2024

Parallelization Project

CAB401 – ASSIGNMENT 2

N11085614 – HARRY JAMES PATRICK WRIGHT

Contents

1 Explanation of original sequential algorithm

1.1 Overview:

The 'Digital Music Analysis' application, designed by students at the Queensland University of Technology, was developed to assist beginner violin players by analysing their performance on provided sheet music. The application processes an audio file (e.g., a recording of the player) and a corresponding XML file containing the sheet music. It compares the actual performance (captured in the audio file) with the intended performance (as indicated in the sheet music). Providing detailed feedback in real-time on pitch accuracy, timing and other musical attributes. See Appendix 2 Figure7 for further context on the UI.

The application is built using C# within the .NET framework, which offers a robust set of libraries and tools specifically optimized for the windows platform. This ecosystem facilitates seamless integration with the operating system but may be further leveraged for its powerful parallel computing capabilities through libraries such as Task Parallel Library (TPL), which are central to the application's planned performance enhancements.

1.2 Software Design/Architecture:

The 'Digital Music Analysis' application is structured around several key classes, utilising the C# language and .NET environment for its robust libraries and seamless object-oriented design. Each class is responsible for different aspects of the processing and analysis pipeline. Available for further context are graphics Figure 5 and 6 from Appendix 1.

1.2.1 MainWindow (UI Management and Control Flow):

The 'MainWindow' class is the central 'hub' of the application, handling the loading of audio (WAV) and XML files through dialogues, enabling users to select and import the files for analysis. Thus, it also comprises event handling, such as button clicks and starting/stopping playback, and dynamically updates the user interface to reflect analysis results, including displaying visual feedback on pitch accuracy and timing. It also initialises the audio processing pipeline by calling relevant classes and methods like onset detection to perform analyses like STFT (Short-Time Fourier Transform) and FFT (Fast Fourier Transform).

1.2.2 wavefile (Audio File Handling):

The 'wavefile' class is responsible for reading and interpreting WAV files, reading headers to gather format information such as sample rate, bit depth and channels, and converting the raw audio bytes into a format suitable for analysis through the 'timefreq' class. Specifically, the music is represented through floating-point, which is easier to manipulate.

1.2.3 timefreq (Time-Frequency Analysis):

The 'timefreq' class performs STFT on blocks of audio data, converting time-domain signals into frequency-domain representations, which is crucial for identifying the frequencies present in the audio over time. The Fast Fourier Transform (FFT), an efficient algorithm for computing the discrete Fourier transform (Heckbert, 1995), is used within the STFT process to optimize these calculations. The resulting STFT output is structured for easy manipulation by other components of the application.

1.2.4 musicNote (Pitch Analysis and Representation):

The ‘musicNote’ class represents individual music notes and conducts pitch analysis. It calculates the pitch of detected notes based on their frequency and determines how closely these pitches match the expected pitches according to the sheet music and assists the application UI by translating pitch information into a graphical format understandable by users. Including calculating pitch deviation / errors between performed and written notes.

1.2.5 noteGraph (Visualization and Histogram Generation):

The ‘noteGraph’ class is responsible for generating the visual elements of the UI such as histograms that represent audio analysis results. Visiualising aspects such as frequency distribution or volume over time. It transforms the analysis data into these visualised formats, which are integrated into the UI.

1.3 Application Sequence:

The architecture of the 'Digital Music Analysis' application enforces a clear separation of data processing, audio handling, and user interface management, characteristic of object-oriented design in C#. The 'MainWindow' class oversees the process, delegating audio parsing to the 'wavefile' class and sheet music parsing to the 'musicNote' class, ensuring both inputs are properly prepared for analysis. The 'timefreq' class handles computationally intensive tasks, performing a Short-Time Fourier Transform (STFT) to generate a time-frequency representation of the audio. The 'musicNote' class then analyzes this data, comparing it to the sheet music to calculate pitch errors and assess performance accuracy. The 'noteGraph' class provides visual feedback in the form of histograms and graphical elements, synchronized with audio playback.

2 Analysis of Potential Parallelism

The initial analysis began with profiling application’s CPU usage, specifically, within Visual Studio 2022. Figure 1 displays results after allowing program to perform all functionality (including UI) for the runtime of the provided song and sheetmusic .xml representation, reaching a peak of ~7% CPU usage.

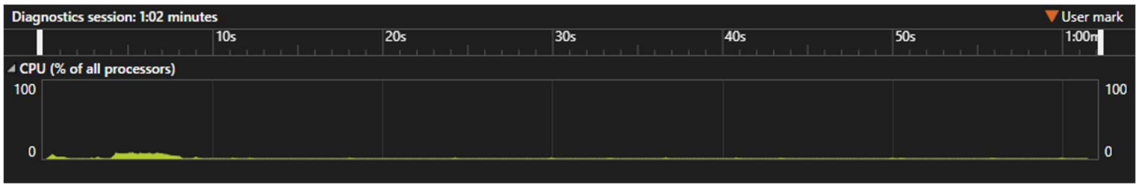

Figure 1: ‘Digital Music Analysis’ profiling runtime and CPU usage. Peaks indicate when files were loaded and when calculations were being performed.

Figure 2 is a summary of function CPU usage where which CPU usage peaked during calculations in order to make data-driven decisions about where best parallelizing may be targeted.

