KAPITAŁ LUDZKI
NARODOWA STRATEGIA SPÓJNOŚCI

UNIA EUROPEJSKA
EUROPEJSKI
FUNDUSZ SPOŁECZNY

Silesian University of Technology
Faculty of Automatic Control, Electronics and Computer Science
Institute of Computer Science

Thesis for the degree of Doctor of Philosophy
in Computer Science

Quaternions based human motion analysis algorithms implemented with data flow processing framework for Motion Data Editor software

Mateusz Janiak

Thesis Advisor:	prof. dr hab. inż. Konrad Wojciechowski
Consultants:	dr inż. Agnieszka Szczęsna dr inż. Janusz Słupik

Gliwice, September 2013

Streszczenie

1 Wprowadzenie

Nowe technologie pozwalają rejestrować coraz dokładniej dane o otaczającym nas świecie. To bezpośrednio przekłada się na zwiększone wymagania dla mocy obliczeniowych do przetwarzania dużych zbiorów danych oraz dla przestrzeni dyskowych niezbędnych do ich przechowywania. Proponuje się specjalizowane rozwiązania dla zbiorów danych o różnych wielkościach. Dla dużych zbiorów - tak zwanych *big data* [69] - stosuje się hurtownie danych i dedykowane, rozproszone bazy danych. Dla średnich i małych zbiorów proponowane są systemy algebry komputerowej (Computer Algebra Systems (CAS) [73]), arkusze kalkulacyjne oraz specjalizowane, naukowe języki programowania [21].

Obecnie analiza danych wykonywana jest nie tylko przez wykwalifikowaną kadrę w dziedzinie analizy matematycznej z doświadczeniem z zakresu programowania, ale również przez mniej doświadczonych użytkowników w tych dziedzinach, którzy bazują na gotowych rozwiązaniach i opracowaniach interesujących ich zagadnień statystyki.

Na rynku jest niewiele aplikacji dedykowanych ogólnemu przetwarzaniu danych, które wspierałyby użytkowników w często powtarzanych operacjach na danych: ładowanie, konwersja, proces przetwarzania czy raportowanie rezultatów analiz. Taki stan rzeczy spowalnia i utrudnia postęp w prowadzonych badaniach, ograniczając przy tym przepływ wiedzy pomiędzy członkami grup badawczych. Pociąga to za sobą potrzebę stworzenia nowych narzędzi, łatwych w użyciu i umożliwiających automatyczne, optymalne wykorzystanie dostępnych mocy obliczeniowych. Ich celem jest pomóc użytkownikom skupić się na celach, które chcą osiągnąć, a nie na technologiach i środkach, jakimi można te cele osiągnąć.

Przeglądając dostępne aplikacje wspierające analizę ruchu można zauwa-

żyć znaczące braki w ich funkcjonalności w zakresie przetwarzania danych. Pozwalają one jedynie na przeglądanie zgromadzonych danych i proste raportowanie. Uderza to w sporą grupę potencjalnych użytkowników, obejmującą lekarzy, trenerów sportowych i naukowców. Brak kompletnych rozwiązań dla analizy i przetwarzania ruchu ogranicza rozwój następujących dziedzin:

- medycyna (ortopedzi, neurochirurdzy i neurolodzy),
- sport (nowe miary postępu w treningach, spersonalizowane treningi, porównywanie techniki zawodników),
- bezpieczeństwo (rozpoznawanie osób po ich ruchach, wykrywanie potencjalnie niebezpiecznych sytuacji),
- rozrywka (realistyczne animacje syntezowanego ruchu, rozszerzona rzeczywistość).

Aby udostępnić użytkownikom kompletne narzędzia dla analizy danych w PJWSTK stworzono aplikację MDE. Jest ona zorientowana na wsparcie ogólnego przetwarzaniu danych, niezależne od ich formatu, typu i charakteru. MDE wprowadza dobrze zdefiniowane standardy dla efektywnego przetwarzania i analizy danych.

Bazując na wielorozdzielczych narzędziach zaproponowano nowe podejście dla analizy i przetwarzania danych ruchu w reprezentacji kwaternionowej [52] aby wesprzeć badania nad ruchem ludzkiego ciała. Metoda ta jest rozwinięciem obecnych rozwiązań [26, 27, 37, 38, 39, 10, 18, 7, 6, 40, 41, 5], operujących głównie na klasycznej analizie falkowej rotacji zapisanych jako niezależne kąty Eulera.

Celem pracy jest zaprezentowanie architektury i logiki aplikacji MDE - wyjaśnienie w jaki sposób standaryzują one procedurę analizy i przetwarzania danych przy pomocy wbudowanych funkcjonalności. Aby zweryfikować zalety MDE oraz możliwość łatwego użycia tego rozwiązania dla własnych potrzeb, spróbowano zaimplementować i przetestować zbiór nowych narzędzi dla analizy ruchu, opartych na wielorozdzielczej analizie danych ruchu w zapisie kwaternionowym.

2 Tezy pracy

W pracy sformułowano dwie tezy:

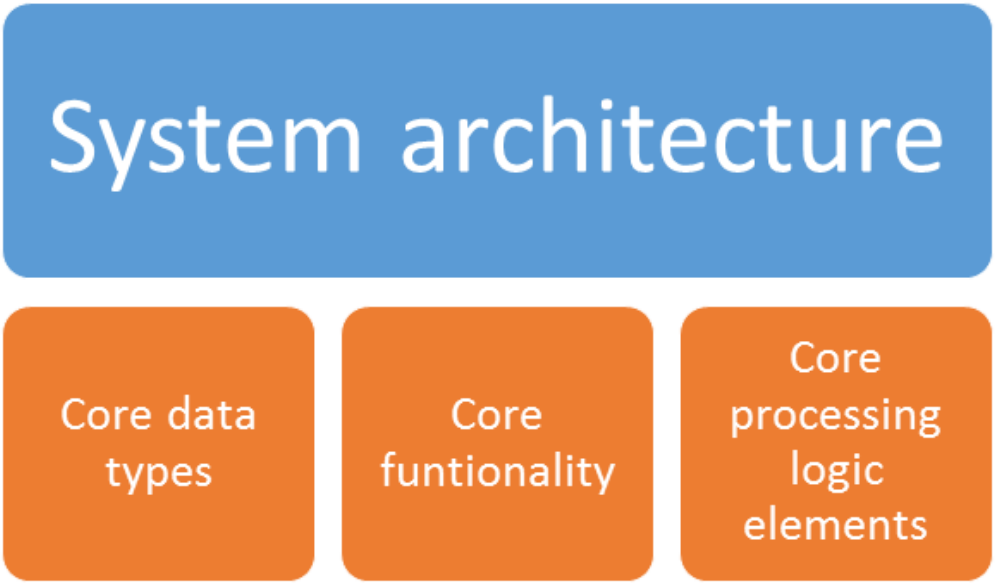
Teza 1 Aplikacja MDE jest dojrzałym, stabilnym narzędziem, opartym na ogólnym szkielecie (framework) dla przetwarzania danych. MDE może być elastycznie rozszerzany i konfigurowany na potrzeby specyficznych wymagań użytkowników poprzez dedykowany mechanizm wtyczek (plugins). Narzędzie to wspiera pełną, dobrze zdefiniowaną procedurę ładowania danych. Gwarantuje wydajne zarządzanie danymi w zunifikowany sposób, niezależnie od ich typu, oferując mechanizmy automatycznego i optymalnego wykorzystania dostępnych zasobów obliczeniowych. Przeglądanie danych jest znormalizowane dla wszystkich obsługiwanych typów danych, przedstawiając je z różnych perspektyw, specyficznych dla danego typu.

