Федеральное агентство связи

Ордена Трудового Красного Знамени

федеральное государственное бюджетное образовательное учреждение высшего образования

«Московский технический университет связи и информатики»

Кафедра Теории вероятностей и прикладной математики



ОТЧЕТ

по учебной практике

*Разработка программы, определяющей частоту и ноту звука, с помощью преобразования Фурье*

Выполнил:

студент гр. БПМ1701

Кахиани М.Т

Руководитель практики от МТУСИ:

доц. каф ТВ и ПМ Смирнова Н.И.

Москва 2017

**ОГЛАВЛЕНИЕ**

Введение

Основная часть

Заключение

Список литературы

**Введение**

Задача – разработать программу, способную определить частоту и ноту звука.

План выполнения:

1) Изучение алгоритма БПФ.

2) Реализация алгоритма на языке C#.

3) Работа со звуком в среде разработки Visual Studio, при помощи библиотек DirectX.

4) Определение ноты, в соответствии с полученной частотой звука.

**Основная часть**

1. Изучение алгоритма БПФ.

Начнем с извлечения звука на музыкальном инструменте. К примеру, возьмем гитару. Исполняя композицию, музыкант извлекает звук одной, или нескольких струн, в результате чего несколько синусоидальных сигналов накладываются друг на друга образуя рисунок. Для того чтобы определить ноту конкретно извлекаемого звука, необходимо разложить суммарный сигнал на исходные синусоиды. По сути, этот процесс и представляет собой прямое преобразование Фурье. А извлечение звука есть процесс обратного преобразования Фурье. Для анализа сигнала с помощью преобразования Фурье, мы идеализируем ситуацию и исходим из предположения, что он периодический на текущем временном интервале и состоит из элементарных синусоид.

Процесс прямого преобразования Фурье представляет собой выделение некоторого момента сигнала, условно считаемого периодическим, который состоит из дискретных отсчетов. В результате преобразования получаем массив из комплексных чисел – информацию о спектрах сигнала. Массив заполнен полезной информацией ровно на половину, другая половина является лишь зеркальным отражением первой и спокойно может быть исключена из рассмотрения, что является следствием из теоремы Котельникова о том, что частота дискретизации должна быть не меньше максимальной удвоенной частоты сигнала. Полученные Спектры так же являются дискретными, с шагом равным отношению частоты дискретизации к количеству отсчетов. Чем больше отсчетов, тем точнее получаем частоту, но также увеличивается временной интервал выделенного момента сигнала. Это приводит к наложению нескольких нот, если они имеют малую длительность звучания. В случае использования программы как тюнер, этот способ подходит, но для распознавания всех звучащих нот в композиции – негоден.

1. Реализация алгоритма на языке C#

Существует много разновидностей алгоритмов БПФ для реализации в виде программного кода, один из них - алгоритм Кули-Тьюки, применяемый на практике чаще всего.

Принцип алгоритма:

* ДПФ размерности N{\displaystyle N\,\!} выражается через сумму ДПФ более малых размерностей N1{\displaystyle N\_{1}\,\!} и N2{\displaystyle N\_{2}\,\!}.
* Аналогичным образом N1{\displaystyle N\_{1}\,\!} и N2{\displaystyle N\_{2}\,\!} рекурсивно выражаются через ДПФ ещё более малых размерностей.
* Разложение ДПФ в сумму 2 частей можно организовать таким образом, что первая часть будет представлять собой сумму по чётным индексам, а вторая — по нечётным.

Пусть {\displaystyle {\vec {x}}\,\!} — вектор, для которого необходимо вычислить ДПФ. Тогда его Фурье-образ можно представить в виде последовательностиТогда, Фурье-образ приводится к следующему виду:

где {\displaystyle SX\_{chet}=\sum \_{i\in chet}X\_{i,k}} — сумма по чётным индексам;

— сумма по нечётным индексам.

В силу периодичности ДПФ:

откуда:

Таким образом, все компоненты ДПФ можно рекурсивно раскладывать по уменьшению размерности до Фурье-образов 2 точек, которые определяются по формулам:

Прямое ДПФ требует ~ операций сложения и умножения комплексных чисел. Алгоритм БПФ Кули-Тьюки позволяет уменьшить сложность вычислений до .

