

POLITECHNIKA POZNAŃSKA

WYDZIAŁ AUTOMATYKI, ROBOTYKI I ELEKTROTECHNIKI

INSTYTUT ROBOTYKI I INTELIGENCJI MASZYNOWEJ

ZAKŁAD STEROWANIA I ELEKTRONIKI PRZEMYSŁOWEJ



PRACA DYPLOMOWA MAGISTERSKA

MIKROPROCESOROWY INKLINOMETR  
OPRACOWANY Z WYKORZYSTANIEM  
AKCELEROMETRU MEMS

ALIAKSANDR HILEVICH

PROMOTOR:

DR INŻ. DOMINIK ŁUCZAK

DOMINIK.LUCZAK@PUT.POZNAN.PL

POZNAŃ 2021



## STRESZCZENIE

Praca dotyczy badania opracowania mikroprocesowego inklinometru z czujnikiem MEMS oraz badania algorytmów filtracji z surowych danych.

Do badań wybrano następujące algorytmy filtrowania danych:

- filtr komplementarny,
- filtr Madgwicka,
- filtr Kalmana.

W pierwszej części pracy się opowiada o opracowaniu algorytmów w Matlab i następnie weryfikacji ich pracy na danych syntetycznych.

Głównym punktem pracy jest implementacja algorytmów w układzie rzeczywistym opartym na płycie NUCLEO-F756GZ z mikrokontrolerem STM32F7. Dane surowe zbierane za pomocą czujnika MEMS LSM6DSOX.

## ABSTRACT

The work concerns the study of the development of a microprocessor inclinometer with a MEMS sensor and the study of raw data filtration algorithms.

The following data filtering algorithms were selected for the research:

- Complementary filter,
- Madgwick filter,
- Kalman filter.

The first part of the thesis tells about the development of algorithms in Matlab and then the verification of their work on synthetic data.

The main point of work is the implementation of the algorithms in a real system based on the NUCLEO-F756GZ board with the STM32F7 microcontroller. Raw data collected with the MEMS LSM6DSOX sensor.

## Spis treści

1	Wstęp.....	6
2	Wyznaczenie kątów nachylenia. Filtry.....	7
2.1	Filtr komplementarny .....	9
2.2	Filtr Madgwicka .....	10
2.3	Filtr Kalmana. Dyskretny algorytm.....	12
3	Implementacja algorytmów w środowisku Matlab wraz z Weryfikacją na zbiorze danych. ....	14
3.1	Filtr komplementarny .....	16
3.2	Filtr Madgwicka .....	17
3.3	Filtr Kalmana.....	18
4	Weryfikacja praktyczna.....	21
4.1	Mikrokontroler STM32F756ZG.....	21
4.2	STEVAL-MKI197V1A z czujnikiem MEMS LSM6DOX.....	21
4.3	Schemat połączenia .....	22
4.4	Oprogramowanie .....	23
4.5	Konfiguracja w STM32CubeMX.....	24
4.6	Zebrańie surowych danych z czujnika. Główny program .....	26
4.7	Odczyt danych przez port COM.....	28
5	Weryfikacja pracy algorytmów z wykorzystaniem surowych danych.....	29
5.1	Dane surowe .....	29
5.2	Wyniki badań .....	30
6	Implementacja algorytmów w układzie rzeczywistym .....	32
6.1	Implementacja filtru komplementarnego .....	34
6.2	Implementacja filtru Kalmana.....	35
6.3	Implementacja filtru Madgwicka .....	34
6.4	wyniki.....	37
7	Podsumowanie.....	40
	Literatura .....	41

# 1 WSTĘP

Czujniki mikroelektromechaniczne (MEMS, ang. Microelectromechanical systems) prędkości kątowej (żyroskopy) i przyspieszenia (akcelerometry) stanowią jeden z aktywnie rozwijających się obszarów technologii mikrosystemów. Czujniki MEMS to zintegrowane systemy o rozmiarach od kilku mikrometrów do kilku milimetrów, które łączą komponenty mechaniczne i elektroniczne. Zasada działania takich czujników polega na zamianie na sygnał elektryczny pojemności różnicowej utworzonej przez ruchome i nieruchome mikromechaniczne płytki grzebieniowe. Zmiana pojemności pod wpływem przyspieszenia liniowego (w akcelerometrach) lub siły Coriolisa (w żyroskopach) umożliwia oszacowanie wartości amplitudy tych wpływów.

Pomimo niewielkich rozmiarów, wagi i zużycia energii, praktyczne zastosowanie czujników MEMS w systemach orientacji i nawigacji jest ograniczone przez niską czułość, niestabilność współczynnika skali oraz wysoki poziom szumu sygnału wyjściowego w porównaniu z innymi typami żyroskopów i akcelerometrów. Spośród istniejących typów żyroskopów MEMS mają największy dryf, co nie pozwala na ich użycie bez okresowego filtrowania współrzędnych kątowych.

Celem pracy jest porównanie algorytmów filtrowania sygnałów z inercyjnych czujników MEMS w celu wyznaczenia współrzędnych kątowych obiektów obracających się w trzech płaszczyznach.

Do badań wybrano następujące algorytmy filtrowania danych:

- filtr komplementarny,
- filtr Madgwicka,
- filtr Kalmana.

Układ rzeczywisty jest oparty na płytce NUCELO-F756GZ z mikrokontrolerem STM32F7. Do zbierania danych surowych będzie wykorzystany trzyosiowy czujnik MEMS LSM6DSOX.

## 2 WYZNACZENIE KĄTÓW NACHYLENIA. FILTRY

### Żyroskop

Urządzenie zdolne do reagowania na zmiany kątów orientacji obiektu, na którym jest zainstalowane, względem bezwładnościowego układu odniesienia.

W przypadku żyroskopu kąt nachylenia można łatwo obliczyć poprzez całkowania jego prędkości obrotowej. Metodę całkowania naiwnego.

$$\alpha(t) = \alpha(t - 1) + G_x * dt \quad (2.1)$$

$$\beta(t) = \beta(t - 1) + G_y * dt \quad (2.2)$$

$$\gamma(t) = \gamma(t - 1) + G_z * dt \quad (2.3)$$

gdzie:

$\alpha(t)$  - kąt nachylenia w osi X

$\beta(t)$  - kąt nachylenia w osi Y

$\gamma(t)$  - kąt nachylenia w osi Z

$\alpha(t - 1)$  - kąt nachylenia w osi X w poprzednim momencie

$\beta(t - 1)$  - kąt nachylenia w osi Y w poprzednim momencie

$\gamma(t - 1)$  - kąt nachylenia w osi Z w poprzednim momencie

$G_x$  - prędkość obrotowa wokół osi X

$G_y$  - prędkość obrotowa wokół osi Y

$G_z$  - prędkość obrotowa wokół osi Z

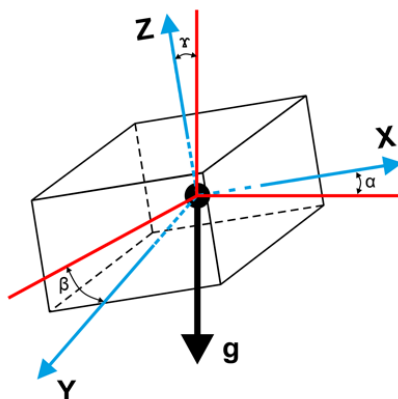
$dt$  – czas próbkowania

Żyroskop MEMS ma jedną wadę, którą nazywa się dryfem zerowym. Istota tej wady sprowadza się do tego, że gdy obrót żyroskopu się zatrzyma, nadal będzie pokazywał wartość inną niż zero. Inną wadą jest stosowanie procedury całkowania dyskretnego, która ze swej natury daje niedokładny wynik, co prowadzi do stopniowej kumulacji błędów obliczeniowych kątów ze względu na ograniczoną dokładność zmiennych mikrokontrolera.

Wyeliminowanie dryfu żyroskopowego jest ważnym zadaniem przy określaniu kąta pochylenia. Aby częściowo skompensować dryf, zastosujemy prostą, ale skuteczną metodę. Na danych testowych zebranych podczas postoju systemu uśredniamy wartości w każdej osi. Te wartości będą przesunięciami ich osi.

## Akcelerometr

Akcelerometr to urządzenie, które mierzy przyspieszenie pod wpływem sił zewnętrznych. Za pomocą tego czujnika można również obliczyć kąty  $\alpha, \beta, \gamma$  (Rys. 1).



Rys.1 Orientacja obiektu w 3 osiach [12]

$$A_x = g * \sin(\alpha) \rightarrow \alpha = \arcsin\left(\frac{A_x}{g}\right) \quad (2.4)$$

$$A_y = g * \sin(\beta) \rightarrow \beta = \arcsin\left(\frac{A_y}{g}\right) \quad (2.5)$$

$$A_z = g * \sin(\gamma) \rightarrow \gamma = \arcsin\left(\frac{A_z}{g}\right) \quad (2.6)$$

$g$  - przyspieszenie grawitacyjne

$A_x$  - rzut przyspieszenia grawitacyjnego na oś akcelerometru X.

$A_y$  - rzut przyspieszenia grawitacyjnego na oś akcelerometru Y.

$A_z$  - rzut przyspieszenia grawitacyjnego na oś akcelerometru Z.

Przy tej metodzie obliczeń występuje problem z pełnym pokryciem kąta 360, sygnał wyjściowy akcelerometru jest taki sam dla kątów  $\alpha$  i  $\pi - \alpha$ . Dlatego zakres pomiarowy wynosi 180 stopni. Dlatego zaleca się stosowanie stosunku odczytów akcelerometru i obliczanie kąta pochylenia za pomocą funkcji arctangens.

$$\alpha = \arctan\left(\frac{A_x}{\sqrt{A_y^2 + A_z^2}}\right) \quad (2.7)$$

$$\beta = \arctan\left(\frac{A_y}{\sqrt{A_x^2 + A_z^2}}\right) \quad (2.8)$$



$$\gamma = \arctan\left(\frac{A_z}{\sqrt{A_x^2 + A_y^2}}\right) \quad (2.9)$$

$A_x, A_y, A_z$  to wartości zwracane przez czujnik.

Jak w przypadku każdego systemu elektromechanicznego, czynniki takie jak hałas, wibracje, przemieszczenie, dryf temperaturowy itp. wpływają na jakość pomiarów akcelerometru. Szum akcelerometru jest spowodowany szumem elektroniki, wahaniami napięcia, błędami ADC. Naturalny biały szum jest reprezentowany przez gęstość szumu w specyfikacji czujnika. Nawet jeśli nie ma własnego szumu, czujnik może doświadczyć zewnętrznych wibracji, które psują sygnał. Dokładność pomiaru można poprawić, uśredniając wynik dla  $N$  próbek, ale zwiększenie liczby próbek zwiększy opóźnienie pomiaru.

## 2.1 FILTR KOMPLEMENTARNY

Filtr ten jest nakładany na dwie wielkości mierzone przez różne czujniki i koryguje jedną z nich tak, aby powoli zbliżała się do drugiej. W cyklu pomiarowym filtr realizowany w następujący sposób.

$$\alpha(t) = K * (\alpha(t-1) + G_x * dt) + (1-K) * A_x \quad (2.10)$$

$$\beta(t) = K * (\beta(t-1) + G_y * dt) + (1-K) * A_y \quad (2.11)$$

$$\gamma(t) = K * (\gamma(t-1) + G_z * dt) + (1-K) * A_z \quad (2.12)$$

Jak widać ze wzorów, całkowity kąt nachylenia jest sumą wartości zintegrowanego żyroskopu i chwilowej wartości akcelerometru. Głównym zadaniem filtra komplementarnego jest zniwelowanie dryftu zerowego żyroskopu oraz dyskretnych błędów całkowania za pomocą wskazań akcelerometru. Na każdym kroku całkowania (kroku cyklu kontrolnego) korygujemy całkę kąta pochylenia za pomocą wskazań akcelerometru. Siła tej korekty jest określona przez współczynnik filtra  $K$ . Wybór współczynnika  $K$  zależy od wielkości dryftu zerowego żyroskopu, od szybkości akumulacji błędów obliczeniowych oraz od warunków użytkowania maszyna. Np. zbyt duża wartość  $K$  spowoduje, że na wynik działania filtra silnie wpłyną zewnętrzne wibracje. Zbyt mała wartość  $K$  może okazać się niewystarczająca do wyeliminowania dryftu zera żyroskopu. Z reguły współczynnik filtra komplementarnego wybiera się za pomocą wzoru 2.13.

$$K = \frac{\tau}{\tau + dt} \quad (2.13)$$

Gdzie:

$\tau$  - stała filtra czasu

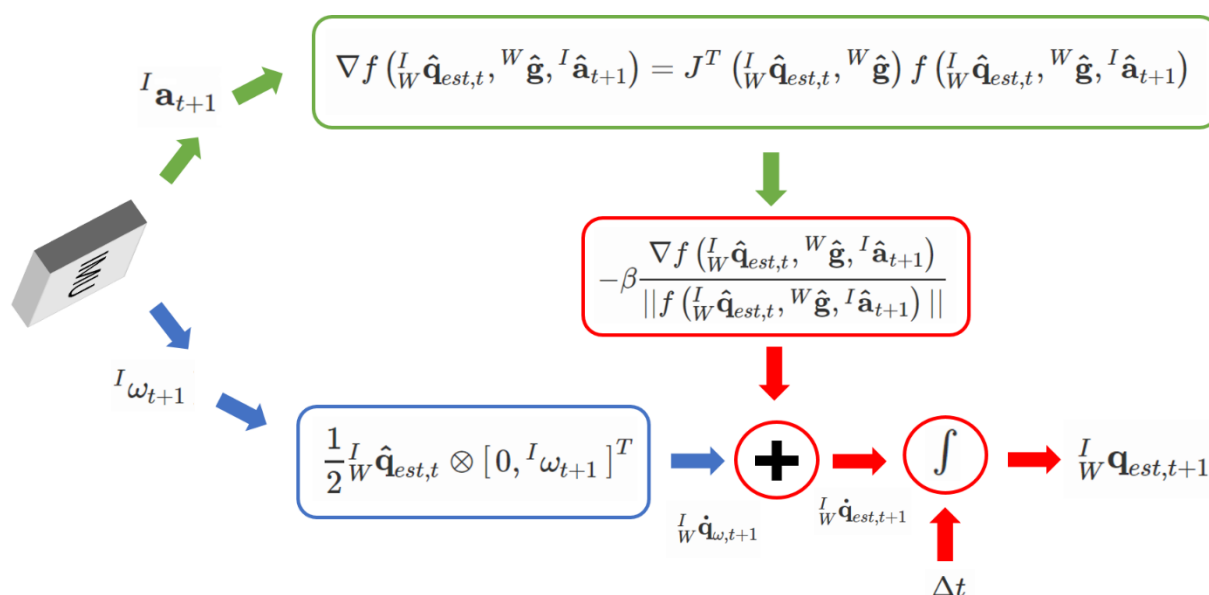
## 2.2 FILTR MADGWICKA

Filtr Madgwicka [1] używa kwaternionów. Kwaterniony to jest czterowymiarowa liczba zespolona, której można użyć do przedstawienia współrzędnych w przestrzeni trójwymiarowej.

Dane z żyroskopu i akcelerometru są przekształcane na pochodne kwaternionów opisujących prędkość czujnika i kierunek pola grawitacyjnego w układzie odniesienia Ziemi względem układu odniesienia czujnika. W celu optymalizacji danych wykorzystuje się jedną z lokalnych metod optymalizacji – metodę spadku gradientu.

