Trabajo Práctico 3: Homenaje a Los Pumas

Universidad de Buenos Aires
Facultad de Ciencias Exáctas y Naturales
(FCEyN)
Departamento de Computación

González Sergio (gonzalezsergio2003@yahoo.com.ar)
González Emiliano (XJesse_JamesX@hotmail.com)
Ferro Mariano (eltrencitomasverde@gmail.com)

5 de noviembre, 2007

Resumen

En esta investigación abordaremos un proceso de simulación y reconstrucción, para un análisis tomográfico, mediante el uso de imágenes. Este procedimiento utiliza diversos recursos matemáticos para tornarse resoluble, una de ellos es emplear un gran sistema de ecuaciones donde se alojaran todas las señales emitidas para el proceso de atravezar la imagen, y utilizar métodos conocidos de resolución y aproximación de soluciones sobre ella. Para llevar a buen término este procedimiento, primero se obtienen los valores originales de los pixels de la imagen, luego se la discretiza, y con esto se calculan los tiempos que tardan en recorrer cada celda de la discretización las señales emitidas. Una vez obtenidos estos datos se pasa, ahora sí, al momento de la reconstrucción, intentando aproximar los valores de los pixels a los originales tanto como se pueda. Continuando con el objetivo de simular un análisis real, se genera ruido aleatorio para cada valor de los tiempos obtenidos en el primer paso y se procede a aplicar el método de de resolución de cuadrados mínimos lineales.

Palábras clave:

- Cuadrados mínimos lineales.
- Señales.
- Analisis Tomográfico.
- Ruido.
- Simulación.

Introducción:

Es frecuente el problema de tener un conjunto de mediciones y desear ajustar los datos observados a funciones que describan la relación entre la variable dependiente y la independiente. El problema original parece estar asociado al nombre de Gauss, quien trató de ajustar curvas a datos experimentales obtenidos en observaciones astronómicas. En nuestros días el planteamiento permite establecer un modelo matemático basado en optimización, para dar solución al problema de calcular los coeficientes que satisfacen el criterio de mínimos cuadrados.

Mínimos cuadrados es una técnica de optimización matemática que, dada una serie de mediciones, intenta encontrar una función que se aproxime a los datos (un "mejor ajuste"). Intenta minimizar la suma de cuadrados de las diferencias ordenadas (llamadas residuos) entre los puntos generados por la función y los correspondientes en los datos. Existen variantes para esta técnica, entre las que se encuentran el método de mínimos cuadrados promedio, mínimos cuadrados ponderados, mínimos cuadrados lineales, siendo esta última la utilizada durante este trabajo.

Un requisito implícito para que funcione el método de mínimos cuadrados es que los errores de cada medida estén distribuidos de forma aleatoria. El teorema de Gauss-Markov prueba que los estimadores mínimos cuadráticos carecen de sesgo y que el muestreo de datos no tiene que ajustarse, por ejemplo, a una distribución normal. También es importante que los datos recogidos estén bien escogidos, para que permitan visibilidad en las variables que han de ser resueltas (para dar más peso a un dato en particular, se emplea mínimos cuadrados ponderados).

Para explicar como funciona el método de aproximación por cuadrados mínimos lineales veamos un pequeño ejemplo: Supóngase que el conjunto de datos consiste en los puntos (xi,yi) siendo i=1,2,...,n . Queremos encontrar una función f tal que $f(x_i) \approx y_i$. Para llegar a este objetivo, suponemos que la función f es de una forma particular que contenga algunos parámetros que necesitamos determinar. Por ejemplo, supongamos que es cuadrática, lo que quiere decir que $f(x) = ax^2 + bx + c$, donde no conocemos aún , y . Ahora buscamos los valores de , y que minimicen la suma de los cuadrados de los residuos (S):

$$S = \sum_{i=1}^{n} (y_i - f(x_i))^2$$

Esto explica el nombre de mínimos cuadrados.

El caso que nos atañe es aquel en el que las variables se presentan en forma lineal, lo que reduce la dificultad del problema a un sistema de ecuaciones lineales. Este sistema se puede definir en forma matricial a través de su ecuación normal:

$$A^t Ax = A^t b$$

Esta fórmula siempre tiene solución, aunque puede no ser única.

En nuestro caso particular, debemos emular el comportamiento de un tomógrafo, que reconstruye una imagen (por lo general una parte del cuerpo vista desde dentro) mediante la interpretación de los tiempos que necesita un grupo de señales para atravesar el objeto analizado. Considerando las mediciones, el aparato dictamina que tipo de tejido (por la densidad) se halla en el camino de cada señal acústica.

Para ello, contamos con un cuerpo que suponemos bidimensional y cuadrado, representado con una imagen, que será discretizada, y sometida a un bombardeo de señales. Con las distancias que hayan recorrido éstas, y las velocidades con que pasen por cada punto de la discretización se calculará cuanto tiempo le llevo a cada señal atravesar el cuerpo. Este proceso es el emulador del procedimiento tomográfico.

Una vez echo esto, se perturban los tiempos de recorrida y, con los datos de las distancias, se trata de volver a la imagen original, reconstruyendo las velocidades. Para ello se emplea el método de cuadrados mínimos lineales.

Como la imagen reconstruida es una aproximación, se utiliza el error cuadrático medio para apreciar la diferencia con la original. Esta función contrastará los valores de la reconstrucción y la imagen original, para medir la calidad de la aproximación.

Desarrollo:

El problema que se plantea, es el de simular el proceso de tomografía y reconstrucción de cuerpos en dos dimensiones. Para hacer esto se trabaja a partir de una imagen en escala de grises, con lo que las tres componentes (RGB) de cada píxel tienen el mismo valor. Sabemos por las condiciones del trabajo que ésta debe ser cuadrada. Para trabajar con ella., la imagen es discretizada en la cantidad de pixels donde cada uno de estos es considerado un cuadrado de dimensión uno. El valor de un píxel corresponde a la velocidad que tendría una señal acústica al pasar por ese sector.

La imagen se encuentra en formato BMP, éste contiene una cabecera de 54 Bytes con los datos del archivo y las características de la imagen, el tipo, ancho y alto en pixels, tamaño del archivo y tamaño de la cabecera, seguido de la imagen propiamente dicha. Esta información tiene importancia al momento de obtener la matriz con los pixels que conforman la imagen, por lo tanto se implementó un procedimiento que obtiene dicha matriz de manera que se pueda abordar el problema sin preocuparnos por datos ajenos a la estructura que queremos utilizar como modelo. Además, para el manejo de las imagenes se esta empleando una librería para C++, EasyBMP, que realiza todas las manipulaciones pertinentes sobre el encabezado, y crea los flujos de entrada y salida de a cuerdo a nuestras necesidades.

Las velocidades obtenidas se guardan en un tipo Matriz, diseñada especialmente con los métodos nativos a esta como trasponer, triangular (que utiliza el método de Gauss con pivoteo parcial), resolver (que implementa sustitución hacia atrás y resuelve el sistema de ecuaciones), multiplicar y operaciones para copiar y obtener sus características (cantidad del filas, cantidad de columnas y

visualizar un elemento en alguna posición).

Una vez hecho esto se generarán las señales que atravesarán el cuerpo. Hay tres métodos diferentes para distribuir las señales, sin embargo en el programa final solo habrá uno disponible, los otros se incluyen ya que con ellos se realizaron pruebas y se compararon sus resultados para elegir el método que consideramos más eficiente.

Antes de explicar las diferentes distribuciones es necesario conocer como se genera una señal. El mecanismo es simple, ya que no se utiliza más que dos puntos de la misma para obtener un punto de pase y una pendiente, con ellos se genera la recta (de la forma $Y = aX + b$), una vez conocidos éstos, se hace variar el valor de X sobre los enteros que estén dentro del rango de la dimensión de la matriz que contiene la imagen (que por disposición del trabajo es cuadrada) obteniendo los correspondientes Y. La operación se repite para valores de Y, despejando en este caso el valor correspondiente al eje X. De esta forma se obtienen los pares (X, Y) que están incluidos en los bordes de los cuadrados que representan los pixels y por donde pasa la señal lanzada. Una vez hecho esto los pares son ordenados de menor a mayor con respecto a la variable X y luego se anulan los repetidos (esto puede ocurrir cuando, por ejemplo, la señal pasa por las esquinas de los cuadrados) y se calcula la distancia recorrida por la señal en cada píxel por el cual pasó.

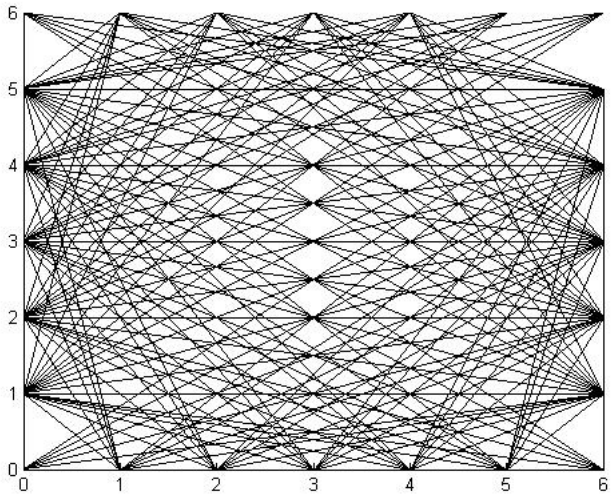
El procedimiento anterior se hace una vez por cada señal a emitir, y su finalidad es ir llenando la matriz que contiene las distancias (en el enunciado del trabajo, la matriz llamada D).

