Applications of signal processors

SIGNAL GENERATION

on digital signal processors

Author: Grzegorz Szwoch

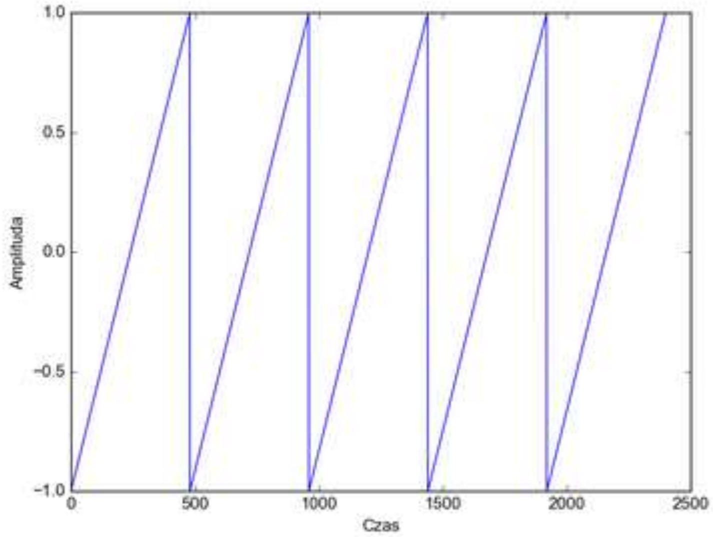
Gdańsk University of Technology, Department of Multimedia Systems

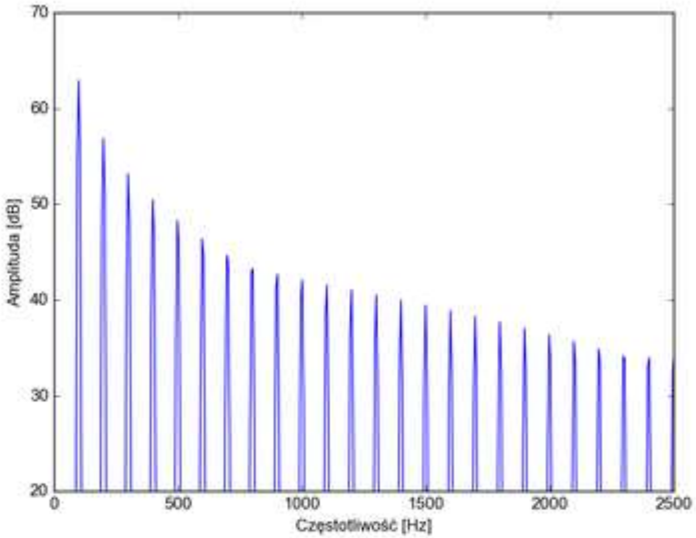
Introduction

- Usually, signal processors operate on signals that are fed to its inputs.
- We can also use DSPs to generate signals.
- In this lecture, we will talk about:
  - generating digital harmonic signals,
  - generating a sine wave,
  - generating pseudo-random signals (noise),
  - generating signals by reading samples from memory,
  - interpolation of samples stored in memory.

Sawtooth wave

- Harmonic signals: their spectrum consists of partials at harmonic frequencies – multiples of the fundamental frequency.
- Example: sawtooth wave.
  Time and spectral plot:





Sawtooth wave

- Amplitude changes linearly.
- To generate the wave, we use an accumulator – we sum up the consecutive amplitude steps.
- Initialization:

```
int amplitude = 0;
const int step = ???;
```

- For each sample, output *y*:

```
y = amplitude;
amplitude = amplituda + step;
```

- What is the value of *step*?

