TECHNIKI PROGRAMOWANIA – projekt 3

Wiktor Kyć 203572, Igor Petelski 203871, ACiR, gr. III

O PROJEKCIE

Projekt implementuje bibliotekę C++ do przetwarzania sygnałów jako moduł pythonowy przy użyciu pybind11. Poniższy plik zawiera opis i wizualizację wszystkich funkcjonalności podstawowych, tj.:

- wizualizacji sygnałów z wykorzystaniem biblioteki matplot++;
- DFT i transformacji odwrotnej;
- filtracji 1D i 2D;
- generowania sygnałów o zadanej częstotliwości (sinusoidalny, cosinusoidalny, prostokątny, piłokształtny);

a także odpowiedniej dla numeru grupy projektowej (4), wymagania dodatkowego, tj.

- wykrywania piku w sygnale dowolną metodą.

Opisy poszczególnych funkcjonalności i ich wizualizacje przedstawione są w dalszej części raportu.

1) GENEROWANIE SYGNAŁÓW O ZADANEJ CZĘSTOTLIWOŚCI

Program umożliwia generowanie czterech rodzajów sygnałów: sinusoidalnego, cosinusoidalnego, prostokątnego i piłokształtnego. Każda z generowanych funkcji miała następujące parametry:

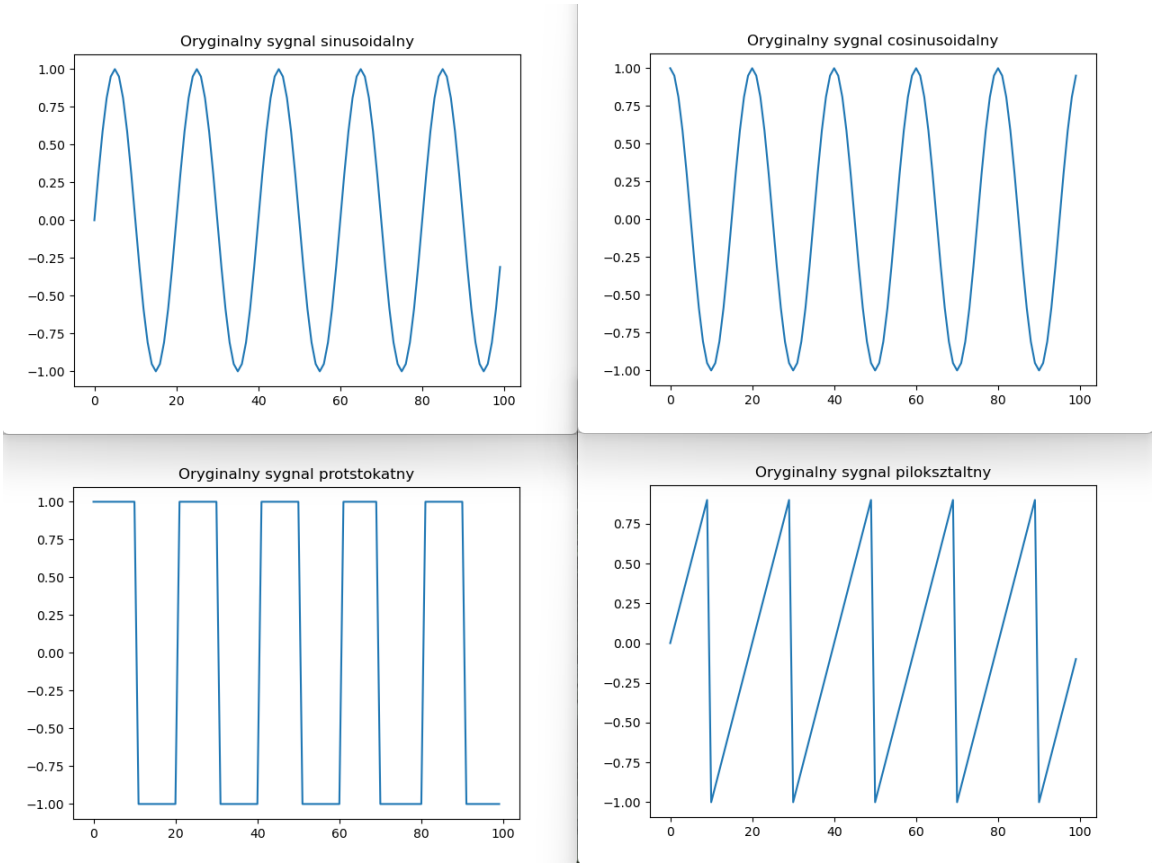
- liczba próbek (N) = 100;
- częstotliwość (f) = 5 Hz;
- czas początkowy (t0) = 0 s;
- czas końcowy (t1) = 1 s;
- amplituda (A) = 1;

