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

Reducción de Ruido con la Transformada Discreta del Coseno

Trabajo Práctico 1,
Métodos Numéricos,
Primer Cuatrimestre de 2013

Apellido y Nombre	LU	E-mail
María Candela Capra Coarasa	234/11	canduh_27@hotmail.com
Leandro Lovisolo	645/11	leandro@leandro.me
Lautaro José Petaccio	443/11	lausuper@gmail.com

Resumen:
Se exploran dos métodos basados en la Transformada Discreta del Coseno para reducir distintos tipos de ruidos aperiódicos sobre señales de una y dos dimensiones.

Palabras clave: DCT, ruido, sonido, imágenes, PSNR, frecuencia.

Índice

1. Introducción Teórica

En este trabajo exploramos un conjunto de métodos basados en la Transformada Discreta del Coseno para eliminar ruidos aperiódicos sobre muestras de señales de una y dos dimensiones (ejemplo: audio e imágenes, respectivamente).

La Transformada Discreta del Coseno, de ahora en más DCT, permite expresar un vector en $\mathbb{R}^n$ como combinación lineal de vectores de la forma $\{(cos(0 * t_0), \dots cos(0 * t_{n-1})), \dots (cos(n-1 * t_0), \dots cos(n-1 * t_{n-1}))\}$, donde n es el tamaño de la muestra y $t_i = (i + \frac{1}{2})\frac{\pi}{n}$. Gráficamente, se puede interpretar este cambio de base como la escritura de la muestra como una suma de cosenos de distintas frecuencias, donde las coordenadas del vector transformado son coeficientes que determinan la amplitud de cada coseno, y se presentan ordenados de menor a mayor frecuencia.

En el caso de señales bidimensionales representadas con matrices $\mathbb{R}^{n \times n}$, la matriz transformada obtenida es análoga al caso unidimensional, en la que se pueden obtener los coeficientes ordenados de menor a mayor frecuencia recorriendo las coordenadas diagonalmente de derecha a izquierda y arriba a abajo, partiendo de la coordenada superior izquierda.

2. Desarrollo

Evaluamos dos métodos para reducir los tipos de ruido descritos a continuación. Para cada método realizamos una serie de experimentos con distintos tipos de muestras, y utilizamos la ecuación de *peak signal-noise ratio*, de ahora en más PSNR[1] para medir el nivel de información recuperado.

Los tipos de ruido tratados son los siguientes:

Ruido aditivo: Suma a cada elemento de la muestra una variable aleatoria. En nuestros experimentos, utilizamos variables aleatorias con distribución normal, con media cero y varianza 10.

Observamos que estos parámetros generan una deformación perceptible en todas nuestras muestras, pero no las distorsionan al punto de volverse irreconocibles.

Ruido impulsivo: Reemplaza cada elemento de la muestra por max o min con probabilidad p, donde max y min representan el valor máximo y mínimo que adquieren los elementos de la muestra, y p variable de acuerdo al experimento.

Utilizamos $p = 0{,}01$ en todos nuestros experimentos. Observamos que probabilidades más altas distorsionan demasiado las muestras y dificultan el análisis.

Presentamos los dos métodos evaluados a continuación.

2.1. Atenuar

Dados una constante k y un índice i, multiplicamos por k los coeficientes de la señal transformada con índice mayor o igual a i.

Decidimos usar índices $i = n * 0{,}5$ e $i = n * 0{,}3$ para el caso de muestras unidimensionales y bidimensionales, respectivamente. Para las muestras evaluadas, estos índices maximizan el nivel de PSNR observados luego de aplicar el método. Al transformar estas muestras, observamos que el grueso de la información se encuentra en los coeficientes más bajos, y por lo tanto al atenuar los coeficientes correspondientes a frecuecias más elevadas se produce una mejora en la relación señal ruido dejando intactos los coeficientes con la mayoría de la información.

Similarmente, el k elegido es 0,1. Valores más altos no logran reducir significativamente el ruido, y anular los coeficientes de índice mayor a i implica una importante pérdida de información.

Presentamos a continuación los resultados de distintos experimentos aplicando este método.

2.1.1. Ruido aditivo en audio

Muestra: ramp1234.txt

PSNR inicial: 22.55 dB

PSNR final: 25.03 dB

[1] La fórmula del PSNR se encuentra en el apéndice A.

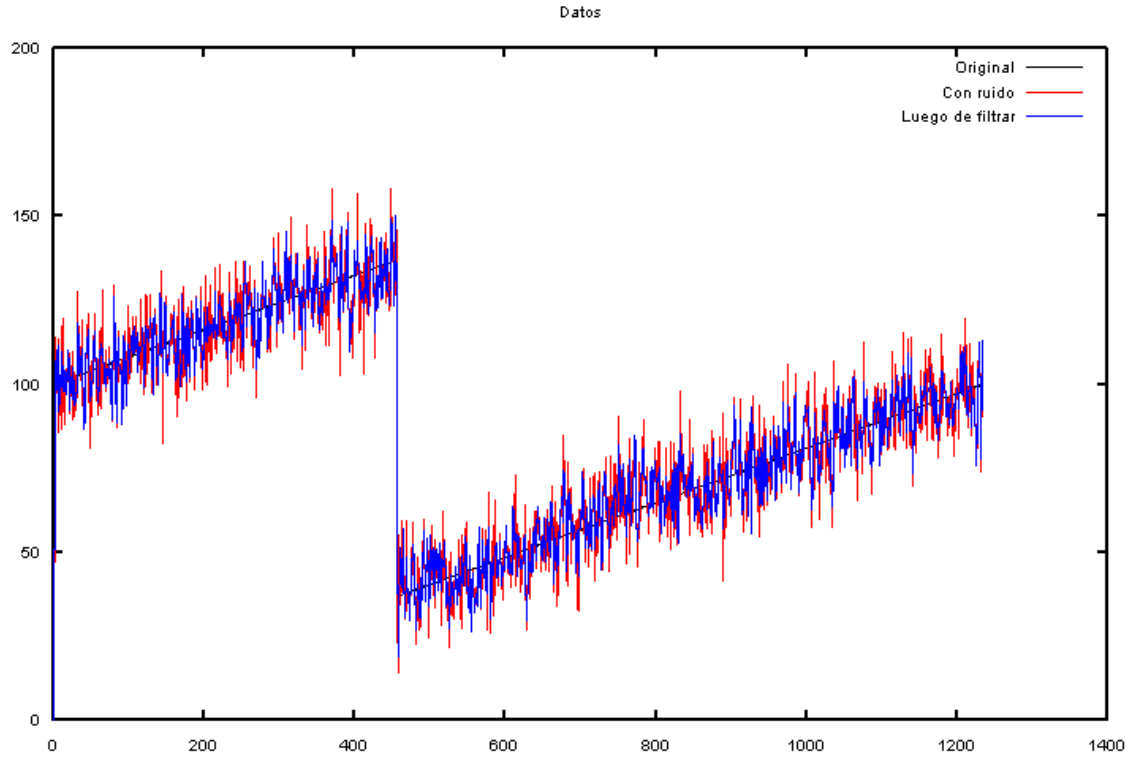

Figura 1: Muestra en base canónica.

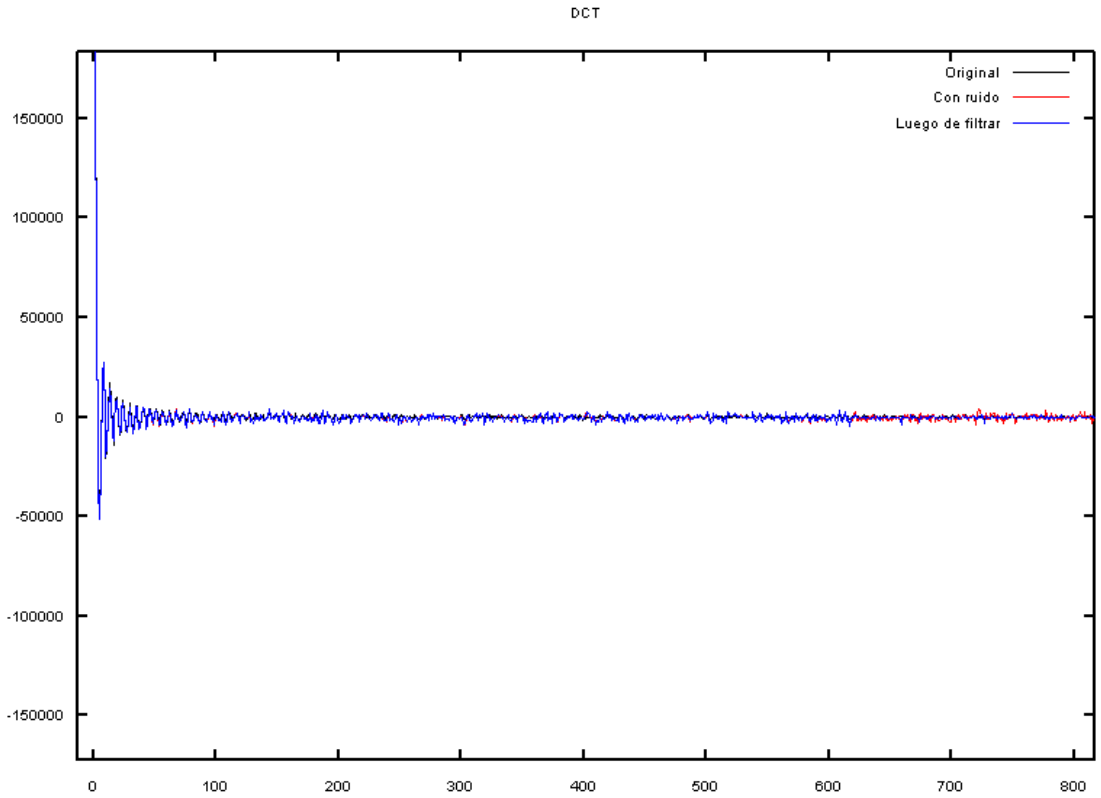

Figura 2: Muestra en base DCT.

