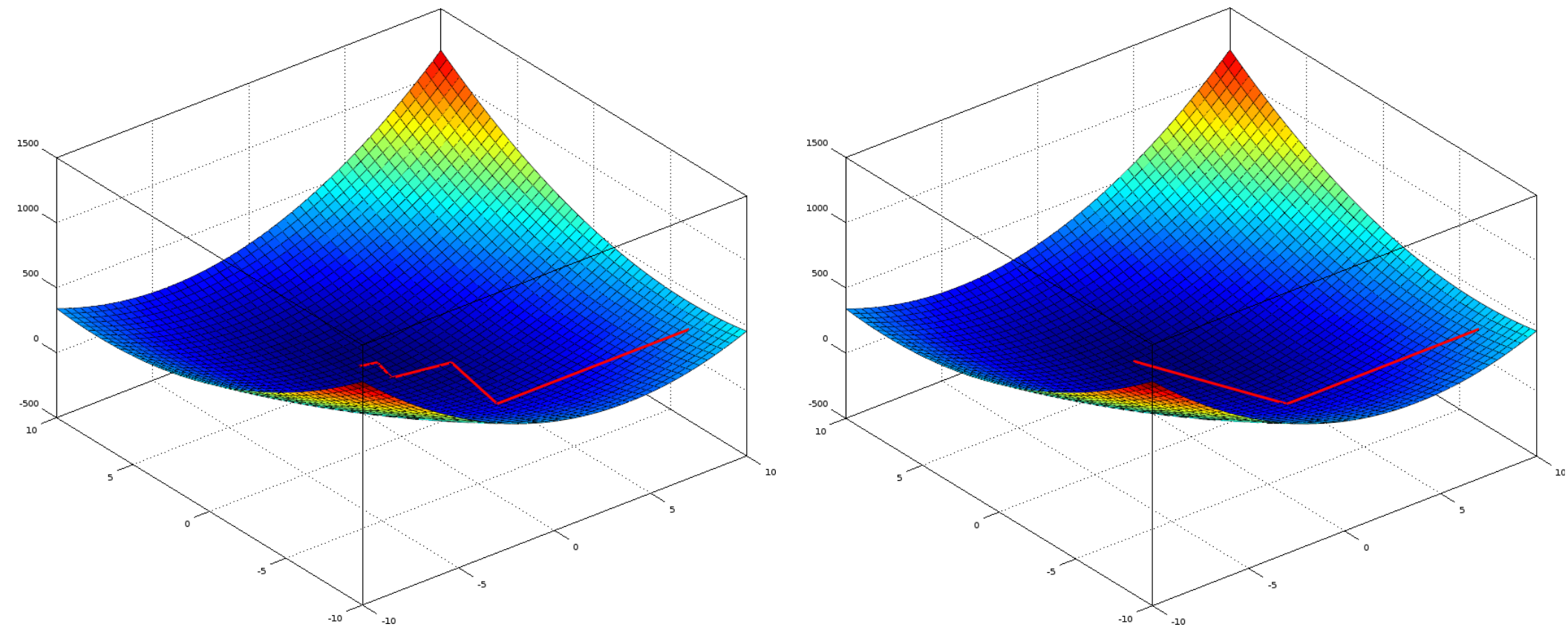
Método de gradiente conjugado

Es un método iterativo para minimizar funciones cuadráticas convexas $f:\mathbb{R}^n \to \mathbb{R}$ de la forma

$$f(\mathbf{x}) = \frac{1}{2}\mathbf{x}^{\mathrm{T}}\mathbf{A}\mathbf{x} - \mathbf{x}^{\mathrm{T}}\mathbf{b},$$

donde $\mathbf{x}, \mathbf{b} \in \mathbb{R}^n$ y $\mathbf{A} \in \mathbb{R}^{n \times n}$ es una matriz simétrica positiva definida.

Es un método de descenso de gradiente.



Ejemplo para $n=2$. Descenso de gradiente (izquierda), gradiente conjugado (derecha)

Para minimizar $f(\mathbf{x})$ calculamos primero su gradiente,

$$\nabla f(\mathbf{x}) = \nabla\left(\frac{1}{2}\mathbf{x}^{\mathrm{T}}\mathbf{A}\mathbf{x} - \mathbf{x}^{\mathrm{T}}\mathbf{b}\right) = \mathbf{A}\mathbf{x} - \mathbf{b},$$

buscamos minimizar, por tanto igualamos a cero

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

es decir, buscamos resolver un sistema lineal de ecuaciones.

