FACULTAD DE INGENIERÍA
DE LA
UNIVERSIDAD DE BUENOS AIRES

Algoritmos y Programación II [95.12]

Trabajo Práctico n.° 1:

Objetos y algoritmos

Integrantes: Grassi, Tomás Miguel (99551) - tomas96@gmail.com

Martinez Mikulic, Mateo (99602) - mmartinezmikulic@gmail.com

Wagner, Marcos (98607) - marcoswagneer.18@gmail.com

Profesor: Ing. Calvo, Patricia

Ing. Santi, Leandro

Lic. Santi, Lucio

Curso 1
Jueves 14 de Junio de 2018

1. Introducción

En este trabajo se busca obtener conocimientos de programación orientada a objetos y de diseño de algoritmos. Para ello se realiza un programa utilizando el patrón de diseño Strategy y se mejora el diseño original previo de la transformada discreta de Fourier implementando la transformación rápida de Fourier (FFT) y su transformada inversa (IFFT).

2. Diseño e implementación del programa

2.1. Diseño del programa

Se implementó para este programa la clase **Complex** y la clase **Vector**, ambas implementadas como templates. De esta forma se logra una mayor simplicidad, ya que se codifica una única función sin importar el tipo de dato que se le pase como parámetro, evitando la duplicación de código y logrando un código más mantenible. Además, a partir del uso del uso de templates se logra una generalización de cada clase, ya que la misma se puede utilizar para distintos tipos de datos. Por ejemplo, la implementación de la clase **Complex** como plantilla permite utilizar el tipo de dato que se desee para armar el complejo, según la precisión que se requiera en la aplicación en la que se lo va a utilizar. Por otro lado, también para la clase vector, gracias a la implementación en forma de plantilla, es posible que cada instancia del vector contenga cualquier objeto, como un complejo, o tipo de dato que se necesite.

La clase **Complex** es utilizada para cargar cada uno de los datos leídos que luego se ordenan progresivamente (según el orden de entrada) dentro de la clase Vector.

Para la clase **Complex** se sobrecargaron los operadores correspondientes a la suma, resta, multiplicación, división, los cuales son utilizados en la implementación de todas las transformadas. Además se sobrecargaron los operadores de asignación, negación y comparación. De esta forma se logra un código más mantenible y legible. Por otro lado para la clase vector se sobrecargaron los operadores de asignación e indexación.

Por último, se utilizó el código provisto para el manejo de argumentos, realizando los cambios necesarios para que parsee el archivo de señales, de forma que lea una señal por línea.

2.2. Patrón de diseño y jerarquía de clases

Se utilizó para la transformada de Fourier el patrón de diseño *Strategy*. Esto permite reutilizar código y proporciona al usuario una interfaz uniforme, independiente del algoritmo que vaya a utilizar y de su implementación. La transformada de Fourier se representa como un objeto de la clase **FourierTransform**, cuyo único constructor público recibe un puntero a un elemento de la clase **FourierAlgorithm**, alojado con memoria dinámica. Esta clase es abstracta, y sus clases derivadas contienen contienen la implementación que será llamada al llamar al método *compute()*.

Los diversos algoritmos para calcular la transformada de Fourier estarán encapsulados en clases concretas derivadas de **FourierAlgorithm**. Dado lo mucho en común entre la DFT y la IDFT, se implementó una clase abstracta llamada **Discrete**, que hereda de **FourierAlgorithm**. Esto se debe a que el algoritmo para calcular la transformada discreta es prácticamente igual en ambos casos, con diferencias en qué coeficientes se utilizan; la implementación de ésta está contenida en la clase **Discrete**. De esta clase abstracta heredan las clases concretas **DFT** e **IDFT**, que sobrecargan el método puramente virtual protegido que **Discrete** llama para el cálculo de coeficientes.

La misma situación se repite en el caso de la transformada rápida de Fourier: existe una clase abstracta **Fast**, que deriva de **FourierAlgorithm**. Esta clase contiene la implementación del cálculo de la transformada rápida de Fourier, llamando su código a funciones puramente virtuales que

devuelven coeficientes y un divisor. Los valores de estos últimos son lo único que difiere entre la transformada directa y la inversa, y es por eso que se definen clases concretas que heredan de **Fast**, llamadas **FFT** e **IFFT**, que sobrecargan los métodos correspondientes en su clase padre.

