



Elektrotehnički fakultet  
Univerzitet u Banjoj Luci

# **IZVJEŠTAJ PROJEKTOG ZADATAKA**

iz predmeta

## **SISTEMI ZA DIGITALNU OBRADU SIGNALA**

Student:  
Đaković Anja 1193/20

Mentori:

prof. dr	Knežić Mladen
prof. dr	Simić Mitar
ma	Jovanović Vedran
dipl. inž.	Obradović Dimitrije

Januar 2025. godine

## 1. Opis projektnog zadatka

U sklopu projektnog zadatka potrebno je realizovati sistem za generisanje empirijske vremensko-frekvencijske dekompozicije signala (Empirical Mode Decomposition - EMD) te rad ovog sistema testirati na fuziji multifokusiranih slika.

Sistem se sastoji iz 4 osnovna dijela:

- a. **Predprocesiranje** : Učitavanje 2D multifokusiranih slika u .bmp formatu, njihovo pretvaranje u Gray-scale, te skladištenje Gray-scale dobijene verzije u trajnu memoriju. Pikseli Gray-scale verzije originalne slike se čuvaju kao 1D vektor, zajedno sa informacijom o dužini tog niza, širini i visini slike, unutar header fajla. Korak predprocesiranja služi za pribavljanje svih neophodnih podataka iz 2D slike, a to su niz piksela, veličina niza, širina i visina slike, za dalju obradu.
- b. **EMD dekompozicija** : Određivanje IMF1 za signal koji smo dobili iz prethodnog koraka predprocesiranja. Za svaku od multifokusiranih slika smo kao izlaz iz prethodnog koraka predprocesiranja dobili po niz piksela. Svaki tako dobijen niz predstavlja ulaz za EMD algoritam, koji nam treba dati prvu IMF komponentu. Od izdvojenih IMF1 komponenata formiramo nove Gray-scale slike, tako da za niz piksela uzimamo upravo IMF1 niz. Imamo onoliko IMF1 komponenata koliko imamo i ulaznih nizova piksela, te za svaku od tih IMF1 komponenata formiramo po jednu 2D Gray-scale sliku.
- c. **Fuzija multifokusiranih slika** : Fuzija multifokusiranih slika kombinuje više fotografija iste scene, ali sa različitim dijelovima u fokusu, kako bi stvorila jednu oštru tj. potpuno fokusiranu sliku. EMD je jedna od tehnika za fuziju slika. Proces možemo opisati na sledeći način:
  - a. Kreće se od slika A1 i B1 kreiranih na osnovu prethodno izdvojene IMF1 komponente
  - b. Računaju se lokalne varijanse za svaki piksel na slikama A1 i B1
  - c. Računaju se težinski koeficijenti za prethodno dobijene varijanse
- d. **Konstrukcija fuzione slike** : Kako bismo dobili bolje fokusiranu sliku, kombinujemo slike A1 i B1 dobijene EMD algoritmom ali uzimajući u obzir i dobijene težinske koeficijente koji daju informaciju o mjestima dobre fokusiranosti na pojedinim slikama.

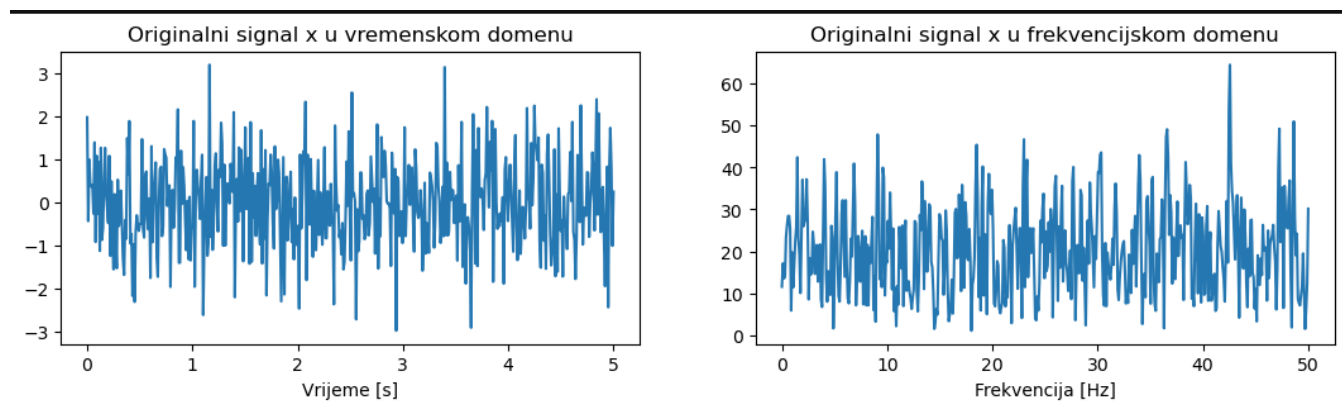
## 2. Izrada projektnog zadatka

### EMD algoritam

EMD (*Empirical Mode Decomposition*) je iterativni proces koji razbija signal u komponente zvane IMF (*intrinsic mode functions*). Svaka IMF komponenta sadrži najviše frekvencijsignala iz prethodne iteracije. Demonstriraćemo EMD algoritam na primjeru bijelog šuma.

Bijeli šum je odabran iz razloga što ne favorizuje niti jedan frekvencijski opseg. Bijeli šum predstavlja širokopolasni šum kojem je spektralna gustina neovisna o frekvenciji. Ima istu gustinu energije u svakom frekvencijskom opsegu. Prema tome, na linearnoj frekvencijskoj skali, oblik spektra bijelog šuma je ravna linija.

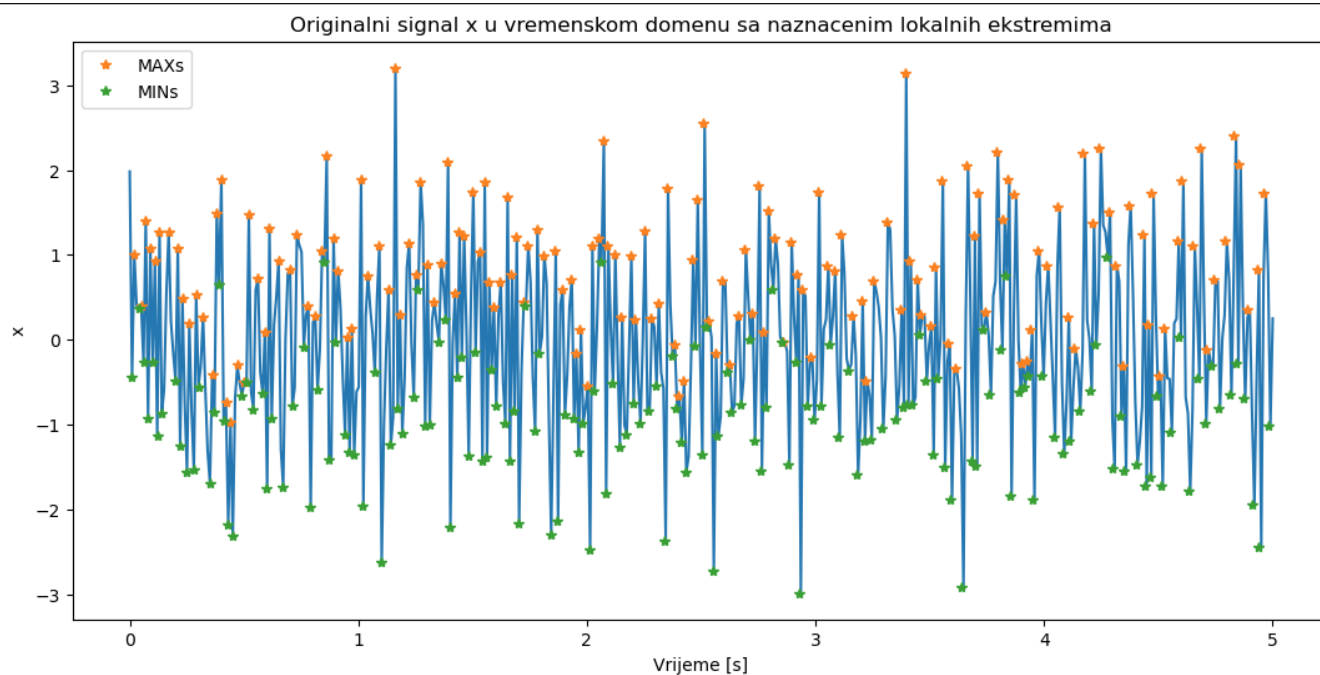
\*Python kod za demonstraciju rada EMD algoritma sa svim potrebnim graphicima i dodatnim objašnjenjima dat je u sklopu *emd.ipynb* fajla.



