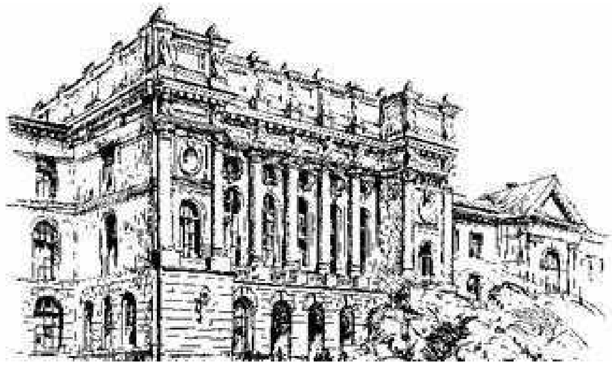
Санкт-Петербургский Политехнический Университет Петра Великого
Институт компьютерных наук и технологий
Кафедра компьютерных систем и программных технологий



Программирование

Отчет по курсовой работе
Приложение "Sun Radio"

Работу выполнила:
Кремнева В.Э.
Группа: 23501/4
Преподаватель:
Вылегжанина К.Д.

Санкт-Петербург
2017

Содержание

1 Проектирование приложения

В современном мире музыка является неотъемлемой частью жизни многих и многих людей. Музыка часто сопровождает нас на работе, в дороге, во время досуга. Порой музыка может играть долгие часы. Настолько долгие, что за время прослушивания солнце может изменить свое положение относительно прослушивающего. Со временем суток так же меняется и настроение, и запас сил. И как было бы здорово, если бы музыка подстраивалась под наше самоощущение, дополняла и обогащала его.

1.1 Задание

Разработать приложение, позволяющее пользователям автоматически изменять тональность и громкость воспроизводимой музыки в соответствии с уровнем освещения.

1.2 Концепция

Изменения в музыкальные файлы вносятся с помощью преобразования Фурье. Информация об освещенности поступает с фоторезистра.

1.3 Минимально работоспособный продукт

Консольное приложение, получающее на вход музыкальный файл и воспроизводящее его в соответствии с текущим уровнем освещенности.

1.4 Решаемые задачи

В процессе проектирования приложения было выделено три основных задачи.

- **Получение спектра файла**

  Для удобства работы используются .wav файлы. Удобство в том, что в каждом фрейме хранятся значения амплитуд, то есть имеем зависимость амплитуды от времени, что является входными данными для преобразования Фурье. С помощью этого преобразования получим спектр файла – зависимость амплитуды от частоты. Но не всё так просто. При последовательном чтении фреймов накапливаются ошибки, поэтому необходимо использовать оконную функцию и читать данные с перекрытием. Эмпирически получено наилучшее перекрытие – в одну шестнадцатую от количества читаемых фреймов при условии, что читаем по 2048 фреймов. Число фреймов, равное степени двойки, выбрано не случайно: это задел на будущее. Для быстрого преобразования Фурье необходимо число фреймов, равное степени двойки. Используемая оконная функция – функция Блэкмана-Наталла. Выбрана она за минимальный размер боковых лепестков и удовлетворительное "растяжение"спектра. При использовании входного фильтра необходим выходной фильтр. Выходной фильтр должен быть таким, чтобы сумма произведений входного и выходного фильтров в каждой точке была равна единице. Для этого здесь я просто нормализую входную оконную функцию.

- **Корректные преобразования**

  Необходимо изменить тон и громкость воспроизведения. Для изменения тона необходимо растянуть файл в N раз, интерполируя значения амплитуды и фазы между точками, а затем ускорить воспроизведение в N раз. Таким образом получим изменение тона при сохранении скорости воспроизведения.

- **Воспроизведение файла**

  Воспроизведение файла производится средствами javax.sound.sampled.*

1.5 Выводы

В данном разделе рассмотрен процесс проектирования приложения для модуляции звука в зависимости от уровня освещенности. Выделены основные задачи и предложены варианты их решения.

2 Реализация приложения

2.1 Среда разработки

Операционная система: Windows 8.1
Среда разработки: IntelliJ IDEA 2016.3.4
Компилятор: javac, `JDK 1.8.0_102`

2.2 Выделенные классы

В приложении были выделены следующие классы:

- **SunRadio** - главный класс. Осуществляет основную работу приложения.
- **Complex** - класс для работы с комплексными числами. Взят из открытого источника.
- **WavFile** - класс для работы с .wav файлами. Взят из открытого источника.
- **WavFileException** - исключения для класса WavFile. Взят из открытого источника.
- **LightLevel** - класс, с помощью которого можно получить текущий уровень освещенности.
- **AM** - класс, содержащий средства для амплитудной модуляции.
- **DFTStraight** - реализует прямое дискретное преобразование Фурье.
- **DFTInverse** - реализует обратное дискретное преобразование Фурье.
- **Filter** - реализует оконную функцию.
- **Interpolation** - реализует линецную интерполяцию по двум точкам.
- **Scale** - реализует масштабирование в заданных пределах.
- **ToneModulation** - реализует модуляцию тона.
- **ToneModulationException** - исключения для класса ToneModulation.

2.3 Выводы

В данном разделе были описаны все классы, выделенные в процессе работы над проектом.

3 Процесс обеспечения качества и тестирование

3.1 Тестирование

Для проверки работы библиотеки использовались автоматические тесты, покрывающие основную функциональность ядра. Также в процессе разработки приложения проводилось ручное тестирование программы.

3.2 Выводы

В данном разделе описан процесс тестирования программы.

4 Выводы

В результате работы над курсовым проектом было реализовано приложение, предназначенное для изменения музыки в соответствии с уровнем освещенности. Изучено: преобразование Фурье, оконные функции, амплитудная модуляция, линейная интерполяция. Так же приобретены навыки разработки приложения на языке Java, а также навыки разработки тестов на языке Groovy.

5 Приложение 1

Листинг 1: Complex.java

```
/*
 * This work by W. Patrick Hooper is free of known copyright restrictions.
 * The work is in the public domain.
 *
 * Author's website: <a href="http://wphooper.com">http://wphooper.com</a>.
 */

package com.external;

```

```java
/**
 * This class stores a complex number, and allows the user to do arithmetic
 * with these numbers.
 *
 * Note that our complex numbers are immutable. That is, once they are
 * constructed, they will not change. In particular, all our algebraic
 * operations create a new complex number rather than updating the current one.
 *
 * @author W. Patrick Hooper
 */
public final class Complex {
    // The number stored is x+I*y.
    final private double x, y;
    // I don't want to allow anyone to access these numbers so I've labeled
    // them private.

    /** Construct a point from real and imaginary parts. */
    public Complex(double real_part, double imaginary_part) {
        x=real_part;
        y=imaginary_part;
    }

    /** Construct a real number. */
    public Complex(double real_part) {
        x=real_part;
        y=0;
    }

    // A static constructor.

    /** Construct a complex number from the given polar coordinates. */
    public static Complex fromPolar(double r, double theta) {
        return new Complex(r*Math.cos(theta), r*Math.sin(theta));
    }

    // Basic operations on Complex numbers.

    /** Return the real part. */
    public double re(){
        return x;
    }

    /** Return the imaginary part. */
    public double im(){
        return y;
    }

    /** Return the complex conjugate */
    public Complex conj() {
        return new Complex(x,-y);
    }

    /** Return the square of the absolute value. */
    public double absSquared() {
        return x*x+y*y;
    }

    /** Return the absolute value. */
    public double abs() {
        // The java.lang.Math package contains many useful mathematical functions,
        // including the square root function.
        return Math.sqrt(absSquared());
    }

    // ARITHMETIC

    /** Add a complex number to this one.
     *
     * @param z The complex number to be added.
     * @return A new complex number which is the sum.
     */
    public Complex add(Complex z) {
        return new Complex(x+z.x, y+z.y);
    }

    /** Subtract a complex number from this one.
```

```java
     *
     * @param z The complex number to be subtracted.
     * @return A new complex number which is the sum.
     */
    public Complex minus(Complex z) {
        return new Complex(x-z.x, y-z.y);
    }

    /** Negate this complex number.
     *
     * @return The negation.
     */
    public Complex neg() {
        return new Complex(-x, -y);
    }

    /** Compute the product of two complex numbers
     *
     * @param z The complex number to be multiplied.
     * @return A new complex number which is the product.
     */
    public Complex mult(Complex z) {
        return new Complex(x*z.x-y*z.y, x*z.y+z.x*y);
    }

    /** Divide this complex number by a real number.
     *
     * @param q The number to divide by.
     * @return A new complex number representing the quotient.
     */
    public Complex div(double q) {
        return new Complex(x/q,y/q);
    }

    /** Return the multiplicative inverse. */
    public Complex inv() {
        // find the square of the absolute value of this complex number.
        double abs_squared=absSquared();
        return new Complex(x/abs_squared, -y/abs_squared);
    }

    /** Compute the quotient of two complex numbers.
     *
     * @param z The complex number to divide this one by.
     * @return A new complex number which is the quotient.
     */
    public Complex div(Complex z) {
        return mult(z.inv());
    }

    /** Return the complex exponential of this complex number. */
    public Complex exp() {
        return new Complex(Math.exp(x)*Math.cos(y),Math.exp(x)*Math.sin(y));
    }


    // FUNCTIONS WHICH KEEP JAVA HAPPY:

    /** Returns this point as a string.
     * The main purpose of this function is for printing the string out,
     * so we return a string in a (fairly) human readable format.
     */
    // The _optional_ override directive "@Override" below just says we are
    // overriding a function defined in a parent class. In this case, the
    // parent is java.lang.Object. All classes in Java have the Object class
    // as a superclass.
    @Override
    public String toString() {
        // Comments:
        // 1) "" represents the empty string.
        // 2) If you add something to a string, it converts the thing you
        // are adding to a string, and then concatentates it with the string.

        // We do some voodoo to make sure the number is displayed reasonably.
        if (y==0) {
            return ""+x;
```

```java
        }
        if (y>0) {
            return ""+x+"+"+y+"*I";
        }
        // otherwise y<0.
        return ""+x+"-"+(-y)+"*I";
    }

    /** Return true if the object is a complex number which is equal to this complex number. */
    @Override
    public boolean equals(Object obj) {
        // Return false if the object is null
        if (obj == null) {
            return false;
        }
        // Return false if the object is not a Complex number
        if (!(obj instanceof Complex)) {
            return false;
        }

        // Now the object must be a Complex number, so we can convert it to a
        // Complex number.
        Complex other = (Complex) obj;

        // If the x-coordinates are not equal, then return false.
        if (x != other.x) {
            return false;
        }
        // If the y-coordinates are not equal, then return false.
        if (y != other.y) {
            return false;
        }
        // Both parts are equal, so return true.
        return true;
    }

    // Remark: In Java, we should really override the hashcode function
    // whenever we override the equals function. But, I don't want to
    // get into this for a light introduction to programming in java.
    // Hash codes are necessary for various of Java's collections. See HashSet for instance.
    // The following was generated by Netbeans.
    @Override
    public int hashCode() {
        int hash = 3;
        hash = 83 * hash + (int) (Double.doubleToLongBits(this.x) ^ (Double.doubleToLongBits(this.x) >>> 32));
        hash = 83 * hash + (int) (Double.doubleToLongBits(this.y) ^ (Double.doubleToLongBits(this.y) >>> 32));
        return hash;
    }
}
```