Код:

// БПФ Кули — Туки

public static class FftAlgorithm

{

//х - входные данные

//return - спектрограмма данных

public static double[] Calculate(double[] x)

{

int length;

int bitsInLength;

if (IsPowerOfTwo(x.Length))

{

length = x.Length;

bitsInLength = Log2(length) - 1;

}

else

{

bitsInLength = Log2(x.Length);

length = 1 << bitsInLength;

}

ComplexNumber[] data = new ComplexNumber[length];

for (int i = 0; i < x.Length; i++)

{

int j = ReverseBits(i, bitsInLength);

data[j] = new ComplexNumber(x[i]);

}

for (int i = 0; i < bitsInLength; i++)

{

int m = 1 << i;

int n = m \* 2;

double alpha = -(2 \* Math.PI / n);

for (int k = 0; k < m; k++)

{

// e^(-2\*pi/N\*k)

ComplexNumber oddPartMultiplier = new ComplexNumber(0, alpha \* k).PoweredE();

for (int j = k; j < length; j += n)

{

ComplexNumber evenPart = data[j];

ComplexNumber oddPart = oddPartMultiplier \* data[j + m];

data[j] = evenPart + oddPart;

data[j + m] = evenPart - oddPart;

}

}

}

//вычисление спектрограмы

double[] spectrogram = new double[length];

for (int i = 0; i < spectrogram.Length; i++)

{

spectrogram[i] = data[i].AbsPower2();

}

return spectrogram;

}

//получает количество значащих байт

//количество битов для хранения числа

private static int Log2(int n)

{

int i = 0;

while (n > 0)

{

++i; n >>= 1;

}

return i;

}

//Меняет биты в числах

//n - число

//bitsCount кол-во значащих битов в числе

private static int ReverseBits(int n, int bitsCount)

{

int reversed = 0;

for (int i = 0; i < bitsCount; i++)

{

int nextBit = n & 1;

n >>= 1;

reversed <<= 1;

reversed |= nextBit;

}

return reversed;

}

//проверяет является ли число степенью двойки

//true если n=2^k и k - положительное целое

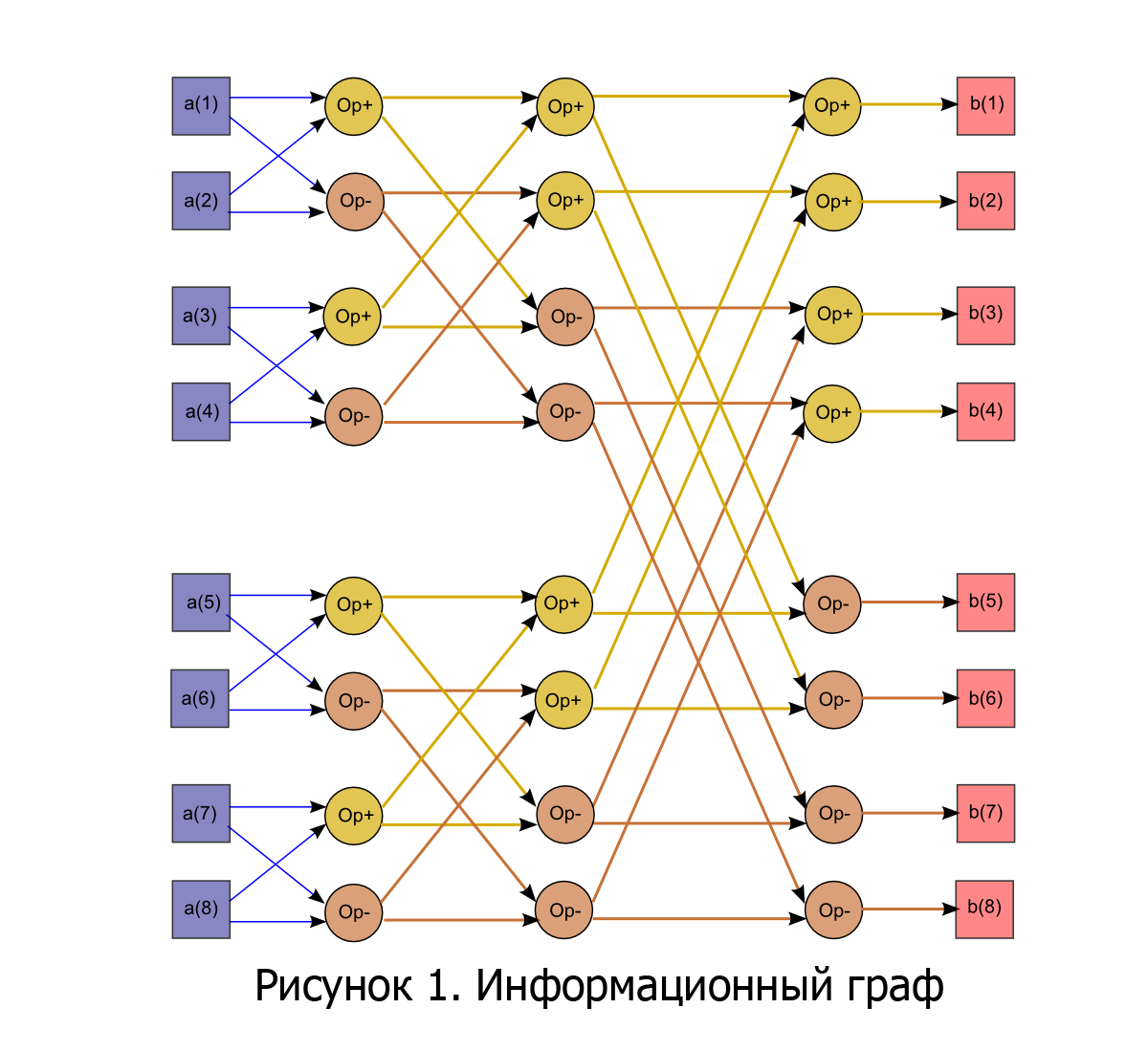
private static bool IsPowerOfTwo(int n)