El método de descenso de gradiente es:

Input: $\mathbf{A}$, $\mathbf{x}_0$, $\mathbf{b}$, ε
$k \leftarrow 0$
repetir
$\mathbf{r}_k \leftarrow \mathbf{b} - \mathbf{A}\mathbf{x}_k$
$\alpha_k \leftarrow \dfrac{\mathbf{r}_k^{\mathrm{T}}\mathbf{r}_k}{\mathbf{r}_k^{\mathrm{T}}\mathbf{A}\mathbf{r}_k}$
$\mathbf{x}_{k+1} \leftarrow \mathbf{x}_k + \alpha_k \mathbf{r}_k$
$k \leftarrow k+1$
hasta que $\|\mathbf{r}_k\| < \varepsilon$

$\mathbf{x}_0$ es una coordenada inicial.

Si $\mathbf{A}$ es simétrica positiva definida, entonces

$$\mathbf{x}^T \mathbf{A}\, \mathbf{x} \; > \; 0, \text{ para todo } \mathbf{x} \neq \mathbf{0}.$$

Así, podemos a definir un producto interno

$$\langle \mathbf{x}, \mathbf{y} \rangle_{\mathbf{A}} \; = \; \mathbf{x}^T \mathbf{A}\, \mathbf{y}.$$

Decimos que un vector $\mathbf{x}$ es **conjugado** a otro vector $\mathbf{y}$ con respecto a una matriz $\mathbf{A}$ si

$$\langle \mathbf{x}, \mathbf{y} \rangle_{\mathbf{A}} = 0, \text{ con } \boldsymbol{x} \neq \boldsymbol{y}.$$

La idea del algoritmo es utilizar direcciones conjugadas para el descenso en la búsqueda del punto óptimo $\mathbf{x}^*$, es decir

$$\mathbf{x}^* = \alpha_1 \mathbf{p}_1 + \alpha_2 \mathbf{p}_2 + \ldots + \alpha_n \mathbf{p}_n,$$

los coeficientes están dados a partir de la combinación lineal

$$\mathbf{A}\, \mathbf{x}^* = \alpha_1 \mathbf{A}\, \mathbf{p}_1 + \alpha_2 \mathbf{A}\, \mathbf{p}_2 + \ldots + \alpha_n \mathbf{A}\, \mathbf{p}_n = \mathbf{b}.$$

A partir de una matriz $\mathbf{A}$ de rango n sólo se pueden definir n vectores $\mathbf{A}$-conjugados, por lo tanto el algoritmo de gradiente conjugado garantiza la obtención de una solución en un máximo de n iteraciones.

Definamos el residual $\mathbf{r}_k$ como

$$\mathbf{r}_k = \mathbf{A}\,\mathbf{x}_k - \mathbf{b},\ k = 1, 2, \ldots$$

la idea es lograr de forma iterativa que el residual tienda a ser cero, buscando cada vez mejores $\mathbf{x}_k$.

El procedimiento será de forma iterativa, la forma de actualización será

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha\,\mathbf{p}_k,$$

tomando $\mathbf{p}_k$ como una dirección de descenso.

El tamaño de paso α_k que minimiza la función $f(\mathbf{x})$ a lo largo de la dirección $\mathbf{x}_k + \alpha_k\,\mathbf{p}_k$ es

$$\alpha_k = -\frac{\mathbf{r}_k^{\mathrm{T}}\,\mathbf{p}_k}{\mathbf{p}_k^{\mathrm{T}}\,\mathbf{A}\,\mathbf{p}_k}.$$

Si definimos $\mathbf{p}_{k+1}$ como la dirección más cercana al residual $\mathbf{r}_k$ bajo la restricción de ser conjugado.

Esta dirección está dada por la proyección de $\mathbf{r}_k$ en el espacio ortogonal a $\mathbf{p}_k$ con respecto al producto interno inducido por $\mathbf{A}$, así

$$\mathbf{p}_{k+1} = -\mathbf{r}_k + \underbrace{\frac{\mathbf{p}_k^{\mathrm{T}} \mathbf{A}\, \mathbf{r}_k}{\mathbf{p}_k^{\mathrm{T}} \mathbf{A}\, \mathbf{p}_k}}_{\beta_k} \mathbf{p}_k.$$

El algoritmo es el siguiente [Noce06 p108]:

Input: $\mathbf{A}$, $\mathbf{x}_0$, $\mathbf{b}$, ε

$\mathbf{r}_0 \leftarrow \mathbf{A}\mathbf{x}_0 - \mathbf{b}$

$\mathbf{p}_0 \leftarrow -\mathbf{r}_0$

$k \leftarrow 0$

mientras $\|\mathbf{r}_k\| > \varepsilon$

$$\alpha_k \leftarrow -\frac{\mathbf{r}_k^{\mathrm{T}}\mathbf{p}_k}{\mathbf{p}_k^{\mathrm{T}}\mathbf{A}\mathbf{p}_k}$$

$\mathbf{x}_{k+1} \leftarrow \mathbf{x}_k + \alpha_k \mathbf{p}_k$

$\mathbf{r}_{k+1} \leftarrow \mathbf{A}\mathbf{x}_{k+1} - \mathbf{b}$

$$\beta_{k+1} \leftarrow \frac{\mathbf{r}_{k+1}^{\mathrm{T}}\mathbf{A}\mathbf{p}_k}{\mathbf{p}_k^{\mathrm{T}}\mathbf{A}\mathbf{p}_k}$$

$\mathbf{p}_{k+1} \leftarrow -\mathbf{r}_{k+1} + \beta_{k+1}\mathbf{p}_k$

$k \leftarrow k+1$

$\mathbf{x}_0$ es una coordenada inicial (puede ser igual a $\mathbf{0}$).

Generalmente no es necesario realizar las n iteraciones, se puede definir la precisión deseada limitando la convergencia con una tolerancia ε.

El algoritmo de gradiente conjugado mejorado es el siguiente [Noce06 p112], con la variación de utilizar sólo una multiplicación matriz-vector:

Input: $\mathbf{A}$, $\mathbf{x}_0$, $\mathbf{b}$, ε

$\mathbf{r}_0 \leftarrow \mathbf{A}\mathbf{x}_0 - \mathbf{b}$

$\mathbf{p}_0 \leftarrow -\mathbf{r}_0$

$k \leftarrow 0$

mientras $\|\mathbf{r}_k\| > \varepsilon$

$\mathbf{w} \leftarrow \mathbf{A}\mathbf{p}_k$

$\alpha_k \leftarrow \dfrac{\mathbf{r}_k^{\mathrm{T}}\mathbf{r}_k}{\mathbf{p}_k^{\mathrm{T}}\mathbf{w}}$

$\mathbf{x}_{k+1} \leftarrow \mathbf{x}_k + \alpha_k \mathbf{p}_k$

$\mathbf{r}_{k+1} \leftarrow \mathbf{r}_k + \alpha \mathbf{w}$

$\beta_{k+1} \leftarrow \dfrac{\mathbf{r}_{k+1}^{\mathrm{T}}\mathbf{r}_{k+1}}{\mathbf{r}_k^{\mathrm{T}}\mathbf{r}_k}$

$\mathbf{p}_{k+1} \leftarrow -\mathbf{r}_{k+1} + \beta_{k+1}\mathbf{p}_k$

$k \leftarrow k+1$

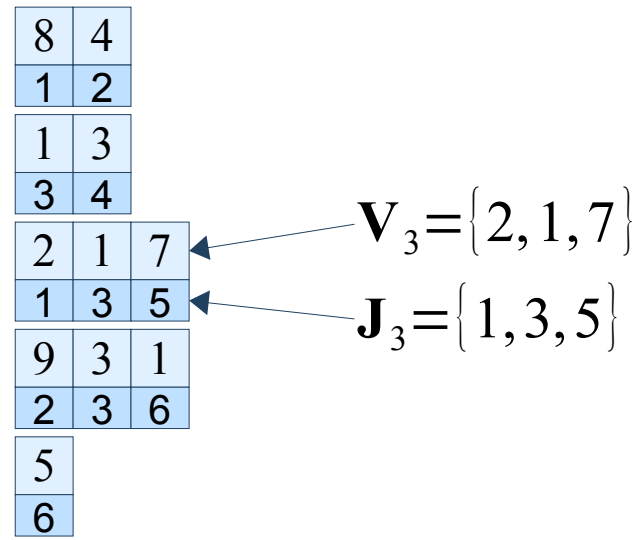
El proceso más lento del algoritmo es la multiplicación matriz-vector.

Se puede implementar eficientemente usando almacenamiento con compresión por renglones.