Листинг 2: WavFile.java

```java
package com.external;

import java.io.*;

import javax.sound.sampled.AudioFormat;
import javax.sound.sampled.AudioSystem;


/**
 * Wav file IO class
 * http://www.labbookpages.co.uk
 *
 * File format is based on the information from
 * http://www.sonicspot.com/guide/wavefiles.html
 * http://www.blitter.com/~russtopia/MIDI/~jglatt/tech/wave.htm
 *
 * @author A.Greensted
 * @version 1.0
 */
public class WavFile
{
```

```java
  private enum IOState {READING, WRITING, CLOSED};
  private final static int BUFFER_SIZE = 4096;

  private final static int FMT_CHUNK_ID = 0x20746D66;
  private final static int DATA_CHUNK_ID = 0x61746164;
  private final static int RIFF_CHUNK_ID = 0x46464952;
  private final static int RIFF_TYPE_ID = 0x45564157;

  private File file;                    // File that will be read from or written to
  private IOState ioState;              // Specifies the IO State of the Wav File (used for snaity checking)
  private int bytesPerSample;           // Number of bytes required to store a single sample
  private long numFrames;               // Number of frames within the data section
  private FileOutputStream oStream;     // Output stream used for writting data
  private FileInputStream iStream;      // Input stream used for reading data
  private double floatScale;            // Scaling factor used for int <-> float conversion
  private double floatOffset;           // Offset factor used for int <-> float conversion
  private boolean wordAlignAdjust;      // Specify if an extra byte at the end of the data chunk is required for word alignment

  // Wav Header
  private int numChannels;              // 2 bytes unsigned, 0x0001 (1) to 0xFFFF (65,535)
  private long sampleRate;              // 4 bytes unsigned, 0x00000001 (1) to 0xFFFFFFFF (4,294,967,295)
                                        // Although a java int is 4 bytes, it is signed, so need to use a long
  private int blockAlign;               // 2 bytes unsigned, 0x0001 (1) to 0xFFFF (65,535)
  private int validBits;                // 2 bytes unsigned, 0x0002 (2) to 0xFFFF (65,535)

  // Buffering
  private byte[] buffer;                // Local buffer used for IO
  private int bufferPointer;            // Points to the current position in local buffer
  private int bytesRead;                // Bytes read after last read into local buffer
  private long frameCounter;            // Current number of frames read or written

  // Cannot instantiate WavFile directly, must either use newWavFile() or openWavFile()
  private WavFile()
  {
    buffer = new byte[BUFFER_SIZE];
  }

  public int getNumChannels()
  {
    return numChannels;
  }

  public long getNumFrames()
  {
    return numFrames;
  }

  public long getFramesRemaining()
  {
    return numFrames - frameCounter;
  }

  public long getFrameCounter() {
    return frameCounter;
  }

  public long getSampleRate()
  {
    return sampleRate;
  }

  public int getValidBits()
  {
    return validBits;
  }
  public int getblockAlign()
  {
    return blockAlign;
  }

  public AudioFormat getAudioFormat() throws Exception
  {
```

```java
    return AudioSystem.getAudioInputStream(file).getFormat();
  }

  public static WavFile newWavFile(File file, int numChannels, long numFrames, int validBits, long sampleRate) throws IOException, WavFileException
  {
    // Instantiate new Wavfile and initialise
    WavFile wavFile = new WavFile();
    wavFile.file = file;
    wavFile.numChannels = numChannels;
    wavFile.numFrames = numFrames;
    wavFile.sampleRate = sampleRate;
    wavFile.bytesPerSample = (validBits + 7) / 8;
    wavFile.blockAlign = wavFile.bytesPerSample * numChannels;
    wavFile.validBits = validBits;

    // Sanity check arguments
    if (numChannels < 1 || numChannels > 65535) throw new WavFileException("Illegal number of channels, valid range 1 to 65536");
    if (numFrames < 0) throw new WavFileException("Number of frames must be positive");
    if (validBits < 2 || validBits > 65535) throw new WavFileException("Illegal number of valid bits, valid range 2 to 65536");
    if (sampleRate < 0) throw new WavFileException("Sample rate must be positive");

    // Create output stream for writing data
    wavFile.oStream = new FileOutputStream(file);

    // Calculate the chunk sizes
    long dataChunkSize = wavFile.blockAlign * numFrames;
    long mainChunkSize =  4 + // Riff Type
                  8 + // Format ID and size
                  16 +  // Format data
                  8 +   // Data ID and size
                  dataChunkSize;

    // Chunks must be word aligned, so if odd number of audio data bytes
    // adjust the main chunk size
    if (dataChunkSize % 2 == 1) {
      mainChunkSize += 1;
      wavFile.wordAlignAdjust = true;
    }
    else {
      wavFile.wordAlignAdjust = false;
    }

    // Set the main chunk size
    putLE(RIFF_CHUNK_ID,  wavFile.buffer, 0, 4);
    putLE(mainChunkSize,  wavFile.buffer, 4, 4);
    putLE(RIFF_TYPE_ID, wavFile.buffer, 8, 4);

    // Write out the header
    wavFile.oStream.write(wavFile.buffer, 0, 12);

    // Put format data in buffer
    long averageBytesPerSecond = sampleRate * wavFile.blockAlign;

    putLE(FMT_CHUNK_ID,        wavFile.buffer, 0, 4);     // Chunk ID
    putLE(16,              wavFile.buffer, 4, 4);     // Chunk Data Size
    putLE(1,                wavFile.buffer, 8, 2);     // Compression Code (Uncompressed)
    putLE(numChannels,         wavFile.buffer, 10, 2);    // Number of channels
    putLE(sampleRate,          wavFile.buffer, 12, 4);    // Sample Rate
    putLE(averageBytesPerSecond,  wavFile.buffer, 16, 4);    // Average Bytes Per Second
    putLE(wavFile.blockAlign,    wavFile.buffer, 20, 2);    // Block Align
    putLE(validBits,           wavFile.buffer, 22, 2);    // Valid Bits

    // Write Format Chunk
    wavFile.oStream.write(wavFile.buffer, 0, 24);

    // Start Data Chunk
    putLE(DATA_CHUNK_ID,         wavFile.buffer, 0, 4);     // Chunk ID
    putLE(dataChunkSize,         wavFile.buffer, 4, 4);     // Chunk Data Size

    // Write Format Chunk
    wavFile.oStream.write(wavFile.buffer, 0, 8);

    // Calculate the scaling factor for converting to a normalised double
```

```java
    if (wavFile.validBits > 8)
    {
      // If more than 8 validBits, data is signed
      // Conversion required multiplying by magnitude of max positive value
      wavFile.floatOffset = 0;
      wavFile.floatScale = Long.MAX_VALUE >> (64 - wavFile.validBits);
    }
    else
    {
      // Else if 8 or less validBits, data is unsigned
      // Conversion required dividing by max positive value
      wavFile.floatOffset = 1;
      wavFile.floatScale = 0.5 * ((1 << wavFile.validBits) - 1);
    }

    // Finally, set the IO State
    wavFile.bufferPointer = 0;
    wavFile.bytesRead = 0;
    wavFile.frameCounter = 0;
    wavFile.ioState = IOState.WRITING;

    return wavFile;
  }

  public static WavFile openWavFile(File file) throws IOException, WavFileException
  {
    // Instantiate new Wavfile and store the file reference
    WavFile wavFile = new WavFile();
    wavFile.file = file;

    // Create a new file input stream for reading file data
    wavFile.iStream = new FileInputStream(file);

    // Read the first 12 bytes of the file
    int bytesRead = wavFile.iStream.read(wavFile.buffer, 0, 12);
    if (bytesRead != 12) throw new WavFileException("Not enough wav file bytes for header");

    // Extract parts from the header
    long riffChunkID = getLE(wavFile.buffer, 0, 4);
    long chunkSize = getLE(wavFile.buffer, 4, 4);
    long riffTypeID = getLE(wavFile.buffer, 8, 4);

    // Check the header bytes contains the correct signature
    if (riffChunkID != RIFF_CHUNK_ID) throw new WavFileException("Invalid Wav Header data, incorrect riff chunk ID");
    if (riffTypeID != RIFF_TYPE_ID) throw new WavFileException("Invalid Wav Header data, incorrect riff type ID");

    // Check that the file size matches the number of bytes listed in header
    if (file.length() != chunkSize+8) {
      throw new WavFileException("Header chunk size (" + chunkSize + ") does not match file size (" + file.length() + ")");
    }

    boolean foundFormat = false;
    boolean foundData = false;

    // Search for the Format and Data Chunks
    while (true)
    {
      // Read the first 8 bytes of the chunk (ID and chunk size)
      bytesRead = wavFile.iStream.read(wavFile.buffer, 0, 8);
      if (bytesRead == -1) throw new WavFileException("Reached end of file without finding format chunk");
      if (bytesRead != 8) throw new WavFileException("Could not read chunk header");

      // Extract the chunk ID and Size
      long chunkID = getLE(wavFile.buffer, 0, 4);
      chunkSize = getLE(wavFile.buffer, 4, 4);

      // Word align the chunk size
      // chunkSize specifies the number of bytes holding data. However,
      // the data should be word aligned (2 bytes) so we need to calculate
      // the actual number of bytes in the chunk
      long numChunkBytes = (chunkSize%2 == 1) ? chunkSize+1 : chunkSize;