Muestra: dopp512.txt

PSNR inicial: 14.02 dB

PSNR final: 16.28 dB

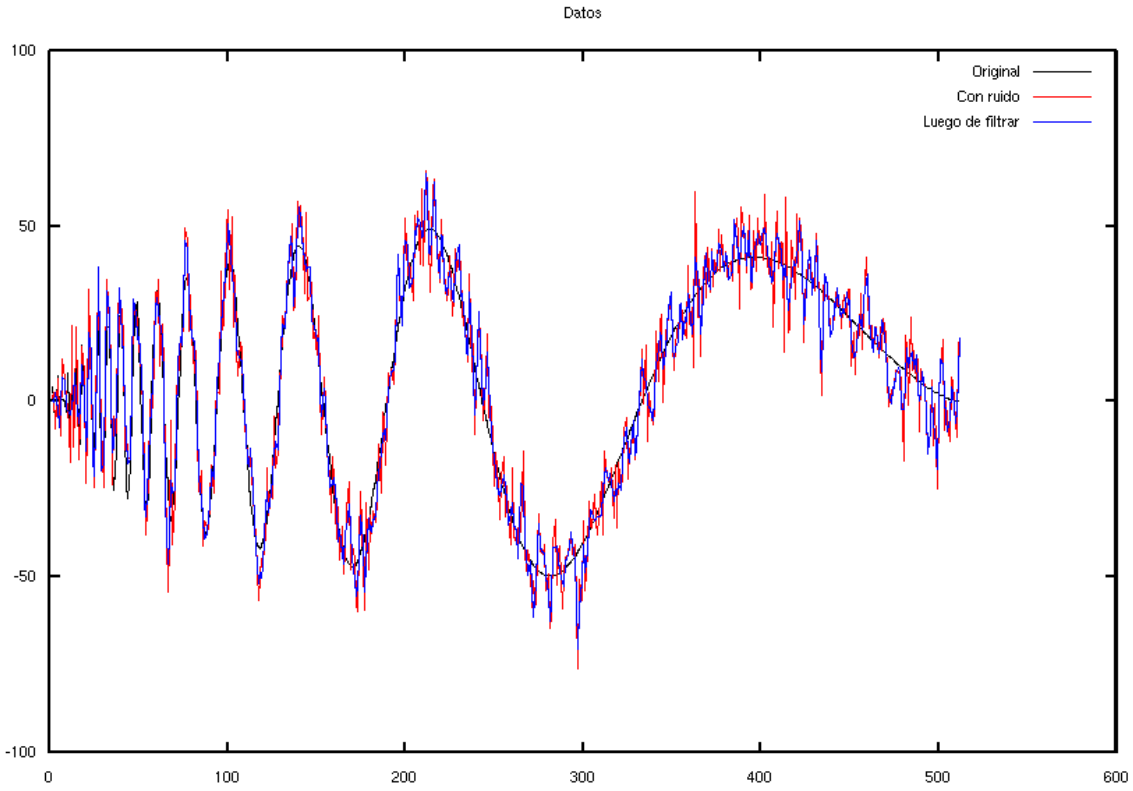

Figura 3: Muestra en base canónica.

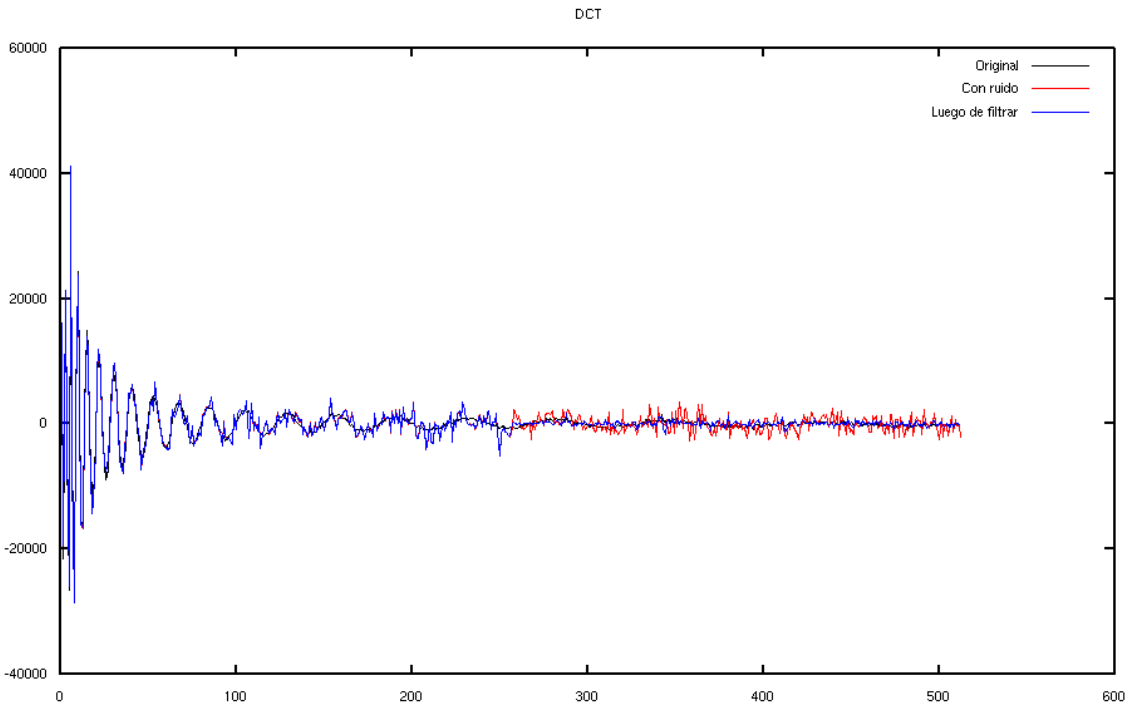

Figura 4: Muestra en base DCT.

2.1.2. Ruido aditivo en imágenes

Muestra: lena.pgm

PSNR inicial: 27.23 dB

PSNR final: 28.62 dB

(a) Antes de atenuar. (b) Después de atenuar.

Figura 5: Ruido aditivo en imágenes.

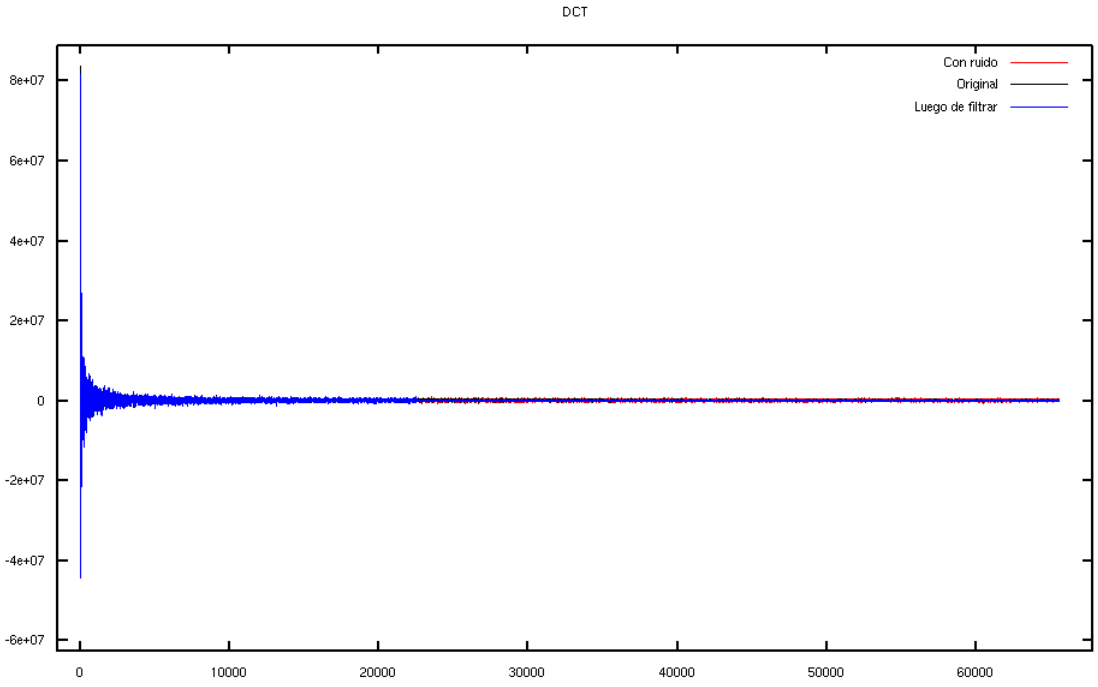

Figura 6: Muestra en base DCT.

