Algoritmos y programación II (95.12)

Integrantes:

Freites, Lucas # 96138

Mancini, Lucas # 98091

Rizzo, Gonzalo Gabriel # 96772

|  |  |
| --- | --- |
| Entrega/Fecha | 2/12/16 |
| Nota |  |

**Introducción**

En el presente trabajo se espera ejercitar técnicas de diseño, análisis e implementación de algoritmos recursivos

**Desarrollo**

1. **Invocación**

La interacción con el programa se realiza mediante línea de comandos. A continuación se especifica las opciones disponibles a incluir durante su invocación en la línea de comandos y las respuestas del programa ante los mismos. En el caso que las opciones no sean especificadas, o a continuación de las mismas se reciba “-”, las opciones (en el caso que se indique) toman su valor por defecto:

* Entrada:
  + Flag: “-i”, o “--input”
  + Por defecto se tomará como entrada el flujo estándar *cin.* El ingreso de datos se detiene al encontrar el EOF (*“end of file”.* Linux: Ctrl+D. Win: Ctrl+Z)
  + A continuación de esta bandera, se especifica el nombre del archivo donde se hará la recepción de datos que serán procesados por el programa.
  + Los requisitos que debe cumplir el archivo de entrada son:
    - Debe ser un archivo plano en formato *txt*.
    - Los números complejos deben estar en el formato de par ordenado, o en el caso de solo poseer la parte real, un número. Los mismos deben estar separados unos de otros por espacios en blanco (tabulación, espacio o salto de línea).
    - No se permiten otros caracteres distintos a los antes mencionados.
    - No se permiten expresiones matemáticas.
* Salida:
  + Flag: “-o”, o ”--output”
  + Por defecto se tomará como entrada el flujo estándar *cout.*
  + A continuación de esta bandera, se especifica el nombre del archivo donde se despliega el resultado final del programa.
  + En el caso de que el archivo exista previamente, se incluyen los nuevos datos a continuación.
* Selección de algoritmo de entrada:
  + Flag: “-f”, o “--forward-op”
  + Permite especificar el algoritmo de cómputo para transformar el stream de entrada.
  + Son distintos algoritmos para realizar el mismo cálculo, todos representan la transformada de Fourier.
  + Sus opciones son: “dft” o “fft”.
  + Por defecto se asigna el “dft”.
* Selección de algoritmo de entrada:
  + Flag: “-r”, o “--reverse-op”
  + Permite especificar el algoritmo de cómputo para transformar y obtener el stream de salida.
  + Son distintos algoritmos para realizar el mismo cálculo, todos representan la inversa de la transformada de Fourier.
  + Sus opciones son “idft” o “ifft”
  + Por defecto se asigna el idft.
* Tamaño de bloques:
  + Flag: “-b”, o “--block-shift”
  + Por defecto, block\_size = 1
  + Los requisitos que debe cumplir el valor son:
    - Debe ser mayor o igual a 0.
    - No puede se permiten expresiones matemáticas.
  + Controla el tamaño de bloque, es decir, el tamaño del vector de elementos del stream de entrada que se agrupan y transforman, siendo block\_size = .
* Coeficiente de amplificación:
  + Flag: “-t”, o “--taps”.
  + Por defecto se rellena un vector “Taps” (con el tamaño “block\_size”) con el valor de 1. En caso que en el archivo una cantidad de coeficientes menor a “block\_shift”, se muestra un mensaje de error y procede a la terminación del programa.
  + Va a determinar la forma de los coeficientes complejos que deben usarse para calcular la multiplicación en el dominio transformado.
* Ayuda:
  + Flag: “-h”.
  + Es una opción que indica cómo es la correcta invocación del programa, especificando el nombre del mismo.

**B. Procesamiento**

La función principal del programa, *main*, consta de 4 procesos fundamentales:

* *cmdline cmdl(options);*

Se crea un objeto del tipo *cmdline* para manejar los argumentos obtenidos en la invocación del programa. El mismo se inicializa con la tabla *options* del tipo *options\_t* declarada globalmente, la cuál contiene un arreglo con opciones estandarizadas.

* *cmdl.parse(argc,argv);*

Se instancia el objeto *cmdl* para que se parseen los respectivos argumentos con los que se invocó el programa.

* *build\_fourier\_transformation\_type(FourierTransform\_t type);*

Se encarga de crear (con memoria dinámica, utilizando “new”) el tipo de clase virtual correspondiente según el parámetro obtenido del tipo enumerativo “FourierTransform\_t”(el cual a su vez fue parseado de un argumento de línea de órdenes) y lo devuelve.

* *destroy\_fourier\_transformation\_type(FourierTransform\_t type);*

Destruye lo pedido con “new” anteriormente, se llama a su contraparte “destroy”.

* *process\_signal((FourierTransform \*, FourierTransform \*);*

Dentro de la función “process\_signal” se realizan 3 procesos básicos: Una primera transformada a los elementos ingresados, una amplificación/ecualización de los transformados, y una transformación inversa que dará como resultado lo valores de salida.

**Documentación**

Para el trabajo se utilizaron los los scripts provistos por la cátedra de:

***cmdline.h cmdline.cc complejo.h complejo.cc***

A continuación se presentan en forma impresa todos los scripts codificados para el trabajo:

***main.cpp***

#include <iostream>

#include <fstream>

#include <iomanip>

#include <sstream>

#include <cstdlib>

#include <cmath>

//Librerias nuestras

#include "cmdline.h"

#include "vector.h"

#include "complejo.h"

#include "FourierTransform.h"

using namespace std;

//Prototipos:

static void opt\_input(string const &);

static void opt\_output(string const &);

static void opt\_forward\_op(string const &);

static void opt\_reverse\_op(string const &);

static void opt\_block\_shift(string const &);

static void opt\_taps(string const &);

static void opt\_help(string const &);

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Elementos globales \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

static option\_t options[] = {

{1, "i", "input", "-", opt\_input, OPT\_DEFAULT},

{1, "o", "output", "-", opt\_output, OPT\_DEFAULT},

{1, "f", "forward-op", "-", opt\_forward\_op, OPT\_DEFAULT},

{1, "r", "reverse-op", "-", opt\_reverse\_op, OPT\_DEFAULT},

{1, "b", "block-shift", "-", opt\_block\_shift, OPT\_DEFAULT},

{1, "t", "taps", "-", opt\_taps, OPT\_DEFAULT},

{0, "h", "help", NULL, opt\_help, OPT\_DEFAULT},

{0, },

};

static istream \*iss = 0; // Input Stream (clase para manejo de los flujos de entrada)

static ostream \*oss = 0; // Output Stream (clase para manejo de los flujos de salida)

static fstream ifs; // Input File Stream (derivada de la clase ifstream que deriva de istream para el manejo de archivos)

static fstream ofs; // Output File Stream (derivada de la clase ofstream que deriva de ostream para el manejo de archivos)

static istream \*taps = 0; //Variable para manejar el archivo con los coeficientes de ecualizacion

static fstream taps\_fs; //Variable para la apertura del archivo con los coeficientes de ecualizacion

static int block\_shift;

FourierTransform\_t fourier\_transformation\_type;

FourierTransform\_t fourier\_transformation\_type\_inv;

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

//Implementacion de Prototipos:

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*\*\*\*\*\*\*\*Funciones correspondientes a "command line"\*\*\*\*\*\*\*\*\*\*\*/

static void

opt\_input(string const &arg){

if (arg == "-")

iss = &cin; // Establezco la entrada estandar cin como flujo de entrada

else {

ifs.open(arg.c\_str(), ios::in);

iss = &ifs;

}

if (!iss->good()) {

cerr << "cannot open "

<< arg

<< "."

