Transformada Rápida de Fourier

Ezequiel Treister

1 de Diciembre, 2000

1 Introducción

Una gran cantidad de problemas computacionales importantes caen en la categoría general de "métodos de transformada de Fourier" o "métodos espectrales". Para algunos de estos problemas, la transformada de Fourier es simplemente una herramienta computacional eficiente para realizar ciertas manipulaciones de datos. En otros casos, nos encontramos con problemas para los cuales la transformada de Fourier es en sí de interes intrínseco. Estos dos tipos de problemas comparten una metodología de resolución comun.

Un proceso físico puede ser descrito en un *espacio temporal* por los valores de una cierta cantidad h que es función del tiempo, i.e. $h(t)$, o bien puede ser descrito en un *espacio de frecuencias* donde el proceso esta descrito por una amplitud H (que en general es un número complejo pues incluye una fase) que es función de la frecuencia f, con $-\infty < f < \infty$. Se puede pasar de una a otra representación a traves de las ecuaciones conocidas como *Transformadas de Fourier* que son:

$$H(f) = \int_{-\infty}^{\infty} h(t) e^{2\pi i f t} dt$$

$$h(t) = \int_{-\infty}^{\infty} H(f) e^{-2\pi i f t} df$$

Para evitar estos factores 2π es que generalmente la frecuencia f es reemplazada utilizándose para describir el sistema a la *frecuencia angular* w, cuya relación con la frecuencia es:

$$w = 2\pi f$$

con lo que las ecuaciones anteriores se transforman en:

$$H(f) = \int_{-\infty}^{\infty} h(t) e^{iwt} dt$$

$$h(t) = \int_{-\infty}^{\infty} H(f) e^{-iwt} dw$$

De estas ecuaciones es directo inferir que una Transformación de Fourier es una transformación lineal. La transformada de la suma de dos funciones corresponde a la suma de las transformadas de cada una de las funciones mientras si la función es multiplicada por una constante entonces la transformada tambien estará multiplicada por la misma constante.

Dadas dos funciones $h(t)$ y $g(t)$ y sus correspondientes transformadas $H(f)$ y $G(f)$ se pueden formar dos combinaciones de especial interes. La *convolución* de las dos funciones, denotada por $g * h$ es definida como:

$$g * h = \int_{-\infty}^{\infty} g(\tau)h(t-\tau)d\tau$$

Cabe notar que $g*h$ es una función en el espacio del tiempo y que $g*h = h*g$. Una propiedad interesante de la convolución es que:

$$g * h \Longleftrightarrow G(f)H(f)$$

es decir, la transformada de fourier de la convolución es simplemente la multiplicación de las transformadas.

La correlación de dos funciones, denotada por $\mathrm{Corr}(g, h)$ esta definida por:

$$Corr(g,h) \equiv \int_{-\infty}^{\infty} g(\tau+t)h(\tau)d\tau$$

Una propiedad interesante de la correlación de dos funciones es:

$$Corr(g,h) \Longleftrightarrow G(f)H^{*}(f)$$

La correlación de una función con sí misma es llamada *autocorrelación*. En este caso, la relación anterior se escribe como:

$$Corr(g,g) \Longleftrightarrow |G(f)|^2$$

La potencia total en una función es la misma tanto si la calculamos en el espacio de los tiempos como en el espacio de las frecuencias. Este resultado es conocido como *Teorema de Parseval*:

$$PTotal \equiv \int_{-\infty}^{\infty} |h(t)|^2 \, dt = \int_{-\infty}^{\infty} |H(f)|^2 \, df$$

2 Transformada de Fourier de un conjunto discreto

Generalmente, la función $h(t)$ es muestreada (es decir, su valor es obtenido) sólo en intervalos equiespaciados en el tiempo. Denotemos por Δ al intervalo de tiempo entre una medición y la siguiente. La secuencia de valores muestreados es entonces $h_n = h(n\Delta)$. El valor recíproco del intervalo de tiempo Δ es conocido como *tasa de muestreo*.

2.1 Teorema del muestreo y Aliasing

Para cualquier intervalo de muestreo Δ existe una frecuencia especial f_c conocida como *frecuencia crítica de Nyquist*, dada por

$$f_c = \frac{1}{2\Delta}$$

Si una onda sinusoidal de esta frecuencia es muestreada en su valor peak positivo, entonces la segunda muestra sera tomada en el valor peak negativo, la siguiente será en el valor positivo y así sucesivamente.

Esta frecuencia crítica es importante por dos razones distintas pero relacionadas. Una es una buena noticia y la otra es una mala noticia. La buena noticia es lo que se conoce como teorema del muestreo: Si una función continua $h(t)$,que es muestreada con intervalos de largo Δ, tiene un ancho de banda limitado a frecuencias menores en magnitud que f_c , es decir si $H(f) = 0$ para todo $|f| > f_c$ entonces la función $h(t)$ está completamente determinada por sus muestreos h_n. De hecho, $h(t)$ está dado explícitamente por la fórmula

$$h(t) = \Delta \sum_{n=-\infty}^{+\infty} h_n \frac{\sin[2\pi f_c(t - n\Delta)]}{\pi(t - n\Delta)}$$

Este teorema es importante debido a que muchas veces la función que se está analizando por razones físicas tiene un ancho de banda limitado con lo que en este caso para conocer toda la información de la función necesitamos utilizar una tasa de muestreo Δ^{-1} igual a dos veces la frecuencia máxima de la función contínua.

Ahora las malas noticias. Si una función no esta limitada en su ancho de banda ocurre que la densidad de potencia espectral en espacio de frecuencias ubicada fuera del rango $-f_c < f < f_c$ se mueve espontaneamente hacia ese rango. Este fenomeno es conocido como aliasing. En otras palabras, cualquier componente cuya frecuencia este fuera del rango $(-f_c\ ,\ f_c)$ será erroneamente transladada a este rango por el sólo hecho de utlizar un muestreo discreto.