Slika 2.1. Prikaz polaznog signala x u vremenskom i frekvencijskom domenu.

## Prva iteracija ( pronalazak IMF1 )

### # 1 Pronalaženje lokalnih ekstrema polaznog signala x



Slika 2.2. Prikaz polaznog signala x u vremenskom domenu sa naznacenim lokalnim ekstremima

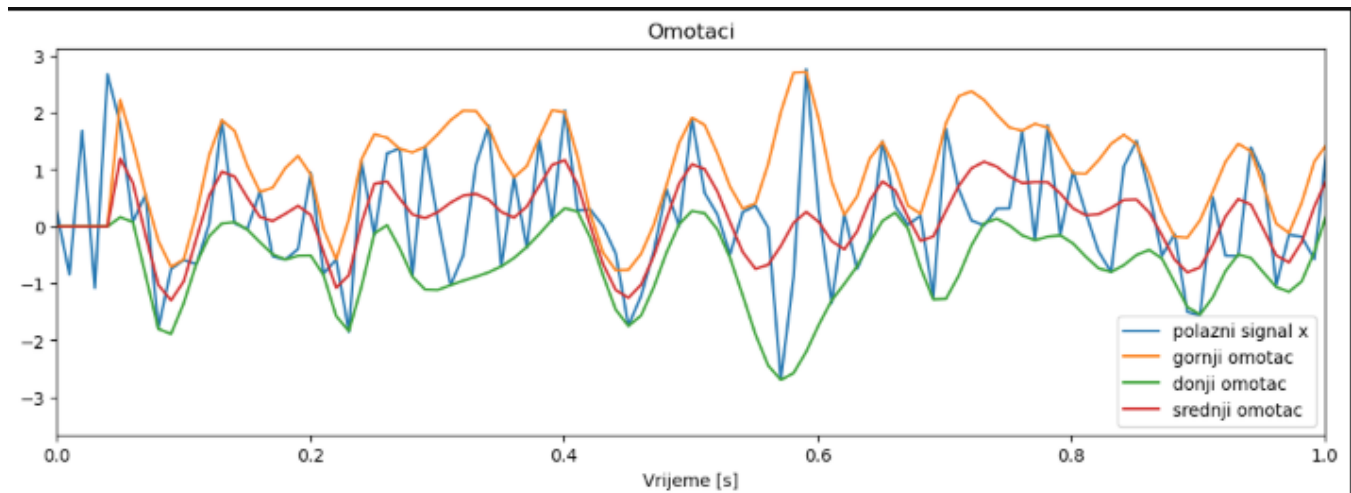
## # 2 Interpolacija lokalnih ekstrema s ciljem dobijanja srednjeg omotača

Interpoliramo lokalne maksimume kako bismo dobili gornji omotač

Interpoliramo lokalne minimume kako bismo dobili donji omotač

Srednji omotač dobijamo kao srednju vrijednost gornjeg i donjeg omotača (jednačina 2.1)

$$srednjiOmotac = \frac{gornjiOmotac + donjiOmotac}{2} \quad (2.1)$$



Slika 2.3. Omataci dobijeni interpolacijom lokalnih ekstrema polaznog signala x

## # 3 Izračunavanje kandidata za IMF1 komponentu

Prva IMF komponenta se definiše kao razlika polaznog (originalnog) signala i srednjeg omotača (jednačina 2.2)

$$tempIMF1 = x - srednjiOmotac \quad (2.2)$$

Ako dobijeni kandidat *tempIMF1* zadovoljava kriterijume 1 i 2, onda se on uzima kao prvi IMF.

Ako ne zadovoljava, postupak se ponavlja od #1 koraka (pronalaženje ekstrema), ali sa *tempIMF1* kao polaznim signalom ( $x = tempIMF1$ ), te se traže lokalni ekstremi nad *tempIMF1* signalom, a zatim se računaju interpolacije nad tako dobijenim ekstremima. Na #4 korak se prelazi tek onda kada dobijemo *tempIMF1* koji zadovoljava 1 i 2 kriterijume.

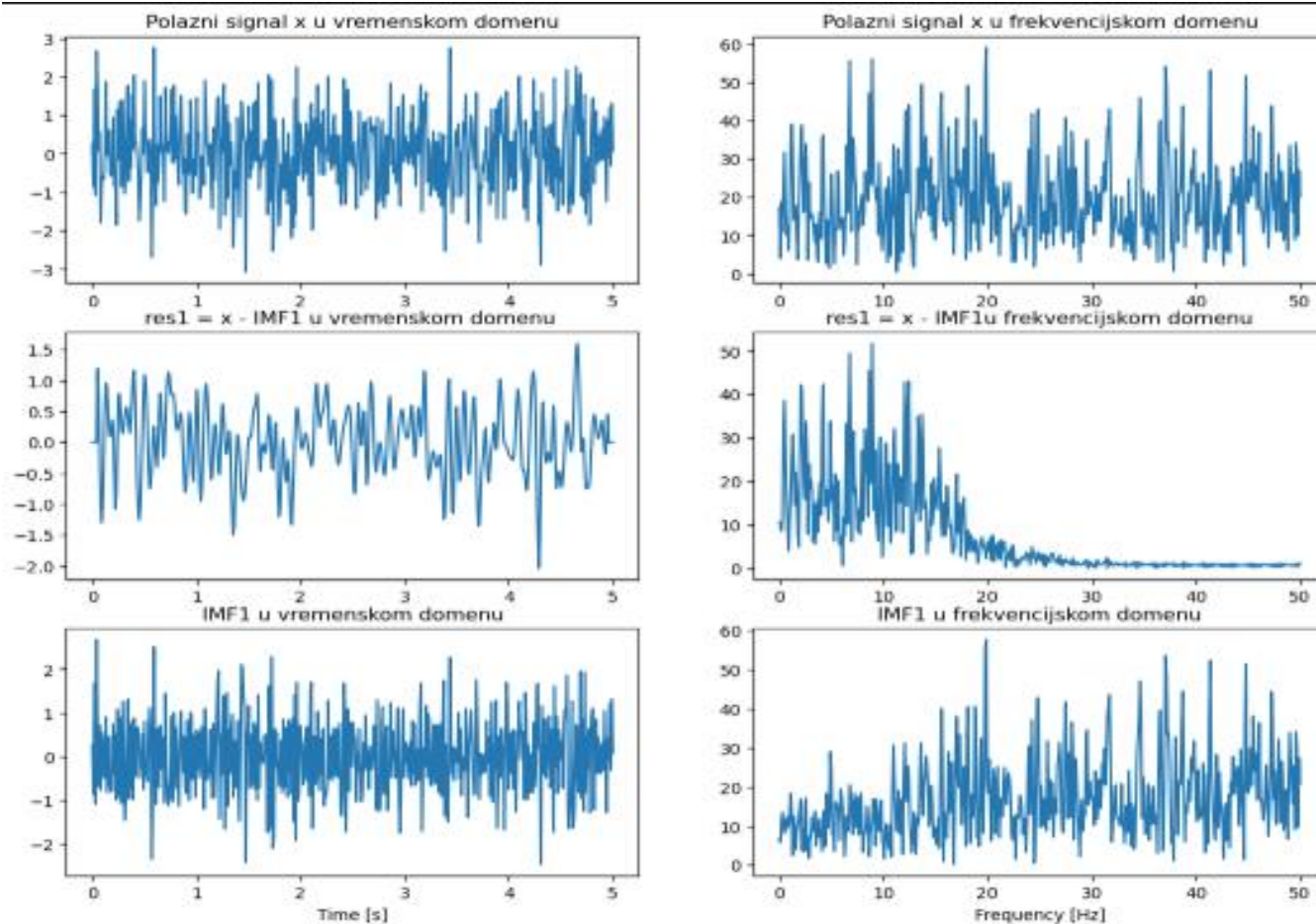
1. Broj prolazaka kroz nulu i broj lokalnih ekstrema se mogu razlikovati najviše za 1
2. Srednja vrijednost omotača treba biti blizu nule

#### # 4 Dekompozicija polaznog signala

Nakon izdvajanja prve IMF1 komponente, naš polazni signal  $x$  možemo napisati kao (2.3)

$$x = res1 + IMF1 \quad (2.3)$$

IMF1 sadrži najviše frekvencije polaznog signala  $x$ , a  $res1$  predstavlja polazni signal  $x$  umanjen za te najviše frekvencije. Sa slike 2.4. možemo da vidimo da spektar signala  $res1$  ne sadrži najviše frekvencijske komponente jer se one sada nalaze u sklopu IMF1 signala.