<< endl;

exit(1);

}

}

static void

opt\_output(string const &arg){

if (arg == "-") {

oss = &cout; // Establezco la salida estandar cout como flujo de salida

}

else {

ofs.open(arg.c\_str(), ios::out);

oss = &ofs;

}

if (!oss->good()) {

cerr << "cannot open "

<< arg

<< "."

<< endl;

exit(1);

}

}

static void

opt\_help(string const &arg)

{

cout << "tp1 [-f funcion(dft o fft)] [-r funcion(rdft o rfft)] [-b int] [-t int] [-i file] [-o file]"

<< endl;

exit(0);

}

/\*A continuación se implementan las funciones propias del presente Trabajo

"opt\_forward\_op" y "opt\_reverse\_op" asignan en nuevos tipos enumerativos que operaciones van a realizarse para la transformación y la inversa. La razón por la cuál solo se asignan tipos enumerativos en dichas funciones es para, a la hora de crear los objetos correspondientes solo se creen los necesarios\*/

static void

opt\_forward\_op(string const &arg){

if (arg == "fft") {fourier\_transformation\_type= FFT\_FLAG;}

else{

fourier\_transformation\_type = DFT\_FLAG;

}

}

static void

opt\_reverse\_op(string const &arg){

if (arg == "ifft") {fourier\_transformation\_type\_inv= FFT\_FLAG;}

else{

fourier\_transformation\_type\_inv = DFT\_FLAG;

}

}

static void

opt\_block\_shift(string const &arg){

if (arg == "-") {

block\_shift = 1;

}

else{

istringstream iss(arg);

if(!(iss >> block\_shift) || !iss.eof()){

cerr << "non-integer: "

<< arg

<< "."

<< endl;

exit(1);

}

if (iss.bad()) {

cerr << "cannot read integer factor."

<< endl;

exit(1);

}

}

block\_shift = pow(2,block\_shift);

}

static void

opt\_taps(string const &arg){

if (arg == "-"){}//no se realiza ninguna operacion, mas adelante en el programa, si el puntero "\*taps\_is" es cero, el vector con los coeficientes se rellena con el valor 1

else{

taps\_fs.open(arg.c\_str(), ios::in);

taps = &taps\_fs;

if(!taps->good()) {

cerr << "cannot open "

<< arg

<< "."

<< endl;

exit(1);

}

}

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*\*\*\*\*\*\*\*Funcionas correspondientes a "main"\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

//La siguiente función crea el objeto que se utilizara para la transformacion (o inversa)

FourierTransform\* build\_fourier\_transformation\_type(FourierTransform\_t type){

FourierTransform \* aux;

switch(type){

case DFT\_FLAG:

aux= new Dft;

break;

case FFT\_FLAG:

aux= new Fft;

break;

default:

cerr << "fourier transformation not specified" << endl;

exit(1);

}

return aux;

}

//La siguiente función destruye el objeto que se creo anteriormente,ya sea para el foward o el reverse

void destroy\_fourier\_transformation\_type(FourierTransform\* aux){delete aux;}

//La siguiente funcion realiza la transformacion de la señal de entrada hasta su paso por la ecualizacion e imprime su resultado

void process\_signal(FourierTransform \*pFT, FourierTransform \*pFT\_inv){

bool is\_EOF=false, last\_block = true;

Vector Xin(block\_shift), Xout(block\_shift);

Vector Xout\_inv(block\_shift);

complejo aux;

size\_t i;

Vector Taps(block\_shift);

//se inicializa el vector con los coeficientes de ecualizacion

for(i=0;i<(size\_t)block\_shift;i++){

if( (taps==0 ) || !(\*taps >> aux ) ){

if(i<(size\_t)block\_shift)//se rellena el resto del vector con 1

for (; i < (size\_t)block\_shift; i++)

Taps[i] = 1;

break;

}

Taps[i] = aux;

}

//se inicia el ciclo de procesamiento de la señal

while (is\_EOF == false){

for(i=0;i<(size\_t)block\_shift;i++){

if( !(\*iss >> aux )){

is\_EOF = true, last\_block = true;

if(i<(size\_t)block\_shift && (i != 0))//se rellena el resto del vector con 0

for (; i < (size\_t)block\_shift; i++)

Xin[i] = 0;

break;

}

Xin[i] = aux;

}

if(i==0){break;}//Si no hay nuevos elementos, se sale del bucle "while"

if(is\_EOF == false || last\_block == true){

pFT->fourier\_transform(Xin, Xout); //Se aplica la transformacion

for(size\_t i = 0; i<(size\_t)block\_shift;i++) //Se multiplica por los coeficientes

Xout[i] \*= Taps[i];

// for(size\_t i=0;i<(size\_t)block\_shift;i++) //Si se quisiera

// cout << Xout[i] << endl; visualizar lo que hace la transformada

pFT\_inv->inverse\_fourier\_transform(Xout, Xout\_inv); //Se aplica la inversa

for(size\_t i=0;i<(size\_t)block\_shift;i++)

\*oss << Xout\_inv[i] << "\t";

}

}//Fin del bucle "while"

\*oss << endl;

if (oss->bad()) {

cerr << "unsuccessfull write (output file)" << endl;

exit(1);

}

if (iss->bad()) {

cerr << "unsuccessfull read (input file)" << endl;

exit(1);

}

if (!iss->eof()) {

cerr << "there isn't EOF on input file" << endl;

exit(1);

}

}

int main(int argc, char \* const argv[]){

cmdline cmdl(options);

cmdl.parse(argc, argv);

FourierTransform \*p\_FT = 0, \*p\_FT\_inv = 0;

p\_FT = build\_fourier\_transformation\_type(fourier\_transformation\_type);

p\_FT\_inv = build\_fourier\_transformation\_type(fourier\_transformation\_type\_inv);

process\_signal(p\_FT, p\_FT\_inv);

destroy\_fourier\_transformation\_type(p\_FT);

destroy\_fourier\_transformation\_type(p\_FT\_inv);

return 0;

}

***FourierTransform.h***

#ifndef \_FOURIER\_TRANSFORM\_

#define \_FOURIER\_TRANSFORM\_

#include "complejo.h"

#include "vector.h"

#include <cmath>

#define PI 3.14159265359

typedef enum{DFT\_FLAG, FFT\_FLAG} FourierTransform\_t;

class FourierTransform{

public:

virtual void fourier\_transform(Vector &in\_vector, Vector &out\_vector){};

virtual void inverse\_fourier\_transform(Vector &in\_vector, Vector &out\_vector){};

virtual ~FourierTransform(){};

};

//Clase derivada de "FourierTransform":

class Dft : public FourierTransform{ //se crea la clase Dft defivada de la transformada de fourier

public:

void fourier\_transform(Vector &in\_vector, Vector &out\_vector); //convierte a "fourier\_transform" en funcion no virtual

void inverse\_fourier\_transform(Vector &in\_vector, Vector &out\_vector);