2.2 Transformada de Fourier Discreta

Ahora estimaremos la transformada de fourier de una función a partir de un número finito de valores medidos. Supongamos que tenemos N valores consecutivos etiquetados como $h_k \equiv h(t_k)$ donde $t_k \equiv k\Delta$ con $k = 0, 1, 2, ..., N - 1$ de manera que el intervalo de muestreo es Δ. Para hacer las cosas más simples supondremos que N es par. Si la función $h(t)$ es diferente de cero sólo en un intervalo finito de tiempo entonces podemos suponer que todo este intervalo de tiempo esta contenido en los N puntos dados. Alternativamente, si la función $h(t)$ no esta restringida a un intervalo temporal entonces supondremos que el intervalo considerado es representativo de lo que ocurre para cualquier otro tiempo.

Con N numeros de entrada sólo se pueden generar a lo más N números de salida independientes, por lo que en ves de tratar de estimar el valor de la transformada de Fourier $H(f)$ para cualquier valor de f en el rango $-f_c$ a f_c trataremos de estimarlo sólo para valores discretos dados por $f_n \equiv \frac{n}{N\Delta}$ con $n = -\frac{N}{2}, ..., \frac{N}{2}$.

Luego, podemos aproximar la integral que define la transformada de Fourier por una suma discreta dada por:

$$H(f_n) = \int_{-\infty}^{\infty} h(t)e^{2\pi i f_n t}dt \approx \sum_{k=0}^{N-1} h_k e^{2\pi i f_n t_k}\Delta = \Delta \sum_{k=0}^{N-1} h_k e^{2\pi i k n/N}$$

La ultima suma en el lado izquierdo es conocida como *Transformada de Fourier Discreta* de los N puntos h_k , y es escrita usualmente como H_n.

La transformada de Fourier Discreta mapea N números complejos (los h_k) en N números complejos (los H_n). La relación entre la transformada de Fourier Discreta y la contínua puede ser vista como una serie de muestras de la función contínua usando un intervalo Δ, luego esta relación puede ser escrita como:

$$H(f_n) \approx \Delta H_n$$

Hasta ahora, hemos considerado que el índice n varía desde $-N/2$ a $N/2$, sin embargo, puede verse claramente que esta función es periódica en n con período N. Luego, $H_{-n} = H_{N-n}$. Con esta conversión en mente, generalmente se deja que n vaya desde 0 hasta N-1 (un período completo). Cuando esta convención es seguida, se debe recordar que la frecuencia cero corresponde a $n = 0$, las frecuencias positivas $0 < f < f_c$ corresponden a los valores $1 \leq n \leq N/2 - 1$ mientras que las frecuencias negativas $-f_c < f < 0$ corresponden al intervalo $N/2 + 1 \leq n \leq N - 1$. El valor $n = N/2$ corresponde al mismo tiempo a $f = f_c$ y $f = -f_c$.

La fórmula para la inversa de la Transformada de Fourier Discreta que recupera exactamente los valores h_k a partir del conjunto de los H_n está dada por:

$$h_k = \frac{1}{N} \sum_{n=0}^{N-1} H_n e^{-2\pi i k n/N}$$

Cabe destacar que esta expresión sólo difiere de la transformada de Fourier discreta en el signo de la exponencial y en el factor $1/N$ de manera que el mismo programa que calcula la transformada de Fourier Discreta puede calcular la correspondiente inversa con sólo algunas modificaciones.

La forma discreta para el teorema de Parseval es entonces:

$$\sum_{k=0}^{N-1} |h_k|^2 = \frac{1}{N} \sum_{n=0}^{N-1} |H_n|^2$$

3 Transformada Rápida de Fourier (FFT)

Cual es la complejidad involucrada en calcular la transformada de Fourier discreta de N puntos? Por muchos años, hasta mediados de la década del 60, la respuesta estandar fue la siguiente: Definamos W como el número complejo

$$W \equiv e^{2\pi i/N}$$

Luego, la transformación discreta puede escribirse como

$$H_n = \sum_{k=0}^{N-1} W^{nk} h_k$$

En otras palabras, el vector de los h_k es multiplicado por una matriz cuyo elemento (n,k) es la constante W elevada a la potencia nk. Esta multiplicación de matrices produce como resultado un vector cuyas componentes son los H_n. Esta matriz requiere de N^2 multiplicaciones complejas más unas pocas operaciones para generar W, de manera que este algoritmo es $\mathrm{O}(\mathrm{N}^2)$. Sin embargo, veremos a continuación que la transformada de Fourier discreta puede ser calculada con un algoritmo que es $\mathrm{O}(\mathrm{N}\log_2\mathrm{N})$ llamado *Transformada Rápida de Fourier* (FFT).

La diferencia entre $\mathrm{N}\log_2\mathrm{N}$ y N^2 es inmensa. Por ejemplo, si $\mathrm{N}=10^6$ el primer algoritmo toma 2 semanas de tiempo de CPU mientras que el segundo sólo demora 30 segundos de tiempo de CPU en entregar la respuesta en un computador con ciclos de un microsegundo. La existencia de este algoritmo se conoce recién a partir de la decada del 60, a través del trabajo de J.W. Cooley y J.W. Tukey, sin embargo se ha descubierto que implementaciones eficientes de FFT fueron inventadas (y en algunos casos implementadas) por docenas de personas, empezando por Gauss en 1805!

Danielson y Lanczos mostraron que la transforma de Fourier discreta (DFT) de largo N puede ser escrita como la suma de dos DFT cada una de largo N/2. Una de ellas está formada por los puntos con índice par, mientras que la otra está formada por los puntos con índice impar, luego:

$$F_k = \sum_{j=0}^{N-1} e^{2\pi ijk/N} f_j$$

$$= \sum_{j=0}^{N/2-1} e^{2\pi ik(2j)/N} f_{2j} + \sum_{j=0}^{N/2-1} e^{2\pi ik(2j+1)/N} f_{2j+1}$$

$$= \sum_{j=0}^{N/2-1} e^{2\pi ikj/(N/2)} f_{2j} + W^k \sum_{j=0}^{N/2-1} e^{2\pi ikj/(N/2)} f_{2j+1}$$

$$= F_k^e + W^k F_k^o$$

En la última línea, F_k^e denota a la k-ésima componente de la DFT de largo N/2 formada por las componentes pares de los f_j originales, mientras que los F_k^o son las correspondientes transformaciones de largo N/2 formadas por las componentes impares. Es interesante notar que en la última línea, k varía entre 0 y N, no entre 0 y N/2. Sin embargo, las transformaciones F_k^e y F_k^o son periódicas en k con largo N/2 de manera que cada una repite dos ciclos para obtener F_k.