Slika 2.4. Vremenski i frekvencijski prikaz rezultata prve EMD iteracije

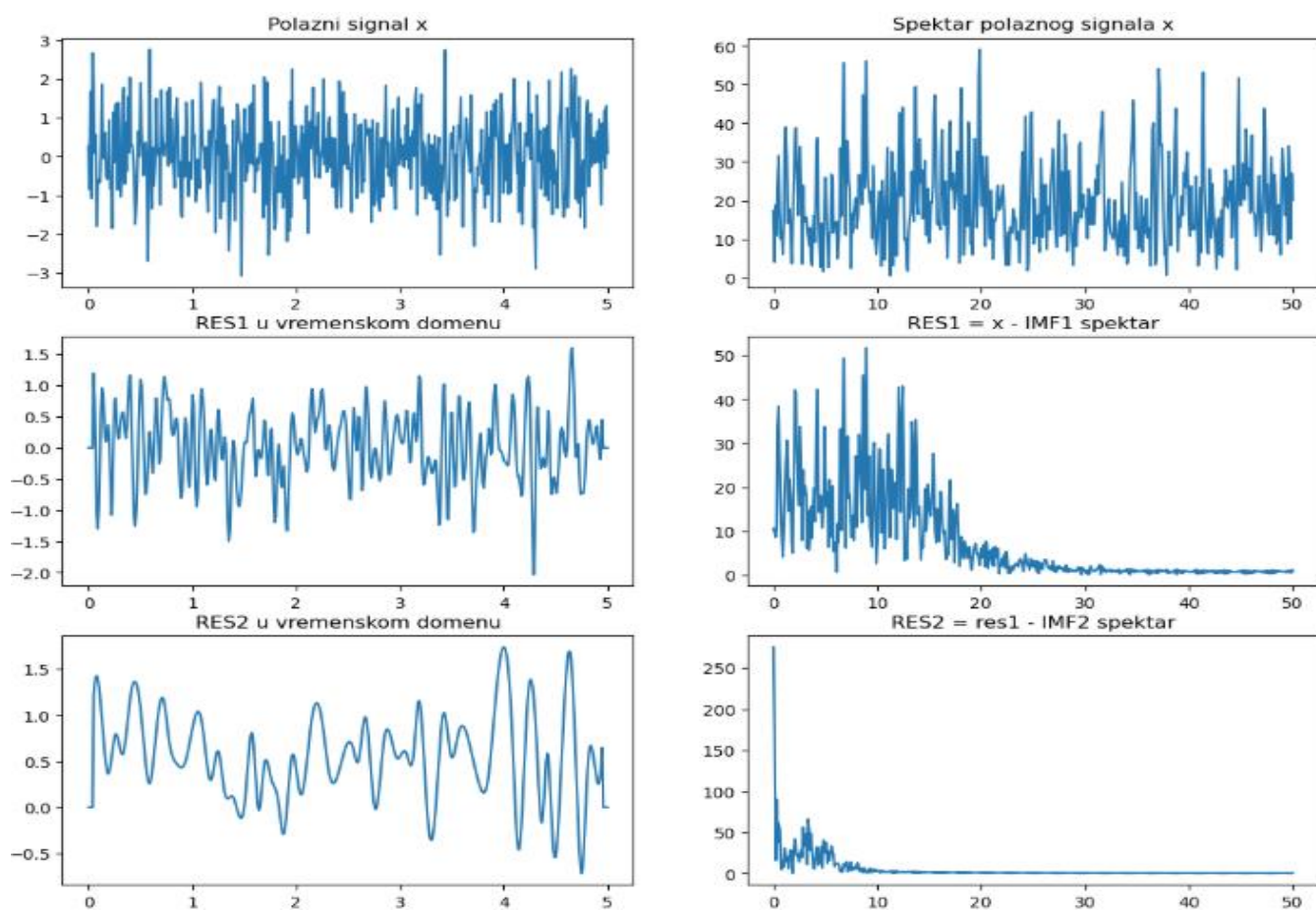
## Druga iteracija ( pronalazak IMF2 )

Rezidualni signal  $res1$  dobijen iz prve EMD iteracije (jednačina 2.3) se koristi kao polazni signal (ulaz) za drugu EMD iteraciju i pronalaženje IMF2 komponente. Tako da je sada polazni signal  $x$  u drugoj iteraciji zapravo  $res1$  signal.

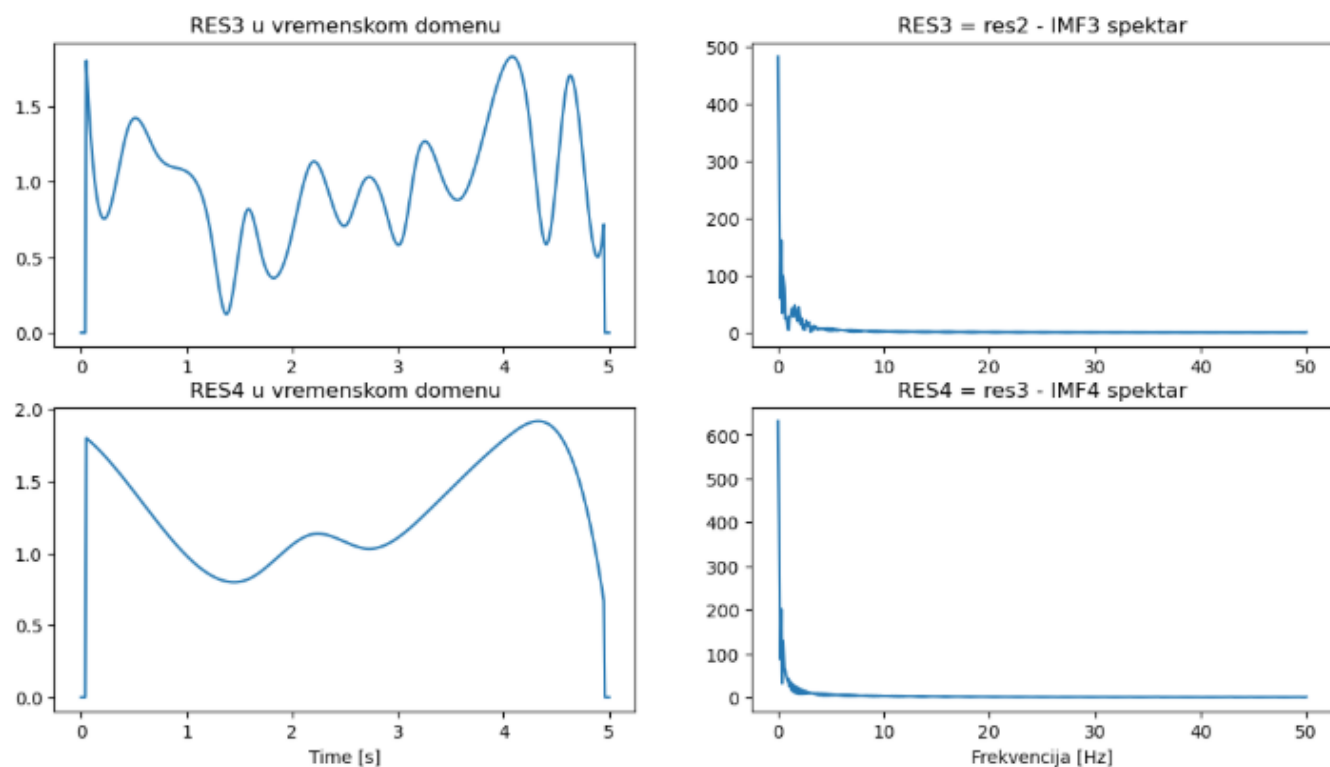
## Rezultat EMD algoritma

Proces se ponavlja sve dok rezidualni signal  $resX$  ne postane monotona funkcija bez lokalnih minimuma i maksimuma. Na kraju procesa, originalni polazni signal  $x$ , je dekomponovan u sumu IMF-ova i rezidualnog signala  $resX$  (jednačina 2.4)