Calculating the amplitude step

- Let's assume frequency 1 Hz (period 1 s), fs = 48 kHz.
- We need 48000 samples to change amplitude from -32768 to 32768.
- Amplitude change per one sample is:

$$d = \frac{2 \cdot 32768}{48000} = 1.365333\ldots$$

- And if we need 100 Hz (sample 0.01 s)?

$$d = \frac{2 \cdot 32768}{48000/100} = 136.5333\ldots$$

Calculating the amplitude step

- For any frequency *f*, amplitude step as a Q15 number is:

  d = round(f * 1.36533)

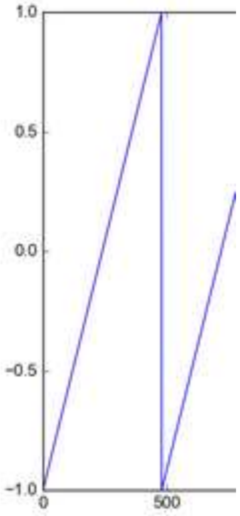
- For example: $f = 440$ Hz $\rightarrow$ $d = 601$
- If we need to compute this step in code:

$$d = f\,\frac{65536}{48000} = f\,\frac{2\cdot 22368}{32768} = \left(f * 22368\right) >> 14$$

- Remember that it’s not possible to write any frequency value in a fixed-point notation.

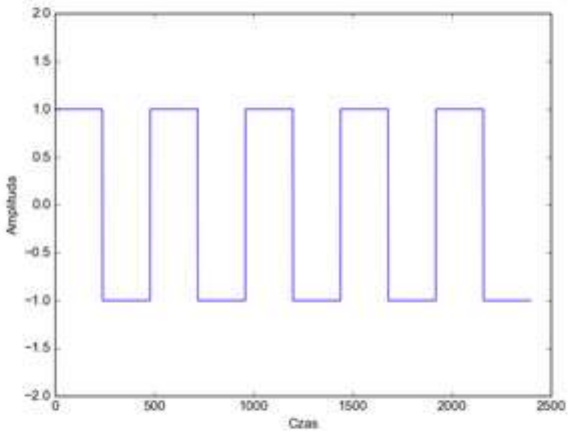
Overflow in sawtooth generation

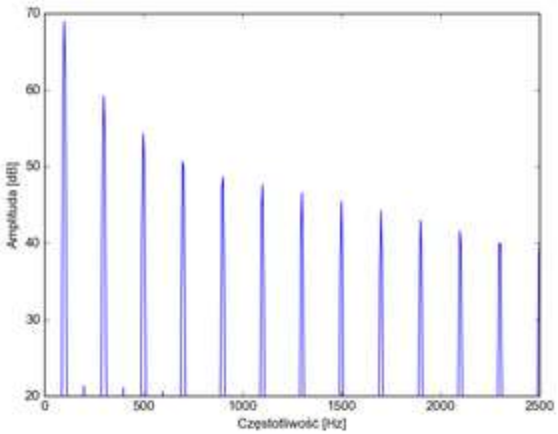
- Important: range overflow occurs when amplitude steps are accumulated, for example:
  32750 + 25 = „32775” = –32761
- Amplitude “wraps around”
  - this is exactly what we need!
- It is one of rare cases in which range overflow is actually useful.



Square / pulse wave

- Another harmonic signal: square or pulse wave.
- Signal amplitude changes between -*A* and +*A*.
- Pulse width: a ratio of duration of the positive part to the wave period (0 to 1).
- Time and spectral plots for pulse width = 0.5:



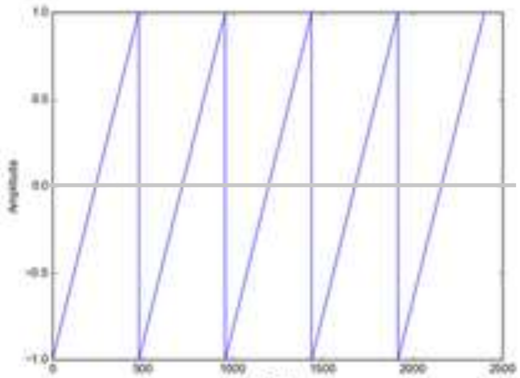


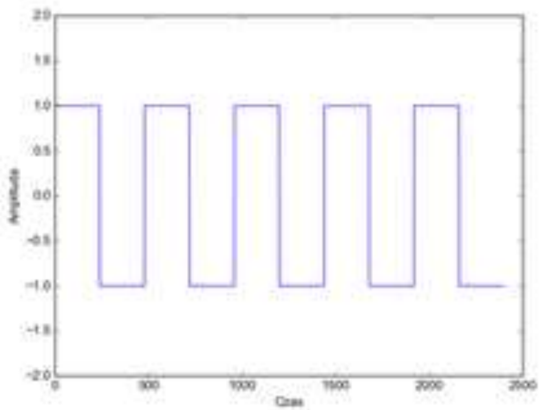
Square / pulse wave

- Square wave may be calculated from the sawtooth wave:

```
if (amplitude < threshold)
    y = 32767;    // or another amplitude
else
    y = -32768;
```

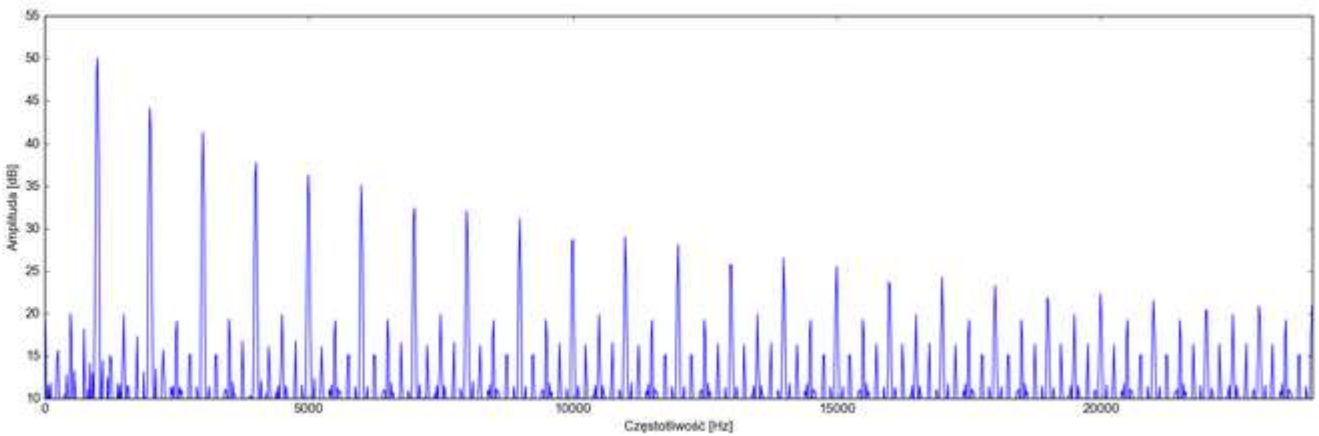
- Value of *threshold* depends on the pulse width:
  threshold = 2 * *pulse_width* – 1
- A regular square wave (50/50): threshold = 0.





Aliasing problem

- Analog harmonic waves (such as square or sawtooth) have infinite spectrum.
- If we try to generate these waves digitally “from definition”, usually aliasing will occur.
- The problem increases for higher wave frequencies.
- The resulting signal is inharmonic.



Aliasing problem

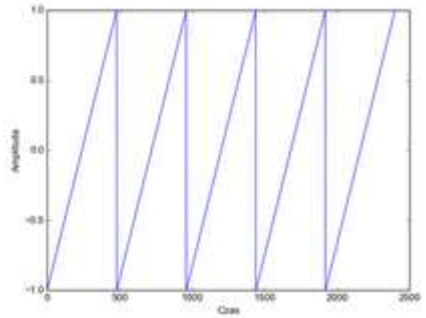
There ware various method of generating alias-free waves.

- Generating waves with higher sampling frequency (oversampling), then decimation.
- Using Fourier series. For example, sawtooth:

$$x(n) = \frac{A}{2} - \frac{A}{\pi}\sum_{k=1}^{N}(-1)^k \frac{\sin(2\pi knf / fs)}{k}$$

  - for $k \cdot f$ below Nyquist frequency,
  - the signal is distorted in time domain (lack of high frequency components).

