Métodos Numéricos
TP1

3 de septiembre de 2015

Integrante	LU	Correo electrónico
Martin Baigorria	575/14	martinbaigorria@gmail.com
Federico Beuter	827/13	federicobeuter@gmail.com
Mauro Cherubini	835/13	cheru.mf@gmail.com
Rodrigo Kapobel	695/12	rok_35@live.com.ar

Reservado para la cátedra

Instancia	Docente	Nota
Primera entrega		
Segunda entrega		

Resumen: El siguiente trabajo practico tiene como objetivo implementar, utilizar y evaluar el método de eliminación gausiana y la factorización LU para resolver un problema que involucra la propagación del calor en la pared de un horno descripta con un laplaciano. Para ello se discretizara esta ecuación diferencial y luego se planteara un sistema matricial de la forma $Ax = b$ para calcular la temperatura en los diferentes puntos de la discretizacion en la pared del horno. Se analizaran diferentes escenarios en las condiciones del problema para evaluar en que escenarios una factorización supera a la otra. Se evaluaran algunas cuestiones relacionadas con el problema en sí, como por ejemplo la presición de un algoritmo de búsqueda de isotermas en la pared del horno y la velocidad de convergencia a la isoterma empírica. Finalmente concluiremos que la factorización LU supera ampliamente a la eliminación gausiana en cuanto a complejidad temporal en escenarios donde cambia el vector b de forma recurrente. A su vez, notamos que la presición del algoritmo de búsqueda de isotermas no es estrictamente decreciente en función de la discretización, aunque por supuesto mejora a medida que aumenta en múltiplos de 2.

Keywords: Gaussian Elimination, LU Factorization.

Índice

1. Introducción

Existe una gran variedad de problemas que pueden ser modelados por medio de sistemas de equaciones lineales. Estas ecuaciones pueden ser expresadas mediante un sistema matricial que se puede escribir de la forma $Ax = b$ donde $A \in \mathbb{R}^{n \times n}$ y $x, b \in \mathbb{R}^{n \times 1}$. Una vez representado, se debe buscar alguna forma de resolver el sistema, es decir, buscar el vector x. Existen numerosas maneras de resolver este problema, entre ellas tenemos por ejemplo el clásico algoritmo de eliminación gaussiana y la factorización LU.

El objetivo de este trabajo práctico es modelar y resolver el problema de la difusión del calor en la pared de un horno circular. A priori, lo que sabemos es que el calor se propaga siguiendo la ecuación diferencial dada por el laplaciano en función del ángulo y la distancia desde el centro del horno. Aunque esta ecuación diferencial tiene una solución analítica, el trabajo practico apunta a que modelemos este problema discretizando el dominio en coordenadas polares y planteando el sistema de ecuaciones dado por el laplaciano de forma matricial. De esta forma podemos encontrar una aproximación de la temperatura en cada punto de la discretización.

Por cuestiones de seguridad e integridad estructural del horno, la isoterma 500°C (una curva inducida por la temperatura interna y externa, y en donde la temperatura es efectivamente de 500°C) deberá hallarse en el rango definido entre el radio interno y externo de la pared. Por esta razón, una vez calculada la temperatura aproximada en diferentes puntos de la discretización, tendremos como objetivo encontrar de alguna forma ésta isoterma, y asi comprobar la estabilidad del horno. Para ello propondremos diferentes algoritmos que aprovechen dicha discretización para aproximar la ubicación de la misma. Estas problemáticas pueden verse claramente en las siguientes figuras:

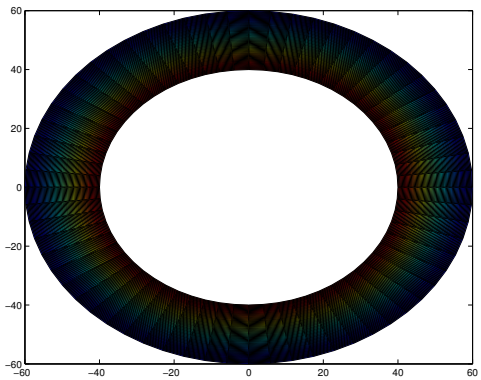

Figura 1: Heat map.

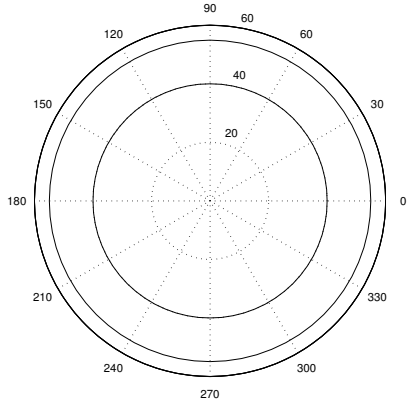

Figura 2: Isoterma.

En el heat map se puede ver como la temperatura representada con colores va bajando a medida que uno se aleja de la pared interna del horno. A su vez, en el gráfico de la isoterma podemos ver la pared interna del horno y la isoterma 500°C. En este caso, dado que no toca la pared externa del horno decimos que el mismo es estable. Notar que las isotermas no necesariamente son circulares, dado que la temperatura externa puede variar dependiendo del angulo. Este es simplemente un ejemplo ilustrativo.

En un primer momento, implementaremos y experimentaremos con el método de eliminación gausiana y la factorización LU. Evaluaremos que método es mejor dependiendo de las diferentes condiciones del horno y del grado de granularidad. A su vez analizaremos que método se comporta mejor al tener condiciones de temperatura variables, como por ejemplo en el caso en el que el vector de variables independientes b_t varie con el tiempo. A priori, sabemos que para una única instancia la eliminación gaussiana y la factorización LU pertenecen a $\mathcal{O}(n^3)$. Esto se debe a que la eliminación gaussiana simplemente transforma el sistema original en uno equivalente que es triangular superior en $\mathcal{O}(n^3)$, en donde n es el numero de incógnitas. Luego se resuelve este sistema en $\mathcal{O}(n^2)$. Por otro lado, la factorizacion LU transforma el sistema original en un sistema del tipo $LUx = b$, donde L es una matriz triangular inferior y U es una matriz triangular superior en costo $\mathcal{O}(n^3)$. Finalmente, se resuelven los sistemas $Ly = b$ y $Ux = y$ para obtener una solución x en $\mathcal{O}(n^2)$.

En términos asintóticos, ambos métodos tienen la misma complejidad. Sin embargo, la factorización LU tiene la ventaja de que para instancias adicionales la solución del sistema se puede computar en $\mathcal{O}(n^2)$, mientras que la eliminación gaussiana debe repetir todo el procedimiento nuevamente en $\mathcal{O}(n^2)$. Por lo tanto, esperamos que la experimentación confirme este resultado teórico a medida que aumentemos la dimension y el número de instancias.

Una parte importante de este trabajo práctico es evaluar la integridad estructural de los hornos. Por lo tanto, analizaremos la velocidad de convergencia de nuestro algoritmo a la isoterma teórica dependiendo del nivel de discretización y de las variables del horno. Finalmente analizaremos el `trade off` entre tiempo de ejecución y que tan buenas son las aproximaciones de la isoterma al cambiar la granularidad.

2. Desarrollo

Dado el problema del horno, en primer lugar debemos modelarlo. Consideremos la sección horizontal de un horno de acero cilíndrico como el de la siguiente figura. El sector A es la pared del horno, y el sector B es el interior del mismo, en el cual se funde el acero a temperaturas elevadas. Tanto el borde externo como el borde interno de la pared son circulares con un centro en común. Suponemos que la temperatura del acero dentro del horno es constante e igual a 1500^oC.

Existen sensores ubicados en la parte externa del horno para medir la temperatura exterior. La misma se encuentra habitualmente entre 50^oC y 200^oC. El problema que debemos resolver consiste en estimar la isoterma de 500^oC dentro de la pared del horno. Si esta isoterma está demasiado cerca de la pared externa del horno, existe el riesgo que la integridad estructural del horno este comprometida.

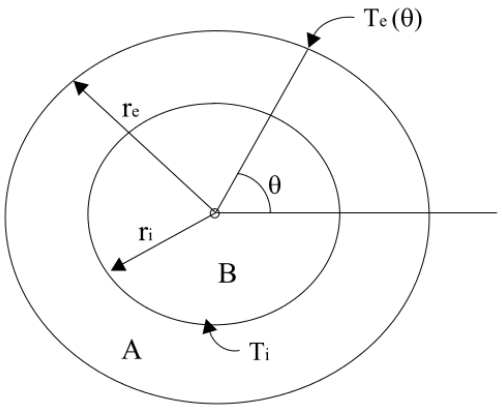

Figura 3: Sección circular del horno

Sea $r_e \in \mathbb{R}$ el radio exterior de la pared y sea $r_i \in \mathbb{R}$ el radio interior de la pared. Llamemos $T(r, \theta)$ a la temperatura en el punto dado por las coordenadas polares (r, θ), siendo r el radio y θ el angulo polar de dicho punto. En el `estado estacionario`, esta temperatura satisface la ecuación del calor dada por el laplaciano:

$$\frac{\partial^2 T(r,\theta)}{\partial r^2} + \frac{1}{r}\frac{\partial T(r,\theta)}{\partial r} + \frac{1}{r^2}\frac{\partial^2 T(r,\theta)}{\partial \theta^2} = 0 \tag{1}$$

Para resolver esta ecuación de forma numérica, discretizamos la superficie de la pared como en la siguiente figura y luego aproximamos las derivadas parciales utilizando diferencias finitas.

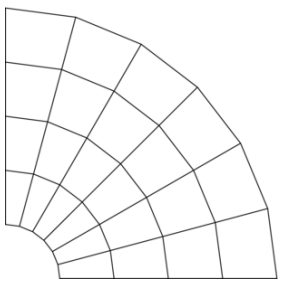

Figura 4: Discretización de la pared del horno.

$$\frac{\partial^2 T(r,\theta)}{\partial r^2}(r_j, \theta_k) \cong \frac{t_{j-1,k} - 2t_{jk} + t_{j+1,k}}{(\Delta r)^2} \tag{2}$$

$$\frac{\partial T(r,\theta)}{\partial r}(r_j, \theta_k) \cong \frac{t_{j,k} - t_{j-1,k}}{\Delta r} \tag{3}$$

$$\frac{\partial^2 T(r,\theta)}{\partial \theta^2}(r_j, \theta_k) \cong \frac{t_{j,k-1} - 2t_{jk} + t_{j,k+1}}{(\Delta \theta)^2} \tag{4}$$

Reemplazando la aproximación numérica en el laplaciano y el radio por su respectiva discretización obtenemos:

$$\frac{t_{j-1,k} - 2t_{jk} + t_{j+1,k}}{(\Delta r)^2} + \frac{1}{r_j}\frac{t_{j,k} - t_{j-1,k}}{\Delta r} + \frac{1}{r_j^2}\frac{t_{j,k-1} - 2t_{jk} + t_{j,k+1}}{(\Delta\theta)^2} = 0$$

Donde $r_j = r_i + j \times \Delta r$, $\Delta r = \frac{(r_e - r_i)}{m}$ y $\Delta\theta = \frac{2\pi}{n}$.

De esta manera aproximamos de forma discreta la ecuación diferencial dada por el laplaciano.

Si llamamos $T_i \in \mathbb{R}$ a la temperatura en el interior del horno (sector B) y $T_e : [0, 2\pi] \to \mathbb{R}$ a la función de temperatura en el borde exterior del horno (de modo tal que el punto (r_e, θ) tiene temperatura $T_e(\theta)$), entonces tenemos que

$$T(r, \theta) = T_i \quad \textit{para todo punto } (r, \theta) \textit{ con } r \leq r_i \tag{5}$$

$$T(r_e, \theta) = T_e(\theta) \quad \textit{para todo punto } (r_e, \theta) \tag{6}$$

2.1. Discretización

Para resolver este problema computacionalmente, discretizamos el dominio del problema (el sector A) en coordenadas polares. Consideramos una partición $0 = \theta_0 < \theta_1 < ... < \theta_n = 2\pi$ en n ángulos discretos con $\theta_k - \theta_{k-1} = \Delta\theta$ para $k = 1, ..., n$, y una partición $r_i = r_0 < r_1 < ... < r_m = r_e$ en $m + 1$ radios discretos con $r_j - r_{j-1} = \Delta r$ para $j = 1, ..., m$.