$$x = resX + \sum IMF_i \quad (2.4)$$







U ovom konkretnom slučaju, jednačina 2.4 će izgledati

$$x = res4 + \sum_{i=1}^4 IMF_i$$

Algoritam je testiran i za 2D ulazne podatke u RGB formatu, date na slici 2.5



Slika 2.5. primjer ulaznih slika u boji



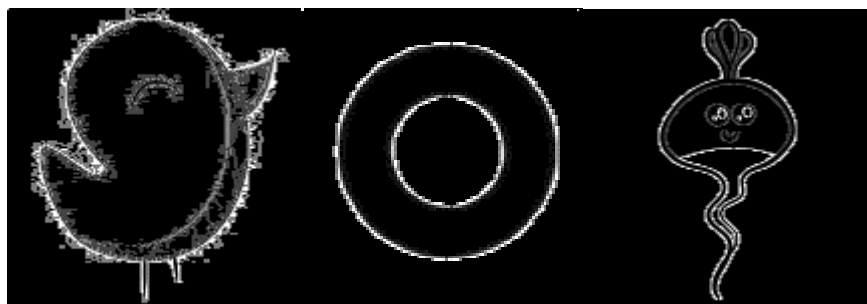
Slike su prvo prebačene u Gray-scale format, kao na slici 2.6



Slika 2.6.Gray-scale verzija ulaznih slika

Zatim je izvršena ekstrakcija samo prve IMF komponente EMD algoritmom. Svaka IMF komponenta sadrži najviše frekvencije signala iz prethodne iteracije, u našem slučaju to su najviše frekvencije Gray-scale verzije ulaznih slika (slika 2.6.). Kako prva IMF komponenta sadrži visoko-frekventni sadržaj polaznog Gray-scale signala, slika nastala na bazi IMF1 će isticati brze promjene intenziteta sive boje poput ivica.

Nova slika kreirana od IMF1 ima iste dimenzije kao originalna slika, ali će sadržavati samo one delove informacije koji odgovaraju visokofrekventnim komponentama. Što se tiče vizuelne interpretacije nove slike, ivice objekata i oštre promene će biti istaknute, dok će ravni, uniformni regioni slike (sa sporim promenama sive boje) tj homogene oblasti postati tamni ili neutralni. Rezultati su dati na slici 2.7



Slika 2.7. Slike nastale na bazi prve IMF komponente

U slučaju jednofokusiranih slika, kao što su slike 2.8 i 2.9,



Slika 2.8 Primjer single-focus slike



Slika 2.9 Primjer single-focus slike

nakon prethodne konverzije navedenih slika u Gray-scale, EMD algoritmom dolazimo do prve IMF komponente na osnovu koje kreiramo slike 2.10 i 2.11



Slika 2.10 Slika nastala na bazi IMF1 primjenom EMD nad slikom 2.8

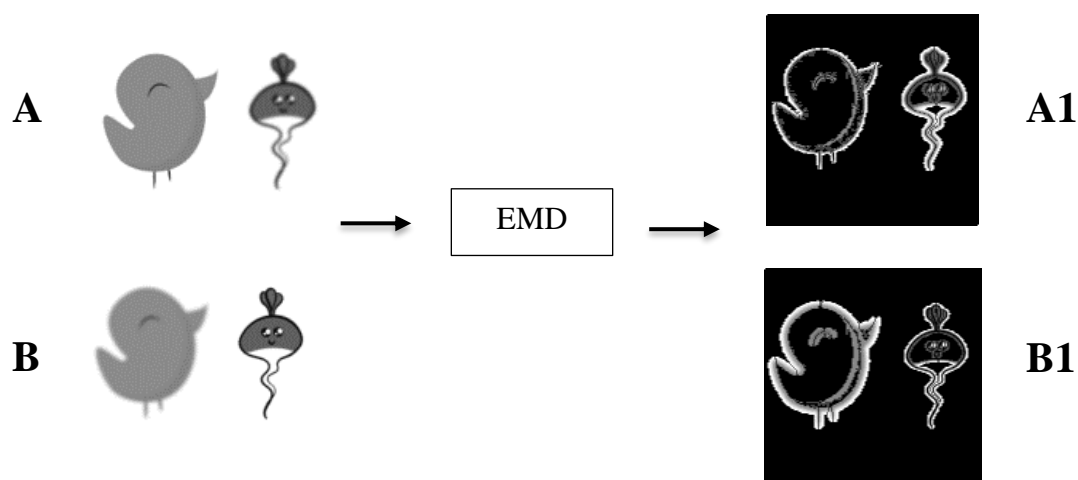


Slika 2.11 Slika nastala na bazi IMF1 primjenom EMD nad slikom 2.9

## Fuzija multifokusiranih slika

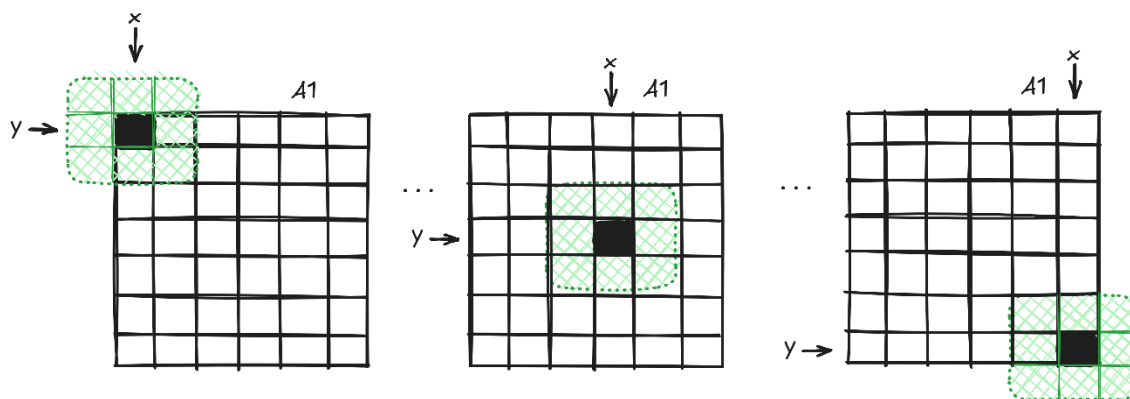
Kada fotografiramo scenu, dio slike je u fokusu, dok su drugi dijelovi zamućeni. Fuzija multifokusiranih slika kombinuje više fotografija iste scene, ali sa različitim dijelovima u fokusu, kako bi stvorila jednu oštru tj. potpuno fokusiranu sliku. Upravo je **Empirical Mode Decomposition** jedna od tehnika za fuziju slika.

Fuziju multifokusiranih slika tehnikom EMD možemo najjednostavnije objasniti na primjeru dvije polazne Gray-scale slike A i B sa različitim fokusima iste scene. Dakle slike A i B su Gray-scale jednofokusirane slike iste scene. Primjenom EMD algoritma nad slikama A i B dobijamo IMF1 za obe slike, na osnovu koga kreiramo 2 nove slike A1 i B1 koje sadrže visokofrekventni sadržaj polaznih A i B slika.