2.1.3. Ruido impulsivo en audio

Muestra: dopp512.txt

PSNR inicial: 15.54 dB

PSNR final: 17.55 dB

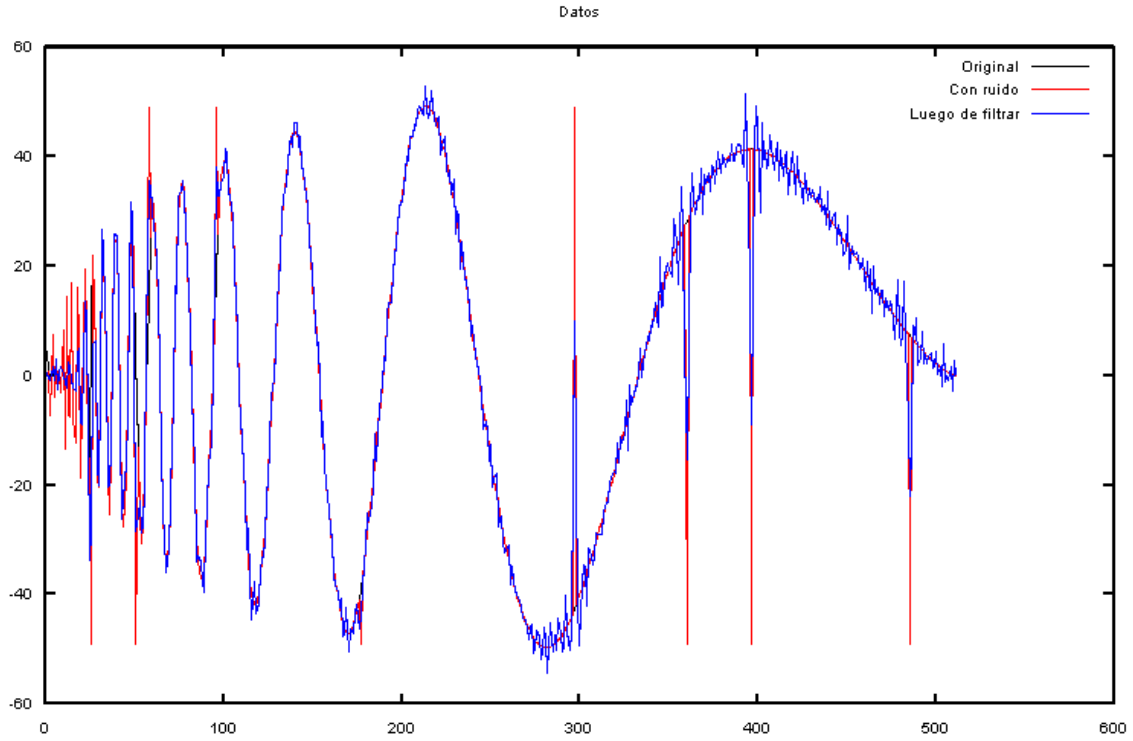

Figura 7: Muestra en base canónica.

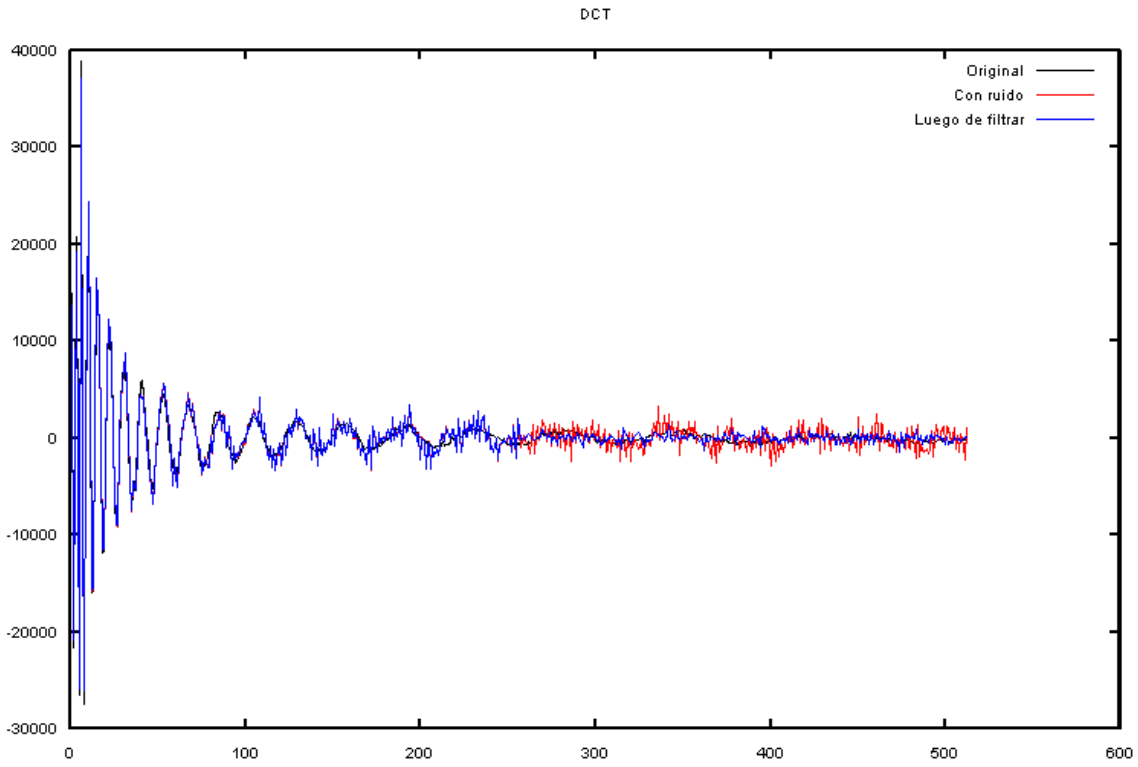

Figura 8: Muestra en base DCT.

2.1.4. Ruido impulsivo en imágenes

Muestra: lena.pgm

PSNR inicial: 21.53 dB

PSNR final: 24.92 dB

(a) Antes de atenuar. (b) Después de atenuar.

Figura 9: Ruido impulsivo en imágenes.

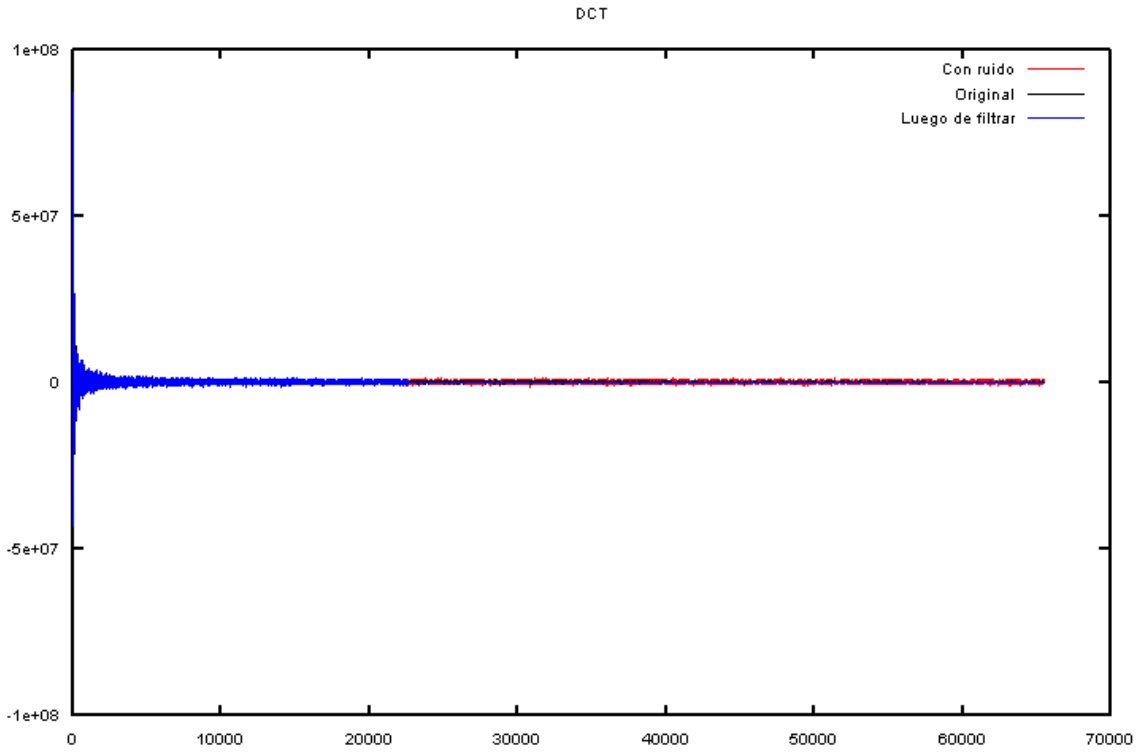

Figura 10: Muestra en base DCT.

2.2. Umbralizar

Dado un umbral β y un índice i, anulamos todos los coeficientes con índice mayor o igual a i y valor absoluto menor a β.

Por los mismos motivos analizados en el método *Atenuar*, usamos índices $i = n * 0{,}5$ e $i = n * 0{,}3$ para el caso de muestras unidimensionales y bidimensionales, respectivamente.

Despues de probar con distintos umbrales, logramos maximizar el PSNR obtenido utilizando $\beta = (max - min) * 0{,}1$ en la mayoría de los experimentos realizados, donde max y min representan el valor máximo y mínimo que adquieren los elementos de la muestra.