De esta forma, utilizar una transformada, cualquiera sea, consiste en llamar al constructor de **FourierTransform**, pasándole como argumento un puntero a una nueva instancia de la transformada que se quiera utilizar, alojada con memoria dinámica (será el destructor de **FourierTransform** el encargado de liberar esa memoria). Llamar al método *compute()* sobre ese objeto llamará al método implementado en la clase abstracta correspondiente (**Fast** o **Discrete**), la cual, a su vez, llamará a las funciones correspondientes de la clase concreta con la que se haya construido la instancia de **FourierTransform**.

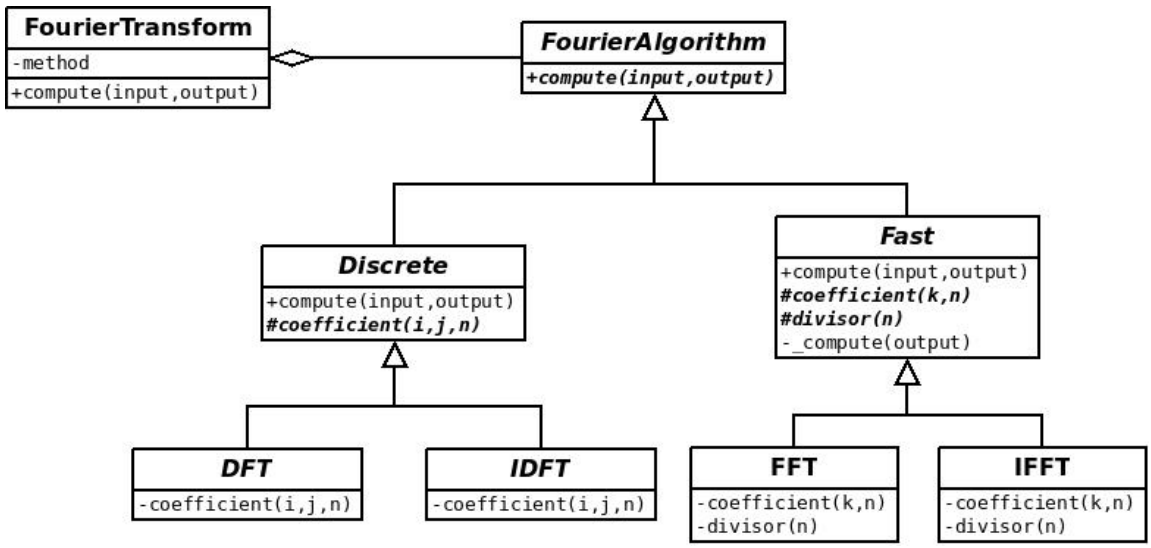

Figura 1: Diagrama de clases según el lenguaje unificado de modelado UML

2.3. Diseño del algoritmo de la transformada rápida de Fourier

La transformada rápida de Fourier se realizó de manera recursiva con el método dividir y conquistar, aprovechando la propiedad de las raíces complejas de la unidad y su periodicidad, que permite computar la DFT en tiempo $O(n \cdot log(n))$, en lugar de $O(n^2)$, siendo n el número de muestras de la señal de entrada a transformar. El requisito para esto es que el número de entradas sea una potencia de 2. Al realizar dividir y conquistar se divide en cada pasada recursiva a los elementos en posiciones par y a los posicionados en numeros impares, y se los procesa por separado, logrando reducir la cantidad de operaciones totales.

2.4. Interfaz

La interacción con el programa es a través de comandos en línea de ordenes. Tanto la entrada como la salida, puede direccionarse desde o hacia un archivo utilizando el flag correspondiente (*-i, -o* respectivamente) seguido del nombre del archivo. En caso de no indicar ningún archivo, el programa utiliza los flujos estándar de entrada y de salida. Por otro lado, también es posible indicar la transformada que se desea utilizar a partir del flag de método *-m* e indicando luego el método por su abreviatura (DFT, IDFT, FFT o IFFT). En caso de que esta opción no sea indicada, el programa realiza la FFT por defecto.

2.5. Formato de entrada y salida

El archivo de entrada será un archivo de texto con pares ordenados de complejos (re, im), separados por espacios. Cada línea en el archivo de entrada será una señal diferente. La salida tendrá el mismo formato, siendo cada línea la transformada o antitransformada de la señal correspondiente.