```

```
      if (chunkID == FMT_CHUNK_ID)
      {
        // Flag that the format chunk has been found
        foundFormat = true;

        // Read in the header info
        bytesRead = wavFile.iStream.read(wavFile.buffer, 0, 16);

        // Check this is uncompressed data
        int compressionCode = (int) getLE(wavFile.buffer, 0, 2);
        if (compressionCode != 1) throw new WavFileException("Compression Code " + compressionCode + " not supported");

        // Extract the format information
        wavFile.numChannels = (int) getLE(wavFile.buffer, 2, 2);
        wavFile.sampleRate = getLE(wavFile.buffer, 4, 4);
        wavFile.blockAlign = (int) getLE(wavFile.buffer, 12, 2);
        wavFile.validBits = (int) getLE(wavFile.buffer, 14, 2);

        if (wavFile.numChannels == 0) throw new WavFileException("Number of channels specified in header is equal to zero");
        if (wavFile.blockAlign == 0) throw new WavFileException("Block Align specified in header is equal to zero");
        if (wavFile.validBits < 2) throw new WavFileException("Valid Bits specified in header is less than 2");
        if (wavFile.validBits > 64) throw new WavFileException("Valid Bits specified in header is greater than 64, this is greater than a long can hold");

        // Calculate the number of bytes required to hold 1 sample
        wavFile.bytesPerSample = (wavFile.validBits + 7) / 8;
        if (wavFile.bytesPerSample * wavFile.numChannels != wavFile.blockAlign)
          throw new WavFileException("Block Align does not agree with bytes required for validBits and number of channels");

        // Account for number of format bytes and then skip over
        // any extra format bytes
        numChunkBytes -= 16;
        if (numChunkBytes > 0) wavFile.iStream.skip(numChunkBytes);
      }
      else if (chunkID == DATA_CHUNK_ID)
      {
        // Check if we've found the format chunk,
        // If not, throw an exception as we need the format information
        // before we can read the data chunk
        if (foundFormat == false) throw new WavFileException("Data chunk found before Format chunk");

        // Check that the chunkSize (wav data length) is a multiple of the
        // block align (bytes per frame)
        if (chunkSize % wavFile.blockAlign != 0) throw new WavFileException("Data Chunk size is not multiple of Block Align");

        // Calculate the number of frames
        wavFile.numFrames = chunkSize / wavFile.blockAlign;

        // Flag that we've found the wave data chunk
        foundData = true;

        break;
      }
      else
      {
        // If an unknown chunk ID is found, just skip over the chunk data
        wavFile.iStream.skip(numChunkBytes);
      }
    }

    // Throw an exception if no data chunk has been found
    if (foundData == false) throw new WavFileException("Did not find a data chunk");

    // Calculate the scaling factor for converting to a normalised double
    if (wavFile.validBits > 8)
    {
      // If more than 8 validBits, data is signed
      // Conversion required dividing by magnitude of max negative value
      wavFile.floatOffset = 0;
```

```java
      wavFile.floatScale = 1 << (wavFile.validBits - 1);
    }
    else
    {
      // Else if 8 or less validBits, data is unsigned
      // Conversion required dividing by max positive value
      wavFile.floatOffset = -1;
      wavFile.floatScale = 0.5 * ((1 << wavFile.validBits) - 1);
    }

    wavFile.bufferPointer = 0;
    wavFile.bytesRead = 0;
    wavFile.frameCounter = 0;
    wavFile.ioState = IOState.READING;

    return wavFile;
  }

  // Get and Put little endian data from local buffer
  // ------------------------------------------------
  private static long getLE(byte[] buffer, int pos, int numBytes)
  {
    numBytes --;
    pos += numBytes;

    long val = buffer[pos] & 0xFF;
    for (int b=0 ; b<numBytes ; b++) val = (val << 8) + (buffer[--pos] & 0xFF);

    return val;
  }

  private static void putLE(long val, byte[] buffer, int pos, int numBytes)
  {
    for (int b=0 ; b<numBytes ; b++)
    {
      buffer[pos] = (byte) (val & 0xFF);
      val >>= 8;
      pos ++;
    }
  }

  // Sample Writing and Reading
  // --------------------------
  private void writeSample(long val) throws IOException
  {
    for (int b=0 ; b<bytesPerSample ; b++)
    {
      if (bufferPointer == BUFFER_SIZE)
      {
        oStream.write(buffer, 0, BUFFER_SIZE);
        bufferPointer = 0;
      }

      buffer[bufferPointer] = (byte) (val & 0xFF);
      val >>= 8;
      bufferPointer ++;
    }
  }

  private long readSample() throws IOException, WavFileException
  {
    long val = 0;

    for (int b=0 ; b<bytesPerSample ; b++)
    {
      if (bufferPointer == bytesRead)
      {
        int read = iStream.read(buffer, 0, BUFFER_SIZE);
        if (read == -1) throw new WavFileException("Not enough data available");
        bytesRead = read;
        bufferPointer = 0;
      }

      int v = buffer[bufferPointer];
      if (b < bytesPerSample-1 || bytesPerSample == 1) v &= 0xFF;
      val += v << (b * 8);
```

```java

      bufferPointer++;
    }

    return val;
  }

  // Integer
  // -------
  public int readFrames(int[] sampleBuffer, int numFramesToRead) throws IOException, WavFileException
  {
    return readFrames(sampleBuffer, 0, numFramesToRead);
  }

  public int readFrames(int[] sampleBuffer, int offset, int numFramesToRead) throws IOException, WavFileException
  {
    if (ioState != IOState.READING) throw new IOException("Cannot read from WavFile instance");

    for (int f=0 ; f<numFramesToRead ; f++)
    {
      if (frameCounter == numFrames) return f;

      for (int c=0 ; c<numChannels ; c++)
      {
        sampleBuffer[offset] = (int) readSample();
        offset++;
      }

      frameCounter++;
    }

    return numFramesToRead;
  }

  public int readFrames(int[][] sampleBuffer, int numFramesToRead) throws IOException, WavFileException
  {
    return readFrames(sampleBuffer, 0, numFramesToRead);
  }

  public int readFrames(int[][] sampleBuffer, int offset, int numFramesToRead) throws IOException, WavFileException
  {
    if (ioState != IOState.READING) throw new IOException("Cannot read from WavFile instance");

    for (int f=0 ; f<numFramesToRead ; f++)
    {
      if (frameCounter == numFrames) return f;

      for (int c=0 ; c<numChannels ; c++) sampleBuffer[c][offset] = (int) readSample();

      offset++;
      frameCounter++;
    }

    return numFramesToRead;
  }

  public int writeFrames(int[] sampleBuffer, int numFramesToWrite) throws IOException, WavFileException
  {
    return writeFrames(sampleBuffer, 0, numFramesToWrite);
  }

  public int writeFrames(int[] sampleBuffer, int offset, int numFramesToWrite) throws IOException, WavFileException
  {
    if (ioState != IOState.WRITING) throw new IOException("Cannot write to WavFile instance");

    for (int f=0 ; f<numFramesToWrite ; f++)
    {
      if (frameCounter == numFrames) return f;
```

```
			for (int c=0 ; c<numChannels ; c++)
			{
				writeSample(sampleBuffer[offset]);
				offset ++;
			}

			frameCounter ++;
		}

		return numFramesToWrite;
	}

	public int writeFrames(int[][] sampleBuffer, int numFramesToWrite) throws IOException, WavFileException
	{
		return writeFrames(sampleBuffer, 0, numFramesToWrite);
	}

	public int writeFrames(int[][] sampleBuffer, int offset, int numFramesToWrite) throws IOException, WavFileException
	{
		if (ioState != IOState.WRITING) throw new IOException("Cannot write to WavFile instance");

		for (int f=0 ; f<numFramesToWrite ; f++)
		{
			if (frameCounter == numFrames) return f;

			for (int c=0 ; c<numChannels ; c++) writeSample(sampleBuffer[c][offset]);

			offset ++;
			frameCounter ++;
		}

		return numFramesToWrite;
	}

	// Long
	// ----
	public int readFrames(long[] sampleBuffer, int numFramesToRead) throws IOException, WavFileException
	{
		return readFrames(sampleBuffer, 0, numFramesToRead);
	}

	public int readFrames(long[] sampleBuffer, int offset, int numFramesToRead) throws IOException, WavFileException
	{
		if (ioState != IOState.READING) throw new IOException("Cannot read from WavFile instance");

		for (int f=0 ; f<numFramesToRead ; f++)
		{
