ENGG 151.01 - Signals, Spectra, and Signal Processing, Lecture
ENGG 151.01 A 2025-2
Luisito L. Agustin

20260219

Project 2
Fourier Transform

Project 2
Fourier Transform

Due: start of class time, March 19, 2026

Progress Report 1	Feb. 26
Progress Report 2	Mar. 5
Progress Report 3	Mar. 12
Early Work Submission	Mar. 16

DSP Project

Fourier Transform

Luisito L. Agustin
2023 02 22

General Specifications

Develop an application in C++ for computing the Discrete Fourier Transform (DFT) of a finite-duration signal.

Detailed Specifications

The value of the Fourier Transform at a digital frequency ω is given by

$$X(\omega)=\sum_{n=0}^{L-1} x(n)e^{-j\omega n}$$

given a signal consisting of *L* points. The Fourier Transform becomes a Discrete Fourier Transform when evaluated at a discrete set of values of ω.

The application will be invoked in the following manner from the command line:

```
>dft signal-file sampling-rate start-freq end-freq nSteps (logfile)
```

where

`dft`	is the assumed name of the executable,
`signal-file`	is the name of the signal file,
`sampling-rate`	is the sampling rate of the signal in Hz,
`start-freq`	is the start frequency in Hz (the lower limit of the range of analog frequencies at which the Fourier Transform would be evaluated), the application converts this to digital frequency
`end-freq`	the end frequency in Hz, the application converts this to digital frequency
`nSteps`	is the number of steps to be taken in going from the start frequency to the end frequency (the number of points at which the DFT would be evaluated is `nSteps+1`), and
`logfile`	is the name of the file to which all outputs will be written to.

e.g.

```
>dft data.csv 8000 430 450 40 log1.text
```

would invoke `dft` on the data in `data.csv`. The DFT conputation will assume the data were sampled at 8000 Hz in computing the DFT in increments of 0.5 Hz from 430 Hz to 450 Hz, so that there would be 40 steps of size 0.5 Hz each going from 430 Hz to 450 Hz, a total of 41 evaluations of the DFT including the evaluation at the starting point 430 Hz. All output will be appended to `log1.text`.

The Signal File Format is specified in a separate document. In computing the Fourier Transform, the starting index determined from the signal file should be ignored and the signal is assumed to start at index 0.

Implement all *Recommendations for Handling Signal Files* in the *Signal File Format* specification.

Provide feedback on the result of parsing the signal file. If a valid signal was extracted, issue a one-line statement stating both the duration of the signal extracted and the filename from which the signal was extracted. E.g. "Signal of duration 16 extracted from test.signal."

Implement a function for computing the DFT. The function does all the processing needed to compute the DFT and store it in memory, including allocating the needed memory. This function does not read any data from files nor write any results to files. All data must be in arrays at this point. Vectors are not acceptable for use with this function.

Possible prototypes (do not deviate significantly from these):

```
void computeDFT(
  double * xData, int xDuration,
  double samplingFreq, double startFreq, double endFreq, int nSteps,
  double ** realPart, double ** imagPart,
  double ** magnitude, double ** phase);

void computeDFT(
  double * xData, int xDuration,
  double samplingFreq, double startFreq, double endFreq, int nSteps,
  double ** realPart, double ** imagPart);
// The magnitude and phase could be computed separately from the
// real and imaginary parts produced by the function
```

The function does not call on other functions except possibly those available thru the C++ standard library. The function should not get any input from a file or from the console or send any output to a file or the console.

The name of the log file defaults to `dftlog.txt` if not specified.

The application creates the log file if it doesn't exist yet. If the file exists, the application appends to the file. The previous contents of the existing file should not be modified.

Provide feedback on the filename used for the log file.

Two sets of outputs will be written to the log file.

The first set of outputs will consist of the header "frequency \t real part \t imaginary part \n" followed by the DFT results with each result line consisting of a frequency in Hz followed by the real part of the DFT at that frequency followed by the imaginary part of the DFT at that frequency . Use `" \t"` to separate entries in one row.

The second set of outputs will consist of the header "frequency \t magnitude \t phase \n" followed by the DFT results with each result line consisting of a frequency in Hz followed by the magnitude of the DFT at that frequency followed by the phase of the DFT in degrees at that frequency. Use `" \t"` to separate entries in one row.

Frequencies provided by the user and reported in the log file are in Hz. The application must convert between analog frequencies in Hz and digital frequencies in radians whenever needed. The application must also convert between phase in degrees and phase in radians whenever needed.

Do not modify any filename provided by the user. Do not assume, modify, require, nor add any extension. Use whatever filename is provided.

If, and only if, `nSteps < 10`, the output sent to the log file is sent to the console as well.

Reference

[Proakis] (book) John Proakis and Dimitris Manolakis: Digital Signal Processing, 4th ed, 2007.

Revision History - Fourier Transform

2020 12 03:	first version
2021 10 21:	purely command line application
2021 03 22:	signal file now released separately, improved clarity of instructions
2023 02 22:	required DFT function; required output to console for small cases; required feedback on signal read, output file;
2024 09 19	required self-contained Fourier Transform function

Variable Specifications

Sections marked "(variable)" and specifications and instructions from this point onwards may vary from one use of the document to the next. These are not tracked as revisions to the project specifications.

