# CPSC 501 - Assignment 2

**Name:** Alex Tanasescu

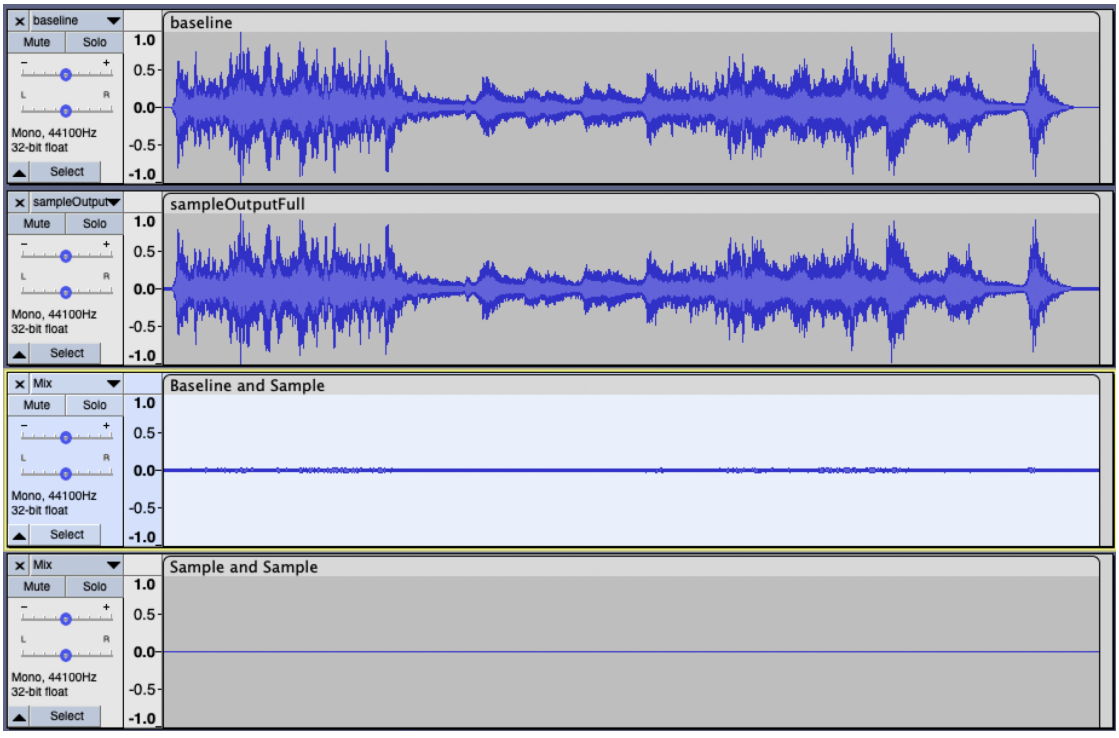
**UCID:** 30041538

**TA Notes:** Access to my gitlab repository can be found by clicking this link: https://gitlab.cpsc.ucalgary.ca/alex.tanasescu/assignment-2-alex-tanasescu. You should've also received an invitation to it in your email when I first invited you to my repo.

**Regression Testing:** For regression testing I will perform 3 different tests. The first test I will perform is the **audacity test:** importing the outputs from the two different programs in audacity, inverting one of the outputs and mixing them together in one track. If the two files are identical, then the frequencies will cancel each other out and the mix output will be silent. Below is an example comparing my baseline convolution program to the prof's sample output as well as identical sample output files for reference. We can see that the identical files make no sound and the graph is flat while there is a slight difference in my baseline and sample output. This can be confirmed with the **cmp test** where we use the linux command cmp to see if two files are the same, if they are there will be nothing returned however if they aren't, it will return the first byte at which they differ also demonstrated below with my baseline output and the prof's sample output. Finally the last test I will do is the **hearing test** where I listen if the outputs are similar. Even if the bits of the baseline and the sample are different, the outputs are similar which will be good enough for the purposes of this assignment.

**Timings:** To measure the timings, I ran each program 5 times (except the baseline since it takes a really long time) to have a fair sample size to understand how the code behaves. I then got rid of the lowest and high test time. I then averaged the 3 remaining times for the final comparison.

**Instructions to Run:** to run the program you can use the **g++ -O2 convolve.cpp -o convolve** to compile and **./convolve guitar_dry.wav big_hall_IR_mono.wav output.wav** to run. output.wav will be the output of the program which you can listen to

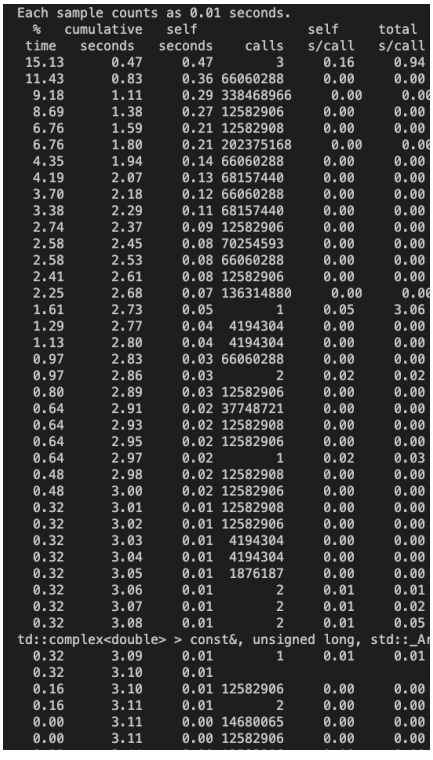


```
alex.tanasescu@linux02-wa:~/CPSC-501/A2$ cmp baseline.wav sampleOutputFull.wav
baseline.wav sampleOutputFull.wav differ: byte 45, line 1
alex.tanasescu@linux02-wa:~/CPSC-501/A2$
```

