Rok akademicki 2014/2015

Politechnika Warszawska
Wydział Elektroniki i Technik Informacyjnych
Instytut Informatyki

PRACA DYPLOMOWA INŻYNIERSKA

Łukasz Kamiński

Aplikacja do nawigacji inercyjnej na urządzenia mobilne z systemem Android

Opiekun pracy
Dr inż. Grzegorz Tarapata

Ocena: .....

.....

Podpis Przewodniczącego
Komisji Egzaminu Dyplomowego

Miejsce na zeskanowane lub wklejane zdjęcie

Kierunek: Informatyka

Specjalność: Inżynieria Systemów Informatycznych

Data urodzenia: 1992.07.13

Data rozpoczęcia studiów: 2011.10.01

Życiorys

Nazywam się Łukasz Kamiński. Urodziłem się 13. lipca 1992 r. w Łomży. Edukację zacząłem w Szkole Podstawowej nr 9 w Łomży, następnie kontynuowałem naukę w Publicznym Gimnazjum nr 6 w Łomży. Stąd zdałem do I LO im. Tadeusza Kościuszki w Łomży, gdzie przez trzy lata uczyłem się w klasie o profilu matematyczno-informatycznym. Po osiągnięciu wysokich wyników na maturze w 2011 r. rozpocząłem studia informatyki na Wydziale Elektroniki i Technik Informacyjnych.

.....
Podpis studenta

EGZAMIN DYPLOMOWY

Złożył egzamin dyplomowy w dniu ..... 2015 r

z wynikiem .....

Ogólny wynik studiów: .....

Dodatkowe wnioski i uwagi Komisji: .....

.....

.....

STRESZCZENIE

Nawigacja inercyjna jest od dawna stosowana w takich dziedzinach jak militaria i transport. Dzięki miniaturyzacji, wzrostowi jakości i spadkowi ceny sensorów inercyjnych, czujniki takie jak żyroskop, akcelerometr czy kompas zaczęły być montowane w telefonach komórkowych na skalę masową. Dotychczas jednak używano ich jedynie w celu wspomagania nawigacji GPS. Niniejsza praca opisuje realizację aplikacji na system Android do nawigacji użytkownika w obiektach zamkniętych przy użyciu wyłącznie czujników urządzenia. Przeanalizowano sposoby realizacji takiego programu oraz przedstawiono przykładową implementację. Opisano metody filtracji sygnałów pomiarowych jak również wykorzystanie fuzji sensorów.

Słowa kluczowe: nawigacja inercyjna, czujniki, fuzja sensorów, filtracja sygnałów, Android

---

INERTIAL NAVIGATION APPLICATION FOR DEVICES RUNNING ANDROID

Inertial navigation systems are widely used by military and in transport. Thanks to miniaturization, increase of quality and decrease of price of inertial sensors it has become common for smartphone producers to equip their products with sensors, in particular accelerometer, gyroscope and magnetometer. Thus far, inertial navigation was used as a support for GPS. This work presents prototype implementation of Android application used for navigating the user using only phone sensors. It also shows an analysis of methods that could be used for creating such application. Moreover it explains methods of digital signal filtering and usage of sensor fusion.

Keywords: inertial navigation, sensors, sensor fusion, digital signal filtering, Android OS

Składam serdeczne podziękowania dr inż. Grzegorzowi Tarapata za pomoc oraz wszystkie cenne wskazówki, bez których napisanie niniejszej pracy byłoby niemożliwe.

1 Spis treści

2 Wstęp

2.1 Motywacja

Dla urządzeń mobilnych z systemem Android istnieje wiele aplikacji umożliwiających lokalizację urządzenia (a z nim użytkownika) przy pomocy GPS i nawigację do wybranego adresu. Jednak brakuje rozwiązań problemu nawigacji w przypadku kiedy korzystanie z GPS lub lokalizacja na podstawie nadajników GSM jest utrudniona bądź niemożliwa.

Przykładem takiej sytuacji jest wnętrze budynku, takiego jak wielkie centrum handlowe, duży urząd lub inny budynek użyteczności publicznej. W takim przypadku nawigacja oparta o GPS nie będzie w stanie określić położenia użytkownika na tyle dokładnie by wskazać mu drogę do zamierzonego celu lub do wyjścia. Pozostaje też kwestia posiadania planów budynku, na których taki system miałby się opierać.

Alternatywą w tym przypadku jest nawigacja inercyjna (inaczej inercjalna lub bezwładnościowa). System nawigacji inercyjnej korzysta z wbudowanych w urządzenie mobilne czujników: akcelerometru, żyroskopu i kompasu. Przy ich użyciu system ten stara się oszacować położenie użytkownika w odniesieniu do punktu startowego.

2.2 Cele pracy

Celem pracy jest stworzenie aplikacji pozwalającej na dotarcie do celu lub odnalezienie drogi powrotnej w sytuacji, w której niemożliwe lub utrudnione jest użycie nawigacji GPS. Program powinien potrafić zarejestrować trasę użytkownika i, na jego prośbę, poprowadzić go tą samą drogą do punktu wyjścia.

Założenia:

- telefon z systemem Android w wersji co najmniej 4.4 (KitKat), z co najmniej dwurdzeniowym procesorem
- użytkownik podczas korzystania z aplikacji trzyma telefon przed sobą, ekranem do góry
- aplikacja obsługuje rozpoznawanie zakrętów (oznaczanie ręczne lub wykrywanie automatyczne)
- ręczne lub automatyczne dostosowywanie długości kroku użytkownika
- użytkownik podczas używania aplikacji idzie, nie biegnie

3 Technologie zastosowane w procesie tworzenia aplikacji

3.1 Stos technologiczny

3.1.1 Programowanie na system Android

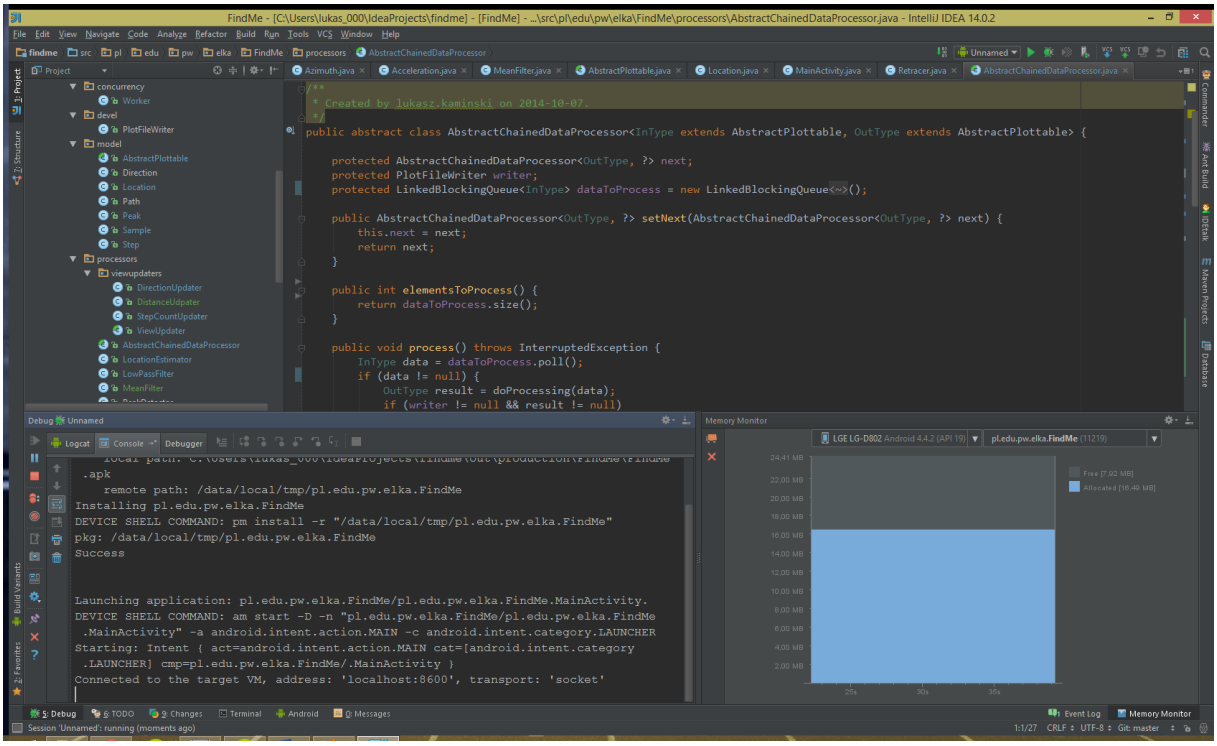
Podstawą do tworzenia aplikacji na platformę Android jest Android SDK – Android Software Developer Kit. Jest to zestaw bibliotek wraz z kodem źródłowym, niezbędnych w tworzeniu aplikacji na wspomniany system operacyjny. Dodatkowo na SDK składają się: dokumentacja, skompilowane biblioteki oraz przykładowe kody źródłowe. Biblioteki te umożliwiają użycie takich komponentów jak sensory, obsługa dotyku, komunikacja międzyprocesowa i wielu innych.

Niezbędne jest także użycie ADB – Android Debug Bridge. Jest to wszechstronne narzędzie obsługujące połączenie komputera programisty z telefonem lub emulatorem urządzenia. ADB umożliwia instalację skompilowanej aplikacji na urządzeniu, uruchomienie w trybie debugowania, przeglądanie logów i wiele innych. W przypadku tej pracy inżynierskiej użycie emulatora urządzenia nie było możliwe, gdyż w obecnej wersji emulator nie obsługuje symulowania działania czujników takich jak akcelerometr czy żyroskop.

Aplikacje dla systemu Android pisane są w języku Java, w tym przypadku Java 1.7.

3.1.2 Opis środowiska programistycznego

Aplikację wykonano przy użyciu profesjonalnych narzędzi oraz nowoczesnych technologii. Na zintegrowane środowisko programistyczne wybrano JetBrains IntelliJ Idea 14 Ultimate – lidera wśród środowisk do programowania w językach na maszynę wirtualną Java. Zintegrowane środowisko programistyczne (ang. *Integrated Development Environment – IDE*) to użyteczne narzędzie wspierające programistę w całym procesie wytwarzania oprogramowania. IDE dba o jakość i poprawność kodu, porządkuje pliki projektu, automatyzuje konfigurację, kompilację, budowę oraz instalację aplikacji. Przykładowy ekran obrazujący część funkcjonalności przedstawiono na rys. 1. Jak widać aplikacja posiada wiele funkcji między innymi narzędzia do debugowania, monitor zajętości pamięci, widok na wyjście konsoli, kolorowanie składni, podpowiedzi w kodzie, dynamiczne sprawdzanie błędów, wspomaganie nawigacji po plikach projektu. Wszystkie wymienione przykłady stanowią jedynie niewielką część listy funkcjonalności opisywanego środowiska. Jego zadaniem jest wspomagać programistę, tak aby mógł skoncentrować się na rozwiązywanym problemie.



Rys. 1. Ekran aplikacji IntelliJ Idea

3.1.3 Nowoczesny warsztat programistyczny

Docelowo narzędziem budującym całą aplikację miał być Gradle, jednak ostateczny wybór padł na nieco starszą technologię Ant. Ta decyzja została podjęta ze względu na to, że Ant jest narzędziem sprawdzonym i w zupełności wystarczającym przy tak małym projekcie. Ant jest narzędziem do budowania projektów, obecnie opiekę nad nim sprawuje fundacja Apache. Dzięki oprogramowaniu Ant programista nie musi martwić się o kompilację oraz dołączanie bibliotek do projektu.

Każdy doświadczony programista wie, że system kontroli wersji jest niezbędny w każdym projekcie i może okazać się nieocenioną pomocą. W opisywanej aplikacji rolę systemu kontroli wersji pełniła aplikacja Git – najpopularniejsze obok SVN narzędzie do zarządzania kodem. Repozytorium zostało umiejscowione w serwisie internetowym zajmującym się przetrzymywaniem repozytoriów kodu źródłowego, stronie bitbucket.org.

Urządzeniem docelowym, na którym instalowano powstającą aplikację był telefon komórkowy LG G2 z najnowszym wówczas systemem operacyjnym Android 4.4.2 KitKat. Jest to urządzenie wysokiej klasy, którego specyfikacja została dokładnie opisana w rozdz. 5.1.

Aplikacja została wytworzona na komputerze osobistym pracującym pod kontrolą systemu operacyjnego Windows 8.1 Professional.

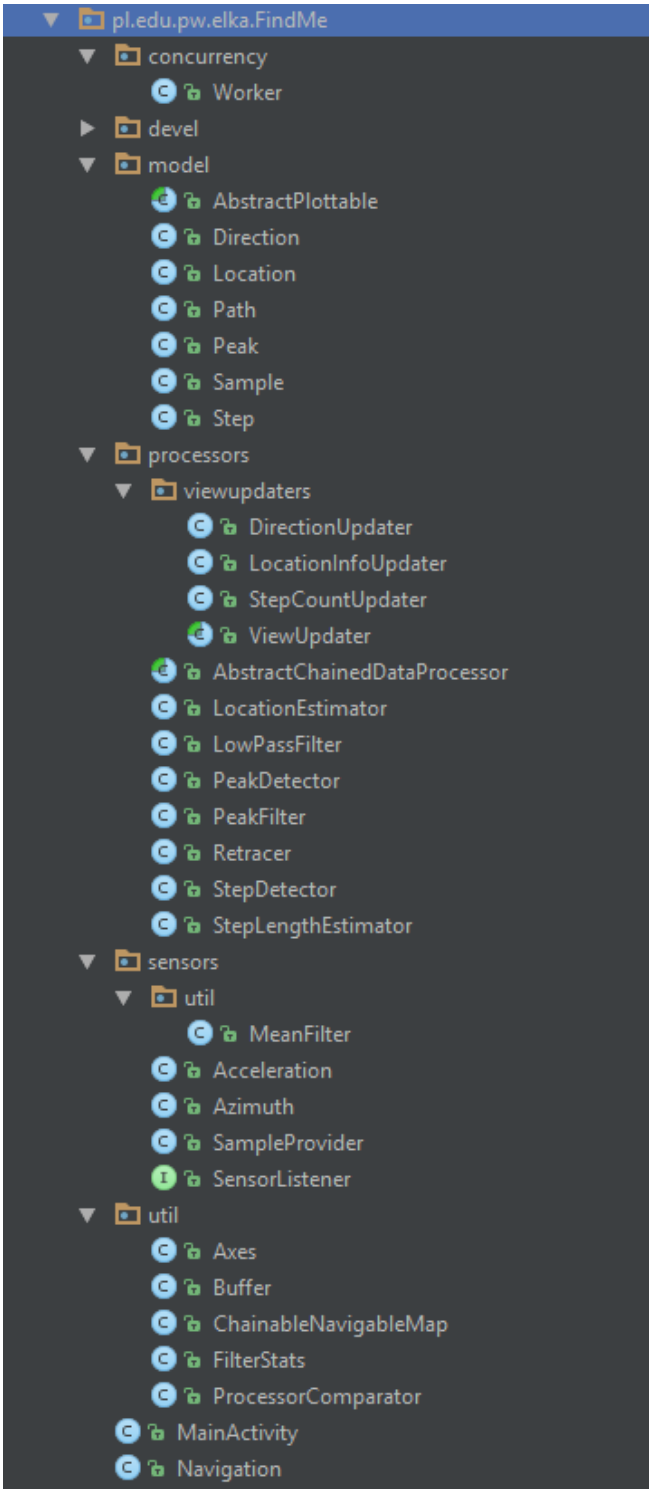
3.2 Opis stosowanych praktyk programistycznych

Już dawno społeczność programistyczna zrozumiała zalety wytwarzania kodu o jak najlepszej jakości. Rodzi to mniej problemów w fazie utrzymania aplikacji, sprawia że oprogramowanie jest bardziej niezawodne oraz ułatwia jego rozwój.

Rozwiązywany problem został zdekomponowany na małe, logicznie powiązane części i na tej podstawie został napisany kod aplikacji. Pisano go tak, aby odpowiadał dziedzinie problemu, zgodnie z podejściem Domain-Driven Design (DDD). DDD jest sposobem wytwarzania oprogramowania kładącego nacisk na takie definiowanie obiektów i komponentów systemu oraz ich zachowań, aby wiernie odzwierciedlały rzeczywistość. Jak pokazuje rys. 2. rozbicie problemu na wiele małych podproblemów zaowocowało dużą liczbą klas o jednoznacznych, zrozumiałych nazwach.

Przy nazewnictwie obiektów, zmiennych i metod zastosowano najpopularniejsze podejście camelCase czyli system notacji ciągów wyrazów, w którym kolejne wyrazy pisze się łącznie, rozpoczynając każde następne słowo wielką literą.

Konstrukcja obiektów i metod zakłada nie więcej niż dwa argumenty na metodę, jak najmniej pól w klasie i wiele innych praktyk, dzięki którym kod jest przejrzysty, czytelny i zrozumiały.

