# ECE 5630 Fast Convolution

Aaron Pettit

12/6/2018

# Introduction

This lab compares the algorithms of a time domain filter and fast convolution using the FFT and the overlap-save or overlap-add method. Plots and some characteristics are given for each of the algorithms

# Filter

FIR filter specification

* 256 coefficients
* Lowpass filter
* Passband of 300 Hz with unit gain
* Stopband of 400 Hz
* Sample rate of 11,025 Hz

The Ideal filter cuts off at exactly 300 Hz with no transition band.

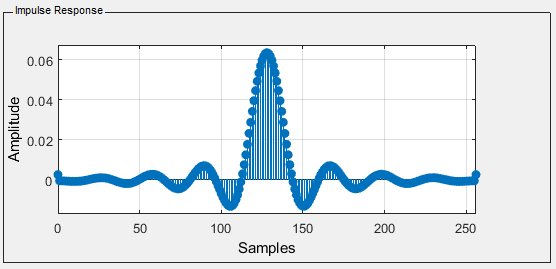


Figure h[n] Impulse Response

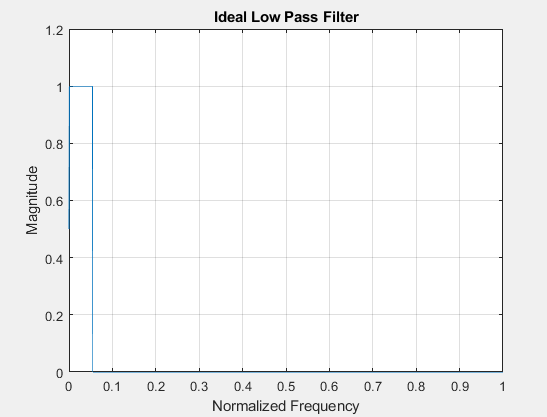


Figure Desired H(s) Frequency Response

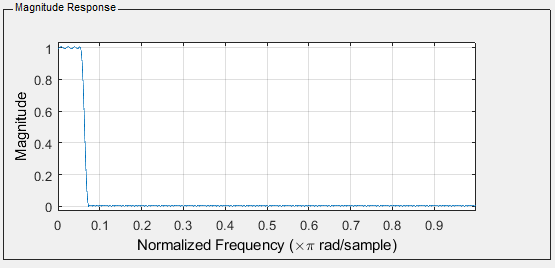


Figure Actual H(w) Magnitude Response

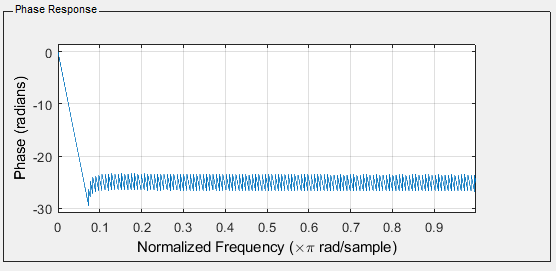