~Dft(){};

};

//Clase derivada de "FourierTransform":

class Fft : public FourierTransform{

public:

void fourier\_transform(Vector &in\_vector, Vector &out\_vector);

void inverse\_fourier\_transform(Vector &in\_vector, Vector &out\_vector);

~Fft(){};

};

#endif

***FourierTransform.cc***

#include "FourierTransform.h"

void Dft::fourier\_transform(Vector &in\_vector, Vector &out\_vector){

size\_t s = in\_vector.RetSize();

for(size\_t k = 0; k<s;k++){

double real\_sum = 0;

double imag\_sum = 0;

for(size\_t n = 0; n<s;n++){

double fase = n \* k \* 2 \* PI / s;

real\_sum += in\_vector[n].re() \* cos(fase) + in\_vector[n].im() \* sin(fase);

imag\_sum += -in\_vector[n].re() \* sin(fase) + in\_vector[n].im() \* cos(fase);

}

out\_vector[k] = complejo(real\_sum, imag\_sum);

}

}

void Dft::inverse\_fourier\_transform(Vector &in\_vector, Vector &out\_vector){

size\_t s = in\_vector.RetSize();

for(size\_t n = 0; n<s;n++){

double real\_sum = 0;

double imag\_sum = 0;

for(size\_t k = 0; k<s;k++){

double fase = n \* k \* 2 \* PI / s;

real\_sum += in\_vector[k].re() \* cos(fase) - in\_vector[k].im() \* sin(fase);

imag\_sum += in\_vector[k].re() \* sin(fase) + in\_vector[k].im() \* cos(fase);

}

out\_vector[n] = (complejo(real\_sum, imag\_sum))/s;

}

}

Vector fft(Vector &X){

complejo aux;

size\_t N;

Vector Xout(N=X.RetSize());

if( N >= 2){

Vector P(N/2+ N%1), P\_TF(N/2);

for(size\_t i=0, j=0; j<P.RetSize(); j++, i= i+2)

P[j] = X[i];

Vector Q(N/2), Q\_TF(N/2);

for(size\_t i=1, j=0; j<Q.RetSize(); j++, i= i+2)

Q[j] = X[i];

P\_TF = fft(P);

Q\_TF = fft(Q);

for(size\_t k = 0; k<N/2;k++){

complejo angle = complejo(cos(2\*PI\*k/N),-sin(2\*PI\*k/N)) \* Q\_TF[k];

Xout[k] = P\_TF[k] + angle;

Xout[k+ N/2] = P\_TF[k] - angle;

}

}

else{

Xout = X;

}

return Xout;

}

void Fft::fourier\_transform(Vector &in\_vector, Vector &out\_vector){

out\_vector = fft(in\_vector);

}

void Fft::inverse\_fourier\_transform(Vector &in\_vector, Vector &out\_vector){

for(size\_t i = 0, N = out\_vector.RetSize(); i < N; i++)

in\_vector[i].conjugar();

out\_vector = fft(in\_vector);

for(size\_t i = 0, N = out\_vector.RetSize(); i < N; i++)

out\_vector[i] = out\_vector[i] / N;

}

***Vector.h***

#ifndef \_VECTOR\_H\_

#define \_VECTOR\_H\_

#include <iostream>

#include "complejo.h"

#include "ErrorSubin.h"

class Vector{

private:

complejo \*array;

size\_t size;

public:

//Constructores y destructores

Vector();

Vector(const Vector &);

Vector(size\_t);

~Vector();

//Metodos

size\_t RetSize(void){return size;}

//Operadores

complejo &operator[ ]( size\_t );

Vector &operator=( const Vector & );

};

#endif

***Vector.cc***

#include "vector.h"

Vector::Vector(){

size = 1;

array = new complejo[size];

}

Vector::Vector(const Vector &V){

size = V.size;

array = new complejo[size];

for(size\_t i = 0; i<size; i++)

array[i] = V.array[i];

}

Vector::Vector(size\_t sz){

size = sz;

array = new complejo[size];

}

Vector::~Vector(){

delete [] array;

}

complejo& Vector::operator [ ]( size\_t subscript ){

if( (0 > subscript)|| (subscript > size) )throw ErrorSubin();

else{

return array[ subscript ]; // retorna referencia

}

}

Vector& Vector::operator=( const Vector &rigth ){

if ( &rigth != this ){ // chequea para evitar autoasignaciones

// para arrays de diferentes tamaños, libera espacio usado y

//luego genera zona nueva y la carga

if ( size != rigth.size ){

complejo \*aux; //auxiliar por si no se obtiene el espacio; en ese caso se eligió dejar el array como estaba

aux=new complejo[ rigth.size ]; //se pide espacio; si no se obtiene new lanza bad\_alloc

delete [] array; // si llegó acá es que obtuvo el espacio;libera el anterior espacio

size =rigth.size ; // asigna el tamaño adecuado

array=aux; //apunta a la nueva zona

for ( size\_t i = 0; i < size; i++ )

array[ i ] = rigth.array[ i ]; // copia el array en el nuevo objeto

return \*this; // al retornar una referencia permite x = y = z;

}

else{ //si tienen el mismo tamaño, sólo hay que reasignar valores

for ( size\_t i = 0; i < size; i++ )array[ i ] = rigth.array[ i ]; // copia el array en el nuevo objeto

}

}

return \*this;

}

***Makefile***

CXXARGS = -g -Wall

CXXFLAGS = -I. $(CXXARGS)

all: tp1.exe

tp1.exe: main.o vector.o cmdline.o complejo.o FourierTransform.o

$(CXX) $(CXXFLAGS) -o tp1.exe main.o vector.o cmdline.o complejo.o FourierTransform.o

main.o: main.cpp cmdline.h vector.h

$(CXX) $(CXXFLAGS) -o main.o -c main.cpp

vector.o: vector.cc vector.h complejo.h

$(CXX) $(CXXFLAGS) -o vector.o -c vector.cc

cmdline.o: cmdline.cc cmdline.h complejo.h

$(CXX) $(CXXFLAGS) -o cmdline.o -c cmdline.cc

complejo.o: complejo.cc complejo.h

$(CXX) $(CXXFLAGS) -o complejo.o -c complejo.cc

FourierTransform.o: FourierTransform.cc FourierTransform.h

$(CXX) $(CXXFLAGS) -o FourierTransform.o -c FourierTransform.cc

clean:

$(RM) -vf \*.o \*.~ \*.t \*.out \*.err

**Análisis de algoritmos**

* DFT:

Para el análisis de la complejidad temporal de este algoritmo, se puede observar que presenta varios elementos de tiempo constante , luego unas sentencias que contienen dos ciclos anidados que realizan una operación de en cada “vuelta” , es decir que se representa en notación asintótica (peor caso) como n\*. Entonces haciendo un abuso de notación podemos expresar lo anterior como:

Luego para su complejidad espacial, el mismo trabaja sobre los dos vectores que recibe de tamaño n, y nada más necesita algunas variables auxiliares que no modifican de forma significativa. Por lo tanto su complejidad espacial es:

* IDFT:

El análisis es el mismo que en el DFT, por lo tanto tiene la misma complejidad temporal y espacial.

