Computer Assignment 2

Sample Rate Converter

Nik Jensen
ECE 5630
A02250195
11/19/2021

Objective

The objective of this assignment was to practice programming for digital signal processing relating to multi-rate processing. This assignment covered topics of decimation and interpolation and as well as polyphase filtering for optimization. The assigned task was to manipulate the sample rate of the provided inputs by the ratio provided, L/M = 3/2. This was done by passing the input through decimation filters and interpolation filters to create a converted sample scaled by 3/2. This task was then expanded by using polyphase filtering to minimize computations during processing.

FIR Low-Pass Filter Design (part 1)

To begin, a low-pass filter was needed for both types conversion structures. To create this filter, the built in Matlab tool, fdatool was used. Referencing *Figure 1* below, we can see the process to create this filter and as well as the plot for the magnitude and phase response. We can also see in *Figure 2*, the impulse response of the filter that was created by plotting the created coefficients. Keep in mind, for the sample rate conversion process, this filter's amplitude was scaled by L=3 to match the correct conversion process.

Next, the cutoff frequency was needed in order to create an appropriate conversion. This was determined by taking half of the sampling frequency, 11025Hz, and finding the minimum value when either dividing that value by L or M. For this assignment, the cutoff frequency was determined to be 5512.5/3 = 1837.5Hz. With this value the frequency stop and frequency pass were calculated to be Fpass = 1653.75Hz and Fstop = 2021.25Hz. Then, the ripple in the pass-band was declared to be 0.001dB and the peak side-lobe level was calculated to be Astop = 80 + 20*log(L) = 89.54. This was to accommodate for our filter's upsampling of L.

Lastly, the filter was normalized for ease of reading in *Figure 1*. The pass and stop band frequencies were determined by dividing their values by the length of the filter, 11025/2 = 5512.50.

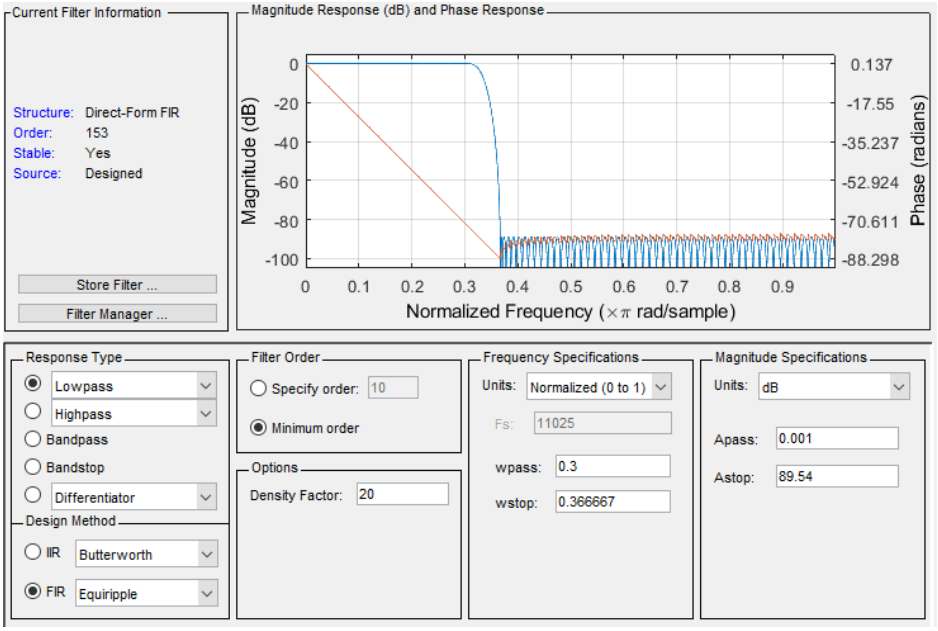

Figure 1 fdatool filter design. The plot contains the normalized magnitude and phase response of our low-pass filter