Teza 2 Zaproponowane narzędzia dla analizy ruchu, oparte o wielorozdzielczą analizę danych ruchu w reprezentacji kwaternionowej, oferują bardzo dobre właściwości dekompozycji danych dla zastosowań odfiltrowywania szumów oraz kompresji. Algorytmy te, zrealizowane w formie bibliotek ogólnego przeznaczenia, można w bardzo prosty sposób zaimplementować w ramach aplikacji MDE. Niezbędne dane do analizy oraz otrzymane wyniki eksperymentów są automatycznie obsługiwane przez specjalizowane moduły aplikacji, umożliwiając łatwe definiowanie i realizację badań wraz z analizą wyników. Wykorzystanie wcześniej przygotowanych bibliotek wymaga niewielkiego nakładu pracy programistycznej, dostosowując istniejące rozwiązania do akceptowalnych przez MDE interfejsów. Taka implementacja automatycznie gwarantuje optymalne wykorzystanie wszystkich zasobów obliczeniowych bez samodzielnego zarządzania wątkami.

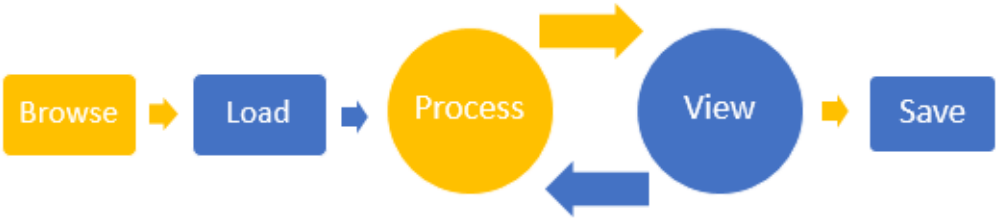
3 Charakterystyka aplikacji Motion Data Editor

Architektura MDE oparta jest na trzech podstawowych komponentach (Rysunek 6.1). Podstawowe typy danych obejmują:

- zunifikowany mechanizm przechowywania i zarządzania danymi w pamięci, niezależnie od ich typów dla języka programowania C++,
- generyczny typ dla danych o charakterze czasowym,
- typy danych narzędziowych ogólnego przeznaczenia.

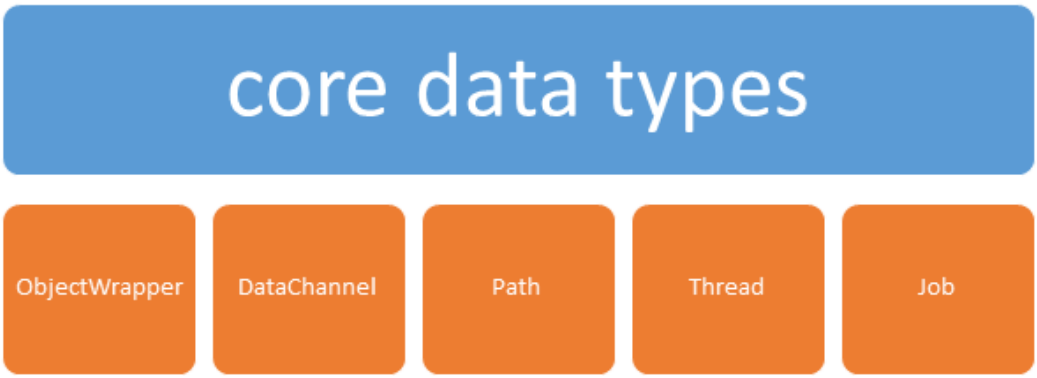


Rysunek 6.1: Ogólna architektura aplikacji



Rysunek 6.2: Ogólna procedura przetwarzania i analizy danych

MDE dostarcza dedykowany moduł pozwalający optymalnie wykorzystać wszystkie dostępne zasoby obliczeniowe na potrzeby realizacji różnych obliczeń. Dodatkowo, możliwe jest zunifikowane logowanie wiadomości na temat aktualnego stanu aplikacji. Dostarczone rozwiązania wspierają tworzenie i projektowanie specjalizowanych widoków dla realizowanej logiki przetwarzania. Aplikacja oferuje mechanizm wtyczek, pozwalający rozszerzać jej funkcjonalność o nowe możliwości. Wbudowane mechanizmy zarządzające elementami logiki przetwarzania danych wspierają proces analizy danych (Rysunek 6.2). Podstawowe elementy logiki przetwarzania danych obejmują obiekty dostarczające i ładujące dane do aplikacji (źródła i parsery), wizualizujące dane (wizualizatory) oraz szeroko pojęte nowe funkcjonalności aplikacji w formie serwisów.



Rysunek 6.3: Podstawowe typy danych

3.1 Podstawowe typy danych

Dwoma najważniejszymi typami danych w MDE są OW oraz *DataChannel*. Przedstawiona architektura aplikacji jest w całości oparta na tych obiektach oraz ich właściwościach. Wspierają one uniwersalny mechanizm zarządzania dowolnymi typami danych oraz standaryzują obsługę danych o charakterze czasowym. Rysunek 6.3 przedstawia wszystkie bazowe typy danych MDE.

3.1.1 ObjectWrapper

OW wprowadza nowe podejście dla ogólnego zarządzania danymi dowolnych typów dla języka programowania C++. Koncepcja oparta jest na wspólnym typie danych dla wszystkich obiektów, znanym z takich języków programowania jak *C#* oraz *Java*, gdzie bazą dla wszystkich danych jest ogólny typ *Object*. OW oparty jest na programowaniu generycznym. Wprowadza jednolity interfejs dla zapytań o informację o typach dla opakowanych danych, wypakowywanie danych oraz ich inicjalizację. Dodatkowo OW wspiera mechanizm leniwej inicjalizacji danych oraz system meta-danych w formie pary literałów $[klucz, wartosc]$. Mechanizm OW jest konfigurowalny w ramach dwóch polityk:

- wskaźnika - typ wskaźnika używanego do przechowywania danego typu danych,
- klonowanie - sposób tworzenia kopii danych.

Opcje te pozwalają dostosować OW do obsługi dowolnych typów danych, od wbudowanych podstawowych typów dla języka C++, po własne typy danych, zdefiniowane na potrzeby konkretnych zastosowań. OW wspiera informację o hierarchii opakowanych typów. Ta funkcjonalność działa tylko

dla typów pochodnych, dla których opis hierarchii typów bazowych jest dostępny. Obecna implementacja tej funkcjonalności pozwala obsługiwać jedynie liniowe hierarchie typów, więc dla dziedziczenia wielobazowego tylko jeden typ bazowy może być użyty. Mechanizm ten wykorzystywany jest do filtrowania danych według typu oraz przedstawiania danych z różnych perspektyw, wynikających z ich hierarchii dziedziczenia.