Compressed Row Storage

El método *Compressed Row Storage* (compresión por renglones) [Saad03 p362], se guardan las entradas no cero de cada renglón de **A** por separado.

$$\begin{pmatrix} 8 & 4 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 3 & 0 & 0 \\ 2 & 0 & 1 & 0 & 7 & 0 \\ 0 & 9 & 3 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 5 \end{pmatrix}$$



Con este método, por cada renglón de la matriz se guardan dos arreglos para las entradas distintas de cero de cada renglón. Un vector $\mathbf{J}_i$ conteniendo los índices y otro $\mathbf{V}_i$ los valores, con $i = 1, \ldots, m$.

Este esquema es adecuado cuando todos (o casi todos) los renglones tienen al menos una entrada distinta de cero.

Este tipo de esquema se puede crear utilizando un vector de vectores.

Multiplicación matriz vector

En la multiplicación matriz-vector $\mathbf{c} = \mathbf{A}\,\mathbf{b}$ el órden de búsqueda es $\mathrm{O}(1)$, esto es porque no se hace una búsqueda de las entradas del rengón, se toman las entradas una tras otra.

Sea $\mathbf{J}_i$ el conjunto de índices de las entradas no cero del renglón i de $\mathbf{A}$.

$|\mathbf{J}_i|$ es el número de entradas no cero del renglón i de $\mathbf{A}$. Nótese que $|\mathbf{V}_i| = |\mathbf{J}_i|$.

$$\begin{pmatrix} c_1 \\ c_2 \\ c_3 \\ c_4 \\ c_5 \\ c_6 \end{pmatrix} = \begin{pmatrix} 8 & 4 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 3 & 0 & 0 \\ 2 & 0 & 1 & 0 & 7 & 0 \\ 0 & 9 & 3 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 5 \end{pmatrix} \begin{pmatrix} 5 \\ 4 \\ 0 \\ 1 \\ 2 \\ 9 \end{pmatrix}$$

$$\begin{pmatrix} c_1 \\ c_2 \\ c_3 \\ c_4 \\ c_5 \\ c_6 \end{pmatrix} = \begin{matrix} 8 & 4 \\ 1 & 2 \end{matrix} \quad \begin{matrix} 1 & 3 \\ 3 & 4 \end{matrix} \quad \begin{matrix} 2 & 1 & 7 \\ 1 & 3 & 5 \end{matrix} \quad \begin{matrix} 9 & 3 & 1 \\ 2 & 3 & 6 \end{matrix} \quad \begin{matrix} 5 \\ 6 \end{matrix} \begin{pmatrix} 5 \\ 4 \\ 0 \\ 1 \\ 2 \\ 9 \end{pmatrix}$$

$$c_i = \sum_{i=1}^{n} a_{ij} b_j$$

$$c_i = \sum_{k=1}^{|\mathbf{J}_i|} \mathbf{V}_i^k b_{\mathbf{J}_i^k}$$

La ventaja de utilizar compresión por renglones es que los datos de cada renglón de la matriz de rigidez son accesados en secuencia uno tras otro, esto producirá una ventaja de acceso al entrar el bloque de memoria de cada renglón en el *cache* del CPU.

El pseudocódigo es:

ConjugateGradient $\left(\mathbf{A}, \mathbf{x}, \mathbf{b}, \varepsilon, step_{max}\right)$

$\mathbf{A} = \left(\mathbf{V}_i, \mathbf{J}_i\right)$, $i = 1, 2, \ldots, n$ CRS matrix
$\mathbf{x} \in \mathbb{R}^n$ Initial value
$\mathbf{b} \in \mathbb{R}^n$ Right side vector
$\varepsilon \in \mathbb{R}$ Tolerance
$step_{max}$ Maximum number of steps

$\mathbf{r} \in \mathbb{R}^n$ Residual
$\mathbf{p} \in \mathbb{R}^n$ Descent direction
$\mathbf{w} \in \mathbb{R}^n$ Result of matrix*vector

$dot\_rr \leftarrow 0$
for $i \leftarrow 1, 2, \ldots, n$
· $sum \leftarrow 0$
· for $k \leftarrow 1, 2, \ldots, \left|\mathbf{V}_i\right|$
· · $j \leftarrow \mathbf{J}_i^k$
· · $sum \leftarrow sum + \mathbf{V}_i^k x_j$
· $r_i \leftarrow sum - b_i$
· $p_i \leftarrow -r_i$
· $dot\_rr \leftarrow dot\_rr + r_i r_i$

