algoritmo 4** Método GMRES.

1: Escoger un $x_0$; $r_0 = b - Ax_0$
2: **for** k=1,2,... **do** ▷ Algoritmo de Arnoldi
3: $y = Aq_k$
4: **for** $j = 1, \ldots, k$ **do**
5: $h_{jk} = q_j^T y$
6: $y = y - h_{jk} q_j$
7: **end for**
8: $h_{k+1\,k} = \|y\|_2$
9: **if** $h_{k+1\,k} = 0$ **then**
10: **stop**
11: **end if**
12: $q_{k+1} = y / h_{k+1\,k}$
13: Resolver $Hc_k = \|r_0\|_2 e_1^T$ ▷ Sistema rectangular
14: $x_k = x_0 + Q_k c_k$
15: **end for**

Una implementación del algoritmo GMRES, viene dada en la función `mygmres`.

```
function [x,normrn] = mygmres(A,b,x0,tol,maxit)
%
% Resuelve Ax = b con el metodo gmres
% input: A - m x m matriz
%        b - m x 1 vector
% output: x -  solucion aproximada
%         normrn - norm(b-A*x) en cada iteracion del algoritmo
%

Q = []; H = 0;
normb = norm(b);
Q(:,1) = (b-A*x0)/normb;
eres = norm(Q(:,1));
```

```
normrn=eres*norm(b);
% bucle
n=0;
while ((eres>=tol)&(n<=maxit))
 n=n+1;
% Metodo de Arnoldi
 v = A*Q(:,n);
 for j = 1:n
  H(j,n) = Q(:,j)'* v;
  v = v - H(j,n)*Q(:,j);
 end
 Hn = H(1:n,1:n);
 H(n+1,n) = norm(v);
 if H(n+1,n) == 0, break, end   % breakdown stop
 Q(:,n+1) = v/H(n+1,n);
 e1 = [1;zeros(n,1)];
 y = H\(normb*e1); %  Se usa la \ de Matlab
 eres=norm(H*y-normb*e1);
 normrn = [normrn,eres];          % norma residual
 x = x0+Q(:,1:n)*y;
end   % del while
end   % de la funcion
```

Utilizando el siguiente código podemos comparar el funcionamiento de la función `gmres` de Matlab y la implementación que tenemos en la función `mygmres`.

```
[x,flag,relres,iter,resvec] = gmres(A,b,6,tol, maxit);

x0=zeros(N,1);
[x,err1]=mygmres(A,b,x0,tol,maxit);

%
vec1=0:size(resvec,1)-1;
vec2=0:length(err1)-1;
semilogy(vec1,resvec/norm(b),'r*',vec2,err1/norm(b),'k')
legend('gmresMatlab','mygmres')
%
xlabel('Iteraciones')
ylabel('Residuo relativo')
```

Obtenemos la gráfica que se muestra en la Figura 4.6 La diferencia que se observa es debido a que la rutina de Matlab devuelve el error para cada una de las iteraciones internas que realiza ya que realiza un reinicio cada 5 iteraciones. Mientras que en la función `mygmres()` el método GMRES no se reinicia.

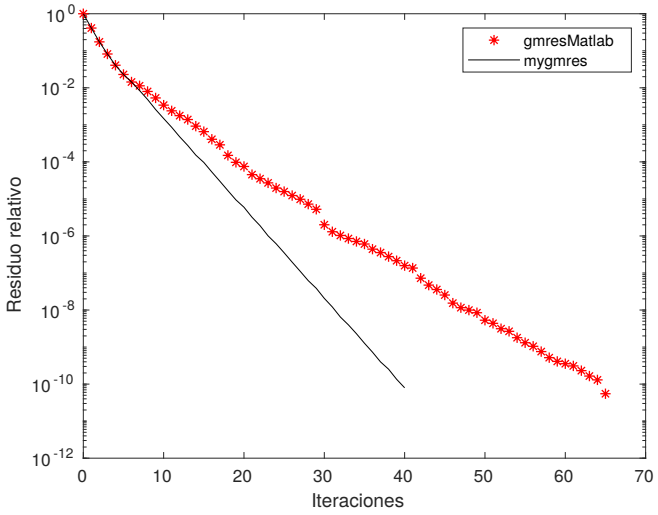



Figura 4.6: Velocidad de convergencia para la matriz Toeppens con las funciones `gmres` y `mygmres`.

