Applications of digital processors

DIGITAL FILTERS

on signal processors

Author: Grzegorz Szwoch

Gdańsk University of Technology, Department of Multimedia Systems

Introduction

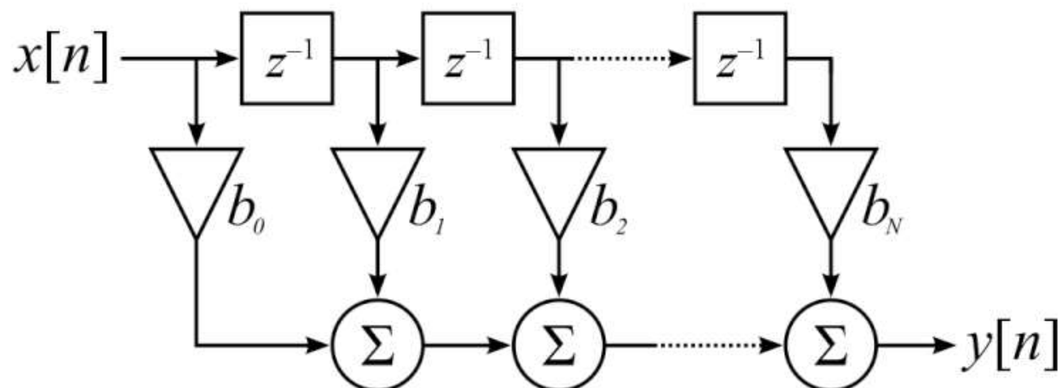
- Digital filters are basic algorithms that are implemented on digital signal processors.
- Two types of digital filters are used:
  - finite impulse response – FIR (“convolutional”),
  - infinite impulse response – IIR (recursive).
- Both types are used in practice.
- Theory on FIR and IIR filters was presented in the *Sound and image processing* course lecture and it will not be repeated here.

FIR filters

Digital finite impulse response (FIR) filters:

- the latest *N* signal samples (*x*) are multiplied by the filter coefficients (*b*),
- the results of multiplication are summed up and sent to the filter output (*y*).

$$y(n) = b_0 x(n) + b_1 x(n-1) + b_2 x(n-2) + \ldots + b_N x(n-N)$$



FIR filter design

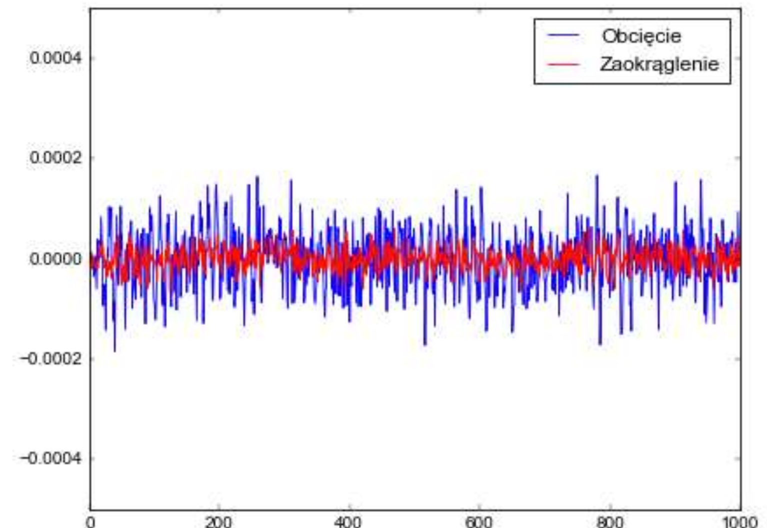
- We decide on the filter response (e.g. lowpass), cut-off frequency (e.g. 4 kHz), minimum attenuation (e.g. 40 dB), width of the transition band (or filter length).
- We compute (floating point) filter coefficients using software (Matlab, Python/SciPy), using one of available methods (windowing, Parks-McClellan).
- We check the filter gain in the passband – it should be close to one. We normalize the filter gain if needed (for lowpass filters: we divide the coefficients by the sum of all coefficients).

Filter coefficients on a fixed-point DSP

- The filter coefficients are calculated as floating-point numbers.
- If we use a fixed-point DSP, the coefficients must be quantized and written in Q15 format.
- We multiply the coefficients by 32768 and round them.
- We need to check the gain. If the gain exceeds the range (for low-pass filter: sum of coefficients exceeds 32767), we need to reduce the gain. For example, we can multiply the coefficients by a lower number, e.g. 32760.