			if (frameCounter == numFrames) return f;

			for (int c=0 ; c<numChannels ; c++)
			{
				sampleBuffer[offset] = readSample();
				offset ++;
			}

			frameCounter ++;
		}

		return numFramesToRead;
	}

	public int readFrames(long[][] sampleBuffer, int numFramesToRead) throws IOException, WavFileException
	{
		return readFrames(sampleBuffer, 0, numFramesToRead);
	}

	public int readFrames(long[][] sampleBuffer, int offset, int numFramesToRead) throws IOException, WavFileException
	{
```

```
        if (ioState != IOState.READING) throw new IOException("Cannot read from WavFile instance");

        for (int f=0 ; f<numFramesToRead ; f++)
        {
            if (frameCounter == numFrames) return f;

            for (int c=0 ; c<numChannels ; c++) sampleBuffer[c][offset] = readSample();

            offset ++;
            frameCounter ++;
        }

        return numFramesToRead;
    }

    public int writeFrames(long[] sampleBuffer, int numFramesToWrite) throws IOException, WavFileException
    {
        return writeFrames(sampleBuffer, 0, numFramesToWrite);
    }

    public int writeFrames(long[] sampleBuffer, int offset, int numFramesToWrite) throws IOException, WavFileException
    {
        if (ioState != IOState.WRITING) throw new IOException("Cannot write to WavFile instance");

        for (int f=0 ; f<numFramesToWrite ; f++)
        {
            if (frameCounter == numFrames) return f;

            for (int c=0 ; c<numChannels ; c++)
            {
                writeSample(sampleBuffer[offset]);
                offset ++;
            }

            frameCounter ++;
        }

        return numFramesToWrite;
    }

    public int writeFrames(long[][] sampleBuffer, int numFramesToWrite) throws IOException, WavFileException
    {
        return writeFrames(sampleBuffer, 0, numFramesToWrite);
    }

    public int writeFrames(long[][] sampleBuffer, int offset, int numFramesToWrite) throws IOException, WavFileException
    {
        if (ioState != IOState.WRITING) throw new IOException("Cannot write to WavFile instance");

        for (int f=0 ; f<numFramesToWrite ; f++)
        {
            if (frameCounter == numFrames) return f;

            for (int c=0 ; c<numChannels ; c++) writeSample(sampleBuffer[c][offset]);

            offset ++;
            frameCounter ++;
        }

        return numFramesToWrite;
    }

    // Double
    // ------
    public int readFrames(double[] sampleBuffer, int numFramesToRead) throws IOException, WavFileException
    {
        return readFrames(sampleBuffer, 0, numFramesToRead);
    }

    public int readFrames(double[] sampleBuffer, int offset, int numFramesToRead) throws
```

```
    IOException, WavFileException
{
    if (ioState != IOState.READING) throw new IOException("Cannot read from WavFile instance");

    for (int f=0 ; f<numFramesToRead ; f++)
    {
        if (frameCounter == numFrames) return f;

        for (int c=0 ; c<numChannels ; c++)
        {
            sampleBuffer[offset] = floatOffset + (double) readSample() / floatScale;
            offset ++;
        }

        frameCounter ++;
    }

    return numFramesToRead;
}

public int readFrames(double[][] sampleBuffer, int numFramesToRead) throws IOException, WavFileException
{
    return readFrames(sampleBuffer, 0, numFramesToRead);
}

public int readFrames(double[][] sampleBuffer, int offset, int numFramesToRead) throws IOException, WavFileException
{
    if (ioState != IOState.READING) throw new IOException("Cannot read from WavFile instance");

    for (int f=0 ; f<numFramesToRead ; f++)
    {
        if (frameCounter == numFrames) return f;

        for (int c=0 ; c<numChannels ; c++) sampleBuffer[c][offset] = floatOffset + (double) readSample() / floatScale;

        offset ++;
        frameCounter ++;
    }

    return numFramesToRead;
}

public int readFramesWithOverlap(double[] sampleBuffer, int numFramesToRead, int overlap) throws IOException, WavFileException
{
    numFramesToRead = readFrames(sampleBuffer, 0, numFramesToRead);
    long coeff = frameCounter * overlap / numFramesToRead - overlap + 1;
    frameCounter = coeff * numFramesToRead / overlap;

    return numFramesToRead;
}


public int writeFrames(double[] sampleBuffer, int numFramesToWrite) throws IOException, WavFileException
{
    return writeFrames(sampleBuffer, 0, numFramesToWrite);
}

public int writeFrames(double[] sampleBuffer, int offset, int numFramesToWrite) throws IOException, WavFileException
{
    if (ioState != IOState.WRITING) throw new IOException("Cannot write to WavFile instance");

    for (int f=0 ; f<numFramesToWrite ; f++)
    {
        if (frameCounter == numFrames) return f;

        for (int c=0 ; c<numChannels ; c++)
        {
            writeSample((long) (floatScale * (floatOffset + sampleBuffer[offset])));
```

```
        offset++;
      }

      frameCounter++;
    }

    return numFramesToWrite;
  }

  public int writeFrames(double[][] sampleBuffer, int numFramesToWrite) throws IOException, WavFileException
  {
    return writeFrames(sampleBuffer, 0, numFramesToWrite);
  }

  public int writeFrames(double[][] sampleBuffer, int offset, int numFramesToWrite) throws IOException, WavFileException
  {
    if (ioState != IOState.WRITING) throw new IOException("Cannot write to WavFile instance");

    for (int f=0 ; f<numFramesToWrite ; f++)
    {
      if (frameCounter == numFrames) return f;

      for (int c=0 ; c<numChannels ; c++) writeSample((long) (floatScale * (floatOffset + sampleBuffer[c][offset])));

      offset++;
      frameCounter++;
    }

    return numFramesToWrite;
  }


  public void close() throws IOException
  {
    // Close the input stream and set to null
    if (iStream != null)
    {
      iStream.close();
      iStream = null;
    }

    if (oStream != null)
    {
      // Write out anything still in the local buffer
      if (bufferPointer > 0) oStream.write(buffer, 0, bufferPointer);

      // If an extra byte is required for word alignment, add it to the end
      if (wordAlignAdjust) oStream.write(0);

      // Close the stream and set to null
      oStream.close();
      oStream = null;
    }

    // Flag that the stream is closed
    ioState = IOState.CLOSED;
  }

  public void display()
  {
    display(System.out);
  }

  public void display(PrintStream out)
  {
    out.printf("File: %s\n", file);
    out.printf("Channels: %d, Frames: %d\n", numChannels, numFrames);
    out.printf("IO State: %s\n", ioState);
    out.printf("Sample Rate: %d, Block Align: %d\n", sampleRate, blockAlign);
    out.printf("Valid Bits: %d, Bytes per sample: %d\n", validBits, bytesPerSample);
  }

  public static void main(String[] args)
```

```
  {
    if (args.length < 1)
    {
      System.err.println("Must supply filename");
      System.exit(1);
    }

    try
    {
      for (String filename : args)
      {
        WavFile readWavFile = openWavFile(new File(filename));
        readWavFile.display();

        long numFrames = readWavFile.getNumFrames();
        int numChannels = readWavFile.getNumChannels();
        int validBits = readWavFile.getValidBits();
        long sampleRate = readWavFile.getSampleRate();

        WavFile writeWavFile = newWavFile(new File("out.wav"), numChannels, numFrames, validBits, sampleRate);

        final int BUF_SIZE = 5001;

//          int[] buffer = new int[BUF_SIZE * numChannels];
//          long[] buffer = new long[BUF_SIZE * numChannels];
        double[] buffer = new double[BUF_SIZE * numChannels];

        int framesRead = 0;
        int framesWritten = 0;

        do
        {
          framesRead = readWavFile.readFrames(buffer, BUF_SIZE);
          framesWritten = writeWavFile.writeFrames(buffer, BUF_SIZE);
          System.out.printf("%d %d\n", framesRead, framesWritten);
        }
        while (framesRead != 0);

        readWavFile.close();
        writeWavFile.close();
      }

      WavFile writeWavFile = newWavFile(new File("out2.wav"), 1, 10, 23, 44100);
      double[] buffer = new double[10];
      writeWavFile.writeFrames(buffer, 10);
      writeWavFile.close();
    }
    catch (Exception e)
    {
      System.err.println(e);
      e.printStackTrace();
    }
  }
}
```