{

return n > 1 & (n & (n - 1)) == 0;

}

}

**

3) Работа со звуком в среде разработки Visual Studio, при помощи библиотек DirectX (DirectSound).

Для получения сигнала в программе используется набор сторонних библиотек, обеспечивающих ей практически непосредственный доступ к аппаратуре звукового адаптера. DirectSound построена по объектно-ориентированному принципу и состоит из набора интерфейсов. Каждый интерфейс отвечает за объект определенного типа - устройство, буфер, службу уведомления и т.п. По сути, интерфейс представляет собой обычный набор управляющих функций, или методов, организованных в класс объектно-ориентированного языка.

В программе создано отдельное решение, содержащее базовый класс для захвата аудио сэмплов (SoundCaptureBase.cs) и устройства записи (SoundCaptureDevice.cs)

Класс SoundCaptureBase.cs

//Класс для захвата аудио сэмплов

public abstract class SoundCaptureBase : IDisposable

{

//свойства звука

const int BufferSeconds = 3;

const int NotifyPointsInSecond = 2;

const int BitsPerSample = 16; //разрешение сэмпла (кол-во бит)

const int ChannelCount = 1; //количество каналов записи

int sampleRate = 44100; //частота дискретизации (сэмплов в секунку)

bool isCapturing = false;

bool disposed = false;

public bool IsCapturing

{

get { return isCapturing; }

}

public int SampleRate

{

get { return sampleRate; }

set

{

if (sampleRate <= 0) throw new ArgumentOutOfRangeException();

sampleRate = value;

}

}

Capture capture;

CaptureBuffer buffer;

Notify notify;

int bufferLength;

AutoResetEvent positionEvent;

SafeWaitHandle positionEventHandle;

ManualResetEvent terminated;

Thread thread;

SoundCaptureDevice device;

public SoundCaptureBase(SoundCaptureDevice device)

{

this.device = device;

positionEvent = new AutoResetEvent(false); //уведомление потока, что событие еще не произошло

positionEventHandle = positionEvent.SafeWaitHandle;

terminated = new ManualResetEvent(true); //уведомление потока, что событие произошло

}

//Начало захвата

public void Start()

{

isCapturing = true;