## 4.4. Precondicionadores en Matlab

En Matlab se puede obtener el precondiconador de Jacobi de forma sencilla, así dada una matriz $A$, el precondicionador en formato de almacenamiento disperso se obteiene como

```
M= sparse(diag(diag(A)))
```

Si se tiene una matriz simétrica y definida positiva, $A$, un posible precondicionador es una descomposición de Choleski incompleta. Esta descomposición se puede calcular mediante la función `ichol( )`.

Dos posibles llamadas de esta descomposición son

```
R = ichol(A)
R = ichol(A,opts)
```

En el primer caso se obtiene una descomposición incompleta de Choleski que es simétrica y definida positiva y que no produce relleno, iChol(0). En el segundo caso se calcula la descomposicón incompleta de Choleski con las opciones que se indican en la estructura `opts`.

Si la matriz $A$ no es simétrica y definida positiva se utiliza como precondicionador la descomposición $LU$ incompleta de $A$. Esta descomposicón se

calcula mediante la función `ilu( )` que tiene un funcionamiento similar a la `ichol( )`. Dos posibles llamadas de la función `ilu( )` son:

```
[L,U] = ilu(A)
[L,U] = ilu(A,setup)
```

En el primer caso se obtiene una descomposición incompleta $ILU(0)$ de la matriz $A$. En el segundo caso calcula la descomposicón $LU$ incompleta de la matriz $A$ con las opciones que se indican en la estructura `setup`.

Supongamos que se quiere estudiar el efecto de usar un cierto precondiconador en las funciones definidas en Matlab para la resolución de sistemas mediante un método iterativo. Para ilustrar una posible estrategia a seguir consideraremos la matriz simétrica y definida positiva obtenida al discretizar la ecuación de Poisson,

```
A=gallery('poisson',50)
```

El precondiconador obtenido a partir de la descomposición de Choleski incompleta se calcula del siguiente modo

```
R=ichol(A)
```

Como término independiente del sistema se elige un vector de unos

```
b=ones(length(A),1)
```

Para resolver el sistema mediante el método del gradiente conjugado hacemos la llamada

```
[x,flag1,relres,iter1,resvec1]=pcg(A,b,1.e-8,50)
```

Para resolver el sistema mediante el método del gradiente conjugado con el precondicionador obtenido a partir de la descomposición de Choleski, hacemos la llamada

```
[x,flag2,relres,iter2,resvec2]=pcg(A,b,1.e-8,50,R',R)
```

Comparamos la evolución del error con el número de iteraciones para los dos casos mediante una gráfica en escala semilogarítmica

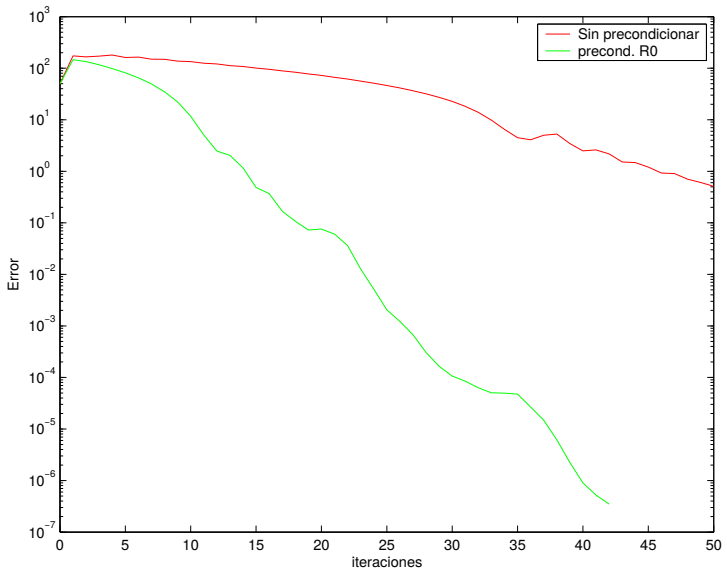



Figura 4.7: Evolución del error al resolver el sistema asociado a la matriz de Poisson.

```
semilogy(0:iter1,resvec1,'r',0:iter2,resvec2,'g')
legend('Sin precondicionar','precond. R0')
xlabel('iteraciones')
ylabel('Error')
```