Za wygenerowanie odpowiednich rodzajów sygnałów odpowiadały funkcje „generuj_*rodzaj sygnału*” zaimplementowane w pliku „przetwarzanie_sygnalu.cpp”. Przykład funkcji generującej sygnał sinusoidalny:

```
vector<double> generuj_sinus(int N, double f, double t0, double t1, double A)
{
    vector<double> s(N);
    double dt = (t1 - t0) / N;
    for (int i = 0; i < N; ++i)
    {
        double t = t0 + i * dt;
        s[i] = A * sin(2 * M_PI * f * t);
    }
    return s;
}
```

2) WIZUALIZACJA SYGNAŁÓW Z WYKORZYSTANIEM BIBLIOTEKI MATPLOT++

Wygenerowane sygnały są następnie wizualizowane z wykorzystaniem biblioteki matplot++:

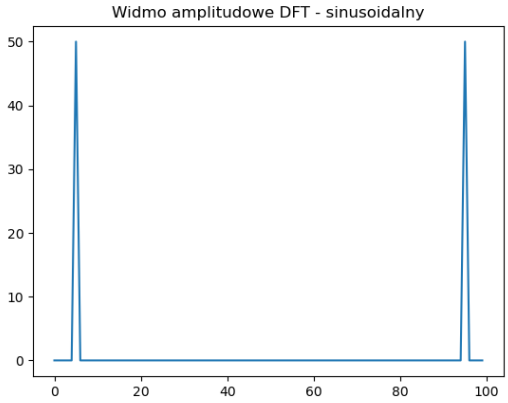


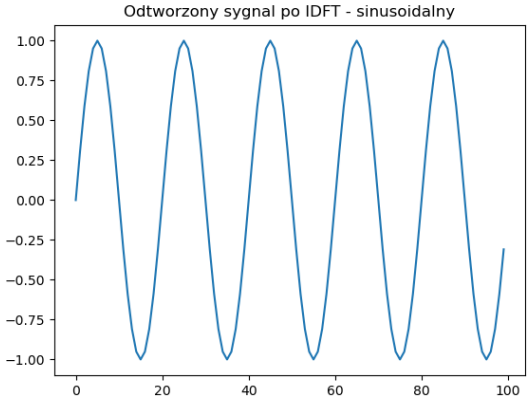
3) DFT I TRANSFORMACJA ODWROTNA

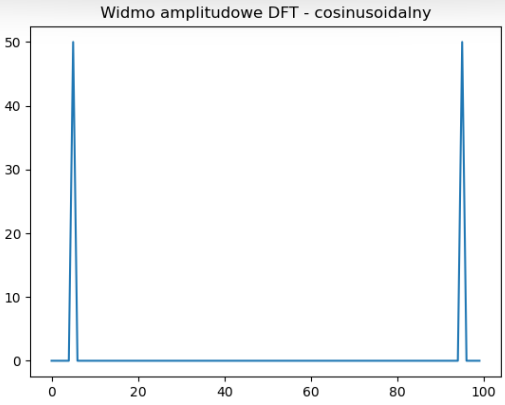
Wymaganie realizują kolejne dwie funkcje zaimplementowane w pliku „przetwarzanie_sygnalu.cpp”:

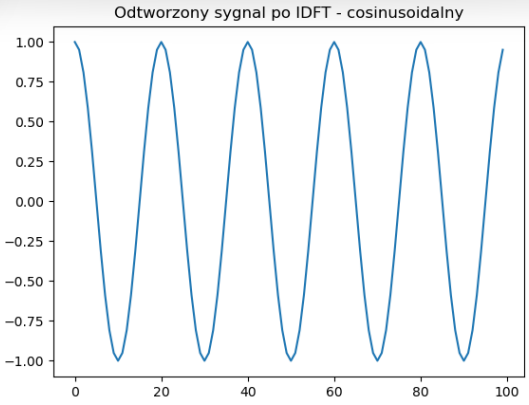
```
vector<complex<double>> dft(const vector<double>& sygnal)
{
    int N = static_cast<int>(sygnal.size());
    vector<complex<double>> X(N);
    for (int k = 0; k < N; ++k)
    {
        complex<double> suma = 0;
        for (int n = 0; n < N; ++n)
        {
            double kat = -2.0 * M_PI * k * n / N;
            suma += sygnal[n] * complex<double>(cos(kat), sin(kat));
        }
        X[k] = suma;
    }
    return X;
}

vector<complex<double>> idft(const vector<complex<double>>& widmo)
{
    int N = static_cast<int>(widmo.size());
    vector<complex<double>> x(N);
    for (int n = 0; n < N; ++n)
    {
        complex<double> suma = 0;
        for (int k = 0; k < N; ++k)
        {
            double kat = 2.0 * M_PI * k * n / N;
            suma += widmo[k] * complex<double>(cos(kat), sin(kat));
        }
        x[n] = suma / static_cast<double>(N);
    }
    return x;
}
```

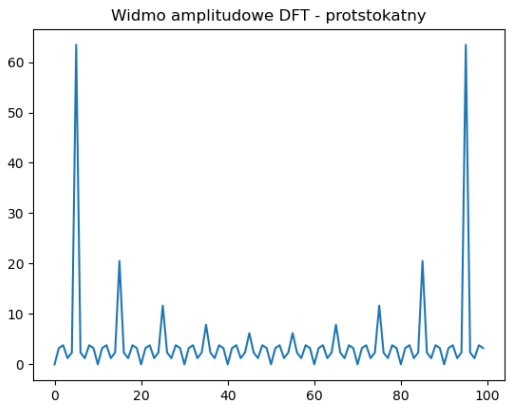
Otrzymane widma amplitudowe DFT i odtworzone po IDFT sygnały, identyczne, jak oryginalne, prezentowane są na wykresach:

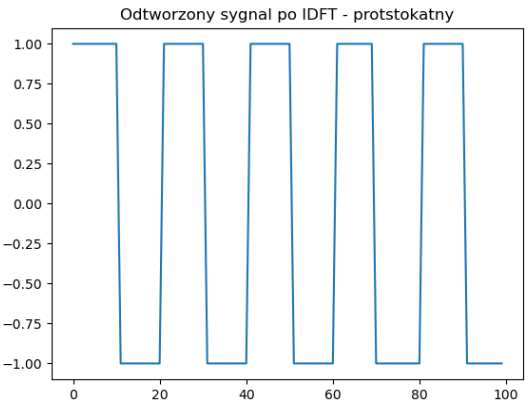

Widmo amplitudowe DFT - sinusoidalny
0
10
20
30
40
50
0
20
40
60
80
100

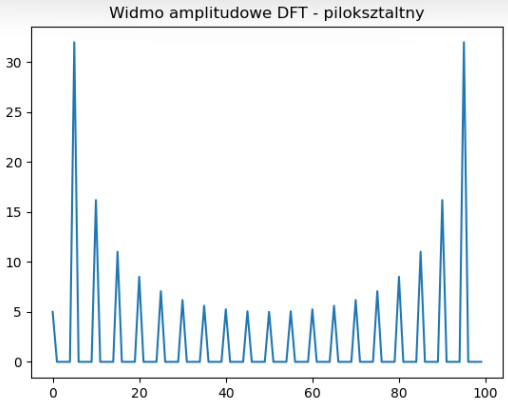

Odtworzony sygnal po IDFT - sinusoidalny
−1.00
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75
1.00
0
20
40
60
80
100

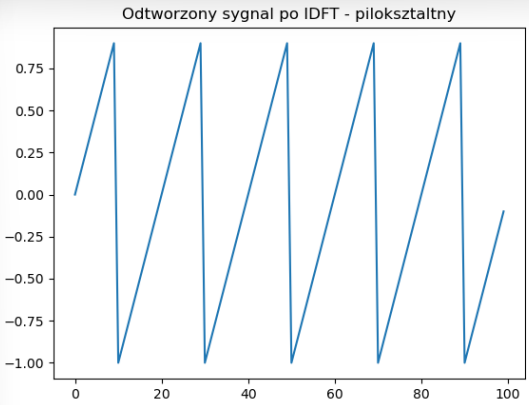

Widmo amplitudowe DFT - cosinusoidalny
0
10
20
30
40
50
0
20
40
60
80
100


Odtworzony sygnal po IDFT - cosinusoidalny
−1.00
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75
1.00
0
20
40
60
80
100