$step \leftarrow 0$
while $\left(step < step_{max}\right)$ and $\left(\sqrt{dot\_rr} > \varepsilon\right)$
· $dot\_pw \leftarrow 0$
· for $i \leftarrow 1, 2, \ldots, n$
· · $sum \leftarrow 0$
· · for $k \leftarrow 1, 2, \ldots, \left|\mathbf{V}_i\right|$
· · · $j \leftarrow \mathbf{J}_i^k$
· · · $sum \leftarrow sum + \mathbf{V}_i^k p_j$
· · $w_i \leftarrow sum$
· · $dot\_pw \leftarrow dot\_pw + p_i w_i$

· $\alpha \leftarrow \dfrac{dot\_rr}{dot\_pw}$

· $new\_dot\_rr \leftarrow 0$
· for $i \leftarrow 1, 2, \ldots, n$
· · $x_i \leftarrow x_i + \alpha p_i$
· · $r_i \leftarrow r_i + \alpha w_i$
· · $new\_dot\_rr \leftarrow new\_dot\_rr + r_i r_i$

· $\beta \leftarrow \dfrac{new\_dot\_rr}{dot\_rr}$

· for $i \leftarrow 1, 2, \ldots, n$
· · $p_i \leftarrow \beta p_i - r_i$

· $dot\_rr \leftarrow new\_dot\_rr$
· $step \leftarrow step + 1$

Número de condición

El siguiente sistema de ecuaciones [Kind07], $\mathbf{A}\,\mathbf{x} = \mathbf{b}_1$

$$\underbrace{\begin{pmatrix} 0.1 & 3.0 & 3.0 & 4.0 \\ 0.4 & 12.2 & 20.1 & 26.1 \\ 0.1 & 3.1 & 7.0 & 9.0 \\ 0.2 & 6.1 & 10.1 & 13.0 \end{pmatrix}}_{\mathbf{A}} \underbrace{\begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix}}_{\mathbf{x}} = \underbrace{\begin{pmatrix} 10.1 \\ 58.8 \\ 19.2 \\ 29.4 \end{pmatrix}}_{\mathbf{b}_1}, \text{ tiene solución } \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} 1.0 \\ 1.0 \\ 1.0 \\ 1.0 \end{pmatrix},$$

nótese que $\|\mathbf{b}_1\|_2 = 69.227523428.$

Si perturbamos el vector de la derecha un poco

$$\underbrace{\begin{pmatrix} 0.1 & 3.0 & 3.0 & 4.0 \\ 0.4 & 12.2 & 20.1 & 26.1 \\ 0.1 & 3.1 & 7.0 & 9.0 \\ 0.2 & 6.1 & 10.1 & 13.0 \end{pmatrix}}_{\mathbf{A}} \underbrace{\begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix}}_{\mathbf{x}} = \underbrace{\begin{pmatrix} 10.1 - 0.005 \\ 58.8 + 0.005 \\ 19.2 - 0.005 \\ 29.4 - 0.005 \end{pmatrix}}_{\mathbf{b}_1 + \boldsymbol{\Delta}} = \underbrace{\begin{pmatrix} 10.095 \\ 58.805 \\ 19.195 \\ 29.395 \end{pmatrix}}_{\mathbf{b}_2}, \text{ entonces } \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} 351.45 \\ -11.00 \\ 1.05 \\ 1.20 \end{pmatrix},$$

ahora $\|\mathbf{b}_2\|_2 = 69.227531373.$

Si en vez de perturbar el vector de la derecha, perturbamos la matriz

$$\begin{pmatrix} 0.100 & 3+0.005 & 3.000 & 4.000 \\ 0.400 & 12.200 & 20.100 & 26.100 \\ 0.100 & 3.100 & 7-0.005 & 9.000 \\ 0.200 & 6.100 & 10.100 & 13-0.005 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} 10.1 \\ 58.8 \\ 19.2 \\ 29.4 \end{pmatrix}, \text{ entonces } \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} -3.425 \\ 1.150 \\ 1.053 \\ 0.957 \end{pmatrix}.$$

Un sistema de ecuaciones $\mathbf{A}\,\mathbf{x} = \mathbf{b}$ es considerado

bien condicionado si un pequeño cambio en los valores de $\mathbf{A}$
o un pequeño cambio en $\mathbf{b}$ resulta en un pequeño cambio en $\mathbf{x}$.

Un sistema de ecuaciones $\mathbf{A}\,\mathbf{x} = \mathbf{b}$ es considerado

mal condicionado si un pequeño cambio en los valores de $\mathbf{A}$
o un pequeño cambio en $\mathbf{b}$ resulta en un cambio grande en $\mathbf{x}$.

El **número de condición** indica que tan sensible es una función a pequeños cambios en la entrada.