Explicada la manera para generar una señal, se pasan a describir los tres métodos que las distribuyen:

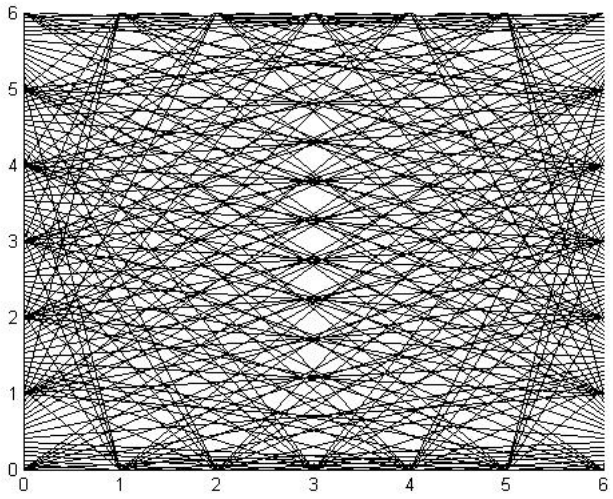
- El primer método se estaciona en cada coordenada entera sobre las "paredes"derecha e izquierda de la matriz discretizada. Desde estos puntos lanza rectas contra cada una de las otras tres "paredes", uniendo el origen de cada andanada de señales, con todas las coordenadas enteras que delimitan la discretización de la imagen en ejes cartesianos. Este método genera $6n^2 - 2n$ señales, siendo n la dimension de la matriz de pixels, ya que por cada uno de los 2n-2 puntos de las paredes lanza 3n-3 señales. Esto último se debe a que no se contemplan las rectas verticales en este modelo.

- El segundo método posiciona una fuente de señales en la base de la discretización y lanza una recta a cada punto entero del eje X, otra fuente, situada a aproximadamente un tercio de la altura del mismo cuadrado en la pared izquierda irradia cada coordenada entera de la pared opuesta, una tercera, colocada aproximadamente a dos tercios de la altura (siempre dentro de la misma area de la imagen discretizada), traza rectas contra el "techo". Luego de disparar una señal, a cada uno de estos puntos-fuente se les incrementa en un decimal (que crece con cada iteración) la coordenada correspondiente a las Y. De esta forma, cada segmento de longitud uno sobre el eje Y, bombardea las tres paredes generando un total de 3n-1 señales por cada uno, con lo que al terminar de recorrer la dimensión de

la imagen tenemos $3(n-1)$ rectas que surcan nuestra discretización. Este procedimiento se realiza también utilizando la pared derecha como fuente, lo que nos da un total de $6(n-1)$ aces.



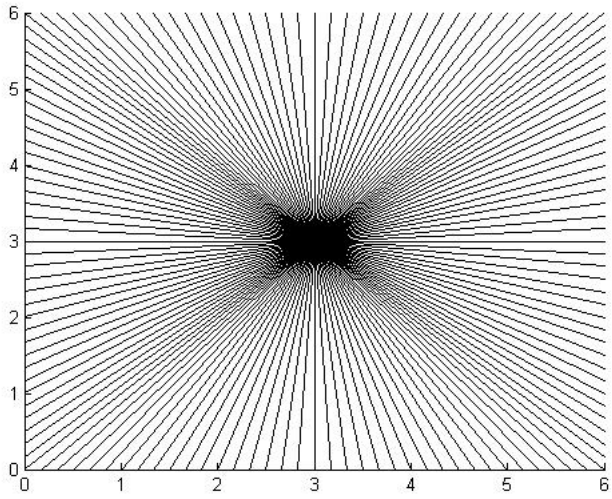
Metodo 1



Metodo 2

- El tercer método, lanza rectas desde la pared izquierda y la base de la discretización, comenzando en lo que sería el origen de nuestro sistema cartesiano y avanzando de a pasos de 1/n hacia los extremos. Toma como epicentro para todas las señales el punto central de la imagen, mientras

que el segundo punto para calcular cada recta es el antes mencionado. Considerando que este método se detiene cuando ambos focos alcanzan el extremo correspondiente de la imagen, cada uno emite n*n señales acústicas, lo que da un total de $2n^2$, siendo éste el procedimiento que menos rectas traza.



Metodo 3

Algunas aclaraciones comunes a todos los métodos:

- Cada método borra (si la hubiera) la anterior matriz D, ya que puede ser que las dimensiones de dicha estructura no concuerden con la distribución a efectuarse.
- Los procedimientos son estáticos, es decir, no importa en que circunstancias se ejecuten, siempre siguen el mismo patron.
- En ninguna distribución se contempla la posibilidad de lanzar rectas verticales.
- El 'n' al que se hace referencia es la dimensión de la imagen cuadrada.

Una vez trasladados todos los datos a la matriz D pasamos a resolver el sistema, que consiste en la matriz antes mencionada, multiplicada por una matriz columna con las velocidades invertidas e igualada a otro vector columna cuyos componentes resultarán ser los tiempos de recorrida de las señales en el total de la imagen.

En este punto conocemos todos los datos del problema: las velocidades, las distancias y los tiempos de recorrida de cada señal en forma exacta. Hasta aquí consideramos que resolvimos una primera parte del trabajo que denominamos "la ida".

Completada esta etapa, acometeremos el verdadero objetivo del trabajo. Éste consiste en reconstruir la imagen original (que es equivalente a obtener las velocidades) a partir de las distancias (la matriz D) y los tiempos (que en esta etapa son distorsionados en un grado elegido por el usuario). Para reconstruir los datos, aplicamos la técnica de cuadrados mínimos lineales, y utilizando las ecuaciones normales resolvemos el sistema, aplicando eliminación gaussiana con pivoteo parcial. En el caso de que el pivoteo parcial no pueda efectuarse (si solo hay ceros debajo del elemento de la diagonal) se utiliza la técnica de pivoteo total.

Resuelto el sistema, se obtiene un vector columna que contiene las velocidades invertidas. Cabe aclarar que, en este punto, las velocidades son una aproximación de las originales, ya que el método de cuadrados mínimos puede no obtener los valores reales de la imagen debido al ruido introducido en los tiempos .Con éstas, se procede a reconstruir la matriz de pixels, y convertirla a formato BMP. Este proceso implica copiar tres veces el valor de cada velocidad (para formar un píxel con las componentes RGB igualadas, por ser la imagen una escala de grises) y de agregar al final de cada fila de pixels los Bytes de "basura"que sean necesarios según el formato. Recordemos que para estas tareas se emplea una librería especializada para manipulación de BMP.

Detalles de la implementación:

Todo este proceso esta sustentado en los módulos 'Matriz' y 'Senales', el primero incluye el utilizado en el trabajo anterior, con los siguientes agregados y modificaciones:

- Traspuesta: esta función recibe un objeto matriz por referencia, y altera los punteros del miembro privado 'm', que era la estructura elegida para almacenar los datos de nuestro objeto (arreglo de arreglos de C++). Ya que modifica la matriz, no devuelve valor alguno.

- Multiplicar: este método debe ser aplicado sobre la matriz resultado, y recibe dos matrices que serán los factores. El procedimiento para multiplicar es el estándar (fila por columna) y con el se van llenando los elementos de la matriz resultado uno a uno.

- Triangular: trabaja en forma muy similar al método original, salvo que fue adaptado para que pueda recibir matrices no cuadradas. Además, siempre que se puede se utiliza el pivoteo parcial, y cuando no, se agrego el pivoteo total.

- CuadradosMinimosLineales: este procedimiento recibe dos matrices (A y b) y devuelve el resultado (X) de la ecuación normal $A^tAX = A^tb$, en la matriz que se utilizó para llamarlo. El método esta dividido en dos series de ciclos anidados:

  El primero calcula A^tA, no mediante el método explicado arriba, sino aprovechando el hecho de que A^tA es simétrica. Toma los elementos de A,

y los multiplica por su "traspuesto"(trocando filas por columnas), tras lo cual suma cada resultado y lo coloca en una nueva matriz.

El otro grupo de ciclos se encarga de calcular $A^t b$, toma cada elemento de A como si fuera su "traspuesto", y lo multiplica por los elementos de b, para luego guardarlos en una tercera matriz.

De esta forma se trata de evitar el overhead de memoria, trabajando in situ en lugar de emplear varias matrices temporales para los resultados intermedios, como A^t. Sin embargo, $A^t A$ y $A^t b$ consumen bastante memoria, pero esta será liberada al terminar la función.

Finalmente se triangula y resuelve el sistema que forman las matrices creadas arriba y la matriz X, utilizando los métodos correspondientes.

Hasta aquí los cambios que sufrió el módulo 'Matriz', ahora comentaremos las características de los métodos del módulo 'Senales':

Esta clase guarda sus datos en un objeto Matriz, siendo ésta la única estructura de datos compleja empleada por el módulo. La dimensión de la imagen (el ancho o el alto, de ahora en mas 'n') y la cantidad de señales que la recorren estan almacenados en sendos enteros sin signo como miembros privados de la clase. La matriz (que a partir de ahora será llamada D) tiene dimensión $m * n^2$, donde m es la cantidad de señales emitidas para surcar el cuerpo, y n^2 es la cantidad de celdas en las que fue discretizada la imagen (siendo esta cuadrada y de lado de longitud n). En ésta, cada fila almacena la distancia recorrida por la señal correspondiente en cada una de las celdas de la discretización, éstas se distribuyen en la fila con un criterio lineal, una fila de celdas es seguida por su inmediata siguiente, hasta completar las n filas. Resulta claro tras un breve análisis que las señales recorren como máximo una cantidad igual a $2n - 1$ celdas, los valores que corresponden a los cuadrados no tocados por una recta son cero.

Esta forma de guardar los datos apareja una distribución no previsible de los valores distintos de cero, a menos que se conozca el método empleado (en detalle) y el tamaño de la imagen. Por ello no es sencillo aprovechar el hecho de que D es una matriz rala.