2.2.1. Ruido aditivo en audio

Muestra: ramp1234.txt

PSNR inicial: 22.54 dB

PSNR final: 24.60 dB

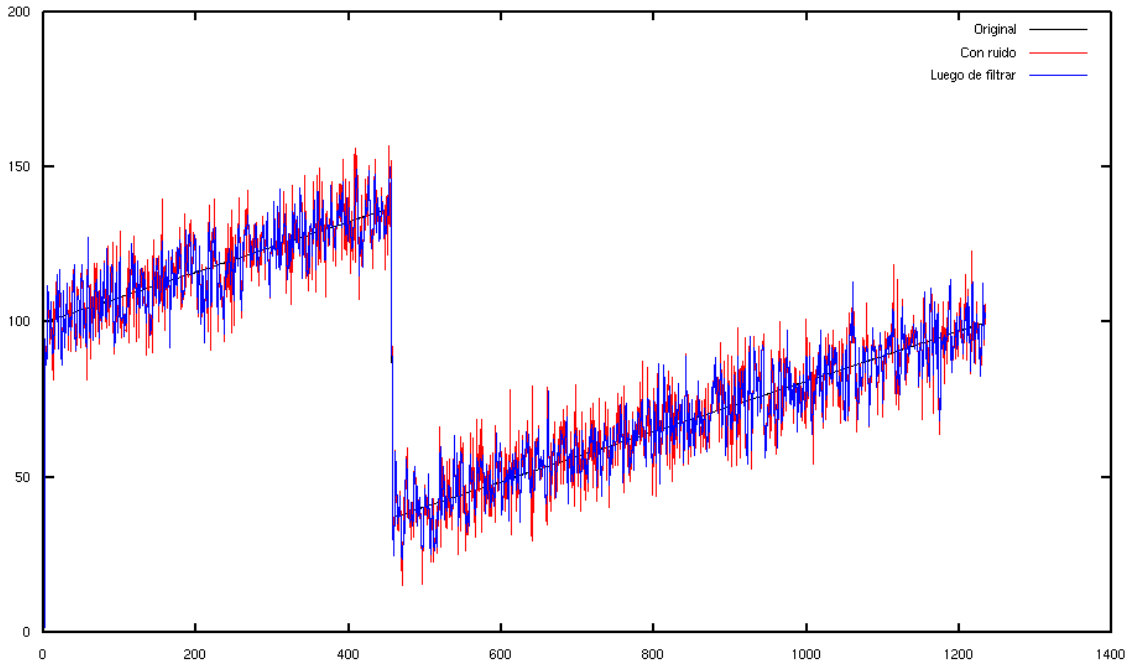

Figura 11: Muestra en base canónica.

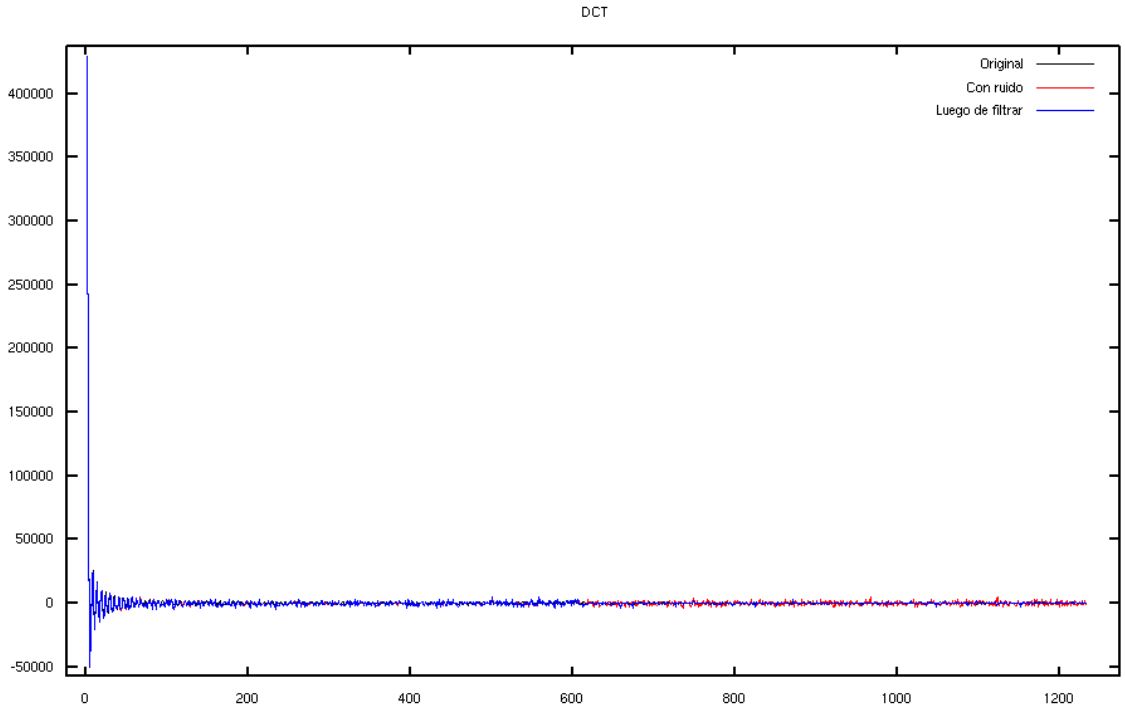

Figura 12: Muestra en base DCT.

Muestra: dopp512.txt

PSNR inicial: 14.31 dB

PSNR final: 16.45 dB

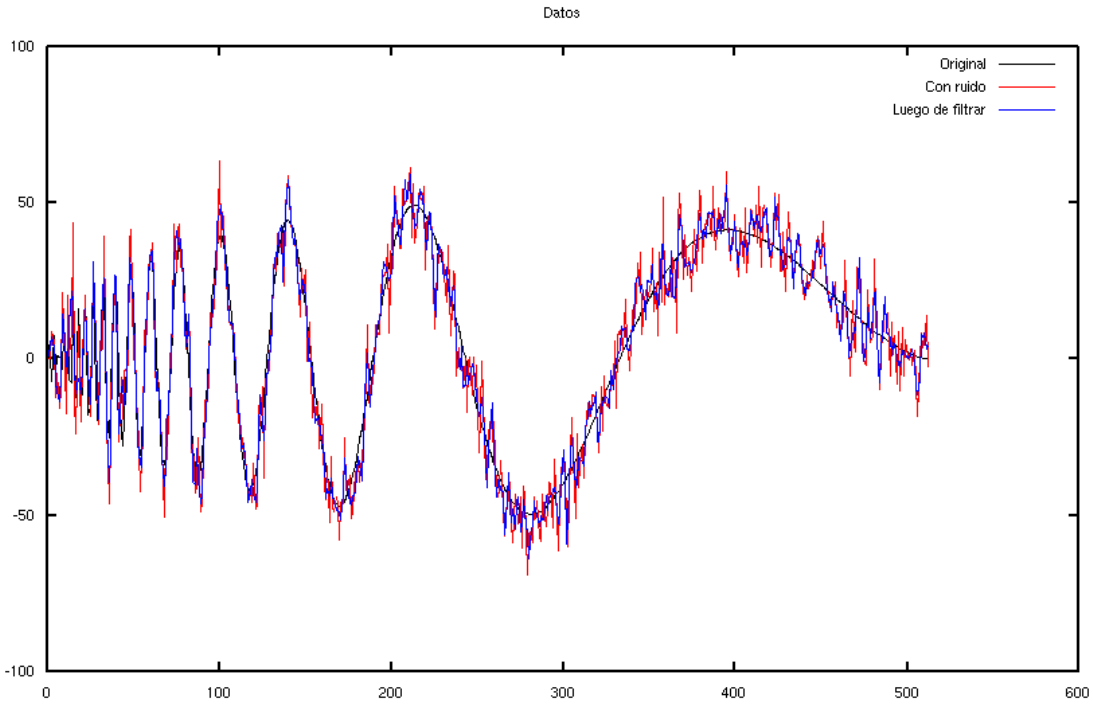

Figura 13: Muestra en base canónica.

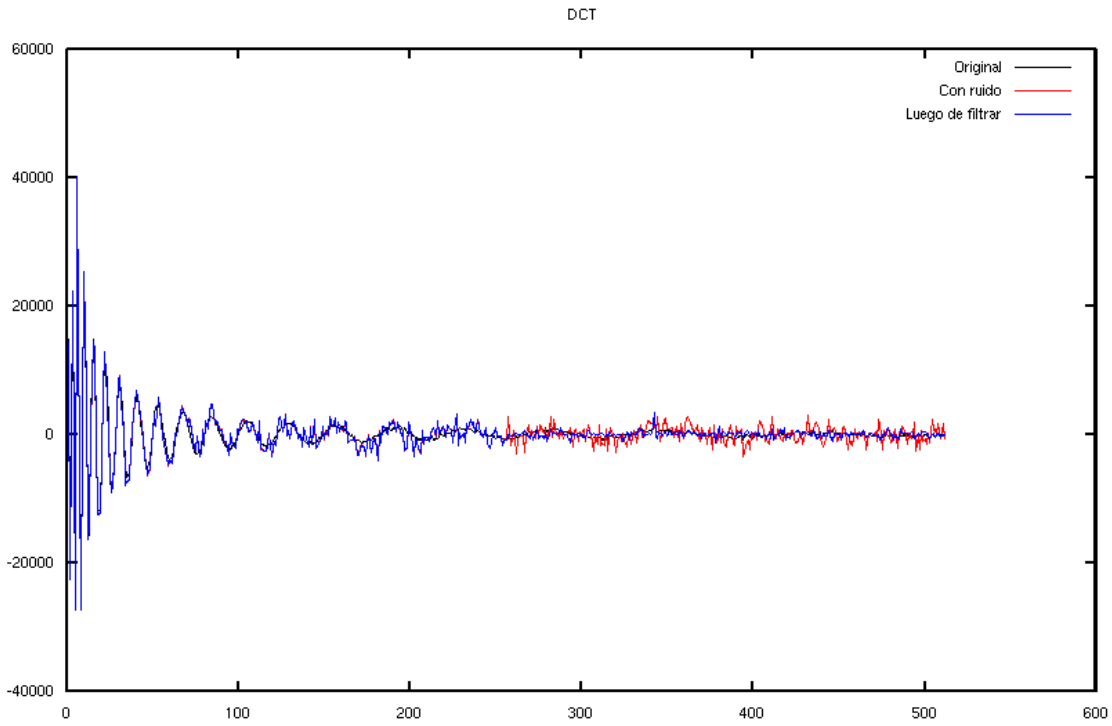

Figura 14: Muestra en base DCT.