El paso siguiente consiste en utilizar el lema de Danielson-Lanczos recursivamente. De esta manera, el problema de calcular F_k^o se reduce a calcular F_k^e y F_k^o, pero cada uno se puede reducir a resolver el problema a una DFT de largo N/4 para los datos pares y otra para los datos impares, las cuales pueden escribirse como F_k^{ee} , F_k^{eo} , F_k^{oe} y F_k^{oo}.

Pese a que existen formas para tratar cualquier otro caso, es mucho más fácil considerar el caso en el que N es una potencia entera de 2. De hecho, se recomienda ampliamente sólo utilizar FFT con N potencia de dos. Con esta restricción, es evidente que el lema de Danielson-Lanczos puede utilizarse hasta escribir la transformación de largo 1, pero la DFT de largo 1 es sólo la operación identidad que copia un dato de entrada en uno de salida!En otras palabras, para cada combinación de $\log_2 N$ e's y o's existe un número de entrada tal que

$$F_k^{eoeoeoeoeoeee....oee} = f_n$$

para algún n.

El paso siguiente es tratar de descubrir cual es el valor de n correspondiente a una combinación de e's y o's en la ecuación anterior. La respuesta es: Invierta el patrón de e's y o's, haga $e = 0$ y $o = 1$ y tendrá (en binario) el valor de n.

Esta idea de invertir los bits puede ser explotada de una forma muy astuta junto con el lema de Danielson-Lanczos que hace a las FFT muy prácticas: Supongamos que tomamos el vector original de datos f_j y lo reordenamos utilizando el orden de bit inverso. Entonces la aplicación del lema se vuelve extraordinariamente simple. Los puntos obtenidos son las correspondientes transformaciones de un punto. Luego combinamos dos puntos adyacentes y encontramos la transformada de dos puntos, luego combinamos los pares adyacentes y tenemos la transformada de 4 puntos, y asi sucesivamente hasta encontrar la transformada de N puntos. Cada combinación toma O(N) operaciones, pero son evidentemente $\log_2 N$ combinaciones, de manera que todo el algoritmo es O(N$\log_2 N$).

La rutina mostrada a continuación calcula la FFT basándose en una aplicación creada originalmente por N.M Brenner. Las cantidades de entradas son los nn números complejos, el arreglo de datos data[1..2*nn] y **isign** que puede tomar valores ± 1, que corresponden al signo en la exponencial. De esta manera, si **isign**=1 estamos calculando la transformada, y si **isign**=-1 estamos calculando la transformada inversa excepto por el factor 1/N. Cabe notar que los números son complejos, de manera que los valores ingresados en data son los pares tales que data[1] es la parte real de f_0 y data[2] es la parte imaginaria, y así sucesivamente.

Luego, la rutina en C es:

```
#include <math.h>
#define SWAP(a,b) tempr=(a);(a)=(b);(b)=tempr

void four1(float data[], unsigned long nn, int isign)
{
        unsigned long n, mmax, m,j,istep,i;
        double wtemp,wr,wpr,wpi,wi,theta;
        float tempr,tempi;

        n=nn <<1;
        j=1;
        for(i=1;i<n;i+=2) {
                if(j>i) {
                        SWAP(data[j],data[i]);
                        SWAP(data[j+1],data[i+1]);
                }
                m=n >>1;
                while(m>=2 && j>m) {
                        j-=m;
                        m >>=1;
                }
                j+=m;
        }
        mmax=2;
        while(n>mmax) {
                istep=mmax <<1;
                theta=isign*(6.28318530717959/mmax);
                wtemp=sin(0.5*theta);
                wpr=-2.0*wtemp*wtemp;
                wpi=sin(theta);
                wr=1.0;
                wi=0.0;
                for(m=1;m<mmax;m+=2){
                        for(i=m;i<=n;i+=istep) {
                                j=i+mmax;
                                tempr=wr*data[j]-wi*data[j+1];
                                tempi=wr*data[j+1]+wi*data[j];
                                data[j]=data[i]-tempr;
                                data[j+1]=data[i+1]-tempi;
                                data[i]+=tempr;
                                data[i+1]+=tempi;
                        }
```

```
            wr=(wtemp=wr)*wpr-wi*wpi+wr;
            wi=wi*wpr+wtemp*wpi+wi;
        }
        mmax=istep;
    }
}
```

4 FFT de funciones reales

Frecuentemente ocurre que los datos para los que se quiere calcular la DFT son sólo números reales. En ese caso, utilizar la implementación presentada anteriormente resulta ineficiente tanto en tiempo de ejecución como en espacio de almacenamiento requerido.

Existen entonces dos formas de superar este problema. Por un lado se puede utilizar este mismo programa para calcular la DFT de dos funciones diferentes y luego separar las correspondientes transformadas en el resultado. Esto es particularmente útil cuando se están calculando correlaciones o convoluciones, puesto que en estos casos se necesitan las DFT de dos funciones distintas.

El segundo método consiste en almacenar la función real inteligentemente en un arreglo complejo de la mitad del largo, luego se calcula utilizando FFT en este intervalo menor y se extrae el resultado esperado.

4.1 FFT de dos funciones reales simultaneamente

Primero mostraremos como explotar la simetría de la transformación F_n para manejar dos funciones reales simultaneamente: Dado que los datos de entrada f_j son reales, las componentes de la DFT satisfacen que

$$F_{N-n} = (F_n)^*$$

donde el asterisco denota al complejo conjugado. Así mismo, la DFT de un conjunto g_j de imaginarios puros posee la simetría opuesta:

$$G_{N-n} = -(G_n)^*$$

Entonces, podemos calcular la DFT de estas dos funciones reales simultaneamente ingresando estas dos funciones como la parte real y la parte imaginaria del arreglo recibido por el programa anterior. Luego, las transformaciones resultantes son separadas en dos arreglos complejos con la ayuda de estas simetrías.