Group Work

The implementation of this project in C/C++ may be worked on by groups. Documentation, testing, and all other parts of the project must be done individually.

It is expected that C++ code submitted by groupmates may be similar or exactly the same. Points earned for the entire project will be subject to penalties for unauthorized collaboration if documentation submitted by at least two students contain at least one distinctive phrase, image, or item in common.

Basic Test

signal file

x.signal

```
0       -0.130733614
        -1.238039900
        -1.440074782
        -0.589107641
        0.692622920
        1.467898298
        1.169826725
        0.016362137
        -1.149066665
        -1.474282487
        -0.721483153
        0.558874353
        1.430575426
        1.256220532
        0.163300312
        -1.049027290
        -1.494292047
        -0.846910389
        0.419743521
        1.379475331
        1.330516250
        0.308665816
        -0.938885208
        -1.499910758
        -0.964181415
        0.276570329
        1.315090134
        1.391998368
        0.451058699
        -0.819701149
        -1.491084507
        -1.072166846
```

command line

```
dft x.signal 32 4 8 8
```
(sampling rate = 32, 4 Hz to 8 Hz, 8 steps)

resulting DFT

frequency (Hz)	real part	imaginary part
4	-16.016414	-1.271327
4.5	-2.091738	23.908673
5	14.523652	1.379051
5.5	0.000000	0.000000
6	4.458294	0.479847
6.5	0.000000	0.000000
7	2.498667	0.291875
7.5	0.000000	0.000000
8	1.690067	0.206077

frequency (Hz)	magnitude	phase (degrees)
4	16.066792	-175.461577
4.5	24.000000	95.000000
5	14.588977	5.424093
5.5	0.000000	-62.499637
6	4.484042	6.143102
6.5	0.000000	-0.912811
7	2.515656	6.662659
7.5	0.000000	7.360520
8	1.702584	6.951986

Submission

Do the basic test with exactly the same data provided. Obtain a screenshot of the application with the results of the basic test on the console. Since the number of steps in the basic test is less than 10, the output sent to the log file must be sent to the console as well. Take a screenshot of the console at this point with the results onscreen.

Implementation and Testing Environment Report

Provide details of at least one platform used in implementing and testing the project.

* name and version of compiler
* name and version of integrated development environment (IDE)
* operating system name/distribution and version

screenshot of compilation

The screenshot must show the files involved and show feedback from the IDE reporting successful compilation and linking; if the project is compiled on the command line, the screenshot must show the commands issued; if a makefile was used, include a copy of the makefile in addition to the screenshot.

screenshot of program run on the console

Copy the executable to a location whose path contains a recognizable version of your name. If working with a group, make sure that the path does not contain any version of the names of any groupmates. Create a new folder named after yourself as needed. Run the executable by typing its name at the command prompt, and take a screenshot as the executable starts. Do not run the application from within an IDE.

Code locator

Fill in the table locating the specified code items. Fill in this table after the code has been finalized. Inaccurate details will be penalized.

code items	file name	line numbers
DFT function		
declaration of double * that are passed to the DFT function		
separate processing of first line of signal file		
input validation of integers in processing signal files		
input validation of floating point numbers in processing signal files		

Do a self-evaluation using the project evaluation table and evaluation procedures provided. Fill in your scores and add your points correctly.

Place the screenshot of the basic test, the code locator table, and the self-evaluation table together in a single document and export this document to pdf.

Submit a single zip file containing
* C++ implementation files and header files for the project
* required documentation

Do NOT include executables in your submission.

Project Evaluation - Fourier Transform		
Item	Points	Rubrics
early work	8/8	(all or nothing) 8: at most 10% of the code in the final implementation differs from that in early work submission
implementation and testing environment report	4/4	4 - all instructions followed correctly
Basic Test	50/50	50 - correctly computes the DFT of basic test data
	6/6	6 - screenshot submitted showing the correctly computed DFT of basic test data as it appears on the console output
generality of DFT	6/6	6 - the DFT is computed correctly and efficiently at all times
DFT function	6/6	6 - the DFT is computed correctly and efficiently at all times, by a function as specified
processing of signal files	4/4	4 - The file is opened only once and the first line of the signal file is processed correctly and separately from code processing the rest of the signal file
input validation	4/4	4 - integers and floating point numbers are fully validated when signal files are processed
command line	4/4	4 - application processes command line arguments correctly at all times
application feedback	4/4	4 - application produces required outputs and provides appropriate feedback at all times, as specified
self-evaluation	4/4	4 - self-evaluation accurate (or evaluating this item leads to an error)
total	100/100	

Evaluation Procedures

Special Cases

Score for the entire project is 0 if
* no source files are submitted
* the source files, as submitted, will not compile as a C++ project
* the implementation uses resources that are not allowed

early work

Early work credits are based on the final code submitted, assuming the final and early work code use the same style. The total number of lines in the final code that are not in the early work version, or that were modified from the early work version, other than formatting changes that do not modify compiled code, must not be more than 10% of the number of lines in the final code, excluding blank lines.