Quantization of coefficients

- When we convert the coefficients to Q15, we perform quantization – we use the nearest value that has a representation in the Q15 format.
- A difference between real and quantized values has a form of a quantization noise.
- We get slightly different filter than the designed one.
- In floating-point DSPs, the quantization effect is still present, but it has much lower influence on the results.



Filter coefficients in C code

- Coefficients are written in a *const* array.
- The array is declared globally (outside of functions).
- If we have several filters, a good practice is declaring their coefficients in a separate file that is #included.
- Example for a fixed-point DSP (the number of coefficients is #defined as a number *N* for convenience):

```
#define N 30
const int lp_filter[] = {-4, 39, 97, 149, 133, -23, -337, -701,
-883, -595, 360, 1955, 3883, 5634, 6677, 6677, 5634, 3883, 1955,
360, -595, -883, -701, -337, -23, 133, 149, 97, 39, -4};
```

FIR filter implementation

In practice, we use optimized filter implementations provided by the DSP maker. We will write our own FIR algorithm in C, for demonstration purposes.

- We need to store the latest *N* signal samples in memory.
- We use a circular buffer.
- The buffer is declared globally.

```
int filter_buffer[N];
```

- We need an index (a global variable) that points at the write position (the oldest sample) in the buffer:

```
int buffer_index = 0;
```

FIR filter implementation

- We need to fill the filter buffer with zeros:

```
int i;
for (i = 0; i < N; i++)
    filter_buffer[i] = 0;
```

- We write a new signal sample into the buffer.
- We need to iterate over the buffer, multiply the samples by the filter coefficients and sum up the results.
- We need a second index (*pos*) to read from the buffer.
- We move the index with _*circ_incr* function.
- Multiplication and accumulation is performed with _*smac*.

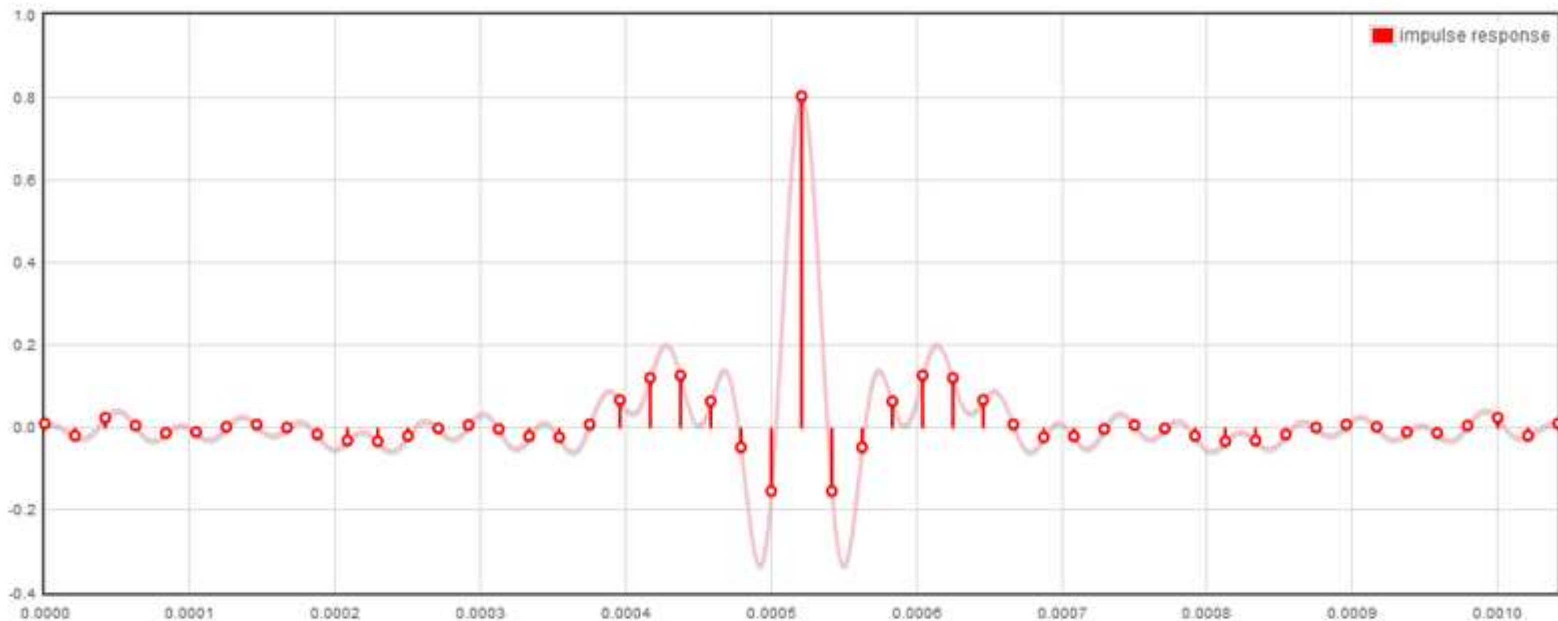
FIR filter implementation

```
// x: the current signal sample (int)
filter_buffer[buffer_index] = x;
int pos = buffer_index; // index of buffer read
long y = 0;  // y: filter result

// loop over the coefficients and samples
for (i = 0; i < N; i++) {
    // y = y + x[n]*h[n]
    y = _smac(y, filter_buffer[pos], lp_filter[i]);
    pos = _circ_incr(pos, -1, N); // pos = pos - 1
}

// result: a sample sent to the filter output (int)
result = (int)(_sround(y) >> 16);
// update the buffer write index (b_i = b_i + 1)
buffer_index = _circ_incr(buffer_index, 1, N);
```