De esta manera, terminamos con un sistema de $(m + 1) * n$ ecuaciones lineales, que puede ser experesado como $Ax = b$. Para cada temperatura $t_{j,k}$, tendremos un laplaciano. Esto no sucede con los valores de las temperaturas en las puntas, donde ya a priori sabemos el valor final t_i y $t_e(\theta)$. Estas temperaturas en las puntas formaran parte del vector de valores independientes b al armar el sistema, al que le corresponde un canónico. La discretización muchas veces depende de los valores anteriores y posteriores, por lo que hay que tener cuidado de no caer en uno de estos casos borde al formular el sistema.

2.2. Sistema Lineal

Para formular el sistema lineal, en primer lugar debemos despejar cada una de las variables $t_{j,k}$ de la aproximación discreta del laplaciano:

$$\frac{t_{j-1,k} - 2t_{j,k} + t_{j+1,k}}{(\Delta r)^2} + \frac{1}{r_j}\frac{t_{j,k} - t_{j-1,k}}{\Delta r} + \frac{1}{r_j^2}\frac{t_{j,k-1} - 2t_{j,k} + t_{j,k+1}}{(\Delta\theta)^2} = 0$$

Reescribiendo:

$$\alpha_{j,k} \times t_{j,k} + \alpha_{j-1,k} \times t_{j-1,k} + \alpha_{j+1,k} \times t_{j+1,k} + \alpha_{j,k+1} \times t_{j,k+1} + \alpha_{j,k-1} \times t_{j,k-1} = 0$$

Donde:

$$\alpha_{j,k} = \frac{-2}{(\Delta r)^2} + \frac{1}{r_j \times \Delta r} + \frac{-2}{r_j^2 \times (\Delta\theta)^2} \tag{7}$$

$$\alpha_{j,k+1} = \frac{1}{r_j^2 \times (\Delta\theta)^2} \tag{8}$$

$$\alpha_{j,k-1} = \frac{1}{r_j^2 \times (\Delta\theta)^2} \tag{9}$$

$$\alpha_{j+1,k} = \frac{1}{(\Delta r)^2} \tag{10}$$

$$\alpha_{j-1,k} = \frac{1}{(\Delta r)^2} - \frac{1}{r_j \times \Delta r} \tag{11}$$

2.2.1. Matriz del sistema

Con las ecuaciones *(7)-(11)* armamos la matriz del sistema, en donde los coeficientes serán los α anteriores, expresando en cada fila su valor para cada temperatura. Hay que tener en cuenta que en todas las ecuaciones habrá un α para cada uno de los $t_{j,k}$, valiendo 0 en aquellos casos en que no aparezca dicha incógnita, 1 en caso de ser t_i o $t_e\theta$, o en su defecto α como se han definido en el desarrollo anterior.

Por cuestiones de optimización organizamos la matriz con el orden de sus columnas (inducidas por los $t_{k,j}$), de acuerdo al orden de aparición impartido por el recorrido de θ_0 a θ_{n-1} (variando el radio luego de cada vuelta) desde r_i hasta r_e inclusive.
Gráficamente dicha matriz queda definida de la siguiente forma:

$$\begin{pmatrix} \alpha_{0,0} & \alpha_{0,1} & \cdots & \alpha_{0,(m+1)(n-1)} \\ \alpha_{1,0} & \alpha_{1,1} & \cdots & \alpha_{1,(m+1)(n-1)} \\ \vdots & \vdots & \ddots & \vdots \\ \vdots & \vdots & & \vdots \\ \vdots & \vdots & & \vdots \\ \vdots & \vdots & & \vdots \\ \alpha_{(m+1)(n-1),0} & \alpha_{(m+1)(n-1),1} & \cdots & \alpha_{(m+1)(n-1),(m+1)(n-1)} \end{pmatrix}$$

Notese que la indexación de los coeficientes empieza en 0, en vez de en 1 como es habitual; el motivo es el de generar una mayor cohesión con la idexación de las temperaturas, que a su vez, son indexadas de acuerdo a los r_i y los θ_i correspondientes (indexados efectivamente desde el 0).

Ahora bien como las primeras y últimas n filas corresponden al radio interior y exterior respectivamente, sabemos que en ese rango $\alpha_{jj} = 1$ y $\alpha_{ji} = 0, \forall j \neq i$. Lo que nos genera una matriz identidad de $n * n$ en las esquinas superior izquierda e inferior derecha; dando además a la matriz del sistema la condición de matriz *banda*.

$$\begin{pmatrix} 1 & 0 & \cdots & 0 & \cdots & 0 & 0 & \cdots & 0 \\ 0 & \ddots & 0 & \vdots & \cdots & \vdots & \vdots & \ddots & \vdots \\ \vdots & 0 & 1 & 0 & \cdots & 0 & 0 & \cdots & 0 \\ \alpha_{n-2,0} & \cdots & \cdots & \alpha_{n-2,n-2} & \cdots & \alpha_{n-2,(m+1)(n-1)-n} & \cdots & \cdots & \alpha_{n-2,..} \\ \vdots & \ddots & & \vdots & \cdots & \vdots & & \ddots & \vdots \\ \alpha_{....,....} & & \ddots & \alpha_{(m+1)(n-1)-n,n-2} & \cdots & \alpha_{(m+1)n-n,(m+1)(n-1)-n} & & & \alpha_{....,....} \\ 0 & \cdots & 0 & 0 & \cdots & 0 & 1 & 0 & \cdots \\ \vdots & \ddots & \vdots & \vdots & \cdots & \vdots & 0 & \ddots & 0 \\ 0 & \cdots & 0 & 0 & \cdots & 0 & \cdots & 0 & 1 \end{pmatrix}$$

Si bien no se muestra explicitamente en la figura, para sistemas lo suficientemente grandes se podrá apreciar como debajo de la Id superior izquierda y arriba de la Id inferior derecha, se generan submatrices diagonales que estrecharan la anchura de la banda, dejandonos una matriz *banda-n*.

2.3. Isoterma

Una vez que resolvamos el sistema lineal, debemos buscar la isoterma. Dado que es una aproximación discreta, es muy probable que no encontremos valores de temperatura iguales a la curva de nivel. Por lo tanto debemos tener cierta tolerancia de error, o hasta interpolar de alguna manera las curvas de nivel adyacentes en la discretización. Es decir, debemos definir algún criterio de búsqueda sobre el vector b. Nuestros criterios de búsqueda se basaran en el supuesto de que a medida que nos alejamos del centro del horno en `estado estacionario`, las temperaturas serán menores. Una consideración sumamente importante en el contexto de nuestro problema: intentaremos buscar métodos que sean pesimistas, es decir, que sigan el principio de prudencia. Esto significa que nuestros algoritmos intentaran ubicar la aproximación de la isoterma mas cerca de la pared del horno de lo que efectivamente esta, para poder evaluar de forma certera si el horno es estructuralmente estable o no.

2.3.1. Algoritmo: Ultimo mayor

El primer criterio de búsqueda consiste en tomar el primer valor de temperatura menor al que estamos buscando para cada angulo en la discretización. Esto se puede ver en el siguiente pseudo-código:

Inserte pseudo-codigo aqui.

2.3.2. Algoritmo: Promedio pesado

Este criterio buscara ubicar la isoterma de forma inteligente, buscando el valor mayor y menor entre los que se ubica la isoterma y haciendo un promedio pesado. Esto se puede ver en el siguiente pseudo-código:

Inserte pseudo-codigo aqui.

2.4. Propiedades del sistema

Como ya se ha visto en la materia, no es posible aplicar los métodos propuestos para la resolución a cualquier sistema de ecuaciones. Por ello deberemos demostrar la siguiente proposición.

Proposición 2.1 *Sea $A \in \mathbb{R}^{n\times n}$ la matriz obtenida para el sistema definido por (1)-(6). Demostrar que es posible aplicar Eliminación Gaussiana sin pivoteo.*

Con este proposito es que analizaremos algunas propiedades que satisface nuestra matriz, y como aún bajo el proceso de Eliminación Gaussiana, se mantienen fieles e invariantes.

A es d.d.(no estricta)

Demostremos primero que la matriz A del sistema lineal, definida como antes, cumple la propiedad de ser diagonal dominante (no estricta).

Esto en nuestro caso es pedir que:

$$|\alpha_{jj}| \geq \sum_{k=0,k\neq j}^{(m+1)(n-1)} |\alpha_{jk}|, \forall j \in \{0,...,(m+1)(n-1)\} \tag{12}$$

En el caso de que $j \in \{0,...,n-1\} \cup \{(m+1)(n-1)-n,...,(m+1)(n-1))\}$, es decir, que se trate de las primeras o últimas n filas, (12) es satisfecha trivialmente, pues $|\alpha_{jj}| = 1 \geq \sum_{k=0,k\neq j}^{(m+1)(n-1)} |\alpha_{jk}| = \underbrace{0+\ldots+0}_{(m+1)(n)} = 0$ pues $\alpha_{jk_{k\neq j}} = 0$

Nos queda ver el caso contrario. Para ello debemos desarrollar la sumatoria y despejar los α_{jk} mediante (7)...(11), de los cuales los siguientes 5 seán los únicos α_{jk} distintos de 0.
De este caso, en particular llegaremos a una equivalencia.

$$|\alpha_{jj}| = |\alpha_{j+1k}| + |\alpha_{j-1k}| + |\alpha_{jk-1}| + |\alpha_{jk+1}| \tag{13}$$

$$\left|\frac{-2}{(\Delta r)^2} + \frac{1}{r_j \times \Delta r} + \frac{-2}{r_j^2 \times (\Delta\theta)^2}\right| = \left|\frac{1}{(\Delta r)^2}\right| + \left|\frac{1}{(\Delta r)^2} - \frac{1}{r_j \times \Delta r}\right| + \left|\frac{1}{r_j^2 \times (\Delta\theta)^2}\right| + \left|\frac{1}{r_j^2 \times (\Delta\theta)^2}\right| \tag{14}$$

$$\left|-\left(\frac{-2}{(\Delta r)^2} + \frac{1}{r_j \times \Delta r} + \frac{-2}{r_j^2 \times (\Delta\theta)^2}\right)\right| = \left|\frac{1}{(\Delta r)^2}\right| + \left|\frac{1}{(\Delta r)^2} - \frac{1}{r_j \times \Delta r}\right| + \left|\frac{2}{r_j^2 \times (\Delta\theta)^2}\right| \tag{15}$$

$$\left|\frac{1}{(\Delta r)^2} + \underbrace{\frac{1}{(\Delta r)^2} - \frac{1}{r_j \times \Delta r}}_{\geq 0} + \frac{2}{r_j^2 \times (\Delta\theta)^2}\right| = \left|\frac{1}{(\Delta r)^2}\right| + \left|\frac{1}{(\Delta r)^2} - \frac{1}{r_j \times \Delta r}\right| + \left|\frac{2}{r_j^2 \times (\Delta\theta)^2}\right| \tag{16}$$

$$\left|\frac{1}{(\Delta r)^2}\right| + \left|\frac{1}{(\Delta r)^2} - \frac{1}{r_j \times \Delta r}\right| + \left|\frac{2}{r_j^2 \times (\Delta\theta)^2}\right| = \left|\frac{1}{(\Delta r)^2}\right| + \left|\frac{1}{(\Delta r)^2} - \frac{1}{r_j \times \Delta r}\right| + \left|\frac{2}{r_j^2 \times (\Delta\theta)^2}\right| \quad \blacksquare \tag{17}$$

Con este resultado probamos que la matriz inicial A es diagonal dominante (no estricta), ahora nos toca ver que el algoritmo preserva esta propiedad a lo largo de cada iteración. Naturalmente lo probaremos por inducción.