WaveFormat format = new WaveFormat(); //структура библиотеки DirectX, определяющая формат аудио

format.Channels = ChannelCount;

format.BitsPerSample = BitsPerSample;

format.SamplesPerSecond = SampleRate;

format.FormatTag = WaveFormatTag.Pcm;

format.BlockAlign = (short)((format.Channels \* format.BitsPerSample + 7) / 8); //минимальная единица данных

format.AverageBytesPerSecond = format.BlockAlign \* format.SamplesPerSecond; //скорость передачи данных байт/сек

bufferLength = format.AverageBytesPerSecond \* BufferSeconds;

CaptureBufferDescription desciption = new CaptureBufferDescription(); //структура описывающая обьект DirectSound.CaptureBuffer

desciption.Format = format;

desciption.BufferBytes = bufferLength; //размер буфера в байтах

capture = new Capture(device.Id);

buffer = new CaptureBuffer(desciption, capture);

int waitHandleCount = BufferSeconds \* NotifyPointsInSecond;

BufferPositionNotify[] positions = new BufferPositionNotify[waitHandleCount];

for (int i = 0; i < waitHandleCount; i++)

{

BufferPositionNotify position = new BufferPositionNotify();

position.Offset = (i + 1) \* bufferLength / positions.Length - 1;

position.EventNotifyHandle = positionEventHandle.DangerousGetHandle();

positions[i] = position;

}

notify = new Notify(buffer);

notify.SetNotificationPositions(positions);

terminated.Reset();

thread = new Thread(new ThreadStart(ThreadLoop));

thread.Name = "Sound capture";

thread.Start();

}

private void ThreadLoop()

{

buffer.Start(true);

try

{

int nextCapturePosition = 0;

WaitHandle[] handles = new WaitHandle[] { terminated, positionEvent };

while (WaitHandle.WaitAny(handles) > 0)

{

int capturePosition, readPosition;

buffer.GetCurrentPosition(out capturePosition, out readPosition);

int lockSize = readPosition - nextCapturePosition;

if (lockSize < 0) lockSize += bufferLength;

if((lockSize & 1) != 0) lockSize--;

int itemsCount = lockSize >> 1;

short[] data = (short[])buffer.Read(nextCapturePosition, typeof(short), LockFlag.None, itemsCount);// Считываются данные в массив

ProcessData(data); // Данные посылаются на обработку

nextCapturePosition = (nextCapturePosition + lockSize) % bufferLength;

}

}

finally

{

buffer.Stop();

}

}

//обработка захваченных данных

protected abstract void ProcessData(short[] data);

//высвобождение данных, прерыв потока записи данных

public void Stop()

{

if (isCapturing)

{

isCapturing = false;

terminated.Set();

thread.Join();

notify.Dispose();

buffer.Dispose();

capture.Dispose();

}

}

void IDisposable.Dispose()

{

Dispose(true);

}

~SoundCaptureBase()

{

Dispose(false);

}

private void Dispose(bool disposing)

{

if (disposed) return;

disposed = true;

if (IsCapturing) Stop();

positionEventHandle.Dispose();

positionEvent.Close();

terminated.Close();

Данный класс описывает характеристики звука, такие как

* Размер буфера
* Разрешение сэмпла (Количество бит)
* Количество каналов записи
* Частота дискретизации (Сэмплов в секунду)

И работу с ними.

Класс SoundCaptureDevice.cs

//Записывающее устройство.

public class SoundCaptureDevice

{

Guid id;

string name;

public bool IsDefault

{

get { return id == Guid.Empty; }

}

//Название устройства

public string Name

{

get { return name; }

}

public Guid Id

{

get { return id; }

}

public SoundCaptureDevice(Guid id, string name)

{

this.id = id;

this.name = name;

}

public static SoundCaptureDevice[] GetDevices()

{

CaptureDevicesCollection captureDevices = new CaptureDevicesCollection();

List<SoundCaptureDevice> devices = new List<SoundCaptureDevice>();

foreach (DeviceInformation captureDevice in captureDevices)

{

devices.Add(new SoundCaptureDevice(captureDevice.DriverGuid, captureDevice.Description));

}

return devices.ToArray();

}

}

Класс SoundCaptureDevice.cs позволяет получить список устройств и входов, способных принимать звуковой сигнал.

1. Определение ноты, в соответствии с полученной частотой звука.

Последней задачей программы является определить ноту. На данном этапе для этого все есть.

