Code for all parts of the lab

main.cpp:

```
#define _USE_MATH_DEFINES
#include <iostream>
#include <string>
#include <cmath>
#include "../Shared/dspf.hpp"

struct complex {
          float Re, Im;
          complex operator= (float b) {
                    Re = b; Im = 0;
                    return *this;
          }
          complex operator+= (complex b) {
                    Re += b.Re; Im += b.Im;
                    return *this;
          }
          complex operator*=(complex b) {
                    float x = Re, y = Im;
                    Re = (x * b.Re) - (y * b.Im);
                    Im = (x * b.Im) + (b.Re * y);
                    return *this;
          }
};
complex operator* (complex a, complex b) {
          complex c = { 0, 0 };
          c.Re = (a.Re * b.Re) - (a.Im * b.Im);
          c.Im = (a.Re * b.Im) + (b.Re * a.Im);
          return c;
}
complex operator* (float a, complex b) {
          complex c = b;
          c.Re *= a; c.Im *= a;
          return c;
}
complex operator+ (complex a, complex b) {
          complex c = a;
          return c += b;
}
complex operator- (complex a, complex b) {
          complex c = a;
          return c += (-1.0f * b);
}
complex polar(double mag, double angle) {
          complex c = { 0, 0 };
          c.Re = (float)(mag * cos(angle));
          c.Im = (float)(mag * sin(angle));
          return c;
}

void tune(DSPFile& in, DSPFile& out, const float& station) {
          const float FS = 8.0f;
          const float FC = 94.8f;
          complex x[IOBUFFSIZE];
          unsigned long i = 0; // On the order of 75 million (int only goes up to about 4 million)

          /**
           *  Attempts to optimize the loop
           */
          // Attempt 1 - use fopen_s instead of std::fstream (see dspf.cpp)
          // Net Savings per loop: Uncertain

          // Attempt 2 - Move some multiplication out of the loop
          float prescale = (float)(-2 * M_PI * (station - FC) / FS); // Move some operations outside of the loop
          // Net Savings per loop: 2 mult ops, 1 divide op, 1 add op

          // Attempt 3 - Use periodicity of complex exponentials to precalculate values from sin/cos in polar()
          float precision = 0.0001f;
          std::vector<complex> periodicity;
          unsigned int n = 0; // n turns out to be on the order of 16,620
          periodicity.push_back(polar(1, prescale * n++));
          while (true) {
                    periodicity.push_back(polar(1, prescale * n));
                    complex diff = periodicity[n++] - periodicity[0];
                    if (abs(diff.Re) < precision && abs(diff.Im) < precision) {
                              break;
                    }
          }
          // Net Savings per loop: 3 mult ops, 2 func calls (sin/cos ops, probably table-lookup based), 2 assign ops

          /**
          *  All the above efforts don't seem to have made a significant impact on time to process 600MB file
          */

          // Attempt 4 - change IOBUFFSIZE from 1024 -> 32768 (ok, we have impact now)
          // Result: Processing time reduced from ~10 minutes to ~1 minute

          // Attempt 5 - Revert back to std::fstream since it did not improve after attempt 1
          // Result: We're back at the old speed all the sudden....wth?!
```

```
		/**
		 *  End optimization attemps
		 */

		int xlen = in.read_n((float*)x, 100 * 2) / 2; // Skip the first garbage 100 samples
		if (xlen == 100) {
			xlen = in.read_n((float*)x, IOBUFFSIZE * 2) / 2;
			while (xlen > 0) {
				for (int j = 0; j < xlen; ++j) {
					// Performance breakdown:
					// 1 mod op, 2 access ops, 4 mult ops, 2 add ops, 1 func call, 2 float alloc, 2 assign op
					x[j] *= periodicity[i++ % n]; // polar(1, 2pi(ft-fc)/fs * i++)
				}
				out.write_d((float*)x, xlen * 2);
				xlen = in.read_n((float*)x, IOBUFFSIZE * 2) / 2;