System notacji: Indeks dolny z przodu oznacza docelowy układ odniesienia, a indeks górny z przodu wskazuje układ odniesienia, dla którego określona jest zmienna. Na przykład  ${}^S_E\hat{q}$  - opisuje orientację osi E (Ziemi) w stosunku do osi S (czujnika).

Poniżej przedstawiono kroki do szacowania położenia za pomocą filtra Madgwick (Rys. 2).



Rys.2 Filtr Madgwicka [13]

### 1. Określanie orientacji na podstawie danych żyroskopu

#### 1.1. Przekształcony kwaternion prędkości kątowej ma postać:

$$S_w = [0 \ w_x \ w_y \ w_z] \quad (2.14)$$

#### 1.2. Pochodny kwaternion opisujący prędkość w układzie odniesienia Ziemi w odniesieniu do układu odniesienia czujnika

$${}^S_E\dot{\hat{q}}_{w,t} = \frac{1}{2} {}^S_E\hat{q}_{est,t-1} \times S_{w,t} \quad (2.15)$$

#### 1.3. Całkując pochodny kwaternion, znajdujemy orientację w globalnym układzie odniesienia

$${}^S_E q_{w,t} = {}^S_E \hat{q}_{est,t-1} + {}^S_E \dot{q}_{w,t} \Delta t \quad (2.16)$$

${}^S_E \hat{q}_{est,t-1}$  - poprzedni wynik oceny orientacji

$\Delta t$  – czas próbkowania

## 2. Określanie orientacji na podstawie danych akcelerometru

Jeśli kierunek pola Ziemi jest znany w jej układzie współrzędnych, pomiar kierunku w układzie współrzędnych czujnika obliczy położenie układu współrzędnych czujnika względem układu współrzędnych Ziemi.

$$\min_{{}^S_E \hat{q} \in R^4} f({}^S_E \hat{q}_{est,t-1}, {}^E \hat{d}, {}^S \hat{a}_t) \quad (2.17)$$

Gdzie  $f({}^S_E \hat{q}_{est,t-1}, {}^E \hat{d}, {}^S \hat{a}_t)$  jest funkcją celu,  ${}^S \hat{a}_t$  - kierunek pola w układzie współrzędnych czujnika,  ${}^E \hat{d}$  - kierunek pola w układzie współrzędnych Ziemi.

### 2.1. Gradient funkcji celu

$$\nabla f({}^S_E \hat{q}_{est,t-1}, {}^E \hat{d}, {}^S \hat{a}_t) = J^T({}^S_E \hat{q}_{est,t-1}, {}^E \hat{d}) f({}^S_E \hat{q}_{est,t-1}, {}^E \hat{d}, {}^S \hat{a}_t) \quad (2.18)$$

$${}^S \hat{a}_t = [0 \ a_x \ a_y \ a_z] \quad (2.19)$$

$$f({}^S_E \hat{q}_{est,t-1}, {}^E \hat{d}, {}^S \hat{a}_t) = \begin{bmatrix} 2(q_2 q_4 - q_1 q_3) - a_x \\ 2(q_1 q_2 - q_3 q_4) - a_y \\ \left(\frac{1}{2} - q_2^2 - q_3^2\right) - a_z \end{bmatrix} \quad (2.20)$$

$$J({}^S_E \hat{q}_{est,t-1}, {}^E \hat{d}) = \begin{bmatrix} -2q_3 & 2q_4 & -2q_1 & 2q_2 \\ 2q_2 & 2q_1 & 2q_4 & 2q_3 \\ 0 & -4q_2 & -4q_3 & 0 \end{bmatrix} \quad (2.21)$$

### 2.2. Spadek gradientu

$${}^S_E q_{\nabla,t} = {}^S_E q_{est,t-1} - \mu_t \frac{\nabla f}{\|\nabla f\|} \quad (2.22)$$

Gdzie  $\mu_t$  – współczynnik długości kolejnych kroków

$\frac{\nabla f}{\|\nabla f\|}$  - normalizacja gradientu.

$$\|\nabla f\| = \sqrt{f(1)^2 + f(2)^2 + f(3)^2} \quad (2.23)$$

## 3. Łączenie obliczeń orientacji

$${}^S_E q_{est,t} = {}^S_E q_{w,t} - \beta * {}^S_E q_{\nabla,t} \quad (2.24)$$

$\beta$  - współczynnik kompensacji błędów

## 4. Wyjście filtru

$${}^S_E q_{est,t} = {}^S_E q_{est,t-1} + {}^S_E q_{est,t} * dt \quad (2.25)$$

## 2.3 FILTR KALMANA. DYSKRETNY ALGORYTM.

Filtr Kalmana wykorzystuje dynamiczny model systemu, znane działania kontrolne i pomiary, aby utworzyć optymalne oszacowanie stanu. Algorytm składa się z dwóch powtarzających się faz (Rys. 3): przewidywania (ang. Predict) i korekcji (ang. Correct). W pierwszej kolejności obliczana jest prognoza stanu w następnej chwili czasowej (z uwzględnieniem niedokładności ich pomiaru). Z drugiej strony nowa informacja z czujnika koryguje przewidywaną wartość (uwzględniając również niedokładność i szum tej informacji):

Do opisu zarówno procesu jak i systemu pomiarowego stosują się modele matematyczne.

$$x_k = Ax_{k-1} + Bu_{k-1} + w_{k-1} \quad (2.26)$$

$$z_k = Hx_k + v_k \quad (2.27)$$

Pierwsze równanie to jest model procesu, drugie – model pomiaru. Gdzie:

$x_k$  – wartość stanu.

$x_{k-1}$  – wartość stanu w poprzednim kroku.

$A$  – macierz stanu

$u_{k-1}$  – wartość sterowania w poprzednim kroku.

$B$  – macierz wejścia (sterowania).

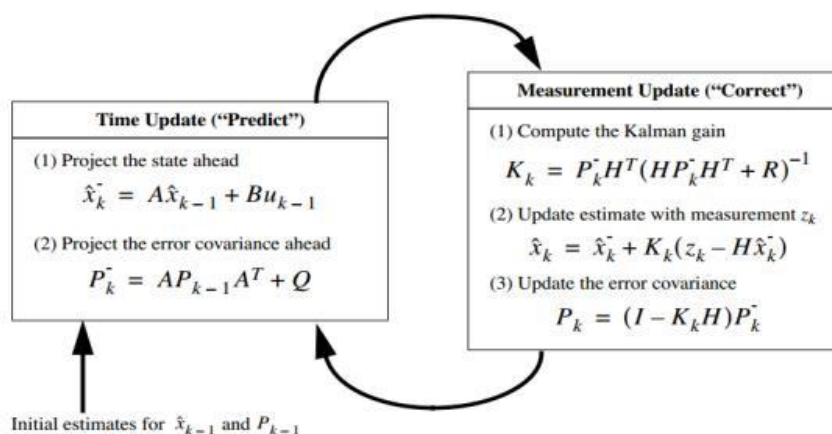
$z_k$  – wartość pomiaru.

$H$  – macierz wyjścia.

$w_{k-1}$  – szum pomiaru.

$v_k$  – zakłócenia.

W trakcie predykcji, bazując na stanie z poprzedniego kroku wyznacza się estymowana wartość  $\hat{x}(k)$  oraz jego kowariancję i są to wartości a priori. Pomiar  $z$  w drugiej fazie jest pewną formą sprzężenia zwrotnego. Na jego podstawie dokonuje się wyznaczenia wartości a posteriori dla stanu i jego kowariancji.



Rys.3 Dyskretny algorytm filtracji Kalmana. [14]

Równania pierwszej fazy:

$$\hat{x}_k^- = A\hat{x}_{k-1} + Bu_{k-1} \quad (2.28)$$

$$P_k^- = AP_{k-1}A^T + Q \quad (2.29)$$

Gdzie  $\hat{x}_k^-$  i  $P_k^-$  - to prognozowane wartości stanu i kowariancji a priori,  $\hat{x}_{k-1}$  i  $P_{k-1}$  - to optymalne szacowane wartości a posteriori wykonane w poprzednim kroku.

Druga faza się zaczyna z wyliczenia wzmocnienia Kalmana, to wzmocnienie pokazując z jaką wagą wpłynie faza korekcji na estymowany czas.

$$K_k = P_k^- H^T (HP_k^- H^T + R)^{-1} \quad (2.30)$$

Następnie, na podstawie wszystkich dotychczasowych pomiarach, obliczamy optymalne skorygowanie prognozy w czasie  $k$ .

$$\hat{x}_k = \hat{x}_k^- + K_k(z_k - H\hat{x}_k^-) \quad (2.31)$$

Gdzie  $z_k$  to pomiar, a różnica  $z_k - H\hat{x}_k^-$  - innowacja pomiarowa (ang. measurement innovation).

Na końcu drugiej fazy korygujemy macierz kowariancji.

$$P_k = (I - K_k H)P_k^- \quad (2.32)$$

Gdzie  $I$  to macierz jednostkowa.

### 3 IMPLEMENTACJA ALGORYTMÓW W ŚRODOWISKU MATLAB WRAZ Z WERYFIKACJĄ NA ZBIORZE DANYCH.

W tym rozdziale przeprowadzona implementacja algorytmów filtracji, które były opisane w poprzednim rozdziale, oraz sprawdzenie ich pracy na danych syntetycznych. Matlab jest używany w wersji R2020b.

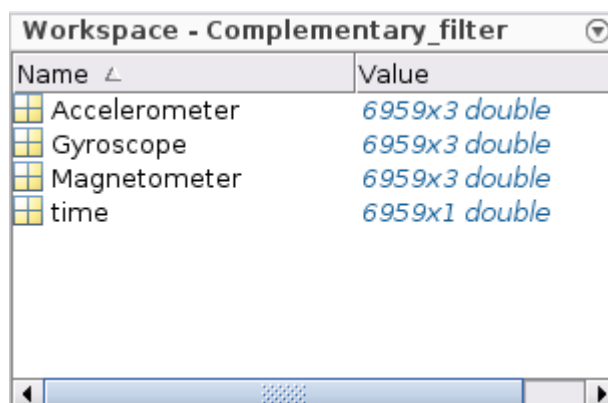
Gotowy zbiór danych zawiera wartości prędkości kątowej w 3 osiach XYZ oraz rzuty przyspieszenia grawitacyjnego na osi XYZ. Dane były pobrane ze źródła [5]. Dane te uzyskano obracając platformę pomiarową, na której zamocowano akcelerometr, żyroskop i magnetometr. Najpierw platformę obrócono o  $90^\circ$  wokół osi x, następnie o  $180^\circ$  w kierunku przeciwnym, a następnie o  $90^\circ$  w celu powrotu platformy do swojej pierwotnej pozycji. Platforma pozostawała nieruchoma przez 3-5 sekund pomiędzy każdym obrotem. Ta sekwencja została następnie powtórzona wokół osi y, a następnie osi z. Pełny opis akwizycji danych przedstawiono w pracy [5].

Przetwarzanie i wyświetlanie danych opisano poniżej w Listingu 1.

*Listing 1 Ładowanie danych i wyświetlanie ich*

```
load('ExampleData.mat'); % ładowanie danych
figure('Name', 'Dane syntetyczne');
axis(1) = subplot(2,1,1);
hold on;
plot(time, Gyroscope(:,1), 'r'); % prędkość kątowa X
plot(time, Gyroscope(:,2), 'g'); % prędkość kątowa Y
plot(time, Gyroscope(:,3), 'b'); % prędkość kątowa Z
legend('X', 'Y', 'Z');
xlabel('Time (s)');
ylabel('Angular rate (deg/s)');
title('Gyroscope');
hold off;
axis(2) = subplot(2,1,2);
hold on;
plot(time, Accelerometer(:,1), 'r');
plot(time, Accelerometer(:,2), 'g');
plot(time, Accelerometer(:,3), 'b');
legend('X', 'Y', 'Z');
xlabel('Time (s)');
ylabel('Acceleration (g)');
title('Accelerometer');
hold off;
```

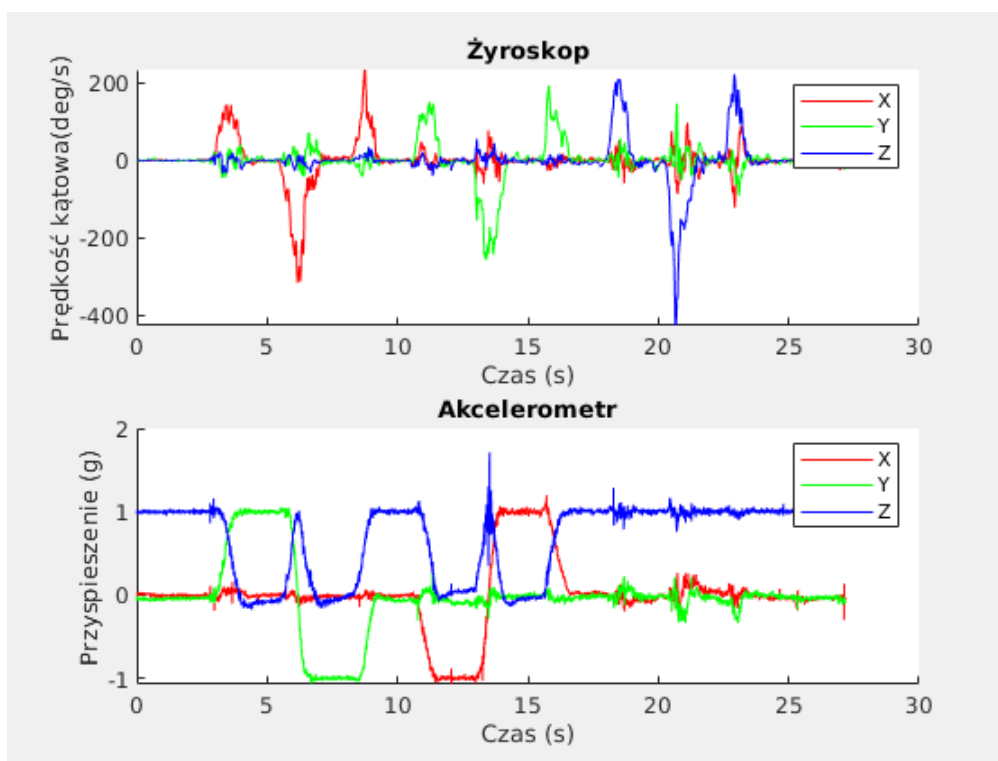
Do ładowania danych wykorzystana funkcja *load*, do wyświetlania *plot* (Rys. 5).



Name	Value
Accelerometer	6959x3 double
Gyroscope	6959x3 double
Magnetometer	6959x3 double
time	6959x1 double

*Rys.4 Dane syntetyczne.*

Po przetwarzaniu danych w okienku workspace mamy 4 macierze (Rys. 4). Dane „*Magnetometer*” nie będą wykorzystywane w pracy.



Rys.5 Wykresy danych.

Kąty z danych akcelerometru obliczamy według wzorów 2.7, 2.8 i 2.9, korzystając z funkcji *atan* (listing 2).

Listing 2 Obliczenie kątów z akcelerometru

```
Ax = atan(Accelerometer(:,1)./(sqrt(Accelerometer(:,2).^2 + Accelerometer(:,3).^2)));
Ay = atan(Accelerometer(:,2)./(sqrt(Accelerometer(:,1).^2 + Accelerometer(:,3).^2)));
Az = atan(Accelerometer(:,3)./(sqrt(Accelerometer(:,1).^2 + Accelerometer(:,2).^2)));
```

Tworzymy 3 puste macierze *Alpha*, *Beta*, *Gamma* do przechowywania kątów po filtrze (listing 3).