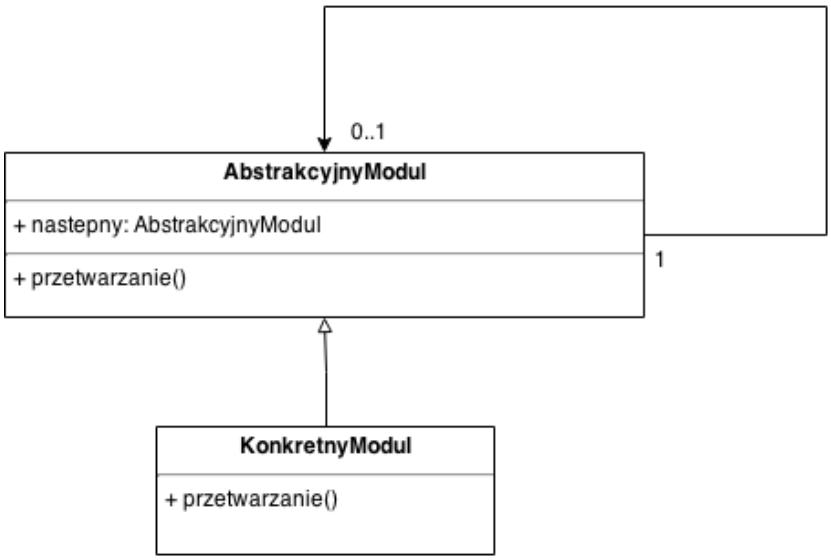


Rys. 2. Lista klas i pakietów aplikacji

3.3 Przykłady zastosowania wzorców w programowaniu

3.3.1 Wzorzec łańcucha zobowiązań

Najważniejszym wzorcem zastosowanym podczas tworzenia opisywanej aplikacji jest dostosowany do potrzeb problemu wzorzec projektowy łańcucha zobowiązań.



Rys. 3. Diagram klas dla wzorca łańcuch zobowiązań.

Jak pokazano na rys. 3. zastosowanie powyższego wzorca polega na stworzeniu abstrakcyjnej klasy przetwarzającej dane i posiadającej wskazanie na następny element łańcucha. Każda klasa rozszerzająca klasę abstrakcyjną definiuje sposób przetwarzania danych. Zasadza działania według tego wzorca jest następująca: dane wejściowe trafiają do każdego elementu łańcucha i są przetwarzane zgodnie z definicją (na rys. 3 byłaby to metoda *przetwarzanie()*). Po ukończeniu pracy z informacją klasa przekazuje ją do następnego ogniwa łańcucha jeśli ten jest zdefiniowany.

W opisywanej implementacji klasa abstrakcyjna narzuca także deklarację przez programistę typów danych wejściowych i wyjściowych w klasach rozszerzających. Dodatkowo każda klasa rozszerzająca klasę bazową posiada kolejkę danych oczekujących na przetworzenie. Dzięki powyższym zabiegom działanie aplikacji mogło zostać zrównoleglone, a wymuszenie deklaracji typów powoduje, że poprawność logiczna łańcucha przetwarzania jest sprawdzana już na etapie kompilacji – poprzez mechanizm statycznego sprawdzania typów kompilatora języka Java.

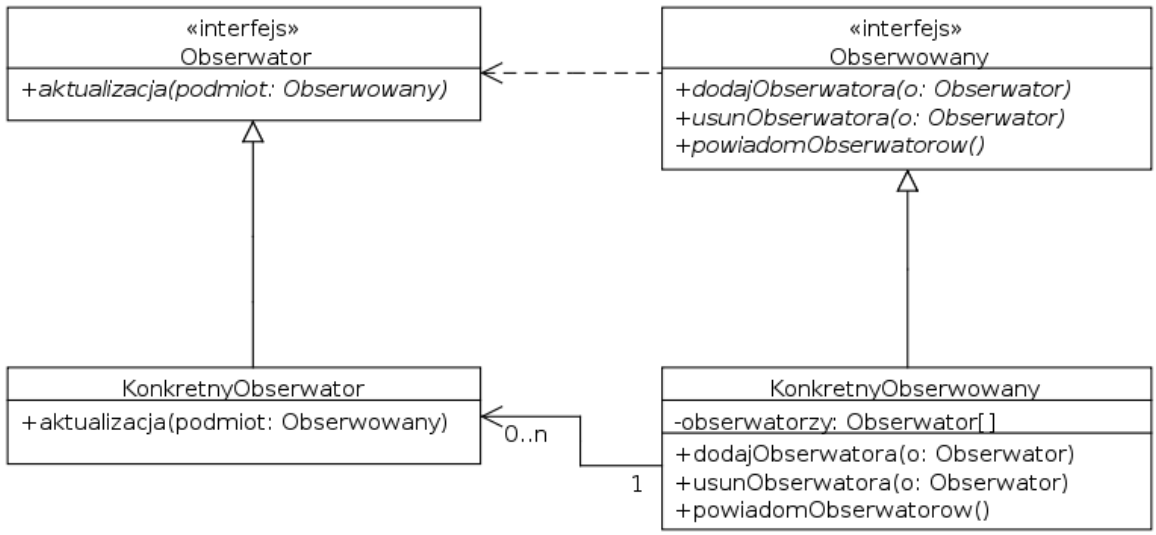
Wzorzec łańcucha zobowiązań został wykorzystany ze względu na swoje zalety. Pierwszą z nich jest łatwość zmieniania łańcucha (nawet w trakcie działania programu). W

dowolnej chwili programista może podmienić ogniwo, zamienić dwa elementy miejscami czy dodać lub usunąć moduł w dowolnym miejscu. Znacznie ułatwia to prace nad aplikacją – przykładowo w trakcie rozwoju aplikacji wstawiono do łańcucha moduły wyświetlające na ekranie dane przez nie przepływające, co znacznie pomogło w procesie znajdowania błędów w programie.

Kolejnym atutem powyższego wzorca jest niezależność modułów. Programowanie dowolnego ogniwa nie zakłada znajomości działania pozostałych, jak również ich ułożenia w łańcuchu, co ułatwia skupienie się na rozwiązywanym problemie. Wynikiem podziału aplikacji na małe, niezależne problemy jest ułatwienie śledzenia błędów.

3.3.2 Wzorzec obserwatora

Jest to wzorzec projektowy należący do grupy wzorców czynnościowych. Używany jest do powiadamiania zainteresowanych obiektów o pewnych (ustalonych) zmianach.



Rys. 4. Diagram klas dla wzorca obserwatora.

Jak widać na diagramie klas (rys. 4) wzorzec składa się z dwóch zależnych interfejsów – obserwatora deklarującego metodę obsługi powiadomienia o nowych danych oraz obserwowanego deklarującego metody zarządzające listą obserwatorów oraz metodę powiadamiającą wszystkich zainteresowanych.

Wzorzec obserwatora w niniejszej pracy wykorzystany w dwóch miejscach. Pierwszym z nich jest korzystanie z czujników telefonu – użycie wzorca w tym przypadku zostało wymuszone przez interfejs programistyczny systemu Android. Po zadeklarowaniu chęci

korzystania z sensora ten będzie regularnie (z żądaną częstotliwością) wywoływał zdefiniowaną u obserwatora metodę przekazując odczyty.

Drugim przypadkiem są klasy przykrywające wykorzystanie czujników i znajdujące się w pakiecie *pl.edu.pw.elka.FindMe.sensors*. W szczególności klasy *Acceleration* i *Azimuth*, których obiekty dostarczają informacji o przyspieszeniu liniowym urządzenia oraz kierunku poruszania się – są to informacje generowane na podstawie danych z akcelerometru, kompasu i żyroskopu. Obiekty wspomnianych klas są jednocześnie obserwatorami (dla czujników telefonu) i obserwowanymi (przez moduł generujący dane przekazywane do łańcucha przetwarzania).

W obu przypadkach wykorzystano strategię wypychania, w której obiekt obserwowany sam przekazuje dane z każdym wywołaniem metody aktualizacji w obiektach obserwowanych. Jest to przeciwieństwo dla strategii wyciągania, w której obserwator przechowuje referencję do obiektu obserwowanego i, w przypadku wywołania metody aktualizacji, sam pobiera od niego potrzebne informacje.

3.3.3 Wzorzec puli obiektów

Zgodnie ze wzorcem puli obiektów zainicjowane obiekty umieszczane są w pewnej kolekcji i są w stanie gotowości do użycia. Pula obiektów jest obsługiwana przez klientów puli zwanych pracownikami. Pracownik żąda obiektu z puli i wykonuje na nim pewne operacje. Po skończeniu nie niszczy obiektu tylko zwraca go do puli. Ze względu na sposób działania wzorzec ten po angielsku nazywa się *worker-crew pattern* co można przetłumaczyć na wzorzec ekipy pracowniczej, co trafniej oddaje filozofię pracy tego wzorca.

W opisywanej implementacji rolę puli obiektów wypełnia kolejka FIFO, w której znajdują się zainicjowane moduły łańcucha przetwarzania. Każdy z nich definiuję metodę *doProcessing()* (wymusza to klasa nadrzędna), w której opisany jest sposób przetwarzania danych oczekujących w kolejce (por. rozdział 4.1). Pracownikami w tym przypadku są specjalnie stworzone na tę potrzebę obiekty klasy *Worker*. Każdy z nich startuje własny wątek, po czym cyklicznie pobiera moduł z kolejki, wywołuje na nim metodę *doProcessing()* a następnie odkłada moduł na koniec kolejki.

Zaletą wykorzystania tego rozwiązania jest zrównoleglenie przetwarzania danych i wyższa wydajność oraz jakość pracy aplikacji. Także dzięki użyciu wzorca puli obiektów aplikacja może dostosować się do mocy obliczeniowej urządzenia, na którym pracuje – poprzez zmniejszenie liczby wątków dla pracowników.

Do wad należy zaliczyć brak ustalonej kolejności obsługi modułów w puli. W efekcie moduły musiały zostać wyposażone w kolejki danych oczekujących na przetworzenie. Skutkiem tego jest zwiększenie ilości miejsc, w których używane są klasy bezpieczne dla wykorzystania wielowątkowego (na przykład listy synchronizowane). Ich wydajność jest niższa niż wydajność ich jednowątkowych odpowiedników, zwiększa się też podatność aplikacji na błędy.

4 Przegląd stanu wiedzy

4.1 Wstęp teoretyczny

Lokalizacja inercyjna znalazła dotychczas zastosowanie w wielu dziedzinach takich jak żegluga morska, militaria, robotyka, modelarstwo i innych [1]. Miniaturyzacja i zwiększenie precyzji urządzeń pomiarowych pozwoliło na zastosowanie tego typu lokalizacji na skalę masową w urządzeniach mobilnych.

Czujniki w telefonach są wykorzystywane przez wiele aplikacji, od prostych poziomic i kompasów, wykorzystujących pojedyncze sensory, aż po skomplikowane systemy do śledzenia aktywności fizycznej użytkownika integrujące wiele urządzeń pomiarowych. Sama nawigacja inercjalna w smartfonach dotychczas sprawdzała się głównie jako wspomaganie nawigacji GPS (np. zastępczo w tunelach).

System opracowany w ramach tej pracy inżynierskiej ma na celu ułatwienie znalezienia drogi powrotnej w dużych obiektach takich jak urzędy, centra handlowe, lotniska, zoo, lasy itp. bazując na drodze przebytej przez użytkownika – droga ta jest śledzona metodą inercyjną. Taka aplikacja jest podstawą do tworzenia programów wspomagających codzienne zadania. Dla przykładu dodanie komunikatów głosowych pomogłoby osobom niewidomym lub niedowidzącym. Innym przykładem użycia może być stworzenie elektronicznego przewodnika po obiekcie – muzeum lub galerii. W szczególności taka aplikacja daje podbudowę do stworzenia systemu nawigacji w budynkach opartego o plany pomieszczeń – na przykład plany ewakuacyjne.

Istnieje kilka rozwiązań dla nawigacji w budynkach, jednak systemy lokalizacji bezwładnościowej pracują na dedykowanych urządzeniach. Istnieją nieliczne aplikacje na iOS, Windows Phone oraz Android, w których algorytmy wykrywania ścieżki bazują na analizie planu budynku oraz sile sygnału nadajników Wi-Fi. Czujniki takie jak żyroskop czy akcelerometr jeśli są używane to pełnią jedynie rolę wspomagania powyższych algorytmów.

4.1.1 Metody lokalizacji inercyjnej

Metoda całkowania przyspieszeń

Głównym problemem, który należy rozwiązać jest znalezienie przemieszczenia urządzenia (a co za tym idzie użytkownika). Akcelerometr i żyroskop dostarczają informacje o chwilowych przyspieszeniach we wszystkich sześciu stopniach swobody urządzenia.

Przyspieszenie liniowe jest definiowane jako pochodną prędkości po czasie:

$$a = \frac{dv}{dt} \qquad (1)$$

Gdzie v oznacza prędkość, a t – czas. Na podstawie powyższego wzoru (1) można wyprowadzić zależność odwrotną pozwalającą obliczyć prędkość na podstawie przyspieszenia:

$$v(t) = \int_{t_0}^{t_1} a\, dt + v(t_0) \qquad (2)$$

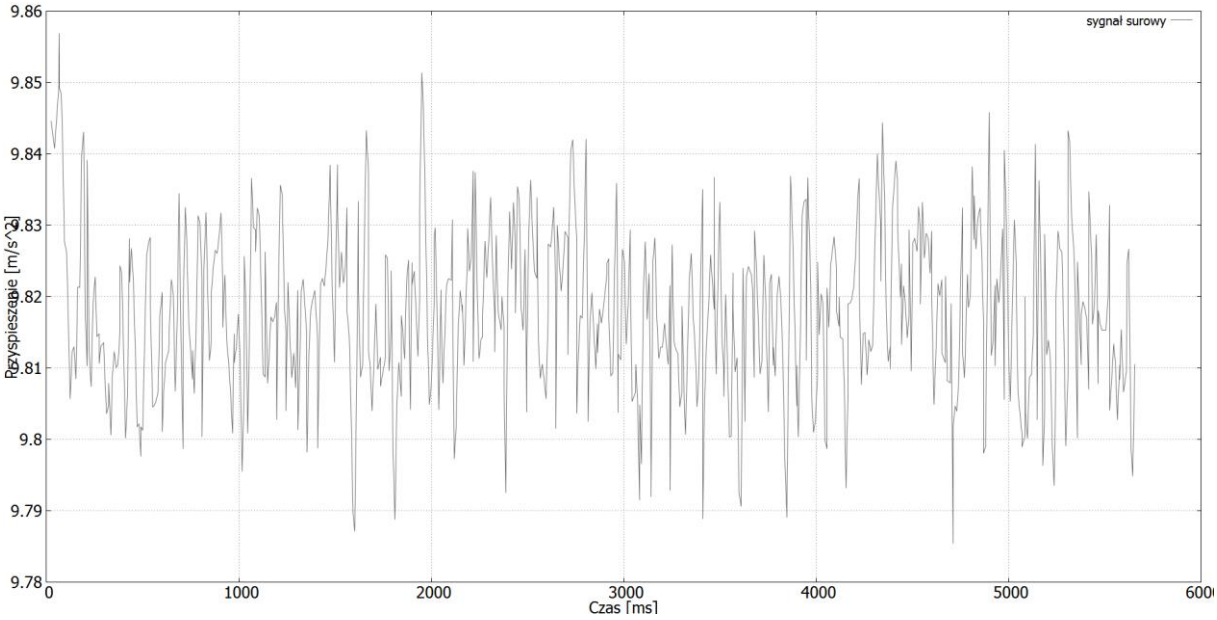
Znając prędkość i używając wzoru podobnego do (2) można obliczyć drogę:

$$s(t) = \int_{t_0}^{t_1} v\, dt + s(t_0) \qquad (3)$$

Intuicyjnym wydaje się być podejście polegające na wykorzystaniu wzorów (2) i (3) czyli całkowaniu przyspieszeń po czasie celem otrzymania prędkości, a następnie scałkowanie tychże otrzymując w wyniku szukany wektor przemieszczenia chwilowego. Takie rozwiązanie jest stosowane i sprawdza się w takich dziedzinach jak żegluga morska czy lokalizacja dronów, gdzie czujniki są dedykowane i znacznie dokładniejsze niż te montowane w smartfonach.

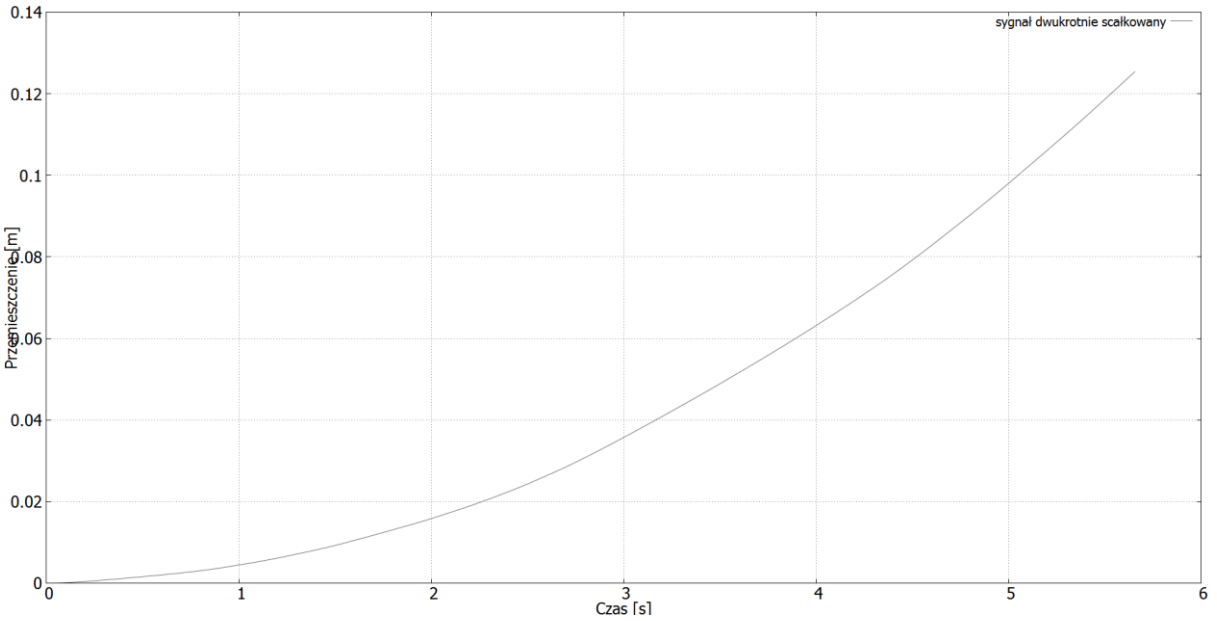
W urządzeniach mobilnych to podejście musiało zostać odrzucone. Odczyty z urządzeń pomiarowych są obarczone narastającym w czasie błędem zwanym dryfem. Są także zaszumione.