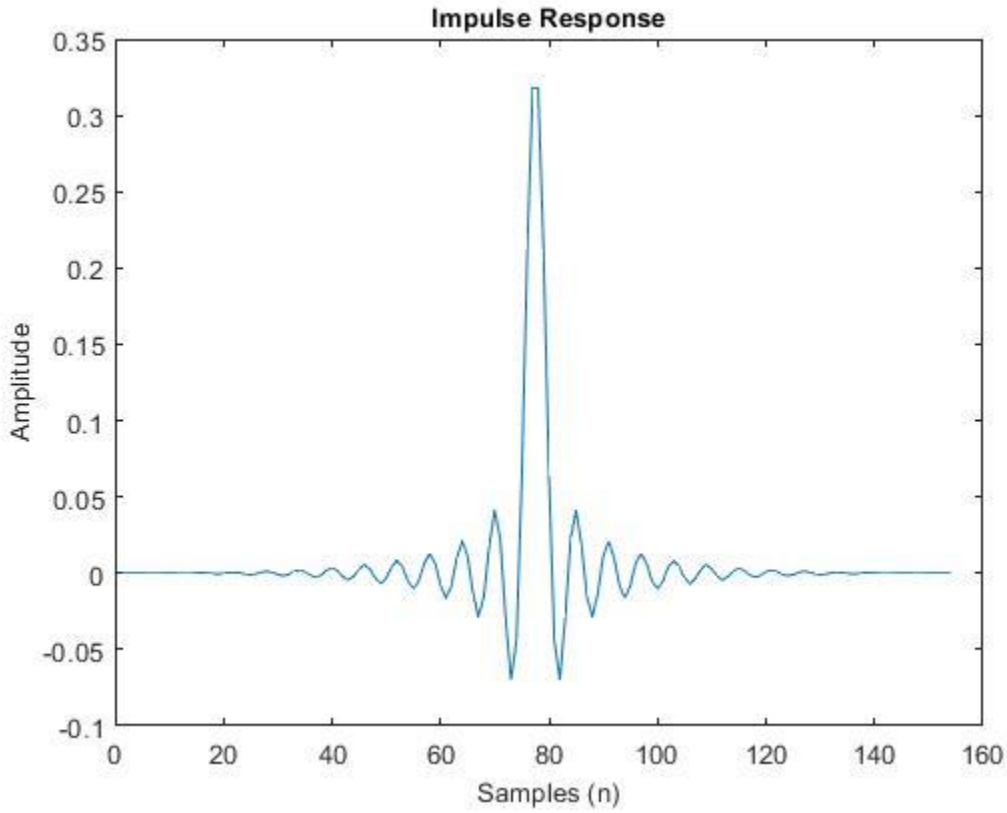

Figure 2 The low-pass filter's impulse response when plotted.

Polyphase Signal Flow Design & Type II Filters (part 2 & 3)

For this task, a polyphase filter was required to change the sample rate of the input by 3/2. The requirement was to reduce the number of operations per sample to a minimum. To my knowledge, I have implemented the filter provided below in *Figure 3*. Using this filter, the output has a sample rate of 16.538kHz. The implemented polyphase filter also has a total of 22 multiplies and 5 adds per sample.

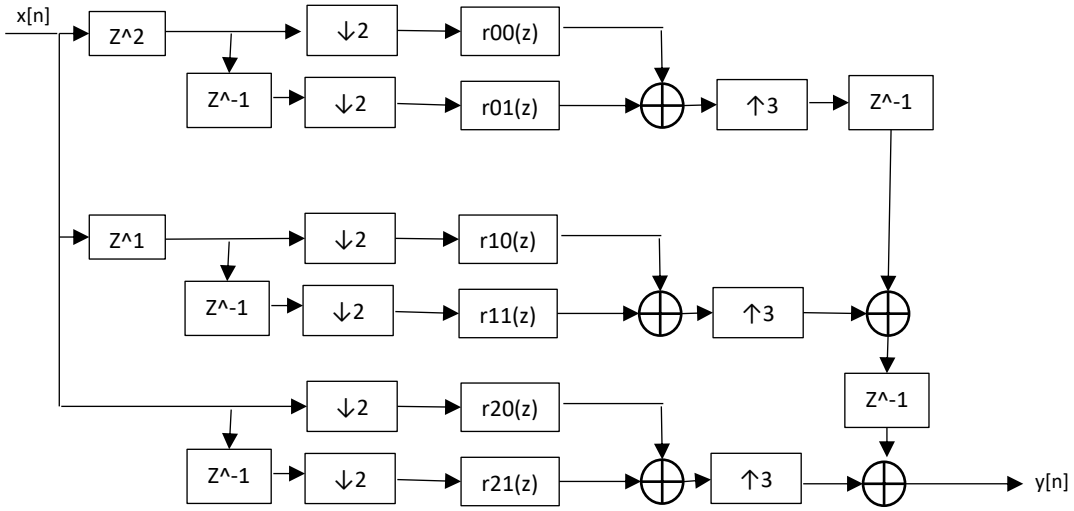

Figure 3 polyphase filtering for L = 3 and M = 2.

A difficulty with this implementation was determining the type II polyphase filters. This was done by splitting the impulse response into three sections which were then split into two more filters per section. An example of how these filters were created is in *Figure 4* below.

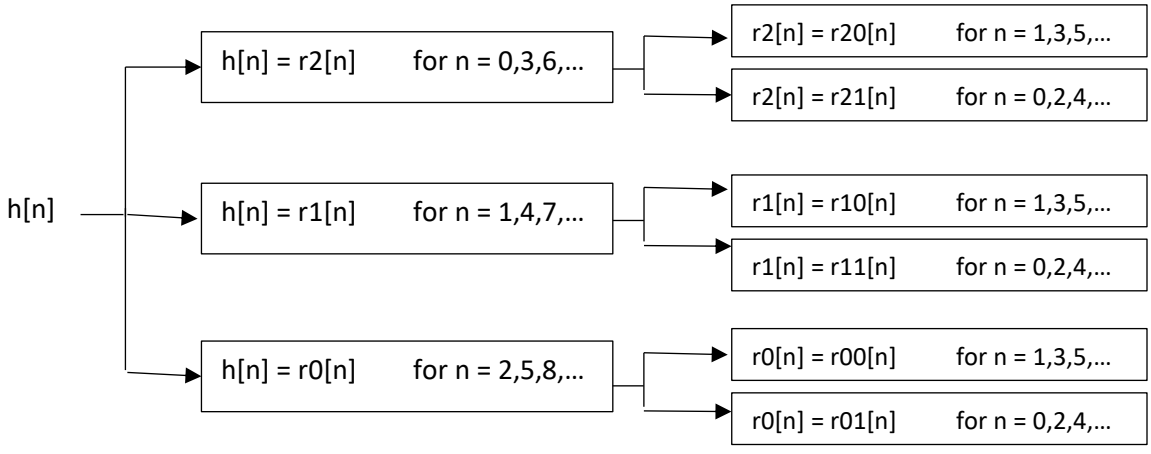

Figure 4 conversion from the impulse response to our polyphase type 2 filters

Cosine Implementation (4 & 5)