FIR filter symmetry

- Usually, we design linear phase FIR filters, their impulse response (coefficients) is symmetric:
  - odd length filters (type I) – two symmetrical halves and one unpaired central coefficient,
  - even length filters (type II, only low-pass and band-stop): two symmetrical halves.
- We can use the symmetry to reduce the number of multiplications: we add two samples and multiply the result by one coefficient.
- Only the non-repeating part of the coefficients must be stored in the memory.

FIR filter symmetry




```
const int lp_filter[] = {15, 58, 109, 135, 68, -151, -493, -796,
-788, -201, 1076, 2884, 4798, 6263, 6812, 6263, 4798, 2884,
1076, -201, -788, -796, -493, -151, 68, 135, 109, 58, 15};
```

```
const int lp_filter[] = {15, 58, 109, 135, 68, -151, -493, -796,
-788, -201, 1076, 2884, 4798, 6263, 6812};
```

Range overflow problem

- When we add two samples in Q15, range overflow may occur:

```
int sumsamp = _sadd(buffer[pos1], buffer[pos2]);
```

- We can divide the samples by two (>>1) before adding:

```
int sumsamp = _sadd(buffer[pos1]>>1, buffer[pos2]>>1);
```

- We must compensate the final result (<<1).
- We lose one bit of precision (gaining computation speed).

Implementation of a symmetric FIR filter

```
// we need two buffer read indices
int pos1 = buffer_index;                        // the latest sample
int pos2 = _circ_incr(pos1, 1, N);              // the oldest sample
int sumsamp;
long result = 0;

for (i = 0; i < (N>>1); i++) {
     sumsamp = _sadd(buffer[pos1]>>1, buffer[pos2]>>1);
     result = _smac(result, sumsamp, lp_filter[i]);
     pos1 = _circ_incr(pos1, -1, N);            // index goes back
     pos2 = _circ_incr(pos2,  1, N);            // index goes forward
}
// the middle coefficient (if N is odd)
if (N & 1)                                      // N is odd
     result = _smac(result, buffer[pos1]>>1, lp_filter[i]);

// here we compensate the division by 2:
output = (int)(_sround(result<<1) >> 16);
buffer_index = _circ_incr(buffer_index, 1, N);
```

FIR filters in DSPLIB

- DSPLIB contains optimized filtering algorithms for DSPs from Texas Instruments. For fixed-point DSPs: in Assembler.
- The most important FIR functions (DSPLIB for C55x):
  - *fir* – standard implementation,
  - *fir2* – optimized for *dual MAC* mode, requires specific buffer alignment in memory,
  - *firs* – for even-length symmetric filters.
- Separate functions for special filters (Hilbert, decimation, interpolation, ladder).