Para realizar el proceso inverso, es decir calcular la antitransformada de dos funciones complejas simultaneamente de las cuales sabemos que su antitransformada es real debemos recordar que la FFT es lineal y formar la suma de la primera más i veces la segunda, luego se invierte esta función "compleja" utilizando el programa anterior y las partes real e imaginaria corresponden a las dos funciones reales deseadas.

4.2 FFT de una función real

Para implementar el segundo método, que permite calcular la FFT de una función real sin perder eficiencia debemos dividir los datos en dos, de manera de formar dos arreglos reales de la mitad de largo. Luego aplicamos el programa anterior a estos dos reales, pero por supuesto el resultado no será la transformación de los datos originales. Para recuperar la transformada deseada a partir del resultado obtenido debemos realizar lo siguiente:

La forma correcta de separar los datos originales es tomar los f_j con índice par como un arreglo y los que tengan índice impar como el segundo. La belleza de esto es que podemos tomar el arreglo real original y tratarlo como si fuera un arreglo complejo h_j de la mitad de largo. De esta manera, obtenemos que $h_j = f_{2j} + if_{2j+1}$ con $j = 0, 1, 2..., N/2 - 1$, luego le entregamos este arreglo al programa **four1** y este entregará un arreglo complejo $H_n = F_n^e + iF_n^o$ donde

$$F_n^e = \sum_{k=0}^{N/2-1} f_{2k} e^{2\pi ikn/(N/2)}$$

$$F_n^o = \sum_{k=0}^{N/2-1} f_{2k+1} e^{2\pi ikn/(N/2)}$$

De la discusión de la parte anterior sabemos como extraer estas dos transformaciones por separado de H_n luego sólo falta obtener la transformación F_n pedida. Pero para esto vemos que

$$F_n = F_n^e + e^{2\pi in/N} F_n^o$$

Expresado directamente en términos de la transformación H_n el resultado es

$$F_n = \frac{1}{2}(H_n + H_{N/2-n}^*) - \frac{i}{2}(H_n - H_{N/2-n}^*)e^{2\pi in/N}$$

Todo esto es llevado a la practica a continuación en el siguiente programa en C que calcula la DFT de una función real:

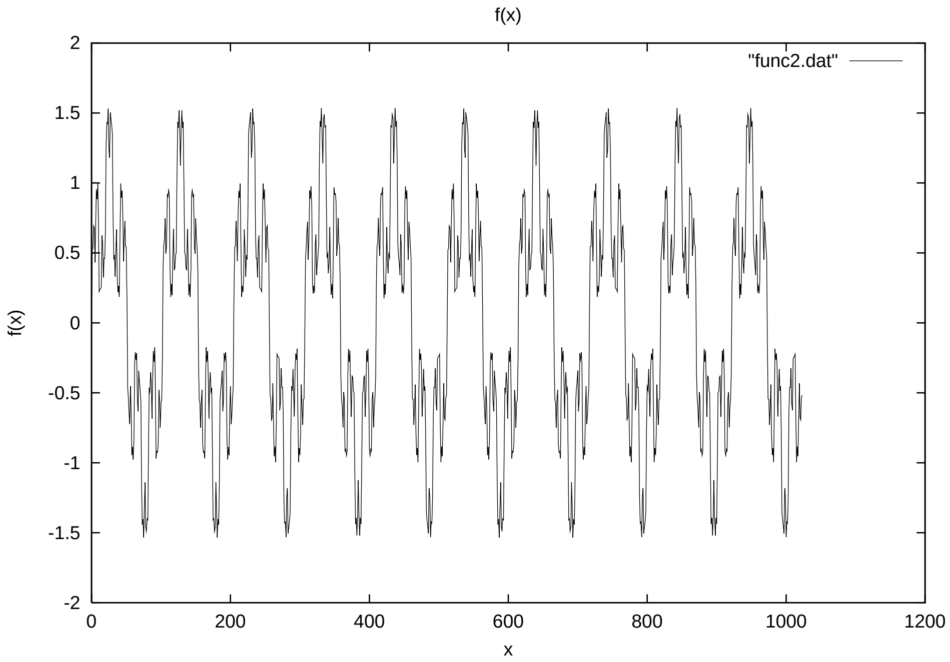
```
void realft(float data[], unsigned long n, int isign)
{
        unsigned long i,i1,i2,i3,i4,np3;
        float c1=0.5,c2,h1r,h1i,h2r,h2i;
        double wr,wi,wpr,wpi,wtemp,theta;

        theta=(double)(M_PI/(double)(n>>1));
        if(isign==1) {
                c2=-0.5;
                four1(data,n>>1,1);
        } else {
```

```
        c2=0.5;
        theta=-theta;
    }
    wtemp=sin(0.5*theta);
    wpr=-2.0*wtemp*wtemp;
    wpi=sin(theta);
    wr=1.0+wpr;
    wi=wpi;
    np3=n+3;
    for(i=2;i<=(n>>2);i++) {
        i4=1+(i3=np3-(i2=1+(i1=i+i-1)));
        h1r=c1*(data[i1]+data[i3]);
        h1i=c1*(data[i2]-data[i4]);
        h2r=-c2*(data[i2]+data[i4]);
        h2i=c2*(data[i1]-data[i3]);
        data[i1]=h1r+wr*h2r-wi*h2i;
        data[i2]=h1i+wr*h2i+wi*h2r;
        data[i3]=h1r-wr*h2r+wi*h2i;
        data[i4]=-h1i+wr*h2i+wi*h2r;
        wr=(wtemp=wr)*wpr-wi*wpi+wr;
        wi=wi*wpr+wtemp*wpi+wi;
    }
    if(isign==1) {
        data[1]=(h1r=data[1])+data[2];
        data[2]=h1r-data[2];
    } else {
        data[1]=c1*((h1r=data[1])+data[2]);
        data[2]=c1*(h1r-data[2]);
        four1(data,n>>1,-1);
    }
}
```

