Poznań University of Technology

Faculty of Control, Robotics and Electrical Engineering

Institute of Robotics and Machine Intelligence

Division of Control and Industrial Electronics

Digital Signal Processing

Mobile and embedded applications for Internet of Things

Teaching materials for laboratory

Dominik Łuczak, Ph.D.; Adrian Wójcik, M.Sc.

Dominik.Luczak@put.poznan.pl
Adrian.Wojcik@put.poznan.pl

I. Goal

Knowledge

The aim of the course is to familiarize yourself with:

- an example of the DSP use in IoT applications,
- using dedicated tools for DSP in the application development process,
- test-driven software development technique,

Skills

The aim of the course is to acquire skills in:

- creating automatic scripts that generate and analyze DSP algorithms,
- implementing simple DSP algorithms in general-purpose languages,
- proper testing of DSP algorithms in target environments,
- integration of DSP algorithms with IoT applications.

Social competences

The aim of the course is to develop proper attitudes:

- using systematic software development techniques and methodologies,
- strengthening understanding and awareness of the importance of non-technical aspects and effects of the engineer's activities, and the related responsibility for the decisions taken,
- choosing the right technology and programming tools for the given problem,
- testing developed IT system.

II. Laboratory report

Complete laboratory tasks as per the instructor's presentation. Work alone or in a team of two. **Keep safety rules while working!** Prepare laboratory report documenting and proving the proper execution of tasks. Editorial requirements and a report template are available on the *eKursy* platform. The report is graded in two categories: tasks execution and editorial requirements. Tasks are graded as completed (1 point) or uncompleted (0 points). Compliance with the editorial requirements is graded as a percentage. The report should be sent as a *homework* to the *eKursy* platform by Thursday, June 17, 2021 by 23:59.

III. Prepare to course

a) Know the safety rules

All information on the laboratory's safety instructions are provided in the laboratory and on Division website [1]. All inaccuracies and questions should be clarified with the instructor. It is required to be familiar with and apply to the regulations.

Attend the class prepared. Knowledge from all previous topics is mandatory.

B) Example DSP algorithm: low pass filter

Digital Signal Processing (DSP) is a key element of modern control and measurement systems. Therefore, it is also a necessary element of many applications for Internet of Things (IoT). The correct implementation of DSP algorithms - which in practice means *sufficiently tested* - is therefore an important element in the development of IoT systems. However, the implementation of DSP algorithms is a complex, interdisciplinary task involving signals and dynamic systems theory, numerical analysis and software development. Consequently, it requires knowledge of theoretical basis of the issue and use of adequate programming tools.

One of the DSP algorithms used in a very wide range of applications are *digital filters*. The instruction presents an example of the implementation of a low-pass filter with a finite impulse response (FIR filter). In fig. 1 a diagram of the adopted software development technique is presented. The presented procedure consists of the following stages:

1. Filter design using the GNU Octave [2] environment and (alternatively) the SciPy [3] library.
2. The implementation of the filter for client applications of the IoT system in three general-purpose languages: Java, JavaScript and C#, for three different environments, respectively: a mobile application for Android, a web application and a desktop application for Windows.
3. Conducting automatic unit tests of implemented algorithms.
4. Integration of algorithms with demo applications with network communication and a graphical user interface (GUI), i.e. emulating real use cases of DSP.

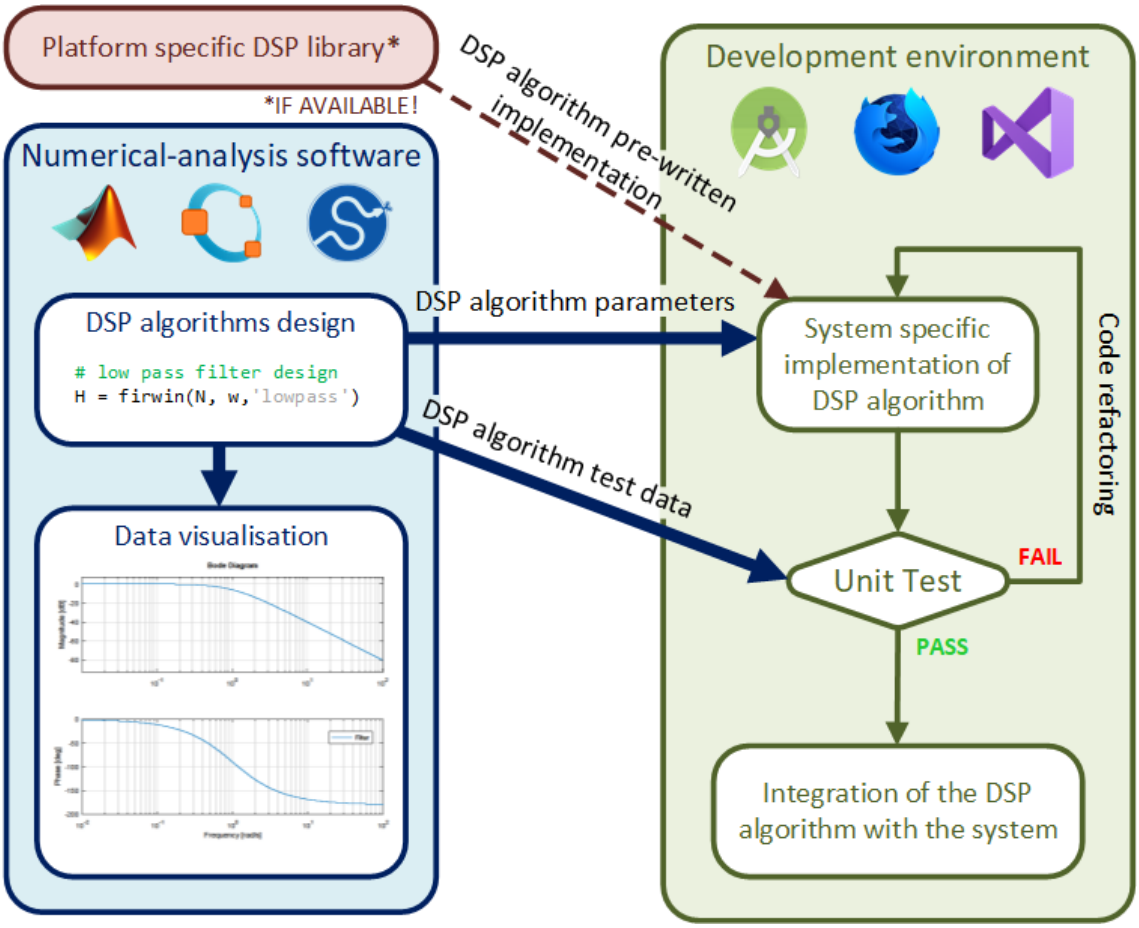

Fig. 1. Schematic diagram of the adopted development technique for implementing DSP algorithm.

