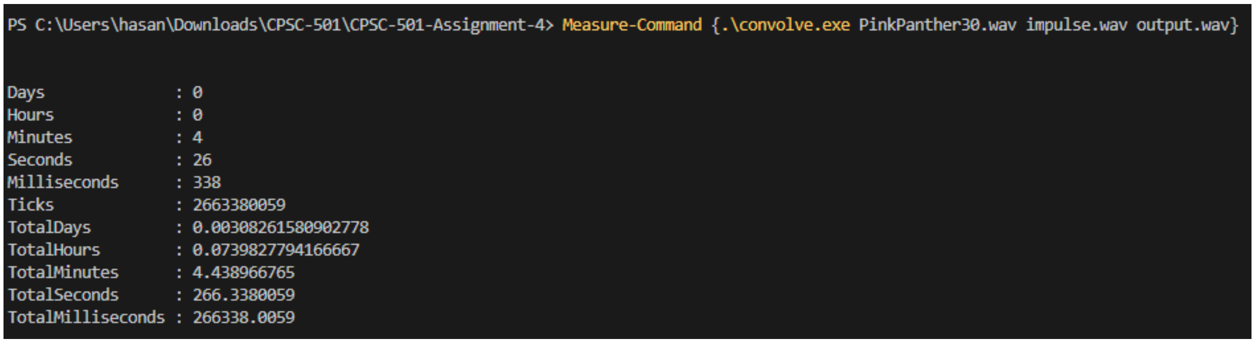
CPSC 501 Assignment 4 Report

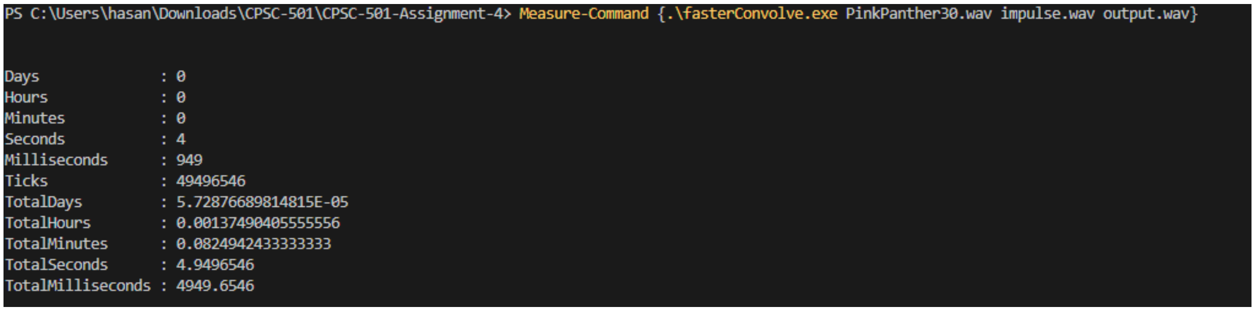
1) With the baseline program it took my program 4 minutes and 41 seconds to run a 30 second input wav file with a 2 second impulse file.



```
PS C:\Users\hasan\Downloads\CPSC-501\CPSC-501-Assignment-4> Measure-Command {.\convolve.exe PinkPanther30.wav impulse.wav output.wav}


Days              : 0
Hours             : 0
Minutes           : 4
Seconds           : 26
Milliseconds      : 338
Ticks             : 2663380059
TotalDays         : 0.00308261580902778
TotalHours        : 0.0739827794166667
TotalMinutes      : 4.438966765
TotalSeconds      : 266.3380059
TotalMilliseconds : 266338.0059
```

With the Algorithm Optimization it took my program 4.95 seconds to run the same inputs as the other program



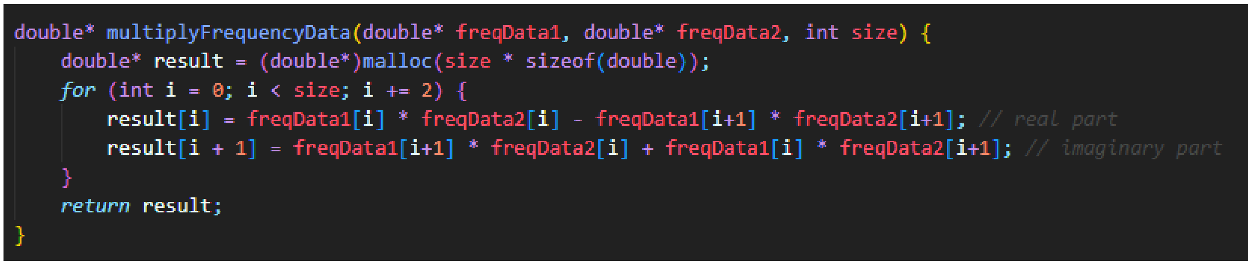
```
PS C:\Users\hasan\Downloads\CPSC-501\CPSC-501-Assignment-4> Measure-Command {.\fasterConvolve.exe PinkPanther30.wav impulse.wav output.wav}


Days              : 0
Hours             : 0
Minutes           : 0
Seconds           : 4
Milliseconds      : 949
Ticks             : 49496546
TotalDays         : 5.72876689814815E-05
TotalHours        : 0.00137490405555556
TotalMinutes      : 0.0824942433333333
TotalSeconds      : 4.9496546
TotalMilliseconds : 4949.6546
```