* De ahora en mas notaremos como $\alpha_{jk}^{(i)}$ al coeficiente situado en la posición de α_{jk} en A, de la matriz $A^{(i)}$ obtenida como resultado tras aplicar las primeras $i-1$ iteraciones de la Eliminación Gaussiana. *Ej: $\alpha_{jk}^{(2)}$ alude al nuevo α_{jk} obtenido tras triangular sólo la seccion de la primera columna.*

Veamos el caso base:

Siendo A *d.d.* (no estricta), probemos que $\left|\alpha_{jj}^{(2)}\right| \geq \sum_{k=0,k\neq j}^{(m+1)(n-1)} \left|\alpha_{jk}^{(2)}\right|, \forall j \in \{1, ..., (m+1)(n-1)\}$

$$\alpha_{jk}^{(2)} = \alpha_{jk} - \frac{\alpha_{0k}}{\alpha_{00}}\alpha_{j0} \tag{18}$$

$$\sum_{k=1,k\neq j}^{(m+1)(n-1)} \left|\alpha_{jk}^{(2)}\right| = \sum_{k=1,k\neq j}^{(m+1)(n-1)} \left|\alpha_{jk} - \frac{\alpha_{0k}}{\alpha_{00}}\alpha_{j0}\right| \leq \sum_{k=1,k\neq j}^{(m+1)(n-1)} |\alpha_{jk}| + \sum_{k=1,k\neq j}^{(m+1)(n-1)} \left|\frac{\alpha_{0k}\alpha_{j0}}{\alpha_{00}}\right| \tag{19}$$

$$\leq |\alpha_{jj}| - |\alpha_{j0}| + \frac{|\alpha_{j0}|}{|\alpha_{00}|}\left(|\alpha_{00}| - |\alpha_{0j}|\right) \leq |\alpha_{jj}| - \frac{|\alpha_{j0}|\,|\alpha_{0j}|}{|\alpha_{00}|} \leq \left|\alpha_{jj} - \frac{\alpha_{j0}\alpha_{0j}}{\alpha_{00}}\right| = \left|\alpha_{jj}^{(2)}\right| \tag{20}$$

Paso inductivo:

Por hipotesis tenemos que $\left|\alpha_{jj}^{(i)}\right| \geq \sum_{k=0,k\neq j}^{(m+1)(n-1)} \left|\alpha_{jk}^{(i)}\right|, \forall j \in \{i-2, ..., (m+1)(n-1)\}$ para las primeras $i-1$ iteraciones.

Queremos ver que: $\left|\alpha_{jj}^{(i+1)}\right| \geq \sum_{k=0,k\neq j}^{(m+1)(n-1)} \left|\alpha_{jk}^{(i+1)}\right|, \forall j \in \{i-1, ..., (m+1)(n-1)\}$

$$\alpha_{jk}^{(i+1)} = \alpha_{jk}^{(i)} - \frac{\alpha_{i-2,k}^{(i)}}{\alpha_{i-2,i-2}^{(i)}}\alpha_{j,i-2}^{(i)} \tag{21}$$

$$\sum_{k=i-1,k\neq j}^{(m+1)(n-1)} \left|\alpha_{jk}^{(i+1)}\right| = \sum_{k=i-1,k\neq j}^{(m+1)(n-1)} \left|\alpha_{jk}^{(i)} - \frac{\alpha_{i-2,k}^{(i)}}{\alpha_{i-2,i-2}^{(i)}}\alpha_{j,i-2}^{(i)}\right| \leq \sum_{k=i-1,k\neq j}^{(m+1)(n-1)} \left|\alpha_{jk}^{(i)}\right| + \sum_{k=i-1,k\neq j}^{(m+1)(n-1)} \left|\frac{\alpha_{i-1,k}^{(i)}\alpha_{j,i-2}^{(i)}}{\alpha_{i-2,i-2}^{(i)}}\right| \tag{22}$$

$$\overset{\text{Por Hip.}}{\leq} \left|\alpha_{jj}^{(i)}\right| - \underbrace{\sum_{k=0,k\neq j}^{i-2} \left|\alpha_{j,k}^{(i)}\right|}_{=0} + \frac{\left|\alpha_{j,i-2}^{(i)}\right|}{\left|\alpha_{i-2,i-2}^{(i)}\right|}\left(\left|\alpha_{i-2,i-2}^{(i)}\right| - \underbrace{\sum_{k=0,k\neq j}^{i-2} \left|\alpha_{i-2,k}^{(i)}\right|}_{=0}\right) \tag{23}$$

$$\leq \left|\alpha_{jj}^{(i)}\right| - \frac{\left|\alpha_{j,i-2}^{(i)}\right|\left|\alpha_{i-2,j}^{(i)}\right|}{\left|\alpha_{i-2,i-2}^{(i)}\right|} \leq \left|\alpha_{jj}^{(i)} - \frac{\alpha_{j,i-2}^{(i)}\alpha_{i-2,j}^{(i)}}{\alpha_{i-2,i-2}^{(i)}}\right| = \left|\alpha_{jj}^{(i+1)}\right| \quad \blacksquare \tag{24}$$

Finalizada así esta demostración, sabemos que podemos contar con que tanto la matriz inicial A como $A^{(i)}$, son *d.d.* (no estricta). Por desgracia, si bien esta hipotesis parece alentadora, es insuficiente para garantizar que a A se le pueda aplicar la Eliminación Gaussiana sin pivoteo. Esto es así porque $A^{(i)}$ aún puede caer en un caso borde, el de tener al 0 como valor en alguno de los elementos de su diagonal principal, y obviamente a algún otro distinto de 0 en la parte inferior de su misma columna; lo que induciría a un intercambio de filas (pivoteo).

Teniendo en claro cual es el conflicto al que nos enfrentamos, de ahora en mas, nos enfocaremos en probar que para la matriz $A^{(i)}$ del sistema esta situación no se nos presenta.

$A^{(i)}$ no tiene al 0 en su diagonal

Para empezar podemos observar que dicha situación sólo podría suceder en cierta sección conflictiva, en particular sobre la submatriz central que resulta de *recortar* las primeras y últimas n filas y columnas. La razón de esto, es porque las primeras n filas son e_1 a e_n ordenadas de manera triangular; de esta forma no solo ese tramo de la diagonal posee todos sus elementos distintos de 0, sino que además la Eliminación Gaussiana triangulará la primera sección de n columnas, sin alterar a la sección complementaria; y obviamente sin emplear pivoteo. Analogamente resulta que

las últimas n filas son $e_{(m+1)(n-1)-n}$ a $e_{(m+1)(n-1)}$, de nuevo, ordenadas de manera triangular; por lo que al llegar a aquella sección, ya no es necesario seguir transformando la matriz, dado que la obtenida hasta el momento, ya es triangular. Dicho esto, llamemos $A^{(*)}$ a la sección de A delimitara por aquel *recorte*.

Sobre $A^{(*)}$, veremos que se cumple una segunda hipótesis bastante útil, y es que todos los elementos de la diagonal son negativos, y los que no lo son, son 0 o positivo; al menos 1 de ellos es de esta última forma (ignorando el caso trivial de que $A^{(*)}$ sea de $1x1$). Para esto basta ver que los únicos coeficientes definidos (distinto de 0) en las filas de $A^{(*)}$ son $\alpha_{jj}, \alpha_{j-1,j}, \alpha_{j+1,j}, \alpha_{j,j-1}, \alpha_{j,j+1}$, donde los últimos tres son cocientes positivos, puesto que sólo involucran a Δr, $\Delta\theta$, y rj, como denominador (que son positivos). Y como $rj \times \Delta r > (\Delta r)^2$, $\alpha_{j-1,j}$ también es positivo. Sólo queda analizar si efectivamente α_{jj} es negativo.

$$\alpha_{jj} = \frac{-2}{(\Delta r)^2} + \frac{1}{r_j \times \Delta r} + \frac{-2}{r_j^2 \times (\Delta\theta)^2} < \frac{-2}{(\Delta r)^2} + \frac{1}{r_j \times \Delta r} < \frac{-2}{r_j \times \Delta r} + \frac{1}{r_j \times \Delta r} = \frac{-1}{r_j \times \Delta r} < 0 \quad \blacksquare \tag{25}$$

Con esto, ya podemos justificar porque $A^{(i)}$ no tiene ceros en algún elemento de su diagonal. Usando una obsevación mas, en la que nos restrigiremos a $A^{(*)}$ e ignoraremos la última fila (pues ya no necesitamos usar el elemento de su diagonal). Primero notemos que para cada fila habrá un elemento detrás (con respecto al recorrido de un radio en particular) del de la diagonal (el $\alpha_{j,j+1}$), que como ya vimos, es positivo. Al ser este elemento positivo, en la iteración número i al elemento $\alpha_{j,j+1}$ se le restará $\alpha_{i,j+1} * \alpha j, i * (\alpha_{ii})^{-1}$, que al ser $\alpha_{i,j+1}$ y $\alpha j, i$ positivos, y $\alpha_{i,i}$ negativo, $\alpha_{j,j+1}$ no podrá hacerse mas chico (pues se le esta restando algo negativo). Habrá que rescatar un caso borde; en el argumento anterior estamos usando que $\alpha_{j,j+1}$ esta situado detrás de $\alpha_{j,j}$, pero esto sólo ocurre si $\alpha_{j,j}$ no es el último elemento del *anillo* conformado por el radio r_j. No obstante habrá otro elemento situado mas hacia atrás (el $\alpha_{j+1,j}$) que corresponde al radio inmediatamente superior, también mayor a cero. El último caso borde ocurre si ese radio inmediatamente superior es el r_e (que no figura en $A^{(*)}$), pero a esto se le puede restar importancia, pues en ese caso $\alpha_{j,j}$ será el último elemento de la diagonal de $A^{(*)}$. Toda esta descripción de la matriz se ve para un sistema lo suficientemente grande, como su condición de matriz *banda*.

Ahora por último, como sabemos que $A^{(i)}$ es *d.d.*(no estricta) y que además siempre tiene un elemento positivo en cada fila (ignorando a la última), α_{jj} jamas podrá ser igual 0, porque sería equivalente a pedir que todos los elementos de su fila, fueran 0.

3. Experimentación

3.1. Metodología

Los siguientes experimentos fueron generados a partir de archivos de entrada para nuestro algoritmo que siguen ciertas reglas. En todos los casos donde analizamos el tiempo de ejecución de los algoritmos, decidimos generar 100 instancias para el mismo experimento y luego tomar la media. Esto se debe a que el uso del CPU no es uniforme y lleva a que las mediciones estén sesgadas, principalmente causado por el algoritmo de scheduling del SO y el uso de memoria.

3.2. Evaluación de los métodos

3.2.1. Tiempo de ejecución variando la dimensión

Para este experimento, decidimos analizar el tiempo de ejecución de nuestro algoritmo en función de la dimension de la matriz A. A priori, dado que ambos algoritmos son $\mathcal{O}(n^3)$ y solo se están ejecutando sobre una única instancia, esperamos que los tiempos de ejecución sean sumamente similares.

Para la generación de instancias, utilizamos $m = 3$, $n = 3$, $r_i = 10$, $r_e = 100$, $ninst = 1$, $m = 10$ y n aumentando de a 1. Aumentamos solo n para aumentar la dimension de la matriz A dado que lo que relevante al tiempo de ejecución final del algoritmo es la dimension total, no de donde proviene la misma.