Wykonano eksperyment, w którym obserwowano przebieg sygnału akcelerometru w osi pionowej (Z), podczas gdy telefon przez ponad pięć sekund leżał nieruchomo, płasko na stole.



Rys. 5. Przebieg szumu akcelerometru w osi Z.

Przebieg zarejestrowanego szumu (rys. 5) został dwukrotnie poddany całkowaniu metodą numeryczną. Wynik tej operacji przedstawiono na rys. 6.



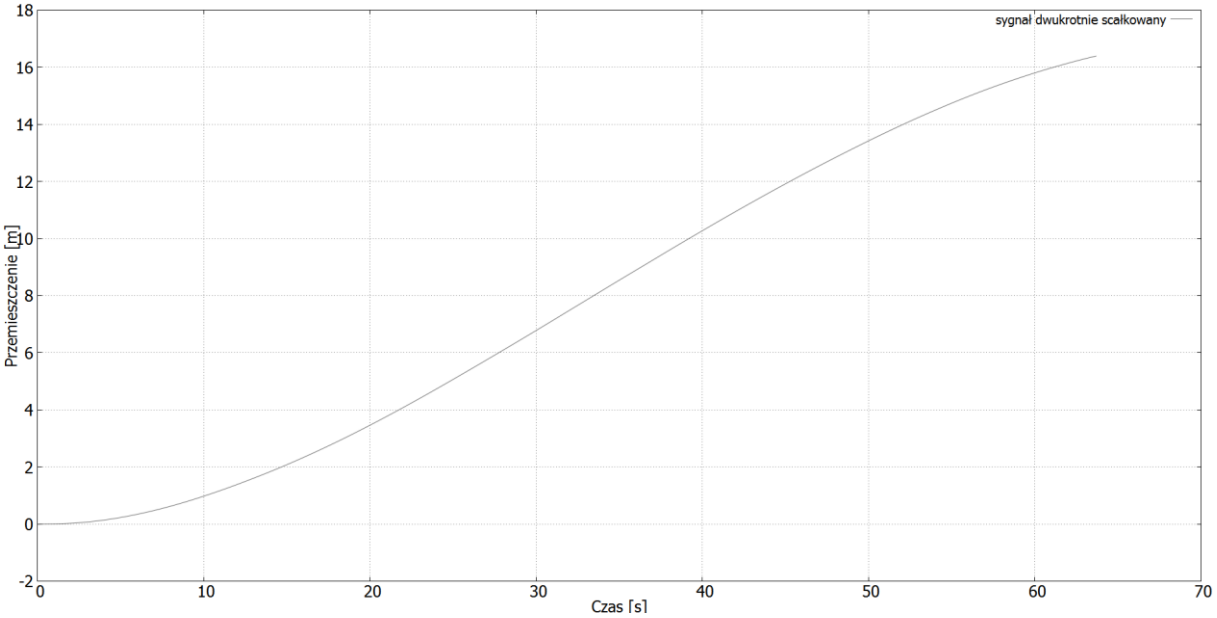
Rys. 6. Wykres dwukrotnie scałkowanego przebiegu szumu akcelerometru.

Zaobserwowano dryf wartości, w ciągu niecałych 6 sekund telefon zarejestrował przemieszczenie ok. 13 cm.

Akcelerometr podaje dane o przyspieszeniach z uwzględnieniem przyspieszenia ziemskiego. W powyższym teście, racji tego że telefon leżał płasko, od informacji z czujnika odejmowano wartość stałej g. Jednakże w ogólnym przypadku należy odjąć wartości

składowych siły grawitacji działających na każdą z osi układu współrzędnych urządzenia. W tym celu należy najpierw uzyskać (np. przy użyciu żyroskopu) informację o obrocie urządzenia, która jest podatna na błędy z racji niedokładności sensorów. Błąd pomiaru kąta obrotu dodaje stałą do wartości przyspieszenia liniowego, która podczas całkowania zostaje wyolbrzymiona.

Przeprowadzono test, symulując pomyłkę szacowania kąta o 0,8 stopnia. Telefon przechylono pod kątem 0,8 stopnia do poziomu i w takiej pozycji obserwowano szum akcelerometru przez minutę. Uzyskany przebieg sygnału poddano dwukrotnemu całkowaniu otrzymując rezultat jak na rys. 7. Po minucie w bezruchu urządzenie zarejestrowało przemieszczenie o 16 m.

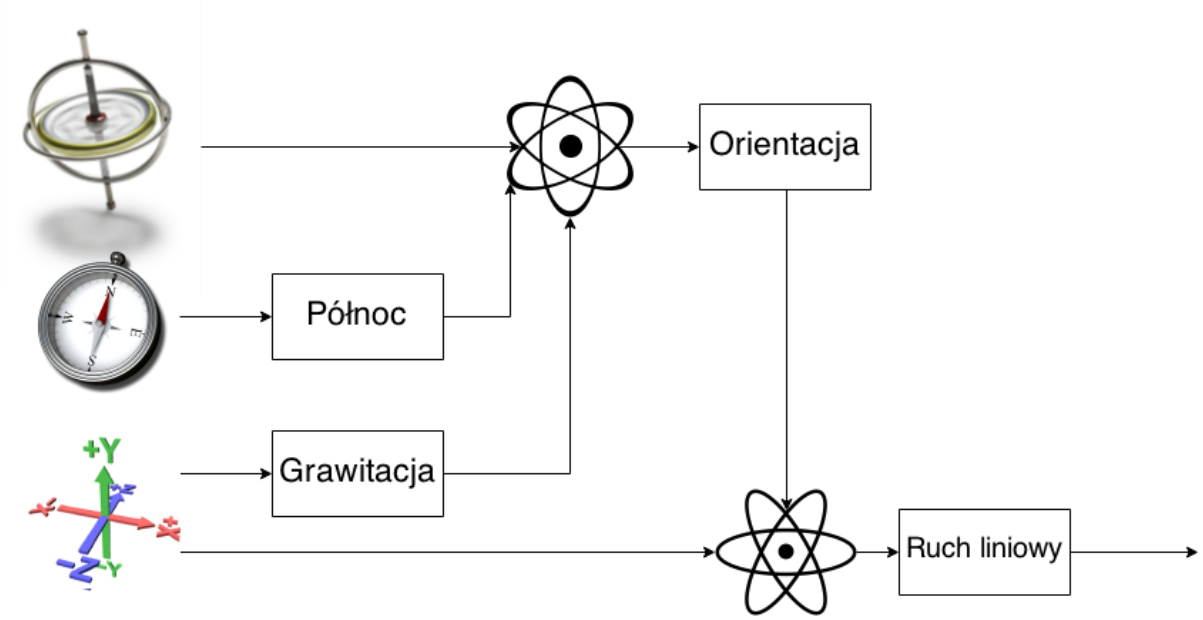


Rys. 7. Wykres podwójnego całkowania szumu akcelerometru ze składową stałą.

Powyższe eksperymenty pokazują, że wykorzystanie opisywanej metody jest niemożliwe z racji na wyolbrzymianie niedokładności urządzeń pomiarowych.

Metoda wykrywania kroków i kierunku poruszania

Drugim podejściem jest integracja różnych czujników oraz implementacja systemu liczącego kroki użytkownika, mierzącego ich długość oraz zwrot.



Rys. 8. Schemat obrazujący sposób integracji czujników.

Jak widać na rys. 8 kompas dostarcza informacji o kierunku N, a akcelerometr o składowych siły grawitacji. Dzięki temu można dokładnie oszacować obrót urządzenia. Dodatkowo żyroskop daje dobrą odpowiedź w krótkim czasie, co pozwala wykryć zmianę obrotu. Integracja powyższych sensorów daje możliwość dokładnego szacowania azymutu urządzenia. Znając kierunek poruszania można użyć danych z akcelerometru do wykrywania kroków i w ten sposób śledzić trasę użytkownika. To podejście zostało wybrane do realizacji celów niniejszej pracy.

4.1.2 Czujniki stosowane w smartfonach

System Android wspiera jedenaście typów sensorów wbudowanych w urządzenie [2]. Na tę liczbę składają się:

- akcelerometr
- termometr (temperatura otoczenia)
- czujnik grawitacji
- żyroskop
- czujnik oświetlenia
- czujnik przyspieszenia liniowego
- kompas
- barometr

- czujnik zbliżenia
- czujnik wilgotności
- termometr (temperatura urządzenia)

Każdy z nich może dostarczać dane do odbiorników z jedną z czterech zdefiniowanych częstotliwości próbkowania:

- najszybsza –maksymalna częstotliwość pomiarowa czujnika
- częstotliwość dla gier – ok. 50 Hz
- częstotliwość dla aplikacji – ok. 16,7 Hz
- częstotliwość normalna – ok. 5 Hz

Dodatkowo programista może wybrać własną częstotliwość.

Na rynku znajduje się wiele telefonów od wielu producentów, stąd wiele możliwych kombinacji producentów i modeli czujników montowanych w telefonach.

Tab. 1. Porównanie czujników przyspieszenia zainstalowanych w różnych telefonach

Model telefonu	**Producent czujnika**	**Pobór mocy**	**Rozdzielczość**	**Minimalny okres próbkowania**
LG G2	STMicroelectronics	0,28 mA	0,001 m/s^2	8333 µs
Samsung Galaxy S3	STMicroelectronics	0,23 mA	0,010 m/s^2	10000 us
Samsung Galaxy Note 2	Invensense	0,25 mA	0,001 m/s^2	5000 µs
Samsung Galaxy Ace 2	STMicroelectronics	0,25 mA	0,005 m/s^2	5000 µs
Sony Xperia Tipo	Bosch	0,2 mA	0,077 m/s^2	Brak danych

Jak pokazano w Tab. 1 na przykładzie akcelerometrów czujniki montowane w smartfonach różnią się parametrami. Naturalnym jest, że programista aplikacji korzystającej z sensora nie może panować nad tym z jakiego czujnika jego program będzie korzystał. W związku z tym Android wprowadza jednolity sposób korzystania z czujników znacznie ułatwiając tworzenie aplikacji, które je wykorzystują. Programista nie musi znać producenta, modelu urządzenia czy specyfikacji czujników. Wystarczy, że zadeklaruje w programie chęć

korzystania z sensora podając obiekt odbierający dane (implementujący odpowiedni interfejs [2]), żądany typ sensora oraz częstość próbkowania. Na listingu 1 znajduje się przykład zarejestrowania odbiornika dla danych z czujnika przyspieszenia liniowego:

```
Sensor linearAcceleration = sensorManager.getDefaultSensor(Sensor.TYPE_LINEAR_ACCELERATION);
sensorManager.registerListener(this, linearAcceleration, SensorManager.SENSOR_DELAY_NORMAL);
```

Listing 1. Przykład rejestracji odbiornika dla danych z czujnika przyspieszenia liniowego.

W zależności od modelu telefonu nie wszystkie czujniki muszą być w danym urządzeniu obecne lub mogą oferować różne parametry. W przypadku większości sensorów programista musi programowo sprawdzić czy czujnik jest obecny w urządzeniu i zadecydować co zrobić w przypadku jego braku – może da się Jednak system Android potrafi emulować działanie niektórych z nich – dla przykładu czujnik przyspieszenia liniowego może mieć dwojaką implementację: sprzętową, jako układ odejmujący wartości czujnika przyspieszenia ziemskiego od wartości akcelerometru, lub programową, jako moduł systemu operacyjnego realizujący tę samą funkcjonalność. Niezależnie od implementacji interfejs programistyczny pozostaje taki sam.

4.1.3 Filtracja sygnałów pomiarowych

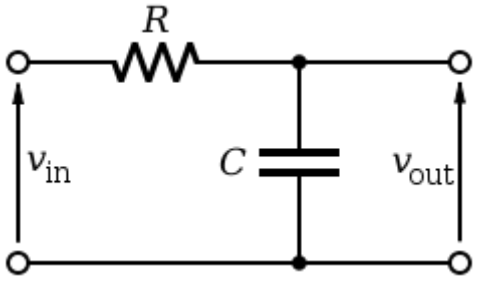
Sygnały z czujników są pełne szumów czyli niepożądanych składowych. Powodem powstawania szumu może być niska czułość urządzenia lub zakłócenia zewnętrzne. Z tego względu sygnały sensorów wymagają filtracji.

Filtracja to obróbka sygnału wejściowego w celu poprawienia jego własności. Polega ona na wykonania pewnych operacji na zbiorze próbek wejściowych sąsiadujących z bieżącą próbką oraz pewnej ilości poprzednich próbek sygnału wyjściowego. Powyższa charakterystyka dotyczy filtrów o nieskończonej odpowiedzi impulsowej, czyli filtrów NOI (ang. *Infinite Impulse Response, IIR*).

Do poprawiania własności sygnałów korzysta się w opisywanej aplikacji z dwóch filtrów, dolnoprzepustowego i uśredniającego. W przypadku badanego problemu doskonałe efekty przynosi zastosowanie dyskretnego filtru Kalmana. Niestety jest on skomplikowany matematycznie i trudny w implementacji. Dodatkowo implementacja takiego filtra wymaga urządzenia o dużej mocy obliczeniowej na co – w przypadku telefonów komórkowych – nie można sobie pozwolić.

Filtr dolnoprzepustowy

Dla danych z akcelerometru przetwarzanych przez krokomierz został wykorzystany filtr dolnoprzepustowy, którego schemat obrazuje rys. 9.



Rys. 9. Elektryczny model zastępczy filtru dolnoprzepustowego.

Poprzez dyskretyzację równania dla filtra dolnoprzepustowego otrzymuje się wzór (4).

$$y_i = \alpha x_i + (1 - \alpha) y_{i-1} \qquad (4)$$

Gdzie α zawiera się w przedziale <0;1> i nazywana jest *współczynnikiem wygładzania*, x_i to i-ta wartość na wejściu filtru, a y_i to i-ta wartość na wyjściu filtru.

Filtr uśredniający

Ten filtr został użyty dla danych z akcelerometru, kompasu i żyroskopu w komponencie szacującym azymut. W nieskończonym czasie daje on taką samą odpowiedź jak filtr dolnoprzepustowy. Działanie tego filtru polega na uśrednianiu wartości M ostatnich odczytów.

$$y_n = \frac{1}{M} \Sigma_{k=0}^{M-1} x_{n-k} \qquad (5)$$

We wzorze (5) stała M oznacza rozmiar okna pamięci. Ów wzór opisuje działanie filtru uśredniającego i, tak jak w przypadku filtru dolnoprzepustowego, jest podstawą do implementacji algorytmu filtracji.

4.1.4 Metody wykrywania i pomiaru długości kroku

Wykrywanie kroku

Istnieje wiele implementacji krokomierzy dla systemu Android o różnym stopniu skomplikowania i różnej dokładności. W systemach w wersji od 4.4 (KitKat) dostępne jest dla programisty wirtualne urządzenie - krokomierz [2]. Programista deklaruje chęć korzystania z niego tak jak w przypadku wszystkich czujników, w szczególności podając obiekt, który obsługuje dane przychodzące z krokomierza. Po wykryciu kroku (lub później, gdyż zadania w

systemie są kolejkowane) krokomierz wywołuje metodę obsługi informując aplikację, że użytkownik wykonał krok. Jest to prosta implementacja na bazie akcelerometru o dokładności zadowalającej w większości przypadków użycia. Największą wadą tego rozwiązania jest fakt, że bardzo łatwo można „oszukać" algorytm poruszając urządzeniem w górę i w dół. Niestety nie można było wykorzystać tej metody podczas realizacji niniejszej pracy inżynierskiej, gdyż dane przychodzą z opóźnieniem i nie ma możliwości żeby powiązać je z kierunkiem, w którym użytkownik się poruszał. Stąd wynikła potrzeba do stworzenia własnej implementacji krokomierza.

W pracy zaimplementowano krokomierz od podstaw w podobny sposób. Taka metoda liczenia kroków nie wymaga dużej mocy obliczeniowej i daje dobre rezultaty, jest to najpopularniejsze podejście.

Inne metody są bardziej skomplikowane i wykorzystują na przykład algorytmy uczenia maszynowego, różne wariacje filtru Kalmana czy algorytmy heurystyczne.

Długość kroku

Długość kroku zależy od wielu czynników, należą do nich m. in:

- wzrost użytkownika
- masa użytkownika
- obuwie
- podłoże
- prędkość chodu
- sposób chodu

Najprostszym rozwiązaniem byłoby ustalenie stałej długości kroku, jednak byłoby to wysoce nieefektywne.

Inne podejścia stosowane w tego typu aplikacjach (niekoniecznie dla systemu Android, ale także dla iOS czy Windows Phone) korzystają z informacji podawanych przez użytkownika na przykład o jego wzroście. Te informacje są potem podstawą działania nieskomplikowanych algorytmów estymujących długość kroku.

W tej pracy zastosowano podejście bazujące na założeniu, że im szybciej się poruszamy idąc tym większe robimy kroki. W związku z tym długość kroku rośnie liniowo ze wzrostem częstotliwości wykrywania kroków.