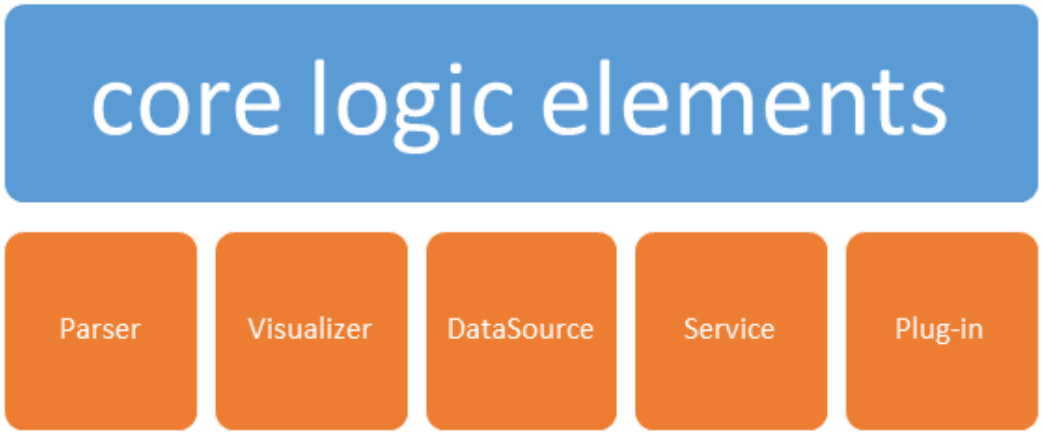
3.1.2 Dane o charakterze czasowym

Generyczny typ *DataChannel* wprowadzono aby zunifikować obsługę danych o charakterze czasowym. Pozwala on na przechowywanie próbek w formie $[indeks(czas) \rightarrow wartosc]$, gdzie indeksy są unikalne i mają ściśle zdefiniowaną relację mniejszości. Indeksy są ponadto niemodyfikowalne, ponieważ porządkują dane w domenie czasu. Ładowanie danych do obiektów typu *DataChannel* musi odbywać się w porządku rosnącym dla czasu, ponieważ wewnętrznie dane są dodatkowo opatrzone indeksami reprezentowanymi przez liczby całkowite, odpowiadające numerom próbek w kolejności w jakiej zostały załadowane. *DataChannel* oparty jest na łączeniu interfejsów oraz dedykowanych im implementacji (mix-ins). Pozwala to zmieniać charakter danych bez potrzeby ich kopiowania, poprzez przykrywanie instancji danych dedykowanym interfejsem, zmieniającym typ przechowywanych w *DataChannel* wartości. Obiekty typu *DataChannel* można podzielić na dwie kategorie ze względu na wartości znaczników czasu:

- równo-odległe indeksy czasu - stała różnica pomiędzy czasami kolejnych próbek danych,
- nieregularne indeksy czasu - zmienna różnica pomiędzy czasami kolejnych próbek danych.

Ta własność używana jest do optymalnego dostępu do wartości dla zapytań poprzez indeks czasowy danych.

Jako rozszerzenie dla obiektów *DataChannel* zaprojektowano kilka pomocniczych typów, pozwalających traktować dane dyskretne w sposób ciągły. Oferują one gotowe metody interpolacji oraz możliwość dostarczania własnych interpolatorów. Klasy te są używane kiedy *DataChannel* przeglądany jest w większej rozdzielczości niż wynika to z surowych danych. Struktura wielu algorytmów wymaga zapytań o indeksy próbek spoza danego zakresu. Aby wesprzeć tego rodzaju operacje zaprojektowano kilka schematów ekstrapolacji dla *DataChannel*:



Rysunek 6.4: Bazowe elementy logiki przetwarzania danych

- wyjątki - przy zapytaniu o dane spoza dostępnego zakresu rzucane są wyjątki,
- wartości brzegowe - odpowiednio najmniejsza lub największa próbka są powielane dla wszystkich zapytań spoza zakresu,
- symulacja periodyczności danych - zapytanie o dane spoza zakresu jest zaokrąglane do właściwego przedziału danych na podstawie odległości pomiędzy brzegowymi próbkami.

Obiekty z danymi czasowymi często są rozszerzane o stan opisujący ich aktualny czasu. Dla *DataChannel* wprowadzono typ *Timer*. Łączy on stan czasu z *DataChannel* pozwalając pobierać dane dla aktualnie ustawionego w *Timer* czasu. Dodatkowo umożliwia on prowadzenie niezależnego stanu czasu dla tych samych danych bez potrzeby ich kopiowania.

3.2 Bazowe elementy logiki przetwarzania danych

Rysunek 6.4 przedstawia podstawowe elementy logiki przetwarzania danych. Odpowiedzialne są one za ładowanie nowych danych do aplikacji, przeglądanie danych oraz rozszerzanie MDE o nowe funkcjonalności poprzez dedykowany mechanizm wtyczek.

3.2.1 Parsery

Koncepcja parserów powstała aby znormalizować proces wypakowywania danych z różnych źródeł (najczęściej plików bądź strumieni). Każdy parser dostarcza informacji o obsługiwanych źródłach oraz typach danych, które

potencjalnie może z nich dostarczyć. Ponadto, każdy parser musi charakteryzować się minimum jedną z dwóch funkcjonalności obsługujących różne sposoby dostępu do danych:

- indywidualne operacje wejścia wyjścia - parser sam wykonuje niskopoziomowe operacje dostępu do danych,
- obsługa strumieni danych - parser dostarcza danych z ściśle zdefiniowanych strumieni danych.

Taka realizacja parserów pozwala na optymalizację ładowania danych dla plików. Plik może być wczytany raz do pamięci i dostarczony w formie strumienia do parsera. Możliwe jest również obsłużenie pliku przez kilka parserów, wzajemnie się uzupełniających pod kątem wypakowywanych danych.

Mechanizm parserów współpracuje z mechanizmem leniwej inicjalizacji OW, gdzie faktyczne parsowanie przeprowadzane jest w momencie odpytywania OW o dane. Pozwala to ograniczyć potrzebną dla danych pamięć, gdyż nie wszystkie dane muszą być używane podczas analizy i przetwarzania.

3.2.2 Źródła danych

Idea źródeł danych (*DataSource*) została zaproponowana, aby ujednolicić sposób przeglądania dostępnych danych na potrzeby analizy i przetwarzania. Obiekty tego typu odpowiedzialne są za wskazywanie i dostarczanie danych w ich kontenerach (ścieżka do lokalnego pliku, ściągnięcie archiwum z serwera FTP, odpytanie bazy danych, otwarcie połączenia z danym urządzeniem) oraz ładowanie danych z kontenerów do aplikacji (najczęściej z pomocą parserów i dedykowanych menadżerów). Źródła danych pozwalają MDE obsługiwać specyficzne sposoby dostarczania danych, których przykładem może być HMDB [19] - scentralizowana usługa dostarczająca danych ruchu nagranych w HML.