2.2.2. Ruido aditivo en imágenes

Muestra: lena.pgm

PSNR inicial: 27.27 dB

PSNR final: 28.08 dB

(a) Antes de umbralizar. (b) Después de umbralizar.

Figura 15: Ruido aditivo en imágenes.

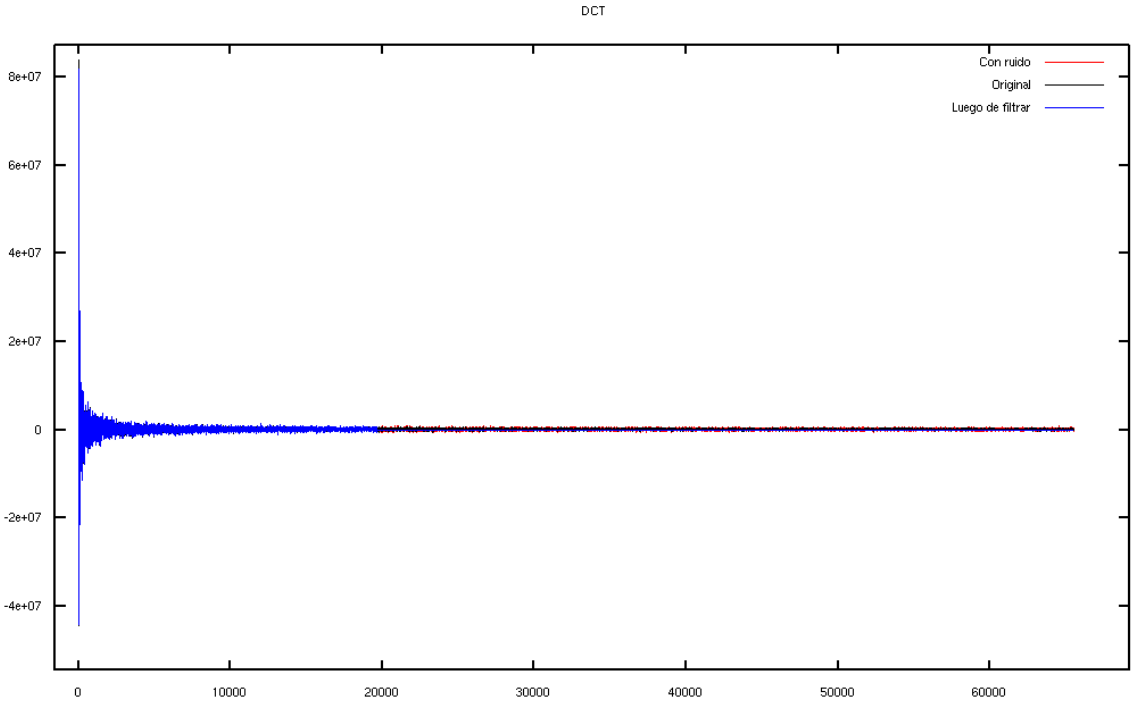

Figura 16: Muestra en base DCT.

2.2.3. Ruido impulsivo en audio

Muestra: dopp512.txt

PSNR inicial: 14.87 dB

PSNR final: 16.96 dB

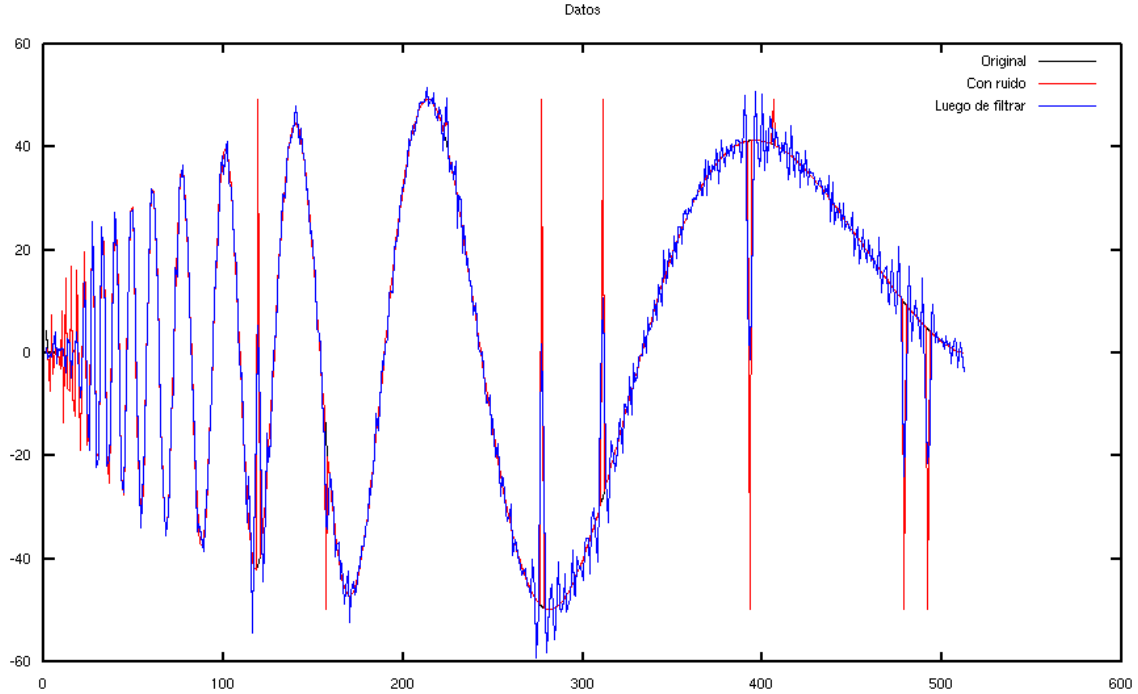

Figura 17: Muestra en base canónica.

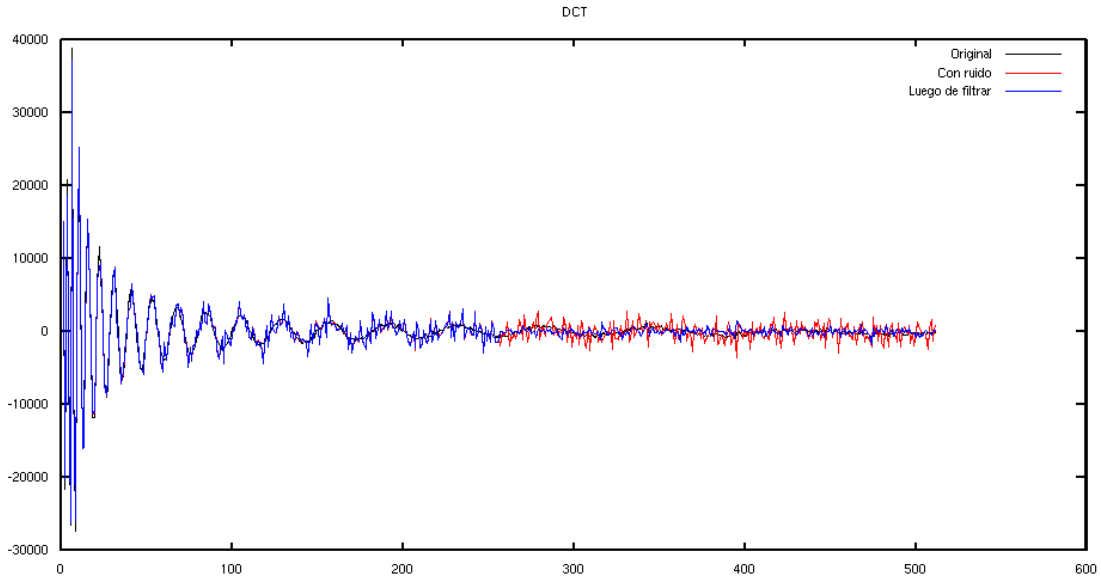

Figura 18: Muestra en base DCT.

2.2.4. Ruido impulsivo en imágenes

Muestra: lena.pgm

PSNR inicial: 21.39 dB

PSNR final: 24.92 dB

(a) Antes de umbralizar. (b) Después de umbralizar.

Figura 19: Ruido impulsivo en imágenes.