Listing 3 Puste macierze Alpha, Beta, Gamma

```
Alpha = zeros(size(Gyroscope(:,1)));
Beta = zeros(size(Gyroscope(:,2)));
Gamma = zeros(size(Gyroscope(:,3)));
```

### 3.1 FILTR KOMPLEMENTARNY

W tym rozdziale został zaimplementowany algorytm filtru komplementarnego (listing 4). Algorytm został napisany według wzorów 2.10, 2.11 i 2.12.

*Listing 4 Filtr Komplementarny. Realizacja w języku Matlab*

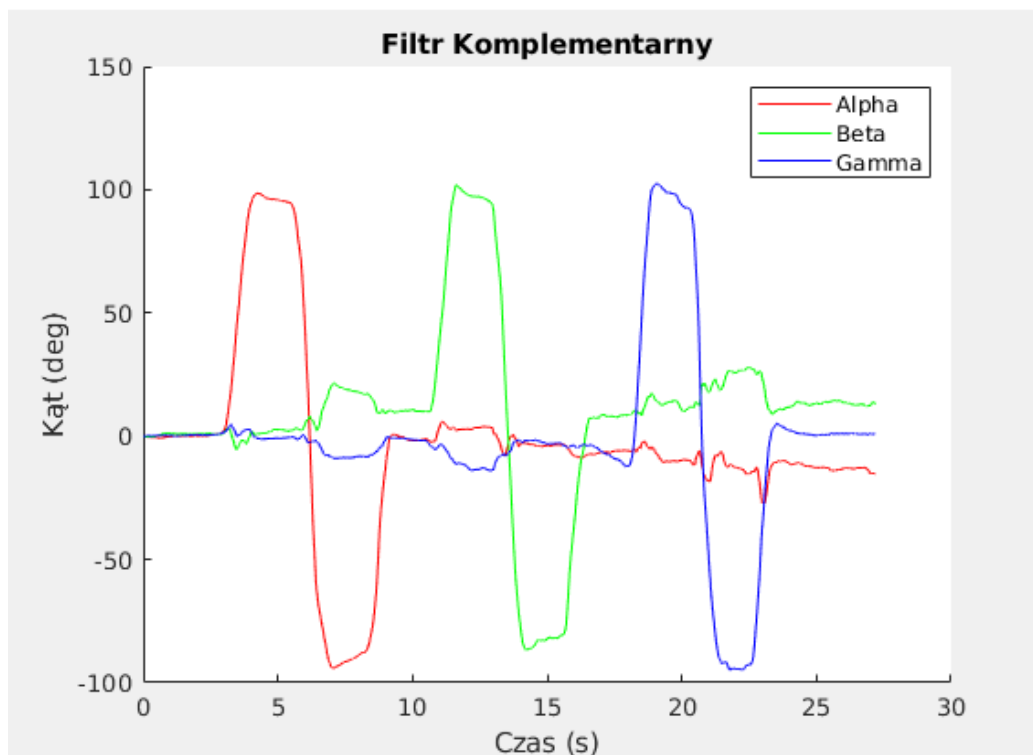
```
dt = 1/256;

for t = 1:length(time)
    if t == 1
        Alpha(t) = K * (Alpha(t) * dt) + (1-K) * Ax(t);
        Beta(t) = K * (Beta(t) * dt) + (1-K) * Ay(t);
        Gamma(t) = K * (Gamma(t) * dt) + (1-K) * Az(t);
    else
        Alpha(t) = K * (Alpha(t-1) + Gyroscope_(t,1) * dt) + (1-K) * Ax(t);
        Beta(t) = K * (Beta(t-1) + Gyroscope_(t,2) * dt) + (1-K) * Ay(t);
        Gamma(t) = K * (Gamma(t-1) + Gyroscope_(t,3) * dt) + (1-K) * Az(t);
    end
end
```

Współczynnik filtra K jest siłą korekcji całkowanego kąta nachylenia z wykorzystaniem danych akcelerometru.

#### Wyniki symulacji

Na wykresie (Rys. 6) widać, odpowiedź filtru jest dość szybka, kąt nachylenia jest prawidłowy w okolicach 90 stopni (zgodnie z opisem w pracy [5]). Pod sam koniec symulacji można zauważyć niewielki dryft zera i z czasem błąd wzrasta, i zero dryfuje co raz mocniej.



*Rys.6 Wykres danych po filtrze komplementarnym*



## 3.2 FILTR MADGWICKA

W danym punkcie pokazana została realizacja filtra Madgwicka (listing 5), który został w pełni opisany w rozdziale 2.2.

*Listing 5 Filtr Madgwicka. Realizacja w języku Matlab*

```
q = obj.Quaternion;

if(norm(Accelerometer) == 0), return; end

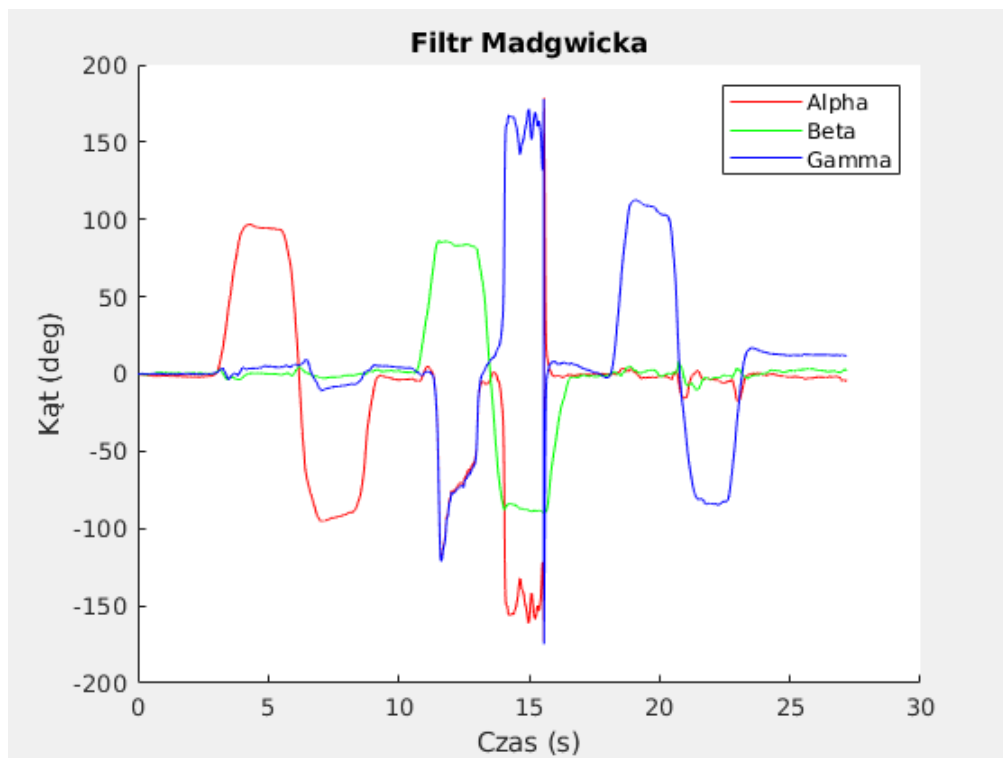
Accelerometer = Accelerometer / norm(Accelerometer);

F = [2*(q(2)*q(4) - q(1)*q(3)) - Accelerometer(1)
     2*(q(1)*q(2) + q(3)*q(4)) - Accelerometer(2)
     2*(0.5 - q(2)^2 - q(3)^2) - Accelerometer(3)];
J = [-2*q(3),      2*q(4),      -2*q(1), 2*q(2)
     2*q(2),      2*q(1),      2*q(4), 2*q(3)
     0,          -4*q(2),     -4*q(3), 0    ];
step = (J'*F);
step = step / norm(step);

qDot = 0.5 * quaternProd(q, [0 Gyroscope(1) Gyroscope(2) Gyroscope(3)]) - obj.Beta * step';
q = q + qDot * obj.SamplePeriod;
obj.Quaternion = q / norm(q);
```

## Wyniki symulacji

W sytuacji z filtrem Madgwicka (Rys. 7) dryftu zera w osiach X i Y nie ma, ale oś Z, ze względu na brak magnetometru, niewiele pływa. Przy nachyleniach około  $\pm 90$  stopni w osi Y, widzimy skoki, jest to błąd związany z budową żyroskopu, przy małych prędkościach kątowych ramy żyroskopu mogą się składać.



*Rys.7 Wykres danych po filtrze Madgwicka*

### 3.3 FILTR KALMANA

Równanie sytemu ma postać

$$\theta_k = \theta_{k-1} + (w_{k-1} - g_{bias})dt \quad (3.1)$$

Gdzie  $\theta$  – odchylenie kątowe,  $w$  – prędkość kąтова, a  $g_{bias}$  – to dryft żyroskopu.

W fazie pierwszej będzie uwzględniony pomiar z żyroskopu,  $w$  posłuży jako sterowanie  $u$ . Można zapisać równanie systemu w postaci:

$$x_k = Ax_{k-1} + Bu \quad (3.2)$$

(2.22)

Wektor stanu:

$$x_k = [\theta_x \ \theta_y \ \theta_z \ g_{bias,x} \ g_{bias,y} \ g_{bias,z}] \quad (3.3)$$

Macierz stanu:

$$A = \begin{bmatrix} 1 & 0 & 0 & -dt & 0 & 0 \\ 0 & 1 & 0 & 0 & -dt & 0 \\ 0 & 0 & 1 & 0 & 0 & -dt \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \quad (3.4)$$

Wektor sterowania:

$$u = [w_x \ w_y \ w_z] \quad (3.5)$$

Macierz wejścia:

$$B = \begin{bmatrix} dt & 0 & 0 \\ 0 & dt & 0 \\ 0 & 0 & dt \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \quad (3.6)$$

Wektor pomiaru:

$$z = [A_x \ A_y \ A_z] \quad (3.7)$$

Macierz wyjścia:

$$H = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \end{bmatrix} \quad (3.8)$$

Zaprogramowanie algorytmu filtracji Kalmana pokazano w Listingu 6,7 i 8.

Zdefiniujemy wszystkie macierze według wzorów 3.3 – 3.8.

*Listing 6 Filtr Kalmana. Definiowanie modelu procesu*

```
x = zeros(6,length(time)); %wektor stanu
A = [1 0 0 -dt 0 0;
     0 1 0 0 -dt 0;
     0 0 1 0 0 -dt;
     0 0 0 1 0 0;
     0 0 0 0 1 0;
     0 0 0 0 0 1]; %macierz stanu 6x6
u = [W_alpha W_beta W_gamma]';
B = [dt 0 0;
     0 dt 0;
     0 0 dt;
     0 0 0;
     0 0 0;
     0 0 0]; %macierz wejścia(sterowania) 6x3
H = [1 0 0 0 0 0;
     0 1 0 0 0 0;
     0 0 1 0 0 0]; %macierz wyjścia 3x6
z = [Ax Ay Az]'; %wektor pomiaru
```

Następnie zdefiniujemy macierzy kowariancji Q oraz R. Wartość  $r$  – dyspersja błędu poszczególnych osi akcelerometru. Według dokumentacji [7] szum akcelerometru równa się  $70 \mu\text{g}/\sqrt{\text{Hz}}$ .

*Listing 7 Filtr Kalmana. Definicja macierzy kowariancji*

```
q = 0.00001;
r = 70;
Q = eye(6).* q; %macierz kowariancji modelu
R = eye(3).* r; %macierz kowariancji pomiarów
```

Kolejnym i ostatnim krokiem będzie realizacja filtru Kalmana w pętli *for*.

*Listing 8 Filtr Kalmana. Realizacja*

```
x_post = zeros(6,1);
P_post = zeros(6,6);

for k = 2:length(time)

    %Predykcja
    x_pri = A * x_post + B * u(:,k-1);
    P_pri = A * P_post .* A' + Q;

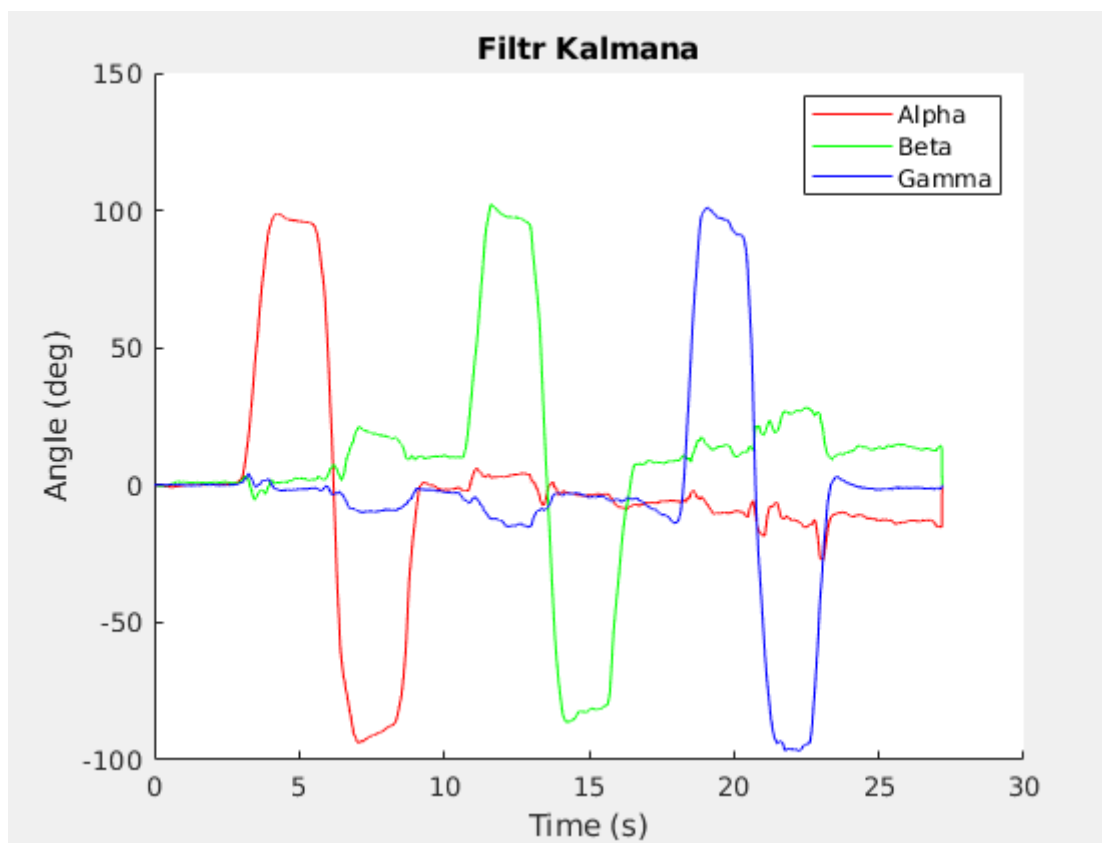
    %Korekcja
    K = P_pri * H' * (H * P_pri * H' + R)^(-1);
    x_post = x_pri + K * (z(:,k) - (H * x_pri));
    P_post = (I - K * H) * P_pri;

    %zapis
    Alpha(k) = x_post(1);
    Beta(k) = x_post(2);
    Gamma(k) = x_post(3);

end
```

## Wyniki symulacji

W tym rozdziale zostało zbadane 3 filtry na zestawie gotowych danych. Filtr Madgwicka najlepiej radzi z dryftem zera żyroskopu, ale ma skoki na 180 stopni. Filtr Kalmana i filtr komplementarny mają podobne wykresy (Rysunek 8 i 6 odpowiednio), jest trochę więcej szumów niż w filtrze Madgwicka. Wyznaczone kąty są prawidłowe we wszystkich trzech filtrach. Reakcja układu na zmiany orientacji jest dość szybka.