3.2.3 Wizualizatory

Koncepcja wizualizatorów (*Visualizer*) wprowadza warstwę abstrakcji dla przeglądania danych. Obiekty tego typu obsługują mechanizm serii danych, pozwalających przeglądać rożne typy danych. Zaproponowane rozszerzenie dla serii danych umożliwia prezentację danych o charakterze czasowym, dla których wprowadzono dodatkowe operacje: skalowanie w czasie (scale), przesunięcie w czasie (offset) oraz modyfikację aktualnego czasu dla danych w serii. Wizualizatory mogą wspierać różne ilości serii danych, zależnie od

charakteru prezentowanych danych i właściwości wizualizatora. Wśród serii danych można wyróżnić aktywną serię danych, aktualnie zarządzaną przez użytkownika, dla której wizualizator może dostarczać dodatkowych funkcjonalności, specyficznych dla danego typu. Każdy wizualizator jest opisany typami danych, które może obsłużyć, co ułatwia użytkownikom przeglądanie danych za pomocą różnych wizualizatorów, prezentujących inne perspektywy danych.

3.2.4 Serwisy

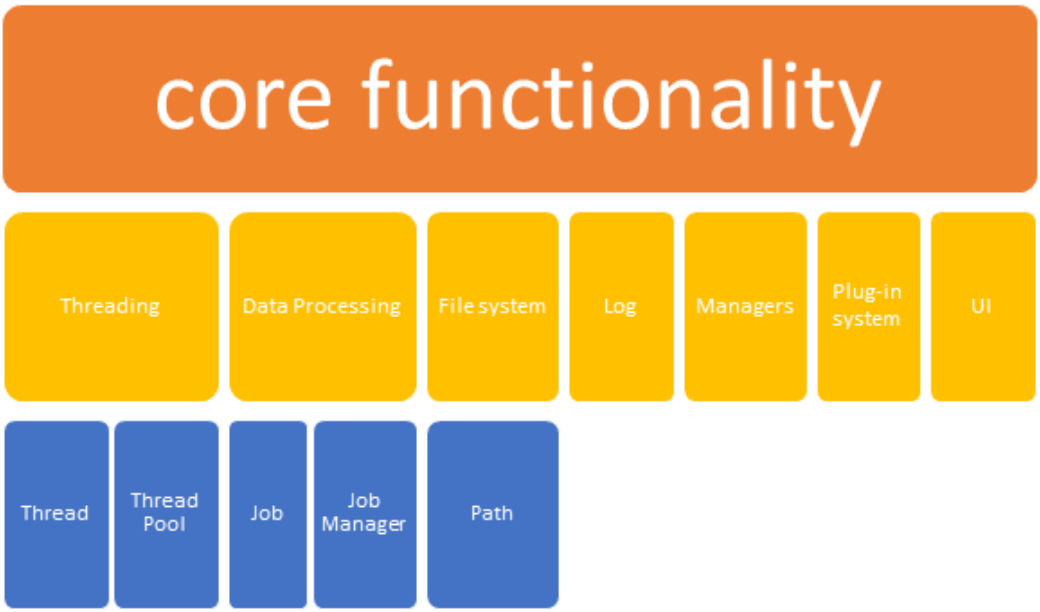
Serwisy wprowadzono jako uogólnienie nowych, szeroko pojętych funkcjonalności aplikacji. Trudno jest zaproponować zbiór wspólnych operacji dla serwisów, ponieważ mogą one prezentować całkowicie odmienne rozszerzenia aplikacji. Z tego też powodu serwisy są najbardziej uprzywilejowanymi elementami logiki MDE, mającymi dostęp do wszystkich modułów logiki aplikacji, aby umożliwić tworzenie różnorodnych, nowych funkcjonalności aplikacji.

3.2.5 Wtyczki

Każda analiza danych operuje na innych typach danych oraz narzędziach w ramach ściśle zdefiniowanej, powtarzalnej procedury. Aby ułatwić użytkownikom dostosowywanie uniwersalnego mechanizmu wspierającego przetwarzanie danych, opartego na logice zaimplementowanej w MDE, wprowadzono system wtyczek, pozwalających rozszerzać możliwości aplikacji. Wtyczki pozwalają rejestrować w aplikacji nowe typy danych, parsery, wizualizatory, źródła danych i serwisy. Ponieważ wtyczki reprezentowane są jako niezależne, dynamiczne biblioteki, ładowane podczas startu aplikacji, należało wprowadzić dedykowane rozwiązania, weryfikujące ich kompatybilność z MDE. Brane są tutaj pod uwagę wersje bibliotek zależnych, używanych w aplikacji i rozwiązaniach dostarczanych z wtyczkami, wersja publicznego interfejsy aplikacji oraz wersja interfejsu użyta do budowy wtyczki. Dodatkowo sprawdzany jest typ kompilacji wtyczki i aplikacji (debug lub release).

Timeline Jedną z najistotniejszych wtyczek dla MDE jest *Timeline*. Serwis ten pozwala na synchronizację danych o charakterze czasowym w ramach ściśle zdefiniowanej hierarchii. *Timeline* umożliwia edycję właściwości czasu dla danych, bez konieczności ich kopiowania. Wszystkie operacje na czasie są automatycznie propagowane na całą hierarchię. Nowymi operacjami *Timeline* są:

- podziel - tworzone są dwa niezależne zestawy danych dla zadanego punktu podziału,



Rysunek 6.5: Wbudowane funkcjonalności

- scal - dwa niezależne zestawy danych są połączone w zadanej kolejności.

Operacje te pozwalają ograniczyć zakres analizowanych danych w domenie czasu do zadanego okna czasu. *Timeline* dostarcza również mechanizmu do odtwarzania danych o charakterze czasowym w wizualizatorach.

3.3 Wbudowane funkcjonalności

Rysunek 6.5 przedstawia wbudowane funkcjonalności MDE, tworzące warstwę abstrakcji dla operacji specyficznych dla systemów operacyjnych, systemów plików oraz tworzenia GUI.

3.3.1 Obsługa systemu plików

Zaproponowano zunifikowany typ do obsługi ścieżek dla systemu plików - *Path*, ponieważ większość danych dostarczana jest w formie plików. Jest on bazą dla zbioru podstawowych operacji na plikach i folderach. MDE dostarcza w ten sposób informacji o specyficznych zasobach aplikacji.

3.3.2 Log

Aby umożliwić prezentację istotnych komunikatów użytkownikowi, zaprojektowano mechanizm hierarchicznych logów. Oparty jest on na kilkupozio-

mowym statusie informacji, gdzie użytkownik może zdefiniować minimalny poziom, o którym ma być powiadamiany. Pozostałe informacje są automatycznie filtrowane i usuwane. Log ma konfigurowalne ujścia dla informacji, od prostej konsoli, przez sformatowane pliki tekstowe, po graficzne okno. Każda wtyczka inicjowana jest dedykowanym poziomem loga, pozwalającym zidentyfikować źródło i kontekst wiadomości.

