ECE 4822: Engineering Computation IV

Homework No. 6: Parallel DSP Filters Using OpenMP

By Shahzad Khan

The goal of this assignment is to implement both an FIR and IIR filter and parallelize them using OpenMP.

The Finite Impulse Response (FIR) filter is a digital filter whose output is a weighted sum of finite numbers of past inputs. Which means that it takes an input signal, and outputs will eventually settle to 0 given infinite time, hence the finite in the name. The equation for an FIR filter is the following:

$$y[n] = \sum_{i=0}^{N} b_i * x[n-i]$$

$x[n]$ = input signal

$y[n]$= output signal

N= filter order

b_i= coefficient of the filter

The advantage of FIR filters is that they are inherently stable which means no matter what coefficients are given, the output won't explode to infinity. The coefficients are the main part of FIR filters, because depending on what values are given the filter can change. There are a couple ways to calculate the coefficients for the filter. For example, to calculate the ideal impulse response for a low pass filter we can use the $sin(x)$ function:

$$h_d[n] = \frac{\sin(\omega_c n)}{\pi n}$$

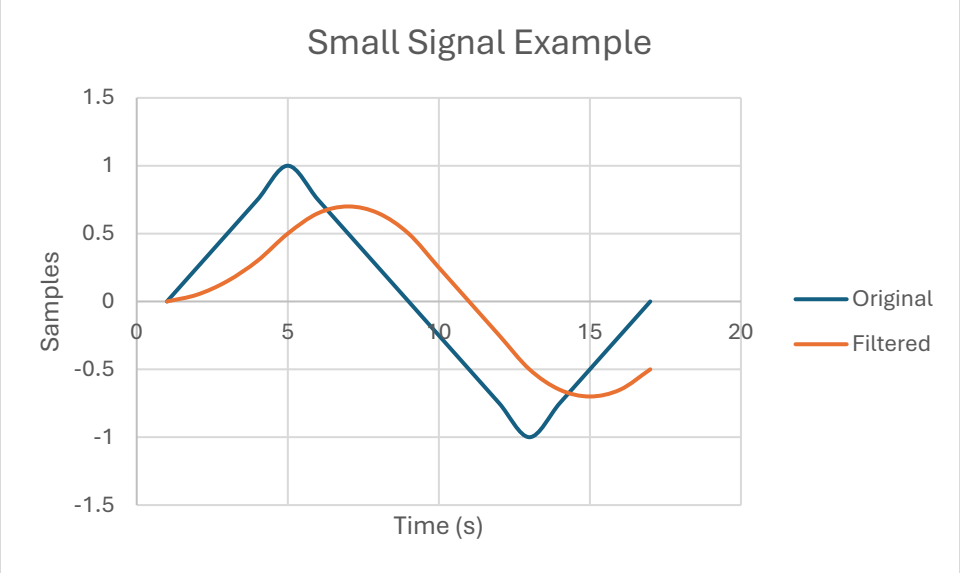
And to calculate the cut-off frequency we can use $\omega_c = 2\pi * \left(\frac{f_c}{f_s}\right)$ where f_s sampling frequency and f_c cutoff frequency in Hz. But a problem occurs because $\sin(x)$ function is infinite, we must convert it to a discrete time so we can use it in our program. The way we can do this is by centering our waves:

$$n = n - \frac{N-1}{2}$$

So, our low pass filter is because

$$h_d[n] = \sin\frac{\left(w_c\left(n - \frac{N-1}{2}\right)\right)}{\pi\left(n - \frac{N-1}{2}\right)}$$

Now, using this we can calculate our coefficients. For our case, we can take$\frac{1}{tabs}$] to calculate the coefficients for a low-pass filter. However, with this method you are unable to choose a cut-off frequency. By doing that, we can get a coefficient of [0.2, 0.2,0.2,0.2, 0.2]:



The coefficients act as a moving average where the input being a triangle wave, has the filter applied to smoothing the signal out.

Moving to how I parallelized the code. OpenMp makes this process simple on my end. The first problem we must deal with is a large file size, the solution to that is to use a circular buffer to feed the signal in blocks one at a time. In order to do this we first calculate the number of blocks:

```
 const int BLOCK_SIZE = 1024; // Number of samples per block

 int numBlocks = (samples.size() + BLOCK_SIZE - 1) / BLOCK_SIZE;
```

Then we can use OpenMp to parallelize each block, since the FIR filter doesn't use any feedback loop, we can do this:

```
// Parallel block processing
#pragma omp parallel for
 for (int b = 0; b < numBlocks; ++b) {
  int start = b * BLOCK_SIZE;
  int end = std::min(start + BLOCK_SIZE, (int)samples.size());

  // History buffer for this block
  std::vector<float> history(n, 0.0f);
  int circIndex = 0;

  // Initialize history from previous block if not first block
  if (b > 0) {
   int prevEnd = start;
   int copyStart = std::max(0, prevEnd - (n-1));
   int copyCount = prevEnd - copyStart;
   for (int i = 0; i < copyCount; ++i)
    history[i] = samples[copyStart + i];
   circIndex = copyCount % n;
  }

  // Process block
  for (int i = start; i < end; ++i) {
   output[i] = fir_filter_circ(samples[i], coef.data(), n, history.data(), &circIndex);
  }
 }
```

A history variable keeps track of the location in the block. Giving fully parallelized FIR filter. Inputting a file that is around 1GB the program finishes at a time of **5.123 (s):**

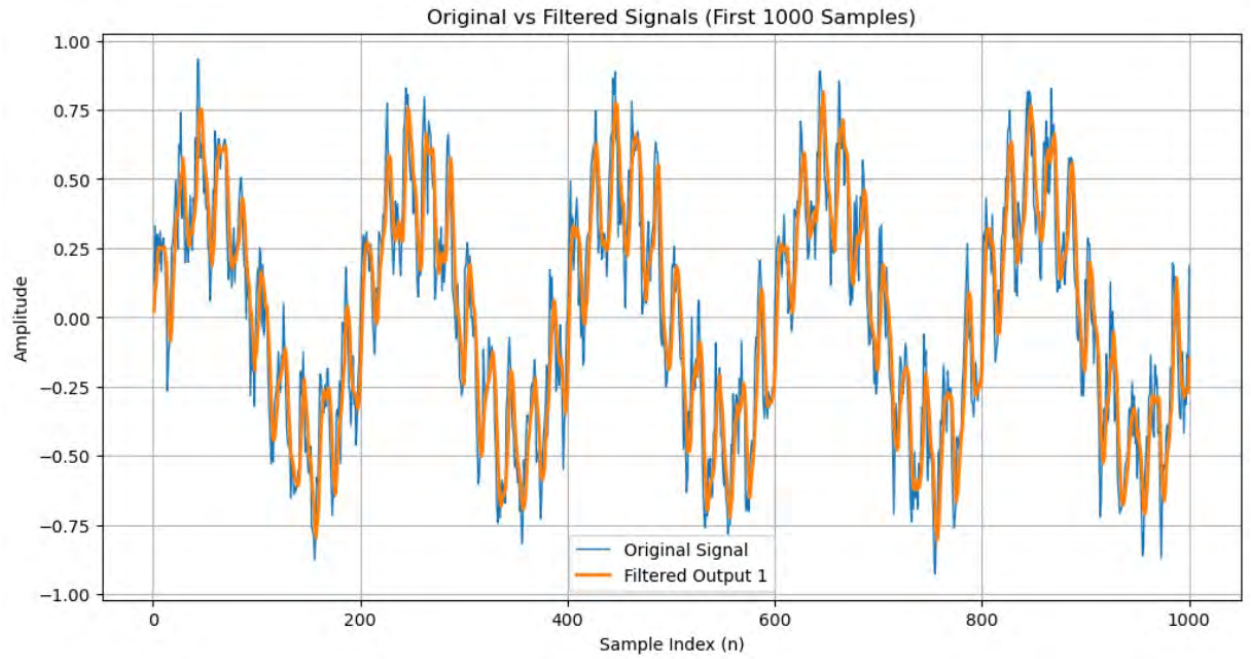


Figure 1

This is using our 0.2 moving average coefficient, but I also calculated some more for a cutoff frequency of $1000Hz$:

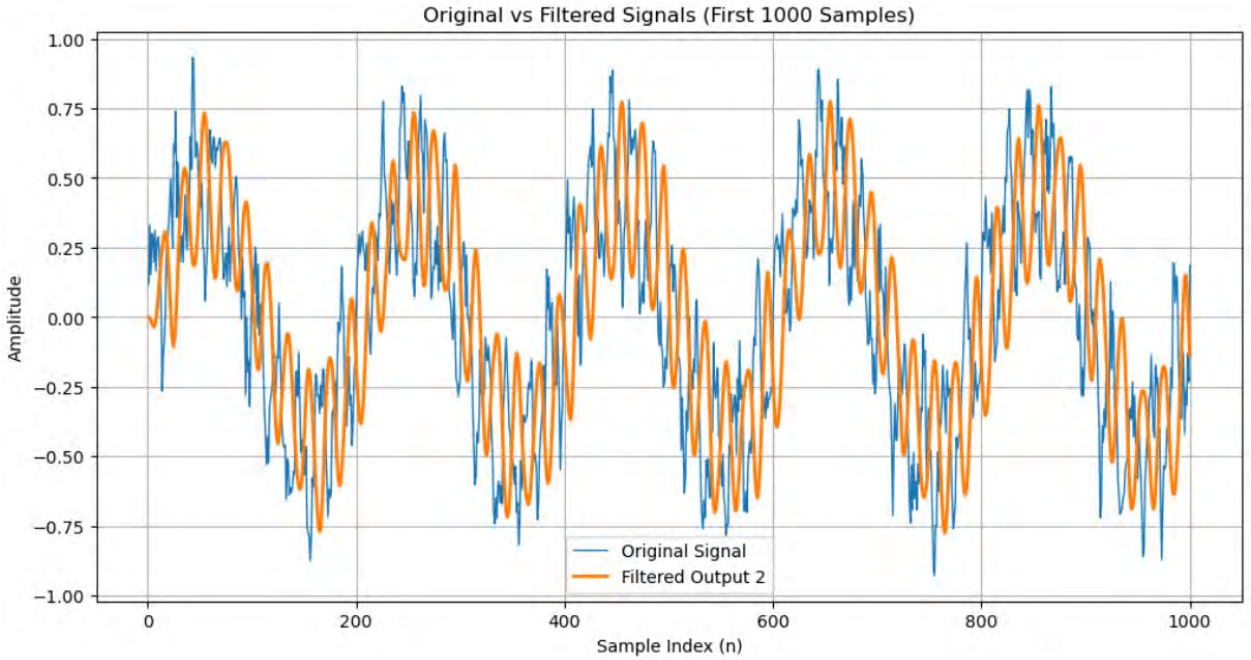


Figure 2

The graphs above show the filtered and unfiltered signal FIR with different coefficients, the filters acting as low pass filters, making sure the higher frequency get damped well passing the lower frequency.

1

The Infinite Impulse Response (IIR) is another digital filter that, as the name suggests, is similar to the FIR filter. However, there are a couple of key differences. For example, the FIRs response is finite and given infinite time will reach 0. The IIRs response is infinite and because of the recursive nature of the IIR, any impulse that goes through the signal never fades and effects the entire signal

$$y[n] = \sum_{i=0}^{P} b_i x[n-i] + \sum_{i=1}^{Q} a_i y[n-i]$$

P = Feed-Forward Filter

b_i = Feed-Forward Coefficient

Q = Feed-Back Filter

a_i= Feed-Back Coefficient

$x[n]$=input signal

$y[n]$ = output signal

As seen from the above equation, the IIR filter contains two parts. The first part being an FIR filter, the second being the recursive statement which gives the IIR filter its infinite response. The IIR is more efficient overall as a filter with the tradeoff of possible instability if incorrect poles are calculated.