The next task was to design and implement a standard sample rate conversion and the polyphase sample rate conversion. This was done by creating cosines of different frequencies (f_o=1/16, 1/8,1/4) using the input equation $x[n] = \cos(2\pi f_o n)$. Because these cosines can not be infinite, the length chose was n=0,1,…,N-1 where N-1 is 154,000. 100 times the size of our low-pass filter. The processing was done using C code, which can be found in the Appendix of this document. Below, Figure 5, 6, and 7, is the comparison of the two sample rate converter structures' DFT magnitude plots.

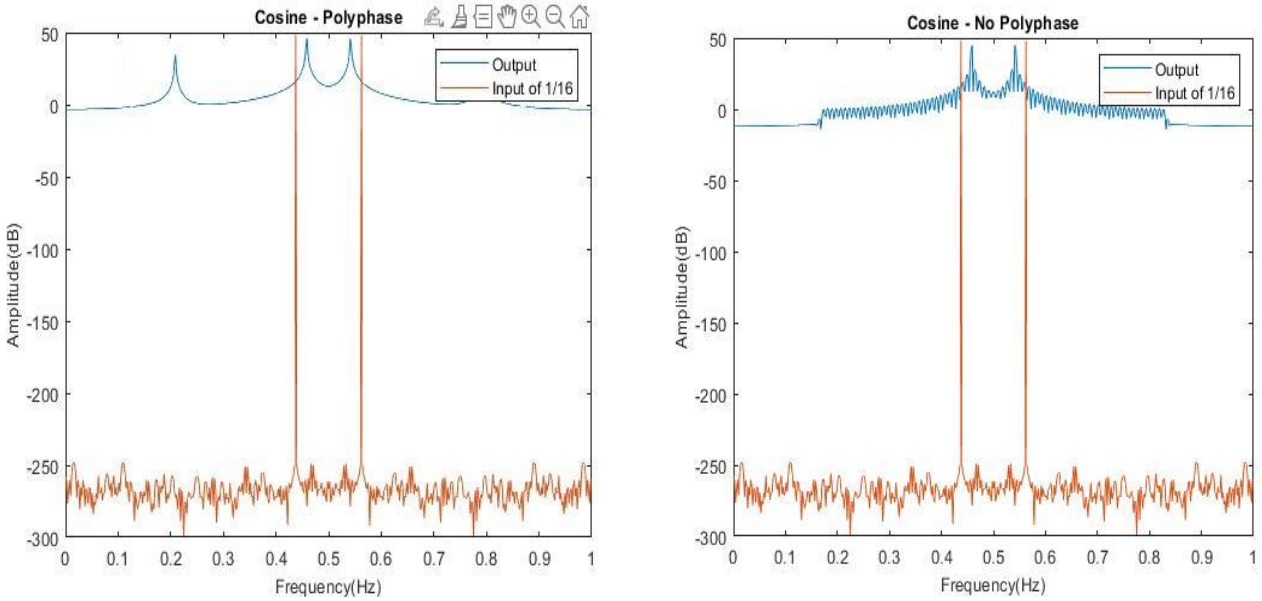

Figure 5 cosine frequency equal to 1/16 and its output in a DFT plot. Left is with polyphase, right is without.