4.1.5 Metody estymacji kierunku

W tym przypadku naturalnym jest użycie żyroskopu, który dostarcza informacji o chwilowym przyspieszeniu kątowym wokół trzech osi. Niestety algorytmy wykorzystujące sam żyroskop dają złe rezultaty z powodu braku możliwości kompensacji dryfu. Znakomite wyniki daje tu integracja sensorów, w szczególności żyroskopu, magnetometru i akcelerometru oraz zastosowanie filtru komplementarnego. Niech:

α_k – k-ty mierzony kąt

η – współczynnik określający wpływ akcelerometru na wynik działania filtra

$\eta \epsilon < 0; 1 >$, najczęściej (także w tym przypadku) $\eta = 0{,}98$.

g – dane z żyroskopu skorygowane przy pomocy danych z kompasu

a – dane z akcelerometru

Wtedy działanie filtru komplementarnego można opisać wzorem (6)

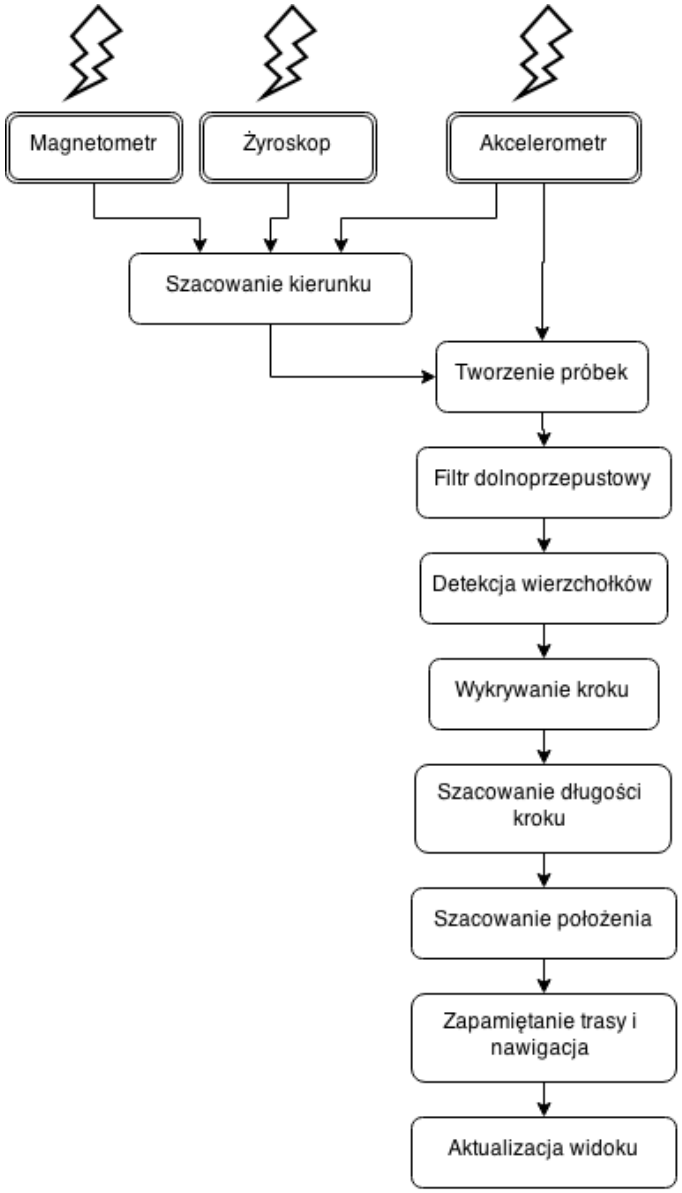
$$\alpha_k = \eta * (\alpha_{k-1} + g * dt) + (1 - \eta) * a \qquad (6)$$

5 Realizacja aplikacji

5.1 Wysokopoziomowy opis architektury aplikacji

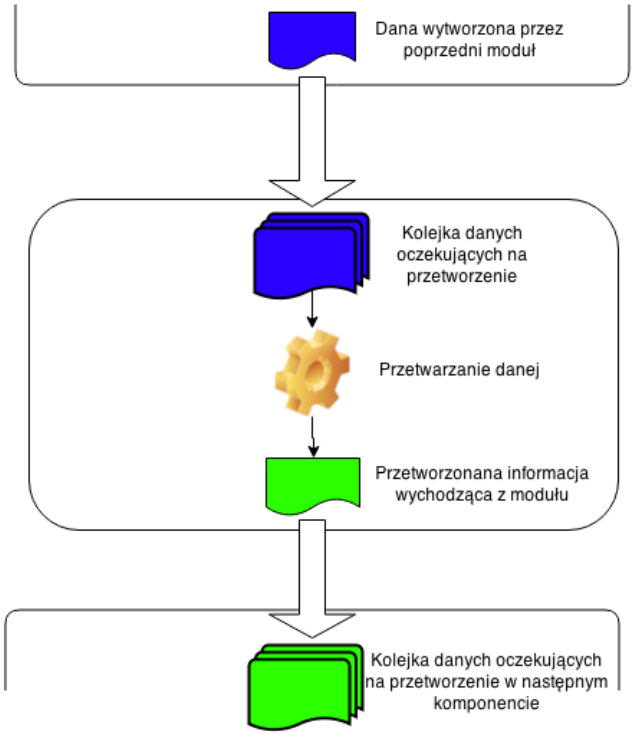
System został zaprojektowany tak, aby był przejrzysty oraz łatwy w modyfikacji, rozszerzaniu i testowaniu. Dodatkowo program potrafi dostosować się do mocy obliczeniowej urządzenia, na którym pracuje – na słabszych urządzeniach pogorszy się jakość pracy (na przykład kroki będą zliczane z mniejszą dokładnością), ale funkcjonalność będzie zapewniona.

Powyższe cechy udało się osiągnąć linearyzując ścieżkę przetwarzania danych z sensorów co pozwoliło na zaprojektowanie aplikacji jako szeregu małych modułów, obrazuje to rys. 10.



Rys. 10. Schemat architektury aplikacji

Każdy moduł deklaruje typ danych wejściowych i wyjściowych oraz definiuje sposób w jaki dane mają zostać przetworzone. Dane przychodzące wstawiane są do kolejki FIFO i oczekują na przetworzenie. Po przetworzeniu dana wyjściowa jest natychmiast przekazywana do następnego ogniwa łańcucha. Pojedynczy moduł ukazany został na rys. 11.



Rys. 11. Pojedynczy moduł

Takie podejście pozwala na rozbicie pracy pomiędzy dowolną liczbę wątków. System skonfigurowano na liczbę wątków odpowiadającą liczbie rdzeni urządzenia, na którym został uruchomiony. Dodatkowo aplikacja potrafi wykryć przepełnienie kolejki danych do przetworzenia – jest to informacja, że należy zmniejszyć ich ilość. Osiąga się to stopniowo zmniejszając częstotliwość próbkowania czujników do momentu, w którym system pracuje stabilnie. Efektem ubocznym tych działań jest spadek jakości przebiegów otrzymywanych z sensorów i dalej spadek dokładności liczenia kroków czy wyznaczania azymutu urządzenia.

Wszystkie elementy łańcucha przetwarzania znajdują się w kolejce FIFO. Wątki aplikacji (zwane pracownikami) kolejno pobierają z kolejki moduły. Każdy wątek uruchamia metodę przetwarzającą dane, a po skończeniu pracy wstawia moduł z powrotem do kolejki. Wyjątkiem

jest tu element wykrywający kierunek. Ma on najwyższy priorytet, gdyż kierunek jest wiązany z próbkami z akcelerometru na bieżąco – wymaga tego liniowy sposób przetwarzania danych.

5.2 Opis szczegółowy

5.2.1 Opis filtrów

Celem filtracji sygnałów pomiarowych w aplikacji wykorzystano dwa rodzaje filtrów: dolnoprzepustowy dla części programu wykrywającej kroki i uśredniający dla przebiegów używanych w module estymacji kierunku. Dodatkowo przy integracji sensorów użyto filtru komplementarnego.

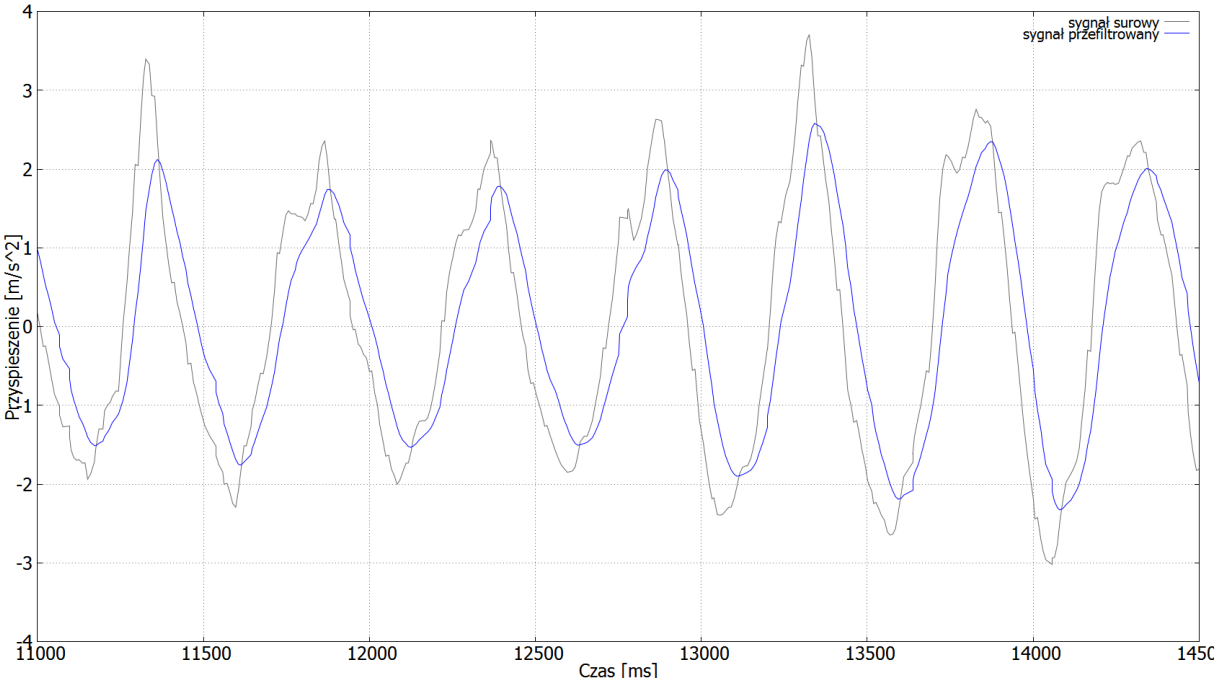
Realizacja filtru dolnoprzepustowego

W realizacji filtr dolnoprzepustowy jest programowym odzwierciedleniem wzoru (4), a najistotniejsza część jego implementacji mieści się w jednej linii.

```
double filteredValueValue = lastSample.acceleration + smoothingLevel * (newSample.acceleration -
lastSample.acceleration);
```

Listing 2. Implementacja filtru dolnoprzepustowego.

Listing 2 pokazuje jak prosto został zaimplementowany wspomniany filtr. Metodą prób i błędów sprawdzono, że najlepsze rezultaty daje stała wygładzania (współczynnik α we wzorze (4)) na poziomie 0,85.

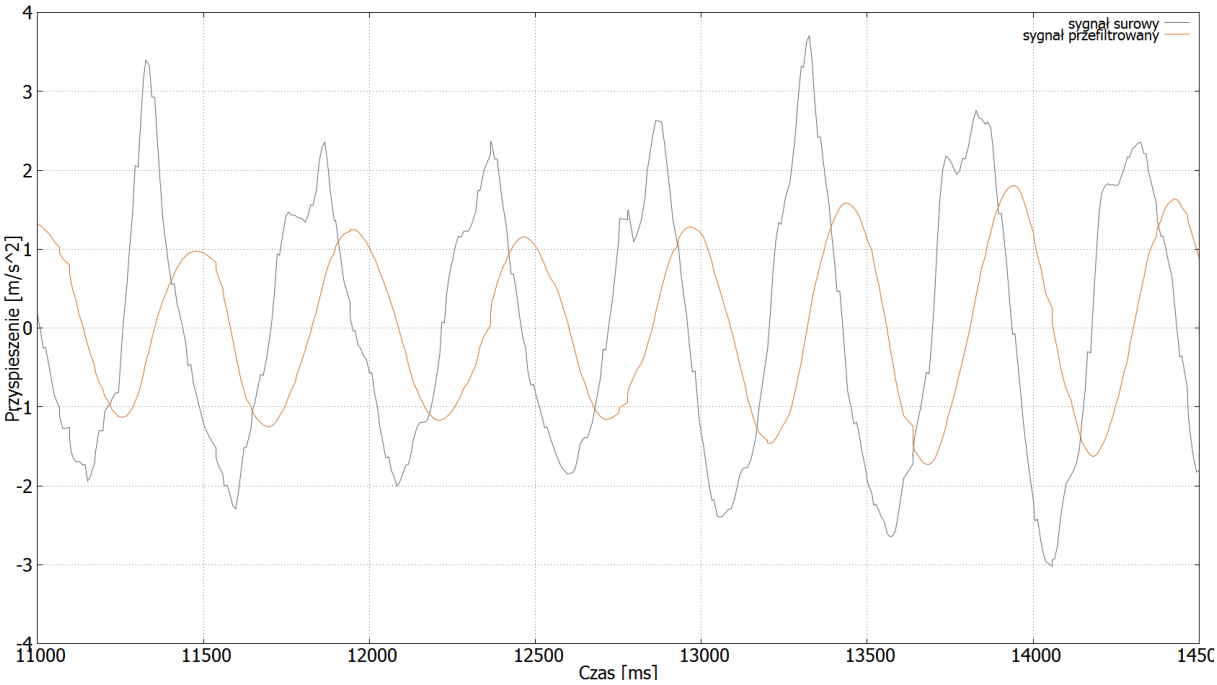


Rys. 12. Demonstracja działania filtru dolnoprzepustowego o współczynniku wygładzania równym 0,85

Rys. 12 pokazuje, że mimo prostoty, filtr dolnoprzepustowy daje zadowalające rezultaty. Dodatkowym atutem jest niska złożoność obliczeniowa, a co za tym idzie mniejsze zapotrzebowanie na zasoby oraz mniejsze zużycie energii baterii – są to cechy pożądane w przypadku urządzeń mobilnych.

Realizacja filtru uśredniającego

Przychodzące dane są wstawiane do bufora cyklicznego o pojemności 10 elementów. Za każdym razem obliczana jest średnia wartość elementów bufora i jest ona jednocześnie odpowiedzią filtru.

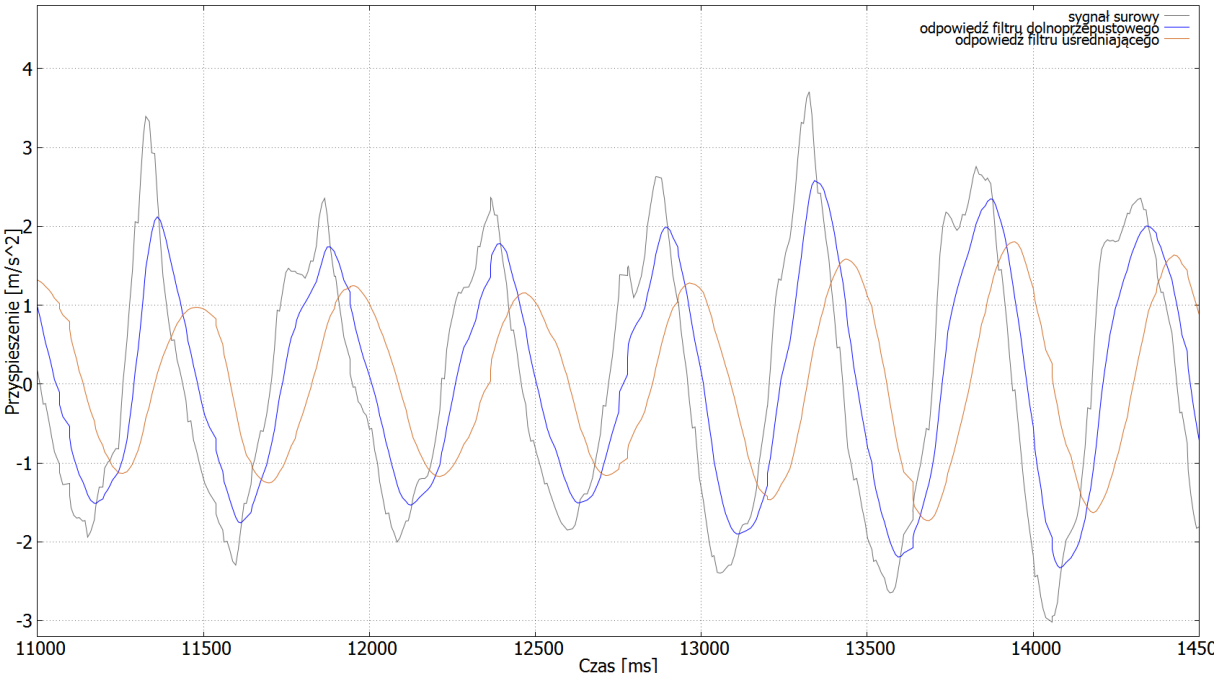


Rys. 13. Demonstracja działania filtru uśredniającego z buforem o pojemności 30 elementów.

