Project Report: C# Chromatic Guitar Tuner

Author: Joel Marji

Date: December 2nd, 2025

Platform: Windows (.NET Framework / .NET Core)

Class: CET 423

Abstract:

This project presents the design and implementation of a precision chromatic instrument tuner application developed using C#, Windows Forms and .NET. The system integrates a signal processing pipeline to ensure accuracy. The application utilizes the SDL2 library for low latency audio acquisition and MathNet.Numerics for spectral analysis using the Fast Fourier Transform (FFT) algorithm. Algorithmic implementations include a Hann windowing function for signal smoothing, the Harmonic Product Spectrum (HPS) method for fundamental frequency isolation, and Quadratic Interpolation to achieve ~0.1Hz frequency resolution. The resulting software delivers real-time, visual feedback through a GUI with user-defined microphone input and concert pitch adjustment, successfully demonstrating a robust solution for musical instrument tuning in a Windows desktop environment.

Contents

1. Problem Statement

The Challenge

Tuning a Guitar by ear is a skill that most Guitar players do not posess due to the widespread use of digital instrument tuners. How difficult would it be to create an **accurate** tuner application from scratch, and what Digital Signal Processing techniques need to be used?

The Objective

To develop a desktop GUI application that captures real-time audio from a microphone and identifies the musical pitch with high precision. The system must filter out background noise, suppress misleading harmonics, and provide immediate visual feedback to the user.

2. Budget

This project relies primarily on open-source software and existing hardware. Below is the breakdown of resources used.

Item	Description	Cost
Development IDE	Visual Studio Community 2022	$0.00 (Free License)
Language & Frameworks	C#, .NET, Windows Forms	$0.00 (Open Source)
Libraries	SDL2 (Audio Capture), MathNet.Numerics (FFT)	$0.00 (MIT/Zlib Licenses)
Hardware	PC with Microphone Input	Existing Asset
Test Equipment	Acoustic Guitar, Signal Generator App	Existing Asset
Total Monetary Cost		**$0.00**

3. System Architecture & Schematic Diagrams

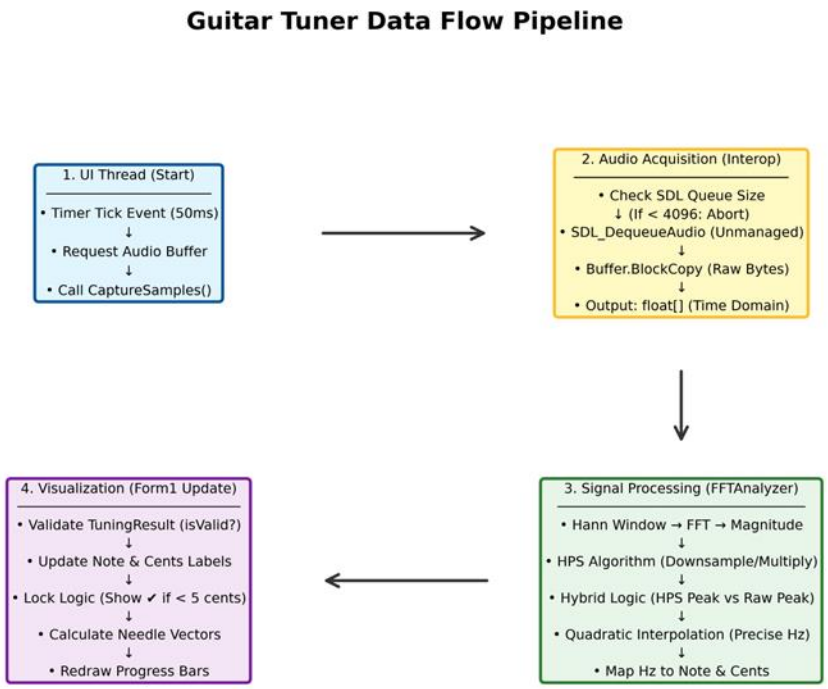
3.1 Component Architecture

The application follows a modular design to separate hardware interaction, mathematical analysis, and user interface.