Nótese que este módulo fue pensado únicamente para la resolución de este trabajo, de hecho, hasta último momento las funciones que serán descriptas a continuación estaban diseminadas, sin formar parte de una clase. Es por ello que el modulo no ofrece posibilidades de reutilización (a diferencia del modulo 'Matriz') y está tan embebido del problema que nos acaece.

En cuanto a los métodos que completan la clase, 'Senales' cuenta con constructor por defecto y por copia, pero la función que hace trabajar verdaderamente este módulo es su constructor con parámetros. En él se llama a los métodos de distribución de las señales, en este momento, para testear cada uno y comprobar cual es el más idóneo para las distintas imágenes, todos los métodos están disponibles para ser llamados. El constructor recibe el ancho de la imagen y un numero que decide que método de distribución se utilizará (ambos unsigned int). En el caso de que pretenda llamarse un método que no es ninguno

de los tres disponibles (por ejemplo un hipotetico método cinco), se lanza un mensaje de error y se ejecuta por defecto el número uno.

Otras funciones relevantes de 'Senales' son:

- realizarTomografia: este método condensa gran parte del trabajo realizado, ya que es el responsable de obtener los tiempos de recorrida de las señales (mediante una multiplicación de distancias y velocidades inversas, las distancias son miembro de la clase, y las velocidades son una matriz pasada como parámetro). Luego estos tiempos son modificados, por un valor que es función del último valor (factorRuido) que se le da a este procedimiento. Con éstos, se trata de aproximar las velocidades originales a través del método de cuadrados mínimos. Esta aproximación se devuelve en el mismo parámetro en el cual se ingresaron las velocidades inversas.

Los métodos de emisión de señales son públicos, y ya han sido comentados más arriba.

- tirarSenal: También sobre éste ya se han dado algunos datos sobre su comportamiento, pero ahora se lo abordara en detalle. Este procedimiento recibe cuatro valores long double y un entero. Los primeros son dos parejas de puntos (X,Y), que se toman como puntos de paso de una hipotética recta, que será construida utilizando estas coordenadas en las dos ecuaciones necesarias para despejar pendiente y punto de pase (real). El parámetro restante, indica qué fila de la matriz D se debe completar con la corriente llamada al método. Los datos con los que se llenará son calculados de la siguiente manera: en un arreglo de tuplas se guardan los pares (X,Y) para cada X entero, según la ecuacion de la recta y el 'a' y el 'b' conocidos, mientras se van almacenando se marcan los Y que tienen el mismo valor que las X (cuando la recta traza una diagonal por el cuadrado de la discretización). Tras completar todas las coordenadas enteras sobre abscisas, se procede a llenar la segunda parte del arreglo, mediante un procedimiento equivalente sobre las coordenadas Y. Es en este momento cuando se contrastan los elementos que se van obteniendo con el registro antes mencionado, de forma que no se cuenten dos veces los pares de elementos enteros. Al terminar, tenemos un arreglo de longitud $2 * n$, en el cual la primera mitad esta ocupada por las coordenadas con los X enteros, y en la segunda se hallan las tuplas con Y entero. Nótese que ésta puede no estar completa debido a que se ignoran los repetidos. Con estos datos se procede a completar la fila indicada por el parámetro correspondiente, obteniendo la distancia entre los puntos mediante el cálculo de la norma dos. Recordemos que el criterio de llenado de las filas era lineal, es decir que los primeros n elementos de la fila se corresponden con la distancia recorrida en cada uno de los cuadrados de la base de la discretización, los segundos con las coordenadas que tienen X en uno etc. Una vez hecho todo lo anterior se libera la memoria de los arreglos creados y con ello se termina el método.

- ordenarParesMenorAMayor: Este método recibe un arreglo de pares para ordenar y la longitud de este arreglo. Ordena los pares de menor a mayor con un Selection Sort.
- ordenarParesMayorAMenor: Mismo que el anterior pero ordena de mayor a menor.
- anularRepetidos: A esta función se le pasa un arreglo de pares y su longitud. El método recorre el arreglo y cuando encuentra dos elementos iguales elimina el segundo, poniendo un -1 en su lugar.

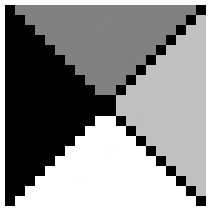
Resultados:

En esta sección se procedera a la muestra de diferentes resultados realizados al sistema, con distintas imágenes. En una primera parte se mostrará como se comportan los diferentes métodos de distribución de señales al momento de reconstruir una imagen. La segunda parte se encargará de explayar graficamente los errores cuadraticos medios de cada método aplicado a una misma imagen de 10x10 pixels.

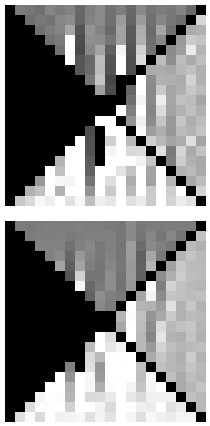
En la parte final se verán los tiempos requeridos para reconstruir una imagen en función del tamaño de la imagen, para obsvervar su eficiencia.

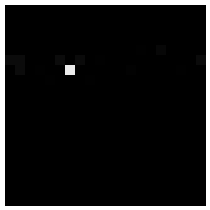
En los siguientes graficos se probaran los 3 metodos de distribución de señales en una imagen de 20x20 pixels, y usando factor ruido igual a 50.

La imagen original es la siguiente:



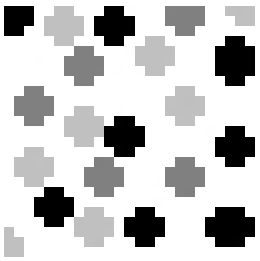
Se prueban en orden los metodos 1, 2 y 3 respectivamente.



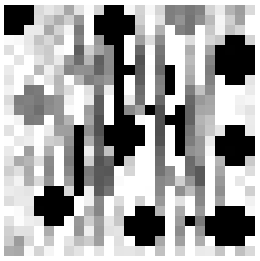


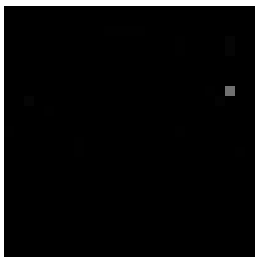
Las siguientes pruebas se realizaran sobre una imagen de 25x25 con nivel de ruido 50.

La imagen original es la siguiente:



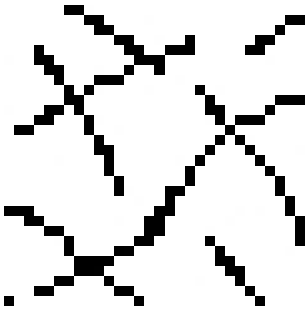
Ahora se muestran en orden los metodos 1, 2 y 3.



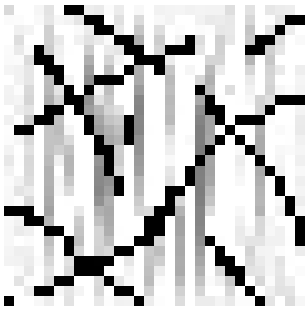


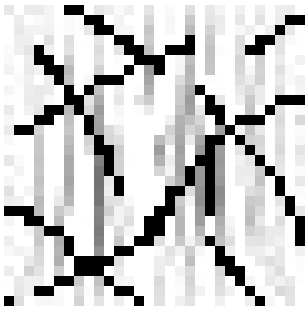
Finalizando esta primera parte pruebas, se muestran los metodos sobre una imagen de 30x30 pixels y con grado de ruido 50.

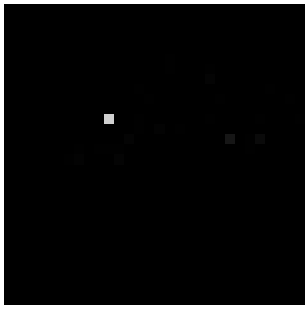
La imagen original:



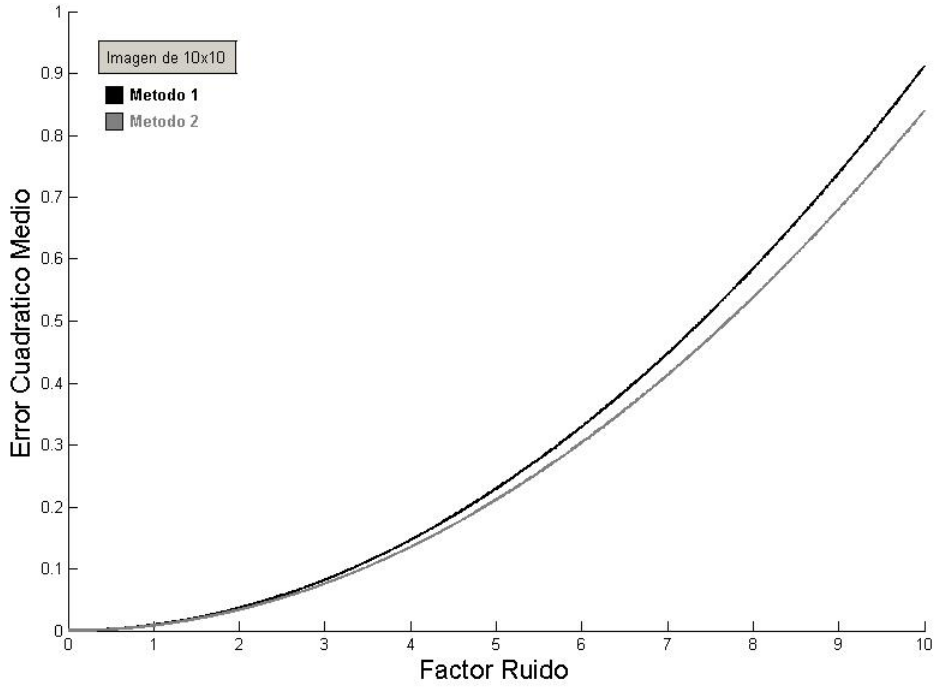
Ahora los metodos en orden:



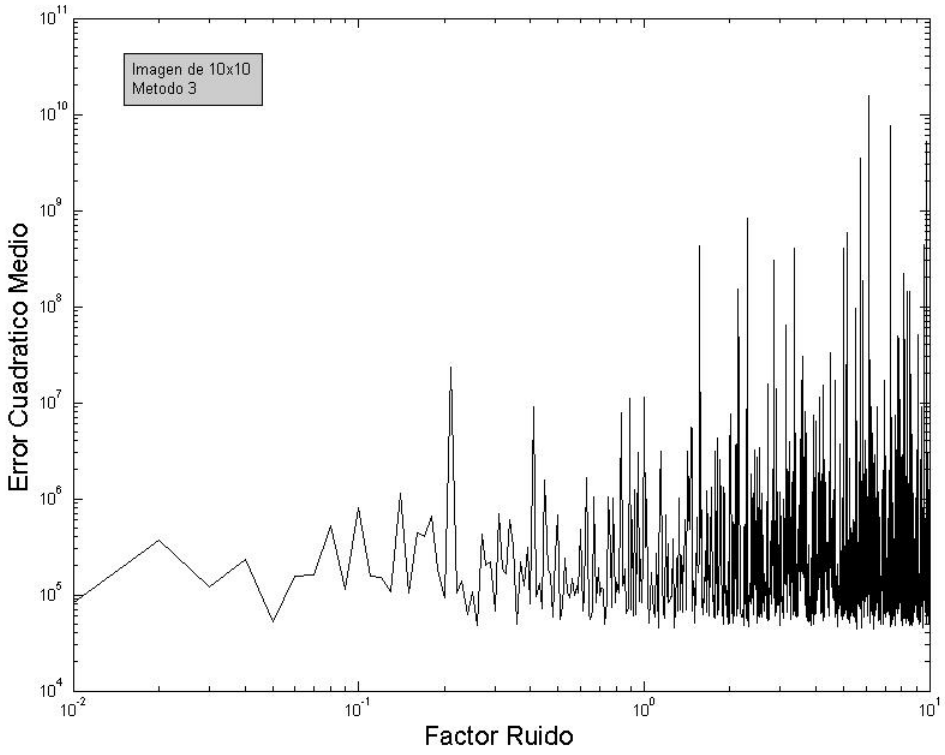




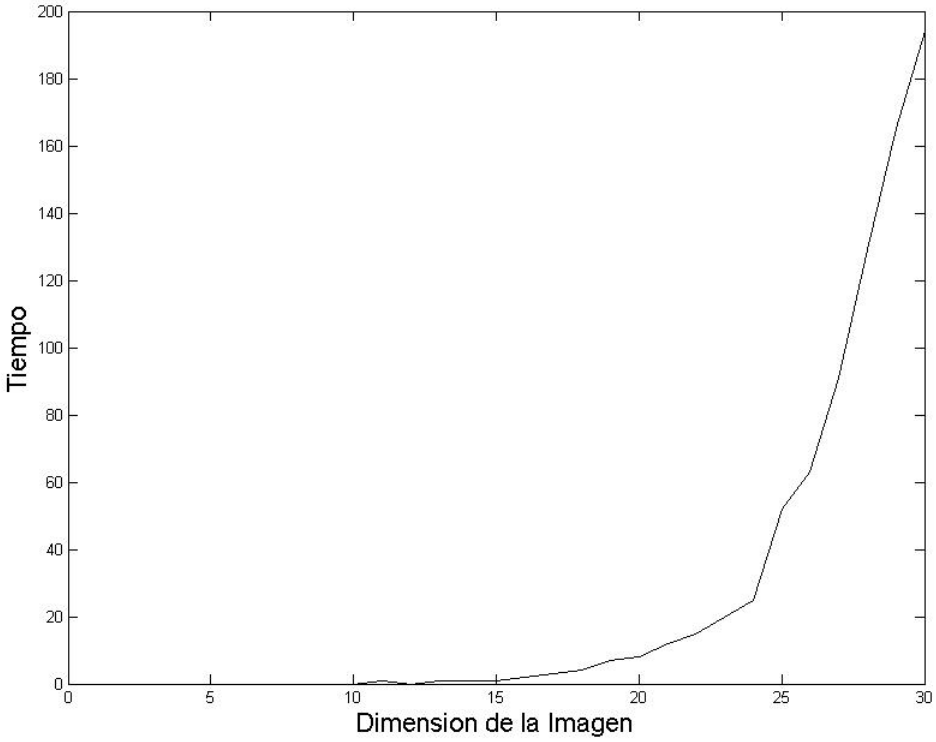
La segunda parte de nuestras pruebas muestra el error cuadratico medio en función del factor ruido para los métodos 1 y 2, sobre una imagen de 10x10 pixels.



Para el método 3, mostraremos un gráfico aparte ya que el error cuadrático medio en función del factor ruido se comporta de forma muy inestable para la misma imagen de 10x10 pixels. En este caso se observan valores más elevados de lo que espararíamos para el error cuadrático medio para niveles de ruido muy pequeños, esto indica que el método no es eficiente (y tal vez ni siquiera eficaz) a la hora de reconstruir la imagen original.



En esta ultima parte, el siguiente gráfico mostrara el tiempo (en segundos) requerido para reconstruir una imagen en función del tamaño en pixels, en este caso las dimensiones de la imagen estan acotadas a 30x30 pixels como máximo.



Discusiones:

Como se pudo apreciar en la sección Resultados, no fueron incluidas reconstrucciones sin ruido, ya que estas son totalmente fieles a las imágenes originales para los métodos 1 y 2, esto sucede gracias a que, con ambas distribuciones la técnica de cuadrados mínimos lineales cuenta con suficiente información como para dar una aproximación 'exacta', y porque concuerdan perfectamente los datos con los que se la utiliza.

Nótese que en las ecuaciones normales que empleamos para resolver el método de CML se puede quitar la At y nos queda el sistema original.

A la luz de las imágenes aproximadas por el método 3, se puede decir que éste tiene una performance bastante menor a los otros dos. Esto se debe a que es el diagrama que menos rectas traza, o también (sin ser excluyente) a una mala distribución de las señales al momento de surcar la imagen.

Como era de esperarse, las imágenes respetan la relación inversa entre calidad y ruido. Esto sucede a causa de la disparidad entre los datos originales y los tiempos que utiliza la técnica de CML. De todas formas consideramos que ésta tiene un muy buen desempeño al aproximar las velocidades, teniendo en cuenta que la magnitud del ruido llega a ser bastante alta.

Por último, casi no se aprecia diferencia entre las aproximaciones realizadas con los métodos 1 y 2, esto era bastante esperable, ya que como puede observarse en el gráfico correspondiente sus ECM, se mantienen en ordenes similiares para todas las dimensiones, con distintos factores de ruido.

Las magnitudes con que se distorsionan los tiempos de recorridas forman un elemento crucial al momento de la reconstrucción. Si bien éstas no son un resultado del trabajo, su cálculo afecta a todas las imágenes que sean producto de nuestro programa. Es por esto que se incluyen en esta sección nuestras apreciaciones sobre el tema.

Una de las cosas que nos consumió bastante tiempo de pruebas y deliberación fue el poder encontrar una expresión que simule bien el factor ruido. En total surgieron tres versiones diferentes de ella. La primera tenía la siguiente ecuación:

$$(longdouble)((rand()\,\%100) + 1) * (factorRuido/1000))$$

La idea era calcular el ruido dependiendo de un número pseudo-aleatorio (random) acotado entre uno y cien de manera tal de poder hacer una suerte de porcentaje del valor original. La división por mil hace que los valores de la cuenta no se excedan en magnitud, tornándose muy elevados o irreales. Este método se descarto al surgir otra manera de calcular el ruido, ya que la fórmula anterior no tenía la posibilidad de que el ruido fuese cero, y además los valores que calculaba esta expresión se hacían demasiado grandes, ya que consideramos que el parámetro tenía que ser más bien pequeño, sin contar que nos resultó incomodo a la hora de testear el programa.

Una vez reformulado, el segundo método quedo expresado de la siguiente manera

$$(longdouble)(rand()\,\%100) * (factorRuido/1000000)$$

Para resolver los problemas de la función anterior (para que el factor sea más pequeño), optamos por aumentar el divisor a 1000000. Después de hacer varias pruebas nos dimos cuenta que este método al calcular el "mod 100.$^{\text{en}}$ el random hacia que el ruido introducido fuese demasiado inestable, y deseábamos algo que a pesar de ser aleatorio, tuviera una distribución más uniforme del ruido. Por ello, abandonamos esta segunda forma de cálculo para buscar una nueva expresión.

El tercer y último método se obtiene mediante el siguiente cálculo

$$(longdouble)(rand()\,\%10) * (factorRuido/100000))$$

Este solucionó los problemas de inestabilidad que se presentaba con el algoritmo anterior, y centra los valores del ruido en diez posibilidades, que si bien no son muchas, consideramos una cantidad suficiente para distorsionar los tiempos de recorrida. Realizando distintas pruebas nos dimos por satisfechos y creemos que esta expresión funciona bastante bien en en cuanto a los valores de ruido que genera.

