Métodos Numéricos

Primer cuatrimestre 2018

Departamento de Computación
Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

Trabajo Práctico 3

Tomografía computada

Integrante	LU	Correo electrónico
Ingani, Bruno	50/13	chino117@hotmail.com
Hernández, Nicolás Carlos	122/13	nicoh22@hotmai.com
Beccar García, Augusto	267/13	abg101@gmail.com

En el presente trabajo utilizarmos el método de Page Rank para calcular la importancia de un conjunto de páginas web enlazadas entre sí. Para lograrlo implementamos Eliminación Gaussians sobre Matriz rala, aprovechando la estructura de la web. Medimos su eficiencia y realizamos pruebas sobre grafos ilustrativos de los rasgos de PageRank. Conlcuimos que la calidad del ranking calculado depende del contexto de aplicación y de si se hace uso de sus características más relevantes como ser la ponderación de links según el puntaje de la página enlazadora.

Palabras claves: Page Rank, Eliminación Gaussiana, Matriz rala, Páginas web

Reservado para la cátedra

Instancia	Docente	Nota
Primera entrega		
Segunda entrega		

Índice

1. Introducción

2. Desarrollo

3. Resultados y discusión

4. Conclusión

5. Referencias

6. Apéndice

6.1. A: Enunciado

Métodos Numéricos
1^o Cuatrimestre 2018
Trabajo Práctico 3

Departamento de Computación
Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

Tomografía computada

Introducción

El objetivo del trabajo práctico es evaluar un método para reconstruir imágenes tomográficas sujetas a ruido, utilizando el método de aproximación por cuadrados mínimos.

En muchos ámbitos de la medicina se trabaja con estudios que buscan obtener los mejores resultados posibles. Sin embargo, con el fin de evitar procesos invasivos en los pacientes o disminuir los efectos colaterales se hace uso de la tecnología. Ejemplo de esto son las tomografías computadas, donde se atraviesa el objeto de estudio con rayos X. Luego, según la proporción de la radiación inicial que llega al otro lado, se consigue estimar la densidad del objeto atravesado.

Dado que cada envío de un rayo atraviesa una sección se realizan diversos disparos con diferentes ángulos para tratar de captar diferentes combinaciones con las diferentes partes del objeto. Como además los instrumentos tienen errores en la medición es recomendable realizar repetidas veces el mismo envío para tratar de disminuir los efectos negativos asociados. Dada una cierta granularidad elegida para diagramar la imagen se tomará luego una serie de mediciones con los rayos y se determinará un sistema de ecuaciones. La particularidad de dicho sistema es que no será determinado sino sobredeterminado, debido a que se toman múltiples medidas y a los errores numéricos en ellas. Para conocer los niveles de densidad en cada parte del objeto en la discretización basta entonces resolver este sistema, para lo que se usarán métodos númericos vistos en la materia.

Así como en muchas aplicaciones, no siempre es posible contar con datos reales ya sea porque son difíciles o caros de conseguir o, como en este caso, porque no es razonable realizar sucesivas tomografías sobre un individuo por la cantidad de radiación que podría recibir. En este trabajo realizaremos la *simulación* sobre una imagen obtenida en una tomografía lo que nos permitirá además juzgar que tan buena es la aproximación obtenida.

El método de reconstrucción

El análisis tomográfico de una sección (que suponemos bidimensional y cuadrada) de un cuerpo consiste en emitir señales de rayos X que lo atraviesan en diferentes direcciones, midiendo el tiempo[1] que tarda cada una en atravesarlo. Por ejemplo, la Figura 1 muestra una posible distribución de estas señales.

Suponemos que el cuerpo se discretiza en $n \times n$ celdas cuadradas, de modo tal que en cada celda las señales de rayos X tienen velocidad constante. Si d_{ij}^k es la distancia que recorre el k-ésimo rayo X en la celda ij (notar que $d_{ij}^k = 0$ si el rayo no pasa por esta celda) y v_{ij} es la velocidad de la señal de rayo X en esa celda, entonces el tiempo de recorrido de la señal completa es de

$$t_k = \sum_{i=1}^{n} \sum_{j=1}^{n} d_{ij}^k v_{ij}^{-1}. \tag{1}$$

[1]Esto es una simplificación, en realidad se mide la intensidad de los rayos.