Function Name	Total CPU [unit, %]	Self CPU [unit, %]	Module
DigitalMusicAnalysis (PID: 9812)	3970 (100.00%)	369 (9.29%)	DigitalMusicAnaly...
[External Call] System.Windows.Application.Run()	3592 (90.48%)	157 (3.95%)	presentationfram...
DigitalMusicAnalysis.App.Main()	3592 (90.48%)	0 (0.00%)	digitalmusicanalysis
DigitalMusicAnalysis.MainWindow.ctor()	3415 (86.02%)	7 (0.18%)	digitalmusicanalysis
DigitalMusicAnalysis.MainWindow.freqDomain()	1756 (44.23%)	28 (0.71%)	digitalmusicanalysis
DigitalMusicAnalysis.timefreq.ctor(float32[], int)	1728 (43.53%)	38 (0.96%)	digitalmusicanalysis
DigitalMusicAnalysis.timefreq.stft(System.Numeri...	1689 (42.54%)	89 (2.24%)	digitalmusicanalysis
DigitalMusicAnalysis.timefreq.fft(System.Numeric...	1588 (40.00%)	1419 (35.74%)	digitalmusicanalysis
DigitalMusicAnalysis.MainWindow.onsetDetectio...	1546 (38.94%)	210 (5.29%)	digitalmusicanalysis
DigitalMusicAnalysis.MainWindow.fft(System.Nu...	1127 (28.39%)	994 (25.04%)	digitalmusicanalysis
[External Call] System.Numerics.Complex.op_Addi...	162 (4.08%)	162 (4.08%)	system.runtime.nu...
[External Call] System.Numerics.Complex.op_Mult...	128 (3.22%)	128 (3.22%)	system.runtime.nu...
[External Call] System.Numerics.Complex.Pow(Sys...	95 (2.39%)	95 (2.39%)	system.runtime.nu...
[External Call] System.Numerics.Complex.Exp(Syst...	50 (1.26%)	50 (1.26%)	system.runtime.nu...
DigitalMusicAnalysis.MainWindow.playBack()	47 (1.18%)	1 (0.03%)	digitalmusicanalysis

Figure 2: Main function CPU usage after profiling in Visual Studio 2022.

2.1 Profiling Results

Seen in Figure 2, profiling results indicate several functions within the 'Digital Music Analysis' app where significant CPU resources are being consumed. The majority of the computational load occurs during initialisation and audio processing. The MainWindow.ctor() method - responsible for setting up the application- accounts for 86.02% of total usage, indicating incredible resource intensity and time-consumption during initialisation.

Following initialisation, time-frequency calculations (FFT/STFT) within the 'freqDomain()' method consume ~40% of total CPU time each, making it the next largest contributor to overall resource usage. The recursive 'fft()' method alone uses 40% of CPU time, and a significant 35.74% of self-CPU usage. Similarly, onset detection in the 'onsetDetection()' method is another area of concern, accounting for 38.94% of CPU usage, as it scans and processes audio data to detect musical note onsets.

UI updates (such as 'updateSlider()' and 'loadHistogram()') and playback represent an insignificant fraction of CPU usage, accounting for less than 2% total CPU usage when combined.

2.1.1 Granularity Considerations

The potential for coarse-grained parallelism is higher within this analysis in computationally heavy operations such as STFT, FFT and onset detection. These operations involve significant independent computation in loops, and the overhead of parallel thread management is justified by the computational load. The main focus of this analysis is to focus on high-impact components of the 'Digital Music App', even if the operations are relatively fine-grained. In other words, this "parallelise everything" approach seeks to maximise speedup wherever possible.

2.2 Potential Parallelism Opportunities

Results in 2.1 indicate clear areas where parallelism can be applied to reduce computational overhead and improve execution time. In line with the .NET framework's strengths, the focus on leveraging its native parallelisation libraries, such as TPL, is a logical step for achieving this.

Given the available hardware (my computer) and operating system, Windows-optimised libraries will ensure any parallelisation strategies are effective and maintainable. Careful attention must be paid to data dependencies, especially within time-frequency calculations and onset detection processes, which exhibit potential for parallelism identified in recent profiling.

2.2.1 Time-Frequency Calculations

The STFT and FFT processes within the timefreq class represent some of the most computationally expensive parts of the application. Therein represents opportunities for parallelism, but also contain certain data dependencies to be considered.

2.2.1.1 STFT (Short-Time Fourier Transform)

The 'stft' method processes each window of audio data and applies FFT to it. There are multiple parallelisable loops, which can be viewed in Appendix 3, Figure 8 and 9 and in both cases, it is possible to parallelise the outermost ii loop to process windows independently through TPL. In the case of the former, there is concern of a flow dependence, as 'fftMax' is written and read by subsequent iterations. Additionally, the loop writes to 'fftMax' multiple times (output dependence), which potentially introduces a possible race condition which must be addressed.

2.2.1.2 FFT (Fast Fourier Transform)

The 'fft' function is recursive, and processes even and odd-indexed elements separately, as seen in Appendix 3, Figure 10. It is tempting to parallelise this code, given its heavy resource usage, but the potential for race conditions is incredibly high. The fft algorithm relies on subproblems (dividing the input into even and odd parts), which is dangerous to parallelise because they invoke dependencies between recursive calls, and as such are best avoided.

2.2.2 Onset Detection

The onsetDetection() function includes several computational steps in the form of loops which occur in stages. These processes could benefit from parallelism, but also introduce unique data dependencies that must be addressed.

2.2.2.1 HFC (Harmonic Frequency Content)

The first loop calculates the HFC values for each window of the audio data, visible under Figure 11 of Appendix 3, and is potentially parallelisable. However, since each iteration of the loop depends on the prior value of 'HFC[jj]' to accumulate results, there is a flow dependency, as the loop reads from HFC[jj] and writes to the same index.

2.2.2.2 HFC Normalisation

The HFC[] array is normalised after its calculation by dividing each element by the maximum value, as seen in Figure 12 Appendix 3. This loop could be executed in parallel through TPL, noting that it cannot occur until HFC.Max() is known.

2.2.2.3 Pitch Detection

Seen in Figure 13 Appendix 3 is multiple loops within an outermost loop, responsible for pitch detection. Through TPL, the outermost loop can be run in parallel for each note detected, and the calculation of compX, twiddles and magnitude can all be executed in parallel. However, the

calculation of maxInd cannot, as it has an output dependency which updates maxInd iteratively.

2.2.3 UI Updates

Given UI updates occur on the main thread, direct parallelism is not feasible. As indicated in 2.1, UI and playback components amount to less than 2% of total CPU contribution across the application. In order to streamline the execution of primary resource intensive tasks, and better monitor responsiveness, these components will be removed, and parallelism will focus on the computationally expensive calculations.