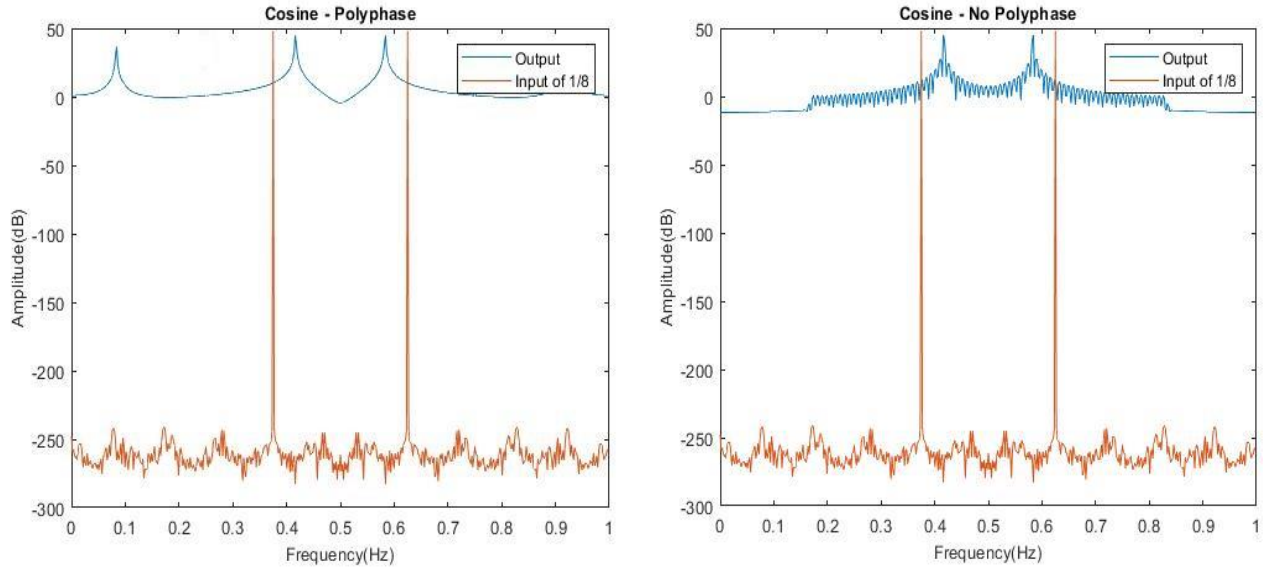

Figure 6 cosine frequency equal to 1/8 and its output in a DFT plot. Left is with polyphase, right is without.

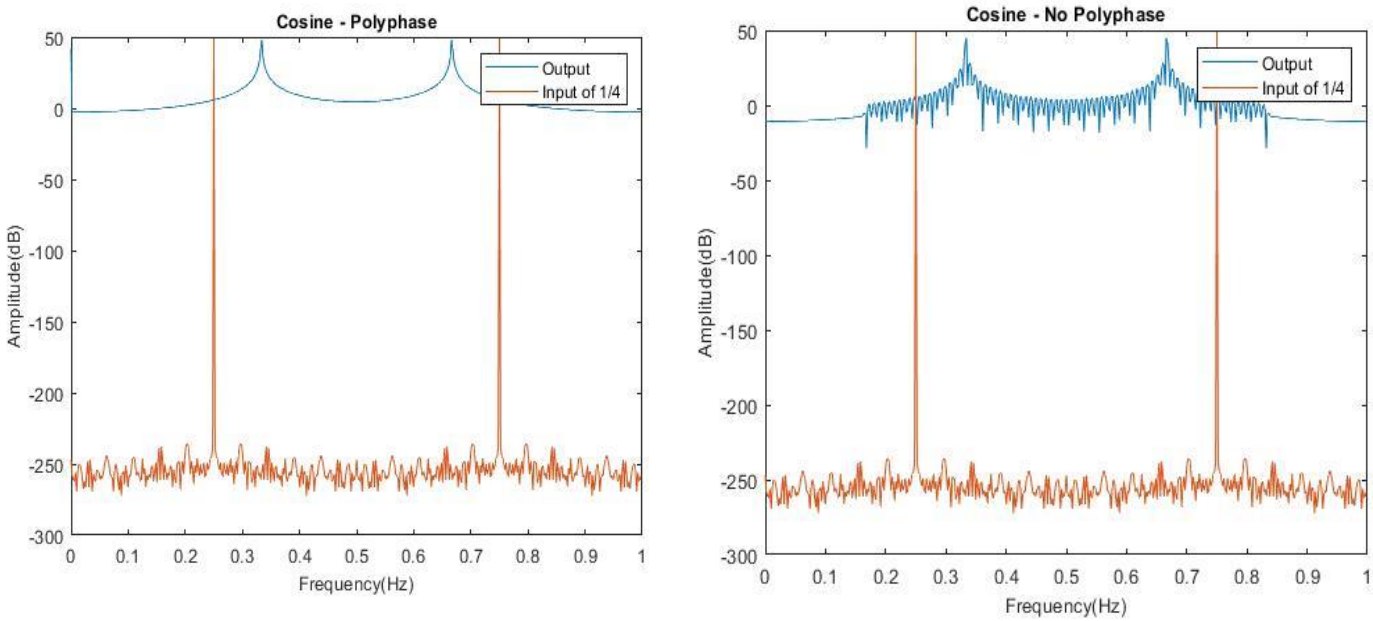

Figure 7 cosine frequency equal to 1/4 and its output in a DFT plot. Left is with polyphase, right is without.

Referencing the table below, we can see how both filters result in the same output frequency.

Input Frequency f	Standard Realization f_o	Polyphase Realization f_o
1/16	3/32	3/32
1/8	3/16	3/16
1/4	3/8	3/8

In Figure 8 below, is the implementation for the standard realization of sample rate conversion.

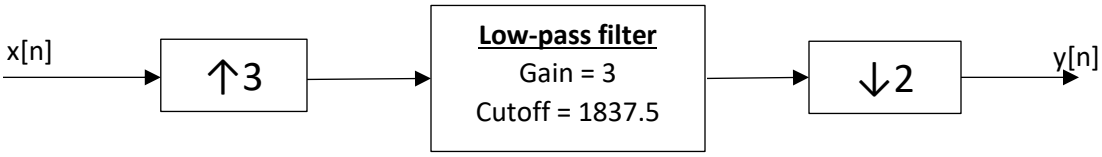

Figure 8 standard realization of sample rate conversion with ratio L/M = 3/2.

Ghostbusters (part 6)

The last section involved using both structures of sample rate conversion used. The task was to process the provided *ghostbustersray.wav* file into the sample rate converters, play them back, and compare both the sound and the execution time within the code. To be expected, the processed signals both sounded the same and had the same matching sample rate and length. The more interesting note is the execution time. The resulting execution times resulted in the standard structure at 7.8 milliseconds and the polyphase structure at 1.5 milliseconds. What surprised me is how fast the polyphase structure was. The processed sample rate was converted from 11,025 Hz to 16,538 Hz.

