Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Szélessávú Hírközlés és Villamosságtan Tanszék

Rádióátviteli mérések laboratórium 2

9. mérés
Digitális KF

Szilágyi Gábor NOMK01

Budapest, 2023. március 15.

1. A feladat

A feladat egy komplex jel mintasorozatából egy megadott szórókóddal kiterjesztett spektrumú bináris adatcsomag kinyerése volt. A vett mintasorozat felfogható egy OFDM adásnak is, amelynek egy alvivőjéről kell visszaállítani az adatot. A szórókód egy 128 hosszú QPSK szimbólum-sorozat. Ezzel a szórókóddal korreláltatva a megfelelően előfeldolgozott jelből ki lehet nyerni a megfejtést, ami néhány bájtnyi bináris adatot jelent. Az adat visszaállítását segítendő, egy előre megadott prefix-szel kezdődik az adatcsomag (`0xAAAA2DD4`). A személyre szóló szórókód az én esetemben a következő:

```
1 32221230330320130231012321111203223331302123222220312003132303222
2 11202110103222101322032231122132031130312132232321200311012111111
```

Ez a kód úgy értelmezendő, hogy minden számjegy egy szimbólumnak felel meg: $0 \to 1$, $1 \to j$, $2 \to -1$, $3 \to -j$. A kód melléknyaláb-elnyomása 19,8 dB. A kiindulási mintasorozat 2×32 bites, komplex lebegőpontos mintákból áll $(I_1, Q_1, I_2, Q_2, ...)$. Ez a kiindulási adat zajjal terhelt és más szórókódokkal kiterjesztett, más vivőfrekvenciájú jeleket is tartalmaz. Ehhez adott néhány (személyre szóló) adat:

- $f_s = 4\,\text{MHz}$ (Mintavételi frekvencia)
- $f_v = 1{,}44\,\text{MHz}$ (Vivőfrekvencia)
- $\Delta f_v = 80\,\text{kHz}$ (A szomszédos vivőfrekvenciák távolsága)
- $f_{dr} = 40\,\text{kHz}$ (Szimbólumsebesség)

2. A megoldás lépései

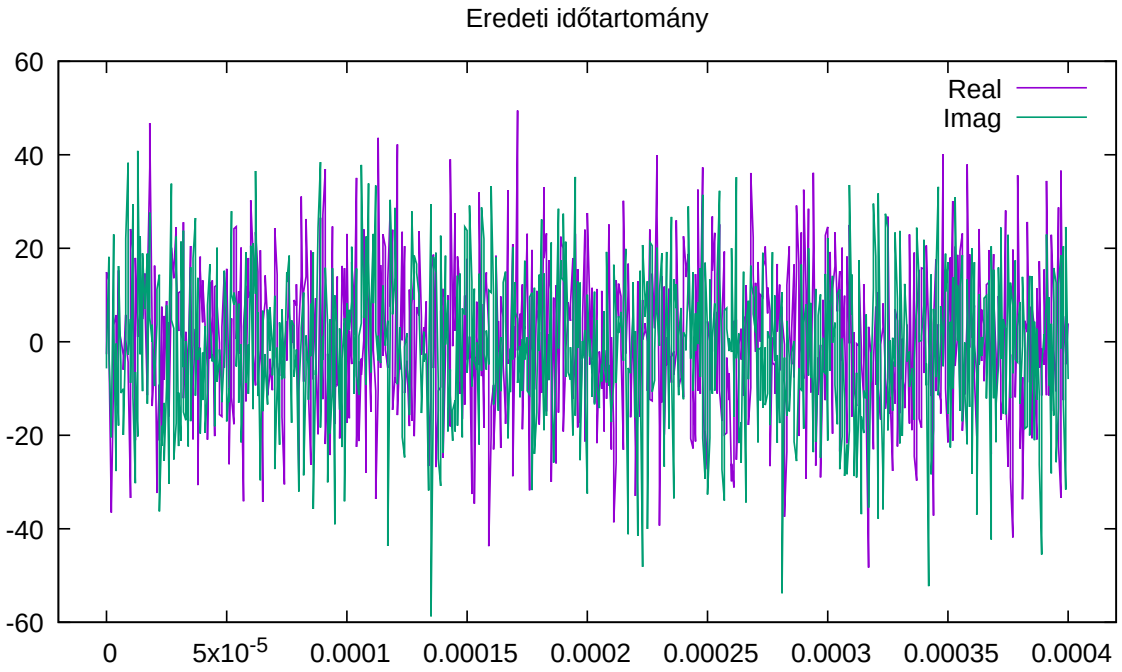
A lépések nagyvonalakban a következők.