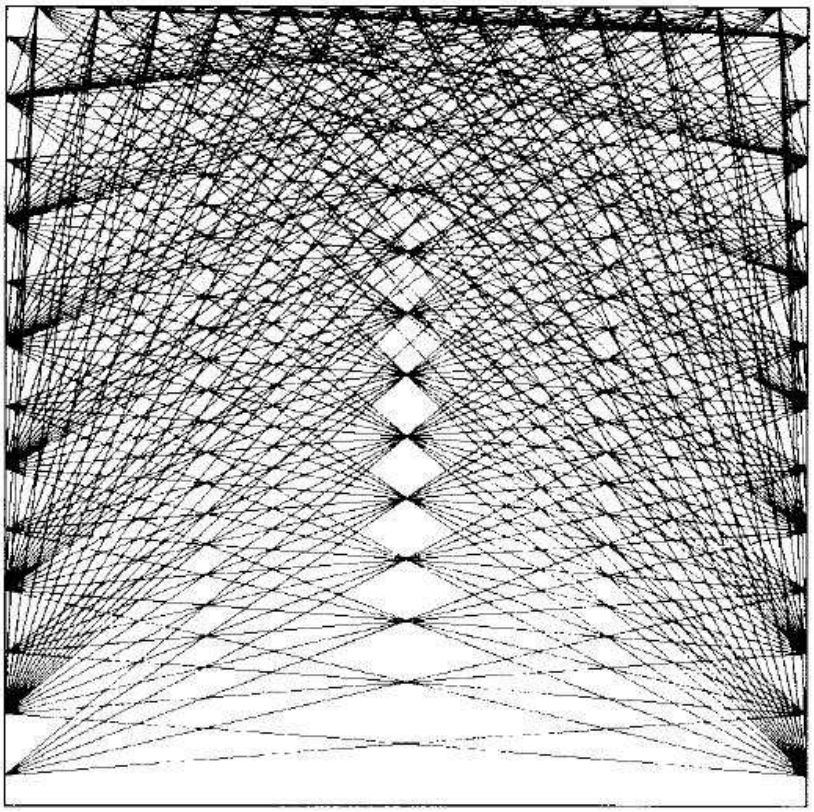

Figura 1: Ejemplo de configuración de las señales.

Como resultado del análisis tomográfico, se tienen mediciones del tiempo que tardó cada señal en recorrer el cuerpo. Si se emitieron m señales, entonces el resultado es un vector $t \in \mathbb{R}^m$, tal que t_k indica el tiempo de recorrida de la k-ésima señal. Sea $D \in \mathbb{R}^{m \times n^2}$ una matriz cuyas filas se corresponden con las m señales y cuyas columnas se corresponden con las n^2 celdas de la discretización del cuerpo de forma vectorizada, y tal que la fila k contiene los valores d_{ij}^k en las columnas correspondientes a cada celda. Entonces, las velocidades de recorrida originales v_{ij} se pueden reconstruir resolviendo el sistema de ecuaciones $Ds = t$. El vector solución $s \in \mathbb{R}^{n^2}$ contiene los valores inversos de las velocidades originales en cada celda.

Un problema fundamental que debe considerarse es la presencia de errores de medición en el vector t de tiempos de recorrido. Dado que el sistema estará en general sobredeterminado, la existencia de estos errores hará que no sea posible encontrar una solución que satisfaga al mismo tiempo todas las ecuaciones del sistema. Para manejar este problema, el sistema $Ds = t$ se resuelve utilizando el método de aproximación por *cuadrados mínimos*, obteniendo así una solución que resuelve de forma aproximada el sistema original.