obteniendo la gráfica mostrada en la Figura 4.7. Observamos pues que el sistema precondicionado converge más rápidamente que el sistema sin precondicionar.

Ahora utilizaremos la descomposición LU incompleta para precondicionar la matriz `Weston`. Para ello, se hace uso del código

```
matriz=mmread('west0479.mtx');
b=ones(length(matriz),1);

max_it = 500;
tol = 1.e-6;
restrt =50;

setup.type = 'ilutp';
setup.droptol = 0.000001;

[L,U]=ilu(matriz,setup);

[x,flag,relres,iter,resvec2] = bicg(matriz,b,tol,max_it,L,U);
```

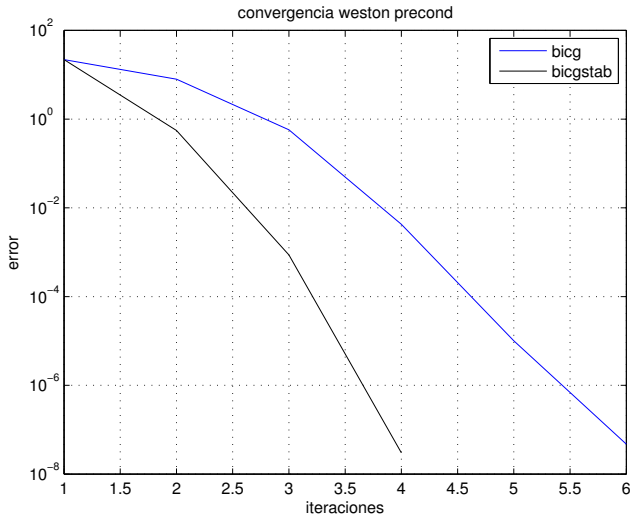



Figura 4.8: Evolución del error para la matriz Weston precondicionada.

```
[x,flag,relres,iter,resvec3] = bicgstab(matriz,b,tol,max_it,L,U);
figure
resvec3=resvec3(1:2:end);
semilogy(1:length(resvec2),resvec2,'b',...
1:length(resvec3),resvec3,'k')
legend('bicg', 'bicgstab')
grid
xlabel('iteraciones')
ylabel('error')
title('convergencia␣weston␣precond')
```

con el que se hace uso del precondicionador $ILU(0)$ que se ha calculado realizando permutaciones y con una tolerancia (opción `ilutp`).

Se obtiene la evolución del error que se muestra en la Figura 4.8.

En ocasiones no se dispone de la matriz de un sistema, pero si se puede obtener el producto matriz-vector a partir de una función. En el siguiente ejemplo vemos cómo es posible hacer una llamada al método `gmres` utilizando funciones para el producto matriz vector y para la aplicación del precondicionador.

Si introducimos

```
n = 21;
A = gallery('wilk',n);
spy(A)
```

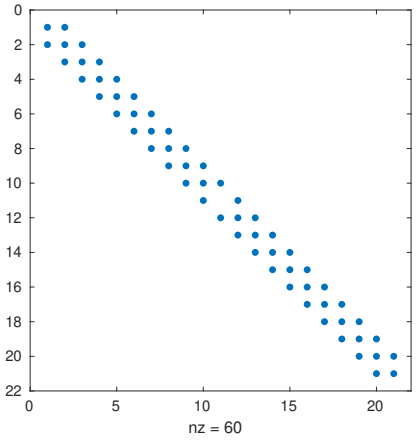



Figura 4.9: Matriz de Wilkinson.

obtenemos la matriz de Wilkinson, que es una matriz tridiagonal con pares de autovalores casi iguales, que se muestra en la Figura 4.9.