1. Be kell olvasni az adott mintafájlból a komplex mintákat. $(a[i])$
2. Egy komplex numerikus oszcillátorral le kell keverni az f_v vivőfrekvenciáról DC-re a jelet. $(a[i] \to b[i])$
3. Aluláteresztő szűrni kell, hogy javuljon a jel-zaj viszony és a decimálásnál ne lapolódjanak egymásra a spektrum különböző részei. $(b[i] \to c[i])$
4. Decimálni kell a megfelelő faktorral, hogy egy szimbólum éppen olyan hosszú ideig tartson, mint a mintavételi periódusidő. Ez leegyszerűsíti a következő korrelációs lépést. $(c[i] \to d[i])$
5. A decimált jelet korreláltatni kell a szórókód konjugáltjával. $(d[i] \to e[i])$
6. A korreláció-jelről le lehet olvasni a pozitív és negatív csúcsokat, amelyek a bináris adat 0-s és 1-es bitjeihez vannak rendelve.

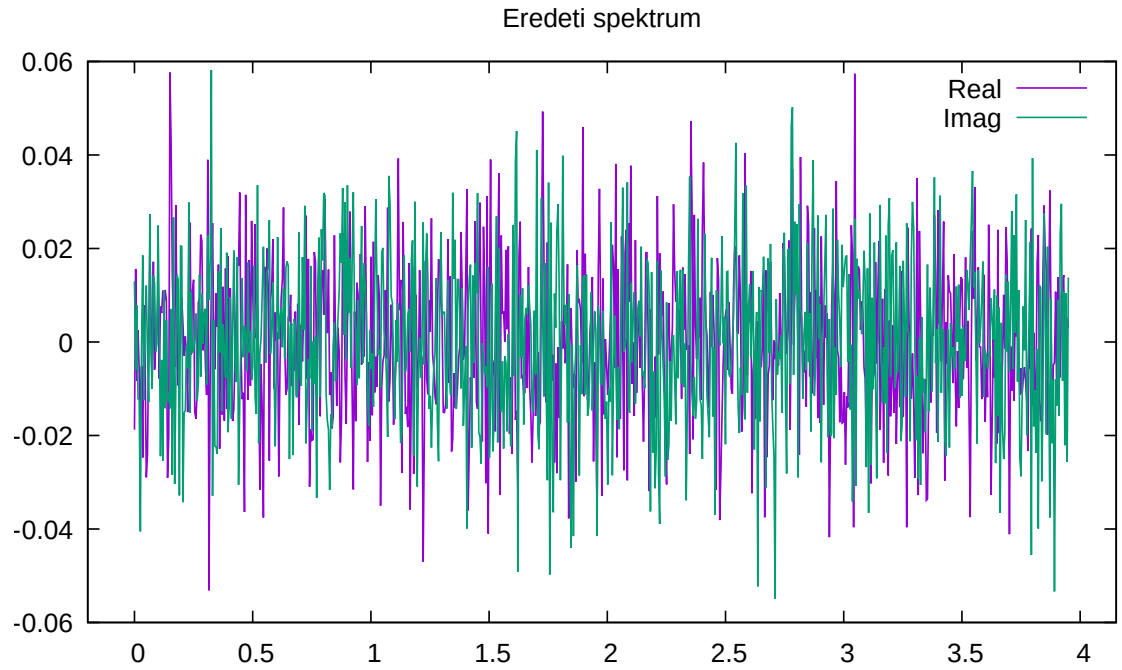
Itt az $x[i]$ jelenti a komplex vektorok i-edik elemét, a következőkben pedik j az imaginárius egységet.

3. A megoldás

A feladatra egy C programot írtam, amelyben az FFTW3 [1] könyvtárat használtam fel a spektrumok kiszámolásához és az aluláteresztő szűrés megvalósításához, valamint a Gnuplot [2] külső programot az ábrák generálásához. Az 1. ábrán látható az eredeti jel



(a) Az eredeti időtartománybeli jel első 1/512-ed része (0,4 ms).



(b) Az eredeti jel spektruma.

1. ábra. Az eredeti jel.

3.1. Keverés

A diszkrét időtartománybeli komplex mintasorozat keverése egy komplex körforgó vektorral való szorzással lehetséges. Ehhez ki kell számolni, hogy milyen $\Delta\varphi$ szöggel kell elfordulnia a keverő vektornak mintánként.