Enunciado

El trabajo práctico consiste en implementar un programa en `C++` que simule el proceso de tomografía y reconstrucción, realizando los siguientes pasos:

1. Leer desde un archivo una imagen con los datos reales (discretizados) de un cuerpo. Para los fines de este procedimiento, se considerará que el valor de cada pixel de la imagen corresponde a la velocidad de recorrida de las señales de rayos X al atravesar ese pixel.
2. Ejecutar el proceso tomográfico, generando un conjunto relevante de señales y sus tiempos de recorrida exactos.
3. Perturbar los tiempos de recorrida con ruido aleatorio.
4. Ejecutar el método propuesto en la sección anterior sobre los datos perturbados, con el objetivo de reconstruir el cuerpo original. La imagen resultante se debe guardar en un archivo de salida.

El programa deberá tomar como parámetros los nombres de los archivos de entrada y de salida, junto con un parámetro que permita especificar el nivel de ruido a introducir en la imagen. El formato de los archivos de entrada y salida queda a elección del grupo. Para simplificar la implementación, es posible utilizar un formato propio que no sea ninguno de los formatos estándar para guardar imágenes[2].

Experimentación

Sobre la base de la implementación, se **pide** medir la calidad de la imagen reconstruida en función del nivel de ruido. Para medir el error de la imagen resultante se deberá utilizar el error cuadrático medio, definido como

$$ECM(v, \tilde{v}) = \frac{\sum_{i=1}^{n} \sum_{j=1}^{n} (v_{ij} - \tilde{v}_{ij})^2}{n^2}, \tag{2}$$

siendo v_{ij} la velocidad de recorrido de la celda ij del cuerpo original, y $\tilde{v}_{ij}$ la velocidad en la misma celda del cuerpo reconstruido.

Además, se **deben** reportar los tiempos de procesamiento.

Se **deben** probar con distintas estrategias geométricas para generar las señales de rayos X (paquetes de señales radiando de puntos fijos o puntos de inicio móviles, señales partiendo de los cuatro lados de la imagen o sólo de algunos lados, rango de ángulos de salida de las señales, etc.) para buscar la estrategia que minimice el error de la reconstrucción.

Por otra parte, puede ser interesante medir el número de condición de la matriz asociada al sistema de ecuaciones normales que resuelve el problema de cuadrados mínimos, para analizar la estabilidad de la resolución. En caso de que el sistema tenga un número de condición alto, se pueden intentar esquemas de regularización para mejorar la estabilidad de la solución obtenida.

[2]Se sugiere utilizar el formato pgm ya visto en el TP2.

Dataset

Para la experimentación se deberá utilizar, además de las imágenes provistas por la cátedra, alguno de los conjunto de datos provisto en el siguiente link: `http://www.via.cornell.edu/databases/`. [3] Las imágenes de estos dataset se encuentran en el formato DICOM, que es un formato de imagen en escalada de grises con una profundidad de pixel de 16 bit. Las mismas puede ser leídas mediante la función `dicomread` de Matlab/Octave. Se recomienda, convertir este formato a ppm de 16 bit.

Fecha de entrega

- Formato Electrónico: Martes 26 de Junio de 2018 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 `[TP3]` seguido de la lista de apellidos de los integrantes del grupo separados por punto y coma `;`. Ejemplo: `[TP3] Lennon; McCartney; Starr; Harrison`

  Se ruega no sobrepasar el máximo permitido de archivos adjuntos de 20MB. Tener en cuenta al realizar la entrega de no ajuntar bases de datos disponibles en la web, resultados duplicados o archivos de backup.

- Formato físico: Miércoles 27 de Junio de 2018 a las 18 hs. en la clase de laboratorio.

- Pautas de laboratorio:
  `https://campus.exactas.uba.ar/pluginfile.php/79576/course/section/12820/pautas.pdf`

Importante: El horario es estricto. Los correos recibidos después de la hora indicada serán considerados re-entrega.

[3] Tener en cuenta que se deberán registrar primero.

