BENEMÉRITA UNIVERSIDAD AUTÓNOMA DE PUEBLA

FACULTAD DE CIENCIAS FÍSICO MATEMÁTICAS

ESTUDIO DEL MÉTODO DE GRADIENTE CONJUGADO

TESIS

PARA OBTENER EL GRADO DE:

LICENCIATURA EN MATEMÁTICAS APLICADAS

PRESENTA:

JUAN ANTONIO VÁZQUEZ MORALES

DIRECTOR DE TESIS:

DR. GUILLERMO LÓPEZ MAYO

PUEBLA, PUEBLA. 2018

A mi familia, compañeros de aula y profesores

Agradecimientos

Agradezco a mis padres, Isela y Constantino, y a mi abuelita Concepción por su apoyo moral y económico para alcanzar este objetivo. A mis hermanas Rocio y Lucero por su paciencia y comprensión durante estos años.

A mis compañeros y amigos de la facultad con los que trabajé de manera amena en diversas ocasiones, en especial a Edgar, Gustavo, Julio y Roque. A mis amigas América, Lizbeth, Silvia y Solehyr por su compañía, ayuda, comprensión y regaños durante mis raras ideas.

A mi asesor de tesis, el Dr. Guillermo López Mayo, por el cual tengo un gran respeto, gracias por su apoyo.

A mi tutor académico, Dra. Hortensia Josefina Reyes Cervantes, por su ayuda y guía a lo largo de mi formación.

Al Dr. Iván Hernández Orzuna, por sus asesorías y su ayuda a una mejor visión de las matemáticas. Al M.C. Sergio Adán Juárez, por su ayuda en la mejora del código.

A mis sinodales, Dra. María de Lourdes Sandoval Solís, Dr. Jorge Bustamante González y Dr. Miguel Antonio Jiménez Pozo por haber aceptado revisar este trabajo y aportar su conocimiento para la mejora del mismo.

Si te sientes cansado y tus piernas tiemblan sin cesar,
avanza un solo paso y no te dejes vencer.

Índice general

Capítulo 1

Introducción

Entre los métodos iterativos para la resolución de sistemas de ecuaciones lineales y no lineales, así como en la resolución de problemas de optimización sin restricciones, no es muy práctico la utilización de métodos directos u otros métodos como la descomposición de Cholesky (véase [2]), ya que estos pueden ser tardados. En tales casos un método iterativo que puede ser de utilidad es el de Gradientes Conjugados (véase [4]), desarrollado por Magnus Hestenes y Eduard Stiefel en la década de los 50.

En la resolución de sistemas de ecuaciones lineales de dimensiones altas, los métodos usuales como el de Gauss-Jordan, el número de operaciones es muy grande, además de que se necesita ir modificando los datos de entrada, lo que representa un gasto computacional muy alto. Esta problemática se supera mediante el método de gradiente conjugado, del cual conserva intactos los datos del sistema a resolver, además de que en la aritmética exacta, se asegura que se obtiene la solución en un número finito de iteraciones, incluso, después de cada iteración, el punto obtenido se acerca más a la solución.

El método de gradiente conjugado, a pesar de su importancia y utilidad, en la facultad no es un tema abordado en los cursos. En este trabajo, se hace una descripción de este método, los tipos de problemas que resuelve, su funcionamiento como un método iterativo directo, sus propiedades de convergencia y se agrega una modificación del método que usualmente no destaca en la literatura.

El objetivo del trabajo, además del desarrollo de la teoría, consiste en explicar el por qué la propuesta usual no es muy adecuada para su implementación computacional, y el por qué la modificación del algoritmo propuesto se puede usar en diversos softwares o incluso ser programado en cualquier

lenguaje de programación. Por consiguiente, se presenta una propuesta en el lenguaje C, que estará documentada, a manera de que cualquier persona pueda probar, o incluso modificar las líneas ya escritas.

El progreso del trabajo consistió, en primer lugar, de la consulta de literatura para el desarrollo de la teoría. Después, escribir las definiciones y resultados más significativos. Las pruebas de los resultados se escribirán con los detalles que se requieren para su fácil comprensión. Por último, se expondrá la modificación del algoritmo y su implementación computacional. El avance del escrito se hará con el apoyo del Dr. Guillermo López Mayo.

Con esta investigación se desea tener una mejor comprensión del método de gradiente conjugado, poner en practica las habilidades y conocimientos que se desarrollaron durante la formación académica, y ver si uno es capaz de difundir y explicar la información recopilada.

Capítulo 2

Métodos de Gradiente Conjugado

En este capítulo, se hace la descripción del Método de Gradiente Conjugado, el tipo de problemas que resuelve, su funcionamiento como método iterativo directo y sus propiedades de convergencia. Todas las afirmaciones sobre el método estarán demostradas, y solo se requiere de un conocimiento de álgebra, calculo diferencial y análisis convexo. Sin más que decir, se procede a realizar el análisis del método.

El Método de Gradiente Conjugado, es un método para resolver sistemas de ecuaciones lineales

$$Ax = b, \tag{2.1}$$

donde A es una matriz de tamaño $n \times n$ simétrica y definida positiva. Por otro lado, el problema (2.1) es equivalente al problema de minimizar:

$$\phi(x) := \frac{1}{2}x^T Ax - b^T x, \tag{2.2}$$

pues el problema anterior se reduce ha encontrar el punto x^* donde el gradiente de ϕ se anule, ya que se trata de un problema convexo. De está observación, se deduce que los problemas (2.1) y (2.2) tienen la misma solución.

Para el análisis de este método, usaremos la definición siguiente.

Definición 2.1. *El residuo del sistema lineal o el gradiente de ϕ, se define como*

$$r(x) := \nabla\phi = Ax - b, \tag{2.3}$$

en particular, si $x = x_k$,

$$r_k := Ax_k - b. \tag{2.4}$$

2.1. Método de Direcciones Conjugadas

El Método de Gradiente Conjugado tiene la propiedad de generar un conjunto de vectores no nulos con una propiedad especial, la conjugación, explicada en la definición siguiente.

Definición 2.2. *Se dice que* $\{p_0, p_1, \ldots, p_l\}$ *es un* ***conjunto de direcciones*** *respecto a* A *si es conjugado respecto a* A *(véase la Definición A.10).*

Dado un punto (llamado punto inicial) $x_0 \in \mathbb{R}^n$ y un conjunto de direcciones conjugadas $\{p_0, p_1, \ldots, p_{n-1}\}$, el método de direcciones conjugadas genera la sucesión $\{x_k\}$ de la manera siguiente:

$$x_{k+1} = x_k + \alpha_k p_k,$$

donde α_k (llamado longitud de paso) es el mínimo de la función $\phi(\cdot)$ a lo largo de $x_k + \alpha p_k$, el cual, al buscar el único punto donde la derivada respecto a α se anula, se encuentra que

$$\alpha_k = -\frac{r_k^T p_k}{p_k^T A p_k}.$$

Dada la información anterior se formula el algoritmo siguiente.

Algoritmo 2.1 Método de Direcciones Conjugadas

Entrada: $x_0,\ b,\ A$ y $\{p_0, p_1, \ldots, p_{n-1}\}$ conjunto de direcciones conjugadas respecto a A.

Salida: x^*.

Sea $r_0 \leftarrow Ax_0 - b,\ k \leftarrow 0$;

mientras $r_k \neq 0$ **hacer**

$$\alpha_k \leftarrow -\frac{r_k^T p_k}{p_k^T A p_k}; \tag{2.5a}$$

$$x_{k+1} \leftarrow x_k + \alpha_k p_k; \tag{2.5b}$$

$$r_{k+1} \leftarrow Ax_{k+1} - b; \tag{2.5c}$$

$$k \leftarrow k + 1; \tag{2.5d}$$

fin mientras

Como se observa, el Algoritmo 2.1 tiene como criterio de paro $r_k = 0$ y que solamente se necesitan las direcciones de búsqueda $\{p_0, p_1, \ldots, p_{n-1}\}$. De ahí la importancia de que dichas direcciones deban cumplir la propiedad de conjugación, lo que permite llegar a la solución x^* en a lo más n iteraciones. A continuación se demuestra este hecho.

Teorema 2.1. *Para cualquier $x_0 \in \mathbb{R}^n$, el Algoritmo 2.1 converge a la solución x^* del sistema lineal (2.1) en a lo más n iteraciones.*

Demostración. Por la Proposición A.1, los vectores d_k generan todo el espacio $\mathbb{R}^n$, así se puede escribir la diferencia entre x^* y x_0 como sigue:

$$x^* - x_0 = \sigma_0 p_0 + \sigma_1 p_1 + \cdots + \sigma_{n-1} p_{n-1},$$

para algunos escalares σ_k. Premultiplicando la expresión anterior por $p_k^T A$ y usando (A.1) se obtiene

$$p_k^T A(x^* - x_0) = \sigma_k p_k^T A p_k,$$

despejando σ_k, se tiene que

$$\sigma_k = \frac{p_k^T A(x^* - x_0)}{p_k^T A p_k}. \tag{2.6}$$

Para demostrar el resultado, se mostrará que los coeficientes σ_k coinciden con las longitudes de paso α_k dadas por (2.5a).

Si x_k está dado por (2.5a) y (2.5b), entonces

$$x_k = x_0 + \alpha_0 p_0 + \alpha_1 x_1 + \cdots + \alpha_{k-1} p_{k-1}.$$

Premultiplicando por $p_k^T A$ y por la propiedad de conjugación,

$$p_k^T A x_k = p_k^T A x_0 \Rightarrow p_k^T A(x_k - x_0) = 0.$$

Así,

$$\begin{aligned} p_k^T A(x^* - x_0) &= p_k^T A(x^* - x_k + x_k - x_0) \\ &= p_k^T A(x^* - x_k) + p_k^T A(x_k - x_0) \\ &= p_k^T (b - A x_k) \\ &= -p_k^T r_k. \end{aligned}$$

Sustituyendo en (2.6),

$$\sigma_k = -\frac{p_k^T r_k}{p_k^T A p_k} = -\frac{r_k^T p_k}{p_k^T A p_k} = \alpha_k \quad \text{para } k = 0, 1, \ldots, n-1.$$

Por lo tanto,

$$x_n - x_0 = \sum_{k=1}^{n-1} \sigma_k p_k = \sum_{k=1}^{n-1} \alpha_k p_k = x^* - x_0,$$

es decir,

$$x_n = x^*.$$

□

Interpretación geométrica del Método de Direcciones Conjugadas

Si la matriz A es diagonal, las curvas de nivel de ϕ son elipses cuyos ejes están alineados con los ejes coordenados. Se puede encontrar el mínimo de la función ϕ realizando minimizaciones a lo largo de las direcciones coordenadas $e_1, e_2, \ldots, e_n$ sucesivamente, como se puede ver en la Imagen 2.1.

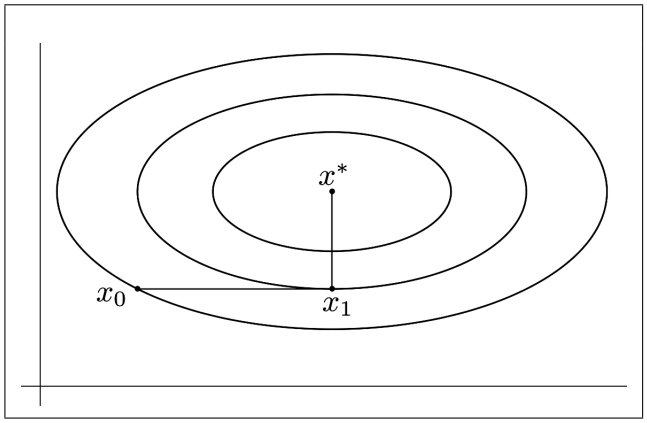

Imagen 2.1: Las minimizaciones sucesivas a lo largo de las direcciones coordenadas de una función cuadrática con Hessiano diagonal, encuentran la solución en a lo más n iteraciones.

Cuando A no es diagonal, las curvas de nivel son de tipo elíptico, pero generalmente las curvas de nivel no están alineadas con las direcciones

coordenadas. La estrategia de realizar minimizaciones sucesivas a lo largo de las direcciones coordenadas no es efectiva, ya que en general no se obtiene la solución óptima en n iteraciones, o incluso, en un número finito como se observa en la Imagen 2.2.

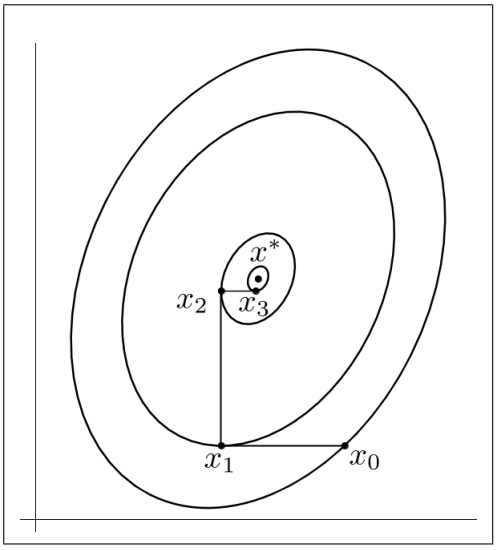

Imagen 2.2: Las minimizaciones sucesivas, a lo largo de los ejes coordenados, no llegan a la solución en n iteraciones para una función cuadrática convexa general.

Para recuperar el buen comportamiento de la Imagen 2.1, en la cual las curvas de nivel están alineados con los ejes coordenados, se transforma el problema, es decir, hacer a A diagonal, entonces la estrategia de realizar minimizaciones a lo largo de las direcciones coordenadas es efectiva. Para realizar lo anterior, se transforma el problema definiendo nuevas variables $\hat{x}$ como:

$$\hat{x} = S^{-1}x. \tag{2.7}$$

donde S es la matriz definida por

$$S = \begin{bmatrix} p_0 & p_1 & \cdots & p_{n-1} \end{bmatrix},$$

y $\{p_0, p_1, \ldots, p_{n-1}\}$ es el conjunto de direcciones conjugadas con respecto a A. Con este cambio la función cuadrática ϕ definida por (2.2) se convierte en

$$\hat{\phi}(\hat{x}) := \phi(S\hat{x}) = \frac{1}{2}\hat{x}^T(S^TAS)\hat{x} - (S^Tb)^T\hat{x}.$$

Por la propiedad de conjugación (A.1),

$$\begin{aligned} S^T AS &= \begin{bmatrix} p_0^T \\ p_1^T \\ \vdots \\ p_{n-1}^T \end{bmatrix} A \begin{bmatrix} p_0 & p_1 & \cdots & p_{n-1} \end{bmatrix} \\ &= \begin{bmatrix} p_0^T Ap_0 & p_0^T Ap_1 & \cdots & p_0^T Ap_{n-1} \\ p_1^T Ap_0 & p_1^T Ap_1 & \cdots & p_1^T Ap_{n-1} \\ \cdot & \cdot & \cdots & \cdot \\ p_{n-1}^T Ap_0 & p_{n-1}^T Ap_1 & \cdots & p_{n-1}^T Ap_{n-1} \end{bmatrix} \\ &= \begin{bmatrix} p_0^T Ap_0 & 0 & \cdots & 0 \\ 0 & p_1^T Ap_1 & \cdots & 0 \\ \cdot & \cdot & \cdots & \cdot \\ 0 & 0 & \cdots & p_{n-1}^T Ap_{n-1} \end{bmatrix}, \end{aligned}$$

es decir, la matriz $S^T AS$ es diagonal, por lo que se puede encontrar el mínimo valor de $\hat{\phi}$ realizando la misma estrategia de cuando A es diagonal. Sin embargo, debido a la relación (2.7), cada dirección coordenada en el $\hat{x}$-espacio corresponde a una dirección p_i en el x-espacio. Por lo tanto, esta estrategia aplicada a ϕ es equivalente al Algoritmo 2.1. Por el Teorema 2.1, se obtendrá la solución en a lo más n iteraciones.

Cuando la matriz A es diagonal, cada iteración determina una coordenada de la solución x^*. Esto dice que después de k iteraciones, la función ϕ encuentra su mínimo valor en la variedad lineal $x_0 + \text{gen}\{e_1, e_2, \ldots, e_k\}$. El Teorema siguiente probará este hecho en general, es decir, cuando A no es necesariamente diagonal. Para mostrar el resultado, antes se necesita la observación siguiente, que se obtiene de las relaciones (2.4) y (2.5b)

$$\begin{aligned} r_{k+1} &= Ax_{k+1} - b \\ &= A(x_k + \alpha_k p_k) - b \\ &= Ax_k - b + \alpha_k Ap_k \\ &= r_k + \alpha_k Ap_k. \end{aligned} \tag{2.8}$$

Teorema 2.2 (Del Subespacio en Expansión)**.** *Sea $x_0 \in \mathbb{R}^n$ un punto inicial y supóngase que la sucesión $\{x_k\}$ es generada por el Algoritmo 2.1. Entonces*

$$r_k^T p_i = 0 \quad \textit{para } i = 0, \ldots, k-1, \tag{2.9}$$

y x_k es el mínimo de $\phi(x) = \frac{1}{2}x^TAx - b^Tx$ sobre la variedad lineal

$$x_0 + gen\{p_0, p_1, \ldots, p_{k-1}\}. \tag{2.10}$$

Demostración. Se comienza mostrando que un punto $\tilde{x}$ minimiza ϕ sobre el conjunto (2.10) si y solo si $r(\tilde{x})^T p_i = 0$, para cada $i = 0, 1, \ldots, k-1$. Se define $h(\sigma) = \phi(x_0+\sigma_1 p_0+\sigma_1 p_1+\cdots+\sigma_{k-1}p_{k-1})$, donde $\sigma = (\sigma_0, \sigma_1, \ldots, \sigma_{k-1})$. Dado que $\phi(h)$ es una cuadrática estrictamente convexa, tiene un único mínimo σ^* que satisface

$$\frac{\partial h(\sigma^*)}{\partial \sigma_i} = 0, \quad i = 0, 1, \ldots, k-1.$$

Por la regla de la cadena, esto implica que

$$\nabla\phi(x_0 + \sigma_0^* p_0 + \cdots + \sigma_{k-1}^* p_{k-1})^T p_i = 0, \quad i = 0, 1, \ldots, k-1.$$

Recordando la Definición 2.1, para el mínimo $\tilde{x} = x_0 + \sigma_0^* p_0 + \sigma_1^* p_1 + \cdots + \sigma_{k-1}^* p_{k-1}$ sobre la variedad lineal (2.10), se tiene que $r(\tilde{x})^T p_i = 0$, como se afirmó.