3.3.3 Wielowątkowość

Optymalne wykorzystanie zasobów obliczeniowych komputera wymaga poprawnego użycia wątków do równoległej realizacji zadań. Ponieważ zarządzanie wątkami jest operacją specyficzną dla systemów operacyjnych, w MDE wprowadzono typ *Thread*, tworzący warstwę abstrakcji niezależną od użytkowanej platformy. Aby kontrolować ilość tworzonych wątków dla celów diagnostycznych oraz zapewnienia stabilności i wydajności aplikacji, wprowadzono typ *ThreadPool*. Obiekt ten pozwala na tworzenie ściśle zdefiniowanej ilości wątków. Kiedy wartość ta zostanie osiągnięta, nie można utworzyć nowych wątków do momentu ukończenia obliczeń przez już stworzone wątki i zwolnienia ich zasobów. Aby zminimalizować narzut związany z tworzeniem nowych wątków, *ThreadPool* utrzymuje zdefiniowaną, minimalną ilość wolnych wątków jako bufor dla najbliższych zapytań o nowe wątki, starając się ponownie wykorzystać wątki, które zakończyły swoje zadania.

3.3.4 Przetwarzanie danych

Ciągłe tworzenie nowych wątków oraz inicjowanie ich działania może drastycznie pogorszyć wydajność aplikacji. Aby temu zapobiec zaprojektowano mechanizm zarządzający i kolejkujący zadania do wykonania. Ściśle zdefiniowana grupa wątków odpowiada za przetwarzanie zleconych zadań. *Job* i *JobManager* pozwalają efektywnie wykorzystać dostępne zasoby obliczeniowe dzięki utrzymywaniu optymalnej ilości wątków przetwarzających, która dla współczesnych procesorów wynosi $rdzenie\_procesora * 2 - 1$. Jeden z wątków pozostaje wolny na potrzeby obsługiwania graficznego interfejsu użytkownika. Zadania pobierane są ze wspólnej kolejki przez wyznaczone wątki, w kolejności w jakiej zostały dodane.

3.3.5 Menadżery

W centrum logiki aplikacji leżą dedykowane menadżery dla podstawowych elementów logiki przetwarzania i analizy danych. Część z nich została już omówiona (wątki i zadania). Funkcjonalność menadżerów została zdekomponowana na operacje niemodyfikujące i modyfikujące dany obiekt. Takie

podejście pozwala udostępniać globalnie niemodyfikującą część operacji oraz dostarczać lokalnie funkcjonalności pozwalające zmieniać stan menadżerów do ściśle zdefiniowanych elementów architektury, gdzie takie zachowanie zostało przewidziane. Taka dekompozycja wprowadza porządek w logice, określając dokładnie obszary odpowiedzialności poszczególnych modułów oraz ich potencjalne możliwości, eliminując niepotrzebne udostępnianie wszystkich funkcjonalności.

Operacje na menadżerach są synchronizowane. Aby zwiększyć wydajność grupy operacji na danym menadżerze oraz wprowadzić izolację dla tych operacji zaprojektowano mechanizm transakcji. Szeregują on inne operacje na danym obiekcie do momentu zakończenia aktualnej transakcji. Efekty transakcji mogą być zatwierdzone - stan obiektu zostaje trwale zmodyfikowany, lub mogą zostać wycofane, gdzie wszystkie zmiany na danym obiekcie są anulowane, a jego stan sprzed transakcji jest przywrócony.

3.4 Proces Ciągłej Integracji (CI)

MDE oparte jest na wielu zewnętrznych bibliotekach, co pozwoliło zaoszczędzić czas na implementowanie dobrze znanych i przetestowanych rozwiązań. Niestety, ilość zależności stopniowo utrudniała rozwój aplikacji, gdyż kompilacja potrzebnych bibliotek zajmowała coraz więcej czasu i była miejscem wielu błędów ludzkich. Procedurę CI wprowadzono, aby w pierwszej kolejności zautomatyzować budowę zewnętrznych bibliotek. Potem rozszerzono ją o budowę własnych projektów, testy i kontrolę jakości kodu. Dla poprawnie zweryfikowanych artefaktów CI generuje instalatory gotowych produktów, udostępniając je użytkownikom do instalacji i aktualizacji. Dodatkowo tworzona jest też dokumentacja techniczna wewnętrznych bibliotek.

3.5 Rozwój aplikacji

Aby zapewnić wysoką jakość kodu dla MDE wprowadzono zbiór dobrze zdefiniowanych i zalecanych praktyk programistycznych:

- programowanie generyczne,
- wzorce projektowe,
- dekompozycja i minimalna zależność między modułami,
- minimalne komentarze nagłówków,

- samo-komentujący się kod implementacji wraz z ściśle zdefiniowanym stylem kodowania,
- dostarczanie prostego, minimalnego, ale kompletnego API dla użytkowników aplikacji.

Zastosowanie tych metod pozwoliło utrzymać elastyczność MDE na nowe rozwiązania, ograniczyć możliwości potencjalnych błędów oraz skrócić czas na poprawę błędów.

4 Potokowe przetwarzanie danych

Celem zapewnienia użytkownikom konfigurowalnego, łatwego w użyciu narzędzia do projektowania i realizacji dowolnych schematów przetwarzania danych opracowano dedykowaną wtyczkę. Wprowadza ona do MDE serwis pozwalający tworzyć dobrze zdefiniowane potoki przepływu danych. Ich modele oparte są na strukturze grafu. W ramach modelu można wyróżnić trzy typy węzłów ze względu na ich funkcję:

- źródła - dostarczają nowych danych do potoku,
- procesory - odpowiadają za przetwarzanie danych,
- terminatory - utrwalają wyniki przetwarzania.

W przeciwieństwie do grafów, węzły w potoku nie łączą cię bezpośrednio ze sobą. Wprowadzono nowy element - tak zwany *pin* - poprzez który węzły mogą być ze sobą łączone. Każdy węzeł opisany jest stałą konfiguracją pinów, którą można porównać do sygnatury funkcji w językach programowania. W ten sposób rozumiane piny opisują listę parametrów wejściowych funkcji oraz jej zwracane wartości. Aby dokładniej odzwierciedlić definiowane węzłów jako funkcje wprowadzono dodatkowe rozszerzenia do opisu modelu na poziomie pinów, gdzie można scharakteryzować piny dodatkowymi właściwościami:

- wymagany - pin wejściowy musi być połączony z innym pinem, aby zapewnić minimalną funkcjonalność węzła (argument wymagany),
- zależny - na danym pinie można spodziewać się rezultatów wyłącznie jeśli jego piny zależne są podłączone.

Dostosowanie opracowanych już algorytmów przetwarzania danych na potrzeby potoków polega na opakowaniu ich interfejsami węzłów (najczęściej

procesorów). Logika przetwarzania gwarantuje, że przetwarzanie w potoku jest automatycznie zrównoleglone, poprzez delegowanie obliczeń w formie zadań do *JobManagera*. Dzięki temu użytkownik otrzymuje narzędzie standaryzujące proces przetwarzania i automatycznie wykorzystujące całą dostępną moc obliczeniową na potrzeby własnych obliczeń, bez potrzeby indywidualnego tworzenia i zarządzania wątkami oraz ich synchronizacji.