Figure Actual H(w) Phase Response

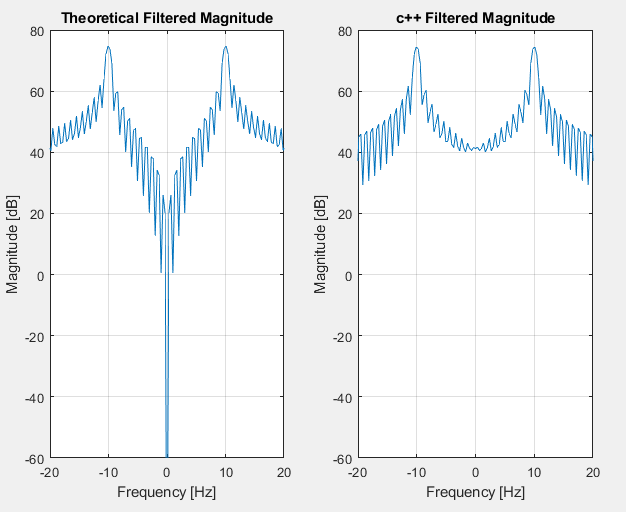
# Time Domain Filter

Adds and multiplies per output sample

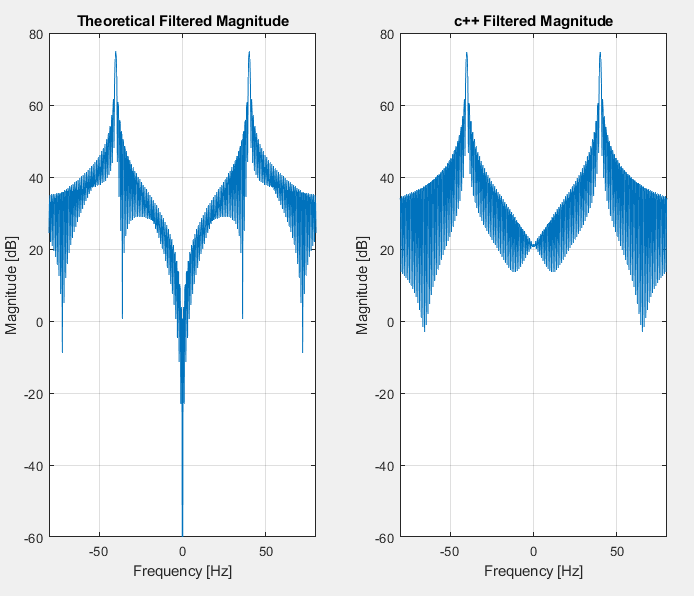
Adds: Length of Filter-1 -> 256

Multiplies: Length of Filter -> 257

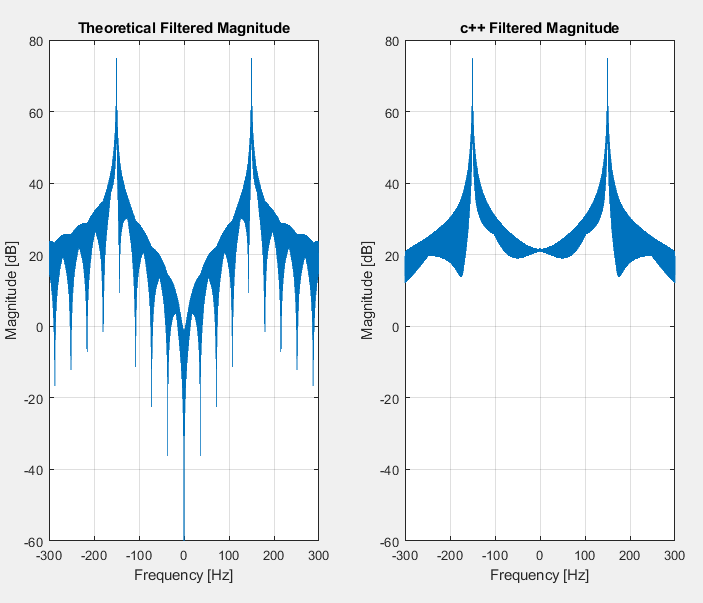
## Sinusoid f=10 Hz



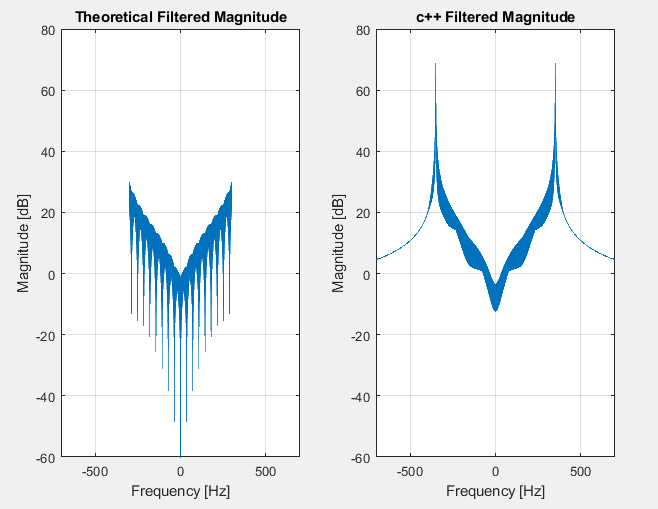
## Sinusoid f=40 Hz



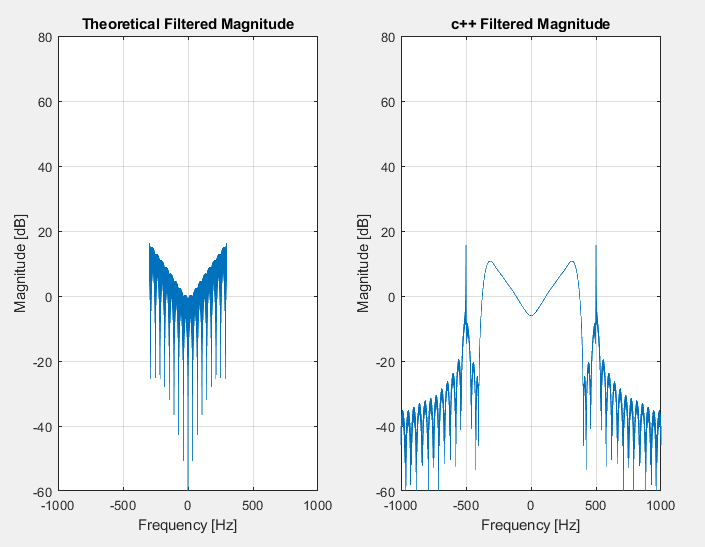
## Sinusoid f=150 Hz



## Sinusoid f=350 Hz



## Sinusoid f=500 Hz



## Max Theoretical vs Actual Amplitudes

|  |  |  |
| --- | --- | --- |
| Frequency (Hz) | Theoretical Amplitude (dB) | Actual Amplitude (dB) |
| 10 | 74.7416 | 74.5218 |
| 40 | 74.8102 | 74.7655 |
| 150 | 74.7780 | 74.7024 |
| 350 | 29.9919 | 68.6562 |
| 500 | 16.3423 | 15.6156 |