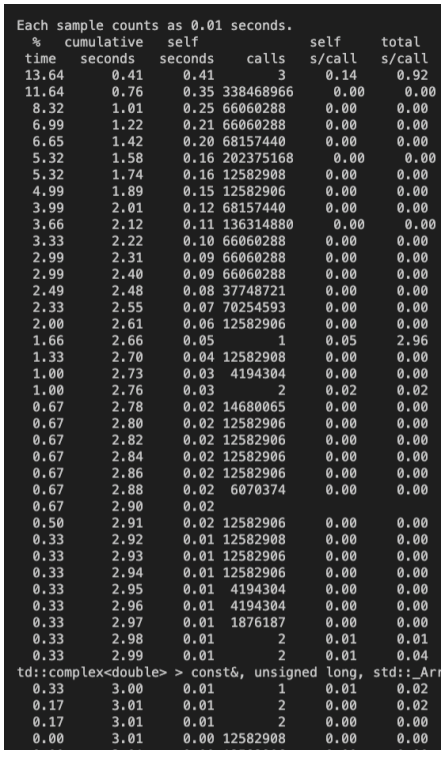
**Baseline:** The timings for my baseline took around just under 8 minutes (463 second/ 60 seconds = 7.7 min) as pictured below. I already demonstrated above how similar it is to the sample output even if bitwise it's different so it will make a good standard. This program is what we will use to compare all our optimizations.

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
100.11    463.59   463.59                             main
  0.01    463.63     0.04        2    20.03    20.03  readWavFile(int*, int*, char*)
  0.00    463.65     0.02        1    20.03    20.03  writeWavFile(double*, int, int, char*)
  0.00    463.65     0.00  1876187     0.00     0.00  std::abs(double)
  0.00    463.65     0.00       10     0.00     0.00  freadIntLSB(_IO_FILE*)
  0.00    463.65     0.00        8     0.00     0.00  freadShortLSB(_IO_FILE*)
  0.00    463.65     0.00        5     0.00     0.00  fwriteIntLSB(int, _IO_FILE*)
  0.00    463.65     0.00        4     0.00     0.00  fwriteShortLSB(short, _IO_FILE*)
  0.00    463.65     0.00        2     0.00     0.00  readWavFileHeader(int*, int*, _IO_FILE*)
  0.00    463.65     0.00        1     0.00     0.00  _GLOBAL__sub_I_main
  0.00    463.65     0.00        1     0.00     0.00  writeWavFileHeader(int, int, double, _IO_FILE*)
  0.00    463.65     0.00        1     0.00     0.00  __static_initialization_and_destruction_0(int, int)
```

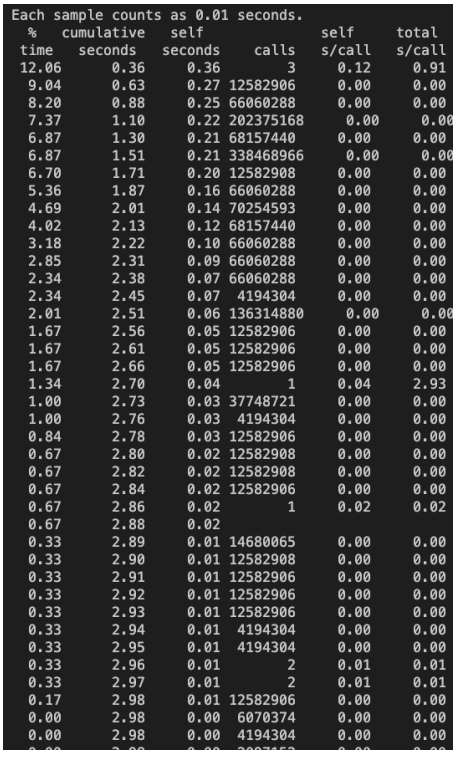
**Algorithmic Optimization:** The first optimization I did is performing the convolution using the Fast Fourier Transform. This increased the speed of execution dramatically from 7.7 minutes to 3.03 seconds using the timings below.