Wielokrotne przeliczanie średniej liczb w buforze skutkuje długim czasem odpowiedzi, co pokazano na rys. 13 jako widoczne opóźnienie odpowiedzi w stosunku do surowego sygnału. Dodatkowo stwierdzono, że działanie opisywanego filtru daje dobre rezultaty.

Porównanie działania filtrów dolnoprzepustowego i uśredniającego

Przeprowadzono eksperyment, w którym dla tych samych danych wejściowych obserwowano sygnały wyjściowe z filtrów dolnoprzepustowego (o współczynniku wygładzania 0,85) oraz uśredniającego (z buforem o pojemności 30 elementów).



Rys. 14. Działanie filtrów dolnoprzepustowego i uśredniającego na tych samych danych.

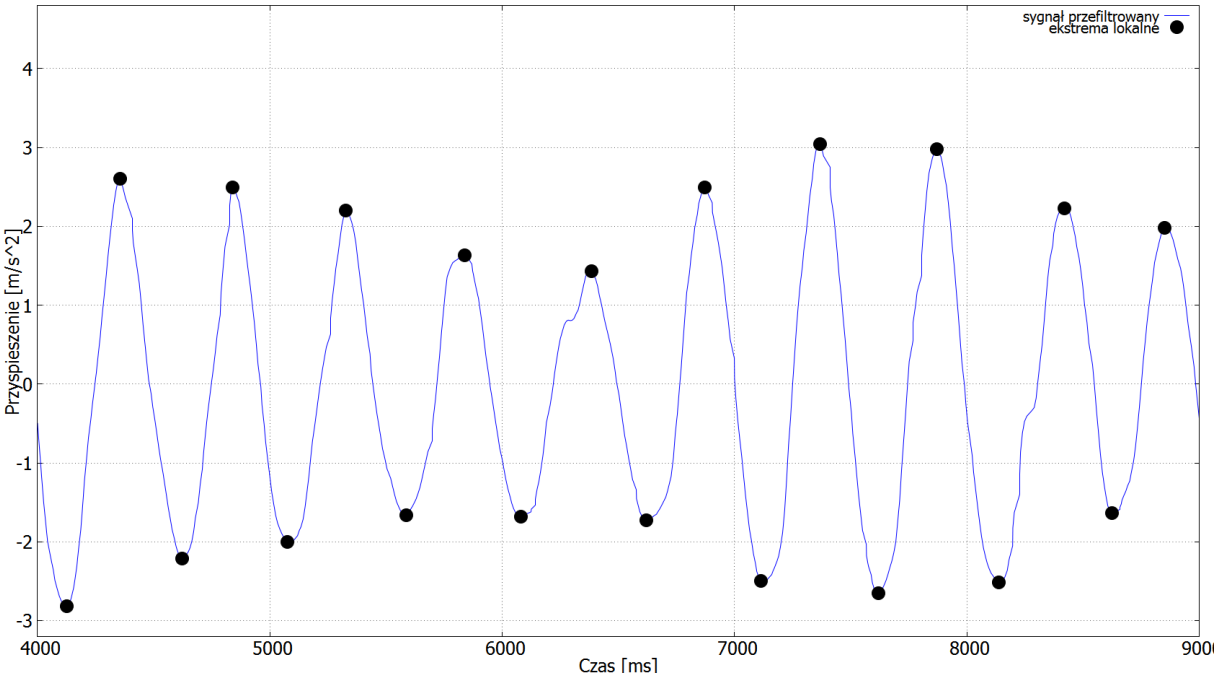
Wynik tego eksperymentu został ukazany na rys. 14. Zaobserwowano mniejsze zmiany amplitudy na wyjściu filtra uśredniającego w stosunku do dolnoprzepustowego co jest wynikiem dużej pojemności bufora, a tym samym większego wpływu poprzednich próbek na próbkę wyjściową. Zauważono także, że przebiegi sygnału z obu filtrów są bardzo podobne, jedynie różnią się opóźnieniem w stosunku do sygnału surowego. Z tego powodu przeprowadzono test, gdzie zbadano średni czas odpowiedzi filtra na próbie około tysiąca iteracji.

Średni czas odpowiedzi filtru dolnoprzepustowego wyniósł 0,1806 ms, podczas gdy średni czas działania filtru uśredniającego to 3,5474 ms. Pokazuje to, że implementacja filtru dolnoprzepustowego działa prawie 20 razy szybciej niż implementacja filtru uśredniającego, dając przy tym odpowiedź o porównywalnej jakości. Tak długi czas przetwarzania jest skutkiem cyklicznego przeliczania średniej wartości próbek w buforze. Skróceniem czasu odpowiedzi mogłoby poskutkować zmniejszenie pojemności bufora.

W związku z powyższym eksperymentem ocenia się, że wskazane jest używanie filtru dolnoprzepustowego, gdyż pracuje szybciej, zużywa mniej pamięci (nie potrzeba alokować pamięci na bufor cykliczny) a także mniej czasu procesora, czyniąc ten filtr bardziej energooszczędnym, a to wszystko przy porównywalnej jakości.

5.2.2 Opis detektora kroku

Wykrywacz kroku korzysta ze sprawdzonego a zarazem prostego algorytmu. Jego działanie opiera się na pracy wykrywacza ekstremów lokalnych przefiltrowanego sygnału. Zlicza on pary minimum – maksimum, lub maksimum – minimum, dlatego kluczowe było tu użycie filtru przed analizą sygnału.



Rys. 15. Demonstracja działania wykrywacza ekstremów lokalnych.

Czarne punkty na rys. 15 symbolizują znalezione ekstrema lokalne, które są danymi wejściowymi dla modułu wykrywającego krok. Każda para kolejnych wierzchołków stanowi krok dla opisywanego komponentu zgodnie z algorytmem pokazanym na listingu 3.

```
public boolean makesStepWith(Peak another) {
    return     (this.isLocalMaximum() ^ another.isLocalMaximum())
            && Math.signum(this.acceleration) != Math.signum(another.acceleration)
            ;
}
```

Listing 3. Metoda stwierdzająca czy dwa wierzchołki tworzą krok.

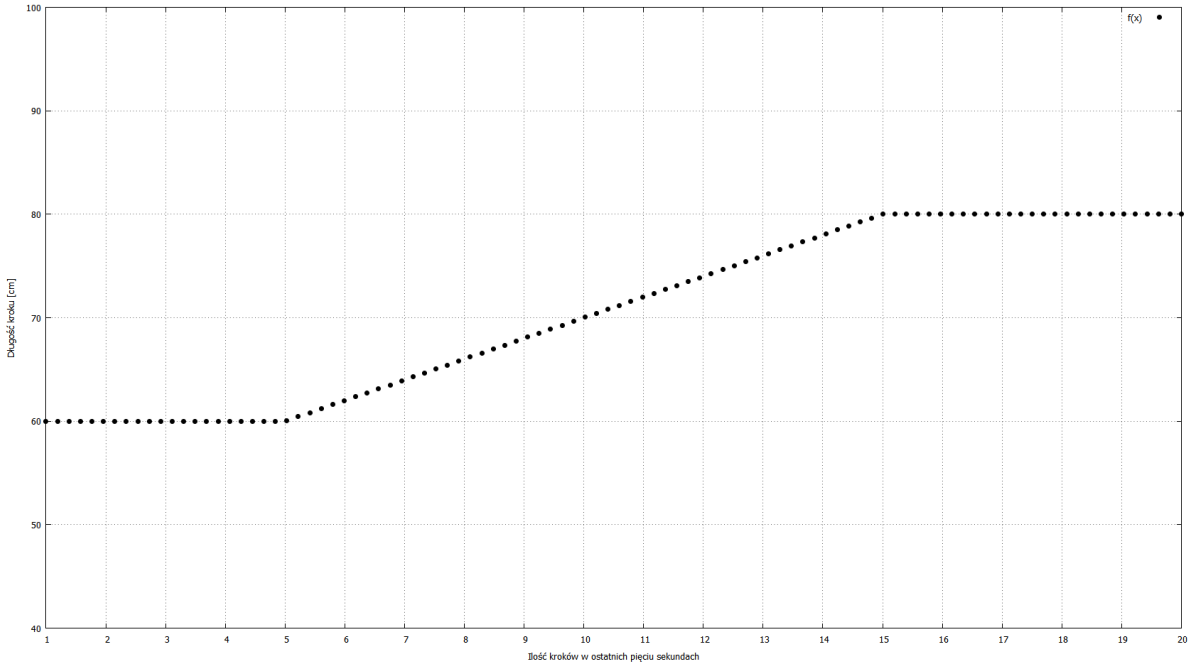
Listing 3 pokazuje, że jest to nieskomplikowany sposób wykrywania kroku. Do zalet tego rozwiązania należą: brak skomplikowania, niskie zapotrzebowanie na zasoby i energię. Zdecydowaną wadą jest podatność na pomyłki – kroki mogą być zliczane kiedy użytkownik potrząsa telefonem. Detektor kroku jako dane wejściowe przyjmuje ostatnio wykryte na przefiltrowanych przebiegach akcelerometru wierzchołki, na wyjściu pojawia się informacja czy wykonano krok i, jeśli tak, to w jakim kierunku.

5.2.3 Opis komponentu mierzącego długość kroku

Praca modułu mierzącego długość kroku polega na mierzeniu częstotliwości stawiania kroków. Na tej podstawie szacowana jest długość kolejnych kroków. Szacuje się, że średnia długość kroku człowieka waha się od 60 cm (u kobiet) do 80 cm (u mężczyzn). Na tej podstawie ustalono średnią na poziomie 70 cm. Szacuje się, że człowiek idący standardowym tempem porusza się z prędkością 5 km/h. Powyższe dane zostały wykorzystane do wyprowadzenia wzoru na długość kroku w zależności od ilości kroków postawionych w ciągu pięciu sekund.

Komponent został zaprojektowany tak, aby zliczać kroki użytkownika w trakcie ostatnich pięciu sekund działania aplikacji. Na bazie tej liczby obliczana jest długość kroku zgodnie ze wzorem (7), a zależność obrazuje rys. 16.

$$f(x) = \begin{cases} 60, dla\ x < 5 \\ 60 + \frac{x-5}{10} \cdot 20, dla\ x\ \epsilon < 5; 15) \\ 80, dla\ x \geq 15 \end{cases} \qquad (7)$$



Rys. 16. Wykres funkcji f(x) ze wzoru (7).

5.2.4 Opis komponentu wykrywającego zmianę kierunku

Moduł estymacji azymutu nasłuchuje na dane z czterech czujników jednocześnie, na tę liczbę składają się: akcelerometr, czujnik grawitacji, magnetometr i żyroskop. Opisywany

moduł został zapożyczony z aplikacji Gyroscope Explorer i zaadaptowany tak, aby pasował do wytwarzanego programu.

Obsługa kompasu

Kompas dostarcza informacji o składowych siły pola magnetycznego działającego na osie układu współrzędnych urządzenia. Są one filtrowane przez filtr uśredniający i zapisywane jako aktualne (listing 4).

```
public void onMagneticSensorChanged(float[] magnetic, long timeStamp)
{
    // Get a local copy of the raw magnetic values from the device sensor.
    System.arraycopy(magnetic, 0, this.magnetic, 0, magnetic.length);
    this.magnetic = meanFilterMagnetic.filterFloat(this.magnetic);
}
```

Listing 4. Algorytm reakcji na dane z kompasu.

Obsługa czujnika grawitacji i akcelerometru

Dane z czujnika grawitacji opisują działanie składowych siły grawitacji na każdą z osi lokalnego układu współrzędnych urządzenia. Nowe dane są filtrowane przy użyciu filtru uśredniającego i uaktualniane, następnie zostaje wywołana metoda przeliczająca na nowo orientację urządzenia (listing 7). Analogicznie zrealizowano obsługę danych z akcelerometru. Listing 5 obrazuje powyższe zachowanie.

```
public void onGravitySensorChanged(float[] gravity, long timeStamp)
{
    // Get a local copy of the raw magnetic values from the device sensor.
    System.arraycopy(gravity, 0, this.gravity, 0, gravity.length);
    this.gravity = meanFilterAcceleration.filterFloat(this.gravity);
    calculateOrientation();
}
```

Listing 5. Algorytm reakcji na dane z czujnika grawitacji.

Jak pokazano na listingu 6. przeliczanie orientacji polega na użyciu dostarczanej przez system funkcji *getRotationMatrix()* (zwracającej macierz obrotu urządzenia względem układu współrzędnych świata obliczaną na podstawie informacji o grawitacji i sile pola magnetycznego działającego na urządzenie), a następnie skorzystaniu z metody *getOrientation()*, która na podstawie macierzy obrotu określa obrót urządzenia względem każdej jego osi.

```
private void calculateOrientation()
{
    if (SensorManager.getRotationMatrix(rotationMatrix, null, gravity,
            magnetic))
    {
        SensorManager.getOrientation(rotationMatrix, orientation);
    }
}
```

Listing 6. Algorytm przeliczania orientacji.

Obsługa żyroskopu

Dane z żyroskopu po normalizacji trafiają do filtru komplementarnego (por. 3.1.5) razem z danymi z pozostałych sensorów co skutkuje aktualizacją informacji o azymucie, stąd dane przekazywane są do odbiorców tego sensora.

5.2.5 Opis komponentu nawigacji

Komponent nawigacji w każdym momencie pamięta lokalizację urządzenia. Co każdy wykryty krok użytkownika aktualizuje położenie o nowe przemieszczenie. Moduł działa w dwóch trybach. Początkowo jego praca polega na zapamiętywaniu lokacji po każdym kroku. Każda zapamiętana lokacja dodawana jest do listy będącej odzwierciedleniem ścieżki poruszania się użytkownika.

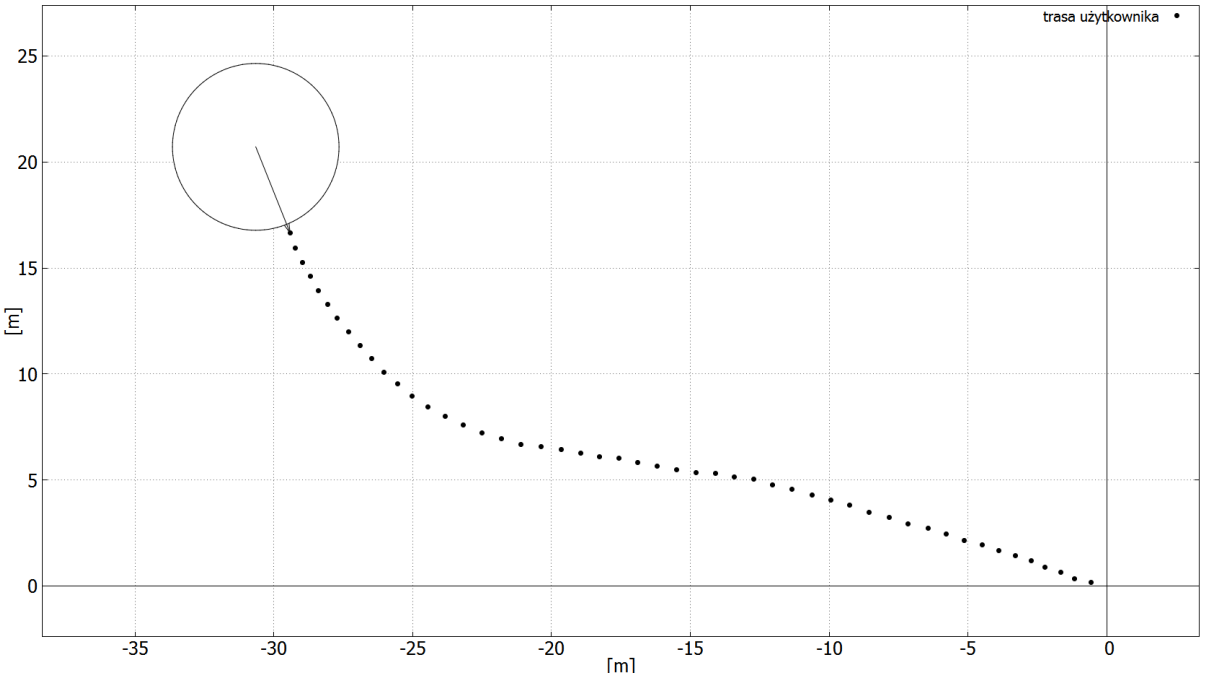
W drugim trybie część nawigująca znając aktualne położenie urządzenia oraz ścieżkę poruszania stara się poprowadzić użytkownika wzdłuż tej ścieżki. Moduł przestaje wtedy budować ścieżkę powrotną. Obliczany jest wektor prowadzący z aktualnej lokacji do ostatniej zapamiętanej lokacji i na tej podstawie wyświetlane są na ekranie podpowiedzi dotyczące kierunku poruszania się. Jeśli urządzenie znajdzie się w okolicy (ok. 2,5 m) punktu, do którego nawiguje opisywany komponent, wspomniany punkt zostaje usunięty z listy. Algorytm kończy pracę w momencie, w którym na liście nie ma już punktów, do których można kierować użytkownika.

```
if (!isBacktracking()) {
    path.add(location);
    return null;
} else {
    return path.size() > 0 ? getDirectionFromTo(location, path.getLast()) : null;
}
```

Listing 7. Zapis algorytmu nawigacji

Istota działania algorytmu została zawarta na listingu 7. Komponent przyjmuje na wejściu aktualną lokację urządzenia, a danymi wyjściowymi są wskazówki, w którym kierunku użytkownik powinien iść.

Działanie komponentu obrazuje rys. 17. Użytkownik znajduje się w środku okręgu o promieniu 2,5m, a podpowiadany jest kierunek do najbliższego zapamiętanego punktu. Jeśli punkt, do którego nawigowany jest użytkownik znajdzie się w okręgu zostaje usunięty z listy.