Rys.8 Wykres danych po filtrze Kalmana

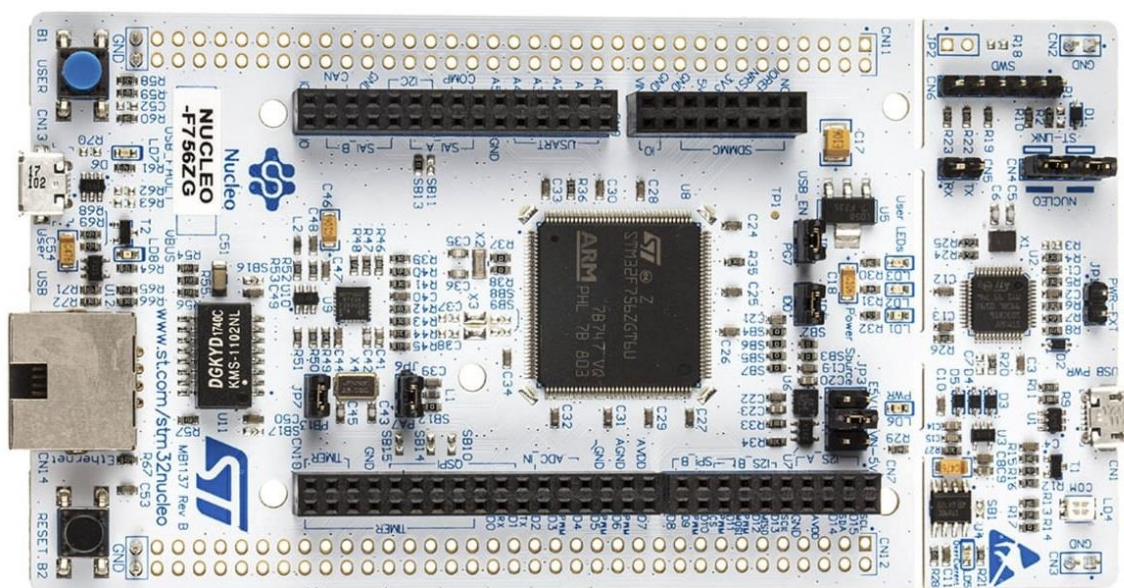
## 4 WERYFIKACJA PRAKTYCZNA

W pracy wykorzystano płytke NUCLEO-144 (Rys. 9) opartą na mikrokontrolerze STM32F756ZG oraz sześcioksiowy moduł inercyjny iNEMO LSM6DSOX na płycie adaptera STEVAL-MKI197V1a (Rys. 10).

### 4.1 MIKROKONTROLER STM32F756ZG

Mikrokontroler STM32F756 jest oparty na wysokowydajnym 32-bitowym rdzeniu RISC ARM Cortex-M7 pracującym z częstotliwością do 216 MHz. Rdzeń Cortex-M7 charakteryzuje się precyzją pojedynczej jednostki zmiennoprzecinkowej (ang. SFPU - single floating point unit).

W pracy wykorzystamy interfejs I2C do odczytu danych z czujnika oraz USB do przesłania ich do komputera.



Rys.9 Nucleo-F756ZG [15]

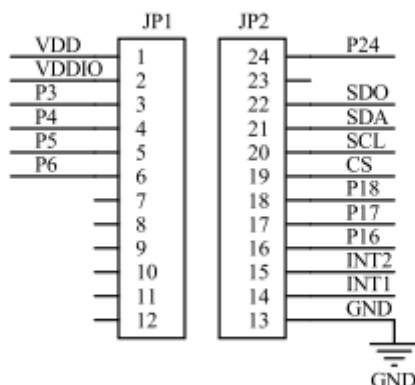
### 4.2 STEVAL-MKI197V1A Z CZUJNIKIEM MEMS LSM6DOX

Najważniejsze parametry układu to:

- pomiar przyspieszenia liniowego w zakresie  $\pm 2/\pm 4/\pm 8/\pm 16$  g
- pomiar przyspieszenia kąowego w zakresie  $\pm 125/\pm 250/\pm 500/\pm 1000/\pm 2000$  dps
- Analogowe napięcie zasilania: 1,71 V do 3,6 V
- Niezależne zasilanie IO (1,62 V)
- Kompaktowa powierzchnia: 2,5 mm x 3 mm x 0,83 mm
- Interfejs szeregowy SPI / I<sup>2</sup>C i MIPI I3CSM z synchronizacją danych procesora głównego



Rys.10 STEVAL-MKI197V1 [16]



Rys.11 Schemat płyty STEVAL-MKI197V1 [10]

### 4.3 SCHEMAT POŁĄCZENIA

Zastosowano dwa rodzaje interfejsów szeregowych I2C oraz USB.

#### I2C

I2C jest jednym z najpopularniejszych standardów komunikacyjnych występującym w świecie układów elektronicznych. I2C do komunikacji wykorzystuje tylko dwie linie transmisyjne: SDA (Serial Data Line) - linia danych, SCL (Serial Clock Line) - linia zegara.

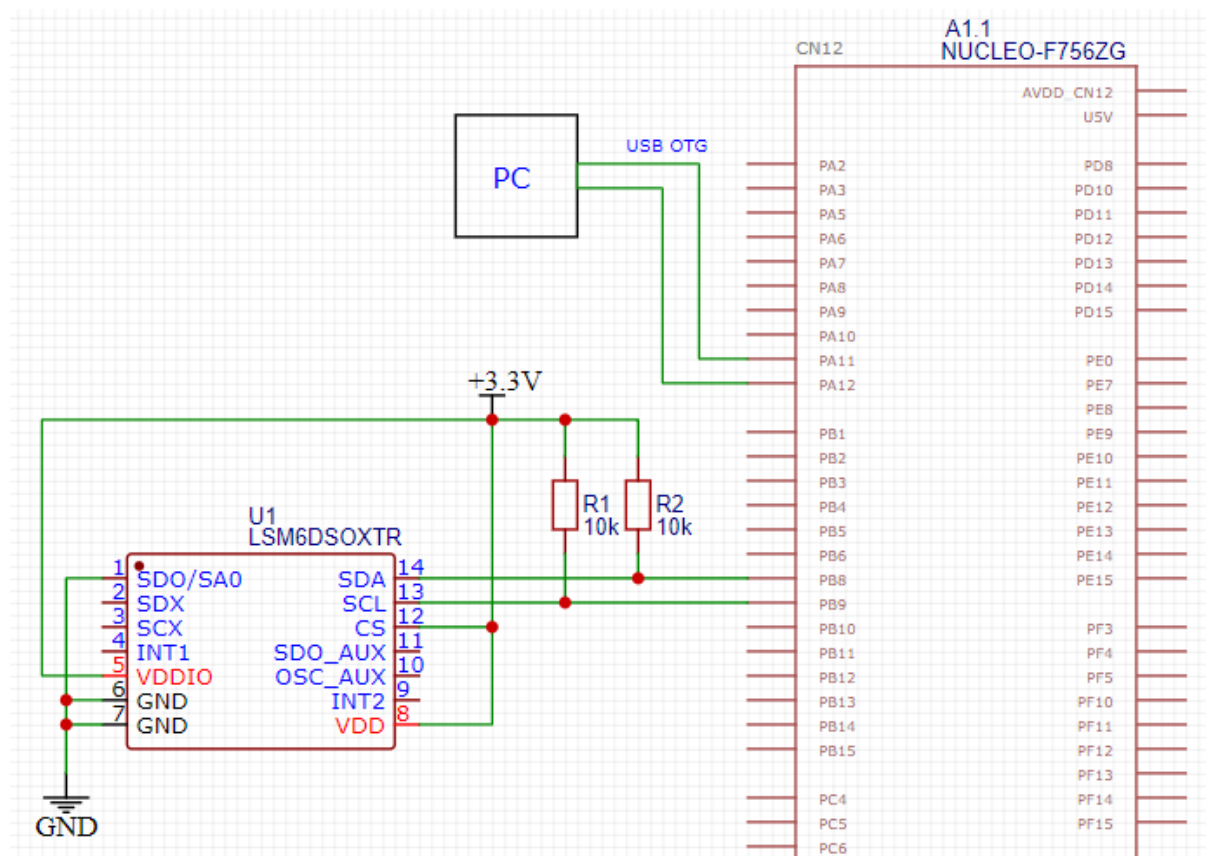
Linia SDA jest dwukierunkowa, co umożliwia transmisję half-duplexową (obydwa urządzenia mogą nadawać i odbierać dane, ale nie jednocześnie). Linie są na stałe podciągnięte do źródła zasilania przez rezystory podciągające (R1, R2 – 10 kOm).

#### USB

Uniwersalna Magistrala Szeregowa (ang. Universal Serial Bus) jest standardem komunikacyjnym definiującym kable, złącza oraz protokół transmisji danych. Magistrala USB umożliwia podłączanie, wymianę danych oraz zasilania pomiędzy komputerami i urządzeniami elektronicznymi.

USB może być również wykorzystywane do transmisji danych poprzez wirtualny COM port, CDC (ang. Communication Device Class) definiuje klasę urządzenia, za jaką może się podawać kontroler USB. Jest ona używana do transmisji danych w sieciach komputerowych. Jedną z możliwości klasy CDC jest stworzenie wirtualnego portu COM w celu imitacji protokołu RS-232.

Uwzględniając wyprowadzenie płyty STEVAL-MKI197V1, pokazane na rysunku 11, został zaprojektowany schemat połączenia płyty głównej z czujnikiem i komputerem (Rys. 12).



Rys.12 Schemat połączenia

#### 4.4 OPROGRAMOWANIE

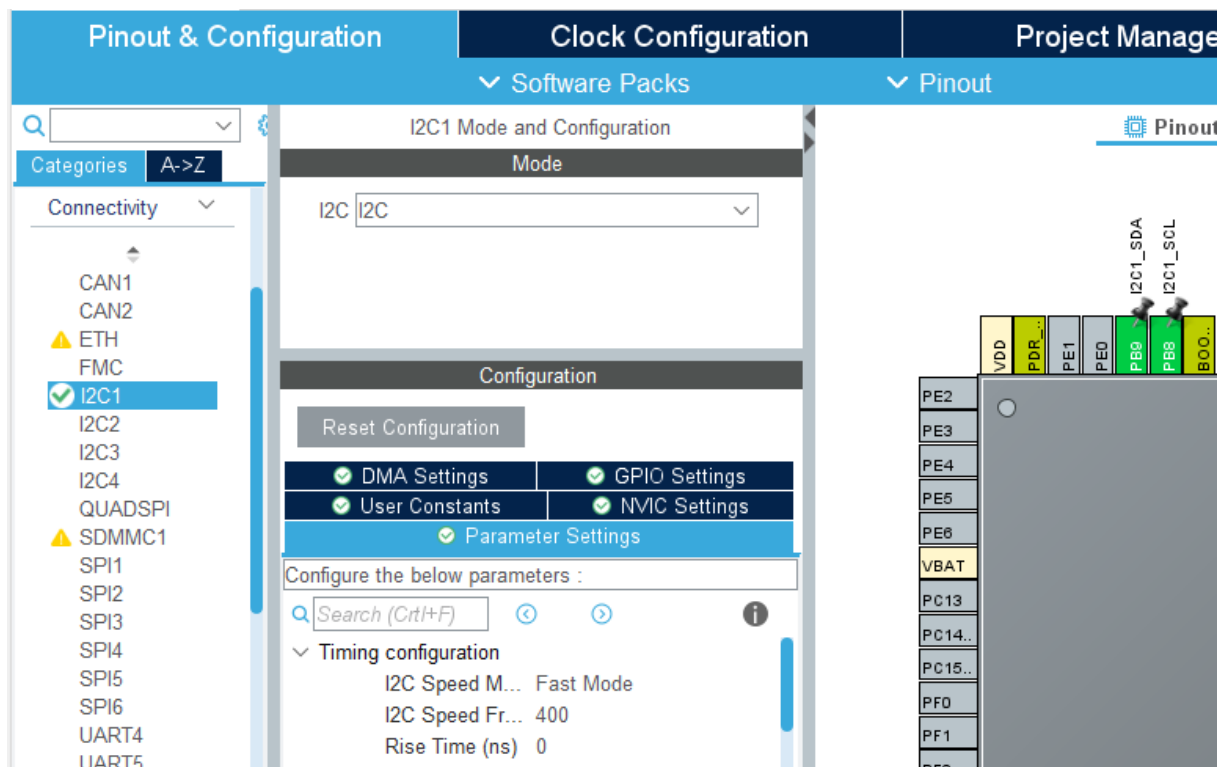
W pracy wykorzystano oficjalne środowisko dla programowania mikrokontrolerów STM32 STM32CubeIDE. STM32CubeIDE jest zaawansowaną platformą projektową opartą o framework Eclipse/CDT przeznaczoną do tworzenia projektów w języku C/C++ i zawiera zestaw narzędzi przeznaczony do edycji, kompilowania i debugowania kodu. Kompilator C/C++ wykorzystuje GCC (GNU Compiler Collection), a debugger GDB (GNU Project Debugger). Co najważniejsze, w końcu STM32Cube IDE jest ściśle zintegrowany z konfiguratorem STM32CubeMX.

STM32CubeMX jest programem pozwalającym na konfigurację peryferiów mikrokontrolerów STM32 za pomocą interfejsu graficznego. Umożliwia on także automatyczne wygenerowanie kodu konfiguracyjnego w języku C wykorzystującego biblioteki HAL (ang. (Hardware Abstraction Layer)) w postaci gotowego projektu.

Biblioteki HAL (ang. Hardware Abstraction Layer) stanowią wysokopoziomowy interfejs do części sprzętowej mikrokontrolera, pozwalając tym samym korzystać z peryferiów w bardzo prosty i przejrzysty sposób.

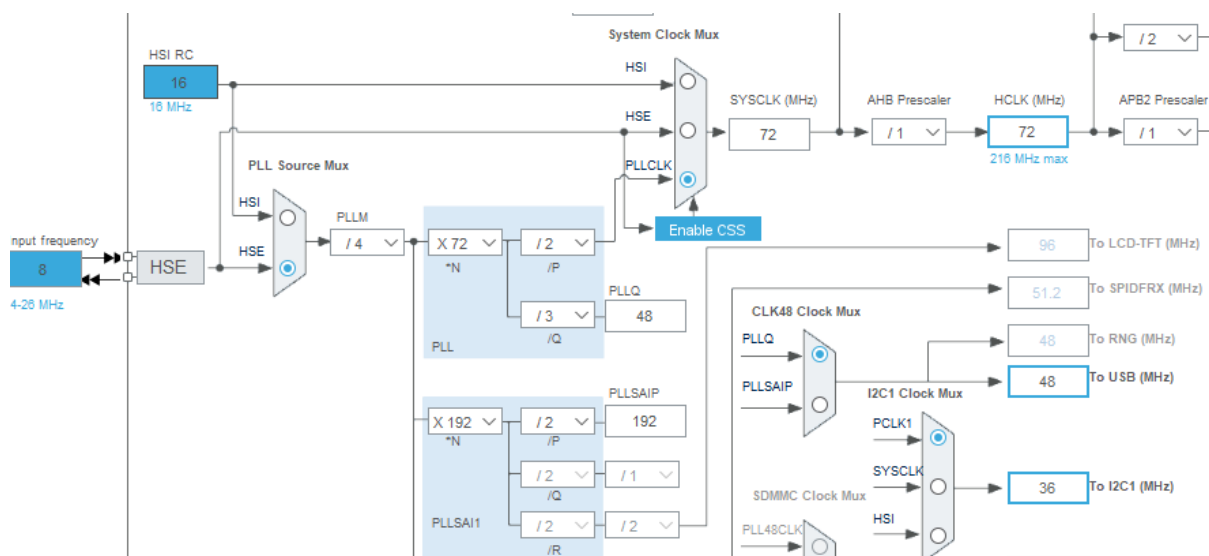
## 4.5 KONFIGURACJA W STM32CUBEMX

Za pomocą STM32CubeMX (Rys. 13), został skonfigurowany interfejs I2C, wyjścia PB8 i PB9 w mikrokontrolerze zdefiniowane jako SDA i SCL, również została zmieniona częstotliwość linii zegarowej na 400 kHz (Fast mode).