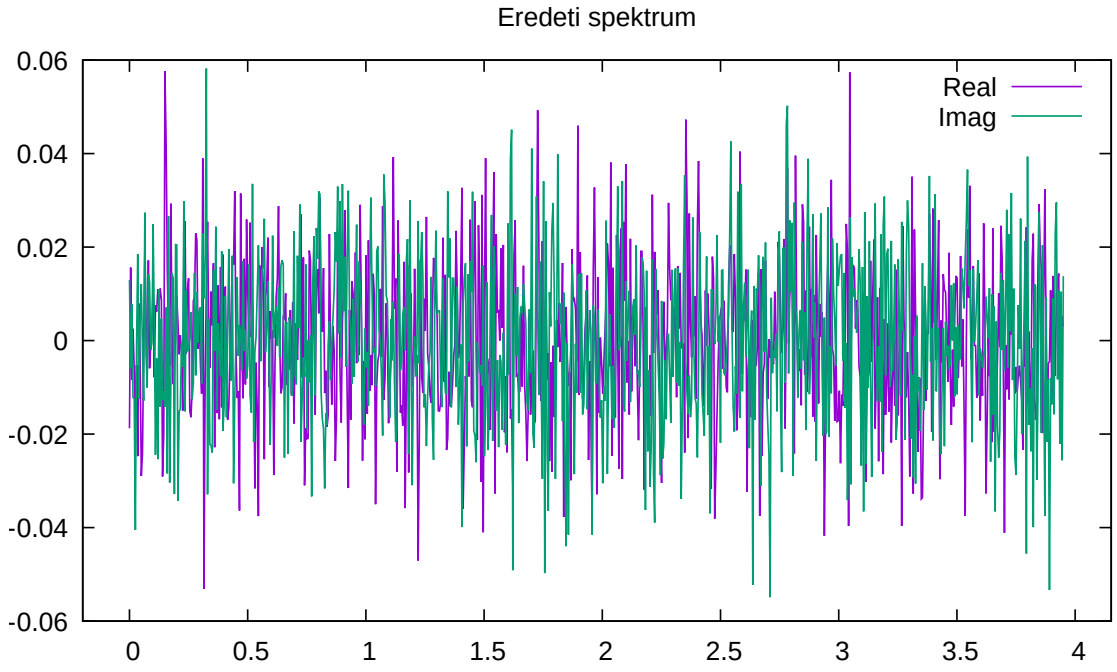
$$b[i] = a[i] \cdot \exp(-j \cdot i \cdot \Delta\varphi) \tag{1}$$

A keveréssel az f_v komplex vivőfrekvenciáról keverek 0 frekvenciára, így

$$\exp(j \cdot i \cdot 2\pi \frac{f_v}{f_s}) \times \exp(-j \cdot i \cdot \Delta\varphi) = 1 \tag{2}$$

$$\Delta\varphi = 2\pi \frac{f_v}{f_s} \tag{3}$$

A keverés után jellegre változatlan mind az időtartománybeli jel ($a[i]$), mind a spektruma ($A[i]$), de valójában az egész spektrum eltolódott balra 1,44 MHz-cel. Ez az eltolódás azért nem látszik az ábrákon, mert nem minden mintát plotolok ki, hanem csak minden 1024-ediket, hogy kicsi maradjon a fájlméret. Az időtartománybeli jeleknél viszont inkább csak a teljes időintervallum első 1/1000-edére ábrázolok, de azon belül minden mintát.



2. ábra. Az keverés utáni jel spektruma.

3.2. Aluláteresztő szűrés

Az én vivőmhöz tartozó jel feltehetőleg az $f_v \pm \frac{\Delta f_v}{2}$ sávon belül van, így a kevert jel spektrumának ($B[i]$) -40 kHz alatti és 40 kHz feletti részét kinullázva, majd ezt vissza Fourier-transzformálva egy aluláteresztő-szűrt jelet lehet kapni ($c[i]$). Ehhez csak azt kell