Conclusion

After working through this programming assignment, I gained a broader understanding of multi-rate processing, polyphase structures, sample rates, and digital signal processing. It is always a lot of fun to get a physicality into what I am working on. Using the *ghostbustersray.wav* was a great representation for multi-rate processing. In terms of results, it is clear to see that the polyphase structure is the optimal structure for multi-rate processing. While complex, with a little more calculations and writing, the polyphase implementation can save lots of computation time. This can be beneficial for many topics such as real-time data processing for computers, data collection, and possibly memory management. While seemingly correct, I am curious if my polyphase implementation was fully correct. Referencing Figure 5, 6, and 7, there is a second frequency spike in the resulting DFT plots. Although the sound file sounded the same as the input, I could not tell a difference with this possible frequency spike problem.

Appendix

Polyphase.c – polyphase implementation

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include "filter_3.h"
#include "cos_16.h"
#include "cos_8.h"
#include "cos_4.h"
#include <time.h>

/*...

#define L 3
#define M 2

typedef struct
{
 int ndim; //number of dimensions
 int nchan; //number of channels
 int d0;  //length of the first dimension
 int d1;  //length of second dimension or sample rate if audio
 int d2;  //length of third dimension
} dsp_file_header;

typedef struct
{
    float* r0;
    float* r1;
    float* r2;
} r_filters;

/**...
float* upsample(int l, float* x, float* xl, int Lzl, int Lx, int Lh){
    printf("Start Upsampling\n");
    int j = 0;
    int i;
    if(&Lh == NULL){ //in case of no zero padded input
        for(i = 0; i < (Lzl); i++){
            if(i%(L) == 0){
                // printf("i = %d, j = %d\n",i,j);
                xl[i] = x[j];
                j++;
            }
        }
    }
```

```
    else{ //zero padded input
        for(i = Lh-1; i < (Lzl-Lh-1); i++){
        // for(int i = Lh-1, j = 0; i < (Lzl-Lh-1), j < Lx; i++, j++){
            // printf("i = %d\tj = %d\n",i,j);
            if(i%(L) == 0){
                // printf("i = %d, j = %d\n",i,j);
                xl[i] = x[j];
                j++;
            }
        }
    }
    // printf("%d\n",sizeof(xl)); //i = 144175      j = 47899
    return xL;
}

/**...
float* downsample(int m, float* x, float* y, int Lx, int Ly, int Lh){
    printf("Start Downsampling\n");
    int j = 0;
    if(&Lh == NULL){ //in case of no zero padded input
        for(int i = 0; i < Lx; i++){
            if(i%(m) == 0){
                // printf("i = %d, j = %d\n",i,j);
                y[j] = x[i];
                j++;
            }
        }
    }
    else{ //zero padded input
        for(int i = Lh-1; i < (Lx-Lh-1); i++){
            if(i%(m) == 0){
                // printf("i = %d, j = %d\n",i,j);
                y[j] = x[i];
                j++;
            }
        }
    }
    return y;
}
```

```
float* conv(float* x, float* h, float* y, int Lx, int Lh, int Ly){
    printf("Start Convolution\n");
    int j;
    for (int i = 0; i < Ly; i++) {
        for (j = 0; j < Lh; j++) {
            y[i] += h[j] * x[i + j]; //multiply and accumulate (MAC)
        }
    }
    return y;
}

/**...
float* shift(float* input, float* output, int in_length, int out_length, int delay_val){
    for(int i = 0; i < in_length; i++){
        if(i+delay_val < out_length && i+delay_val > 0){
            output[i+delay_val] = input[i]; //get a move on!
        }
    }
    return output;
}

/**...
float* fetch_r_into_3(float* r0, float* r1, float* r2, float* h, int Lh){
    int j = 0, k = 0, n = 0;
    for(int i = 0; i < Lh; i++){
        if(i%3 == 0){ //n = 0, 3, 6, ...
            r2[j] = h[i];
            j++;
        }
        else if(i%3 == 1){ //n = 1, 4, 7, ...
            r1[k] = h[i];
            k++;
        }
        else if(i%3 == 2){ //n = 2, 5, 8, ...
            r0[n] = h[i];
            n++;
        }
    }
    return r0,r1,r2;
}
/**...
float* split_r_in_2(float* input, float* r0, float* r1, int length){
    int j = 0;
    int k = 0;
    for(int i = 0; i < length; i++){
        if(i%2 == 0){ //n = 0, 2, 4, ...
            r1[j] = input[i];
            j++;
        }
```

```
        else if(i%2 == 1){ //n = 1, 3, 5, ...
            r0[k] = input[i];
            k++;
        }
    }
    return r0, r1;
}

void main(int argc, char** argv){
    //read in file
    FILE* fx;
    FILE* fy;
    if (NULL == (fx = fopen(argv[1], "rb"))) { //error check and open file
        printf("error: Cannot open input file.\n");
        return;
    }
    if (NULL == (fy = fopen(argv[2], "wb"))) { //error check and open file
        printf("error: Cannot open output file for writing.\n");
        return;
    }
    //grab headers of each file
    dsp_file_header h0, h1, ho;
    fread(&h0, sizeof(dsp_file_header), 1, fx);
    memcpy(&ho, &h0, sizeof(dsp_file_header));
    int a = L;
    int b = M;
    float fs = (float)h0.d1;
    float fs_out = ((float)a/(float)b)*fs;
    printf("%d, %d\n",a,b);
    printf("%f\n",fs_out);
    ho.d1 = (int)16537.5;
    fwrite(&ho, sizeof(dsp_file_header), 1, fy);
    printf("ndim = %d, nchan = %d, d0 = %d, d1 = %d, d2 = %d\n", h0.ndim, h0.nchan, h0.d0, h0.