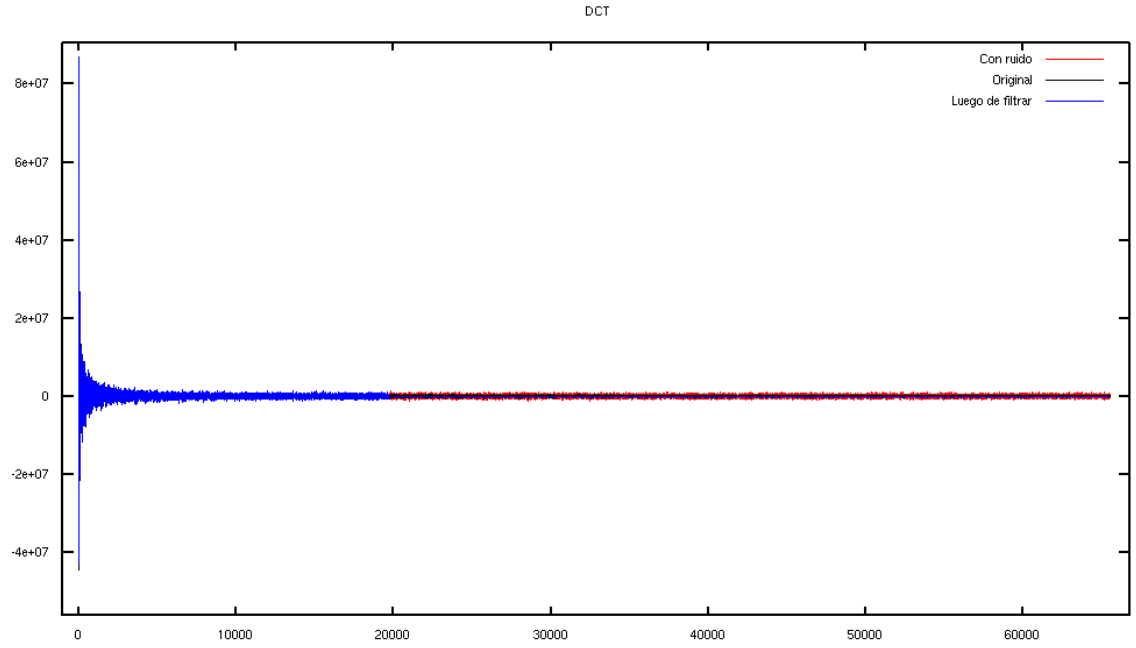

Figura 20: Muestra en base DCT.

3. Conclusiones

En el caso de las imágenes, observamos que al transformarlas al dominio de la DCT, los coeficientes de frecuencias más bajas contienen la mayoría de la información. Observamos además que los ruidos analizados modifican los coeficientes más altos de la imagen, donde se definen los detalles más finos de la misma. Esto nos permite reducir este tipo de ruido fácilmente sin una pérdida signficativa de información reduciendo o eliminando los coeficientes de frecuencias altas.

En contraste, los tipos de ruidos evaluados sobre señales unidimensionales se manifiestan en todos los coeficientes en el espacio de la DCT. No obstante, las muestras evaluadas presentan el grueso de la información en coeficientes medios y bajos, por lo que es posible alcanzar una mejora en PSNR atenuando los coeficientes de frecuencias más elevadas sin pérdidas significativas de información. Esto significa que la muestra resultante conserva el ruido original, pero con menor intensidad en las frecuencias más altas.

En cuanto a los métodos analizados, observamos incrementos de PSNR similares en ambos métodos, pero *Atenuar* resulta superior a *Umbralizar* en casi todos los experimentos, ya que conserva más información al atenuar coeficientes en lugar de eliminarlos completamente.

En conclusión, los métodos basados en la DCT resultan una buena herramienta para eliminar estos tipos de ruido presentes en imágenes. Al aplicarlos en señales unidimensionales también se obtiene una mejora en el PSNR, pero menor que el caso bidimensional ya que no modifican el ruido en sus coeficientes más bajos.

4. Apéndice A: Enunciado del Trabajo Práctico

Introducción

La Transformada Discreta del Coseno (DCT, por sus siglas en inglés) es una herramienta que nos permite representar cualquier señal en el plano de las frecuencias. Dado que es utilizada por el estándar de compresión de imágenes JPEG y formato de video MPEG, se encuentra implementada en más lugares de lo que pensamos: en cada cámara digital o teléfono móvil. La DCT no solo tiene aplicaciones al mundo de la compresión (donde los valores transformados pueden ser codificados de forma eficiente), sino también al procesamiento: el análisis de qué frecuencias están presentes en las señales es esencial en ciertos contextos de aplicación.

La idea intuitiva de esta transformada, en el plano continuo, consiste en representar una función $f : \mathbb{R} \to \mathbb{R}$ en la base de funciones $\mathcal{B} = \{1, \cos(x), \cos(2x), ...\}$. En el plano discreto, la DCT se corresponde a un cambio de base: cada una de las funciones de la base $\mathcal{B}$ se discretiza en ciertos puntos pasando a ser una base de vectores en $\mathbb{R}^n$, (donde n es la dimensión del vector o señal a transformar). Es decir, dado un vector o señal $x \in \mathbb{R}^n$ existe una matriz $M \in \mathbb{R}^{n \times n}$ de cambio de base que define la transformada DCT, donde $y = Mx$ es el vector o señal transformado al espacio de frecuencias por la DCT (ver apéndice 4.1). Esta operación es fácilmente extensible a señales de dos dimensiones (ver apéndice 4.1.1).

Enunciado

El objetivo del trabajo es eliminar ruido sobre una señal ruidosa $x \in \mathbb{R}^n$. Para ello se realiza el siguiente proceso:

1. $y := Mx$ [Transformar usando Ec. (1) de Ap. 4.1]
2. $\tilde{y} := f(y)$ [Modificar]
3. Resolver $M\tilde{x} = \tilde{y}$ [Reconstruir]

Una forma de medir la calidad visual de la señal reconstruida $\tilde{x}$, es a través del PSNR (*Peak Signal-to-Noise Ratio*). EL PSNR es una métrica 'perceptual' (acorde a lo que perciben los humanos) y nos da una forma de medir la calidad de una imagen perturbada, siempre y cuando se cuente con la señal original. Cuanto mayor es el PSNR, mayor es la calidad de la imagen. La unidad de medida es el decibel (db) y se considera que una diferencia de 0.5 db ya es notada por la vista humana. El PSNR se define como:

$$PSNR = 10 \cdot \log_{10}\left(\frac{MAX_x^2}{ECM}\right)$$

donde MAX_x define el rango máximo de la señal (en caso de entradas de 8 bits sin signo, sería 255) y ECM es el *error cuadrático medio*, definido como: $\frac{1}{n}\sum_i (x_i - \tilde{x}_i)^2$, donde n es la cantidad de elementos de la señal, x es la señal original y $\tilde{x}$ es la señal recuperada.

En la implementación realizada deben llevar a cabo los siguientes experimentos:

- Para varias señales con distintos niveles de ruido, se deberán experimentar con al menos 2 estrategias (definiendo f de forma conveniente) para modificar la señal transformada y (paso 2) con el objetivo de que la señal recuperada $\tilde{x}$ contenga menos ruido; se deberán extraer conclusiones en cuanto a la calidad de la señal recuperada, en función de la estrategia utilizada.
- Se deberán repetir los anteriores experimentos también sobre imágenes adaptando el método para aplicar la transformada DCT en dos dimensiones según se explica en la apéndice 4.1.1.
- **(Opcional)** Se deberá analizar la aplicación de la DCT 'por bloques' sobre imágenes. Por ejemplo, si tenemos una imagen de 64×64 píxeles podemos subdividirla en: 4 bloques de 32×32, o 16 bloques de 16×16, o 64 bloques de 8×8, y aplicar la DCT en 2D sobre cada uno de los bloques (considerar un tamaño mínimo de 8×8 para cada bloque).
  Elegir una estrategia utilizada para señales unidimiensionales y sacar conclusiones respondiendo a

los siguientes interrogantes (realizando experimentos que justifiquen la respuesta): ¿Es lo mismo eliminar ruido sobre la imagen entera que de a bloques? ¿Qué forma es más conveniente en cuanto a la calidad visual? ¿Qué forma es más rápida?

Formatos de archivos de entrada

Las señales serán leídas de un archivo de texto en cuya primer línea figuran la cantidad de datos y en la línea siguiente se encuentran los datos en ASCII separados por espacios. Para leer y escribir imágenes sugerimos utilizar el formato *raw* binario `.pgm`[2]. El mismo es muy sencillo de implementar y compatible con muchos gestores de fotos[3] y Matlab.

Fecha de entrega:

- *Formato electrónico:* jueves 16 de mayo de 2013, hasta las 23:59 hs., enviando el trabajo (informe+código) a `metnum.lab@gmail.com`. El subject del email debe comenzar con el texto `[TP2]` seguido de la lista de apellidos de los integrantes del grupo.
- *Formato físico:* viernes 17 de abril de 2013, de 18 a 20hs (en la clase del labo).

4.1. Transformada Coseno Discreta

Para generar la matriz $M \in \mathbb{R}^{n \times n}$ que define la transformada de Coseno Discreta definimos:

- Vector de frecuencias: $g = \begin{pmatrix} 0 \\ 1 \\ \vdots \\ n-1 \end{pmatrix}$
- Vector de muestreo: $s = \frac{\pi}{n} \begin{pmatrix} \frac{1}{2} \\ 1 + \frac{1}{2} \\ \vdots \\ (n-1) + \frac{1}{2} \end{pmatrix}$
- Constante de normalización: $C(k) = \begin{cases} \sqrt{\frac{1}{n}} & k = 1 \\ \sqrt{\frac{2}{n}} & k > 1 \end{cases}$

Siendo $T = \cos(g \cdot s^t)$ la matriz resultante de aplicarle el coseno a cada elemento de la matriz $g \cdot s^t$, finalmente definimos, $\widehat{M}_{i,j} = C(i) \cdot T_{i,j}$

Para obtener una versión entera de la transformación que define la matriz M, la cual será aplicada a señales (o vectores) enteras en el rango $[0, q]$, definimos:

$$M = \left\lfloor \frac{q\widehat{M} + 1}{2} \right\rfloor \tag{1}$$

donde $\lfloor \cdot \rfloor$ indica la parte entera inferior[4]. (Es decir, escalamos los elementos de la matriz M por $q/2$ y luego redondeamos los valores.)

[2] `http://netpbm.sourceforge.net/doc/pgm.html`
[3] XnView `http://www.xnview.com/`
[4] El redondeo de un número m pude definirse como $\lfloor m + 1/2 \rfloor$. Luego, M se define como el redondeo de $\widehat{M} \cdot (q/2)$.