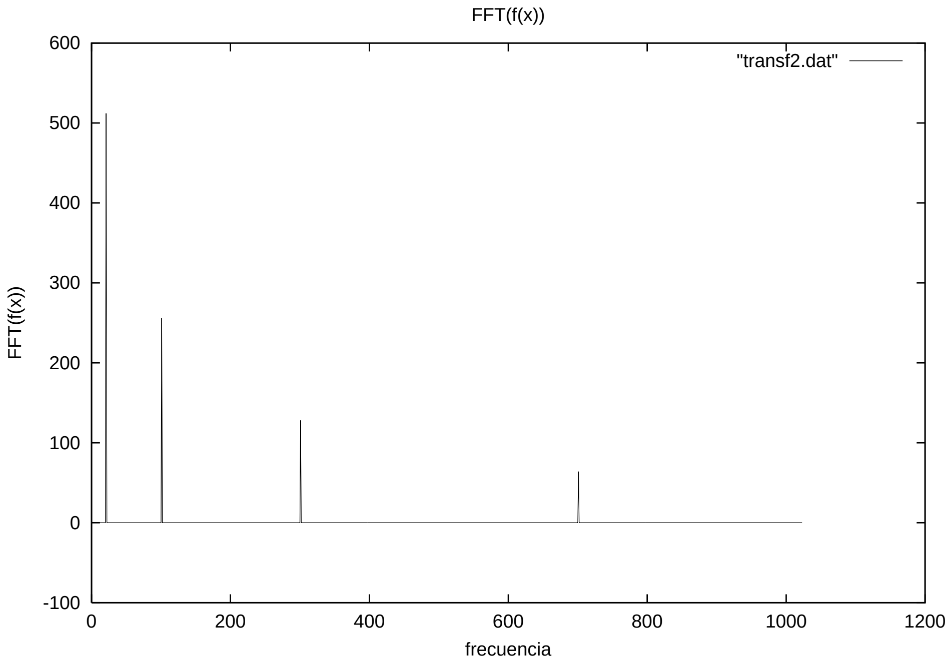
Luego, a modo de ejemplo se introdujo a este programa una muestra de 1024 valores equiespaciados de la función:

$$f_i = \sin(20\pi i/N)+0.5*\sin(100\pi i/N)+0.25*\sin(300\pi i/N)+0.125*\sin(700\pi i/N)$$

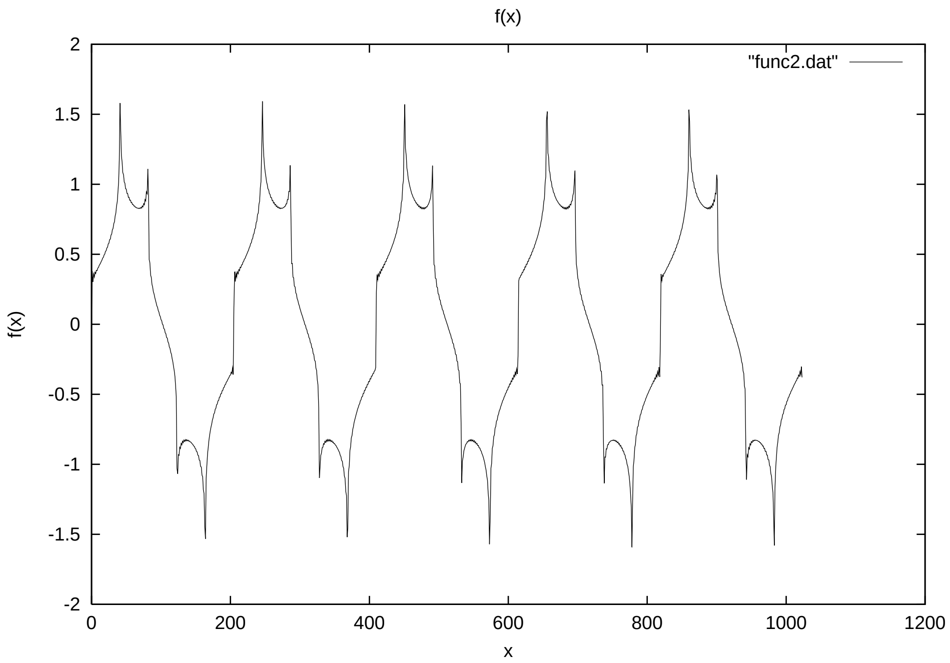
que gráficamente se ve así:



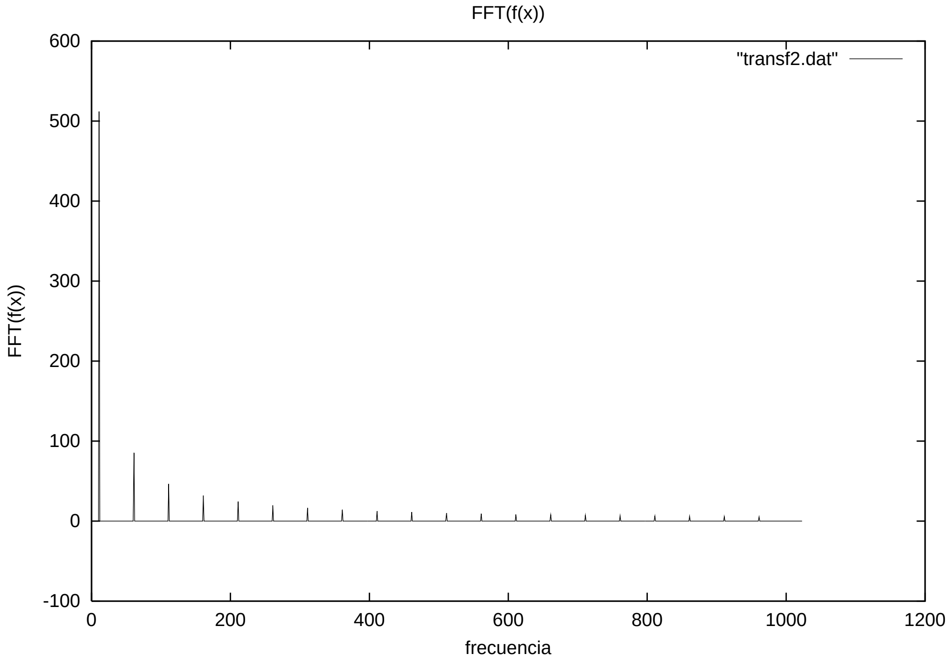
Luego, su transformada es conocida, pues deben aparecer 4 peaks en cada una de las frecuencias utilizadas y la amplitud de cada uno de estos peaks debe ser la mitad de la anterior, y es esto efectivamente lo que se obtuvo.