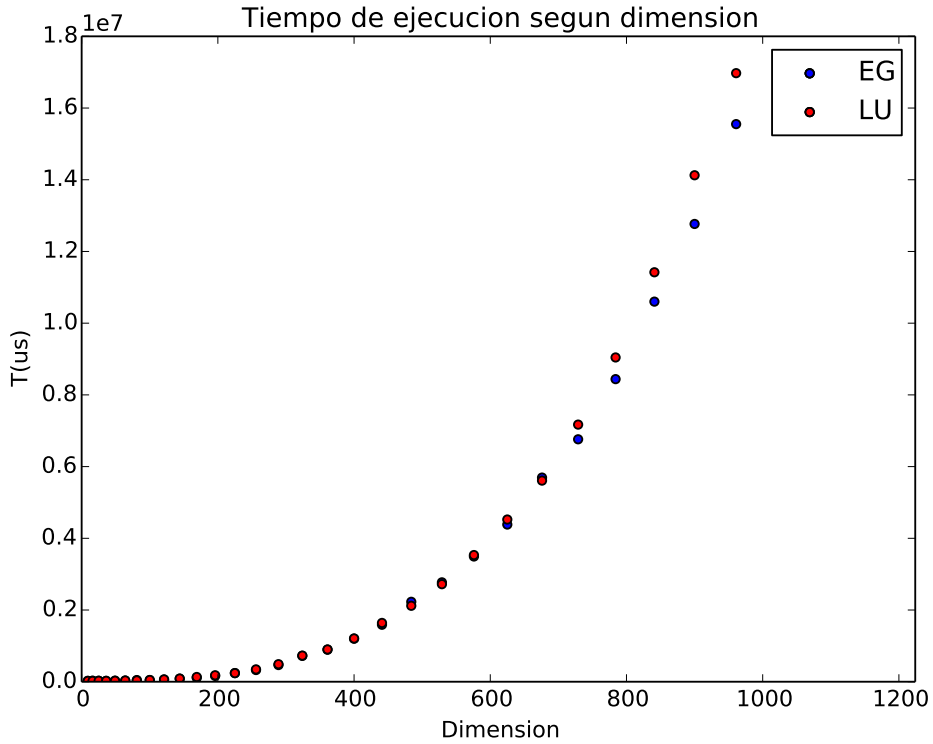

Figura 5: Tiempos de ejecución según dimensión, EG vs LU.

Como se puede observar en estos gráficos, ambos algoritmos tienen casi los mismos tiempos de ejecución para una dimension dada. Esto coincide con la teoría.

3.2.2. Tiempo de ejecución variando número de instancias

Para este experimento la idea es ver la diferencia en performance de la eliminación gausiana y la factorización LU a medida que aumenta el numero de instancias, es decir, a medida que cambiamos la matriz b. Aunque ambos algoritmos pertenecen a $\mathcal{O}(n^3)$, esperamos que la factorización LU sea superior a la eliminación gausiana dado que para la factorización LU el costo adicional de resolver otras instancias es de $\mathcal{O}(n^2)$ (es un simple producto de matrices) mientras que la eliminación gausiana debe repetir todo el procedimiento en $\mathcal{O}(n^3)$.

Para generar las instancias utilizadas en el gráfico, estos fueron los valores que utilizamos: $m = 15$, $n = 15$, $r_i = 10$, $r_e = 100$, $ninst = 1$, $ninst$ aumentando de a 1.

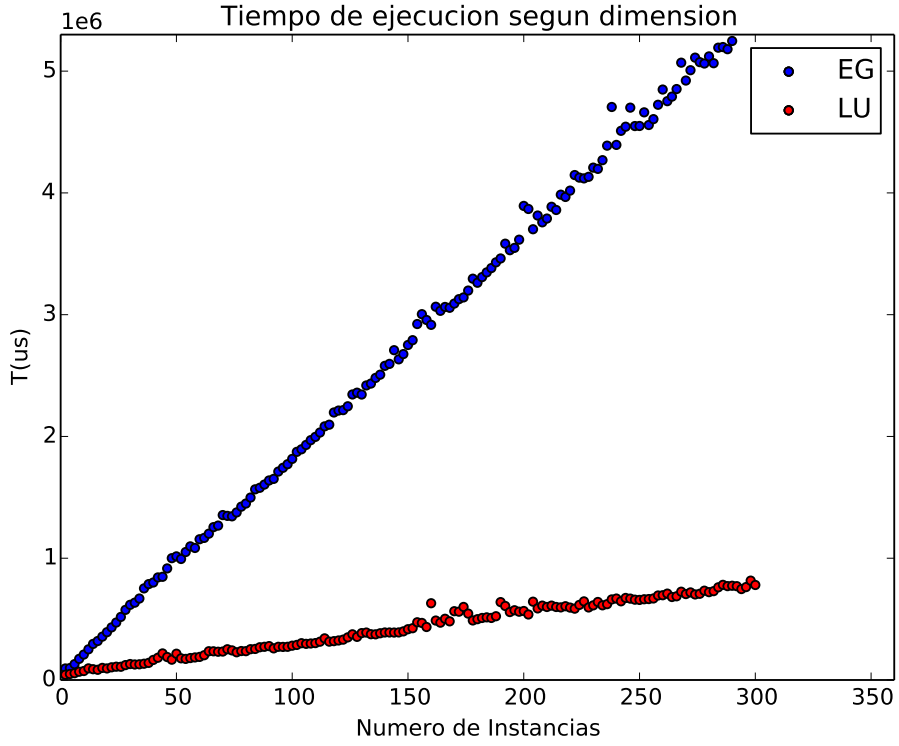

Figura 6: Tiempos de ejecución variando numero de instancias, EG vs LU.

Coincidiendo nuevamente con la teoría, aquí es donde se pueden ver efectivamente las ganancias de la factorización LU. Aunque ambos algoritmos son cúbicos, en este problema la EG se ejecuta en $\mathcal{O}(ninst \times n^3)$ mientras que la factorización LU en $\mathcal{O}(n^3 + ninst \times n^2) \in \mathcal{O}(n^3)$. Esto se puede ver claramente en el gráfico. Como n esta fijo, es razonable que los tiempos de ejecución formen dos rectas.

3.3. Comportamiento del Sistema

3.4. Isoterma Empírica

La idea de los siguientes experimentos es ver como varia la diferencia entre la isoterma empírica y la isoterma numérica a medida que cambiamos el nivel de granularidad, es decir, a medida que cambiamos la cantidad de radios y ángulos. Haremos este análisis para nuestros dos algoritmos de búsqueda.

Variando el numero de radios

En este experimento vamos a evaluar la calidad de las isotermas a medida que aumentamos el numero de radios. A priori, esperábamos que la calidad de la solución fuese monótona creciente con el nivel de granularidad, en este caso particular en la cantidad de radios. A su vez, pensábamos que hacer un promedio pesado nos iba a acercar mas a la isoterma empírica que simplemente utilizando el método lower.

Para este experimento utilizamos instancias con los siguientes parámetros: $m = 8$, $n = 2$, $r_i = 10$, $r_e = 100$, $ninst = 1$, con m aumentando de a 4.

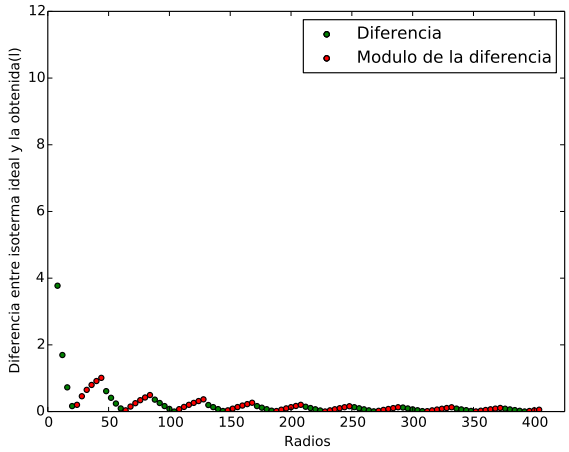

Figura 7: Algoritmo: Lower

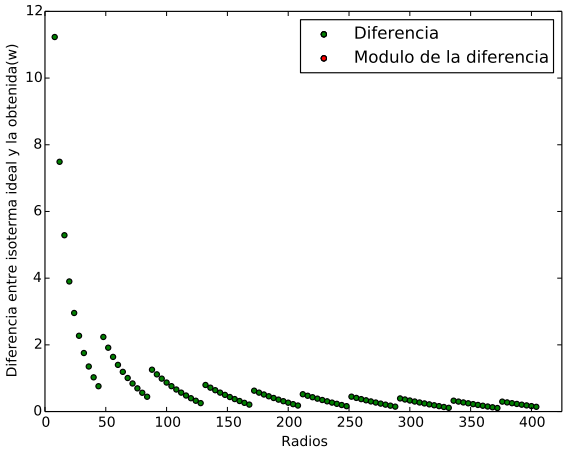

Figura 8: Algoritmo: Weighted

En contra de lo que pensábamos que iba a suceder, la calidad de la solución no es monótona creciente con el nivel de granularidad. A su vez, se puede ver que el algoritmo de tomar la isoterma justo menor al que se busca supera ampliamente al método weighted. En un principio pensábamos que nuestros experimentos podían estar mal, pero la realidad es que existe una explicación para estos resultados.

Cuando discretizamos el dominio de búsqueda en coordenadas polares y resolvemos el sistema lineal, lo que estamos haciendo es básicamente buscar una aproximación de la temperatura en cada punto de la discretización. Esta aproximación cambia a medida que variamos la granularidad. A mayor granularidad, la aproximación de la temperatura en cierto punto dado por cualquiera de nuestros algoritmos no es necesariamente mejor. La única forma de garantizar esto es que la nueva discretización no solo pase por los puntos que tenia la vieja discretización, si no que también agregue nuevos puntos. Dada la definición de nuestro problema, esto solo sucede cuando la granularidad en cualquiera de sus dimensiones aumenta en múltiplos de 2. Al observar estos múltiplos, aquí si podremos observar como la isoterma se comporta de la forma en la que nosotros originalmente habíamos esperado.

Otro factor sumamente importante que debemos considerar es el `trade off` entre calidad de la solución y tiempo de computo (Figura 5). A medida que aumentamos la cantidad de radios, la ganancia adicional en términos de calidad de la solución es cada vez menor. Sin embargo, dado que los algoritmos para resolver el sistema son $\mathcal{O}(n^3)$, el tiempo de computo sube de forma cubica. A fines prácticos no es una buena idea tender la granularidad a infinito dado que aunque la solución numérica tienda a la solución empírica, el tiempo de ejecución seria extremadamente alto. Por supuesto que esto depende de la tolerancia de error que tenga el usuario.

En cuanto a la razón de por que el método lower se comporta mejor que el weighted, conjeturamos que esto sucede debido a que quizás la temperatura no se propaga de forma uniforme en la pared del horno. El método weighted hace un promedio pesado asumiendo uniformidad, y quizá este supuesto es incorrecto.

Hacer un testeo que muestre si el calor se propaga de forma uniforme. En clase te lo comente. Duplicas la temperatura externa, te fijas en cuanto se mueve la isoterma.

Variando el numero de ángulos

Para este experimento, utilizaremos instancias con las siguientes características: $m = 20$, $n = 4$, $r_i = 10$, $r_e = 100$, $ninst = 1$, con n aumentando de a 4.

A partir de las conclusiones del experimento anterior, ya no esperamos que la calidad de la solución mejore monotonamente a medida que aumentamos la cantidad de ángulos.