basic test

The correct result must have been computed from the provided test data, not hardcoded in any manner. Score 0 if there is any anomaly in the basic test.

screenshot

must show the command line and the complete results produced on the console; values must be the same as expected. Score 0 if the complete results are not all visible, or values are not the same as expected, or screenshot shows the contents of the output file instead of the console output. Score 0 if there is any anomaly in the basic test.

generality of DFT

Score 0 if basic test fails.
Score 0 if limits are placed on the amount of data.
Stress test application.
Change command line parameters in the basic test. Check for correctness of corresponding outputs produced. Generate random data. Compare results with those from online tools.
Change the test data and check results.
Do more tests if in doubt.
Check for suspicious code. Verify that suspicious code leads to errors.
Score 0 if any error manifests.

DFT function

Score 0 if generality test fails. Score 0 if code not put into a function.
Check that location of function is reported correctly. Check that parameters are correct.
Scale points by half if related code elements are not located accurately.

processing of signal files

Score 0 if signal files are opened more than once.

Score 0 if the first line is not processed correctly or not processed separately from the rest of the signal file. Check that a "1" as the first token on the first line is interpreted differently from a "1." .

Check that data extracted are put into dynamically allocated arrays of doubles. Check that contents of vectors used while extracting data from signal files are copied into arrays. Do more tests if in doubt. Insert comments. Insert blank lines. Test with invalid files. Score 0 if any deficiency manifests. Scale points by half if related code elements are not located accurately.

input validation

Score 0 if not done.

Make a signal value invalid by adding an invalid character to a signal value (e.g. -2a). Check that this is treated as a comment and the signal extracted shorter in duration than if there were no invalid character. Do more tests if in doubt. Score 0 if any deficiency manifests. Scale points by half if related code elements are not located accurately.

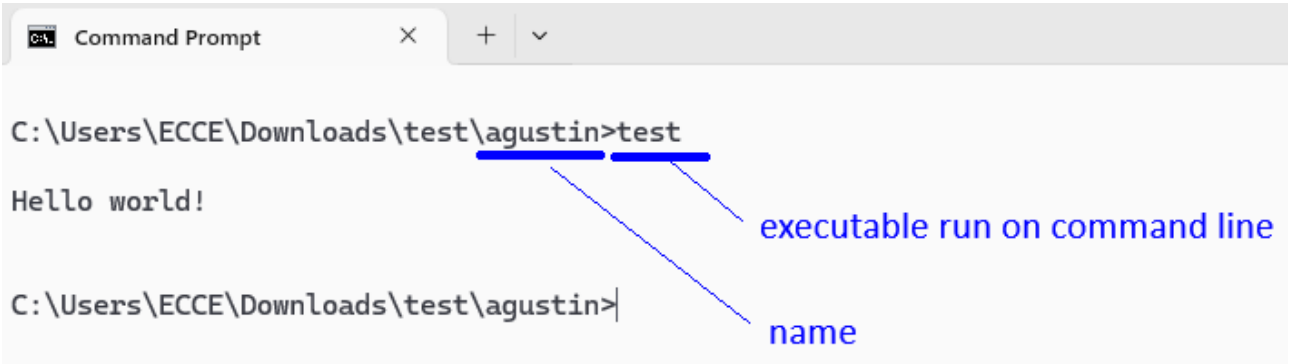
command line

Check that application accepts filenames specified on the command line. Change filename extensions and check that the application accepts them (e.g. remove the .txt, change .txt to .exe or .jpg, etc). Score 0 if there are deficiencies.