* FFT:

Este algoritmo es un algoritmo recursivo, por lo tanto la expresión matemática obtenida será recursiva también, la cual se puede obtener analizando el algoritmo de la siguiente forma; el algoritmo se llama a si mismo, es decir recursivamente, 2 veces por cada pasada y se realizarán unas veces.

La expresión resultante de este análisis es:

Se puede resolver mediante el “Teorema Maestro” y obtenemos como resultado que:

Este resultado es el esperado, ya que es la complejidad temporal conocida del algoritmo más común para la fft, el algoritmo Cooley-Tukey.

Para la complejidad espacial, se puede observar que teniendo un vector de entrada de tamaño n se realizan 4 subvectores de tamaño n/2 y luego se aplica la recursividad mencionada anteriormente, la expresión es:

Expresión que se puede resolver por teorema maestro también, resultando:

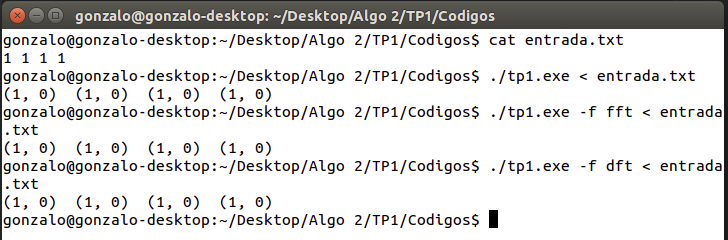
* IFFT:

El análisis temporal y espacial es , nuevamente, el mismo que el de su correspondiente inversa.

**Corridas de prueba**

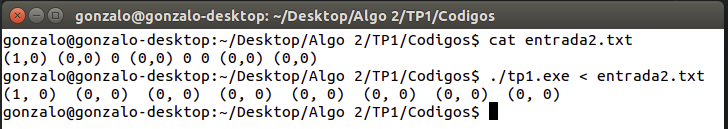
A continuación se muestran los resultados de las invocaciones especificadas en la guía del trabajo práctico:

*Ejemplo 1:*



Como podemos apreciar, el programa devuelve exactamente lo que recibe en la entrada, por lo tanto la transformación y su inversa fue un éxito, coincidiendo con los valores que se muestran en la guía del trabajo práctico. Además, verificamos que ambas transformadas “dft” y “fft” despliegan el mismo resultado.

*Ejemplo 2*



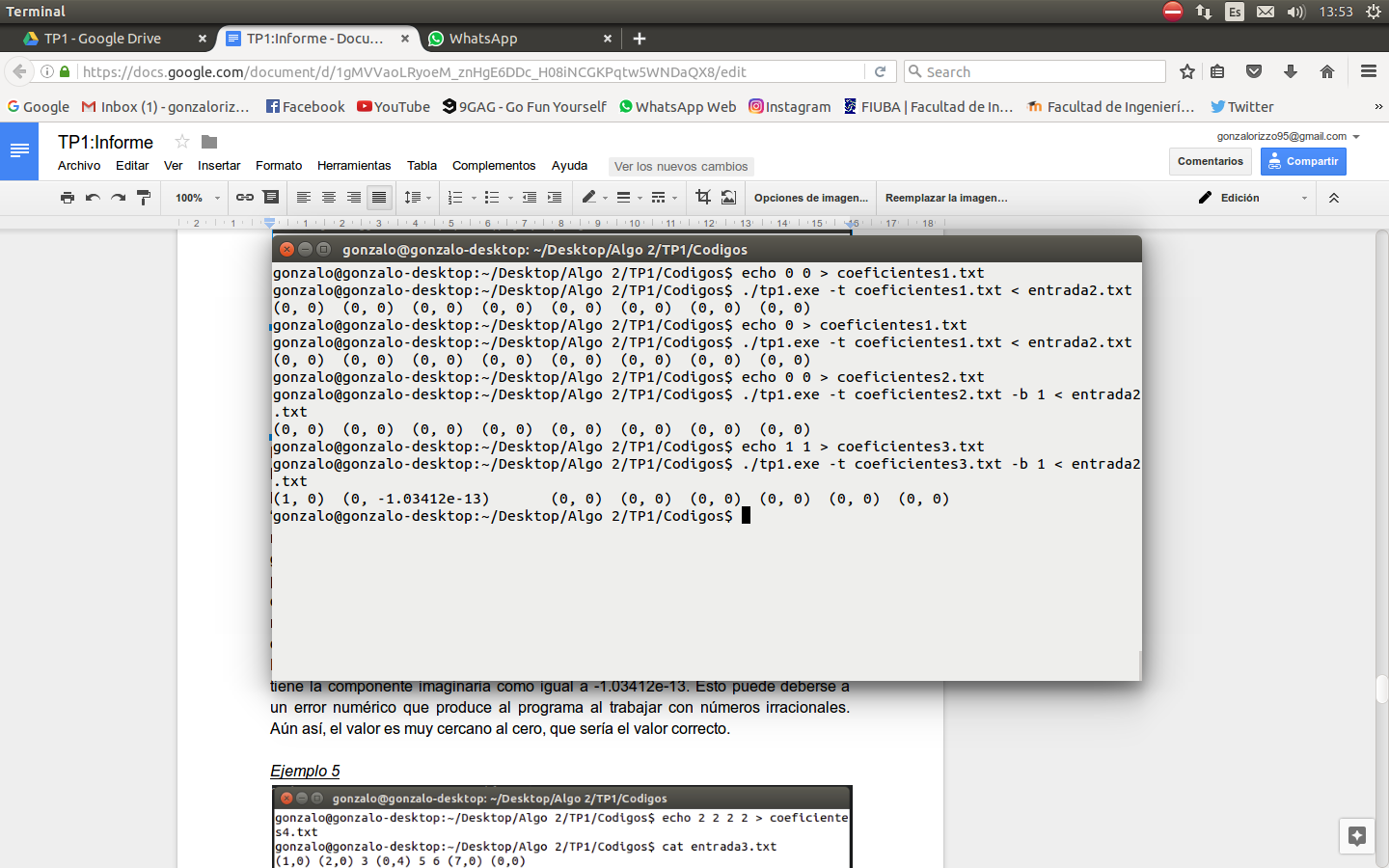
Una vez más, se verifica el correcto funcionamiento del programa, esta vez ensayando su desempeño utilizando una entrada con números complejos.

*Ejemplo 3*



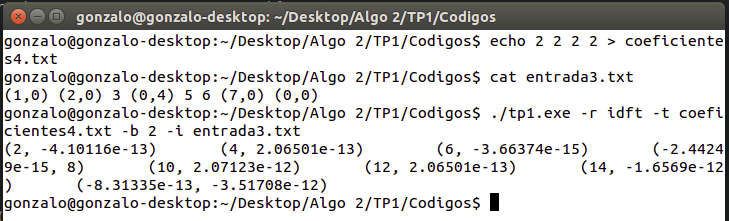
Nuevamente, el programa despliega la salida esperada. En este ejemplo, se verificaron distintas combinaciones de transformada y su inversa, para verificar que las salidas del programa sean las mismas.

*Ejemplo 4*



En este caso podemos observar 2 cosas. Primero, en estas corridas de programa se le pusieron argumentos al flag “-t” con el nombre de distintos archivos. Notamos que en las primeras invocaciones, se obtiene el resultado que presenta en la guía del trabajo práctico. Luego vemos que en la última invocación, en el segundo valor, se tiene la componente imaginaria como igual a -1.03412e-13. Esto puede deberse a un error numérico que produce al programa al trabajar con números irracionales. Aún así, el valor es muy cercano al cero, que sería el valor correcto.