Por inducción se muestra que x_k satisface (2.9). Para $k = 1$, se tiene que $r_1^T p_0 = 0$, del hecho que $x_1 = x_0 + \alpha_0 p_0$ minimiza ϕ a lo largo de p_0. Ahora, se supone la hipótesis de inducción, es decir, que $r_{k-1}^T p_i = 0$ para $i = 0, 1, \ldots, k-2$. Por (2.8)

$$r_k^T = r_{k-1}^T + \alpha_{k-1} p_{k-1}^T A,$$

así, usando la definición (2.5a) de α_{k-1}, se tiene

$$\begin{aligned} r_k^T p_{k-1} &= r_{k-1}^T p_{k-1} + \alpha_{k-1} p_{k-1}^T A p_{k-1} \\ &= p_{k-1}^T r_{k-1} - \left(\frac{r_{k-1}^T p_{k-1}}{p_{k-1}^T A p_{k-1}} \right) p_{k-1}^T A p_{k-1} \\ &= 0. \end{aligned}$$

Mientras que para los vectores p_i, $i = 0, 1, \ldots, k-2$,

$$r_k^T p_i = r_{k-1}^T p_i + \alpha_{k-1} p_{k-1}^T A p_i = 0,$$

esto debido a la hipótesis de inducción y a la propiedad de conjugación de los vectores p_i. Se ha mostrado que $r_k^T p_i = 0$ para $i = 0, 1, \ldots, k-1$, lo que se quería demostrar. □

Como se puede observar, el Método de Direcciones Conjugadas requiere del conjunto de direcciones conjugadas $\{p_0, p_1, \ldots, p_{n-1}\}$. Este conjunto se puede calcular de diversas maneras, por ejemplo, obteniendo los vectores propios de la matriz A, que no son solo conjugados respecto a A, sino que también forman una base ortogonal. Para aplicaciones donde n es muy grande, el cálculo del conjunto de direcciones conjugadas requiere una gran cantidad de operaciones, así como el almacenamiento de estas. En la siguiente sección se habla del Método de Gradiente Conjugado, que evita la problemática planteada.

2.2. Método de Gradiente Conjugado

El Método de Gradiente Conjugado no es más que una modificación del Método de Direcciones Conjugadas. Esta modificación permite calcular una nueva dirección p_k en cada iteración, que solo depende de la anterior, por lo que no es importante almacenar todas las direcciones generadas, y además, esta nueva dirección en automático será conjugada a todas las direcciones anteriores. La última idea garantiza, en particular, que no se regresa a una dirección ya calculada previamente y, por tanto, no se cae en una situación estacionaria.

Propiedades del Método de Gradiente Conjugado

En el Método de Gradiente Conjugado, cada nueva dirección de búsqueda p_k se elige como una combinación lineal del residuo negativo (que por (2.3) es la dirección de descenso más rápido) y la dirección previa p_{k-1}, es decir,

$$p_k = -r_k + \beta_k p_{k-1}, \tag{2.11}$$

donde el escalar β_k está determinada por la condición de que p_k y p_{k-1} deben ser conjugadas con respecto a A. Premultiplicando (2.11) por $p_{k-1}^T A$ e imponiendo la relación (A.1) se encuentra de manera explicita a β_k.

$$\begin{aligned} 0 &= p_{k-1}^T A p_k \\ &= p_{k-1}^T A(-r_k + \beta_k p_{k-1}) \\ &= -p_{k-1}^T A r_k + \beta_k p_{k-1}^T A p_{k-1}, \end{aligned}$$

despejando β_k,

$$\beta_k = \frac{r_k^T A p_{k-1}}{p_{k-1}^T A p_{k-1}}.$$

Se elige la primera dirección de búsqueda p_0 como la dirección de descenso más rápido en el punto x_0, es decir, $-r_0$. Al igual que el Método de Direcciones Conjugadas, se procede a realizar minimizaciones a lo largo de las direcciones de búsqueda. Por lo tanto, podemos escribir un algoritmo completo, que se muestra a continuación.

Algoritmo 2.2 Método de Gradiente Conjugado (Versión Preliminar)

Entrada: $x_0,\ b$ y A.

Salida: x^*.

Sea $r_0 \leftarrow Ax_0 - b,\ p_0 \leftarrow -r_0,\ k \leftarrow 0$;

mientras $r_k \neq 0$ **hacer**

$$\alpha_k \leftarrow -\frac{r_k^T p_k}{p_k^T A p_k}; \tag{2.12a}$$

$$x_{k+1} \leftarrow x_k + \alpha_k p_k; \tag{2.12b}$$

$$r_{k+1} \leftarrow Ax_{k+1} - b; \tag{2.12c}$$

$$\beta_{k+1} \leftarrow \frac{r_{k+1}^T A p_k}{p_k^T A p_k}; \tag{2.12d}$$

$$p_{k+1} \leftarrow -r_{k+1} + \beta_{k+1} p_k; \tag{2.12e}$$

$$k \leftarrow k + 1; \tag{2.12f}$$

fin mientras

Está versión es útil para estudiar las propiedades esenciales del Método de Gradiente Conjugado, las cuales se mostrarán en el teorema próximo. Antes de empezar se da la definición siguiente.

Definición 2.3. *Se define al subespacio de Krylov de grado k para r_0 como*

$$\mathcal{K}(r_0; k) := gen\{r_0, Ar_0, \ldots, A^k r_0\}. \tag{2.13}$$

Teorema 2.3. *Si la k-ésima iteración generada por el Algoritmo 2.2 no obtiene la solución x^*, las cuatro propiedades siguientes son ciertas:*

$$gen\{r_0, r_1, \ldots, r_k\} = \mathcal{K}(r_0; k), \tag{2.14}$$

$$gen\{p_0, p_1, \ldots, p_k\} = \mathcal{K}(r_0; k), \tag{2.15}$$

$$p_k^T A p_i = 0 \quad para\ i = 0, 1, \ldots, k-1, \tag{2.16}$$

$$r_k^T r_i = 0 \quad para\ i = 0, 1, \ldots, k-1. \tag{2.17}$$

Entonces, la sucesión $\{x_k\}$ converge a x^* en a lo más n iteraciones.

Demostración. La prueba es por inducción. Las expresiones (2.14) y (2.15) son claramente verdaderas para $k = 0$, mientras que (2.16) por construcción es cierta para $k = 1$. Se asume ahora que estas expresiones son ciertas para k y se mostraran para $k + 1$.

Para probar (2.14), se muestra primero que el lado izquierdo esta contenido en el lado derecho. Por la hipótesis de inducción, de (2.14) y (2.15) se tiene que

$$r_k \in \mathcal{K}(r_0; k), \quad p_k \in \mathcal{K}(r_0; k),$$

mientras que multiplicando la segunda expresión por A, se obtiene

$$Ap_k \in \{Ar_0, \ldots, A^{k+1} r_0\}. \tag{2.18}$$

Por (2.8), se encuentra que

$$r_{k+1} = r_k + \alpha_k A p_k \in \mathcal{K}(r_0; k+1).$$

Combinando la última expresión con la hipótesis de inducción para (2.14), se concluye que

$$\text{gen}\{r_0, r_1, \ldots, r_k, r_{k+1}\} \subset \mathcal{K}(r_0; k+1).$$

Para probar la otra contención, se usa la hipótesis de inducción de (2.15), deduciendo que

$$A^{k+1} r_0 = A(A^k r_0) \in \text{gen}\{Ap_0, Ap_1, \ldots, Ap_k\}.$$

Por (2.8) se tiene que $Ap_i = \frac{r_{i+1} - r_i}{\alpha_i}$ para $i = 0, 1, \ldots, k$, resulta

$$A^{k+1} r_0 \in \text{gen}\{r_0, r_1, \ldots, r_k, r_{k+1}\}.$$

Combinado esta expresión con la hipótesis inductiva para (2.14), se tiene

$$\mathcal{K}(r_0; k+1) \subset \{r_0, r_1, \ldots, r_k, r_{k+1}\}.$$

Por lo tanto, la relación (2.14) es verdadera.

Ahora, se mostrará que (2.15) se cumple para $k+1$.

$$\begin{aligned}
&\text{gen}\{p_0, p_1, \ldots, p_k, p_{k+1}\} \\
&\quad = \text{gen}\{p_0, p_1, \ldots, p_k, r_{k+1}\} && \text{por (2.12e)} \\
&\quad = \text{gen}\{r_0, Ar_0, \ldots, A^k r_0, r_{k+1}\} && \text{por h. i. para (2.15)} \\
&\quad = \text{gen}\{r_0, r_1, \ldots, r_k, r_{k+1}\} && \text{por (2.14)} \\
&\quad = \mathcal{K}(r_0; k+1) && \text{por (2.14) para k+1.}
\end{aligned}$$

A continuación se probará la propiedad (2.16) para $k+1$. Multiplicando (2.12e) a la derecha por Ap_i, $i = 0, 1, \ldots, k$, se obtiene

$$p_{k+1}^T A p_i = -r_{k+1}^T A p_i + \beta_{k+1} p_k^T A p_i. \tag{2.19}$$

Por (2.12d) el lado derecho es igual a cero cuando $i = k$. Para $i \leq k-1$ hay que notar algunas cosas. Primero, la hipótesis de inducción para (2.16) implica que las direcciones $p_0, p_1, \ldots, p_k$ son conjugadas, así por el Teorema 2.2 se deduce que

$$r_{k+1}^T p_i = 0, \quad \text{para } i = 0, 1, \ldots, k. \tag{2.20}$$

Segundo, aplicando repetidamente (2.15), se encuentra que para $i = 0, 1, \ldots, k-1$ la contención siguiente

$$\begin{aligned}
Ap_i \in A \cdot \mathcal{K}(r_0; i) &= \text{gen}\{Ar_0, A^2 r_0, \ldots, A^{i+1} r_0\} \\
&\subset \mathcal{K}(r_0; i+1) \\
&= \text{gen}\{p_0, p_1, \ldots, p_{i+1}\}
\end{aligned} \tag{2.21}$$

Por la combinación de (2.20) y (2.21), se deduce que

$$r_{k+1}^T A p_i = 0, \quad \text{para } i = 0, 1, \ldots, k-1,$$

así el primer término del lado derecho de (2.19) se anula para $i = 0, 1, \ldots, k-1$. Debido a la hipótesis de inducción para (2.16), el segundo término también se anula, entonces se concluye que $p_{k+1}^T A p_i = 0$, $i = 0, 1, \ldots, k$. Esto muestra que (2.16) es verdadera.

Se deduce de (2.16) que el Método de Gradiente Conjugado es, de hecho, una modificación del Método de Direcciones Conjugadas, así por el Teorema 2.1, se tiene que el Algoritmo 2.2 termina en a lo más n iteraciones.

Para concluir la demostración, se probara (2.17). Dado que el conjunto de direcciones generado por el Algoritmo 2.2 son conjugadas, entonces por (2.9) se tiene que $r_k^T p_i = 0$ para $i = 0, 1, \ldots, k-1$, con $k = 1, 2, \ldots, n-1$. Por (2.12e) se encuentra

$$r_i = -p_i + \beta_i p_{i-1},$$

así, $r_i \in \text{gen}\{p_i, p_{i-1}\}$ para todo $i = 1, \ldots, k-1$. Se concluye que $r_k^T r_i = 0$. Para completar la prueba, usando la definición de r_0 en el Algoritmo 2.2 se tiene que

$$r_k^T r_0 = -r_k^T p_0 = 0,$$

esto recordando (2.9). □

Las ecuaciones (2.14) y (2.15) muestran la relación entre los vectores de dirección y los residuos. La relación (2.16) solo verifica que el método es de direcciones conjugadas. Por último, la relación (2.17) dice que cada nuevo residuo r_k será ortogonal con los anteriores, hecho que será útil para intuir una forma de calcular α_k y β_k de tal manera que requiera menos operaciones, haciendo su implementación más rápida.

Forma práctica del Método de Gradiente Conjugado

Para deducir una forma más económica del Método de Gradiente Conjugado se usarán los resultados de los Teoremas 2.2 y 2.3. Para calcular α_k, usando (2.12e) y (2.9), se tiene que

$$\alpha_k = -\frac{r_k^T p_k}{p_k^T A p_k} = -\frac{r_k^T(-r_k + \beta_k p_{k-1})}{p_k^T A p_k} = \frac{r_k^T r_k}{p_k^T A p_k}.$$

Para el cálculo de β_k, de (2.8) se tiene que $\alpha_k A p_k = r_{k+1} - r_k$, así mediante la aplicación de (2.12e) y (2.9) se deduce

$$\begin{aligned}
\beta_{k+1} &= \frac{r_{k+1}^T A p_k}{p_k^T A p_k} \cdot \frac{\alpha_k}{\alpha_k} \\
&= \frac{r_{k+1}^T r_{k+1}}{-p_k^T r_k} \\
&= \frac{r_{k+1}^T r_{k+1}}{-(-r_k + \beta_k p_{k-1})^T r_k} \\
&= \frac{r_{k+1}^T r_{k+1}}{r_k^T r_k}.
\end{aligned}$$

Como se puede ver, la aplicación de estas nuevas expresiones significa una mejora en el número de operaciones, porque es más rápido calcular un producto de dos vectores que el producto de una matriz por un vector. Otra consideración a hacer es el cálculo de r_{k+1} en el Algoritmo 2.2, ya que se puede calcular por medio de la relación (2.8), esto evita el almacenamiento de b, lo que significa ahorro de memoria (en aplicaciones computacionales). Con estás aclaraciones se formula una versión más eficiente del Método de Gradiente Conjugado.

Algoritmo 2.3 Método de Gradiente Conjugado

Entrada: $x_0,\ b$ y A.

Salida: x^*.

Sea $r_0 \leftarrow Ax_0 - b,\ p_0 \leftarrow -r_0,\ k \leftarrow 0$;

mientras $r_k \neq 0$ **hacer**

$$\alpha_k \leftarrow \frac{r_k^T r_k}{p_k^T A p_k}; \tag{2.22a}$$

$$x_{k+1} \leftarrow x_k + \alpha_k p_k; \tag{2.22b}$$

$$r_{k+1} \leftarrow r_k + \alpha_k A p_k; \tag{2.22c}$$

$$\beta_{k+1} \leftarrow \frac{r_{k+1}^T r_{k+1}}{r_k^T r_k}; \tag{2.22d}$$

$$p_{k+1} \leftarrow -r_{k+1} + \beta_{k+1} p_k; \tag{2.22e}$$

$$k \leftarrow k + 1; \tag{2.22f}$$

fin mientras

Cotas de Convergencia

En una platica con el Dr. Gilberto Calvillo Vives durante la ENOAN 2017, al preguntar sobre la velocidad de convergencia del Método de Gradiente Conjugado, su respuesta fue la siguiente:

"Hablar de la rapidez de convergencia no tiene sentido, ya que el método de gradiente obtiene la solución de manera exacta en a lo más n iteraciones, por lo que siempre encontraremos la solución exacta en un número finito de pasos, aunque si tenemos la suerte de dar un punto inicial sobre un vector propio, el algoritmo sólo necesitará de una iteración y no las n."

En cuanto a la respuesta dada por el Dr. Calvillo, ya se ha probado que en la aritmética exacta del Método de Gradiente Conjugada, se obtiene la solución en a lo más en n iteraciones, solo falta comprobar la segunda parte de su respuesta.

Si $x_0 \neq x^*$ está sobre una recta $x^* + \alpha v_i$ (véase Imagen 2.3), donde v_i es un vector propio asociado al valor propio $\lambda_i > 0$, entonces para algún $\sigma > 0$,

$x_0 = x^* + \sigma v_i$, así

$$r_0 = Ax_0 - b = A(x^* + \sigma v_i) - b = \sigma\lambda_i v_i,$$

por lo que

$$p_0 = -\sigma\lambda_i v_i,$$

y la longitud de paso está determinado por

$$\alpha_0 = \frac{(\sigma\lambda_i v_i)^T(\sigma\lambda_i v_i)}{(-\sigma\lambda_i v_i)^T A(-\sigma\lambda_i v_i)} = \frac{v_i^T v_i}{\lambda_i v_i^T v_i} = \frac{1}{\lambda_i},$$

entonces

$$x_1 = x^* + \alpha v_i + \frac{1}{\lambda_i}(-\sigma\lambda_i v_i) = x^*.$$

Por lo que solo se necesita una iteración en este caso. Es claro que no es fácil elegir x_0 de tal manera que ocurra lo visto en la Imagen 2.3, pues seria un caso muy favorable.

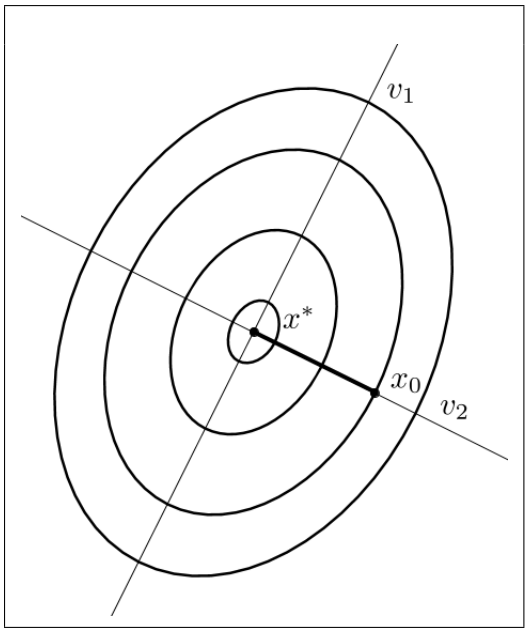

Imagen 2.3: El punto inicial x_0 está sobre un eje definido por un vector propio, por lo que la convergencia ocurre en una sola iteración.