# FFT Filter

The method used in the c++ program was overlap-save. NFFT = 2^10 used in the program, P = 256 is the length of the filter.

Adds and multiplies for the fft (assuming all radix-2) is N\*log2(N) for N data points.

Output comes in batches so it is possible to have a fractional number of operation for a single input, but is an integer when taking into account the batch.

## Overlap-save

Adds: (2\*fft + 2\*P[zero pad x and add y])/N -> 2\*log2(N) +2\*P/N -> 20+512/1024 =20.5

Multiplies: (2\*fft + N)/N -> 2\*log2(N) + 1 -> 21

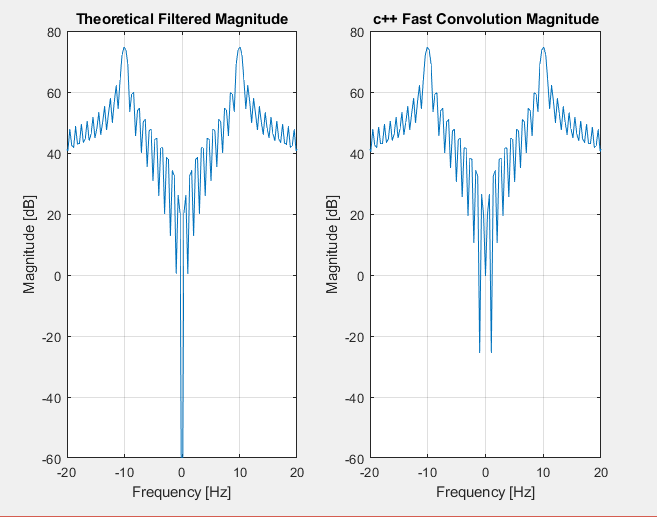
Overlap-add

Adds: added P points from the previous x[n] to the current x[n]

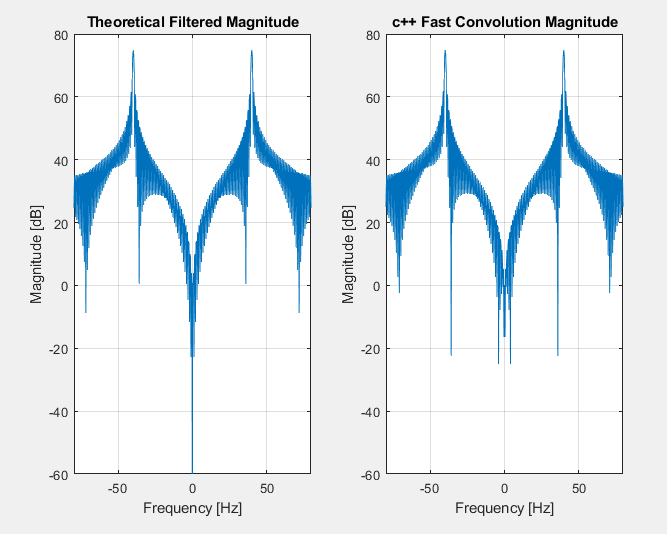
(2\*fft+P)/(N-P) -> (20\*1024+256)/(1024-256) = 27