1. Filter design procedure

Most software tools for digital signal processing and analysis include graphical digital filter design tools. However, in software development, automation and repeatability of processes - including design processes - is often very important. Hence the presented example is based on *scripted* procedure for designing FIR filter. Available libraries for DSP, such as the **signal** package for the GNU Octave environment or the **SciPy** library written in Python, contain procedures to obtain FIR filter coefficients based on minimum necessary information: filter order, cutoff frequencies and filter type (low-, high- or bandpass). The FIR filter can be expressed as (1). The filter design involves finding a set of b_k coefficients that will allow to obtain the assumed frequency or time characteristics.

$$x_f(k) = b_0 x(k) + b_1 x(k-1) + b_2 x(k-2) + \cdots + b_N x(k-N) \tag{1}$$

where:
- x_f - filter output - a filtered digital signal,
- x - filter input - an original digital signal,
- k - sample number,
- N - filter order,
- $b_{0\cdots N}$ - filter feedforward coefficients,

In the example, assumptions were made regarding the frequency characteristics of the filter: cutoff frequencies and attenuation level. *harris* method, presented in the equation (2), was used to estimate the low-pass filter order based on the assumed transition bandwidth and attenuation in the stopband [4].

$$\hat{N} = \frac{f_s}{f_2 - f_1} \cdot \frac{A_{dB}}{22} \tag{2}$$

where:
- $\hat{N}$ - approximated filter order,
- f_s - sampling frequency,
- f_1 - passband cutoff frequency,
- f_2 - stopband cutoff frequency,
- A_{dB} - assumed attenuation in stopband,

The example assumes the following design parameters for the filter:

- Sampling period $t_s = 100$ ms; i.e. sampling frequency $f_s = 10$ Hz - this is a frequency that can easily be achieved using home wireless networks (e.g. WiFi).
- Passband cutoff frequency $f_1 = 0.5$ Hz - so we assume that frequencies higher than 0.5 Hz are caused by noise e.g. measurement errors.
- Stopband cutoff frequency $f_2 = 0.8$ Hz - at this frequency the filter should have the assumed attenuation of A_{dB}.
- Assumed attenuation in stopband $A_{dB} = 60$ dB - attenuation corresponding to a 1000-fold decrease in signal amplitude.
- With the adopted assumptions and filter order estimation method (2), the filter order N will be equal to 91, which corresponds to 92 filter coefficients.

The filter design and analysis procedure was carried out as follows:

1. A test signal vector was generated consisting of four harmonic components with unit amplitude and frequencies $f = \{0.1, 0.2, 0.7, 1.0\}$. The amplitude of the first two components should not change after applying filter, while the amplitude of the last two should be significantly reduced, i.e. the last two components will be *filtered*. Signal amplitude spectrum was computed.

2. Assumed filter parameters have been defined.
3. Filter order was estimated using(2).
4. Filter coefficients were obtained using the available library functions: `fir1` in GNU Octave [5] and `firwin` in SciPy [6]. Both of these methods by deafault use the Hamming window [7].
5. The frequency characteristics of the resulting filter were generated by performing Fourier transform on filter coefficients - for the FIR filter it is an equivalent to transform of filter impulse response.
6. The test signal was filtered using the available library functions: `filter` in GNU Octave [8] and `lfilter` in SciPy [9]. Signal amplitude spectrum after filtration was computed.

2. Filter design in GNU Octave environment

In listing 1. presents GNU Octave environment script implementing procedure of designing, analyzing and testing digital filter. In fig. 2. results of this procedure are presented in a visual form: signal time series before and after filtration and the signal amplitude spectrum before and after filtration in combination with frequency characteristics of designed filter.

Listing 1. Filter design and analysis - Octave environment.

```
pkg load signal

% sampling frequency
ts = 0.1;  % [s]
fs = 1/ts; % [Hz]

%% Test signal n
% frequency components
fcomp = [0.1, 0.2, 0.7, 1.0]; % [Hz]

% signal (in sample domain)
signal = @(n)( sin(2*pi*fcomp(1)*(n/fs)) + ...
               sin(2*pi*fcomp(2)*(n/fs)) + ...
               sin(2*pi*fcomp(3)*(n/fs)) + ...
               sin(2*pi*fcomp(4)*(n/fs)) );

nvec = 0 : 10^4 - 1;
xvec = signal(nvec)';
% test signal amplitude spectrum
% frequency vector
frange = (-1/2 : 1/length(nvec) : 1/2-1/length(nvec)); % [-]
fxvec = frange*fs;                                     % [Hz]
% amplitude response
Axvec = abs(fftshift(fft(xvec,length(nvec)))); % [-]
Axvec = Axvec / length(nvec);

%% Filter desired parameters
% cut-off frequency
f1 = 0.5; % [Hz]
% frequency at the end transition band
f2 = 0.8; % [Hz]
% transition band length
df = f2 - f1;
% stopband attenuation of A [dB] at f2 [Hz]
A = 60; % [dB]

%% Filter order estimation
% with 'fred harris rule of thumb'
N = round( (fs/df)*A / 22 );

```

```
%% Filter object with fir1 funcion
% normalised frequency
w = f1 / (fs/2); % [-]
% low pass filter
H = fir1(N, w, 'low');

%% Filter frequency response
% no. of samples
n = 10^4; % [-]
% frequency vector
frange = (-1/2 : 1/n : 1/2-1/n); % [-]
fhvec = frange*fs;               % [Hz]
% amplitude response
Ahvec_v1 = abs(freqz(H, 1, 2*pi*frange)); % [-] frequency response
Ahvec_v1 = 20*log10(Ahvec_v1); % [dB]
Ahvec_v2 = abs(fftshift(fft(H,n)));       % [-] impulse response fft
Ahvec_v2 = 20*log10(Ahvec_v2); % [dB]

%% Test signal filtration
xfvec = filter(H,1,xvec);

% filtered signal amplitude spectrum
% amplitude response
Axfvec = abs(fftshift(fft(xfvec,length(nvec)))); % [-]
Axfvec = Axfvec / length(nvec);

%% Test signal filtratrion - reference implementation
b = H;         % feedforward filter coefficients
x_state = zeros(size(H));   % initial state
xfvec2 = zeros(size(xvec)); % filtration result

myFir = MyFir(b, x_state);  % Object-oriented implementation

for i = 1 : length(xvec);
  [xfvec2(i), myFir] = Execute(myFir, xvec(i));
endfor
```