El número de condición κ de una matriz $\mathbf{A}$ no singular, para una norma $\|\cdot\|$ está dado por

$$\kappa(\mathbf{A}) = \|\mathbf{A}\| \cdot \|\mathbf{A}^{-1}\|.$$

Para la norma $\|\cdot\|_2$,

$$\kappa(\mathbf{A}) = \|\mathbf{A}\| \cdot \|\mathbf{A}^{-1}\| = \frac{\sigma_{max}(\mathbf{A})}{\sigma_{min}(\mathbf{A})},$$

donde σ son los valores singulares de la matriz.

Para una matriz $\mathbf{A}$ simétrica positiva definida,

$$\kappa(\mathbf{A}) = \frac{\lambda_{max}(\mathbf{A})}{\lambda_{min}(\mathbf{A})},$$

donde λ son los eigenvalores de $\mathbf{A}$.

Así, matrices con un número de condición cercano a 1 se dicen que están bien condicionadas.

Al reducir el número de condición de un sistema, se puede acelerar la velocidad de convergencia del gradiente conjugado.

Gradiente conjugado precondicionado

Entonces, en vez de resolver el problema

$$\mathbf{A}\,\mathbf{x}-\mathbf{b}=0,$$

se resuelve el problema

$$\mathbf{M}^{-1}\left(\mathbf{A}\,\mathbf{x}-\mathbf{b}\right)=0,$$

con $\mathbf{M}^{-1}$ una matriz cuadrada, la cual recibe el nombre de **precondicionador**.

El mejor precondicionador sería claro $\mathbf{M}^{-1}=\mathbf{A}^{-1}$, así $\mathbf{x}=\mathbf{M}^{-1}\mathbf{b}$, y el gradiente conjugado convergiría en un paso.

Al igual que la matriz $\mathbf{A}$, el precondicionador $\mathbf{M}^{-1}$ tiene que ser simétrico positivo definido.

Hay dos tipos de precondicionadores, implícitos $\mathbf{M}$ y explícitos $\mathbf{M}^{-1}$.

En algunos casos es costoso calcular $\mathbf{M}^{-1}$, en general se utilizan precondicionadores con inversas fáciles de calcular o precondicionadores implicitos que se puedan factorizar (con Cholesky, por ejemplo $\mathbf{M}=\mathbf{L}\,\mathbf{L}^{\mathrm{T}}$).

El algoritmo es el siguiente:

Input: $\mathbf{A}$, $\mathbf{x}_0$, $\mathbf{b}$,ε

$\mathbf{r}_0 \leftarrow \mathbf{A}\mathbf{x}_0 - \mathbf{b}$

$\mathbf{q}_0 \leftarrow \mathbf{M}^{-1}\mathbf{r}_0$

$\mathbf{p}_0 \leftarrow -\mathbf{q}_0$

$k \leftarrow 0$

mientras $\left\|\mathbf{r}_k\right\| > \varepsilon$

$\mathbf{w} \leftarrow \mathbf{A}\mathbf{p}_k$

$\alpha_k \leftarrow \dfrac{\mathbf{r}_k^{\mathrm{T}}\mathbf{q}_k}{\mathbf{p}_k^{\mathrm{T}}\mathbf{w}}$

$\mathbf{x}_{k+1} \leftarrow \mathbf{x}_k + \alpha_k \mathbf{p}_k$

$\mathbf{r}_{k+1} \leftarrow \mathbf{r}_k + \alpha \mathbf{w}$

$\mathbf{q}_{k+1} \leftarrow \mathbf{M}^{-1}\mathbf{r}_{k+1}$, or solve $\mathbf{M}\mathbf{q}_{k+1} \leftarrow \mathbf{r}_{k+1}$

$\beta_k \leftarrow \dfrac{\mathbf{r}_{k+1}^{\mathrm{T}}\mathbf{q}_{k+1}}{\mathbf{r}_k^{\mathrm{T}}\mathbf{q}_k}$

$\mathbf{p}_{k+1} \leftarrow -\mathbf{q}_{k+1} + \beta_{k+1}\mathbf{p}_k$

$k \leftarrow k+1$

Nótese que ahora el algoritmo require aplicar el precondicionador en cada paso.