Rys. 17. Demonstracja sposobu działania komponentu nawigacji.

6 Testy

6.1 Specyfikacja urządzenia testowego

Do testów został wykorzystany telefon komórkowy LG G2.

Tab. 2. Specyfikacja czujników w urządzeniu testowym.

Typ czujnika	**Producent**	**Pobór mocy**	**Rozdzielczość**	**Minimalny okres próbkowania**
Akcelerometr	STMicroelectronics	0,280 mA	0,001 m/s^2	8333 µs
Czujnik przyspieszenia liniowego	Qualcomm	6,380 mA	0,001 m/s^2	8333 µs
Czujnik grawitacji	Qualcomm	6,380 mA	0,001 m/s^2	8333 µs
Żyroskop	STMicroelectronics	6,100 mA	0,001 rad/s	5000 µs
Magnetometr	AKM	5,000 mA	0,15 µT	16666 µs

W tab. 2. podano specyfikację czujników telefonu testowego. Warty odnotowania jest fakt, że czujniki przyspieszenia liniowego i grawitacji zostały zrealizowane w urządzeniu w postaci układów scalonych.

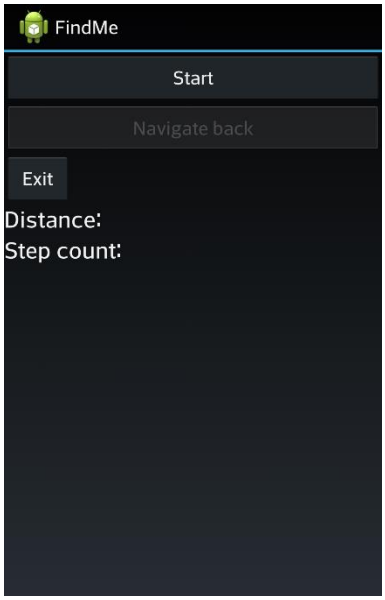
Urządzenie testowe jest bardzo wydajne obliczeniowo. Działa on pod kontrolą czterordzeniowego procesora Qualcomm Snapdragon 800 z zegarem o taktowaniu 2,26 GHz. Przetwarzaniem grafiki zajmuje się wydajny układ Adreno 330. Telefon posiada pamięć RAM o pojemności 2 GB.

6.2 Testy i omówienie wyników

W ramach testów funkcjonalnych rozpatrzono wiele przypadków uwzględniając różne trasy i prędkości chodu. Dodatkowo zostały przeprowadzone testy zużycia zasobów, w szczególności pamięci, czasu przetwarzania dla każdego modułu, poboru mocy. Sprawdzono też wielkość obszaru zajmowanego w przez aplikację w pamięci trwałej.

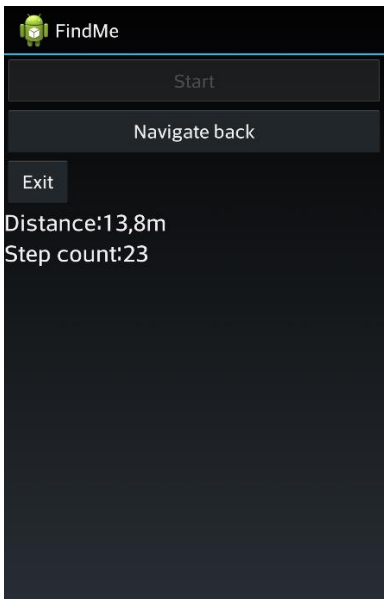
6.2.1 Wygląd aplikacji

Wykonano zrzuty ekranów aplikacji w różnych stanach obrazując funkcjonalność programu.



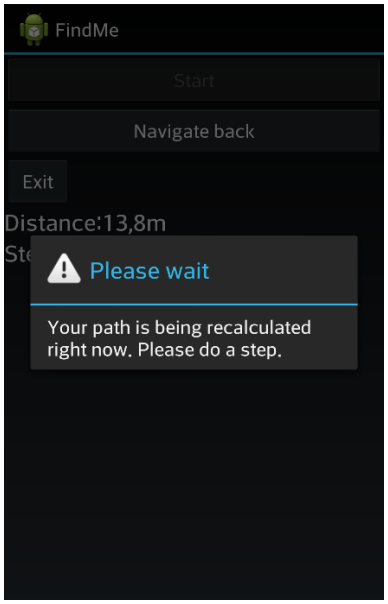
Rys. 18. Ekran startowy aplikacji.

Po uruchomieniu programu użytkownik może rozpocząć zapamiętywanie trasy lub wyjść z programu (rys. 18).



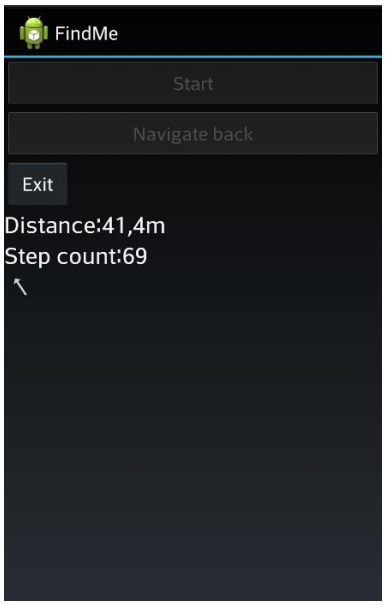
Rys. 19. Ekran aplikacji w trybie zapamiętywania trasy.

Po kliknięciu przycisku Start program rozpoczyna śledzenie trasy użytkownika (rys. 19). Po każdym kroku wyświetlane są informacje o szacowanej długości przebytej trasy oraz ilości kroków użytkownika. Stąd użytkownik może zakończyć działanie programu lub przejść do trybu nawigacji.



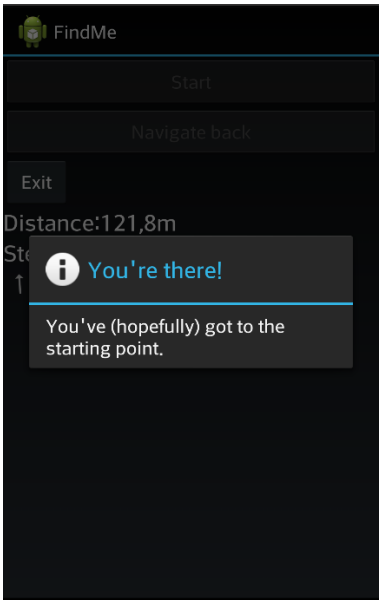
Rys. 20. Ekran aplikacji w momencie przejścia w tryb nawigacji.

W trakcie przejścia w tryb nawigacji algorytm przetwarza dane z kolejek modułów aby skończyć budowanie trasy powrotnej w tym czasie wyświetlając stosowny komunikat (rys. 20). Po zrobieniu kroku przez użytkownika przechodzi w tryb nawigacji.



Rys. 21. Ekran aplikacji w trybie nawigacji.

Po zniknięciu komunikatu o przeliczaniu trasy zostaje wyświetlona strzałka informująca użytkownika o kierunku, w którym powinien iść (rys. 21). Kierunek wskazywany przez strzałkę jest aktualizowany po każdym kroku.



Rys. 22. Ekran aplikacji po zakończeniu pracy.

Kiedy użytkownik zjawi się w okolicy (poniżej 2,5m) celu zostaje wyświetlony stosowny komunikat (rys. 22). Algorytm kończy działanie, a użytkownik może jedynie wyjść z aplikacji.

Interfejs graficzny został zaprojektowany tak, aby był intuicyjny dla użytkownika i jednocześnie jak najprostszy pod względem wykonania.

6.2.2 Testy zużycia zasobów

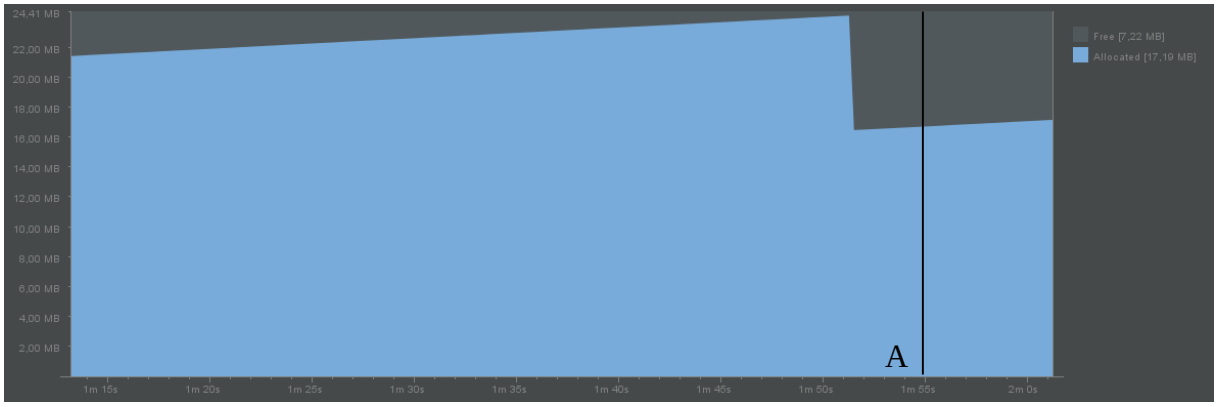
Zajętość pamięci trwałej urządzenia

Zmierzono, że zainstalowana aplikacja zajmuje 116 KB. Jest to dobry wynik wziąwszy pod uwagę, że aplikacje na system Android zajmują od kilkuset kilobajtów do kilkuset megabajtów, a w przypadku gier – nawet kilku gigabajtów. Tak mała objętość jest prawdopodobnie wynikiem braku elementów graficznych w programie.

Zużycie pamięci RAM

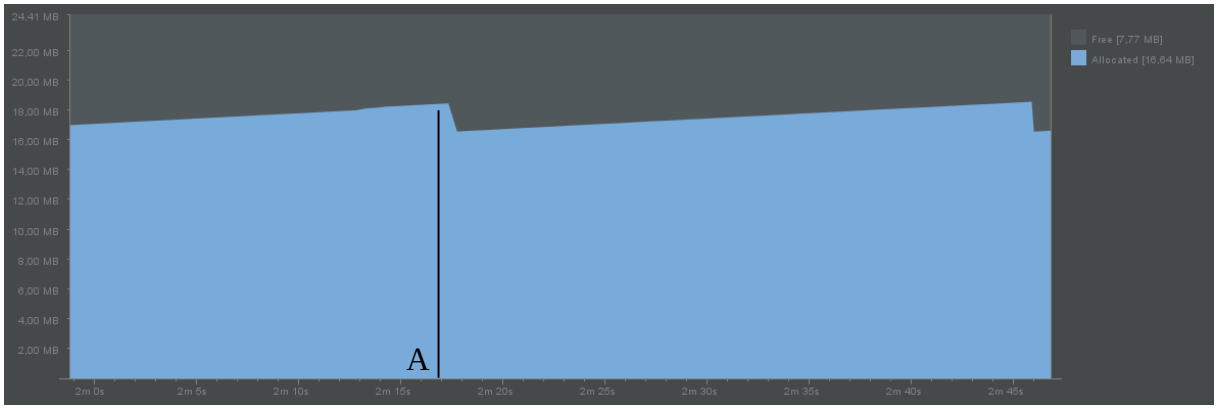
W celu przetestowania zużycia pamięci krótkotrwałej posłużono się narzędziem monitora pamięci (ang. *Memory Monitor*) zintegrowanego środowiska programistycznego IntelliJ Idea (por. 2.1.3).

W ramach testu używano aplikacji przez około 3 minuty wprawiając urządzenie w intensywne drgania harmoniczne w osi Z urządzenia (pion), tym samym imitując ruch podczas chodu. Zaobserwowano, że w momencie uruchomienia, program zużywa ok 16 MB pamięci.



Rys. 23. Wykres zużycia pamięci przez aplikację w trybie zapamiętywania trasy.

Rys. 23. Pokazuje liniowy wzrost zajętości pamięci w czasie, w tempie ok. 68,27 KB/s. Ma to związek z alokowaniem pamięci w kolejkach danych oczekujących na przetworzenie w każdym module oraz, w trybie śledzenia trasy, zapamiętywania lokacji użytkownika po każdym kroku. Po około 1 minucie i 51 sekundach (punkt A) pracy zaobserwowano odśmiecanie pamięci (ang. *Garbage collection*) – proces zwalniania pamięci zaalokowanej dla nieużywanych już obiektów. Jest to część maszyny wirtualnej Javy (ang. *Java virtual machine, JVM*), działa samoczynnie, cyklicznie i ma za zadanie odciążyć programistę. Dzięki temu osoba tworząca aplikację nie musi pamiętać o zarządzaniu pamięcią. Odśmiecanie pamięci wizualizuje wyraźny skok na wykresie.

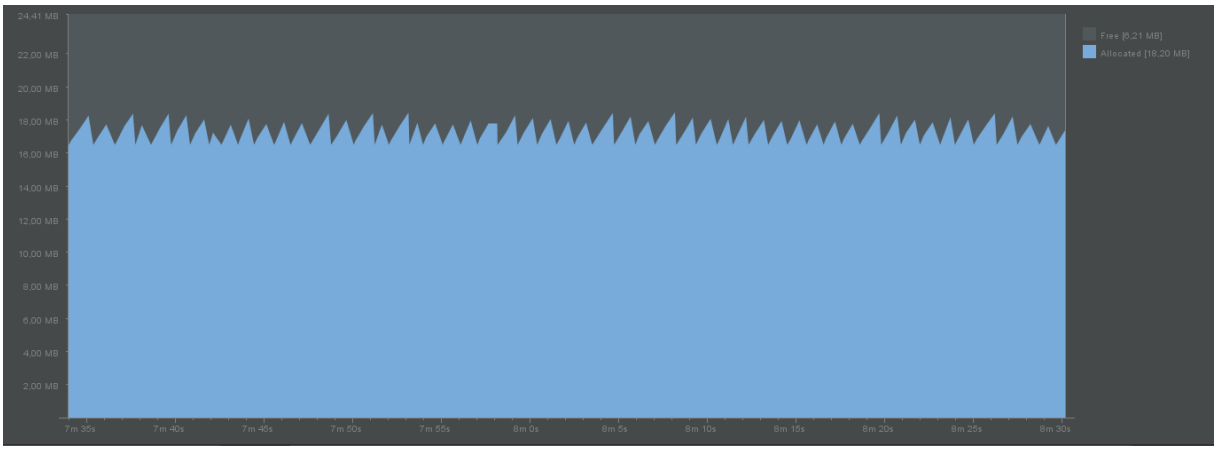


Rys. 24. Wykres zużycia pamięci przez aplikację w trybie nawigacji.

W trakcie eksperymentu zmieniono tryb pracy programu na nawigację do punktu startu i dalej obserwowano zużycie pamięci. Rys. 24. przedstawia tę sytuację. Około dwóch minut i trzynastu sekund po starcie aplikacji (punkt A) przełączono jej tryb pracy na nawigację powrotną. Widać w tym momencie niewielki skok związany z alokacją pamięci na obiekty pomocnicze, na przykład okienko informacyjne. Nastąpił także wzrost tempa alokacji pamięci do poziomu około 78,77 KB/s. Ma to związek z tworzeniem nowych obiektów do aktualizacji

widoku i wskazania kierunku użytkownikowi. Jednakże po odśmiecaniu pamięć dalej wracała do poziomu około 16 MB.

Należy zaznaczyć, że wykresy zajętości pamięci przedstawione na rys. 23 i 24 zostały zarejestrowane w trybie debugowania, gdzie życie obiektów jest celowo przedłużane celem umożliwienia wglądu do nich, a odśmiecanie wykonywane znacznie rzadziej niż w normalnych warunkach. Zostało to zobrazowane na Rys. 25. Wynika z niego, że program nie alokuje więcej niż 18 MB pamięci naraz.



Rys. 25. Wykres zajętości pamięci w normalnym trybie pracy.

W świetle powyższego eksperymentu ocenia się, że zużycie pamięci przez opisywany program mieści się w akceptowalnych granicach. Dużą oszczędnością mogłoby okazać się ograniczenie pojemności kolejek w modułach lub zrezygnowanie z nich w ogóle – rezygnując tym samym ze współbieżności.

Czas przetwarzania

Zaimplementowano miernik czasu w klasie bazowej wszystkich modułów. Na koniec wykonania program zebrał dane ze wszystkich komponentów i wypisał je na wyjście połączonego z urządzeniem środowiska programistycznego.

Tab. 3. Średnie czasy przetwarzania danych przez kolejne moduły aplikacji.

Zadanie	Czas przetwarzania [ms]
Filtr dolnoprzepustowy	0,181
Wykrywanie ekstremów lokalnych	0,655
Filtrowanie ekstremów lokalnych	0,511
Wykrywanie kroków	0,573
Liczenie ilości kroków	0,670
Estymacja długości kroku	0,709
Liczenie przebytego dystansu	0,736
Szacowanie położenia	0,286
Śledzenie trasy / nawigacja	0,659
Wyznaczanie kierunku do celu	0,524

Jak pokazano w tab. 3. Poszczególne moduły pracują bardzo szybko, przetwarzanie porcji danych nie zajęło żadnemu z nich więcej niż 1 ms. Dodatkowo maksymalny czas życia próbki, w trakcie którego przechodzi pełną ścieżkę łańcucha przetwarzania wynosi 5,5 ms. Oznacza to, że maksymalna częstość próbkowania akcelerometru, który generuje dane wejściowe dla łańcucha mogłaby wynosić nawet 181 Hz.

