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.

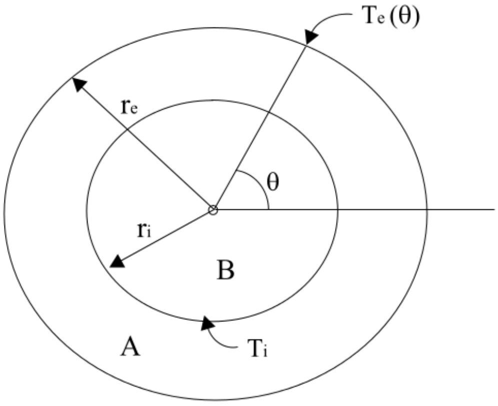

Figura 1: Sección circular del horno

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:

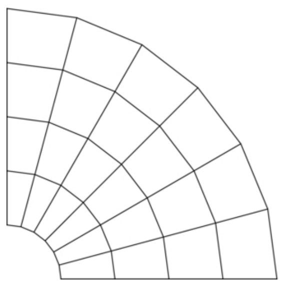

Figura 2: Discretizacion de la pared del horno.

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

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

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

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:

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

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

```cpp
/*
 * 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;

};

```

```
template<class T>
Matrix<T>::Matrix()
    : _values(1), _rows(1), _columns(1)
{
    _values[0].resize(1);
}

template<class T>
Matrix<T>::Matrix(int rows)
    : _values(rows), _rows(rows), _columns(1)
{
    for(int i = 0; i < rows; i++) {
        _values[i].resize(1);
    }
}

template<class T>
Matrix<T>::Matrix(int rows, int col)
    : _values(rows), _rows(rows), _columns(col)
{
    for(int i = 0; i < rows; i++) {
        _values[i].resize(col);
    }
}

template<class T>
Matrix<T>::Matrix(const Matrix<T>& other)
    : _values(other._values), _rows(other._rows), _columns(other._columns)
{}

template<class T>
Matrix<T>::~Matrix() {}

template<class T>
Matrix<T>& Matrix<T>::operator=(const Matrix<T>& other) {
  if (&other == this)
    return *this;

  int new_rows = other._rows;
  int new_columns = other._columns;

  _rows = new_rows;
  _columns = new_columns;

  _values.resize(new_rows);
  for (int i = 0; i < new_columns; i++) {
      _values[i].resize(new_columns);
  }

  for(int i = 0; i < new_rows; i++) {
    for(int j = 0; j < new_columns; j++) {
      _values[i][j] = other(i, j);
    }
  }

  return *this;
}

template<class T>
```

```
Matrix<T> Matrix<T>::operator*(const Matrix<T>& other) {
    // ASUME QUE LAS DIMENSIONES DAN
    Matrix<T> result(_rows, other._columns);

    int innerDim = _columns; // Tambien podria ser other._rows

    for(int i = 0; i < result._rows; i++) {
        for(int j = 0; j < result._columns; j++) {
            result(i,j) = 0;
            for(int k = 0; k < innerDim; k++) {
                result(i,j) += _values[i][k] * other(k,j);
            }
        }
    }

    return result;
}

template<class T>
Matrix<T>& Matrix<T>::operator*=(const Matrix<T>& other) {
    Matrix<T> result = (*this) * other;
    (*this) = result;
    return (*this);
}

template<class T>
Matrix<T> Matrix<T>::operator+(const Matrix<T>& other) {
    // ASUME QUE LAS DIMENSIONES DAN
    Matrix<T> result(_rows, other._columns);

    for(int i = 0; i < result._rows; i++) {
        for(int j = 0; j < result._columns; j++) {
            result(i,j) = _values[i][j] + other(i,j);
        }
    }

    return result;
}

template<class T>
Matrix<T>& Matrix<T>::operator+=(const Matrix<T>& other) {
    Matrix<T> result = (*this) + other;
    (*this) = result;
    return (*this);
}

template<class T>
Matrix<T> Matrix<T>::operator-(const Matrix<T>& other) {
    // ASUME QUE LAS DIMENSIONES DAN
    Matrix<T> result(_rows, other._columns);

    for(int i = 0; i < result._rows; i++) {
        for(int j = 0; j < result._columns; j++) {
            result(i,j) = _values[i][j] - other(i,j);
        }
    }

    return result;
}
```

```
template<class T>
Matrix<T>& Matrix<T>::operator-=(const Matrix<T>& other) {
    Matrix<T> result = (*this) - other;
    (*this) = result;
    return (*this);
}

template<class T>
Matrix<T> Matrix<T>::operator*(const T& scalar) {
    Matrix<T> result(_rows, _columns);

    for(int i = 0; i < result._rows; i++) {
        for(int j = 0; j < result._columns; j++) {
            result(i,j) = _values[i][j] * scalar;
        }
    }

    return result;
}

template<class T>
Matrix<T> Matrix<T>::operator/(const T& scalar) {
    Matrix<T> result(_rows, _columns);

    for(int i = 0; i < result._rows; i++) {
        for(int j = 0; j < result._columns; j++) {
            result(i,j) = _values[i][j] / scalar;
        }
    }

    return result;
}

template<class T>
T& Matrix<T>::operator ()(int a, int b) {
    return _values[a][b];
}

template<class T>
const T& Matrix<T>::operator ()(const int a, const int b) const {
    return _values[a][b];
}

template<class T>
T& Matrix<T>::operator ()(int a) {
    return _values[a][0];
}

template<class T>
const T& Matrix<T>::operator ()(const int a) const {
    return _values[a][0];
}

template<class T>
int Matrix<T>::rows() {
    return _rows;
}
```

```cpp
template<class T>
int Matrix<T>::columns() {
    return _columns;
}

template<class T>
void Matrix<T>::printMatrix() {
    for(int i = 0; i < _rows; i++) {;
        for(int j = 0; j < _columns; j++) {
            cout << _values[i][j] << " ";
        }
        cout << endl;
    }
    cout << endl;
}

#endif  /* MATRIX_H */
```