- **AudioProcessor (Hardware Layer):** Handles the initialization of the SDL2 unmanaged library, enumerates audio devices, and manages the circular buffer for audio capture.
- **FFTAnalyzer (Logic Layer):** Contains the mathematical core. It holds the logic for HPS (Harmonic Product Spectrum), windowing, and the conversion of Frequency (Hz) to Musical Pitch (Cents/Semitones).
- **Form1 (Presentation Layer):** Handles the event loop (Timer), user input (Adjustable A4 frequency), and visual rendering (Progress bars/Needles).

3.2 Data Flow Diagram (The Signal Pipeline)

This diagram illustrates how raw audio is transformed into user-readable tuning data.



4. Code Samples

This section details the specific implementation of the core algorithms and interface logic used in the application.

4.1 Signal Acquisition using SDL

The AudioProcessor class interfaces with the SDL2 library to handle the circular buffer. The CaptureSamples method is responsible for dequeuing raw data from the audio device and converting it into a float array for analysis.

```
public float[] CaptureSamples(int bufferSamples)
{
    if (!isDeviceOpen)
    {
        return null;
    }

    SDL.SDL_PauseAudioDevice(micDeviceID, 0);

    int floatSize = sizeof(float);
    byte[] rawBuffer = new byte [bufferSamples * floatSize];

    // Return null if not enough data is queued.
    if (SDL.SDL_GetQueuedAudioSize(micDeviceID) < rawBuffer.Length)
    {
        return null;
    }

    float[] floatBuffer = new float[bufferSamples];

    if (SDL.SDL_GetQueuedAudioSize(micDeviceID) >= rawBuffer.Length)
    {
        unsafe
        {
            fixed (byte* ptr = rawBuffer)
            {
                SDL.SDL_DequeueAudio(micDeviceID, (IntPtr)ptr,
(uint)rawBuffer.Length);
            }
        }
        Buffer.BlockCopy(rawBuffer, 0, floatBuffer, 0, rawBuffer.Length);
    }
    return floatBuffer;
}
```

4.2 Signal Processing

All signal processing logic is contained within the FFTAnalyzer class in the file FFTAnalyzer.cs. The processing pipeline transforms time-domain audio into a precise frequency value.

4.2.1 Hann Window

To mitigate spectral leakage caused by processing discrete chunks of audio, a Hann Window function is applied to the buffer. This mathematically tapers the amplitude at the start and end of the sample buffer to zero, to ensure accurate FFT analysis.

```
float[] windowedSamples = new float[samples.Length];
for (int i=0; i<samples.Length; i++)
{
    // Calculate hann window multiplier
    double multiplier = 0.5 * (1 - Math.Cos(2 * Math.PI * i /(samples.Length - 1)));

    windowedSamples[i] = (float)(samples[i] * multiplier);
}
```

4.2.2 FFT Algorithm (Fast Fourier Transform)

The heart of the application, the FFT algorithm converts time domain data into frequency domain magnitudes. The magnitudes array can then be sorted and used to find the dominant frequency of the audio sample.

```
Complex[] fftBuffer = new Complex[samples.Length];
for (int i = 0; i < samples.Length; i++)
{
    fftBuffer[i] = new Complex(windowedSamples[i], 0);
}

// Perform FFT
Fourier.Forward(fftBuffer, FourierOptions.Matlab);

double[] magnitudes = new double[samples.Length / 2];
for (int i = 0; i < magnitudes.Length; i++)
{
    magnitudes[i] = fftBuffer[i].Magnitude;
}
```

4.2.3 Harmonic Product Spectrum (HPS)

To distinguish the fundamental frequency from harmonic overtones, the HPS algorithm multiplies the signal by downsampled copies of itself. A fallback mechanism is implemented to ensure sensitivity is maintained for quiet signals.

```
for (int i = 1; i<magnitudes.Length/3; i++) // Check 3 harmonics
{
    //"Squish" samples
    hpsMagnitudes[i] *= magnitudes[i * 2];
    hpsMagnitudes[i] *= magnitudes[i * 3];
}
```