			}
		} else {
			std::cout << "Invalid input signal file..." << std::endl;
		}
}

void resample(const dsig& hsig, const int& U, const int& D, DSPFile& in, DSPFile& out) {
		// Determine bounds on computed arrays
		const int L = (int)hsig.size();

		// Adjust the header on the output file
		out.Header = in.Header;
		out.Header.dim0 = (out.Header.dim0 * U) / D;
		out.Header.dim1 = (out.Header.dim1 * U) / D;
		out.write_h();

		// It would be really nice if Dr. Gunther explained in his slides what the heck this is...
		int M = L / U + ((L % U) > 0);
		int N = M * U; // Padded impulse response length

		int d = 0, k = 0;
		complex* x = new complex[L];
		const float* h = hsig.data();
		complex
			xbuff[IOBUFFSIZE],
			ybuff[IOBUFFSIZE];

		// Zero out circular buffer to clear garbage
		for (int i = 0; i < M; ++i) { x[i] = 0; }

		//x[i] = in.read_1();
		int xlen = in.read_n((float*)xbuff, IOBUFFSIZE * 2) / 2;
		int ylen = 0;
		while (xlen > 0) {
			for (int i = 0; i < xlen; ++i) {
				k = (k + M - 1) % M;
				x[k] = xbuff[i];

				if (d == 0) { // Downsampling discards D - 1 values
					for (int j = 0; j < U; ++j) {
						complex y = { 0, 0 }; int m = 0, n = 0;
						// Upsampling creates 0 every U elements of x (skipping over h because
convolution is associative)
						for (; n < M; ++n, m += U) {
							y += h[m + j] * x[(n + k) % M];
						}
						ybuff[ylen++] = y;
						if (ylen == IOBUFFSIZE) {
							out.write_d((float*)ybuff, ylen * 2);
							ylen = 0;
						}
					}
					d = D - 1;
				} else { --d; }
			}

			xlen = in.read_n((float*)xbuff, IOBUFFSIZE * 2) / 2;
		}
		if (ylen > 0) {
			out.write_d((float*)ybuff, ylen * 2);
			ylen = 0;
		}

		delete[] x;
}

int main() {
		float station = 96.7f;
		std::string
			f_h1 = "output\\h1.bin",
			f_h2 = "output\\h2.bin",
			f_h3 = "output\\h3.bin",
			f_h4 = "output\\h4.bin",
			f_h5 = "output\\h7.bin",
			f_radio = "output\\freq94_8_bw_4.bin",
			f_y0 = "output\\y0.bin",
			f_y1 = "output\\y1.bin",
			f_y2 = "output\\y2.bin",
			f_y3 = "output\\y3.bin",
			f_y4 = "output\\y4.bin",
```

```
                f_x = "output\\x.bin",
                f_r1 = "output\\r1.bin",
                f_r2 = "output\\r2.bin",
                f_r3 = "output\\r3.bin";
        DSPFile
                fin(DSP::Mode::Read | DSP::Mode::NoHeader),
                fout(DSP::Mode::Write | DSP::Mode::NoHeader),
                fh1(f_h1, DSP::Mode::Read),
                fh2(f_h2, DSP::Mode::Read),
                fh3(f_h3, DSP::Mode::Read),
                fh4(f_h4, DSP::Mode::Read),
                fh5(f_h5, DSP::Mode::Read);
        dsig
                h1 = fh1.read_all(),
                h2 = fh2.read_all(),
                h3 = fh3.read_all(),
                h4 = fh4.read_all(),
                h5 = fh5.read_all();

        fout.close();
        fin.open(f_radio); fout.open(f_y0);
        tune(fin, fout, station);

        fout.close();
        fin.open(f_y0); fout.open(f_y1);
        resample(h1, 1, 2, fin, fout);

        fout.close();
        fin.open(f_y1); fout.open(f_y2);
        resample(h2, 1, 2, fin, fout);

        fout.close();
        fin.open(f_y2); fout.open(f_y3);
        resample(h3, 1, 2, fin, fout);

        fout.close();
        fin.open(f_y3); fout.open(f_y4);
        resample(h4, 1, 2, fin, fout);

        fout.mode(DSP::Mode::Write);
        fin.open(f_x); fout.open(f_r1);
        fin.Header = {1, 1, 4789058, 500000, 0 }; // Hard coded because the values are known and running out of time
        resample(h5, 1, 2, fin, fout);

        fin.mode(DSP::Mode::Read); fout.close();
        fin.open(f_r1); fout.open(f_r2);
        resample(h5, 3, 5, fin, fout);

        fout.close();
        fin.open(f_r2); fout.open(f_r3);
        resample(h5, 1, 5, fin, fout);

        system("pause");
        return 0;
}
```