Conclusiones:

Con respecto a lo observado podemos sacar las siguientes conclusiones: en las pruebas de error cuadrático medio se vio un resultado menor para imágenes que contienen valores más bajos, más cercanos a 0 (colores más oscuros, en escala de grises), esto se debe al cálculo que se realiza para obtener los tiempos, ya que al hacer las cuentas y tener una velocidad (valor del pixel) muy grande, la división se vuelve chica por lo cual se esta más propenso a errores grandes de redondeo, en contraste con un número calculado con un menor valor de la velocidad. Esto implica tener los datos con un cierto nivel de error, que al momento de hacer la reconstrucción, se suma al que ya es propio de la aproximación.

Con esto se puede inferir lo siguiente: imágenes (en escala de grises) con mayoría de pixels oscuros, tienen más oportunidad de generar error adicional al cometido durante la reconstrucción. Sin embargo, teniendo un método de distribución de señales eficiente, esto no se vuelve un problema grave, lo que es seguro, es que su incidencia en el resultado final debe ser tenido en cuenta.

Siguendo con los patrones de distribución, se observó durante el trabajo que no siempre el mejor método para reconstruir una imagen es el que llega a atravezar todos los puntos de la discretizacion, si no el que puede surcar cada punto con más de una recta y además en diferentes direcciones, la mayor cantidad que sea posible, en función de recopilar la mayor cantidad de datos posibles para cada sector a reconstruir. En la sección de resultados se puede observar que el método 3 no es nada eficiente a pesar que genera un número importante de señales. Esto se debe a que la distribución de la emisión de señales no es óptima por lo explicado anteriormente. En síntesis, se puede concluir que el resultado a obtener esta directamente relacionado (entre otras cosas) con el tipo de distribucion de estas señales, y que una mala elección de esta puede llevar a resultados incongurentes con lo esperado y por lo tanto incorrectos.

En cuanto al aspecto de eficiencia computacional en memoria y tiempo podemos decir que nos manejamos en rangos aceptables, aunque sin duda mejorables, en particular, con una revision del código y una redistribución y reagrupación del mismo se podrían llegar a bajar las constantes de la complejidad. Sin embargo, se prefirió asignar más tiempo a las pruebas experimentales en lugar de pulir el código, en detrimento de la eficiencia. Cabe aclarar que con ese trabajo tan solo se podrían disminuir las constantes, que no son el factor preponderante en el calculo de complejidades asintóticas. Para calcular los diagramas de distribución de las señales, incurrimos en una complejidad (en los tres métodos) de orden cúbico, es decir O(n^3), en la dimensión de la matriz que se desea analizar. Esto se debe a que se ejecuta, generalizando, el método tirarSenal, que es O(n), dentro de ciclos anidados que lo repiten n^2 veces (todo sin tener en cuenta las constantes que multiplican estas magnitudes), lo que, por propiedades del "álgebra de ordenes", nos da una complejidad cúbica. El otro método que incurre en una complejidad similar es cuadradosMinimosLineales, del módulo Matriz. Las multiplicaciones que tiene que realizar éste hacen que su orden sea O($fil2 * col$) y la función resolver que emplea es de una complejidad O(col^3), con fil y col filas y columnas de la matriz D. Siendo ésta mas 'alta' que 'ancha' (todos las

distribuciones de señales generan muchas más rectas que n^2) es preponderante la complejidad cúbica en las columnas (col es del orden de n^2), así el algoritmo tiene un abrumador orden final de $O(n^6)$, lo que vuelve un tanto fútil bajar las constantes con la intención de mejorar los tiempos de ejecución. Por esto, el limite de ancho o alto para imágenes a tratar con nuestro software es de 30 pixels, superarlo no lo hace fallar, pero insume una gran cantidad de tiempo. El tema de la memoria es un tanto diferente, el manejo de matrices grandes trae, inherentemente, un consumo de espacio importante, y el hecho de no conocer con exactitud la distribución de los elementos distintos de cero (ya se hizo notar que D es rala) hace difícil cualquier intento de evitar almacenar solamente estos últimos. Se tomaron algunas medidas para paliar esto, como calcular algunos resultados intermedios in situ, pero no se gana gran cosa.

Basándonos en nuestras experiencias durante este trabajo, hemos llegado a las siguientes conclusiones:

- Siempre es deseable una distribución de las señales que minimice la cantidad de filas linealmente dependientes de la matriz D. Con esto se garantiza una buena aproximación mediante los CML.
- Extender estos métodos a imágenes de dimensión mayor a 30, no tiene problemas en el plano teorico, pero encuentra trabas en el práctico debido a las grandes complejidades temporales. Se descuenta que en futuras versiones se optimizara el código.
- La magnitud del ruido introducido es un factor esencial, que influye grandemente en el resultado final del trabajo. Este es un componente muy sensible en la implementación, que debe ser manipulado con mucho cuidado y a conciencia.

Apendice A:

Enunciado:

Laboratorio de Métodos Numéricos - Segundo cuatrimestre 2007
Trabajo Práctico Número 3: Homenaje a Los Pumas

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

El método de reconstrucción

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

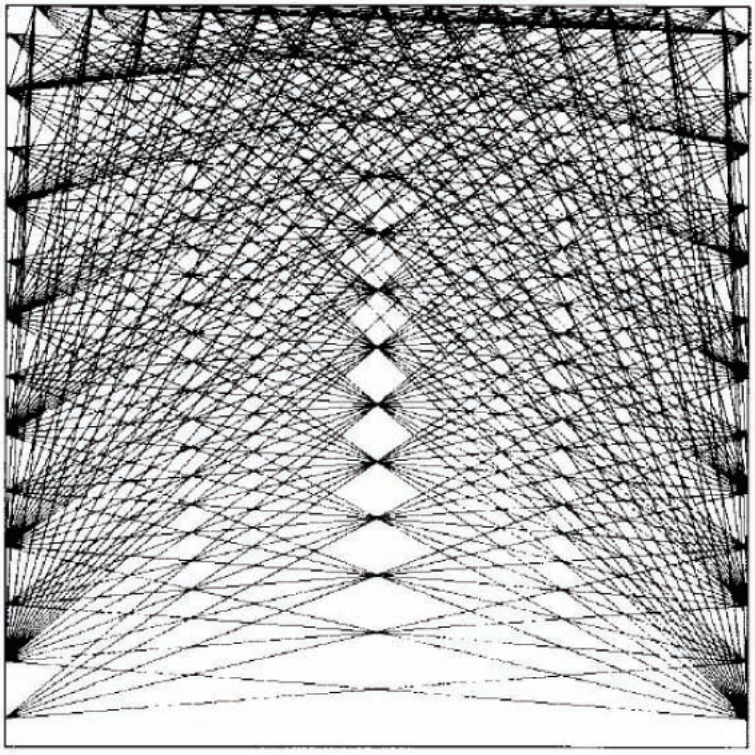


Figura 1: Ejemplo de configuración de las señales.

Suponemos que el cuerpo está dividido en $n \times n$ celdas cuadradas, de modo tal que en cada celda las señales acústicas tienen velocidad constante. Si d_{ij}^k es la distancia que recorre la k-ésima señal en la celda ij (notar que $d_{ij}^k = 0$ si

la señal no pasa por esta celda) y v_{ij} es la velocidad de la señal en esa celda, entonces el tiempo de recorrido de la señal completa es de

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

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 $t \in \mathbf{R}^m$, tal que t_k indica el tiempo de recorrida de la k-ésima señal. Sea $D \in \mathbf{R}^{m \times n^2}$ 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, 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 $s \in \mathbf{R}^{n^2}$ 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 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 considera que el valor de cada *pixel* de la imagen corresponde a la velocidad de recorrida de las señales acústicas 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 debe 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[1].

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

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

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.

Objetivos adicionales

En forma optativa, se sugiere probar con distintas estrategias geométricas para generar las señales acústicas (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.

Fecha de entrega: Lunes 5 de Noviembre

[1] Un formato para almacenar imágenes que es de fácil lectura y tiene cierta difusión es el formato RAW, ver por ejemplo `http://local.wasp.uwa.edu.au/~pbourke/dataformats/bitmaps`.

Apendice B:

1. **Senales.h**

```
#ifndef _SENALES_H
#define _SENALES_H

#include <iostream>
#include <string.h>
#include <assert.h>
#include "Matriz.h"
using namespace std;

class Senales{
  public:
    Senales(){D = NULL;}
    Senales(unsigned int dimension, unsigned int metodo);
    Senales(Senales& s);

    //hace un txt para graficar el metodo, enviado por parametro, en matlab
    void realizarTomografia(Matriz& velocidades, long double factorRuido);
    void prepararParaGraficarMetodo(ostream & os, int metodo);
    unsigned int getCantidadSenales(void);
    ~Senales();

  private:
    unsigned int dimension;
    unsigned int numSenales;
    Matriz* D;

    //tira señales primero desde la pared izquierda hacia las demas
    //y despues desde la pared derecha hacia las demas
    void metodo1(void);

    //similar que el metodo uno, salvo que por cada señal enviada se desplaza
    //el punto de pase.
    void metodo2(void);

    void metodo3(void);

    //Toma una pendiente, un punto de pase para generar una recta
    //una matriz, un n que representa la dimension de la matriz
    //imagen y el numero de la matriz a llenar
    void tirarSenal(long double x1, long double y1, long double x2,
                    long double y2, int filaALlenar);

    //ordena un vector de tuplas por su primer componente de menor a mayor
    void ordenarParesMenorAMayor(long double** pares, int filas);
```

```
    //ordena un vector de tuplas por su primer componente de mayor a menor
    void ordenarParesMayorAMenor(long double** pares, int filas);

    //recorre el vector de pares de coordenadas y anula los
    //repetidos poniendo un -1 en la coordenada x, indicando que
    //esta repetida.
    void anularRepetidos(long double** pares, int cantPares);

    void graficarMetodo1(ostream & os);
    void graficarMetodo2(ostream & os);
    void graficarMetodo3(ostream & os);
};

#endif /*_SENALES_H*/
```