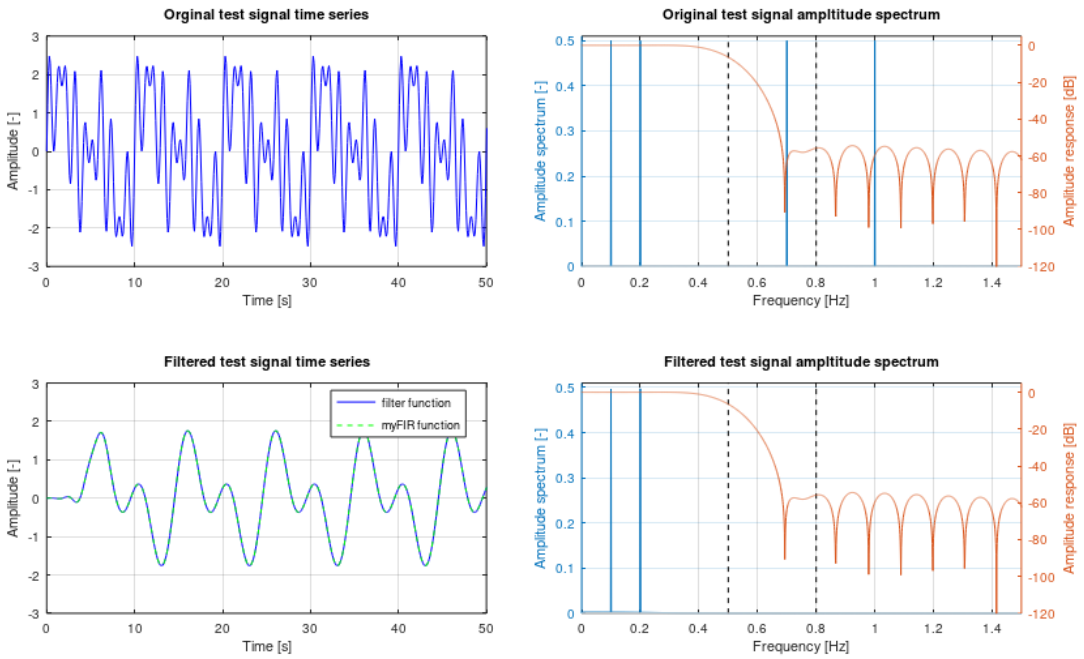

Fig. 2. Visualization of filter analysis - Octave environment.

Based on the obtained results, were generated source files (`.java`, `.js`, `.cs`) containing parameters (vector `H`) and initial filter state (vector with `N+1` zeros) as well as unit test data in the form of 500 samples of both original test signal and signal after filtration.

3. Filter design in Python with SciPy

In listing 2. is presented a script written in Python using the SciPy library that implements the procedure of designing, analyzing and testing a digital filter. In fig. 2. results of this procedure are presented in a visual form: signal time series before and after filtration and the signal amplitude spectrum before and after filtration in combination with frequency characteristics of designed filter. Comparing listings 1. and 2. it is not difficult to notice similarity of all stages of both procedure as well as similarity of library functions interfaces. As expected, the results of both procedures coincide.

Listing 2. Filter design and analysis - Python with SciPy library.

```
01. import math
02. import numpy as np
03. from scipy.fftpack import fft, fftshift
04. from scipy.signal import firwin, lfilter
05. import matplotlib.pyplot as plt
06.
07. from MyFir.MyFir import MyFir
08. #    path . file         class
09.
10. # sampling frequency
11. ts = 0.1;  # [s]
12. fs = 1/ts; # [Hz]
13.
14. ## Test signal
15. # frequency components
16. fcomp = [0.1, 0.2, 0.7, 1.0]; # [Hz]
17.
18. # signal (in sample domain)
19. signal = lambda n : (math.sin(2*math.pi*fcomp[0]*(n/fs))+
20.                      math.sin(2*math.pi*fcomp[1]*(n/fs))+
21.                      math.sin(2*math.pi*fcomp[2]*(n/fs))+
22.                      math.sin(2*math.pi*fcomp[3]*(n/fs)))
23.
24. nvec = list(range(10**4))
25. xvec = [signal(n) for n in nvec]
26. # test signal amplitude spectrum
27. # frequency vector
28. frange = np.arange(-1/2, 1/2, 1/len(nvec)) # [-]
29. fxvec = frange*fs                           # [Hz]
30. # amplitude response
31. Axvec = abs(fftshift(fft(xvec)))     # [-]
32. Axvec = [a/len(nvec) for a in Axvec] # [-]
33.
34. ## Filter desired parameters
35. # cut-off frequency
36. f1 = 0.5 # [Hz]
37. # frequency at the end transition band
38. f2 = 0.8 # [Hz]
39. # transition band length
40. df = f2 - f1
41. # stopband attenuation of A [dB] at f2 [Hz]
42. A = 60 # [dB]
43.
44. ## Filter order estimation
45. # with 'fred harris rule of thumb'
46. N = round( (fs/df)*A / 22 )
```

```
47.
48. ## Filter object with fir1 funcion
49. # normalised frequency
50. w = f1 / (fs/2) # [-]
51. # low pass filter
52. H = firwin(N+1, w, pass_zero='lowpass')
53.
54. ##Filter frequency response
55. # no. of samples
56. n = 10**4; # [-]
57. # frequency vector
58. frange = np.arange(-1/2, 1/2, 1/n) # [-]
59. fhvec = frange*fs                  # [Hz]
60. # amplitude response
61. Ahvec = abs(fftshift(fft(H,n=n)))                    # [-] impulse response fft
62. Ahvec = [20*math.log10(a + 1.0e-15) for a in Ahvec] # [dB]
63.
64. ## Test signal filtration
65. xfvec = lfilter(H,1,xvec)
66.
67. # filtered signal amplitude spectrum
68. # amplitude response
69. Axfvec = abs(fftshift(fft(xfvec)))      # [-]
70. Axfvec = [a/len(nvec) for a in Axfvec] # [-]
71.
72. ## Test signal filtratrion - reference implementation
73. b = H.tolist()           # feedforward filter coefficients
74. x_state = np.zeros(len(H)).tolist()   # initial state
75. xfvec2 = np.zeros(len(xvec)).tolist() # filtration result
76.
77. myFir = MyFir(b, x_state)   # Object-oriented implementation
78.
79. for i in range(len(xvec)):
80.   xfvec2[i] = myFir.Execute(xvec[i])
```