2.2.4 Initialisation

Due to its inherently sequential nature, the MainWindow.ctor() method presents fewer opportunities for parallelism. Instead, it can be modified to remove UI calls, import data directly and time the runtime of individual components.

3 Parallel Implementation

3.1 Hardware

Profiling results, as shown in Figure 1, were obtained using a prebuilt computer model _____, which serves as the primary hardware available for evaluating the performance of the 'Digital Music Analysis' application before and after parallelism is exploited. In its unmodified state, this system is equipped with 8 cores and approximately 5200 threads, supported by 16 GB of RAM and 8 GB of dedicated GPU memory. This configuration provides substantial computational resources, well-suited to the .NET library envisioned for our parallelisation strategies, TPL. Our parallelisation methods are unlikely to utilise the full extent of the available hardware but should ensure robust handling of the application's computational demands.

3.2 Implementation

Parallelism was introduced to identified opportunities primarily through TPL, a built-in parallelisation library for the .NET framework. The core philosophy behind the implementation was to break down tasks into independently executable components where possible, minimising inter-thread dependencies and avoiding race conditions. This section outlines the approach used for each identified area of parallelism, including the handling of dependencies.

3.2.1 STFT

The STFT function contained two key loops that were identified as candidates for parallelisation. They iterate over independent segments of data, which presents the opportunity to process them simultaneously.

The first ii loop (Figure 8 of Appendix 3) iterates over segments of audio data, transforming each using FFT. As outlined previously, the fftMax variable in this loop introduced concerns regarding both flow and output dependencies. Specifically, each thread writes to and reads from fftMax, potentially creating race conditions if updated simultaneously. To resolve this, a thread-local variable was introduced, allowing each each thread to calculate its local maximum. The global fftMax variable is then updated atomically using interlocked.CompareExchange, ensuring that

the highest value is retained without risking race conditions. This allows for safe parallel execution of the loop, as shown in Appendix 4, Figure 14.

The second ii loop is responsible for normalising the FFT results. With no inter-iteration data dependencies, the outermost loop could be parallelised without concern. This loop was parallelised with TPL's Parallel.For() as well, as indicated in appendix 4, Figure 15.

3.2.2 FFT

The FFT algorithm posed significant challenges in terms of parallel implementation. As it relies heavily on recursive operations, it is inherently sequential. Attempts were made to parallelise it however, initial attempts sought to divide the even and odd calculations of the input for separate threads, though this led to synchronization issues and race conditions. This is because threads tried to update shared resources concurrently. Thread-local variables and atomic operations were implemented to solve this, but deadlocking issues still pursued. As such, parallelising FFT remains unsolved due to its inherent data dependencies.

3.2.3 HFC

The HFC calculation loop is central to the onset detection processes and exhibited a flow dependency on the HFC array. Because each iteration of the loop modified parts of the array sequentially, it made it unsuitable for direct parallelisation. A reduction strategy was employed to address this: each thread now works on independent segments of the HFC array. When processing concludes, the results are summed in a final pass to produce an overall HFC value. This eliminates the flow dependency by ensuring no thread writes to the same location simultaneously, resulting in correct behaviour. The result is visible in Appendix 4, Figure 16.

3.2.4 HFC Normalisation

With HFC normalisation, the flow dependency identified for the HFC.Max() calculation proved inconsequential. The maximum value of the HFC array can be calculated in a sequential step prior to parallelisation as it does not change during the normalisation process. The loop itself was parallelised without issue, in a similar fashion to Appendix 4, Figure 15. The parallelised code is visible in Appendix 4, Figure 17.

3.2.5 Pitch Detection

Pitch detection involved several loops suitable for parallelisation. Going in order, the outermost loop, responsible for iterating through note starts and stops, was parallelised directly with TPL's Parallel.For, as there were no inter-iteration dependencies to be concerned of. Allowing each iteration to safely calculate the pitch for an individual note without interfering with others.

Similarly, the loop that generates twiddle factors for the FFT calculation and the magnitude calculation loop, which converts complex FFT results to magnitudes, were also directly parallelised as they exhibited no notable dependencies. Allowing for threads to calculate independently in parallel.

The loop which populates the compX array (which holds the input for the FFT) was parallelised, although being careful to avoid the flow dependencies on the noteStarts and lengths variables. In this regard, these variables are initialised sequentially before the loop, which allows the loop to be parallelised safely. Parallelised code for pitch detection is found in Appendix 4 Figure 18.

4 Results

4.1 Timing

The inbuilt Stopwatch() object was used to time key processes within the application. This timing was carried out on a modified version of the application, before and after parallelism, which stripped all nonessential UI items, including the importation of files. All tests were carried out on the same music file and music sheet .xml file and run fifty times to obtain the best result in milliseconds. The best sequential runtime in milliseconds is displayed below:

Best Sequential	
Task	Time(ms)
Overall	1720
STFT	1294
OnsetDetection	1462

For the parallel tests, TPL's inbuilt control over the number of processors to tasks was exploited. The following table shows the best results from the parallel version of the program at different numbers of processors, from 1 to the maximum available on the hardware, 8.

Parallel Overall Runtime	
Num Processors	Execution time (ms)
1	1008
2	729
3	675
4	594
5	541
6	630
7	586
8	603

Parallel STFT Runtime	
Num Processors	Execution time (ms)
1	775
2	550
3	472
4	439
5	397
6	461
7	445
8	447

Parallel OnsetDetection Runtime	
Num Processors	Execution time (ms)
1	883
2	604
3	508

4	467
5	421
6	483
7	465
8	467

Overall parallel speedup was then graphed in Figure 3, which indicates a runtime peak at 5 processors, thereafter decreasing in terms of speedup at and beyond 6 cores, with fluctuation.