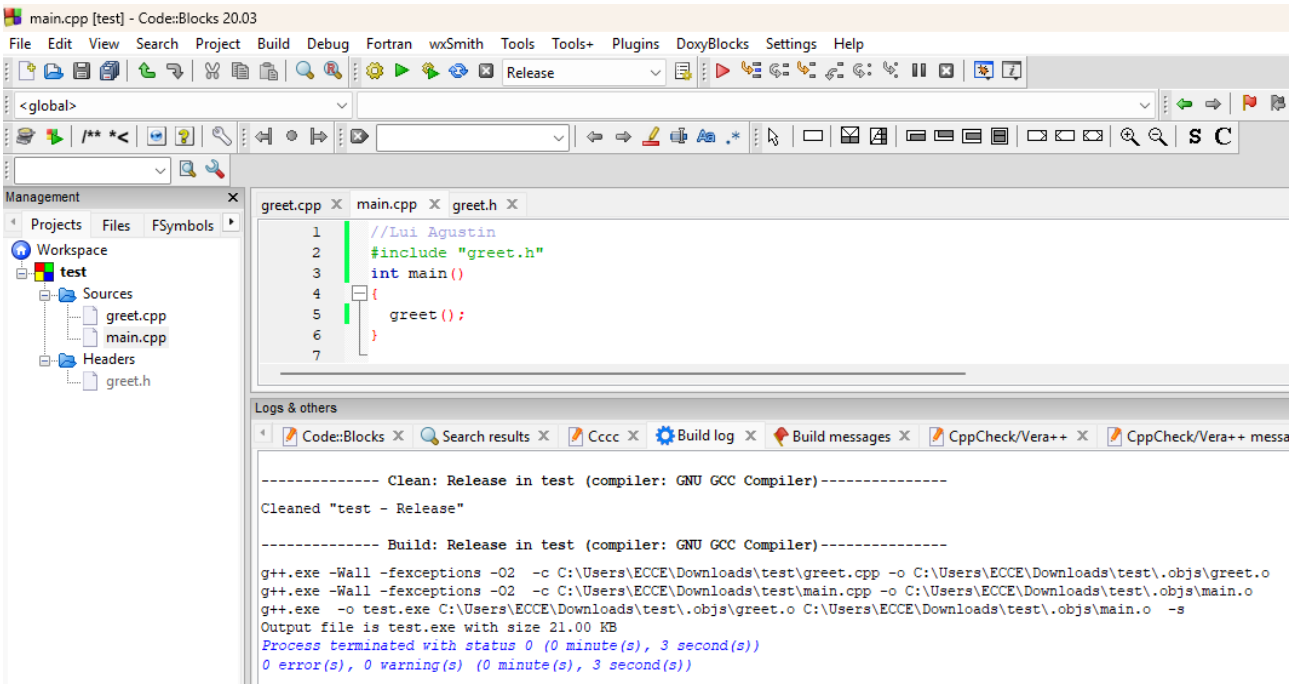
application feedback

Score 0 if application does not produce required output or provide feedback in some instances, or feedback from application misses some details. Score 0 if application ever crashes, goes into an infinite loop, or behaves erratically.

Implementation and Testing Environment (4 pts)
1 - all details of implementation platform are provided
1 - screenshot of compilation: all instructions are followed correctly
1 - screenshot of program run on the console: all instructions are followed correctly
1 - all filenames appearing in the screenshots are consistent with files in the project submission



sample screenshot - executable test.exe run by typing its name at the prompt from a path with name of author