Proces je sljedeći:

1. **Racunanje lokalne varijanse  $var_{loc}$**  svakog piksela slike A1 i B1



- e. Prozor velicine  $N \times N$  se pomjera kroz sliku A1 na nacin da se posjeti svaki piksel slike
- f. Sabiraju se vrijednosti svih piksela koji su obuhvaceni prozorom

$$sum = \sum image\_pixels[i] \quad , \quad image\_pixels[i] \in Window \quad (2.5)$$

- g. Dobijena suma vrijednosti piksela obuhvacenih prozorom se dijeli sa brojem piksela koji su obuhvaceni prozorom cime dobijamo srednju vrijednost piksela u prozoru  $N \times N$

$$avg = \frac{sum}{n} \quad (2.6)$$

```
#define N 3          // Veličina prozora N×N za izračunavanje lokalne varijanse
#define E 2          // Prag odlucivanja
```

```
int offset = N / 2;

for (int y = 0; y < height; y++)
{
    for (int x = 0; x < width; x++)
    {
        /* -----
         *          Računanje srednje vrijednosti piksela u prozoru N×N
         * ----- */

        float mean = 0.0;
        unsigned int count = 0, sum = 0;

        // Prolazak kroz prozor
        for (int wy = -offset; wy <= offset; wy++)
        {
            for (int wx = -offset; wx <= offset; wx++)
            {
                int ny = y + wy;
                int nx = x + wx;

                // Provjera da li je piksel unutar granica slike
                if (ny >= 0 && ny < height && nx >= 0 && nx < width)
                {
                    int index = ny * width + nx;
                    sum += image[index];
                    count++;
                }
            }
        }
        mean = (float)sum/count;
    }
}
```

- h. Varijansu dobijamo kao prosjecno kvadratno odstupanje piksela u prozoru od srednje vrijednosti. Lokalna varijansa piksela  $var_{loc}$  se računa koristeći sve piksele koji su obuhvaćeni prozorom veličine N

$$var_{loc} = \frac{\sum (image\_pixels[i] - avg)^2}{n} \quad (2.7)$$

```

/* -----
 * Računanje varijanse kao prosjecno kvadratno odstupanje
 * piksela slike u prozoru NxN od srednje vrednosti
 * ----- */

float var = 0.0;

// Prolazak kroz prozor
for (int wy = -offset; wy <= offset; wy++)
{
    for (int wx = -offset; wx <= offset; wx++)
    {
        int ny = y + wy;
        int nx = x + wx;

        // Proveri da li je piksel unutar granica slike
        if (ny >= 0 && ny < height && nx >= 0 && nx < width)
        {
            int index = ny * width + nx;
            var += ( (float)image[index] - mean) * ((float)image[index] - mean);
        }
    }
}

// Rezultat se skladišti u 1D niz variance, gde je svaki element niza varijansa odgovarajućeg
// piksela slike
variance[y * width + x] = var / count;

```

## 2. Racunanje težinskih koeficijenata

- i. Porede se lokalne varijanse izmedju 2 slike A1 i B1 za svaki piksel
- j. Na osnovu razlike varijansi određuje težinski koeficijent  $\alpha(x,y)$  za sliku A1:

$$\alpha(x,y) = \begin{cases} 0 & var_{loc}\{A1(x,y)\} - var_{loc}\{B1(x,y)\} < -\varepsilon \\ 0,5 & var_{loc}\{A1(x,y)\} - var_{loc}\{B1(x,y)\} < \varepsilon \\ 1 & var_{loc}\{A1(x,y)\} - var_{loc}\{B1(x,y)\} > \varepsilon \end{cases} \quad (2.8)$$

$\alpha(x,y)$  je matrica koja predstavlja masku koja daje informaciju o mjestima dobre fokusiranosti na pojedinim slikama.

$\varepsilon > 0$  je prag odlucivanja.

- k. Nije potrebno računati težinske koeficijente za sliku B1 iz razloga sto težinski koeficijenti  $\alpha(x,y)$  za sliku A1 implicitno određuju i  $\beta(x,y)$

$$\beta(x,y) = 1 - \alpha(x,y) \quad (2.9)$$

```

void calculateWeights(float *varA1, float *varB1, float *weightsA, int N){
    for (int i = 0; i < N; i++)
    {
        if ( (varA1[i] - varB1[i]) < (-1.0*E) )
            weightsA[i] = 0.0;
        else if ( (varA1[i] - varB1[i]) < E )
            weightsA[i] = 0.5;
        else if ( (varA1[i] - varB1[i]) > E )
            weightsA[i] = 1.0;
    }
}

```

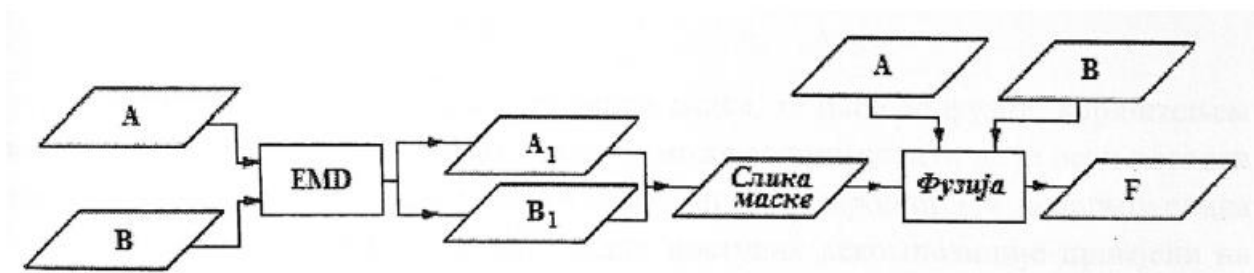
### 3. Konstrukcija fuzionisane slike

1. Za svaki piksel u fuzionisanoj slici  $F$  doprinosi dolaze iz:

- $\alpha(x, y) * A1(x, y)$
- $\beta(x, y) * B1(x, y) = (1 - \alpha(x, y)) * B1(x, y)$

Ako zelimo konstruisati finalnu fuzionisanu sliku  $F(x, y)$

$$F(x, y) = \alpha(x, y) * A1(x, y) + \beta(x, y) * B1(x, y) \quad (2.10)$$



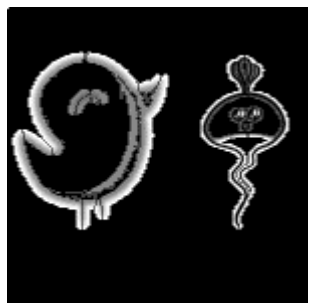
Slika 2.12 Blok dijagram algoritma za fuziju multifokusiranih slika A i B

```
for (int i = 0; i < N; i++)
    fused[i] = (unsigned char)(weightsA[i] * imf1_1[i] + (1.0 - weightsA[i]) * imf1_2[i]);
```

A1



B1



FUZIJA



Fused

### 3. Optimizacija

- Odsustvo `#pragma section("SECTION")` direktive

#### main.c

```
#include "input_data1.h"
#include "input_data2.h"
#include "emd.h"
#include "fusion.h"

static unsigned char imf1_1[N1];
static unsigned char imf1_2[N2];

int main(int argc, char *argv[]){
    float weightsA[N1] = {0};
    unsigned char fused[N1];
    ...
}
```

#### input\_data1.h

```
#define height1 150
#define width1 150
#define N1 22500

const unsigned char data1[N1] = { ... }
```

#### input\_data2.h

```
#define height2 150
#define width2 150
#define N2 22500

const unsigned char data2[N2] = { ... }
```

#### fusion.h

```
void calculateWeights( ... )
{
    float varA1[N1];
    float varB1[N2];
}
```

#### emd.h

```
void emd( ... )
{
    unsigned char localMAXs[N1];
    unsigned char localMINs[N1];
}
```

Nizovi `imf1_1`, `imf1_2`, `data1`, `data2` se nalaze u **uninitialized data(BSS)** i **initialized data** sekciji, dok su `varA1`, `varB1`, `localMAXs`, `localMINs`, `weightsA`, `fused` u **stack** sekciji. Sve ove sekcije se u memoriji nalaze na određenim mestima koje definiše linker skripta, a njihov tačan položaj zavisi od arhitekture, operativnog sistema i konfiguracije memorije.

**Stack** sekcija je smještena u

- `mem_block1_dm32` ( 50KB za `block_1` ali 2KB za **stack** )

a mi već sa 2 lokalna niza (~50KB) premasujemo dostupnu memoriju. Pritom se i dio **bss** sekcije nalazi u ovom bloku `block1`, a broj varijabli i nizova koji se smjestaju u `block1` je mnogo veći od ova 2 lokalna niza.