Si usamos el método `gmres` para resolver un sistema asociado a esta matriz podemos hacer

```
b = sum(A,2);
tol = 1e-12;
maxit = 15;
M = diag([10:-1:1 1 1:10]);
x = gmres(A,b,10,tol,maxit,M)
```

donde se ha utilizado un precondicionador de Jacobi.

Alternativamente, podemos definir una función que realice el producto matriz vector

```
function y = afun(x,n)
   y = [0; x(1:n-1)] + [((n-1)/2:-1:0)';...
         (1:(n-1)/2)'].*x+[x(2:n); 0];
end
```

y otra función que aplique el precondicionador sobre un vector

```
function y = mfun(r,n)
y = r ./ [((n-1)/2:-1:1)'; 1; (1:(n-1)/2)'];
end
```

y la resolución del sistema se hace de forma alternativa mediante la llamada

```
x1 = gmres(@(x)afun(x,n),b,10,tol,maxit,@(x)mfun(x,n))
```

## 4.5. Ejercicios

1. Construye una función de Matlab que implemente el método de Jacobi y que devuelva la solución para una tolerancia dada así como el número de iteraciones necesarias para alcanzar la solución. Construye otra función de Matlab que implemente el método de Gauss Seidel. Compara el método de Jacobi y el método de Gauss Seidel para la resolución del sistema

$$\begin{aligned} 10x_1 - x_2 + 2x_3 &= 6 \ , \\ -x_1 + 11x_2 - x_3 + 3x_4 &= 25 \ , \\ 2x_1 - x_2 + 10x_3 - x_4 &= -11 \ , \\ 3x_2 - x_3 + 8x_4 &= 15 \ . \end{aligned}$$

   Consideremos los sistemas de ecuaciones

$$A_i x = b \ ,$$

   con

$$A_1 = \begin{pmatrix} 3 & 2 & 1 \\ 2 & 3 & 2 \\ 1 & 2 & 3 \end{pmatrix} \ , \quad A_2 = \begin{pmatrix} 1 & 2 & -2 \\ 1 & 1 & 1 \\ 2 & 2 & 1 \end{pmatrix} \ , \quad b = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} \ ,$$

   comprueba que para la matriz $A_1$ el método de Jacobi diverge mientras que el método de Gauss-Seidel converge y que para la matriz $A_2$ ocurre alrevés.

2. Dado el sistema

$$\begin{aligned} 4x_1 + 3x_2 &= 24 \\ 3x_1 + 4x_2 - x_3 &= 30 \\ -x_2 + 4x_3 &= -24 \end{aligned}$$

   compara 5 iteraciones del método de SOR y el método SSOR tomando $x_0 = (0, 0, 0)^T$, y $\omega = 1.25$. Determina empíricamente cuál sería el parámetro $\omega$ óptimo para cada método si se quiere resolver el sistema.

3. La matriz obtenida para la discretización de la ecuación de Poisson

$$-\Delta u = f \ ,$$

sobre el rectángulo unidad sobre una malla regular da lugar a una matriz dispersa $A$ de tamaño $n \times n$. Esta matriz se puede generar en Matlab mediante la instrucción `A=gallery('poisson',n)`.

Compara el número de iteraciones necesario para resolver el sistema $Au = f$, donde el vector $f$ viene dado por

$$f = \frac{1}{(n+1)^2} \begin{pmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{pmatrix} ,$$

si se utiliza el método de Jacobi y el método de Gauss-Seidel. Usa como criterio de parada que el residuo relativo sea menor que $10^{-5}$ y el vector de ceros como vector inicial. Construye una gráfica donde se compare el número de iteraciones frente a la dimensión para los dos métodos iterativos.

4. Implementa el método del Gradiente Conjugado Precondiconado (GCP), a partir del algoritmo que hemos presentado en la práctica, en una función de Matlab, que tenga la estructura

   ```
   [x,err,iter]=migcp(A,b,x0,tol,M)
   ```

   Compara su funcionamiento con la función `pcg( )` implementada ya en Matlab. Utiliza para ello la matriz generada con la instrucción `A=gallery('poisson',50)`.