A continuación se presentan algunas cotas de convergencia que dependen de las características de los valores propios de A, ya que la distribución de

estos influye en el número de iteraciones necesarias para obtener la solución. Sin más que agregar se procede al análisis para calcular estas cotas.

De las relaciones (2.22b) y (2.15), se tiene

$$\begin{aligned} x_{k+1} &= x_0 + \alpha_0 p_0 + \cdots + \alpha_k p_k \\ &= x_0 + \rho_0 r_0 + \rho_1 A r_0 + \cdot + \rho_k A^k r_0, \end{aligned} \tag{2.23}$$

para algunos escalares $\rho_i \in \mathbb{R}$. Con esto se define a $P_k^*(\cdot)$ como el polinomio de grado k con coeficientes $\rho_0, \rho_1, \ldots, \rho_k$. Para la matriz A, tenemos

$$P_k^*(A) = \rho_0 I + \rho_1 A + \cdots + \rho_k A^k,$$

que permite escribir (2.23) como

$$x_{k+1} = x_0 + P_k^*(A) r_0, \tag{2.24}$$

ya que

$$r_0 = Ax_0 - b = Ax_0 - Ax^* = A(x_0 - x^*),$$

se obtiene

$$x_{k+1} - x^* = x_0 + P_k^*(A) r_0 - x^* = [I + P_k^*(A)A](x_0 - x^*). \tag{2.25}$$

Ahora se mostrará que el Algoritmo 2.3 hace un buen trabajo minimizando la distancia de la solución x^* al espacio de Krylov $\mathcal{K}(r_0; k)$ cuando se usa la norma inducida por A.

Definición 2.4. *Si A es definida positiva, entonces para $z \in \mathbb{R}^n$ se define la norma inducida por A como*

$$\|z\|_A^2 = z^T A z. \tag{2.26}$$

Proposición 2.1. *Para cualquier $x \in \mathbb{R}^n$, se cumple que*

$$\frac{1}{2}\|x - x^*\|_A^2 = \phi(x) - \phi(x^*). \tag{2.27}$$

Demostración. Sea $x \in \mathbb{R}$, entonces

$$\begin{aligned} \frac{1}{2}\|x - x^*\|_A^2 &= \frac{1}{2}(x - x^*)^T A (x - x^*) \\ &= \frac{1}{2}(x^T A x - 2x^{*T} A x + x^{*T} A x^*) \\ &= \frac{1}{2}x^T A x - b^T x + \frac{1}{2}x^{*T} A x^* - x^{*T} A x^* + x^{*T} A x^* \\ &= \frac{1}{2}x^T A x - b^T x - \frac{1}{2}x^{*T} A x^* + b^T x^* \\ &= \phi(x) - \phi(x^*). \end{aligned}$$

□

El Teorema 2.2 establece que x_{k+1} minimiza ϕ, y por la Proposición 2.1 a $\|x - x^*\|_A^2$, sobre la variedad lineal $x_0 + \text{gen}\{p_0, p_1, \ldots, p_k\}$, que por (2.15) es lo mismo que $x_0 + \text{gen}\{r_0, Ar_0, \ldots, A^k r_0\}$. Se sigue de (2.25) que el polinomio P_k^* resuelve el problema

$$\min_{P_k} \|x_0 + P_k(A)r_0 - x^*\|_A, \tag{2.28}$$

donde el mínimo se toma en el espacio de todos los polinomios con coeficientes reales de grado k.

Por otro lado, sean $0 < \lambda_1 \leq \lambda_2 \leq \cdots \leq \lambda_n$ los valores propios de A, y $v_1, v_2, \ldots, v_n$ sus correspondientes vectores propios, que se pueden suponer unitarios y ortogonales. Dado que los vectores propios generan todo el espacio $\mathbb{R}^n$, se puede escribir

$$x_0 - x^* = \sum_{i=1}^{n} \xi_i v_i, \tag{2.29}$$

para algunos escalares $\xi_i \in \mathbb{R}$. Como cualquier vector propio de A es también vector propio de $P_k(A)$, esto para cualquier polinomio de grado k, se tiene

$$P_k(A)v_i = P_k(\lambda_i)v_i.$$

Por sustituir (2.29) en (2.25), se tiene que

$$\begin{aligned} x_{k+1} - x^* &= [I + P_k^*(A)A] \sum_{i=1}^{n} \xi_i v_i \\ &= \sum_{i=1}^{n} [v_i + \lambda_i P_k^*(A)v_i]\xi_i \\ &= \sum_{i=1}^{n} [I + \lambda_i P_k^*(A)]\xi_i v_i \\ &= \sum_{i=1}^{n} [1 + \lambda_i P_k^*(\lambda_i)]\xi_i v_i \end{aligned} \tag{2.30}$$

Por la descomposición espectral, A se puede expresar como

$$A = \sum_{i=1}^{n} \lambda_i v_i v_i^T,$$

así,

$$\|z\|_A^2 = z^T A z = z^T \left(\sum_{i=1}^{n} \lambda_i v_i v_i^T \right) z = \sum_{i=1}^{n} \lambda_i (v_i^T z)^2.$$

Con la relación (2.30) y lo anterior,

$$\begin{aligned} \|x_{k+1} - x^*\|_A^2 &= \sum_{i=1}^{n} \lambda_i \left(v_i^T \sum_{j=1}^{n} [1 + \lambda_j P_k^*(\lambda_j)] \xi_j v_j \right)^2 \\ &= \sum_{i=1}^{n} \lambda_i ([1 + \lambda_i P_k^*(\lambda_i)] \xi_i v_i^T v_i)^2 \\ &= \sum_{i=1}^{n} \lambda_i [1 + \lambda_i P_k^*(\lambda_i)]^2 \xi_i^2. \end{aligned}$$

Dado que el polinomio P_k^* generado por el Método de Gradiente Conjugado es óptimo con la norma $\|\cdot\|_A$, se deduce

$$\|x_{k+1} - x^*\|_A^2 = \min_{P_k} \sum_{i=1}^{n} \lambda_i [1 + P_k(\lambda_i)]^2 \xi_i^2.$$

Por extraer el máximo de los términos $[1 + \lambda_i P_k(\lambda_i)]^2$ de esta expresión, se obtiene

$$\begin{aligned} \|x_{k+1} - x^*\|_A^2 &\leq \min_{P_k} \max_{1 \leq i \leq n} [1 + \lambda_i P_k(\lambda_i)]^2 \left(\sum_{j=1}^{n} \lambda_j \xi_j^2 \right) \\ &= \min_{P_k} \max_{1 \leq i \leq n} [1 + \lambda_i P_k(\lambda_i)]^2 \|x_0 - x^*\|_A^2, \end{aligned} \tag{2.31}$$

donde

$$\|x_0 - x^*\|_A^2 = \sum_{j=1}^{n} \lambda_j \xi_j^2.$$

La expresión (2.31) permite obtener cotas de convergencia del Algoritmo 2.3 estimando a cantidad no negativa

$$\min_{P_k} \max_{1 \leq i \leq n} [1 + \lambda_i P_k(\lambda_i)]^2. \tag{2.32}$$

Ahora se procede a analizar (2.32) para obtener algunas cotas de convergencia del Método de Gradiente Conjugado.

Teorema 2.4. *Si A tiene solo r valores propios distintos, entonces el Método de Gradiente Conjugado termina en a lo más r iteraciones.*

Demostración. Se supone que los valores $\lambda_1, \lambda_2, \ldots, \lambda_n$ toman r valores distintos $\tau_1 < \tau_2 < \cdots < \tau_r$. Se define un polinomio $Q_r(\lambda)$ como

$$Q_r(\lambda) = \frac{(-1)^r}{\tau_1 \tau_2 \cdots \tau_r}(\lambda - \tau_1)(\lambda - \tau_2)\cdots(\lambda - \tau_r),$$

el cual $Q_r(\lambda_i) = 0$ para $i = 1, 2, \ldots, n$ y $Q_r(0) = 1$. De esta observación, se deduce que $Q_r(\lambda) - 1$ es un polinomio de grado r y con raíz en $\lambda = 0$, así por división polinomial se define

$$\tilde{P}_{r-1}(\lambda) = \frac{Q_r(\lambda) - 1}{\lambda},$$

el cual es de grado $r - 1$. Sea $k = r - 1$ en (2.32), se tiene

$$0 \leq \min_{P_{r-1}} \max_{1 \leq i \leq n} [1 + \lambda_i P_{r-1}(\lambda_i)]^2 \leq \max_{1 \leq i \leq n} [1 + \lambda_i \tilde{P}_{r-1}(\lambda_i)]^2 = \max_{1 \leq i \leq n} Q_r^2(\lambda_i) = 0.$$

Por lo tanto, la constante (2.32) es cero para el valor $k = r - 1$, así que de (2.31) se tiene que $\|x_r - x^*\|_A^2 = 0$, por lo que $x_r = x^*$. □

Ahora se da otra cota útil para estudiar el Método de Gradiente Conjugado.

Teorema 2.5. *Si A tiene valores propios $\lambda_1 \leq \lambda_2 \leq \cdots \leq \lambda_n$, entonces para el Algoritmo 2.3 se tiene que*

$$\|x_{k+1} - x^*\|_A^2 \leq \left(\frac{\lambda_{n-k} - \lambda_1}{\lambda_{n-k} + \lambda_1}\right)^2 \|x_0 - x^*\|_A^2. \tag{2.33}$$

Demostración. De la relación (2.31) se tiene que

$$\|x_{k+1} - x^*\|_A^2 \leq \min_{P_k} \max_{1 \leq i \leq n} [1 + \lambda_i P_k(\lambda_i)]^2 \|x_0 - x^*\|_A^2.$$

Se elije $\tilde{P}_k$ de modo que $Q_{k+1}(\lambda) = 1 + \lambda \tilde{P}_k(\lambda)$ de grado $k+1$ que se anule en $\frac{\lambda_{n-k} + \lambda_1}{2}$ y en los k valores propios más grandes de A, esto se ilustra en la Imagen 2.4. Para esta elección de $\tilde{P}_k$, de (2.31) se deduce

$$\|x_{k+1} - x^*\|_A^2 \leq \max_{1 \leq i \leq n-k} [1 + \lambda_i \tilde{P}_k(\lambda_i)]^2 \|x_0 - x^*\|_A^2.$$

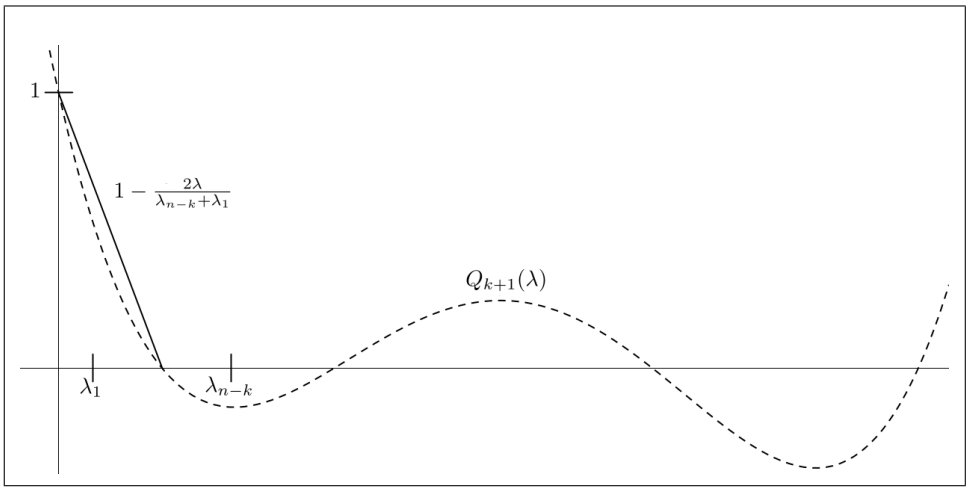

Imagen 2.4: Comportamiento del polinomio $Q_{k+1}(\lambda)$.

Como el polinomio $Q_{k+1}(\lambda)$ tiene $k+1$ raíces reales, $Q'_{k+1}(\lambda)$ tendrá k raíces reales que se alternan en el eje real con las raíces de $Q_{k+1}(\lambda)$. Asimismo, $Q''_{k+1}(\lambda)$ tendrá $k-1$ raíces reales que se alternaran con las raíces de $Q'_{k+1}(\lambda)$. Como $Q_{k+1}(\lambda)$ no tiene raíces en el intervalo $\left(-\infty, \frac{\lambda_{n-k}+\lambda_1}{2}\right)$, se observa que $Q''_{k+1}(\lambda)$ no cambia de signo en ese intervalo, y además $Q''_{k+1}(\lambda) > 0$ para $\lambda < \frac{\lambda_{n-k}+\lambda_1}{2}$, por lo que $Q_{k+1}(\lambda)$ es convexa en dicho intervalo. Por tanto, en $\left[0, \frac{\lambda_{n-k}+\lambda_1}{2}\right]$, $Q_{k+1}(\lambda)$ está por debajo de la recta $1 - \frac{2\lambda}{\lambda_{n-k}+\lambda_1}$. Así, se concluye que

$$Q_{k+1}(\lambda) \leq 1 - \frac{2\lambda}{\lambda_{n-k}+\lambda_1},$$

en $\left[0, \frac{\lambda_{n-k}+\lambda_1}{2}\right]$ y que

$$Q'_{k+1}\left(\frac{\lambda_{n-k}+\lambda_1}{2}\right) \geq -\frac{2}{\lambda_{n-k}+\lambda_1}.$$

Se puede observar que en $\left[\frac{\lambda_{n-k}+\lambda_1}{2}, \lambda_{n-k}\right]$,

$$Q_{k+1}(\lambda) \geq 1 - \frac{2\lambda}{\lambda_{n-k}+\lambda_1},$$

ya que para que $Q_{k+1}(\lambda)$ cruce primero la recta $1 - \frac{2\lambda}{\lambda_{n-k}+\lambda_1}$, y después el eje λ, se requiere dos cambios de signo de $Q''_{k+1}(\lambda)$, en tanto que existe como máximo una raíz de $Q''_{k+1}(\lambda)$ a la izquierda de la segunda raíz de de $Q_{k+1}(\lambda)$. Entonces, se observa que la desigualdad

$$|1 + \lambda \tilde{P}_k(\lambda)| \leq \left| 1 - \frac{2\lambda}{\lambda_{n-k} + \lambda_1} \right|,$$

es válida en el intervalo $[\lambda_1, \lambda_{n-k}]$. Así,

$$\|x_{k+1} - x^*\|_A^2 \leq \left(1 - \frac{2\lambda}{\lambda_{n-k} + \lambda_1} \right)^2 \|x_0 - x^*\|_A^2,$$

y en especial en $\lambda = \lambda_1$,

$$\|x_{k+1} - x^*\|_A^2 \leq \left(\frac{\lambda_{n-k} - \lambda_1}{\lambda_{n-k} + \lambda_1} \right)^2 \|x_0 - x^*\|_A^2.$$

□

Para la última cota se necesita las definiciones siguientes.

Definición 2.5. *El número de condición de la matriz A no singular se define como*

$$\kappa(A) = \|A\|\|A^{-1}\|,$$

donde $\|\cdot\|$ *es cualquier norma matricial.*

Definición 2.6. *Para cualquier matriz A se define*

$$\|A\|_2 \equiv \textit{el mayor valor propio de } (A^T A)^{1/2}.$$

Teorema 2.6. *Para el Algoritmo 2.3 se tiene que*

$$\|x_k - x^*\|_A \leq 2 \left(\frac{\sqrt{\kappa(A)} - 1}{\sqrt{\kappa(A)} + 1} \right)^k \|x_0 - x^*\|_A, \tag{2.34}$$

donde

$$\kappa(A) = \|A\|_2 \|A^{-1}\|_2 = \frac{\lambda_n}{\lambda_1} \geq 1.$$

Demostración. Si $\lambda_1 = \lambda_n$, entonces el Método de Gradiente Conjugado obtiene la solución en una sola iteración, por lo que (2.34) se verifica para $\kappa(A) = 1$.

Si $\lambda_1 < \lambda_n$, por la relación (2.31) se tiene

$$\|x_k - x^*\|_A \leq \min_{P_{k-1}} \max_{1 \leq i \leq n} |1 + \lambda_i P_{k-1}(\lambda_i)| \|x_0 - x^*\|_A.$$

Se considera

$$Q_k = \frac{T_k\left(\frac{\lambda_n + \lambda_1 - 2\lambda}{\lambda_n - \lambda_1}\right)}{T_k\left(\frac{\lambda_n + \lambda_1}{\lambda_n - \lambda_1}\right)}, \quad \lambda \in [\lambda_1, \lambda_n], \tag{2.35}$$

donde T_k es el polinomio de Chebyshev de grado k, que está definido por:

$$T_k = \begin{cases} \frac{1}{2}\left[\left(z + \sqrt{z^2 - 1}\right)^k + \left(z - \sqrt{z^2 - 1}\right)^k\right], & z \in \mathbb{R} \setminus (-1, 1) \\ \cos(k \arccos z), & z \in [-1, 1] \end{cases} \tag{2.36}$$

y además se observa que

$$Q_k(0) = 1,$$

lo cual implica que $Q_k(\lambda) = 1 + \lambda \tilde{P}_{k-1}(\lambda)$ para algún polinomio $\tilde{P}_{k-1}$ de grado $k - 1$.