Sine generation



- Phase of a sine wave looks like a sawtooth wave.
- We know how to generate a sawtooth wave. Now, we have to convert phase (angle) into the amplitude.
- We can approximate the sine with Taylor series:

$$\sin(x) \cong x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \frac{x^9}{9!} - \dots$$

- Function *sine* from DSPLIB for C55x uses this method.

Sine wave generation with DSPLIB

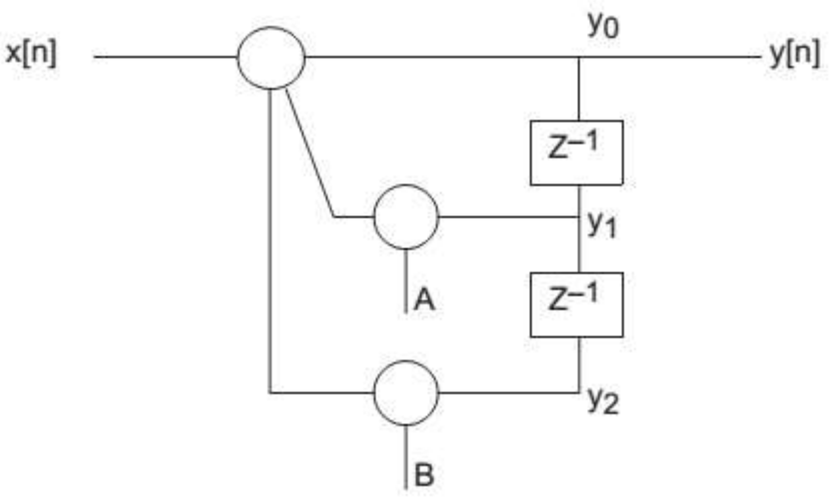
sine	*Sine*
Function	ushort oflag = sine (DATA *x, DATA *r, ushort nx)

- *x* – pointer to the buffer with phase (angle) values. We generate a sawtooth wave of a desired frequency and write its values to the buffer.
- *r* – pointer to the buffer in which values of a sine wave will be stored.
- *nx* – number of samples to generate (buffer length).

Note that *sine* function does not generate a sine wave signal by itself, it only computes sine values from given angles.

Sine wave from IIR generator

- Alternative method of sine wave generation:
  we use a marginally-stable second-order IIR system.
- We use an impulse to start the generator:
  $y(0) = -\sin(2\pi f/fs)$, $y(1) = 0$
- The system goes into oscillations – generates sine wave values.
- On a fixed-point DSP, implementation is problematic (insufficient numerical precision).



$$a = 2\cos\left(\frac{2\pi f}{fs}\right)$$

$$y(n) = a \cdot y(n-1) - y(n-2)$$

White noise generation

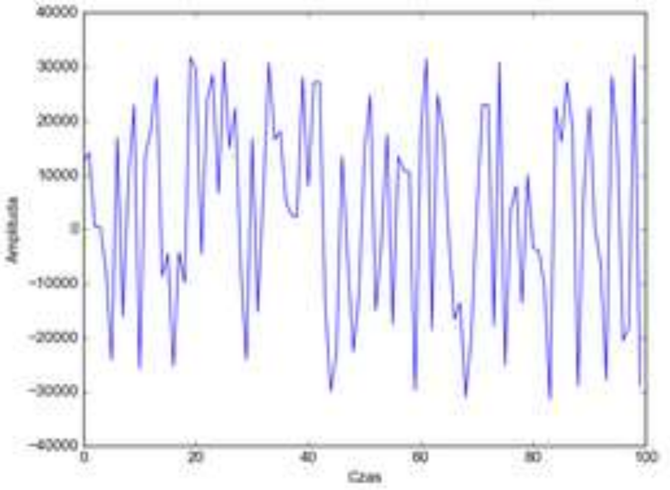
- White noise – a random signal with flat spectrum.
- To generate a digital noise, we use (pseudo)random number generators – RNG.
- Noise samples are computed by the algorithm.
- Example of a simple noise generation algorithm:
  LCG – linear congruent generator:

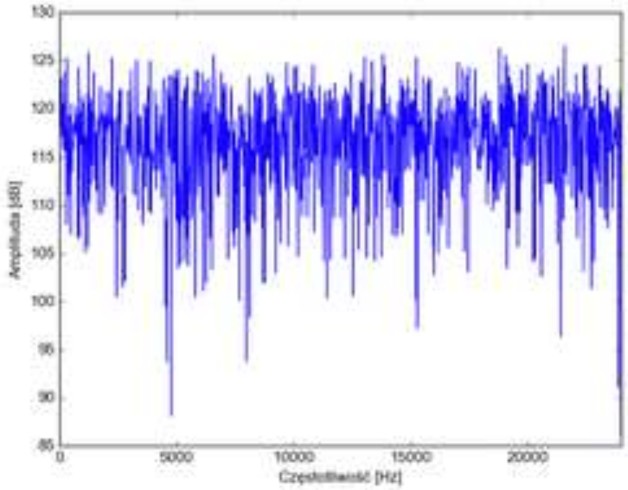
  $$y(n) = \left[a \cdot y(n-1) + b\right] \bmod M$$

  mod – modulo, remainder of integer division by *M*
- For professional applications, such as cryptography, more accurate algorithms are needed (e.g., Mersenne Twister).

White noise generation

- Initial value $y(0)$ is called a seed. Given the same seed, the algorithm will always generate the same sequence of pseudo-random numbers.
- In practice, we set the seed to a constantly changing value, e.g., a counter of processor cycles.
- Example: $a = 2045$, $b = 0$, $M = 2^{20}$, $y(0) = 12345$.
  Time and spectrum plots:





White noise generation in DSPLIB

- Initialization – just once, when the program starts:

```
rand16init();
```

- Writing *nr* samples into buffer *r*:

rand16	*Random Number Generation Algorithm*
Function	ushort oflag= rand16 (DATA *r, ushort nr)

LCG: $a = 31821$, $b = 13849$, $M = 65536$

```
rand16(bufor, 2048);
```

Signal from a wavetable

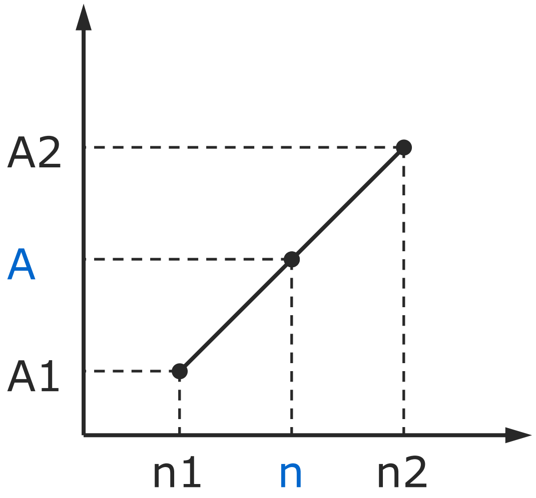
- Any signal can be generated by reading its samples stored in memory, in a wavetable (a buffer of samples).
- For example, we can store 480 samples of one period of a sine wave.
- If we read them with speed of 48 kHz, we get a sine wave of frequency 100 Hz.
- If we read every second sample, we have 240 samples per period, therefore $f = 48000 / 240 = 200$ Hz
- If we loop the samples we read, we get a continuous wave.
- The problem: how can we generate *any* frequency?

Signal from a wavetable

- A general case: generating wave of any frequency.
- Step to move the read index in the wavetable:
  $s = f \cdot N / fs$ (N – number of samples in wavetable).
- For f = 456 Hz, N = 4800: s = 45.6
- Usually, step s is not an integer.
- So, we need to "read between samples".
- Interpolation of samples: estimating values between samples stored in the memory.