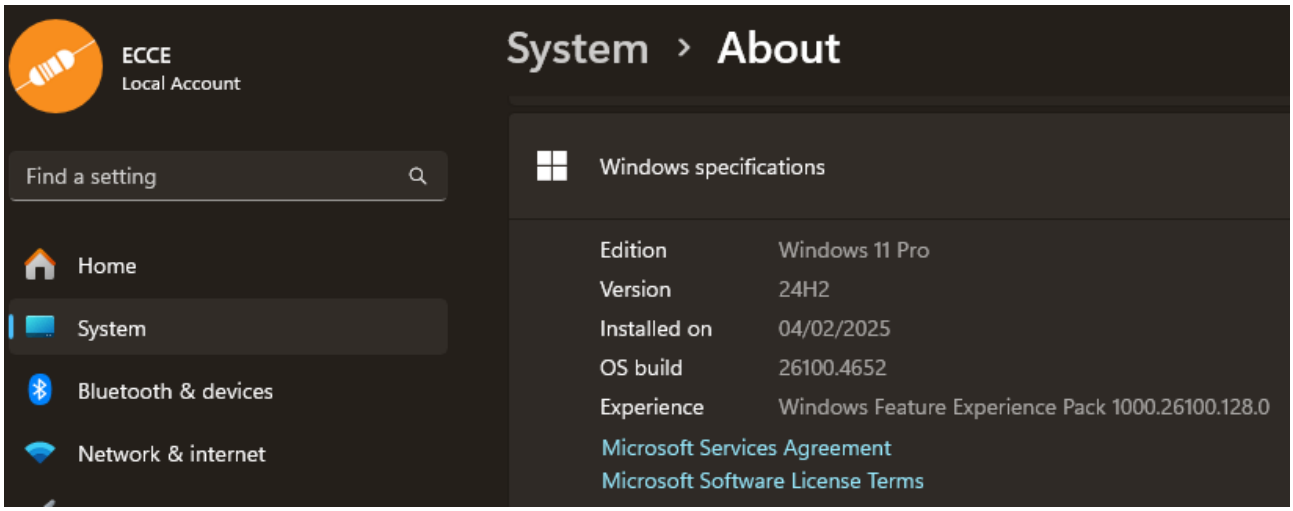


sample screenshot - compilation on Code::Blocks

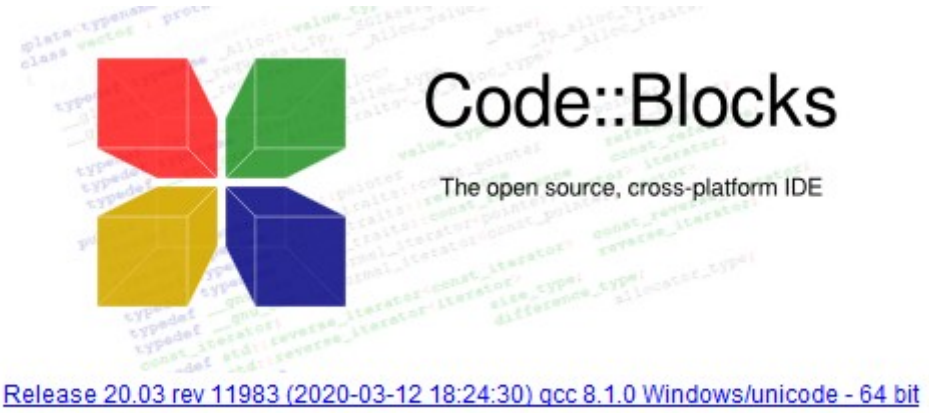
```
C:\Users\ECCE\codeblocks\MinGW\bin>g++ -v
Using built-in specs.
COLLECT_GCC=g++
COLLECT_LTO_WRAPPER=C:/Users/ECCE/codeblocks/MinGW/bin/../libexec/gcc/x86_64-w64-mingw32/8.1.0/lto-wrapper.exe
Target: x86_64-w64-mingw32
Configured with: ../../../src/gcc-8.1.0/configure --host=x86_64-w64-mingw32 --build=x86_64-w64-mingw32 --target=x86_64-w64-mingw32 --prefix=/mingw64
 --with-sysroot=/c/mingw810/x86_64-810-posix-seh-rt_v6-rev0/mingw64 --enable-shared --enable-static --disable-multilib --enable-languages=c,c++,fort
ran,lto --enable-libstdcxx-time=yes --enable-threads=posix --enable-libgomp --enable-libatomic --enable-lto --enable-graphite --enable-checking=rele
ase --enable-fully-dynamic-string --enable-version-specific-runtime-libs --disable-libstdcxx-pch --disable-libstdcxx-debug --enable-bootstrap --disa
ble-rpath --disable-win32-registry --disable-nls --disable-werror --disable-symvers --with-gnu-as --with-gnu-ld --with-arch=nocona --with-tune=core2
 --with-libiconv --with-system-zlib --with-gmp=/c/mingw810/prerequisites/x86_64-w64-mingw32-static --with-mpfr=/c/mingw810/prerequisites/x86_64-w64-
mingw32-static --with-mpc=/c/mingw810/prerequisites/x86_64-w64-mingw32-static --with-isl=/c/mingw810/prerequisites/x86_64-w64-mingw32-static --with-
pkgversion='x86_64-posix-seh-rev0, Built by MinGW-W64 project' --with-bugurl=https://sourceforge.net/projects/mingw-w64 CFLAGS='-O2 -pipe -fno-ident
 -I/c/mingw810/x86_64-810-posix-seh-rt_v6-rev0/mingw64/opt/include -I/c/mingw810/prerequisites/x86_64-zlib-static/include -I/c/mingw810/prerequisite
s/x86_64-w64-mingw32-static/include' CXXFLAGS='-O2 -pipe -fno-ident -I/c/mingw810/x86_64-810-posix-seh-rt_v6-rev0/mingw64/opt/include -I/c/mingw810/
prerequisites/x86_64-zlib-static/include -I/c/mingw810/prerequisites/x86_64-w64-mingw32-static/include' CPPFLAGS=' -I/c/mingw810/x86_64-810-posix-se
h-rt_v6-rev0/mingw64/opt/include -I/c/mingw810/prerequisites/x86_64-zlib-static/include -I/c/mingw810/prerequisites/x86_64-w64-mingw32-static/includ
e' LDFLAGS='-pipe -fno-ident -L/c/mingw810/x86_64-810-posix-seh-rt_v6-rev0/mingw64/opt/lib -L/c/mingw810/prerequisites/x86_64-zlib-static/lib -L/c/m
ingw810/prerequisites/x86_64-w64-mingw32-static/lib '
Thread model: posix
gcc version 8.1.0 (x86_64-posix-seh-rev0, Built by MinGW-W64 project)

C:\Users\ECCE\codeblocks\MinGW\bin>
```

compiler - gcc version 8.1.0 (x86_64-posix-seh-rev0, Built by MinGW-W64 project)

ENGG 151.01 - Signals, Spectra, and Signal Processing, Lecture
ENGG 151.01 A 2025-2
Luisito L. Agustin



operating system - Windows 11 Pro, 24H2, build 26100.4652



IDE -Code::Blocks 20.03
rev 11983 (2020-03-12 18:24:30) gcc 8.1.0 Windows/unicode - 64 bit