```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call
 15.13      0.47     0.47        3     0.16     0.94
 11.43      0.83     0.36 66060288     0.00     0.00
  9.18      1.11     0.29 338468966     0.00     0.00
  8.69      1.38     0.27 12582906     0.00     0.00
  6.76      1.59     0.21 12582908     0.00     0.00
  6.76      1.80     0.21 202375168     0.00     0.00
  4.35      1.94     0.14 66060288     0.00     0.00
  4.19      2.07     0.13 68157440     0.00     0.00
  3.70      2.18     0.12 66060288     0.00     0.00
  3.38      2.29     0.11 68157440     0.00     0.00
  2.74      2.37     0.09 12582906     0.00     0.00
  2.58      2.45     0.08 70254593     0.00     0.00
  2.58      2.53     0.08 66060288     0.00     0.00
  2.41      2.61     0.08 12582906     0.00     0.00
  2.25      2.68     0.07 136314880     0.00     0.00
  1.61      2.73     0.05        1     0.05     3.06
  1.29      2.77     0.04  4194304     0.00     0.00
  1.13      2.80     0.04  4194304     0.00     0.00
  0.97      2.83     0.03 66060288     0.00     0.00
  0.97      2.86     0.03        2     0.02     0.02
  0.80      2.89     0.03 12582906     0.00     0.00
  0.64      2.91     0.02 37748721     0.00     0.00
  0.64      2.93     0.02 12582908     0.00     0.00
  0.64      2.95     0.02 12582906     0.00     0.00
  0.64      2.97     0.02        1     0.02     0.03
  0.48      2.98     0.02 12582908     0.00     0.00
  0.48      3.00     0.02 12582906     0.00     0.00
  0.32      3.01     0.01 12582908     0.00     0.00
  0.32      3.02     0.01 12582906     0.00     0.00
  0.32      3.03     0.01  4194304     0.00     0.00
  0.32      3.04     0.01  4194304     0.00     0.00
  0.32      3.05     0.01  1876187     0.00     0.00
  0.32      3.06     0.01        2     0.01     0.01
  0.32      3.07     0.01        2     0.01     0.02
  0.32      3.08     0.01        2     0.01     0.05
td::complex<double> > const&, unsigned long, std::_Ar
  0.32      3.09     0.01        1     0.01     0.01
  0.32      3.10     0.01
  0.16      3.10     0.01 12582906     0.00     0.00
  0.16      3.11     0.01        2     0.00     0.00
  0.00      3.11     0.00 14680065     0.00     0.00
  0.00      3.11     0.00 12582906     0.00     0.00
```

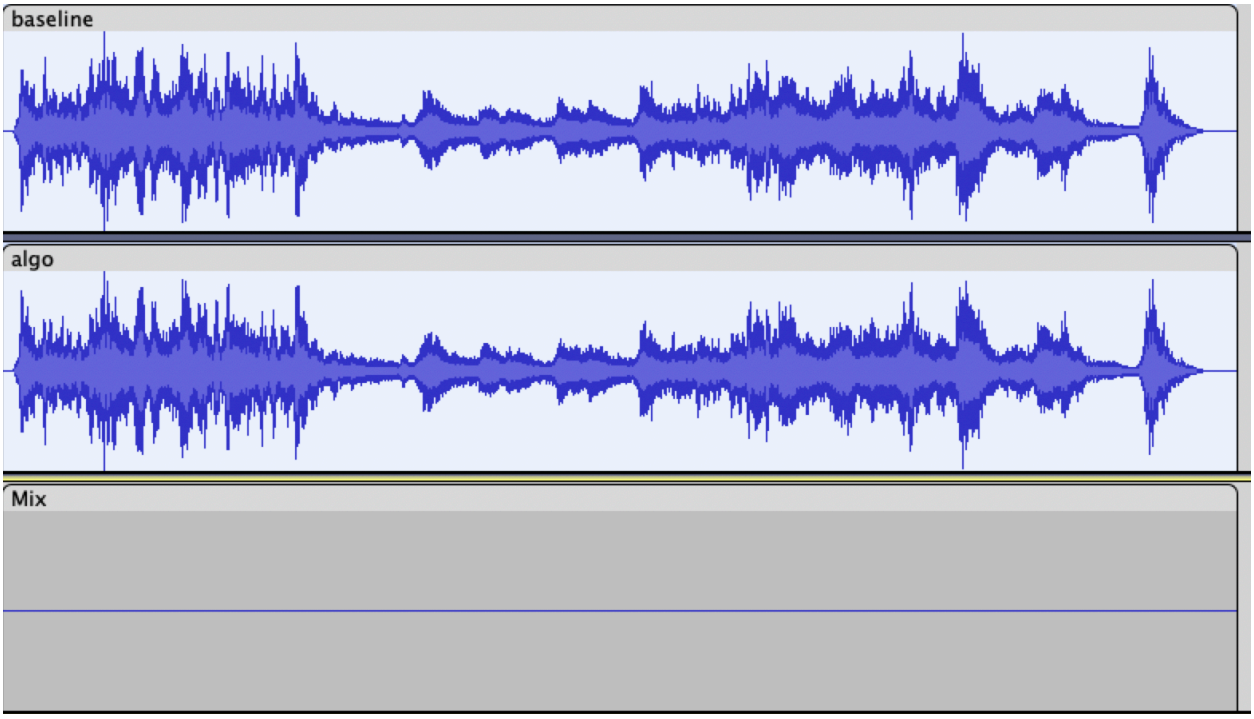


```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call
 13.64      0.41     0.41        3     0.14     0.92
 11.64      0.76     0.35 338468966     0.00     0.00
  8.32      1.01     0.25 66060288     0.00     0.00
  6.99      1.22     0.21 66060288     0.00     0.00
  6.65      1.42     0.20 68157440     0.00     0.00
  5.32      1.58     0.16 202375168     0.00     0.00
  5.32      1.74     0.16 12582908     0.00     0.00
  4.99      1.89     0.15 12582906     0.00     0.00
  3.99      2.01     0.12 68157440     0.00     0.00
  3.66      2.12     0.11 136314880     0.00     0.00
  3.33      2.22     0.10 66060288     0.00     0.00
  2.99      2.31     0.09 66060288     0.00     0.00
  2.99      2.40     0.09 66060288     0.00     0.00
  2.49      2.48     0.08 37748721     0.00     0.00
  2.33      2.55     0.07 70254593     0.00     0.00
  2.00      2.61     0.06 12582906     0.00     0.00
  1.66      2.66     0.05        1     0.05     2.96
  1.33      2.70     0.04 12582908     0.00     0.00
  1.00      2.73     0.03  4194304     0.00     0.00
  1.00      2.76     0.03        2     0.02     0.02
  0.67      2.78     0.02 14680065     0.00     0.00
  0.67      2.80     0.02 12582906     0.00     0.00
  0.67      2.82     0.02 12582906     0.00     0.00
  0.67      2.84     0.02 12582906     0.00     0.00
  0.67      2.86     0.02 12582906     0.00     0.00
  0.67      2.88     0.02  6070374     0.00     0.00
  0.67      2.90     0.02
  0.50      2.91     0.02 12582906     0.00     0.00
  0.33      2.92     0.01 12582908     0.00     0.00
  0.33      2.93     0.01 12582906     0.00     0.00
  0.33      2.94     0.01 12582906     0.00     0.00
  0.33      2.95     0.01  4194304     0.00     0.00
  0.33      2.96     0.01  4194304     0.00     0.00
  0.33      2.97     0.01  1876187     0.00     0.00
  0.33      2.98     0.01        2     0.01     0.01
  0.33      2.99     0.01        2     0.01     0.04
td::complex<double> > const&, unsigned long, std::_Arr
  0.33      3.00     0.01        1     0.01     0.02
  0.17      3.01     0.01        2     0.00     0.02
  0.17      3.01     0.01        2     0.00     0.00
  0.00      3.01     0.00 12582908     0.00     0.00
```