Aby uprościć wykorzystanie potokowego przetwarzania danych użytkownikom mniej biegłym w programowaniu, zaprojektowano graficzne środowisko programowania. Moduł ten umożliwia wizualną realizację modeli przetwarzania poprzez łączenie prostych bloków funkcyjnych, realizujących poszczególne zadania. Użytkownik informowany jest na bieżąco o możliwościach łączenia pinów, wynikających z kompatybilności typów danych które reprezentują i reguł łączenia modelu. Wizualne środowisko pozwala również na grupowanie połączonych i skonfigurowanych już węzłów, tworząc bardziej rozbudowane funkcjonalności, które można użyć w późniejszym czasie bez potrzeby ponownego ich tworzenia. Takie złożone węzły mogą być wymieniane pomiędzy użytkownikami.

5 Wielorozdzielcza analiza danych ruchu w zapisie kwaternionowym

Zaproponowano nowe podejście do analizy danych ruchu na bazie falek drugiej generacji - schemat liftingu [62] i kwaternionowego zapisu rotacji dla stawów. Schemat liftingu pozwala na dekompozycje danych do wielopoziomowej reprezentacji, przedstawionej jako sumę reprezentacji szczegółowych i zgrubnych. Operacja ta oparta jest na trzech krokach, powtarzanych rekursywnie:

blok podziału dzieli dane na dwa podzbiory dla próbek indeksowanych wartościami parzystymi i nieparzystymi,

blok predykcji estymuje próbki indeksowane wartościami nieparzystymi przy pomocy próbek indeksowanych wartościami parzystymi. Estymowana próbka zastępowana jest różnicą pomiędzy jej wartością a proponowaną estymatą.

blok uaktualnienia aktualizuje wartości próbek parzystych, aby ich średnia odpowiadała wejściowej średniej sygnału.

W porównaniu do kątów Eulera, kwaterniony opisują rotacje w sposób kompaktowy, oferując przy tym lepsze metody interpolacji [11] z punktu widzenia realizacji rotacji:

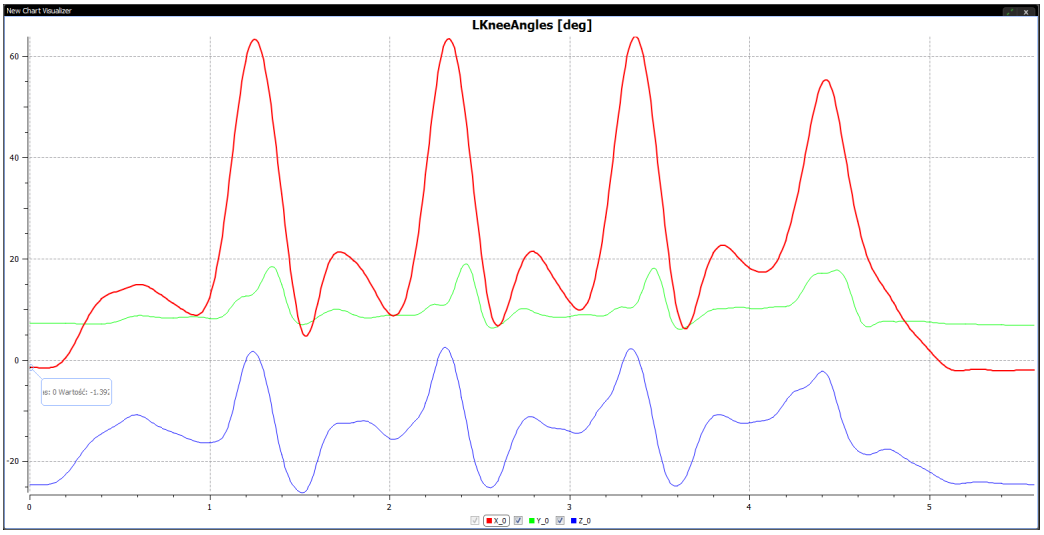
- slerp,
- squad.

Na bazie tych metod i własności kwaternionów zaproponowano szereg schematów liftingu. Głównym problemem przy tworzeniu tych algorytmów wydaje się być miara odległości oraz interpretacja wartości średniej dla kwaternionów, które nie są jednoznacznie określone [45]. Schemat liftingu wymaga jednak tylko zachowania wartości średniej dla danych na każdym poziomie rozdzielczości, dlatego można przyjmować dowolne interpretacje dla średniej, aby najlepiej pasowały do poszczególnych zastosowań. Zaproponowane schematy operują również na kwaternionach rozumianych jako wektory przestrzeni $\mathbb{R}4$, aby zweryfikować ich poprawność i możliwość zastosowania dla analizy ruchu. Ponadto przedstawiona metoda interpolacji kwaternionów w przestrzeni stycznej ($\mathbb{R}3$) wskazuje kierunek dla nowych metod interpolacji dla kwaternionów na bazie większej ilości próbek dla osiągnięcia gładkiej reprezentacji ruchu.

Aby zweryfikować zaproponowane narzędzia wielorozdzielczej analizy ruchu wprowadzono pojęcie odległości pomiędzy dwoma seriami danych w reprezentacji kwaternionowej. Odległość ta oparta jest na skumulowanej wartości kątów obrotu dla różnicy (iloraz) pomiędzy kolejnymi parami kwaternionów w obu seriach danych. Wartość ta przedstawiona jest w radianach i jest tym mniejsza im kwaterniony przedstawiają bardziej zbliżone obroty, zmierzając do 0 dla identycznych danych.

5.0.1 Testy

Wykonano szereg testów sprawdzających poprawność wszystkich transformat - jakość detali oraz rekonstrukcja sygnału po dekompozycji. Rysunek 6.6 przedstawia dane testowe. Dodatkowo, przetestowano zastosowanie rezultatów dekompozycji dla redukcji szumów oraz kompresji danych ruchu. Rysunek 6.7 przedstawia detale danych dla schematu liftingu opartego na interpolacji metodą *squad*. Po zastosowaniu proponowanej stratnej metody kompresji na przedstawionych detalach osiągnięto poziom kompresji równy 87.5% przy średnim zniekształceniu sygnału po dekompresji na poziomie 0.002 radiana na kąt obrotu kwaternionu (Rysunek 6.10). Równie dobrze wypadają testy odfiltrowywania szumów. Dla zniekształconych danych po