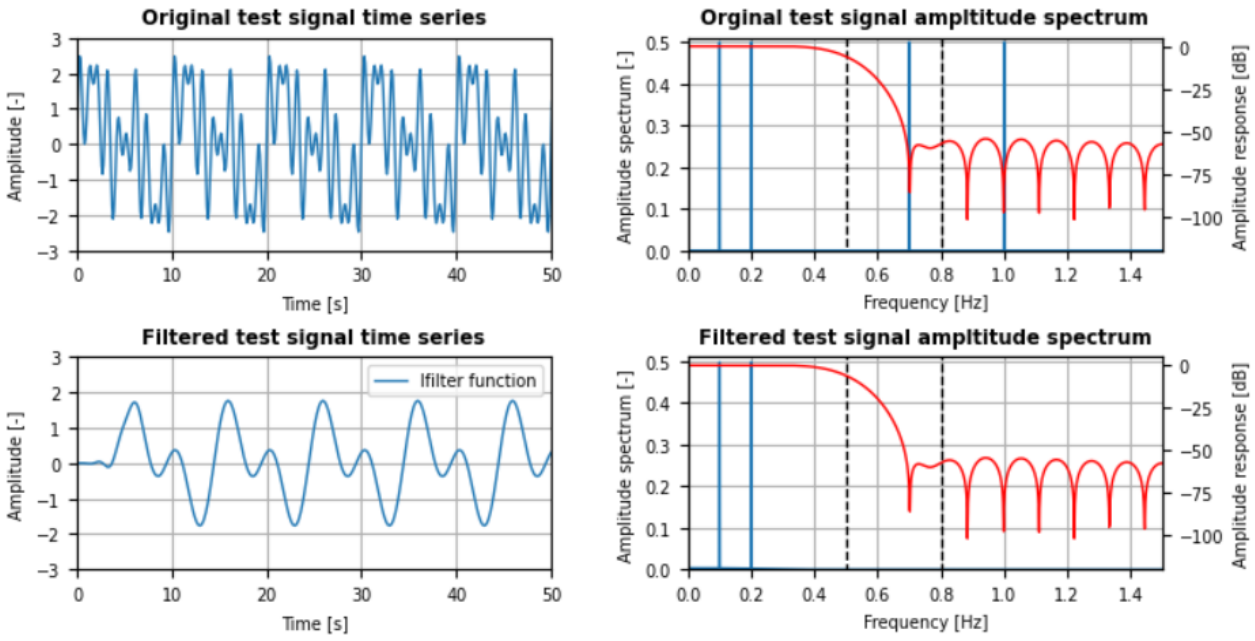

Fig. 3. Visualization of filter analysis - Python with SciPy library.

c) Implementation of DSP in general-purpose programming languages

After carrying out presented filter design procedure and obtaining source files for mobile app (Java), web app (JavaScript) and desktop app (C#) the implementation of the FIR filter algorithm was carried out in all above-mentioned general-purpose languages. The filtration task is a frequently used software procedure, so for each of the languages listed you can use available (often free) libraries containing ready implementations. In the example, however, attention is drawn to the fact that the language in which DSP algorithm is implemented is a secondary issue to the algorithm itself. Unless dedicated equipment is used, the algorithm stages will usually be implemented in a similar way regardless of the language used, assuming the same paradigm is used. In all the examples of implementation of FIR filter object-oriented technique is used: filter is an object containing information about the coefficients and filter state, the initialization procedure in the form of a class constructor and the `Execute` method performing a single algorithm step.

It is worth noting that in a typical use of the *test driven development*, i.e. test-based software development technique, test creation should be done ahead of the actual implementation of functionality [10].

1. FIR filter in Java

In listing 3. an example implementation of FIR filter in Java is presented, based on the equation (1), in the form of `MyFir` class. In order to easily modify the filter state, the `ArrayList` class was used as the container for this data.

Listing 3. FIR filter implementation in Java.

```
01. public class MyFir {
02.   //!< Array of FIR filter feedforward coefficients
03.   private Double[] feedforward_coefficients;
04.   //!< List of FIR filter state values
05.   private List<Double> state;
06.
07.   /**
08.    * @brief Initialization of FIR filter algorithm.
09.    * @param ffc Array of FIR filter feedforward coefficients.
10.    * @param st Array of FIR filter input initial state.
11.    *         Must be the same size as coefficients array.
12.    */
13.   public MyFir(Double[] ffc, Double[] st)
14.   {
15.     feedforward_coefficients = ffc;
16.     state = new ArrayList<>(Arrays.asList(st));
17.   }
18.   /**
19.    * @brief Execute one step of the FIR filter algorithm.
20.    * @param x Input signal.
21.    * @retval Output [filtered] signal.
22.    */
23.   public Double Execute(Double x)
24.   {
25.     // update state
26.     state.add(0, x);
27.     state.remove(state.size() - 1);
28.     // compute output
29.     Double xf = 0.0;
30.     for (int i = 0; i < state.size(); i++) {
31.         xf += feedforward_coefficients[i] * state.get(i);
32.     }
33.     return xf;
34.   }
35. }
```

2. FIR filter in JavaScript

In listing 4. an example implementation of FIR filter in JavaScript is presented, based on the equation (1), in the form of `MyFir` class.

Listing 4. FIR filter implementation in JavaScript.

```
01. class MyFir {
02.   /**
03.    * @brief Initialization of FIR filter algorithm.
04.    * @param ffc Array of FIR filter feedforward coefficients.
05.    * @param st Array of FIR filter input initial state.
06.    *         Must be the same size as coefficients array. */
07.    constructor(ffc, st) {
08.     this.feedforward_coefficients = ffc;
09.     this.state = st;
10.    }
11.   /**
12.    * @brief Execute one step of the FIR filter algorithm.
13.    * @param x Input signal.
14.    * @retval Output [filtered] signal.
15.    */
16.   Execute(x) {
17.     // update state
18.     this.state.unshift(x);
19.     this.state.pop();
20.     // compute output
21.     let xf = 0.0;
22.     for (let i = 0; i < this.state.length; i++) {
23.         xf += this.feedforward_coefficients[i]*this.state[i];
24.     }
25.     return xf;
26.   }
27. }
```

3. FIR filter in C#

In listing 5. an example implementation of the FIR filter in C# is presented, based on the equation (1), in the form of the `MyFir` class. In order to easily modify the filter state, the `List` class was used as the container for this data.