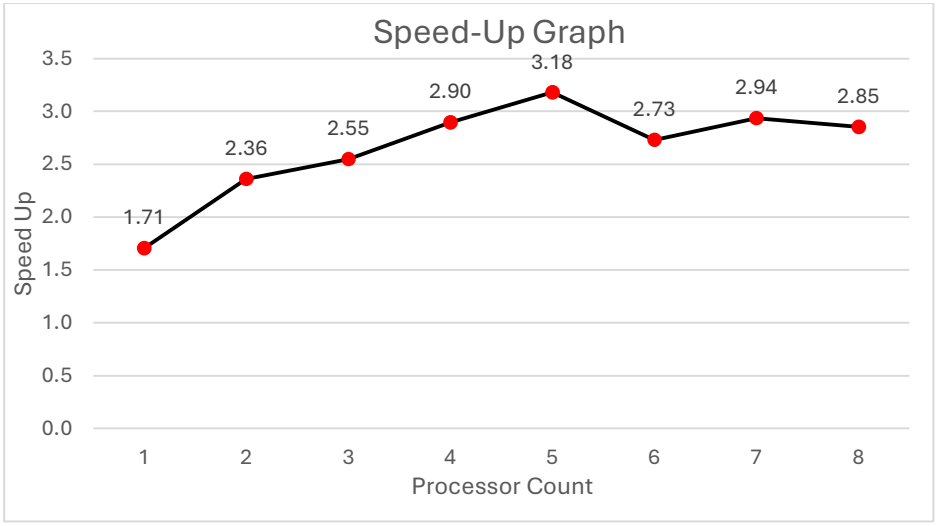

Figure 3: Speed-up graph showing sequential runtime as a fraction of parallel overall runtime against the number of processors used.

4.2 Profiling

CPU Profiling results in Figure 4 reveal the primary expenditure is situated on the stft and fft calculations, which originate from onset detection and within stft itself. It indicates how impactful fft is to runtime, and how unfortunate it is that parallelising it is so difficult. It aligns with the parallel STFT and OnsetDetection runtimes being so similar, indicating that parallelisation has drastically mitigated their effect on performance.

Current View: Functions

Function Name	Total CPU [unit, %]	Self CPU [unit, %]	Module
DigitalMusicAnalysis (PID: 39824)	2939 (100.00%)	0 (0.00%)	DigitalMusicAnalysis
[External Call] RtlUserThreadStart	2928 (99.63%)	392 (13.34%)	ntdll
DigitalMusicAnalysis.timefreq.stft.AnonymousMethod__0(int)	1892 (64.38%)	122 (4.15%)	digitalmusicanalysis
DigitalMusicAnalysis.timefreq.fft(System.Numerics.Complex[])	1712 (58.25%)	707 (24.06%)	digitalmusicanalysis
[External Call] coreclr.dll!0x00007ffd7eb9d3bb	917 (31.20%)	917 (31.20%)	coreclr
DigitalMusicAnalysis.App.Main()	770 (26.20%)	0 (0.00%)	digitalmusicanalysis
[External Call] System.Windows.Application.Run()	694 (23.61%)	327 (11.13%)	presentationframework
DigitalMusicAnalysis.MainWindow.ctor()	366 (12.45%)	0 (0.00%)	digitalmusicanalysis
DigitalMusicAnalysis.MainWindow.freqDomain()	295 (10.04%)	6 (0.20%)	digitalmusicanalysis
DigitalMusicAnalysis.timefreq.ctor(float32[], int)	286 (9.73%)	6 (0.20%)	digitalmusicanalysis
[External Call] system.threading.tasks.parallel.dll!0x00007ffd9bfec857	276 (9.39%)	7 (0.24%)	system.threading.tasks.parallel
DigitalMusicAnalysis.timefreq.stft(System.Numerics.Complex[], int)	265 (9.02%)	0 (0.00%)	digitalmusicanalysis
DigitalMusicAnalysis.MainWindow.onsetDetection.AnonymousMethod__0(int)	123 (4.19%)	39 (1.33%)	digitalmusicanalysis
[External Call] coreclr.dll!0x00007ffd7eed8fc9	84 (2.86%)	84 (2.86%)	coreclr
[External Call] System.Windows.Application..ctor()	76 (2.59%)	76 (2.59%)	presentationframework
DigitalMusicAnalysis.App.ctor()	76 (2.59%)	0 (0.00%)	digitalmusicanalysis
[External Call] System.Windows.Window..ctor()	31 (1.05%)	31 (1.05%)	presentationframework

Figure 4: CPU ‘Function’ view of profiling results obtained from Visual Studio 2022.

5 Conclusion & Reflection

The ‘Digital Music App’ was never expected to achieve linear or near-linear speedup, and a best speed-up of ~3.2x over the ‘original’ runtime is worth celebrating. It marks my first utilisation of parallel techniques and overcoming data dependencies in order to squeeze improvements out of every available -worthwhile- loop and calculation.

However, timing and profiling results indicate a scalability limitation potentially related to synchronisation, thread management and/or parallelism implementation. In particular, my philosophy of dividing operations as much as possible was exploited across the current implementation. However, it may benefit from future investigation could be switched from fine-grained parallelism to coarse-grained parallelism for greater control.

Further, successfully parallelising FFT would open up an avenue for further improvement. The implementation of a parallel or more efficient algorithm would certainly be worth investigating.

A link to a GitHub repository is given in Appendix 5.