Code Tuning Optimizations

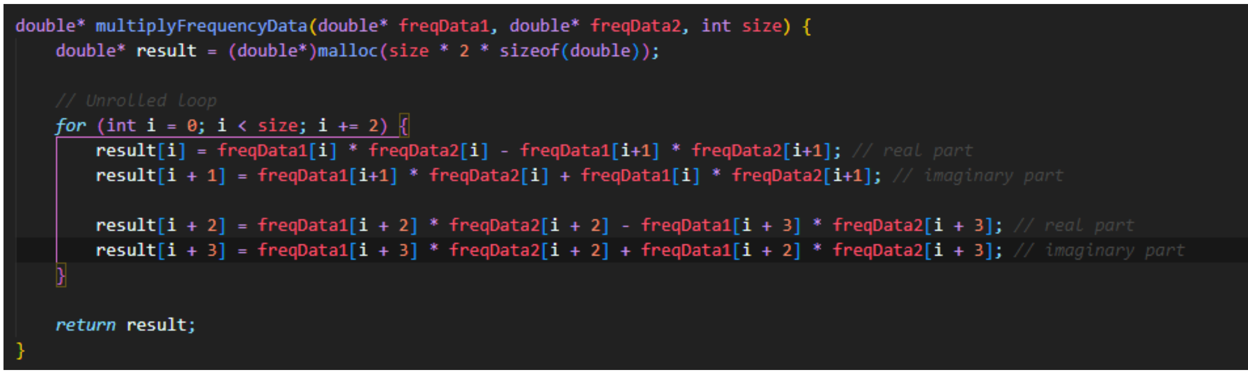
1) The first code tuning optimization I did is partially unrolling a loop which allows the code to process more elements at a time increasing the speed of the program.

Before:



```
double* multiplyFrequencyData(double* freqData1, double* freqData2, int size) {
    double* result = (double*)malloc(size * sizeof(double));
    for (int i = 0; i < size; i += 2) {
        result[i] = freqData1[i] * freqData2[i] - freqData1[i+1] * freqData2[i+1]; // real part
        result[i + 1] = freqData1[i+1] * freqData2[i] + freqData1[i] * freqData2[i+1]; // imaginary part
    }
    return result;
}
```

After:



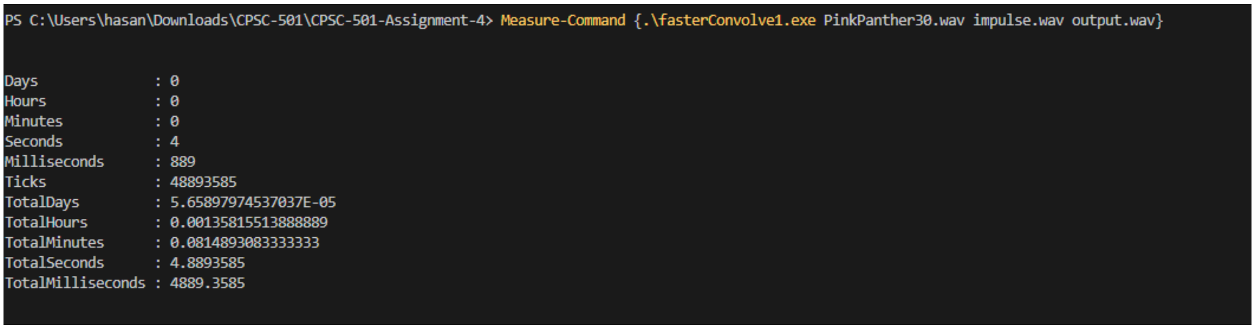
```
double* multiplyFrequencyData(double* freqData1, double* freqData2, int size) {
    double* result = (double*)malloc(size * 2 * sizeof(double));

    // Unrolled loop
    for (int i = 0; i < size; i += 2) {
        result[i] = freqData1[i] * freqData2[i] - freqData1[i+1] * freqData2[i+1]; // real part
        result[i + 1] = freqData1[i+1] * freqData2[i] + freqData1[i] * freqData2[i+1]; // imaginary part

        result[i + 2] = freqData1[i + 2] * freqData2[i + 2] - freqData1[i + 3] * freqData2[i + 3]; // real part
        result[i + 3] = freqData1[i + 3] * freqData2[i + 2] + freqData1[i + 2] * freqData2[i + 3]; // imaginary part
    }

    return result;
}
```

Optimized Timing:

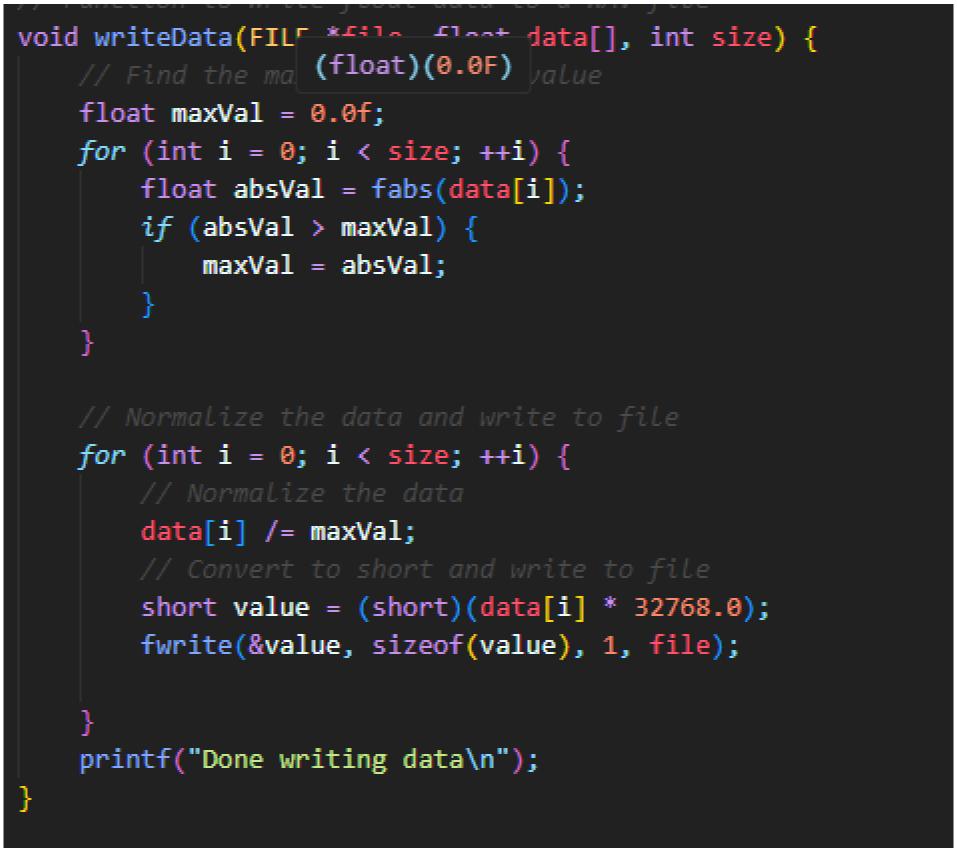


```
PS C:\Users\hasan\Downloads\CPSC-501\CPSC-501-Assignment-4> Measure-Command {.\fasterConvolve1.exe PinkPanther30.wav impulse.wav output.wav}


Days              : 0
Hours             : 0
Minutes           : 0
Seconds           : 4
Milliseconds      : 889
Ticks             : 48893585
TotalDays         : 5.65897974537037E-05
TotalHours        : 0.00135815513888889
TotalMinutes      : 0.0814893083333333
TotalSeconds      : 4.8893585
TotalMilliseconds : 4889.3585
```

2) The second code tuning optimization I did is minimizing work inside of a loop by combining the maximum absolute value calculation and normalization factor calculation. Furthermore I moved the normalization factor calculation outside the loop. This further increases the speed of the program

Before:

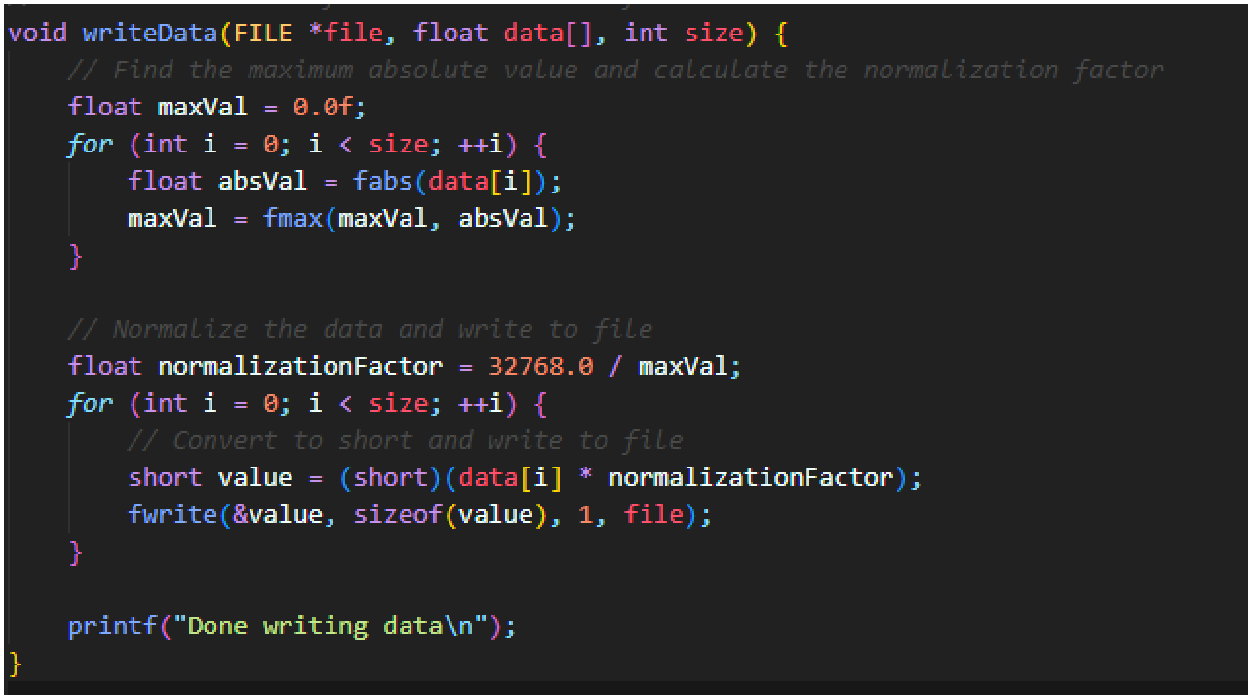


```
void writeData(FILE *file, float data[], int size) {
    // Find the ma (float)(0.0F) value
    float maxVal = 0.0f;
    for (int i = 0; i < size; ++i) {
        float absVal = fabs(data[i]);
        if (absVal > maxVal) {
            maxVal = absVal;
        }
    }

    // Normalize the data and write to file
    for (int i = 0; i < size; ++i) {
        // Normalize the data
        data[i] /= maxVal;
        // Convert to short and write to file
        short value = (short)(data[i] * 32768.0);
        fwrite(&value, sizeof(value), 1, file);

    }
    printf("Done writing data\n");
}
```

After:

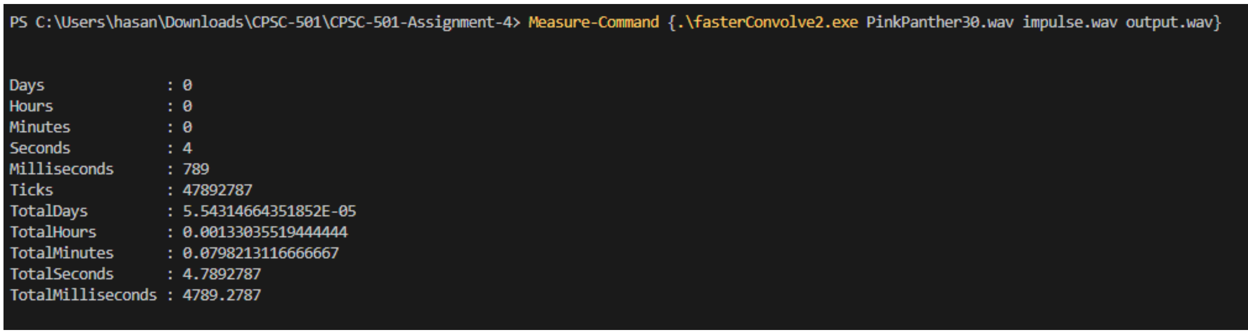


```
void writeData(FILE *file, float data[], int size) {
    // Find the maximum absolute value and calculate the normalization factor
    float maxVal = 0.0f;
    for (int i = 0; i < size; ++i) {
        float absVal = fabs(data[i]);
        maxVal = fmax(maxVal, absVal);
    }

    // Normalize the data and write to file
    float normalizationFactor = 32768.0 / maxVal;
    for (int i = 0; i < size; ++i) {
        // Convert to short and write to file
        short value = (short)(data[i] * normalizationFactor);
        fwrite(&value, sizeof(value), 1, file);
    }

    printf("Done writing data\n");
}
```

Optimization Time

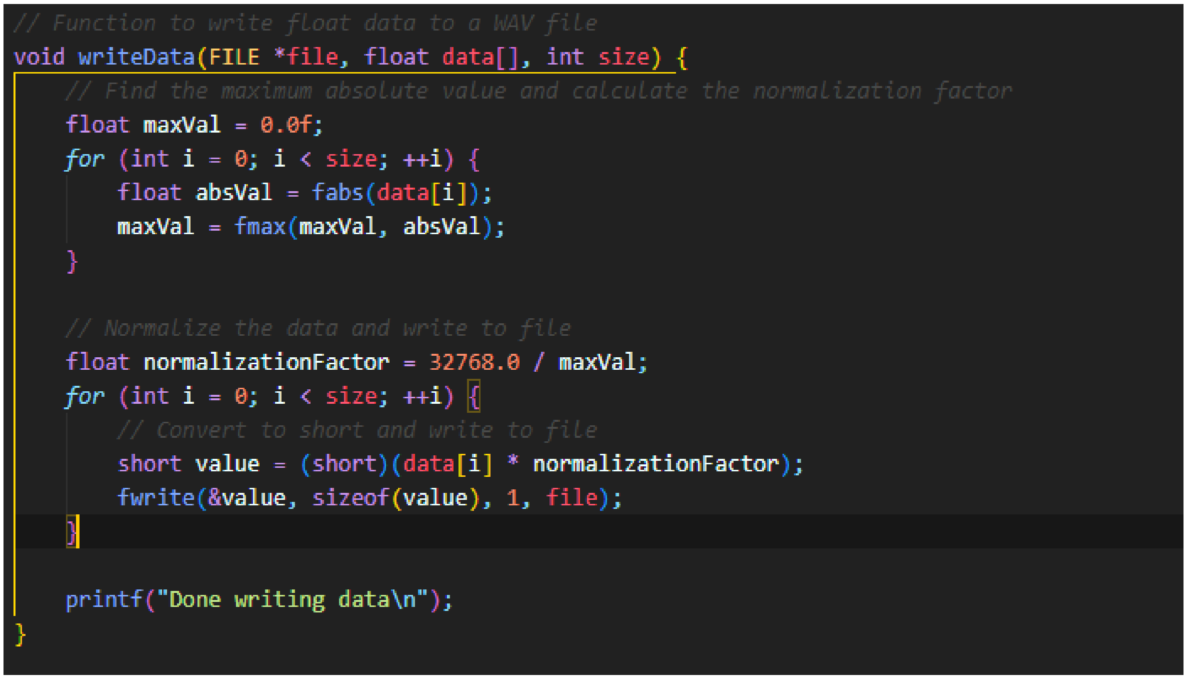


```
PS C:\Users\hasan\Downloads\CPSC-501\CPSC-501-Assignment-4> Measure-Command {.\fasterConvolve2.exe PinkPanther30.wav impulse.wav output.wav}


Days              : 0
Hours             : 0
Minutes           : 0
Seconds           : 4
Milliseconds      : 789
Ticks             : 47892787
TotalDays         : 5.54314664351852E-05
TotalHours        : 0.00133035519444444
TotalMinutes      : 0.0798213116666667
TotalSeconds      : 4.7892787
TotalMilliseconds : 4789.2787
```

3) The third code optimization I did is the code tuning strength reduction, the optimization is in the writeData function. The optimization I did here is to adjust my loop to use integers since it is more efficient than floating point operations.

Before:

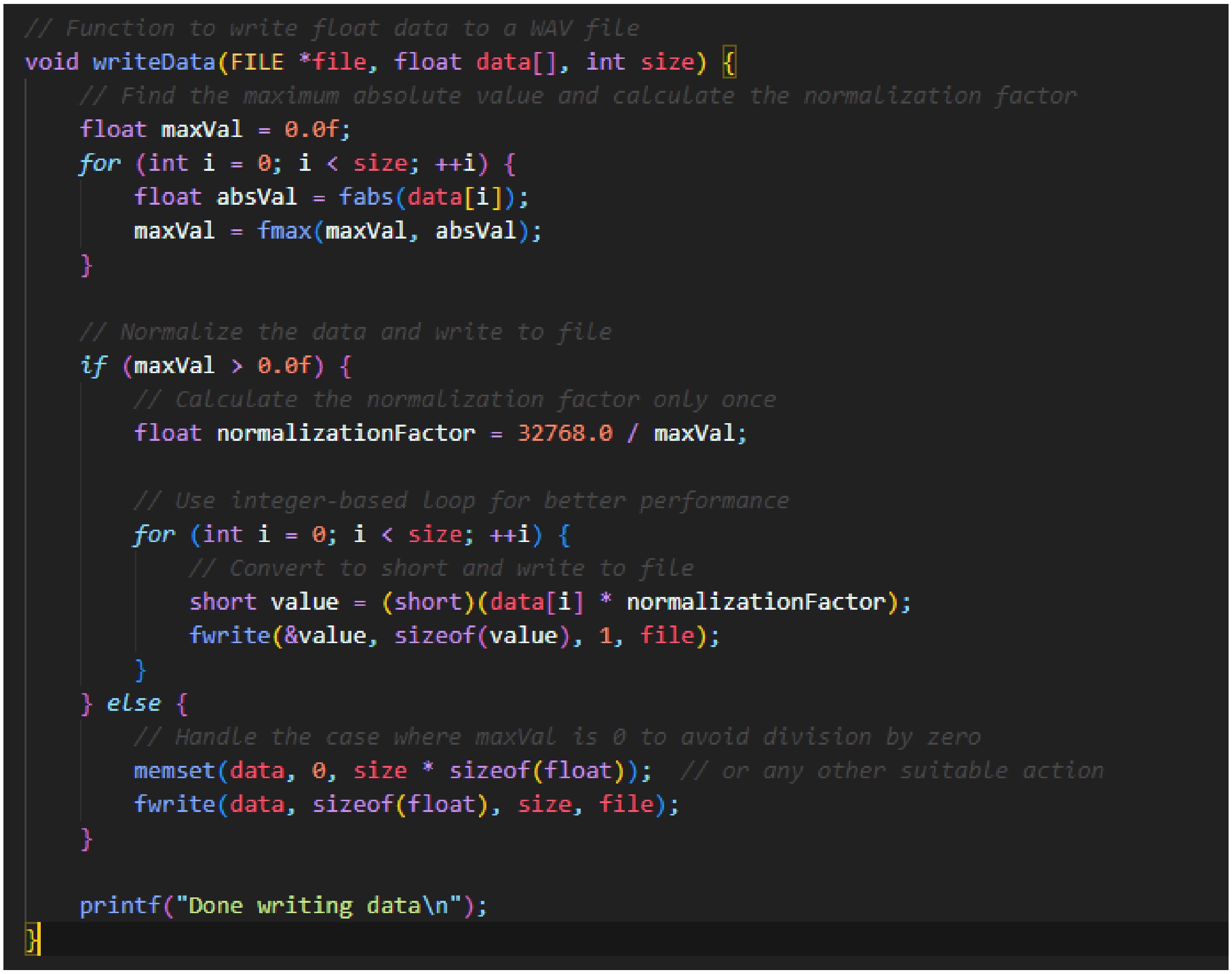


```
// Function to write float data to a WAV file
void writeData(FILE *file, float data[], int size) {
    // Find the maximum absolute value and calculate the normalization factor
    float maxVal = 0.0f;
    for (int i = 0; i < size; ++i) {
        float absVal = fabs(data[i]);
        maxVal = fmax(maxVal, absVal);
    }

    // Normalize the data and write to file
    float normalizationFactor = 32768.0 / maxVal;
    for (int i = 0; i < size; ++i) {
        // Convert to short and write to file
        short value = (short)(data[i] * normalizationFactor);
        fwrite(&value, sizeof(value), 1, file);
    }

    printf("Done writing data\n");
}
```

After:



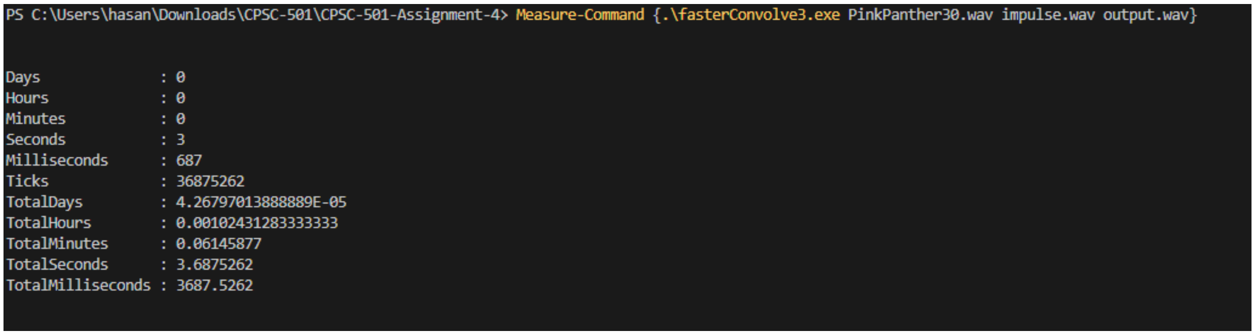
```c
// Function to write float data to a WAV file
void writeData(FILE *file, float data[], int size) {
    // Find the maximum absolute value and calculate the normalization factor
    float maxVal = 0.0f;
    for (int i = 0; i < size; ++i) {
        float absVal = fabs(data[i]);
        maxVal = fmax(maxVal, absVal);
    }

    // Normalize the data and write to file
    if (maxVal > 0.0f) {
        // Calculate the normalization factor only once
        float normalizationFactor = 32768.0 / maxVal;

        // Use integer-based loop for better performance
        for (int i = 0; i < size; ++i) {
            // Convert to short and write to file
            short value = (short)(data[i] * normalizationFactor);
            fwrite(&value, sizeof(value), 1, file);
        }
    } else {
        // Handle the case where maxVal is 0 to avoid division by zero
        memset(data, 0, size * sizeof(float));  // or any other suitable action
        fwrite(data, sizeof(float), size, file);
    }

    printf("Done writing data\n");
}
```