```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call
 12.06      0.36     0.36        3     0.12     0.91
  9.04      0.63     0.27 12582906     0.00     0.00
  8.20      0.88     0.25 66060288     0.00     0.00
  7.37      1.10     0.22 202375168     0.00     0.00
  6.87      1.30     0.21 68157440     0.00     0.00
  6.87      1.51     0.21 338468966     0.00     0.00
  6.70      1.71     0.20 12582908     0.00     0.00
  5.36      1.87     0.16 66060288     0.00     0.00
  4.69      2.01     0.14 70254593     0.00     0.00
  4.02      2.13     0.12 68157440     0.00     0.00
  3.18      2.22     0.10 66060288     0.00     0.00
  2.85      2.31     0.09 66060288     0.00     0.00
  2.34      2.38     0.07 66060288     0.00     0.00
  2.34      2.45     0.07  4194304     0.00     0.00
  2.01      2.51     0.06 136314880     0.00     0.00
  1.67      2.56     0.05 12582906     0.00     0.00
  1.67      2.61     0.05 12582906     0.00     0.00
  1.67      2.66     0.05 12582906     0.00     0.00
  1.34      2.70     0.04        1     0.04     2.93
  1.00      2.73     0.03 37748721     0.00     0.00
  1.00      2.76     0.03  4194304     0.00     0.00
  0.84      2.78     0.03 12582906     0.00     0.00
  0.67      2.80     0.02 12582908     0.00     0.00
  0.67      2.82     0.02 12582908     0.00     0.00
  0.67      2.84     0.02 12582906     0.00     0.00
  0.67      2.86     0.02        1     0.02     0.02
  0.67      2.88     0.02
  0.33      2.89     0.01 14680065     0.00     0.00
  0.33      2.90     0.01 12582908     0.00     0.00
  0.33      2.91     0.01 12582906     0.00     0.00
  0.33      2.92     0.01 12582906     0.00     0.00
  0.33      2.93     0.01 12582906     0.00     0.00
  0.33      2.94     0.01  4194304     0.00     0.00
  0.33      2.95     0.01  4194304     0.00     0.00
  0.33      2.96     0.01        2     0.01     0.01
  0.33      2.97     0.01        2     0.01     0.01
  0.17      2.98     0.01 12582906     0.00     0.00
  0.00      2.98     0.00  6070374     0.00     0.00
  0.00      2.98     0.00  4194304     0.00     0.00
```

We can also see from the audacity test and the cmp test that the baseline and algorithm optimization wav files are identical. I also hear no difference between the two sound files




```
alex.tanasescu@linux02-wa:~/CPSC-501/A2$ cmp baseline.wav algo.wav
alex.tanasescu@linux02-wa:~/CPSC-501/A2$
```

To perform the optimization I changed the code from the convolution given to us in class

```
89      double *outputArray = new double[outputArraySize];
90
91
92      for (int i = 0; i < IRArraySize; i++)
93      {
94          for (int j = 0; j < inputArraySize; j++)
95          {
96              outputArray[i + j] += IRArray[i] * inputArray[j];
97          }
98      }
99
```

To the following convolution using the Fast Fourier Transform algorithm:

```
149
150 double* convolution(double* inputArray, int inputArraySize, double* IRArray, int IRArraySize, int* outputArraySize)
151 {
152     double *outputArray = new double[*outputArraySize];
153     int complexArraySize = *outputArraySize;
154
155     if (!isPowerOfTwo(complexArraySize))
156     {
157         complexArraySize = nextPowTwo(complexArraySize);
158     }
159
160     double* inputPadded = new double[complexArraySize];
161     double* IRPadded = new double[complexArraySize];
162
163     for (int i = 0; i < complexArraySize; i++)
164     {
165         if (i < inputArraySize)
166         {
167             inputPadded[i] = inputArray[i];
168         }
169         else
170         {
171             inputPadded[i] = 0;
172         }
173     }
174
175     for (int i = 0; i < complexArraySize; i++)
176     {
177         if (i < IRArraySize)
178         {
179             IRPadded[i] = IRArray[i];
180         }
181         else
182         {
183             IRPadded[i] = 0;
184         }
185     }
186
187     double* inputComplex = new double[complexArraySize*2];
188     double* IRComplex= new double[complexArraySize*2];
189
190     for (int i = 0; i < complexArraySize; i++)
191     {
192         inputComplex[i*2] = inputPadded[i];
193     }
194
195     for (int i = 0; i < complexArraySize; i++)
196     {
197         inputComplex[i*2+1] = 0;
198     }
199
200     for (int i = 0; i < complexArraySize; i++) {
201         IRComplex[i*2] = IRPadded[i];
202     }
203
204     for (int i = 0; i < complexArraySize; i++) {
205         IRComplex[i*2+1] = 0;
206     }
207
208     CArray inputData((Complex*)inputComplex, complexArraySize);
209     CArray IRData((Complex*)IRComplex, complexArraySize);
210
211     fft(inputData);
212     fft(IRData);
213
214     double* outputData = new double[*outputArraySize];
215     for (int i = 0; i < inputData.size(); i++)
216     {
217         inputData[i] = inputData[i]*IRData[i];
218     }
219
220     ifft(inputData);
221
222     for (int i = 0; i < *outputArraySize; i++)
223     {
224
225         outputData[i] = inputData[i].real();
226     }
227
228     return outputData;
229 }
230
```

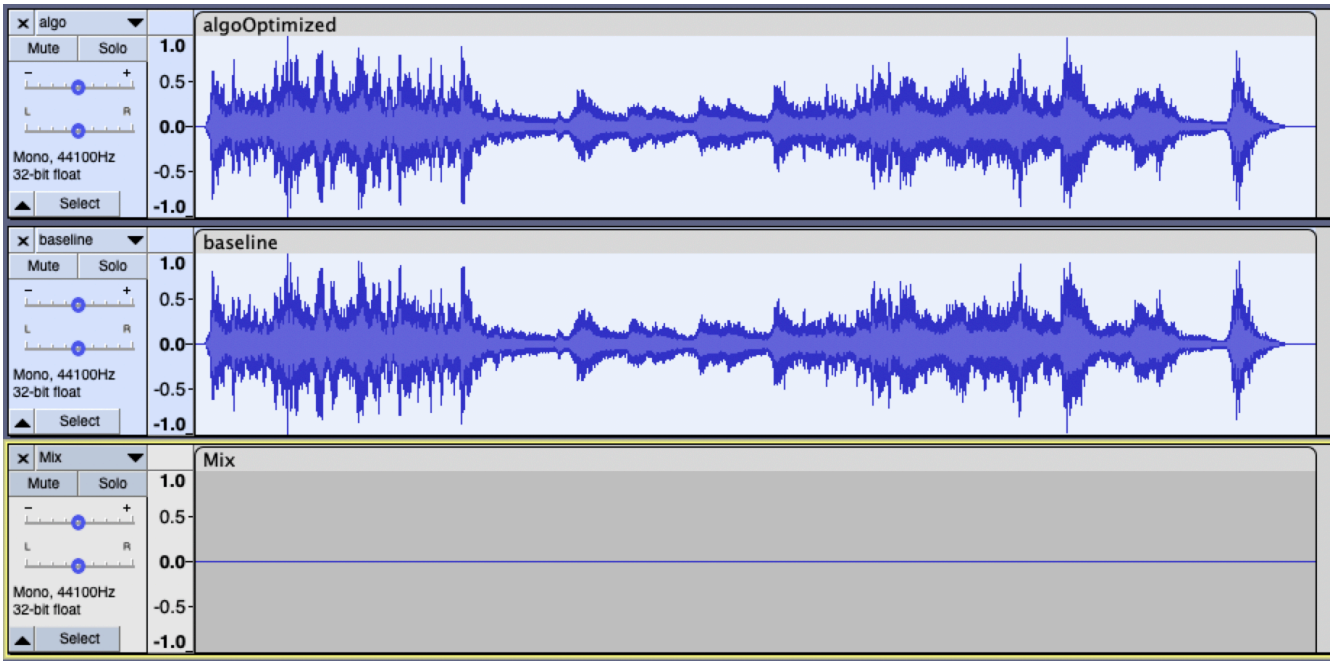
**Compiler Optimization:** The second optimization I performed is the compiler optimization. I used the -O2 flag to optimize my code. We can see another dramatic speed up from the 3.03 seconds we had with the basic Fast Fourier Transformation to an average of 0.58 seconds using the Fast Fourier Transformation with the optimization flag. We can also see that both wav files are exactly the same with our audacity and cmp test which is expected since we didn't change any code. As well as sounding the same to my ears. We will use this flag for all of our following optimizations.