4.2.4 Quadratic Interpolation

FFT bins are discrete steps (approx 10Hz wide), Quadratic Interpolation fits a parabola to the peak bin and its neighbors to calculate the fractional offset, significantly improving tuning precision.

```
double leftMag = magnitudes[indexToUse - 1];
double rightMag = magnitudes[indexToUse + 1];

double p double offset =
(0.5 * (leftMag - rightMag)) / (leftMag - (2 * peakMag) + rightMag);
double preciseIndex = indexToUse + offset;eakMag = magnitudes[indexToUse];
```

4.2.5 Frequency Calculation

After the buffer goes through all previous steps, the frequency is calculated using the formula
Freq = (K/N)*R

```
double dominantFreq = ((double)sampleRate / samples.Length) * preciseIndex;
return dominantFreq;
```

4.2.6 Note Mapping and Cents Calculation

The “Bridge” between the UI and the backend logic, the GetNote method contains logic that handles the logarithmic conversion of frequencies to MIDI note numbers, and deviation from the perfect note in cents.

```
double noteNumber = 12 * Math.Log((freq / BaseFrequency), 2) + 69;
int roundedNoteNumber = (int)Math.Round(noteNumber);
int cents = (int)((noteNumber - roundedNoteNumber) * 100);
```

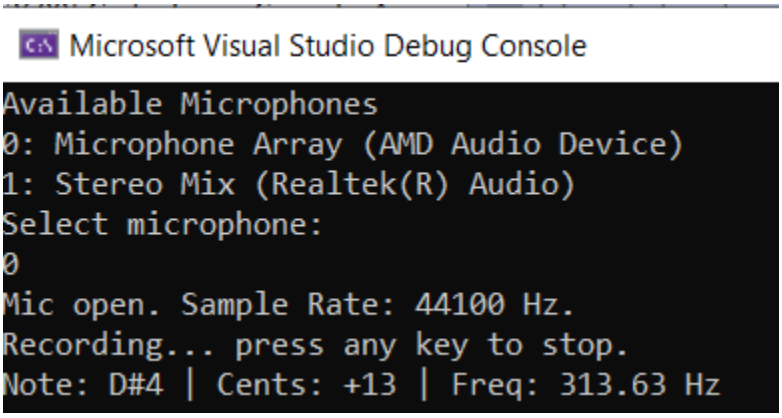
4.3 User Interface

The user interface interprets the frequency data into visual feedback. The project contains a console application and a GUI application.

4.3.1 Console Application

The console application is text based and uses a single while loop to control program flow. It uses a carriage return (\r) to overwrite the current line, creating a flicker free, real time update loop.

```
while (!Console.KeyAvailable)
{
    float[] samples = audio.CaptureSamples(4096);
    if (samples != null && samples.Length > 0)
    {
        TuningResult result = analyzer.GetNote(samples);
        string sign;
        if (result.CentsDeviation >= 0)
        {
            // If the number is positive (sharp), add a "+"
            sign = "+";
        }
        else
        {
            sign = "";
        }

        string output = $"Note: {result.NoteName} | Cents:
{sign}{result.CentsDeviation} | Freq: {result.Frequency:F} Hz";
        string finalOutput = output.PadRight(Console.BufferWidth - 1);
        Console.Write($"\r{finalOutput}");
    }
    // Avoid tight cpu cycle
    System.Threading.Thread.Sleep(50);
}
```



