Métodos Numéricos
TP1

19 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

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.

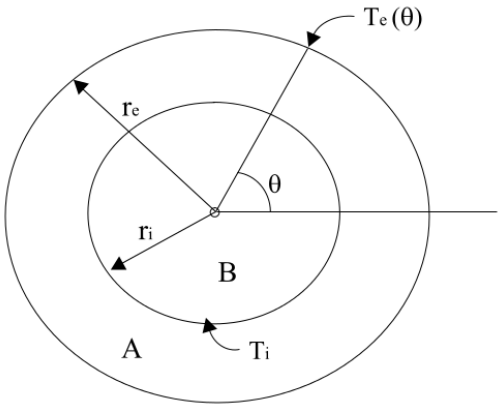

Figura 1: Sección circular del horno

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:

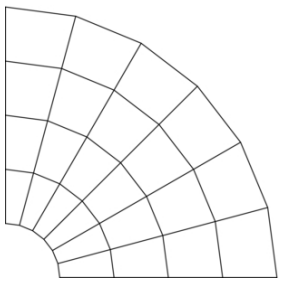

Figura 2: Discretizacion de la pared del horno.

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

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

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

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

$$\frac{t_{j-1,k} - 2t_{jk} + t_{j+1,k}}{(\Delta r)^2} + \frac{1}{r_j}\frac{t_{j,k} - t_{j-1,k}}{\Delta r} + \frac{1}{r_j^2}\frac{t_{j,k-1} - 2t_{jk} + t_{j,k+1}}{(\Delta\theta)^2} = 0 \tag{5}$$

Donde $r_j = r_i + j \times \Delta\theta$.

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

$$T(r, \theta) = T_i \quad \textit{para todo punto } (r, \theta) \textit{ con } r \leq r_i \tag{6}$$

$$T(r_e, \theta) = T_e(\theta) \quad \textit{para todo punto } (r_e, \theta) \tag{7}$$

1.1. 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 $(m-1)*(n+1)$ 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 $t_s(\theta)$. 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.

1.2. 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:

$$\frac{t_{j-1,k} - 2t_{jk} + t_{j+1,k}}{(\Delta r)^2} + \frac{1}{r_j}\frac{t_{j,k} - t_{j-1,k}}{\Delta r} + \frac{1}{r_j^2}\frac{t_{j,k-1} - 2t_{jk} + t_{j,k+1}}{(\Delta\theta)^2} = 0 \tag{8}$$

Reescribiendo:

$$\alpha_{j,k} \times t_{j,k} + \alpha_{j-1,k} \times t_{j-1,k} + \alpha_{j+1,k} \times t_{j+1,k} + \alpha_{j,k+1} \times t_{j,k+1} + \alpha_{j,k-1} \times t_{j,k-1} = 0 \tag{9}$$

Donde:

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

$$\alpha_{j,k+1} = \frac{1}{r_j^2 \times \Delta\theta^2} \tag{11}$$

$$\alpha_{j,k-1} = \frac{1}{r_j^2 \times \Delta\theta^2} \tag{12}$$

$$\alpha_{j+1,k} = \frac{1}{(\Delta r)^2} \tag{13}$$

$$\alpha_{j-1,k} = \frac{1}{(\Delta r)^2} - \frac{1}{r_j \times \Delta r} \tag{14}$$

1.3. 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. Codigo

2.1. matrix.h

```
/*
 * File:   matrix.h
 * Author: Federico
 *
 * Created on August 16, 2015, 9:54 PM
 */

#ifndef MATRIX_H
#define MATRIX_H

#include <algorithm>
#include <math.h>
#include <vector>

using namespace std;

// La matriz respeta la notacion de la catedra, es decir, el primer subindice
// es la fila y el segundo es la columna

template<class T>
class Matrix {
    public:
        Matrix();
        Matrix(int rows); // Columnas impllicitas (col = 1)
        Matrix(int rows, int col);
        Matrix(const Matrix<T>& other);
        ~Matrix();

        Matrix<T>& operator=(const Matrix<T>& other);
        Matrix<T> operator*(const Matrix<T>& other);
        Matrix<T>& operator*=(const Matrix<T>& other);
        Matrix<T> operator+(const Matrix<T>& other);
        Matrix<T>& operator+=(const Matrix<T>& other);
        Matrix<T> operator-(const Matrix<T>& other);
        Matrix<T>& operator-=(const Matrix<T>& other);

        Matrix<T> operator*(const T& scalar);
        Matrix<T> operator/(const T& scalar);

        T& operator()(int a, int b);
        const T& operator()(const int a, const int b) const;
        T& operator()(int a);
        const T& operator()(const int a) const;

        int rows();
        int columns();
        void printMatrix();

    private:
        vector<vector<T> > _values;
        int _rows;
        int _columns;

};

```

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

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

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

```
233 template<class T>
234 int Matrix<T>::columns() {
235     return _columns;
236 }
237
238 template<class T>
239 void Matrix<T>::printMatrix() {
240     for(int i = 0; i < _rows; i++) {;
241         for(int j = 0; j < _columns; j++) {
242             cout << _values[i][j] << " ";
243         }
244         cout << endl;
245     }
246     cout << endl;
247 }
248
249 #endif  /* MATRIX_H */
```

2.2. 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         //vector<T> solve(const vector<T>& 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     int coef;
35     int i, j, k, l, m;
36
37     // Armar la matriz lower con la diagonal en uno
38     lower = Matrix<T>(upper.rows(), upper.columns());
39     for(i = 0; i < lower.rows(); i++){
```