```
alex.tanasescu@linux02-wa:~/CPSC-501/A2$ cmp algoOptimized.wav baseline.wav
alex.tanasescu@linux02-wa:~/CPSC-501/A2$
```

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call
 86.89      0.52     0.52        3   173.78   173.78
 10.03      0.58     0.06        1    60.15   591.50
  1.67      0.59     0.01        1    10.03    10.03
  1.67      0.60     0.01        1    10.03   183.80
  0.00      0.60     0.00  4194304     0.00     0.00
  0.00      0.60     0.00       10     0.00     0.00
  0.00      0.60     0.00        8     0.00     0.00
```

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call
 87.72      0.53     0.53        3   175.45   175.45
 10.03      0.59     0.06        1    60.15   591.50
  0.84      0.59     0.01  4194304     0.00     0.00
  0.84      0.60     0.01        2     2.51     2.51
  0.84      0.60     0.01        1     5.01     5.01
  0.00      0.60     0.00       10     0.00     0.00
  0.00      0.60     0.00        8     0.00     0.00
```

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call
 87.50      0.48     0.48        3   160.41   160.41
 12.76      0.55     0.07        1    70.18   551.40
  0.00      0.55     0.00  4194304     0.00     0.00
  0.00      0.55     0.00       10     0.00     0.00
  0.00      0.55     0.00        8     0.00     0.00
```

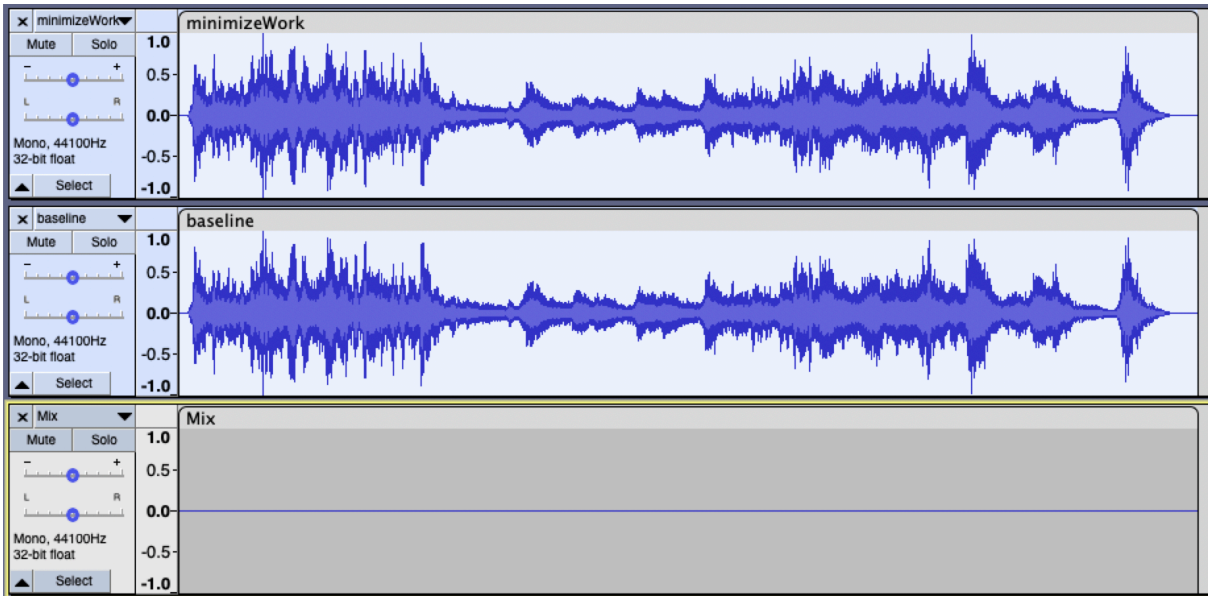


**1st Code Tuning - Minimize Work:** The first code tuning I performed is the minimize work. Inside my Fast Fourier method, that contains a loop and inside that it calculates the polar form of a complex number. This calculation is performed every time the loop is run. To optimize this, I simply did a calculation outside the loop and multiplied that with the loop counter. Shown below is snippets of my code before and after optimization for reference.

```
111     // combine
112     for (size_t k = 0; k < N/2; ++k)
113     {
114         Complex t = polar(1.0, -2 * PI * k / N) * odd[k];
115         x[k    ] = even[k] + t;
116         x[k+N/2] = even[k] - t;
117     }
```

```
111     // combine
112     double toMultiply = -2 * PI * 1/N;
113     for (size_t k = 0; k < N/2; ++k)
114     {
115         Complex t = polar(1.0, toMultiply*k) * odd[k];
116         x[k    ] = even[k] + t;
117         x[k+N/2] = even[k] - t;
118     }
119 }
120
```