Appendix 1 – Graphs

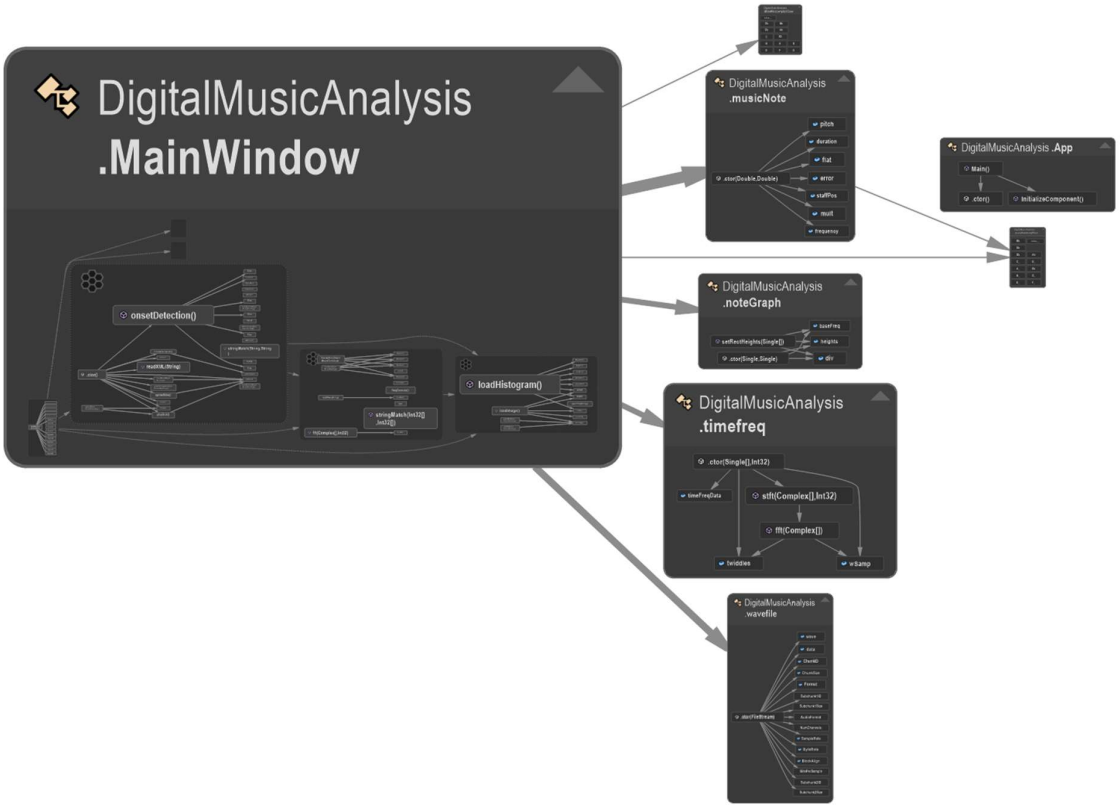

Figure 5: Dependency Graph of 'DigitalMusicApp' generated by NDepend extension within Visual Studio 2022. It showcases the calling relationship/s between the 'MainWindow' class and all other classes within the application, their subroutines and relationships.

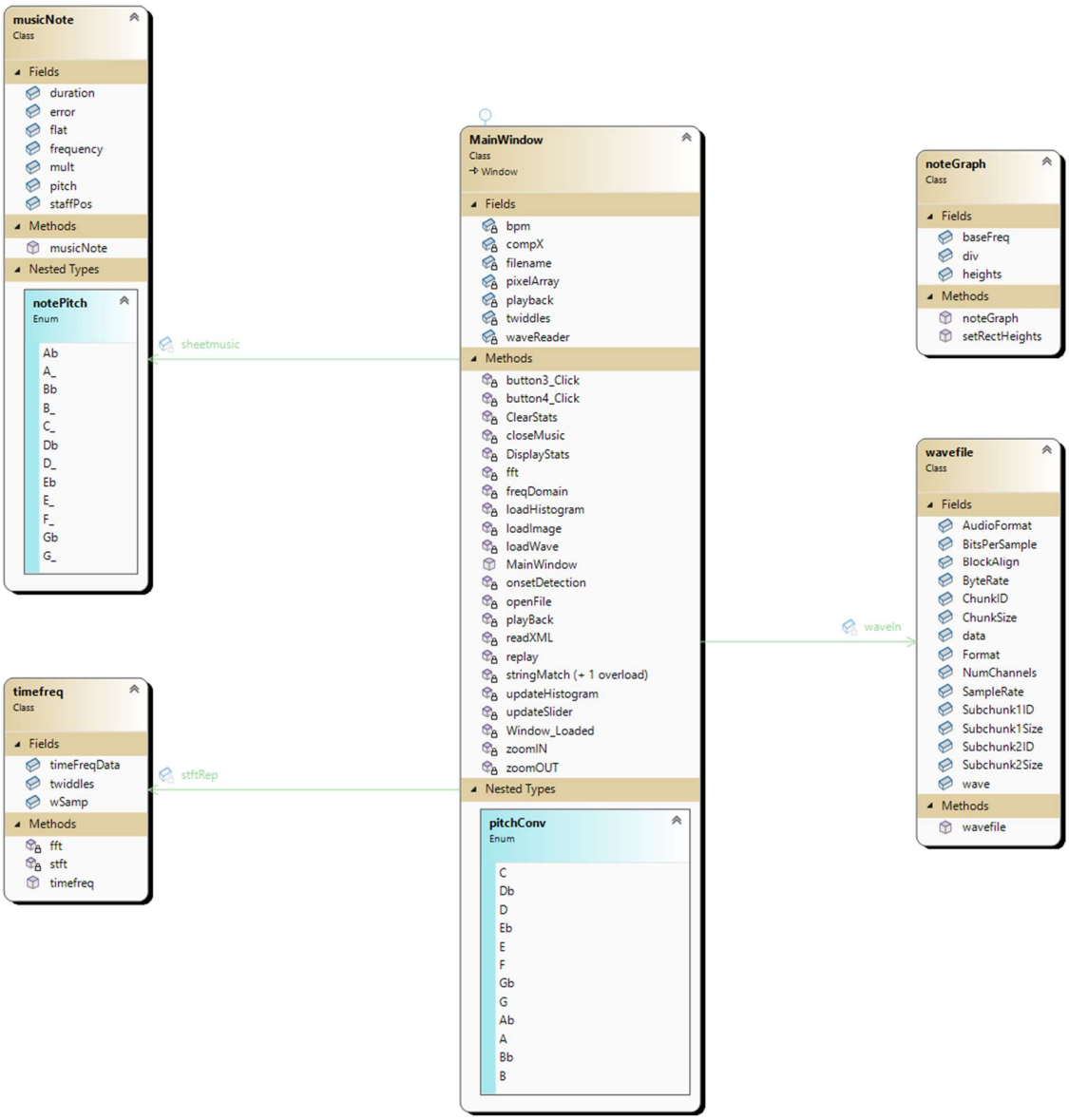

Figure 6: Class diagram of 'Digital Music App' generated within Microsoft Visual Studio 2022. Shows the Fields, Methods and Nested Types of each class within the application and their direct connections with one another. noteGraph is called within a function known as 'loadHistogram', not a field. Thus, Visual Studio is incapable of creating the annotation given the 'indirect' connection, even though it should be present.