Rysunek 6.6: Dane testowe - lewe kolano 26-letniego, zdrowego mężczyzny

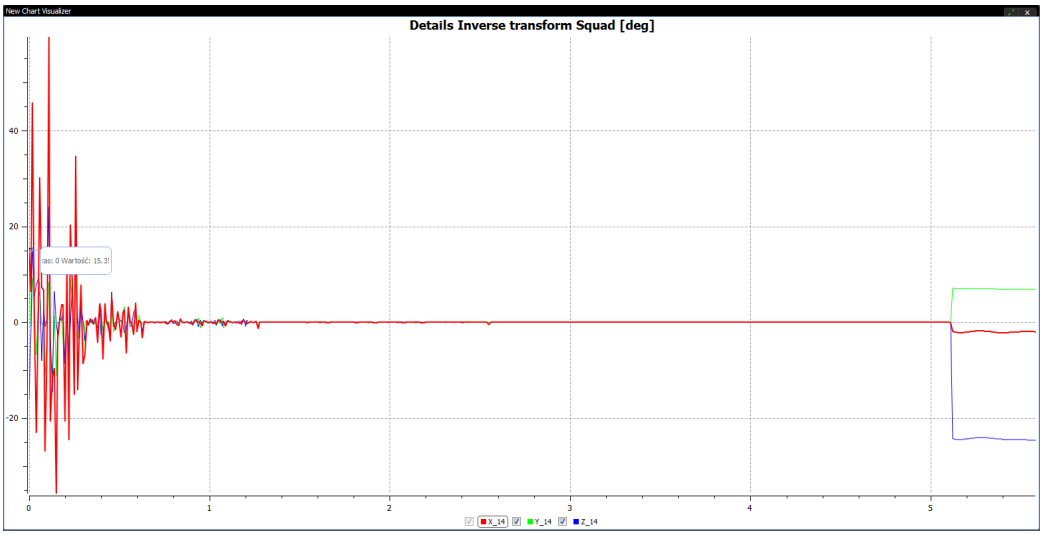
dodaniu białego szumu opisanego funkcją Gaussa dla $\sigma = 5°$ (Rysunek 6.8) udało się zrekonstruować dane ze zredukowanymi zakłóceniami o blisko 50% (Rysunek 6.9). Przedstawione wyniki obejmują cały zakres sygnału, chociaż schemat liftingu operował tylko na części początkowych próbek, będącej największą możliwą potęgą liczby 2, wynikającą z binarnego podziału w bloku split. Dla porównania przedstawiano również niezmodyfikowaną część danych.

5.0.2 Implementacja

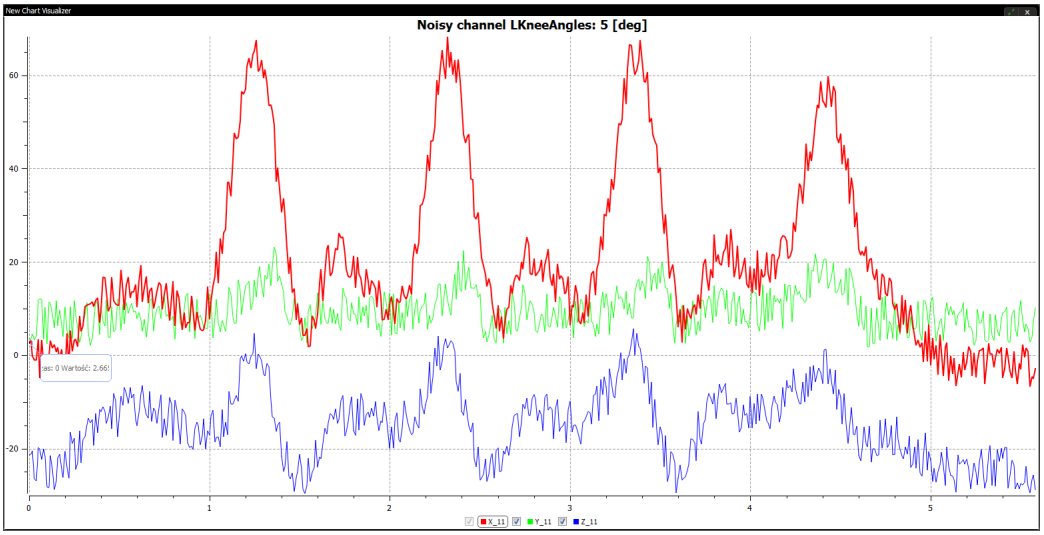
Wszystkie zaproponowane schematy liftingu zostały zaimplementowane w formie algorytmów ogólnego użytku. Zostały one opakowane interfejsami dla węzłów przetwarzających potokowego przetwarzania danych w MDE. Dane do analizy pobrano z dostępnej bazy danych ruchu (HMDB) poprzez wbudowane źródło danych. Eksperymenty i wyniki powstały w ramach możliwości oferowanych przez MDE. Dedykowany wizualizator pozwolił przedstawić dane w formie kątów Eulera, które są łatwiejsze do analizy wizualnej od kwaternionów.

6 Oryginalne wyniki

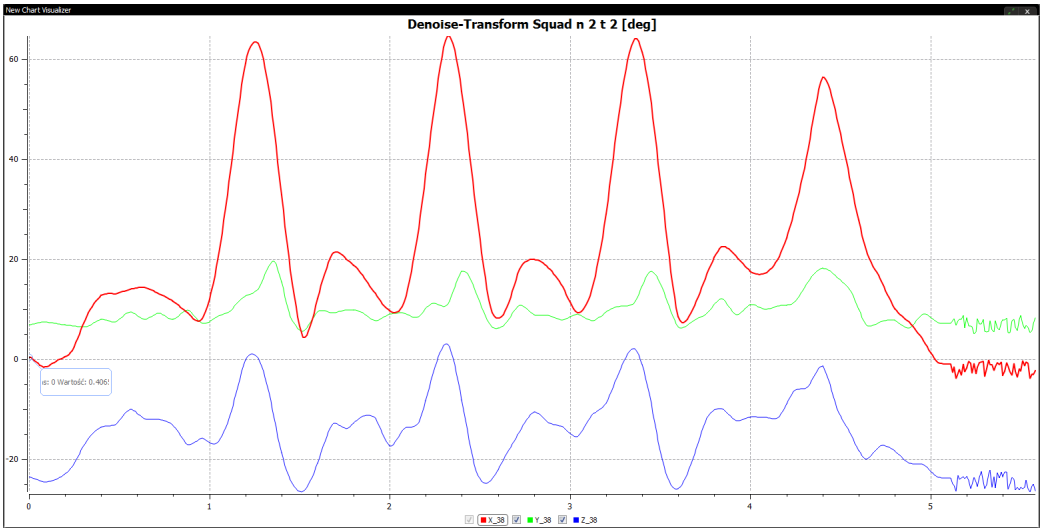
Jako główne, autorskie elementy przedstawione w pracy należy wymienić następujące zagadnienia:



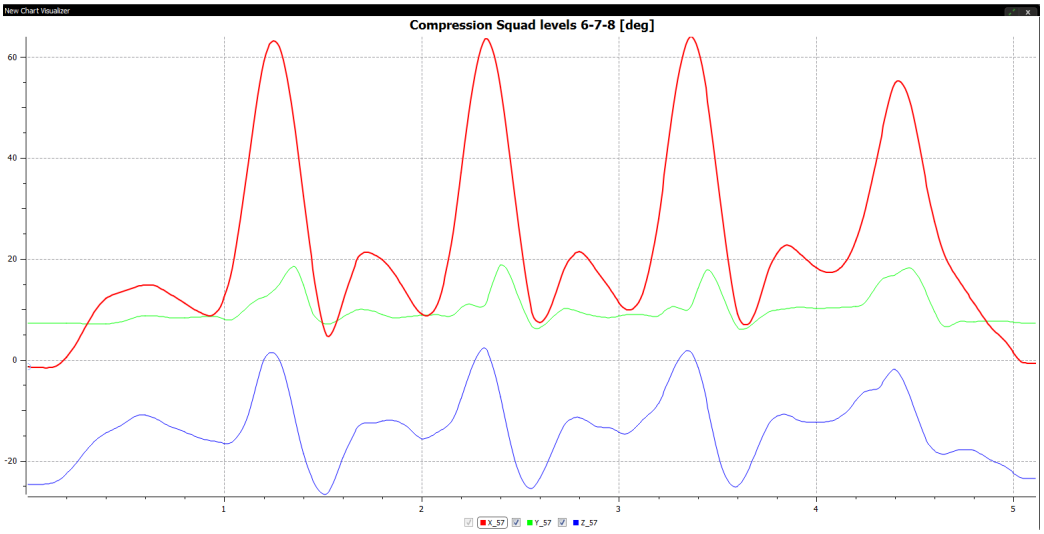
Rysunek 6.7: Detale wszystkich rozdzielczości dla interpolacji metodą *squad*



Rysunek 6.8: Zniekształcone dane przez dodanie białego szumu Gaussa z $\sigma = 5°$



Rysunek 6.9: Odfiltrowany szum schematem liftingu opartym na *squad* dla wartości progowej kątów obrotu = 2° i zmodyfikowanych rozdzielczościach: 6, 7, 8



Rysunek 6.10: Wyniki dekompresji dla schematu liftingu opartego na *squad* po usunięciu wszystkich detali dla rozdzielczości: 6, 7, 8 (87,5% wszystkich danych)

- dla rozwoju aplikacji MDE:
  - zaprojektowanie architektury i logiki dla MDE, które unifikują proces analizy i przetwarzania danych,
  - wprowadzenie koncepcji *DataChannel* z pomocniczymi funkcjonalnościami, które standaryzują obsługę danych o charakterze czasowym,
  - zaproponowanie procedury leniwej inicjalizacji poprzez parsery oraz ich dualnej natury dla obsługi źródeł danych, które pozwalają przyspieszyć ładowanie danych do aplikacji,
  - opracowanie obiektu typu *ThreadPool* w ramach MDE, pozwalającego kontrolować ilość wątków aplikacji,
  - wprowadzenie koncepcji przetwarzania danych opartej o *Job* i *JobManager* umożliwiającej automatycznie wykorzystywać wszystkie dostępne zasoby obliczeniowe,
  - zaprojektowanie *Timeline* jako warstwy abstrakcji dla operacji na czasie,
  - opracowanie procedury weryfikacji kompatybilności ładowanych wtyczek z aplikacją oraz ich inicjalizacji kontekstem aplikacji,
  - wprowadzenie koncepcji hierarchicznego modelu logowania wiadomości,
  - zaprojektowanie elastycznego i prostego w użyciu modułu potokowego przetwarzania danych wraz z graficznym środowiskiem programowania,
  - wprowadzenie procesu Ciągłej Integracji, wspierającego rozwijanie własnych projektów poprzez automatyzacje wielu zadań,
  - opracowanie dedykowanych skryptów na potrzeby konfiguracji nowych projektów, wspierających proces wyszukiwania i używania bibliotek zewnętrznych oraz zarządzania zależnościami pomiędzy projektami
- dla badań nad analizą ruchu:
  - opracowanie nowego podejścia do analizy danych ruchu na bazie schematu liftingu oraz interpolacji kwaternionów metodą *squad*,
  - opracowanie testów i ich wyników dla porównania różnych schematów liftingu na kwaternionach orz ich zastosowania dla kompresji i filtrowania szumu,

– przykładowa implementacja własnych narzędzi w ramach potokowego przetwarzania danych dla MDE.

7 Podsumowanie

W pracy przedstawiono w jaki sposób aplikacja MDE stworzona w PJWSTK wspiera ogólne przetwarzanie i analizę danych. Opisano standardy dla analizy ruchu, jakie wprowadza MDE poprzez zaproponowaną architekturę i logikę. Omówiono mechanizmy wspierające wydajne przetwarzanie danych z automatycznym wsparciem dla wielowątkowego przetwarzania oraz nowym mechanizmem jednolitego przechowywania i zarządzania danymi dowolnego typu dla C++. Rozwiązania te dowodzą doskonałych właściwości MDE jako platformy dla różnego typu projektów badawczych, gdzie wydajne i proste przetwarzanie danych stanowią kluczowym element badań. Ponadto pokazano jak prosto można rozszerzać możliwości aplikacji o własne rozwiązania poprzez dedykowany mechanizm wtyczek. W tym celu zaprezentowano możliwość prostego dostosowania zewnętrznych bibliotek na potrzeby potokowego przetwarzania danych w ramach MDE (wcześniej stworzone biblioteki ogólnego użytku z przedstawionymi schematami liftingu zostały opakowane niezbędnymi interfejsami dla potokowego przetwarzania danych). Wszystkie przedstawione eksperymenty i ich rezultaty zostały przeprowadzone w graficznym środowisku programowania, ułatwiającym tworzenie schematów przetwarzania danych. Implementując gotowe rozwiązania udało się dodatkowo zdekomponować je na prostsze operacje, kompatybilne z innymi modułami przetwarzającymi. Ponadto uzyskano optymalne wykorzystanie zasobów obliczeniowych bez dodatkowych nakładów pracy, co pozwoliło znacząco skrócić czas potrzebny na dostosowanie się do nowego mechanizmu przetwarzania danych, jak i czas potrzebny na przeprowadzenie niezbędnych testów.

Postawione w pracy tezy zostały udowodnione poprzez szczegółowy opis architektury oraz pozytywne wyniki przeprowadzonych testów dla proponowanych narzędzi analizy ruchu. Udało się również osiągnąć wszystkie postawione w pracy cele implementując istniejące wcześniej rozwiązania w ramach MDE i przeprowadzając testy za pomocą potokowego przetwarzania danych. Przedstawione rezultaty pracy mogą być użyte jako baza dla wielu istniejących algorytmów przetwarzania danych ruchu. Pokazano szereg zalet i zastosowań dla danych w reprezentacji wielorozdzielczej z użyciem proponowanych schematów liftingu. Ponadto, przedstawiono możliwości zastosowań MDE na potrzeby prac badawczych, jako narzędzia wielofunkcyjnego, wspierającego ładowanie, przetwarzanie i analizę dowolnego rodzaju danych.

Wskazano również nowe kierunki dla rozwoju MDE w niedalekiej przyszłości, które jeszcze bardziej uproszczą proces analizy i przetwarzania danych oraz umożliwią swobodną wymianę wiedzy pomiędzy użytkownikami aplikacji.