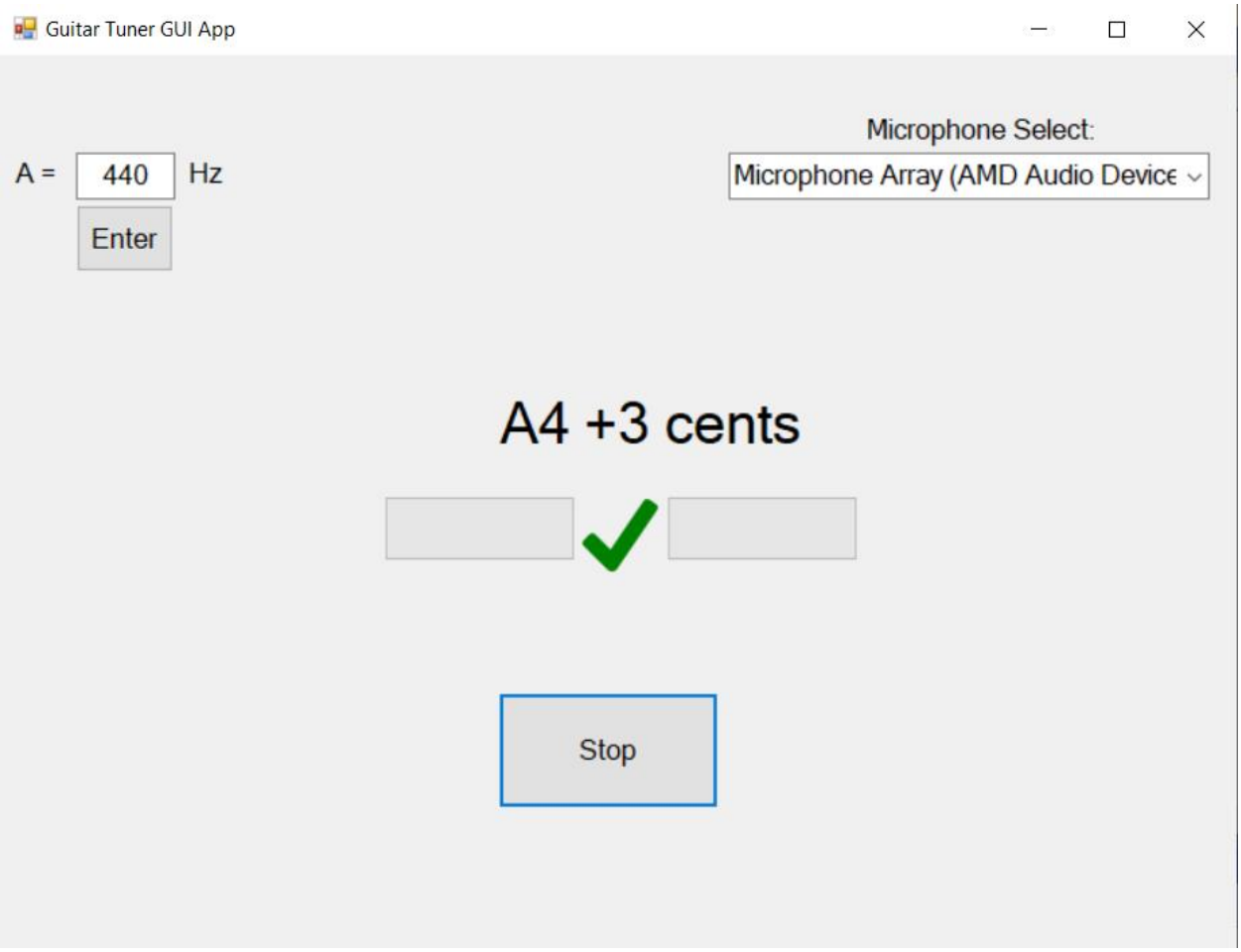
4.3.2 GUI Application

The Windows Forms application uses a Timer Tick event to drive the UI.

```
// Screen variables are updated and audio buffer is analyzed with every tick
private void analysisTimer_Tick(object sender, EventArgs e)
{
    float[] samples = audio.CaptureSamples(4096); // Collect audio sample buffer
    if (samples == null || samples.Length == 0)
    {
        return;
    }

    TuningResult result = analyzer.GetNote(samples); // Pass in audio samples
    if (result.isValid)
    {
        noteLbl.Text = result.NoteName;
        string sign = result.CentsDeviation >= 0 ? "+" : "";
        centsLbl.Text = $"{sign}{result.CentsDeviation} cents";
        // Set icon
        if (result.CentsDeviation > 5 || result.CentsDeviation < -5)
        {
            CheckImage.Visible = false;
            xImage.Visible = true;
        }
        else
        {
            CheckImage.Visible = true;
            xImage.Visible = false;
        }

        tunerNeedleFlat.Value = 0;
        tunerNeedleSharp.Value = 0;

        int deviation = result.CentsDeviation;

        if (deviation < 0)
        {
            tunerNeedleFlat.Value = Math.Min(tunerNeedleFlat.Maximum,
Math.Abs(deviation));
        }
        else if (deviation > 0)
        {
            tunerNeedleSharp.Value = Math.Min(tunerNeedleSharp.Maximum, deviation);
        }
    }
}
```

The interface is designed for high visibility. It features a microphone selector, a frequency adjustment input, and a large visual tuning indicator.

- **Top Left:** User input for Concert Pitch (A4 = 440Hz).
- **Top Right:** Dropdown for selecting input devices (e.g., Microphone Array).
- **Center:** Note readout ("-- Cents") and the visual needle (currently centered/inactive in screenshot).
- **Center Icon:** An "X" indicating no note is locked, which switches to a Checkmark when tuned.



4.3.3 Dynamic Configuration (Adjustable A4)

The GUI includes a specific feature to change the concert pitch from (A4=440Hz) to a range between 400-480Hz. The BaseFrequency is then passed into the FFTAnalyzer.

```
if (double.TryParse(input, out double parsedFreq))
 {
     // Clamping logic
     if (parsedFreq < 400) parsedFreq = 400;
     else if (parsedFreq > 480) parsedFreq = 480;