Linear interpolation

- The simplest interpolation is linear. We connect the known samples with a straight line, and we look for a value at a given position on the line.
- Let *index* = 45.6. We interpolate between samples *x*[45] and *x*[46] – the previous and the next one.



$$\frac{A_2 - A_1}{n_2 - n_1} = \frac{A - A_1}{n - n_1}$$

$$n_2 - n_1 = 1$$

$$A = A_1 + (A_2 - A_1)(n - n_1)$$

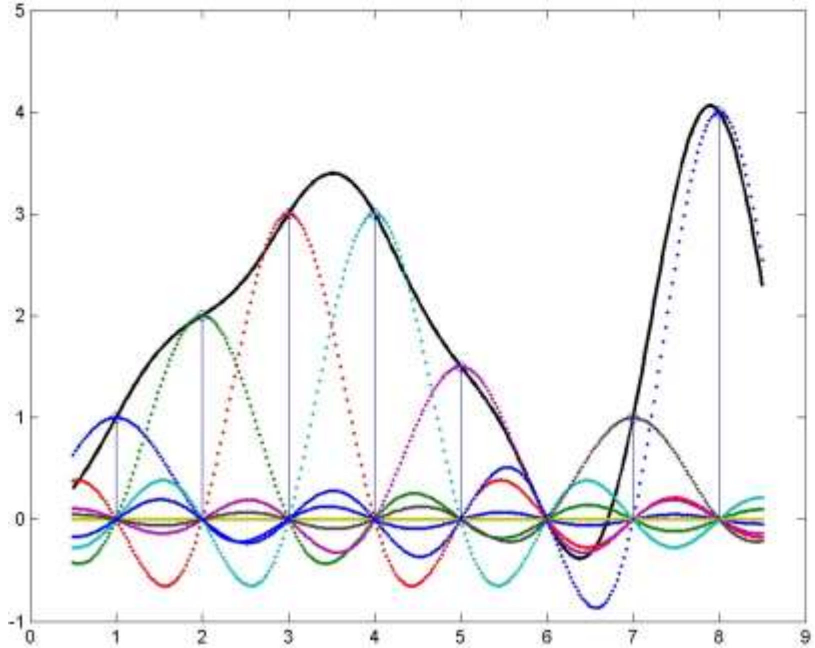
$$x[45.6] = x[45] + \left(x[46] - x[45]\right) \cdot 0.6$$

Reading samples with interpolation

```
// Example: index = 45.6
int index_c = 45;             // integer part (45), int
int index_u = 19661;          // fractional part (0.6), Q15

// Read samples from table
int a1 = buffer[index_c];     // the previous sample
int a2 = buffer[index_c+1];   // the next sample

// linear interpolation
long a = index_u * (long)(a2 – a1)   // (n-n1)*(a2-a1)
a = (_sround(a<<1) >> 16) + a1       // + a1

// value of the interpolated sample
y = (int)a;
```

Other interpolation methods

- Linear interpolation is simple, but not accurate.
- More accurate methods:
  - polynomial interpolation – of degree 2 (square), 3 (cubic) and higher degrees.
  - interpolation with sin(x)/x functions (sinc interpolation):



Signal from a wavetable

- The more samples in the buffer, the better (lower interpolation errors).
- We can store any signal in the wavetable.
- This methods works fine if we read and loop a wave period.
- If we do not loop, frequency change results in changing the duration – the signal becomes shorter or longer.
- Interpolation of signals with complex spectrum, such as square wave, may result in aliasing. Usually, we need to store a number of wave versions of different frequencies in the wavetable.

Signal from a wavetable

Reading a sound signal from table with different steps:

- step = 1:
  - samples read with the original sampling frequency,
  - sound pitch – the same as the original sampled sound;
- step < 1:
  - we read samples more slowly – sound becomes longer
  - the sound pitch is lower than the original;
- step > 1:
  - we read samples quicker – sound becomes shorter
  - the sound pitch is higher than the original.

Sampler

- A practical example: a sampler – digital musical instrument that plays back sound samples stored in memory. Only fragments of samples are looped, or they are not looped.
- A DSP in the sampler does transposition
  - changes the pitch of generated sounds by altering the step of memory read index and by interpolation.
- Temporal distortions occur
  – the sound becomes longer or shorter. Therefore, we need to use a set of samples with different pitch (multisampling).