Otro ejemplo más interesante es el siguiente:



La transformada de Fourier de esta función es la siguiente:



- Puede reconocer cual es la función usada a partir del primer gráfico?
- Y a partir del segundo?

5 FFT en dos o más dimensiones

Dada una función compleja $h(k_1, k_2)$ definida sobre una grilla bidimensional $0 \leq k_1 \leq N_1 - 1$, $0 \leq k_2 \leq N_2 - 1$ podemos definir la DFT en dos dimensiones como una función compleja $H(n_1, n_2)$ definida sobre la misma grilla como:

$$H(n_1, n_2) \equiv \sum_{k_2=0}^{N_2-1} \sum_{k_1=0}^{N_1-1} e^{2\pi i k_2 n_2/N_2} e^{2\pi i k_1 n_1/N_1} h(k_1, k_2)$$

Luego, si factorizamos todos los elementos que tienen el subíndice 2 fuera de la suma sobre k_1 o hacemos lo mismo con los subíndices 1 vemos que la FFT en dos dimensiones puede ser calculada tomando la FFT secuencialmente en cada índice de la función original. Es decir,

H(n_1, n_2)=FFT en el índice 1(FFT en índice 2[$h(k_1, k_2)$])
=FFT en el índice 2(FFT en índice 1[$h(k_1, k_2)$])

Sin embargo, programar FFT en dos dimensiones utilizando este método puede ser un poco más complicado de lo que parece. Esto debido a que la rutina unidimensional requiere que su entrada este en orden consecutivo como un arreglo de una dimensión complejo, por lo que uno pasaría más tiempo copiando elementos de un arreglo a otro , por lo que esta no es una técnica recomendada. En ves de esto, se sugiere utilizar una rutina multidimensional como la mostrada a continuación.

La generalización de las ecuaciones anteriores a más de dos (digamos L) dimensiones es evidente:

$$H(n_1, ..., n_L) \equiv \sum_{k_L=0}^{N_L-1} ... \sum_{k_1=0}^{N_1-1} e^{2\pi i k_L n_L/N_L} \times ... \times e^{2\pi i k_1 n_1/N_1} h(k_1, ..., k_L)$$

Vemos que en la ecuación anterior hay muchos llamados a la FFT unidimensional. Para cada valor de $k_1, k_2, ..., k_{L-1}$ se debe transformar la dimensión L. Luego, para cada valor de $k_1, k_2, ..., k_{L-2}$ y n_L se debe transformar la dimensión L-1 y así sucesivamente. Como se puede deducir, es mucho más efectivo hacer todo este trabajo de una sola vez y para siempre.

La transformación inversa en L dimensiones es exactamente lo que se espera: se cambian todas las i en las exponenciales por $-i$ y luego se multiplica por un factor $1/(N_1 \times ... \times N_L)$.

La rutina **fourn** que se incluye a continuación es la descendiente de la de una dimensión escrita anteriormente. Esta rutina requiere como entrada (i) un escalar que indica el número de dimensiones, (ii) un vector que indica el largo del arreglo en cada dimensión. Es importante notar que se requiere que cada uno de estos valores sea una potencia de 2 y que son todos números complejos en cada dimensión, (iii) el escalar usual ± 1 que indica si se está calculando una transformada o una inversa y (iv) el arreglo de datos.

Respecto a este arreglo de puntos: **fourn** accede a un arreglo unidimensional de números reales, es decir, data[1..(2N$_1$N$_2$...N$_L$)] de largo igual al doble del producto de los largos de las L dimensiones. Se asume que el arreglo representa un arreglo complejo de L dimensiones cuyas componentes individuales están ordenadas de la siguiente manera:

(i) cada valor complejo utiliza las dos ubicaciones secuenciales con la parte real seguida por la parte imaginaria, (ii) el primer subíndice es el que cambia más lento a medida que se recorre el arreglo; el ultimo subíndice es el que cambia más rápido; (iii) los subíndices van desde 1 hasta sus correspondientes valores máximos, no desde 0 hasta $N_1 - 1$, $N_2 - 1$,...,$N_L - 1$ como es usual. La mayoría de las fallas en el funcionamiento del programa ocurren por esta razón.