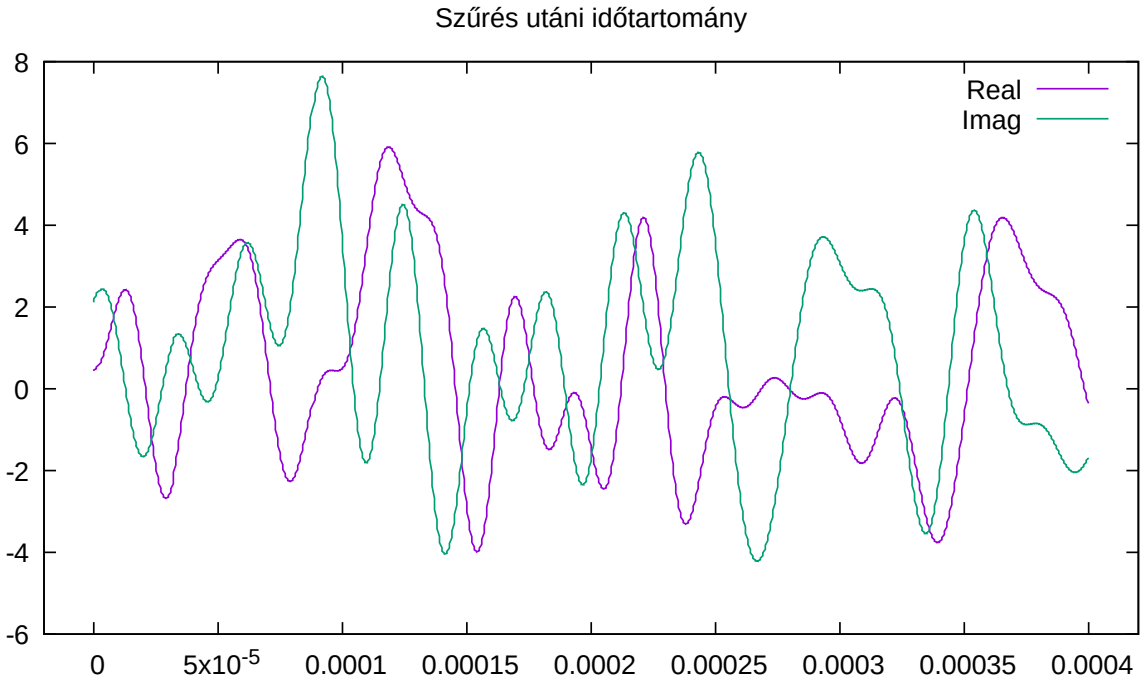
meghatározni, hogy a diszkrét spektrum melyik indexei között kell nullázni. A diszkrét spektrumban egy frekvencia-bin szélessége:

$$B_{bin} = \frac{f_s}{N} \tag{4}$$

Ahol N a minták száma. Tehát a kezdeti index (k) és végső index (v), ahol nullázni kell:

$$k = \frac{40\,\text{kHz} \cdot N}{f_s} \tag{5}$$

$$v = N - k \tag{6}$$



3. ábra. A szűrt időtartománybeli jel ($c[i]$) első 1/512-ed része (0,4 ms).

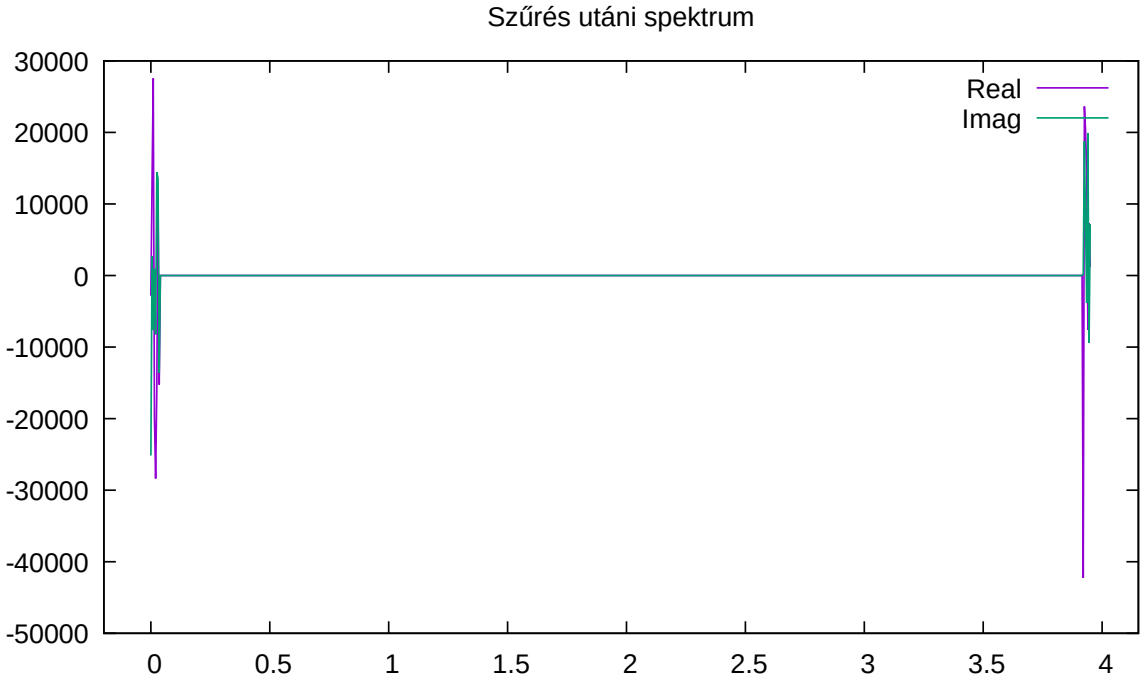
3.3. Decimálás

A decimálást úgy végeztem el, hogy meghatároztam a decimálás F faktorát, majd F db szűrt minta ($c[i]$) átlagát vettem egy mintának a decimált jelben ($d[i]$).

$$d[i] = \frac{1}{F} \sum_{l=(F-1)\cdot i}^{F\cdot i} c[l] \tag{7}$$

A faktor egyszerűen a következőképpen adódik:

$$F = \frac{f_s}{f_{df}} = 100 \tag{8}$$



4. ábra. A szűrt jel spektruma ($C[i]$).

3.4. Korrelálás

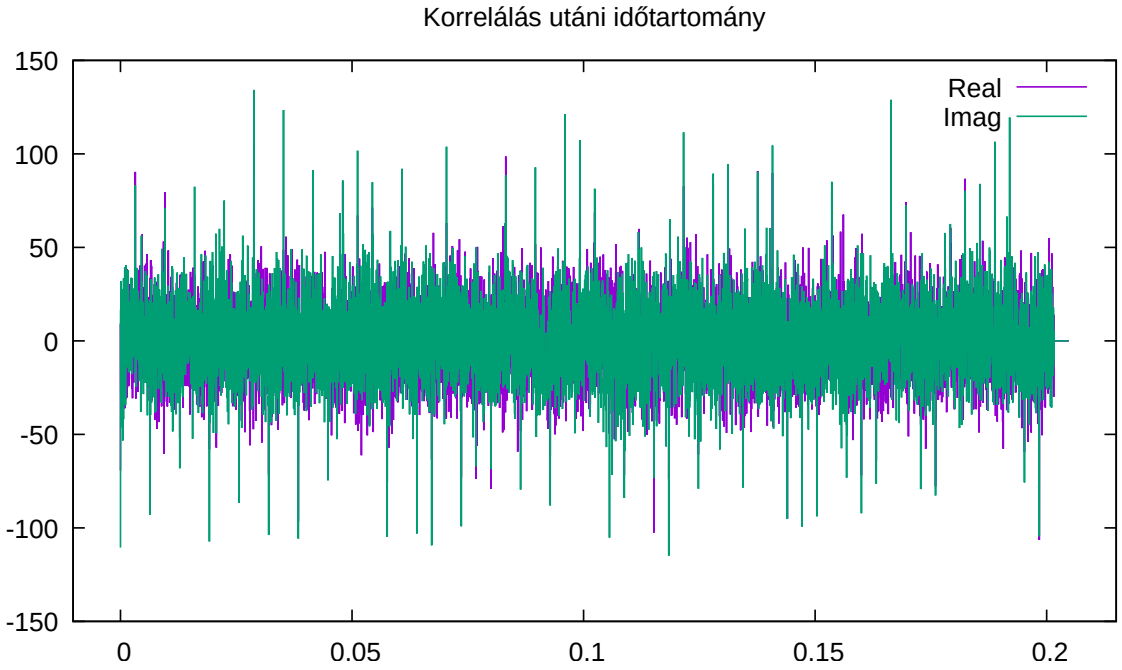
A kód számjegyeinek a sorozatából a hozzájuk tartozó QPSK szimbólumok konjugáltjait rendelte, majd ezzel egy csúszóablakos szorzatösszeg-számítással kaptam a korrelációt. Az 5. ábrán jól láthatóak a tüskék, amelyek egy a jelben lévő kódsorozat végét jelentik. A 6. ábrán bejelöltem a szimbólumok határait piros vonalakkal, valamint a leolvasott biteket. A kezdeti szinkronizáló bitsorozatból az derül ki, hogy a Q csatorna negatív csúcsa felel meg az 1-es bitnek, a pozitív csúcsa pedig a 0-s bitnek.

Tehát a megfejtés szinkronizáló minta utáni része a következő: `0x46D277306`

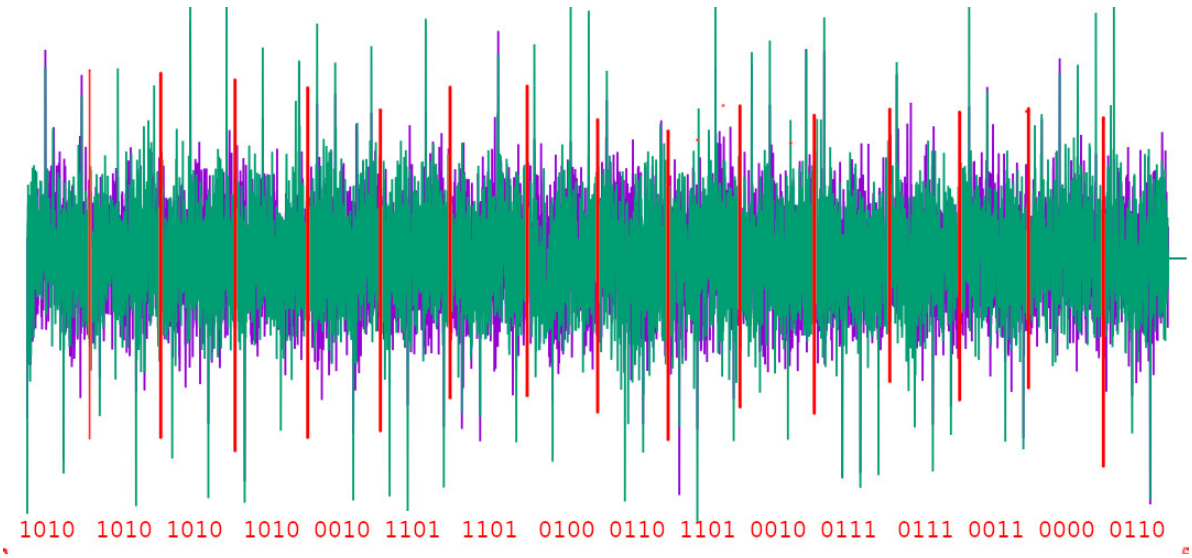
Hivatkozások

[1] FFTW3. http://fftw.org/ (elérve: 2023.03.15.).

[2] Gnuplot. http://www.gnuplot.info/ (elérve: 2023.03.15.).



5. ábra. A szórókóddal vett korreláció.



6. ábra. A megfejtés leolvasása.

A. digitkf.h

```c
#include <stdio.h>
#include <stdlib.h>
#include <fftw3.h>
#include <math.h>

// (*a)=(*a)*(*b)
void complex_mul(fftwf_complex *a, fftwf_complex *b);

// plot an array of complex numbers with gnuplot
void plot(float *xdata, fftwf_complex *arr, int length, int stride, const char *name,
    const char *title, int select);

void normalize(fftwf_complex *arr, int N);
```

B. digitkf.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <fftw3.h>
#include <math.h>
#include <digitkf.h>

// (*a)=(*a)*(*b)
void complex_mul(fftwf_complex *a, fftwf_complex *b)
{
    float im = (*a)[0]*(*b)[1] + (*a)[1]*(*b)[0];
    float re = (*a)[0]*(*b)[0] - (*a)[1]*(*b)[1];
    (*a)[0] = re;
    (*a)[1] = im;
    return;
}
//  fftwf_complex a, b;
//  a[0]=1.0f;
//  a[1]=2.0f;
//  b[0]=3.0f;
//  b[1]=4.0f;
//  printf("a=%f+%fi b=%f+%fi\n",a[0],a[1],b[0],b[1]);
//  complex_mul(&a, &b);
//  printf("a=%f+%fi b=%f+%fi\n",a[0],a[1],b[0],b[1]);


// plot an array of complex numbers with gnuplot
void plot(float *xdata, fftwf_complex *arr, int length, int stride, const char *name,
    const char *title, int select)
{
    if(select<1 || select>7)
    {
        printf("wrong plot selection\n");
        return;
    }

    FILE *fs = fopen("tmp/samples.txt","w");
    fprintf(fs, "#serial real imag abs\n");
    for(int i=0; i<length; i+=stride)
        fprintf(fs, "%f %f %f %f\n", xdata[i], arr[i][0], arr[i][1], sqrt(arr[i][0]*arr[i
    ][0] + arr[i][1]*arr[i][1]));
    fclose(fs);

    FILE *fp = fopen("tmp/plotter.gp", "w");
    fprintf(fp, "set terminal pdfcairo\n");
    fprintf(fp, "set output \"output/%s.pdf\"\n", name);
    fprintf(fp, "set title \"%s\"\n", title);
    fprintf(fp, "set xr [%f:%f]\n", (xdata[0]-xdata[length-1])*0.05f, xdata[length
    -1]*1.05f);

    if(select&1)
```

```
48      {
49          fprintf(fp, "plot \"tmp/samples.txt\" u 1:2 t \"Real\" w l,\\\n");
50      }
51
52      if(select&2)
53      {
54          if(select&1)
55          {
56              fprintf(fp, "     \"tmp/samples.txt\" u 1:3 t \"Imag\" w l,\\\n");
57          }
58          else
59          {
60              fprintf(fp, "plot \"tmp/samples.txt\" u 1:3 t \"Imag\" w l,\\\n");
61          }
62      }
63
64      if(select&4)
65      {
66          if(select&1 || select&2)
67          {
68              fprintf(fp, "     \"tmp/samples.txt\" u 1:4 t \"Abs\" w l");
69          }
70          else
71          {
72              fprintf(fp, "plot \"tmp/samples.txt\" u 1:4 t \"Abs\" w l");
73          }
74      }
75
76      fclose(fp);
77
78      if(system("gnuplot tmp/plotter.gp"))
79          printf("gnuplot error ...\n");
80
81      return;
82  }
83
84  // normalize time of frequency domain data after (I)FFT
85  void normalize(fftwf_complex *arr, int N)
86  {
87      float Nf = (float)N;
88      for(int i=0; i<N; i++)
89      {
90          arr[i][0] = arr[i][0]/Nf;
91          arr[i][1] = arr[i][1]/Nf;
92      }
93      return;
94  }
```

C. main.c