Листинг 3: WavFileException.java

```
package com.external;

/**
 * Exception for WavFile Class
 * http://www.labbookpages.co.uk
 *
 * @author A.Greensted
 */

public class WavFileException extends Exception
{
  public WavFileException()
  {
    super();
  }

  public WavFileException(String message)
  {
```

```
        super(message);
    }

    public WavFileException(String message, Throwable cause)
    {
        super(message, cause);
    }

    public WavFileException(Throwable cause)
    {
        super(cause);
    }
}
```

Листинг 4: LightLevel.java

```
package com.sunradio.core;

import com.sunradio.math.Scale;

import static java.lang.Math.*;
import java.util.Random;

/**
 * LightLevel describes how we get an array of data with current light level
 * @author V.Kremneva
 */
public class LightLevel {

    /**
     * Get randomised light level
     *
     * @param amount of how many measurement of light level we want
     * @return scaled to [0;1] array of light values
     */
    private static double[] getFakeLightLevel(int amount, Double minAmplitude, Double maxAmplitude) {

        //fake it 'till you make it
        Integer[] lightLevel = new Integer[amount];
        for (int i = 0; i < amount; i++)
            lightLevel[i] = (int)(sin(2 * PI * i / 100)*1000); //for smoothness


        return Scale.run(lightLevel, minAmplitude, maxAmplitude);
    }

    /**
     * Get light level from Arduino //TODO: specify
     *
     * @param amount of how many measurement of light level we want
     * @return scaled to [0;1] array of light values
     */
    static double[] getLightLevel(int amount, Double minAmplitude, Double maxAmplitude) {
        return getFakeLightLevel(amount, minAmplitude, maxAmplitude);
    }

    /**
     * Get light level
     * @param values an array to whom get light level
     * @return light level for 'values'
     */
    public static double[] getLightLevel(double[] values) {
        double maxVal, minVal;
        maxVal = Double.MIN_VALUE; minVal = Double.MAX_VALUE;
        for (double val: values) {
            if (maxVal > val) maxVal = val;
            if (maxVal < val) minVal = val;
        }

        return getLightLevel(values.length, minVal, maxVal);
    }

    private static int getAverageFakeLevel() {
        Random random = new Random();
        return random.nextInt();
```

```
    }

    static int getAverageLightLevel(double[] values) {
        return getAverageFakeLevel();
    }
}
```

Листинг 5: SunRadio.java

```
package com.sunradio.core;

import com.external.WavFile;
import com.sunradio.math.DFTInverse;
import com.sunradio.math.DFTStraight;
import com.sunradio.math.Filter;
import com.sunradio.math.ToneModulation;

import java.io.File;
/*
import javax.sound.sampled.AudioInputStream;
import javax.sound.sampled.AudioSystem;
import javax.sound.sampled.Clip;
*/

/**
 * @author V.Kremneva
 */
public class SunRadio {
    private final int FRAMES = 2048; //amount of frames to read
    private final int OVERLAP = 16; //coefficient of overlap

    private WavFile wavInput; //input file
    private WavFile wavOutput; //output file

    private int bufferIndAmount; //amount of indexes in 'buffer' array needed to read data to
    private int overlapIndAmount; //amount of indexes to work with overlap
    private int offset; //amount of new frames read in each step of cycle
    private int outputBufferIndAmount; //amount of indexes in am array to write
    private long wholeIndAmount; //amount of pieces to read in whole file

    /**
     * Open file and set some fields depending on it
     *
     * @param inputPath path to file to open
     */
    private void openWavFile(String inputPath) {
        try {

            wavInput = WavFile.openWavFile(new File(inputPath));

        } catch (Exception e) {
            System.err.println(e.toString());
        }

        int numChannels = wavInput.getNumChannels();
        bufferIndAmount = FRAMES * numChannels;
        overlapIndAmount = OVERLAP * numChannels;
        offset = FRAMES / OVERLAP;
        outputBufferIndAmount = bufferIndAmount + overlapIndAmount;
        wholeIndAmount = wavInput.getNumFrames() * numChannels;
    }

    /**
     * Create empty output file like input file but stretched
     *
     * @param outputPath path where to create output file
     */
    private void createStretchedOutputFile(String outputPath, int stretch) {
        try {

            wavOutput = WavFile.newWavFile(new File(outputPath),
                    wavInput.getNumChannels(), wavInput.getNumFrames() * stretch,
                    wavInput.getValidBits(), wavInput.getSampleRate());

        } catch (Exception e) {
            System.err.println(e.toString());
```

```java
        }
    }

    private void run() {
        double[] buffer = new double[bufferIndAmount];
        double[] outputBuffer = new double[outputBufferIndAmount];
        double[] outputWindowFunction;
        int lightLevel, frames_read;
        DFTStraight transformable;
        ToneModulation toneModulation;

        transformable = new DFTStraight();
        toneModulation = new ToneModulation(bufferIndAmount);

        int counter = 0;
        try {
            do {
                //read next 'FRAMES' into buffer -- amplitudes(t)
                frames_read = wavInput.readFramesWithOverlap(buffer, FRAMES, OVERLAP);

                //get current level of light
                lightLevel = LightLevel.getAverageLightLevel(buffer);

                for (int i = 0; i < overlapIndAmount; i++) {
                    //apply window filter. first and last 'offset' goes without filter
                    if (!(wavInput.getFrameCounter() == offset) && !(wavInput.getFrameCounter() == (wholeIndAmount - offset))) {
                        buffer = Filter.apply(buffer, Filter.BlackmanNuttall(bufferIndAmount));
                    }

                    //run Fourier transform
                    transformable.run(buffer);

                    //stretch in 'light level' times
                    toneModulation.setCurrentData(transformable);
                    transformable.setData(toneModulation.stretch(lightLevel));

                    //run inverse Fourier transform
                    buffer = DFTInverse.run(transformable.getData());

                    //apply output window function
                    outputWindowFunction = Filter.getOutputWindowFunc(Filter.BlackmanNuttall(bufferIndAmount));
                    buffer = Filter.apply(buffer, outputWindowFunction);

                    for (int j = 0; j < bufferIndAmount; j++)
                        outputBuffer[j + i] += buffer[j];

                    //read next 'FRAMES' into buffer -- amplitudes(t)
                    frames_read = wavInput.readFramesWithOverlap(buffer, FRAMES, OVERLAP);
                }

                //write data to new .waw file
                wavOutput.writeFrames(outputBuffer, FRAMES);

                //move data for overlap
                outputBuffer = move(outputBuffer, overlapIndAmount);

                toneModulation.setPreviousData(transformable);
                counter++;
            } while (frames_read != 0);

            //todo: adjust volume
            //todo: fasten velocity of playback
            //play(outputPath);


        } catch (Exception e) {
            System.err.println(e.toString());
        }
    }

    private void closeFiles() {
        try {

```

```
                wavInput.close();
                wavOutput.close();

            } catch (Exception e) {
                System.err.println(e.toString());
            }
        }

        /**
         *  Move data to the left with filling with 0
         * @param data data to move
         * @param offset amount of steps to move
         * @return array with nulls in the end and 'data' values moved on offset
         */
        public static double[] move(double[] data, int offset) {
            double[] result = new double[data.length];

            System.arraycopy(data, offset, result, 0, data.length - offset);

            return result;
        }

        /*private static void play(String pathname) {
            try {
                Clip c = AudioSystem.getClip();
                AudioInputStream ais = AudioSystem.getAudioInputStream(new File(pathname));

                c.open(ais);
                c.loop(0);

                Thread.sleep(1000);
            } catch (Exception e) {
                System.err.println(e.toString());
            }
        }*/

        public static void main(String[] args) {

           if (args.length < 2) throw new IllegalArgumentException("As the arguments of the 
        ↪ program " +
                       "input path and output path are needed.");

            SunRadio radio = new SunRadio();

            radio.openWavFile(args[0]);

            radio.run();

            radio.closeFiles();
        }
}
```

Листинг 6: AM.java

```
package com.sunradio.math;

/**
 * Amplitude modulation
 * @author V.Kremneva
 */
public class AM {

    /**
     * Modulate values according to conditions with the coefficient of modulation -0.65.
     * Assume 'values' as amplitude values therefore perform an Amplitude Modulation.
     *
     * @param values an array with amplitude values to modulate
     * @param conditions an array of modulation conditions for each amplitude value
     * @return a double array of modulated values of amplitudes
     * @throws IllegalArgumentException if amount of conditions is less than amount of values
     */
    public static double[] modulate(double[] values, double[] conditions) {
        return modulate(values, conditions, -0.65);
    }

    /**
```

```java
     * Modulate values according to conditions.
     * Assume 'values' as amplitude values therefore perform an Amplitude Modulation.
     *
     * @param values an array with amplitude values to modulate
     * @param conditions an array of modulation conditions for each amplitude value
     * @param modulationCoeff modulation coefficient which picks up by trial and error
     * @return a double array of modulated values of amplitudes
     * @throws IllegalArgumentException if amount of conditions is less than amount of values
     */
    private static double[] modulate(double[] values, double[] conditions, double modulationCoeff)
            throws IllegalArgumentException {

        if (values.length > conditions.length) {
            throw new IllegalArgumentException("A size of an array with conditions " +
                    "should be equal or more than size of an array of values.\n" +
                    "Conditions size = " + conditions.length + ". Values size = " + values.length);
        }

        double[] modulated;
        modulated = new double[values.length];

        double maxCond;
        maxCond = conditions[0];
        for (int i = 1; i < values.length; i++)
            if (conditions[i] > maxCond) maxCond = conditions[i];

        for (int i = 0; i < values.length; i++)
            modulated[i] = values[i] * (1 + modulationCoeff * conditions[i] / Math.abs(maxCond));

        return modulated;
    }
}
```