Multiplies: (2\*fft+N)/(N-P) -> (20\*1024+1024)/(1024-256) = 28

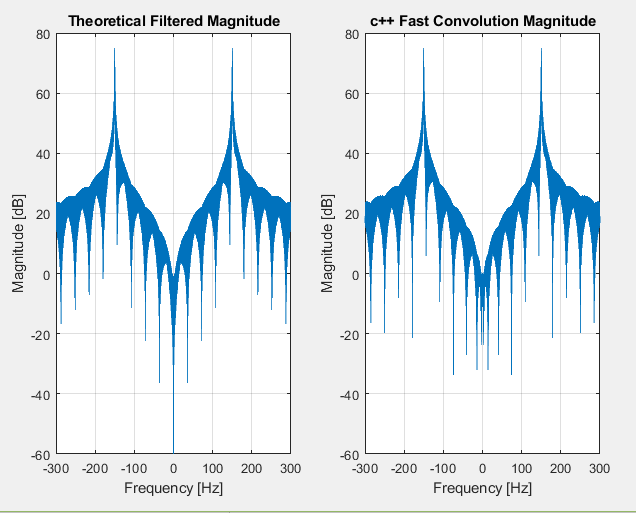
## Sinusoid f=10 Hz



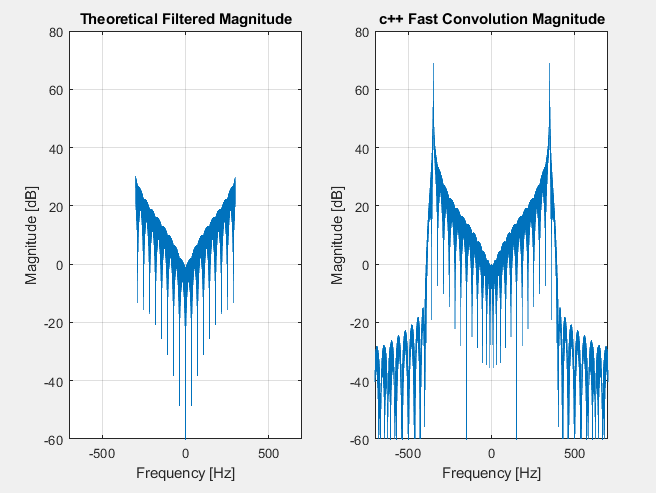
## Sinusoid f=40 Hz



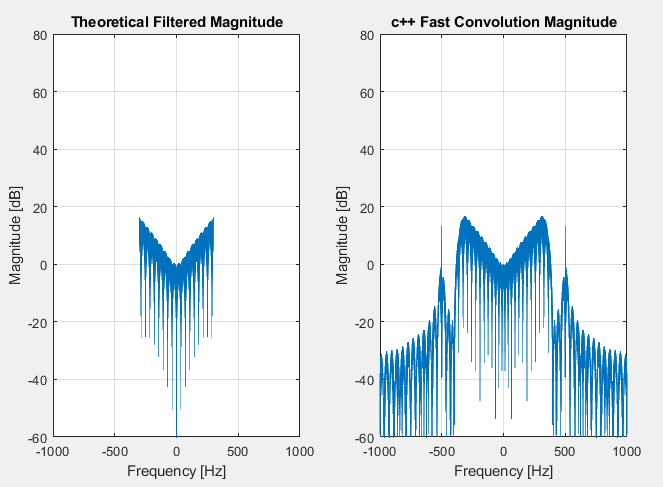
## Sinusoid f=150 Hz



## Sinusoid f=350 Hz



## Sinusoid f=500 Hz



## Max Theoretical vs Actual Amplitudes

|  |  |  |
| --- | --- | --- |
| Frequency (Hz) | Theoretical Amplitude (dB) | Actual Amplitude (dB) |
| 10 | 74.7416 | 74.7040 |
| 40 | 74.8102 | 74.8588 |
| 150 | 74.7780 | 74.8042 |
| 350 | 29.9919 | 68.7622 |
| 500 | 16.3423 | 16.7502 |

# Audio Clip

Filter Time: 93.420 seconds

Fast Convolution Time: 2.999 seconds

Fast Convolution is approximately 31.15 times faster than filtering in the time domain.

The processed sound signal has the low frequency notes (bass) making up most of the sound with the flute just barely heard. This means the high frequencies were mostly filtered out, but they still exist but at a smaller amplitude.

# Conclusion

Fast convolution is much faster than time domain filtering at approximately 31.15 times as fast. Overlap-add takes in less input samples at a time than overlap-save but uses less computations. Overlap-save throws out some of its output data, but the input doesn’t need to be zero padded each time so larger segments of input can be calculated at once.

# Appendix

## Main.cpp

#include "Convolution.h"

#include "Timer.h"

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

{

if (argc < 3)

{

printf("Wrong number of arguments");

exit(EXIT\_FAILURE);

}

Convolution filter(argv[1], argv[2]);

#ifdef \_FFT

Timer::Start();

filter.FastConvolution();

Timer::Stop();

printf("Fast Convolution using overlap-save duration: %0.3f seconds\n", Timer::GetDuration());

#else

Timer::Start();

filter.Filter();

Timer::Stop();

printf("Filter duration: %0.3f seconds\n", Timer::GetDuration());

#endif

system("pause");

return 0;

}

## Timer.h

#pragma once

#include <chrono>

class Timer

{

public:

static void Start()

{

begin = Clock::now();

}

static void Stop()

{

end = Clock::now();

}

enum Time { seconds, milliseconds };

static double GetDuration(const Time baseTime = seconds)

{

double time;

if (baseTime == seconds)