Entonces, el código correspondiente es el siguiente:

```
#include <math.h>
#define SWAP(a,b) tempr=(a);(a)=(b);(b)=tempr

void fourn(float data[],unsigned long nn[], int ndim, int isign)
{
        int idim;
        unsigned long i1,i2,i3,i2rev,i3rev,ip1,ip2,ip3,ifp1,ifp2;
        unsigned long ibit,k1,k2,n,nprev,nrem,ntot;
```

```
float tempi,tempr;
double theta,wi,wpi,wpr,wr,wtemp;

for(ntot=1,idim=1;idim<=ndim;idim++)
        ntot*=nn[idim];
nprev=1;
for(idim=ndim;idim>=1;idim–) {
        n=nn[idim];
        nrem=ntot/(n*nprev);
        ip1=nprev<<1;
        ip2=ip1*n;
        ip3=ip2*nrem;
        i2rev=1;
        for(i2=1;i2<=ip2;i2+=ip1) {
                if(i2<i2rev) {
                        for(i1=i2;i1<=i2+ip1-2;i1+=2) {
                                for(i3=i1;i3<=ip3;i3+=ip2) {
                                        i3rev=i2rev+i3-i2;
                                        SWAP(data[i3],data[i3rev]);
                                        SWAP(data[i3+1],data[i3rev+1]);
                                }
                        }
                }
                ibit=ip2>>1;
                while(ibit>=ip1 && i2rev>ibit) {
                        i2rev-=ibit;
                        ibit>>=1;
                }
                i2rev+=ibit;
        }
        ifp1=ip1;
        while(ifp1<ip2) {
                ifp2=ifp1<<1;
                theta=isign*2.0*M_PI/(ifp2/ip1);
                wtemp=sin(0.5*theta);
                wpr=-2.0*wtemp*wtemp;
                wpi=sin(theta);
                wr=1.0;
                wi=0.0;
                for(i3=1;i3<=ifp1;i3+=ip1) {
                        for(i1=i3;i1<=i3+ip1-2;i1+=2) {
                                for(i2=i1;i2<=ip3;i2+=ifp2) {
                                        k1=i2;
                                        k2=k1+ifp1;
                            tempr=(float)wr*data[k2]-(float)wi*data[k2+1];
                           tempi=(float)wr*data[k2+1]+(float)wi*data[k2];
```

```
                    data[k2]=data[k1]-tempr;
                    data[k2+1]=data[k1+1]-tempi;
                    data[k1]+=tempr;
                    data[k1+1]+=tempi;
                }
            }
            wr=(wtemp=wr)*wpr-wi*wpi+wr;
            wi=wi*wpr+wtemp*wpi+wi;
        }
        ifp1=ifp2;
    }
    nprev*=n;
  }
}
```

6 Correlaciones y autocorrelaciones usando FFT

Ya se había definido previamente la correlación entre dos funciones contínuas $g(t)$ y $h(t)$ que es escrita como Corr(g, h) y que es función del tiempo t. La correlación será grande para algún valor de t si la primera función es una copia cercana de la segunda pero desfasada en el tiempo en t. Similarmente, la correlación será grande para algún valor negativo de t si la primera función se adelanta a la segunda en t segundos. Si estas funciones son intercambiadas se cumple que

$$Corr(g,h)(t) = Corr(h,g)(-t)$$

La correlación discreta de dos funciones muestreadas g_k y h_k cada una de período N está definida por:

$$Corr(g,h)_i \equiv \sum_{k=0}^{N-1} g_{j+k}h_k$$

El teorema de correlaciones discretas dice que la relación entre la correlación discreta de dos funciones reales g y h y su transformada está dada por:

$$Corr(g,h)_j \Longleftrightarrow G_k H_k^*$$

donde G_k y H_k son las DFT de g_j y h_j respectivamente.

Luego, podemos calcular correlaciones utilizando FFT con el siguiente procedimiento:

- Se obtienen las FFT de cada una de las funciones
- Se multiplica una de ellas por el complejo conjugado de la otra
- Se calcula la FFT inversa del producto

El resultado (llamémoslo r_k) será formalmente un vector complejo de largo N. Sin embargo, ocurre que todas las partes imaginarias son cero si los datos originales eran ambos reales. Las componentes de r_k son los valores de la correlación para los diferentes valores de t, con los valores ordenados de la misma manera que en los casos anteriores.

Luego, el programa que calcula correlaciones es el siguiente:

```
#include <math.h>
#include "nrutil.h"   /* libreria de utilidades de NR */

void correl(float data1[], float data2[], unsigned long n, float ands[])
{
        void realft(float data[], unsigned long n, int isign);
        void twofft(float data1[], float data2[], float fft1[], float fft2[], unsigned
long n);
        /* Se deben incluir las funciones anteriores */
        unsigned long no2,i;
        float dum, *fft;

        fft=vector(1,n<<1);
        twofft(data1,data2,fft,ans,n);
        no2=n>>1;
        for(i=2;i<=n+2;i+=2) {
                ans[i-1]=(fft[i-1]*(dum=ans[i-1])+fft[i]*ans[i])/no2;
                ans[i]=(fft[i]*dum-fft[i-1]*ans[i])/no2;
        }
        ans[2]=ans[n+1];
        realft(ans,n,-1);
        free_vector(fft,1,n<<1);
}
```

La autocorrelación discreta de una función discreta g_j es sólo la correlación de una función con ella misma, luego, para calcular la autocorrelación de una función basta con utilizar el programa anterior pero donde data1[] es igual a data2[] y es la función g_j.

7 Filtros Digitales en el Dominio Temporal

Sunpongamos que se tiene una señal que se quiere filtrar digitalmente. Por ejemplo, puede ocurrir que se quiera aplicar un filtro pasa altos o pasa bajos para eliminar el ruido a altas o bajas frecuencias; o quizás se está interesado sólo en el intervalo de la señal cuya frecuencia está en un rango arbitrario. Finalmente, si las mediciones están contaminadas por la interferencia de 60 Hz causada por las líneas de potencia entonces interesa construir un filtro que elimine los elementos de la señal cuya frecuencia esté en una banda angosta centrada en esta frecuencia.

En esta parte, se hablará especificamente del caso en el que se quiere realizar tal filtro en el dominio del tiempo.

Antes de continuar, se sugiere reconsiderar esta decisión. Es muy conveniente realizar el filtro en el dominio de Fourier. Para aplicar un filtro en el dominio de Fourier basta con realizar una FFT a los datos, multiplicar el resultado por una función de filtro $H(f)$ y luego realizar una FFT inversa para encontrar los datos filtrados en el dominio temporal. A continuación se incluyen algunos consejos que deben seguirse cuando se apliquen filtros en el dominio de Fourier:

- Recuerde que la función de filtro debe estar definida tanto para valores positivos como negativos de la frecuencia y que la magnitud de las frecuencias extremas es siempre la frecuencia de Nyquist $1/(2\Delta\ )$.

- Si los datos medidos son reales y se quiere que la salida filtrada tambien sea real entonces la función de filtro debe cumplir que $H(-f) = H(f)^*$. Esto puede arreglarse fácilmente haciendo que la función H sea real y par en f.