    int Lh = sizeof(h)/sizeof(h[0]); //coefficients from filter.h

    // int Lx = h0.d0;
    // int Lx = sizeof(x_16)/sizeof(x_16[0]);
    // int Lx = sizeof(x_8)/sizeof(x_8[0]);
    int Lx = sizeof(x_4)/sizeof(x_4[0]);
    int Lxl = h0.d0 * L;
    int Lz = Lxl + 2*(Lh-1); //length output for upsampled
    int Lv = Lxl + (Lh-1); //length of filtered upsampled signal
    int Ly = Lv / M; //final output length
```

```
// printf("Lh = %d, Lx = %d, Ly = %d, Lv = %d, Lzl = %d\n", Lh, Lx, Ly, Lv, Lz);
// float* x = calloc(sizeof(float), Lx); //allocate data for file store
// float* v = calloc(sizeof(float), Lz);
// float* y = calloc(sizeof(float), Ly);
// float* xl = calloc(sizeof(float), Lz);

// while (!feof(fx)) {
//     fread(x, sizeof(float), Lx, fx);
// }
printf("Create r statements\n");
int Lrk = Lh/L;
int Lrkk = Lrk/M;
float* r0 = calloc(sizeof(float), Lrk);
float* r1 = calloc(sizeof(float), Lrk);
float* r2 = calloc(sizeof(float), Lrk);
r0,r1,r2 = fetch_r_into_3(r0, r1, r2, h, Lh);
float* r00 = calloc(sizeof(float), Lrkk);
float* r01 = calloc(sizeof(float), Lrkk);
printf("SPLIT r statements\n");
r00, r01 = split_r_in_2(r0, r00, r01, Lrk);
float* r10 = calloc(sizeof(float), Lrkk);
float* r11 = calloc(sizeof(float), Lrkk);
printf("SPLIT r statements\n");
r10, r11 = split_r_in_2(r1, r10, r11, Lrk);
float* r20 = calloc(sizeof(float), Lrkk);
float* r21 = calloc(sizeof(float), Lrkk);
printf("SPLIT r statements\n");
r20, r21 = split_r_in_2(r2, r20, r21, Lrk);
free(r0);
free(r1);
free(r2);

int Ld_pad = (Lx/2)+2*(Lh-1);
int Lvk = (Lx/2)+(Lh-1);
clock_t begin = clock();
//1.)
float* x0_1 = calloc(sizeof(float), Lx);
float* x0_2 = calloc(sizeof(float), Lx);
//x0_1 = shift(x, x0_1, Lx, Lx, -2);
// x0_1 = shift(x_16, x0_1, Lx, Lx, -2);
// x0_1 = shift(x_8, x0_1, Lx, Lx, -2);
x0_1 = shift(x_4, x0_1, Lx, Lx, -2);
x0_2 = shift(x0_1, x0_2, Lx, Lx, 1);
float* xd0_1 = calloc(sizeof(float), Ld_pad);
float* xd0_2 = calloc(sizeof(float), Ld_pad);
xd0_1 = downsample(M, x0_1, xd0_1, Lx, Ld_pad, Lh);
xd0_2 = downsample(M, x0_2, xd0_2, Lx, Ld_pad, Lh);
float* v0_1 = calloc(sizeof(float), Lvk);
float* v0_2 = calloc(sizeof(float), Lvk);
```

```
v0_1 = conv(xd0_1, r00, v0_1, Ld_pad, Lrkk, Lvk);
v0_2 = conv(xd0_2, r01, v0_2, Ld_pad, Lrkk, Lvk);
float* v_0 = calloc(sizeof(float), Lvk);
for(int i = 0; i < Lvk; i++){
    v_0[i] = v0_1[i] + v0_2[i];
}
float* yu_0 = calloc(sizeof(float), Lvk*3);
yu_0 = upsample(L, v_0, yu_0, 3*Lvk, Lvk, NULL);
float* y_0 = calloc(sizeof(float), 3*Lvk);
y_0 = shift(yu_0, y_0, 3*Lvk, 3*Lvk, 1);
free(x0_1);
free(x0_2);
free(xd0_1);
free(xd0_2);
free(v0_1);
free(v0_2);
free(v_0);
free(yu_0);
free(r00);
free(r01);