Widmo amplitudowe DFT - protstokatny
0
10
20
30
40
50
60
0
20
40
60
80
100


Odtworzony sygnal po IDFT - protstokatny
−1.00
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75
1.00
0
20
40
60
80
100


Widmo amplitudowe DFT - piloksztaltny
0
5
10
15
20
25
30
0
20
40
60
80
100

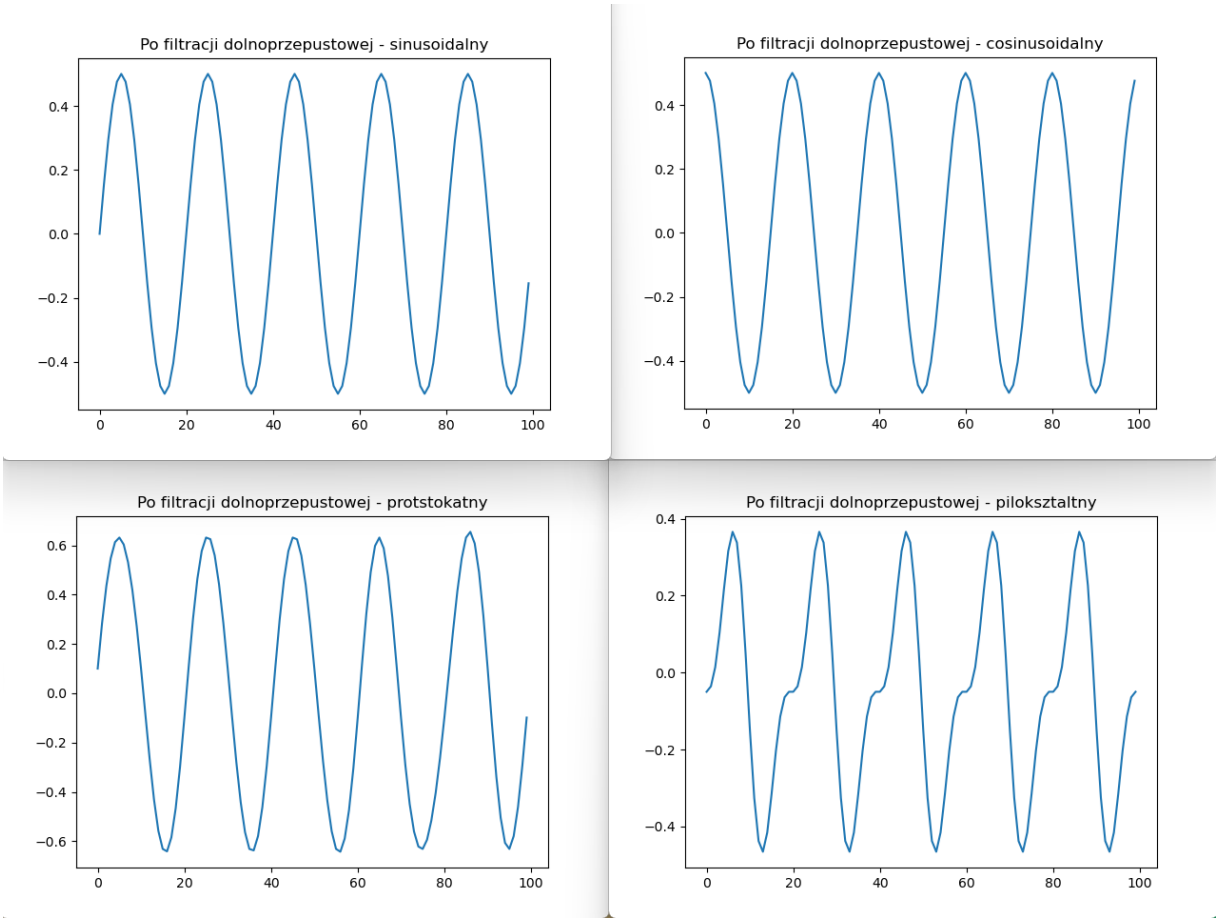

Odtworzony sygnal po IDFT - piloksztaltny
−1.00
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75
0
20
40
60
80
100