5. Una matriz simétrica y definida positiva se puede obtener con la intrucción

   ```
   matriz=gallery('minij',50)
   ```

   Comprueba que la matriz es simétrica y definida positiva (puedes usar la función `chol( )`). Resuelve un sistema de ecuaciones lineales asociado a la matriz construida, usando como término independiente el vector de unos, mediante el método GCP. Estudia el efecto de usar el precondiconador de Jacobi.

6. Como ya vimos los vectores almacenados en los ficheros `V.dat`, `I.dat` y `J.dat` definen una matriz dispersa en formato coordenando, $A$. Compara el funcionamiento de los métodos BICG, BICGSTAB y GMRES para la resolución de un sistema

   $$Ax = b ,$$

donde $b$ es el vector de unos.

Compara el funcionamiento de estos métodos al utilizar el precondicionador obtenido al calcular la descomposición $LU$ incompleta de $A$ usando las tolerancias $10^{-3}$, $10^{-4}$ y $10^{-5}$. Para comparar el funcionamiento, haz una gráfica semilogarítmica del error en función de las iteraciones

7. La instrucción `A=gallery('tridiag', 10000, -1, 4, -1)` produce una matriz tridiagonal. Construye una función de Matlab que implemente el método de descenso rápido que se ha visto en teoría. Compara la convergencia del método de descenso rápido con la del método del gradiente conjugado para la matriz A.

8. Considera la matriz no simétrica que se almacena en el fichero `pores2.dat` en formato coordenado. Esta matriz corresponde a la matriz de coeficientes de un problema de modelización de un depósito de petróleo. Resuelve un sistema de ecuaciones asociado a esta matriz tomando como término independiente el vector de unos. Utiliza para ello el método GMRES(m). Considera los valores de `m=size(A,1)`, `m=80`, `m=70`. Compara la evolución del error en función de las iteraciones en los tres casos. Escala la matriz para obtener una matriz cuya diagonal principal sean unos y estudia la convergencia del método GMRES(m) en este caso. Investiga el efecto de usar una descomposición incompleta de la matriz A como precondicionador.

9. Construye dos funciones de Matlab que implementen dos métodos de Gauss-Seidel, (hacia delante y hacia atrás), que devuelvan la solución y un vector donde se tenga el error cometido en cada iteración.

   Considera la ecuación de convección-difusión

   $$-\epsilon\nabla^2 u + v_1\frac{\partial u}{\partial x} + v_2\frac{\partial u}{\partial y} = \sin(\pi x)\sin(\pi y), \quad (x,y) \in [0,1]\times[0,1]$$

   con condiciones de contorno nulas. Usa un método de diferencias finitas para discretizar el problema y compara el funcionamiento de los dos métodos de Gauss-Seidel si se discretiza el dominio utilizando una malla igualmente espaciada de $(N+1)\times(N+1)$ nodos con $N = 800, 900$ y 1000, utilizando los valores $\epsilon = 0.01$, $v_1 = 1$, $v_2 = -1$.

10. Una ecuación de Fredholm de segunda especie es una ecuación de la forma

    $$\varphi(x) = f(x) + \int_a^b dt\, K(x,t)\varphi(t).$$

Si se discretiza el intervalo $[a, b]$ mediante subintervalos igualmente espaciados

$$x_i = a + ih, \quad h = \frac{b-a}{N}, \quad i = 0, 1, \ldots, N,$$

y se usa la aproximación de los trapecios para la integral definida, se obtiene un sistema de ecuaciones para $\varphi(x_i)$. Resolver, mediante un método iterativo, el sistema resultante para la ecuación