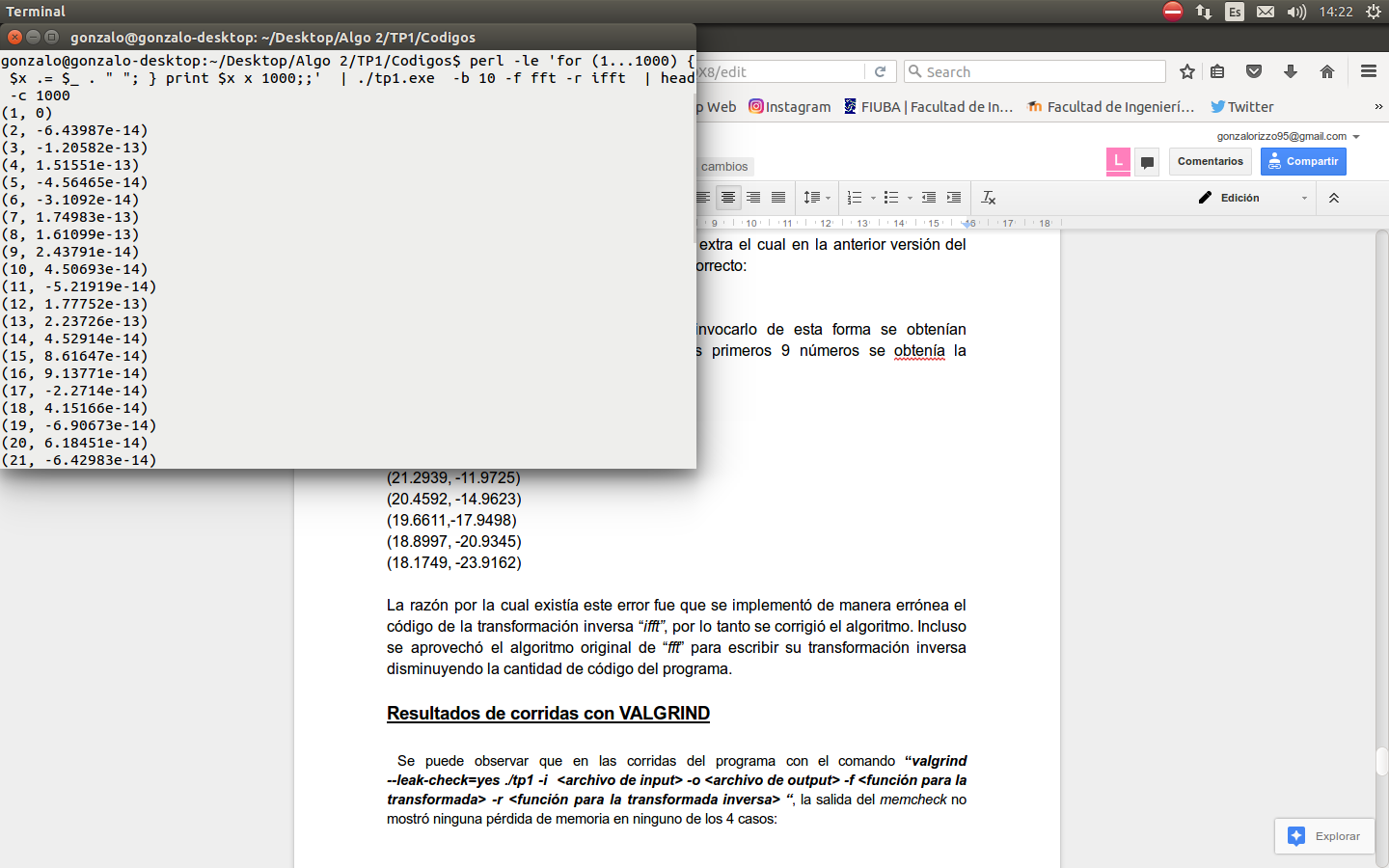
*Ejemplo 5*



Para este ejemplo, se aplicó un archivo de “Taps” el cual se espera que la salida del programa sean la entrada multiplicada por 2. En este caso, se repite la situación del ejemplo anterior, donde algunos valores que deberían ser 0, dan números relativamente cercanos a este, lo que se debe a un error numérico.

*Ejemplo 6:*

En este caso, incluimos un caso de prueba extra el cual en la anterior versión del programa el resultado que recibiamos era incorrecto:



En la anterior versión del programa, al invocarlo de esta forma se obtenían resultados incorrectos. Por ejemplo en los primeros 9 números se obtenía la siguiente salida:

(1, 0)

(24.0184, -2.99426)

(23.0735, -5.98806)

(22.1653,-8.98096)

(21.2939, -11.9725)

(20.4592, -14.9623)

(19.6611,-17.9498)

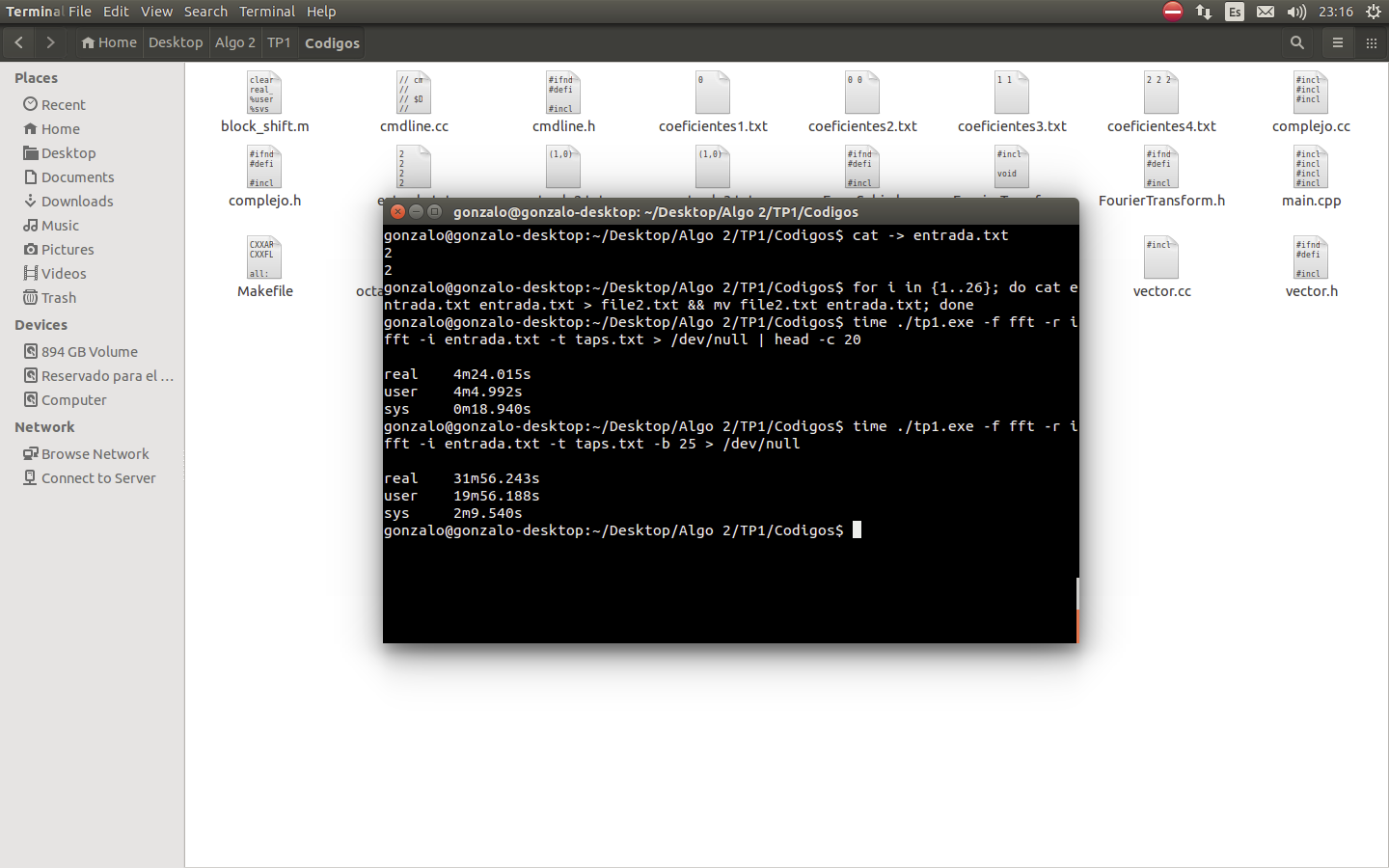
(18.8997, -20.9345)