```
1   #include <stdio.h>
2   #include <stdlib.h>
3   #include <fftw3.h>
4   #include <math.h>
5   #include <digitkf.h>
6
7
8   #define FILENAME "../noisy.cf32"
9   #define SAMPLE_FREQ 4000000.0f
10  #define CARRIER_FREQ 1440000.0f
11  #define DATA_RATE 40000.0f
12  #define PI 3.14159265358979323846264338327795f
13  #define CODE_LENGTH 128
14  #define PLOT_REAL 1
15  #define PLOT_IMAG 2
16  #define PLOT_REAL_IMAG 3
17  #define PLOT_ABS 4
18  #define PLOT_REAL_ABS 5
19  #define PLOT_IMAG_ABS 6
```

```
#define PLOT_REAL_IMAG_ABS 7

int main(void)
{
    int code_int[] = {3, 2, 2, 2, 1, 2, 3, 0, 3, 3, 0, 3, 2, 0, 1, 3, 0, 2, 3, 1, 0, 1,
    2, 3, 2, 1, 1, 1, 1, 2, 0, 3, 2, 2, 3, 3, 3, 1, 3, 0, 2, 1, 2, 3, 2, 2, 2, 2, 2, 0,
    3, 1, 2, 0, 0, 3, 1, 3, 2, 3, 0, 3, 2, 2, 1, 1, 2, 0, 2, 1, 1, 0, 1, 0, 3, 2, 2, 2,
    1, 0, 1, 3, 2, 2, 0, 3, 2, 2, 3, 1, 1, 2, 2, 1, 3, 2, 0, 3, 1, 1, 3, 0, 3, 1, 2, 1,
    3, 2, 2, 3, 2, 3, 2, 1, 2, 0, 0, 3, 1, 1, 0, 1, 2, 1, 1, 1, 1, 1};
    fftwf_complex *code = malloc(sizeof(fftwf_complex)*CODE_LENGTH);
    for(int i=0; i<CODE_LENGTH; i++)
    {
        switch(code_int[i])
        {
            case 0:
                code[i][0] = 1.0f;
                code[i][1] = 0.0f;
                break;
            case 1:
                code[i][0] = 0.0f;
                code[i][1] = -1.0f;
                break;
            case 2:
                code[i][0] = -1.0f;
                code[i][1] = 0.0f;
                break;
            case 3:
                code[i][0] = 0.0f;
                code[i][1] = 1.0f;
                break;
        }
        //printf("%d -> %+01.0f%+01.0fi\n",code_int[i],code[i][0],code[i][1]);
    }


    // open file and determine number of complex samples
    FILE* noisy_file = fopen(FILENAME,"r");

    fseek(noisy_file, 0, SEEK_END);
    int N = ftell(noisy_file)/8;
    fseek(noisy_file, 0, SEEK_SET);

    fftwf_complex *samples = fftwf_malloc(sizeof(fftwf_complex) * N);
    fftwf_complex *spectrum = fftwf_malloc(sizeof(fftwf_complex) * N);
    size_t RET_CODE = fread(samples, sizeof(fftwf_complex), N, noisy_file);
    if(RET_CODE == N)
        printf("%s read successfully\n", FILENAME);
    else
        printf("error while reading %s\n", FILENAME);

    fclose(noisy_file);

    fftwf_plan pf, pb;
    pf = fftwf_plan_dft_1d(N, samples, spectrum, FFTW_FORWARD, FFTW_ESTIMATE);
    pb = fftwf_plan_dft_1d(N, spectrum, samples, FFTW_BACKWARD, FFTW_ESTIMATE);

    // frequency vector used for plotting
    float *freq_N;
    freq_N = malloc(sizeof(float)*N);
    float freq_step = 1.0f/(float)N*SAMPLE_FREQ/1000000.0f;
    //float freq_step = 1.0f/(float)N*SAMPLE_FREQ;
    float freq_stepper=0.0f;
    for(int i=0; i<N; i++)
    {
        freq_N[i] = freq_stepper;
        freq_stepper += freq_step;
    }

    // time vector used for plotting
    float *time_N;
    time_N = malloc(sizeof(float)*N);
    float time_step = 1.0f/SAMPLE_FREQ;
    //float freq_step = 1.0f/(float)N*SAMPLE_FREQ;
```