We can see in the audacity and cmp test that the baseline and the output of the code tuning have the same contents so nothing has been changed as expected, as well as sounding the same. We can also see in the timings that there is a small but noticeable speedup in the code when we apply the code tuning technique in our original code along with the compiler optimization flag from 0.58 seconds to an average of 0.54. This is all shown on the next page




```
alex.tanasescu@linux02-wa:~/CPSC-501/A2$ cmp minimizeWork.wav baseline.wav
alex.tanasescu@linux02-wa:~/CPSC-501/A2$
```

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call
 88.91      0.47     0.47        3   157.07   157.07
  9.46      0.52     0.05        1    50.13   521.33
  1.89      0.53     0.01        1    10.03    10.03
  0.00      0.53     0.00  4194304     0.00     0.00
```

```
 time   seconds   seconds    calls  ms/call  ms/call
 88.16      0.51     0.51        3   170.43   170.43
  8.64      0.56     0.05        1    50.13   561.43
  1.73      0.57     0.01        2     5.01     5.01
  1.73      0.58     0.01        1    10.03    10.03
  0.00      0.58     0.00  4194304     0.00     0.00
```

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call
 86.76      0.45     0.45        3   150.38   150.38
  7.71      0.49     0.04        1    40.10   501.28
  1.93      0.50     0.01        2     5.01     5.01
  1.93      0.51     0.01        2     5.01     5.01
unClos<std::_ValArray, std::complex<double> >, std::co
  1.93      0.52     0.01        1    10.03    10.03
  0.00      0.52     0.00  4194304     0.00     0.00
  0.00      0.52     0.00       10     0.00     0.00
```

**2nd Code Tuning - Loop Jamming:** The second code tuning technique I performed is loop jamming. This is when two or more loops are merged in a single loop. This helps speed up execution time since it reduces the time to compile each separate loop. I had two different sets of two loops. I ended up merging the two loops in each set into one loop since they had the same loop counter and exit condition as shown in the code below.

```
177     for (int i = 0; i < complexArraySize; i++)
178     {
179         inputComplex[i*2] = inputPadded[i];
180     }
181
182     for (int i = 0; i < complexArraySize; i++)
183     {
184         inputComplex[i*2+1] = 0;
185     }
186
187     for (int i = 0; i < complexArraySize; i++) {
188         IRComplex[i*2] = IRPadded[i];
189     }
190
191     for (int i = 0; i < complexArraySize; i++) {
192         IRComplex[i*2+1] = 0;
193     }
194
```

```
176
177     for (int i = 0; i < complexArraySize; i++)
178     {
179         inputComplex[i*2] = inputPadded[i];
180         inputComplex[i*2+1] = 0;
181     }
182
183     for (int i = 0; i < complexArraySize; i++) {
184         IRComplex[i*2] = IRPadded[i];
185         IRComplex[i*2+1] = 0;
186     }
```

We can see on the average of the timings below that there is another small but noticeable speed up from the timing. We went from our time of 0.54 from the first optimization to 0.51 now.

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call
 85.64      0.41     0.41        3   137.02   137.02
  6.27      0.44     0.03        1    30.08   177.12
  6.27      0.47     0.03
  2.09      0.48     0.01  4194304     0.00     0.00
  0.00      0.48     0.00        2     0.00     0.00
```

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call
 91.73      0.43     0.43        3   143.70   143.70
  8.53      0.47     0.04
  0.00      0.47     0.00  4194304     0.00     0.00
  0.00      0.47     0.00        2     0.00     0.00
```

```
 time   seconds   seconds    calls  ms/call  ms/call
 88.16      0.51     0.51        3   170.43   170.43
  8.64      0.56     0.05        1    50.13   561.43
  1.73      0.57     0.01        2     5.01     5.01
  1.73      0.58     0.01        1    10.03    10.03
  0.00      0.58     0.00  4194304     0.00     0.00
```

We can also see from the audacity and cmp test that the two wav files are the same in content. The hearing test confirms this as I hear no difference when playing the two sounds

```
alex.tanasescu@ms160-1ee:~/CPSC-501/A2$ cmp loopJam.wav baseline.wav
alex.tanasescu@ms160-1ee:~/CPSC-501/A2$
```

**3rd Code Tuning - Unswitching:** My 3rd code tuning is a form of unswitching. My initial code has loops that contain an if-else statement. This increases execution time since the compiler has to evaluate the if-else statement each time the loop is run. To fix this, since I know the if statement is going to be true for the first half of the loop and the else statement is going to be true for the second half, I split the loop into two sub loops and put the respective statements from each condition in each respective loop. Below is my code for reference

```
149
150     for (int i = 0; i < complexArraySize; i++)
151     {
152         if (i < inputArraySize)
153         {
154             inputPadded[i] = inputArray[i];
155         }
156         else
157         {
158             inputPadded[i] = 0;
159         }
160     }
161
162     for (int i = 0; i < complexArraySize; i++)
163     {
164         if (i < IRArraySize)
165         {
166             IRPadded[i] = IRArray[i];
167         }
168         else
169         {
170             IRPadded[i] = 0;
171         }
172     }
```

```
149
150     for (int i = 0; i < inputArraySize; i++)
151     {
152         inputPadded[i] = inputArray[i];
153     }
154
155     for (int i = inputArraySize; i < complexArraySize; i++ )
156     {
157         inputPadded[i] = 0;
158     }
159
160     for (int i = 0; i < IRArraySize; i++)
161     {
162         IRPadded[i] = IRArray[i];
163     }
164
165     for (int i = IRArraySize; i < complexArraySize; i++ )
166     {
167         IRPadded[i] = 0;
168     }
169
```

We can see from the timings below that there is another small speedup but it is still there the speed up is another 0.03 seconds from our timings of before of 0.51 to 0.48