Gradiente conjugado + precondicionador Jacobi

El precondicionador Jacobi es el más sencillo, consiste en hacer

$$\mathbf{M} = \operatorname{diag}\left(\mathbf{A}\right),$$

de esta forma el precondicionador es una matriz diagonal, cuya inversa es fácil de calcular

$$\left(\mathbf{M}^{-1}\right)_{i\,j} = \begin{cases} \dfrac{1}{\mathbf{A}_{i\,i}} & \text{si } i = j \\ 0 & \text{si } i \neq j \end{cases}.$$

No se almacena todo $\boldsymbol{M}^{-1}$, lo usual es guardar un vector con sólo los elementos de la diagonal.

El pseudocódigo es...

ConjugateGradientJacobi$\left(\mathbf{A}, \mathbf{x}, \mathbf{b}, \varepsilon, step_{max}\right)$

$\mathbf{A}=\left(\mathbf{V}_i, \mathbf{J}_i\right)$, $i=1,2,\ldots,n$ CRS matrix

$\mathbf{x}\in\mathbb{R}^n$ Initial value

$\mathbf{b}\in\mathbb{R}^n$ Right side vector

$\varepsilon\in\mathbb{R}$ Tolerance

$step_{max}$ Maximum number of steps

$\mathbf{r}\in\mathbb{R}^n$ Residual

$\mathbf{p}\in\mathbb{R}^n$ Descent direction

$\mathbf{M}\in\mathbb{R}^n$ Preconditioner stored as vector

$\mathbf{q}\in\mathbb{R}^n$ Applied preconditioner

$\mathbf{w}\in\mathbb{R}^n$ Result of matrix*vector

$dot\_rr \leftarrow 0$
$dot\_rq \leftarrow 0$
for $i \leftarrow 1,2,\ldots,n$
· $sum \leftarrow 0$
· for $k \leftarrow 1,2,\ldots,\left|\mathbf{V}_i\right|$
· · $j \leftarrow \mathbf{J}_i^k$
· · $sum \leftarrow sum + \mathbf{V}_i^k x_j$
· $r_i \leftarrow sum - b_i$
· $M_i \leftarrow A_{ii}$
· $q_i \leftarrow r_i / M_i$
· $p_i \leftarrow -q_i$
· $dot\_rr \leftarrow dot\_rr + r_i r_i$
· $dot\_rq \leftarrow dot\_rq + r_i q_i$

$step \leftarrow 0$
while $\left(step < step_{max}\right)$ and $\left(\sqrt{dot\_rr} > \varepsilon\right)$
· $dot\_pw \leftarrow 0$
· for $i \leftarrow 1,2,\ldots,n$
· · $sum \leftarrow 0$
· · for $k \leftarrow 1,2,\ldots,\left|\mathbf{V}_i\right|$
· · · $j \leftarrow \mathbf{J}_i^k$
· · · $sum \leftarrow sum + \mathbf{V}_i^k p_j$
· · $w_i \leftarrow sum$
· · $dot\_pw \leftarrow dot\_pw + p_i w_i$
· $\alpha \leftarrow \dfrac{dot\_rq}{dot\_pw}$
· $new\_dot\_rr \leftarrow 0$
· $new\_dot\_rq \leftarrow 0$
· for $i \leftarrow 1,2,\ldots,n$
· · $x_i \leftarrow x_i + \alpha\, p_i$
· · $r_i \leftarrow r_i + \alpha\, w_i$
· · $q_i \leftarrow r_i / M_i$
· · $new\_dot\_rr \leftarrow new\_dot\_rr + r_i r_i$
· · $new\_dot\_rq \leftarrow new\_dot\_rq + r_i q_i$
· $\beta \leftarrow \dfrac{new\_dot\_rq}{dot\_rq}$
· for $i \leftarrow 1,2,\ldots,n$
· · $p_i \leftarrow \beta\, p_i - q_i$
· $dot\_rr \leftarrow new\_dot\_rr$
· $dot\_rq \leftarrow new\_dot\_rq$
· $step \leftarrow step + 1$

¿Por qué dan distinto resultado?

Es la misma suma, pero...

```
// sum1.cpp

float a[] = {1.0, 1.0e-8, 2.0e-8,
             3.0e-8, 4.0e-8, 5.0e-8,
             1.0e-8, 2.0e-8, 3.0e-8,
             4.0e-8, 5.0e-8};

int main()
{
    float sum = 0;
    for (int i = 0; i < 11; ++i)
        sum += a[i];

    printf("Sum = %1.9f\n", sum);

    return 0;
}
```

```
Sum = 1.000000000
```

```
// sum2.cpp

float a[] = {1.0e-8, 2.0e-8, 3.0e-8,
             4.0e-8, 5.0e-8, 1.0e-8,
             2.0e-8, 3.0e-8, 4.0e-8,
             5.0e-8, 1.0};

int main()
{
    float sum = 0;
    for (int i = 0; i < 11; ++i)
        sum += a[i];

    printf("Sum = %1.9f\n", sum);

    return 0;
}
```

```
Sum = 1.000000358
```

Las solución exacta es 1.000000300.