4) FILTRACJA 1D

Za realizację tego wymagania odpowiada funkcja filtruj_dolnoprzepustowo zaimplementowana w pliku „przetwarzanie_sygnalu.cpp”:

```cpp
vector<complex<double>> filtruj_dolnoprzepustowo(const vector<complex<double>>& widmo, double prog)
{
    int N = static_cast<int>(widmo.size());
    vector<complex<double>> filtrowane(N);
    for (int k = 0; k < N; ++k) {
        double czestotliwosc = static_cast<double>(k) / N;
        if (czestotliwosc <= prog) {
            filtrowane[k] = widmo[k];
        } else {
            filtrowane[k] = 0;
        }
    }
    return filtrowane;
}
```

Wizualizację sygnałów po filtracji 1D można przeanalizować na odpowiednich wykresach:

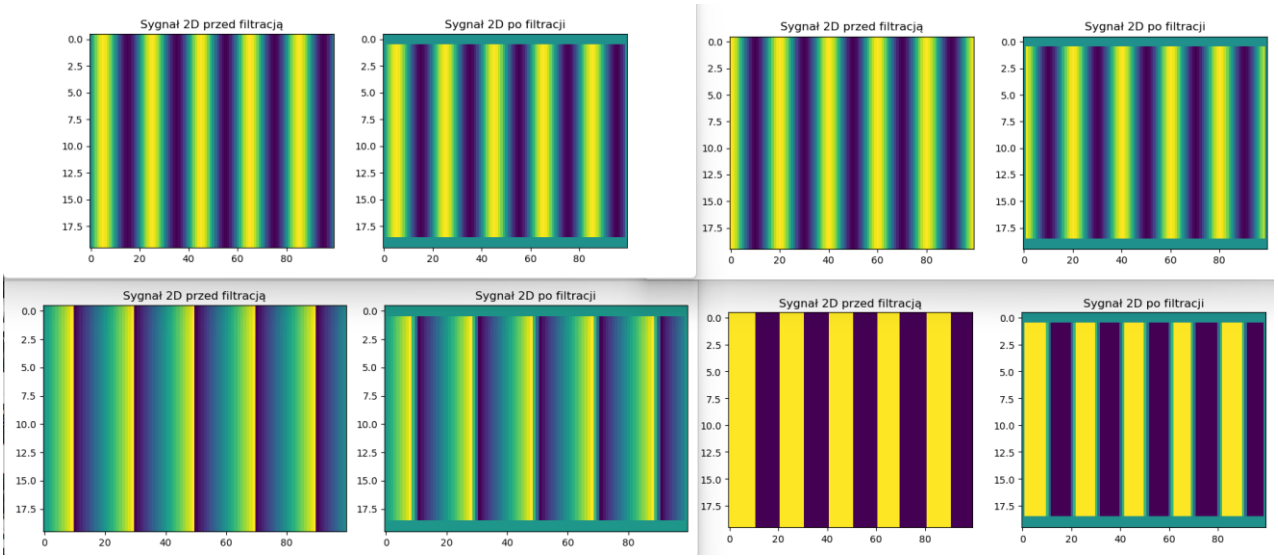


5) FILTRACJA 2D

Kolejne z wymagań realizowane przez odpowiednią funkcję z pliku „przetwarzanie sygnalu.cpp”:

```cpp
vector<vector<double>> filtruj_2d_srednia(const vector<vector<double>>& obraz)
{
    int rows = obraz.size();
    int cols = obraz[0].size();
    vector<vector<double>> wynik(rows, vector<double>(cols, 0.0));

    for (int i = 1; i < rows - 1; ++i)
    {
        for (int j = 1; j < cols - 1; ++j)
        {
            double suma = 0.0;
            for (int di = -1; di <= 1; ++di)
            {
                for (int dj = -1; dj <= 1; ++dj)
                {
                    suma += obraz[i + di][j + dj];
                }
            }
            wynik[i][j] = suma / 9.0;
        }
    }
    return wynik;
}
```

Wynik wizualizacji otrzymanych w ten sposób sygnałów obserwujemy na poniższych wykresach (od lewego górnego rogu, zgodnie z ruchem wskazówek zegara sygnały: sinusoidalny, cosinusoidalny, prostokątny i piłokształtny):

