Métodos Numéricos
TP1

31 de agosto de 2015

Integrante	LU	Correo electrónico
Martin Baigorria	575/14	martinbaigorria@gmail.com
Federico Beuter	827/13	federicobeuter@gmail.com
Rodrigo Kapobel	864/13	jangamesdev@gmail.com
Mauro Cherubini	835/13	cheru.mf@gmail.com

Reservado para la cátedra

Instancia	Docente	Nota
Primera entrega		
Segunda entrega		

Índice

1. Introducción **3**
1.1. Discretizacion 4
1.2. Sistema Lineal 4

2. Desarrollo **6**
2.1. Discretizacion 7
2.2. Sistema Lineal 7
2.3. Isoterma 8
2.4. Propiedades del sistema. 8

3. Codigo **11**
3.1. matrix.h 11
3.2. eqsys.h 15
3.3. buildSystem.cpp 19

1. Introducción

2. Desarrollo

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.

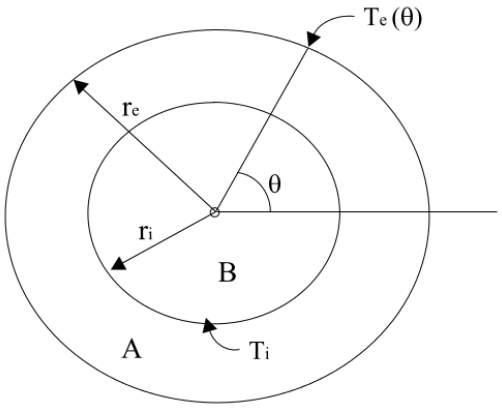

Figura 1: Sección circular del horno

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 y luego aproximamos las derivadas parciales:

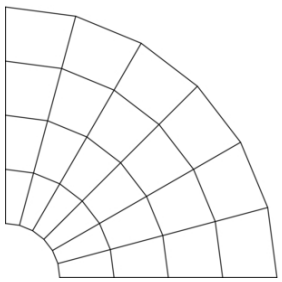

Figura 2: Discretizacion de la pared del horno.

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

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

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

Reemplazando la aproximación numérica en el laplaciano y el radio por su respectiva discretizacion 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 $r_j = r_i + j \times \Delta r$.

Por lo tanto, 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. Discretizacion

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. La discretizacion 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:

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

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

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

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

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

Luego armamos la matriz del sistema, en donde los coeficientes serán los α anteriores, expresando en cada fila su valor para cada temperatura. Tengase 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 incognita, 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. De esta forma, la matriz del sistema será una matriz *banda*.

Graficamente dicha matriz queda definida de la siguiente forma:

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

Y 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.

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

2.3. Isoterma

Una vez que resolvamos el sistema lineal, debemos buscar la isoterma. Dada que es una aproximación discreta, no es muy probable que encontremos valores de temperatura justo iguales a la curva de nivel. Por lo tanto debemos tener cierta tolerancia de error, o hasta interpolar de alguna manera curvas de nivel adyacentes. Es decir, debemos hacer una búsqueda inteligente sobre el vector b. De hecho, dado que el calor se propaga de forma uniforme, podríamos hasta interpolar un circulo con las temperaturas adyacentes. La única forma de que la isoterma tenga una forma elíptica es que la temperatura exterior no sea uniforme. En este caso, simplemente hay que interpolar una elipse.

2.4. Propiedades del sistema.

Como ya se ha visto en la materia, no es posible aplicar los metodos 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 matienen 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 posicion 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 $\left|\alpha_{jj}^{(2)}\right| \geq \sum_{k=0,k\neq j}^{(m+1)(n-1)} \left|\alpha_{jk}^{(2)}\right|, \forall j \in \{1, ..., (m+1)(n-1)\}$

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

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

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

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

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

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

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

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

Finalizada asi esta demostración, sabemos que podemos contar con que tanto la matriz incial 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 asi 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 induciria 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 seccion conflictiva, en particular sobre la submatriz central que resulta de *recortar* las primeras y últimas n filas y columnas. La razon 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 seccion de n columnas, sin alterar a la seccion 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 seccion, ya no es necesario seguir transformando la matriz, dado que la obtenida hasta el momento, ya es triangular. Dicho esto, llamemos $A^{(*)}$ a la seccion de A delimitara por aquel *recorte*.

Sobre $A^{(*)}$, veremos que se cumple una segunda hipotesis bastante util, 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 algun 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 detras (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 detras 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 atras (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. Codigo

3.1. 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 */
```

3.2. 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 */
```

3.3. 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
122     for (int k = 0; k < n; k++) {
123         for (int j = 0; j < m; j++) {
124             if (b(j * n + k) < iso || j == m-1) {
125                 fprintf(pFile, "%f\r\n", r_i + j*dR);
126                 break;
127             }
128         }
129     }
```

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

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

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

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