{

time = static\_cast<double>(std::chrono::duration\_cast<std::chrono::seconds>(end - begin).count());

time += static\_cast<double>(std::chrono::duration\_cast<std::chrono::milliseconds>(end - begin).count()) / 1000.0 - time;

}

else

{

time = static\_cast<double>(std::chrono::duration\_cast<std::chrono::milliseconds>(end - begin).count());

}

return time;

}

private:

Timer() = default;

typedef std::chrono::high\_resolution\_clock Clock;

std::chrono::time\_point<std::chrono::steady\_clock> static begin;

std::chrono::time\_point<std::chrono::steady\_clock> static end;

};

std::chrono::time\_point<std::chrono::steady\_clock> Timer::begin;

std::chrono::time\_point<std::chrono::steady\_clock> Timer::end;

## FileRead.h

#pragma once

#include <cstdio>

#include<string>

#include "dspFileHeader.h"

#include <fstream>

using namespace std;

class FileRead

{

public:

FileRead(const string& inFile, const string& outFile) : eof(false)

{

in.open(inFile, ios::binary);

out.open(outFile, ios::binary);

if (!in.is\_open())

{

printf("Error opening input file\n");

exit(EXIT\_FAILURE);

}

if (!out.is\_open())

{

printf("error opening output file\n");

exit(EXIT\_FAILURE);

}

in.read(reinterpret\_cast<char\*>(&inFileHeader), sizeof(dsp\_file\_header));

outFileHeader = inFileHeader;

outFileHeader.dim0 = 0;

dataIndex = in.tellg();

//write out initial header, will be updated in the destructor

out.write(reinterpret\_cast<char\*>(&outFileHeader), sizeof(dsp\_file\_header));

}

~FileRead()

{

//go to beginning and update header

out.seekp(0);

out.write(reinterpret\_cast<char\*>(&outFileHeader), sizeof(dsp\_file\_header));

in.close();

out.close();

}

float GetValue()

{

float retVal;

in.read(reinterpret\_cast<char\*>(&retVal), sizeof(float));

eof = in.eof();

//last value read in is garbage for some reason

if (eof) retVal = 0.0f;

return retVal;

}

void WriteValue(float val)

{

out.write(reinterpret\_cast<char\*>(&val), sizeof(float));

outFileHeader.dim0++;

}

void GoToStartOfData()

{

in.seekg(dataIndex);

}

dsp\_file\_header inFileHeader {};

dsp\_file\_header outFileHeader {};

bool eof;

private:

ifstream in;

ofstream out;

fstream::pos\_type dataIndex {};

};

## dspFileHeader.h

#pragma once

typedef struct

{

// signal (1)

// image (2)

// video (3)

int ndim;

// signal (1= mono , 2= stereo , etc .)

// grayscale image / video (1)

// color image / video (3)

int nchan;

// signal = > length

// image or video = > number rows

int dim0;

// signal = > if audio , then dim1 = sample rate

// image or video = > number columns

int dim1;

// signal = > 0

// image = > 0

// video = > number\_frames

int dim2;

} dsp\_file\_header;

## Convolution.h

#pragma once

#include <string>

#include <deque>

#include <vector>

#include <cmath>

#include <algorithm>

#include "fileRead.h"

#include "Fft842.h"

#include <complex>

using namespace std;

class Convolution

{

public:

/\*\*

\* **\brief** Performs filtering of the input file.

\* **\param** inFile input file containing x[n] data

\* **\param** outFile desired output file for y[n]

\*/

Convolution(const string& inFile, const string& outFile)

: file(inFile, outFile)

{ }

/\*\*

\* **\brief** The filter system for performing filtering in the time domain

\*/

void Filter()

{

printf("performing normal filter\n");

xBuff.clear();

xBuff.resize(FILTER\_SIZE, complx { 0.0f, 0.0f });

file.GoToStartOfData();

while (!file.eof)

{

file.WriteValue(Filter(file.GetValue()));

}

}

/\*\*

\* **\brief** The system to convolve the given system by using the fft using overlap-save

\*/

void FastConvolution()

{

const auto nFft = static\_cast<unsigned int>(pow(2, 10));

auto eofXIndex = nFft;

// ReSharper disable once CppInconsistentNaming

vector<complx> H(nFft);

vector<complx> yBuff(nFft, complx { 0.0f, 0.0f });

vector<complx> lastXValues(FILTER\_SIZE, complx { 0.0f, 0.0f });

printf("Performing fast convolution\n");

xBuff.clear();

xBuff.resize(nFft, complx { 0.0f, 0.0f });

file.GoToStartOfData();

//Create the fft of the FIR filter

for (auto i = 0u; i < nFft; i++)

{

//fill with 0's if the end of the filter has been reached

H[i] = complx { i < FILTER\_SIZE ? static\_cast<float>(FIRValues[i]) : 0.0f, 0.0f };

}

fft842(0, nFft, &H[0]);