```
40          for(j = 0; j < lower.columns(); j++) {
41              if(i == j) {
42                  lower(i,j) = 1;
43              } else {
44                  lower(i,j) = 0;
45              }
46          }
47      }
48
49      for(i = 0; i < upper.columns(); i++) {
50          for(j = i + 1; j < upper.rows(); j++) {
51              if(upper(i, i) == 0) {
52                  // Hay que buscar la proxima fila sin cero
53                  for(k = i + 1; k < upper.rows(); k++) {
54                      if(upper(k, i) != 0) {
55                          break;
56                      }
57                  }
58
59                  if(k == upper.rows()) { // No hay filas para permutar
60                      abort();
61                  } else {
62                      if(!isPermutated){
63                          // Generamos la matriz de permutacion con uno en la diagonal
64                          isPermutated = true;
65                          permutation = Matrix<T>(upper.rows(), upper.columns());
66
67                          for(l = 0; l < permutation.rows(); l++) {
68                              for(m = 0; m < permutation.columns(); m++) {
69                                  if(l == m) {
70                                      permutation(l,m) = 1;
71                                  } else {
72                                      permutation(l,m) = 0;
73                                  }
74                              }
75                          }
76                      }
77                      // Permutamos las filas
78                      for(l = 0; l < permutation.columns(); l++) {
79                          if(l == k) {
80                              permutation(i, l) = 1;
81                          } else {
82                              permutation(i, l) = 0;
83                          }
84                          if(l == i) {
85                              permutation(k, l) = 1;
86                          } else {
87                              permutation(k, l) = 0;
88                          }
89                      }
90                      // Hacemos el producto para efectivamente permutar upper
91                      upper = permutation * upper;
92                  }
93              }
94
95              // Calculamos y guardamos el coeficiente
96              coef = upper(j, i)/upper(i, i);
97              lower(j, i) = coef;
98
```

```
99              // Colocamos cero en la columna bajo la diagonal
100             for(k = i; k < upper.columns(); k++) {
101                 upper(j, k) = upper(j, k) - coef * upper(i, k);
102             }
103         }
104     }
105     // COSAS PARA TESTEAR NOMAS, IGNORAR
106
107     upper.printMatrix();
108     lower.printMatrix();
109
110     lower *= upper;
111     lower.printMatrix();
112 }
113
114 template<class T>
115 class EquationSystem{
116     public:
117         EquationSystem(const Matrix<T>& inicial);
118
119         vector<T> solve(const vector<T>& values);
120
121     private:
122         Matrix<T> matriz;
123 };
124
125 #endif  /* EQSYS_H */
```

2.3. buildSystem.cpp

```
1  #include <iostream>
2  #include <math.h>
3  #include "eqsys.h"
4
5  #define INNER_TEMP 20
6
7  using namespace std;
8
9  void insertValue(Matrix<double>& A, Matrix<double>& b, int j, int k, double r_i,
       double r_e, int n, int m, double t_e);
10
11 int main() {
12
13     // granularity
14     int n = 2; // O0 < 0_k < ... < 0_n
15     int m = 3; // r0 < r_j < ... < r_m
16
17     // system parameters
18     double r_i = 1;
19
20     double r_e = 3;
21     double t_e = 10;
22
23     // build system: Ax = b
24     Matrix<double> A((m-1)*(n+1),(m-1)*(n+1));
25     Matrix<double> b((m-1)*(n+1), 1);
26
27     /* each temperature has 1 laplacian, and depends on 4 temperatures.
28      * i'm looking for t_j,k in the valid range.
```

```
29       */
30      for (int k = 0; k <= n; k++) {
31          for (int j = 1; j < m; j++) { // avoid borders
32              insertValue(A,b,j,k,r_i,r_e,n,m,t_e);
33          }
34      }
35
36      cout << endl;
37      cout << "Matrix A" << endl;
38      A.printMatrix();
39
40      cout << "Matrix b" << endl;
41      b.printMatrix();
42
43      return 0;
44  }
45
46  /* t_j,k
47   * r0 < r_j < ... < r_m
48   * O0 < 0_k < ... < 0_n
49   * b = | t1,0    | rows with fixed angle first.
50   *     | ..... |
51   *     | tm-1,0 |
52   *     | ..... |
53   *     | t1,n    |
54   *     | ..... |
55   *     | tm-1,n |
56   */
57  void insertValue(Matrix<double>& A, Matrix<double>& b, int j, int k, double r_i,
        double r_e, int n, int m, double t_e) {
58
59      cout << "j: " << j << " k: " << k << " m: " << m << " n: " << n << endl;
60
61      double dO = 2*M_PI / (n+1);
62      double dR = (r_e - r_i) / m;
63
64      int r = k * (m - 1) + (j - 1);
65      double r_j = r_i + j*dR;
66
67      // t_j,k
68      A(r,r) += - (2/pow(dR, 2)) + (1/(r_j*dR)) - (2/pow(r_j, 2)*pow(dO, 2));
69
70      // t_j,k+1, border case! k > n, angle = 0
71      A(r, (r + (m-1)) %(m-1)*(n+1)) += 1/(pow(r_j, 2)*pow(dO, 2));
72
73      // t_j,k-1, border case! k < 0
74      A(r,(r - (m-1)) %(m-1)*(n+1))  += 1/(pow(r_j, 2)*pow(dO, 2));
75
76      // t_j-1,k
77      if (j == 1) { // inner circle
78          b(r) += INNER_TEMP * (2/pow(dR, 2) + 1/(r_e * dR));
79      } else {
80          A(r,r - 1) += - 2/pow(dR, 2) - 1/(r_j * dR);
81      }
82
83      // t_j+1,k
84      if (j+1 == m) { // outer circle
85          b(r) += t_e * (1/pow(dR, 2));
86      } else {
```

```
87              A(r,r + 1) += (1/pow(dR, 2));
88          }
89
90  }
```