main.m

```
clear all;

h = [ ...
    {firpm(4, [0, 1/80, 1/2 - 1/80, 1/2]*2, [1, 1, 0, 0]), 'h1', 8*10^6}; ...
    {firpm(4, [0, 1/40, 1/2 - 1/40, 1/2]*2, [1, 1, 0, 0]), 'h2', 4*10^6}; ...
    {firpm(6, [0, 1/20, 1/2 - 1/20, 1/2]*2, [1, 1, 0, 0]), 'h3', 2*10^6}; ...
    {firpm(8, [0, 1/10, 1/2 - 1/10, 1/2]*2, [1, 1, 0, 0]), 'h4', 1*10^6}; ...
    {firpm(6, [0, 3/50, 1/2 - 3/50, 1/2]*2, [1, 1, 0, 0]), 'h5', 500*10^3}; ...
    {firpm(6, [0, 3/25, 1/2 - 3/25, 1/2]*2, [1, 1, 0, 0]), 'h6', 250*10^3}; ...
    {firpm(6, [0, 3/15, 1/2 - 3/15, 1/2]*2, [1, 1, 0, 0]), 'h7', 150*10^3}; ...
];

h{5, 1} = lpf(2, 256); % I gave up trying to make firpm work
h{6, 1} = lpf(5, 256); % I gave up trying to make firpm work
h{7, 1} = lpf(5, 256); % I gave up trying to make firpm work

% Write header binary files
for i = 1:7
    fid = fopen(sprintf('output\\%s.bin', h{i, 2}), 'wb');
    fwrite(fid, [1 1 length(h{i, 1}) 1 0], 'int');
    fwrite(fid, h{i, 1}, 'float');
    fclose(fid);
end

% Plot h1-h4
figure(1);
for i = 1:4
    subplot(4, 1, i);
    [w, F, theta, r, db] = getFFT(h{i, 1}, 10, h{i, 3});
    plot(w, db);
    ylabel('Magnitude (dB)');
    xlabel('Frequency (Hz)');
    title(sprintf('LPF %i', i));
end

sig = [ ... % file, f, f_c
```

```
    {'freq94_8_bw_4', 0, 0, 'Original Signal', 8*10^6, 0, 0, 0, 0, 0}; ...
    {'y0', 0, 0, 'After Fruency Shift', 8*10^6, 0, 0, 0, 0, 0}; ...
    {'y1', 0, 0, 'After LPF 1', 4*10^6, 0, 0, 0, 0, 0}; ...
    {'y2', 0, 0, 'After LPF 2', 2*10^6, 0, 0, 0, 0, 0}; ...
    {'y3', 0, 0, 'After LPF 3', 1*10^6, 0, 0, 0, 0, 0}; ...
    {'y4', 0, 0, 'After LPF 4', 500*10^3, 0, 0, 0, 0, 0}; ...
];

% Read binary files for each stage
for file = 1:6
    fid = fopen(sprintf('output\\%s.bin', sig{file, 1}), 'rb');
    sig{file, 2} = fread(fid, inf, 'float');
    fclose(fid);
    sig{file, 3} = reshape(sig{file, 2}, [2, size(sig{file, 2}, 1)/2]).';
    sig{file, 3} = complex(sig{file, 3}(:,1), sig{file, 3}(:,2));

    [w, F, theta, r, db] = getFFT(sig{file, 3}, 21, sig{file, 5});
    sig{file, 6} = w;
    sig{file, 7} = F;
    sig{file, 8} = theta;
    sig{file, 9} = r;
    sig{file, 10} = db;
end

%Plot the FFT
figure(2);
for file = 1:6
    subplot(6, 1, file);
    plot(sig{file, 6}, sig{file, 10});
    ylabel('Magnitude (dB)');
    xlabel('Frequency (Hz)');
    title(sig{file, 4});
end

% Recover x[n]
d = firpm(66, [0, 0.2, 0.25, 0.5]/0.5, [0, 1, 0, 0], 'differentiator');
f = sig{6, 3};
u1 = real(f);
v1 = imag(f);

v2 = conv(v1, d);
u2 = conv(u1, d);

delay = zeros(1, 67);
delay(34) = 1;

u1 = conv(u1, delay);
v1 = conv(v1, delay);

f = ((u1 .* v2) - (v1 .* u2)) ./ sqrt(abs(u1).^2 + abs(v1).^2);
f(isnan(f)) = 0;
[w, F, theta, r, db] = getFFT(f, 21, 500*10^3);

% Plot the spectrum of x[n]
figure(3);
plot(w, db);
ylabel('Magnitude (dB)');
xlabel('Frequency (Hz)');
title('Recovered Radio Signal');

% Plot h5-h7
figure(4);
for i = 5:7
    [w, F, theta, r, db] = getFFT(h{i, 1}, 10, h{i, 3});
    subplot(3, 1, i - 4);
    plot(w, db);
    ylabel('Magnitude (dB)');
    xlabel('Frequency (Hz)');
    title(sprintf('LPF %i', i));
end

Write binary file for recovered x[n]
fid = fopen('output\\x.bin', 'wb');
fwrite(fid, f, 'float');
fclose(fid);

% Read binary files for each stage
audio = [ ... % file, x, fs
    {'r1.bin', 0, 0}; ...
    {'r2.bin', 0, 0}; ...
    {'r3.bin', 0, 0}; ...
];
figure(5);
for file = 1:3
    [audio{file, 2}, audio{file, 3}] = bin2audio(audio{file, 1});
    [w, F, theta, r, db] = getFFT(audio{file, 2}, 21, audio{file, 3});
    subplot(3, 1, file);
    plot(w, db);
    ylabel('Magnitude (dB)');
    xlabel('Frequency (Hz)');
    title(sprintf('After LPF %i', file));
end

% Play the bad news
soundsc(audio{3,2}, audio{3,3});
```

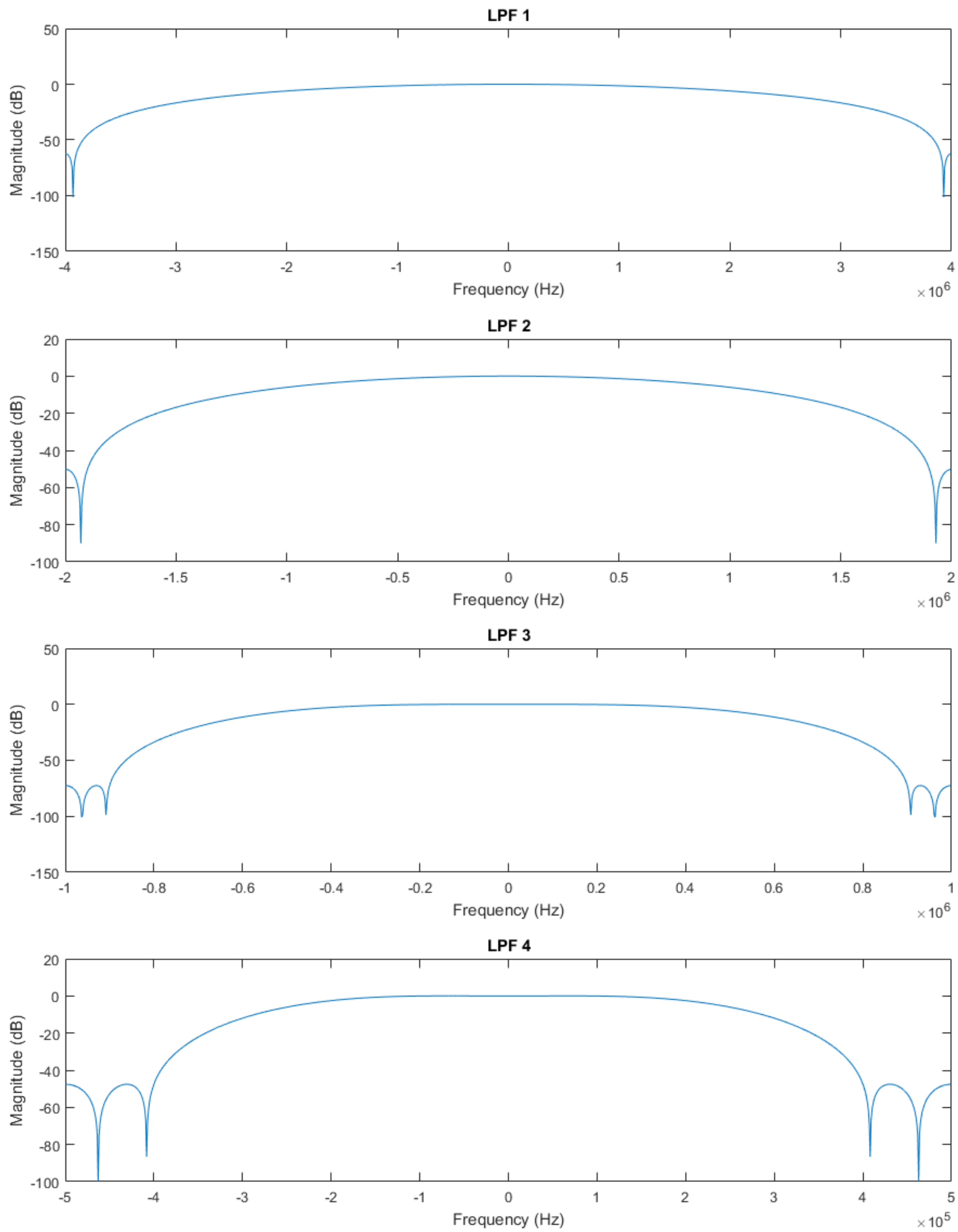
lpf.m

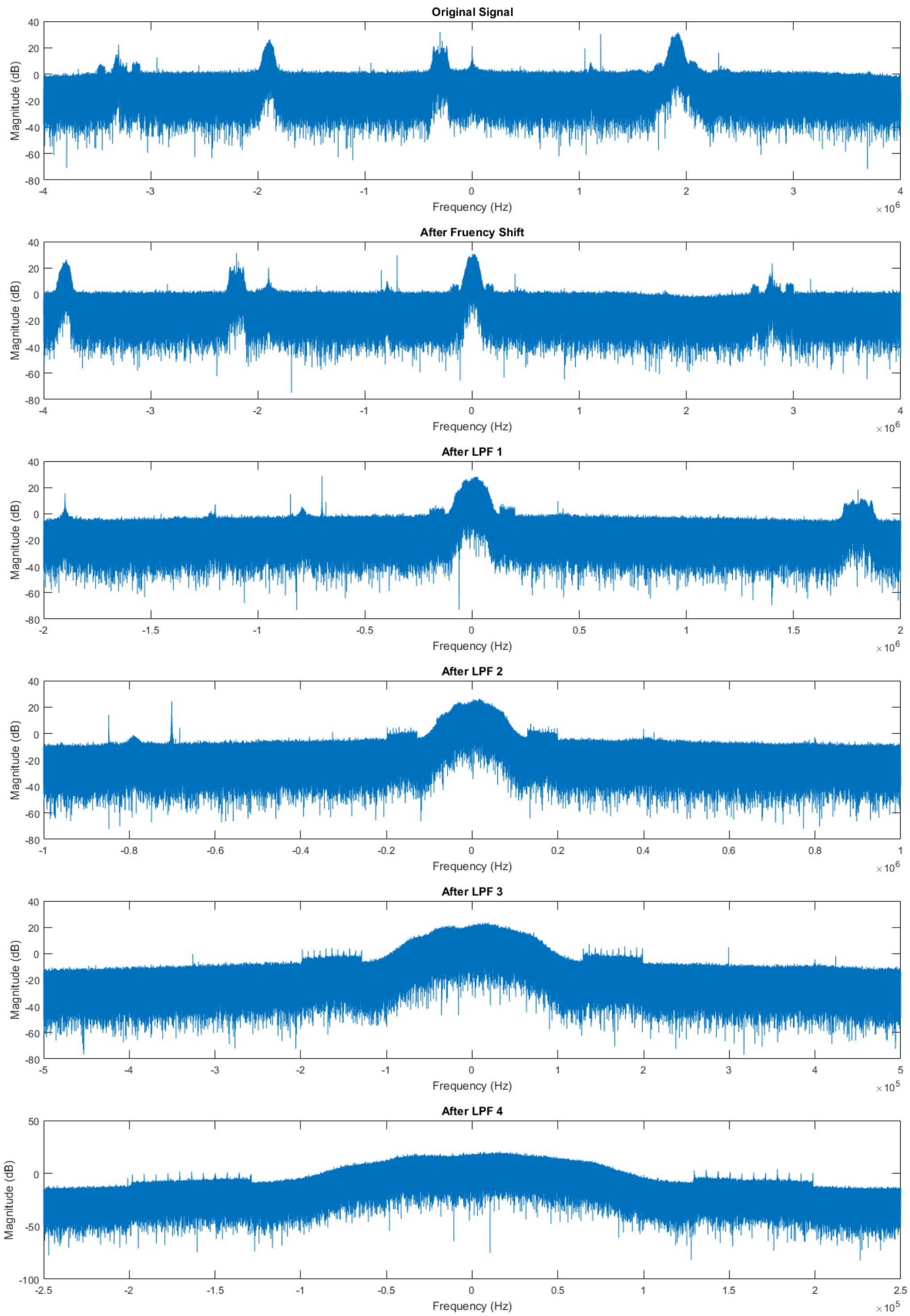
```
function [h] = lpf(D, L)
    fpass = 0.97/(2*D);
    fstop = 1.03/(2*D);
    f1 = (fstop + fpass)/2;
    f2 = (fstop - fpass)/2;
    n = (-L:L).';
    h = (1/D)*sinc(2*f1*n).*sinc(2*f2*n);
end
```

getFFT.m

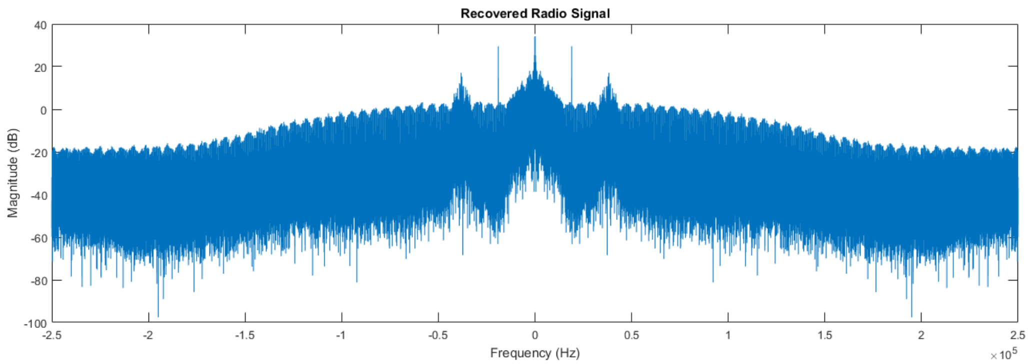
```
function [freq, F, phase, mag, db] = getFFT(f, size, scale)
    if nargin == 2
        scale = 1;
    end
    NFFT = 2 ^ size;
    freq = (((0:NFFT-1)/NFFT) - 0.5) * scale;
    F = fftshift(fft(f, NFFT));
    phase = angle(F);
    mag = abs(F);
    db = 20*log10(mag);
end
```

Output for Part 1

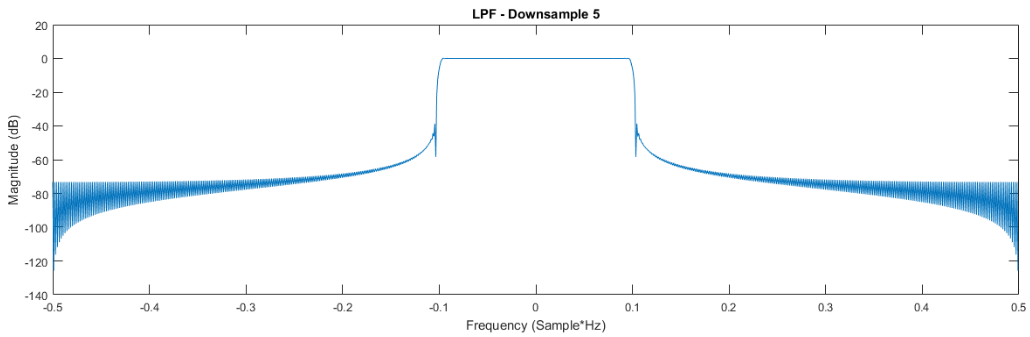


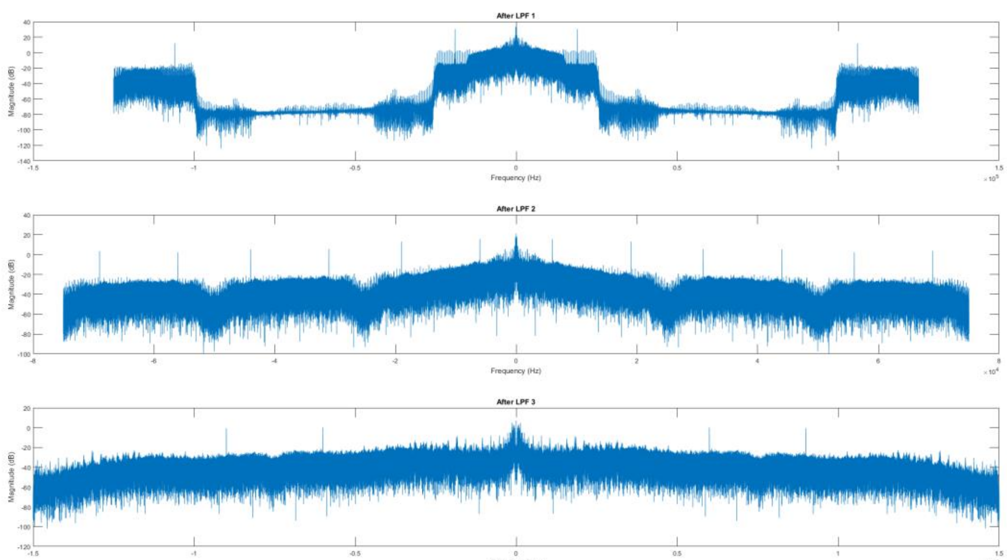

Original Signal
Magnitude (dB)
Frequency (Hz)
$\times 10^6$
After Fruency Shift
Magnitude (dB)
Frequency (Hz)
$\times 10^6$
After LPF 1
Magnitude (dB)
Frequency (Hz)
$\times 10^6$
After LPF 2
Magnitude (dB)
Frequency (Hz)
$\times 10^6$
After LPF 3
Magnitude (dB)
Frequency (Hz)
$\times 10^5$
After LPF 4
Magnitude (dB)
Frequency (Hz)
$\times 10^5$

Output for Part 2



Output for Part 3





As for the filter used in part 3: I had a hard time getting firpm to create a filter according to the needs of the lab, so I borrowed from code from lab 4 to create a filter for downsampling by 5 (see lpf.m). For the first downsample by 2 since the actual content of interest is significantly lower frequency than half the spectrum there is no harm in filtering out additional frequency. So, for convenience, I used the same filter for all three steps.