Listing 5. FIR filter implementation in C#.

```
01. public class MyFir
02. {
03.   //!< Array of FIR filter feedforward coefficients
04.   private double[] feedforward_coefficients;
05.   //!< List of FIR filter state values
06.   private List<double> state;
07.   /**
08.     * @brief Initialization of FIR filter algorithm.
09.     * @param ffc Array of FIR filter feedforward coefficients.
10.     * @param st Array of FIR filter input initial state.
11.     *         Must be the same size as coefficients array.
12.     */
13.   public MyFir(double[] fc, double[] s)
14.   {
15.     feedforward_coefficients = fc;
16.     state = new List<double>(s);
17.   }
```

```
18.   /**
19.    * @brief Execute one step of the FIR filter algorithm.
20.    * @param x Input signal.
21.    * @retval Output [filtered] signal.
22.    */
23.   public double Execute(double x)
24.   {
25.     // update state
26.     state.Insert(0, x);
27.     state.RemoveAt(state.Count - 1);
28.     // compute output
29.     double xf = 0.0;
30.     for (int i = 0; i < state.Count; i++)
31.     {
32.         xf += feedforward_coefficients[i] * state[i];
33.     }
34.     return xf;
35.   }
36. }
```

It is not difficult to notice that all presented implementations use similar steps and differ only in the details resulting from the syntax of individual languages and used data containers. The main conclusion from this observation should be that understanding the operation of DSP algorithms at a basic level will allow you to implemented them in almost any environment or platform without much difficulty.

d) Verification of correct DSP implementation: unit tests

The concept of *unit testing* is based on creating dedicated code that **automatically** tests **unit** code: class, object or function in isolation from other classes in the system; the code is tested only in a given unit of code and all dependencies are simulated by so-called *mocks* i.e. code that pretends to implement specific functionality, e.g. database or server (also called *stub*, *fake*, *test spy*, *dummy* - the nomenclature in this matter is broad and inconsistent in the literature on the subject). This section provides examples of unit tests in three different environments:

- Android Studio IDE for implementation in Java,
- Chrome web browser using developer tools for implementation in JavaScript,
- Visual Studio IDE for implementation in C#.

All presented unit tests check the implementation of `MyFir` class by creating a new object of this class, performing 500 filtration steps and comparing the responses with those obtained in the GNU Octave environment. The adopted performance error metric is root mean square error (RMSE). The assumed error tolerance is 10^{-10}. In practice, all the tests presented are characterized by zero error, however, in the case of more complex, non-linear algorithms, minor discrepancies between different environments should be expected.

Please note that unit testing in only *start* of the multi-stage software testing process. After performing unit tests, it is necessary to carry out, among others integration tests. Integration tests check correct operation of many code units to a different extent: from several connected classes to the use of different systems, e.g. database, server, user interfaces. Integration tests based on external systems, i.e. those that use e.g. a database system or application server, are called system tests. Integration tests that also use the user interface are called functional or end-to-end (e2e) tests.

1. FIR filter unit test in Java

Listing 6. shows unit test code for FIR filter written in Java. Test was performed with JUnit framework and Android Studio IDE [11]. In fig. 4. is presented user interface of the environment after a successful test.

Listing 6. Unit test code for the `MyFir` class (Java) in Android Studio IDE.

```
/**
 * @brief Unit Tests of 'MyFir' class.
 */
public class MyFirUnitTest {
  /**
   * @brief Unit Tests of 'Execute' method.
   *        Calculates RMSE of the filter response
   *        relative to the original design (Octave).
   *        Error tolerance: 10^(-10).
   */
  @Test
  public void ExecuteTest() {
    double maxError = 1e-10;
    double meanError = 0;
    int N = MyFirTestData.Input.length;

    MyFir filter = new MyFir(MyFirData.feedforward_coefficients,MyFirData.state);

    for(int k = 0; k < N; k++){
        double output = filter.Execute(MyFirTestData.Input[k]);
        double outDelta = (MyFirTestData.RefOutput[k] - output);
        meanError += outDelta*outDelta;
    }
    meanError = Math.sqrt(meanError/N);
    assertEquals(0, meanError, maxError); //!< JUnit
  }
}
```

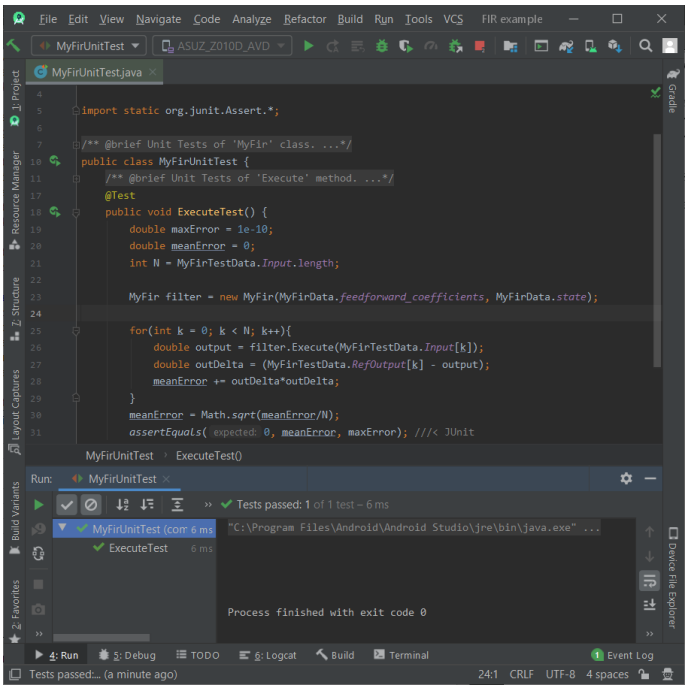

Fig. 4. Unit test of the `MyFir` class (Java) in Android Studio IDE completed successfully.

2. FIR filter unit test in JavaScript

Listing 7. shows unit test code for FIR filter written in JavaScript. Tests was performed using Chrome browser and the Unit.js library [12]. A simple HTML document was created in the browser to run the test. In fig. 5. is presented HTML document and information in the web console after a successful test.

Listing 7. Unit test code for the `MyFir` class (JavaScript) in Chrome browser.

```
01. /**
02.  * @brief Unit Tests of 'MyFir' class.
03.  */
04. var MyFirUnitTest = {
05.   /**
06.    * @brief Unit Tests of 'Execute' method.
07.    *        Calculates RMSE of the filter response
08.    *        relative to the original design (Octave).
09.    *        Error tolerance: 10^(-10).
10.    */
11.   ExecuteTest: () => {
12.     let maxError = 1e-20;
13.     let meanError = 0;
14.     let N = MyFirTestData.Input.length;
15.
16.     let filter = new MyFir(MyFirData.feedforward_coefficients, MyFirData.state);
17.
18.     for(let k = 0; k < N; k++){
19.         let output = filter.Execute(MyFirTestData.Input[k]);
20.         let outDelta = (MyFirTestData.RefOutput[k] - output);
21.         meanError += outDelta*outDelta;
22.     }
23.     meanError = Math.sqrt(meanError/N);
24.
25.     unitjs.number(meanError).isBetween(0, maxError); //!< Unit.js
26.   }
27. }
```