2. Senales.cpp

```
#include <math.h>
#include "Matriz.h"
#include "Senales.h"

/************************************/
/*          METODOS PUBLICOS        */
/************************************/

Senales :: Senales(unsigned int dimImagen, unsigned int metodo)
{
    D = NULL;
    dimension = dimImagen;
    switch(metodo){
        case 1:
            metodo1();
            break;
        case 2:
            metodo2();
            break;
        case 3:
            metodo3();
            break;
        default:
            cout << "No existe el metodo. En su lugar se ejecutara el metodo 1";
            cout << endl << endl;
            metodo1();
            break;
    }
}

Senales :: Senales(Senales& s)
{
    D = new Matriz(s.D->filas(), s.D->columnas());
```

```
    *D = *(s.D);
    dimension = s.dimension;
    numSenales = s.numSenales;
}

void Senales :: prepararParaGraficarMetodo(ostream & os, int metodo)
{
    switch(metodo){
        case 1:
            graficarMetodo1(os);
            break;
        case 2:
            graficarMetodo2(os);
            break;
        case 3:
            graficarMetodo3(os);
            break;
        default:
            break;
    }
}

void Senales :: realizarTomografia(Matriz& resultado, long double factorRuido)
{
    Matriz t(numSenales, 1);

    t.multiplicar(*D, resultado);
    //ya tenemos el vector t (tiempos de cada senial) calculado,
    //ahora tenemos que degenerarlo y volver a calcular las
    //velocidades (valores de los pixels) con
    //cuadrados minimos, para reconstruir la imagen

    for (int i = 0; i < t.filas(); i++){
        t.asignar(i, 0, t.ver(i,0) +
                  (long double)(rand() % 100)*(factorRuido/1000000));
    }

    resultado.cuadradosMinimosLineales(*D, t);
}

unsigned int Senales :: getCantidadSenales(void)
{
    return numSenales;
}

Senales :: ~Senales()
{
    delete D;
}
```

```
/************************************/
/*          METODOS PRIVADOS        */
/************************************/

/*
 * metodo1: Tira señales desde las 2 paredes verticales de la imagen hacia los
 *          demas pixels que no esten en la misma pared.
 *          Este metodo genera 6*n^2 señales
 */
void Senales :: metodo1(void)
{
    numSenales = 6*dimension*dimension - 2*dimension;
    //este metodo genera 6*n^2 - 2*n señales, siendo n = dimension
    delete D;
    D = new Matriz(numSenales, dimension*dimension);

    int fila = 0;                                          //fila a llenar
    for(unsigned int i = 1; i < dimension; i++){
        for(unsigned int j = 1; j <= dimension; j++){
            int filaoff = 0;
            //tiro las rectas de la pared izquierda
            //sobre el piso
            tirarSenal(0, i, j, 0, fila);
            filaoff++;
            //sobre pared derecha
            tirarSenal(0, i, dimension, j, fila + filaoff);
            filaoff++;
            if(j != dimension){
                //sobre el techo
                tirarSenal(0, i, j, dimension, fila + filaoff);
                filaoff++;
            }

            //tiro las rectas de la pared derecha
            //sobre el piso
            tirarSenal(dimension, i, j, 0, fila + filaoff);
            filaoff++;
            //sobre pared izquierda
            tirarSenal(dimension, i, 0, j - 1, fila + filaoff);
            filaoff++;
            if(j != dimension){//sobre el techo
                tirarSenal(dimension, i, j - 1, dimension, fila + filaoff);
                filaoff++;
            }

            fila += filaoff;
        }
    }
}
```

```
void Senales :: metodo2(void){
    //este metodo genera 6*n^2 - 2*n señales, siendo n = dimImagen

    numSenales = 6*dimension*dimension - 2*dimension;
    delete D;
    D = new Matriz(6*dimension*dimension - 2*dimension, dimension*dimension);

    int fila = 0;                                   //fila a llenar
    //todos los puntos de las demas paredes
    int cantDestinos = 3*dimension - 1;
    long double desplazamiento = 0.5*(1/(long double)cantDestinos);

    for(unsigned int i = 0; i < dimension; i++){
        for(unsigned int j = 1; j <= dimension; j++){
            int filaoff = 0;
            //pared izquierda
            //tiro la señal hacia el piso
            tirarSenal(0, (long double)j/(long double)cantDestinos +
                       i - desplazamiento, j, 0, fila + filaoff);
            filaoff++;

            //tiro la señal hacia la pared derecha
            tirarSenal(0, (long double)(dimension + j)/(long double)cantDestinos
                       + i - desplazamiento, dimension, j, fila + filaoff);
            filaoff++;

            //tiro la señal hacia el techo
            if (((j != dimension) && (i != (dimension - 1))) ||
               (i == (dimension - 1) && j != dimension  && j != (dimension-1))){
                tirarSenal(0,((long double)2*dimension+j)/
                           (long double)cantDestinos +
                           i, dimension - j - desplazamiento,
                           dimension, fila + filaoff);
                filaoff++;
            }


            //pared derecha
            //tiro la señal hacia el piso
            tirarSenal(dimension, (long double)j/(long double)cantDestinos +
                       i - desplazamiento, dimension - j, 0, fila + filaoff);
            filaoff++;

            //tiro la señal hacia la pared izquierda
            tirarSenal(dimension, (long double)(dimension + j)/
                      (long double)cantDestinos + i - desplazamiento, 0, j,
                      fila + filaoff);
            filaoff++;

            //tiro la señal hacia el techo
```

```
            if (((j != dimension) && (i != (dimension - 1))) ||
               (i == (dimension - 1) && j != dimension  && j != (dimension-1))){
                tirarSenal(dimension, (long double)(2*dimension + j)/
                          (long double)cantDestinos + i, j, dimension,
                          fila + filaoff);
                filaoff++;
            }

            fila += filaoff;
        }
    }
}

void Senales :: metodo3(void){
    //genera 3*(n + 1)
    numSenales = 2*dimension*dimension;
    delete D;
    D = new Matriz(numSenales, dimension*dimension);

    long double xMedio = (long double)dimension/2;
    long double yMedio = (long double)dimension/2;

    int fila = 0;
    //envio las señales desde la pared izquierda
    for(unsigned int i = 0; i <= dimension*dimension; i++){
        tirarSenal(0, (long double)i/(long double)dimension, xMedio, yMedio,
                  fila);
        fila++;
    }

    //envio las señales desde el piso
    for(unsigned int i = 1; i < dimension*dimension; i++){
        tirarSenal((long double)i/(long double)dimension, 0, xMedio, yMedio,
                  fila);
        fila++;
    }
}

void Senales :: tirarSenal(long double x1, long double y1, long double x2,
                           long double y2, int filaALlenar){
    if (x1 == x2){   //si la senial es vertical hacemos las cuentas "a mano"
                     //una senial vertical pasa por toda una columna de pixel
                     //con distancia 1, recorriendo
                     //una distancia de 1 por cada pixel
        for (unsigned int fil = 0; fil < dimension; fil++)
            if (x1 == dimension)
                D->asignar(filaALlenar, fil*dimension + dimension - 1, 1);
            else
                D->asignar(filaALlenar, fil*dimension + (int)x1, 1);
    }
```

```
else{
    //primero construyo la recta
    //tengo un sistema de 2x2 que es: y1 = x1*a + b
    //                                y2 = x2*a + b
    //entonces: b = y1 - x1*a
    //          y2 = x2*a + y1 - x1*a
    //          a = (y2 - y1) / (x2 - x1)
    //          b = y1 - x1*((y2 - y1) / (x2 - x1))
    //          x2 - x1 != 0 paratodo x2, x1 reales, salvo que sea una
    //          senial vertical, lo cual tratamos anteriormente
    long double a = (y2 - y1) / (x2 - x1);
    long double b = y1 - x1*((y2 - y1) / (x2 - x1));
    long double** pares = new long double* [2*(dimension + 1)];

    //2*(n+1) lo peor es que pase por la
    //diagonal entonces va a haber 2*(n+1) pares;

    long double yEntero;                 //para ver si hay pares repetidos
    int y = 0;                           //para tener el valor entero
    bool* yRepetidos = new bool [dimension + 1];

    for(unsigned int i = 0; i < dimension+1; i++){
        yRepetidos[i] = false;
    }

    for(unsigned int i = 0; i < 2*(dimension+1); i++){
        pares[i] = new long double [2];
    }

    int cantPares = 0;

    //genero el vector pares donde estan los resultados de aplicar
    // y = a*x + b
    // x = (y - b)/a
    // y lo guardo en pares x,y

    for (unsigned int i = 0; i <= dimension; i++){
        long double temp = a*i + b;

        if((temp >= 0) && (temp <= dimension)){
            pares[cantPares][0] = i;
            pares[cantPares][1] = temp;

            y = (int)temp;          //paso el valor a entero
            yEntero = y;            //lo guardo para comparar

            //veo si temp es entero
            //si lo es lo marco para despues no contarlo
            //(por que va a estar repetido)
            if (yEntero == temp){
```

```
            yRepetidos[y] = true;
        }
        cantPares++;
    }
}

for (unsigned int i = 0; (i <= dimension) && (a != 0); i++){
    long double temp = (i - b)/a;

    if(temp >= 0 && temp <= dimension){
        //veo si ese "y" ya estaba antes
        if(!yRepetidos[i]){
            pares[cantPares][0] = temp;
            pares[cantPares][1] = i;
            cantPares++;
        }
    }
}
//ordeno por la primer componente
ordenarParesMenorAMayor(pares, cantPares);

anularRepetidos(pares, cantPares);

//ahora calculo la distancia que recorre la señal por cada pixel y
//genero la fila para la matriz D con estos datos
for(int i = 0; i < cantPares; i++){
    int fil;
    int col;

    //veo si no es un valor duplicado
    if( (pares[i][0] != -1)){
        //si no es duplicado, veo si el que sigue es un duplicado y el
        //siguiente a este es un valor valido(si no me pase de rango)
        if( (pares[i+1][0] == -1) && (i+2 < cantPares) ){
            //menor en x
            col = (int)pares[i][0];

            //menor en y, arreglo para coordenadas de la matriz
            if((int)pares[i][1] < (int)pares[i+2][1]){
                fil = dimension - 1 - (int)pares[i][1];
            }
            else
                fil = dimension - 1 - (int)pares[i+2][1];

            //calculo la distancia recorrida por la señal en cada
            //pixel por donde paso
            long double x = pares[i][0] - pares[i+2][0];
            long double y = pares[i][1] - pares[i+2][1];

            D->asignar(filaALlenar, fil*dimension + col, sqrt(pow(x,2) +
```

```
                                pow(y,2)));

                        i++; //si hay un repetido adelante tengo que avanzar 2
                    }
                    //si no, veo si fallo la primer guarda
                    //si el siguiente no es un duplicado y es una posicion valida
                    else if((pares[i+1][0] != -1) && (i+1 < cantPares))
                    {
                        //menor en x
                        col = (int)pares[i][0];
                        //menor en y, arreglo para coordenadas de la matriz
                        if((int)pares[i][1] < (int)pares[i+1][1]){
                            fil = dimension - 1 - (int)pares[i][1];
                        }
                        else
                            fil = dimension - 1 - (int)pares[i+1][1];

                        //calculo la distancia recorrida por la señal en cada
                        //pixel por donde paso
                        long double x = pares[i][0] - pares[i+1][0];
                        long double y = pares[i][1] - pares[i+1][1];

                        D->asignar(filaALlenar, fil*dimension + col, sqrt(pow(x,2) +
                                    pow(y,2)));
                    }
                }
            }

            for(unsigned int i = 0; i < 2*(dimension+1); i++){
                delete [] pares[i];
            }
            delete [] pares;
            delete [] yRepetidos;
        }
    }

    void Senales :: ordenarParesMenorAMayor(long double** pares, int filas){
        int topeInf = 0;
        long double temp1;  //variable temporal para el primer componente de la
                            //tupla
        long double temp2;  //variable temporal para el segundo componente de la
                            //tupla


        while (topeInf < (filas - 1)){
            int i = topeInf;
            int posMenor = i;
            long double menor = pares[i][0];

            while (i < filas){
```

```
            if(pares[i][0] < menor){
                menor = pares[i][0];
                posMenor = i;
            }
            i++;
        }

        /* hago el swap de los elementos */
        temp1 = pares[topeInf][0];
        temp2 = pares[topeInf][1];

        pares[topeInf][0] = pares[posMenor][0];
        pares[topeInf][1] = pares[posMenor][1];

        pares[posMenor][0] = temp1;
        pares[posMenor][1] = temp2;

        topeInf++;

    }
}

void Senales :: ordenarParesMayorAMenor(long double** pares, int filas){
    int topeInf = 0;
    long double temp1;  //variable temporal para el primer componente de la
                        //tupla
    long double temp2;  //variable temporal para el segundo componente de la
                        //tupla


    while (topeInf < (filas - 1)){
        int i = topeInf;
        int posMayor = i;
        long double mayor = pares[i][0];

        while (i < filas){
            if(pares[i][0] > mayor){
                mayor = pares[i][0];
                posMayor = i;
            }
            i++;
        }

        /* hago el swap de los elementos */
        temp1 = pares[topeInf][0];
        temp2 = pares[topeInf][1];

        pares[topeInf][0] = pares[posMayor][0];
        pares[topeInf][1] = pares[posMayor][1];
```

```
        pares[posMayor][0] = temp1;
        pares[posMayor][1] = temp2;

        topeInf++;

    }
}

void Senales :: anularRepetidos(long double** pares, int cantPares){
    int cant = cantPares;

    for (int i = 0; i < (cant - 1); i++) {
        if(pares[i][0] == pares[i+1][0]){
            pares[i+1][0] = -1;
            i++;            //salteo el anulado
        }
    }
}

void Senales :: graficarMetodo1(ostream & os){
    os << "COPIAR Y PEGAR LO SIGUIENTE EN EL MATLAB" << endl;
    os << "hold on" << endl;
    os << "axis([0 " << dimension << " 0 " << dimension << " ])" << endl;
    for(unsigned int i = 1; i < dimension; i++){
        for(unsigned int j = 1; j <= dimension; j++){
            //tiro las rectas de la pared izquierda
            os << "plot([0 " << j << "], [" << i << " 0], 'k')" << endl;
            os << "plot([0 " << dimension << "], [" << i << " " << j
               << "], 'k')" << endl;
            if(j != dimension)
                os << "plot([0 " << j << "], [" << i << " " << dimension
                   << "], 'k')" << endl;

            //tiro las rectas de la pared derecha
            os << "plot([" << dimension << " " << j << "], [" << i
               << " 0], 'k')" << endl;
            os << "plot([" << dimension << " 0], [" << i << " " << j-1
               << "], 'k')" << endl;
            if(j != dimension)
                os << "plot([" << dimension << " " << j-1 << "], [" << i
                   << " " << dimension << "], 'k')" << endl;
        }
    }
}

void Senales :: graficarMetodo2(ostream & os)
{
    os << "COPIAR Y PEGAR LO SIGUIENTE EN EL MATLAB" << endl;
    os << "hold on" << endl;
    os << "axis([0 " << dimension << " 0 " << dimension << " ])" << endl;
```

```
    int cantDestinos = 3*dimension - 1;
    long double desplazamiento = 0.5*(1/(long double)cantDestinos);

    for(unsigned int i = 0; i < dimension; i++){
        for(unsigned int j = 1; j <= dimension; j++){
            //pared izquierda
            //tiro la señal hacia el piso
            os << "plot([" << 0 << " " << j << "], ["
               << (long double)j/(long double)cantDestinos + i - desplazamiento
               << " " << 0 << "], 'k')" << endl;

            //tiro la señal hacia la pared derecha
            os << "plot([" << 0 << " " << dimension << "], ["
               << (long double)(dimension + j)/(long double)cantDestinos + i -
                   desplazamiento << " " << j << "], 'k')" << endl;

            //tiro la señal hacia el techo
            if (((j != dimension) && (i != (dimension - 1))) ||
               (i == (dimension - 1) && j != dimension  && j != (dimension-1))){
                os << "plot([" << 0 << " " << dimension - j << "], ["
                   << (long double)(2*dimension + j)/(long double)cantDestinos +
                      i - desplazamiento << " " << dimension << "], 'k')"
                   << endl;
            }

            //pared derecha
            //tiro la señal hacia el piso
            os << "plot([" << dimension << " " << dimension - j << "], ["
               << (long double)j/(long double)cantDestinos + i - desplazamiento
               << " " << 0 << "], 'k')" << endl;

            //tiro la señal hacia la pared izquierda
            os << "plot([" << dimension << " " << 0 << "], ["
               << (long double)(dimension + j)/(long double)cantDestinos + i -
                  desplazamiento << " " << j << "], 'k')" << endl;

            //tiro la señal hacia el techo
            if (((j != dimension) && (i != (dimension - 1))) ||
               (i == (dimension - 1) && j != dimension  && j != (dimension-1))){
                os << "plot([" << dimension << " " << j << "], ["
                   << (long double)(2*dimension + j)/(long double)cantDestinos +
                      i - desplazamiento << " " << dimension << "], 'k')"
                   << endl;
            }
        }
    }
}

void Senales :: graficarMetodo3(ostream & os)
```

```
{
    os << "COPIAR Y PEGAR LO SIGUIENTE EN EL MATLAB" << endl;
    os << "hold on" << endl;
    os << "axis([0 " << dimension << " 0 " << dimension << " ])" << endl;

    long double xMedio = (long double)dimension/2;
    long double yMedio = (long double)dimension/2;

    //envio las señales desde la pared izquierda
    for(unsigned int i = 0; i <= dimension*dimension; i++){
        os << "plot([" << 0 << " " << xMedio << "], ["
           << (long double)i/(long double)dimension << " " << yMedio
           << "], 'k')" << endl;
    }

    //envio las señales desde el piso
    for(unsigned int i = 1; i <= dimension*dimension; i++){
        os << "plot([" << (long double)i/(long double)dimension << " " << xMedio
           << "], [" << 0 << " " << yMedio << "], 'k')" << endl;
    }

    for(unsigned int i = 1; i <= dimension*dimension; i++){
        os << "plot([" << xMedio << " " << dimension << "], [" << yMedio << " "
           << (long double)i/(long double)dimension << "], 'k')" << endl;
    }

    for(unsigned int i = 1; i < dimension*dimension; i++){
        os << "plot([" << xMedio << " " << (long double)i/(long double)dimension
           << "], [" << yMedio << " " << dimension << "], 'k')" << endl;
    }
}
```

3. Matriz.h

```
#ifndef _MATRIZ_H
#define _MATRIZ_H

#include <iostream>
#include <string.h>
#include <assert.h>
using namespace std;

class Matriz{
friend ostream& operator<<(ostream&, const Matriz&);
friend ostream& mostrarParaGraficar(ostream&, const Matriz&);
  public:
    Matriz(int = 4, int = 4);
    Matriz(const Matriz& mat);
    int filas() const;
```

```
    int columnas() const;

    long double ver(int fila, int columna) const;
    void asignar(int fila, int columna, long double valor);

    void traspuesta (Matriz& res) const;
    void multiplicar (const Matriz& A, const Matriz& B);
    void triangular (Matriz *b = NULL);
    void resolver (Matriz &X, Matriz &b);
    void cuadradosMinimosLineales (const Matriz& A, const Matriz& b);

    void operator= (const Matriz &m1);
    ~Matriz();
  private:
    long double** m;
    int fil;
    int col;
    /* metodos privados */
    void permutarFilas(int fila1, int fila2, Matriz *b);
    void permutarColumnas(int columna1, int columna2);
    void pivoteoParcial(int, Matriz *b);
    void pivoteoTotal(int, Matriz *b);
    void restarFilas(long double coef, int filaAanular, int filaActual, Matriz *b);
};

#endif /*_MATRIZ_H*/
```