//Overlap-save

while (!file.eof)

{

//copy over the last FILTER\_SIZE values from the end of xBuff into the beginning

copy(lastXValues.begin(), lastXValues.end(), xBuff.begin());

//Fill xBuff with nFft items. If eof is reached, fill the rest of xBuff with zeros

for (auto i = FILTER\_SIZE; !file.eof && i < xBuff.size(); i++)

{

xBuff[i] = complx { file.GetValue(),0.0f };

if (file.eof) eofXIndex = i;

}

//fill end of xBuff with zeros if required

for (auto it = xBuff.begin() + eofXIndex; it != xBuff.end(); ++it)

{

it->re = 0;

}

//store last FILTER\_SIZE x values before the fft

copy(xBuff.end() - FILTER\_SIZE, xBuff.end(), lastXValues.begin());

//perform fft of x

fft842(0, nFft, &xBuff[0]);

//multiply X\*H. All containers are the same size

for (auto xVal = xBuff.begin(), hVal = H.begin(), yVal = yBuff.begin(); xVal != xBuff.end(); ++xVal, ++hVal, ++yVal)

{

\*yVal = complx { xVal->re\*hVal->re - xVal->im\*hVal->im, xVal->re\*hVal->im + xVal->im\*hVal->re };

}

fft842(1, nFft, &yBuff[0]);

//write yBuff out to file, but do not include the first FILTER\_SIZE values

for (auto yVal = yBuff.begin() + FILTER\_SIZE; yVal != yBuff.end(); ++yVal)

{

file.WriteValue(yVal->re);

}

}

}

private:

FileRead file;

const unsigned int FILTER\_SIZE = 256;

vector<complx> xBuff;

/\*\*

\* **\brief** Shift the given array right by one and put 0 at x[0]

\* **\param** x array to shift

\*/

static void ShiftXBuffRight(vector<complx> &x)

{

for (auto it = x.rbegin(); it != x.rend() - 1; ++it)

{

\*it = \*(it + 1);

}

x[0] = complx { 0.0f, 0.0f };

}

/\*\*

\* **\brief** filters the given x value through a FIR filter.

\* **\param** x input x'[n] value

\* **\return** output y'[n] value

\*/

float Filter(const float x)

{

auto y = 0.0;

// Step 1. Shift xbuff and put x into xbuff[0]

xBuff.erase(xBuff.end() - 1);

xBuff.insert(xBuff.begin(), complx { x,0.0f });

// Step 3. Accumulate filter output into y

for (auto i = 0u; i < FILTER\_SIZE; i++)

{

y += FIRValues[i] \* xBuff[i].re;

}

return static\_cast<float>(y);