//2.)
float* x1_1 = calloc(sizeof(float), Lx);
float* x1_2 = calloc(sizeof(float), Lx);
// x1_1 = shift(x, x1_1, Lx, Lx, -2);
// x1_1 = shift(x_16, x1_1, Lx, Lx, -2);
// x1_1 = shift(x_8, x1_1, Lx, Lx, -2);
x1_1 = shift(x_4, x1_1, Lx, Lx, -2);
x1_2 = shift(x1_1, x1_2, Lx, Lx, 1);
float* xd1_1 = calloc(sizeof(float), Ld_pad);
float* xd1_2 = calloc(sizeof(float), Ld_pad);
xd1_1 = downsample(M, x1_1, xd1_1, Lx, Ld_pad, Lh);
xd1_2 = downsample(M, x1_2, xd1_2, Lx, Ld_pad, Lh);
float* v1_1 = calloc(sizeof(float), Lvk);
float* v1_2 = calloc(sizeof(float), Lvk);
v1_1 = conv(xd1_1, r10, v1_1, Ld_pad, Lrkk, Lvk);
v1_2 = conv(xd1_2, r11, v1_2, Ld_pad, Lrkk, Lvk);
float* v_1 = calloc(sizeof(float), Lvk);
for(int i = 0; i < Lvk; i++){
    v_1[i] = v1_1[i] + v1_2[i];
}
float* y_1 = calloc(sizeof(float), Lvk*3);
y_1 = upsample(L, v_1, y_1, 3*Lvk, Lvk, NULL);
free(x1_1);
free(x1_2);
free(xd1_1);
free(xd1_2);
free(v1_1);
free(v1_2);
```

```
    free(v_1);
    free(r10);
    free(r11);



    // //3.)
    float* x2_1 = calloc(sizeof(float), Lx);
    float* x2_2 = calloc(sizeof(float), Lx);
    // x2_1 = shift(x, x2_1, Lx, Lx, -2);
    // x2_1 = shift(x_16, x2_1, Lx, Lx, -2);
    // x2_1 = shift(x_8, x2_1, Lx, Lx, -2);
    x2_1 = shift(x_4, x2_1, Lx, Lx, -2);
    x2_2 = shift(x2_1, x2_2, Lx, Lx, 1);
    float* xd2_1 = calloc(sizeof(float), Ld_pad);
    float* xd2_2 = calloc(sizeof(float), Ld_pad);
    xd2_1 = downsample(M, x2_1, xd2_1, Lx, Ld_pad, Lh);
    xd2_2 = downsample(M, x2_2, xd2_2, Lx, Ld_pad, Lh);
    float* v2_1 = calloc(sizeof(float), Lvk);
    float* v2_2 = calloc(sizeof(float), Lvk);
    v2_1 = conv(xd2_1, r20, v2_1, Ld_pad, Lrkk, Lvk);
    v2_2 = conv(xd2_2, r21, v2_2, Ld_pad, Lrkk, Lvk);
    float* v_2 = calloc(sizeof(float), Lvk);
    for(int i = 0; i < Lvk; i++){
        v_2[i] = v2_1[i] + v2_2[i];
    }
    float* y_2 = calloc(sizeof(float), Lvk*3);
    y_2 = upsample(L, v_2, y_2, 3*Lvk, Lvk, NULL);
    free(x2_1);
    free(x2_2);
    free(xd2_1);
    free(xd2_2);
    free(v2_1);
    free(v2_2);
    free(v_2);
    free(r20);
    free(r21);


    // //finish section
    float* y_01 = calloc(sizeof(float), 3*Lvk);
    for(int i = 0; i < 3*Lvk; i++){
        y_01[i] = y_0[i] + y_1[i];
    }
    float* y01 = calloc(sizeof(float), 3*Lvk);
    y01 = shift(y_01, y01, 3*Lvk, 3*Lvk, 1);
    float* y = calloc(sizeof(float), 3*Lvk);
    for(int i = 0; i < 3*Lvk; i++){
        y[i] = y_2[i] + y01[i];
    }
    clock_t end = clock();
    double time_spent = (double)(end - begin)/CLOCKS_PER_SEC;
    printf("Time of standard form = %f\n", time_spent);
    fwrite(y, sizeof(float), 3*Lvk, fy);
    printf("Lh = %d\n", Lh);
    // //output to file
    fclose(fx);
    fclose(fy);
}
```

Standard_conversion.c – standard implementation with cosine implementations. (not showing convolution, upsample, and downsample functions since they were shown in polyphase.c)

```c
void main(int argc, char** argv){
    //read in file
    FILE* fx;
    FILE* fy;
    if (NULL == (fx = fopen(argv[1], "rb"))) { //error check and open file
        printf("error: Cannot open input file.\n");
        return;
    }
    if (NULL == (fy = fopen(argv[2], "wb"))) { //error check and open file
        printf("error: Cannot open output file for writing.\n");
        return;
    }
    //grab headers of each file
    dsp_file_header h0, h1, ho;
    fread(&h0, sizeof(dsp_file_header), 1, fx);
    memcpy(&ho, &h0, sizeof(dsp_file_header));
    int a = L;
    int b = M;
    float fs = (float)h0.d1;
    float fs_out = ((float)a/(float)b)*fs;
    printf("%d, %d\n",a,b);
    printf("%f\n",fs_out);
    // ho.d1 = (int)16537.5;
    ho.d1 = (int)fs_out;
    fwrite(&ho, sizeof(dsp_file_header), 1, fy);
    printf("ndim = %d, nchan = %d, d0 = %d, d1 = %d, d2 = %d\n", h0.ndim, h0.nchan, h0.d0, h0.d1, h0.d2);