(18.1749, -23.9162)

La razón por la cual existía este error fue que se implementó de manera errónea el código de la transformación inversa “*ifft”*, por lo tanto se corrigió el algoritmo. Incluso se aprovechó el algoritmo original de “*fft*” para escribir su transformación inversa disminuyendo la cantidad de código del programa. Para ello, se conjugan todos los elementos del vector al que se le aplicará la inversa, se le aplica la transformación *“fft”* y luego se multiplica cada elemento por el factor 1/N.

*Ejemplo 7: caso extremo*

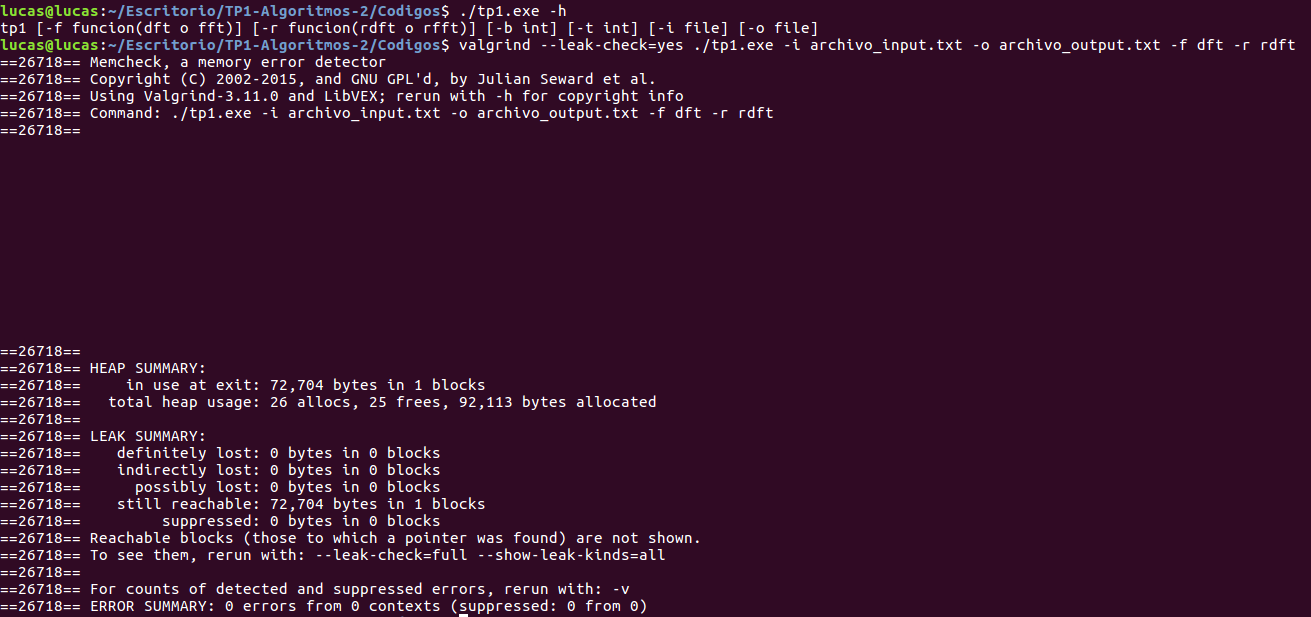
En este caso se pasó por el programa una entrada excesiva para ver que el mismo termina sin errores.



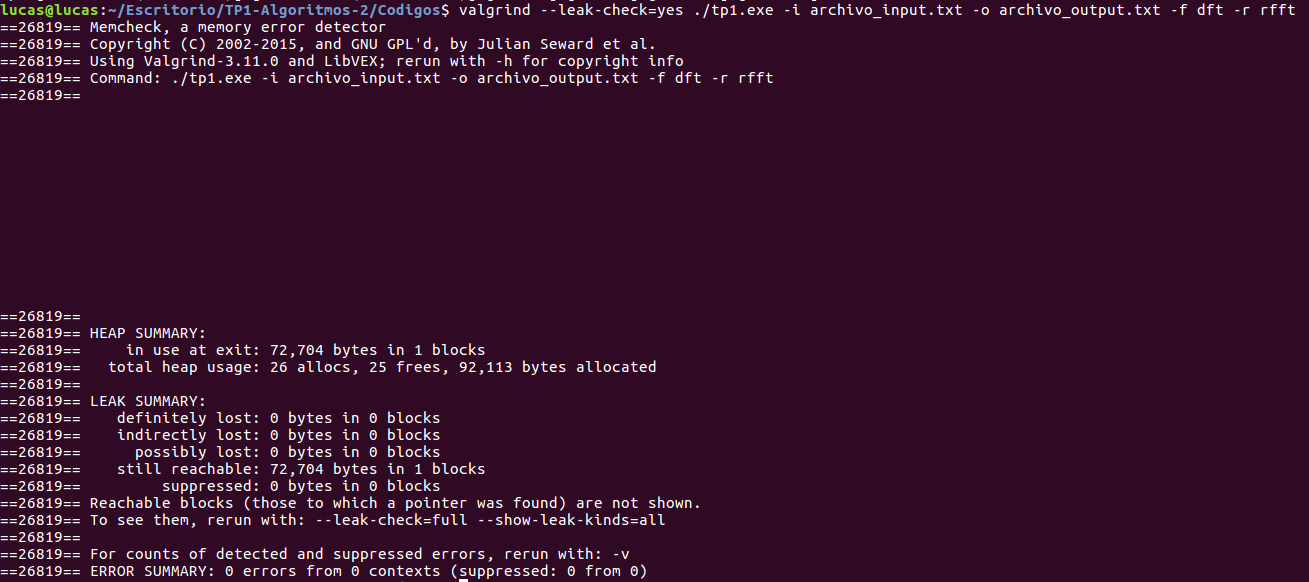
**Resultados de corridas con VALGRIND**

Se puede observar que en las corridas del programa con el comando **“*valgrind --leak-check=yes ./tp1 -i <archivo de input> -o <archivo de output> -f <función para la transformada> -r <función para la transformada inversa> “***, la salida del *memcheck* no mostró ninguna pérdida de memoria en ninguno de los 4 casos:

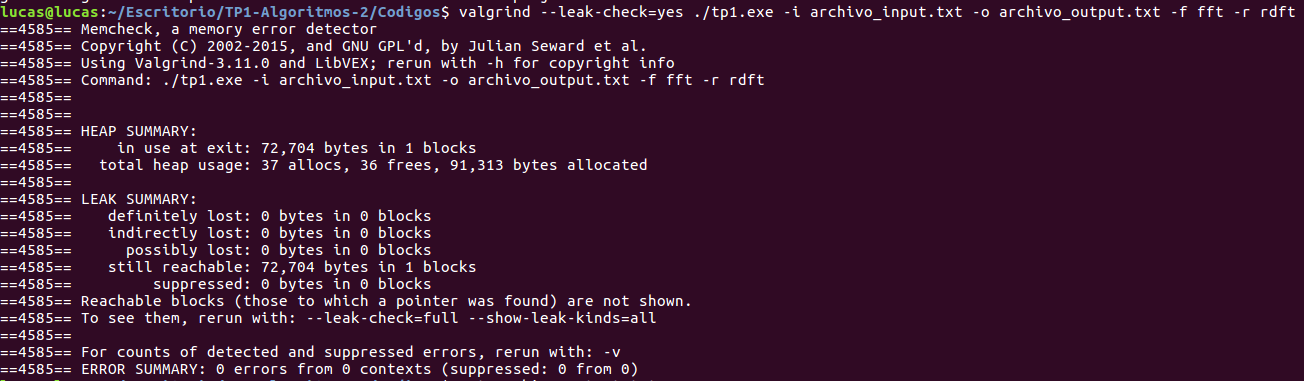
* Con “-f **dft”** y “-r **idft”**



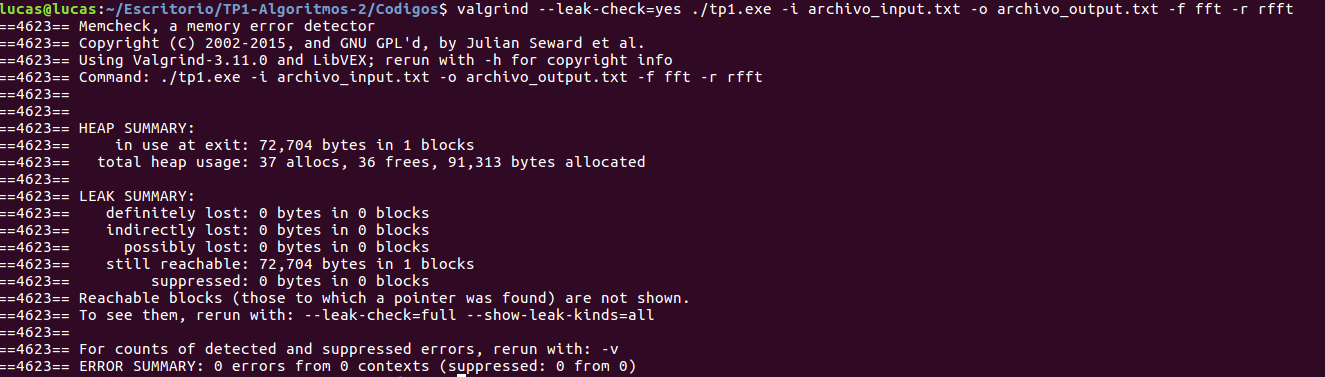
* Con “-f **dft”** y “-r **ifft”**



* Con “-f **fft”** y “-r **idft”**



* Con “-f **fft”** y “-r **ifft”**



**Valor ideal de “block\_shift” para minimizar tiempo ejecución**

Ante una entrada de 10 millones de números, se tomaron los tiempos de ejecución del programa para distintos block shift a modo de averiguar el valor óptimo de este para minimizar el tiempo de ejecución del programa. Para esto, se utilizó el comando de *Shell*, *time,* antes del nombre del programa de la siguiente manera (utilizando siempre la transformación *fft*):

* $ (time ./tp1.exe -b **x** -f fft -r ifft -i entrada.txt > /dev/null) &>> result.txt

Graficando los resultados obtenidos del tiempo real medido, con block\_shift desde 0 hasta 25, se obtuvo el siguiente gráfico:

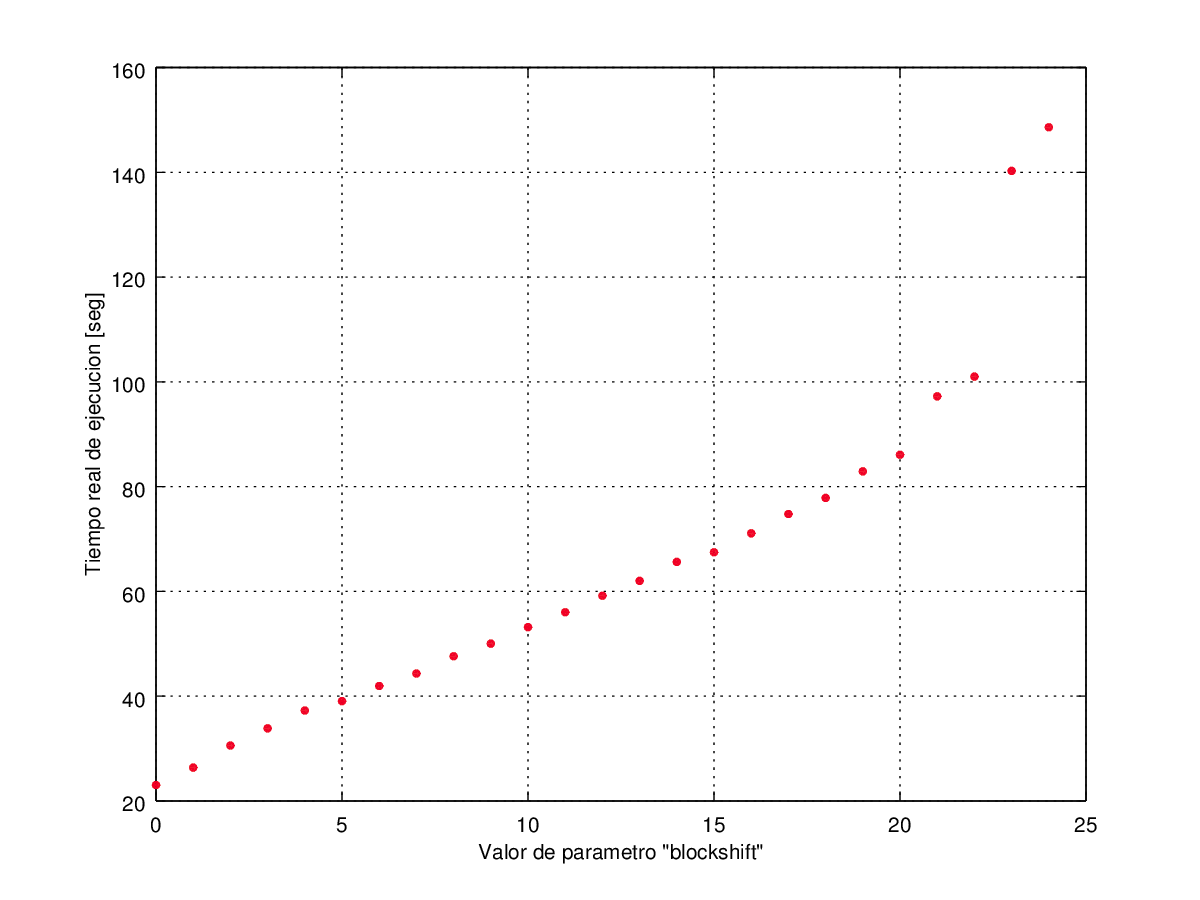


Gráfico 1: tiempos de ejecución para obtener valor ideal de block\_shift.