fir function in DSPLIB

- The documentation describes the function header:

```
ushort oflag = fir (DATA *x, DATA *h, DATA *r, DATA *dbuffer, ushort nx,
ushort nh)
```

- *DATA* is an alias of *int* (16-bit).
- Asterisk (*) denotes a pointer:
  - for arrays (buffers): their name is a pointer,
  - for "scalar" variables, we must take the pointer by putting & before the variable name.
- The circular buffer must be created in global memory and filled with zeros. It cannot be modified in the program!

dbuffer[nh+2]	Pointer to delay buffer of length nh = nh + 2

- The returned *oflag* is 1 if the overflow occurred.

fir function in DSPLIB

```
ushort oflag = fir (DATA *x, DATA *h, DATA *r, DATA *dbuffer, ushort nx,
ushort nh)
```

- *x* – pointer to input samples. For one sample: e.g. `&input`
- *h* – pointer to filter coefficients vector; we need to cast it to (DATA*) to remove *const*: `(DATA*)LP_filter`
- *r* – pointer to the output samples buffer. For one sample: e.g. `&output`
- *dbuffer* – pointer to a circular buffer that we created (length: *nh*+2).
- *nx* – number of processed samples.
- *nh* – number of filter coefficients.

fir function in DSPLIB

```
ushort oflag = fir (DATA *x, DATA *h, DATA *r, DATA *dbuffer, ushort nx,
ushort nh)
```

Declaration of the circular buffer (globally):

```
#define N 30                      // number of coefficients
DATA fir_buffer[N+2];             // circular buffer
const int filtr_dp[] = {...}  // filter coefficients
```

Filtering one sample (we ignore the returned flag):

```
fir(&input, (DATA*)lp_filter, &output, fir_buffer, 1, N);
```

Block processing with FIR filters

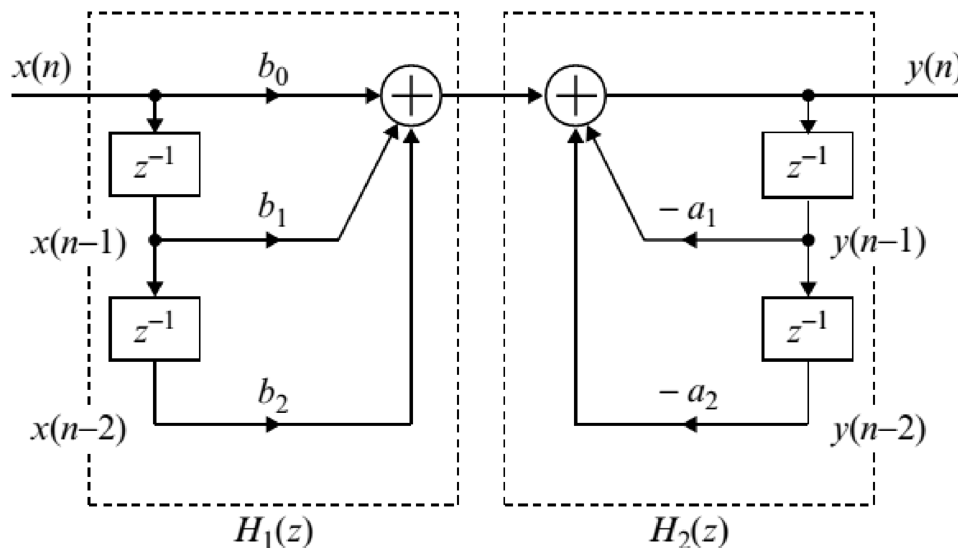
- Sometimes it is more convenient to filter a block of samples instead of each sample separately.
- We can use *fir* function from DSPLIB (time domain filtering).
- For longer filters it is faster to use convolution in the spectral domain.
- The results from all blocks are merged with *overlap-add* (OLA) or *overlap-save* (OLS) algorithm.
- This approach was presented in the *Sound and image processing* course lecture. It is based on FFT, as presented in the previous lecture.

IIR filters

Infinite impulse response (IIR) – the second type of digital filters.