4.1.1. Extensión a 2D

Dada una matriz $B \in \mathbb{R}^{n \times n}$, podemos extender fácilmente la transformada DCT a señales de dos dimensiones. Para ello, aplicamos la transformación por filas y por columnas: $M\,B\,M^t$

5. Apéndice B: Código Fuente

5.1. Metodos.cpp

```
#include <iostream>
#include <random>
#include <cmath>
#include <chrono>
#include "Ecuaciones.h"
#include "Metodos.h"

using namespace std;

void agregarRuidoAditivo(Matriz &m, const double mu, const double sigma) {
        unsigned seed = chrono::system_clock::now().time_since_epoch().count();
        default_random_engine generator(seed);
        normal_distribution<double> distribution(mu,sigma);
        for(int i = 0; i < m.filas(); i++) {
                for(int j = 0; j < m.columnas(); j++) {
                        //cout << "Que larga: " << distribution(generator) << endl;
                        if(m.columnas() == 1) {
                                m.elem(i,j) += distribution(generator);
                        } else {
                                m.elem(i,j) += floor(distribution(generator));
                        }
                }
        }
}

void agregarRuidoImpulsivo(Matriz &m, const double p) {
        int max, min;
        unsigned seed = chrono::system_clock::now().time_since_epoch().count();
        std::default_random_engine generator(seed);
        std::uniform_real_distribution<double> distribution(0.0,1.0);
        if(m.columnas() == 1) {
                max = m.max();
                min = m.min();
        }
        else {
                max = 255;
                min = 0;
        }
        for(int i = 0;i < m.filas(); i++) {
                for(int j = 0;j < m.columnas(); j++) {
                        double randNum = distribution(generator);
                        if(randNum < p) {
                                m.elem(i,j) = max;
                        }
                        else if(randNum >= 1 - p) {
                                m.elem(i,j) = min;
                        }
                }
        }
}

void atenuarIntervaloSonido(Matriz &m, const int coefInicial, const int coefFinal, const double k) {
        for(int i = coefInicial; i <= coefFinal; i++) {
```

```
                m.elem(i,0) *= k;
        }
}

void atenuarImagen(Matriz &m, int coefInicial, int coefFinal, double k) {
        // El par (x0, y0) indica el primer elemento de la diagonal actual
        int x0 = 0;
        int y0 = 0;

        // El par (x, y) indica el coeficiente actual
        int x = x0;
        int y = y0;

        for(int i = 0; i < m.filas() * m.columnas(); i++) {
                // Atenúo el coeficiente actual
                if(coefInicial <= i && i <= coefFinal) m.elem(y, x) = k * m.elem(y, x);

                // Avanzo al siguiente elemento de la diagonal
                x--;
                y++;

                // Verifico si llegué al final de la diagonal actual
                if(x < 0 || y >= m.filas()) {
                        // Avanzo a la siguiente diagonal
                        if(x0 < m.columnas() - 1) x0++;
                        else y0++;

                        // Avanzo al primer elemento de la nueva diagonal
                        x = x0;
                        y = y0;
                }
        }
}

void umbralizarIntervaloSonido(Matriz &m, const int coefInicial, const int coefFinal, const double k
        for(int i = coefInicial; i <= coefFinal; i++) {
                if(abs(m.elem(i,0)) < k) m.elem(i,0) = 0;
        }
}

void umbralizarImagen(Matriz &m, int coefInicial, int coefFinal, double umbral) {
        // El par (x0, y0) indica el primer elemento de la diagonal actual
        int x0 = 0;
        int y0 = 0;

        // El par (x, y) indica el coeficiente actual
        int x = x0;
        int y = y0;

        for(int i = 0; i < m.filas() * m.columnas(); i++) {
                // Umbralizo el coeficiente actual
                if(coefInicial <= i && i <= coefFinal) {
                        if(abs(m.elem(y, x)) < umbral) m.elem(y, x) = 0;
                }

                // Avanzo al siguiente elemento de la diagonal
                x--;
                y++;
```

```
                // Verifico si llegué al final de la diagonal actual
                if(x < 0 || y >= m.filas()) {
                        // Avanzo a la siguiente diagonal
                        if(x0 < m.columnas() - 1) x0++;
                        else y0++;

                        // Avanzo al primer elemento de la nueva diagonal
                        x = x0;
                        y = y0;
                }
        }
}
```

5.2. Ecuaciones.cpp

```
#include <cmath>
#include <iostream> //Para algunos cout, borrar si no estan

#include "Ecuaciones.h"
#include "Metodos.h"

using namespace std;

Matriz* MatrizG(const int n) {
        Matriz* mat = new Matriz(n,1);
        for(int i = 0; i < n; i++) {
                mat->elem(i, 0) = i;
        }
        return mat;
}

Matriz* MatrizSt(const int n) {
        Matriz m(1, n);
        for(int j = 0; j < n; j++) {
                m.elem(0, j) = j + (1 / 2.0); // (n-1) + 1/2, Como empieza en i = 0, ((n+1)-1) + 1/2
        }
        return m * (M_PI / n);
}

double C(const int k, const int n) {
        if(k == 1) return sqrt(1.0 / n);
        return sqrt(2.0 / n);
}

Matriz* MatrizT(const int n) {
        Matriz* st = MatrizSt(n);
        Matriz* g = MatrizG(n);
        Matriz* t = (*g) * (*st);
        for(int j = 0; j < t->columnas(); j++) {
                for(int i = 0; i < t->filas(); i++) {
                        t->elem(i, j) = cos(t->elem(i, j)); // Aplico coseno a todos los elementos
                }
        }
        delete st;
        delete g;
        return t;
```

```
}

Matriz* MatrizMsombrero(const int n) {
        Matriz* t = MatrizT(n);
        for(int j = 0; j < t->columnas(); j++) {
                for(int i = 0; i < t->filas(); i++) {
                        t->elem(i, j) = C(i + 1, n) * t->elem(i, j); // Mutiplico por la función C.
                }
        }
        return t;
}

Matriz* MatrizM(const int n, const int rango) { //Rango es Q = max - min de la señal
        Matriz* Msombrero = MatrizMsombrero(n);
        for(int j = 0; j < Msombrero->columnas(); j++) {
                for(int i = 0; i < Msombrero->filas(); i++) {
                        Msombrero->elem(i, j) = floor((rango * Msombrero->elem(i ,j) + 1) / 2.0);
                }
        }
        return Msombrero;
}

// PSNR para ondas de sonido
double PSNR(Matriz& matOriginal, Matriz& matPerturbada, const int rangoMax) {
        //10 * log10( rango^2 / ECM)
        double ecm = ECM(matOriginal, matPerturbada);
        return 10 * log10( pow(rangoMax, 2) / ecm);
}

//Error cuadrático medio para ondas de sonido
double ECM(Matriz& matOriginal, Matriz& matPerturbada) {
        double acum = 0;
        for(int i = 0; i < matOriginal.filas(); i++) {
                for(int j = 0; j < matOriginal.columnas(); j++) {
                        acum += pow(matOriginal.elem(i, j) - matPerturbada.elem(i, j) , 2);
                }
        }
        return acum / (matOriginal.filas() * matOriginal.columnas());
}

Matriz* aplicarDCT(Matriz& x) {
        Matriz* resultado;
        if(x.columnas() == 1) {
                Matriz* M = MatrizM(x.filas(), x.rango());
                resultado = (*M) * x;
                delete M;
        }
        else {
                //255 para el rango de imagenes
                Matriz* M = MatrizM(x.filas(), 255);
                Matriz *temp = (*M) * x;
                M->transponer();
                resultado = (*temp) * (*M);
                delete temp;
                delete M;
        }
        return resultado;
}
```

```
Matriz* revertirDCT(Matriz& xTransformada, const int rango) {
        Matriz* x = new Matriz(xTransformada);
        if(x->columnas() == 1) {
                Matriz* M = MatrizM(x->filas(), rango);
                //Hago M^(-1) * xTransformada, donde xTransformada es M*x
                Matriz *j = xTransformada.multiplicarPorInversa(*M);
                delete M;
                delete x;
                return j;
        }
        else {
                //Rango 255 para imagenes
                Matriz* M = MatrizM(x->filas(), 255);
                //Hago M^(-1) * cada columna
                for(int i = 0; i < x->columnas(); i++) {
                        Matriz* columna = x->submatriz(0, x->filas()-1 ,i ,i);
                        Matriz* j = columna->multiplicarPorInversa(*M);
                        x->cambiarColumna(*j, i);
                        delete j;
                }
                //Me queda x*Mt , hago (x*Mt)t = M*xt
                x->transponer();
                for(int i = 0; i < x->columnas(); i++) {
                        Matriz* columna = x->submatriz(0, x->filas()-1 ,i ,i);
                        Matriz* j = columna->multiplicarPorInversa(*M);
                        x->cambiarColumna(*j, i);
                        delete j;
                }
                //Me dá xt
                x->transponer();
                delete M;
                return x;
        }
}

Matriz* convertirImagenAVector(Matriz& imagen) {
        Matriz* vector = new Matriz(pow(imagen.filas(),2), 1);
        int x0 = 0;
        int y0 = 0;
        int x = x0;
        int y = y0;
        for(int i = 0;i < vector->filas(); i++) {
                vector->elem(i,0) = imagen.elem(y,x);

                x--;
                y++;

                if(x < 0 || y >= imagen.filas()) {
                        if(x0 < imagen.filas()-1) {
                                x0++;
                        }
                        else {
                                y0++;
                        }
                        x = x0; y = y0;
                }
        }
```

```
        return vector;
}
```