```
    float time_stepper=0.0f;
    for(int i=0; i<N; i++)
    {
        time_N[i] = time_stepper;
        time_stepper += time_step;
    }

    // print original time domain data
    plot(time_N, samples, N/512, 2, "original_samples", "original samples",
    PLOT_REAL_IMAG);

    // calculate and print original spectrum
       fftwf_execute(pf);
    normalize(spectrum, N);
    plot(freq_N, spectrum, N, 1024, "original_spectrum", "original spectrum",
    PLOT_REAL_IMAG);

    // mix from the carrier to DC
    float dfi = 2.0f*PI/SAMPLE_FREQ*CARRIER_FREQ;
    fftwf_complex mixer, rotator;
    rotator[0] = cos(dfi);
    rotator[1] = -sin(dfi);
    mixer[0] = 1.0f;
    mixer[1] = 0.0f;
    for(int i=0; i<N; i++)
    {
        complex_mul(&mixer, &rotator);
        complex_mul(&(samples[i]), &mixer);
    }

    // calculate mixed spectrum
    fftwf_execute(pf);

    // print mixed spectrum
    plot(freq_N, spectrum, N, 1024, "mixed_spectrum", "mixed spectrum", PLOT_REAL_IMAG);

    // brute force low pass filter
        // calculate first and last index to be zeroed
    int first = (int)(40000.0f*N/SAMPLE_FREQ);
    int last = N-first;
    for(int i=first; i<last+1; i++)
    {
        spectrum[i][0] = 0.0;
        spectrum[i][1] = 0.0;
    }

    // print filtered spectrum
    plot(freq_N, spectrum, N, 1024, "filtered_spectrum", "filtered spectrum",
    PLOT_REAL_IMAG);

    // print filtered time domain data
    fftwf_execute(pb);
    normalize(samples, N);
    plot(time_N, samples, N/512, 2, "filtered_samples", "filtered samples",
    PLOT_REAL_IMAG);

    // decimate time domain data
        // calculate decimation factor so that one spreading symbol lasts 16 samples
    int factor = (int)(SAMPLE_FREQ/DATA_RATE);
    printf("dec factor: %d\n",factor);
    int n = N/factor;
    fftwf_complex *decimated_samples = fftwf_malloc(sizeof(fftwf_complex)*n);
    fftwf_complex *correlated_samples = fftwf_malloc(sizeof(fftwf_complex)*n);
    fftwf_complex *decimated_spectrum = fftwf_malloc(sizeof(fftwf_complex)*n);

    // averaging decimation
    fftwf_complex sum;
    for(int i=0; i<n; i++)
    {
        sum[0]=0.0f;
        sum[1]=0.0f;
        for(int j=0; j<factor; j++)
        {
```

```
            sum[0] += samples[i*factor+j][0];
            sum[1] += samples[i*factor+j][1];
        }
        decimated_samples[i][0] = sum[0]/(float)factor;
        decimated_samples[i][1] = sum[1]/(float)factor;
    }

    // time vector used for plotting
    float *time_decimated;
    time_decimated = malloc(sizeof(float)*n);
    time_step = 1.0f/SAMPLE_FREQ*(float)factor;
    //float freq_step = 1.0f/(float)N*SAMPLE_FREQ;
    time_stepper=0.0f;
    for(int i=0; i<n; i++)
    {
        time_decimated[i] = time_stepper;
        time_stepper += time_step;
    }

    // plot decimated time domain data
    plot(time_decimated, decimated_samples, N/100/factor, 1, "decimated_samples", "
    decimated samples", PLOT_REAL_IMAG);

    //correlate with the spreading code
    for(int i=0; i<n-CODE_LENGTH; i++)
    {
        correlated_samples[i][0] = 0.0f;
        correlated_samples[i][1] = 0.0f;
        for(int j=0; j<CODE_LENGTH; j++)
        {
            correlated_samples[i][0] += decimated_samples[i+j][0]*code[j][0] -
    decimated_samples[i+j][1]*code[j][1];
            correlated_samples[i][1] += decimated_samples[i+j][0]*code[j][1] +
    decimated_samples[i+j][1]*code[j][0];
        }
    }

    // plot correlation
    plot(time_decimated, correlated_samples, n, 1, "correlated_samples", "correlated
    samples", PLOT_REAL_IMAG);

    free(code);
    free(time_decimated);
    fftwf_free(decimated_samples);
    fftwf_free(correlated_samples);
    fftwf_free(decimated_spectrum);
    free(freq_N);
    free(time_N);
    fftwf_destroy_plan(pf);
    fftwf_destroy_plan(pb);
    fftwf_free(samples);
    fftwf_free(spectrum);
    return 0;
}
```