Листинг 7: DFTStraight.java

```java
package com.sunradio.math;

import com.external.Complex;
import java.util.Arrays;
import static java.lang.Math.*;

/**
 * Straight discrete Fourier transform.
 *
 * @author V.Kremneva
 */
public class DFTStraight {

    private Complex[] data; //Complex data
    private double maxAmplitude; //maximum value of real amplitude in 'data' array
    private double minAmplitude; //minimum value of real amplitude in 'data' array
    private int size; //size of 'data' array
    private boolean isTransformed; //flag whether was 'data' transformed by DFT or not

    public DFTStraight() {
        maxAmplitude = 0.0;
        minAmplitude = 0.0;
        size = 0;
        isTransformed = false;
    }

    public double getMaxAmplitude() {
        return maxAmplitude;
    }

    public double getMinAmplitude() {
        return minAmplitude;
    }

    public boolean isTransformed() {
        return isTransformed;
    }

```

```
    public Complex[] getData() {
        return data;
    }

    public int getSize() {
        return size;
    }

    /** Get a double phases from 'data' array.
     *
     * @return a double array which contains phase values
     */
    public double[] getPhases() {
        double[] phases = new double[size];
        double allowance;
        for (int i = 0; i < size; i++) {

            if (data[i].re() > 0) allowance = 0;
            else if (data[i].im() > 0) allowance = PI;
            else allowance = -PI;

            phases[i] = allowance + atan(data[i].im() / data[i].re());
        }

        return phases;
    }

    /** Get a phase value on specific harmonic
     *
     * @param n frequency value of harmonic
     * @return double value of phase of harmonic
     */
    public double getPhase(int n) {
        double allowance;
        if (data[n].re() > 0) allowance = 0;
        else if (data[n].im() > 0) allowance = PI;
        else allowance = -PI;

        return allowance + atan(data[n].im() / data[n].re());
    }

    public static double getPhase(Complex[] data, int n) {
        double allowance;
        if (data[n].re() > 0) allowance = 0;
        else if (data[n].im() > 0) allowance = PI;
        else allowance = -PI;

        return allowance + atan(data[n].im() / data[n].re());
    }

    /** Get a double amplitudes from 'data' array.
     *
     * @return a double array which contains amplitude values
     */
    public double[] getAmplitudes() {
        double[] amplitudes = new double[size];

        for (int i = 0; i < size; i++)
            amplitudes[i] = data[i].abs() / ((size - 1) * 2); // '-1)*2' due to cutting in half

        return amplitudes;
    }

    static double[] getAmplitudes(Complex[] data) {
        double[] amplitudes = new double[data.length];

        for (int i = 0; i < data.length; i++)
            amplitudes[i] = data[i].abs() / ((data.length - 1) * 2); // '-1)*2' due to cutting in half

        return amplitudes;
    }

    static double[] getPhases(Complex[] data) {
        double[] phases = new double[data.length];
```

```java
        double allowance;
        for (int i = 0; i < phases.length; i++) {

            if (data[i].re() > 0) allowance = 0;
            else if (data[i].im() > 0) allowance = PI;
            else allowance = -PI;

            phases[i] = allowance + atan(data[i].im() / data[i].re());
        }

        return phases;
    }

    /** Get an amplitude value on specific harmonic
     *
     * @param n frequency value of harmonic
     * @return double value of amplitude of harmonic
     */
    public double getAmplitude(int n) {
        return data[n].abs() / ((size - 1) * 2);
        // '-1)*2' due to cutting in half
    }

    public void setData(Complex[] newData) {
        size = newData.length;
        data = Arrays.copyOf(newData, size);

        double max, min;
        max = Double.MIN_VALUE; min = Double.MAX_VALUE;
        for (int i = 0; i < size; i++) {
            if (this.getAmplitude(i) > max) max = this.getAmplitude(i);
            if (this.getAmplitude(i) < min) min = this.getAmplitude(i);
        }
        maxAmplitude = max; minAmplitude = min;
    }

    /**
     * Cut Complex array in two pieces.
     * We need this because the periods of the input data become split into "positive"
     * and "negative" frequency complex components. As a result, only half of array
     * contains data we are interested in and the rest of array is just a reflection with
     * opposite sign.
     *
     * @param dataToCut Complex data we want to be cut
     */
    private void cutDataInHalf(Complex[] dataToCut) {
        size = size / 2 + 1; //'+1' to include center value

        data = new Complex[size];
        data = Arrays.copyOf(dataToCut, size);
    }

    /** Run transform with search of max and min value of amplitude.
     *
     * @param buffer an array of the magnitudes
     * @return a Complex array which contains amplitude and phase values
     * @throws IllegalArgumentException if buffer is empty
     */
    public Complex[] run(double[] buffer) throws IllegalArgumentException {

        size = buffer.length;
        Double realAmplitude;
        Complex cBuffer, expDegree, tempData[];
        tempData = new Complex[size];

        if (size == 0) throw new IllegalArgumentException("Size of a buffer cannot be < 1.\n size = " + size);

        maxAmplitude = Double.MIN_VALUE;
        minAmplitude = Double.MAX_VALUE;
        for (int i = 0; i < size; i++) {
            tempData[i] = new Complex(0, 0);

            for (int j = 0; j < size; j++) {
                cBuffer = new Complex(buffer[j]);
                expDegree = new Complex(0, -2 * PI * j * i / size);
```

```java
                tempData[i] = tempData[i].add(cBuffer.mult(expDegree.exp()));
            }

            realAmplitude = tempData[i].abs() / size;
            if (realAmplitude > maxAmplitude) maxAmplitude = realAmplitude;
            if (realAmplitude < minAmplitude) minAmplitude = realAmplitude;
        }

        isTransformed = true;

        cutDataInHalf(tempData);

        return data;
    }

    /**
     * Change phase values in 'data' without changing amplitudes
     *
     * @param newPhases an array of phase values we want to apply
     */
    void applyNewPhases(double[] newPhases) {
        double a, b, allowance;
        for (int i = 0; i < size; i++) {
            if (data[i].re() > 0) allowance = 0;
            else if (data[i].im() > 0) allowance = -PI;
            else allowance = PI;

            a = data[i].abs() / sqrt(1 + pow(tan(newPhases[i] + allowance), 2.0));
            b = a * tan(newPhases[i] + allowance);

            //we get 'b' from equation for phase and 'a' from my condition:
            //i want the real amplitudes be the same

            data[i] = new Complex(a, b);
        }
    }

    static Complex[] applyNewPhases(double[] newPhases, Complex[] oldData) {
        double a, b, allowance;
        Complex[] result = new Complex[newPhases.length];
        for (int i = 0; i < newPhases.length; i++) {
            if (oldData[i].re() > 0) allowance = 0;
            else if (oldData[i].im() > 0) allowance = -PI;
            else allowance = PI;

            a = oldData[i].abs() / sqrt(1 + pow(tan(newPhases[i] + allowance), 2.0));
            b = a * tan(newPhases[i] + allowance);

            //we get 'b' from equation for phase and 'a' from my condition:
            //i want the real amplitudes be the same

            result[i] = new Complex(a, b);
        }
        return result;
    }

    /**
     * Change amplitude values in 'data' without changing phases
     *
     * @param newAmplitudes an array of amplitude values we want to apply
     */
    void applyNewAmplitudes(double[] newAmplitudes) {
         int sign;
         double a, b;
         for (int i = 0; i < size; i++) {
             if (this.getPhase(i) < 0) sign = -1;
             else sign = 1;
             b = pow(data[i].im(), 2.0) * pow(newAmplitudes[i], 2.0) * pow(size, 2.0);
             b = b / (pow(data[i].re(), 2.0) + pow(data[i].im(), 2.0));
             b = sqrt(b);

             a = sign * sqrt(pow(newAmplitudes[i], 2.0) * pow(size, 2.0) - pow(b, 2.0));

            //we get 'a' from equation for real amplitude and 'b' from my condition:
            //i want the real phase be the same

```

```
                data[i] = new Complex(a, b);
            }
    }

    static Complex[] applyNewAmplitudes(double[] newAmplitudes, Complex[] oldData) {
        int sign;
        double a, b;
        Complex[] result = new Complex[newAmplitudes.length];
        for (int i = 0; i < newAmplitudes.length; i++) {
            if (getPhase(oldData, i) < 0) sign = -1;
            else sign = 1;
            b = pow(oldData[i].im(), 2.0) * pow(newAmplitudes[i], 2.0) * pow(newAmplitudes.length, 2.0);
            b = b / (pow(oldData[i].re(), 2.0) + pow(oldData[i].im(), 2.0));
            b = sqrt(b);

            a = sign * sqrt(pow(newAmplitudes[i], 2.0) * pow(newAmplitudes.length, 2.0) - pow(b, 2.0));

            //we get 'a' from equation for real amplitude and 'b' from my condition:
            //i want the real phase be the same

            result[i] = new Complex(a, b);
        }

        return result;
    }
}
```

Листинг 8: DFTInverse.java

```
package com.sunradio.math;

import com.external.Complex;

import java.util.Arrays;

import static java.lang.Math.PI;

/**
 * Inverse discrete Fourier transform.
 * @author V.Kremneva
 */
public class DFTInverse {

    /**
     * Restore frequency complex components from just one half.
     *
     * @param dataToRestore array of data to restore
     * @return full spectrum Complex array
     */
    private static Complex[] restoreData(Complex[] dataToRestore) {

        int oldSize = dataToRestore.length;
        int newSize = oldSize * 2 - 2;
        Complex[] result = new Complex[newSize];

        System.arraycopy(dataToRestore, 0, result, 0, oldSize);

        for (int i = oldSize; i < newSize; i++)
            result[i] = dataToRestore[newSize - i].conj();

        return result;
    }

    /** Run transform.
     *
     * @param transformed a Complex array which contains amplitude and phase values
     * @return an array of amplitudes
     */
    public static double[] run(Complex[] transformed){

        Complex[] data;
        data = restoreData(transformed);

        int size = data.length;
```

```
        Complex exp_degree, magnitude;
        double[] result = new double[size];

        for (int i = 0; i < size; i++) {
            magnitude = new Complex(0, 0);

            for (int j = 0; j < size; j++) {
                exp_degree = new Complex(0, 2 * PI * j * i / size);
                magnitude = magnitude.add(data[j].mult(exp_degree.exp()));
            }

            result[i] = magnitude.re() / size;
        }

        return result;
    }
}
```

Листинг 9: Filter.java

```
package com.sunradio.math;

import static java.lang.Math.*;

/**
 * Filter functions
 * @author V.Kremneva
 */
public class Filter {

    /**
     * Window function of Blackman-Nuttall
     *
     * @param size is the length of array to be windowed
     * @return an array of values of this window
     */
    public static double[] BlackmanNuttall(int size) {
        double[] result = new double[size];

        //constants from formula of -BlackmanNuttall window
        final double A0 = 0.3635819;
        final double A1 = 0.4891775;
        final double A2 = 0.1365995;
        final double A3 = 0.0106411;

        double firstCos, secondCos, thirdCos;
        for (int i = 0; i < size; i++) {
            firstCos = A1 * cos((2 * PI * i) / (size - 1));
            secondCos = A2 * cos((4 * PI * i) / (size - 1));
            thirdCos = A3 * cos((6 * PI * i) / (size - 1));

            result[i] = A0 - firstCos + secondCos - thirdCos;
        }

        return result;
    }

    /**
     * Window function of Blackman-Nuttall
     *
     * @param n point where value of this function is needed
     * @param size is the length of array to be windowed
     * @return value of this window function in 'n'
     */
    public static double BlackmanNuttall(int n, int size) {
        //constants from formula of -BlackmanNuttall window
        final double A0 = 0.3635819;
        final double A1 = 0.4891775;
        final double A2 = 0.1365995;
        final double A3 = 0.0106411;

        double firstCos, secondCos, thirdCos;
        firstCos = A1 * cos((2 * PI * n) / (size - 1));
        secondCos = A2 * cos((4 * PI * n) / (size - 1));
        thirdCos = A3 * cos((6 * PI * n) / (size - 1));

```

```java
        return A0 - firstCos + secondCos - thirdCos;
    }

    /**
     * Apply window function to an array of values
     *
     * @param amplitudes an array to be windowed
     * @param windowFunc an array with values of window function
     * @return an array with transformed 'amplitudes' according to 'windowFunc'
     * @throws IllegalArgumentException if length of 'windowFunc' is less than length of 'amplitudes'
     */
    public static double[] apply(double[] amplitudes, double[] windowFunc)
            throws IllegalArgumentException {

        int size = amplitudes.length;

        if (size > windowFunc.length) throw new IllegalArgumentException("Length of window function is too small");

        double[] result = new double[size];
        for (int i = 0; i < size; i++) {
            result[i] = amplitudes[i] * windowFunc[i];
        }
        return result;
    }

    /**
     * Remove noise from amplitudes array
     *
     * @param toDenoise array to remove noise from
     * @return sort of clean array
     */
    public static double[] denoise(double[] toDenoise) {
        final double DENOISE_COEFF = 0.1; //got it by trial and error
        double maxAmplitude, eps;

        maxAmplitude = toDenoise[0];
        for (double val: toDenoise)
            if(val > maxAmplitude) maxAmplitude = val;

        eps = maxAmplitude * DENOISE_COEFF;
        for (int i = 0; i < toDenoise.length; i++)
            if (abs(toDenoise[i]) < eps) toDenoise[i] = 0.0;

        return toDenoise;
    }

    /**
     * Get function to filter output data before writing to file
     *
     * @param inputWindowFunc function used to filter input data
     * @return function to filter output data
     */
    public static double[] getOutputWindowFunc(double[] inputWindowFunc) {
        double[] outputWindowFunc = new double[inputWindowFunc.length];
        double sum = 0.0;

        for (double val: inputWindowFunc)
            sum += val * val;

        for (int i = 0; i < outputWindowFunc.length; i++)
            outputWindowFunc[i] = inputWindowFunc[i] / sum;

        return outputWindowFunc;
    }
}
```

Листинг 10: Interpolation.java

```java
package com.sunradio.math;

/**
 * Class to interpolate.
 *
 * @author V.Kremneva
```

```
 */
class Interpolation {
    /**
     * Perform linear interpolation by two points
     * @param x0 x from the first point
     * @param y0 f(x) from the first point
     * @param x1 x from the second point
     * @param y1 f(x) from the second point
     * @param x x from the point we are interested in
     * @return f(x) from the point we are interested in
     */
    static double linearByX(double x0, double y0, double x1, double y1, double x) {
        return y0 + (x - x0) * (y1 - y0) / (x1 - x0);
    }

    /**
     * Perform linear interpolation by two points
     * @param x0 x from the first point
     * @param y0 f(x) from the first point
     * @param x1 x from the second point
     * @param y1 f(x) from the second point
     * @param y f(x) from the point we are interested in
     * @return x from the point we are interested in
     */
    static double linearByY(double x0, double y0, double x1, double y1, double y) {
        return x0 + (y - y0) * (x1 - x0) / (y1 - y0);
    }
}
```

Листинг 11: Scale.java

```
package com.sunradio.math;

/**
 * Helps to scale data
 * @author V.Kremneva
 */
public class Scale {

    /**
     * Scale numeric data
     *
     * @param arr is an array of data to scale
     * @param from is the lower bound of scaling
     * @param to is the upper bound of scaling
     * @return scaled array of data
     * @throws IllegalArgumentException if the data array is empty
     */
    public static<T extends java.lang.Number> double[] run(T[] arr, double from, double to)
            throws IllegalArgumentException {

        int size = arr.length;
        if (size < 1) throw new IllegalArgumentException("Array␣of␣the␣values␣cannot␣be␣empty"
    ↪ );
        if (from > to) {
            double temp = from;
            from = to; to = temp;
        }

        double maxLevel = arr[0].doubleValue();
        double minLevel = arr[0].doubleValue();
        double current;
        for (int i = 1; i < size; i++) {
            current = arr[i].doubleValue();

            if (current > maxLevel) maxLevel = current;
            else if (current < minLevel) minLevel = current;
        }

        double step = (to - from) / (maxLevel - minLevel);

        double [] scaledLightLevel = new double [size];
        for (int i = 0; i < size; i++)
            scaledLightLevel[i] = (arr[i].doubleValue() - minLevel) * step + from;

        return scaledLightLevel;
```

```
    }
}
```

Листинг 12: ToneModulation.java

```
package com.sunradio.math;

import com.external.Complex;

/**
 * Tone modulation.
 *
 * @author V.Kremneva
 */
public class ToneModulation {
    private double[] previousPhases; //values of previous phases
    private double[] previousAmplitudes; //values of previous amplitudes
    private double[] currentPhases; //values of current phases
    private double[] currentAmplitudes; //values of current amplitudes

    private int length; //length of all of the arrays

    private boolean previousIsSet; //indicates whether previous data was set
    private boolean currentIsSet; //indicates whether current data was set

    public ToneModulation() {
        previousIsSet = false;
        currentIsSet = false;
    }

    public ToneModulation(int size) {
        length = size;

        previousPhases = new double[size];
        previousAmplitudes = new double[size];

        previousIsSet = true;
        currentIsSet = false;
    }

    public void setPreviousData(DFTStraight prevData) {
        previousAmplitudes = prevData.getAmplitudes();
        previousPhases = prevData.getPhases();

        previousIsSet = true;
    }

    public void setCurrentData(DFTStraight currentData) {
        currentAmplitudes = currentData.getAmplitudes();
        currentPhases = currentData.getPhases();

        currentIsSet = true;
    }

    /**
     * Stretch data in N times
     *
     * @param coefficient in how many times to stretch
     * @return stretched data
     */
    public Complex[] stretch(int coefficient) throws ToneModulationException {
        if (!previousIsSet || !currentIsSet)
            throw new ToneModulationException("Previous and current data must be set");

        int newSize = length * coefficient;
        Complex[] result = new Complex[newSize];
        double[] newPhases = new double[newSize];
        double[] newAmplitudes = new double[newSize];
        double[] velocity = new double[length];

        for (int i = 0; i < length; i++)
            velocity[i] = currentPhases[i] - previousPhases[i];

        for (int i = 0; i < length; i++)
            for (int j = 0; j < coefficient; j++)
                newPhases[i + j] = currentPhases[i] + velocity[i];
```

```

        result = DFTStraight.applyNewPhases(newPhases, result);

        for (int i = 0; i < length; i++)
            for (int j = 0; j < coefficient; j++)
                newAmplitudes[i + j] = Interpolation.linearByX(0, previousAmplitudes[i],
    ↪ coefficient, currentAmplitudes[i], j);

        result = DFTStraight.applyNewAmplitudes(newAmplitudes, result);

        return result;
    }
}
```

Листинг 13: ToneModulationException.java

```
package com.sunradio.math;

/**
 * Exception for ToneModulation class
 *
 * @author V.Kremneva
 */
public class ToneModulationException extends Exception {

    public ToneModulationException(String message) {
        super(message);
    }
}
```

Листинг 14: SunRadioTest.groovy

```
package com.sunradio.core

import com.external.WavFile
import com.sunradio.math.AM
import com.sunradio.math.DFTInverse
import com.sunradio.math.DFTStraight
import com.sunradio.math.Filter

class SunRadioTest extends GroovyTestCase {
    final FRAMES = 100
    final EPS = 0.00001
    DFTStraight transformable = new DFTStraight()

    int numChannels, indAmount, framesRead
    double[] buffer, lightLevel, modulated, amplitudes


    void testRun() {
        try {
           // inputPath = "C:\\Users\\LEV\\IdeaProjects\\SunRadio\\launch.wav";
           // outputPath = "C:\\Users\\LEV\\IdeaProjects\\SunRadio\\new1.wav";

        } catch (Exception e) {
            System.err.println(e.toString())
        }
    }

    void testMove() {
        int offset = 3
        double[] before = [1, 2, 3, 4, 5, 6, 7]
        double[] afterExpected = [4, 5, 6, 7, 0, 0, 0]
        double[] afterActual

        afterActual = SunRadio.move(before, offset)

        for (int i = 0; i < before.length; i++)
            assertEquals(afterExpected[i], afterActual[i])
    }
}
```

Листинг 15: DFTStraightTest.groovy

```
package com.sunradio.math

import static java.lang.Math.*

class DFTStraightTest extends GroovyTestCase {

    final EPS = 0.00001
    final ITERATIONS = 100
    final SPLIT = 100
    final NUMBER = 80.0

    double[] buffer = new double[ITERATIONS]
    double[] result = new double[ITERATIONS]
    DFTStraight dftStraight = new DFTStraight()

    //f(t) = sin(t)
    void testSin() {

        //Split the sinus period in SPLIT pieces and take the sinus value in each of them
        for (int i = 0; i < ITERATIONS; i++)
            buffer[i] = sin(2 * PI * i / SPLIT)

        dftStraight.run(buffer)
        result = dftStraight.getAmplitudes()

        int amount = 0, size = dftStraight.getSize()
        for (int i = 0; i < size; i++)
            if (result[i] > EPS)
                amount++

        assertEquals(1, amount)
    }

    //f(t) = NUMBER*sin(t)
    //Test passes with any number
    void testConstMultSin() {

        //Split the sinus period in SPLIT pieces and take the sinus value in each of them
        for (int i = 0; i < ITERATIONS; i++)
            buffer[i] = NUMBER * sin(2 * PI * i / SPLIT)

        dftStraight.run(buffer)
        result = dftStraight.getAmplitudes()

        int amount = 0, size = dftStraight.getSize()
        for (int i = 0; i < size; i++)
            if (result[i] > EPS)
                amount++

        assertEquals(1, amount)
    }

    //f(t) = sin(NUMBER*t)
    void testSinMultConst() throws IllegalArgumentException{

        if ((NUMBER > SPLIT) || (NUMBER < SPLIT / 2))
            throw new IllegalArgumentException("Number should be less than SPLIT and more than SPLIT/2 due to sinus period")

        //Split the sinus period in SPLIT pieces and take the sinus value in each of them
        for (int i = 0; i < ITERATIONS; i++)
            buffer[i] = sin(NUMBER * 2 * PI * i / SPLIT)

        dftStraight.run(buffer)
        result = dftStraight.getAmplitudes()

        int amount = 0, size = dftStraight.getSize()
        for (int i = 0; i < size; i++)
            if (result[i] > EPS)
                amount++

        assertEquals(1, amount)
    }

    //f(t) = NUMBER*sin(t) + sin(Number*t)
    void testTwoSinuses() {
```

```

        if ((NUMBER > SPLIT) || (NUMBER < SPLIT / 2))
            throw new IllegalArgumentException("Number should be less than SPLIT and more than SPLIT/2 due to sinus period")

        //Split the sinus period in SPLIT pieces and take the sinus value in each of them
        for (int i = 0; i < ITERATIONS; i++)
            buffer[i] = NUMBER*sin(2 * PI * i / SPLIT) + sin(NUMBER * 2 * PI * i / SPLIT)

        dftStraight.run(buffer)
        result = dftStraight.getAmplitudes()

        int amount = 0, size = dftStraight.getSize()
        for (int i = 0; i < size; i++)
            if (result[i] > EPS)
                amount++

        assertEquals(2, amount)
    }

    void testApplyNewAmplitudes() {
        double[] modulation
        DFTStraight dftStraightApplied = new DFTStraight()

        for (int i = 0; i < ITERATIONS; i++)
            buffer[i] = sin(2 * PI * i / SPLIT)

        dftStraight.run(buffer)

        modulation = new double[dftStraight.size]
        Random random = new Random()
        for (int i = 0; i < dftStraight.size; i++)
            modulation[i] = abs(random.nextDouble())

        dftStraightApplied.setData(dftStraight.getData())
        dftStraightApplied.applyNewAmplitudes(modulation)

        double oldPhase, newPhase
        for (int i = 0; i < dftStraight.size; i++) {
            oldPhase = dftStraight.getPhase(i)
            newPhase = dftStraightApplied.getPhase(i)

            assert abs(oldPhase - newPhase)  < EPS
        }
    }

    void testApplyNewPhases() {
        double[] modulation
        DFTStraight dftStraightApplied = new DFTStraight()

        for (int i = 0; i < ITERATIONS; i++)
            buffer[i] = sin(2 * PI * i / SPLIT)

        dftStraight.run(buffer)

        modulation = new double[dftStraight.size]
        Random random = new Random()
        for (int i = 0; i < dftStraight.size; i++)
            modulation[i] = abs(random.nextDouble())

        dftStraightApplied.setData(dftStraight.getData())
        dftStraightApplied.applyNewPhases(modulation)

        double oldAmplitude, newAmplitude
        for (int i = 0; i < dftStraight.size; i++) {
            oldAmplitude = dftStraight.getAmplitude(i)
            newAmplitude = dftStraightApplied.getAmplitude(i)

            assert abs(oldAmplitude - newAmplitude) < EPS
       }
    }
}
```

Листинг 16: DFTInverseTest.groovy

```
package com.sunradio.math
```

```

import static java.lang.Math.PI
import static java.lang.Math.sin

class DFTInverseTest extends GroovyTestCase {
    final EPS = 0.00001
    final ITERATIONS = 100
    final SPLIT = 100
    final NUMBER = 80.0

    double[] income = new double[ITERATIONS]
    DFTStraight outcome_straight = new DFTStraight()
    double[] outcome_inverse = new double[ITERATIONS]

    // f(t) = sin(t)
    public void testSin() {
        //Split the sinus period in SPLIT pieces and take the sinus value in each of them
        for (int i = 0; i < ITERATIONS; i++)
            income[i] = sin(2 * PI * i / SPLIT)

        outcome_straight.run(income)
        outcome_inverse = DFTInverse.run(outcome_straight.getData())

        double difference;
        int amount = 0;
        for (int i = 0; i < ITERATIONS; i++) {
            difference = outcome_inverse[i] - income[i]
            if (difference > EPS) amount++
        }

        assertEquals(0, amount)
    }

    //f(t) = NUMBER*sin(t) + sin(Number*t)
    public void testTwoSin() throws IllegalArgumentException {
        if ((NUMBER > SPLIT) || (NUMBER < SPLIT / 2))
            throw new IllegalArgumentException("Number should be less than SPLIT and more than SPLIT/2 due to sinus period")

        //Split the sinus period in SPLIT pieces and take the sinus value in each of them
        for (int i = 0; i < ITERATIONS; i++)
            income[i] = NUMBER*sin(2 * PI * i / SPLIT) + sin(NUMBER * 2 * PI * i / SPLIT)

        outcome_straight.run(income)
        outcome_inverse = DFTInverse.run(outcome_straight.getData())

        double difference;
        int amount = 0;
        for (int i = 0; i < ITERATIONS; i++) {
            difference = outcome_inverse[i] - income[i]
            if (difference > EPS) amount++
        }

        assertEquals(0, amount)
    }
}
```

Листинг 17: FilterTest.groovy

```
package com.sunradio.math

import static java.lang.Math.PI
import static java.lang.Math.sin

class FilterTest extends GroovyTestCase {
    final ITERATIONS = 100
    final SPLIT = 80
    final EPS = 0.00001

    double[] buffer = new double[ITERATIONS]
    double[] winFunc = new double[ITERATIONS]
    double[] applied = new double[ITERATIONS]

    void testApply() {

        //Split the sinus period in SPLIT pieces and take the sinus value in each of them
```

```
        for (int i = 0; i < ITERATIONS; i++)
            buffer[i] = sin(2 * PI * i / SPLIT)

        winFunc = Filter.BlackmanNuttall(ITERATIONS)
        applied = Filter.apply(buffer, winFunc)

        for (int i = 0; i < ITERATIONS; i++)
            assert applied[i] - (buffer[i] * winFunc[i]) < EPS
    }

    void testGetOutputFilter() {
        double[] inputWindowFunc = Filter.BlackmanNuttall(ITERATIONS)
        double[] outputWindowFunc = Filter.getOutputWindowFunc(inputWindowFunc)
        double sum = 0.0

        for (int i = 0; i < ITERATIONS; i++)
            sum += inputWindowFunc[i]*outputWindowFunc[i]

        assertEquals(1.0, sum)
    }
}
```

Листинг 18: ScaleTest.groovy

```
package com.sunradio.math

class ScaleTest extends GroovyTestCase {
    void testRun() {
        final EPS = 0.00000001
        def arr = [5, 8, 7, 2] as Integer[]
        def from = 0.0
        def to = 1.0

        def outputExpected = [0.5, 1.0, 5 / 6, 0.0] as double[]

        double[] outputValue = Scale.run(arr, from, to)
        for (int i = 0; i < 4; i++)
            assertTrue((outputExpected[i] - outputValue[i]) < EPS)
    }
}
```