# Fast Fourier Transform

The goal of this assignment is to write a program that implements the

- Discrete Fourier Transform (DFT),
- Inverse Discrete Fourier Transform (IDFT),
- Fast Fourier Transform (FFT) and
- Inverse Fast Fourier Transform (IFFT),

which is a very powerful tool in signal processing.

## Resources

In addition to this handout, the assignment webpage also contains:

- An archive of template code, including a GUI you may use. The GUI can already do the following for you (1) load a .wav file and store it as a waveform (list of numbers) in the program; (2) save a waveform (list of numbers) as a .wav file; (3) display a waveform (in the time domain) or a spectrum (in the frequency domain). The code includes the following files:
  - The **SoundWaveForm.java** file provides the GUI, which includes saving and loading a .wav file, and displaying the waveform (in time domain) and the spectrum (in frequency domain, which will be obtained by DFT/FFT). **You will have to fill in the DFT, IDFT, FFT and IFFT methods**.
  - The **ComplexNumber.java** file is a class for complex numbers, which will be needed for FFT and IFFT. It contains several basic operations, such as constructor, setter and getter. **You will have to fill in the operations between complex numbers (e.g. addition, subtraction, etc) that are necessary for DFT/IDFT/FFT/IFFT**.
  - The **WaveformLoader.java** file contains the **doSave()** method for saving a waveform (list of numbers) to a .wav file, and the **doLoad()** method for loading a .wav file to a list of numbers.
  - The **ecs100.jar** library for some helper methods.
- A set of .wav files.
- A marking guide.

# Fast Fourier Transform Pipeline

Discrete/Fast Fourier Transform (DFT/FFT) can be used to analyse a waveform (e.g. a series of numbers). Specifically, DFT/FFT transforms a waveform from the time domain to the frequency domain, while Inverse DFT/FFT (IDFT/IFFT) is from the frequency domain to the time domain. Visualising the spectrum in the frequency domain can show different aspects that are hidden in the time domain.

For this assignment, you must write a program that performs DFT/FFT and IDFT/IFFT, using the algorithms described in the lectures.

The template code already provides the methods for loading and saving a .wav file, and displaying a waveform in the time domain, as well as displaying a spectrum in the frequency domain. You will need to write the DFT, IDFT, FFT and IFFT methods (ideally creating a new **FourierTransform.java** class file for them), as well as the supporting methods, that is, the operations between complex numbers, in the **ComplexNumber.java** file.

# Outline of your pipeline

Ideally, your program should have the following functionalities:

- Read a .wav file and display it (already provided in the template code).
- Apply DFT to the waveform in the time domain and transform it to the frequency domain (*NOTE: this can be very slow*).
- Apply FFT to the waveform in the time domain and transform it to the frequency domain.
- Apply IDFT to the series in the frequency domain and transform it back to the time domain (*NOTE: this can be very slow*).
- Apply IFFT to the series in the frequency domain and transform it back to the time domain.
- (*Challenge*) Modify the magnitude of the spectrum to change the waveform.

# Stages and marking

See the provided marking guide for a more detailed breakdown of the marks for different stages.

**Stage 1** – Complete the **SoundWaveform.dft()** and **SoundWaveform.idft()** methods to implement the DFT and IDFT based on the formulas given in the lecture. You need to implement the necessary operations between complex numbers in the **ComplexNumber.java** file as supporting methods.

*Hint*: You can test the DFT and IDFT on the small .wav files. On large sound files they will be VERY SLOW.

*Hint*: Below are three simple test cases to help you debug your code.

```
// Case 1:
waveform = [1,2,3,4,5,6,7,8];
spectrum = [
    36.0+0.0i,
    -4.000000000000002+9.65685424949238i,
    -4.000000000000002+3.9999999999999987i,
    -4.0+1.6568542494923788i,
    -4.0-2.4492935982947065E-15i,
    -3.999999999999999-1.656854249492381i,
    -3.999999999999998-4.000000000000001i,
    -3.999999999999995-9.65685424949238i
];
```

```
// Case 2:
waveform = [1,2,1,2,1,2,1,2];
spectrum = [
    12.0+0.0i,
    -4.688471024089327E-16-1.2246467991473537E-16i,
    -4.898587196589413E-16-2.449293598294707E-16i,
    -2.2391774257946197E-16-1.2246467991473537E-16i,
    -4.0-9.797174393178826E-16i,
    2.2391774257946197E-16-1.2246467991473525E-16i,
    4.898587196589413E-16-2.4492935982947054E-16i,
    4.688471024089326E-16-1.2246467991473517E-16i
];
```

```
// Case 3:
waveform = [1,2,3,4,4,3,2,1];
spectrum = [
    20.0+0.0i,
    -5.82842712474619-2.414213562373096i,
    -6.123233995736766E-16-6.123233995736766E-16i,
    -0.1715728752538097-0.4142135623730945i,
    0.0-1.2246467991473533E-15i,
    -0.1715728752538097+0.4142135623730937i,
    6.123233995736766E-16-6.123233995736765E-16i,
    -5.828427124746191+2.4142135623730936i
];
```

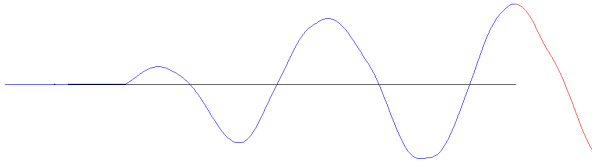
**NB**: The numbers end with E-15(16)(17) are very close to zero, so can be treated as zero. No need to worry about the rounding errors, as long as your corresponding numbers are

also close to zero.

**Stage 2** – Complete the **SoundWaveform.fft()** method to implement the FFT based on the algorithm described in the lecture.