6.2. B: Código

Matriz.h

```
const double epsilon = 0.000000001;

/// Indexa de 1 a n/m ///

class Matriz
{

public:

    Matriz();

    Matriz(int, int);

    ~Matriz();

    int Filas();

    int Columnas();

    Matriz & Set(double val, int fila, int col);

    double Get(int, int) const;

    void escalar(double k);

    vector<double> multiply(const vector<double> & x) const;

    void resolver(vector<double> & ranking, vector<double> &b);

    void eliminacionGaussiana(vector<double>& b);

    void backwardSubstitution(vector<double>& ranking, vector<double>& b);

    friend bool operator == (const Matriz & a, const Matriz & b);

    friend bool operator != (const Matriz & a, const Matriz & b);

    friend vector<double> operator * (const Matriz & a, const vector<double> & b);

    friend ostream & operator << (ostream & os, const Matriz & matrix);

    void columnaPorMenosPSobreSuGrado(vector<int>& grados, double p);

    int GetNoNulos();


private:

    vector<list<pair<int,double> > > filas_ptr;
    int nnz;

    int filas;
    int columnas;

    void construct(int fila, int columna);
    void validarCoordenadas(int fila, int col) const;


};
```

```cpp
#include "Matriz.h"


Matriz::Matriz()
{
    filas = 0;
    columnas = 0;
}

Matriz::Matriz(int filas, int columnas)
{
    this->construct(filas, columnas);
}

Matriz::~Matriz()
{

}

int Matriz::Filas()
{
    return filas;
}

int Matriz::Columnas()
{
    return columnas;
}

void Matriz::construct(int filas, int columnas)
{
    if (filas < 1 || columnas < 1) {
        throw "La dimension de la matriz no puede ser cero o negativa";
    }

    this->filas = filas;
    this->columnas = columnas;

    list<pair<int,double> > lista;
    vector<list<pair<int,double> > > vec(filas,lista);

    filas_ptr = vec;

    nnz = 0;
}


Matriz & Matriz::Set(double val, int fila, int col)
{

        this->validarCoordenadas(fila, col);

        if (abs(val) < epsilon){
           auto it = filas_ptr[fila-1].begin();
           for (it; it != filas_ptr[fila-1].end() && it->first != col; ++it){}
           if (it != filas_ptr[fila-1].end()){

                filas_ptr[fila-1].erase(it);
                --nnz;
                }
        }else{
            auto it = filas_ptr[fila-1].begin();


            for (it; it != filas_ptr[fila-1].end()&& it->first < col; ++it){}
            if (it != filas_ptr[filas-1].end() && it->first == col)
            {
                it->second = val;
```

```
            }else{
                pair<int,double> nodo;
                nodo.first = col; nodo.second = val;
                if (it == filas_ptr[filas-1].end())
                {
                      filas_ptr[filas-1].emplace(it,nodo);
                }else{
                    if (it->first == col)
                    {
                         it->second = val;
                    }else{
                        filas_ptr[filas-1].insert(it,nodo);
                    }

                }

                 ++nnz;
            }

        }

    return *this;
}



double Matriz::Get(int fila, int col) const
{
    this->validarCoordenadas(fila, col);

    auto it = filas_ptr[fila-1].begin();
    for (it; it != filas_ptr[fila-1].end()&& it->first != col; ++it ){}

    if ( it != filas_ptr[fila-1].end())
    {

           return (it->second);
    }

    return 0;
}

int Matriz::GetNoNulos(){

    return nnz;
}

void Matriz::escalar(double k)
{
    for (int i = 0; i < filas; ++i)
    {
        for (auto it = filas_ptr[i].begin(); it != filas_ptr[i].end(); ++it)
        {
            it->second = it->second*k;
        }
    }
}

vector<double> Matriz::multiply(const vector<double> & x) const
{
    vector<double> res (filas, 0.0);

    for (int i = 0; i < filas; ++i)
    {
        double acum = 0.0;
        for (auto it = filas_ptr[i].begin(); it != filas_ptr[i].end(); ++it)
        {
            acum = acum + it->second*x[it->first];
        }
        res[i] = acum;
    }
```

```
    return res;
}


void Matriz::validarCoordenadas(int fila, int col) const
{
    if (fila < 1 || col < 1 || fila > this->filas || col > this->columnas) {