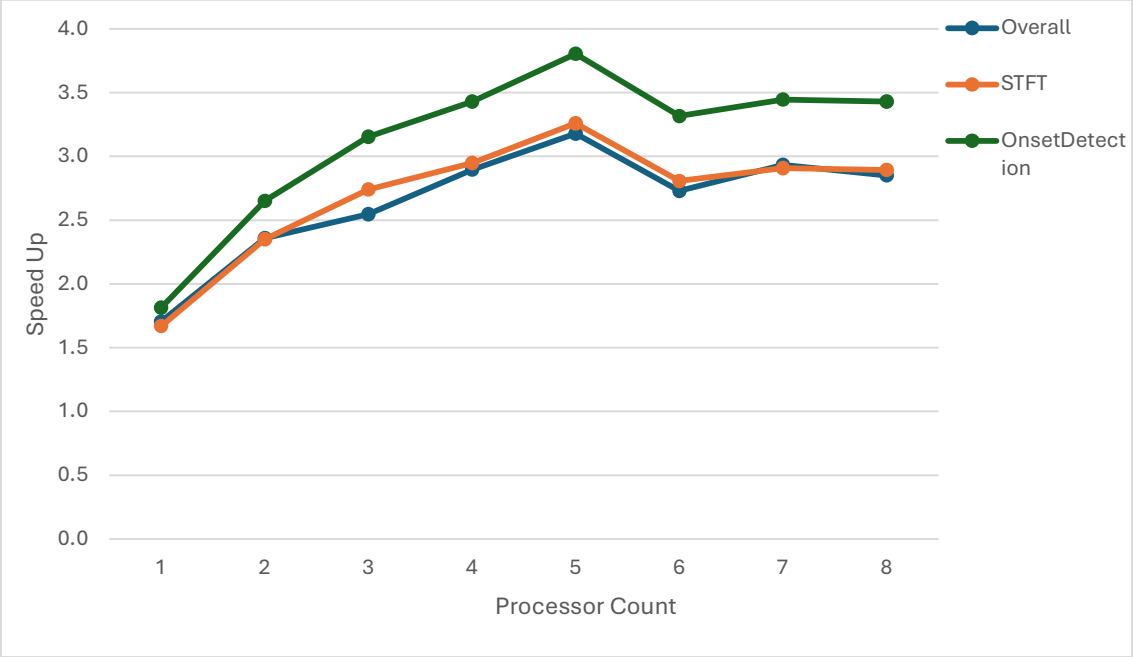

Figure 19: Unused Speedup graph comparing every class speedup across provided processor count.

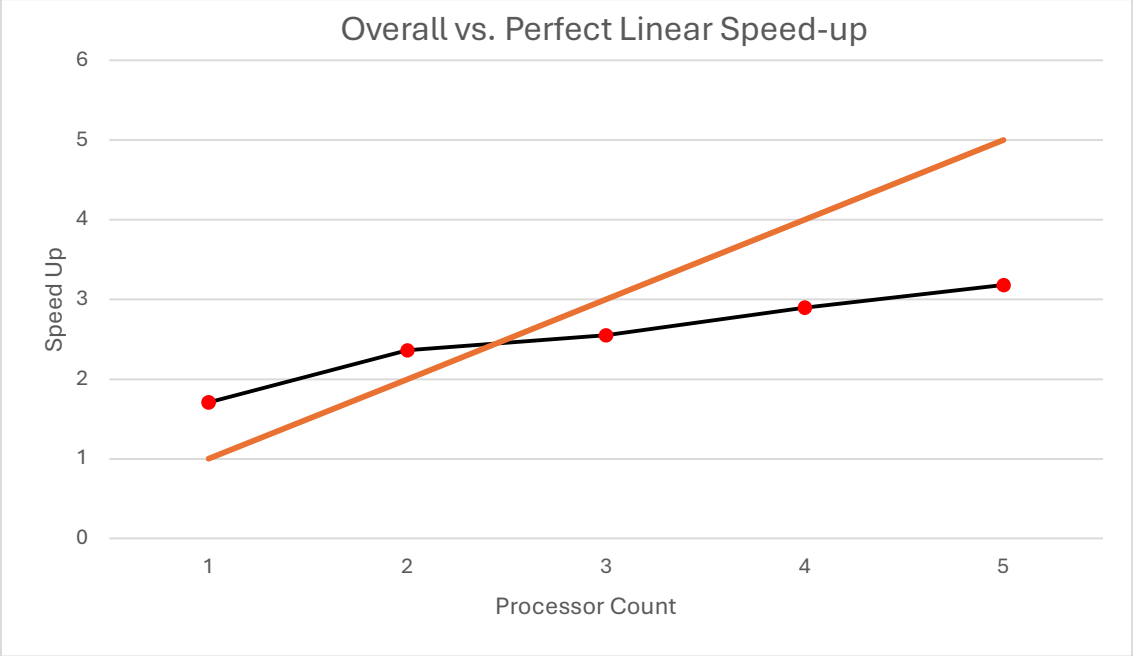

Figure 20: Unused Speedup graph comparing overall runtime against perfect linear speedup.