2.2. eqsys.h

```cpp
/*
 * File:   eqsys.h
 * Author: Federico
 *
 * Created on August 17, 2015, 5:57 PM
 */

#ifndef EQSYS_H
#define EQSYS_H

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

template<class T>
class EquationSystemLU {
    public:
        EquationSystemLU(const Matrix<T>& inicial);

        //vector<T> solve(const vector<T>& values);

    private:
        Matrix<T> lower;
        Matrix<T> upper;
        bool isPermutated;
        Matrix<T> permutation;
};

template<class T>
EquationSystemLU<T>::EquationSystemLU(const Matrix<T>& inicial)
    : upper(inicial), isPermutated(false)
{
    int coef;
    int i, j, k, l, m;

    // Armar la matriz lower con la diagonal en uno
    lower = Matrix<T>(upper.rows(), upper.columns());
    for(i = 0; i < lower.rows(); i++){
```

```
        for(j = 0; j < lower.columns(); j++) {
            if(i == j) {
                lower(i,j) = 1;
            } else {
                lower(i,j) = 0;
            }
        }
    }

    for(i = 0; i < upper.columns(); i++) {
        for(j = i + 1; j < upper.rows(); j++) {
            if(upper(i, i) == 0) {
                // Hay que buscar la proxima fila sin cero
                for(k = i + 1; k < upper.rows(); k++) {
                    if(upper(k, i) != 0) {
                        break;
                    }
                }

                if(k == upper.rows()) { // No hay filas para permutar
                    abort();
                } else {
                    if(!isPermutated){
                        // Generamos la matriz de permutacion con uno en la diagonal
                        isPermutated = true;
                        permutation = Matrix<T>(upper.rows(), upper.columns());

                        for(l = 0; l < permutation.rows(); l++) {
                            for(m = 0; m < permutation.columns(); m++) {
                                if(l == m) {
                                    permutation(l,m) = 1;
                                } else {
                                    permutation(l,m) = 0;
                                }
                            }
                        }
                    }
                    // Permutamos las filas
                    for(l = 0; l < permutation.columns(); l++) {
                        if(l == k) {
                            permutation(i, l) = 1;
                        } else {
                            permutation(i, l) = 0;
                        }
                        if(l == i) {
                            permutation(k, l) = 1;
                        } else {
                            permutation(k, l) = 0;
                        }
                    }
                    // Hacemos el producto para efectivamente permutar upper
                    upper = permutation * upper;
                }
            }

            // Calculamos y guardamos el coeficiente
            coef = upper(j, i)/upper(i, i);
            lower(j, i) = coef;