        throw "Coordenadas fuera del rango";
    }
}


bool operator == (const Matriz & a, const Matriz & b)
{
    return a.filas_ptr == b.filas_ptr;
}


bool operator != (const Matriz & a, const Matriz & b)
{
    return !(a == b);
}


ostream & operator << (ostream & os, const Matriz & matrix)
{
    for (int i = 1; i <= matrix.filas; i++) {
        for (int j = 1; j <= matrix.columnas; j++) {
            if (j != 1) {
                os << " ";
            }

            os << matrix.Get(i, j);
        }

        if (i < matrix.filas) {
            os << endl;
        }
    }

    return os;
}

vector<double> operator * (const Matriz & a, const vector<double> & b) {

    return a.multiply(b);
}


void Matriz::columnaPorMenosPSobreSuGrado(vector<int>& grados, double p){

   for (int i = 0; i < filas; ++i)
    {
        for(auto it = filas_ptr[i].begin(); it != filas_ptr[i].end(); ++it){
            it->second = (-1.0)*(it->second)*p/grados[it->first-1];
        }

    }
}

void Matriz::resolver(vector<double> & ranking, vector<double> &b){

    this->eliminacionGaussiana(b);

    this->backwardSubstitution(ranking,b);

}


void Matriz::eliminacionGaussiana(vector<double>& b){

    for (int k = 0; k < filas; ++k)
    {
```

```
    for (int i = k+1; i < filas; ++i)
      {
          //iterador de la fila k. La que es usada para restarle. Es el primero de la fila porque hasta la columna k-1 fue triangulada.
          auto it_k = filas_ptr[k].begin();
          // iterador de la fila de las columnas debajo del pivot. Si el primer valor es de la columna k+1 entonces tengo que anularlo.
          auto it_i = filas_ptr[i].begin();


         if (it_i->first == k+1)
          {

              double coef = (double) it_i->second/it_k->second;

                for (; it_k != filas_ptr[k].end(); ++it_k)
                {
                     //si en la fila i a la que le quiero anular la col k+1 hay elementos entre las columnas de la fila k los dejo como estan.

                     while(it_i != filas_ptr[i].end() && (it_i->first < it_k->first)){
                        ++it_i;
                     }

                     double resta = 0.0;

                     //si estamos en la msima columna resto.
                     if (it_i != filas_ptr[i].end() && (it_i->first == it_k->first))
                     {
                        resta = it_i->second - coef*it_k->second;
                        if(abs(resta) > epsilon){ //actualizo valor si es distinto de cero e incrmento iterador.
                            it_i->second = resta;
                             ++it_i;
                        }
                        else{ //si es cero lo scao.
                            auto it_temp = it_i;
                            ++it_i;
                            filas_ptr[i].erase(it_temp);
                            --nnz;
                        }

                     }else{ // columna de i es mayor a columna de k. No hay elem no nulo y tengo que insertarlo.

                        double resta = (double) (-1.0)*coef*it_k->second;

                        if(abs(resta) > epsilon) {

                            pair<int,double> nodo; nodo.first = it_k->first; nodo.second = resta;
                            filas_ptr[i].insert(it_i, nodo);
                            ++nnz;
                        }

                     }

                }
                b[i] = b[i] - coef*b[k];

          }
         }
    }

}

void Matriz::backwardSubstitution(vector<double>& ranking, vector<double>& b){

    double acum = 0.0;

    for (int k = filas-1; k >= 0; --k)
    {
        // indice en vals donde empieza la fila k y por lo tanto a_k_k.

        //recorro la fila en busca de coeficientes no nulos que multipliquen a las
        // x_l mayores que a x_i_i.

        auto it = filas_ptr[k].begin();
```

```
        double a_k_k = it->second;
        for (it = ++it; it != filas_ptr[k].end(); ++it){
            acum = acum + it->second*ranking[it->first -1];

        }

        //resto b_i menos el acum y lo divido por el coef de a_i_i.

        ranking[k] = (double) ((b[k]-acum) / a_k_k);
        acum = 0.0;

    }

}
```