3. Corridas de prueba

Las pruebas realizadas son similares a las realizadas en el proyecto anterior. Solo se agregaron las pruebas correspondientes al procesamiento de datos con FFT e IFFT. En las pruebas realizadas con un número de entradas diferente a una potencia de dos, el resultado varía al desarrollado por la DFT, debido a que la transformada rápida rellena con ceros hasta que el vector tenga una potencia de dos.

Para realizar las pruebas de funcionamiento decidimos usar Google C++ Testing Framework, una herramienta que brinda herramientas para realizar pruebas sean independientes y repetibles. Permite separar las pruebas en módulos separados. Al fallar una prueba, según la naturaleza de la falla, la prueba puede seguir desarrollándose o simplemente detener ese módulo para continuar con el siguiente.

En este proyecto se realizaron pruebas sobre las clases desarrolladas y luego sobre el procesamiento de datos. Al realizar pruebas sobre las clases se verificó que las funciones correspondientes a la clase **Complex** funcionaran correctamente, tomando como patrón de comparación la clase **std::complex** definida en el archivo de cabecera *<complex.h>* provisto por la biblioteca estándar de C++. Al desarrollar las pruebas de procesamiento de datos el enfoque fue primero la verificación del correcto funcionamiento tanto de la función *DFT()* como de la función *IDFT()* y luego la robustez del proceso, realizando pruebas que variaban en el volumen de datos pseudo-aleatorios obtenidos con *rand()*.

Asimismo, se realizaron pruebas para determinar cuál debía ser el valor máximo en que pueden diferir dos números para ser considerados iguales. Para esto se fueron ejecutando las pruebas creadas para números complejos y para la DFT variando, de a potencias de diez, el factor por el que se multiplicaba a *std::numeric_limits<long double>::epsilon()*, el cual es la diferencia mínima que debe haber entre 1 y un número para que éste sea considerado el siguiente valor respresentable. De esta forma se llegó a una cota en el valor mínimo de comparación: 10^{-6}.

4. Problemas durante el desarrollo y soluciones

Al realizar el algoritmo de un proceso tan complejo como la transformada rápida de Fourier se presentaron diferentes problemas y contratiempos hasta que se pudo lograr una versión del algoritmo funcional. Estos errores fueron causados por el uso de la recursividad en un algoritmo del cual desconocíamos su funcionamiento. Los valores en el exponente en *coefficient()* y el desarrollo de la función inversa fue lo que más confusión causó. Por otro lado, se utilizó como herramienta Valgrind para rastrear la presencia de fugas de memoria. Además, el Google C++ Testing Framework facilita la tarea de probar el funcionamiento y permite de manera sencilla verificar el funcionamiento a medida que diferentes cambios son realizados al código.

5. Bibliografía

- Ghezzi, Carlo & Jazayeri, Mehdi & Mandrioli, Dino (1991). *Fundamentals of Software Engineering* (1.^{era} ed.) Upper Saddle River, NJ 07458: Prentice Hall, Inc.

- Stroustrup, Bjarne (1988). *The C++ Programming Language* (4.^{ta} ed.) Upper Saddle River, NJ 07458: Addison-Wesley.
- Cormen, Thomas & Leiserson, Charles & Rivest, Ronald & Stein, Clifford (1989). *Introduction to Algorithms* (1.^{era} ed.) Upper Saddle River, NJ 07458: MIT Press.