Se observa que

$$1 < \frac{\lambda_1 + \lambda_n}{\lambda_n} < \frac{\lambda_n + \lambda_1}{\lambda_n - \lambda_1},$$

así de (2.36) y la definición de $\kappa(A)$ se tiene

$$\begin{aligned}
&2T_k\left(\frac{\lambda_n+\lambda_1}{\lambda_n-\lambda_1}\right)\\
&=\left[\frac{\lambda_n+\lambda_1}{\lambda_n-\lambda_1}+\sqrt{\left(\frac{\lambda_n+\lambda_1}{\lambda_n-\lambda_1}\right)^2-1}\right]^k+\left[\frac{\lambda_n+\lambda_1}{\lambda_n-\lambda_1}-\sqrt{\left(\frac{\lambda_n+\lambda_1}{\lambda_n-\lambda_1}\right)^2-1}\right]^k\\
&=\left[\frac{\lambda_n+\lambda_1+2\sqrt{\lambda_n\lambda_1}}{\lambda_n-\lambda_1}\right]^k+\left[\frac{\lambda_n+\lambda_1-2\sqrt{\lambda_n\lambda_1}}{\lambda_n-\lambda_1}\right]^k\\
&=\left[\frac{\left(\sqrt{\lambda_n}+\sqrt{\lambda_1}\right)^2}{\left(\sqrt{\lambda_n}-\sqrt{\lambda_1}\right)\left(\sqrt{\lambda_n}+\sqrt{\lambda_1}\right)}\right]^k+\left[\frac{\left(\sqrt{\lambda_n}-\sqrt{\lambda_1}\right)^2}{\left(\sqrt{\lambda_n}-\sqrt{\lambda_1}\right)\left(\sqrt{\lambda_n}+\sqrt{\lambda_1}\right)}\right]^k\\
&>\left[\frac{\left(\sqrt{\lambda_n}+\sqrt{\lambda_1}\right)^2}{\left(\sqrt{\lambda_n}-\sqrt{\lambda_1}\right)\left(\sqrt{\lambda_n}+\sqrt{\lambda_1}\right)}\right]^k=\left(\frac{\sqrt{\lambda_n}+\sqrt{\lambda_1}}{\sqrt{\lambda_n}-\sqrt{\lambda_1}}\right)^k=\left(\frac{\sqrt{\kappa(A)}+1}{\sqrt{\kappa(A)}-1}\right)^k
\end{aligned} \tag{2.37}$$

Si $\lambda\in[\lambda_1,\lambda_n]$, entonces $\frac{\lambda_n+\lambda_1-2\lambda}{\lambda_n-\lambda_1}\in[-1,1]$, y por (2.36)

$$|T_k(z)|\le 1 \quad \text{para todo } z\in[-1,1].$$

Así, (2.35) y (2.37) implica

$$Q_k(\lambda)<\frac{1}{\frac{1}{2}\left(\frac{\sqrt{\kappa(A)}+1}{\sqrt{\kappa(A)}-1}\right)^k}=2\left(\frac{\sqrt{\kappa(A)}-1}{\sqrt{\kappa(A)}+1}\right)^k.$$

Recordando (2.31),

$$\|x_k-x^*\|_A\le \max_{1\le i\le n}|Q_k(\lambda_i)|\|x_0-x^*\|_A\le 2\left(\frac{\sqrt{\kappa(A)}-1}{\sqrt{\kappa(A)}+1}\right)^k\|x_0-x^*\|_A.$$

□

Teorema 2.7. *Si los valores propios ocurren en r clusters distintos (es decir, los valores están agrupados en r intervalos de longitud pequeña), el Método de Gradiente Conjugado se aproxima a la solución en un número de iteraciones cercano a r.*

Demostración. Se elige P_{r-1} tal que $Q(\lambda) = 1+\lambda P_{r-1}(\lambda)$ tenga ceros en cada uno de los clusters. Se considera $Q(\lambda)$ definido de $[0, \lambda_n + 1]$ a $\mathbb{R}$, donde λ_n es el mayor valor propio. El polinomio Q no necesariamente tendrá valor cero al ser evaluado en cada uno de los valores propios, pero al ser los polinomios uniformemente continuos en un dominio compacto, $Q(\lambda_i)$ tomará valores pequeños, así que (2.32) toma un valor pequeño, con lo cual se concluye el resultado. □

2.3. Método de Gradiente Conjugado Parcial

Un tipo de procedimiento que es natural considerar es aquel en que el Método de Gradiente Conjugado, se realiza para $m < n-1$ pasos y entonces, en lugar de continuar con la iteración $m+2$, se vuelve a iniciar el Algoritmo 2.3 desde el punto x_{m+1} desde el inicio. Esto se ilustra en el Algoritmo 2.4.

Algoritmo 2.4 Método de Gradiente Conjugado Parcial

Entrada: $x_0,\ b,\ A,\ m < n-1$ y $ERROR > 0$.
Salida: x^*.

mientras $\|r_0\| \geq ERROR$ **hacer**
Sea $r_0 \leftarrow Ax_0 - b,\ p_0 \leftarrow -r_0,\ k \leftarrow 0$;
mientras $k < m+1$ **hacer**

$$\begin{aligned}
\alpha_k &\leftarrow \frac{r_k^T r_k}{p_k^T A p_k};\\
x_{k+1} &\leftarrow x_k + \alpha_k p_k;\\
r_{k+1} &\leftarrow r_k + \alpha_k A p_k;\\
\beta_{k+1} &\leftarrow \frac{r_{k+1}^T r_{k+1}}{r_k^T r_k};\\
p_{k+1} &\leftarrow -r_{k+1} + \beta_{k+1} p_k;\\
k &\leftarrow k+1;
\end{aligned}$$

fin mientras
$x_0 \leftarrow x_k$;
fin mientras

Como se puede observar, en el Algoritmo 2.4 no se tiene como criterio de paro que el residuo en el punto encontrado sea igual a cero, pues al restringir el número de iteraciones en el Método de Gradiente Conjugado no se asegura que se llegue a una solución exacta.

El Teorema 2.5 es útil para predecir el comportamiento del Algoritmo 2.4. Se considera que los valores propios de A son de dos tipos: m valores grandes y $n-m$ valores propios más pequeños en el intervalo $[\lambda_1, \lambda_{n-m}]$. Si $\epsilon = \lambda_{n-m} - \lambda_1$, y además $\lambda_{n-m} + \lambda_1 > 1$, entonces de la relación (2.33) se tiene que

$$\|x_{m+1} - x^*\|_A \leq \epsilon \|x_0 - x^*\|_A.$$

Para un valor pequeño de ϵ, se concluye que las iteraciones del Método de Gradiente Conjugado proporcionan una buena estimación de la solución en tan solo $m+1$ pasos.

La Imagen 2.5 muestra el comportamiento del Algoritmo 2.3 cuando A tiene cinco valores propios cercanos a 1 (a una distancia menor de 0.05) y tres valores propios considerados grandes. Se traza la poligonal de $\log \|x_k - x^*\|$. Para este caso el Teorema 2.5, predice una fuerte disminución en la medida del error en la iteración 4, aunque después de dos iteraciones más, la aproximación es mejor como se observa en la Imagen 2.5 (con $\|r_k\| < 0.0001$). Por otro lado, la disminución no cambio mucho con respecto a la iteración 4 con respecto a la anterior, esto muestra que (2.33) sólo es una cota superior, y la tasa de convergencia puede ser más rápida de lo que predice.

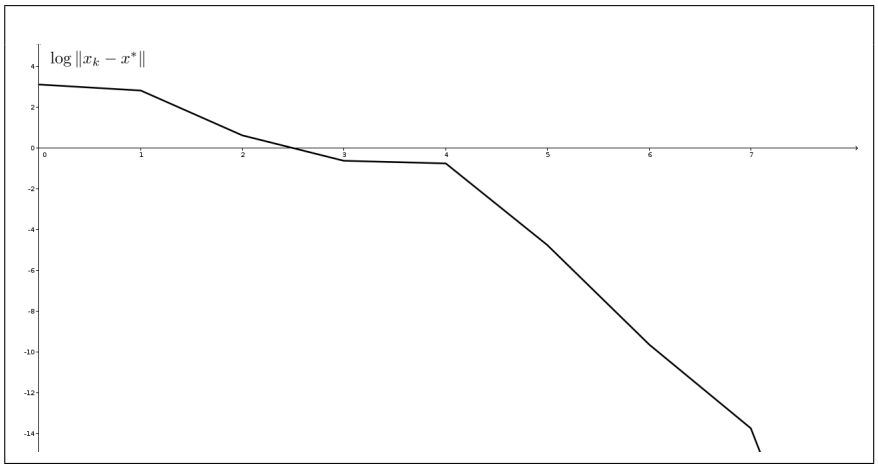

Imagen 2.5: Comportamiento del Método de Gradiente Conjugado en un problema con tres valores propios grandes y cinco alrededor de 1.

Por otra parte, la Imagen 2.6 muestra el comportamiento del Método de Gradiente Conjugado cuando la matriz A tiene cuatro valores propios cercanos a 2 y tres cercanos a 10 (a una distancia menor de 0.05) y tres valores propios considerados grandes. El Teorema 2.7, en este caso, predice una buena aproximación en un número cercano a 5 iteraciones, ya que se puede considerar que los valores propios están en 5 clusters. Como se observa en la Imagen 2.6 se obtiene una buena solución en 8 iteraciones (con $\|r_k\| < 0.0001$).

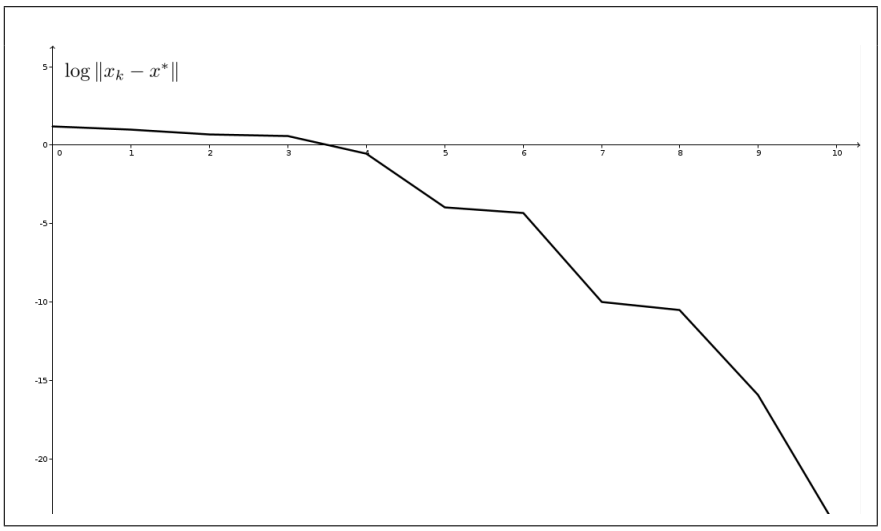

Imagen 2.6: Comportamiento del Método de Gradiente Conjugado con dos cluster alrededor de 2 y 10, cada uno con cuatro y tres elementos respectivamente, y tres valores propios distintos considerados grandes.

Capítulo 3

Implementación Computacional

Cuando se trabaja con un problema del tipo (2.1) donde n es pequeño, los cálculos se pueden realizar a mano. Para n grande, hacer los cálculos ha mano implica una gran cantidad de tiempo, por lo que se opta por utilizar un lenguaje de programación o software para resolver esta problemática.

En los algoritmos anteriores se supone que se trabaja con una aritmética exacta, lo que permite que el Algoritmo 2.3 obtenga la solución exacta, pero como ya se sabe, la computadora trabaja con aproximaciones de los números, en consecuencia, la solución no es exacta.

No siempre se puede disponer de un buen software que realice operaciones simbólicas, por lo que se opta por realizar códigos de programación para implementar algoritmos, los cuales pueden ser tan básicos o elaborados, esto depende de las necesidades del programador.

Sin mas que decir, se procede a desarrollar los algoritmos para su implementación en el lenguaje C (aunque se pueden implementar en otros lenguajes) y posteriormente se presenta el código.

3.1. Algoritmo Gradiente Conjugado

A continuación se comentan las modificaciones que se realizaran para hacer más eficaz la implementación del Método de Gradiente Conjugado.

Se agregan las siguientes variables:

1. $m \leq n-1$ tiene el número de iteraciones que el usuario quiere realizar, esté será el primer criterio de paro.

2. $ERROR > 0$ almacena el error que se desea permitir, con lo que el segundo criterio de paro es $\|r_k\| < ERROR$.

3. z guarda el valor de Ap_k en cada iteración.

4. γ y ρ tienen los valores de $r_k^T r_k$ y $r_{k+1}^T r_{k+1}$ respectivamente en cada iteración.

Con estás modificaciones se propone el siguiente algoritmo que permite un mejor manejo de la memoria y evita repetir operaciones.

Algoritmo 3.1 Algoritmo de Gradiente Conjugado Parcial para su implementación computacional

---

Entrada: $n > 0$, $m \leq n-1$, x_0, b, A.
Salida: x^*.

Sea $k \leftarrow 0$, $r \leftarrow Ax - b$, $p \leftarrow -r$ y $\rho \leftarrow r^T r$.
mientras $k < m+1$ **y** $\sqrt{\rho} \geq ERROR$ **hacer**

$$
\begin{aligned}
&z \leftarrow Ap;\\
&\alpha \leftarrow \frac{\rho}{p^T z};\\
&x \leftarrow x + \alpha p;\\
&r \leftarrow r + \alpha z;\\
&\gamma \leftarrow \rho;\\
&\rho \leftarrow r^T r;\\
&\beta \leftarrow \frac{\rho}{\gamma};\\
&p \leftarrow -r + \beta p;\\
&k \leftarrow k + 1;
\end{aligned}
$$

fin mientras

Observación: Si en el Algoritmo 3.1 $m = n-1$, entonces se tiene el Método de Gradiente Conjugado.

Número de operaciones por iteración

Una de las formas de analizar la eficacia de un código es por tiempo de ejecución o el número de operaciones. Para comparar los Algoritmos 3.1 y 2.3 se considerara el número de operaciones por iteración. Se considera una "operación" a la suma, resta, multiplicación o división entre dos escalares.

A continuación analizaremos algunas operaciones necesarias en el Algoritmos 2.3 y 3.1.

Número de operaciones en cálculos matriciales

Operación matricial	**Forma matemática**	**Número de operaciones**
Matriz por vector	Ab	$2n^2 - n$
Productor escalar	$b^T c$	$2n - 1$
Suma o resta de vectores	$b \pm c$	n
Escalar por vector	βb	n

Tabla 3.1: Número de operaciones en algunas operaciones matriciales. Se considera $A \in M_{n\times n}(\mathbb{R})$, $b, c \in \mathbb{R}^n$ y $\beta \in \mathbb{R}$.

La Tabla 3.1 contiene el número de operaciones de los cálculos matriciales que utiliza el Método de Gradiente Conjugado, con dicha información se procede a calcular el número de operaciones de los Algoritmos 2.3 y 3.1, lo cual se muestra en las Tablas 3.2 y 3.3 respectivamente.

Operaciones por iteración en el Algoritmo 2.3

Operación	**Frecuencia**	**Total**
Matriz por vector	2	$4n^2 - 2n$
Productor escalar	4	$8n - 4$
Suma o resta de vectores	3	$3n$
Escalar por vector	3	$3n$
Operaciones independientes	3	3
	Total	$4n^2 + 12n - 1$

Tabla 3.2: Operaciones en el Algoritmo 2.3. Se considera $A \in M_{n\times n}(\mathbb{R})$, $b, c \in \mathbb{R}^n$ y $\beta \in \mathbb{R}$.

Operaciones por iteración en el Algoritmo 3.1

Operación	Frecuencia	Total
Matriz por vector	1	$2n^2 - n$
Productor escalar	2	$4n - 2$
Suma o resta de vectores	3	$3n$
Escalar por vector	3	$3n$
Operaciones independientes	3	3
	Total	$2n^2 + 9n + 1$

Tabla 3.3: Operaciones en el Algoritmo 3.1. Se considera $A \in M_{n\times n}(\mathbb{R})$, $b, c \in \mathbb{R}^n$ y $\beta \in \mathbb{R}$.

De las Tablas 3.2 y 3.3 se concluye que el número de operaciones del Algoritmo 3.1 es menor al Algoritmo 2.3, por lo que es una buena alternativa para ser implementado.

3.2. Explicación del Código

El código realizado trabaja de la siguiente manera:

1. Al correr el programa es necesario un argumento, el nombre del archivo donde se encuentra la información del sistema $Ax = b$. Este incluye el punto inicial x_0, el número de variables, y las entradas de A.

2. Se pide el número de bits que serán utilizados para almacenar reales.

3. Se procede a crear los vectores y matrices de tamaño adecuado para implementar el código, si no es posible, se mostrara un mensaje de memoria insuficiente.

4. Después de leer la información, se procede a pedir el número máximo de iteraciones, este dato se guarda en la variable $m < n - 1$.

5. Se pide el $ERROR$, que debe ser un número positivo.

6. Se aplica el Algoritmo 3.1.

7. Al final se muestra el número de iteraciones que se utilizaron, el punto obtenido y la norma del último residuo obtenido.

3.3. Comparación real de los Algoritmos

Ya se exhibió que matemáticamente el Algoritmo 3.1 realiza menos operaciones por iteración que el Algoritmo 2.3, por lo que se procede a compararlos en ejemplos reales. Con el objetivo de comparar Algoritmos lo haremos de la siguiente manera:

Método para comparar Algoritmos

1. Se obtiene aleatoriamente una matriz $R \in M_{n\times n}(\mathbb{R})$ triangular superior, tal que:

   1.1. Las entradas r_{ij} que están por encima de la diagonal superior son enteros que varían entre los enteros -50 y 50.

   1.2. Las entradas r_{ii} de la diagonal principal son enteros que varían entre los -50 y 50, pero evitando el valor 0, esto para evitar que la matriz R sea singular.

2. Se calcula $A = R^T R$, la cual es simétrica y definida positiva por la forma en la que se define R. Es posible multiplicar a A por un múltiplo positivo, esto con el fin de que el valor propio más pequeño no sea tan pequeño.

3. Una vez calculada A, se procede a crear el sistema $Ax = b$ y el punto inicial x_0, donde b y x_0 también tendrán enteros entre -50 y 50 de forma aleatoria. Con el fin de tener más ejemplos, también se puede multiplicar en esta parte a A, b y x_0 por un múltiplo positivo.

Con los pasos anteriores se crearon ejemplos en C, los cuales fueron puestos en un archivo `txt`, para la ejecución en una computadora.

Con el fin de comparar los tiempos de ejecución al Algoritmo 3.1, en el código que se presenta cobre el Método de Gradiente Conjugado se le realizan unas pequeñas modificaciones, las cuales constan de lo siguiente:

1. El número de bits de precision no se pedirá, sino que dejara fijo desde el inicio.

2. El número de iteraciones máximo se dejara en n.

3. El ERROR se fijara en 0.0001.

Para el Algoritmo 2.3, se modificara el código anteriormente descrito, esta versión no evitara los cálculos repetitivos que se mencionan anteriormente.

Los ejemplos se implementaron en una laptop con las siguientes especificaciones:

- Marca del procesador: AuthenticAMD.
- Familia del procesador: 22.
- Nombre del procesador: AMD E1-6010 APU with AMD Radeon R2 Graphics.
- Número de núcleos del procesador: 2.
- Memoria del procesador: 3.492316 Gb.
- Sistema operativo: GNU LINUX Ubuntu 17.04.
- Versión de C: 6.3.0 20170406.

Por último, cabe mencionar que este ambas versiones se corrieron 10 veces en cada ejemplo, y tomando el tiempo más pequeño, se obtuvieron los siguientes resultados:

Comparación de Algoritmos para n = 10

Ejemplo	**Tiempo**		**Bits de precisión**	**Norma de $\mathbf{r_n}$**
	Alg. 2.3	**Alg. 3.1**		
1	0.004 seg.	0.004 seg	50	1.14669×10^{-6}
2	0.004 seg.	0.004 seg	80	5.48713×10^{-22}
3	0.004 seg.	0.004 seg	70	2.20587×10^{-18}
4	0.004 seg.	0.004 seg	70	4.43693×10^{-21}
5	0.004 seg.	0.004 seg	70	1.127×10^{-20}

Tabla 3.4: Comparación de los tiempos entre los Algoritmos 2.3 y 3.1 en ejemplos donde $n = 10$.

Para los ejemplos donde $n = 10$, no es observable un cambio en el tiempo de ejecución en el tiempo de ejecución entre los Algoritmos 2.3 y 3.1.

Comparación de Algoritmos para n = 25

Ejemplo	Tiempo		Bits de precisión	Norma de $\mathbf{r_n}$
	Alg. 2.3	Alg. 3.1		
1	0.019 seg.	0.012 seg	200	1.28488×10^{-22}
2	0.019 seg.	0.013 seg	200	1.96642×10^{-23}
3	0.019 seg.	0.013 seg	200	8.04133×10^{-22}
4	0.019 seg.	0.013 seg	200	2.01922×10^{-23}
5	0.018 seg.	0.012 seg	200	9.14591×10^{-20}

Tabla 3.5: Comparación de los tiempos entre los Algoritmos 2.3 y 3.1 en ejemplos donde $n = 25$.

Para los ejemplos donde $n = 25$ se empieza a observar un cambio en el tiempo de ejecución en el tiempo de ejecución entre los Algoritmos 2.3 y 3.1 de milésimas de segundos.

Comparación de Algoritmos para n = 50

Ejemplo	Tiempo		Bits de precisión	Norma de $\mathbf{r_n}$
	Alg. 2.3	Alg. 3.1		
1	0.132 seg.	0.078 seg	350	1.81227×10^{-6}
2	0.154 seg.	0.090 seg	400	3.33214×10^{-12}
3	0.158 seg.	0.089 seg	400	8.27725×10^{-9}
4	0.158 seg.	0.091 seg	400	7.53353×10^{-19}
5	0.158 seg.	0.090 seg	400	$3.56639 \times 10^{--9}$

Tabla 3.6: Comparación de los tiempos entre los Algoritmos 2.3 y 3.1 en ejemplos donde $n = 50$.

Para los ejemplos donde $n = 50$ se observa un cambio en el tiempo de ejecución entre los Algoritmos 2.3 y 3.1 de décimas de segundo.

3.4. Código en C

Para programar el Algoritmo 3.1 en el lenguaje C, se usa memoria dinámica. El código principal es el siguiente:

```
/* Implementación del Método de Gradiente Conjugado Parcial
 *
 *
```

```
 * Juan Antonio Vázquez Morales
 * 8 de enero de 2018
 *
 * Las siguientes lineas de código se crean con el propósito de implementar
 * el Algoritmo de Gradiente Conjugado Parcial, y para mejorar la precisión
 * se utiliza la librería GMP, versión 6.1.2.
 *
 * Para mas información de la librería GMP visitar https://gmplib.org/.
 */

/*------------------------------Librerías básicas de C------------------------------*/
#include "stdio.h"
#include "stdlib.h"

/* La librería GMP permite hacer cálculos mas exactos, lo cual es importante
 * para problemas donde n es grande.
 */
#include "gmp.h"

/*----------Librerías creadas para la implementación del Algoritmo----------*/

/* La librería FuncionesMemoriaYArchivosMGC.h contiene funciones para el
 * manejo de la memoria dinámica y los archivos, adaptadas para el manejo de
 * la librería GMP.
 */
#include "MemoriaYArchivosMGC.h"

/* La librería FuncionesMatricesMGC.h contiene funciones para operaciones
 * relacionadas con operaciones de matrices y asignación de variables de
 * acuerdo al Método de Gradiente Conjugado Parcial, adaptadas a la librería
 * GMP.
 */
#include "MatricesMGC.h"

/*----------------Comienza las lineas de código para el MGC----------------*/
int main(int argc, char **argv){
        /* Variable que guarda -1.0*/
        mpf_t Menos1;
        mpf_init(Menos1);
        mpf_set_d(Menos1,-1.0);

        /* Definición de variables */
        /* Variables enteras */
        int n,m,k,i;
        /* Matrices y vectores */
        mpf_t **A,*b,*x,*r,*p,*z;
        /* Variables reales */
        mpf_t ERROR,alpha,beta,gamma,rho,aux;

        /* Precisamos el numero de bits de precisión */
        printf("Dar bits de precisión: ");
        scanf("%d",&n);
        mpf_set_default_prec(n);

        /*Se inician variables*/
        mpf_init(ERROR);
        mpf_init(alpha);
```

```
mpf_init(beta);
mpf_init(gamma);
mpf_init(rho);
mpf_init(aux);

/* Se crea un apuntador para el archivo donde se almacena los datos
 * del sistema
 */
FILE *Archivo;

/* Se abre el archivo que contiene el sistema lineal */
Archivo=fopen(argv[1],"r");

/* Se verifica que el archivo este abierto */
if(Archivo!=NULL){

        /* Se lee la dimensión del sistema */
        fscanf(Archivo,"%d",&n);

        /* Memoria para la matriz A de dimensión nxn */
        A=NuevaMatriz(n);

        /* Memoria para los vectores */
        x=NuevoVector(n);
        b=NuevoVector(n);
        r=NuevoVector(n);
        p=NuevoVector(n);
        z=NuevoVector(n);

        /* Se verifica que los vectores se hallan creado */
        if(A!=NULL && b!=NULL && x!=NULL &&
         r!=NULL && p!=NULL && z!=NULL){
                /* Se inicia la matriz  */
                IniciarMatriz(A,n);

                /* Se inician vectores */
                IniciarVector(x,n);
                IniciarVector(b,n);
                IniciarVector(r,n);
                IniciarVector(p,n);
                IniciarVector(z,n);

                /* Se comienza por leer los datos */
                CargarSistema(Archivo,n,A,b,x);

                /*Se pide el número de iteraciones máximo */
                do{
                        printf("Máximo de iteraciones 0<m<%d? ",n);
                        scanf("%d",&m);
                }while(m<=0 || m>n-1);

                /* Se pide el ERROR */
                do{
                        printf("ERROR>0? ");
                        gmp_scanf("%Ff",&ERROR);
                }while(ERROR<=0);
```

```
/* Se inicia el Método de Gradiente Conjugado
 * Parcial
 */
/* Datos iniciales */
k=0;

/* Cálculo de r0 */
Calculo_r0(n,A,x,b,r);

/* Cálculo de p0 */
ProductoConstanteVector(n,Menos1,r,p);

/* Cálculo de rho */
ProductoVectores(rho,n,r,r);

mpf_sqrt(aux,rho);

/* Comienzan las iteraciones */
/* */
while(k<m+1 && mpf_cmp(aux,ERROR)>=0){
        /* Cálculo de z */
        ProdMatrizVector(n,A,p,z);

        /* Cálculo de alphak */
        ProductoVectores(aux,n,p,z);

        mpf_div(alpha,rho,aux);

        /* Cálculo de xk */
        Calculo_dk(n,alpha,p,x);

        /* Cálculo de rk */
        Calculo_dk(n,alpha,z,r);

        /* Cálculo de gamma */
        mpf_set(gamma,rho);

        /* Cálculo de rho */
        ProductoVectores(rho,n,r,r);

        /* Cálculo de betak*/
        mpf_div(beta,rho,gamma);

        /* Cálculo de pk */
        Calculo_pk(n,beta,r,p);

        /* Se incrementa k */
        k=k+1;

        /* Se actualiza aux */
        mpf_sqrt(aux,rho);
}

/* Se imprime solución */
printf("Iteraciones: %d\n",k);
for(i=0;i<n;i++) gmp_printf("\tx%d= %Fg\n",i+1,x[i]);
gmp_printf("Norma del último residuo: %Fg\n",aux);
```

```
                }
                else printf("Memoria insuficiente");

                /* Se libera memoria */
                /* Se libera la matriz A si es necesario */
                if(A!=NULL) LiberarMatriz(A,n);
                /* Se libera los vectores si es necesario */
                if(x!=NULL) free(x);
                if(b!=NULL) free(b);
                if(r!=NULL) free(r);
                if(p!=NULL) free(p);
                if(z!=NULL) free(z);

                /* Se cierra el archivo */
                fclose(Archivo);
        }
        else gmp_printf("El archivo no fue abierto");
}
```

Como se puede observar en el código se crearon dos librerías que contienen diversas funciones útiles para su implementación, `MemoriaYArchivosMGC.h` y `MatricesMGC.h`.

Librería MemoriaYArchivosMGC.h

```
/* Librería MemoriaYArchivosGMP.h
 *
 * Juan Antonio Vázquez Morales.
 * 8 de enero de 2018.
 *
 * Funciones diversas para manipulación de archivos y reservar memoria de
 * matrices, las cuales se adaptaron para ser compatibles con la librería
 * GMP.
 *
 * Todas estas funciones fueron creadas para matrices cuadradas, es decir, la
 * matriz A es de tamaño nxn.
 */


/*----------------Declaración de todas las funciones o procesos----------------*/
/*Funciones para archivos*/
void CargarSistema(FILE *arch,int n,mpf_t **A,mpf_t b[],mpf_t x0[]);

/*Funciones para memoria*/
mpf_t *NuevoVector(int n);
mpf_t **NuevaMatriz(int n);
void LiberarMatriz(mpf_t **a,int m);
void IniciarMatriz(mpf_t **A, int n);
void IniciarVector(mpf_t *v,int n);


/*--------------------Funciones o Procesos para archivos--------------------*/
```

```
void CargarSistema(FILE *arch,int n,mpf_t **A,mpf_t b[],mpf_t x0[]){
        /* Proceso para leer los datos del sistema Ax = b, y el punto x0.
         *
         * Parámetros:
         * arch, apuntador al archivo.
         * n, número de variables.
         * A, matriz A.
         * b, vector b.
         * x0, punto inicial.
         */

        /* El formato del archivo debe contener de preferencia la
         * siguiente estructura:
         * 1.- La primera línea es un entero, el valor de n seguido de un
         *     salto de línea.
         * 2.- Se tiene la matriz [A|b], cada elemento cada elemento de
         *     cada fila es separado por un tabulador, y cada fila por un
         *     salto de línea, incluso la última fila.
         * 3.- El punto inicial x0 es de tal forma que cada elemento del
         *     vector esta separado por un salto de línea.
         */

        /* Declaración de variables para el ciclo de lectura */
        int i,j;

        /* Se lee [A|b] */
        for(i=0;i<n;i++){
                /* Se lee la i-ésima fila de Matriz A */
                for(j=0;j<n;j++) gmp_fscanf(arch,"%Ff",&A[i][j]);
                /* Se lee la i-ésima entrada del vector b */
                gmp_fscanf(arch,"%Ff",&b[i]);
        }
        /* Lectura del punto inicial */
        for(i=0;i<n;i++) gmp_fscanf(arch,"%Ff",&x0[i]);
}

/*---------------------- -Funciones y procesos para Memoria---- ----------------------*/

mpf_t *NuevoVector(int n){
        /* Función para crear un vector de tamaño n.
         *
         * Parámetros:
         * n, tamaño del vector.
         */

        /* Se crea apuntador del vector */
        mpf_t *y;

        /* Se asigna la memoria para el vector */
        y=(mpf_t *)malloc(n*sizeof(mpf_t));

        /* Se regresa la dirección de la memoria */
        return y;
}


mpf_t **NuevaMatriz(int n){
```

```
        /* Función que crea una matriz cuadrada
         *
         * Parámetros:
         * n, tamaño de la matriz.
         */

        /* Apuntador para la matriz */
        mpf_t **p;

        /* Declaración de variable para el ciclo */
        int i;

        /* Memoria que guarda la dirección de cada fila */
        p=(mpf_t **)malloc(n*sizeof(mpf_t *));

        /* Se verifica si el vector de direcciones fue creado */
        if(p!=NULL){
                /* Se procede a crear cada fila */
                for(i=0;i<n;i++){
                        /* Memoria para cada fila */
                        p[i]=(mpf_t *)malloc(n*sizeof(mpf_t));

                        /* Se verifica si se crea la fila, en caso contrario
                         * se rompe el ciclo.
                         */
                        if(p[i]==NULL) break;
                }

                /* Se verifica la creación de todas las filas */
                if(i!=n){
                        printf("No se creo matriz\n");
                        /* Se librera la memoria si se rompe el ciclo */
                        LiberarMatriz(p,i);
                }
        }

        /* Se devuelve la dirección de la matriz */
        return p;
}


void LiberarMatriz(mpf_t **a,int m){
        /* Proceso que libera la memoria de una matriz
         *
         * Parámetros:
         * a, apuntador de la memoria de la matriz.
         * m, número de filas que necesitan ser liberadas.
         */

        /* Declaración de variable para el ciclo */
        int i;

        /* Ciclo que libera las filas necesarias */
        for(i=0;i<m;i++) free(a[i]);

        /* Se libera el vector de direcciones de las filas */
        free(a);
```

```
        /* Se retorna un valor nulo */
        a=NULL;
}

void IniciarMatriz(mpf_t **A, int n){
        /* Proceso que inicia variables de una matriz.
         *
         * Parámetros:
         * A, apuntador de la memoria de la matriz.
         * n, número de filas y columnas de la matriz.
         */

        /* Declaración de variables para el ciclo */
        int i,j;

        /* Se inician entradas de la matriz A */
        for(i=0;i<n;i++){
                for(j=0;j<n;j++) mpf_init(A[i][j]);
        }
}

void IniciarVector(mpf_t *v,int n){
        /* Proceso que inicia variables de un un vector.
         *
         * Parámetros:
         * v, apuntador de la memoria del vector.
         * n, número de elementos de v.
         */

        /* Declaración de variable para el ciclo */
        int i;

        /* Se inician entradas del vector A */
        for(i=0;i<n;i++) mpf_init(v[i]);
}
```