\The poles must first be designed because the coefficients for the IIR cannot be calculated directly from the frequency response. The equation to calculate the poles for a filter is given as:

$$x(t) = e^{\sigma} * e^{j\omega t}$$

In a stable filter sigma is negative, allowing for e to have a decay effect. When sigma is positive it has a growth effect, expanding $x(t)$ to infinity. Which is an unstable filter.

$x(t)$ must be converted to discrete time, giving the following equation:

$$x[n] = r^n * e^{jn\theta}$$

Now we see that r^n is our growth or decay coefficient and if n is greater than 1, the function once again explodes as a result of the aforementioned growth effect of a decay which makes our filter stable.

We can use this equation to find the poles of a first order low pass filter.

$$z = 1 + \frac{1 + \frac{sT}{2}}{1 - \frac{sT}{2}}$$

Transfer Function^

$$H(z) = \frac{b_0}{1 - a_1 z^{-1}}$$

When we solve for the denominator = 0 so we can find the poles

$$z = a_1$$

Then we can find the poles using the cutoff and sampling frequency

$$a_1 = \frac{1 - \alpha}{1 + \alpha}, \quad \alpha = \frac{\omega_c}{f_s 2}$$

Calculating a cutoff of $1000Hz$ with a $f_s = 10{,}000Hz$

$$2 * \pi * \frac{1000}{10000} = 0.6283 \frac{rad}{sample}$$

Plugging those into the transfer function

$$z = \frac{1 + \frac{sT}{2}}{1 - \frac{sT}{2}} = \frac{1 - 0.31415}{1 + 0.31415} = \frac{0.68585}{1.31415} = 0.522$$