5.3. Matriz.h

```
#ifndef MATRIZ_H_
#define MATRIZ_H_

#include <tuple>

class Matriz {
private:
        double *vectorMatriz;
        int _filas;
        int _columnas;

public:
        Matriz(const int filas, const int columnas);
        Matriz(Matriz& otra);
        ~Matriz();

        static Matriz* identidad(int n);

        int filas() const;
        int columnas() const;
        void transponer();
        double &elem(const int fila, const int columna);
        void cambiarColumna(Matriz& mat, const int columna);

        double max(); // Máximo elemento de la matriz
        double min(); // Mínimo elemento de la matriz
        double rango(); // max() - min()

        Matriz* operator+(Matriz &m);
        Matriz* operator*(Matriz &m);
        Matriz* operator*(double k);

        Matriz* submatriz(const int desdeFil, const int hastaFil, const int desdeCol, const int hast
        Matriz* multiplicarPorInversa(Matriz &M);
        void print();

private:
        void intercambiarFilas(const int i, const int j);
        void intercambiarFilas(const int i, const int j, const int hasta);
        int filaConMayorAbsEnCol(const int col, const int desde);
        std::tuple <Matriz*, Matriz*, Matriz*> factorizacionPLU();
        Matriz* backwardSubstitution(Matriz &b);
        Matriz* forwardSubstitution(Matriz &b);
};

#endif
```

5.4. Matriz.cpp

```
#include <cmath>
```

```
#include <iostream>

#include "Matriz.h"

using namespace std;

Matriz::Matriz(const int filas, const int columnas) {
        _filas = filas;
        _columnas = columnas;
        vectorMatriz = new double[filas * columnas];
}

Matriz::Matriz(Matriz& otra) {
        _filas = otra._filas;
        _columnas = otra._columnas;
        vectorMatriz = new double[_filas * _columnas];
        for(int i = 0; i < _filas * _columnas; i++) vectorMatriz[i] = otra.vectorMatriz[i];
}

Matriz::~Matriz() {
        delete vectorMatriz;
}

Matriz* Matriz::identidad(int n) {
        Matriz *m = new Matriz(n, n);
        for(int i = 0; i < n; i++) {
                for(int j = 0; j < n; j++) {
                        m->elem(i, j) = (i == j ? 1 : 0);
                }
        }
        return m;
}

Matriz* Matriz::submatriz(const int desdeFil, const int hastaFil, const int desdeCol, const int hast
        Matriz *submatriz = new Matriz((hastaFil - desdeFil + 1), (hastaCol - desdeCol + 1));
        for(int i = desdeFil; i <= hastaFil; i++) {
                for(int j = desdeCol; j <= hastaCol; j++) {
                        submatriz->elem(i - desdeFil, j - desdeCol) = elem(i,j);
                }
        }
        return submatriz;
}

void Matriz::cambiarColumna(Matriz& mat, const int columna) {
        for(int i = 0;i < _filas; i++) {
                elem(i,columna) = mat.elem(i,0);
        }
}

int Matriz::filas() const {
        return _filas;
}

int Matriz::columnas() const {
        return _columnas;
}

void Matriz::transponer(){
```

```
        int temp = _filas;
        _filas = _columnas;
        _columnas = temp;
        double *tempMat = new double[_filas * _columnas];
        for(int i = 0; i < _filas ; i++) {
                for(int j = 0; j < _columnas; j++) {
                        tempMat[j * _columnas + i] = elem(i,j);
                }
        }
        delete vectorMatriz;
        vectorMatriz = tempMat;
}

double &Matriz::elem(const int fila, const int columna) {
        return vectorMatriz[fila * _columnas + columna];
}

double Matriz::max() {
        double max = elem(0, 0);
        for(int i = 0; i < _filas; i++) {
                for(int j = 0; j < _columnas; j++) {
                        if(elem(i ,j) > max) {
                                max = elem(i,j);
                        }
                }
        }
        return max;
}

double Matriz::min() {
        double min = elem(0, 0);
        for(int i = 0; i < _filas; i++) {
                for(int j = 0; j < _columnas; j++) {
                        if(elem(i, j) < min) {
                                min = elem(i, j);
                        }
                }
        }
        return min;
}

double Matriz::rango() {
        return max() - min();
}

Matriz* Matriz::operator+(Matriz &m) {
        Matriz* suma = new Matriz(*this);
        for(int i = 0; i < _filas; i++) {
                for(int j = 0; j < _columnas; j++) {
                        suma->elem(i,j) += m.elem(i,j);
                }
        }
        return suma;
}

Matriz* Matriz::operator*(Matriz &m) {
        Matriz* producto = new Matriz(_filas, m._columnas);
        for(int i = 0; i < _filas; i++) {
```

```
                for(int j = 0; j < m._columnas; j++) {
                        producto->elem(i, j) = 0;
                        for(int k = 0; k < _columnas; k++) {
                                producto->elem(i, j) += elem(i, k) * m.elem(k, j);
                        }
                }
        }
        return producto;
}

Matriz* Matriz::multiplicarPorInversa(Matriz &M) {
        tuple<Matriz*, Matriz*, Matriz*> plu = M.factorizacionPLU();
        //Hago Ly = Px
        Matriz* Px = (*get<0>(plu)) * (*this);
        Matriz* y = get<1>(plu)->forwardSubstitution(*Px);
        //Hago Uj = y
        Matriz* j = get<2>(plu)->backwardSubstitution(*y);
        delete y;
        delete Px;
        delete get<0>(plu);
        delete get<1>(plu);
        delete get<2>(plu);
        return j;
}

Matriz* Matriz::operator*(double k) {
        Matriz* producto = new Matriz(*this);
        for(int i = 0; i < _filas; i++) {
                for(int j = 0; j < _columnas; j++) {
                        producto->elem(i, j) = producto->elem(i, j) * k;
                }
        }
        return producto;
}

tuple <Matriz*, Matriz*, Matriz*> Matriz::factorizacionPLU() {
        Matriz *P = identidad(_filas);
        Matriz *L = identidad(_filas);
        Matriz *U = new Matriz(*this);

        for(int j = 0; j < _columnas - 1; j++) {
                // Intercambio la fila con el máximo absoluto por la actual
                // Tener en cuenta que las columnas también determinan la fila a la cual intecambiar
                // ya que vamos moviendonos diagonalmente, (j,j) va a tener siempre el maximo

                int jp = U->filaConMayorAbsEnCol(j, j);
                P->intercambiarFilas(jp, j);
                L->intercambiarFilas(jp, j, j);
                U->intercambiarFilas(jp, j);

                for(int i = j + 1; i < _filas; i++) {
                        L->elem(i, j) = U->elem(i, j) / U->elem(j, j); // Pongo en L Mij
                        U->elem(i, j) = 0; // Pongo en 0 el elemento eliminado
                        for(int x = j + 1; x < _columnas; x++) {
                                U->elem(i, x) = U->elem(i, x) - L->elem(i, j) * U->elem(j, x);
                        }
                }
        }
```

```
        return make_tuple(P, L, U);
}

Matriz* Matriz::backwardSubstitution(Matriz &b) {
        Matriz *x = new Matriz(_columnas, 1);

        // Voy de la fila de abajo para arriba
        for(int i = _filas - 1; i >= 0; i--) {
                // Me armo un acumulador del nuevo valor Xi, voy construyendo el X de abajo hacia ar
                // Utilizo Xi = (Bi - sum(Aij, Xj)/Aii j=i+1 hasta n (cols))
                double valorX = b.elem(i, 0);

                // Hago Xi = (Bi - sum(Aij, Xj)
                // Recorro las columnas de la posición + 1 en que tengo mi incognita
                for(int j = i + 1; j < _columnas; j++) {
                        valorX -= elem(i,j) * x->elem(j,0);
                }

                // Xi/Aii para terminar
                x->elem(i,0) = valorX / elem(i, i);
        }
        return x;
}

Matriz* Matriz::forwardSubstitution(Matriz &b) {
        Matriz *x = new Matriz(this->_columnas,1);

        for(int i = 0; i < _filas; i++) {
                // Me armo un acumulador del nuevo valor Xi, voy construyendo el X de arriba hacia a
                // Utilizo Xi = (Bi - sum(Aij, Xj)/Aii j=i+1 hasta n (cols))
                double valorX = b.elem(i, 0);

                // Hago Xi = (Bi - sum(Aij, Xj)
                for(int j = 0; j < i; j++) {
                        valorX -= elem(i, j) * x->elem(j, 0);
                }

                //Xi/Aii para terminar
                x->elem(i, 0) = valorX / elem(i, i);
        }
        return x;
}

void Matriz::print(){
        for(int i = 0; i < _filas; i++) {
                for(int j = 0; j < _columnas; j++) {
                        cout << elem(i,j) << '\t';
                }
                cout << endl;
        }
}

void Matriz::intercambiarFilas(const int i, const int j) {
        intercambiarFilas(i, j, _columnas);
}

void Matriz::intercambiarFilas(const int i, const int j, const int hasta) {
        if(i == j) return;
```

```
        for(int x = 0; x < hasta; x++) {
                double elemento = elem(i, x);
                elem(i,x) = elem(j,x);
                elem(j,x) = elemento;
        }
}

int Matriz::filaConMayorAbsEnCol(const int col, const int desde) {
        int filaMayor = desde;
        int mayor = elem(desde, col);
        for(int y = desde + 1; y < _filas; y++) {
                if(abs(elem(y, col)) > mayor) {
                        mayor = abs(elem(y, col));
                        filaMayor = y;
                }
        }
        return filaMayor;
}
```