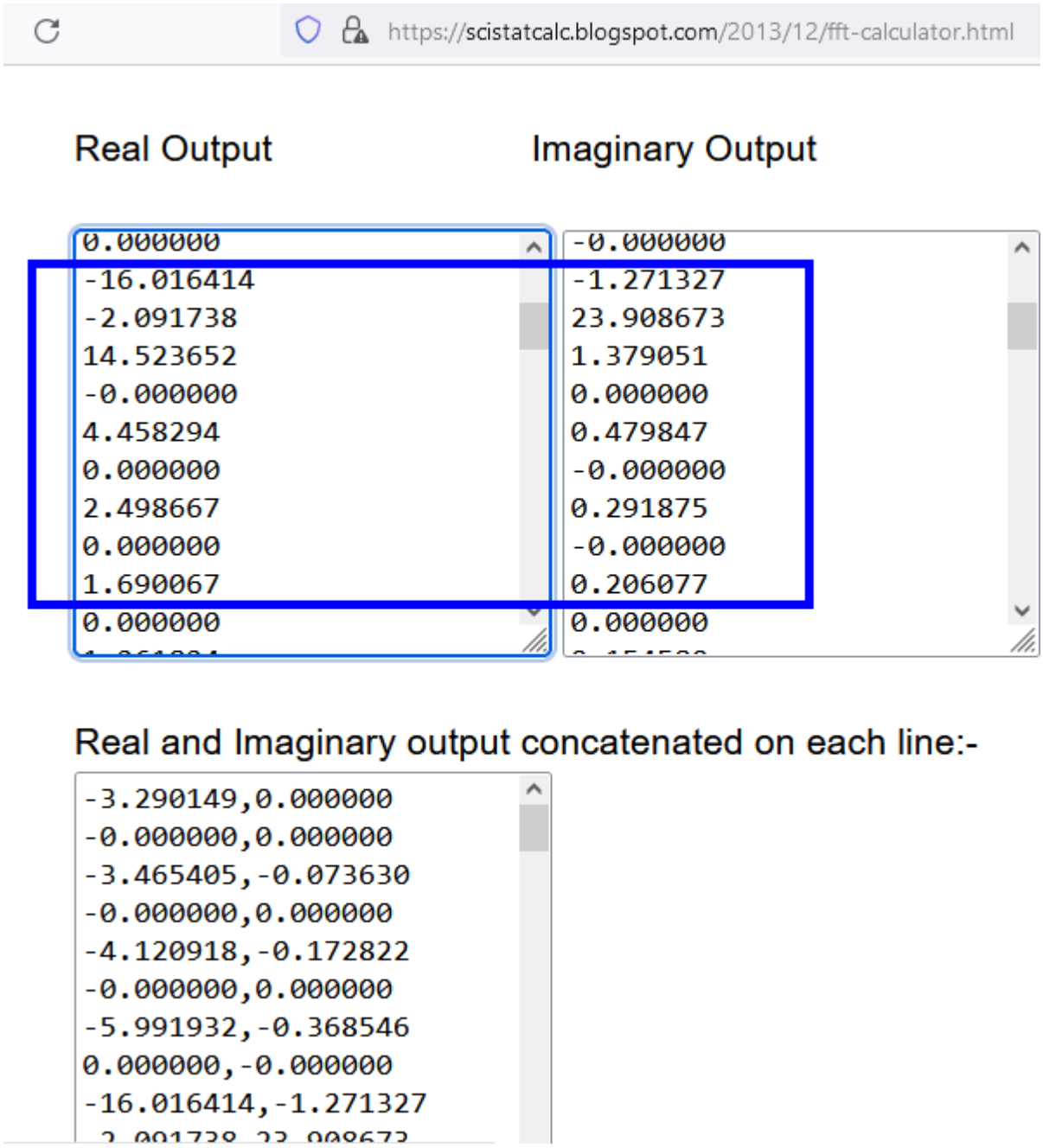
self-evaluation
0 - not done
2 - self-evaluation score differs from project score by more than 10 points or scores not tallied properly
4 - self-evaluation accurate (or evaluating this item leads to an error)

Notes

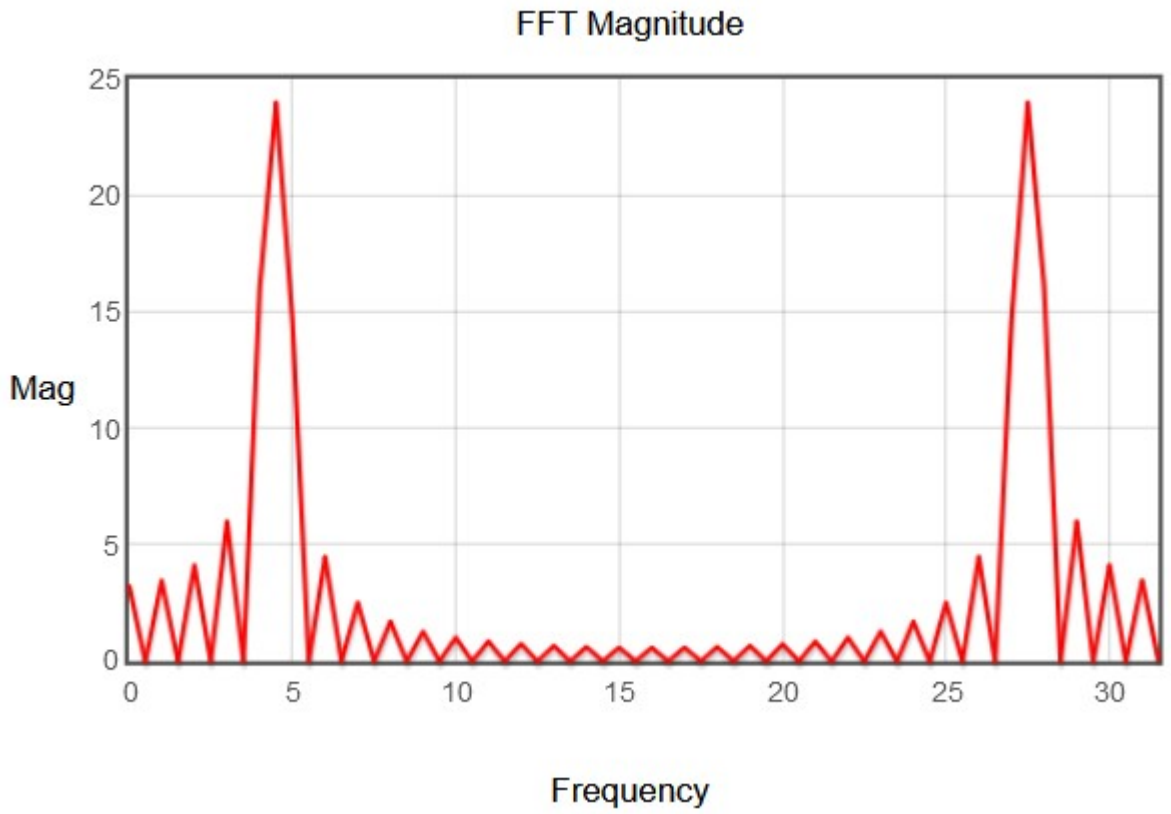
Some signal files for testing may easily be obtained from previous exercises that have been done.