**Data (uninitialized + initialized)** sekcija je smjestena u

- mem\_block1\_dm32 ( ukupno za blok1 49 151B )
- mem\_block2\_pm32 ( ukupno za blok2 32 767B )
- mem\_block3\_dm32 ( ukupno za blok3 32 767B )

[Error li1040] Out of memory in output section 'dx\_block1\_dm\_data\_prio3'

[Error li1040] Out of memory in output section 'dx\_block3\_dm\_data\_prio3'

[Error li1040] Out of memory in output section 'dx\_block2\_overflow\_data'

- Upotreba **#pragma section("SECTION")** direktive

```
#include "emd.h"
#include "fusion.h"
#include "preprocessing.h"
#include "input_data1.h"
#include "input_data2.h"

#pragma section("seg_block1")
static unsigned char imf1_1[N1];

#pragma section("seg_block2")
static unsigned char imf1_2[N2];

int main(int argc, char *argv[])
{
    ...
}
```

#### input\_data1.h

```
#define height1 150
#define width1 150
#define N1 22500

#pragma section("seg_sram")
const unsigned char data1[N1] = { ... }
```

#### input\_data2.h

```
#define height2 150
#define width2 150
#define N2 22500

#pragma section("seg_sram")
const unsigned char data2[N2] = { ... }
```

Linkerska skripta:

```
dx_seg_block1
{
    INPUT_SECTIONS( $OBJECTS(seg_block1) )
} > mem_block1_dm32

dx_seg_block2
{
    INPUT_SECTIONS( $OBJECTS(seg_block2) )
} > mem_block2_pm32
```

Uz upotrebu pretprocesorske direktive za smještanje podataka u zeljene memorijske sekcije ADSP procesora, nemamo više problema sa nedostatkom prostora jer smo rasporedili nizove podataka tako da ne dodje do prekoračenja niti jedne sekcije. Program uspješno prolazi build.

## - Stack overflow

```
int main(int argc, char *argv[])
{
    float weightsA[N1] = {0};
    . . .
}
```

Stack je smješten unutar **mem\_block1\_dm32** i zauzima oko 2KB, te je očigledno doslo do njegovog prepunjavanja (weightsA ~ 22KB)

```
dx_block1_dm_data_prio0
{
    RESERVE(heap_and_system_stack_in_Internal, heap_and_system_stack_in_Internal_length = 2048, 2)
    INPUT_SECTIONS( $OBJ_LIBS(seg_int_data) )
} > mem_block1_dm32

dx_block1_stack NO_INIT
{
    RESERVE_EXPAND(heap_and_system_stack_in_Internal, heap_and_system_stack_in_Internal_length, 0, 2)
    ldf_stack_space = heap_and_system_stack_in_Internal;
    ldf_stack_end = (ldf_stack_space + (heap_and_system_stack_in_Internal_length - 2)) & 0xffffffe;
    ldf_stack_length = ldf_stack_end - ldf_stack_space;
} > mem_block1_dm32
```

Rjesenje je da ovaj lokalni niz definisemo kao staticki ili globalni te da mu pretprocesorskom direktivom odredimo segment memorije u koji ce biti smjesten bez problema.

```
#pragma section("seg_block1")
float weightsA[N1];

int main(int argc, char *argv[])
{
    . . .
}
```

- Smanjenje broja procesorskih ciklusa utrošenih na izvršavanje algoritma

```
#pragma section("seg_flash")
float varA1[N];

#pragma section("seg_flash")
float varB1[N];

void calculateWeights(unsigned char* imf1_1, unsigned char* imf1_2, float *weightsA, int size, int
width, int height)
{
    calculate_local_variance(imf1_1, varA1, width, height);
    calculate_local_variance(imf1_2, varB1, width, height);

    for (int i = 0; i < size; i++)
    {
        if ( (varA1[i] - varB1[i]) < (-1.0*E) )
            weightsA[i] = 0.0;
        else if ( (varA1[i] - varB1[i]) < E )
            weightsA[i] = 0.5;
        else if ( (varA1[i] - varB1[i]) > E )
            weightsA[i] = 1.0;
    }
}

Broj ciklusa: 81 867 151
```

<pre>#pragma section("seg_block3") float varA1[N];  #pragma section("seg_block1") float varB1[N];  void calculateWeights (...) {~}  Broj ciklusa: 58 502 113</pre>	↓	<pre>dxs_seg_block3 {     INPUT_SECTIONS( \$OBJECTS(seg_block3) ) } &gt; mem_block3_dm32  dxs_seg_block1 {     INPUT_SECTIONS( \$OBJECTS(seg_block1) ) } &gt; mem_block1_dm32</pre>
--	---	---

Algoritam – funkcija `calculateWeights` je ostala identična, tako da je jedina promjena ovdje mjesto skladištenja nizova u memoriji. Mnogo je efikasniji i brzi pristup memorijskim lokacijama blize procesora nego flesh memoriji.

```
#pragma section("seg_block3")      #pragma section("seg_block1")
float varA1[N];                    float varB1[N];

void calculateWeights(unsigned char* imf1_1, unsigned char* imf1_2, float *weightsA, int size, int
width, int height)
{
    calculate_local_variance(imf1_1, varA1, width, height);
    calculate_local_variance(imf1_2, varB1, width, height);

    for (int i = 0; i < size; i++)
    {
        if ( (varA1[i] - varB1[i]) < (-1.0*E) )
            weightsA[i] = 0.0;
        else if ( (varA1[i] - varB1[i]) < E )
            weightsA[i] = 0.5;
        else if ( (varA1[i] - varB1[i]) > E )
            weightsA[i] = 1.0;
    }
}
```

Broj ciklusa: 58 502 113

```
#pragma section("seg_block3")      #pragma section("seg_block1")
float varA1[N];                    float varB1[N];

void calculateWeights(unsigned char* imf1_1, unsigned char* imf1_2, float *weightsA, int size, int
width, int height)
{
    calculate_local_variance(imf1_1, varA1, width, height);
    calculate_local_variance(imf1_2, varB1, width, height);

    for (int i = size-1; i >= 0 ; i--)
    {
        if ( (varA1[i] - varB1[i]) < (-1.0*E) )
            weightsA[i] = 0.0;
        else if ( (varA1[i] - varB1[i]) < E )
            weightsA[i] = 0.5;
        else if ( (varA1[i] - varB1[i]) > E )
            weightsA[i] = 1.0;
    }
}
```

Broj ciklusa: 58 462 264

Za `for (int i = 0; i < size; i++)` imamo **Broj ciklusa: 58 502 113**  
a za `for (int i = size - 1; i >= 0 ; i--)` imamo **Broj ciklusa: 58 462 264**

Svako poredjenje se svodi na poredjenje sa nulom, te je efikasnije da eksplicitno poredimo sa nulom nego da pustimo kompajler da sam svede operaciju poredjenja na poredjenje sa nulom.

```
void findExtremes(const unsigned char *data)
{
    for (int i = 1; i < N - 1; i++)
    {
        if (data[i] >= data[i - 1] && data[i] >= data[i + 1])
            localMAXs[i] = data[i];

        if (data[i] <= data[i - 1] && data[i] <= data[i + 1])
            localMINs[i] = data[i];
    }
}
```

Broj ciklusa: 16 846 988

```
inline void findExtremes(const unsigned char *data)
{
    for (int i = 1; i < N - 1; i++)
    {
        if (data[i] >= data[i - 1] && data[i] >= data[i + 1])
            localMAXs[i] = data[i];

        if (data[i] <= data[i - 1] && data[i] <= data[i + 1])
            localMINs[i] = data[i];
    }
}
```

Broj ciklusa: 16 846 988

Jednostavne funkcije koje

- sadrže samo nekoliko linija koda
- sadrže jednostavne matematičke operacije
- imaju mnogo poziva u različitim dijelovima programa

njihova deklaracija kao **inline** može eliminirati trosak funkcijskog poziva i dati kompajleru uvid u njihov kod na mjestu poziva te se na taj način može i bolje optimizovati.

findExtremes je najjednostavnija funkcija programa, ali nije dovoljno jednostavna da bi bila proglašena za **inline**. Takođe, findExtremes funkcija se ne poziva dovoljno te trošak optimizacije inline-a ne opravdava dodatni prostor u binarnom fajlu.

