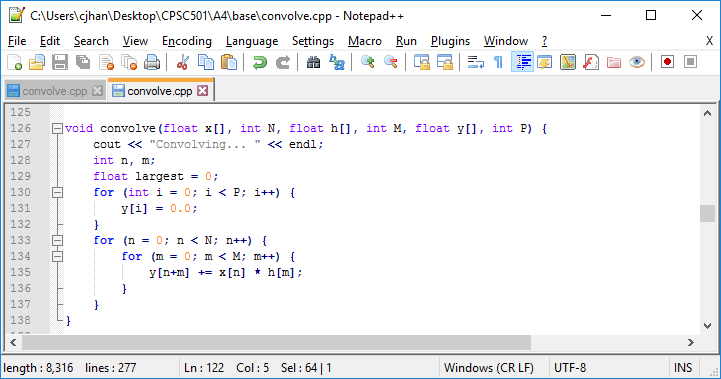
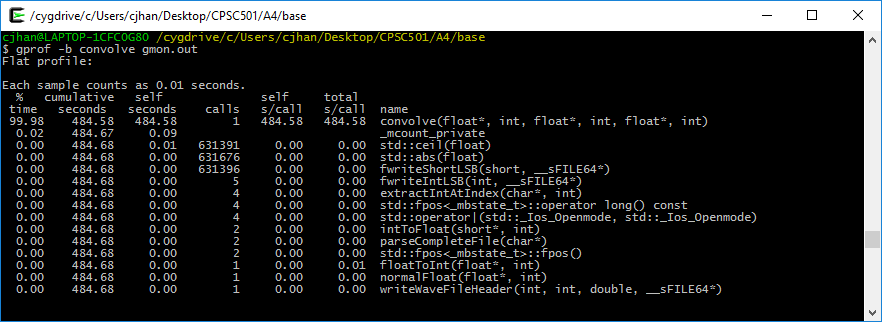
**CPSC 501 Assignment 4 Optimization Report**

Baseline Program: Our implementation begins with an input-side time-domain convolution algorithm. It operates by parsing both the input and impulse response .wav files for their respective signal data. Once the samples have been extracted, they convolved together using the following convolve function:

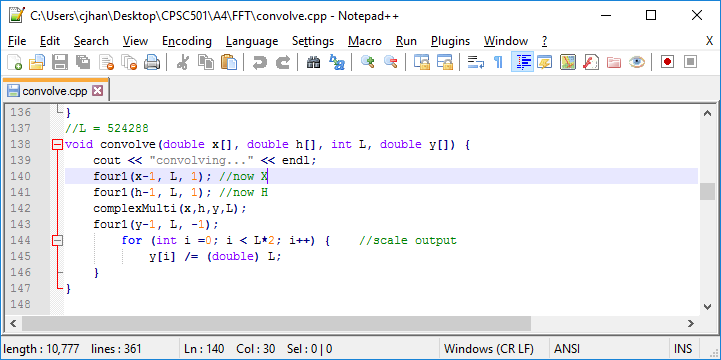


This convolution function, while accurate, is extremely slow. Timing data, collected at initial execution of the convolution of a two-second-long constant 440Hz sine tone with the provided five-second-long impulse response file l960auto\_park.wav resulted in a total profiled convolution time of 484.68 seconds as reported by gprof profiling software, from which all timing data was extracted:

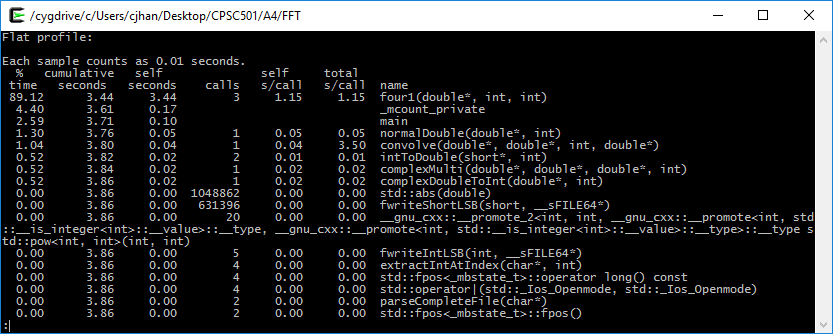


It should be noted that due to the limited power of my laptop, it was not feasible to convolve larger files, as the total convolution time was in excess of an hour. However, due to the small size of the files, timing values are subject to slight fluctuation.