Librería MatricesMGC.h

```
/* Librería MatricesMGC.h
 *
 * Juan Antonio Vázquez Morales
 * 8 de enero de 2018
 *
 *
 * Funciones diversas de las operaciones con matrices necesarias en el Método
 * de Gradiente Conjugado, las cuales fueron adaptadas para ser compatibles
 * con la librería GMP.
 *
 * Las funciones y/o procesos que involucran una matriz requieren que esta
 * sea cuadrada.
 */
```

```
/*----------Declaración de todas las funciones o procesos----------*/
void ProdMatrizVector(int n,mpf_t **Q,mpf_t x[],mpf_t res[]);
void ProductoVectores(mpf_t res,int n,mpf_t x[],mpf_t y[]);
void SumaVectores(int n,mpf_t x[],mpf_t y[],mpf_t res[]);
void RestaVectores(int n,mpf_t x[],mpf_t y[],mpf_t res[]);
void ProductoConstanteVector(int n,const mpf_t c,mpf_t x[],mpf_t res[]);
void Calculo_r0(int n,mpf_t **A,mpf_t x[],mpf_t b[],mpf_t r[]);
void Calculo_dk(int n,const mpf_t c,mpf_t v[],mpf_t d[]);
void Calculo_pk(int n,const mpf_t beta,mpf_t r[],mpf_t p[]);


/*----------Funciones y procedimientos----------*/

void ProdMatrizVector(int n,mpf_t **Q,mpf_t x[],mpf_t res[]){
        /* Proceso que obtiene el el resultado de multiplicar Qx
         *
         * Parámetros:
         * n, tamaño de la matriz y vectores.
         * Q, dirección de la matriz Q.
         * x, dirección del vector x.
         * res, dirección del vector resultante.
         */

        /* Declaración de variables para el ciclo */
        int i,j;
        mpf_t aux;

        /* Se inicia la variable aux */
        mpf_init(aux);

        /* Ciclo que obtiene la multiplicación Qx */
        for(i=0;i<n;i++){
                mpf_set_d(res[i],0.0);
                for(j=0;j<n;j++){
                        mpf_mul(aux,Q[i][j],x[j]);
                        mpf_add(res[i],res[i],aux);
                }
        }
        /* Se libera variable */
        mpf_clear(aux);
}


void ProductoVectores(mpf_t res,int n,mpf_t x[],mpf_t y[]){
        /* Función que obtiene <x,y>
         *
         * Parámetros:
         * res, variable resultado.
         * n, tamaño de los vectores.
         * x, dirección del vector x.
         * x, dirección del vector y.
         */

        /* Declaración de la variable donde se guarda el resultado */
        mpf_t aux;

        /* Se inicia la variable aux */
```

```
        mpf_init(aux);

        /* Asignación del valor 0.0 a res */
        mpf_set_d(res,0.0);

        /* Declaración de variable para el ciclo */
        int i;

        /* Ciclo que obtiene <x,y> */
        for(i=0;i<n;i++){
                mpf_mul(aux,x[i],y[i]);
                mpf_add(res,res,aux);
        }

        /* Se libera variable */
        mpf_clear(aux);
}


void SumaVectores(int n,mpf_t x[],mpf_t y[],mpf_t res[]){
        /* Proceso que obtiene x+y.
         *
         * Parámetros:
         * n, tamaño de la vectores.
         * x, dirección del vector x.
         * y, dirección del vector y.
         * res, dirección del vector resultante.
         */

        /* Declaración de variable para el ciclo */
        int i=0;

        /* Ciclo que obtiene la suma */
        for(i=0;i<n;i++) mpf_add(res[i],x[i],y[i]);

        /* Nota: En esta función si x se actualiza mediante está operación,
         * res puede tener la dirección de x, lo cual no afecta el
         * resultado.
         */

}


void RestaVectores(int n,mpf_t x[],mpf_t y[],mpf_t res[]){
        /* Proceso que obtiene x-y.
         *
         * Parámetros:
         * n, tamaño de la vectores.
         * x, dirección del vector x.
         * y, dirección del vector y.
         * res, dirección del vector resultante.
         */

        /* Declaración de variable para el ciclo */
        int i=0;

        /* Ciclo que obtiene la diferencia */
```

```
        for(i=0;i<n;i++) mpf_sub(res[i],x[i],y[i]);

        /* Nota: En esta función si x se actualiza mediante está operación,
         * res puede tener la dirección de x, lo cual no afecta el
         * resultado.
         */
}


void ProductoConstanteVector(int n,const mpf_t c,mpf_t x[],mpf_t res[]){
        /* Proceso que obtiene el producto de un vector por un escalar.
         *
         * Parámetros:
         * n, tamaño del vector.
         * c, constante real por la que se multiplica el vector.
         * x, dirección del vector x.
         * res, dirección del vector resultante.
         */

        /* Declaración de variable para el ciclo */
        int i;

        /* Ciclo que obtiene c*y */
        for(i=0;i<n;i++) mpf_mul(res[i],c,x[i]);

        /* Nota: En esta función si x se actualiza mediante está operación,
         * res puede tener la dirección de x, lo cual no afecta el
         * resultado.
         */
}


void Calculo_r0(int n, mpf_t **A, mpf_t x[],mpf_t b[],mpf_t r[]){
        /* Proceso que calcula el residuo inicial r0=Ax-b.
         *
         * Parámetros:
         * n, tamaño de la vectores.
         * A, dirección de la matriz.
         * x, dirección del vector x.
         * b, dirección del vector b.
         * r, dirección del vector resultante.
         */

        /* Se calcula Ax */
        ProdMatrizVector(n,A,x,r);

        /* Se procede a calcular Ax-b */
        RestaVectores(n,r,b,r);
}


void Calculo_dk(int n,const mpf_t c,mpf_t v[],mpf_t d[]){
        /* Proceso que actualiza vectores de la forma d=d+cv.
         *
         * Parámetros:
         * n, tamaño de la vectores.
         * c, longitud de paso.
```

```
         * v, dirección del vector v.
         * d, dirección del vector d.
         */

        /* Declaración de variable para el ciclo */
        int i;
        mpf_t aux;

        /* Se inicia la variable aux */
        mpf_init(aux);

        /* Ciclo que obtiene d */
        for(i=0;i<n;i++){
                mpf_mul(aux,c,v[i]);
                mpf_add(d[i],d[i],aux);
        }
        /* Se libera variable */
        mpf_clear (aux);
}

/*Calculo de pk*/
void Calculo_pk(int n,const mpf_t beta,mpf_t r[],mpf_t p[]){
        /* Proceso que actualiza vectores de la forma p=-r+cp.
         *
         * Parámetros:
         * n, tamaño de la vectores.
         * beta, constante del MGC.
         * r, dirección del vector r.
         * p, dirección del vector p.
         */

        /* Declaración de variable para el ciclo */
        int i;
        mpf_t aux;

        /* Se inicia la variable aux */
        mpf_init(aux);

        /* Ciclo que obtiene pk del MGC */
        for(i=0;i<n;i++){
                mpf_mul(aux,beta,p[i]);
                mpf_sub(p[i],aux,r[i]);
        }
        /* Se libera variable */
        mpf_clear (aux);
        /* Nota: Este proceso actualiza p */
}
```

Forma del Archivo

En la librería `MemoriaYArchivosMGC.h` se propone una forma en la que el archivo debe estar escrito. A continuación se muestra un ejemplo.

```
3
```

```
11        −3        3         1143660
−3        29        −9        −1181880
3         −9        19        2725380
1237
1597
1003
```

En el ejemplo, se observa que $n = 3$ y que demás elementos del sistema $Ax = b$ están dados por:

$$A = \begin{bmatrix} 11 & -3 & 3 \\ -3 & 29 & -9 \\ 3 & -9 & 19 \end{bmatrix},$$

$$b = \begin{bmatrix} 1143660 \\ -1181880 \\ 2725380 \end{bmatrix}$$

y el punto inicial es

$$x_0 = \begin{bmatrix} 1237 \\ 1597 \\ 1003 \end{bmatrix}$$

Aunque el archivo no es necesario que los datos presenten la forma propuesta, si deben aparecer en ese orden.

Capítulo 4

Conclusiones

El objetivo del trabajo fue desarrollar la teoría del método de gradiente conjugado de tal manera que cualquier persona interesada en el tema pueda comprender la información sin dificultad. Además, a partir de un previo conocimiento del Algoritmo 2.3, se había pensado en una pequeña modificación, la cual se demostró que era mejor en cuestión del número de operaciones por iteración, y no solo esto, se planeó utilizar esa idea para realizar un código en C que cualquier persona podría utilizar o incluso modificar a sus necesidades.

1. Después de la comprensión de la teoría, se comprobó que la modificación no afectaba las propiedades del método, por lo que era posible implementar la modificación que se planeaba en cada iteración. Con un mejor conocimiento de las cotas de convergencia y conocimiento de la programación, se mejoro esta idea para proponer finalmente el Algoritmo 3.1, la cual no solo es fiel a la idea de las operaciones a realizar por iteración, sino que se agregó el número de iteraciones máximo que se dehesen realizar.

2. La modificación propuesta se comprobó que fue un éxito al momento de comparar el número de operaciones por iteración, la cual fue posible con solo agregar un producto interno al inicializar el Algoritmo 3.1, esta no representa un incremento muy alto, ya por iteración la modificación evita calcular 2 productos escalares, con lo que esta es una mejora, afirmando la hipótesis.

3. La mayor dificultad que se encontró en el desarrollo del trabajo, es que las demostraciones de la teoría no eran muy desarrolladas, por lo

que fue complicado explicar de manera detallada estos resultados, y en ocasiones los resultados no eran demostrados. Encontrar la literatura donde estos eran expuestos, o el querer demostrarlos por cuenta propia fue una tarea exhaustiva por la falta de experiencia, pero el esfuerzo dio sus frutos y se expusieron todos los resultados deseados.

4. Con este trabajo, se aprendió que a pesar que diversos resultados ya fueron demostrados, su difusión no es muy amplia, y es complicado encontrarlos si uno no sabe bien el ¿cómo? o el ¿dónde? buscar. Esto no fue razón que desmotivara la investigación, fue un motivo más para querer desarrollar la teoría.

   Por otro lado, la respuesta a la pregunta formulada al Dr. Calvillo sobre la rapidez de convergencia del método de gradiente conjugado (ver cotas de convergencia en la Sección 2.2), es más adecuada la siguiente respuesta.

   "Hablar de la rapidez de convergencia del método de gradiente conjugado no es muy adecuado, lo que tiene sentido es hablar sobre las cotas de convergencia del método. Sabiendo algunas características de la matriz A, podemos decidir el número de iteraciones máximo a realizar para obtener una buena aproximación de la solución, pues en problemas de grandes dimensiones, no resulta muy práctico realizar las n iteraciones."

5. Este trabajo, aporta una mejor comprensión del método de gradiente conjugado, para que cualquier individuo que requiera su uso, pueda decidir si es adecuado o no a sus necesidades, y si quiere la validez todos los resultados expuestos, lo pueda comprender sin dificultad. Se necesitan conceptos básicos de álgebra, análisis convexo y cálculo diferencial, que cualquier persona que tenga relación con la matemática conoce, y si no fuera este el caso, se agrega un apéndice con los conceptos y resultados necesarios. Adicionalmente, el código que se creó referente al método, está documentado de tal manera que pueda ser comprendido y si se requiere modificado de manera sencilla para adecuarse a las diversas necesidades.

6. Aunque este trabajo puede ser considerado elemental para investigadores del área de métodos y análisis numérico, he incluso de optimización,

es desconocido por la mayor parte de los estudiantes de licenciatura, por lo que consideró que la investigación realizada es una buena aportación, en especial la parte de la programación. A lo largo de mi formación académica, he encontrado a diversos compañeros que no les agrada utilizar una computadora, o incluso, a crear sus propios programas para resolver problemas que les absorbe una considerable cantidad de tiempo, y la razón de esta actitud es una mala comprensión de la estructura de un código o algoritmo.

7. Para concluir, es claro que faltan cosas por realizar, por ejemplo, adaptar este algoritmo a sistemas de ecuaciones no lineales, a problemas de optimización sin restricciones donde la función objetivo no es cuadrática, y mejorar el uso de memoria dinámica en el código. Estás problemáticas se salen de los objetivos de este trabajo, pero no se descarta la idea de seguir trabajando en un futuro con el tema.

Apéndice A

Conceptos y Resultados básicos

A continuación se presentan algunos conceptos y resultados necesarios para la mejor comprensión de este trabajo.

A.1. Matrices y Vectores

Definición A.1. *El conjunto $M_{n\times m}(\mathbb{R})$ denota el conjunto de todas las matrices de tamaño $n \times m$, es decir, si $Q \in M_{n\times m}(\mathbb{R})$, entonces*

$$Q = \begin{bmatrix} q_{11} & q_{12} & \cdots & q_{1m} \\ q_{21} & q_{22} & \cdots & q_{2m} \\ \cdot & \cdot & \cdots & \cdot \\ q_{n1} & q_{n2} & \cdots & q_{nm} \end{bmatrix}$$

donde $q_{ij} \in \mathbb{R}$, para $i = 1, \ldots, n$ y $j = 1, \ldots, m$, y se puede denotar a las entradas por

$$Q = [q_{ij}].$$

Definición A.2. *Un vector en $\mathbb{R}^n$ es una matriz v en $M_{n\times 1}(\mathbb{R})$, y se le denomina* ***vector columna*** *o simplemente* ***vector****.*

Combinaciones Lineales

Definición A.3. *Un vector v es una* ***combinación lineal*** *de vectores $v_1, v_2, \ldots, v_k$ si existen escalares $c_1, c_2, \ldots, c_k$ tales que $v = c_1v_1 + c_2v_k + \cdots + c_kv_k$. Los escalares $c_1, c_2, \ldots, c_k$ se llaman* ***coeficientes de la combinación lineal****.*

Definición A.4. *Un conjunto de vectores $v_1, v_2, \ldots, v_k$ es* ***linealmente dependiente*** *si existen escalares $c_1, c_2, \ldots, c_k$, donde al menos uno no es cero, tales que*

$$c_1 v_1 + c_2 v_2 + \cdots + c_k v_k = 0.$$

Un conjunto de vectores que no es linealmente dependiente se llama ***linealmente independiente****.*

Definición A.5. *El gen$\{v_0, v_1, \ldots, v_k\}$ denota el conjunto de todas las combinaciones lineales de los vectores $v_0, v_1, \ldots, v_k$.*

Definición A.6. *Si V es el espacio vectorial asociado a E, se dice que F es* ***variedad lineal*** *si existen un subespacio vectorial S de V y un $a \in E$ de manera que $F = a + S = \{a + b : b \in S\}$.*

Propiedades de las matrices

Definición A.7. *La* ***transpuesta*** *de una matriz $A = [a_{ij}] \in M_{n\times m}(\mathbb{R})$ es la matriz $A^T = [a_{ij}^T] \in M_{m\times n}(\mathbb{R})$ donde $a_{ij}^T = a_{ji}$ para $i = 1, \ldots, n$ y $j = 1, \ldots, m$.*

Definición A.8. *Una matriz $A = [a_{ij}] \in M_{n\times n}(\mathbb{R})$ se dice* ***simétrica*** *si la entrada a_{ij} es igual a a_{ji} para todo $i, j = 1, \ldots, n$.*

Definición A.9. *Un matriz $A \in M_{n\times n}(\mathbb{R})$ se dice que es* ***semidefinida positiva*** *si para todo $d \in \mathbb{R}^n \setminus \{0\}$, se tiene que*

$$d^T A d \geq 0.$$

Si la desigualdad es estricta, entonces A se dice que es ***definida positiva****.*

Definición A.10. *Un conjunto de vectores no nulos $\{p_0, p_1, \ldots, p_l\}$ se dice que es conjugado con respecto a la matriz simétrica definida positiva A si*

$$p_i^T A p_j = 0 \quad \textit{para todo } i \neq j. \tag{A.1}$$

Nota: Si A es la matriz identidad I de tamaño $n \times n$, entonces la definición anterior es la de un **conjunto ortogonal** sin hacer referencia a la matriz I, pues se trata del producto euclidiano usual. Además, en este caso, si los vectores son unitarios, se dice que $\{p_0, p_1, \ldots, p_l\}$ es un **conjunto ortonormal**.

Proposición A.1. *Un conjunto de vectores no nulos* $\{p_0, p_1, \ldots, p_l\}$ *que cumplen (A.1) es un conjunto linealmente independiente.*

Demostración. Se supone $\gamma_0, \gamma_1, \ldots, \gamma_l \in \mathbb{R}$ tales que

$$\sum_{i=0}^{l} \gamma_i p_i = 0,$$

entonces,

$$\begin{aligned} 0 &= \left(\sum_{i=0}^{l} \gamma_i p_i \right)^T A \left(\sum_{j=0}^{l} \gamma_j p_j \right) \\ &= \sum_{i=0}^{l} \sum_{j=0}^{l} \gamma_i \gamma_j p_i^T A p_j, \end{aligned}$$

dado que los vectores cumplen la propiedad de conjugación, lo anterior se escribe como

$$0 = \sum_{i=0}^{l} \gamma_i^2 p_i^T A p_i,$$

y dado que la matriz A es definida positiva, se tiene que $\gamma_i = 0$ para todo $i = 0, 1, \ldots, l$, concluyendo el resultado. □

Valores y Vectores Propios

Definición A.11. *Un* ***vector propio*** *de una matriz* $A \in M_{n\times n}(\mathbb{R})$ *es un vector* $v \in \mathbb{R}^n$ *no nulo tal que*

$$Av = \lambda v,$$

para algún $\lambda \in \mathbb{R}$*. Al número* λ *se le denomina el* ***valor propio*** *asociado a* v*.*

Teorema A.1. *Si* $A \in M_{n\times n}(\mathbb{R})$ *es simétrica, entonces los valores propios de A son reales.*

Demostración. Ver [8]. □

Definición A.12. *Una matriz $Q \in M_{n\times m}(\mathbb{R})$ cuyas columnas forman un conjunto ortonormal se llama matriz ortogonal.*

Definición A.13. *Una matriz $A \in M_{n\times n}(\mathbb{R})$ es* ***diagonalizable ortogonalmente*** *si existe una matriz ortogonal $Q \in M_{n\times n}(\mathbb{R})$ y una matriz diagonal D tales que $Q^T AQ = D$.*

Teorema A.2 (El teorema espectral)**.** *Sea $A \in M_{n\times n}(\mathbb{R})$. Entonces A es simétrica si y sólo si es diagonalizable ortogonalmente.*

Demostración. Ver [8]. □

El teorema espectral permite escribir una matriz simétrica real A en la forma $A = QDQ^T$, donde Q es ortogonal y D es diagonal. Las entradas diagonales de D son justo los valores propios de A, y si las columnas de Q son los vectores ortonormales $q_1, \ldots, q_n$, entonces, al usar la representación columna-fila del producto, se tiene

$$\begin{aligned} A = QDQ^T &= \begin{bmatrix} q_1 & \cdots & q_n \end{bmatrix} \begin{bmatrix} \lambda_1 & \cdots & 0 \\ \cdot & \cdots & \cdot \\ 0 & \cdots & \lambda_n \end{bmatrix} \begin{bmatrix} q_1^T \\ \vdots \\ q_n^T \end{bmatrix} \\ &= \begin{bmatrix} \lambda_1 q_1 & \cdots & \lambda_n q_n \end{bmatrix} \begin{bmatrix} q_1^T \\ \vdots \\ q_n^T \end{bmatrix} \\ &= \sum_{i=1}^{n} \lambda_i q_i q_i^T \end{aligned}$$

A esto se le conoce como la **descomposición espectral** de A.

Teorema A.3. *Una matriz $A \in M_{n\times n}(\mathbb{R})$ es definida positiva si y solo si sus valores propios son todos positivos.*

Demostración. Primero se supone que A es definida positiva, y supongamos $v \in \mathbb{R}^n$ vector propio de A con valor propio λ, además a v lo podemos suponer unitario, entonces

$$0 < v^T Av = v^T(\lambda v) = \lambda(v^T v) = \lambda,$$

por lo que los valores propios son positivos.