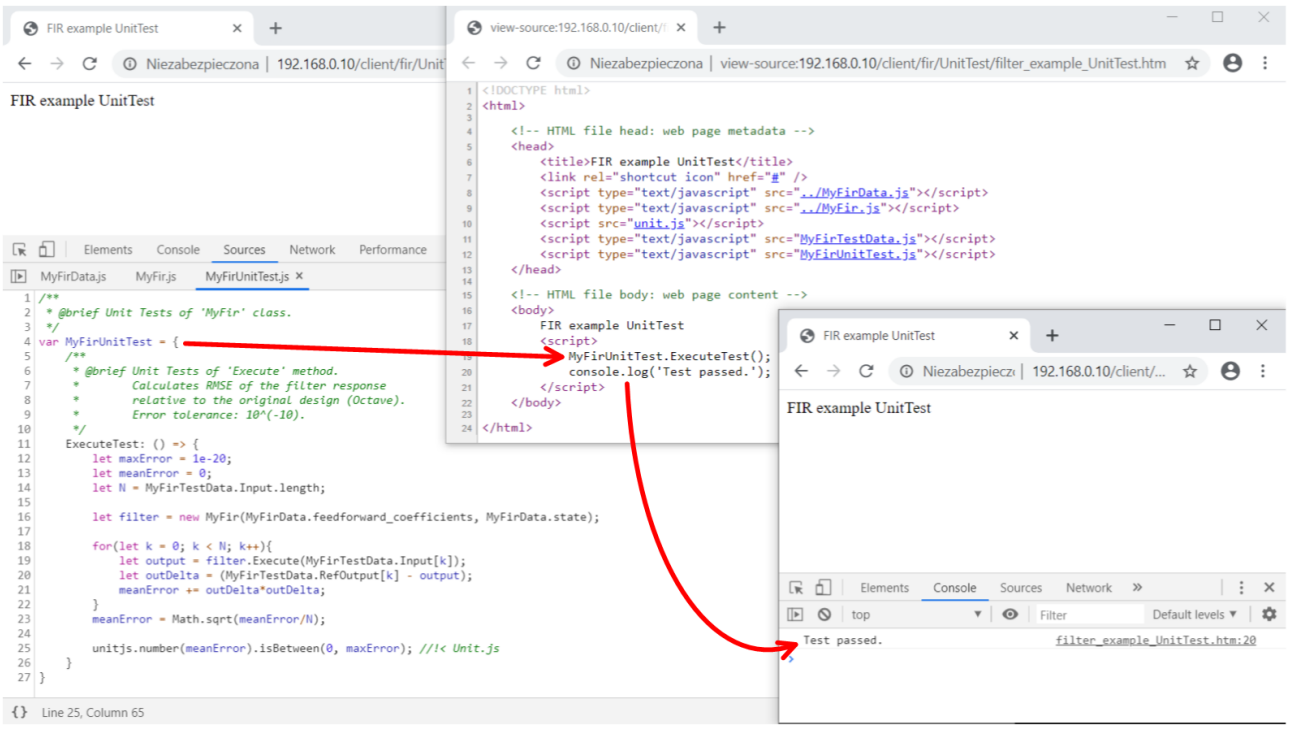

Fig. 5. Unit test of the `MyFir` class (JavaScript) in Chrome browser completed successfully.

3. FIR filter unit test in C#

Listing 8. shows unit test code for FIR filter written in Java. Test was performed with MSTest V2 framework and Visual Studio IDE [13]. In fig. 6. is presented user interface of the environment after a successful test.

Listing 8. Unit test code for the `MyFir` class (C#) in Visual Studio IDE.

```
/**
  * @brief Unit Tests of 'MyFir' class.
  */
[TestClass()]
public class MyFirTests
{
  /**
   * @brief Unit Tests of 'Execute' method.
   *        Calculates RMSE of the filter response
   *        relative to the original design (Octave).
   *        Error tolerance: 10^(-10).
   */
  [TestMethod()]
  [Timeout(100)]  // [ms]
  public void ExecuteTest()
  {
    double maxError = 1e-10;
    double meanError = 0;
    int N = MyFirTestData.Input.Length;

    MyFir filter = new MyFir(MyFirData.feedforward_coefficients,MyFirData.state);

    for (int k = 0; k < N; k++)
    {
      double output = filter.Execute(MyFirTestData.Input[k]);
      double outDelta = (MyFirTestData.RefOutput[k] - output);
      meanError += outDelta * outDelta;
    }
    meanError = Math.Sqrt(meanError/N);
    Assert.AreEqual(0.0, meanError, maxError); //!< MSTest V2
  }
}
```

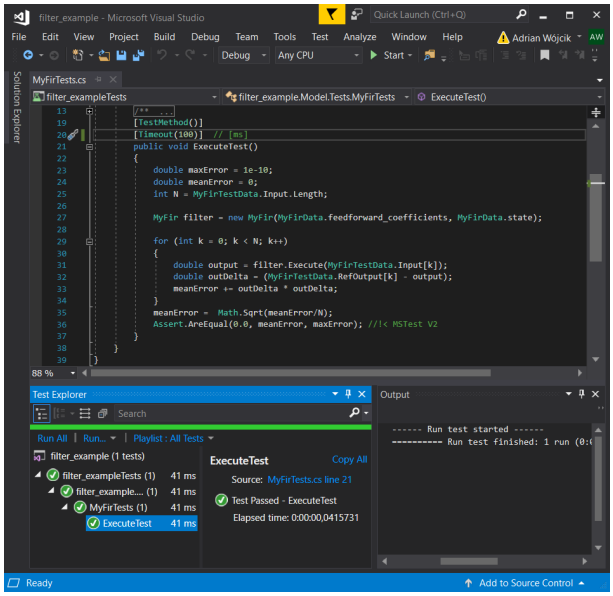

Fig. 6. Unit test of the `MyFir` class (C#) in Visual Studio IDE completed successfully.

e) Usage of DSP in demo applications

After performing successful unit tests, we can proceed to integrate DSP algorithm with other system modules. As part of this example, three demo applications were implemented that integrate FIR filter with the user interface and network communication. User interfaces include a time series graph showing the original test signal and the signal after filtration (updated in real time), information about the sampling frequency and a button to start the demo. Each application allows you to choose between the *mock* server an actual server application using the radio buttons. *Mock* server is simply a test signal generating function while actual server application generates an identical signal with CGI script.

Procedure for using filter in all presented environments has following steps:

1. Obtaining the current sample of test signal based on the internal time counter, depending on the user's choice, from *mock* or server application.
2. `Execute` function call from filter object.
3. Charts update.
4. Time counter update.

1. Demo of mobile application with the FIR filter

In listing 9. code of filtration procedure for demo mobile application for Android is presented. This procedure is called by a timer with a sampling period of $t_s = 100$ ms. The `runOnUiThread` function was used to avoid parallel access to the chart collection in case of delays in network communication. To execute requests in blocking (synchronous) mode, the class `RequestFuture` from the library Volley was used [14]. GraphView [15] was used to create the chart. In fig. **??**. application GUI is shown.