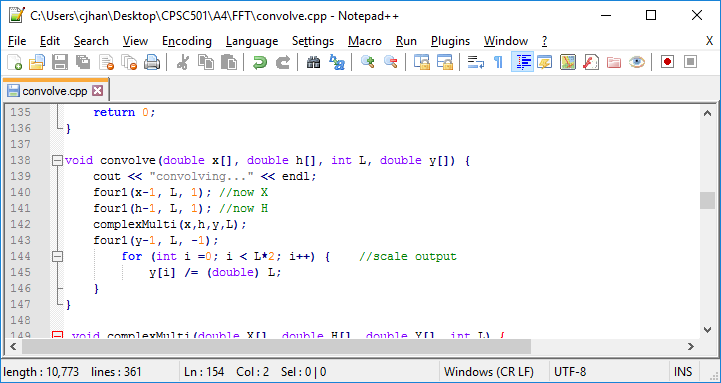
Algorithm-Based Optimization: The baseline program was optimized by changing the nature of the convolution from a time-domain input-side convolution to a frequency-domain convolution utilizing the provided Fast Fourier Transform function four1 to create a new convolve function:



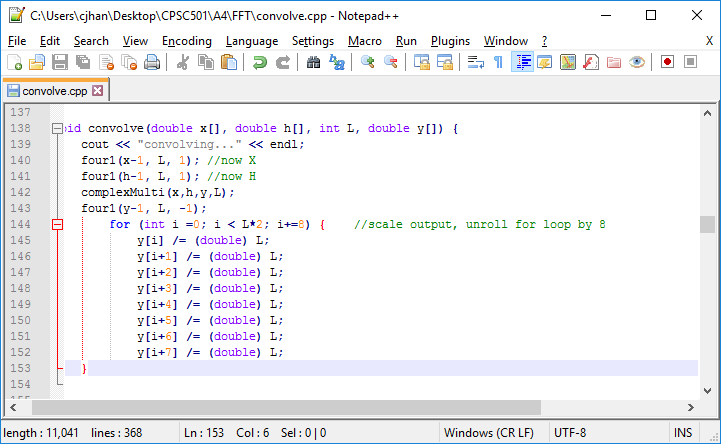
This change in algorithm results in a radical improvement to our program, generating a convolution of the same two files in only 3.86 seconds:



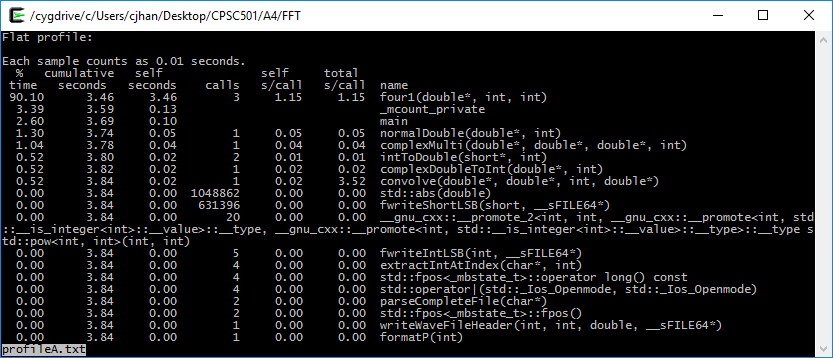
Hand Tuning 1: As we can see from the above profile, while four1 utilizes the majority of the execution time, there are several other potential candidates for improvements. We will look to them first before attempting any optimizations of four1, and Convolve seems a good candidate for a partial loop unrolling:



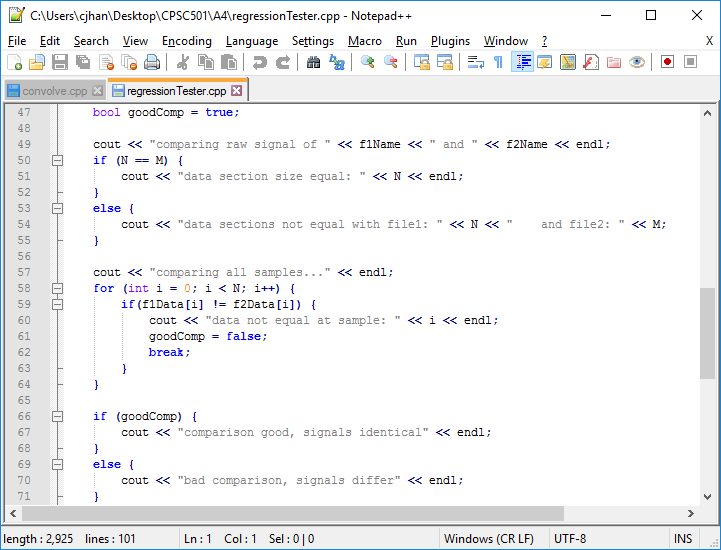
We can immediately detect that the brunt of convolve must come from the for loop which is scaling output from the IFFT call to four1. We can likely reduce the cost of convolve by unrolling the for loop. As we know that L\*2 is a power of 2 by construction, we can be sure that we can handle any smaller power of two cases inside the unrolled for loop. To enhance the effect, we will unroll a total of 8 values, leading to the following optimization:



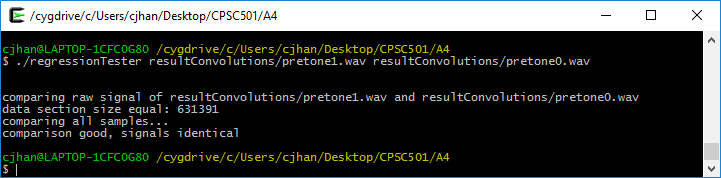
Unrolling this for loop results in a savings of 0.02 seconds and a reduction of the convolve function’s time percentage by 0.52% for an profiled runtime of 3.84 seconds.



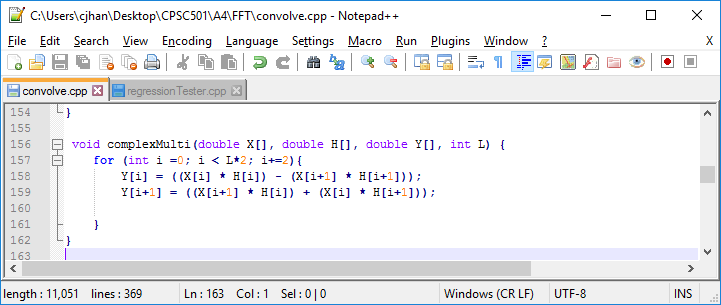
Regression testing was performed via a program called regressionTester, which will read in a pair of .wav files, parse out their data section lengths and then compare every sample in the data sections index by index. Here is the main comparison function:



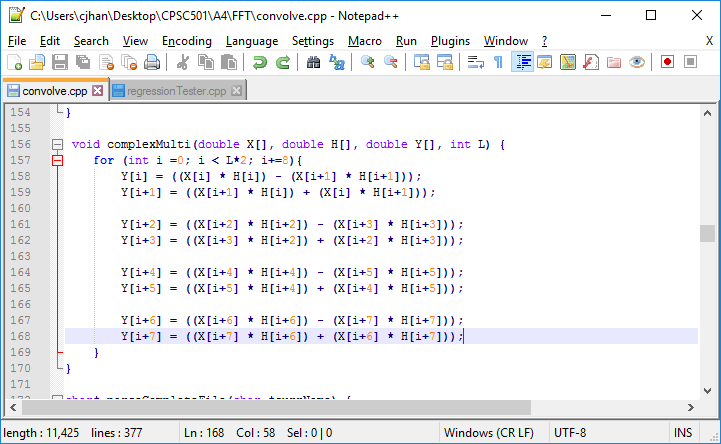
As we can see, this function should capture any regression in the raw signal data, and indicate at which position it lies relative to any convolution generated by older code. All convolved .wav files are stored separately and are named pretone#.wav, where # represents the optimization number. Running the regression test of our new pretone1.wav against the original algorithm optimized convolution pretone0.wav yields the following affirmative output:

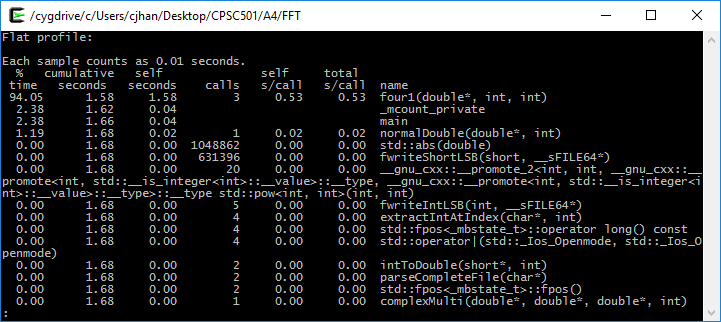


Hand Tuning 2: Examining our new profile, we see that some of the computation time is being used up by the complexMulti function. This function intakes two double arrays and provides the result of a point-by-point complex multiplication to a third double array. While it would be nice to replace the multiplications with repeated additions, we would be obligated to put in place a pair of nested for loops to achieve it, which would certainly increase our execution time. We can at least unroll the for loop for a slight optimization.

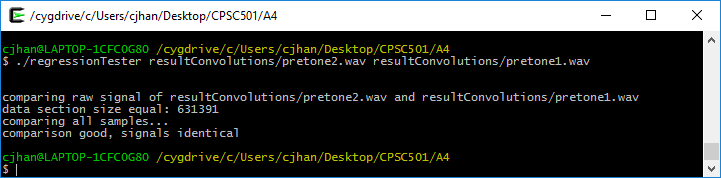


Once again, we know that the controlling factor of the for loop is L, which we have created by construction to be a power of two, we can maintain accurate results while unrolling the for loop to some smaller power of 2, in this case 8. This optimization results in the following code:

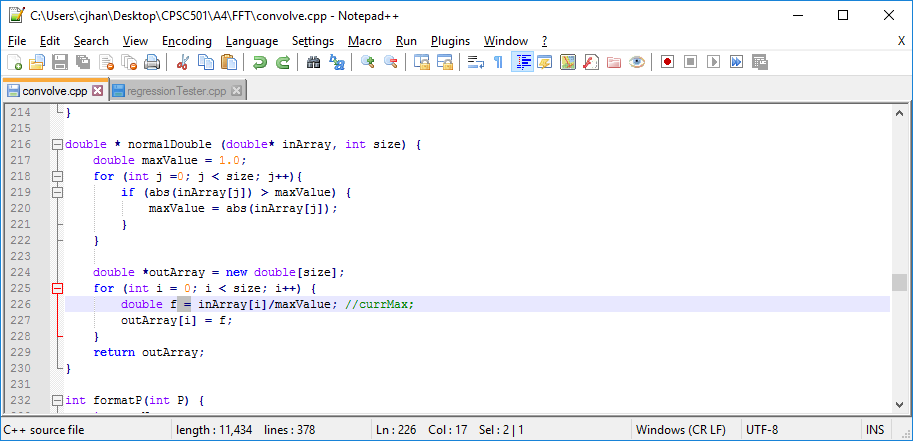




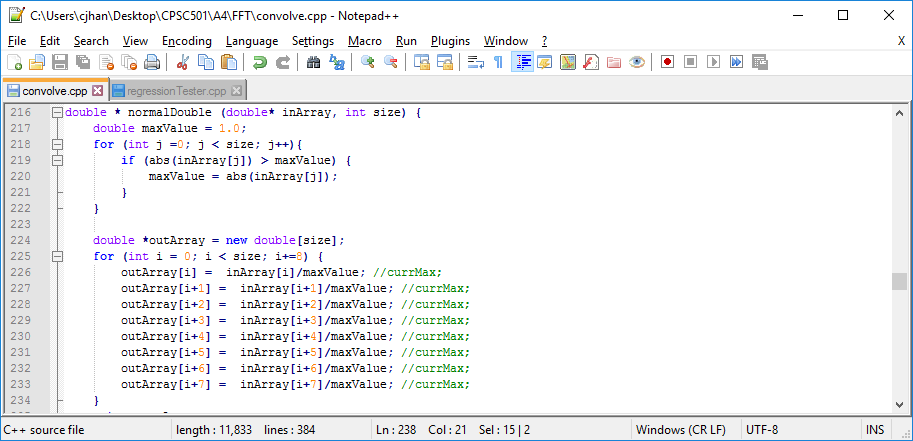
The new post-optimization profile demonstrates that we have reduced our total computation time to roughly 1.68 seconds, with a savings in complexMulti of 0.04 seconds. Testing regression against pretone1.wav is also successful, with the new tone being identical:



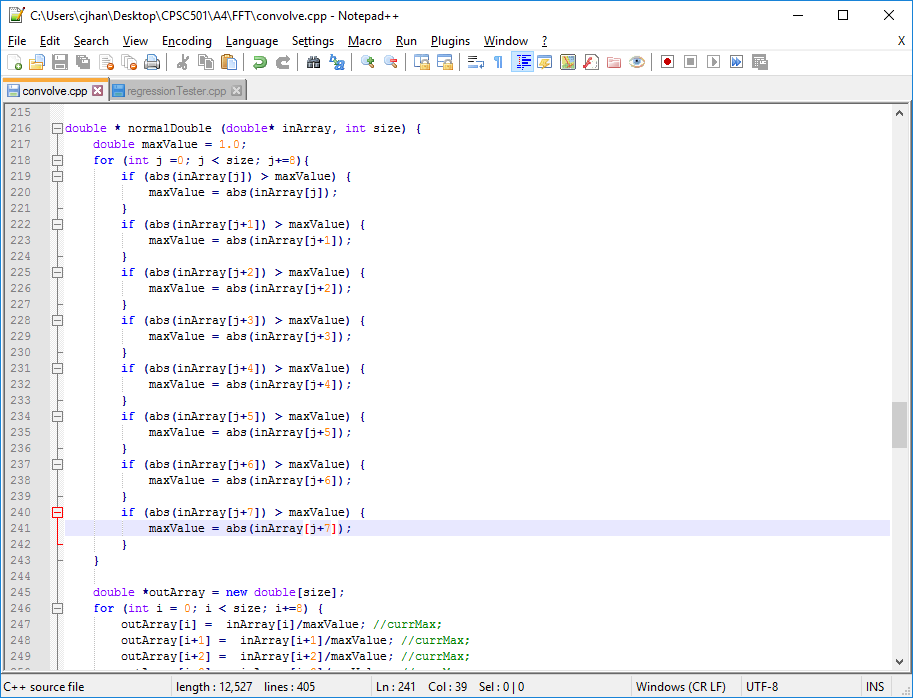
Hand Tuning 3 and 4: The function normalDouble is still taking up some measurable computation time. We see that it involves a pair of four loops, the first of which parses the input double array and locates its maximal value. The second then scales the same input array and returns the scaled values back in another array which is returned.



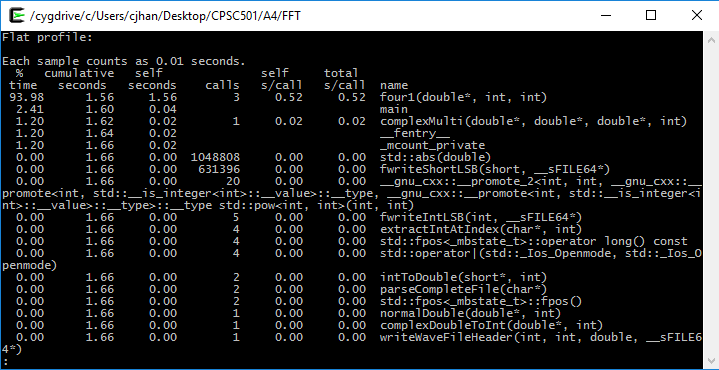
Once again, due to the large magnitude of L, which is passed in for the parameter size, any increase of efficiency of these for loops will result in good savings. The second for loop seems to be the more intuitive place to start. We begin by removing the useless local variable f and assigning the output array values directly. We then use our usual value of 8 to unroll the for loop and increment each iteration by 8, resulting in the following code:



While this seems to be a good improvement, we would do well to scale the other for loop by the same amount, to really get a worthwhile optimization out of this function. We will have to evaluate each case individually to prevent the introduction of a nested for loop:



Profiling after these two optimizations has reduced the cost of normalDouble by 0.02 seconds for a total time reduction to 1.66 seconds.



Once again the regression testing is evaluated against the tones generated by each optimization from the tone previous to them, pretone2:

