C code:

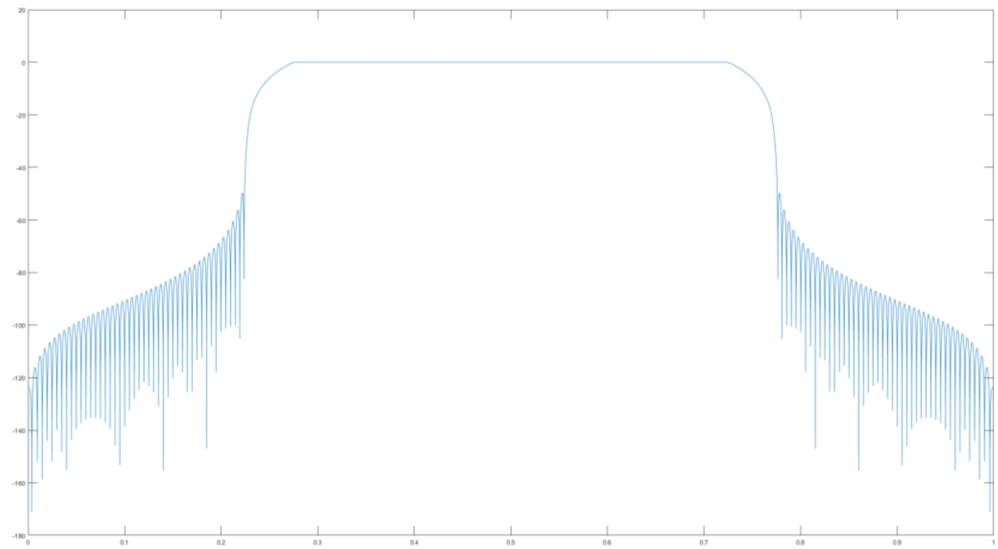
```
#include <iostream>
#include <string>
#include "../Shared/dspf.hpp"

void resample(dsig hsig, int U, int D, DSPFile& in, DSPFile& out) {
        // Determine bounds on computed arrays
        const int L = (int)hsig.size();
        const int IOBUFFSIZE = 1024;

        // 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;
        float* x = new float[L];
        float* h = hsig.data();
        float
                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(xbuff, IOBUFFSIZE);
        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) {
                                        float 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(ybuff, ylen);
                                                ylen = 0;
                                        }
                                }
                                d = D - 1;
                        } else { --d; }
                }

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

        delete[] x;
        //delete[] h;
}

int main() {
        int argc = 6;
        const char* argv[6] = { "Lab 4.exe",
                "output\\lpf_U2_D1.bin",
                "output\\galway11_mono_45sec.bin",
                "output\\galway11_U2_D1.bin",
                "2", "1" };

//int main(int argc, char** argv) {
        int U, D;
        std::string h, in, out;
```

```
        if (argc != 6) {
                std::cout << "Invalid Args" << std::endl;
                system("pause");
                return -1;
        }

        h = std::string(argv[1]);
        in = std::string(argv[2]);
        out = std::string(argv[3]);
        U = atoi(argv[4]);
        D = atoi(argv[5]);

        DSPFile lpf(h), fin(in), fout(out, DSP::Mode::Write);
        resample(lpf.read_all(), U, D, fin, fout);

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

Matlab:

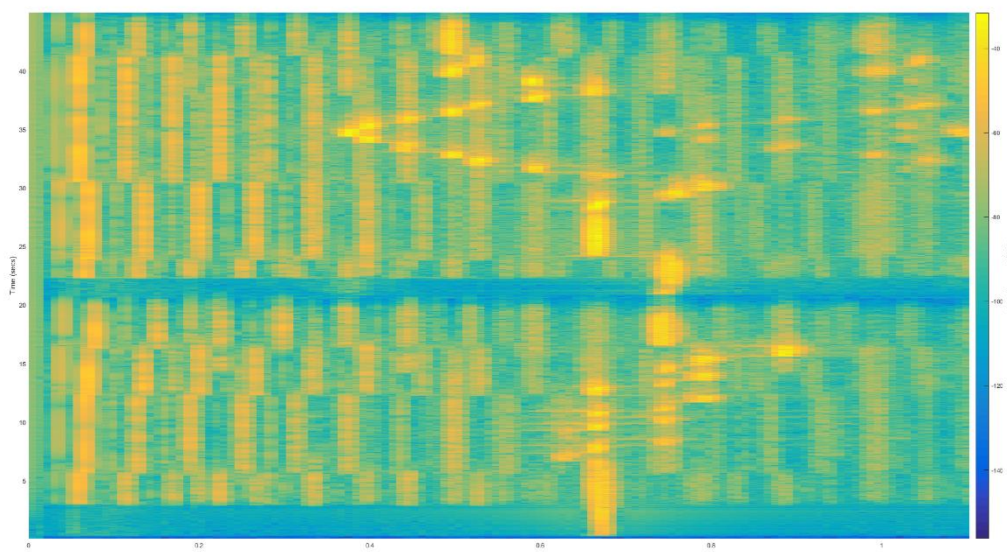
```
clear all;

% Parse the audio
[x, fs] = audio2bin('galway11_mono_45sec.wav');
h = lpf_resamp(2, 1);
plotFFT(h, 10);
plot_spectrogram(x, 10, fs);
soundsc(x(1:10*fs), fs);
[x_rs, fs_rs] = bin2audio('galway11_U2_D1.bin');
plot_spectrogram(x_rs, 8, fs_rs);


function [h] = lpf_resamp(U, D)
    N = max([U D]);
    fpass = 0.9/(2*N);
    fstop = 1.1/(2*N);
    f1 = (fstop + fpass)/2;
    f2 = (fstop - fpass)/2;
    L = 100;
    n = (-L:L).';
    h = (1/N)*sinc(2*f1*n).*sinc(2*f2*n);

    % Write out the filter file
    file = sprintf('output\\lpf_U%d_D%d.bin', U, D);
    fid = fopen(file, 'wb');
    fwrite(fid, [1 1 length(h) 1 0], 'int');
    fwrite(fid, h, 'float');
    fclose(fid);
end
function [] = plot_spectrogram(f, size, Fs)
    NFFT = 2^size
    spectrogram(f, hamming(NFFT), round(0.8*NFFT), NFFT, Fs);
end
function [] = plotFFT(f, size)
    NFFT = 2 ^ size;
    freq = (0:NFFT-1)/NFFT;
    plot(freq, 20*log10(abs(fftshift(fft(f, NFFT)))));
end
```

So I've had a problem getting up-sampling to behave properly. I've compared with other students who have got this lab working, as well as Dr. Gunther's code (it's efficient, but not easily understandable or self-documenting, and he doesn't cover it in his slides, and we don't talk about it in class, </vent>) and can find no disparities. My down-sampling appears to work properly.

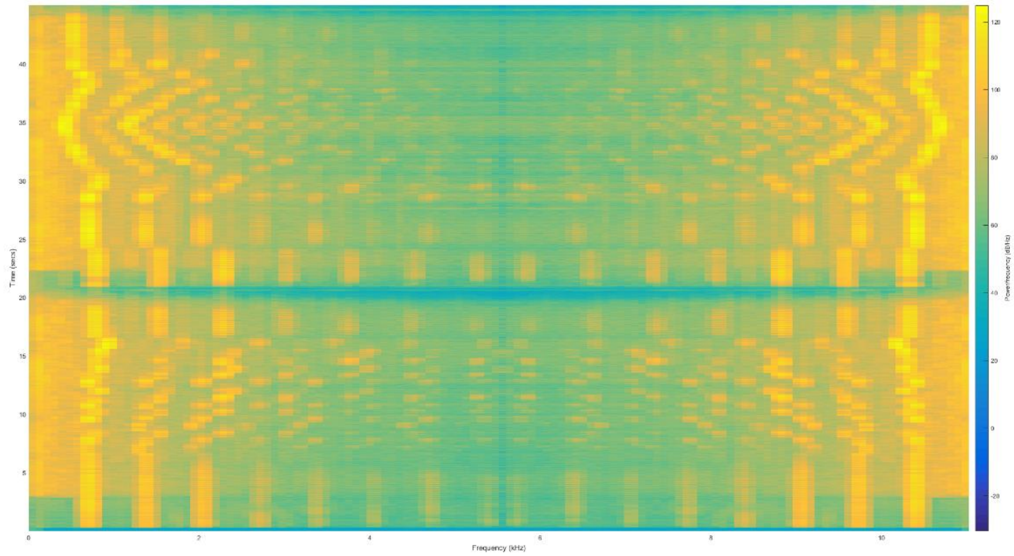


Above is the filter for up-sampling by 2. This appears to match what everyone else is getting (note I use fftshift, so the "moustache" is split over the frequency boundary).

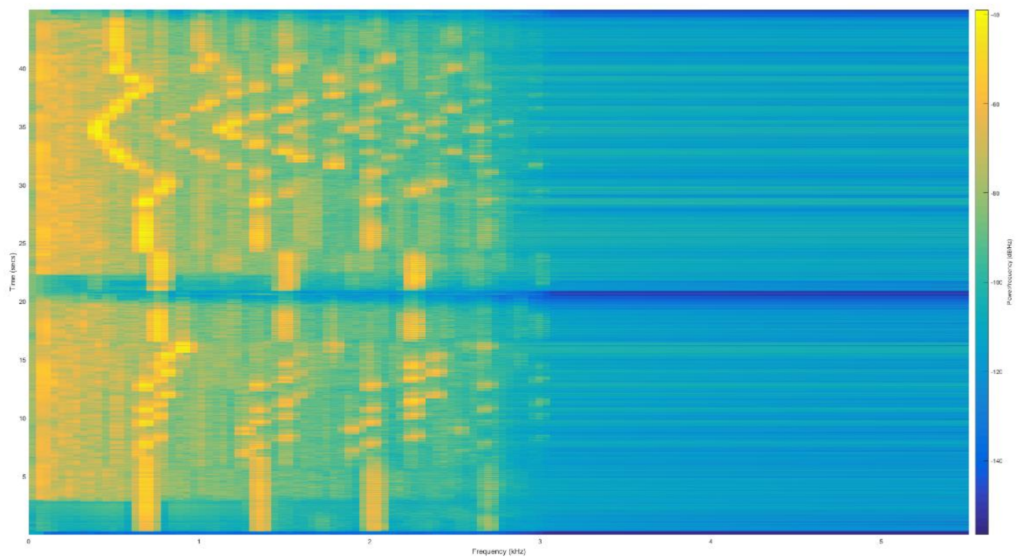
You can see down-sampling by 5 (used a slightly different filter but it has the same form, just narrower) results in correct behavior.



But I expect for up-sampling by 2 half of the screen to be blue. Instead I see aliasing….



One of the things I looked at in debugging was if I did the convolution when hard-coding U = 1 (so the filter should make it look squeezed in half).



This graph looks about right except the scale should be twice what it is on the frequency axis. I can't figure out what the deal is here, honestly. When I set U = 2 again in my c code I see this wacky aliasing at f=10, instead of a cutoff at f=5. Where as in U = 1 there is a cutoff at f=2.5 (and null space up to f=5).