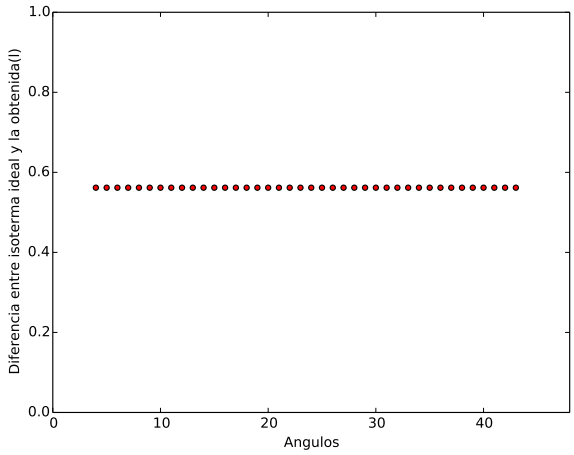

Figura 9: Algoritmo: Lower

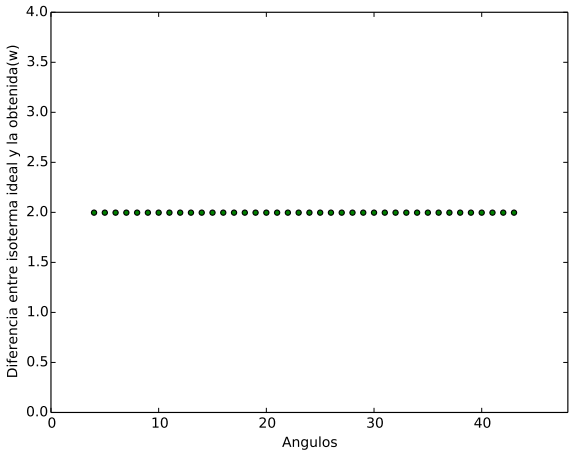

Figura 10: Algoritmo: Weighted

Como se puede observar, ambos algoritmos tienen una performance sumamente similar y no se observan mejoras significativas en la calidad de las isotermas a medida que aumenta la cantidad de ángulos. Esto se debe a que nuestro caso de prueba fue un caso simétrico, donde para cualquier angulo las temperaturas para cualquier radio son idénticas. Esto nos trae una conclusión bastante importante: en casos que sean bastante simétricos, formar un caso simétrico tomando como temperatura externa el máximo, tomando solo 1 angulo y aumentando solo la cantidad de radios es la estrategia optima para evaluar la integridad estructural de un horno. Bajamos el tiempo de ejecución al no tener que evaluar todos los ángulos explotando la simetría del problema y solo aumentamos la granularidad en la dimension m relevante.

Variar el numero de ángulos cuando la temperatura es sumamente uniforme en el exterior puede tener efectos significativos sobre la calidad de la solución, aunque no tan significativos como la de variar los radios.

Experimentar un poco con esta proposición?

4. Conclusiones

El modelado de problemas utilizando sistemas de ecuaciones lineales de la forma $Ax = b$ es sumamente útil, principalmente debido a que se presentan de forma frecuente y sus metodologías de resolución han sido ampliamente estudiadas en la literatura. Para resolver este tipo de problemas, en este trabajo sólo utilizamos la eliminación gausiana y la factorización LU, aunque cabe resaltar que existen muchos mas métodos que toman provecho de la estructura del problema en cuestión y logran una complejidad temporal y espacial superior. Para problemas 'chicos', estos factores pueden no ser muy importantes, aunque en la practica muchas veces se presentan sistemas donde si se tornan relevantes.

En el desarollo de este trabajo practico utilizamos dos métodos de resolución clásicos, la eliminación gaussiana y la factorización LU. Estos métodos son conocidos por tener una complejidad en $\mathcal{O}(n^3)$. Sin embargo, al cambiar el vector del sistema lineal b, la eliminación gaussiana pierde en su proceso algorítmico información sumamente relevante que podría ser utilizada para evitar las operaciones entre filas utilizadas para encontrar un sistema triangular superior. Aquí es donde entra la factorización LU, que guarda la información de las operaciones entre filas y logra resolver instancias adicionales en $\mathcal{O}(n^2)$. Este resultado teórico fue luego confirmado experimentalmente. No obstante, a priori, no toda matriz cuadrada admite esta factorización; esto nos ha a llevado a estudiar un poco mas la naturaleza del sistema. En concecuencia, tras develar algunas propiedades, hemos podido demostrar que la aplicación de la eliminación gaussiana reduce nuestra matriz a una triangular superior con elementos distintos de cero en su diagonal (de lo que se deduce que es no singunlar) y sin necesidad de aplicar pivoteo total ni parcial. Como hemos visto en las clases teóricas, dichas hipotesis nos llevan a concluir que el sistema siempre tendrá una única solución, y en particular que posee factorización LU.

Otra carácteristica que pudimos comprobar en la matriz del sistema es su condición de *banda-n*, esto a priori, puedo no representarnos ventajas para los dos metodos empleados; no obstante somos concientes que usando otras estructuras de datos, y aprovechando la redundancia de ceros acumulados en la esquina superior izquierda e inferior derecha, se podría reducir para casos lo suficientemente grandes, la complejidad espacial (almacenando sólo aquellos elementos que se encuentre dentro del rango de la banda) y también la complejidad temporal, pudiendo ignorar aquellas filas en las cuales el elemento de la misma columna del pivot actual se encuentra fuera del rango de la banda.

Para resolver el problema del horno, tuvimos en primer lugar que buscar alguna manera de buscar una isoterma dentro de una aproximación numérica de las temperaturas. Esto en principio es problemático, dado que no siempre las aproximaciones numéricas son buenas, y ademas porque es necesario definir algún tipo de criterio de interpolación al no poder encontrar temperaturas exactamente iguales a las que estamos buscando. Por esta razón definimos dos métodos y luego llevamos a cabo experimentaciones numéricas para definir cual era mejor.

Nos sorprendió notar que aumentar la granularidad no necesariamente aumenta la calidad de las isotermas encontradas. Al aumentar la cantidad de radios, esto solo sucede cuando aumentamos la granularidad en múltiplos de 2, dado que bajo estas condiciones todos los puntos anteriores al aumento se encuentran contenidos en la nueva discretización.

Por otro lado, muchas veces nos encontramos ante problemas simétricos, donde la temperatura externa es uniforme para todos los ángulos. Aquí es donde notamos que podemos explotar esta simetría y solo considerar un angulo para el sistema. Utilizando esto podemos lograr una presición mucho mayor de las isotermas para una dada dimension de la matriz A.

5. Apéndice A: Enunciado

Laboratorio de Métodos Numéricos - Segundo Cuatrimestre 2015
Trabajo Práctico Número 1: Con 15 θs discretizo alto horno...

Introducción

Consideremos la sección horizontal de un horno de acero cilíndrico, como en la Figura 1. El sector A es la pared del horno, y el sector B es el horno propiamente dicho, en el cual se funde el acero a temperaturas elevadas. Tanto el borde externo como el borde interno de la pared forman círculos. Suponemos que la temperatura del acero dentro del horno (o sea, dentro de B) es constante e igual a 1500^oC.

Tenemos sensores ubicados en la parte externa del horno para medir la temperatura de la pared externa del mismo, que habitualmente se encuentra entre 50^oC y 200^oC. El problema que debemos resolver consiste en estimar la isoterma de 500^oC dentro de la pared del horno, para estimar la resistencia de la misma. Si esta isoterma está demasiado cerca de la pared externa del horno, existe peligro de que la estructura externa de la pared colapse.

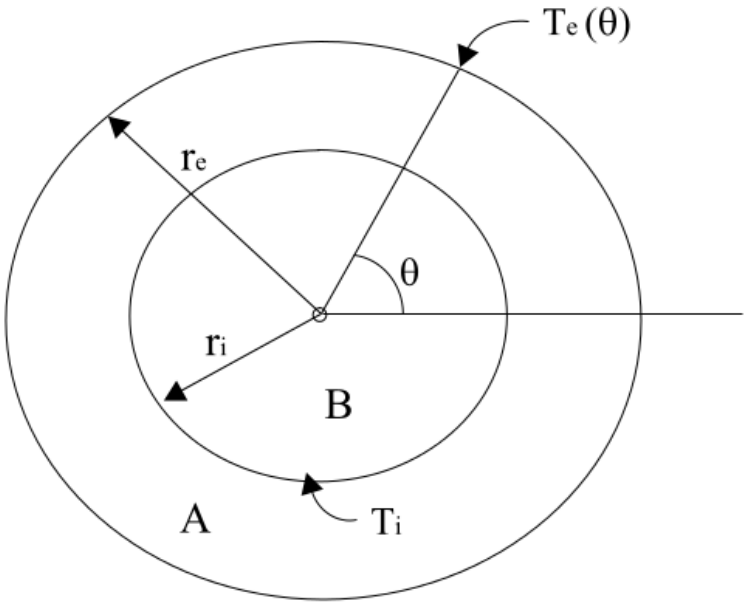

Figura 11: Sección circular del horno

El objetivo del trabajo práctico es implementar un programa que calcule la isoterma solicitada, conociendo las dimensiones del horno y las mediciones de temperatura en la pared exterior.

El Modelo

Sea $r_e \in \mathbb{R}$ el radio exterior de la pared y sea $r_i \in \mathbb{R}$ el radio interior de la pared. Llamemos $T(r, \theta)$ a la temperatura en el punto dado por las coordenadas polares (r, θ), siendo r el radio y θ el ángulo polar de dicho punto. En el estado estacionario, esta temperatura satisface la ecuación del calor:

$$\frac{\partial^2 T(r,\theta)}{\partial r^2} + \frac{1}{r}\frac{\partial T(r,\theta)}{\partial r} + \frac{1}{r^2}\frac{\partial^2 T(r,\theta)}{\partial \theta^2} = 0 \tag{1}$$

Si llamamos $T_i \in \mathbb{R}$ a la temperatura en el interior del horno (sector B) y $T_e : [0, 2\pi] \to \mathbb{R}$ a la función de temperatura en el borde exterior del horno (de modo tal que el punto (r_e, θ) tiene temperatura $T_e(\theta)$), entonces tenemos que

$$T(r, \theta) = T_i \quad \textit{para todo punto } (r, \theta) \textit{ con } r \leq r_i \tag{2}$$

$$T(r_e, \theta) = T_e(\theta) \quad \textit{para todo punto } (r_e, \theta) \tag{3}$$

El problema en derivadas parciales dado por la primera ecuación con las condiciones de contorno presentadas recientemente, permite encontrar la función T de temperatura en el interior del horno (sector A), en función de los datos mencionados en esta sección.

Para resolver este problema computacionalmente, discretizamos el dominio del problema (el sector A) en coordenadas polares. Consideramos una partición $0 = \theta_0 < \theta_1 < ... < \theta_n = 2\pi$ en n ángulos discretos con $\theta_k - \theta_{k-1} = \Delta\theta$ para $k = 1, ..., n$, y una partición $r_i = r_0 < r_1 < ... < r_m = r_e$ en $m + 1$ radios discretos con $r_j - r_{j-1} = \Delta r$ para $j = 1, ..., m$.

El problema ahora consiste en determinar el valor de la función T en los puntos de la discretización (r_j, θ_k) que se encuentren dentro del sector A. Llamemos $t_{jk} = T(r_j, \theta_k)$ al valor (desconocido) de la función T en el punto (r_j, θ_k).