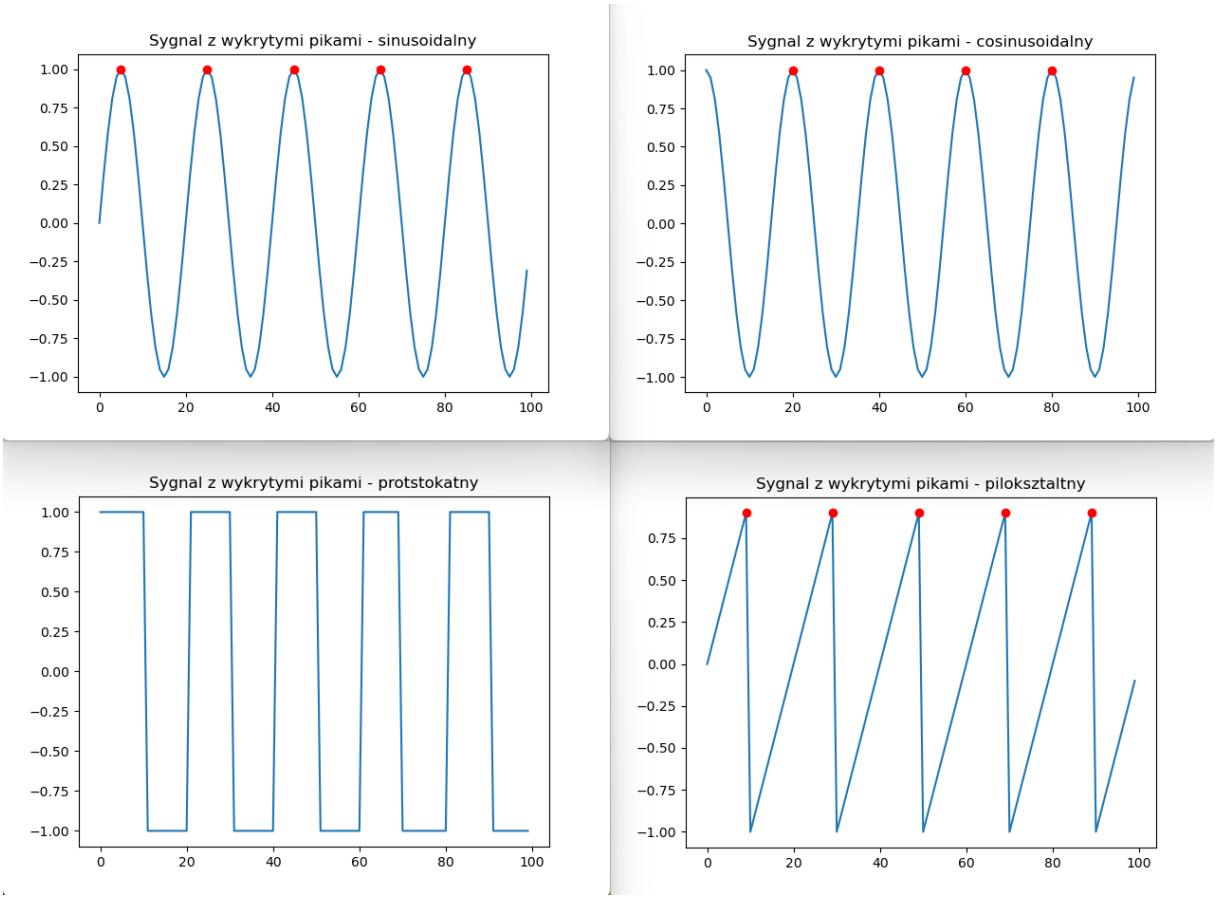


6) WYKRYWANIE PIKU W SYGNALE

Za wymaganie odpowiada stosowna funkcja w pliku „piki.cpp”:

```cpp
vector<size_t> znajdz_piki(const vector<double>& sygnal, double prog, size_t min_odleglosc)
{
    vector<size_t> piki;
    size_t ostatni = -min_odleglosc - 1;
    for (size_t i = 1; i + 1 < sygnal.size(); ++i)
    {
        if (sygnal[i] > prog && sygnal[i] > sygnal[i - 1] && sygnal[i] > sygnal[i + 1] && (i - ostatni > min_odleglosc))
        {
            piki.push_back(i);
            ostatni = i;
        }
    }
    return piki;
}
```

Oryginalne sygnały z zaznaczonymi na wykresie „pikami” przedstawione są poniżej:



PODSUMOWANIE

Projekt spełnia wszystkie funkcjonalności podstawowe, jak i jedną z funkcjonalności dodatkowych, zgodną z numerem grupy projektowej. Wszystkie z nich zostały omówione w niniejszym raporcie. Wszystkie funkcje są zaimplementowane w C++ i dostępne z poziomu pliku „test.py” dzięki pybind11.