Za funkcije koje

- koriste rekurzije, petlje, složene izraze
- se ne pozivaju dovoljno često

nije pametno koristiti **inline** jer se može značajno povećati veličina binarnog fajla bez rezultata u performansama.

```
void findExtremes(const unsigned char *data)
{
    for (int i = 1; i < N - 1; i++)
    {
        if (data[i] >= data[i - 1] && data[i] >= data[i + 1])
            localMAXs[i] = data[i];

        if (data[i] <= data[i - 1] && data[i] <= data[i + 1])
            localMINs[i] = data[i];
    }
}
```

Broj ciklusa: 2 029 553

▼

```
void findExtremes(const unsigned char *data)
{
    for (int i = N - 2; i >= 1; i--)
    {
        if (data[i] >= data[i - 1] && data[i] >= data[i + 1])
            localMAXs[i] = data[i];

        if (data[i] <= data[i - 1] && data[i] <= data[i + 1])
            localMINs[i] = data[i];
    }
}
```

Broj ciklusa: 2 029 437

▼

```
void findExtremes(const unsigned char *data)
{
    unsigned char isLocalMax;
    unsigned char isLocalMin;

    for (int i = N - 2; i >= 1; i--)
    {
        isLocalMax = (data[i] >= data[i - 1]) & (data[i] >= data[i + 1]);
        isLocalMin = (data[i] <= data[i - 1]) & (data[i] <= data[i + 1]);

        localMAXs[i] = data[i] * isLocalMax;
        localMINs[i] = data[i] * isLocalMin;
    }
}
```

Broj ciklusa: 1 979 837

Uslovi u petlji trebaju da se izbjegavaju jer uslovi u petlji, cak i ako imaju predikciju grananja (expected true/false) se vise-manje svode na situaciju sa pozivom funkcije u petlji samo sa manjim posljedicama jer nemamo kreiranje stek okvira. U situaciji gdje imamo uslove unutar for petlji, pipeline pati tj ne moze se u potpunosti ispuniti.

```
int isIMF(unsigned char *imf)
{
    int zeroCrossings = 0;
    int extremaCount = 0;

    unsigned char prevSign = imf[0] & 0x80;
    unsigned char currentSign;

    for (int i = 1; i < N; i++)
    {
        currentSign = imf[i] & 0x80;

        if (currentSign != prevSign)
        {
            zeroCrossings++;
            prevSign = currentSign;
        }

        if ( ((signed char)imf[i] > (signed char)imf[i - 1] && (signed char)imf[i] > (signed char)imf[i + 1]) ||
            ((signed char)imf[i] < (signed char)imf[i - 1] && (signed char)imf[i] < (signed char)imf[i + 1]))
            extremaCount++;
    }
    . . .
}
```

Broj ciklusa: 1 137 752

▼

```
int isIMF(unsigned char *imf)
{
    int zeroCrossings = 0;
    int extremaCount = 0;

    unsigned char prevSign = imf[0] & 0x80;
    unsigned char currentSign;

    for (int i = 1; i < N; i++)
    {
        currentSign = imf[i] & 0x80;

        zeroCrossings += ( (currentSign - prevSign) != 0);
        prevSign = currentSign;

        if ( ((signed char)imf[i] > (signed char)imf[i - 1] && (signed char)imf[i] > (signed char)imf[i + 1]) ||
            ((signed char)imf[i] < (signed char)imf[i - 1] && (signed char)imf[i] < (signed char)imf[i + 1]))
            extremaCount++;
    }
    . . .
}
```


Broj ciklusa: 1 064 575

Zamjena uslovnog grananja logickim i aritmetickim operacijama je smanjila broj utrosenih procesorskih ciklusa za vise od 70K.



```
void calculateWeights(...)
{
    calculate_local_variance(imf1_1, varA1, width, height);
    calculate_local_variance(imf1_2, varB1, width, height);

    for (int i = size-1; i >= 0 ; i--)
    {
        if ( (varA1[i] - varB1[i]) < (-1.0*E) )
            weightsA[i] = 0.0;
        else if ( (varA1[i] - varB1[i]) < E )
            weightsA[i] = 0.5;
        else if ( (varA1[i] - varB1[i]) > E )
            weightsA[i] = 1.0;
    }
}
Broj ciklusa: 1 309 252
```



```
void calculateWeights(...)
{
    calculate_local_variance(imf1_1, varA1, width, height);
    calculate_local_variance(imf1_2, varB1, width, height);

    float diff = 0;

    for (int i = size-1; i >= 0 ; i--)
    {
        diff = varA1[i] - varB1[i];

        weightsA[i] = 0.0 + ((diff - E) < 0)*0.5 + ((diff - E) > 0)*1.0;
    }
}
Broj ciklusa: 1 170 012
```

```
void convolve(bmp_header_t* bmp_header, unsigned char* data, unsigned int* var)
{
    unsigned int susjed1, susjed2, susjed3, susjed4, susjed5, susjed6, susjed7, susjed8, i = 0, sum = 0;
    float avg = 0;

    for (int y = 1; y < bmp_header->height - 1; y++)
    {
        for (int x = 1; x < bmp_header->width - 1; x++)
        {
            sum = data[susjed1] * kernel[0][0] + data[susjed2] * kernel[0][1] + data[susjed3] * kernel[0][2] +
                  data[susjed4] * kernel[1][0] + data[i] * kernel[1][1] + data[susjed5] * kernel[1][2] +
                  data[susjed6] * kernel[2][0] + data[susjed7] * kernel[2][1] + data[susjed8] * kernel[2][2];

            avg = sum/9.0;

            var[i] = ((data[susjed1] - avg)*((data[susjed1] - avg)) +
                     (data[susjed2] - avg)*((data[susjed2] - avg)) +
                     (data[susjed7] - avg)*((data[susjed7] - avg)) +
                     (data[susjed8] - avg)*((data[susjed8] - avg)))/9.0;
        }
    }
}
Broj ciklusa: 3 643 710
```

```
void convolve(bmp_header_t* bmp_header, unsigned char* data, unsigned int* var)
{
    unsigned int susjed1, susjed2, susjed3, susjed4, susjed5, susjed6, susjed7, susjed8, i = 0, sum = 0;
    unsigned int avg = 0;
    fract f = 0.1ur;

    for (int y = 1; y < bmp_header->height - 1; y++)
    {
        for (int x = 1; x < bmp_header->width - 1; x++)
        {
            sum = data[susjed1] * kernel[0][0] + data[susjed2] * kernel[0][1] + data[susjed3] * kernel[0][2] +
                  data[susjed4] * kernel[1][0] + data[i] * kernel[1][1] + data[susjed5] * kernel[1][2] +
                  data[susjed6] * kernel[2][0] + data[susjed7] * kernel[2][1] + data[susjed8] * kernel[2][2];

            avg = muliur(sum, f);

            var[i] = muliur(((data[susjed1] - avg)*((data[susjed1] - avg)) +
                             (data[susjed8] - avg)*((data[susjed8] - avg))), f);
        }
    }
}
Broj ciklusa: 3 123 710
```

SHARC procesor ima hardversku podršku za aritmetiku fiksnog zareza. Deklarisali smo promjenjivu sa fiksnim zarezom  $fract f = 1/9 = 0.1$ . Funkcija *muliur* omogućava množenje podatka tipa int podatkom tipa unsigned fract i kao rezultat dobija se podatak tipa int.

Sve što je implemetirano u hardveru daje na brzini. Aritmetika pokretnog zareza treba da se izbjegava pogotovo u slučaju kada procesor nema floating-point jedinicu (hardverski sklop za FP instrukcije) jer tada CPU emulira FPU jedinicu što je softverska implementacija koja dodatno troši CPU cikluse. Zbog činjenice da imamo hardversku podršku za aritmetiku fiksnog zareza, dobijamo bolje rezultate kada umjesto floating-point aritmetike koristimo fixnu aritmetiku.

```
void convolve(bmp_header_t* bmp_header, unsigned char* data, unsigned int* var)
{
    unsigned int susjed1, susjed2, susjed3, susjed4, susjed5, susjed6, susjed7, susjed8, i = 0, sum = 0;
    unsigned int avg = 0;
    fract f = 0.1ur;

    for (int y = 1; y < bmp_header->height - 1; y++)
    {
        for (int x = 1; x < bmp_header->width - 1; x++)
        {
            i = (y * bmp_header->width) + x;
            susjed1 = (y-1)* bmp_header->width + (x-1);
            ~
            susjed8 = (y+1)* bmp_header->width + (x+1);
            ~
        }
    }
}
```

Broj ciklusa: 3 123 709

```
void convolve(bmp_header_t* bmp_header, unsigned char* data, unsigned int* var)
{
    unsigned int susjed1, susjed2, susjed3, susjed4, susjed5, susjed6, susjed7, susjed8, i = 0, sum = 0;
    unsigned int avg = 0;
    fract f = 0.1ur;

    int width = bmp_header->width;
    int height = bmp_header->height;

    for (int y = 1; y < height - 1; y++)
    {
        for (int x = 1; x < width - 1; x++)
        {
            i = (y * width) + x;
            susjed1 = (y-1)* width + (x-1);
            ~
            susjed8 = (y+1)* width + (x+1);
            ~
        }
    }
}
```

Broj ciklusa: 2 895 431

Kompajleru je jednostavnije da varijablu za kontrolu petlje smjesti u registar tokom izvršavanja petlje kada je ona lokalna promjenjiva, nego kada se radi o globalnoj promjenjivoj.