Para encontrar estos valores, transformamos la ecuación (1) en un conjunto de ecuaciones lineales sobre las incógnitas t_{jk}, evaluando (1) en todos los puntos de la discretización que se encuentren dentro del sector A. Al hacer esta evaluación, aproximamos las derivadas parciales de T en (1) por medio de las siguientes fórmulas de diferencias finitas:

$$\frac{\partial^2 T(r,\theta)}{\partial r^2}(r_j, \theta_k) \cong \frac{t_{j-1,k} - 2t_{jk} + t_{j+1,k}}{(\Delta r)^2} \tag{4}$$

$$\frac{\partial T(r,\theta)}{\partial r}(r_j, \theta_k) \cong \frac{t_{j,k} - t_{j-1,k}}{\Delta r} \tag{5}$$

$$\frac{\partial^2 T(r,\theta)}{\partial \theta^2}(r_j, \theta_k) \cong \frac{t_{j,k-1} - 2t_{jk} + t_{j,k+1}}{(\Delta \theta)^2} \tag{6}$$

Es importante notar que los valores de las incógnitas son conocidos para los puntos que se encuentran sobre el borde exterior de la pared, y para los puntos que se encuentren dentro del sector B. Al realizar este procedimiento, obtenemos un sistema de ecuaciones lineales que modela el problema discretizado. La resolución de este sistema permite obtener una aproximación de los valores de la función T en los puntos de la discretización.

Enunciado

Se debe implementar un programa en `C` o `C++` que tome como entrada los parámetros del problema (r_i, r_e, $m+1$, n, valor de la isoterma buscada, T_i, $T_e(\theta)$) que calcule la temperatura dentro de la pared del horno utilizando el modelo propuesto en la sección anterior y que encuentre la isoterma buscada en función del resultado obtenido del sistema de ecuaciones. El método para determinar la posición de la isoterma queda a libre elección de cada grupo y debe ser explicado en detalle en el informe.

El programa debe formular el sistema obtenido a partir de las ecuaciones (1) - (6) y considerar dos métodos posibles para su resolución: mediante el algoritmo clásico de Eliminación Gaussiana y la Factorización LU. Finalmente, el programa escribirá en un archivo la solución obtenida con el formato especificado en la siguiente sección.

Como ya se ha visto en la materia, no es posible aplicar los métodos propuestos para la resolución a cualquier sistema de ecuaciones. Sin embargo, la matriz del sistema considerado en el presente trabajo cumple con ser diagonal dominante (no estricto) y que, ordenando las variables y ecuaciones convenientemente, es posible armar un sistema de ecuaciones cuya matriz posee la propiedad de ser *banda*. Luego, se pide demostrar (o al menos dar un esquema de la demostración) el siguiente resultado e incluirlo en el informe:

Proposición 5.1 *Sea $A \in \mathbb{R}^{n \times n}$ la matriz obtenida para el sistema definido por (1)-(6). Demostrar que es posible aplicar Eliminación Gaussiana sin pivoteo.*[1]

La solución del sistema de ecuaciones permitirá saber la temperatura en los puntos de la discretización. Sin embargo, nuestro interés es calcular la isoterma 500, para poder determinar si la estructura se encuentra en peligro. Luego, se pide lo siguiente:

- Dada la solución del sistema de ecuaciones, proponer una forma de estimar en cada ángulo de la discretización la posición de la isoterma 500.
- En función de la aproximación de la isoterma, proponer una forma (o medida) a utilizar para evaluar la peligrosidad de la estructura en función de la distancia a la pared externa del horno.

En función de la experimentación, se busca realizar dos estudios complementarios: por un lado, analizar cómo se comporta el sistema y, por otro, cuáles son los requerimientos computacionales de los métodos. Se pide como mínimo realizar los siguientes experimentos:

1. Comportamiento del sistema.
    - Considerar al menos dos instancias de prueba, generando distintas discretizaciones para cada una de ellas y comparando la ubicación de la isoterma buscada respecto de la pared externa del horno. Se sugiere presentar gráficos de temperatura o curvas de nivel para los mismos, ya sea utilizando las herramientas provistas por la cátedra o implementando sus propias herramientas de graficación.

[1] Sugerencia: Notar que la matriz es diagonal dominante (no estrictamente) y analizar qué sucede al aplicar un paso de Eliminación Gaussiana con los elementos de una fila.

- Estudiar la proximidad de la isoterma buscada respecto de la pared exterior del horno en función de distintas granularidades de discretización y las condiciones de borde.

2. Evaluación de los métodos.

   - Analizar el tiempo de cómputo requerido para obtener la solución del sistema en función de la granularidad de la discretización. Se sugiere presentar los resultados mediante gráficos de tiempo de cómputo en función de alguna de las variables del problema.
   - Considerar un escenario similar al propuesto en el experimento 1. pero donde las condiciones de borde (i.e., T_i y $T_e(\theta)$) cambian en distintos instantes de tiempo. En este caso, buscamos obtener la secuencia de estados de la temperatura en la pared del horno, y la respectiva ubicación de la isoterma especificada. Para ello, se considera una secuencia de *ninst* vectores con las condiciones de borde, y las temperaturas en cada estado es la solución del correspondiente sistema de ecuaciones. Se pide formular al menos un experimento de este tipo, aplicar los métodos de resolución propuestos de forma conveniente y compararlos en términos de tiempo total de cómputo requerido para distintos valores de *ninst*.

De manera opcional, aquellos grupos que quieran ir un poco más allá pueden considerar trabajar y desarrollar alguno(s) de los siguientes puntos extra:

1. Notar que el sistema resultante tiene estructura *banda*. Proponer una estructura para aprovechar este hecho en términos de la *complejidad espacial* y como se adaptarían los algoritmos de Eliminación Gaussiana y Factorización LU para reducir la cantidad de operaciones a realizar.

2. Implementar dicha estructura y las adaptaciones necesarias para el algoritmo de Eliminación Gaussiana.

3. Implementar dicha estructura y las adaptaciones necesarias para el algoritmo de Factorización LU.

Finalmente, se deberá presentar un informe que incluya una descripción detallada de los métodos implementados y las decisiones tomadas, el método propuesto para el cálculo de la isoterma buscada y los experimentos realizados, junto con el correspondiente análisis y siguiendo las pautas definidas en el archivo `pautas.pdf`.

Programa y formato de archivos

Se deberán entregar los archivos fuentes que contengan la resolución del trabajo práctico. El ejecutable tomará tres parámetros por línea de comando, que serán el archivo de entrada, el archivo de salida, y el método a ejectutar (0 EG, 1 LU).

El archivo de entrada tendrá la siguiente estructura:

- La primera línea contendrá los valores r_i, r_e, $m+1$, n, *iso*, *ninst*, donde *iso* representa el valor de la isoterma buscada y *ninst* es la cantidad de instancias del problema a resolver para los parámetros dados.

- A continuación, el archivo contendrá *ninst* líneas, cada una de ellas con $2n$ valores, los primeros n indicando los valores de la temperatura en la pared interna, i.e., $T_i(\theta_0), T_i(\theta_1), \ldots, T_i(\theta_{n-1})$, seguidos de n valores de la temperatura en la pared externa, i.e., $T_e(\theta_0)$,$T_e(\theta_1)$,...,$T_e(\theta_{n-1})$.

El archivo de salida obligatorio tendrá el vector solución del sistema reportando una componente del mismo por línea. En caso de $ninst > 1$, los vectores serán reportados uno debajo del otro.

Junto con el presente enunciado, se adjunta una serie de scripts hechos en `python` y un conjunto instancias de test que deberán ser utilizados para la compilación y un testeo básico de la implementación. Se recomienda leer el archivo `README.txt` con el detalle sobre su utilización.

Fechas de entrega

- *Formato Electrónico:* Jueves 3 de Septiembre de 2015, hasta las 23:59 hs, enviando el trabajo (informe + código) a la dirección `metnum.lab@gmail.com`. El subject del email debe comenzar con el texto `[TP1]` seguido de la lista de apellidos de los integrantes del grupo.

- *Formato físico:* Viernes 4 de Septiembre de 2015, de 17:30 a 18:00 hs.

Importante: El horario es estricto. Los correos recibidos después de la hora indicada serán considerados re-entrega. Los grupos deben ser de 3 o 4 personas, sin excepción. Es indispensable que los trabajos pasen satisfactoriamente los casos de test provistos por la cátedra.

6. Apéndice B: Código

6.1. matrix.h

```
1  /*
2   * File:   matrix.h
3   * Author: Federico
4   *
5   * Created on August 16, 2015, 9:54 PM
6   */
7
8  #ifndef MATRIX_H
9  #define MATRIX_H
10
11 #include <algorithm>
12 #include <math.h>
13 #include <vector>
14 #include <stdio.h>
15
16 using namespace std;
17
18 // La matriz respeta la notacion de la catedra, es decir, el primer subindice
19 // es la fila y el segundo es la columna
20
21 template<class T>
22 class Matrix {
23     public:
24         Matrix();
25         Matrix(int rows); // Columnas impllicitas (col = 1)
26         Matrix(int rows, int col);
27         Matrix(int rows, int col, const T& init);
28         Matrix(const Matrix<T>& other);
29         ~Matrix();
30
31         Matrix<T>& operator=(const Matrix<T>& other);
32         Matrix<T> operator*(const Matrix<T>& other);
33         Matrix<T>& operator*=(const Matrix<T>& other);
34         Matrix<T> operator+(const Matrix<T>& other);
35         Matrix<T>& operator+=(const Matrix<T>& other);
36         Matrix<T> operator-(const Matrix<T>& other);
37         Matrix<T>& operator-=(const Matrix<T>& other);
38
39         Matrix<T> operator*(const T& scalar);
40         Matrix<T> operator/(const T& scalar);
41
42         T& operator()(int a, int b);
43         const T& operator()(const int a, const int b) const;
44         T& operator()(int a);
45         const T& operator()(const int a) const;
46
47         int rows();
48         int columns();
49         void printMatrix();
50
51     private:
52         vector<vector<T> > _values;
53         int _rows;
54         int _columns;
55
```

```
56  };
57
58  template<class T>
59  Matrix<T>::Matrix()
60      : _values(1), _rows(1), _columns(1)
61  {
62      _values[0].resize(1);
63  }
64
65  template<class T>
66  Matrix<T>::Matrix(int rows)
67      : _values(rows), _rows(rows), _columns(1)
68  {
69      for(int i = 0; i < rows; i++) {
70          _values[i].resize(1);
71      }
72  }
73
74  template<class T>
75  Matrix<T>::Matrix(int rows, int col)
76      : _values(rows), _rows(rows), _columns(col)
77  {
78      for(int i = 0; i < rows; i++) {
79          _values[i].resize(col);
80      }
81  }
82
83  template<class T>
84  Matrix<T>::Matrix(int rows, int col, const T& init)
85      : _values(rows), _rows(rows), _columns(col)
86  {
87      for(int i = 0; i < rows; i++) {
88          _values[i].resize(col, init);
89      }
90  }
91
92  template<class T>
93  Matrix<T>::Matrix(const Matrix<T>& other)
94      : _values(other._values), _rows(other._rows), _columns(other._columns)
95  {}
96
97  template<class T>
98  Matrix<T>::~Matrix() {}
99
100 template<class T>
101 Matrix<T>& Matrix<T>::operator=(const Matrix<T>& other) {
102   if (&other == this)
103     return *this;
104
105   int new_rows = other._rows;
106   int new_columns = other._columns;
107
108   _rows = new_rows;
109   _columns = new_columns;
110
111   _values.resize(new_rows);
112   for (int i = 0; i < new_columns; i++) {
113       _values[i].resize(new_columns);
114   }
```

```
115
116   for(int i = 0; i < new_rows; i++) {
117     for(int j = 0; j < new_columns; j++) {
118       _values[i][j] = other(i, j);
119     }
120   }
121
122   return *this;
123 }
124
125 template<class T>
126 Matrix<T> Matrix<T>::operator*(const Matrix<T>& other) {
127     // ASUME QUE LAS DIMENSIONES DAN
128     Matrix<T> result(_rows, other._columns);
129
130     int innerDim = _columns; // Tambien podria ser other._rows
131
132     for(int i = 0; i < result._rows; i++) {
133         for(int j = 0; j < result._columns; j++) {
134             result(i,j) = 0;
135             for(int k = 0; k < innerDim; k++) {
136                 result(i,j) += _values[i][k] * other(k,j);
137             }
138         }
139     }
140
141     return result;
142 }
143
144 template<class T>
145 Matrix<T>& Matrix<T>::operator*=(const Matrix<T>& other) {
146     Matrix<T> result = (*this) * other;
147     (*this) = result;
148     return (*this);
149 }
150
151 template<class T>
152 Matrix<T> Matrix<T>::operator+(const Matrix<T>& other) {
153     // ASUME QUE LAS DIMENSIONES DAN
154     Matrix<T> result(_rows, other._columns);
155
156     for(int i = 0; i < result._rows; i++) {
157         for(int j = 0; j < result._columns; j++) {
158             result(i,j) = _values[i][j] + other(i,j);
159         }
160     }
161
162     return result;
163 }
164
165 template<class T>
166 Matrix<T>& Matrix<T>::operator+=(const Matrix<T>& other) {
167     Matrix<T> result = (*this) + other;
168     (*this) = result;
169     return (*this);
170 }
171
172 template<class T>
173 Matrix<T> Matrix<T>::operator-(const Matrix<T>& other) {
```

```
174     // ASUME QUE LAS DIMENSIONES DAN
175     Matrix<T> result(_rows, other._columns);
176 
177     for(int i = 0; i < result._rows; i++) {
178         for(int j = 0; j < result._columns; j++) {
179             result(i,j) = _values[i][j] - other(i,j);
180         }
181     }
182 
183     return result;
184 }
185 
186 template<class T>
187 Matrix<T>& Matrix<T>::operator-=(const Matrix<T>& other) {
188     Matrix<T> result = (*this) - other;
189     (*this) = result;
190     return (*this);
191 }
192 
193 template<class T>
194 Matrix<T> Matrix<T>::operator*(const T& scalar) {
195     Matrix<T> result(_rows, _columns);
196 
197     for(int i = 0; i < result._rows; i++) {
198         for(int j = 0; j < result._columns; j++) {
199             result(i,j) = _values[i][j] * scalar;
200         }
201     }
202 
203     return result;
204 }
205 
206 template<class T>
207 Matrix<T> Matrix<T>::operator/(const T& scalar) {
208     Matrix<T> result(_rows, _columns);
209 
210     for(int i = 0; i < result._rows; i++) {
211         for(int j = 0; j < result._columns; j++) {
212             result(i,j) = _values[i][j] / scalar;
213         }
214     }
215 
216     return result;
217 }
218 
219 template<class T>
220 T& Matrix<T>::operator ()(int a, int b) {
221     return _values[a][b];
222 }
223 
224 template<class T>
225 const T& Matrix<T>::operator ()(const int a, const int b) const {
226     return _values[a][b];
227 }
228 
229 template<class T>
230 T& Matrix<T>::operator ()(int a) {
231     return _values[a][0];
232 }
```

```
233
234 template<class T>
235 const T& Matrix<T>::operator ()(const int a) const {
236     return _values[a][0];
237 }
238
239 template<class T>
240 int Matrix<T>::rows() {
241     return _rows;
242 }
243
244 template<class T>
245 int Matrix<T>::columns() {
246     return _columns;
247 }
248
249 template<class T>
250 void Matrix<T>::printMatrix() {
251     for(int i = 0; i < _rows; i++) {;
252         for(int j = 0; j < _columns; j++) {
253             printf("%3.3f ", _values[i][j]);
254             // cout << _values[i][j] << " ";
255         }
256         cout << endl;
257     }
258     cout << endl;
259 }
260
261 #endif  /* MATRIX_H */
```