Optimization

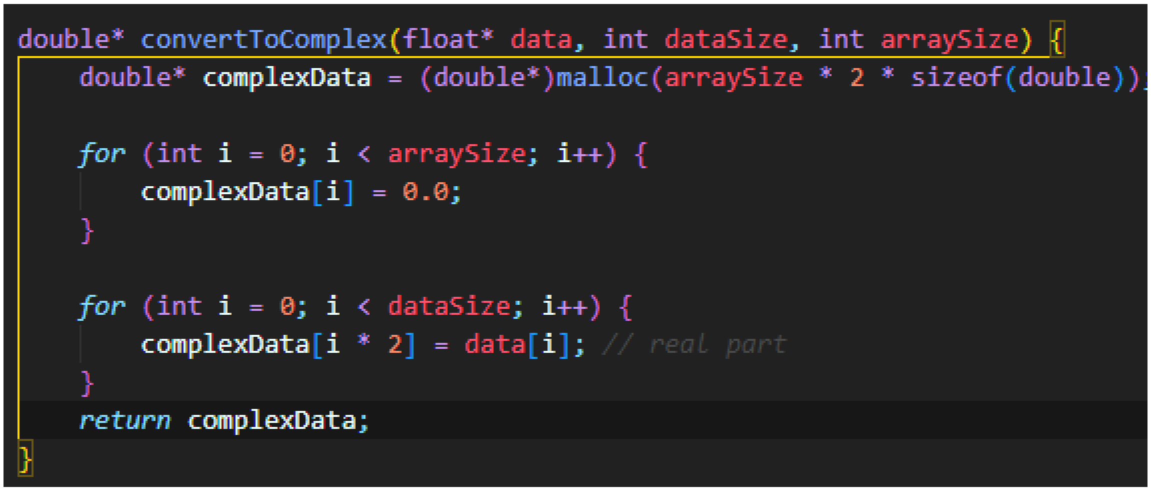


```
PS C:\Users\hasan\Downloads\CPSC-501\CPSC-501-Assignment-4> Measure-Command {.\fasterConvolve3.exe PinkPanther30.wav impulse.wav output.wav}


Days              : 0
Hours             : 0
Minutes           : 0
Seconds           : 3
Milliseconds      : 687
Ticks             : 36875262
TotalDays         : 4.26797013888889E-05
TotalHours        : 0.00102431283333333
TotalMinutes      : 0.06145877
TotalSeconds      : 3.6875262
TotalMilliseconds : 3687.5262
```

4) The fourth code tuning I did is data transformation on the method convertToComplex. I performed my code tuning by changing malloc to calloc in order to initialize my array with 0's instead of calling malloc and initializing it separately. This is a data transformation because it affects how the memory is initialized on complexData

Before:

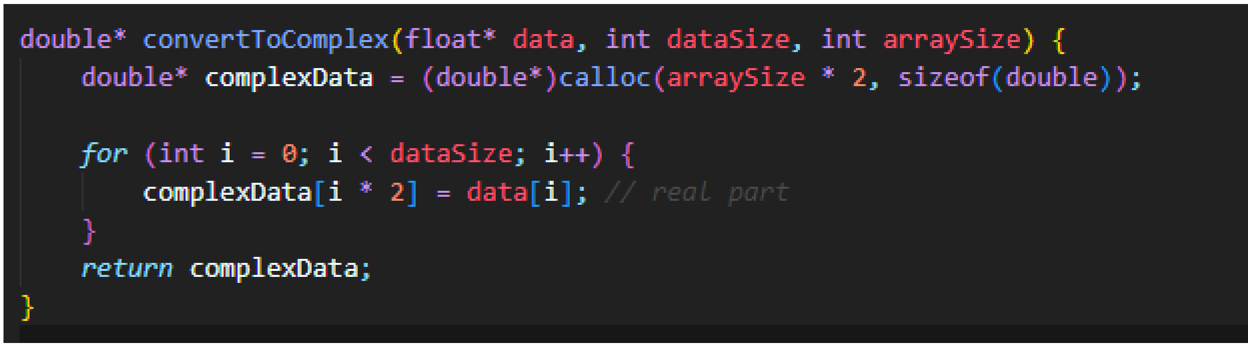


```
double* convertToComplex(float* data, int dataSize, int arraySize) {
    double* complexData = (double*)malloc(arraySize * 2 * sizeof(double));

    for (int i = 0; i < arraySize; i++) {
        complexData[i] = 0.0;
    }

    for (int i = 0; i < dataSize; i++) {
        complexData[i * 2] = data[i]; // real part
    }
    return complexData;
}
```