6. Script de compilación

makefile

```
CXXFLAGS = -g -Wall -Wpedantic -Wdeprecated -std=c++11 -O3
SRC = source
INCLUDE = include
TESTS = source/tests
GOOGLETEST =  source/tests/googletest
CXXARGS = -I. -iquote $(INCLUDE) -isystem $(GOOGLETEST)/include -pthread

all: fourier.exe Complex_test.exe fourier_test.exe

cmdline.o: $(SRC)/cmdline.cpp $(INCLUDE)/cmdline.h
   $(CXX) $(CXXFLAGS) $(CXXARGS) -c $(SRC)/cmdline.cpp -o cmdline.o

fourier.o: $(SRC)/fourier.cpp $(INCLUDE)/fourier.h $(INCLUDE)/Complex.h $
   (INCLUDE)/Vector.h
   $(CXX) $(CXXFLAGS) $(CXXARGS) -c $(SRC)/fourier.cpp -o fourier.o

io.o: $(SRC)/io.cpp $(INCLUDE)/io.h $(INCLUDE)/Complex.h $(INCLUDE)/
   Vector.h
   $(CXX) $(CXXFLAGS) $(CXXARGS) -c $(SRC)/io.cpp -o io.o

main.o: $(SRC)/main.cpp $(INCLUDE)/main.h $(INCLUDE)/io.h $(INCLUDE)/
   Complex.h $(INCLUDE)/Vector.h $(INCLUDE)/cmdline.h $(INCLUDE)/fourier.h
   $(CXX) $(CXXFLAGS) $(CXXARGS) -c $(SRC)/main.cpp -o main.o

fourier.exe: cmdline.o fourier.o io.o main.o
   $(CXX) $(CXXFLAGS) $(CXXARGS) cmdline.o fourier.o  io.o main.o -o
      fourier.exe

gtest-all.o:
   $(CXX) $(CXXFLAGS) -isystem $(GOOGLETEST)/include -I$(GOOGLETEST) -
      pthread -c $(GOOGLETEST)/src/gtest-all.cc -o gtest-all.o

Complex_test.o: $(TESTS)/Complex_test.cpp $(INCLUDE)/Complex_test.h $(
   INCLUDE)/Complex.h $(INCLUDE)/Vector.h
   $(CXX) $(CXXFLAGS) $(CXXARGS) -c $(TESTS)/Complex_test.cpp -o
      Complex_test.o

Complex_test.exe: gtest-all.o Complex_test.o cmdline.o
   $(CXX) $(CXXFLAGS) $(CXXARGS) gtest-all.o Complex_test.o cmdline.o -o
      Complex_test.exe

```

```
34 fourier_test.o: $(TESTS)/fourier_test.cpp $(INCLUDE)/fourier_test.h $(
      INCLUDE)/Complex.h $(INCLUDE)/Vector.h $(INCLUDE)/io.h $(INCLUDE)/
      cmdline.h $(INCLUDE)/fourier.h
35    $(CXX) $(CXXFLAGS) $(CXXARGS) -c $(TESTS)/fourier_test.cpp -o
         fourier_test.o
36
37 fourier_test.exe: fourier_test.o gtest-all.o cmdline.o
38    $(CXX) $(CXXFLAGS) $(CXXARGS) gtest-all.o fourier_test.o fourier.o
         cmdline.o io.o -o fourier_test.exe
39
40 clean:
41    $(RM) -vf *.o *.exe *.t *.out *.err
```

7. Código fuente

main.h

```
1  #ifndef _MAIN_H_INCLUDED_
2  #define _MAIN_H_INCLUDED_
3
4  #include "cmdline.h"
5  #include "Complex.h"
6  #include "Vector.h"
7  #include "io.h"
8  #include "fourier.h"
9
10 static void opt_input(std::string const &);
11 static void opt_output(std::string const &);
12 static void opt_method(std::string const &);
13 static void opt_help(std::string const & = "");
14 FourierAlgorithm* choose_method(std::string read_method);
15
16 #endif   // _MAIN_H_INCLUDED_
```