Listing 9. Integration of DSP algorithm with a mobile client application.

```
01. /** @brief Demo of signal filtering procedure using the FIR filter. */
02. private void  FilterProcedure() {
03.   if (k <= sampleMax) {
04.     // get signal
05.     final Double x;
06.     if (signalMock) {
07.       // from mock object
08.       x = serverMock.getTestSignal(k);
09.     } else {
10.       // from server
11.       x = server.getTestSignal(k);
12.     }
13.     // filter signal
14.     final Double xf = filter.Execute(x);
15.     // display data (GraphView)
16.     runOnUiThread(()→{
17.       signal[0].appendData(new DataPoint(k*sampleTime,x), false, sampleMax);
18.       signal[1].appendData(new DataPoint(k*sampleTime,xf), false, sampleMax);
19.       chart.onDataChanged(true, true);
20.     });
21.     // update time
22.     k++;
23.   } else {
24.     filterTimer.cancel();
25.     filterTimer = null;
26.   }
27. }
```

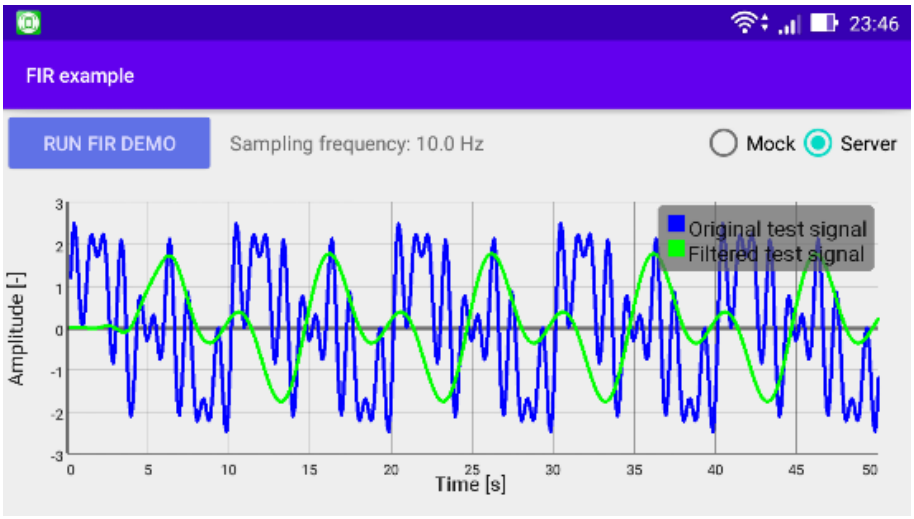

Fig. 7. GUI of demo mobile application.

2. Demo of web application with the FIR filter

In listing 10. code of filtration procedure for demo web application running in the Chrome browser is presented. This procedure is called by a timer with a sampling period of $t_s = 100$ ms. In order to synchronize filtration process with network communication, the `async` modifier and `await` operator were used [16]. The jQuery library was used for network communication [17]. Chart.js was used to create the chart [18]. In fig. 8. application GUI is shown.

Listing 10. Integration of DSP algorithm with a web client application.

```
/** @brief Demo of signal filtering procedure using the FIR filter. */
async function FilterProcedure() {
  if( k <= samplesMax ){
    // get signal
    if(signalLocal) {
      // from TestSignal object
      x = serverMock.getTestSignal(k);
    } else {
      // from server
      x = await server.getTestSignal(k)
    }
    // filter signal
    xf = filter.Execute(x);
    // display data (Chart.js)
    signal[1].push(x);
    signal[0].push(xf);
    chart.update();
    // update time
    k++;
  } else {
    clearInterval(filterTimer);
    filterTimer = null;
  }
}
```

Fig. 8. GUI of demo web application.

3. Demo of desktop application with the FIR filter

In listing 11. code of filtration procedure for demo Windows desktop application is presented. This procedure is called by a timer (dedicated for Windows) with a sampling period of $t_s = 100$ ms. In order to synchronize filtration process with network communication, the `async` modifier and `await` operator were used [19]. The `HttpClient` class was used for network communication [20]. OxyPlot was used to create the chart [21]. In fig. 9. application GUI is shown.

Listing 11. Integration of DSP algorithm with a desktop client application.

```
/** @brief Demo of signal filtering procedure using the FIR filter. */
private async void FilterProcedure(object sender, EventArgs e)
{
  if (k <= samplesMax) {
    // get signal
    double x;
    if (signalMock) {
      // from mock object
      x = serverMock.getTestSignal(k);
    } else {
      // from server
      x = await server.getTestSignal(k);
    }
    // filter signal
    double xf = filter.Execute(x);
    // display data (OxyPlot)
    (chart.Series[0] as LineSeries).Points.Add(new DataPoint(k*sampleTime,x));
    (chart.Series[1] as LineSeries).Points.Add(new DataPoint(k*sampleTime,xf));
    chart.InvalidatePlot(true);
    // update time
    k++;
  } else {
    filterTimer.Stop();
    filterTimer = null;
  }
}
```

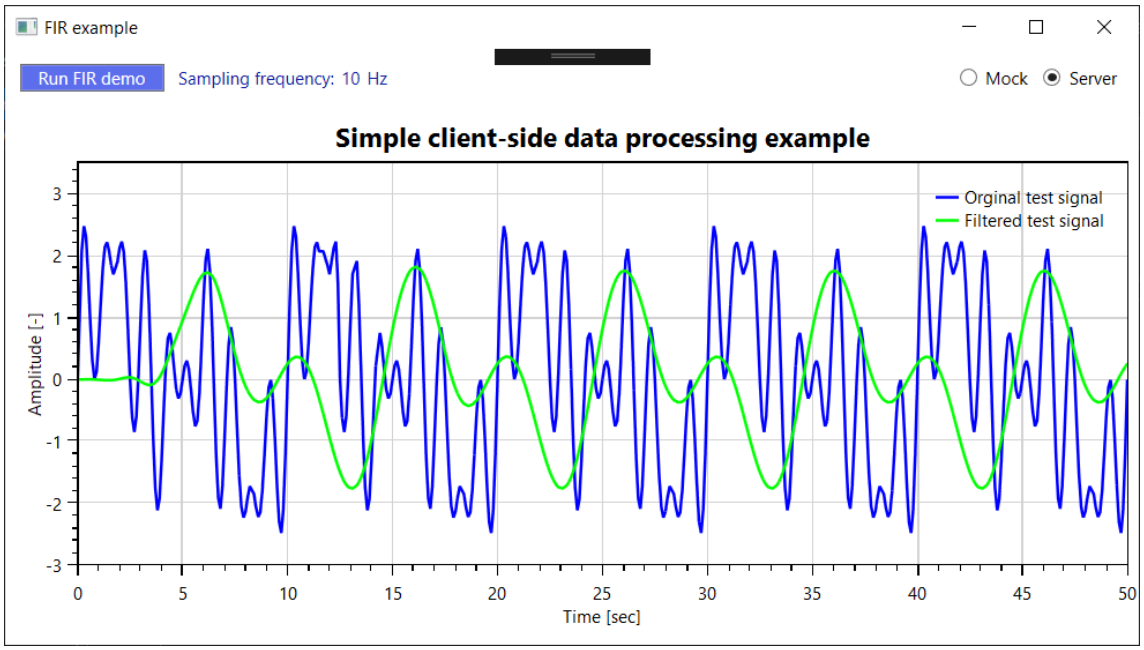

Fig. 9. GUI of demo desktop application.