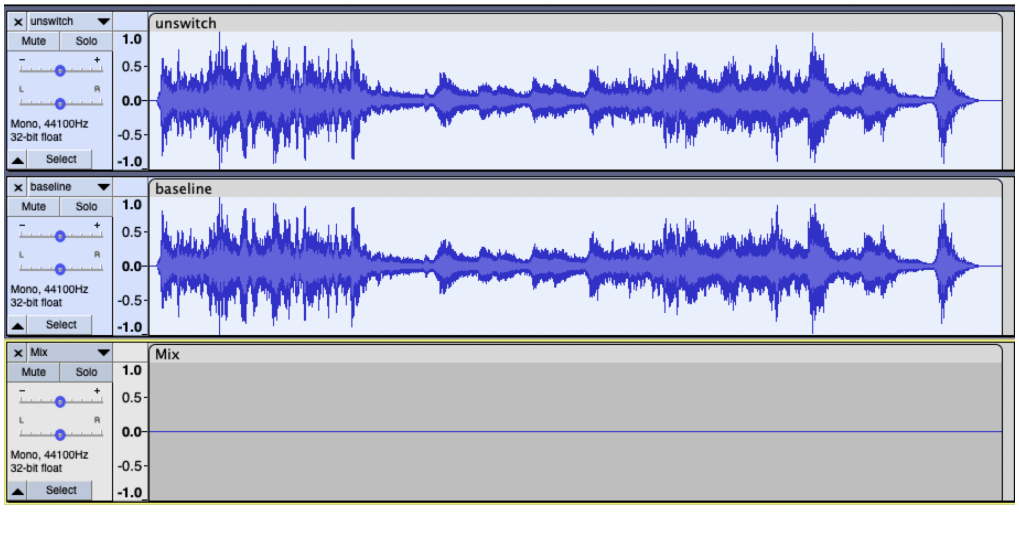
```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call
 80.83      0.40     0.40        3   132.02   132.02
  6.14      0.43     0.03
  6.14      0.46     0.03
  4.09      0.48     0.02        1    20.05   162.10
  2.05      0.49     0.01  4194304     0.00     0.00
  1.02      0.49     0.01
  0.00      0.49     0.00        2     0.00     0.00
  0.00      0.49     0.00        1     0.00     0.00
```

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call
 90.24      0.45     0.45        3   150.40   150.40
  8.02      0.49     0.04
  2.01      0.50     0.01
  0.00      0.50     0.00  4194304     0.00     0.00
  0.00      0.50     0.00        2     0.00     0.00
```

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call
 89.37      0.41     0.41        3   137.03   137.03
  6.54      0.44     0.03
  2.18      0.45     0.01  4194304     0.00     0.00
  2.18      0.46     0.01
  0.00      0.46     0.00        2     0.00     0.00
```

We can see in the audacity and cmp test that the baseline and the output of the code tuning have the same contents so nothing has been changed as expected. As well as sounding the same to my ears.

```
alex.tanasescu@ms160-1ee:~/CPSC-501/A2$ cmp unswitch.wav baseline.wav
alex.tanasescu@ms160-1ee:~/CPSC-501/A2$
```



**4th Code Tuning - Partial Unrolling:** My last and final code tuning I performed on my code is Partial Unrolling. Partial unrolling is an optimization technique used to decrease execution time. We do this by reducing the number of iterations the loop runs for and adjusts for these cases in the loop body and and if statements after the loop body. Pictures are shown below for reference to explain how I applied this technique in my code.

```
double* outputData = new double[*outputArraySize];
for (int i = 0; i < inputData.size(); i++)
{
    inputData[i] = inputData[i]*IRData[i];
}

ifft(inputData);

for (int i = 0; i < *outputArraySize; i++)
{

    outputData[i] = inputData[i].real();
}
```

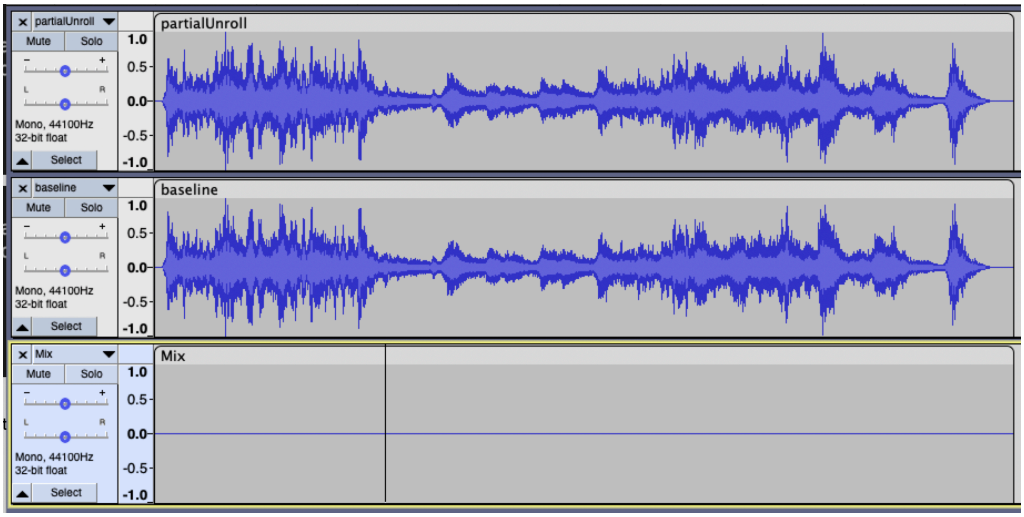
```
    int i;
    for ( i = 0; i < inputData.size() - 1; i+=2)
    {
        inputData[i] = inputData[i]*IRData[i];
        inputData[i+1] = inputData[i+1]*IRData[i+1];
    }
    if (i == (inputData.size() - 1))
    {
        inputData[i] = inputData[i]*IRData[i];
    }

    ifft(inputData);
    int j;
    for (j = 0; j < *outputArraySize - 1; j+=2)
    {

        outputData[j] = inputData[j].real();
        outputData[j+1] = inputData[j+1].real();

    }

    if (j == ( *outputArraySize - 1))
    {
        outputData[j] = inputData[j].real();
    }
```

We can see the from the timings below we have as big of a speedup as the other optimizations (0.03 seconds) as the average speed of our code improved from 0.48 seconds to 0.45 seconds. We can also see from the audacity and cmp test that the two wav files are the same in content. The hearing test confirms this as I hear no difference when playing the two sounds