Appendix 2 – Screen Snippets

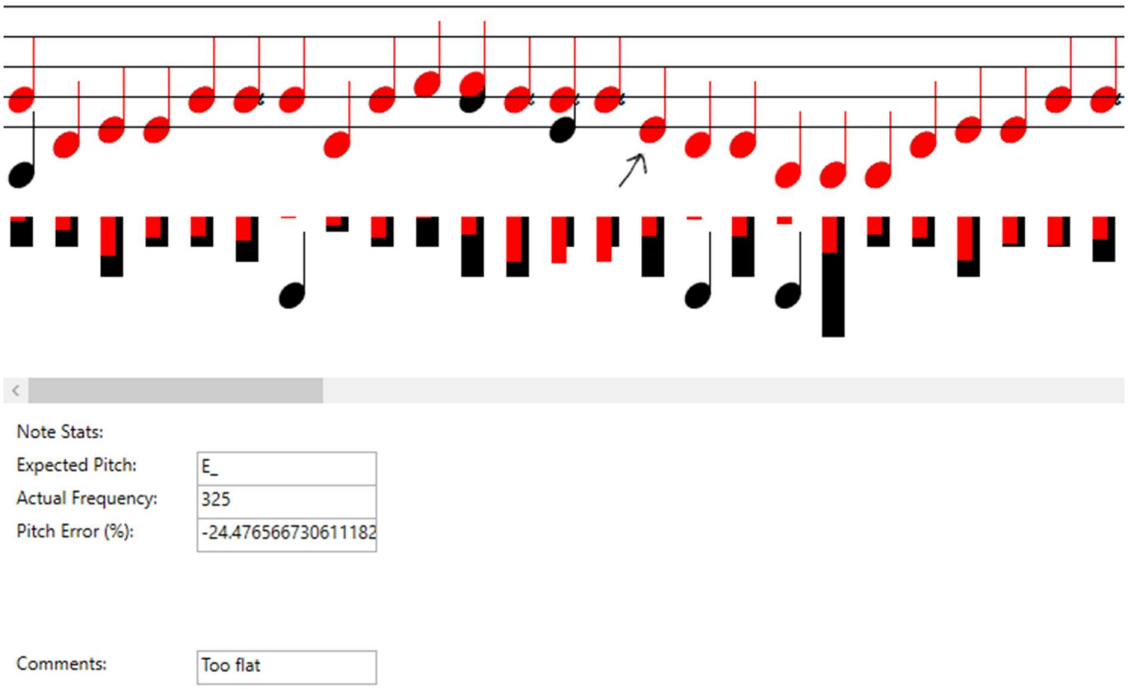

Figure 7: Screenshot of 'Staff' screen of the 'Digital Music Analysis' app on the music composition 'Jupiter'. The arrow indicates which note is receiving feedback.

Appendix 3 – Original Code

```
for (ii = 0; ii < 2 * Math.Floor((double)N / (double)wSamp) - 1; ii++)
{

    for (jj = 0; jj < wSamp; jj++)
    {
        temp[jj] = x[ii * (wSamp / 2) + jj];
    }

    tempFFT = fft(temp);

    for (kk = 0; kk < wSamp / 2; kk++)
    {
        Y[kk][ii] = (float)Complex.Abs(tempFFT[kk]);

        if (Y[kk][ii] > fftMax)
        {
            fftMax = Y[kk][ii];
        }
    }
}
```

Figure 8: Parallelisable loop identified within the timefreq stft function. Lines 74-93.

```
for (ii = 0; ii < 2 * Math.Floor((double)N / (double)wSamp) - 1; ii++)
{
    for (kk = 0; kk < wSamp / 2; kk++)
    {
        Y[kk][ii] /= fftMax;
    }
}
```

Figure 9: Second parallelisable loop identified within the timefreq stft function. Lines 97-103.

```
Complex[] fft(Complex[] x)
{
    int ii = 0;
    int kk = 0;
    int N = x.Length;

    Complex[] Y = new Complex[N];


    if (N == 1)
    {
        Y[0] = x[0];
    }
    else{

        Complex[] E = new Complex[N/2];
        Complex[] O = new Complex[N/2];
        Complex[] even = new Complex[N/2];
        Complex[] odd = new Complex[N/2];

        for (ii = 0; ii < N; ii++)
        {

            if (ii % 2 == 0)
            {
                even[ii / 2] = x[ii];
            }
```

```
            if (ii % 2 == 1)
            {
                odd[(ii - 1) / 2] = x[ii];
            }
        }

        E = fft(even);
        O = fft(odd);

        for (kk = 0; kk < N; kk++)
        {
            Y[kk] = E[(kk % (N / 2))] + O[(kk % (N / 2))] * twiddles[kk * wSamp /
N];
        }
    }

   return Y;
}
```

Figure 10: whole fft code from timefreq.cs. Lines 118-163.

```
for (int jj = 0; jj < stftRep.timeFreqData[0].Length; jj++)
{
    for (int ii = 0; ii < stftRep.wSamp / 2; ii++)
    {
        HFC[jj] = HFC[jj] + (float)Math.Pow((double)stftRep.timeFreqData[ii][jj]
* ii, 2);
    }
}
```

Figure 11: Parallelisable loop identified within the onsetDetection function within MainWindow.xaml.cs. Lines 318-324.

```
float maxi = HFC.Max();

for (int jj = 0; jj < stftRep.timeFreqData[0].Length; jj++)
{
    HFC[jj] = (float)Math.Pow((HFC[jj] / maxi), 2);
}
```

Figure 12: Parallelisable loop identified within the onsetDetection function within MainWindow.xaml.cs. Lines 326-331.

```
for (int mm = 0; mm < lengths.Count; mm++)
{
    int nearest = (int)Math.Pow(2, Math.Ceiling(Math.Log(lengths[mm], 2)));
    twiddles = new Complex[nearest];
    for (ll = 0; ll < nearest; ll++)
    {
        double a = 2 * pi * ll / (double)nearest;
        twiddles[ll] = Complex.Pow(Complex.Exp(-i), (float)a);
    }

    compX = new Complex[nearest];
    for (int kk = 0; kk < nearest; kk++)
    {
        if (kk < lengths[mm] && (noteStarts[mm] + kk) < waveIn.wave.Length)
        {
            compX[kk] = waveIn.wave[noteStarts[mm] + kk];
        }
        else
```

```
        {
            compX[kk] = Complex.Zero;
        }
    }

    Y = fft(compX, nearest);

    absY = new double[nearest];

    double maximum = 0;
    int maxInd = 0;

    for (int jj = 0; jj < Y.Length; jj++)
    {
        absY[jj] = Y[jj].Magnitude;
        if (absY[jj] > maximum)
        {
            maximum = absY[jj];
            maxInd = jj;
        }
    }

    for (int div = 6; div > 1; div--)
    {
        if (maxInd > nearest / 2)
        {
            if (absY[(int)Math.Floor((double)(nearest - maxInd) / div)] /
absY[(maxInd)] > 0.10)
            {
                maxInd = (nearest - maxInd) / div;
            }
        }
        else
        {
            if (absY[(int)Math.Floor((double)maxInd / div)] / absY[(maxInd)] >
0.10)
            {
                maxInd = maxInd / div;
            }
        }
    }

    if (maxInd > nearest / 2)
    {
        pitches.Add((nearest - maxInd) * waveIn.SampleRate / nearest);
    }
    else
    {
        pitches.Add(maxInd * waveIn.SampleRate / nearest);
    }
}
```

Figure 13: Parallelisable loop identified within the onsetDetection function within MainWindow.xaml.cs. Lines 364-430.