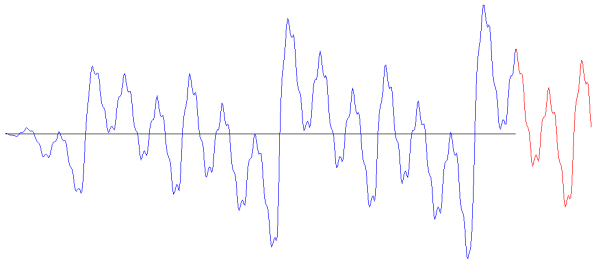
*Hint*: You have to cut the tail of the waveform to make its size equal to some power of two, so you can run FFT properly.

*Hint*: The following figures show the waveforms and corresponding spectrum obtained by FFT for the small datasets.

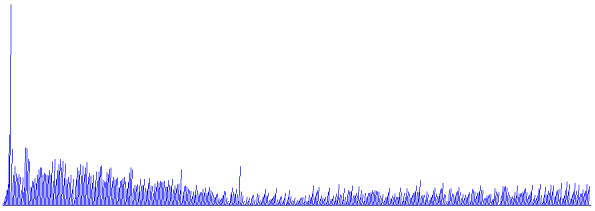


(a) The waveform of `Brass`.

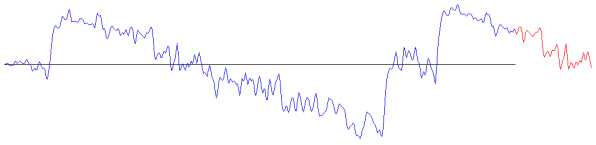
(b) The spectrum of `Brass`.



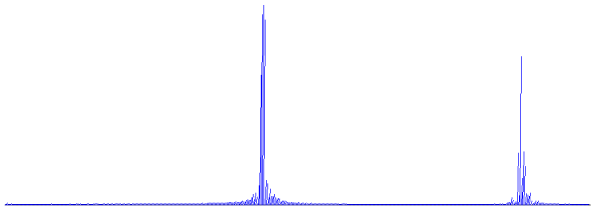
(a) The waveform of `Sympy`.



(b) The spectrum of `Sympy`.



(a) The waveform of `Xplore`.



(b) The spectrum of `Xplore`.

**Stage 3** – Complete the **SoundWaveform.ifft()** method to implement the IFFT. It uses the divide-and-conquer algorithm which is very similar to the FFT. You will need to figure out the difference between the two methods and get the IFFT correct.

*Hint*: You can compare the results of FFT (IFFT) and DFT (IDFT) to see if they can obtain the same results. You can also compare the computational time of FFT (IFFT) and DFT (IDFT) on the small .wav files to understand the efficiency advantage of FFT (IFFT).

*Hint*: The loading, FFT and IFFT process can take time. To avoid breaking the program, you should wait until the text screen shows "Loading/FFT/IFFT completed!"

*Hint*: If you compare the results of DFT and FFT and expect to obtain the same spectrum, you have to make sure that **the input waveform of DFT and FFT are the same**. Specifically, you MUST cut the tail of the waveform before doing DFT as well.

**Stage 4** – *(Challenge)* Add a **SoundWaveform.doMouse()** method to implement the functionality that modify the magnitude of different frequencies of the spectrum so as to change the waveform after the Inverse Fast Fourier Transform correct. The functionality should work as follows:

- In the display of the spectrum, click any location using the mouse. The x-axis value of the clicked location will be the selected frequency, and the y-axis value of the clicked location will be its new magnitude.
- Extract the selected frequency, and calculate its new magnitude.
- Calculate the corresponding term in the spectrum that is affected by the magnitude change.

# Data files

There are five .wav files that you can use to test your program:

- `Brass.wav`: a basic small sound file.
- `Sympy.wav`: another basic small sound file.
- `Xplore.wav` a third basic small sound file.
- `Forest.wav`: a big sound file ($\sim$10MB), with some sounds recorded in a forest.
- `Sea.wav` a big sound file ($\sim$10MB) recorded by a sea.

For each file, you can load it using the "Load" button in the GUI, which will also display the waveform (in the time domain) in the window.

After loading the file, you can test your FFT and IFFT methods. If everything is correct, you should be able to display exactly the same waveform after applying FFT and then IFFT.

You can save the file with a different file name. The file will be a truncated sound file of the original file, due to the truncation to be the power of 2 in the FFT.

You can play the saved file to see if the sound is the same as the original file (easier to figure out for the `Forest.wav` and `Sea.wav` files).

# To Submit

You should submit the following things:

- All the source code for your program, including the template code. **Please make sure you do this**, without it we cannot give you any marks. **Again: submit all your `.java` files**.
- Any other files your program needs to run that *aren't* the data files provided.
- A report on your program to help the marker understand the code. The report should:
  - describe what your code does and doesn't do.
  - describe any bugs that you have not been able to resolve.

  The report should be clear, but it does not have to be fancy – very plain formatting is all that is needed. It must be either a `txt` or a `pdf` file. It does not need to be long, but you need to help the marker see what you did.

# Implementation hints

- The waveform loaded from a .wav file is an array list of `Double`. However, the FFT method takes a list of `ComplexNumber`. Therefore, you need to transform a `Double` to a `ComplexNumber`, which is easy to do.
- Also, after IFFT, you will need to obtain the waveform as a list of `Double`. This requires converting a list of `ComplexNumber` to a list of `Double`. A simple way is to get its real part, since the imaginary part is supposed to be (almost) zero.
- The FFT algorithm works only when the size of the series equals some power of 2. Therefore, cut the tail of the input list before doing FFT.