Rys.13 Konfiguracja I2C w STM32CubeMX

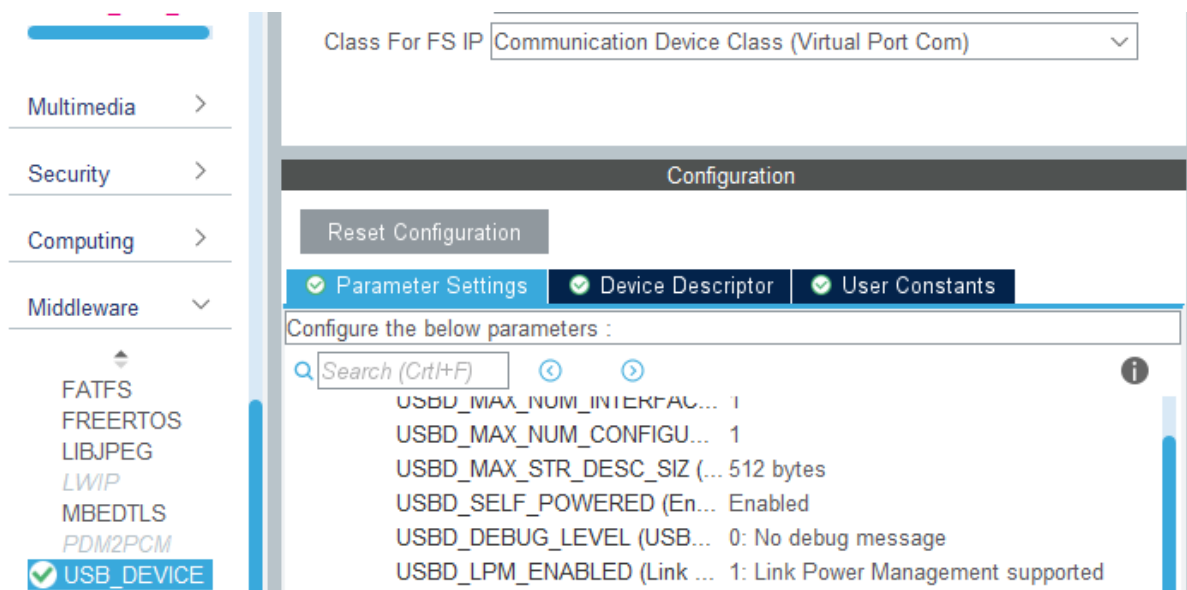
Następnie został uruchamiany USB\_OTG\_FS w trybie Device\_Only i skonfigurowany zegar mikrokontrolera (Rys. 14), aby częstotliwość na magistrali USB wynosiła 48MHz.



Rys.14 Konfiguracja zegara mikrokontrolera

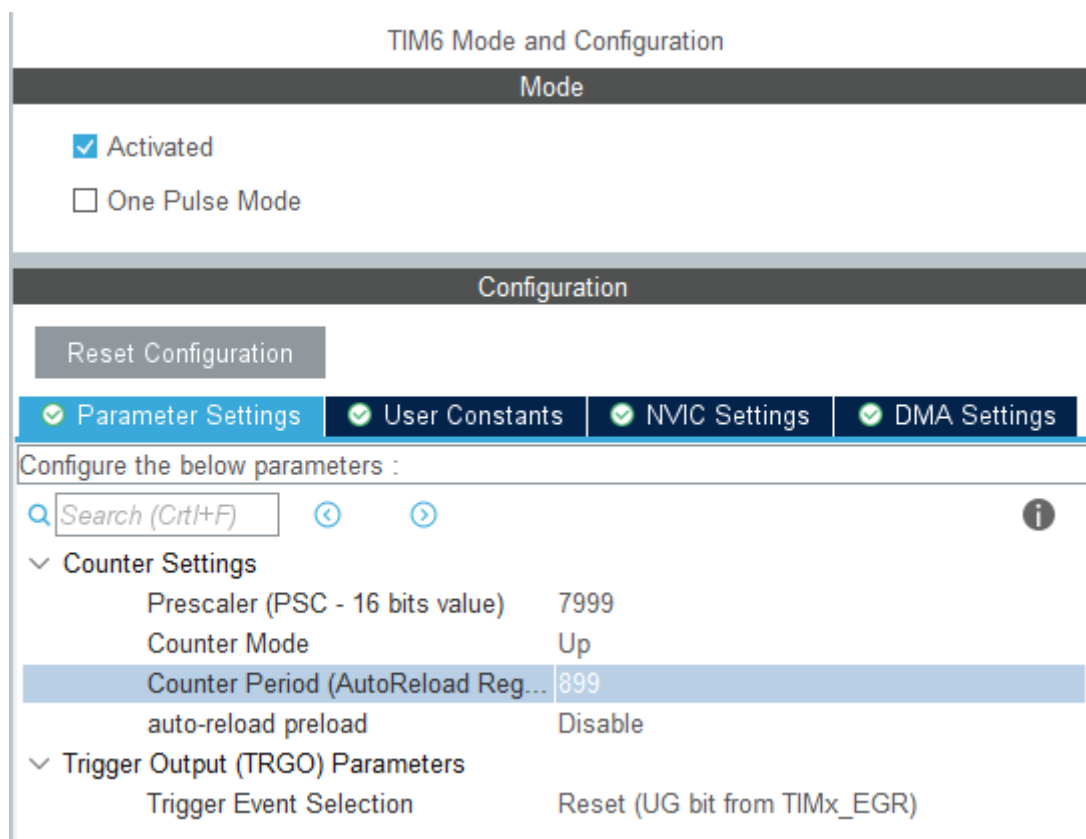


Dalej był ustawiony klasa urządzenia CDC (ang. Communication Device Class). Dzięki temu mikrokontroler będzie widziany w komputerze jako wirtualny port COM. Parametry COM poru przedstawione na rysunku 15.

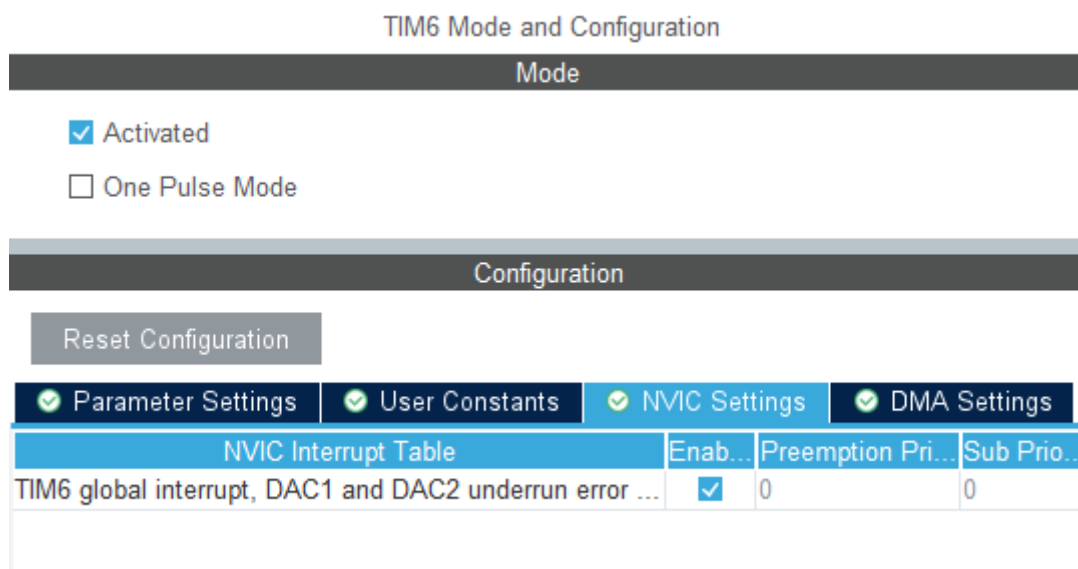


Rys.15 Konfiguracja USB w STM32CubeMX

Do pomiaru stałej czasowej próbkowania zastosowano timer generujący przerwania (TIM6). Konfiguracja timera jest pokazana na rysunkach 16 i 17, częstotliwość przerwań wynosi 10 Hz.



Rys.16 Konfiguracja timera w STM32CubeMX



Rys.17 Aktywacja przerwań timera w STM32CubeMX

## 4.6 ZEBRANIE SUROWYCH DANYCH Z CZUJNIKA. GŁÓWNY PROGRAM

Do inicjalizacji i konfiguracji czujnika wykorzystano biblioteki producenta STMicroelectronics „lsm6dsox.h”. Aby pracować z biblioteką należy pobrać dwa pliki *lsm6dsox.h* i *lsm6dsox.c* z [17], następnie dodać ich do folderu z projektem i w pliku *main.c* napisać `#include "lsm6dsox.h"`.

Na początku programu sprawdzam połączenie z czujnikiem LSM6DSOX (Listing 9).

Listing 9 Sprawdzenie adresu czujnika

```
lsm6dsox_device_id_get(&dev_ctx, &whoami);
if (whoami != LSM6DSOX_ID)
while(1)
{
    HAL_GPIO_TogglePin(Red_led_GPIO_Port,
                        Red_led_Pin);
    HAL_Delay(500);
}
```

Za tym resetowanie do ustawień fabrycznych (Listing 10).

Listing 10 Resetowanie czujnika

```
lsm6dsox_reset_set(&dev_ctx, PROPERTY_ENABLE);

do {
    lsm6dsox_reset_get(&dev_ctx, &rst);
} while (rst);
```

Konfiguracja czujnika polega na ustawieniu skali do akcelerometru i żyroskopu, oraz ustawieniu częstotliwości odświeżania danych wewnątrz czujnika (Listing11).

Listing 11 Konfiguracja czujnika

```
lsm6dsox_xl_data_rate_set(&dev_ctx, LSM6DSOX_XL_ODR_417Hz);
lsm6dsox_gy_data_rate_set(&dev_ctx, LSM6DSOX_GY_ODR_417Hz);
```

```
lsm6dsox_xl_full_scale_set(&dev_ctx, LSM6DSOX_2g);  
lsm6dsox_gy_full_scale_set(&dev_ctx, LSM6DSOX_2000dps);
```

Od tego momentu zaczyna się pętla główna *While*, ale ona jest pusta, ponieważ korzystamy wyłącznie z przerw, aby stabilnie mieć stałą częstotliwość próbkowania. Funkcja *HAL\_TIM\_PeriodElapsedCallback* (Listing 12) jest wywoływana po upływie sprecyzowanego wcześniej czasu (częstotliwość przerw, rozdział 4.5)

*Listing 12 Funkcja HAL\_TIM\_PeriodElapsedCallback*

```
void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim)
```

Za pomocą funkcji *lsm6dsox\_acceleration\_raw\_get*, *lsm6dsox\_angular\_rate\_raw\_get* odczytuje dane i kopie ich do tablic *data\_raw\_acceleration*, *data\_raw\_angular\_rate* (Listing 13).

*Listing 13 Odczyt danych*

```
lsm6dsox_acceleration_raw_get(&dev_ctx, data_raw_acceleration);  
lsm6dsox_angular_rate_raw_get(&dev_ctx, data_raw_angular_rate);
```

Uzyskane wartości są przedstawione w słowach bitowych, należy ich przekonwertować, dane akcelerometru w ‘g’ (przyspieszenie grawitacyjne), dane z żyroskopu w ‘deg/s’ (stopnie na sekundę). kod konwersji danych jest pokazany na Listingu 14.

*Listing 14 Konwersja danych*

```
acceleration_mg[0] = lsm6dsox_from_fs2_to_mg(data_raw_acceleration[0]);  
acceleration_mg[1] = lsm6dsox_from_fs2_to_mg(data_raw_acceleration[1]);  
acceleration_mg[2] = lsm6dsox_from_fs2_to_mg(data_raw_acceleration[2]);  
  
angular_rate_mdps[0] = lsm6dsox_from_fs2000_to_mdps(data_raw_angular_rate[0]);  
angular_rate_mdps[1] = lsm6dsox_from_fs2000_to_mdps(data_raw_angular_rate[1]);  
angular_rate_mdps[2] = lsm6dsox_from_fs2000_to_mdps(data_raw_angular_rate[2]);
```

Ostatnim działaniem programu to kopiowanie danych do buforu *tx\_buffer* za pomocą funkcji *sprintf* i wysyłanie ich do komputera przez USB w moment generowania przerwania (listing 15 i 16).

*Listing 15 Kopiowanie danych*

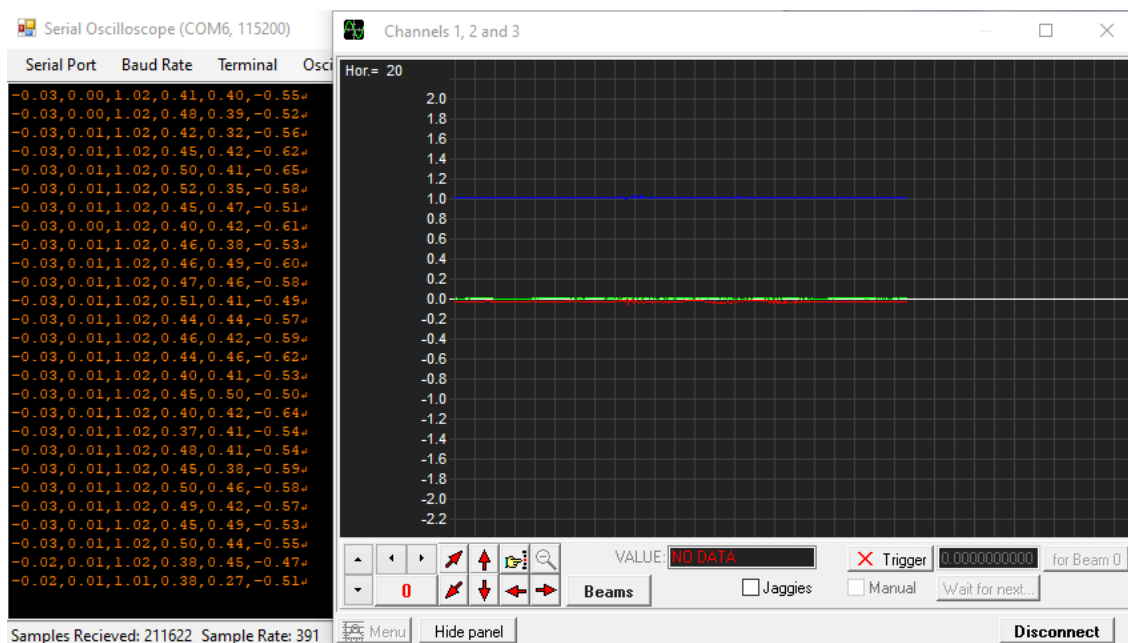
```
sprintf((char *)tx_buffer, "%4.2f,%4.2f,%4.2f,%4.2f,%4.2f,%4.2f\r",  
acceleration_mg[0], acceleration_mg[1], acceleration_mg[2],  
angular_rate_mdps[0], angular_rate_mdps[1], angular_rate_mdps[2]);
```

*Listing 16 Wysyłanie danych*

```
tx_com(tx_buffer, strlen((char const *)tx_buffer));
```

## 4.7 ODCZYT DANYCH PRZEZ PORT COM

W tym rozdziale przedstawiony opis odczytu danych przez port COM po stronie komputera. Do połączenia z płytą główną był wybrany program „Serial Oscilloscope”, pobrany z [11]. Interfejs programu pokazano na rysunku 18.