Del gráfico resulta evidente al observar el gráfico que el valor óptimo para block\_shift resulta ser el 0. Lo cual es lógico ya que de otra forma, el programa necesita tiempo para rellenar el vector block\_size el cual crece con el tamaño de block\_shift.

**FFT vs DFT**

Para poder hacer una comparación entre la transformación realizada con el algoritmo **dft** contra el algoritmo **fft**, se realizaron las pruebas con un archivo que contenía la cantidad de números (entre 0 y 10) mostrado en la primera columna (todas las pruebas fueron realizadas con un blocksize de 0 (-b 0) y el mismo algoritmo inverso).

|  |  |  |  |
| --- | --- | --- | --- |
| **Tamaño del archivo input** | **dft (con idft)**  **[seg]** | **fft (con idft)**  **[seg]** | **fft (con ifft)**  **[seg]** |
| 100 | 0.003 | 0.003 | 0.003 |
| 500 | 0.007 | 0.008 | 0.010 |
| 1000 | 0.015 | 0.012 | 0.011 |
| 10000 | 0.123 | 0.072 | 0.055 |
| 50000 | 0.456 | 0.253 | 0.163 |
| 100000 | 1.520 | 0.660 | 0.462 |
| 250000 | 2.100 | 1.433 | 1.050 |
| 500000 | 3.590 | 2.622 | 2.520 |
| 750000 | 4.000 | 3.061 | 3.000 |
| 1000000 | 5.480 | 4.421 | 3.970 |
| 2000000 | 10.100 | 7.918 | 6.884 |
| 4000000 | 20.560 | 15.976 | 14.645 |
| 8000000 | 43.600 | 33.027 | 28.654 |
| 10000000 | 57.240 | 41.726 | 35.018 |

Para hacer la diferencia más evidente, se graficaron en escala lineal los tiempos de ejecución de cada una en el Gráfico 2. Como era de esperarse, la aplicación de *fft* tiene menor tiempo real de ejecución que la *dft.*

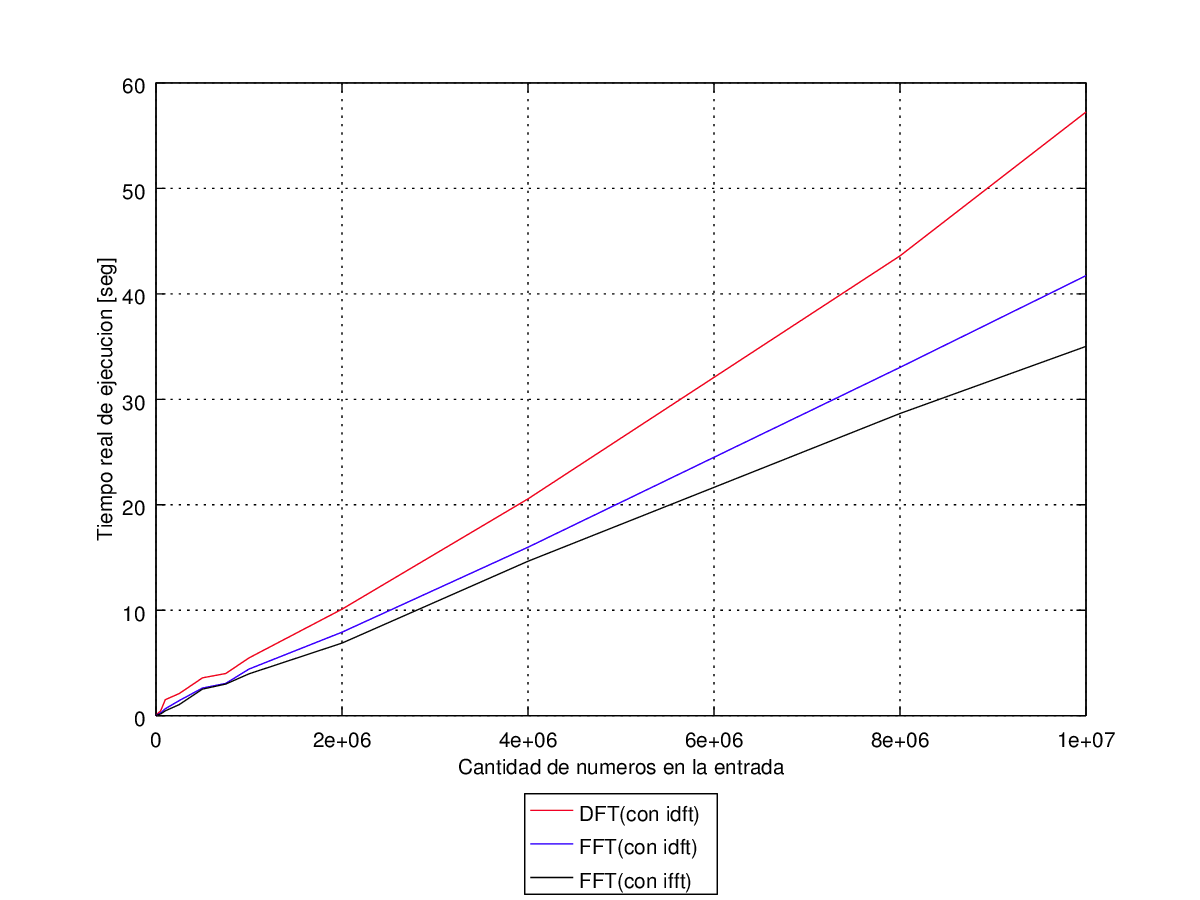


Gráfico 2: Tiempos de ejecución DFT vs FFT en escala lineal.

**Conclusiones**

Luego del análisis y corridas de prueba del programa producido, pudimos verificar que el mismo cumple con las especificaciones pedidas. Para este caso particular, se presentó el problema de la transformada de Fourier, la cual se buscaba comparar a nivel algorítmico dos de sus implementaciones, uno que utilizaba técnicas iterativas y otro técnicas recursivas. El resultado dio a lugar que el algoritmo recursivo “fft” presenta una rapidez mayor que el “dft”, bajando del orden a . Este beneficio se pagó en nuestra implementación con el uso de la memoria, ya que el algoritmo recursivo, en sus repetidas llamadas, creaba más vectores auxiliares. Aún así, es posible en un futuro realizar un mejoramiento del algoritmo “fft”, realizando cálculo *in-place* para ahorrar memoria, o realizando una variante iterativa del algoritmo “fft”.