Wydajność aplikacja jest więcej niż zadowalająca i powinna być zupełnie wystarczająca do dokładnego wykrycia kroku i oszacowania kierunku – dzięki krótkiemu okresowi próbkowania przebiegi sygnałów czujników są dokładniejsze.

Pobór mocy

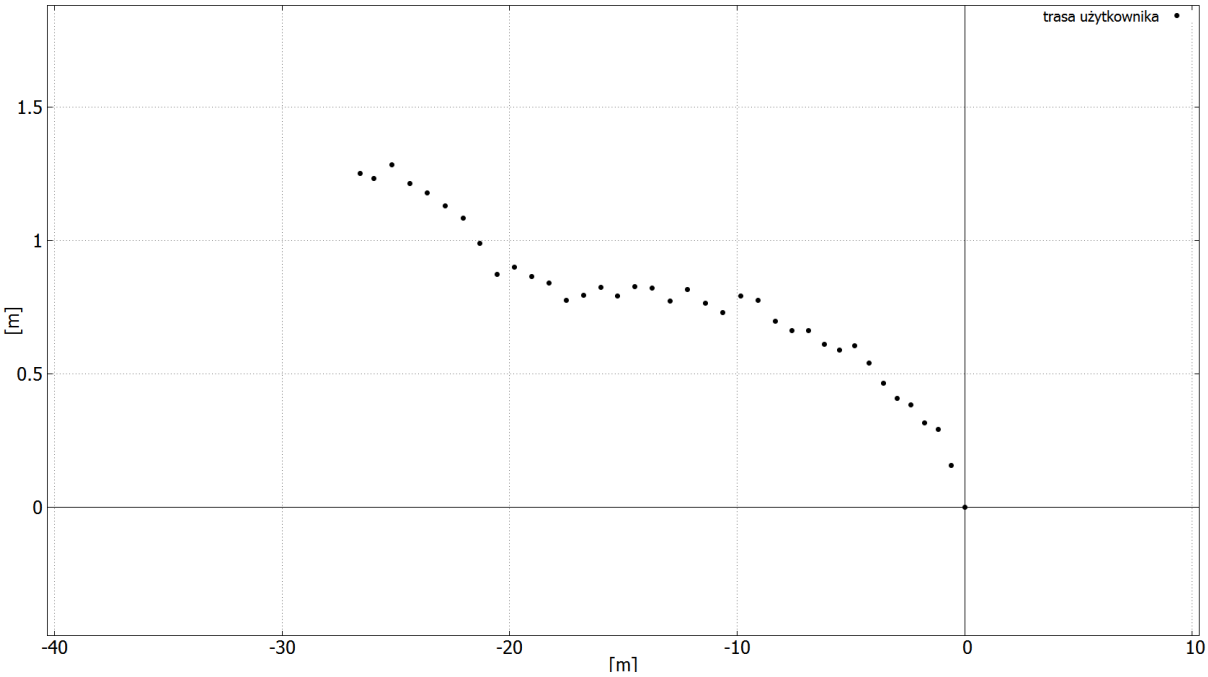
Przy pomocy aplikacji Power Tutor zmierzono pobór mocy - kształtuje się on na poziomie 5,8 mW/s. Jest to bardzo wysoki wynik w porównaniu do innych aplikacji. Tak wysoki wynik został spowodowany wieloma czynnikami, w szczególności wymuszeniem niewygaszania ekranu podczas działania aplikacji oraz wysoką częstotliwością próbkowania czujników (akcelerometr, czujnik przyspieszenia liniowego, żyroskop, magnetometr) co znacząco wpływa na użycie procesora.

6.2.3 Testy funkcjonalne

Przeprowadzono serię eksperymentów mających na celu sprawdzenie jakości pracy komponentów systemu, jak również działania aplikacji jako całości.

Porównanie długości trasy przebytej przez użytkownika ze wskazaniami aplikacji

Test polegał na przejściu 30 kroków ze stałą prędkością około 5 km/h w linii prostej i obserwacji zachowania programu. Celem eksperymentu było głównie sprawdzenie działania krokomierza i estymatora długości kroku, ale także obserwacja dryfu estymatora azymutu.



Rys. 26. Kolejne lokacje użytkownika w teście długości trasy.

W trakcie eksperymentu użytkownik szedł w linii prostej, jednak trasa widoczna na rys. 26 jest widocznie zakrzywiona, punkty nie układają się na prostej. Przewiduje się, że jest to spowodowane wadami komponentu wyznaczającego azymut. Niewielkie zakrzywienie trasy spowodowane jest dryfem (nadal znacznie mniejszym niż dryf samego żyroskopu).

Tab. 4. Porównanie wartości rzeczywistych ze wskazaniami aplikacji.

	Rzeczywista wartość	**Wskazanie aplikacji**	**Wielkość błędu**
Ilość kroków	30	37	23%
Przebyty dystans	Około 24,0 m	26,7 m	11%

W tab. 4 porównano dystans przebyty przez użytkownika z wartościami wyświetlanymi przez aplikację. Jak wykazał eksperyment krokomierz zawyża wartości o 23%. Do przyczyn zalicza się wstrząsy biorące się z nierównego podłoża (test wykonano na otwartej przestrzeni), na które podatny jest prosty algorytm zliczania kroków. Błędy na tym poziomie były przewidywalne, biorąc pod uwagę nieskomplikowanie krokomierza. Zadowalający efekt przyniosło zaś szacowanie odległości przebytej przez użytkownika.

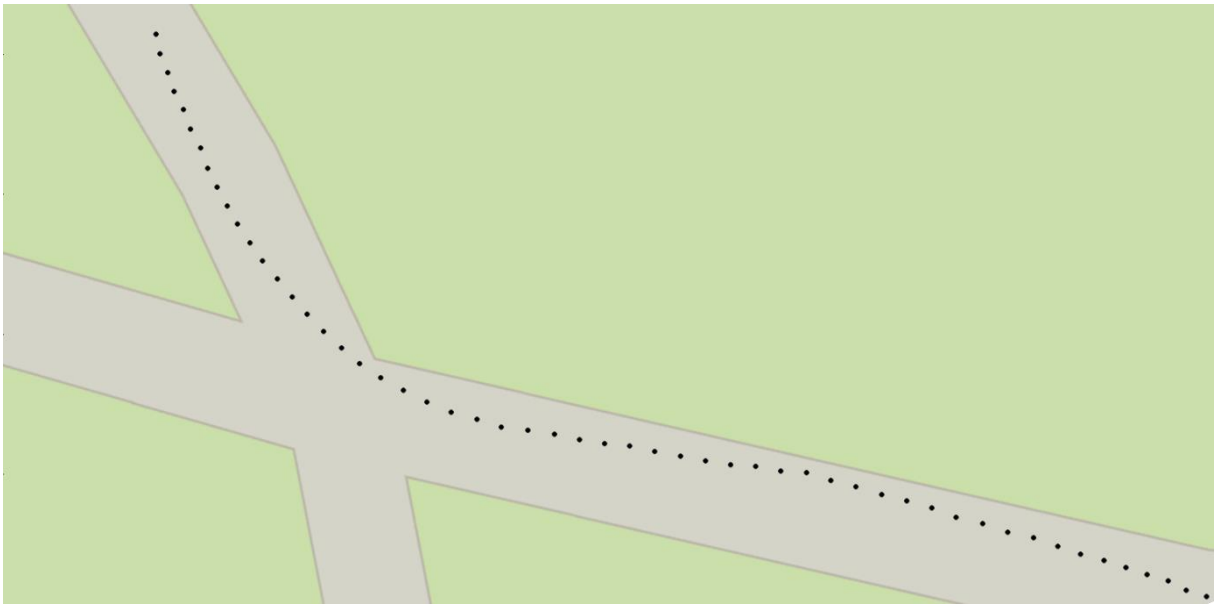
Test powtórzono dla odcinka o długości 16 m, tym razem na płaskiej powierzchni (korytarz). Użytkownik przebył trasę trzy razy, za każdym razem innym tempem. Wyniki przedstawia tab. 5. Zaobserwowano bardzo dokładne szacowanie dla tempa szybkiego i średniego (ok 6 i 5 km/h), gdzie błąd szacowania wynosił 6 i 8 cm. Stwierdzono także zadowalający efekt działania krokomierza. Niestety aplikacja okazała się być niedokładna przy tempie wolnym – 4 km/h. W wyniku dużego przeszacowania ilości kroków podany dystans był znacznie zawyżony.

Tab. 5. Porównanie wskazań aplikacji z rzeczywistością dla różnego tempa chodu.

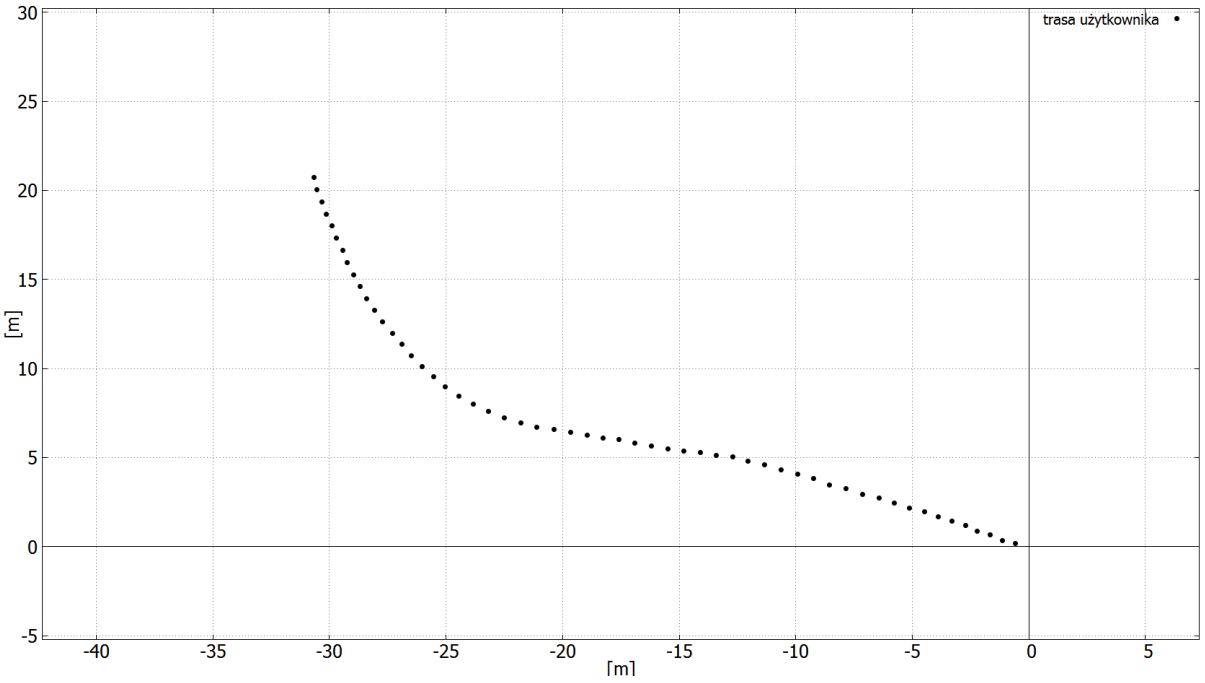
Tempo	Czas [s]	Prędkość [km/h]	Rzeczywista ilość kroków	Wykryta ilość kroków	Zarejestrowany dystans [m]
Szybkie	9,3	6,19	20	23	15,94
Średnie	11,8	4,88	22	24	16,08
Wolne	14,4	4,00	24	35	24,90

Test komponentu szacującego azymut dla łagodnych zakrętów

Następnym eksperymentem było przetestowanie jakości pomiarów kierunku poruszania się użytkownika na dłuższej trasie z łagodnym zakrętem na otwartej przestrzeni. Trasę zarejestrowaną przez urządzenie nałożono na mapę miejsca, w którym wykonano test (rys. 27). Użytkownik przebył dystans około 35 m



Rys. 27. Trasa zarejestrowana przez urządzenie nałożona na mapę.



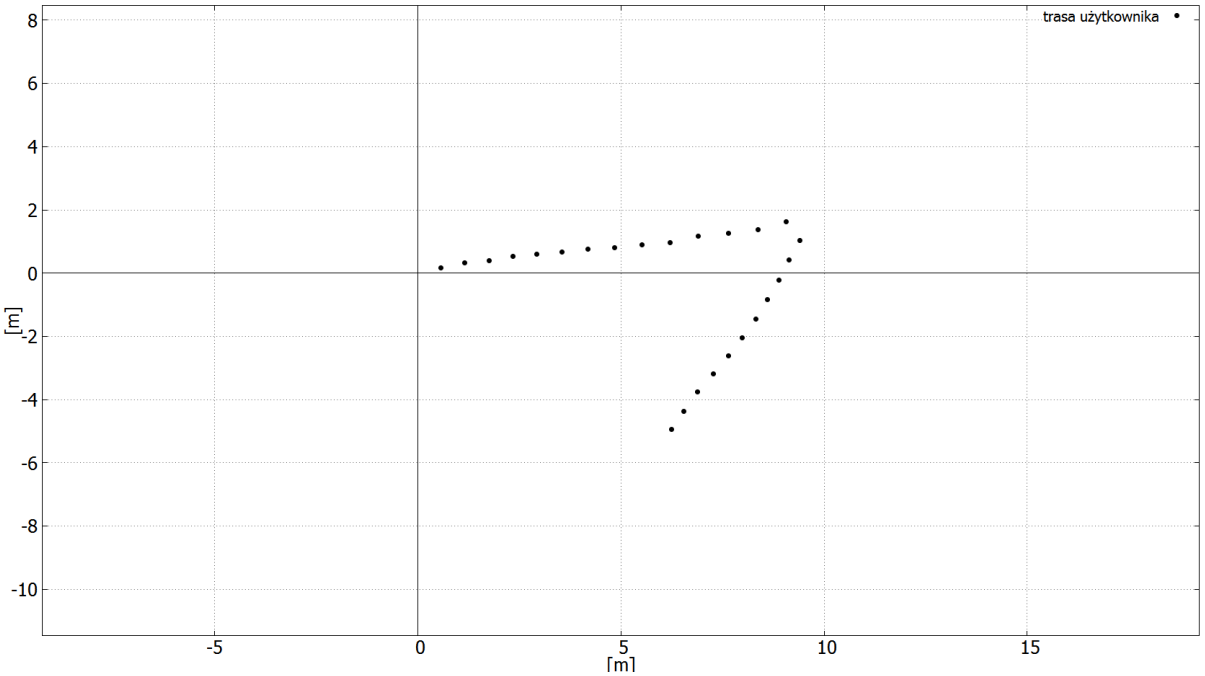
Rys. 28. Trasa użytkownika na łagodnym zakręcie.

Przebieg trasy zarejestrowanej przez urządzenie obrazuje rys. 28. Zaobserwowano łagodny łuk co jest zgodne z oczekiwaniami. Przekłamanie, prawdopodobnie spowodowane dryfem, nastąpiło około punktu (-11, 5).

Aplikacja zarejestrowała 40,06 m (błąd na poziomie 14%). Wynik testu uznano za zadowalający, aplikacja utrzymała trasę w sensownych granicach.

Test komponentu szacującego azymut dla zakrętu o kącie 90 stopni

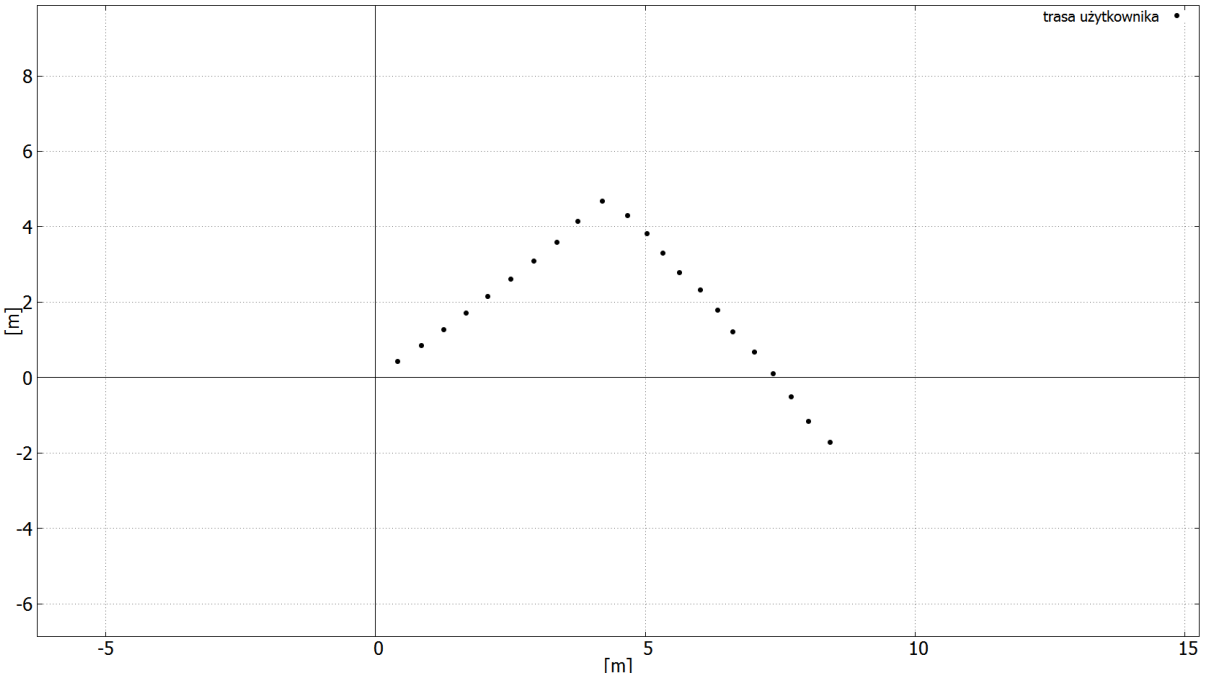
Rozpatrzono przypadek, w którym użytkownik po przebyciu kilku (ok. 8) metrów zatrzymuje się, obraca o 90 stopni i pokonuje kolejne kilka metrów w linii prostej.