- Un filtro digital es llamado causal o físicamente realizable si la salida para un tiempo particular depende sólo de la entrada para tiempos iguales o menores a este. Opuestamente, se dice que es acausal si la salida depende de tiempos posteriores. El proceso de filtrado en el dominio de Fourier es generalmente acausal debido a que los datos son procesados sin importar su ordenamiento temporal, sin embargo esto no es ninguna dificultad. Al contrario, los filtros acausales muestrán ser más efectivos en el sentido de que tienen una menor dispersión de fases, son menos asimétricos y tienen una mejor respuesta ante impulsos. La razón para utilizar filtros causales es que para muchas aplicaciones el filtro se quiere aplicar en tiempo real, por lo que los filtros acausales no pueden ser implementados. Por esta razón en lo sucesivo se analizarán sólo filtros causales.

7.1 Filtros Lineales

El filtro lineal más general toma una secuencia de entrada de x_k puntos y produce una secuencia de y_n puntos de salida usando la fórmula:

$$y_n = \sum_{k=0}^{M} c_k x_{n-k} + \sum_{j=1}^{N} d_j y_{n-j}$$

Los M+1 coeficientes c_k y los N coeficientes d_j son fijos y definen la respuesta del filtro. El filtro produce nuevos valores de salida a partir de el valor actual y de los M valores previos de entrada y de los N valores previos de la salida. Si N=0 entonces la segunda suma no aparece y el filtro es llamado no recursivo o filtro de respuesta finita a impulsos (en inglés FIR). Si $N \neq 0$ entonces el filtro es llamado recursivo o filtro de respuesta infinita a impulsos (en inglés IIR).

La relación entre los c_k , los d_j y la función $H(f)$ está dada por:

$$H(f) = \frac{\sum_{k=0}^{M} c_k e^{-2\pi i k(f\Delta)}}{1 - \sum_{j=1}^{N} d_j e^{-2\pi i j(f\Delta)}}$$

Esta ecuación nos dice como determinar $H(f)$ a partir de los c y los d pero usualmente uno necesita el proceso inverso, es decir, dado el filtro que se quiere construir calcular los c y los d necesarios.

Como en cualquier "problema inverso" este problema no tiene solución única pues se trata de un problema muy complicado y que sólo tiene solución clara en algunos casos, para casos más generales la solución se debe generar "a mano".

Los casos más interesantes ocurren con los filtros recursivos, por lo que este caso será tratado a continuación.

7.2 Filtros IIR (recursivos)

Los filtros recursivos generalmente tienen una performance que es superior a los no recursivos que tienen el mismo número total de coeficientes (o el mismo número de operaciones flotantes por punto de entrada). La razón es clara por inspección de la ecuación anterior: Un filtro no recursivo tiene una respuesta en frecuencia que es polinomial en la variable $1/z$ donde

$$z \equiv e^{2\pi i(f\Delta)}$$

Por contraste, un filtro recursivo tiene una respuesta en frecuencia que es una función racional en $1/z$.

Veamos a continuación un ejemplo.

Supongamos que se quiere designar un filtro de bandas, cuya frecuencia inferior de corte corresponde a un valor $w = a$ mientras que la frecuencia superior de corte corresponde a otro valor $w = b$ con a y b positivos. Una función racional simple que cumple con esto es

$$|H(f)|^2 = \left(\frac{w^2}{w^2 + a^2}\right)\left(\frac{b^2}{w^2 + b^2}\right)$$

Los polos de esta ecuación están obviamente en $w = \pm ia$ y $w = \pm ib$. Luego una función razonable es:

$$H(f) = \left(\frac{w}{w - ia}\right)\left(\frac{ib}{w - ib}\right) = \frac{\left(\frac{1-z}{1+z}\right)b}{\left[\frac{(1-z)}{(1+z)} - a\right]\left[\frac{(1-z)}{(1+z)} - b\right]}$$

donde la i en el numerador del segundo factor es para que el resultado sea un valor real. Reescribiendo esta ecuación obtenemos que:

$$H(f) = \frac{-\frac{b}{(1+a)(1+b)} + \frac{b}{(1+a)(1+b)} z^{-2}}{1 - \frac{(1+a)(1-b)+(1-a)(1+b)}{(1+a)(1+b)} z^{-1} + \frac{(1-a)(1-b)}{(1+a)(1+b)} z^{-2}}$$

de donde se obtienen los coeficientes:

$$c_0 = -\frac{b}{(1+a)(1+b)}$$

$$c_1 = 0$$

$$c_2 = \frac{b}{(1+a)(1+b)}$$

$$d_1 = \frac{(1+a)(1-b)+(1-a)(1+b)}{(1+a)(1+b)}$$

$$d_2 = -\frac{(1-a)(1-b)}{(1+a)(1+b)}$$

que definen completamente a este filtro.

Supongamos que ahora se quiere crear un filtro notch, es decir, un filtro que sólo elimina una banda de frecuencias muy angosta en torno a una frecuencia $w = w_0$ donde w_0 es un número positivo,

$$H(f) = (\frac{w - w_0}{w - w_0 - i\varepsilon w_0})(\frac{w + w_0}{w + w_0 - i\varepsilon w_0})$$

$$= \frac{w^2 - w_0^2}{(w - i\varepsilon w_0)^2 - w_0^2}$$

donde ε es un número positivo chico, que representa el ancho deseado para la banda. Realizando algunas manipulaciones aritméticas y substituyendo z por w se encuentran los coeficientes para el filtro:

$$c_0 = \frac{1 + w_0^2}{(1+\varepsilon w_0)^2 + w_0^2}$$

$$c_1 = -2\frac{1 - w_0^2}{(1+\varepsilon w_0)^2 + w_0^2}$$

$$c_2 = \frac{1 + w_0^2}{(1+\varepsilon w_0)^2 + w_0^2}$$

$$d_1 = 2\frac{1 - \varepsilon^2 w_0^2 - w_0^2}{(1+\varepsilon w_0)^2 + w_0^2}$$

$$d_2 = -\frac{(1-\varepsilon w_0)^2 + w_0^2}{(1+\varepsilon w_0)^2 + w_0^2}$$