Кусок кода из основного класса программы:

static string[] NoteNames = { "A", "A#", "B", "C", "C#", "D", "D#", "E", "F", "F#", "G", "G#" };

static double ToneStep = Math.Pow(2, 1.0 / 12);

private void FindClosestNote(double frequency, out double closestFrequency, out string noteName)

{

const double AFrequency = 440.0;

const int ToneIndexOffsetToPositives = 120;

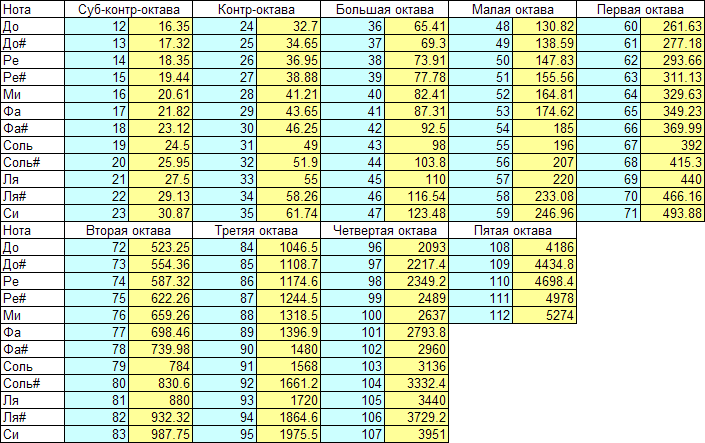
int toneIndex = (int)Math.Round(Math.Log(frequency / AFrequency, ToneStep));

noteName = NoteNames[(ToneIndexOffsetToPositives + toneIndex) % NoteNames.Length];

closestFrequency = Math.Pow(ToneStep, toneIndex) \* AFrequency;

}

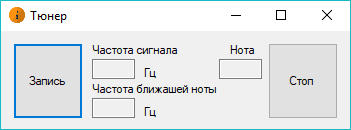
Задается массив из нот (NoteNames), обозначенных буквами латинского алфавита, и переменная, обозначающая шаг значения частоты между двумя нотами, равное . Это число берется из соображения того, что в октаве имеется 12 нот, и также ноты, отстоящие на одну октаву, отличаются частотой в 2 раза.

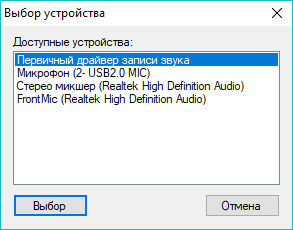


Далее описывается метод, определяющий ближайшую ноту, к частоте подающейся на вход. Переменная AFrequency – частота эталонной ноты «Ля», равная 440 Гц. Переменная toneIndex определяет, на сколько полутонов отличается нота, от эталона. Именно здесь и происходит вычисление ближайшей ноты. Воспользуемся формулой , где – входящая нота, которую необходимо определить, – камертон нота «Ля», i – количество полутонов в интервале от 440 Гц до искомой ноты. Запишем формулу в другом виде . Так как число i должно быть целым, округлим его. Затем, из заданного массива определяем ноту, на основе рассчитанного значения . В итоге получаем ближайшую ноту, исходя из заданной частоты.

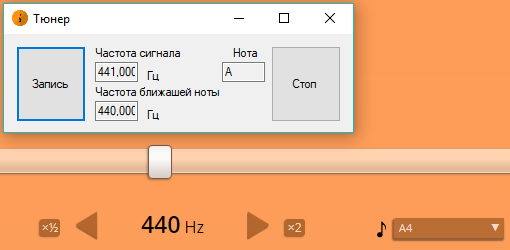
**Заключение**

Разработанная программа отлично подходит для точной настройки гитары, либо иных струнных инструментов. Благодаря библиотекам DirectSound возможна запись не только с микрофона, но и напрямую со звуковой карты или линейного входа:





Проверка программы, с помощью генератора частот.



**Список Литературы**

1. Основы спектрального анализа звуков

<https://pandia.ru/text/77/481/644.php>

1. Основная схема быстрого преобразования Фурье

https://parallel.ru/fpga/fft

1. Преобразование Фурье в действии: точное определение частоты сигнала и выделение нот

<https://habr.com/ru/post/247385/>

1. Программирование звука в DirectSound

http://www.codenet.ru/progr/directx/dxsound.php#001