Al calcular los productos punto para el gradiente conjugado se tienen que sumar muchos números con punto flotante.

Hay tres soluciónes para evitar pérdida de información.

1. Ordenar los números antes de sumarlos

```cpp
// sum1.cpp
#include <stdio.h>

float a[] = {1.0, 1.0e-8, 2.0e-8,
             3.0e-8, 4.0e-8, 5.0e-8,
             1.0e-8, 2.0e-8, 3.0e-8,
             4.0e-8, 5.0e-8};

int main()
{
    float sum = 0;
    for (int i = 0; i < 11; ++i)
        sum += a[i];

    printf("Sum = %1.9f\n", sum);

    return 0;
}
```

```
Sum = 1.000000000
```

```cpp
// sum3.cpp
#include <stdio.h>

float a[] = {1.0e-8, 1.0e-8, 2.0e-8,
             2.0e-8, 3.0e-8, 3.0e-8,
             4.0e-8, 4.0e-8, 5.0e-8,
             5.0e-8, 1.0};

int main()
{
    float sum = 0;
    for (int i = 0; i < 11; ++i)
        sum += a[i];

    printf("Sum = %1.9f\n", sum);

    return 0;
}
```

```
Sum = 1.000000358
```

2. Usar tipos de punto flotante más grandes para acumular la suma

```cpp
// sum1.cpp
#include <stdio.h>

float a[] = {1.0, 1.0e-8, 2.0e-8,
             3.0e-8, 4.0e-8, 5.0e-8,
             1.0e-8, 2.0e-8, 3.0e-8,
             4.0e-8, 5.0e-8};

int main()
{
    float sum = 0;
    for (int i = 0; i < 11; ++i)
    {
        sum += a[i];
    }

    printf("Sum = %1.9f\n", sum);

    return 0;
}
```

```
Sum = 1.000000000
```

```cpp
// sum4.cpp
#include <stdio.h>

float a[] = {1.0, 1.0e-8, 2.0e-8,
             3.0e-8, 4.0e-8, 5.0e-8,
             1.0e-8, 2.0e-8, 3.0e-8,
             4.0e-8, 5.0e-8};

int main()
{
    double sum = 0;
    for (int i = 0; i < 11; ++i)
    {
        sum += a[i];
    }

    printf("Sum = %1.9f\n", sum);

    return 0;
}
```

```
Sum = 1.000000300
```

Cuando se suman “doubles”, se puede acumular el resultado utilizando “long double”.

¿Qué pasa cuando no hay puntos flotantes de mayor capacidad?

3. El algoritmo de sumación de Kahan

Este algoritmo [Gold91] evita perder la contribución de los números pequeños.

```
// sum4.cpp
float a[] = {1.0, 1.0e-8, 2.0e-8,
             3.0e-8, 4.0e-8, 5.0e-8,
             1.0e-8, 2.0e-8, 3.0e-8,
             4.0e-8, 5.0e-8};

int main()
{
    float sum = 0;
    float c = 0;
    for (int i = 0; i < 11; ++i)
    {
        float y = a[i] + c;
        float t = sum + y;
        c = y - (t - sum);
        sum = t;
    }

    printf("Sum = %1.9f\n", sum);
    return 0;
}
```

c guardará la compensación para los valores pequeños

suma la compensación al valor leido
si sum es grande, los valores de y se pierden
(t - sum) es la parte grande de y, y - (t - sum) es la chica

```
1.000000358
```

http://en.wikipedia.org/wiki/Kahan_summation_algorithm

¿Preguntas?

miguelvargas@cimat.mx

Referencias

[Noce06] J. Nocedal, S. J. Wright. Numerical Optimization. Springer, 2006.

[Piss84] S. Pissanetzky. Sparse Matrix Technology. Academic Press, 1984.

[Saad03] Y. Saad. Iterative Methods for Sparse Linear Systems. SIAM, 2003.

[Kind07] U. Kindelán. Resolución de sistemas lineales de ecuaciones: Método del gradiente conjugado. Universidad Politécnica de Madrid. 2007

[Gold91] D. Goldberg. What Every Computer Scientist Should Know About Floating-Point Arithmetic. Computing Surveys. Association for Computing Machinery. 1991.