IV. Scenario for the class

A) Teaching resources

Hardware	• computer,	
	• mobile device with Android OS,	(optional)
	• single-board computer Raspberry Pi,	(optional)
Software	• text editor (e.g. Notepad++, Visual Studio Code),	
	• Python interpreter,	
	• GNU Octave,	(optional)
	• web server (eg. Lighttpd, Apache),	(optional)
	• Android Studio IDE,	(optional)
	• web browser (e.g. Chrome, Firefox),	(optional)
	• Visual Studio IDE,	(optional)

B) Tasks

Familiarize yourself with presented examples of DSP design and implementation. Based on the proposed procedure, implement filter with an infinite impulse response (IIR filter) for the IoT server application written in Python. Assume the sampling period $t_s = 10$ ms.

1. Write a script that generates IIR filter parameters and unit test data. Use e.g. the GNU Octave environment or the SciPy library. Use library functions. The filter should pass frequencies less than 10 Hz and stop frequencies greater than 15 Hz. Adopt a bandwidth attenuation of at least 60 dB. Use any type of IIR filter (Chebyshev type I or type II, Butterworth, Bessel, etc.). To obtain data for the unit test, generate an example test signal and filter it.

2. Write your own implementation of digital IIR filter. in Python Do not use library functions - the implementation should be based only on loops, addition, multiplication and operations on arrays.

3. Write a unit test of your own implementation of the IIR filter using e.g. the library `unittest` [22]. Use previously generated data and filter parameters. Perform the test and make sure it is done correctly.

4. **(*)** Integrate the algorithm with the IoT system. Write an application containing measurement signal filtration procedure. The measurement signal can come from *mock* or one of the Sense Hat sensors (emulator or physical device). Use timer for synchronization. Application in each step should add: sample number, measurement signal value, filtered signal value to text file. The program should act as a (*daemon*), not as a CGI script.

References

1. *Regulations, health and safety instructions* [online]. [N.d.] [visited on 2019-09-30]. Available from: http://zsep.cie.put.poznan.pl/materialy-dydaktyczne/MD/Regulations-health-and-safety-instructions/.
2. *GNU Octave* [online]. [N.d.] [visited on 2020-05-13]. Available from: https://www.gnu.org/software/octave/. Library Catalog: www.gnu.org.
3. *SciPy.org* [online]. [N.d.] [visited on 2020-05-13]. Available from: https://www.scipy.org/.
4. MIDDLESTEAD, Richard W. *Digital Communications with Emphasis on Data Modems: Theory, Analysis, Design, Simulation, Testing, and Applications*. John Wiley & Sons, 2017. ISBN 978-0-470-40852-0. Google-Books-ID: RxsjDgAAQBAJ.
5. *Function Reference: fir1* [online]. [N.d.] [visited on 2020-05-13]. Available from: https://octave.sourceforge.io/signal/function/fir1.html.
6. *scipy.signal.firwin — SciPy v1.4.1 Reference Guide* [online]. [N.d.] [visited on 2020-05-13]. Available from: https://docs.scipy.org/doc/scipy/reference/generated/scipy.signal.firwin.html.
7. *Window function* [online]. 2020 [visited on 2020-05-13]. Available from: https://en.wikipedia.org/w/index.php?title=Window_function&oldid=951701883. Page Version ID: 951701883.
8. *Function Reference: filter* [online]. [N.d.] [visited on 2020-05-13]. Available from: https://octave.sourceforge.io/octave/function/filter.html.
9. *scipy.signal.lfilter — SciPy v1.4.1 Reference Guide* [online]. [N.d.] [visited on 2020-05-14]. Available from: https://docs.scipy.org/doc/scipy/reference/generated/scipy.signal.lfilter.html.
10. *Introduction to Test Driven Development (TDD)* [online]. [N.d.] [visited on 2020-05-13]. Available from: http://agiledata.org/essays/tdd.html.
11. *Build local unit tests* [online]. [N.d.] [visited on 2020-05-13]. Available from: https://developer.android.com/training/testing/unit-testing/local-unit-tests. Library Catalog: developer.android.com.
12. *Unit testing framework for Javascript - Unit JS* [online]. [N.d.] [visited on 2020-05-14]. Available from: https://unitjs.com/.
13. NCARANDINI. *Testowanie jednostkowe C# z MSTest i .NET Core - .NET Core* [online]. [N.d.] [visited on 2020-05-13]. Available from: https://docs.microsoft.com/pl-pl/dotnet/core/testing/unit-testing-with-mstest. Library Catalog: docs.microsoft.com.
14. *Volley overview* [Android Developers] [online] [visited on 2021-04-12]. Available from: https://developer.android.com/training/volley.
15. GEHRING, Jonas. *jjoe64/GraphView* [online]. 2021 [visited on 2021-04-18]. Available from: https://github.com/jjoe64/GraphView. original-date: 2011-07-04T13:25:41Z.
16. *funkcja async* [online]. [N.d.] [visited on 2020-05-13]. Available from: https://developer.mozilla.org/pl/docs/Web/JavaScript/Referencje/Polecenia/funkcja_async. Library Catalog: developer.mozilla.org.

17. JS.FOUNDATION, JS Foundation-. *jQuery* [online]. [N.d.] [visited on 2020-05-13]. Available from: https://jquery.com/. Library Catalog: jquery.com.
18. *Chart.js | Open source HTML5 Charts for your website* [online]. [N.d.] [visited on 2020-05-13]. Available from: https://www.chartjs.org/.
19. BILLWAGNER. *asynchronizowania — odwołanie do języka C#* [online]. [N.d.] [visited on 2020-05-13]. Available from: https://docs.microsoft.com/pl-pl/dotnet/csharp/language-reference/keywords/async. Library Catalog: docs.microsoft.com.
20. *HttpClient Klasa (System.Net.Http)* [online]. [N.d.] [visited on 2020-05-06]. Available from: https://docs.microsoft.com/pl-pl/dotnet/api/system.net.http.httpclient. Library Catalog: docs.microsoft.com.
21. *OxyPlot* [online]. [N.d.] [visited on 2020-05-13]. Available from: https://github.com/oxyplot. Library Catalog: github.com.
22. *unittest — Unit testing framework — Python 3.8.3rc1 documentation* [online]. [N.d.] [visited on 2020-05-13]. Available from: https://docs.python.org/3/library/unittest.html.