main.cpp

```
1  #include <fstream>
2  #include <iostream>
3  #include <sstream>
4  #include <cstdlib>
5
6  #include "main.h"
7
8  using namespace std;
9
10 static option_t options[] = {
11    {1, "i", "input", "-", opt_input, OPT_DEFAULT},
12    {1, "o", "output", "-", opt_output, OPT_DEFAULT},
13    {1, "m", "method", "FFT", opt_method, OPT_DEFAULT},
14    {0, "h", "help", nullptr, opt_help, OPT_DEFAULT},
15    {0, },
16 };
```

```
17
18 static char *program_name;
19 static FourierTransform *transform;
20 static istream *iss = nullptr;
21 static ostream *oss = nullptr;
22 static fstream ifs;
23 static fstream ofs;
24
25 static void
26 opt_input(string const &arg)
27 {
28     if (arg == "-") {
29         iss = &cin;
30     }
31     else {
32         ifs.open(arg.c_str(), ios::in);
33         iss = &ifs;
34     }
35
36     if (!iss->good()) {
37         cerr << "Cannot open "
38              << arg
39              << "."
40              << endl;
41         exit(1);
42     }
43 }
44
45 static void
46 opt_output(string const &arg)
47 {
48     if (arg == "-") {
49         oss = &cout;
50     } else {
51         ofs.open(arg.c_str(), ios::out);
52         oss = &ofs;
53     }
54
55     if (!oss->good()) {
56         cerr << "Cannot open "
57              << arg
58              << "."
59              << endl;
60         exit(1);
61     }
62 }
63
64 static void
65 opt_method(string const &arg)
66 {
67     istringstream iss(arg);
68     string read_method;
```

```
69
70      if (!(iss >> read_method) || iss.bad()) {
71          cerr << "Cannot read method."
72               << endl;
73          exit(1);
74      }
75
76      FourierAlgorithm *chosen_method = choose_method(read_method);
77
78      if (chosen_method == nullptr) {
79          cerr << "Not a posible method: "
80               << arg
81               << "."
82               << endl;
83          opt_help();
84          exit(1);
85      }
86      ::transform = new FourierTransform(chosen_method);
87      if (!::transform)
88          exit(1);
89  }
90
91  static void
92  opt_help(string const & arg)
93  {
94      cout << "Usage: "
95           << program_name
96           << " [-m FFT | IFFT | DFT | IDFT] [-i file] [-o file]"
97           << endl;
98      exit(0);
99  }
100
101 FourierAlgorithm *
102 choose_method (string read_method)
103 {
104     if (read_method == "FFT")
105         return new FFT;
106     if (read_method == "IFFT")
107         return new IFFT;
108     if (read_method == "DFT")
109         return new DFT;
110     if (read_method == "IDFT")
111         return new IDFT;
112     return nullptr;
113 }
114
115 int
116 main(int argc, char * const argv[])
117 {
118     program_name = argv[0];
119     cmdline cmdl(options);
120     cmdl.parse(argc, argv);
```

```
121
122     // Cuestiones de formato para la impresión:
123     oss->setf(ios::fixed, ios::floatfield);
124     oss->precision(6);
125
126     bool status = process(*::transform, *iss, *oss);
127
128     delete ::transform;
129     ::transform = nullptr;
130
131     return status;
132 }
```

fourier.h

```
1  #ifndef _FOURIER_H_INCLUDED_
2  #define  _FOURIER_H_INCLUDED_
3
4  #include "Complex.h"
5  #include "Vector.h"
6
7  using ComplexVector = Vector <Complex <long double> >;
8
9  class FourierAlgorithm {
10 public:
11     virtual bool compute(ComplexVector const & input, ComplexVector &
           output) = 0;
12     virtual ~FourierAlgorithm() {}
13 };
14
15 class FourierTransform {
16 public:
17     FourierTransform(FourierAlgorithm *method) : _method(method) {}
18     virtual ~FourierTransform() {
19         delete _method;
20     }
21     bool compute(ComplexVector const & input, ComplexVector & output) {
22         return _method? _method->compute(input, output) : false;
23     }
24 private:
25     FourierAlgorithm *_method;
26 };
27
28 class Discrete : public FourierAlgorithm {
29 public:
30     bool compute(ComplexVector const & input, ComplexVector & output);
31 protected:
32     virtual const Complex<> _coefficient(int const i, int const j, int
           const n) = 0;
33 };
34
35 class DFT : public Discrete {
36     const Complex <> _coefficient(int const i, int const j, int const n)
```

```
        override {
37          return exp(-I * 2.0 * M_PI * i * j / n);
38      }
39  };
40
41  class IDFT : public Discrete {
42      const Complex <> _coefficient(int const i, int const j, int const n)
            override {
43          return exp(I * 2.0 * M_PI * i * j / n) / n;
44      }
45  };
46
47  class Fast : public FourierAlgorithm {
48  public:
49      bool compute(ComplexVector const & input, ComplexVector & output);
50  protected:
51      virtual const Complex <> _coefficient(int const k, int const n) = 0;
52      virtual const size_t _divisor(size_t const n) = 0;
53  private:
54      ComplexVector _compute(ComplexVector const & input);
55  };
56
57  class FFT : public Fast {
58      const Complex <> _coefficient(int const k, int const n) override {
59          return exp(-I * 2.0 * M_PI * k / n);
60      }
61      const size_t _divisor(size_t const n) {
62          return 1;
63      }
64  };
65
66  class IFFT : public Fast {
67      const Complex <> _coefficient(int const k, int const n) override {
68          return exp(I * 2.0 * M_PI * k / n);
69      }
70      const size_t _divisor(size_t const n) {
71          return n;
72      }
73  };
74
75
76  #endif    // _FOURIER_H_INCLUDED_
```