$$\varphi(x) = x + \int_0^1 dt\, 2e^{x+t}\varphi(t),$$

y comparar el resultado obtenido con la solución analítica

$$\varphi(x) = x + \frac{2e^x}{2 - e^2}.$$

11. El método CGNR (Conjugate Gradient Method on the Normal Equations) se puede implementar en el siguiente algoritmo:

Aproximación inicial $x_0$. $r_0 = b - Ax_0$, $p_0 = A^T r_0$

Mientras $\|r_{j-1}\| / \|r_0\| \geq \varepsilon$, $(j = 1, 2, \ldots)$, hacer

$$\alpha_j = \frac{\langle A^T r_j, A^T r_j \rangle}{\langle Ap_j, Ap_j \rangle}$$
$$x_{j+1} = x_j + \alpha_j p_j$$
$$r_{j+1} = r_j - \alpha_j A p_j$$
$$\beta_j = \frac{\langle A^T r_{j+1}, A^T r_{j+1} \rangle}{\langle A^T r_j, A^T r_j \rangle}$$
$$p_{j+1} = A^T r_{j+1} + \beta_j p_j$$

Fin

Implementa el método en una función de Matlab y compara su funcionamiento con el método GMRES utilizando la matriz **FS 183 1** del Matrix Market.

12. Un precondicionador polinomial para un sistema de la forma

$$Ax = b$$

viene dado por el truncamiento de la serie

$$A^{-1} = (I - (I - A)) = I + \sum_{k \geq 1} (I - A)^k,$$

que funciona bien cuando $\|I - A\| < 1$. Construye una función que implemente este tipo de precondiconador y estudia su funcionamiento resolviendo un sistema asociado a la ecuación de Poisson 2D.

13. Dado el siguiente problema asociado a la ecuación de Love, que es de utilidad en electrostática,

$$u(x) - \frac{1}{\pi}\int_{-1}^{1} \frac{1}{1+(x-t)^2} u(t)\, dt = 1.$$

Para obtener una aproximación numérica de la solución, se divide el intervalo $[-1, 1]$ mediante una malla uniforme, $x_i = h(i - 1/2)$, con $h = 2/n$. Esto permite aproximar la solución resolviendo el sistema de ecuaciones lineales

$$\sum_{j=1}^{n} a_{ij} u_j = 1, \quad 1 \le i \le n,$$

donde

$$a_{i,j} = \delta_{i,j} - \frac{h}{\pi(1+(i-j)^2 h^2)}, \ 1 \le i, j \le n,$$

y la delta de Kronecker es

$$\delta_{i,j} = \begin{cases} 1 & \text{si } i = j \\ 0 & \text{si } i \neq j \end{cases}$$

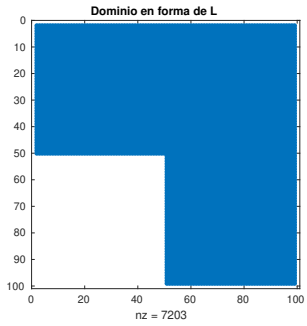
Estudia la eficiencia de los métodos `gmres()`, `bicg()` y `bicgstab()` si $n = 100, 200, 500$.

14. El número de condición de la matriz de Poisson es proporcional a $h^{-2}$ donde $h$ es el paso espacial de la discretización. Verifica la dependencia del número de iteraciones necesarias con $\sqrt{\kappa(A)}$ al resolver un sistema mediante el método del gradiente conjugado con esta matriz. Para ello, resuelve cuatro sistemas $Ax = b$, generando $A$ con la instrucción `A=delsq(numgrid(('S',nx+2))` con $nx = 100, 200, 300$ y $400$ y $b$ el vector de unos. De este modo se obtiene la matrix de Poisson en el rectángulo $[-1, 1] \times [-1, 1]$.

Genera una tabla con el número de iteraciones necesarias en función de $h$.

Compara los resultados obtenidos si se utiliza como precondicionador la descomposición IC con una tolerancia de $10^{-2}$.

15. La instrucción: `G = numgrid('L',n)` genera una malla en forma de L, como se muestra en la figura:




y la instrucción: `A = delsq(G)` genera la matriz asociada al laplaciano bidimensional en esta malla. Construye una función de matlab que implemente el método SSOR que devuelva un vector con los residuos relativos en cada iteración. Compara el funcionamiento de esta función con la función pcg() para resolver un sistema $Ax = b$ donde $A$ se ha generado a partir de la malla $G$ con $n = 10000$ y $b$ es el vector de unos.