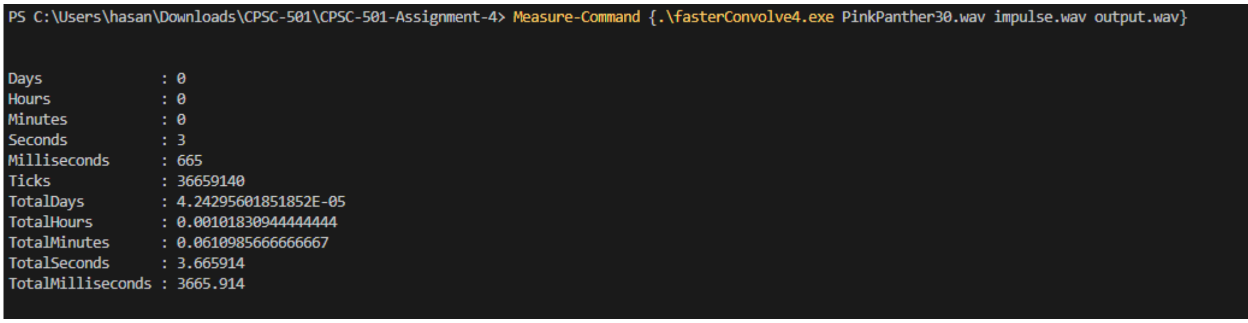
After:



```
double* convertToComplex(float* data, int dataSize, int arraySize) {
    double* complexData = (double*)calloc(arraySize * 2, sizeof(double));

    for (int i = 0; i < dataSize; i++) {
        complexData[i * 2] = data[i]; // real part
    }
    return complexData;
}
```

Optimization:

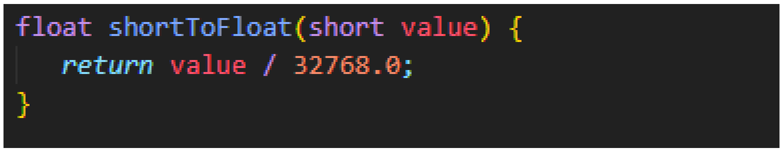


```
PS C:\Users\hasan\Downloads\CPSC-501\CPSC-501-Assignment-4> Measure-Command {.\fasterConvolve4.exe PinkPanther30.wav impulse.wav output.wav}


Days              : 0
Hours             : 0
Minutes           : 0
Seconds           : 3
Milliseconds      : 665
Ticks             : 36659140
TotalDays         : 4.24295601851852E-05
TotalHours        : 0.00101830944444444
TotalMinutes      : 0.0610985666666667
TotalSeconds      : 3.665914
TotalMilliseconds : 3665.914
```

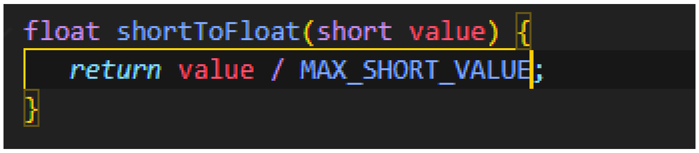
5) My final optimization is code tuning my expressions at compile time by using more constants. In this optimization I changed the max short value used in my shortToFloat function to use a constant value instead of a magic number which should lead to performance improvements.

Before:



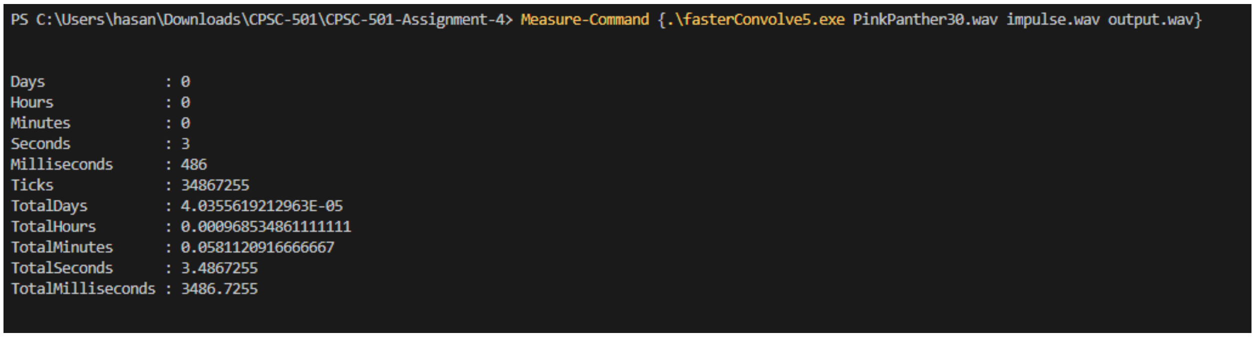
```
float shortToFloat(short value) {
    return value / 32768.0;
}
```

After:



```
float shortToFloat(short value) {
    return value / MAX_SHORT_VALUE;
}
```

Optimization:



```
PS C:\Users\hasan\Downloads\CPSC-501\CPSC-501-Assignment-4> Measure-Command {.\fasterConvolve5.exe PinkPanther30.wav impulse.wav output.wav}


Days              : 0
Hours             : 0
Minutes           : 0
Seconds           : 3
Milliseconds      : 486
Ticks             : 34867255
TotalDays         : 4.0355619212963E-05
TotalHours        : 0.000968534861111111
TotalMinutes      : 0.0581120916666667
TotalSeconds      : 3.4867255
TotalMilliseconds : 3486.7255
```

Chart displaying optimizations including base:

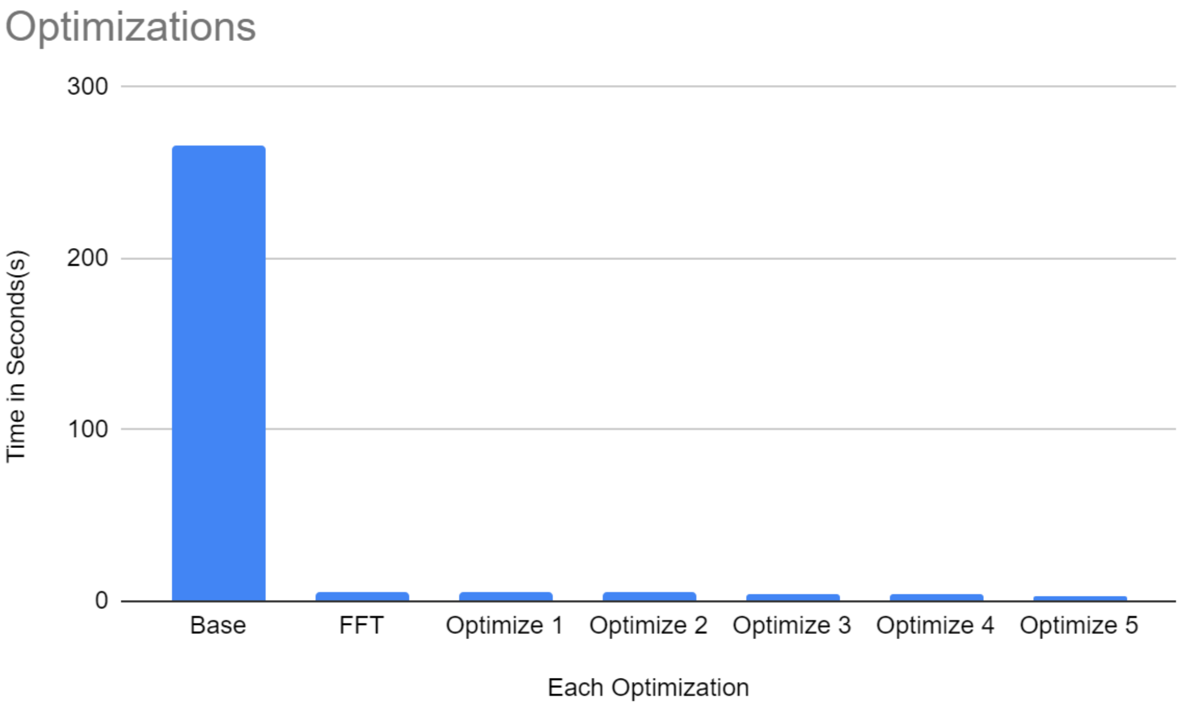
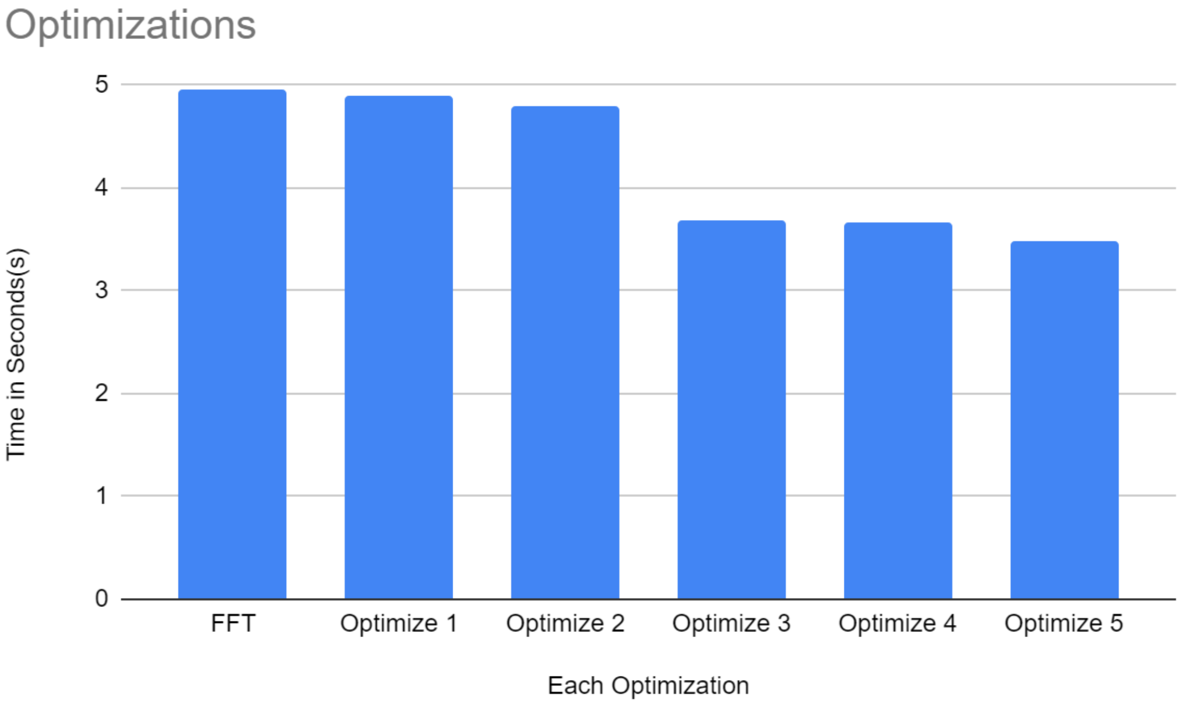


Chart displaying optimizations excluding base:


Optimizations
Time in Seconds(s)
FFT
Optimize 1
Optimize 2
Optimize 3
Optimize 4
Optimize 5
Each Optimization