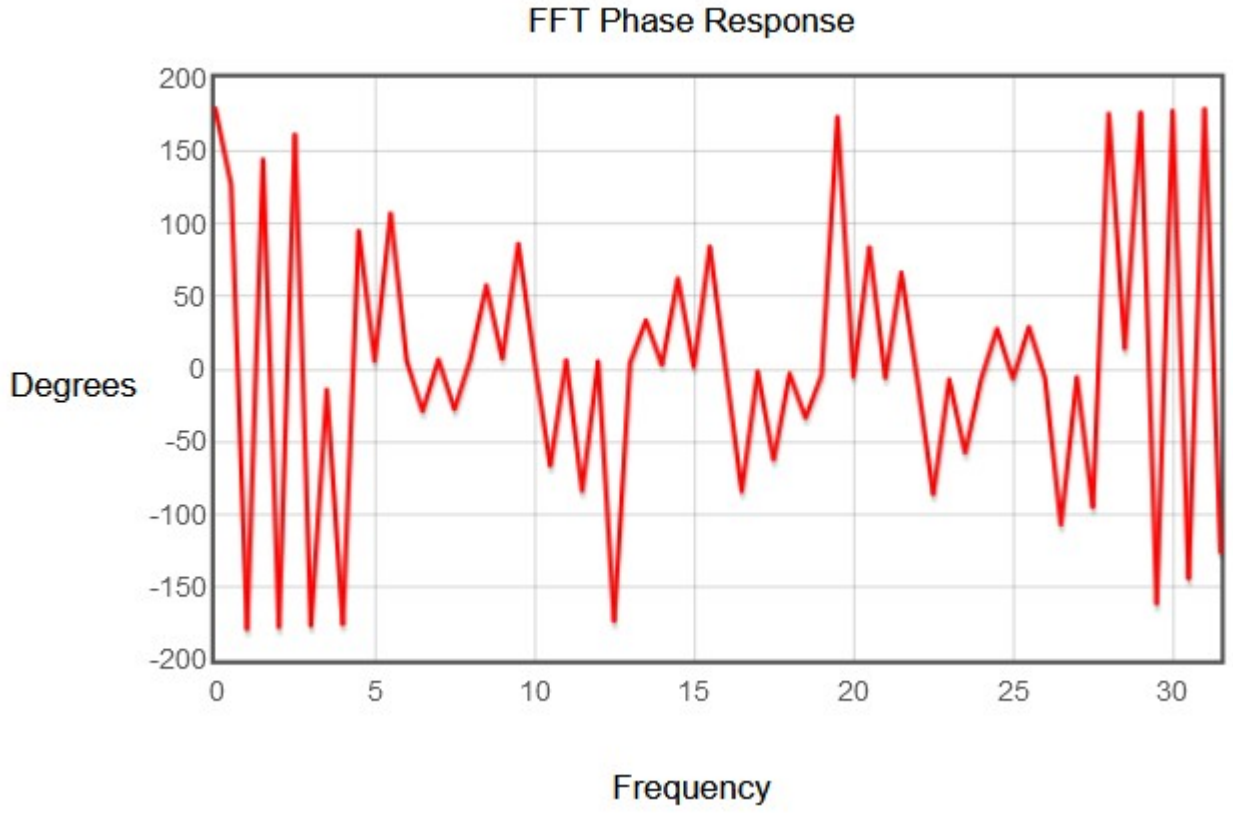
It is sufficient to always open a log file for appending. The file will be created if it doesn't exist yet.

Verification of basic test
using https://scistatcalc.blogspot.com/2013/12/fft-calculator.html



The computed phase is not significant when the magnitude is 0 or very small.