fourier.cpp

```
1   #include <iostream>
2   #include <cmath>
3
4   #include "fourier.h"
5
6   bool
7   Discrete::compute(ComplexVector const & input, ComplexVector & output)
8   {
```

```
9     size_t n = input.size();
10    output.reserve(n);
11    Complex <> sum = 0;
12    for (size_t i = 0; i < n; ++i) {
13       for (size_t j = 0; j < n; ++j) {
14          sum += input[j] * _coefficient(i, j, n);
15       }
16       output.push_back(sum);
17       sum = 0;
18    }
19    return true;
20 }
21
22 bool
23 Fast::compute(ComplexVector const & input, ComplexVector & output)
24 {
25    output.clear();
26    size_t n = input.size();
27
28    if (n & (n - 1)) { // si el ñtamao no es una potencia de dos...
29       ComplexVector auxInput(input);
30       while (n & (n - 1)) {
31          auxInput.push_back(0); // ...rellenar con ceros hasta que lo sea
                .
32          ++n;
33       }
34       output = _compute(auxInput);
35    }
36    else
37       output = _compute(input);
38
39    for (size_t i = 0; i < output.size(); ++i)
40       output[i] /= _divisor(output.size());
41
42    return true;
43 }
44
45 ComplexVector
46 Fast::_compute(ComplexVector const & input)
47 {
48    size_t n = input.size();
49    if (n <= 1) {
50       return ComplexVector(input);
51    }
52
53    ComplexVector inputEven(n/2);
54    ComplexVector inputOdd(n/2);
55
56    for (size_t i = 0; i < n / 2; ++i) {
57       inputEven[i] = input[2 * i];
58       inputOdd[i] = input[2 * i + 1];
59    }
```

```
60
61    ComplexVector outputEven(_compute(inputEven));
62    ComplexVector outputOdd(_compute(inputOdd));
63
64    ComplexVector output(n);
65    for (size_t k = 0; k < n / 2; ++k){
66       output[k] = outputEven[k] + _coefficient(k, n) * outputOdd[k];
67       output[k + n/2] = outputEven[k] - _coefficient(k, n) * outputOdd[k
             ];
68    }
69
70    return output;
71 }
```

io.h

```
1  #ifndef _IO_H_INCLUDED_
2  #define _IO_H_INCLUDED_
3
4  #include "Complex.h"
5  #include "Vector.h"
6  #include "fourier.h"
7
8  bool load_signal(std::istream &, Vector<Complex<> > &);
9  bool print_signal(std::ostream &, Vector<Complex<> > const &);
10 void print_msg_and_exit(std::string const & msg);
11 bool process(FourierTransform& transform, std::istream& is, std::ostream&
       os);
12
13 #endif   //_IO_H_INCLUDED_
```

io.cpp

```
1  #include <iostream>
2  #include <cstdlib>
3  #include <sstream>
4
5  #include "io.h"
6
7  using namespace std;
8
9  bool
10 load_signal(istream & is, Vector<Complex<> > & input)
11 {
12    Complex <long double> c;
13    while (is >> c)
14       input.push_back(c);
15    if (is.bad())
16       return false;
17    return true;
18 }
19
20 bool
21 print_signal(ostream & os, Vector<Complex<> > const & output)
```

```
{
    for (size_t i = 0; i < output.size(); ++i)
        os << output[i] << ' ';
    os << endl;
    if (os.bad())
        return false;
    return true;
}

void
print_msg(string const & msg)
{
    cerr << msg
         << endl;
    exit(1);
}

bool
process(FourierTransform& transform, istream& is, ostream& os)
{
    ComplexVector inSignal;
    ComplexVector outSignal;
    istringstream line;
    string s;
    bool status;

    for (int lineNo = 1; getline(is, s); ++lineNo) {
        if (is.bad()) {
            print_msg("An error occured while processing line " + to_string(
                lineNo) + ".");
            return false;
        }

        line.clear(); // vacía los flags del istringstream.
        line.str(s);

        status = load_signal(line, inSignal);
        if (!status) {
            print_msg("Error processing \"" + line.str() + "\" (line " +
                to_string(lineNo) + ").");
            return false;
        }

        status = transform.compute(inSignal, outSignal);
        if (!status) {
            print_msg("An error occured while performing the requested
                operation.");
            return false;
        }

        status = print_signal(os, outSignal);
        if (!status) {
```

```
71            print_msg("Cannot write to output stream.");
72            return false;
73        }
74
75        // vacía los vectores para reutilizarlos en el siguiente ciclo.
76
77        inSignal.clear();
78        outSignal.clear();
79    }
80    return true;
81}
```