- Recursive filters – require N+1 latest signal samples and N previous filtering results.
- Example for a second order filter (a *biquad*):

$$y(n) = b_0 x(n) + b_1 x(n-1) + b_2 x(n-2) - a_1 y(n-1) - a_2 y(n-2)$$



Quantization of IIR filter coefficients

- The coefficients are computed with a software.
- A problem: one coefficient (a_1) is outside the [-1, 1) range. How can we write it in a fixed-point notation?

```
0.0039, 0.0078, 0.0039, -1.8153, 0.8310
```

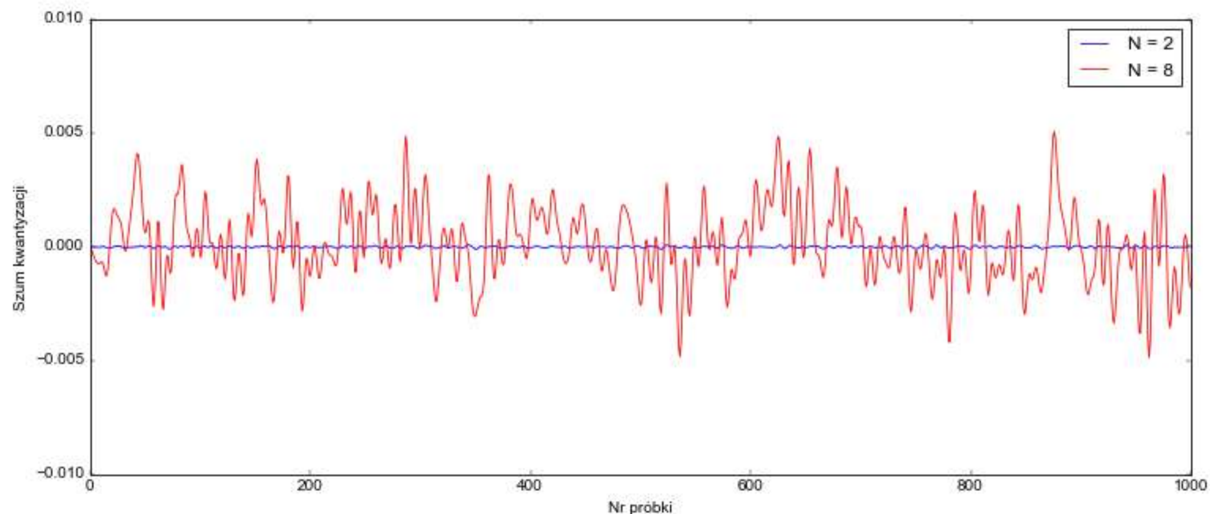
- There is no other way than give away one bit of precision and write the value in Q1.14, multiplying the floating-point values by 16384.

```
const int iir[] = {64, 128, 64, -29743, 13615};
//                 b0,  b1, b2,     a1,    a2
```

- We should check the gain and the stability of the filter.

Quantization of IIR filter coefficients

- Quantization noise also occurs in IIR filters.
- Noisy filtering results are used to process the following samples! The noise effects accumulates.
- Quantization noise increases with the filter order.
- Possible problems: range overflow and filter instability.



Cascade form

- To reduce the quantization noise, IIR filters are designed in a cascade form: as second order sections (SOS, biquads).
- In a floating-point implementation, construction of sections is not relevant.
- In a fixed-point implementation, assigning filter zeros and poles to sections may decide whether range overflow occurs or not.
- Building an optimal cascade structure for a fixed-point IIR implementation is a complex task.

What can go wrong?

Two undesirable effects may occur.

- Range overflow
  - the output of the filter is a wideband noise instead of a filtered signal,
  - it happens when at least one section has too large gain,
  - solution: reduce the filter gain.
- Range underflow
  - the output of the filter is all zeros,
  - it happens when one or more sections have too small gain,
  - solution: redistribute the filter gain among sections.

Filter design

- We compute filter coefficients in the cascade form (SOS) using software.
- For a fixed-point DSP, we convert coefficients to Q1.14.
- We check the gain of each section and of the whole filter. All gains should be similar, none should exceed 1. We modify the gain if needed (by scaling the coefficients *b*).
- We test stability of the quantized filter.
- Coefficients are stored in the program code.