}

// ReSharper disable once CppInconsistentNaming

/\*\*

\* **\brief** LPF: **\n**

\* pass band 300 Hz**\n**

\* stop band 400 Hz**\n**

\* sampling rate 11.025 kHz**\n**

\* order 255

\*/

const vector<double> FIRValues =

{

0.002654502205485941186691167814615255338,

-0.000533350612502407443737961756369259092,

-0.000523481331803436842481369595247997495,

-0.000536265772457803196332504214183245494,

-0.000565977232532050098662290693596332858,

-0.000605369468223258908740058714670340123,

-0.000648881855188379336879522796266428486,

-0.000689511600067384960999206100495939609,

-0.000722306835381734523710006712349240843,

-0.000741095523384984865983282631418660458,

-0.000742235551558927814097543773641518783,

-0.000721102558569612812899851661541106296,

-0.000676145911868729354576856849234900437,

-0.000604809419986361215815229286363319261,

-0.000508235677052883919740944307363861299,

-0.000386355595209130799061458816368030966,

-0.000243257551255629880793129871108249063,

-0.000081237092824537181917632366889847617,

0.000092607234619735290942488248333575029,

0.000274406753606500162373516005231977033,

0.000454006515754755597680247802827580017,

0.000628494424768365287509164218704427185,

0.000783640961252869781145102834329918551,

0.000917187868590693591158702346888276224,

0.001022901360418262614593309933752607321,

0.001086549731248858795956691025708096277,

0.001109595552648575018939158809416767326,

0.001086676041785410162798797273353557102,

0.001017102996069627583730632380820679828,

0.000898418159924229027535336200571691734,

0.000733543136105446486744385836686888069,

0.000525535196974220505938824032909906236,

0.000281785001313752166834936119244048314,

0.000009733059236404027064759603149468603,

-0.000279859230762246474766286041813145857,

-0.000576534242109856542857215799813275225,

-0.000867474063631959099729407114409696078,

-0.001140591258773813402443986220191618486,

-0.001382648243103430391737207472147019871,

-0.001581978304470268818782141551082531805,

-0.001726996584912693540048156393140743603,

-0.001808663669303817514433241342430846998,

-0.001819247032479836493484848247703666857,

-0.001754186972933681129549476906959171174,

-0.001612127169501652822203308090820428333,

-0.001392848667946171813125655347676001838,

-0.001103477976040057647461045675640889385,

-0.000750103194709418087671171093688826659,

-0.000343923051628442725397016843658093421,

0.000100100645252879441593485410066932673,

0.000565829473927381178255391347420300008,

0.001035229733584466533385937125899545208,

0.001489161079134999898787583560988423415,

0.001907273891100903480746997509243101376,

0.002269959010425023904211627723270794377,

0.002558669228934338027436901086275611306,

0.002757423476532034484209887637007341255,

0.002852638945384879110012121827821829356,

0.00283475791595781418524824246674143069 ,

0.002697680742786473864769369868099602172,

0.002440552685068319259398261777960215113,

0.002066860532067003528022830494137451751,

0.001585888519149795342771946948801087274,

0.001011147444769284337029446874112181831,

0.000361726580225135013746395884481898975,

-0.00034025578352630397871436684908985626 ,

-0.00106785143813435722329185928458628041 ,

-0.001793072112757641486946447173522756202,

-0.002485511490834980442843349379700157442,

-0.003114104665111180891834408157592406496,

-0.003650045138933155977611910714131226996,

-0.004065273066141038282861330799278221093,

-0.004335610488843737349295981431396285188,

-0.004441528976883934998198810717440210283,

-0.004369354126972123560090910387998519582,

-0.00411145258536815914190976073427918891 ,

-0.003667433487270737105245821396692917915,

-0.003044482371017765386100872859742594301,

-0.002257877208273909985097871455650420103,

-0.001330056601311772543350109287985105766,

-0.000290730601350479289646244751565973274,

0.000824398759250688419747410407012466749,

0.001974227580704707683822984876087502926,

0.003113779001801766314200259344602272904,

0.004195639767784059559474485467944759876,

0.00517189209259954693620464638570410898 ,

0.005995920396462673523929609586957667489,

0.006624234399567093499461289951568687684,

0.007018602216282739635067056127581963665,

0.007147016182928411960617598452927268227,

0.006986972529026767853022050758227123879,

0.006524581848674985973257633986577275209,

0.005757827930862290111146961635313346051,

0.004696272255225764601094251560198244988,

0.003361454013858048220364516112113051349,

0.001787102779591722822444266327579498466,

0.000018772826991039683747208854924792831,

-0.001887405559453204234587886389817867894,

-0.003865954782148284946119431992883619387,

-0.005843830078811446462894174658231349895,

-0.007742059100272889737848203139947145246,

-0.009478521556002015430175688948111201171,

-0.010970630837229029472257302302296011476,

-0.012137627196270183924098340355612890562,

-0.012903783872366771101614091321607702412,

-0.013200511883456160500127651857837918214,

-0.012969557566902266063091353487379819853,

-0.01216484461893429548651202054543318809 ,

-0.010755104010294043101469974033079779474,

-0.008724723986257558461243633018966647796,

-0.006075674421187112622666504080370941665,

-0.002827252144644222097563668327779851097,

0.000982916392378119490452759166032592475,

0.00529967382911480152513084007637189643 ,

0.010051809537935427196764237578463507816,

0.015152675226749102999068696817630552687,

0.020503116012398597822752321917505469173,

0.025993579746129705926316333375325484667,

0.031507182864714593228949723879850353114,

0.0369226360846745538246693740802584216 ,

0.042117825407992366681941831529911723919,

0.046973104317882034830500259658947470598,

0.051374929697165318265295042010620818473,

0.055218465865475976905862864896334940568,

0.058411766909792744995044699862773995847,