Vector.h

```
1 #ifndef _VECTOR_H_INCLUDED_
2 #define _VECTOR_H_INCLUDED_
3
4 #include <cassert>
5
6 template <typename T>
7 class Vector {
8    T* data;
9    size_t allocated;
10   size_t used;
11   const static size_t init_size = 15;
12   const static size_t chop_size = 20;
13public:
14   Vector() : data(new T[init_size]), allocated(init_size), used(0) {
15   }
16   // Reserva espacio para count elementos y les asigna el valor value:
17   //
18   Vector(size_t count, T const & value = 0) : data(new T[count]{value}),
         allocated(count), used(count) {
19   }
20   Vector(const Vector& v) : data(new T[v.used]), allocated(v.used), used
        (v.used) {
21       for (size_t i = 0; i < used; ++i)
22           data[i] = (v.data)[i];
23   }
24   ~Vector() {
25       delete[] data;
26       data = nullptr;
27   }
28   Vector& operator=(Vector const & v) {
29       // Check for self-assignment:
30       //
31       if (this == &v)
32           return *this;
33
34       // Same size optimization:
35       //
36       if (used == v.used) {
37           for (size_t i = 0; i < used; ++i)
```

```
38              data[i] = (v.data)[i];
39           return *this;
40        }
41        delete[] data;
42        used = v.used;
43        data = new T[used];
44        allocated = used;
45        for (size_t i = 0; i < used; ++i)
46           data[i] = (v.data)[i];
47        return *this;
48     }
49     // versión const:
50     //
51     const T& operator[](size_t position) const {
52        if (position >= used)
53           assert("Illegal position.");
54        return data[position];
55     }
56     // versión no const:
57     //
58     T& operator[](size_t position) {
59        if (position >= used)
60           assert("Illegal position.");
61        return data[position];
62     }
63     size_t size() const {
64        return used;
65     }
66     size_t capacity() const {
67        return allocated;
68     }
69     bool empty() const {
70        return (bool) used;
71     }
72     // agrega un elemento al final:
73     //
74     void push_back(T const & value) {
75        if (used == allocated)
76           reserve(allocated + chop_size);
77        data[used] = value;
78        ++used;
79     }
80     // llena el vector con count copias de valor value
81     //
82     void assign(size_t count, T const & value) {
83        if (count > allocated)
84           reserve(count);
85        used = count;
86        for (size_t i = 0; i < used; ++i)
87           data[i] = value;
88     }
89     // reserva espacio para new_capacity elementos
```

```
    //
    void reserve(size_t new_capacity) {
        if (new_capacity > allocated) {
            T* new_data = new T[new_capacity];
            allocated = new_capacity;
            for (size_t i = 0; i < used; ++i)
                new_data[i] = data[i];
            delete[] data;
            data = new_data;
        }
    }
    void clear() {
        for (size_t i = 0; i < allocated; ++i)
            data[i] = 0;
        used = 0;
    }
};

#endif    //_VECTOR_H_INCLUDED_
```