IIR filters in DSPLIB

- From many IIR functions in DSPBLIB, the most useful one is *iircas51*:

```
ushort oflag = iircas51 (DATA *x, DATA *h, DATA *r, DATA *dbuffer, ushort nbiq,
ushort nx)
```

- Cascade form (*cas*) of a direct form I (1),
  5 coefficients per section.
- Software usually calculates 6 coefficients per section.
  We omit the fourth coefficient (a_0), it is always 1.
- Direct form II is not recommended for fixed-point processors,
  it is more susceptible for range overflow.

IIR filters in DSPLIB

```
ushort oflag = iircas51 (DATA *x, DATA *h, DATA *r, DATA *dbuffer, ushort nbiq,
ushort nx)
```

- *x* – pointer to the input sample (*&sample*) or to buffer of input samples (*buffer_in*)
- *h* – pointer to filter coefficients, in order: *b0, b1, b2, a1, a2*; section by section (without *a0*!)
- *r* – pointer to the output sample (*&result*) or to the buffer of output samples (*buffer_out*)
- *dbuffer* – buffer of length (4 * number of sections + 1), we need to create the buffer and fill it with zeros,
- *nbiq* – number of second order sections,
- *nx* – number of samples to process (e.g. 1).

IIR filters in DSPLIB

- Function *iircas51* assumes Q15 format. If the coefficients are Q1.14, the result after each section is two times too small.
- We can compensate the value at the output, but the underflow may occur, and we lose precision.
- Modified code (provided in the project template) – we compensate this effect after each section.

```
238          || RPTBLOCAL OuterLoopEnd-1                       ;outer loop: process a new input
239                    MOV      *AR0+ << #16, AC1                ; HI(AC1) = x(n)
240                    ||RPTBLOCAL      InnerLoopEnd-1           ;inner loop: process a bi-quad
241                            NOP_16                            ; CPU_116: Remark 5682
242                         || MPYM     *AR1+, AC1, AC0          ; AC0 = b0*x(n)
243                            MACM     *AR1+, *(AR6+T0), AC0    ; AC0 += b1*x(n-1)
244                            MACM     *AR1+, *AR6, AC0         ; AC0 += b2*x(n-2)
245                            MOV      HI(AC1), *(AR6+T1)       ; x(n) replaces x(n-2)
246                            MASM     *AR1+, *(AR4+T0), AC0    ; AC0 -= a0*y(n-1)
247                            MASM     *AR1+, *AR4, AC0, AC1    ; AC1 -= a1*y(n-2)
248                            SFTS     AC1, #1                  ; AC1 *= 2 (correction for Q14)
249                            MOV      rnd(HI(AC1)), *(AR4+T1)  ; y(n) replaces y(n-2)
250     InnerLoopEnd:
251                    AMOV     AR7, AR1                         ;reinitialize coeff pointer
252                 || MOV      rnd(HI(AC1)), *AR2+              ;store result to output buffer
```

IIR filters in DSPLIB

Declaration of coefficients (4 sections) and the work buffer

WARNING: we omit *a0* coefficient!

```
const int iir_filter[] = {
     62,    124,     62,  -28801,  12665
     63,    126,     63,  -29307,  13176
     65,    131,     65,  -30291,  14168
     68,    137,     68,  -31681,  15570};

int buffer[17];  // remember to fill it with zeros!
```

Filtering the sample in *input* and writing the result to *output*:

```
iircas51(&input, (DATA*)iir_filter, &output, buffer, 4, 1);
```

Summary - FIR filters on DSP

From the point of view of DSP implementation:

- FIR filters require significantly more multiplications and additions than IIR filters to achieve the same result.
- Features of DSPs allow for fast FIR filter implementation:
  - instructions: MAC, dual MAC, FIRLS, etc.,
  - fast loop execution without overhead,
  - special addressing modes, e.g. circular addressing.
- Therefore, FIR filters are usually the first choice in modern DSP applications.
- For “long” filters it is recommended to use spectral convolution (OLA algorithm).

Summary - IIR filters on DSP

From the point of view of DSP implementation:

- IIR filters are more problematic than FIR filters:
  - problem of dividing the filter into sections (overflow, underflow, instability),
  - high quantization noise (inaccuracy of results).
- We should consider IIR filters when:
  - available processor cycles and memory are limited (for example, we run several filters),
  - small processing delay is required (FIR filters introduce higher delay).
- We should also remember that IIR filters distort signal phase.