     currentA4Freq = parsedFreq;
     analyzer.BaseFrequency = currentA4Freq;
 }
```

6. Time Management & Planning

The project was executed over a 15-week timeline.

Phase	Duration	Tasks Completed
1. Research & Analysis	Week 1	Investigated SDL2 for audio. Selected Math.NET for FFT. Prototyped audio imput and handling buffer data.
2. Core Logic Implementation	Week 2-4	Implemented FFT algorithm. Encountered accuracy issues when detecting frequency. Split audio project into TunerCore and UI.
3. Signal Refinement	Week 5-6	Researched and added Hann Window, HPS and Quadratic interpolation to improve accuracy.
4. GUI Development	Week 7-8	Migrated code to Windows Forms. Designed the "Needle" UI using dual progress bars. Implemented the Event-Driven timer loop.
5. Refinement & Testing	Week 9-11	Added features for adjustable A4 frequency. Tuned signal processing logic. Polished application.
6. Documentation & Report Writing	Week 12	Create README, presentation and final report. Make GitHub repository public

7. Conclusion

Summary

The project successfully delivered a functional, high-precision chromatic tuner. By leveraging the Harmonic Product Spectrum (HPS) algorithm, the application solves the issue of octave errors, while Quadratic Interpolation allows for tuning precision far exceeding standard FFT bin resolution.

Key Learnings

1. **Signal Processing:** Mathematical theory does not always match real-world acoustics; a pure HPS algorithm was too "strict" for quiet guitar signals, necessitating a hybrid fallback system to improve user experience.
2. **UI Threading:** Moving from a linear Console loop to a GUI event timer required refactoring the resource management to prevent memory leaks and UI freezing.
3. **System Architecture:** Dividing code into UI and backend logic simplified code significantly. Managing and storing data safely was also a concern.

Future Scope

Future iterations of this project could include:

- Some notes (E4) read inaccurately by <5 cents, refinement to signal processing may be required
- GUI is still rudimentary, custom widgets/images would go a long way to improve user experience
- Feature to play notes using a synthesizer to allow tuning by ear
- Visualizer to map out frequency data