6.2. eqsys.h

```
1  /*
2   * File:   eqsys.h
3   * Author: Federico
4   *
5   * Created on August 17, 2015, 5:57 PM
6   */
7
8  #ifndef EQSYS_H
9  #define EQSYS_H
10
11 #include <algorithm>
12 #include <math.h>
13 #include <vector>
14 #include "matrix.h"
15
16 template<class T>
17 class EquationSystemLU {
18     public:
19         EquationSystemLU(const Matrix<T>& inicial);
20
21         Matrix<T> solve(Matrix<T>& b_values);
22
23     private:
24         Matrix<T> lower;
25         Matrix<T> upper;
26         bool isPermutated;
27         Matrix<T> permutation;
```

```
28  };
29
30  template<class T>
31  EquationSystemLU<T>::EquationSystemLU(const Matrix<T>& inicial)
32      : upper(inicial), isPermutated(false)
33  {
34      T coef;
35      int i, j, k, l;
36
37      // Armar la matriz lower
38      lower = Matrix<T>(upper.rows(), upper.columns(), 0);
39
40      for(i = 0; i < upper.columns(); i++) {
41          for(j = i + 1; j < upper.rows(); j++) {
42              if(upper(i, i) == 0) {
43                  // Hay que buscar la proxima fila sin cero
44                  for(k = i + 1; k < upper.rows(); k++) {
45                      if(upper(k, i) != 0) {
46                          break;
47                      }
48                  }
49
50                  if(k == upper.rows()) { // No hay filas para permutar
51                      abort();
52                  } else {
53                      if(!isPermutated){
54                          // Generamos la matriz de permutacion con uno en la diagonal
55                          isPermutated = true;
56                          permutation = Matrix<T>(upper.rows(), upper.columns(), 0);
57
58                          for(l = 0; l < permutation.rows(); l++) {
59                              permutation(l,l) = 1;
60                          }
61                      }
62                      // Permutamos las filas
63                      for(l = 0; l < permutation.columns(); l++) {
64                          if(l == k) {
65                              permutation(i, l) = 1;
66                          } else {
67                              permutation(i, l) = 0;
68                          }
69                          if(l == i) {
70                              permutation(k, l) = 1;
71                          } else {
72                              permutation(k, l) = 0;
73                          }
74                      }
75                      // Hacemos el producto para efectivamente permutar
76                      upper = permutation * upper;
77                      lower = permutation * lower;
78                  }
79              }
80
81              // Calculamos y guardamos el coeficiente
82              // cout << upper(j,i) << " , " << upper(i,i) << endl;
83              coef = upper(j, i) / upper(i, i);
84              lower(j, i) = coef;
85              // Colocamos cero en la columna bajo la diagonal
86              upper(j,i) = 0;
```

```
87              for(k = i + 1; k < upper.columns(); k++) {
88                  upper(j, k) = upper(j, k) - coef * upper(i, k);
89              }
90          }
91      }
92      // Agrego la diagonal de unos a lower
93      for(i = 0; i < lower.rows(); i++){
94          lower(i,i) = 1;
95      }
96  }
97
98  template<class T>
99  Matrix<T> EquationSystemLU<T>::solve(Matrix<T>& b_values) {
100
101     Matrix<T> temp_values = Matrix<T>(b_values);
102     Matrix<T> y_values = Matrix<T>(b_values.rows());
103     Matrix<T> x_values = Matrix<T>(b_values.rows());
104
105     if(isPermutated) {
106         temp_values = permutation * temp_values;
107     }
108
109     for(int i = 0; i < temp_values.rows(); i++) {
110         for (int j = 0; j < i; j++) {
111             temp_values(i) -= y_values(j) * lower(i,j);
112         }
113         if(i == 0) {
114             y_values(0) = temp_values(0) / lower(0,0); // Calculo aparte el primer
                valor
115         } else {
116             y_values(i) = temp_values(i) / lower(i,i);
117         }
118     }
119
120     temp_values = y_values;
121     for(int i = temp_values.rows() - 1; i >= 0; i--) {
122         for (int j = temp_values.rows() - 1; j > i; j--) {
123             temp_values(i) -= x_values(j) * upper(i,j);
124         }
125         if(i == x_values.rows() - 1) {
126             x_values(x_values.rows() - 1) = temp_values(temp_values.rows() - 1) /
                upper(upper.rows() - 1, upper.columns() - 1);
127         } else {
128             x_values(i) = temp_values(i) / upper(i,i);
129         }
130     }
131
132     //if(isPermutated) {
133     //    x_values = permutation * x_values;
134     //}
135
136     return x_values;
137
138 }
139
140
141
142 template<class T>
143 class EquationSystem{
```

```
144     public:
145         EquationSystem(const Matrix<T>& inicial);
146 
147         Matrix<T> solve(const Matrix<T>& b_values);
148 
149     private:
150         Matrix<T> _matrix;
151 };
152 
153 template<class T>
154 EquationSystem<T>::EquationSystem(const Matrix<T>& inicial)
155     : _matrix(inicial)
156 {}
157 
158 template<class T>
159 Matrix<T> EquationSystem<T>::solve(const Matrix<T>& b_values) {
160     T coef;
161     int i, j, k, l;
162     bool isPermutated;
163     Matrix<T> temp_matrix(_matrix);
164     Matrix<T> temp_values(b_values);
165     Matrix<T> permutation;
166 
167     for(i = 0; i < temp_matrix.columns(); i++) {
168         for(j = i + 1; j < temp_matrix.rows(); j++) {
169             if(temp_matrix(i, i) == 0) {
170                 // Hay que buscar la proxima fila sin cero
171                 for(k = i + 1; k < temp_matrix.rows(); k++) {
172                     if(temp_matrix(k, i) != 0) {
173                         break;
174                     }
175                 }
176 
177                 if(k == temp_matrix.rows()) { // No hay filas para permutar
178                     abort();
179                 } else {
180                     if(!isPermutated){
181                         // Generamos la matriz de permutacion con uno en la diagonal
182                         isPermutated = true;
183                         permutation = Matrix<T>(temp_matrix.rows(), temp_matrix.
                                columns(), 0);
184 
185                         for(l = 0; l < permutation.rows(); l++) {
186                             permutation(l,l) = 1;
187                         }
188                     }
189                     // Permutamos las filas
190                     for(l = 0; l < permutation.columns(); l++) {
191                         if(l == k) {
192                             permutation(i, l) = 1;
193                         } else {
194                             permutation(i, l) = 0;
195                         }
196                         if(l == i) {
197                             permutation(k, l) = 1;
198                         } else {
199                             permutation(k, l) = 0;
200                         }
201                     }
```

```
202                     // Hacemos el producto para efectivamente permutar
203                     temp_matrix = permutation * temp_matrix;
204                     temp_values = permutation * temp_values;
205                 }
206             }
207 
208             // Calculamos y guardamos el coeficiente
209             coef = temp_matrix(j, i) / temp_matrix(i, i);
210             // Colocamos cero en la columna bajo la diagonal
211             temp_matrix(j, i) = 0;
212             for(k = i + 1; k < temp_matrix.columns(); k++) {
213                 temp_matrix(j, k) = temp_matrix(j, k) - coef * temp_matrix(i, k);
214             }
215             temp_values(j) = temp_values(j) - coef * temp_values(i);
216         }
217     }
218 
219     Matrix<T> x_values = Matrix<T>(temp_values.rows());
220 
221     for(int i = temp_values.rows() - 1; i >= 0; i--) {
222         for (int j = temp_values.rows() - 1; j > i; j--) {
223             temp_values(i) -= x_values(j) * temp_matrix(i,j);
224         }
225         if(i == x_values.rows() - 1) {
226             x_values(x_values.rows() - 1) = temp_values(temp_values.rows() - 1) /
                    temp_matrix(temp_matrix.rows() - 1, temp_matrix.columns() - 1);
227         } else {
228             x_values(i) = temp_values(i) / temp_matrix(i,i);
229         }
230     }
231 
232     return x_values;
233 }
234 
235 #endif  /* EQSYS_H */
```