Rys. 29. Trasa zarejestrowana przez urządzenie podczas eksperymentu z zakrętem pod kątem prostym.

Opisany eksperyment nie powiódł się, z racji dużego przekłamania, przy szybkim zakręcie. Trasa zarejestrowana przez aplikację jest widoczna na rys. 29. Oczekiwanym wynikiem była zmiana kąta poruszania się o 90 stopni, jednak system zarejestrował zmianę rzędu 135 stopni (błąd 50%). Jak pokazała powyższa próba dynamiczna odpowiedź żyroskopu ma zbyt duży wpływ na estymację kąta.

Test powtórzono zmieniając jednak parametr η filtru komplementarnego z 0.5 na 0.2 (por. 3.1.5).

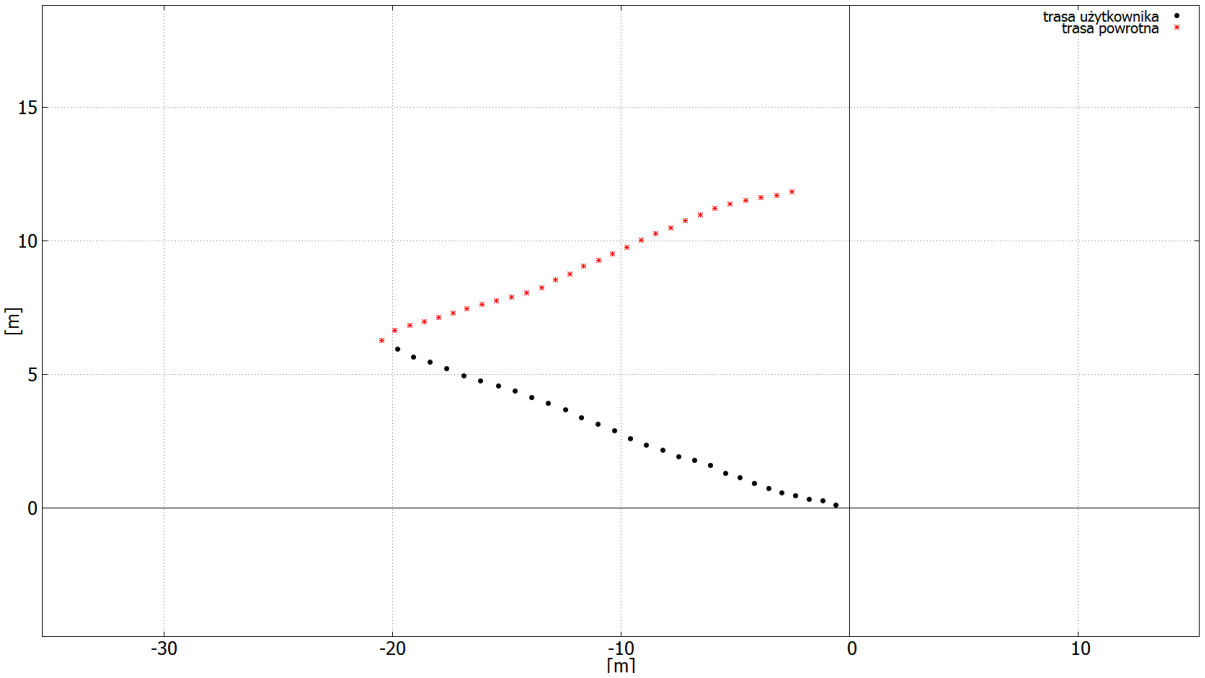


Rys. 30. Trasa użytkownika z ostrym zakrętem ze zmniejszonym wpływem żyroskopu.

Wynik testu przedstawia rys. 30. Zmniejszenie wpływu żyroskopu przyniosło znakomite rezultaty, nie tylko w dynamicznej estymacji kąta, ale także znacznie zmniejszyło dryf komponentu estymującego azymut.

Test nawigacji na prostej trasie

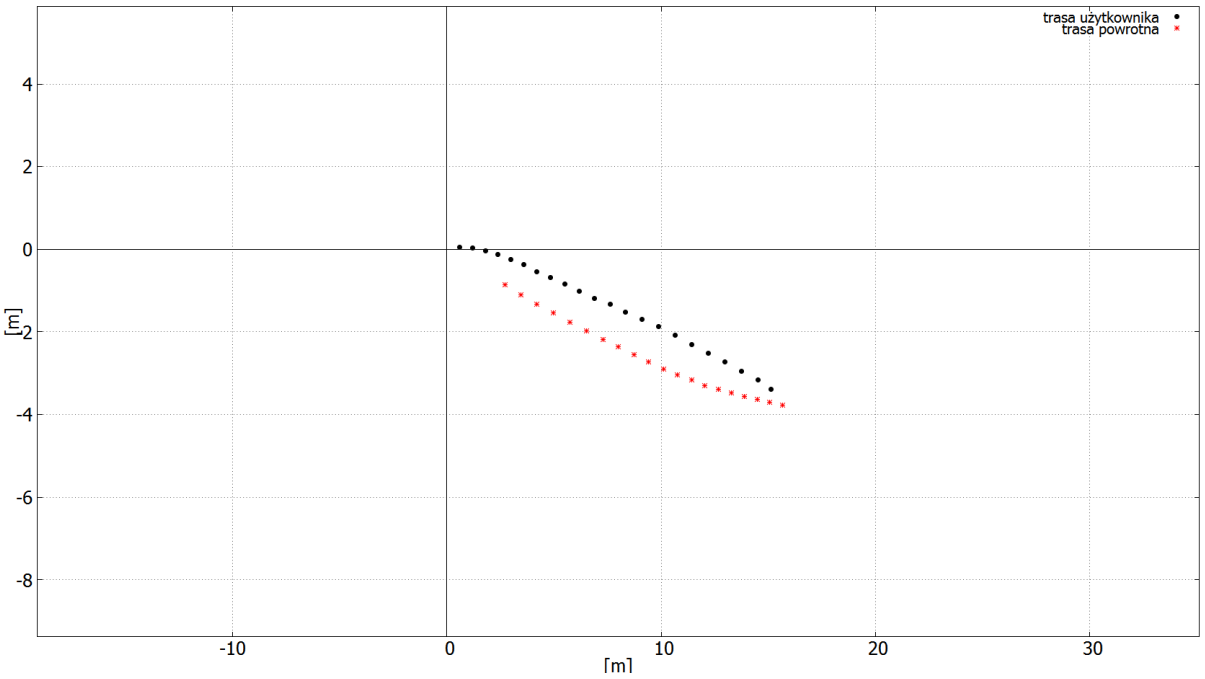
W ramach testu użytkownik miał za zadanie przebycie prostego odcinka, przejście w tryb nawigacji i powrót tą samą trasą bez korzystania z podpowiedzi urządzenia.



Rys. 31. Trasa powrotna użytkownika po przejściu prostego odcinka.

Trasy w obie strony zarejestrowane przez aplikację przedstawia rys. 31. Jak widać drogi przebyte w obie strony mają podobną długość, jednak są w innym kierunku, gdzie pożądanym wynikiem było pokrycie się tras.

Testy powtarzano korygując wagę żyroskopu w wyniku filtru komplementarnego – jej wartość na poziomie 0,4 przyniosła pożądany efekt (rys. 32).

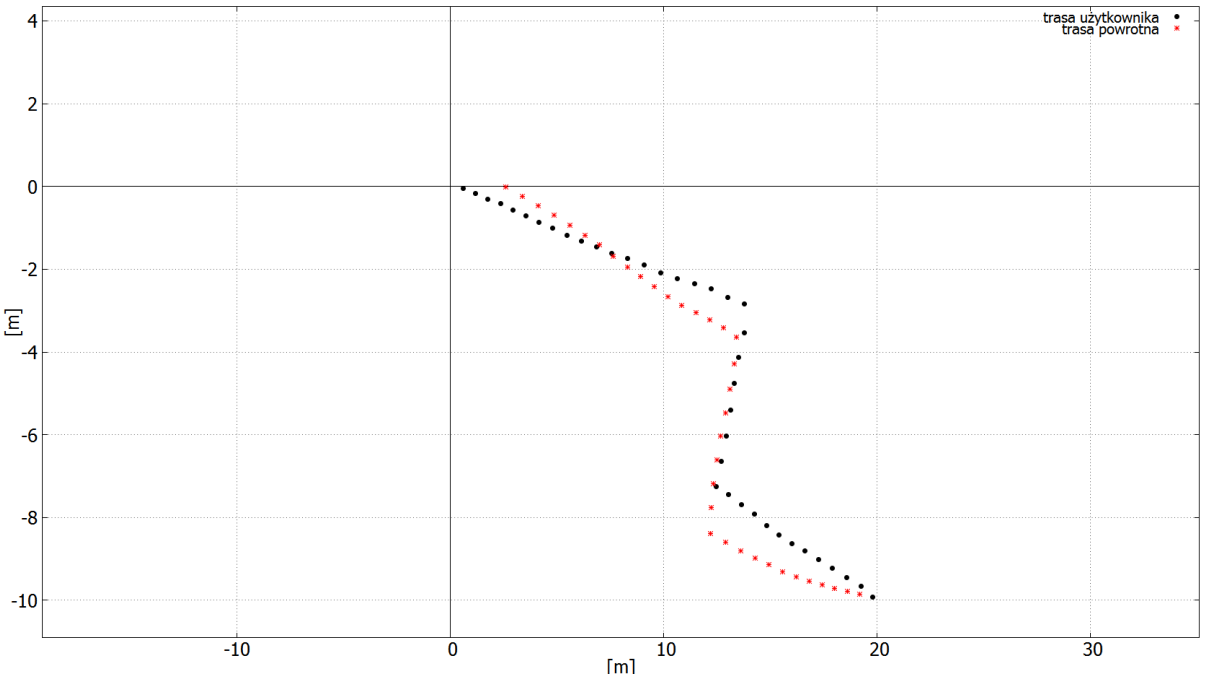


Rys. 32. Trasa powrotna w linii prostej po korekcji wagi żyroskopu.

Niestety ta zmiana przywraca w pewnym stopniu problem przedstawiony na rys. 29, w którym skręt pod kątem prostym powodował błąd estymacji azymutu. Interesujący jest fakt, że manipulacja wspomnianym parametrem nie ma wpływu na wynik testu z łagodnym zakrętem.

Test nawigacji w budynku

Przeprowadzono test w budynku, żeby sprawdzić skuteczność aplikacji w naturalnym środowisku. Użytkownik miał za zadanie poruszać się w drodze powrotnej zgodnie ze wskazaniami aplikacji.



Rys. 33. Trasy zarejestrowane przez program podczas nawigacji w budynku.

Zostały zarejestrowane trasy wykryte przez aplikację (rys. 33). Użytkownik dotarł w okolice celu – według aplikacji odległość od celu była mniejsza niż 2,5 m, w rzeczywistości odległość wynosiła 3,5 m. Użytkownik nie miał wątpliwości co do wyboru korytarza, którym należało wrócić. Aplikacja zarejestrowała dystans w jedną stronę na poziomie 26,58 m przy około 21 metrach przebytych w rzeczywistości (przeszacowanie około 26%). Zwiększony błąd szacowania dystansu bierze się z tego, że użytkownik podczas testu zmieniał prędkość chodu.

Omówienie pozostałych eksperymentów

Sprawdzono, że przeszacowanie długości kroku mieści się w przedziale 10 – 30 % w zależności od tempa chodu, jednak wzrost był nieliniowy. Wynika to z uproszczonej implementacji estymatora długości kroku, dużą rolę w takim przypadku gra także amplituda przebiegów sygnału akcelerometru.

Zbadano przeszacowanie ilości kroków. W zależności od podłoża ilość kroków była zawyżana o 3 – 25%. Dla chodników i ulic – podłóż nierównych – przeszacowania były największe. Przejście po nierównym terenie, w szczególności w ciężkim, zimowym obuwiu, zwiększa ilość drgań pionowych urządzenia, tym samym zwodząc krokomierz. Przewiduje się, że uwzględnienie amplitudy sygnału przy zliczaniu kroków mogłoby zniwelować opisany efekt.

W teście na otwartej przestrzeni stwierdzono, że użytkownik miał problem z poruszaniem się po prostej ścieżce, z racji na zbyt rzadką aktualizację strzałki wskazującej sugerowany kierunek. Aktualizacja dopiero po postawieniu kroku powodowała problemy ze znalezieniem właściwego kierunku, w którym należy się dalej poruszać, w efekcie zmuszając użytkownika do kluczenia pomiędzy właściwymi punktami trasy. Stwierdzono niską użyteczność aplikacji dla nawigacji na przestrzeniach otwartych, w których trasa nie jest ograniczona (ścianą, krawężnikiem) w związku z problemem opisanym powyżej.

7 Wnioski

Jak pokazano w niniejszej pracy możliwe jest wytworzenie aplikacji na urządzenia mobilne z systemem Android przy użyciu metod innych niż całkowanie przyspieszeń. Przedstawione proste implementacje algorytmów wystarczyły, by wspomóc użytkownika w znalezieniu trasy powrotnej. Trudność rozwiązywanego problemu leży umiejętnym wykorzystaniu fuzji sensorów oraz znalezieniu właściwej zależności przebiegu sygnału akcelerometru od długości i ilości kroków.

7.1 Stopień pokrycia celów pracy

Wytworzona aplikacja spełnia założone cele. Pracuje na szybszych i wolniejszych urządzeniach, automatycznie wykrywa zakręty. Nie udało się natomiast zapewnić dostosowania długości kroku, gdyż zastosowany algorytm nie sprawdził się. Ocenia się, że jako prototypowy, program realizuje cel z zadowalającą jakością. Użytkownik podczas testów nie miał problemów z wyborem ścieżki powrotnej i trafieniem do celu, zakładając że telefon był trzymany w odpowiedniej pozycji.

W celu poprawienia jakości pracy aplikacji należałoby zmienić sposób estymacji długości kroku, jak wykazały eksperymenty nie zależy ona jedynie od częstotliwości stawianych kroków. Zmiana sposobu zapamiętywania trasy lub nawigacji mogłaby ułatwić użytkownikowi dotarcie do celu. Dla przykładu, gdyby komponent nawigacji zapamiętywał tylko szczególne punkty na trasie (takie jak zakręty) i interpolował położenia między nimi, podpowiedzi dla użytkownika byłyby bardziej intuicyjne. Niewielkiej poprawki wymaga także algorytm krokomierza.

Aby trafić do użytku komercyjnego program wymagałby rozwinięcia funkcjonalności. Wystarczyłoby skorygować zachowanie komponentu szacującego azymut oraz krokomierza, dzięki czemu program działałby w dowolnym położeniu. Dodatkowo należałoby dodać opcję pracy w tle – wtedy aplikacja zużywałaby mniej energii. Istotnym mogłoby okazać się odejście od zlinearyzowanego sposobu przetwarzania danych, który w obecnej postaci może (ale nie musi) powodować opóźnienia w informowaniu użytkownika o kierunku, w którym powinien się poruszać.

8 Bibliografia

[1] O. J. Woodman, „An introduction to inertial navigation," University of Cambridge, Cambridge, 2007.

[2] Wikipedia, „Inertial navigation system," 2009. [Online]. Adres: http://en.wikipedia.org/wiki/Inertial_navigation_system. [Data uzyskania dostępu: 30 1 2015].

[3] P. Warszawska, „Akcelerometr i żyroskop. Instrukcja do ćwiczenia," Warszawa.

[4] K. Tran, T. Le i T. Dinh, „A high-accuracy step counting algorithm for iPhones using Accelerometer," 2012.

[5] D. Sachs, „Sensor Fusion on Android Devices: A Revolution in Motion Processing," 2010. [Online]. Adres: https://www.youtube.com/watch?v=C7JQ7Rpwn2k.

[6] R. C. Martin, Czysty kod. Podręcznik dobrego programisty, 2010.

[7] P.-J. V. d. Maele, „Reading a IMU Without Kalman: The Complementary Filter," 2013. [Online]. Adres: http://www.pieter-jan.com/node/11.

[8] F. Li, C. Zhao, G. Ding, J. Gong, C. Liu i F. Zhao, „A Reliable and Accurate Indoor Localization Method Using Phone Inertial Sensors," pp. 1-5, 2012.

[9] P. Lawitzki, „ANDROID SENSOR FUSION TUTORIAL," 2012. [Online]. Adres: http://www.thousand-thoughts.com/2012/03/android-sensor-fusion-tutorial/.

[10] K. Kircher, „Android: Gyroscope Fusion," 2014. [Online]. Adres: http://www.kircherelectronics.com/blog/index.php/11-android/sensors/16-android-gyroscope-fusion. [Data uzyskania dostępu: 30 1 2015].

[11] W. Kang, S. Nam, Y. Han i S. Lee, „Improved Heading Estimation for Smartphone-Based Indoor Positioning Systems," pp. 1-3, 2012.

[12] S. Kulesza, „Cyfrowe przetwarzanie sygnałów. Układy z dyskretnym czasem (03)," 2013, pp. 4-5.

[13] „Sensors Overview," [Online]. Adres: http://developer.android.com/guide/topics/sensors/sensors_overview.html. [Data uzyskania dostępu: 30 1 2015].