    int Lh = sizeof(h)/sizeof(h[0]); //coefficients from filter.h

    int Lx = h0.d0;
    int Lxl = h0.d0 * L;
    int Lz = Lxl + 2*(Lh-1); //length output for upsampled
    int Lv = Lxl + (Lh-1); //length of filtered upsampled signal
    int Ly = Lv / M; //final output length
    printf("Lh = %d, Lx = %d, Ly = %d, Lv = %d, Lzl = %d\n", Lh, Lx, Ly, Lv, Lz);
    float* x = calloc(sizeof(float), Lx); //allocate data for file store
    float* v = calloc(sizeof(float), Lz);
    float* y = calloc(sizeof(float), Ly);
    float* xl = calloc(sizeof(float), Lz);

    while (!feof(fx)) {
        fread(x, sizeof(float), Lx, fx);
    }
    clock_t begin = clock();
    xl = upsample(L, x, xl, Lz, Lx, Lh);
    v = conv(xl, h, v, Lz, Lh, Lv);
    y = downsample(M, v, y, Lv, Lx);
    clock_t end = clock();
    double time_spent = (double)(end - begin)/CLOCKS_PER_SEC;
    printf("Time of standard form = %f\n", time_spent);
    fwrite(y, sizeof(float), Ly, fy);
    printf("Lh = %d, Lx = %d, Ly = %d, Lv = %d, Lz = %d\n", Lh, Lx, Ly, Lv, Lz);
    printf("ndim = %d, nchan = %d, d0 = %d, d1 = %d, d2 = %d\n", ho.ndim, ho.nchan, ho.d0, ho.d1, ho.d2);
```

```
    //-------------------------------------------------------------------------------------------------------cos_16
    int Lx_16 = sizeof(x_16)/sizeof(x_16[0]);
    int Lxl_16 = Lx_16 * L;
    int Lz_16 = Lxl_16 + 2*(Lh-1); //length output for upsampled
    int Lv_16 = Lxl_16 + (Lh-1); //length of filtered upsampled signal
    int Ly_16 = Lv_16 / M; //final output length
    float* v_16 = calloc(sizeof(float), Lz_16);
    float* y_16 = calloc(sizeof(float), Ly_16);
    float* xl_16 = calloc(sizeof(float), Lz_16);
    printf("Lh_16 = %d, Lx_16 = %d, Ly_16 = %d, Lv_16 = %d, Lzl_16 = %d\n", Lh, Lx_16, Ly_16, Lv_16, Lz_16);

    xl_16 = upsample(L, x_16, xl_16, Lz_16, Lx_16, Lh);
    // fwrite(xl_16, sizeof(float), Lxl_16, fy);
    v_16 = conv(xl_16, h, v_16, Lz_16, Lh, Lv_16);
    // fwrite(v_16, sizeof(float), Lv_16, fy);
    y_16 = downsample(M, v_16, y_16, Lv_16, Ly_16);
    fwrite(y_16, sizeof(float), Ly_16, fy);
    //-------------------------------------------------------------------------------------------------------cos_8
    int Lx_8 = sizeof(x_8)/sizeof(x_8[0]);
    int Lxl_8 = Lx_8 * L;
    int Lz_8 = Lxl_8 + 2*(Lh-1); //length output for upsampled
    int Lv_8 = Lxl_8 + (Lh-1); //length of filtered upsampled signal
    int Ly_8 = Lv_8 / M; //final output length
    float* v_8 = calloc(sizeof(float), Lz_8);
    float* y_8 = calloc(sizeof(float), Ly_8);
    float* xl_8 = calloc(sizeof(float), Lz_8);
    printf("Lh_8 = %d, Lx_8 = %d, Ly_8 = %d, Lv_8 = %d, Lzl_8 = %d\n", Lh, Lx_8, Ly_8, Lv_8, Lz_8);

    xl_8 = upsample(L, x_8, xl_8, Lz_8, Lx_8, Lh);
    v_8 = conv(xl_8, h, v_8, Lz_8, Lh, Lv_8);
    y_8 = downsample(M, v_8, y_8, Lv_8, Ly_8);
    fwrite(y_8, sizeof(float), Ly_8, fy);
    //-------------------------------------------------------------------------------------------------------cos_4
    int Lx_4 = sizeof(x_4)/sizeof(x_4[0]);
    int Lxl_4 = Lx_4 * L;
    int Lz_4 = Lxl_4 + 2*(Lh-1); //length output for upsampled
    int Lv_4 = Lxl_4 + (Lh-1); //length of filtered upsampled signal
    int Ly_4 = Lv_4 / M; //final output length
    float* v_4 = calloc(sizeof(float), Lz_4);
    float* y_4 = calloc(sizeof(float), Ly_4);
    float* xl_4 = calloc(sizeof(float), Lz_4);
    printf("Lh_4 = %d, Lx_4 = %d, Ly_4 = %d, Lv_4 = %d, Lzl_4 = %d\n", Lh, Lx_4, Ly_4, Lv_4, Lz_4);

    xl_4 = upsample(L, x_4, xl_4, Lz_4, Lx_4, Lh);
    v_4 = conv(xl_4, h, v_4, Lz_4, Lh, Lv_4);
    printf("v_4[10] = %f\n", v_4[10]);
    y_4 = downsample(M, v_4, y_4, Lv_4, Ly_4);
    printf("y_4[10] = %f\n", y_4[10]);
    fwrite(y_4, sizeof(float), Ly_4, fy);
    //-------------------------------------------------------------------------------------------------------
    // //output to file
    fclose(fx);
    fclose(fy);
}
```