```

```
            // Colocamos cero en la columna bajo la diagonal
            for(k = i; k < upper.columns(); k++) {
                upper(j, k) = upper(j, k) - coef * upper(i, k);
            }
        }
    }
    // COSAS PARA TESTEAR NOMAS, IGNORAR

    upper.printMatrix();
    lower.printMatrix();

    lower *= upper;
    lower.printMatrix();
}

template<class T>
class EquationSystem{
    public:
        EquationSystem(const Matrix<T>& inicial);

        vector<T> solve(const vector<T>& values);

    private:
        Matrix<T> matriz;
};

#endif /* EQSYS_H */
```

2.3. buildSystem.cpp

```
#include <iostream>
#include <math.h>
#include "eqsys.h"

#define INNER_TEMP 20

using namespace std;

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);

int main() {

    // granularity
    int n = 3; // O0 < 0_k < ... < 0_n
    int m = 3; // r0 < r_j < ... < r_m

    // system parameters
    double r_i = 1;

    double r_e = 5;
    double t_e = 10;

    // build system: Ax = b
    Matrix<double> A((m-1)*(n+1),(m-1)*(n+1));
    Matrix<double> b((m-1)*(n+1));

    /* each temperature has 1 laplacian, and depends on 4 temperatures.
     * i'm looking for t_j,k in the valid range.
```

```
     */
    for (int k = 0; k <= n; k++) {
        for (int j = 1; j < m; j++) { // avoid borders
            insertValue(A,b,j,k,r_i,r_e,n,m,t_e);
        }
    }

    cout << endl;
    cout << "Matrix A" << endl;
    A.printMatrix();

    cout << "Matrix b" << endl;
    b.printMatrix();

    return 0;
}

/* t_j,k
 * r0 < r_j < ... < r_m
 * O0 < 0_k < ... < 0_n
 * b = | t1,0    | rows with fixed angle first.
 *     | ......  |
 *     | tm-1,0  |
 *     | ......  |
 *     | t1,n    |
 *     | ......  |
 *     | tm-1,n  |
 */
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) {

    cout << "j: " << j << " k: " << k << " m: " << m << " n: " << n << endl;

    double dO = 2*M_PI / (n+1);
    double dR = (r_e - r_i) / m;

    int r = k * (m - 1) + (j - 1);
    double r_j = r_i + j*dR;

    // t_j,k
    A(r,r) += - (2/pow(dR, 2)) + (1/(r_j*dR)) - (2/pow(r_j, 2)*pow(dO, 2));

    // t_j,k+1, border case! k > n, angle = 0
    A(r, (r + (m-1)) % (m-1)*(n+1)) += 1/(pow(r_j, 2)*pow(dO, 2));

    // t_j,k-1, border case! k < 0
    A(r,(r - (m-1)) % (m-1)*(n+1))  += 1/(pow(r_j, 2)*pow(dO, 2));

    // t_j-1,k
    if (j == 1) { // inner circle
        b(r) -= INNER_TEMP * (1/pow(dR, 2) - 1/(r_j * dR));
    } else {
        A(r,r - 1) += 1/pow(dR, 2) - 1/(r_j * dR);
    }

    // t_j+1,k
    if (j+1 == m) { // outer circle
        b(r) -= t_e * (1/pow(dR, 2));
    } else {
```

```
            A(r,r + 1) += (1/pow(dR, 2));
    }

}
```