Ahora se supone que los valores propios de A $\lambda_1, \ldots, \lambda_n > 0$, como se puede formar una base ortonormal de vectores propios (ver [8]) $\{v_1, \ldots, v_n\}$ con $Av_i = \lambda_i v_i$, por lo que cualquier $x \in \mathbb{R}^n$ no nulo, se tiene

$$x = \sum_{i=1}^{n} c_i v_i,$$

para algunos $c_i \in \mathbb{R}$, así

$$\begin{aligned} x^T Ax &= \left(\sum_{i=1}^{n} c_i v_i\right)^T A \left(\sum_{j=1}^{n} c_j v_j\right) \\ &= \left(\sum_{i=1}^{n} c_i v_i\right)^T \left(\sum_{j=1}^{n} c_j \lambda_j v_j\right) \\ &= \sum_{i=1}^{n}\sum_{j=1}^{n} \lambda_j c_i c_j (v_i^T v_j) \\ &= \sum_{i=1}^{n} \lambda_i c_i^2 (v_i^T v_i) \\ &= \sum_{i=1}^{n} \lambda_i c_i^2 > 0, \end{aligned}$$

pues existe algún $c_i > 0$, pues x es no nulo, entonces A es definido positivo. □

A.2. Derivación

Definición A.14. *Sea $f : S \subseteq \mathbb{R}^n \to \mathbb{R}$ y $x \in S$, se define el* ***gradiente*** *de f en $x = (x^1, \ldots, x^n)^T$ (x^i es la i-ésima coordenada del vector x) como:*

$$\nabla f(x) = \begin{bmatrix} \frac{\partial f(x)}{\partial x^1} \\ \vdots \\ \frac{\partial f(x)}{\partial x^n} \end{bmatrix}.$$

Definición A.15. *Sea $f : S \subseteq \mathbb{R}^n \to \mathbb{R}$ y $x \in S$, se define el* ***hessiano*** *o* ***matriz hessiana*** *de f en $x = (x^1, \ldots, x^n)$ como:*

$$\nabla^2 f(x) = \begin{bmatrix} \frac{\partial^2 f(x)}{\partial x^1 \partial x^1} & \frac{\partial^2 f(x)}{\partial x^1 \partial x^2} & \cdots & \frac{\partial^2 f(x)}{\partial x^1 \partial x^n} \\ \frac{\partial^2 f(x)}{\partial x^2 \partial x^1} & \frac{\partial^2 f(x)}{\partial x^2 \partial x^2} & \cdots & \frac{\partial^2 f(x)}{\partial x^2 \partial x^n} \\ \cdot & \cdot & \cdots & \cdot \\ \frac{\partial^2 f(x)}{\partial x^n \partial x^1} & \frac{\partial^2 f(x)}{\partial x^n \partial x^2} & \cdots & \frac{\partial^2 f(x)}{\partial x^n \partial x^n} \end{bmatrix}.$$

Derivación de una función cuadrática

Teorema A.4. *Sea $f : \mathbb{R}^n \to \mathbb{R}$ definida por*

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

donde $b = (b^1, b^2, \ldots, b^n)^T \in \mathbb{R}^n$ y $Q \in M_{n\times n}(\mathbb{R})$ con Q simétrica, entonces

$$\nabla f(x) = Qx - b$$

y

$$\nabla^2 f(x) = Q$$

Demostración. Primero se analiza $b^T x$. Se considera $x = (x^1, x^2, \ldots, x_n)^T$, así

$$b^T x = \sum_{i=1}^{n} b^i x^i$$
$$\Rightarrow \frac{\partial b^T x}{\partial x^h} = b^h, \quad h = 1, \ldots, n.$$

Así $\nabla b^T x = b$. Por otro lado, es claro que $\nabla^2 b^T x = 0$.

Ahora se procede a analizar $x^T Q x$. Suponiendo $Q = [q_{ij}]$, donde $q_{ij} = q_{ji}$

para $i, j = 1, \ldots, n$, se tiene

$$x^T Qx = \sum_{i=1}^{n}\sum_{j=1}^{n} q_{ij}x^i x^j$$
$$\Rightarrow \frac{\partial x^T Qx}{\partial x^h} = \sum_{j=1}^{n} q_{hj}x^j + \sum_{i=1}^{n} q_{ih}x^i$$
$$\Rightarrow \frac{\partial x^T Qx}{\partial x^h} = 2\sum_{i=1}^{n} q_{ih}x^i \quad h = 1, \ldots, n.$$

Así $\nabla x^T Qx = 2Qx$.

Además,

$$\frac{\partial x^T Qx}{\partial x^h} = 2\sum_{i=1}^{n} q_{ih}x^i \quad h = 1, \ldots, n$$
$$\Rightarrow \frac{\partial^2 x^T Qx}{\partial x^k \partial x^h} = 2q_{kh} \quad k, h = 1, \ldots, n.$$

Ahora se concluye que:

$$\nabla f(x) = Qx - b$$

y

$$\nabla^2 f(x) = Q.$$

□

A.3. Conjuntos Convexos

Definición A.16. *Un conjunto $S \in \mathbb{R}^n$ es **convexo si** para toda $x_1, x_2 \in S$ y todo $\alpha \in [0, 1]$, el punto $\alpha x_1 + (1 - \alpha)x_2 \in S$.*

.

Proposición A.2. *Los conjuntos convexos de $\mathbb{R}^n$ satisfacen las siguientes relaciones:*

1. *Si C es un conjunto convexo en $\mathbb{R}^n$, y $\beta \in \mathbb{R}$, el conjunto*

$$\beta C = \{x : x = \beta c, c \in C\}$$

es convexo.

2. Si C y D son conjuntos convexos, el conjunto

$$C + D = \{x : x = c + d, c \in C, d \in D\}$$

es convexo.

3. La intersección de cualquier familia de conjuntos convexos es convexa.

La demostración de la proposición anterior es sencilla, solo se necesita una buena comprensión de la definición.

Observación. En la Proposición A.2, en el punto *2*, si C es el conjunto unitario $\{c\}$, abusando de la notación,

$$c + D := \{c\} + D.$$

A.4. Mínimos de una Función

Al minimizar una función $f : S \subseteq \mathbb{R}^n \to \mathbb{R}$ existen dos tipos de respuestas: los mínimos locales y mínimos globales.

Definición A.17. *Un punto $x^* \in S$ es un **mínimo relativo** o **mínimo local** de f si existe un $\epsilon > 0$ tal que $f(x) \geq f(x^*)$ para todo $x \in S$ si $|x - x^*| < \epsilon$. Si $f(x) > f(x^*)$ para todo $x \in S \setminus \{x^*\}$ tal que $|x - x^*| < \epsilon$, entonces se dice que x^* es un **mínimo relativo estricto** o **mínimo local estricto** de f en S.*

Definición A.18. *Un punto $x^* \in S \subseteq \mathbb{R}^n$ es un **mínimo global** de f en S si $f(x) \geq f(x^*)$ para todo $x \in S$. Si $f(x) > f(x^*)$ para todo $x \in S \setminus \{x^*\}$, entonces se dice que x^* es un **mínimo global estricto** de f en S.*

Condiciones de primer orden

Definición A.19. *Sea $S \subseteq \mathbb{R}^n$, $d \in \mathbb{R}^n$ es una **dirección factible** si existe $\overline{\alpha} > 0$ tal que $x + \alpha d \in S$ para toda $\alpha \in [0, \overline{\alpha}]$.*

Proposición A.3 (Condiciones necesarias de primer orden)**.** *Sea S un subconjunto de $\mathbb{R}^n$, y $f \in C^1$, una función en S. Si x^* es un mínimo relativo de f en S, entonces para cualquier $d \in \mathbb{R}^n$ que sea una dirección factible en x^*, se tiene que $\nabla f(x^*)^T d \geq 0$.*

Demostración. Ver [6]. □

Corolario A.1 (Caso sin restricciones)**.** *Sea $S \subseteq \mathbb{R}^n$ y se $f \in C^1$ una función en S. Si x^* es un mínimo relativo de f en S y x^* es un punto interior de S, entonces $\nabla f(x^*) = 0$.*

Condiciones de segundo orden

Proposición A.4 (Condiciones necesarias de segundo orden)**.** *Sea $S \subseteq \mathbb{R}^n$, y $f \in C^2$, una función en S. Si x^* es un mínimo relativo de f en S, entonces para cualquier $d \in \mathbb{R}^n$ que sea una dirección factible en x^*, resulta*

1. $\nabla f(x^*)^T d \geq 0$,

2. *si* $\nabla f(x^*)^T d = 0$, *entonces* $d^T \nabla^2 f(x^*) d \geq 0$.

Demostración. Ver [6]. □

Proposición A.5 (Condiciones necesarias de segundo orden: caso sin restricciones)**.** *Sea x^* un puntos interior de S, suponiendo que x^* es un mínimo relativo en S de la función $f \in C^2$. Entonces:*

1. $\nabla f(x^*) = 0$,

2. *para toda* d, $d^T \nabla^2 f(x^*) d \geq 0$.

Proposición A.6 (Condiciones suficientes de segundo orden: caso sin restricciones)**.** *Sea $f \in C^2$ definida en una región, en la que el punto x^* es un punto interior. Además, supóngase que*

1. $\nabla f(x^*) = 0$.

2. $\nabla^2 f(x^*)$ *es definida positiva.*

Entonces, x^* es un punto mínimo relativo de f.

Demostración. Ver [6]. □

A.5. Funciones Convexas

Definición A.20. *Una función $f : S \subset \mathbb{R}^n \to \mathbb{R}$ donde S es un conjunto convexo, se dice que f es* ***convexa*** *si para todo $x_1, x_2 \in S$ y toda $\alpha \in [0, 1]$, se cumple:*

$$f(\alpha x_1 + (1 - \alpha)x_2) \leq \alpha f(x_1) + (1 - \alpha)f(x_2).$$

Si, para toda $\alpha \in (0, 1)$ y $x_1 \neq x_2$, se cumple

$$f(\alpha x_1 + (1 - \alpha)x_2) < \alpha f(x_1) + (1 - \alpha)f(x_2),$$

entonces, se dice que f es ***estrictamente convexa****.*

Caracterización de las funciones convexas

Proposición A.7. *Sea $f \in C^1$ (es decir, f es diferenciable continua). Entonces f es convexa en el conjunto convexo S si y solo si,*

$$f(y) \geq f(x) + \nabla f(x)^T (y - x)$$

para todo $x, y \in S$.

Demostración. Ver [6]. □

Proposición A.8. *Sea $f \in C^2$. Entonces, f es convexa en un conjunto convexo S que contenga un punto interior si y solo si, la matriz hessiana $\nabla^2 f(x)$ es semidefinida positiva en todo S.*

Demostración. Ver [6]. □

Minimización de funciones convexas

Teorema A.5. *Sea $f : S \subset \mathbb{R}^n \to \mathbb{R}$ donde S es un conjunto convexo y f es una función convexa, si x^* es un mínimo relativo, entonces*

1. *x^* es una solución global.*

2. *Si f es estrictamente convexa, entonces la solución x^* es única.*

Demostración. Ver [1]. □

A.6. Polinomios de Chebyshev

Definición A.21. *El polinomio de Chevyshev $T_n(x)$ de primer tipo, esta definido por*

$$T_n(x) = \cos n\theta \tag{A.2}$$

donde n es un entero no negativo, $x = \cos\theta$ y $0 \leq \theta \leq \pi$.

En la definición anterior se afirma que la relación (A.2) es efectivamente un polinomio, lo cual se exhibe en la siguiente proposición, además del hecho que es un polinomio de grado n.

Proposición A.9. *Para todo n no negativo se tiene que*

$$T_n(x) = \frac{1}{2}\left[\left(x + \sqrt{x^2-1}\right)^n + \left(x - \sqrt{x^2-1}\right)^n\right].$$

Demostración. Recordando que

$$\cos n\theta = \frac{e^{in\theta} + e^{-in\theta}}{2} = \frac{e^{(i\theta)^n} + e^{(i\theta)^{-n}}}{2},$$

y con la fórmula de Moivre:

$$e^{i\theta} = \cos\theta + i\,\mathrm{sen}\,\theta = \cos\theta + i\sqrt{1-\cos^2\theta} = \cos\theta + \sqrt{\cos^2\theta - 1},$$

entonces

$$\cos n\theta = \frac{\left(\cos\theta + \sqrt{\cos^2\theta - 1}\right)^n + \left(\cos\theta + \sqrt{\cos^2\theta - 1}\right)^{-n}}{2},$$

reemplazando $\cos\theta$ por x,

$$T_n(x) = \frac{\left(x + \sqrt{x^2-1}\right)^n + \left(x + \sqrt{x^2-1}\right)^{-n}}{2},$$

por último veamos que

$$\left(x + \sqrt{x^2-1}\right)\left(x - \sqrt{x^2-1}\right) = x^2 - (x^2-1) = 1,$$

así,

$$T_n(x) = \frac{\left(x + \sqrt{x^2-1}\right)^n + \left(x - \sqrt{x^2-1}\right)^n}{2}.$$

□

Apéndice B

Problema del Cálculo de los Valores Propios

En diversas aplicaciones se trabaja con funciones cuadráticas convexas, o bien se les hacen algunas modificaciones a su hessiano para asegurar la convexidad, por lo que en estos casos no es necesario verificar este hecho.

A continuación se muestra un código en el lenguaje C, y una breve explicación de que consiste para poder decidir si una matriz es definida positiva con ayuda de una aproximación de los valores propios.

Para más detalles sobre los métodos empleados consultar [2].

B.1. Explicación del Código

El código que se mostrará en este apéndice consiste de lo siguiente:

1. Al correr el programa es necesario dos argumentos, el primero es el nombre del archivo donde está la información de la matriz simétrica con entradas reales A, y el segundo contendrá el nombre del archivo donde se guardan los valores propios (aproximados) de la matriz A.

2. Después de que se lee la matriz A, se procede a calcular una matriz simétrica tridiagonal similar a A con el Método de Householder.

3. Después de que se se obtiene la matriz tridiagonal (la cual seguiremos llamando A), con el método QR se procede ha hacer un aproximado de los valores propios, en un número finito de iteraciones dado por el usuario. Los valores propios se escriben en el archivo deseado.

4. Al final, se manda un mensaje en el cual nos dice si la matriz es definida positiva o no, o en su efecto si el número de iteraciones no fue suficiente para dar respuesta.

En este código no se verifica que la matriz A sea simétrica, como tal, las lineas de código referentes al Método de Householder lo toman como un hecho.

Por último, la finalidad de que se cree un archivo con los valores propios calculados, es que cualquier persona que utilice está propuesta, pueda analizar los valores propios según su criterio, ya sea manualmente o incluso utilizando sus propios códigos, ya sea en C o en otro lenguaje de su agrado.

B.2. Código en C

El siguiente código usa memoria dinámica, y las líneas principales son las siguientes:

```
/* Verificación de si una matriz es definida positiva.
 *
 *
 * Juan Antonio Vázquez Morales
 * 20 de noviembre de 2017
 *
 * Las siguientes líneas de código se crean para calcular los valores
 * propios de una matriz combinando el método de Householder y el método QR.
 */

/*------------------------------Librerías básicas de C------------------------------*/
#include "stdio.h"
#include "stdlib.h"
#include "math.h"

/*----------Librerías creadas para la implementación del Algoritmo----------*/

/* La librería MemoriaYArchivosCVP.h contiene funciones para el
 * manejo de la memoria dinámica y los archivos.
 */
#include "MemoriaYArchivosCVP.h"

/* La librería FuncionesMH.h contiene funciones necesarias para el Método de
 * Householder.
 */
#include "FuncionesMH.h"

/* La librería FuncionesMQR.h contiene funciones necesarias para el Método
 * QR.
 */
#include "FuncionesMQR.h"
```

```
/* Se define la variable TOL que tendrá una tolerancia para aproximar a los
 * valores propios.
 */
#define TOL 0.0001

/* Ahora se define una función que obtiene el mínimo de los valores que se
 * encuentran en el archivo creado.
 */
double MinimoValores(FILE *Arch);

/*----------Comienza las lineas del código CVP----------*/
int main(int argc, char **argv){
        /* Definición de variables */
        /* Variables enteras */
        int k,n,m;
        int i,j;
        /* Matrices y vectores */
        double **A,*cs,*r,*s,*v,*u,*z,*x,*y;
        /* Variables reales */
        double q,alpha,rsq,prod,shift,b,c,d,mu1,mu2;

        /* Se crea un apuntador para el archivo donde se almacena los datos
         * del sistema
         */
        FILE *Archivo,*ArchivoNuevo;

        /* Se abre el archivo que contiene la matriz, es el primer argumento
         * que se manda al ejecutar el programa
         */
        Archivo=fopen(argv[1],"r");

        /* Se crea el archivo en el que se guardaran los valores propios que
         * se obtengan calcular
         */
        ArchivoNuevo=fopen(argv[2],"w+");

        /* Se verifica que los archivos estén abiertos */
        if(Archivo!=NULL && ArchivoNuevo!=NULL){

                /* Se lee la dimensión de la matriz */
                fscanf(Archivo,"%d",&n);

                /* Memoria para la matriz A de dimensión nxn */
                A=NuevaMatriz(n);

                /* Memoria para los vectores */
                cs=NuevoVector(n);
                r=NuevoVector(n);
                s=NuevoVector(n);
                v=NuevoVector(n);
                u=NuevoVector(n);
                x=NuevoVector(n);
                y=NuevoVector(n);
                z=NuevoVector(n);
```

```
/* Se verifica la creación de los vectores */
if(A!=NULL && cs!=NULL && r!=NULL && s!=NULL
   && v!=NULL && u!=NULL && x!=NULL && y!=NULL && z!=NULL){

        /* Se comienza por leer los datos */
        CargarA(Archivo,n,A);

        /* Comienza el método de Householder */
        for(k=1;k<=n-2;k++){

                /* Se calcula q */
                q=Calcular_q(n,k,A);

                /* Se obtiene alpha */
                /* Se ve si la entrada correspondiente es
                 * cero
                 */
                if(fabs(A[k][k-1])<=TOL) alpha=-sqrt(q);
                else alpha=-sqrt(q)*A[k][k-1]
                     /fabs(A[k][k-1]);

                /* Se obtiene rsq */
                rsq=(alpha*alpha)-alpha*A[k][k-1];

                /* Obtención de v */
                Calcular_v(n,k,alpha,A,v);

                /* Obtención de u */
                Calcular_u(n,k,rsq,v,A,u);

                /* Obtención de prod */
                prod=Calcular_prod(n,k,v,u);

                /* Obtención de z */
                Calcualr_z(n,k,prod,rsq,u,v,z);

                /* Obtención de A^(k+1) */
                Actualizar_A(n,k,v,z,A);
        }

        /* Se termina el método de Householder */


        /* Empieza el método QR */
        shift=0.0;

        /* Se pide el número de iteraciones máximo */
        do{
                printf("Máximo de iteraciones m>=%d? ",n);
                scanf("%d",&m);
        }while(m<n);

        /* Se empiezan hacer iteraciones sin exceder el
         * número de máximo de iteraciones
         */
        for(k=1;k<m;k++){
```

```
/* Se verifica si se tiene un nuevo valor
 * propio
*/

/* Se ve si tenemos un valor propio y se
 * modifica A
 */

if(fabs(A[n-1][n-2])<=TOL){
        fprintf(ArchivoNuevo,"%df\n",
                A[n-1][n-1]+shift);
        n=n-1;
}
if(fabs(A[1][0])<=TOL){
        fprintf(ArchivoNuevo,"%df\n",
                A[0][0]+shift);
        n=n-1;
        A[0][0]=A[1][1];
        for(j=1;j<n;j++){
                A[j][j]=A[j+1][j+1];
                A[j][j-1]=A[j+1][j];
        }
}

if(n==0) break;
if(n==1){
        fprintf(ArchivoNuevo,"%df\n"
                ,A[0][0]+shift);
        break;
}
/* Se termina la verificación */

/* Se obtiene el shift */
b=-(A[n-2][n-2]+A[n-1][n-1]);
c=A[n-1][n-1]*A[n-2][n-2]
    -(A[n-1][n-2]*A[n-1][n-2]);
d=sqrt(b*b-4.0*c);

/* Cálculo de mu1 y mu2 */
if(b>0){
        mu1=-2.0*c/(b+d);
        mu2=-(b+d)/2.0;
}
else{
        mu1=(d-b)/2.0;
        mu2=2.0*c/(d-b);
}
if(n==2){
        fprintf(ArchivoNuevo,"%df\n"
                ,mu1+shift);
        fprintf(ArchivoNuevo,"%df\n"
                ,mu2+shift);
        break;
}

/* Se obtiene el mu más cercano a
 * A[n-1][n-1]
```

```
                                        */
                                        alpha=Calcular_sigma(A[n-1][n-1],mu1,mu2);

                                        /* shift acumulado */
                                        shift+=alpha;

                                        /* shift realizado guardado en v*/
                                        for(j=0;j<n;j++) v[j]=A[j][j]-alpha;

                                        /* Obtención de la matriz R^k*/
                                        Calcular_Rk(n,cs,r,s,u,v,x,y,z,A);

                                        /* Obtención de A^(k+1) */
                                        Actualizar_AQR(n,cs,u,s,x,z,A);
                                }

                                /* Se verifica si se obtuvieron todos los valores
                                 * propios
                                 */
                                if(n<3){
                                        /* Se calcula el mínimo valor propio */
                                        q=MinimoValores(ArchivoNuevo);

                                        /* Se da respuesta si la matriz A es definida
                                         * positiva
                                         */
                                        if(q>0.0)
                                           printf("\nA es definida positiva\n");
                                        else
                                           printf("\nA no es definida positiva\n");
                                }
                                else printf("\nIteraciones insuficientes \n");
                        }
                        else printf("Memoria insuficiente\n");

                        /* Se libera Memoria */
                        /* Se libera la matriz A si es necesario */
                        if(A!=NULL) LiberarMatriz(A,n);

                        /* Se libera los vectores si es necesario */
                        if(cs!=NULL) free(cs);
                        if(r!=NULL) free(r);
                        if(s!=NULL) free(s);
                        if(v!=NULL) free(v);
                        if(u!=NULL) free(u);
                        if(x!=NULL) free(x);
                        if(y!=NULL) free(y);
                        if(z!=NULL) free(z);

                        /* Se cierran los archivos si es necesario */
                        if(Archivo!=NULL) fclose(Archivo);
                        if(ArchivoNuevo!=NULL) fclose(ArchivoNuevo);
                }
                else printf("El archivo no fue abierto\n");
}
```

```
/*-------------------------------Funciones y Procesos-------------------------------*/
double MinimoValores(FILE *Arch){
        /* Función que obtiene el mínimo de los valores de un archivo.
         *
         * Parámetros:
         * Arch, apuntador al archivo.
         */

        /* Variable a retornar */
        double min,aux;

        /* Se mueve el apuntador al inicio del archivo */
        fseek(Arch,0.0,SEEK_SET);

        /* Se lee el primer termino */
        fscanf(Arch,"%lf",&min);

        /* Se empieza  */
        while(feof(Arch)==0){
                fscanf(Arch,"%lf",&aux);
                if(aux<min) min=aux;
        }
        return min;
}
```