Takodje, kada koristimo globalnu varijablu za kontrolu petlje, ta varijabla mora biti ponovno učitana za svaku iteraciju petlje jer kompajler ponekad ne može biti siguran da li je u međuvremenu došlo do promjene vrijednosti globalne promjenjive.

U ovom konkretnom slučaju, ne samo da se kontrola petlje vrši lokalnom varijablom, već je i višestruki pristup vrijednosti mnogo brži kada se ta vrijednost nalazi unutar istog konteksta, nego kada moramo da skacemo sa jedne lokacije u programu na drugu.

```
for(int i = 0; i < height ; i++)
{
    for (int j = 0; j < paddedRowSize; j++)
    {
        if (j+3 <= unpaddedRowSize)
        {
            b1 = fgetc(in); j++;
            b2 = fgetc(in); j++;
            b3 = fgetc(in);
            unsigned char gray = ceil(b1*0.3 + b2*0.59 + b3*0.11);
            pixels[index++] = gray;
        }
        else
            fgetc(in); // Skip padding
    }
}
```

Broj ciklusa: 5 018 915

```
for(int i = 0; i < height ; i++)
{
    for (int j = 0; j < paddedRowSize; j++)
    {
        if (j+3 <= unpaddedRowSize)
        {
            b1 = fgetc(in); j++;
            b2 = fgetc(in); j++;
            b3 = fgetc(in);
            unsigned char gray = muliur(b1 , 0.3ur) + muliur(b2 , 0.59ur) + muliur(b3 , 0.11ur);
            pixels[index++] = gray;
        }
        else
            fgetc(in); // Skip padding
    }
}
```

Broj ciklusa: 4 750 691

Floating point aritmetika je zamijenjena aritmetikom fiksnog zareza jer SHARC procesor ima hardversku podršku za aritmetiku fiksnog zareza, a sve što je hardverski implementirano daje na brzini izvršavanja.


```
for(int i = 0; i < height ; i++)
{
    for (int j = 0; j < paddedRowSize; j++)
    {
        if (expected_true(j+3 <= unpaddedRowSize))
        {
            b1 = fgetc(in); j++;
            b2 = fgetc(in); j++;
            b3 = fgetc(in);
            unsigned char gray = muliur(b1 , 0.3ur) + muliur(b2 , 0.59ur) + muliur(b3 , 0.11ur);
            pixels[index++] = gray;
        }
        else
            fgetc(in); // Skip padding
    }
}
```

Broj ciklusa: 4 750 691

```
unsigned int tStep = 10;

for (int i = 0; i < N-1; i++)
{
    ~
    for (int j = 0; j < tStep; j++)
    {
        float t = (float)j / tStep;
        localMAXs[i * tStep + j] = (a0 * t * t * t + a1 * t * t + a2 * t + a3);
        localMINs[i * tStep + j] = (b0 * t * t * t + b1 * t * t + b2 * t + b3);
    }
}
```

Broj ciklusa: 27 323 792



```
unsigned int tStep = 10;

for (int i = 0; i < N-1; i++)
{
    for (int j = 0; j < tStep; j++)
    {
        unsigned fract t = urdivi(j , tStep);

        localMAXs[i * tStep + j] = muliur(a0 , t * t * t) + muliur(a1 , t * t) + muliur(a2 , t) + a3;
        localMINs[i * tStep + j] = muliur(b0 , t * t * t) + muliur(b1 , t * t) + muliur(b2 , t) + b3;
    }
}
```

Broj ciklusa: 154 588 394

```
for (int i = 0; i < N ; i--)  
    data[i] = (unsigned char)(data[i] - ((localMAXs[i] + localMINs[i]) / 2) ); // IMF1
```

Broj ciklusa: 1 106 434

```
for (int i = 0; i < N ; i--)  
    data[i] = (unsigned char)(data[i] - ((localMAXs[i] + localMINs[i]) >> 2) ); // IMF1
```

Broj ciklusa: 1 106 434

```
float weightsA[30000];  
  
for (int i = N-1; i >= 0 ; i--)  
{  
    int diff = localMAXs[i] - localMINs[i];  
  
    weightsA[i] = 0.0 + ((diff - E) < 0)*0.5 + ((diff - E) > 0)*1.0;  
}  
  
for(int i = 0; i < N; i++)  
    data1[i] = (unsigned char)(weightsA[i] * data1[i] + (1.0 - weightsA[i]) * data2[i]);
```

Broj ciklusa: 2 849 902

```
unsigned char weightsA[30000];  
  
for (int i = N-1; i >= 0 ; i--)  
{  
    int diff = localMAXs[i] - localMINs[i];  
  
    weightsA[i] = 0 + ((diff - E) > 0);  
}  
  
for(int i = 0; i < N; i++)  
    data1[i] = weightsA[i]*data1[i] + (1 - weightsA[i])*data2[i] ;
```

Broj ciklusa: 2 414 621

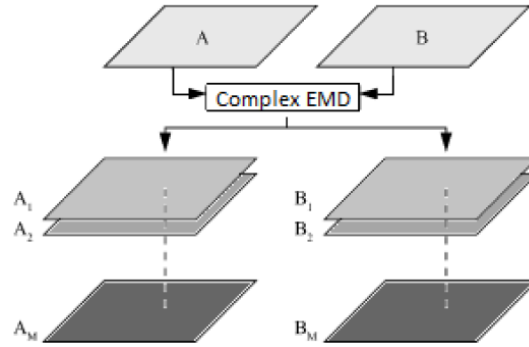
Procesor mnogo efikasnije barata sa cjelobrojnim vrijednostima nego sa floating-point vrijednostima iako ADSP-214xx SHARC® Processor ima hardversku podrsku za FP aritmetiku.

## 4. Zaključak

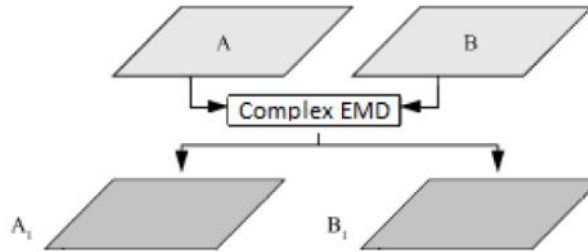
EMD algoritam razlaže polazni signal  $x$  na IMF komponente i reziduum kao

$$x = \sum_{i=1}^m IMF_i + r_m \quad (4.1)$$

Ukoliko od izdvojenih IMF komponenata kreiramo po sliku, dobit ćemo  $m$  novih slika  $A_1, \dots, A_m$ . Najveći dio korisnih informacija se zapravo nalazi u prvoj IMF komponenti, tj. slika  $A_1$  je ta koja sa sobom nosi najznačajnije informacije kada je riječ o oblastima dobrog fokusa. Dakle, postoje IMF1 sadrži informacije o najvišim frekvencijama signala  $x$ , a upravo je to ono što nam treba za analizu fokusiranih oblasti, nije potrebno da računamo sve IMF komponente već samo prvu.



Slika 4.1. Cjelovita EMD dekompozicija dvije slike A i B vodjena jednačinom 4.1



Slika 4.2. EMD dekompozicija prvog nivoa dvije slike A i B – izdvajanje samo prve IMF komponente

Na ovaj način smo ubrzali algoritam jer nema potrebe za računanjem svih IMF komponenata i težinskih koeficijenata  $var_{loc}\{Ai(x, y)\} - var_{loc}\{Bi(x, y)\}, i > 1$ .