Appendix 4 – Parallelised Code

```
Parallel.For(0, (int)(2 * Math.Floor((double)N / (double)wSamp) - 1), new
ParallelOptions { MaxDegreeOfParallelism = MainWindow.numThreads }, ii =>
{
    Complex[] localTemp = new Complex[wSamp]; // Thread-local temp array

    // Populate temp in parallel
    for (int jj = 0; jj < wSamp; jj++)
    {
        localTemp[jj] = x[ii * (wSamp / 2) + jj];
    }

    // Perform FFT sequentially (due to FFT recursion)
    Complex[] localTempFFT = fft(localTemp);

    float localFftMax = 0; // Thread-local fftMax to avoid race conditions

    for (int kk = 0; kk < wSamp / 2; kk++)
    {
        Y[kk][ii] = (float)Complex.Abs(localTempFFT[kk]);
        if (Y[kk][ii] > localFftMax)
        {
            localFftMax = Y[kk][ii];
        }
    }

    // Atomic max update using Interlocked.CompareExchange
    float initialMax, computedMax;
    do
    {
        initialMax = fftMax;
        computedMax = Math.Max(initialMax, localFftMax);
    } while (initialMax != Interlocked.CompareExchange(ref fftMax, computedMax,
initialMax));
});
```

Figure 14: First piece of parallelised stft code present in the timefreq class. Lines 72-103.

```
Parallel.For(0, 2 * (int)Math.Floor((double)N / (double)wSamp) - 1, new
ParallelOptions { MaxDegreeOfParallelism = MainWindow.numThreads }ii =>
{
    for (int kk = 0; kk < wSamp / 2; kk++)
    {
        Y[kk][ii] /= fftMax;
    }
});
```

Figure 15: Second piece of parallelised stft code present in the timefreq class. Lines 106-112.

```
Parallel.For(0, stftRep.timeFreqData[0].Length, new ParallelOptions
{ MaxDegreeOfParallelism = numThreads }, jj =>
{
    for (int ii = 0; ii < stftRep.wSamp / 2; ii++)
    {
        HFC[jj] += (float)Math.Pow((double)stftRep.timeFreqData[ii][jj] * ii, 2);
    }
});
```

Figure 16: First piece of parallelised onset detection code. Lines 317-323.

```
float maxi = HFC.Max();
Parallel.For(0, stftRep.timeFreqData[0].Length, jj =>
{
    HFC[jj] = (float)Math.Pow((HFC[jj] / maxi), 2);
});
```

Figure 17: Second piece of parallelised onset detection code. Lines 326-329.

```
Parallel.For(0, lengths.Count, new ParallelOptions { MaxDegreeOfParallelism =
numThreads }, mm =>
{
    int nearest = (int)Math.Pow(2, Math.Ceiling(Math.Log(lengths[mm], 2)));
    Complex[] twiddles = new Complex[nearest];

    // Calculate twiddles in parallel
    Parallel.For(0, nearest, ll =>
    {
        double a = 2 * pi * ll / (double)nearest;
        twiddles[ll] = Complex.Pow(Complex.Exp(-i), (float)a);
    });

    Complex[] compX = new Complex[nearest];

    // Populate compX array in parallel
    Parallel.For(0, nearest, kk =>
    {
        if (kk < lengths[mm] && (noteStarts[mm] + kk) < waveIn.wave.Length)
        {
            compX[kk] = waveIn.wave[noteStarts[mm] + kk];
        }
        else
        {
            compX[kk] = Complex.Zero;
        }
    });

    // Perform FFT sequentially (due to recursive nature)
    Complex[] Y = fft(compX, nearest);

    double[] absY = new double[nearest];
    double maximum = 0;
    int maxInd = 0;

    // Compute magnitudes and find max index
    Parallel.For(0, Y.Length, jj =>
    {
        absY[jj] = Y[jj].Magnitude;
        if (absY[jj] > maximum)
        {
            maximum = absY[jj];
            maxInd = jj;
        }
    });

    // Process maxInd (Sequential due to control dependencies)
    for (int div = 6; div > 1; div--)
    {
        if (maxInd > nearest / 2)
        {
            if (absY[(int)Math.Floor((double)(nearest - maxInd) / div)] /
absY[maxInd] > 0.10)
            {
```

```
                maxInd = (nearest - maxInd) / div;
            }
        }
        else
        {
            if (absY[(int)Math.Floor((double)maxInd / div)] / absY[maxInd] >
0.10)
            {
                maxInd = maxInd / div;
            }
        }
    }

    // Add pitch to the thread-safe collection
    if (maxInd > nearest / 2)
    {
        pitches.Add((nearest - maxInd) * waveIn.SampleRate / nearest);
    }
    else
    {
        pitches.Add(maxInd * waveIn.SampleRate / nearest);
    }
});
```

Figure 18: Third piece of parallelised onset detection code. Lines 358-431.

Appendix 5 – GitHub Link

Standard Link: https://github.com/HunterRixon/CAB401-Assignment-2

Clone: https://github.com/HunterRixon/CAB401-Assignment-2.git

References

Heckbert, P. (1995, Feburary). Fourier Transforms and the Fast Fourier Transform (FFT) Algorithm. *Computer Graphics 2*, 15-463.