16. Dado un sistema

$$Ax = b,$$

donde $A$ es una matriz no simétrica, se puede considerar el problema de las ecuaciones lineales

$$A^T Ax = A^T b.$$

El resultado de aplicar el método del gradiente conjugado a este nuevo sistema da lugar al método CGNR, cuyo algoritmo es el dado en el algoritmo 5.

Comparar el funcionamiento de este método con el del método GMRES utilizando la matrix `e40r1000` del MatrixMarket y el término independiente que proporciona la colección.

17. El número de condición de la matriz de Poisson es proporcional a $h^{-2}$ donde $h$ es el parámetro de discretización de la malla. Verificar la dependencia del número de iteraciones del método gradiente conjugado con la raíz cuadrada del número de condición de $A$, $\sqrt{\kappa(A)}$, resolviendo 4 sistemas lineales $Ax = b$ con `A=delsq(numgrid('S',nx+2))` y $nx = 100, 200, 300, 400$. En particular

**Algoritmo 5** Método CGN

1: Calcular $r^0 = b - Ax^0$, $z_0 = A^T r_0$, $p_0 = z_0$, a partir de un vector inicial $x^0$.
2: **for** $i = 1, 2, \ldots$ **do**
3: $\quad w_i = Ap_i$
4: $\quad \alpha_i = \|z_i\|^2 / \|w_i\|^2$
5: $\quad x_{i+1} = x_i + \alpha_i p_i$
6: $\quad r_{i+1} = r_i - \alpha_i w_i$
7: $\quad$ Comprobar el criterio de parada. Si se satisface salir.
8: $\quad z_{i+1} = A^T r_{i+1}$
9: $\quad \beta_i = \|z_{i+1}\|_2^2 \,/\, \|z_i\|_2^2$
10: $\quad p_{i+1} = z_{i+1} + \beta_i p_i$
11: **end for**

(a) Establecer el término independiente $b$ correspondiente a la solución exacta $x$ con componentes

$$x_i = \frac{1}{\sqrt{i}}, \quad i = 1, \ldots, n.$$

Utiliza tol $= 10^{-8}$ y 2000 como número máximo de iteraciones.

(b) Resuelve los cuatro sistemas lineales con (a) el método del gradiente conjugado sin precondicionar, (b) el método del gradiente precondicionado usando $IC(0)$, (c) el método del gradiente precondicionado con IC y `droptol` $= 10^{-2}$ y (d) el método del gradiente precondicionado con IC y `droptol` $= 10 - 3$.

(c) Elaborar una Tabla con los valores de $\sqrt{\kappa(A)}$ para los distintos valores de $nx$ y el número de iteraciones del (P)CG.

- Dado el problema precondicionado

$$M^{-1}Ax = M^{-1}b,$$

se puede buscar el precondicionador $M$ como la matriz $N$ que resulta de resolver el problema de optimización

$$\min_{M \in S} \|MA - I\|_F = \|NA - I\|_F \ ,$$

donde la norma de Frobenius de una matriz real es

$$\|A\|_F = \left(\operatorname{tr}\left(A^T A\right)\right)^{\frac{1}{2}} = \left(\sum_{j=1}^{n}\sum_{k=1}^{n} |a_{j,k}|^2\right)^{\frac{1}{2}},$$

y $S$ es un subespacio de las matrices cuadradas $n \times n$. Si tomamos $S$ el subespacio de las matrices diagonales $n \times n$, la solución del problema de optimización es

$$N = \operatorname{diag}\left(\frac{a_{11}}{\|e_1^T A\|^2}, \frac{a_{22}}{\|e_2^T A\|^2}, \ldots, \frac{a_{nn}}{\|e_n^T A\|^2}\right),$$

donde $e_i$ son los vectores de la base canónica. Construye una función que multiplique la inversa de este precondicionador por un vector y compara su funcionamiento con el precondicionador de Jacobi, para resolver un problema de la forma

$$Ax = b$$

donde $A$ es la matriz de Wilkinson de orden 21 y $b_i = \sum_{j=1}^{n} a_{ij}$