Rys.18 Serial Oscilloscope

Zapis danych odbywał się za pomocą funkcji zapisu z dziennika (ang. log) do pliku. LoggedData.csv. Pierwsze trzy kolumny to wartości zwracane z akcelerometri w postaci X, Y, Z, następne trzy to z żyroskopu również w postaci X, Y, Z.

Zbieranie danych rozpoczęło się od pozycji początkowej o współrzędnych 0,0,0 odpowiednio w osiach X, Y, Z. Następnie wykonano nachylenie w osi X o  $90 \pm 5$  stopni, po czym nastąpił powrót do pozycji początkowej. Powyższe zostało zrobione dla każdej osi.

## 5 WERYFIKACJA PRACY ALGORYTMÓW Z WYKORZYSTANIEM SUROWYCH DANYCH

W tym rozdziale zostało sprawdzone działanie filtrów na danych surowych, zebranych w poprzednim rozdziale. Będziemy korzystać z kodu, który używaliśmy w rozdziale 3 do filtracji danych z gotowego zbioru danych.

### 5.1 DANE SUROWE

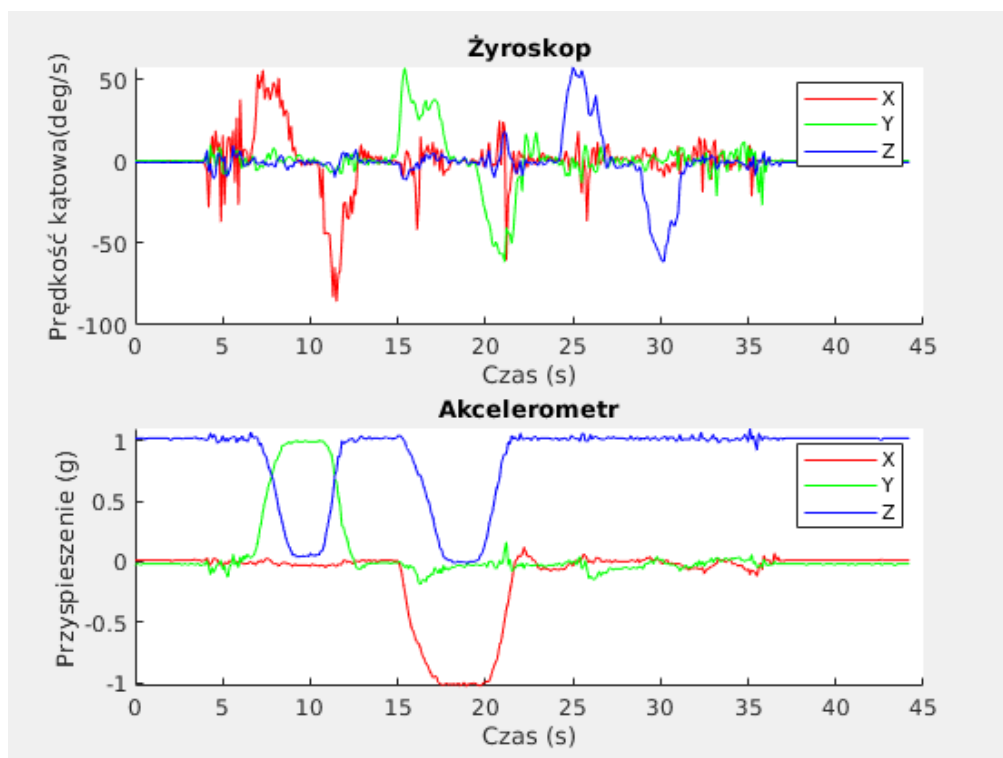
W danym punkcie przedstawiono przetwarzanie i przygotowywanie danych do filtrowania.

Najpierw importujemy dane używając funkcji *cvsread* (Listing 17).

*Listing 17 Kopiowanie danych*

```
NewData = csvread('LoggedData.csv',1,0);  
NewData_ = readtable('LoggedData.csv','NumHeaderLines',1);  
Accelerometer = NewData(:,1:3);  
Gyroscope = NewData(:,4:6);
```

Wykres zebranych danych (Rys. 19) jest podzielony na dwie części: wykres danych z żyroskopu i wykres danych akcelometru.



*Rys.19 Dane surowe*

Wykres żyroskopowy pokazuje zbocza wykonane dla wszystkich trzech osi, widoczny jest również silny szum sygnału.

Na wykresie akcelometru spadki są widoczne tylko w dwóch osiach X i Y, podobnie jak w przypadku osi Z, podczas skrećania rzut tej osi jest równoległy do osi grawitacji, więc akcelometr nie może określić jej kąta.

Przed rozpoczęciem określenia kątów skalibrujemy dane z żyroskopu, za pomocą funkcji *mean*, która uśrednia wartości, które zostały zebrane w statycznym położeniu czujnika. (listing 18).

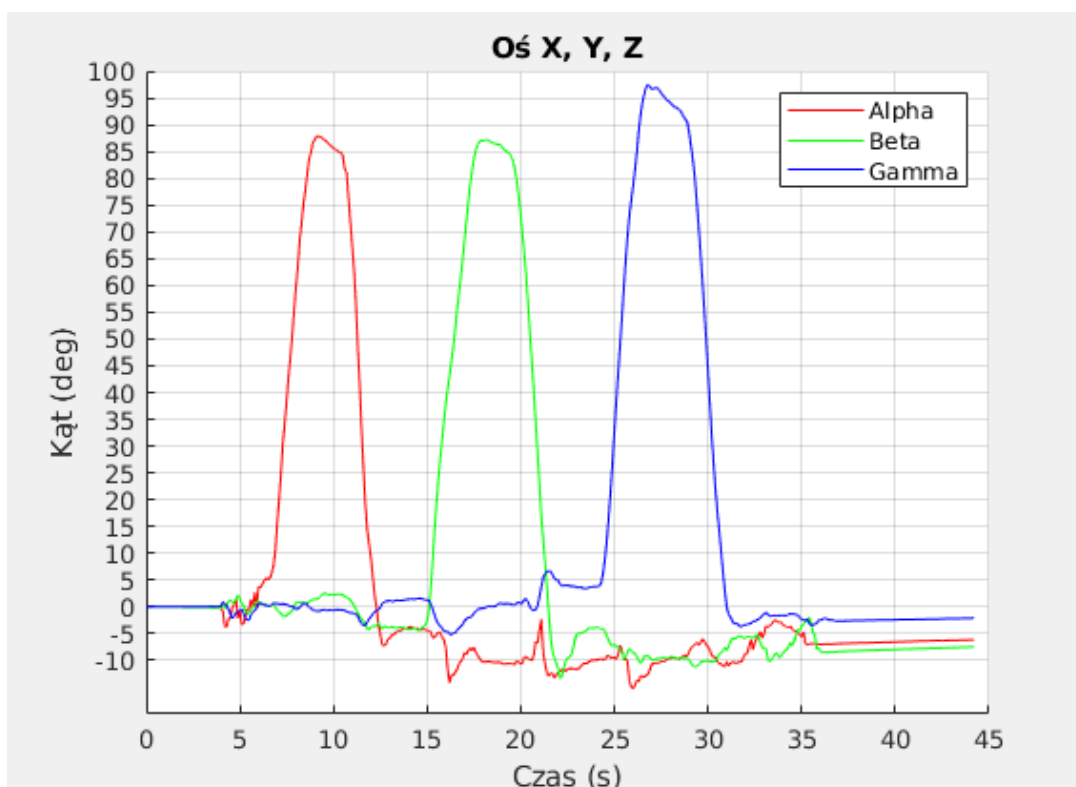
Listing 18 Kalibracja danych żyroskopu

```
DataStatic = csvread('LoggedDataStatic.csv',1,0);
GyroscopeStatic = DataStatic(:,4:6);
OffsetX = mean(GyroscopeStatic(1:250,1));
OffsetY = mean(GyroscopeStatic(1:250,2));
OffsetZ = mean(GyroscopeStatic(1:250,3));
Gyroscope_ = zeros(length(Gyroscope(:,1)),3);
Gyroscope_(:,1) = Gyroscope(:,1) - OffsetX;
Gyroscope_(:,2) = Gyroscope(:,2) - OffsetY;
Gyroscope_(:,3) = Gyroscope(:,3) - OffsetZ;
```

## 5.2 WYNIKI BADAŃ

### Filtr Komplementarny

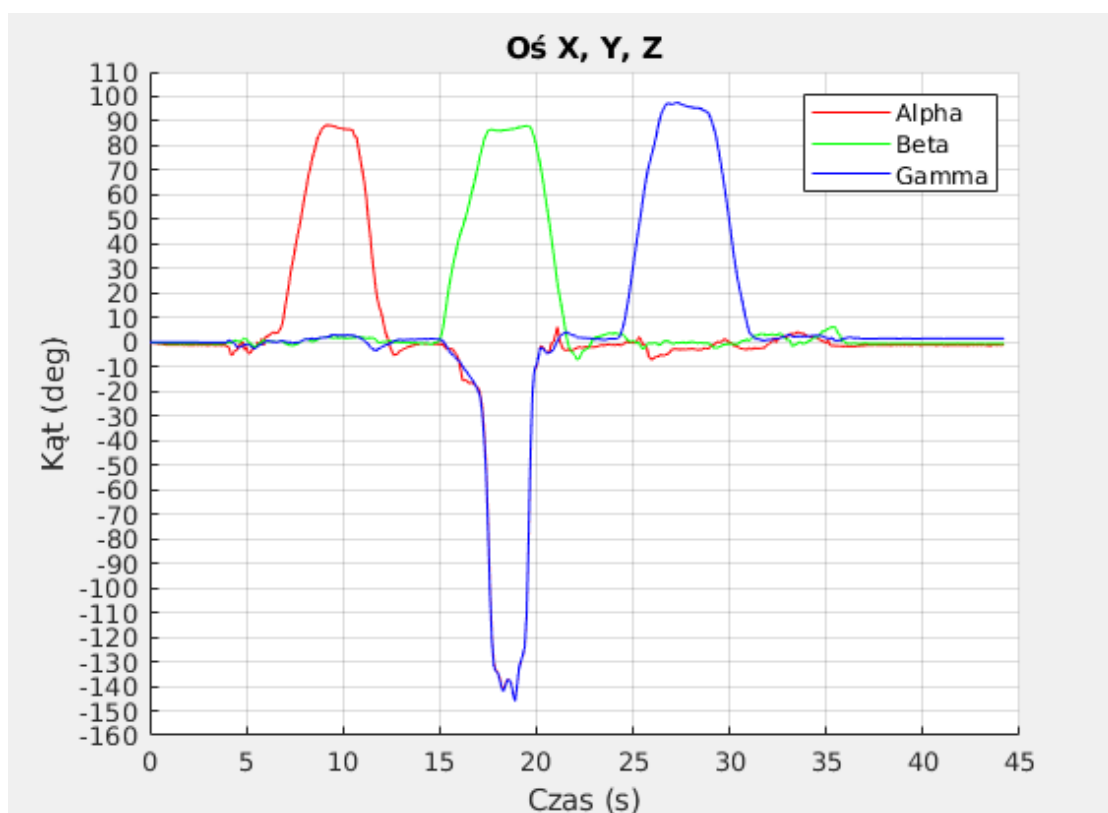
Na powyższym wykresie (Rys. 20) widać brak szumów, podane kąty obrotu 50 stopni są wyznaczone poprawnie z dość szybką reakcją, ale dryft zerowy żyroskopu wzrasta z każdą iteracją, można śmiało powiedzieć, że ten filtr nie radzi sobie z zadaniem, a akcelerometr nie kompensuje dryfu zera.



Rys.20 Filtr Komplementarny na danych surowych

### Filtr Madgwicka

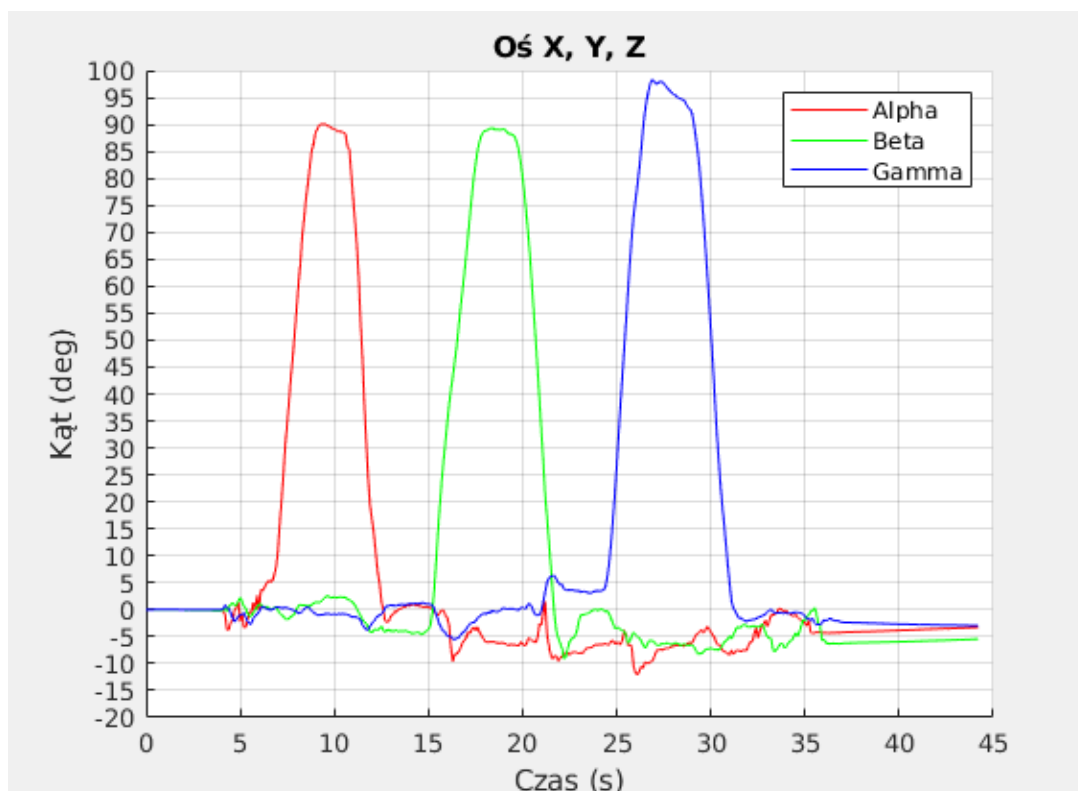
Na wykresie filtra Madgwicka (Rys. 21) widać prawie idealne przebiegi w osiach X i Y, brak szumów, początkowa pozycja czujnika znajduje się w okolicach zera przez cały czas, kąty położenia czujnika są również prawidłowo określone. Z kolei dryft zerowy jest wyraźnie widoczny na osi Z, ale algorytm w tej osi również dobrze tłumi szumy i poprawnie wyznacza rzeczywiste kąty odchylenia.