FFT Magnitude
Mag
25
20
15
10
5
0
0
5
10
15
20
25
30
Frequency


FFT Phase Response
Degrees
200
150
100
50
0
-50
-100
-150
-200
0
5
10
15
20
25
30
Frequency

Appendix: Signal File Format

See next page.

Signal File Format

Luisito L. Agustin
2023 01 18

Introduction

This document specifies a text file format for representing finite-duration signals. In this file format, signal values are expected to be floating point values in general.

All signal values of a finite-duration signal, $x(n)$, are zero except for values of the integer index n in a finite range from $n = start$ to $n = end$ where $start < end$. The signal is then said to have a *duration* given by

$$duration = end - start + 1 .$$

A signal file is a text file regardless of whatever extension may be used in the filename.

Format

The format for a signal file is as follows:

[optional starting index] **signal value** optional comments
signal value optional comments
.
.
.
signal value optional comments
optional comments

The finite set of signal values must be the first or leftmost tokens in consecutive lines of the text file starting with the first line of the file, except that the first signal value (on the first line) may be preceded by an integer indicating the index $n = start$ of the signal value that follows on the line. Each signal value may be preceded by any amount of white space on each line. Only the starting index needs to be specified. It is understood that succeeding signal values correspond to successive increments of the index.

Any text that follows a signal value is considered a comment and may be ignored. A line which does not start with a valid floating point value is a comment. A signal file does not encode any valid signal if the first line does not start with a valid integer or floating point number.

The duration of the signal is determined from the number of consecutive lines from which a valid signal value can be extracted. The signal values either terminate when the end of a file is reached or when the next line extracted from the file does not contain a valid signal value as its first token.

If the optional starting index is absent, the implied starting index is 0. That is, if the first item on the first line is not a valid integer but is a valid floating point number, then the start index of the signal is zero and the signal value at $n = 0$ is the floating point number extracted from the first line.

If the first line contains a single numerical value, that numerical value must be a signal value and the implied start index is 0.

Examples

```
-4 2 This is a comment to be ignored
-1 Comment: the -4 on the first line means the signal starts with x(-4) = 2
3
7
1
2
-3
This line does not start with a floating point value.
The parser will conclude that -3 is the end of the finite duration signal.
Everything that comes after may be ignored.
This signal is the sequence x(n) from Example 2.6.1 in
John Proakis and Dimitris Manolakis: Digital Signal Processing, 4th ed, 2007.
x(-4) = 2
x(-3) = -1
x(-2) = 3
x(-1) = 7
x(0) = 1
x(1) = 2
x(2) =-3

start = -4
end = 2
duration = 7  ( = 2 - (-4) + 1 )
```

Without comments, the signal file would simply contain:

```
-4 2
-1
3
7
1
2
-3
```

Comments do not change the signal being represented.

Notes

Some confusion may be possible in the first line if there is no starting index (implied start = 0) but the floating point signal value appears as an integer. The signal value might be incorrectly interpreted as the start index.

Recommendations for Handling Signal Files

The extraction operator (>>) in C++ skips white space and terminates processing on any white space character. It would skip whatever leading white space there might be before the signal values.

The first line of a signal file needs to be processed differently from the rest of a signal file. It should not be processed as part of a loop.

There's no limit to how large a signal file could be. Applications should not impose any limit. Nothing in the file format specifies the duration of the signal. A C++ vector should be used to store the signal values obtained until the last signal value is extracted from the file, at which point the duration would be known. Once the duration is known, an array must be dynamically allocated and the contents of the vector copied to the dynamically allocated array. Any processing to be done must be done on the dynamically allocated array. Processing arrays is faster than processing vectors.

Whether a signal file is valid depends on its contents, never on the extension used for the filename. While a .txt extension may be common for text files, applications that process signal files should neither expect nor require a .txt extension. They should work with whatever filename is provided, as is, without any modification.

A signal file should be opened only once, and its lines processed one at a time from start to finish without any need to process any line twice.

Suggested C++ Signal Class API

```
class engg151Signal
{
  public:
    engg151Signal ();
    // default constructor
    // suggested default: one-element signal with value 0.0, start index 0

    engg151Signal ( double * x, int start, int duration );

    bool importSignalFromFile ( string filename);
    // returns true if a valid signal was actuallly obtained from filename
    // returns false otherwise

    bool exportSignalToFile ( string filename );
    // returns true if the signal was successfully exported to a file
    // returns false otherwise

    int start();
    int end();
    int duration();
    // as implied by the names,
    // return the start index, end index, and duration of the signal
    // respectively

    double * data();
    // returns a pointer to an array of double containing the signal values
}

engg151Signal normalizedXCorr ( engg151Signal x, engg151Signal y);
// computes the normalized crosscorrelation of x crosscorrelated with y
// and returns the normalized crosscorrelation as a signal object.
// Note that the crosscorrelation is not commutative.
// This does not need to be a class member.
```

Revision History - Signal File Format

2021 02 17	first version
2023 01 18	revisions for clarity, recommendations