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call
 89.11      0.40     0.40        3   133.66   133.66
  8.91      0.44     0.04
  2.23      0.45     0.01        1    10.02   143.69
  0.00      0.45     0.00  4194304     0.00     0.00
```

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call
 90.92      0.39     0.39        3   130.32   130.32
  9.33      0.43     0.04
  0.00      0.43     0.00  4194304     0.00     0.00
  0.00      0.43     0.00        2     0.00     0.00
```

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call
 89.58      0.42     0.42        3   140.34   140.34
  6.40      0.45     0.03
  2.13      0.46     0.01  4194304     0.00     0.00
  2.13      0.47     0.01        1    10.02   160.39
  0.00      0.47     0.00        2     0.00     0.00
```

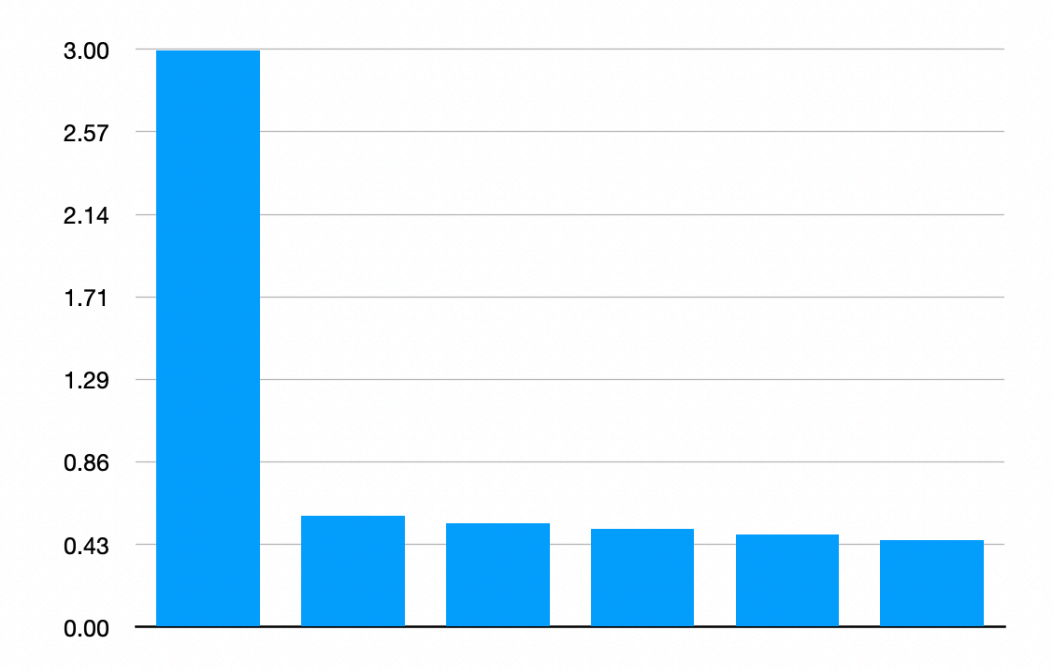



```
alex.tanasescu@linux01-ec:~/CPSC-501/A2$ cmp partialUnroll.wav baseline.wav
alex.tanasescu@linux01-ec:~/CPSC-501/A2$
```

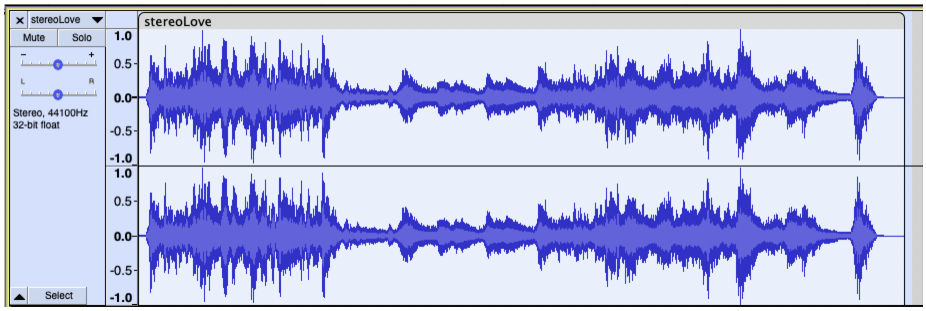
**Conclusion:** As we can see from the chart and table below(I didn't include baseline in the chart as it took too long and we wouldn't be able to how the other optimizations performed) the code tuning optimizations I applied gave us a small boost in speed performance but not as much as the compiler optimization or the algorithmic substitution but it's still something that you can do if you want to speed up your code

Execution Speed for Different Optimizations

| Optimization | Execution Speed |
|---|---|
| **Baseline** | 463.59 |
| **FFT Algorithm w/o O2 Flag** | 3.03 |
| **FFT Algorithm w/ O2 Flag** | 0.58 |
| **Minimize Work w/ O2 Flag** | 0.54 |
| **Loop Jamming w/ O2 Flag** | 0.51 |
| **Unswitching w/ O2 Flag** | 0.48 |
| **Partial Unrolling w/O2 Flag** | 0.45 |



**Bonus:** I did attempt and finished the bonus portion of the assignment. I convolved the dry recording of the guitar with the stereo big hall impulse recording file. Not quite sure on how to test if it worked properly but I managed to download a VS Code extension that shows you the properties of a wav file and it shows that there are two channels.I also checked Audacity and that shows that there are two channels now so I'm assuming it worked. Below is also the code I added to my program to detect stereo files basically I check if the IR recording has two channels and if it does I convert the dry recording to stereo as well and tell the write wav file header method to write to a stereo file.




| Key | Value |
|---|---|
| format | 1 (uncompressed PCM) |
| number of channel | 2 (stereo) |
| sampleRate | 44100 |
| byteRate | 176400 |
| blockAlign | 4 |
| bitsPerSample (bit depth) | 16 |
| fileSize | 7430730 byte |
| duration | 42.1240589569161s |

```
if (IRChannels = 2)
{
    inputChannels = 2;
    outputChannels = 2;
    inputArray = convertToStereo(inputArray, &inputArraySize, inputArraySize);
}
```

```
/*  Number of channels  */

if (channels == 1)
{
    fwriteShortLSB((short) MONOPHONIC, outputFile);
}
else
{
    fwriteShortLSB((short) STEREOPHONIC, outputFile);
}
```