4. Matriz.cpp (nuevos metodos incluidos)

```
void Matriz :: traspuesta (Matriz& res) const
{
    //primero borro la matriz que ya estaba
    if (res.m != NULL){
        for(int i = 0; i < fil; i++)
            delete res.m[i];
        delete res.m;
    }

    res.fil = col;
    res.col = fil;
    res.m = new long double* [fil];

    for(int i = 0; i < res.fil; i++){
        res.m[i] = new long double [col];
        for(int j = 0; j < res.col; j++){
            res.m[i][j] = m[j][i];
        }
    }
}
```

```
void Matriz :: multiplicar (const Matriz& A, const Matriz& B)
{
    assert(A.col == B.fil);

//primero borro la matriz que ya estaba
    if (m != NULL){
        for (int i = 0; i < fil; i++)
            delete m[i];
        delete m;
    }

    fil = A.fil;
    col = B.col;
    m = new long double* [fil];

    for (int i = 0; i < fil; i++){
        m[i] = new long double [col];
        for (int j = 0; j < col; j++){
            long double suma = 0;
    for (int k = 0; k < A.col; k++){
                suma += (A.m[i][k])*(B.m[k][j]);
            }
            m[i][j] = suma;
        }
    }
}

void Matriz :: cuadradosMinimosLineales(const Matriz& A, const Matriz& b)
{
    //En este algoritmo se aprobecha que A(traspuesta)*A es simetrica

    Matriz AtA(A.col, A.col);
    Matriz Atb(A.col, 1);

    for (int i = 0; i < A.fil; i++){
        for (int j = i; j < A.col; j++){
            long double temp = 0;
            for (int c = 0; c < A.fil; c++){
                temp += A.m[c][i]*A.m[c][j];
            }
            AtA.m[i][j] = temp;
            AtA.m[j][i] = temp;
        }
    }

    for (int i = 0; i < Atb.fil; i++){
        long double temp = 0;
        for (int j = 0; j < A.fil; j++){
            temp += A.m[j][i]*b.m[j][0];
        }
```

```
        Atb.m[i][0] = temp;
    }

    AtA.triangular(&Atb);
    AtA.resolver(*this, Atb);
}

void Matriz :: pivoteoTotal(int fila, Matriz *b)
{
    int maxI = fila;
    int maxJ = fila;

    for (int i = fila; i < fil; i++) {
        for (int j = fila; j < col; j++) {
            if(MOD(m[i][j]) > MOD(m[maxI][maxJ])){
                maxI = i;
                maxJ = j;
            }
        }
    }

    permutarFilas(fila, maxI, b);
    permutarColumnas(fila, maxJ);
}
```

5. Main.cpp

```
#include <iostream>
#include <fstream>
#include <time.h>
#include "Matriz.h"
#include "Senales.h"
#include "EasyBMP.h"

using namespace std;

int main(int argc, char* argv[]){
    bool ayuda = false;
    if(argc >= 5){
        unsigned int metodo = atoi(argv[3]);
    long double factorRuido = atof(argv[4]);
        if (factorRuido < 0){
            factorRuido = 0;
        }

        srand((int)time(NULL));

        /*
         *  Levanto el archivo de entrada
         */
```

```
    cout << "Levantando archivo " << argv[1] << " ... ";
BMP imagen;
    imagen.ReadFromFile(argv[1]);

    //Creo la matriz para las velocidades inversas
    Matriz velocidadesInversas(imagen.TellWidth()*imagen.TellHeight(),1);
    //velocidades originales para calcular el Error cuadratico medio
    Matriz velocidadesOrig(imagen.TellWidth()*imagen.TellHeight(),1);

    for (int i = 0; i < imagen.TellHeight(); i++){
        for (int j = 0; j < imagen.TellWidth(); j++){
            long double inverso = (1 /
                                   ((long double)imagen.GetPixel(j, i).Blue
                                   + 1));

            velocidadesInversas.asignar(i*imagen.TellWidth() + j, 0,
                                        inverso);
            //guardo las velocidades orginales para hacer el ECM
            velocidadesOrig.asignar(i*imagen.TellWidth() + j, 0,
                                    (long double)imagen.GetPixel(j, i).Blue +
                                     1);
        }
    }
    cout << "OK!" << endl << endl;

    //inicio: para calcular cuanto tarda el algoritmo
    int inicio = (int)time(NULL);
    Senales D(imagen.TellHeight(), metodo);

    cout << "Usando el metodo " << metodo;
    cout << ", generando " << D.getCantidadSenales();
    cout << " senales..." << endl << endl;
    cout << "Operando ... ";

    D.realizarTomografia(velocidadesInversas, factorRuido);
    int fin = (int)time(NULL);

    cout << "OK!" << endl << endl;
    cout << "El algoritmo termino en " << fin - inicio << " segundos.\n"
         << endl;

    //muestro el Error cuadratico medio para ver la calidad
    //de la imagen reconstruida
    long double ecm = 0;
    for (int i = 0; i < imagen.TellHeight(); i++){
        for (int j = 0; j < imagen.TellWidth(); j++){
            long double orig = velocidadesOrig.ver(i*imagen.TellWidth() +j,
                                                   0);
            long double aprox =
            1/velocidadesInversas.ver(i*imagen.TellWidth() + j, 0);
```

```
        ecm += (orig - aprox)*(orig - aprox);
    }
}
ecm = ecm/(imagen.TellWidth()*imagen.TellWidth());
cout << "Error cuadratico Medio: " << ecm << endl << endl;

//ya reconstrui la imagen, ahora la guardo
cout << "Guardando archivo " << argv[2] << " ... ";

for (int i = 0; i < imagen.TellWidth(); i++){
    for (int j = 0; j < imagen.TellHeight(); j++){
        RGBApixel nuevoPixel;
        long double valorPixel =
        (1/(velocidadesInversas.ver(i*imagen.TellWidth() + j,0))) - 1;

        if (valorPixel < 0)
            valorPixel = 0;

        if (valorPixel > 255)
            valorPixel = 255;
        nuevoPixel.Alpha = 0;
        nuevoPixel.Red = (unsigned char)valorPixel;
        nuevoPixel.Green = (unsigned char)valorPixel;
        nuevoPixel.Blue = (unsigned char)valorPixel;
        imagen.SetPixel(j, i, nuevoPixel);
    }
}
imagen.WriteToFile(argv[2]);
cout << "OK!" << endl << endl;

//veo si se selecciono la opcion para graficar
int parametro = 5;
while (parametro < argc){
    if (strcmp(argv[parametro], "-g") == 0){
        parametro++;
        if (parametro == argc){
            cout << "Parametro -g mal utilizado." << endl << endl;
            ayuda = true;
        }
        else{
            cout << "Preparando grafico ... ";
            ofstream paraMatlab;
            paraMatlab.open(argv[parametro], ios_base::out);
            if (paraMatlab.fail()){
                cout << "FAIL!" << endl << endl;
                ayuda = true;
            }
            else{
                D.prepararParaGraficarMetodo(paraMatlab, metodo);
```

```
                    }
                    paraMatlab.close();
                    cout << "OK!" << endl << endl;
                }
            }

            if (strcmp(argv[parametro], "-h") == 0){
                ayuda = true;
            }

            parametro++;
        }
    }
    else
        ayuda = true;

    //muestro la ayuda
    if (ayuda){
        cout << "\nSimular el proceso de tomografia y reconstruccion\n\n";
        cout << argv[0] << " <Input_file_bmp> <Output_file_bmp> <Metodo>"
             << "Factor_ruido> [Opciones...]" << endl << endl;

        cout << "Input_file_bmp:  Imagen de entrada en formato BMP.\n";
        cout << "Output_file_bmp: Imagen de salida en formato BMP.\n";
        cout << "Metodo:          Metodo de distribucion de señales de 1 a 3\n";
        cout << "Factor_ruido:  Factor de distorcion a aplicar en la imagen.\n";
        cout << "\nOpciones:\n" << endl;
        cout << "-h                    Imprime esta ayuda" << endl;
        cout << "-g <output_file_txt>  Prepara un txt para graficar en matlab\n"
             << endl;
    }
    system("PAUSE");
return 0;
}
```

Referencias

[1] http://www.cplusplus.com/ . Referencia y consulta para C++

[2] http://148.204.224.249/esimetic/seminario2007/mod_01/graficasconmatlab933.pdf Graficos con Matlab.

[3] pcmap.unizar.es/ pilar/latex.pdf . Introduccion a latex.

[4] http://oc.wikipedia.org/wiki/Ajuda:Formulas_TeX_e_LaTeX Formulas Latex.

[5] webs.uvigo.es/mat.avanzadas/PracME_1.pdf . Introducción a MatLab.

[6] ingenierias.uanl.mx/10/pdf/10_Salvador_Acha_et_al_Ajuste_datos_exp.pdf Ajuste de muestras por cuadrados minimos.