Rys.21 Filtr Madgwicka na danych surowych

### Filtr Kalmana

Porównując poprzednie dwa wykresy z wykresem filtra Kalmana (Rys. 22), możemy powiedzieć, że ten algorytm jest średnią między filtrem komplementarnym a filtrem Madgwicka. Jest w nim dryft zerowy, ale nie tak duży jak w komplementarnym, nie ma szumu, a kąty odpowiadają zadany.



Rys.22 Filtr Kalmana na danych surowych

## 6 IMPLEMENTACJA ALGORYTMÓW W UKŁADZIE RZECZYWISTYM

W niniejszym rozdziale przeprowadzono implementację trzech algorytmów filtracji w układzie rzeczywistym, które są opisane w poprzednim rozdziale. Układ rzeczywisty jest oparty na płytce NUCLEO - F756ZG oraz czujnik LSM6DSOX. Konfiguracja układu jest opisana w rozdziale 4.

Do wyświetlenia danych w czasie rzeczywistym był wybrany program „Serial Oscilloscope”.

Na początku programu zostały zadeklarowane funkcje filtrów oraz zmienne globalne do przechowywania wartości wejściowych, wyjściowych oraz tymczasowych (Listingi 19 i 20).

*Listing 19 Deklaracja funkcji*

```
int32_t complementary_filter(float acc_data[], float gyr_data[]);
int32_t kalman_filter(float acc_data[], float gyr_data[]);
int32_t madgwick_filter(float acc_data[], float gyr_data[]);
void mult_matrix(int m, int n, int l, float A[m][n], float B[n][l],
float C[m][l]);
float invSqrt(float x);
```

*Listing 20 Zmienne globalne*

```
float output_data[3] = {0.0f, 0.0f, 0.0f};
float old_data[3] = {0.0f, 0.0f, 0.0f};
volatile float pi = 3.141592653589793238f;
const float dt = 0.1f;

// zmienne do filtra Madgwicka
volatile float q0 = 1.0f, q1 = 0.0f, q2 = 0.0f, q3 = 0.0f;
volatile float beta = 0.001f;
volatile float sampleFreq = 10.0f;
volatile double time = 0.0f;

// zmienne do filtru Kalmana
// macierz stanu
float A[6][6] = {{1.0f, 0, 0, -dt, 0, 0},
                 {0, 1.0f, 0, 0, -dt, 0},
                 {0, 0, 1.0f, 0, 0, -dt},
                 {0, 0, 0, 1.0f, 0, 0},
                 {0, 0, 0, 0, 1.0f, 0},
                 {0, 0, 0, 0, 0, 1.0f}};

//macierz wejścia
float B[6][3] = {{dt, 0, 0},
                 {0, dt, 0},
                 {0, 0, dt},
                 {0, 0, 0},
                 {0, 0, 0},
                 {0, 0, 0}};

//macierz wyjścia
float H[3][6] = {{1.0f, 0, 0, 0, 0, 0},
                 {0, 1.0f, 0, 0, 0, 0},
                 {0, 0, 1.0f, 0, 0, 0}};

const float Kq = 0.0001f; //błąd modelu
const float Kr = 10.0f; // błąd pomiaru
// macierz kowariancji modelu
float Q[6][6] = {{Kq, 0, 0, 0, 0, 0},
                 {0, Kq, 0, 0, 0, 0},
                 {0, 0, Kq, 0, 0, 0},
```



```
        {0, 0, 0, Kq, 0, 0},  
        {0, 0, 0, 0, Kq, 0},  
        {0, 0, 0, 0, 0, Kq}}};  
// macierz kowariancji pomiaru  
float R[3][3] = {{Kr, 0, 0},  
                 {0, Kr, 0},  
                 {0, 0, Kr}}};  
// macierz jednostkowa  
float I[6][6] = {{1.0f, 0, 0, 0, 0, 0},  
                 {0, 1.0f, 0, 0, 0, 0},  
                 {0, 0, 1.0f, 0, 0, 0},  
                 {0, 0, 0, 1.0f, 0, 0},  
                 {0, 0, 0, 0, 1.0f, 0},  
                 {0, 0, 0, 0, 0, 1.0f}}};
```

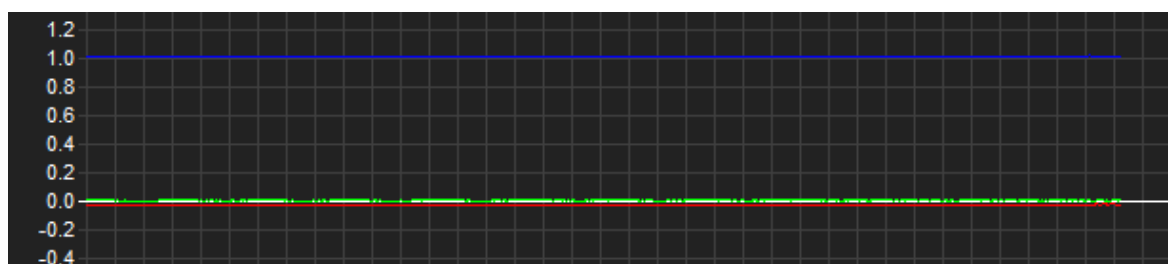
Przed zastosowaniem filtrów, sprawdzimy układ, a dokładnie odczyt danych, wysyłanie przez USB do komputera oraz wizualizację w czasie rzeczywistym.

Przebiegi danych z czujnika zaprezentowane na rysunkach 23 i 24.

Oś X – przebieg czerwonego koloru.

Oś Y – przebieg zielonego koloru.

Oś Z – przebieg niebieskiego koloru.



Rys.23 Dane akcelerometru



Rys.24 Dane żyroskopu

## 6.1 IMPLEMENTACJA FILTRU KOMPLEMENTARNEGO

W tym rozdziale został zaimplementowany algorytm filtra komplementarnego na mikrokontrolerze w języku C. Funkcja *complementary\_filter* (Listing 21) przejmuje dane z akcelerometru oraz żyroskopu. Za pomocą funkcji *atan* i *atan2* obliczamy kąty z danych akcelerometru (aby korzystać z tych funkcji, musimy podłączyć bibliotekę `<math.h>`).

*Listing 21 Funkcja filtra komplementarnego w języku C*

```
int32_t complementary_filter(float acc_data[], float gyr_data[])
{
    float acc_data_3[] = {0.0f, 0.0f, 0.0f};
    acc_data[0] = (180.0f/pi) * atan(acc_data[0]/sqrt(acc_data[1] *
        acc_data[1] + acc_data[2] * acc_data[2]));
    acc_data[1] = (180.0f/pi) * atan(acc_data[1]/sqrt(acc_data[0] *
        acc_data[0] + acc_data[2] * acc_data[2]));
    acc_data[2] = (180.0f/pi) * atan(acc_data[2]/sqrt(acc_data[0] *
        acc_data[0] + acc_data[1] * acc_data[1]));

    float k = 0.998f;

    for (int i = 0; i < 3; i++)
    {
        output_data[i] = k * (old_data[i] + gyr_data[i] * 0.1f) + (1-k) *
acc_data[i];
        old_data[i] = output_data[i];
    }

    return 0;
}
```

## 6.2 IMPLEMENTACJA FILTRU MADGWICKA

Końcowym praktycznym rozdziałem będzie implementacja filtra Madgwicka w języku C.

Na początku funkcji przekonwertujemy dane z żyroskopu ze stopni/sek na radiany/sek. Następnie obliczamy prędkość zmiany kwaternionów z danych żyroskopu według wzoru 2.15 (Listing 22)

*Listing 22 Fragment kodu filtra Madgwicka w języku C*

```
float gx = gyr_data[0] * (pi/180.0f);
float gy = gyr_data[1] * (pi/180.0f);
float gz = gyr_data[2] * (pi/180.0f);

qDot1 = 0.5f * (-q1 * gx - q2 * gy - q3 * gz);
qDot2 = 0.5f * (q0 * gx + q2 * gz - q3 * gy);
qDot3 = 0.5f * (q0 * gy - q1 * gz + q3 * gx);
qDot4 = 0.5f * (q0 * gz + q1 * gy - q2 * gx);
```

Dalej normalizujemy dane z akcelerometru i według wzoru 2.22 obliczamy spadek gradientu (Listing 23).

*Listing 23 Fragment kodu filtra Madgwicka w języku C*

```
s0 = _4q0 * q2q2 + _2q2 * ax + _4q0 * q1q1 - _2q1 * ay;
s1 = _4q1 * q3q3 - _2q3 * ax + 4.0f * q0q0 * q1 - _2q0 * ay - _4q1 +
    _8q1 * q1q1 + _8q1 * q2q2 + _4q1 * az;
s2 = 4.0f * q0q0 * q2 + _2q0 * ax + _4q2 * q3q3 - _2q3 * ay - _4q2 +
    _8q2 * q1q1 + _8q2 * q2q2 + _4q2 * az;
s3 = 4.0f * q1q1 * q3 - _2q1 * ax + 4.0f * q2q2 * q3 - _2q2 * ay;
recipNorm = invSqrt(s0 * s0 + s1 * s1 + s2 * s2 + s3 * s3);

s0 *= recipNorm;
s1 *= recipNorm;
s2 *= recipNorm;
s3 *= recipNorm;

qDot1 -= beta * s0;
qDot2 -= beta * s1;
qDot3 -= beta * s2;
qDot4 -= beta * s3;
```

Całkujemy szybkość zmian kwaternionów, aby uzyskać kwaternion, a następnie normalizujemy i konwertujemy na kąty Eulera (Listing 24).

*Listing 24 Fragment kodu filtra Madgwicka w języku C*

```
q0 += qDot1 * (1.0f / sampleFreq);
q1 += qDot2 * (1.0f / sampleFreq);
q2 += qDot3 * (1.0f / sampleFreq);
q3 += qDot4 * (1.0f / sampleFreq);

recipNorm = invSqrt(q0 * q0 + q1 * q1 + q2 * q2 + q3 * q3);
q0 *= recipNorm;
q1 *= recipNorm;
q2 *= recipNorm;
q3 *= recipNorm;

output_data[0] = (180.0f/pi) * atan2(2*(q0*q1+q2*q3),1-2*(q1*q1+q2*q2));
output_data[1] = (180.0f/pi) * asin(2*(q0*q2-q3*q1));
output_data[2] = (180.0f/pi) * atan2(2*(q0*q3+q1*q2),1-2*(q2*q2+q3*q3));
```

### 6.3 IMPLEMENTACJA FILTRU KALMANA

W tym rozdziale sprawdzimy działanie filtra Kalmana na układzie rzeczywistym. Definicja zmiennych pokazana w Listingu 19. Deklaracja głównej funkcji filtra pokazana w Listingu 25.

*Listing 25 Deklaracja funkcji filtra Kalmana w języku C*

```
int32_t kalman_filter(float acc_data[3], float gyr_data[3]);
```

Na początku funkcji obliczamy kąty z danych akcelerometru według wzorów 2.7 – 2.9. Następnie za pomocą funkcji *memset*, zerujemy zmienne które będą podszepnę do przechowywania tymczasowych wyników (listing 26).

*Listing 26 Fragment funkcji filtru Kalmana w języku C*

```
acc_data_[0] = (180.0f/pi) * atan(acc_data[0]/sqrt(acc_data[1] * acc_data[1] +
acc_data[2] * acc_data[2]));
acc_data_[1] = (180.0f/pi) * atan(acc_data[1]/sqrt(acc_data[0] * acc_data[0] +
acc_data[2] * acc_data[2]));
acc_data_[2] = (180.0f/pi) * atan(acc_data[2]/sqrt(acc_data[0] * acc_data[0] +
acc_data[1] * acc_data[1]));
memset(x_pri, 0.0f, sizeof(x_pri));
memset(p_pri, 0.0f, sizeof(p_pri));
```

Cała funkcja składa się wyłącznie z mnożenia macierzy. Biorąc pod uwagę, że w języku C mnożenie macierzy nie jest możliwe, dla tego stosujemy pętlę *for*. Listing 27 pokazuje fragment kodu danego filtra.

*Listing 27 Fragment funkcji filtru Kalmana w języku C*

```
// A * x_post
for (int i = 0; i < 6; i++)
{
    for (int j = 0; j < 6; j++)
    {
        mult_A_xpost[i] += A[i][j] * x_post[j];
    }
}
// B * gyr_data
for (int i = 0; i < 6; i++)
{
    for (int j = 0; j < 3; j++)
    {
        mult_B_gyr_data[i] += B[i][j] * gyr_data[j];
    }
}
//x_pri = A * x_post + B * gyr_data
for (int i = 0; i < 6; i++)
{
    x_pri[i] = mult_A_xpost[i] + mult_B_gyr_data[i];
}
```

Aby poprawić czytelność kodu stworzono funkcję *mult\_matrix* mnożącą macierze 6x6 i 6x3, dana funkcja jest pokazana w listingu 28.

*Listing 28 Funkcji dla mnożenia macierzy w języku C*

```
void mult_matrix(int m, int n, int l, float A[m][n], float B[n][l], float C[m][l])
{
    for (int k = 0; k < 6; k++)
    {
        for (int i = 0; i < 6; i++)
        {
            for (int j = 0; j < 6; j++)
            {
```

```

        C[k][i] += A[k][j] * B[j][i];
    }
}
}

```

## 6.4 WYNIKI

W tym rozdziale zostały zaprezentowane wyniki pracy algorytmów filtracji.

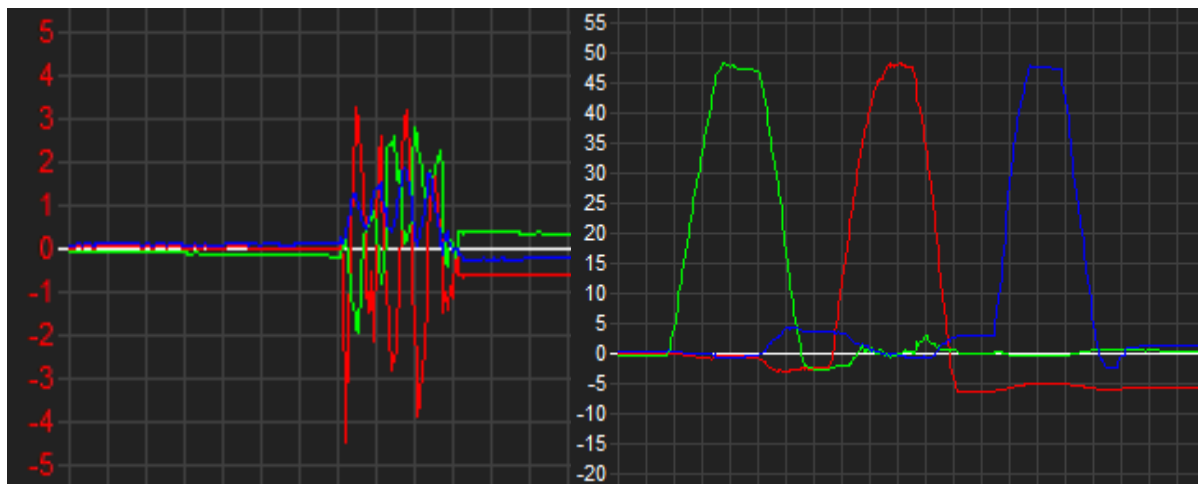
Testy przeprowadzono w następujący sposób:

- Układ sprawdzamy pod kątem stabilności podnosząc go w górę i w dół, również tą metodą sprawdzam, jak dobrze filtr radzi sobie z szumami.
- Sprawdzenie poprawności wyznaczenia kąta, a także szybkości reakcji na zmiany odchyłki

Wykresy 25, 26 i 27 przedstawiają najlepsze wyniki testów opisanych powyżej. Na koniec badań możemy powiedzieć, że algorytmy wszystkich 3 filtrów wykazały podobne wyniki, niemniej jednak wyróżniłbym filtr Madgwicka jako najlepszy, ponieważ zera osi po wychyleniu i późniejszym powrocie do pozycji początkowej są najbliższe rzeczywistego, maksymalny błąd podczas badań  $1.6^\circ$ , w przypadku filtra Kalmana –  $2.5^\circ$ , filtra komplementarnego – w granicach  $5^\circ$ .