Como se observa, se ocupa también la librería `MemoriaYArchivosCVP.h`, pero ahora se ocupan otras dos librerías, `FuncionesMH.h` y `FuncionesMQR.h`.

Librería MemoriaYArchivosCVP.h

```
/* Librería MemoriaYArchivos.h
 *
 * Juan Antonio Vázquez Morales.
 * 7 de septiembre de 2017.
 *
 * Funciones diversas para manipulación de archivos y reservar memoria de
 * matrices.
 *
 * Todas estas funciones fueron creadas para matrices cuadradas, es decir, la
 * matriz A es de tamaño nxn.
 */


/*----------------Declaración de todas las funciones o procesos----------------*/
/*Funciones para archivos*/
void CargarA(FILE *arch,int n,double **A);

/*Funciones para memoria*/
double *NuevoVector(int n);
double **NuevaMatriz(int n);
void LiberarMatriz(double **a,int m);


/*--------------------Funciones o Procesos para archivos--------------------*/
```

```
void CargarA(FILE *arch,int n,double **A){
        /* Proceso para leer los datos del sistema Ax = b, y el punto x0.
         *
         * Parámetros:
         * arch, apuntador al archivo.
         * n, tamaño de la matriz.
         * A, matriz cuadrada.
         */

        /* El formato del archivo debe contener de preferencia la
         * siguiente estructura:
         * 1.- La primera línea es un entero, el valor de n seguido de un
         *     salto de línea.
         * 2.- Se tiene la matriz A, cada elemento cada elemento de
         *     cada fila es separado por un tabulador, y cada fila por un
         *     salto de linea.
         */

        /* Declaración de variables para el ciclo de lectura */
        int i,j;

        /* Se lee A */
        for(i=0;i<n;i++){
                /* Se lee la i-ésima fila de Matriz A */
                for(j=0;j<n;j++) fscanf(arch,"%lf",&A[i][j]);
        }
}

/*-------------------- -Funciones y procesos para Memoria---- ------------------*/

double *NuevoVector(int n){
        /* Función para crear un vector de tamaño n.
         *
         * Parámetros:
         * n, tamaño del vector.
         */

        /* Se crea apuntador del vector */
        double *y;

        /* Se asigna la memoria para el vector */
        y=(double *)malloc(n*sizeof(double));

        /* Se regresa la dirección de la memoria */
        return y;
}


double **NuevaMatriz(int n){
        /* Función que crea una matriz cuadrada
         *
         * Parámetros:
         * n, tamaño de la matriz.
         */

        /* Apuntador para la matriz */
```

```
        double **p;

        /* Declaración de variable para el ciclo */
        int i;

        /* Memoria que guarda la dirección de cada fila */
        p=(double **)malloc(n*sizeof(double *));

        /* Se verifica si el vector de direcciones fue creado */
        if(p!=NULL){
                /* Se procede a crear cada fila */
                for(i=0;i<n;i++){
                        /* Memoria para cada fila */
                        p[i]=(double *)malloc(n*sizeof(double));

                        /* Se verifica si se crea la fila, en caso contrario
                         * se rompe el ciclo.
                         */
                        if(p[i]==NULL) break;
                }

                /* Se verifica la creación de todas las filas */
                if(i!=n){
                        printf("No se creo matriz\n");
                        /* Se librera la memoria si se rompe el ciclo */
                        LiberarMatriz(p,i);
                }
        }

        /* Se devuelve la dirección de la matriz */
        return p;
}


void LiberarMatriz(double **a,int m){
        /* Proceso que libera la memoria de una matriz
         *
         * Parámetros:
         * a, apuntador de la memoria de la matriz.
         * m, número de filas que necesitan ser liberadas.
         */

        /* Declaración de variable para el ciclo */
        int i;

        /* Ciclo que libera las filas necesarias */
        for(i=0;i<m;i++) free(a[i]);

        /* Se libera el vector de direcciones de las filas */
        free(a);

        /* Se retorna un valor nulo */
        a=NULL;
}
```

Librería FuncionesMH.h

```
/* Librería FuncionesMH.h
 *
 * Juan Antonio Vázquez Morales.
 * 20 de noviembre de 2017.
 *
 * Funciones diversas del Algoritmo del Método de Householder (AMH)
 *
 */

/*----------------Declaración de todas las funciones y procesos----------------*/
double Calcular_q(int n,int k,double **A);
void Calcular_v(int n,int k,double alpha,double **A,double *v);
void Calcular_u(int n,int k,double rsq,double *v,double **A, double *u);
double Calcular_prod(int n,int k,double *v,double *u);
void Calcualr_z(int n,int k,double prod,double rsq,double *u,double *v
     ,double *z);
void Actualizar_A(int n, int k,double *v,double *z,double **A);


/*-------------------------Funciones y procedimientos-------------------------*/


double Calcular_q(int n, int k,double **A){
        /* Función que obtiene q del AMH.
         *
         * Parámetros:
         * n, tamaño de la matriz.
         * k, número de iteración.
         * A, matriz A^(k-1).
         */

        /* Declaración de variables para el ciclo */
        int j;
        /* Declaración de la variable a devolver */
        double q=0.0;

        /* Ciclo que obtiene q */
        for(j=k;j<n;j++) q+=A[j][k-1]*A[j][k-1];

        /* Se retorna q */
        return q;
}


void Calcular_v(int n,int k,double alpha,double **A,double *v){
        /* Proceso que obtiene el vector v del AMH.
         *
         * Parámetros:
         * n, tamaño de la matriz.
         * k, número de iteración.
         * alpha, constante necesaria del AMH.
         * A, matriz A^(k-1).
         * v, vector resultante.
         */
```

```
        /* Declaración de variables para el ciclo */
        int j;

        /* Se obtiene v */
        v[k-1]=0.0;
        v[k]=A[k][k-1]-alpha;
        for(j=k+1;j<n;j++) v[j]=A[j][k-1];
}


void Calcular_u(int n,int k,double rsq,double *v,double **A, double *u){
        /* Proceso que obtiene el vector u del AMH.
         *
         * Parámetros:
         * n, tamaño de la matriz.
         * k, número de iteración.
         * rsq, constante necesaria del AMH.
         * v, vector necesario del AMH.
         * A, matriz A^(k-1).
         * u, vector resultante del AMH.
         */

        /* Declaración de variables para el ciclo */
        int i,j;

        /* Se procede a calcular u */
        for(j=k-1;j<n;j++){
                u[j]=0.0;
                for(i=k;i<n;i++) u[j]+=(A[j][i]*v[i]);
                u[j]=(u[j]/rsq);
        }
}


double Calcular_prod(int n,int k,double *v,double *u){
        /* Función que obtiene prod del AMH.
         *
         * Parámetros:
         * n, tamaño de la matriz.
         * k, número de iteración.
         * v, vector necesario del AMH.
         * u, vector necesario del AMH.
         */

        /* Declaración de variable para el ciclo */
        int i;

        /* Declaración de variable a retornar */
        double prod=0.0;

        /* calculo de prod */
        for(i=k;i<n;i++) prod+=v[i]*u[i];

        return prod;
}
```

```
void Calcualr_z(int n,int k,double prod,double rsq,double *u,double *v
     ,double *z){
        /* Proceso que obtiene el vector z del AMH.
         *
         * Parámetros:
         * n, tamaño de la matriz.
         * k, número de iteración.
         * prod, constante necesaria del AMH.
         * rsq, constante necesaria del AMH.
         * u, vector necesario del AMH.
         * v, vector necesario del AMH.
         * z, vector resultante del AMH.
         */

        /* Declaración de variable para el ciclo */
        int j;

        /* Declaración de la constante por la que se multiplican los
         * elementos de v
         */
        double c=(prod/(2.0*rsq));

        /* Obtención de z */
        for(j=k-1;j<n;j++) z[j]=u[j]-c*v[j];
}


void Actualizar_A(int n, int k,double *v,double *z,double **A){
        /* Proceso que obtiene A^(k+1) del AMH actualizando A.
         *
         * Parámetros:
         * n, tamaño de la matriz.
         * k, número de iteración.
         * v, vector necesario del AMH.
         * z, vector resultante del AMH.
         * A, matriz a modificar.
         */

        /* Declaración de variables para el ciclo */
        int l,j;

        /* Se procede a actualizar A mediante el AMH */
        for(l=k;l<n-1;l++){
                for(j=l+1;j<n;j++){
                        A[j][l]=A[j][l]-v[l]*z[j]-v[j]*z[l];
                        A[l][j]=A[j][l];
                }
                A[l][l]=A[l][l]-2.0*v[l]*z[l];
        }

        A[n-1][n-1]=A[n-1][n-1]-2.0*v[n-1]*z[n-1];

        for(j=k+1;j<n;j++){
                A[k-1][j]=0.0;
                A[j][k-1]=0.0;
        }
```

```
        A[k][k-1]=A[k][k-1]-v[k]*z[k-1];
        A[k-1][k]=A[k][k-1];
}
```

Librería FuncionesMQR.h

```
/* Librería FuncionesMQR.h
 *
 * Juan Antonio Vázquez Morales.
 * 20 de noviembre de 2017.
 *
 * Funciones diversas del Algoritmo del Método QR (MQR).
 *
 */

/*----------------Declaración de todas las funciones y procesos----------------*/
double Calcular_sigma(double an,double mu1,double mu2);
void Calcular_Rk(int n,double *c,double *r,double *s,double *u
                 ,double *v,double *x,double *y,double *z,double **A);
void Actualizar_AQR(int n,double *cs,double *u,double *s,double *x,double *z
                    ,double **A);

/*-------------------------Funciones y procedimientos-------------------------*/

double Calcular_sigma(double an,double mu1,double mu2){
        /* Función que obtiene sigma del MQR.
         *
         * Parámetros:
         * An, entrada a_n del vector del algoritmo del MQR.
         * mu1, primer valor propio calculado
         * mu2, segundo valor propio calculado.
         */

        /* Se calcula el valor propio más cercano a an */
        if(fabs(mu1-an)<fabs(mu2-an)) return mu1;
        else return mu2;
}

void Calcular_Rk(int n,double *cs,double *r,double *s,double *u
                 ,double *v,double *x,double *y,double *z,double **A){
        /* Proceso que obtiene R^(k) del algoritmo MQR.
         *
         * Parámetros:
         * n, tamaño de los vectores.
         * qs, cs, r, s, u, x, y, & z, vectores necesarios para calcular
         * R^(k).
         *
         * A, matriz.
         */

        /* Variable para el ciclo */
        int j;
```

```
        x[0]=v[0];
        y[0]=A[1][0];

        for(j=1;j<n;j++){
                z[j-1]=sqrt(x[j-1]*x[j-1]+A[j][j-1]*A[j][j-1]);
                cs[j]=x[j-1]/z[j-1];
                s[j]=A[j][j-1]/z[j-1];
                u[j-1]=cs[j]*y[j-1]+s[j]*v[j];
                x[j]=-s[j]*y[j-1]+cs[j]*v[j];
                if(j!=(n-1)){
                        r[j-1]=s[j]*A[j+1][j];
                        y[j]=cs[j]*A[j+1][j];
                }
        }
}

void Actualizar_AQR(int n,double *cs,double *u,double *s,double *x,double *z
                    ,double **A){
        /* Proceso que obtiene A^(k+1) del algoritmo MQR.
         *
         * Parámetros:
         * n, tamaño de los vectores.
         * cs, u, s, x, z, vectores necesarios para calcular A^(k+1).
         * A, matriz a actualizar.
         */

        /* Variable para efectuar ciclo */
        int j;

        /* Se empieza la actualización */
        z[n-1]=x[n-1];
        A[0][0]=s[1]*u[0]+cs[1]*z[0];
        A[1][0]=s[1]*z[1];

        for(j=1;j<(n-1);j++){
                A[j][j]=s[j+1]*u[j]+cs[j]*cs[j+1]*z[j];
                A[j+1][j]=s[j+1]*z[j+1];
        }

        A[n-1][n-1]=cs[n-1]*z[n-1];
}
```

Forma del Archivo

En la librería `MemoriaYArchivos.h` se propone una forma en la que el archivo que contiene el archivo debe estar escrito, a continuación se muestra un ejemplo.

```
6
8       2       5       2       7       2
2       5       4       7       2       8
5       4       7       3       5       4
2       7       3       3       7       9
```

```
7       2       5       7       4       1
2       8       4       9       1       9
```

En este caso, A es una matriz cuadrada de tamaño 6, el cual es el primer dato, y además

$$A = \begin{bmatrix} 8 & 2 & 5 & 2 & 7 & 2 \\ 2 & 5 & 4 & 7 & 2 & 8 \\ 5 & 4 & 7 & 3 & 5 & 4 \\ 2 & 7 & 3 & 3 & 7 & 9 \\ 7 & 2 & 5 & 7 & 4 & 1 \\ 2 & 8 & 4 & 9 & 1 & 9 \end{bmatrix},$$

la cual es una matriz simétrica.

Aunque el archivo no es necesario que presente esta forma, si es necesario que los datos aparezcan en ese orden.

Referencias

[1] Bazaraa, M., Sherali, H., and Shetty, C. *Nonlinear programming: theory and algorithms*, 3 ed. John Wiley & Sons, 2006.

[2] Burden, R., and Faires, J. *Análisis Numérico*, 7 ed. Thomson Learning, 2002.

[3] Calvillo, G. *Introducción a la Optimización, Apuntes para ENOAN 2017*. 2017.

[4] Hestenes, M., and Stiefel, E. *Methods of conjugate gradients for solving linear systems*, vol. 49. NBS, 1952.

[5] Krízek, M., and Neittaanmäki, P. *Mathematical and Numerical Modelling in Electrical Engineering: Theory and applications*. Springer Science & Business Media, 1996.

[6] Luenberger, D., and Ye, Y. *Linear and Nonlinear Programming*. Springer, 2016.

[7] Nocedal, J., and Wright, S. *Numerical optimization*, 2 ed. Springer, 2006.

[8] Poole, D. *Álgebra lineal: una introducción moderna*, 3 ed. Cengage Learning Editores, 2011.

[9] Rivlin, T. *Chebyshev Polynomials. From Approximation Theory to Algebra and Number Theory*, 2 ed. John Wiley & Sons, 1990.