0.060877467040941476983384461618697969243,

0.062555638000706834112030207961652195081,

0.063405158466785682680466607052949257195,

0.063405158466785682680466607052949257195,

0.062555638000706834112030207961652195081,

0.060877467040941476983384461618697969243,

0.058411766909792744995044699862773995847,

0.055218465865475976905862864896334940568,

0.051374929697165318265295042010620818473,

0.046973104317882034830500259658947470598,

0.042117825407992366681941831529911723919,

0.0369226360846745538246693740802584216 ,

0.031507182864714593228949723879850353114,

0.025993579746129705926316333375325484667,

0.020503116012398597822752321917505469173,

0.015152675226749102999068696817630552687,

0.010051809537935427196764237578463507816,

0.00529967382911480152513084007637189643 ,

0.000982916392378119490452759166032592475,

-0.002827252144644222097563668327779851097,

-0.006075674421187112622666504080370941665,

-0.008724723986257558461243633018966647796,

-0.010755104010294043101469974033079779474,

-0.01216484461893429548651202054543318809 ,

-0.012969557566902266063091353487379819853,

-0.013200511883456160500127651857837918214,

-0.012903783872366771101614091321607702412,

-0.012137627196270183924098340355612890562,

-0.010970630837229029472257302302296011476,

-0.009478521556002015430175688948111201171,

-0.007742059100272889737848203139947145246,

-0.005843830078811446462894174658231349895,

-0.003865954782148284946119431992883619387,

-0.001887405559453204234587886389817867894,

0.000018772826991039683747208854924792831,

0.001787102779591722822444266327579498466,

0.003361454013858048220364516112113051349,

0.004696272255225764601094251560198244988,

0.005757827930862290111146961635313346051,

0.006524581848674985973257633986577275209,

0.006986972529026767853022050758227123879,

0.007147016182928411960617598452927268227,

0.007018602216282739635067056127581963665,

0.006624234399567093499461289951568687684,

0.005995920396462673523929609586957667489,

0.00517189209259954693620464638570410898 ,

0.004195639767784059559474485467944759876,

0.003113779001801766314200259344602272904,

0.001974227580704707683822984876087502926,

0.000824398759250688419747410407012466749,

-0.000290730601350479289646244751565973274,

-0.001330056601311772543350109287985105766,

-0.002257877208273909985097871455650420103,

-0.003044482371017765386100872859742594301,

-0.003667433487270737105245821396692917915,

-0.00411145258536815914190976073427918891 ,

-0.004369354126972123560090910387998519582,

-0.004441528976883934998198810717440210283,

-0.004335610488843737349295981431396285188,

-0.004065273066141038282861330799278221093,

-0.003650045138933155977611910714131226996,

-0.003114104665111180891834408157592406496,

-0.002485511490834980442843349379700157442,

-0.001793072112757641486946447173522756202,

-0.00106785143813435722329185928458628041 ,

-0.00034025578352630397871436684908985626 ,

0.000361726580225135013746395884481898975,

0.001011147444769284337029446874112181831,

0.001585888519149795342771946948801087274,

0.002066860532067003528022830494137451751,

0.002440552685068319259398261777960215113,

0.002697680742786473864769369868099602172,

0.00283475791595781418524824246674143069 ,

0.002852638945384879110012121827821829356,

0.002757423476532034484209887637007341255,

0.002558669228934338027436901086275611306,

0.002269959010425023904211627723270794377,

0.001907273891100903480746997509243101376,

0.001489161079134999898787583560988423415,

0.001035229733584466533385937125899545208,

0.000565829473927381178255391347420300008,

0.000100100645252879441593485410066932673,

-0.000343923051628442725397016843658093421,

-0.000750103194709418087671171093688826659,

-0.001103477976040057647461045675640889385,

-0.001392848667946171813125655347676001838,

-0.001612127169501652822203308090820428333,

-0.001754186972933681129549476906959171174,

-0.001819247032479836493484848247703666857,

-0.001808663669303817514433241342430846998,

-0.001726996584912693540048156393140743603,

-0.001581978304470268818782141551082531805,

-0.001382648243103430391737207472147019871,

-0.001140591258773813402443986220191618486,

-0.000867474063631959099729407114409696078,

-0.000576534242109856542857215799813275225,

-0.000279859230762246474766286041813145857,

0.000009733059236404027064759603149468603,

0.000281785001313752166834936119244048314,

0.000525535196974220505938824032909906236,

0.000733543136105446486744385836686888069,

0.000898418159924229027535336200571691734,

0.001017102996069627583730632380820679828,

0.001086676041785410162798797273353557102,

0.001109595552648575018939158809416767326,

0.001086549731248858795956691025708096277,

0.001022901360418262614593309933752607321,

0.000917187868590693591158702346888276224,

0.000783640961252869781145102834329918551,

0.000628494424768365287509164218704427185,

0.000454006515754755597680247802827580017,

0.000274406753606500162373516005231977033,

0.000092607234619735290942488248333575029,

-0.000081237092824537181917632366889847617,

-0.000243257551255629880793129871108249063,

-0.000386355595209130799061458816368030966,

-0.000508235677052883919740944307363861299,

-0.000604809419986361215815229286363319261,

-0.000676145911868729354576856849234900437,

-0.000721102558569612812899851661541106296,

-0.000742235551558927814097543773641518783,

-0.000741095523384984865983282631418660458,

-0.000722306835381734523710006712349240843,

-0.000689511600067384960999206100495939609,

-0.000648881855188379336879522796266428486,

-0.000605369468223258908740058714670340123,

-0.000565977232532050098662290693596332858,

-0.000536265772457803196332504214183245494,

-0.000523481331803436842481369595247997495,

-0.000533350612502407443737961756369259092,

0.002654502205485941186691167814615255338

};

};