6.3. buildSystem.cpp

```
1  #include <iostream>
2  #include <math.h>
3  #include <stdio.h>
4  #include <fstream>
5  #include <sstream>
6  #include <stdio.h>
7  #include <string.h>
8  #include <time.h>
9  #include <new>
10 #include "eqsys.h"
11 
12 using namespace std;
13 
14 void load_a(Matrix<double>& A, double r_i, double r_e, int n, int m);
15 void load_b(Matrix<double>&b, double r_i, double r_e, int n, int m, double* t_i,
       double* t_e);
16 void insert_a(Matrix<double>& A, int j, int k, double r_i, double r_e, int n, int m);
17 void insert_b(Matrix<double>& b, int j, int k, double r_i, double r_e, int n, int m,
       double* t_i, double* t_e);
18 void load_temps(ifstream& inputFile, double* t_i, double* t_e, int n);
19 void save_result(Matrix<double>& m, FILE * pFile);
```

```
20 void generate_isotherm_lower(FILE * pFile, Matrix<double>& b, int m, int n, double
       r_i, double r_e, double iso);
21 void generate_isotherm_weighted(FILE * pFile, Matrix<double>& b, int m, int n, double
       r_i, double r_e, double iso);
22 int mod(int a, int b);
23
24 int main(int argc, char** argv) {
25
26     if (argc < 4) {
27         printf("Usage: %s inputFile outputFile method (0: EG, 1: LU) isoFile (
               optional)\n", argv[0]);
28         return 0;
29     }
30
31     ifstream inputFile(argv[1]);
32
33     if (!inputFile.good()) {
34         printf("Non-existant input file.\n");
35         return 0;
36     }
37
38     // granularity
39     int n; // 0 = O0 < 0_k < ... < 0_n = 2PI
40     int m; // ri = r0 < r_j < ... < r_m = re
41
42     double r_i, r_e;
43
44     double iso;
45     int ninst; // instances of the problem to solve
46
47     string line;
48     getline(inputFile, line);
49     sscanf(line.c_str(),"%lf %lf %d %d %lf %d",&r_i,&r_e,&m,&n,&iso,&ninst);
50
51     int solver = (int) (*argv[3] - '0');
52     if (solver != 0 && solver != 1) {
53         printf("Error: Invalid solver.\n");
54         return 0;
55     }
56
57     cout << "r_i: " << r_i << " r_e: " << r_e << " m+1: " << m << " n: " << n << "
           iso: " << iso << " ninst: " << ninst << endl;
58     cout << "inputFile: " << argv[1] << ", outputFile: " << argv[2] << ", method: "
           << argv[3] << endl;
59
60     double t_i[n];
61     double t_e[n];
62
63     FILE * pFile = fopen(argv[2],"w");
64
65     // build system: Ax = b
66     Matrix<double> A(n*m,n*m,0);
67     Matrix<double> b(n*m,1,0);
68     FILE * pIsoFile = NULL;
69     if (solver == 0) { // Gaussian Elimination
70         load_a(A,r_i,r_e,n,m);
71         clock_t before = clock();
72         EquationSystemLU<double> e(A); //temp
73         for (int j = 0; j < ninst; ++j) { // for every instance
```

```
74              load_temps(inputFile, t_i, t_e, n);
75              load_b(b,r_i,r_e,n,m,t_i,t_e);
76
77              Matrix<double> result(e.solve(b));
78              save_result(result, pFile);
79              if (argc == 5 && j == 0) {
80                  pIsoFile = fopen(argv[4],"w");
81              }
82              if (argc == 5) {
83                  // generate_isotherm_lower(pIsoFile, result, m, n, r_i, r_e, iso);
84                  generate_isotherm_weighted(pIsoFile, result, m, n, r_i, r_e, iso);
85              }
86          }
87          clock_t result = clock() - before;
88          cout << "method 0 takes: " << result/float(CLOCKS_PER_SEC) << " seconds" <<
                endl;
89      } else {
90          load_a(A,r_i,r_e,n,m);
91          clock_t before = clock();
92          EquationSystemLU<double> e(A); //temp
93          for (int j = 0; j < ninst; ++j) { // for every instance
94              load_temps(inputFile, t_i, t_e, n);
95              load_b(b,r_i,r_e,n,m,t_i,t_e);
96              Matrix<double> result(e.solve(b));
97              save_result(result, pFile);
98              if (argc == 5 && j == 0) {
99                  pIsoFile = fopen(argv[4],"w");
100             }
101             if (argc == 5) {
102                 generate_isotherm_lower(pIsoFile, result, m, n, r_i, r_e, iso);
103                 // generate_isotherm_weighted(pIsoFile, result, m, n, r_i, r_e, iso);
104             }
105         }
106         clock_t result = clock() - before;
107         cout << "method 1 takes: " << result/float(CLOCKS_PER_SEC) << " seconds" <<
                endl;
108     }
109
110     inputFile.close();
111     fclose(pFile);
112
113     if (pIsoFile != NULL) fclose(pIsoFile);
114
115     return 0;
116 }
117
118 void generate_isotherm_lower(FILE * pFile, Matrix<double>& b, int m, int n, double
        r_i, double r_e, double iso) {
119
120     double dR = (r_e - r_i) / (m - 1);
121     for (int k = 0; k < n; k++) {
122         for (int j = 0; j < m; j++) {
123             if ((b(j * n + k) <= iso || j == m-1) && j != 0) {
124                 fprintf(pFile, "%f\r\n", r_i + j*dR);
125                 break;
126             } else if (b(j * n + k) <= iso && j == 0) {
127                 fprintf(pFile, "%f\r\n", r_i);
128                 break;
129             }
```

```
130         }
131     }
132
133 }
134
135 void generate_isotherm_weighted(FILE * pFile, Matrix<double>& b, int m, int n, double
        r_i, double r_e, double iso) {
136
137     double dR = (r_e - r_i) / (m - 1);
138
139     for (int k = 0; k < n; k++) {
140         for (int j = 0; j < m; j++) {
141             if (b(j * n + k) < iso && j != m-1 && j != 0) {
142                 fprintf(pFile, "%f\r\n", r_i + (j-1)*dR + (b((j-1) * n + k) - b(j * n
                        + k)) / iso * dR);
143                 break;
144             } else if (b(j * n + k) < iso && j == 0) {
145                 fprintf(pFile, "%f\r\n", r_i);
146                 break;
147             } else if (j == m-1) {
148                 fprintf(pFile, "%f\r\n", r_i + j*dR);
149                 break;
150             }
151         }
152     }
153
154 }
155
156 void save_result(Matrix<double>& m, FILE * pFile) {
157
158     if (pFile != NULL) {
159         for (int i = 0; i < m.rows(); i++) {
160             fprintf(pFile, "%1.6f\r\n", m(i));
161         }
162         // fprintf(pFile, "\r");
163     } else {
164         cout << "Failed to open file." << endl;
165     }
166 }
167
168 void load_temps(ifstream& inputFile, double* t_i, double* t_e, int n) {
169     string line;
170     getline(inputFile, line);
171
172     char* buffer = strtok(strdup(line.c_str()), " ");
173
174     for (int i = 0; i < n; ++i) {
175         sscanf(buffer, "%lf", &t_i[i]);
176         buffer = strtok(NULL, " ");
177     }
178
179     for (int i = 0; i < n; ++i) {
180         sscanf(buffer, "%lf", &t_e[i]);
181         buffer = strtok(NULL, " ");
182     }
183 }
184
185 void load_a(Matrix<double>& A, double r_i, double r_e, int n, int m) {
186     for (int j = 0; j < m; j++) { // radius
```

```
187             for (int k = 0; k < n; k++) { // angle
188                 insert_a(A,j,k,r_i,r_e,n,m);
189             }
190         }
191     }
192
193     void load_b(Matrix<double>&b, double r_i, double r_e, int n, int m, double* t_i,
        double* t_e) {
194         for (int j = 0; j < m; j++) { // radius
195             for (int k = 0; k < n; k++) { // angle
196                 insert_b(b,j,k,r_i,r_e,n,m,t_i,t_e);
197             }
198         }
199     }
200
201     /* Laplacian matrix helper function
202      * r0 < r_j < ... < r_m
203      * O0 < 0_k < ... < 0_n
204      */
205     void insert_a(Matrix<double>& A, int j, int k, double r_i, double r_e, int n, int m)
        {
206
207         double dO = 2*M_PI / n;
208         double dR = (r_e - r_i) / (m - 1);
209
210         int r = j * n + k;
211         double r_j = r_i + j*dR;
212
213         if (j == 0) {
214             A(r,r) = 1;
215             return;
216         } else if (j == m - 1) {
217             A(r,r) = 1;
218             return;
219         }
220
221         // t_j,k
222         A(r,r) += (-2/pow(dR, 2)) + (1/(r_j*dR)) - 2/(pow(r_j, 2)*pow(dO, 2));
223
224         // t_j,k+1, border case! k > n, angle = 0
225         A(r, j * n + mod(k+1, n)) += 1/(pow(r_j, 2)*pow(dO, 2));
226
227         // t_j,k-1, border case! k < 0
228         A(r, j * n + mod(k-1, n)) += 1/(pow(r_j, 2)*pow(dO, 2));
229
230         // t_j-1,k
231         if (j != 1) { // inner circle
232             A(r,(j-1) * n + k) += 1/pow(dR, 2) - 1/(r_j * dR);
233         }
234
235         // t_j+1,k
236         if (j+1 != m-1) { // outer circle
237             A(r, (j+1) * n + k) += (1/pow(dR, 2));
238         }
239
240     }
241
242     void insert_b(Matrix<double>& b, int j, int k, double r_i, double r_e, int n, int m,
        double* t_i, double* t_e) {
```

```
243
244     double dR = (r_e - r_i) / (m - 1);
245
246     int r = j * n + k;
247     double r_j = r_i + j*dR;
248
249     b(r) = 0;
250
251     if (j == 0) {
252         b(r) = t_i[k];
253         return;
254     } else if (j == m - 1) {
255         b(r) = t_e[k];
256         return;
257     }
258
259     if (j == 1) { // inner circle
260         b(r) -= t_i[k] * (1/pow(dR, 2) - 1/(r_j * dR));
261     }
262
263     // t_j+1,k
264     if (j+1 == m-1) { // outer circle
265         b(r) -= t_e[k] * (1/pow(dR, 2));
266     }
267
268 }
269
270 void insertValue(Matrix<double>& A, Matrix<double>& b, int j, int k, double r_i,
    double r_e, int n, int m, double* t_i, double* t_e) {
271
272     double dO = 2*M_PI / n;
273     double dR = (r_e - r_i) / (m - 1);
274
275     int r = j * n + k;
276     double r_j = r_i + j*dR;
277
278     if (j == 0) {
279         A(r,r) = 1;
280         b(r) = t_i[k];
281         return;
282     } else if (j == m - 1) {
283         A(r,r) = 1;
284         b(r) = t_e[k];
285         return;
286     }
287
288     // t_j,k
289     A(r,r) += (-2/pow(dR, 2)) + (1/(r_j*dR)) - 2/(pow(r_j, 2)*pow(dO, 2));
290
291     // t_j,k+1, border case! k > n, angle = 0
292     A(r, j * n + mod(k+1, n)) += 1/(pow(r_j, 2)*pow(dO, 2));
293
294     // t_j,k-1, border case! k < 0
295     A(r, j * n + mod(k-1, n)) += 1/(pow(r_j, 2)*pow(dO, 2));
296
297     // t_j-1,k
298     if (j == 1) { // inner circle
299         b(r) -= t_i[k] * (1/pow(dR, 2) - 1/(r_j * dR));
300     } else {
```

```
301             A(r,(j-1) * n + k) += 1/pow(dR, 2) - 1/(r_j * dR);
302         }
303
304         // t_j+1,k
305         if (j+1 == m-1) { // outer circle
306             b(r) -= t_e[k] * (1/pow(dR, 2));
307         } else {
308             A(r, (j+1) * n + k) += (1/pow(dR, 2));
309         }
310
311     }
312
313     int mod(int a, int b) {
314         int r = a %b;
315         return r < 0 ? r + b : r;
316     }
```