The pole would be $a_1 = 1 - 0.522$

$$b_0 + b_1 = 1 - a_1 = 1 - 0.522 = 0.478$$

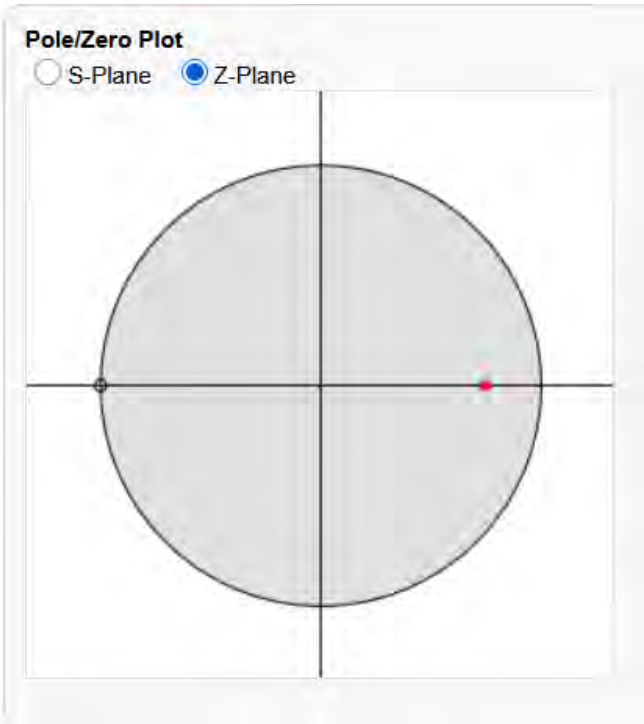


Figure 3

The coefficient for the lowpass filter with a cutoff of $1000Hz$ would be

$$D = [1, -0.522]$$

$$N = [0.239, 0.239]$$

The parallelization of the IIR filter is a little more complex, since there is a feedback loop integrated. We must make sure that that we can use history from our previous iteration to us when we do our current calculation.

```
// Precompute histories for each block
std::vector<std::vector<float>> xHistories(numBlocks, std::vector<float>(bCoef.size(), 0.0f));
std::vector<std::vector<float>> yHistories(numBlocks, std::vector<float>(aCoef.size()-1, 0.0f));



// Copy last samples from previous block for history initialization
for(int b = 1; b < numBlocks; ++b){
 int prevEnd = b*BLOCK_SIZE;
 int copyCountX = std::min((int)bCoef.size()-1, prevEnd);
 int copyCountY = std::min((int)aCoef.size()-1, prevEnd);

 for(int i = 0; i < copyCountX; ++i)
  xHistories[b][i] = samples[prevEnd - copyCountX + i];

 for(int i = 0; i < copyCountY; ++i)
  yHistories[b][i] = output[prevEnd - copyCountY + i]; // initially zero for first iteration
}
```

Then the code can be parallelized with our precomputed history blocks

```
// Process each block in parallel
#pragma omp parallel for
for(int b = 0; b < numBlocks; ++b){
int start = b * BLOCK_SIZE;
int end = std::min(start + BLOCK_SIZE, (int)samples.size());

auto& xHist = xHistories[b];
auto& yHist = yHistories[b];

for(int i = start; i < end; ++i){
output[i] = iir_filter(samples[i], bCoef, aCoef, xHist, yHist);
}
}
```

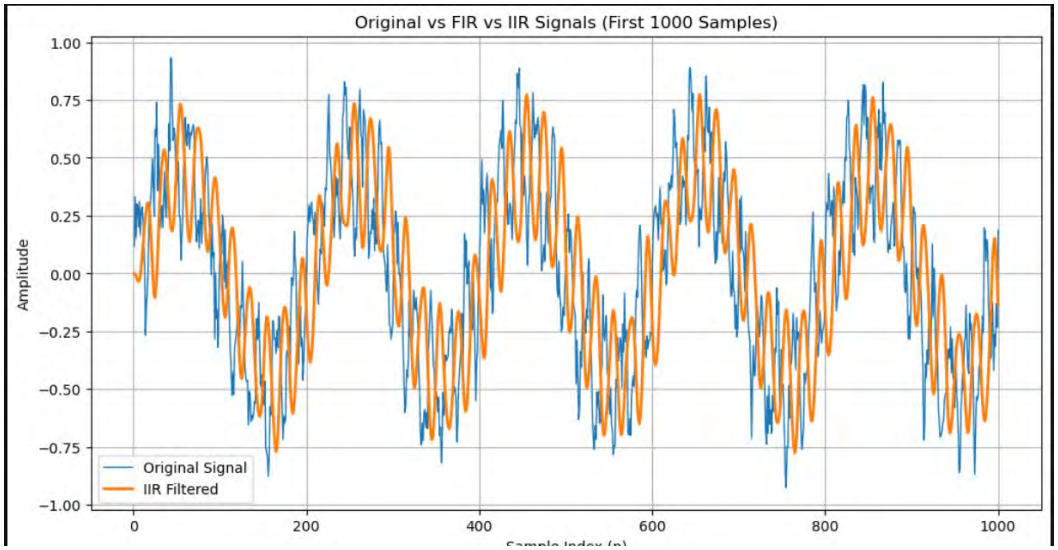


Figure 4

Above the graph shows the output of the IIR filter compared to noise signal, and showing that it does filter out the higher frequency and spikes well.

The program runs with a 1GB file in **4.780s.**

In conclusion, through this homework, I was able to use OpenMP to parallelize both an FIR and IIR filter. Through this I was able to gain a better understanding of both filters , the challenges with parallelizing functions that utilize a feedback loop, along side how digital filters work and the challenges with their implementation such as taking into account how the data will be fed to the function, My most important take away is how filters tie back to the rest of my classes and it allowed for me to get a better understanding of basic analog signal filtering going on in my classes currently. how we can convert then to discrete time , an example is the stability of a IIR filter, and how every analog filter we are using is an IIR filter and the difference between the s and z planes.