CSE 220: Signals and Linear Systems
Offline Assignment: Discrete Fourier Transform & FFT

Department of Computer Science and Engineering

Due Date: 23-02-2026

General Instructions

- All implementations must use **Object-Oriented Programming (OOP)**.
- Use of any built-in Fourier Transform functions is **strictly prohibited**, including `np.fft`, `scipy.fft`, etc.
- You are allowed to use `numpy` for array manipulations and `scipy.io.wavfile` for audio I/O.
- **Starter Code:** The provided code is fully runnable. It launches the UI for both tasks. However, the buttons and processing logic will produce no result (or silence) until you implement the underlying mathematical frameworks.
- You must separate your implementation into the files specified in the task breakdown.
- **LLM-generated code is strictly prohibited; any evidence of it will result in -100%.**

Required Libraries

You will need to install the following Python libraries. Ensure they are installed in your environment:

1. `numpy` - For numerical computations.
2. `scipy` - For loading WAV files.
3. `sounddevice` - For audio playback.

To install them, run:

```
pip install numpy scipy sounddevice
```

Note: on Linux, you may also need to install `libportaudio2` via your package manager (e.g., `sudo apt-get install libportaudio2`) for `sounddevice` to work.

Task 0: Object-Oriented Discrete Signal Framework

Objective

Design a reusable object-oriented framework for modeling discrete-time signals and computing the Discrete Fourier Transform (DFT) using the standard $\mathcal{O}(N^2)$ summation approach. This framework will serve as the backbone for the applications in Tasks 2 and 3.

Theoretical Background

The Discrete Fourier Transform (DFT) transforms a finite sequence of N complex numbers $x[n]$ into a sequence of N complex numbers $X[k]$.

Analysis Equation (DFT):

$$X[k] = \sum_{n=0}^{N-1} x[n] \cdot e^{-j\frac{2\pi}{N}kn} \quad \text{for } k = 0, \ldots, N-1$$

Synthesis Equation (Inverse DFT):

$$x[n] = \frac{1}{N} \sum_{k=0}^{N-1} X[k] \cdot e^{j\frac{2\pi}{N}kn} \quad \text{for } n = 0, \ldots, N-1$$

Class Specifications

Implement the following classes in `discrete_framework.py`:

1. `DiscreteSignal` **(Base Class)**

- **Attributes**:
  - `data`: A numpy array representing the discrete signal samples (can be complex).
- **Methods**:
  - `__init__(data)`: Initializes the signal.
  - `pad(new_length)`: Zero-pads or truncates the signal to `new_length`.

2. `DFTAnalyzer`

- **Purpose**: Computes the DFT using the naive summation method ($\mathcal{O}(N^2)$).
- **Methods**:
  - `compute_dft(signal)`: Returns the frequency domain representation $X[k]$.
  - `compute_idft(spectrum)`: Returns the time domain signal $x[n]$.

Task 1: Fourier Epicycles (The "Doodling" App)

Overview

You will implement the backend for a 2D drawing application. When a user draws a continuous loop on the canvas, your program will interpret the drawing as a complex signal $z[n] = x[n] + jy[n]$. By computing the DFT of this path, you can reconstruct the drawing using a sum of rotating vectors (epicycles).

Requirements

Using the provided `task1.py` UI framework:

1. **Signal Extraction**: convert the list of 2D coordinates (x, y) from the user's drawing into a complex discrete signal using your `DiscreteSignal` class.
2. **Compute DFT**: Compute the Discrete Fourier Transform of the complex signal using your `DFTAnalyzer` class.
3. **Reconstruction**: Implement the logic in `update_frame` to reconstruct the path dynamically. The position of the "pen" at time t is given by:

$$z(t) \approx \sum_{k} X[k] \cdot e^{j\frac{2\pi}{N}kt}$$

4. **Drawing Epicycles**: Visualise the rotating vectors (epicycles). You are provided with a helper method `draw_epicycle(x, y, radius)` in the starter code. Use the magnitude $|X[k]|$ as the radius and the phase $\angle X[k]$ to determine the direction.
5. **Animation**: The starter code handles the loop; you must complete the logic to update the frame at each time step.

The visual should look something like this.

Task 2: Audio Equalizer

Overview

Implement a 5-band audio equalizer using the provided Tkinter interface in `task2.py`.

Requirements

Provided Components

The starter code handles the following:

- **User Interface**: Sliders for the 5 frequency bands, a DFT/FFT toggle, and Load/Play/Stop buttons.
- **Audio I/O**: Loading the WAV file (`load_file`) and playing audio via `sounddevice`.

Implementation

You must implement the `process_and_play` method to perform the following Digital Signal Processing (DSP) chain:

1. **Input**: Access the loaded audio data from `self.original_audio`.
2. **Block Processing**: Divide the audio signal into chunks (e.g., 1024 samples).
3. **Frequency Analysis**: Apply your DFT implementation to each chunk to get the frequency spectrum.
4. **Filtering**:
   - Divide the frequency spectrum into 5 equalizer bands (Low, Low-Mid, Mid, High-Mid, High).
   - Apply the gain factor from the corresponding UI sliders to the magnitude of the signal in each band.
5. **Reconstruction**: Apply the Inverse Transform (IDFT/IFFT) to convert the filtered spectrum back to the time domain.
6. **Output**: Stitch the processed chunks together and assign the result to `self.processed_audio`.
7. **Playback**: Play the processed audio using `sd.play`.

Task 3: Fast Fourier Transform (FFT)

Motivation

The $\mathcal{O}(N^2)$ complexity of the naive DFT makes the drawing app lag for long strokes and the audio equalizer slow for high-quality audio. You must implement the Fast Fourier Transform (FFT) to make it fast.

Requirements

1. Implement the **Radix-2 Decimation-in-Time (DIT) Cooley-Tukey Algorithm** in a new class `FastFourierTransform` (inheriting `DFTAnalyzer`).
2. **Complexity**: Must achieve $\mathcal{O}(N \log N)$.
3. **UI Integration**: Update your `run_transform` (Task 1) and `process_and_play` (Task 2) methods to check the state of the UI toggle button.
   - If **DFT** is selected, use `DFTAnalyzer`.
   - If **FFT** is selected, use `FastFourierTransform`.

Handling Non-Power-of-2 Inputs

The Radix-2 algorithm requires N to be a power of 2.

- **For Drawing (Task 2)**: You must implement an **Interpolation** method. Resample the user's drawing to the nearest power of 2 using linear interpolation.
- **For Audio (Task 3)**: Zero-padding is acceptable, or simply choose a chunk size that is already a power of 2 (e.g., 2048).

Bonus (10 Marks)

Implement a **Mixed-Radix FFT** or/and **Bluestein's Algorithm** to handle arbitrary input sizes N efficiently without padding/interpolation.

Marks Distribution

Section	**Criteria**	**Marks**
Task 0	Class Structure & Naive DFT/IDFT	25
Task 1	Complex Signal Mapping & Epicycle Logic	20
Task 2	Audio Chunking & Frequency Binning	20
Task 3	FFT Implementation (Radix-2)	25
Task 3	Interpolation Logic (for Drawing)	10
Total		**100**
Bonus	Arbitrary N FFT	+20