Complex.h

```
#ifndef _COMPLEX_H_INCLUDED_
#define _COMPLEX_H_INCLUDED_

#include <iostream>
#include <limits>
#include <algorithm>
#include <cmath>

// Función para la comparación con margen de error:
template<typename T>
inline const bool almostEqual(T a, T b);

template <typename T = long double>
class Complex {
    T x;
    T y;
public:
    Complex(T real = 0, T imag = 0) : x(real), y(imag) {}
    Complex(const Complex& C) : x(C.x), y(C.y) {}
    ~Complex() {}

    T re() const { return x; }
    T im() const { return y; }

    const Complex conj() const {
        return Complex(x, -y);
    }
    const T norm() const {
        return sqrt(x*x + y*y);
    }
    const T arg() const {
```

```
32      return std::atan2(y,x);
33   }
34   const Complex operator+() const {
35      return Complex(+x, +y);
36   }
37   const Complex operator-() const {
38      return Complex(-x, -y);
39   }
40   const Complex operator+(const Complex& c) const {
41      return Complex(x + c.x, y + c.y);
42   }
43   const Complex operator-(const Complex& c) const {
44      return Complex(x - c.x, y - c.y);
45   }
46   const Complex operator*(const Complex& c) const {
47      return Complex(x*c.x - y*c.y, y*c.x + x*c.y);
48   }
49   const Complex operator/(const Complex& c) const {
50      return Complex((x*c.x + y*c.y) / (c.x*c.x + c.y*c.y),
51                     (y*c.x - x*c.y) / (c.x*c.x + c.y*c.y));
52   }
53   Complex& operator=(const Complex& c) {
54      x = c.x;
55      y = c.y;
56      return *this;
57   }
58   Complex& operator+=(const Complex& c) {
59      x += c.x;
60      y += c.y;
61      return *this;
62   }
63   Complex& operator-=(const Complex& c) {
64      x -= c.x;
65      y -= c.y;
66      return *this;
67   }
68   Complex& operator*=(const Complex& c) {
69      x = x*c.x - y*c.y;
70      y = y*c.x + x*c.y;
71      return *this;
72   }
73   Complex& operator/=(const Complex& c) {
74      x = (x*c.x + y*c.y) / (c.x*c.x + c.y*c.y);
75      y = (y*c.x - x*c.y) / (c.x*c.x + c.y*c.y);
76      return *this;
77   }
78   bool operator==(const Complex & c) const {
79      return almostEqual(x, c.x) && almostEqual(y, c.y);
80   }
81   bool operator!=(const Complex& c) const {
82      return !almostEqual(x, c.x) || !almostEqual(y, c.y);
83   }
```

```
84      friend std::ostream& operator<<(std::ostream& os, const Complex& c) {
85          return os << '('
86                    << c.x
87                    << ','
88                    << ' '
89                    << c.y
90                    << ')';
91      }
92      friend std::istream& operator>>(std::istream& is, Complex& c) {
93          bool good = false;
94          bool bad = false;
95          T re = 0;
96          T im = 0;
97          char ch;
98          if (is >> ch && ch == '(') {
99              if (is >> re
100             && is >> ch
101             && ch == ','
102             && is >> im
103             && is >> ch
104             && ch == ')')
105                 good = true;
106             else
107                 bad = true;
108         }
109         else if (is.good()) {
110             is.putback(ch);
111             if (is >> re)
112                 good = true;
113             else
114                 bad = true;
115         }
116         if (good) {
117             c.x = re;
118             c.y = im;
119         }
120         else if (bad)
121             is.setstate(std::ios::badbit);
122         return is;
123     }
124 };
125
126 const Complex <long double> I(0, 1);
127 const long double Complex_acceptableDelta = 10e-6;
128
129 template <typename T> Complex <T>
130 exp(const Complex <T> & c)
131 {
132     return typename Complex<T>::Complex( std::exp (c.re()) * std::cos(c.im
        ()),
133              std::exp(c.re()) * std::sin(c.im()));
134 }
```

```
135
136 // Función para la comparación que comprueba si dos números difieren en
    lo
137 // suficientemente poco.
138 //
139 template <typename T> inline const bool
140 almostEqual(T a, T b)
141 {
142    const T absA = std::abs(a);
143    const T absB = std::abs(b);
144    const T absDelta = std::abs(a - b);
145    const bool deltaIsAcceptable = absDelta <= Complex_acceptableDelta;
146
147    if (a == b) // si son iguales, incluso inf
148       return true;
149    // si a o b son cero, o están lo suficientemente cerca
150    //
151    if (a == 0 || b == 0 || deltaIsAcceptable)
152       return true;
153    // sino, usar el error relativo
154    return Complex_acceptableDelta >
155           absDelta / std::min<T>(absA + absB, std::numeric_limits<T>::max
               ());
156 }
157
158 #endif   //_COMPLEX_H_INCLUDED_
```