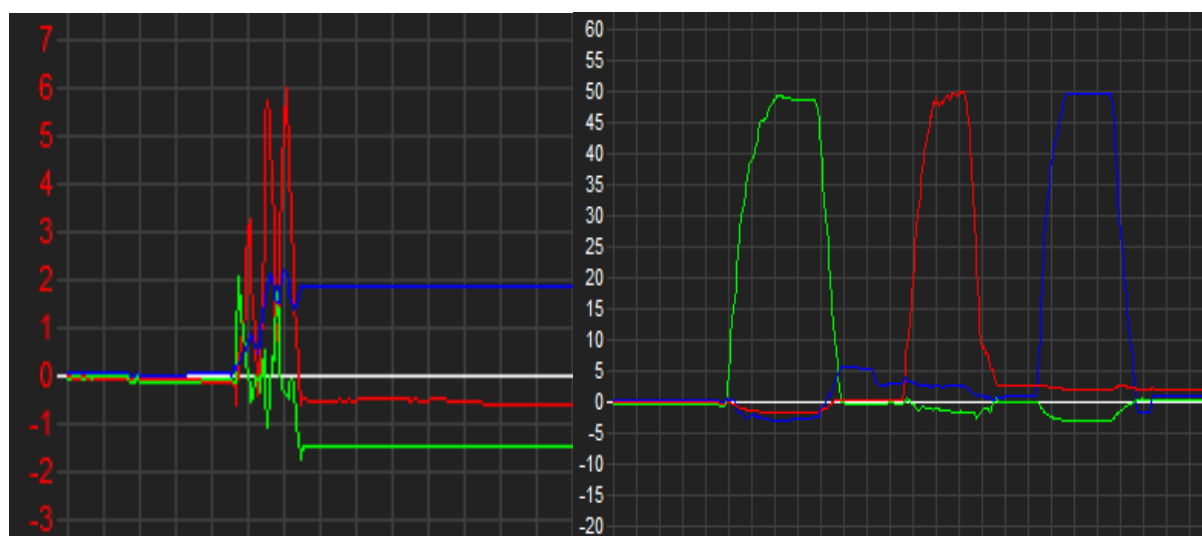
W teście wyznaczania nachylenia wszystkie 3 filtry poradziły sobie doskonale, wartości nachylenia zostały obliczone poprawnie, przy zachowaniu dobrej odpowiedzi i redukcji szumów.

### Filtr Komplementarny



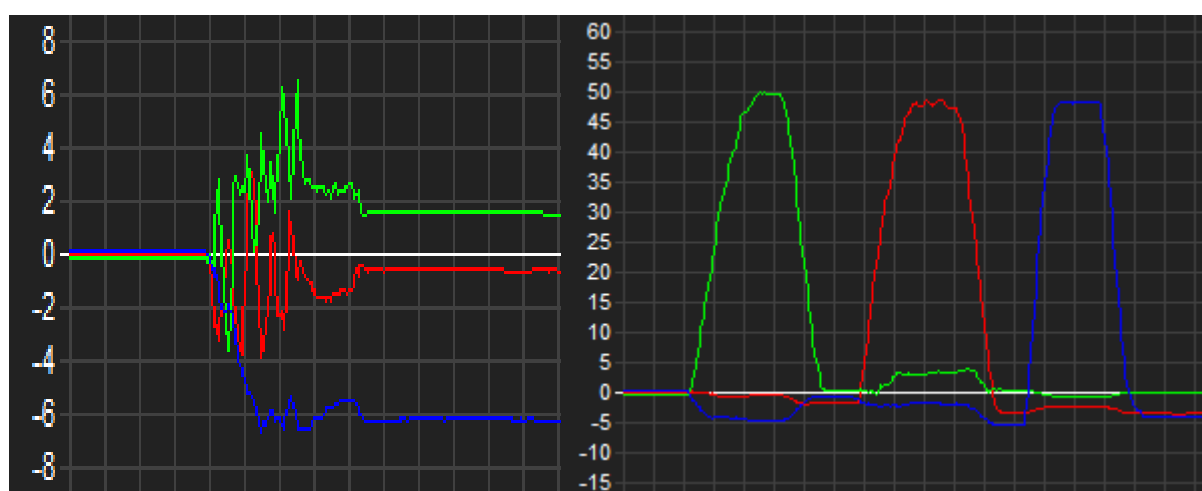
Rys.25 Wykres z Filtru Komplementarnego.

## Filtr Madgwicka



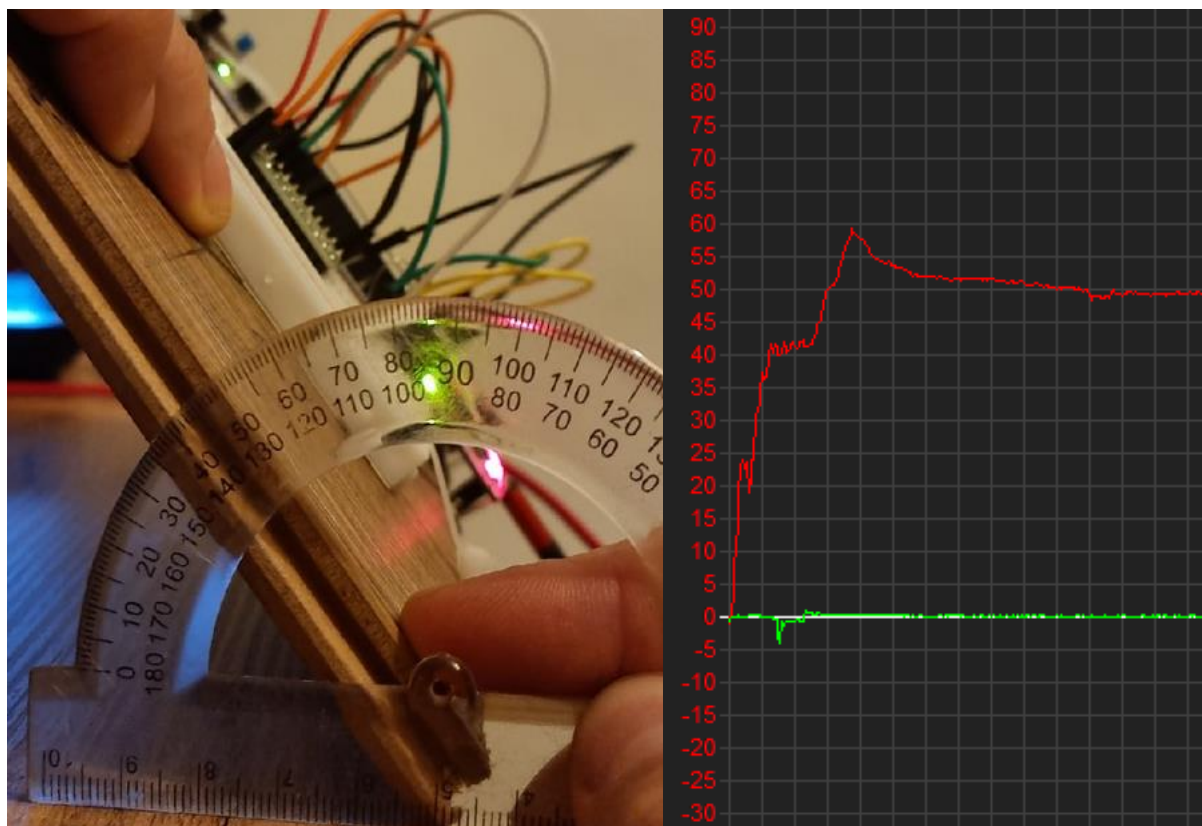
Rys.26 Wykres z Filtru Madgwicka.

## Filtr Kalmana



Rys.27 Wykres z Filtru Kalmana.

Aby zademonstrować poprawność działania systemu, przeprowadzono test z kątomierzem ręcznym, na podstawie wyników wcześniejszych badań wybrano filtr Madgwicka. Rysunek 28 pokazuje, że inklinometr działa poprawnie, kąt nachylenia na zdjęciu w lewej stronie jest taki sam jak na wykresie po prawej stronie i się równa  $50^\circ$ . Również cały proces demonstracji był nagrany na wideo.



Rys.28 Demonstracja sprawdzania kąta. Filtr Madgwicka.

## 7 PODSUMOWANIE

W niniejszej pracy zostało zbadane algorytmy trzech filtrów: filtru komplementarnego, filtru Madgwicka i filtru Kalmana do opracowania inklinometru mikroprocesorowego.

Układ rzeczywisty jest oparty na mikrokontroler STM32NULCEO – F756ZG oraz czujnik MEMS LSM6DSOX.

Badania przeprowadzono na różnych zbiorach danych za pomocą programu Matlab, następnie filtry zostały przepisane do C dla testów w czasie rzeczywistym.

Najlepszym filtrem okazał się filtr Madgwicka, który ma najlepsze wyniki zarówno w testach w Matlab, jak i w testach przeprowadzanych w czasie rzeczywistym. Istotnym warunkiem dobrej filtracji jest dobór parametrów filtru (współczynnik  $\beta$  dla filtru Madgwicka,  $k$  dla filtru komplementarnego oraz macierze kowariancji dla filtru Kalmana) oraz wstępna kalibracja czujnika.

Po tym wszystkim, możemy śmiało powiedzieć, że czujniki MEMS do rozwoju inklinometrów mikroprocesorowych z nawiązką spełniają wysokie wymagania rynku. Dzięki większej liczbie testów w czasie rzeczywistym, przeprowadzanych bez interwencji człowieka na specjalistycznych maszynach, można osiągnąć wyższą dokładność niż w niniejszej pracy.



## LITERATURA

1. Sebastian Madgwick. An efficient orientation filter for inertial and inertial/magnetic sensor arrays. [udostępniono 30.04.2010]. Pobrano: [https://www.x-io.co.uk/res/doc/madgwick\\_internal\\_report.pdf](https://www.x-io.co.uk/res/doc/madgwick_internal_report.pdf)
2. Pengfei Gui, Liqiong Tang, S.C. Mukhopadhyay. MEMS based IMU for tilting measurement: Comparison of complementary and Kalman filter based data fusion. [udostępniono 05.02.2018] Pobrano: [https://www.researchgate.net/publication/308850497\\_MEMS\\_based\\_IMU\\_for\\_tilting\\_measurement\\_Comparison\\_of\\_complementary\\_and\\_kalman\\_filter\\_based\\_data\\_fusion](https://www.researchgate.net/publication/308850497_MEMS_based_IMU_for_tilting_measurement_Comparison_of_complementary_and_kalman_filter_based_data_fusion)
3. Sebastian O.H. Madgwick, Andrew J.L. Harrison, Ravi Vaidyanathan. Estimation of IMU and MARG orientation using a gradient descent algorithm. [udostępniono 01.07.2011]. Pobrano: <https://ieeexplore.ieee.org/document/5975346>
4. Josef Justa, Václav Šmídl, Aleš Hamáček. Fast AHRS Filter for Accelerometer, Magnetometer, and Gyroscope Combination with Separated Sensor Corrections. [udostępniono 09.07.2020]. Pobrano: <https://www.mdpi.com/1424-8220/20/14/3824/pdf>
5. Jan Kędzierski. Filtr Kalmana - zastosowania w prostych układach sensorycznych. [udostępniono 15.03.2016]. Pobrano: [https://www.researchgate.net/profile/Jan-Kedzierski/publication/298352096\\_Filtr\\_Kalmana\\_-\\_zastosowania\\_w\\_prostych\\_ukladach\\_sensorycznych/links/56e8621b08ae166360e519c5/Filtr-Kalmana-zastosowania-w-prostych-ukladach-sensorycznych.pdf](https://www.researchgate.net/profile/Jan-Kedzierski/publication/298352096_Filtr_Kalmana_-_zastosowania_w_prostych_ukladach_sensorycznych/links/56e8621b08ae166360e519c5/Filtr-Kalmana-zastosowania-w-prostych-ukladach-sensorycznych.pdf)
6. Open source IMU and AHRS algorithms. [udostępniono 08.2020]. Pobrano: <https://x-io.co.uk/open-source-imu-and-ahrs-algorithms/>.
7. Datasheet - LSM6DSOX - iNEMO inertial module. [udostępniono 01.2019]. Pobrano: <https://www.st.com/resource/en/datasheet/lsm6dsox.pdf>
8. STM32 Nucleo-144 boards (MB1137) - User manual. [udostępniono 27.11.2014]. Pobrano: [https://www.st.com/resource/en/user\\_manual/dm00244518-stm32-nucleo144-boards-mb1137-stmicroelectronics.pdf](https://www.st.com/resource/en/user_manual/dm00244518-stm32-nucleo144-boards-mb1137-stmicroelectronics.pdf)
9. Reference Manual - RM0385- STM32F75xxx and STM32F74xxx advanced Arm®-based 32-bit MCUs. [udostępniono 06.2018]. Pobrano: [https://www.st.com/resource/en/reference\\_manual/dm00124865-stm32f75xxx-and-stm32f74xxx-advanced-arm-based-32-bit-mcus-stmicroelectronics.pdf](https://www.st.com/resource/en/reference_manual/dm00124865-stm32f75xxx-and-stm32f74xxx-advanced-arm-based-32-bit-mcus-stmicroelectronics.pdf)
10. STEVAL-MKI197V1 - Data brief. [udostępniono 10.2018]. Pobrano: [https://www.st.com/resource/en/data\\_brief/steval-mki197v1.pdf](https://www.st.com/resource/en/data_brief/steval-mki197v1.pdf)
11. Program "Serial Oscilloscope" [brak daty]. Pobrano: <https://www.oscilloscope-lib.com/>
12. Dokładny pomiar pochylenia obiektu za pomocą dedykowanych czujników ST MEMS. [online] [16.11.2020]. Pobrano: [https://www.compel.ru/wordpress/wp-content/uploads/2020/10/ris\\_8-2.png](https://www.compel.ru/wordpress/wp-content/uploads/2020/10/ris_8-2.png)
13. Madgwick Filter. [online] [brak daty]. Pobrano: <https://nitinjsanket.github.io/img/tutorials/attitudeest/MadgwickFilterOverview.png>
14. Greg Welch, Gary Bishop. An Introduction to the Kalman Filter [08.02.2001]. Pobrano: <https://d3i71xaburhd42.cloudfront.net/3d190dd77d51246d58fc70efa6d3c486f6b1da25/6-Figure1-2-1.png>
15. STM32 Nucleo-144 development board with STM32F756ZG MCU, supports Arduino, ST Zio and morpho connectivity [online] [brak daty]. Pobrano: <https://www.st.com/bin/e-commerce/api/image.PF266520.en.feature-description-include-personalized-no-cpn-medium.jpg>

16. LSM6DSOX adapter board for a standard DIL24 socket [online] [brak daty]. Pobrano:  
<https://www.st.com/bin/ecommerce/api/image.PF267129.en.feature-description-include-personalized-no-cpn-large.jpg>
17. Sterownik do czujnika LSM6DSOX napisany w języku programowania C. [online] [brak daty].  
Pobrano:  
<https://github.com/STMicroelectronics/lsm6dsox/tree/63e7471a85f2ef5992cd8c0aa653ea2f17065316>